DNC Implementation

Dylan Flaute

December 2017

1 Introduction

The differentiable neural computer (DNC) is a recursive neural network. It features multiple memory vectors, unlike the LSTM. In abstract, the DNC consists of a **controller** and a **memory module**. The memory module consists of a collection of write heads that alter memory content and a collection of read heads that access memory content. The controller concatenates an input sequence with the information the DNC accessed at the end of the previous time-step. This is the primary recursive step. The controller processes this information to produce a "pre-output" and an interface vector. The **interface vector** is a set of controls for the memory module. It contains the information to write or erase from memory slots, the content to compare to the slots in memory, and controls for "how much" the heads should act. The write heads adjust the information in memory before the read heads collect information from memory. That information is then linearly transformed into an appropriate shape and added with the pre-output to form the final output.

Let us define some integers to track the shape of our model. Let W be the size of a "bit" or **slot** in memory, and let N be the number of memory locations. Let H be the number of **write heads**, or opportunities to write memory per time-step. Each write head has a set of actions to carry out, which are partially determined by the interface vector and partially determined by recurrence relationships. These heads have mechanisms to consider where other heads are writing. Let R be the number of read heads. Each **read head** has a set of reading instructions to carry out based on recurrence relationships and commands from the interface vector. Each read head acts independently of the others. Finally, let B denote batch size. The DNC operates completely independently on each batch and recurrently over a sequence of inputs. We provide batched dimensions for completeness, but still refer to three-dimensional tensors as matrices. At the end of the sequence, the DNC returns a sequence prediction. Then, it reinitializes its states (unless specifically operating as stateful) and operates on the next batch of sequences.

2 State

Before we discuss implementation, let us preempt with the non-trained variables that define the DNC state. These are all necessary for the recurrence relationships we previously mentioned. We have

- $\mathbf{u}_0 \in [0,1]^{B\times N}$, initialized to zeros. This collection of **usage vectors** tracks how much a memory slot is used.
- $\mathbf{w}_0^{w,j} \in [0,1]^{B\times N\times 1_j}\ \forall 1 \leq j \leq H$, initialized to $1e-6$. The **write weights** determine a weighting over the N slots in memory for each write head h.
- $M_0 \in \mathbb{R}^{B\times N\times W}$, initialized to zeros. This **memory matrix** holds the N vectors of length W that are stored by our model.
- $L_0 \in [0,1]^{B\times H\times N\times N}$, initialized to zeros. The **temporal linkage matrix** for some write head tracks the order information was written. Element h,k is high if slot h was written after slot k.
- $\mathbf{p}_0^j \in [0,1]^{B\times N\times 1_j}\ \forall 1 \leq j \leq H$, initialized to zeros. This collection of **precedence vectors** is responsible for tracking memory slots to which write head j just wrote.
- $\mathbf{w}_0^{r,i} \in [0,1]^{B\times N\times 1_i}\ \forall 1 \leq i \leq R$, initialized to $1e-6$. The **read weights** determine a weighting over the N slots in memory for the operation of each read head i.
- $\mathbf{r}_0^i \in \mathbb{R}^{B\times 1_i\times N}\ \forall 1 \leq i \leq R$, initialized randomly. The **read vectors** carry the information that was read from memory by head i at time-step t.

3 Controller

Implement whatever neural network you want that does the following:

$$\text{ctrlr}([\mathbf{x}_t; \mathbf{r}_{t-1}^1; ...; \mathbf{r}_{t-1}^R]) \mapsto \mathbf{a}_t^y$$

where $[\cdot;\cdot]$ denotes concatenation of vectors along the long dimension, e.g. two column vectors becomes one, longer column vector. Note that, in practice, the read vectors r_{t-1}^i will be in a matrix of shape $B \times R \times W$. In other words, we do not keep track of R separate vectors; rather, each read vector "lives" in a row of a matrix. We refer to this matrix as a **collection** of read vectors. We hold several other vectors in such a collection, although generally we use the last dimension, columns. We denote the ith element of a collection as, for example, $\mathbf{r}_t^i$ and the entire collection as r_t. Finally, the controller returns $\mathbf{a}_t^y$, a vector whose length equals the length of the final time-step output vector, ℓ_y, summed with

$$\ell_\zeta = (R \times W) + (H \times W) + 2R + 3H + 2(W \times H) + (R \times (2H + 1)).$$

This is the length of the interface vector. When we discuss its partitioning, this length will make more sense.

