# Training Neural Machines with Trace-Based Supervision

**Matthew B. Mirman** [1] **Dimitar Dimitrov** [1] **Pavle Djordjevic** [1] **Timon Gehr** [1] **Martin Vechev** [1]

## Abstract

We investigate the effectiveness of trace-based supervision methods for training existing neural abstract machines. To define the class of neural machines amenable to trace-based supervision, we introduce the concept of a differential neural computational machine ($\partial$NCM) and show that several existing architectures (NTMs, NRAMs) can be described as $\partial$NCMs. We performed a detailed experimental evaluation with NTM and NRAM machines, showing that additional supervision on the interpretable portions of these architectures leads to better convergence and generalization capabilities of the learning phase than standard training, in both noise-free and noisy scenarios.

## 1. Introduction

Recently, there has been substantial interest in neural machines that can induce programs from examples (Graves et al., 2014; Reed & de Freitas, 2016; Graves et al., 2016; Zhang et al., 2015; Zaremba & Sutskever, 2015; Kaiser & Sutskever, 2015; Gaunt et al., 2016; Vinyals et al., 2015; Feser et al., 2016; 2015; Frankle et al., 2016; Kurach et al., 2016; Bošnjak et al., 2017). While significant progress has been made towards learning interesting algorithms (Graves et al., 2016), ensuring these machines converge to the desired solution during training is challenging. Interestingly however, even though they differ architecturally, most of these machines rely on components (e.g., neural memory) that are more interpretable than typical neural networks (e.g., an LSTM). This allows one to provide additional supervision and help bias the learning towards the desired solution.

In this work, we investigate whether (and by how much) additional amounts of supervision provided to these interpretable components during training can improve learning. The particular type of supervision we consider is *partial trace supervision*, providing more detailed information, beyond input-output examples, during learning. To help describe the type of architectures our results apply to, we introduce the notion of a differential neural computational machine ($\partial$NCM), a formalism which allows to cleanly characterize the neural machines that can benefit from *any* amount of extra trace-based supervision. We show that common existing architectures such as Neural Turing Machines (NTMs) and Neural Random Access Machines (NRAMs) can be described as $\partial$NCMs. We also explain why other machines such as the Neural Program Interpreter (NPI) (Reed & de Freitas, 2016) or its recent extensions (e.g., the Neural Program Lattice, Li et al. (2017)) cannot be instantiated as a $\partial$NCM and are thus restricted to require large (and potentially prohibitive) amounts of supervision.

We performed an extensive evaluation investigating how partial trace information affects training of both NTMs and NRAMs. Overall, our experimental results indicate that additional supervision can substantially improve convergence while leading to better generalization and interpretability, under both noise-free and noisy supervision scenarios. Interestingly, we also show that on a more complex task, the NRAM architecture trained with additional supervision can generalize better than more recent architectures such as the DNGPU (Freivalds & Liepins, 2017) which are difficult to interpret and thus, provide additional supervision to.

The work closest to ours is the Differentiable Forth of Bošnjak et al. (2017), which also investigates the effects of extra supervision. The main difference between the two works is the kind of supervision provided. In Bošnjak et al. (2017), the teacher provides a sketch of the program to be learned. The machine then needs to fill in the holes of the sketch. In our work, there is no program sketch: the teacher hints (part of) the operations that the machine should execute in specific executions (i.e., hints on specific execution traces). The machine then has to learn these operations. We believe that both training methods are interesting and worth investigating in the context of neural machines. We also remark that both methods have already been investigated in the context of traditional (non-neural) programming, for example by Lau et al. (2003) and Solar-Lezama et al. (2006).

[1]Department of Computer Science, ETH Zurich, Switzerland. Correspondence to: Matthew B. Mirman <matthew.mirman@inf.ethz.ch>, Martin Vechev <martin.vechev@inf.ethz.ch>.

*Proceedings of the 35th International Conference on Machine Learning*, Stockholm, Sweden, PMLR 80, 2018. Copyright 2018 by the author(s).

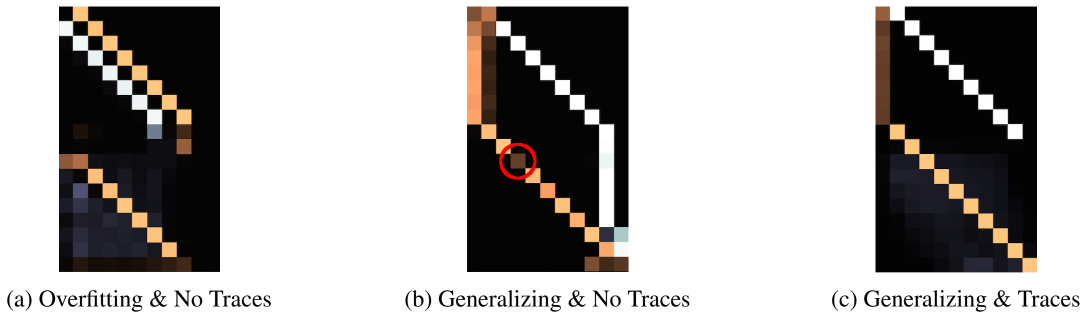


*Figure 1.* Traces of locations accessed by read/write heads for the `Flip3rd` task in three different training setups. The $y$-axis represents time (descending); the $x$-axis represents head locations. First, two NTMs are trained without partial trace information. White represents the distribution of the write head; orange the distribution of the read head; (b) and (c) generalize and are more interpretable than (a); (c) was trained using partial trace information and is more interpretable than (b).

To intuitively illustrate our approach, consider the task of training an NTM to flip the third bit in a bit stream (called `Flip3rd`) – such tasks have been extensively studied in the area of program synthesis with noise (Raychev et al., 2016) and without (Jha et al., 2010). An example input-output pair for this task could be $[0, 1, 0, 0] \rightarrow [0, 1, 1, 0]$. Given a set of such examples, our goal is to train an NTM that solves this task. Figure 1c shows an example NTM run (on an input that is longer than those shown during training) that demonstrates good generalization and has an understandable trace. The $y$-axis indicates time (descending), the $x$-axis indicates the accessed memory location, the white squares represent the write head of the NTM, and the orange squares represent the read head. As we can see, the model writes the input sequence to the tape and then reads from the tape in the same order. However, in the absence of richer supervision, the NTM (and other neural architectures) can easily overfit to the training set – an example of an overfitting NTM is shown in Figure 1a. Here, the traces are chaotic and difficult to interpret. Further, even if the NTM generalizes, it can do so with erratic reads and writes – an example is shown in Figure 1b. Here, the NTM learns to read from the third bit (circled) with a smaller weight than from other locations, and also reads and writes erratically near the end of the sequence. This model is less interpretable than the one in Figure 1c because it is unclear why the NTM reads the third bit with less weight, and whether this helps flip it.

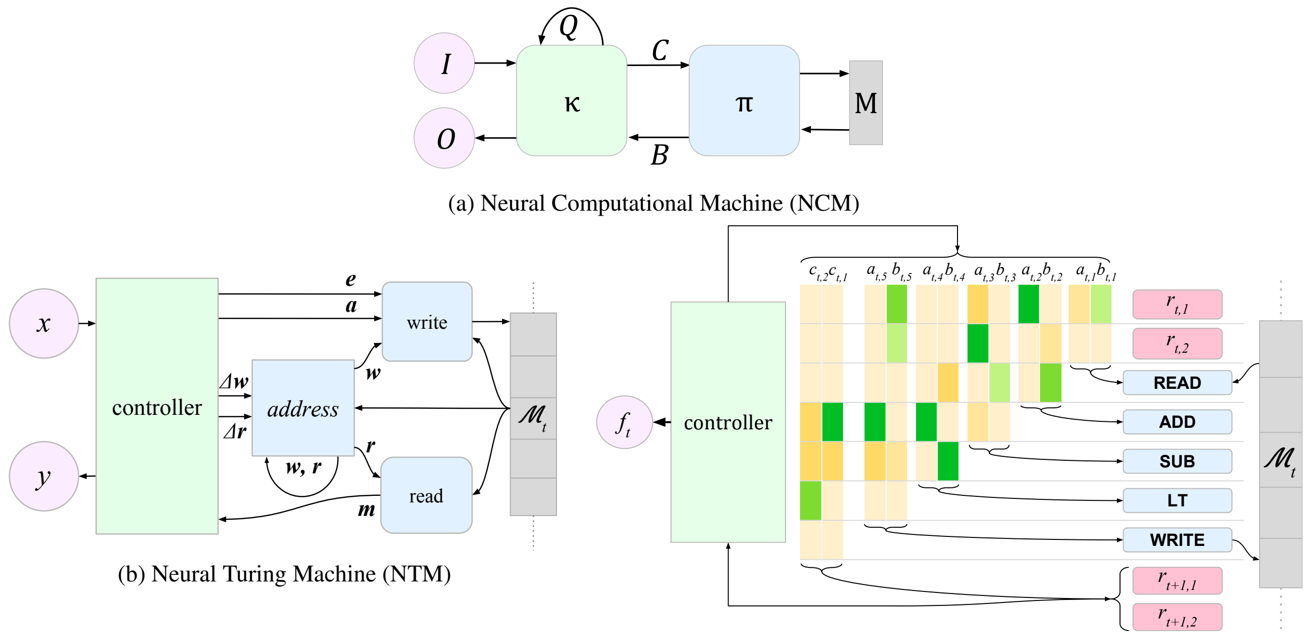
In this work we develop systematic ways to guide the training of a neural machine (e.g., NTM, NRAM) towards more interpretable behavior [1]. For instance, for `Flip3rd`, providing partial trace information on the NTM's read heads for 10% of input-output examples is sufficient to bias learning towards the NTM shown in Figure 1c 100% of the time.

