STATIC PREDICTION OF RUNTIME ERRORS BY LEARNING TO EXECUTE PROGRAMS WITH EXTERNAL RESOURCE DESCRIPTIONS

David Bieber
Google Research
dbieber@google.com

Rishab Goel
Mila
rgoel0112@gmail.com

Daniel Zheng
Google Research
danielzheng@google.com

Hugo Larochelle
Google Research
hugolarochelle@google.com

Daniel Tarlow
Google Research
dtarlow@google.com

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.

1 INTRODUCTION

We investigate applying neural machine learning methods to the static analysis of source code for the 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?

Recent work has made considerable progress toward applying machine learning to learning for code tasks (Allamanis et al., 2018a). Large language models show promise in source code generation, but have struggled on tasks that require reasoning about the execution behavior of programs (Chen et al., 2021a; Austin et al., 2021), requiring detailed step-by-step supervision to make headway on the task (Anonymous, 2022). We introduce the runtime error prediction task as a challenge problem, and in line with these findings, observe that it is a challenging task for generic models like Transformers.

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) (Bieber et al., 2020) 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 real-world code, adding the ability to "learn to execute" descriptions of the contents of external resources, and extending the architecture to model exception handling. We perform evaluations comparing these interpreter-inspired architectures against Transformer, LSTM, and GGNN baselines. Results show that our combined improvements lead to a large increase in accuracy in predicting runtime errors and to interpretability that allows us to predict the location of errors even though the models are only trained on the presence or absence and class of error.

In total, we summarize our contributions as:

- We introduce the runtime error prediction task and Python Runtime Errors dataset, with runtime error labels for millions of competition Python programs.
- We demonstrate for the first time that IPA-GNN architectures are practical for processing real world programs, scaling them to a depth of 174 layers, and finding them to outperform generic models on the task.
- We demonstrate that external resource descriptions such as descriptions of the contents 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

Execution behavior for identifying and localizing errors in source code Program analysis is a rich family of techniques for detecting defects in programs, including static analyses which are performed without executing code (Livshits & Lam, 2005; Xie & Aiken, 2006; Ayewah et al., 2008) and dynamic analyses which are performed at runtime (Cadar et al., 2008; Sen et al., 2005; Godefroid et al., 2005). Linters and type checkers are popular error detection tools that use static analysis.

There has been considerable recent interest in applying machine learning to identifying and localizing errors in source code (Allamanis et al., 2018a). Puri et al. (2021) makes a large dataset of real world programs available, which we build on in constructing the Python Runtime Errors dataset. Though dozens of tasks such as variable misuse detection and identifying "simple stupid bugs" (Chen et al., 2021b; Allamanis et al., 2018b; Karampatsis & Sutton, 2020; Hellendoorn et al., 2020) have been considered, most do not require reasoning about execution behavior, or else do so on synthetic data (Zaremba & Sutskever, 2014; Bieber et al., 2020), or using step-by-step trace supervision (Anonymous, 2022). Chen et al. (2021a) and Austin et al. (2021) find program execution to be a challenging task even for large-scale multiple billion parameter models. In contrast with the tasks so far considered, the task we introduce requires reasoning about the execution behavior of real programs. Several works including Vasic et al. (2019) and Wang et al. (2021) perform error localization in addition to identification, but they use localization supervision, whereas this work does not.

Interpreter-inspired models Several neural architectures draw inspiration from program interpreters (Bieber et al., 2020; Graves et al., 2014; Bošnjak et al., 2017; Gaunt et al., 2017; Łukasz Kaiser & Sutskever, 2016; Dehghani et al., 2019; Reed & de Freitas, 2016; Graves et al., 2016). Our work is most similar to Bieber et al. (2020) and Bošnjak et al. (2017), focusing on how interpreters handle control flow and exception handling, rather than on memory allocation and function call stacks.

3 Runtime Error Prediction

The goal in the *runtime error prediction* task is to determine statically whether a program is liable to encounter a runtime error when it is run. The programs will not be executable, as they depend on external resources which are not available. Descriptions of the contents of these external resources are available, which makes reasoning about the execution behavior of the programs plausible, despite there being insufficient information available to perform the execution. Examples of external resources include program inputs, file contents, and the network.

