

Université de Montréal

**Learned Interpreters: Structural and Learned  
Systematicity in Neural Networks for Program  
Execution**

par

**David Bieber**

Département d'informatique et de recherche opérationnelle  
Faculté des arts et des sciences

Thèse présentée en vue de l'obtention du grade de  
Philosophiæ Doctor (Ph.D.)  
en Discipline

July 3, 2023



# Université de Montréal

Faculté des arts et des sciences

---

Cette thèse intitulée

## **Learned Interpreters: Structural and Learned Systematicity in Neural Networks for Program Execution**

présentée par

**David Bieber**

a été évaluée par un jury composé des personnes suivantes :

*Pierre-Luc Bacon*

---

(président-rapporteur)

*Hugo Larochelle*

---

(directeur de recherche)

*Daniel Tarlow*

---

(codirecteur)

*Jian Tang*

---

(membre du jury)

*Swarat Chaudhuri*

---

(examineur externe)

*François Perron*

---

(représentant du doyen de la FESP)



# Résumé

---

Les architectures de réseaux de neurones profonds à usage général ont fait des progrès surprenants dans l'apprentissage automatique pour le code, permettant l'amélioration de la complétion de code, la programmation du langage naturel, la détection et la réparation des bogues, et même la résolution de problèmes de programmation compétitifs à un niveau de performance humain. Néanmoins, ces méthodes ont du mal à comprendre le processus d'exécution du code, même lorsqu'il s'agit de code qu'ils écrivent eux-mêmes. À cette fin, nous explorons une architecture du réseau neuronal inspiré d'interpréteur de code, via une nouvelle famille d'architecture appelée Instruction Pointer Attention Graph Neural Networks (IPA-GNN). Nous appliquons cette famille d'approches à plusieurs tâches nécessitant un raisonnement sur le comportement d'exécution du programme : apprendre à exécuter des programmes complets et partiels, prédire la couverture du code pour la vérification du matériel, et prédire les erreurs d'exécution dans des programmes de compétition. Grâce à cette série de travaux, nous apportons plusieurs contributions et rencontrons de multiples résultats surprenants et prometteurs. Nous introduisons une bibliothèque Python pour construire des représentations de graphes des programmes utiles dans la recherche sur l'apprentissage automatique, qui sert de fondement à la recherche dans cette thèse et dans la communauté de recherche plus large. Nous introduisons également de riches ensembles de données à grande échelle de programmes annotés avec le comportement du programme (les sorties et les erreurs soulevées lors de son exécution) pour faciliter la recherche dans ce domaine. Nous constatons que les méthodes IPA-GNN présentent une forte généralisation améliorée par rapport aux méthodes à usage général, fonctionnant bien lorsqu'ils sont entraînés pour exécuter uniquement des programmes courts mais testés sur des programmes plus longs. En fait, nous constatons que les méthodes IPA-GNN surpassent les méthodes génériques sur chacune des tâches de modélisation du comportement que nous considérons dans les domaines matériel et logiciel. Nous constatons même que les méthodes inspirées par l'interpréteur de code qui modélisent explicitement la gestion des exceptions ont une propriété interprétative souhaitable, permettant la prédiction des emplacements d'erreur même lorsqu'elles n'ont été entraînées qu'à prédire la présence d'erreur et le type d'erreur. Au total, les architectures inspirées des interpréteurs de code comme l'IPA-GNN représentent un chemin prometteur à

suivre pour imprégner des réseaux de neurones avec de nouvelles capacités pour apprendre à raisonner sur les exécutions de programme.

Mots-clés : apprentissage automatique, réseaux de neurones profonds, analyse de programmes, génie logiciel

# Abstract

---

General purpose deep neural network architectures have made startling advances in machine learning for code, advancing code completion, enabling natural language programming, detecting and repairing bugs, and even solving competitive programming problems at a human level of performance. Nevertheless, these methods struggle to understand the execution behavior of code, even when it is code they write themselves. To this end, we explore interpreter-inspired neural network architectures, introducing a novel architecture family called instruction pointer attention graph neural networks (IPA-GNN). We apply this family of approaches to several tasks that require reasoning about the execution behavior of programs: learning to execute full and partial programs, code coverage prediction for hardware verification, and predicting runtime errors in competition programs. Through this series of works we make several contributions and encounter multiple surprising and promising results. We introduce a Python library for constructing graph representations of programs for use in machine learning research, which serves as a bedrock for the research in this thesis and in the broader research community. We also introduce rich large scale datasets of programs annotated with program behavior like outputs and errors raised to facilitate research in this domain. We find that IPA-GNN methods exhibit improved strong generalization over general purpose methods, performing well when trained to execute only on short programs and tested on significantly longer programs. In fact, we find that IPA-GNN methods outperform generic methods on each of the behavior modeling tasks we consider across both hardware and software domains. We even find that interpreter-inspired methods that model exception handling explicitly have a desirable interpretability property, enabling the prediction of error locations even when only trained on error presence and kind. In total, interpreter-inspired architectures like the IPA-GNN represent a promising path forward for imbuing neural networks with novel capabilities for learning to reason about program executions.

Keywords: machine learning, deep neural networks, program analysis, software engineering





# Contents

---

<b>Résumé</b> .....	5
<b>Abstract</b> .....	7
<b>List of tables</b> .....	15
<b>List of figures</b> .....	19
<b>List of acronyms and abbreviations</b> .....	23
<b>Remerciements</b> .....	27
<b>Chapter 1. Introduction</b> .....	29
1.1. Neural networks for software engineering .....	29
1.2. Execution behavior of programs .....	30
1.3. Systematicity .....	31
1.4. Thesis Contributions .....	32
1.5. Additional Contributions .....	33
1.6. Thesis Outline .....	34
<b>Chapter 2. Background</b> .....	37
2.1. Setup: Program analysis tasks .....	37
2.1.1. The diversity of program analysis tasks .....	37
2.1.2. Machine learning terminology .....	38
2.1.3. Systematic generalization .....	38
2.1.4. The challenge of reasoning about program execution .....	39
2.2. Approach: Neural Network Methods .....	40
2.2.1. Sequence Representations of Source Code .....	40
2.2.2. Graph Representations of Source Code .....	41
2.2.3. Sequence Models .....	42

2.2.4.	Graph neural networks .....	44
2.2.5.	Interpreter-inspired machine learning models .....	45
<b>Article 1. A Library for Representing Python Programs as Graphs for Machine Learning .....</b>		<b>47</b>
1.	Introduction .....	49
2.	Background .....	49
3.	Capabilities, Possible Extensions, and Limitations .....	50
3.1.	Capabilities .....	50
3.1.1.	Control-Flow Graphs .....	51
3.1.2.	Data-Flow Analyses .....	52
3.1.3.	Composite Program Graphs .....	52
3.1.4.	Cyclomatic Complexity .....	53
3.2.	Possible Extensions .....	54
3.2.1.	Alternative Composite Program Graphs .....	54
3.2.2.	Inter-procedural Control-Flow Graphs .....	54
3.2.3.	Data-Flow Graphs .....	56
3.2.4.	Span-Mapped Graphs .....	56
3.2.5.	Additional Data-Flow Analyses .....	56
3.3.	Limitations .....	57
4.	Use cases .....	57
4.1.	Graph Representations as Model Inputs .....	57
4.2.	Program Graphs as Targets .....	58
5.	Case Study: Project CodeNet .....	58
6.	Discussion .....	61
<b>Article 2. Learning to Execute Programs with Instruction Pointer Attention Graph Neural Networks .....</b>		<b>63</b>
1.	Introduction .....	65
2.	Background and Related Work .....	66
3.	Task .....	67
3.1.	Learning to Execute as Static Analysis .....	67
3.2.	Formal Specification and Evaluation Metrics .....	69

4.	Approach	69
4.1.	Instruction Pointer RNN Models	69
4.2.	Instruction Pointer Attention Graph Neural Network	71
4.3.	Relationship with IP-RNNs	72
4.4.	Relationship with GNNs	73
5.	Experiments	74
5.1.	Dataset	74
5.2.	Evaluation Criteria	74
5.3.	Training	75
5.4.	Program Execution Results	75
6.	Conclusion and Future Work	76
7.	Broader Impact	77
<b>Article 3. Learning Semantic Representations to Verify Hardware Designs</b>		<b>79</b>
1.	Introduction	81
2.	Design2Vec: Representation learning of design abstractions	83
2.1.	Architecture	84
2.2.	RTL IPA-GNN architecture	86
2.3.	Gradient-based search for test generation	87
3.	Impact of Design2Vec solutions in practical verification	88
4.	Coverage prediction experiments	88
5.	Test generation using Design2Vec	91
6.	Related work	93
7.	Conclusion	95
<b>Article 4. Static Prediction of Runtime Errors by Learning to Execute Programs with External Resource Descriptions</b>		<b>97</b>
1.	Introduction	99
2.	Related Work	100
3.	Runtime Error Prediction	101
4.	Approach: IPA-GNNs as Relaxations of Interpreters	102

4.1. Extending the IPA-GNN to Real Programs .....	102
4.2. Executing with Resource Descriptions .....	104
4.3. Modeling Exception Handling .....	105
4.4. Unsupervised Localization of Errors .....	106
5. Experiments .....	106
5.1. Evaluation of IPA-GNN Against Baselines .....	106
5.2. Incorporating Resource Descriptions .....	108
5.3. Interpretability and Localization .....	109
6. Discussion .....	111
<b>Conclusion</b> .....	<b>113</b>
1. Summary .....	113
2. Placing in Modern Context .....	114
3. Future Directions .....	116
<b>References</b> .....	<b>119</b>
<b>Appendixes</b> .....	<b>134</b>
A. Appendixes for Article 1 .....	134
A.1. Program Graph Visualizations .....	134
A.2. Histograms of Program Graph Metrics .....	142
B. Appendixes for Article 2 .....	143
B.1. Architecture Details .....	143
B.2. Data Generation .....	143
C. Appendixes for Article 3 .....	145
C.1. RTL CDFGs .....	145
C.2. Industrial verification flow .....	145
C.3. Sizes of designs .....	146
C.4. Examples of generated tests .....	148
C.5. Experimental hyperparameters .....	148
C.6. Architectural ablation studies .....	149
C.7. Comparison of Design2Vec and black-box optimizer tests .....	153
D. Appendixes for Article 4 .....	154

D.1.	Python Runtime Error Dataset Details .....	154
D.2.	Under-approximation of Error Labels .....	156
D.3.	IPA-GNN Architecture .....	157
D.4.	Input Modulation .....	157
D.5.	Training Details .....	158
D.6.	Metric Variances .....	159
D.7.	Static Analysis Baseline .....	160
D.8.	Localization by Modeling Exception Handling .....	161
D.9.	Localization by Multiple Instance Learning .....	162
D.10.	Broader Impact .....	163
D.11.	Example Visualizations .....	164



## List of tables

---

1	Descriptions of the edge types supported by <code>python_graphs</code> . A directed edge from node <i>src</i> to node <i>dest</i> has the semantics given in this table. ....	54
2	Example programs and their associated control-flow graphs and program graphs. Enlarged versions of the program graph figures are included in Appendix A.1. ..	55
3	Control-flow graph construction success rates on a dataset of both valid and invalid Python submissions to competitive programming problems. ....	59
4	Frequencies of edge types in the composite program graphs for the Project CodeNet Python submissions. # / Program is the average number of occurrences of the edge type across all programs. Freq. (%) shows the percent of programs exhibiting the edge type at least once. ....	59
5	Summary statistics for various graph metrics across the dataset. The diameter and maximum betweenness centrality metrics do not include graphs exceeding 5000 nodes. ....	60
6	The IPA-GNN model is a message passing GNN. Selectively replacing its components with those of the GGNN yields two baseline models, NoControl and NoExecute. <b>Blue expressions</b> originate with the IPA-GNN, and <b>orange expressions</b> with the GGNN. ....	73
7	Accuracies on $D_{\text{test}}$ (%) .....	75
8	Accuracy at coverage prediction for Design2Vec (with the RTL IPA-GNN layer) compared to a black-box multi-layer perceptron (MLP), which does not have information about the design. ....	90
9	Coverage prediction accuracies of different GNN architectures within Design2Vec.	91
10	Comparison of Design2Vec and black-box optimizer tests for covering hard to cover points: IBEX v1. Number of tests are RTL simulations. ....	93
11	Comparison of Design2Vec and black-box optimization to hit target cover points: TPU. Number of tests are RTL simulations. ....	93

12	Comparing test generation of easy, medium and hard to cover points. Design2Vec is very efficient at generating tests for hard to cover points. Summarized result of table in Appendix C.7 Table 23. ....	94
13	Distribution of target classes in the runtime errors dataset. † denotes the balanced test split. ....	101
14	Error classification and error localization results on the balanced test set with and without resource descriptions (R.D.).....	108
15	A comparison of early and late fusion methods for incorporating external resource description information into interpreter-inspired models. ....	108
16	Per-node localization predictions from the BASELINE and DOCSTRING Exception IPA-GNN models on a sample program from the validation split. The target class is EOFERROR, occurring on line 2 ( $n = 2$ ). BASELINE predicts NO ERROR with confidence 0.708, while R.D. predicts EOFERROR with confidence 0.988, localized at line 3 ( $n = 3$ ). The input description shows the cause for error: there are more <code>input()</code> calls than the number of expected inputs. ....	110
17	Comparison of relative sizes of RTL CDFGs between IBEX and TPU design.....	148
18	Example tests generated by Design2Vec for hard to cover points. Multiple cover points that are local neighbors of the target point are provided as input to Design2Vec, which helps the GNN-based architecture. ....	148
19	Comparison of training and validation accuracy across variants of the Design2Vec architecture on coverage prediction on the TPU design. ....	149
22	Comparing the training and validation accuracy of the Design2Vec model using a k-hop edge augmented graph ( $k \in \{2, 4, 16\}$ ) across a variety of experimental setups.....	149
20	Comparing the train and validation accuracy across different split selection methods: whether to hide test parameters, and whether to sample the training set via every-k sampling or uniformly random cover points.....	152
21	Comparing the train and validation accuracy on TPU while varying the numbers of GCN layers. We report the results across three seeds for each network depth..	152
23	Comparison of Design2Vec and black-box optimizer tests for covering overall cover points in different cover point probability buckets.....	153
24	Hyperparameter settings for each of the three Transformer sizes. ....	158



25	Hyperparameters considered for random search during model selection.....	159
26	The parameter count, training latency (sec/step), and inference latency (sec/batch) for the best performing instance of each model variant. Training and inference latencies use batch size 32.....	160
27	Mean and standard deviation for each metric is calculated from three training runs per model, using the hyperparameters selected via model selection.....	160
28	The pylint baseline for runtime error prediction predicts the error class shown when it encounters any of the corresponding pylint findings. Many of pylint's 235 finding types do not indicate runtime errors. This table shows the mapping used by the pylint baseline. ....	162



## List of figures

---

1	The input formats accepted by the <code>python_graphs</code> library are (a) function, (b) source code, and (c) AST. The code snippets here demonstrate construction of each input format for the example function <code>fn1</code> .....	51
2	<code>python_graphs</code> supports construction and analysis of control-flow graphs for arbitrary Python functions.....	52
3	Example usage of data-flow analysis and program graph construction in <code>python_graphs</code> .....	53
4	The relationship between program length and cyclomatic complexity for Python submissions in Project CodeNet.....	59
5	Box plots for various metrics of program graphs for Python submissions in Project CodeNet. The vertical blue line in each boxplot shows the median of the data as usual, while the blue $\times$ shows the mean.....	59
6	<b>Program representation.</b> Each line of a program is represented by a 4-tuple tokenization containing that line's (indentation level, operation, variable, operand), and is associated with a node in the program's statement-level control flow graph.	67
7	<b>Model paths comparison.</b> The edges in these graphs represent a possible set of paths along which information may propagate in each model. The pills indicate the positions where a model makes a learned branch decision. The Hard IP-RNN's branch decision is discrete (and in this example, incorrect), whereas in the IPA-GNN the branch decision is continuous. ....	70
8	<b>A single IPA-GNN layer.</b> At each step of execution of the IPA-GNN, an RNN over the embedded source at each line of code produces state proposals $a_{t,n}^{(1)}$ . Distinct state values are shown in distinct colors. A two output unit dense layer produces branch decisions $b_{t,n,:}$ , shown as pill lightness and edge width. These are used to aggregate the soft instruction pointer and state proposals to produce the new soft instruction pointer and hidden states $p_{t,n}$ and $h_{t,n}$ . ....	73
9	Accuracy of models as a function of program length on the program execution tasks. Spread shows one standard error of accuracy.....	75

10	<b>Instruction Pointer Attention.</b> Intensity plots show the soft instruction pointer $p_{t,n}$ at each step of the IPA-GNN on two programs, each with two distinct initial values for $v_0$ . . . . .	76
11	An example snippet of a Verilog RTL source code module and the corresponding CDFG.	84
12	The Design2Vec architecture that takes as input a cover point and an input test vector, and predicts the corresponding coverage. CDFG: Control and data flow graph. TP: test parameter. CP: cover point. . . . .	85
13	Test Input Generation . . . . .	87
14	The overall workflow to use Design2Vec in generative loop to generate input test parameters. . . . .	87
15	A sample program and its execution under discrete interpreters $I$ and $I'$ (Algorithm 1) and under continuous relaxation $\tilde{I}'$ of interpreter $I'$ . $p_{I_t}$ denotes the instruction pointer under $I$ at step $t$ . . . . .	103
16	Heatmap of instruction pointer values produced by BASELINE and DOCSTRING Exception IPA-GNNs for the example in Table 16. The x-axis represents timesteps and the y-axis represents nodes, with the last two rows respectively representing $n_{\text{exit}}$ and $n_{\text{error}}$ . The BASELINE instruction pointer value is diffuse, with most probability mass ending at $n_{\text{exit}}$ . The R.D. instruction pointer value is sharp, with almost all probability mass jumping to $n_{\text{error}}$ from node 2. . . . .	109
17	Program graph for Program #1 from Table 2. . . . .	134
18	Program graph for Program #2 from Table 2. . . . .	135
19	Program graph for Program #3 from Table 2. . . . .	136
20	Program graph for Program #4 from Table 2. . . . .	137
21	Program graph for Program #5 from Table 2. . . . .	138
22	Program graph for Program #6 from Table 2. . . . .	139
23	Program graph for Program #7 from Table 2. . . . .	140
24	Program graph for Program #8 from Table 2. . . . .	141
25	Histograms for various metrics of program graphs from the Project CodeNet dataset. Red bars include graphs where the metric lies outside the range covered by the other bars. . . . .	142
26	Grammar describing the generated programs comprising the dataset in this paper.	144

27	Intensity plots show the soft instruction pointer $p_{t,n}$ at each step of the IPA-GNN during full program execution for four randomly sampled programs. . . . .	144
28	The same programs as in Figure 27, with a single statement masked in each. The intensity plots show the soft instruction pointer $p_{t,n}$ at each step of the IPA-GNN during partial program execution. . . . .	145
29	RTL and CDFG execution (simulation) over three cycles $t-1, t, t+1$ . . . . .	146
30	Input stimulus and corresponding branches that are covered. Coverage is a path tracing through the CDFG. . . . .	146
31	Industrial verification flow with manually generated testbench, tests, constraints and coverage feedback. This flow takes multiple person-years of engineer productivity to converge . . . . .	147
32	Value proposition of Design2Vec when integrated into the loop of an industrial verification flow. It learns about the design state space and generates tests to cover different uncovered cover points (holes). It can potentially be used to generate constraints. . . . .	147
33	A histogram showing the distribution of program lengths, measured in lines, represented in the runtime errors dataset train split. . . . .	156
34	The distribution of statement lengths, measured in tokens, in the runtime errors dataset train split. . . . .	156
35	The target error kind is INDEXERROR, occurring on line 5 ( $n = 5$ ). BASELINE incorrectly predicts NO ERROR with confidence 0.808. DOCSTRING correctly predicts INDEXERROR with confidence 0.693, but localizes to line 3 ( $n = 2$ ). Both BASELINE and DOCSTRING instruction pointer values start out sharp and become diffuse when reaching the for-loop. The BASELINE instruction pointer value ends with most probability mass at $n_{\text{exit}}$ . The DOCSTRING instruction pointer value has a small amount of probability mass reaching $n_{\text{exit}}$ , with most probability mass ending at $n_{\text{error}}$ . . . . .	164
36	The target error kind is VALUEERROR, occurring on line 1 ( $n = 0$ ). BASELINE incorrectly predicts INDEXERROR with confidence 0.319 on line 1 ( $n = 0$ ). DOCSTRING correctly predicts VALUEERROR with confidence 0.880 on line 2 ( $n = 1$ ), corresponding to $\mathbf{A}[\mathbf{n}]$ . Both BASELINE and DOCSTRING instruction pointer values start out sharp and quickly shift most of the probability mass to the exception node. . . . .	165

37 The target error kind is NO ERROR. BASELINE correctly predicts NO ERROR with confidence 0.416. DOCSTRING also correctly predicts NO ERROR with confidence 0.823. The BASELINE instruction pointer value makes its largest probability mass contribution to  $n_{\text{error}}$  at  $n = 0$  and ends up with mass split between  $n_{\text{exit}}$  and  $n_{\text{error}}$ . The DOCSTRING instruction pointer value accumulates little probability in  $n_{\text{error}}$  and ends up with most probability mass in  $n_{\text{exit}}$ . ..... 166

## List of acronyms and abbreviations

---

ALU	Arithmetic Logic Unit
BPE	Byte-Pair Encoding
C.A.	Cross Attention
CDFG	Control Data Flow Graph
CFG	Context Free Grammar
CFG	Control Flow Graph
CGRU	Convolutional Gated Recurrent Unit
CP	Cover Point
E. IPA-GNN	Exception Instruction Pointer Attention Graph Neural Network
FiLM	Feature-wise Linear Modulation
GAT	Graph Attention Network

GCN	Graph Convolution Network
GCP	Google Cloud Platform
GGNN	Gated Graph Neural Network
GNN	Graph Neural Network
GNN-MLP	Graph Neural Network with Multilayer Perceptron
GPU	Graphics Processing Unit
GREAT	Global Relational Embedding Attention Transformer
HDL	Hardware Description Language
HYBRO	Hybrid Analysis and Branch Coverage Optimizations
ICFG	Interprocedural Control Flow Graph
IID	Independent and identically distributed
IP-RNN	Instruction Pointer Recurrent Neural Network
IPA	Instruction Pointer Attention



IPA-GNN	Instruction Pointer Attention Graph Neural Network
LSTM	Long Short Term Memory
MIL	Multiple Instance Learning
ML	Machine Learning
MLP	Multi-layer Perceptron
PCFG	Probabilistic Context Free Grammar
PRE	Python Runtime Errors
QKV	Queries, Keys, and Values
R-GAT	Relational Graph Attention Network
R.D.	Resource Description
RISC-V	Reduced Instruction Set Computer (RISC) Five
RNN	Recurrent Neural Network
RTL	Register-Transfer Level

RTL IPA-GNN	Register-Transfer Level Instruction Pointer Attention Graph Neural Network
SAT	Satisfiability
SMT	Satisfiability Modulo Theories
SRP	Smart Regression Planner
TBPTT	Truncated backpropagation through time
TPU	Tensor Processing Unit
UDF	User-Defined Function

## Remerciements

---

I owe thanks to a great number of people, who supported and encouraged me throughout my PhD work. To my Mom and Dad, thank you for teaching me the value of hard work and alacrity. You've always been there, encouraging me, in everything I've pursued. To my sister Nicole, thank for being an amazing role model and teaching me to program at a young age. You've been an inspiration for thirty years and counting now. Adriana, thank you for your constant support through the program. I'm glad to be able to contribute a small piece to the large pile of degrees we've accumulated between us.

I have many people to thank at Google for making my PhD, with all its logistical needle-threading, a reality. George Dahl, thank you for manifesting Google's support for the PhDs of myself and my colleagues. Jeff Dean, thank you for supporting the realization of this vision. Phing Lee, Dan Nanas, Leslie Phillips, and Natacha Mainville, thank you for the hard work you've done to help so many of us navigate the student employee life. Samy Bengio, Charles Sutton, Martin Abadi, and Chandu Thekkath, thank you for providing the support that allowed me to pursue this degree.

I was fortunate to have a wonderful pair of advisors helping me throughout my PhD, Danny Tarlow and Hugo Larochelle. The brainstorming sessions, research discussions, and problem solving we did together formed the backbone of the degree. Danny, thank you for not being afraid to get your hands dirty diving into code and error messages alongside me. Hugo, thanks for the persistent bureaucracy-busting that was required to balance this logistically interesting arrangement.

Kensen Shi, Pengcheng Yin, Vincent Hellendoorn, Petros Maniatis, Rishabh Singh, Shobha Vasudevan, Joe Jiang, and Sarath Chandar, you are never short on good ideas. Thank you for the thoughtful discussions that were essential to this work.

Daniel Zhang, Rishab Goel, Disha Shrivastava, Daniel Johnson, Jacob Austin, Breandan Considine, Justine Gehring, thank you for all the excellent discussions and kindnesses throughout. For the working sessions, the gossip, and the good food. Quoc Le, Mohammad Norouzi, Naavdeep Jaitly, thank you for welcoming me into the world of deep learning and starting me on this adventure. David Dohan, thank you for encouraging me to go. Ryan

Dahl, Sergio Guadarrama, Kevin Murphy, Jonathon Shlens, Rui Zhou, and Kevin Swersky, thank you for ensuring I had a good time when I got there.

I look forward to continuing on this adventure together.

# Chapter 1

---

## Introduction

### 1.1. Neural networks for software engineering

Over the last two decades, companies like Google and GitHub have accumulated massive codebases, surpassing a billion lines of code [49, 117]. As a common practice, software engineers rely on automated tools to analyze and manipulate programs at every step of the software development process [1]. Through machine learning, we can design methods that take advantage of these large codebases to assist developers further, enhancing their productivity.

Already, neural network techniques have achieved impressive results on a number of software engineering tasks, but they struggle on tasks that directly call for reasoning about program executions. Neural networks have enabled new capabilities in program analysis [94, 118], program repair [32, 138, 141], code completion [96, 102], programming by example [38], and even competitive programming [95]. These approaches achieve their success through pattern recognition, learning from millions of labeled examples of the task at hand. However, tasks that require modeling the execution behavior of programs have presented a persistent challenge for neural network methods; a variety of settings have highlighted that when a task directly requires reasoning about a program’s execution behavior, general purpose neural network approaches have so far fallen short.

This thesis looks specifically at neural methods for modeling the execution behavior of programs to advance neural approaches to program analysis. Modeling execution behavior is central to a wide range of tasks, from predicting program properties like execution time or runtime errors, to security assessments like vulnerability detection. We will consider the tasks of learning to execute full and partial programs, predicting test coverage for hardware verification, and identifying runtime errors in real programs without having to run them. Our main approach will be to model execution behavior in a neural network by drawing from program interpreters in our design of novel neural architectures. Through this approach, we’ll not only improve performance on the tasks of interest, we’ll also discover interpretability

properties that give our models novel capabilities and we'll design and release a library that facilitates a range of research into using graph representations of programs for machine learning.

## 1.2. Execution behavior of programs

When a program runs, a series of instructions are executed. These instructions have many possible functions: i.e. performing calculations, reading and writing values from memory, interfacing with peripheral devices, and controlling which instructions will execute next. By performing many of these instructions in quick succession, programs serve an enormous range of purposes; everything from Google Search to banking software works in this way, and the ubiquity and importance of software globally cannot be overstated. The end-to-end behavior of a program – the relationship between that program's inputs and its outputs – is closely linked to the execution behavior of that program. This relationship is governed by strict rules, the semantics of the programming language, which indicate the precise meaning of each possible instruction. General purpose neural networks, even when trained on millions of examples of the end-to-end behavior of programs, do not recover these precise rules. We say that tasks which require learning precise algorithmic rules to solve perfectly require *systematicity*.

Programming language semantics typically have properties that make reasoning about program executions distinct from other machine learning tasks. Particularly, programming language semantics are precise and uniform. While software exhibits naturalness like natural language, it also is governed by precise rules, far more well-defined and with fewer exceptions than the rules of human languages. The semantics of a programming language are uniform across the entire language; there is little implicit context modifying the semantics compared with human languages which occur in ever-changing complicated environments. This precision and uniformity in programming language semantics enables compositionality and generalization. What this means is that if one understands independently what two programming language instructions do, then the behavior of the composition of these instructions follows naturally. So if one learns how a language works well enough to understand all ten-line programs, we might expect that they can also understand hundred-line programs. General purpose neural methods don't usually exhibit this property, and we aim to take advantage of it in designing neural architectures for reasoning about programs.

Though programming language behavior is uniform in any particular environment, there are many possible execution environments and many languages, each with their own semantics. Our approach will be flexible enough to admit a variety of non-standard notions of program execution, such as executing both Python and Verilog, executing with a constrained compute budget, and even executing only partially observable programs.

In Chapter 2 we’ll look more closely at when and why neural networks struggle to learn the execution behavior of programs. Then we’ll aim to overcome those limitations with neural architectures that we design to model execution behavior. To this end, we’ll employ strategies for enabling neural networks to learn systematic and generalizable solutions.

### 1.3. Systematicity

To behave with systematicity means to behave according to precise and algorithmic rules, as opposed to behaving according to fuzzier pattern matching, intuition, or in a less principled, less organized manner. One of our goals, in seeking neural networks that can reason about program executions, is to obtain neural networks exhibiting systematicity. This aligns naturally with the precision and uniformity properties of programming language semantics, which govern the execution behavior of a program given the program’s source and its inputs.

The reason to seek solutions exhibiting systematicity is for improvements in systematic generalization as well as for desirable interpretability properties. A systematic solution that uses algorithmic rules often will continue to function outside of the range of inputs it was designed for. This differs from purely pattern recognition based approaches which may degrade in performance when tested in new environments. In a machine learning setting, this amounts to out of distribution generalization, where a model continues to exhibit good performance even when tested on a different distribution than the one it was trained on. For reasoning about programs, we specifically are interested in solutions that can learn about language semantics and then apply what they learn on unseen programs, possibly longer than any training program and using unseen compositions of language features. Systematicity in a solution can also improve its interpretability. A systematic solution may permit inspection, allowing a developer to understand what the latent variables represent. In some systematic solutions, the components of the solution can be composed, with the compositions retaining meaning. Examining the models introduced in this thesis, we observe both improvements in systematic generalization as well as interpretability benefits; the IPA-GNN learns to execute from small programs, and continues to work on longer programs, and the Exception IPA-GNN’s interpretability allows for predicting runtime error locations despite never seeing error locations during training.

Consider two strategies for imbuing a neural network with systematicity: systematicity through structure and systematicity through learning. Using structure to induce systematicity in a neural network means carefully selecting the neural architecture to promote systematicity in the functions the model learns. By varying the architecture, we modify a neural network’s inductive bias, which can make learning precise rules more or less likely. The contributions we consider in this thesis focus on systematicity through structure. As an alternative strategy,

one can use the training data and training technique, in conjunction with a general purpose architecture, to again encourage systematicity in the network’s solutions. My research makes contributions via both strategies, though we will focus only on systematicity through structure for the duration of this thesis. To illustrate the range of my contributions, I will now present an overview of my thesis contributions and my research contributions beyond those of this thesis.

## 1.4. Thesis Contributions

My thesis contributions center around the use of interpreter-inspired neural architectures and graph representations of programs for machine learning for program analysis. I facilitate research into the use of graph representations of programs for machine learning via the `python_graphs` library, and I introduce the instruction pointer attention graph neural network (IPA-GNN) family of architectures, showing its value in a variety of scenarios.

My first contribution is a Python library that enables the construction of graph representations of Python programs for use in machine learning systems. This library enables not only our work on IPA-GNN architectures for Python programs, but also has enabled research into graph representations of programs for machine learning along multiple other fronts. I showcase the research that this library enables and perform a case study illustrating properties of the graphs that the library produces on a dataset of millions of programs authored in a competition setting.

Second, I introduce the instruction pointer attention graph neural network (IPA-GNN) family of architectures. In this work we consider the tasks of learning to execute programs and learning to execute partial programs (programs for which a single statement has been masked.) We take special interest in systematic generalization, measuring the capability of models to generalize by learning from shorter programs and testing on longer programs. In both settings, full and partial program execution, the IPA-GNN approach yields improved generalization properties compared with general purpose models.

We next find that the IPA-GNN architecture is suitable for performing hardware verification, not just software analysis, with appropriate modifications. Adjusting the architecture to model the concurrent and repeating execution found in Verilog, a hardware description language, the resultant RTL IPA-GNN architecture performs well for predicting test coverage during hardware verification for chip design.

Finally we apply the IPA-GNN family of architecture to a developer assistance task, identifying runtime errors as static analysis. Though this task is typically performed via dynamic analysis, we show that IPA-GNN derived methods can also exhibit strong performance on this task while treating it as a static analysis task. As part of this work we contribute a new dataset derived from millions of competitive programming submissions to facilitate



further work in this direction. We also design an extension of the IPA-GNN that models Python exception handling, which we term the Exception IPA-GNN, and find that it exhibits useful interpretability, performing unsupervised error localization despite not being trained on error location data.

In total my thesis contributions show the value of imbuing neural networks with systematicity by modifying the neural architecture to more closely resemble a program interpreter. My research contributions also include introducing systematicity through other mechanisms, which we shall enumerate next, though we will not consider these in detail in the thesis.

## 1.5. Additional Contributions

Beyond the contributions presented in this thesis, I have made additional research contributions during my PhD, which I enumerate here.

First, I have made contributions toward applying machine learning to programming by example. In TF-Coder [131] and BUSTLE [111] we achieve systematicity through classical (non-neural) algorithms, namely via enumerative search. With TF-Coder we show that a carefully optimized search algorithm with aggressive value-based and argument-based pruning achieves useful performance on program synthesis in a practical domain, tensor manipulations. In BUSTLE we guide a bottom-up enumerative search via a neural network model in the inner loop of the search. These approaches achieve systematicity through structure not of their neural components, but rather by using machine learning components inside a non-neural algorithm.

Frequently in deep learning, program understanding is cast as natural language understanding, with programs treated as mere sequences of tokens without regard to their underlying semantics. This approach is often quite effective, but can also miss some of the advantages of leveraging our prior knowledge of how programs are structured and how they operate. GREAT [62] explores how we can simultaneously leverage structural knowledge of programs and still treat them as flat sequences. In GREAT, we introduce the Global Relational Embedding Attention Transformer as well as sandwich models. The former is a graph neural network that encodes relations between all nodes. The later alternates layers between a graph neural network with only local connectivity, and a Transformer (i.e. a graph neural network with global connectivity.) The key insight behind the success of these approaches is that there is independently valuable information in both the local and global structures of programs.

Across many domains in machine learning, including program synthesis and natural language processing, a common operation is sampling from probability distributions. Frequently it is desirable to sample without replacement. In Shi et al. [130] we introduce a more efficient

method for sampling without replacement that can act as a drop-in replacement for Python’s random number generation library.

In Vasic et al. [141] we show it’s useful to train a model to simultaneously localize and repair variable-misuse bugs. Supervision from one part of the task is useful for the other.

In Zhao et al. [163] we show the value of modeling edits to source code, rather than merely modeling source code as static language. Treating code as dynamic, rather than static, allows for models to take into account user intent, not just what the user has written so far. This modeling consideration allows models to have a conception of how code is developing by considering its history, rather than treating it as a static flat piece of text.

I have also made contributions toward multi-step reasoning with machine learning models. My works with IPA-GNN are one form of this, where each layer of the model corresponds to one or more steps of program execution. Separately, also motivated by the challenge of modeling program executions, we published Nye et al. [110]. In this work, we introduce a new way to prompt a large language model, where the model is encouraged to show its work before producing a final answer to a task, effectively giving the model a “scratchpad” space in which to think aloud before reaching a conclusion. This approach is effective for teaching a model to execute programs step-by-step.

As the field pushes the size and quality of large language models ever higher, we identify in Pei et al. [115] that large language models can reason about program behaviors both in a single pass or a multi-step manner, directly predicting invariants at program points in Java programs.

Looking beyond simple sequential decoding from a model toward drawing samples from models as part of larger programs, we published Dohan et al. [41]. Here we cast repeated interactions with probabilistic models in the language of probabilistic programming, showing how several recent techniques for eliciting knowledge from large language models fit this perspective.

Collectively, my contributions make progress toward the systematicity through machine learning for code through both structural and learning approaches. For the remainder of the thesis, we focus on interpreter-inspired neural network approaches that achieve their systematicity through structurally imposed inductive biases.

## 1.6. Thesis Outline

Here I provide a succinct outline of the remaining chapters of the thesis. In Chapter 2 I provide background on graph neural networks for source code (Section 2.2.4) and on execution-aware machine learning for source code (Section 2.2.5). In Article 1 I introduce the `python_graphs` library for constructing graph representations of Python programs, which enables our subsequent research. In Article 2 I introduce the Instruction Pointer Attention

Graph Neural Network (IPA-GNN) architecture, applied to the tasks of executing full and partial programs where it demonstrates systematic generalization improvements. In Article 3 I extend the IPA-GNN to support hardware designs in Verilog, where it achieves state of the art performance at predicting test coverage for chip design. In Article 4 I further explore the capabilities of the IPA-GNN family for predicting runtime errors, extending the IPA-GNN to model programs that raise exceptions, and I find the model's interpretability admits a new capability: unsupervised error localization. In the Conclusion I place these findings in a modern context and discuss the direction this work points us in going forward.



# Chapter 2

---

## Background

Advancing program analysis using machine learning is a core challenge in machine learning. To contextualize our work, we provide background on the diversity of program analysis tasks machine learning has been applied to and the range of neural approaches that have been considered. In doing so, we motivate the particular set of challenges that we consider in this thesis: learning to execute, coverage prediction, and runtime error prediction, all challenging program analysis tasks requiring reasoning about execution behavior.

### 2.1. Setup: Program analysis tasks

Machine learning has been applied to a wide range of program analysis tasks, yielding advancements in program analysis capabilities. We highlight here the diversity of these tasks, introduce the terminology that we will use throughout the thesis, and emphasize two important challenges that guide our work: systematic generalization, and the difficulty of modeling program execution behavior.

#### 2.1.1. The diversity of program analysis tasks

Neural network approaches have been applied to many program analysis tasks. We list several: variable misuse detection [4, 78], joint localization and repair [141], variable name prediction [2, 121], type inference [151], clone detection [58, 69, 147, 152, 158], vulnerability detection [155, 164], build error repair [138], method name prediction [7, 8], and various forms of code completion [29, 96, 102]. In variable misuse detection the task is to identify when the wrong variable has been used; this type of mistake commonly results from copy/paste actions, where the developer renames several instances of a variable but fails to rename all of them. In joint localization and repair, the task is to simultaneously locate a bug (e.g. a variable misuse bug) in code and identify what the correct version of the code should be. Most of these tasks are approached as static analysis, where the machine learning method must make a determination about the program under analysis given access only to its source code, but

not its inputs or an interpreter. It is quite common for researchers to study bugs that have been synthetically produced, e.g. by injecting a bug into otherwise clean code [4, 78, 118]. This is a useful strategy for producing data that allows for more rapid iteration, but it comes with drawbacks. There is a distribution shift when solutions learned from these synthetic data are applied to real-world code, causing the solutions to work less well in practice than the results on the synthetic data would suggest. To remedy this, one of our contributions (Article 4) is a new dataset consisting of millions of human-authored programs, many of which contain runtime errors. Unlike previous datasets, our contribution consists of bugs that have been verified by actually running the program of interest.

### 2.1.2. Machine learning terminology

To aid in introducing the background material, consider the fault detection task of identifying runtime errors. We use this as a representative task to introduce the terminology and notation used throughout the thesis, and we will study this task extensively in Article 4. Given a valid Python program  $x$  we must determine whether it contains a runtime error, and if so, what kind of runtime error it contains. This is indicated by an integer label  $y$  taking on one of  $K$  values. The label  $y = 0$  indicates no error, and  $y = 1, \dots, K - 1$  each indicate one of  $K - 1$  distinct error classes. We have a large dataset  $D = \{(x^i, y^i)\}$  of programs  $x^i$  each labeled with their runtime error class  $y^i$ . As is common in supervised learning,  $D$  is divided into train, validation, and test splits. When we consider other tasks, like learning to execute (Article 2) or coverage prediction (Article 3), we will use the same notation with the value  $K$  varying between tasks.

### 2.1.3. Systematic generalization

A primary interest of our study is systematic generalization. That is, we seek models that exhibit favorable generalization characteristics when evaluated on test datasets with distributions that are different, but systematically related to, the training distribution. This is important for program analysis because certain aspects of software are consistent across all programs written in the same language, while other aspects will be unique to a project, with no similar examples present in the training data. In particular, programming language semantics are stable, whereas variable and function names vary widely across programs, with projects frequently using identifiers that are not used in any other project. We are trying to improve systematic generalization so that our models will continue to work on new programs, even when they contain novel combinations of programming elements. We seek generalization properties that indicate a model has learned something fundamental and composable about the semantics of the programming language it is being trained on. Therefore in addition to independent and identically distributed (i.i.d.) train, validation, and test splits, in Article 2

we will also evaluate models on test splits with longer programs than any seen at training time.

