A Not that Comprehensive Introduction to Neural Programming

Lili Mou
doublepower.mou@gmail.com
• Introduction

• Symbolic execution
  – (Fully supervised) Neural programmer interpreter
  – (Weakly supervised) Neural symbolic machine
  – “Spurious programs” and inductive programming
    • Learning semantic parsers from denotations
    • DeepCoder
    • More thoughts on “spurious programs”

• Distributed execution
  – Learning to execute
  – Neural enquirer
  – Differentiable neural computer

• Hybrid execution
  – Neural programmer
  – Coupling approach
Outline

- Introduction
- Symbolic execution
  - (Fully supervised) Neural programmer interpreter
  - (Weakly supervised) Neural symbolic machine
  - “Spurious programs” and inductive programming
    - Learning semantic parsers from denotations
    - DeepCoder
    - More thoughts on “spurious programs”
- Distributed execution
  - Learning to execute
  - Neural enquirer
  - Differentiable neural computer
- Hybrid execution
  - Neural programmer
  - Coupling approach
NEURAL PROGRAMMER-INTERPRETERS

Scott Reed & Nando de Freitas
Google DeepMind
London, UK
scott.ellison.reed@gmail.com
nandodefreitas@google.com
MAKING NEURAL PROGRAMMING ARCHITECTURES GENERALIZE VIA RECURSION

Jonathon Cai, Richard Shin, Dawn Song
Department of Computer Science
University of California, Berkeley
Berkeley, CA 94720, USA
{jonathon, ricshin, dawnsong}@cs.berkeley.edu
\[ s_t = f_{enc}(e_t, a_t) \]

\[ h_t = f_{lstm}(s_t, p_t, h_{t-1}) \]

\[ r_t = f_{end}(h_t), p_{t+1} = f_{prog}(h_t), a_{t+1} = f_{arg}(h_t) \]
When the program is called, the current context including the caller’s memory state is stored on a stack; when the program returns, the stored context is popped off the stack to resume execution in the previous caller’s context.

**Algorithm 1** Neural programming inference

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

• Grade-school addition

\[ f_{enc}(Q, i_1, i_2, i_3, i_4, a_t) = MLP( [Q(1, i_1), Q(2, i_2), Q(3, i_3), Q(4, i_4), a_t(1), a_t(2), a_t(3)] ) \]

<table>
<thead>
<tr>
<th>Input 1</th>
<th>0 0 0 9 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input 2</td>
<td>0 0 1 2 5</td>
</tr>
<tr>
<td>Carry</td>
<td>0 0 1 1 1</td>
</tr>
<tr>
<td>Output</td>
<td>0 0 6 2 1</td>
</tr>
</tbody>
</table>

\[ \text{on-chip size } k=11 \]
\[ 0 \sim 9 + N \]
Example

• Grade-school addition

```
ADD  Add for one digit (a wrapper)
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
...
```
Example

- Grade-school addition

\[ f_{\text{enc}}(Q, i_1, i_2, i_3, i_4, a_t) = MLP([Q(1, i_1), Q(2, i_2), Q(3, i_3), Q(4, i_4), a_t(1), a_t(2), a_t(3)]) \]

ADD
- Write “1” at the output slip

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
- ...

\[ \begin{array}{c|cccc|c|}
\text{Input 1} & 0 & 0 & 0 & 9 & 6 \\
\text{Input 2} & 0 & 0 & 1 & 2 & 5 \\
\text{Carry} & 0 & 0 & 1 & 1 & 1 \\
\text{Output} & 0 & 0 & 6 & 2 & 1 \\
\end{array} \]
Example

- Grade-school addition

Input 1: 0 0 0 9 6
Input 2: 0 0 1 2 5
Carry: 0 0 1 1
Output: 0 0 6 2 1

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
Move printers left

...
## Recursion?

### Non-Recursive

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ADD</td>
</tr>
<tr>
<td>2</td>
<td>ADD1</td>
</tr>
<tr>
<td>3</td>
<td>WRITE OUT 1</td>
</tr>
<tr>
<td>4</td>
<td>CARRY</td>
</tr>
<tr>
<td>5</td>
<td>PTR CARRY LEFT</td>
</tr>
<tr>
<td>6</td>
<td>WRITE CARRY 1</td>
</tr>
<tr>
<td>7</td>
<td>PTR CARRY RIGHT</td>
</tr>
<tr>
<td>8</td>
<td>LSHIFT</td>
</tr>
<tr>
<td>9</td>
<td>PTR INP1 LEFT</td>
</tr>
<tr>
<td>10</td>
<td>PTR INP2 LEFT</td>
</tr>
<tr>
<td>11</td>
<td>PTR CARRY LEFT</td>
</tr>
<tr>
<td>12</td>
<td>PTR OUT LEFT</td>
</tr>
<tr>
<td>13</td>
<td>ADD1</td>
</tr>
<tr>
<td>14</td>
<td>...</td>
</tr>
</tbody>
</table>

