# A Neural Forth Abstract Machine

**Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, Sebastian Riedel**
Department of Computer Science
University College London
London, UK
`{m.bosnjak, t.rocktaschel, j.narad, s.riedel}@cs.ucl.ac.uk`

## 1 Introduction

Recently, a renewed interest in neural models of computation has sparked a surge of invention in neural architectures that can learn to perform algorithms akin to traditional computers, using primitives such as memory access and stack manipulation (Graves et al., 2014; Joulin & Mikolov, 2015; Kaiser & Sutskever, 2016; Kurach et al., 2016; Reed & de Freitas, 2016). These architectures can be trained from data through standard gradient descent methods, enabling machines to learn complex algorithmic behavior. However, for many tasks training data is scarce. In these cases the programmer may know the rough structure of the program, or how to implement sub-routines that are likely necessary to solve the task. In these scenarios, the question remains how to best exploit this type of knowledge when learning algorithms.

In this work we present a Neural Forth Abstract Machine ($\partial 4$) which enables programmers to integrate their procedural knowledge with neural networks through the use of program *sketches* (Solar-Lezama et al., 2005), specified in a traditional programming language. The sketch defines one part of the neural network behaviour, while the missing part is learned from data. The core insight that enables this approach is the fact that programming languages can usually be formulated in terms of an abstract machine that executes the commands of the language. By implementing these machines as neural networks, and constraining parts of the networks to follow the sketched behaviour, we can learn neural programs consistent with our procedural prior knowledge and optimised with respect to the training data.

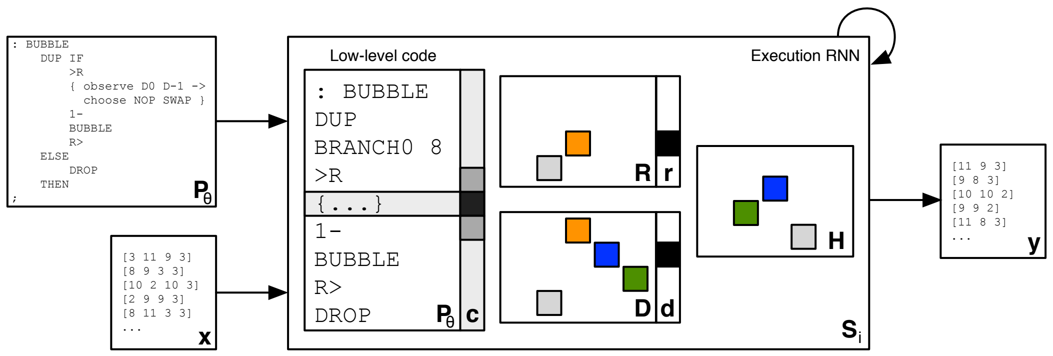
## 2 Neural Forth Abstract Machine

Forth is a simple but Turing-complete stack-based programming language (ANSI, 1994; Brodie, 1986) with a simple abstract machine. A standard Forth program defines a deterministic sequence of state transition functions on a Forth abstract machine.

A Forth sketch is an extension of a standard Forth program for which some behavior is undefined and learnable from data (Riedel et al., 2016). We refer to these open choices as *slots*. Slots accommodate for cases where a developer's procedural knowledge is incomplete, and may vary from simple element manipulation, command/function invocation, to a full neural network transition function which operates on the full machine state. $\partial 4$ is fully differentiable with respect to each time step transition as well as the distributed input representations in the continuous machine memories. This enables us to train it from input-output examples with gradient optimisation methods, and thus complete the missing part of the code.

We describe the three primary components of the $\partial 4$, as depicted in Figure 1: a differentiable machine state, differentiable Forth words, and the execution RNN.

**Differentiable Machine State** The symbolic machine state of Forth consists of a data stack $D$, return pointer stack $R$, the memory heap $H$ and the program counter $c$ indicating which command of the code $C$ is being executed. The differentiable machine state maps these elements into two



```
: BUBBLE
    DUP IF
        >R
        { observe D0 D-1 ->
          choose NOP SWAP }
        1-
        BUBBLE
        R>
    ELSE
        DROP
    THEN
;
```

```
[3 11 9 3]
[8 9 3 3]
[10 2 10 3]
[2 9 9 3]
[8 11 3 3]
...
```

```
: BUBBLE
DUP
BRANCH0 8
>R
{...}
1-
BUBBLE
R>
DROP
```

```
[11 9 3]
[9 8 3]
[10 10 2]
[9 9 2]
[11 8 3]
...
```

Figure 1: Neural Forth Abstract Machine. Forth code is translated to a low-level code, with slots (denoted with {...}) substituted by a parametrised neural network. The low-level code, together with the program counter $\mathbf{c}$, data stack $\mathbf{D}$, data stack pointer $\mathbf{d}$, return stack $\mathbf{R}$, return stack pointer $\mathbf{r}$, and the heap $\mathbf{H}$ are differentiable structures which comprise the machine state of the machine.

differentiable stacks (Das et al., 1992; Mozer & Das, 1993; Das et al., 1993) $\mathcal{D}$ and $\mathcal{R}$, a differentiable flat memory $\mathcal{H}$, and an attention vector $\mathbf{c}$ indicating which command of the sketch $\mathbf{P}$ is being executed at the current time step.

