Learning Semantic Program Embeddings with Graph Interval Neural Network

YU WANG*, Nanjing University, China
KE WANG*, Visa Research, USA
FENGJUAN GAO, Nanjing University, China
LINZHANG WANG, Nanjing University, China

Learning distributed representations of source code has been a challenging task for machine learning models. Earlier works treated programs as text so that natural language methods can be readily applied. Unfortunately, such approaches do not capitalize on the rich structural information possessed by source code. Of late, Graph Neural Network (GNN) was proposed to learn embeddings of programs from their graph representations. Due to the homogeneous (*i.e.* do not take advantage of the program-specific graph characteristics) and expensive (*i.e.* require heavy information exchange among nodes in the graph) message-passing procedure, GNN can suffer from precision issues, especially when dealing with programs rendered into large graphs. In this paper, we present a new graph neural architecture, called Graph Interval Neural Network (GINN), to tackle the weaknesses of the existing GNN. Unlike the standard GNN, GINN generalizes from a curated graph representation obtained through an abstraction method designed to aid models to learn. In particular, GINN focuses exclusively on intervals (generally manifested in looping construct) for mining the feature representation of a program, furthermore, GINN operates on a hierarchy of intervals for scaling the learning to large graphs.

We evaluate GINN for two popular downstream applications: variable misuse prediction and method name prediction. Results show in both cases GINN outperforms the state-of-the-art models by a comfortable margin. We have also created a neural bug detector based on GINN to catch null pointer deference bugs in Java code. While learning from the same 9,000 methods extracted from 64 projects, GINN-based bug detector significantly outperforms GNN-based bug detector on 13 unseen test projects. Next, we deploy our trained GINN-based bug detector and Facebook Infer, arguably the state-of-the-art static analysis tool, to scan the codebase of 20 highly starred projects on GitHub. Through our manual inspection, we confirm 38 bugs out of 102 warnings raised by GINN-based bug detector compared to 34 bugs out of 129 warnings for Facebook Infer. We have reported 38 bugs GINN caught to developers, among which 11 have been fixed and 12 have been confirmed (fix pending). GINN has shown to be a general, powerful deep neural network for learning precise, semantic program embeddings.

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Computing methodologies** → **Neural networks**.

*Both authors contributed equally to this work.

Authors' addresses: Yu Wang, State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing, Jiangsu, 210023, China, yuwang_cs@smail.nju.edu.cn; Ke Wang, Security, Cryptography, and Blockchain, Visa Research, Palo Alto, CA, USA, kewang@visa.com; Fengjuan Gao, State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing, Jiangsu, 210023, China, fjgao@smail.nju.edu.cn; Linzhang Wang, State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing, Jiangsu, 210023, China, lzwang@nju.edu.cn.

2475-1421/2020/11-ART137
https://doi.org/10.1145/3428205

Additional Key Words and Phrases: Program embeddings, Control-flow graphs, Intervals, Graph neural networks, Null pointer dereference detection

ACM Reference Format:
Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. 2020. Learning Semantic Program Embeddings with Graph Interval Neural Network. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 137 (November 2020), 27 pages. https://doi.org/10.1145/3428205

1 INTRODUCTION

Learning distributed representations of source code has attracted growing interest and attention in programming language research over the past several years. Inspired by the seminal work word2vec [Mikolov et al. 2013], many methods have been proposed to produce vectorial representation of programs. Such vectors, commonly known as program embeddings, capture the semantics of a program through their numerical components such that programs denoting similar semantics will be located in close proximity to one another in the vector space. A benefit of learning program embedding is to enable the application of neural technology to a board range of programming language (PL) tasks, which were exclusively approached by logic-based, symbolic methods before.

Existing Models of Source Code. Earlier works predominately considered source code as text, and mechanically transferred natural language methods to discover their shallow, textual patterns [Gupta et al. 2017; Hindle et al. 2012; Pu et al. 2016]. Following approaches aim to learn semantic program embeddings from Abstract Syntax Tree (AST) [Alon et al. 2019a,b; Maddison and Tarlow 2014]. Despite the significant progress, tree-based models are still confined to learning syntactic program features due to the limited offering from AST. Recently, another neural architecture, called Graph Neural Network (GNN), has been introduced to the programming domain [Allamanis et al. 2018; Li et al. 2016]. The idea is to reduce the problem of learning embeddings from source code to learning from their graph representations. Powered by a general, effective message-passing procedure [Gilmer et al. 2017], GNN has achieved state-of-the-art results in a variety of problem domains: variable misuse prediction [Allamanis et al. 2018], program summarization [Fernandes et al. 2019], and bug detection and fixing [Dinella et al. 2020]. Even though GNN has substantially improved the prior works, their precision and efficiency issues remain to be dealt with. First, GNN is indiscriminate in which types of graph data they deal with. In other words, once programs are converted into graphs, they will be processed in the same manner as any other graph data (*e.g.* social networks, molecular structures). As a result, it misses out on the opportunity to capitalize on the unique graph characteristics possessed by programs. Second, perhaps more severely, GNN has difficulties in learning from large graphs due to the high cost of its underlying message-passing procedure. Ideally, every node should pass messages directly or indirectly to every other node in the graph to allow sufficient information exchange. However, such an expensive propagation is hard to scale to large graphs without incurring a significant precision loss.

Graph Interval Neural Network. In this paper, we present a novel, general neural architecture called Graph Interval Neural Network (GINN) for learning semantic embeddings of source code. The design of GINN is based on a key insight that by learning from abstractions of programs, models can focus on code constructs of greater importance, and ultimately capture the essence of program semantics in a precise manner. At the technical level, we adopt GNN's learning framework for their cutting-edge performance. An important challenge we need to overcome is how to abstract programs into more efficient graph representations for GINN to learn. To this end, we develop a principled abstraction method directly applied on control flow graphs. In particular, we derive from the control flow graph a hierarchy of intervals — subgraphs that generally represent looping constructs — as a new form of program graphs. Under this abstracted graph representation, GINN

focuses exclusively on intervals for extracting deep, semantic program features; hence scales its generalization to large graphs efficiently. Note that GINN's scalability-enhancing approach is fundamentally different from the existing techniques [Allamanis et al. 2018], where extra edges are required to link nodes that are far apart in a graph. Moreover, Allamanis et al. [2018] have not quantified the improvement of model scalability due to the added edges, let alone identify which edges to add *w.r.t.* the different graphs and downstream tasks.

We have realized GINN as a new model of source code and extensively evaluated it. To demonstrate its generality, we evaluate GINN in multiple downstream tasks. First, we pick two prediction problems defined by the prior works [Allamanis et al. 2018; Alon et al. 2019b] in which GINN significantly outperforms RNN Sandwich and sequence GNN, the state-of-the-art models in predicting variable misuses and method names respectively. Second, we evaluate GINN in detecting null pointer dereference bugs in Java code, a task that is barely attempted by learning-based approaches. Results show GINN-based bug detector is significantly more effective than the baseline built upon GNN. We also deploy our trained bug detector and Facebook Infer [Berdine et al. 2006; Calcagno et al. 2015], arguably the state-of-the-art static analysis tool, to search bugs in 20 Java projects that are among the most starred on GitHub. After manually inspecting the alarms raised by both tools, we find that our bug detector yields a higher precision (38 bugs out of 102 alarms) than Infer (34 bugs out of 129 alarms). We have reported the 38 bugs detected by GINN to developers, out of which 11 are fixed and 12 are confirmed (fix pending). Our evaluation suggests GINN is a general, powerful deep neural network for learning precise, semantic program embeddings.

Contributions. Our main contributions are:

- We propose a novel, general deep neural architecture, Graph Interval Neural Network, which generalizes from a curated graph representation, obtained through a graph abstraction method on the control-flow graph.
- We realize GINN as a new deep model of code, which readily serves multiple downstream PL tasks. More importantly, GINN outperforms the state-of-the-art model by a comfortable margin in each downstream task.
- We present the details of our extensive evaluation of GINN in variable misuse prediction, method name prediction, and null pointer dereference detection.
- We publish our code and data at https://github.com/GINN-Imp/GINN to aid future work.

2 PRELIMINARY

In this section, we briefly revisit connected, directed graphs, interval [Allen 1970] and GNN [Gori et al. 2005], which lay the foundation for our work.

2.1 Graph

A graph $G = (V, E)$ consists of a set of nodes $V = \{v_1, ..., v_n\}$, and a list of directed edge sets $E = (E_1, ..., E_K)$ where K is the total number of edge types and E_k $(1 \leq k \leq K)$ is a set of edges of type k. We denote by $(v_s, v_d, k) \in E_k$ an edge of type k directed from node v_s to node v_d. For graphs with only one edge type, an edge is represented as (v_s, v_d).

The immediate successors of a node v_i, denoted by $post(v_i)$, are all of the nodes v_j for which (v_i, v_j) is an edge in one edge set in E. The immediate predecessors of node v_j, denoted by $pre(v_j)$, are all of the nodes v_i for which (v_i, v_j) is an edge in one edge set in E.

A path is an ordered sequence of nodes $(v_p, ..., v_q)$ and their connecting edges, in which each node v_t, $t \in (p, \ldots, q-1)$, is an immediate predecessor of v_{t+1}. A closed path is a path in which the first and last nodes are the same. The successors of a node v_t, denoted by $post^*(v_t)$, are all of

Algorithm 1: Finding intervals for a given graph

Input: a set of nodes V on a graph
Output: a set of intervals $\mathcal{S}$

```
1  // Assume v0 is the unique entry node of the graph
2  H = {v0};
3  while H ≠ ∅ do
4      // remove next h from H
5      h = H.pop();
6      I(h) = {h};
7      // only nodes that are neither in the current interval nor any other interval will be considered
8      while {v ∈ V | v ∉ I(h) ∧ ∄s(s ∈ S ∧ v ∈ s) ∧ pre(v) ⊆ I(h)} ≠ ∅ do
9          I(h) = I(h) ∪ { v };
10     // find next headers
11     while {v ∈ V | ∄s1(s1 ∈ S ∧ v ∈ s1) ∧ ∃m1, m2(m1, m2 ∈ pre(v) ∧ m1 ∈ I(h) ∧ m2 ∉ I(h))} ≠ ∅ do
12         H = H ∪ { v };
13     S = S ∪ I(h);
```

the nodes v_x for which there exists a path from v_t to v_x. The predecessors of a node v_t, denoted by $pre^*(v_t)$, are all of the nodes v_y for which there exists a path from v_y to v_t.

2.2 Interval

Introduced by Allen [1970], an interval *I(h)* is the maximal, single entry subgraph in which h is the only entry node and all closed paths contain h. The unique interval node h is called the interval head or simply the header node. An interval can be expressed in terms of the nodes in it: $I(h) = \{v_1, v_2, ..., v_m\}$.

By selecting the proper set of header nodes, a graph can be partitioned into a set of disjoint intervals. We show the partition procedure proposed by Allen [1970] in Algorithm 1. The key is to add to an interval a node only if all of its immediate predecessors are already in the interval (Line 8 to 9). The intuition is such nodes when added to an interval keep the original header node as the single entry of an interval. To find a header node to form another interval, a node is picked that is not a member of any existing intervals although it must have a (but not all) immediate predecessor being a member of the interval that is just computed (Line 11 to 12). We repeat the process until reaching the fixed-point where all nodes are members of an interval.

