

---

# A Survey of Graph Neural Networks for Programming Languages

---

Hemil Desai<sup>\*1</sup> Sripath Mishra<sup>\*1</sup> Justin Yi<sup>\*1</sup>

## 1. Background

```
var clazz=classTypes["Root"].Single() as JsonCodeGenerator.ClassType;  
Assert.NotNull(clazz);  
var first=classTypes["RecClass"].Single() as JsonCodeGenerator.ClassType;  
Assert.NotNull(clazz);  
Assert.Equal("string", first.Properties["Name"].Name);  
Assert.False(clazz.Properties["Name"].IsArray);
```

Figure 1: A snippet of a detected bug in an open-source project (details anonymized). The code has been slightly simplified. Our model detects correctly that the variable used in the highlighted (yellow) slot is incorrect. Instead, `first` should have been placed at the slot.

In this section we will be focusing on representing programs as graphs. We are using the paper (Allamanis et al., 2017) to present the current state of the art procedure to represent programs as graphs for GNN processing. The authors present how to construct graphs from source code and how to scale Gated Graph Neural Networks training to such large graphs. We do this by encoding programs as graphs, in which edges represent syntactic relationships (e.g. “token before/after”) as well as semantic relationships (“variable last used/written here”, “formal parameter for argument is called stream”, etc.). They evaluate their method on two tasks: VARNAMING, in which a network attempts to predict the name of a variable given its usage, and VARMISUSE, in which the network learns to reason about selecting the correct variable that should be used at a given program location. The VARNAMING task, in which given some source code, the “correct” variable name is inferred as a sequence of subtokens. This requires some understanding of how a variable is used, i.e., requires reasoning about lines of code far apart in the source file. The variable misuse prediction task (VARMISUSE), aims to infer which variable should be used in a program location. To illustrate the task, Figure above shows a slightly simplified snippet of a bug their model detected in a popular open-source project. The Code used for experiments are published at <https://github.com/dmitrykazhdan/Representing-Programs-with-Graphs>.

### 1.1. Formulation of task description

In the proposed formulation a source code file is viewed as a sequence of tokens  $t_0 \dots t_N = T$ , in which some tokens  $t_{\lambda_0}, t_{\lambda_1} \dots$  are variables. Furthermore, let  $V_t \subset V$  refer to the set of all type-correct variables in scope at the location

of  $t$ , i.e., those variables that can be used at  $t$  without raising a compiler error. They call the location  $t$  where they want to predict the correct variable usage a slot. A separate task is defined for each slot  $t_\lambda$ : Given  $t_0 \dots t_{\lambda-1}$  and  $t_{\lambda+1}, \dots, t_N$ , correctly select  $t_\lambda$  from  $V_{t_\lambda}$ . For training and evaluation purposes, a correct solution is one that simply matches the ground truth, but note that in practice, several possible assignments could be considered correct (i.e., when several variables refer to the same value in memory).

### 1.2. Gated Graph Neural Networks

A graph  $G = (V, E, X)$  is composed of a set of nodes  $V$ , node features  $X$ , and a list of directed edge sets  $E = (E_1, \dots, E_K)$  where  $K$  is the number of edge types. Each  $v$  is annotated with a real-valued vector  $x(v) \in \mathbb{R}^D$  representing the features of the node (e.g., the embedding of a string label of that node). Every node  $v$  is associated with a state vector  $h(v)$ , initialized from the node label  $x(v)$ . The sizes of the state vector and feature vector are typically the same. To propagate information throughout the graph, “messages” of type  $k$  are sent from each  $v$  to its neighbors, where each message is computed from its current state vector as  $m_k^{(v)} = f_k(h^{(v)})$ . Here,  $f_k$  can be an arbitrary function; A linear layer is chosen in this case. By computing messages for all graph edges at the same time, all states can be updated at the same time. In particular, a new state for a node  $v$  is computed by aggregating all incoming messages as  $m_k^{(u)} = g(m_k^{(u)})$  — there is an edge of type  $k$  from  $u$  to  $v$ .  $g$  is an aggregation function, on  $k$  as element wise summation. Given the aggregated message  $m^{(v)}$  and the current state vector  $h(v)$  of node  $v$ , the state of the next time step  $h^{(v)}$  is computed as  $h^{(v)} = \text{GRU}(m^{(v)}, h^{(v)})$ , where GRU is the recurrent cell function of gated recurrent unit (GRU). The dynamics defined by the above equations are repeated for a fixed number of time steps. Finally, the state vectors from the last time step are the node representation.

### 1.3. Program Graphs

The backbone of a program graph is the program’s abstract syntax tree (AST). Add NextToken edges connecting each syntax token to its successor. Let a token  $v$ , let  $D^R(v)$  be the set of syntax tokens at which the variable could have been used last. Similarly, let  $D^W(v)$  be the set of syntax

tokens at which the variable was last written to. Graph to chain all uses of the same variable using LastLexicalUse edges. Return tokens are connected to the method declaration using ReturnsTo edges. While, arguments in method calls are connected to the formal parameters that they are matched to with FormalArgName edges. The authors introduce their respective backwards edges (transposing the adjacency matrix), doubling the number of edges and edge types

#### 1.4. Leveraging Variable Type Information

The authors assume that a statically typed language and that the source code can be compiled, and thus each variable has a (known) type  $\tau(v)$ . To use it, a learnable embedding function  $r(\tau)$  for known types and additionally define an “UNKTYPE” for all unknown/unrepresented types. Rich type hierarchy that is available in many object-oriented languages is also taken into account. They map a variable’s type  $\tau(v)$  to the set of its supertypes, i.e.  $\tau^*(v) = \tau : \tau(v)$  implements type  $\tau \cup \tau(v)$ . Then the type representation  $r^*(v)$  of a variable  $v$  as the element-wise maximum of  $r(\tau) : \tau \in \tau^*(v)$  is computed. the maximum is chosen as it is a natural pooling operation for representing partial ordering relations (such as type lattices). Using all types in  $\tau^*(v)$  allows us to generalize to unseen types that implement common supertypes or interface. These types implement a common interface (IList) and share common characteristics.

#### 1.5. Initial Node Representation

The proposed method splits the name of a node into subtokens (e.g. classTypes will be split into two subtokens class and types) on camelCase and pascal case. Then average is taken of the embeddings of all subtokens to retrieve an embedding for the node name. Finally, the learned type representation  $r^*(v)$  is concatenated.

#### 1.6. Programs Graphs for VarNaming

Given a program and an existing variable  $v$ , They build a program graph as discussed above and then replace the variable name in all corresponding variable tokens by a special  $\text{;SLOT}_i$  token. The initial node labels computed is used as the concatenation of learnable token embeddings and type embeddings as discussed above, GGNN propagation is repeated for 8 time steps and then variable usage representation by averaging the representations for all  $\text{;SLOT}_i$  tokens is computed. This representation is then used as the initial state of a one-layer GRU, which predicts the target name as a sequence of subtoken.

#### 1.7. Programs Graphs for VarMisuse

To model VARMISUSE with program graphs, the graph is modified. First, to compute a context representation  $c(t)$  for a slot  $t$  where the used variable is predicted, a new node  $v_{\langle \text{SLOT} \rangle}$  at the position of  $t$  is inserted, corresponding to a “hole” at this point, and connection is made to the remaining graph using all applicable edges that do not depend on the chosen variable at the slot (i.e., everything but LastUse, LastWrite, and LastLexicalUse edges). Then, to compute the usage representation  $u(t, v)$  of each candidate variable  $v$  at the target slot, a “candidate” node  $v_{t,v}$  for all  $v$  in  $V_t$  is inserted, and connected to the graph by inserting the LastUse, LastWrite and LastLexicalUse edges that would be used if the variable were to be used at this slot. Each of these candidate nodes represents the speculative placement of the variable within the scope. Using the initial node representations, concatenated with an extra bit that is set to one for the candidate nodes  $v_{t,v}$ , The GGNN propagation is repeated for 8 time steps. The context and usage representation are then the final node states of the nodes, i.e.,  $c(t) = h(v_{\langle \text{SLOT} \rangle})$  and  $u(t,v) = h(v_{t,v})$ . Finally, the correct variable usage at the location is computed as  $\arg \max_v c(t)^T u(t, v)$ . The model is trained using maximum likelihood.

### 2. Program synthesis

#### 2.1. Introduction

A general formulation of program synthesis called syntax-guided synthesis (SyGuS) seeks to synthesize a program that follows a given grammar and satisfies a given logical specification. Both the logical specification and the grammar have complex structures and can vary from task to task, posing significant challenges for learning across different tasks. Moreover, supervision is often unavailable for domain-specific synthesis tasks. To address these challenges, In this paper we survey a metalearning framework that learns a transferable policy using only weak supervision. The program synthesizer takes as input a logical formula  $\varphi$  and a grammar  $G$ , and produces as output a program in  $G$  that satisfies  $\varphi$ . In this formulation,  $\varphi$  constitutes a semantic specification that describes the desired functional requirements, and  $G$  is a syntactic specification that constrains the space of possible programs. They assume a fixed grammar (i.e., syntactic specification  $G$ ) across tasks.

Learning to understand and generate programs is an important building block for procedural artificial intelligence and more intelligent software engineering tools. It is also an interesting task in the research of structured prediction methods: while imbued with formal semantics and strict syntactic rules, natural source code carries aspects of natural languages, since it acts as a means of communicating intent among developers. Early works in the area have shown

that approaches from natural language processing can be applied successfully to source code, whereas the programming languages community has had successes in focusing exclusively on formal semantics. More recently, methods handling both modalities (i.e., the formal and natural language aspects) have shown successes on important software engineering tasks and semantic parsing. However, current generative models of source code mostly focus on only one of these modalities at a time. For example, program synthesis tools based on enumeration and deduction are successful at generating programs that satisfy some (usually incomplete) formal specification but are often obviously wrong on manual inspection, as they cannot distinguish unlikely from likely, “natural” programs. On the other hand, learned code models have succeeded in generating realistic-looking programs. However, these programs often fail to be semantically relevant, because variables are not used consistently.

Along with the aforementioned problems, many modern APIs have an incredibly steep learning curve, due to their hundreds of functions handling many arguments, obscure documentation, and frequently changing semantics. For APIs that perform data transformations, novices can often provide an I/O example demonstrating the desired transformation, but may be stuck on how to translate it to the API.

## 2.2. Related work

A large body of work has been dedicated to solving program synthesis problems. Numerous systems have been developed targeting a variety of domains such as string processing [(Gulwani, 2011); (Parisotto et al., 2017)], data wrangling [(Feng et al., 2018)], data processing [(Smith & Albarghouti, 2016)], syntax transformations [(Rolim et al., 2017)], database queries [(Yaghmazadeh et al., 2017)] and bit-vector manipulations [(Jha et al., 2010)]. One paper attempts to categorise these works at a coarse level according to the high-level synthesis strategy used in their respective systems. Then summarise how strategy of using neural-backed generators compares and relates to these strategies.

CEGIS [(Solar Lezama, 2008)] is a general framework for program synthesis that synthesizes programs satisfying a specification  $\phi$ . The basic algorithm involves two components: a synthesizer and verifier, where the synthesizer generates candidate programs and the verifier confirms whether the candidate is correct. The synthesizer also takes the space of possible candidates as an input, either explicitly or implicitly. This space may be defined in multiple ways, for example using syntactic definitions of valid programs [(Alur et al., 2013)]. A large fraction of techniques, including ours, fall under this general CEGIS framework, with differing strategies for producing program candidates, which are broadly classified below.

