Metalearned Neural Memory

Tsendsuren Munkhdalai, Alessandro Sordoni, Tong Wang, Adam Trischler
Microsoft Research
Montréal, Québec, Canada
tsendsuren.munkhdalai@microsoft.com

Abstract

We augment recurrent neural networks with an external memory mechanism that builds upon recent progress in metalearning. We conceptualize this memory as a rapidly adaptable function that we parameterize as a deep neural network. Reading from the neural memory function amounts to pushing an input (the key vector) through the function to produce an output (the value vector). Writing to memory means changing the function; specifically, updating the parameters of the neural network to encode desired information. We leverage training and algorithmic techniques from metalearning to update the neural memory function in one shot. The proposed memory-augmented model achieves strong performance on a variety of learning problems, from supervised question answering to reinforcement learning.

1 Introduction

Many information processing tasks require memory, from sequential decision making to structured prediction. As such, a host of past and recent research has focused on augmenting statistical learning algorithms with memory modules that rapidly record task-relevant information [38, 8, 41]. A core desideratum for a memory module is the ability to store information such that it can be recalled from the same cue at later times; this reliability property has been called *self-consistency* [41].

Furthermore, a memory should exhibit some degree of *generalization*, by recalling useful information for cues that have not been encountered before, or by recalling information associated with what was originally stored (the degree of association may depend on downstream tasks). Memory structures should also be *efficient*, by scaling gracefully with the quantity of information stored and by enabling fast read-write operations.

Figure 1: Schematic illustration of the MNM model. Green and blue arrows indicate data flows for writing and reading operations, respectively. k_t^r, k_t^w, v_t^w and β_t denote read-in key, write-in key, target value and update rate vectors.

In the context of neural networks, one widely successful memory module is the soft look-up table [8, 46, 3]. This module stores high-dimensional key and value vectors in tabular format and is typically accessed by a controlled attention mechanism [3]. While broadly adopted, the soft look-up table has several shortcomings. Look-up tables are efficient to write, but they may grow without bound if information is stored naïvely in additional slots. Usually, their size is kept fixed and a more efficient writing mechanism is either learnt [8] or implemented heuristically [35]. The read operation common to most table-augmented models, which is based on soft attention, does not scale well in terms of the number of slots used or in the dimensionality of stored information [32]. Furthermore, soft look-up tables generalize only via convex combinations of stored values. This burdens the controller with estimating useful key and value representations.
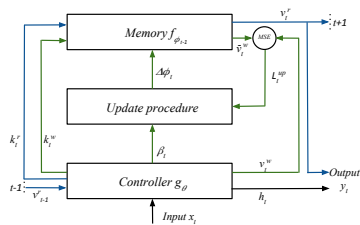
In this paper, we seek to unify few-shot metalearning and memory. We introduce an external memory module that we conceptualize as a rapidly adaptable function parameterized as a deep neural network. Reading from this "neural memory" amounts to a forward pass through the network: we push an input (the key vector) through the function to produce an output (the value vector). Writing to memory means updating the parameters of the neural network to encode desired information. We hypothesize that modelling memory as a neural function will offer compression and generalization beyond that of soft look-up tables: deep neural networks are powerful function approximators capable of both strong generalization and memorization [50, 12, 11], and their space overhead is constant.

For a neural network to operate as a useful memory module, it must be possible to record memories rapidly, i.e., to update the network in one shot based on a single datum. We address this challenge by borrowing techniques from few-shot learning through *metalearning* [1, 33, 6, 27]. Recent progress in this domain has shown how models can learn to implement data-efficient, gradient-descent-like update procedures that optimize their own parameters. To store information rapidly in memory, we propose a novel, layer-wise learned update rule. This update modifies the memory parameters to minimize the difference between the neural function's predicted output (in response to a key) and a target value. We find our novel update rule to offer faster convergence than gradient-based update rules used commonly in the metalearning literature [7].

We combine our proposed neural memory module with an RNN controller and train the full model end-to-end (Figure 1). Our model *learns to remember*: we meta-train its reading and writing mechanisms to store information rapidly and reliably. Meta-training also promotes incremental storage, as discussed in §3.5. We demonstrate the effectiveness of our metalearned neural memory (MNM) on a diverse set of learning problems, which includes several algorithmic tasks, synthetic question answering on the bAbI dataset, and maze exploration via reinforcement learning. Our model achieves strong performance on all of these benchmarks.

2   Related Work

Several neural architectures have been proposed recently that combine a controller and a memory module. Neural Turing Machines (NTMs) extend recurrent neural networks (RNNs) with an external memory matrix [8]. The RNN controller interacts with this matrix using read and write heads. Despite NTM's sophisticated architecture, it is unstable and difficult to train. A possible explanation is that its fixed-size matrix and the lack of a deallocation mechanism lead to information collisions in memory. Neural Semantic Encoders [28] address this drawback by way of a variable-size memory matrix and by writing new content to the most recently read memory entry. The Differentiable Neural Computer [9] maintains memory usage statistics to prevent information collision while still relying on a fixed-size memory matrix. Memory Networks [46, 40] circumvent capacity issues with a read-only memory matrix that scales with the number of inputs. The read-out functions of all these models and related variants (e.g., [20]) are based on a differentiable attention step [3] that takes a convex combination of all stored memory vectors. Unfortunately, this attention step does not scale well to large memory arrays [32].

Another line of work incorporates dynamic, so-called "fast" weights [14, 38, 37] into the recurrent connections of RNNs to serve as writable memory. For instance, the Hebbian learning rule [13] has been explored extensively in learning fast-weight matrices for memory [38, 2, 23, 26]. HyperNetworks generate context dependent weights as dynamic scaling terms for the weights of an RNN [10] and are closely related to conditional normalization techniques [21, 5, 31].

