
Learning to Execute Programs with Instruction Pointer Attention Graph Neural Networks

— Hemil Desai, Zixiang Chen, Jiafan He —

Static analysis

Static analysis is the process of analyzing a program without executing it

- code completion
- bug finding
- program repair

Use a statement-level control flow graph to analyze the static, where nodes represent individual statements in the source program.

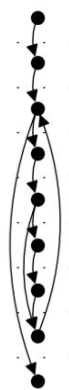
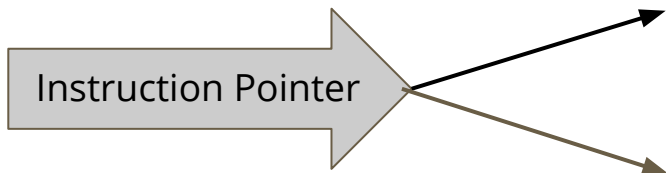
n	Source	Tokenization (x_n)				Control flow graph ($n \rightarrow n'$)	$N_{in}(n)$	$N_{out}(n)$
0	v0 = 23	0	=	v0	23		\emptyset	$\{1\}$
1	v1 = 6	0	=	v1	6		$\{0\}$	$\{2\}$
2	while v1 > 0:	0	while	>	v1 0		$\{1, 7\}$	$\{3, 8\}$
3	v1 -= 1	1	-=	v1	1		$\{2\}$	$\{4\}$
4	if v0 % 10 <= 3:	1	if	<=	% v0 3		$\{3\}$	$\{5\}$
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>	-	-	-	-		$\{2, 8\}$	$\{8\}$

Figure 1: 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.



Combining GNN and RNN

GNN can leverage local program structure to complete static analysis tasks

- Parse trees
- Control flow graphs
- Data flow graphs

Not suited for reasoning
about program execution !

RNN suited for sequential reasoning, but provide no mechanism for learning about complex program structures.

Learning to Execute as Static Analysis

Access: Textual source of a program

May Access: Parse tree of the program, Common static analysis results(program's control flow graph)

Not Access: Compiler or interpreter for the source language, dependencies, a test suite, other unavailable artifacts.

Full and Partial Program execution

Full Program Execution receives a full program as input and must determine some semantic property of the program, such as the program's output.

Canonical, sequential steps of reasoning

Partial program execution task is similar to the full program execution task, except part of the program has been masked.

Benefit the requirements of designing a heuristic function

Bounded Execution

Bounded execution, restricting the model to use fewer steps than are required by the ground truth trace(Default). Choose number of steps allowed such that each loop body may be executed at least twice.

Benefits:

This forces the model to learn short-cuts in order to predict the result in the allotted steps.

In section 5, we find that the bounded execution IPA-GNN achieves better performance on certain length programs.

Limited Complexity Training

Strategy: Train on programs with limited complexity and test on more complex programs

- To learn something meaningful about the language semantics and avoid overfitting
- Execution traces of real-world programs are very long, on the order of thousands or even millions of steps, which is very challenging.

Notation

- Dataset \mathbf{D} consisting of pairs (\mathbf{x}, \mathbf{y}) ; \mathbf{x} denotes a program, and \mathbf{y} denotes some semantic property of the program, such as the program's output.
- A complexity function $\mathbf{c}(\mathbf{x})$ measures the complexity of the program \mathbf{x} .
- Dataset is partitioned according to $\mathbf{c}(\mathbf{x})$. $\mathbf{D}_{\text{train}}$ only consists only of examples (\mathbf{x}, \mathbf{y}) with $\mathbf{c}(\mathbf{x}) < \mathbf{C}$ while \mathbf{D}_{test} consists of those examples (\mathbf{x}, \mathbf{y}) with $\mathbf{c}(\mathbf{x}) > \mathbf{C}$.
- Define \mathbf{x}_n as a statement comprising the program, with \mathbf{x}_0 denoting the start statement.

Notation

- $\mathbf{N}_{in}(\mathbf{n})$ denotes the set of statements that can immediately precede \mathbf{x}_n according to the control flow graph, while $\mathbf{N}_{out}(\mathbf{n})$ denotes the set of statements that can immediately follow \mathbf{x}_n
- We further define $\mathbf{N}_{All}(\mathbf{n})$ as the full set of neighboring statements to \mathbf{x}_n which is a union of $\mathbf{N}_{in}(\mathbf{n})$ and $\mathbf{N}_{out}(\mathbf{n})$

Since branch decisions are binary, we always have the cardinality of $\mathbf{N}_{out}(\mathbf{n})$ no greater than 2, but we don't have such property for $\mathbf{N}_{in}(\mathbf{n})$ or $\mathbf{N}_{out}(\mathbf{n})$.