We treat the task as a classification task, with each error type as its own class, with "no error" as one additional class, and with each program having only a single target.

Python Runtime Errors We construct the *Python Runtime Errors* dataset from submissions to competitive programming problems from Project CodeNet (Puri et al., 2021), running each program on a sample input in a sandboxed environment to determine its target error class. We describe the full dataset generation and filtering process in precise detail in Appendix A.

Our dataset contains 2.44 million syntactically valid Python 3 submissions with control flow graphs, 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, which we use as the ground truth for the unsupervised localization task of Section 5.3. The dataset divides 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, thereby reducing 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.

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 real-world 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 interpreter defined by the pseudocode in Algorithm 1, ignoring the magenta text, which we call Interpreter A. We frame these modifications in relation to specific lines of the algorithm: scaling the IPA-GNN to real-world 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 Interpreter B (Section 4.3). We showcase the behavior of both Interpreter A and B on a sample program in Figure 1, and illustrate an execution of the same program by a continuous relaxation of Interpreter B alongside it.

Algorithm 1 Interpreter implemented by Exc. IPA-GNN

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

4.1 Extending the IPA-GNN to Real-world Code

Bieber et al. (2020) 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 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 2), computes a branch decision (Lines 7-8, Equation 4), and performs mean-field averaging over the resulting states and instruction pointers (Equations 5 and 6 in Bieber et al. (2020)).

Unlike in Bieber et al. (2020) where program statements are simple enough to be uniformly encoded as four-tuples, the programs in the Python Runtime Errors dataset consist of arbitrarily complex Python statements authored by real programmers in a competition setting. The language features

```
n  SOURCE
1  x = input()
2  if x > 0:
3    y = 4/3 * x
4  else:
5    y = abs(x)
6  z = y + sqrt(x)
7  <exit>
8  <raise>
```

(a) Sample program illustrative of Algorithm 1 behavior.

STDIN	`-3`
STDIN DESCRIPTION	`"A single integer -10..10"`

(b) Resource description indicates likely values the sample program will receive on stdin.

t	$h_{A,B}$	p_A	p_B	$h_{\tilde{B}}$	$p_{\tilde{B}}$
0	`{}`	1	1	□	`[10000000]`
1	`{x: -3}`	2	2	■	`[01000000]`
2	`{x: -3}`	5	5	■	`[00001000]`
3	`{x: -3, y: 3}`	6	6	🟦	`[00000100]`
4	`ValueError(lineno=6)`	7	8	🟨	`[00000001]`

(c) Step-by-step execution of the sample program according to Interpreters A and B, and under continuous relaxation $\tilde{B}$. Distinct colors represent distinct embedding values.

Figure 1: A sample program and its execution under discrete interpreters A and B (Algorithm 1) and under a continuous relaxation $\tilde{B}$ of Interpreter B.

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 4.

To compute per-statement embeddings $\text{Embed}(x_n)$ for each statement x_n in an arbitrary program x's statement-level control flow graph, we first apply either a *local* or *global* Transformer encoder to produce per-token embeddings, and then apply one of four pooling variants to a span of such embeddings. 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

$$\text{Embed}(x_n) = \text{Pool}\left(\text{Transformer}(x)_{\text{Span}(x,n)}\right). \tag{1}$$

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 real-world code requires up to 174 steps of the IPA-GNN. To reduce the memory requirements, we apply rematerialization at each step of the model (Griewank & Walther, 2000; Chen et al., 2016).

4.2 Executing with Resource Descriptions

Each program x in the dataset is accompanied by a description of what values stdin may contain at runtime, which we tokenize to produce embedding $d(x)$. Analogous to Algorithm 1 Line 1, IPA-GNN architectures initialize with per-node hidden states $h_{0,:} = 0$ and soft instruction pointer $p_{0,n} = \mathbb{1}\{n = 0\}$. 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

$$a_{t,n}^{(1)} = \text{RNN}(h_{t-1,n}, \text{Modulate}(\text{Embed}(x_n), d(x), h_{t-1,n})). \tag{2}$$

Here, magenta text shows our modification to the IPA-GNN architecture to incorporate external resource descriptions. We consider both *Feature-wise Linear Modulation* (FiLM) (Perez et al., 2017) and *cross-attention* (Lee et al., 2019) for the Modulate function, which we define in Appendix B. This allows the IPA-GNN to execute differently at each step depending on what information the resource description provides, whether that be type information, value ranges, or candidate values.