Source code generation has been studied in a wide range of different settings [(Allamanis et al., 2018)]. Early works approach the task by generating code as sequences of tokens [(Hindle et al., 2016)], whereas newer methods have focused on leveraging the known target grammar and generate code as trees [(Maddison & Tarlow, 2014)]. While modern models succeed at generating “natural-looking” programs, they often fail to respect simple semantic rules. For example, variables are often used without initialization or written several times without being read inbetween.

**Neural program synthesis** Several recent works have used neural networks to accelerate the discovery of desired programs. These include DeepCoder [(Balog et al., 2017)], Bayou [(Murali et al., 2018)], RobustFill [(Devlin et al., 2017)], Differentiable FORTH [(Bošnjak et al., 2017)], neuro-symbolic program synthesis [(Bunel et al., 2018)], neural-guided deductive search [(Kalyan et al., 2018b)], learning context-free parsers [(Chen et al., 2018)], and learning program invariants [(Si et al., 2018)]. The syntactic specification in these approaches is fixed by defining a domain-specific language upfront. Also, with the exception of (Si et al., 2018), the semantic specification takes the form of input-output examples. Broadly, these works have difficulty with symbolic constraints, and are primarily concerned with avoiding over fitting, coping with few examples, and tolerating noisy examples.

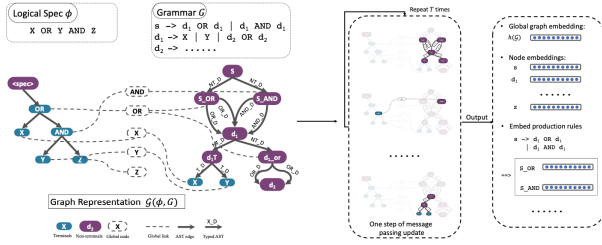
One class of machine learning approaches predict programs directly using neural networks [(Parisotto et al., 2017)] while another employs neural networks to guide the symbolic techniques above [(Kalyan et al., 2018a)]. In both cases, the models involved take the input-output example directly and make predictions accordingly. However the domains tackled so far are simpler; (Balog et al., 2017); (Devlin et al., 2017); (Kalyan et al., 2018a) target string-processing and lists of bounded integers where machine learning models such as cross-correlational networks, LSTMs with attention are applicable. In contrast, these models cannot target Data Frames due to the issues. Additionally, since Data Frames are not of a fixed size, the CNNs used to encode fixed-size grid-based examples in (Bunel et al., 2018) are also not applicable.

**Graph Neural Networks** (Dai et al., 2018) [2017]; (Kool et al., 2019). [2019] use graph-neural networks to solve difficult combinatorial optimization problems such as the Travelling Salesman Problem (TSP). Although their use-case is different, the adopted approach is similar in spirit to the use of graph neural networks in AutoPandas. Their network iteratively constructs the solution by selecting and adding a node to the solution in every iteration. This updated graph is then used as input in the next iteration.

### 2.3. SyGus

A framework that is general in that it makes few assumptions on specific grammars or constraints, and has meta-learning capability that can be utilized in solving unseen tasks more efficiently.

#### 2.3.1. PROBLEM FORMULATION



The Syntax-Guided Synthesis (SyGuS) problem is to synthesize a function  $f$  that satisfies two kinds of constraints: 1) A syntactic constraint specified by a context-free grammar (CFG)  $G$ , and 2) A semantic constraint specified by a formula  $\phi$  built from symbols in a background theory  $T$  along with  $f$ . The authors investigate how to efficiently synthesize the function  $f$ . Specifically, given a dataset of  $N$  tasks  $D = (\phi_i, G_i)_{i=1}^N$ , The following two tasks are addressed: 1) learning an algorithm  $A_\theta : (\phi, G) \rightarrow f$  parameterized by  $\theta$  that can find the function  $f_i$  for  $(\phi_i, G_i) \in D$ ; 2) given a new task set  $D'$ , adapt the above learned algorithm  $A_\theta$  and execute it on new tasks in  $D'$ . This setting poses two difficulties in learning. First, the ground truth target function  $f$  is not readily available, making it difficult to formulate as a supervised learning problem. Second, the constraint  $\phi$  is typically verified using an SAT or SMT solver, and this solver in turns expects the generated  $f$  to be complete. This means the weak supervision signal will only be given after the entire program is generated. Thus, it is natural to formulate  $A_\theta$  as a reinforcement learning algorithm. Since each instance  $(\phi_i, G_i) \in D$  is an independent task with different syntactic and semantic constraints, the key to success is the design of such meta-learner.

#### 2.3.2. FORMAL DEFINITION

**semantic spec  $\phi$**  The spec itself is a program written using some grammar. The grammar used in spec  $\phi$  is different from the grammar  $G$  that specifies the syntax of the output program. However, in many practical cases the tokens (i.e., the dictionary of terminal symbols) may be shared across the input spec and the output program.

**CFG  $G$**  A context free grammar (CFG) is defined as  $G = \langle V, E, R, s \rangle$ . Here  $V$  denotes the non-terminal tokens, while  $E$  represents the terminal tokens.  $s$  is a special token

that denotes the start of the language, and the language is generated according to the production rules defined in  $R$ . For a given non-terminal, the associated production rules can be written as  $\alpha \rightarrow \beta_1 | \beta_2 | \dots | \beta_{n_\alpha}$ , where  $n_\alpha$  is the branching factor for non-terminal  $\alpha \in V$ , and  $\beta_i = u_1 u_2 \dots u_{|\beta_i|} \in (V \cup E)$ . Each production rule  $\alpha \rightarrow \beta_1 \in R$  represents a way of expanding the grammar tree, by attaching nodes  $u_1 u_2 \dots u_{|\beta_i|}$  to node  $\alpha$ .

**Output function  $f$**  The output is a program in the language generated by  $G$ . A valid output  $f$  must satisfy both the syntactic constraints specified by  $G$  and the semantic constraints specified by  $\phi$ .

#### 2.3.3. TASK REPRESENTATION

Semantic spec program  $\phi$  and the CFG  $G$  have rich structural information. To construct the graph, the abstract syntax tree (AST) is built first for the semantic spec program  $\phi$ , according to its own grammar (typically different from the output grammar  $G$ ). To represent the grammar  $G$ , each symbol in  $V \rightarrow E$  with a node representation. Furthermore, for a non-terminal  $\alpha$  and its corresponding production rules  $\alpha \rightarrow \beta_1 | \beta_2 | \dots | \beta_{n_\alpha}$ , Additional nodes  $\alpha_i$  are created for each substitute  $\beta_i$ . The purpose is to enable grammar adaptive program generation.

As a simplification, all nodes  $\alpha_i$  is merged representing  $\beta_i$  that is a single terminal token into one node. Finally, the global nodes for shared tokens in  $E$  are created to link together the shared variable and operator nodes. This enables information exchange between the syntactic and semantics specifications. To encode the joint graph  $G(\phi, G)$ , graph neural networks is used to get the vector representation for each node in the graph. Specifically, for each node  $v \in G$ , the following parameterization for one step of message passing style update:

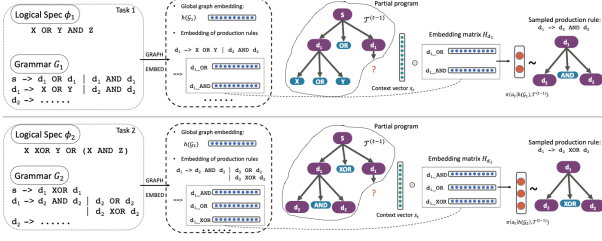
$$h_v^{(t+1)} = \text{Aggregate}(F(h_u^t, e_{u,v})_{u \in N(v)})$$

Lastly,  $h_{v \in G}^t$ ,  $h_v^t \in \mathbb{R}^d$  are the set of node embeddings. Here  $N(v)$  is the set of neighbor nodes of  $v$ , and  $e_{u,v}$  denotes the type of edge that links the node  $u$  and  $v$ .  $F$  is parameterized as,  $F(h^t, e) = \sigma(W_t^{eT} h^t)$  where different matrices  $W \in \mathbb{R}^{d \times d}$  are used for different edge types and different propagation steps  $t$ . All the node embeddings are aggregated to get the global graph embedding  $h(G)$ . The embedding matrix for each non-terminal node is computed. Specifically, given node  $\alpha$ , the embedding matrix is  $H_\alpha \in \mathbb{R}^{n_\alpha \times d}$ , where the  $i$ th row of  $H_\alpha^{(i)}$  is the embedding of node  $\alpha_i$  that corresponds to substitution  $\beta_i$ .

#### 2.3.4. GRAMMAR ADAPTIVE POLICY NETWORK

The key idea is to make the policy parameterized by decision embedding, rather than a fixed set of parameters.





The embedding matrix  $H_\alpha \in R^{n_\alpha \times d}$  is used to perform decision for this time step. This allows us to build policy network in an auto-regressive way. Specifically, the policy  $\pi(f|\phi, G)$  can be parameterized as:  $\pi(f|\phi, G) = \prod_{t=1}^{|f|} \pi(a_t|h(G), T^{t-1})$ , where  $T^{t-1} = \alpha_1 \dots \alpha_{t-1}$  denotes the partial tree. Here the probability of each action (in other words, each tree expansion decision) is defined as  $\pi(a_t|h(G), T^{t-1}) \propto \exp(H_\alpha^{(i)T} s_t)$ , where  $s_t \in R$  is the context vector that captures the state of  $h(G)$  and  $T^{t-1}$ .  $s_t$  is tracked by a LSTM decoder whose hidden state is updated by the embedding of the chosen action  $h_{\text{alphanat}}$ . The initial state  $s_0$  is obtained by passing graph embedding  $h(G)$  through a dense layer with matching size.

### 2.3.5. SOLVING VIA REINFORCED LEARNING

let  $\theta$  denote the parameters of graph embedding and adaptive policy network. For a given pair of instances  $(\phi, G)$ , a learned policy  $\pi_\theta(f|\phi, G)$  parameterized by  $\theta$  that generates  $f$  such that  $\phi \equiv f$ .

