NEURAL TURING MACHINES

Motivation

For the first thirty years of artificial intelligence research, neural networks were largely seen as an unpromising research direction. From the 1950s to the late 1980s, AI was dominated by a symbolic approach, which attempted to explain how information processing systems like the human brain might function in terms of symbols, structures and rules that could manipulate said symbols and structures. It wasn't until 1986 that a serious alternative theory emerged to challenge symbolic AI; its authors used the term Parallel Distributed Processing, but the more commonly used term today is connectionism. You might not have heard of this approach, but you've likely heard of its most famous modeling technique - artificial neural networks.

Two criticisms were made against neural networks as tools capable of helping us better understand intelligence. First, neural networks with fixed-size inputs are seemingly unable to solve problems with variable-size inputs. Second, neural networks seem unable to bind values to specific locations in data structures. This ability of writing to and reading from memory is critical in the two information processing systems we have available to study: brains and computers. How could these two criticisms be answered?

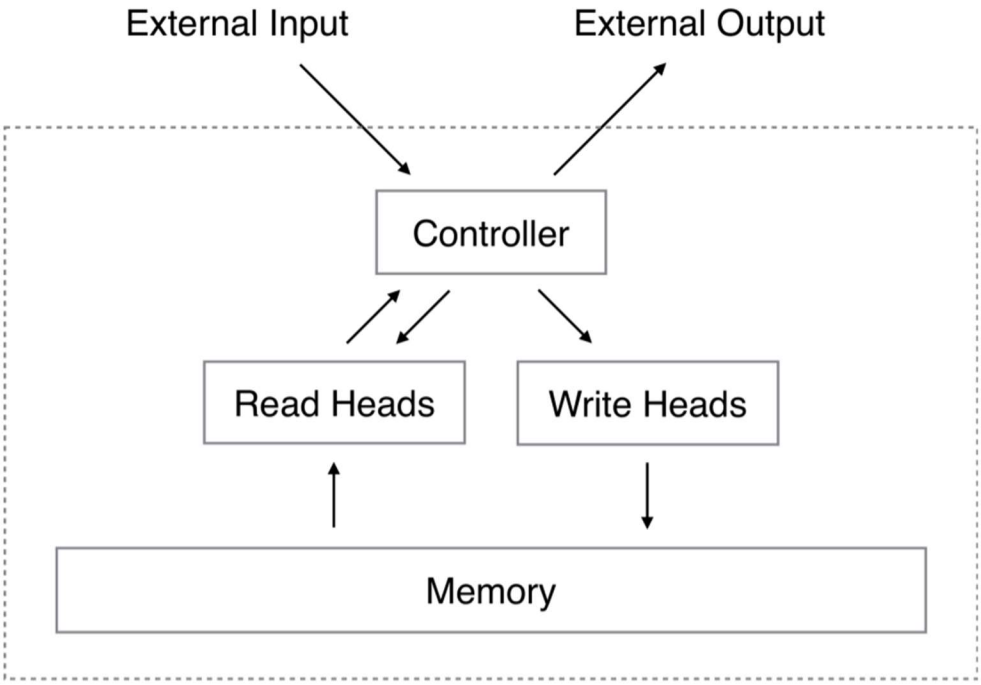
The first was answered with the creation of recurrent neural networks (RNNs). RNNs can process variable-size inputs without needing to be modified by adding a time component to their operation - when translating a sentence, or recognizing handwritten text, an RNN repeatedly receives a fixed-size input for as many time steps as is necessary. In this paper, Graves et al. seek to answer the second criticism by giving a neural network an external memory and the capacity to learn how to use it. They call their system a **Neural Turing Machine (NTM)**.

Background

For computer scientists, the need for a memory system is clear. Computers have advanced tremendously over the past half century, but they are still composed of the three components: memory, control flow and arithmetic/logical operations. However, there's also biological evidence to suggest that having a memory system for quick storage and retrieval of information is helpful. This memory system has been termed working memory, and the NTM paper lists several earlier papers that have studied working memory from a computational neuroscience perspective.

Intuition

A NTM is fundamentally composed of a neural network, called the **controller**, and a 2D matrix called the memory bank, memory matrix or just plain **memory**. At each time step, the neural network receives some input from the outside world, and sends some output to the outside world. However, the network also has the ability to read from select memory locations and the ability to write to select memory locations. Graves et al. draw inspiration from the traditional Turing machine and use the term "head" to describe the specification of memory location(s). In the below image, the dotted line demarcates which parts of the architecture are "inside" the system as opposed to the outside world.



