Estimating Minimum Operation Steps via Memory-based Recurrent Calculation Network

Lei Sha*
MOE Key Lab of Computational Linguistics
Peking University
Beijing China
shalei@pku.edu.cn

Chen Shi*
MOE Key Lab of Computational Linguistics
Peking University
Beijing China
shichen@pku.edu.cn

Qi Chen
Microsoft Research
Beijing China
cheqi@microsoft.com

Lintao Zhang
Microsoft Research
Beijing China
lintaoz@microsoft.com

Houfeng Wang
MOE Key Lab of Computational Linguistics
Peking University
Beijing China
wanghf@pku.edu.cn

***Abstract*—To estimate time complexity for a given algorithm is important for algorithm designers. Usually, time complexity means the "analytical" time complexity which needs to be proved by strict math derivation. We propose to estimate the "numerical" time complexity (NTC), which measures the minimum number of operations an algorithm has to spend, as well as capture the intrinsic laws of time complexity. The unique challenges include: (1) How to make a machine learning model has the same ability as a real-world CPU (2) How to measure the minimum number of required arithmetic operations for a given problem. To tackle these challenges, we first propose a memory-based recurrent calculation network to mimic the functions of CPU and then we propose a self-adaptive selection gate for deciding when the mimic calculation process should stop. In addition, we use a symbolic learning method to find the time complexity formula. We train and test our model on four basic algorithms: long integer addition, 1-dim max-pooling, outer product, and sorting. Experiment results demonstrate that our model can precisely predict the numerical time complexity as well as the time complexity formula for each algorithm. We also conduct many visualizations to prove the effectiveness and correctness of our model.**

I. Introduction

When designing an algorithm, we have to try hard to make the time complexity as low as possible. But the analytical time complexity has to be calculated via a lot of math derivation. To alleviate the burden of time complexity calculation, we propose a new task, which estimates the numerical time complexity [NTC] (the minimum number of operation steps) for a given problem.

In our task, we take the input-output examples of a certain algorithm as the input of our model, and the model should be designed to answer the two questions: "how many operations at least should the given problem cost" and "what intrinsic law of time complexity does the given problem has."

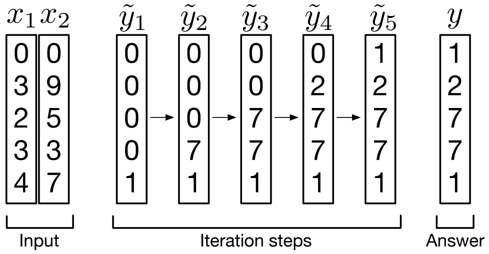


Fig. 1. An example of estimating the numerical time complexity of adding 5-digit numbers.

As far as we have studied, no previous work has proposed any methods for estimating time complexity or capturing the "intrinsic laws" of time complexity for a given problem. To take a specific example, in Figure 1, there is an addition task for 4-digit numbers. We would like to take one-digit addition as an elementary operation, and we can see that in Figure 1, it takes us 5 steps to obtain the final result. So in this case, the numerical time complexity is 5. Similarly, for 5-digit numbers addition, the numerical time complexity would be 6. When a model takes different lengths of input into consideration, it may find that n-digit addition task may have a time complexity of $n + 1$.

In general, there are several challenges for designing a numerical time complexity estimator:

(1) The generalization ability to different kinds of arithmetic operations: we hope that our model can act as a real CPU, which is easy to be generalized to many arithmetic operations.

(2) The judgment of when the program mimic process should stop: we hope that our model can find a proper stop point where the given problem can be solved successfully (which means the calculated result of each given input is exactly the same as the given output).

Some end-to-end approaches can learn simple arithmetic operations in a step-by-step manner, such as neural GPU [10], Neural Turing machine [5] and Lie-Access Neural Turing machine [16]. However, their architectures did not fully mimic the real CPU's architecture, which makes them fail to complete

*The first two authors contributed equally to this work.

978-1-7281-6926-2/20/$31.00 ©2020 IEEE

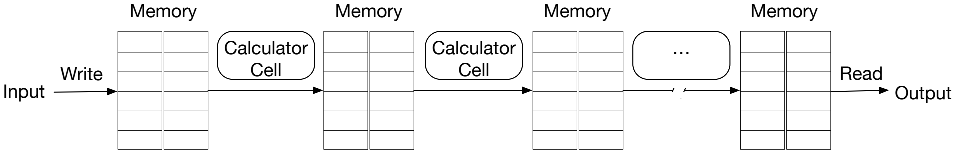


Fig. 2. The procedure of an algorithm composed of neural elementary operation.

some particular tasks such as max-pooling, outer product and sort.

To tackle these challenges, we follow the model of Neural Turing Machine [5] and propose a memory-based recurrent calculation network (MRCN) to estimate the numerical time complexity. MRCN use an LSTM-RNN architecture to mimic the performing process of assembly instructions. On the other hand, a series of memory network structures are also applied to mimic memories and registers of a real CPU. The neural mimic of CPU guarantees that MRCN can handle different kinds of arithmetic operation. In addition, we designed a self-adaptive selection gate for finding the numerical time complexity of the given problem. We conduct experiments on four basic algorithms: long integer addition, 1-dim max-pooling, outer product and sorting, and our model achieves good performance on all the four problems. Our model not only correctly predicts the numerical time complexity of the four problems, the general equations of time complexity are also captured.

