A Survey of Graph Neural Networks for Programming Languages

Hemil Desai [*1] **Sripath Mishra** [*1] **Justin Yi** [*1]

Abstract

Research at the intersection of Programming Languages, Deep Learning and Graph Neural Networks has been rapidly evolving over the past few years. Deep Learning and Graph Neural Network techniques has shown tremendous progress and potential across a variety of sub domains in Programming Languages. Our goal with this paper is to survey the rise of Neural techniques and the integration and use of Graph Neural Networks in these techniques, spanning some interesting subdomains. We also point the users towards a common benchmark across Programming Languages and NLP and present ideas on generalized representation learning of Code using GNNs.

1. Introduction

Our objective for this paper is to create a comprehensive survey paper over the usage of graph neural networks and deep learning in the domain of programming languages. Till now we have seen several advancements in programming techniques which use graph neural networks to comprehend programming languages. We have also seen works being done in bug detection, similarity checking, synthesis, semantics and decompilation of programs. There is also a growing interest in combining programming languages with other domains like NLP (Code x NLP). These subdomains however are in their infancy and have huge potentials for advancements. In the current state these subdomains are separate and interdomain work is limited. This leads to lack of survey papers which connect these domains and show the complete potential of using graph neural networks in the complete process of programming. This survey paper aims to diminish the separation between these subdomains by aggregating state of the art work and providing background information to understand the current and potential usage of graph neural networks in programming languages. In this paper we will be focusing on the aforementioned subdomains and going in detail over the state of the art techniques in each one of them.

We first start with an overview of how to represent code using Graph Neural Networks, since this will be the parent step in most applications. Next, we delve into the specific subdomains. We give an introduction of the subdomain and the relevant challenges, datasets and existing work. We then dive into the existing neural methods used to accomplish the tasks in the subdomain and survey the different methods and how they have evolved with time. Towards the end, we survey an interesting benchmark across the domains of Programming Language and NLP and present some ideas on how GNNs can be used to accelerate common and cross domain research in the area. We wrap up with some future work and ideas along with the conclusion.

2. 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 [5] 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

[*]Equal contribution [1]Department of Computer Science, University of California, Los Angeles. Correspondence to: Hemil Desai <hemil10@ucla.edu>, Sripath Mishra <mishra60@ucla.edu>, Justin Yi <joostinyi00@gmail.com>.

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.

2.1. Formulation of task description

In the proposed formulation a source code file is viewed as a sequence of tokens $t_0 \ldots t_N = T$, in which some tokens t_{λ_0} , t_{λ_1} . . . 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 tλ: Given $t_0 \ldots t_{\lambda-1}$ and $t_{\lambda+1}$, . . . , λ_N , correctly select t_λ from V_{t_λ} . 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).

2.2. Gates 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,...,E_K) where K is the number of edge types. Each v is annotated with a real-valued vector $x(v) \in R^D$ 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 $m_k^{(v)}$ = $g(m_k^{(u)}$ — 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 $h'^{(v)}$ = 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.

2.3. 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

2.4. 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 τ(v). To use it, a learnable embedding function r(τ) 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 τ(v) to the set of its supertypes, i.e. τ^*(v) = τ: τ(v) implements type $\tau \cup \tau$(v). Then the type representation r^*(v) of a variable v as the element-wise maximum of r(τ) : $\tau \in \tau^*$(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 τ^*(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.

2.5. Initial Node Representation

The poposed 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.

2.6. 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 ¡SLOT¿ 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 ¡SLOT¿ 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.

2.7. 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 $v_{<SLOT>}$ 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 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) = $h^{(v_{<SLOT>})}$ and u(t,v) = $h^{(v_t,v)}$. Finally, the correct variable usage at the location is computed as arg maxv $c(t)^T$ u(t, v). The model is trained using maximum likelihood.

3. Program Similarity and Classification

3.1. Introduction

Graph similarity search is one of the most important graph-based applications, with extensive applications to complex graph querying systems such as finding similar chemical compounds. However, Graph Edit Distance (GED) and Maximum Common Subgraph (MCS), the core operations of graph similarity/distance computation are very costly in practice. As graph databases become increasingly dense, it becomes paramount to be able to efficiently mine these large repositories for fundamental tasks of similarity computation and classification. Additionally, domain specific relational graphs, expert knowledge and reasoning of both meaningful structure and semantics is required – suggesting the need for automation.

3.2. SimGNN

SimGNN [9] proposes

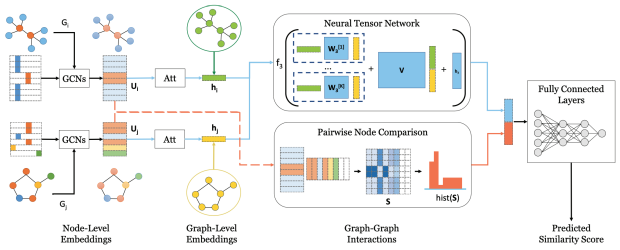
1. Designing a learnable embedding function that maps every graph to an embedding vector.
   A novel attention mechanism is utilized subject to a specific similarity metric
2. Design a pairwise node comparison method to supplement the graph-level embeddings with find-grained node-level information

Approach to speed-up graph-similarity computation by finding approximate values heuristically via a learnable mapping function. Resultant process is quadratic with respect to graph size, which is still among the most efficient methods for graph similarity computation.

Formally, the *Graph Edit Distance* between $G_1, G_2 = GED(G_1, G_2)$, the number of edit operations in the optimal alignments that transform G_1 to G_2, where an edit operation is an insertion or deletion of a vertex/edge or relabelling of a vertex. If two graphs are isomorphic, their GED is 0.

3.2.1. Graph-Level Embedding

Node embedding is generated via Graph Convolutional Network (GCN) specifically:
$conv(\mathbf{u_n}) = f_1(\sum_{m\in\mathcal{N}(n)} \frac{1}{\sqrt{d_n d_m}} \mathbf{u_m W_1^{(1)}} + \mathbf{b_1^{(1)}})$ where $\mathcal{N}(n)$ is the set of one hop neighbors of a node n, n inclusive, d_n is the degree of node n + 1 $\mathbf{W_1^{(1)}} \in \mathbb{R}^{D^l \times D^{l+1}}$ is the l-th layer weight matrix, $\mathbf{b_1^{(1)}} \in \mathbb{R}^{\mathbf{D^{l+1}}}$ is the bias, and $f_1(\cdot)$ is an activation function.
Denote input node embeddings as $U \in \mathbb{R}^{N\times D}, u_n \in \mathbb{R}^D$ is the embedding of node n. Global graph context $c \in \mathbb{R}^D, c = tanh(\frac{1}{N} W_2 \sum_{n=1}^{N} u_n)$, $W_2 \in \mathbb{R}^{D\times D}$ is a learnable weight matrix. For a node n, to imbue awareness of global context c in its attention a_n, take the inner product of the context c and the node embedding Graph Embedding $h \in \mathbb{R}^D$ is the attention weighted sum of the node embeddings: $h = \sum_{n=1}^{N} \sigma(u_n^T c) u_n$.



Use Neural Tensor Networks (NTN) to model the relation between two graph-level embeddings: $g(h_i, h_j) = f_3(h_i^T W_3^{[1:k]} h_j + V \begin{bmatrix} h_1 \\ h_j \end{bmatrix} + b_3)$ where $W_3^{[1:k]} \in \mathbb{R}^{D\times D\times K}$ is a weight tensor, $[\cdot]$ is the concatenation operation. $V \in \mathbb{R}^{K\times 2D}$ is a weight vector, $b_3 \in \mathbb{R}^K$ is a bias vector. K is a hyperparameter controlling the number of interaction scores produced by the model for each graph embedding pair.
Feed these similarity scores into a Multi-Layer Perceptron

(MLP) to one score $\hat{s}_{ij} \in \mathbb{R}$ to be compared directly in the loss function: $\mathcal{L} = \frac{1}{|D|}\sum_{(i,j)\in D}(\hat{s}_{i,j} - s(G_i, G_j))^2$ where D is the set of training graph pairs and $s(G_i, G_j)$ is the ground truth similarity.

3.2.2. Pairwise Node Comparison

To overcome limitations of graph-level embeddings (loss of node feature distribution and graph size) use the node embeddings directly. Consider graphs G_i, G_j with node embeddings $U_i \in \mathbb{R}^{N_i \times D}, U_j \in \mathbb{R}^{N_j \times D}$ with pairwise interaction scores $S = \sigma(U_i U_j^T)$, zero padding the smaller graph $\implies S \in \mathbb{R}^{N \times N}, N = max(N_i, N_j)$. To ensure invariance to graph representations, histogram features $hist(S) \in \mathbb{R}^B$ are extracted, where B is a hyperparameter indicating the number of bins. This histogram feature vector is normalized and concatenated with graph level interaction scores and fed to the fully connected layers.

3.2.3. Evaluation

Experiments conducted on the following datasets:

- *AIDS*: collection of antivirus screen chemical compounds containing 42,687 structures with Hydrogen omitted. Of these 700 were selected, having ≤ 10 nodes.
- *LINUX*: 48,747 Program Dependence Graphs (PDG) generated from the Linux kernel. Each represents a function, where a node represents a statement and an edge represents the dependency of the statements. (10 nodes or less)
- *IMDB*: 15000 ego-networks of actors/actresses, edges determined if they costared in the same film.

Shown here are the results for the *LINUX* dataset across the *Spearman's Rank Correlation Coefficient* (ρ), *Kendall's Rank Correlation Coefficient* (τ), and *precision at k* (p@k) – computed by taking the intersection of the top k predicted and ground truth values divided by k. It can be observed that SimGNN exhibits the best performance on *LINUX* across all metrics.

Table 4: Results on LINUX.

Method	mse(10^{-3})	ρ	τ	p@10	p@20
Beam	9.268	0.827	0.714	**0.973**	0.924
Hungarian	29.805	0.638	0.517	0.913	0.836
VJ	63.863	0.581	0.450	0.287	0.251
SimpleMean	16.950	0.020	0.016	0.432	0.465
HierarchicalMean	6.431	0.430	0.525	0.750	0.618
HierarchicalMax	6.575	0.879	0.740	0.551	0.575
AttDegree	8.064	0.742	0.609	0.427	0.460
AttGlobalContext	3.125	0.904	0.781	0.874	0.864
AttLearnableGC	2.055	0.916	0.804	0.903	0.887
SimGNN	**1.509**	**0.939**	**0.830**	**0.942**	**0.933**

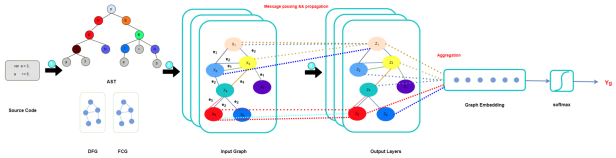
3.2.4. Future Directions

Work can be done to address the integration of edge features and scalability to larger graphs.

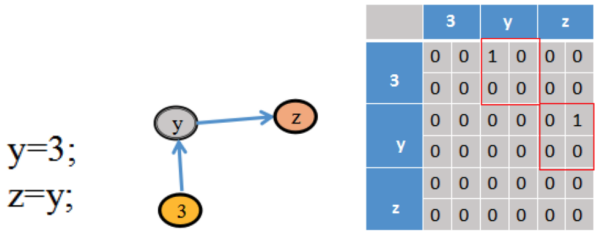
3.3. Gated Graph Attention Neural Network (GGANN)

Large scale online programming source code platforms such as Github are rapidly increasing both the number and size of their repositories – necessitating the ability to mine large codebases. Proposed is the use of Abstract Syntax Trees (AST) to encode the syntactical structure of source code, even attaching data flow edges to encode semantic program meaning in the local scope. To extend this concept to coarse-grained and global function-call relations, Gated Graph Attention Neural Networks [48] are proposed. Contributions can be summarized as follows:

- Program graph which integrates both data flow and function-call information into AST to characterize syntax and semantics
- Improvements made to GGNN model by introducing attention mechanism



Directed graphs are represented as $G = (V, E)$ with edge type set $L_K = l_1, l_2, \ldots, l_k$, connection matrix $A \in \mathbb{R}^{|V| \times 2|V|}$, where the element A_{ij} is a $d \times d$ matrix, where d represents the feature dimension of a node.