But there's a catch. Suppose that we index the memory $\mathbb{M}$ by specifying the row and the column, just like a typical matrix. We'd like to train our NTM using backpropagation and our favorite optimization method (e.g. stochastic gradient descent), but how do we take the gradient of a specific index? We can't. Instead, the controller will read and write using "blurry" operations that interact with all elements in memory to varying degrees. The Controller will produce weightings over memory locations that allow it to specify memory locations in a differentiable manner. Following the paper's lead, I'll explain how these weight vectors are generated after explaining how they're used (doing so makes understanding the system easier).

Mathematics

Reading

Let's refer to the memory matrix, with R rows and C elements per row, at time t as M_t. In order to read (and write), we'll need an attention mechanism that dictates where the head should read from. The attention mechanism will be a length-R normalized weight vector w_t. We'll refer to individual elements in the weight vector as $w_t(i)$. By "normalized," the authors mean that the following two constraints hold:

$$0 \leq w_t(i) \leq 1 \quad \sum_{i=1}^{R} w_t(i) = 1 \quad (1) \quad (1)$$

The read head will return a length-C vector r_t that is a linear combination of the memory's rows $M_t(i)$ scaled by the weight vector:

$$r_t \leftarrow \sum_{i}^{R} w_t(i) M_t(i) \quad (2) \quad (2)$$

Writing

Writing is a little trickier than reading, since writing involves two separate steps: erasing, then adding. In order to erase old data, a write head will need a new vector, the length-C erase vector e_t, in addition to our length-R normalized weight vector w_t. The erase vector is used in conjunction with the weight vector to specify which elements in a row should be erased, left unchanged, or something in between. If the weight vector tells us to focus on a row, and the erase vector tells us to erase an element, the element in that row will be erased.

$$M_{erased_t}(i) \leftarrow M_{t-1}(i)[1 - w_t(i) e_t] \quad (3) \quad (3)$$

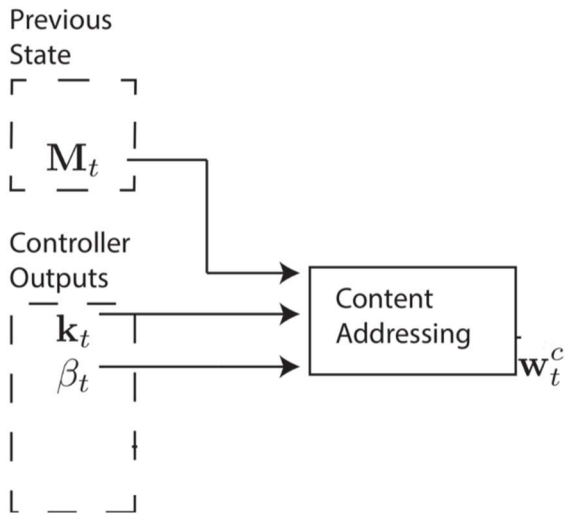
After M_{t-1} has been converted to M_{erased_t}, the write head uses a length-C add vector a_t to complete the writing step.

$$M_t(i) \leftarrow M_{erased_t}(i) + w_t(i) a_t \quad (4) \quad (4)$$

Addressing

Creating these weight vectors to determine where to read and write is tricky, so I'd like to step through the four-stage process. Each stage generates an intermediate weight vector that gets passed to the next stage. The first stage's goal is to generate a weight vector based on how similar each row in memory is to a length-C vector k_t emitted by the controller. I'll refer to this

