Implementation and Optimization of Differentiable Neural Computers

Carol Hsin
Graduate Student in Computational & Mathematical Engineering
Stanford University
cshsin[at]stanford.edu

Abstract

We implemented and optimized Differentiable Neural Computers (DNCs) as described in the Oct. 2016 DNC paper [1] on the bAbI dataset [25] and on copy tasks that were described in the Neural Turning Machine paper [12]. This paper will give the reader a better understanding of this new and promising architecture through the documentation of the approach in our DNC implementation and our experience of the challenges of optimizing DNCs. Given how recently the DNC paper has come out, other than the original paper, there were no such explanation, implementation and experimentation of the same level of detail as this project has produced, which is why this project will be useful for others who want to experiment with DNCs since this project successfully trained a high performing DNC on the copy task while our DNC performance on the bAbI dataset was better than or equal to our LSTM baseline.

1. Introduction

One of the main challenges with LSTMs/GRUs is that they have difficulty learning long-term dependencies due to having to store their memories in hidden units that are essentially compressed, weight-selected input sequences. Hence, the excitement when DeepMind released a paper this October documenting their results on a new deep learning architecture, dubbed the Differentiable Neural Computer (DNCs), with read and write/erase access to an external memory matrix, thus allowing the model to learn over much longer time sequences than current LSTMs models.

In this paper, we first give a brief overview of how the DNC fits into the deep learning historical framework since we view DNCs as RNNs the way LSTMs are RNNs, as explained in Sec. 3. The rest of the paper focuses on the implementation and optimization of DNCs, which is essentially an unconstrained, non-convex optimization problem for which the KKT stationarity condition must hold for local optima (given that vanishing gradients is essentially not a problem for DNCs). A basic machine learning and linear algebra background has been assumed, so common functions and concepts are used without introduction with some explained in the appendix in Sec. 6.

We discovered that understanding and implementing a correct DNC is a non-trivial process as documented in Sec. 3. We also discovered that the DNCs are difficult to train, take a long time to converge and are prone to over-fitting and instability throughout the learning process, which were the same challenges researchers have experienced in training NTMs, which are the DNC's direct predecessors [27]. Due to the computational cost in both memory and time of training DNCs, we were not able to train a DNC on the full joint bAbI tasks, but on a joint subset of tasks that would be trainable within a reasonable time frame (3 days instead of weeks) as explained in Sec. 4.2. The DNC model was successful at performing better than or equal to the LSTM baseline on these tasks. We also ran experiments on the copy task described in [12], of which the results were highly successful and are documented in the appendix in Sec. 6.1 since this paper coult not fit all of the copy task experimentations and visualizations in addition to discussing the approach (which also served as a background on the theory of DNCs in addition to documenting our implementation process) and the full bAbI experiments, since even some of the bAbI plots and figures had to be placed in the appendix. We want to emphasize that the reader should consider the copy task section in the appendix as a continuation of the paper (even though it was placed in the appendix to save space) since given the simplicity and low RAM requirements of the copy task, more visualizations using Tensorboard was possible, so Sec. 6.1 contains many plots that can help the reader better understand DNCs, such as the visualization of DNC memory matrices in Fig. 10.

We implemented the DNC with much unit-testing in Python using Tensorflow 1.0. The experiments were on a desktop CPU and GPU, and an Azure VM with one GPU.

2. Background/related work

While neural networks (NNs) have a long history [20], they have mainly recently gained popularity due to the recent availability of large datasets and massively parallel computing power, usually in the form of GPUs, having created an environment that allowed the training of deeper and more complex architectures in a reasonable time frame. As such, deep learning techniques have received widespread attention by outperforming alternative methods (e.g. kernel machines and HMMs) on benchmark problems first in computer vi-

sion [16] and speech recognition [8] [11], and now in natural languge processing (NLP). The later by outperforming on tasks including dependency parsing [4] [2], sentiment analysis [21] and machine translation [15]. Deep learning techniques have also performed well against benchmarks in tasks that combine computer vision and NLP, such as in image-captioning [24] and lip-reading [7]. With these successes, there seems to be a shift from traditional algorithms using human engineered features and represenations to deep learning algorithms that learn the representations from raw inputs (pixels, characters) to produce the desired output (class, sequence). In the case of NLP, much of the improvements come from the sequence modeling capabilities of recurrent NNs (RNNs) [10] [23], with their ability to model long-term dependencies improved by using a gated activation function, such as in the long short-term memory (LSTM) [14] and gated-feedback unit (GRU) [5] [6] RNN architectures.

The RNN model extends conventional feedforward neural networks (FNNs) by reusing the weights at each time step (thus reducing the number of parameters to train) and conditioning the output on all previous words by a hidden "memory" state (thus "remembering" the past). In an RNN, each input is a sequence $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t, \ldots, \boldsymbol{x}_T)$ of vectors $\boldsymbol{x}_t \in \mathbb{R}^X$ and each output is the prediction sequence $(\hat{\boldsymbol{y}}_1, \ldots, \hat{\boldsymbol{y}}_t, \ldots, \hat{\boldsymbol{y}}_T)$ of vectors $\hat{\boldsymbol{y}}_t \in \mathbb{R}^Y$, usually made into a probability distribution using the softmax function. The hidden state $\boldsymbol{h}_t \in \mathbb{R}^H$ is updated at each time step as a function of a linear combination of the input $\boldsymbol{x}_t$ and the previous hidden state $\boldsymbol{h}_{t-1}$. Below, $f(\cdot)$ is usually a smooth, bounded function such as the logistic sigmoid $\sigma(\cdot)$ or hyperbolic tangent $\tanh(\cdot)$ functions

$$\boldsymbol{h}_t = f\left(\boldsymbol{W}\begin{bmatrix}\boldsymbol{x}_t \\ \boldsymbol{h}_{t-1}\end{bmatrix}\right) \qquad \hat{\boldsymbol{y}}_t = softmax(\boldsymbol{U}\boldsymbol{h}_t)$$

Theoretically, RNNs can capture any long-term dependencies in arbitrary input sequences, but in practice, training an RNN to do so is difficult since the gradients tend to vanish (to zero) or explode (to NaN, though solved by clipped gradients) [18] [13] [14]. The LSTM and GRU RNN models both address this by using gating units to control the flow of information into and out of the memories; the LSTM, in particular, have dedicated memory cells and is the baseline architecture used in our experiments. There are many LSTM formulations and this project uses the variant from [1] without the multiple layers:

$$\begin{aligned}
\boldsymbol{i}_t &= \sigma(\boldsymbol{W}_i[\boldsymbol{x}_t; \boldsymbol{h}_{t-1}] + \boldsymbol{b}_i) \\
\boldsymbol{f}_t &= \sigma(\boldsymbol{W}_f[\boldsymbol{x}_t; \boldsymbol{h}_{t-1}] + \boldsymbol{b}_f) \\
\boldsymbol{s}_t &= \boldsymbol{f}_t \circ \boldsymbol{s}_{t-1} + \boldsymbol{i}_t \circ \tanh(\boldsymbol{W}_s[\boldsymbol{x}_t; \boldsymbol{h}_{t-1}] + \boldsymbol{b}_s) \\
\boldsymbol{o}_t &= \sigma(\boldsymbol{W}_o[\boldsymbol{x}_t; \boldsymbol{h}_{t-1}] + \boldsymbol{b}_o) \\
\boldsymbol{h}_t &= \boldsymbol{o}_t \circ \tanh(\boldsymbol{s}_t)
\end{aligned}$$

where $\boldsymbol{i}_t$, $\boldsymbol{f}_t$, $\boldsymbol{s}_t$ and $\boldsymbol{o}_t$ are the input gate, forget gate, state and output gate vectors, respectively, and the $\boldsymbol{W}$'s and $\boldsymbol{b}$'s are the weight matrices and biases to be learned. Note that [1] calls the memory cell (or state) $\boldsymbol{s}_t$ instead of the $\boldsymbol{c}_t$ generally used in literature [5] [6] [11] since [1] uses the letter $\boldsymbol{c}$ to denote content vectors in the DNC equations, see Sec. 3.3.1.

With the LSTM and GRU models, longer-term dependencies can be learned, but in practice, these models have limits on how long memories can persist, which become relevant in tasks such as question answering (QA) where the relevant information for an answer could be at the very start of the input sequence, which could consist of the vectorized words of several paragraphs [19]. This is the motivation for fully differentiable (thus trainable by gradient descent) models that contain a long-term, relatively isolated memory component that the model can learn to store inputs to and read from when computing the predicted output. Such memory-based models include Neural Turing Machines [12], Memory Networks (MemNets and MemN2Ns) [26] [22], Dynamic Memory Networks (DMN) [17] and Differentiable Neural Computers (DNCs) [1], which can be viewed as a type of NTM since they were designed by the same researchers. MemNets and DMNs are based on using multiple RNNs (LSTMs/GRUs) as modules, such separate RNN modules for input and output processing as in [23]. The DMN uses a GRU as the memory component as opposed to the addressable memory in MemNets/MemN2Ns, NTMs and DNCs. While MemNets/MemN2Ns have primarily been tested on NLP task, NTMs were tested mainly on algorithmic tasks, such as copy, recall and sort, while DNCs were tested a wider variety of tasks–achieving high performance in NLP tasks on the bAbI dataset [25], algorithmic tasks such as computing shortest path on graphs and a reinforcement learning task on a block puzzle game.

Thus, DNCs are powerful models that have a long-term memory isolated from computation, which RNNs/LSTMs lack, and have the potential to replicate the capabilities of a modern day computer while being fully differentiable and thus trainable using conventional gradient-based methods, which is why they are the main topic of interest of this project.

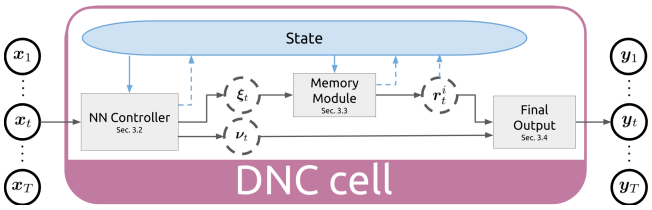
3. Approach: DNC overview & implementation

We implemented the DNC by inheriting from the Tensorflow *RNNCell* interface, which is an abstract object representing an RNN layer, even though it is called a "cell". This implementation approach takes advantage of Tensorflow's built-in unraveling capabilities so all that is needed is to program the DNC logic in $(output,\ new\_state) = self.\_\_call\_\_(inputs,\ state)$, in addition to the other, more trivial, required functions. As shown in Fig. 1, this project organizes the main DNC logic into three main module, shown in grey boxes which contain the in which section they are described.

3.1. DNC utility functions

Most of the DNC utility functions in Sec. 6.2.2 were easy to implement or already provided in Tensorflow. Others were implemented based on recognizing how they could be vectorized, such as for the content similarity based weighting

Figure 1: Our approach to implementing the DNC.