(a) Example (b) Data flow graph (c) Connection matrix

Directed graphs are represented as $G = (V, E)$ with edge type set $L_K = l_1, l_2, \ldots, l_k$, connection matrix $A \in \mathbb{R}^{|V| \times 2|V|}$, where the element A_{ij} is a $d \times d$ matrix, where d represents the feature dimension of a node. Neighbor state aggregation: $m_i^{(t)} = \sum_{j \in N_i} A_{ij} \cdot h_j^{(t-1)}$ with per node state information: $h_i^{(t)} = GRU(m_t^{(t)}, h_i^{(t-1)})$

3.3.1. Program Graph Construction

Program graph construction consists of the construction of the AST, Function Call Graph (FCG), and Data Flow Graph

(DFG). AST is as described in preceding sections. Each node in FCG represents a function, while each edge denotes a function-call relation between two functions. A node in DFG can be an entity, such as a variable, operator, structure identifier, etc., while an edge represents the data transfer between two entities.

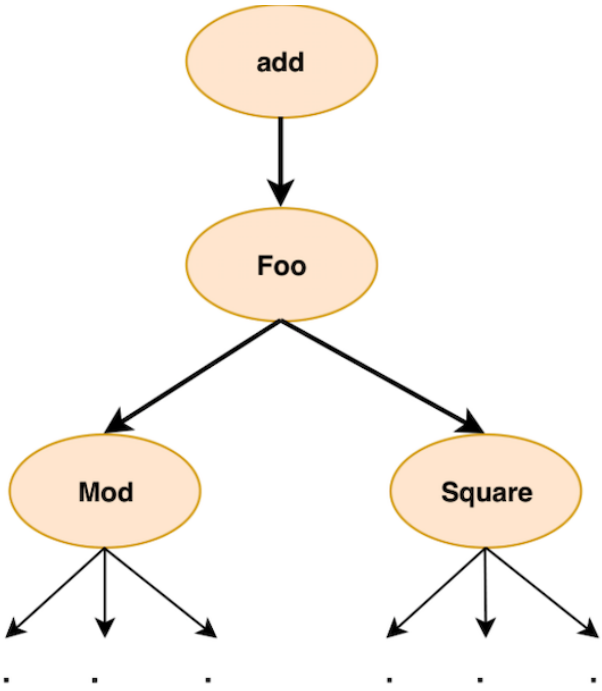
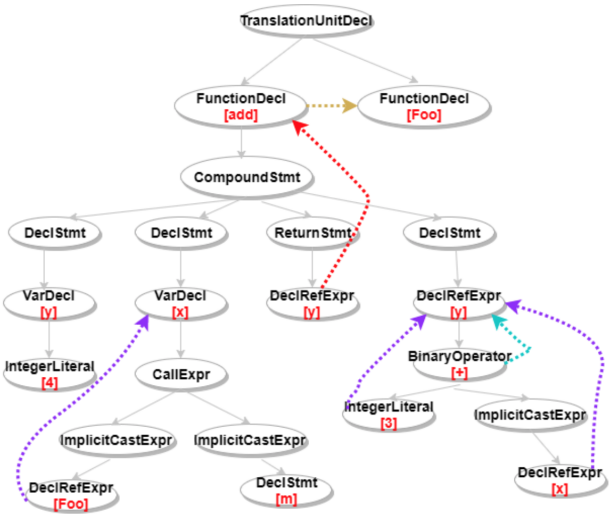


Figure 1. FCG example

It is desireable to integrate these representations into a new program graph that encapsulates their joint information: called the integrated FDA graph.



In summary, the node information aggregation is as follows:

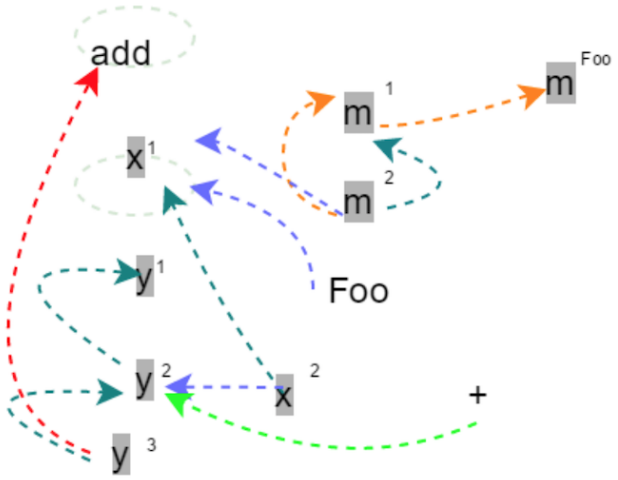


Figure 2. DFG example

$m_i^{(t)} = \sum_{j \in N_i} \alpha_{ij} A(h_{ij}^{'(t)}) h_j^{(t-1)}$ with hidden state of directed edge e_{ij}, $h'_{ij} = U_e(h_{ij}^{'(t-1)}, h_i^{(t-1)}, h_j^{(t-1)})$ where U_e is a multi-layer neural network. Propagation matrix A_{ij} can be replaced with a propagation matrix function, $A(h'_{ij}) : \mathbb{R}^d \to \mathbb{R}^{d \times d}$ which learns the propagation matrix dynamically according to the dynamic information of the hidden state, realized by a multilayer neural network. Information strength $\alpha_{ij} = softmax(a(h_i^{(t-1)}, h_j^{(t-1)}))$, where $a : \mathbb{R}^{d \times d} \to \mathbb{R}$ is an attention mechanism. Vector representation for the FDA graph is unknown before calculation of the node's contribution, so it is approximated from the updated hidden state of each node after T iterations of information propagation for sufficiently large T. Embedding vector of the entire FDA graph can be evaluated by computing the weighted sum of node embeddings and an activation. $h_G = softmax(\sum_{i \in V} f(h_i^{(T)}, x_i) \odot g(h_i^{(T)}))$

3.3.2. Experimental Evaluation

Experiments were conducted on the Online Judge (OJ) programming source code files in C++, to be evaluated on classification accuracy, semantic analysis of learned representation, and attention analysis.

TABLE II: The program classification accuracy of different models on the similar programming tasks.

Accuracy	Average	Minimum	Maximum
Similarity(GGANN)	**93.9%**	**88.7%**	**97.7%**
Similarity(GGNN)	90.3%	84.2%	94.8%
Similarity(TBCNN)	85.7%	83.2%	87.9%

Node representations similarity was evaluated using K-means to see if similar nodes clustered into meaningful groups.

The attention mechanism was visualized via

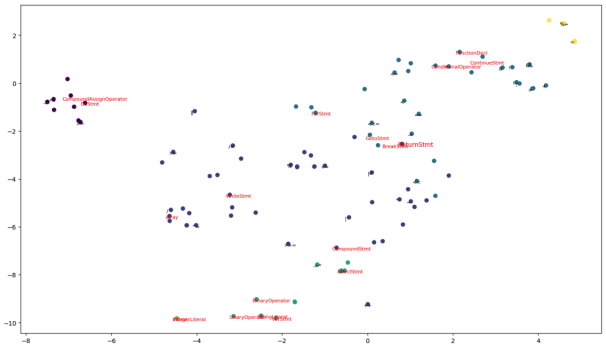
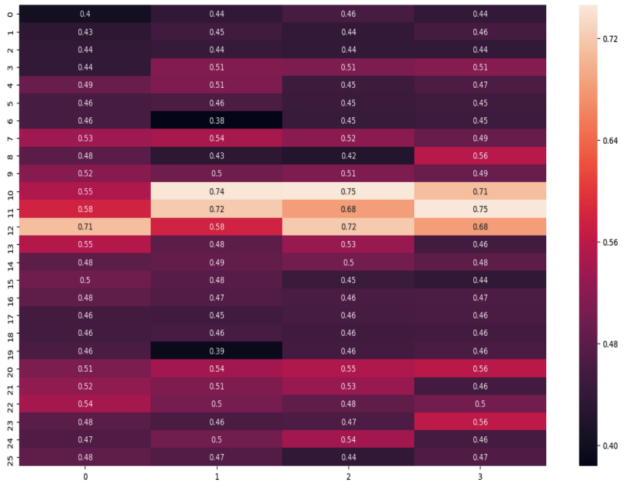


Fig. 13: A t-SNE plot of the learned node representations, where different node colors denote different clusters.

Figure 3. Attention scores of nodes when merged to graph representation

a heat map of the node attentions when aggregated to entire graph representations.



3.4. Graph Matching Networks (GMN)

Instead of computing graph representations independently for each graph, GMNs compute a similarity score through a cross-graph attention mechanism to associate nodes across graphs and identify differences. Proposed models are evaluated on three tasks: a synthetic graph edit-distance learning task which captures structural similarity exclusively, and binary function similarity search and mesh retrieval, requiring added semantic similarity. The contributions of the paper can be summarized as follows:

- Demonstration of how GNNs can be used to produce graph embeddings for similarity learning
- Proposal of new Graph Matching Networks that compute similarity through cross-graph attention based matching

3.4.1. Graph Embedding Models

This GNN embedding model consists of an encoder, propagation layers, and an aggregator.
Encoder maps the node and edge features to initial node and edge vectors through separate
MLPs: $h_i^{(0)} = MLP_{node}(x_i)\ \forall i \in V$
$e_{ij} = MLP_{edge}(x_{ij}),\ \forall (i,j) \in E$
Propagation Layers
$m_{j\to i} = f_{message}(h_i^{(t)}, h_j^{(t)}, e_{ij})$
$h_i^{(t+1)} = f_{node}(h_i^{(t)}, \sum_{j:(j,i)\in E} m_{j\to i})$ where $f_{message}$ is typically an MLP and f_{node} can either be an MLP or a recurrent neural network core.
Aggregator
After T rounds of propagations, we compute the following graph level representation
$h_G = MLP_G(\sum_{i\in V} \sigma(MLP_{gate}(h_i^{(T)})) \odot MLP(h_i^{(T)}))$
which transforms node representations and then uses a weighted sum with gating vectors to aggregate across nodes.

3.4.2. Graph Matching Networks

In this network, node updates will take into account aggregated messages at an inter/intra graph perspective, defined as follows $m_{j\to i} = f_{message}(h_i^{(t)}, h_j^{(t)}, e_{ij}), \forall (i,j) \in E_1 \cup E_2$
$\mu_{j\to i} = f_{match}(h_i^{(t)}, h_j^{(t)}), \forall i \in V_1, j \in V_2$, or $i \in V_2, j \in V_1$
$h_i^{(t+1)} = f_{node}(h_i^{(t)}, \sum_j m_{j\to i}, \sum_{j'} \mu_{j'\to i})$
$h_{G_1} = f_G(h_i^{(T)}{}_{i\in V_1})$
$h_{G_1} = f_G(h_i^{(T)}{}_{i\in V_2})$
$s = f_s(h_{G_1}, h_{G_2})$
where f_s is a standard vector space similarity and f_{match} is a function that incorporates an attention based module:
$a_{j\to i} = \frac{exp(s_h(h_i^{(t)}, h_j^{(t)}))}{\sum_j exp(s_h(h_i^{(t)}, h_j^{(t)}))}$
$\mu_{j\to i} = a_{j\to i}(h_i^{(t)} - h_j^{(t)})$ and therefore,
$\sum_j \mu_{j\to i} = \sum_j a_{j\to i}(h_i^{(t)} - h_j^{(t)}) = h_i^{(t)} - a_{j\to i} h_j^{(t)}$ with attention weights $a_{j\to i}$ and $\sum_j \mu_{j\to i}$ intuitively measures the difference between $h_i^{(t)}$ and its closest neighbor in the other graph. The matching network has the ability to adjust its graph representations based on the compared graph, amplifying disparities if they do not match pairwise.