### Recursive

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ADD</td>
</tr>
<tr>
<td>2</td>
<td>ADD1</td>
</tr>
<tr>
<td>3</td>
<td>WRITE OUT 1</td>
</tr>
<tr>
<td>4</td>
<td>CARRY</td>
</tr>
<tr>
<td>5</td>
<td>PTR CARRY LEFT</td>
</tr>
<tr>
<td>6</td>
<td>WRITE CARRY 1</td>
</tr>
<tr>
<td>7</td>
<td>PTR CARRY RIGHT</td>
</tr>
<tr>
<td>8</td>
<td>LSHIFT</td>
</tr>
<tr>
<td>9</td>
<td>PTR INP1 LEFT</td>
</tr>
<tr>
<td>10</td>
<td>PTR INP2 LEFT</td>
</tr>
<tr>
<td>11</td>
<td>PTR CARRY LEFT</td>
</tr>
<tr>
<td>12</td>
<td>PTR OUT LEFT</td>
</tr>
<tr>
<td>13</td>
<td>ADD</td>
</tr>
<tr>
<td>14</td>
<td>...</td>
</tr>
</tbody>
</table>
The Magic Here

- No magic
- Fully supervised
  - What “ADD” sees is
    - ADD1, LSHIFT, ADD, RETURN
  - What “ADD1” sees is
    - WRITE (parameters: OUT, 1)
    - CARRY, RETURN
  - Etc.