We perform two learned linear transformations on $\mathbf{a}_t^y$. We transform $\mathbf{a}_t^y$ into an interface vector ζ_t that controls memory operations and into a "pre-output" $\hat{\mathbf{y}}_t$. Let

$$\zeta_t = \mathbf{a}_t^t W^\zeta,$$
$$\hat{\mathbf{y}}_t = \mathbf{a}_t^y W^y,$$

where W^ζ is of size $(\ell_\zeta + \ell_y) \times \ell_\zeta$ and W^y is of size $(\ell_\zeta + \ell_y) \times \ell_y$. Note that these matrices do not train at every time-step. They are trained after the network has seen a full sequence, in typical RNN fashion. Hence, they have no subscript t.

4 Memory

Now we look into the memory operation at time-step t. This is the heart of the differentiable neural computer. Recall that we have H write heads and R read heads. We define a weighting over the N memory locations for each of the H write heads. This acts as a probability distribution for altering a memory location. Non-used locations are generally preferred for writing. The write heads avoid writing to the same location. Similarly, we define a read weighting over the memory locations for each of the R read heads that determines the probability for reading from the locations. The read heads prefer recently used locations. We produce a read (past tense) vector for each head and use that information to augment our "pre-output" from the controller.

4.1 Interface

Since we left-multiply the matrix W^ζ by the vector a_t^y to obtain ζ_t, we expect that ζ_t is of shape $B \times \ell_\zeta \times 1$. We are going to split ζ_t at various points along the ℓ_ζ axis. Using i as an index along the R dimension to denote a vector for each read head, and j as an index along the H axis, we partition as follows:

- First WR elements $\to \mathbf{k}_t^{r,i}$, collected in shape $B \times W \times R$,
- Next R elements $\to (\beta_t^{r,i})'$, collected in shape $B \times 1 \times R$,
- Next WH elements $\to \mathbf{k}_t^{w,j}$, collected in shape $B \times W \times H$,
- Next H elements $\to (\beta_t^{w,j})'$, collected in shape $B \times 1 \times H$,
- Next WH elements $\to \hat{\mathbf{e}}_t^j$, collected in shape $B \times W \times H$,
- Next WH elements $\to \mathbf{v}_t^j$, collected in shape $B \times W \times H$,
- Next R elements $\to (f_t^{r,i})'$, collected in shape $B \times 1 \times R$,

- Next H elements $\rightarrow (g_t^{a,j})'$, collected in shape $B \times 1 \times H$,
- Next H elements $\rightarrow (g_t^{w,j})'$, collected shape $B \times 1 \times H$,
- Next $R(2H+1)$ elements $\rightarrow \hat{\pi}_t^i$, collected in shape $B \times (2H+1) \times R$.

What do all those primes and hats mean? Read, "unscaled". We use $\hat{\cdot}$ to denote an version of a vector outside the original defined domain and $\cdot'$ to denote a version of a scalar. For many of these interface parameters, we wish to restrict their domain. Let us consider some functions: Let softmax : $\mathbb{R}^n \mapsto \mathrm{S}_n$ be defined as

$$\mathrm{softmax}(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{i=1}^{K} e^{z_i}}.$$

This maps a vector to a vector of equal length with entries in $[0, 1]$ that sum up to 1. We denote this space S_n. Let us denote the sigmoid function $\sigma : \mathbb{R} \mapsto [0, 1]$ be an element-wise operator such that

$$\sigma(\mathbf{z})_i = \frac{1}{1 + e^{-z_i}}.$$

And let oneplus : $\mathbb{R} \mapsto [1, \infty)$ be an element-wise operator such that

$$\mathrm{oneplus}(\mathbf{z})_i = \ln(e^{z_i} + 1) + 1 = \mathrm{softplus}(\mathbf{z})_i + 1.$$

With these definitions, we may scale our values into the appropriate domain:

- $\beta_t^{r,i} = \mathrm{oneplus}((\beta_t^{r,i})')$, $\beta_t^{w,j} = \mathrm{oneplus}((\beta_t^{w,j})')$,
- $\mathbf{e}_t^j = \sigma(\hat{\mathbf{e}}_t^j)$, $f_t^{r,i} = \sigma((f_t^{r,i})')$, $g_t^{a,j} = \sigma((g_t^{a,j})')$, $g_t^{w,j} = \sigma((g_t^{w,j})')$,
- $\pi_t^i = \mathrm{softmax}(\hat{\pi}_t^i)$ on R dimension, so π_t^i is a probability distribution for all i.

4.2 Write Heads

Each of the H write heads has specific information to erase, $\mathbf{e}_t^j$, and to write, $\mathbf{v}_t^j$. How does it decide where to put the information assigned to it, or if it should put it anywhere? The attention mechanism partially depends on the interface gates. It also depends on the similarity between the content keys $\mathbf{k}_t^{w,j}$ and each slot in memory, as well as recurrence relationships.