The intervals on the original graph are called the first order intervals, denoted by $I^1(h)$, and the graph from which they were derived is called first order graph G^1. Partitioning the first order graph results in a set of first order intervals, denoted by $\mathcal{S}^1$ *s.t.* $I^1(h) \in \mathcal{S}^1$. By making each first order interval into a node and each interval exit edge into an edge, the second order graph can be derived, from which the second order intervals can also be defined. The procedure can be repeated to derive successively higher order graphs until the n-th order graph consists of a single interval. Figure 1 illustrates such a sequence of derived graphs.

2.3 Graph Neural Network

Graph Neural Network (GNN) [Gori et al. 2005] is a specialized machine learning model designed to learn from graph data. The intuitive idea underlying GNN is that nodes in a graph represent objects and edges represent their relationships. Thus, each node v can be attached to a vector, called state, which collects a representation of the object denoted by v. Naturally, the state of v can be

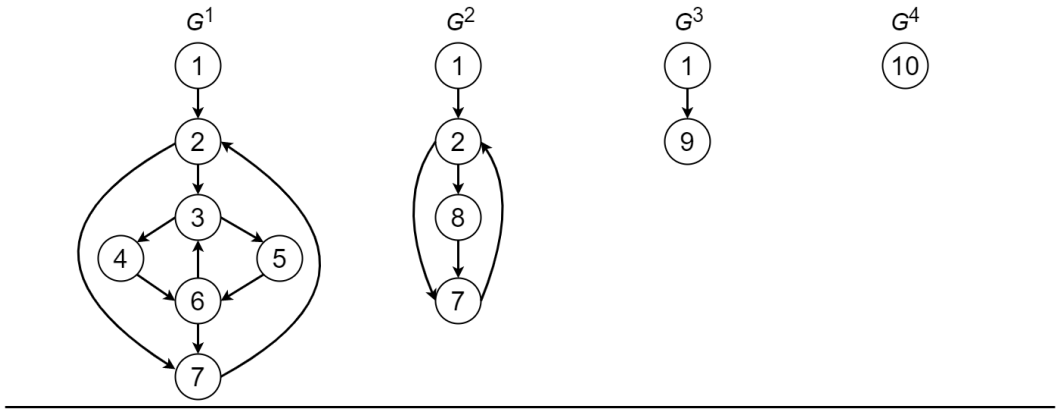

Fig. 1. n-th order intervals and graphs. The set of intervals on $\mathcal{S}^1$ are $I^1(1)$={1}, $I^1(2)$={2}, $I^1(3)$={3,4,5,6}, $I^1(7)$={7}. The set of intervals on $\mathcal{S}^2$ are $I^2(1)$={1} and $I^2(2)$={2,7,8}. $I^3(1)$={1,9} and $I^4(10)$={10} are the only intervals on $\mathcal{S}^3$ and $\mathcal{S}^4$ respectively.

specified using the information provided by nodes in the neighborhood of v. Technically, there are many realizations of this idea. Here, we describe the message-passing technique [Gilmer et al. 2017], a widely applied method for GNN to compute the states of nodes.

We reuse the terminology of graph introduced in Section 2.1. Given a graph $G=(V,E)$ where V and E are the inputs, a neural message-passing GNN computes the state vector for each node in a sequence of steps. In each step, every node first sends messages to all of its neighbors, and then update its state vector by aggregating the information received from its neighbors.

$$\mu_v^{(l+1)} = \phi(\{\mu_u^{(l)}\}_{u\in\mathcal{N}(v)}) \tag{1}$$

$\mu_v^{(l+1)}$ denotes the state of v in $l+1$ steps, which is determined by the state of its neighbors in the previous step. $\mathcal{N}(v)$ denotes the neighbors that are connected to v. Formally, $\mathcal{N}(v) = \{u|(u,v,k) \in E_k, \forall k \in \{1,2,...,K\}\}$. $\phi(\cdot)$ is a non-linear function that performs the aggregation. After the state of each node is computed with many rounds of message passing, a graph representation can also be obtained through various pooling operations (*e.g.* average or sum).

Some GNN [Si et al. 2018] computes a separate node state *w.r.t.* an edge type (*i.e.* $\mu_v^{(l+1),k}$ in Equation 2) before aggregating them into a final state (*i.e.* $\mu_v^{(l+1)}$ in Equation 3).

$$\mu_v^{(l+1),k} = \phi_1(\sum_{u\in\mathcal{N}^k(v)} \mathrm{W}_1\mu_u^{(l)}), \forall k \in \{1,2,...,K\} \tag{2}$$

$$\mu_v^{(l+1)} = \phi_2(\mathrm{W}_2[\mu_v^{(l),1}, \mu_v^{(l),2}, ..., \mu_v^{(l),K}]) \tag{3}$$

W_1 and W_2 are variables to be learned, and ϕ_1 and ϕ_2 are some nonlinear activation functions.

Li et al. [2016] proposed Gated Graph Neural Network (GGNN) as a new variant of GNN. Their major contribution is a new instantiation of $\phi(\cdot)$ using Gated Recurrent Units [Cho et al. 2014]. The following equations describe how GGNN works:

$$m_v^l = \sum_{u\in\mathcal{N}(v)} f(\mu_u^{(l)}) \tag{4}$$

$$\mu_v^{(l+1)} = GRU(m_v^l, \mu_v^{(l)}) \tag{5}$$

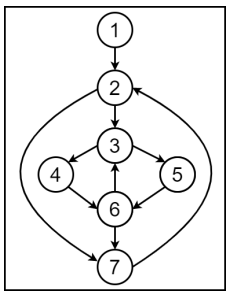

Fig. 2

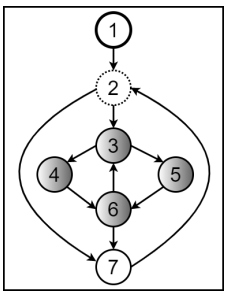

Fig. 3

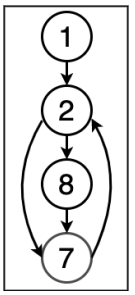

Fig. 4

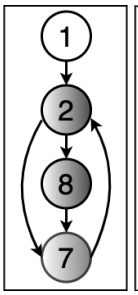

Fig. 5

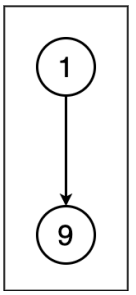

Fig. 6

To update the state of node v, Equation 4 computes a message m_v^l using $f(\cdot)$ (*e.g.* a linear function) from the states of its neighboring nodes $\mathcal{N}(v)$. Next, a *GRU* takes m_v^l and $\mu_v^{(l)}$ — the current state of node v — to compute the new state $\mu_v^{(l+1)}$ (Equation 5).

GNN's Weakness. Although the message-passing technique has highly empowered GNN, it severely hinders GNN's capability of learning patterns formed by nodes that are far apart in a graph. From a communication standpoint, a message sent out by the start node needs to go through many intermediate nodes en route to the end node. Due to the aggregation operation (regardless of which implementation in Equation 1, 3, or 5 is adopted), the message gets diluted whenever it is absorbed by an intermediate node for updating its own state. By the time the message reaches the end node, it is too imprecise to bear any reasonable resemblance to the start node. This problem is particularly challenging in the programming domain where long-distance dependencies are common, important properties models have to capture. Allamanis et al. [2018] designed additional edges to connect nodes that exhibit a relationship of interest (*e.g.* data or control dependency) but are otherwise far away from each other in an AST. The drawbacks are: first, their idea, albeit conceptually appealing, has not been rigorously tested in practice, therefore, the extent to which it addresses the scalability issue is unknown; second, they have not offered a principled solution regarding which edges to add in order to achieve the best scalability. Instead, they universally added a predefined set of edges to any graph, an approach that is far from satisfactory.

3 METHODOLOGY AND APPROACH

Our intuition is that generalization can be made easier when models are given abstractions (of programs) to learn. Considering its generality and capacity, we choose GNN's learning framework to incorporate our abstraction methodology. A key challenge arises: how to abstract programs into favorable graph representations for GNN to learn. A natural idea would be abstracting programs at the level of source code and then reducing the abstracted programs into graphs. On the other hand, a simpler and perhaps more elegant approach is to abstract programs directly on their graph representation, which this paper explores. Specifically, we propose a new graph model, called Graph Interval Neural Network (GINN), which uses control flow graph to represent an input program,[1] and abstracts it with three primitive operators: partitioning, heightening, and lowering. Each primitive operator is associated with an extra operator for computing the state of each node on the resultant graph after the primitive operator is applied. Using the control flow graph depicted in Figure 2 as our running example, we explain in details of GINN's learning approach.

To start with, GINN uses the partitioning operator to split the graph into a set of intervals. Figure 3 adopts different styling of the nodes to indicate the four intervals on the graph. Then, it proceeds to

[1]We discuss a few variations in Section 4.

compute the node states using the extra operator. Unlike the standard message-passing mechanism underpinning the existing GNN, GINN restricts messages to be passed within an interval. If an interval is a node itself, its state will not get updated by the extra operator. For example, node 3, 4, 5, and 6 in Figure 3 will freely pass messages to the neighbors while evolving their own states. On the other hand, node 1 and 2, which are alone in their respective intervals, keep their states as they were. The propagation that occurs within an interval is carried out in the same manner as it is in the existing GNN (Equation 1, 3, or 5 depending on the actual implementation). We formalize the partitioning operator as follows:

Definition 3.1. (*Partitioning*) Partitioning operator, denoted by ρ, is a sequence (ρ_{abs}, ρ_{com}) where $\rho_{abs} : G^n \rightarrow \mathcal{S}^n$ is a function that maps an n-th order graph into a set of n-th order intervals, and $\rho_{com} : \forall I^n(h) \in \mathcal{S}^n \rightarrow \mu_{v_1}, \mu_{v_2}, \ldots \mu_{v_T} \in \mathbb{R}^e(v_1, v_2, \ldots, v_T \in I^n(h))$ is a function that computes a e-dimensional vector for each node in each n-th order interval. Equation 6 defines ρ_{com}.

Depends on how many nodes $I^n(h)$ is composed of, two cases to be considered:

$$\begin{cases} \mu_v^{(l+1)} = \phi(\{\mu_u^{(l)}\}_{u \in \mathcal{N}(v)}) & v \in I^n(h) \wedge u \in I^n(h) \\ \mu_v^{(l+1)} = \mu_v^{(l)} & v \in I^n(h) \wedge |I^n(h)| = 1 \end{cases} \tag{6}$$

Next, GINN applies the heightening operator to convert the first order graph into the second order graph (Figure 4). In particular, the heightening operation replaces each active interval (of multiple nodes) on the lower order graph with a single node on the higher order graph (*e.g.* node 3, 4, 5, and 6 are replaced with node 8). In terms of the states of the freshly created nodes, GINN initializes them to be the weighted sum of the states of replaced nodes. To motivate this design decision, we also consider simply averaging the states of replaced nodes in Section 4.3.5 and 4.4.2. The heightening operator is defined below.