3.4.3. Learning

If similarity learning model is trained pairwise, samples must be labeled as similar and dissimilar, whereas triple training needs only relative similarity between the three samples. Loss functions are defined respectively: $L_{pair} = \mathbb{E}_{(G_1,G_2,t)}[max\{0, \gamma - t(1 - d(G_1, G_2))\}]$
$L_{triplet} = \mathbb{E}_{(G_1,G_2,G_3)}[max\{0, d(G_1, G_2) - d(G_1, G_3) +$

$\gamma\}]$ where t is a binary similarity label.

3.4.4. Evaluation

Overall results demonstrate that GMNs excel on graph similarity learning and consistently outperform other approaches, shown here is the binary function similarity search. The model was evaluated on data generated by compiling an open source vido processing software `ffmpeg` using different compilers `gcc` and `clang` with various configurations.

Model	Pair AUC	Triplet Acc
Baseline	96.09	96.35
GCN	96.67	96.57
Siamese-GCN	97.54	97.51
GNN	97.71	97.83
Siamese-GNN	97.76	97.58
GMN	**99.28**	**99.18**

Function Similarity Search

3.5. Neural Code Comprehension [12]

3.5.1. Introduction

A novel processing technique to learn code semantics, and apply it to a variety of program analysis tasks – based on an Intermediate Representation (IR), leveraging underlying data- and control-flow of programs.

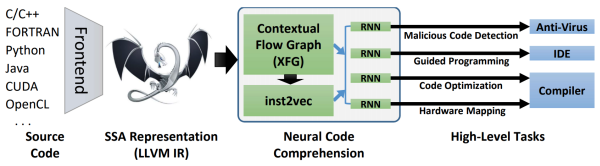


Figure 4. Component overview of Neural Code Comprehension pipeline

Naturalness Hypothesis: Software is a form of human communication; software corpora have similar statistical properties to natural language corpora; and these properties can be exploited to build better software engineering tools This work's contributions are as follows:

- Formulation of a robust distributional hypothesis for code, from which a novel distributed representation of code statements based on contextual flow and LLVM IR is drawn.
- conteXtual Flow Graphs (XFGs): representation designed for statement embeddings that combine data and control flow
- Evaluation using clustering, analogies, semantic tests, and tasks
- Using simple LSTM architectures to show leading performance.

Robust Distributional Hypothesis of Code can be paraphrased as follows: Statements that occur in the same contexts tend to have similar semantics.

3.5.2. Contextual Flow Processing

XFGs are directed multigraphs that provide a notion of context, where nodes (variables or label identifiers) can be connected by more than one edge (data-dependence or execution dependence). *XFG Construction*

1. Read LLVM IR statements once, storing function names and return statements
2. Second pass over the statements, adding nodes and edges according to the rule-set:
   (a) Data dependencies within a basic block are connected
   (b) Inter-block dependencies are both connected directly and through the label identifier
   (c) Identifiers without a dataflow parent are connected to their root (label or program root)

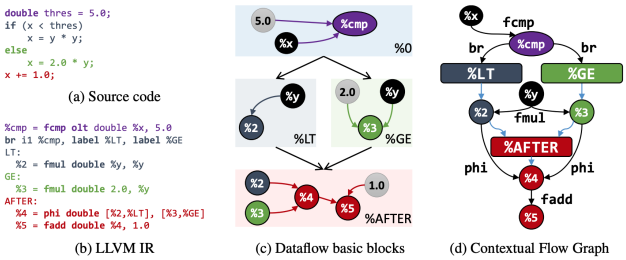


Figure 5. Contextual flow processing scheme

3.5.3. inst2vec: Embedding Statements in Continuous Space

Desiderata for an embedding space include: (a) statements that are in close proximity should have similar artifacts on a system (i.e. use the same resources); and (b) changing the same attributes (e.g. data type) for different instructions should result in a similar offset in the space.

```
store float %250, float* %82, align 4, !tbaa !1
%10 = fadd fast float %9, 1.3
%8 = load %"struct.aaa"*, %"struct.aaa"** %2
```
(a) LLVM IR

```
store float %ID, float* %ID, align 4
%ID = fadd fast float %ID, <FLOAT>
%ID = load { float, float }*, { float, float }** %ID
```
(b) `inst2vec` statements

Figure 5: Before and after preprocessing LLVM IR to `inst2vec` statements.

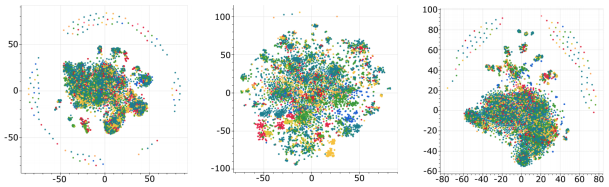
`inst2vec` was generated using corpora from different disciplines, even synthetically generated programs. Given

a set of XFGs, neighboring statement pairs up to a certain context size via the skip-gram model are generated. Pairs are obtained by constructing a dual graph in which statements are nodes, omitting duplicate edges.

Table 1: `inst2vec` training dataset statistics

Discipline	Dataset	Files	LLVM IR Lines	Vocabulary Size	XFG Stmt. Pairs
Machine Learning	Tensorflow [1]	2,492	16,943,893	220,554	260,250,973
High-Performance Computing	AMD APP SDK [9]	123	1,304,669	4,146	45,081,359
	BLAS [22]	300	280,782	566	283,856
Benchmarks	NAS [57]	268	572,521	1,793	1,701,968
	Parboil [59]	151	118,575	2,175	151,916
	PolybenchGPU [27]	40	33,601	577	40,975
	Rodinia [14]	92	103,296	3,861	266,354
	SHOC [21]	112	399,287	3,381	12,096,508
Scientific Computing	COSMO [11]	161	152,127	2,344	2,338,153
Operating Systems	Linux kernel [42]	1,988	2,544,245	136,545	5,271,179
Computer Vision	OpenCV [36]	442	1,908,683	39,920	10,313,451
	NVIDIA samples [17]	60	43,563	2,467	74,915
Synthetic	Synthetic	17,801	26,045,547	113,763	303,054,685
Total (Combined)	—	24,030	50,450,789	8,565	640,926,292

3.5.4. Evaluation



(a) Context size = 1 (b) Context size = 2 (c) Context size = 3

Figure 6. t-SNE plots for learned embeddings

Using `inst2vec` an RNN was constructed that reads source code and outputs a predicted program class on the POJ-104 dataset, collected from the Pedagogical Open Judge System.

Table 3: Algorithm classification test accuracy

Metric	Surface Features [49] (RBF SVM + Bag-of-Trees)	RNN [49]	TBCNN [49]	inst2vec
Test Accuracy [%]	88.2	84.8	94.0	**94.83**

Figure 7. Algorithm classfication test accuracy

4. Program synthesis

4.1. 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 suevey a metalearning framework that learns a transferable policy using only weak supervision. The program synthesizer takes as input a logical formula φ and a grammar G, and produces as output a program in G that satisfies φ. In this formulation, φ 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.

4.2. 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 [[28]; [54]], data wrangling [[24]], data processing [[63]], syntax transformations [[56]], database queries [[71]] and bit-vector manipulations [[37]]. 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 [[64]] is a general framework for program synthesis that synthesizes programs satisfying a specification ϕ. 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 [[7]]. 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 [[6]]. Early works approach the task by generating code as sequences of tokens [[32]], whereas newer methods have focused on leveraging the known target grammar and generate code as trees [[50]]. 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 [[10]], Bayou [[52]], RobustFill [[21]], Differentiable FORTH [[13]], neuro-symbolic program synthesis [[16]], neural-guided deductive search [[39]], learning context-free parsers [[17]], and learning program invariants [[61]]. The syntactic specification in these approaches is fixed by defining a domain-specific language upfront. Also, with the exception of [61], 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.

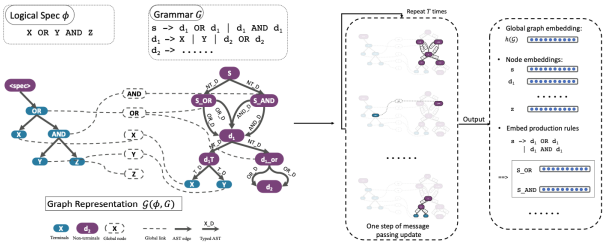
One class of machine learning approaches predict programs directly using neural networks [[54]] while another employs neural networks to guide the symbolic techniques above [[38]]. In both cases, the models involved take the input-output example directly and make predictions accordingly. However the domains tackled so far are simpler; [10]; [21]; [38] 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 [16] are also not applicable.

Graph Neural Networks [20] [2017]; [43]. [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.

4.3. 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.

4.3.1. 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 ϕ 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 = $(\phi_i, G_i)N_i$=1, The following two tasks are addressed: 1) learning an algorithm $A_\theta : (\phi, G) \rightarrow f$ parameterized by θ 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_θ 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 φ 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 Aθ 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.

4.3.2. Formal Definition

semantic spec φ The spec itself is a program written using some grammar. The grammar used in spec φ 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 = ⟨V , E, R, s⟩. 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 $\alpha \rightarrow \beta_1|\beta_2....|\beta_{n_\alpha}$, where n_α is the branching factor for non-terminal $\alpha \in V$, and β_i = $u_1u_2 \ldots u_{|\beta_i|} \in$ (V ∪ E). Each production rule $\alpha \rightarrow \beta_1 \in$ R represents a way of expanding the grammar tree, by attaching nodes $u_1u_2 \ldots u_{|\beta_i|}$ to node α.

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 φ.

4.3.3. Task Representation

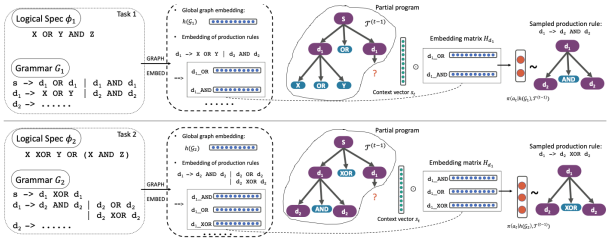
Semantic spec program φ and the CFG G have rich structural information.To construct the graph, the abstract syntax tree (AST) is built first for the semantic spec program φ, according to its own grammar (typically different from the output grammar G). To represent the grammar G, each symbol in V → E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

As a simplification, all nodes α_i is merged representing β_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(φ, G), graph neural networks is used to get the vector representation for each node in the graph. Specifically, for each node v ∈ G, the following parameterization for one step of message passing style update:

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

Lastly, $h_{v v \in G}^t$, $h_v^t \in r^d$ 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∈ R^{dxd} 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 α, the embedding matrix is $H_\alpha \in R^{n_\alpha xd}$, where the ith row of $H_\alpha^{(i)}$ is the embedding of node α_i that corresponds to substitution β_i.

4.3.4. 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 $H_\alpha \in R^{n_\alpha xd}$ is used to perform decision for this time step. This allwos us to build policy network in an auto-regressive way. Specifically, the policy $\pi(f|\phi, G)$ can be parameterized as:$\pi(f|\phi, G) = \prod_{t=1}^{|b|} \pi(a_t|h(G), T^{t-1}$, where $T^{t-1} = \alpha_1...\alpha_{t-1}$ 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_{alphat} . The initial state s_0 is obtained by passing graph embedding h(G) through a dense layer with matching size.

4.3.5. Solving Via Reinforced Learning

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

Reward design The RL episode starts by accepting the representation of tuple ⟨ϕ, G⟩ 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 ϕ or not. To smooth the reward: for each specification ϕ the solution maintains a test case buffer B_ϕ that stores all input examples observed so far. Each time the SAT solver is invoked for ϕ, if f passes then a full reward of 1 is given, otherwise the solver will generate a counter-example 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_ϕ and B_b where f has the equivalent output as ϕ

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

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(ϕ_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 θ of each instance as

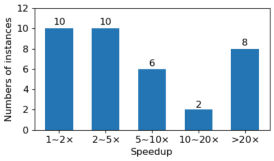
$$d\theta \leftarrow \sum_{t=1}^{|f|} \bigtriangledown_\theta log\pi(\alpha_t|h(G), T^{(t)})(\gamma^{|f|-t}r - V(s_t; w)),$$

where γ 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.