#### **2.1.4. The challenge of reasoning about program execution**

Though neural approaches have demonstrated powerful capabilities on a wide range of program analysis tasks, recent evidence indicates general purpose neural approaches struggle on tasks that require reasoning about program execution. We present this evidence, and use it to motivate the tasks that we study in this thesis. First, Austin et al. [11] finds that large language models of code, even of 137 billion parameters, perform poorly on the task of executing small programs. This result holds both when the task is cast as few-shot prompting and as fine-tuning. This is significant because on other program analysis tasks, not directly related to program execution, large language models of code produce state of the art results. The result is echoed in the BIG-bench tasks [18]. BIG-bench is a large suite of tasks meant to challenge large language models, ranging across a wide variety of subject matter. It contains five tasks that resemble program execution, and all five present a challenge for even the largest models evaluated on the benchmark, including models as large as 540 billion parameters. This is reaffirmed yet again in our contribution in Nye et al. [110]. In this work we consider a method of prompting or tuning large language models where the model is instructed via examples to output the step-by-step reasoning for solving a task prior to producing the task solution. For the task of learning to execute, this approach has the model output a step-by-step concrete execution trace, rather than directly computing the output of program execution in a single step of decoding. The findings of this work reaffirm the difficulty language models exhibit in learning to execute programs even when fine-tuned directly on end-to-end program execution behavior, and also reveals an insight: when the model is trained to reason directly about the intermediate execution behavior of a program, not just its end-to-end behavior, it can begin to make progress on this challenging class of tasks. We will see similar results in Article 2 and Article 4, where on both learning to execute and runtime error prediction tasks, general purpose models like the Transformer, RNNs, and GNNs fall short of the performance of interpreter-inspired architectures that more directly model program executions.

This evidence that general purpose neural approaches struggle on tasks that require reasoning about program execution motivates our choice of tasks for this thesis. We study the following tasks in this work: learning to execute full and partial programs, test coverage prediction during hardware design and verification, and runtime error prediction in competitive programming. Each task is closely tied to the execution behavior of programs, seeming to require an understanding of the step-by-step execution of a program. Learning to execute full programs is the most clear-cut instance of this; the task is to predict the output of a program

given access to the program’s source code. In these problems, the program inputs have been hard coded into the programs and the programs do not rely on external dependencies. A change at any point in a program will change the program behavior at all downstream program points. The challenge of the task is for the model to learn the programming language semantics from a large number of examples, and ideally to do so in a manner that generalizes from shorter programs to longer programs. The task of learning to execute partial programs again requires the model predict the output of a program, but now the model has restricted access to the program, with one line occluded by a mask. This task is more difficult, with perfect accuracy impossible due to inherent uncertainty about the code behind the mask. To succeed on this task, the model must learn to model both the language semantics and the uncertainty, in order to make predictions about the program output despite imperfect information. Test coverage prediction during hardware verification requires modeling a different type of execution, as the semantics of the hardware description language differ from those of a typical software programming language like Python. Modeling execution of tests of hardware requires accounting for the concurrent and repeating nature of the blocks found in hardware descriptions. The final task we consider, runtime error prediction, is directly applicable to the construction of developer tools. We examine it in the context of competitive programming, introducing a runtime error prediction dataset containing millions of programs and hundreds of thousands of runtime errors encountered by competitive programmers. These errors arise from the execution behavior of programs across many program points, and the source of the bug in code is not always the same location as the program point where the error is raised by the program. Each of these tasks strains the capabilities of general purpose neural network approaches since they require reasoning about the fine details of program executions.

## **2.2. Approach: Neural Network Methods**

The first step in processing source code with neural networks is to select a representation of source code. We’ll review the choices and trade-offs one makes in selecting a representation of source code for processing with neural architectures. We’ll then review several neural approaches to processing source code including general purpose sequence models and graph neural networks, and more specialized interpreter-inspired neural network architectures.

### **2.2.1. Sequence Representations of Source Code**

When source code is written by a human and saved to disk, the representation of each file is a sequence of characters. Files are given filepaths, themselves sequences of characters describing a hierarchy indicating where a source code file is stored. Though the interactions



between files are critical when developing real world large scale applications, we'll keep our focus on representations of individual source code files to begin with.

Like ordinary natural language text, a source code file can be comprised of arbitrary characters in arbitrary arrangements, including non-ascii unicode characters. In practice, however, source code tends to have some simplifying properties. Lines often are limited in length to 80 or 120 characters. Most programming languages use keywords consisting only of ascii characters, and so the use of non-ascii characters tends to be relegated to strings and sometimes variable names.

### 2.2.2. Graph Representations of Source Code

Importantly, source code is highly structured in a way that natural language texts are not. Valid programs adhere to a strict grammar, which varies between programming languages. This adherence to a strict grammar has a few implications. First, it contributes to the shape of the distribution of text appearing in source code repositories. Keywords are repeated considerably, and the strict grammatical structure results in some n-grams having elevated frequency, while at the same time there is a long tail of tokens with few occurrences (e.g. occurring only in a single file or project) since programmers can choose arbitrary variable names. Never before seen variable names at test time are common, and a “rare” token like this will occur not just once when it does occur, but potentially many times.

Second, the strict grammar admits additional non-sequential representations of source code. Though files are viewed as character sequences by programmers, and stored as character sequences on disk, it is much more natural to think about a program in terms of the structures that it contains. We will introduce the abstract syntax tree and control-flow graph to provide two such ways of viewing and representing a program.

An abstract syntax tree (AST) is a tree representation of source code. Each node in an abstract syntax tree represents some programming construct, such as a module, function, while-statement, expression, operation, or variable name. Rather than relying on syntactic structures like braces or whitespace to group related elements, the edge relations in the AST convey these relationships. For example, the body and condition of a while loop are each represented by nodes with their parent given by the while loop's node. An abstract syntax tree omits information from the source code that is purely cosmetic, such as parenthesis and whitespace placement, and comments. This information can be important for certain learning algorithms, and in those settings an extension to the abstract syntax tree known as a full syntax tree can be used. The abstract syntax tree is also a fundamental building block for processing source code further for computing control-flow graphs, data-flow graphs, or for compiling source code to other representations.

A control-flow graph (CFG) is a graph structure representing the possible execution paths through a program. Typically, the nodes in a control-flow graph are basic blocks. A basic block is a straight-line section of code, with no exits or entrances in the middle of the block. A directed edge from node  $n_1$  to  $n_2$  in a control-flow graph represents that execution may flow from  $n_1$  to  $n_2$ ; that is,  $n_2$  may be executed immediately following the completion of  $n_1$ .

In our work we will make the most use of statement-level control-flow graphs. In this kind of control-flow graph, nodes represent individual statements rather than full basic blocks. As before, edges represent the possible paths control may flow along.

### 2.2.3. Sequence Models

We now introduce recurrent neural networks (RNNs) and the Transformer architecture, two neural network architectures well suited for processing sequential data.

**Recurrent Neural Networks.** The recurrent neural network architecture is well suited for sequence processing tasks. It operates on sequential inputs of arbitrary length by successively applying the same operation with shared weights to each element of the input sequence, and maintaining a “memory” across applications.

Given an input sequence  $x_i$ , the simplest possible RNN – the vanilla RNN – computes its memory at time step  $t$  as  $h_t = \text{ReLU}(W^{(i)}x_t + W^{(h)}h_{t-1} + b)$ . Here,  $W^{(i)}$ ,  $W^{(h)}$ , and  $b$  are learned parameters, with superscripts  $(i)$  and  $(h)$  indicating the weight matrices for the input and hidden state respectively.

A desirable property of a sequence processing architecture is to be able to perform long chains of sequential reasoning. On their own, RNNs struggle to learn long-range dependencies. Multiple techniques use gating to mitigate this issue. The Gated Recurrent Units (GRU) and Long Short-Term Memory (LSTM) architectures each take advantage of gating to improve upon the RNN’s capability of handling long-range dependencies.

The LSTM architecture makes two key advancements over the vanilla RNN. The first is that it includes the addition of a memory distinct from its output. The second is its use of gating to determine dynamically when and how much to update the memory.

At each step of an LSTM RNN, a candidate memory  $\tilde{C}_t = \tanh(W^{(Ci)}x_t + W^{(Ch)}h_{t-1} + b^{(C)})$  is proposed. Aside from using a different nonlinearity, this is performing the same computation as the vanilla RNN. However, rather than blindly accepting this value as the new memory and the new output, the LSTM dynamically determines how to incorporate  $\tilde{C}_t$  into the memory and output.

It does so through the use of three gates: the input, forget, and output gates. Each are computed as a nonlinear function of the input and current cell memory:

$$\begin{aligned}i_t &= \sigma(W^{(Ii)}x_t + W^{(Ih)}h_{t-1} + b^{(I)}) \\f_t &= \sigma(W^{(Fi)}x_t + W^{(Fh)}h_{t-1} + b^{(F)}) \\o_t &= \sigma(W^{(Oi)}x_t + W^{(Oh)}h_{t-1} + b^{(O)})\end{aligned}$$

The input and forget gates are used to determine how much of the candidate memory and existing memory respectively will contribute to the new value of the network’s memory according to  $C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$ . This allows the network to use its inputs and past outputs to determine which information to keep and which to discard, which aids in handling long-range dependencies in its inputs.

Finally the LSTM computes its per time step output as  $h_t = o_t * \tanh(C_t)$ . By separating its memory from its outputs and through its use of gating, the LSTM improves over the vanilla RNN by being able to selectively store information needed for future predictions for long durations.

RNNs are additionally prone to the problem of exploding gradients when trained on long sequences. “Exploding gradients” can occur when large gradient errors accumulate over the many steps of an RNN. Techniques for mitigating this include gradient clipping, regularization, and truncated backpropagation through time (TBPTT). Gradient clipping refers to the practice of imposing a maximum norm on the gradients used in gradient-based optimization. As regularization, adding an L1 or L2 norm penalty to the loss function is a common mitigation. TBPTT mitigates exploding gradients by only performing a limited number of steps of backpropagation, limiting the amount of error accumulation that can occur.

We observe that some tasks, such as processing a linear sequence of steps in order, share the causal structure modeled by RNN architectures – a linear chain. These tasks are particularly well suited for processing with RNNs, and this observation will inform our design of the IPA-GNN architecture.

**Transformer Architecture.** We next introduce the Transformer. Like RNNs, the Transformer architecture processes sequences, though it uses a finite fixed length receptive field to do so. Variants of the Transformer architecture achieve state of the art results on machine translation and other natural language processing tasks.

A central element of the Transformer is multi-head attention. Given three equal length sequences – queries  $Q$ , keys  $K$ , and values  $V$  – the *attention* operation computes an output sequence as a weighted sum of the values. The weights for the  $i^{\text{th}}$  element of the output are given by the scaled dot-product similarity between the  $i^{\text{th}}$  query and all keys. We say that

element  $i$  attends to the values of the sequence with these weights. The resulting values are given by  $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_K}}\right)$ .

In multi-head attention with  $H$  heads, the architecture produces  $H$  different key, query, and value sequences from the same inputs using distinct learned weight matrices  $W_h^Q$ ,  $W_h^K$ , and  $W_h^V$ . For head  $h$ , we write  $Q_h = QW_h^Q$ ,  $K_h = KW_h^K$ , and  $V_h = VW_h^V$ . Each head attends independently to a distinct representation of the values, giving  $\text{head}_h = \text{Attention}(Q_h, K_h, V_h)$ , and the resultant values are combined:  $\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_H)W^O$ .

Transformer models may follow an encoder-decoder structure. Each layer of the encoder uses self-attention, meaning the keys, queries, and values are all derived from the same input, namely the output of the previous encoder layer. Each layer of the decoder uses attention twice, once to attend over the outputs of the encoder, and once to process the output auto-regressively. In the attention layer processing the output, a causal mask is used to prevent information leakage. The architecture described thus far treats the input as a set; to distinguish elements of the input by their order, positional encodings are added to its embeddings. We refer the reader to Vaswani et al. [143] for additional architectural details, including the use of skip connections and layer normalization.

**Comparison with RNNs.** Unlike RNNs, the Transformer architecture has a finite fixed size context window and so cannot process input sequences of arbitrary length. In spite of this, the Transformer architecture has excelled at sequence processing tasks such as machine translation, and variations of the Transformer commonly achieve state of the art results. While the limited context window appears as a drawback relative to RNNs, in practice RNN architectures struggle to remember inputs for as many steps as commonly used as the size of a Transformer’s context window. The Transformer exhibits the following advantages. It scales well and trains efficiently using teacher forcing. Through its use of attention, the gradient paths end up smaller than in comparable RNNs, and full information flow between all pairs of locations is achieved with every step. This allows it to learn long range dependencies with as much ease as short range dependencies.

Also unlike RNNs, which naturally model a linear chain causal structure, the causal structure most closely modeled by a Transformer is an all-to-all causal structure. This makes this Transformer architecture not well suited for systematic generalization in sequence length.

## 2.2.4. Graph neural networks

Since it is natural to represent programs as graphs containing structural information, it is useful to study graph neural networks (GNNs), neural architectures that process graph data structures. We introduce message passing neural networks (MPNNs) as a general framework for processing graph structured data. We then describe two specific graph neural

networks, Graph Attention Networks (GAT) and Gated Graph Neural Networks (GGNN), both instances of MPNNs.

**Message Passing Neural Networks.** Consider a directed graph  $G = (V, E)$  with vertices  $v_i \in V$  and edges  $e_i \in E$ . Suppose each vertex is annotated with an embedding  $h_{0,i} \in \mathbb{R}^H$ , and each edge is annotated with discrete edge type  $e_i \in \{1, \dots, T\}$ . A single layer of a message passing neural network computes a new hidden state  $h_{t,i}$  through a few steps:

- (1) It may compute per-node messages as a function of the previous hidden state at the nodes  $a_{t,i}^{(1)} = f^{(1)}(h_{t-1,i})$ .
- (2) It may compute per-edge messages as a function of the node messages and the edge-type  $a_{t,i,j}^{(2)} = f^{(2)}(h_{t-1,i}, e_{i,j})$
- (3)  $h_{t,i}$  is computed as an aggregation of all messages incident on  $i$ ;  $h_{t,i} = f^{(3)}(a_{t,i}^{(1)}, \cup a_{t,i,j}^{(2)})$ .

In this formulation, information is propagated along edges at each timestep. Typically these neural networks are trained using a fixed number of layers, thereby inducing a radius around each node which can affect the node’s final value.

**Gated Graph Neural Network.** The gated graph neural network (GGNN) is an MPNN with the following components. Per-edge messages are computed as a dense layer applied to the node embeddings, where the dense weights vary by edge type.  $a_{t,i,j}^{(2)} = W^{(e_{i,j})} h_{t-1,i}$ . The incoming messages to each node are averaged and a GRU layer gates the aggregation of this average with the existing hidden state  $h_{t,i} = \text{GRU}(h_{t-1,i}, \mu(\cup a_{t,i,j}^{(2)}))$ .

**Graph Attention Network.** The GAT architecture also conforms to the MPNN framework. It uses attention to update the node values, similar to the attention operation used by the Transformer. Though the initial GAT paper does not support edge types, the architecture is easily extended to a “relational graph attention network” (R-GAT) to support edge types in a manner analogous to the GGNN.

### 2.2.5. Interpreter-inspired machine learning models

Several neural approaches are more specialized than the general purpose sequence and graph processing architectures described so far, loosely mimicking components of a program interpreter. We review these interpreter-inspired architectures briefly here. Neural Turing Machines [54] augment a recurrent neural network with an external memory, allowing the network to perform reads and writes to the memory as differentiable operations. In Neural Programmer [107], a neural model is augmented with a series of discrete non-differentiable operations, which the model may apply in a differentiable manner, thereby learning discrete programs through gradient descent. The Neural Programmer-Interpreter [123] acts like a program interpreter in that it uses a recurrent neural network trained to perform computation as seen in detailed program traces, as well as to select learned subprograms to call and what arguments to call them with. The Neural Random-Access Machine model [85] has operations

for reading and writing to an external memory at locations specified by pointers, which, similar to the Neural Programmer, may be called in a differentiable manner. The Neural GPU applies a convolutional gated recurrent unit (CGRU) at each layer and is thereby able to learn algorithms from examples and apply them on larger examples. Bošnjak et al. [20] implements a differentiable interpreter for the Forth programming language that admits programs with differentiable slots whose behavior can be learned from data.

Continuing in this tradition, the models we focus on in this thesis are interpreter-inspired and draw in particular on an underexplored facet of program interpreters for understanding program executions: control-flow. The IPA-GNN family of architecture that we introduce in Article 2 will perform learned executions of programs. It will proceed step-by-step through the program under analysis, respecting its control-flow structure, while simultaneously considering multiple plausible paths through the program and learning additional language semantics only from end-to-end examples.

## Article 1.

# A Library for Representing Python Programs as Graphs for Machine Learning

by

David Bieber<sup>1</sup>, Kensen Shi<sup>1</sup>, Petros Maniatis<sup>1</sup>, Charles Sutton<sup>1</sup>,  
Vincent Hellendoorn<sup>1</sup>, Daniel Johnson<sup>1</sup>, and Daniel Tarlow<sup>1</sup>

This article was published in 2022.

I present my contributions and the contributions of the coauthors.

I designed, implemented, and open sourced the `python_graphs` library, including control-flow and data-flow analysis, program graph construction, and cyclomatic complexity; I performed the literature review of graph representations of programs for machine learning research; I applied `python_graphs` at scale to Project CodeNet programs, conducting the case study presented within; I authored the article reproduced here.

Kensen Shi performed extensive analysis on the program graphs obtained from two datasets and added essential documentation to the `python_graphs` project. Petros Maniatis made important contributions to the object-graph format and toward marshalling graphs between protocol buffer formats. Charles Sutton made contributions on the program graphs protocol buffer representation, performed Python 2 to Python 3 updates, and contributed to the hole-filling dataset generator. Vincent Hellendoorn added indentation tracking to the library's syntax node construction and contributed toward Python 3 compatibility. Daniel Johnson made the `python_graphs` library enforce having exactly one program graph node per AST node and ensured deterministic read orders. Daniel Tarlow provided many hours of

advising and feedback on this project along the way.

**ABSTRACT.** Graph representations of programs are commonly a central element of machine learning for code research. We introduce an open source Python library `python_graphs` that applies static analysis to construct graph representations of Python programs suitable for training machine learning models. Our library admits the construction of control-flow graphs, data-flow graphs, and composite “program graphs” that combine control-flow, data-flow, syntactic, and lexical information about a program. We present the capabilities and limitations of the library, perform a case study applying the library to millions of competitive programming submissions, and showcase the library’s utility for machine learning research.

**Keywords:** graph neural networks, representations of source code, static analysis



# 1. Introduction

In this report we present `python_graphs`<sup>1</sup>, a Python library for constructing graph representations of Python programs for use in machine learning research. This report details the capabilities and limitations of the library as they pertain to applying machine learning to source code.

A standard class of approaches in applying machine learning to code is to construct a graph representation of a program, and then to perform the analysis of interest on that graph representation, learning from a large dataset of labeled example programs. Graph representations of programs used for machine learning include the abstract syntax tree (AST), control-flow graph (CFG), data-flow graphs, inter-procedural control-flow graph (ICFG), interval graph, and composite “program graphs” that encode information from multiple of the aforementioned graphs, possibly with additional program-derived data.

The `python_graphs` library directly allows for the construction of some of these graph types (e.g., control-flow graphs and composite program graphs) from arbitrary Python programs, and it provides tools that aid in constructing the others. It has been used successfully in a variety of machine learning for code publications, and we make it available as free and open source software to allow for broader use.

In Section 2 we present an overview of the use of graph representations of code in machine learning. In Section 3 we describe the capabilities (Section 3.1), possible extensions (Section 3.2), and limitations (Section 3.3) of `python_graphs`. In the context of this thesis, the `python_graphs` library enables the construction of the graph structures supporting the research in Articles 2 and 4. Section 4 highlights the applications of `python_graphs` for machine learning research, showcasing its utility beyond our own research efforts. Section 5 presents a case study applying `python_graphs` to 3.3 million programs from Project CodeNet [119].

## 2. Background

**Graph representations of code in machine learning.** Graph representations of source code are regularly used in machine learning research. Most common among these is the abstract syntax tree. Several works learn directly from ASTs [7, 8, 9, 24, 81, 93, 99, 147, 148, 152, 158, 160, 165] or produce an AST as output [120, 156], while Johnson et al. [70] learns to dynamically augment an AST with new edges useful for a downstream task. Other works operate on a program’s control-flow graph [15, 16, 36, 146] or data-flow graph [58, 79], or joint control and data flow graph (CDFG) [142]. A typical composite program graph uses an AST backbone with some subset of control-flow, data-flow, lexical, and syntactic information encoded as additional edges [4, 5, 62, 90, 92, 155, 161, 164]. Swarna et al. [137] meanwhile

---

<sup>1</sup><https://github.com/google-research/python-graphs>

uses AST, CFG, and program dependence graph (PDG) representations concurrently, without unifying them into a single graph. Pashakhanloo et al. [113] forms a graph via CodeQL queries to a database representing a program. Georgiev et al. [48] forms a hypergraph, where edges can connect more than two nodes, containing again control-flow, data-flow, lexical, and syntactic information. Still other representations include a program’s interval graph [149] or a graph formed from the pointers in the heap [94]. A graph can also encode additional information, e.g., as in Tarlow et al. [138] which constructs a graph jointly representing code, a compiler error, and a build file.

Our work directly admits constructing control-flow graphs, performing data-flow analyses, and constructing certain composite program graphs from Python programs (Section 3.1). It can also be extended for constructing interprocedural control-flow graphs, novel composite program graphs, additional data-flow graphs, or span-mapped graphs (Section 3.2).

**Tools for constructing graph representations.** We compare `python_graphs` with existing Python static analysis tools. Tree-sitter [23] can build a concrete syntax tree for a given source file and update it incrementally as the source changes. It supports over 40 languages including Python. Our system must operate directly on the built-in Python AST rather than a language agnostic syntax tree. CodeQL [105] is a query language for source code. These queries admit searching for control-flow and data-flow paths in source code. `pycfg` [53] generates control-flow graphs from Python source in a similar manner to `python_graphs`, but lacks support for certain language features like exceptions and generators. Scalpel [89] similarly generates control-flow graphs from Python and also performs additional static analyses, e.g., call graph construction. `python_graphs` performs data-flow analyses on top of its control-flow graphs, producing composite program graphs containing control-flow, data-flow, syntactic, and lexical information in one graph.

### 3. Capabilities, Possible Extensions, and Limitations

We provide a comprehensive overview of the capabilities of the `python_graphs` library, a discussion of how `python_graphs` can enable still further capabilities (i.e. assisting in constructing the graph types not directly supported by the library today), and a discussion of the library’s limitations.

#### 3.1. Capabilities

The `python_graphs` library enables a number of static analyses on Python source code. The main use cases are computing control-flow graphs, performing data-flow analyses, constructing composite “program graphs,” and measuring cyclomatic complexity of Python programs and functions [1]. Each of these operations may be applied to a full Python program or an individual Python function. The library handles any of the following input

<pre> 1  def fn1(): 2      x = 0 3      for i in range(5): 4          x += i 5      return x </pre> <p>(a) Example function under analysis. <code>python_graphs</code> accepts function objects such as <code>fn1</code>.</p> <pre> 1  import ast 2  syntax_tree = ast.parse(source) </pre> <p>(c) AST object for code in the string <code>source</code>.</p>	<pre> 1  import inspect 2  source = inspect.getsource(fn1) 3 4  assert source == """def fn1(): 5      x = 0 6      for i in range(5): 7          x += i 8      return x 9  """ </pre> <p>(b) Source for <code>fn1</code> as a string object. The built-in <code>inspect</code> module facilitates accessing source for functions when available.</p>
---	--

**Fig. 1.** The input formats accepted by the `python_graphs` library are (a) function, (b) source code, and (c) AST. The code snippets here demonstrate construction of each input format for the example function `fn1`.

types: (a) Python function, (b) source code string, or (c) abstract syntax tree. Figure 1 shows constructing all three valid input formats for a sample program. In all cases, the library first converts the input to an abstract syntax tree for analysis.

3.1.1. Control-Flow Graphs. A control-flow graph represents the possible paths of execution through a program. Each node in a control-flow graph represents a basic block. A basic block is a straight-line section of source code that is executed contiguously. The only branches into a basic block enter at the start, and the only branches out of a basic block exit at the end (other than Exceptions). An edge in a control-flow graph represents a possible path of execution. There is an edge between node A and node B in a program’s control-flow graph if and only if it is possible to execute basic block B immediately following the conclusion of executing basic block A [1].

In addition to producing these standard control-flow graphs, the `python_graphs` library can also produce statement-level control-flow graphs. A node in a statement-level control-flow graph represents a single line or instruction, rather than a complete basic block. An edge between two nodes indicates that the two lines may be executed in succession. Figure 2b shows the statement-level CFG for the `fn1` program.

A control-flow graph is useful for machine learning for source code in two respects. First, it is a useful representation of code suitable for processing with graph neural networks, for example in Bieber et al. [15, 16]. Second, the control-flow graph forms the basis for a number of further analyses including data-flow analyses (liveness analysis, reachability, etc.), computing cyclomatic complexity, and constructing program graphs, each implemented by the `python_graphs` library.

In Python, any line of code can raise an exception. Taking this form of execution into account, this limits basic blocks to a single line of code, since a raised exception is an exit

```

1 # 1. Use control_flow to construct a CFG.
2 from python_graphs import control_flow
3 graph = control_flow.get_control_flow_graph(fn1)

1 # 2. Access a particular basic block by source.
2 block = graph.get_block_by_source("x += i")

1 # 3. Convert the CFG to a pygraphviz.AGraph.
2 from python_graphs import control_flow_graphviz
3 agraph = control_flow_graphviz.to_graphviz(graph)

```

(a) Example usage of `python_graphs` to (1) construct a CFG, (2) access basic blocks by source, and (3) convert to `pygraphviz` for visualization.

<i>n</i>	SOURCE	CFG
1	<code>x = 0</code>	
2	<code>.0 = range(5)</code>	
3	<code>i = next(.0)</code>	
4	<code>x += i</code>	
5	<code>return x</code>	

(b) The statement-level control-flow graph for the program `fn1` introduced in Figure 1. `.0` denotes the iterator constructed by the for loop in `fn1`.

**Fig. 2.** `python_graphs` supports construction and analysis of control-flow graphs for arbitrary Python functions.

branch. Rather than restrict basic blocks to a single line of code, we take a more pragmatic approach, and introduce a second optional edge type, “interrupting edges”, in our control-flow graph data structure that represents control flow due to exceptions. An interrupting edge from block A to block B indicates that an exception raised during the execution of A can cause control to flow to block B. `python_graphs` control-flow graphs can be used with or without these interrupting edges.

To construct a control-flow graph with `python_graphs`, use the `control_flow` module’s `get_control_flow_graph` as in Figure 2.

3.1.2. Data-Flow Analyses. A data-flow analysis computes information about how the variables in the program are used, such as which variables are *live* at a given program location. A live variable is one that may be read at some point in the future before its value is overwritten. The `python_graphs` library implements two best-effort data-flow analyses: liveness and last-access analysis.

Data-flow analyses are performed through iterative application of the data-flow equations until a fixed point is reached. The `python_graphs` library supports both forward and backward data-flow analysis, and so can be extended to support additional data-flow analyses. Liveness is implemented as a backward analysis, and last-access as a forward analysis.

An example of using the liveness analysis to obtain the set of loop variables in a while loop is provided with the library, a necessary step in rewriting Python while loops into their functional form. The `data_flow` module provides the data-flow analyses, as in Figure 3.

3.1.3. Composite Program Graphs. The `python_graphs` library implements a single kind of composite program graph, based closely on that of Allamanis et al. [4]. In this document we refer to these composite program graphs simply as “program graphs”, though of course other kinds of program graphs are possible, with different node and edge types.

```

1 # 1. Use data_flow to perform liveness analysis.
2 from python_graphs import data_flow
3 analysis = data_flow.LivenessAnalysis()
4 for block in graph.get_exit_blocks():
5     analysis.visit(block)

1 # 2. Construct a program graph with program_graph.
2 from python_graphs import program_graph
3 pg = program_graph.get_program_graph(program)

1 # 3. Access a particular node by source.
2 node = pg.get_node_by_source_and_identifier(
3     'return x', 'x')

```

(a) Example usage of `python_graphs` to (1) perform liveness analysis on a program’s control-flow graph. Independently, (2) shows constructing a composite program graph, and (3) accessing one of its node by source.

#	SOURCE	Live in	Live out
1	<code>x = 0</code>	$\emptyset$	
	<code>.0 = range(5)</code>		$\{x\}$
2	<code>i = next(.0)</code>	$\{x\}$	$\{x, i\}$
3	<code>x += i</code>	$\{x, i\}$	$\{x\}$
4	<code>return x</code>	$\{x\}$	$\emptyset$

(b) The results of the liveness data-flow analysis. Live in and live out indicate the variables that are live at the start and end of each basic block respectively. # denotes basic block number. The analysis is for the program `fn1` introduced in Figure 1 and Figure 2.

**Fig. 3.** Example usage of data-flow analysis and program graph construction in `python_graphs`.

A program graph has the abstract syntax tree of the program it represents as its backbone. Each node in the program graph directly corresponds to a single node in the AST, and vice versa. Lists and primitive values in the AST have corresponding nodes in the program graph as well. Corresponding to each syntax element in the program (leaf nodes in the AST) is a syntax node in the program graph. Each edge in the AST also appears in the program graph. The program graph then has additional edges representing the following relationships between program pieces: “NEXT\_SYNTAX”, “LAST\_LEXICAL\_USE”, “CFG\_NEXT”, “LAST\_READ”, “LAST\_WRITE”, “COMPUTED\_FROM”, “CALLS”, “FORMAL\_ARG\_NAME”, and “RETURNS\_TO”. Collectively, the edges in a program graph convey control-flow, data-flow, lexical, and syntactic information about the program.

We summarize the edge types and their meanings in Table 1. These edge types are also useful for constructing other graph types (Section 3.2): interprocedural control-flow graphs, data-flow graphs, and other composite program graphs.

The `python_graphs` library provides the function `get_program_graph` for constructing a program graph from any of the supported input types (source code, an abstract syntax tree, or Python function). Figure 3a shows example usage.

Table 2 lists several programs along with their control-flow graphs and program graphs as computed by `python_graphs`.

3.1.4. Cyclomatic Complexity. Cyclomatic complexity is a standard measure of program complexity based on the set of possible paths through a program. It measures the number of linearly independent execution paths through a program. The `python_graphs` library can compute the cyclomatic complexity of a Python function. This functionality is available via the function `cyclomatic_complexity`, which accepts a Python function (as source, AST, or

Python function object) and returns an integer. To compute the cyclomatic complexity of a program, `python_graphs` first constructs its control-flow graph. In a control-flow graph with  $E$  edges,  $N$  nodes, and  $P$  distinct connected components, the cyclomatic complexity  $M$  is given by  $M = E - N + 2P$ .

## 3.2. Possible Extensions

The following capabilities are possible to implement using `python_graphs`, but are not directly provided by `python_graphs` out of the box.


3.2.1. Alternative Composite Program Graphs. The program graphs generated by `python_graphs`'s `get_program_graph` make specific choices for what nodes and edges are included in the graph. Other choices are possible. For alternative composite program graph schemes, the source of `python_graphs`'s `get_program_graph` function serves as an illustrative example for how to construct a composite program graph with the desired set of nodes and edges.

3.2.2. Inter-procedural Control-Flow Graphs. `python_graphs`'s `get_control_flow_graph` function constructs the control-flow graph for a single function or program; it does not include edges indicating control flow between functions.

**Table 1.** Descriptions of the edge types supported by `python_graphs`. A directed edge from node *src* to node *dest* has the semantics given in this table.

EDGE TYPE	DESCRIPTION
FIELD	<i>dest</i> is a field of AST node <i>src</i> .
NEXT_SYNTAX	<i>dest</i> is the syntax element immediately following <i>src</i> in top-to-bottom left-to-right order.
LAST_LEXICAL_USE	The variable at node <i>dest</i> has its previous appearance at <i>src</i> in top-to-bottom left-to-right order.
CFG_NEXT	The statement indicated by <i>dest</i> can be executed immediately following that indicated by <i>src</i> .
LAST_READ	When <i>src</i> is about to execute, it may be that the variable at <i>src</i> was most recently read at <i>dest</i> .
LAST_WRITE	When <i>src</i> is about to execute, it may be that the variable at <i>src</i> was most recently written to at <i>dest</i> .
COMPUTED_FROM	<i>src</i> indicates a variable on an assignment's left hand side, and <i>dest</i> a variable on its right hand side.
CALLS	<i>src</i> is an AST call node, and <i>dest</i> is the definition of the function being called.
FORMAL_ARG_NAME	<i>src</i> is an argument in a function call; <i>dest</i> is the corresponding parameter in the function definition.
RETURNS_TO	<i>src</i> is the return node in a function definition, and <i>dest</i> is the AST call node that calls that function.

**Table 2.** Example programs and their associated control-flow graphs and program graphs. Enlarged versions of the program graph figures are included in Appendix A.1.

#	SOURCE	<i>n</i>	STATEMENT	CFG	PROGRAM GRAPH
1	<pre>def fn1(a, b):     return a + b</pre>	1	a, b ← args		 Figure 17
2	<pre>def fn2(a, b):     c = a     if a &gt; b:         c -= b     return c</pre>	1	a, b ← args		 Figure 18
3	<pre>def fn3(a, b):     c = a     if a &gt; b:         c -= b         c += 1         c += 2         c += 3     else:         c += b     return c</pre>	1	a, b ← args		 Figure 19
4	<pre>def fn4(i):     count = 0     for i in range(i):         count += 1     return count</pre>	1	i ← args		 Figure 20
5	<pre>def fn5(i):     count = 0     for _ in range(i):         if count &gt; 5:             break         count += 1     return count</pre>	1	i ← args		 Figure 21
6	<pre>def fn6():     count = 0     while count &lt; 10:         count += 1     return count</pre>	1	count = 0		 Figure 22
7	<pre>def fn7():     try:         raise ValueError('N/A')     except ValueError as e:         del e     return</pre>	1	raise ValueError('N/A')		 Figure 23
8	<pre>def fn8(a):     a += 1</pre>	1	a ← args		 Figure 24

An interprocedural control-flow graph (ICFG) is a control-flow graph that shows the control flow possible between functions, not just within a function. We can view an ICFG as a composite program graph consisting of the control-flow graphs of a program and all its functions, as well as `CALLS` and `RETURNS_TO` edges indicating the possible interprocedural control flows in the program. `python_graphs` provides the capability for constructing the necessary control-flow graphs and additional edges, making it possible to write a function to construct ICFGs as well.

3.2.3. Data-Flow Graphs. A data-flow graph represents the data dependencies present in a program. The nodes in a data-flow graph represent the variable access locations in a program, and the edges in a data-flow graph denote relationships between these accesses. An example of such a relationship is an edge with *dest* indicating where a variable is assigned to and *src* indicating where the assigned value is subsequently used (equivalent to `python_graphs`'s `LAST_WRITE` edges). We can therefore view data-flow graphs as composite program graphs consisting of a subset of AST nodes (just those representing variable accesses) and selected edge types like `LAST_READ` or `LAST_WRITE`.

3.2.4. Span-Mapped Graphs. In order to use the graphs produced by the `python_graphs` library for machine learning applications, it can be useful to tokenize the sections of code corresponding to each node. We suggest two approaches to handling this: (1) whole program tokenization and (2) per-node tokenization.

In *whole program tokenization* we tokenize the entire program first. Then, using `python_graphs`, we create a graph structure for the program. Finally we extract for each node the span of tokens from the whole program tokenization corresponding to that node. This approach allows for the possibility that a token consisting of multiple characters will be part of two consecutive nodes, and we must choose which node(s) to associate that token with.

In *per-node tokenization*, we instead split the program source into chunks according to which node in the graph they are part of, and then tokenize those chunks independently.

The key data required by these approaches is a mapping from a graph node to a span in the textual representation of program source (approaches 1 and 2) or to a span in the tokenized representation of program source (approach 1 only). We call a graph augmented with this data a *span-mapped graph*. Both approaches are possible using `python_graphs`. Bieber et al. [16] uses approach 1, with code freely available online<sup>2</sup>. This same code example is informative for any project wishing to implement approach 2.

3.2.5. Additional Data-Flow Analyses. `python_graphs` implements liveness and last-access data-flow analyses, and provides a framework for implementing additional analyses. This

---

<sup>2</sup><https://github.com/google-research/runtime-error-prediction>



framework allows somewhat straightforward implementation of definite assignment analysis or computing reaching definitions, for example.

### 3.3. Limitations

Python source code is difficult to analyze statically because so much of Python’s behavior is determined dynamically. We perform a “best-effort” analysis, which we do not guarantee will be correct considering all of Python’s language features. Inspection in Python allows manipulation of stack frames or of local or global variables, causing hard-to-detect effects on data and control flow. `eval` and `exec` permit the execution of arbitrary code constructed dynamically and inaccessible to our analysis. Operations can be overloaded dynamically, so e.g. even a simple addition operation can have effects overlooked by our analyses. These language features are empowering to Python users, but restrict the guarantees our analyses can provide.

## 4. Use cases

We next show how the `python_graphs` library is used in machine learning research. Uses include both building graph representations of programs as inputs to neural networks, and providing supervision for models that output graphs.

### 4.1. Graph Representations as Model Inputs

**Instruction Pointer Attention Graph Neural Networks.** The instruction pointer attention graph neural network (IPA-GNN) model family [15, 16] operates on control-flow graphs as its primary input. IPA-GNN architectures then perform a soft execution of the input program, maintaining a soft instruction pointer representing at each model step a probability distribution over the statements in a program. These works use `python_graphs` to produce the control-flow graphs for the programs under consideration, which include both simple synthetic programs [15] and complex human-authored programs from a competition setting [16].

The original IPA-GNN work [15] uses the control-flow edges as produced by `python_graphs`’s default settings, and represents each statement with a 4-tuple of values, which is possible because the domain of statements is restricted. By contrast, the follow-up work on competition programs [16] uses a larger control-flow graph that additionally includes interrupting edges, indicating to where control would flow from each node if an exception were raised during the execution of that node. Further, a sequence of tokens is associated with each node in the control-flow graph, following Section 3.2.4, allowing it to handle arbitrary human-authored Python statements.

**Global Relational Models of Source Code.** Hellendoorn et al. [62] investigates models that combine global information (like a Transformer) and structural information (like a GNN), i.e. Graph-Sandwich models and the GREAT (Graph-Relational Embedding Attention Transformer) model. This paper uses `python_graphs` to construct the composite program graphs of Section 3.1.3. The models accept these program graphs as input and uses them to identify variable misuse bugs.

## 4.2. Program Graphs as Targets

**Graph Finite-State Automaton (GFSA) Layers.** Johnson et al. [70] introduces a neural network layer that adds a new learned edge type to an input graph. Toward learning static analyses of Python code, it trains a neural model to take an AST as input and predict a composite program graph as output. The model thereby learns to perform both control-flow and data-flow analyses from data. For its targets, it produces a composite program graph with `python_graphs`, selecting a subset of the default edge types and introducing a few additional edge types (as in Section 3.2.1).

**Learning to Extend Program Graphs to Work-in-Progress Code.** Li et al. [92] learns to predict edge relations from work-in-progress code, even when the code does not parse. The composite program graphs of Section 3.1.3 form the ground truth edge relation targets for this work.

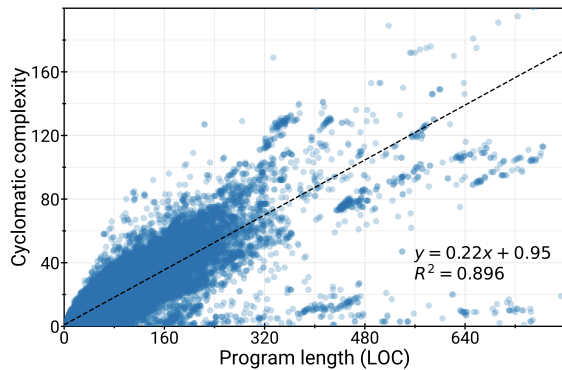
## 5. Case Study: Project CodeNet

In order to evaluate the `python_graphs` library on the diversity of language features found in realistic code, we obtain a dataset of 3.3 million programs from Project CodeNet [119]. For each program, we use `python_graphs` to construct a control-flow graph and a composite program graph complete with syntactic, control-flow, data-flow, and lexical information about the program. We collect metrics about the resulting graphs to provide information about the robustness of `python_graphs` and the size, complexity, and connectedness of the program graphs it produces.

STATUS	# PROGRAMS	FREQ. (%)
Success	3,157,463	96.08
ast.parse failures	126,751	3.86
SyntaxError	114,817	3.49
IndentationError	8,893	0.27
TabError	3,032	0.09
RecursionError	5	0.00
ValueError	4	0.00
RuntimeError	2,100	0.06
return outside function	1,719	0.05
break outside loop	330	0.01
continue outside loop	51	0.00
<b>Total</b>	<b>3,286,314</b>	<b>100%</b>

**Table 3.** Control-flow graph construction success rates on a dataset of both valid and invalid Python submissions to competitive programming problems.