Definition 3.2. (*Heightening*) Heightening operator, denoted by η, is a sequence (η_{abs}, η_{com}) where $\eta_{abs} : G^n \rightarrow G^{n+1}$ is a function that maps an n-th order graph into an (n+1)-th order graph, and $\eta_{com} : \mu_{v_1^n}, \mu_{v_2^n}, \ldots \mu_{v_N^n} \in \mathbb{R}^e \rightarrow \mu_{v_1^{n+1}}, \mu_{v_2^{n+1}}, \ldots \mu_{v_{N'}^{n+1}} \in \mathbb{R}^e(\{v_1^n, v_2^n, \ldots v_N^n\}$ denotes the set of nodes on G^n; $\{v_1^{n+1}, v_2^{n+1}, \ldots v_{N'}^{n+1}\}$ denotes the set of nodes on G^{n+1}) is a function that maps a set of e-dimensional vectors for each node on G^n to another set of e-dimensional vectors for each node on G^{n+1}. Equation 7 defines η_{com}.

Like Equation 6, we consider two cases:

$$\begin{cases} \mu_{v_i^{n+1}}^{(0)} = \mu_{v_i^n}^{(l)} & |I^n(h)| = 1 \wedge v_i^n \in I^n(h) \\ \mu_{v_h^{n+1}}^{(0)} = \sum_{v \in I^n(h)} \alpha_v \mu_v^{(l)} & |I^n(h)| > 1 \end{cases} \tag{7}$$

$$\alpha_v = \frac{exp\,(||\mu_v^{(l)}||)}{\sum\limits_{v' \in I^n(h)} exp\,(||\mu_{v'}^{(l)}||)} \tag{8}$$

given that v_i^{n+1} on the (n+1)-th order graph is the same node as v_i^n on the n-th order graph, and v_h^{n+1} is the node that replaces the interval $I^n(h)$. The weight of each state vector, α_v, is determined based on its length (Equation 8).

Now GINN uses the partitioning operator again to produce a set of intervals from the second order graph (Figure 5). Due to the restriction outlined earlier, messages are only passed among node 2, 7, and 8. However, since node 8 can be deemed as a proxy of node 3~6, this exchange, in essence, covers all but node 1 on the first order graph. In other words, even though messages are always exchanged within an interval, heightening abstraction expands the region an interval covers, thus enabling the communication of nodes that were otherwise far apart.

Then, GINN applies the heightening operator the second time, as a result, the third order graph emerges (Figure 6). Since the third order graph is an interval itself, GINN computes the state of

node 1 and 9 in the same way as existing GNN using partitioning operator. Now, GINN reaches an important point, which we call *sufficient propagation*. That is every node has passed messages either directly or indirectly to every other reachable node in the original control flow graph. In other words, the heightening operator is no longer needed.

Upon completion of the message-exchange between node 1 and 9, GINN applies for the first time the lowering operator to convert the third order graph back to the second order graph. In particular, nodes that were created on the higher order graphs for replacing an interval on the lower order graph will be split back into the nodes of which the interval previously consists (*e.g.* node 9 to node 2, 7, and 8). The idea is, after reaching sufficient propagation, GINN begins to recover the state of each node on the original control flow graph. We define the lowering operator below.

Definition 3.3. (*Lowering*) Lowering operator, denoted by λ, is a sequence $(\lambda_{abs}, \lambda_{com})$ where $\lambda_{abs} : G^{n+1} \rightarrow G^n$ is the inverse function of η_{abs}, which maps an (n+1)-th order graph into an n-th order graph, and $\lambda_{com} : \mu_{v_1^{n+1}}, \mu_{v_2^{n+1}}, \ldots \mu_{v_{N'}^{n+1}} \in \mathbb{R}^e \rightarrow \mu_{v_1^n}, \mu_{v_2^n}, \ldots \mu_{v_N^n} \in \mathbb{R}^e (\{v_1^{n+1}, v_2^{n+1}, \ldots v_{N'}^{n+1}\}$ denotes the set of nodes on G^{n+1}; $\{v_1^n, v_2^n, \ldots v_N^n\}$ denotes the set of nodes on G^n) is a function that maps a set of e-dimensional vectors for each node on G^{n+1} to another set of e-dimensional vectors for each node on G^n. Equation 9 defines λ_{com}.

In general, λ_{com} can be considered as a reversing operation of η_{com}:

$$\begin{cases} \mu_{v_i^n}^{(0)} = \mu_{v_i^{n+1}}^{(l)} & |I^{n+1}(h)| = 1 \wedge v_i^{n+1} \in I^{n+1}(h) \\ \mu_v^{(0)} = \alpha_v \mu_{v_h^{n+1}}^{(l)} * |I^{n+1}(h)| & |I^{n+1}(h)| > 1 \end{cases} \tag{9}$$

given that v_i^n on the n-th order graph is the same node as v_i^{n+1} on the (n+1)-th order graph, and v_h^{n+1} is the node that is split into a multitude of nodes, each of which is assigned α_v, the weight defined in Equation 8. $|I^{n+1}(h)|$ is the regulation term that aims to bring node v's initial state and its final state (prior to being replaced by the heightening operator) within the same order of magnitude.

As GINN revisits the second order graph, it re-applies the partitioning operator to produce the only interval to be processed (*i.e.* node 2, 7, and 8). However, there lies a crucial distinction in computing the states of node 2, 7, and 8 this time around. That is the communication among the three nodes is no longer local to their own states, but also inclusive of node 1's thanks to the exchange between node 1 and 9 on the third order graph. In general, such recurring interval propagation improves the precision of node representations by incorporating knowledge gained about the entire graph.

Next, GINN applies the lowering operator again to recover the original control flow graph. After the partitioning operation, GINN computes the state vector for every node on the graph, signaling the completion of a whole learning cycle. The process will be repeated for a fixed number of steps. Then, we use the state vectors from the last step as the node representations. We generalize the above explanation with the running example to formalize GINN's abstraction cycle.

Definition 3.4. (*Abstraction Cycle*) GINN's abstraction cycle is a sequence composed of three primitive operators: partitioning, denoted by ρ; heightening, denoted by η; and lowering, denoted by λ. The sequence can be written in the form of $((\rho \rightarrow \eta)^* \rightarrow \rho' \rightarrow (\lambda \rightarrow \rho)^*)^*$ where $|(\rho \rightarrow \eta)^*| = |(\lambda \rightarrow \rho)^*|$. ρ' denotes the partitioning operator on the highest order graph in which case the graph is a single interval itself.

To sum up, GINN adopts control flow graphs as the program representation. At first, it uses heightening operator to increase the order of graphs, as such, involving more nodes to communicate within an interval. Later, by applying the lowering operator, GINN restores local propagation on the reduced order of graphs, and eventually recovers the node states on the original control flow graph.

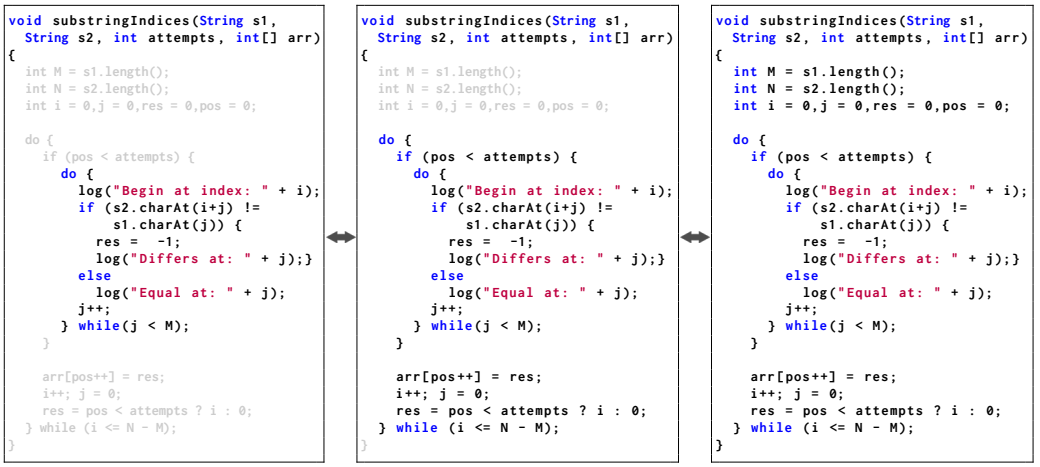

Fig. 7. GINN's focus in source code when transitioning back and forth between different order graphs.

Worth mentioning heightening and lowering operators are complementary and their cooperation benefits the overall GINN's learning objective. Specifically, heightening operator enables GINN to capture the global properties of a graph, with which lowering operator helps GINN to compute precise node representations. As the interleaving between heightening and lowering operator goes on, node representations will continue to be refined to reflect the properties of a graph.

Abstracting the Source Code. We aim to demystify what the above graph abstraction method translates to at the source code level, and why it helps GINN to learn. Since intervals play a pivotal role in all three operators of the graph abstraction method, it is imperative that we understand what an interval represents in a program. As a simple yet effective measure, we use a program, which has the same control flow graph as the one in Figure 2, to disclose what code construct an interval correlates to, and more importantly, how the abstraction method works at the source code level. The program is depicted in Figure 7, which computes the starting indices of the substring `s1` in string `s2`. Recall the only interval GINN processed on the first order graph, the one that consists of node 3~6, which represents the inner `do-while` loop in the function. In other words, GINN focuses exclusively on the inner loop when learning from the first order graph. Similarly, node 2, node 7, and node 8 consist of the only interval from which GINN learns at the second order graph. This interval happens to be the nested `do-while` loop. Finally, GINN spans across the whole program as node 1 and node 9 cover every node in the first order graph. Figure 7 illustrates GINN's transition among the first, second, and third order graph, in particular, the distinct focus on each order graph. Inspired from the exposition above, we present Theorem 3.1 and Corollary 3.1.

Theorem 3.1 (Interval Constituents)**.** Given a loop structure l on a control flow graph, l's loop header, denoted by h, is the entry node of an interval that does not contain any predecessor of h outside of l. However, the interval contains every node inside of l assuming l does not contain an inner loop construct.

Proof. First, we prove the non-coexistence of h and its predecessors outside of l. Assume otherwise, there is an interval I containing at least one predecessor of h outside of l and h itself. By construction, I already had an entry node h'. Due to the loop-forming back edge connecting node m back to h within l, I now has two entry nodes, h' and h, if m is not already in I, which violates the single-entry

rule of an interval. For the other possibility that if m is in I, then by construction all immediate predecessors of m are also in I given that l does not contain an inner loop, recursively all nodes in l are in I, meaning there will be closed paths within I that do not go through h' (*i.e.* those going through h), which violates the definition of interval.

Second, we prove h is the entry node of the interval it is in. Assume otherwise, there exists an interval I, which h is a part of, having a different entry node h'. Since none of h's predecessors outside of l are part of I, at least one of them will connect to h, thus making h the other entry node of I in addition to h'. Therefore, the single-entry rule of an interval is violated.