4.3.6. Results

# candidates generated	Percentage solved		
	20%	40%	60%
Out-of-Box Solver	2564	18K	102K
Meta-Solver	205	4.5K	59K
Reduction	12.5×	4.0×	1.7×

(a)



(b)

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.

4.4. ExprGen

4.4.1. 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 a 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.

4.4.2. Graph Decoding for source code

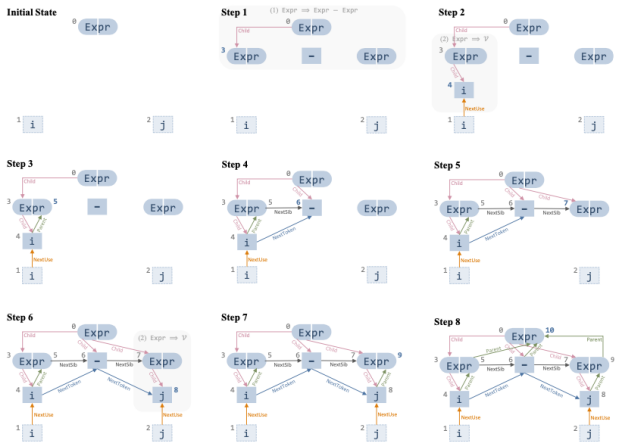
Tasks include to encode the code context c, $v_1, ..., v_l$ and to construct a model that can learn p$(a_t|c, a_{<t})$ well. Both these encoders yield a distributed vector representation for the overall context, representations $h_{t_1}, ..., h_{t_T}$ for all tokens in the context, and separate representations for each of the in-scope variables $v_1, ..., v_l$, 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

Edge set $\mathcal{E} \leftarrow \varnothing$
if v is inherited **then**
  $\mathcal{E} \leftarrow \mathcal{E} \cup \{\langle \mathsf{parent}(a, v), \mathit{Child}, v\rangle\}$
  if v is terminal node **then**
    $\mathcal{E} \leftarrow \mathcal{E} \cup \{\langle \mathsf{lastToken}(a, v), \mathit{NextToken}, v\rangle\}$
    if v is variable **then**
      $\mathcal{E} \leftarrow \mathcal{E} \cup \{\langle \mathsf{lastUse}(a, v), \mathit{NextUse}, v\rangle\}$
  if v is not first child **then**
    $\mathcal{E} \leftarrow \mathcal{E} \cup \{\langle \mathsf{lastSibling}(a, v), \mathit{NextSib}, v\rangle\}$
else
  $\mathcal{E} \leftarrow \mathcal{E} \cup \{\langle u, \mathit{Parent}, v\rangle \mid u \in \mathsf{children}(a, v)\}$
  $\mathcal{E} \leftarrow \mathcal{E} \cup \{\langle \mathsf{inheritedAttr}(v), \mathit{InhToSyn}, v\rangle\}$
return $\mathcal{E}$

Notion as graphs where nodes u, v, . . . 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 τ 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 parent(a, v) and 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) : Expr $\implies$ V, choosing i for the first variable and j for the second variable.

Edges in $\mathbf{a}_{<t}$ - 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 emb(l_v) of the node label l_v using a function g:

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

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 nonterminal node v with label l_v and inherited attributes h_v, pickProduction(l_v,h_v) = argmax P(rule —l_v,h_v) = argmax[e(h_v)+m_{l_v}]. (3) Here, m_{l_v} is a mask vector whose value is 0 for valid productions $l_v \implies$ ... 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 pickVariable(V,h_v) = argmax$_{var\in V}$ P(var—h_v) = argmax$_{var\in V}$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 $t_1 ... t_T$ from the context.Thus,to pick a literal at node v, and define pickLiteral(V,h_v) = argmax$_{lit\in L\cup t_1...t_T}$ P(lit—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 P (lit — 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 ex-

panded node v as a key and the context token representations $h_{t_1}, ..., h_{t_T}$ 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 porivded. 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:

$h_v = g(emb(l_v), \sum_{<u_i,t_i,v>\in E_v} f_{t_i}(h_{u_i}) + \sum_{<u_i,t_i,l_i,v>\in E_v^l} f_{t_i}(h_{u_i}, emb_e(l_i)))$

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 Expr $\implies$ 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.

4.4.3. 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†	–	–	*34.8%*	*42.9%*	–	–	*28.0%*	*37.3%*
$Seq \rightarrow Seq$	87.48	32.4%	21.8%	28.1%	130.46	23.4%	10.8%	16.8%
$Seq \rightarrow \mathcal{NAG}$	6.81	53.2%	17.7%	33.7%	8.38	40.4%	8.4%	15.8%
$\mathcal{G} \rightarrow Seq$	93.31	40.9%	27.1%	34.8%	28.48	36.3%	17.2%	25.6%
$\mathcal{G} \rightarrow Tree$	4.37	49.3%	26.8%	48.9%	5.37	41.2%	19.9%	36.8%
$\mathcal{G} \rightarrow \mathcal{ASN}$	2.62	78.7%	45.7%	62.0%	3.03	74.7%	32.4%	48.1%
$\mathcal{G} \rightarrow Syn$	2.71	84.9%	50.5%	66.8%	3.48	84.5%	36.0%	52.7%
$\mathcal{G} \rightarrow \mathcal{NAG}$	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

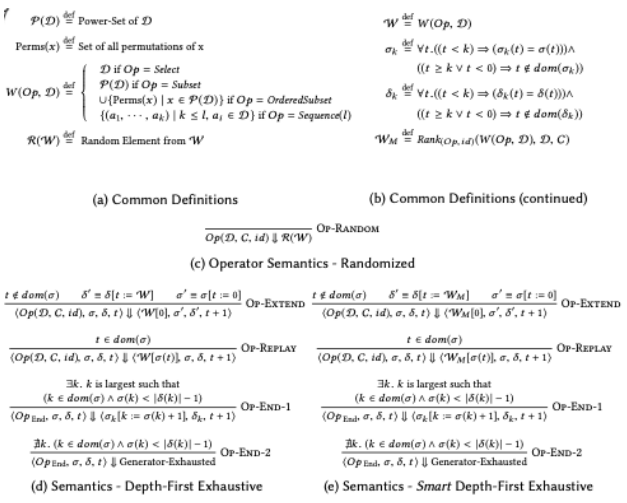
4.5. AutoPandas

4.5.1. 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 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 , σ and δ . The variable t keeps track of the number of operator calls made in the current invocation of the generator. The variable σ 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. δ 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 σ and δ 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 $\notin$ dom(σ)). The operator returns the first value from W and records this choice in σ . It also stores W in δ for future use. (2) Op-Replay - The hypothesis t $\in$ dom(σ) signifies that this operator call needs to replay the choice as dictated by σ (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 σ . 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 (¿ k) from σ and δ. 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 σ 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

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

possible executions of the generator.



Generator-Based Program Synthesis To build an enumerative synthesis engine E using generators.The engine consists 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 Ge 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 Ge 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 Ge . 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 Ge 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 Ge 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_es 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, itis 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 $\Gamma_e \equiv$ 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. $f_k : R^{D+|\tau_e|}$ßR^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 $h_v^{(k+1)}$ = 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 . 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 di for di 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 though 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 $(p_1, , p_n)$. 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 though 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 $p_{i_k} = 1 \wedge$ p_{e_k} =0 if element d_k is present in the correct output,else $p_{i_k} = 0 \wedge$ 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 ot 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 ($a_0, , a_k$) 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.

4.5.2. RESULTS

Benchmark	Depth	Candidates Explored		Sequences Explored		Solved		Time(s)	
		AUTOPANDAS	BASELINE	AUTOPANDAS	BASELINE	AUTOPANDAS	BASELINE	AUTOPANDAS	BASELINE
SO_11881165	1	**15**	64	1	1	Y	Y	**0.54**	1.46
SO_11941492	1	783	**441**	8	8	Y	Y	12.55	**2.38**
SO_13647222	1	**5**	15696	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	**10**	208	1	1	Y	Y	2.24	**2.01**
SO_11418192	2	158	**80**	1	1	Y	Y	**0.71**	1.46
SO_49567723	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_14023037	3	-	-	-	-	N	N	-	-
SO_53762029	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_39656670	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.

5. Bug Detection

5.1. 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.

5.2. Related work

The most closely related work is GNNs. [51] 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 ([68]); 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., [8]; [45]; [65]) 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.

5.3. 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/ggnn.

5.3.1. Graph Representation- GNN and GATED

GNNs are a general neural network architecture defined according to a graph structure G = (V,E). Nodes v ∈ V take unique values from 1, . . . , $|V|$, and edges are pairs e = (v, v′) ∈ V × V. The authors focus in this work on directed graphs, so (v, v′) represents a directed edge v → v′, but note that the framework can easily be adapted to undirected graphs; see [59]. The node vector (or node representation or node embedding) for node v is denoted by $h_v \in R^D$. Graphs may also contain node labels $l_v \in 1,...,L_V$ for each node v and edge labels or edge types $l_e \in 1,...,L_E$ 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′ —(v′,v) ∈ E returns the set of predecessor nodes v′ with v′ →v. Analogously, OUT(v) = v′ — (v, v′) ∈ E is the set of successor nodes v′ with edges v → v′. The set of all nodes neighboring v is NBR(v) = IN(v) ∪ OUT(v), and the set of all edges incoming to or outgoing from v is CO(v)=(v′,v′′)∈E|v=v′Vv=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 ∈ 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.

5.3.2. 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 [59] including positional graph forms, node-specific updates, and alternative representations of neighborhoods. Concretely, [59] suggest decomposing $f_*(\dot{})$ to be a sum of per-edge terms.

$$\mathbf{h}_v^{(1)} = [\boldsymbol{x}_v^\top, \mathbf{0}]^\top \quad (1)$$

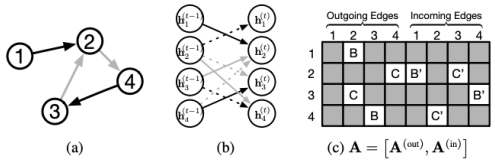
$$\mathbf{a}_v^{(t)} = \mathbf{A}_{v:}^\top \left[\mathbf{h}_1^{(t-1)\top} \dots \mathbf{h}_{|V|}^{(t-1)\top}\right]^\top + \mathbf{b} \quad (2)$$

$$\mathbf{z}_v^t = \sigma\left(\mathbf{W}^z \mathbf{a}_v^{(t)} + \mathbf{U}^z \mathbf{h}_v^{(t-1)}\right) \quad (3)$$

$$\mathbf{r}_v^t = \sigma\left(\mathbf{W}^r \mathbf{a}_v^{(t)} + \mathbf{U}^r \mathbf{h}_v^{(t-1)}\right) \quad (4)$$

$$\widetilde{\mathbf{h}_v^{(t)}} = \tanh\left(\mathbf{W}\mathbf{a}_v^{(t)} + \mathbf{U}\left(\mathbf{r}_v^t \odot \mathbf{h}_v^{(t-1)}\right)\right) \quad (5)$$

$$\mathbf{h}_v^{(t)} = (1 - \mathbf{z}_v^t) \odot \mathbf{h}_v^{(t-1)} + \mathbf{z}_v^t \odot \widetilde{\mathbf{h}_v^{(t)}}. \quad (6)$$



The matrix A ∈ $R^{D|V|x2D|V|}$ 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. $A_v \in R^{D|V|x2D|V|}$ 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.

5.3.3. 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 ∈ 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 ∈ 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.

