Neural Random-Access Machine

Yuxi Liu

2018-11-7

Introduction

论文信息

标题	Neural Random-Access Machine
作者	Karol Kurah & Marcin Andrychowicz & Ilya Sutskever (Google)
发表	ICLR 2016

Inspiration

Deep learning is successful for two reasons

- able to represent the “right” kind of functions
- trainable

Inspiration

Deep learning is successful for two reasons

- able to represent the “right” kind of functions
- trainable

DNN can be improved if

- deeper and have fewer parameters
- maintaining trainability

Inspiration

Deep learning is successful for two reasons

- able to represent the “right” kind of functions
- trainable

DNN can be improved if

- deeper and have fewer parameters
- maintaining trainability

Examples:

- Neural Turing Machine
- Stack-Augmented RNN
- Grid-LSTM

Inspiration (Cont.)

Key characteristic:

- depth
- size of their short term memory
- number of parameters

are no longer confounded and can be altered independently

in contrast to models like the LSTM:

# parameters grows quadratically with the size of short term memory

Inspiration (Cont.)

A fundamental operation of modern computers is **pointer manipulation** and **dereferencing**

Thus, provide DNN with primitive operations to

- manipulate
- store
- dereference

pointers in memory

possible to train models on problems whose solutions require pointer manipulation and chasing

Model

Overview

Controller + Registers + Modules

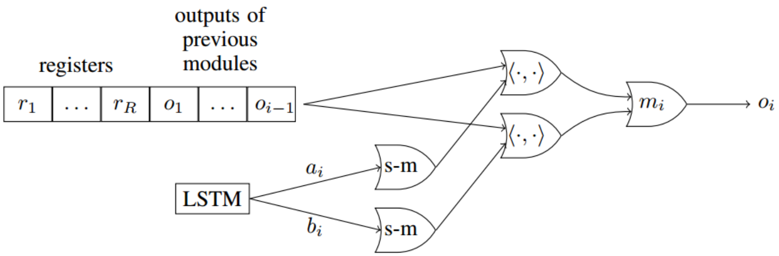

registers
outputs of previous modules
r_1
...
r_R
o_1
...
o_{i-1}
LSTM
a_i
b_i
s-m
s-m
$\langle\cdot,\cdot\rangle$
$\langle\cdot,\cdot\rangle$
m_i
o_i

Figure 1: The execution of the module m_i. Gates *s-m* represent the softmax function and $\langle\cdot,\cdot\rangle$ denotes inner product. See Eq. 1 for details.

Figure 1

High Level

Each time step:

1. Controller get **some** input from registers
2. Controller update its internal state
3. Controller outputs fuzzy circuit description
4. Values of the registers are overwritten with the outputs of the circuit

Register

R registers

- holds an integer
- **trainable** $\Rightarrow$ differentiable
    - $\Rightarrow$ interger as a distribution over $0, 1, \ldots, M-1$
    - $p \in \mathbb{R}^M, \sum_i p_i = 1$
    - no special form assumed
- controller has no direct access to registers
- interact with **predefined** modules (e.g. integer addition, equality test)

Module

Each module is a function

$$m_i \;:\; \{0,1,\dots,M-1\} \times \{0,1,\dots,M-1\} \to \{0,1,\dots,M-1\}$$

Need to generalize to probability distributions

$$\forall_{0\le c<M}$$

$$\mathbb{P}(m_i(A,B)=c) = \sum_{0\le a,b<M} \mathbb{P}(A=a)\mathbb{P}(B=b)[m_i(a,b)=c]$$

Fuzzy Circuit

For each module m_i:

- input chosen from $\{r_1, \ldots, r_R, o_1, \ldots, o_{i-1}\}$ by controller
  - r_j: value of j-th register at current time step
  - o_j: output of the module m j at the current timestep

$$o_i = m_i\Big((r_1, \ldots, r_R, o_1, \ldots, o_{i-1})^T \mathbf{softmax}(a_i), \\ (r_1, \ldots, r_R, o_1, \ldots, o_{i-1})^T \mathbf{softmax}(b_i)\Big)$$

a_i, b_i: control vectors

Update Register

$$r_i := (r_1, \ldots, r_R, o_1, \ldots, o_Q)^T \mathbf{softmax}(c_i)$$

c_i: control vectors

Summary

Controller + Registers + Modules

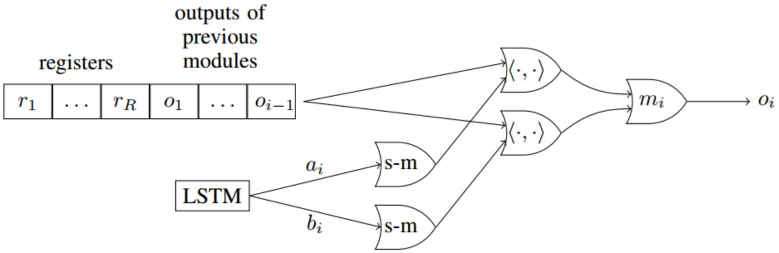

outputs of previous modules
registers
r_1
...
r_R
o_1
...
o_{i-1}
$\langle \cdot, \cdot \rangle$
$\langle \cdot, \cdot \rangle$
m_i
o_i
a_i
b_i
LSTM
s-m
s-m

Figure 1: The execution of the module m_i. Gates *s-m* represent the softmax function and $\langle \cdot, \cdot \rangle$ denotes inner product. See Eq. 1 for details.

Figure 2

Controller's Input

- Naive approach: values of registers
  - problem: values are distributions over $\mathbb{R}^M$
  - parameters will depend on M
  - M links to size of memory, **undesirable**

Controller's Input

- Naive approach: values of registers
  - problem: values are distributions over $\mathbb{R}^M$
  - parameters will depend on M
  - M links to size of memory, **undesirable**

- What we want: one scalar for each register

Controller's Input

- Naive approach: values of registers
  - problem: values are distributions over $\mathbb{R}^M$
  - parameters will depend on M
  - M links to size of memory, **undesirable**
- What we want: one scalar for each register
- Our approach: $\mathbb{P}(r_i = 0)$
  - benefits: limit information for controller
  - rely on modules, rather than solve on its own
  - if $M = 2$, then $r_i \in \{0, 1\}$, **bool** value
    - the scalar retains all information
    - e.g. output of bool module
    - such as inequality test module $m_i(a, b) = [a < b]$

Memory Tape

So far...

Can do sequence-to-sequence transformation

- initialize registers with input
- produce output to registers after certain time steps
- **disadvantage**: unable to generalize to longer sequences
  - length of sequence = number of registers
  - which is constant

Memory Tape

- Extend with a variable-size memory tape
- M memory cells, each stores a distribution over $\{1, \ldots, M\}$
  - distribution $\Leftrightarrow$ *fuzzy address*
  - cell $\Leftrightarrow$ *fuzzy pointer*
- state of memory tape, described with $\mathcal{M} \in \mathbb{R}_M^M$
  - $\mathcal{M}_{i,j}$: probability that i-th cell holds j

READ and WRITE

The model interacts with the memory tape solely using two kinds of modules:

- READ
  - 1st parameter as address, discard 2nd
  - return value of memory tape on that address
  - $\mathbf{READ}(p, a) = \mathcal{M}^T p$
- WRITE
  - 1st parameter as address, 2nd as value
  - store the value on memory tape at given address
  - $\mathbf{WRITE}(p, a)$ performs $\mathcal{M} := (J - p)J^T \cdot \mathcal{M} + pa^T$
  - J is a column vector of M ones
  - $\cdot$ is element-wise multiplication

NRAM

Controller + Registers + Modules + Memory Tape

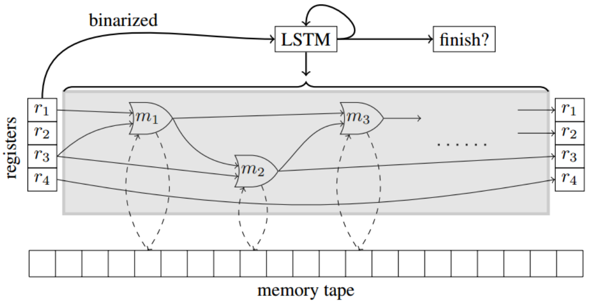


Figure 2: One timestep of the NRAM architecture with $R = 4$ registers. The LSTM controller gets the „binarized" values $r_1, r_2, \ldots$ stored in the registers as inputs and outputs the description of the circuit in the grey box and the probability of finishing the execution in the current timestep (See Sec. 3.3 for more detail). The weights of the solid thin connections are outputted by the controller. The weights of the solid thick connections are trainable parameters of the model. Some of the modules (i.e. `READ` and `WRITE`) may interact with the memory tape (dashed connections).

