Recent Advancements in Neural Computation:
Differentiable Neural Computers and End-To-End Memory Networks

Konstantinos Kogkalidis [6230067]

June 4, 2018

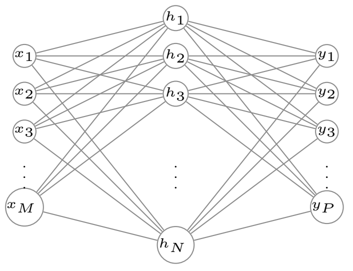
1 Introduction

In the last decade, neural networks have considerably revolutionized computation and artificial intelligence. Neural networks have been setting new benchmarks in tasks from a broad range of fields, including computer vision, natural language processing and pattern recognition; many of these tasks previously unsolved, and some of these benchmarks surpassing human-level performance. Until recently, this astounding potential has for the most part been limited to tasks involving sensory processing and representation learning. Memory-centric tasks, i.e. tasks requiring data storage and retrieval, have however proven difficult for neural networks to model. This paper serves as a literature review of two recent approaches towards bypassing this limitation, namely Differentiable Neural Computers (DNCs) and End-to-End Memory Networks. The rest of this section will briefly introduce some important background concepts, before the models' details are examined in sections 2 and 3, respectively. Section 4 provides a short discussion and a comparative analysis on the two approaches, and section 5 presents some concluding remarks.

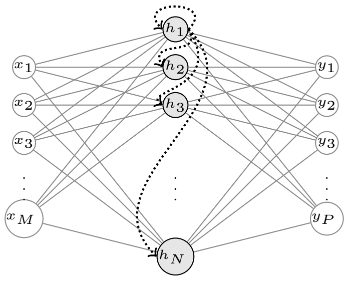
A particularly interesting subclass of neural networks are *Recurrent Neural Networks* (RNNs). Unlike traditional networks, which accept a fixed-size input vector and produce a fixed-size output vector over a finite number of computational steps, RNNs instead operate on sequences of vectors. This additional axis is captured by allowing the RNN to accept one vector at a time and construct an intermediate representation of it, which then combines with the next input vector to affect the computation of the next such representation (Fig.1). This elegant addition accounts for a drastic improvement in the representational capacity of the network. Whereas a simple net acts as a fixed function, a RNN acts as a program operating on sequences of inputs. Under this viewpoint, the RNN's hidden representation can be perceived as the program's internal variables, updating as new inputs are received and affecting the computation as it progresses.

Although RNNs have been introduced more than three decades ago, their potential was only recognized recently, as the availability of computational power surged. Their ability to model sequential patterns and their interesting properties have garnered a lot of attention, yet they still suffer from two major drawbacks. The first relates to the statefulness achieved by their intermediate representation; even though it is distributed, the hidden vector's size is finite and fixed, therefore capable of only limited, lossy storage. The second is a more intrinsic one and is not only tied to RNNs but neural networks in general; there is no separation between memory and processing, as each neuron is simultaneously a minimal storage unit and a computational one. Although appealing from a biological perspective, this goes against the von Neumann model and our programming paradigm.

A series of advancements has seeked to enhance RNNs with stronger memory management. Starting with the introduction of the *Long Short-Term Memory*, which permits finite storage over arbitrary long computations, research in RNNs has bloomed towards many different directions.



(a) Simple Neural Net



(b) Recurrent Neural Network

Figure 1: *Comparison between a simple neural network and a RNN. (a) The simple network applies two transformations $\boldsymbol{x} \mapsto \boldsymbol{h}$ and $\boldsymbol{h} \mapsto \boldsymbol{y}$, the composition of which can be seen as a function $\boldsymbol{x} \mapsto \boldsymbol{y}$. (b) The recurrent network allows through-time connections, depicted with the dotted lines, such that $\boldsymbol{x}_t \times \boldsymbol{h}_{t-1} \mapsto \boldsymbol{h}_t$. The output then becomes a function of the full sequence of inputs received:* $\underbrace{\boldsymbol{x}_0 \times \boldsymbol{x}_1 \times \cdots \times \boldsymbol{x}_t}_{\text{Sequence Length}} \mapsto \boldsymbol{y}_t$.