5.3.4. Gated Graph Sequence Neural Networks

several GG-NNs operate in sequence to produce an output sequence $o^{(1)}...o^{(K)}$. For the k^{th} output step, the matrix of node annotations is denoted as $X^{(k)} = [x_1^{(k)}; ...; x_{|V|}^{(k)}]^T \in R^{|V|L_V}$. 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 $H^{(k,t)} = [h_1^{(k,t)}; ...; h_{|V|}^{(k,t)}]^T \in R^{|V|D}$. 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:

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

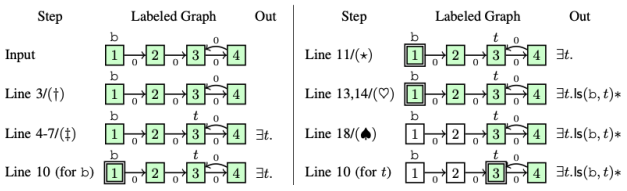
There are two settings for training GGS-NNs: specifying all intermediate annotations $X^{(k)}$, or training the full model end-to-end given only $X^{(1)}$, graphs and target sequences. The former can improve performance when the suthors have domain knowledge about specific intermediate information that 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.

5.3.5. 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) ≡ $x = y \vee \exists v$,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 ∧ 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 ∃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 verifica-

tion 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 $v_0, ..., v_k$ is stored. Graph edges reflect these pointer values, i.e., v has edges labeled with 0, . . . , k that point to nodes $v_0, ..., v_k$, 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 $\exists x_1, ..., x_n.a_1 * ... * a_m$, 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
$\mathcal{X} \leftarrow$ compute initial labels from $\mathcal{G}$
$\mathcal{H} \leftarrow$ initialize node vectors by 0-extending $\mathcal{X}$
while $\exists$ quantifier needed **do** ▷ **Graph-level Classification** (†)
  $t \leftarrow$ fresh variable name
  $v \leftarrow$ pick node ▷ **Node Selection** (‡)
  $\mathcal{X} \leftarrow$ turn on "is-named" for v in $\mathcal{X}$
  print "$\exists t$."
end while
for node v_ℓ with label "is-named" in $\mathcal{X}$ **do**
  $\mathcal{H} \leftarrow$ initialize node vectors, turn on "active" label for v_ℓ in $\mathcal{X}$
  $pred \leftarrow$ pick data structure predicate ▷ **Graph-level Classification** (⋆)
  if $pred =$ ls **then**
    $\ell_{end} \leftarrow$ pick list end node ▷ **Node Selection** (♡)
    print "ls(ℓ, ℓ_{end}) ∗"
  else
    print "$pred(\ell)$ ∗"
  end if
  $\mathcal{X} \leftarrow$ update node annotations in $\mathcal{X}$ ▷ **Node Annotation** (♠)
end for

The authors use the full GGS-NN model where $F_o^{(k)}$ and $F_o^{(k)}$ have separate propagation models for 10 times steps and use D=16 dimensional node represntations. 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 togeter, which is the key for aggregating predictions in a batch. Compute the score for a particular named variable t as $o_t = \sum_g o^g_{V_g(t)}$, where $V_g(t)$ maps variable name t to a node in graph g, and $o^g_{V_g(t)}$ 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(toselect = t) = $\prod_g p_g(toselect = V_g(t))$. For graph-level classification outputs, add up scores of a particular class across the batch of graphs, or equivalently compute p(class = k) =$\prod_g p_g$(class = k). Node annotation outputs are updated for each graph independently as different graphs have completely different set of nodes.

5.3.6. RESULTS.

On comparision 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.

5.4. 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 it 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.

5.4.1. 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:

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

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:

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

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

where n_k is the number of node in layer k, sigm(c) = $1/(1 + x^{-c})$, 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, η 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, Wij 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.

5.4.2. 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, DBNis 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

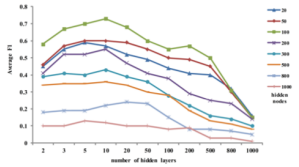


Figure 5: Defect prediction performance with different parameters

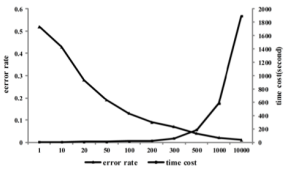


Figure 6: Average error rate and time cost for different

6. Decompilation

6.1. Introduction

Compiled languages are languages whose source code is compiled into machine understandable code by compilers. This machine code can directly be read and executed by the processor. This low-level machine code can either be in binary form or some other intermediate form. Due to the compilation process, these languages are generally faster and more efficient in addition to giving fine-grained control over hardware variables like memory, CPU and GPU, etc. Most popular compiled languages include Java, C, C++, Go, Rust, etc. The process of converting high level source code into low level machine code or other intermediate formats is called Compilation. This is achieved by compilers and is pretty straight forward once we have access to the language specific compiler.

Naturally, the inverse process involves taking machine code in binary format and retrieving the source code. This inverse process is called Decompilation. This process is not so trivial and involves many complex challenges. The semantics of the high level programming language are often lost during compilation. Fu et al. [26] state that there are both intra-statement and inter-statement dependencies in the low-level instructions that need to be preserved by the decompiler. Although challenging, Decompilation is necessary in a variety of applications. Security analysis is crucial for low level code, and decompilation can help in areas like malware analysis and vulnerable software patching as presented in Kolbitsch et al. [42] and Yakdan et al. [73]. Moreover, Lee et al. [46] show that malicious attackers can use decompilers to reverse engineer low level code and exploit commercial software. Katz et al. [41] also mention that decompilation can be useful for porting code to different hardware as well as source level analysis and optimization tools. Hence, having efficient and accurate decompilation tools can be of utmost importance.

6.2. Conventional Decompilers

Lots of research have been done for Decompilers over the years [19; 23; 14; 11; 57; 73; 15]. Many rule-based and pattern matching decompilers have been developed like Hex-Rays [31], Phoenix [15] and the most recent one RetDec [44]. These compilers use pre-defined rules to match patterns in the low level code to control flow graphs of the high level code and translate the instructions accordingly. When such patterns are not found, the decompiler uses *goto* statements in C. This is just like a line-to-line translation from low level code to C. Some decompilers like DREAM++ [72; 73] exist which do not use *goto* statements. However, these are still hand crafted.

Such hand-crafted compilers face a variety of problems like:

- Most of these decompilers are handcrafted for a specific source-target language pair. This requires a lot of effort in itself, and this effort has to be replicated for each new source-target pair for which we want to build a decompiler.
- The output high level code fails to preserve the functionality of the low level code in most cases.
- The output high level code is not semantically readable and often lacks human friendliness. It also fails to use abstractions to represent the high level code as humans do. As a result, the reversed code is often hard to interpret and use it in corresponding applications.

6.3. Problem Definition

Fu et al. [26] gives a nice technical definition of the problem which we can reuse here. Let P denote the high level program and τ denote the compiler. Hence, we can get the low level code as $\phi = \tau(P)$. They define the decompiler τ^{-1} that satisfies $\tau(P^{'}) = \tau(P)$ where $P^{'} = \tau^{-1}(\phi)$.

6.4. Data Generation

Data generation for decompilation tasks is relatively trivial due to the free availability of compilers. However, there are still some challenges. Generating source-target pairs can be done by taking the source code and running it through a compiler to get the executable or low level code. Nuances involve selecting the source code, selecting the different architectures for the compiler, selecting whether to produce the executable or the assembly instructions, etc.

Katz et al. [40] suggest creating a database of source-target pairs by compiling programs using a customized compiler based on Clang and LLVM. This compiler pairs AST trees and subtrees of source code with the corresponding low level machine code output by the compiler.They constrain the pairs by adding the following restrictions - 112 or fewer

binary low level tokens and 88 or fewer source code tokens. The programs they use for their dataset are from open source RPMs for Fedora containing C source code. Comprehensive details are provided in their paper.

Katz et al. [41] generate samples for their decompiler to train on using random code samples from a subset of the C programming language. They do this by sampling the grammar of the language. These samples are guaranteed to be syntactically and grammatically correct. They then compile their code samples using the provided compiler. Doing so results in a dataset of matching pairs of statement, one in C and the other in Lll , that can be used by the model for training and validation. Note the two differing approaches here - taking C code from open source repos vs. using sampling to generate C source code.

Fu et al. [26] use a two-step approach to decompilation (which we will discuss later). To build the dataset for stage 1, they randomly generate 50,000 pairs of high level programs with their corresponding compiled low level code. The compilation occurs using clang with all optimizations disabled. The source code programs can be roughly classified into short and long programs with an average length of 15 and 30 each. Their corresponding tree representations have a maximum depth of 3. They also use real world programs like neural network construction programs in pytorch C++ API and Hacker's Delight loop-free programs. The dataset for the next stage is constructed by injecting various types of errors into the high level program. 10-20% token errors are injected. The locations for these errors are sampled randomly using a uniform distribution. Once again, detailed statistics and methods are provided in their paper.

N-Bref [4] assess the performance of their model on various benchmarks by using both generated high level code using their dataset generator as well as code written by humans in the form of Leetcode solutions to Problems (2017). The binary low level code is assembled using the GNU binary utilities to obtain the assembly code. To generate ASTs for the high level C code, the authors implement a parser using clang compiler for python. Their dataset generator for generated samples is built on top of csmith [74] - a code generation tool for compiler debugging. Their dataset generator avoids expression collision (EC). For instance, unary operators like i++ are converted to binary operators like $i = i + 1$ and all while loops are converted to for loops. This improves performance and also reduces confusion. The dataset generator also has several hyperparameters to control the depth, size, nesting, etc of the high level code. Here also the code is compiled with no optimizations. Previous works [15; 46; 47] also disabled optimizations, because many optimizations change variable types and rewrite source code (e.g. loop-invariant optimization) which will result in unfair accuracy evaluations. Their dataset is available to download from https://www.dropbox.com/s/33fop57jjq0wwa9/POJ-104.tar.gz?dl=1.

A common theme across approaches is to take high level source code (mostly in C) and use compilers with some flags to generate the corresponding dataset pairs. Each approach tunes some nuances to generate their own datasets, but at the high level they have a lot of similarities. For future research, it will be nice to have a common benchmark and dataset for decompilation tasks. A common benchmark will aid in unbiased comparison of different methods and will also help more people easily research novel methods once a common dataset is easily available.

6.5. Evaluation

There are many different ways to evaluate decompilation tasks. You could compare the difference between the original source code and the generated source code using something like Levenshtein edit distance (LD) [35]. You could compare the output of the original source code and the generated source code by compiling and then executing the binaries. Other works use BLEU, edit distance ratio, and syntactic correctness ratio [53]. We will take a look at some of the approaches used by papers that use neural methods for decompilation.

Katz et al. [40] use a separate test dataset to evaluate their model. They feed their trained model sequences of binary input code. The model outputs a series of tokens corresponding to the predicted output in the source language C aka the generated C source code. They use two evaluation metrics to assess the quality of the C source code generated by their technique. Since they use RNNs, the first metric, perplexity, is based on the RNN's internal sampled softmax loss function, as described in Jean et al. [36], and is used during RNN training. It is a measure of the accuracy of a probabilistic model in predicting a sample [60]. They use the perplexity measurement as a proxy for RNN performance to determine when to end training. The second metric is based on the Levenshtein edit distance between the sequence of tokens in the ground truth source code associated with a given binary sequence and the sequence of tokens output by the RNN. They use two variations on edit distance. First is to look at straight edit distance between the tokens in the prediction and those in the ground truth snippet. Second, to ensure differences in identifier names and constants do not influence results, they lex both the output translations and ground truth snippets to obtain sequences of token types and perform the edit distance calculation between corresponding sequences of token types. They report the average edit distance for the exact tokens predicted; the average edit distance for the types of tokens predicted; and the percent of predicted token sequences that match the ground truth token sequences perfectly, without post-processing.

On the other hand, Katz et al. [41] uses the deterministic nature of the compiler to evaluate their model. After translating the inputs, for each pair of input i and corresponding translation t (i.e. the decompiled code), they recompile t and compare the output to i. This allows them to keep track of progress and success rates, even when the correct translation is not known in advance. Exact details are described in the paper. Despite holding the ground-truth for the test set (the C used to generate the set), they decided not to compare the decompiled code to the ground-truth because in some cases, different C statements could be compiled to the same low-level code (e.g. the statements x = x + 1 and x++). They believe that their evaluation method allows for such occurrences and is closer to what would be applied in a real use-case.