[1]All of the code, tasks and experiments are available at: `https://github.com/eth-sri/ncm`

## 2. Neural Computational Machines

We now define the notion of a *neural computational machine* (NCM) which we use throughout to formalize our ideas. We introduce NCMs for two reasons. First, they allow one to capture the class of neural machines to which our supervision methods apply and explain why other machines fall outside this class. That is, the NCM abstraction clearly delineates end-to-end differentiable architectures such as NTM (Graves et al., 2014) and NRAM (Kurach et al., 2016), which can train with little to no trace supervision, from architectures that are not end-to-end differentiable, such as NPI (Reed & de Freitas, 2016), and hence require a certain minimum amount of trace information. In Section 3, we show how to phrase two existing neural architectures (NTM and NRAM) as an NCM. Second, it enables us to cleanly define a general, abstract loss function that captures different kinds of trace-based supervision at the NCM level and makes it clear which components of that loss function need to be instantiated for the particular NCM architecture (e.g., NTM, NRAM). In Section 4, we show how to specify the general trace-based loss function at the NCM level and how to instantiate key components of that loss function for NTMs and NRAMs.

We note that while NCMs do allow for a clean formalization of the ideas and help overall understanding, they are not meant to be a complete neural model where one specifies the additional supervision only at the NCM level without any awareness of the details of the underlying architecture. Indeed, the user still has to be aware of the interpretable portions of the particular architecture (e.g., NRAM, NTM) and have some intuition for how a solution to the task at hand would use those components. We do believe however that this is a reasonable trade-off to investigate: a little bit of extra knowledge can enable substantially better results.



(a) Neural Computational Machine (NCM)

(b) Neural Turing Machine (NTM)

(c) Neural Random Access Machine (NRAM)

*Figure 2.* (a) depicts the generic NCM structure, (b) is a high-level overview of an NTM, and (c) is a high-level overview of an NRAM. For NRAM, the controller outputs a circuit, which in this case contains the modules READ, ADD, SUB, LT, and WRITE. The controller encodes the two inputs to the modules as probability vectors, $a$ and $b$, over the possible choices. Darker colors capture higher probability, e.g., dark green is the most likely choice. The only input to the controller are the registers $r_1$ and $r_2$.

## 2.1. NCM design

The design of an NCM mimics classic computational machines with a controller and a memory. Concretely, an NCM is a triple of functions: a processor, a controller, and a loss. We define these components below.

**Processor** The processor $\pi_\theta \colon C \times M \to B \times M$ performs a pre-defined set of commands $C$, which might involve manipulating memories in $M$. The commands may produce additional feedback in $B$. Also, the processor's operation may depend on parameters $\theta$.

**Controller** Controller $\kappa_\theta \colon B \times Q \times I \to C \times Q \times O$ decides which operations the machine performs at each step. It receives external inputs from $I$ and returns external outputs in $O$. It can also receive feedback $B$ from the processor and command it to do certain operations (e.g., memory read). The decisions the controller takes may depend on its internal state in $Q$. The controller can also depend on additional parameters $\theta$. For instance, if the controller is a neural network, then $\theta$ gives the network's weights.

**Loss Function** The loss function $L_E \colon \mathit{Trace} \times E \to \mathbb{R}$ indicates how close an execution trace $\tau \in \mathit{Trace}$ of the machine (defined below) is to a behavior from a set $E$. Plugging the behavior of the machine into the loss, we get:

$$l(\theta; x, e) = L_E(\tau_\theta(x), e) \tag{1}$$

Averaging $l$ over a set of input-output examples $(x, e)$ gives us the loss surface that is being minimized during training. As standard, we require $l$ to be continuous and piecewise differentiable w.r.t $w$ for any given example $(x, e)$.

**Traces** The execution of the machine begins with an input sequence $x = \{x_t\}_1^n$ and initial values of the controller state $q_0$, memory $\mathcal{M}_0$, and processor feedback $b_0$. At each time step $t = 1 \ldots n$, controller and processor take turns:

$$\begin{aligned} (c_t, q_t, y_t) &= \kappa(b_{t-1}, q_{t-1}, x_t) \\ (b_t, \mathcal{M}_t) &= \pi(c_t, \mathcal{M}_{t-1}) \end{aligned} \tag{2}$$

To avoid clutter, we keep the parameters $\theta$ implicit. A *trace* $\tau(x, b_0, \mathcal{M}_0, q_0) = \{(c_t, b_t, q_t, y_t, \mathcal{M}_t)\}_1^n$ keeps track of the values of these quantities at every time step. We occasionally write $\tau_C, \tau_B, \ldots$ for the trace projected onto one of its components $c, b, \ldots$.

**∂NCMs** Note that the differentiability conditions we impose on $l$ do not imply the functions $\pi$, $\kappa$ and $L_E$ are continuous or differentiable w.r.t all parameters. They indeed can be highly discontinuous as in NCMs like Memory Networks (Weston et al., 2014), or as in Neural Programmer-Interpreters (Reed & de Freitas, 2016). In order to ensure the differentiability of $l$, these architectures train with strong supervision: in this case the loss function $L_E$ requires examples $e \in E$ that provide a value for each discontinuous quantity in the traces. In contrast, what we call *differentiable neural computational machines* ($\partial$NCM), have $\kappa$, $\pi$ and $L_E$ continuous and piecewise differentiable. Then, there is no need to specify corresponding values in the examples, and so we can train with as much trace information as available.

## 3. NTMs and NRAMs as NCMs

We now describe NTMs and NRAMs as $\partial$NCMs.

**NTM as $\partial$NCM** An NTM (Graves et al., 2014) (Figure 2b) has access to a memory $\mathcal{M} \in \mathbb{R}^{c\times n}$ of $c$ cells of $n$ real numbers each. We suppose the machine has one read head and one write head, whose addresses are, respectively, the probability vectors $r, w \in [0,1]^{\{1\ldots c\}}$. At every time step, the read head computes the expected value $m \in \mathbb{R}^n$ of a random cell at index $i \sim r$. This value together with the current input are fed into a controller neural network, which then decides on the command. The command consists of several pieces: (i) a decision on what fraction $e \in \mathbb{R}^n$ to erase, (ii) how much $a \in \mathbb{R}^n$ to add to the cells underneath the write head, and (iii) an indication of the head movement with two probability vectors $\Delta r, \Delta w \in [0,1]^{\{-1,0,+1\}}$. Finally, the controller produces the current output value. In terms of NCMs, NTM variables fall into the following classes:

| I/O | State | Communication |
|---|---|---|
| $x \in I$ | $q \in Q$ | $(e, a, \Delta r, \Delta w) \in C$ |
| $y \in O$ | $(r, w, \mathcal{M}) \in M$ | $m \in B$ |

Each of these variables change over time according to certain equations (found in the supplementary ). The processor $\pi$ and the controller $\kappa$ functions for each time step satisfy:

$$((e_t, a_t, \Delta r_t, \Delta w_t),\ \ q_t,\ \ y_t) = \kappa(m_t, q_{t-1}, x_t) \quad (3)$$

$$(m_{t+1},\ (r_t, w_t, \mathcal{M}_t)) = \pi((e_t, a_t, \Delta r_t, \Delta w_t),\ (r_{t-1}, w_{t-1}, \mathcal{M}_{t-1})). \quad (4)$$

That is, the processor computes the new values of the read and write heads by convolving their old values with $\Delta r$ and $\Delta w$, updates the memory by using $e$, $a$, the write head $w$ and the previous value of the memory, and also produces the output read value by using the memory and the read head $r$. Formal details are provided in the supplementary material.

The standard loss function $L_E$ for the NTM includes a term, such as cross-entropy or $L_2$ distance, for the machine output at every time step. Each of these compare the machine output to the respective values contained in examples $e \in E$.

**NRAM as $\partial$NCM** A Neural Random Access Machine (NRAM) (Kurach et al., 2016) is a neural machine designed for ease of pointer access. The NRAM has a memory and registers that store probability vectors on a fixed set $\{1\ldots c\}$. The memory is, therefore, a matrix $\mathcal{M} \in \mathbb{R}^{c\times c}$ of $c$ cells, while the register file is a matrix $r \in \mathbb{R}^{n\times c}$ of $n$ registers. The controller receives no external inputs, but it takes as feedback the probability of 0 for each register. It also produces no external output, except for a halting probability $f \in [0,1]$ produced at every time step. The "output" of the run is considered to be the final memory.

At each time step the NRAM can execute several modules from a fixed sequence. Each module implements a simple integer operation/memory manipulation lifted to probability vectors. For example, addition lifts to convolution, while memory access is as in the NTM. At every time step, the controller organizes the sequence of modules into a circuit, which is then executed. The circuit is encoded by a pair of probability distributions per module, see Figure 2c. These distributions specify respectively which previous modules or registers will provide a given module's first/second arguments. The distributions are stacked in the matrices $a$ and $b$. A similar matrix $c$ is responsible for specifying what values should be written to the registers at the end of the time step. The NCM instantiation of an NRAM is:

| I/O | State | Communication |
|---|---|---|
| $\{1\} = I$ | $q_t \in Q$ | $(a_t, b_t, c_t) \in C$ |
| $f_t \in O$ | $(r_t, \mathcal{M}_t) \in M$ | $r_{t,-,0} \in B$ |

The equations for these quantities are found in the supplementary material. The processor $\pi$ and the controller $\kappa$ are:

$$\begin{aligned}((a_t, b_t, c_t), q_t, f_t) &= \kappa(r_{(t-1),-,0}, q_{t-1}, 1)\\ (r_{t,-,0}, (r_t, \mathcal{M}_t)) &= \pi((a_t, b_t, c_t),\ (r_{t-1}, \mathcal{M}_{t-1})).\end{aligned} \quad (5)$$

The NRAM loss is an expectation with respect to the distribution $p$ of the halting time step, as determined by the halting probabilities $f_t$ (see the supplementary material). Each example is a vector $e \in \{1\ldots c\}^c$ holding the desired value of each memory cell. The loss considers the negative log likelihood that the $i$-th memory cell at time step $t$ equals the value $e_i$ from the example, independently for each $i$:

$$L_E(\tau, e) = -\sum_{t<|\tau|} p_t \sum_{i<c} \log(\mathcal{M}_{t,i,e_i}). \quad (6)$$

## 4. Subtrace Supervision of NCMs

Incorporating supervision during training of an NCM-instantiated machine can be helpful with: (i) *convergence*: additional bias may steer the minimization of the loss function $L_E$ away from local minima that do not correspond to good solutions, (ii) *interpretability*: the bias can be useful in guiding the machine towards learning a model that is more intuitive/explainable to a user (especially if the user already has an intuition how components of the model should be used), and (iii) *generalization*: the bias can help with finding solutions which minimize the loss on more difficult examples than those seen during training.