Gradient-based fast weights have also been studied in the context of metalearning. Meta Networks [27] define fast and slow weight branches in a single layer and train a meta-learner that generates fast weights, while conditionally shifted neurons [29] map loss gradients to fast biases, in both cases for one-shot adaptation of a classification network. Our proposed memory controller adapts its neural memory model through a set of input and output pairs (called interaction vectors below) without directly interacting with the memory weights. Another related approach from the metalearning literature [16, 43, 35, 33, 24] is MAML [6]. MAML discovers a parameter initialization from which a few steps of gradient descent rapidly adapt a model to several related tasks.

Recently, [47, 48] extended the Sparse Distributed Memory (SDM) of [18] as a generative memory mechanism [49], wherein the content matrix is parameterized as a linear Gaussian model. Memory access then corresponds to an iterative inference procedure. Memory mechanisms based on iterative

and/or neural functions, as in [47, 48] and this work, are also related to frameworks that cast memory as dynamical systems of attractors (for some background, see [42]).

3 Proposed Method

At a high level, our proposed memory-augmented model operates as follows. At each time step, the controller RNN receives an external input. Based on this input and its internal state, the controller produces a set of memory *interaction vectors*.

In the process of *reading*, the controller passes a subset of these vectors, the read-in keys, to the neural memory function. The memory function outputs a read-out value (i.e., a memory recall) in response. In the process of *writing*, the controller updates the memory function based on the remaining subset of interaction vectors: the write-in keys and target values.

We investigate two ways to bind keys to values in the neural memory: (i) by applying one step of modulated gradient descent to the memory function's parameters (§3.3); or (ii) by applying a learned local update rule to the parameters (§3.4). The parameter update reduces the error between the memory function's predicted output in response to the write-in key and the target value. It may be used to create a new association or strengthen existing associations in memory.

Finally, based on its internal state and the memory read-out, the controller produces an output vector for use in some downstream task. We meta-train the controller and the neural memory end-to-end to learn effective memory access procedures (§3.5), and call the proposed model the *Metalearned Neural Memory* (MNM). In the sections below we describe its components in detail. Figure 1 illustrates the MNM model schematically.

3.1 The Controller

The controller is a function g_θ with parameters $\theta = \{W, b\}$. It uses the LSTM architecture [15] as its core. At each time step it takes in the current external input, $x_t \in \mathbb{R}^{d_i}$, along with the previous memory read-out value $v_{t-1}^r \in \mathbb{R}^{d_v}$ and hidden state $h_{t-1} \in \mathbb{R}^{d_h}$. It outputs a new hidden state: $h_t = \text{LSTM}(x_t, v_{t-1}^r, h_{t-1})$. The controller also produces an output vector to pass to external modules (e.g., a classification layer) for use in a downstream task. The output is computed as $y_t = W_y[h_t; v_t^r] + b_y \in \mathbb{R}^{d_o}$ and depends on the memory read-out vector v_t^r. The read-out vector is computed by the memory function, as described in §3.2.

From the controller's hidden state h_t we obtain a set of interaction vectors for reading from and writing to the memory function. These include read-in keys $k_t^r \in \mathbb{R}^{d_k}$, write-in keys $k_t^w \in \mathbb{R}^{d_k}$, target values $v_t^w \in \mathbb{R}^{d_v}$, and a rate vector $\beta_t' \in \mathbb{R}^{d_k}$:

$$[k_{t,1}^r; \dots ; k_{t,H}^r; k_{t,1}^w; \dots ; k_{t,H}^w; v_{t,1}^w \dots ; v_{t,H}^w; \beta_t'] = \tanh(W_v h_t + b_v). \tag{1}$$

The controller outputs H vectors of each interaction type, where H is the number of parallel interaction heads. The single rate vector is further projected down to a scalar and squashed into $[0, 1]$: $\beta_t = \text{sigmoid}(W_\beta \beta_t' + b_\beta)$. Rate β_t controls the strength with which the corresponding (key, value) pairs should be stored in memory. The write-in keys and target values determine the content to be stored, whereas the read-in keys are used to retrieve content from the memory. We use separate keys and values for reading and writing because the model interacts with its memory in two distinct modes at each time step: It reads information stored there at previous time steps that it deems useful to the task at hand, and it writes information related to the current input that it deems will be useful in the future. The rate β_t enables the controller to influence the dynamics of the gradient-based and local update procedures that encode information in memory (§3.3 and §3.4).

3.2 The Memory Function

We model external memory as an adaptive function, f_{ϕ_t}, parameterized as a feed-forward neural network with weights $\phi_t = \{M^l\}$. Note that these weights, unlike the controller parameters θ, change rapidly from time step to time step and store associative bindings as the model encodes information. Reading from the memory corresponds to feeding the set of read-in keys through the memory function to generate a set of read-out values, $\{v_{t,i}^r\} = f_{\phi_t}(\{k_{t,i}^r\})$.

At hidden layer l, the memory function's forward computation is defined as $z^l = \sigma(M^l z^{l-1})$, where σ is a nonlinear activation function, $z^l \in \mathbb{R}^{D_l}$ is the layer's activation vector, and we have dropped the time-step index. We execute this computation in parallel on the set of H read-in keys $k_{t,i}^r$ at each time step, yielding H read-out value vectors. We take their mean to construct the single read-out value v_t^r that feeds back into the controller and the output computation for y_t.

3.3 Writing to Memory with Gradient Descent

A write to the memory consists in rapidly binding the write-in keys $\{k_{t,i}^w\}$ to map to the target values $\{v_{t,i}^w\}$ in the parameters of the neural memory. One way to do this is by updating the memory parameters to optimize an objective that encourages binding. We denote this memory objective by $\mathcal{L}_t^{\text{up}}$ and in this work implement it as a simple mean-squared error:

$$\mathcal{L}_t^{\text{up}} = \frac{1}{H} \sum_{i=1}^{H} ||f_{\phi_{t-1}}(k_{t,i}^w) - v_{t,i}^w||_2^2 \tag{2}$$

We aim to encode the target values $\{v_{t,i}^w\}$ obtained from the controller by optimizing Eq. 2. We obtain a set of memory prediction values, $\{\hat{v}_{t,i}^w\} = f_{\phi_{t-1}}(\{k_{t,i}^w\})$, by feeding the controller's write-in keys through the memory function as parameterized at the previous time step (by ϕ_{t-1}). The model binds the write-in keys to the target values at time t by taking a gradient step to minimize $\mathcal{L}_t^{\text{up}}$:

$$\phi_t \leftarrow \phi_{t-1} - \beta_t \nabla_{\phi_{t-1}} \mathcal{L}_t^{\text{up}}. \tag{3}$$

Here, β_t is the update rate obtained from the controller, which modulates the size of the gradient step. In principle, by diminishing the update rate, the controller can effectively avoid writing into memory, achieving an effect similar to a gated memory update [8].