One particularly exciting recent advancement is the so-called *attentional interface*, which allows a network to selectively shift its focus on particular indices over a set of dimensions of an arbitrary tensor, thus enabling content-based adressing. The implications of this seemingly simple trick are tremendous; when employing fully differentiable attention (soft-attention), a network can learn how to pick out and combine relevant elements out of a (potentially very large) tensor, instead of lossily encoding it. This translates to arbitrary information retrieval; in other words, the same representations can now have different effects on the computation depending on the present context, similar to how a computer program would refer only to the variables dictated by its currently run instruction. Since its inception, attention has been a key-concept in many influential publication; the creative liberty it is offering, in terms of new potential operations, has also paved the way for architectures which decouple storage from computation. Such architectures, including but not limited to the two presented in this paper, utilize large external matrices and employ attention to perform read and write operations on them.

2 Notation and Preliminaries

Given the technical nature of the works presented, it is important to establish a notational common ground. The $(N-1)$-simplex, i.e. the set of all variations of N non-negative scalars that sum up to 1, will be indicated by $\mathcal{S}^N$. A vector will be denoted by a boldface lower letter, e.g. $\boldsymbol{v}$, whereas a matrix will be indicated by a boldface capital letter, e.g. $\boldsymbol{M}$. Vector concatenation, i.e. merging together N vectors in the order they are given to a create a new, long vector is denoted by a bracketing of vectors separated by the semicolon, e.g. $[\boldsymbol{x}; \boldsymbol{y}; \boldsymbol{z}]$ reads as the concatenation of $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{z}$. When using the comma rather than the semicolon, the notation instead presents a sequence of vectors, e.g. $[\boldsymbol{a}_1, \ldots, \boldsymbol{a}_N]$ denotes a sequence of vectors $\boldsymbol{a}_1$ to $\boldsymbol{a}_N$. A subscript on a vector or matrix is used to index a single temporal instance from a sequence of such vectors or matrices, e.g. $\boldsymbol{a}_i$ refers to the instance at timestep i of the time-varying vector sequence $[\boldsymbol{a}_1, \ldots]$. It should not be confused with intra-vector or matrix indexing, i.e. singling out particular elements out of a vector or matrix, which is denoted by $\boldsymbol{a}[i]$ and reads as the i-th element of a vector $\boldsymbol{a}$. Finally, a reminder that a matrix $\boldsymbol{M} \in \mathbb{R}^{K \times L}$ can be seen as a linear map $M : \mathbb{R}^L \to \mathbb{R}^K$; linear application of $\boldsymbol{v} \in \mathbb{R}^V$ on $\boldsymbol{M} \in \mathbb{R}^{M \times V}$ (or matrix-vector multiplication) is thus simply denoted $\boldsymbol{Mv}$.

3 Differentiable Neural Computers

3.1 Description

Inspired by the von Neumann architecture, DNCs model computation as the interfacing between two units; a RNN controller and an external memory matrix, functioning analogously to the CPU and the RAM, respectively. The memory matrix can be adressed using soft attention mechanisms, allowing selective reading and writing. As both the controller and the interfacing are differentiable, the full architecture is end-to-end differentiable, essentially allowing it to be trained purely with data and minimal human supervision. As the system is already equiped with the necessary tools to simulate traditional computation, i.e. adressing mechanisms, information storage and retrieval, logical flows and arithmetic processing, a DNC in training is learning the instructions of a program rather than the parameters of a function, and a trained DNC should be able to mimic the behavior of a traditionally hand-written program.