**Reward design** The RL episode starts by accepting the representation of tuple  $\langle \phi, G \rangle$  as initial observation. During the episode, the model executes a sequence of actions to expand non-terminals in  $f$ , and finishes the episode when  $f$  is complete. Upon finishing, the SAT solver is invoked and will return a binary flag indicating whether  $f$  satisfies  $\phi$  or not. To smooth the reward: for each specification  $\phi$  the solution maintains a test case buffer  $B_\phi$  that stores all input examples observed so far. Each time the SAT solver is invoked for  $\phi$ , if  $f$  passes then a full reward of 1 is given, otherwise the solver will generate a counterexample  $b$  besides the binary flag. Then sample interpolated examples around  $b$  which is denoted in the set as  $B_b$ . Then the reward is given as the fractions of examples in  $B_\phi$  and  $B_b$  where  $f$  has the equivalent output as  $\phi$

$$r = \sum_{b \in B_\phi \cup B_b} [f(b) \equiv \phi(b)] / |B_\phi \cup B_b|$$

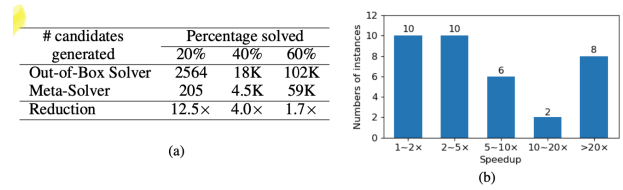
At the end of the episode, the buffer is updated as  $B_\phi \leftarrow B_\phi \cup B_b$  for next time usage. The authors utilize the Advantage Actor-Critic (A2C) for model training. Given a training set  $D$ , a minibatch of instances are sampled from  $D$  for each epoch. For each instance  $(\phi_i, G_i)$ , the model performs a complete rollout using policy  $\pi_\theta(f|\phi_i, G_i)$ . The

actor-critic method computes the gradients w.r.t to  $\theta$  of each instance as

$$d\theta \leftarrow \sum_{t=1}^{|f|} \nabla_\theta \log \pi(a_t|h(G), T^{(t)}) (\gamma^{|f|-t} r - V(s_t; w)),$$

where  $\gamma$  denotes the discounting factor and  $V(s_t; w)$  is a state value estimator parameterized by  $w$ . In implementation, this is modeled as a standard MLP with scalar output. It is learned to fit the expected return, i.e.,  $\min_w E \sum_{t=1}^{|f|} \gamma^{-t} r - V(s_t; w)$ . Gradients obtained from each instance are averaged over the minibatch before applying to the parameter.

### 2.3.6. RESULTS



Performance improvement with meta-learning. (a) Accumulated number of candidates generated in order to solve 20 percent, 40 percent, and 60 percent of the testing tasks; and (b) speedup distribution over individual instances.

## 2.4. ExprGen

### 2.4.1. BACKGROUND

The most general form of the code generation task is to produce a (partial) program in a programming language given some context information  $c$ . This context information can be natural language (as in, e.g., semantic parsing), input-output examples (e.g., inductive program synthesis), partial program sketches. The key idea is to construct the AST sequentially, by expanding one node at a time using production rules from the underlying programming language grammar. Then, the probability of generating a given AST  $a$  given some context  $c$  is

$$p(a|c) = \prod_t p(a_t|c, a_{<t})$$

where  $a_t$  is the production choice at step  $t$  and  $a_{<t}$  the partial syntax tree generated before step  $t$ .

**Code Generation as Hole Completion** the authors introduce the ExprGen task of filling in code within a hole of an otherwise existing program. They assume information about the following code as well and aim to generate whole expressions rather than single tokens. They restrict themselves to expressions that have Boolean, arithmetic or string type, or arrays of such types.

### 2.4.2. GRAPH DECODING FOR SOURCE CODE

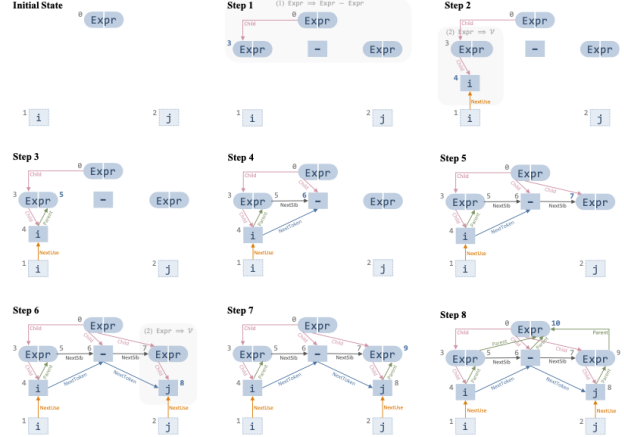
Tasks include to encode the code context  $c, v_1, \dots, v_l$  and to construct a model that can learn  $p(a_t | c, a_{<t})$  well. Both these encoders yield a distributed vector representation for the overall context, representations  $h_{t_1}, \dots, h_{t_T}$  for all tokens in the context, and separate representations for each of the in-scope variables  $v_1, \dots, v_l$ , summarizing how each variable is used in the context. The decoder model follows the grammar-driven AST generation strategy is used. A variation of attribute grammars (Knuth, 1967) from compiler theory to derive the structure of this graph. Each node in the AST is associated with two fresh nodes representing inherited resp. synthesized information (or attributes). Inherited information is derived from the context and parts of the AST that are already generated, whereas synthesized information can be viewed as a “summary” of a subtree. In classical compiler theory, inherited attributes usually contain information such as declared variables and their types (to allow the compiler to check that only declared variables are used), whereas synthesized attributes carry information about a subtree “to the right” (e.g., which variables have been declared). Traditionally, to implement this, the language grammar has to be extended with explicit rules for deriving and synthesizing attributes. Attributes are represented by distributed vector representations and train neural networks to learn how to compute attributes. A deterministic procedure that turns a partial AST  $a_{<t}$  into a graph by adding additional edges that encode attribute relationships, and a graph neural network that learns from this graph.

#### Algorithm 2 Pseudocode for ComputeEdge

**Input:** Partial AST  $a$ , node  $v$

- 1: Edge set  $\mathcal{E} \leftarrow \emptyset$
- 2: **if**  $v$  is inherited **then**
- 3:  $\mathcal{E} \leftarrow \mathcal{E} \cup \{\langle \text{parent}(a, v), \text{Child}, v \rangle\}$
- 4: **if**  $v$  is terminal node **then**
- 5:  $\mathcal{E} \leftarrow \mathcal{E} \cup \{\langle \text{lastToken}(a, v), \text{NextToken}, v \rangle\}$
- 6: **if**  $v$  is variable **then**
- 7:  $\mathcal{E} \leftarrow \mathcal{E} \cup \{\langle \text{lastUse}(a, v), \text{NextUse}, v \rangle\}$
- 8: **if**  $v$  is not first child **then**
- 9:  $\mathcal{E} \leftarrow \mathcal{E} \cup \{\langle \text{lastSibling}(a, v), \text{NextSib}, v \rangle\}$
- 10: **else**
- 11:  $\mathcal{E} \leftarrow \mathcal{E} \cup \{\langle u, \text{Parent}, v \rangle \mid u \in \text{children}(a, v)\}$
- 12:  $\mathcal{E} \leftarrow \mathcal{E} \cup \{\langle \text{inheritedAttr}(v), \text{InhToSyn}, v \rangle\}$
- 13: **return**  $\mathcal{E}$

**Notion** as graphs where nodes  $u, v, \dots$  are either the AST nodes or their associated attribute nodes, and typed directed edges  $\langle u, \tau, v \rangle \in E$  connect the nodes according to the flow of information in the model. The edge types  $\tau$  represent different syntactic or semantic relations in the information



flow.  $E_v$  is defined as a set of incoming edges into  $v$ . They also use functions like  $\text{parent}(a, v)$  and  $\text{lastSibling}(a, v)$  that look up and return nodes from the AST  $a$  (e.g. resp. the parent node of  $v$  or the preceding AST sibling of  $v$ ) expanded to the set of known variables using the production rule (2):  $\text{Expr} \Rightarrow V$ , choosing  $i$  for the first variable and  $j$  for the second variable.

**Edges in  $a_{<t}$**  - Child (red) edges connect an inherited attribute node to the inherited attributes nodes of its children, as seen in the edges from node 0. Parent (green) edges connect a synthesized attribute node to the synthesized attribute node of its AST parent, as seen in the edges leading to node 10. These are the additional connections used by the R3NN decoder. NextSib (black) edges connect the synthesized attribute node to the inherited attribute node of its next sibling (e.g. from node 5 to node 6). These allow information about the synthesized attribute nodes from a fully generated subtree to flow to the next subtree. NextUse (orange) edges connect the attribute nodes of a variable (since variables are always terminal nodes, the posposed solution does not distinguish inherited from synthesized attributes) to their next use. They just follow the lexical order. NextToken (blue) edges connect a terminal node (a token) to the next token in the program text, for example between nodes 4 and 6. InhToSyn edges (not shown in Fig) connect the inherited attributes nodes to its synthesized attribute nodes. This is not strictly adding any information, but was found helpful with training.

**Attribute Node Representations** To compute the neural attribute representation  $h_v$  of an attribute node  $v$  whose corresponding AST node is labeled with  $l_v$ , incoming edges are obtained and then use the state update function from Gated Graph Neural Networks (GGNN). The attribute representations  $h_{u_i}$  at edge sources  $u_i$  is used to transform them according to the corresponding edge type  $t_i$  using a learned

function  $f_{t_i}$ . They are aggregated (by elementwise summation) and combine them with the learned embedding  $\text{emb}(l_v)$  of the node label  $l_v$  using a function  $g$ :

$$h_v = g(\text{emb}(l_v), \sum_{\langle u_i, t_i, v \rangle \in E_v} f_{t_i}(h_{u_i}))$$

The authors use a single linear layer for  $f_{t_i}$  and implement  $g$  as a gated recurrent unit. Node representations is computed in such an order that all  $h_{u_i}$  appearing on the right are already computed.

**Choosing Productions, Variables & Literal** For a non-terminal node  $v$  with label  $l_v$  and inherited attributes  $h_v$ ,  $\text{pickProduction}(l_v, h_v) = \arg\max \text{P}(\text{rule} \rightarrow l_v, h_v) = \arg\max [e(h_v) + m_{l_v}]$ . (3) Here,  $m_{l_v}$  is a mask vector whose value is 0 for valid productions  $l_v \Rightarrow \dots$  and  $-\infty$  for all other productions. In practice,  $e$  is implemented using a linear layer. Similarly, variables are picked from the set of variables  $V$  in scope using their representations  $h_{v_{var}}$  (initially the representation obtained from the context, and later the attribute representation of the last node in the graph in which they have been used) by using a pointer network. Concretely, to pick a variable at node  $v$ , learnable linear function  $k$  is used and define  $\text{pickVariable}(V, h_v) = \arg\max_{var \in V} \text{P}(\text{var} \rightarrow h_v) = \arg\max_{var \in V} k(h_v, h_{v_{var}})$  (4) Note that since the model always picks a variable from the set of in-scope variables  $V$ , this generation model can never predict an unknown or out-of-scope variable. They combine a small vocabulary  $L$  of common literals observed in the training data and special UNK tokens for each type of literal with another pointer network that can copy one of the tokens  $t_1 \dots t_T$  from the context. Thus, to pick a literal at node  $v$ , and define  $\text{pickLiteral}(V, h_v) = \arg\max_{lit \in L \cup t_1 \dots t_T} \text{P}(\text{lit} \rightarrow h_v)$  (5). This is implemented by learning two functions  $s_L$  and  $s_c$ , such that  $s_L(h_v)$  produces a score for each token from the vocabulary and  $s_c(h_v, h_{t_i})$  computes a score for copying token  $t_i$  from the context. By computing a softmax over all resulting values and normalizing it by summing up entries corresponding to the same constant, they can learn to approximate the desired  $\text{P}(\text{lit} \rightarrow h_v)$

**Training & Training Objective** note that given a ground truth target tree, one can easily augment it with all additional edges. Given that full graph, one can compute a propagation schedule (intuitively, a topological ordering of the nodes in the graph, starting in the root node) that allows to repeatedly apply (2) to obtain representations for all nodes in the graph. By representing a batch of graphs as one large (sparse) graph with many disconnected components,

