Time Series Analysis — DNC & EntNet.
Master-Seminar report.

Jonas Daugalas

December 20, 2020

1 Introduction

Time series and sequential data is at the heart of many important problems. A few big areas of time dependent data problems include the processing of signals, natural languages, and event streams. The last decade has witnessed a rise of successful machine learning approaches to these problems. Speech recognition, natural language translation, anomaly detection are just a few of the better known examples.

In the scenario of temporally dependent data, to successfully reason about a given time-step, it is usually necessary to have knowledge about the past. In machine learning this is achieved by equipping models with a working memory. Such memory can be used by the models to maintain information about the previous time steps and to influence the outputs for the later time steps.

Although the classical descendants of the Recurrent Neural Networks (RNN) [1, 2] such as Gated Recurrent Units (GRU) [3] or Long Short-Term Memory (LSTM) [4] networks, have been successfully applied in practice for sequential data problems, yet they have a weakness for tasks where a large working memory is needed. If we want to increase the working memory size (hidden state size or cell state size in terminology of RNNs), the number of trainable parameters also increases. The problem is that for RNNs the relationship between the memory size and the number of learnable parameters is $\mathcal{O}(N^2)$, where N is memory size.

The need for decoupling network computation capacity from its memory size can be motivated by a question answering task. Two texts can have the same length but be of completely different difficulty, and vice versa. Depending on the task we may want to control the neural network memory size and computational capacity separately.

Recent works have proposed approaches to tackle the problem of memory size and computational capacity coupling in neural networks.

The focus of this work is to review and summarize a thread of few memory augmented machine learning models in the context of time series analysis. The methods are presented in the loose order of increasing complexity of the memory architecture. We begin with the classical neural networks for sequential data, namely the RNN and the LSTM. Then we focus on more sophisticated memory interfaces as described in works of Recurrent Entity Networks (EntNet) [5], Neural Turing Machines (NTM) [6], and Differentiable Neural Computers (DNC) [7]. After that we analyze the scalability solutions as addressed in the work on Sparse Access Memory (SAM) [8]. Finally, we outline several research directions related to this class of memory augmented neural networks.

2 Memory Augmented Neural Networks (MANN) for time series analysis

In this section we overview several important approaches for neural networks to persist and manipulate memory over time.

2.1 Recurrent Neural Network

One of the foundational methods in time series analysis with a memory mechanism over time steps is the RNN [1, 9, 2]. Memory over time steps in RNNs is achieved by the recurrent connections between some network neurons. In contrast to feed-forward neural networks (FFNN), in which every input is processed independently of previously processed inputs, RNNs pass outputs of some neurons, while processing inputs at time step t, as inputs for the next processing step at time $t + 1$. These transmitted values can be seen as memory between two time steps.

In figure 1 on the left we see a schematic of a simple RNN model with the red arrow depicting the recurrent connection, on the right side of the figure we see the "unrolled" in time representation of the same model. The "unrolled" representation of the RNN is similar to FFNN, but with weight sharing over time steps (and doesn't need to be predefined for a fixed number of time steps).

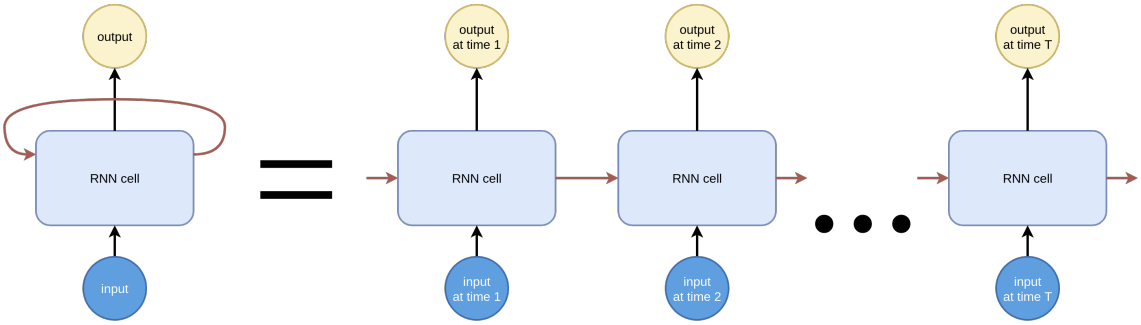

Figure 1: Diagram of a Recurrent Neural Network (RNN). On the left side of the equation is the compact RNN representation. On the right side — the "unrolled" over time representation.

Figure 2 shows computation graph of the RNN cell (as defined by [10]). $\mathbf{W}_h$, $\mathbf{W}_i$, $\mathbf{W}_o$, $\mathbf{b}_h$, and $\mathbf{b}_o$ are trainable parameters, $\times$ is vector/matrix multiplication, and g_h and g_o are activation functions.

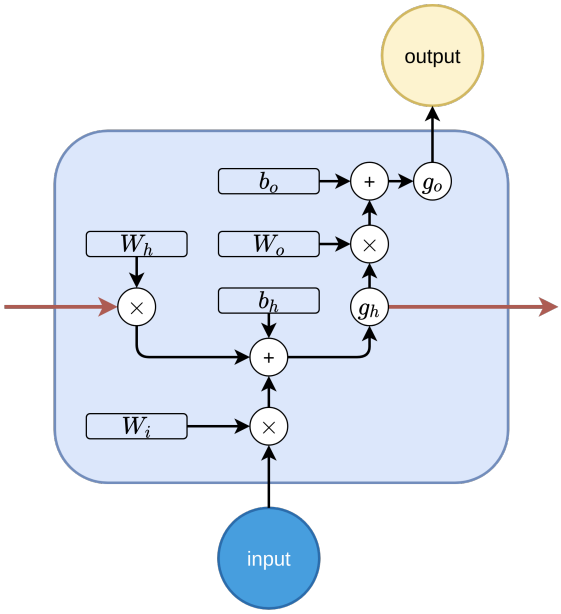

Figure 2: Computation graph of an RNN cell.

2.2 Long short-term memory

While in theory RNNs have the means to maintain memory over time steps, in practice they do not offer clear advantages over, for example, FFNNs with limited time windows [11]. Popular and in practice used RNN training methods rely on gradient based optimization — for example Backpropagation Through Time (BPTT) [9], and Real-Time Recurrent Learning (RTRL) [2]. With either of these training methods, error signals "flowing backwards in time" tend to either blow up or vanish [11]. This problem makes long-term dependencies hard to learn [12].

Long Short-Term Memory (LSTM) [4] architecture offers a solution to the vanishing and exploding gradients problem of RNNs. This is achieved by replacing matrix multiplication based cell state update with an additive update controlled by the input gate, and an element-wise down-scaling controlled by the forget gate.