The way we provide additional supervision to NCMs is, for example, by encoding specific commands issued to the processor into extra loss terms. Let us illustrate how we can bias the learning with an NTM. Consider the task of copying the first half of an input sequence $\{x_t\}_1^{2l}$ into the second half of the machine's output $\{y_t\}_1^{2l}$, where the last input $x_l$ from the first half is a special value indicating that the first half ended. Starting with both heads at position 1, the most direct solution is to consecutively store the input to the tape during the first half of the execution, and then read out the stored values during the second half. In such a solution, we expect the write/read head positions to be:

$$e_w(t) = \begin{cases} \text{one-hot}(t) & \text{if } t = 1 \dots l \\ \text{one-hot}(l) & \text{if } t \geq l+1 \end{cases} \tag{7}$$

$$e_r(t) = \begin{cases} \text{one-hot}(1) & \text{if } t = 1 \dots l \\ \text{one-hot}(t-l) & \text{if } t \geq l+1 \end{cases} \tag{8}$$

Here one-hot$(i)$ denotes the probability vector whose $i$-th component equals 1 (the distribution concentrated on $i$).

To incorporate this information into the training, we add loss terms that measure the cross-entropy ($H$) between $e_w(t)$ and $w_t$ as well as between $e_r(t)$ and $r_t$. Importantly, we need not add terms for every time-step, but instead we can consider only the corner cases where heads change direction:

$$\sum_{t=1,l+1,2l} H(e_w(t), w_t) + H(e_r(t), r_t).$$

## 4.1. Generic Subtrace Loss for NCMs

We now describe the general shape of the extra loss terms for arbitrary NCMs. Since, typically, we can interpret only the memory and the processor in terms of well-understood operations, we will consider loss terms only for the memory state and the communication flow between the controller and the processor. We leave the controller's hidden state unconstrained – this also permits us to use the same training procedure with different controllers.

The generic loss is expressed with four loss functions for the different components of an NCM trace:

$$\begin{aligned} L_C &: C \times E_C \to \mathbb{R} & L_B &: B \times E_B \to \mathbb{R} \\ L_O &: O \times E_O \to \mathbb{R} & L_M &: M \times E_M \to \mathbb{R} \end{aligned} \tag{9}$$

For each $\alpha \in \{C, B, O, M\}$, we give *hints* $(t, v, \mu)$ that indicate a time step $t$ at which the hint applies, an example $v \in E_\alpha$ for the relevant component, and a weight $\mu \in \mathbb{R}$ of the hint. The weight is included to account for hints having a different importance at different time-steps, but also to express our confidence in the hint, e.g., hints coming from noisy sources would get less weight.

A *subtrace* is a collection $\sigma$ of hints used for a particular input-output example $e$. We call it a subtrace because it typically contains hints for a proper subset of the states traced by the NCM. With $\sigma_\alpha$ we denote the hints in $\sigma$ designated for a loss component $\alpha$. The net *hint loss* is the sum of all losses per hint, weighted and normalized:

$$L_\sigma(\tau, \sigma) = \frac{\sum_{\alpha \in \{C,B,O,M\}} \sum_{(t,v,\mu) \in \sigma_\alpha} \mu L_\alpha(\tau_{\alpha,t}, v)}{\sum_{\alpha \in \{C,B,O,M\}} \sum_{(t,v,\mu) \in \sigma_\alpha} \mu}$$

The total loss for a given input-output example and subtrace equals the scaled hint loss $L_\sigma$ plus the original loss $L_E$, where the scaling of the hint loss is a hyperparameter $\lambda$:

$$L(\tau, (\sigma, e)) = \lambda L_\sigma(\tau, \sigma) + L_E(\tau, e). \tag{10}$$

## 4.2. Subtraces for NTM

For NTMs, we allow hints on the output $y$, heads $r$ and $w$, and the tape $\mathcal{M}$. We include extra loss terms for the memory state only (i.e., except $L_M$, all other loss terms are zero). For addresses, $L_M$ is defined as:

$$\begin{aligned} L_M((r_t, w_t, \mathcal{M}_t), (\texttt{wr}, v)) &= H(v, w_t) \\ L_M((r_t, w_t, \mathcal{M}_t), (\texttt{rd}, v)) &= H(v, r_t) \end{aligned} \tag{11}$$

Unlike addresses, values on the tape are interpreted according to an encoding internal to the controller (which emerges only during training). Forcing the controller to use a specific encoding for the tape, as we do with NTM output, can have a negative effect on training (in our experiments, training diverged consistently). To remedy this, we do not apply the loss to the tape directly, but to a decoded version of a cell on the tape. While a decoder might find multiple representations and overfit, we found that it forced just enough consistency to improve the convergence rate. The decoder itself is an auxiliary network $\phi$ trained together with the NTM, which takes a single cell from memory as input. The output of the decoder is compared against the expected value $v$ which should be in that cell:

$$L_M((-, -, \mathcal{M}_t), (\texttt{tape}, i, v)) = H(\phi(\mathcal{M}_{t,i}), v). \tag{12}$$

For all subtraces we provide in our experiments with NTMs, the hints have the same unit weight.

## 4.3. Subtraces for NRAM

For NRAMs, we hint which connections should be present in the circuit the controller constructs at each step, including

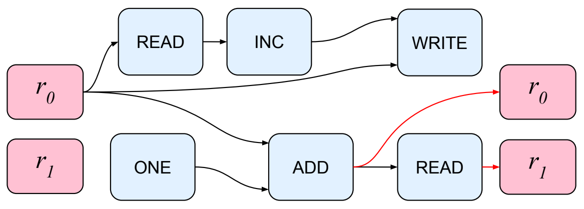


*Figure 3.* A circuit for the task of adding one to all memory cells. The arrows for register updates are shown in red. Technically, modules take two arguments, but some ignore an argument, such as INC or READ. For them, we show only the relevant arrows.

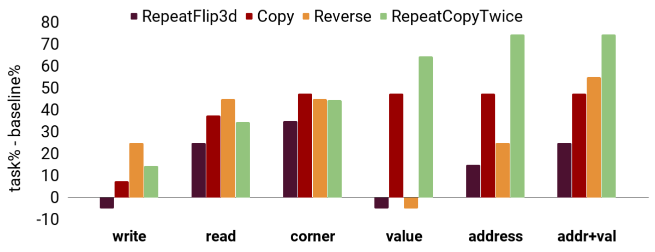


*Figure 4.* Fraction of training instances that generalized out of 10 supervised runs per task for the NTM compared to unsupervised baseline. We provide a subtrace 100% of the time and use $\lambda = 1$.

the ones for register updates. An example circuit is shown in Figure 3. In terms of an NCM, this amounts to providing loss for commands and no loss for anything else. We set the loss to the negative log likelihood of the controller choosing specific connections revealed in the hint:

$$\begin{aligned} L_C((a_t, b_t, c_t), (\texttt{module}, m, i, j)) &= \\ -\log(a_{t,m,i}) - \log(b_{t,m,j}) \\ L_C((a_t, b_t, c_t), (\texttt{register}, r, i)) &= \\ -\log(c_{t,r,i}) \end{aligned} \tag{13}$$

Here, $m$ is the module, $i$ is the index of the first argument (a module or a register); similarly for $j$ for the second argument. In our experiments, we observed that assigning higher weight to hints at earlier timesteps is crucial for convergence of the training process. For a hint at time-step $t$, we use the weight $\mu = (t+1)^{-2}$. A possible reason for why this helps is that the machine's behavior at later time-steps is highly dependent on its behavior at early time-steps. Thus, the machine cannot reach a later behavior that is right before it fixes its early behavior. Unless the behavior is correct early on, the loss feedback from later time-steps will be mostly noise, masking the feedback from early time-steps.

**Other Architectures** The NCM can be instantiated to architectures as diverse as a common LSTM network or End-To-End Differentiable Memory Networks. Any program inducing neural network with at least partially interpretable intermediate states for which the dataset contains additional hints could be considered a good candidate for application of this abstraction.

## 5. Experimental Evaluation

We evaluated the effects of different kinds of trace-based supervision on training NTM and NRAM architectures. The main questions we investigated are: (i) Does trace supervision help convergence, interpretability, and generalization? (ii) How much supervision is needed to train successfully? We focused on algorithmic tasks (mostly from the NTM and NRAM papers) since these architectures were designed to solve such tasks, but also because the tasks are unambiguous and supervision can be provided more easily.

**Effects on NTMs** We measured the frequency of succeeding to train an NTM that generalizes strongly. We consider a model to generalize strongly if whenever we train with examples of size $\leq n$, the trained NTM achieves perfect accuracy on all tests of size $\leq 1.5n$, and at least 90% accuracy on all tests of size $\leq 2n$.

Figure 4 reports the average improvement over a baseline trained with input/output examples. We experimented with four tasks and various types of hints (see supplementary). The hint types are: (1) *read/write* give the respective head addresses for all time steps; (2) *address* combines read and write; (3) *corner* reveals the heads' addresses but only when the heads change direction; (4) *value* gives a value for a single cell; (5) *addr+val* combines address and value hints.

Trace supervision improves generalization in all except 3 cases. Interestingly, among the most challenging tasks, a small but non-negligible amount of extra supervision (i.e., corner hints) lead to greatest improvement. For example, for `RepeatFlip3d` the baseline generalizes only 5% of the time, while corner hints achieve 8-fold improvement, reaching 40%. Another task with an even larger ratio is `RepeatCopyTwice`, where success increases from 15.5% to 100%. Both results are illustrated in the supplementary.