Fu et al. [26] (Coda) assess the performance of their model on various synthetic benchmarks with different difficulty levels and real-world applications. Given the binary executable as the input, we use an open-source disassembler [2; 3] for MIPS [30] and x86-64 [27] architecture to generate the corresponding assembly code that is fed to the model. They evaluate the performance of the Coda using two main metrics: token accuracy and program accuracy. Token accuracy is defined as the percentage of the predicted tokens in high-level Programming Language that match with the ground-truth ones. Program accuracy is defined as the ratio between the number of predicted programs with 100

NBref [4] uses different metrics for different stages of their model. For the source code generation stage, they use token accuracy and the graph edit distance [58] without enforcing the match between predicted and ground truth AST on graph generation. The metric is fair to evaluate decompilation tasks as we remove all the Expression collision. For the data-type solver stage, they use two metrics: (i) macro accuracy and (ii) micro accuracy.

Often, these metrics are combined and calculated in ways that give the success rates of decompilation in percentages. The evaluation metrics and protocol seem to vary across different neural decompilation approaches. This makes consistent evaluation across different methods difficult. Since this is a fairly recent research area, a standard evaluation benchmark and metric along with the standard dataset proposed earlier will aid in moving research forward in this domain.

Next we will discuss the different neural decompilation approaches at a high level.

6.6. RNN and Neural Machine Translation (NMT) based approaches

RNNs have become very popular for sequential Machine Learning tasks like translation, question answering ,etc [66]. RNNs have feedback loops which permit persistence of data which in turn allows sequential processing [62]. An unrolled recurrent neural network has a natural sequence-like structure which makes it well suited for translation tasks. There are several variations on RNNs including the LSTM [33] which has increasingly become the go-to RNN of choice. LSTMs help preserve long term dependencies and have shown major improvements when dealing with longer sequences. A particular variant of RNNs is the sequence to sequence model or the seq2seq, which takes in an input sequence and generates an output sequence. As you can imaging, such seq2seq models are very useful for tasks like natural language translation [18].

Decompilation can be seen as a natural expansion of the language translation task, but for programming languages. Translation for programming languages can also include translating between two high level languages like Java and Python. But decompilation deals with translation between a high level language and a low level binary executable/machine code.

Katz et al. [40] is one of the first approaches to adapt seq2seq RNN models for decompilation. Their sequence-to-sequence model is composed of two recurrent neural networks: an encoder, which processes the input sequence, and a decoder, which creates outputs. Further, the sequence-to-sequence model contains a hidden state, C, which summarizes the entire input sequence. The output is generated from the decoder by taking in the hidden state and the current input and yielding a probability distribution from an activation function like Softmax. To modify this model for decompilation, they make many changes in all steps of the pipeline including pre-processing, model architecture, post-processing, etc. The first step involves tokenization of the input-output pair i.e. the source-target pair of the high level code/ low level code pair. The appropriate tokenizer is chosen and built based on expert domain knowledge. The authors also use bucketing to separate out different lengths of the input sequence [1]. Each bucket has its own RNN comprising of the encoder and decoder. Exact details can be found in the paper. The pipeline for an input pair looks like this -

- The binary low level code is tokenized and converted to a list of integers corresponding to the indices in the dictionary for the low level code of the tokenizer.
- The list of integers goes through bucketing to choose the relevant bucket and RNN
- The list then goes through the encoder to generate a hidden state for the binary code.
- The decoder then takes this state and outputs a probability distribution representing tokens in the high level

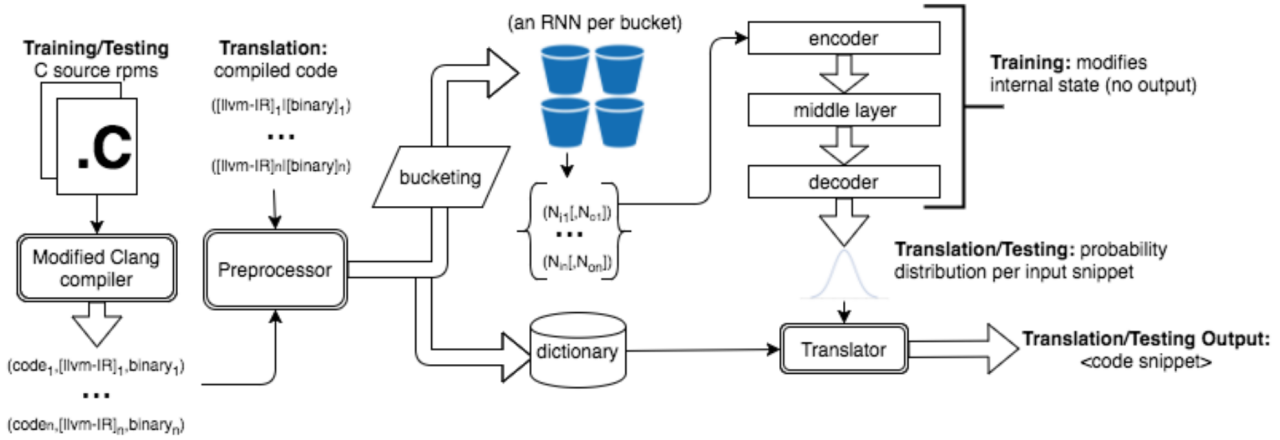


Figure 8. Katz et al. [40]'s Architecture of RNN decompilation system

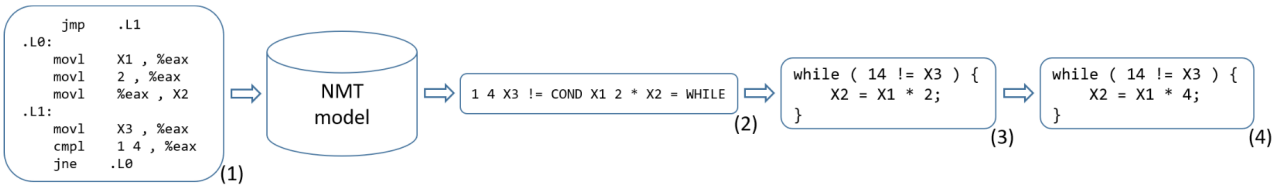


Figure 9. Katz et al. [41]'s two phase pipeline. (1) canonicalized x86 input, (2) NMT output, (3) templated output, (4) final fixed output.

code. This distribution is then converted to a list of integers with the integers corresponding to the indices in the dictionary for the high level code in the tokenizer.

- The tokens for the high level code are retrieved and combined from that dictionary.

This approach is summarized in Fig. 8. It does not take into consideration the differences between natural languages and programming languages, thus leading to ineffective results. For example, as Katz et al. [41] point out, the code they generate often cannot be compiled or is not equivalent to the original source code. This approach did however show the possibility of NMT for decompilation, and served as a stepping stone for the next approach proposed by Katz et al. [41].

Katz et al. [41] combines Neural Machine Translation (NMT) and Program analysis to improve the performance of their Neural Decompilation model and make it more practically useful and viable. They also propose a framework called TraFix that can automatically learn a decompiler from a compiler and check the correctness of the decompiler using a verifier. Their approach consists of two complimentary phases:

- Generating a code template that, when compiled, matches the computation structure of the input. The input is the low level binary code.

- Filling the template with values and constants that result in code equivalent to the input. This means that the output from running the executable will be the same for both cases.

They use a NMT model combined with a feedback loop which facilitates active learning based on the outputs of phase 1. Phase 1 results are compared to the computation structure of the input. If they match, the generated code moves to phase 2. If not, the dataset is updated and the model is trained again. The authors use LSTMs for their NMT model. The two phased pipeline is summarized in Fig. 9.

NMT based techniques are still primarily sequential and fail to incorporate the graph and tree structures of code like control flow graphs and abstract syntax trees. We will take a look at some newer methods which try to incorporate these properties.

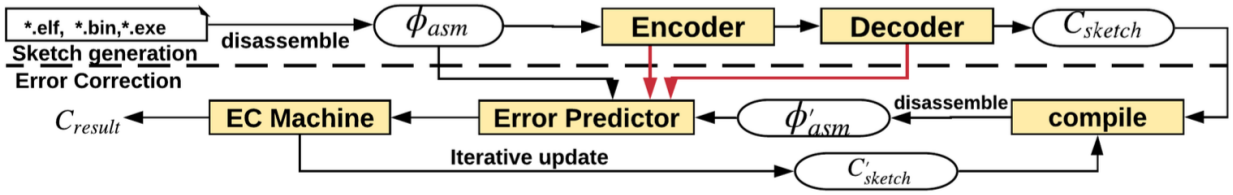


Figure 10. Fu et al. [26]. Coda's end to end pipeline

6.7. Abstract Syntax Tree (AST) based Neural Decompilation

Coda [26] also uses a two phase learning approach and an NMT model at its base. But it tries to incorporate the code structure into the encoder and decoder of the RNN. Its two phase approach can be described as:

- It uses an instruction type-aware encoder and a abstract syntax tree (AST) decoder for translating the input binary into the target PL. Their encoder deploys separate RNNs for different types of statements, thus the statement boundaries are preserved (This is different from bucketing used by Katz et al. [41]). Furthermore, the control and data dependency in the input program are translated to the connections between the hidden states of corresponding RNNs. The output from the AST decoder maintains the dependency constraints and statement boundaries using terminal nodes, which facilitates learning the grammar of the target PL.
- In this stage, Coda employs an RNN-based error predictor (EP) to identify potential prediction mistakes in the output program from Phase 1. Ensembling method can be used to boost the performance of the error prediction. The EP is used to guide the iterative correction of the output program. Unlike traditional decompilers which utilize only the syntax information from the input, Coda leverages the Levenshtein edit distance (LD) between the compilation of the updated output program and the ground-truth low-level code to prevent false alarms induced by the EP. This can be considered conceptually similar to phase 2 of Katz et al. [41].

Coda's two phase approach is summarized in Figure. 10. Additionally, they also employ a technique called Attention Feeding to combine different contexts in the AST decoder. This can be considered similar to message passing in graphs, since trees are a form of graph. The Attention Feeding is comprised of parent attention feeding and input instruction attention feeding. Parent feeding refers to context from the parent of the current node, and input feeding refers to context from the current node's corresponding node in the input.

6.8. Bringing Graph Neural Networks into the picture

Even after incorporating AST structure, the base models of the previous approach are still derived from natural language based seq2seq models. This disregards the intrinsic structure of code (graphs/trees). To the best of our knowledge, [4] is the first Neural Decompilation approach which utilizes Graph Neural Networks (GNNs) to represent code in the context of decompilation. They use a backbone structural transformer [67] with inductive Graph Neural Networks [29] to represent the low-level code (LLC) as control/dataflow dependency graphs and source code as Abstract Syntax Tree (AST).

Their pipeline also involves a two step approach - data type solver (DT-Solver) and source code generator (SC-Gen). The output of the data type solver is used as the decoder input of the source code generation. Both use the same backbone with GNNs.

Structural Transformer with GNNs The backbone structural transformer has three components: (1) LLC encoder, (2) AST encoder, and (3) AST decoder. The LLC encoder takes the low-level instructions converted from binaries using disassembler as input. AST encoder takes the input of a previous (partial) AST, and the predictions of AST decoder are AST nodes, which can be equivalently converted to the high-level program. Here we will focus on the encoders and how they use GNNs. The GNN of choice in this paper is GraphSAGE (Hamilton et al. [29]). GNNs are used in two places - 1) To embed the initial code/AST structure 2) To augment the Attention Layers in the Transformer

Initial GNN embeddings The LLC encoder converts the initial low level code into a graph where each node is representated by its feature vector including features like register / instruction type (var_t / ins_t), position in the instruction (n_{pos}), node id (n_{id}) and numerical field (n_{num}). Edges are generated using hand crafted rules. Aggregation is done using max pooling. Aggregation vector and current state of the node are fed into a fully connected layer to get the next state. 3 layers are used for the GNN.