**Additional Improvements** (3) is extended with an attention mechanism that uses the state  $h_v$  of the currently expanded node  $v$  as a key and the context token representations  $h_{t_1}, \dots, h_{t_T}$  as memories. Experimentally, it was found that extending Eqs. 4, 5 similarly did not improve results, probably due to the fact that they already are highly dependent on

the context information. Additional information for Child edges is provided. To allow this, the setup is changed so that some edge types also require an additional label, which is used when computing the messages sent between different nodes in the graph. Concretely, (2) is extended by considering sets of unlabeled edges  $E_v$  and labeled edges  $E_v^l$ :

$$h_v = g(\text{emb}(l_v), \sum_{\langle u_i, t_i, v \rangle \in E_v} f_{t_i}(h_{u_i}) + \sum_{\langle u_i, t_i, l_i, v \rangle \in E_v^l} f_{t_i}(h_{u_i}, \text{emb}_e(l_i)))$$

Thus for labeled edge types,  $f_{t_i}$  takes two inputs and a learnable embedding for the edge labels. The authors found it useful to label Child with tuples consisting of the chosen production and the index of the child, they would label the edge from 0 to 3 with (2,0), the edge from 0 to 6 with (2,1), etc. the model was extended pickProduction to also take the information about available variables into account. Intuitively, this is useful in cases of productions such as  $\text{Expr} \Rightarrow \text{Expr.Length}$ , which can only be used in a well-typed derivation if an array-typed variable is available. Thus,  $e(h_v)$  is extended from (3) to additionally take the representation of all variables in scope into account, i.e.,  $e(h_v, r(h_{v_{var}} | var \in V))$ , where  $r$  is a max pooling operation.

#### 2.4.3. RESULTS

Model	Test (from seen projects)				Test-only (from unseen projects)			
	Perplexity	Well-Typed	Acc@1	Acc@5	Perplexity	Well-Typed	Acc@1	Acc@5
PHOG <sup>1</sup>	—	—	34.8%	42.9%	—	—	28.0%	37.3%
$\text{Seq} \rightarrow \text{Seq}$	87.48	32.4%	21.8%	28.1%	130.46	23.4%	10.8%	16.8%
$\text{Seq} \rightarrow \text{NAG}$	6.81	53.2%	17.7%	33.7%	8.38	40.4%	8.4%	15.8%
$G \rightarrow \text{Seq}$	93.31	40.9%	27.1%	34.8%	28.48	36.3%	17.2%	25.6%
$G \rightarrow \text{Tree}$	4.37	49.3%	26.8%	48.9%	5.37	41.2%	19.9%	36.8%
$G \rightarrow \text{ASN}$	2.62	78.7%	45.7%	62.0%	3.03	74.7%	32.4%	48.1%
$G \rightarrow \text{Syn}$	2.71	84.9%	50.5%	66.8%	3.48	84.5%	36.0%	52.7%
$G \rightarrow \text{NAG}$	2.56	86.4%	52.3%	69.2%	3.07	84.5%	38.8%	57.0%

Evaluation of encoder and decoder combinations on predicting an expression from code context. PHOG (Bielik et al., 2016) is only conditioned on the tokens on the left of the expression

### 2.5. AutoPandas

#### 2.5.1. TECHNIQUE

**Generators** The authors first formally describe generators. In their setting, a generator  $G$  is a program that, when invoked, outputs values from a space of possible values. their generators  $G$  can contain arbitrary Python code, along with a set of stateful operators that govern the behavior of  $G$  across runs. An example of such an operator is **Select**. **Operators** Apart from **Select**, three other operators are supported, namely (1) **Subset**, (2) **OrderedSubset** and (3) **Sequence**. An informal description of their behavior is provided in Table above. The behavior of the generator across runs can be

Operator	Description
Select(domain)	Returns a single item from domain
Subset(domain)	Returns an unordered subset, without replacement, of the items in domain
OrderedSubset(domain)	Returns an ordered subset, without replacement, of the items in domain
Sequence(len)(domain)	Returns an ordered sequence, with replacement, of the items in domain with a maximum length of len

controlled by changing the semantics of these operators. **Randomized:** The simplest case is for the generator to be randomized. **Exhaustive (Depth-First).** Another option is to have an exhaustive generator which systematically explores all possible execution paths as governed by the constituent operator calls. The operator semantics uses three internal state variables  $t$ ,  $\sigma$  and  $\delta$ . The variable  $t$  keeps track of the number of operator calls made in the current invocation of the generator. The variable  $\sigma$  is a map from the operator call index to the choice to be made by the operator call in the current invocation of the generator.  $\delta$  represents a map from the operator call index to the collection of possible values  $W$  as defined by the operator type and the passed domain  $D$ . The variables  $\sigma$  and  $\delta$  are initialized to empty maps before the first invocation of the generator, but are persisted across the later ones. However  $t$  is reset to zero before every fresh generator invocation. A special operator is introduced called **Op End** that is implicitly invoked at the end of each invocation in the generator. Then the authors now briefly explain the rationale behind all of these variables, **Op End** and the rules themselves. (1) **Op-Extend** - This rule governs the behavior of the operator when it is invoked for the first time (as signified by  $t \notin \text{dom}(\sigma)$ ). The operator returns the first value from  $W$  and records this choice in  $\sigma$ . It also stores  $W$  in  $\delta$  for future use. (2) **Op-Replay** - The hypothesis  $t \in \text{dom}(\sigma)$  signifies that this operator call needs to replay the choice as dictated by  $\sigma(t)$ . (3) **Op-End-1** - This rule captures the first behavior of the special operator **Op End**. It finds the last (deepest) operator call, indexed by  $k$ , that has not exhausted all possibilities, and increments its entry in  $\sigma$ . This is key to enforcing depth-first semantics - a later call explores all possibilities before previous calls do the same. Note that it also pops-off the later entries ( $i > k$ ) from  $\sigma$  and  $\delta$ . This is required as the generator may take an entirely new path based on the new value returned by this operator and may therefore make an entirely new set of operator calls. Together, these two steps maintain the invariant that  $\sigma$  stores the choice to be made by the operators in the current generator run. (4) **Op-End-2** - The final rule covers the case when all operator calls have exhausted all possible values. This makes the special **Op End** operator signal **Generator-Exhausted** after the last invocation of the generator, indicating that they authors have explored all possible executions of the generator.

**Generator-Based Program Synthesis** To build an enumerative synthesis engine  $E$  using generators. The engine con-

$\mathcal{P}(\mathcal{D}) \stackrel{\text{def}}{=} \text{Power-Set of } \mathcal{D}$ $\text{Perms}(x) \stackrel{\text{def}}{=} \text{Set of all permutations of } x$ $W(\text{Op}, \mathcal{D}) \stackrel{\text{def}}{=} \begin{cases} \mathcal{D} \text{ if } \text{Op} = \text{Select} \\ \mathcal{P}(\mathcal{D}) \text{ if } \text{Op} = \text{Subset} \\ \cup \{ \text{Perms}(x) \mid x \in \mathcal{P}(\mathcal{D}) \} \text{ if } \text{Op} = \text{OrderedSubset} \\ \{ (a_1, \dots, a_k) \mid k \leq l, a_i \in \mathcal{D} \} \text{ if } \text{Op} = \text{Sequence}(l) \end{cases}$ $\mathcal{R}(W) \stackrel{\text{def}}{=} \text{Random Element from } W$	$W \stackrel{\text{def}}{=} W(\text{Op}, \mathcal{D})$ $\sigma_k \stackrel{\text{def}}{=} \forall t. ((t < k) \Rightarrow (\sigma_k(t) = \sigma(t))) \wedge ((t \geq k \vee t < 0) \Rightarrow t \notin \text{dom}(\sigma_k))$ $\delta_k \stackrel{\text{def}}{=} \forall t. ((t < k) \Rightarrow (\delta_k(t) = \delta(t))) \wedge ((t \geq k \vee t < 0) \Rightarrow t \notin \text{dom}(\delta_k))$ $W_M \stackrel{\text{def}}{=} \text{Rank}_{(\text{Op}, \mathcal{D})}(W(\text{Op}, \mathcal{D}), \mathcal{D}, C)$
(a) Common Definitions	(b) Common Definitions (continued)
$\frac{}{\text{Op}(\mathcal{D}, C, id) \models \mathcal{R}(W)} \text{Op-RANDOM}$ (c) Operator Semantics - Randomized	
$\frac{t \notin \text{dom}(\sigma) \quad \delta' = \delta[t := W] \quad \sigma' = \sigma[t := 0]}{\text{Op}(\mathcal{D}, C, id), \sigma, \delta, t \models \langle W[0], \sigma', \delta', t+1 \rangle} \text{Op-EXTEND}$ $\frac{t \in \text{dom}(\sigma)}{\text{Op}(\mathcal{D}, C, id), \sigma, \delta, t \models \langle W[\sigma(t)], \sigma, \delta, t+1 \rangle} \text{Op-REPLAY}$ $\frac{\exists k. k \text{ is largest such that } (k \in \text{dom}(\sigma) \wedge \sigma(k) <  \delta(k)  - 1)}{\text{Op}_{\text{End}}, \sigma, \delta, t \models \langle \sigma_k[k := \sigma(k) + 1], \delta_k, t+1 \rangle} \text{Op-END-1}$ $\frac{\exists k. (k \in \text{dom}(\sigma) \wedge \sigma(k) <  \delta(k)  - 1)}{\text{Op}_{\text{End}}, \sigma, \delta, t \models \text{Generator-Exhausted}} \text{Op-END-2}$	$\frac{t \notin \text{dom}(\sigma) \quad \delta' = \delta[t := W_M] \quad \sigma' = \sigma[t := 0]}{\text{Op}(\mathcal{D}, C, id), \sigma, \delta, t \models \langle W_M[0], \sigma', \delta', t+1 \rangle} \text{Op-EXTEND}$ $\frac{t \in \text{dom}(\sigma)}{\text{Op}(\mathcal{D}, C, id), \sigma, \delta, t \models \langle W_M[\sigma(t)], \sigma, \delta, t+1 \rangle} \text{Op-REPLAY}$ $\frac{\exists k. k \text{ is largest such that } (k \in \text{dom}(\sigma) \wedge \sigma(k) <  \delta(k)  - 1)}{\text{Op}_{\text{End}}, \sigma, \delta, t \models \langle \sigma_k[k := \sigma(k) + 1], \delta_k, t+1 \rangle} \text{Op-END-1}$ $\frac{\exists k. (k \in \text{dom}(\sigma) \wedge \sigma(k) <  \delta(k)  - 1)}{\text{Op}_{\text{End}}, \sigma, \delta, t \models \text{Generator-Exhausted}} \text{Op-END-2}$
(d) Semantics - Depth-First Exhaustive	(e) Semantics - Smart Depth-First Exhaustive

sists of two components **D(1)** a program candidate generator and **(2)** a checker that checks if the candidate program produces the correct output. **Program Candidate Generator.** A program candidate generator  $P$  is a generator that, given an input-output example, generates program candidates.

**Building an Exhaustive Depth-First Enumerative Synthesis Engine** Using the exhaustive depth-first semantics for operators presented in figure above for the generator gives an exhaustive depth-first synthesis engine.

