Making Neural Programming Architectures Generalize via Recursion

Jonathon Cai[1], Richard Shin[1], Dawn Song[1]

[1]University of California, Berkeley

ICLR,2017

Presenter: Arshdeep Sekhon

Introduction

1. Task: Learn programs from data
2. 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

Previous Approaches

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

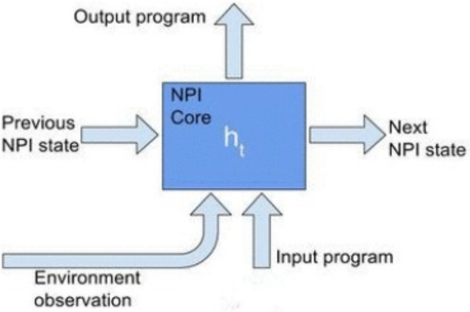

Figure: NPI Core

Neural Programming Interpreter Architecture

$$
\begin{aligned}
s_t &= f_{enc}(e_t, a_t) \\
h_t &= f_{lstm}(s_t, p_t, h_{t-1}) \\
r_t &= f_{end}(h_t) \\
k_t &= f_{prog}(h_t) \\
a_{t+1} &= f_{arg}(h_t)
\end{aligned} \tag{1}
$$

1. e_t current environment state; for example: progress/which digit is currently beeing 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

$$
\begin{aligned}
s_t &= f_{enc}(e_t, a_t) \\
h_t &= f_{lstm}(s_t, p_t, h_{t-1}) \\
r_t &= f_{end}(h_t) \\
k_t &= f_{prog}(h_t) \\
a_{t+1} &= f_{arg}(h_t)
\end{aligned} \tag{2}
$$

1. k_t: program key that points to the progrma's embedding
2. $f_{enc} : \mathbb{E} \times \mathbb{A} \to \mathbb{R}^D$ is a domain specific encoder. $f_{end} : \mathbb{R}^M \to [0,1], f_{prog} : \mathbb{R}^M \to \mathbb{R}^K, f_{arg} : \mathbb{R}^M \to \mathbb{A}$

Algorithm 1 Neural programming inference

Inputs: Environment observation e, program p, arguments a, stop threshold α

function RUN(e, p, a)

 $h \leftarrow \mathbf{0}, r \leftarrow 0$

 while $r < \alpha$ **do**

  $s \leftarrow f_{enc}(e, a), h \leftarrow f_{lstm}(s, p, h)$

  $r \leftarrow f_{end}(h), p_2 \leftarrow f_{prog}(h), a_2 \leftarrow f_{arg}(h)$

  if p is a primitive function **then**

   $e \leftarrow f_{env}(e, p, a)$.

  else

   function RUN(e, p_2, a_2)

Figure: NPI Algorithm

NPI Inference

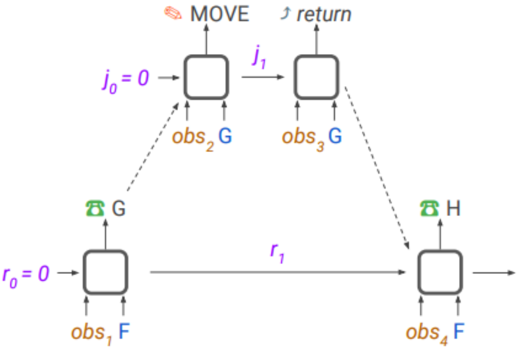

Figure: NPI algorithm

Grade School Addition Execution Trace

Non-Recursive

```
ADD
  ADD1
    WRITE OUT 1
    CARRY
      PTR CARRY LEFT
      WRITE CARRY 1
      PTR CARRY RIGHT
  LSHIFT
    PTR INP1 LEFT
    PTR INP2 LEFT
    PTR CARRY LEFT
    PTR OUT LEFT
  ADD1
    ...
```

Figure: Addition

NPI Inference

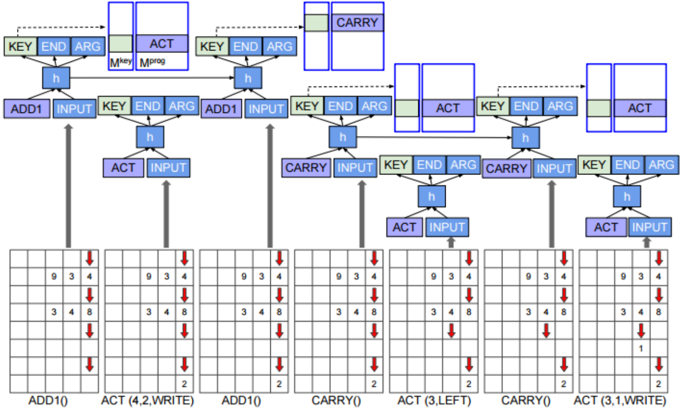

Figure: Addition using NPI

Training of NPI

1. Use execution traces
2. $\xi_t^{inp} : \{e_t, i_t, a_t\}$ and $\xi_t^{out} : \{r_t, i_{t+1}, a_{t+1}\}$ for $t = 1, \ldots, T$
3. Curriculum learning

Poor Generalization

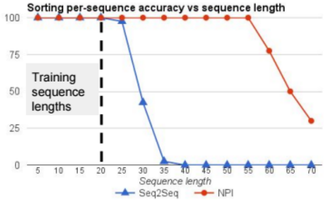

Figure: Previous models suffer from poor generalization beyond a threshold level of complexity

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

Recursion

1. Base Case: termination criteria/ no more recusrion
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

Adding Recursion to NPI

Non-Recursive

```
ADD
  ADD1
    WRITE OUT 1
    CARRY
      PTR CARRY LEFT
      WRITE CARRY 1
      PTR CARRY RIGHT
  LSHIFT
    PTR INP1 LEFT
    PTR INP2 LEFT
    PTR CARRY LEFT
    PTR OUT LEFT
  ADD1
    ...
```

Recursive

```
ADD
  ADD1
    WRITE OUT 1
    CARRY
      PTR CARRY LEFT
      WRITE CARRY 1
      PTR CARRY RIGHT
  LSHIFT
    PTR INP1 LEFT
    PTR INP2 LEFT
    PTR CARRY LEFT
    PTR OUT LEFT
  ADD
    ...
```

Figure: Recursive Addition

1. To add recursion, change the execution traces: new training traces that explicitly contain recursive elements

Provable Guarantees of Generalization

Verification Theorem

$\forall i \in V, M(i) \Downarrow P(i)$

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

Number of Vertices	**Non-Recursive**	**Recursive**
5	6.7%	100%
6	6.7%	100%
7	3.3%	100%
8	0%	100%
70	0%	100%

Table 3: Accuracy on Randomly Generated Problems for Quicksort

Length of Array	**Non-Recursive**	**Recursive**
3	100%	100%
5	100%	100%
7	100%	100%
11	73.3%	100%
15	60%	100%
20	30%	100%
22	20%	100%
25	3.33%	100%
30	3.33%	100%
70	0%	100%

Figure: Sorting