II. Related Work

To the best of our knowledge, we are the first to research about estimating time complexity for algorithms. Similar research topics includes neural program generation [2], [3] and neural algorithm learning [1], [5], [10], [11], [13], [14].

Neural program generation means to use neural network methods to generate code lines for specific problems. AI programmer [3] uses a genetic algorithm to search for a code sequence which can lead to the correct output. Deep coder [2] use a similar procedure, which takes neural network as a guide to search for a program with a set of input-output examples. However, due to the large search space, search based program generating methods have to limit the possible operations to a small scope and prevent the generated program from being too long.

Neural algorithm learning methods tend to learn "implicit code" and information of these codes are hidden in the neural network weights. Nearly all of these kinds of works take memory architecture as a component of the model. Neural Turing Machines (NTM) [5] is a differentiable end-to-end model which learn a series of read, write and memory access sequence. NTM uses attention mechanism to access the memory. The reinforcement learning version of NTM [18] uses reinforce algorithm to learn a "hard attention" to access the memory and has achieved reasonable performance. On the other hand, hierarchical attentive memory [1] uses hierarchical attention to learn a series of basic sequence transformations. Differentiable neural computer (DNC) [6] has multiple read heads and uses a "linkage-based addressing" mechanism to track consecutively used memory slots, which is an improvement of NTM. Also, Lie Access Neural Turing Machine (LANTM) [16] stores the memories in a Euclidean key space and uses a Lie group to manipulate the access to the memory.

Stack, Queue, DeQueue networks [7], [9] extends LSTM with a stack, a FIFO queue or a double-ended queue and has achieved good performance on several tasks such as sequence copying and bigram flipping.

Neural RAM [13] controls the shifting of pointers to an external variable-size random-access memory. Similarly, [17] use pointers to access non-differential memories and use reinforcement learning to train this model. An advantage is that they can guarantee constant time memory access.

Grid LSTM [11] and Neural GPU [10] can learn to multiply 15-digit decimal numbers. Among them, Neural GPU is especially suited for parallel computation. These models can be trained end-to-end and can correctly work out simple arithmetic problems.

Also, there are models which need to be trained by synthetic execution traces, such as NPI [15] and recursive NPI [4]. However, due to their requirement of strong supervision in the form of execution traces, they are not easy to be trained only be input-output examples.

Different from them, our work focuses on estimating the numerical time complexity for a given problem. In addition, we propose to use a symbolic learning method to capture the intrinsic law of the time complexity of a given problem. Our model mimics the currently multi-core CPU which have multiple read heads and multiple write heads with a register structure for the storage of intermediate result. These mechanics make it possible for our model to complete an elementary operation in one read-write action.

III. Estimating the Numerical Time Complexity

A. Basic Calculation Process

According to its definition, time complexity is usually estimated by counting the number of elementary operations performed by an algorithm. Since an elementary operation takes a fixed amount of time to perform, the amount of time cost by the algorithm and the number of elementary operations differ by at most a constant factor.

To avoid the math derivation when computing analytical time complexity, we propose to count the number of necessary elementary operations for numerical time complexity. Usually, elementary operation means a set of operations including reading from memory and writing into the memory. Therefore, in

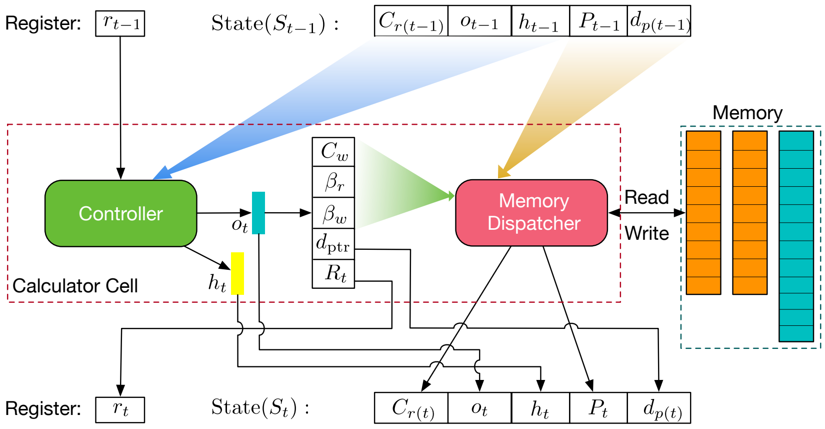


Fig. 3. The general architecture of MRCN.

this scenario, an elementary operation can represent nearly every one-digit arithmetic operation such as add, minus, multiply, shift, bitwise, compare, etc. Thus an algorithm can be regarded as a combination of many such elementary operations.

In this paper, we use a calculator cell to model the elementary operation as a neural version. After the input of the algorithm has been written into the memory, we use a calculator cell to decide on which place should the information be read from and where the calculated result should be written into. This process can continue for several iterations, and then we read the memory for the final result. The number of iterations is the numerical time complexity of the given problem. So our task is to estimate the minimum number of iterations under the precondition that the neural algorithm can make every example right. The process of the neural algorithm is shown in Figure 2.