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
    ...
```
Design Philosophy

- Define a “local” context, which the current step of execution fully depends on.
- Manually program in terms of the above “language.”
- Generate execution sequences (a.k.a. program traces) with some (not many) examples.
• Introduction

• Symbolic execution
  – (Fully supervised) Neural programmer interpreter
  – (Weakly supervised) Neural symbolic machine
  – “Spurious programs” and inductive programming
    • Learning semantic parsers from denotations
    • DeepCoder
    • More thoughts on “spurious programs”

• Distributed execution
  – Learning to execute
  – Neural enquirer
  – Differentiable neural computer

• Hybrid execution
  – Neural programmer
  – Coupling approach
Neural Symbolic Machines: Learning Semantic Parsers on Freebase with Weak Supervision

Chen Liang,* Jonathan Berant,† Quoc Le, Kenneth D. Forbus, Ni Lao

Northwestern University, Evanston, IL
{chenliang2013,forbus}@u.northwestern.edu
Tel-Aviv University, Tel Aviv-Yafo, Israel
joberant@cs.tau.ac.il
Google Inc., Mountain View, CA
{qvl,nlao}@google.com
• Generally seq2seq
But with tricks
- Entities marked with GRU output as embeddings
- Searched with attention
Training

- Generally REINFORCE

\[ \nabla_{\theta} J^{RL}(\theta) = \sum_{q} \sum_{a_{0:T}} P(a_{0:T}|q, \theta)[R(q, a_{0:T}) - B(q)] \nabla_{\theta} \log P(a_{0:T}|q, \theta) \]
Training

• But with more tricks
  – Beam search instead of sampling
  – Keep the “best” obtained programs so far and perform supervised training in addition to REINFORCE
Outline

- Introduction
- Symbolic execution
  - (Fully supervised) Neural programmer interpreter
  - (Weakly supervised) Neural symbolic machine
  - “Spurious programs” and inductive programming
    - Learning semantic parsers from denotations
    - DeepCoder
    - More thoughts on “spurious programs”
- Distributed execution
  - Learning to execute
  - Neural enquirer
  - Differentiable neural computer
- Hybrid execution
  - Neural programmer
  - Coupling approach
Inferring Logical Forms From Denotations

Panupong Pasupat  
Computer Science Department  
Stanford University  
ppasupat@cs.stanford.edu

Percy Liang  
Computer Science Department  
Stanford University  
pliang@cs.stanford.edu

(ACL-16)

- Spurious programs
- Fictitious worlds
### The Problem of “Spurious Programs”

<table>
<thead>
<tr>
<th>Year</th>
<th>Venue</th>
<th>Position</th>
<th>Event</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001</td>
<td>Hungary</td>
<td>2nd</td>
<td>400m</td>
<td>47.12</td>
</tr>
<tr>
<td>2003</td>
<td>Finland</td>
<td>1st</td>
<td>400m</td>
<td>46.69</td>
</tr>
<tr>
<td>2005</td>
<td>Germany</td>
<td>11th</td>
<td>400m</td>
<td>46.62</td>
</tr>
<tr>
<td>2007</td>
<td>Thailand</td>
<td>1st</td>
<td>relay</td>
<td>182.05</td>
</tr>
<tr>
<td>2008</td>
<td>China</td>
<td>7th</td>
<td>relay</td>
<td>180.32</td>
</tr>
</tbody>
</table>

### Correct

\[ z_1: \mathbf{R}[Venue].\text{argmax}(\text{Position.1st}, \text{Index}) \]

Among rows with Position = 1st, pick the one with maximum index, then return the Venue of that row.

\[ z_2: \mathbf{R}[Venue].\text{Index.max}(\mathbf{R}[\text{Index}].\text{Position.1st}) \]

Find the maximum index of rows with Position = 1st, then return the Venue of the row with that index.

\[ z_3: \mathbf{R}[Venue].\text{argmax}(\text{Position.Number.1}, \]

**\[ \mathbf{R}[\lambda x.\mathbf{R}[\text{Date}].\mathbf{R}[\text{Year}].x] \]**

Among rows with Position number 1, pick one with latest date in the Year column and return the Venue.

### Spurious

\[ z_4: \mathbf{R}[Venue].\text{argmax}(\text{Position.Number.1}, \]

**\[ \mathbf{R}[\lambda x.\mathbf{R}[\text{Number}].\mathbf{R}[\text{Time}].x] \]**

Among rows with Position number 1, pick one with maximum Time number. Return the Venue.

\[ z_5: \mathbf{R}[Venue].\text{Year.Number.} ( \]

**\[ \mathbf{R}[\text{Number}].\mathbf{R}[\text{Year}].\text{argmax}(\text{Type.Row, Index}) - 1 \]**

Subtract 1 from the Year in the last row, then return the Venue of the row with that Year.

