Contents lists available at ScienceDirect

Neural Networks

journal homepage: www.elsevier.com/locate/neunet

A programmable neural virtual machine based on a fast store-erase learning rule

Garrett E. Katz[a,*], Gregory P. Davis[b], Rodolphe J. Gentili[c], James A. Reggia[b]

[a] Department of Elec. Engr. and Comp. Sci., Syracuse University, Syracuse, NY, USA
[b] Department of Computer Science, University of Maryland, College Park, MD, USA
[c] Department of Kinesiology, University of Maryland, College Park, MD, USA

ARTICLE INFO

Article history:
Received 20 January 2019
Received in revised form 22 June 2019
Accepted 21 July 2019
Available online 26 July 2019

Keywords:
Programmable neural networks
Local learning
Itinerant attractor dynamics
Multiplicative gating
Symbolic processing

ABSTRACT

We present a neural architecture that uses a novel local learning rule to represent and execute arbitrary, symbolic programs written in a conventional assembly-like language. This Neural Virtual Machine (NVM) is purely neurocomputational but supports all of the key functionality of a traditional computer architecture. Unlike other programmable neural networks, the NVM uses principles such as fast non-iterative local learning, distributed representation of information, program-independent circuitry, itinerant attractor dynamics, and multiplicative gating for both activity and plasticity. We present the NVM in detail, theoretically analyze its properties, and conduct empirical computer experiments that quantify its performance and demonstrate that it works effectively.

1. Introduction

Humans readily think, reason, and communicate in symbolic terms, whether through natural language, mathematical formalisms, or computer programs. Designing neural networks that can represent, execute, and learn these types of symbolic processes is a long-standing research topic in artificial intelligence. It has been known for some time that in theory, universal computation can be implemented in neural networks (Pollack, 1987; Siegelmann & Sontag, 1991). There also exist practical-purpose methods for "compiling" human-authored programs into neurocomputational representations (Dehaene & Changeux, 1997; Gruau, Ratajszczak, & Wiber, 1995; Neto, Siegelmann, & Costa, 2003; Siegelmann, 1994; Sylvester & Reggia, 2016). Most recently, neural architectures have been developed that can *learn* algorithmic behavior from training data (Bošnjak, Rocktäschel, Naradowsky, & Riedel, 2017; Bunel, Desmaison, Mudigonda, Kohli, & Torr, 2016; Devlin, Bunel, Singh, Hausknecht and Kohli, 2017; Graves et al., 2016; Neelakantan, Le, & Sutskever, 2016; Reed & De Freitas, 2016; Rocktäschel & Riedel, 2017; Sylvester & Reggia, 2016).

Most of these approaches involve one or both of the following paradigms:

- *Local representation*. For example, program stacks may be represented by arbitrary-precision values in single artificial neurons (Pollack, 1987; Siegelmann & Sontag, 1991); individual variables or symbols may be assigned individual artificial neurons (Abdelbar, Andrews, & Wunsch II, 2003); data structures may be represented by distributed patterns at a single time-step, rather than temporal sequences of patterns (Plate, 1995); different programs may require different program-specific circuitry (Dehaene & Changeux, 1997; Neto et al., 2003); or working memory is activation-based, wherein each item currently stored in memory must remain simultaneously active in separate neural populations (Graves et al., 2016). From a scientific standpoint, many brain mechanisms are thought to employ distributed, not local, representation of information, and from an engineering standpoint, local representation lacks fault tolerance and graceful degradation.
- *Non-local learning*. In particular, most recent work relies heavily on error back-propagation throughout the network, with unlimited repeated access to offline training examples. Further, many older works use hand-crafted program-specific circuitry and weights. Non-local learning is generally more computationally expensive and less biologically plausible than local learning, in which changes to a synaptic weight only depend on the recent activity of the neurons directly connected by that synapse.

There are some exceptions that break this mold, such as Sylvester and Reggia (2016). However, as yet these exceptions do not fully address universal computation: more work is needed to confirm their Turing completeness, and from a practical perspective,

* Corresponding author.
E-mail addresses: gkatz01@syr.edu (G.E. Katz), gpdavis@cs.umd.edu (G.P. Davis), rodolphe@umd.edu (R.J. Gentili), reggia@cs.umd.edu (J.A. Reggia).

https://doi.org/10.1016/j.neunet.2019.07.017
0893-6080/© 2019 Elsevier Ltd. All rights reserved.

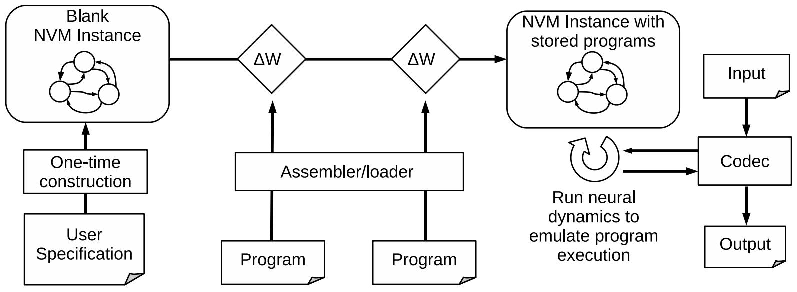


Fig. 1. The Neural Virtual Machine (NVM) workflow. First a blank NVM instance is constructed according to user-provided specifications (supplied as parameters to an API call). Thereafter, human-authored programs can be "assembled" and "loaded" into the instance via local learning rules. The instance can store new programs without forgetting previously learned programs, and execute existing programs at any time (only two programs are shown here for brevity). Program execution is emulated by running the underlying neural activation dynamics, which involves additional fast local weight updates. A "codec" converts between neural representations and human-readable input/output, as described further in the text.

they do not support many constructs and toolchains typically available in traditional programming languages (e.g., instruction operands, pointers, sub-routines, user-friendly "compilation" to neural encoding, etc.).

The main contribution of this paper is an approach to *universal neural programming* based on *non-local representation* and *local learning*, wherein synaptic weight changes depend only on information that is nearby in time and space. In this approach, working memory uses a novel local learning rule, and represents structured programs and data using temporal sequences of distributed neural activity patterns. We explain how our neural model can emulate a Harvard computer architecture (Rosen, 1969) that stores multiple arbitrary programs simultaneously. Harvard architectures use separate physical storage for programs and data and form the basis of many computing systems today. We also explain how our model can asymptotically simulate any Turing machine. Emulating a von Neumann architecture might also be possible with a similar approach, but we target the Harvard architecture because separate program and data memory is more readily implemented using separate neural layers, and avoids the complexity of coordinating program and data segments within a single memory.

We refer to our model as a "Neural Virtual Machine" (NVM). A reference implementation is open-source and freely available online.[1] The high-level NVM workflow is shown in Fig. 1. First, the NVM is initialized with a one-time non-local procedure, to meet user-provided specifications (layer sizes, activation functions, etc.), which are supplied via an API (Application Programming Interface) call. Subsequently, any human-authored program, written in an assembly-like language we designed, can be "assembled" and "executed" in the underlying neural network solely by virtue of local weight updates — with no rewiring of connectivity required. New programs can be added without erasing previously stored programs. An assembly-like language was chosen for greater simplicity and because assembly is a proven platform for any higher-level language feature.

Each human-readable symbol in a specific NVM program is represented by randomly chosen neural activity patterns. The symbol-to-pattern mappings are stored in an external non-neural lookup table that we call a "codec", borrowing the portmanteau of "encoder–decoder" from the field of video processing, since symbols are "encoded" by neural activity. During program execution, a human can use the codec to convert NVM input from, or NVM output to, human-readable symbols. However, the NVM itself is a purely neural system that can emulate programs from start to finish without relying on a codec.

[1] Source-code available at https://github.com/garrettkatz/nvm/releases/tag/v1.0.

The NVM is not intended as a veridical model of the brain, but rather a neuroengineering system that can emulate traditional computer programs using artificial neural computation. Our specific contributions include a novel local learning rule for emulating computer memory, the NVM architectural designs, some theoretical analysis that helps characterize our approach, and empirical validations using hand-written and randomly generated programs. We conclude with an assessment of the NVM and discussion of future research directions.

2. Implementation of the Neural Virtual Machine

We first explain the symbolic architecture being emulated, before explaining how that emulation is done in neural form.

2.1. Overview of the emulated harvard architecture

The NVM emulates a symbolic machine (SM) with a Harvard architecture, pictured in Fig. 2(a). It consists of several registers and memory modules. The SM processes symbols that we denote in teletype font (e.g. `a`, `b`, `c`, etc.). Each register $\mathtt{r}_i$ is a temporary storage location that contains one symbol at a time. The general purpose registers are accessible to the programmer, and a program can move symbols between registers or compare the symbols in two registers for equality. A special flags register called `co` ("compare output") contains the result of the most recent equality comparison. Programs can perform conditional jumps based on the results of these comparison operations.

In addition to temporary storage, some registers can also be dedicated to I/O communication with the external world. The contents of these registers can be altered asynchronously and exogenously by external processes outside the NVM. By moving symbols into these registers the NVM can potentially influence external processes as well. It is the programmer's responsibility to write "drivers" that properly handle sudden changes in I/O registers. From an implementation perspective, the NVM provides an API that allows other software to asynchronously set or get register contents.

Lastly, registers can interact with long-term contiguous memory storage, illustrated in more detail in Fig. 2(b). Memory is accessed by a single read/write head whose current address can be incremented or decremented. Register contents can be written to, or read from, the current head address. In addition, the current memory address can be saved in a register and then restored at a later time, akin to pointer reference and dereference, which enables a form of "random-access" memory. Heap memory is exposed to the programmer and can be used with the general purpose registers. Separate program and stack memories are used internally to store and execute programs.

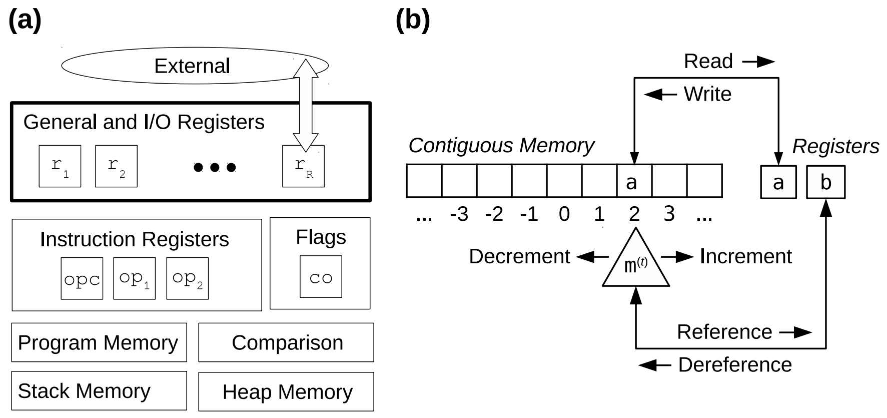


Fig. 2. (a) The symbolic machine architecture emulated by the NVM, consisting of various memory modules and registers described in the text. (b) A more detailed depiction of the contiguous memory model used by heap, stack, and program memory. Numbers $\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$ are memory addresses. a and b are symbols that can be stored in registers or memory. $\mathtt{m}^{(t)}$ is the current location of the read-write head at time t (triangle).

```
# Stream a list into memory and then retrieve list items in order
in:     ref rloc        # save starting memory location in rloc
loop1:  mov rval rinp   # get current item from input
        mem rval        # write current item to memory
        nxt             # advance read-write head to next memory location
        cmp rval nil    # compare current item with nil terminator
        jie out         # if rval equaled nil, jump out of loop 1
        mov rinp sep    # put separator in rinp for I/O protocol
        jmp loop1       # repeat on next item
out:    drf rloc        # restore starting memory address from rloc
loop2:  rem rval        # retrieve current item from memory
        mov rout rval   # send current item to output
        nxt             # advance read-write head to next memory location
        cmp rval nil    # compare current item with nil terminator
        jie done        # if rval equaled nil, jump out of loop 2
        mov rout sep    # put separator in rout for I/O protocol
        jmp loop2       # repeat on next item
done:   exit            # halt execution
```

Fig. 3. A small example NVM assembly program. '#' indicates a comment. This program is written for an NVM instance with four registers: `rloc`, `rval`, `rinp`, and `rout`. The latter two are I/O registers used to stream a list of symbols in from, or out to, the external environment. The program assumes that the environment obeys an I/O protocol in which successive list items are divided by a special separator symbol `sep`, and lists are terminated by a special symbol `nil`. This program streams in a list and saves it in memory, and then streams the full list back out in order. It uses `rloc` to save the memory location of the first list item, and `rval` to store the symbolic values of the list items as they are streamed in or out.

Programs for the SM are written in an imperative assembly-like language. The instructions of this language are listed in Table 1 and an example program is pictured in Fig. 3. Programs are sequences of (optionally labeled) instructions, one per line. Each instruction consists of an opcode and zero or more operands. Operands can be register names, labels, or literal symbolic values, depending on the opcode. Register names are provided as part of the user specification when an NVM instance is first constructed, so that they can be treated as "reserved words" and not arbitrary symbols. There are instructions for register movements and comparisons, conditional and unconditional jumps, sub-routine calls, and heap memory operations. A more detailed description of the NVM assembly language is in Appendix A.

Programs are stored in contiguous program memory, distinct from heap memory, and loaded by setting the program memory's read-write head to the address of the first instruction. Before executing an instruction, its opcode and operands are read into the dedicated registers opc, $\mathtt{op}_1$, and $\mathtt{op}_2$. After the instruction is executed, the program is normally advanced one line by incrementing the program memory read-write head. However, if a jump occurs, the read-write head is moved non-linearly by dereferencing the label of the jump's target line. In addition, if the instruction is a sub-routine call, the current position of the program memory read-write head is pushed onto stack memory, and subsequently popped off when the next return instruction is encountered.

2.2. Neural model

The NVM is a purely neural emulation of the SM described above. The NVM's design is inspired by the "gated regions and pathways" organizational principle (Sylvester & Reggia, 2016), as illustrated in Fig. 4. The figure shows three generic regions for the sake of example, but our actual architecture involves many

Table 1
The NVM instruction set.[a]

Syntax	Description
`nop`	Do nothing (no operation).
`mov` *dst src*	Move (copy) the value of *src* into register *dst*.
`jmp` *lab*	Jump to the line labeled *lab*.
`cmp` s_1 s_2	Compare the values of s_1 and s_2 for equality.
`jie` *lab*	Jump, if the most recent compare was equal, to the line labeled *lab*.
`sub` *lab*	Call the sub-routine starting on the line labeled *lab*.
`ret`	Return from the current sub-routine to the line where it was called.
`nxt`	Shift the read/write head to the next memory location ("increment").
`prv`	Shift the read/write head to the previous memory location ("decrement").
`mem` *reg*	Write the symbol in register *reg* to the current memory location.
`rem` *reg*	Read the symbol at the current memory location into register *reg*.
`ref` *reg*	Save the current read/write head location in register *reg* ("reference").
`drf` *reg*	Move the read-write head to the position saved in *reg* ("dereference").
`exit`	Halt execution.

[a]The operands *src*, *lab*, s_1, and s_2 can either be literal symbolic values, or register names. In the latter case, the symbol contained in the named register is used.

more regions as detailed in the following. The key idea is that some neural units (i.e., artificial neurons) double as multiplicative gates that modulate activity *and learning* in other regions and pathways. The use of multiplicative gating is not new (Greff, Srivastava, Koutník, Steunebrink, & Schmidhuber, 2017) or even necessary for emulating universal computation in a neural network (Siegelmann & Sontag, 1991). However, a key novelty in our approach is the use of gated *plasticity*: gates influence not only activity, but also local weight changes (i.e. learning). This enables our main contribution of using *local learning* to emulate symbolic computation.

Mathematically, the NVM state evolves over time according to:

$$\mathbf{v}^{\mathrm{q}}(t+1) = \sigma_{\mathrm{q}}\left(\underbrace{\big(1-d_{\mathrm{q}}(t)\big)}_{\text{decay}}\underbrace{\omega_{\mathrm{q}}\mathbf{v}^{\mathrm{q}}(t)}_{\text{saturation}} + \underbrace{\sum_{\mathrm{r}} s_{\mathrm{q,r}}(t)W^{\mathrm{q,r}}(t)\mathbf{v}^{\mathrm{r}}(t)}_{\text{synaptic input}}\right) \quad (1)$$

$$W^{\mathrm{q,r}}(t+1) = W^{\mathrm{q,r}}(t) + \underbrace{\ell_{\mathrm{q,r}}(t)\Delta W^{\mathrm{q,r}}(t)}_{\text{gated learning}} \quad (2)$$

where

- $\mathbf{v}^{\mathrm{q}}(t) \in \mathbb{R}^{N_{\mathrm{q}}}$ is a real-valued vector of neural activity at time-step t in a region q with N_{q} neural units,
- $\sigma_{\mathrm{q}} : \mathbb{R}^{N_{\mathrm{q}}} \to \mathbb{R}^{N_{\mathrm{q}}}$ is an element-wise sigmoidal activation function,
- $d_{\mathrm{q}}(t) \in \{0, 1\}$ is a "decay" gate at time-step t that can cause neural activity in region q to become zero,
- $\omega_{\mathrm{q}} \in \mathbb{R}$ is a scalar "saturation" self-weight, larger than 1, which (in the absence of decay and synaptic input) causes $\mathbf{v}^{\mathrm{q}}$ to approach a stable fixed point in its current orthant,
- $s_{\mathrm{q,r}}(t) \in \{0, 1\}$ is a "synaptic input" gate at time-step t that can prevent or allow neural activity to propagate over the pathway from region r to region q,[2]
- $W^{\mathrm{q,r}}(t) \in \mathbb{R}^{N_{\mathrm{q}}\times N_{\mathrm{r}}}$ is the synaptic weight matrix at time-step t parameterizing the pathway from region r to region q,
- $\ell_{\mathrm{q,r}}(t) \in \{0, 1\}$ is a "learning" gate at time-step t that can prevent or allow local weight changes within the pathway from region r to region q, and
- $\Delta W^{\mathrm{q,r}}(t) \in \mathbb{R}^{N_{\mathrm{q}}\times N_{\mathrm{r}}}$ is a matrix of synaptic weight changes at time-step t in the pathway from region r to region q, determined by a (potentially pathway-specific) local learning rule.

[2] Note that we adopt the indexing scheme *destination*, *source* to be consistent with the conventional notation for synaptic weight indices.

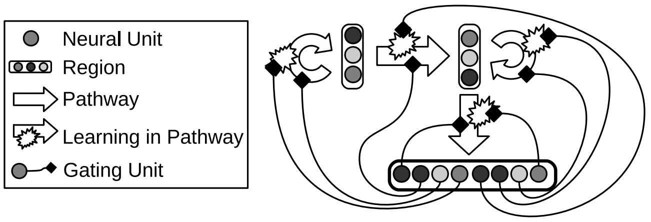


Fig. 4. A neural network with gated regions and pathways. Neural units (artificial neurons) in some regions (bold outline) can serve as gating units that modulate activity and learning in other pathways. Note that each block arrow represents a set of full connections (not individually shown); these are the connections whose activity flow and learning is gated.

