Learning to Execute Programs with Instruction Pointer Attention Graph Neural Networks

Authors: David Bieber, Charles Sutton, Hugo Larochelle, Daniel Tarlow (Google)

Presenter: Xingjian Zhang (UMich)

Outline

- Background
- Problem Formulation
- Approach
- Experiment
- Conclusion

Outline

- Background
  - What is “learning to execute program”?
  - Why GNN?
- Problem Formulation
- Approach
- Experiment
- Conclusion

GNNs are powerful tool for learning SDE tasks

- Many applications
  - Code completion
  - Bug finding
  - Program repair
  - Learning to execute programs
- Inherent graph structure in programs
  - parse trees
  - data flow graphs
  - control flow graphs (see example later)

Learning to execute programs

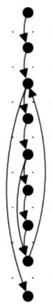
Produce the output of a program, without actually running the program.

Introduced by Wojciech Zaremba and Ilya Sutskever. Learning to execute, 2014 (applied RNN to this task)

- Challenges
  - Complex structure of programs
    - contains branch created by “if-else”, “while” loop, etc.
  - Complex reasoning about program executions
    - hard to predict the discrete branch decisions
- Non-trivial for vanilla GNN/RNNs
  - GNN - good at complex structure, but not sequential reasoning
  - RNN - good at sequential reasoning, but not complex structure

Example: Control Flow Graph

n	Source	Tokenization (x_n)				Control flow graph ($n \to n'$)	$N_{\text{in}}(n)$	$N_{\text{out}}(n)$
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>	-	-	-	-		$\{2,8\}$	$\{8\}$

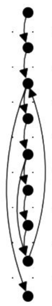


Nodes: individual line statements

Directed edges: possible sequences of execution of statements

Instruction pointer (IP): indicates the next instruction to execute

Example: Control Flow Graph

n	Source	Tokenization (x_n)				Control flow graph ($n \to n'$)	$N_{\text{in}}(n)$	$N_{\text{out}}(n)$
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>	-	-	-	-		$\{2,8\}$	$\{8\}$

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.

Notice that branch decisions are always binary. i.e. $|N_{out}(n)| <= 2$.

Outline

- Background
- Problem Formulation
  - What information are available?
  - What restrictions are imposed?
  - What practical evaluations are considered?
- Approach
- Experiment
- Conclusion

Two variants

- Full program execution
  - Input: Full program
  - Output: some semantic property of the program (e.g. program output)
  - Evaluate the expressiveness and learnability
- Partial program execution
  - Input: Partially masked program
  - Output: Same as above
  - Useful in certain downstream applications

Problem Formulation (ML for Static Analysis)

- Access to
  - Textual source of the program
  - Parse tree of the program
  - Any common static analysis results, e.g. control flow graph
- No access to
  - Compiler (compile-time)
  - Interpreter (runtime)
  - Dependencies, test suite, etc.
  - Any artifacts not readily available for normal static analysis

The settings are similar to standard static analysis.

Other Specifications

- Bounded execution
  - Restrict the model to use fewer steps than are required by the ground truth trace
  - Force short-cuts learning
  - *Motivation: Static analysis requires low latency*
- Systematic generalization
  - Train the model with limited complexity
  - Test the model on more complex programs
  - *Motivation: People write programs to do things that have not been done before*
  - *Motivation: Reduce training cost while preserving performance on complex real-world codebase*

Formal Specification & Notations

- Given
  - $c(x)$: Complexity function
  - D: Dataset consisting of pairs (x, y)
    - x: program (code & control flow graph)
      - x_n: n-th line statement of x
      - $N_{in}(n)$: the set of statements that can immediately precede x_n
      - $N_{out}(n)$: the set of statements that can immediately follow x_n
      - $N_{all}(n)$: $N_{in}(n) \cup N_{out}(n)$
    - y: a semantic property of the program (**integer target is used in paper*)
    - D_{train} contains examples such that $c(x) <= C$, D_{test} contains others.

The task follows a standard supervised learning formulation.