**Building a Smart Enumerative Synthesis Engine** The generator in figure above describes a space of programs that is extremely large for such an enumerative pandas synthesis engine to explore in reasonable time.

**Neural-Network Query** The query  $Q$  to each neural network model, regardless of the operator, is of the form  $Q = (D, C)$  where  $D$  and  $C$  are the domain and context passed to the operator.

**Query Encoding** Encoding this query into a neural-network suitable format poses several challenges. Recall that the context and the domain passed to operators in the pandas program candidate generator contain complex structures such as dataframes. Dataframes are 2-D structures which can contain arbitrary Python objects as primitive elements. Even restricting to strings or numbers, the set of possible primitive elements is infinite. This renders all common value-to-value map-based encoding techniques popular in machine learning, such as one-hot encoding, inapplicable. At the same time, the encoding needs to retain enough information about the context to generalize to unseen queries which may occur when the synthesis engine is deployed in practice. Therefore, simply abstracting away the exact values is not viable. In summary, a suitable encoding needs to (1) abstract away only irrelevant information and (2) be



suitably structured for neural processing. a graph-based encoding was designed that possesses all these desirable properties.

**Graph-Based Encoding** the authors describe how to encode the domain  $D$  and the context  $C$  as a graph, consisting of nodes, edges between pairs of nodes, and labels on nodes and edges. The overall rationale is that it is not the concrete values, but rather the relationships amongst values, that really encode the transformation at hand. That is, relationship edges should be sufficient for a neural network to learn from. For example, the essence of transformation is that the values of the column ‘Category’ now become the columns of the pivoted dataframe, with the ‘Date’ column as an index, and the ‘Expense’ as values. The concrete names are immaterial. Recall that the domain and context are essentially collections of elements. How to encode each such element  $e$  individually as a graph  $G_e$  is described below. followed by the description of the procedure to combine these graphs into a single graph  $G_Q$  representing the graph-encoding of the full query  $Q$ .

**Encoding Primitives** If the element  $e$  is a primitive value (strings, ints, float, lambda, NaN etc.), its graph encoding  $G_e$  contains a single node representing  $e$ . This node is assigned a label based on the data-type of the element as well as the source of the element. The source of an element indicates whether it is part of the domain, if it is one of the input-outputs in the I/O example, if it is one of the intermediates, or none of these.

**Encoding DataFrames** If the element  $e$  is a dataframe, each cell element in the dataframe is encoded as a node in the graph  $G_e$ . The label of the node includes the type of the element (string, number, float, lambda, NaN, etc.). The label also includes the source of the dataframe, i.e. whether the dataframe is part of the domain, input, output, intermediate, or none of these. Nodes are added to  $G_e$  that represent the schema of the dataframe, by creating a node for each row index and column name of the dataframe. Finally, a representor node is added to  $G_e$  that represents the whole of the dataframe. The label of this node contains the type (dataframe) as well as the source of the parent dataframe. Note that this additional representor node is not created when encoding primitive elements. The node representing the primitive element itself acts as its representor node. The graph encoding of a dataframe also contains three kinds of edges to retain the structure of the dataframe. The first kind is adjacency edges. These are added between each pair of cell nodes, column name nodes or row index nodes that are adjacent to each other in the dataframe. Adjacency edges in the four cardinal directions only. The second kind is indexing edges, which are added between each column name node (resp. row index node) and all the cell nodes that belong to that column (resp. row). Finally, the third kind of

edge is a representation edge, between the representor node to all the other nodes corresponding to the contents of the dataframe.

**Encoding the Query  $Q$**  To encode  $Q = (D, C)$ ,  $G_e$  is constructed for each element in  $D$  and  $C$  as described above, and create a graph  $G$  containing these  $G_e$ s as sub-graphs. Additionally, to capture relationships amongst these elements, a fourth kind of edge is added - the equality edge, between nodes originating in different  $G_e$ s such that the elements they represent are equal. Formally, an equality edge between nodes  $n_1$  and  $n_2$  is added if  $n_1 \in G_{e_i} \wedge n_2 \in G_{e_j} \wedge i \neq j \wedge V(n_1) = V(n_2)$  where  $V$  is a function that given  $n$ , retrieves the value encoded as  $n$ . For representor nodes,  $V$  returns the whole element it represents. For example, for a dataframe,  $V$  would return the dataframe itself for the representor node. Equality edges are key to capturing relationships between the inputs and the output in the I/O example, as well as the domain  $D$  and the I/O example. The neural network model can then learn to extract these relationships and use them to infer the required probability distribution.

**Operator-Specific Graph Neural Network Models** Given the graph-based encoding  $G_Q$  of a query  $Q$ , it is fed to a graph neural network model. Each operator has a different model. The input to all network models is a undirected graph  $G = (V, E, X)$ .  $V$  and  $X$  characterize the nodes, where  $V$  is the set of nodes and  $X$  is the embedding  $X : V \rightarrow R^D$ . Effectively,  $X$  maps each node to a one-hot encoding of its label of size  $D$ , where  $D$  is a hyper-parameter.  $E$  contains the edges, where each edge  $e \in E$  is a 3-tuple  $(v_s, v_t, t_e)$ . The source and target nodes are  $v_s$  and  $v_t$ , respectively. The type  $t_e$  of the edge is one of  $\Gamma_e \equiv \text{adjacency, indexing, representor, equality}$  and is also one-hot encoded. Each node  $v$  is assigned as  $t$  at  $e$  vector  $h_v \in R^D$ . The vector to the node embedding  $h(0) = X(v)$  is initialized. The network then propagates information via  $r$  rounds of message passing. During round  $k$  ( $0 \leq k < r$ ), messages are sent across edges. In particular, for each edge  $(v_s, v_t, t_e)$ ,  $v_s$  sends the message  $m_{v_s \rightarrow v_t} = f_k(h_{v_s}^{(k)}, t_e)$  to  $v_t$ .  $f_k : R^{D+|\tau_e|} \rightarrow R^D$  is a single linear layer. These are parameterized by a weight matrix and a bias vector, which are learnable parameters. Each node  $v$  aggregates its incoming messages into  $m_v = g(m_{v_s \rightarrow v} | (v_s, v, t_e) \in E)$  using the aggregator  $g$ . the authors take  $g$  to be the element-wise mean of the incoming messages. The new node state vector  $h_v^{(k+1)}$  for the next round is then computed as  $h_v^{(k+1)} = \text{GRU}(m_v, h_v^{(k)})$  where GRU is the gated recurrent unit [Cho et al. 2014] with start state as  $h_v^{(k)}$  and input  $m_v$ .  $r = 3$  rounds of message passing, as experimentally noticed that further increasing the number of message passing rounds did not increase validation accuracy. After message passing is completed, they have the updated state vectors  $h_v^{(r)}$  for each node  $v$ . Now depending on the operator, these node vectors are further

processed in different ways as described below to obtain the corresponding probability distributions over space of values defined by the operator.

**Select** Element-wise sum-pooling of the node state vectors is performed  $h_v^{(r)}$  into a graph state vector  $h_G$ . Followed by concatenation of  $h_G$  with the node state vectors  $h_{di}^{(r)}$  of the representor nodes  $d_i$  for  $d_i$  each element in the domain  $D$  in the query  $Q$ , to obtain vectors  $h_i = h_G * h_{di}^{(r)}$ .  $h_{i_s}$  is passed through a multi-layer perceptron with one hidden layer and a one-dimensional output layer, and apply softmax over the output values for all the elements to produce a probability distribution over the domain elements  $(p_1, \dots, p_n)$ . For inference, this distribution is returned as the result, while during training they compute cross-entropy loss w.r.t this distribution and the correct distribution where  $p_i = 1$  for the correct choice  $i$  and  $\forall j \neq i, p_j = 0$ .

**Subset** As in Select, element-wise sum-pooling of the node state vectors is performed and concatenate it with the state vectors of representor nodes to obtain the vectors  $h_i = h_G * h_{di}^{(r)}$  for each element in the domain. However, the  $h_{i_s}$  is passed through a multi-layer perceptron with one hidden layer and apply softmax activation on the output layer to obtain a distribution  $(p_{i_k}, p_{e_k})$  over two label classes "include" and "exclude" for each of the domain element  $d_k$  individually. Recall that the space of possible outputs for the Subset operator is the power-set of the domain  $D$ . The probability of these labels corresponds to the probability with which an element is included and excluded from the output set respectively. To compute the probability distribution, the probability of each possible output set is computed as simply the product of the "include" probabilities for the elements included in the set and the "exclude" probabilities for the elements excluded from the set. Again, this distribution is returned as the result during inference, while during training, loss is computed w.r.t this distribution and the correct individual distribution of the elements where  $p_{i_k} = 1/\sum p_{e_k} = 0$  if element  $d_k$  is present in the correct output, else  $p_{i_k} = 0/\sum p_{e_k} = 1$ .

**OrderedSubset and Sequence** element-wise sum-pooling of the node state vectors  $h_v^{(r)}$  is performed into a graph state vector  $h_G$ .  $h_G$  is passed to an LSTM that is unrolled for  $T + 1$  time-steps,  $vGG$  where  $T = |D|$  for OrderedSubset and  $T = l$  for Sequence( $l$ ) where  $l$  the max-length parameter passed to Sequence. The extra time-step is to accommodate a terminal token. For each time-step  $t$ , the output  $o_t$  is concatenated with the node state vectors  $h_{di}^{(r)}$  of the representor nodes  $d_{i_s}$  for each element in the domain passed to the operator to obtain vectors  $h_i^t = o_t * h_{di}^{(r)}$ . At time-step  $t$ , in a similar fashion as Select, a probability distribution is then computed over the domain elements plus an arbitrary terminal token term. The terminal token is used to indicate the end of a

sequence/set. Now, to compute the probability distribution, the probability of each set or sequence  $(a_0, \dots, a_k)$  where  $(k \leq T)$  is simply the product of probabilities of  $a_i$  at time-step  $i$  and the probability of the terminal token term at time-step  $k + 1$ . As before, this distribution is directly returned during inference, while during training, loss is aggregated over individual time-steps; the loss for a time-step is computed as described in Select. All the network models are trained with the ADAM optimizer using cross-entropy loss.

**Training Neural-Backed Generators for Pandas Tuples** of the form  $(I, O, P, K)$  where  $P$  is a pandas program such that  $P(I) = O$  i.e. it produces  $O$  when executed on inputs  $I$ . Also,  $K$  is the sequence of choices made by the operators in the generator such that the generator produces the program  $P$  when it is fed  $I$  and  $O$  as inputs. Then, it is straight-forward to extract training data tuples  $(C, D, c)$  for each operator call by simply running the generator on  $I$  and  $O$  and recording the concrete context  $C$  and domain  $D$  passed to the operator, and forcing the operator to make the choice  $c$ . These tuples are also meaningful by construction, as the operators make choices that lead to the generation of the program  $P$  which solves the synthesis task described by  $I$  and  $O$ . The second insight is that one can obtain these  $(I, O, P, K)$  tuples by using the generator itself. After generating random inputs  $I$  (DataFrames), run the generator on  $I$  using the randomized semantics while simultaneously recording the choices made as  $K$ . The program  $P$  returned by the generator is then run on  $I$  to yield  $O$ .