In experiments we find that the mean-squared error is an effective memory objective: minimizing it encodes the target values in the memory function, in the sense that they can be read out (approximately) by passing in the corresponding keys.

3.4 Writing to Memory with a Learned Local Update

Writing to memory with gradient descent poses challenges. Writing a new item to a look-up table can be as simple as adding an element to the corresponding array. In a neural memory, by contrast, multiple costly gradient steps may be required to store a key-value vector pair reliably in the parameters. Memory parameter updates are expensive because, in end-to-end training, they require computation of higher-order gradients (see §3.5; this issue is common in gradient-based metalearning). Sequential back-propagation of the memory error through the layers of the memory model also adds a computational bottleneck.

Other researchers have recognized this problem and proposed possible solutions. For example, the direct feedback alignment algorithm [30], a variant of the feedback alignment method [22], enables weight updates via error alignment with random skip connections. Because these feedback connections are not computed nor used sequentially, updates can be parallelized for speed. However, fixed random feedback connections may be inefficient. The synthetic gradient methods [17] train a model to locally predict the error gradient and this requires the true gradient as a target.

We propose a memory writing mechanism that is fast and gradient-free. The key idea is to represent each neural memory layer with decoupled forward computation and backward feedback prediction functions (BFPF) and perform local updates to the memory layers. Unlike feedback alignment methods, the BFPF and the local update rules are then meta-trained, jointly. Concretely, for neural memory layer l, the BFPF is a function q_ψ^l with trainable parameter ψ_l that makes a prediction for an expected activation as: $z'^l = q_\psi^l(v_t^w)$. We then adopt the perceptron learning rule [34] to update the layer locally:

$$M_t^l \leftarrow M_{t-1}^l - \beta_t^l(z^l - z'^l)z^{l-1^T} \tag{4}$$

where β_t^l is the local update rate that can be learned for each layer with the controller or separately with the BFPF q_ψ^l. The perceptron update rule uses the predicted activation as a true target and approximate the loss gradient w.r.t M_{t-1}^l via $\beta_t^l(z^l - z'^l)z^{l-1^T}$. Therefore, the (approximate) gradient

is near zero when $z^l \approx z'^l$ and there are no changes to the weights. But if the predicted and the forward activations don't match, we update the weights such that the predicted activations can be reconstructed from the weights given the activations z^{l-1} from the previous layer $l-1$.

Therefore, the BFPF module first proposes a regression problem locally for each layer and then the perceptron learning mechanism here solves the local regression problem. One can use a different local update mechanism, rather than the perceptron method. Note that it is helpful to choose the update mechanism that is differentiable w.r.t to its solution to the problem, since the BFPF module is trained to propose problems whose solutions minimize the meta and task loss (§3.5).

With the proposed local and gradient-free update rule, the neural memory writes to its weights in parallel and its computation graph need not be tracked during writing. This makes it straightforward to add complex structural biases, such as recurrence, into the neural memory itself. The proposed approach can readily be applied in the few-shot learning setup as well. For example, we can utilize the learned local update method as an inner-loop adaptation mechanism in a model agnostic way. We leave this to future work.

3.5 End-to-end Training via Meta and Task Objectives

It is important to note that the memory objective function, Eq. 2, and the memory updates, Eqs. 3, 4, modify the neural memory function; these updates occur even at test time, as the model processes and records information (they require no external labels). We train the controller parameters θ and the memory initialization ϕ_0 and the BFPF parameters ψ end-to-end through the combination of a task objective and a meta objective. The former, denoted by $\mathcal{L}^{\text{task}}$, is specific to the task the model performs, e.g., classification or regression. The meta objective, $\mathcal{L}^{\text{meta}}$, encourages the controller to learn effective update strategies for the neural memory that work well across tasks. These two objectives take account of the model's behavior over an episode, i.e., a sequence of time steps, $t = 1, \ldots, T$, in which the model performs some temporally extended task (like question answering or maze navigation).

For the meta objective, we make use again of the mean squared error between the memory prediction values and the target values, as in Eq. 2. For the meta objective, however, we obtain the prediction values from the *updated* neural function at each time step, f_{ϕ_t}, after the update step in Eq. 3 or Eq. 4 has been applied. We also introduce a recall delay parameter, τ:

$$\mathcal{L}^{\text{meta}} = \frac{1}{TH} \sum_{\tau=0}^{\mathcal{T}} \sum_{t=1}^{T} \sum_{i=1}^{H} \lambda_\tau ||f_{\phi_t}(k_{t-\tau,i}^w) - v_{t-\tau,i}^w||_2^2 \tag{5}$$

The sum over τ can be used to reinforce older memory values, and λ_τ is a decay weight for the importance of older (key, value) pairs. The latter can be used, for instance, to implement a kind of exponential decay. We found that λ_τ is task specific; in this work, we always set the maximum recall delay as $\mathcal{T} = 0$ to focus on reliable storage of new information.

At the end of each episode, we take gradients of the meta objective and the task objective with respect to the controller parameters, θ, and use these for training the controller:

$$\theta \leftarrow \theta - \nabla_\theta \mathcal{L}^{\text{task}} - \nabla_\theta \mathcal{L}^{\text{meta}}. \tag{6}$$

These gradients propagate back through the memory-function updates (requiring higher-order gradients if using Eq. 3) so that the controller learns *how* to modify the memory parameters via the interaction vectors (Eq. 1) and the gradient steps or local updates (Eq. 3 or Eq. 4, respectively).