3.2 Architecture

Overview A differentiable neural computer consists of two main components; a recurrent neural network **controller network** and an external **memory matrix** $M \in \mathbb{R}^{N \times W}$. The matrix contains N memory adresses, each of size W, which can be read by the controller through R *read heads*, or written to by one *write head.* At timestep t, the system as a whole receives input in the form of a vector, $\boldsymbol{x}_t \in \mathbb{R}^X$, and produces output in the form of a vector, $\boldsymbol{y}_t \in \mathbb{R}^Y$. The input may be samples from any dataset environment, e.g. some encoding of sensory signals. The output can either represent an approximation for a target vector $\boldsymbol{z}_t \in \mathbb{R}^Y$ (in the supervised learning setting) or encode some action (in the reinforcement learning setting).

Controller The aforementioned input is passed to the controller network, alongside the contents of each of the read heads as drawn from the memory at the previous time instance: $\boldsymbol{r}_{t-1}^i$, $i \in 1, \ldots, R$. The concatenation of the above yields a vector $\boldsymbol{\chi}_t \in \mathbb{R}^{X+RW} : \boldsymbol{\chi}_t = [\boldsymbol{x}_t; \boldsymbol{r}_{t-1}^1; \ldots \boldsymbol{r}_{t-1}^R]$. A full sequence of such inputs, $[\boldsymbol{\chi}_1, \ldots, \boldsymbol{\chi}_T]$ is passed through a set of dynamic state equations $\mathcal{N}$, as implemented by the controller, that yield back two intermediate outputs, $\boldsymbol{v}_T$ and $\boldsymbol{\xi}_T$, such that $\mathcal{N}([\boldsymbol{\chi}_1, \ldots, \boldsymbol{\chi}_T]) = (\boldsymbol{v}_T,\ \boldsymbol{\xi}_T)$. The exact implementation of $\mathcal{N}$ is not essential to the architecture as a whole; $\mathcal{N}$ may be any neural network. Deep long short-term memory networks are the state of the art in sequential modeling, and therefore make for a reasonable choice, since the controller needs to be expressive enough to handle potentially complex data streams.

Out of the vectors $(\boldsymbol{v}_t,\ \boldsymbol{\xi}_t)$ returned by the controller, the first corresponds to the intermediate user-level output. Before conveying this to the user, the system is allowed one last peak at the memory. This is necessitated by the fact that the read head contents at the final timestep of the computation cannot be used in $\mathcal{N}$, as this would introduce circular dependencies in the architecture's computational graph. The memory conditioning process is therefore done outside the controller, as captured by a simple trainable linear map

$$\boldsymbol{y}_t = \boldsymbol{v}_t + \boldsymbol{W}_R[\boldsymbol{r}_t^1; \ldots; \boldsymbol{r}_t^R]$$

The result of the operation is the current instance of the user-level output stream, $\boldsymbol{y}_t$, which represents what the DNC is yielding back to the world while computing.

The second controller output, $\boldsymbol{\xi}_t$, corresponds to the *interface vector*, which defines the controller's interaction with the memory. The interface vector can be subvidided into multiple smaller components, each one of which is responsible for a specific subtask related to memory adressing. Different activation functions are applied to these components, so as to enforce their distributions to fall within their task-specific domain. Post activation application, we have the following vectors and scalars:

- Read keys: $\{\boldsymbol{k}_t^{r,i} \in \mathbb{R}^W,\ \forall i \in \{1,\dots,R\}\}$
- Read strengths: $\{\beta_t^{r,i} \in [1,\infty)\ \forall i \in \{1,\dots,R\}\}$
- Write key: $\boldsymbol{k}_t^w \in \mathbb{R}^W$
- Write strength: $\beta_t^w \in [1,\infty)$
- Erase vector: $\boldsymbol{e}_t \in [0,1]^W$
- Write vector: $\boldsymbol{v}_t \in \mathbb{R}^W$
- R free gates: $\{f_t^i \in [0,1]\ \forall i \in \{1,\dots,R\}\}$
- Allocation gate: $g_t^a \in [0,1]$
- Write gate: $g_t^w \in [0,1]$
- Read modes: $\{\boldsymbol{\pi}_t^i \in \mathcal{S}^3\ \forall i \in \{1,\dots,R\}\}$

The remainder of this section is dedicated to examining the role of each of these components.