In addition to this experiment, we performed an extensive evaluation of different setups, varying the global $\lambda$ parameter of the loss (10), and providing hints for just a *fraction* of the examples. Full results are in the supplementary; here we provide those for `Flip3rd` in Figure 5. The results reveal that the efficacy of our method heavily depends on these two parameters. The best results in this case are for the read/corner type of hints for $\frac{1}{2}$ or $\frac{1}{10}$ of the examples, with $\lambda \in \{0.1, 1\}$. Interestingly, while the type of supervision which works best varies from task to task, for each task there exists a trace which can be provided only 1% of the time and still greatly improve the performance over the baseline. This suggests that a small amount of extra supervision can

| **density** | | 100 | 100 | 100 | 100 | 50 | 50 | 50 | 50 | 10 | 10 | 10 | 10 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **λ** | | 1 | 0.3 | 0.1 | 0.03 | 1 | 0.3 | 0.1 | 0.03 | 1 | 0.3 | 0.1 | 0.03 | 1 | 0.3 | 0.1 | 0.03 |
| baseline | **45** | | | | | | | | | | | | | | | | |
| addr+val | | **30** | **50** | **50** | **50** | **50** | **50** | **70** | **80** | **60** | **80** | **40** | **40** | **40** | **60** | **40** | **10** |
| address | | **0** | **60** | **40** | **40** | **20** | **70** | **80** | **90** | **90** | **70** | **60** | **50** | **50** | **50** | **60** | **60** |
| value | | **60** | **50** | **40** | **60** | **80** | **70** | **40** | **10** | **50** | **10** | **40** | **70** | **50** | **30** | **40** | **60** |
| read | | **40** | **60** | **70** | **40** | **30** | **80** | **90** | **90** | **100** | **70** | **80** | **50** | **30** | **50** | **60** | **30** |
| write | | **0** | **30** | **50** | **20** | **30** | **40** | **60** | **20** | **40** | **40** | **40** | **40** | **20** | **50** | **70** | **50** |
| corner | | **50** | **70** | **80** | **80** | **40** | **40** | **90** | **70** | **80** | **70** | **60** | **40** | **50** | **60** | **60** | **40** |

*Figure 5.* The number of training runs (out of 100) that generalized for `Flip3rd`. The different supervision types are shown vertically, while the proportion of examples that receive extra subtrace supervision (*density*) and the extra loss term weight ($\lambda$) are shown horizontally.

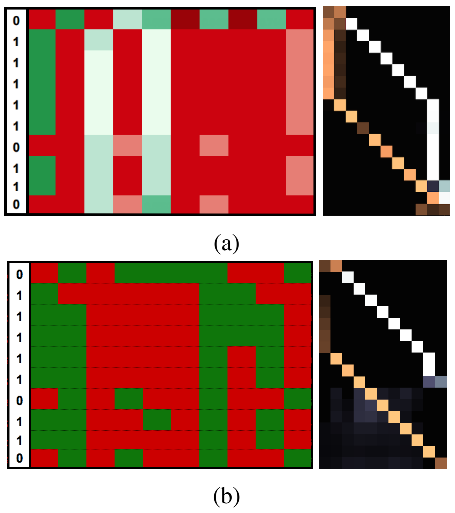


*Figure 6.* Execution traces of two NTMs trained on `Flip3rd` until generalization. First is baseline (no trace supervision); second is trained with corner hints. Time flows top to bottom. The first pane from every pair shows the value written to tape; second shows head locations. Figures show that a little extra supervision helps the NTM write sharper 0–1 values and have more focused heads.

improve performance significantly, but the kind of supervision may differ. This, of course, raises the question of what the best type and amount of hints are for a given task.

Finally, we observed that in all cases where training with trace supervision converged, it successfully learned the head movements/tape values we had intended. This shows that trace supervision can bias the architecture towards more interpretable behaviors. In those cases, the NTM learned consistently sharper head positions/tape values than the baseline, as Figure 6 shows for `Flip3rd`.

**Effect of subtraces on NRAMs** For the NRAM we measured the average test accuracy over the whole set of trained models. Unlike the original NRAM paper (Kurach et al., 2016), we used the more conservative 0–1 loss for testing: the output gets a score 0 even if it is incorrect in just one position. We did that because we consider the metric in the NRAM paper to be too subjective: it reports the fraction of

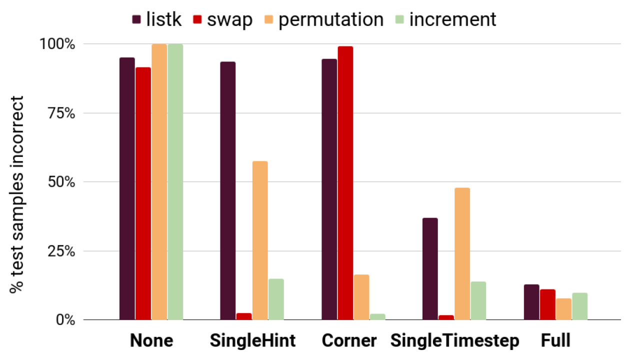


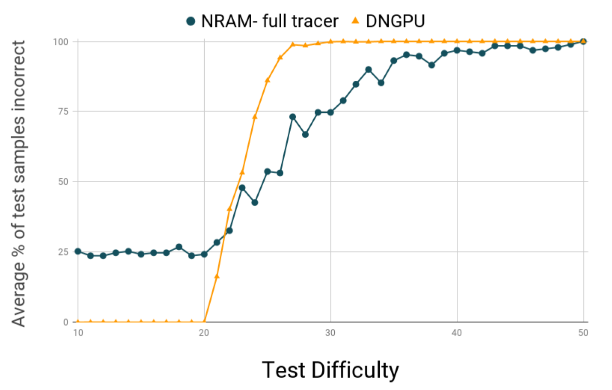
*Figure 7.* Error rates for NRAM showing the average number of errors after training had completed for NRAM with different kinds of supervision.

incorrect output positions over a manually specified region.

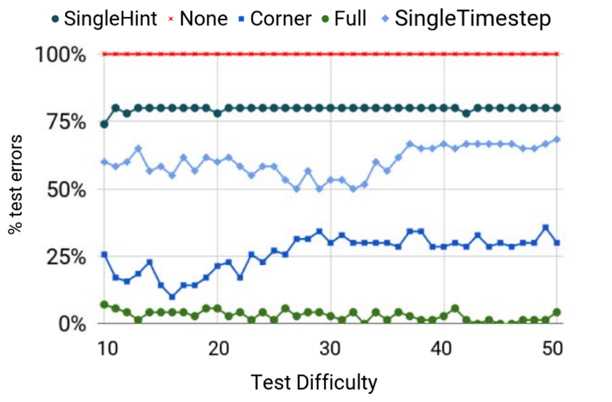
We show the results for four different tasks / types of hints in Figure 7. The types of hints are: (1) *single hint* reveals a random part of the circuit at one random time step; (2) *corner* shows the entire circuit at the first and the last time steps; (3) *single time step* gives the full circuit at a random time step; (4) *full* provides the full circuit at all time steps. We show examples in the supplementary.

As shown in the figure, the NRAM baseline performed poorly. We attribute this mainly to the difficulty of training an NRAM that generalizes well. For example, Neelakantan et al. (2015) reported that only 5 to 22 out of 100 training runs succeed over three possible hyperparameter choices. Thus, it is difficult to find hyperparameters that make the baseline model generalize well.

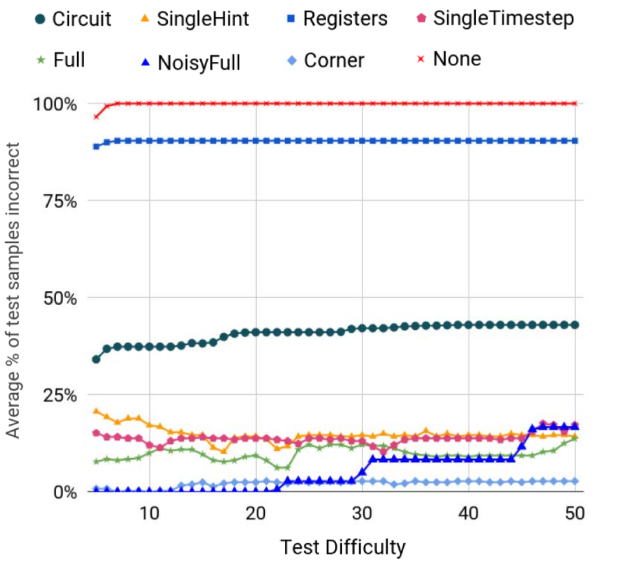
Interestingly, however, as seen in the figure, adding extra supervision dramatically improved the baseline. We trained with up to 5000 examples. Here, the full supervision led to good results consistently, unlike the NTM case, where corner hints were usually better. Also, even though consistent, full supervision did not always lead to the best improvement: for the easier tasks, `Swap` and `Increment` less supervision was, in fact, more effective. This again shows that training is sensitive to the amount and type of supervision.



(a) Average generalization of DNGPU (Freivalds & Liepins, 2017) and NRAM using full hints for `Merge`.



(b) `Permute` with noise for NRAM: the distribution of errors to problem length (one character of noise in 10% of samples).



(c) `Increment` with NRAM: comparing average generalization to sequence length (including with noisy hints).

*Figure 8.* Results comparing (a) NRAM vs. DNGPU (with supervision), (b) supervision with noisy training for NRAM, and (c) generalization for NRAM with different supervision types.

**NRAM with Supervision vs. DNGPU** We also compared to the DNGPU (Freivalds & Liepins, 2017), a state-of-the-art architecture, which can often perform better than the baseline NTM and NRAM architectures on certain tasks, but which is also much more difficult to interpret, and provide hints for. Given the gap between the baselines of DNGPU and NRAM, we were curious whether NRAM with full supervision can get closer to the DNGPU accuracy. Towards that, we compared the NRAM and the DNGPU on the challenging `Merge` task, where two sorted lists must be merged into one sorted list. A maximum of 10000 samples were used for the DNGPU and 5000 for the NRAM. The DNGPU was run out of the box from the code supplied by the authors. 20 runs were averaged for the DNGPU and 38 runs for the NRAM. Again, without extra supervision, we did not observe the NRAM to generalize. Interestingly, as can be seen in Figure 8a, the DNGPU performs well on tests within the training length (20), but on larger lengths its accuracy drops rapidly and the NRAM with full supervision outperforms the DNGPU (its accuracy drops less rapidly).

**Robustness to Noise** Finally, we investigated the effect of subtrace supervision when training with noise. Towards that, we performed two experiments: one where noise was introduced in the training examples and one where noise was introduced in the extra hints themselves.

For the noisy examples, we corrupted a single character in 10% of the training examples for the `Permute` task. The effect is shown in Figure 8b. Without any subtrace hints, training did not converge within the time limit, whereas with just corner hints, the test error was around 25%.