$\mathcal{C}(\boldsymbol{M}, \boldsymbol{k}, \beta)$, which results in a probability distribution vector, i.e. in $\mathcal{S}^N$. The equation in the original paper was written using cosine similarity $\mathcal{D}(\cdot, \cdot)$ as

$$\mathcal{C}(\boldsymbol{M}, \boldsymbol{k}, \beta)[i] = \frac{exp\{\mathcal{D}(\boldsymbol{k}, \boldsymbol{M}[i, \cdot])\, \beta\}}{\sum_j exp\{\mathcal{D}(\boldsymbol{k}, \boldsymbol{M}[i, \cdot])\, \beta\}} \quad \forall i = [1 : N]$$

but it could be vectorized and implemented as

$$\boldsymbol{z} = \hat{\boldsymbol{M}}\hat{\boldsymbol{k}} \qquad \mathcal{C}(\boldsymbol{M}, \boldsymbol{k}, \beta) = softmax(\beta \boldsymbol{z})$$

where $\hat{\boldsymbol{M}}$ is M with ℓ^2-normalized rows and $\hat{\boldsymbol{k}}$ is ℓ^2-normalized k. Notice that normalization (used in the cosine similarity function Sec. 6.2.2) will result in NaN errors due to division by zero since the memory matrix $\boldsymbol{M}$ is initialized to zero. This could be solve by adding a factor of $\epsilon = 1e-8$ to the denominator or inserting a condition to ignore the computation if the denominator was zero, both were implemented before the discovery of Tensorflow's built-in normalize function, which was the route taken in the end.

3.2. DNC NN Controller

We first concatenate the input vector of the current time step $\boldsymbol{x}_t$ and the read vectors from the previous time step $\boldsymbol{r}^1_{t-1}, \ldots, \boldsymbol{r}^R_{t-1}$ to form the NN controller input

$$\boldsymbol{\chi}_t = [\boldsymbol{x}_t;\ \boldsymbol{r}^1_{t-1};\ \cdots;\ \boldsymbol{r}^R_{t-1}]$$

which is then fed into the NN *cell* of the NN controller, where *cell* can be an LSTM or a FNN, which was coded using Tensorflow's default cell functions with the mathematical definitions in Sec. 6.2.4.

For consistency with the LSTM, call h_t the output of the NN *cell* at time step t, then the final NN controller's output is a matrix product with h_t:

$$\begin{bmatrix} \boldsymbol{\xi}_t \\ \boldsymbol{\nu}_t \end{bmatrix} = \boldsymbol{W}_h \boldsymbol{h}_t$$

where $\boldsymbol{W}_h$ is among the weights $\boldsymbol{\theta}$ learned by the gradient descent. Thus, if we let $\mathcal{N}$ refer to the NN controller, then we can write the NN controller as:

$$(\boldsymbol{\xi}_t, \boldsymbol{\nu}_t) = \mathcal{N}([\boldsymbol{\chi}_1;\ \cdots;\ \boldsymbol{\chi}_t];\ \boldsymbol{\theta})$$

where $[\boldsymbol{\chi}_1;\ \cdots;\ \boldsymbol{\chi}_t]$ indicates dependence on previous elements of the current sequence. In this project, after the vector concatenation, the controller is implemented using Tensorflow's default functions $rnn.BasicLSTMCell$ and $layers.dense$ for the NN *cell*, then a final weight matrix multiply and vector slicing to get the two vectors $\boldsymbol{\xi}_t$ and $\boldsymbol{\nu}_t$.

As shown in Sec. 6.2.5, the interface vector $\boldsymbol{\xi}_t$ is split into the interface parameters, some of which were processed to the desired range by the utility functions in Sec. 6.2.2. The interface parameters are then used in the memory updates as explained in Sec. 3.3. The other NN Controller output, $\boldsymbol{\nu}_t$ is used to compute the final DNC output $\boldsymbol{y}_t$ as shown in Sec. 6.2.7 in a linear combination with the read vectors $[\boldsymbol{r}^1_t;\ \ldots;\ \boldsymbol{r}^R_t]$ from the memory as shown in Sec. 3.3.1.

3.3. DNC memory module implementation

The interface parameters are used to update the memory matrix (denoted as $\boldsymbol{M} \in \mathbb{R}^{N \times W}$ where N is the number of memory locations, aka "cells", and W is the memory word size, aka "cell" size) and compute intermediate memory-computation related vectors that are used to compute the memory read vectors $[\boldsymbol{r}^t;\ \cdots;\ \boldsymbol{r}^R_t]$ that are used to both compute the output of the current time step $\boldsymbol{y}_t$ and to feed as input into the NN controller as part of $\boldsymbol{\chi}_{t+1}$ in the next time step.

In this project, the memory interactions are organized based on how these interactions can be divided into writing to memory and reading from memory, and then further subdivided based on content-based address weightings and what this project calls "history"-based address weightings, since they are computed based on previous read and write weightings. We use this notation after observing similarities between content vs "history" focusing in the DNC equations and content vs location focusing in the NTMs equations [12].

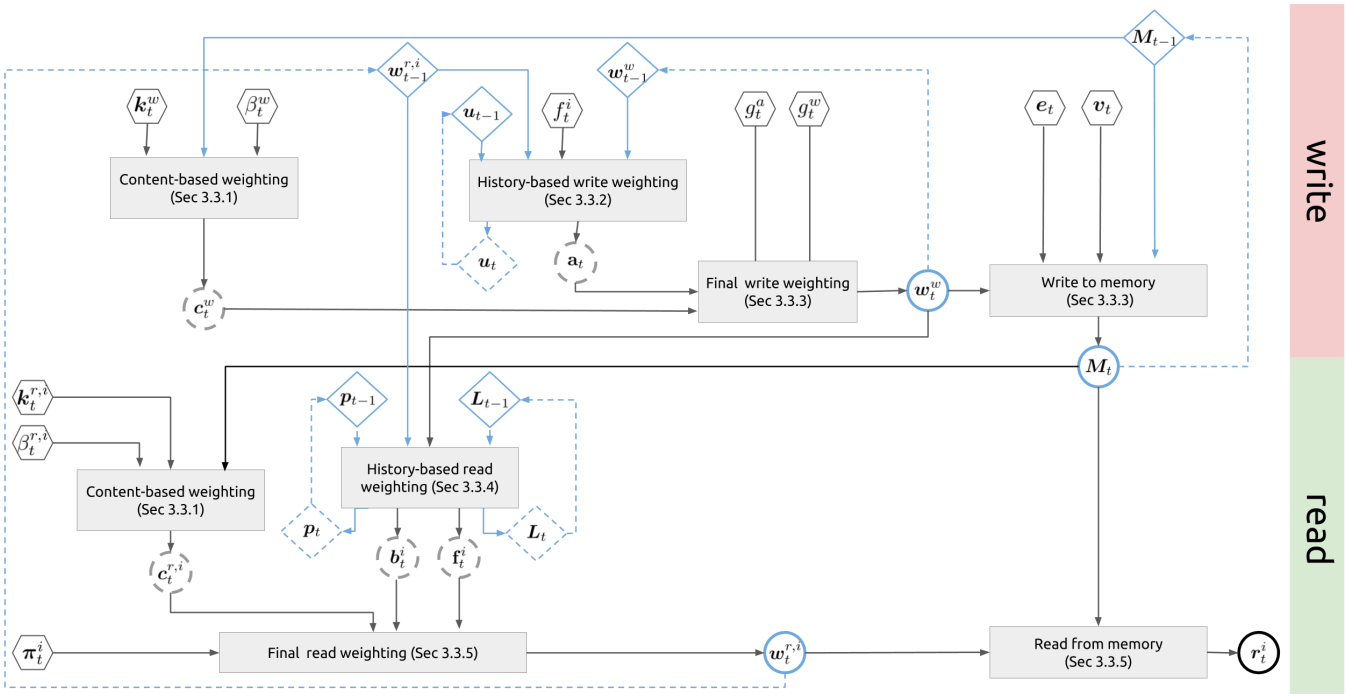
To make this part of the paper more clear, we created a diagram of this project's organization of the DNC memory implementation shown in Fig. 2. Hexagons mark the interface variables computed from the NN controller. Blue diamonds are variables that are required to be kept and updated each time step. Dashed circles imply intermediate computed values that are merely consumed and forgotten. The variables used in the figures are the same as defined in [1], with the DNC glossary in [1] reproduced in Sec. 6.2.1 for convenience, as are the memory equations from the [1], which can be viewed in Sec. 6.2.6.

Observe from Fig. 2 that the DNC is structurally similar to an LSTM except the DNC has multiple vector-storing memory cells (vs just one in an LSTM) and the DNC has addressing mechanisms to choose from which memory cell(s) to write and read.

3.3.1 Content-based write and read weightings

An advantage of memory cells with vector values is to allow for a content-based addressing, or soft attention, mechanism that can select a memory cell based on the similarity (cosine for the DNC and NTM) of its contents to a specified key $\boldsymbol{k}$ [3] [9]. Recall that the DNC uses the same content-based focusing mechanism used in NTMs [12]; the main changes between the two papers were in notation, which in the DNC is

Figure 2: Diagram of our approach to organizing and implementing the DNC memory module.



$\boldsymbol{c} = \mathcal{C}(\boldsymbol{M}, \boldsymbol{k}, \beta)$ as defined in Sec. 6.2.2 and implemented by this project as softmax over the matrix product of the normed tensors ($\boldsymbol{M}$ and $\boldsymbol{k}$) scaled by $\beta \in [0, \infty)$ as shown in Sec. 3.1. As stated in Sec. 3.1, the result $\boldsymbol{c}$ can be viewed as a probability distribution over the rows of $\boldsymbol{M}$ based on the similarity of each row (cell) to $\boldsymbol{k}$ as weighted by β. Thus, in Fig. 2, the box Content-based weighting for write is

$$\boldsymbol{c}_t^w = \mathcal{C}(\boldsymbol{M}_{t-1}, \boldsymbol{k}_t^w, \beta_t^w)$$

and the box Content-based weighting for read is

$$\boldsymbol{c}_t^{r,i} = \mathcal{C}(\boldsymbol{M}_t, \boldsymbol{k}_t^{r,i}, \beta_t^{r,i}) \quad \forall\, i \in [1:R]$$

From Fig. 2, observe that $\boldsymbol{c}_t^w$, is computed first in order to compute the final write weighting $\boldsymbol{w}_t^w$ to update (erase and write to) the memory, so $M_{t-1} \leftarrow M_t$, the update shown in the dashed lines. Then each $\boldsymbol{c}_t^{r,i}$ is used to compute each final read weighting $\boldsymbol{w}_{r,i}^w$ used to produce each read vector $\boldsymbol{r}_t^i$ from the updated memory, $\boldsymbol{M}_t$.