Outline

- Background
- Problem Formulation
- Approach
  - Instruction Pointer RNN Models (IP-RNN)
  - Instruction Pointer Attention GNN (IPA-GNN)
  - Standard MPNN (GGNN)
- Experiment
- Conclusion

Instruction Pointer (IP) Modeling **without** Branch

- Let's start with a special case.
- Consider a classical interpreter to execute **straight-line** program
  - maintains a state consisting of the values of all variables in the program
  - maintains an instruction pointer indicating the next statement to execute (always pointing to the next statement)
  - updates the instruction pointer to next statement when a statement is executed (causal structure)
- RNN is a natural choice of architecture
  - Reason: same causal structure

Model 1: Line-by-Line RNN

- At step t of interpretation, $h_t = \mathrm{RNN}\left(h_{t-1}, \mathrm{Embed}\left(x_{n_{t-1}}\right)\right)$
  - h_t is the hidden state ("state in interpreter")
  - $n_t = t$ is the model's instruction pointer
    - At each step, always increment the IP by 1
    - Correct only for straight-line program

Instruction Pointer (IP) Modeling **with** Branch

- In general, setting $n_t = t$ is problematic!
  - Most programs are not straight-line.
- One can define different rules for updating the IP.
  - We call these model variants Instruction Pointer RNNs (IP-RNNs)

Model 2: Trace RNN (Oracle)

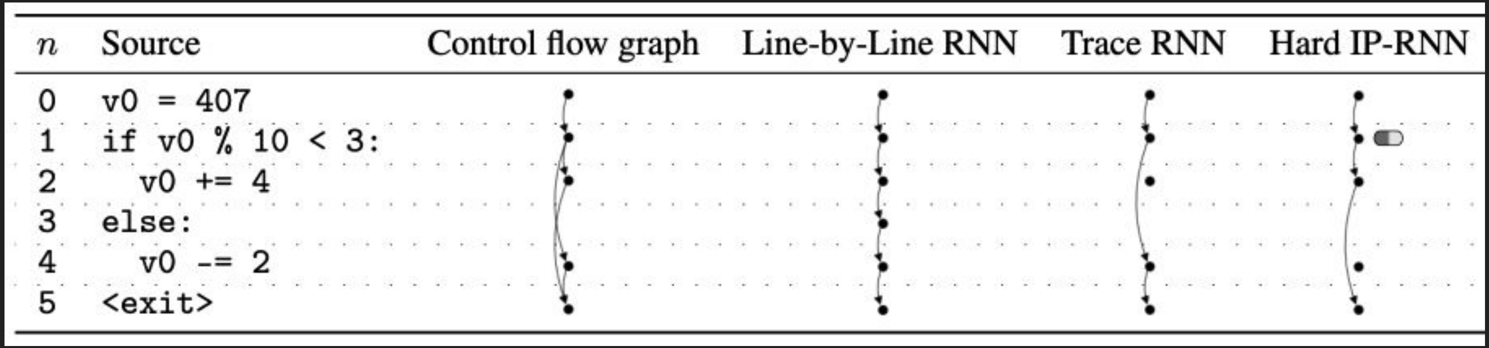
- At step t of interpretation, $h_t = \mathrm{RNN}\left(h_{t-1}, \mathrm{Embed}\left(x_{n_{t-1}}\right)\right)$
  - n_t is the model's ground truth trace of IP
    - At each step, always points to the next exact statement to execute
    - Correct for all programs

Model 3: Hard IP-RNN

- At step t of interpretation, $h_t = \mathrm{RNN}\left(h_{t-1}, \mathrm{Embed}\left(x_{n_{t-1}}\right)\right)$
  - $n_t = N_{\text{out}}(n_{t-1})|_j$ where $j = \arg\max \mathrm{Dense}(h_t)$
    - Not differentiable but might be trained by a supervised fashion*
    - More precisely, invalidates fully differentiable end-to-end training
  - The authors did not provide implementation or report experimental results for this model

Summary of IP-RNN

Oracle IP



Sequential IP

Predictive IP w/o gradient

Continuous Relaxation of Discrete Branch Choice

- Consider soft branch decision
  - A distribution over the possible branches (from a particular statement)
- Replace n_t with $p_{t,n}$
  - At step t, $p_{t,n}$ is a distribution over all statements x_n.
- Replace h_t with $h_{t,n}$
  - At step t, $h_{t,n}$ models the representation assuming the program is executing statement n.
  - Intuition: we want the model to have a different representation of the program state for possible IPs.