We also consider one additional method that applies to any model: 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 B, computing a raise decision (Lines 4-6, Equation 3). With this modification, we call the architecture that results the “Exception IPA-GNN”.

Whereas execution always proceeds from statement to next statement in Interpreter A and in the IPA-GNN, Interpreter B admits another behavior. In Interpreter B 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_{error}. In the sample execution in Figure 1c we see at step $t = 4$ the instruction pointer p_B updates to $n_{\text{error}} = 8$.

We write that the IPA-GNN makes raise decisions as

$$b_{t,n,r(n)}, (1 - b_{t,n,r(n)}) = \text{softmax}\left(\text{Dense}(a^{(1)}_{t,n})\right). \tag{3}$$

The dense layer here has two outputs representing the case that an error is raised and that no error is raised. Here $r(n)$ denotes the node that statement n raises to in program x; $r(n) = n_{\text{error}}$ if n is not contained in a try/except frame, and $b_{t,n,n'}$ denotes the probability assigned by the model to transitioning from executing 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.

$$b_{t,n,n_1}, b_{t,n,n_2} = (1 - b_{t,n,r(n)}) \cdot \text{softmax}\left(\text{Dense}(a^{(1)}_{t,n})\right). \tag{4}$$

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 one successor, rather than separate true and false branches, 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. (2020), detailed in Appendix C. After $T(x)$ steps of the architecture, the model directly uses the probability mass at n_{exit} and n_{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

$$P(\text{no error}) \propto p_{T(x),n_{\text{exit}}} \text{ and} \tag{5}$$

$$P(\text{error}) \propto p_{T(x),n_{\text{error}}}, \text{ with} \tag{6}$$

$$P(\text{error} = k \mid \text{error}) = \text{softmax}\left(\text{Dense}(h_{T(x),n_{\text{error}}})\right). \tag{7}$$

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

4.4 Unsupervised Localization of Errors

When we model exception handling explicitly in the IPA-GNN, the model makes soft decisions as to when to raise exceptions. We can interpret these decisions as predictions of the location where a program might raise an error. We can then evaluate how closely these location predictions match the true locations where exceptions are raised, despite never training the IPA-GNN with supervision that indicates error locations.

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 exception node across all time steps. This gives an estimate of *exception provenance* as

$$p(\text{error at statement } n) = \sum_t p_{t,n} \cdot b_{t,n,n_{\text{error}}}. \tag{8}$$

For programs with a try/except frame, we must trace the exception back to the statement that originally raised it. We provide the calculation for this in Appendix D.

5 Experiments

In our experiments we evaluate the following research questions:

RQ1: How does the adaptation of the IPA-GNN to real-world code compare against standard architectures like the 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 latent instruction pointer quantities inside the Exception IPA-GNN for localizing where errors arise? How does unsupervised localization with the Exception IPA-GNN compare to alternative unsupervised approaches to localization based on multiple instance learning and standard architectures? (Section 5.3)

5.1 Evaluation of IPA-GNN Against Baselines

We first compare the IPA-GNN architectures with Transformer (Vaswani et al., 2017), GGNN (Li et al., 2017), and LSTM (Hochreiter & Schmidhuber, 1997) baselines. In all approaches, we use the 30,000 token vocabulary constructed in Appendix A, applying Byte-Pair Encoding (BPE) tokenization (Sennrich et al., 2016) 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. (2020) we encode programs for a GGNN using six edge types, and use a two-layer LSTM for both the LSTM baseline and for the RNN in all IPA-GNN variants.