Approach - IPA-GNNs

Introduction

- The authors aim to design models that share a causal structure with a classical interpreter which improves systematic generalization
- This leads to the design of Instruction Pointer Attention Graph Neural Networks (IPAGNN)
- IPAGNN takes the form of a message passing GNN

Example of a Simple Program

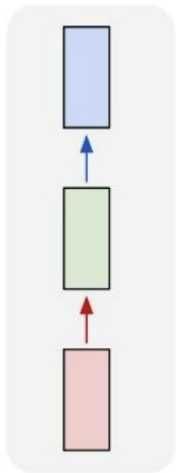
n	Source
0	<code>v0 = 407</code>
1	<code>if v0 % 10 < 3:</code>
2	<code> v0 += 4</code>
3	<code>else:</code>
4	<code> v0 -= 2</code>
5	<code><exit></code>

Classical Interpreters

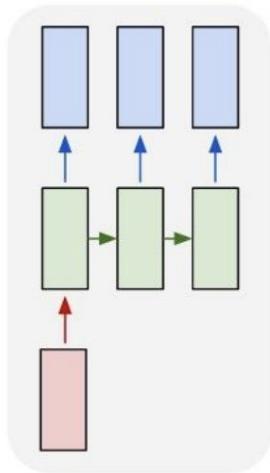
- Exhibit a simple causal structure
- At each step, it maintains:
 - State consisting of the values of all variables
 - Instruction pointer indicating the next statement to execute
- When a statement is executed, the state and instruction pointer is updated
- A natural architecture for modeling this is Recurrent Neural Networks