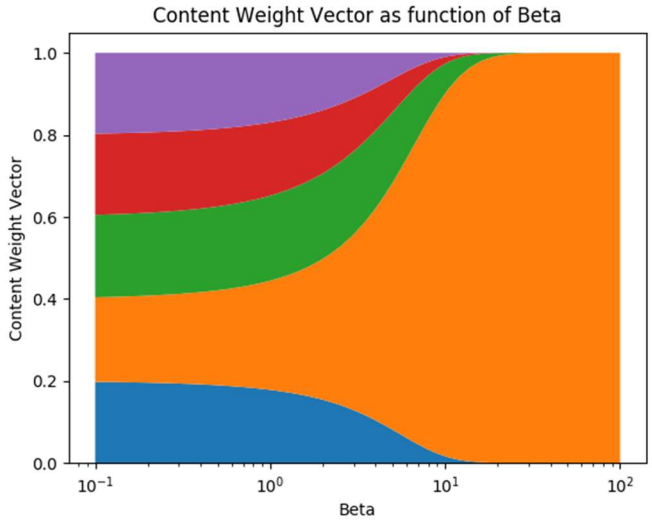
intermediary weight vector w_{ct}wtc as the content weight vector. Another parameter, β_tβt, will be explained in just a second.



This content weight vector allows the controller to select values similar to previously seen values, which is called content-based addressing. For each head, the controller produces a key vector $k_t$kttthat is compared to each row of M_tMt using a similarity measure. In this paper, the authors use cosine similarity, defined as:

$$K(u,v)=u\cdot v\|u\|\cdot\|v\|(6)(6)K(u,v)=u\cdot v\|u\|\cdot\|v\|$$

A positive scalar parameter $\beta_t>0$βt>0, called the key strength, is used to determine how concentrated the content weight vector should be. For small values of beta, the weight vector will be diffuse, but for larger values of beta, the weight vector will be concentrated on the most similar row in memory. To visualize this, if a a key and memory matrix produces a similarity vector `[0.1, 0.5, 0.25, 0.1, 0.05]`, here's how the content weight vector varies as a function of beta.

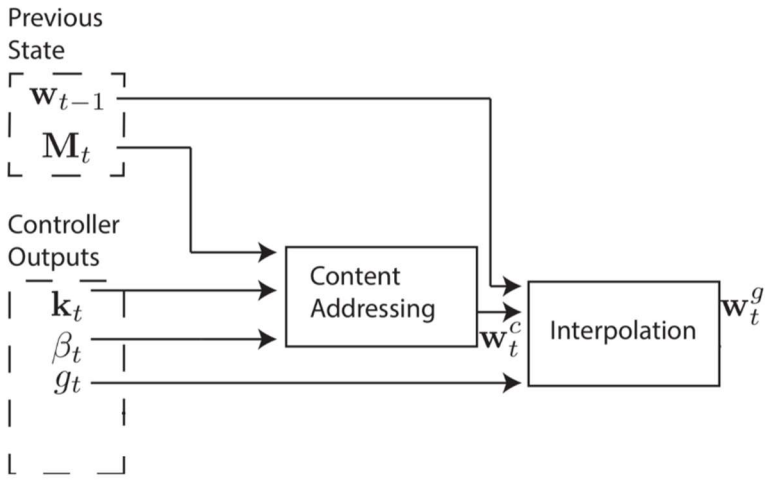


The content weight vector thus is calculated as follows:

$$w_{ct}(i)=\exp(\beta_t K(k_t,M_t(i)))\sum_j \exp(\beta_t K(k_t,M_t(j)))(5)(5)wtc(i)=\exp(\beta tK(kt,Mt(i)))\sum jexp(\beta tK(kt,Mt(j)))$$

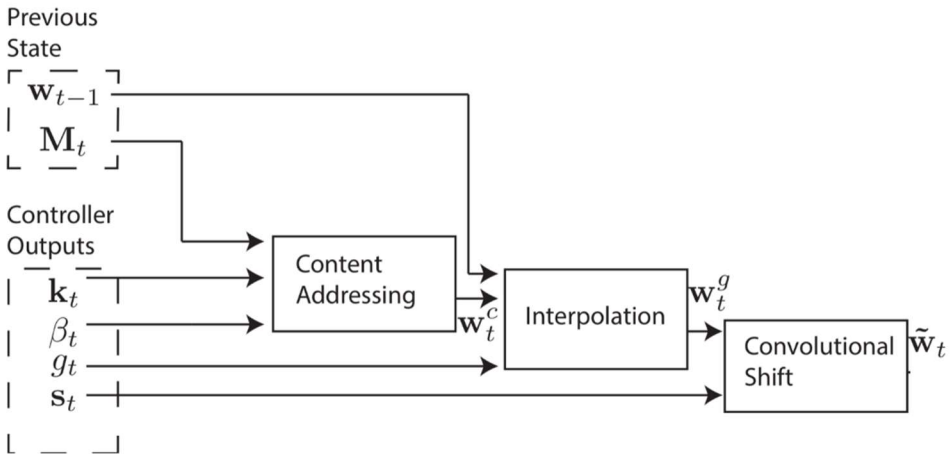
However, in some cases, we may want to read from specific memory locations instead of reading specific memory values. The example the authors give is the function $f(x,y)=x*yf(x,y)=x*y$. In this case, we don't care what the values of x and y are, just that x and y are consistently read from the same memory locations. This is called location-based addressing, and to implement it, we'll need three more stages. In the second stage, a scalar parameter $g_t\in(0,1)gt\in(0,1)$, called the interpolation gate, blends the content weight vector $w_{ct}wtc$ with the previous time step's weight vector $w_{t-1}wt-1$ to produce the gated weighting $w_{gt}wtg$. This allows the system learn when to use (or ignore) content-based addressing.

