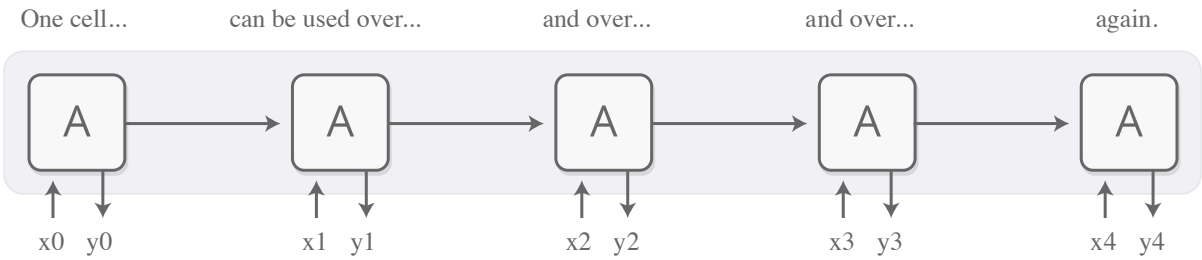
Attention and Augmented Recurrent Neural Networks

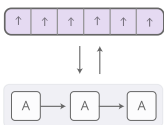
CHRIS OLAH Google Brain
SHAN CARTER Google Brain
Sept. 8 2016
Citation: Olah & Carter, 2016

Recurrent neural networks are one of the staples of deep learning, allowing neural networks to work with sequences of data like text, audio and video. They can be used to boil a sequence down into a high-level understanding, to annotate sequences, and even to generate new sequences from scratch!



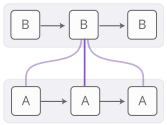
The basic RNN design struggles with longer sequences, but a special variant—"long short-term memory" networks [1]—can even work with these. Such models have been found to be very powerful, achieving remarkable results in many tasks including translation, voice recognition, and image captioning. As a result, recurrent neural networks have become very widespread in the last few years.

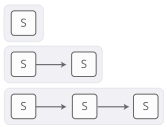
As this has happened, we've seen a growing number of attempts to augment RNNs with new properties. Four directions stand out as particularly exciting:



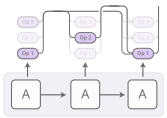
Neural Turing Machines have external memory that they can read and write to.

Attentional Interfaces allow RNNs to focus on parts of their input.





Adaptive Computation Time allows for varying amounts of computation per step.



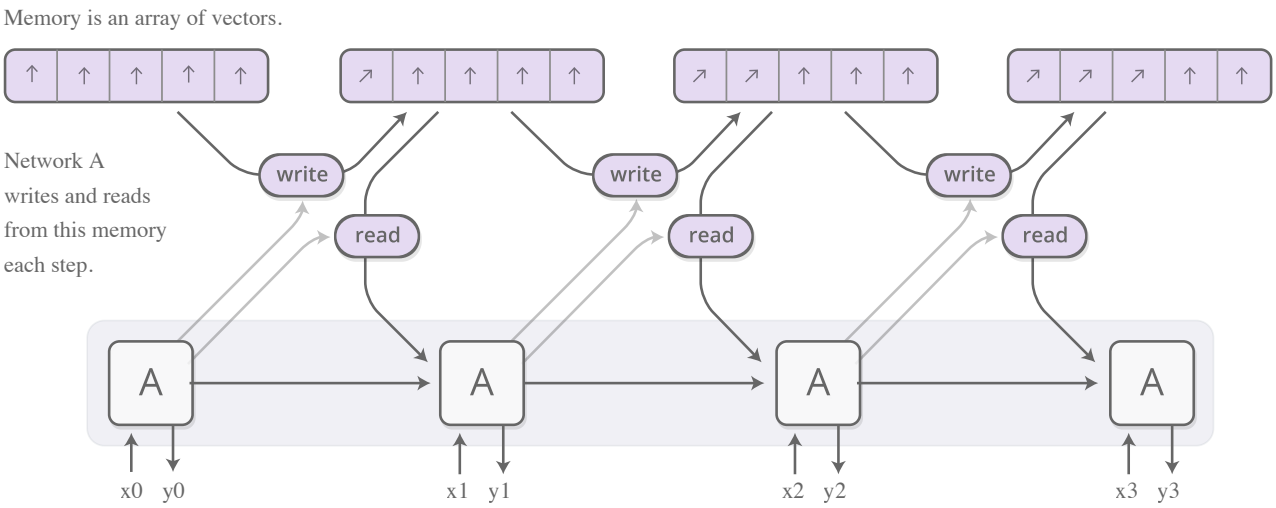
Neural Programmers can call functions, building programs as they run.