B. Memory-based Recurrent Calculation Network

In our model, the calculator cell is composed of a controller and a memory dispatcher, and it uses a register and a state to manipulate the read and write operations on the memory as is shown in Figure 3 . The memory M is a multi-column memory with $|r|$ columns for read and $|w|$ columns for write. Some columns are both for read and write, so the actual column number can be less than $|r| + |w|$. The size of the memory is $|M|$. Similar to the real CPU architecture, the register vector r_t represents for a group of registers, which records some intermediate result. The state records the calculation process which include 5 variants: the read content $C_r(t)$, the output gate o_t, the hidden state h_t, the pointer position P_t and the deviant of pointer $d_{p(t)}$. The read content $C_r(t)$ represents what has just been read from the memory. o_t and h_t are the same as in LSTM cell [8]. The pointer position P_t contains a bunch of read pointers and write pointers: $P_t = [r_1, \cdots, r_{|r|}, w_1, \cdots, w_{|w|}]$, these pointers represents the memory position where the model intends to read or write. And the deviant of pointer $d_{p(t)}$ has the same size as P_t, which measures the movement of each pointer in each time step.

Note that register is very important for many tasks like add and sort because they always require a structure to store the carry bit or the swap operation's temporary variant. On the other hand, single memory would make the read/write heads confused about the storage boundary of different inputs and output. Our multi-column memory provides more reasonable data storage so that tasks like outer product can be easily completed.

1) Controller: The controller is an LSTM cell, which takes the register and part of the state (read content, output gate and hidden state) as input. We concatenate all the inputs together as:

$$I_{t-1} = [r_{t-1}, C_{r(t-1)}, o_{t-1}, h_{t-1}] \tag{1}$$

And the controller's output and hidden state can be calculated as:

$$\begin{bmatrix} i_t \\ f_t \\ o_t \\ \widetilde{c}_t \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{bmatrix} (W_L \cdot \begin{bmatrix} h_{t-1} \\ I_{t-1} \end{bmatrix} + b_L) \tag{2}$$

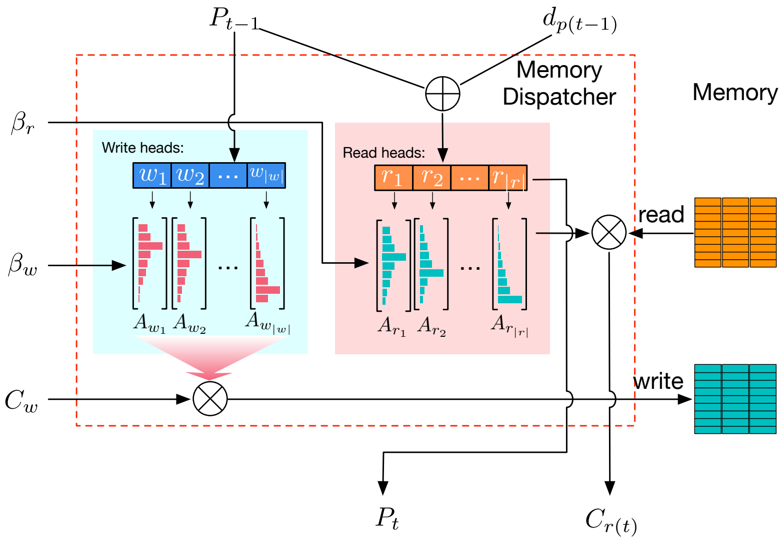
$$c_t = i_t \odot \widetilde{c}_t + f_t \odot c_{t-1} \tag{3}$$

$$h_t = o_t \odot \tanh(c_t) \tag{4}$$

Where i_t, f_t and $\widetilde{c}_t$ represent for the LSTM's input gate, forget gate and memory cell, respectively.

Since the controller manipulates everything in the calculation process, we need to use a linear layer to analyze the implicit information in o_t. As is shown in Figure 3, we have five different variants in o_t: the write content C_w, two temperature variants β_r, β_w, the deviant of pointer d_{ptr} and the register value R_t. C_w is the value what controller is going to write into the memory. β_r and β_w decides how "sharp" is a distribution, the larger β_* is, the sharper the distribution is. d_{ptr} represents the pointer movement of the next time step. R_t will be written into the register as the latest value.

2) Memory dispatcher: As is illustrated in Figure 3, the memory dispatcher takes the write content, the current pointer position, the pointer deviant, and the two temperature variants as input. The memory dispatcher controls the reading and writing operations on multi-column memory according to the



P_{t-1}
$d_{p(t-1)}$
Memory Dispatcher
Memory
β_r
Write heads:
w_1 w_2 $\cdots$ $w_{|w|}$
Read heads:
r_1 r_2 $\cdots$ $r_{|r|}$
β_w
A_{w_1} A_{w_2} $A_{w_{|w|}}$
A_{r_1} A_{r_2} $A_{r_{|r|}}$
read
write
C_w
P_t
$C_{r(t)}$

Fig. 4. The architecture of memory dispatcher.