Content-based Addressing DNCs perform content-based addressing via *autoassociative memory.* Autoassociative memory is a strong mechanism of content-lookup, that allows a system to navigate rich data structures and selectively retrieve relevant memory objects when only shown parts thereof. The implementation involves a *matching function* $\mathcal{D} : \mathbb{R}^W \times \mathbb{R}^W \to \mathbb{R}$, which is applied on a read/write key $\boldsymbol{k}$, as given by the controller, and the contents of a memory address, $\boldsymbol{M}[i,:]$, and returns a scalar indicating how similar these two are. Any metric of distance may be used as $\mathcal{D}$, but the most obvious choice is cosine similarity:

$$\mathcal{D}(\boldsymbol{x},\boldsymbol{y}) = cos(\boldsymbol{x},\boldsymbol{y}) = \frac{\boldsymbol{x}\cdot\boldsymbol{y}}{||\boldsymbol{x}||\cdot||\boldsymbol{y}||} \qquad \text{(Cosine Similarity)}$$

The controller may assign varying relative importance on different keys; this is accomplished by exponentially scaling the matches obtained by $\mathcal{D}$, using the strength vector associated with each key. These weighted matches are finally normalized by their sum, constraining the results within $\mathcal{S}^N$. The above are captured by a *weighting function* $\mathcal{C} : \mathbb{R}^{N\times W} \times \mathbb{R}^W \times \mathbb{R} \to \mathbb{R}^N$, such that:

$$\mathcal{C}(\boldsymbol{M},\boldsymbol{k},\beta)[i] = \frac{\mathcal{D}(\boldsymbol{M}[i,:],\boldsymbol{k})^\beta}{\sum_{j=1}^N \mathcal{D}(\boldsymbol{M}[j,:],\boldsymbol{k})^\beta} \qquad \text{(Weighting Function)}$$

This is a fully differentiable form of attention, which allows the controller to construct minimal representations of objects that it's looking for in the memory. The key benefit of such a process is its ability to perform partial matching or pattern completion; for instance, the controller may simply construct the beginning of a sequence (e.g. the identifier of a variable), and, upon finding its match, it can then retrieve the full sequence (e.g. the value of that variable) back from the memory.

The weighting function is used to obtain content-based weightings for both the read heads, denoted by $[\boldsymbol{w}_t^{r,i},\dots,\boldsymbol{w}_t^{r,R}]$, and the the write head, $\boldsymbol{w}_t^w$. Given such weightings, a read head will contain a weighted average of the memory matrix:

$$\boldsymbol{r}_t^i = \boldsymbol{M}_t^\top \boldsymbol{w}_t^{r,i} \qquad \text{(Memory Reading)}$$

Writing to the memory is also dictated by a similar operation, which must now be defined via a recurrency relation to take into account past memory contents as well as the erase and write vectors emitted by the controller:

$$\boldsymbol{M}_t = \underbrace{\boldsymbol{M}_{t-1} \circ (\boldsymbol{1} - \boldsymbol{w}_t^w \boldsymbol{e}_t^\top)}_{\text{Partially erased past memory}} + \underbrace{\boldsymbol{w}_t^w \boldsymbol{v}_t^\top}_{\text{New memory}} \qquad \text{(Memory Writing)}$$