An LSTM block of a popular variation including a "forget" gate, is described in equation 1 (with notation inspired by [13]), and figure 3 shows its schematic. In the equation we have:

- Trainable parameters — $\mathbf{W}_{xz}$, $\mathbf{W}_{hz}$, $\mathbf{b}_z$, $\mathbf{W}_{xi}$, $\mathbf{W}_{hi}$, $\mathbf{b}_i$, $\mathbf{W}_{xf}$, $\mathbf{W}_{hf}$, $\mathbf{b}_f$, $\mathbf{W}_{xo}$, $\mathbf{W}_{ho}$, and $\mathbf{b}_o$;
- Activation functions — g_z, and g_h;
- Sigmoid function — σ;
- Element-wise multiplication (Hadamard product) — $\circ$.

$$\mathbf{z}^t = g_z(\mathbf{W}_{xz}\mathbf{x}^t + \mathbf{W}_{hz}\mathbf{h}^{t-1} + \mathbf{b}_z) \quad \text{block input} \quad (1a)$$
$$\mathbf{i}^t = \sigma(\mathbf{W}_{xi}\mathbf{x}^t + \mathbf{W}_{hi}\mathbf{h}^{t-1} + \mathbf{b}_i) \quad \text{input gate} \quad (1b)$$
$$\mathbf{f}^t = \sigma(\mathbf{W}_{xf}\mathbf{x}^t + \mathbf{W}_{hf}\mathbf{h}^{t-1} + \mathbf{b}_f) \quad \text{forget gate} \quad (1c)$$
$$\mathbf{o}^t = \sigma(\mathbf{W}_{xo}\mathbf{x}^t + \mathbf{W}_{ho}\mathbf{h}^{t-1} + \mathbf{b}_o) \quad \text{output gate} \quad (1d)$$
$$\mathbf{c}^t = \mathbf{i}_t \circ \mathbf{z}^t + \mathbf{f}^t \circ \mathbf{c}^{t-1} \quad \text{cell state} \quad (1e)$$
$$\mathbf{h}^t = \mathbf{o}^t \circ g_h(\mathbf{c}^t) \quad \text{block output} \quad (1f)$$

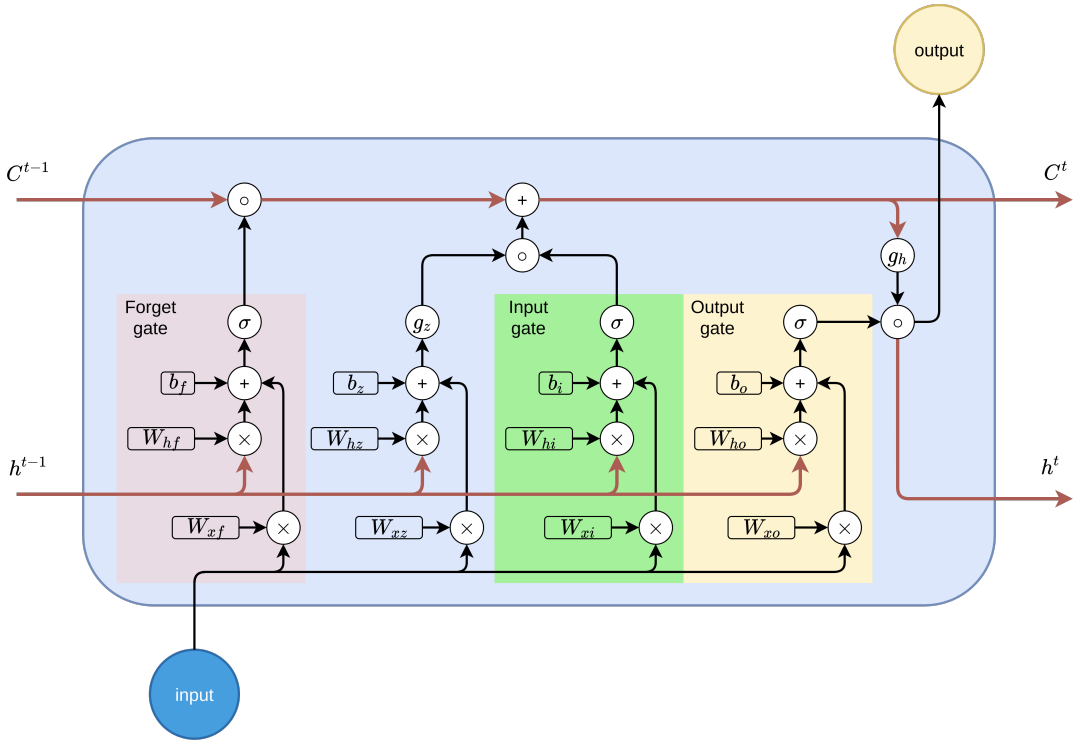

Figure 3: LSTM cell diagram.

Considering the RNN and LSTM implementations, it is apparent where the quadratic relationship between hidden state size and number of trainable parameters comes from. Looking at the LSTM diagram (3) we note that every weight matrix $\mathbf{W}_{hf}$, $\mathbf{W}_{hz}$, $\mathbf{W}_{hi}$, and $\mathbf{W}_{ho}$ must be of shape $M \times M$, where M would be the length of the cell state vector $\mathbf{c}^t$.

2.3 Recurrent Entity Networks

One idea for decoupling neural network computational complexity from memory size comes from the work on Recurrent Entity Networks (EntNet) [5]. The general idea is to apply parameter tying for the hidden state updates — learn parameters of a function which updates not single memory vector, but multiple of them. In comparison with an LSTM, EntNet can be seen as groups of hidden units with tied weights, and with additional content-based matching term between input and hidden state. We will use the terminology from the original work [5] to describe the model in more detail.

The model consists of three logical components: an input encoder, a dynamic memory, and an output layer.

2.3.1 Input encoder

Input encoder is needed to summarize input at time t to a fixed length vector representation. For EntNet one can choose this encoding module to be any standard sequence encoder, for example, the final state of an RNN run over the input subsequence.

2.3.2 Dynamic memory

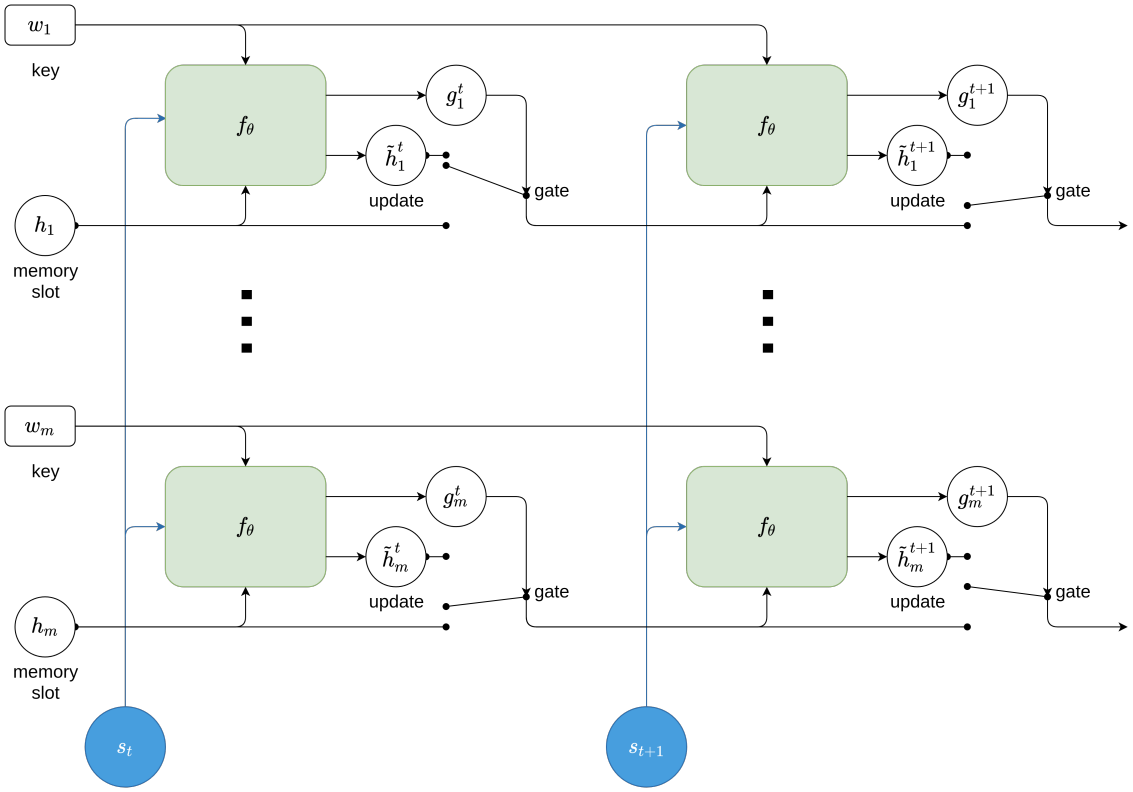

Figure 4: EntNet architecture diagram. Time step are presented along the horizontal axis and memory blocks along the vertical.

