Making Neural Programming Architectures Generalize via Recursion

Jonathon Cai, Richard Shin, Dawn Song

University of California, Berkeley
Program Synthesis

Example Applications:

- End-user programming
- Performance optimization of code
- Virtual assistant
Neural Program Synthesis

<table>
<thead>
<tr>
<th>Training data</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>612</td>
<td>979</td>
</tr>
<tr>
<td>234</td>
<td>367</td>
<td></td>
</tr>
<tr>
<td>357</td>
<td>797</td>
<td></td>
</tr>
</tbody>
</table>
Neural Program Synthesis

Training data:
- 123
- 234
- 357

Input:
- 452
- 345
- 612
- 367

Output:
- 797
- 979

Neural Program Architecture

Test input:
- 50
- 70

Learned neural program

Test output:
- 120
Neural Program Architectures

Neural Turing Machine (Graves et al)

Stack Recurrent Nets (Joulin et al)

Learning Simple Algorithms from Examples (Zaremba et al)

Reinforcement Learning Neural Turing Machines (Zaremba et al)

Neural Programmer (Neelakantan et al)

Neural Programmer-Interpreter (Reed et al)

Neural GPU (Kaiser et al)

Differentiable Neural Computer (Graves et al)

Neural Program Synthesis Tasks: Copy, Grade-school addition, Sorting, Shortest Path
Challenge 1: Generalization

Training data:
- 123
- 234
- 357

Input:
- 452
- 345
- 612
- 367
- 979

Output:
- 797

Neural Program Architecture

Test input:
- 34216
- 24320

Test output:
- 54321

Learned neural program

length = 5

length = 3
Challenge 1: Existing Neural Program Architectures Do Not Generalize Well

Hypothesis: Spurious dependencies on length in the training data

- NPI (Reed et al, 2016)
  - Trained up to: length 20
  - Fails from: length 60

- Stack Recurrent Nets (Joulin et al, 2015)
  - Trained up to: length 20
  - Fails from: length 20
Challenge 2: No Proof of Generalization

Training data
- 123
- 234
- 357

Input
- 452
- 345
- 612
- 367
- 979

Output
- 797
- 979

Test input
- 34216
- 24320

Neural Program Architecture

Test output
- 58536
Problem Statement

For program synthesis tasks like addition and sorting:

- What challenges are we trying to address?
  - Generalization to more complex inputs
  - Proof of generalization

→ Which approach will solve these challenges?
→ How do we implement the approach?
Our Approach: Introduce Recursion

Learn recursive neural programs
Recursion

Fundamental concept in Computer Science and Math.

Solve whole problem by reducing it to smaller subproblems (*reduction rules*).

*Base cases* (smallest subproblems) are easier to reason about.
Our Contributions

For program synthesis tasks like addition and sorting:

● What challenges are we trying to address?
  ✓ Generalization to more complex inputs
  ✓ Proof of generalization

● Which approach will solve these challenges?
  ○ Recursion in neural programs

● How do we implement the approach?
  ○ Instantiation: Incorporate recursion into Neural Programmer-Interpreter
  ○ Training method: As a first step, strong supervision with \textit{explicitly recursive execution traces} to learn a recursive neural program

Main Contribution!
Outline

Challenges in Neural Program Architectures
Overview of Our Approach: Recursion

→ Background: Neural-Programmer Interpreter
  Learning Recursive Neural Programs
  Provably Perfect Generalization
Experimental Results
Conclusion
Neural Program Architectures

Neural Turing Machine (Graves et al)

Stack Recurrent Nets (Joulin et al)

Learning Simple Algorithms from Examples (Zaremba et al)

Reinforcement Learning Neural Turing Machines (Zaremba et al)

Neural Programmer (Neelankatan et al)

Neural Programmer-Interpreter (Reed et al)

Neural GPU (Kaiser et al)

Differentiable Neural Computer (Graves et al)

Neural Program Synthesis Tasks: Copy, Grade-school addition, Sorting, Shortest Path
Neural Programmer-Interpreter (NPI)
Execution of NPI

Calling a function creates a new NPI controller state (LSTM hidden state).
Execution traces in NPI

The sequence of operations forms an execution trace.