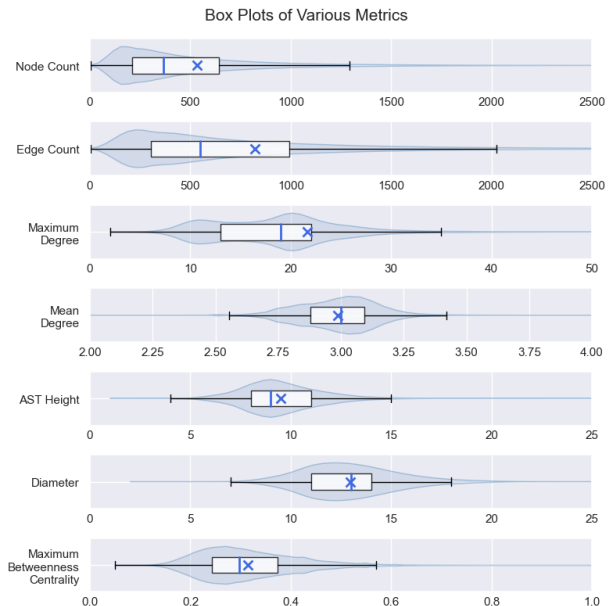
`python_graphs` cannot construct graph representations for every submission in Project CodeNet, as many of them do not parse. Table 3 shows how many of the programs graph construction succeeds for, as well as the failure reasons for the remaining graphs. The



**Fig. 4.** The relationship between program length and cyclomatic complexity for Python submissions in Project CodeNet.

EDGE TYPE	# / PROGRAM	FREQ. (%)
FIELD	370.4	100.00
SYNTAX	163.4	99.99
NEXT_SYNTAX	162.4	99.99
LAST_LEXICAL_USE	26.1	99.12
CFG_NEXT	18.1	99.06
LAST_READ	38.0	92.29
LAST_WRITE	30.6	98.83
COMPUTED_FROM	11.7	98.55
CALLS	0.5	21.37
FORMAL_ARG_NAME	0.6	12.99
RETURNS_TO	0.7	15.56

**Table 4.** Frequencies of edge types in the composite program graphs for the Project CodeNet Python submissions. # / Program is the average number of occurrences of the edge type across all programs. Freq. (%) shows the percent of programs exhibiting the edge type at least once.



**Fig. 5.** Box plots for various metrics of program graphs for Python submissions in Project CodeNet. The vertical blue line in each box-plot shows the median of the data as usual, while the blue  $\times$  shows the mean.

**Table 5.** Summary statistics for various graph metrics across the dataset. The diameter and maximum betweenness centrality metrics do not include graphs exceeding 5000 nodes.

METRIC	MIN	MEDIAN	MEAN	MAX
Node Count	3	364	534.8	751,817
Edge Count	2	548	822.4	4,675,600
Maximum Degree	2	19	21.6	150,004
Mean Degree	1.3	3.0	3.0	79.9
AST Height	1	9	9.5	269
Diameter	2	13	13.0	143
Maximum Betweenness Centrality	0.0	0.3	0.3	1.0

programs for which `python_graphs` cannot produce graph representations are predominantly those which fail to parse under Python’s own parser: `ast.parse`. The majority of such programs cause the parser to raise a `SyntaxError`, `IndentationError`, or `TabError`, with just nine leading the built in parser to raise a `RecursionError` or `ValueError`. The `python_graphs` library rejects an additional 2100 programs (0.07%) because they contain either `return` outside of a function body, or `break` or `continue` outside of a loop. In total, this result gives us confidence there are no language feature corner cases that elude the `python_graphs` library and cause failures for well-formed programs that otherwise can be run by a standard Python interpreter. In Table 4 we report for each program graph edge type the fraction of programs it appears in as well as the mean number of appearances across all programs.

We next use `python_graphs` to measure the cyclomatic complexity of each of the submissions. Figure 4 plots the relationship between program length and cyclomatic complexity. We measure program length in non-empty lines of code (LOC). Omitting as outliers those programs longer than 800 LOC or with complexity exceeding 200 (just 118 programs out of 3.16 million), we perform linear regression and observe  $R^2 = 0.896$ , in line with prior work [87, 139].

We measure the size of program graphs according to their node counts and edge counts, the height of their AST backbone, and graph diameter. As measures of connectedness, we compute the maximum degree of a node, mean degree of the nodes, and maximum betweenness centrality of a node in the graph. The distributions of each of these metrics are shown with boxplots in Figure 5, and key summary statistics are listed in Table 5. Appendix A.2 contains histograms showing the distribution of each metric across the dataset. These metrics convey the scale and diversity of submissions to online programming contests and the graph sizes needed for processing these submissions as `python_graphs` with graph neural networks.

## 6. Discussion

The core capabilities of `python_graphs` for machine learning research are generating control-flow graphs, performing data-flow analyses, generating composite program graphs, and computing cyclomatic complexity of Python programs. For our research, we have been fruitfully using `python_graphs` for graph representations of programs for multiple years. It serves an instrumental role in the research in Articles 2 and 4. The library is robust and flexible, having been successfully run on millions of programs and used in several published papers. Still, several open challenges remain for providing insights into program semantics to machine learners. First, due to the dynamic nature of Python the library’s analyses are limited to providing best-effort results, not considering the possible effects of e.g. dynamic execution or introspection. A further key limitation of the library is its restriction to processing Python programs. This makes getting a consistent graph representation across programming languages challenging, which is important when training a multi-lingual model of code. While significant recent progress has been made in machine learning for code research, many fundamental problems in the space remain open research challenges. Examples of these challenges include learning about program semantics from end-to-end program behavior, and identifying neural models exhibiting systematic generalization. For these challenges, where the structure and semantics of programs are important, `python_graphs` provides a framework to study how graph representations of programs may contribute to forward progress.



Article 2.

# Learning to Execute Programs with Instruction Pointer Attention Graph Neural Networks

by

David Bieber<sup>1</sup>, Charles Sutton<sup>1</sup>, Hugo Larochelle<sup>1</sup>, and Daniel Tarlow<sup>1</sup>

This article was published in NeurIPS 2020.

I present my contributions and the contributions of the coauthors.

I proposed early forms of the core ideas and iterated on them with my coauthors; I implemented the IPA-GNN model and all baseline models and the training and evaluation framework, as well as the data generation procedures; I performed and organized the full and partial program and systematic generalization experiments; I authored the article reproduced here.

This work was performed under the advising of Charles Sutton, Hugo Larochelle, and Daniel Tarlow, with Daniel Tarlow additionally contributing key elements of the core ideas including formalization of the task and the implementation of the shepherds method of graph batching.

**ABSTRACT.** Graph neural networks (GNNs) have emerged as a powerful tool for learning software engineering tasks including code completion, bug finding, and program repair. They benefit from leveraging program structure like control flow graphs, but they are not well-suited to tasks like program execution that require far more sequential reasoning steps than number of GNN propagation steps. Recurrent neural networks (RNNs), on the other hand, are well-suited to long sequential chains of reasoning, but they do not naturally incorporate program structure and generally perform worse on the above tasks. Our aim is to achieve the best of both worlds, and we do so by introducing a novel GNN architecture, the Instruction Pointer Attention Graph Neural Network (IPA-GNN), which achieves improved systematic generalization on the task of learning to execute programs using control flow graphs. The model arises by considering RNNs operating on program traces with branch decisions as latent variables. The IPA-GNN can be seen either as a continuous relaxation of the RNN model or as a GNN variant more tailored to execution. To test the models, we propose evaluating systematic generalization on learning to execute using control flow graphs, which tests sequential reasoning and use of program structure. More practically, we evaluate these models on the task of learning to execute partial programs, as might arise if using the model as a heuristic function in program synthesis. Results show that the IPA-GNN outperforms a variety of RNN and GNN baselines on both tasks.

**Keywords:** learning to execute, recurrent neural networks, graph neural networks



# 1. Introduction

Static analysis methods underpin thousands of programming tools from compilers and debuggers to IDE extensions, offering productivity boosts to software engineers at every stage of development. Recently machine learning has broadened the capabilities of static analysis, offering progress on challenging problems like code completion [22], bug finding [4, 62], and program repair [138]. Graph neural networks in particular have emerged as a powerful tool for these tasks due to their suitability for learning from program structures such as parse trees, control flow graphs, and data flow graphs.

These successes motivate further study of neural network models for static analysis tasks. However, existing techniques are not well suited for tasks that involve reasoning about program execution. Recurrent neural networks are well suited for sequential reasoning, but provide no mechanism for learning about complex program structures. Graph neural networks generally leverage local program structure to complete static analysis tasks. For tasks requiring reasoning about program execution, we expect the best models will come from a study of both RNN and GNN architectures. We design a novel machine learning architecture, the Instruction Pointer Attention Graph Neural Network (IPA-GNN), to share a causal structure with an interpreter, and find it exhibits close relationships with both RNN and GNN models.

To evaluate this model, we select two tasks that require reasoning about program execution: full and partial program execution. These “learning to execute” tasks are a natural choice for this evaluation, and capture the challenges of reasoning about program execution in a static analysis setting. Full program execution is a canonical task used for measuring the expressiveness and learnability of RNNs [159], and the partial program execution task aligns with the requirements of a heuristic function for programming by example. Both tasks require the model produce the output of a program, without actually running the program. In full program execution the model has access to the full program, whereas in partial program execution some of the program has been masked out. These tasks directly measure a model’s capacity for reasoning about program execution.

In our study of neural networks exhibiting systematicity, we evaluate our models for systematic generalization to out-of-distribution programs, on both the full and partial program execution tasks. In the program understanding domain, systematic generalization is particularly important as people write programs to do things that have not been done before. Evaluating systematic generalization provides a strict test that models are not only learning to produce the results for in-distribution programs, but also that they are getting the correct result because they have learned something meaningful about the language semantics. Models that exhibit systematic generalization are additionally more likely to perform well in a real-world setting.

We evaluate against a variety of RNN and GNN baselines, and find the IPA-GNN outperforms baseline models on both tasks. Observing its attention mechanism, we find it has learned to produce discrete branch decisions much of the time, and in fact has learned to execute by taking short-cuts, using fewer steps to execute programs than used by the ground truth trace.

We summarize our contributions as follows:

- We introduce the task of learning to execute assuming access only to information available for static analysis (Section 3).
- We show how an RNN trace model with latent branch decisions is a special case of a GNN (Section 4).
- We introduce the novel IPA-GNN model, guided by the principle of matching the causal structure of a classical interpreter.
- We show this outperforms other RNN and GNN baselines. (Section 5).
- We illustrate how these models are well-suited to learning non-standard notions of execution, like executing with limited computational budget or learning to execute programs with holes.

## 2. Background and Related Work

Static analysis is the process of analyzing a program without executing it [1]. One common static analysis method is control flow analysis, which produces a control flow graph [6]. Control flow analysis operates on the parse tree of a program, which can be computed for any syntactically correct source file. The resulting control flow graph contains information about all possible paths of execution through a program. We use a statement-level control flow graph, where nodes represent individual statements in the source program. Directed edges between nodes represent possible sequences of execution of statements. An example program and its control flow graph are shown in Figure 6.

When executing a program, an interpreter’s *instruction pointer* indicates the next instruction to execute. At each step, the instruction pointer corresponds to a single node in the statement-level control flow graph. After the execution of each instruction, the instruction pointer advances to the next statement. When the control flow graph indicates two possible next statements, we call this a branch decision, and the value of the condition of the current statement determines which statement the instruction pointer will advance to.

Recurrent neural networks have long been recognized as well-suited for sequence processing [135]. Another model family commonly employed in machine learning for static analysis is graph neural networks [125], which are particularly well suited for learning from program structures [4]. GNNs have been applied a variety of program representations and static analysis tasks [4, 40, 62, 126, 134, 138, 151]. Like our approach, the graph attention network

$n$	SOURCE	TOKENIZATION ( $x_n$ )	CONTROL FLOW GRAPH ( $n \rightarrow n'$ )	$N_{in}(n)$	$N_{out}(n)$
0	v0 = 23	0 = v0 23		$\emptyset$	{1}
1	v1 = 6	0 = v1 6		{0}	{2}
2	while v1 > 0:	0 while > v1 0		{1, 7}	{3, 8}
3	v1 -= 1	1 -= v1 1		{2}	{4}
4	if v0 % 10 <= 3:	1 if <= % v0 3		{3}	{5}
5	v0 += 4	2 += v0 4		{4}	{6}
6	v0 *= 6	2 *= v0 6		{5}	{7}
7	v0 -= 1	1 -= v0 1		{4, 6}	{2}
8	<exit>	- - - -	{2, 8}	{8}	

**Fig. 6. Program representation.** Each line of a program is represented by a 4-tuple tokenization containing that line’s (indentation level, operation, variable, operand), and is associated with a node in the program’s statement-level control flow graph.

(GAT) architecture [144] is a message passing GNN using attention across edges. It is extended by R-GAT [25] to support distinct edge types.

The task of learning to execute was introduced by [159], who applied RNNs. We are aware of very little work on learning to execute that applies architectures that are specific to algorithms. In contrast, *neural algorithm induction* [27, 47, 54, 55, 56, 71, 74, 123] introduces architectures that are specially suited to learning algorithms from large numbers of input-output examples. The modeling principles of this work inspire our approach. The learning to execute problem is different because it includes source code as input, and the goal is to learn the semantics of the programming language.

Systematic generalization measures a model’s ability to generalize systematically to out-of-distribution data, and has been an area of recent interest [13]. Systematic generalization and model design go hand-in-hand. Motivated by this insight, our architectures for learning to execute are based on the structure of an interpreter, with the aim of improving systematic generalization.

### 3. Task

Motivated by the real-world setting of static analysis, we introduce two variants of the “learning to execute” task: full and partial program execution.

#### 3.1. Learning to Execute as Static Analysis

In the setting of machine learning for static analysis, models may access the textual source of a program, and may additionally access the parse tree of the program and any common static analysis results, such as a program’s control flow graph. However, models may not access a compiler or interpreter for the source language. Similarly, models may not access dependencies, a test suite, or other artifacts not readily available for static analysis of a single

file. Prior work applying machine learning for program analysis also operates under these restrictions [4, 62, 126, 132, 151, 159]. We introduce two static analysis tasks both requiring reasoning about a program’s execution statically: full and partial program execution.

In full program execution, the model receives a full program as input and must determine some semantic property of the program, such as the program’s output. Full program execution is a canonical task that requires many sequential steps of reasoning. The challenge of full program execution in the static analysis setting is to determine the target without actually running the program. In previous work, full program execution has been used as a litmus test for recurrent neural networks, to evaluate their expressiveness and learnability [159]. It was found that long short-term memory (LSTM) RNNs can execute some simple programs with limited control flow and  $\mathcal{O}(N)$  computational complexity. In a similar manner, we are employing this variant of learning to execute as an analogous test of the capabilities of graph neural networks.

The partial program execution task is similar to the full program execution task, except part of the program has been masked. It closely aligns with the requirements of designing a heuristic function for programming by example. In some programming by example methods, a heuristic function informs the search for satisfying programs by assigning a value to each intermediate partial program as to whether it will be useful in the search [75]. A model performing well on partial program execution can be used to construct such a heuristic function.

We consider the setting of bounded execution, restricting the model to use fewer steps than are required by the ground truth trace. This forces the model to learn short-cuts in order to predict the result in the allotted steps. We select the number of steps allowed such that each loop body may be executed at least twice. In our experiments, we use bounded execution by default. However, we compare in Section 5 to a Trace RNN model that follows the ground truth control flow, and surprisingly we find that the bounded execution IPA-GNN achieves better performance on certain length programs.

In both task settings we train the models on programs with limited complexity and test on more complex programs to evaluate the models for systematic generalization. This provides a quantitative indication of whether the models are not only getting the problem right, but getting it right for the right reasons – because they’ve learned something meaningful about the language semantics. We expect models that exhibit systematic generalization will be better suited for making predictions in real world codebases, particularly when new code may be added at any time. Another perhaps less appreciated reason to focus on systematic generalization in learning to execute tasks is that the execution traces of real-world programs are very long, on the order of thousands or even millions of steps. Training GNNs on such programs is challenging from an engineering perspective (memory use, time) and a training dynamics perspective (e.g., vanishing gradients). These problems are significantly mitigated if

we can strongly generalize from small to large examples (e.g., in our experiments we evaluate GNNs that use well over 100 propagation steps even though we only ever trained with 10s of steps, usually 16 or fewer).

### 3.2. Formal Specification and Evaluation Metrics

We describe both tasks with a single formalization. We are given a complexity function  $c(x)$  and dataset  $D$  consisting of pairs  $(x, y)$ ;  $x$  denotes a program, and  $y$  denotes some semantic property of the program, such as the program’s output. The dataset is drawn from an underlying distribution of programs  $D$ , and is partitioned according to the complexity of the programs.  $D_{\text{train}}$  consists only of examples  $(x, y)$  satisfying  $c(x) \leq C$ , the threshold complexity, while  $D_{\text{test}}$  consists of those examples satisfying  $c(x) > C$ . For each program  $x$ , both the textual representation and the control flow graph are known.  $x_n$  denotes a statement comprising the program  $x$ , with  $x_0$  denoting the start statement.  $N_{\text{in}}(n)$  denotes the set of statements that can immediately precede  $x_n$  according to the control flow graph, while  $N_{\text{out}}(n)$  denotes the set of statements that can immediately follow  $x_n$ . Since branch decisions are binary,  $|N_{\text{out}}(n)| \leq 2 \forall n$ , while  $|N_{\text{in}}(n)|$  may be larger. We define  $N_{\text{all}}(n)$  as the full set of neighboring statements to  $x_n$ :  $N_{\text{all}}(n) = N_{\text{in}}(n) \cup N_{\text{out}}(n)$ . The rest of the task setup follows the standard supervised learning formulation; in this paper we consider only instances of this task with categorical targets, though the formulation is more general. We measure performance according to the empirical accuracy on the test set of programs.

## 4. Approach

In this section we consider models that share a causal structure with a classical interpreter. This leads us to the design of the Instruction Pointer Attention Graph Neural Network (IPA-GNN) model, which we find takes the form of a message passing graph neural network. We hypothesize that by designing this architecture to share a causal structure with a classical interpreter, it will improve at systematic generalization over baseline models.

### 4.1. Instruction Pointer RNN Models

When a classical interpreter executes a straight-line program  $x$ , it exhibits a simple causal structure. At each step of interpretation, the interpreter maintains a state consisting of the values of all variables in the program, and an instruction pointer indicating the next statement to execute. When a statement is executed, the internal state of the interpreter is updated accordingly, with the instruction pointer advancing to the next statement in sequence.

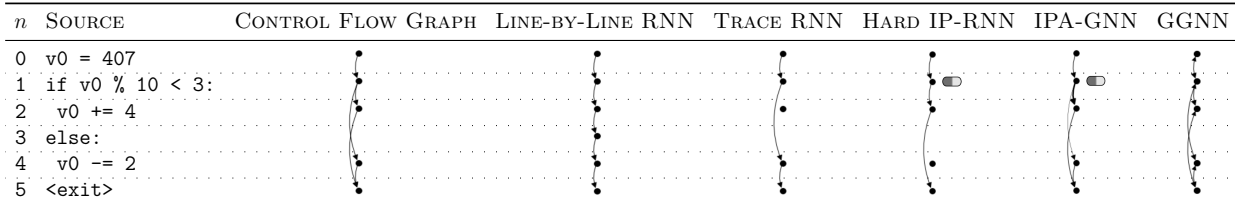
A natural neural architecture for modeling the execution of straight-line code is a recurrent neural network because it shares this same causal structure. At step  $t$  of interpretation, the model processes statement  $x_{t-1}$  to determine its hidden state  $h_t \in \mathbb{R}^H$ , which is analogous to

the values of the variables maintained in an interpreter’s state. In this model, which we call the Line-by-Line RNN model, the hidden state is updated as

$$h_t = \text{RNN}(h_{t-1}, \text{Embed}(x_{n_{t-1}})), \quad (4.1)$$

where  $n_t = t$  is the model’s instruction pointer.

Each of the models we introduce in this section has the form of (4.1), but with different definitions for the instruction pointer  $n_t$ . We refer to these models as Instruction Pointer RNNs (IP-RNNs).



**Fig. 7. Model paths comparison.** The edges in these graphs represent a possible set of paths along which information may propagate in each model. The pills indicate the positions where a model makes a learned branch decision. The Hard IP-RNN’s branch decision is discrete (and in this example, incorrect), whereas in the IPA-GNN the branch decision is continuous.

Consider next when program  $x$  includes control flow statements. When a classical interpreter executes a control flow statement (e.g. an if-statement or while-loop), it makes a branch decision by evaluating the condition. The branch decision determines the edge in the program’s control flow graph to follow for the new value of the instruction pointer. At each step of execution, values of variables may change, a branch decision may be made, and the instruction pointer is updated.

To match this causal structure when  $x$  has control flow, we introduce latent branch decisions to our model family. The result is an RNN that processes some path through the program, with the path determined by these branch decisions. The simplest model is an oracle which has access to the ground-truth trace, a sequence  $(n_0^*, n_1^*, \dots)$  of the statement numbers generated by executing the program. Then, if the instruction pointer is chosen as  $n_t = n_t^*$ , the resulting model is an RNN over the ground truth trace of the program. We refer to this model as the Trace RNN.

If instead we model the branch decisions with a dense layer applied to the RNN’s hidden state, then the instruction pointer is updated as

$$n_t = N_{\text{out}}(n_{t-1})|_j \quad \text{where } j = \text{argmax Dense}(h_t). \quad (4.2)$$

This dense layer has two output units to predict which of the two branches is taken. We call this the Hard IP-RNN model (in contrast with the soft instruction pointer based models in Section 4.2). It is a natural model that respects the causal structure of a classical interpreter,

but is not differentiable. A fully differentiable continuous relaxation will serve as our main model, the IPA-GNN. The information flows of each of these models, as well as that of a gated graph neural network, are shown in Figure 7.

## 4.2. Instruction Pointer Attention Graph Neural Network

To allow for fully differentiable end-to-end training, we introduce the Instruction Pointer Attention Graph Neural Network (IPA-GNN) as a continuous relaxation of the Hard IP-RNN. Rather than discrete branch decisions, the IPA-GNN makes soft branch decisions; a soft branch decision is a distribution over the possible branches from a particular statement. To accommodate soft branch decisions, we replace the instruction pointer of the Instruction Pointer RNN models ( $n_t$ ) with a soft instruction pointer  $p_{t,n}$ , which at step  $t$  is a distribution over all statements  $x_n$ . For each statement  $x_n$  in the support of  $p_{t,:}$ , we might want the model to have a different representation of the program state. So, we use a different hidden state for each time step and statement. That is, the hidden state  $h_{t,n} \in \mathbb{R}^H$  represents the state of the program, assuming that it is executing statement  $n$  at time  $t$ .

As with the IP-RNN models, the IPA-GNN emulates executing a statement with an RNN over the statement’s representation. This produces a state proposal  $a_{t,n}^{(1)}$  for each possible current statement  $n$

$$a_{t,n}^{(1)} = \text{RNN}(h_{t-1,n}, \text{Embed}(x_n)). \quad (4.3)$$

When executing straight-line code ( $|N_{\text{out}}(x_n) = 1|, n \rightarrow n'$ ) we could simply have  $h_{t,n'} = a_{t,n}^{(1)}$ , but in general we cannot directly use the state proposals as the new hidden state for the next statement, because there are sometimes multiple possible next statements ( $|N_{\text{out}}(x_n) = 2|$ ). Instead, the model computes a soft branch decision over the possible next statements and divides the state proposal  $a_{t,n}^{(1)}$  among the hidden states of the next statements according to this decision. When  $|N_{\text{out}}(x_n)| = 1$ , the soft branch decision  $b_{t,n,n'} = 1$ , and when  $|N_{\text{out}}(x_n)| = 2$ , write  $N_{\text{out}}(x_n) = \{n_1, n_2\}$  and

$$b_{t,n,n_1}, b_{t,n,n_2} = \text{softmax}(\text{Dense}(a_{t,n}^{(1)})). \quad (4.4)$$

All other values of  $b_{t,n,:}$  are 0. The soft branch decisions determine how much of the state proposals  $a_{t,n}^{(1)}$  flow to each next statement according to

$$h_{t,n} = \sum_{n' \in N_{\text{in}}(n)} p_{t-1,n'} \cdot b_{t,n',n} \cdot a_{t,n'}^{(1)}. \quad (4.5)$$

A statement contributes its state proposal to its successors in an amount proportional both to the probability assigned to itself by the soft instruction pointer, and to the probability given to its successor by the branch decision. The soft branch decisions also control how much probability mass flows to each next statement in the soft instruction pointer according

to

$$p_{t,n} = \sum_{n' \in N_{\text{in}}(n)} p_{t-1,n'} \cdot b_{t,n',n}. \quad (4.6)$$

The execution, branch, and aggregation steps of the procedure are illustrated in Figure 8.

By following the principle of respecting the causal structure of a classical interpreter, we have designed the IPA-GNN model. We hypothesize that by leveraging this causal structure, our model will exhibit better systematic generalization performance than models not respecting this structure.

### 4.3. Relationship with IP-RNNs

As a continuous relaxation of the Hard IP-RNN model, the IPA-GNN bears a close relationship with the Instruction Pointer RNN models. Under certain conditions, the IPA-GNN is equivalent to the Hard IP-RNN model, and under still further conditions it is equivalent to the Trace RNN and Line-by-Line RNN models. Specifically:

- If the IPA-GNN’s soft branch decisions saturate, the IPA-GNN and Hard IP-RNN are equivalent.
- If the Hard IP-RNN makes correct branch decisions, then it is equivalent to the Trace RNN.
- If the program  $x$  is straight-line code, then the Trace RNN is equivalent to the Line-by-Line RNN.

To show the first two we can express the Hard IP-RNN and Trace RNN models in terms of two dimensional state  $h_{t,n}$  and soft instruction pointer  $p_{t,n}$ . As before, let  $N_{\text{out}}(x_n) = \{n_1, n_2\}$ . Using this notation, the Hard IP-RNN’s branch decisions are given by

$$b_{t,n,n_1}, b_{t,n,n_2} = \text{hardmax} \left( \text{Dense}(a_{t,n}^{(1)}) \right), \quad (4.7)$$

where  $\text{hardmax}$  projects a vector  $v$  to a one-hot vector, i.e.,  $(\text{hardmax } v)|_j = 1$  if  $j = \text{argmax } v$  and 0 otherwise. The Trace RNN’s branch decisions in this notation are given by

$$b_{t,n,n'} = \mathbb{1}\{n_{t-1}^* = n \wedge n_t^* = n'\}. \quad (4.8)$$

Otherwise the model definitions are the same as for the IPA-GNN. The final assertion follows from observing that straight-line code satisfies  $n_t^* = t$ .

These connections reinforce that the IPA-GNN is a natural relaxation of the IP-RNN models, and that it captures the causal structure of the interpreter it is based on. These connections also suggest there is a conceivable set of weights the model could learn that would exhibit strong generalization as a classical interpreter does if used in an unbounded execution setting.



**Table 6.** The IPA-GNN model is a message passing GNN. Selectively replacing its components with those of the GGNN yields two baseline models, NoControl and NoExecute. Blue expressions originate with the IPA-GNN, and orange expressions with the GGNN.

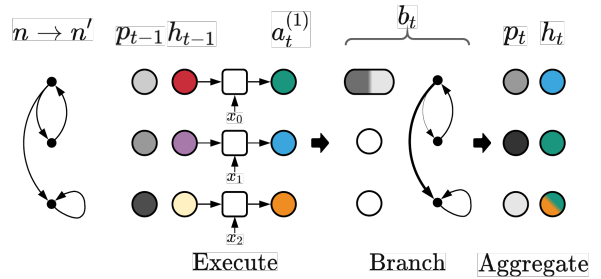
	IPA-GNN (OURS)	NoCONTROL	NoEXECUTE	GGNN
$h_{0,n}$	$= 0$	$= 0$	$= \text{Embed}(x_n)$	$= \text{Embed}(x_n)$
$a_{t,n}^{(1)}$	$= \text{RNN}(h_{t-1,n}, \text{Embed}(x_n))$	$= \text{RNN}(h_{t-1,n}, \text{Embed}(x_n))$	$= h_{t-1,n}$	$= h_{t-1,n}$
$a_{t,n,n'}^{(2)}$	$= p_{t-1,n'} \cdot b_{t,n',n} \cdot a_{t,n}^{(1)}$	$= 1 \cdot a_{t,n}^{(1)}$	$= p_{t-1,n'} \cdot b_{t,n',n} \cdot \text{Dense}(a_{t,n}^{(1)})$	$= 1 \cdot \text{Dense}(a_{t,n}^{(1)})$
$\tilde{h}_{t,n}$	$= \sum_{n' \in N_{\text{in}}(n)} a_{t,n,n'}^{(2)}$	$= \sum_{n' \in N_{\text{in}}(n)} a_{t,n,n'}^{(2)}$	$= \sum_{n' \in N_{\text{in}}(n)} a_{t,n,n'}^{(2)}$	$= \sum_{n' \in N_{\text{in}}(n)} a_{t,n,n'}^{(2)}$
$h_{t,n}$	$= \tilde{h}_t$	$= \tilde{h}_t$	$= \text{GRU}(h_{t-1,n}, \tilde{h}_{t,n})$	$= \text{GRU}(h_{t-1,n}, \tilde{h}_{t,n})$

#### 4.4. Relationship with GNNs

Though the IPA-GNN is designed as a continuous relaxation of a natural recurrent model, it is in fact a member of the family of message passing GNN architectures. To illustrate this, we select the gated graph neural network (GGNN) architecture [94] as a representative architecture from this family. Table 6 illustrates the shared computational structure held by GGNN models and the IPA-GNN.

The GGNN model operates on the bidirectional form of the control flow graph. There are four possible edge types in this graph, indicating if an edge is a true or false branch and if it is a forward or reverse edge. The learned weights of the dense layer in the GGNN architecture vary by edge type.

The comparison highlights two changes that differentiate the IPA-GNN from the GGNN architecture. In the IPA-GNN, a per-node RNN over a statement is analogous to a classical interpreter’s execution of a single line of code, while the soft instruction pointer attention and aggregation mechanism is analogous to the control flow in a classical interpreter. We can replace either the control flow components or the execution components of the IPA-GNN model with the expressions serving the equivalent function in the GGNN. If we replace both the control flow structures and the program execution components in the IPA-GNN, the result is precisely the GGNN model. Performing just the control flow changes introduces the No-



**Fig. 8.** A single IPA-GNN layer. At each step of execution of the IPA-GNN, an RNN over the embedded source at each line of code produces state proposals  $a_{t,n}^{(1)}$ . Distinct state values are shown in distinct colors. A two output unit dense layer produces branch decisions  $b_{t,n,:}$ , shown as pill lightness and edge width. These are used to aggregate the soft instruction pointer and state proposals to produce the new soft instruction pointer and hidden states  $p_{t,n}$  and  $h_{t,n}$ .

Control model, while performing just the execution changes introduces the NoExecute model.

We evaluate these intermediate models as baselines to better understand the source of the performance of the IPA-GNN relative to GNN models.

## 5. Experiments

Through a series of experiments on generated programs, we evaluate the IPA-GNN and baseline models for systematic generalization on the program execution as static analysis tasks.

### 5.1. Dataset

We draw our dataset from a probabilistic grammar over programs using a subset of the Python programming language. The generated programs exhibit variable assignments, multi-digit arithmetic, while loops, and if-else statements. The variable names are restricted to `v0`,  $\dots$ , `v9`, and the size of constants and scope of statements and conditions used in a program are limited. Complex nesting of control flow structures is permitted though. See the supplementary material for details.

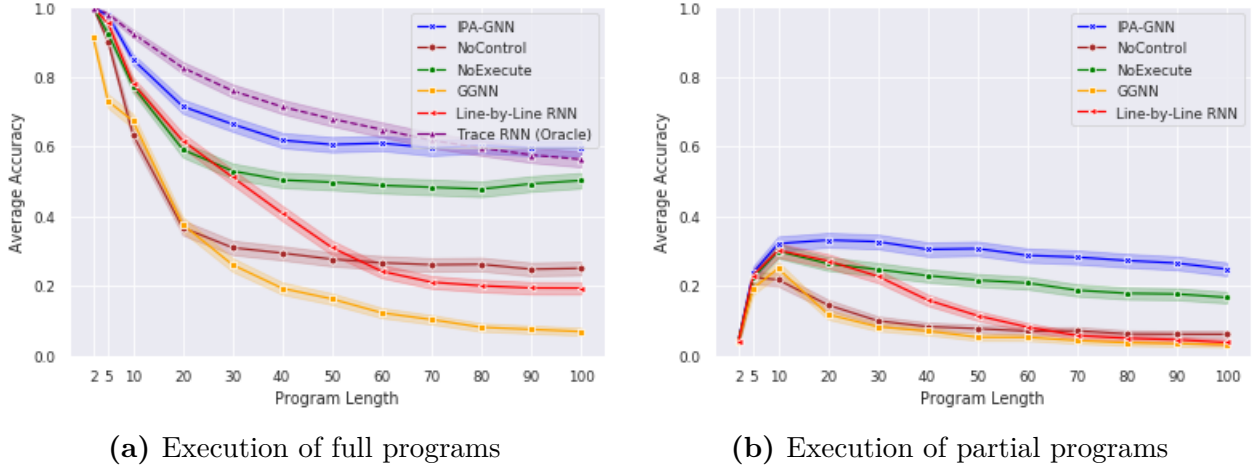
From this grammar we sample 5M examples with complexity  $c(x) \leq C$  to comprise  $D_{\text{train}}$ . For this filtering, we use program length as our complexity measure  $c$ , with complexity threshold  $C = 10$ . We then sample 4.5k additional samples with  $c(x) > C$  to comprise  $D_{\text{test}}$ , filtering to achieve 500 samples each at complexities  $\{20, 30, \dots, 100\}$ . An example program is shown in Figure 6.

For each complete program, we additionally construct partial programs by masking one expression statement, selected uniformly at random from the non-control flow statements. The target output remains the result of “correct” execution behavior of the original complete program.

In both the full program execution and partial program execution tasks, we let the target  $y$  be the final value of `v0` mod 1000. We select this target output to reduce the axes along which we are measuring generalization; we are interested in generalization to more complex program traces, which we elect to study independently of the complexity introduced by more complex data types and higher precision values. The orthogonal direction of generalization to new numerical values is studied in [132, 140]. We leave the study of both forms of generalization together to future work.

### 5.2. Evaluation Criteria

On both tasks we evaluate the IPA-GNN against the Line-by-Line RNN baseline, R-GAT baseline, and the NoControl, NoExecute, and GGNN baseline models. On the full program execution task, we additionally compare against the Trace RNN model, noting that this



**Fig. 9.** Accuracy of models as a function of program length on the program execution tasks. Spread shows one standard error of accuracy.

model requires access to a trace oracle. Following Zaremba and Sutskever [159], we use a two-layer LSTM as the underlying RNN cell for the RNN and IPA-GNN models.

We evaluate these models for systematic generalization. We measure accuracy on the test split  $D_{\text{test}}$  both overall and as a function of complexity (program length). Recall that every example in  $D_{\text{test}}$  has greater program length than any example seen at training time.

### 5.3. Training

We train the models for three epochs using the Adam optimizer [82] and a standard cross-entropy loss using a dense output layer and a batch size of 32. We perform a sweep, varying the hidden dimension  $H \in \{200, 300\}$  and learning rate  $l \in \{0.003, 0.001, 0.0003, 0.0001\}$  of the model and training procedure. For the R-GAT baseline, we apply additional hyperparameter tuning, yet we were unable to train an R-GAT model to competitive performance with the other models. For each model class, we select the best model parameters using accuracy on a withheld set of examples from the training split each with complexity precisely  $C$ .

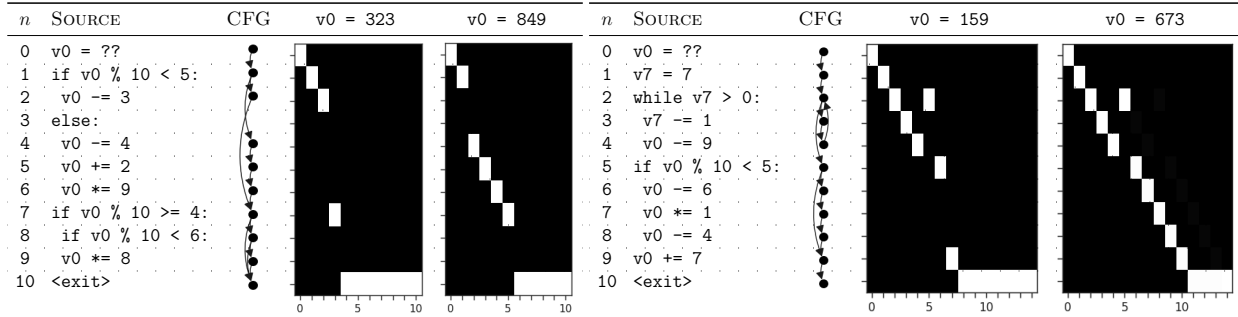
### 5.4. Program Execution Results

Table 7 shows the results of each model on the full and partial execution tasks. On both tasks, IPA-GNN outperforms all baselines.

Figure 9 breaks the results out by complexity. At low complexity values used during training, the Line-by-Line RNN model performs almost as well as the IPA-GNN. As complexity increases, however,

**Table 7.** Accuracies on  $D_{\text{test}}$  (%)

MODEL	FULL	PARTIAL
Trace RNN (Oracle)	66.4	—
Line-by-Line RNN	32.0	11.5
NoControl	28.1	8.1
NoExecute	50.7	20.7
GGNN	16.0	5.7
IPA-GNN (Ours)	<b>62.1</b>	<b>29.1</b>



**Fig. 10. Instruction Pointer Attention.** Intensity plots show the soft instruction pointer  $p_{t,n}$  at each step of the IPA-GNN on two programs, each with two distinct initial values for  $v0$ .

the performance of all baseline models drops off faster than that of the IPA-GNN. Despite using the ground truth control flow, the Trace RNN does not perform as well as the IPA-GNN at all program lengths.

Examining the results of the ablation models, NoExecute significantly outperforms NoControl, indicating the importance of instruction pointer attention for the IPA-GNN model.

Figure 10 shows the values of the soft instruction pointer  $p_{t,n}$  over the course of four predictions. The IPA-GNN learns to frequently produce discrete branch decisions. The model also learns to short-circuit execution in order to produce the correct answer while taking fewer steps than the ground truth trace. From the first program, we observe the model has learned to attend only to the path through the program relevant to the program’s result. From the second program, we see the model attends to the while-loop body only once, where the ground truth trace would visit the loop body seven times. Despite learning in a setting of bounded execution, the model learned a short-circuited notion of execution that exhibits greater systematic generalization than any baseline model.

## 6. Conclusion and Future Work

Following a principled approach, we designed the Instruction Pointer Attention Graph Neural Network architecture based on a classical interpreter. Importantly, however, the IPA-GNN learns much of the Python language semantics entirely from data, rather than having those semantics baked into the architecture; the IPA-GNN is a learned interpreter. By closely following the causal structure of an interpreter, the IPA-GNN exhibits stronger systematic generalization than baseline models on tasks requiring reasoning about program execution behavior. The IPA-GNN outperformed all baseline models on both the full and partial program execution tasks.

These tasks, however, only capture a subset of the Python programming language. The programs in our experiments were limited in the number of variables considered, in the

magnitude of the values used, and in the scope of statements permitted. Even at this modest level of difficulty, though, existing models struggled with the tasks, and thus there remains work to be done to solve harder versions of these tasks and to scale these results to real world problems. Fortunately the domain naturally admits scaling of difficulty and so provides a good playground for studying systematic generalization.

We believe our results suggest a promising direction for models that solve real world tasks like programming by example. We proffer that models like the IPA-GNN may prove useful for constructing embeddings of source code that capture information about a program’s semantics.

## 7. Broader Impact

Our work introduces a novel neural network architecture better suited for program understanding tasks related to program executions. Lessons learned from this architecture will contribute to improved machine learning for program understanding and generation. We hope the broader impact of these improvements will be improved tools for software developers for the analysis and authoring of new source code. Machine learning for static analysis produces results with uncertainty, however. There is risk that these techniques will be incorporated into tools in a way that conveys greater certainty than is appropriate, and could lead to either developer errors or mistrust of the tools.



## Article 3.

# Learning Semantic Representations to Verify Hardware Designs

by

Shobha Vasudevan<sup>1</sup>, Wenjie Jiang<sup>1</sup>, David Bieber<sup>1</sup>, Rishabh Singh<sup>1</sup>, Hamid Shojaei<sup>1</sup>, C. Richard Ho<sup>1</sup>, and Charles Sutton<sup>1</sup>

This article was published in NeurIPS 2021.

I present my contributions and the contributions of the coauthors.

I designed and implemented the register transfer level instruction pointer attention graph neural network architecture (RTL IPA-GNN). I augmented the Verilog control and data flow graphs with features suitable for use with the RTL IPA-GNN. I trained and evaluated the neural models on this data for the cover point coverage experiments on each of IBEX v1, IBEX v2, and TPU, iterating on our findings. I coauthored the article reproduced here.

Shobha Vasudevan, Hamid Shojaei, and Richard Ho bring to this collaboration their chip design and hardware verification expertise, while Wenjie Jiang, Charles Sutton, Rishabh Singh, and myself focus on the machine learning aspects of the project. Wenjie and Shobha led the effort and together with Hamid and Richard they managed coordination with the design verification engineers. Wenjie, Richard, and Hamid performed the initial dataset construction, and Wenjie set up the first graph neural network implementation and the train eval pipeline.