Background - RNNs

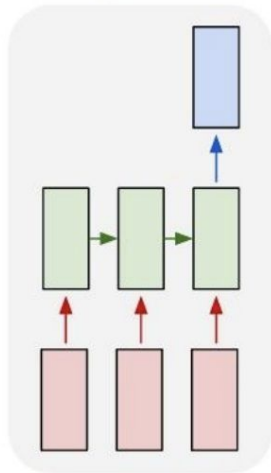
one to one



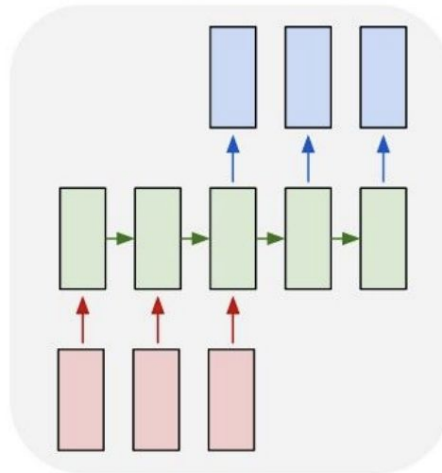
one to many



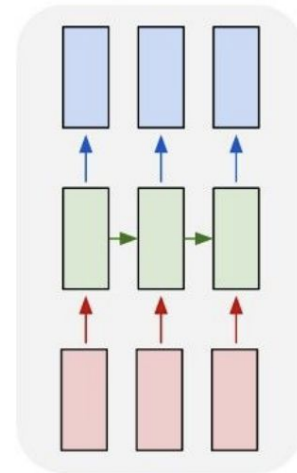
many to one



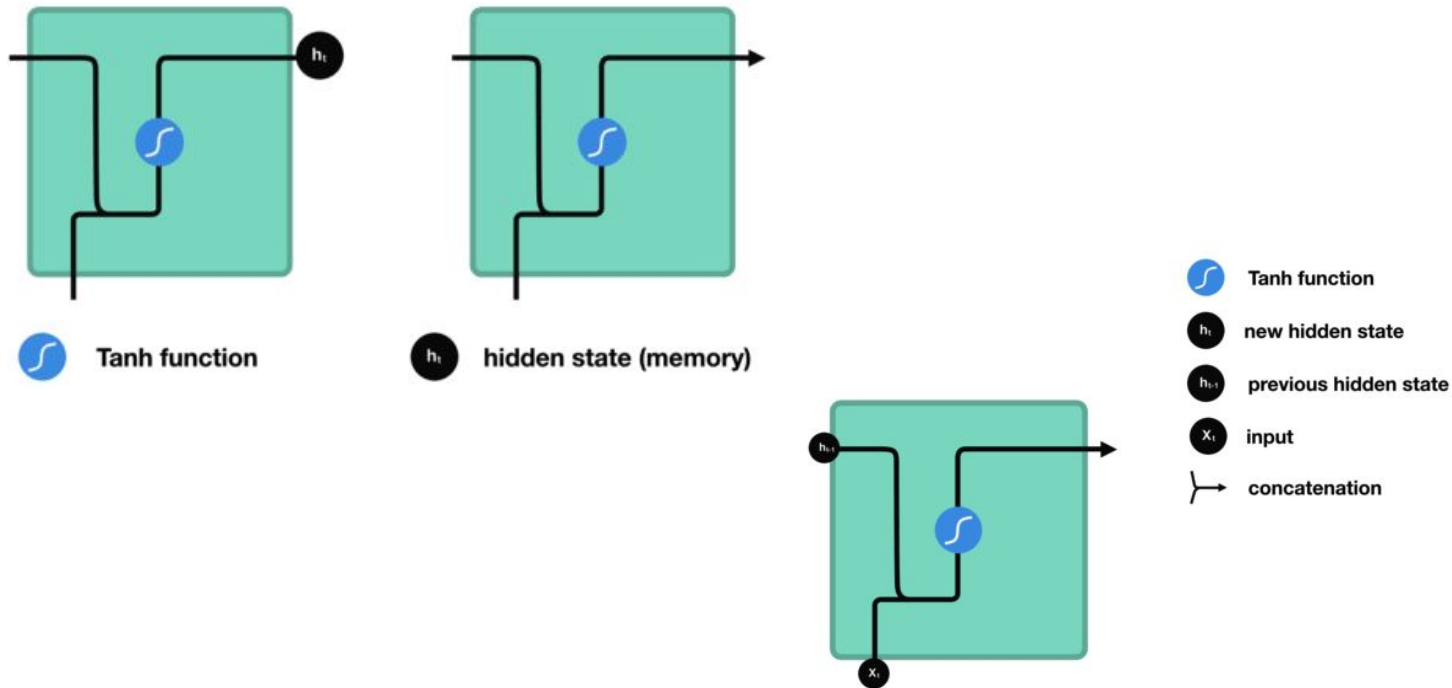
many to many



many to many



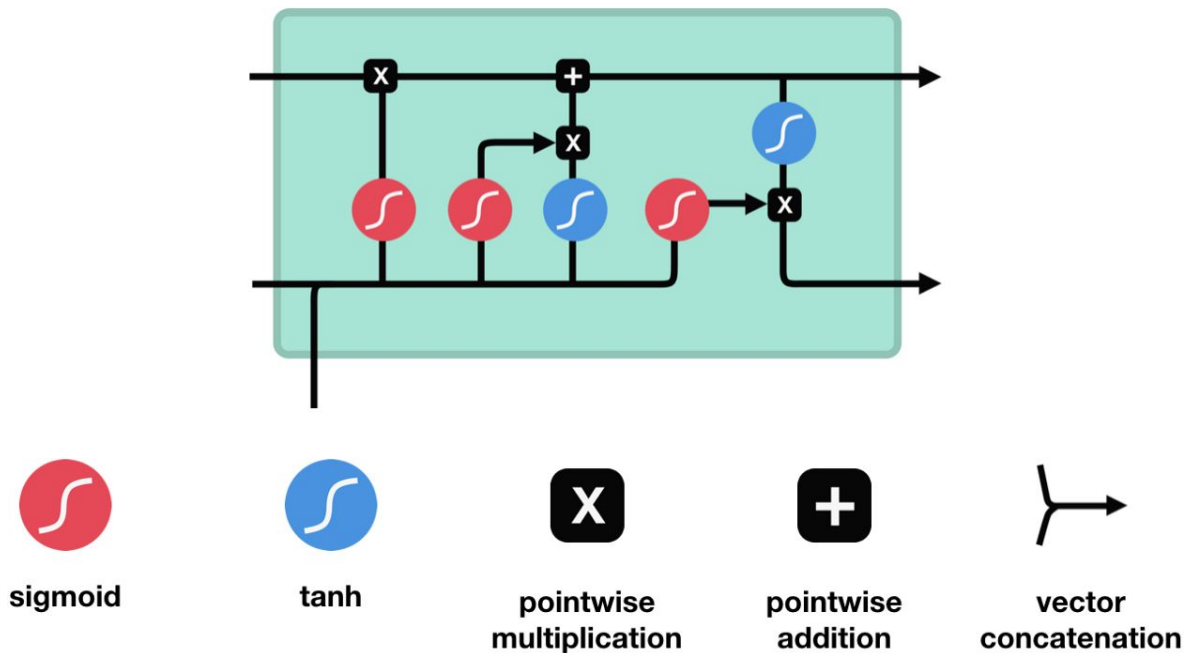
Background - RNNs



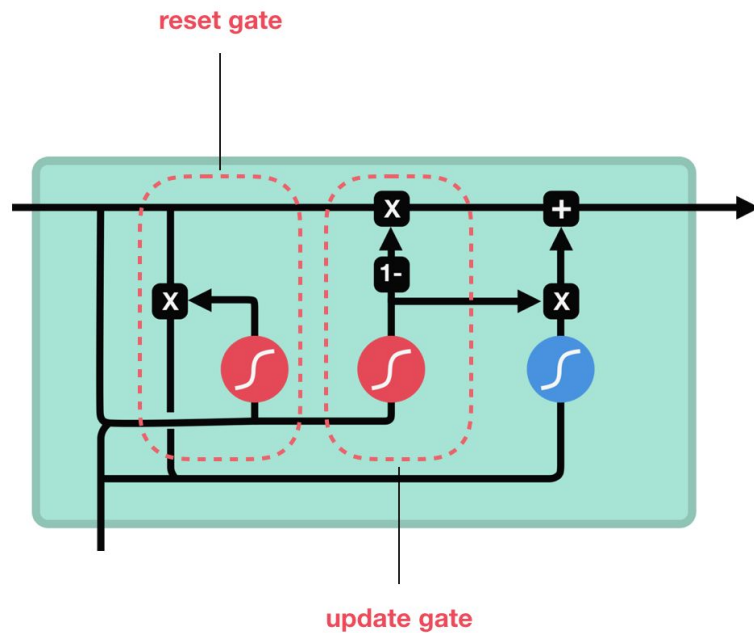
Background - Vanishing Gradients



Background - LSTMs



Background - GRU



IP-RNNs

Notation: t – Step of Interpretation

x_t – Statement at step t

$h_t \in \mathbb{R}^H$ – Hidden State at step t (analogous to the state of variables in the interpreter)

n_t – Instruction Pointer

Line by Line RNN: $h_t = \text{RNN}(h_{t-1}, \text{Embed}(x_{n_{t-1}}))$

$$n_t = t$$

Instruction Pointer RNN: A family of RNNs with different definitions for the instruction pointer

