NEU0

Karthik R,[*] **Aman Achpal**[*] **& Vinayshekhar BK**[*]
Department of Computer Science
PES Institute of Technology
Bangalore, 560085, India
{karthikradharishnan96,aman.achpal,vinayshekhar000}@gmail.com

Anantharaman Palacode Narayana Iyer
JNResearch
Bangalore, India
ananth@jnresearch.com

Channa Bankapur
Department of Computer Science
PES Institute of Technology
channabankapur@pes.edu

ABSTRACT

MU0 is a deterministic computer that can store data in memory, manipulate it using programs, enabling decision making. Neu0 is a neural computational core modeled around the same principles. We create an ensemble of Neural Networks capable of executing ARM code, and discuss generalizations of our framework. We showcase the advantage of our technique by correctly executing malformed instructions, and discuss efficient memory management techniques.

1 INTRODUCTION

Recent trends such as the work done by Neelakantan et al. (2015), Reed & de Freitas (2015), Bunel et al. (2016) explore the ability of MLPs to execute code. Each introduces an architectural trick, augmenting standard networks to allow for this functionality. Unlike previous works, we introduce a general neural framework to execute code with novel memory management techniques like efficient index based location addressing and caching. As noted byZaremba & Sutskever (2014), there are considerable challenges in training MLPs to execute high level languages. We train our system on the lower level ARM Instruction set; however, our system is language agnostic and can be readily extended to other assembly level languages. Sophisticated instructions can be composed from the primitives with which we augment our controller. Our architecture is a composition of components, each component being an indeterministic equivalent of the Von Neumann architecture. This allows for interchangeability of components, such as replacing our AU with the Neural GPU, Kaiser & Sutskever (2015). By supporting branch instructions, we allow for the execution of non-trivial code. Furthermore, by training with random noise, we make our system robust to mutilated instructions, allowing it to serve as a computational core to future applications, such as Machine Translation from algorithms/pseudo-code to code, that might not yield perfect outputs.

Our contributions include Neu0[1], a machine capable of executing ARM code. We argue for compositionality of components, as different components require different training procedures. Finally, since ARM machines allow for random access of external memory based on indices, we describe a novel and efficient method of accessing memory that exploits integer based indexing. Though not required for our system, we introduce the concept of cache memory, and explain how it could be used in a generalization of our system where content based addressing is used. To the best of our knowledge, this is the first ensemble of Neural Networks that use efficient index based location addressing and execute ARM code.

2 ARCHITECTURE

Figure 1 shows the high level architecture of our system.

[*]Equal Contribution
[1]Data and code will be made available at https://github.com/Neu0/neu0

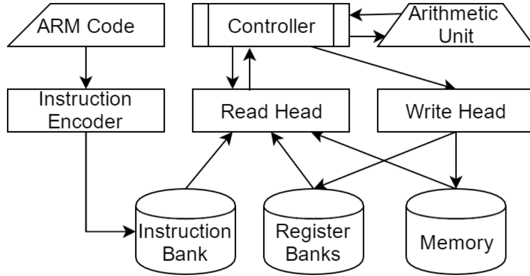


Figure 1: High Level Architecture of Neu0

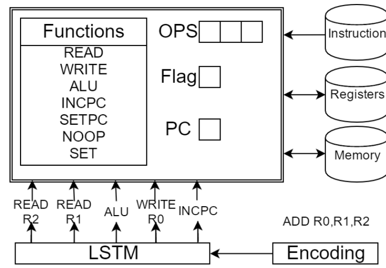


Figure 2: Controller of Neu0

2.1 Instruction Encoder

We transformed ARM instructions to vector embeddings by passing the text character by character to a deep many-to-one LSTM and storing the ultimate hidden layer activations.Zaremba & Sutskever (2014) found that LSTMs are unable to accurately perform decimal multiplication, as was found by us. Hence, following Bunel et al. (2016), the constants in the instruction were preloaded into the registers prior to execution. The forward pass steps are summarized below:

$$h_t = f_{LSTM}(c_t, h_{t-1})$$

$$inst = f_{inst}(h_T) \qquad r0 = f_{r0}(h_T) \qquad r1 = f_{r1}(h_T) \qquad r2 = f_{r2}(h_T) \qquad cond = f_{cond}(h_T)$$

where T, inst, r0, r1, r2, cond correspond to the final timestep of the LSTM, and the instruction, registers, and condition probability distributions respectively.