We attempted to learn the memory initialization (similar to MAML) by updating the initial parameters ϕ_0 w.r.t. the meta loss. We found that this led to severe overfitting. Therefore, we initialize the memory function *tabula rasa* at each task episode from a fixed random parameter set.

Optimizing the episodic task objective will often require the memory to recall information stored many time steps back, after newer information has also been written. This requirement, along with the fact that the optimization involves propagating gradients back through the memory update steps, promotes incremental learning in the memory function, because overwriting previously stored information would harm task performance.
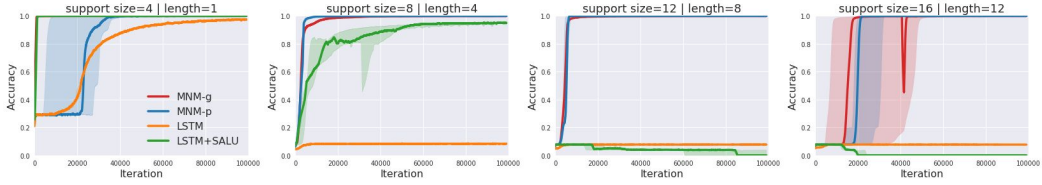
Figure 2: Training curves on the dictionary inference task.

4 Experimental Evaluation and Analysis

4.1 Algorithmic Tasks

We first introduce a synthetic dictionary inference task to test MNM's capacity to store and recall associated information. This can be considered a toy translation problem. To construct it, we randomly partition the 26 letters of the English alphabet evenly into a source and a target vocabulary, and define a random, bijective mapping F between the two sets. Following the few-shot learning setup, we then construct a support set of k source sequences with their corresponding target sequences. Each source sequence consists of l letters randomly sampled (with replacement) from the source vocabulary, which are then mapped to the corresponding target sequence using F. The objective of the task is to predict the target given a previously unseen source sequence whose composing letters have been observed in the support set. For example, after observing the support examples abc→def;tla→qzd, the model is expected to translate input sequence tca to the output qfd.

The difficulty of the task varies depending on the sequence length l and the size of the support set k. Longer sequences introduce long-term dependencies, whereas a larger number of observations requires efficient memory structure and compression. We constructed four different task instances with support set size of 4, 8, 12, and 16 and sequence length of 1, 4, 8 and 12.

We trained the MNM models with both gradient-based (MNM-g) and local memory updates (MNM-p). The models have a controller network with 100 hidden units and a three-layer feed-forward memory with 100 hidden units and $\tanh$ activation. We compare against two baseline models: a vanilla LSTM model and a memory-augmented model with the soft-attention look-up table as memory (LSTM+SALU). In the LSTM+SALU model, we replace the feed-forward neural memory with the look-up table, providing an unbounded memory to the LSTM controller. The training setup is given in Appendix A.

Figure 2 shows the results for our dictionary inference task (averaged over 5 runs). All models solved the first task instance and all memory-augmented models converged for the second case. As the task difficulty increased for the last two cases, only the MNM models converged and solved the task with zero error.

In Figure 3, we compared the training wallclock time and the memory size of MNM(-g, -p) against LSTM+SALU models on these task runs. When the input length is small, the LSTM+SALU model is faster than MNM-g and similar in speed to MNM-p, and has a smaller memory footprint than both. However, as soon as the input length exceeds the size of the MNM memory's hidden units, LSTM+SALU becomes less efficient. It exhibits quadratic growth whereas the MNM models grow approximately linearly in the wallclock time. Figure 3's left plot also demonstrates that that learned
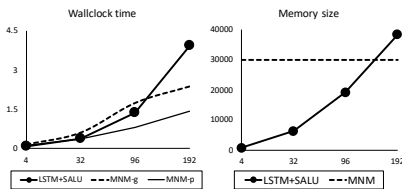

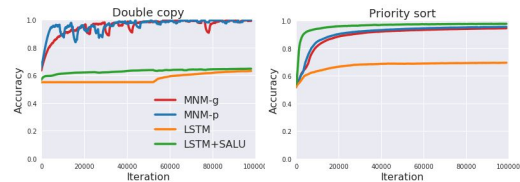Figure 3: Model comparison over varying input lengths (x-axes).

Figure 4: Training curves on programming tasks.

6

Table 1: Results on bAbI question answering.

	Sentence-level		Word-level			
	EntNet	TPR-RNN	DNC	SDNC	MNM-g	MNM-p
Mean Error	9.7 ± 2.6	1.34 ± 0.52	12.8 ± 4.7	6.4 ± 2.5	3.2 ± 0.5	**0.55 ±0.74**
Failed Tasks ($> 5\%$ error)	5 ± 1.2	0.86 ± 1.11	8.2 ± 2.5	4.1 ± 1.6	1.3 ± 0.8	**0.08 ± 0.28**

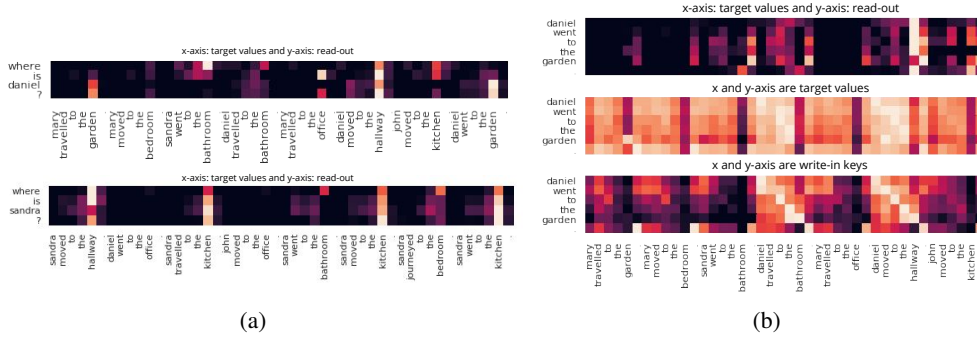(a)   (b)

Figure 5: Visualization of learned memory operations. (a) At each question word (y-axis), the model recalls memory contents written for entities in the story (x-axis) that are most closely related to the question type (e.g., locations for *where* questions). (b) Towards the end of a story (y-axis), the model learns to access and update the memory conditioned on structured information (e.g., location and character) memorized earlier in the story (x-axis).

local updates (MNM-p) confer significant speed benefits over gradient-based updates (MNM-g), since the former can be applied in parallel.