3.3.2 History-based write weighting

In Fig. 2, the box History-based write weighting is also k the process dubbed *dynamic memory allocation* by [1]

$$\begin{aligned}
\boldsymbol{\psi}_t &= \prod_{i=1}^{R} (\mathbb{1} - f_t^i \boldsymbol{w}_{t-1}^{r,i}) \\
\boldsymbol{u}_t &= (\boldsymbol{u}_{t-1} + \boldsymbol{w}_{t-1}^w - (\boldsymbol{u}_{t-1} \circ \boldsymbol{w}_{t-1}^w)) \circ \boldsymbol{\psi}_t \\
\boldsymbol{\phi}_t &= SortIndicesAscending(\boldsymbol{u}_t) \\
\mathbf{a}_t[\boldsymbol{\phi}_t[j]] &= (1 - \boldsymbol{u}_t[\boldsymbol{\phi}_t[j]]) \prod_{i=1}^{j-1} \boldsymbol{u}_t[\boldsymbol{\phi}_t[j]]
\end{aligned}$$

As defined in Sec. 6.2.1, we have as inputs the R free gates $f_t^i \in [0, 1]$ from the interface vector, the R previous time-step read weightings $\boldsymbol{w}_{t-1}^{r,i} \in \Delta_N$, the previous time-step write weighting $\boldsymbol{w}_{t-1}^w \in \Delta_N$ and the previous time-step usage vector $\boldsymbol{u}_t$. The output is the allocation ("history"-based) write weighting $\mathbf{a}_t \in \Delta_N$, which will be convexly combined with the content-based write weighting $\boldsymbol{c}_t^w$ for the final write weighting $\boldsymbol{w}_t^w$. The intermediate variables are the memory retention vector $\boldsymbol{\psi}_t \in [0, 1]^N$ and indices $\boldsymbol{\phi}_t \in \mathbb{N}^N$ of slots sorted by usage.

The general idea behind these equations is to bias the selection of memory cells (to erase and write to) as determined by $\mathbf{a}_t$ toward those that the DNC has recently read from ($\boldsymbol{\psi}_t$) and away from cells the DNC has recently written to ($\boldsymbol{w}_{t-1}^w$). See that $\boldsymbol{\psi}_t$ will be lower for memory slots that have been recently read, which can be analogous to indicating that a memory has been "parsed" or "consumed", and if a high free gate, determined to be insignificant, so the DNC can forget those memories and write new ones to those slots; and the converse for slots with high $\boldsymbol{\psi}_t$, indicating they should be retained (not erased). Intuitively, the usage vector $\boldsymbol{u}_t$ tracks memory cells still being "used", i.e. was used ($\boldsymbol{u}_{t-1}$), been written to ($\boldsymbol{w}_{t-1}^w$) and deemed significant (by $\boldsymbol{\psi}_t$), so retained.

The only complication in the implementation was the sorting and rearrangement. We sorted using Tensorflow's top_k, which gives the indices along with a sorted tensor. Let $\hat{\boldsymbol{u}}_t$ denote the sorted $\boldsymbol{u}_t$ and $\hat{\mathbf{a}}_t$ be the $\mathbf{a}_t$ in arrangement by $\boldsymbol{\phi}_t$. Then $(\text{-}\hat{\boldsymbol{u}}_t, \boldsymbol{\phi}_t) = top\_k(\text{-}\boldsymbol{u}_t, k = N)$ and $\hat{\mathbf{a}}_t$ is trivially computed from $\hat{\boldsymbol{u}}_t$. The main difficulty is in the rearrangement since Tensorflow tensors are immutable, one cannot preallo-

cate a tensor and then index into it to change its value. Our solution was to construct a permutation matrix by turning the indices into one-hot vectors, so if we had $\hat{\mathbf{a}}_t = [a;\ b;\ c]$ with $\boldsymbol{\phi}_t = [2;\ 3;\ 1]$, then

$$\mathbf{a}_t = \begin{bmatrix} b \\ c \\ a \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = P(\boldsymbol{\phi}_t)\hat{\mathbf{a}}_t$$

where $P(\cdot)$ turns its vector input into a permutation matrix. Note that this method may not be the most space efficient and may have added to the RAM and computational time of the DNC training process, but we could not think of a better and cleaner way of implementing this.

3.3.3 Final write weighting and memory writes

In Fig. 2, the box Final write weighting is

$$\boldsymbol{w}_t^w = g_t^w[g_t^a \mathbf{a}_t + (1 - g_t^a)\boldsymbol{c}_t^w]$$

Since $g_t^a \in [0, 1]$, observe that $g_t^a \mathbf{a}_t + (1 - g_t^a)\boldsymbol{c}_t^w$ is a convex combination of the "history"-based write weighting $\mathbf{a}_t$ and the content-based weighting $\boldsymbol{c}_t^w$, so if $g_t^a \approx 1$, then the DNC ignores the content-based weightings and vice versa in choosing which memory slots to erase/write. The write gate $g_t^w \in [0, 1]$ dictates whether to update at all, so if $g_t^w \approx 0$, then no slots will be erased/written. As shown in Fig. 2, the write weighting $\boldsymbol{w}_t^w$ is used to update the memory matrix, compute the "history"-based read weightings and then is kept as the "new" $\boldsymbol{w}_{t-1}^w$ for the next time-step.

The box Write to memory, which is

$$\boldsymbol{M}_t = \boldsymbol{M}_{t-1} \circ (\boldsymbol{E} - \boldsymbol{w}_t^w \boldsymbol{e}_t^\top) + \boldsymbol{w}_t^w \boldsymbol{v}_t^\top$$

completes the writing portion of the memory module. The erase vector $\boldsymbol{e}_t \in [0, 1]^W$ and write vector $\boldsymbol{v}_t \in \mathbb{R}^W$ were unpacked from the interface vector, see Sec. 6.2.5 and $\boldsymbol{E}$=$\{1\}^{N \times W}$, so the equation can be written without $\boldsymbol{E}$ as

$$\boldsymbol{M}_t = \boldsymbol{M}_{t-1} - \underbrace{\boldsymbol{M}_{t-1} \circ \boldsymbol{w}_t^w \boldsymbol{e}_t^\top}_{\text{erase}} + \underbrace{\boldsymbol{w}_t^w \boldsymbol{v}_t^\top}_{\text{write}}$$

so row j of $\boldsymbol{w}_t^w \boldsymbol{e}_t^\top$ is the erase vector scaled by the write weighting of memory cell j; similarly for $\boldsymbol{w}_t^w \boldsymbol{v}_t^\top$. Note that the hadamard product with $\boldsymbol{M}_{t-1}$ isolates erasing from writing since without it, empty memory slots would get written to by the erase instead of leaving a cleaner slot for the write.

These implementations are trivial, so not discussed.

3.3.4 History-based read weightings

Recall from Sec. 3.3.2 that the idea behind the history-based **write** weighting was to bias the selection of memory cells (to be erased/written to) toward those that were a combination of being most recently read from, least recently written to, or deemed inconsequential by the free gates. In contrast, the R history-based **read** weightings select memory cells (to read from) based on the order in which the cells were written to in relation to the writing time of the cells read from in the previous time-step, so the preference is for cells written to right before (as measured by backward weightings $\boldsymbol{b}_t^i \in \Delta_N$) or after (as measured by forward weightings $\mathbf{f}_t^i \in \Delta_N$) the time at which the cells the DNC just read from were written to, which is evident from the equations for the Fig. 2 box History-based read weightings:

$$\begin{aligned}
\boldsymbol{p}_t &= \left(1 - \sum_{i=1}^N \boldsymbol{w}_t^w[i]\right) \boldsymbol{p}_{t-1} + \boldsymbol{w}_t^w \\
\boldsymbol{L}_t[i, j] &= (1 - \boldsymbol{w}_t^w[i] - \boldsymbol{w}_t^w[j])\boldsymbol{L}_{t-1}[i, j] + \boldsymbol{w}_t^w[i]\boldsymbol{p}_{t-1}^w[j] \\
\boldsymbol{L}_t[i, i] &= 0 \quad \forall\, i \in [1 : N] \\
\mathbf{f}_t^i &= \boldsymbol{L}_t \boldsymbol{w}_{t-1}^{r,i} \\
\boldsymbol{b}_t^i &= \boldsymbol{L}_t^\top \boldsymbol{w}_{t-1}^{r,i}
\end{aligned}$$

where the precedence weighting $\boldsymbol{p}_t \in \Delta_N$ keeps track of the degree each memory slot was most recently written to. Observe that the DNC updates $\boldsymbol{p}_t$ based on how much writing occurred at the current time-step as measured by the current write weighting $\boldsymbol{w}_t^w \in \Delta_N$, so if $\boldsymbol{w}_t^w \approx \mathbf{0}$, then barely any writing happened at this time-step, so $\boldsymbol{p}_t \approx \boldsymbol{p}_{t-1}$, indicating that the write history is carried over; and if $\mathbb{1}^\top \boldsymbol{w}_t^w \approx 1$, the previous precedence is nearly replaces, so $\boldsymbol{p}_t \approx \boldsymbol{w}_t^w$.

Updating based on how much writing happened is also built into the recursive equations for the temporal memory link matrix $\boldsymbol{L}_t \in [0, 1]^N$, which tracks the order in which memory locations were written to such that each row and column are in Δ_N, so $\mathbb{1}^\top \boldsymbol{L}_t \leq \mathbb{1}^\top$ and $\boldsymbol{L}_t \mathbb{1} \leq \mathbb{1}$ where $\leq$ is applied element-wise. Observe that $\boldsymbol{w}_t^w[i]\boldsymbol{p}_{t-1}^w[j]$ is the amount written to memory location i at this time-step times the extent location j was written to recently, so $\boldsymbol{L}_t[i, j]$ is the extent memory slot i was written to **just after** memory slot j was written to in the previous time-step. Further observe that the more the DNC writes to either i or j, the more the DNC updates $\boldsymbol{L}_t[i, j]$, so if not much was written to those slots at the current time-step, the previous time-step links are mostly carried over. Also observe that the link matrix recursion can be vectorized as

$$\begin{aligned}
\hat{\boldsymbol{L}}_t &= \left[\boldsymbol{E} - \boldsymbol{w}_t^w \mathbb{1}^\top - \mathbb{1}(\boldsymbol{w}_t^w)^\top\right] \circ \boldsymbol{L}_{t-1} + \boldsymbol{w}_t^w (\boldsymbol{p}_{t-1}^w)^\top \\
\boldsymbol{L}_t &= \hat{\boldsymbol{L}}_t \circ (\boldsymbol{E} - \boldsymbol{I}) \qquad \text{removes self-links}
\end{aligned}$$

where $\boldsymbol{I}$ is the usual identity matrix. Note that these equations are trvially implemented with Tensorflow broadcasting, so there is no need to actually compute the two outer products of $\boldsymbol{w}_t^w$ with $\mathbb{1}$.

As explained earlier, time goes forward from columns to rows in the link matrix, so the equations for $\mathbf{f}_t^i$ (propagating forward once) and $\boldsymbol{b}_t^i$ (propagating backward once) are intuitive, as are the implementations, so neither is discussed.