Model 4: IPA-GNN

- State proposal is produced for **each possible current statement n**
    - IPA-GNN: $a_{t,n} = \mathrm{RNN}\big(h_{t-1,n}, \mathrm{Embed}(x_n)\big)$
    - IP-RNN: $h_t = \mathrm{RNN}\Big(h_{t-1}, \mathrm{Embed}\big(x_{n_{t-1}}\big)\Big)$ only for statement n_{t-1}
- Consider special case
    - For straight-line code, i.e. $|N_{out}(x_n)| = 1$, $n \rightarrow n'$
        - Simply have $h_{t,n'} = a_{t,n}$

However, in general, a lot of potential traces may lead to current statement

Model 4: IPA-GNN

- Soft branch determines how much of the state proposals flow to each next statement
  - When branching between n_1 and n_2

$$b_{t,n,n_1}, b_{t,n,n_2} = \text{softmax}\big(\text{Dense}(a_{t,n})\big)$$

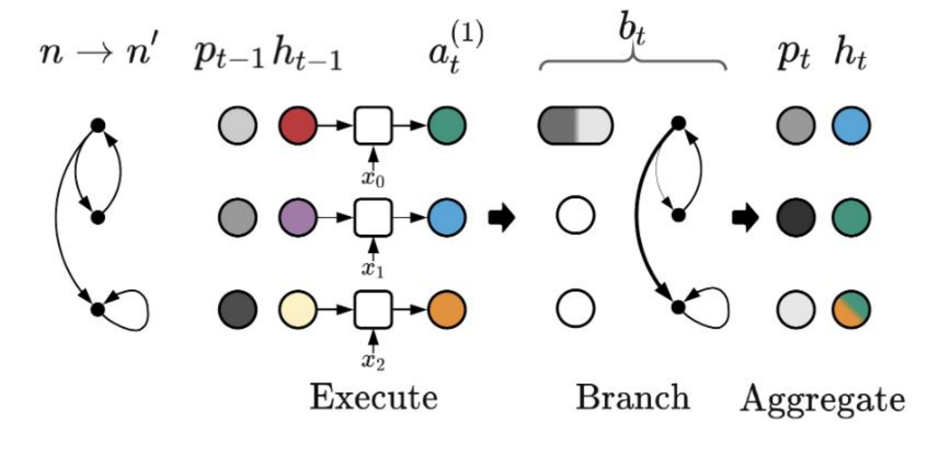
  - Update hidden state from all potential traces

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

  - Update IPA (probability distribution of IP)

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

Intuition of IPA-GNN



1. **Execute:** generate state proposal
2. **Branch:** generate soft branch decision (and update IPA)
3. **Aggregate:** generate new hidden states by weighted sum from 1 and 2

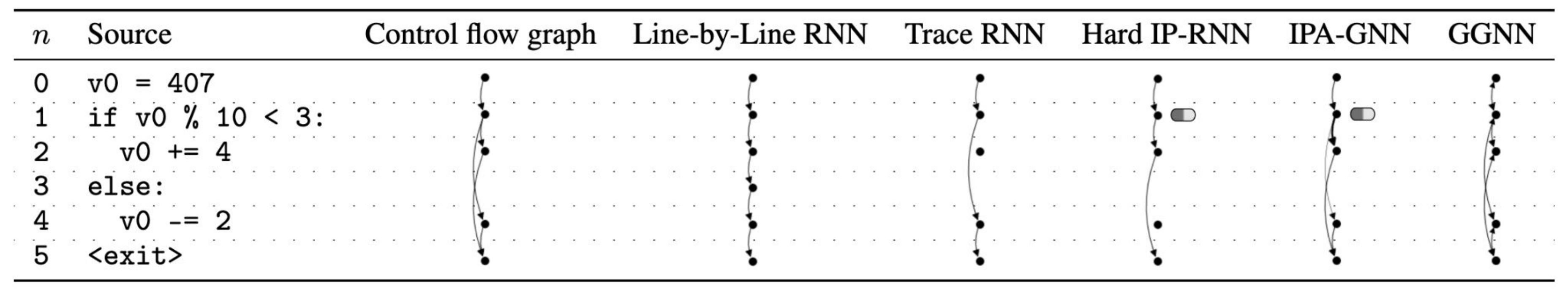
Model 5: Gated GNN (GGNN)