We further evaluated these models on two standard memorization tasks: double copy and priority sort. As shown in Figure 4, the MNM models quickly solve the double copy task with input length 50. On the priority sort problem, the LSTM+SALU model demonstrated the strongest result. This is surprising, since the task was previously shown to be hard to solve [8]. It suggests that the unbounded memory table and the look-up operation are especially useful for sorting. The MNM models' strong performance across the suite of algorithmic tasks, which require precise recall of past information, indicates that MNM can store information reliably.

4.2   bAbI Question Answering

bAbI is a synthetic question-answering benchmark that has been widely adopted to evaluate long-term memory and reasoning [45]. It presents 20 reasoning tasks in total. We aim to solve all of them with one generic model. Previous memory-augmented models addressed the problem with sentence-level [40, 20] or word-level inputs [9, 32]. Solving the task based on word-level input is harder, but more general [36]. We trained word-level MNM models following the setup for the DNC [9].

The results are summarized in Table 1. The MNM model with the learned local update solved all the bAbI tasks with near zero error, outperforming the result of TPR-RNN [36] (previously the best model operating at sentence-level). It also outperformed the DNC by around 12% and the Sparse DNC [9] by around 6% in terms of mean error. We report the best results and all 12 runs of MNM-p in Appendix B. MNM-g with the gradient-based update solved 19 tasks, failing to solve only the basic induction task.

We also attempted to train a word-level LSTM+SALU baseline as described in the previous section. However, multiple LSTM+SALU runs did not solve any task and converged to 77.5% mean error after 62K training iterations. With the same number of iterations, the MNM runs converged to a 9.5% mean error and solved 16.5 tasks on average. This suggests the importance of a deep neural memory and a learned memory access mechanism for reasoning.

Analyzing Learned Memory Operations: To understand the MNM more deeply, we analyzed its temporal dynamics through the similarity between the keys and values of its read/write operations as it processed bAbI. Here cosine distance is used as a similarity metric. We found that the neural memory exhibits readily interpretable structures as well as efficient self-organization.

Intuitively, keys and values correspond to the locations and contents of memory read/write operations. Consequently the temporal comparison of various combinations of these vectors can have meaningful interpretations. For example, given two time steps $t_1 < t_2$, when comparing $v_{t_2}^r$ against $v_{t_1}^w$, higher similarity (brighter colors in Figure 5) indicates that the content stored at t_1 is being retrieved at t_2. We first applied this comparison to a bAbI story (x-axis in Figure 5a) and the corresponding question (y-axis), since they are read consecutively by the model. Upon reading the question word "where", the model successfully retrieves the location-related memories. When the model reads in the character names, the retrieval is then "filtered" down to only the locations of the characters in question. Furthermore, the retrieval also appears to be closer to more recent locations, effectively modeling this strong prior in the data distribution of bAbI.

Similarly, we analyzed the memory operations towards the end of a story (y-axis) and examined how the model uses the memory developed earlier (x-axis). Again, comparing $v_{t_2}^r$ and $v_{t_1}^w$ (row 1 in Figure 5b), the bright vertical stripe at "hallway" indicates that the memory retrieval focuses more on Daniel's most recent location (while ignoring both his previous locations and locations of other characters). In addition, $v_{t_2}^w$ and $v_{t_1}^w$ are compared in row 2, Figure 5b, where the dark vertical stripes indicate that the memory is being updated aggressively with new contents whenever a new location is mentioned — potentially establishing new associations between locations and characters. In the comparison between $k_{t_2}^w$ and $k_{t_1}^w$ (row 3 in Figure 5b), two bright diagonals are observed in the sentences related to the matching character Daniel, suggesting that (a) the model has likely acquired an entity-based structure and (b) it is capable of leveraging this structure for efficient memory reuse.

More examples can be found in the appendix. Overall, the patterns above are salient and consistent, indicating our model's ability to disentangle objects and their roles from a story, and to use that information to dynamically establish and update associations in a structured, efficient manner — all of which are key to neural-symbolic reasoning [39] and effective generalization.

4.3 Maze Exploration by Reinforcement Learning

External memory may be helpful or even necessary for agents operating in partially observable environments, where current decision making depends on a sequence of past observations. However, memory mechanisms also add complexity to an agent's learning process [19], since the agent must learn to access its memory during experience. In our final set of experiments, we train an RL agent augmented with our metalearned neural memory. We wish to discover whether an MNM agent can perform well in a sequential decision making problem and use its memory to improve performance or sample efficiency.

Figure 6: Training curves on the maze exploration task.

We train MNM agents on a maze exploration task from the literature on meta-reinforcement learning [4, 44]. Specifically, we adopted the grid world setup from [23]. In this task, for each episode, the agent explores a grid-based maze and attempts to reach a goal position. Reaching the goal earns the agent a reward of 10 and relocates it to a random position. The agent must then return to the goal to collect more reward before the episode terminates. To the agent, the goal is invisible in the maze and its position is chosen randomly at the beginning of each episode. Inputs to the agent are its surrounding 3×3 cells, the time step, and the reward from the previous time step. The agent receives -0.1 reward for hitting a wall and 0 reward otherwise. A 9×9 grid maze is shown in Figure 7 (Appendix A) for illustration.