For each approach, we perform an independent hyperparameter search using random search. We list the hyperparameter space considered and model selection criteria in Appendix C. 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 1 (rows without check marks). Weighted F1 score weights per-class F1 scores by class frequency, while weighted error F1 score restricts consideration to examples with a runtime error. We perform additional evaluations using the same experimental setup but distinct initializations to compute measures of variance (Appendix E).

Table 1: Accuracy, weighted F1 score (W. F1), and weighted error F1 score (E. F1) on the Python Runtime Errors balanced test set.

	Model	R.D.?	Acc.	W. F1	E. F1
Baselines	GGNN		62.8	58.9	45.8
	Transformer		63.6	60.4	48.1
	LSTM		66.1	61.4	48.4
Ablations	GGNN	✔	68.3	66.5	56.8
	Transformer	✔	67.3	65.1	54.7
	LSTM	✔	68.1	66.8	58.3
	IPA-GNN		68.3	64.8	53.8
	E. IPA-GNN		68.7	64.9	53.3
Ours	IPA-GNN	✔	71.4	70.1	62.2
	E. IPA-GNN	✔	**71.6**	**70.9**	**63.5**

RQ1: The interpreter-inspired architectures show significant gains over the GGNN, Transformer, and baseline approaches on the runtime error prediction task. We attribute this to the model's inductive bias toward mimicking program execution.

Table 2: A comparison of early and late fusion methods for incorporating external resource description information into interpreter-inspired models.

	Baseline			Docstring			FiLM			Cross-attention		
Model	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 from 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 in Table 1 (compare rows with check marks to corresponding rows 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 2. 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 the test split.

RQ2: Across all architectures it is clear that external resource descriptions are essential for improved 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. 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 2 the instruction pointer values of two Exception IPA-GNN models on a single example (shown in Table 8). The model with the resource description recognizes 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 the Python 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 G.

5.3 Interpretability and Localization

We next investigate the Exception IPA-GNN model's surprising 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 supervision indicating error presence and kind.

Multiple Instance Learning Baselines The unsupervised localization task may be viewed as multiple instance learning (MIL) (Dietterich et al., 1997) by treating the subtask of predicting whether a particular line contains an error as an instance. We thus consider as baselines two variations on the Transformer architecture, each using multiple instance learning. 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 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, aggregated to form an overall prediction (details in Appendix F).

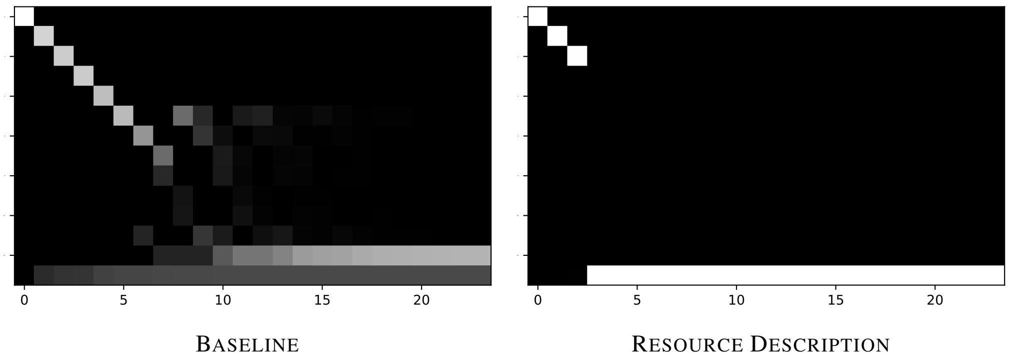
Localization Experiment We use the same experimental protocol as in Section 5.1, and train each of the MIL Transformer and Exception IPA-GNN models. As before, the models are trained only to minimize cross-entropy loss on predicting error kind and presence, receiving no supervision as to the location of the errors. We report the localization results in Table 3, measuring the percent of test examples with errors for which the model correctly predicts the line number of the error.

RQ3: The Exception IPA-GNN's unsupervised localization capabilities far exceed that of the baseline approaches. In Figure 2 we see the flow of instruction pointer mass during the execution of a sample program (Table 8) by two Exception IPA-GNN models, including the steps where the models

Table 3: Localization accuracy (%) for the MIL Transformers and Exception IPA-GNN on the Python Runtime Errors balanced test split.