### Inconsistent

\[ \hat{z}: \mathbf{R}[Venue].\text{argmin}(\text{Position.1st}, \text{Index}) \]

Among rows with Position = 1st, pick the one with minimum index, then return the Venue. (\(=\) Finland)

---

**x:** “Where did the last 1st place finish occur?”

**y:** Thailand
Solution: Generating fictitious data

<table>
<thead>
<tr>
<th>Year</th>
<th>Venue</th>
<th>Position</th>
<th>Event</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001</td>
<td>Hungary</td>
<td>2nd</td>
<td>400m</td>
<td>47.12</td>
</tr>
<tr>
<td>2003</td>
<td>Finland</td>
<td>1st</td>
<td>400m</td>
<td>46.69</td>
</tr>
<tr>
<td>2005</td>
<td>Germany</td>
<td>11th</td>
<td>400m</td>
<td>46.62</td>
</tr>
<tr>
<td>2007</td>
<td>Thailand</td>
<td>1st</td>
<td>relay</td>
<td>182.05</td>
</tr>
<tr>
<td>2008</td>
<td>China</td>
<td>7th</td>
<td>relay</td>
<td>180.32</td>
</tr>
</tbody>
</table>

$x$: “Where did the last 1st place finish occur?”

$y$: Thailand

Figure 4: From the example in Figure 1, we generate a table for the fictitious world $w_1$.

<table>
<thead>
<tr>
<th>$w_1$</th>
<th>$w_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>z1</td>
<td>Thailand</td>
</tr>
<tr>
<td>z2</td>
<td>Thailand</td>
</tr>
<tr>
<td>z3</td>
<td>Thailand</td>
</tr>
<tr>
<td>z4</td>
<td>Thailand</td>
</tr>
<tr>
<td>z5</td>
<td>Thailand</td>
</tr>
<tr>
<td>z6</td>
<td>Thailand</td>
</tr>
</tbody>
</table>

- Equivalent classes
  - $q_1=\{z_1, z_2, z_3\}$
  - $q_2=\{z_4\}$
  - $q_3=\{z_5, z_6\}$
- Choose worlds with most information gain
• Introduction

• Symbolic execution
  – (Fully supervised) Neural programmer interpreter
  – (Weakly supervised) Neural symbolic machine
  – “Spurious programs” and inductive programming
    • Learning semantic parsers from denotations
    • DeepCoder
    • More thoughts on “spurious programs”

• Distributed execution
  – Learning to execute
  – Neural enquirer
  – Differentiable neural computer

• Hybrid execution
  – Neural programmer
  – Coupling approach
DeepCoder: Learning to Write Programs

Matej Balog*  
Department of Engineering  
University of Cambridge

Alexander L. Gaunt, Marc Brockschmidt,  
Sebastian Nowozin, Daniel Tarlow  
Microsoft Research
an input-output example:

Input:

\[-17, -3, 4, 11, 0, -5, -9, 13, 6, 6, -8, 11\]

Output:

\[-12, -20, -32, -36, -68\]

Figure 1: An example program in our DSL that takes a single integer array as its input.

Figure 2: Neural network predicts the probability of each function appearing in the source code.
Search

• DFS
  – DFS can opt to consider the functions as ordered by their predicted probabilities from the neural network
• “Sort and add” enumeration
• Sketch

• Not really “structured” prediction
• Introduction

• Symbolic execution
  – (Fully supervised) Neural programmer interpreter
  – (Weakly supervised) Neural symbolic machine
  – “Spurious programs” and inductive programming
    • Learning semantic parsers from denotations
    • DeepCoder
    • More thoughts on “spurious programs”

• Distributed execution
  – Learning to execute
  – Neural enquirer
  – Differentiable neural computer

• Hybrid execution
  – Neural programmer
  – Coupling approach
• Introduction

• Symbolic execution
  – (Fully supervised) Neural programmer interpreter
  – (Weakly supervised) Neural symbolic machine
  – “Spurious programs” and inductive programming
    • Learning semantic parsers from denotations
    • DeepCoder
    • More thoughts on “spurious programs”

• Distributed execution
  – Learning to execute
  – Neural enquirer
  – Differentiable neural computer

• Hybrid execution
  – Neural programmer
  – Coupling approach
Essentially a seq2seq model

Input:
```
j=8584
    for x in range(8):
        j+=920
    b=(1500+j)
    print((b+7567))
Target: 25011.
```

Input:
```
i=8827
    c=(i-5347)
    print((c+8704) if 2641<8500 else 5308)
Target: 12184.
```
• Introduction

• Symbolic execution
  – (Fully supervised) Neural programmer interpreter
  – (Weakly supervised) Neural symbolic machine