read heads and write heads. The read heads are calculated according to the sum of current pointer P_{t-1} and the deviant $d_{p(t-1)}$, we select the read pointers $r_1, \cdots, r_{|r|}$ from the sum and transform them to a series of sharp distributions as is shown in Figure 4. The distribution $A_{r_i} \in \mathbb{R}^{|M|}$ is calculated as follows:

$$A_{r_i} = \frac{\exp\{s_r(i)\}}{\sum_j \exp\{s_r(j)\}}; \quad s_r(j) = -\beta_r (j - r_i)^2 \tag{5}$$

According to Eq 5, in vector A_{r_i}, the value corresponds to position r_i is the largest and it becomes smaller as the distance to position r_i gets longer. Specifically, β_r is a trainable parameter, which can be used to tune the sharpness of the distribution, A_{r_i} will become closer and closer to one-hot vector as β_r gets larger and larger. Therefore, when we multiply the distribution A_{r_i} and the i-th column readable memory, we obtain the i-th read value:

$$C_{r(t)}(i) = A_{r_i} * M_r(i) \tag{6}$$

Similarly, the write heads are calculated according to the current pointer P_{t-1}, and the write distribution is calculated as:

$$A_{w_i} = \frac{\exp\{s_w(i)\}}{\sum_j \exp\{s_w(j)\}}; \quad s_w(j) = -\beta_w (j - w_i)^2 \tag{7}$$

The write content C_w contains $|w|$ values. We write the i-th content into the memory as follows:

$$M_{w(t)}(i) = C_w(i) * A_{w_i} + M_{w(t-1)}(i) * (1 - A_{w_i}) \tag{8}$$

Besides, we would like to point out that our MRCN is inspired by the Neural Turing Machine [NTM] [5] and Differential Neural Computer [DNC] [6]. NTM contains a controller to manipulate the movement of a read head and a write head. Likewise, we design the controller as a much more powerful component mimics the currently multi-core CPU, which can simultaneously manipulate the movement of multiple read heads and multiple write heads by the pointer deviant. DNC uses a "linkage-based addressing" to tract consecutively used memory slot. Although efficient in some tasks, it is different from the architecture of a real CPU, which is not convenient for us to estimate the time complexity. Therefore, we use the deviant of the pointer as a mimic of the real CPU architecture.

C. Self-adaptive Selection Gate

We propose a self-adapted selection gate to select a minimum time step number to finish a given problem.

For estimating the numerical time complexity, we first assume that the time complexity satisfies an equation, an example of polynomial complexity is shown as follows:

$$\mathcal{C}(n) = n^a + b \tag{9}$$

where n is the input vector's length, a and b are trainable parameters. Also, the complexity function can be logarithm-like: $\mathcal{C}(n) = n^c(\log_2 n)^d + b$. NTC is the value of $\mathcal{C}(n)$.

Then we set an maximum step number L, which guarantees $\mathcal{C}(n) < L$. And then we use the same method as in Eq 5 and Eq 7 to generate a self-adaptive selection gate $g_n \in \mathbb{R}^L$ with the sharpest position at $\mathcal{C}(n)$.

$$g_n = \frac{\exp\{s_g(i)\}}{\sum_j \exp\{s_g(j)\}}; \quad s_g(j) = -\beta(j - \mathcal{C}(n))^2 \tag{10}$$

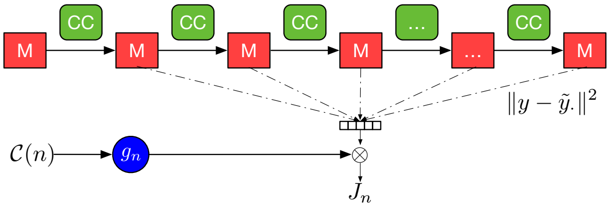


Fig. 5. The gate selection process. "CC" represents for the calculator cell, "M" represents for the memory.

where β is a trainable parameter. As is shown in Figure 5, in each time step t, we can read an output vector $\tilde{y}_t$ from the writable memory. Given the input-output example (x, y), we can calculate the loss at input length n as:

$$J_n = \sum_t ||y - \tilde{y}_{nt}||^2 g_n(t) \tag{11}$$

During the training process, g_n can be tuned to guarantee the sharpest position has the lowest loss.

D. Training

For each input length n, we have a corresponding training objective J_n. We split the input x into separate digits. Note that x may contain multiple numbers,

$$\begin{aligned} x &= \{x_1, x_2, \cdots\} \\ &= \{\{x_{1(1)}, \cdots, x_{1(n)}\}, \{x_{2(1)}, \cdots, x_{2(n)}\}, \cdots\} \end{aligned}$$

where $x_{i(j)}$ is a 10-dim one-hot vector represents for the j-th digit (count from the lowest digit) of the i-th input number. For example, the first digit of number 12345 is 5. After we input one digit, we put it into the corresponding readable memory by directly copying. The training input-output examples can be of any length, and then for each length n we use calculator cell to deal with the memory for L_n times. Finally, we use the self-adaptive selection gate to obtain J_n. The general loss function is as follows:

$$J = \sum_i N_i \Big(J_i + \lambda C(i) \Big) \tag{12}$$

where N_i is the number of examples with length i. The regularization term is to find the minimal numerical time complexity. We used Adam [12] as the optimization algorithm.