Finally, we prove that all nodes of l are in the same interval as h if l does not contain an inner loop construct. Assume otherwise, there exists at least one node in l that is not in the interval, I, for which h is the entry node. By construction, at least one immediate predecessor of the node not in I is also not in I. Since l does not contain an inner loop construct, none of the nodes in l has a predecessor outside of l. By recursion, ultimately, the loop header h will be the only immediate predecessor of a node that is not in I. Therefore, h is also not in I, which contradicts the assumption. □

Corollary 3.1 (Hierarchical Intervals for Nested Loops)**.** Given a nested loop structure on a control flow graph, nodes of the n-th most inner loop lie in an interval on the n-th order graph.

Proof. Let P(n) be the statement that nodes of the n-th most inner loop lie in an interval on the n-th order graph. We give a proof by induction on n.

Base case: Show that the statement holds for the smallest number n = 1.
According to Theorem 3.1, P(0) is clearly true: nodes of the most inner loop lie in an interval on the first order graph.

Inductive step: Show that for any k ≥ 1, if P(k) holds, then P(k+1) also holds.
Assume the induction hypothesis that for a particular k, the single case n = k holds, meaning P(k) is true: nodes of the k-th most inner loop lie in an interval on the k-th order graph. By definition, nodes of the k-th most inner loop will be rendered into a single node within the (k+1)-th most inner loop on the (k+1)-th order graph. According to Theorem 3.1, we deduce that nodes of the (k+1)-th most inner loop will also lie in an interval on the (k+1)-th order graph. That is, the statement P(k+1) also holds, establishing the inductive step.
Conclusion: Since both the base case and the inductive step have been shown, by mathematical induction the statement P(n) holds for every number n. □

GINN's Strengths. To conclude, the graph abstraction method translates to a loop-based abstraction scheme at the source code level. In particular, it sorts programs into hierarchies, where each level presents unique looping constructs for models to learn. Since looping constructs are integral parts of a program, this loop-based abstraction scheme helps GINN to focus on the core of a program for learning its feature representation. In scalability regard, the graph abstraction method also offers a crucial advantage. Given the abstracted program graphs, even without the manually designed edges, GINN mostly deals with small, concise intervals. As a result, information rarely needs to be propagated over long distances, thus significantly enhancing GINN's capability in generalizing from the large graphs, an aspect that the existing GNN struggles with.

4 EVALUATION

This section presents an extensive evaluation of GINN on three PL tasks. For each task, we compare GINN against the state-of-the-art.

4.1 Selection Criteria

To select the right downstream applications for evaluating GINN, we devise a list of criteria below in the order of importance.

(1) We prefer tasks for which datasets are publicly accessible and baseline models are open sourced. Furthermore, we only consider models that are built upon Tensorflow, a widely used deep learning library, for reducing the engineering load.
(2) We prefer tasks solved by an influential work that is highly cited in the literature.
(3) We prefer tasks where implementations of GNN-based models are publicly accessible so that we can save our effort in building the baseline with GNN. The reason is it's not sufficient to only compare GINN with the state-of-the-art, which GNN may also outperform.

4.2 Evaluation Tasks

In the end, we select variable misuse prediction task [Allamanis et al. 2018] (150+ citations) in which models aim to infer which variable should be used in a program location. As shown in Figure 8, the variable `clazz` highlighted in red is misused. Instead, variable `first` should have been passed as the argument of the function `Assert.NotNull`. At first, Allamanis et al. [2018] use GGNN to predict the misuse variable from AST-based program graphs (more precisely AST with additional edges). Later, the joint model proposed by [Hellendoorn et al. 2020; Vasic et al. 2019] has achieved better results and thus becomes the new state-of-the-art.

```
var clazz = classTypes["Root"].Single() as ...
Assert.NotNull(clazz);

var first = classTypes["RecClass"].Single() as ...
Assert.NotNull(clazz);

Assert.Equal("string", first.Properties["Name"].Name);
Assert.False(clazz.Properties["Name"].IsArray);
```

Fig. 8. An example (extracted from [Allamanis et al. 2018]) of variable misuse. `clazz` highlighted in red is misused. Instead, `first` should have been used.

In addition, we select method name prediction in which models aim to infer the name of a method when given its body. Figure 9 shows the problem setting where the function name has been stripped for models to predict. The correct answer in this case is `reverseArray`. Alon et al. [2019b] (120+ citations) create the first large-scale, cross-project classification task in method names. Later works adopted a generative approach that generates method names as sequences of words [Alon et al. 2019a; Fernandes et al. 2019; Wang and Su 2020].

```
public string[] f(string[] array)
{
    string[] newArray = new string[array.Length];
    for (int index = 0; index < array.Length; index++)
        newArray[array.Length-index-1] = array[index];

    return newArray;
}
```

Fig. 9. An example function extracted from [Alon et al. 2019b] whose name is left out for models to predict.

Last, we design a new task to evaluate GINN. Granted, machine learning has inspired a distinctive approach to a broad range of problems in program analysis. However, most problems are catered to

the strength of machine learning models in mining statistical patterns from data. It's unclear how well they can solve the long-standing problems that are traditionally tackled by symbolic, logic-based methods. In this task, we examine if models can accurately detect null pointer dereference, a type of deep, semantic, and prevalent bugs in software. In particular, we compare GINN against not only the standard GNN but also arguably the state-of-the-art static analysis tool Facebook Infer [Berdine et al. 2006; Calcagno et al. 2015].

4.3 Variable Misuse Prediction

The state-of-the-art model for predicting variable misuse bugs is proposed by Hellendoorn et al. [2020]. Their model is an instantiation of the conceptual framework invented by Vasic et al. [2019]. In this section, we first describe this framework, then we explain how to instantiate it using GINN.

4.3.1 Joint Model for Localization and Repair. Vasic et al. [2019] propose a joint model structure that learns to localize and repair variable misuse bugs simultaneously. Intuitively, their design exploits an important property of variable misuse bugs. That is both the misuse (*i.e.* incorrectly used) and the repair (*i.e.* should have been used) variable have already been defined somewhere in the program, therefore, the key is to identify their locations. Vasic et al. [2019] learned probability distributions over the sequence of tokens of a buggy program from which they pick the token of the highest probability to be the misuse or repair variable. At a high-level, the joint model consists of three major components: initial embedding layer, core model, and two separate classifiers. Below we describe each component in detail.

Initial Embedding Layer. Initial embedding layer is responsible for turning each symbol from the input vocabulary (*e.g.* a token or a type of AST node) into a numerical vector, the format that is amenable to deep neural networks. A simple, crude method is to use the one-hot vectors, a $N \times N$ matrix representing every word in a vocabulary (N denotes the size of the vocabulary). Each row corresponds to a word, which consists of 0s in all cells with the exception of a single 1 in a cell used uniquely to identify the word. For example, given a vocabulary, {a, +, b, `binary-exp`}, the matrix M in Equation 10 depicts the one-hot vectors of each token. A drawback of one-hot vectors is they don't capture the semantic relationship of words in the vocabulary as vectors are uniformly scattered in a high dimensional space (*i.e.* the Euclidean distance between any vector with every other vector is a constant). A common remedy is to have the one-hot vectors multiply another matrix $W_1 \in \mathbb{R}^{N \times d}$ to produce $\mathcal{E} \in \mathbb{R}^{N \times d}$. The intuition is to expand the network's capacity with more learnable parameters in W_1 for searching a precise embedding matrix $\mathcal{E}$ for each word in the vocabulary. d denotes the number of columns in W_1, which is also the size of the embedding vectors. Equation 10 gives an example where d equals to 3. The values of W_1 is initialized randomly and will be learned simultaneously with the network during training.

$$\underbrace{\begin{matrix}\mathsf{a}:\\ \mathsf{+}:\\ \mathsf{b}:\\ \mathsf{binary{-}exp}:\end{matrix}\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}}_{M}\underbrace{\begin{bmatrix}\alpha_{11} & \alpha_{12} & \alpha_{13}\\ \alpha_{21} & \alpha_{22} & \alpha_{23}\\ \alpha_{31} & \alpha_{32} & \alpha_{33}\\ \alpha_{41} & \alpha_{42} & \alpha_{43}\end{bmatrix}}_{W_1} = \underbrace{\begin{matrix}\mathsf{a}:\\ \mathsf{+}:\\ \mathsf{b}:\\ \mathsf{binary{-}exp}:\end{matrix}\begin{bmatrix}\beta_{11} & \beta_{12} & \beta_{13}\\ \beta_{21} & \beta_{22} & \beta_{23}\\ \beta_{31} & \beta_{32} & \beta_{33}\\ \beta_{41} & \beta_{42} & \beta_{43}\end{bmatrix}}_{\mathcal{E}} \tag{10}$$

Core Model. Given the embedding vector of each token, the core model computes the numerical representation of the input program, expressed as another matrix C where each row represents a token. Many neural architectures can play the role of core model. Vasic et al. [2019] used Long Short Term Memory networks (LSTM) [Hochreiter and Schmidhuber 1997], a specialization of Recurrent Neural Networks (RNN). Hellendoorn et al. [2020] explored other alternatives including

GGNN, Transformer [Vaswani et al. 2017], the state-of-the-art sequence model, and RNN Sandwich, a mixture of sequence and graph models, which achieves the state-of-the-art results. For now, a core model can be considered as a black-box with an abstract interface $\phi : \mathcal{E} \rightarrow C$. We defer the discussion of two instantiations of the core model to later sections.

Classifiers for Misuse and Repair Variables. This layer is responsible for predicting the location of misuse and repair variable[2]. It distributes two probabilities to each token in the program: one for identification of the misuse variable and the other for the repair variable. The token with the highest probability of being the misuse or repair variable will be picked as the output. At a technical level, Vasic et al. [2019] performed a linear transformation on matrix $C \in \mathbb{R}^{k \times h}$ using another matrix $W_2 \in \mathbb{R}^{h \times 2}$ to produce $P_{val} \in \mathbb{R}^{k \times 2}$. As explained before, k is the number of tokens in a program; h is the size of the embedding vector produced by a core model.

$$P_{val} = CW_2$$

Since P_{val} is a non-normalized output, we apply $softmax$ function, a common practice in machine learning, to map P_{val} into probability distributions over the token sequence of an input program. Note that $axis = 1$ in Equation 11 means the normalization happens along each column in which each value denotes the probability of one token being the misuse or repair variable.

$$P_{pro} = softmax(P_{val}, axis = 1) \tag{11}$$

Finally, the model is trained to nudge P_{pro} as close to the true distribution of misuse and repair variables as possible. In particular, we minimize the cross-entropy loss expressed by the difference between P_{pro} and the true distributions, denoted by Loc and Rep. Similar to the one-hot vectors, Loc (*resp.* Rep) assigns a value of 1 to the actual index of misuse (*resp.* repair) variable among all tokens of the input program and 0 otherwise.

4.3.2 Instantiations of the Core Model. In this section, we review two instantiations of the core model presented in Hellendoorn et al. [2020]: GGNN and RNN Sandwich. In particular, we discuss how each of them computes the matrix C introduced earlier.

As explained in Section 4.2, GGNN works with AST as the backbone of graphs. To collect the numerical representations of each token in the program, Hellendoorn et al. [2020] extracted states of terminals nodes after GGNN had completed the message-passing routine, and then stacked them into matrix C (Equation 12).

$$C = stack([\mu_{v_1}, \ldots, \mu_{v_n}], axis = 0), \forall v_i \in \Sigma \tag{12}$$

where Σ denotes the set of terminal nodes in AST; $axis = 0$ indicates each μ_v forms a row in C.