4.2.1 Usage

To update the usage, we must consider the previous write weights, $\mathbf{w}_{t-1}^w$, a $B \times N \times H$ tensor, and the previous usage, $\mathbf{u}_{t-1}$, a $B \times N$ tensor. The read weights $\mathbf{w}_{t-1}^{r,i}$ are of shape $B \times N \times R$. To calculate the **retention** of a memory slot during the reading process, let

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

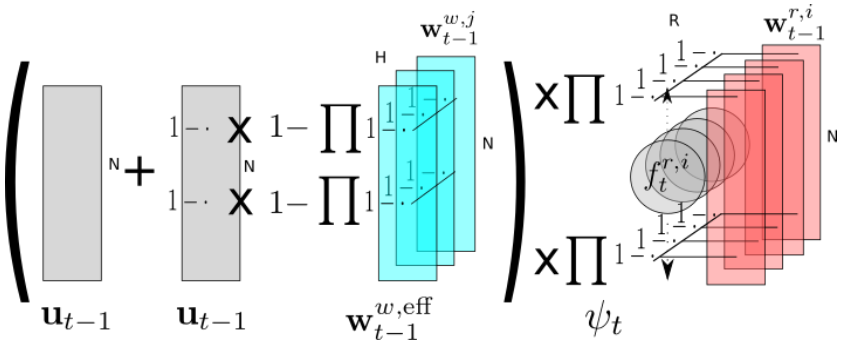


Figure 1: An illustration of the usage update. Note that the order of operations are relaxed in this illustration, i.e. there are no parentheses around $1 - \prod_j (1 - \mathbf{w}_{t-1}^{w,j})$.

Intuitively, the **free gate** $f_t^{r,i}$, tries to delete all the information that was marked for reading by the read weighting. When the free gate is high, the usage is low for all N slots in memory. Low usage is a necessary condition for a high write weight. Alternatively, a low free gate indicates that the controller does not want to mark any slots as writeable. Conversely, if all free gates are low, all information is retained, and similarly if all write weight in a particular slot is low.

The calculation of ψ_t is trivial: Perform a `reduce_product` on the R axis of $(1 - \mathbf{f}_t w_{t-1}^r)$, where $\mathbf{f}_t, w_{t-1}^r$ denote collections. Note that ψ_t is of shape $B \times N$.

Now, how do we find what slots to keep or delete based on the write heads? Let

$$\mathbf{w}_{t-1}^{w,\text{eff}} = 1 - \prod_{j=1}^{H} (1 - \mathbf{w}_{t-1}^{w,j})$$

denote the "effect" of all the write weight vectors. The intuition here is not obvious. Suppose a write weight entry $w_{t-1}^{w,j}[k] \in [0, 1]$ is small. Then write head j is not writing much information to memory slot k. This implies $(1 - w_{t-1}^{w,j}[k])$ is large. If $w_{t-1}^{w,j}[k]$ is small for all $1 \leq j \leq H$, no write head is focusing on slot k. Then, $(\prod_j 1 - w_{t-1}^{w,j})[i]$ is large. Since it is also bounded in $[0, 1]$, $\mathbf{w}_{t-1}^{w,\text{eff}}$ is small. So, if no write head focuses on a particular slot, the effective write weight of that slot is low. On the other hand, if some write head focuses on a particular slot, the effective write weight is relatively high. The vector is of shape $B \times N$. Now we apply the formula $\mathbf{u}_{t-1} + (1 - \mathbf{u}_{t-1}) \odot \mathbf{w}_{t-1}^{w,\text{eff}}$. That is, if the previous usage of slot i is low and the slot has a high effective write weight, increase the usage value of slot k from what it was. Otherwise, do not significantly change usage. This structure is defined in the DeepMind code [5].

Then finally, we want to mark a memory slot as used if retention is high *and*

previous usage was low and the effect was high. That is,

$$\mathbf{u}_t = (\mathbf{u}_{t-1} + (1 - \mathbf{u}_{t-1}) \odot \mathbf{w}_{t-1}^{w,\text{eff}}) \odot \psi_t. \tag{1}$$

Carol Hsin [3] proves that the elements of usage remain in the range $[0, 1]$ when using a single write head. The argument extends easily.