IV. Experiments

In this section, we conduct experiments to show that our model can successfully learn arithmetic algorithms and exactly predicts the numerical time complexity. We start with four tasks we focused on: long integer addition, 1-dim max-pooling, outer product, and sorting. For evaluation, we create standard datasets for each of the four tasks. Then, to demonstrate the generality of the model, we present a series of "train short test long" experiments to show its effect on much more challenging tasks.

	add	pool	outer product	sort
Neural GPU	$\surd$	$\surd$	−	−
NTM	−	−	−	$\surd$
Lie-NTM	$\surd$	−	−	$\surd$
MRCN-reg	−	$\surd$	$\surd$	−
MRCN (1M)	$\surd$	−	−	−
MRCN	$\surd$	$\surd$	$\surd$	$\surd$

TABLE I
THE PERFORMANCE OF DIFFERENT CALCULATION MODELS ON 4 TASKS. $\surd$ MEANS THE MODEL CAN ACHIEVE AN EM OF 100% ON THIS TASK. "-REG" MEANS TO EXCLUDE THE REGISTER STRUCTURE. "(1M)" MEANS ONLY HAVE ONE MEMORY, THE READ HEADS AND WRITE HEAD ARE ALL ON THE SAME MEMORY.

A. Dataset

Since the four tasks are all concrete arithmetic algorithms with a precise definition, it's very easy to automatically generate a large dataset. For different tasks, the digit numbers of input and output may be different. Given two integer inputs of length n, the output of the addition algorithm may have length $n+1$, the outer product algorithm may have length n^2. For the pooling and sorting algorithm, there is only one input, both input and output have length n. In our experiments, n can be $4, 5, \cdots, 10$, and for the four tasks, we randomly sample 35,000 input-output examples for each length of numbers and split the examples to training set (20,000 cases), dev set (5,000 cases) and test set (10,000 cases).

B. Evaluation Metrics

To evaluate our model as a neural calculator, we leverage the exact match [EM] as evaluation metric, which measures the percentage of model outputs that match the ground truth outputs exactly. Macro-averaged digital accuracy measures the average overlap between the model output and ground truth output. We treat the model output and the ground truth as bags of tokens, and compute the accuracy of corresponding positions, then average over all of the examples. Micro-averaged digital accuracy treats all model outputs and all ground truth outputs as a whole bag of tokens and compute the accuracy of corresponding positions.

C. Model Ability

According to Table I, we compared the performance of our model with some baseline models. We can see that MRCN is the only model which can perfectly complete all of the four tasks. Some ablation tests are also conducted. Without the register structure, our model cannot perform well on add and sort tasks. When we use only one memory, we found that our model cannot complete all tasks except for addition. To conduct time complexity estimation, we must first guarantee that the model is able to achieve 100% EM. So according to the above result, MRCN is the most appropriate model.

D. Addition

We choose long integer addition task to test our model because addition is a fundamental task, and for n-digit addition, the time complexity is $O(n)$. Addition task needs two readable memories, each has a read head, and a writable memory with a write head. Usually, we add numbers of the same length, but 0s are allowed to appear at the start. For example, $62345 + 82861 = 145206$.

As is shown in Table II, we use three possible time complexity equations to see if our model can predict the parameters exactly. All of these parameter estimation results are recorded when the model achieves an EM of 100% on the test set. According to the polynomial equation results $[n^a + b]$, when we use single-length examples to train our model (add@4, $\cdots$, add@10) and use single-length examples to test, our model tends to predict the correct NTC for all of the experiments. However, although NTC is correct, there are big differences

	$n^a + b$			$n^c(\log n)^d + b$				$n^a + n^c(\log n)^d + b$				
	a	b	NTC	c	d	b	NTC	a	c	d	b	NTC
add@4	1.142	0.311	5	1.436	-0.367	-1.259	5	1.018	0.206	-0.404	-0.073	5
add@5	0.933	1.391	6	0.897	0.704	0.452	6	-1.256	1.166	0.202	-0.866	6
add@6	1.042	0.763	7	0.891	0.450	0.647	7	0.926	0.501	-1.756	0.398	7
add@7	1.013	0.391	8	0.889	0.333	0.623	8	1.031	-0.043	-0.299	0.221	8
add@8	1.082	-0.401	9	0.030	2.827	0.974	9	-0.654	0.920	0.235	0.377	9
add@9	1.023	0.781	10	0.198	2.230	0.629	10	0.304	0.765	0.379	0.332	10
add@10	0.983	1.234	11	0.511	1.367	0.885	11	1.000	-0.036	0.674	-0.464	11
add@4,5 ⇒ 5	1.000	0.582	6	0.665	1.394	0.469	6	0.346	0.498	1.365	0.141	6
add@4,5,6 ⇒ 6	1.091	0.310	7	0.891	0.450	0.647	7	-1.251	1.190	-0.425	0.474	7
add@4-10 ⇒ 8	1.010	1.102	9	1.101	-0.241	0.976	9	-0.591	0.800	0.561	0.587	9

TABLE II
THE PARAMETER ESTIMATION OF THREE POSSIBLE TIME COMPLEXITY FORMULAS FOR INTEGER ADDITION TASK. NTC (NUMERICAL TIME COMPLEXITY) REPRESENTS THE MINIMUM NUMBER OF ELEMENTARY OPERATIONS. ALL OF THESE PARAMETER ESTIMATION RESULTS ARE RECORDED WHEN THE MODEL ACHIEVES AN EM OF 100% ON THE TEST SET.

between the predicted equations and the ground truth equation $\mathcal{C}(n) = n + 1$. When we use multiple-length examples to train our model (add@4,5 ⇒ 5, add@4,5,6 ⇒ 6, add@4-10 ⇒ 8), we found that the predicted equation is much closer to the ground truth. Especially, we found that larger variety of different lengths of training examples may lead to more accurate estimation, for example, "add@4-10 ⇒ 8" estimates the time complexity as $n^{1.01} + 1.10$, which is very close to $n + 1$.