## 2.5.2. RESULTS

Benchmark	Depth	Candidates Explored		Sequences Explored		Solved		Time(s)	
		AUTO/PANDAS	BASELINE	AUTO/PANDAS	BASELINE	AUTO/PANDAS	BASELINE	AUTO/PANDAS	BASELINE
SO_11881165	1	35	64	1	1	Y	Y	0.54	1.46
SO_11941492	1	783	441	8	8	Y	Y	12.55	2.38
SO_11847222	1	5	15496	1	1	Y	Y	3.32	53.07
SO_18172851	1	-	-	-	-	N	N	-	-
SO_49583055	1	-	-	-	-	N	N	-	-
SO_49592930	1	2	4	1	1	Y	Y	1.1	1.43
SO_49572546	1	3	4	1	1	Y	Y	1.1	1.44
SO_13261175	1	39537	-	18	-	Y	N	300.20	-
SO_13793321	1	92	1456	1	1	Y	Y	4.16	5.76
SO_14085517	1	30	208	1	1	Y	Y	2.24	2.01
SO_14181192	2	136	80	1	1	Y	Y	0.71	1.46
SO_49567728	2	1684022	-	2	-	Y	N	753.10	-
SO_13261691	2	65	612	1	1	Y	Y	2.96	3.22
SO_13659881	2	2	15	1	1	Y	Y	1.38	1.41
SO_13807758	2	711	263	2	2	Y	Y	7.21	1.81
SO_34365578	2	-	-	-	-	N	N	-	-
SO_10982266	3	-	-	-	-	N	N	-	-
SO_11811392	3	-	-	-	-	N	N	-	-
SO_49581206	3	-	-	-	-	N	N	-	-
SO_12065885	3	924	2072	1	1	Y	Y	0.9	4.67
SO_13576164	3	22966	-	5	-	Y	N	339.25	-
SO_14028037	3	-	-	-	-	N	N	-	-
SO_59762029	3	27	115	1	1	Y	Y	1.90	1.50
SO_21982987	3	8385	8278	10	10	Y	Y	30.80	13.91
SO_39656470	3	-	-	-	-	N	N	-	-
SO_23321300	3	-	-	-	-	N	N	-	-
Total						17/26	14/26		

Performance on Real-World Benchmarks. Dashes (-) indicate timeouts by the technique. The code is published at <https://github.com/rbavishi/autopandas>.

### 3. Bug Detection

#### 3.1. Introduction

Graph-structured data appears frequently in domains including chemistry, natural language semantics, social networks, and knowledge bases. In part of this survey, they study feature learning techniques for graph-structured inputs. Starting point is previous work on Graph Neural Networks, which the authors modify to use gated recurrent units and modern optimization techniques and then extend to output sequences. The result is a flexible and broadly useful class of neural network models that has favorable inductive biases relative to purely sequence-based models (e.g., LSTMs) when the problem is graph-structured.

Another paper presents an alternative approach to creating static bug finders. Instead of relying on human expertise, they utilize deep neural networks to train static analyzers directly from data. In particular, they frame the problem of bug finding as a classification task and train a classifier to differentiate the buggy from non-buggy programs using Graph Neural Network (GNN). Static analysis is an effective technique to catch bugs early when they are cheap to fix. Unlike dynamic analysis, static analysis reasons about every path in a program, offering formal guarantees for its runtime behavior. As an evidence of their increasing maturity and popularity, many static analyzers have been adopted by major tech companies to prevent bugs leaked to their production code. Examples include Google’s Tricorder, Facebook’s Getafix and Zoncolan, and Microsoft’s Visual Studio IntelliCode.

Despite the significant progress, static analyzers suffer from several well-known issues. One, in particular, is the high false positive rate which tends to overshadow true positives and hurt usability. The reason for this phenomenon is well-known: all nontrivial program properties are mathematically undecidable, meaning that automated reasoning of software generally must involve approximation. On the other hand, problems of false negatives also need to be dealt with. Recently, Habib and Pradel investigated how effective the state-of-the-art static analyzers are in handling a set of real-world bugs. Habib et al. show more than 90 percent of the bugs are missed, exposing the severity of false negatives.

Software defect prediction, which predicts defective code regions, can help developers find bugs and prioritize their testing efforts. To build accurate prediction models, previous studies focus on manually designing features that encode the characteristics of programs and exploring different machine learning algorithms. Existing traditional features often fail to capture the semantic differences of programs, and such a capability is needed for building accurate prediction models.

#### 3.2. Related work

The most closely related work is GNNs. (Micheli, 2009) proposed another closely related model that differs from GNNs mainly in the output model. GNNs have many of the same desirable properties of pointer networks ((Vinyals et al., 2017)); when using node selection output layers, nodes from the input can be chosen as outputs. There are two main differences: first, in GNNs the graph structure is explicit, which makes the models less general but may provide stronger generalization ability; second, pointer networks require that each node has properties (e.g., a location in space), while GNNs can represent nodes that are defined only by their position in the graph, which makes them more general along a different dimension. GGS-NNs are related to soft alignment and attentional models (e.g., (Bahdanau et al., 2016); (Kumar et al., 2016); (Sukhbaatar et al., 2015)) in two respects: first, the graph representation uses context to focus attention on which nodes are important to the current decision; second, node annotations in the program verification example keep track of which nodes have been explained so far, which gives an explicit mechanism for making sure that each node in the input has been used over the sequence of producing an output.

#### 3.3. Gated Graph Sequence Neural Networks

This paper’s main contribution is an extension of Graph Neural Networks that outputs sequences. Previous work on feature learning for graph-structured inputs has focused on models that produce single outputs such as graph-level classifications, but many problems with graph inputs require outputting sequences. There are two settings for feature learning on graphs: (1) learning a representation of the input graph, and (2) learning representations of the internal state during the process of producing a sequence of outputs. The implementation is published at <https://github.com/yujiali/gggnn>.

##### 3.3.1. GRAPH REPRESENTATION- GNN AND GATED

GNNs are a general neural network architecture defined according to a graph structure  $G = (V, E)$ . Nodes  $v \in V$  take unique values from  $1, \dots, |V|$ , and edges are pairs  $e = (v, v') \in V \times V$ . The authors focus in this work on directed graphs, so  $(v, v')$  represents a directed edge  $v \rightarrow v'$ , but note that the framework can easily be adapted to undirected graphs; see (Scarselli et al., 2009). The node vector (or node representation or node embedding) for node  $v$  is denoted by  $h_v \in \mathbb{R}^D$ . Graphs may also contain node labels  $l_v \in 1, \dots, L_V$  for each node  $v$  and edge labels or edge types  $l_e \in 1, \dots, L_E$  for each edge. They overload notation and let  $h_S = h_v | v \in S$  when  $S$  is a set of nodes, and  $l_S = l_e | e \in S$  when  $S$  is a set of edges. The function  $IN(v) = v' \mid (v', v) \in E$  returns the set of predecessor nodes  $v'$  with  $v' \rightarrow v$ . Analogously,  $OUT(v) =$

$v' \leftarrow (v, v') \in E$  is the set of successor nodes  $v'$  with edges  $v \rightarrow v'$ . The set of all nodes neighboring  $v$  is  $NBR(v) = IN(v) \cup OUT(v)$ , and the set of all edges incoming to or outgoing from  $v$  is  $CO(v) = \{(v', v'') \in E \mid v = v' \vee v = v''\}$ . GNNs map graphs to outputs via two steps. First, there is a propagation step that computes node representations for each node; second, an output model  $o_v = g(h_v, l_v)$  maps from node representations and corresponding labels to an output  $o_v$  for each  $v \in V$ . In the notation for  $g$ , they leave the dependence on parameters implicit.

In GNNs, there is no point in initializing node representations because the contraction map constraint ensures that the fixed point is independent of the initializations. This is no longer the case with GG-NNs, which lets the authors to incorporate node labels as additional inputs. To distinguish these node labels used as inputs from the ones introduced before, called node annotations, and use vector  $x$  to denote these annotations. This will cause the first dimension of node representation to be copied along forward edges. With this setting of parameters, the propagation step will cause all nodes reachable from  $s$  to have their first bit of node representation set to 1. The output step classifier can then easily tell whether node  $t$  is reachable from  $s$  by looking whether some node has nonzero entries in the first two dimensions of its representation vector.

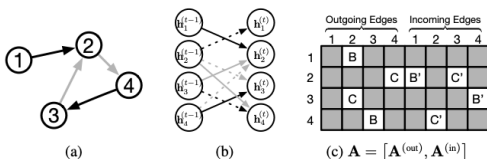
### 3.3.2. PROPOGATION MODEL- GNN AND GATED

An iterative procedure propagates node representations. Initial node representations  $h_v$  are set to arbitrary values, then each node representation is updated following the recurrence below until convergence, where  $t$  denotes the timestep:

$$h_v^{(t)} = f^*(l_v, l_{CO(v)}, l_{NBR(v)}, h_{NBR(v)}^{t-1})$$

Several variants are discussed in (Scarselli et al., 2009) including positional graph forms, node-specific updates, and alternative representations of neighborhoods. Concretely, (Scarselli et al., 2009) suggest decomposing  $f_*(\cdot)$  to be a sum of per-edge terms.

$$\begin{aligned} h_v^{(1)} &= [x_v^\top, 0]^\top & (1) \quad r_v^t &= \sigma(W^r a_v^{(t)} + U^r h_v^{(t-1)}) & (4) \\ a_v^{(t)} &= A_{v,:} [h_1^{(t-1)\top} \dots h_{|V|}^{(t-1)\top}]^\top + b & (2) \quad \tilde{h}_v^{(t)} &= \tanh(W^a a_v^{(t)} + U^a (r_v^t \odot h_v^{(t-1)})) & (5) \\ z_v^t &= \sigma(W^z a_v^{(t)} + U^z h_v^{(t-1)}) & (3) \quad h_v^{(t)} &= (1 - z_v^t) \odot h_v^{(t-1)} + z_v^t \odot \tilde{h}_v^{(t)} & (6) \end{aligned}$$



The matrix  $A \in R^{D|V| \times 2D|V|}$  determines how nodes in the graph communicate with each other. The sparsity structure and parameter tying in  $A$  is illustrated in Figure above. The sparsity structure corresponds to the edges of the graph, and the parameters in each submatrix are determined by the edge type and direction.  $A_v \in R^{D|V| \times 2D|V|}$  are the two columns of blocks in  $A^{(out)}$  and  $A^{(in)}$  corresponding to node  $v$ . Eq. 1 is the initialization step, which copies node annotations into the first components of the hidden state and pads the rest with zeros. Eq. 2 is the step that passes information between different nodes of the graph via incoming and outgoing edges with parameters dependent on the edge type and direction. Contains activations from edges in both directions. The remaining are GRU-like updates that incorporate information from the other nodes and from the previous timestep to update each node's hidden state.  $z$  and  $r$  are the update and reset gates,  $\sigma(x) = 1/(1 + e^{-x})$  is the logistic sigmoid function, and multiplication is element-wise multiplication. Initial experiments with a vanilla recurrent neural network-style update, but in preliminary experiments found this GRU-like propagation step to be more effective.