$$w_{gt}\leftarrow g_t w_{ct}+(1-g_t)w_{t-1}(7)(7)wtg\leftarrow gtwtc+(1-gt)wt-1$$



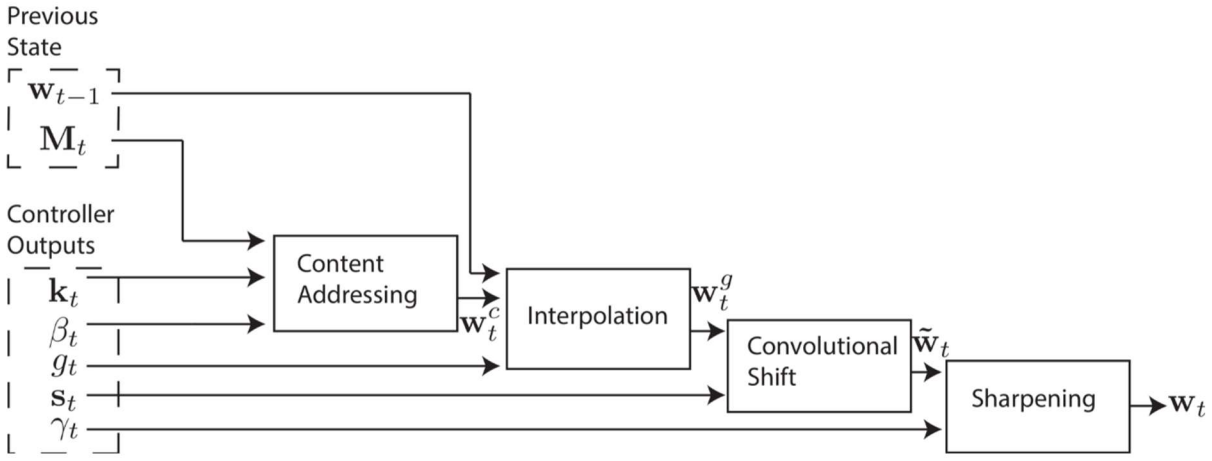
We'd like the controller to be able to shift focus to other rows. Let's suppose that as one of the system's parameters, the range of allowable shifts is specified. For example, a head's attention could shift forward a row (+1), stay still (0), or shift backward a row(-1). We'll perform the shifts modulo RR so that a shift forward at the bottom row of memory moves the head's attention to the top row, and similarly for a shift backward at the top row. After interpolation, each head emits a normalized shift weighting $s_t st$, and the following convolutional shift is performed to produce the shifted weight $w\sim_t w\sim t$.

$$w\sim_t(i)\leftarrow\sum_{j=0R-1} w_{gt}(j)s_t(i-j)(8)(8)w\sim t(i)\leftarrow\sum j=0R-1wtg(j)st(i-j)$$



The fourth and final stage, sharpening, is used to prevent the shifted weight $w\sim_t w\sim t$ from blurring. To do this, a scalar $\gamma \geq 1 \gamma \geq 1$ is required.

$$w_t(i) \leftarrow w\sim_t(i)_{\gamma_t} \sum_j w\sim_t(j)_{\gamma_t} (9)(9) w_t(i) \leftarrow w\sim t(i) \gamma t \sum j w\sim t(j) \gamma t$$



And that's it! A weight vector can be computed that determines where to read from and write to, and better yet, the system is entirely differentiable and thus end-to-end trainable.