Dynamic Allocation Memory management is an integral part of any computer program. The DNC needs to be able to allocate and deallocate memory dynamically, according to the current execution's needs. This is achieved through a differentiable variation of the *free-list scheme.* In its original form, the free-list scheme allows a system to keep track of unused memory blocks by connencting them with one another as a linked-list. Its adaptation for the purposes of the DNC is slightly more involved, as described next. Before writing to the memory, the controller emits a free gate, f_t^i, for each read head i, which controls to what extend a recently read memory address may be deallocated. From the latter , the *retention vector* can be obtained as:

$$\boldsymbol{\psi}_t = \prod_{i=1}^{R} (\mathbf{1} - f_t^i \boldsymbol{w}_t^{r,i}) \qquad \text{(Retention Vector)}$$

The retention vector is a measure of how much each memory address needs to be retained. Its complement, the *usage-tracking vector* $\boldsymbol{u}_t$, annotates each memory address with a scalar so to as to signal how strongly it is being used. It is recursively defined as follows:

$$\boldsymbol{u}_t = \underbrace{(\boldsymbol{u}_{t-1} + \boldsymbol{w}_t^w - \boldsymbol{u}_{t-1} \circ \boldsymbol{w}_t^w)}_{\text{Previous usage OR New write}} \circ \boldsymbol{\psi}_t \qquad \text{(Usage-tracking Vector)}$$

Contrary to first impressions, the above equation is quite intuitive. The parenthesized term sums the previous usage tracking vector with the current write weighting minus their dot product, acting as a real-valued variation of the logical OR function. A single element of this term, $\boldsymbol{u}_t[i]$ would denote the pre-retention usage tracking of memory address i; it is close to 1 if it was either already in use, or just written to. Multiplied by its corresponding element of the retention vector, it can then be (partially) deallocated.

Finally, the free-list $\boldsymbol{\phi}_t$ is defined as the sorted list of indices of the usage-tracking vector, i.e. $\boldsymbol{\phi}_t[i]$ is the index of the i-th least used address in the memory. The *allocation vector* $\boldsymbol{a}_t$ is then used to annotate the availability of each memory address:

$$\boldsymbol{a}_t[\boldsymbol{\phi}_t[j]] = (1 - \boldsymbol{u}_t[\boldsymbol{\phi}_t[j]]) \prod_{i=1}^{j-1} \boldsymbol{u}_t[\boldsymbol{\phi}_t[i]] \ \in [0,1] \qquad \text{(Allocation Vector)}$$

To showcase the functionality of this last equation, we will consider a simple example. Let $\boldsymbol{u}_t = [0.4, 0.8, 0.1]$ be the usage-tracking vector of a 3-address memory at timestep t. Then the free-list would be $\boldsymbol{\phi}_t = [3, 1, 2]$. Iterating over Allocation Vector we then get:

$$\begin{aligned}
\boldsymbol{a}_t[\boldsymbol{\phi}_t[1]] &= \boldsymbol{a}_t[3] = (1 - \boldsymbol{u}_t[3]) \cdot 1 = (1 - 0.1) = 0.9 \\
\boldsymbol{a}_t[\boldsymbol{\phi}_t[2]] &= \boldsymbol{a}_t[1] = (1 - \boldsymbol{u}_t[1]) \cdot \boldsymbol{u}_t[3] = (1 - 0.4) \cdot 0.1 = 0.06 \\
\boldsymbol{a}_t[\boldsymbol{\phi}_t[3]] &= \boldsymbol{a}_t[2] = (1 - \boldsymbol{u}_t[2]) \cdot \boldsymbol{u}_t[3] \cdot \boldsymbol{u}_t[1] = (1 - 0.8) \cdot 0.1 \cdot 0.4 = 0.008
\end{aligned}$$

Therefore $\boldsymbol{a}_t = [0.06, 0.08, 0.9]$.