All three memory structures, data stack, return stack and the heap, are based on a differentiable flat memory – $\mathbf{D}, \mathbf{R}, \mathbf{H} \in \mathbb{R}^{l \times v}$ buffers of length $l$ and value width $v$, with well defined differentiable reading and writing procedures, similarly to the Neural Turing Machine memory (Graves et al., 2014). In addition to the flat memory buffer, the data stack and return stack keep separate pointers to their top-of-the-stack, $\mathbf{d}$ and $\mathbf{r}$. This allows for pushing and popping to be implemented in a straightforward manner, by writing into the buffer matrix and incrementing/decrementing top-of-the-stack pointers, respectively. Reading the top of the stack is achieved by simply multiplying the top-of-the-stack pointer and the appropriate memory buffer.

**Differentiable Forth words** Forth sketches merge two important ingredients: fixed procedural structure, and learnable functions – slots. Slots are in-place parametrised neural networks whose parameters are learned, whereas the fixed procedural structure gets translated to neural network modules, which depend on the used Forth words. In Forth a *word* is one of a set of pre-defined language primitives (e.g. DROP, to discard the top element of the stack, or DUP, to duplicate the top element of the stack) which acts as a function on the underlying discrete machine state. It is easy to convert Forth words to their differentiable counterparts (functions operating in the continuous space) by implementing their behavior via matrix operations. For example, the DROP word is simply realised by multiplying the top-of-the-stack pointer $\mathbf{d}$ with a circular permutation matrix $\mathbf{M}^{-}$ that circularly shifts the vector. We analogously define differentiable variants of other core Forth words, including control words for branching and looping.

**Execution RNN** Having defined the differentiable machine state and the differentiable Forth words, we can model the execution of a Forth program using an RNN which produces a state $\mathbf{S}_{i+1}$ conditioned on a previous state $\mathbf{S}_i$. This is achieved by passing the current state to each word $\mathbf{w}_i$ in the program, and weighting each of the produced states by its component of the program counter vector $\mathbf{c}_i$ that corresponds to program index $i$, effectively using $\mathbf{c}$ as an attention vector over code. Clearly this recursion, and its final state, are differentiable with respect to the program sketch $\mathbf{P}$, and its inputs. The final state of this RNN corresponds to the final state of a symbolic execution.

## 3 Preliminary experiments

We tested the $\partial 4$ on two problems: bubble sort and elementary-school addition. Both tasks are difficult for RNNs, which often fail to generalize to sequences marginally longer than the ones they have been trained on. The results of our experiments show that it is possible to learn to complete the sketches from a small number of training examples, and that the degree of procedural knowledge incorporated into the sketch has a measurable effect on learning (see Figure 2 for the performance on two different

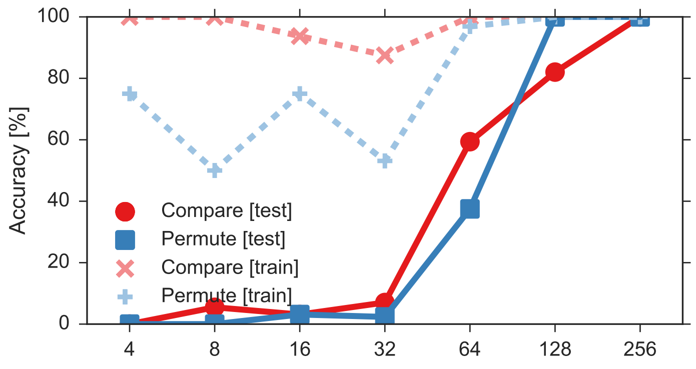


Figure 2: Train and test accuracy for varying number of training examples for bubble sort sketches based on comparison and permutation. A COMPARISON sketch compares two numbers and chooses whether to swap them. A PERMUTATION sketch permutes elements of the machine state, all based on the current elements of the machine state.

sketches). Since the resulting neural network is able to execute the sketch deterministically, the resulting sketch is able to generalise to all sequence lengths.

## References

ANSI. *Programming Languages - Forth*, 1994. American National Standard for Information Systems, ANSI X3.215-1994.

Leo Brodie. *Starting Forth*. Prentice-Hall, 1986.

Sreerupa Das, C Lee Giles, and Guo-Zheng Sun. Learning context-free grammars: Capabilities and limitations of a recurrent neural network with an external stack memory. In *Proceedings of The Fourteenth Annual Conference of Cognitive Science Society. Indiana University*, pp. 14, 1992.

Sreerupa Das, C Lee Giles, and Guo-Zheng Sun. Using prior knowledge in a {NNPDA} to learn context-free languages. *Advances in neural information processing systems*, pp. 65–65, 1993.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in Neural Information Processing Systems*, pp. 190–198, 2015.

Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *ICLR*, 2016.

Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. In *ICLR*, 2016.

Michael C Mozer and Sreerupa Das. A connectionist symbol manipulator that discovers the structure of context-free languages. *Advances in neural information processing systems*, pp. 863–863, 1993.

Scott Reed and Nando de Freitas. Neural programmer-interpreters. *ICLR*, 2016.

Sebastian Riedel, Matko Bošnjak, and Tim Rocktäschel. Programming with a differentiable forth interpreter. *arXiv preprint arXiv:1605.06640v1*, 2016.

Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. Programming by Sketching for Bit-streaming Programs. In *Proc. PLDI*, pp. 281–294, 2005.