### 3.3.3. OUTPUT MODEL AND LEARNING- GNN AND GATED

The output model is defined per node and is a differentiable function  $g(h_v, l_v)$  that maps to an output. This is generally a linear or neural network mapping, independent per node, which are implemented by mapping the final node representations  $h_v^{(T)}$ , to an output  $o_v = g(h_v, l_v)$  for each node  $v \in V$ . To handle graph-level classifications. Learning is done via the Almeida-Pineda algorithm, which works by running the propagation to convergence, and then computing gradients based upon the converged solution. This has the advantage of not needing to store intermediate states in order to compute gradients. The disadvantage is that parameters must be constrained so that the propagation step is a contraction map.

GG-NNs support node selection tasks by making  $o_v = g(h_v, x_v)$  for each node  $v \in V$  output node scores and applying a softmax over node scores. Second, for graph-level outputs, a graph level representation vector is defined as

$$h_G = \tanh(\sum_{v \in V} \sigma(i(h_v^{(T)}, x_v)) X(\tanh(j(h_v^{(T)}, x_v))))$$

where  $\sigma(i(h_v^{(T)}, x_v))$  acts as a soft attention mechanism that decides which nodes are relevant to the current graph-level task.  $i$  and  $j$  are neural networks that take the concatenation of  $h_v^{(T)}$  and  $x_v$  as input and outputs real-valued vectors.

### 3.3.4. GATED GRAPH SEQUENCE NEURAL NETWORKS

several GG-NNs operate in sequence to produce an output sequence  $o^{(1)} \dots o^{(K)}$ . For the  $k^{th}$  output step, the matrix of

node annotations is denoted as  $X^{(k)} = [x_1^{(k)}; \dots; x_{|V|}^{(k)}]^T \in R^{|V| \times L_V}$ . Two GG-NNs are used:  $F_o^{(k)}$  and  $F_x^{(k)}$ :  $F_o^{(k)}$  for predicting  $o^{(k)}$  from  $X^{(k)}$ , and  $F_x^{(k)}$  for predicting  $X^{(k+1)}$  from  $X^{(k)}$ .  $X^{(k+1)}$  can be seen as the states carried over from step  $k$  to  $k+1$ . Both  $F_o^{(k)}$  and  $F_x^{(k)}$  contain a propagation model and an output model. In the propagation models, the matrix of node vectors at the  $t^{th}$  propagation step of the  $k^{th}$  output step is denoted as  $H^{(k,t)} = [h_1^{(k,t)}; \dots; h_{|V|}^{(k,t)}]^T \in R^{|V| \times D}$ . As before, in step  $k$ ,  $H^{(k,t)}$  is set by 0-extending  $X^{(k)}$  per node. Alternatively,  $F_o^{(k)}$  and  $F_x^{(k)}$  can share a single propagation model, and just have separate output models. This simpler variant is faster to train and evaluate, and in many cases can achieve similar performance level as the full model. But in cases where the desired propagation behavior for  $F_o^{(k)}$  and  $F_x^{(k)}$  are different, this variant may not work as well.

A node annotation output model for predicting  $X^{(k+1)}$  from  $H^{(k,T)}$  is introduced. The prediction is done for each node independently using a neural network  $j(h_v^{(T)}, x_v)$  of  $h_v$  and  $x_v$  as input and outputs a vector of real-valued scores:

$$x_v^{k+1} = \sigma(j(h_v^{(k,T)}, x_v^{(k)}))$$

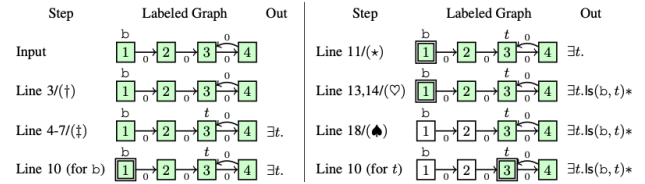
There are two settings for training GGS-NNs: specifying all intermediate annotations  $X^{(k)}$ , or training the full model end-to-end given only  $X^{(1)}$ , graphs and target sequences. The former can improve performance when the authors have domain knowledge about specific intermediate information that should be represented in the internal state of nodes, while the latter is more general.

The sequence prediction task decomposes into single step prediction tasks and can be trained as separate GG-NNs. Sequence outputs with latent annotations: When intermediate node annotations  $X^{(k)}$  are not available during training, the proposed solution treats them as hidden units in the network, and train the whole model jointly by back propagating through the whole sequence.

### 3.3.5. PROGRAM VERIFICATION WITH GGS-NNs

Separation logic is used, which uses inductive predicates to describe abstract data structures. For example, a list segment is defined as  $ls(x, y) \equiv x = y \vee \exists v, n. ls(n, y) * x \mapsto val : v, next : n$ , where  $x \mapsto val : v, next : n$  means that  $x$  points to a memory region that contains a structure with  $val$  and  $next$  fields whose values are in turn  $v$  and  $n$ . The  $*$  connective is a conjunction as  $\wedge$  in Boolean logic, but additionally requires that its operators refer to “separate” parts of the heap. Thus,  $ls(cur, NULL)$  implies that  $cur$  is either  $NULL$ , or that it points to two values  $v, n$  on the heap, where  $n$  is described by  $ls$  again. The formula  $\exists t. ls(a, cur) * ls(cur, NULL) * ls(b, t)$  is an invariant of the loop (i.e., it holds when entering the loop, and after every iteration). Using it, one can prove that no program run will

fail due to dereferencing an unallocated memory address (this property is called memory safety) and that the function indeed concatenates two lists using a Hoare-style verification scheme. The authors propose to use machine learning. Given a program, run it a few times and extract the state of memory (represented as a graph; see below) at relevant program locations, and then predict a separation logic formula. Representing Heap State as a Graph As inputs they consider directed, possibly cyclic graphs representing the heap of a program. These graphs can be automatically constructed from a program’s memory state. Each graph node  $v$  corresponds to an address in memory at which a sequence of pointers  $v_0, \dots, v_k$  is stored. Graph edges reflect these pointer values, i.e.,  $v$  has edges labeled with  $0, \dots, k$  that point to nodes  $v_0, \dots, v_k$ , respectively. A subset of nodes are labeled as corresponding to program variables. Output Representation they restrict ourselves to a syntactically restricted version of separation logic, in which formulas are of the form  $\exists x_1, \dots, x_n. a_1 * \dots * a_m$ , where each atomic formula  $a_i$  is either  $ls(x, y)$  (a list from  $x$  to  $y$ ),  $tree(x)$  (a binary tree starting in  $x$ ), or  $none(x)$  (no data structure at  $x$ ). Existential quantifiers are used to give names to heap nodes which are needed to describe a shape, but not labeled by a program variable.



Algorithm 1 Separation logic formula prediction procedure

```

Input: Heap graph  $\mathcal{G}$  with named program variables
1:  $\mathcal{X} \leftarrow$  compute initial labels from  $\mathcal{G}$ 
2:  $\mathcal{H} \leftarrow$  initialize node vectors by 0-extending  $\mathcal{X}$ 
3: while  $\exists$  quantifier needed do ▷ Graph-level Classification (†)
4:    $t \leftarrow$  fresh variable name
5:    $v \leftarrow$  pick node ▷ Node Selection (†)
6:    $\mathcal{X} \leftarrow$  turn on “is-named” for  $v$  in  $\mathcal{X}$ 
7:   print “ $\exists t$ .”
8: end while
9: for node  $v_\ell$  with label “is-named” in  $\mathcal{X}$  do
10:   $\mathcal{H} \leftarrow$  initialize node vectors, turn on “active” label for  $v_\ell$  in  $\mathcal{X}$ 
11:   $pred \leftarrow$  pick data structure predicate ▷ Graph-level Classification (*)
12:  if  $pred = ls$  then
13:     $\ell_{end} \leftarrow$  pick list end node ▷ Node Selection (∇)
14:    print “ $ls(\ell, \ell_{end}) *$ ”
15:  else
16:    print “ $pred(\ell) *$ ”
17:  end if
18:   $\mathcal{X} \leftarrow$  update node annotations in  $\mathcal{X}$  ▷ Node Annotation (●)
19: end for
    
```

The authors use the full GGS-NN model where  $F_o^{(k)}$  and  $F_x^{(k)}$  have separate propagation models for 10 times steps and use  $D=16$  dimensional node representations. To make batch predictions, run one GGS-NN for each graph simultaneously. For each prediction step, the outputs of all the GGS-NNs at that step across the batch of graphs are aggregated. For node selection outputs, the common named



variables link nodes on different graphs together, which is the key for aggregating predictions in a batch. Compute the score for a particular named variable  $t$  as  $o_t = \sum_g o_{V_g(t)}^g$ , where  $V_g(t)$  maps variable name  $t$  to a node in graph  $g$ , and  $o_{V_g(t)}^g$  is the output score for named variable  $t$  in graph  $g$ . When applying a softmax over all names using  $o_t$  as scores, this is equivalent to a model that computes  $p(\text{toselect} = t) = \prod_g p_g(\text{toselect} = V_g(t))$ . For graph-level classification outputs, add up scores of a particular class across the batch of graphs, or equivalently compute  $p(\text{class} = k) = \prod_g p_g(\text{class} = k)$ . Node annotation outputs are updated for each graph independently as different graphs have completely different set of nodes.

### 3.3.6. RESULTS.

On comparison to GGS-NN-based model with a method developed earlier. The earlier approach treats each prediction step as standard classification, and requires complex, manual, problem-specific feature engineering, to achieve an accuracy of 89.11 percent. In contrast, this new model was trained with no feature engineering and very little domain knowledge and achieved an accuracy of 89.96 percent.

## 3.4. DBN

Specifically, this paper leverages Deep Belief Network (DBN) to automatically learn semantic features from token vectors extracted from programs' Abstract Syntax Trees (ASTs). The code is published at <https://github.com/mingkaic/defector>.

The authors refer to the set of instances used for building models as the training set, whereas the set of instances used to evaluate the trained models as the test set. When performing within-project defect prediction (following existing work, they call this WPDP), the training and test sets are from the same project A. When performing cross-project defect prediction (following existing work [41] they call this CPDP), prediction models are trained by training set from a project A (source), and test set is from a different project B (target). In this study, the authors examine the performance of learned semantic features on both WPDP and CPDP.