The sequence of operations forms an execution trace.
Training NPI with Execution Traces

Execution trace divided into training sequences, according to the caller function.

\[ j_0 = 0 \quad \rightarrow \quad j_1 \]

\[ \text{obs}_2 \rightarrow \text{MOVE} \quad \text{obs}_3 \rightarrow \text{return} \]

\[ r_0 = 0 \quad \rightarrow \quad r_1 \]

\[ \text{obs}_1 \rightarrow \text{F} \quad \rightarrow \quad \text{obs}_4 \rightarrow \text{H} \]

\[ \Rightarrow: \text{change environment} \quad \Rightarrow: \text{call function} \]
Simplified Execution Traces

For brevity, we omit details in the trace.

sequence for G

\[
\begin{align*}
  j_0 &= 0 \\
  j_1 &
\end{align*}
\]

\[
\begin{align*}
  obs_2 G \\
  obs_3 G 
\end{align*}
\]

\(\Rightarrow\) MOVE \(\uparrow\) return

sequence for F

\[
\begin{align*}
  r_0 &= 0 \\
  r_1 &
\end{align*}
\]

\[
\begin{align*}
  obs_1 F \\
  obs_4 F 
\end{align*}
\]

\(\Rightarrow\): change environment \(\Rightarrow\): call function
Simplified Execution Traces

For brevity, we omit details in the trace.
NPI Trains on Execution Traces, Not Input-Output Pairs

The training data for each architecture:

<table>
<thead>
<tr>
<th>Input</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>452</td>
<td>612</td>
<td></td>
</tr>
<tr>
<td>234</td>
<td>345</td>
<td>367</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>357</td>
<td>797</td>
<td>979</td>
<td></td>
</tr>
</tbody>
</table>

Neural Turing Machine
Neural GPU
Differentiable Neural Computer
etc.

NPI
Outline

Challenges in Neural Program Architectures
Overview of Our Approach: Recursion
Background: Neural-Programmer Interpreter

Learning Recursive Neural Programs
Provably Perfect Generalization
Experimental Results
Conclusion
Learn recursive neural programs

Incorporate recursion into NPI
What is a Recursive NPI Program?

Trace from an example recursive NPI program: ADD calls itself

```
ADD
  ADD1  LSHIFT
  ADD1
  LSHIFT
  ADD1
  LSHIFT
...
```

Repeated inside one function call:

```
ADD
  ADD1
  LSHIFT
  ADD
    ADD1
    LSHIFT
    ADD
...
```

Execution trace of non-recursive program (previous work)

Execution trace of recursive program (our work)
Grade-School Addition

From right to left (smallest to largest position):

1. Add three values in the column.
2. If resulting sum exceeds 10, put a 1 in the next carry position.

```
  1  1  1  
+ 1 2 3 4  
+ 5 6 7 8  
_______
  6 9 1 2  
```

1st number carry
2nd number output

NPI (Reed et al, 2016)
Grade-School Addition

Scratchpad (environment):

<table>
<thead>
<tr>
<th>INP1</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>INP2</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>CARRY</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Observation: value at each pointer; in this example, (4, 8, ∅, ∅)

Three functions

- ADD1: adds 1 column
- LSHIFT: move to next column
- CARRY: write carry digit if needed

