Making Neural Programming Architectures Generalize via Recursion

Jonathon Cai\textsuperscript{1}, Richard Shin\textsuperscript{1}, Dawn Song\textsuperscript{1}

\textsuperscript{1}University of California, Berkeley

ICLR, 2017
Presenter: Arshdeep Sekhon
Task: Learn programs from data
For example, Addition, sorting, etc.
Introduction

1. Task: Learn programs from data
2. For example, Addition, sorting, etc.
3. Not only sort an array, but learn a specific sorting algorithm
Introduction

1. Task: Learn programs from data
2. For example, Addition, sorting, etc.
3. Not only sort an array, but learn a specific sorting algorithm
4. Evaluating the model: Check how well the model performs on more complex inputs
Two categories based on type of training data:

1. Neural Turing Machine, Pointer Networks, etc: input-output pairs
2. Neural programming Interpreter: Synthetic execution traces
Neural Programming Interpreter

Figure: NPI Core
Neural Programming Interpreter Architecture

\[ s_t = f_{enc}(e_t, a_t) \]
\[ h_t = f_{lstm}(s_t, p_t, h_{t-1}) \]
\[ r_t = f_{end}(h_t) \]
\[ k_t = f_{prog}(h_t) \]
\[ a_{t+1} = f_{arg}(h_t) \]  

1. \( e_t \) current environment state; for example: progress/which digit is currently being added
2. \( a_t \) the input value: For example, while writing output, the number that is to be written
3. \( r_t \): the probability whether to stop execution of program and return to caller
NPI Architecture

\[
\begin{align*}
 s_t &= f_{\text{enc}}(e_t, a_t) \\
 h_t &= f_{\text{lstm}}(s_t, p_t, h_{t-1}) \\
 r_t &= f_{\text{end}}(h_t) \\
 k_t &= f_{\text{prog}}(h_t) \\
 a_{t+1} &= f_{\text{arg}}(h_t)
\end{align*}
\] (2)

1. \(k_t\): program key that points to the program’s embedding

2. \(f_{\text{enc}}: \mathbb{E} \times \mathbb{A} \rightarrow \mathbb{R}^D\) is a domain specific encoder.
   
   \(f_{\text{end}}: \mathbb{R}^M \rightarrow [0, 1], f_{\text{prog}}: \mathbb{R}^M \rightarrow \mathbb{R}^K, f_{\text{arg}}: \mathbb{R}^M \rightarrow \mathbb{A}\)
Algorithm 1 Neural programming inference

1: **Inputs:** Environment observation $e$, program $p$, arguments $a$, stop threshold $\alpha$
2: function RUN($e, p, a$) 
3: $h \leftarrow 0, r \leftarrow 0$
4: while $r < \alpha$ do 
5: $s \leftarrow f_{enc}(e, a), h \leftarrow f_{lstm}(s, p, h)$
6: $r \leftarrow f_{end}(h), p_2 \leftarrow f_{prog}(h), a_2 \leftarrow f_{arg}(h)$
7: if $p$ is a primitive function then 
8: $e \leftarrow f_{env}(e, p, a)$.
9: else 
10: function RUN($e, p_2, a_2$)

**Figure:** NPI Algorithm
Figure: NPI algorithm
Non-Recursive

1. ADD
2.   ADD1
3.   WRITE OUT 1
4. CARRY
5.   PTR CARRY LEFT
6.   WRITE CARRY 1
7.   PTR CARRY RIGHT
8. LSHIFT
9.   PTR INP1 LEFT
10.  PTR INP2 LEFT
11.  PTR CARRY LEFT
12.  PTR OUT LEFT
13.  ADD1
14.   ...

Figure: Addition
Figure: Addition using NPI
Training of NPI

1. Use execution traces

2. $\xi^\text{inp}_t : \{e_t, i_t, a_t\}$ and $\xi^\text{out}_t : \{r_t, i_{t+1}, a_{t+1}\}$ for $t = 1, \ldots, T$

3. Curriculum learning
Poor Generalization

Figure: Previous models suffer from poor generalization beyond a threshold level of complexity

1. Curriculum Learning: train on more complex inputs
2. No change in learnt semantics
3. Model ends up learning overly complex representations, example, dependence on length
4. Learn recursion
Recursion

1. Base Case: termination criteria / no more recursion
2. Rules: to reduce all problems towards base case

NPI can easily incorporate Recursion.

1. NPI has a call structure
2. Implement recursion as a program calling itself.
Adding Recursion to NPI

1. Recursion helps to generalize as well as makes it easier to prove generalization
2. To prove generalization:
   1. Learns base cases correctly
   2. Learns reduction rules correctly
3. Reduction rules and base cases are finite for programs, unlike infinite possible complex inputs
4. reduces the number of configurations that need to be considered
To add recursion, change the execution traces: new training traces that explicitly contain recursive elements.
Provable Guarantees of Generalization

Verification Theorem

∀ i ∈ V, M(i) ⇓ P(i)

i: a sequence of step inputs
V: set of valid sequences of step inputs
P: correct program/algorithm
M: Model

For the same sequence of step inputs, the model produces exact same step output as the program it tries to learn
Constructing Verification Set for Addition

For non recursive:
1. $1 + 1 = 2$
2. $99 + 99 = 198$
3. $99..99 + 99..99 =$
4. Infinite input sequences

For Recursive cases:
1. Only need to take care of two columns
2. 20000 cases
### Results

#### Table 2: Accuracy on Randomly Generated Problems for Topological Sort

<table>
<thead>
<tr>
<th>Number of 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>6</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>

#### Table 3: Accuracy on Randomly Generated Problems for 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>22</td>
<td>20%</td>
<td>100%</td>
</tr>
<tr>
<td>25</td>
<td>3.33%</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>

**Figure**: Sorting