Having the allocation vector, the controller can now shift its attention towards unused memory addresses using the allocation gate g_t^a. The shifted weightings are then gated by the write gate g_t^w, which acts as a memory-locking mechanism :

$$\underbrace{\boldsymbol{w}_t^w}_{\text{Updated weights}} = \underbrace{g_t^w}_{\text{Write-lock}} \ [\ \underbrace{g_t^a \boldsymbol{a}_t}_{\text{Shifted weights}} + \underbrace{(1 - g_t^w)\ \boldsymbol{w}_t^w}_{\text{Writeable original weights}}\ ] \qquad \text{(Allocation Shift)}$$

Temporal Linking One last mechanism of memory addressing employed by DNCs is temporal linking. Temporal linking is necessary in order to perform sequential retrieval, when sequences are stored in disparate, non-contiguous blocks. The key-element of the process is a temporal transition matrix $\boldsymbol{L} \in [0,1]^{N\times N}$, which keeps track of the temporal order by which memory addresses have been written to. More conretely, $\boldsymbol{L}[i,j]$ is close to 1 if j was strongly written to after i, or close to 0 otherwise. $\boldsymbol{L}$ is again defined via a recurrency relation; a slightly simplified version of the one in the original paper is presented below, for the sake of brevity and clarity.

$$\boldsymbol{L}[i,j] = \underbrace{(1 - \boldsymbol{w}_t^w[i] - \boldsymbol{w}_t^w[j])\boldsymbol{L}_{t-1}[i,j]}_{\text{Part of last transition}} + \underbrace{\boldsymbol{w}_{t-1}^w[i]\boldsymbol{w}_t^w[j]}_{\text{New transition}} \tag{Temporal Update}$$

The above equation is also quite intuitive. The first term of the addition allows the transition value between i and j to propagate through time. The extend of propagation of the old transition, $\boldsymbol{L}_{t-1}[i,j]$, is complentary to the amount of writing performed in these addresses; even if we had a transition in the past, it no longer holds if either i or j were just written to. On the other hand, if i was written to in the previous timestep and j was written to in the current timestep, then a new transition is instantiated. This is captured by the second term of the addition.

Mode Interpolation The temporal transition matrix can be used as one last means of attention; simply multiplying the matrix by a weighting vector allows the controller to shift its attention over the memory, now through time. $\boldsymbol{L}_t\boldsymbol{w}$ shifts the weightings $\boldsymbol{w}$ forwards in time, whereas $\boldsymbol{L}_t^\top\boldsymbol{w}$ shifts them backwards. The degree to which the controller performs forward, backward, or no temporal iteration is dictated by the read modes $\boldsymbol{\pi}_t^i$ emitted by the controller for each read head i:

$$\underbrace{\boldsymbol{w}_t^{r,i}}_{\text{Updated weights}} = \underbrace{\boldsymbol{\pi}_t^i[1]\boldsymbol{L}_t\boldsymbol{w}_t^{r,i}}_{\text{Forward shift}} + \underbrace{\boldsymbol{\pi}_t^i[2]\boldsymbol{w}_t^{r,i}}_{\text{No shift}} + \underbrace{\boldsymbol{\pi}_t^i[3]\boldsymbol{L}_t^\top\boldsymbol{w}_t^{r,i}}_{\text{Backward shift}} \tag{Temporal Shift}$$

4 End-to-End Memory Networks

4.1 Description

An end-to-end memory network is a neural architecture that also connects a RNN with a large memory array. The external nature of this array is disanalogous to traditional architectures, such as the LSTM, where the memory is encapsulated within the neural layers. Under this perspective, it can be seen as true long-term memory. The memory can be read from (potentially multiple times) during computation, using a straightforward attentional layer; the architecture therefore benefits from context-sensitive addressing capabilities. The overall model is simple, yet efficient, and includes no discontinuities in its operations, thus being trainable with gradient-based methods.

4.2 Architecture

Overview An end-to-end memory network receives two inputs; a sequence of vectors $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ and a query $\boldsymbol{q}$. Both $\boldsymbol{x}_i$ and $\boldsymbol{q}$ are defined as symbols originating from a vocabulary $\mathcal{V}$. The network processes these inputs to return an answer $\boldsymbol{a}$. This output is produced by first constructing continuous representations for both the input sequence and the query, and then iterating over these representations in many *hops*, i.e. computational steps. The authors treat two different cases of this process; the first and simplest one employing a single layer, and the second, more general one, employing multiple layers. As this approach is favorable for understanding the architecture, we are also going to follow along the same route, modulo a few minor alterations and clarifications.