Model	R.D.?	Local.
Local MIL Transformer		33.0
Local MIL Transformer	✓	48.9
Global MIL Transformer		48.2
Global MIL Transformer	✓	48.8
E. IPA-GNN		50.8
E. IPA-GNN + Docstring	✓	64.7
E. IPA-GNN + FiLM	✓	64.5
E. IPA-GNN + C.A.	✓	**68.8**

raise probability mass to n_{error}. Tallying the contributions to n_{error} from each node yields the exception provenance values in the right half of Table 8. This shows 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.



Baseline

Resource Description

Figure 2: Heatmap of instruction pointer values produced by Baseline and Docstring Exception IPA-GNNs for the example in Table 8 (Appendix G). The x-axis represents timesteps and the y-axis represents program statement nodes, with the last two rows respectively representing n_{exit} and n_{error}.

6 Discussion

In this work, we introduce the new task of predicting runtime errors in competitive programming problems and advance the capabilities of interpreter-inspired models. We tackle the additional complexity of real-world code and demonstrate how natural language descriptions of external resources can be leveraged to reduce the ambiguity that arises in a static analysis setting. We show that the resulting models outperform standard alternatives like Transformers and that the inductive bias built into these models allows 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 more steps of program execution with each model step, to reduce the number of model steps necessary for long programs. Extending the interpreter-inspired models with additional interpreter features, or to support multi-file programs or programs with multiple user-defined functions is also an interesting avenue for future work.

Learning to understand programs remains a rich area of inquiry for machine learning research because of its complexity and the many interacting facets 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.

References

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, 2018a.

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

Anonymous. Show your work: Scratchpads for intermediate computation with language models. In *Submitted to The Tenth International Conference on Learning Representations*, 2022. URL `https://openreview.net/forum?id=iedYJm92o0a`. under review.

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.

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.

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.

Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. Programming with a differentiable forth interpreter, 2017.

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, pp. 209–224, 2008.

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, 2021a.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost, 2016.

Zimin Chen, Vincent J Hellendoorn, Petros Maniatis, Pascal Lamblin, Pierre-Antoine Manzagol, Danny Tarlow, and Subhodeep Moitra. Plur: A unifying, graph-based view of program learning, understanding, and repair. 2021b.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers, 2019.

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. URL `https://www.sciencedirect.com/science/article/pii/S0004370296000343`.

Alexander L. Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. Differentiable programs with neural libraries, 2017.

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*, pp. 213–223, 2005.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines, 2014.

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`.

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`.

Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, and Petros Maniatis. Global relational models of source code. In *International Conference on Learning Representations*, 2020.

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`.

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`.

Juho Lee, Yoonho Lee, Jungtaek Kim, Adam R. Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks, 2019.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks, 2017.

V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX security symposium*, volume 14, pp. 18–18, 2005.

Flemming Nielson and Hanne Riis Nielson. Interprocedural control flow analysis. In *ESOP*, 1999.

Ethan Perez, Florian Strub, Harm de Vries, Vincent Dumoulin, and Aaron Courville. Film: Visual reasoning with a general conditioning layer, 2017.

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.

Scott Reed and Nando de Freitas. Neural programmer-interpreters, 2016.

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.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units, 2016.

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.

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.

Shangwen Wang, Kui Liu, Bo Lin, Li Li, Jacques Klein, Xiaoguang Mao, and Tegawendé F. Bissyandé. Beep: Fine-grained fix localization by learning to predict buggy code elements, 2021.

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.

Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security Symposium*, volume 15, pp. 179–192, 2006.

Wojciech Zaremba and Ilya Sutskever. Learning to execute, 2014.

Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms, 2016.

A Python Runtime Error Dataset Details

We describe in detail the construction of the Python Runtime Error dataset from the submissions in Project CodeNet (Puri et al., 2021). 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.

A.1 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, 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. A program may result in a runtime error during parsing if it contains return, break, continue keywords outside of an appropriate frame.

A.2 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.