Trace RNNs

- When program has control flow statements, the authors introduce latent branch decisions to IP-RNNs
- RNN processes some path through the program determined by the latent branch decisions
- The simplest model is just an IP-RNN which has access to an oracle that will tell you the correct next statement.
- This oracle has the ground truth trace of the program - $(n_0^*, n_1^*, n_2^*, \dots)$
- Trace RNN is when $n_t = n_t^*$

Hard IP-RNN

- In real world scenarios, you won't have access to such an oracle
- If you use a Dense Layer on top of the hidden state and use hardmax you get the Hard IP-RNN

$$n_t = N_{out}(n_{t-1}) \mid_j \quad \text{where } j = \arg \max(\text{Dense}(h_t))$$

N_{out} is the possible set of statements from n_{t-1}

- However, argmax is non differentiable

IPA-GNNs

- Continuous Relaxation of the Hard IP-RNN
- IPAGNN makes soft branch decisions by modeling a probability distribution over the instruction pointer
- Soft instruction pointer $p_{t,n}$ – Distribution over all possible statements at step t
- Each t,n pair has its specific hidden state $h_{t,n} \in \mathbb{R}^H$

Execute and Branch

- An RNN step over each statement n produces a respective state proposal

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

- In straight line code, $|N_{out}(x_n) = 1|$, $n \rightarrow n'$ gives $h_{t,n'} = a_{t,n}^{(1)}$
- In general, model computes a soft branch decision

$b_{t,n,n'}$ (which is 1 in case of straight line code)

- When $|N_{out}(x_n) = 2|$, $N_{out}(x_n) = \{n_1, n_2\}$

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

$b_{t,n,:} = 0$ for all other statements

Aggregate

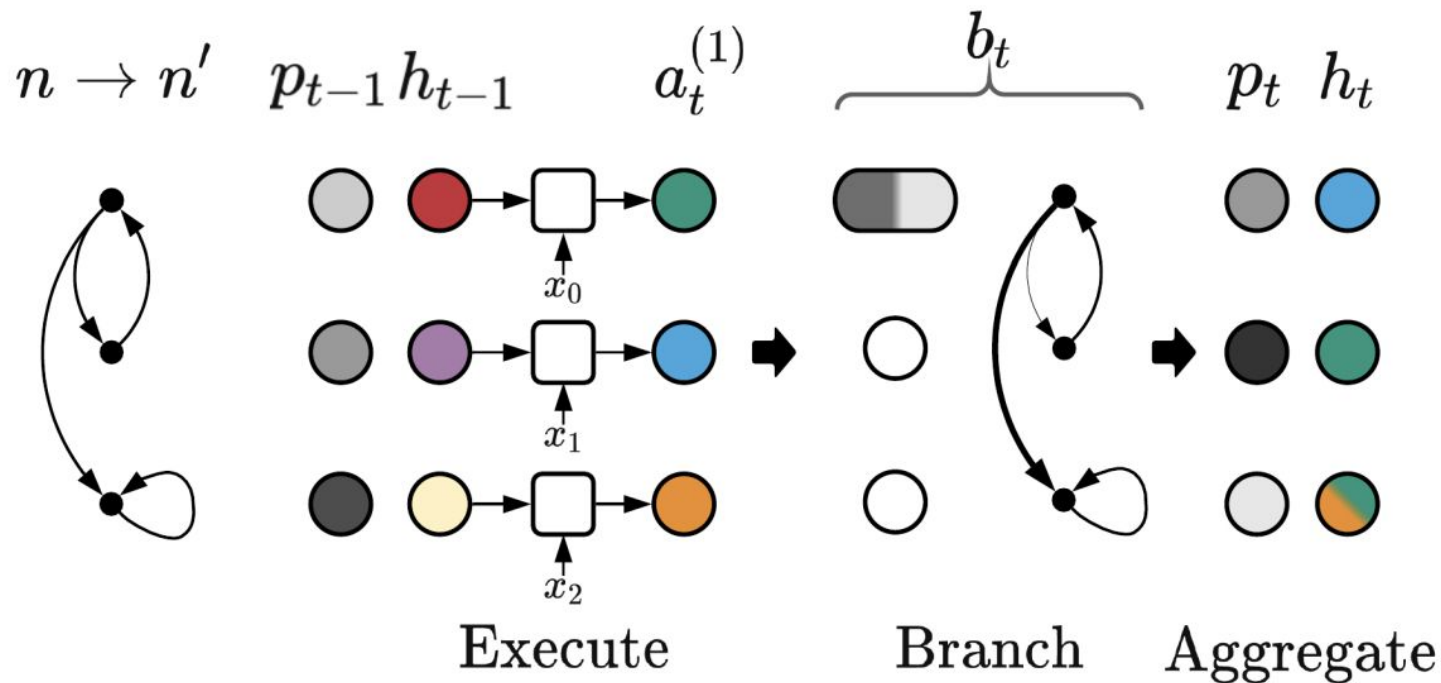
- Hidden State

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

- Soft Instruction Pointer

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

Single IPA-GNN layer



Relationship with IP-RNNs

- If soft branch decisions saturate, we get the Hard IP-RNN
$$b_{t,n,n_1}, b_{t,n,n_2} = \text{hardmax}\left(\text{Dense}\left(a_{t,n}^{(1)}\right)\right)$$
- If Hard IP-RNN makes the correct decisions, we get Trace RNN
$$b_{t,n,n'} = \text{Indicator}\{n_{t-1}^* = n \text{ and } n_t^* = n'\}$$
- If program is straight line code, we get line by line RNN