Next, we discuss RNN Sandwich [Hellendoorn et al. 2020], the state-of-the-art model in predicting variable misuse bugs. In a nutshell, RNN Sandwich adopts a hybrid neural architecture combining a sequence model RNN and a graph model GGNN intending to get the best of both worlds. That is, not only exploiting the semantic code structure using GGNN but also streamlining information flow through token sequences using RNN. Technically, the way it works is the following: (1) first, the matrix $\mathcal{E}$ produced in the initial embedding layer will be fed into an RNN. The results are a list of vectors, $h_1, \ldots, h_n$, each of which represents a token. Compared to the initial embedding matrix $\mathcal{E}$, $h_1, \ldots, h_n$ increases the precision of the token representation by capturing the temporal properties they display in a sequence. Equation 13 computes $h_1, \ldots, h_n$. In simplest terms, RNN takes two vectors at each time step — the embedding vector of the t-th token in the sequence, denoted by

[2]We followed Hellendoorn et al. [2020]'s approach to consider a single variable misuse bug per program.

$\mathcal{E}[t]$ (assuming the t-th row in $\mathcal{E}$ is the embedding of the t-th token), and RNN's current hidden state after consuming the embedding of the previous token. The output is the new hidden state h_t.

$$h_t = RNN(\mathcal{E}[t], h_{t-1}) \tag{13}$$

(2) Next, GGNN takes over and computes the state vector for each node via the message-passing protocol. For node initialization, the token representations computed in Step (1) are assigned to the terminal nodes in an AST (Equation 14) while the non-terminal nodes keep their representations

$$\mu_{v_i} = h_i, \forall v_i \in \Sigma, \forall h_i \in [h_1, \ldots, h_n] \tag{14}$$

computed from the initial embedding layer. (3) Finally, after GGNN has computed the state of each node in an AST, RNN takes back those of the terminal nodes and computes a new representation for each token, $h'_1, \ldots, h'_n$, according to Equation 15 where μ_{v_t} denotes the state vector of the node

$$h'_t = RNN(\mu_{v_t}, h'_{t-1}) \tag{15}$$

corresponding to the t-th token in the program. Finally, the matrix C can be computed by Equation 16.

$$C = stack([h'_1, \ldots, h'_n], axis = 0) \tag{16}$$

4.3.3 New Instantiations of Core Models Using GINN. We propose two new instantiations built upon GINN to pair with GGNN and RNN Sandwich. In order to adapt GINN as an instantiation of the core model, an important issue needs to be addressed. Recall the interface defined for the core model: $\phi : \mathcal{E} \rightarrow C$, since GINN can not compute representations of tokens from a standard control flow graph, matrix C can't be produced. Therefore, we modify control flow graphs as follows: (1) first we split a graph node representing a basic block into multiple nodes, each of which represents a single statement. Subsequently, we add additional edges to connect every statement with its immediate successor within the same basic block. For edges on the original control flow graphs, we change their start (*resp.* end) nodes from a basic block to its last (*resp.* first) statement after the split; (2) next, we replace each statement node with a sequence of nodes, each of which represents a token of the statement. Specifically, every token node is connected to its immediate successor, and the first token node will become the new start or end node of edges that were connecting the statement nodes before.

Figure 10b depicts an example of the new graph representation given the function in Figure 10a. Note that our modification will not alter the partitioning of a graph into intervals. Instead, we only replace nodes denoting basic blocks with those denoting token as the new constituents of an interval. We have included a detailed explanation in Appendix A for readers' perusal. Under the new graph representation, GINN can compute the state vector for every token in the program, and in turn the matrix C. Therefore, we can now instantiate the core model with GINN. Similarly, we can easily construct a variant of RNN Sandwich by swapping out GGNN for GINN as the new graph component.

4.3.4 Experimentation. Given the conceptual framework of the joint model and various instantiations discussed above, we now describe the experiment setup.

Dataset, Metric, and Baseline. Hellendoorn et al. [2020] worked with ETH Py150 [Raychev et al. 2016], a publicly accessible dataset of python programs, on top of which they introduced buggy programs to suit the task of variable misuse prediction. Since the actual dataset used in their experiment is not publicly available, we follow their pre-processing steps in an attempt to replicate their dataset. Major tasks include the deduplication of ETH Py150, generation of buggy programs, and preservation of original programs as correct examples. Like their dataset, we maintain a balance between buggy and correct programs. In the end, we have 3M programs in total, among which

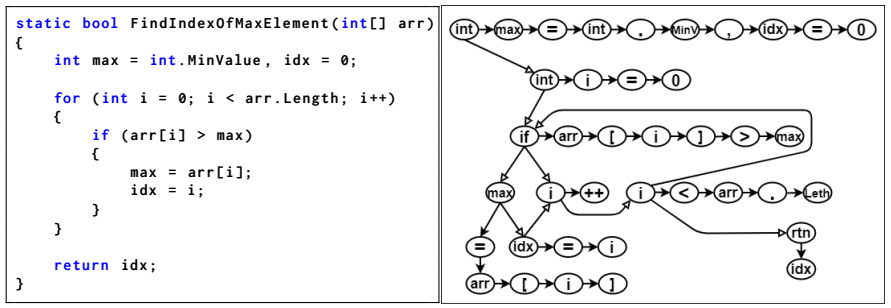

Fig. 10. The program graph in (b) that represents the function in (a). Edges with hallow arrow heads represent the control flows between program statements. The other edges connect tokens in a sequence.

We use 2M for training, 245K for validation, and 755K for test. A detailed description of the data generation is provided in Appendix B. Similarly, we adopt the metrics proposed in [Vasic et al. 2019] to measure the model performance, such as (1) classification accuracy, the percentage of programs in the test set that are correctly classified as either buggy or correct; (2) localization accuracy, the percentage of buggy programs for which the bug location is correctly predicted; and (3) localization+repair accuracy (or joint accuracy), the percentage of buggy programs for which both the location and repair are correctly predicted. Regarding baselines, we use GGNN and RNN Sandwich as the two instantiations of the core model.

Implementation. Since ETH Py150 provides AST for programs in the dataset, we convert each AST into the graph representation that GINN consumes (Appendix C). We also implement Algorithm 1 to partition the graphs into intervals. Since prior works have not open-sourced the joint model framework, we implemented our own in Tensorflow. Regarding the core models, we use Allamanis et al. [2018]'s implementation of GGNN, on top of which we realize GINN's graph abstraction method while keeping all GGNN's default parameters (*e.g.* size of node embeddings, number of layers, *etc.*) intact. We only implement one abstraction cycle of GINN's (original graph→highest order graphs→original graphs) as more cycles do not make a significant difference. For RNN Sandwich, we use the large sandwich model [Hellendoorn et al. 2020], which is shown to work better than the vanilla model presented in Section 4.3.2. The improvement is due to the increased frequency of the alternation between sequence and graph models. That is, instead of inserting RNN only before and after GNN's computation, they wrap every few rounds of message-passing inside of GNN with an RNN to facilitate a thorough exchange between two neural architectures. We couple GINN (*resp.* GGNN) with an RNN to implement the GINN- (*resp.* GGNN-) powered RNN Sandwich. Note that Hellendoorn et al. [2020] also propose a lightweight graph representation by keeping only the terminal nodes from an AST as the input for both GGNN and RNN Sandwich. Even though these new program graphs have made models faster to train, their accuracy remains more or less the same. Hence, we don't consider the lightweight graphs in our evaluation.

All experiments including those to be presented later are performed on a desktop that runs Ubuntu 16.04 having 3.7GHz i7-8700K CPU, 32GB RAM, and NVIDIA GTX 1080 GPU. As a pre-test, we repeat the experiment presented in [Hellendoorn et al. 2020], and observed comparable model performance for both GGNN and GGNN-based RNN Sandwich, signaling the validity of both our model implementation and data curation.

4.3.5 Results. First, we compare the performance of GINN (*resp.* GINN-powered RNN Sandwich) against GGNN (*resp.* GGNN-powered RNN Sandwich). Then we investigate the scalability of these

four neural architectures. Later, we conduct ablation studies to gain a deeper understanding of GINN's inner workings. Finally, we evaluate an alternative design of GINN.

Table 1. Results for the different instantiations of core models. For GINN and GINN-powered RNN Sandwich, we also include their results of an ablation study and an evaluation on an alternative design.

Models	Configuration	Classification Accuracy	Localization Accuracy	Localization+Repair Accuracy
GGNN	Original	74.0%	58.1%	56.0%
GINN	Original	**74.9%**	**60.5%**	**60.0%**
	Ablated	74.7%	58.8%	57.6%
	Alternative	74.0%	59.5%	58.7%
Sandwich (GGNN)	Original	74.7%	62.9%	61.1%
Sandwich (GINN)	Original	**75.8%**	**69.3%**	**68.4%**
	Ablated	75.1%	62.4%	62.3%
	Alternative	75.0%	63.1%	62.0%

Accuracy. Table 1 depicts the results of all four models using the aforementioned metrics. We focus on rows for the original configuration of each model, and discuss the rest later. Note that the results we obtained for the two baseline models are lesser than those reported in Hellendoorn et al. [2020] (*e.g.* 5.9%/6.5% in localization accuracy that GGNN/RNN Sandwich achieved on the *test* set). The reason for this discrepancy is that the actual datasets used by the two works are different even though both are curated out of ETH-Py150, furthermore, our dataset is technically harder to deal with since larger functions occupy a higher proportion. For example, programs exceeding 250 tokens make up 6.5% of their test set [Hellendoorn et al. 2020] compared to 13.4% of our test set. More details can be found in Appendix B.

Regarding the classification accuracy, all models perform reasonably well, and GINN-powered RNN Sandwich and GINN are the top two core models despite by a small margin. For more challenging tasks like localization or repair, GINN-powered RNN Sandwich now displays a significant advantage over all other models, in particular, it outperforms GGNN-powered RNN Sandwich, the state-of-the-art model in variable misuse prediction, by 6.4% in localization and 7.3% in joint accuracy. To dig deeper, we manually inspect the predictions made by each model and find that GINN-powered RNN Sandwich is considerably more precise at reasoning about the semantics of a program. Below we give two examples to illustrate our findings.