4.2.2 Allocation

However, we want to sharpen the locations we write new information into even further, and invert them. Then we write to memory slots that are relatively unused. We introduce the allocation weighting. Reassign usage to a dummy variable $\bar{\mathbf{u}}_t^1 \leftarrow \bar{\mathbf{u}}_t$ For each write head j,

- Sort usage ascendingly along the N axis. Then slots with low usage values appear first.
- Store the original indices in the new order in a vector $\phi \in \mathbb{N}^N$. That is, $\bar{u}^j_{\text{asc},t}[k] = \bar{u}^j[\phi[k]]$.
- Calculate

  $$\mathbf{a}^j_{\text{asc},t}[k] = (1 - \bar{\mathbf{u}}^j_{\text{asc},t})[k] \prod_{n=1}^{k-1} \bar{u}^j_{\text{asc},t}[n]$$

  for $k \geq 2$, such that $\mathbf{a}^j_{\text{asc},t}[1] = (1 - \bar{\mathbf{u}}^j_{asc,t})[1]$. Think of $\mathbf{a}^j_{\text{asc},t}$ as nonusage. Notice that element k is high when usage is low, i.e. when k is low. The product term plays the sharpening role. As k becomes larger, the product becomes smaller since $0 \leq \bar{u}^j_{\text{asc},t}[k] \leq 1$. That is, if usage is extremely low, allocation is extremely close to 1. A small increase in usage results in a much larger drop in allocation.

  We may represent $\prod \bar{u}^j_{\text{asc},t}[n]$ as the vector

  $$\left(1 \quad \bar{u}^j_{\text{asc},t}[1] \quad \cdots \quad \prod_{n=1}^{k-1} \bar{u}^j_{\text{asc},t}[n] \quad \cdots \quad \prod_{n=1}^{N-1} \bar{u}^j_{\text{asc},t}[n]\right),$$

  which is efficiently calculated by many deep learning libraries as an exclusive `cumprod`. Then we have the vectorization

  $$\mathbf{a}^j_{\text{asc},t} = (1 - \bar{u}^j_{\text{asc},t}) \odot \overset{\text{excl.}}{\prod} \bar{u}^j_{\text{asc},t}.$$

- Then unsort to obtain the final allocation weighting over the N memory locations:

  $$\mathbf{a}^j_t[\phi[k]] = \mathbf{a}^j_{\text{asc},t}[k].$$

- Update usage according to

  $$\bar{u}^{j+1}_t \leftarrow \bar{u}^j_t + g^{w,j}_t (1 - \bar{u}^j_t) \odot \mathbf{a}^j_t. \tag{2}$$

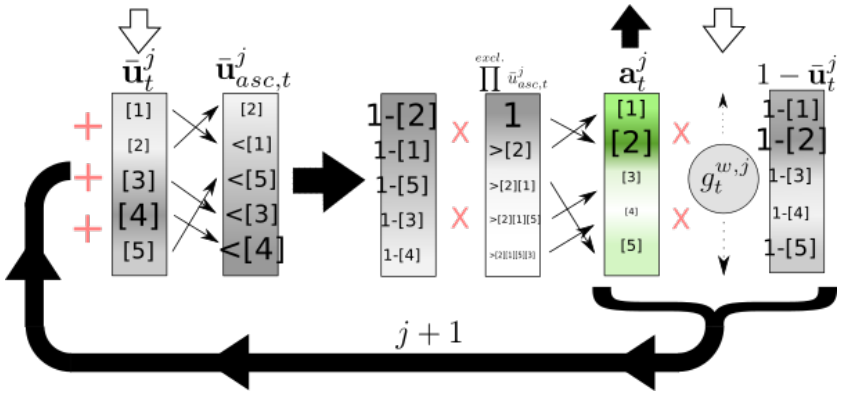


Figure 2: An illustration of the allocation algorithm. Notice the sharpening of the color gradient, reflected by the font size. Notice the white arrows denote a dependency outside the algorithm. Specifically, $\bar{\mathbf{u}}_t^1 = \mathbf{u}_t$ and $g_t^{w,j}$ is determined by the interface vector for all j.

> That is, add to the usage of a location the amount that we anticipate on writing to that location (low usage, high allocation, and high write gating means high writing). For instance, if the **write gate** $g_t^{w,j}$ is low for head j, we do not expect much change in usage. On the other hand, if allocation and the write gate are high while usage is low (the relationship between allocation and usage that we expect), we change that memory location significantly before write head $j+1$ may act. We want to reflect that when allocating for head $j+1$.

Note that this has the effect of making allocation small when usage is high. The allocation for head j is stored in column j of the allocation matrix. After iterating through each head, we have an allocation weighting tensor of shape $B \times N \times H$.