We trained agents on a 9×9 maze following the setup of [23] to provide a direct comparison with the differential plasticity agents of that work. We used the Advantage Actor-Critic (A2C) algorithm for optimization, a non-asynchronous variant of the A3C method [25]. The MNM agent has a neural memory and controller with 100 and 200 hidden units, respectively. The training curve for the 9×9 maze (averaged over 10 runs) is plotted in Figure 6, along with results from [23]. As can be seen, the agents with *differential plasticity* (denoted Plastic and Homogenous Plastic) converge to a reward of 175 after training on nearly 1M episodes. MNM, on the other hand, reaches the same reward in only 250K episodes. It obtains significantly higher final reward after 1M episodes. This result shows
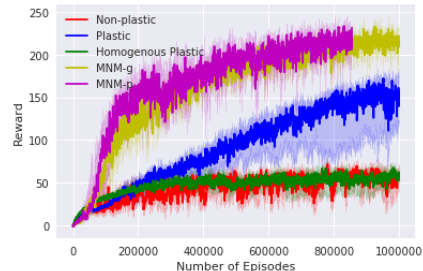
that the MNM fosters improved performance and sample efficiency in a sequential decision making scenario and, promisingly, it can be trained in conjunction with an RL policy.

5 Conclusion

We cast external memory for neural models as a rapidly adaptable function, itself parameterized as a deep neural network. Our goal was for this memory mechanism to confer the benefits of deep neural networks' expressiveness, generalization, and constant space overhead. In order to write to a neural network memory rapidly, in one shot, and incrementally, such that newly stored information does not distort existing information, we adopted training and algorithmic techniques from metalearning. The proposed memory-augmented model, MNM, was shown to achieve strong performance on a wide variety of learning problems, from supervised question answering to reinforcement learning. Our learned local update algorithm can be applied in an other setup than the memory one. In future work, we will investigate different neural architectures for metalearned memory and the effects of recall delays in the meta objective.

Acknowledgements

References

[1] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, pages 3981–3989, 2016.

[2] Jimmy Ba, Geoffrey E Hinton, Volodymyr Mnih, Joel Z Leibo, and Catalin Ionescu. Using fast weights to attend to the recent past. In *Advances in Neural Information Processing Systems*, pages 4331–4339, 2016.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[4] Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl^2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

[5] Vincent Dumoulin, Jonathon Shlens, and Manjunath Kudlur. A learned representation for artistic style. *CoRR, abs/1610.07629*, 2(4):5, 2016.

[6] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1126–1135, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[7] Chelsea Finn, Tianhe Yu, Tianhao Zhang, Pieter Abbeel, and Sergey Levine. One-shot visual imitation learning via meta-learning. *arXiv preprint arXiv:1709.04905*, 2017.

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

[9] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471, 2016.

[10] David Ha, Andrew Dai, and Quoc V Le. Hypernetworks. In *ICLR 2017*, 2017.

[11] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[12] Babak Hassibi and David G Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in neural information processing systems*, pages 164–171, 1993.

[13] Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Psychology Press, 1949.

[14] Geoffrey E Hinton and David C Plaut. Using fast weights to deblur old memories. In *Proceedings of the ninth annual conference of the Cognitive Science Society*, pages 177–186, 1987.

[15] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[16] Sepp Hochreiter, A Steven Younger, and Peter R Conwell. Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks*, pages 87–94. Springer, 2001.

[17] Max Jaderberg, Wojciech Marian Czarnecki, Simon Osindero, Oriol Vinyals, Alex Graves, David Silver, and Koray Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1627–1635. JMLR. org, 2017.

[18] Pentti Kanerva. *Sparse distributed memory*. MIT press, 1988.

[19] Arbaaz Khan, Clark Zhang, Nikolay Atanasov, Konstantinos Karydis, Vijay Kumar, and Daniel D Lee. Memory augmented control networks. *arXiv preprint arXiv:1709.05706*, 2017.

[20] Ankit Kumar, Ozan Irsoy, Peter Ondruska, Mohit Iyyer, James Bradbury, Ishaan Gulrajani, Victor Zhong, Romain Paulus, and Richard Socher. Ask me anything: Dynamic memory networks for natural language processing. In *International Conference on Machine Learning*, pages 1378–1387, 2016.

[21] Jimmy Lei Ba, Kevin Swersky, Sanja Fidler, et al. Predicting deep zero-shot convolutional neural networks using textual descriptions. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4247–4255, 2015.

[22] Timothy P Lillicrap, Daniel Cownden, Douglas B Tweed, and Colin J Akerman. Random synaptic feedback weights support error backpropagation for deep learning. *Nature communications*, 7, 2016.

[23] Thomas Miconi, Jeff Clune, and Kenneth O Stanley. Differentiable plasticity: training plastic neural networks with backpropagation. *arXiv preprint arXiv:1804.02464*, 2018.

[24] Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. A simple neural attentive meta-learner. 2018.

[25] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[26] Tsendsuren Munkhdalai and Adam Trischler. Metalearning with hebbian fast weights. *arXiv preprint arXiv:1807.05076*, 2018.

[27] Tsendsuren Munkhdalai and Hong Yu. Meta networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2554–2563, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[28] Tsendsuren Munkhdalai and Hong Yu. Neural semantic encoders. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, pages 397–407. Association for Computational Linguistics, 2017.

[29] Tsendsuren Munkhdalai, Xingdi Yuan, Soroush Mehri, and Adam Trischler. Rapid adaptation with conditionally shifted neurons. In *International Conference on Machine Learning*, pages 3661–3670, 2018.

[30] Arild Nøkland. Direct feedback alignment provides learning in deep neural networks. In *Advances in Neural Information Processing Systems*, pages 1037–1045, 2016.

[31] Ethan Perez, Florian Strub, Harm De Vries, Vincent Dumoulin, and Aaron Courville. Film: Visual reasoning with a general conditioning layer. *arXiv preprint arXiv:1709.07871*, 2017.