	IPA-GNN (Ours)	NoControl	NoExecute	GGNN
$h_{0,n}$	$= 0$	$= 0$	$= \mathrm{Embed}(x_n)$	$= \mathrm{Embed}(x_n)$
$a^{(1)}_{t,n}$	$= \mathrm{RNN}(h_{t-1,n}, \mathrm{Embed}(x_n))$	$= \mathrm{RNN}(h_{t-1,n}, \mathrm{Embed}(x_n))$	$= h_{t-1,n}$	$= h_{t-1,n}$
$a^{(2)}_{t,n,n'}$	$= p_{t-1,n'} \cdot b_{t,n',n} \cdot a^{(1)}_{t,n}$	$= 1 \cdot a^{(1)}_{t,n}$	$= p_{t-1,n'} \cdot b_{t,n',n} \cdot \mathrm{Dense}(a^{(1)}_{t,n})$	$= 1 \cdot \mathrm{Dense}(a^{(1)}_{t,n})$
$\tilde{h}_{t,n}$	$= \sum_{n' \in N_{\mathrm{in}}(n)} a^{(2)}_{t,n,n'}$	$= \sum_{n' \in N_{\mathrm{all}}(n)} a^{(2)}_{t,n,n'}$	$= \sum_{n' \in N_{\mathrm{in}}(n)} a^{(2)}_{t,n,n'}$	$= \sum_{n' \in N_{\mathrm{all}}(n)} a^{(2)}_{t,n,n'}$
$h_{t,n}$	$= \tilde{h}_t$	$= \tilde{h}_t$	$= \mathrm{GRU}(h_{t-1,n}, \tilde{h}_{t,n})$	$= \mathrm{GRU}(h_{t-1,n}, \tilde{h}_{t,n})$

- IPA-GNN shares similar computational structure with message passing neural nets (MPNN) like GGNN.
- IPA-GNN has two extra mechanisms
  - Modeling of execution (RNN)
  - Modeling of controlling (p and b)
- 3 baselines: NoControl, NoExecute, GGNN (NoControlNoExecute)

Summary

Oracle IP Predictive IP w/o gradient



Sequential IP Predictive IP w/o gradient MPNN

- IPA-GNN is closely related to both IP-RNNs and GGNN (MPNN)
  - Continuous relaxation of IP-RNNs
  - Adaptation of MPNNs on sequential reasoning

Outline

- Background
- Problem Formulation
- Approach
- Experiment
  - Experiment settings
  - Evaluation Criteria
  - Results
- Conclusion