Input & Output

The memory tape also serves as an input-output channel

- memory initialized with input
- produce output to the memory
- each timestep, controller decides whether to finish
    - $f_t \in [0, 1]$
    - probability execution not finish at t: $\prod_{i=1}^{t-1}(1 - f_i)$
    - output produced exactly at t: $p_t = f_t \cdot \prod_{i=1}^{t-1}(1 - f_i)$
    - maximum steps T, and set $p_T = 1 - \sum_{i=1}^{T-1} p_i$

Loss Function

for each training pair (x, y)

memory tape initialized as x

loss function is the probability that we get y in memory up to T steps

$$-\sum_{t=1}^{T}\left(p_t \cdot \sum_{i=1}^{M} \log\left(\mathcal{M}_{i,y_i}^{(t)}\right)\right)$$

or

$$-\sum_{t=1}^{T} \log\left(\sum_{i=1}^{M} p_t \cdot \mathcal{M}_{i,y_i}^{(t)}\right)$$

note the loss is calculated on all memory cells which contains the output

Discretization

- Computing over distributions are costly
  - e.g. `READ` module needs $\Theta(M^2)$
- suggest (and verified) learned representation has very low entropy
- *discretized* during interference
  - each register and memory cell stores an integer
  - replace **softmax** with one-hot at the max value
- for small register numbers (e.g. $R \leq 20$)
  - because inputs to controller are just binarized values (=0?)
  - can precompute controller's output for each possible configuration

Experiment

Techniques

- Curriculum learning
- Gradient clipping
- Noise (added to gradients)
- Enforcing Distribution Constraints
    - to make sure they're still distributions
- Entropy
    - penalization for small entropy
    - decay over time steps
    - encourage out of local minimum
- Limiting the values of logarithms

Curriculum Learning

- "From easy to hard"

$$Q_\lambda(z) \propto W_\lambda(z)P(z)\ \forall_z$$

- $P(z)$: target distribution
- $W_\lambda(z)$: weight distribution at step λ

$$Q_1(z) = P(z)\ \forall_z$$

Curriculum Learning (Cont.)

Definition: We call the corresponding sequence of distributions Q_λ a **curriculum** if the *entropy of these distributions increases*

$$H(Q_\lambda) < H(Q_{\lambda+\epsilon}) \ \forall_{\epsilon>0}$$

$$W_{\lambda+\epsilon}(z) \geq W_\lambda(z) \ \forall_z, \forall_{\epsilon>0}$$

if Q_λ concentrates on a finite set of examples

then following the sequence means adding new and **hard** examples

Curriculum Leaning (Cont.)

In this paper, we denote the length of a sequence or the size of a tree as training complexity

D for difficulty, when error rate less than certain threshold, increase D

sample d according to D:

- 10%: uniformally from all possible difficulties
- 25%: uniformally from $[1, D+e]$, where e is sampled from geometric distribution, factor $= 1/2$
- 65%: $d = D + e$

Tasks

1. **Access**: Given a value k and an array A, return $A[k]$.
2. **Increment**: Given an array, increment all its elements by 1.
3. **Copy**: Given an array and a pointer to the destination, copy all elements from the array to the given location.
4. **Reverse**: Given an array and a pointer to the destination, copy all elements from the array in reversed order.
5. **Swap**: Given two pointers p, q and an array A, swap elements $A[p]$ and $A[q]$.

Tasks (Cont.)

6. **Permutation**: Given two arrays of n elements: P (contains a permutation of numbers $(1, \ldots, n)$ and A (contains random elements), permutate A according to P.
7. **ListK**: Given a pointer to the head of a linked list and a number k, find the value of the k-th element on the list.
8. **ListSearch**: Given a pointer to the head of a linked list and a value v to find return a pointer to the first node on the list with the value v.
9. **Merge**: Given pointers to 2 sorted arrays A and B, merge them.
10. **WalkBST**: Given a pointer to the root of a Binary Search Tree, and a path to be traversed (sequence of left/right steps), return the element at the end of the path.