The AST encoder encodes the AST tree from the AST decoder to guide future tree expansions. They treat the AST

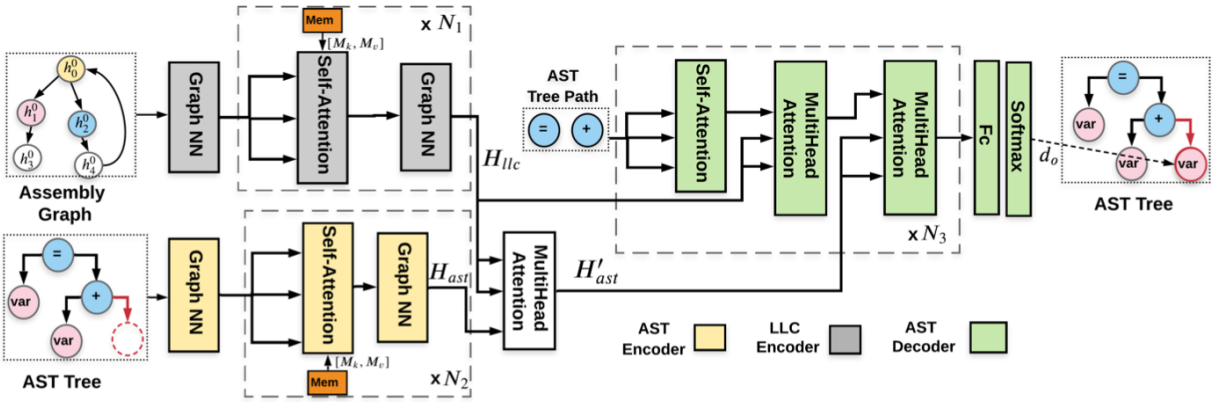


Figure 11. NBref's [4] backbone neural architecture design.

$(lib, b_{size}, b_{depth})$	(a) Data Type Recovery $Acc_{mac}(Acc_{mic})$					(b) AST Generation (token accuracy)			
	REWARD	baseline	lang2logic	Ins2AST+att	N-Bref	baseline	lang2logic	Ins2AST+att	N-Bref
(math,1,1)	85.1	96.6 (70.2)	92.1	94.1	**99.6** (71.1)	90.3	84.3	88.6	**94.5**
(math,2,2)	66.2	94.1 (71.4)	88.6	91.3	**97.3** (72.5)	87.3	81.5	85.6	**92.2**
(math,3,3)	53.1	91.9 (73.4)	86.4	88.3	**95.3** (75.1)	84.0	77.8	82.4	**89.5**
(str,1,1)	82.4	95.3 (71.6)	91.1	92.9	**98.3** (73.3)	88.6	80.8	85.1	**92.9**
(str,2,2)	63.1	93.1 (72.5)	88.3	90.8	**97.0** (74.9)	84.3	75.6	81.5	**90.6**
(str,3,3)	50.9	91.5 (73.6)	84.8	88.6	**95.3** (75.4)	79.1	70.3	76.9	**86.8**
leetcode	73.3	91.9 (73.8)	85.4	89.1	**96.0** (75.9)	82.3	73.1	78.6	**88.3**

Figure 12. NBref's [4] results compared to baselines and Coda (Ins2AST + attn).

Table 1. Comparison of different Neural Decompilation Techniques

Name	Citation	Year	Phases	Model Architecture	Model Layer	GNNs?
Katz et al. [40] RNN	[40]	2018	1	Encoder-Decoder	RNN	No
TraFix	[41]	2019	2	Encoder-Decoder	RNN	No
Coda	[26]	2019	2	Encoder-Decoder	Attention + LSTM	No
NBref	[4]	2021	2	Transformer	Self-Attention + GNN	Yes

tree as a graph (V) and embed it using GraphSAGE as well. The input of the GNN includes meta-features of the tokenized AST node and a boolean indicating whether the node is expanded in this step. The output hidden states from the GNN is fed into the self-attention module. At the end of the AST encoder, they integrate the AST encoder output and LLC encoder output using an additional multi-head attention layer.

GNN Augmentation for Attention The output of the attention layers is converted to a graph using the same edge rules in the LLC encoder and those states are then fed again into the GNN to obtain embeddings that incorporate the inductive biases of GNNs.

The neural architecture along with the GNN layers are visualized in Figure. 11. Experiments show that Nbref outperforms the corresponding transformer baseline and previous Neural Decompilers like Coda [26]. It also outperforms other models on human written Leetcode solutions. The performance numbers are visible in Figure 12. The performance gain over baseline Transformer clearly shows the value of using GNNs. Since most state of the art NMT models use transformers now, performance gains over previous Neural Decompilers using RNNs is expected.

Critics for Nbref have mentioned that they do not propose anything novel, but the use of GNN based representations in Neural Decompilation is definitely the first we have come across. GNNs to represent code is a natural choice in many domains related to programming languages because of the graph and tree based structure of code. We believe future research will definitely involve the use of GNNs for learning representations of code. The way this is done might

be tailored differently for decompilation to improve performance and efficiency. Once again, a common dataset and benchmark may help accelerate research in Neural Decompilation.

6.9. Comparison

Table 1 compares the relevant Neural Decompilation Approaches over the last few years. As we can see, only recently have GNNs been used in Neural decompilation. However, using GNNs has directly contributed to considerable improvements in performance as can be seen from Figure 12. We expect the use of GNNs to become mainstream in Neural Decompilation in the coming years.

7. CodeXGlue

Benchmarks in Natural Language Processing have played an important role in the progress of the field. Particularly, GLUE [69] is a comprehensive benchmark for training, evaluating and analyzing natural language understanding systems. It includes 9 tasks with corresponding datasets and evaluation guidelines, a diagnostic dataset to analyze NLP systems and a public leaderboard to track progress in the field. As expected, Deep Learning systems have dominated the benchmark and even surpassed human performance. As a result of this, SuperGLUE [70] has been released as a continuation of GLUE with harder challenges in order to further progress the field.

Benchmarks provide a unified and system agnostic way for several different researchers to compare their approaches on a common ground. Most NLP papers dealing with finetuning language models will mention GLUE or SuperGLUE, and publish their results to the public leaderboard [1]. Looking at the incredible success of such benchmarks in NLP warrants a similar benchmark in applying Machine Learning to the domain of Programming Languages and Code. That is precisely what the recently released CodeXGlue (Lu et al. [49]) aims to achieve. Similar to GLUE, CodeXGlue contains a bunch of tasks paired with their corresponding processed datasets and a public leaderboard. The leaderboard can be found at `https://microsoft.github.io/CodeXGLUE/` and the github repo is at `https://github.com/microsoft/CodeXGLUE`.

We have so far discussed many domains of Programming Languages across which Deep Learning and Graph Neural Networks have shown tremendous potential. Now, we intend to cover this benchmark as a unifying evaluation point that will guide further research in applying Deep Learning and GNNs to the domain of programming languages. We will cover the tasks of the benchmark in brevity and also include short descriptions of the baselines. We will end this section by looking at how Graph Neural Networks can be utilized to vastly improve scores across all tasks in the benchmark. We hope this will serve as viable research direction for interested people.

[1] https://gluebenchmark.com/leaderboard

7.1. Tasks

CodeXGlue includes 14 datasets for 10 diversified code intelligence tasks covering the following scenarios:

- code-code (clone detection, defect detection, cloze test, code completion, code repair, and code-to-code translation)
- text-code (natural language code search, text-to-code generation)
- code-text (code summarization)
- text-text (documentation translation)

The tasks can be summarized in Figure 13 which is taken from their github repo. The last category of tasks *text-text* leans more towards NLP but the rest contain code as their prime or major component. With code comes the intrinsic structure of programming languages including trees and graphs, and this varies greatly from natural languages. Therefore, we believe the inductive biases of Graph Neural Networks are very well suited to represent the code aspect in these tasks. Since this is a fairly recent benchmark, the leaderboard mostly contains baselines as of now. Most of the baselines mentioned in the CodeXGlue paper and papers like CodeBERT [25] use transformer models which are directly adapted from their NLP counterparts. Hence, there is plenty of room for innovation and climbing the leaderboard on this benchmark. We will briefly cover the baselines and then see how Graph Neural Networks can be used to bolster these deep learning models.

7.2. Baselines

The baselines use different variants of Transformers as their backbone. The tasks can be approximately classified into understanding tasks like code search, code clone detection, defect detection, cloze test (similar to masked language modeling) and generative tasks like code repair, code translation, code completion, code generation, etc.

For code understanding tasks, the authors provide CodeBERT [25] as a baseline. CodeBERT is very similar to BERT [22] and treats programming language as sequential. The model is pretrained on the CodeSearchNet dataset [34] using a masked language modeling objective like BERT and a combination of comments and code as input. It is then fine tuned on the downstream CodeXGlue tasks.

Category	Task	Dataset Name	Language	Train/Dev/Test Size	Baselines	Task definition
Code-Code	Clone Detection	BigCloneBench	Java	900K/416K/416K	CodeBERT	Predict semantic equivalence for a pair of codes.
		POJ-104	C/C++	32K/8K/12K		Retrieve semantically similar codes.
	Defect Detection	Devign	C	21k/2.7k/2.7k		Identify whether a function is vulnerable.
	Cloze Test	CT-all	Python, Java, PHP, JavaScript, Ruby, Go	-/-/176k		Tokens to be predicted come from the entire vocab.
		CT-max/min	Python, Java, PHP, JavaScript, Ruby, Go	-/-/2.6k		Tokens to be predicted come from {max, min}.
	Code Completion	PY150	Python	100k/5k/50k	CodeGPT	Predict following tokens given contexts of codes.
		GitHub Java Corpus	Java	13k/7k/8k		
	Code Repair	Bugs2Fix	Java	98K/12K/12K	Encoder-Decoder	Automatically refine codes by fixing bugs.
	Code Translation	CodeTrans	Java-C#	10K/0.5K/1K		Translate the codes from one programming language to another programming language.
Text-Code	NL Code Search	CodeSearchNet, AdvTest	Python	251K/9.6K/19K	CodeBERT	Given a natural language query as input, find semantically similar codes.
		CodeSearchNet, WebQueryTest	Python	251K/9.6K/1k		Given a pair of natural language and code, predict whether they are relevant or not.
	Text-to-Code Generation	CONCODE	Java	100K/2K/2K	CodeGPT	Given a natural language docstring/comment as input, generate a code.
Code-Text	Code Summarization	CodeSearchNet	Python, Java, PHP, JavaScript, Ruby, Go	908K/45K/53K	Encoder-Decoder	Given a code, generate its natural language docstring/comment.
Text-Text	Documentation Translation	Microsoft Docs	English-Latvian/Danish/Norwegian/Chinese	156K/4K/4K		Translate code documentation between human languages (e.g. En-Zh), intended to test low-resource multi-lingual translation.

Figure 13. Lu et al. [49]. CodeXGlue's tasks

For generative tasks like code generation and code completion, the authors provide CodeGPT as the baseline. CodeGPT is inspired from GPT2 [55]. CodeGPT contains pre-trained monolingual models on Python and Java corpora from the CodeSearchNet dataset, which includes 1.1M Python functions and 1.6M Java methods. They provide two variants of this baseline, one trained from scratch and one using GPT2 as the starting point.

For tasks which are more seq2seq in nature like code translation, the authors provide a encoder-decoder baseline with CodeBERT as the encoder and a custom transformer as the decoder.

All the baselines are summarized in Figure 14.

7.3. Graph Neural Networks

The baselines include models adapted from the NLP domain. As a result, the baselines treat code as natural language and disregards its intrinsic structure and syntactic biases. As we have seen throughout the paper, Graph Neural Networks have shown great promise in tasks related to programming languages. In section 1, we saw the latest research in representing code using graph neural networks. In the rest of the sections, we saw how GNNs and representations using GNNs are adapted to specific domains.