Single-Layer Case First, note that a symbol x from a strictly total-ordered vocabulary $\mathcal{V}$ can be represented as a vector $\boldsymbol{x} \in \mathbb{R}^V$, where $V = ||\mathcal{V}||$ the size of the vocabulary. This is often referred to as *one-hot encoding*. Given a character x that coincides with the i-th element of $\mathcal{V}$, one-hot encoding is achieved by mapping x to a vector $\boldsymbol{x}$ whose elements are all 0, except the i-th element which is 1. This is the assumed form of input representation for the sequence $\{\boldsymbol{x}_i\}$ and the query $\boldsymbol{q}$.

The above are transformed into two new, continuous spaces of dimensionality d; from each input $\boldsymbol{x}_i$ we get a memory object $\boldsymbol{m}_i$, whereas the query $\boldsymbol{q}$ becomes an internal representation $\boldsymbol{u}$. The transformations may be the result of any trainable operation, with the easiest choice being two *embedding matrices* $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{d \times V}$, such that:

$$\boldsymbol{m}_i = \boldsymbol{A}\boldsymbol{x}_i \qquad \text{(Memory Transformation)}$$

$$\boldsymbol{u} = \boldsymbol{B}\boldsymbol{q} \qquad \text{(Query Transformation)}$$

As soon as we obtain the above, the matching vector $\boldsymbol{p}$ between the query transformation and each memory object can be computed by passing the product of the two, $\boldsymbol{u}^\top \boldsymbol{m}_i$, through a Softmax activation:

$$\boldsymbol{p}_i = Softmax(\boldsymbol{u}^\top \cdot \boldsymbol{m}_i) = \frac{e^{\boldsymbol{u}^\top \boldsymbol{m}_i}}{\sum_j e^{\boldsymbol{u}^\top \boldsymbol{m}_j}} \qquad \text{(Memory Matching)}$$

The inner product can be seen as a measure of similarity (reminiscent of Cosine Similarity, minus the normalization), and the softmax activation is used to constrain $\boldsymbol{p}$ within $\mathcal{S}^N$.

A third transformation, via a third embedding matrix $\boldsymbol{C} \in \mathbb{R}^{d \times V}$, is used to map inputs $\{\boldsymbol{x}_i\}$ to their output representations $\{\boldsymbol{c}_i\}$, from which the memory response can be derived as their match-weighted sum:

$$\boldsymbol{c}_i = \boldsymbol{C}\boldsymbol{x}_i \qquad \text{(Output Transformation)}$$

$$\boldsymbol{o} = \sum_i \boldsymbol{p}_i \boldsymbol{c}_i \qquad \text{(Memory Response)}$$

The ingenuity of this last equation is perhaps masked by its simplicity. It essentially accounts for a form of soft attention; the query dictates which parts of the output representation to focus on, depending on how it matched with the input representation. In other words, even though we always get the same output representation sequence $\{\boldsymbol{c}_i\}$ from an input sequence $\{\boldsymbol{x}_i\}$, the query $\boldsymbol{q}$ plays an important role on how the network utilizes it.

Finally, the memory response is combined with the query representation through some binary operation (e.g. concatenation, element-wise addition or multiplication, etc.) and is passed through one last layer responsible for the final prediction. In the original paper, which assumes a classification setting, the authors propose a Softmax-activated linear map over the addition of the query and the response:

$$\hat{\boldsymbol{a}} = Softmax(\boldsymbol{W}(\boldsymbol{o} + \boldsymbol{u})) \qquad \text{(Final Prediction)}$$

Of course, this may be altered according to the requirements and restrictions imposed by the particular experimental environment.

Multi-Layer Case

5 Comparison

6 Conclusion