Modules

- `READ`
- `ZERO`$(a, b) = 0$
- `ONE`$(a, b) = 1$
- `TWO`$(a, b) = 2$
- `INC`$(a, b) = (a + 1) \mod M$
- `ADD`$(a, b) = (a + b) \mod M$
- `SUB`$(a, b) = (ab) \mod M$
- `DEC`$(a, b) = (a1) \mod M$
- `LESS-THAN`$(a, b) = [a < b]$
- `LESS-OR-EQUAL-THAN`$(a, b) = [a \leq b]$
- `EQUALITY-TEST`$(a, b) = [a = b]$
- `MIN`$(a, b) = \min(a, b)$
- `MAX`$(a, b) = \max(a, b)$
- `WRITE`

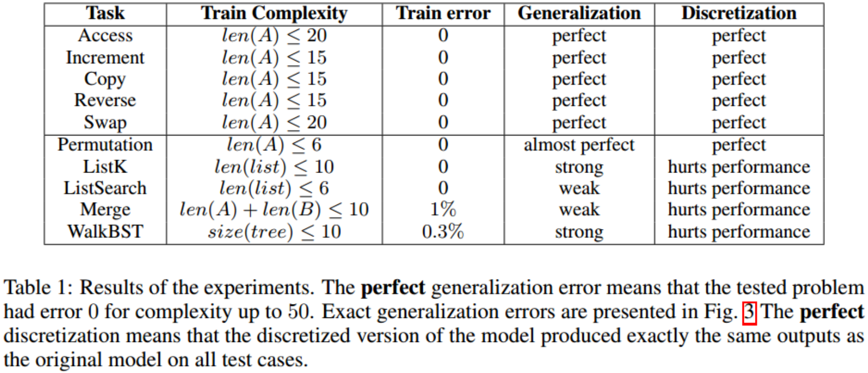
Modules (Cont.)

The setting of module sequence is pre-specified

We also considered settings in which the module sequence is repeated many times

The number of repetitions is a hyperparameter

Result



Task	Train Complexity	Train error	Generalization	Discretization
Access	$len(A) \leq 20$	0	perfect	perfect
Increment	$len(A) \leq 15$	0	perfect	perfect
Copy	$len(A) \leq 15$	0	perfect	perfect
Reverse	$len(A) \leq 15$	0	perfect	perfect
Swap	$len(A) \leq 20$	0	perfect	perfect
Permutation	$len(A) \leq 6$	0	almost perfect	perfect
ListK	$len(list) \leq 10$	0	strong	hurts performance
ListSearch	$len(list) \leq 6$	0	weak	hurts performance
Merge	$len(A) + len(B) \leq 10$	1%	weak	hurts performance
WalkBST	$size(tree) \leq 10$	0.3%	strong	hurts performance

Table 1: Results of the experiments. The **perfect** generalization error means that the tested problem had error 0 for complexity up to 50. Exact generalization errors are presented in Fig. 3 The **perfect** discretization means that the discretized version of the model produced exactly the same outputs as the original model on all test cases.

Figure 4

Result (Cont.)

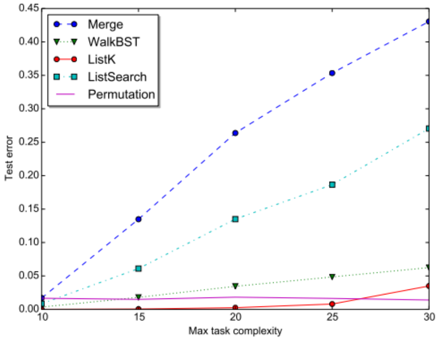


Figure 3: Generalization errors for hard tasks. The **Permutation** and **ListSearch** problems were trained only up to complexity 6. The remaining problems were trained up to complexity 10. The horizontal axis denotes the *maximal* task complexity, i.e., $x = 20$ denotes results with complexity sampled uniformly from the interval $[1, 20]$.