NPI (Reed et al, 2016)
Non-Recursive Grade-School Addition

✎: change environment ☎: call function

بيب ADD
بيب ADD1

NPI (Reed et al, 2016)
Non-Recursive
Grade-School Addition

 nok: change environment  nok: call function

ADD
ADD1
WRITE OUT 2

INP1
INP2
CARRY
OUT

NPI (Reed et al, 2016)
Non-Recursive
Grade-School Addition

__: change environment  ☎: call function

☎ ADD
☎ ADD1
✎ WRITE OUT 2
☎ CARRY
✎ PTR CARRY LEFT

NPI (Reed et al, 2016)
Non-Recursive
Grade-School Addition

キー：
- ADD
- ADD1
  - WRITE OUT 2
  - CARRY
    - PTR CARRY LEFT
    - WRITE CARRY 1

NPI (Reed et al, 2016)
Non-Recursive Grade-School Addition

✎: change environment  ☎: call function

☎ ADD
☎ ADD1
✎ WRITE OUT 2
☎ CARRY
✎ PTR CARRY LEFT
✎ WRITE CARRY 1
✎ PTR CARRY RIGHT

NPI (Reed et al, 2016)
Non-Recursive
Grade-School Addition

☎: change environment  📞: call function

☎ ADD
☎ ADD1
☎ LSHIFT
   ⇛ PTR INP1 LEFT
   ⇛ PTR INP2 LEFT
   ⇛ PTR CARRY LEFT
   ⇛ PTR OUT LEFT

INP1
INP2
CARRY
OUT

NPI (Reed et al, 2016)
Non-Recursive

Grade-School Addition

✎: change environment  📞: call function

☎ ADD
☎ ADD1
☎ LSHIFT

INP1
INP2
CARRY
OUT

NPI (Reed et al, 2016)
Non-Recursive
Grade-School Addition

生产总的环境转换：

ADD
ADD1
LSHIFT
ADD1
LSHIFT

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>INP1</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>INP2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>CARRY</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td>OUT</td>
</tr>
</tbody>
</table>

NPI (Reed et al, 2016)
Non-Recursive
Grade-School Addition

✎: change environment  🎨: call function

ADD
ADD1
LSHIFT
ADD1
LSHIFT
ADD1
LSHIFT

INP1
INP2
CARRY
OUT

NPI (Reed et al, 2016)
Non-Recursive
Grade-School Addition

✎: change environment  📏: call function

ADD
ADD1
LSHIFT
ADD1
LSHIFT
ADD1
LSHIFT
ADD1
LSHIFT

INP1
INP2
CARRY
OUT

NPI (Reed et al, 2016)
Non-Recursive
Grade-School Addition

.shiro: change environment  🌐: call function

ADD
ADD1
LSHIFT
ADD1
LSHIFT
ADD1
LSHIFT
ADD1
LSHIFT

Repeated x4 in one call

INP1
INP2
CARRY
OUT

NPI (Reed et al, 2016)
Non-Recursive vs Recursive Grade-School Addition

monicograph: change environment
monicograph: call function

Non-recursive (previous work)

مبادئية ADD

مبادئية ADD1

مبادئية LSHIFT

Recursive (our work)

مبادئية ADD

مبادئية ADD1

مبادئية LSHIFT

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

INP1

INP2

CARRY

OUT
Non-Recursive vs Recursive
Grade-School Addition

Non-recursive (previous work)

- ADD
  - ADD1
  - LSHIFT
  - ADD1
  - LSHIFT

Recursive (our work)

- ADD
  - ADD1
  - LSHIFT
  - ADD
  - ADD1
  - LSHIFT

INP1
INP2
CARRY
OUT

✎: change environment
☎: call function
# Non-Recursive vs Recursive Grade-School Addition

**Non-recursive** (previous work)

- ADD
- ADD1
- LSHIFT
- ADD1
- LSHIFT
- ADD1
- LSHIFT

**Recursive** (our work)

- ADD
- ADD1
- LSHIFT
- ADD1
- LSHIFT
- ADD
- ADD1
- LSHIFT
- ADD
- ADD1
- LSHIFT

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

- : change environment
- : call function

<table>
<thead>
<tr>
<th></th>
<th>INP1</th>
<th>INP2</th>
<th>CARRY</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>40</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Recursive Grade-School Addition

- ADD
- ADD1
- LSHIFT
- ADD
- ADD1
- LSHIFT
- ADD
- ADD1
- LSHIFT
- ADD

<table>
<thead>
<tr>
<th></th>
<th>INP1</th>
<th>INP2</th>
<th>CARRY</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>INP1</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>INP2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>CARRY</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>OUT</td>
</tr>
</tbody>
</table>

خفض اللغة العربية، وليس اللغة الإنجليزية.