**ABSTRACT.** Verification is a serious bottleneck in the industrial hardware design cycle, routinely requiring person-years of effort. Practical verification relies on a “best effort” process that simulates the design on test inputs. This suggests a new research question: Can this simulation data be exploited to learn a continuous representation of a hardware design that allows us to predict its functionality? As a first approach to this new problem, we introduce Design2Vec, a deep architecture that learns semantic abstractions of hardware designs. The key idea is to work at a higher level of abstraction than the gate or the bit level, namely the Register Transfer Level (RTL), which is similar to software source code, and can be represented by a graph that incorporates control and data flow. This allows us to learn representations of RTL syntax and semantics using a graph neural network. We apply these representations to several tasks within verification, including predicting what cover points of the design will be covered (simulated) by a test, and generating new tests to cover desired cover points. We evaluate Design2Vec on three real-world hardware designs, including the TPU, Google’s industrial chip used in commercial data centers. Our results demonstrate that Design2Vec dramatically outperforms baseline approaches that do not incorporate the RTL semantics and scales to industrial designs. It generates tests that cover design points that are considered hard to cover with manually written tests by design verification experts in a fraction of the time.

**Keywords:** hardware verification, test coverage, graph neural networks



# 1. Introduction

Hardware designs are verified to check if a design implements the architectural specification correctly. Verification is widely considered the most serious bottleneck in the contemporary industrial hardware design cycle [44, 45, 46]. It requires 60-75% of the time, compute, and human resources during the design phase, routinely taking multiple person-years of effort. Verification is a complex problem because modern hardware has billions of inputs and flops, and the number of states is exponential in the number of flops and inputs. Checking every state of the system is infeasible due to a combinatorial state space explosion. While there is excellent research in automatic hardware verification techniques involving formal and static analysis, their applicability in practice is limited by scale. Instead, practical design verification sacrifices automation and completeness for a “best effort, risk reducing” process based on simulating the design on test inputs.<sup>3</sup>

Traditionally, formal verification methods analyze synchronous hardware designs through reachability analysis of a gate level state transition graph. Each node in this graph corresponds to a single value of bit-level state of all registers, and edges correspond to the legal changes in state that the design can make in a single clock cycle. It is well known in the formal methods community that most questions about hardware functionality can be posed as reachability questions in the state transition graph. This suggests a fundamental new research question in representation learning: *Can we learn a continuous representation of a hardware design that allows us to predict its functionality?*

This paper presents a first approach to this problem. Since designs are practically verified by simulations using millions of test inputs, this gives us a ready source of training data. To avoid the combinatorial explosion at the gate level, we approach the problem at a higher level of abstraction, the Register Transfer Level (RTL) that describes hardware at the bit-vector or integer level. RTL is described in a Hardware Description Language (Verilog RTL [145]) that is syntactically similar to the source code of software, while modeling the concurrent, non-deterministic, non-terminating, and reactive semantics of hardware. Despite the higher level of abstraction, RTL static analysis approaches for reachability analysis [10] and test input generation [103] do not scale to even reasonably sized practical designs. So our research question can be restated as: *Can we use simulation data to learn a tractable continuous representation that can predict the answers to the state reachability queries in hardware?*

In this paper, we introduce Design2Vec, an architecture for learning continuous representations that capture the semantics of a hardware design. We choose to represent the design as a control data flow graph (CDFG) at the RTL level. Based on the CDFG, we use graph neural networks (GNN) to learn representations that predict states reached by the design

---

<sup>3</sup>All future references to verification will imply simulation based design verification as the majority practice in industry, not formal verification.

when simulated on test inputs. While standard GNN architectures do work well, we achieve improved performance by introducing a new propagation layer that specifically incorporates the concurrent and non-terminating semantics of RTL. Design2Vec is trained to predict if, given a simulation test input, a particular branch (like a case statement in software) will be covered. The CDFG is an abstraction of the gate-level state space since one edge in the CDFG maps to many edges in the gate-level transition graph. We can therefore interpret Design2Vec as learning an *abstraction* of the intractable gate-level design state space.

We apply Design2Vec to two practical problems in hardware verification: coverage prediction and test generation. The coverage prediction model can act as a proxy simulator, and an engineer can query it to estimate coverage in seconds, instead of waiting for a night of simulations. Our test generation method uses a gradient-based search over the trained Design2Vec model to generate new tests. It is desirable to find tests for *hard to cover* points, or points human experts find difficult to cover after reaching around 80% (or more) cumulative coverage using random testing. This coverage plateau can take months to close, effectively taking multiple expert months of productivity and effort.

We demonstrate Design2Vec is able to successfully learn representations of multiple designs: (i) IBEX v1 [67], a RISC-V design (ii) IBEX v2, IBEX enhanced with security features, and (iii) Tensor Processing Unit (TPU) [72], Google’s industrial scale commercial infrastructure ML accelerator chip, and performs dramatically better than black-box baselines that are uninformed by knowledge of the design (up to 50% better, and on average 20% better for real designs). Our results show that Design2Vec achieves over 90% accuracy in coverage prediction on the industrial TPU design, making it ideal to serve as a proxy simulator that can evaluate if a test can cover a given point within seconds. Our results on test generation show that Design2Vec is able to successfully find tests for hard to cover points in real designs in a fraction of the time (up to 88 fewer simulator calls for TPU and 40 fewer for IBEX) as compared to random testing and a black-box optimizer. This order of magnitude improvement can potentially lead to huge savings. Our key contributions follow.

- We introduce the problem of learning continuous representations (abstractions) of hardware semantics that can be used for various tasks in hardware verification. We present Design2Vec, a model that encodes the syntactic and semantic structure of an RTL design using a GNN based architecture.
- We propose the RTL IPA-GNN, which extends a recent architecture for learning to execute software [15] to model the concurrent and non-terminating semantics of RTL.
- We introduce a test generation algorithm that uses Design2Vec to generate focused tests for a subset of cover points. The tests generated by Design2Vec are able to successfully cover hard to cover points using significantly fewer simulations than state of practice human expert guided random testing and a state of the art black-box optimization tool.

- We demonstrate the scalability of the approach to large industrial scale designs. Our approach is practical in industrial settings.

## 2. Design2Vec: Representation learning of design abstractions

Verilog RTL source code is a powerful tool for writing non-terminating, continuous, reactive “programs” to design hardware. A program in RTL is structured as a modular hierarchy, with multiple concurrent blocks executing within each module. We consider the subset of behavioral RTL that can be synthesized into gate level designs [145].

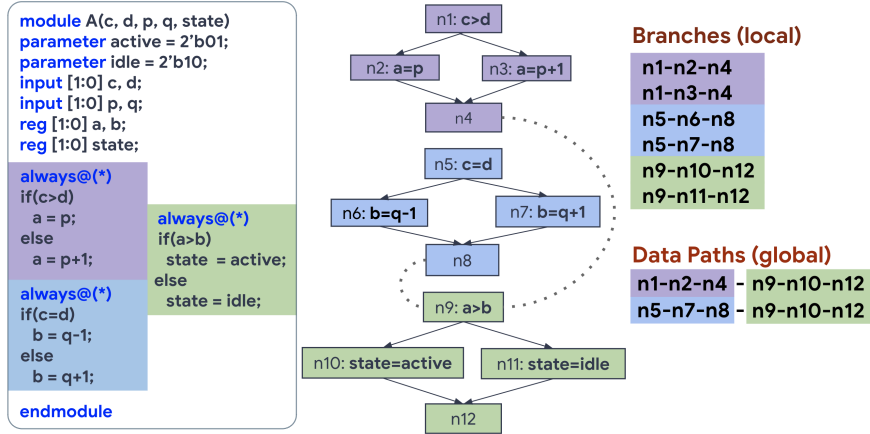
Figure 11 shows an example snippet of Verilog RTL source code. A “module” corresponds to a hardware construct (e.g. a decoder or ALU). A module can instantiate other modules (e.g. a cache module may instantiate a prefetch module). Each “variable” in the RTL corresponds to either an input signal (input), an output signal (output), a register that stores values (reg) or a temporary variable (wire). The bit width of each such variable is declared. In the example, `a`, `b`, and `state` are two-bit registers, `c`, `d`, `p`, and `q` are two-bit input signals.

This design has three concurrent (always) blocks denoted by three colors. For synchronous hardware, an external clock signal triggers execution of each concurrent block per *clock cycle*. On each clock cycle, one statement in each always block is executed. The statements in an always block are thereby executed in sequence over multiple consecutive clock cycles. When executing a statement, the input values of variables in the current cycle come from the output values of variables from the previous cycle. There is an implicit loop from the end of each block back to its start, simulating the non-terminating nature of hardware. Within a cycle, the order in which the always blocks are executed is *non-deterministic*. In practice, it is as determined by an RTL simulator. Figure 29 in Appendix C.1 shows the execution (simulation) over a three cycle window of this example.

RTL designs are typically simulated using an RTL design simulator, which does not suffice for the design goals of our analysis. For use with machine learning, we choose to represent RTL as a control data flow graph (CDFG). The CDFG we construct for each Verilog RTL program encodes the program’s simulation semantics, so that our models may make inferences about the their behavior.

The CDFG has nodes and heterogeneous edges. Nodes correspond to statements in code. Edges corresponds to either data flow or control flow. Dotted lines show data dependencies between concurrently executing blocks. The root node corresponds to the `begin` node of the top module. Branches denote localized node sequences. The designs that we consider contain branches, but not loops, which are atypical in synthesizable RTL.

A *test* is a set of high level parameters, each of which can be Boolean, integer or categorical. A test defines a distribution over input signals to the design. When a test is run, the testbench



**Fig. 11.** An example snippet of a Verilog RTL source code module and the corresponding CDFG.

samples many inputs from this distribution, and simulates the design on that sequence of inputs. The output of the testbench is Boolean input vectors applied to the RTL design under test. These input vectors execute different design paths. All the branches executed along different design paths are said to be *covered*.

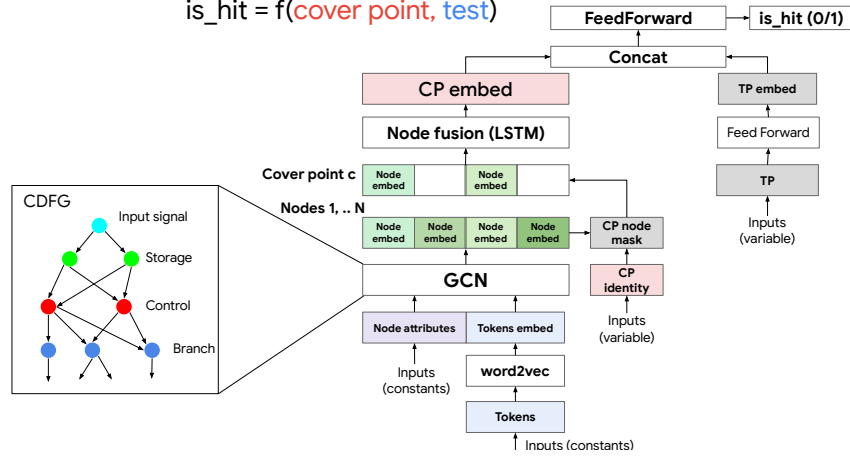
A test is written to achieve different types of design code coverage. Branch coverage tends to be the most important metric. Branch coverage of a test refers to branches executed by that test. Each branch is referred to as a *cover point*. A cover point over the RTL CDFG is identified as a sequence of nodes starting from the branch control node and ending at the destination of the branch. Two local branch cover points lie along the same global path if they have ancestor/descendant relationship. Figure 30 in Appendix C.1 shows the executed branches the input arrives at a certain clock cycle.

## 2.1. Architecture

Our Design2Vec architecture, shown in Figure 12, is trained on the supervised task of coverage prediction. The network takes as input a cover point  $C$ , represented as a sequence of CDFG notes, and a test parameter vector  $I$ , and outputs the probability  $\text{is\_hit}(C, I)$  that the test  $I$  covers  $C$ .

**Cover point embedding:** We use a graph convolution network (GCN) over the Verilog CDFG. Each node  $n \in N$  has attributes/features denoted by  $f_j^n$  such as node identifier, node type, fan in and fan out. In addition, the RTL token sequence  $s^n = \langle s_1^n, s_2^n, \dots, s_k^n \rangle$  in the program statement is also considered as an additional attribute. We first perform an embedding of the token sequence  $\phi(s^n)$  using an LSTM (or alternatively using a pre-trained word2vec module):

$$\phi(s^n) = \text{LSTM}(\langle s_1^n, s_2^n, \dots, s_k^n \rangle). \quad (2.1)$$



**Fig. 12.** The Design2Vec architecture that takes as input a cover point and an input test vector, and predicts the corresponding coverage. CDFG: Control and data flow graph. TP: test parameter. CP: cover point.

The initial embedding of a node  $\phi^{(0)}(n)$  is computed as concatenation of the node attribute values and the token sequence embedding:

$$\phi^{(0)}(n) = \text{Concat}(\langle \phi(s^n), f_1^n, \dots, f_i^n \rangle). \quad (2.2)$$

Let the embedding of a graph  $G$  at a step  $t$  be denoted by  $\psi^{(t)}(G) = \{\phi^{(t)}(n) | \forall n \in N\}$ . We use a GCN (with learnable parameters  $\theta$ ) to perform a sequence of graph convolution steps to update the node embeddings:

$$\psi^{(t+1)}(G) = f_{\theta}(\psi^{(t)}(G)) \quad \forall t \in \{0, 1, \dots, T\}. \quad (2.3)$$

A cover point  $C = \langle n_1, n_2, \dots, n_m \rangle$  is a sequence of nodes. The cover point identity is used to select the corresponding sequence of node embeddings of the cover point nodes. Since these sequences are of varying length, an LSTM layer is used to produce the cover point embeddings

$$\phi^{(T)}(C) = \text{LSTM}(\langle \phi^{(T)}(n_1), \phi^{(T)}(n_2), \dots, \phi^{(T)}(n_m) \rangle). \quad (2.4)$$

**Test parameter embedding:** For the input test parameters  $I$ , each of which can be integers or categorical, we learn an embedding for the parameters by passing them through a feed forward MLP layer as

$$\phi(I) = \text{MLP}(\text{Concat}(i_1, \dots, i_p)). \quad (2.5)$$

**MLP layer:** Finally, the test parameter  $\phi(I)$  and cover point embeddings  $\phi^{(T)}(C)$  are concatenated and provided to a feed forward layer with sigmoid activation to predict for each cover point

$$\text{is\_hit}(C, I) = \text{MLP}_{\sigma}(\text{Concat}(\phi^{(T)}(C), \phi(I))). \quad (2.6)$$

Given a supervised dataset of test inputs and the corresponding coverage information of different cover points in the RTL design, the network is trained using binary cross entropy loss. We find that our model learns more from local paths than global paths (root to cover

point). A reason for this is that GNNs are not proficient at learning long path information. In the case of a control flow, tracing and learning global paths is relevant. We use shortcut edges, where we add edges along every  $k$ th node along a path, to reinforce the relationship of long paths (results in Appendix D). We use a variation of the GNN, called the GNN-MLP [21] to propagate edge information. We also used gating layers and residual layers in the GGNN.

## 2.2. RTL IPA-GNN architecture

We enhance the recently proposed Instruction Pointer Attention Graph Neural Network (IPA-GNN) architecture [15] that explicitly models control flow in programs. This section motivates and precisely describes each of the architectural enhancements our novel RTL IPA-GNN architecture makes over the original IPA-GNN in order to model the concurrent, non-terminating semantics of RTL.

First, since RTL hardware is highly parallel, the RTL IPA-GNN maintains a separate instruction pointer for each always block in the hardware design. The original includes only a single instruction pointer. Making this change allows modeling the concurrent execution of all modules in an RTL specification. Accordingly, the RTL IPA-GNN instantiates its soft instruction pointer  $p_{t,n}$  as

$$p_{0,n} = \mathbf{1}_{\text{is an always block start node}},$$

and our RTL IPA-GNN implementation computes its hidden state proposals as

$$a_{t,n}^{(1)} = \text{Dense}(h_{t-1,n}).$$

Second, the domain of RTL requires the model support switch conditions, not just binary conditions. So, we modify the soft branch decision mechanism of the IPA-GNN. In the RTL IPA-GNN, the soft branch decisions at timestep  $t$  are given as

$$b_{t,n,m} = \text{softmax}(\text{Dense}(h_{t-1,n}) \cdot \text{Embed}(e_{n,m})),$$

where  $m \in N_{\text{out}}(x_n)$  is a control node child of  $x_n$ .

$\text{Embed}(e_{n,m})$  is an embedding for the control edge from  $x_n$  to  $x_m$ . It includes an embedding of whether the condition is positive or negative, the first variable referenced by the condition, and the form of the condition. It is computed as a concatenation of a learned embedding of each of these three properties of the condition represented by the edge.

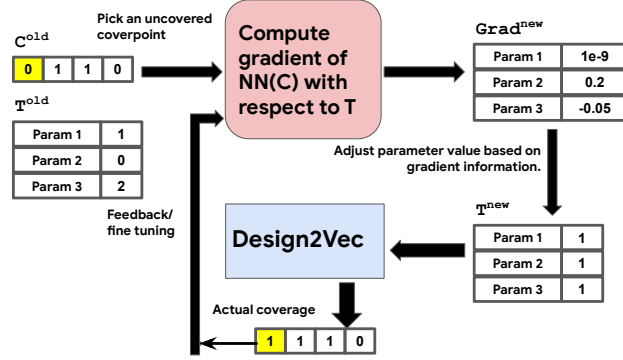
Third, data flow in an RTL design is also more complex than in a single-threaded program. We model data flow in the RTL IPA-GNN architecture by propagating messages between nodes along data flow edges at each step of the model. This entails aggregating hidden state proposals both from control node state proposals (these are within a node’s always block) and from the proposals of other parent nodes (which may be from a different always block),

**Fig. 13.** Test Input Generation

**Input:** A set of uncovered points  $\mathcal{C} = \{C_1, \dots, C_m\}$

- 1:  $\mathcal{I} = \{\}$
- 2: **while**  $\mathcal{C} \neq \emptyset$  **do**
- 3:      $C = \text{PickRandomCoverPoint}(\mathcal{C})$
- 4:      $\mathcal{C} = \mathcal{C} \setminus C$
- 5:     **for**  $j = 1 \dots K$  **do**
- 6:         ▷ Optimize cover prob. wrt test parameters  $I$
- 7:          $I \leftarrow \text{random}()$
- 8:         **while**  $I$  has not converged **do**
- 9:              $I \leftarrow I + \nabla_{I \text{ is\_hit}}(C, I)$
- 10:          $\mathcal{I} \leftarrow \mathcal{I}.\text{append}(I)$

**return**  $\mathcal{I}$



**Fig. 14.** The overall workflow to use Design2Vec in generative loop to generate input test parameters.

giving

$$h'_{t,n} = \sum_{n' \in N_{\text{in}}(n) \cap N_{\text{ctrl}}} p_{t-1,n'} \cdot b_{t,n',n} \cdot a_{t,n'}^{(1)} + \sum_{n' \in N_{\text{in}}(n) \cap \bar{N}_{\text{ctrl}}} a_{t,n'}^{(1)}.$$

Here  $N_{\text{ctrl}} \subseteq N$  denotes the set of control nodes in the CDFG.

Finally, we introduce a control edge from the end of each always block to the start of the always block to model non-termination, a critical RTL property. With this update to the CDFG, the instruction pointer probability mass in the RTL IPA-GNN returns to the start of each always block after reaching its conclusion. Incorporating RTL semantics into our modeling decisions improves coverage prediction performance in some settings as compared with domain independent GNNs.

### 2.3. Gradient-based search for test generation

We now present an algorithm to generate tests to cover different cover points in an RTL design using a trained Design2Vec model. Figure 14 shows the overall flow. The key idea of Algorithm 13 is to perform a gradient based search to maximize the Design2Vec predicted probability of covering desired cover points. The algorithm takes as input a set  $\mathcal{C}$  of uncovered points for which we would like to generate tests. For each uncovered point  $C \in \mathcal{C}$ , the algorithm first selects a random test  $I$ , a vector of test parameters. The algorithm then computes the objective function as the predicted probability of covering the input cover point, and computes gradients of the objective function with respect to each parameter of  $I$ . The test parameters in  $I$  are updated using a gradient ascent method until  $I$  converges (Line 9 in Algorithm 13). This process is repeated from different random initializations of  $I$  to get a list of  $K$  test parameters. Finally, the tests with highest predicted coverage are run through an RTL simulator to get the actual coverage.

### 3. Impact of Design2Vec solutions in practical verification

In this section, we detail the verification flow, and highlight the value proposition brought by the Design2Vec solutions to this problem. Figure 31 in the Appendix A.2 shows the industrial verification flow. The current approach for verifying RTL designs is constrained random verification [28]. This involves a complex program called the testbench. The input to the testbench is a set of test parameters written by verification engineers that define the distribution over inputs that will be applied to the RTL design. Some inputs need to be constrained, while others will be randomized by the testbench to widely sample the input space. The testbench output is a Boolean level stimulus that is applied to the primary inputs of the RTL design. Each such high level “test” corresponds to millions of cycles of Boolean input stimuli that run through an overnight regression. Bug reports and coverage are assessed at the end of a nightly regression. This process is iterated upon until there are 0 bugs and 100% of the design cover points are covered. Simulator calls are costly and need to be minimized. Effectively, this entire process costs multiple engineer years of productivity and resources.

Figure 32 illustrates how Design2Vec would integrate in the loop. We select two tasks for Design2Vec to provide practical value in constrained random verification: (i) **Coverage prediction**, or predicting which cover points in the design would be covered by a given test. Such a predictive model can serve as a **proxy simulator**, and an engineer can query it to estimate coverage instantaneously, instead of waiting for a night of simulation. (ii) **Test parameter generation**, or automating tests towards faster coverage closure, especially for covering **hard-to-cover points**.<sup>4</sup> in the verification cycle.

### 4. Coverage prediction experiments

We first evaluate the Design2Vec architecture on the task of coverage prediction. We evaluate two key research questions: (i) whether Design2Vec is able to exploit information from the RTL hardware design to improve predictions and (ii) whether coverage prediction is good enough for Design2Vec to serve as a proxy simulator.

We evaluate our methods on the three real-world designs from Table 17. For each design, we obtain training data by generating random tests and sampling each test parameter uniformly. For each test, we use the testbench to randomly sample input test stimulus, and a Verilog RTL simulator to obtain ground-truth labels of whether a cover point is covered by that test or not, in the form of a coverage vector. We sample 1696 tests for IBEX v1, 1938 for IBEX v2, and 1995 for TPU. Some cover points are very easy to predict, e.g., they are

---

<sup>4</sup>In this paper we use a tool that is better than human guided random testing as a baseline for hard to cover points.



covered by almost 0% or nearly 100% of the random tests. To avoid evaluating our models on these trivial cases, we only include cover points in the data if they are covered by between 10% and 90% of the random tests. After this filtering, there are 160 cover points in the data set for the IBEX v1 design, 177 for IBEX v2, and 3781 for TPU.

Furthermore, there is a particular type of generalization that is vital to applying coverage prediction in practice. Recall that when a design engineer uses a coverage prediction model, it is because they are proposing a new test that should exercise a cover point that is not exercised by the existing test. Therefore, in practice the primary concern is how well the model predicts whether a test that it has never seen during training will exercise a cover point *that the model has never seen how to cover during training*. For this reason, we divide the data into training and test point by cover point, such that no cover point and no test occurs in both the training and validation set for the learned models.<sup>5</sup> All results are the median over three random train-validation splits.

We ask two key questions in our experiments: 1) does representation learning allow Design2Vec to generalize from cover points in the training set to related cover points (e.g. neighbors in CDFG) and 2) do the graph neural networks provide deeper learning of CDFG graph structure and cover point relationships than more shallow representations?

Table 8 compares Design2Vec to three baselines across all three designs, varying the size of the training set. One is the naive statistical frequency baseline of guessing the most common value; it is the average positive rate over all cover points in the validation data (or (1-average positive rate), whichever is larger). This baseline does not take into account any correlations between cover points or test inputs. The second baseline is a multilayer perceptron (MLP) that treats the design as a black box, and does not take into account any information from the RTL. The MLP takes as input the test parameters, represented the same as in Design2Vec, and the numeric index of the cover point, represented as a one-hot vector. While it cannot generalize across cover points, the MLP can still learn to generalize across test parameters (e.g., some test parameters activate many cover points, some few). In that sense it is stronger than statistical frequency.

The third baseline is node sequence embedding, a stronger baseline that enhances MLP by allowing it to generalize across cover points. Recall that every cover point is defined as a sequence of nodes down a control-flow path from the root of an always block to a particular node, e.g., **n1-n2-n4** in Figure 1. We use node sequences over all the training cover points in a Word2Vec model to learn embeddings for each control flow node, which is then concatenated and padded into a cover point embedding. This representation of the cover point is used instead of the one-hot representation in the MLP. Cover points that have structural proximity in the CDFG graph, e.g., **n1-n2-n4** and **n1-n3-n4** would have similar embeddings. This

---

<sup>5</sup>We avoid using the term “test set” to refer to the data on which we evaluate our machine learning methods, to avoid confusion with tests of the RTL design.

**Table 8.** Accuracy at coverage prediction for Design2Vec (with the RTL IPA-GNN layer) compared to a black-box multi-layer perceptron (MLP), which does not have information about the design.

TRAIN COVER POINTS	IBEX v1			IBEX v2			TPU		
	50%	80%	90%	50%	80%	90%	50%	80%	90%
DESIGN2VEC	<b>74.2</b>	<b>77.3</b>	<b>77.8</b>	<b>73.4</b>	<b>78.0</b>	<b>80.3</b>	<b>90.5</b>	<b>90.6</b>	<b>91.1</b>
NODE SEQ EMBEDDING	59.7	59.1	63.2	58.5	57.3	59.0	87.9	88.4	88.6
MLP	57.5	56.8	56.8	58.7	58.0	58.2	42.8	42.5	34.7
STATISTICAL FREQUENCY	50.5	51.6	50.8	54.1	54.5	54.7	68.5	68.6	68.6

baseline can generalize to new cover points if there is a nearby coverpoint in training. While it takes graph structure into account, it does not have the full flexibility of GNN-style message propagation. First two baselines answer question 1 while node sequence embedding helps answer 3.

In these results, Design2Vec uses the RTL IPA-GNN (subsection 2.2) as the GNN layer. Hyperparameters including number of layers, learning rates, and embedding dimensions are reported in Appendix C.5.

Notably, the MLP performs catastrophically poorly on the test data. It is slightly better than statistical frequency. Indeed, on the largest design, the industrial TPU design, Design2Vec has an accuracy of 47% higher *in absolute terms* than the MLP. We observe that the training accuracy of MLP is high (above 95%), so the MLP overfits and fails to generalize to cover points outside the training set. This is expected, since it has no information about which of the training cover points are most close to the cover points in validation set. The MLP prediction is based only on test parameter features. Node sequence embedding performs much better than the two blackbox models in every case, indicating that even shallow representation learning using RTL CDFG structure helps generalization.

The Design2Vec model wins by a substantial margin in every case, indicating that the GNN based architecture is able to learn deep relationships in the design and generalize effectively to majority of the cover points unseen during training.

This pattern holds across all three designs. On IBEX v1 and v2, the difference between the two models is smaller (although still around 15% absolute). This is due to the irregular structure of the Ibox designs, with fewer repeated hardware modules, causing the control flow path to be harder to predict. On the TPU industrial design is the largest and most complex of the three, Design2Vec has over 90% accuracy. The node sequence embedding also has high accuracy on this design. This can be attributed to the highly regular structure of TPU with a relatively easier to predict control flow.

For Design2Vec, we also compare several variants of graph neural networks: graph convolution networks [83], gated graph neural networks [94], GNN-MLP [21], and the RTL IPA-GNN (subsection 2.2). These results are shown in Table 9. Additionally, we vary the

**Table 9.** Coverage prediction accuracies of different GNN architectures within Design2Vec.

ARCHITECTURE	NODE FUSION	IBEX v1			IBEX v2			TPU		
		50%	80%	90%	50%	80%	90%	50%	80%	90%
GCN	LSTM	74.0	73.0	73.2	70.4	74.5	73.9	90.9	90.5	91.1
	MEAN	74.1	75.8	74.0	69.0	73.9	72.5	—	—	—
GGNN	LSTM	<b>75.0</b>	75.9	74.5	71.4	77.0	76.0	<b>91.1</b>	90.4	<b>91.2</b>
	MEAN	73.8	75.6	76.4	70.0	75.1	72.3	—	—	—
GNN-MLP	LSTM	73.8	76.8	<b>78.5</b>	70.7	74.5	71.2	91.0	90.1	91.1
	MEAN	73.0	76.4	74.4	70.1	75.1	72.0	—	—	—
RTL IPA-GNN	LSTM	74.2	<b>77.3</b>	77.8	<b>73.4</b>	<b>78.0</b>	<b>80.3</b>	90.5	<b>90.6</b>	91.1
	MEAN	73.6	77.2	76.3	72.2	75.8	78.9	—	—	—

method by which representations of CDFG nodes are aggregated to represent cover points. Recall from subsection 2.1 that to represent a cover point, we take the final node embeddings from the GNN, apply a mask to obtain only a few relevant CDFG, and then aggregate them. We compare using an LSTM, as in (2.4), to simply taking the mean. We compare this for different GNN architectures, and note that aggregation methods performed similarly on IBEX, so we evaluate only LSTM aggregation on TPU. Overall, all GNN variants performed similarly. RTL IPA-GNN performs similarly to the other GNN architectures on IBEX v1 and TPU, but is significantly better on IBEX v2. Since the RTL IPA-GNN performs similarly or better than other GNNs across the three designs, we use this as the main Design2Vec model in Table 8. We also tuned several other features of the architecture, including presence of the residual connections, embedding size, and label smoothing (details in Appendix B and C).

**Given that TPU is a typical ML accelerator chip whose architecture will have similar properties across generations, the high coverage prediction accuracy (>90%) shows the potential of Design2Vec to serve as a proxy simulator taking seconds instead of a night of simulations.**

## 5. Test generation using Design2Vec

In this section, we evaluate Design2Vec on the test generation task, especially for hard to cover points during coverage closure. In general, there are two RTL testing approaches, namely directed testing and random testing. Directed testing refers to targeted testing of specific functionality in RTL, whereas random testing refers to undirected, random perturbation of inputs with the goal of covering the design space maximally. Design2Vec based test generation is directed and should ideally be compared with a directed testing baseline. However, we did not find a comparable directed testing tool in the open source (or commercially).<sup>6</sup>

<sup>6</sup>At this time, there is no practical tool for automated directed testing in industry. Some directed testing tools have been proposed in literature but are not available in the public domain for comparison. Random testing tools in open source use different tool flows and RTL language, making a comparison infeasible.

We present two comparative studies between Design2Vec and Vizier SRP, a random testing tool [52] that uses black-box optimization to maximize total coverage all cover points with every test. This comparison is inherently unequal due to the difference in their objectives of the two tools. We compare both tools with respect to coverage achieved and number of simulator calls made by each tool to achieve that coverage. Given that simulator calls are expensive, this is a relevant metric to compare.

Vizier SRP functions as follows. In each trial, it generates a test for the total set of cover points in the design, and calls the simulator. It uses actual coverage feedback in an active learning loop along with Bayesian optimization to generate tests with progressively higher coverage.

In the first study, we find points that are *to hard to cover* for Vizier SRP and challenge Design2Vec to generate tests for those points. To find hard to cover points, we run Vizier SRP until around 80% cumulative coverage is reached. From Table 10, it is seen that the remaining uncovered points are indeed rare from cover probabilities on a randomly sampled dataset (not used for training).

We intercept Vizier SRP after running for a number of trials (200 for IBEX v1, 243 for TPU). We provide the same sample tests collected by Vizier SRP as training data for Design2Vec. For each point uncovered until that point by Vizier SRP, we provide them as (unseen) target points for the Design2Vec test generation algorithm described in Table 13.

Design2Vec generates multiple test recommendations, of which we select the top ranking recommendation and call the simulator for evaluating actual coverage. If the target cover point is uncovered, we generate a different test using Design2Vec and repeat the same procedure until it is covered. For this study, we run a small number of Design2Vec tests (25). Table 18 in Appendix A demonstrates some example parameterized tests generated by Design2Vec.

We continue to run Vizier SRP further (upto 400 trials) and record the coverage at every trial (between 201-400). If an uncovered point gets covered, we measure the number of simulator calls (trials) taken by Vizier SRP to cover it for the first time and compare with the number of simulator calls (tests) used by Design2Vec to cover it. Note that Vizier SRP uses active learning for the remaining trials until 400, while Design2Vec uses zero shot learning in this configuration.

After 200 trials of Vizier, there were 22 uncovered points in IBEX and 23 uncovered points in the TPU. Table 10 shows the results of this comparison. Both Vizier SRP and Design2Vec cover 3 cover points. Vizier SRP covers 2 cover points that Design2Vec does not. Design2Vec covers 1 cover point within 12 tests that Vizier SRP does not upto 400. Similarly, for 882, due to its low probability, Vizier SRP needs 24 tests to cover it, while Design2Vec uses only 3 tests to cover. Neither Design2Vec nor Vizier SRP cover 16 cover points. We expect Design2Vec to cover more if it is run beyond 25 tests. Table 11 shows similar results on the hard to cover points in the TPU design.

In the second study, we compare Design2Vec with Vizier as a test generation tool for overall coverage. We train the two tools independently with different datasets, closer to the practical use case. We evaluate the total coverage of Design2Vec as compared to Vizier for a sample of randomly selected points with varying cover probabilities (in the spectrum of always covered to rarely/never covered). We randomly select 10 cover points from each of the three buckets of cover probabilities and hid these 30 cover points from Design2Vec. We generate tests for each of the 30 cover points using individual cover points as the search objective and report the number of tests required for Design2Vec and Vizier to hit the cover point for the first time. Table 23 in Appendix C.7 shows the summarized observations over the 30 randomly selected points. Design2Vec is clearly valuable when generating tests hard to cover points.

In practice, the intended use case of Design2Vec is to complement Vizier SRP. While Vizier SRP can maximize for cumulative coverage, Design2Vec can generate tests for hard to cover points. Notably, hard to cover points take a lot of time and resources to cover in practice. **These results show the power of representation learning in Design2Vec. Despite the lack of examples covering hard to cover points, Design2Vec is able to generate a test to cover these cover points with orders of magnitude fewer simulator calls than a black-box optimizer and random testing.**

COVER POINT		NUMBER OF USED TESTS	
ID	PROB.	VIZIER	DESIGN2VEC
401	0.0012	Not covered	<b>29</b>
526	0.0059	Not covered	<b>12</b>
528	0.0035	Not covered	<b>10</b>
879	0.0071	42	<b>14</b>
882	0.0059	24	<b>3</b>
886	0.0018	93	<b>25</b>
664	0.01	55	Not covered
881	0.0053	108	Not covered
14 cover points	Not covered	Not covered	

**Table 10.** Comparison of Design2Vec and black-box optimizer tests for covering hard to cover points: IBEX v1. Number of tests are RTL simulations.

COVER POINT		NUMBER OF USED TESTS	
ID	PROB.	VIZIER	DESIGN2VEC
35793	0.0	22	<b>17</b>
36996	0.0	90	<b>2</b>
36372	0.0	78	<b>17</b>

**Table 11.** Comparison of Design2Vec and black-box optimization to hit target cover points: TPU. Number of tests are RTL simulations.

## 6. Related work

**Automated test generation for RTL:** Most previous techniques for RTL test generation generate Boolean level stimuli at the inputs of the RTL design [86, 103, 104, 124, 153].

**Table 12.** Comparing test generation of easy, medium and hard to cover points. Design2Vec is very efficient at generating tests for hard to cover points. Summarized result of table in Appendix C.7 Table 23.

Cover probabilities	Difficulty	Summary of comparison
[0.5,1.0)	Easy	Both Vizier and Design2Vec cover all points with a single test.
[0.2,0.5)	Medium	Vizier and Design2Vec cover 9 out of 10. Design2Vec takes 3 fewer tests on average.
[0.05, 0.2)	<b>Hard</b>	<b>Design2Vec takes 20 fewer tests than Vizier on average.</b> Up to 40 fewer in cases.

By virtue of operating at a higher abstraction level of parameterized tests, it is orders of magnitude more scalable than the Boolean input stimulus generation techniques. Static analysis based approaches for test generation [14, 59, 112] are based on traversing the state transition graph at a logic gate level, or the RTL design have inherent scalability limitations. Approaches that combine static and dynamic analysis like HYBRO [97] and concolic testing [103, 104, 153] in RTL rely on SMT/SAT solvers, which are also limited by scale. Other approaches random forests and decision trees with static analysis and formal verification [43, 98] require manual feature engineering and handcrafted algorithms.

**Neural program testing:** While our approach learns the semantics of a design, learning based fuzzing [51] does not take program semantics into account. In contrast to our approach that models semantics of an RTL design, Neuzz [129] approximates a program using only the input-output behavior of the program, which is similar to our black-box MLP baseline. GMETAEXP [34] models test generation as a reinforcement learning problem with the objective of maximizing total program coverage, where programs are represented using a GGNN.

**Learning to execute:** Among methods that model program semantics, [159] present an approach to use LSTM to embed a program as input and generate the output as the output sequence. IPA-GNN [15] represents the program execution using an RNN augmented with a differentiable mechanism to represent next instruction after a statement execution. Unlike these works that learn to generate program output, we tackle the problem of reachability, or learning to reach a state/node in a graph.

**Abstractions for verification:** Abstraction techniques have been used to scale design verification for many decades. Some techniques are property specific [33, 84], design specific [60], data abstractions (word level abstractions) [80, 157], language based (abstract interpretation) [35, 66], execution or structure based [42]. These abstractions are typically defined manually and are challenging to create in a precise way to both scale the verification as well as maintain desired precision. In contrast, GNN based abstractions are task-specific, in the form of continuous representations of the CDFG nodes, that generalizes both across tasks, as well as designs.

## 7. Conclusion

We present an approach that is able to learn representations of hardware designs that can predict their functionality. With Design2Vec, we demonstrate, for the first time, the ability of deep learning in creating abstractions of the hardware design state space. These abstractions outperform black-box baselines and human baselines (state of practice) in verification by orders of magnitude in performance as well as scale. They can also generalize across different designs. Since Design2Vec learns reachability over the RTL design graph, it can also potentially generalize to other verification tasks like debugging, property generation and root causing. The utility of the RTL IPA-GNN architecture in Design2Vec’s performance points to the value of designing architectures with systematicity. Through its construction, the architecture has a bias toward mimicking the simulation of the RTL programs it is trained on, often resulting in improvements on the learning task. More broadly, this work shows the power and potential of deep learning to make a significant leap in progress in the area of verification, which is considered a longstanding practical and scientific challenge in computing.





Article 4.

# Static Prediction of Runtime Errors by Learning to Execute Programs with External Resource Descriptions

by

David Bieber<sup>1</sup>, Rishab Goel<sup>1</sup>, Daniel Zheng<sup>1</sup>, Hugo Larochelle<sup>1</sup>, and Daniel Tarlow<sup>1</sup>

This article was published in ICLR 2023.

I present my contributions and the contributions of the coauthors.

I identified the problem setting and research need and prepared the dataset by implementing a pipeline to instrument and run each of the Python submissions. I implemented the models and organized and ran the experiments. I led the team discussions, coordinated the team's activities, and authored the paper reproduced here.

Rishab Goel implemented the token RNN baseline and tracked and performed neural network experiments. Daniel Zheng visualized programs and instruction pointers with intensity plots, and made extensive training, evaluation, debugging, and code review contributions. This work was done under the advising of Hugo Larochelle and Daniel Tarlow.

**ABSTRACT.** The execution behavior of a program often depends on external resources, such as program inputs or file contents, and so cannot be run in isolation. Nevertheless, software developers benefit from fast iteration loops where automated tools identify errors as early as possible, even before programs can be compiled and run. This presents an interesting machine learning challenge: can we predict runtime errors in a “static” setting, where program execution is not possible? Here, we introduce a real-world dataset and task for predicting runtime errors, which we show is difficult for generic models like Transformers. We approach this task by developing an interpreter-inspired architecture with an inductive bias towards mimicking program executions, which models exception handling and “learns to execute” descriptions of the contents of external resources. Surprisingly, we show that the model can also predict the location of the error, despite being trained only on labels indicating the presence/absence and kind of error. In total, we present a practical and difficult-yet-approachable challenge problem related to learning program execution and we demonstrate promising new capabilities of interpreter-inspired machine learning models for code.

**Keywords:** program analysis, runtime errors, interpreter-inspired architectures

# 1. Introduction

We investigate applying neural machine learning methods to the static analysis of source code for early prediction of runtime errors. The execution behavior of a program is in general not fully defined by its source code in isolation, because programs often rely on external resources like inputs, the contents of files, or the network. Nevertheless, software developers benefit from fast iteration loops where automated tools identify errors early, even when program execution is not yet an option. Therefore we consider the following machine learning challenge: can we predict runtime errors in a “static” setting, where program execution is not possible?