Figure 5

Simple tasks

first 5 tasks achieve 0 error rate in

- training
- testing
- generalization up to 50
- discretization

Copy & **Increment** are verified to generalize to arbitary length

Hard tasks

With many techniques mentioned above, achieved up to 1% error rate during training

- generalize well: **Permutation** , **ListK** & **WalkBST**
- discretize well: **Permutation**, others error rate above 70%

Comparison to existing models

“NTM can solve tasks like **Copy** or **Reverse**

but it suffers from the inability to naturally store a pointer to a fixed location in the memory

unlikely to solve tasks such as **ListK**, **ListSearch** or **WalkBST**

since the pointers used in these tasks refer to absolute positions

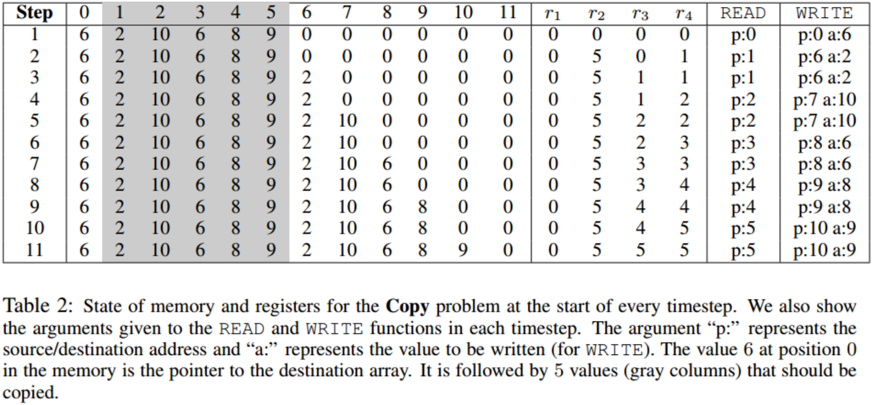
Lack of content-based addressing - delibrate design - because it slows down memory access speed

Conclusions

- NRAM: manipulation and deferencing pointers
- some tasks generalize well, even up to arbitary length

Examplary execution

Copy



Step	0	1	2	3	4	5	6	7	8	9	10	11	r_1	r_2	r_3	r_4	READ	WRITE
1	6	2	10	6	8	9	0	0	0	0	0	0	0	0	0	0	p:0	p:0 a:6
2	6	2	10	6	8	9	0	0	0	0	0	0	0	5	0	1	p:1	p:6 a:2
3	6	2	10	6	8	9	2	0	0	0	0	0	0	5	1	1	p:1	p:6 a:2
4	6	2	10	6	8	9	2	0	0	0	0	0	0	5	1	2	p:2	p:7 a:10
5	6	2	10	6	8	9	2	10	0	0	0	0	0	5	2	2	p:2	p:7 a:10
6	6	2	10	6	8	9	2	10	0	0	0	0	0	5	2	3	p:3	p:8 a:6
7	6	2	10	6	8	9	2	10	6	0	0	0	0	5	3	3	p:3	p:8 a:6
8	6	2	10	6	8	9	2	10	6	0	0	0	0	5	3	4	p:4	p:9 a:8
9	6	2	10	6	8	9	2	10	6	8	0	0	0	5	4	4	p:4	p:9 a:8
10	6	2	10	6	8	9	2	10	6	8	0	0	0	5	4	5	p:5	p:10 a:9
11	6	2	10	6	8	9	2	10	6	8	9	0	0	5	5	5	p:5	p:10 a:9

Table 2: State of memory and registers for the **Copy** problem at the start of every timestep. We also show the arguments given to the `READ` and `WRITE` functions in each timestep. The argument "p:" represents the source/destination address and "a:" represents the value to be written (for `WRITE`). The value 6 at position 0 in the memory is the pointer to the destination array. It is followed by 5 values (gray columns) that should be copied.

Figure 6

Copy (Cont.)