4.2.3 Allocation Alternative

Since the original DNC paper was published, a new allocation mechanism has come to light. A rather simple weighted softmax allocation suffices to sharpen and invert the usage vector. That is, assign $\bar{\mathbf{u}}_t^1 \leftarrow \mathbf{u}_t$. Then,

$$\mathbf{a}_t^j = \text{softmax}(\beta_t^{a,j}(1 - \bar{\mathbf{u}}_t^j))$$

and update the dummy usage according to equation (2). $\beta_t^{a,j} = \text{oneplus}(\hat{\beta}_t^{a,j}) \in [1, \infty)$ is an additional interface parameter, known as the **allocation strength**. The collection is of shape $B \times 1 \times H$, necessitating that ℓ_ζ be increased by H.

This method was proposed by Ben-Ari and Bekker [2] in publishing and simultaneously by Albert [4] on Github.

4.2.4 Content Lookup

Next, we look up which memory locations have similar information to that in the write **keys** $\mathbf{k}_t^{w,j}$. This task is referred to as **content lookup**, and we perform it again when calculating the read weighting. We use the cosine similarity metric between of the H key vectors and each of the N rows in memory. Specifically,

$$\text{sim}(\mathbf{u}, \mathbf{v}) = \cos(\theta) = \frac{\mathbf{u} \cdot \mathbf{v}}{|\mathbf{u}||\mathbf{v}|}.$$

Then, calculate a probability distribution over the N slots in memory, using **write strength** $\beta_t^{w,j} \in [1, \infty)$. That is,

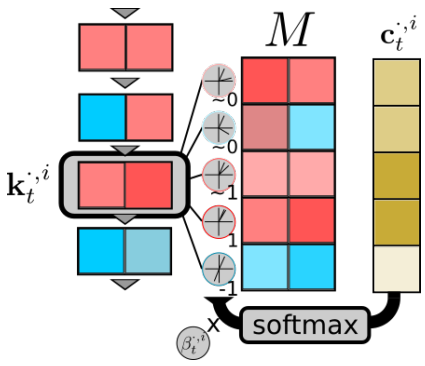
$$\mathbf{c}_t^{\cdot,h}[i] = \frac{\exp(\beta_t^{\cdot,h} \text{sim}(\mathbf{k}_t^{\cdot,h}, M_*[i,:])}{\sum_{n=1}^{N} \exp(\beta_t^{\cdot,h} \text{sim}(\mathbf{k}_t^{\cdot,h}, M_*[n,:])}$$

We use : to mean "all," as in Matlab syntax. Why $\cdot$ and $*$? Later, we perform the same operation with reading parameters, hence $\cdot$ stands for w now and r later. Before the write heads update memory, we must use M_{t-1}. However, when we perform content lookup for the read weighting, we have M_t. Hence, $*$ stands for the appropriate choice based on $\cdot$.

We vectorize the calculation by normalizing each row in memory $M_*[n,:]$ and normalizing the key $k_t^{\cdot} \in \mathbb{R}^{B \times W \times \star}$ along the W axis, giving $\hat{M}_*$ and $\hat{k}_t^{\cdot}$. Then we matrix multiply $\hat{M}_* \hat{k}_t^{\cdot}$ to produce a collection of raw similarity vectors of shape $B \times N \times \star$. Then we convert each of the H or R similarity scores into a probability distribution according to

$$c_t^{\cdot} = \text{softmax}(\beta_t^{\cdot} \hat{M}_* \hat{k}_t^{\cdot})[:,:].$$

This takes advantage of broadcasted scalar multiplication over the $\cdots$ axis. We collect this into a tensor of shape $B \times N \times \star$.



4.2.5 Write Weighting

Now we calculate the write weighting $\mathbf{w}_t^{w,j}$. Simply,

$$\mathbf{w}_t^{w,j} = g_t^{w,j}(g_t^{a,j}\mathbf{a}_t^j + (1 - g_t^{a,j})\mathbf{c}_t^{w,j}).$$

Then, the **allocation gate** $g_t^{a,j}$ controls whether to write to a location with low usage, i.e. allocated (high allocation gate) or a location with similar content (low allocation gate). Notice that $\mathbf{w}_t^{w,j} \in [0,1]^N$ and that $\sum_i^N \mathbf{w}_t^{w,j} \leq 1$. This is not immediately obvious. It follows from weighting two vectors with these properties by a scalar $g_t^{a,j}$ in $[0,1]$ and its compliment $1 - g_t^{a,j}$. Then, the write gate

controls whether we write at all. Thinking of the write weights as a probability distribution over the memory locations, the remainder of $1 - \sum_{i=1}^{N} \mathbf{w}_t^{w,j}$ is the probability that a write head will take no action at all. This is a head's **null operation**.