中国汽车

<table>
<thead>
<tr>
<th></th>
<th>INP1</th>
<th>INP2</th>
<th>CARRY</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>CARRY</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Recursive Grade-School Addition

_dirty: change environment
_magic: call function

**ADD**
- **ADD1**
- **LSHIFT**
- **ADD**
  - **ADD1**
  - **LSHIFT**
  - **ADD**
    - **ADD1**
    - **LSHIFT**
    - **ADD**
      - ...

```
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
```

INP1
INP2
CARRY
OUT

_recursive calls_
Non-Recursive vs Recursive Addition

Non-recursive execution trace (previous work)

Recursive execution trace (our work)
Non-Recursive vs Recursive Addition

$n = \text{number of input digits}$

1 sequence of size $2n$

$\text{variable length}$

$n$ sequences with length 3

$\text{fixed length}$

✓ Generalization to more complex inputs
✓ Proof of generalization
How to Learn a Recursive NPI Program

- In NPI, any function can call any function, including itself (but original NPI didn’t explicitly make use of recursive calls)

- To learn a recursive NPI program:
  - No architecture change
  - Only change the training data, instead of the architecture
How to Learn a Recursive NPI Program

Non-recursive training trace

Non-recursive NPI program

Recursive training trace

Recursive NPI program
Outline

Challenges in Neural Program Architectures
Overview of Our Approach: Recursion
Background: Neural-Programmer Interpreter
Learning Recursive Neural Programs

→ Provably Perfect Generalization
Experimental Results
Conclusion
Verifying Perfect Generalization

Oracle
(correct program behavior)

 learns

Learned neural program
Verifying Perfect Generalization

Observation Sequences

\[
\begin{align*}
  j_0 &= 0 \\
  r_0 &= 0 \\
  j_1 &= 0 \\
  r_1 &= 0
\end{align*}
\]

\[
\begin{align*}
  \text{obs}_1 &\xrightarrow{\text{F}} \text{obs}_2 \\
  \text{obs}_3 &\xrightarrow{\text{G}} \text{obs}_4 \\
  \text{F} &\xrightarrow{\text{H}} \text{G} \\
  \text{MOVE} &\xrightarrow{\text{H}} \text{F}
\end{align*}
\]
Creating the Verification Set

Verifying Perfect Generalization

All feasible observation sequences

other functions...

ADD1

CARRY
Creating the Verification Set

Verifying Perfect Generalization

Input problems

9 + 0

ADD1

CARRY

other functions...

All feasible observation sequences
Verifying Perfect Generalization

Creating the Verification Set

Input problems

9 + 0
81 + 19

ADD1
CARRY

other functions...

All feasible observation sequences
Verifying Perfect Generalization

Creating the Verification Set

Input problems

9 + 0
81 + 19
61 + 79

ADD1

CARRY

other functions...

All feasible observation sequences
Verifying Perfect Generalization

Creating the Verification Set

Input problems

9 + 0
81 + 19
61 + 79
...

All feasible observation sequences

ADD1
CARRY

other functions...
Creating the Verification Set

Verifying Perfect Generalization

Input problems

Verification set

9 + 0
81 + 19
61 + 79

Full coverage

ADD1
CARRY

other functions...

All feasible observation sequences
Verifying Perfect Generalization

Oracle Matching

Verification set

Oracle

Learned neural program

Output operations (execution trace)
Recursion Induces Boundedness

Neural network needs to solve:

Without recursion (previous work)

With recursion (our work)
Recursion Enables Verification

Recursion allows for a *finite* (and therefore tractable) verification set, for certain domains.

Verification sets for addition:

Without recursion (previous work):

- 1 + 1 = 2
- 99 + 99 = 198
- 99...99 + 99...99 = ...
- ...