NTM-Lasagne: A Library for Neural Turing Machines

Recurrent Neural Networks (RNNs) have become a major player in sequence modeling over the past few years. Due to their efficiency and flexibility, it is tempting to think that they will constitute the future of sequence learning. However, research in Deep Learning is moving at such a fast pace that no one knows what that future is going to be made of.

In fact, among the recent innovations in Deep Learning, the idea of adding some form of external memory to existing architectures has led to promising results, especially in Natural Language Processing (NLP). For the most part, these *memory-augmented networks* have been motivated by the need to reproduce the notion of working memory theorized by neurosciences, which is responsible for inductive reasoning and the creation of new concepts.

Here, we will focus on one of the earliest examples of memory-augmented neural networks: the Neural Turing Machines (Graves, Wayne & Danihelka, 2014). We will see in details how they work, as well as how they can be suitable for learning algorithmic tasks.

Along with this blog post, Snips is also open sourcing NTM-Lasagne on Github, our Neural Turing Machines library built with Theano, on top of the excellent Lasagne. It includes code, pre-trained models, as well as all the examples covered in this post.

Memory-augmented Neural Networks

A feature that all neural networks share, whether they are recurrent or feed-forward, is that they map data into an abstract representation. However, adding knowledge from the external environment, like a whole thesaurus in NLP, is very hard.

Rather than artificially increasing the size of the hidden state in the RNN, we would like to arbitrarily increase the amount of knowledge we add to the model while making minimal changes to the model itself. Basically, we can augment the model with an independent memory that acts as a *knowledge base* from which the network can read and write on demand. You can think of the neural network as the processing unit (CPU), and this new external memory as the RAM.

The choice of the structure for the memory is crucial to keep the read/write operations fast, no matter the size of the memory. Multiple designs have been proposed by the Deep Learning community. For example the Neural Turing Machines, as well as the Neural Random Access Machines (Kurach & Andrychowicz et al., 2015) and the Neural GPU

(Kaiser et al., 2015), use a tape with read and write heads. In Grefenstette et al., 2015, the authors use continuous versions of structures like stacks or (double-ended) queues.

An interesting side-effect of these memory-augmented networks is the ability to keep track of intermediate computations. For instance, in the case of Question-Answering (QA) problems in NLP, it can be valuable to memorize a story as the model reads it to eventually answer a question. The Memory Networks (Sukhbaatar et al., 2015) and the Dynamic Memory Networks (Kumar et al., 2015), for example, take advantage of the memory to perform well on QA tasks.

The idea of adding extra memory to RNNs is not new. In fact, the Long Short Term Memory networks (LSTMs, Hochreiter & Schmidhuber, 1997) already have a basic memory cell on which information is stored at every time-step. For an introduction to LSTMs, I recommend reading this great post by Chris Olah that details step-by-step how they work, including the role of this memory cell. To make it short, the main function of this cell is to simplify the learning of RNNs, and maintain long term dependencies between elements in the input sequence.

So, what is the point of adding a memory with such low-level addressing mechanisms, compared to LSTMs? The answer is *structure*. LSTMs typically have a distributed representation of information in memory, and perform global changes across the whole memory cell at each step. This is not an issue in itself, and experience has shown that they sure can memorize *some* structure out of the data.

Unlike LSTMs, memory-augmented networks encourage (but don't necessarily *ensure*) local changes in memory. This happens to help not only to find the structure in the training data, but also to generalize to sequences that are beyond the generalization power of LSTMs, such as longer sequences in algorithmic tasks (we will see some examples below).