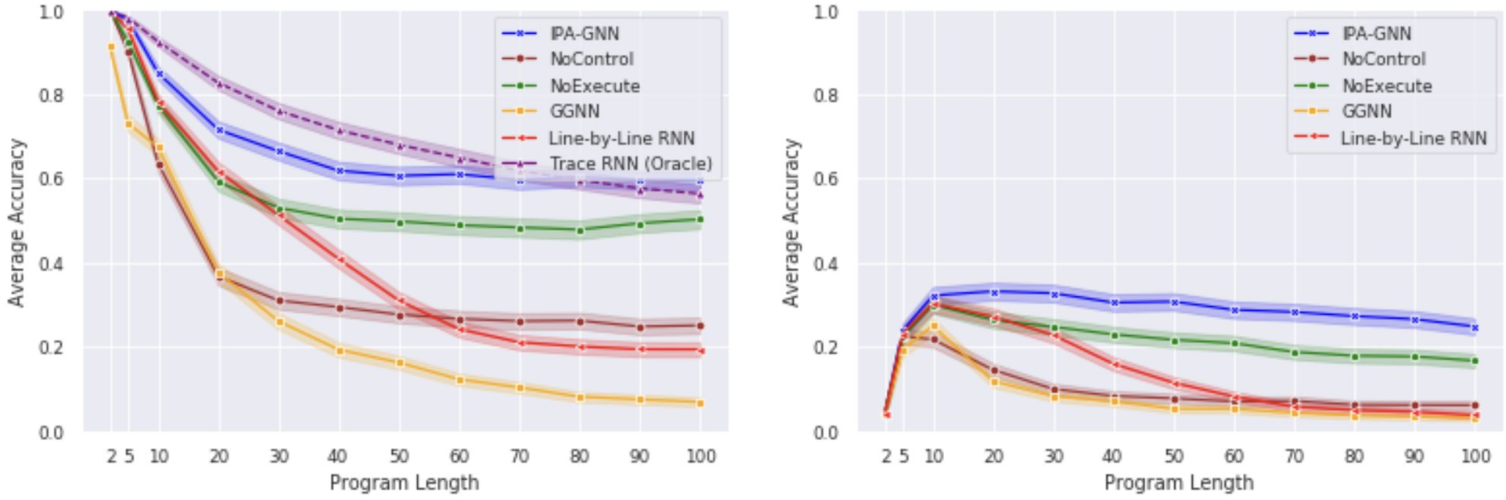
Dataset

- x: Python codes
- Statements include
  - variable assignments (v0, …, v9)
  - multi-digit arithmetic
  - while loops
  - if-else statements
- y: integer target
  - v0 mod 1000
- Complex nesting is allowed
- Use program (lines) length as complexity measure
  - c(x) := len(x)
- Train test split forces systematic generalization
  - Train: 5M len(x) <= 10 Test: 4.5k len(x) $\in \{20, 30, \ldots, 100\}$
- Partial execution
  - mask out 1 non-control statement uniformly at random

Other specifications

- Use a two-layer LSTM as the underlying RNN cell for the RNN and IPA-GNN models
- Hparams
  - batch size: 32
  - hidden dimension: $H \in \{200, 300\}$