For the noisy subtrace hints, we took full supervision and corrupted a single hint in 20% of the traces for the `Increment` task. As seen in Figure 8c, for small enough sequence lengths, `NoisyFull` training actually obtained better performance than full supervision without noise.

## 6. Conclusion

We investigated the effects of additional trace-based supervision when training neural abstract machines. The basic idea was to provide this supervision (called partial trace information) over the interpretable components of the machine and to thus more effectively guide the learning towards the desired solution. We introduced the ∂NCM architecture in order to precisely capture the neural abstract machines to which trace-based supervision applies. We showed how to formulate partial trace information as abstract loss functions, how to instantiate common neural architectures such as NTMs and NRAMs as ∂NCMs and concretize the ∂NCM loss functions. Our experimental results indicate that partial trace information is effective in biasing the learning of both NTMs and NRAMs towards better convergence, generalization and interpretability of the resulting models.

## References

Bošnjak, M., Rocktäschel, T., Naradowsky, J., and Riedel, S. Programming with a differentiable forth interpreter. In *International Conference on Machine Learning (ICML)*, 2017.

Feser, J. K., Chaudhuri, S., and Dillig, I. Synthesizing data structure transformations from input-output examples. In *Proceedings of Programming Language Design and Implementation (PLDI)*, 2015.

Feser, J. K., Brockschmidt, M., Gaunt, A. L., and Tarlow, D. Differentiable functional program interpreters. *arXiv preprint arXiv:1611.01988*, 2016.

Frankle, J., Osera, P.-M., Walker, D., and Zdancewic, S. Example-directed synthesis: a type-theoretic interpretation. In *Symposium on Principles of Programming Languages (POPL)*, 2016.

Freivalds, K. and Liepins, R. Improving the neural gpu architecture for algorithm learning. *arXiv preprint arXiv:1702.08727*, 2017.

Gaunt, A. L., Brockschmidt, M., Kushman, N., and Tarlow, D. Lifelong perceptual programming by example. *arXiv preprint arXiv:1611.02109*, 2016.

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

Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S. G., Grefenstette, E., Ramalho, T., Agapiou, J., Badia, A. P., Hermann, K. M., Zwols, Y., Ostrovski, G., Cain, A., King, H., Summerfield, C., Blunsom, P., Kavukcuoglu, K., and Hassabis, D. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626), Oct 2016.

Jha, S., Gulwani, S., Seshia, S. A., and Tiwari, A. Oracle-guided component-based program synthesis. In *International Conference on Software Engineering (ICSE)*, volume 1, 2010.

Kaiser, Ł. and Sutskever, I. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

Kurach, K., Andrychowicz, M., and Sutskever, I. Neural random-access machines. *ERCIM News*, 2016(107), 2016.

Lau, T., Domingos, P., and Weld, D. S. Learning programs from traces using version space algebra. In *International Conference on Knowledge Capture (KCAP)*, 2003.

Li, C., Tarlow, D., Gaunt, A. L., Brockschmidt, M., and Kushman, N. Neural program lattices. In *5th International Conference on Learning Representations (ICLR)*, 2017.

Neelakantan, A., Vilnis, L., Le, Q. V., Sutskever, I., Kaiser, L., Kurach, K., and Martens, J. Adding gradient noise improves learning for very deep networks. *arXiv preprint arXiv:1511.06807*, 2015.

Raychev, V., Bielik, P., Vechev, M. T., and Krause, A. Learning programs from noisy data. In *Symposium on Principles of Programming Languages (POPL)*, 2016.

Reed, S. and de Freitas, N. Neural programmer-interpreters. In *4th International Conference on Learning Representations (ICLR)*, 2016.

Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., and Saraswat, V. Combinatorial sketching for finite programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

Vinyals, O., Fortunato, M., and Jaitly, N. Pointer networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2015.

Weston, J., Chopra, S., and Bordes, A. Memory networks. *arXiv preprint arXiv:1410.3916*, 2014.

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

Zhang, W., Yu, Y., and Zhou, B. Structured memory for neural turing machines. *arXiv preprint arXiv:1510.03931*, 2015.

## A. NTM Equations

The controller for the NTM consists of the networks $\varphi$, $\psi_y$, $\psi_e$, $\psi_a$, $\chi_r$, $\chi_w$, which operate on the variables:

$$\begin{array}{llllll} x \text{ – in} & q \text{ – controller state} & r \text{ – read address} & \Delta r \text{ – change in } r & e \text{ – erase} & \mathcal{M} \text{ – tape} \\ y \text{ – out} & m \text{ – read value} & w \text{ – write address} & \Delta w \text{ – change in } w & a \text{ – add} & \end{array} \tag{14}$$

The equations that describe NTM executions are:

$$\begin{aligned} \Delta r_t &= \chi_r(q_t) & r_t &= \mathit{address}(\Delta r_t, r_{t-1}, \mathcal{M}_{t-1}) \\ \Delta w_t &= \chi_w(q_t) & w_t &= \mathit{address}(\Delta w_t, w_{t-1}, \mathcal{M}_{t-1}) \\ y_t &= \psi_y(q_t) & m_t &= r_t M_{t-1} \\ e_t &= \psi_e(q_t) & \mathcal{M}_t &= \mathcal{M}_{t-1} - (w_t \otimes e_t) \odot \mathcal{M}_{t-1} + w_t \otimes a_t \\ a_t &= \psi_a(q_t) & q_t &= \varphi(x_t, q_{t-1}, m_t). \end{aligned} \tag{15}$$

## B. NRAM Equations

The controller of the NRAM consists of the networks $\varphi$, $\psi_a$, $\psi_b$, $\psi_c$, $\psi_f$, which operate on the variables:

$$\begin{array}{llll} a \text{ – lhs circuit} & b \text{ – rhs circuit} & c \text{ – register inputs} & o \text{ – module outputs} \\ r \text{ – register state} & \mathcal{M} \text{ – memory tape} & h \text{ – controller state} & f \text{ – stop probability.} \end{array} \tag{16}$$

The equations that describe the NRAM execution are:

$$\begin{aligned} a_t &= \text{softmax}(\psi_a(q_t)) & A_{t,i} &= (r_1, \ldots, r_R, o_0, \ldots, o_{i-1})^T a_{t,i} & &\forall i < M \\ b_t &= \text{softmax}(\psi_b(q_t)) & B_{t,i} &= (r_1, \ldots, r_R, o_0, \ldots, o_{i-1})^T b_{t,i} & &\forall i < M \\ c_t &= \text{softmax}(\psi_c(q_t)) & r_{t,i} &= (r_1, \ldots, r_R, o_1, \ldots, o_Q)^T c_{t,i} & &\forall i < R \\ f_t &= \psi_f(q_t) & o_{t,i,k} &= \sum_{0 \le a,b < M} A_{t,i,a} B_{t,i,b} [m_i(a,b) = k] & &\forall i \notin \{\rho, \omega\}, k < M \\ q_t &= \varphi(q_{t-1}, r_{t,-,0}) & o_{t,\rho} &= \mathcal{M}_t A_{t,\rho} \\ & & \mathcal{M}_t &= (J - A_{t,\omega}) J^T \cdot \mathcal{M}_{t-1} + A_{t,\omega} B_{t,\omega}^T \end{aligned} \tag{17}$$

$$p_t = f_t \prod_{i<t} (1 - f_i) \qquad p_T = 1 - \sum_{i<T} p_i \tag{18}$$

## C. Setup for NTM

For all our NTM experiments we use a densely connected feed-forward controller. There are two architectural differences from the original NTM (Graves et al., 2014) that helped our baseline performance: (i) the feed-forward controller, the erase and the add gates use `tanh` activation, and (ii) the output layer uses `softmax`. In the original architecture these are all logistic sigmoids. For the newly introduced tape decoder (active only during training) we used two alternative implementations: a `tanh-softmax` network, and a single affine transformation. We tested the NTM's learning ability on five different tasks for sequence manipulation, two of which have not been previously investigated in this domain. These tasks can be found in Appendix E.

We performed experiments using several combinations of losses as summarized in Appendix F. The observed training performance per task is shown in Appendix 10, with rows corresponding to the different loss setups. The *corner* setup differs from the *address* setup in that the example subtraces were defined only for a few important corner cases. For example in `RepeatCopyTwice`, the write head was provided once at the beginning of the input sequence, and once at the end. Similarly, the read head was revealed at the beginning and at the end of every output repetition. In all other setups we provide full subtraces (defined for all time steps).

The supervision amount can be tuned by adjusting the $\lambda$ weight from Equation 10. Further, we can also control the fraction of examples which get extra subtrace supervision (the *density* row in Figure 10). The performance metric we use is the percentage of runs that do generalize after 100k iterations for the given task and supervision type. By *generalize* we mean that the NTM has perfect accuracy on all testing examples up to $1.5\times$ the size of the max training length, and also perfect accuracy on 90% of the testing examples up to $2\times$ the maximum training length.

We used a feed-forwad controller with $2 \times 50$ units, except for `RepeatCopyTwice`, which uses $2 \times 100$ units. For training we used the Adam optimizer (**?**), a learning rate of $10^{-3}$ for all tasks except `RepeatFlip3d` and `Flip3rd` which use $5 \cdot 10^{-4}$. The lengths of the training sequences for the first four tasks are from 1 to 5, whereas the generalization of the model was tested with sequences of lengths up to 20. For `Flip3rd` and `RepeatFlip3d`, the training sequence length was up to 16, whereas the testing sequences have maximum length of 32.

## D. Setup for NRAM

Like in the NTM, we use a densely connected two layer feed forward controller for our experiments, and use ReLU as the activation function. We make no modifications to the original architecture, and use noise with parameter $\eta = 0.3$ as suggested by Neelakantan et al. (2015), and curriculum learning as described by **?**. We stop training once we get to a difficulty specified by the task, and increase the difficulty once 0 errors were found on a new testing batch of 10 samples. Each training iteration trains with 50 examples of the currently randomly sampled difficulty. Regardless of whether the model had converged, training is stopped after 5000 samples were used. Such a low number is used to replicate the potential conditions under which such a model might be used. As with the NTM, the Adam optimizer was used. The specific tasks we use are described in Appendix G, and the specific kinds of supervision we give are described in Appendix H. The $\lambda$ we used here was 40. The system was implemented using PyTorch.