A.3 Parsing Input Space Information from Problem Statements

For each problem, we parse the problem statement to extract the *input description* and *input constraints*, if they exist. Problem statements are written either in English or Japanese, and so we write our parser to support both languages. When one or more of these two sections are present in the problem statement, we construct an *input space description* containing the contents of the present sections. For the experiments that use input space information as a docstring, we prepend to each submission our the input space description for its corresponding problem. Similarly the input space descriptions are used in the experiments that process input space information with either cross-attention or FiLM.

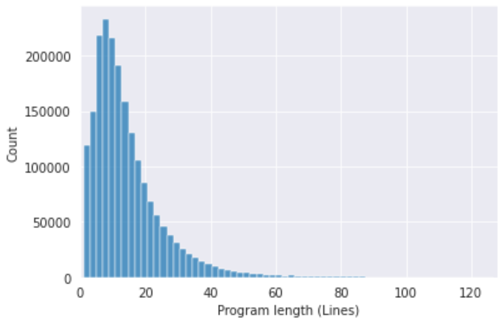


Figure 3: A histogram showing the distribution of program lengths, measured in lines, represented in the Python Runtime Errors train split.

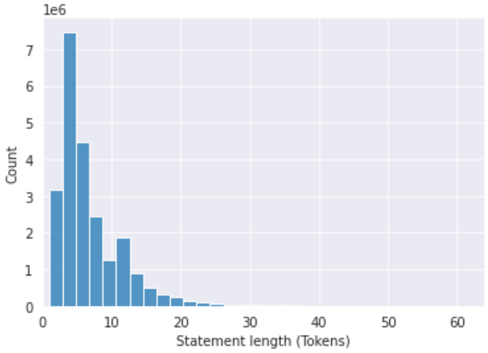


Figure 4: The distribution of statement lengths, measured in tokens, in the Python Runtime Errors train split.

A.4 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 (Sennrich et al., 2016). 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 C 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.

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) (Nielson & Nielson, 1999). We leave the investigation of modelling user-defined functions to further work.

A.5 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 4. Figure 3 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 4.

B Input Modulation

We consider *cross-attention* (Lee et al., 2019) and *Feature-wise Linear Modulation* (FiLM) (Perez et al., 2017) as the $\mathrm{Modulate}$ function in Section 4.2. After embedding the node and the resource

Table 4: Distribution of target classes in the Python Runtime Errors dataset. † denotes examples in the balanced test split.

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

description we use cross-attention as follows to modulate the input.

$$\text{MultiHead}(\text{Embed}(x_n), d(x), h_{t-1,n}) = \text{Concat}(\text{Concat}(head_1, ..., head_h)W^O, \text{Embed}(x_n)) \tag{9}$$

$$\text{where } head_i = \text{softmax}\left(\frac{QK'}{\sqrt{d_k}}\right)V \tag{10}$$

$$Q = W_i^Q\,\text{Concat}(Embed(x_n), h_{t-1,n}) \tag{11}$$

$$K = W_i^K d(x) \tag{12}$$

$$V = W_i^V d(x) \tag{13}$$

Here, $W^O \in R^{hd_v \times d_{model}}$, $W_i^Q \in R^{d_k \times (d_{model} + d_{\text{Embed}(x_n)})}$, $W_i^K \in R^{d_k \times d_{d(x)}}$, 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:

$$\text{FiLM}(\text{Embed}(x_n), d(x), h_{t-1,n}) = \text{Concat}(\beta \cdot d(x) + \gamma, \text{Embed}(x_n)) \tag{14}$$

$$\text{where } \beta = \sigma(W_\beta\,\text{Concat}(x_n, h_{t-1,n}) + b_\beta), \tag{15}$$

$$\gamma = \sigma(W_\gamma\,\text{Concat}(x_n, h_{t-1,n}) + b_\gamma), \tag{16}$$

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

C 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 6. All architectures use a Transformer encoder, and the Transformer sizes considered in the search are listed in Table 6 and defined further in Table 5.

Table 5: 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

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. (2020) 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.