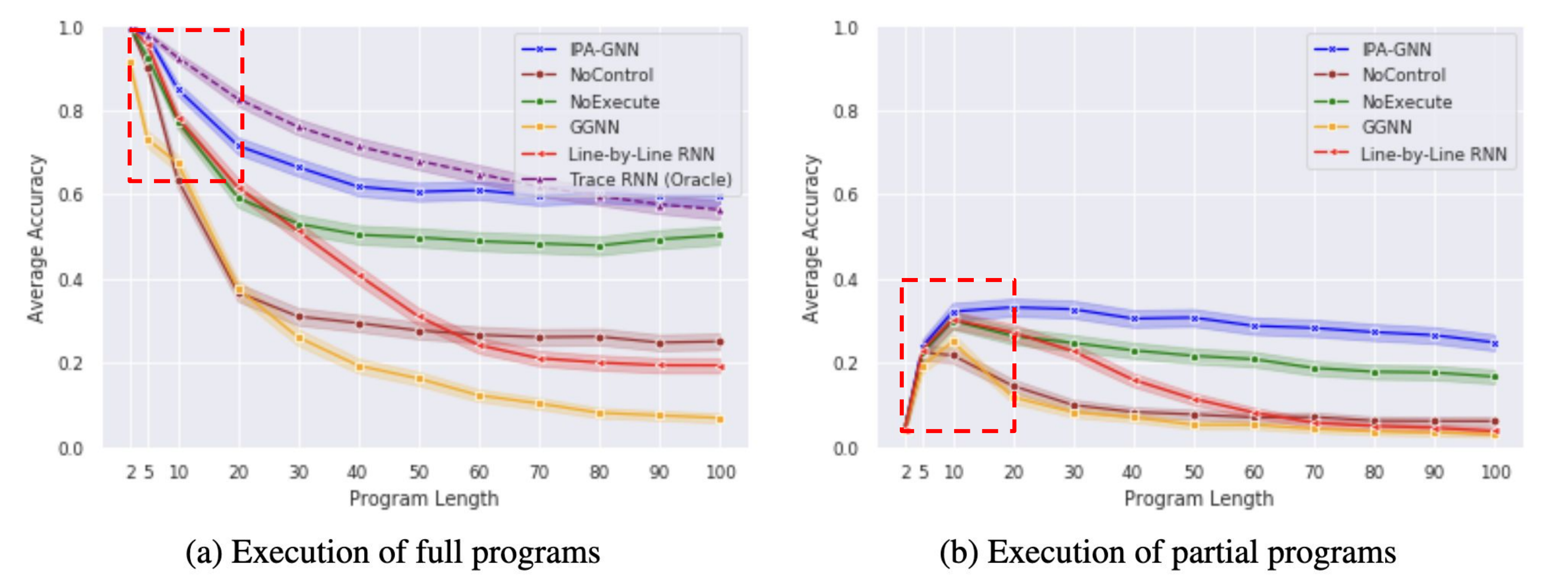
Accuracy vs. Program Length



(a) Execution of full programs

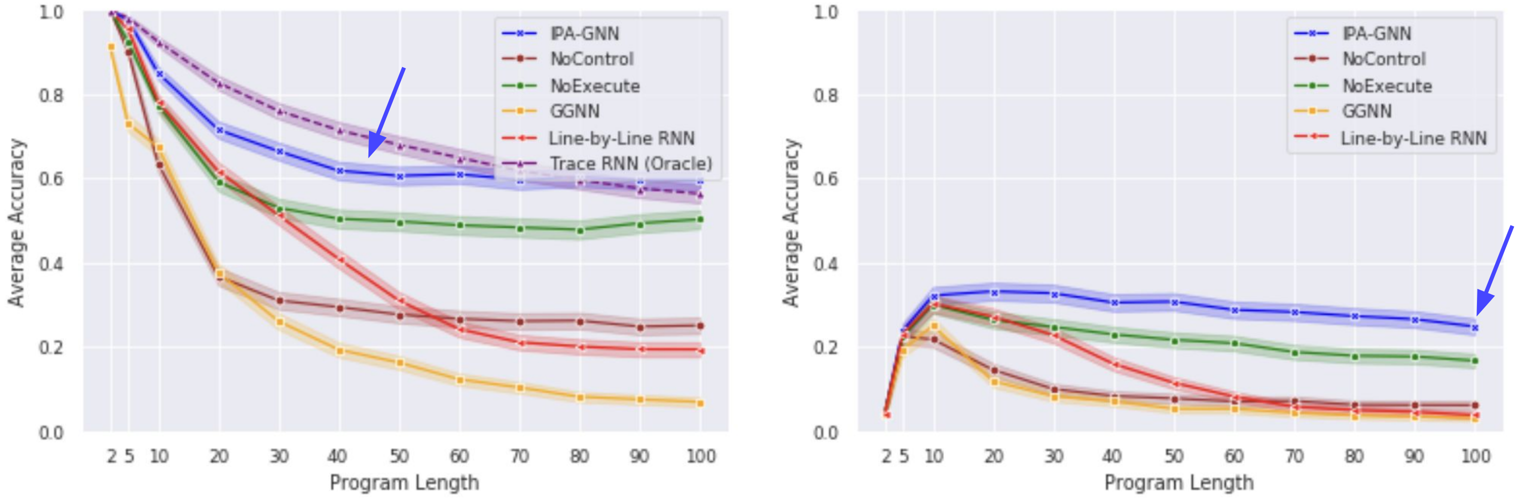
(b) Execution of partial programs

Accuracy vs. Program Length



At low complexity, the Line-by-Line RNN model performs almost as well as the IPA-GNN.