Relation with Gated Graph Neural Networks (GGNN)

- GGNN operate on the bidirectional form of the control flow graph
- 4 possible edge types - True or False branch, Forward or Reverse edge
- Two changes in IPA-GNN from GGNN:
 - RNN over a statement analogous to the execution of one line of code
 - Soft instruction pointer and attention analogous to the control flow of an interpreter
- Selectively replacing these components yields networks in between the two

Relation with Gated Graph Neural Networks (GGNN)

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

Blue - IPA-GNN Orange - GGNN

Implementation Details

- Final hidden state is computed as $h_{final} = h_{T(x), n_{exit}}$
 n_{exit} is the index of the exit statement
 $T(x)$ is the number of neural network layers

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





































$\text{LoopNesting}(i)$ denotes the number of loops with loop body including statement x_i

- Logits and loss is calculated as

$$s = \text{softmax}(\text{Dense}(h_{final}))$$

$$L = - \sum_i^K 1_{y=i} \{\log(s_i)\}$$

Recap

n	Source	Control flow graph	Line-by-Line RNN	Trace RNN	Hard IP-RNN	IPA-GNN	GGNN
0	<code>v0 = 407</code>						
1	<code>if v0 % 10 < 3:</code>						
2	<code> v0 += 4</code>						
3	<code>else:</code>						
4	<code> v0 -= 2</code>						
5	<code><exit></code>						

Experiment

1. Data generation
2. Model training
3. Experiment result

Data set:

- Limited Python programming language
- Generated by a probabilistic grammar

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

n	Source
0	<code>v0 = 36</code>
1	<code>if v0 % 10 >= 7:</code>
2	<code> v0 *= 3</code>
3	<code> if v0 % 10 > 3:</code>
4	<code> v0 *= 4</code>
5	<code> v5 = 3</code>
6	<code> while v5 > 0:</code>
7	<code> v5 -= 1</code>
8	<code> break</code>
9	<code> v0 *= 2</code>
10	<code><exit></code>

Figure 6: Grammar describing the generated programs comprising the datasets in this paper.

Grammar and Data Generation

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 $O := > \mid < \mid >= \mid <=$
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 $M := 0 \mid 1 \mid 2 \mid \dots \mid 999$

n	Source
0	<code>v0 = 36</code>
1	<code>if v0 % 10 >= 7:</code>
2	<code> v0 *= 3</code>
3	<code> if v0 % 10 > 3:</code>
4	<code> v0 *= 4</code>
5	<code> v5 = 3</code>
6	<code> while v5 > 0:</code>
7	<code> v5 -= 1</code>
8	<code> break</code>
9	<code> v0 *= 2</code>
10	<code><exit></code>

Program
P

Grammar and Data Generation

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 $O := > \mid < \mid >= \mid <=$
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 $M := 0 \mid 1 \mid 2 \mid \dots \mid 999$

n	Source
0	<code>v0 = 36</code>
1	<code>if v0 % 10 >= 7:</code>
2	<code> v0 *= 3</code>
3	<code> if v0 % 10 > 3:</code>
4	<code> v0 *= 4</code>
5	<code> v5 = 3</code>
6	<code> while v5 > 0:</code>
7	<code> v5 -= 1</code>
8	<code> break</code>
9	<code> v0 *= 2</code>
10	<code><exit></code>

Program

I B (rule P:= I B)

Grammar and Data Generation

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 $O := > \mid < \mid >= \mid <=$
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 $M := 0 \mid 1 \mid 2 \mid \dots \mid 999$

n	Source
0	<code>v0 = 36</code>
1	<code>if v0 % 10 >= 7:</code>
2	<code> v0 *= 3</code>
3	<code> if v0 % 10 > 3:</code>
4	<code> v0 *= 4</code>
5	<code> v5 = 3</code>
6	<code> while v5 > 0:</code>
7	<code> v5 -= 1</code>
8	<code> break</code>
9	<code>v0 *= 2</code>
10	<code><exit></code>

Program

`v0 = M (rule I:= v0 = M)`

`B`

Grammar and Data Generation

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 $O := > \mid < \mid >= \mid <=$
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 $M := 0 \mid 1 \mid 2 \mid \dots \mid 999$

probabilistic grammar

$M = 0 \mid 1 \mid 2 \mid \dots \mid 999$

<i>n</i>	Source	Program
0	<code>v0 = 36</code>	<code>v0 = 36 (rule M:= 36)</code>
1	<code>if v0 % 10 >= 7:</code>	<code>B</code>
2	<code> v0 *= 3</code>	
3	<code> if v0 % 10 > 3:</code>	
4	<code> v0 *= 4</code>	
5	<code> v5 = 3</code>	
6	<code> while v5 > 0:</code>	
7	<code> v5 -= 1</code>	
8	<code> break</code>	
9	<code>v0 *= 2</code>	
10	<code><exit></code>	

Grammar and Data Generation

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 $O := > \mid < \mid >= \mid <=$

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 $M := 0 \mid 1 \mid 2 \mid \dots \mid 999$

probabilistic grammar

$B := B S \mid S$

<i>n</i>	Source
0	<code>v0 = 36</code>
1	<code>if v0 % 10 >= 7:</code>
2	<code> v0 *= 3</code>
3	<code> if v0 % 10 > 3:</code>
4	<code> v0 *= 4</code>
5	<code> v5 = 3</code>
6	<code> while v5 > 0:</code>
7	<code> v5 -= 1</code>
8	<code> break</code>
9	<code> v0 *= 2</code>
10	<code><exit></code>