The dynamic memory module is a gated recurrent network with some weights tied in a block structured configuration. The hidden state is divided into blocks, also called memories, $h_1, h_2, \ldots, h_m$ — the whole hidden state is the concatenation of these blocks. Each memory block also has a corresponding key vector $\mathbf{w}_j$, which can be learned during training (not updated during inference) to relate its memory block with an entity of the task. At each time step the j-th memory is updated using the corresponding key vector $\mathbf{w}_j$ and the encoded input $\mathbf{s}_t$. Memory update is described in equation 2. Figure 4 shows a schematic of the EntNet dynamic memory module.

$$g_j \leftarrow \sigma(\mathbf{s}_t^\mathsf{T}\mathbf{h}_j + \mathbf{s}_t^\mathsf{T}\mathbf{w}_j) \tag{2a}$$
$$\tilde{\mathbf{h}}_j \leftarrow \phi(\mathbf{U}\mathbf{h}_j + \mathbf{V}\mathbf{w}_j + \mathbf{W}\mathbf{s}_t) \tag{2b}$$
$$\mathbf{h}_j \leftarrow \mathbf{h}_j + g_j \circ \tilde{\mathbf{h}}_j \tag{2c}$$
$$\mathbf{h}_j \leftarrow \frac{\mathbf{h}_j}{||\mathbf{h}_j||}, \tag{2d}$$

where σ is sigmoid function, g_j is gating function deciding how much the hidden state $\mathbf{h}_j$ is updated, $\tilde{\mathbf{h}}_j$ is the new candidate value for the memory update, ϕ is the activation function, and matrices $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{W}$ are trainable model parameters, shared between blocks.

The gating function g_j is controlled by a "content" term $\mathbf{s}_t^\mathsf{T}\mathbf{h}_j$ and a "location" term $\mathbf{s}_t^\mathsf{T}\mathbf{w}_j$. The "content" term causes the gate to open for memories whose content is similar to input. The "location" term causes the gate to open for memories whose key is similar to the input. "Forgetting" is implicitly modeled in the memory normalization step — this step makes memory values lie on the unit sphere thus information is contained in their phase. Hence adding any update other than itself reduces the similarity (cosine distance) between the original and the updated memory [5].

2.3.3 Output module

The output module is responsible for producing the model output. This module receives the current hidden state and a query vector $\mathbf{q}$. The output $\mathbf{y}$ is computed in the following way (equation 3):

$$p_j = \text{Softmax}(\mathbf{q}^\mathsf{T}\mathbf{h}_j) \tag{3a}$$

$$\mathbf{u} = \sum_j p_j \mathbf{h}_j \tag{3b}$$

$$\mathbf{y} = \mathbf{R}\phi(\mathbf{q} + \mathbf{H}\mathbf{u}), \tag{3c}$$

where matrices R and H are additional trainable parameters and ϕ is an activation function.

2.3.4 Outcome

EntNet successfully achieves linear relationship between the memory size and the number of trainable parameters instead of quadratic, as seen for RNN and LSTM. For any additional memory block that we choose to add to the network, we only need to add a constant length key vector.

As a bonus, the EntNet architecture provides an interpretation in the context of tasks where an agent is observing the world and needs to reason about it. An example is bAbI [14] question answering tasks (see section 3). In the case of bAbI tasks the agent (neural network) is consuming textual information, sentence by sentence, about the world and needs to be able to answer questions about the world state. Seen from this context, the memory blocks can be interpreted as pieces of information describing entities in the observed world; and the shared controller can be seen as an agent that learns the laws of the world and how to update the internal representation of the world with each new information.

2.4 Neural Turing Machine

As we have already seen, RNN maintains memory between consecutive computation steps by forwarding neuron activations from the earlier time steps. LSTM introduces mechanisms for advanced internal state control enabling the model to learn and make use of longer-term dependencies. EntNet introduces a way of reusing shared state-update-controller for multiple memory blocks (entities). In the work on Neural Turing Machine (NTM) [6] authors take a step further: they propose to equip a neural network controller with an external memory bank, and provide the controller with mechanisms to interact with the memory.

NTM is made up of two main components (see NTM architecture in figure 5): a neural network controller and a memory unit. Similar to other conventional neural networks, the controller receives external inputs and emits outputs. The main difference to other networks lies in the additional interface with the memory unit. Inheriting the naming from the traditional Turing Machine, authors refer to the interface with the memory as read and write "heads". These heads are differentiable functions that receive parameters from the controller and either return a vector of read memory, in the case of read head, or update the memory matrix contents, in the case of write head.

Important architectural feature is the interaction with the memory: reading, writing, and addressing mechanisms.

2.4.1 Reading

Reading is accomplished via read heads, and there can be be one or more of them. Every time-step the controller outputs few parameters from which a weighting (addressing) vector $\mathbf{w}_t^R \in \mathbb{R}^N$ for each read head is constructed. The read head then returns a read vector $\mathbf{r}_t \in \mathbb{R}^M$ defined as a

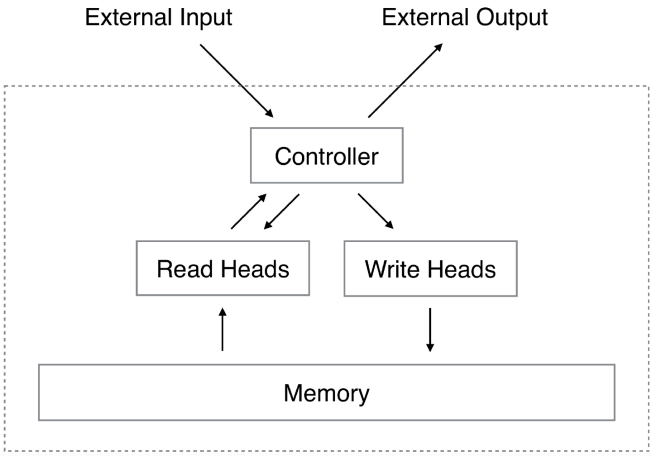

Figure 5: Neural Turing Machine (NTM) architecture. Source: [6]. At each time step the controller receives inputs and emits outputs. The controller also reads from and writes to memory via functions called read/write heads.

convex combination of the memory row vectors $M_t(i)$ [6]:

$$\mathbf{r}_t \longleftarrow \sum_i w_t^R(i)\mathbf{M}_t(i). \tag{4}$$

2.4.2 Writing

Analogous to reading, writing is achieved via one or more write heads. However, in contrast to reading, writing is composed of two parts: erasing and adding. Therefore, in addition to the weighting vectors $\mathbf{w}_t \in \mathbb{R}^N$, the controller needs to provide another two vectors to each write head: an erase-vector $\mathbf{e}_t \in (0,1)^M$, and an add vector $\mathbf{a}_t \in \mathbb{R}^M$. In essence, erase vector specifies which memory columns and by how much should be erased, add vector is the new information to be written, and the weighting vector defines the intensities of both erase and add operations for each memory entry. Erase and write operations are defined in equations 5 and 6.

$$\hat{\mathbf{M}}_t(i) \longleftarrow \mathbf{M}_{t-1}(i)[\mathbf{1} - w_t^W(i)\mathbf{e}_t], \tag{5}$$

$$\mathbf{M}_t(i) \longleftarrow \hat{\mathbf{M}}_t(i) + w_t^W(i)\mathbf{a}_t. \tag{6}$$

2.4.3 Memory addressing

Weighting vectors $\mathbf{w}_t$ supplied to the read and write heads indicate how strong and to which memory locations the heads will attend. In the terminology of NTM, this attention is called addressing. Authors of the NTM propose the construction of the weighting vectors by a combination of two focusing methods: focusing by content — "content-based addressing" ("Content Addressing" in the figure 6), and focusing by location — "location-based addressing" ("Convolutional Shift" in the figure 6).