A lot of those techniques can be used across multiple tasks in CodeXGlue and we believe they can give consistent improvements as compared to the baselines. We will explore some ways that we envision GNNs being used across tasks in CodeXGlue.

- Given the initial code, we can obtain the Abstract Syntax Tree and use the AST to generate graph representations. The AST can be passed to a GNN like GraphSage [29] as done in Nbref [4]. The AST can be converted into a graph as shown in Section 1 as well. The Node Representations can then be used as inputs to the baselines shown in Figure 14.

- The transformer baselines can be adapted to have a GNN layer on top of the self-attention layer. This can capture the graphical nature of code and generate better representations.

- Graph representations of the entire code can be compared using vector similarity to aid in code similarity based tasks. Either the graph representations can be combined with the transformer representations or they can be used independently.

- Novel ways can be researched to combine Node embeddings from GNN with text embeddings for tasks that involve a combination of text and code.

These are interesting avenues of research and we hope to explore some of them in the future. These ideas are currently based on our understanding and hence all of them may

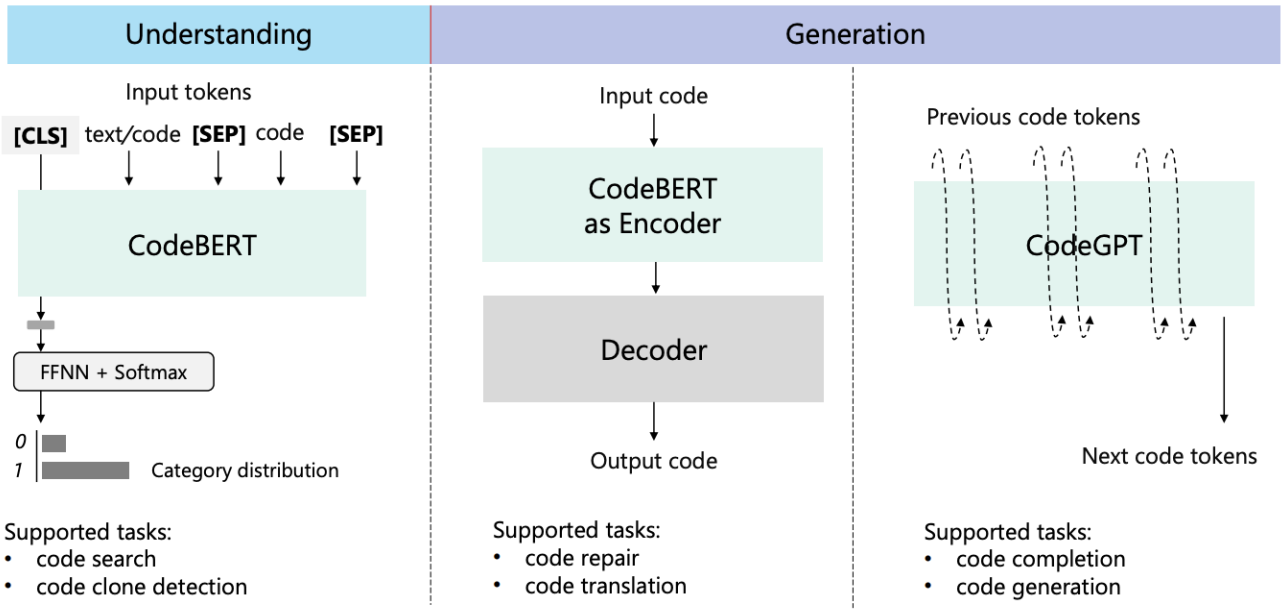


Figure 14. Lu et al. [49]. CodeXGlue's baseline architectures

not work, but it is still interesting to explore and generate common GNN representations of code.

8. Future Work

Great strides have been made in using GNN in Programming languages. It is a continuous process which leads to new state of the art models. This provides new avenues to refine our paper and present more up to date models. We will also start focusing on further evaluation of the presented models. We will intermix data sets or use different previously unused data sets to rigorously test the models. We can also shift out focus on more sub domains in GNN related to programming languages. This will allow us to present a more complete picture of the use of GNN in programming languages. Another potential future work might be the comparison of the GNN models with the other state of the art non GNN methods for solving problems in their respective sub domains.

9. Conclusion

Research in Neural methods and Graph Neural Networks for Programming Languages has evolved tremendously over the past few years. We have seen the boosts in performance using GNNs in a variety of different subdomains in this paper. However, the use of GNNs in Code is still at its infancy and there is still a lot of potential and future work to be done. Cross domain adpataion and generalized representation learning of code is also gaining a lot of interest and bound to see huge progress in the coming years. We hope this survey provides the readers with a quick debriefing of the use of GNNs in different domains of Programming Languages, and guides them towards developing better techniques.

References

[1] Tutorial: Sequence-to-sequence models. https://www.tensorflow.org/tutorials/seq2seq.

[2] MIPS. https://github.com/MIPT-ILab/mipt-mips.

[3] RedASM, 2019. https://github.com/REDasmOrg/REDasm.

[4] N-bref: A high-fidelity decompiler exploiting programming structures, 2021. https://openreview.net/pdf?id=6GkL6qM3LV.

[5] 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/.

[6] Allamanis, M., Barr, E. T., Devanbu, P., and Sutton, C. A survey of machine learning for big code and naturalness, 2018.

[7] 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.

[8] Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate, 2016.

[9] Bai, Y., Ding, H., Bian, S., Chen, T., Sun, Y., and Wang, W. Simgnn: A neural network approach to fast graph similarity computation. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, pp. 384–392, 2019.

[10] Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. Deepcoder: Learning to write programs, 2017.

[11] Bao, T., Burket, J., Woo, M., Turner, R., and Brumley, D. {BYTEWEIGHT}: Learning to recognize functions in binary code. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pp. 845–860, 2014.

[12] Ben-Nun, T., Jakobovits, A. S., and Hoefler, T. Neural code comprehension: A learnable representation of code semantics. *arXiv preprint arXiv:1806.07336*, 2018.

[13] Bošnjak, M., Rocktäschel, T., Naradowsky, J., and Riedel, S. Programming with a differentiable forth interpreter, 2017.

[14] Brumley, D., Jager, I., Avgerinos, T., and Schwartz, E. J. Bap: A binary analysis platform. In *International Conference on Computer Aided Verification*, pp. 463–469. Springer, 2011.

[15] Brumley, D., Lee, J., Schwartz, E. J., and Woo, M. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pp. 353–368, 2013.

[16] Bunel, R., Hausknecht, M., Devlin, J., Singh, R., and Kohli, P. Leveraging grammar and reinforcement learning for neural program synthesis, 2018.

[17] Chen, X., Liu, C., and Song, D. Towards synthesizing complex programs from input-output examples, 2018.

[18] Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[19] Cifuentes, C. *Reverse compilation techniques*. Citeseer, 1994.

[20] Dai, H., Khalil, E. B., Zhang, Y., Dilkina, B., and Song, L. Learning combinatorial optimization algorithms over graphs, 2018.

[21] 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.

[22] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[23] Emmerik, M. and Waddington, T. Using a decompiler for real-world source recovery. In *11th Working Conference on Reverse Engineering*, pp. 27–36. IEEE, 2004.

[24] 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.

[25] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[26] Fu, C., Chen, H., Liu, H., Chen, X., Tian, Y., Koushanfar, F., and Zhao, J. Coda: An end-to-end neural program decompiler. *Advances in Neural Information Processing Systems*, 32:3708–3719, 2019.

[27] Guide, P. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, 2(11), 2011.

[28] 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.

[29] Hamilton, W. L., Ying, R., and Leskovec, J. Inductive representation learning on large graphs. *arXiv preprint arXiv:1706.02216*, 2017.

[30] Hennessy, J., Jouppi, N., Przybylski, S., Rowen, C., Gross, T., Baskett, F., and Gill, J. Mips: A microprocessor architecture. *ACM SIGMICRO Newsletter*, 13 (4):17–22, 1982.

[31] Hex-Rays. Hex-Ray. https://www.hex-rays.com/products/decompiler/.

[32] Hindle, A., Barr, E. T., Su, Z., Devanbu, P. T., , and Gabel, M. On the naturalness of software. *Communications of the ACM: Invited Research Hilights (CACM)*, pp. 122–131, 2016. URL http://softwareprocess.ca/pubs/hindle2016CACM.pdf.

[33] Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[34] Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[35] Hyyrö, H. Explaining and extending the bit-parallel approximate string matching algorithm of myers. Technical report, Citeseer, 2001.

[36] Jean, S., Cho, K., Memisevic, R., and Bengio, Y. On using very large target vocabulary for neural machine translation. *arXiv preprint arXiv:1412.2007*, 2014.

[37] 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.

[38] Kalyan, A., Mohta, A., Polozov, O., Batra, D., Jain, P., and Gulwani, S. Neural-guided deductive search for real-time program synthesis from examples, 2018.

[39] Kalyan, A., Mohta, A., Polozov, O., Batra, D., Jain, P., and Gulwani, S. Neural-guided deductive search for real-time program synthesis from examples, 2018.

[40] Katz, D. S., Ruchti, J., and Schulte, E. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 346–356. IEEE, 2018.

[41] Katz, O., Olshaker, Y., Goldberg, Y., and Yahav, E. Towards neural decompilation. *arXiv preprint arXiv:1905.08325*, 2019.

[42] Kolbitsch, C., Comparetti, P. M., Kruegel, C., Kirda, E., Zhou, X.-y., and Wang, X. Effective and efficient malware detection at the end host. In *USENIX security symposium*, volume 4, pp. 351–366, 2009.

[43] 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.

[44] Křoustek, J., Matula, P., and Zemek, P. Retdec: An open-source machine-code decompiler, 2017.

[45] 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.

[46] Lee, J., Avgerinos, T., and Brumley, D. Tie: Principled reverse engineering of types in binary programs. 2011.

[47] Lin, Z., Zhang, X., and Xu, D. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*, pp. 1–1, 2010.

[48] Lu, M., Tan, D., Xiong, N., Chen, Z., and Li, H. Program classification using gated graph attention neural network for online programming service. *arXiv preprint arXiv:1903.03804*, 2019.

[49] Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[50] 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.

[51] 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.

[52] Murali, V., Qi, L., Chaudhuri, S., and Jermaine, C. Neural sketch learning for conditional program generation, 2018.

[53] Nguyen, A. T., Nguyen, T. T., and Nguyen, T. N. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 651–654, 2013.

[54] 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/.

[55] Radford, A., Wu, J., Amodei, D., Amodei, D., Clark, J., Brundage, M., and Sutskever, I. Better language models and their implications. *OpenAI Blog https://openai. com/blog/better-language-models*, 2019.

[56] 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.

[57] Rosenblum, N. E., Zhu, X., Miller, B. P., and Hunt, K. Learning to analyze binary computer code. In *AAAI*, pp. 798–804, 2008.

[58] Sanfeliu, A. and Fu, K.-S. A distance measure between attributed relational graphs for pattern recognition. *IEEE transactions on systems, man, and cybernetics*, (3):353–362, 1983.

[59] 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.

[60] Shannon, C. E. A mathematical theory of communication. *ACM SIGMOBILE mobile computing and communications review*, 5(1):3–55, 2001.

[61] 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.

[62] Siegelmann, H. T. *Foundations of recurrent neural networks*. PhD thesis, Citeseer, 1993.

[63] 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.

[64] 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.

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

[66] Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215*, 2014.

[67] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

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

[69] Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

[70] Wang, A., Pruksachatkun, Y., Nangia, N., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. Superglue: A stickier benchmark for general-purpose language understanding systems. *arXiv preprint arXiv:1905.00537*, 2019.

[71] 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.

[72] Yakdan, K., Eschweiler, S., Gerhards-Padilla, E., and Smith, M. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS*. Citeseer, 2015.

[73] Yakdan, K., Dechand, S., Gerhards-Padilla, E., and Smith, M. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 158–177. IEEE, 2016.

[74] Yang, X., Chen, Y., Eide, E., and Regehr, J. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pp. 283–294, 2011.