For ease of presentation we omit explicitly listing bias terms that can be included along with the saturation self-weight and the synaptic input weights. In Eq. (1), the summation runs over every region r with synaptic connections to q. This can include the case q = r when the model includes a recurrent pathway from region q to itself, distinct from the scalar saturation self-weight ω_{q}.

Each gate is determined by a neural unit in a special "**g**ating **o**utput" region, which we abbreviate by "go" (see Fig. 5, explained below). This corresponds to the bold outlined region in the Fig. 4 example. The output of these go units determine the respective gate values. Mathematically, we equate $\mathbf{v}^{\mathrm{go}}$ with a concatenation of all the gates:

$$\mathbf{v}^{\mathrm{go}}(t) = [\ldots, d_{\mathrm{q}}(t), \ldots, s_{\mathrm{q,r}}(t), \ldots, \ell_{\mathrm{q,r}}(t), \ldots]^{\top} \quad (3)$$

where q and r range over all regions in the model. Computationally, at every time-step, the entries of $\mathbf{v}^{\mathrm{go}}$ are first accessed to populate the respective gate values. Then, $\mathbf{v}^{\mathrm{go}}$ evolves according to the same rule as every other region, by setting q = go in Eq. (1).

As illustrated in Fig. 1, when a blank NVM instance is first constructed, the user can specify the architectural details, such as a list of instance-specific region names (representing general-purpose and I/O registers), the region sizes N_{q}, the activation functions σ_{q}, and the local learning rules for $\Delta W^{\mathrm{q,r}}$. In this paper, we use hyperbolic tangent as the activation function for every region except $\mathbf{v}^{\mathrm{go}}$, which uses the Heaviside step function to produce binary gates in {0, 1}. When synaptic weight matrices are supplemented with bias vectors it is possible to use other activation functions such as logistic sigmoid. Various region sizes are also possible, although larger regions are generally required to emulate larger programs. As reported later, we systematically tested a wide range of region sizes to assess the practical scalability of the approach. Finally, various learning rules may be

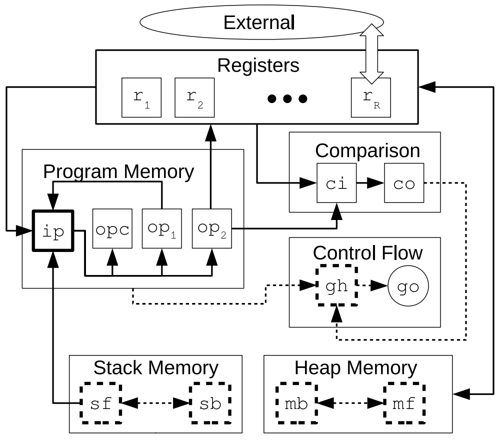


Fig. 5. The NVM network architecture. Small squares are neural regions. Dashed lines are pathways with fixed weights that are initialized when an NVM instance is constructed. Solid lines are pathways that undergo weight changes during program learning and execution. Recurrent self-to-self pathways are indicated by bold outlines around a region rather than an arrow. Each register region also has a pathway to every other register region (not shown). The circular go region gates all pathways in the architecture (not shown). Note that each arrow denotes an entire *pathway* between regions, which is a full set of synaptic connections (not individually shown).

possible, but our successful implementation relied on a specific novel learning rule that we detail in Section 2.4.3.

The full set of NVM regions and pathways is shown in Fig. 5, and explained in the following sections. There is roughly one neural region per register, with additional regions to implement memory, comparison, and control flow. We now explain how this architecture represents symbols and programs.

2.3. Representing individual symbols

One last user-configurable parameter is $\rho_{\mathtt{q}} \in (0,1)$, a per-region constant that specifies the steady-state magnitude of each entry in a neural activity vector. For a given region q, a symbol c is represented by a neural activity vector denoted $\mathbf{v}^{\mathtt{q}}[\mathtt{c}] \in \{-\rho_{\mathtt{q}}, +\rho_{\mathtt{q}}\}^{N_{\mathtt{q}}}$. Note that we use parentheses to denote an activity pattern at a certain time, but square brackets to denote a time-independent activity pattern representing a certain symbol. The pattern at some time t (i.e., $\mathbf{v}^{\mathtt{q}}(t)$) may or may not be the pattern for some symbol c (i.e., $\mathbf{v}^{\mathtt{q}}[\mathtt{c}]$). These mappings from symbols to patterns are stored in the NVM codec.

Once $\rho_{\mathtt{q}}$ is fixed, the saturation parameter $\omega_{\mathtt{q}}$ can be automatically chosen so that the saturating dynamics converge to vectors with $\pm\rho_{\mathtt{q}}$ entries. More formally, when synaptic input and decay are both gated off ($d_{\mathtt{q}}(t) = s_{\mathtt{q},\mathtt{r}}(t) = 0$), Eq. (1) reduces to $\mathbf{v}^* = \sigma(\omega_{\mathtt{q}}\mathbf{v}^*)$ at a fixed point $\mathbf{v}^* = \mathbf{v}^{\mathtt{q}}(t) = \mathbf{v}^{\mathtt{q}}(t+1)$. Setting each $|v_i|$ to $\rho_{\mathtt{q}}$ and isolating $\omega_{\mathtt{q}}$, one obtains the formula $\omega_{\mathtt{q}} \leftarrow \sigma^{-1}(\rho_{\mathtt{q}})/\rho_{\mathtt{q}}$.

As detailed below, most SM instructions are emulated with sparse gating patterns, which keep most regions in saturation mode most of the time. Consequently, NVM regions spend most of their time converging to fixed points, interspersed with occasional transitions when pathways are ungated and incoming synaptic input causes a transition to a different basin of attraction. As such, the time evolution of the NVM can be viewed as a form of itinerant attractor dynamics (Hoshino, Usuba, Kashimori, & Kambara, 1997; Sylvester & Reggia, 2016).

2.4. Emulating the symbolic machine with local learning rules

We next describe the local learning rules of the NVM and how they are used to implement the symbolic machine (SM) in Fig. 2. First we explain how several individual instructions are performed, presenting the local learning rules along the way. We then explain how *multiple* instructions are stored as a program in program memory. Finally we explain how programs are executed sequentially, accounting for non-linear jumps. As detailed later, non-linear program execution is accomplished by properly initializing the fixed weights in the control flow regions of Fig. 5, which implement the instruction cycle of the emulated SM.

2.4.1. Register moves

The simplest instruction is mov, which copies a symbol c from one register to another. The NVM can readily "copy" the underlying pattern from one region to another by ungating synaptic activity flow between them and decay in the target region ($d_{\mathtt{q}}(t) = s_{\mathtt{q},\mathtt{r}}(t) = 1$). Under this gating pattern, Eq. (1) reduces to $\mathbf{v}^{\mathtt{q}}(t+1) = \sigma_{\mathtt{q}}(W^{\mathtt{q},\mathtt{r}}(t)\mathbf{v}^{\mathtt{r}}(t))$, where q and r are the respective target and source registers. Therefore the NVM can perform the copy in a single time-step if the weight matrix satisfies $\mathbf{v}^{\mathtt{q}}[\mathtt{c}] = \sigma_{\mathtt{q}}(W^{\mathtt{q},\mathtt{r}}(t)\mathbf{v}^{\mathtt{r}}[\mathtt{c}])$. The requisite weights are produced by associative learning, applied to each register pair and each symbol in a program when it is assembled into the NVM. Later, when mov is encountered during execution, it can be emulated by simply ungating synaptic flow in the respective pathway. Note that in this scheme, the same symbol can be represented by different patterns in different regions. The associative learning rule is described in the next two sections.

2.4.2. Memory operations

Hetero-associative learning in the NVM is crucial not only for register moves but also for the six memory operations shown in Fig. 2(b). Each possible memory address ...,−2,−1,0,1,2, ...is also treated as a symbol and receives its own *address pattern* in dedicated regions. Associative learning can then be applied with those address patterns in three ways:

- Contiguity of linear address space can be encoded by associating each address pattern with its successor in a recurrent pathway (using temporally asymmetric learning).
- Register contents can be "written" to a memory address, by associating the address pattern with the content pattern, and forgetting any previously associated content pattern.
- Register contents can be used as "pointers" to a memory address, by associating the content pattern with the address pattern (using a pathway in the opposite direction from memory writes).

Note that the latter two items require fast learning *during* program execution, not before. Once these associations are properly learned, the remaining memory operations (increments/decrements, memory reads, and pointer dereferences) can be emulated by ungating activity flow, rather than associative learning, in the respective pathways. The specific NVM regions that store heap address patterns are mf ("memory forward") and mb ("memory backward"), which encode forward and backward motion of the read-write head, respectively (see Fig. 5). The pathways from each of these regions to themselves, to each other, and to the general purpose registers are where associative learning or activity flow occurs during memory operations. Each memory operation uses an appropriate gating pattern applied to these pathways, as illustrated in Fig. 6.

As a concrete example, suppose at time t the NVM memory contains the symbol b at address 1. This is represented by previously learned weights that satisfy $\mathbf{v}^{\mathtt{r}_i}[\mathtt{b}] = \sigma_{\mathtt{r}_i}(W^{\mathtt{r}_i,\mathtt{mf}}(t)\mathbf{v}^{\mathtt{mf}}[\mathtt{1}])$, where $\mathtt{r}_i$ is a register. Suppose further that the read-write head is currently positioned at address 1: i.e., $\mathbf{v}^{\mathtt{mf}}(t) = \mathbf{v}^{\mathtt{mf}}[\mathtt{1}]$. Lastly, suppose that a *different* symbol than b, namely c, is currently stored in the register: i.e., $\mathbf{v}^{\mathtt{r}_i}(t) = \mathbf{v}^{\mathtt{r}_i}[\mathtt{c}]$. A write operation would ungate learning to forget the previous association of $\mathtt{1}$ with $\mathtt{b}$, and form a new association with $\mathtt{c}$, producing new weights at the next time-step $t+1$ which satisfy $\mathbf{v}^{\mathtt{r}_i}[\mathtt{c}] = \sigma_{\mathtt{r}_i}(W^{\mathtt{r}_i,\mathtt{mf}}(t+1)\mathbf{v}^{\mathtt{mf}}[\mathtt{1}])$. A subsequent increment operation would ungate activity flow in the recurrent $\mathtt{mf}$ pathway, retrieving the previously learned temporally asymmetric association of $\mathtt{1}$ with $\mathtt{2}$: i.e.,

$$\begin{aligned}\mathbf{v}^{\mathtt{mf}}(t+2) &= \mathbf{v}^{\mathtt{mf}}[\mathtt{2}] = \sigma_{\mathtt{mf}}(W^{\mathtt{mf},\mathtt{mf}}(t+1)\mathbf{v}^{\mathtt{mf}}[\mathtt{1}]) \\ &= \sigma_{\mathtt{mf}}(W^{\mathtt{mf},\mathtt{mf}}(t+1)\mathbf{v}^{\mathtt{mf}}(t+1)).\end{aligned}$$

At the same time, the $\mathtt{mf} \to \mathtt{mb}$ pathway is also ungated, so that similar previously learned associations also advance $\mathtt{mb}$ to position $\mathtt{2}$ and keep it in "lock-step" with $\mathtt{mf}$. This way, a future decrement operation will decrement from the current head position $\mathtt{2}$ rather than the previous position $\mathtt{1}$. The reason for two regions $\mathtt{mf}$ and $\mathtt{mb}$, instead of one, is to provide distinct pathways (i.e. weight matrices) for representing the distinct directions through address space (forwards and backwards). If our model were augmented so that a single region could have multiple distinct recurrent weight matrices, then the notation would be more complicated, but one region for memory address patterns would suffice. A more formal description of all six memory operations and their neural implementation is provided in Appendix B.

2.4.3. A fast store-erase learning rule

Next, we present a local learning rule capable of forming the associations required above. In general terms, the rule should be able to associate a pattern $\mathbf{v}^{\mathtt{r}}$ in a source region $\mathtt{r}$ (e.g., a memory address in $\mathtt{mf}$) with a pattern $\mathbf{v}^{\mathtt{q}}$ in a target region $\mathtt{q}$ (e.g., a symbol in a register) *in a single time-step*. Moreover, if later a new target $\hat{\mathbf{v}}^{\mathtt{q}}$ is to be associated with the same source $\mathbf{v}^{\mathtt{r}}$ (e.g., overwriting the same memory address with a new symbol), the rule must readily forget the previous association when storing the new one.

To this end, we propose the following "Fast Store-Erase Learning Rule:"

$$\Delta W^{\mathtt{q},\mathtt{r}}(t) = \left(\underbrace{\sigma_{\mathtt{q}}^{-1}(\mathbf{v}^{\mathtt{q}}(t))}_{\text{store new target}} - \underbrace{W^{\mathtt{q},\mathtt{r}}(t)\mathbf{v}^{\mathtt{r}}(t)}_{\text{erase old target}}\right)\underbrace{(\mathbf{v}^{\mathtt{r}}(t))^{\top}}_{\text{source}}/\underbrace{(\rho_{\mathtt{q}}^2 N_{\mathtt{r}})}_{\text{normalize}}, \tag{4}$$

where $\sigma_{\mathtt{q}}$ is hyperbolic tangent, $N_{\mathtt{r}}$ is the size of region $\mathtt{r}$, and $\rho_{\mathtt{q}}$ is the magnitude of each neural unit's activity at a saturated fixed point (i.e., $\sigma_{\mathtt{q}}(\omega_{\mathtt{q}}\rho_{\mathtt{q}}) = \rho_{\mathtt{q}}$). Intuitively, this is basic Hebbian storage of a new association, accounting for the non-linear $\sigma_{\mathtt{q}}$, and with an additional "forget" term that erases the effect of a previously stored association. Under the right conditions, if $\mathbf{v}^{\mathtt{r}}(t)$ is a new pattern that has not been used previously and is being stored for the first time, then $W^{\mathtt{q},\mathtt{r}}(t)\mathbf{v}^{\mathtt{r}}(t) \approx 0$, so that the "erase" term has negligible effect and the rule reduces to normal Hebbian learning. These claims are substantiated in Sections 4 and 5, which provide several theoretical and empirical results characterizing this learning rule. Most importantly, in the limit $N_{\mathtt{r}} \to \infty$, this rule can approximate memory writes to an infinite memory tape. Other activation functions can also be accommodated with additional bias terms and constants in an analogous learning rule (omitted here for simplicity).

To our knowledge, this particular form of local learning has not been proposed before. We note that it encompasses aspects of both Hebbian and anti-Hebbian learning, and is very different from gradient-based learning that is most widely used in programmable neural nets today. In particular it is a one-step, online learning rule that does not require large amounts of training data. Clearly, Eq. (4) is local in time. Moreover, it has a form of spatial locality: changes to each row of $W^{\mathtt{q},\mathtt{r}}$ are independent of the other rows.

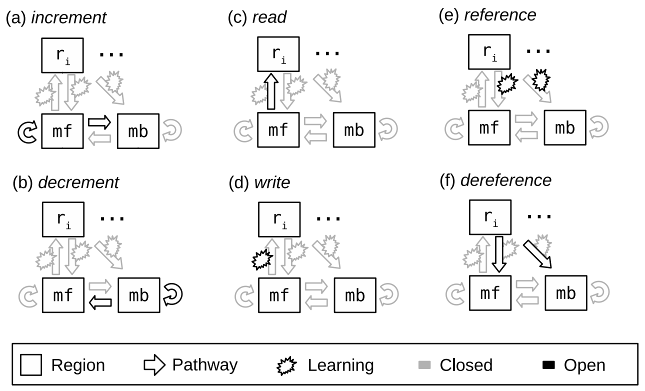


Fig. 6. Gated regions and pathways organization underlying contiguous memory emulation. Only relevant regions are shown. Pathways that are gated closed (i.e., no learning or activity) are shown in light gray, and open pathways are shown in black. Here, an "open" learning gate between $\mathtt{q}$ and $\mathtt{r}$ means that $\ell_{\mathtt{q},\mathtt{r}}(t) = 1$. "Open" activity flow means that $d_{\mathtt{q}}(t) = s_{\mathtt{q},\mathtt{r}}(t) = 1$: this causes any previous pattern in $\mathtt{q}$ to "decay" (i.e., be multiplied by $1 - d_{\mathtt{q}}(t) = 0$), and a new pattern to form on the basis of synaptic input from $\mathtt{r}$. Each sub-figure (a)–(f) shows all gates that are simultaneously open during a particular memory operation.

2.4.4. Emulating symbol comparison

Unlike memory operations and register copies, each of which require a *single* gating pattern, we implement comparison with a *sequence* of gating patterns over four time-steps, illustrated in Fig. 7. Symbol comparison is performed using two NVM regions $\mathtt{ci}$ ("compare input") and $\mathtt{co}$ ("compare output"). $\mathtt{co}$ uses a randomly chosen pattern to represent a $\mathtt{true}$ symbol, and its negation to represent $\mathtt{false}$: $\mathbf{v}^{\mathtt{co}}[\mathtt{true}] = -\mathbf{v}^{\mathtt{co}}[\mathtt{false}]$. Comparison operates on any two registers $\mathtt{r}_i$ and $\mathtt{r}_j$. First, the symbol in $\mathtt{r}_i$ (call it $\mathtt{c}$) is moved into $\mathtt{ci}$. Second, a special fast learning rule (Eqs. (5)–(6)) associates $\mathbf{v}^{\mathtt{ci}}[\mathtt{c}]$ with $\mathbf{v}^{\mathtt{co}}[\mathtt{true}]$. Third, the symbol in $\mathtt{r}_j$ is moved into $\mathtt{ci}$. Finally, $\mathtt{ci}$ is allowed to activate $\mathtt{co}$. If $\mathtt{r}_j$ also contained symbol $\mathtt{c}$, then the final step will produce $\mathtt{true}$ in $\mathtt{co}$, by virtue of the association that was just learned. On the other hand, if $\mathtt{r}_j$ had contained any other symbol, the learned weights should produce $\mathtt{false}$ in $\mathtt{co}$. Given that all symbol patterns have $\pm\rho_{\mathtt{ci}}$ entries, one can check that the following learning rule suffices:

$$W^{\mathtt{co},\mathtt{ci}}(t+1) = \sigma^{-1}(\mathbf{v}^{\mathtt{co}}[\mathtt{true}])(\mathbf{v}^{\mathtt{ci}}(t))^{\top}/\rho_{\mathtt{ci}}^2 \tag{5}$$

$$\mathbf{u}^{\mathtt{co},\mathtt{ci}}(t+1) = -\sigma^{-1}(\mathbf{v}^{\mathtt{co}}[\mathtt{true}])(N_{\mathtt{ci}} - 1) \tag{6}$$

where a bias vector $\mathbf{u}$ is required in this case, and the net synaptic input from $\mathtt{ci}$ into $\mathtt{co}$ (sans gating) is $W^{\mathtt{co},\mathtt{ci}}(t)\mathbf{v}^{\mathtt{ci}}(t)+\mathbf{u}^{\mathtt{co},\mathtt{ci}}(t)$ at any given time.[3] An analogous rule can be formed for activation functions other than hyperbolic tangent with additional constants (omitted here for brevity). This rule is only used in the $\mathtt{ci} \to \mathtt{co}$ pathway, and only when it is ungated at time $t+1$ of a comparison operation as shown in Fig. 7. Fast learning in all other pathways and during all other instructions uses the fast store-erase learning rule (4).