### 3.4.1. DBN FORMULATION

DBN contains one input layer and several hidden layers, and the top layer is the output layer that used as features to represent input data. Each layer consists of several stochastic nodes. The number of hidden layers and the number of nodes in each layer vary depending on users' demand. In this study, the size of learned semantic features is the number of nodes in the top layer. The idea of DBN is to enable the network to reconstruct the input data using generated features by adjusting weights between nodes in different

layers. DBN models the joint distribution between input layer and the hidden layers as follows:

$$P(x, h^1, \dots, h^l) = P(x, h^1) \left( \prod_{k=1}^l P(h^k | h^{k+1}) \right)$$

where  $x$  is the data vector from input layer,  $l$  is the number of hidden layers, and  $h^k$  is the data vector of  $k^{th}$  layer ( $1 \leq k \leq l$ ).  $P(h^k | h^{k+1})$  is a conditional distribution for the adjacent  $k$  and  $k + 1$  layer. To calculate  $P(h^k | h^{k+1})$ , each pair of two adjacent layers in DBN are trained as a Restricted Boltzmann Machines (RBM). A RBM is a two-layer, undirected, bipartite graphical model where the first layer consists of observed data variables, referred to as visible nodes, and the second layer consists of latent variables, referred to as hidden nodes.  $P(h^k | h^{k+1})$  can be efficiently calculated as:

$$P(h^k | h^{k+1}) = \prod_{j=1}^{n_k} P(h_j^k | h^{k+1})$$

$$P(h_j^k | h^{k+1}) = \text{sigm}(b_j^k + \sum_{a=1}^{n_{k+1}} W_{aj}^k h_a^{k+1})$$

where  $n_k$  is the number of node in layer  $k$ ,  $\text{sigm}(c) = 1/(1 + e^{-c})$ ,  $b$  is a bias matrix,  $b_j^k$  is the bias for node  $j$  of layer  $k$ , and  $W^k$  is the weight matrix between layer  $k$  and  $k + 1$ . DBN automatically learns  $W$  and  $b$  matrices using an iteration process.  $W$  and  $b$  are updated via log-likelihood stochastic gradient descent:

$$W_{ij}(t + 1) = W_{ij}(t) + \eta(\partial \log(P(v|h)))/(\partial W_{ij})$$

$$b_k^o(t + 1) = b_k^o(t) + \eta(\partial \log(P(v|h)))/(\partial b_k^o)$$

where  $t$  is the  $t^{th}$  iteration,  $\eta$  is the learning rate,  $P(v|h)$  is the probability of the visible layer of a RBM given the hidden layer,  $i$  and  $j$  are two nodes in different layers of the RBM,  $W_{ij}$  is the weight between the two nodes, and  $b_k^o$  is the bias on the node  $o$  in layer  $k$ . To train the network, one first initializes all  $W$  matrices between two layers via RBM and sets the biases  $b$  to 0. They can be well-tuned with respect to a specific criterion. They use the number of training iterations as the criterion for tuning  $W$  and  $b$ . The well-tuned  $W$  and  $b$  are used to set up a DBN for generating semantic features for both training and test data.

### 3.4.2. APPROACH

Their approach first extracts a vector of tokens from the source code of each file in both the training and test sets. Since DBN requires input data in the form of integer vectors, a mapping is built between integers and tokens and convert the token vectors to integer vectors. To generate semantic features, the integer vectors of the training set are used to build a DBN. Then, the DBN is used to automatically generate semantic features from the integer vectors of the training and test sets. Finally, based on the generated semantic features, defect prediction models are built from the training set, and evaluate their performance on the test set.

AST nodes are excluded that are not one of these three categories, such as assignment and intrinsic type declaration, because they are often method-specific or class-specific, which may not be generalizable to the whole project. Adding them may dilute the importance of other nodes. For cross-project defect prediction, they extract all of AST nodes.

To detect and eliminate mislabeling data, and help DBN learn common knowledge between the semantic information of buggy and clean files, the edit distance similarity computation algorithm is adopted to define the distances between instances. The edit distances are sensitive to both the tokens and order among the tokens. Given two token sequences A and B, the edit distance  $d(A, B)$  is the minimum-weight series of edit operations that transform A to B. The smaller  $d(A, B)$  is, the more similar A and B are.

DBN takes only numerical vectors as inputs, and the lengths of the input vectors must be the same. To use DBN to generate semantic features by using DBN, a mapping is built between integers and tokens, and encode token vectors to integer vectors. Each token has a unique integer identifier while different method names and class names will be treated as different tokens. Since integer vectors may have different lengths, 0 is appended to the integer vectors to make all the lengths consistent and equal to the length of the longest vector. Adding zeros does not affect the results, and it is simply a representation transformation to make the vectors acceptable by DBN.

To generate semantic features for distinguishing buggy and clean files, DBN is trained by using the training data. To train an effective DBN for learning semantic features, three parameters are tuned, which are: 1) the number of hidden layers, 2) the number of nodes in each hidden layer, and 3) the number of training iterations. Existing work that leveraged DBN to generate features for NLP and image recognition reported that the performance of DBN-generated features is sensitive to these parameters. The number of nodes to be the same in each layer. Through these hidden layers and nodes, DBN obtains characteristics that are difficult to be observed but are capable of capturing semantic differences. For each node, DBN learns probabilities of traversing from this node to the nodes of its top level. Through back-propagation validation, DBN reconstructs the input data using generated features by adjusting weights between nodes in different layers. DBN requires the values of input data ranging from 0 to 1, while data in input vectors can have any integer values due to mapping approach. To satisfy the input range requirement, they normalize the values in the data vectors of the training and test sets by using min-max normalization.

After training a DBN, both the weights  $w$  and the biases  $b$  are fixed. The authors input the normalized integer vectors of the training data and the test data into the DBN respectively, and then obtain semantic features for the training and

test data from the output layer of the DBN. This leads to the following table and graphs for results.

Project	Description	Releases	Avg Files	Avg Buggy Rate (%)
ant	Java based build tool	1.5,1.6,1.7	488	13.4
camel	Enterprise integration framework	1.2,1.4,1.6	1,046	18.7
jEdit	Text editor designed for programmers	3.2,4.0,4.1	645	19.2
log4j	Logging library for Java	1.0,1.1	150	49.7
lucene	Text search engine library	2.0,2.2,2.4	402	35.8
xalan	A library for transforming XML files	2.4,2.5	992	29.6
xerces	XML parser	1.2,1.3	549	15.7
ivy	Dependency management library	1.4,2.0	311	20.0
synapse	Data transport adapters	1.0,1.1,1.2	220	22.7
poi	Java library to access Microsoft format files	1.5,2.5,3.0	416	40.7

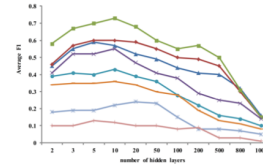


Figure 5: Defect prediction performance with different parameters

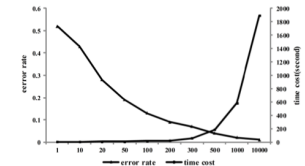


Figure 6: Average error rate and time cost for different numbers of iterations

## References

- Allamanis, M., Brockschmidt, M., and Khademi, M. Learning to represent programs with graphs. Technical Report MSR-TR-2017-44, November 2017. URL <https://www.microsoft.com/en-us/research/publication/learning-represent-programs-graphs/>.
- Allamanis, M., Barr, E. T., Devanbu, P., and Sutton, C. A survey of machine learning for big code and naturalness, 2018.
- Alur, R., Bodik, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pp. 1–8, 2013. doi: 10.1109/FMCAD.2013.6679385.
- Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate, 2016.
- Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. Deepcoder: Learning to write programs, 2017.
- Bošnjak, M., Rocktäschel, T., Naradowsky, J., and Riedel, S. Programming with a differentiable forth interpreter, 2017.
- Bunel, R., Hausknecht, M., Devlin, J., Singh, R., and Kohli, P. Leveraging grammar and reinforcement learning for neural program synthesis, 2018.
- Chen, X., Liu, C., and Song, D. Towards synthesizing complex programs from input-output examples, 2018.

- Dai, H., Khalil, E. B., Zhang, Y., Dilkina, B., and Song, L. Learning combinatorial optimization algorithms over graphs, 2018.
- Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., rahman Mohamed, A., and Kohli, P. RobustFill: Neural program learning under noisy I/O. In Precup, D. and Teh, Y. W. (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 990–998, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. URL <http://proceedings.mlr.press/v70/devlin17a.html>.
- Feng, Y., Martins, R., Bastani, O., and Dillig, I. Program synthesis using conflict-driven learning. *SIGPLAN Not.*, 53(4):420–435, June 2018. ISSN 0362-1340. doi: 10.1145/3296979.3192382. URL <https://doi.org/10.1145/3296979.3192382>.
- Gulwani, S. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’11, pp. 317–330, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304900. doi: 10.1145/1926385.1926423. URL <https://doi.org/10.1145/1926385.1926423>.
- Hindle, A., Barr, E. T., Su, Z., Devanbu, P. T., , and Gabel, M. On the naturalness of software. *Communications of the ACM: Invited Research Highlights (CACM)*, pp. 122–131, 2016. URL <http://softwareprocess.ca/pubs/hindle2016CACM.pdf>.
- Jha, S., Gulwani, S., Seshia, S. A., and Tiwari, A. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE ’10, pp. 215–224, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605587196. doi: 10.1145/1806799.1806833. URL <https://doi.org/10.1145/1806799.1806833>.
- Kalyan, A., Mohta, A., Polozov, O., Batra, D., Jain, P., and Gulwani, S. Neural-guided deductive search for real-time program synthesis from examples, 2018a.
- Kalyan, A., Mohta, A., Polozov, O., Batra, D., Jain, P., and Gulwani, S. Neural-guided deductive search for real-time program synthesis from examples, 2018b.
- Kool, W., van Hoof, H., and Welling, M. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019. URL <https://openreview.net/forum?id=ByxBFsrQYm>.
- Kumar, A., Irsoy, O., Ondruska, P., Iyyer, M., Bradbury, J., Gulrajani, I., Zhong, V., Paulus, R., and Socher, R. Ask me anything: Dynamic memory networks for natural language processing, 2016.
- Maddison, C. J. and Tarlow, D. Structured generative models of natural source code. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML’14, pp. II–649–II–657. JMLR.org, 2014.
- Micheli, A. Neural network for graphs: A contextual constructive approach. *IEEE Transactions on Neural Networks*, 20(3):498–511, 2009. doi: 10.1109/TNN.2008.2010350.
- Murali, V., Qi, L., Chaudhuri, S., and Jermaine, C. Neural sketch learning for conditional program generation, 2018.
- Parisotto, E., Mohamed, A., Singh, R., Li, L., Zhou, D., and Kohli, P. Neuro-symbolic program synthesis. In *5th International Conference on Learning Representations (ICLR 2017)*, February 2017. URL <https://www.microsoft.com/en-us/research/publication/neuro-symbolic-program-synthesis-2/>.
- Rolim, R., Soares, G., D’Antoni, L., Polozov, O., Gulwani, S., Gheyi, R., Suzuki, R., and Hartmann, B. Learning syntactic program transformations from examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 404–415, 2017. doi: 10.1109/ICSE.2017.44.
- Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. The graph neural network model. *Trans. Neur. Netw.*, 20(1):61–80, January 2009. ISSN 1045-9227. doi: 10.1109/TNN.2008.2005605. URL <https://doi.org/10.1109/TNN.2008.2005605>.
- Si, X., Dai, H., Raghothaman, M., Naik, M., and Song, L. Learning loop invariants for program verification. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS’18, pp. 7762–7773, Red Hook, NY, USA, 2018. Curran Associates Inc.
- Smith, C. and Albarghouthi, A. Mapreduce program synthesis. *SIGPLAN Not.*, 51(6):326–340, June 2016. ISSN 0362-1340. doi: 10.1145/2980983.2908102. URL <https://doi.org/10.1145/2980983.2908102>.
- Solar Lezama, A. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008. URL <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html>.

Sukhbaatar, S., Szlam, A., Weston, J., and Fergus, R. End-to-end memory networks, 2015.

Vinyals, O., Fortunato, M., and Jaitly, N. Pointer networks, 2017.

Yaghmazadeh, N., Wang, Y., Dillig, I., and Dillig, T. Sqlizer: Query synthesis from natural language. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi: 10.1145/3133887. URL <https://doi.org/10.1145/3133887>.