Content-based addressing is useful in order to retrieve a piece of memory knowing only a part of it or only an approximation of it. But content-based addressing falls short when modelling variables for which the content is arbitrary, while the variable still needs to be addressable. For the latter case authors also define a location-based addressing.

For the content-based addressing, NTM uses a key vector which is compared with every row of the memory matrix by a cosine similarity. A key strength parameter β_t attenuates the precision of the

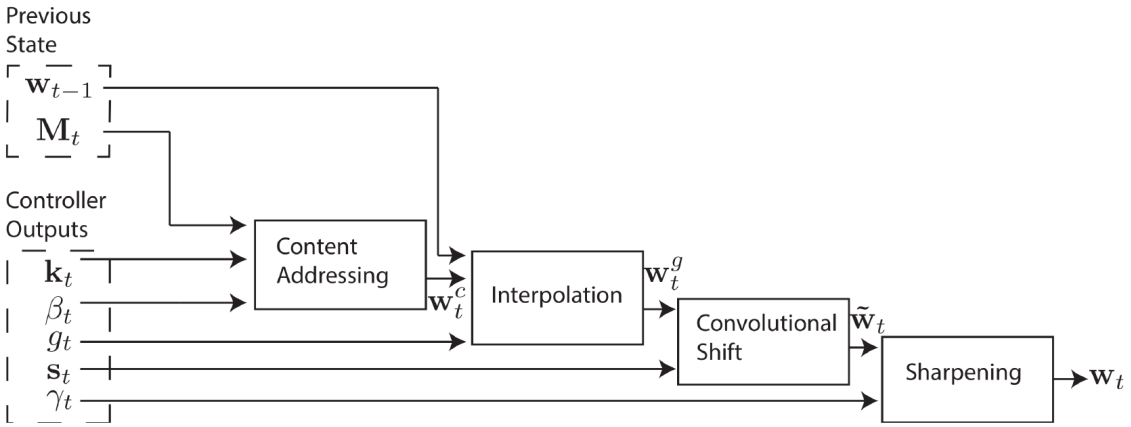

Figure 6: Flow diagram of the NTM addressing mechanism. Source: [6]. First, content-based weighting is produced from key vector $\mathbf{k}_t$ and strength parameter β_t. Parameter g_t controls the interpolation between content-based weighting $\mathbf{w}_t^c$ and weighting from the previous time step $\mathbf{w}_{t-1}$. The shifting kernel $\mathbf{s}_t$ decides by how much the addressing weighting should be rotated. Finally, γ_t can be used to sharpen the weighting distribution.

focus. Finally, the similarities are normalized with a softmax function, and the resulting vector is the weighting (addressing) vector $\mathbf{w}_t^c$:

$$\mathbf{w}_t^c(i) \longleftarrow \frac{\exp\left(\beta_t K[\mathbf{k}_t, \mathbf{M}_t(i)]\right)}{\sum_j \exp\left(\beta_t K[\mathbf{k}_t, \mathbf{M}_t(j)]\right)}, \tag{7}$$

$$K[\mathbf{u}, \mathbf{v}] = \frac{\mathbf{u} \cdot \mathbf{v}}{||\mathbf{u}|| \cdot ||\mathbf{v}||}. \tag{8}$$

Location-based addressing is useful for both iterative access of memories in sequence and arbitrary jumps. It is implemented by rotation of the weighting vector. Prior to rotation, each head interpolates (equation 9)between the weighting vector $\mathbf{w}_{t-1}$ of the previous time step and content-focused weighting $\mathbf{w}_t^c$ at the current time step. Then the shifting of the interpolation result is achieved by a circular convolution (equation 10), which is parametrized by the shift vector $\mathbf{s}_t$. Shift vector $\mathbf{s}_t$ defines a normalized distribution over the allowed integer shifts.

$$\mathbf{w}_t^g \longleftarrow g_t \mathbf{w}_t^c + (1 - g_t)\mathbf{w}_{t-1}. \tag{9}$$

$$\tilde{w}_t(i) \longleftarrow \sum_{j=0}^{N-1} w_t^g(j) s_t(i-j), \tag{10a}$$

$$w_t(i) \longleftarrow \frac{\tilde{w}_t(i)^{\gamma_t}}{\sum_j \tilde{w}_t(j)^{\gamma_t}}, \tag{10b}$$

where N memory locations are indexed from 0 to $N-1$, all index arithmetic is computed modulo N, $\gamma_t \geq 1$ is a scalar controller output for the head to perform final sharpening of the weighting vector.

2.4.4 Results

Authors demonstrate that the NTM is capable of learning simple algorithmic tasks from example data and achieving good generalization. One interesting experiment is the **Repeat Copy** task. This task requires networks to read in a random length sequence of inputs, then consume a scalar

(on a separate input channel), which indicates a number of requested repetitions. The network then needs to replay the input sequence a specified number of times and emit an end-of-sequence marker. In terms of classical data structures and control flows, such task tests the ability of networks to model and iterate an array data structure, and also treat it as a subroutine to be repeated several times.

Both the NTM and the LSTM are able to learn the repeat copy task. The big difference between two architectures becomes clear only when they are tasked to generalize beyond training data. Figure 7 visualizes how NTM and LSTM, both trained with sequences of length up to 10 and with up to 10 repeats, are performing repeat copy task with longer sequences or more repetitions. While NTM makes only few mistakes, the LSTM clearly fails the generalization test.