2.2 Controller

The controller is an LSTM that updates an environment representation. Figure 2 shows the architecture of the controller. The controller performs two main tasks:
Step Generator: Similar to Neural Programmer-Interpreters by Reed & de Freitas (2015), each higher level ARM instruction is broken down into a set of primitives. The controller outputs a list of instructions to be executed conditioned on the input instruction encoding. An example can be seen in Figure 2, where we see the set of primitives corresponding to an ADD instruction.
Updating Environment: The controller updates the program counter with the index of the next instruction encoding to be read, allowing for branch instructions to be executed. The environment also has temporary memory locations, which the Arithmetic Unit reads from, and writes results to. The controller syncs these temporary locations with the register bank prior to and post execution.

2.3 Register Bank

The reads and writes to the register bank happens as described in NTM by Graves et al. (2014). However, we do not use an erase vector as ARM always overwrites the destination register.

2.4 Arithmetic Unit

We support three arithmetic operations {ADD,SUB,MUL}. As Neural Networks cannot directly model multiplicative interactions of their inputs, we model multiplication as repeated addition. The Arithmetic unit consists of two components. The first, an interface to the controller which converts the input to a suitable form and retrieves the weighted result according to a confidence score, similar to the Neural Programmer by Neelakantan et al. (2015). The second component performs each operation. Gradient Descent performed unsatisfactorily, and model parameters were instead obtained using the Normal Equation. Figure 3 shows the architecture of the Arithmetic Unit.

$$\theta = (X^T X)^{-1} X^T y \qquad \text{and} \qquad Result = \textstyle\sum_{i \in AU} R_i \cdot C_i$$