[32] Jack Rae, Jonathan J Hunt, Ivo Danihelka, Timothy Harley, Andrew W Senior, Gregory Wayne, Alex Graves, and Timothy Lillicrap. Scaling memory-augmented neural networks with sparse reads and writes. In *Advances in Neural Information Processing Systems*, pages 3621–3629, 2016.

[33] Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. In *ICLR 2017*, 2017.

[34] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[35] Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. Meta-learning with memory-augmented neural networks. In *International conference on machine learning*, pages 1842–1850, 2016.

[36] Imanol Schlag and Jürgen Schmidhuber. Learning to reason with third order tensor products. In *Advances in Neural Information Processing Systems*, pages 10003–10014, 2018.

[37] J Schmidhuber. Reducing the ratio between learning complexity and number of time varying variables in fully recurrent nets. In *International Conference on Artificial Neural Networks*, pages 460–463. Springer, 1993.

[38] Jürgen Schmidhuber. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139, 1992.

[39] Paul Smolensky. Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial intelligence*, 46(1-2):159–216, 1990.

[40] Sainbayar Sukhbaatar, Jason Weston, Rob Fergus, et al. End-to-end memory networks. In *Advances in neural information processing systems*, pages 2440–2448, 2015.

[41] Wen Sun, Alina Beygelzimer, Hal Daumé III, John Langford, and Paul Mineiro. Contextual memory trees. *arXiv preprint arXiv:1807.06473*, 2018.

[42] Adam Trischler. *A Computational Model for Episodic Memory Inspired by the Brain*. PhD thesis, 2016.

[43] Oriol Vinyals, Charles Blundell, Tim Lillicrap, Daan Wierstra, et al. Matching networks for one shot learning. In *Advances in Neural Information Processing Systems*, pages 3630–3638, 2016.

[44] Jane X Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016.

[45] Jason Weston, Antoine Bordes, Sumit Chopra, Alexander M Rush, Bart van Merriënboer, Armand Joulin, and Tomas Mikolov. Towards ai-complete question answering: A set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698*, 2015.

[46] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. *CoRR*, abs/1410.3916, 2014.

[47] Yan Wu, Greg Wayne, Alex Graves, and Timothy Lillicrap. The kanerva machine: A generative distributed memory. *ICLR 2018*, 2018.

[48] Yan Wu, Gregory Wayne, Karol Gregor, and Timothy Lillicrap. Learning attractor dynamics for generative memory. In *Advances in Neural Information Processing Systems*, pages 9401–9410, 2018.

[49] Richard S Zemel and Michael C Mozer. A generative model for attractor dynamics. In *Advances in neural information processing systems*, pages 80–88, 2000.

[50] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.

Table 2: Hyperparameters used in the experiments

	Controller				Memory				
	Layers	d_h	d_o	d_i	Layers	d_k	d_v	D_l	H
Algorithmic tasks	1	100	100	100	3	100	100	100	1
bAbI task	1	256	256	256	3	100	100	100	3
Maze expl.	1	200	200	-	3	100	100	100	3

A Training Details

We used `tanh` activation function and three layer feed-forward neural net for our memory. For the BFPF q_ψ^l, we performed an initial experiment evaluating feed-forward and LSTM architectures with different inputs and a simple single-layer MLP with input v_t^w worked well. The number of parallel heads H was 1, 3 and 3 for the algorithmic, bAbI and maze exploration experiments, respectively.

For dictionary inference task, we used four special input characters in addition to the main vocabulary for the end of a sequence, a support example separator, the end of a support set and an input place holder for target. For double copy task, input sequences of length 50 were constructed by randomly sampling (with replacement) from 10 unique characters. For the sort task, input sequences were length of 20 and consisted of 8-bit binary vectors along with their scalar weights. The model were trained to predict the first 16 vectors sorted. This follows the setup of NTM [8].

For bAbI question answering, we appended each question to the end of its related story and inserted additional placeholders for answer tokens. The models read the story first and then the question word-by-word, and once reaching the answer placeholders produce a prediction. The standard data splits were used in the experiment. We perform an early-stopping based on the standard development set and evaluate on the test set.

The batch sizes were set to 32 and 128 for the algorithmic and bAbI experiments, respectively. All models were optimized using Adam optimizer. The hyperparameters for Adam optimizer were set to default values (alpha=0.001 and beta=0.9) for all learning problems except the RL one. For the RL task, we used the same hyperparameters as [23]. Table 2 lists our model hyperparameters. In Figure 7, we have shown an instance of 9×9 maze.

B Detailed Results on bAbI Task

Table 3 and 4 show the best results of the compared models and the detailed runs of our best performing model.

Table 3: Best results on bAbI question answering.

Task	Sentence-level		Word-level			
	EntNet	TPR-RNN	DNC	SDNC	MNM-g	MNM-p
1: one supporting fact	0.1	0	0	0	0	0
2: two supporting facts	2.8	0.4	0.4	0.6	0.2	0.1
3: three supporting facts	10.6	3.4	1.8	0.7	1.8	0.9
4: two argument rel.	0	0.2	0	0	0	0
5: three argument rel.	0.4	1.0	0.8	0.3	0.4	0.3
6: yes/no questions	0.3	0.1	0	0	0	0
7: counting	0.8	1.0	0.6	0.2	0.2	0.3
8: lists/sets	0.1	0.5	0.3	0.2	0.2	0
9: simple negation	0	0.3	0.2	0	0	0
10: indefinite kd.	0	0.4	0.2	0.2	0.1	0
11: basic coref.	0	1.3	0	0	0	0
12: conjunction	0	0.2	0	0.1	0	0
13: compound coref.	0	2.1	0	0.1	0	0
14: time reasoning	3.6	0.2	0.4	0.1	0.5	0.1
15: basic deduction	0	0	0	0	0	0
16: basic induction	52.1	0.4	55.1	54.1	51.2	0.7
17: positional reasoning	11.7	0.6	12.0	0.3	0	0
18: size reasoning	2.1	0	0.8	0.1	0	0.2
19: path finding	63.0	4.2	3.9	1.2	0.7	0.9
20: agent's motivation	0	0	0	0	0	0
Mean Error:	7.38	0.81	3.8	2.9	2.76	**0.175**
Failed Tasks (> 5% error):	4	0	2	1	1	**0**