For Figure 11a, GINN-powered RNN Sandwich is the only model that not only locates but also fixes the misused variable (*i.e.* `orig_sys_path` highlighted within the shadow box). In contrast, all baseline models consider `orig_sys_path` as the correctly used variable, which is not totally unreasonable. In fact, appending an item (`item`) when it is not yet in the list (`orig_sys_path`) is a very common pattern models need to learn. However, in this case, if `orig_sys_path` was the correct variable, `new_sys_path` would have been an empty list when being assigned to `sys.path[:0]` in the last line. GINN-powered RNN Sandwich is capable of capturing the nuance of the semantics this program denotes, and not getting trapped by the common programming paradigms. As for Figure 11b, GINN and GGNN-powered RNN Sandwich also correctly localizes the misused variable `request`, but none of them predicts the right repair variable `sq`. The signal there is the fact that `fts` is a list, which does not have `keys()` method. GINN-powered RNN Sandwich is again the only neural architecture that produces the correct prediction end-to-end, demonstrating its higher precision in reasoning about the semantics of a program.

```python
def add_vendor_lib():
    orig_sys_path = set(sys.path)

    new_sys_path = []
    for item in list(sys.path):
        if item not in orig_sys_path:
            orig_sys_path.append(item)
            sys.path.remove(item)

    sys.path[:0] = new_sys_path
```

(a)

```python
def list_stored_queries(self, request):
    sq = super(GeoDjangoWFSAdapter,
               self).list_stored_queries(request)
    fts = list(self.models.keys())

    for k in request.keys():
        sq[k] = StoredQueryDescription(name=k, feature_types=fts,
                                       title=k, parameters=[])

    return sq
```

(b)

Fig. 11. Two programs only GINN-powered RNN Sandwich makes the correct prediction end-to-end.

Scalability. We further analyze the results we obtained from the previous experiment to investigate the scalability of the four neural architectures. In particular, we divide the entire test set into ten subsets, each of which consists of graphs of similar size. We then record the performance of all models on each subset starting from the smallest to the largest graphs. Note that GGNN, at the core of both compared baselines, has already had additional edges incorporated into the program graphs to alleviate the scalability concern [Allamanis et al. 2018]. However, as explained in Section 1, Allamanis et al. [2018] do not provide a principled guideline as to where exactly to add the edges given a program graph specifically. Therefore, we can only follow their approach by universally adding all types of edges they proposed to all program graphs. In contrast, we do not incorporate any additional edge apart from those that are present in the graph representations that GINN deals with.

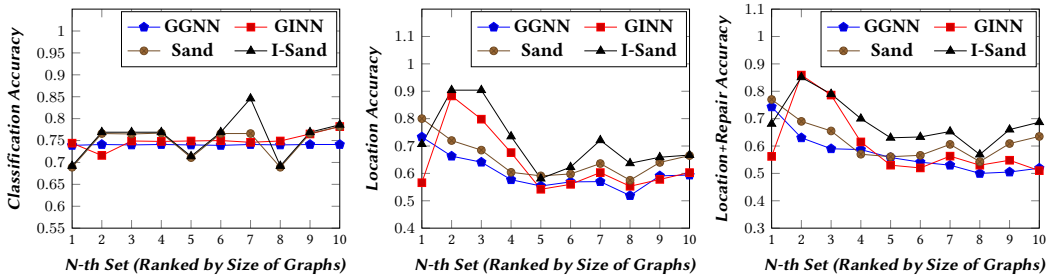

Fig. 12. Investigating the scalability of all four neural architectures. I-Sand (*resp.* Sand) denotes the GINN-powered RNN Sandwich (*resp.* GGNN-powered RNN Sandwich).

Figure 12 shows how the performance of each model varies with the size of graphs[3]. We skip the metric of classification accuracy under which models are largely indistinguishable. For localization and joint accuracy, we make several observations of the performance trend for each model. First, GGNN suffers the largest performance drop among all models — more than 10% (*resp.* 20%) under location (*resp.* joint) accuracy. This phenomenon shows there is indeed a scalability issue with GGNN. Second, the combination of sequence and graph model helps to make the neural architecture more scalable than GGNN alone. Furthermore, by replacing GGNN with GINN in RNN Sandwich, we obtain a model that is clearly the most scalable. Barring few sets of graphs, GINN-powered RNN Sandwich is considerably more accurate than any other model, especially under the joint accuracy.

[3]The reason that GINN (or GINN-based Sandwich) is almost always worse than GGNN (or GGNN-based Sandwich) on the first set of graphs is mainly because 99.4% of the methods in the set do not have loops, as a result, their graphs will be treated as a single interval, in which case GINN is functionally equivalent to GGNN.

Finally, GINN alone also manages to improve the scalability of GGNN. As described earlier, the improvement is magnified under the assistance of the RNN in the sandwich model.

Ablation Study. Unlike the prior work [Allamanis et al. 2018], all edges in our graph representation are indispensable, thus can not be ablated. Therefore, the goal of this ablation study is to quantify the influence of the graph abstraction method, the crux of GINN's learning approach, on GINN's performance. Because GINN is identical to GGNN without the graph abstraction operators, we evaluate GGNN when GINN's program graphs — a variant of control flow graph — are provided as input data. Rows for ablated configuration in Table 1 show GINN (*resp.* GINN-powered RNN Sandwich) becomes only slightly more accurate than GGNN (*resp.* GGNN-powered RNN Sandwich) after the ablation, in other words, our graph abstraction method indeed helps models to learn.

Alternative Design. As mentioned in Section 3, we evaluate an alternative design of GINN in which freshly created nodes — due to the heightening operator — are initialized to be the average of the replaced nodes; conversely, for nodes that are created by the lowering operator, they receive an equal share of the split node that they emerge from. In essence, we assign the weight, α_v, used in Equation 7 and 9 to be $1/|I_h^n|$. Rows of alternative configuration in Table 1 show that the new design results in a notable decrease in model accuracy, which motivates the original design of GINN.

4.4 Method Name Prediction

The state-of-the-art model in method name prediction, sequence GNN [Fernandes et al. 2019], adopts a typical encoder-decoder architecture [Cho et al. 2014; Devlin et al. 2014]. In a similar vein to RNN Sandwich, sequence GNN also employs a combination of graph and sequence model to encode a program, however, the synergy between the two neural architectures in this case is considerably simpler. They only perform the first two steps of the vanilla joint model presented in Section 4.3.2. That is, RNN first learns the sequence representation of each token in a program before GGNN computes the state for every node in the AST. For decoder, they use another RNN, which generates the method name as a sequence of words. While decoding, it also attends to the states of nodes in the AST, a common technique for improving the accuracy of models adopting encoder-decoder architecture [Bahdanau et al. 2015].

Instantiating the framework of sequence GNN's with GINN is a straightforward task. We use GINN as the new graph model for the encoder while keeping the remaining parts of the framework intact. Like prior experiments, GINN works with the same graph format derived from control flow graphs and customized to include nodes of tokens.

4.4.1 Experimentation. Given the sequence GNN's framework and the instantiation with GINN, we describe the setup of our experiment.

Dataset, Metric, and Baseline. We consider Java-small proposed by Alon et al. [2019a], a publicly available dataset used in [Fernandes et al. 2019], adopting the same train-validation-test splits they have defined. We also follow Fernandes et al. [2019]'s approach to measure performance using F1 and ROGUE score over the generated words. For sequence GNN baseline, we use the model that achieves state-of-the-art results on Java-small, which employs bidirectional-LSTM as the sequence and GGNN as the graph model for encoder and another LSTM for decoder. Apart from the aforementioned attention mechanism, decoder also integrates a pointer network [Vinyals et al. 2015] to directly copy tokens from the input program.

Implementation. We use JavaParser to extract AST out of programs in Java-small. We adopt the same procedure as before to convert AST into the graph representation that GINN consumes. Regarding the model implementation, we use the model code of sequence GNN open-sourced on GitHub, within which we implemented GINN's graph abstraction methods. Like before, all

```java
private int extractModifiers(GroovySourceAST ast) {
  GroovySourceAST modifiers = ast.childOfType(MODIFIERS);

  if (modifiers == null) return 0;
  int modifierFlags = 0;
  for (GroovySourceAST child =
      (GroovySourceAST) modifiers.getFirstChild();
      child != null;
      child = (GroovySourceAST) child.getNextSibling()) {
    switch (child.getType()) {
    case LITERAL_private:
      modifierFlags |= Modifier.PRIVATE; break;
    case LITERAL_protected:
      modifierFlags |= Modifier.PROTECTED; break;
    case LITERAL_public:
      modifierFlags |= Modifier.PUBLIC; break;
    case FINAL:
      modifierFlags |= Modifier.FINAL; break;
    case LITERAL_static:
      modifierFlags |= Modifier.STATIC; break;
  }}

  return modifierFlags;
}
```

(a) Predicted to be `SelectModifier` by the baseline.

```java
public void applyRules(@Nullable String pluginId, Class<?> clazz) {
  ModelRegistry modelRegistry = target.getModelRegistry();

  Iterable<Class<? extends RuleSource>> declaredSources =
      ruleDetector.getDeclaredSources(clazz);

  for (Class<? extends RuleSource> ruleSource : declaredSources) {
    Iterable<ExtractedModelRule> rules =
        ruleInspector.extract(ruleSource).getRules();
    for (ExtractedModelRule rule : rules) {
      for (Class<?> dependency : rule.getRuleDependencies()) {
        target.getPluginManager().apply(dependency);
      }

      rule.apply(modelRegistry, ModelPath.ROOT);
    }
  }
}
```

(b) Predicted to be `CheckSource` by the baseline.

Fig. 13. Two programs sequence GINN predicts correctly but not the baseline.

default model parameters in their implementation are kept as they are and we only implement one abstraction cycle within GINN. We name our model built out of GINN sequence GINN.

4.4.2 Results. In this section, we first compare the performance of sequence GINN against sequence GNN. Then we investigate the scalability of both neural architectures. Finally, we conduct a similar ablation study and alternative design evaluation with those presented in the previous task.

Accuracy. Table 2 depicts the results of the two models. Sequence GINN significantly outperforms the sequence GNN across all metrics (*e.g.* close to 10% in F1). Through our manual inspection, we find that sequence GINN's strength stands out when dealing with programs of greater complexity. Take programs in Figure 13 as examples, even though both of them denote relatively simple semantics, sequence GNN struggles with their complexity (*e.g.* large program size for Figure 13a, and nested looping construct for Figure 13b). Therefore, its predictions do not precisely reflect the semantics of either program. sequence GNN predicts the name of the program in Figure 13a (*resp.* 13b) to be `SelectModifier` (*resp.* `CheckSoucre`) whereas sequence GINN produces correct predictions for both programs, which are highlighted within the shadow boxes.

Table 2. The first two rows show the results for sequence GINN and sequence GNN; the next two rows are the results of the ablation study and the evaluation on alternative design.

Models	Configuration	F1	ROGUE-2	ROGUE-L
Sequence GNN	Original	51.3%	24.9%	49.8%
Sequence GINN	Original	**60.2%**	**29.5%**	**54.7%**
	Ablated	56.3%	26.3%	51.6%
	Alternative	52.2%	24.8%	50.4%

Scalability. In the same setup as the scalability experiment of variable misuse prediction task, we investigate the scalability of both models. To strengthen sequence GNN, we incorporate all additional edges to its program graphs [Allamanis et al. 2018]. As depicted in Figure 14, under all three metrics, sequence GINN outperforms sequence GNN throughout the entire test set, especially in F1 where sequence GINN displays a wider margin over sequence GNN on larger graphs (*i.e.* last 5 sets of the graphs except the seventh set) than it does on smaller graphs (*i.e.* first 5 sets of graphs).