3.3.5 Final read weighting and memory reads

Similar to the write weighting $\boldsymbol{w}_t^w$, the i^{th} read weighting $\boldsymbol{w}_t^{r,i}$ is a convex combination of the corresponding content-based read weighting, $\boldsymbol{c}_t^{r,i}$, and the history based read weightings, $\mathbf{f}_t^i$ and $\boldsymbol{b}_t^i$, as evident from the equations for Fig. 2 box

Final read weighting

$$\boldsymbol{w}_t^{r,i} = \boldsymbol{\pi}_t^i[1]\boldsymbol{b}_t^i + \boldsymbol{\pi}_t^i[2]\boldsymbol{c}_t^{r,i} + \boldsymbol{\pi}_t^i[3]\mathbf{f}_t^i$$

where $\boldsymbol{\pi}_t^i \in \mathcal{S}_3$ is the read mode vector that governs the extent the DNC prioritizes reading from memory slots based on the reverse order they were written ($\boldsymbol{\pi}_t^i[1]$), content similarity ($\boldsymbol{\pi}_t^i[2]$) or the order they were written ($\boldsymbol{\pi}_t^i[3]$).

As seen in Fig. 2, the i^{th} read weighting $\boldsymbol{w}_t^{r,i}$ is then passed to the Read from memory box

$$\boldsymbol{r}_t^i = \boldsymbol{M}_t^\top \boldsymbol{w}_t^{r,i}$$

thus, producing the i^{th} read vector $\boldsymbol{r}_t^i \in \mathbb{R}^W$.

These implementations are trivial, so not discussed.

3.4. DNC final output

As show in Fig 1, these R read vectors are then concatenated to compute the final DNC output $\boldsymbol{y}_t \in \mathbb{R}^Y$

$$\boldsymbol{y}_t = W_r \begin{bmatrix} \boldsymbol{r}_t^1 \\ \vdots \\ \boldsymbol{r}_t^R \end{bmatrix} + \boldsymbol{\nu}_t$$

as explained in Sec. 6.2.7. The concatenated R read vectors are also concatenated with the next input $\boldsymbol{x}_{t+1}$ to produce $\boldsymbol{\chi}_{t+1}$ to feed into the NN controller at the next time-step as explained in Sec. 3.2.

These implementations are trivial, so not discussed.

4. Experiments and Results

As explained in Sec. 3, we implemented the DNC as a Tensorflow *RNNCell* object, so the DNC can be used as one would use Tensorflow's *BasicLSTMCell*, which is also an *RNNCell*, without needing to manually implement the sequence unrolling. This meant the code used to run a DNC training session can be used to run a session for any *RNNCell* object by switching the *RNNCell* object, i.e. from the DNC to an LSTM, which was the baseline model used for all the experiments in this project. Thus, in all the experiments, an LSTM baseline model was first run, both as a reference for the DNC loss curve and as further correctness verification of the data processing and model training pipelines.

4.1. Copy experiments

At the start of the experimentation on the bAbI dataset, the results were rather poor and chaotic, so to debug and further verify that the DNC was correctly implemented, we decided to experiment with achieving high performance on the simpler copy tasks, which were described in both the DNC [1] and NTM [12] papers. The smaller and easier to visualize copy tasks allowed us to have enough RAM to visualize the memory and temporal link matrices, which helped in the fixing of bugs in the implementations. Since this project should be focused primarily on NLP, the experimentation and results from the copy tasks, which were all highly successful, are in the appendix in Sec. 6.1 due to the page limits.

4.2. bAbI experiments

The original intent of this project was to reproduce the bAbI results of [1], however, given the large and complex dataset, even though we were using the GPU, training the DNC on the full joint bAbI tasks was taking more than 15 hours complete even one epoch, which made the full 20 task joint experiment inconceivable given the time frame and limited compute power. To understand the bottlenecks, we computed statistics over the bAbI dataset as displayed in Fig. 17 and decided that some tasks were just too big, e.g. Task 3 the max input sequence lengthen of 1920, to be trained in a reasonable time frame. Therefore, we selected the smaller-sized tasks from the bAbI dataset, specifically the 6 tasks $1, 4, 9, 10, 11, 14$, for the joint-training instead of the full 20 tasks.

For all of the experiments, unless otherwise specified, the models were trained with a batch size of 16 using RMSPprop with learning rate 1e-4 and momentum 0.9; and the DNC settings were $N = 256, W = 64, R = 4$ with an LSTM controller with $H = 256$. The baseline LSTM model had the same H. The gradients were clipped by the global norm with threshold 5.

4.2.1 Data and metrics

The bAbI tasks inputs consist of word sequences interspersed with questions within self-contained "stories" and while the datasets also included supporting facts that could be used to strongly supervise the learning, we followed the DNC paper's settings and only considered the weakly supervised setting. For each story k, we constructed input sequences $\boldsymbol{x}^{(k)}$ of one-hot vectors $\boldsymbol{x}_t^{(k)} \in [0,1]^{|V|}$ where $|V|$ is the vocabulary size. We reserved a token "-" to be the signal that an answer is required and "*" to be the padding for sequences that were shorter than the maximum sequence length, which we call T. The target output vector consisted of "*" for positions that do not require answers and the vectorized word answers for the positions with "-" in the input sequence, so the maximum sequence length of $\boldsymbol{x}^{(k)}$ is also T and $\boldsymbol{x}_t^{(k)} \in [0,1]^{|V|}$. For each sequence, a mask $\boldsymbol{m}^{(k)} \in [0,1]^T$ such that $\boldsymbol{m}^{(k)}[t] = \mathbb{1}\{\text{answer required at } t\}$ was also computed and passed to the model with the input and target outputs in order to ignore the irrelevant predicted outputs at time-steps where no answers were required.

The loss was the average sotftmax cross entropy with logits loss, so the final output of both the DNC cell and the LSTM cell was passed through the softmax function, so if $\boldsymbol{h}_t$ was the cell output, then $\hat{\boldsymbol{y}}_t = softmax(\boldsymbol{h}_t)$ as defined in Sec. 6.2.2. Thus, the loss for a single prediction is

$$\mathcal{L} = \frac{1}{T}\sum_{t=1}^{T} m_t L(\boldsymbol{y}_t, \hat{\boldsymbol{y}}_t)$$

$$L(\boldsymbol{y}_t, \hat{\boldsymbol{y}}_t) = -\sum_{i=1}^{|V|} \boldsymbol{y}_t[i] \log(\hat{\boldsymbol{y}}_t[i])$$

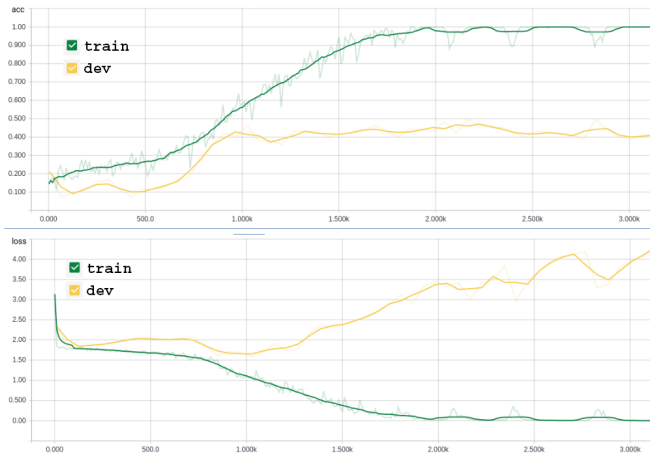
The accuracy is the average number of questions answered exactly correct, so if a question has two words in the answer, the model has to get both words in the right order to have it marked correct.

4.2.2 Single tasks

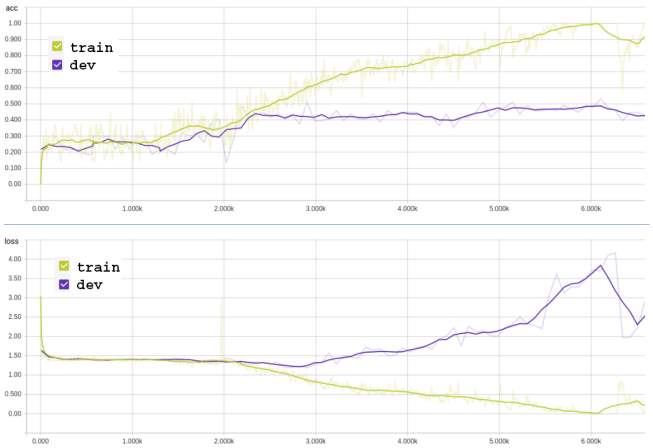
Before expending the computational power to train the DNC on the joint dataset, DNC models were trained separately on single tasks both to verify the hyper-parameters were reasonable and that the training pipeline was correctly implemented. Task 1 and task 15 were chosen for these experiments. For each of the tasks, the dev set was a randomly chosen 10% of the training set reserved for tuning; this was split ratio was also used for the joint training.

Figure 3: Plots showing the DNC overfits tiny datasets

(a) Task 1 DNC loss plots



(b) Task 15 DNC loss plots



DNC overfitted both datasets as shown in Fig. 19 and Fig. 3, but the LSTM also overfits these datasets, see Fig. 18 in the appendix, so it may be that the models are too complex for these smaller tasks. However, these experiments were still useful since a common machine learning "sanity" check is to ensure a model can overfit a tiny subset of the full dataset. Since the full dataset is $1, 4, 9, 10, 11, 14$, the size and variety of the joint dataset may prevent over-fitting in the full training step.

4.2.3 Joint task results

We trained two DNC models, each of which took over two days on the GPU. We trained a DNC model using Adam with learning rate 1e-3 instead of RMSProp and one using the settings from the DNC paper [1], so with RMPSprop with learning rate 1e-4 and momemtum 0.9, clipping the gradients by value to $[-10, 10]$ instead of by global norm. However, we kept the batchsize of 16 instead of 1, which was in [1]. An LSTM baseline model was also trained. Please see the appendix Fig. 20 for the train and dev comparison plots for all models. As can be seen in Fig. 4 and Fig. 5, the DNC does slightly better than the LSTM in terms of loss and accuracy (batch-averaged).