## E. NTM Tasks

Every input sequence ends with a special delimiter $x_E$ not occurring elsewhere in the sequence

**`Copy`** – The input consists of generic elements, $x_1 \dots x_n x_E$. The desired output is $x_1 \dots x_n x_E$.

**`RepeatCopyTwice`** – The input is again a sequence of generic elements, $x_1 \dots x_n x_E$. The desired output is the input copied twice $x_1 \dots x_n x_1 \dots x_n x_E$. Placing the delimiter only at the end of the output ensures that the machine learns to keep track of the number of copies. Otherwise, it could simply learn to cycle through the tape reproducing the given input indefinitely. We kept the number of repetitions fixed in order to increase baseline task performance for the benefit of comparison.

**`DyckWords`** – The input is a sequence of open and closed parentheses, $x_1 \dots x_n x_E$. The desired output is a sequence of bits $y_1 \dots y_n x_E$ such that $y_i = 1$ iff the prefix $x_1 \dots x_i$ is a balanced string of parentheses (a Dyck word). Both positive and negative examples were given.

**`Flip3rd`** – The input is a sequence of bits, $x_1 x_2 x_3 \dots x_n x_E$. The desired output is the same sequence of bits but with the 3rd bit flipped: $x_1 x_2 \bar{x}_3 \dots x_n x_E$. Such a task with a specific index to be updated (e.g., 3rd) still requires handling data dependence on the contents of the index (unlike say the `Copy` task).

**`RepeatFlip3d`** – The input is a sequence of bits, $x_1 x_2 x_3 x_4 x_5 x_5 \dots x_E$. The desired output is the same sequence of bits but with *every* 3rd bit flipped: $x_1 x_2 \bar{x}_3 x_4 x_5 \bar{x}_6 \dots x_E$.

## F. NTM Subtraces

$$\begin{tikzcd}
 & \textit{addr+val} & & \\
\textit{value} \arrow[ur] & & \textit{address/corner} \arrow[ul] & \\
 & \textit{write} \arrow[ur] & & \textit{read} \arrow[ul]
\end{tikzcd}$$

*Figure 9.* A heirarchy of supervision types (but not quantities) for NTMs.

**value traces** provide hints for the memory at every timestep as explained in Equation 12.

**read** – provides a hint for the address of the read head at every timestep.

**write** – provides a hint for the address of the write head at every timestep.

**address** – provides hints for the address of both the read and the write head at every timestep.

**addr+val** – provides value, read and write hints for every timestep.

**corner** – provides hints for the address of both the read and the write head at every "important" timestep - we decided what important means here depends on which task we are referring to. In general, we consider the first and last timesteps important, and also any timestep where a head should change direction. For example, in `RepeatCopyTwice` for an example of size $n$ with $e$ repeats, we'd provide the heads at timesteps $0, n, 2n, 3n \dots, en$.

## G. NRAM Tasks

Below we describe all tasks we experimented with. We predominantly picked tasks that the NRAM is known to have trouble generalizing on. We did not introduce any new tasks, and more detailed descriptions of these tasks can be found in Kurach et al. (2016).

**`Swap`** – Provided two numbers, $a$ and $b$ and an array $p$, swap $p[a]$ and $p[b]$. All elements but that in the last memory cell are not zero.

**`Increment`** – Given an array $p$, return the array with one added to each element. All elements but that in the last cell for the input are not zero. Elements can be zero in the output.

**`Permute`** – Given two arrays $p$ and $q$ return a new array $s$ such that $s[i] = q[p[i]]$. The arrays $p$ and $q$ are preceded by a pointer, $a$, to array $q$. The output is expected to be $a, s[0] \ldots, s[n], q[0], q[n]$.

**`ListK`** – Given a linked list in array form, and an index $k$ return the value at node $k$.

**`Merge`** – given arrays $p$ and $q$, and three pointers $a, b, c$ to array $p$, $q$, and the output sequence (given as zeros initially), place the sorted combination of $p$ and $q$ into the output location.

The following table describes the specific NRAM instantiation used for each task. The *default* sequence (def) is the one described by Kurach et al. (2016). The number of timesteps is usually dependent on the length of the problem instance, $M$ (equivalently the word size or difficulty), and in the case of `ListK`, was given with respect to the argument $k$. The difficulty (D) was simply the length of the sequence used. Training began by providing sequences of length *Start D* and ended when curriculum learning reached sequences of length *End D*.

| Task | No. Regs | Module Sequence | Timesteps | Learn Rate | Start D | End D |
|---|---|---|---|---|---|---|
| `Swap` | 5 | def | 7 | 0.01 | 6 | 10 |
| `Increment` | 2 | def + R | $M+2$ | 0.01 | 4 | 10 |
| `Permute` | 4 | R + def + R + W | $M+3$ | 0.05 | 7 | 12 |
| `ListK` | 6 | def | $k+5$ | 0.05 | 9 | 16 |
| `Merge` | 8 | def + def + def | $M+3$ | 0.05 | 13 | 16 |

## H. NRAM Subtraces

For each of the tasks listed Appendix G, we hand coded a complete circuit for every module and every timestep we would provide.

The following subtrace types describe how we provide hints based on this circuit.

**`None`** – provides no hints.

**`Full`** – provides the entire circuit.

**`SingleHint`** – provides a random hint at a random timestep.

**`SingleTimestep`** – provides the entire circuit at a random timestep.

**`Corner`** – provides the entire circuit at the first and last timesteps.

**`Registers`** – provides hints for the registers at every timestep.

**`Modules`** – provides hints for the modules at every timestep.

## I. NTM Results

**Which Details to Reveal for NTM?** The first dimension listed in the rows of the tables of Figure 10 controls the execution details revealed in a subtrace. We use subtraces showing either the *addresses* without the tape *values*, only the *read heads* or the *write heads*, or even weaker supervision in a few *corner* cases. In tasks `Copy` (Figure 10a), `RepeatCopyTwice` (Figure 10b) and `DyckWords` (Figure 10c), it is frequently the case that when the NTM generalizes without supervision, it converges to an algorithm which we are able to interpret. For them, we designed the *addr+val* traces to match this algorithm, and saw increases in generalization frequency of at least 45%. It can be concluded that when additionally provided supervision reflects the interpretable "natural" behavior of the NTM, the learning becomes significantly more robust to changes in initial weights. Additionally, for tasks `Flip3rd` (Figure 10d) and `RepeatFlip3d` (Figure 10e), both the baseline and other supervision types are outperformed by training with *read* supervision. It is also notable that *corner* supervision in `RepeatFlip3d` achieves highest improvement over the baseline, 60% over 5%. In essence, this means that providing only a small part of the trace can diminish the occurrence of local minima in the loss function.

**How Often to Reveal for NTM?** The second dimension controls the proportion of examples that receive extra subtrace supervision (the *density* columns in Figure 10). For `Flip3rd`, `RepeatCopyTwice` and `DyckWords` we observed that having only a small number of examples with extra supervision leads to models which are more robust to initial weight changes than the baseline, although not necessarily always as robust as providing supervision all the time.

A couple of interesting cases stand out. For `Flip3rd` with 10% *corner* subtraces and $\lambda = 1$, we find a surprisingly high rate of generalization. Providing *address* traces 10% of the time when training `RepeatCopyTwice` leads to better performance all the time. For `RepeatFlip3d`, *write* traces at 1% frequency and $\lambda = 0.1$ generalize 30% of the time vs. 5% for baseline.

While the type of trace which works best varies per task, for each task there exists a trace which can be provided only 1% of the time and still greatly improve the performance over the baseline. This suggests that a small amount of extra supervision can improve performance significantly, but the kind of supervision may differ. It is an interesting research question to find out how the task at hand relates to the optimal kind of supervision.

| density | | 100 | 100 | 100 | 100 | 50 | 50 | 50 | 50 | 10 | 10 | 10 | 10 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| λ | | 1 | 0.3 | 0.1 | 0.01 | 1 | 0.3 | 0.1 | 0.01 | 1 | 0.3 | 0.1 | 0.01 | 1 | 0.3 | 0.1 | 0.01 |
| baseline | 52.5 | | | | | | | | | | | | | | | | |
| addr+val | | 100 | 100 | 100 | 70 | 100 | 100 | 100 | 40 | 60 | 80 | 40 | 30 | 30 | 50 | 60 | 10 |
| address | | 100 | 100 | 100 | 50 | 90 | 100 | 90 | 30 | 80 | 90 | 70 | 30 | 50 | 30 | 40 | 50 |
| value | | 100 | 100 | 70 | 40 | 80 | 20 | 40 | 10 | 10 | 20 | 40 | 30 | 60 | 40 | 20 | 10 |
| read | | 90 | 80 | 70 | 50 | 60 | 20 | 50 | 20 | 40 | 40 | 60 | 20 | 70 | 30 | 40 | 10 |
| write | | 60 | 70 | 80 | 60 | 80 | 80 | 40 | 40 | 50 | 70 | 50 | 40 | 50 | 60 | 50 | 40 |
| corner | | 100 | 100 | 100 | 50 | 100 | 90 | 60 | 70 | 70 | 20 | 50 | 30 | 50 | 60 | 20 | 30 |

(a) `Copy`

| density | | 100 | 100 | 100 | 50 | 50 | 50 | 10 | 10 | 10 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| λ | | 1 | 0.3 | 0.03 | 1 | 0.3 | 0.03 | 1 | 0.3 | 0.03 | 1 | 0.3 | 0.03 |
| baseline | 15.5 | | | | | | | | | | | | |
| addr+val | | 90 | 100 | 60 | 90 | 80 | 40 | 80 | 20 | 10 | 10 | 0 | 0 |
| address | | 90 | 90 | 90 | 100 | 100 | 40 | 100 | 60 | 0 | 0 | 20 | 30 |
| value | | 80 | 70 | 0 | 50 | 50 | 10 | 30 | 30 | 20 | 10 | 30 | 0 |
| read | | 50 | 30 | 30 | 20 | 60 | 30 | 20 | 60 | 10 | 10 | 10 | 0 |
| write | | 30 | 30 | 20 | 10 | 30 | 40 | 20 | 0 | 10 | 10 | 20 | 20 |
| corner | | 60 | 50 | 40 | 50 | 60 | 10 | 20 | 40 | 20 | 10 | 20 | 0 |