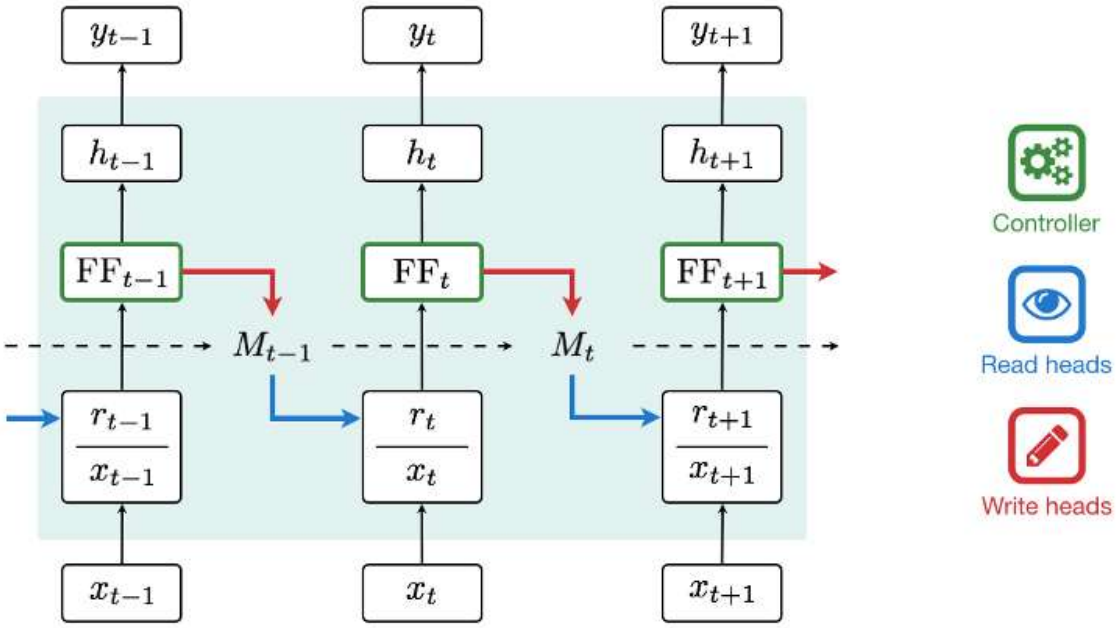
You can picture the value of memory-augmented networks over LSTMs through the idea of the cocktail party effect: imagine that you are at a party, trying to figure out what is the name of the host while listening to all the guests at the same time. Some may know his first name, some may know his last name; it could even be to the point where guests know only parts of his first/last name. In the end, just like with a LSTM, you could retrieve this information by coupling the signals from all the different guests. But you can imagine that it would be a lot easier if a single guest knew the full name of the host to begin with.

Neural Turing Machines

In their respective domains, Computability Theory and Machine Learning have helped pushing forward the abilities of computers. The former defined what a computer is capable of computing, whereas the latter allowed computers to perform tasks that are easy for

humans but were long thought to be impossible for computers, like computer vision for example.

Alan Turing himself studied the strong connections between computing and prospects of Artificial Intelligence in the 1940s. His *Turing Machine* is a classical computational model that operates on an infinite memory tape through a head that either reads or writes symbols. Although this abstract computer has a very low level interface, it is powerful enough to simulate any algorithm. The *Neural Turing Machine* (NTM) takes inspiration from these two fields to make the best of both worlds.



A Neural Turing Machine unfolded in time. The controller (green), a feed-forward neural network, depends on both the input x and the read vector r. It emits read (blue) and write (red) heads to interact with the memory M.

It is worth noting that there is an interesting connection between Turing Machines and RNNs. The latter are known to be *Turing-complete* (Siegelmann, 1995). It means that just like with Turing Machines, any algorithm can be encoded by a RNN with carefully hand-picked parameters (such parameters may not be learnable from data, though).