D 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

$$p(\text{error at statement } n) = \sum_t p_{t,n} \cdot b_{t,n,n_{\text{error}}}. \tag{17}$$

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

$$v_{t,n,n'} = \sum_{k \in N_{\text{in}}(n')} v_{t-1,n,k} \cdot b_{t,k,n'} \cdot p_{t,k} + (1 - \sum v_{t-1,:,n}) \cdot b_{t,n,n'} \cdot p_{t,n} \cdot \mathbb{1}\{n' = r(n)\}. \tag{18}$$

Table 6: 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

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

$$p(\text{error at statement } n) = v_{T(x),n,n_{\text{error}}}, \tag{19}$$

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

E 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 7; ± values are one standard deviation.

Table 7: 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

F Multiple Instance Learning

The Local Transformer and Global Transformer models each compute per-statement node embeddings $\text{Embed}(x_n)$ given by Equation 1. In the multiple instance learning setting, these are transformed into unnormalized per-statement class predictions

$$\phi(\text{class} = k, \text{lineno} = l) = \text{Dense}\left(\text{Embed}(x_n)\right). \tag{20}$$

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

$$\log p(\text{class} = k) \propto \log\left(\sum_l \exp \phi(k,l)\right), \tag{21}$$

$$\log p(\text{lineno} = l) \propto \log\left(\sum_{k \in K} \exp \phi(k,l)\right) \tag{22}$$

where K is the set of error classes.

The *max* and *mean* strategies meanwhile follow Wang et al. (2018) in asserting

$$p(\text{class} = k \mid \text{lineno} = l) = \text{softmax}\left(\phi(k,l)\right), \tag{23}$$

compute the location probabilities as

$$p(\text{lineno} = l) \propto \sum_{k \in K} p(\text{class} = k \mid \text{lineno} = l), \tag{24}$$

and compute the outputs as

$$\log p(\text{class} = k) \propto \log \max_l p(\text{class} = k \mid \text{lineno} = l), \text{ and} \tag{25}$$

$$\log p(\text{class} = k) \propto \log \frac{1}{L} \sum_l p(\text{class} = k \mid \text{lineno} = l) \tag{26}$$

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 $p(\text{lineno})$.

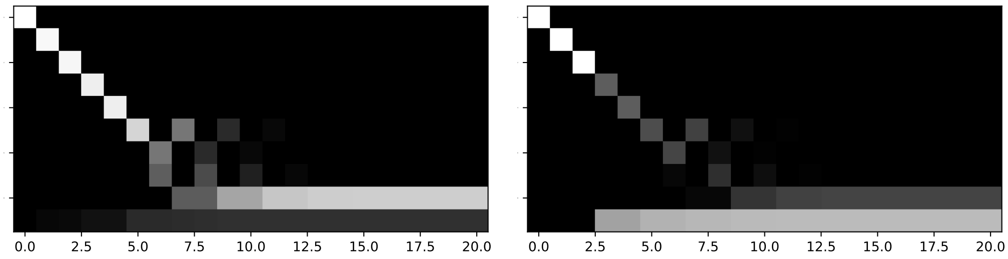
G Example visualizations

Additional randomly sampled examples from the Python Runtime Error dataset validation split are shown here. As in Figure 2, prediction visualizations for these examples are shown for the selected BASELINE and DOCSTRING Exception IPA-GNN model variants.

In instruction pointer value heatmaps, the x-axis represents timesteps and the y-axis represents nodes, with the last two rows respectively representing the exit node n_{exit} and the exception node n_{error}. Note that for loop statements are associated with two spans in the statement-level control flow graph, one for the construction of the 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.

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 Error contrib.	R.D. Error contrib.
0	`N = int(input())`	2.9	0.2
1	`A = list(map(int, input().split()))`	0.8	0.0
2	`res = 0`	3.0	**63.3**
3,4	`for i in range(1, len(A)+1, 2):`	9.8	6.3
5	`  res += A[i] % 2`	0.3	0.1
6	`print(res)`	0.2	2.2



BASELINE RESOURCE DESCRIPTION

Figure 5: The target error kind is INDEXERROR, occuring 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_{exit}. The DOCSTRING instruction pointer value has a small amount of probability mass reaching n_{exit}, with most probability mass ending at n_{error}.