  – “Spurious programs” and inductive programming
    • Learning semantic parsers from denotations
    • DeepCoder
    • More thoughts on “spurious programs”

• Distributed execution
  – Learning to execute
  – Neural enquirer
  – Differentiable neural computer

• Hybrid execution
  – Neural programmer
  – Coupling approach
Neural Enquirer: Learning to Query Tables in Natural Language

Pengcheng Yin\textsuperscript{1}  Zhengdong Lu\textsuperscript{2}  Hang Li\textsuperscript{2}  Ben Kao\textsuperscript{1}
\textsuperscript{1} Department of Computer Science, The University of Hong Kong
\textsuperscript{2} Noah’s Ark Lab, Huawei Technologies
\textsuperscript{1}\{pcyin,kao\}@cs.hku.hk  \textsuperscript{2}\{Lu.Zhengdong,HangLi.HL\}@huawei.com

Which city hosted the longest Olympic game before the game in Beijing?
Three Modules

- Query encoder: BiRNN
- Table encoder
  - Concatenation of cell and field embeddings
  - Further processed by a multi-layer perceptron
- Executor
  - Column attention
    (soft selection)
  - Row annotation
    (distributed selection)
Executor

- The result of one-step execution softmax attention over columns and a distributed
  - Column attention
  - Row annotation
Details

• Let \( r_i^{(t-1)} \) be the previous step’s row annotation results, where the subscript \( i \) indexes a particular row.

• Last step’s execution information
  \[
g^{(t-1)} = \text{MaxPool}_i \{ r_i^{(t-1)} \}
\]

• Current step
  – Column attention
    \[
p_{fj}^{(t)} = \text{softmax} \left( \text{MLP} \left( [q; f_j; g^{(t-1)}] \right) \right)
    \]
  – Row annotation
    \[
c_{\text{select}}^{(t)} [i] = \sum_j p_{fj}^{(t)} c_{ij}
\]
  \[
r_i^{(t)} = \text{MLP} \left( [q, g^{(t-1)}, r^{(t-1)}, c_{\text{select}}^{(t)} [i]] \right)
\]
• Introduction

• Symbolic execution
  – (Fully supervised) Neural programmer interpreter
  – (Weakly supervised) Neural symbolic machine
  – “Spurious programs” and inductive programming
    • Learning semantic parsers from denotations
    • DeepCoder
    • More thoughts on “spurious programs”

• Distributed execution
  – Learning to execute
  – Neural enquirer
  – Differentiable neural computer

• Hybrid execution
  – Neural programmer
  – Coupling approach
Hybrid computing using a neural network with dynamic external memory

Alex Graves¹*, Greg Wayne¹*, Malcolm Reynolds¹, Tim Harley¹, Ivo Danihelka¹, Agnieszka Grabska-Barwińska¹, Sergio Gómez Colmenarejo¹, Edward Grefenstette¹, Tiago Ramalho¹, John Agapiou¹, Adrià Puigdomènech Badia¹, Karl Moritz Hermann¹, Yori Zwols¹, Georg Ostrovski¹, Adam Cain¹, Helen King¹, Christopher Summerfield¹, Phil Blunsom¹, Koray Kavukcuoglu¹ & Demis Hassabis¹
Memory and its Access

- **Memory:** $M \in \mathbb{R}^{N \times W}$
  - $N$: # of memory slots
  - $W$: Width/dim of a slot

- **Reader:**
  - $R$ readers
  - Content addressing
    + Temporal linking addressing

- **Writer:**
  - Exact 1 writer
  - Content addressing + dynamic memory allocation
Architecture

- **Input**
  - Current signal
  - Previous step’s memory read

- **Output**
  - Value
    \[
    \nu_t = W_v[h^1_t; \ldots; h^L_t] \\
    y_t = \nu_t + W_r[r^1_t; \ldots; r^R_t]
    \]
    (so that output is dependent on read memory)
  - Control
    \[
    \xi_t = W_\xi[h^1_t; \ldots; h^L_t]
    \]

\[
\xi_t = \left[ k^L_t; \ldots; k^R_t; \beta^1_t; \ldots; \beta^R_t; k^w_t; \beta^w_t; \hat{\epsilon}_t; v_t; f^1_t; \ldots; f^R_t; \hat{g}^a_t; \hat{g}^w_t; \hat{\pi}^1_t; \ldots; \hat{\pi}^R_t \right]
\]

\[
\chi = [x_t; r^1_{t-1}; \ldots; r^R_{t-1}]
\]
Controller: Preliminary

- To obtain a gate: sigmoid
- To obtain a distribution: softmax
- To constrain in $[1, \infty)$

$$\text{oneplus}(x) = 1 + \log(1 + e^x)$$

- A distribution

$$\mathcal{S}_N = \left\{ \alpha \in \mathbb{R}^N : \alpha_i \in [0,1], \sum_{i=1}^{N} \alpha_i = 1 \right\}$$

- No more than a distribution (detracted by a default null prob.)

$$\Delta_N = \left\{ \alpha \in \mathbb{R}^N : \alpha_i \in [0,1], \sum_{i=1}^{N} \alpha_i \leq 1 \right\}$$
A Glance at the Control Signals