Figure 4: DNC vs LSTM joint-task trained models loss plots

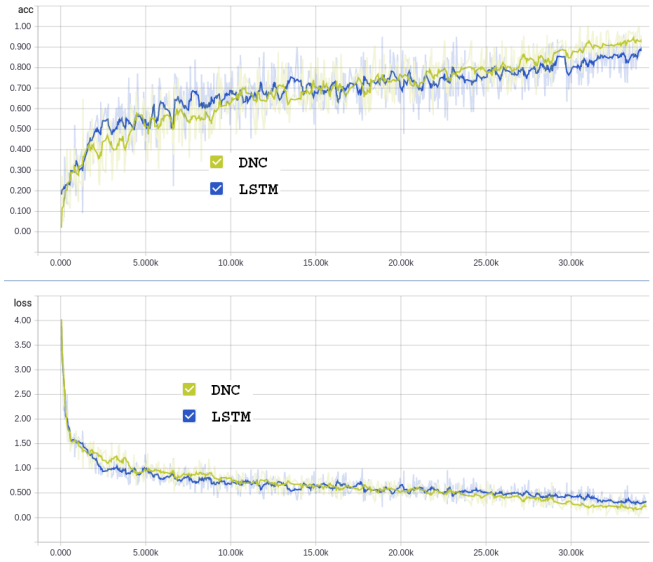


Figure 5: DNC vs LSTM on joint tasks. L=loss, A=accuracy, A*=accuracy on one batch

	Train L	Dev L	Train A*	Dev A
DNC	0.18239	0.98909	0.9091	0.68614
LSTM	0.30888	0.98909	0.8056	0.73753

The joint-trained DNC and the LSTM models were then ran on the bAbI test sets for Tasks $1, 4, 9, 10, 11, 14$ and the results as displayed in Fig. 5 show that our DNC model performs better than or equal to our LSTM baseline on all the tasks. The mean accuracy ranges as defined by the standard deviations from the DNC paper [1] for the DNC and the LSTM were also provided in addition to the weakly super-

Figure 6: DNC vs LSTM joint-task trained models per task accuracy comparision on the test set

Task	our DNC	our LSTM	[1] DNC range	[1] LSTM range	[25] LSTM
1:single-supporting-fact	0.59196	0.48744	[1.00,0.78]	[0.67,0.53]	0.50
4:two-arg-relations	0.99900	0.98699	[1.00,0.99]	[1.00,0.99]	0.61
9:simple-negation	0.82010	0.80804	[1.00,0.84]	[0.86,0.83]	0.64
10:indefinite-knowledge	0.70251	0.68945	[1.00,0.79]	[0.73,0.70]	0.44
11:basic-coreference	0.83015	0.67136	[1.00,0.91]	[0.91,0.84]	0.72
14:single-supporting-fact	0.47236	0.44623	[0.96,0.81]	[0.45,0.43]	0.27

vised LSTM model results from the bAbI paper [25] for the tasks the experiment was run.

Recall that in all our bAbI experiments, we followed the DNC paper's settings in that all the models were **weakly supervised** as the bAbI dataset paper [25] calls it, in that the models do not use the supporting facts to answer the questions in the bAbI tasks, so the models received no data other than the word sequences. The MemNN models that achieved near perfect accuracy on the bAbI dataset were using what [25] termed **strong supervision** and no results on the weakly supervised task was provided, so we could not use their numbers.

Observe that while our LSTM baseline had higher performance than the LSTM baseline from the bAbI paper [25], quite a few were not in the range of the mean and standard deviation of the DNC paper results [1] and the same held for our DNC model. We believe this is because we only had time to train our models for about 30 epochs while [1] had the computing resources to train all of their models to completion in addition to have 20 randomized models per architecture type. The comparisons may also be difficult since our models were only jointly trained on 6 out of the 20 bAbI tasks while the models in the literature were joint-trained on all 20. As stated earlier, the we had tried to train the DNC on all 20, but after running for over 15 hours, it had yet to complete even one epoch, let alone the low bar of 30 epochs we were aiming for

5. Conclusion

Thus, we have implemented DNCs and conducted experiments with them on the copy and bAbI tasks. The copy tasks were highly successful and allowed for the visualization of the DNC throughout the learning process, thus also serving as a certificate of the correctness of the DNC implementation. The bAbI tasks were more complex and therefore required more computational power than we current have, which was why a scaled down experiment was conducted that consist of the joint-training on 6 instead of the full 20 bAbI tasks, each chosen for being smaller as shown in Fig. 17 and therefore faster to train.

Given more time and computing power, we would have liked to train DNC models on the full 20 tasks and be able to iterate over the models to get better hyper-parameters. Given more RAM, we would have liked to produce the visualizations of the DNC learning process the way we did for the smaller copy tasks as shown in Sec 6.1. We would also have liked to write the code to visualize the temporal link matrix as the DNC passes through one input sequence to get a better understanding of the mechanics of the DNC. We would have also liked to do a mathematical exercise on the link matrix equations, kind of like in Sec. 6.3, to better understand the mechanics of its formulation rather than just the intuition. We would also have liked to do more experiments on the single tasks, such the the dropout experiment in Sec. 6.4.1.

In conclusion, we were very thorough in our documentation of our approach to the understanding and implementation of DNCs, along with the challenges we faced in optimizing them. We hope this project will be useful for others interested in DNCs and/or machine learning architectures with external memory.

References

[1] G. W. Alex Graves. Hybrid computing using a neural network with dynamic external memory. *Nature*, 2016.

[2] D. Andor, C. Alberti, D. Weiss, A. Severyn, A. Presta, K. Ganchev, S. Petrov, and M. Collins. Globally normalized transition-based neural networks. *CoRR*, abs/1603.06042, 2016.

[3] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

[4] D. Chen and C. D. Manning. A fast and accurate dependency parser using neural networks. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

[5] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *CoRR*, abs/1409.1259, 2014.

[6] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[7] J. S. Chung, A. W. Senior, O. Vinyals, and A. Zisserman. Lip reading sentences in the wild. *CoRR*, abs/1611.05358, 2016.

[8] G. E. Dahl, D. Yu, L. Deng, and A. Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(1):30–42, 2012.

[9] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[10] A. Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013.

[11] A. Graves, A. Mohamed, and G. E. Hinton. Speech recognition with deep recurrent neural networks. *CoRR*, abs/1303.5778, 2013.

[12] A. Graves, G. Wayne, and I. Danihelka. Neural turing machines. *CoRR*, abs/1410.5401, 2014.

[13] S. Hochreiter, Y. Bengio, and P. Frasconi. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In J. Kolen and S. Kremer, editors, *Field Guide to Dynamical Recurrent Networks*. IEEE Press, 2001.

[14] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, Nov. 1997.

[15] M. Johnson, M. Schuster, Q. V. Le, M. Krikun, Y. Wu, Z. Chen, N. Thorat, F. B. Viégas, M. Wattenberg, G. Corrado, M. Hughes, and J. Dean. Google's multilingual neural machine translation system: Enabling zero-shot translation. *CoRR*, abs/1611.04558, 2016.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[17] A. Kumar, O. Irsoy, J. Su, J. Bradbury, R. English, B. Pierce, P. Ondruska, I. Gulrajani, and R. Socher. Ask me anything: Dynamic memory networks for natural language processing. *CoRR*, abs/1506.07285, 2015.

[18] R. Pascanu, T. Mikolov, and Y. Bengio. Understanding the exploding gradient problem. *CoRR*, abs/1211.5063, 2012.

[19] B. Richardson and Renshaw. Mctest: A challenge dataset for the open-domain machine comprehension of text, 2013.

[20] J. Schmidhuber. Deep learning in neural networks: An overview. *CoRR*, abs/1404.7828, 2014.

[21] R. Socher, A. Perelygin, J. Y. Wu, J. Chuang, C. D. Manning, A. Y. Ng, and C. P. Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*, 2013.

[22] S. Sukhbaatar, a. szlam, J. Weston, and R. Fergus. End-to-end memory networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2440–2448. Curran Associates, Inc., 2015.

[23] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.

[24] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. *CoRR*, abs/1411.4555, 2014.

[25] J. Weston, A. Bordes, S. Chopra, and T. Mikolov. Towards ai-complete question answering: A set of prerequisite toy tasks. *CoRR*, abs/1502.05698, 2015.

[26] J. Weston, S. Chopra, and A. Bordes. Memory networks. *CoRR*, abs/1410.3916, 2014.

[27] W. Zhang, Y. Yu, and B. Zhou. Structured memory for neural turing machines. *CoRR*, abs/1510.03931, 2015.

6. Appendix

6.1. Copy task

Since DNCs are the descendents of NTMs, they should have the same capabilities as NTMs. In the NTM paper, the researchers came up with a copy task where given inputs of random bit-vector sequences such that the sequence lengths varied between 2 and 20, the NTMs were tasked with outputting a copy of the input after it received the entire input sequence and a delimited flag indicating the input has ended and to start the copy [12]. The trained NTMs were then given inputs of sequence length larger than the sequences on which they was trained, such as length 30, to see if the NTMs can generalize the copy algorithm, which is an experiment that was not present in the DNC paper [1] even though the researchers used the copy task to verify memory allocation and the speed of sparse link matrices. We figure copy generalization would be an interesting experiment for the DNC. In addition, the structural simplicity of the copy task also allowed for verification of the DNC implementation on a tiny copy task where the sequence lengths were fixed to be 3.

Throughout these experiments, an LSTM served as the baseline model with hidden layer size $H = 100$. The DNC models had a FNN as the LSTM as the NN controller as opposed to a LSTM so the DNC could only store the memories in its memory matrix. All models were trained using RMSProp with learning rate 1e-5 and momentum 0.9.

6.1.1 Data

We used settings from both the DNC paper [1] and the NTM paper [12] since the NTM paper had more extensive copy task experiments. As in the papers, the inputs consist of a sequence of length 6 random binary vectors, so the domain is $\{0, 1\}^6$, but since the last bit is reserved for the delimiter flag, $\boldsymbol{d}$, which tells the model to reproduce ("predict") the input sequence, the model actually receives bit vectors in $\{0, 1\}^7$. Call T the maximum sequence length, thus the model is fed inputs of sequence length 2T+1, which is the same as the target output sequence length as depicted in Fig. 7.

Figure 7: Example of $T = 3$ and $b = 6$ copy data where $\boldsymbol{x}_4$ is the delimiter, so $\boldsymbol{x}_{1:3}[1 : 6]$ is the relevant input and $\boldsymbol{y}_{5:7}[1 : 6]$ is the relevant output.

(a) Input sequence $\boldsymbol{x}$ (b) Target output sequence $\boldsymbol{y}$

Since the experiments include variable in addition to fixed sequence lengths, call T_k the sequence length of input sequence k, then the input is a (2T+1)-length sequence

$$\boldsymbol{x}^{(k)} = [\boldsymbol{x}_1;\ \ldots;\ \boldsymbol{x}_{T_k};\ \boldsymbol{d};\ \mathbf{0};\ \ldots;\ \mathbf{0}]$$

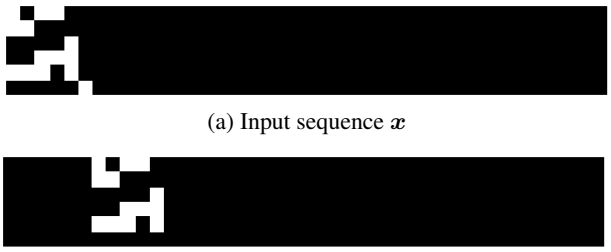
and the target output is a sequence of the same length

$$\boldsymbol{y}^{(k)} = [\mathbf{0};\ \ldots;\ \mathbf{0};\ \boldsymbol{x}_1;\ \ldots;\ \boldsymbol{x}_{T_k};\ \mathbf{0};\ \ldots;\ \mathbf{0}]$$

which is better depicted in Fig. 8. In both fixed and variable sequence length cases, a mask vector $\boldsymbol{m}^{(k)} \in \{0, 1\}^{2\text{T+1}}$ such

that $m_t^{(k)} = \mathbb{1}\{\boldsymbol{y}_t^{(k)} \text{ is relevant}\}$ is also included to be used in the loss and accuracy functions.