This runtime error prediction task is well suited as a challenge problem because it is difficult-yet-approachable, has real-world value for software developers, requires novel modeling considerations that we hypothesize will be applicable to a range of learning for code tasks, and with this work, now has a suitable large dataset of complex human-authored code with error labels. The task is to predict whether a program will exhibit a runtime error when it is run, and if so to determine the error; even when static analysis cannot provide guarantees of an error in the code, patterns learned from data may point to likely errors. Our dataset consists of 2.4 million Python 3 programs from Project CodeNet [119] written by competitive programmers. We have run all programs in a sandboxed environment on sample inputs to determine their error classes, finding the programs exhibit 26 distinct error classes including “no error”. Each program relies on an external resource, the stdin input stream, and we pair each program with a natural language description of the behavior of the stream. We make the task and dataset, along with all models considered in this work, available for the research community to facilitate reproduction of this work and further research<sup>7</sup>.

To make progress on this challenging task, we identify a promising class of models from prior work, interpreter-inspired models, and we demonstrate they perform well on the task. Instruction Pointer Attention Graph Neural Network (IPA-GNN) [15] models simulate the execution of a program, following its control flow structure, but operating in a continuous embedding space. We make a number of improvements to IPA-GNN: scaling up to handle complex programs requiring thousands of execution steps, adding the ability to “learn to execute” descriptions of external resources, and extending the architecture to model exception handling and recover error locations. We evaluate these interpreter-inspired architectures against Transformer, LSTM, and GGNN neural baselines, and against pylint as a static analysis baseline. Our combined improvements lead to increased accuracy in predicting runtime errors and to interpretability allowing for prediction of error locations even though the models are only trained on error presence and error class, not error location. In total, we summarize our contributions as:

---

<sup>7</sup><https://github.com/google-research/runtime-error-prediction>

- We introduce the runtime error prediction task and a large accompanying dataset, providing runtime error annotations for millions of competition Python programs.
- We demonstrate that IPA-GNN architectures are practical for the complexity of real programs by scaling them to handle competition programs, and there we find they outperform generic models.
- We demonstrate that external resource descriptions, such as Japanese or English descriptions of stdin, can be leveraged to improve performance on the task across all model architectures.
- We extend the IPA-GNN to model exception handling, resulting in the Exception IPA-GNN, which we find can localize errors even when only trained on error presence and kind, not error location.

## 2. Related Work

**Program analysis.** Program analysis is a rich family of techniques for detecting defects in programs, including static analyses which are performed without executing code [12, 100, 154] and dynamic analyses which are performed at runtime [26, 50, 127]. Linters and type checkers are popular error detection tools that use static analysis. Static analysis (e.g. symbolic execution) does not typically use concrete inputs, while dynamic analysis requires concrete inputs and program execution. Compared with traditional static analysis, our approach is more flexible in its input representation, using a general “resource description” abstraction, which can represent the entire spectrum from concrete inputs to input constraints to missing inputs.

**Execution-aware models.** Several neural architectures draw inspiration from program interpreters [15, 20, 37, 47, 54, 55, 74, 123]. Our work is most similar to Bieber et al. [15] and Bošnjak et al. [20], focusing on how interpreters handle control flow and exception handling, rather than on memory allocation and function call stacks. Other works use program execution data directly, training with textual representations of execution traces as inputs [109, 110, 114] or performing execution during synthesis [31, 95, 133]. Compared with these, our approach uses weaker supervision, using only runtime error labels for training.

**Fault detection and localization datasets.** There has been considerable recent interest in applying machine learning to identifying and localizing faults in source code [3]. Puri et al. [119] makes a large dataset of real world programs available, which we build on in constructing our runtime errors dataset. Our dataset (i) is large (it has millions of examples), (ii) exhibits many programming language features, (iii) is written by human authors, and (iv) has error labels from the execution behavior of programs. Previous code datasets only exhibit a subset of these properties: large real-world and competition code datasets [63, 65, 76, 95, 119, 122] exhibit properties i, ii, and iii, but not iv, while learning to execute datasets [15, 159] exhibit property iv but not i, ii, or iii. Recent program synthesis datasets [11, 29] exhibit ii and iii

**Table 13.** Distribution of target classes in the runtime errors dataset. † denotes the balanced test split.

TARGET CLASS	TRAIN #	VALID #	TEST #	TARGET CLASS	TRAIN #	VALID #	TEST #
No error	1881303	207162	205343 / 13289†	numpy.AxisError	20	2	3
AssertionError	47	4	8	OSError	19	2	2
AttributeError	10026	509	1674	OverflowError	62	6	11
EOFError	7676	727	797	re.error	5	0	0
FileNotFoundError	259	37	22	RecursionError	2	0	1
ImportError	7645	285	841	RuntimeError	24	5	3
IndentationError	10	0	12	StopIteration	3	0	1
IndexError	7505	965	733	SyntaxError	74	4	3
KeyError	362	39	22	TypeError	21414	2641	2603
MemoryError	8	7	1	UnboundLocalError	8585	991	833
ModuleNotFoundError	1876	186	110	ValueError	25087	3087	2828
NameError	21540	2427	2422	ZeroDivisionError	437	47	125
				Timeout	7816	1072	691
				Other	18	8	2

only. Other datasets obtain error labels by injecting synthetic errors [4, 78, 118] (lacking the realism of iii) or from commit messages [40, 73] (lacking i and iv).

**Fault localization approaches.** Fault localization approaches vary in (i) level of supervision – weak (error labels) [91] vs strong (explicit location labels) [5, 101, 162, 164] – and (ii) localization granularity – statement-level [5, 101, 162] vs method-level [91, 164]. Our approach uses weak supervision in the form of runtime error labels to indirectly learn fault localization at a statement-level.

### 3. Runtime Error Prediction

**Task.** The goal of the runtime error prediction task is to determine statically whether a program is liable to encounter a runtime error when run, and if so, what error kind. The programs cannot be executed directly, as they lack unit tests and depend on external resources which are not available. A textual description of the external resources, which may be the program’s inputs, a file’s contents, or network access, is provided. This makes reasoning about the execution behavior of the programs plausible, even though actually performing the execution is not. We treat this task as one-class classification, with each error type as its own class and with “no error” as an additional class.

**Dataset.** We construct the runtime errors dataset using Python submissions to competitive programming problems from Project CodeNet [119]. Beginning with the 3.28 million Python submissions in Project CodeNet, we filter the submissions to keep only those written in Python 3, which are syntactically valid, which do not make calls to user-defined functions, and which do not exceed a threshold length of 512 tokens once tokenized. By running each submission in a sandboxed environment, we identify its ground truth runtime error class. Each submission is associated with a competitive programming problem whose problem statement we parse to obtain a description in either English or Japanese of the inputs the

program is liable to receive at runtime. This process results in a dataset of 2.44 million submissions, each paired with one of 26 target classes. The “no error” target is most common, accounting for 93.4% of examples. For examples with one of the other 25 error classes, we additionally note the line number at which the error occurs.

We divide the problems into train, validation, and test splits at a ratio of 80:10:10. All submissions to the same problem become part of the same split. This reduces similarities between examples across splits that otherwise could arise from the presence of multiple similar submissions for the same problem. Since there is a strong class imbalance in favor of the no error class, we also produce a balanced version of the test split by sampling the no error examples such that they comprise roughly 50% of the test split. We use this balanced test split for all evaluations. We report the number of examples having each target class in each split in Table 13. We describe the full dataset generation and filtering process in greater detail in Appendix D.1, and we evaluate the limitations of the dataset quantitatively in Appendix D.2.

While there are many datasets in the literature that test understanding of different aspects of code including bug-finding, we believe ours fills a gap: it is large-scale (millions of examples), it has real-world implications and presents a practical opportunity for improvement using ML-based approaches, and it tests a combination of statistical reasoning and reasoning about program execution.

## 4. Approach: IPA-GNNs as Relaxations of Interpreters

We make three modifications to the Instruction Pointer Attention Graph Neural Network (IPA-GNN) architecture. These modifications scale the IPA-GNN to complex code, allow it to incorporate external resource descriptions into its learned executions, and add support for modeling exception handling. The IPA-GNN architecture is a continuous relaxation of the standard interpreter ( $I$ ) defined by the pseudocode in Algorithm 1, minus the **magenta** text. We frame these modifications in relation to specific lines of the algorithm: scaling the IPA-GNN to complex human-authored code (Section 4.1) and incorporating external resource descriptions (Section 4.2) both pertain to interpreting and executing statement  $x_p$  at Line 3, and modeling exception handling adds the **magenta** text at lines 4-6 to yield a new interpreter ( $I'$ ) (Section 4.3). We showcase the behavior of both interpreters  $I$  and  $I'$  on a sample program in Figure 15, and illustrate an execution of the same program by a continuous relaxation of interpreter  $I'$  ( $\tilde{I}'$ ) alongside it.

### 4.1. Extending the IPA-GNN to Real Programs

Bieber et al. [15] interprets the IPA-GNN architecture as a message passing graph neural network operating on the statement-level control-flow graph of the input program  $x$ . Each

---

**Algorithm 1** Interpreter for which the **Exception** IPA-GNN is a continuous relaxation
 

---

**Input:** Program  $x$ 

```

1:  $h \leftarrow \emptyset; p \leftarrow 0$  ▷ Initialize the interpreter.
2: while  $p \notin \{n_{\text{exit}}, n_{\text{error}}\}$  do
3:    $h \leftarrow \text{Evaluate}(x_p, h)$  ▷ Evaluate the current statement.
4:   if  $\text{Raises}(x_p, h)$  then
5:      $p \leftarrow \text{GetRaiseNode}(x_p, h)$  ▷ Raise exception.
6:   else
7:     if  $\text{Branches}(x_p, h)$  then
8:        $p \leftarrow \text{GetBranchNode}(x_p, h)$  ▷ Follow branch.
9:     else
10:       $p \leftarrow p + 1$  ▷ Proceed to next statement.

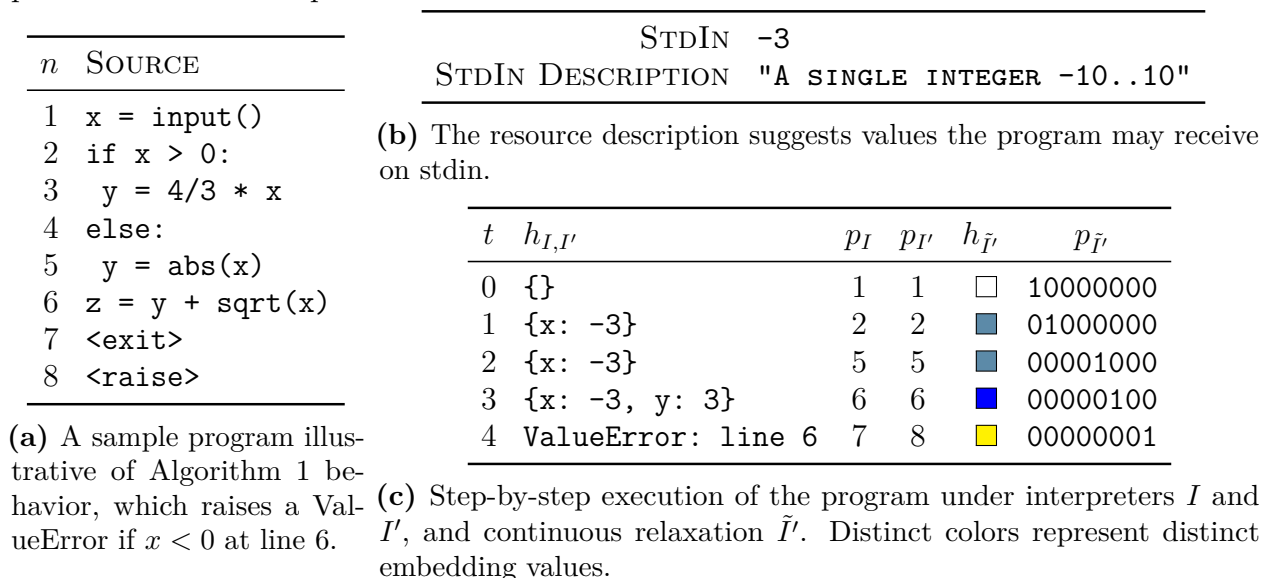
```

---

node in the graph corresponds to a single statement in the program. At each step  $t$  of the architecture, each node performs three steps: it executes the statement at that node (Line 3, Equation 4.2), computes a branch decision (Lines 7-8, Equation 4.4), and performs mean-field averaging over the resulting states and instruction pointers (Appendix D.3, Equations D.3 and D.4).

Unlike in Bieber et al. [15] where program statements are simple enough to be uniformly encoded as four-tuples, the programs in our runtime errors dataset consist of arbitrarily complex Python statements authored by real programmers in a competition setting. The

**Fig. 15.** A sample program and its execution under discrete interpreters  $I$  and  $I'$  (Algorithm 1) and under continuous relaxation  $\tilde{I}'$  of interpreter  $I'$ .  $p_{I_t}$  denotes the instruction pointer under  $I$  at step  $t$ .



language features used are numerous and varied, and so the statement lengths vary substantially, with a mean statement length of 6.7 tokens; we report the full distribution of statement lengths in Figure 34.

The IPA-GNN architecture operates on a program  $x$ 's statement-level control-flow graph, and so requires per-statement embeddings  $\text{Embed}(x_n)$  for each statement  $x_n$ . We first apply either a *local* or *global* Transformer encoder to produce per-token embeddings, and we subsequently apply one of four pooling variants to a span of such embeddings to produce a *node embedding* per statement in a program. In the local approach, we apply an attention mask to limit the embedding of a token in a statement to attending to other tokens in the same statement. In the global approach, no such attention mask is applied, and so every token may attend to every other token in the program. We consider four types of pooling in our hyperparameter search space: *first*, *sum*, *mean*, and *max*. The resulting embedding is given by

$$\text{Embed}(x_n) = \text{Pool}\left(\text{Transformer}(x)_{\text{Span}(x,n)}\right). \quad (4.1)$$

First pooling takes the embedding of the first token in the span of node  $n$ . Sum, mean, and max pooling apply their respective operations to the embeddings of all tokens in the span of node  $n$ .

Finally we find that the programs in our dataset require as many as 174 steps of the IPA-GNN under the model's heuristic for step limit  $T(x)$  (Appendix D.5). To reduce the memory requirements, we apply rematerialization at each layer of the model [30, 57].

## 4.2. Executing with Resource Descriptions

In our dataset, each program  $x$  may be accompanied by a description of what values `stdin` may contain at runtime. We convert this description into embedding  $d(x)$ ; the embeddings, vocabulary, and tokenizer used to produce  $d(x)$  are shared with those used to produce token embeddings from program source. Analogous to Line 1 of Algorithm 1, IPA-GNN architectures initialize with per-node hidden states  $h_{0,:} = 0$  and soft instruction pointer  $p_{0,n} = \mathbb{1}\{n = 0\}$ . Here  $p_{t,n}$  represents the probability under the model that node  $n$  is executing at step  $t$ . Following initialization, each step of an IPA-GNN begins by simulating execution (Line 3) of each non-terminal statement with non-zero probability under the soft instruction pointer to propose a new hidden state contribution

$$a_{t,n}^{(1)} = \text{RNN}(h_{t-1,n}, \text{Modulate}(\text{Embed}(x_n), d(x), h_{t-1,n})). \quad (4.2)$$

The text in **magenta** shows our modification to the IPA-GNN architecture to incorporate external resource descriptions. We consider both *Feature-wise Linear Modulation* (FiLM) [116] and *cross-attention* [88] for the Modulate function, which we define in Appendix D.4. Modulation allows the IPA-GNN to execute differently at each step conditioned on the



information in the resource description, whether it be type information, value ranges, or candidate values.

We also consider an additional method: injecting the description as a *docstring* at the start of the program. This method yields a new valid Python program, and so any model can accommodate it.

### 4.3. Modeling Exception Handling

The final modification we make to the IPA-GNN architecture is to model exception handling. In Algorithm 1, this corresponds to adding the **magenta** text to form interpreter  $I'$ , computing a raise decision (Lines 4-6, Equation 4.3). We call the architecture that results the Exception IPA-GNN.

Whereas execution always proceeds from statement to next statement in interpreter  $I$  and in the IPA-GNN, interpreter  $I'$  admits another behavior. Under  $I'$  and the Exception IPA-GNN, execution may proceed from any statement to a surrounding “except block”, if it is contained in a try/except frame, or else to a special global error node, which we denote  $n_{\text{error}}$ . In the sample execution in Figure 15c we see at step  $t = 4$  the instruction pointer  $p_I$  updates to  $n_{\text{error}} = 8$ .

We write that the IPA-GNN makes raise decisions as

$$b_{t,n,r(n)}, (1 - b_{t,n,r(n)}) = \text{softmax} \left( \text{Dense}(a_{t,n}^{(1)}) \right). \quad (4.3)$$

The dense layer here has two outputs representing the cases that an error is and is not raised. Here  $r(n)$  denotes the node that statement  $n$  raises to;  $r(n) = n_{\text{error}}$  if  $n$  is not contained in a try/except frame, and  $b_{t,n,n'}$  denotes the probability under the model of execution transitioning from  $n$  to  $n'$ .

Next the model makes soft branch decisions in an analogous manner; the dense layer for making branch decisions has distinct weights from the layer for making raise decisions.

$$b_{t,n,n_1}, b_{t,n,n_2} = (1 - b_{t,n,r(n)}) \cdot \text{softmax} \left( \text{Dense}(a_{t,n}^{(1)}) \right). \quad (4.4)$$

The text in **magenta** corresponds to the “else” at Line 6. The model has now assigned probability to up to three possible outcomes for each node: the probability that  $n$  raises an exception  $b_{t,n,r(n)}$ , the probability that the true branch is followed  $b_{t,n,n_1}$ , and the probability that the false branch is followed  $b_{t,n,n_2}$ . In the common case where a node is not a control node and has only a single successor, the probability of reaching that successor is simply  $1 - b_{t,n,r(n)}$ .

Finally, we assign each program a step limit  $T(x)$  using the same heuristic as Bieber et al. [15], detailed in Appendix D.5. After  $T(x)$  steps of the architecture, the model directly uses the probability mass at  $n_{\text{exit}}$  and  $n_{\text{error}}$  to predict whether the program raises an error, and if so it predicts the error type using the hidden state at the error node. We write the modified

IPA-GNN’s predictions as

$$P(\text{no error}) \propto p_{T(x),n_{\text{exit}}} \text{ and } P(\text{error}) \propto p_{T(x),n_{\text{error}}}, \text{ with} \quad (4.5)$$

$$P(\text{error} = k \mid \text{error}) = \text{softmax} \left( \text{Dense}(h_{T(x),n_{\text{error}}}) \right). \quad (4.6)$$

We train with a cross-entropy loss on the class predictions, treating “no error” as its own class.

#### 4.4. Unsupervised Localization of Errors

Since the Exception IPA-GNN makes soft decisions as to when to raise an exception, we aggregate these soft decisions to obtain the model’s prediction for where a program raises an error. We use this to evaluate the model’s localization accuracy despite training without error locations as supervision.

For programs that lack try/except frames, we compute the localization predictions of the model by summing, separately for each node, the contributions from that node to the error node across all time steps. This gives an estimate of *exception provenance* as

$$p(\text{error at statement } n) = \sum_t p_{t,n} \cdot b_{t,n,n_{\text{error}}}. \quad (4.7)$$

For programs with a try/except frame, we must trace the exception back to the statement that originally raised it. To do this we calculate a recurrence as detailed in Appendix D.8.

## 5. Experiments

In our experiments we evaluate the following research questions:

**RQ1:** How does the adaptation of the IPA-GNN to realistic code compare against existing static analysis and against standard architectures like GGNN, LSTM, and Transformer? (Section 5.1)

**RQ2:** What is the impact of including resource descriptions? What methods for incorporating them work best? (Section 5.2)

**RQ3:** How interpretable are the soft instruction pointer values in the Exception IPA-GNN for localizing errors? How does unsupervised localization with the Exception IPA-GNN compare to alternative unsupervised localization approaches based on multiple instance learning? (Section 5.3)

### 5.1. Evaluation of IPA-GNN Against Baselines

We describe the experimental setup for our first experiment, comparing the IPA-GNN architectures with Transformer [143], GGNN [94], and LSTM [64] baselines. In all approaches, we use the 30,000 token vocabulary constructed in Appendix D.1, applying Byte-Pair Encoding

(BPE) tokenization [128] to tokenize each program into a sequence of token indices. The Transformer operates on this sequence of token indices directly, with its final representation computed via mean pooling. For all other models (GGNN, LSTM, IPA-GNN, and Exception IPA-GNN), the token indices are first combined via a masked (local) Transformer to produce per-node embeddings, and the model operates on these per-node embeddings as in Section 4.1. Following Bieber et al. [15] we encode programs for a GGNN using six edge types, and use a two-layer LSTM for the LSTM baseline and in all IPA-GNN variants.

In order to compare against the capabilities of a standard static analysis setup, we also consider a baseline based on pylint. For this baseline, we map a subset of the findings that pylint can identify to runtime error classes that they can indicate. The baseline predicts an error class if pylint identifies a corresponding finding. The purpose of this baseline is to consider a standard tool used by Python developers and see how it is performing on the task. We provide further details in Appendix D.7.

For each neural approach, we perform an independent hyperparameter search using random search. We list the hyperparameter space considered and model selection criteria in Appendix D.5. The models are each trained to minimize a cross-entropy loss on the target class using stochastic gradient descent for up to 500,000 steps with a mini-batch size of 32. In order to more closely match the target class distribution found in the balanced test set, we sample mini-batches such that the proportion of examples with target “no error” and those with an error target is 1:1 in expectation. We evaluate the selected models on the balanced test set, and report the results in Table 14a (see rows without check marks). Weighted F1 score (W. F1) performs a weighted average of the per-class F1 scores by class frequency, and weighted error F1 score (E. F1) does the same while restricting consideration to those examples with a runtime error.

We perform additional evaluations using the same experimental setup but distinct initializations to compute measures of variance, which we detail in Appendix D.6.

**RQ1:** The interpreter-inspired architectures show significant gains over the pylint, LSTM, GGNN and Transformer baseline approaches on the runtime error prediction task. We observe that the pylint baseline can make incorrect predictions because it correctly identifies an issue in the code under analysis when that code does not result in a runtime error in our dataset; pylint’s lower performance on runtime error prediction is not evidence against pylint’s performance for its intended use cases. We attribute the interpreter-inspired architectures’ relative success over other neural architectures to their inductive bias toward mimicking program execution.

**Table 14.** Error classification and error localization results on the balanced test set with and without resource descriptions (R.D.).

**(a)** Accuracy, weighted F1, and weighted error F1 scores. **(b)** Localization accuracy (%) for the MIL Transformers and Exception IPA-GNN.

	MODEL	R.D.?	ACC.	W. F1	E. F1		MODEL	R.D.?	LOCAL.
BASE-LINES	PYLINT		60.4	47.9	23.8		LOCAL MIL TRANSFORMER		33.0
	GGNN		62.8	58.9	45.8		LOCAL MIL TRANSFORMER	✓	48.9
	TRANSFORMER		63.6	60.4	48.1		GLOBAL MIL TRANSFORMER		48.2
	LSTM		66.1	61.4	48.4		GLOBAL MIL TRANSFORMER	✓	48.8
ABLATIONS	GGNN	✓	68.3	66.5	56.8		E. IPA-GNN		50.8
	TRANSFORMER	✓	67.3	65.1	54.7		E. IPA-GNN + DOCSTRING	✓	64.7
	LSTM	✓	68.1	66.8	58.3		E. IPA-GNN + FiLM	✓	64.5
	IPA-GNN		68.3	64.8	53.8		E. IPA-GNN + CROSS ATTENTION	✓	<b>68.8</b>
	E. IPA-GNN		68.7	64.9	53.3				
OURS	IPA-GNN	✓	71.4	70.1	62.2				
	E. IPA-GNN	✓	<b>71.6</b>	<b>70.9</b>	<b>63.5</b>				

**Table 15.** A comparison of early and late fusion methods for incorporating external resource description information into interpreter-inspired models.

MODEL	BASELINE			DOCSTRING			FiLM			CROSS-ATTENTION		
	ACC.	W. F1	E. F1	ACC.	W. F1	E. F1	ACC.	W. F1	E. F1	ACC.	W. F1	E. F1
IPA-GNN	68.3	64.8	53.8	71.4	70.1	62.2	71.6	70.3	62.9	72.0	70.3	62.6
E. IPA-GNN	68.7	64.9	53.3	71.6	70.9	63.5	70.9	68.8	59.8	73.8	72.3	64.7

## 5.2. Incorporating Resource Descriptions

We next evaluate methods of incorporating resource descriptions into the models. For each architecture we apply the docstring approach of processing resource descriptions of Section 4.2. This completes a matrix of ablations, allowing us to distinguish the effects due to architecture change from the effect of the resource description. We follow the same experimental setup as in Section 5.1, and show the results again in Table 14a (compare rows with check marks to those without).

We also consider the FiLM and cross-attention methods of incorporating resource descriptions into the IPA-GNN. Following the same experimental setup again, we show the results of this experiment in Table 15. Note that the best model overall by our model selection criteria on validation data was the IPA-GNN with cross-attention, though the Exception IPA-GNN performed better on test.

**RQ2:** Across all architectures, the results show external resource descriptions improve performance on the runtime error prediction task. On the IPA-GNN architectures, we see further improvements by considering architectures that incorporate the resource description directly into the execution step of the model, but these gains are inconsistent. The pylint baseline is unable to incorporate resource descriptions. Critically, using any resource description method is better than none at all.

To understand how the resource descriptions lead to better performance, we compare in Figure 16 the instruction pointer values of two Exception IPA-GNN models on a single example (shown in Table 16). The model with the resource description predicts that the `input()` calls will read input beyond the end of the `stdin` stream. In contrast, the model without the resource description has less reason to suspect an error would be raised by those calls. The descriptions of `stdin` in our runtime errors dataset also frequently reveal type information, expected ranges for numeric values, and formatting details about the inputs. We visualize additional examples in Appendix D.11.

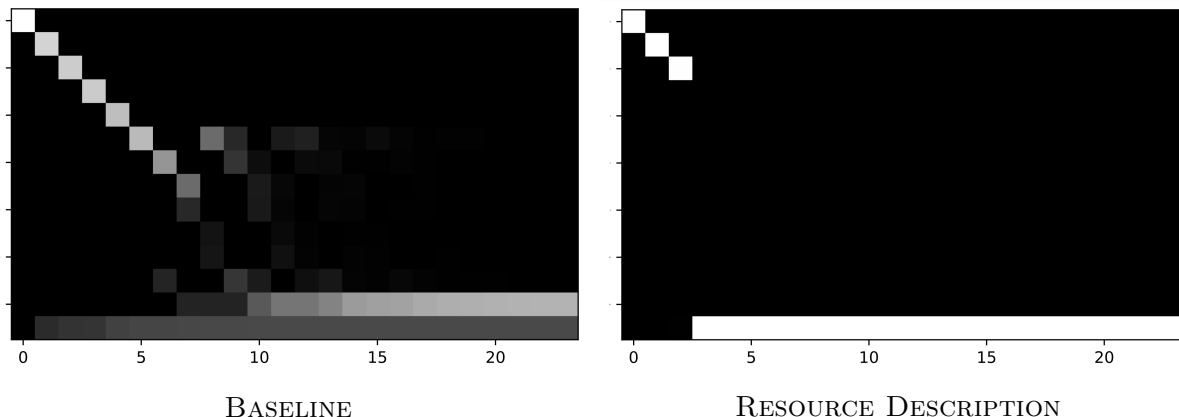
### 5.3. Interpretability and Localization

We next investigate the behavior of the Exception IPA-GNN model, evaluating its ability to localize runtime errors without any localization supervision. In unsupervised localization, the models predict the location of the error despite being trained only with error presence and kind supervision.

**Multiple Instance Learning Baselines.** Unsupervised localization may be viewed as multiple instance learning (MIL) [39]. Consider the subtask of predicting whether a particular line contains an error. In an  $n$ -line program, there are  $n$  instances of this subtask. The available supervision only indicates if any one of these subtasks has an error, but not which one. By viewing each instance as a bag of subtasks, we have cast the problem as MIL.

Using this view, we introduce two variations on the Transformer architecture as multiple instance learning baselines. The first is the “Local MIL Transformer”, in which each statement in the program is encoded individually, as in the local node embeddings computation of

**Fig. 16.** Heatmap of instruction pointer values produced by BASELINE and DOCSTRING Exception IPA-GNNs for the example in Table 16. The x-axis represents timesteps and the y-axis represents nodes, with the last two rows respectively representing  $n_{\text{exit}}$  and  $n_{\text{error}}$ . The BASELINE instruction pointer value is diffuse, with most probability mass ending at  $n_{\text{exit}}$ . The R.D. instruction pointer value is sharp, with almost all probability mass jumping to  $n_{\text{error}}$  from node 2.



**Table 16.** Per-node localization predictions from the BASELINE and DOCSTRING Exception IPA-GNN models on a sample program from the validation split. The target class is EOFERROR, occurring on line 2 ( $n = 2$ ). BASELINE predicts NO ERROR with confidence 0.708, while R.D. predicts EOFERROR with confidence 0.988, localized at line 3 ( $n = 3$ ). The input description shows the cause for error: there are more `input()` calls than the number of expected inputs.

STDIN DESCRIPTION		Input: Input is given from Standard Input in the following format Constraints: Each character of S is A or B.  S  = 3	
$n$	SOURCE	BASELINE	R.D.
0	<code>a = str(input())</code>	16.9	0.4
1	<code>q = int(input())</code>	3.2	0.3
2	<code>s = [input().split() for i in range(q)]</code>	0.5	<b>99.3</b>
3,4	<code>for i in range(q):</code>	6.4	0.0
5	<code>if int(s[i][0]) == 1 and len(a)&gt;1:</code>	0.1	0.0
6	<code>a = a[::-1]</code>	0.7	0.0
7	<code>elif int(s[i][0]) == 2 and int(s[i][1]) == 1:</code>	0.1	0.0
8	<code>a=s[i][2]+a</code>	0.2	0.0
	<code>else:</code>		
9	<code>a=a+s[i][2]</code>	0.0	0.0
10	<code>print(a)</code>	1.1	0.0

Section 4.1. The second is the “Global MIL Transformer”, in which all tokens in the program may attend to all other tokens in the Transformer encoder. In both cases, the models make per-line predictions, which are aggregated to form an overall prediction as defined in Appendix D.9.

**Localization Experiment.** Using the same protocol as Section 5.1, we train each of the MIL Transformer and Exception IPA-GNN models. As before, the models are trained only to minimize cross-entropy loss on predicting error kind and presence, receiving no error location supervision. We report the localization results in Table 14b. Localization accuracy (“LOCAL.”) measures the percent of the test examples with errors for which the model correctly predicts the error line number.

**RQ3:** The Exception IPA-GNN’s unsupervised localization capabilities far exceed that of baseline approaches. In Figure 16 we see the flow of instruction pointer mass during the execution of a sample program (Table 16) by two Exception IPA-GNN models, including the steps where the models raise probability mass to  $n_{\text{error}}$ . Tallying the contributions to  $n_{\text{error}}$  from each node yields the exception provenance values in the right half of Table 16. This shows how the model’s internal state resembles plausible program executions and allows for unsupervised localization. As a beneficial side-effect of learning plausible executions, the Exception IPA-GNN can localize the exceptions it predicts.

## 6. Discussion

In this work, we introduce the task of predicting runtime errors in competitive programming problems and advance the capabilities of interpreter-inspired models. Our models support the complexity of competition code and demonstrate that natural language descriptions of external resources can reduce the ambiguity that arises in a static analysis setting. We show that the interpreter-inspired models outperform standard alternatives and that their inductive biases allow for interesting interpretability in the context of unsupervised localization.

Though they perform best, current IPA-GNN models require taking many steps of execution, up to 174 on this dataset. A future direction is to model multiple steps of program execution with a single model step, to reduce the number of model steps necessary for long programs. Extending the interpreter-inspired models with additional interpreter features, or supporting multi-file programs or programs with multiple user-defined functions are also interesting avenues for future work.

Learning to understand programs remains a rich area of inquiry for machine learning research because of its complexity and the many aspects of code. Learning to understand execution behavior is particularly challenging as programs grow in complexity, and as they depend on more external resources whose contents are not present in the code. Our work presents a challenging problem and advances interpreter-inspired models, both of which we hope are ingredients towards making progress on these difficult and important problems.





# Conclusion

---

We conclude with a summary of the contributions of the works presented, a discussion of how the lessons learned fit in the modern context of the fast-changing field of machine learning, and what direction this points us in going forward.

## 1. Summary

In this thesis we introduced the instruction pointer attention graph neural network (IPA-GNN) family of architectures, including the IPA-GNN, RTL IPA-GNN, and Exception IPA-GNN. Examining this interpreter-inspired model family, we call attention to the systematicity properties of the models, particularly improvements in systematic generalization and interpretability, the flexibility of the architectures to model a variety of non-standard forms of execution, and the range of tasks these models are appropriate for, and the real-world value they provide. By drawing from the structure of program interpreters, these models are well-suited for modeling the execution behavior of programs, a skill that remains difficult for general purpose neural architectures. We now review the contributions of each of the articles of the thesis.

In Article 1 we introduced the `python_graphs` library for constructing graph representations of Python programs for machine learning research. This library not only supports our IPA-GNN research, it also facilitates additional research into the use of the structure of Python programs for machine learning, making such research both easier and more cost effective to pursue. Several works build on it in studying the use of program structure in machine learning for code. We use it extensively in our study of IPA-GNNs. Then, with Article 2, we introduce our first interpreter-inspired neural architecture, the IPA-GNN. IPA-GNN architectures form the basis of our contributions across each of the subsequent articles as well. Each of the IPA-GNN variants we consider can be viewed as a continuous relaxations of some discrete interpreter. In Article 2, this is a simple interpreter that can execute a subset of the Python programming language. Here we observe the IPA-GNN to exhibit improved strong generalization on learning to execute tasks, outperforming baseline models on executing long programs when trained only on short ones. This gave the first

evidence of the utility of the IPA-GNN for performing non-standard notions of execution: the IPA-GNN learns to execute both full and partial programs in a step-limited execution environment. Though the programs in these experiments were simple, the next two articles consider more realistic sets of programs. Article 3 extends the IPA-GNN to support hardware descriptions written in Verilog, yielding the register transfer level (RTL) IPA-GNN. The model family continues to show its adaptability to different forms of execution. Here, the RTL IPA-GNN models execution of a hardware description, where blocks are concurrent and repeating, in contrast with the earlier single threaded non-repeating Python programs. The results of the RTL IPA-GNN on cover point prediction are valuable to design verification engineers, highlighting the real-world utility of this research. Finally, with Article 4, we consider the developer task of predicting runtime errors. The IPA-GNN family of models is effective on this task, outperforming a suite of general purpose baselines including a popular static analysis tool. This task again highlights the real-world utility of the research, and accordingly we make our dataset and models available publicly to facilitate further research. The flexibility of the IPA-GNN to model different forms of execution here allows for modeling exception handling with the Exception IPA-GNN. The inductive bias of this architecture allows a new interpretability capability: it enables localizing errors, despite training the model only on error kind and presence, not error location.

In total these contributions represent both tangible progress on meaningful applications in program analysis, as well as a promising direction for continued improvements to machine learning models of programs and their executions.

## 2. Placing in Modern Context

The works presented in this thesis span multiple years, from 2019 to 2023. In this time, machine learning has advanced considerably. In this section we review the recent rapid advancement of machine learning for code in this time and the implications for our research.

In March 2019, shortly after the start of my PhD, Richard Sutton published “The Bitter Lesson” [136]. Sutton’s message is that, in the long term, the methods that prevail are the ones that scale to absorb increased availability of compute, and that the methods that consistently do this are search and learning. Sure enough, in the years that followed, advancements in model and data scale lead to impressive new capabilities in machine learning for code. In August 2021, OpenAI released Codex which enabled practical open domain natural language programming. That same month, Google published Austin et al. [11] showing large language models performing program synthesis. Then, in February 2022, DeepMind released AlphaCode, notable for achieving median human performance at competitive programming. These milestones on the one hand have reinforced “The Bitter Lesson”. At the same time,

despite the increasing scale of models and data, tasks requiring reasoning about program executions remained difficult.

The rapid progress that these publications indicate was brought about primarily through scale and transfer from unsupervised learning. With unsupervised learning, models can leverage large amounts of data that are not directly related to the task of interest, in order to learn broadly useful representations of inputs. This allows scaling data sources without the expensive labeling process required for scaling supervised learning datasets. Between this and advancements in hardware and frameworks for machine learning, researchers achieved unprecedented model and data scales. The models used in the accomplishments noted above are Transformers with hundreds of billions of parameters, and were trained on hundreds of billions of tokens of input data.

In this time period, there were also advancements in our understanding of scaling laws for deep neural networks, for example with Kaplan et al. [77] published in January 2020. As the amount of data available for any particular task grows, general purpose neural networks become better function approximators. For any desired error level greater than the intrinsic entropy of the task, there is some finite amount of data (possibly intractably large), that allows the model to achieve that error level with a general purpose neural network. Large models trained on large volumes of data demonstrate impressive transfer capabilities, leveraging what they learn from their pre-training data to perform well on tasks related to, but not directly represented by, this data. The impressive recent advances in machine learning for code are possible only because of these properties. The large Transformers used in Codex and AlphaCode were first pretrained on large amounts of code, and then fine-tuned for their downstream tasks. So, in a machine learning landscape dominated by large general purpose models pretrained on large amounts of both natural language and code data, how do our contributions fit in?

Comparatively, the models we examine in this thesis are tiny. The 8 million parameter models in Article 4 are 100,000 times smaller than the largest models in use today. A key question, therefore, is whether the interpreter-inspired models scale effectively. Do they retain their interpretability and systematic generalization benefits as they are scaled?

We can view the inductive bias introduced by using an interpreter-inspired model as providing information about the underlying task to the model. In the large data limit, this additional information baked into the architecture does not contribute value. However, in practice, data is limited, and on the realistic data volumes used in our articles we find that the intrinsic prior indeed does matter substantially. Our setting did not consider pre-training on unsupervised tasks; once this is taken into consideration, the total volume of data is much higher. In this setting, does the intrinsic prior still have value? Are interpreter-inspired architectures like the IPA-GNN suitable for taking advantage of the large amounts of unlabeled code data on GitHub.

We present some ways in which the IPA-GNN architecture family might leverage both increased data scale and transfer learning. Models like the IPA-GNNs of Article 4, which use a Transformer encoder followed by an IPA-GNN interpreter, can be scaled along multiple dimensions. The Transformer encoder can be scaled up independently of the IPA-GNN. To fully take advantage of transfer learning on large datasets, both general purpose pre-training tasks and execution-specific pre-training are possible. The Transformer encoder can leverage the general purpose pre-training tasks that are already useful in natural language processing today. A natural line of research is to ask: what are the intrinsic unsupervised tasks appropriate for pre-training a model centered around modeling execution? These could be tasks that use data from an execution or partial execution (e.g. given some values for variables, predict a property of the state after several steps of execution, predict whether a property will become true after execution, etc), or tasks that rely only on syntactic properties (e.g. if two statements execute, what must have executed in between?). Future work in this direction will be critical for understanding the role large interpreter-inspired models will play in predicting behavioral properties of programs moving forward.

### 3. Future Directions

The context laid out above naturally shapes our curiosity about interpreter-inspired architectures. What are their scaling characteristics, as compared with general purpose models? Supposing we pre-train one or more parts of an interpreter-inspired model, how well does information from unsupervised pre-training transfer for execution-specific tasks? How well do they incorporate transfer learning from unsupervised training? In the runtime error predictions paper, the parameter count is dominated by parameters in the Transformer encoder, not in the IPA-GNN. As we scale an interpreter inspired architecture, what is the trade-off between scaling the encoder that produces statement embeddings, and scaling the IPA-GNN specific parameters that perform relaxed execution over these embeddings? What is the trade-off between scaling data and scaling compute? How does transfer factor in? We know that for producing good embeddings of source code, pre-training on unsupervised tasks is fruitful; are there pre-training tasks particularly fruitful when using embeddings in an execution aware setting?

The IPA-GNN uses many steps of execution, one model step per execution step in the common case. This results in models that use more computation than a fixed depth Transformer of comparable size. One direction we are exploring is modeling multiple steps of execution in a single step of the model. For this, we are considering approaches that allow execution not just following the statement-level control-flow graph directly, but instead executing groups of statements simultaneously. This allows for modeling longer programs

using fewer model steps overall, while still preserving the inductive bias toward mimicking program executions.

The large language models that have led to success in tasks like natural language programming and competitive programming gain their capabilities from both scale and transfer. As we investigate scaling up interpreter-inspired models like the IPA-GNN, we must ask what pre-training tasks are appropriate to give the benefits of transfer. The IPA-GNN in Article 4 uses a Transformer encoder. This can be pretrained using the unsupervised pre-training tasks that have shown success in natural language processing and in execution-agnostic models of code. For example, next token prediction, masked language modeling, and span denoising have each shown great promise.

An open research question is what are the appropriate execution-specific pre-training tasks to maximize the benefits for transfer learning. One approach we are considering is leveraging trace data as additional supervision for training the IPA-GNN, with the goal of reducing the sample complexity allowing the model to learn about the programming language semantics more quickly. Using traces as supervision provides a much greater amount of information per example during training compared with only runtime error labels. Additional research is required to identify pre-training tasks that are suitable for unsupervised learning for the IPA-GNN.