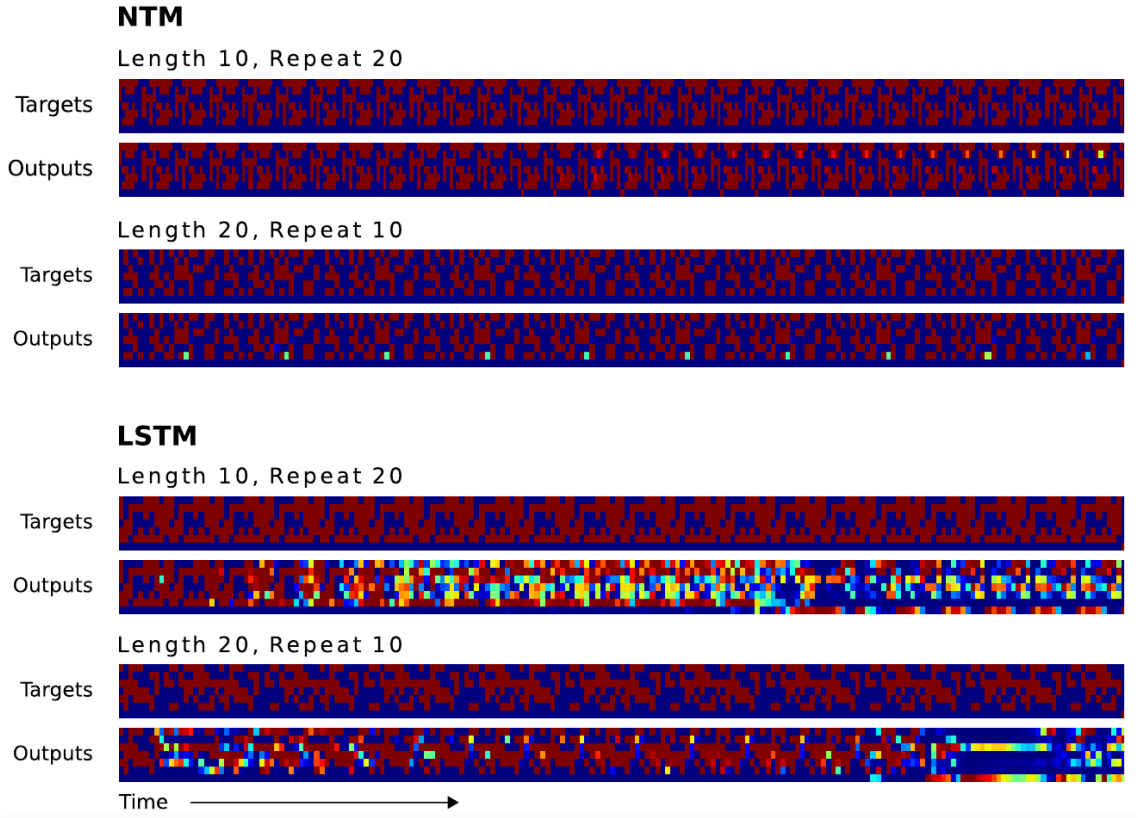

Figure 7: NTM and LSTM generalization of the Repeat Copy Task. Source: [6].
Both models were trained on sequences not longer than 10 elements and repetition number not greater than 10. In this test models had to perform repeat copy either with longer (of length 20) sequences, or do more (20) repetitions. NTM shows a decent ability to generalize, while LSTM is doing many mistakes.

2.5 Differentiable Neural Computer

In a later publication on Differentiable Neural Computers (DNC) [7] authors further build on the ideas of NTM. By redesigning external memory access mechanisms, DNC architecture overcomes few shortcomings of the predecessor NTM.

A DNC system uses a combination of content-based addressing and dynamic memory allocation to control memory write operations, and a combination of content-based addressing and temporal memory linkage to determine where to read [7]. As the content-based addressing is conceptually the same as described for the NTM, here we will only inspect the new mechanisms: dynamic memory allocation, and temporal memory linkage.

2.5.1 Dynamic memory allocation

To allow the controller to free and allocate memory as needed, authors propose a differentiable analogue of the "free list" allocation scheme [7, 15]. The conventional "free list" scheme maintains a list of available memory locations — memory addresses are added and removed to the list to indicate their usage. In the case of DNC, authors describe the method using few elements:

1. Memory usage vector $\mathbf{u}_t \in [0,1]^N$ at time t (with $\mathbf{u}_0 = \mathbf{0}$).
2. Controller emitted free gate f_t^i, one per read head. These gates indicate whether the most recently read locations can be freed.
3. Memory retention vector ψ_t (as defined in 11) representing how much each location will be retained (not freed).

$$\psi_t = \prod_{i=1}^{R} (\mathbf{1} - f_t^i \mathbf{w}_{t-1}^{R,i}), \tag{11}$$

with $\mathbf{w}_{t-1}^{R,i}$ being a read weighting emitted by the controller for time step $t-1$, for i-th read head.

Then the usage vector is defined as in equation 12:

$$\mathbf{u}_t = (\mathbf{u}_{t-1} + \mathbf{w}_{t-1}^W - \mathbf{u}_{t-1} \circ \mathbf{w}_{t-1}^W) \circ \psi_t, \tag{12}$$

here $\mathbf{w}_{t-1}^W$ is the write weighting emitted by the controller for time step $t-1$.

Intuitively, writing to a location increases its usage (to a maximum of 1), and free gates control the decrease of usage (to a minimum of 0). From the usage vector $\mathbf{u}$ the free list $\phi \in \mathbb{Z}^N$ is defined by sorting the memory location indices in ascending order of usage [7]. Finally, the allocation weighting $\mathbf{a} \in \Delta_N$ is defined as follows (equation 13):

$$\mathbf{a}_t[\phi_t[j]] = (1 - \mathbf{u}_t[\phi_t[j]]) \prod_{i=1}^{j-1} \mathbf{u}_t[\phi_t[i]]. \tag{13}$$

The discontinuities introduced by the sorting operation can be ignored during gradient computation, as they don't seem to be relevant for training [7].

In the end, final write weighting is determined by interpolating between content-based addressed location and allocation weighting (equation 14):

$$\mathbf{w}_t^W = g_t^W [g_t^a \mathbf{a}_t + (1 - g_t^a) \mathbf{c}_t^W], \tag{14}$$

where $g_t^W \in [0,1]$ is the write gate, $g_t^a \in [0,1]$ is the allocation gate deciding the interpolation, and $\mathbf{c}_t^W$ is the content addressed weighting constructed in analogous manner as described in NTM content-based addressing 7.

2.5.2 Temporal memory linkage

There are situations when it is useful to have information about the order of writing operations to memory locations. For example, if the model is tasked to record and then later retrieve a sequence of instructions. In the case of NTM the controller could achieve this by writing memories to contiguous memory locations. But such mechanism is not sufficient when we want to choose the least used memory locations for new writes as described earlier (see 2.5.1). To accomplish this goal in DNC, authors propose a temporal linkage matrix $\mathbf{L}_t \in [0,1]^{N \times N}$ for keeping track of consecutively modified memory locations [7].

Matrix element $L[i, j]$ is close to 1 if location i was written right after location j, and otherwise close to 0. Weighting vector multiplication with the linkage matrix smoothly shifts focus of that vector either towards the locations written after the current weighting focus (forward), or towards locations written before (backwards) (see equation 15).

$$\begin{aligned} \mathbf{f}_t^i &= \mathbf{L}_t \mathbf{w}_{t-1}^{R,i} \\ \mathbf{b}_t^i &= \mathbf{L}_t^\mathsf{T} \mathbf{w}_{t-1}^{R,i}, \end{aligned} \tag{15}$$

here $\mathbf{f}_t^i$ is the new weighting with focus shifted towards locations written after the locations focused by the original weighting $\mathbf{w}_{t_1}^{R,i}$; and $\mathbf{b}_t^i$ is the new weighting with focus shifted towards locations written before the location focused by $\mathbf{w}_{t-1}^{R,i}$.

A precedence vector $\mathbf{p}_t \in \Delta_N$ is used to define the linkage matrix. Element $\mathbf{p}_t[i]$ represents the degree to which the location i was the last one written to. The precedence vector is defined recursively in equation 16.

$$\begin{aligned} \mathbf{p}_0 &= \mathbf{0} \\ \mathbf{p}_t &= (1 - \sum_i \mathbf{w}_t^W[i])\mathbf{p}_{t-1} + \mathbf{w}_t^W, \end{aligned} \tag{16}$$

where $\mathbf{w}_t^W$ is the write weighting as defined in equation 14.

Finally, the linkage matrix itself is defined in equation 17. Intuitively, for each write, the strengths of links from and to current locations are reduced, and then the new links are added from the previously written locations.

$$\begin{aligned} &\mathbf{L}_0[i, j] = 0 \quad \forall i, j \\ &\mathbf{L}_t[i, i] = 0 \quad \forall i \\ &\mathbf{L}_t[i, j] = (\mathbf{1} - \mathbf{w}_t^W[i] - \mathbf{w}_t^W[j])\mathbf{L}_{t-1}[i, j] + \mathbf{w}_t^W[i]\mathbf{p}_{t-1}[j]. \end{aligned} \tag{17}$$

2.5.3 Comparison with NTM

With the redesigned memory management methods, DNC brings few improvements over the predecessor NTM. First, the new allocation mechanism allows DNC to write new memories without unintentionally overwriting or interfering with the "used" locations. Second, is the ability to explicitly free memory locations, which again facilitates reusing memory locations in a controlled manner. Third, NTM can dead with sequential information only by iterating over consecutive memories. This approach is useful until write head jumps to another location by content-based addressing, after the jump, the order of writes cannot be recovered [7]. DNC solves this with the temporal linkage matrix which keeps track of consecutive writes regardless of jumps.