Accuracy vs. Program Length

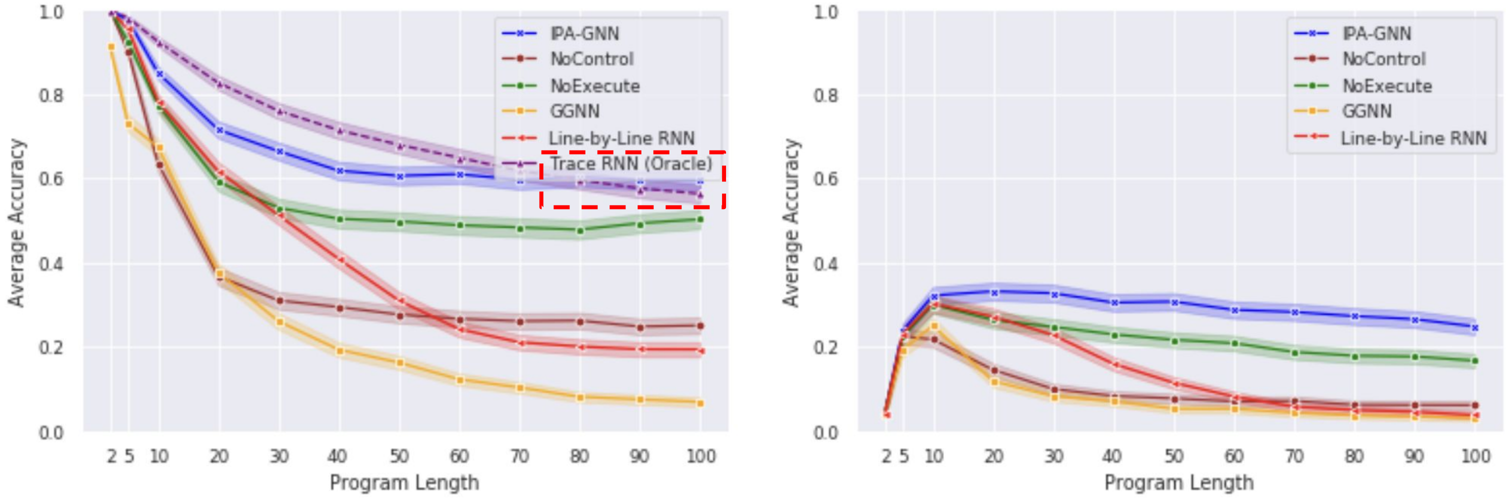


(a) Execution of full programs

(b) Execution of partial programs

As complexity increases, the performance of all baseline models drops off faster than that of the IPA-GNN.

Accuracy vs. Program Length

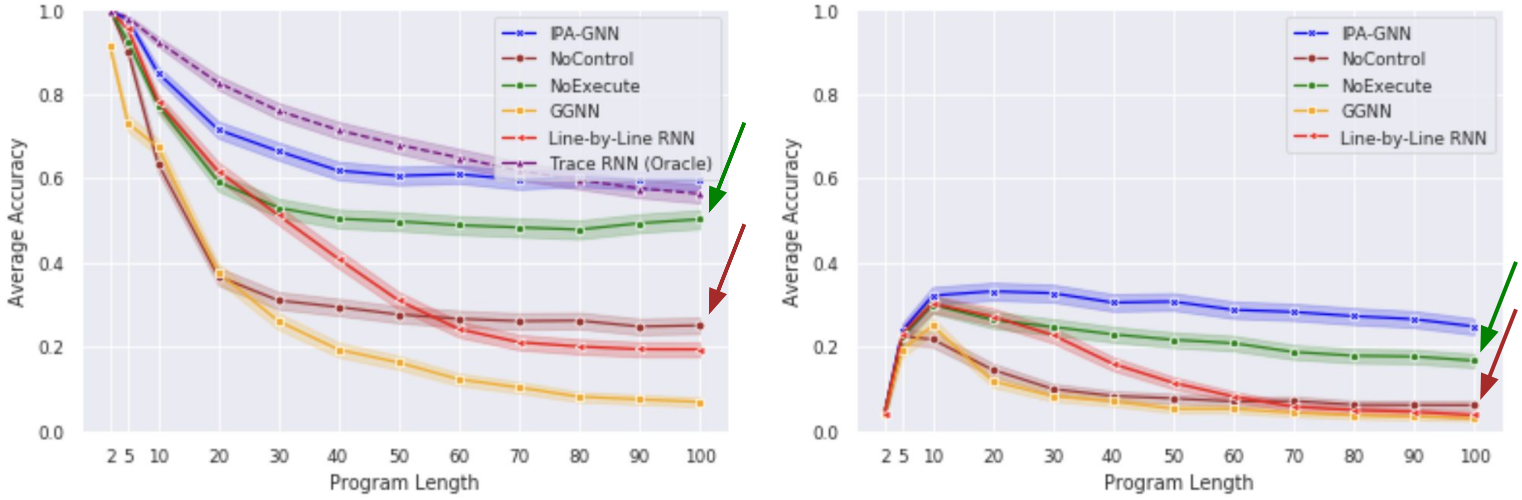


(a) Execution of full programs

(b) Execution of partial programs

Despite using the ground truth control flow, the Trace RNN does not perform as well as the IPA-GNN on long programs.