The interpreter-inspired models that we considered in this thesis focused on modeling control flow. Other aspects of program interpreters have been explored in prior works, and there remains room to model additional components of an interpreter. The key is to strike a balance between imposing a prescriptive inductive bias on the model with allowing the model the freedom to learn the most suitable representations. Key aspects of interpreters to study in further depth are data flow, function call stacks, and the separation of concerns (distinct variables) in program state.

Improvements in execution-aware machine learning bring us closer to building developer tools that can help developers save time. One aspiration of this research is to build a system that can alert developers when they launch a long-running job if that job is likely to fail. If we can make this alert with high confidence, this can save developers significant time, allowing them to fix the software before waiting for it to schedule, run, and crash. The applications that we study in this thesis already demonstrate real world value: coverage prediction during hardware validation assists design verification engineers, and runtime error prediction serves as an early warning developer tool that helps competitive programmers save time. New applications like early crash prediction could provide further value, saving developers time and perhaps preventing service outages.

Two fundamental advances that need to occur to fully tackle these target applications are handling large codebases, and handling multi-service programs that communicate with one another over a network. The research in this thesis processes only single-file programs, and

the interpreter-inspired architectures are not designed to handle interprocess communication or inter-service communication. Another capability that may be useful for crash prediction is giving the models access to the logs of recent similar jobs. Long context models like Jaegle et al. [68] and retrieval models like Borgeaud et al. [19] provide an interesting direction of study for providing these missing capabilities.

One of the reasons we study tasks requiring reasoning about execution behavior is that these tasks retain significant headroom even for large language models. As we measure the scaling properties of interpreter-inspired architectures and compare them with general purpose language model, it becomes important to identify whether there is anything fundamental about these execution tasks that keeps them challenging for fixed depth models. Stochastic depth models (models whose depth varies according to a distribution from example to example) are a promising approach to tasks with widely varying computational requirements like execution, and form an interesting generalization of the variable depth property of IPA-GNN models.

This thesis raises more questions than it answers, and points us in a promising direction going forward toward improved models of code and code execution, and in turn toward a promising future of developer tools.

## References

---

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006. ISBN 0321486811.
- [2] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 281–293, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi: 10.1145/2635868.2635883. URL <https://doi.org/10.1145/2635868.2635883>.
- [3] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81, 2018.
- [4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018. URL <https://openreview.net/forum?id=BJOFETxR->.
- [5] Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. Self-supervised bug detection and repair, 2021. URL <https://arxiv.org/abs/2105.12787>.
- [6] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970. ISSN 0362-1340. doi: 10.1145/390013.808479. URL <https://doi.org/10.1145/390013.808479>.
- [7] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code, 2018. URL <https://arxiv.org/abs/1808.01400>.
- [8] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code, 2018. URL <https://arxiv.org/abs/1803.09473>.
- [9] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models of code, 2019. URL <https://arxiv.org/abs/1910.00577>.
- [10] Viraj Athavale, Sai Ma, Samuel Hertz, and Shobha Vasudevan. Code coverage of assertions using rtl source code analysis. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2014. doi: 10.1145/2593069.2593108.

- [11] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.
- [12] Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008. doi: 10.1109/MS.2008.130.
- [13] Dzmitry Bahdanau, Shikhar Murty, Michael Noukhovitch, Thien Huu Nguyen, Harm de Vries, and Aaron Courville. Systematic generalization: What is required and can it be learned? In *International Conference on Learning Representations*, 2019.
- [14] K. Bansal and M. S. Hsiao. Optimization of mutant space for rtl test generation. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 472–475, 2018.
- [15] David Bieber, Charles Sutton, Hugo Larochelle, and Daniel Tarlow. Learning to execute programs with instruction pointer attention graph neural networks. In *Advances in Neural Information Processing Systems*, 2020.
- [16] David Bieber, Rishab Goel, Daniel Zheng, Hugo Larochelle, and Daniel Tarlow. Static prediction of runtime errors by learning to execute programs with external resource descriptions, 2022. URL <https://arxiv.org/abs/2203.03771>.
- [17] David Bieber, Kensen Shi, Petros Maniatis, Charles Sutton, Vincent Hellendoorn, Daniel Johnson, and Daniel Tarlow. A library for representing python programs as graphs for machine learning, 2022. URL <https://arxiv.org/abs/2208.07461>.
- [18] BIG-bench collaboration. Beyond the imitation game: Measuring and extrapolating the capabilities of language models. *In preparation*, 2021. URL <https://github.com/google/BIG-bench/>.
- [19] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. Improving language models by retrieving from trillions of tokens, 2021. URL <https://arxiv.org/abs/2112.04426>.
- [20] Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. Programming with a differentiable forth interpreter, 2017.
- [21] Marc Brockschmidt. GNN-FiLM: Graph neural networks with feature-wise linear modulation. In *International Conference on Machine Learning (ICML)*, 2020.
- [22] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. Generative code modeling with graphs. In *International Conference on Learning Representations*, 2019. URL <https://openreview.net/forum?id=Bke4KsA5FX>.



- [23] Max Brunsfeld, Patrick Thomson, Andrew Hlynskyi, Josh Vera, Phil Turnbull, Timothy Clem, Douglas Creager, Andrew Helwer, Rob Rix, Hendrik van Antwerpen, Michael Davis, Ika, Tuan-Anh Nguyen, Stafford Brunk, Niranjana Hasabnis, bfredl, Mingkai Dong, Vladimir Panteleev, ikrima, Steven Kalt, Kolja Lampe, Alex Pinkus, Mark Schmitz, Matthew Krupcale, narpfel, Santos Gallegos, Vicent Martí, Edgar, and George Fraser. tree-sitter/tree-sitter: v0.20.6, March 2022. URL <https://doi.org/10.5281/zenodo.6326492>.
- [24] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. Infercode: Self-supervised learning of code representations by predicting subtrees, 2020. URL <https://arxiv.org/abs/2012.07023>.
- [25] Dan Busbridge, Dane Sherburn, Pietro Cavallo, and Nils Y. Hammerla. Relational graph attention networks, 2019. URL <https://openreview.net/forum?id=Bklzkh0qFm>.
- [26] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.
- [27] Jonathon Cai, Richard Shin, and Dawn Song. Making neural programming architectures generalize via recursion, 2017.
- [28] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Balancing scalability and uniformity in SAT witness generator. *CoRR*, abs/1403.6246, 2014. URL <http://arxiv.org/abs/1403.6246>.
- [29] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.
- [30] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost, 2016.
- [31] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2019. URL <https://openreview.net/forum?id=H1gf0iAqYm>.

- [32] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noel Pouchet, Denys Poshyvanyk, and Martin Monperrus. SEQUENCER: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, pages 1–1, 2021. doi: 10.1109/tse.2019.2940179. URL <https://doi.org/10.1109%2Ftse.2019.2940179>.
- [33] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, September 2003.
- [34] Hanjun Dai, Yujia Li, Chenglong Wang, Rishabh Singh, Po-Sen Huang, and Pushmeet Kohli. Learning transferable graph exploration. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d’Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *NeurIPS*, pages 2514–2525, 2019.
- [35] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.
- [36] Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. Path-based function embedding and its application to specification mining, 2018. URL <https://arxiv.org/abs/1802.07779>.
- [37] Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers, 2019.
- [38] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o, 2017. URL <https://arxiv.org/abs/1703.07469>.
- [39] Thomas G. Dietterich, Richard H. Lathrop, and Tomás Lozano-Pérez. Solving the multiple instance problem with axis-parallel rectangles. *Artificial Intelligence*, 89(1):31–71, 1997. ISSN 0004-3702. doi: [https://doi.org/10.1016/S0004-3702\(96\)00034-3](https://doi.org/10.1016/S0004-3702(96)00034-3). URL <https://www.sciencedirect.com/science/article/pii/S0004370296000343>.
- [40] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations*, 2019.
- [41] David Dohan, Winnie Xu, Aitor Lewkowycz, Jacob Austin, David Bieber, Raphael Gontijo Lopes, Yuhuai Wu, Henryk Michalewski, Rif A. Saurous, Jascha Sohl-dickstein, Kevin Murphy, and Charles Sutton. Language model cascades, 2022. URL <https://arxiv.org/abs/2207.10342>.
- [42] E.A. Emerson and A.P. Sistla. Symmetry and model checking. In *Formal Methods in System Design*, pages 105–131, 1996.
- [43] Monica Farkash, Bryan Hickerson, and Michael Behm. Coverage learned targeted validation for incremental hw changes. In *Proceedings of the 51st Annual Design Automation Conference*, page 1–6, 2014.

- [44] Harry Foster. Why the design productivity gap never happened. In Jörg Henkel, editor, *The IEEE/ACM International Conference on Computer-Aided Design, ICCAD'13, San Jose, CA, USA, November 18-21, 2013*, pages 581–584. IEEE, 2013. doi: 10.1109/ICCAD.2013.6691175. URL <https://doi.org/10.1109/ICCAD.2013.6691175>.
- [45] Harry D. Foster. Trends in functional verification: a 2014 industry study. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 48:1–48:6. ACM, 2015. doi: 10.1145/2744769.2744921. URL <https://doi.org/10.1145/2744769.2744921>.
- [46] Harry D. Foster. Functional Verification study. <https://blogs.sw.siemens.com/verificationhorizons/2021/01/06/part-8-the-2020-wilson-research-group-functional-verification-2021>.
- [47] Alexander L Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. Differentiable programs with neural libraries. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1213–1222. JMLR. org, 2017.
- [48] Dobrik Georgiev, Marc Brockschmidt, and Miltiadis Allamanis. HEAT: Hyperedge attention networks, 2022. URL <https://arxiv.org/abs/2201.12113>.
- [49] GitHub, 2020. URL <https://octoverse.github.com/>.
- [50] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.
- [51] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: machine learning for input fuzzing. In Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen, editors, *ASE*, pages 50–59. IEEE Computer Society, 2017.
- [52] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Eliot Karro, and D. Sculley, editors. *Google Vizier: A Service for Black-Box Optimization*, 2017. URL <http://www.kdd.org/kdd2017/papers/view/google-vizier-a-service-for-black-box-optimization>.
- [53] Rahul Gopinath. pycfg: The Python control flow graph, 2019. URL <https://rahul.gopinath.org/post/2019/12/08/python-controlflow/>.
- [54] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing Machines. *arXiv e-prints*, art. arXiv:1410.5401, Oct 2014.
- [55] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, October 2016. ISSN 00280836. URL <http://dx.doi.org/10.1038/nature20101>.

- [56] Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory. In *Advances in neural information processing systems*, pages 1828–1836, 2015.
- [57] Andreas Griewank and Andrea Walther. Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Trans. Math. Softw.*, 26(1):19–45, mar 2000. ISSN 0098-3500. doi: 10.1145/347837.347846. URL <https://doi.org/10.1145/347837.347846>.
- [58] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. GraphCodeBERT: Pre-training code representations with data flow, 2020. URL <https://arxiv.org/abs/2009.08366>.
- [59] Yang Guo, Wanxia Qu, Tun Li, and Sikun Li. Coverage driven test generation framework for rtl functional verification. In *2007 10th IEEE International Conference on Computer-Aided Design and Computer Graphics*, pages 321–326, 2007.
- [60] Rajiv Gupta, Mary Jean, Harrold Mary, and Lou Soffa. Program slicing-based regression testing techniques+. *Journal of Software Testing, Verification, and Reliability*, 6:83–112, 1996.
- [61] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi: 10.1109/CVPR.2016.90.
- [62] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International Conference on Learning Representations*, 2020. URL <https://openreview.net/forum?id=B1lnbRNtwr>.
- [63] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps, 2021. URL <https://arxiv.org/abs/2105.09938>.
- [64] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL <https://doi.org/10.1162/neco.1997.9.8.1735>.
- [65] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search, 2019. URL <https://arxiv.org/abs/1909.09436>.
- [66] Charles Hymans. Design and implementation of an abstract interpreter for vhdl. In Daniel Geist and Enrico Tronci, editors, *Correct Hardware Design and Verification Methods*, pages 263–269, 2003.
- [67] IBEX RTL source. IBEX RTL source. <https://github.com/lowRISC/ibex>, 2016.

- [68] Andrew Jaegle, Felix Gimeno, Andrew Brock, Andrew Zisserman, Oriol Vinyals, and Joao Carreira. Perceiver: General perception with iterative attention, 2021. URL <https://arxiv.org/abs/2103.03206>.
- [69] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *International Conference on Software Engineering (ICSE)*, pages 96–105, 2007. doi: 10.1109/ICSE.2007.30.
- [70] Daniel D. Johnson, Hugo Larochelle, and Daniel Tarlow. Learning graph structure with a finite-state automaton layer, 2020.
- [71] Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in neural information processing systems*, pages 190–198, 2015.
- [72] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, jun 2017. ISSN 0163-5964. doi: 10.1145/3140659.3080246. URL <https://doi.org/10.1145/3140659.3080246>.
- [73] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, San Jose, CA, USA, July 2014. Tool demo.
- [74] Łukasz Kaiser and Ilya Sutskever. Neural GPUs Learn Algorithms. *arXiv e-prints*, art. arXiv:1511.08228, Nov 2015.
- [75] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples, 2018.

- [76] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 5110–5121. PMLR, 13–18 Jul 2020. URL <https://proceedings.mlr.press/v119/kanade20a.html>.
- [77] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.
- [78] Rafael-Michael Karampatsis and Charles Sutton. How often do single-statement bugs occur? *Proceedings of the 17th International Conference on Mining Software Repositories*, Jun 2020. doi: 10.1145/3379597.3387491. URL <http://dx.doi.org/10.1145/3379597.3387491>.
- [79] Samuel J. Kaufman, Phitchaya Mangpo Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. A learned performance model for tensor processing units, 2020. URL <https://arxiv.org/abs/2008.01040>.
- [80] Y. Kesten and A. Pnueli. Control and data abstraction: the cornerstones of practical formal verification. *International Journal of Software Tools for Technology Transfer*, 2: 328–342, 2000.
- [81] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers, 2020. URL <https://arxiv.org/abs/2003.13848>.
- [82] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL <http://arxiv.org/abs/1412.6980>.
- [83] Thomas N Kipf and Max Welling. Semi-Supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.
- [84] Daniel Kroening, Alex Groce, and Edmund Clarke. Counterexample guided abstraction refinement via program execution. In *International Conference on Formal Engineering Methods*, volume 3308, 08 2004.
- [85] Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines, 2015. URL <https://arxiv.org/abs/1511.06392>.
- [86] Kevin Laeuffer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. Rfuzz: Coverage-directed fuzz testing of rtl on fpgas. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2018.
- [87] Davy Landman, Alexander Serebrenik, and Jurgen Vinju. Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods. In *IEEE International Conference on Software Maintenance and Evolution*, pages 221–230, 2014.

- doi: 10.1109/ICSME.2014.44.
- [88] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam R. Kosiosek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks, 2019.
  - [89] Li Li, Jiawei Wang, and Haowei Quan. Scalpel: The Python static analysis framework, 2022. URL <https://arxiv.org/abs/2202.11840>.
  - [90] Mingzhe Li, Jianrui Pei, Jin He, Kevin Song, Frank Che, Yongfeng Huang, and Chitai Wang. Using GGNN to recommend log statement level, 2019. URL <https://arxiv.org/abs/1912.05097>.
  - [91] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 169–180, 2019.
  - [92] Xuechen Li, Chris J. Maddison, and Daniel Tarlow. Learning to extend program graphs to work-in-progress code, 2021. URL <https://arxiv.org/abs/2105.14038>.
  - [93] Yi Li, Shaohua Wang, and Tien N. Nguyen. Fault localization with code coverage representation learning, 2021. URL <https://arxiv.org/abs/2103.00270>.
  - [94] Yujia Li, Richard Zemel, Marc Brockschmidt, and Daniel Tarlow. Gated graph sequence neural networks. In *International Conference on Learning Representations (ICLR)*, 2016.
  - [95] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode, 2022. URL <https://arxiv.org/abs/2203.07814>.
  - [96] Chang Liu, Xin Wang, Richard Shin, Joseph E. Gonzalez, and Dawn Song. Neural code completion, 2017. URL <https://openreview.net/forum?id=rJbPBt9lg>.
  - [97] Lingyi Liu and Shobha Vasudevan. Efficient validation input generation in rtl by hybridized source code analysis. In *2011 Design, Automation Test in Europe*, pages 1–6, 2011.
  - [98] Lingyi Liu, David Sheridan, William Tuohy, and Shobha Vasudevan. A technique for test coverage closure using goldmine. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(5):790–803, 2012.
  - [99] Shangqing Liu, Cuiyun Gao, Sen Chen, Lun Yiu Nie, and Yang Liu. Atom: Commit message generation based on abstract syntax tree and hybrid ranking, 2019. URL <https://arxiv.org/abs/1912.02972>.

- [100] V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX security symposium*, volume 14, pages 18–18, 2005.
- [101] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 664–676, 2021.
- [102] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021. URL <https://arxiv.org/abs/2102.04664>.
- [103] Yangdi Lyu and Prabhat Mishra. Automated test generation for activation of assertions in RTL models. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, page 223–228, 2020.
- [104] Yangdi Lyu, Xiaoke Qin, Mingsong Chen, and Prabhat Mishra. Directed test generation for validation of cache coherence protocols. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(1):163–176, 2019.
- [105] Oege de Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjorn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble. Keynote address: .ql for source code analysis. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 3–16, 2007. doi: 10.1109/SCAM.2007.31.
- [106] Rafael Müller, Simon Kornblith, and Geoffrey E. Hinton. When does label smoothing help? In *Advances in Neural Information Processing Systems*, pages 4696–4705, 2019.
- [107] Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent, 2015. URL <https://arxiv.org/abs/1511.04834>.
- [108] Flemming Nielson and Hanne Riis Nielson. Interprocedural control flow analysis. In *ESOP*, 1999.
- [109] Maxwell Nye, Yewen Pu, Matthew Bowers, Jacob Andreas, Joshua B. Tenenbaum, and Armando Solar-Lezama. Representing partial programs with blended abstract semantics, 2021.
- [110] Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models, 2022. URL <https://openreview.net/forum?id=>



iedYJm92o0a.

- [111] Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, Charles Sutton, and Hanjun Dai. BUSTLE: Bottom-up program synthesis through learning-guided exploration. In *International Conference on Learning Representations*, 2021. URL <https://openreview.net/forum?id=yHeg4PbFHh>.
- [112] Jen-Chieh Ou, Daniel G. Saab, Qiang Qiang, and Jacob A. Abraham. Reducing verification overhead with rtl slicing. In *Proceedings of the 17th ACM Great Lakes Symposium on VLSI*, page 399–404, 2007.
- [113] Pardis Pashakhanloo, Aaditya Naik, Yuepeng Wang, Hanjun Dai, Petros Maniatis, and Mayur Naik. CodeTrek: Flexible modeling of code using an extensible relational representation. In *International Conference on Learning Representations*, 2022. URL <https://openreview.net/forum?id=WQc075jmBmf>.
- [114] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. *StateFormer: Fine-Grained Type Recovery from Binaries Using Generative State Modeling*, page 690–702. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450385626. URL <https://doi.org/10.1145/3468264.3468607>.
- [115] Kexin Pei, Kensen Shi, Pengcheng Yin, Charles Sutton, and **David Bieber**. Can large language models reason about program behavior? *Under review*, 2023.
- [116] Ethan Perez, Florian Strub, Harm de Vries, Vincent Dumoulin, and Aaron Courville. *Film: Visual reasoning with a general conditioning layer*, 2017.
- [117] Rachel Potvin and Josh Levenberg. Why google stores billions of lines of code in a single repository. *Commun. ACM*, 59(7):78–87, June 2016. ISSN 0001-0782. doi: 10.1145/2854146. URL <https://doi.org/10.1145/2854146>.
- [118] Michael Pradel and Koushik Sen. Deepbugs: A learning approach to name-based bug detection, 2018. URL <https://arxiv.org/abs/1805.11683>.
- [119] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks, 2021. URL <https://arxiv.org/abs/2105.12655>.
- [120] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing, 2017. URL <https://arxiv.org/abs/1704.07535>.
- [121] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". *SIGPLAN Not.*, 50(1):111–124, jan 2015. ISSN 0362-1340. doi: 10.1145/2775051.2677009. URL <https://doi.org/10.1145/2775051.2677009>.
- [122] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. *SIGPLAN Not.*, 51(10):731–747, oct 2016. ISSN 0362-1340. doi:

- 10.1145/3022671.2984041. URL <https://doi.org/10.1145/3022671.2984041>.
- [123] Scott Reed and Nando de Freitas. Neural Programmer-Interpreters. *arXiv e-prints*, art. arXiv:1511.06279, Nov 2015.
  - [124] Canakci Sadullah, Delshadtehrani Leila, Eris Furkan, Taylor Michael Bedford, Egele Manuel, and Joshi Ajay. Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing. In *58th IEEE/ACM Design Automation Conference Proceedings, 2021*, 2021.
  - [125] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.
  - [126] Jessica Schrouff, Kai Wohlfahrt, Bruno Marnette, and Liam Atkinson. Inferring javascript types using graph neural networks, 2019.
  - [127] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. *ACM SIGSOFT Software Engineering Notes*, 30(5):263–272, 2005.
  - [128] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units, 2016.
  - [129] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy, SP 2019*,, pages 803–817. IEEE, 2019.
  - [130] Kensen Shi, David Bieber, and Charles Sutton. Incremental sampling without replacement for sequence models. In *Proceedings of the 37th International Conference on Machine Learning*, 2020.
  - [131] Kensen Shi, David Bieber, and Rishabh Singh. TF-Coder: Program synthesis for tensor manipulations. *ACM Trans. Program. Lang. Syst.*, 44(2), may 2022. ISSN 0164-0925. doi: 10.1145/3517034. URL <https://doi.org/10.1145/3517034>.
  - [132] Zhan Shi, Kevin Swersky, Daniel Tarlow, Parthasarathy Ranganathan, and Milad Hashemi. Learning execution through neural code fusion. In *International Conference on Learning Representations*, 2020. URL <https://openreview.net/forum?id=SJetQpEYvB>.
  - [133] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Learning to combine per-example solutions for neural program synthesis, 2021. URL <https://arxiv.org/abs/2106.07175>.
  - [134] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In S Bengio, H Wallach, H Larochelle, K Grauman, N Cesa-Bianchi, and R Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 7751–7762. Curran Associates, Inc., 2018.
  - [135] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and

- K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3104–3112. Curran Associates, Inc., 2014. URL <http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf>.
- [136] Richard Sutton. The bitter lesson. *Incomplete Ideas (blog)*, 13:12, 2019.
- [137] Karthik Chandra Swarna, Noble Saji Mathews, Dheeraj Vagavolu, and Sridhar Chimalakonda. A mocktail of source code representations, 2021. URL <https://arxiv.org/abs/2106.10918>.
- [138] Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. Learning to fix build errors with graph2diff neural networks, 2019. URL <https://arxiv.org/abs/1911.01205>.
- [139] Yahya Tashtoush, Mohammed Al-Maolegi, and Bassam Arkok. The correlation among software complexity metrics with case study. *arXiv e-prints*, 2014. doi: 10.48550/arxiv.1408.4523. URL <https://arxiv.org/abs/1408.4523>.
- [140] Andrew Trask, Felix Hill, Scott Reed, Jack Rae, Chris Dyer, and Phil Blunsom. Neural arithmetic logic units, 2018.
- [141] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. Neural program repair by jointly learning to localize and repair. In *International Conference on Learning Representations*, 2019.
- [142] Shobha Vasudevan, Wenjie (Joe) Jiang, David Bieber, Rishabh Singh, Hamid Shojaei, C. Richard Ho, and Charles Sutton. Learning semantic representations to verify hardware designs. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 23491–23504. Curran Associates, Inc., 2021. URL <https://proceedings.neurips.cc/paper/2021/file/c5aa65949d20f6b20e1a922c13d974e7-Paper.pdf>.
- [143] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. *arXiv e-prints*, art. arXiv:1706.03762, Jun 2017.
- [144] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2017.
- [145] Verilog. IEEE standard for SystemVerilog—unified hardware design, specification, and verification language, 2018.
- [146] Anh Viet Phan, Minh Le Nguyen, and Lam Thu Bui. Convolutional neural networks over control flow graphs for software defect prediction. In *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 45–52, 2017. doi: 10.1109/ICTAI.2017.00019.
- [147] Wenhan Wang, Ge Li, Sijie Shen, Xin Xia, and Zhi Jin. Modular tree network for source code representation learning, 2021. URL <https://arxiv.org/abs/2104.00196>.

- [148] Yanlin Wang and Hui Li. Code completion by modeling flattened abstract syntax trees as graphs, 2021. URL <https://arxiv.org/abs/2103.09499>.
- [149] Yu Wang, Fengjuan Gao, Linzhang Wang, and Ke Wang. Learning semantic program embeddings with graph interval neural network, 2020. URL <https://arxiv.org/abs/2005.09997>.
- [150] Yun Wang, Juncheng Li, and Florian Metze. Comparing the max and noisy-or pooling functions in multiple instance learning for weakly supervised sequence learning tasks, 2018.
- [151] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. Lambdanet: Probabilistic type inference using graph neural networks. In *International Conference on Learning Representations*, 2020. URL <https://openreview.net/forum?id=Hkx6hANtWH>.
- [152] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98, 2016.
- [153] Hasini Witharana, Yangdi Lyu, and Prabhat Mishra. Directed test generation for activation of security assertions in rtl models. *ACM Transactions on Design Automation of Electronic Systems*, 26(4), 2021.
- [154] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security Symposium*, volume 15, pages 179–192, 2006.
- [155] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, 2014. doi: 10.1109/SP.2014.44.
- [156] Pengcheng Yin and Graham Neubig. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation, 2018. URL <https://arxiv.org/abs/1810.02720>.
- [157] Cunxi Yu and Maciej Ciesielski. Automatic word-level abstraction of datapath. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1718–1721, 2016.
- [158] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. Neural detection of semantic code clones via tree-based convolution. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 70–80, 2019. doi: 10.1109/ICPC.2019.00021.
- [159] Wojciech Zaremba and Ilya Sutskever. Learning to execute, 2014. URL <https://arxiv.org/abs/1410.4615>.
- [160] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794, 2019. doi: 10.1109/ICSE.2019.00086.

- [161] Kechi Zhang, Wenhan Wang, Huangzhao Zhang, Ge Li, and Zhi Jin. Learning to represent programs with heterogeneous graphs, 2020. URL <https://arxiv.org/abs/2012.04188>.
- [162] Zhuo Zhang, Yan Lei, Xiaoguang Mao, and Panpan Li. Cnn-fl: An effective approach for localizing faults using convolutional neural networks. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 445–455. IEEE, 2019.
- [163] Rui Zhao, David Bieber, Kevin Swersky, and Daniel Tarlow. Neural networks for modeling source code edits, 2019. URL <https://arxiv.org/abs/1904.02818>.
- [164] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks, 2019. URL <https://arxiv.org/abs/1909.03496>.
- [165] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. Language-agnostic representation learning of source code from structure and context, 2021. URL <https://arxiv.org/abs/2103.11318>.

# A. Appendixes for Article 1

## A.1. Program Graph Visualizations

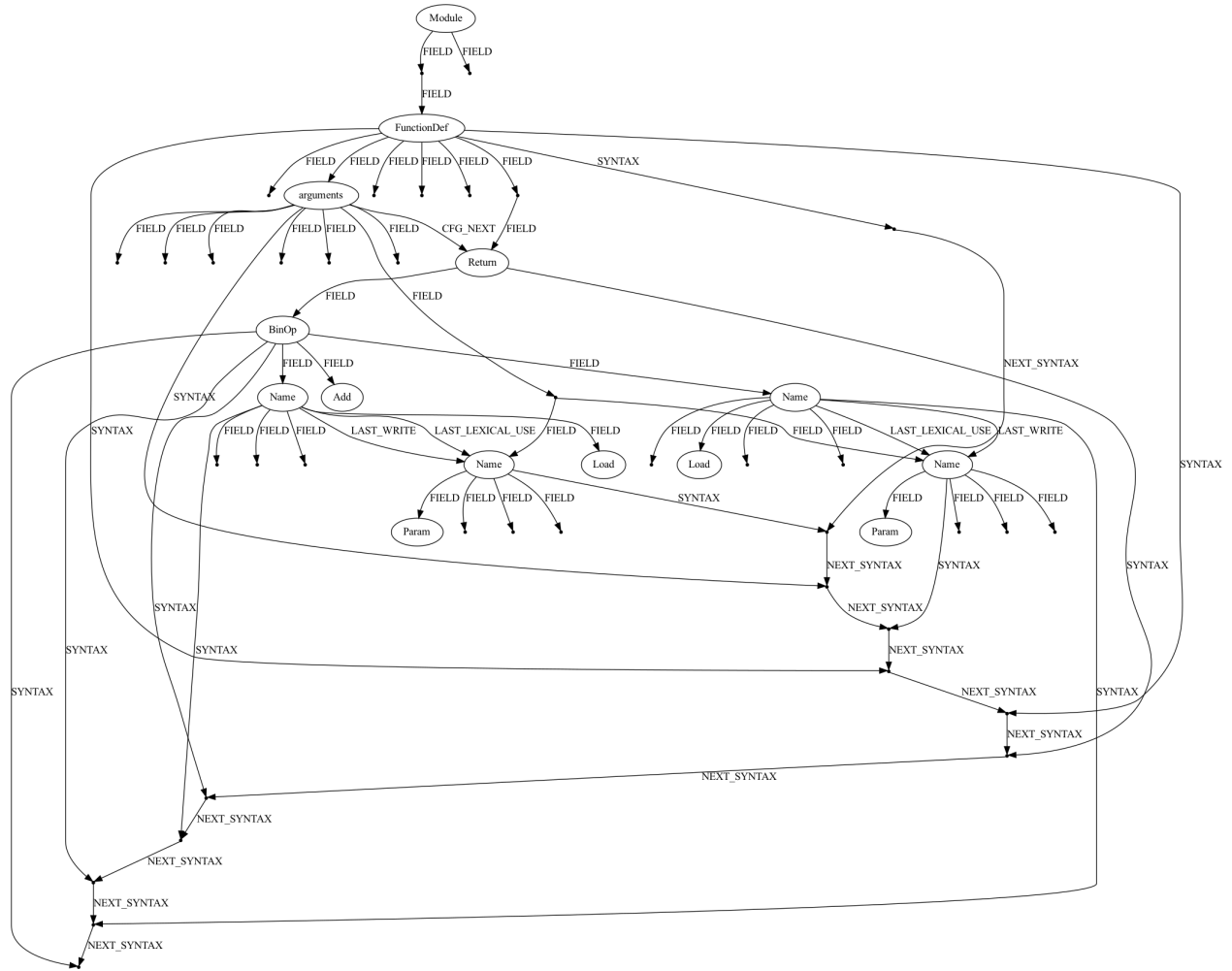
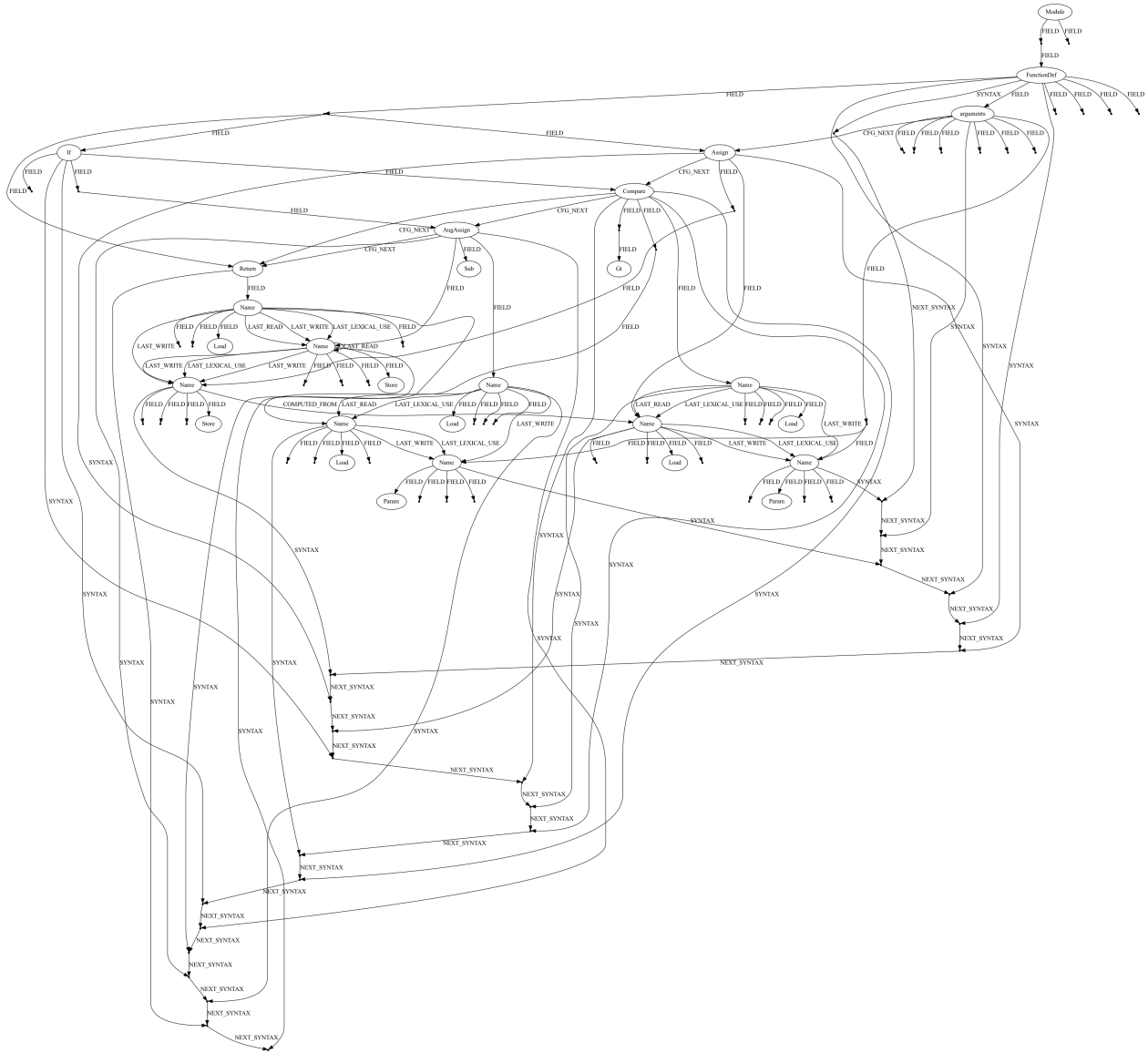


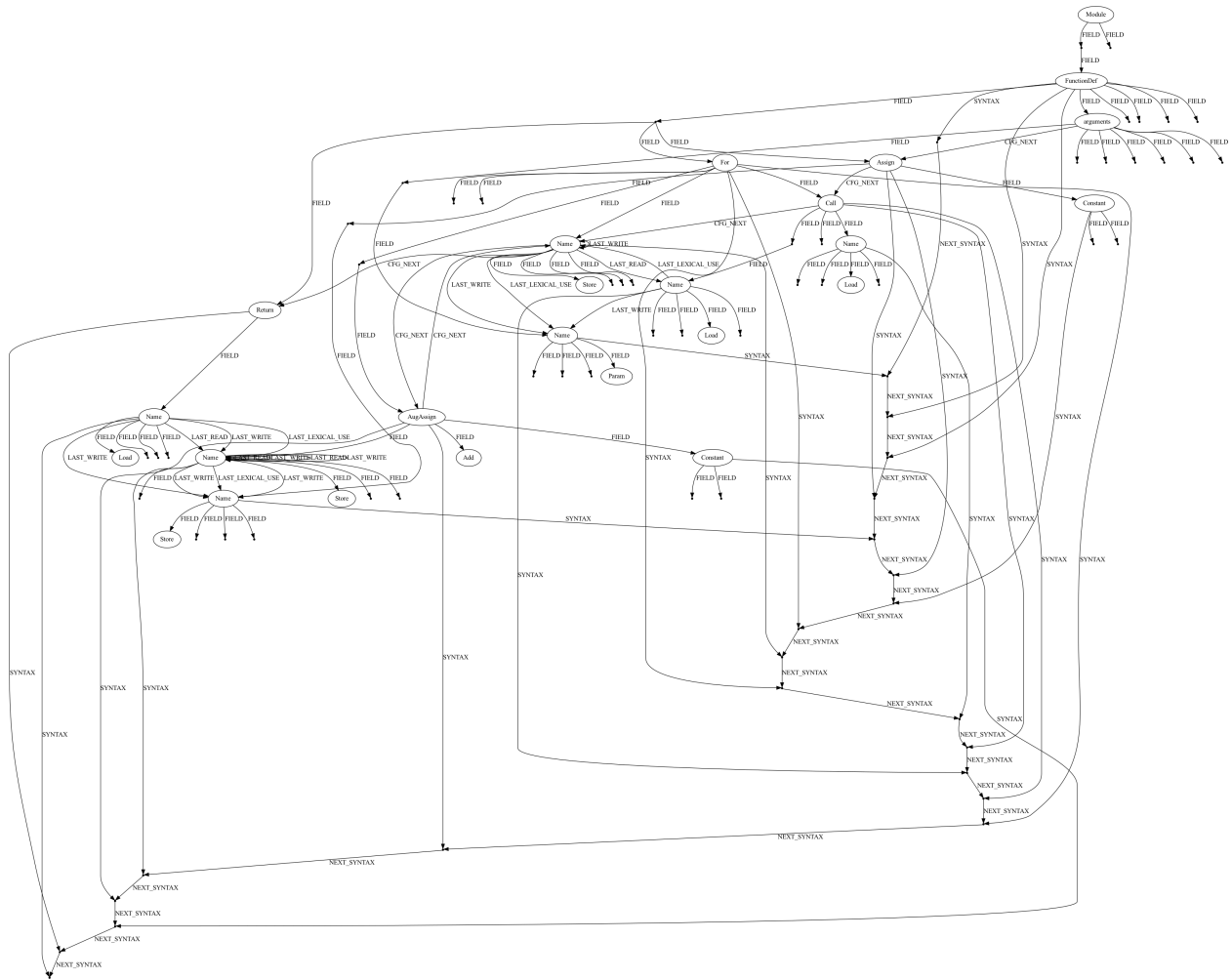
Fig. 17. Program graph for Program #1 from Table 2.



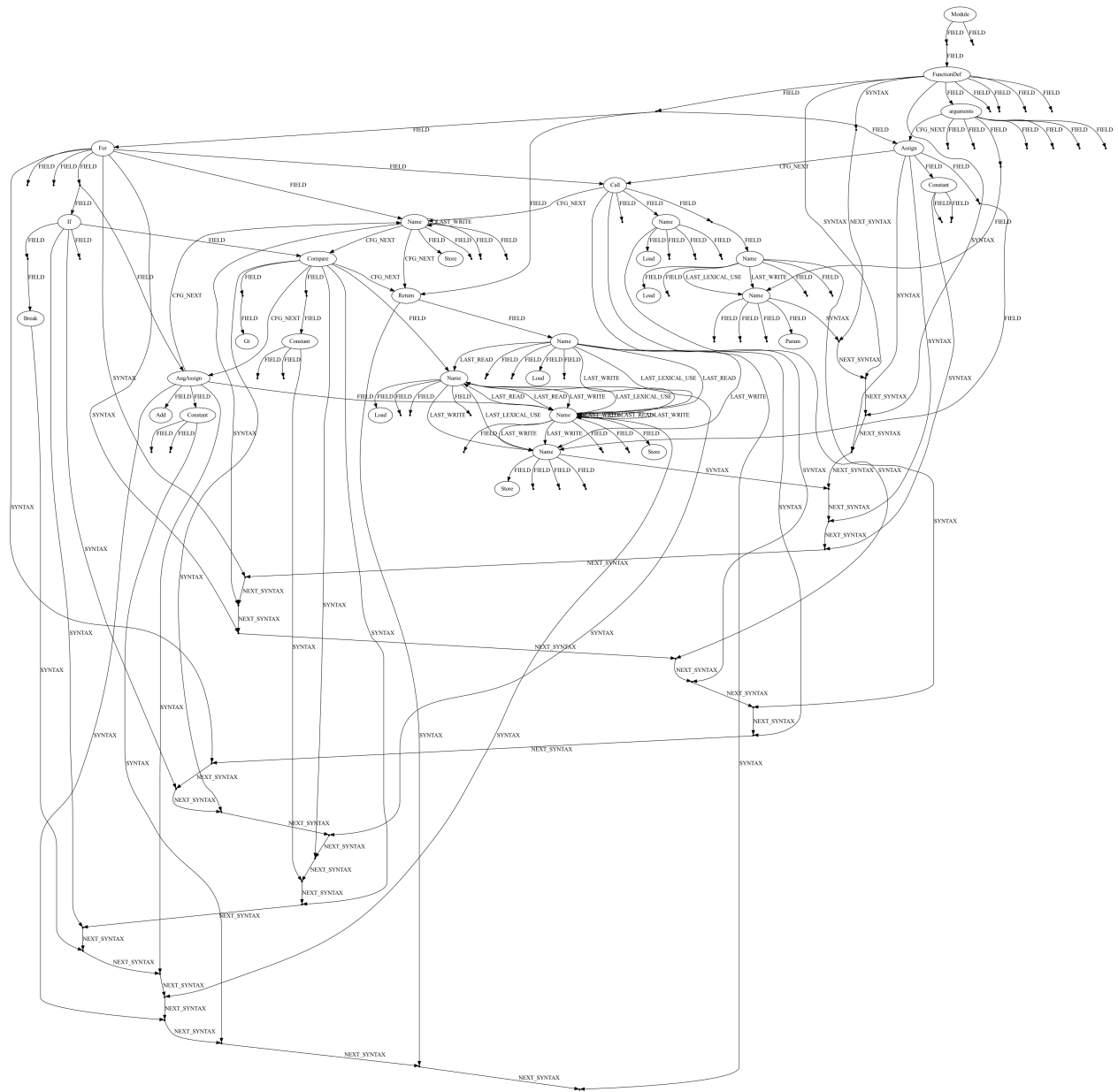
**Fig. 18.** Program graph for Program #2 from Table 2.