- \( R \) read keys \( \{k^{r,i}_t \in \mathbb{R}^W; 1 \leq i \leq R\} \);
- \( R \) read strengths \( \{\beta^{r,i}_t = \text{oneplus}(\hat{\beta}^{r,i}_t) \in [1, \infty); 1 \leq i \leq R\} \);
- the write key \( k^w_t \in \mathbb{R}^W \);
- the write strength \( \beta^w_t = \text{oneplus}(\hat{\beta}^w_t) \in [1, \infty) \);
- the erase vector \( e_t = \sigma(\hat{e}_t) \in [0, 1]^W \);
- the write vector \( v_t \in \mathbb{R}^W \);
- \( R \) free gates \( \{f^i_t = \sigma(\hat{f}^i_t) \in [0, 1]; 1 \leq i \leq R\} \);
- the allocation gate \( g^a_t = \sigma(\hat{g}^a_t) \in [0, 1] \);
- the write gate \( g^w_t = \sigma(\hat{g}^w_t) \in [0, 1] \); and
- \( R \) read modes \( \{\pi^i_t = \text{softmax}(\hat{\pi}^i_t) \in \mathcal{S}_3; 1 \leq i \leq R\} \).
Memory Reading and Writing

- **Reader**
  - Reading weights $w_{t,1}^r, \ldots, w_{t,R}^r \in \Delta_N$
  - Read vectors $r_t^i = M_t^\top w_t^{r,i}$
    where $M: <N \times W>$, each $w: <N \times 1>$
    (no more than attention)

- **Writer**
  - Writing weight regarding a memory slot $w_t^w \in \Delta_N$
  - Erasing vector regarding a memory dimension
    (given by the controller) $e_t = \sigma(\hat{e}_t) \in [0,1]^W$
  - Writing operation $M_t = M_{t-1} \circ (E - w_t^w e_t^\top) + w_t^w v_t^\top$

  E: 1’s | $w \rightarrow 1$ & $e \rightarrow 1$ ==> $\textbf{M}$ is determined by $\textbf{v}$
Memory Addressing

• Content-based addressing
  - e.g., select 5

• Dynamic memory allocation
  - Free(.), Malloc(.) and write to it

• Temporal memory linkage
  - Move to the next element in a linked list

• C.f. CopyNet, Latent Predictor Network
Content-Based Addressing

\[ C(M, k, \beta)[i] = \frac{\exp\{D(k, M[i, \cdot])\beta\}}{\sum_j \exp\{D(k, M[j, \cdot])\beta\}} \quad \text{Size: } <N> \]

\[ D(u, v) = \frac{u \cdot v}{|u||v|} \]

\( k \) and beta given by controller

- \( R \) read keys \( \{k^{r,i}_t \in \mathbb{R}^W; 1 \leq i \leq R\} \);
- \( R \) read strengths \( \{\beta^{r,i}_t = \text{oneplus}(\hat{\beta}^{r,i}_t) \in [1,\infty); 1 \leq i \leq R\} \);
- the write key \( k^w_t \in \mathbb{R}^W \);
- the write strength \( \beta^w_t = \text{oneplus}(\hat{\beta}^w_t) \in [1,\infty) \);
Dynamic Memory Allocation

- Memory usage vector  \( u_t \in [0, 1]^N \)
  
  \( 1=\text{used}, \; 0=\text{unused} \quad u_0 = 0 \)

- Memory retention vector  \( \psi_t \in [0, 1]^N \)
  
  how much each location will not be freed by the free gates
  
  \[ \psi_t = \prod_{i=1}^{R} \left( 1 - f_t^i w_{t-1}^{r,i} \right) \]

  where free gates are given by the controller
  
  \( \{ f_t^i = \sigma(\hat{f}_t^i) \in [0,1]; \; 1 \leq i \leq R \} \)

\( \text{Free is in the sense of “malloc and free,” rather than “free-style.”} \)

Read more => retain less
\( f \text{ larger} \Rightarrow \text{ retrain less} \)
Dynamic Memory Allocation

- Memory usage vector recursion
  - 1=used, 0=unused  \( u_0 = 0 \)
  - \( u_t = (u_{t-1} + w_{t-1}^w - u_{t-1} \circ w_{t-1}^w) \circ \psi_t \)
  - \( u = [u^*(1-w) + 1^*w] \ast \psi_t \)
  - \( w_{t-1}^w \): last step’s writing weight
  - write more => use more

- Free list \( \phi_t \in \mathbb{Z}^N \)
  - Sorted list of memory usage
  - \( \phi_t[1] \): least used memory index
Dynamic Memory Allocation

- Allocation weighting

\[ a_t[\phi_t[j]] = (1 - u_t[\phi_t[j]]) \prod_{i=1}^{j-1} u_t[\phi_t[i]] \]

- \( 1 - u \): used more => allocate less
- \( \phi_t \in \mathbb{Z}^N \): sorted list of memory usage
- \( \prod_{i=1}^{j-1} u_t[\phi_t[i]] \): least used => allocate more
Write Weighting

\[ w_t^w = g_t^w \left[ g_t^a a_t + (1 - g_t^a) c_t^w \right] \]

- \( g_t^w \): writing gate, deciding to write or not
- \( g_t^a \): memory allocation gate for write weighting
- Both given by the controller

the allocation gate \( g_t^a = \sigma(\hat{g}_t^a) \in [0,1] \);

the write gate \( g_t^w = \sigma(\hat{g}_t^w) \in [0,1] \);
Read Weighting

\[ w_{t}^{r,i} = \pi_{t}[1] b_{t}^{i} + \pi_{t}[2] c_{t}^{r,i} + \pi_{t}[3] f_{t}^{i} \]
Temporal memory linkage

\[ w_t^{r,i} = \pi_t^i[1] b_t^i + \pi_t^i[2] c_t^{r,i} + \pi_t^i[3] f_t^i \]