The gating sequence depicted in Fig. 7 is expressed more formally in Table 2. Here we use the notation $(\mathtt{q}, \mathtt{r})$ to abbreviate the gate pattern $d_{\mathtt{q}}(t) = s_{\mathtt{q},\mathtt{r}}(t) = 1$, which reduces Eq. (1) as shown in the table, thereby allowing activity to propagate to $\mathtt{q}$ from $\mathtt{r}$. Similarly, we use the notation $\{\mathtt{q}, \mathtt{r}\}$ to abbreviate the gate pattern $\ell_{\mathtt{q},\mathtt{r}}(t) = 1$, which allows learning to occur (in this case, according to Eqs. (5)–(6)).

[3] To be precise, after this learning rule is applied, a pattern other than $\mathtt{c}$ will produce a vector in $\mathtt{co}$ with the same signs as $\mathbf{v}^{\mathtt{co}}[\mathtt{false}]$ but magnitudes *at least* equal to ρ. This can be remedied by ungating saturation in $\mathtt{co}$, so that the magnitudes rapidly converge back to ρ.

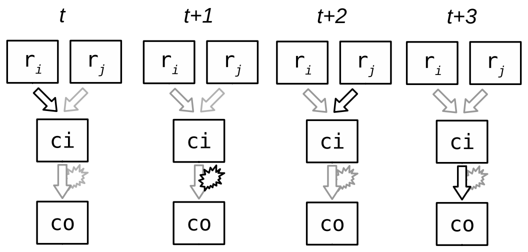


Fig. 7. Gated regions and pathways organization underlying comparison. Only relevant regions and pathways are shown, using the same conventions as Fig. 6. Comparison of two "register" regions $\mathtt{r}_i$ and $\mathtt{r}_j$ occurs over four time-steps from time t to time $t+3$. The gating pattern at each time-step is shown.

Table 2
Gating sequence for comparison.

Gate sequence	Reduced Eqs. (1)–(2)
0. $(\mathtt{ci}, \mathtt{r}_\mathtt{i})$	$\mathbf{v}^{\mathtt{ci}}(1) = \sigma_{\mathtt{ci}}(W^{\mathtt{ci},\mathtt{r}_\mathtt{i}(0)\mathbf{v}^{\mathtt{r}_\mathtt{i}}}(0))$
1. $\{\mathtt{co}, \mathtt{ci}\}$	$W^{\mathtt{co},\mathtt{ci}}(2) = W^{\mathtt{co},\mathtt{ci}}(1) + \Delta W^{\mathtt{co},\mathtt{ci}}(1)$
2. $(\mathtt{ci}, \mathtt{r}_\mathtt{j})$	$\mathbf{v}^{\mathtt{ci}}(3) = \sigma_{\mathtt{ci}}(W^{\mathtt{ci},\mathtt{r}_\mathtt{j}}(2)\mathbf{v}^{\mathtt{r}_\mathtt{j}}(2))$
3. $(\mathtt{co}, \mathtt{ci})$	$\mathbf{v}^{\mathtt{co}}(4) = \sigma_{\mathtt{co}}(W^{\mathtt{co},\mathtt{ci}}(3)\mathbf{v}^{\mathtt{ci}}(3) + \mathbf{u}^{\mathtt{co},\mathtt{ci}}(3))$

2.4.5. Representing and executing programs

Programs are stored in dedicated program memory that is similar to, but simpler than, heap memory. The current program counter is stored in a single region called `ip` ("instruction pointer"), analogous to `mf`, that can be incremented but not decremented. Just as heap memory contents can be read from `mf` into registers, program memory contents (i.e., instructions) can be read from `ip` into the regions `opc`, $\mathtt{op}_1$, and $\mathtt{op}_2$, which represent the opcode and operands.

Programs are initially "assembled" (see Fig. 1) by first writing the instructions to contiguous program memory addresses, using the fast store-erase learning rule. Next, memory reference operations, also using the fast store-erase learning rule, are used to associate each label that occurs in a jump operand with its corresponding address in program memory. Finally, to support comparisons and register moves, each program symbol is encoded in `ci` and in every register, and the pathways between them are updated to associate different encodings of the same pattern. A more formal statement of the assembly procedure is included in Appendix C.

Once a program has been assembled in program memory, it is loaded and executed by initializing `ip` to the address of its first line and then iterating the neural dynamics (Eqs. (1) and (2)). Each instruction cycle is driven by neural dynamics in the control flow regions, described next, that load the current instruction, execute it, and then update the instruction pointer in `ip`. For non-jump instructions, `ip` is updated with a simple increment memory operation that advances to the next position in program memory, which stores the next line of the program. For jump instructions, `ip` is updated by dereferencing the target line label supplied in the instruction operand region. Sub-routine calls are like jumps but also push the current program counter onto stack memory, with a stack head increment followed by a write. Subsequent return instructions then pop `ip` back off stack memory with a read followed by a decrement. These instructions are explained more formally in Appendix D.

The control flow regions are `go` (the gating output), and an auxiliary "gating hidden" region `gh` (see Fig. 5). Each instruction cycle begins with the same gating sequence in `go`, which reads the current instruction from program memory. However, depending on the instruction contents loaded into `opc`, $\mathtt{op}_1$, and $\mathtt{op}_2$, and the comparison flag in `co`, the `go` dynamics must "branch" down one of the several different gating sequences required for different instructions. Consequently, the gating dynamics can be conceptualized as a Finite State Machine (FSM). Formally, the gating FSM is specified by a tuple (S, A, G, τ, γ), where

- S is a set of states (each represented by a pattern in `gh`),
- A is an alphabet of FSM input tokens (each represented by a concatenation of patterns in `opc`, $\mathtt{op}_1$, $\mathtt{op}_2$, and `co`),
- G is an alphabet of FSM output tokens (each represented by a gating pattern in `go`),
- $\tau : S \times A \to S$ is the transition function (represented by recurrent and incoming pathways to `gh`), mapping a current state and input token $(s, a) \in S \times A$ to a new state $s' \in S$, and
- $\gamma : S \to G$ is an output function (represented by the $\mathtt{gh} \to \mathtt{go}$ pathway), mapping each state $s \in S$ to an output token $g \in G$.

Appendix D includes a detailed, concrete example of the gating FSM and its neural representation, in the case of the instruction cycle for a jump instruction.

The gating FSM can be used to generate "training data" for the weight matrices governing `gh` and `go` dynamics. In particular, each output mapping $s \mapsto \gamma(s)$ is a training example for $W^{\mathtt{go},\mathtt{gh}}$: training will seek the matrix satisfying $\forall s \in S : \gamma(s) = \sigma_{\mathtt{go}}(W^{\mathtt{go},\mathtt{gh}}s)$. Similarly, each transition $(s, a) \mapsto \tau(s, a)$ is a training example for the matrices $W^{\mathtt{gh},r}$, with source region r ranging over `opc`, $\mathtt{op}_1$, $\mathtt{op}_2$, and `co`. The FSM is an inherent property of the NVM assembly language and not program-specific, so the weights need only be calculated once when an NVM instance is constructed. A number of training techniques could be used to do this. In the spirit of fast learning, we opted for a non-iterative procedure (detailed in Appendix E) that assigns random patterns in `gh` for every FSM state, and then constructs suitable weights with a linear solver.

3. Turing completeness

In the limit of infinite computer memory, a typical computer is asympototically Turing complete. Similarly, in the limit $N \to \infty$, where N is the number of neural units (artificial neurons) in each NVM region, the NVM has infinitely many memory addresses and can simulate any Turing machine. We can show this by showing how to simulate any Turing machine with an NVM program.

To this end, we initialize an NVM instance with two registers: `rstt` stores a symbol representing the current Turing machine state, and `rsym` stores the tape symbol at the current tape position. The tape is represented by heap memory. The Turing machine state transitions are represented by a program in program memory. This program first reads the current tape symbol from heap memory into `rsym`. It then uses conditional jumps to form one logical branch for each possible (*state, tape symbol*) pair. Each branch emits the appropriate new tape symbol with a memory write, and then shifts the tape left or right with `nxt` or `prv` commands, as dictated by the given Turing machine. Appendix F includes an example program for a small Turing machine, as well as the general algorithm for converting a Turing machine into an NVM program that simulates it.

4. Theoretical analysis

The NVM depends critically on learning rule (4) functioning as expected. This learning rule updates the weights in a given pathway between a "source" region and a "target" region. In one time-step, it should successfully associate a pattern $\mathbf{x}$ in the source region with a pattern $\mathbf{y}$ in the target region. Moreover, the association of $\mathbf{x}$ with $\mathbf{y}$ should overwrite (erase) any previous associations of $\mathbf{x}$ with patterns other than $\mathbf{y}$. This section provides theoretical results characterizing the learning rule and the conditions in which it works as desired.

Formally, let $\mathbf{x}(1), \dots, \mathbf{x}(t), \dots$ and $\mathbf{y}(1), \dots, \mathbf{y}(t), \dots$ be two sequences of source and target patterns, respectively, each drawn from $\{-\rho, +\rho\}^N$. Region sub/super-scripts are omitted to avoid clutter. Suppose rule (4) is used to associate each $\mathbf{x}(t)$ with its corresponding $\mathbf{y}(t)$, one t at a time, and let $W(t)$ denote the resulting matrix after the first t associations have been learned, starting with a matrix $W(0)$ containing all zeros. To further simplify the notation, we use the following abbreviations: $\mathbf{x}_t$ for $\mathbf{x}(t)$, y_t for $\mathbf{y}_i(t)$, $\tilde{y}_t$ for $\sigma^{-1}(y_t)$, and $\mathbf{w}_t$ for the ith row vector of $W(t)$. All results and proofs are identical for each i, so it can be safely omitted. We also use η to abbreviate $\|\mathbf{x}_t\|^2 = N\rho^2$, as well as I to denote the identity matrix, and $\mathcal{P}_\mathbf{x} = I - \mathbf{x}\mathbf{x}^\top/\eta$ to denote the orthogonal projection matrix onto the null space of a vector $\mathbf{x}$. With this notation, the learning rule (4) can be rewritten more compactly as follows:

$$\mathbf{w}_t = \mathbf{w}_{t-1}\mathcal{P}_{\mathbf{x}_t} + \tilde{y}_t\mathbf{x}_t^\top/\eta, \tag{7}$$

with $\mathbf{w}_0$ containing all zeros in the base case.

If the same source pattern is present multiple times before some time T, the rule (7) should "erase" the old target associations and "store" the newest one. In other words, if $\mathbf{x}_t = \mathbf{x}_{t'}$ with $t \neq t'$, only the *last* (i.e., most recent) association for that source pattern should be retained when time T is reached. To denote that "last" association, we introduce a function $m_T(t)$, which returns the *largest* time $t' \leqslant T$ for which $\mathbf{x}_t = \mathbf{x}_{t'}$:

$$m_T(t) = \max_{\substack{t' \leqslant T \\ \mathbf{x}_t = \mathbf{x}_{t'}}} t' \tag{8}$$

$m_T(t)$ is well-defined for all $t \leqslant T$. As an example, if $\mathbf{x}_{t^*}$ is a pattern that only occurs once before time T, then $m_T(t^*) = t^*$. As another example, if $\mathbf{x}_T$ is a pattern that has not occurred previously before time T and is presented for the first time at time T, then $m_T(t) < T$ for every $t < T$. The latter example is relevant to the question of whether "erasing" a pattern that has not yet been stored has ill effect. Both of these examples are special cases subsumed by the following results, meaning that they are correctly accounted for by the learning rule under the right conditions.

It is known for other local learning rules that orthogonal activity patterns lead to superior memory capacity, and we first show an analogous result here:

Proposition 1. *Given any source and target sequences $\{\mathbf{x}_t\}_{t=1}^\infty$ and $\{\tilde{y}_t\}_{t=1}^\infty$, with each entry of each pattern equal to $\pm\rho$, suppose that $\mathbf{w}_t$ is learned according to Eq. (7), and that distinct patterns in the source sequence are mutually orthogonal: $\mathbf{x}_{t'}^\top\mathbf{x}_t \in \{0, \eta\}$ for all t, t'. Then for every T and every $t \leqslant T$,*

$$\mathbf{w}_T\mathbf{x}_t = \tilde{y}_{m_T(t)}. \tag{9}$$

In other words, in the orthogonal case, any source pattern (e.g., memory address) $\mathbf{x}_t$ is guaranteed to produce the most recent target associated with it (e.g., symbol stored at that address), namely $\mathbf{y}_{m_T(t)}$. This remains true *for arbitrarily large T* (e.g., unlimited memory writes).

Proof. We proceed by induction on T. In the base case, (7) gives $\mathbf{w}_1 = \tilde{y}_1\mathbf{x}_1^\top/\eta$. Since $\mathbf{x}_1^\top\mathbf{x}_1/\eta = \eta/\eta = 1$, this implies $\mathbf{w}_1\mathbf{x}_1 = \tilde{y}_1$. Since $m_1(1) = 1$, the proposition holds.

In the inductive case, (7) gives:

$$\mathbf{w}_T\mathbf{x}_t = \mathbf{w}_{T-1}\mathcal{P}_{\mathbf{x}_T}\mathbf{x}_t + \tilde{y}_T\mathbf{x}_T^\top\mathbf{x}_t/\eta. \tag{10}$$

There are two sub-cases: either $\mathbf{x}_T = \mathbf{x}_t$ (a repeated source pattern), or it does not. If it does, then $m_T(t) = T$, so $\tilde{y}_{m_T(t)} = \tilde{y}_T$. Furthermore,

$$\begin{aligned}\mathcal{P}_{\mathbf{x}_T}\mathbf{x}_t &= \mathcal{P}_{\mathbf{x}_T}\mathbf{x}_T = (I - \mathbf{x}_T\mathbf{x}_T^\top/\eta)\mathbf{x}_T = \mathbf{x}_T - \mathbf{x}_T\mathbf{x}_T^\top\mathbf{x}_T/\eta \\ &= \mathbf{x}_T - \mathbf{x}_T\eta/\eta = \mathbf{0},\end{aligned}$$

where $\mathbf{0}$ is an $N \times 1$ vector of all zeros (i.e., $\mathbf{x}_T$ is projected into its own null space). Substituting into Eq. (10) gives $\mathbf{w}_T\mathbf{x}_t = \tilde{y}_{m_T(t)}$ as needed.

In the other sub-case, $\mathbf{x}_T \neq \mathbf{x}_t$, which implies $t \leqslant T-1$ and $m_T(t) = m_{T-1}(t)$. Moreover, given that distinct source patterns are orthogonal, we have $\mathbf{x}_T^\top\mathbf{x}_t = 0$. This implies

$$\mathcal{P}_{\mathbf{x}_T}\mathbf{x}_t = (I - \mathbf{x}_T^\top\mathbf{x}_T/\eta)\mathbf{x}_t = \mathbf{x}_t$$

(since $\mathbf{x}_t$ is already in the null space of $\mathbf{x}_T$, it is unchanged by the projection). Therefore, Eq. (10) reduces to

$$\mathbf{w}_T\mathbf{x}_t = \mathbf{w}_{T-1}\mathbf{x}_t = \tilde{y}_{m_{T-1}(t)} = \tilde{y}_{m_T(t)}, \tag{11}$$

where the second equality follows from the inductive hypothesis and the third follows because $m_{T-1}(t) = m_T(t)$ in this sub-case.

We have shown that in either sub-case, the proposition holds. □

Even if the patterns are not strictly orthogonal, but chosen independently and uniformly at random from $\{-\rho, +\rho\}^N$, they are still orthogonal *in expectation*, since each pair of coordinates have the same sign or different sign with equal probability: $\mathbb{E}[\mathbf{x}_t^\top\mathbf{x}_{t'}] = 0$. This will allow us to show the following:

Theorem 1. *Suppose a source and target sequence, $\{\mathbf{x}_t\}_{t=1}^\infty$ and $\{\tilde{y}_t\}_{t=1}^\infty$, are generated by sampling each pattern of each sequence independently and uniformly from $\{-\rho, +\rho\}^N$, and suppose that $\mathbf{w}_t$ is learned according to Eq. (7). Then for every T and every $t \leqslant T$,*

$$\mathbb{E}[\tilde{y}_{m_T(t)}\mathbf{w}_T\mathbf{x}_t] > 0. \tag{12}$$

In other words, $\sigma(W(T)\mathbf{x}(t))$ is expected to have the same sign in every coordinate as its most recently associated target $\mathbf{y}(m_T(t))$, for arbitrarily large T (e.g., unlimited memory writes). As long as the signs are correct, saturation dynamics can then be used to fully recover $\mathbf{y}$. The proof of Theorem 1 follows.

Proof. From (7), we have

$$\mathbb{E}[\tilde{y}_{m_T(t)}\mathbf{w}_T\mathbf{x}_t] = \mathbb{E}[\tilde{y}_{m_T(t)}(\mathbf{w}_{T-1}\mathcal{P}_{\mathbf{x}_T} + \tilde{y}_T\mathbf{x}_T^\top/\eta)\mathbf{x}_t]. \tag{13}$$

By linearity of expectation, (13) implies

$$\mathbb{E}[\tilde{y}_{m_T(t)}\mathbf{w}_T\mathbf{x}_t] = \mathbb{E}[\tilde{y}_{m_T(t)}\mathbf{w}_{T-1}\mathcal{P}_{\mathbf{x}_T}\mathbf{x}_t] + \mathbb{E}[\tilde{y}_{m_T(t)}\tilde{y}_T\mathbf{x}_T^\top\mathbf{x}_t]/\eta. \tag{14}$$

First we consider the case $t = T$, wherein $\mathbf{x}_T = \mathbf{x}_t$ and $m_T(t) = T$. In this case, $\mathcal{P}_{\mathbf{x}_T}\mathbf{x}_t = \mathcal{P}_{\mathbf{x}_T}\mathbf{x}_T = \mathbf{0}$, and $\mathbf{x}_T^\top\mathbf{x}_t = \mathbf{x}_T^\top\mathbf{x}_T = \eta$. Therefore the right hand side of (14) simplifies to $\mathbb{E}[\tilde{y}_T^2] = \tilde{\rho}^2 > 0$.