The improved DNC architecture allows the model to learn more complex tasks compared to NTM. In the original work [7] authors show that DNC is able to learn several interesting tasks: algorithmic copy tasks; question answering tasks on a synthetic bAbI [14] dataset (more on bAbI dataset in section 3); graph traversal, shortest-path, and inference question tasks; planing tasks in the reinforcement learning setting. Table 1 and 2 includes the comparison of NTM and DNC performance on bAbI question answering dataset, where we see a clear improvement of the DNC over NTM in the number of successfully solved tasks and error rates.

2.6 Sparse Differentiable Neural Computer

One significant problem of DNC, as pointed out in another work [8], is the dense memory operations. First, because DNC models smooth read and write operations — a linear computational overhead on

the accessible memories is incurred per each training time step. Second, to perform backpropagation through time (BPTT), DNC requires a complete duplication of the whole memory for each time step.

In their work authors present Sparse Access Memory (SAM) [8] architecture for memory augmented neural networks. This architecture builds on top of the ideas of NTM and DNC, and includes few innovations: writes and reads are constrained to a sparse subset of memory cells; content-based addressing is implemented using approximate nearest neighbor (ANN) data structure; unused memory is managed in a sparse manner; temporal linking matrix is approximated by sparse matrices for forward and backward links. Together, all these improvements allow the inference and training phases to run in sublinear time and memory with respect to the model external memory size [8].

2.6.1 Reading

For reading SAM begins with the same formula as NTM/DNC (see equation 4), but differently, $\mathbf{w}^R$ has only K non-zero entries, and K is a small constant, independent of memory size.

To avoid naively calculating N similarities and later filtering the largest, which would require $\mathcal{O}(N)$ time to compute $\mathbf{w}^R$, the idea is to use an approximate nearest neighbors (ANN) data structure. In the read weighting computation (analogous to equation 7), K largest values of $\mathbf{w}^R$ correspond to K nearest neighbors of the query key $\mathbf{k}$. This ANN data structure makes it possible to calculate $\mathbf{w}^R$ in $\mathcal{O}(\log N)$ [8].

This type of sparse reading mechanism results in computing gradients for only a small constant K number of non-zero weights (rows of memory) instead of N.

2.6.2 Writing

Sparsity in writing is achieved by restricting it to the interpolation between two modes : **1)** write to previously read locations (update contextually relevant memories), and **2)** write to the least recently accessed location (overwriting unused memories with new content) [8].

Recall the general writing mechanism of the NTM from equations 5 and 6. In the SAM version the weighting vector $\mathbf{w}_t^W$ is constructed in a special way (see equation 18):

$$\mathbf{w}_t^W = \alpha_t(\gamma_t \mathbf{w}_{t-1}^R + (1-\gamma_t)\mathbb{I}_t^U), \tag{18}$$

where γ_t is the interpolation gate parameter, and α_t is the write gate parameter, both emitted by the controller. The read weighting of the previous time step $\mathbf{w}_{t-1}^R$ is sparse, with only K non-zero entries, as defined for SAM reading. The least recently accessed word indicator $\mathbb{I}_t^U$ is a one-hot vector (see definition in equation 19), hence the resulting write weighting $\mathbf{w}_t^W$ is also sparse.

The $\mathbb{I}_t^U$ is defined as an indicator over memory locations, with a value of 1 when the location minimizes the usage measure U_t [8]:

$$\mathbb{I}_t^U(i) = \begin{cases} 1 & \text{if } U_t(i) = \min\limits_{j=1,\dots,N} U_t(j) \\ 0 & \text{otherwise} \end{cases}. \tag{19}$$

For the usage metric authors propose two options: 1) time-discounted sum of write and read weights, 2) number of time steps since non-negligible memory access [8]. The usage metric with time-discounted weights is defined as $U_T^{(1)}(i) = \sum_{t=0}^{T} \lambda^{T-t}(\mathbf{w}_t^W(i) + \mathbf{w}_t^R)$ with λ being the discount factor. The usage metric with non-negligible memory access is defined as $U_t^{(2)}(i) = T - \max\{t : \mathbf{w}_t^W(i) + \mathbf{w}_t^R > \delta\}$ with δ being a threshold hyper-parameter (typically 0.005)[8].

2.6.3 Efficient backpropagation

In addition to efficient forward operations that can be computed in $\mathcal{O}(T \log N)$ time [8], the sparsity of operations also allows reaching efficient space complexity. Instead of naively caching the whole memory state at each time step, SAM allows us to only keep track of changes which are $\mathcal{O}(1)$ in space due to only a small constant number of memory locations written at each time step.

2.6.4 Results

SAM achieves significant improvements in computation time and memory usage compared to NTM. Figure 8 shows how increasing the number of memory slots in the model impacts the time and memory required for a single forward and backward pass over a sequence of 100 time steps. In computation time, for 1M memories, SAM offers around 1600× speed up compared to NTM. In addition, the space overhead of SAM is independent of memory size, which is apparent from the flat line in the plot.

Next critical concern is whether all the introduced approximations, which allow achieving great performance, do not impair model learning abilities. Experiment results (figure 9) prove that on the three algorithmic tasks, namely Copy, Associative Recall, and Priority Sort tasks, SAM is able to learn at least as well as NTM.

In both figures 8 and 9 the new acronyms used are: DAM — Dense Access Memory [8]; SAM linear — Sparse Access Memory not using approximate nearest neighbor data structure for content-based addressing [8]; SAM ANN — Sparse Access Memory [8].

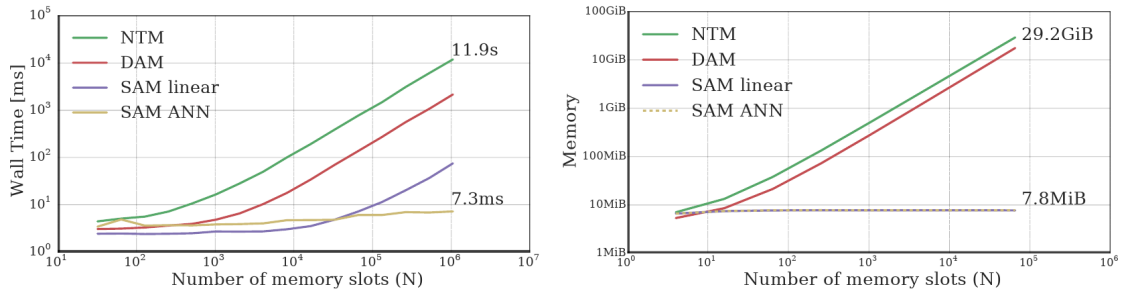

Figure 8: (a) Wall clock time of a single forward and backward pass. (b) Memory used for training over sequence of 100 time steps, excluding initialization of external memory. Source: [8].