4.2.6 Erase and Write

Finally, each write head j erases its **erase vector** $\mathbf{e}_t^j$ from each memory slot i with strength $1 - \mathbf{w}_t^{w,j}[i]$ and writes its **write vector** $\mathbf{v}_t^j$ to each memory slot i with strength $\mathbf{w}_t^{w,j}[i]$. Vectorizing, we have

$$M_t = M_{t-1} \odot (J_{(B \times N \times W)} - w_t^w (e_t)^T) + w_t^w (v_t)^T$$

where $w_t^w \in [0,1]^{B \times N \times H}$ denotes the collection of all H write weights, $e_t \in [0,1]^{B \times W \times H}$ denotes the collection of all erase vectors, and $v_t \in \mathbb{R}^{B \times W \times H}$ is the collection of all write vectors. J denotes the matrix of ones.

Note that this is the only use of the erase and write vectors. We may reshape the vector emitted from the interface into size $B \times H \times W$ to sidestep transposition. We keep the shapes as is because they are consistent with other interface variables.

4.3 Read Heads

Now, we consider the R read heads. Each head can read the information off of memory in forward order, backwards order, or in order of most similar to its content key $\mathbf{k}_t^{r,i}$. Independently for each head, the controller determines how to "mix," or interpolate, the orderings into a distribution over the memory locations. Then, each head uses its respective read-order (in this context, best interpreted as strengths per memory location) to read information from memory into its respective vector, $\mathbf{r}_t^i$.

4.3.1 Temporal Linkage

First, we update the temporal linkage tensor. The linkage tensor is of shape $B \times H \times N \times N$. For this operation, we will need the previous precedence vector of shape $B \times N \times H$ and the current write weights of shape $B \times N \times H$. Expanding the equation provided in the DNC paper [1] to represent the DeepMind code [5],

$$L_t^j[h,k] = (1 - (\mathbf{w}_t^{w,j}[h] + \mathbf{w}_t^{w,j}[k])) \odot L_{t-1}^j[h,k] + \mathbf{w}_t^{w,j}[h] \mathbf{p}_{t-1}^j[k]$$

In the first term, reduce the linkage value if $\mathbf{w}_t^{w,j}[h] + \mathbf{w}_t^{w,j}[k]$ is high. That is, if both slots h and k were highly written during this time-step, reset any old links from h to k and from k to h. The current information was written at the same time-step, so there is no temporal linkage.

In the second term, add the matrix product of the write weighting and the previous precedence. This represents new links. To understand how, we must

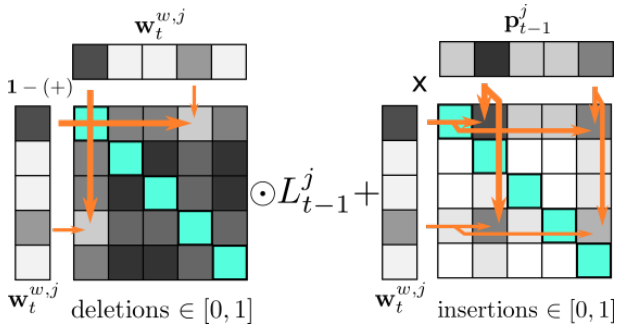


Figure 3: An illustration of the linkage update. Lighter values represent smaller values. On the left, we delete linkage both ways between slots on which the write heads focus at the current time-step. That is, delete $[1, 4]$ and $[4, 1]$. On the right, we see how links form between focused slots and previously focused slots. The diagonal is blued since it has no meaning in the linkage.

touch on the intuition of the precedence. It suffices to know that the precedence is mostly a copy of the previous write weights. So, if head j significantly wrote to slot h during the last time-step, then $p_{t-1}^j[h]$ is high. If head j wrote to slot k during the current time-step, then $\mathbf{w}_t^{w,j}[k]$ is high. So, their product is high. This product is the temporal link.

The vectorized implementation relies on broadcasting to handle all heads at once. For simplicity, we are going to change the shape of the write weights and precedence to $[0, 1]^{B\times H\times N}$. Then expand w_t^w into two tensors: $(w_t^w)^h$ of shape $B \times H \times 1 \times N$ and $(w_t^w)^k$ of shape $B \times H \times N \times 1$. Then, by the standard rules handling of broadcasted addition, we arrive at

$$(1 - (w_t^w)^h - (w_t^w)^k)[h, :, :] = \begin{pmatrix} 1 - w_t^w[1] - w_t^w[1] & 1 - w_t^w[1] - w_t^w[2] & \cdots & 1 - w_t^w[1] - w_t^w[N] \\ 1 - w_t^w[2] - w_t^w[1] & 1 - w_t^w[2] - w_t^w[2] & \cdots & 1 - w_t^w[2] - w_t^w[N] \\ \vdots & \vdots & \ddots & \vdots \\ 1 - w_t^w[N] - w_t^w[1] & 1 - w_t^w[N] - w_t^w[2] & \cdots & 1 - w_t^w[N] - w_t^w[N] \end{pmatrix}$$