Program

`v0 = 36 (rule M:= 36)`

`B S (rule B:= B S)`

```
def generate_block(indent, in_loop=False):
    global length
    if length > COMPLEXITY - 1:
        return generate_statement(indent,
                                   in_loop=in_loop)
```

```
    b = random.randint(0, 1)
    if b == 0:
        s, in_loop = generate_block(indent,
                                     in_loop=in_loop)
        new_s, in_loop =
generate_statement(indent,
in_loop=in_loop)
        s += new_s
        return s, in_loop
    else:
        return generate_statement(indent,
                                   in_loop=in_loop)
```

Grammar and Data Generation

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 $O := > \mid < \mid >= \mid <=$
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 $M := 0 \mid 1 \mid 2 \mid \dots \mid 999$

Random variable v_1, \dots, v_9 for Repeat statement

n	Source
0	<code>v0 = 36</code>
1	<code>if v0 % 10 >= 7:</code>
2	<code> v0 *= 3</code>
3	<code> if v0 % 10 > 3:</code>
4	<code> v0 *= 4</code>
5	<code> v5 = 3</code>
6	<code> while v5 > 0:</code>
7	<code> v5 -= 1</code>
8	<code> break</code>
9	<code> v0 *= 2</code>
10	<code><exit></code>

Statement $S = \text{Repeat}(N, B)$
 $\text{Repeat}(3, B)$

Grammar and Data Generation

```
elif s == 3:
    length += 3
    var = random.randint(1, 9)
    block, _ = generate_block(indent + 1, in_loop=True)
    p = (
        " " * indent
        + f"v{var} = {random.randint(0, 9)}\n"
        + " " * indent
        + f"while v{var} > 0:\n"
        + " " * (indent + 1)
        + f"v{var} -= 1\n{n}{block}"
    )
    return p, in_loop
```

Random variable v_1, \dots, v_9 for Repeat statement

<i>n</i>	Source
0	v0 = 36
1	if v0 % 10 >= 7:
2	v0 *= 3
3	if v0 % 10 > 3:
4	v0 *= 4
5	v5 = 3
6	while v5 > 0:
7	v5 -= 1
8	break
9	v0 *= 2
10	<exit>

Statement S = Repeat (N,B)
Repeat (3,B)

Data set:

- Limited Python programming language
- Generated by a probabilistic grammar

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

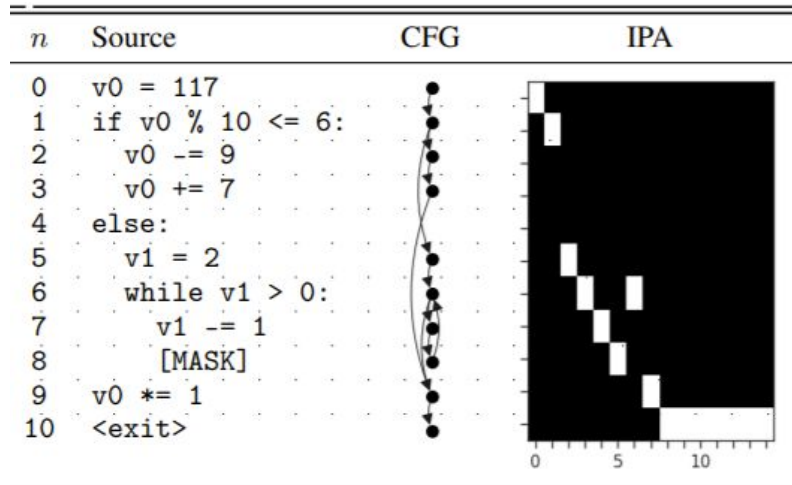
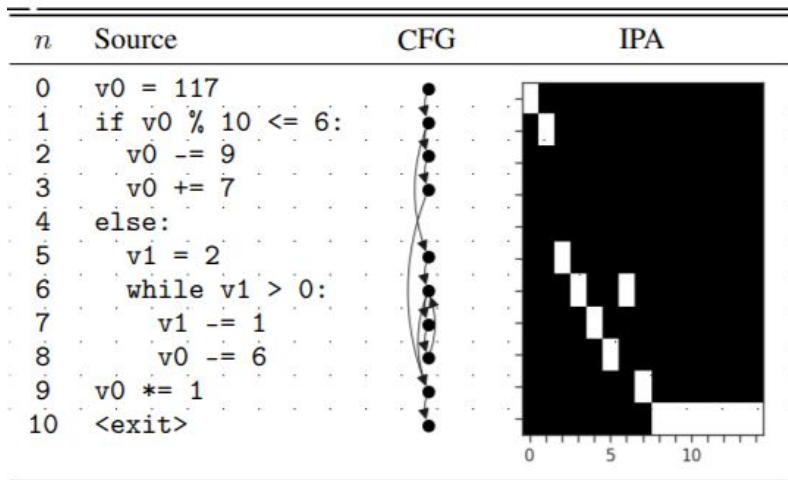
Limited Statement

1. Multi-digit arithmetic (+, -, *) for v_0 (0, 1, ..., 999)
2. While-loops with variable (v_1, \dots, v_9) (0, 1, ..., 9 iterations)
3. If-else statement with variable V_0 (>, <, >=, <=)

Figure 6: Grammar describing the generated programs comprising the datasets in this paper.