The more complicated case is $t < T$. Here, we first note that while $m_T(t)$ is a random variable dependent on both $\mathbf{x}_T$ and $\mathbf{x}_t$, the quantity $\tilde{y}_{m_T(t)}$ is *independent* of them both, since every entry of every target pattern is chosen from $\pm\rho$ independently of every source pattern. Therefore $\mathbb{E}[\tilde{y}_{m_T(t)}\tilde{y}_T\mathbf{x}_T^\top\mathbf{x}_t] = \mathbb{E}[\tilde{y}_{m_T(t)}\tilde{y}_T]\mathbb{E}[\mathbf{x}_T^\top\mathbf{x}_t]$. Moreover, $\mathbb{E}[\mathbf{x}_T^\top\mathbf{x}_t] = 0$, since the entries of $\mathbf{x}_T$ and $\mathbf{x}_t$ are also independently chosen from $\pm\rho$, and their entry-wise products are equally likely to be $\pm(\rho^2)$, which averages to 0. Consequently, (14) simplifies to

$$\mathbb{E}[\tilde{y}_{m_T(t)}\mathbf{w}_T\mathbf{x}_t] = \mathbb{E}[\tilde{y}_{m_T(t)}\mathbf{w}_{T-1}\mathcal{P}_{\mathbf{x}_T}\mathbf{x}_t]. \tag{15}$$

We can condition the expectation in (15) on whether or not $\mathbf{x}_T = \mathbf{x}_t$, obtaining

$$\mathbb{E}[\tilde{y}_{m_T(t)}\mathbf{w}_T\mathbf{x}_t] = \mathbb{E}[\tilde{y}_{m_T(t)}\mathbf{w}_{T-1}\mathcal{P}_{\mathbf{x}_T}\mathbf{x}_t \mid \mathbf{x}_T \neq \mathbf{x}_t]\Pr\{\mathbf{x}_T \neq \mathbf{x}_t\} + \tag{16}$$

$$\mathbb{E}[\tilde{y}_{m_T(t)}\mathbf{w}_{T-1}\mathcal{P}_{\mathbf{x}_T}\mathbf{x}_t \,\big|\, \mathbf{x}_T = \mathbf{x}_t]\Pr\{\mathbf{x}_T = \mathbf{x}_t\} \tag{17}$$

Again, since $\mathcal{P}_{\mathbf{x}_T}\mathbf{x}_T = \mathbf{0}$, term (17) vanishes and we are left with the term on line (16). Moreover, the condition $\mathbf{x}_T \neq \mathbf{x}_t$ implies $m_T(t) = m_{T-1}(t)$. Therefore, substituting into (16) and using the definition of conditional expectation gives:

$$\mathbb{E}[\tilde{y}_{m_T(t)}\mathbf{w}_T\mathbf{x}_t] = \tag{18}$$

$$\mathbb{E}[\tilde{y}_{m_{T-1}(t)}\mathbf{w}_{T-1}\mathcal{P}_{\mathbf{x}_T}\mathbf{x}_t \,\big|\, \mathbf{x}_T \neq \mathbf{x}_t]\Pr\{\mathbf{x}_T \neq \mathbf{x}_t\} = \tag{19}$$

$$\sum_{\underline{y},\underline{\mathbf{w}},\underline{\mathbf{x}},\hat{\underline{\mathbf{x}}}} \underline{y}\,\underline{\mathbf{w}}\mathcal{P}_{\underline{\mathbf{x}}}\hat{\underline{\mathbf{x}}}\Pr\{\tilde{y}_{m_{T-1}(t)} = \underline{y},\ \mathbf{w}_{T-1} = \underline{\mathbf{w}},\ \mathbf{x}_t = \hat{\underline{\mathbf{x}}},$$

$$\mathbf{x}_T = \underline{\mathbf{x}},\ \mathbf{x}_T \neq \mathbf{x}_t\}, \tag{20}$$

where the underlined terms denote possible values for the respective random variables, and the summation ranges over all possibilities for those values. Note that the factor $\Pr\{\mathbf{x}_T \neq \mathbf{x}_t\}$ in (19) cancels the denominator of the conditional probability, leaving the joint probability in (20).

The events

$$\{\tilde{y}_{m_{T-1}(t)} = \underline{y},\ \mathbf{w}_{T-1} = \underline{\mathbf{w}},\ \mathbf{x}_t = \hat{\underline{\mathbf{x}}},\ \mathbf{x}_T = \underline{\mathbf{x}},\ \mathbf{x}_T \neq \mathbf{x}_t\}$$
$$\{\tilde{y}_{m_{T-1}(t)} = \underline{y},\ \mathbf{w}_{T-1} = \underline{\mathbf{w}},\ \mathbf{x}_t = \hat{\underline{\mathbf{x}}},\ \mathbf{x}_T = \underline{\mathbf{x}},\ \mathbf{x}_T \neq \hat{\underline{\mathbf{x}}}\}$$

(which differ only in the last term) are logically equivalent, and $\mathbf{x}_T$ is independent from the other random variables, so

$$\Pr\{\tilde{y}_{m_{T-1}(t)} = \underline{y},\ \mathbf{w}_{T-1} = \underline{\mathbf{w}},\ \mathbf{x}_t = \hat{\underline{\mathbf{x}}},\ \mathbf{x}_T = \underline{\mathbf{x}},\ \mathbf{x}_T \neq \mathbf{x}_t\} =$$
$$\Pr\{\tilde{y}_{m_{T-1}(t)} = \underline{y},\ \mathbf{w}_{T-1} = \underline{\mathbf{w}},\ \mathbf{x}_t = \hat{\underline{\mathbf{x}}}\}\Pr\{\mathbf{x}_T = \underline{\mathbf{x}},\ \mathbf{x}_T \neq \hat{\underline{\mathbf{x}}}\},$$

and since $\mathbf{x}_T$ is chosen uniformly at random,

$$\Pr\{\mathbf{x}_T = \underline{\mathbf{x}},\ \mathbf{x}_T \neq \hat{\underline{\mathbf{x}}}\} = \begin{cases} 2^{-N} & \text{if } \underline{\mathbf{x}} \neq \hat{\underline{\mathbf{x}}}, \\ 0 & \text{otherwise.} \end{cases} \tag{21}$$

Therefore (20) can be factored as follows:

$$\mathbb{E}[\tilde{y}_{m_T(t)}\mathbf{w}_T\mathbf{x}_t] \tag{22}$$

$$= \sum_{\underline{y},\underline{\mathbf{w}},\hat{\underline{\mathbf{x}}}} \underline{y}\,\underline{\mathbf{w}}\left[2^{-N}\sum_{\underline{\mathbf{x}}\neq\hat{\underline{\mathbf{x}}}}\mathcal{P}_{\underline{\mathbf{x}}}\right]\hat{\underline{\mathbf{x}}}\Pr\{\tilde{y}_{m_{T-1}(t)} = \underline{y},\ \mathbf{w}_{T-1} = \underline{\mathbf{w}},\ \mathbf{x}_t = \hat{\underline{\mathbf{x}}}\} \tag{23}$$

$$= 2^{-N}\sum_{\underline{y},\underline{\mathbf{w}},\hat{\underline{\mathbf{x}}}} \underline{y}\,\underline{\mathbf{w}}\left[\left(\sum_{\underline{\mathbf{x}}}\mathcal{P}_{\underline{\mathbf{x}}}\right) - \mathcal{P}_{\hat{\underline{\mathbf{x}}}}\right]\hat{\underline{\mathbf{x}}}\Pr\{\tilde{y}_{m_{T-1}(t)} = \underline{y},$$
$$\mathbf{w}_{T-1} = \underline{\mathbf{w}},\ \mathbf{x}_t = \hat{\underline{\mathbf{x}}}\} \tag{24}$$

$$= 2^{-N}\sum_{\underline{y},\underline{\mathbf{w}},\hat{\underline{\mathbf{x}}}} \underline{y}\,\underline{\mathbf{w}}\left(\sum_{\underline{\mathbf{x}}}\mathcal{P}_{\underline{\mathbf{x}}}\right)\hat{\underline{\mathbf{x}}}\Pr\{\tilde{y}_{m_{T-1}(t)} = \underline{y},\ \mathbf{w}_{T-1} = \underline{\mathbf{w}},\ \mathbf{x}_t = \hat{\underline{\mathbf{x}}}\}, \tag{25}$$

where (25) follows since $\mathcal{P}_{\hat{\underline{\mathbf{x}}}}\hat{\underline{\mathbf{x}}} = \mathbf{0}$. Since $\mathcal{P}_{\underline{\mathbf{x}}} = I - \underline{\mathbf{x}}\,\underline{\mathbf{x}}^\top/\eta$, we can view a sum of outer products $\sum_{\underline{\mathbf{x}}}\underline{\mathbf{x}}\,\underline{\mathbf{x}}^\top$ as a matrix multiplication, and obtain

$$\sum_{\underline{\mathbf{x}}}\mathcal{P}_{\underline{\mathbf{x}}} = 2^N I - XX^\top/\eta, \tag{26}$$

where X is an $N \times 2^N$ matrix with one column for every possible $\underline{\mathbf{x}}$. In fact, $XX^\top = 2^N\rho^2 I$, which can readily be seen from low-dimensional examples like

$$X = \rho\begin{bmatrix} +1 & -1 & +1 & -1 & +1 & -1 & +1 & -1 \\ +1 & +1 & -1 & -1 & +1 & +1 & -1 & -1 \\ +1 & +1 & +1 & +1 & -1 & -1 & -1 & -1 \end{bmatrix}. \tag{27}$$

Therefore, $\sum_{\underline{\mathbf{x}}}\mathcal{P}_{\underline{\mathbf{x}}} = 2^N(1-\rho^2/\eta)I = 2^N(1-1/N)I$, and (25) reduces to

$$\mathbb{E}[\tilde{y}_{m_T(t)}\mathbf{w}_T\mathbf{x}_t] \tag{28}$$

$$= (1-1/N)\sum_{\underline{y},\underline{\mathbf{w}},\hat{\underline{\mathbf{x}}}} \underline{y}\,\underline{\mathbf{w}}\hat{\underline{\mathbf{x}}}\Pr\{\tilde{y}_{m_{T-1}(t)} = \underline{y},\ \mathbf{w}_{T-1} = \underline{\mathbf{w}},\ \mathbf{x}_t = \hat{\underline{\mathbf{x}}}\} \tag{29}$$

$$= (1-1/N)\mathbb{E}[\tilde{y}_{m_{T-1}(t)}\mathbf{w}_{T-1}\mathbf{x}_t] \tag{30}$$

In summary, we have shown that $\mathbb{E}[\tilde{y}_{m_T(t)}\mathbf{w}_T\mathbf{x}_t] = (1-1/N)\mathbb{E}[\tilde{y}_{m_{T-1}(t)}\mathbf{w}_{T-1}\mathbf{x}_t]$ for all $t < T$. Expanding this recursion from the base case $\mathbb{E}[\tilde{y}_1\mathbf{w}_1\mathbf{x}_1] = \tilde{\rho}^2$ gives

$$\mathbb{E}[\tilde{y}_{m_T(t)}\mathbf{w}_T\mathbf{x}_t] = (1-1/N)^T\tilde{\rho}^2 > 0. \tag{31}$$

In both cases $t < T$ and $t = T$, the theorem holds. □

Theorem 1 bounds the expected value above zero for all T. However, the expected value approaches zero as T increases, and moreover, the variance is not characterized. Consequently, memory corruption could still be common in practice. It is known for classical attractor networks that memory performance is best when the number of patterns stored is significantly less than the number of neurons in the network. In our context, this corresponds to a pool of distinct source patterns $\mathcal{X} = \{\mathbf{x}^1, \ldots, \mathbf{x}^P\}$, with P small relative to N, and a sequence of $\mathbf{x}_t$ sampled from $\mathcal{X}$ with replacement. Concretely, for example, we might expect that as long as N is significantly larger than the number of memory addresses required for NVM programs in a given application, those addresses can be successfully rewritten an unlimited number of times. More formally, we conjecture the following analog of Theorem 1:

Conjecture 1. *Let $\mathcal{X} = \{\mathbf{x}^1, \ldots, \mathbf{x}^P\}$, with each $\mathbf{x}^p$ sampled independently and uniformly at random from $\{-\rho, +\rho\}^N$. Suppose the sequential source patterns $\mathbf{x}_t$ are each sampled independently and uniformly with replacement from $\mathcal{X}$. Then for every T and every $t \leqslant T$,*

$$\mathbb{E}[\tilde{y}_{m_T(t)}\mathbf{w}_T\mathbf{x}_t] > K(P, N, \rho) > 0, \tag{32}$$

where $K(P, N, \rho)$ is a value independent of T that increases as P/N decreases.

While a proof of this conjecture has eluded us, it is corroborated by empirical results in the following section.

5. Empirical results

Here we use computer experiments to bolster the results of our theoretical analysis above. We also empirically validate the NVM as a whole on hand-written and automatically generated programs.

5.1. Capacity of the learning rule

A famous result characterizing auto-associative attractor networks is that $P/N \approx .138$ in the large N limit, where N is the number of units and P is the number of patterns that can be reliably stored without an orthogonality constraint (Amit, Gutfreund, & Sompolinsky, 1985). An analogous result in our case considers both P/N and T/N, where as before T is the number of times the learning rule is applied. To empirically investigate storage capacity of learning rule (4), we used a randomized computer experiment in which we systematically varied P, T, and N in each trial, focusing on an isolated associative weight matrix without considering the NVM as a whole. For the purposes of this experiment only, we set $\rho = 1$, $\sigma(\cdot) = \text{sign}(\cdot)$, and omitted σ^{-1} from the learning rule. This was intended to more closely match the classical results and can be viewed as a high-gain limiting case where continuous activity patterns become binary vectors with ± 1 entries. In each trial, we sampled the random patterns without enforcing orthogonality, learned the weight matrix $W(T)$,

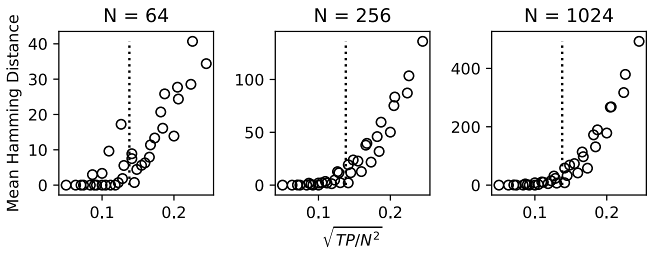


Fig. 8. Memory capacity for different N, P, and T. Dashed vertical lines indicate the classical proportion $\approx .138$.

and measured the following Hamming distance between target and actual output:

$$\sum_{i=1}^{N} \mathbf{1}[\text{sign}(\mathbf{y}_i(m_T(t))) = \text{sign}(W_{i,:}(T)\mathbf{x}(t))] \tag{33}$$

for each $t \leqslant T$. Here $\mathbf{1}[\cdot]$ is an indicator function, which takes the value 1 when the bracketed expression is true, and 0 otherwise. A Hamming distance of 0 is optimal, indicating that the sign of every vector element of the desired association was correctly learned (and therefore the desired association could be fully recovered via saturation dynamics).

The average Hamming distance in each trial (over 30 repetitions) is aggregated and plotted against $\sqrt{(T/N)(P/N)}$ in Fig. 8, for three orders of magnitude of N. The motivation for the formula $\sqrt{(T/N)(P/N)}$ is that when $T = P$, it reduces to the classical P/N, and when $T \neq P$, we might expect that increases in one require decreases in the other to maintain the same performance. Similar to the classical result, we observe a phase transition in reliable storage around $\sqrt{(T/N)(P/N)} \approx .138$, which becomes more pronounced as N increases. However, even for $N = 1024$ there are still some non-negligible errors near .138, including on the ordered side of the phase transition. In practice, if orthogonal patterns cannot be used, and if P and T can be estimated for a program, then conservatively choosing N so that the ratio is in the range $.05 - .10$ appears to be a more reliable rule of thumb.

We ran further experiments to disentangle the effects of increasing P versus T. In these experiments, we used a similar setup as before, but viewed performance as a function of T, for various fixed values of P. In each random trial, after training to time T, we measured the value of $\sigma^{-1}(y_i(m_T(t)))W_{i,:}(T)\mathbf{x}(t)$ for each $i \in \{1, \dots, N\}$ and $t \in \{1, \dots, T\}$. The empirical mean of this value was then aggregated across all i, all t, and across 30 random trials with the same N, P, and T. A large positive value for this mean indicates that $W_{i,:}(T)\mathbf{x}(t)$ is typically bounded away from zero and the same sign as $\sigma^{-1}(y_i(m_T(t)))$, as desired for an effective learning rule and suggested in Conjecture 1. As shown in Fig. 9 (top), the empirical mean appears to converge to a positive number, substantially far from 0, that depends on N and P but not T, suggesting that memory performance is relatively insensitive to the number of re-writes. This corroborates our theoretical results.