and we have a similar application for the links formed by the $\mathbf{p}_{t-1}^j$ and $\mathbf{w}_t^{w,j}$. We expand such that $(p_{t-1})^k \in [0, 1]^{B\times H\times N\times 1}$. Then, using broadcasted element-wise multiplication,

$$((w_t^w)^h(p_{t-1})^k)[h, :, :] = \begin{pmatrix} w_t^w[1]p_{t-1}[i] & w_t^w[1]p_{t-1}[2] & \cdots & w_t^w[1]p_{t-1}[N] \\ w_t^w[2]p_{t-1}[i] & w_t^w[2]p_{t-1}[2] & \cdots & w_t^w[2]p_{t-1}[N] \\ \vdots & \vdots & \ddots & \vdots \\ w_t^w[N]p_{t-1}[i] & w_t^w[N]p_{t-1}[2] & \cdots & w_t^w[N]p_{t-1}[N] \end{pmatrix}.$$

That is,

$$L_t[:, i, j] = (1 - (w_t^w)^h - (w_t^w)^k) \odot L_{t-1} + (w_t^w)^h (p_{t-1})^k.$$

One more aspect: The temporal link between an slot in memory and itself is unclear. Is slot k written before or after slot k? It's a nonsense question. We define $L_t^j[k, k] = 0$ for all $k \leq N, j \leq H$. There are two natural ways to handle this. We have `matrix_set_diag` in TensorFlow and more generally we have $(1 - I_N) \odot L'_t$.

4.3.2 Precedence

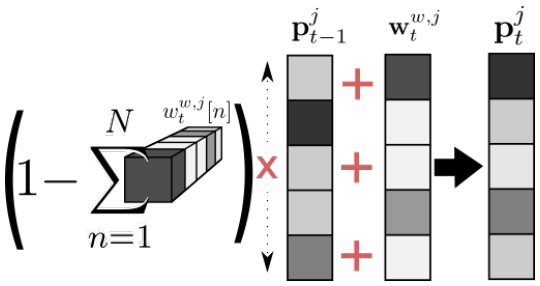


Figure 4: An illustration of the precedence update. Notice how the precedence at time-step t predominantly takes values from the current write weighting. Hints of the previous precedence may propagate since the formula fills the null operation distribution with the previous precedence.

Next, we update the precedence vector. The implementation is trivial:

$$\mathbf{p}_t^j = (1 - \sum_{n=1}^{N} w_t^{w,j}[n])\mathbf{p}_{t-1}^j + \mathbf{w}_t^{w,j}.$$

We have mentioned the interpretation of the write vector while exploring the linkage matrix. Let us expand on this now. Precedence is high when a slot is being written to, but we also want to leave some residual, old write weightings. Since there is no link for a null operation, we add to the current write weighting the old precedence scaled by the probability of null operations. This handles the case where the head entirely takes null operations as well. For instance, during recall a DNC does not need to write to memory. For the most part, precedence is a slightly recursive holder for the write weighting into the next time-step. We collect the precedence vectors into shape $B \times N \times H$.

4.3.3 Read Weighting

At this point, we proceed to calculate the reading weights. We must take care to explain how we transfer from the H axis into the R axis. The DNC paper does

not provide this information, but the DeepMind code does. Write weighting for a read head is composed of several parts: The **forward weighting** based on all write head linkage, the similar **backwards weighting**, and a content weighting for each read head. Pinning it together is a probability distribution from the interface vector.

We perform content lookup again using $\mathbf{k}_t^{r,i}$ and the freshly written memory M_t to give $\mathbf{c}_t^{r,i} \in [0,1]^{B \times N \times 1_i}$, and collected into shape $B \times N \times R$. This is described in Section 4.2.4

First, we calculate the forward read order weighting. Recall L_t is of shape $B \times H \times N \times N$. Let L_t^j denote the jth matrix along the H axis of shape. And, we have $\mathbf{w}_t^{r,i}$ of shape $B \times N \times 1$ for all $i \leq R$. We multiply L_t^j by $\mathbf{w}_t^{r,i}$ for all $i \leq R$ and collect the result into a $B \times N \times R$ matrix. We repeat this for all $j \leq H$ and collect into a tensor of shape $B \times H \times N \times R$. That is,

$$\mathbf{f}_t^{i,j} = L_t^j \mathbf{w}_{t-1}^{r,i} \in [0,1]^{B \times 1_j \times N \times 1_i} \ \forall 1 \leq j \leq H.$$