**Fig. 20.** Program graph for Program #4 from Table 2.



**Fig. 21.** Program graph for Program #5 from Table 2.

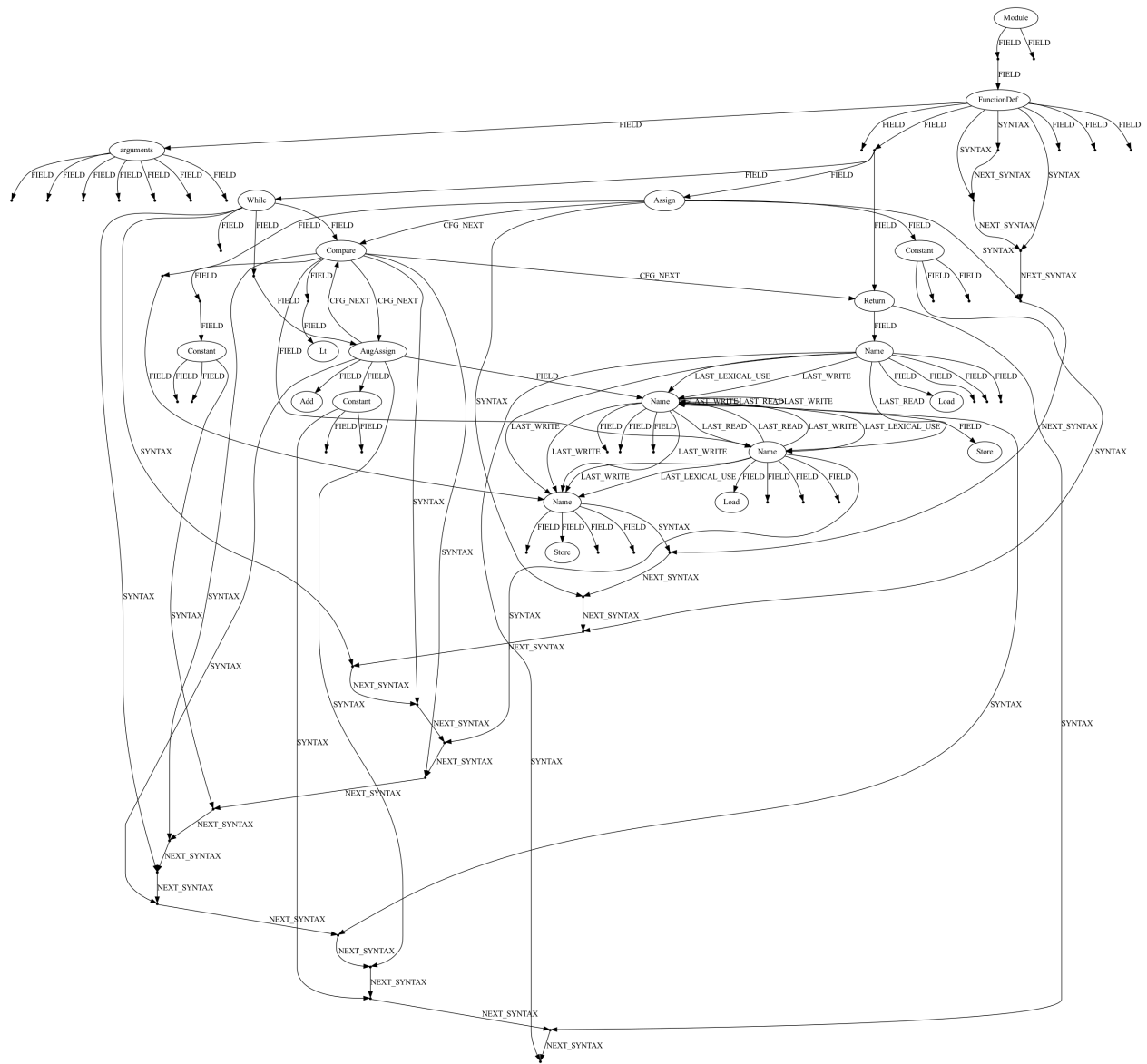
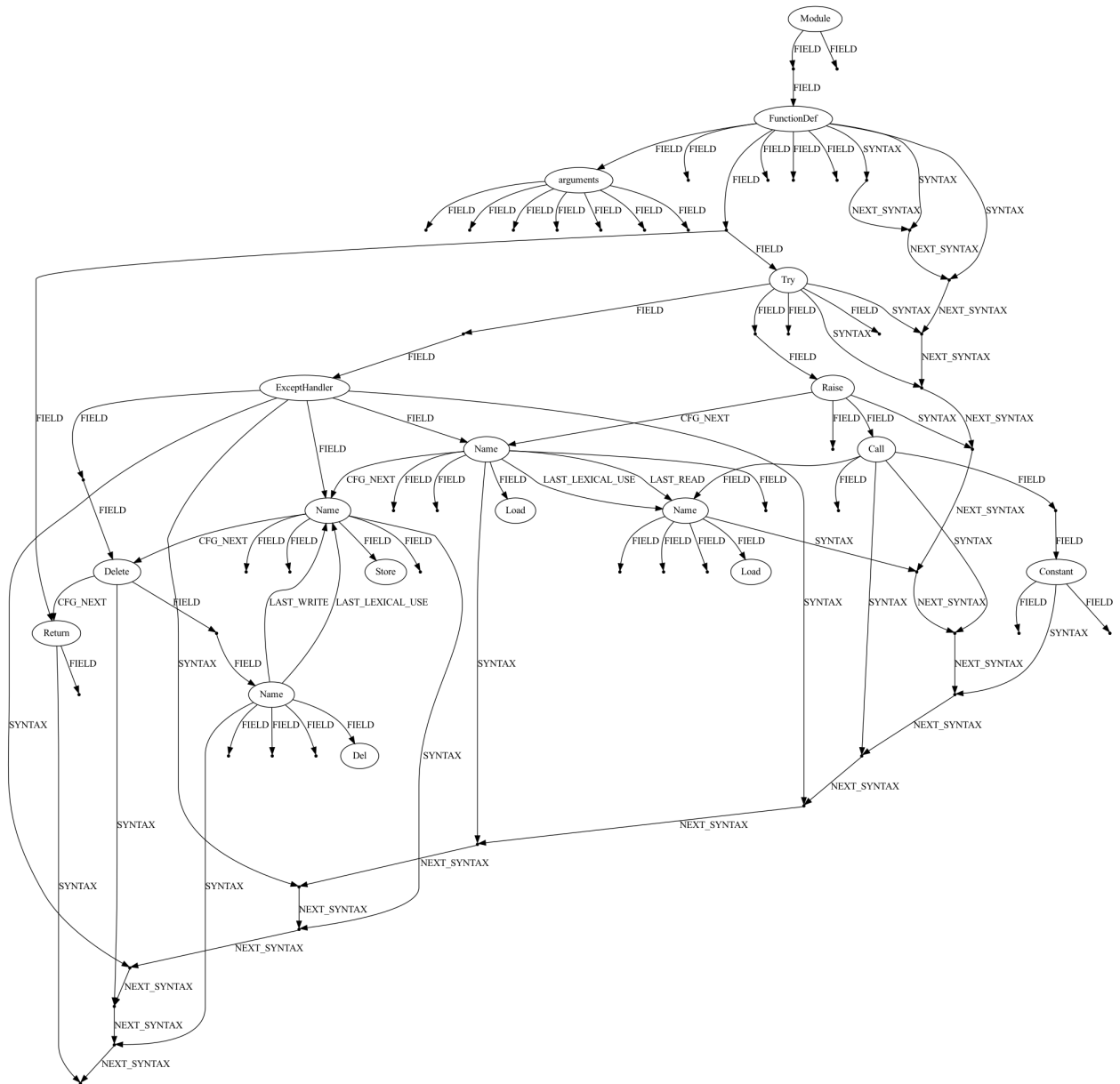
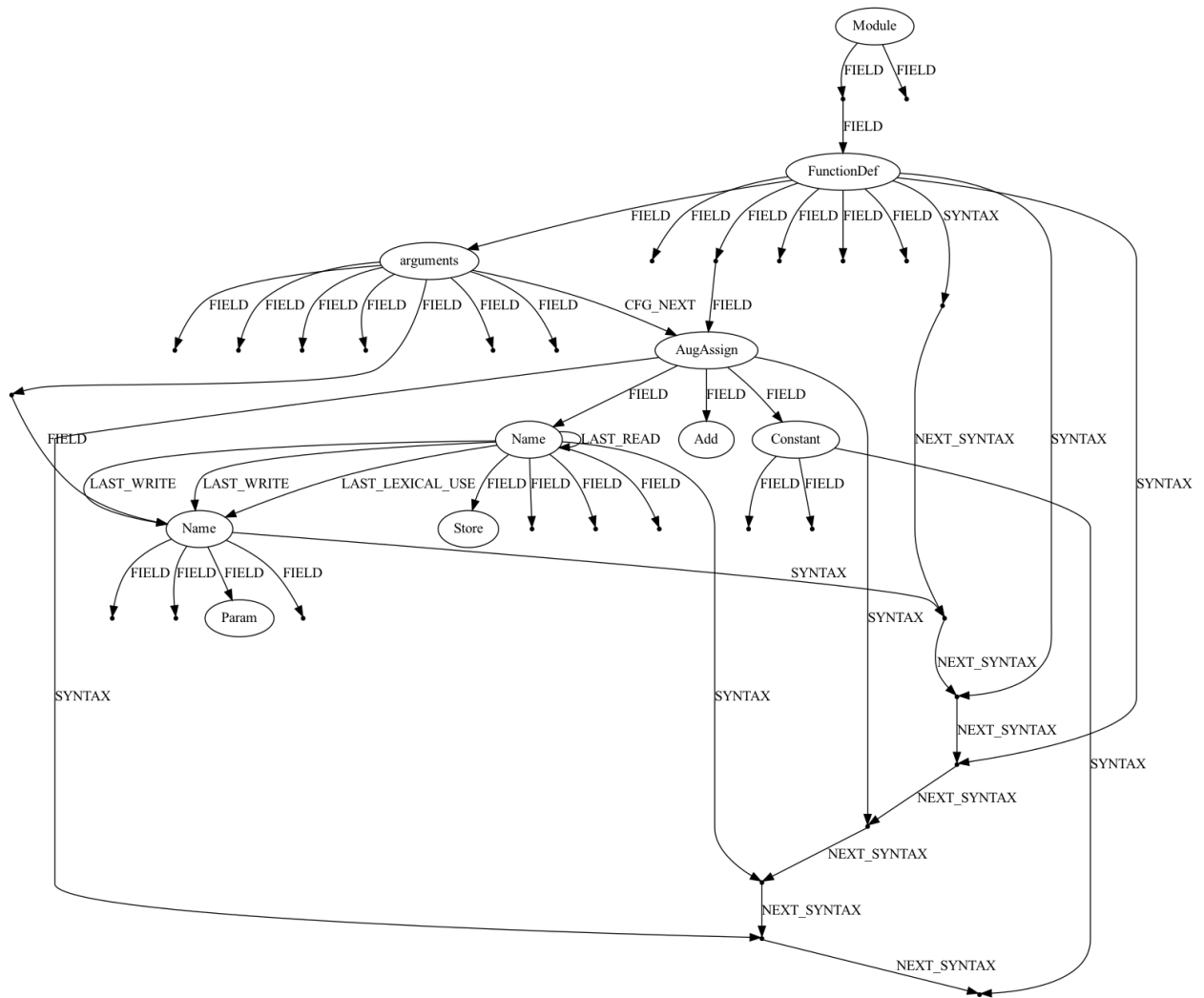


Fig. 22. Program graph for Program #6 from Table 2.

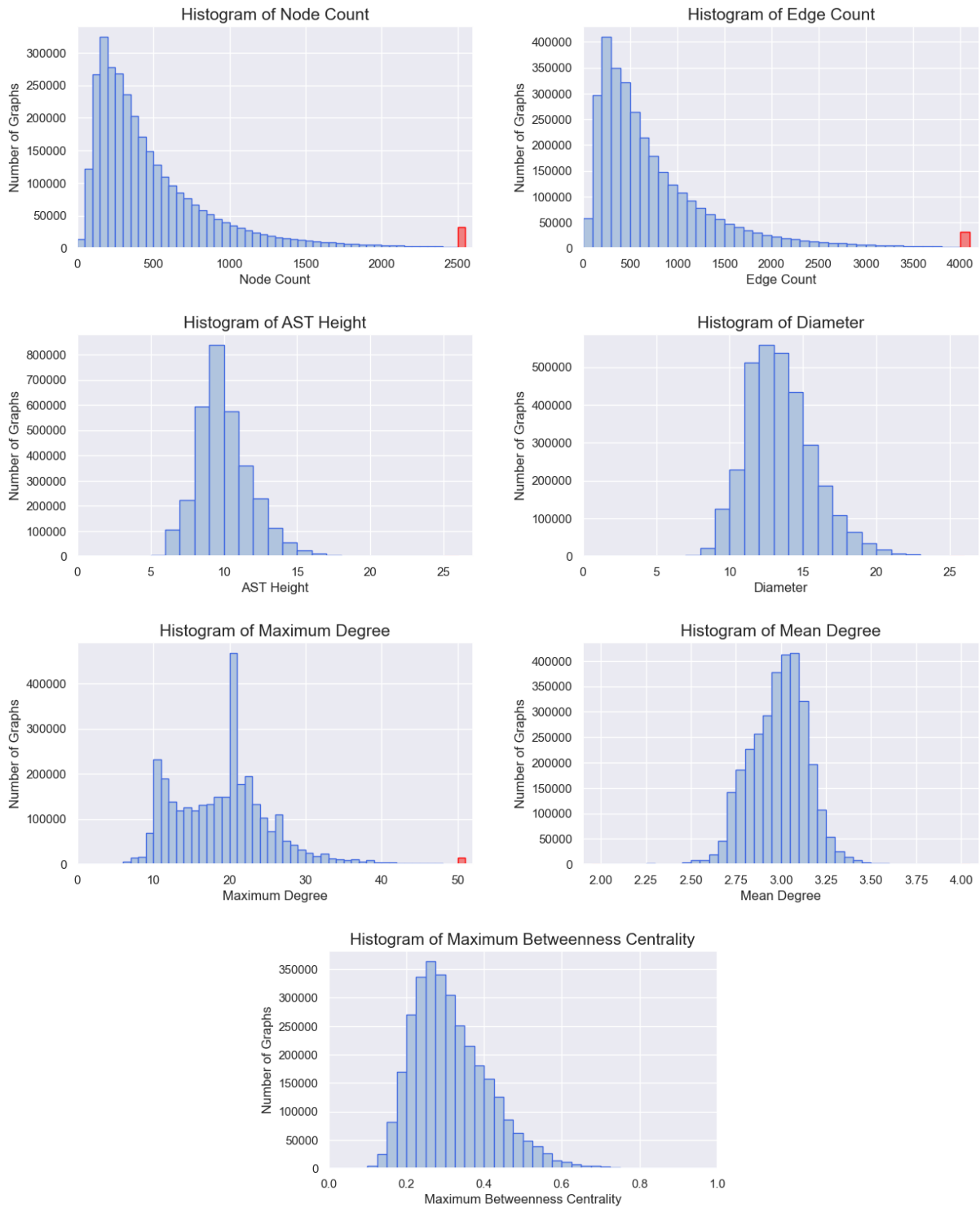


**Fig. 23.** Program graph for Program #7 from Table 2.



**Fig. 24.** Program graph for Program #8 from Table 2.

## A.2. Histograms of Program Graph Metrics



**Fig. 25.** Histograms for various metrics of program graphs from the Project CodeNet dataset. Red bars include graphs where the metric lies outside the range covered by the other bars.

## B. Appendixes for Article 2

### B.1. Architecture Details

We provide additional architectural details here beyond those provided in the paper.

In this work, all GNN models (IPA-GNN, NoExecute, NoControl, GGNN, and R-GAT) compute their final hidden state as  $h_{\text{final}} = h_{T(x), n_{\text{exit}}}$ . Here  $n_{\text{exit}}$  is the index of the program’s exit statement, and the number of neural network layers  $T(x)$  is computed as

$$T(x) = \sum_{0 \leq i \leq n_{\text{exit}}} 2^{\text{LoopNesting}(i)} + \sum_{i \in \text{Loops}(x)} 2^{\text{LoopNesting}(i)} \quad (\text{B.1})$$

$\text{LoopNesting}(i)$  denotes the number of loops with loop-body including statement  $x_i$ . And  $\text{Loops}(x)$  denotes the set of while-loop statements in  $x$ . This provides enough layers to permit message passing along each path through a program’s loop structures twice, but not enough layers for the IPA-GNN to learn to follow the ground truth trace of most programs.

In all models, the output layer consists of the computation of logits, followed by a softmax cross-entropy categorical loss term. The softmax-logits are computed according to

$$s = \text{softmax}(\text{Dense}(h_{\text{final}})). \quad (\text{B.2})$$

The cross entropy loss is then computed as

$$L = - \sum_i^K \mathbb{1}_{y=i} \log(s_i). \quad (\text{B.3})$$

This loss is then optimized using a differentiable optimizer during training.

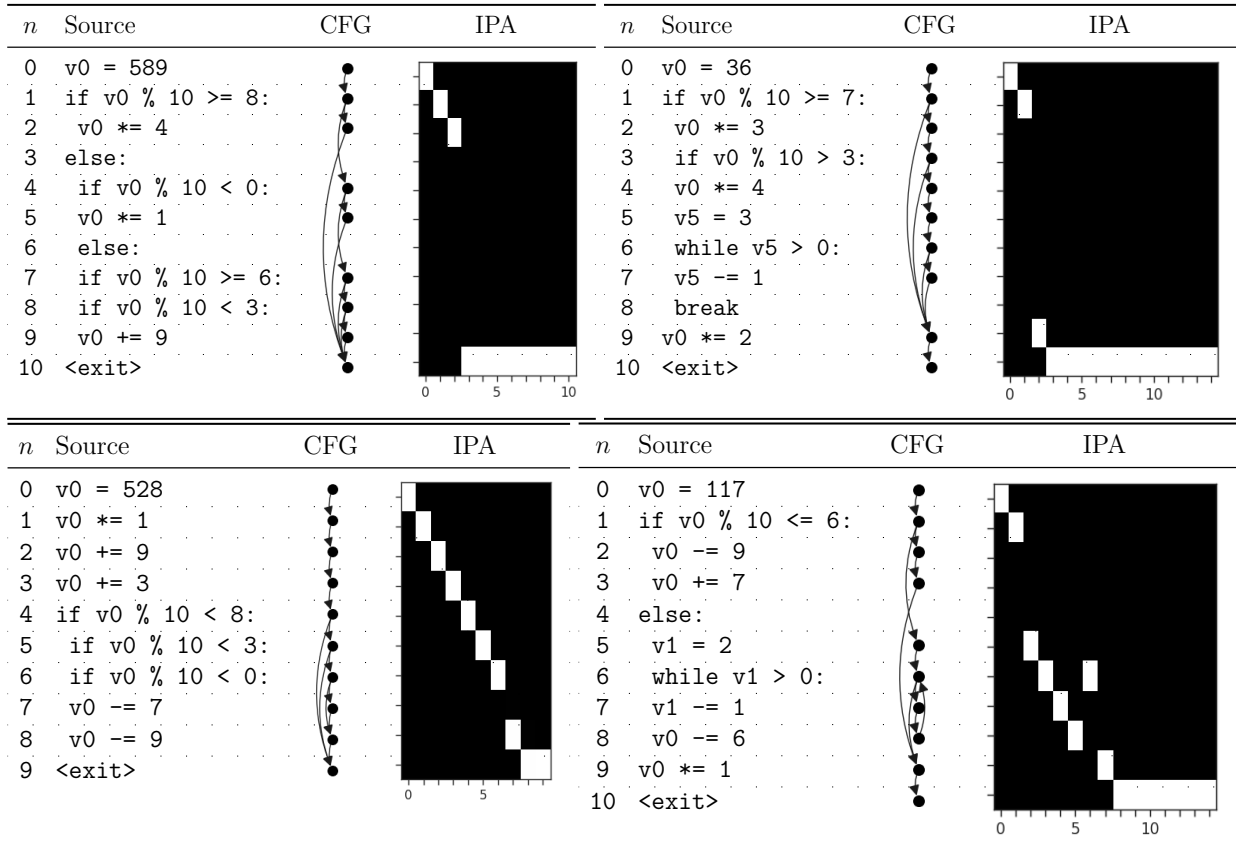
### B.2. Data Generation

For the learning to execute full and partial programs tasks, we generate a dataset from a probabilistic grammar over programs. Figure 26 provides the grammar. **If**, **IfElse**, and **Repeat** statements are translated into their Python equivalents. **Repeat** statements are represented using a while-loop and counter variable selected from `v1..v9` uniformly at random, excluding those variables already in use at the entrance to the **Repeat** statement. We generate the control-flow graphs following Bieber et al. [17].

Attention plots for randomly sampled examples from the full program execution task are shown in Figure 27. We then mask a random statement in each example and run the partial program execution IPA-GNN model over each program, showing the resulting attention plots in Figure 28. All four tasks are solved correctly in the full program execution task task, and the first three are solved correctly in the partial execution task, while the fourth partial execution task shown is solved incorrectly.

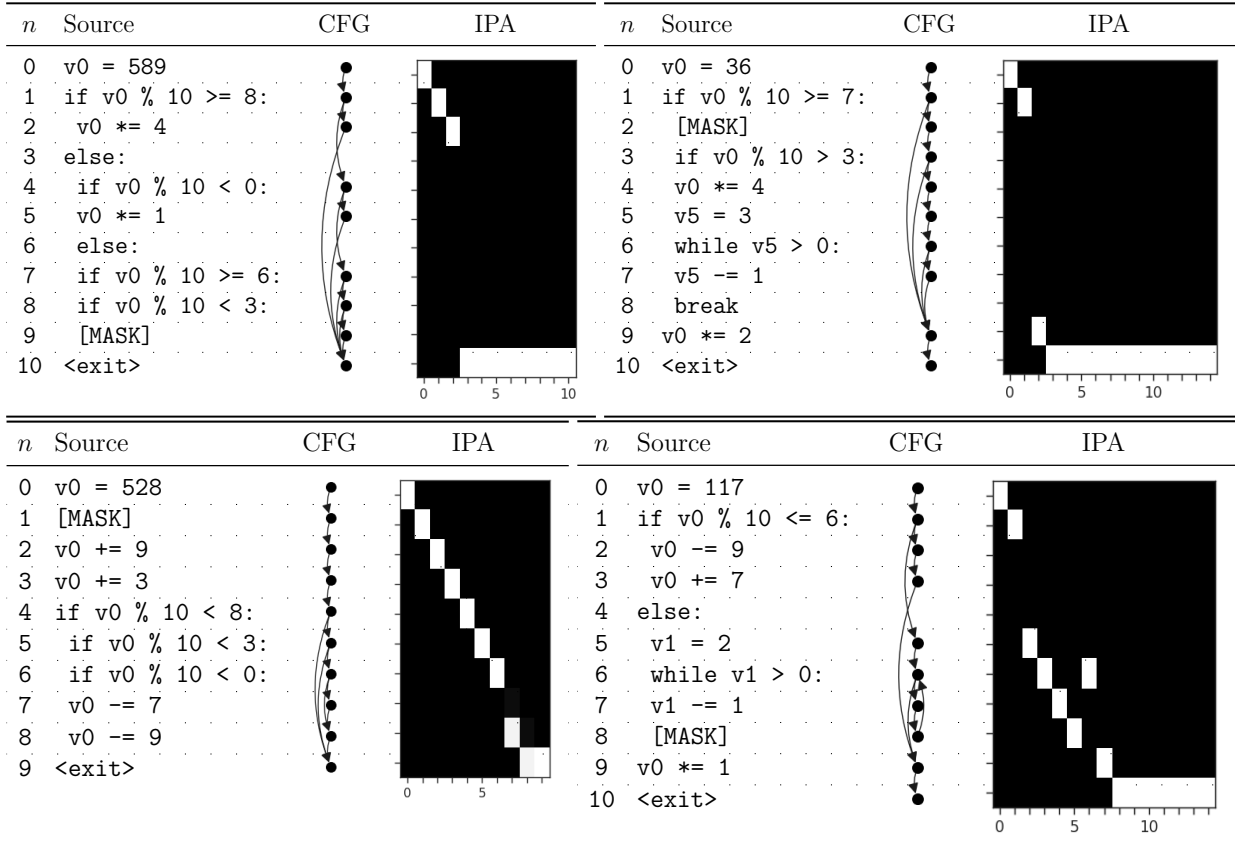
Program  $P := I B$   
 Initialization  $I := v_0 = M$   
 Block  $B := B S \mid S$   
 Statement  $S := E \mid \text{If}(C, B) \mid \text{IfElse}(C, B_1, B_2) \mid \text{Repeat}(N, B)$   
            $\mid \text{Continue} \mid \text{Break} \mid \text{Pass}$   
 Condition  $C := v_0 \bmod 10 O N$   
 Operation  $O := > \mid < \mid \geq \mid \leq$   
 Expression  $E := v_0 += N \mid v_0 -= N \mid v_0 *= N$   
           Integer  $N := 0 \mid 1 \mid 2 \mid \dots \mid 9$   
           Integer  $M := 0 \mid 1 \mid 2 \mid \dots \mid 999$

**Fig. 26.** Grammar describing the generated programs comprising the dataset in this paper.



**Fig. 27.** Intensity plots show the soft instruction pointer  $p_{t,n}$  at each step of the IPA-GNN during full program execution for four randomly sampled programs.





**Fig. 28.** The same programs as in Figure 27, with a single statement masked in each. The intensity plots show the soft instruction pointer  $p_{t,n}$  at each step of the IPA-GNN during partial program execution.

## C. Appendixes for Article 3

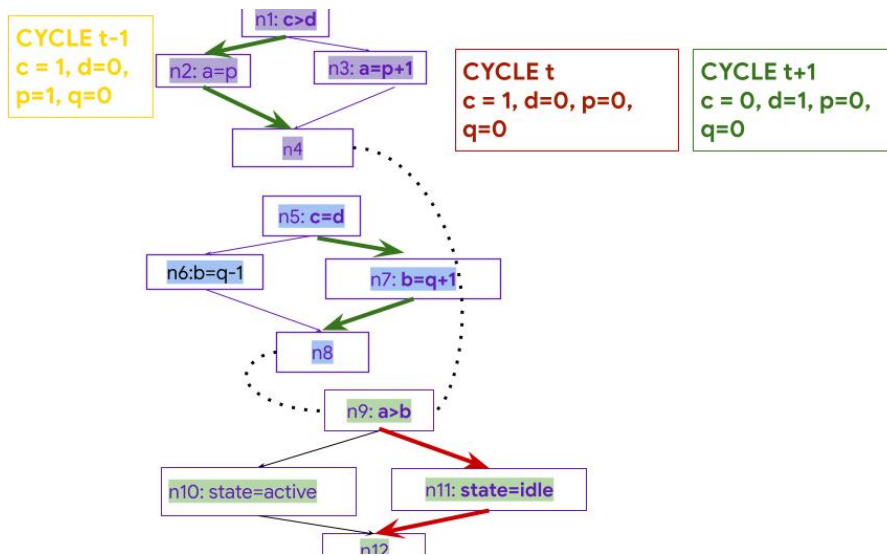
### C.1. RTL CDFGs

We show an example of RTL CDFG execution (simulation) over multiple cycles in Figure 29. In the example, at a given clock cycle  $t$ , the values of  $a$  and  $b$  from the previous clock cycle  $t-1$  will be used for evaluating the condition  $a > b$  in the green always block and the corresponding branch will be executed in that cycle. In the other two always blocks, in cycle  $t$ ,  $b$  and  $a$  will be assigned values based on values of  $c$  and  $d$  from previous cycle  $t-1$ . In the next cycle  $t+1$ ,  $a$  and  $b$  will get values of  $a$  and  $b$  from cycle  $t$ .

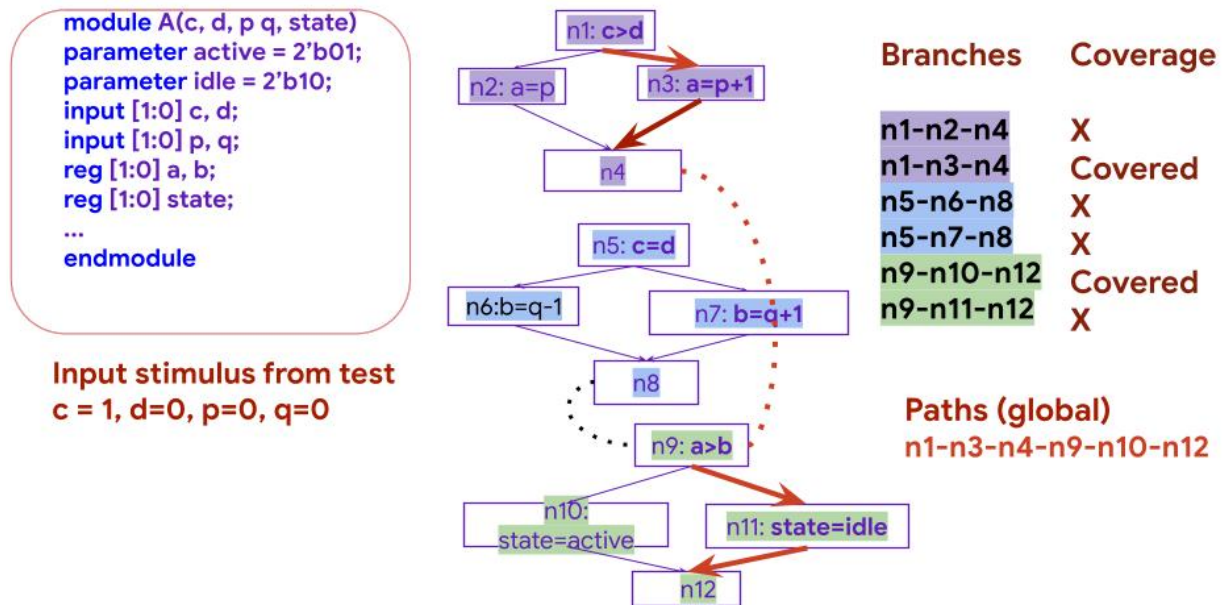
The input stimulus and the branches covered by the simulation are shown in Figure 30.

### C.2. Industrial verification flow

Figure 31 shows the context of our solution within the industrial verification flow. Figure 32 shows the Design2Vec solution inbuilt into the constrained random verification environment.



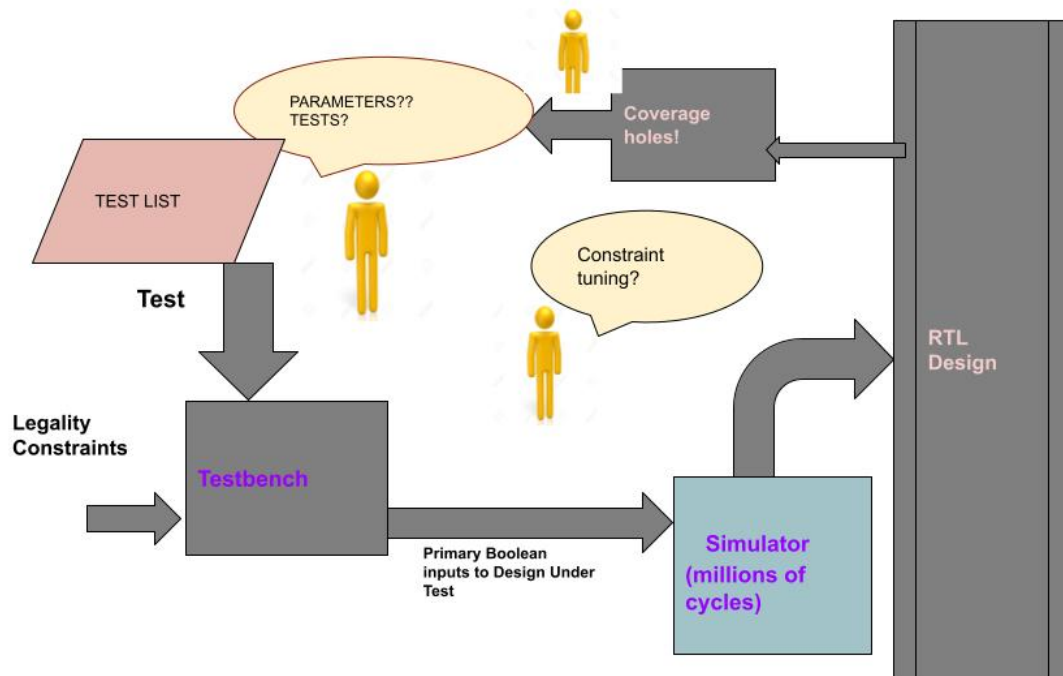
**Fig. 29.** RTL and CDFG execution (simulation) over three cycles t-1, t, t+1



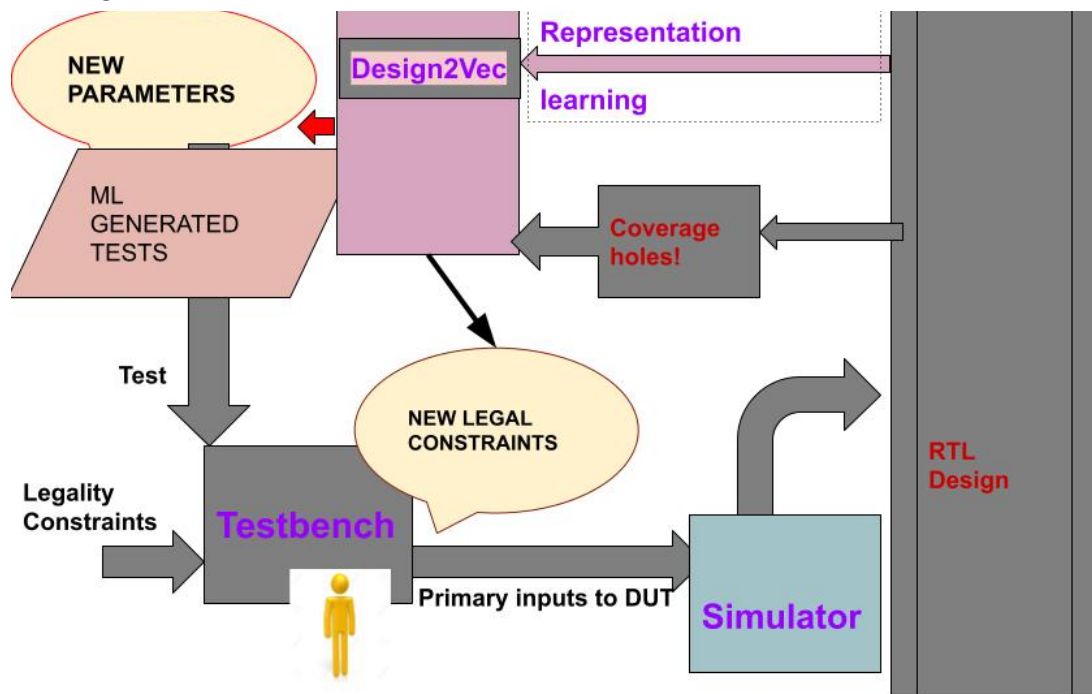
**Fig. 30.** Input stimulus and corresponding branches that are covered. Coverage is a path tracing through the CDFG.

### C.3. Sizes of designs

We show a comparison of the relative sizes of RTL CDFGs between IBEX and the TPU design in Table 17.



**Fig. 31.** Industrial verification flow with manually generated testbench, tests, constraints and coverage feedback. This flow takes multiple person-years of engineer productivity to converge



**Fig. 32.** Value proposition of Design2Vec when integrated into the loop of an industrial verification flow. It learns about the design state space and generates tests to cover different uncovered cover points (holes). It can potentially be used to generate constraints.

**Table 17.** Comparison of relative sizes of RTL CDFGs between IBEX and TPU design.

	IBEX v1	TPU BLOCK
# CDFG NODES	5500	40000
# CDFG EDGES	9300	58000
# BRANCH COVER POINTS	900	45000
# NUMBER OF GATES	10028	1088343

## C.4. Examples of generated tests

See Table 18 for examples of tests generated by Design2Vec for hard to cover points.

**Table 18.** Example tests generated by Design2Vec for hard to cover points. Multiple cover points that are local neighbors of the target point are provided as input to Design2Vec, which helps the GNN-based architecture.

COVER POINTS	TOP TEST RECOMMENDATIONS
{521, ... <b>526</b> , ..., 531}	+instr_cnt=17600 +illegal_instr_ratio=35 +hint_instr_ratio=45 ... +enable_unaligned_load_store=0 +disable_compressed_instr=1 +randomize_csr=0 +no_wfi=1
{881, ... <b>882</b> , ..., 891}	+instr_cnt=17400 +illegal_instr_ratio=25 +hint_instr_ratio=35 ... +enable_unaligned_load_store=0 +disable_compressed_instr=1 +randomize_csr=0 +no_wfi=1

## C.5. Experimental hyperparameters

The methods in Section 4 use the following hyperparameters: number of layers, learning rate, GNN embedding dimension, residual connection frequency, dropout rate, MLP embedding dimensions. We vary these parameters: number of layers  $\in \{4, 8, 16, 32\}$ , learning rate  $\in \{1e-2, 1e-3, 3e-4, 1e-4\}$ . We hold these parameters fixed: GNN embedding dimension = 16, residual connection frequency = 4, dropout rate = 0.1, MLP embedding dimensions = [256, 128, 64]. The IPA-GNN model only has one additional hyperparameter: normalization term  $\in \{1, p_{t,n}\}$ .

Number of layers describes the number of GNN layers in the Design2Vec model. GNN embedding dimension describes the embedding dimension of the intermediate and final node embeddings produced by the GNN. The residual connection frequency indicates where skip connections are added between layers of the GNN. The normalization term is used to normalize  $h_{t,n}$  after each RTL IPA-GNN layer.

Our experiments were run on commodity GPUs in a commercial data center. In total, the experiments reported here required approximately five GPU-weeks of computation.

## C.6. Architectural ablation studies

In this section, we present more detailed comparison of different hyperparameter settings and ablations of Design2Vec. Since we hypothesize that propagating information across longer distances in the graph is important, we were especially interested in the effect of residual connections [61], but we also measure the effect of label smoothing [106]. See results in Table 19. Our default Design2Vec architecture (top row in table) uses residual connections that skip back 4 layers, and does not use label smoothing. In practice, we see that none of these variants have a large effect on performance.

**Table 19.** Comparison of training and validation accuracy across variants of the Design2Vec architecture on coverage prediction on the TPU design.

ARCHITECTURE	TRAIN	VALID
Design2Vec (GCN)	92.1	90.3
Design2Vec + no residual connections	92.1	90.7
Design2Vec + label smoothing	92.1	90.3
Design2Vec + residual (every 2 layers)	92.1	90.1

We further consider architectural ablations that vary the K-hop edges added to the input CDFG in Table 22, as well as the depth of the network in Table 21. We further test each setup in settings that vary the selection of training and validation splits in Tables 20.

**Table 22.** Comparing the training and validation accuracy of the Design2Vec model using a k-hop edge augmented graph ( $k \in \{2, 4, 16\}$ ) across a variety of experimental setups.