With recursion (our work):

- 1 + 1 = 2
- 99 + 99 = 198
- ...

~20000 cases
Outline

Challenges in Neural Program Architectures
Overview of Our Approach: Recursion
Background: Neural-Programmer Interpreter
Learning Recursive Neural Programs
Provably Perfect Generalization

→ Experimental Results
Conclusion
Tasks in Experiments

Grade-School Addition

Bubble Sort

Topological Sort

Quicksort

NEW!
Experimental Results

● Experimental setup:
  ○ Recursive and non-recursive NPI program learned for each task using the same training problems.
  ○ Both evaluated on same (randomly generated) test problems.

● Empirical results:
  ○ Learned recursive programs are 100% accurate on the test problems.
  ○ Non-recursive program accuracy often degrades on the test problems.
## Empirical Accuracy: Quicksort

<table>
<thead>
<tr>
<th>Length of Array</th>
<th>Non-Recursive</th>
<th>Recursive</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>5</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>7</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>11</td>
<td>73.3%</td>
<td>100%</td>
</tr>
<tr>
<td>15</td>
<td>60%</td>
<td>100%</td>
</tr>
<tr>
<td>20</td>
<td>30%</td>
<td>100%</td>
</tr>
<tr>
<td>30</td>
<td>3.33%</td>
<td>100%</td>
</tr>
<tr>
<td>70</td>
<td>0%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Training set: 4 length-5 arrays
Empirical Accuracy: Other Tasks

<table>
<thead>
<tr>
<th>Length</th>
<th>Non-Recursive</th>
<th>Recursive</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>4</td>
<td>10%</td>
<td>100%</td>
</tr>
<tr>
<td>20</td>
<td>0%</td>
<td>100%</td>
</tr>
<tr>
<td>90</td>
<td>0%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Training set: 100 length-2 arrays

<table>
<thead>
<tr>
<th>Vertices</th>
<th>Non-Recursive</th>
<th>Recursive</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6.7%</td>
<td>100%</td>
</tr>
<tr>
<td>7</td>
<td>3.3%</td>
<td>100%</td>
</tr>
<tr>
<td>8</td>
<td>0%</td>
<td>100%</td>
</tr>
<tr>
<td>70</td>
<td>0%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Training set: a graph with 5 vertices

On grade-school addition, both non-recursive and recursive show 100% empirical accuracy (non-recursive matches Reed et al 2016).
Verification of Perfect Generalization

We successfully verified a learned recursive program for each task via the oracle matching procedure.

<table>
<thead>
<tr>
<th>Task</th>
<th>Verification Set Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grade-School Addition</td>
<td>20,181</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>76</td>
</tr>
<tr>
<td>Topological Sort</td>
<td>20</td>
</tr>
<tr>
<td>Quicksort</td>
<td>12</td>
</tr>
</tbody>
</table>
Outline

Challenges in Neural Program Architectures
Overview of Our Approach: Recursion
Background: Neural-Programmer Interpreter
Learning Recursive Neural Programs
Provably Perfect Generalization
Experimental Results

→ Conclusion
Importance of Recursion in Neural Program Architectures

● We introduce recursion, for the first time, into neural program architectures, and learn recursive neural programs

Main Contribution!

● We address two main challenges using recursion:
  ○ Generalization to more complex inputs
  ○ Proof of generalization
Learning Recursive Neural Programs

● Our first step instantiation:
  ○ Architecture: Learn recursive programs in NPI
  ○ Training method: With explicitly recursive execution traces

● Future work and open questions:
  ○ Extend to other architectures beyond NPI
  ○ Learn recursive programs with less supervision
    ■ Without requiring explicitly recursive training traces
    ■ Input-output examples instead of execution traces
  ○ Explore other domains such as perception and control
THANKS!
THANKS!
THANKS!
THANKS!
THANKS!
THANKS!
THANKS!
THANKS!
THANKS!
THANKS!