Table 4: Results from 12 runs of MNM-p model.

| Task	run-1	run-2	run-3	run-4	run-5	run-6	run-7	run-8	run-9	run-10	run-11	run-12	Mean	Best
1: one supporting fact	0	0	0	0	0	0	0	0	0	0	0	0	0 ± 0	0
2: two supporting facts	0.2	0	0.7	0.1	0.1	0.3	0.4	0	0.2	0.1	0.3	0	0.2 ± 0.2	0
3: three supporting facts	2	2.1	2.2	1.7	0.9	1.1	1.9	1.5	2	2.1	2.4	1.7	1.8 ± 0.43	0.9
4: two argument rel.	0	0	0	0	0	0	0	0	0	0	0	0	0 ± 0	0
5: three argument rel.	0.5	0.8	1.1	0.4	0.3	0.7	0.2	0.8	0.8	0.4	0.4	0.7	0.59 ± 0.25	0.2
6: yes/no questions	0	0.2	0	0	0	0	0	0	0	0.1	0.1	0	0.03 ± 0.06	0
7: counting	0	0	0	0	0.3	0.2	0.3	0.3	0.1	0.4	0	0	0.13 ± 0.15	0
8: lists/sets	0	0	0	0.1	0	0.1	0	0	0.1	0.1	0	0	0.03 ± 0.05	0
9: simple negation	0.1	0	0	0	0	0	0	0	0	0	0	0	0.01 ± 0.03	0
10: indefinite kd.	0	0.1	0.1	0.1	0	0	0	0	0.1	0	0	0.1	0.04 ± 0.05	0
11: basic coref.	0	0	0	0	0	0	0	0	0	0	0	0	0 ± 0	0
12: conjunction	0	0	0	0.1	0	0	0	0	0	0	0	0	0.01 ± 0.03	0
13: compound coref.	0	0	0	0	0	0	0	0	0	0	0	0	0 ± 0	0
14: time reasoning	0.1	0.8	3.2	1.3	0.1	0.3	1.9	1	3.7	2.2	3.1	0.2	1.49 ± 1.25	0.1
15: basic deduction	0	0	0	0	0	0	0	0	0	0	0	0	0 ± 0	0
16: basic induction	0.5	0.7	49.2	0.6	0.7	1	0.9	0.3	0.4	0.3	0.4	0.5	4.63 ± 13.44	0.3
17: positional reasoning	0.5	0.3	1.9	0	0	0	0	0	0	1.5	1.3	0	0.46 ± 0.67	0
18: size reasoning	0.1	0.2	0.1	0.6	0.2	0.1	0.3	0.1	0.2	0	0.5	0.6	0.25 ± 0.2	0
19: path finding	1.4	3.1	0.7	4.2	0.9	0.2	0.6	0.1	0.1	1.4	2.1	0	1.23 ± 1.26	0
20: agent's motivation	0	0	0	0	0	0	0	0	0	0	0	0	0 ± 0	0
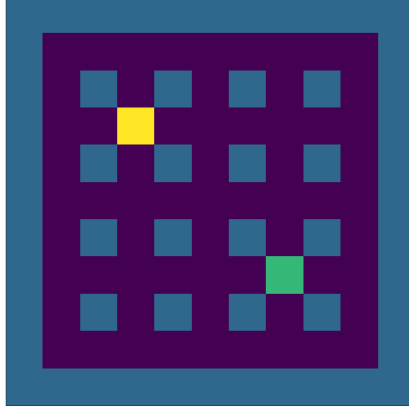
Figure 7: An example maze of 9×9 size. The current goal and agent locations are indicated in yellow and green, respectively.

C Relation to Sparse Distributed Memory

Sparse distributed memory [18] can be seen as a two-layer neural network in which the first layer outputs an address to read from or write to and the second layer encodes the memory content. Concretely, given fixed address matrix $A \in \{-1, 1\}^{D_l \times d_v}$ and content matrix $C_t \in \mathbb{Z}^{d_v \times D_l}$, sparse distributed memory first calculates an activation vector $a_t \in$ according to

$$a_t = \sigma(Ak_t), \tag{7}$$

where k_t is an input and $\sigma(m)$ is an element-wise function that outputs 1 if $\frac{1}{2}(D_l - m) \leq \delta$ and 0 otherwise, with δ a threshold. At read time, a memory output $\hat{v}_t$ is obtained by multiplying the activation vector by the content matrix: $\hat{v}_t = C_t a_t$. At write time, the memory is updated to store a content vector v_t according to:

$$C_t = C_{t-1} + v_t a_t^T. \tag{8}$$

We can derive related computations for a memory function parameterized as a two-layer feed-forward neural network ($L = 2$). The computation at the first layer is $a_t = \sigma(M_t^1 k_t)$, which becomes identical to that of the SDM if we use the address matrix A for M_t^1 and the binary activation function for σ. At read time, the computation at the second layer is $\hat{v}_t = M_t^2 a_t$, where M_t^2 is analogous to the content matrix C_t.

For our memory update, the derivative of the mean squared error with respect to the second-layer parameter matrix is:

$$\frac{\partial \mathcal{L}_t^{\text{up}}}{\partial M_t^2} = \frac{2}{d_v}(\hat{v}_t - v_t)a_t^T. \tag{9}$$

Then we can rewrite the update for the last layer as

$$M_t^2 = M_{t-1}^2 + v_t a_t^T - \hat{v}_t a_t^T, \tag{10}$$

where we have dropped the normalization constant $\frac{2}{d_v}$ and set the update rate β_t to one for convenience. Comparing Eq. 10 with Eq. 8, the gradient-based update incorporates the SDM update rule as a component and performs a slightly smarter computation, since it makes no update whenever the desired value is already stored (i.e., $\hat{v}_t = v_t$).

D Visualization of Learned Memory Operation