If the growth of the weights is also bounded independently of T, that would be another useful property both in theory and practice. To quantify the typical growth in the weights, we also show empirical maximums of $\max_{i,j} |W_{i,j}(T)|$ in Fig. 9 (bottom). This evidence suggests that $|W_{i,j}(T)|$ is indeed bounded for all T, at least with high probability. Finally, to further understand the distribution of possible weights, we automated an exhaustive brute-force computation of every possible $\mathbf{w}_T = W_{i,:}(T)$, by enumerating each possible training sequence and then iterating the fast store-erase rule on that sequence. This is feasible for small N and T, and Fig. 10 (left) plots every possible weight

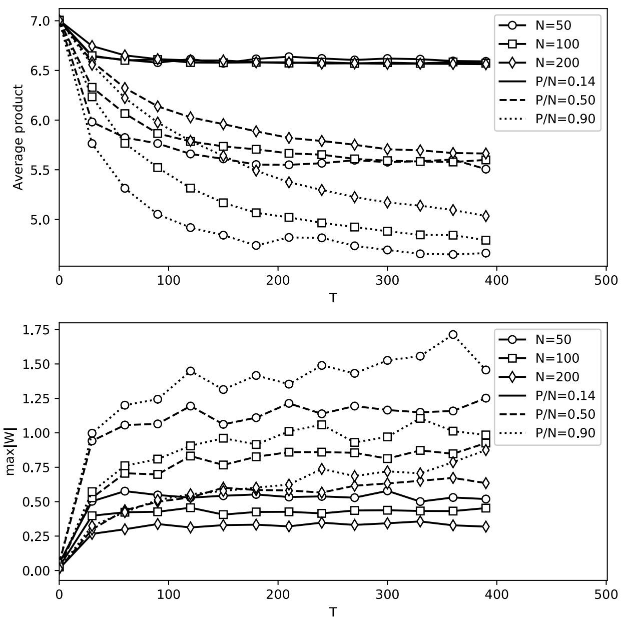


Fig. 9. Long-term non-orthogonal memory performance as a function of T, for various N and P. **Top**: The empirical average of $\sigma^{-1}(y_i(m_T(t)))W_{i,:}(T)\mathbf{x}(t)$, aggregated across all $i \in \{1, \dots, N\}$, $t \in \{1, \dots, T\}$, and all 30 trials for a given N and P. **Bottom**: The empirical global maximum of $\max_{i,j} |W_{i,j}(T)|$, taken across all of the same trials as shown on top.

vector for $N = 3$ and $T \leqslant 6$, with ρ fixed at .99. For illustrative purposes, we show the subset of weight vectors for which the training sequence ended in $\mathbf{x}_T = \rho\mathbf{1}$, and also use $\mathbf{x}_T = \rho\mathbf{1}$ as the line of sight for the visual. However, due to the symmetries of the cube, any choice of the 2^3 possibilities for the final $\mathbf{x}_T$ results in an identical picture. Note that, as per the proof of Theorem 1, $\mathbf{w}_T\mathbf{x}_T$ always equals $\tilde{y}_T$, which defines a plane orthogonal to $\mathbf{x}_T$. Therefore all possibilities for $\mathbf{w}_T$ visible in Fig. 10 (left) are an equal distance from the viewer. It appears that the possible weight vectors may be confined to a non-convex union of bounded, convex sets, and that the possibilities become dense in this union as $T \to \infty$. However, such a union proved difficult to identify and theoretically analyze in higher dimensions for the general case.

Having computed all possible $\mathbf{w}_T$, we also calculated the mean, standard deviation, and maximum of their Euclidean norms (Fig. 10, right). The results suggest that the expected value of the norm is indeed bounded independently of T, and in fact this can be shown using similar manipulations as in the proof of Theorem 1. The theoretical maximum of the norm appears to grow sub-linearly in T, but it is not visually obvious whether it is ultimately bounded.

5.2. Validation of the NVM on real programs

We next validated the NVM end-to-end by using it to emulate a real hand-written program: the list memorization and recall program in Fig. 3. We ran separate tests of this program using both orthogonal and non-orthogonal random activity patterns to represent distinct symbols. Non-orthogonal random patterns were generated by uniformly and independently selecting each pattern from $\{-\rho, +\rho\}^N$. We refer to this method as "Bernoulli". Orthogonal random patterns were generated based on Sylvester's construction for Hadamard matrices (Sylvester, 1867), which have orthogonal columns and ± 1 entries, and are closed under certain elementary row operations (namely, row switching, and row multiplication by ± 1). The canonical Hadamard matrix was

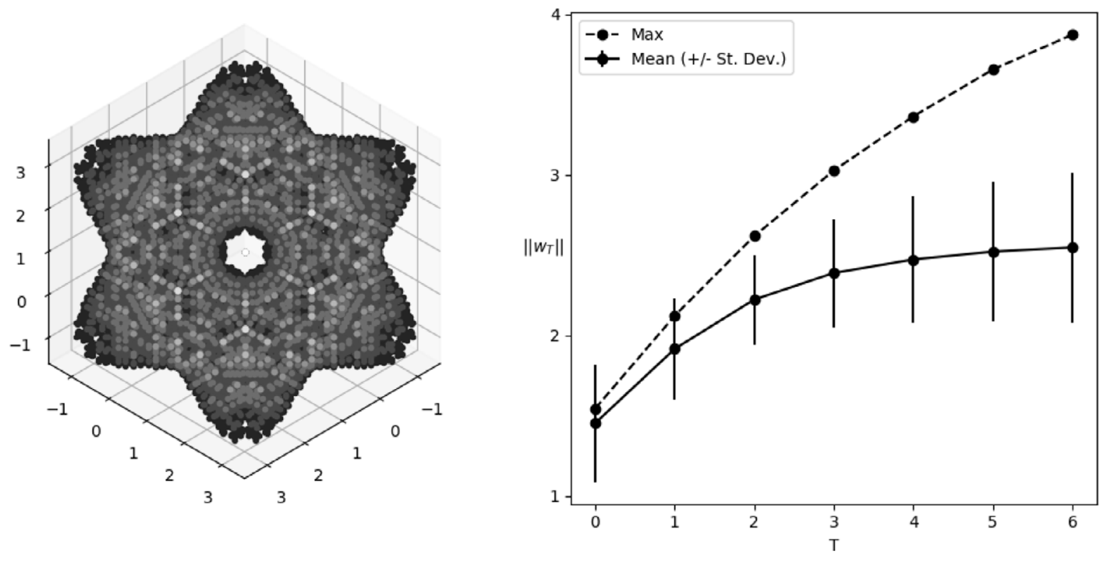


Fig. 10. Brute-force computation of all possible $\mathbf{w}_T$ up to $T = 6$ with $N = 3$. **Left**: All $\mathbf{w}_T$ for which $\mathbf{x}_T = \rho\mathbf{1}$. Weight vectors encountered at larger T are shown in darker gray. The $\mathbf{w}_T$ space is viewed with line of sight parallel to $\mathbf{1}$. **Right**: Maximum, mean, and standard deviation (as error bars) of $\|\mathbf{w}_T\|$ across all possibilities.

first constructed, and then in each trial, for each NVM region, a series of randomly chosen elementary row operations was used to randomize the matrix. The resulting columns were used as distinct patterns for the symbols in that region and trial. The orthogonal trials only used region sizes which were powers of 2, since this is required by Sylvester's construction.

For these experiments, the NVM API was called with a user-specification (see Fig. 1) that prescribed $\rho = .9999$, $\sigma = \tanh$, and four registers (`rloc`, `rval`, `rinp`, and `rout`, as required by the program in Fig. 3). The hidden gate layer `gh` and stack layers `sf` and `sb` (see Fig. 5) used fixed sizes of $N_{\mathtt{gh}} = 512$ and $N_{\mathtt{sf}} = N_{\mathtt{sb}} = 64$, while all other layer sizes in the user-specification were varied across different experimental trials as detailed in the following.

For both Bernoulli and orthogonal patterns, we tested the list program using a range of list input lengths, and a range of NVM region sizes. For each combination of list length and NVM size tested, we ran 30 independent random trials with different random samplings for the pattern encodings and the list contents. List lengths ranged from 10 to 40 elements, and each element was randomly selected with replacement from a set of 26 symbols denoted '`A`,'...,'`Z`.' Baseline sizes for each NVM region were then chosen based on the number of distinct patterns they needed to accommodate, denoted P. For example, `mf` and `mb` were sized proportionally to the length of the list, `ip` was sized proportionally to the number of lines (i.e., 17) in the program, and registers were sized proportionally to the number of distinct program symbols (26 letters, plus the other register names and label symbols in the program). For orthogonal patterns, this proportion was $N = 1 \times P$, although we rounded up to the nearest power of 2 as required by Sylvester's construction. For Bernoulli patterns, based on the rule of thumb identified in the previous section, we used the proportion $N = 1/0.05 \times P = 20 \times P$.

Finally, after sizing the regions for a given list input, an additional scaling factor of 0.75, 1, 1.25, or 1.5 was applied to N to determine final region sizes. This allowed us to assess performance degradation when an NVM instance is not perfectly sized. The final region sizes were then used for 30 independent random trials, in which an NVM instance with those sizes was constructed and used to run the list program on randomly sampled list input. At each time-step, we used the NVM codec (see Fig. 1) to convert between neural activity patterns in the I/O registers (`rinp` and `rout`) and the symbols they represented (letters `A`-`Z` or the separator `sep`). The program output for the trial was then compared with the input to measure the number of list elements that were correctly memorized and recalled. The fraction of correct elements, or "match rate", is shown in Fig. 11. The results show that orthogonal patterns never require additional scale factor beyond 1.0, whereas Bernoulli patterns sometimes require an additional scale factor strictly greater than 1.0. Given the baseline proportions in the previous paragraph, before additional scale factor, this implies that orthogonal patterns enable networks roughly 1/20 the size of Bernoulli patterns. This is notwithstanding the observation that performance can degrade more gracefully in the Bernoulli setting when regions are inadequately sized.

In additional experiments (not shown) we have also verified that the NVM can effectively perform small instances of the "Digit Symbol Substitution Test", a commonly administered cognitive task in human psychological experiments (Bettcher, Libon, Kaplan, Swenson, & Penney, 2011). This work is being used in an ongoing project to model cognitive deficiencies in PTSD patients. Additionally, we are using the NVM to encode causal knowledge used by meteorologists in predicting the weather, as part of a project on spatiotemporally chaotic time-series forecasting.

5.3. Empirical scalability of the NVM

To more comprehensively assess scalability of the NVM, we next devised a large-scale computer experiment using randomly generated programs (of course, such programs are very unlikely to do anything useful). In each experimental trial, a blank NVM instance was constructed and then multiple random programs were generated and assembled into the instance. Next, the programs were each executed for up to 100 time-steps, and the number of successful execution traces was recorded. We automated the success check by implementing a non-neural "reference" virtual machine that can also run NVM assembly, but operates directly on human-readable symbols rather than neural activity vectors. Successful execution was determined by comparing the NVM execution traces against those from the symbolic reference VM when running the same program. In particular, the "execution trace" was the list of symbols stored in each register at each time-step of program execution. In the case of the NVM, this trace was populated from the neural activity vectors at each time-step using the NVM codec (see Fig. 1). Appendix G provides more detail on the random program generation along with an example.

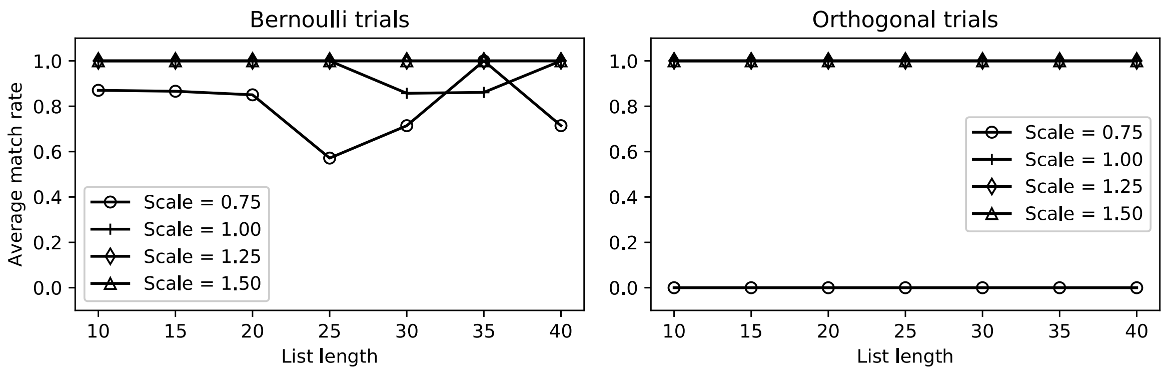


Fig. 11. NVM match rates on the list program for various scale factors and list lengths.

For these experiments, the NVM API was called with a user-specification (see Fig. 1) identical to that used in the previous section, except there were three general purpose registers ($\mathtt{r}_0$, $\mathtt{r}_1$, and $\mathtt{r}_2$). Layer sizes in the user-specification were varied across different experimental trials as follows. For a given set of programs in a particular trial, the number of lines and number of distinct symbols were each extracted. The former was used to size the program counter region ip, and the latter was used to size each register region $\mathtt{r}_i$. As before, we determined baseline region sizes with the ratio $N = 20 \times P$ for Bernoulli trials and $N = 1 \times P$ for orthogonal trials (rounded up to the nearest power of 2). The baseline region sizes were further scaled by factors ranging from 0.5 to 2.0, in order to better gauge performance degradation when an NVM instance is not perfectly sized. Each final scaling configuration was used for a separate trial.

Number of programs per trial ranged from 2 to 4, with number of lines in each program ranging from 4 to 512, and number of sub-routines in each program ranging from 1 to 8. For each configuration of program counts, line counts, and sub-routine counts, 30 independent repetitions of the experiment were performed. Distinct program contents, but with the same counts, were generated in each of the 30 repetitions. Finally, in each repetition, separate NVM instances were generated for scale factors ranging from 0.5 to 2.0 of the baseline sizing. Each instance was tested separately on the program contents for the repetition and compared with the reference VM. Any execution trace that did not match perfectly was marked as a failure. Success rate was then aggregated across the 30 repetitions for each configuration of program sizing and NVM sizing. To assess the scaling requirements of the NVM, we extracted the minimum region sizings required to achieve 100% and 90% success rates on a given program set size. Fig. 12 shows the results in terms of the number of distinct symbols in the programs and the resulting register size requirements. Fig. 13 shows the same data but viewed in terms of the total number of program lines, and the resulting ip size required. In many cases, the same minimum size was required for both 90% and 100% success rate, in which case gray datapoints are obscured by black datapoints at the same position. Bernoulli trials were limited to symbol/line counts on the order of 200, as larger counts had sizing requirements that were too large for our workstation.

As expected, both views demonstrate that empirical sizing requirements scale roughly linearly, and that Bernoulli patterns have much more demanding scaling requirements than orthogonal patterns. These results also show that reasonably sized NVM instances can reliably execute fairly large programs, particularly when orthogonal patterns are used.

6. Discussion

We have presented a Neural Virtual Machine (NVM) that can represent and execute multiple arbitrary programs using local learning and distributed representation. Our system has been explained and analyzed in mathematical detail, implemented, and validated empirically through large-scale computer experiments. Those experiments show that the NVM works in practice and scales to relatively large programs. Further, our computational experiments demonstrated that using orthogonal patterns to represent symbolic entities resulted in a qualitative reduction in the underlying neural network sizes needed to implement a specific NVM instance. Our primary contribution is that, unlike past work, the NVM is built on the principles of local learning and gated plasticity. As part of that contribution, we have introduced and characterized a novel learning rule that can emulate contiguous, random-access memory. We have also developed a more programmer-friendly language and toolchain for this system than other approaches. As noted above (Footnote 1), open source code for the NVM is freely available online.

One major limitation in the current NVM is that it can only learn complete, pre-authored programs. Ultimately, the NVM should learn how to synthesize its own programs based on experience. Future work should explore additional learning paradigms, such as imitation and reinforcement learning, and incorporate recent breakthroughs in neural program induction and synthesis (e.g., Bošnjak et al. (2017), Devlin, Uesato, Bhupatiraju, Singh, Mohamed and Kohli (2017) and Graves et al. (2016)).

Another avenue for future work is further theoretical analysis of the fast store-erase learning rule. We have shown that it always functions as desired in the orthogonal regime, and that in a Bernoulli regime where any source pattern is possible, it functions properly *in expectation*. The latter result should be improved and extended to the case where the set of possible source patterns is small relative to the size of the network. Bounds on variance, in addition to expected value, should also be derived. Lastly, the question of whether weight growth is ultimately bounded should be resolved. One other future research direction is to emulate a von Neumann, rather than Harvard, architecture, in which programs and data are both stored in the same NVM memory regions.

The NVM is certainly not a veridical model of the human brain. However, many of its operating principles have some basis in neuroscientific evidence and theories. While our primary learning rule is not *synapse*-local, like basic Hebbian learning, it is *neuron*-local: changes to a neuron's synapses only depend on information in other synapses of the same neuron at the current time-step (mathematically speaking, changes to one row of the weight matrix are independent of the other rows). Therefore, unlike error backpropagation of signals *across synapses to other neurons*,

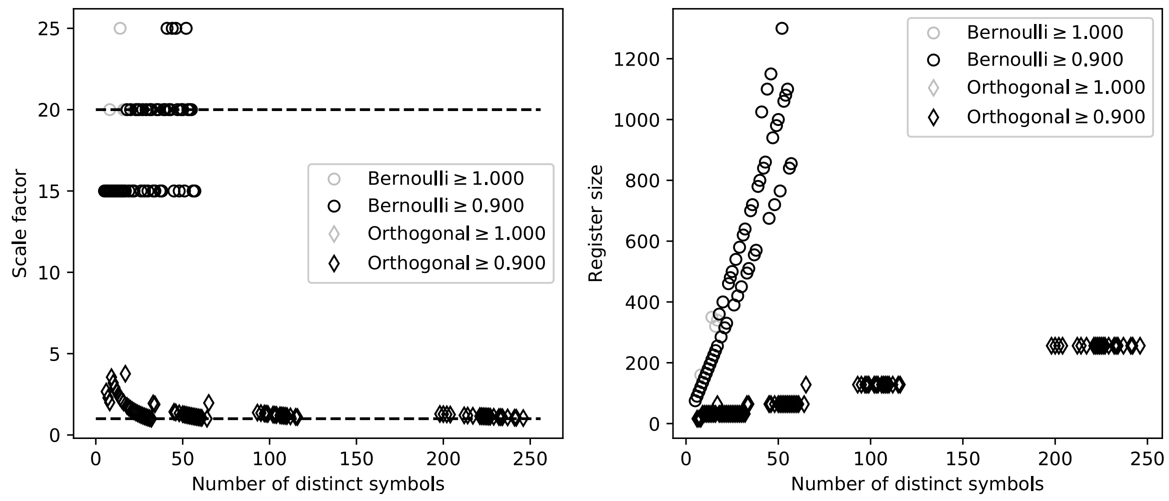


Fig. 12. Minimum NVM region scaling required to reach 100% and 90% success rates as a function of the number of distinct symbols in the programs used for a given trial, both for orthogonal trials and Bernoulli trials. Relative scaling is shown on left and absolute register sizes are shown on right. On left, the upper dashed line is the baseline scale factor expected for Bernoulli trials and the lower dashed line is the baseline for orthogonal trials. Also on the left, non-linear data is an artifact of the next-power-of-2 rounding in orthogonal trials.