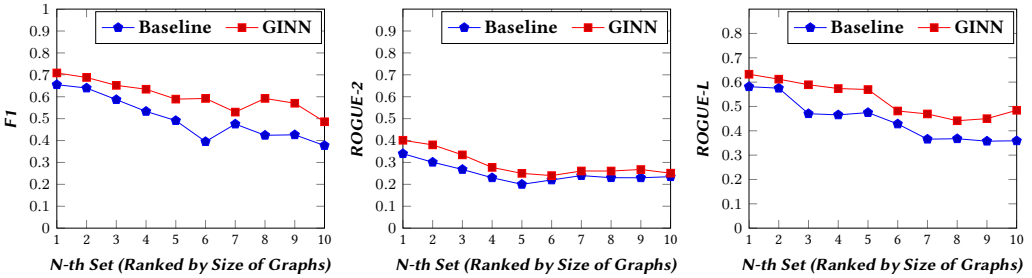

Fig. 14. Investigating the scalability of all four neural architectures.

Ablation Study. We conduct the same ablation study to quantify the influence of the graph abstraction method on the performance of sequence GINN. As shown in Table 2, without the graph abstraction operators, sequence GINN still performs somewhat better than the baseline but notably worse than the original configuration, indicating the considerable influence of the abstraction method on sequence GINN's performance.

Alternative Design. Following the evaluation of the alternative design in variable misuse prediction, we adopt the same approach to initializing the state of freshly created nodes. The last row in Table 2 shows that this design causes GINN to be less accurate by a fairly wide margin.

4.5 Detection of Null Pointer Dereference

First, we present a framework for creating neural bug detectors in catching null pointer dereference. Then, we describe our experiment setup. Finally, we report our evaluation results.

Neural Bug Detection Framework. We devise a simpler program representation for this task. Given a control flow graph, we only split a node of basic block into multiple nodes of statement to aid the localization of bugs at the level of lines. When checking a method, we also include its context by stitching the graphs of all its callers and callees to that of itself. Regarding the initial state of each statement node, we use RNN to learn a representation of its token sequence. As explained in Section 4.3.1, after each token is embedded into a numerical vector (by the initial embedding layer), we feed the embeddings of all tokens in a sequence to the network whose final hidden state (*i.e.* the last h_t in Equation 13) will be extracted as the initial node state.

After GINN computes the state vector for each node, and in turn the entire graph (by aggregating the states of all nodes), we train a feed-forward neural network [Svozil et al. 1997] to predict (using the representation of the entire graph) whether or not a method is buggy. If it is, we will use the same network to predict the bug locations. Our rationale is a graph as a whole provides stronger signals for models to determine the correctness of a method. Correct methods will be refrained from further predictions, leading to fewer false warnings. On the other hand, if a method is indeed predicted to be buggy, our model will then pick the top N statements ranked by their probabilities as potential bug locations. This prediction mode intends to provide flexibility for developers to tune their analysis toward either producing less false warnings or identifying more bugs.

Dataset. We target Java code due to the popularity and availability of Java datasets. In addition to bugs provided by existing datasets [Just et al. 2014; Saha et al. 2018; Ye et al. 2014], we extract additional bugs from Bugswarm projects [Tomassi et al. 2019] to enrich our dataset for this experiment. For example, given a bug description "*Issue #2600...*" contained in a commit message, we search in the bug tracking system (*e.g.* Bugzilla) using id *#2600* to retrieve the details of the bug, against which we match pre-defined keywords to determine if it's a null pointer dereference bug. Next,

we refer back to the commit to find out bug locations, specifically, we consider the lines that are modified by this commit to be buggy and the rest to be clean. We acknowledge the location of a bug and its fix may not be precisely the same thing. If there are ever cases where two locations significantly differ, GINN and the compared baseline will be equally affected.

Any detection is in nature a classification problem having subjects of interest one class of the data, and the rest the other. Therefore, in the context of bug detection, it's also necessary to supply the correct methods apart from the buggy methods for training. A natural approach would be directly taking the fixed version of each buggy method. However, this approach is unlikely to work well in practice. Because in a real problem setting, an analyzer will never set out to differentiate the two versions (*i.e.* correct and buggy) of the same program, instead, it has to separate buggy programs from correct programs that are almost guaranteed to be functionally different. For this reason, we pair each buggy method with a correct method that is syntactically the closest (defined by the tree edit distance between their AST) from the same project. However, due to the shortage of bug instances in existing datasets even after taking into account the additional bugs we extracted ourselves, we include more correct methods *w.r.t.* each buggy method to further enlarge our dataset. Table 6 in Appendix D shows the details including the projects from which we extract the code snippet, a brief description of their functionality, size of their codebase, and the number of buggy methods we extract from each project. We maintain an approximately 3:1 ratio between the correct and buggy methods for each project. In total, we use 64 projects for training and 13 for test.

Objective, Metric, and Baseline. We evaluate how accurately the neural bug detector can localize a null pointer dereference within a method. Because warning developers about buggy functions hardly makes a valuable tool. As we deal with an unbalanced dataset (*i.e.* cleaning lines are more than buggy lines), we forgo the accuracy metric since a bug detector predicting all lines to be correct would yield a decent accuracy but make a totally useless tool. Instead, we opt for precision, recall, and F1 score, metrics are commonly used in defect prediction literature [Pradel and Sen 2018; Wang et al. 2016]. To show our interval-based graph abstraction method can improve a variety of graph models, we use the standard GNN to build a baseline bug detector for this experiment. We do not include classical static analysis tools as additional baselines since comparing them against learning-based bug detectors is likely to be unfair due to the noise issue raised earlier. That is static analyzers could very well discover hidden bugs or report different locations of the same bug. Properly handling those situations require manual inspection, which is hard to scale. On the other hand, machine learning models are less susceptible to this problem as the training set yield in principle the same distribution of the test set.

Implementation. We construct the control flow and interval graphs using Spoon [Pawlak et al. 2015], an open-source library for analyzing Java source code. To efficiently choose correct methods to pair with a buggy method, we run DECKARD [Jiang et al. 2007], a clone detection tool adopting an approximation of the shortest tree edit distance algorithm to identify similar code snippets. To realize the neural bug detector powered by GNN, we make two major changes to Allamanis et al. [2018]'s implementation of GGNN. First, we adopt the method defined in Equation 2 and 3 for updating the states of nodes in the message-passing procedure. Second, we design two loss functions both in the form of cross-entropy for the bug detection task. The first one is designed for the classification of buggy and correct methods and the other for the classification of buggy and clean lines inside of a buggy method. Next, we realize the graph abstraction method within the implementation of GNN-powered bug detector to build GINN-powered bug detector. Similar to prior tasks, we implement only one abstraction cycle for GINN. For the remaining GGNN's parameters after our re-implementation, we keep them in both bug detectors.

Performance. Table 3–5 depict the precision, recall, and F1 for both bug detectors. We also examine

Table 3. Precision.

Methods	Top-1	Top-3	Top-5
GNN (0)	0.209	0.147	0.117
GNN (1)	*0.285*	0.166	0.131
GNN (2)	0.218	0.138	0.101
GINN (0)	0.326	0.174	0.138
GINN (1)	**0.351**	0.167	0.130
GINN (2)	0.305	0.195	0.136
Gain	**0.351** - *0.285* = 0.066		

Table 4. Recall.

Methods	Top-1	Top-3	Top-5
GNN (0)	0.243	0.373	*0.450*
GNN (1)	0.171	0.252	0.302
GNN (2)	0.210	0.337	0.375
GINN (0)	0.282	0.374	0.447
GINN (1)	0.329	0.431	**0.507**
GINN (2)	0.304	0.428	0.500
Gain	**0.507** - *0.450* = 0.057		

Table 5. F1 score.

Methods	Top-1	Top-3	Top-5
GNN (0)	*0.224*	0.211	0.186
GNN (1)	0.214	0.201	0.183
GNN (2)	0.214	0.196	0.159
GINN (0)	0.303	0.238	0.211
GINN (1)	**0.339**	0.241	0.207
GINN (2)	0.304	0.268	0.214
Gain	**0.339** - *0.224* = 0.115		

the impact of the program context — denoted by the number in parenthesis — on the performance of each bug detector. More concretely, (0) means each method will be handled by itself. (1) stitches control flow graphs of all callers and callees (denoted by $\mathcal{F}$) to that of the target method, and (2) further integrates the graphs of all callers and callees of every method in $\mathcal{F}$. Exceeding 2 has costly consequences. First, most data points will have to be filtered out for overflowing the GPU memory. Moreover, a significant portion of the remaining ones would also have to be placed in a batch on its own, resulted in a dramatic increase in training time. We provide the same amount of context for each method in the test set as we do in the training set to maintain a consistent distribution across the entire dataset. Columns in each table correspond to three prediction modes in which models pick 1, 3, or 5 statements to be buggy after a method is predicted buggy.

Overall, the neural bug detector built out of GINN consistently outperforms the baseline using all proposed metrics, especially in F1 score where GINN beats GNN by more than 10%. Regarding the impact of the context, we find that more context mostly but not always leads to improved performance of either bug detector. The reason is, on one hand, more information will always be beneficial. On the other hand, exceedingly large graphs hinders the generalization of graph models, resulted in the degraded performance of the neural bug detectors.

Scalability. Similar to the prior studies, we investigate the scalability of both neural bug detectors. Again, we have strengthened the baseline by adding extra edges to its program graphs [Allamanis et al. 2018]. We fix the context for each method to be 2, which provides the largest variation in graph size. Figure 15 depicts the precision, recall, and F1 score for GINN and the baseline under top-1 prediction mode (top-3 and top-5 yield similar results). Even though GINN was beaten by GNN on few subsets of small graphs, it consistently outperforms the baseline as the size of graphs increases. Overall, we conclude bug detectors built out of GINN are more scalable.

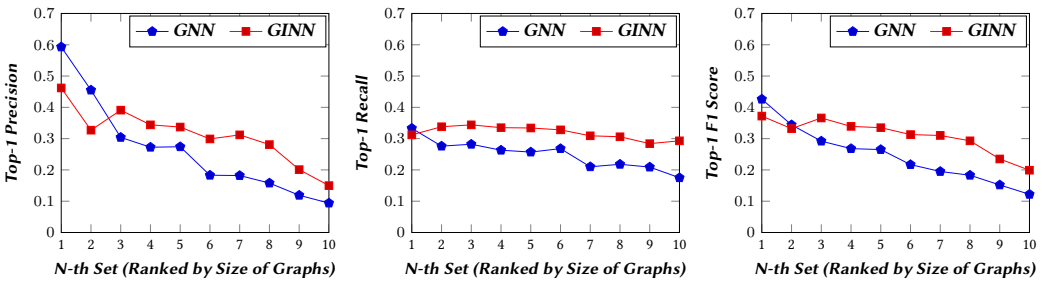

Fig. 15. Investigating the scalability of baseline and GINN.

Catching Bugs in the Wild. We conduct another study to test if the GINN-based bug detector can discover new bugs in the real world. We set up both GINN-based bug detector (previously

trained on 64 projects and configured with top-1 prediction mode) and Facebook Infer [Berdine et al. 2006; Calcagno et al. 2015], arguably the state-of-the-art static analysis tool for Java code, to scan the codebase of 20 projects on GitHub that are highly starred and actively maintained. Note that although Infer is sound by design (based on abstract interpretation [Cousot and Cousot 1977]), it makes engineering compromises for the sake of usability, thus triggering both false positives and false negatives in practice. On the other hand, GINN-powered bug detector is by nature an unsound tool, therefore we only study the utility of both tools in engineering terms.