Accuracy vs. Program Length

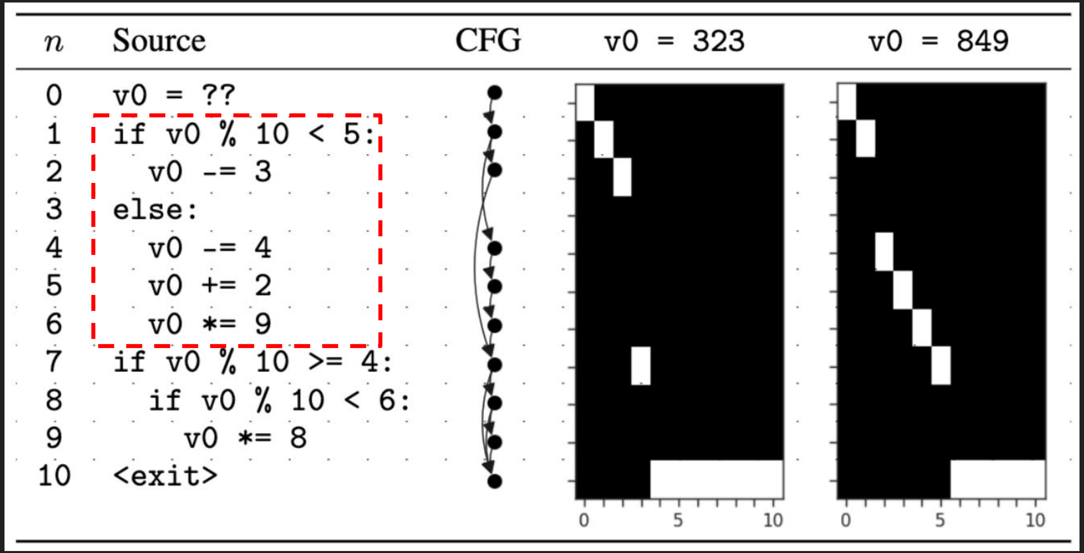


(a) Execution of full programs

(b) Execution of partial programs

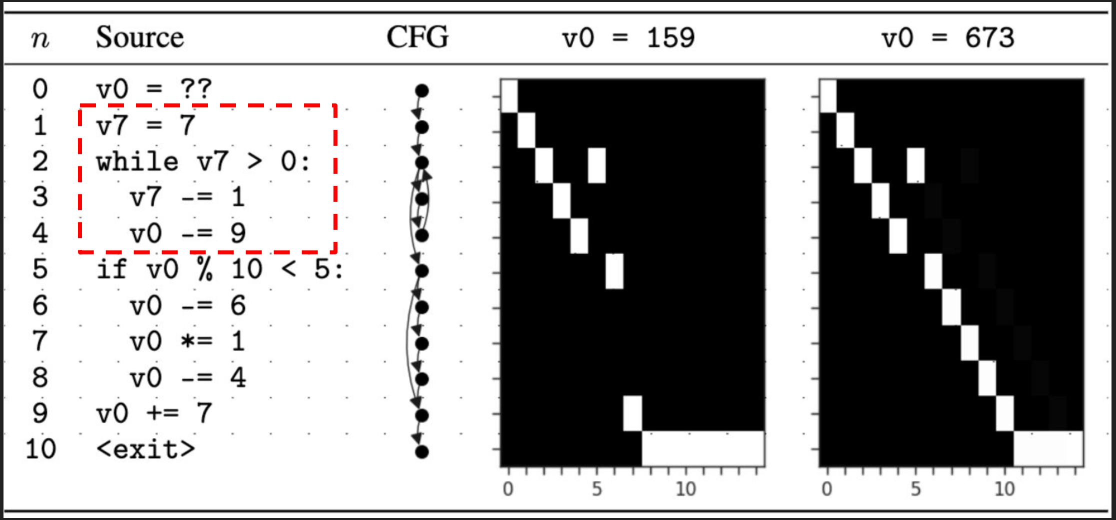
NoExecute significantly outperforms NoControl, indicating the importance of instruction pointer attention for the IPA-GNN model.

Visualization of IPA $p_{t,n}$



- Learns to frequently produce discrete branch decisions.
- Learns to short-circuit execution
- Attends only to the path relevant to the program’s result.

Visualization of IPA $p_{t,n}$



- Learns to frequently produce discrete branch decisions.
- Learns to short-circuit execution
- Attends to the while-loop body only once rather than 7 times.

Outline

- Background
- Problem Formulation
- Approach
- Experiment
- Conclusion

Takeaways

- Learning-to-execute is a challenging ML task for static analysis.
- The authors proposed a new architecture IPA-GNN for this task, inspired by both RNNs and GNNs.
  - Continuous relaxation of IP-RNNs
  - Adaptation of MPNNs on sequential reasoning
- The key components of IPA-GNN is the modeling of Instruction Pointer Attention.
- The proposed method shows systematic generalization (on longer programs).

Thank you for listening!