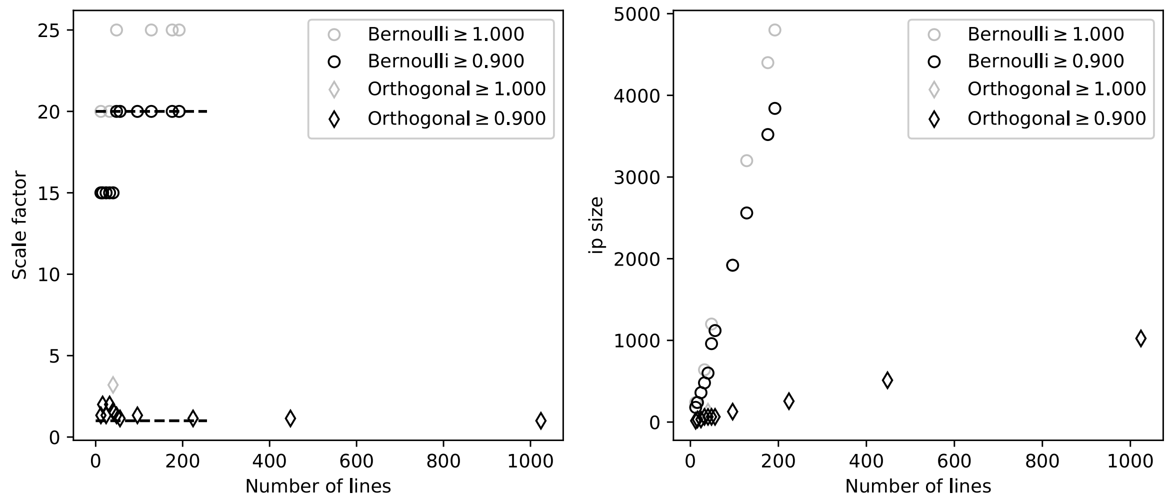


Fig. 13. Minimum NVM region scaling required to reach 100% and 90% success rates as a function of the total number of lines in the programs used for a given trial, both for orthogonal trials and Bernoulli trials. Relative scaling is shown on left and absolute `ip` size is shown on right. On left, the upper dashed line is the baseline scale factor expected for Bernoulli trials and the lower dashed line is the baseline for orthogonal trials.

which is considered biologically implausible, the fast store-erase learning rule only requires bidirectional information flow *within dendrites of the same neuron*, consistent with substantial empirical evidence that information can propagate bidirectionally within a single biological neuron's dendrites (Schiess, Urbanczik, & Senn, 2016). Using fast weight learning for comparison also has its basis in neuroscientific theory (Engel & Wang, 2011). Lastly, the use of multiplicative gating has a strong basis in evidence and a strong precedent in neural networks (Mehaffey, Doiron, Maler, & Turner, 2005; Salinas & Sejnowski, 2001; Shin & Ghosh, 1991). To the extent that the NVM captures underlying mechanisms for symbolic processing in the human brain, it may also be relevant to consciousness studies (Reggia, Huang, & Katz, 2017). Future work in this vein could measure proposed neurocomputational correlates of consciousness in a running NVM instance.

In terms of neuroanatomy (to a rough first approximation), we speculate that NVM regions may have reasonably plausible correlates in the human brain:

- `opc`, $\texttt{op}_1$, and $\texttt{op}_2$ might reasonably be associated with prefrontal cortex (PFC), given their central role in representing the task at hand. More specifically, `opc` might correspond to dorsolateral PFC, and $\texttt{op}_1$ and $\texttt{op}_2$ to ventrolateral PFC, given the characterization of dorsal-ventral gradient in terms of how something is done vs. what it is done to Goodale and Milner (1992) and O'Reilly (2010).
- Given their role in I/O with the external world and storage of task-relevant values, registers might be viewed as more posterior cortical areas (temporal, occipital, parietal). This is based on current theories that posterior cortex deals more with mental representation while anterior cortex deals more with mental processing. Relatedly, recent theories suggest that working memory is not locally represented in PFC per se, but rather distributed via a network between the anterior and posterior brain regions through which high-level cognitive function and mental representations interact (Lara & Wallis, 2015; Nee & D'Esposito, 2016). This further supports the view of registers as posterior cortical areas, given both their role in short-term, activity-based information storage, and the fact that $\texttt{op}_1$ and $\texttt{op}_2$ provide top-down signals that identify registers to be manipulated.

- `ci` and `co` might be related to anterior cingulate cortex, given its implication in error detection and conflict monitoring (Botvinick, Cohen, & Carter, 2004).
- `mf`, `mb`, `sf`, and `sb` are reminiscent of the hippocampus, given their role in memory. Their forward/backward functions were directly inspired by evidence of reverse replay in the hippocampus (Foster & Wilson, 2006). Moreover, the bidirectional pathways between hippocampus and cortex, which are critical for declarative memory (Eichenbaum, 2000), mirror the NVM pathways between `mf`, `mb`, `sf`, `sb` and the registers, in which activity flow and learning in both directions are critical for memory operations.
- `gh` and `go` might relate to portions of the basal ganglia (BG), given their close cooperation with `opc`, $\mathtt{op}_1$ and $\mathtt{op}_2$ in order to select procedural behaviors via top-down modulation of other regions and pathways. The organization of incoming and recurrent pathways for `gh`, followed by a feed-forward pathway to `go`, also mirrors the organization of pathways from cortex through striatum to the pallidal complex in BG (Haber, 2003).
- `ip` is responsible for sequencing instructions, rather than data, but nevertheless it uses a similar mechanism to `mf` and `sf`, which are reminiscent of the hippocampus. On the other hand, its central role in program execution might be more closely associated with PFC. Since incoming signals can cause `ip` to suddenly jump to remote branches or routines in a program, it might be viewed as responsive to motivational signals and involved in goal selection. In that sense, it might be conceptualized as analogous to ventromedial PFC in particular (Bechara, Damasio, Damasio, & Lee, 1999).

Certainly, these proposed neuroanatomical correlates are speculative, imperfect, and incomplete. In future work, the NVM could be refined to more closely match neuroscientific knowledge, and could be used to generate testable neuroscientific predictions. By iteratively testing such predictions and incorporating new findings into its neurocomputational designs, the NVM may one day find potential not only as a tool for artificial intelligence research but also for modeling symbolic processing in the human brain.

Acknowledgments

This work was supported by DARPA award HR00111890044 and ONR award N00014-19-1-2044.

Appendix A. NVM Program grammar and semantics

NVM assembly programs are written as line-separated lists of instructions in a minimalistic imperative style. Each instruction consists of an opcode which specifies what to do (e.g., move a symbol between registers), and zero or more operands (e.g., the source and destination registers for the move). Fig. A.14 shows the formal NVM assembly language syntax in extended Backus–Naur form (ISO/IEC 14977:1996 E, 1996). The full instruction semantics are described in Table A.3, which elaborates on Table 1 in the main text. Depending on the opcode, operands can either be register names, labels, or arbitrary symbols that are treated as literal values. Many opcodes have a '`v`' or '`r`' suffix, indicating whether an operand is a literal **v**alue or a **r**egister name. We include the suffix in the full instruction set here, but omit it in the examples of the main text, because a simple preprocessor is provided with the NVM that determines and appends the appropriate suffix.

```
program      = { line, "\v" }
line         = instruction
             | comment
instruction  = "\h", [label, ":\h"], operation, "\h", [comment]
comment      = "#", { "\w" | "\h" }
operation    = opcode, ["\h", operand, ["\h", operand]]
label        = "\w", { "\w" }
opcode       = "\w", { "\w" }
operand      = "\w", { "\w" }
```

Fig. A.14. The NVM assembly language grammar in extended Backus–Naur form. {...} indicates zero or more repetitions, [...] indicates optional content, | indicates disjunction, and quotes enclose terminals. Within terminal strings we use `\v` to denote vertical whitespace (line breaks), `\h` to denote horizontal whitespace (spaces, tabs) and `\w` to denote alphanumeric and other unreserved characters (excluding whitespace, '`#`', and '`:`').

Appendix B. Formal memory operations

The specific contiguous memory implemented by the NVM is shown in Fig. 2(b). An abstract Turing machine has infinitely many addresses, and each can store one of a finite set of symbols at a given time t. Let A denote the set of symbols and let $\mu^{(t)} : \mathbb{Z} \to A$ be a function that maps each address $m \in \mathbb{Z}$ to the symbol $a \in A$ stored there at time t. For example, in Fig. 2(b), $\mu^{(t)}(2) = \mathtt{a}$. Symbols are read from/written to memory via a "read/write head" that can be incremented or decremented one step at a time. Let $\mathtt{m}^{(t)}$ to denote the position of the read/write head at time t. Mathematically, reading a symbol from address $\mathtt{m}^{(t)}$ at time t amounts to evaluating $\mu^{(t)}(\mathtt{m}^{(t)})$. Writing a symbol a to address $\mathtt{m}^{(t)}$ at time t amounts to updating the μ function, so that $\mu^{(t+1)}(\mathtt{m}^{(t)}) = a$. Increment can be expressed as a function $\psi : \mathbb{Z} \to \mathbb{Z}$ that does not change over time: $\psi(\mathtt{m}^{(t)}) = \mathtt{m}^{(t)} + 1$. Similarly, ψ^{-1} is the decrement function.

To model reference/dereference operations, we allow symbols to be treated not only as values, but also like "pointer variable names" that temporarily point to a particular memory address. Mathematically, we introduce one more function $\phi^{(t)} : A \to \mathbb{Z}$ that maps pointer symbols to memory addresses and can change over time. For example, in Fig. 2(b), $\phi^{(t)}(\mathtt{b}) = 2$. A symbol a can be assigned as a pointer to a memory location m at time t by updating the mapping so that $\phi^{(t+1)}(a) = m$ (a so-called "reference" operation). Similarly, a symbol a can be dereferenced at time t by evaluating $\phi^{(t)}(a)$, thereby producing whatever memory address was previously bound to a. Currently there is no way in NVM assembly to indicate whether symbols are pointers or values; it is the responsibility of the programmer to ensure that no symbol is dereferenced unless it was already referenced earlier during program execution.

We use pathways between the regions `mf`, `mb`, and the registers to encode the functions μ, ψ, and ϕ for heap memory. Similarly, pathways involving `ip`, `opc`, $\mathtt{op}_1$, and $\mathtt{op}_2$ encode these functions for program memory, and pathways involving `sf`, `sb`, and `ip` encode these functions for stack memory. Updates to these functions (i.e., memory writes and pointer references) are performed using the fast store-erase learning rule (4) in the appropriate pathways. Evaluations of these functions (i.e., memory reads, pointer dereferences, and inc/decrements) are performed using activity flow across the appropriate pathways. More specifically, each memory operation is performed as indicated in Table B.4. Note that each neural operation in the table is a special case of the network rules, Eqs. (1) and (2), with the gates appropriately set. For example, a memory write occurs when $\ell_{\mathtt{r}_i,\mathtt{mf}}(t) = 1$, and an increment occurs when $d_{\mathtt{mf}}(t)$, $d_{\mathtt{mb}}(t)$,

Table A.3
The full NVM instruction set semantics.

Syntax	Description
`nop`	Do nothing (no operation).
`movv` *reg sym*	Move the literal symbol *sym* into register *reg*.
`movr` *dst src*	Move (copy) the symbol in register *src* into register *dst*.
`jmpv` *lab*	Jump to the line labeled *lab*.
`jmpr` *reg*	Jump to the line whose label is stored in register *reg*.
`cmpv` *reg sym*	Compare the symbol in register *reg* with the literal symbol *sym*.
`cmpr` *reg1 reg2*	Compare the symbols in registers *reg1* and *reg2*.
`jiev` *lab*	Jump, if the last compare was equal, to the line labeled *lab*.
`jier` *reg*	Like `jiev` but the target label is stored in register *reg*.
`subv` *lab*	Call the sub-routine starting on the line labeled *lab*.
`subr` *reg*	Like `subv` but the target label is stored in register *reg*.
`ret`	Return from the current sub-routine to the line where it was called.
`nxt`	Shift the read/write head to the next memory location.
`prv`	Shift the read/write head to the previous memory location.
`mem` *reg*	Write the symbol in register *reg* to the current memory location.
`rem` *reg*	Read the symbol at the current memory location into register *reg*.
`ref` *reg*	Reference the current memory location by the symbol in register *reg*.
`drf` *reg*	Dereference a new memory location from the symbol in register *reg*.
`exit`	Halt execution.

Table B.4
Mathematical and neural representations of each NVM memory operation.

Operation	Mathematical	Neural
Read	$a = \mu^{(t)}(m)$	$\mathbf{v}^{\mathtt{r}_i}[a] = \sigma_{\mathtt{r}_i}(W^{\mathtt{r}_i,\mathtt{mf}}(t)\mathbf{v}^{\mathtt{mf}}[m])$
Write	$\mu^{(t)} \to \mu^{(t+1)}$	$W^{\mathtt{r}_i,\mathtt{mf}}(t+1) = W^{\mathtt{r}_i,\mathtt{mf}}(t) + \Delta W^{\mathtt{r}_i,\mathtt{mf}}(t)$
Increment	$m + 1 = \psi(m)$	$\mathbf{v}^{\mathtt{mf}}[m+1] = \sigma_{\mathtt{mf}}(W^{\mathtt{mf},\mathtt{mf}}(t)\mathbf{v}^{\mathtt{mf}}[m])$ $\mathbf{v}^{\mathtt{mb}}[m+1] = \sigma_{\mathtt{mb}}(W^{\mathtt{mb},\mathtt{mf}}(t)\mathbf{v}^{\mathtt{mb}}[m])$
Decrement	$m - 1 = \psi^{-1}(m)$	$\mathbf{v}^{\mathtt{mf}}[m-1] = \sigma_{\mathtt{mf}}(W^{\mathtt{mf},\mathtt{mb}}(t)\mathbf{v}^{\mathtt{mb}}[m])$ $\mathbf{v}^{\mathtt{mb}}[m-1] = \sigma_{\mathtt{mb}}(W^{\mathtt{mb},\mathtt{mb}}(t)\mathbf{v}^{\mathtt{mb}}[m])$
Dereference	$m = \phi^{(t)}(a)$	$\mathbf{v}^{\mathtt{mf}}[m] = \sigma_{\mathtt{mf}}(W^{\mathtt{mf},\mathtt{r}_i}(t)\mathbf{v}^{\mathtt{r}_i}[a])$ $\mathbf{v}^{\mathtt{mb}}[m] = \sigma_{\mathtt{mb}}(W^{\mathtt{mb},\mathtt{r}_i}(t)\mathbf{v}^{\mathtt{r}_i}[a])$
Reference	$\phi^{(t)} \to \phi^{(t+1)}$	$W^{\mathtt{mf},\mathtt{r}_i}(t+1) = W^{\mathtt{mf},\mathtt{r}_i}(t) + \Delta W^{\mathtt{mf},\mathtt{r}_i}(t)$ $W^{\mathtt{mb},\mathtt{r}_i}(t+1) = W^{\mathtt{mb},\mathtt{r}_i}(t) + \Delta W^{\mathtt{mb},\mathtt{r}_i}(t)$

$s_{\mathtt{mf},\mathtt{mf}}(t)$, and $s_{\mathtt{mb},\mathtt{mf}}(t)$ are all 1. The requisite gating patterns for each memory operation are shown visually in Fig. 6.

The available addresses in heap and stack memory are independent of any particular program. In other words, ψ and ψ^{-1} for these memories may be evaluated, but are never updated, during program execution. Hence these functions are only encoded (i.e., updated) once when a blank NVM instance is first created (see Fig. 1). Correspondingly, although we include time-step notation for consistency, the weight matrices $W^{\mathtt{mf},\mathtt{mf}}$, etc. shown for increment/decrement in Table B.4 are fixed after NVM instance construction. All other function updates and evaluations for all memories occur during program assembly and execution.

Appendix C. NVM assembly routine

NVM programs are preprocessed, assembled, and loaded as follows. First, the preprocessor appends `v` or `r` to each opcode as described in Appendix A, assigns unlabeled lines default labels based on line number, and populates unused operands with a `null` symbol.

Let L be the number of lines in a program $\mathbb{P}$, let $\mathbb{P}^{(p)}_{\mathtt{ip}} : \mathbb{P}^{(p)}_{\mathtt{opc}}\ \mathbb{P}^{(p)}_{\mathtt{op}_1}\ \mathbb{P}^{(p)}_{\mathtt{op}_2}$ be the pth instruction of $\mathbb{P}$, with p ranging from 1 to L, and let $\mathbb{P}^{(0)}_{\mathtt{ip}}$ denote a special label used to initiate the program. Also let $\Delta W(\mathbf{y}, \mathbf{x})$ abbreviate the fast store-erase learning rule (4) applied to source pattern $\mathbf{x}$ and target pattern $\mathbf{y}$. A program can then be "assembled" in an NVM instance using the procedure in Fig. C.15. The first inner loop writes the sequence of program instructions to program memory. The second inner loop enables jumps by associating any label that might occur as an operand in $\mathtt{op}_1$, or in a register specified by $\mathtt{op}_1$, with its corresponding

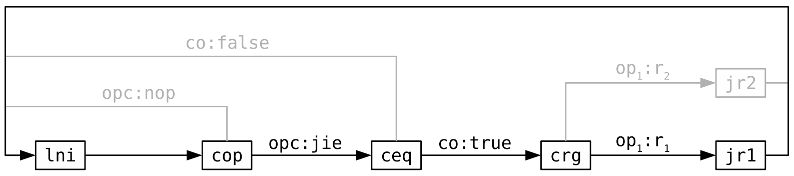
```
1:  for $p \in \{1, ..., L\}$ do
2:      for $\mathtt{q} \in \{\mathtt{ip}, \mathtt{opc}, \mathtt{op}_1, \mathtt{op}_2\}$ do
3:          $W^{\mathtt{q},\mathtt{ip}} \leftarrow W^{\mathtt{q},\mathtt{ip}} + \Delta W^{\mathtt{q},\mathtt{ip}}(\mathbf{v}^{\mathtt{q}}[\mathbb{P}^{(p)}_{\mathtt{q}}], \mathbf{v}^{\mathtt{ip}}[\mathbb{P}^{(p-1)}_{\mathtt{ip}}])$
4:      end for
5:      for $\mathtt{r} \in \{\mathtt{op}_1, \mathtt{r}_1, ...\mathtt{r}_R\}$ do
6:          $W^{\mathtt{ip},\mathtt{r}} \leftarrow W^{\mathtt{ip},\mathtt{r}} + \Delta W^{\mathtt{ip},\mathtt{r}}(\mathbf{v}^{\mathtt{ip}}[\mathbb{P}^{(p)}_{\mathtt{ip}}], \mathbf{v}^{\mathtt{r}}[\mathbb{P}^{(p)}_{\mathtt{ip}}])$
7:      end for
8:  end for
9:  for each program symbol $\mathtt{c}$ do
10:     for $\mathtt{q}, \mathtt{r} \in \{\mathtt{ci}, \mathtt{r}_1, ...\mathtt{r}_R\} \times \{\mathtt{r}_1, ...\mathtt{r}_R\}$ do
11:         $W^{\mathtt{q},\mathtt{r}} \leftarrow W^{\mathtt{q},\mathtt{r}} + \Delta W^{\mathtt{q},\mathtt{r}}(\mathbf{v}^{\mathtt{q}}[\mathtt{c}], \mathbf{v}^{\mathtt{r}}[\mathtt{c}])$
12:     end for
13: end for
```

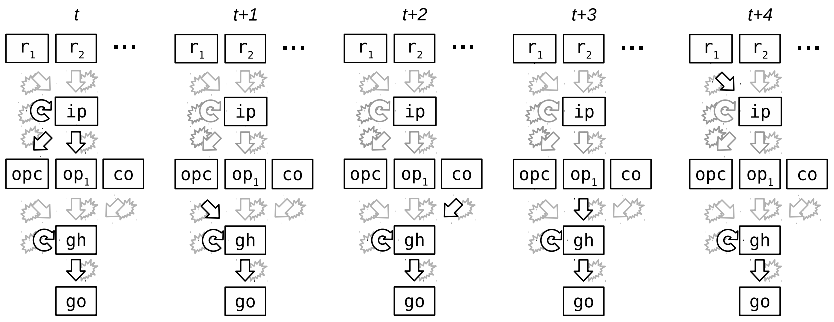
Fig. C.15. The NVM assembly procedure. Commentary provided in the appendix text.

pattern in `ip`. The third inner loop enables register moves and comparisons by associating different patterns in different regions for the same symbol, as described previously. Finally, once a program $\mathbb{P}$ is assembled in the NVM, it can be "loaded" at any time by initializing `ip` to $\mathbb{P}^{(0)}_{\mathtt{ip}}$ and then "executed" by running the network dynamics.