(b) `RepeatCopyTwice`

| density | | 100 | 100 | 100 | 100 | 50 | 50 | 50 | 50 | 10 | 10 | 10 | 10 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| λ | | 1 | 0.3 | 0.1 | 0.01 | 1 | 0.3 | 0.1 | 0.01 | 1 | 0.3 | 0.1 | 0.01 | 1 | 0.3 | 0.1 | 0.01 |
| baseline | 45 | | | | | | | | | | | | | | | | |
| address | | 70 | 90 | 60 | 80 | 90 | 90 | 90 | 50 | 80 | 50 | 90 | 80 | 100 | 80 | 70 | 70 |
| read | | 80 | 90 | 70 | 70 | 100 | 100 | 70 | 50 | 50 | 60 | 70 | 70 | 80 | 70 | 50 | 50 |
| corner | | 60 | 100 | 80 | 80 | 80 | 90 | 90 | 90 | 60 | 60 | 100 | 50 | 90 | 80 | 80 | 50 |

(c) `DyckWords`

| density | | 100 | 100 | 100 | 100 | 50 | 50 | 50 | 50 | 10 | 10 | 10 | 10 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| λ | | 1 | 0.3 | 0.1 | 0.03 | 1 | 0.3 | 0.1 | 0.03 | 1 | 0.3 | 0.1 | 0.03 | 1 | 0.3 | 0.1 | 0.03 |
| baseline | 45 | | | | | | | | | | | | | | | | |
| addr+val | | 30 | 50 | 50 | 50 | 50 | 50 | 70 | 80 | 60 | 80 | 40 | 40 | 40 | 60 | 40 | 10 |
| address | | 0 | 60 | 40 | 40 | 20 | 70 | 80 | 90 | 90 | 70 | 60 | 50 | 50 | 50 | 60 | 60 |
| value | | 60 | 50 | 40 | 60 | 80 | 70 | 40 | 10 | 50 | 10 | 40 | 70 | 50 | 30 | 40 | 60 |
| read | | 40 | 60 | 70 | 40 | 30 | 80 | 90 | 90 | 100 | 70 | 80 | 50 | 30 | 50 | 60 | 30 |
| write | | 0 | 30 | 50 | 20 | 30 | 40 | 60 | 20 | 40 | 40 | 40 | 40 | 20 | 50 | 70 | 50 |
| corner | | 50 | 70 | 80 | 80 | 40 | 40 | 90 | 70 | 80 | 70 | 60 | 40 | 50 | 60 | 60 | 40 |

(d) `Flip3rd`

| density | | 100 | 100 | 100 | 100 | 50 | 50 | 50 | 50 | 10 | 10 | 10 | 10 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| λ | | 1 | 0.5 | 0.1 | 0.03 | 1 | 0.5 | 0.1 | 0.03 | 1 | 0.5 | 0.1 | 0.03 | 1 | 0.5 | 0.1 | 0.03 |
| baseline | 5 | | | | | | | | | | | | | | | | |
| addr+val | | 30 | 20 | 20 | 40 | 30 | 30 | 10 | 10 | 40 | 10 | 0 | 20 | 10 | 10 | 0 | 10 |
| address | | 20 | 50 | 30 | 30 | 30 | 40 | 20 | 40 | 20 | 40 | 20 | 0 | 0 | 0 | 20 | 0 |
| value | | 0 | 0 | 20 | 20 | 0 | 0 | 0 | 0 | 10 | 10 | 10 | 0 | 0 | 0 | 0 | 0 |
| read | | 30 | 10 | 40 | 20 | 10 | 30 | 20 | 40 | 30 | 10 | 0 | 10 | 20 | 0 | 10 | 20 |
| write | | 0 | 0 | 0 | 10 | 0 | 0 | 0 | 0 | 10 | 10 | 0 | 0 | 20 | 20 | 30 | 0 |
| corner | | 40 | 40 | 60 | 20 | 50 | 30 | 10 | 30 | 10 | 10 | 0 | 0 | 0 | 10 | 0 | 10 |

(e) `RepeatFlip3d`

*Figure 10.* Baselines have generalization on over 40 different initial weights. Other tests use 10.

## J. NRAM Results

| Subtrace Type \Task | `Permute` | `Swap` | `Increment` | `ListK` | `Merge` | `PermuteNoise` |
|---|---|---|---|---|---|---|
| `None` | 36 | 44 | 58 | 41 | 13 | 5 |
| `SingleHint` | 24 | 38 | 28 | 22 | 12 | 5 |
| `Corner` | 29 | 23 | 36 | 22 | 9 | 7 |
| `SingleTimestep` | 21 | 52 | 29 | 28 | 12 | 6 |
| `Registers` | 48 | 58 | 73 | 54 | - | - |
| `Modules` | 48 | 58 | 107 | 54 | - | - |
| `Full` | 26 | 33 | 32 | 44 | 21 | 7 |
| `NoisyFull` | - | - | - | 36 | - | - |

*Figure 11.* The number of runs which completed for each task and subtrace type. The Data in the graphs below is taken by averaging the results of these runs.

| Subtrace Type \Task | `Permute` | `Swap` | `Increment` | `ListK` |
|---|---|---|---|---|
| `None` | 6290.29 | 5505.22 | 3500.13 | 5880.11 |
| `SingleHint` | 5565.22 | 3700.64 | 4318.20 | 6574.59 |
| `Corner` | 4468.85 | 6195.75 | 3199.86 | 6601.16 |
| `SingleTimestep` | 6259.05 | 2662.35 | 4042.18 | 5076.17 |
| `Registers` | 6618.12 | 5774.61 | 3839.18 | 6185.54 |
| `Modules` | 6523.16 | 5781.99 | 2335.99 | 6183.74 |
| `Full` | 4919.33 | 4110.14 | 3758.99 | 3216.01 |

*Figure 12.* The average time (in seconds) to finish training for each task and subtrace type. For most tasks it is clear that Full traces while introducing extra computations to individual timesteps, reduce the amount of time to finish training over not using supervision.

| Subtrace Type \Task | `ListK` | `Swap` | `Permute` | `Increment` | `Merge` | `PermuteNoise` |
|---|---|---|---|---|---|---|
| `None` | 95.08 | 91.52 | 99.97 | 99.91 | 99.96 | 99.99 |
| `SingleHint` | 93.61 | 2.41 | 57.55 | 14.86 | 100.0 | 56.90 |
| `Corner` | 94.47 | 99.09 | 16.40 | 2.14 | 100.0 | 20.79 |
| `SingleTimestep` | 36.91 | 1.75 | 47.79 | 13.77 | 100.0 | 33.60 |
| `Full` | 12.77 | 11.01 | 7.83 | 9.89 | 78.44 | 23.57 |
| `Registers` | 93.22 | 93.44 | 99.97 | 90.36 | - | - |
| `Modules` | 93.70 | 95.57 | 86.48 | 40.87 | - | - |

*Figure 13.* The average number of errors on the test set for each task and subtrace type once trained.