A brief example suffices to explain the behavior. Assume slot n was highly written at $t-1$ and $w_{t-1}^{r,i}$ has a high value for slot n. And, suppose slot m was highly written at t. Then $L_t^j[m,n]$ is high. When we multiply L_t^j by $w_{t-1}^{r,i}$, the product has a large value in entry m. This is because row m of L_t^j had a large value in column n and $w_{t-1}^{r,i}$ also had a large value in column n. Then $\mathbf{f}_t^{i,j}$ focuses on slot m. We perform a nearly identical process to obtain the backward weighting. Let

$$\mathbf{b}_t^{i,j} = (L_t^j)^T \mathbf{w}_{t-1}^{r,i} \in [0,1]^{B \times 1_j \times N \times 1_i}.$$

Note that transposing the link matrix effectively reverses the order of the slot links. So, if slot n was just read, increase the likelihood of reading the information written before n by head j.

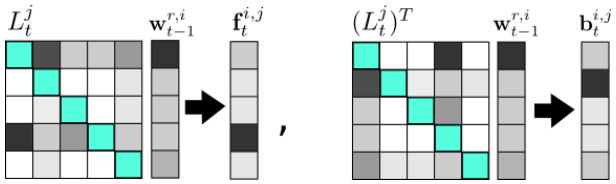


Figure 5: An illustration of how the forward and backwards weighting use the linkage matrix. Here it is supposed that memory slot 2 was written first, then slot 1 at $t-1$, and most recently 4. The previous read weights focused on 1.

We may efficiently calculate each collection f_t and b_t. First, tile w_{t-1}^r to include an H axis. Then use broadcasted element-wise multiplication across the H dimension of the linkage tensor and regular matrix multiplication over the tiled collection w_{t-1}^r. Each has shape $B \times H \times N \times R$.

Now we may calculate the read weighting. Recall that we have the **read modes**, $\pi_t^i \in S_{2H+1}$, collected in shape $B \times 2H+1 \times R$. As a quick aside, it

should be apparent that the vector

$$\left(\sum_{j=1}^{H} \pi_t^i[j] \quad \sum_{j=H+1}^{2H} \pi_t^i[j] \quad \pi_t^i[2H+1]\right)$$

is still a valid probability distribution since the three elements still sum to 1 and each element is still positive. With this in mind,

$$\mathbf{w}_t^{r,i} = \sum_{j=1}^{H} \pi_t^i[j]\mathbf{b}_t^{i,j} + \sum_{j=1}^{H} \pi_t^i[H+j]\mathbf{f}_t^{i,j} + \pi_t^i[2H+1]\mathbf{c}_t^{r,i}.$$

has a nice interpretation as a vector interpolating a distribution of actions to take. If we were to expand the sums, we would see that, e.g. for slot n the odds of head i reading information written before writing to slot n by write head j is determined by the interface vector's read mode $\pi_t^i[j]$. Read head i simply adds up all the actions it is instructed to take and reads the memory according to that weighting. Each of the R vectors are of shape $B \times N \times 1$, and collected into shape $B \times N \times R$.

Using broadcasted element-wise multiplication across the H dimension of the collections b_t and f_t, we can quickly calculate the two "big epsilon" summands. Then, we are left to take a `reduce_sum` down the H axis. The implementation of the content weighting is trivial.

4.3.4 Reading

Finally, each read head reads data off of the memory according to its weighting. That is,

$$\mathbf{r}_t^i = (w_t^{r,i})^T M_t,$$

giving R read vectors of shape $B \times 1 \times W$.

5 Output

Define a new learned weighting $W^o \in \mathbb{R}^{WR \times \ell_y}$. Then let

$$\mathbf{y}_t = \hat{\mathbf{y}}_t + [\mathbf{r}_t^1; \mathbf{r}_t^2; \ldots; \mathbf{r}_t^R]W^o.$$

Note that the action taken with the read vectors is not dynamic over t. That is, W^o is updated by gradient descent after the DNC iterates through the entire sequence. It is a general transformation that suffices for all time-steps.

Bibliography

[1] Alex Graves and et. al. "Hybrid computing using a neural network with dynamic external memory". In: *Nature* 322 (2016), pp. 471–476. DOI: 10.1038/nature20101.

[2] Itamar Ben-Ari and Alan Joseph Bekker. *Differentiable Memory Allocation Mechanism for Neural Computing*. 2017.

[3] Carol Hsin. *Implementation and Optimization of Differentiable Neural Computers*. 2017.

[4] Joshua Albert. *You can avoid the top_k and allow usage to be differentiable.* URL: `https://github.com/deepmind/dnc/issues/21`.

[5] DeepMind. *deepmind/dnc*. URL: `https://github.com/deepmind/dnc`.