Appendix D. The NVM gating FSM

Fig. D.16 illustrates how the NVM gating dynamics correspond to a finite state machine (FSM), encoded by various NVM regions that are coordinated to emulate one instruction cycle during program execution. Note that although the NVM instruction cycle *emulates* a symbolic FSM, it is *implemented* with purely neural distributed processing. As an example we consider a "jump if equal" instruction that performs a conditional jump to a different program position if the most recent comparison operation produced `true`. Otherwise, it advances sequentially to the next program line. This instruction is signified by an opcode `jie` and one operand in $\mathtt{op}_1$. The operand is the name of one of the registers $\mathtt{r}_k$, and the named register is assumed to contain the





Region	*t*	*t+1*	*t+2*	*t+3*	*t+4*
r_1	p'	p'	p'	p'	p'
ip	p	p+1	p+1	p+1	p'
opc		jie	jie	jie	jie
op_1		r_1	r_1	r_1	r_1
co	true	true	true	true	true
gh	lni	cop	ceq	crg	jr1

Fig. D.16. An illustration of one instruction cycle in the NVM, as implemented in the distributed processing of the NVM's recurrent neural networks. **Top:** A small subgraph of the gating FSM. Rectangles denote FSM states and arrows denote transitions, labeled by the input tokens (if any) that produce them. Gray lines indicate FSM paths that might have occurred if different input tokens had been present. **Middle:** The "output symbols" (i.e., gating patterns) produced by the successive states visited by the FSM, using the same conventions as Fig. 6. Only the relevant subset of NVM regions is shown. **Bottom:** The symbols present in the relevant NVM regions during each successive time-step of the FSM sequence. Additional commentary provided in the text.

target label of the jump. In Fig. D.16 we assume that $\mathtt{op}_1$ contains $\mathtt{r}_1$, i.e. $\mathbf{v}^{\mathtt{op}_1}(t) = \mathbf{v}^{\mathtt{op}_1}[\mathtt{r}_1]$, so the instruction being emulated is "jie $\mathtt{r}_1$". This instruction is emulated as follows.

Suppose that the (emulated) FSM is in a "load next instruction" state at time t. For convenience we use mnemonics like lni to refer to this and other FSM states, but these mnemonics are not intended to be programmer-accessible symbols, and they are represented by arbitrary random patterns in gh. At this point in time, we assume the program counter is currently at line p, represented by the pattern $\mathbf{v}^{\mathtt{ip}}[\mathtt{p}]$ in ip. We also assume the most recent comparison operation in the past produced a true symbol in co. Lastly, we assume the symbol p', representing another program position p'$\neq$p+1, was previously stored in register $\mathtt{r}_1$. We make no assumptions about the instruction on line p−1 of the program, indicated by the blank table cells in Fig. D.16 (bottom).

The gating output for FSM state lni opens several pathways. It "reads" the current opcode and operands from program memory, by ungating activity in the $\mathtt{ip} \rightarrow \mathtt{opc}$ and $\mathtt{ip} \rightarrow \mathtt{op}_k$ pathways. It also opens the recurrent $\mathtt{ip} \rightarrow \mathtt{ip}$ pathway. Similarly to memory increments in mf, this advances the instruction pointer to the next line of the program at $\mathtt{p}+1$, so that it will be executed next once the current operation is complete (if the conditional jump does not occur).

Next, the FSM transitions to a "check opcode" (cop) state at time $t+1$ that serves as an FSM branching point depending on the current opcode. The gating output at this point ungates activity in the $\mathtt{opc} \rightarrow \mathtt{gh}$ pathway, so that the pattern in the opc region can serve as an FSM "input token" by influencing the dynamics of gh.

At this point, if the pattern in opc had represented a "no-op" opcode $\mathbf{v}^{\mathtt{opc}}[\mathtt{nop}]$, then the FSM would have done nothing and transitioned back to the lni state so that the instruction at line $\mathtt{p}+1$ would be executed. Instead, since the current opcode in the opc region is "jie", the FSM transitions to a "check equality" (ceq) state at time $t+2$. This state serves as an FSM branching point depending on whether the most recent equality comparison was true, which should determine whether or not to perform a conditional jump. Accordingly, the gating pattern at $t+2$ ungates activity flow in the $\mathtt{co} \rightarrow \mathtt{gh}$ pathway, so that the relevant FSM input token in co (compare output) is accessible.

At this point, if co had contained false, the most recent comparison would have been false, and instead of performing a jump, the FSM would transition back to lni so that the next instruction at line $\mathtt{p}+1$ would be executed. Instead, since the "co" region contains true, the FSM transitions to a "check register" (crg) state at time $t+3$, which serves as a branching point depending on which register contains the jump target. The gating pattern at this point allows $\mathtt{op}_1$ to influence gh so that the FSM can branch depending on which register contains the jump target, which is specified by the first operand.

Next, the FSM branches depending on which register contains the target line for the jump. Since $\mathtt{op}_1$ contains a pattern signifying $\mathtt{r}_1$, the FSM transitions to a "jump register 1" state (jr1) at time $t+4$, which will finalize the jump based on the contents of $\mathtt{r}_1$. If instead $\mathtt{op}_1$ had indicated another register such as $\mathtt{r}_2$, the FSM would have transitioned to state jr2 which would finalize a jump based on $\mathtt{r}_2$ instead of $\mathtt{r}_1$.

Finally, the jump is enacted by the gating pattern output in state jr1. Specifically, activity across the $\mathtt{r}_1 \rightarrow \mathtt{ip}$ pathway is ungated, so that the new program position $\mathtt{p}+1$ is overwritten with whatever position p' was stored in $\mathtt{r}_1$. This is exactly the same principle used for pointer dereferences in mf. As a result,

Table D.5
Gating sequence for `jie` instruction cycle.

Gate sequence	Reduced Eqs. (1)–(2)
0. $(\mathtt{q},\mathtt{ip})$, $\mathtt{q} \in \{\mathtt{ip}, \mathtt{opc}, \mathtt{op}_1, \mathtt{op}_2\}$	$\mathbf{v}^{\mathtt{ip}}(1) = \sigma_{\mathtt{ip}}(W^{\mathtt{ip},\mathtt{ip}}(0)\mathbf{v}^{\mathtt{ip}}(0))$
	$\mathbf{v}^{\mathtt{opc}}(1) = \sigma_{\mathtt{opc}}(W^{\mathtt{opc},\mathtt{ip}}(0)\mathbf{v}^{\mathtt{ip}}(0))$
	$\mathbf{v}^{\mathtt{op}_1}(1) = \sigma_{\mathtt{op}_1}(W^{\mathtt{op}_1,\mathtt{ip}}(0)\mathbf{v}^{\mathtt{ip}}(0))$
	$\mathbf{v}^{\mathtt{op}_2}(1) = \sigma_{\mathtt{op}_2}(W^{\mathtt{op}_2,\mathtt{ip}}(0)\mathbf{v}^{\mathtt{ip}}(0))$
1. $(\mathtt{gh},\mathtt{opc})$	$\mathbf{v}^{\mathtt{gh}}(2) = \sigma_{\mathtt{gh}}(W^{\mathtt{gh},\mathtt{gh}}(1)\mathbf{v}^{\mathtt{gh}}(1) + W^{\mathtt{gh},\mathtt{opc}}(1)\mathbf{v}^{\mathtt{opc}}(1))$
2. $(\mathtt{gh},\mathtt{co})$	$\mathbf{v}^{\mathtt{gh}}(3) = \sigma_{\mathtt{gh}}(W^{\mathtt{gh},\mathtt{gh}}(2)\mathbf{v}^{\mathtt{gh}}(2) + W^{\mathtt{gh},\mathtt{co}}(2)\mathbf{v}^{\mathtt{co}}(2))$
Continues if $\mathbf{v}^{\mathtt{co}}(2) = \mathbf{v}^{\mathtt{co}}[\mathtt{true}]$	
3. $(\mathtt{gh},\mathtt{op}_1)$	$\mathbf{v}^{\mathtt{gh}}(4) = \sigma_{\mathtt{gh}}(W^{\mathtt{gh},\mathtt{gh}}(3)\mathbf{v}^{\mathtt{gh}}(3) + W^{\mathtt{gh},\mathtt{op}_1}(3)\mathbf{v}^{\mathtt{op}_1}(3))$
At this step, $\mathbf{v}^{\mathtt{op}_1}(3) = \mathbf{v}^{\mathtt{op}_1}[\mathtt{r}]$	
4. $(\mathtt{ip},\mathtt{r})$	$\mathbf{v}^{\mathtt{ip}}(5) = \sigma_{\mathtt{ip}}(W^{\mathtt{ip},\mathtt{r}}(4)\mathbf{v}^{\mathtt{r}}(4))$
5. $[\mathtt{ip}]$	$\mathbf{v}^{\mathtt{ip}}(6) = \sigma_{\mathtt{ip}}(\omega_{\mathtt{ip}}\mathbf{v}^{\mathtt{ip}}(5))$

Table D.6
Gating sequences for other NVM instructions.

Instruction	Gate sequence	Reduced Eqs. (1)–(2)
`jmpv a`	2. $(\mathtt{ip},\mathtt{op}_1)$	$\mathbf{v}^{\mathtt{ip}}(3) = \sigma_{\mathtt{ip}}(W^{\mathtt{ip},\mathtt{op}_1}(2)\mathbf{v}^{\mathtt{op}_1}(2))$
	3. $[\mathtt{ip}]$	$\mathbf{v}^{\mathtt{ip}}(4) = \sigma_{\mathtt{ip}}(\omega_{\mathtt{ip}}\mathbf{v}^{\mathtt{ip}}(3))$
`jmpr r`	2. $(\mathtt{gh},\mathtt{op}_1)$	$\mathbf{v}^{\mathtt{gh}}(3) = \sigma_{\mathtt{gh}}(W^{\mathtt{gh},\mathtt{op}_1}(2)\mathbf{v}^{\mathtt{op}_1}(2))$
	At this step, $\mathbf{v}^{\mathtt{op}_1}(2) = \mathbf{v}^{\mathtt{op}_1}[\mathtt{r}]$	
	3. $(\mathtt{ip},\mathtt{r})$	$\mathbf{v}^{\mathtt{ip}}(4) = \sigma_{\mathtt{ip}}(W^{\mathtt{ip},\mathtt{r}}(3)\mathbf{v}^{\mathtt{r}}(3))$
	4. $[\mathtt{ip}]$	$\mathbf{v}^{\mathtt{ip}}(5) = \sigma_{\mathtt{ip}}(\omega_{\mathtt{ip}}\mathbf{v}^{\mathtt{ip}}(4))$
`subv a`	2. $\{\mathtt{ip},\mathtt{sf}\}$	$W^{\mathtt{ip},\mathtt{sf}}(3) = W^{\mathtt{ip},\mathtt{sf}}(2) + \Delta W^{\mathtt{ip},\mathtt{sf}}(2)$
	3. $(\mathtt{sf},\mathtt{sf})$,$(\mathtt{sb},\mathtt{sf})$	$\mathbf{v}^{\mathtt{sf}}(4) = \sigma_{\mathtt{sf}}(W^{\mathtt{sf},\mathtt{sf}}(3)\mathbf{v}^{\mathtt{sf}}(3))$
		$\mathbf{v}^{\mathtt{sb}}(4) = \sigma_{\mathtt{sb}}(W^{\mathtt{sb},\mathtt{sf}}(3)\mathbf{v}^{\mathtt{sf}}(3))$
	4. $(\mathtt{ip},\mathtt{op}_1)$	$\mathbf{v}^{\mathtt{ip}}(5) = \sigma_{\mathtt{ip}}(W^{\mathtt{ip},\mathtt{op}_1}(4)\mathbf{v}^{\mathtt{op}_1}(4))$
	5. $[\mathtt{ip}]$	$\mathbf{v}^{\mathtt{ip}}(6) = \sigma_{\mathtt{ip}}(\omega_{\mathtt{ip}}\mathbf{v}^{\mathtt{ip}}(5))$
`subr r`	2. $\{\mathtt{ip},\mathtt{sf}\}$	$W^{\mathtt{ip},\mathtt{sf}}(3) = W^{\mathtt{ip},\mathtt{sf}}(2) + \Delta W^{\mathtt{ip},\mathtt{sf}}(2)$
	3. $(\mathtt{sf},\mathtt{sf})$,$(\mathtt{sb},\mathtt{sf})$	$\mathbf{v}^{\mathtt{sf}}(4) = \sigma_{\mathtt{sf}}(W^{\mathtt{sf},\mathtt{sf}}(3)\mathbf{v}^{\mathtt{sf}}(3))$
		$\mathbf{v}^{\mathtt{sb}}(4) = \sigma_{\mathtt{sb}}(W^{\mathtt{sb},\mathtt{sf}}(3)\mathbf{v}^{\mathtt{sf}}(3))$
	4.$(\mathtt{gh},\mathtt{op}_1)$	$\mathbf{v}^{\mathtt{gh}}(5) = \sigma_{\mathtt{gh}}(W^{\mathtt{gh},\mathtt{op}_1}(4)\mathbf{v}^{\mathtt{op}_1}(4))$
	At this step, $\mathbf{v}^{\mathtt{op}_1}(4) = \mathbf{v}^{\mathtt{op}_1}[\mathtt{r}]$	
	5. $(\mathtt{ip},\ \mathtt{r})$	$\mathbf{v}^{\mathtt{ip}}(6) = \sigma_{\mathtt{ip}}(W^{\mathtt{ip},\mathtt{r}}(5)\mathbf{v}^{\mathtt{r}}(5))$
	6. $[\mathtt{ip}]$	$\mathbf{v}^{\mathtt{ip}}(7) = \sigma_{\mathtt{ip}}(\omega_{\mathtt{ip}}\mathbf{v}^{\mathtt{ip}}(6))$
`ret`	2. $(\mathtt{sf},\mathtt{sb})$,$(\mathtt{sb},\mathtt{sb})$	$\mathbf{v}^{\mathtt{sf}}(3) = \sigma_{\mathtt{sf}}(W^{\mathtt{sf},\mathtt{sb}}(2)\mathbf{v}^{\mathtt{sb}}(2))$
		$\mathbf{v}^{\mathtt{sb}}(3) = \sigma_{\mathtt{sb}}(W^{\mathtt{sb},\mathtt{sb}}(2)\mathbf{v}^{\mathtt{sb}}(2))$
	3. $(\mathtt{ip},\mathtt{sf})$	$\mathbf{v}^{\mathtt{ip}}(4) = \sigma_{\mathtt{ip}}(W^{\mathtt{ip},\mathtt{sf}}(3)\mathbf{v}^{\mathtt{sf}}(3))$
	4. $[\mathtt{ip}]$	$\mathbf{v}^{\mathtt{ip}}(5) = \sigma_{\mathtt{ip}}(\omega_{\mathtt{ip}}\mathbf{v}^{\mathtt{ip}}(4))$

when the FSM returns to `lni` at time $t + 5$, `ip` will point to the target line `p'` of the jump, and the next instruction cycle will proceed from program position `p'` instead of `p+1`.

The gating sequence depicted in Fig. D.16 is expressed more formally in Table D.5. As in Table 2, we use the notation $(\mathtt{q}, \mathtt{r})$ to abbreviate the gate pattern $d_{\mathtt{q}}(t) = s_{\mathtt{q},\mathtt{r}}(t) = 1$, which reduces Eq. (1) as shown in the table, thereby allowing activity to propagate to `q` from `r`. Additionally, we use [q] to denote the "saturation" gate pattern $d_{\mathtt{q}}(t) = s_{\mathtt{q},\mathtt{r}}(t) = 0$, which reduces Eq. (1) as shown, thereby allowing layers to converge towards the closest attractor state. In fact, this is the default gating pattern for all layers at all time-steps (excluding gh and go) unless otherwise listed in the table. It is made explicit in step 5 because often `ip` needs additional time-steps to sufficiently stabilize at a new program memory address before loading the next instruction. Also note that steps 0–1 are not specific to `jie`, but serve the purpose of loading a new instruction and always occur at the beginning of every instruction cycle. Lastly, we note that the gating operations (gh, gh), (go, gh) always occur at every time-step in order to advance the gating FSM (not shown in the table to avoid clutter).

We have described in detail the gating sequences for register moves (Section 2.4.1), symbol comparisons (Section 2.4.4), memory operations Appendix B, and conditional jumps (this section). The remaining NVM instructions were mentioned in Section 2.4.5, use similar principles, and are presented formally in Table D.6. The numbering starts from 2 to account for the first two steps of the instruction cycle (Table D.5) that load the current instruction (omitted in Table D.6 to avoid clutter).