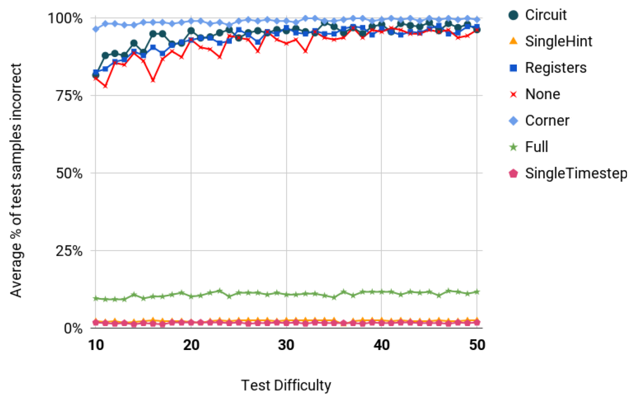


*Figure 14.* Comparing average generalization to sequence length for `Swap`

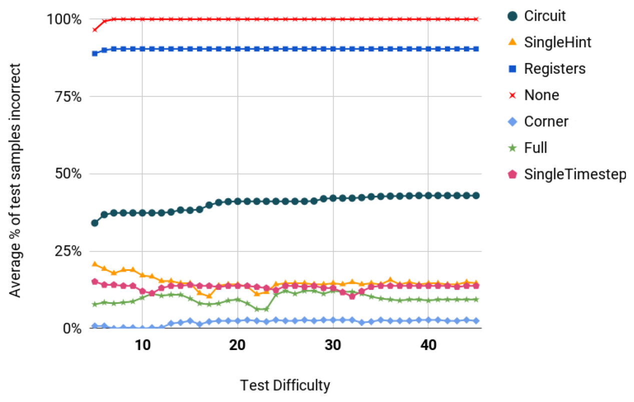


*Figure 15.* Comparing average generalization to sequence length for `Increment`

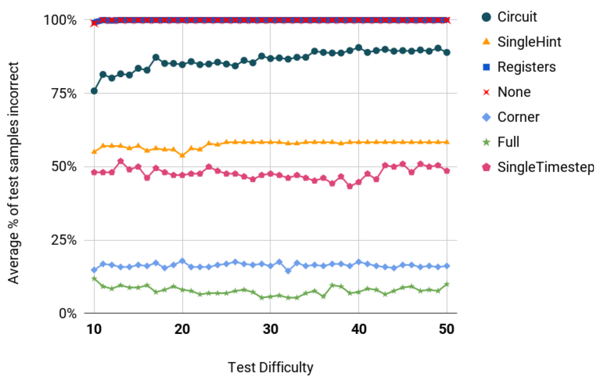


*Figure 16.* Comparing average generalization to sequence length for `Permute`

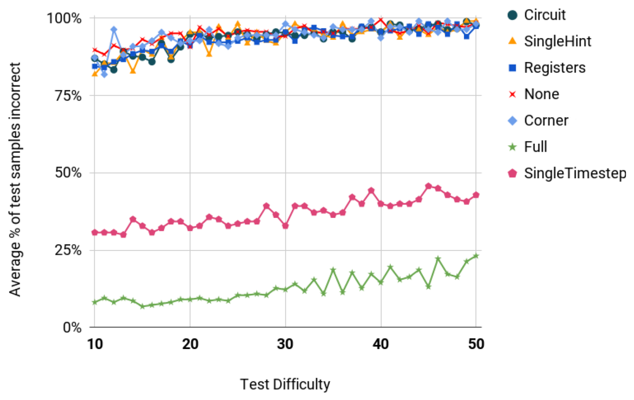


*Figure 17.* Comparing average generalization to sequence length for `ListK`

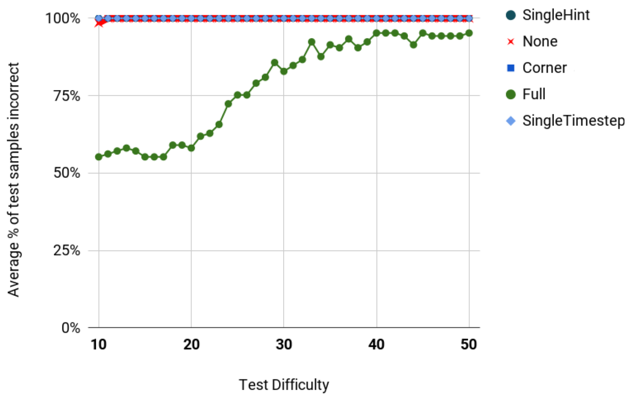


*Figure 18.* Comparing average generalization to sequence length for `Merge`

## K. Programming NRAMs

The NRAM is parametrized by one or more straight-line partial programs, i.e., programs with no branching and no loops, chosen by register states. The machine runs in a loop, repeatedly selecting the program for that register state then executing it. The programs are expressed in a simple single-assignment imperative language. Each program statement $i$ invokes one of the modules of the architecture and assigns the result of the invocation to a local variable $x_i$. That variable cannot be changed later. The final program statement is a parallel-assignment that modifies the machine registers $r_1 \dots r_k$. The values that appear in assignments/invocations can be: variables in scope, machine registers, or holes $?$. These values are not used directly during execution: the actual values needs to be supplied by the NRAM controller. The values are only used as hints for the controller during training, with the whole $?$ denoting no hint. We can describe the language in an EBNF-style grammar:

$$F_n ::= \text{modules of arity } n \tag{19}$$

$$V_i ::= r_1 \mid \cdots \mid r_k \mid ? \mid x_1 \mid \dots \mid x_{i-1} \tag{20}$$

$$S_i ::= x_i \leftarrow F_n(\underbrace{V_i, \dots, V_i}_{n}) \tag{21}$$

$$R_i ::= r_{j_1}, \dots, r_{j_n} \leftarrow \underbrace{V_i, \dots V_i}_{n} \tag{22}$$

$$P_1 ::= S_1 \quad P_i ::= P_{i-1}; S_i \tag{23}$$

$$P ::= P_1; R_1 \mid P_2; R_2 \mid \dots \tag{24}$$

An example program for the **Increment** task would be the following:

$x_1 \leftarrow 1;$
$x_2 \leftarrow \mathsf{READ}(r_1);$
$x_3 \leftarrow \mathsf{ADD}(x_2, x_1);$
$x_4 \leftarrow \mathsf{WRITE}(r_1, x_3);$
$x_5 \leftarrow \mathsf{ADD}(r_1, x_1);$
$r_1 \leftarrow x_5$

Here, the controller is encouraged to read the memory at the location stored in the first register $r_1$, add one to it, then store it back, and then increment the first register.

An alternative to the trace-based approach is to make the controller produce values only for the holes, and use directly the specified variable/register arguments. This way, only the unspecified parts of the program are learned. This is, for example, the approach taken by $\partial$Forth (**?**). There, programs are expressed in a suitably adapted variant of the Forth programming language, which is as expressive as the language discussed above, but less syntactically constrained.

The drawback of this alternative is that whenever an argument other than a whole is specified, one must also specify the time steps to which it applies in *all possible executions* and not just the training ones. That is why, typically, these values are specified either for all or for none of the time steps.

In the following examples, we will describe the register states using “0”, “!0” and “-” meaning respectively that a register has 0, that it contains anything but zero, or that it can contain anything.

## L. NRAM Permutation Program

For any register pattern.

```
$x_1 \leftarrow \mathsf{READ}(r_0)$;
$x_2 \leftarrow \mathsf{WRITE}(0, x_1)$;
$x_3 \leftarrow \mathsf{READ}(r_1)$;
$x_4 \leftarrow \mathsf{ADD}(x_3, x_1)$;
$x_5 \leftarrow \mathsf{READ}(x_4)$;
$x_6 \leftarrow \mathsf{WRITE}(r_1, x_5)$;
$x_7 \leftarrow \mathsf{INC}(r_1)$;
$x_8 \leftarrow \mathsf{DEC}(x_1)$;
$x_9 \leftarrow \mathsf{LT}(x_7, x_8)$;
$r_0 \leftarrow 0$;
$r_1 \leftarrow x_7$;
$r_2 \leftarrow x_9$;
$r_3 \leftarrow 0$;
```

## M. NRAM ListK Program

When the registers are [0,!0,!0,-,-]:

```
    $x_1 \leftarrow \mathsf{READ}(r_0)$;
    $x_2 \leftarrow \mathsf{ADD}(x_1, 2)$;
    $x_3 \leftarrow \mathsf{WRITE}(0, x_1)$;
    $r_0 \leftarrow 1$;
    $r_1 \leftarrow 1$;
    $r_2 \leftarrow 1$;
    $r_3 \leftarrow x_2$;
```

When the registers are [!0,!0,!0,-,-]:

```
    $x_1 \leftarrow \mathsf{READ}(r_1)$;
    $x_2 \leftarrow \mathsf{ADD}(x_1, 2)$;
    $x_3 \leftarrow \mathsf{WRITE}(r_1, x_1)$;
    $r_0 \leftarrow 1$;
    $r_1 \leftarrow 0$;
    $r_2 \leftarrow 1$;
    $r_3 \leftarrow r_3$;
    $r_4 \leftarrow x_2$;
```

When the registers are [!0,0,!0,-,-]:

```
    $x_1 \leftarrow \mathsf{READ}(r_3)$;
    $x_2 \leftarrow \mathsf{WRITE}(r_3, x_1)$;
    $r_0 \leftarrow 1$;
    $r_1 \leftarrow 0$;
    $r_2 \leftarrow 0$;
    $r_3 \leftarrow x_1$;
    $r_4 \leftarrow 4$;
```

When the registers are [!0,0,0,-,-]:

```
    $x_1 \leftarrow \mathsf{READ}(r_4)$;
    $x_2 \leftarrow \mathsf{WRITE}(r_4, r_3)$;
    $r_0 \leftarrow 1$;
    $r_1 \leftarrow 1$;
    $r_2 \leftarrow 0$;
    $r_3 \leftarrow x_1$;
```

When the registers are [0,!0,0,-,-]:

```
    $x_1 \leftarrow \mathsf{READ}(r_2)$;
    $x_2 \leftarrow \mathsf{ADD}(x_1, 2)$;
    $x_3 \leftarrow \mathsf{WRITE}(x_2)$;
    $r_0 \leftarrow 0$;
    $r_1 \leftarrow 1$;
    $r_2 \leftarrow 0$;
```

## N. NRAM ListK Program

Timestep 0:

```
$x_1 \leftarrow \mathsf{READ}(r_0);$
$x_2 \leftarrow \mathsf{INC}(x_1);$
$x_3 \leftarrow 0;$
$x_4 \leftarrow \mathsf{WRITE}(x_3, x_1);$
$r_0 \leftarrow r_1;$
$r_1 \leftarrow r_1;$
$r_2 \leftarrow r_2;$
$r_3 \leftarrow x_2;$
$r_4 \leftarrow r_4;$
$r_5 \leftarrow r_5;$
```

Timestep 1:

```
$x_1 \leftarrow \mathsf{READ}(r_1);$
$x_2 \leftarrow \mathsf{WRITE}(r_1, x_1);$
$x_3 \leftarrow 0;$
$r_0 \leftarrow r_0;$
$r_1 \leftarrow x_1;$
$r_2 \leftarrow r_2;$
$r_3 \leftarrow r_3;$
$r_4 \leftarrow x_3;$
$r_5 \leftarrow r_5;$
```

Timestep 2:

```
$x_1 \leftarrow \mathsf{READ}(r_2);$
$x_2 \leftarrow \mathsf{WRITE}(r_2, x_1);$
$x_3 \leftarrow 0;$
$r_0 \leftarrow r_0;$
$r_1 \leftarrow r_1;$
$r_2 \leftarrow x_1;$
$r_3 \leftarrow r_3;$
$r_4 \leftarrow x_3;$
$r_5 \leftarrow x_3;$
```

Timestep 3 to 3 + k - 1:

```
$x_1 \leftarrow \mathsf{READ}(r_0);$
$x_2 \leftarrow \mathsf{INC}(x_1);$
$x_3 \leftarrow 0;$
$x_4 \leftarrow \mathsf{DEC}(x_1);$
$x_5 \leftarrow \mathsf{WRITE}(r_0, x_1);$
$r_0 \leftarrow x_1;$
$r_1 \leftarrow x_4;$
$r_2 \leftarrow r_2;$
$r_3 \leftarrow x_2;$
$r_4 \leftarrow x_3;$
$r_5 \leftarrow x_3;$
```

Timestep 3 + k:

```
$x_1 \leftarrow \mathsf{READ}(r_3);$
$x_2 \leftarrow \mathsf{WRITE}(r_2, x_1);$
$x_3 \leftarrow 0;$
$x_4 \leftarrow 1;$
$r_0 \leftarrow r_0;$
$r_1 \leftarrow r_1;$
$r_2 \leftarrow r_2;$
$r_3 \leftarrow x_1;$
$r_4 \leftarrow x_4;$
$r_5 \leftarrow x_3;$
```

Rest:

```
$x_1 \leftarrow \mathsf{WRITE}(r_2, r_3);$
$x_2 \leftarrow 1;$
$x_3 \leftarrow 0;$
$r_0 \leftarrow r_0;$
$r_1 \leftarrow r_1;$
$r_2 \leftarrow r_2;$
$r_3 \leftarrow r_3;$
$r_4 \leftarrow x_2;$
$r_5 \leftarrow x_3;$
```