- Precedence weighting \( p_t \in \Delta_N \)
  \[
  p_0 = 0 \\
  p_t = \left(1 - \sum_i w_t^w[i]\right) p_{t-1} + w_t^w
  
  \]
- Degree to which a slot is lastly written
- Current writing weight plus \( p_{t-1} \)
- \( p_{t-1} \) discounted by summed writing weight
Temporal memory linkage

\[ \mathbf{w}^{r,i}_t = \pi^i[1] \mathbf{b}^i_t + \pi^i[2] \mathbf{c}^{r,i}_t + \pi^i[3] \mathbf{f}^i_t \]

- Linking matrix \( L_t \in [0, 1]^{N \times N} \)
  - \( L_0[i,j] = 0 \quad \forall i, j \)
  - \( L_t[i,i] = 0 \quad \forall i \)
  - \( L_t[i,j] = (1 - \mathbf{w}^w_t[i] - \mathbf{w}^w_t[j])L_{t-1}[i,j] + \mathbf{w}^w_t[i]p_{t-1}[j] \)
    - Degree to which location i is written after j
    - slot i written at the current step
      slot j written at the last step
      \( \implies L[i,j] \to 1 \)
    - Slot i not written \( \implies L[i,j] \) tends to remain
    - Slot j written \( \implies L[i,j] \to 0 \), i.e., refresh
Temporal memory linkage

\[ w_{t}^{r,i} = \pi_{t}^{i}[1]b_{t}^{i} + \pi_{t}^{i}[2]c_{t}^{r,i} + \pi_{t}^{i}[3]f_{t}^{i} \]

- **Backward/forward weighting**

  \[ b_{t}^{i} \in \Delta_{N} \quad f_{t}^{i} \in \Delta_{N} \]

  \[ f_{t}^{i} = L_{t}w_{t-1}^{r,i} \]

  \[ b_{t}^{i} = L_{t}^{\top}w_{t-1}^{r,i} \]

  Linking weight * last step’s reading weight
• Introduction
• Symbolic execution
  – (Fully supervised) Neural programmer interpreter
  – (Weakly supervised) Neural symbolic machine
  – “Spurious programs” and inductive programming
    • Learning semantic parsers from denotations
    • DeepCoder
    • More thoughts on “spurious programs”
• Distributed execution
  – Learning to execute
  – Neural enquirer
  – Differentiable neural computer
• Hybrid execution
  – Neural programmer
  – Coupling approach
Neural Programmer: Inducing Latent Programs with Gradient Descent

Arvind Neelakantan*
University of Massachusetts Amherst
arvind@cs.umass.edu

Quoc V. Le
Google Brain
qvl@google.com

Ilya Sutskever
Google Brain
ilyasv@google.com
Idea

- Step-by-step fusion of different operators (and arguments)
- Trained by MSE
- Fully differentiable
- Drawbacks
  - Only numerical results (?)
  - Exponential # of combinatorial states
Arhitecture

- Question RNN
- Selector (Controller)
- Operators
- History RNN
Selector (Controller)

• Select an operator

\[ \alpha_t^{op} = \text{softmax}(U \tanh(W^{op}[q; h_t])) \]

• Select a column for processing

\[ \alpha_t^{col} = \text{softmax}(P \tanh(W^{col}[q; h_t])) \]
## Operators

<table>
<thead>
<tr>
<th>Type</th>
<th>Operation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregate</td>
<td>Sum</td>
<td>$sum_t[j] = \sum_{i=1}^{M} row_select_{t-1}[i] \times table[i][j], \forall j = 1, 2, \ldots, C$</td>
</tr>
<tr>
<td></td>
<td>Count</td>
<td>$count_t = \sum_{i=1}^{M} row_select_{t-1}[i]$</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>Difference</td>
<td>$diff_t = scalar_output_{t-3} - scalar_output_{t-1}$</td>
</tr>
<tr>
<td>Comparison</td>
<td>Greater</td>
<td>$g_t[i][j] = table[i][j] &gt; pivot_g, \forall (i, j), i = 1, \ldots, M, j = 1, \ldots, C$</td>
</tr>
<tr>
<td></td>
<td>Lesser</td>
<td>$l_t[i][j] = table[i][j] &lt; pivot_l, \forall (i, j), i = 1, \ldots, M, j = 1, \ldots, C$</td>
</tr>
<tr>
<td>Logic</td>
<td>And</td>
<td>$and_t[i] = \min(row_select_{t-1}[i], row_select_{t-2}[i]), \forall i = 1, 2, \ldots, M$</td>
</tr>
<tr>
<td></td>
<td>Or</td>
<td>$or_t[i] = \max(row_select_{t-1}[i], row_select_{t-2}[i]), \forall i = 1, 2, \ldots, M$</td>
</tr>
<tr>
<td>Assign</td>
<td>assign</td>
<td>$assign_t[i][j] = row_select_{t-1}[i], \forall (i, j), i = 1, 2, \ldots, M, j = 1, 2, \ldots, C$</td>
</tr>
<tr>
<td>Lookup</td>
<td>Reset</td>
<td>$reset_t[i] = 1, \forall i = 1, 2, \ldots, M$</td>
</tr>
</tbody>
</table>

\[
\beta_{op} = \text{softmax}(ZU(op))
\]

\[
pivot_{op} = \sum_{i=1}^{N} \beta_{op}(i) qn_i
\]
<table>
<thead>
<tr>
<th>Type</th>
<th>Operation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregate</td>
<td>Sum</td>
<td>$\text{sum}<em>t[j] = \sum</em>{i=1}^{M} \text{row}_\text{select}_{t-1}[i] \times \text{table}[i][j], \forall j = 1, 2, \ldots, C$</td>