Appendix E. Flashing an NVM instance

As illustrated in Appendix D, a complete gating FSM can be used to fully specify an "instruction set" for an NVM instance. This instruction set must then be "flashed" onto the fixed control flow weights when the instance is constructed, analogous to how non-volatile microcontroller memory is "flashed" with device firmware that changes infrequently. The NVM flash procedure is done as follows. Let $s' = \tau(s, a)$ be any transition in the FSM, from state s and input token a to new state s'. The states s' and s are represented by patterns in gh. The input token a is represented

```
1: ⊕ "mov rstt $q_0$ # initialize starting state\n"
2: ⊕ "loop:  rem rsym # load symbol from tape\n"
3: for $q \in Q$ do
4:     ⊕ "cmp rstt $q$ # check if current state equals $q$\n"
5:     ⊕ "jie if$q$ # If so, jump to code for $q$\n"
6: end for
7: for $q \in Q$ do
8:     ⊕ "if$q$:  "
9:     if $q \in F$ then
10:         ⊕ "exit\n"
11:     else
12:         for $\gamma \in \Gamma$ do
13:             ⊕ "cmp rsym $\gamma$ # check if current symbol equals $\gamma$\n"
14:             ⊕ "jie if$q\gamma$ # If so, go to ($q$,$\gamma$) transition\n"
15:         end for
16:     end if
17: end for
18: for $q \in Q$ do
19:     for $\gamma \in \Gamma$ do
20:         $(q', \gamma', d) \leftarrow \delta(q, \gamma)$
21:         ⊕ "if$q\gamma$:  mov rstt $q'$ # update current state to $q'$\n"
22:         ⊕ "mov rsym $\gamma'$ # stage new symbol $\gamma'$\n"
23:         ⊕ "mem rsym # write $\gamma'$ to tape\n"
24:         if $d = \mathrm{L}$ then
25:             ⊕ "nxt # shift tape left\n"
26:         else
27:             ⊕ "prv # shift tape right\n"
28:         end if
29:         ⊕ "jmp loop # repeat\n"
30:     end for
31: end for
```

Fig. F.17. The algorithm for converting a Turing machine $\langle Q, q_0, \Gamma, F, \delta\rangle$ to an NVM program. Appending a string to an (initially empty) program is denoted ⊕ "...", and '\n' denotes a line break.

by the vertical concatenation of patterns in co, opc, and each op$_k$, multiplied by 0 if the respective pathway to gh is gated shut during the transition. These column vectors can then be collected into two matrices of "training data" X and Y, where each column of X represents a transition input (s, a) and the corresponding column of Y represents the corresponding transition output $s' = \tau(s, a)$. For example, the portions of X and Y corresponding to the first few transitions in Fig. D.16 are as follows:

$$X = \begin{bmatrix} \dots & \mathbf{v}^{\mathrm{gh}}[\texttt{lni}] & \mathbf{v}^{\mathrm{gh}}[\texttt{cop}] & \mathbf{v}^{\mathrm{gh}}[\texttt{cop}] & \mathbf{v}^{\mathrm{gh}}[\texttt{ceq}] & \mathbf{v}^{\mathrm{gh}}[\texttt{ceq}] & \dots \\ \dots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{v}^{\mathrm{co}}[\texttt{true}] & \mathbf{v}^{\mathrm{co}}[\texttt{false}] & \dots \\ \dots & \mathbf{0} & \mathbf{v}^{\mathrm{opc}}[\texttt{nop}] & \mathbf{v}^{\mathrm{opc}}[\texttt{jie}] & \mathbf{0} & \mathbf{0} & \dots \\ \dots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots \\ \dots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots \end{bmatrix}$$

$$Y = \begin{bmatrix} \dots & \mathbf{v}^{\mathrm{gh}}[\texttt{cop}] & \mathbf{v}^{\mathrm{gh}}[\texttt{lni}] & \mathbf{v}^{\mathrm{gh}}[\texttt{ceq}] & \mathbf{v}^{\mathrm{gh}}[\texttt{crg}] & \mathbf{v}^{\mathrm{gh}}[\texttt{lni}] & \dots \end{bmatrix}$$

Neural dynamics in gh should be such that each column in X transitions to the corresponding column in Y. That is, if

$$\begin{bmatrix} s_{\mathrm{gh,gh}}(t)\mathbf{v}^{\mathrm{gh}}(t) \\ s_{\mathrm{gh,co}}(t)\mathbf{v}^{\mathrm{co}}(t) \\ s_{\mathrm{gh,opc}}(t)\mathbf{v}^{\mathrm{opc}}(t) \\ s_{\mathrm{gh,op_1}}(t)\mathbf{v}^{\mathrm{op_1}}(t) \\ s_{\mathrm{gh,op_2}}(t)\mathbf{v}^{\mathrm{op_2}}(t) \end{bmatrix}$$

matches a column of X, then $\mathbf{v}^{\mathrm{gh}}(t+1)$ should match the corresponding column of Y. The issue then is to find the set of

```
# Simulate a small Turing machine with two states (X, Y) and two tape symbols (A,B)
# (state X, read symbol A) -> (state Y, write symbol B, shift tape right)
# (state X, read symbol B) -> (state X, write symbol A, shift tape left)
# (state Y, read symbol A) -> (state X, write symbol B, shift tape left)
# (state Y, read symbol B) -> (state Y, write symbol A, shift tape right)
# This Turing machine never terminates (no accepting states)

        mov rstt X      # Initialize starting state
loop:   rem rsym        # Load symbol from tape
        # Branch according to current state
        cmp rstt X      # Check if current state equals X
        jie ifX         # If so, jump to code for X
        jmp ifY         # Else jump to code for Y
        # Branch according to current tape symbol
ifX:    cmp rsym A      # Check if current symbol equals A
        jie ifXA        # If so, go to (X, A) transition
        jmp ifXB        # Else jump to (X, B) transition
ifY:    cmp rsym A      # Check if current symbol equals A
        jie ifYA        # If so, go to (Y, A) transition
        jmp ifYB        # Else jump to (Y, B) transition
        # Transition function
ifXA:   mov rstt Y      # Update current state to Y
        mov rsym B      # Stage new symbol B
        mem rsym        # Write B to tape
        prv             # Shift tape right
        jmp loop        # Repeat
ifXB:   mov rstt X      # Update current state to X
        mov rsym A      # Stage new symbol A
        mem rsym        # Write A to tape
        nxt             # Shift tape left
        jmp loop        # Repeat
ifYA:   mov rstt X      # Update current state to X
        mov rsym B      # Stage new symbol B
        mem rsym        # Write B to tape
        nxt             # Shift tape left
        jmp loop        # Repeat
ifYB:   mov rstt Y      # Update current state to Y
        mov rsym A      # Stage new symbol A
        mem rsym        # Write A to tape
        prv             # Shift tape right
        jmp loop        # Repeat
```

Fig. F.18. An NVM assembly program simulating an example Turing machine.

weights

$$W = [W^{\mathrm{gh,gh}}, W^{\mathrm{gh,co}}, W^{\mathrm{gh,opc}}, W^{\mathrm{gh,op_1}}, W^{\mathrm{gh,op_2}}]$$

that makes this true for every pair of respective columns in X and Y. In other words, we wish to solve $Y = \sigma_{\mathrm{gh}}(WX)$, or equivalently $\sigma_{\mathrm{gh}}^{-1}(Y) = WX$, for W.

However, this may not always be possible, for essentially the same reason that single-layer perceptrons are not universal function approximators. In particular, we found in practice that $\sigma_{\mathrm{gh}}^{-1}(Y)$ was full rank but X was not, and hence there was no linear operator W that would solve the requisite equation. To resolve this issue without adding another hidden region, we instead introduced a "hidden time-step", allowing each FSM transition to occur over two iterations of network dynamics instead of one. In this case, the equation to solve is $Y = \sigma_{\mathrm{gh}}(W^{\mathrm{gh,gh}}\sigma_{\mathrm{gh}}(WX))$. This can be done in a non-iterative fashion with standard linear algebraic techniques, by constructing a random matrix Z with the same low rank row space as X, and simultaneously solving

$$Z = \begin{bmatrix} W^{\mathrm{gh,gh}} & W^{\mathrm{gh,co}} & W^{\mathrm{gh,opc}} & W^{\mathrm{gh,op_1}} \end{bmatrix} X$$
$$\sigma_{\mathrm{gh}}^{-1}(Y) = W^{\mathrm{gh,gh}}\sigma_{\mathrm{gh}}(Z)$$

for the weights. In neural terms, Z represents the net synaptic input to gh before the hidden time-step. We found in practice that for a randomly generated low-rank Z, the hidden time-step activity $\sigma_{\mathrm{gh}}(Z)$ will generally become full rank, thereby enabling a solution to the linear system.

We encode the output mapping $\gamma : S \to G$ similarly: Let s' be the state encoded by the jth column of Y, and let $\mathbf{v}^{\mathrm{go}}[\gamma(s')]$ denote the requisite gating output for that state of the gating FSM. Then we can construct the matrix Y^{go}, whose jth column is $\mathbf{v}^{\mathrm{go}}[\gamma(s')]$.

```
start:    jie lab2  # does not jump since co is initialized to false
          sub lab6  # calls sub-routine at line 6
lab2:     exit      # these four lines are never reached
lab3:     jmp lab3
          mem r1
          ret
lab6:     sub lab6  # recursively calls itself
          prv       # these remaining lines are never reached
          jmp lab11
          nxt
          mem r2
lab11:    ret
```

Fig. G.19. An example of a randomly generated NVM assembly program, for an NVM instance with three registers (r0, r1, and r2).

Finally, we can solve the linear equations

$$\sigma_{\text{go}}^{-1}(Y^{\text{go}}) = W^{\text{go,gh}}\sigma_{\text{gh}}(Z)$$

for $W^{\text{go,gh}}$. Since the columns of Z are the "hidden time-steps" immediately before s' is reached, this ensures that gh contains the pattern for s' at the same time-step that go contains the gating pattern for $\gamma(s')$. Since we use the Heaviside function for σ_{go}, its inverse is not strictly defined. However, using the sign function in place of σ_{go}^{-1} was found to work in practice.

The hidden time-steps described above are omitted in Tables 2, D.5, and D.6 to avoid clutter. The default gating pattern during hidden time-steps for all layers other than gh is the saturation dynamics ($d_{\text{q}} = s_{\text{q,r}} = 0$).

Since the weights associated with the gating FSM are found using a linear solver, this "learning rule" is not local. However, it is much faster and more reproducible than iterative methods like gradient descent. Moreover, the instruction set and therefore FSM is independent of any particular program, so the FSM weights need only be "learned" once when an NVM instance is first constructed. Subsequently, specific programs can be assembled and executed using only the local learning rules described in the main text.

Appendix F. Simulating turing machines with NVM programs

Let $\langle Q, q_0, \Gamma, F, \delta\rangle$ be any Turing machine, where Q is the set of machine states, q_0 is the initial state, Γ is the set of tape symbols, $F \subseteq Q$ is the set of accepting states, and $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{\text{L}, \text{R}\}$ is the transition function (Hopcroft & Ullman, 1979). δ maps the current state and tape symbol to a new state, a new symbol to write to the tape, and a direction d to shift the tape, either left ($d =$L) or right ($d =$R).

To simulate the Turing machine with an NVM instance, we use the following strategy. First, we construct the NVM instance with two registers: rstt, for storing the current Turing machine state, and rsym, for storing the current tape symbol. Next, we use heap memory to represent the tape, using read/write head increments (nxt instruction) to shift the tape left and decrements (prv instruction) to shift it right. Finally, we use a program in program memory to repeatedly execute the transition function δ, using a series of conditional branches within an outer loop.

Pseudocode for a procedure which writes this NVM program is shown in Fig. F.17. First it initializes the machine state by moving q_0 into the rstt register. Next, it starts the loop that repeatedly executes δ to run the Turing machine. Each iteration begins by loading the current tape symbol into rsym from the current heap memory address (rem). Next, the current state is compared against each possible state $q \in Q$ one at a time. Once a match is found, a conditional jump (jie) redirects the NVM to the relevant code with label "if$q$". If q is an accepting state, the code for q simply halts the NVM (exit). Otherwise, the current symbol is compared against each possible symbol $\gamma \in \Gamma$. Once a match is found, another conditional jump redirects the NVM to code for the (q, γ) transition, labeled with "if$q\gamma$". Based on the transition output $(q', \gamma', d) = \delta(q, \gamma)$, the program moves the new state q' into rstt, the symbol γ' into rsym, writes γ' to the tape (mem), shifts the tape in the direction d by incrementing or decrementing the heap memory address, and finally makes an unconditional jump (jmp) to repeat the process again, starting from the loop label.

Fig. F.18 shows an NVM program generated in this manner for a small two-state, two-symbol Turing machine with no accepting states. To make the program more concise, we used unconditional jumps for the "else" part of each conditional branch.

Appendix G. Random program generation

Random programs of a given length were created by randomly selecting the opcode and operands for each line, and then post-processing to ensure well-defined program semantics. For example, any randomly created label used as an operand for a jump or sub-routine opcode would also be randomly assigned to another line of the program, so that each jump and sub-routine call would actually have a valid target. Fig. G.19 shows an example of a random program generated in this way.

References

Abdelbar, A. M., Andrews, E. A., & Wunsch II, D. C. (2003). Abductive reasoning with recurrent neural networks. *Neural Networks*, *16*(5–6), 665–673.

Amit, D. J., Gutfreund, H., & Sompolinsky, H. (1985). Storing infinite numbers of patterns in a spin-glass model of neural networks. *Physical Review Letters*, *55*(14), 1530.

Bechara, A., Damasio, H., Damasio, A. R., & Lee, G. P. (1999). Different contributions of the human amygdala and ventromedial prefrontal cortex to decision-making. *Journal of Neuroscience*, *19*(13), 5473–5481.

Bettcher, B. M., Libon, D. J., Kaplan, E., Swenson, R., & Penney, D. L. (2011). Digit symbol substitution test. In *Encyclopedia of clinical neuropsychology* (pp. 849–853). Springer.

Bošnjak, M., Rocktäschel, T., Naradowsky, J., & Riedel, S. (2017). Programming with a differentiable Forth interpreter. In *International conference on machine learning* (pp. 547–556).

Botvinick, M. M., Cohen, J. D., & Carter, C. S. (2004). Conflict monitoring and anterior cingulate cortex: an update. *Trends in Cognitive Sciences*, *8*(12), 539–546.

Bunel, R. R., Desmaison, A., Mudigonda, P. K., Kohli, P., & Torr, P. (2016). Adaptive neural compilation. In *Advances in neural information processing systems* (pp. 1444–1452).

Dehaene, S., & Changeux, J.-P. (1997). A hierarchical neuronal network for planning behavior. *Proceedings of the National Academy of Sciences*, *94*(24), 13293–13298.

Devlin, J., Bunel, R. R., Singh, R., Hausknecht, M., & Kohli, P. (2017). Neural program meta-induction. In *Advances in neural information processing systems* (pp. 2077–2085).

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A.-r., & Kohli, P. (2017). RobustFill: Neural program learning under noisy I/O. In *International conference on machine learning* (pp. 990–998).

Eichenbaum, H. (2000). A cortical–hippocampal system for declarative memory. *Nature Reviews Neuroscience*, *1*(1), 41.

Engel, T. A., & Wang, X.-J. (2011). Same or different? A neural circuit mechanism of similarity-based pattern match decision making. *Journal of Neuroscience*, *31*(19), 6982–6996.

Foster, D. J., & Wilson, M. A. (2006). Reverse replay of behavioural sequences in hippocampal place cells during the awake state. *Nature*, *440*(7084), 680.

Goodale, M. A., & Milner, A. D. (1992). Separate visual pathways for perception and action. *Trends in Neurosciences*, *15*(1), 20–25.

Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., et al. (2016). Hybrid computing using a neural network with dynamic external memory. *Nature*, *538*(7626), 471.

Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., & Schmidhuber, J. (2017). LSTM: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, *28*(10), 2222–2232.

Gruau, F., Ratajszczak, J.-Y., & Wiber, G. (1995). A neural compiler. *Theoretical Computer Science*, *141*(1–2), 1–52.

Haber, S. N. (2003). The primate basal ganglia: parallel and integrative networks. *Journal of Chemical Neuroanatomy*, *26*(4), 317–330.

Hopcroft, J. E., & Ullman, J. D. (1979). *Introduction to automata theory, languages and computation*. Adison-Wesley.

Hoshino, O., Usuba, N., Kashimori, Y., & Kambara, T. (1997). Role of itinerancy among attractors as dynamical map in distributed coding scheme. *Neural Networks*, *10*(8), 1375–1390.

ISO/IEC 14977:1996 E (1996). *Extended Backus-Naur form*. Geneva, CH: Standard: International Organization for Standardization.

Lara, A. H., & Wallis, J. D. (2015). The role of prefrontal cortex in working memory: A mini review. *Frontiers in Systems Neuroscience*, *9*, 173.

Mehaffey, W. H., Doiron, B., Maler, L., & Turner, R. W. (2005). Deterministic multiplicative gain control with active dendrites. *Journal of Neuroscience*, *25*(43), 9968–9977.

Nee, D. E., & D'Esposito, M. (2016). The representational basis of working memory. *Behavioral Neuroscience of Learning and Memory*, 213.

Neelakantan, A., Le, Q. V., & Sutskever, I. (2016). Neural programmer: Inducing latent programs with gradient descent. In *International conference on learning representations*.

Neto, J. P., Siegelmann, H. T., & Costa, J. (2003). Symbolic processing in neural networks. *Journal of the Brazilian Computer Society*, *8*(3), 58–70.

O'Reilly, R. C. (2010). The what and how of prefrontal cortical organization. *Trends in Neurosciences*, *33*(8), 355–361.

Plate, T. A. (1995). Holographic reduced representations. *IEEE Transactions on Neural Networks*, *6*(3), 623–641.

Pollack, J. B. (1987). *On connectionist models of natural language processing* (Ph.D. Thesis), Urbana: University of Illinois.

Reed, S., & De Freitas, N. (2016). Neural programmer-interpreters. In *International conference on learning representations*.

Reggia, J. A., Huang, D.-W., & Katz, G. (2017). Exploring the computational explanatory gap. *Philosophies*, *2*(1), 5.

Rocktäschel, T., & Riedel, S. (2017). End-to-end differentiable proving. In *Advances in neural information processing systems* (pp. 3791–3803).

Rosen, S. (1969). Electronic computers: A historical survey. *ACM Computing Surveys (CSUR)*, *1*(1), 7–36.

Salinas, E., & Sejnowski, T. J. (2001). Gain modulation in the central nervous system: where behavior, neurophysiology, and computation meet. *The Neuroscientist*, *7*(5), 430–440.

Schiess, M., Urbanczik, R., & Senn, W. (2016). Somato-dendritic synaptic plasticity and error-backpropagation in active dendrites. *PLoS Computational Biology*, *12*(2), e1004638.

Shin, Y., & Ghosh, J. (1991). The pi-sigma network: An efficient higher-order neural network for pattern classification and function approximation. In *International joint conference on neural networks* (pp. 13–18). IEEE.

Siegelmann, H. T. (1994). Neural programming language. In *AAAI* (pp. 877–882).

Siegelmann, H. T., & Sontag, E. D. (1991). Turing computability with neural nets. *Applied Mathematics Letters*, *4*(6), 77–80.

Sylvester, J. J. (1867). Thoughts on inverse orthogonal matrices, simultaneous sign successions, and tessellated pavements in two or more colours, with applications to Newton's rule, ornamental tile-work, and the theory of numbers. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, *34*(232), 461–475.

Sylvester, J., & Reggia, J. (2016). Engineering neural systems for high-level problem solving. *Neural Networks*, *79*, 37–52.