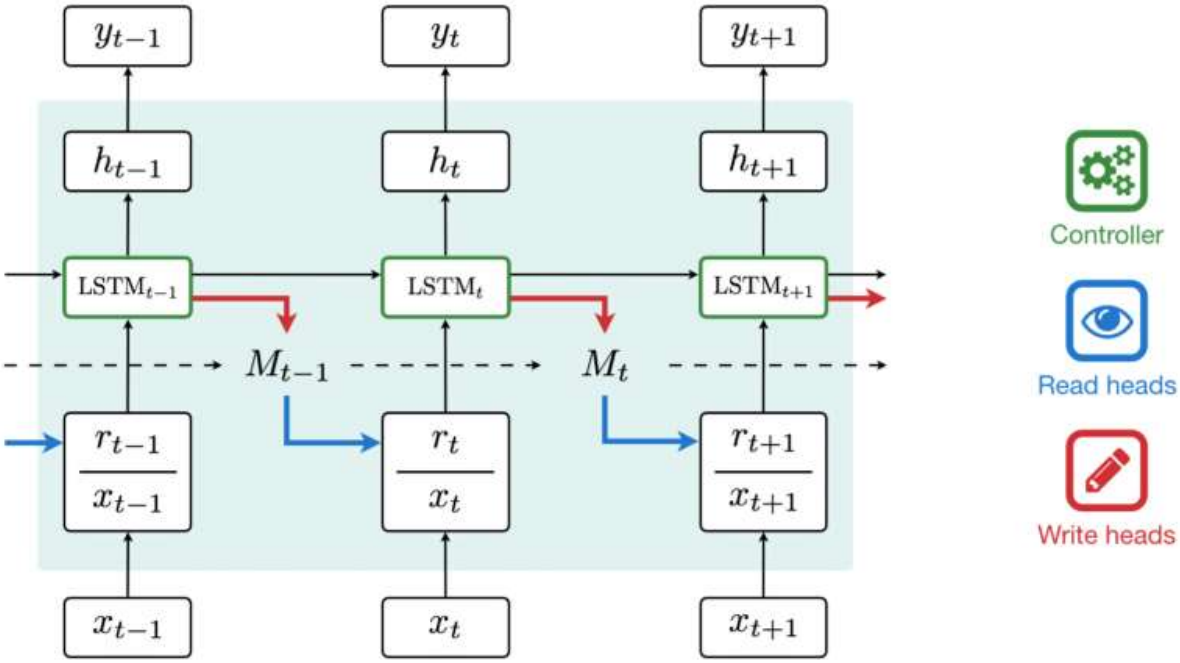
The NTM can be seen as a differentiable version of the Turing Machine. Similarly to a Turing Machine, it has two main components: a (bounded) *memory tape*, and a *controller* that is responsible for making the interface between the external world (ie. the input sequence and the output representation) and the memory through *read and write heads*. This architecture is said to be *differentiable*, in the sense that both the controller and the addressing mechanisms (the heads) are differentiable. The parameters

of the model can then be learned using Stochastic Gradient Descent (SGD). Let's describe these components in more details.

Controller

The controller is a neural network that provides the internal representation of the input that is used by the read and write heads to interact with the memory. Note that this inner representation is not identical to the one that is eventually stored in memory, though the latter is a *function* of this representation.

The type of the controller represents the most significant architectural choice for a Neural Turing Machine. This controller can be either a feed-forward, or recurrent neural network. A feed-forward controller has the advantage over a recurrent controller to be faster, and offers more transparency. This comes at the cost of a lower expressive power though, as it limits the type of computations the NTM can perform per time-step.