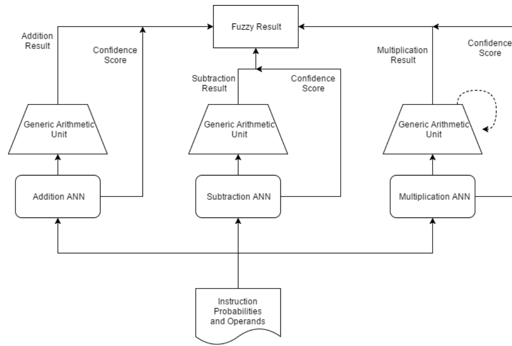


Figure 3: Architecture of arithmetic unit

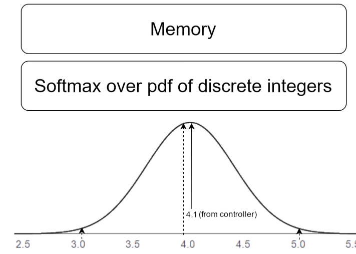


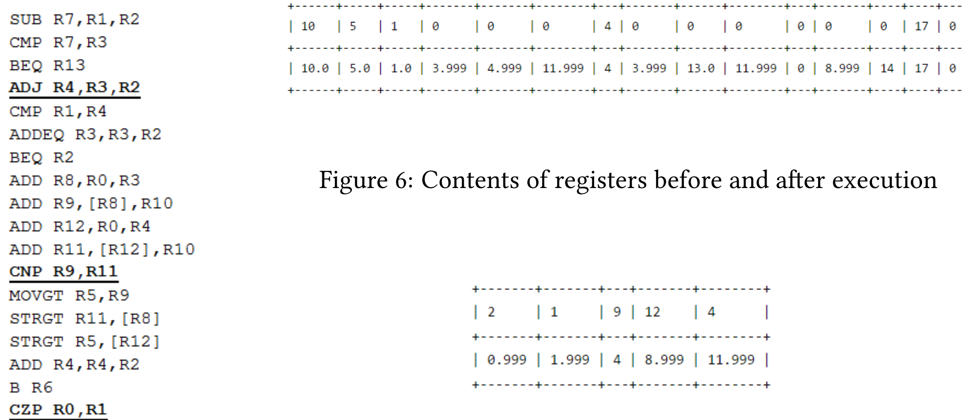
Figure 4: Schematic of Memory

2.5 Memory and Instruction Bank

In ARM, memory is accessed by indexing. Conventionally, location based addressing is a special case of content based addressing, as in DNTMs by Çaglar Gülçehre et al. (2016). With integers, we use squared Euclidean distance as the similarity measure. However, as Rae et al. (2016) noted, this scales poorly. We address this issue in two ways- The first involves augmenting the memory with a cache of size N: A memory controller accesses memory, with a softmax of size N+1, across the location in cache and 'N+1' if not in cache. A 'decay' vector is maintained allowing for LRU to be performed. Second- Since ARM deals with indices rather than m-dimensional address vectors, we exploit this to create a more efficient access mechanism. We plot a Gaussian with the index to be read as the mean, accounting for decimal indices due to prior weighted operations. We train an MLP to predict the standard deviation, finding the bounds at which the Gaussian tends to zero. The PDF is evaluated at integers within the bounds, and a softmax with temperature is evaluated over these to create an attention vector used for reads and writes. Generalizations might include learning a GMM coupled with Active Memory as described by Kaiser & Bengio (2016) allowing for concurrent memory accesses. Figure 4 gives an example of when 4.1 is the index to be read, as emitted by the controller. Controller updates PC with index of next instruction to be executed, and the instruction encoding is read using a Gaussian with mean as the contents of the PC

3 Results

Figure 5 demonstrates the robustness of our system in executing malformed ARM code. The contents of registers and relevant memory locations before and after execution of bubble sort are shown. Stack trace and more examples are available at `https://neu0.github.io`.



```
SUB R7,R1,R2
CMP R7,R3
BEQ R13
ADJ R4,R3,R2
CMP R1,R4
ADDEQ R3,R3,R2
BEQ R2
ADD R8,R0,R3
ADD R9,[R8],R10
ADD R12,R0,R4
ADD R11,[R12],R10
CNP R9,R11
MOVGT R5,R9
STRGT R11,[R8]
STRGT R5,[R12]
ADD R4,R4,R2
B R6
CZP R0,R1
```

Figure 5: Bubble Sort

10	5	1	0	0	0	4	0	0	0	0	0	0	17	0
10.0	5.0	1.0	3.999	4.999	11.999	4	3.999	13.0	11.999	0	8.999	14	17	0

Figure 6: Contents of registers before and after execution

2	1	9	12	4
0.999	1.999	4	8.999	11.999

Figure 7: Contents of relevant memory locations before and after execution

References

Rudy R. Bunel, Alban Desmaison, Pawan Kumar Mudigonda, Pushmeet Kohli, and Philip H. S. Torr. Adaptive neural compilation. In *NIPS*, 2016.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *CoRR*, abs/1410.5401, 2014.

Lukasz Kaiser and Samy Bengio. Can active memory replace attention? *CoRR*, abs/1610.08613, 2016.

Lukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *CoRR*, abs/1511.08228, 2015.

Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. *CoRR*, abs/1511.04834, 2015.

Jack W. Rae, Jonathan J. Hunt, Ivo Danihelka, Timothy Harley, Andrew W. Senior, Gregory Wayne, Alex Graves, and Tim Lillicrap. Scaling memory-augmented neural networks with sparse reads and writes. In *NIPS*, 2016.

Scott E. Reed and Nando de Freitas. Neural programmer-interpreters. *CoRR*, abs/1511.06279, 2015.

Wojciech Zaremba and Ilya Sutskever. Learning to execute. *CoRR*, abs/1410.4615, 2014.

Çaglar Gülçehre, A. P. Sarath Chandar, Kyunghyun Cho, and Yoshua Bengio. Dynamic neural turing machine with soft and hard addressing schemes. *CoRR*, abs/1607.00036, 2016.