Figure 8: Example of $T = 20$ and $T_k = 5$ copy data where $\boldsymbol{x}_6$ is the delimiter flag.



(a) Input sequence $\boldsymbol{x}$

(b) Target output sequence $\boldsymbol{y}$

6.1.2 Loss function and metrics

We trained the model on sigmoid cross entropy loss where irrelevant outputs are masked, so the loss for prediction $[\hat{\boldsymbol{y}}_1;\ \ldots;\ \hat{\boldsymbol{y}}_{2\text{T}+1}]$, target $[\boldsymbol{y}_1;\ \ldots;\ \boldsymbol{y}_{2\text{T}+1}]$ with mask $\boldsymbol{m}$ is

$$\mathcal{L} = \frac{1}{\mathbb{1}^\top \boldsymbol{m}} \sum_{t=1}^{2\text{T}+1} m_t L(\hat{\boldsymbol{y}}_t, \boldsymbol{y}_t)$$

$$L(\hat{\boldsymbol{y}}_t, \boldsymbol{y}_t) = -\frac{1}{6} \sum_{i=1}^{2\text{T}+1} \boldsymbol{y}_t[i] \log \hat{\boldsymbol{y}}_t[i] + (1 - \boldsymbol{y}_t[i]) \log(1 - \hat{\boldsymbol{y}}_t[i])$$

The accuracy of a prediction $[\hat{\boldsymbol{y}}_1;\ \ldots;\ \hat{\boldsymbol{y}}_{2\text{T}+1}]$ is calculated based on the average number of bit matches with the target output $[\boldsymbol{y}_1;\ \ldots;\ \boldsymbol{y}_{2\text{T}+1}]$ using the mask $\boldsymbol{m}$ to ignore irrelevant portions

$$\mathcal{A} = \frac{1}{\mathbb{1}^\top \boldsymbol{m}} \sum_{t=1}^{2\text{T}+1} m_t \frac{1}{6} \sum_{i=1}^{6} \mathbb{1}\{\hat{\boldsymbol{y}}_t[i] = \boldsymbol{y}_t[i]\}$$

6.1.3 Model checks and optimization on tiny task

Due to the expensive computational requirements of training DNCs, as a further correctness check, we trained the DNC with inputs of a fixed sequence length of $T = 3$ rather than the full task of inputs with variable sequence length between 2 and 20. We used this tinier problem as a sanity check on the DNC since it should be able to get to zero loss on this problem very quickly and the tiny task was also used for hyper-parameter tuning along with testing out Tensorboard capabilities. For the tiny copy task, the DNC settings were $N = 10,\ W = 12,\ R = 1$.

As observed from the loss plots in Fig. 9, the DNC with a 1-layer FNN controller reached zero loss by step $6k$ its loss curves closely follows the LSTM baseline, but a DNC with a 2-layer FNN controller reached zero loss by step $3k$. Since it is good practice to sanity check the start loss, observe that since this is basically binary classification, the start loss should be $-\log(0.5) \simeq 0.7$, which is true for all models.

Figure 9: DNC with 2-layer FNN converges much faster than the LSTM or 1-layer FNN DNC.

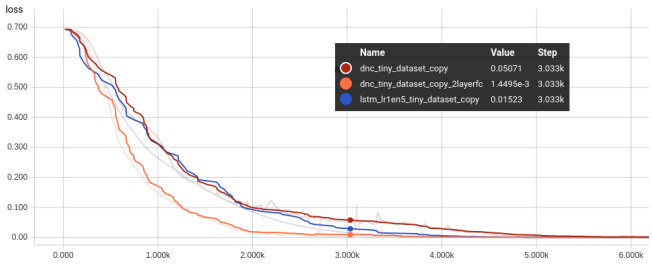
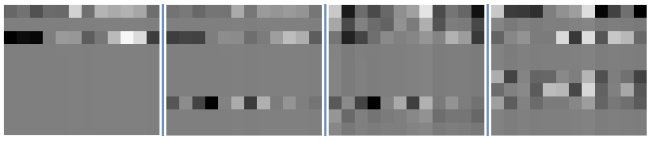
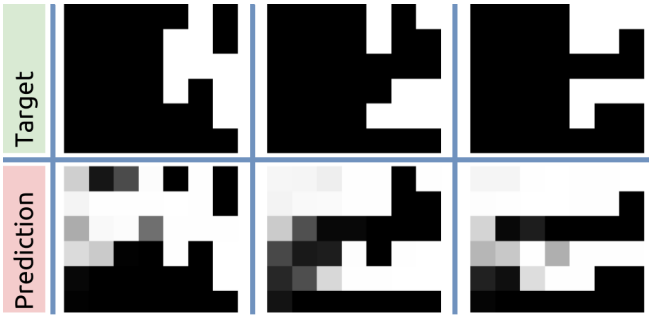


Figure 10: Examples of memory matrices throughout the training of the DNC on the fixed sequence length of $T = 3$.



See Fig. 10 for images of the memory matrix produced with Tensorboard. Note that these memory matrices are not in any order, they were just chosen to show that the DNC is writing to specific memory cells (i.e. rows of $\boldsymbol{M}$) and seems to be erasing others (see the "ghost" rows).

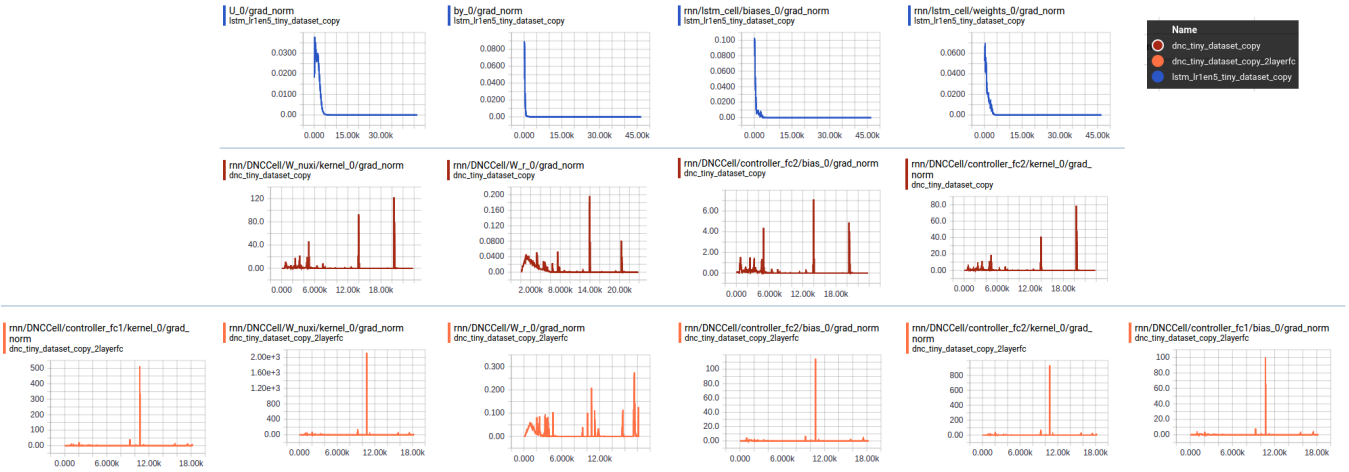
Figure 11: Three examples of prefect prediction results for the fully trained DNC with fixed sequence length $T = 3$, so only $\hat{\boldsymbol{y}}_{5:7}[1:6]$ is relevant.



See Fig. 11 for examples of sample predictions and targets produced with Tensorboard. Note that only the last three columns of each prediction sequence is relevant since DNC was receiving an input of sequence length three followed by a delimiter for the first four time-steps.

Recall that since training deep learning models is equivalent to optimizing a non-convex unconstrained optimization problem, another way to check convergence (granted that we are not experiencing vanishing gradients) is to check that the gradient norms converges to zero, which in the KKT conditions indicate convergence to a local optima. See Fig. 12 for the Tensorboard plot checking that the models have converged. Observe that the DNC models show convergence,

Figure 12: The gradient norms vs iteration for all three models for copy task on $T = 3$



but are more unstable with gradient spikes (over 2000 for the DNC) that need to be clipped to prevent the learning process from getting side-tracked by the plateaus described in [18] [13].

6.1.4 Full copy task

For the full copy task, the DNC settings were $N = 20$, $W = 12$, $R = 1$. Tensorboard was used to display images of the predictions and targets throughout the DNC learning process, which is shown in Fig. 14. For the same number of iterations, the learning process for the DNC took 8 hours and the LSTM took half an hour to training on the CPU (GPU was already being used) using RMSPprop with learning rate 1e-5 and momentum 0.9.

Figure 13: DNC vs LSTM loss plots for seq lens [2,20]

The loss plots in Fig 13 show that the DNC learns faster and better than the LSTM, but also that the DNC learning process is very unstable as there tends to be huge spikes in the loss plots even though the gradients have been clipped by the global norm, as observed in Fig. 15 where the norms do not surpass 5.

Figure 14: Sampling of prediction results from DNC throughout training on varied seq. lens in [2,20].

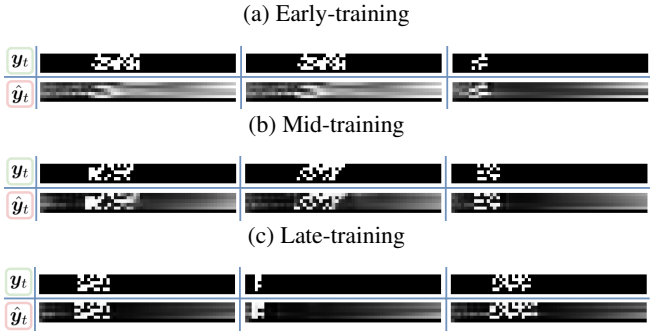


Figure 15: The gradient norms vs iteration for the 2-layer DNC trained on the variable length copy task

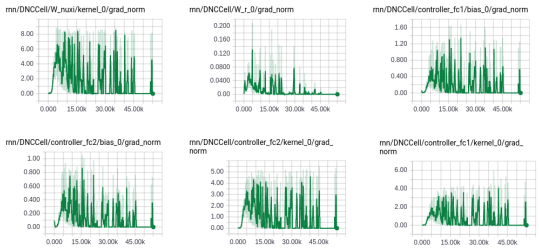


Figure 16: DNC vs LSTM results

Model	Loss on $T_k \in [2, 20]$	Acc on $T_k = 30$
LSTM	0.3248	0.66700
DNC	4.0595e-6 ≈ 0	0.9709

To test how well the models generalize the copy task, they

were tested on a batch of 100 input sequences where the sequence length was $T = 30$. The results in Fig. 16 show that the DNC can better generalize the copy task than the LSTM, which was a confirmation of our hypothesis.