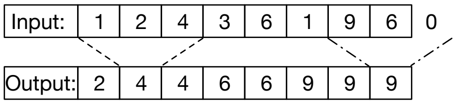
We also use logarithm-like equation $[n^c(\log n)^d + b]$ for evaluation. When using single-length training examples, the model tends to tune parameters c, d, b to make NTC correct. And multi-length training examples can contribute to finding a "general law" of the algorithm's time complexity. For example, "add@4-10 ⇒ 8" tends to predict the exponential of n to be close to 1 and the exponential of $\log n$ to be close to 0, which fits for our expectation.

For testing our model on a more general perspective, we combine the polynomial equation and the logarithm-like equation together as $n^a + n^b(\log n)^c + d$ to see if our model can find the correct law. The result in Table II turn out to be unexpected but reasonable. We compare the predicted equations by "add@4,5,6 ⇒ 6" and "add@4-10 ⇒ 8" with the ground truth equation in Figure 6. Easy to show, the equation trained by length $4, 5, 6$ is very close to the line of ground truth, especially when n is near 4, 5 and 6. On the other hand, the equation trained by $4-10$ is nearly the same as the ground truth when $n \in [1, 10]$. These experiment results illustrate that our model can exactly predict the time complexity of long integer addition task.

We also visualize the movement of the read heads and write head in Figure 7. There are two examples of length 4 and length 10. According to the figure, we found that the two read heads read two operands from two readable memories and write the computing result to the corresponding position in the writable memory in the next elementary operation, which is the same as the addition procedure.

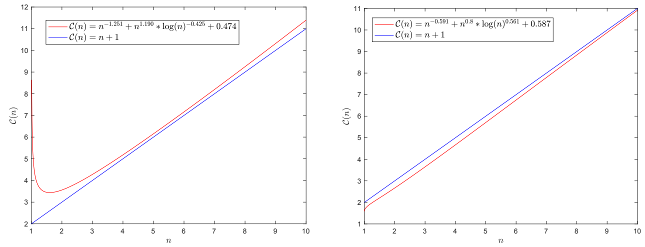
E. 1-dim Max-pooling

The second task is 1-dim max-pooling, for length n input, the complexity is $O(n)$. In this task, we have one readable memory and one writable memory. The difference between max-pooling and the previous task is that max-pooling requires multiple read heads in one readable memory. We set the pooling window to 3, so there are three read heads in the readable memory. An example of max-pooling is shown as follows, if there is not enough element in the pooling window, we would pad it with 0:



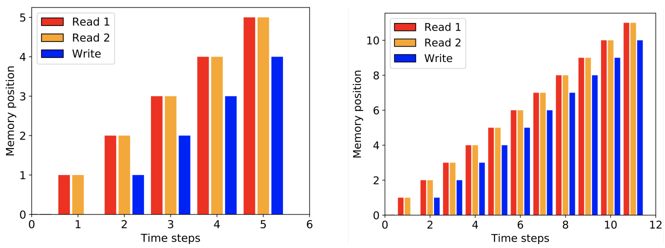
Input: 1 2 4 3 6 1 9 6 0

Output: 2 4 4 6 6 9 9 9



(a) add@4,5,6 ⇒ 6 (b) add@4-10 ⇒ 8

Fig. 6. The comparison of time complexity equations: predicted vs. ground truth.



(a) add@4 (b) add@10

Fig. 7. Long integer addition task: The visualization of read heads' and write head's movement. (a) is for length 4 and (b) is for length 10.

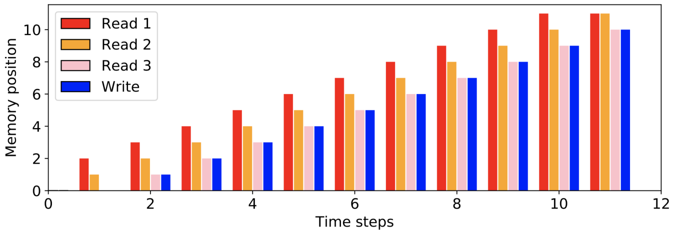


Fig. 8. 1-dim max-pooling task with window length 3: The movement of the three read heads on the readable memory and the write head on the writable memory. The length of the input is 8.

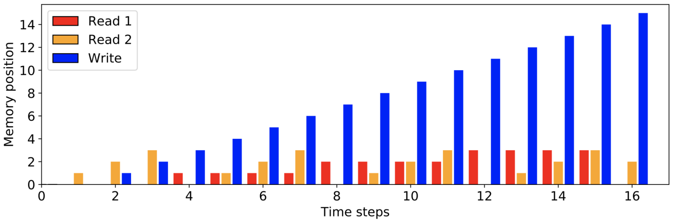


Fig. 9. Outer product task: The visualization of the read heads' and write head's movement. The length of input is 4 and the output is 16.