Partial program

1. Masking one statement
2. Uniformly random
3. Non-control flow statements (remain same execution behavior)



Limited Complexity Training

Strategy: Train on programs with limited complexity and test on more complex programs

1. Complexity measure $c(x)$: program length
2. Complexity threshold for training data: $c(x)=10$ (5M examples)
3. Complexity threshold for test data: $c(x)=\{20,30,\dots,100\}$ (4.5K)
4. Target function $y=f(x)$: the final value of $v_0 \pmod{1000}$

Training and Experiment results

1. Adam optimizer

+ standard cross-entropy loss

2. Training with different parameter and choose the best model parameters by cross-validation

Hidden layer : 200,300

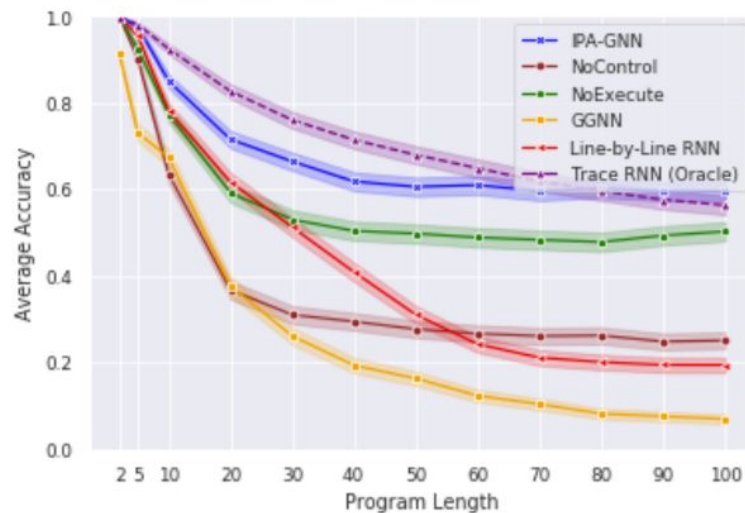
Learning rate: 0.003, 0.001, 0.0003, 0.0001

Table 2: Accuracies on D_{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)	62.1	29.1

Full programs with different length

1. Trace RNN outperforms all model except IPA-GNN
2. IPA-GNN outperforms all other models
3. Line-by-Line RNN model performs almost as well as the IPA-GNN at low complexity