6.2. DNC equations and definitions from [1]

6.2.1 Glossary

$$\Delta_N = \{\boldsymbol{\alpha} \in \mathbb{R}^N \mid \boldsymbol{\alpha}_i \in [0,1], \mathbb{1}^\top \boldsymbol{\alpha} \leq 1\}$$
$$\mathcal{S}_N = \{\boldsymbol{\alpha} \in \mathbb{R}^N \mid \boldsymbol{\alpha}_i \in [0,1], \mathbb{1}^\top \boldsymbol{\alpha} = 1\}$$

Name	Description	Domain
t	time-step	$\mathbb{N}$
N	number of memory locations	$\mathbb{N}$
W	memory word size	$\mathbb{N}$
R	number of read heads	$\mathbb{N}$
$\mathbf{x}_t$	input vector	$\mathbb{R}^X$
$\mathbf{y}_t$	output vector	$\mathbb{R}^Y$
$\mathbf{z}_t$	target vector	$\mathbb{R}^Y$
$\mathbf{M}_t$	memory matrix	$\mathbb{R}^{N\times W}$
$\mathbf{k}_t^{r,i}$	read key i $(1 \leq i \leq R)$	$\mathbb{R}^W$
$\mathbf{r}_t^i$	read vector i	$\mathbb{R}^W$
$\beta_t^{r,i}$	read strength i	$[1,\infty)$
$\mathbf{k}_t^w$	write key	$\mathbb{R}^W$
β_t^w	write strength	$[1,\infty)$
$\mathbf{e}_t$	erase vector	$[0,1]^W$
$\mathbf{v}_t$	write vector	$\mathbb{R}^W$
f_t^i	free gate i	$[0,1]$
g_t^a	allocation gate	$[0,1]$
g_t^w	write gate	$[0,1]$
$\boldsymbol{\psi}_t$	memory retention vector	$\mathbb{R}^N$
$\mathbf{u}_t$	memory usage vector	$\mathbb{R}^N$
$\boldsymbol{\phi}_t$	indices of slots sorted by usage	$\mathbb{N}^N$
$\mathbf{a}_t$	allocation weighting	Δ_N
$\mathbf{c}_t^w$	write content weighting	$\mathcal{S}_N$
$\mathbf{w}_t^w$	write weighting	Δ_N
$\mathbf{p}_t$	precedence weighting	Δ_N
$\mathbf{E}$	matrix of ones $(\mathbf{E}[i,j] = 1\ \forall i,j)$	$\mathbb{R}^{N\times W}$
$\mathbf{L}_t$	temporal link matrix	$\mathbb{R}^{N\times N}$
$\mathbf{f}_t^i$	forward weighting i	Δ_N
$\mathbf{b}_t^i$	backward weighting i	Δ_N
$\mathbf{c}_t^{r,i}$	read content weighting i	$\mathcal{S}_N$
$\mathbf{w}_t^{r,i}$	read weighting i	Δ_N
$\boldsymbol{\pi}_t^i$	read mode i	$\mathcal{S}_3$
W_r	read key weights	$\mathbb{R}^{(RW)\times Y}$
$\boldsymbol{\theta}$	controller weights	$\mathbb{R}^\Theta$
$\boldsymbol{\xi}_t$	interface vector	$\mathbb{R}^{(W\times R)+3W+5R+3}$
$\boldsymbol{\chi}_t$	controller input vector	$\mathbb{R}^{(W\times R)+X}$
$\boldsymbol{\upsilon}_t$	controller output vector	$\mathbb{R}^Y$
$\mathcal{N}(.;\boldsymbol{\theta})$	controller network	

6.2.2 Definitions

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad \in [0,1]$$
$$oneplus(x) = 1 + \log(1+e^x) \in [1,\infty)$$
$$softmax(\boldsymbol{x})_i = \frac{e^{\boldsymbol{x}_i}}{\sum_{j=1}^{|\boldsymbol{x}|} e^{\boldsymbol{x}_j}} \qquad \in [0,1]$$
$$\mathcal{C}(\boldsymbol{M},\boldsymbol{k},\beta)[i] = \frac{exp\{\mathcal{D}(\boldsymbol{k},\boldsymbol{M}[i,\cdot])\,\beta\}}{\sum_j exp\{\mathcal{D}(\boldsymbol{k},\boldsymbol{M}[i,\cdot])\,\beta\}} \qquad \in \mathcal{S}^N$$
$$\mathcal{D}(\boldsymbol{u},\boldsymbol{v}) = \frac{\boldsymbol{u}\cdot\boldsymbol{v}}{|\boldsymbol{u}|\,|\boldsymbol{v}|} \qquad \text{cosine similarity}$$
$$(\boldsymbol{A}\circ\boldsymbol{B})[i,j] = \boldsymbol{A}[i,j]\,\boldsymbol{B}[i,j] \qquad \text{hadamard product}$$
$$(\boldsymbol{x}\circ\boldsymbol{y})[i,j] = \boldsymbol{x}[i]\,\boldsymbol{y}[i] \qquad \text{hadamard product}$$

6.2.3 Initial Conditions

$$\boldsymbol{u_0} = 0;\ \boldsymbol{p_0} = 0;\ \boldsymbol{L_0} = 0;\ \boldsymbol{L_t[i,i]} = 0, \quad \forall i$$

6.2.4 Controller Update

$$\boldsymbol{\chi}_t = [\boldsymbol{x}_t;\ \boldsymbol{r}_{t-1}^1;\ \cdots;\ \boldsymbol{r}_{t-1}^R]$$
$$(\boldsymbol{\xi}_t, \boldsymbol{\nu}_t) = \mathcal{N}([\boldsymbol{\chi}_1;\ \cdots;\ \boldsymbol{\chi}_t];\ \boldsymbol{\theta})$$

where $\mathcal{N}(\cdot)$ is the neural network based controller that consists of a *cell* that in this project was either a FNN or LSTM that outputs $\boldsymbol{h}_t$ as a function of the input $\boldsymbol{\chi}_t$, and $\boldsymbol{h}_{t-1}$ if an LSTM. So if *cell* is a FNN, it has the form:

$$\boldsymbol{h}_t = relu(\boldsymbol{W}_\chi \boldsymbol{\chi}_t + \boldsymbol{b}_\chi)$$

otherwise, *cell* is the LSTM of the form:

$$\begin{aligned}
\boldsymbol{i}_t &= \sigma(\boldsymbol{W}_i[\boldsymbol{\chi}_t; \boldsymbol{h}_{t-1}] + \boldsymbol{b}_i)\\
\boldsymbol{f}_t &= \sigma(\boldsymbol{W}_f[\boldsymbol{\chi}_t; \boldsymbol{h}_{t-1}] + \boldsymbol{b}_f)\\
\boldsymbol{s}_t &= \boldsymbol{f}_t \circ \boldsymbol{s}_{t-1} + i_t \circ \tanh(\boldsymbol{W}_s[\boldsymbol{\chi}_t; \boldsymbol{h}_{t-1}] + \boldsymbol{b}_s)\\
\boldsymbol{o}_t &= \sigma(\boldsymbol{W}_o[\boldsymbol{\chi}_t; \boldsymbol{h}_{t-1}] + \boldsymbol{b}_o)\\
\boldsymbol{h}_t &= \boldsymbol{o}_t \circ \tanh(\boldsymbol{s}_t)
\end{aligned}$$

A linear operation is used to get $(\boldsymbol{\xi}_t, \boldsymbol{\nu}_t)$:

$$\begin{bmatrix}\boldsymbol{\xi}_t\\ \boldsymbol{\nu}_t\end{bmatrix} = \boldsymbol{W}_h \boldsymbol{h}_t$$

6.2.5 Interface ($\boldsymbol{\xi}_t$) unpacking

Split the vector $\boldsymbol{\xi}_t \in \mathbb{R}^{(W\cdot R)+3W+5R+3}$ into the following components, then use the utility functions in Sec. 6.2.2 to preprocess some of the components.

$$\boldsymbol{\xi}_t = \begin{bmatrix}
\boldsymbol{k}_t^{r,1}\\ \vdots\\ \boldsymbol{k}_t^{r,R} & \} \text{ read keys } \boldsymbol{k}_t^{r,i} \in \mathbb{R}^W\\
\hat{\beta}_t^{r,1}\\ \vdots\\ \hat{\beta}_t^{r,R} & \} \text{ read strengths } \beta_t^{r,i} = oneplus(\hat{\beta}_t^{r,i}) \in \mathbb{R}\\
\boldsymbol{k}_t^w & \} \text{ write key } \boldsymbol{k}_t^w \in \mathbb{R}^W\\
\hat{\beta}_t^w & \} \text{ write strength } \beta_t^w = oneplus(\hat{\beta}_t^w) \in \mathbb{R}\\
\hat{\boldsymbol{e}}_t & \} \text{ erase vector } \boldsymbol{e}_t = \sigma(\hat{\boldsymbol{e}}_t) \in \mathbb{R}^W\\
\boldsymbol{v}_t & \} \text{ write vector } \boldsymbol{v}_t \in \mathbb{R}^W\\
\hat{f}_t^1\\ \vdots\\ \hat{f}_t^R & \} \text{ free gates } f_t^i = \sigma(\hat{f}_t^i) \in \mathbb{R}\\
\hat{g}_t^a & \} \text{ allocation gate } g_t^a = \sigma(\hat{g}_t^a) \in \mathbb{R}\\
\hat{g}_t^w & \} \text{ write gate } g_t^w = \sigma(\hat{g}_t^w) \in \mathbb{R}\\
\hat{\boldsymbol{\pi}}_t^1\\ \vdots\\ \hat{\boldsymbol{\pi}}_t^R & \} \text{ read modes } \boldsymbol{\pi}_t^i = softmax(\hat{\boldsymbol{\pi}}_t^i) \in \mathbb{R}^3
\end{bmatrix}$$