For simplicity, we only list the parameter estimation result for polynomial time complexity equation in Table III. Since the ground truth is $\mathcal{C}(n) = n$, our model with multiple-length training examples ("pool@4,5,6 ⇒ 6" and "pool@4-10 ⇒ 8") has made a very accurate estimation. The predicted NTC is also exactly as the ground truth.

We also visualize the movement of the read heads and the write heads in the max-pooling task. As shown in Figure 8, at the beginning of the calculation process, the three read heads are all at memory position 0. After a time step, the three read heads automatically scattered to three adjacent positions (which is exactly the way max-pooling does). And after dealing with the three read contents with the calculator cell, the write contents would be written into the middle position of the three read heads' position one-time step later, which is also the same as the max-pooling task. This experiment shows our model's power in the max-pooling task.

F. Outer Product

Outer product is the tensor product of two coordinate vectors. For two vectors u and v, their outer product $u \otimes v$ is a matrix w which satisfies $w_{ij} = u_i v_j$. Its time complexity is $O(n) = n^2$. Similar to the addition task, outer product also has two readable memory storing two input vectors respectively, and a writable memory prepared for the result. Each readable memory has a read head and the writable memory has a write head. The time complexity equation learning result is shown in Table IV. We can see that the model trained by multiple-length examples figures out the intrinsic law of outer product. The time complexity estimated by "out@4,5,6" ($\mathcal{C}(n) = n^{1.9998} + 0.1411$) is very close to the ground truth. In Figure 9, we visualize the movement of the read heads and write head. Their movement also fits with our anticipation. In particular, the first read head keeps in the same position waiting for the second read head to iterate over the input sequence, which means the controller can learn to manipulate read heads to stop moving forward at a proper time.

	$n^a + b$		
	a	b	NTC
pool@4	0.852	1.220	4
pool@5	0.982	0.254	5
pool@6	1.051	-0.418	6
pool@7	0.995	0.471	7
pool@8	0.963	0.581	8
pool@9	0.917	1.247	9
pool@10	1.046	-0.715	10
pool@4,5 ⇒ 5	1.075	-0.384	5
pool@4,5,6 ⇒ 6	0.983	0.026	6
pool@4-10 ⇒ 8	1.000	-0.107	8

TABLE III
THE PARAMETER ESTIMATION RESULT FOR 1-DIM MAX-POOLING TASK WITH WINDOW SIZE 3. ALL RESULTS ARE RECORDED WHEN THE EM ACHIEVES 100%.

G. Sorting

In the sorting task, we rearrange a given array to ascending order. In our model, it needs two memories, one is readable and the other is both readable and writable since sorting needs to compare between input and output elements. The averaged time complexity of sorting is $\Theta(n \log n)$ (quick sort (avg), heap sort and merge sort), the worst-case complexity is $O(n^2)$ (bubble sort, insertion sort, quick sort (worst)).

For comparison, we also calculate the NTC of quicksort. We take swap operation and compare operation as elementary operations. Then we randomly sample more than 100,000 examples and use quick sort to rearrange the input sequence to ascent order. We compute the average number of elementary operations during quick sort process and list the result in the last column of Table V.

In Table V, we use three different equations to fit the time complexity of sorting. In the experiment result of equation $\mathcal{C}(n) = n^a + b$, we find there is a big difference among different sets of parameters estimated by different model settings. For example, a (the exponent of n) ranges from 1.49 $\sim$ 1.63. Even in the experiments with multiple-length training input, the difference between highest value and lowest value of a

	$n^a + b$		
	a	b	NTC
out@4	2.0418	-0.5654	16
out@5	1.9591	1.4151	25
out@6	2.0069	-0.0274	36
out@4,5 ⇒ 5	1.9834	0.3274	25
out@4,5,6 ⇒ 6	1.9998	0.1411	36

TABLE IV
THE PARAMETER ESTIMATION FOR OUTER PRODUCT. ALL RESULTS ARE RECORDED WHEN THE EM ACHIEVES 100%.

	$n^a + b$			$n^c(\log n)^d + b$				$an^2 + bn + c$				Quick	Bubble
	a	b	NTC	c	d	b	NTC	a	b	c	NTC	Sort	Sort
sort@4	1.5330	0.0283	8	1.1790	1.1790	0.1713	8	0.6634	-0.3366	0.1634	9	9.533	6
sort@5	1.5120	-0.1030	11	1.0176	1.0176	0.0167	12	0.5489	-0.4512	-0.1489	15	13.454	10
sort@6	1.5784	-0.4175	16	1.0133	1.0133	0.3150	16	0.5473	-0.4827	-0.4728	16	17.728	15
sort@7	1.6012	0.1233	23	1.0373	1.0373	0.0359	22	0.5119	-0.4181	-0.0181	22	22.331	21
sort@8	1.5917	0.9523	28	1.0560	1.0561	-0.2125	28	0.5146	-0.4984	-0.9741	28	27.082	28
sort@9	1.5868	0.2334	33	1.0568	1.0568	0.0558	35	0.4382	-0.2618	-0.0618	33	32.301	36
sort@10	1.5816	-0.2336	38	1.0416	1.0416	-0.0473	38	0.4788	-0.6194	-0.0695	42	37.512	45
sort@4,5 ⇒ 5	1.4825	0.3599	11	0.9513	0.9513	-0.0499	10	0.4637	-0.4059	0.2092	10	13.454	10
sort@4,5,6 ⇒ 6	1.5358	0.6740	16	0.9964	0.9963	-0.0026	15	0.5033	-0.4904	0.0161	15	17.728	15
sort@4-10 ⇒ 8	1.6031	0.9631	29	1.0552	1.0552	0.6138	29	0.5230	-0.466	-0.7964	29	27.082	28