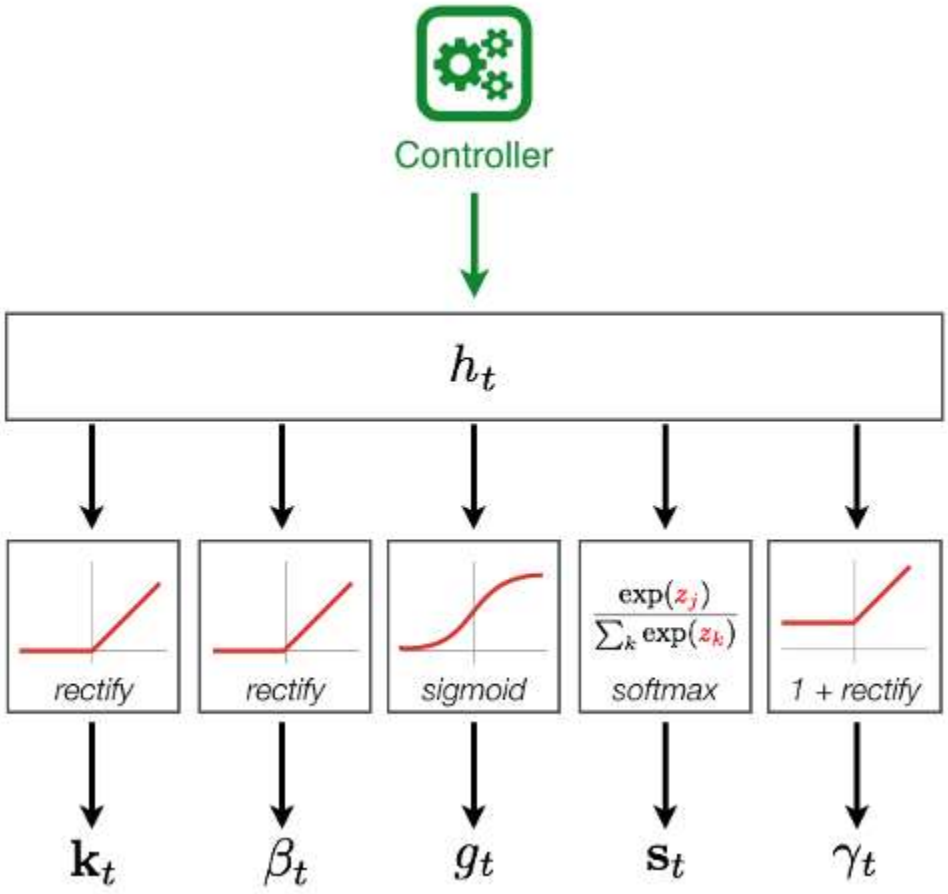


Another example of Neural Turing Machine, where the controller is an LSTM.

Read / Write mechanisms

The read and write heads make the Neural Turing Machines particularly interesting. They are the only components to ever interact directly with the memory. Internally, the behavior of each head is controlled by its own weight vector that gets updated at every time-step. Each weight in this vector corresponds to the *degree of interaction* with each location in

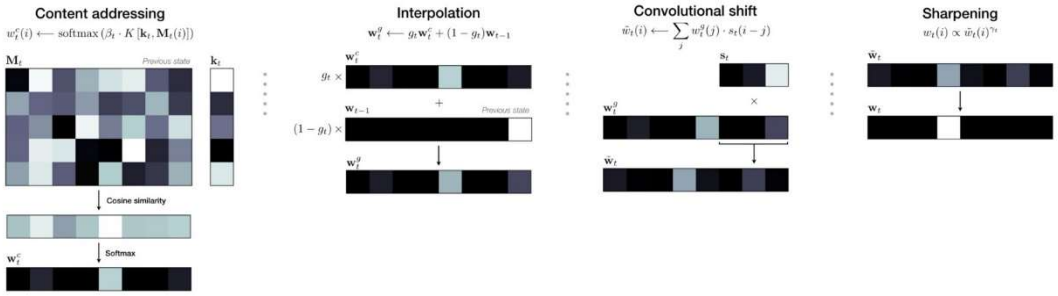
memory (the weight vector sums to 1). A weight of 1 focuses all the attention of the NTM only on the corresponding memory location. A weight of 0 discards that memory location.



Parameters for the weight updates. These parameters are specific to each head (5 parameters per head). Each box corresponds to a 1-layer neural network with the corresponding activation function.

Moreover, we would like these weights to follow two requirements: they should support local changes (read or write) on the memory while keeping their updates differentiable, as we want to train the NTM end to end. To this end, the weight vector gets updated through a series of intermediate smooth operations.

There is a total of four operations per update: *content addressing*, *interpolation*, *convolutional shift* and *sharpening*. They all depend on parameters produced by the controller. More precisely, these parameters are functions of the hidden state emitted by the controller.



Details of the four operations in the update of one of the heads' weight vector. At a higher level, this example can be interpreted as "GOTO pointer, SHIFT LEFT".

In the content-addressing operation, the NTM focuses its attention on the memory locations that are "close" to some *key*. This allows the model to retrieve specific informations in memory. Roughly speaking, a bit like pointers in C. This content-based weighting is then *gated* with the weight vector from the previous time-step in the interpolation operation. If the gate is zero, then the content-addressing is skipped. On the contrary, if the gate is one, then the previous weights are ignored.

The convolutional shift then smoothly shifts the weights left or right. This is very close to the head shifting in a classical Turing Machine. Finally, the shifted vector is sharpened to get a weight vector as focalized as possible. Keep in mind, though, that none of these operations actually *ensures* local changes on the memory, despite the strong emphasis to produce sparse weights. As a consequence, you could end up with a "blurry" weight vector to weakly queries the whole memory.

Once the head has updated its weight vector, it is ready to operate on the memory. If it is a read head, it outputs a weighted combination of the memory locations: the *read vector*. The latter is then fed back to the controller at the following time-step. If it is a write head, the content of the memory is modified softly (depending on its weights) with an *erase* and an *add vector*, both produced by the controller.