6.2.6 Memory Updates

The original paper equations:

$$
\begin{aligned}
\boldsymbol{\psi}_t &= \prod_{i=1}^{R} (\mathbb{1} - f_t^i \boldsymbol{w}_{t-1}^{r,i}) \\
\boldsymbol{u}_t &= (\boldsymbol{u}_{t-1} + \boldsymbol{w}_{t-1}^w - (\boldsymbol{u}_{t-1} \circ \boldsymbol{w}_{t-1}^w)) \circ \boldsymbol{\psi}_t \\
\boldsymbol{\phi}_t &= SortIndicesAscending(\boldsymbol{u}_t) \\
\mathbf{a}_t[\boldsymbol{\phi}_t[j]] &= (1 - \boldsymbol{u}_t[\boldsymbol{\phi}_t[j]]) \prod_{i=1}^{j-1} \boldsymbol{u}_t[\boldsymbol{\phi}_t[j]] \\
\boldsymbol{c}_t^w &= \mathcal{C}(\boldsymbol{M}_{t-1}, \boldsymbol{k}_t^w, \beta_t^w) \\
\boldsymbol{w}_t^w &= g_t^w [g_t^a \mathbf{a}_t + (1 - g_t^a) \boldsymbol{c}_t^w] \\
\boldsymbol{M}_t &= \boldsymbol{M}_{t-1} \circ (\boldsymbol{E} - \boldsymbol{w}_t^w \boldsymbol{e}_t^\top) + \boldsymbol{w}_t^w \boldsymbol{v}_t^\top \\
\boldsymbol{p}_t &= \left(1 - \sum_{i=1}^{N} \boldsymbol{w}_t^w[i]\right) \boldsymbol{p}_{t-1} + \boldsymbol{w}_t^w \\
\boldsymbol{L}_t[i,j] &= (1 - \boldsymbol{w}_t^w[i] - \boldsymbol{w}_t^w[j]) \boldsymbol{L}_{t-1}[i,j] + \boldsymbol{w}_t^w[i] \boldsymbol{p}_{t-1}^w[j] \\
\mathbf{f}_t^i &= \boldsymbol{L}_t \boldsymbol{w}_{t-1}^{r,i} \\
\boldsymbol{b}_t^i &= \boldsymbol{L}_t^\top \boldsymbol{w}_{t-1}^{r,i} \\
\boldsymbol{c}_t^{r,i} &= \mathcal{C}(\boldsymbol{M}_t, \boldsymbol{k}_t^{r,i}, \beta_t^{r,i}) \\
\boldsymbol{w}_t^{r,i} &= \boldsymbol{\pi}_t^i[1] \boldsymbol{b}_t^i + \boldsymbol{\pi}_t^i[2] \boldsymbol{c}_t^{r,i} + \boldsymbol{\pi}_t^i[3] \mathbf{f}_t^i \\
\boldsymbol{r}_t^i &= \boldsymbol{M}_t^\top \boldsymbol{w}_t^{r,i}
\end{aligned}
$$

6.2.7 Output

The final output is a linear combination of the vector $\boldsymbol{\nu}_t$ from the controller and the concatenated read vectors $\boldsymbol{r}_t^1, \ldots, \boldsymbol{r}_t^R$.

$$
\boldsymbol{y}_t = W_r \begin{bmatrix} \boldsymbol{r}_t^1 \\ \vdots \\ \boldsymbol{r}_t^R \end{bmatrix} + \boldsymbol{\nu}_t
$$

6.3. DNC mathematical exercise

In most of the DNC equations, why the authors of the DNC paper [1] formulated the expressions they way they did was intuitive, but some of the equations were not quite so. In our understanding of the DNC equations, we went through some proofs to ensure we understood the "why" behind the mathematical expressions, particularly for the usage vector equation, which, through our mathematical exercise, we think was formulated to ensure $\boldsymbol{u}_t \in [0,1]^N$. To see that $\boldsymbol{u}_t = (\boldsymbol{u}_{t-1} + \boldsymbol{w}_{t-1}^w - (\boldsymbol{u}_{t-1} \circ \boldsymbol{w}_{t-1}^w)) \circ \boldsymbol{\psi}_t \in [0,1]^N$, observe that $a + b - ab \in [0,1]$ if $a, b \in [0,1]$.

$$
\begin{aligned}
a + b - ab &= a + b - ab - 1 + 1 \\
&= (1 - b)a + b - 1 + 1 \\
&= (1 - b)a - (1 - b) + 1 \\
&= (1 - b)(a - 1) + 1
\end{aligned}
$$

Observe that since $a, b \in [0,1]$, $-1 \leq a - 1 \leq 0$ and $0 \leq 1 - b \leq 1$, so $-1 \leq (1 - b)(a - 1) \leq 0$, which implies $0 \leq (1 - b)(a - 1) + 1 \leq 1$. Thus, by the same logic, $\boldsymbol{u}_{t-1} + \boldsymbol{w}_{t-1}^w - (\boldsymbol{u}_{t-1} \circ \boldsymbol{w}_{t-1}^w) \in [0,1]$ and since $\boldsymbol{\psi}_t \in [0,1]$, we must have $\boldsymbol{u}_t \in [0,1]$.

6.4. bAbI experiments, supplementary info

Figure 17: bAbI data tasks statistics.

Task	vocab_size	maxlenX	max#Q
qa1_single-supporting-fact	23.00	93.00	5.00
qa2_two-supporting-facts	37.00	582.00	5.00
qa3_three-supporting-facts	38.00	1920.00	5.00
qa4_two-arg-relations	18.00	24.00	1.00
qa5_three-arg-relations	43.00	820.00	5.00
qa6_yes-no-questions	39.00	191.00	5.00
qa7_counting	47.00	361.00	5.00
qa8_lists-sets	39.00	388.00	5.00
qa9_simple-negation	27.00	109.00	5.00
qa10_indefinite-knowledge	28.00	124.00	5.00
qa11_basic-coreference	30.00	101.00	5.00
qa12_conjunction	24.00	112.00	5.00
qa13_compound-coreference	30.00	111.00	5.00
qa14_time-reasoning	29.00	156.00	5.00
qa15_basic-deduction	21.00	72.00	4.00
qa16_basic-induction	21.00	47.00	1.00
qa17_positional-reasoning	22.00	128.00	8.00
qa18_size-reasoning	22.00	249.00	9.00
qa19_path-finding	27.00	53.00	1.00
qa20_agents-motivations	39.00	161.00	12.00

Figure 18: Loss plots showing LSTM ovefitting Task 1 where dev acc was 0.458 and dev loss was 3.15647

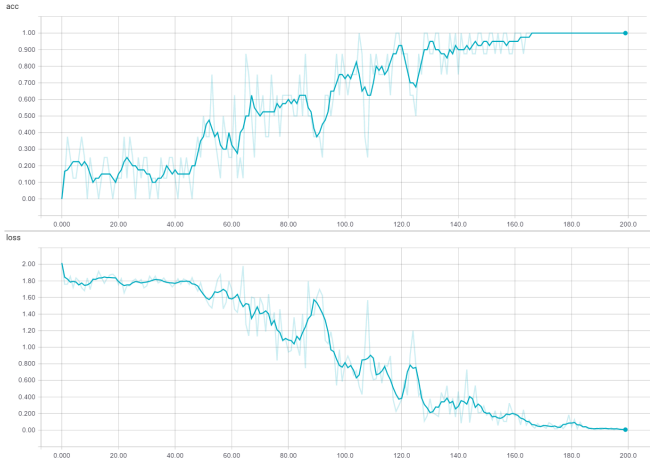
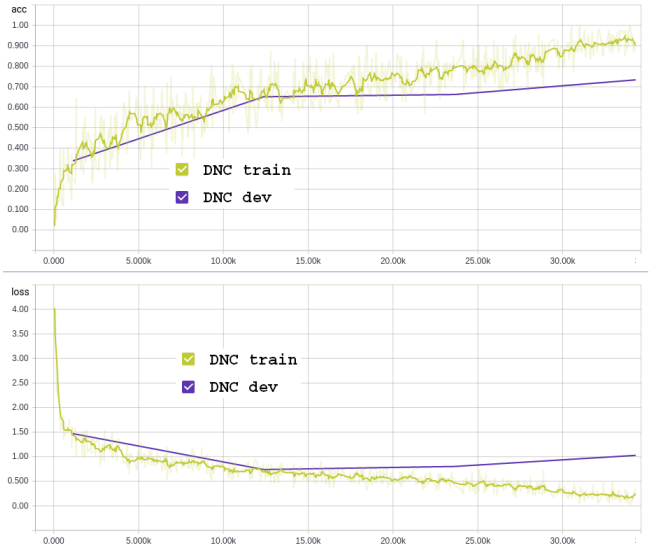


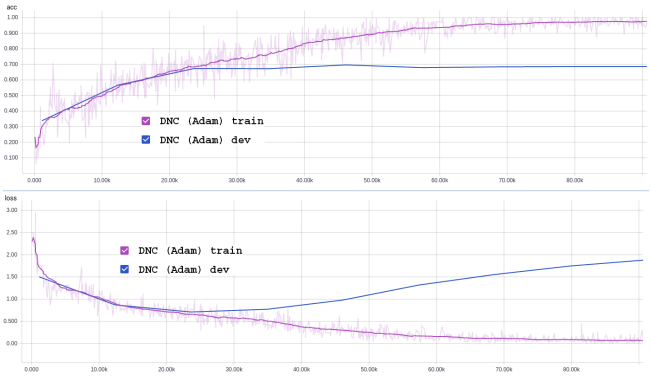
Figure 19: DNC overfitting on single task experiments

	Train Loss	Dev loss	Train Acc	Dev Acc
Task 1	0.07310	3.376	1.0	0.4316
Task 15	0.01135	3.731	1.0	0.4792

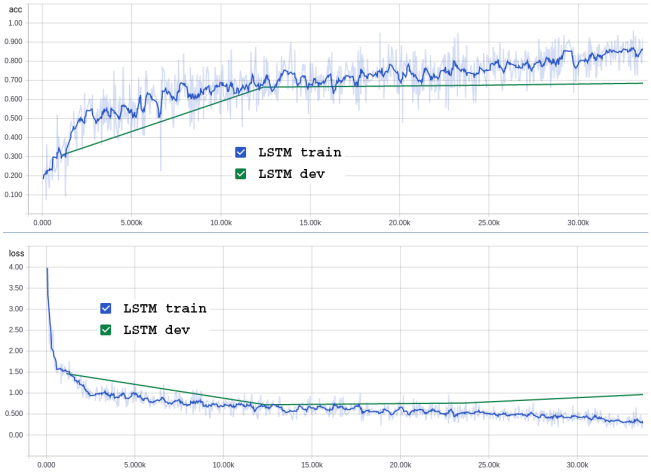
Figure 20: Joint-task models: Training vs dev set loss and accuracy plots



(a) DNC with [1] settings



(b) DNC using Adam: notice the overfitting.



(c) LSTM

6.4.1 Dropout experiment (after joint models)

While writing the conclusion, we had left-over GPU time, so due to the over-fitting on the single tasks, we also ran a dropout experiment to understand what we would have done if we had more time, but the dropout experiment was only on a single task due to computational and temporal limitations. We were hypothesizing that the introduction of regularization would help the model better generalize to unseen data. The dropout experiment was only executed on Task 1 and the results were somewhat promising as shown in Fig. 21 since, like in the experiment without dropout, the accuracy on the dev set starts flat-lining at about 0.5, but the difference between the dev and train loss plots were less drastic. However, the train loss for the DNC with dropout was higher than the DNC alone on all iterations and the DNC with dropout was taking more than twice as long to train to get to a higher loss than the DNC without dropout.

Figure 21: DNC trained on Task 1 with Dropout