Finally, we show in tabular form the error localization predictions corresponding to Figure 2 of Section 5.3.

STDIN DESCRIPTION `Input: Input is given from Standard Input in the following format: H N A_1 A_2 ... A_N Constraints: 1 <= H <= 10^9 1 <= N <= 10^5 1 <= A_i <= 10^4 All values in input are integers.`

n	SOURCE	BASELINE Error contrib.	R.D. Error contrib.

0	`H,N,A = list(map(int,`
	`input().split()))`
		9.7	3.4

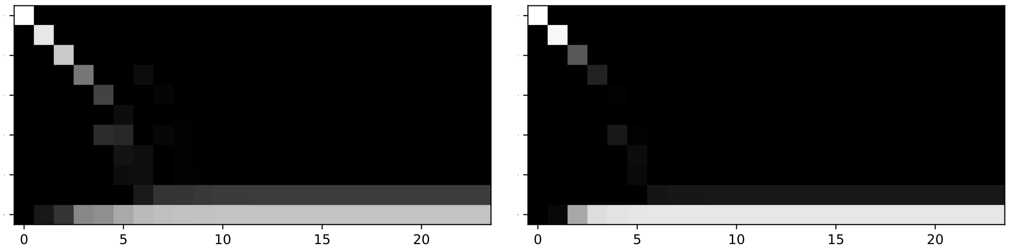
1,2	`for i in A[N]:`	**43.7**	**83.0**

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")`
	`else:`
		9.3	0.7

7	`print("No")`	3.3	0.2



BASELINE RESOURCE DESCRIPTION

Figure 6: The target error kind is VALUEERROR, occuring 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.

Table 8: The error localization predictions on an example program made by two trained Exception IPA-GNN models, highlighting the difference in behavior with and without resource descriptions.

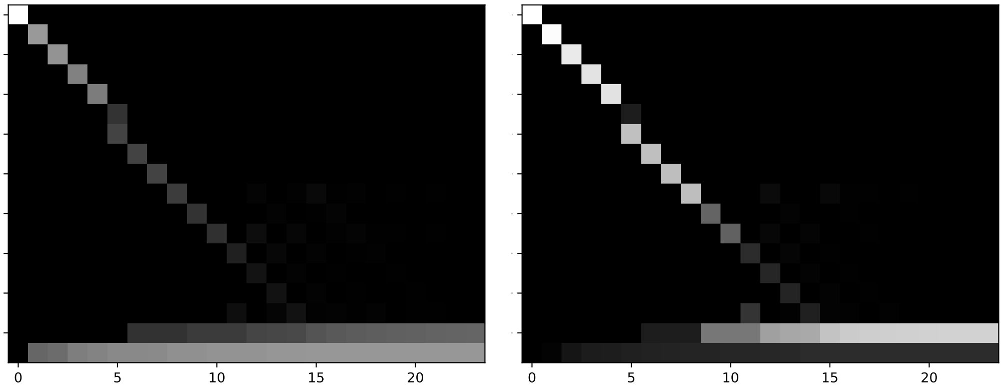
STDIN DESCRIPTION `Input: Input is given from Standard Input in the following format Constraints: Each character of S is A or B. |S| = 3`

n	SOURCE	BASELINE	R.D.
0	`a = str(input())`	16.9	0.4
1	`q = int(input())`	3.2	0.3
2	`s = [input().split() for i in range(q)]`	0.5	**99.3**
3,4	`for i in range(q):`	6.4	0.0
5	`if int(s[i][0]) == 1 and len(a)>1:`	0.1	0.0
6	`a = a[::-1]`	0.7	0.0
7	`elif int(s[i][0])== 2 and int(s[i][1])==1:`	0.1	0.0

8	`a=s[i][2]+a`
	`else:`
		0.2	0.0

9	`a=a+s[i][2]`	0.0	0.0
10	`print(a)`	1.1	0.0

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 Error contrib.	R.D. Error contrib.
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

Figure 7: 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_{error} at $n = 0$ and ends up with mass split between n_{exit} and n_{error}. The DOCSTRING instruction pointer value accumulates little probability in n_{error} and ends up with most probability mass in n_{exit}.