After manually inspecting all alarms produced by both tools, We confirmed 38 bugs out of 102 alarms GINN-based bug detector raises, which rounds up to a precision of 37.3%. Note that even though programs in the wild exhibit a far higher ratio between correct and buggy methods than GINN's test set, GINN is more precise when catching bugs in the wild. The reason is that the test set is filled exclusively with methods that are hard to distinguish given the way it is assembled. When GINN is tested in the wild, it is still such methods lying close to the decision boundary that account for most of the false positives since the remaining correct methods, albeit in large number, are clearly separable from the real bugs. As a concrete piece of evidence, we find many methods GINN flagged are syntactically similar to some real bug wes confirmed (with respect to the tree-edit distance between their ASTs) as GINN never mis-predicts simple methods like getters or setters.

Compared to GINN's performance, Facebook Infer emitted 129 alarms out of which 34 are confirmed (*i.e.* precision = 26.3%). To understand why our bug detector is more precise, we make several important observations about the behavior of both tools through our manual inspection. All examples are included in Appendix E. First, Infer in general raised more alarms that are provably false (*e.g.* Figure 16, 18, and 20) and missed more bugs that are demonstrably real (*e.g.* Figure 22 and 23, both of which have been fixed after our reporting) than GINN-based bug detector. Second, Infer raised many alarms due to its confusion of some common APIs (*e.g.* Figure 24 and 26). Third, regarding path-sensitivity, there's still much room for Infer to improve. For example, the path along which the bug is reported by Infer (Figure 28) is clearly unsatisfiable. Specifically, the branch (Line 112) that triggers the null pointer exception is guarded against the same condition under which the supposed null pointer is in fact instantiated properly (Line 100). We have manually verified that no code in between changes the evaluation of the path condition at Line 111. Unlike Infer, GINN-based bug detector searches for bug patterns it learned from the training data. Because a large number of the false alarms Infer reports clearly deviate from the norm exhibited in the training data, GINN-based bug detector is better at suppressing them, resulted in a higher precision.

We also look into wrong predictions our bug detector made, and find that it tends to struggle when a bug pattern is not local — the dereference is quite distant from where the null pointer is assigned. We conjecture that the cause is the signal networks receive gets weaker when a scattered bug pattern is overwhelmed by irrelevant statements. To address this issue, we apply the slicing technique [Weiser 1981] and obtain encouraging results in a preliminary study. By simply slicing the graph of each tested method, the same model (without retraining) can now detect 39 bugs out of 93 alarms, a moderate improvement over the previous result. We plan to test this idea out more systematically and at a larger scale.

We reported the confirmed bugs (*i.e.* 38 in total mentioned above) that GINN-based bug detector found, out of which 11 are fixed and another 12 are confirmed (fixes pending). Reasons for unconfirmed bugs are: (1) no response from developers; (2) requiring actual inputs that trigger the reported bugs; or (3) third party code developers did not write themselves. Our findings show that GINN-based bug detector is capable of catching new bugs in real-world Java projects.

4.6 Discussion

We summarize the lessons learned from our extensive evaluation. First and foremost, the interval-

based graph abstraction method has shown to improve the generalization of various graph-based models (*e.g.* standard GNN, GGNN, RNN Sandwich) across all three downstream PL tasks. In each task, the GINN-based model outperforms a GNN-based model — on a dataset that the GNN-based model achieves state-of-the-art results — by around 10% in a metric adopted by the GNN-based model. Second, our evaluation reveals the scalability issues existing graph models suffer from — manifested in the significant performance drop against the increasing size of graphs — and suggests an effective solution built out of the graph abstraction method. In fact, the overall accuracy improvement mentioned earlier is in large part attributed to GINN's efficient handling of the large graphs. Finally, our evaluation also shows model-based static analyzers can be a promising alternative to the classical static analysis tools for catching null pointer dereference bugs. Even though our results are still preliminary for drawing a general conclusion on the comparison between statistical-based and logic-based tools, it shows our neural bug detector is not only notably more precise than Infer but also capable of catching new bugs on many popular codebases on GitHub.

5 RELATED WORK

In this section, we survey prior works on learning models of source code. Hindle et al. [2012] pioneer the field of machine learning for source code. In particular, Hindle et al. find that programs that people write are mostly simple and rather repetitive, and thus they are amenable to statistical language models. Their finding is based on the evidence that the n-gram language model captures regularities in software as well as it does in English.

Later, many works propose to elevate the learning from the level of token sequences [Gupta et al. 2017; Hindle et al. 2012; Nguyen et al. 2013; Pu et al. 2016] to that of abstract syntax trees [Alon et al. 2019a,b; Maddison and Tarlow 2014] in an attempt to capture the structure properties exhibited in source code. Notably, Alon et al. [2019b] present a method for function name prediction. Specifically, it decomposes a program to a collection of paths in its abstract syntax tree, and learns the atomic representation of each path simultaneously with learning how to aggregate a set of them.

Nowadays, graph neural networks have become undoubtedly the most popular deep model of source code. Since the introduction of graph neural networks to the programming domain [Allamanis et al. 2018; Li et al. 2016], they have been applied to a variety of PL tasks, such as program summarization [Fernandes et al. 2019], bug localization and fixing [Dinella et al. 2020], and type inference [Wei et al. 2020]. Apart from being thoroughly studied and constantly improved in the machine learning field, GNN's capability of encoding semantic structure of code through a graph representation is a primary contributor to their success. While our work also builds upon GNN, we offer a fundamentally different perspective which no prior works have explored. That is program abstraction helps machine learning models to capture the essence of the semantics programs denote, and in turn facilitating the execution of downstream tasks in a precise and efficient manner.

In parallel to all the aforementioned works, a separate line of works has emerged recently that use program executions (*i.e.* dynamic models) [Wang 2019; Wang et al. 2018; Wang and Su 2020] rather than source code (*i.e.* static models) for learning program representations. Their argument is that source code alone may not be sufficient for models to capture the semantic program properties. Wang and Christodorescu [2019] show simple, natural transformations, albeit semantically-preserving, can heavily influence the predictions of models learned from source code. In contrast, executions that offer direct, precise, and canonicalized representations of the program behavior help models to generalize beyond syntactic features. On the flip side, dynamic models are likely to suffer from the low quality of training data since high-coverage executions are hard to obtain. Wang and Su [2020] address this issue by blending both source code and program executions. Our work is set to improve static models via program abstraction, therefore we don't consider runtime information as a feature dimension, which can be an interesting future direction to explore.

6 CONCLUSION

In this paper, we present a new methodology of learning models of source code. In particular, we argue by learning from abstractions of source code, models have an easier time to distill the key features for program representation, thus better serving the downstream tasks. At a technical level, we develop a principled interval-based abstraction method that directly applies to control flow graph. This graph abstraction method translates to a loop-based program abstraction at the source code level, which in essence makes models focus exclusively on looping construct for learning feature representations of source code. Through a comprehensive evaluation, we show our approach significantly outperforms the state-of-the-art models in two highly popular PL tasks: variable misuse and method name prediction. We also evaluate our approach in catching null pointer dereference bugs in Java programs. Results again show GINN-based bug detector not only beats the GNN-based bug detector but also yields a lower false positive ratio than Facebook Infer when deployed to catch null pointer deference bugs in the wild. We reported 38 bugs found by GINN to developers, among which 11 have been fixed and 12 have been confirmed (fixing pending).

GINN is a general, powerful deep neural network that we believe is readily available to tackle a wide range of problems in program analysis, program comprehension, and developer productivity.

ACKNOWLEDGMENTS

We thank the reviewers for their insightful comments. This work was supported by the National Key R&D Program (No. 2017YFA0700604), the National Natural Science Foundation of China under Grant No.62032010, and partially supported by Postgraduate Research & Practice Innovation Program of Jiangsu Province.

REFERENCES

Miltiadis Allamanis. 2019. The Adverse Effects of Code Duplication in Machine Learning Models of Code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2019)*.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. *International Conference on Learning Representations* (2018).

Frances E. Allen. 1970. Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization*.

Uri Alon, Omer Levy, and Eran Yahav. 2019a. code2seq: Generating sequences from structured representations of code. *International Conference on Learning Representations* (2019).

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019b. Code2Vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* POPL (2019).

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. *International Conference on Learning Representations* (2015).

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2006. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects (FMCO'05)*.

Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods*, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi (Eds.).

Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

P. Cousot and R. Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Jacob Devlin, Rabih Zbib, Zhongqiang Huang, Thomas Lamar, Richard Schwartz, and John Makhoul. 2014. Fast and robust neural network joint models for statistical machine translation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs. *International Conference on Learning Representations* (2020).

Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured Neural Summarization. *International Conference on Learning Representations* (2019).

Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70 (ICML'17)*.

Marco Gori, Gabriele Monfardini, and Franco Scarselli. 2005. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Thirty-First AAAI Conference on Artificial Intelligence*.

Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global Relational Models of Source Code. *International Conference on Learning Representations* (2020).

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-term Memory. *Neural computation* (1997).

L. Jiang, G. Misherghi, Z. Su, and S. Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *29th International Conference on Software Engineering (ICSE'07)*.

René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 437–440.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2016. Gated graph sequence neural networks. *International Conference on Learning Representations* (2016).

Chris Maddison and Daniel Tarlow. 2014. Structured generative models of natural source code. In *International Conference on Machine Learning*.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and Their Compositionality. In *Neural Information Processing Systems (NIPS)*.

Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2013. A Statistical Semantic Language Model for Source Code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*.

Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* (2015).

Michael Pradel and Koushik Sen. 2018. Deepbugs: a learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* OOPSLA (2018).

Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. 2016. Sk_P: A Neural Program Corrector for MOOCs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)*.

Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic Model for Code with Decision Trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*.

Ripon Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul Prasad. 2018. Bugs. jar: a large-scale, diverse dataset of real-world java bugs. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 10–13.

Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*.

Daniel Svozil, Vladimir Kvasnicka, and Jiri Pospichal. 1997. Introduction to multi-layer feed-forward neural networks. *Chemometrics and intelligent laboratory systems* (1997).

David A Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. Bugswarm: mining and continuously growing a dataset of reproducible failures and fixes. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 339–349.

Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh singh. 2019. Neural Program Repair by Jointly Learning to Localize and Repair. *International Conference on Learning Representations* (2019).

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer Networks. In *Advances in Neural Information Processing Systems 28*.

Ke Wang. 2019. Learning Scalable and Precise Representation of Program Semantics. *arXiv preprint arXiv:1905.05251* (2019).

Ke Wang and Mihai Christodorescu. 2019. COSET: A Benchmark for Evaluating Neural Program Embeddings. *arXiv preprint arXiv:1905.11445* (2019).

Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Dynamic Neural Program Embedding for Program Repair. *International Conference on Learning Representations* (2018).

Ke Wang and Zhendong Su. 2020. Blended, Precise Semantic Program Embeddings. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20)*.

Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE.

Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *International Conference on Learning Representations*.

Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*.

Xin Ye, Razvan Bunescu, and Chang Liu. 2014. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 689–699.