TABLE V
THE PARAMETER ESTIMATION RESULT OF THREE POSSIBLE TIME COMPLEXITY FOR SORT TASK. ALL OF THESE PARAMETER ESTIMATION RESULTS ARE RECORDED WHEN THE EM ACHIEVES 100%.

is about 0.2, which is much larger than 0.09 in addition task and pooling task. This phenomenon shows that equations with the form $n^a + b$ do not fit for the sorting task. According to the parameter estimation result of logarithm-like equation $n^c(\log n)^d + b$ in Table V, we find our model tends to keep the exponent of n and $\log n$ to be very close. In addition, when the length of the input sequence is short (shorter than 8), the NTC estimated by our model is less than the quicksort, and when the length of the input sequence is longer than 8, our NTC is larger than quicksort. This phenomenon shows that although our model is faster than quicksort algorithm in short-length sequence sorting, the complexity of the sorting algorithm designed by our model is still a bit larger than quicksort algorithm. When dealing with long sequences, our model cannot beat quicksort. This experiment tells us that the sorting algorithm found by our model is slower than a $n \log n$ algorithm like quicksort. We also use quadratic equation $\mathcal{C}(n) = an^2 + bn + c$ to test our model, which is in order to learn bubble sort (which has the complexity of $0.5n^2 - 0.5n$). According to the estimation result, we found that when the length of test examples is short (shorter than 8), the parameters tend to be higher than bubble sort (0.5 and -0.5). When the length of test examples are longer than 8, the estimated parameters tend to be lower than bubble sort. This phenomenon tells us that the sort algorithm our model has learned is better than the bubble sort in long sequences.

V. CONCLUSION

In this paper, we propose a computation model which mimics the procedure of CPU when solving arithmetic problems. We design a controller and a memory dispatcher as our calculator cell. Then we propose a self-adaptive selection gate to predict the numerical time complexity and estimate the parameters of time complexity equations. We use four arithmetic tasks to test our model: long integer addition, 1-dim max-pooling, outer product and sorting. We estimate the parameters of time complexity equations for the four tasks and found that our model can precisely find the general laws of these tasks and predict the correct equations. We also visualize the movement of read heads and write heads of some tasks, and they appear to move exactly as we have expected.

ACKNOWLEDGEMENTS

Our work is supported by the National Key Research and Development Program of China under Grant No.2017YFB1002101.

REFERENCES

[1] M. Andrychowicz and K. Kurach. Learning efficient algorithms with hierarchical attentive memory. *arXiv preprint arXiv:1602.03218*, 2016.

[2] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[3] K. Becker and J. Gottschlich. AI programmer: Autonomously creating software programs using genetic algorithms. *arXiv preprint arXiv:1709.05703*, 2017.

[4] J. Cai, R. Shin, and D. Song. Making neural programming architectures generalize via recursion. *arXiv preprint arXiv:1704.06611*, 2017.

[5] A. Graves, G. Wayne, and I. Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[6] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

[7] E. Grefenstette, K. M. Hermann, M. Suleyman, and P. Blunsom. Learning to transduce with unbounded memory. In *Advances in Neural Information Processing Systems*, pages 1828–1836, 2015.

[8] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 98:1735–80, 1997.

[9] A. Joulin and T. Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in neural information processing systems*, pages 190–198, 2015.

[10] Ł. Kaiser and I. Sutskever. Neural GPUs learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

[11] N. Kalchbrenner, I. Danihelka, and A. Graves. Grid long short-term memory. *arXiv preprint arXiv:1507.01526*, 2015.

[12] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[13] K. Kurach, M. Andrychowicz, and I. Sutskever. Neural random-access machines. *arXiv preprint arXiv:1511.06392*, 2015.

[14] A. Neelakantan, Q. V. Le, and I. Sutskever. Neural programmer: Inducing latent programs with gradient descent. *arXiv preprint arXiv:1511.04834*, 2015.

[15] S. Reed and N. De Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

[16] G. Yang. Lie access neural turing machine. *arXiv preprint arXiv:1602.08671*, 2016.

[17] W. Zaremba, T. Mikolov, A. Joulin, and R. Fergus. Learning simple algorithms from examples. In *International Conference on Machine Learning*, pages 421–429, 2016.

[18] W. Zaremba and I. Sutskever. Reinforcement learning neural turing machines. *arXiv preprint arXiv:1505.00521*, 419, 2015.