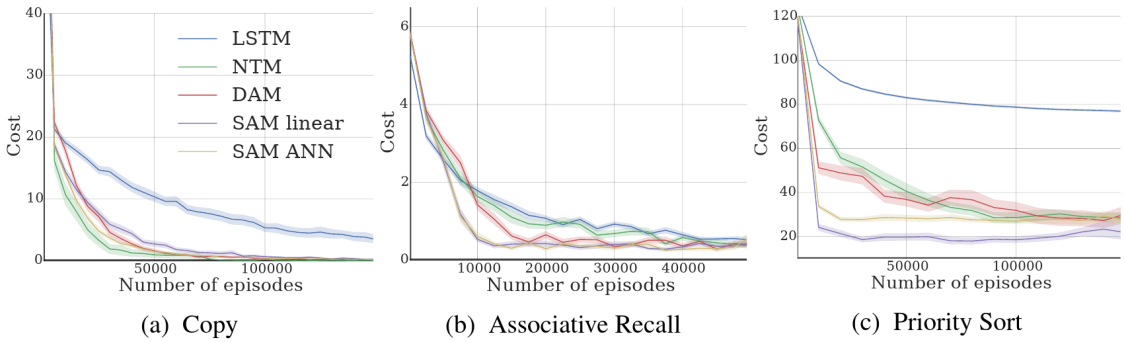

Figure 9: Training curve comparison. SAM trains comparably well for the Copy task, and for Associative Recall and Priority Sort tasks it converges even faster. Light colors indicate one standard deviation over 30 random seeds. Source: [8].

3 Comparisons

Although in different works corresponding methods are evaluated on different tasks and datasets, there is a significant overlap in benchmarking the methods on **bAbI** [14] tasks.

The **bAbI** [14] tasks are from an algorithmically generated question answering dataset, which consists of 20 different types of tasks. A model is presented with a text sequence, and when a question mark is encountered, the model needs to provide an answer which is a word or a set of words. An example of such task is shown in the figure 10.

> John is hungry. John goes to the kitchen. John grabbed the apple there. Daniel is hungry. Where does Daniel go? A:kitchen Why did John go to the kitchen? A:hungry

Figure 10: An example of bAbI task [14]. Red text represents the true labels.

Tables 1 and 2 contain aggregated results from the works discussed in the previous section. In table 1 the results are task error rates averaged over several instances of models trained with different starting seeds. Table 2 contains results of the best performing model instances from the different seeds. In all cases models are trained jointly on all 20 tasks. Tasks are considered successfully solved if test accuracy is greater or equal to 95%. SDNC name, used in the tables, correspond to Sparse Access Memory (SAM) version of DNC as described in [8].

Task	LSTM[1]	NTM[1]	DNC[1]	SDNC[1]	EntNet[2]
1	30.9 ± 1.5	31.5 ± 15.3	2.2 ± 5.6	$\mathbf{0.0 \pm 0.0}$	0.0 ± 0.1
2	57.4 ± 1.2	57.0 ± 1.3	23.9 ± 21.0	$\mathbf{7.1 \pm 14.6}$	15.3 ± 15.7
3	53.0 ± 1.4	49.4 ± 1.3	29.7 ± 15.8	$\mathbf{9.4 \pm 16.7}$	29.3 ± 26.3
4	0.7 ± 0.4	0.4 ± 0.3	$\mathbf{0.1 \pm 0.1}$	0.1 ± 0.1	0.1 ± 0.1
5	4.9 ± 0.9	2.7 ± 1.2	1.3 ± 0.3	0.9 ± 0.3	$\mathbf{0.4 \pm 0.3}$
6	18.8 ± 1.0	18.6 ± 2.7	2.8 ± 5.0	$\mathbf{0.1 \pm 0.2}$	0.6 ± 0.8
7	18.2 ± 1.1	18.7 ± 3.2	7.3 ± 5.9	$\mathbf{1.6 \pm 0.9}$	1.8 ± 1.1
8	20.9 ± 1.4	18.5 ± 5.9	4.0 ± 4.1	$\mathbf{0.5 \pm 0.4}$	1.5 ± 1.2
9	19.4 ± 1.5	17.6 ± 3.4	3.0 ± 5.2	$\mathbf{0.0 \pm 0.1}$	0.0 ± 0.1
10	33.0 ± 1.6	25.6 ± 6.9	3.2 ± 5.9	0.3 ± 0.2	$\mathbf{0.1 \pm 0.2}$
11	15.9 ± 3.3	15.2 ± 9.4	0.9 ± 3.0	$\mathbf{0.0 \pm 0.0}$	0.2 ± 0.2
12	7.0 ± 1.3	14.7 ± 8.9	1.5 ± 1.6	0.2 ± 0.3	$\mathbf{0.0 \pm 0.0}$
13	9.1 ± 1.4	6.8 ± 3.3	1.5 ± 2.5	0.1 ± 0.1	$\mathbf{0.0 \pm 0.1}$
14	57.0 ± 1.6	52.6 ± 5.1	10.6 ± 9.4	$\mathbf{5.6 \pm 2.9}$	7.3 ± 4.5
15	48.1 ± 1.3	42.0 ± 6.9	31.3 ± 15.6	$\mathbf{3.6 \pm 10.3}$	3.6 ± 8.1
16	53.8 ± 1.4	53.8 ± 2.1	54.0 ± 1.9	$\mathbf{53.0 \pm 1.3}$	53.3 ± 1.2
17	40.8 ± 1.8	40.1 ± 1.3	27.7 ± 9.4	12.4 ± 5.9	$\mathbf{8.8 \pm 3.8}$
18	7.3 ± 1.9	5.0 ± 1.2	3.5 ± 1.5	1.6 ± 1.1	$\mathbf{1.3 \pm 0.9}$
19	74.4 ± 1.3	60.8 ± 24.6	44.9 ± 29.0	$\mathbf{30.8 \pm 24.2}$	70.4 ± 6.1
20	1.7 ± 0.4	2.0 ± 0.3	0.1 ± 0.2	$\mathbf{0.0 \pm 0.0}$	0.0 ± 0.0
Mean error (%)	28.7 ± 0.5	26.6 ± 3.7	12.8 ± 4.7	$\mathbf{6.4 \pm 2.5}$	9.7 ± 2.6
Failed tasks	17.1 ± 0.8	15.5 ± 1.7	8.2 ± 2.5	$\mathbf{4.1 \pm 1.6}$	5.0 ± 1.2

Table 1: Results of different methods on bAbI tasks, showing average error and variance for several seeds of trained models. In all cases each model was trained jointly on all tasks. A task is considered failed if error is $> 5\%$.
[1]: results from [8], statistics from 15 runs of different seeds.
[2]: results from [5], statistics from 5 runs of different seeds.

Task	LSTM[1]	NTM[1]	DNC[1]	SDNC[1]	EntNet[2]
1	28.8	16.4	**0.0**	0.0	0.1
2	57.3	56.3	3.2	**0.6**	2.8
3	53.7	49.0	9.5	**0.7**	10.6
4	0.7	**0.0**	0.0	0.0	0.0
5	3.5	2.5	1.7	**0.3**	0.4
6	17.6	9.6	**0.0**	0.0	0.3
7	18.5	12.0	5.3	**0.2**	0.8
8	20.9	6.5	2.0	0.2	**0.1**
9	18.2	7.0	0.1	**0.0**	0.0
10	34.0	7.6	0.6	0.2	**0.0**
11	9.0	2.5	**0.0**	0.0	0.0
12	5.5	4.6	0.1	0.1	**0.0**
13	6.3	2.0	0.4	0.1	**0.0**
14	56.1	44.2	0.2	**0.1**	3.6
15	49.3	25.4	0.1	**0.0**	0.0
16	53.2	52.2	**51.9**	54.1	52.1
17	41.7	39.7	21.7	**0.3**	11.7
18	8.4	3.6	1.8	**0.1**	2.1
19	76.4	5.8	4.3	**1.2**	63.0
20	1.9	2.2	0.1	**0.0**	0.0
Mean error (%)	28.0	17.5	5.2	**2.9**	7.4
Failed tasks	17.0	13.0	4.0	**1.0**	4.0

Table 2: Results of different methods on bAbI tasks, showing errors of the best model from the seed. In all cases each model was trained jointly on all tasks. A task is considered failed if error is $> 5\%$.
$_1$: results from [8].
$_2$: results from [5].