</tr>
<tr>
<td></td>
<td>Count</td>
<td>$\text{count}<em>t = \sum</em>{i=1}^{M} \text{row}_\text{select}_{t-1}[i]$</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>Difference</td>
<td>$\text{diff}<em>t = \text{scalar}_\text{output}</em>{t-3} - \text{scalar}_\text{output}_{t-1}$</td>
</tr>
<tr>
<td>Comparison</td>
<td>Greater</td>
<td>$g_t[i][j] = \text{table}[i][j] &gt; \text{pivot}_t, \forall (i,j), i = 1, \ldots, M, j = 1, \ldots, C$</td>
</tr>
<tr>
<td></td>
<td>Lesser</td>
<td>$l_t[i][j] = \text{table}[i][j] &lt; \text{pivot}_t, \forall (i,j), i = 1, \ldots, M, j = 1, \ldots, C$</td>
</tr>
<tr>
<td>Logic</td>
<td>And</td>
<td>$\text{and}<em>t[i] = \min(\text{row}_\text{select}</em>{t-1}[i], \text{row}_\text{select}_{t-2}[i]), \forall i = 1, 2, \ldots, M$</td>
</tr>
<tr>
<td></td>
<td>Or</td>
<td>$\text{or}<em>t[i] = \max(\text{row}_\text{select}</em>{t-1}[i], \text{row}_\text{select}_{t-2}[i]), \forall i = 1, 2, \ldots, M$</td>
</tr>
<tr>
<td>Assign Lookup</td>
<td>assign</td>
<td>$\text{assign}<em>t[i][j] = \text{row}_\text{select}</em>{t-1}[i], \forall (i,j), i = 1, 2, \ldots, M, j = 1, 2, \ldots, C$</td>
</tr>
<tr>
<td>Reset</td>
<td>Reset</td>
<td>$\text{reset}_t[i] = 1, \forall i = 1, 2, \ldots, M$</td>
</tr>
</tbody>
</table>

More formally, the output variables are given by:

$$
\text{scalar}\_\text{answer}_t = \alpha_t^{op}(\text{count}) \text{count}_t + \alpha_t^{op}(\text{difference}) \text{diff}_t + \sum_{j=1}^{C} \alpha_t^{col}(j) \alpha_t^{op}(\text{sum}) \text{sum}_t[j],
$$

$$
\text{lookup}\_\text{answer}_t[i][j] = \alpha_t^{col}(j) \alpha_t^{op}(\text{assign}) \text{assign}_t[i][j], \forall (i,j), i = 1, 2, \ldots, M, j = 1, 2, \ldots, C
$$

The row selector variable is given by:

$$
\text{row}\_\text{select}_t[i] = \alpha_t^{op}(\text{and}) \text{and}_t[i] + \alpha_t^{op}(\text{or}) \text{or}_t[i] + \alpha_t^{op}(\text{reset}) \text{reset}_t[i] + \sum_{j=1}^{C} \alpha_t^{col}(j) (\alpha_t^{op}(\text{greater}) g_t[i][j] + \alpha_t^{op}(\text{lesser}) l_t[i][j]), \forall i = 1, \ldots, M
$$
Text Matching

• Example: what is the sum of elements in column B whose field in column C is \textbf{word:1} and field in column A is \textbf{word:7}?

• Text matching (no textual operation or output)

\[
\text{row}\_\text{select}_t[i] = \alpha_t^{op}(\text{and}) \, \text{and}_t[i] + \alpha_t^{op}(\text{or}) \, \text{or}_t[i] + \alpha_t^{op}(\text{reset}) \, \text{reset}_t[i] + \\
\sum_{j=K+1}^{C} \alpha_t^{col}(j) (\alpha_t^{op}(\text{greater}) g_t[i][j] + \alpha_t^{op}(\text{lesser}) l_t[i][j]) + \\
\sum_{j=1}^{K} \alpha_t^{col}(j) (\alpha_t^{op}(\text{text\_match}) text\_match_t[i][j], \forall i = 1, \ldots, M)
\]
Training Objective

- Scalar answer

\[ L_{\text{scalar}}(\text{scalar\_answer}_T, y) = \begin{cases} \frac{1}{2} a^2, & \text{if } a \leq \delta \\ \delta a - \frac{1}{2} \delta^2, & \text{otherwise} \end{cases} \]

- List answer

\[ L_{\text{lookup}}(\text{lookup\_answer}_T, y) = -\frac{1}{MC} \sum_{i=1}^{M} \sum_{j=1}^{C} \left( y[i, j] \log(\text{lookup\_answer}_T[i, j]) + (1 - y[i, j]) \log(1 - \text{lookup\_answer}_T[i, j]) \right) \]

- Overall

\[ L = \frac{1}{N} \sum_{k=1}^{N} \left( [n_k = \text{True}] L_{\text{scalar}}^{(k)} + [n_k = \text{False}] \lambda L_{\text{lookup}}^{(k)} \right) \]
• Introduction
• Symbolic execution
  – (Fully supervised) Neural programmer interpreter
  – (Weakly supervised) Neural symbolic machine
  – “Spurious programs” and inductive programming
    • Learning semantic parsers from denotations
    • DeepCoder
    • More thoughts on “spurious programs”
• Distributed execution
  – Learning to execute
  – Neural enquirer
  – Differentiable neural computer
• Hybrid execution
  – Neural programmer
  – Coupling approach
Idea

- Fully neuralized model exhibits (imperfect) symbolic interpretation
- Use neural networks’ intermediate results to learn initial policy of the symbolic executor
- Improve policy by REINFORCE or MML variants
Thank you for listening

Q & A?
References