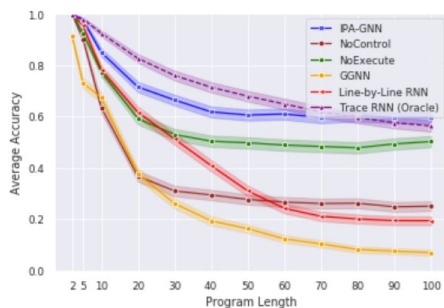


(a) Execution of full programs

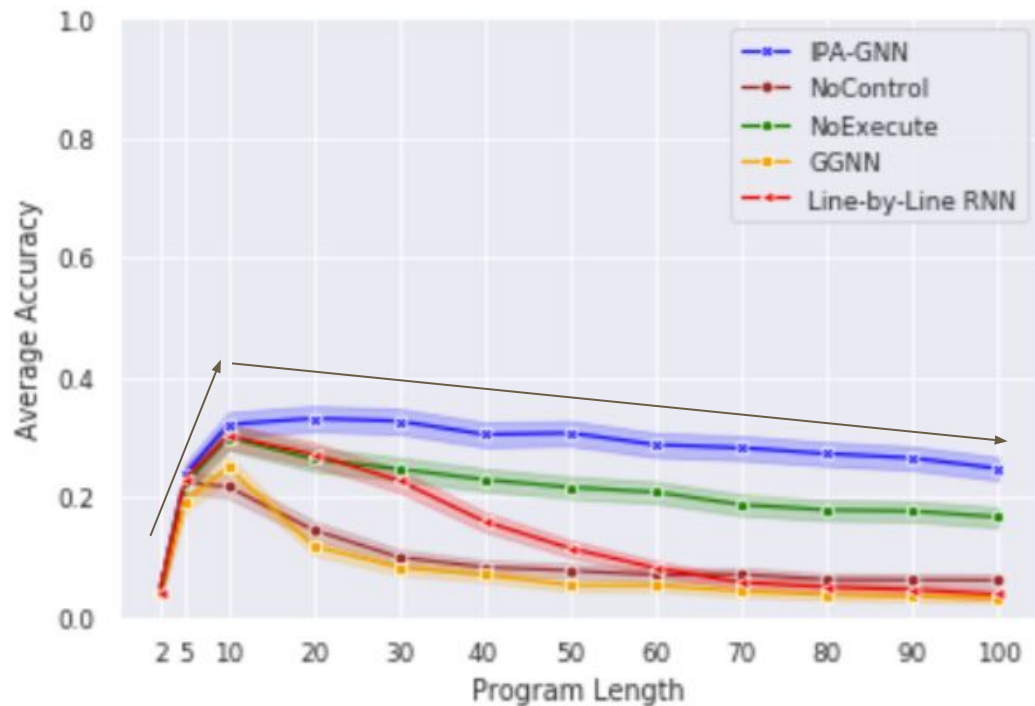
Partial programs with different length

1. Rapid rise (length < 10)



2. Slowly decrease (10 < length)

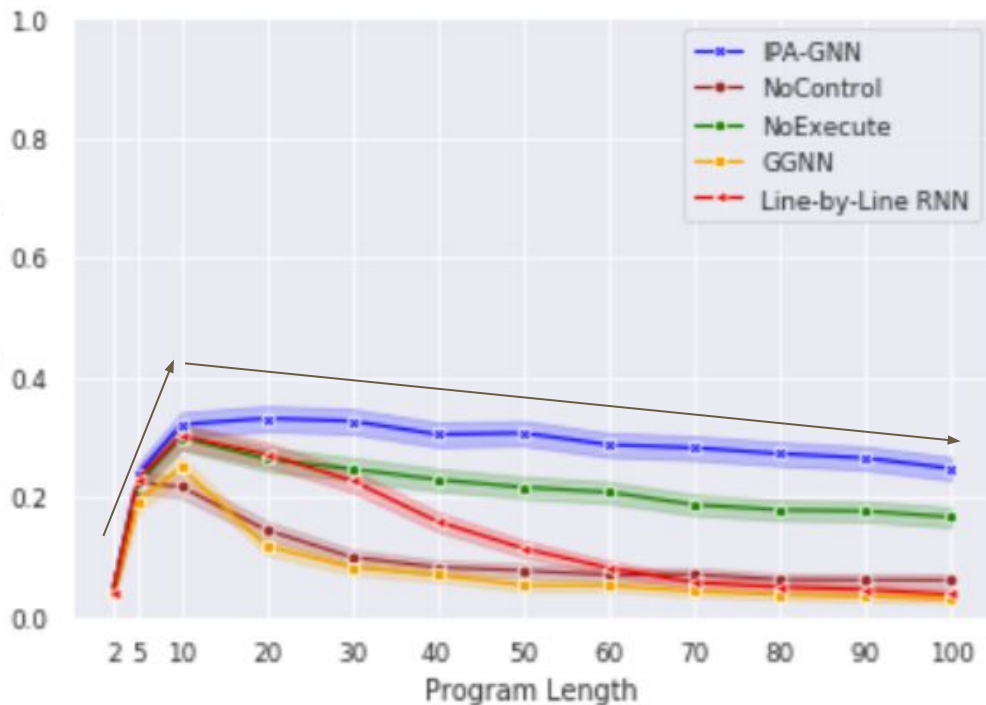


(a) Execution of full programs



Partial programs with different length

<i>n</i>	Source	CFG	IPA
0	<code>v0 = 36</code>		
1	<code>if v0 % 10 >= 7:</code>		
2	<code>[MASK]</code>		
3	<code>if v0 % 10 > 3:</code>		
4	<code> v0 *= 4</code>		
5	<code> v5 = 3</code>		
6	<code> while v5 > 0:</code>		
7	<code> v5 -= 1</code>		
8	<code> break</code>		
9	<code>v0 *= 2</code>		
10	<code><exit></code>		



Other Observation of IPA-GNN

1. Discrete branch decisions

2. Short-circuit execution

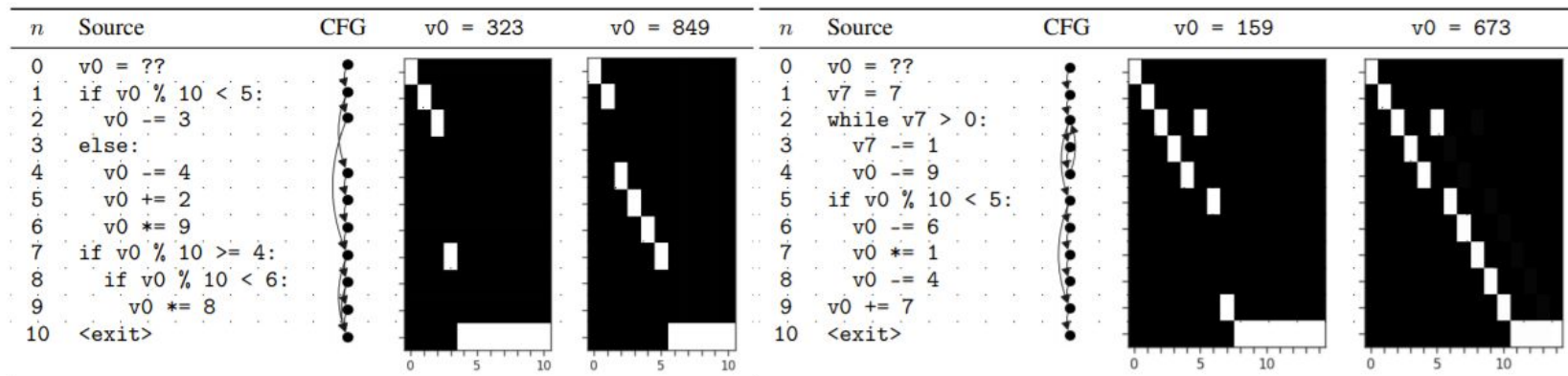


Figure 5: **Instruction Pointer Attention.** Intensity plots show the soft instruction pointer $p_{t,n}$ at each step of the IPA-GNN on two programs, each with two distinct initial values for $v0$.

Thanks for Listening!