Even though a single narrow benchmark dataset such as bAbI may not reveal many important strengths or weaknesses of the models, it nonetheless serves well for a baseline comparison. It is apparent that separating memory from computation, as done in EntNet, NTM, DNC, and SDNC models, helps to outperform a plain LSTM model on the bAbI tasks.

4 Related work and other directions in Memory Augmented Neural Networks

In the previous chapters we reviewed a small selection of research on sequential data and time series analysis. Specifically, the focus was on few popular memory augmented neural network models and their architectures. The surrounding areas of research are vast. Some related work directions are: general improvements over NTM/DNC based methods; various configurations of memory and its access mechanisms; decoupling input length from model computation time; and more.

Depending on the tasks at hand, the base NTM/DNC models are flexible for adjustments and improvements. Few instances of general and task specific improvements are presented by [16] and [17]. Examples of their work include: memory masking for better control over the content-based addressing, allowing the controller to dynamically decide which parts of memory to treat as keys, and which as values; a special setup of dropout on the direct controller output to force usage of the memory unit early in the training; and similar.

Another noteworthy area of related research is the exploration of different memory interfaces. A work on Dynamic Memory Networks [18] takes a constrained approach of writing to the memory only during input preprocessing step: inputs are transformed using learned embedding and preloaded into the memory, then the controller is provided with the embedding of the question and queries the memory for computation without writing to it.

In contrast to random access memory configuration, as seen in NTM and DNC, researchers are also exploring restricted memory interfaces, such as stack, doubly-linked list, queue, and dequeue based memory [19, 20].

On the more sophisticated end of memory architectures we find methods taking advantage of dual memory systems. The Self-Attentive Associative Memory [21] model separates the relational memory from item memory, and shows that the special treatment of relational information allows achieving great results on many tasks (to our knowledge, this work is currently state of the art on bAbI dataset). The work on Neural Stored-porgram memory [22] separates memory yet in another way. This architecture has one memory for data and another one for programs. The program memory stores weights of the controller network. In such scenario, the controller can learn to retrieve different weights from the program memory during runtime in order to better adapt to tasks at hand.

A concern relevant to many machine learning approaches that try attacking more general time series tasks is the question of how long the model should "think" before emitting an answer. We intuitively understand that the amount of time required to state a problem is usually unrelated with the amount of time required to solve that problem. The work on Adaptive Computation Time [23] suggests a "halting unit" for neural networks. This component is tasked to predict whether the computation should continue (the network should proceed with one more computation step), or whether it should stop and provide an output.

5 Conclusion

A large part of data problems fit under the umbrella of temporally dependent data. Possibly, the ability to predict how the world evolves over time is a feature of intelligence. This is at least one of the forces attracting scientific efforts to advance the field of machine learning for sequential data and time series analysis.

In this work we have been focusing on a class of machine learning methods for temporally dependent data, namely a thread of works trying to tackle problems by equipping models with a working memory. All together, the reviewed methods — EntNet, NTM, DNC, SAM — solve a problem relevant for many tasks of today, particularly the scaling of machine learning model memorization abilities. The combined ideas lay strong foundations for developing neural networks with memory size larger than was practically possible before. We have seen that this direction of efforts has yielded great improvements and doesn't seem to plateau yet.

References

[1] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," tech. rep., California Univ San Diego La Jolla Inst for Cognitive Science, 1985. 26947.

[2] R. J. Williams, "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks," p. 10, 1989. 00000.

[3] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation," *arXiv:1406.1078 [cs, stat]*, Sept. 2014. 12033 arXiv: 1406.1078.

[4] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997. 40467 Publisher: MIT Press.

[5] M. Henaff, J. Weston, A. Szlam, A. Bordes, and Y. LeCun, "Tracking the World State with Recurrent Entity Networks," *arXiv:1612.03969 [cs]*, May 2017. 00000 arXiv: 1612.03969.

[6] A. Graves, G. Wayne, and I. Danihelka, "Neural Turing Machines," *arXiv:1410.5401 [cs]*, Dec. 2014. 00000 arXiv: 1410.5401.

[7] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, A. P. Badia, K. M. Hermann, Y. Zwols, G. Ostrovski, A. Cain, H. King, C. Summerfield, P. Blunsom, K. Kavukcuoglu, and D. Hassabis, "Hybrid computing using a neural network with dynamic external memory," *Nature*, vol. 538, pp. 471–476, Oct. 2016. 00000.

[8] J. W. Rae, J. J. Hunt, T. Harley, I. Danihelka, A. Senior, G. Wayne, A. Graves, and T. P. Lillicrap, "Scaling Memory-Augmented Neural Networks with Sparse Reads and Writes," *arXiv:1610.09027 [cs]*, Oct. 2016. 00000 arXiv: 1610.09027.

[9] P. J. Werbos, "Backpropagation through time: what it does and how to do it," 1990.

[10] J. L. Elman, "Finding Structure in Time," *Cognitive Science*, vol. 14, no. 2, pp. 179–211, 1990. 00000 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1207/s15516709cog1402_1.

[11] S. Hochreiter and J. Schmidhuber, "Long short-term memory." 00000.

[12] S. Hochreiter, "The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 06, pp. 107–116, Apr. 1998. 00000.

[13] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A Search Space Odyssey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, pp. 2222–2232, Oct. 2017. 00000 arXiv: 1503.04069.

[14] J. Weston, A. Bordes, S. Chopra, A. M. Rush, B. van Merriënboer, A. Joulin, and T. Mikolov, "Towards AI-Complete Question Answering: A Set of Prerequisite Toy Tasks," *arXiv:1502.05698 [cs, stat]*, Dec. 2015. 00818 arXiv: 1502.05698.

[15] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *Memory Management* (G. Goos, J. Hartmanis, J. Leeuwen, and H. G. Baler, eds.), vol. 986, pp. 1–116, Berlin, Heidelberg: Springer Berlin Heidelberg, 1995. 00000 Series Title: Lecture Notes in Computer Science.

[16] R. Csordás and J. Schmidhuber, "Improving Differentiable Neural Computers Through Memory Masking, De-allocation, and Link Distribution Sharpness Control," *arXiv:1904.10278 [cs]*, Apr. 2019. 00005 arXiv: 1904.10278.

[17] J. Franke, J. Niehues, and A. Waibel, "Robust and Scalable Differentiable Neural Computer for Question Answering," *arXiv:1807.02658 [cs]*, July 2018. 00000 arXiv: 1807.02658.

[18] A. Kumar, O. Irsoy, P. Ondruska, M. Iyyer, J. Bradbury, I. Gulrajani, V. Zhong, R. Paulus, and R. Socher, "Ask Me Anything: Dynamic Memory Networks for Natural Language Processing," *arXiv:1506.07285 [cs]*, Mar. 2016. 00000 arXiv: 1506.07285.

[19] A. Joulin and T. Mikolov, "Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets," *arXiv:1503.01007 [cs]*, June 2015. 00000 arXiv: 1503.01007.

[20] E. Grefenstette, K. M. Hermann, M. Suleyman, and P. Blunsom, "Learning to Transduce with Unbounded Memory," *arXiv:1506.02516 [cs]*, Nov. 2015. 00000 arXiv: 1506.02516.

[21] H. Le, T. Tran, and S. Venkatesh, "Self-Attentive Associative Memory," *arXiv:2002.03519 [cs, stat]*, June 2020. 00000 arXiv: 2002.03519.

[22] H. Le, T. Tran, and S. Venkatesh, "Neural Stored-program Memory," *arXiv:1906.08862 [cs, stat]*, Dec. 2019. 00000 arXiv: 1906.08862.

[23] A. Graves, "Adaptive Computation Time for Recurrent Neural Networks," *arXiv:1603.08983 [cs]*, Feb. 2017. 00226 arXiv: 1603.08983.