K-HOP	HIDE	TEST PARAMS	COVER POINT HIDING METHOD	SEED	TRAIN	VALID
2		FALSE	Deterministic	—	86.88	83.48
2		FALSE	Random cover point	123	87.25	76.87
2		FALSE	Deterministic	—	65.07	69.14
2		FALSE	Random cover point	123	82.47	74.43
2		FALSE	Deterministic	—	87.62	79.08
2		FALSE	Random cover point	123	87.68	77.02
2		FALSE	Deterministic	—	82.66	78.34
2		FALSE	Random cover point	123	66.42	61.58
2		FALSE	Deterministic	—	89.81	78.92
2		FALSE	Random cover point	123	89.26	77.17
2		FALSE	Deterministic	—	84.25	79.11
2		FALSE	Random cover point	123	84.69	74.36
2		TRUE	Deterministic	—	87.00	82.30
2		TRUE	Random cover point	123	87.39	75.69

2	TRUE	Deterministic	—	83.15	83.04
2	TRUE	Random cover point	123	82.87	72.78
2	TRUE	Deterministic	—	87.74	76.93
2	TRUE	Random cover point	123	87.83	76.19
2	TRUE	Deterministic	—	82.83	75.75
2	TRUE	Random cover point	123	83.29	76.86
2	TRUE	Deterministic	—	90.04	76.16
2	TRUE	Random cover point	123	89.62	73.33
2	TRUE	Deterministic	—	84.86	76.69
2	TRUE	Random cover point	123	84.79	73.12
4	FALSE	Deterministic	—	86.86	85.30
4	FALSE	Random cover point	123	87.22	77.75
4	FALSE	Deterministic	—	65.03	69.03
4	FALSE	Random cover point	123	65.90	64.28
4	FALSE	Deterministic	—	87.63	80.32
4	FALSE	Random cover point	123	87.68	78.85
4	FALSE	Deterministic	—	82.83	79.74
4	FALSE	Random cover point	123	82.22	80.57
4	FALSE	Deterministic	—	89.75	79.98
4	FALSE	Random cover point	123	89.30	78.28
4	FALSE	Deterministic	—	84.65	80.27
4	FALSE	Random cover point	123	84.55	74.76
4	TRUE	Deterministic	—	87.03	80.18
4	TRUE	Random cover point	123	87.39	75.28
4	TRUE	Deterministic	—	82.75	83.00
4	TRUE	Random cover point	123	82.20	74.67
4	TRUE	Deterministic	—	87.94	76.56
4	TRUE	Random cover point	123	87.79	77.02
4	TRUE	Deterministic	—	83.05	76.05
4	TRUE	Random cover point	123	83.35	75.64
4	TRUE	Deterministic	—	90.01	77.08
4	TRUE	Random cover point	123	89.26	74.03
4	TRUE	Deterministic	—	81.17	77.00
4	TRUE	Random cover point	123	84.70	73.24
16	FALSE	Deterministic	—	86.97	84.39
16	FALSE	Random cover point	123	87.30	73.10
16	FALSE	Deterministic	—	64.92	68.95

16	FALSE	Random cover point	123	82.62	76.84
16	FALSE	Deterministic	—	87.67	79.56
16	FALSE	Random cover point	123	87.66	76.26
16	FALSE	Deterministic	—	64.94	65.32
16	FALSE	Random cover point	123	83.14	78.83
16	FALSE	Deterministic	—	89.79	79.80
16	FALSE	Random cover point	123	89.37	77.98
16	FALSE	Deterministic	—	84.95	77.44
16	FALSE	Random cover point	123	84.64	73.66
16	TRUE	Deterministic	—	87.04	79.44
16	TRUE	Random cover point	123	87.41	73.71
16	TRUE	Deterministic	—	64.57	66.58
16	TRUE	Random cover point	123	82.43	73.24
16	TRUE	Deterministic	—	87.85	75.15
16	TRUE	Random cover point	123	87.85	73.63
16	TRUE	Deterministic	—	83.45	77.44
16	TRUE	Random cover point	123	66.86	62.25
16	TRUE	Deterministic	—	90.16	76.61
16	TRUE	Random cover point	123	89.65	72.47
16	TRUE	Deterministic	—	84.25	77.18
16	TRUE	Random cover point	123	84.57	70.69

---

**Table 20.** Comparing the train and validation accuracy across different split selection methods: whether to hide test parameters, and whether to sample the training set via every-k sampling or uniformly random cover points.

HIDE TEST PARAMS	COVER POINT HIDING METHOD	SEED	TRAIN	VALID
FALSE	Deterministic	—	86.97	84.54
FALSE	Random cover point	123	87.41	76.86
FALSE	Deterministic	—	82.47	84.45
FALSE	Random cover point	123	80.09	75.93
FALSE	Deterministic	—	87.80	80.76
FALSE	Random cover point	123	87.73	80.51
FALSE	Deterministic	—	83.09	79.62
FALSE	Random cover point	123	83.18	78.80
FALSE	Deterministic	—	89.91	80.58
FALSE	Random cover point	123	89.53	78.06
FALSE	Deterministic	—	78.62	76.34
FALSE	Random cover point	123	83.39	75.08
TRUE	Deterministic	—	87.17	82.67
TRUE	Random cover point	123	87.51	75.92
TRUE	Deterministic	—	82.50	83.97
TRUE	Random cover point	123	82.83	78.51
TRUE	Deterministic	—	88.07	78.32
TRUE	Random cover point	123	87.96	75.98
TRUE	Deterministic	—	83.28	78.23
TRUE	Random cover point	123	83.40	78.06
TRUE	Deterministic	—	90.16	77.27
TRUE	Random cover point	123	89.78	73.84
TRUE	Deterministic	—	62.89	63.79
TRUE	Random cover point	123	83.51	73.63

**Table 21.** Comparing the train and validation accuracy on TPU while varying the numbers of GCN layers. We report the results across three seeds for each network depth.

SEED	GCN LAYERS	TRAIN	VALID
111	3	92.13	90.75
123	3	92.13	90.13
321	3	92.11	90.67
111	12	92.12	90.53
123	12	92.14	90.21
321	12	92.12	90.61
111	24	92.11	90.19
123	24	92.14	90.82
321	24	92.09	90.55



## C.7. Comparison of Design2Vec and black-box optimizer tests

**Table 23.** Comparison of Design2Vec and black-box optimizer tests for covering overall cover points in different cover point probability buckets.

PROB. BUCKET	COVER POINT ID	COVER PROB.	DESIGN2VEC# COVERS?	# D2V TESTS	VIZIER COVERS?	# VIZIER TESTS	# D2V - # VIZIER
[0.5, 1.0)	97	99.94%	Yes	1	Yes	1	0
[0.5, 1.0)	113	99.94%	Yes	2	Yes	1	1
[0.5, 1.0)	158	87.85%	Yes	1	Yes	1	0
[0.5, 1.0)	164	53.36%	Yes	3	Yes	1	2
[0.5, 1.0)	266	98.70%	Yes	1	Yes	1	0
[0.5, 1.0)	394	98.00%	Yes	1	Yes	1	0
[0.5, 1.0)	810	84.79%	Yes	1	Yes	1	0
[0.5, 1.0)	841	97.29%	Yes	1	Yes	1	0
[0.5, 1.0)	850	96.93%	Yes	1	Yes	1	0
[0.5, 1.0)	858	85.14%	Yes	1	Yes	1	0
[0.2, 0.5)	16	35.26%	Yes	1	Yes	2	-1
[0.2, 0.5)	47	24.12%	Yes	1	Yes	4	-3
[0.2, 0.5)	50	24.00%	Yes	1	Yes	4	-3
[0.2, 0.5)	185	34.43%	Yes	2	Yes	5	-3
[0.2, 0.5)	356	49.17%	Yes	2	Yes	3	-1
[0.2, 0.5)	422	49.17%	Yes	4	Yes	3	1
[0.2, 0.5)	813	48.58%	Yes	1	Yes	5	-4
[0.2, 0.5)	816	48.53%	Yes	1	Yes	5	-4
[0.2, 0.5)	817	28.83%	Yes	1	Yes	9	-8
[0.2, 0.5)	818	38.38%	Yes	1	Yes	5	-4
[0.05, 0.2)	400	9.38%	Yes	5	Yes	2	3
[0.05, 0.2)	506	7.72%	Yes	5	Yes	4	1
[0.05, 0.2)	624	10.02%	Yes	1	Yes	13	-12
[0.05, 0.2)	646	6.90%	Yes	1	Yes	45	-44
[0.05, 0.2)	649	6.72%	Yes	5	Yes	45	-40
[0.05, 0.2)	656	5.37%	Yes	1	No	—	NA
[0.05, 0.2)	667	19.16%	No	—	Yes	20	NA
[0.05, 0.2)	677	6.43%	Yes	6	Yes	36	-30
[0.05, 0.2)	700	6.43%	Yes	5	Yes	36	-31
[0.05, 0.2)	708	8.37%	Yes	2	Yes	4	-2

## D. Appendixes for Article 4

### D.1. Python Runtime Error Dataset Details

We describe in detail the construction of the Python Runtime Error dataset from the submissions in Project CodeNet [119]. The Project CodeNet dataset contains over 14 million submissions to 4,053 distinct competitive programming problems, with the submissions spanning more than 50 programming languages. We partition the problems into train, valid, and test splits at an 80:10:10 ratio. By making all submissions to the same problem part of the same split we mitigate concerns about potential data leakage from similar submissions to the same problem. We restrict our consideration to Python submissions, which account for 3,286,314 of the overall Project CodeNet submissions, with 3,119 of the problems receiving at least one submission in Python. In preparing the dataset we execute approximately 3 million problems in a sandboxed environment to collect their runtime error information, we perform two stages of filtering on the dataset, syntactic and complexity filtering, and we construct a textual representation of the input space for each problem from the problem description.

**Syntactic Filtering.** In this first phase of filtering, we remove submissions in Python 2 as well as those which fail to parse and run from our dataset. We remove 76,888 programs because they are in Python 2, 59,813 programs because they contain syntax errors that prohibit parsing, 2,011 programs that result in runtime errors during parsing, and 6 additional programs for which the `python-graphs` library fails to construct a control-flow graph [17]. A program may result in a runtime error during parsing if it contains `return`, `break`, `continue` keywords outside of an appropriate frame.

**Program Execution.** We attempt to run each submission in a sandboxed environment using the sample input provided in the Project CodeNet dataset. The environment is a custom harness running on a Google Cloud Platform (GCP) virtual environment. This allows us to collect standard out and standard error, to monitor for timeouts, and to catch and serialize any Python exceptions raised during execution. We restrict execution of each program to 1 second, marking any program exceeding this time as a timeout error. If the program encounters a Python exception, we use the name of that exception as the target class for the program. If an error type occurs only once in the dataset, we consider the target class to be `Other`. Programs not exhibiting an error or timeout are given target class “no error”.

In addition to collecting the target class, we record for each runtime error the line number at which the error occurs. We use these line numbers as the ground truth for the unsupervised error localization task considered in Section 5.3.

**Extracting Resource Descriptions by Parsing Problem Statements.** For each problem, we parse the problem statement to extract the *input description* and *input constraints*,

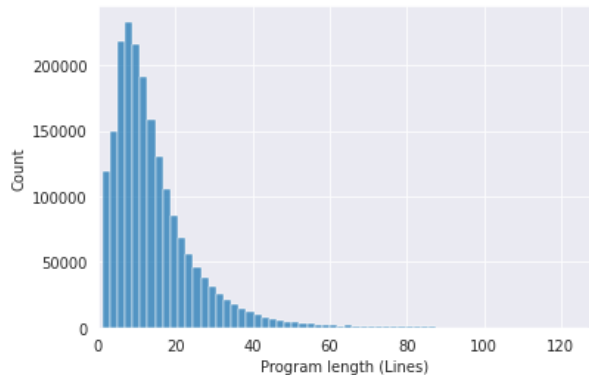
if they exist. These two sections of the problem statement together form the external resource description that accompanies that problem. The problem statements in our dataset are each written either in English or Japanese, and so we write our parser to support both languages. When one or both of these sections are present in the problem statement, we construct the external resource description for the problem by concatenating together the headers and contents of the sections that are present. For the experiments that use the resource description as a docstring, we prepend to each submission a docstring containing the resource description for problem that goes with that submission. Similarly these serve as the resource descriptions in the experiments that process resource descriptions via either cross-attention or FiLM.

**Vocabulary Construction and Complexity Filtering.** All experiments use the same vocabulary and tokenization procedure. For this, we select the standard Byte-Pair Encoding (BPE) tokenization procedure [128]. We construct the vocabulary using 1,000,000 submissions selected from the training split, along with the input space descriptions constructed for all problems in the train split. We use a vocabulary size of 30,000.

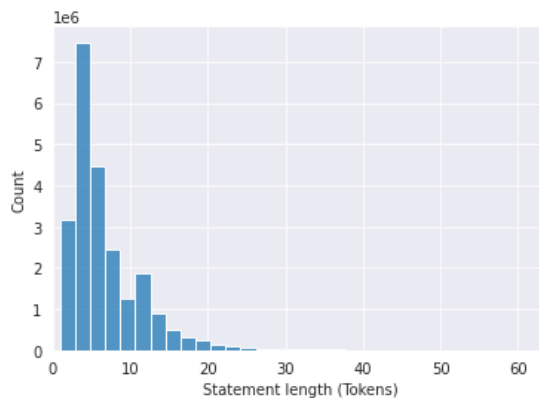
We then apply size-based filtering, further restricting the set of programs considered. First, the program length after tokenization is not to exceed 512 tokens, the number of nodes and edges in the control-flow graph are each not to exceed 128, and the step limit  $T(x)$  for a program computed in Appendix D.5 is not to exceed 174. We select these numbers to trim the long tail of exceptionally long programs, and this filtering reduces the total number of acceptable programs by less than 1%. To achieve consistent datasets comparable across all experiments, we use the longest form of each program (the program augmented with its input space information as a docstring) when computing the program sizes for size-based submission filtering. The control-flow graphs are constructed following Bieber et al. [17].

We further impose the restriction that no user-defined functions (UDFs) are called in a submission; this further reduces the number of submissions by 682,220. A user-defined function is a function defined in the submission source code, as opposed to being a built-in or imported from a third party module. Extending the IPA-GNN models to submissions with UDFs called *at most once* is trivially achieved by replacing the program’s control-flow graph with its interprocedural control-flow graph (ICFG) [108]. We leave the investigation of modeling user-defined functions to further work.

**Final Dataset Details.** After applying syntactic filtering (only keeping Python 3 programs that parse) and complexity filtering (eliminating long programs and programs that call user-defined functions), we are left with a dataset of 2,441,130 examples. The division of these examples by split and by target class is given in Table 13. Figure 33 shows the distribution of program lengths in lines represented in the completed dataset, with an average program length of 14.2 lines. The average statement length is 6.7 tokens, with full distribution shown in Figure 34.



**Fig. 33.** A histogram showing the distribution of program lengths, measured in lines, represented in the runtime errors dataset train split.



**Fig. 34.** The distribution of statement lengths, measured in tokens, in the runtime errors dataset train split.

**Data License.** The Project CodeNet [119] data that we use is available under the CDLA Permissive v2.0 license, and we release our derived dataset under this same license.

## D.2. Under-approximation of Error Labels

As described in Section 3, the ground truth error targets in our dataset are obtained by running each submission on only a single input. We do this because we only have a single input available from the online judges with which to execute the programs. As a result, the error labels we obtain under-approximate the full set of errors liable to appear at runtime. Metadata obtained from [119] indicates whether each submission encountered a runtime error on a larger set of inputs, though it does not indicate the kind or location of these errors when they are present. We use this metadata to determine the degree to which our labels are an under-approximation. We find that on the balanced test set there are 1,076 submissions (4%) which, per the metadata, encounter an error, but for which our evaluation finds no error. These are likely examples for which the program runs without error on the input we have, but for which the program fails on some additional unavailable input.

We next measure generalization from the labels in our dataset to the labels suggested by the metadata without retraining. Since these labels are only binary indicators of error presence, we use our model to perform binary classification by summing the predicted probabilities of all error kinds. The model predicts “no error” on 76.2% of the examples for which our dataset finds no error. On the examples for which the metadata indicates no error, this drops to 75.9%, and on the examples for which the metadata indicates there is an error, this rises to 80.9%. These examples, where a single input detects no error but multiple inputs detect an error, are difficult for the model to classify. We hypothesize that the types of errors

our labels omit systematically differ from those our labels include as an explanation for this 4.7% discrepancy.

### D.3. IPA-GNN Architecture

We provide a concise and precise definition of the IPA-GNN baseline architecture, following the notation of Bieber et al. [15]. The IPA-GNN operates on the statement-level control-flow graph of the input program  $x$ , maintaining per-node per-step hidden states  $h_{t,n}$  and a soft instruction pointer  $p_{t,n}$ . At each step  $t$ , each node  $x_n$  participates in execution, branch prediction, and aggregation. First, the IPA-GNN models executing the statement at each node to produce per-node state proposals

$$a_{t,n}^{(1)} = \text{RNN}(h_{t-1,n}, \text{Embed}(x_n)). \quad (\text{D.1})$$

Then, the model uses these to inform soft branch decisions at every control flow juncture, given as

$$b_{t,n,n_1}, b_{t,n,n_2} = \text{softmax}(\text{Dense}(a_{t,n}^{(1)})), \quad (\text{D.2})$$

where  $\{n_1, n_2\} = N_{\text{out}}(x_n)$  when  $|N_{\text{out}}(x_n)| = 2$ . When  $|N_{\text{out}}(x_n)| = 1$  we have  $b_{t,n,n'} = 1$  for  $n' \in N_{\text{out}}(x_n)$  indicating straight-line code. For all other  $n, n'$ ,  $b_{t,n,n'} = 0$ . The state proposals and branch decisions in turn feed into the computation of the new hidden states

$$h_{t,n} = \sum_{n' \in N_{\text{in}}(n)} p_{t-1,n'} \cdot b_{t,n',n} \cdot a_{t,n}^{(1)} \quad (\text{D.3})$$

and new instruction pointer values

$$p_{t,n} = \sum_{n' \in N_{\text{in}}(n)} p_{t-1,n'} \cdot b_{t,n',n}. \quad (\text{D.4})$$

The hidden state at final time step  $T(x)$  at the program’s exit node  $n_{\text{exit}}$ , given by  $h_{T(x),n_{\text{exit}}}$  are used for downstream predictions.

### D.4. Input Modulation

In Section 4.2, we consider both *cross-attention* [88] and *Feature-wise Linear Modulation* (FiLM) [116] as options for the Modulate function. We provide the definitions of these

operations here. First, cross-attention modules the input as:

$$\text{MultiHead}(\text{Embed}(x_n), d(x), h_{t-1,n}) = \text{Concat}(\text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O, \text{Embed}(x_n)) \quad (\text{D.5})$$

$$\text{where head}_i = \text{softmax}\left(\frac{QK'}{\sqrt{d_k}}\right)V \quad (\text{D.6})$$

$$Q = W_i^Q \text{Concat}(\text{Embed}(x_n), h_{t-1,n}) \quad (\text{D.7})$$

$$K = W_i^K d(x) \quad (\text{D.8})$$

$$V = W_i^V d(x) \quad (\text{D.9})$$

Here,  $W^O \in R^{hd_v \times d_{\text{model}}}$ ,  $W_i^Q \in R^{d_k \times (d_{\text{model}} + d_{\text{Embed}(x_n)})}$ ,  $W_i^K \in R^{d_k \times d_{d(x)}}$ , and  $W_i^V \in R^{d_v \times d_{d(x)}}$  are learnable parameters. Similarly, for FiLM we modulate the input with the resource description as follows:

$$\text{FiLM}(\text{Embed}(x_n), d(x), h_{t-1,n}) = \text{Concat}(\beta \cdot d(x) + \gamma, \text{Embed}(x_n)) \quad (\text{D.10})$$

$$\text{where } \beta = \sigma(W_\beta \text{Concat}(x_n, h_{t-1,n}) + b_\beta), \quad (\text{D.11})$$

$$\gamma = \sigma(W_\gamma \text{Concat}(x_n, h_{t-1,n}) + b_\gamma), \quad (\text{D.12})$$

where  $W_\beta \in R^{d_{d(x)} \times (d_{\text{model}} + d_{\text{Embed}(x_n)})}$ , and  $W_\gamma \in R^{d_{d(x)} \times (d_{\text{model}} + d_{\text{Embed}(x_n)})}$  are learnable parameters.

## D.5. Training Details

**Hyperparameter selection.** We select hyperparameters by performing a random search independently for each model architecture. The hyperparameters considered by the search are listed in Table 25. All architectures use a Transformer encoder, and the Transformer sizes considered in the search are listed in Table 25 and defined further in Table 24.

**Table 24.** Hyperparameter settings for each of the three Transformer sizes.

HYPERPARAMETER	T-128	T-256	T-512
EMBEDDING DIMENSION	128	256	512
NUMBER OF HEADS	4	4	8
NUMBER OF LAYERS	2	2	6
QKV DIMENSION	128	256	512
MLP DIMENSION	512	1024	2048

**Step limit.** For the IPA-GNN and Exception IPA-GNN, the function  $T(x)$  represents the number of execution steps modeled for program  $x$ . We reuse the definition of  $T(x)$  from Bieber et al. [15] as closely as possible, only modifying it to accept arbitrary Python programs,

rather than being restricted to the subset of Python features considered in the dataset of the earlier work.

**Parameter counts.** We provide in Table 26 the total number of parameters in each model, for the best performing hyperparameters in each model class. For all model classes, the maximum number of parameters considered is roughly equal (approximately 8.8 million).

**Compute usage and model speeds.** All models are trained on Google Cloud Platform using TPUv2 accelerators. We use approximately one TPU-week of compute in training each IPA-GNN model. At inference time, IPA-GNN compute is proportional to the number of model steps, which is up to 174 for examples in our dataset. We measure the average inference time on the test set: 0.43 seconds per batch of 32. We also measure the training speed in seconds per step for each method, which we report in Table 26. We observe that the IPA-GNN train times are slower than those of the generic models, a drawback of the IPA-GNN model family in its current implementations. That said, we also note that the IPA-GNN models do not benefit from the same optimizations as basic implementations of the well known general purpose models (GGNN, Transformer, and LSTM), and with further optimizations the IPA-GNN performance can be improved.

## D.6. Metric Variances

Under the experimental conditions of Section 5.1, we perform three additional training runs to calculate the variance for each metric for each baseline model, and for the Exception IPA-GNN model using the docstring strategy for processing resource descriptions. For these new training runs, we use the hyperparameters obtained from model selection. We vary the random seed between runs (0, 1, 2), thereby changing the initialization and dropout behavior of each model across runs. We report the results in Table 27;  $\pm$  values are one standard deviation.

**Table 25.** Hyperparameters considered for random search during model selection.

HYPERPARAMETER	VALUE(S) CONSIDERED	ARCHITECTURE(S)
OPTIMIZER	{SGD}	ALL
BATCH SIZE	{32}	ALL
LEARNING RATE	{0.01, 0.03, 0.1, 0.3}	LSTM, TRANSFORMERS, IPA-GNNs
LEARNING RATE	{0.001, 0.003, 0.01, 0.03}	GGNN
GRADIENT CLIPPING	{0, 0.5, 1, 2}	ALL
HIDDEN SIZE	{64, 128, 256}	ALL
RNN LAYERS	{2}	LSTM, IPA-GNNs
GNN LAYERS	{8, 16, 24}	GGNN
SPAN ENCODER POOLING	{FIRST, MEAN, MAX, SUM}	ALL
CROSS-ATTENTION NUMBER OF HEADS	{1, 2}	IPA-GNNs WITH CROSS-ATTENTION
MIL POOLING	{MAX, MEAN, LOGSUMEXP}	MIL TRANSFORMERS
TRANSFORMER DROPOUT RATE	{0, 0.1}	ALL
TRANSFORMER ATTENTION DROPOUT RATE	{0, 0.1}	ALL
TRANSFORMER SIZE	{T-128, T-256, T-512}	ALL

**Table 26.** The parameter count, training latency (sec/step), and inference latency (sec/batch) for the best performing instance of each model variant. Training and inference latencies use batch size 32.

MODEL	PARAMETER COUNT	TRAIN LATENCY	INFERENCE LATENCY
GGNN	4,831,903	0.055	0.040
TRANSFORMER	8,578,975	0.054	0.051
LSTM	4,361,823	0.058	0.057
IPA-GNN	4,368,161	0.727	0.294
E. IPA-GNN	8,856,099	1.167	0.435

**Table 27.** Mean and standard deviation for each metric is calculated from three training runs per model, using the hyperparameters selected via model selection.

METHOD	R.D.?	Acc.	W. F1	E. F1
GGNN		61.98 ± 1.24	56.62 ± 2.96	41.24 ± 5.51
TRANSFORMER		63.82 ± 0.62	59.86 ± 0.52	46.75 ± 0.93
LSTM		66.43 ± 0.60	62.33 ± 1.12	50.10 ± 1.94
EXCEPTION IPA-GNN	✓	71.44 ± 0.15	70.78 ± 0.07	63.54 ± 0.03

## D.7. Static Analysis Baseline

Our work builds towards a developer tool that predicts runtime errors in programs without running the program, treating the task as static analysis. Existing static analysis tools already inspect Python source code for possible issues, though they are not generally designed with runtime error prediction in mind. Among the most popular such tools are the linters `pylint` and `flake8`, the type analyzer `pytype`, and the formatter `black`. We elect to compare against `pylint` as it is the most common of these tools and hence most representative of a modern developer workflow. Additionally, a formatter is not well suited to the task of predicting errors, and type analysis benefits from type annotations which are rarely utilized in competition code. In our comparison of machine learning methods against `pylint` (Section 5), we build a runtime error classifier based on `pylint`’s output. For each kind of error or warning that `pylint` can detect, we determine whether it is indicative a runtime error class. For example, `pylint`’s error `no-member` (E1101) indicates the `AttributeError` runtime error. The `pylint` baseline predicts a runtime error class whenever `pylint`’s errors or warnings indicate that error class, and “no error” otherwise. Table 28 shows the mapping from `pylint` findings to runtime error.

Only eleven of twenty-six the runtime error classes (those listed in Table 28, and “no error”) can be predicted by this baseline. Additionally, the presence of a `pylint` finding that corresponds to an error does not guarantee the error would actually be present when running the program; for example an undefined variable may appear on an unused control-flow path, benign at runtime. The results of this baseline are reported in Section 5.



## D.8. Localization by Modeling Exception Handling

For programs that lack try/except frames, we compute the localization predictions of the Exception IPA-GNN model by summing, separately for each node, the contributions from that node to the exception node across all time steps. This gives an estimate of exception provenance as

$$p(\text{error at statement } n) = \sum_t p_{t,n} \cdot b_{t,n,n_{\text{error}}}. \quad (\text{D.13})$$

For programs with a try/except frame, however, we must trace the exception back to the statement that originally raised it. To do this, we keep track of the exception provenance at each node at each time step; when an exception raises, it becomes the exception provenance at the statement that it raises to, and when a statement with non-zero exception provenance executes without raising, it propagates its exception provenance to the next node unchanged.

Define  $v_{t,n,n'}$  as the amount of “exception probability mass” at time step  $t$  at node  $n'$  attributable to an exception starting at node  $n$ . Then we write

$$v_{t,n,n'} = \sum_{k \in N_{\text{in}}(n')} v_{t-1,n,k} \cdot b_{t,k,n'} \cdot p_{t,k} + (1 - \sum v_{t-1,;,n}) \cdot b_{t,n,n'} \cdot p_{t,n} \cdot \mathbb{1}\{n' = r(n)\}. \quad (\text{D.14})$$

The first term propagates exception provenance across normal non-raising execution, while the second term introduces exception provenance when an exception is raised. We then write precisely

$$p(\text{error at statement } n) = v_{T(x),n,n_{\text{error}}}, \quad (\text{D.15})$$

allowing the Exception IPA-GNN to make localization predictions for any program in the dataset.

**Table 28.** The pylint baseline for runtime error prediction predicts the error class shown when it encounters any of the corresponding pylint findings. Many of pylint’s 235 finding types do not indicate runtime errors. This table shows the mapping used by the pylint baseline.

ERROR CLASS	PYLINT FINDING	ERROR CLASS	PYLINT FINDING
AssertionError	bad-thread-instantiation (W1506)	TypeError (cont.)	invalid-index-returned (E0305)
AttributeError	misplaced-format-function (E0119)		invalid-repr-returned (E0306)
	no-member (E1101)		invalid-str-returned (E0307)
	not-context-manager (E1129)		invalid-bytes-returned (E0308)
	missing-format-attribute (W1306)		invalid-hash-returned (E0309)
	not-async-context-manager (E1701)		invalid-length-hint-returned (E0310)
ImportError	import-error (E0401)		invalid-format-returned (E0311)
	relative-beyond-top-level (E0402)		invalid-getnewargs-returned (E0312)
	no-name-in-module (E0611)		invalid-getnewargs-ex-returned (E0313)
IndexError	potential-index-error (E0643)		unpacking-non-sequence (E0633)
	too-few-format-args (E1306)		raising-bad-type (E0702)
	invalid-format-index (W1307)		bad-exception-cause (E0705)
KeyError	missing-format-argument-key (W1303)		raising-non-exception (E0710)
	missing-format-string-key (E1304)		notimplemented-raised (E0711)
NameError	used-before-assignment (E0601)		catching-non-exception (E0712)
	undefined-variable (E0602)		bad-super-call (E1003)
RuntimeError	misplaced-bare-raise (E0704)		not-callable (E1102)
	modified-iterating-dict (E4702)		isinstance-...-not-valid-type (W1116)
	modified-iterating-set (E4703)		no-value-for-parameter (E1120)
SyntaxError	syntax-error (E0001)	too-many-function-args (E1121)	
	return-outside-function (E0104)	unexpected-keyword-arg (E1123)	
	yield-outside-function (E0105)	redundant-keyword-arg (E1124)	
	duplicate-argument-name (E0108)	missing-kwargs (E1125)	
	too-many-star-expressions (E0112)	invalid-sequence-index (E1126)	
	invalid-star-assignment-target (E0113)	invalid-slice-index (E1127)	
	star-needs-assignment-target (E0114)	invalid-unary-operand-type (E1130)	
	nonlocal-and-global (E0115)	unsupported-binary-operation (E1131)	
	nonlocal-without-binding (E0117)	repeated-keyword (E1132)	
	used-prior-global-declaration (E0118)	not-an-iterable (E1133)	
	await-outside-async (E1142)	unsupported-membership-test (E1135)	
	yield-inside-async-function (E1700)	unsubscriptable-object (E1136)	
	invalid-unicode-codec (E2501)	unsupported-assignment-operation (E1137)	
	bidirectional-unicode (E2502)	unsupported-delete-operation (E1138)	
	TypeError	abstract-class-instantiated (E0110)	dict-iter-missing-items (E1141)
		bad-reversed-sequence (E0111)	unhashable-member (E1143)
invalid-slots-object (E0236)		bad-format-character (E1300)	
invalid-slots (E0238)		mixed-format-string (E1302)	
inherit-non-class (E0239)		format-needs-mapping (E1303)	
inconsistent-mro (E0240)		bad-string-format-type (E1307)	
duplicate-bases (E0241)		invalid-envvar-value (E1507)	
invalid-enum-extension (E0244)		invalid-envvar-default (W1508)	
invalid-length-returned (E0303)			
invalid-bool-returned (E0304)			
		ValueError	return-in-init (E0101)
			class-variable-slots-conflict (E0242)
			unbalanced-tuple-unpacking (W0632)
			bad-format-string (W1302)
			format-combined-specification (W1305)
			bad-open-mode (W1501)

## D.9. Localization by Multiple Instance Learning

The Local Transformer and Global Transformer models each compute per-statement node embeddings  $\text{Embed}(x_n)$  given by Equation 4.1. In the multiple instance learning setting,

these are transformed into unnormalized per-statement class predictions

$$\phi(\text{class} = k, \text{lineno} = l) = \text{Dense}(\text{Embed}(x_n)). \tag{D.16}$$

We consider three strategies for aggregating these per-statement predictions into an overall prediction for the task. Under the *logsumexp* strategy, we treat  $\phi$  as logits and write

$$\log p(\text{class} = k) \propto \log \left( \sum_l \exp \phi(k, l) \right), \tag{D.17}$$

$$\log p(\text{lineno} = l) \propto \log \left( \sum_{k \in K} \exp \phi(k, l) \right) \tag{D.18}$$

where  $K$  is the set of error classes.

The *max* and *mean* strategies meanwhile follow Wang et al. [150] in asserting

$$p(\text{class} = k \mid \text{lineno} = l) = \text{softmax}(\phi(k, l)), \tag{D.19}$$

compute the location probabilities as

$$p(\text{lineno} = l) \propto \sum_{k \in K} p(\text{class} = k \mid \text{lineno} = l), \tag{D.20}$$

and compute the outputs as

$$\log p(\text{class} = k) \propto \log \max_l p(\text{class} = k \mid \text{lineno} = l), \text{ and} \tag{D.21}$$

$$\log p(\text{class} = k) \propto \log \frac{1}{L} \sum_l p(\text{class} = k \mid \text{lineno} = l) \tag{D.22}$$

respectively, where  $L$  denotes the number of statements in  $x$ . As with all methods considered, the MIL models are trained to minimize the cross-entropy loss in target class prediction, but these methods still allow reading off predictions of  $p(\text{lineno})$ .

## D.10. Broader Impact

Our work builds toward improvements to developer tools, suggesting the possibility of future tools that predict runtime errors in code even when that code lacks unit tests. However, the false positive rate under the current best models present a challenge. A developer tool built using these models may present the developer with incorrect predictions. This could cause the developer to make mistakes, or to lose trust in the tooling, lowering productivity in the short term and making it harder to win back trust in the long term when tools are built upon higher quality models with fewer errors. We therefore recommend that tool developers use a combination of cautious judgement and data driven evaluations when deciding when to implement features that rely on models like the ones we present.

## D.11. Example Visualizations

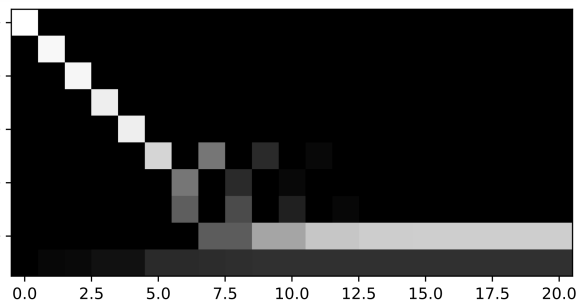
We sample three examples at random from the Python Runtime Error dataset validation split, and visualize them here. As in Figure 16, we show instruction pointer heatmaps for both the BASELINE and DOCSTRING Exception IPA-GNN model variants.

In the heatmaps, the x-axis represents time steps and the y-axis represents nodes, with the last two rows representing the exit node  $n_{\text{exit}}$  and the exception node  $n_{\text{error}}$ . Note that for loop statements are associated with two spans in the statement-level control-flow graph, one for construction of the loop iterator, and a second for assignment to the loop variable. Hence we list two indexes for each for loop statement in these figures, and report the total error contribution for the line.

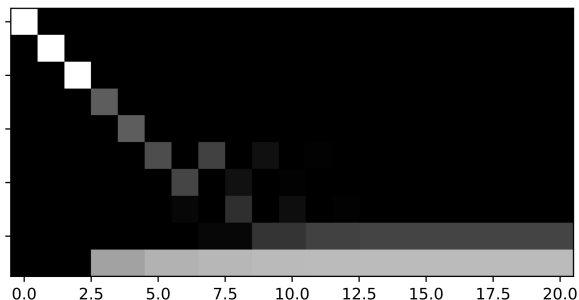
**Fig. 35.** The target error kind is INDEXERROR, occurring on line 5 ( $n = 5$ ). BASELINE incorrectly predicts NO ERROR with confidence 0.808. DOCSTRING correctly predicts INDEXERROR with confidence 0.693, but localizes to line 3 ( $n = 2$ ). Both BASELINE and DOCSTRING instruction pointer values start out sharp and become diffuse when reaching the for-loop. The BASELINE instruction pointer value ends with most probability mass at  $n_{\text{exit}}$ . The DOCSTRING instruction pointer value has a small amount of probability mass reaching  $n_{\text{exit}}$ , with most probability mass ending at  $n_{\text{error}}$ .

```
STDIN DESCRIPTION      Input: Input is given from Standard Input in the following
                        format: N a_1 a_2 ... a_N
                        Constraints: All values in input are integers. 1 <= N ,
                        a_i <= 100
```

$n$	SOURCE	BASELINE	R.D.
0	<code>N = int(input())</code>	2.9	0.2
1	<code>A = list(map(int, input().split()))</code>	0.8	0.0
2	<code>res = 0</code>	3.0	<b>63.3</b>
3,4	<code>for i in range(1, len(A)+1, 2):</code>	9.8	6.3
5	<code>res += A[i] % 2</code>	0.3	0.1
6	<code>print(res)</code>	0.2	2.2



BASELINE

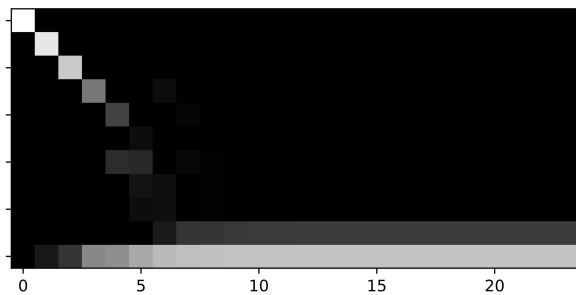


RESOURCE DESCRIPTION

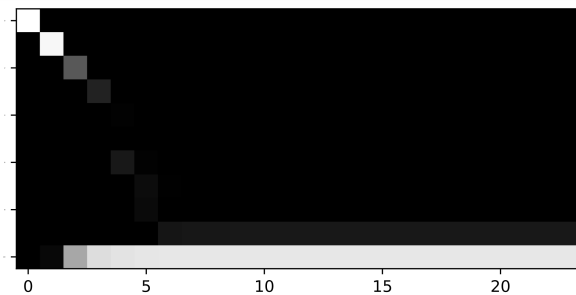
**Fig. 36.** The target error kind is VALUEERROR, occurring on line 1 ( $n = 0$ ). BASELINE incorrectly predicts INDEXERROR with confidence 0.319 on line 1 ( $n = 0$ ). DOCSTRING correctly predicts VALUEERROR with confidence 0.880 on line 2 ( $n = 1$ ), corresponding to  $A[n]$ . Both BASELINE and DOCSTRING instruction pointer values start out sharp and quickly shift most of the probability mass to the exception node.

Input: Input is given from Standard Input in the following  
 STDIN DESCRIPTION format: H N A\_1 A\_2 ... A\_N  
 Constraints:  $1 \leq H \leq 10^9$   $1 \leq N \leq 10^5$   $1 \leq A_i \leq 10^4$   
 All values in input are integers.

$n$	SOURCE	BASELINE	R.D.
0	H,N,A = list(map(int, input().split()))	9.7	3.4
1,2	for i in A[N]:	<b>43.7</b>	<b>83.0</b>
3	if H <= 0: break else:	2.9	2.8
4	H -= A[i]	6.0	0.0
5	if set(A):	0.2	0.1
6	print("Yes")	9.3	0.7
	else:		
7	print("No")	3.3	0.2



BASELINE

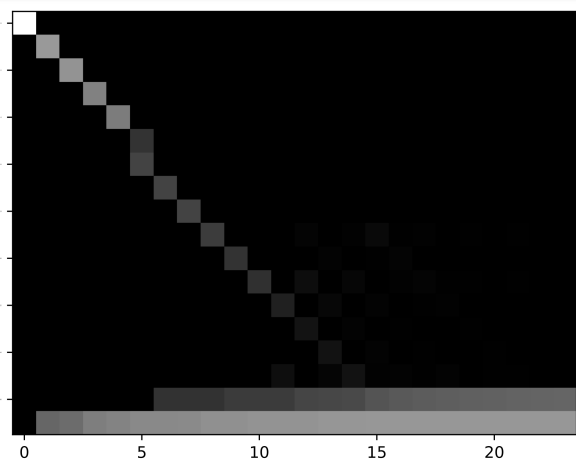


RESOURCE DESCRIPTION

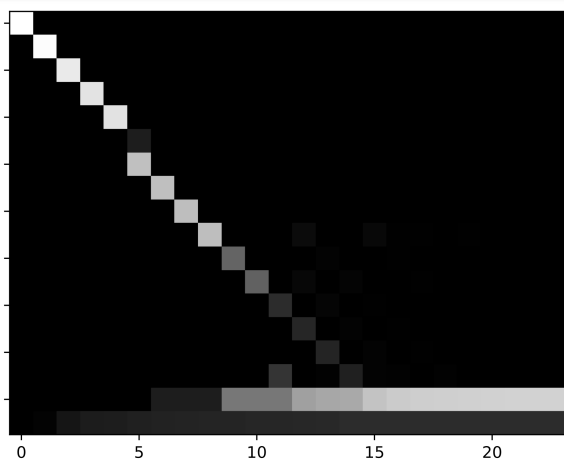
**Fig. 37.** The target error kind is NO ERROR. BASELINE correctly predicts NO ERROR with confidence 0.416. DOCSTRING also correctly predicts NO ERROR with confidence 0.823. The BASELINE instruction pointer value makes its largest probability mass contribution to  $n_{\text{error}}$  at  $n = 0$  and ends up with mass split between  $n_{\text{exit}}$  and  $n_{\text{error}}$ . The DOCSTRING instruction pointer value accumulates little probability in  $n_{\text{error}}$  and ends up with most probability mass in  $n_{\text{exit}}$ .

STDIN DESCRIPTION    Input: n m d1 d2 ... dm Two integers n and m are given in the first line. The available denominations are given in the second line.  
 Constraints: 1 <= n <= 50000 1 <= m <= 20 1 <= denomination <= 10000 The denominations are all different and contain 1.

$n$	SOURCE	BASELINE	R.D.
0	from itertools import combinations_with_replacement as C	40.1	1.3
1	n, m = map(int, input().split())	2.3	7.1
2	coin = sorted(list(map(int, input().split())))	7.2	2.8
3	if n in coin:	2.0	0.2
4	print(1)	2.0	1.5
	else:		
5	end = n // coin[0] + 1	0.3	0.1
6	b = False	0.1	0.3
7,8	for i in range(2, end):	2.4	0.7
9,10	for tup in list(C(coin, i)):	3.4	1.2
11	if sum(tup) == n:	0.3	0.0
12	print(i)	0.3	0.1
13	b = True	0.6	0.9
	break		
14	if b: break	0.1	1.4



BASELINE



RESOURCE DESCRIPTION