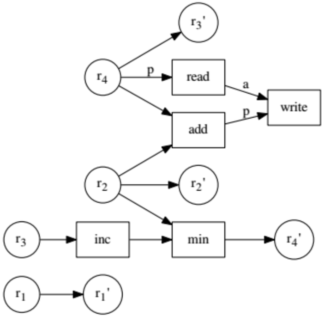


Figure 4: The circuit generated at every timestep ≥ 2. The values of the pointer (p) for READ, WRITE and the value to be written (a) for WRITE are presented in Table 2. The modules whose outputs are not used were removed from the picture.

Figure 7

Copy (Cont.)

- circuits after the 2nd step
- `r2`: offset, update to iteself, so constant
- `r3`: accumulator, increase 1 each time
- `r4`: `min(r2, r3)` reading address
- `add(r4, r2)`: writing address
- so copy one element every 2 steps until finished

Access

C.1 Access

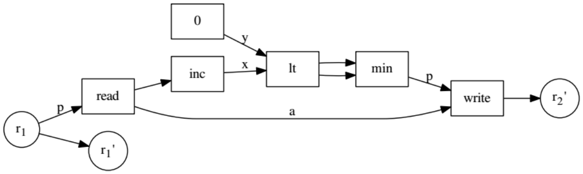


Figure 5: The circuit generated at every timestep ≥ 2 for the task **Access**.

Step	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	r_1	r_2
1	3	1	12	4	7	12	1	13	8	2	1	3	11	11	12	0	0	0
2	3	1	12	4	7	12	1	13	8	2	1	3	11	11	12	0	3	0
3	4	1	12	4	7	12	1	13	8	2	1	3	11	11	12	0	3	0

Table 3: Memory for task **Access**. Only the first memory cell is modified.

Figure 8

Access (Cont.)

circuit above generating 0 all the time, as the writing address

Increment

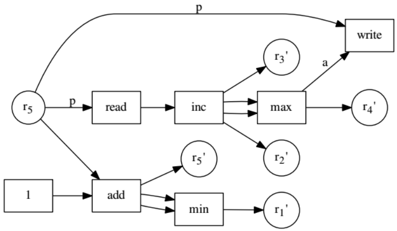


Figure 6: The circuit generated at every timestep for the task **Increment**.

Step	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	r_1	r_2	r_3	r_4	r_5
1	1	11	3	8	1	2	9	8	5	3	0	0	0	0	0	0	0	0	0	0	0
2	2	11	3	8	1	2	9	8	5	3	0	0	0	0	0	0	1	2	2	2	1
3	2	12	3	8	1	2	9	8	5	3	0	0	0	0	0	0	2	12	12	12	2
4	2	12	4	8	1	2	9	8	5	3	0	0	0	0	0	0	3	4	4	4	3
5	2	12	4	9	1	2	9	8	5	3	0	0	0	0	0	0	4	9	9	9	4
6	2	12	4	9	2	2	9	8	5	3	0	0	0	0	0	0	5	2	2	2	5
7	2	12	4	9	2	3	9	8	5	3	0	0	0	0	0	0	6	3	3	3	6
8	2	12	4	9	2	3	10	8	5	3	0	0	0	0	0	0	7	10	10	10	7
9	2	12	4	9	2	3	10	9	5	3	0	0	0	0	0	0	8	9	9	9	8
10	2	12	4	9	2	3	10	9	6	3	0	0	0	0	0	0	9	6	6	6	9
11	2	12	4	9	2	3	10	9	6	4	0	0	0	0	0	0	10	4	4	4	10

Figure 9

Increment (Cont.)

- `r5` as reading add writing address
- `r1`: accumulator

Reverse

C.3 Reverse

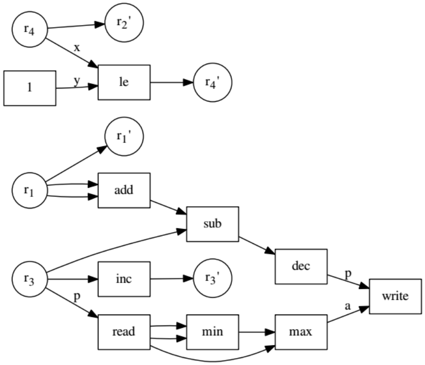


Figure 7: The circuit generated at every timestep ≥ 2 for the task **Reverse**.

Figure 10

Reverse (Cont.)

- `r3` as reading address
- `2 * r3 - 1` as writing address