Individually, these techniques are all potent extensions of RNNs, but the really striking thing is that they can be combined, and seem to just be points in a broader space. Further, they all rely on the same underlying trick—something called attention—to work.

Our guess is that these "augmented RNNs" will have an important role to play in extending deep learning's capabilities over the coming years.

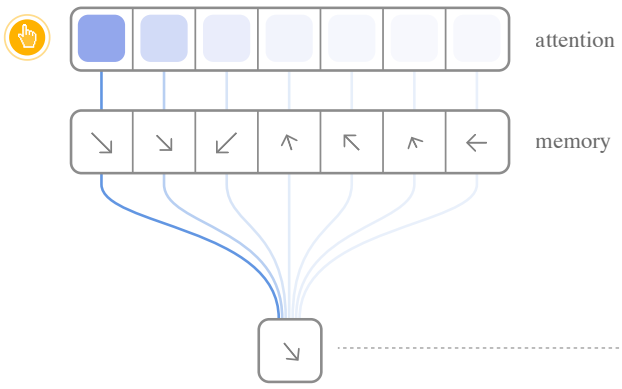
Neural Turing Machines

Neural Turing Machines [2] combine a RNN with an external memory bank. Since vectors are the natural language of neural networks, the memory is an array of vectors:



But how does reading and writing work? The challenge is that we want to make them differentiable. In particular, we want to make them differentiable with respect to the location we read from or write to, so that we can learn where to read and write. This is tricky because memory addresses seem to be fundamentally discrete. NTMs take a very clever solution to this: every step, they read and write everywhere, just to different extents.

As an example, let's focus on reading. Instead of specifying a single location, the RNN outputs an "attention distribution" that describes how we spread out the amount we care about different memory positions. As such, the result of the read operation is a weighted sum.

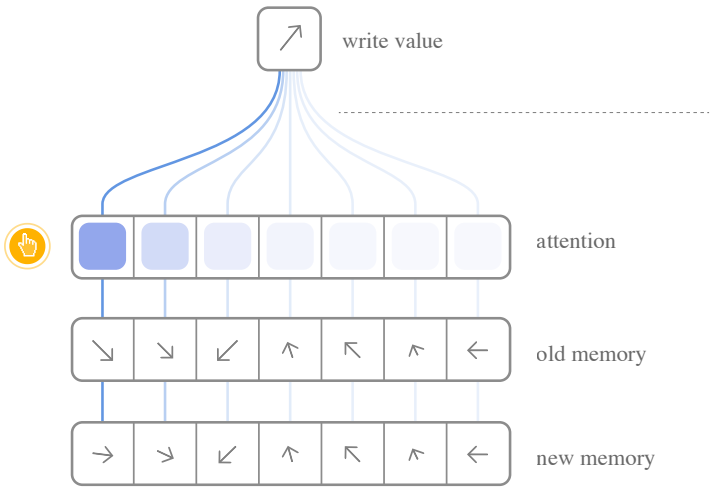


The RNN gives an attention distribution which describe how we spread out the amount we care about different memory positions.

The read result is a weighted sum.

$$r \leftarrow \sum_i a_i M_i$$

Similarly, we write everywhere at once to different extents. Again, an attention distribution describes how much we write at every location. We do this by having the new value of a position in memory be a convex combination of the old memory content and the write value, with the position between the two decided by the attention weight.

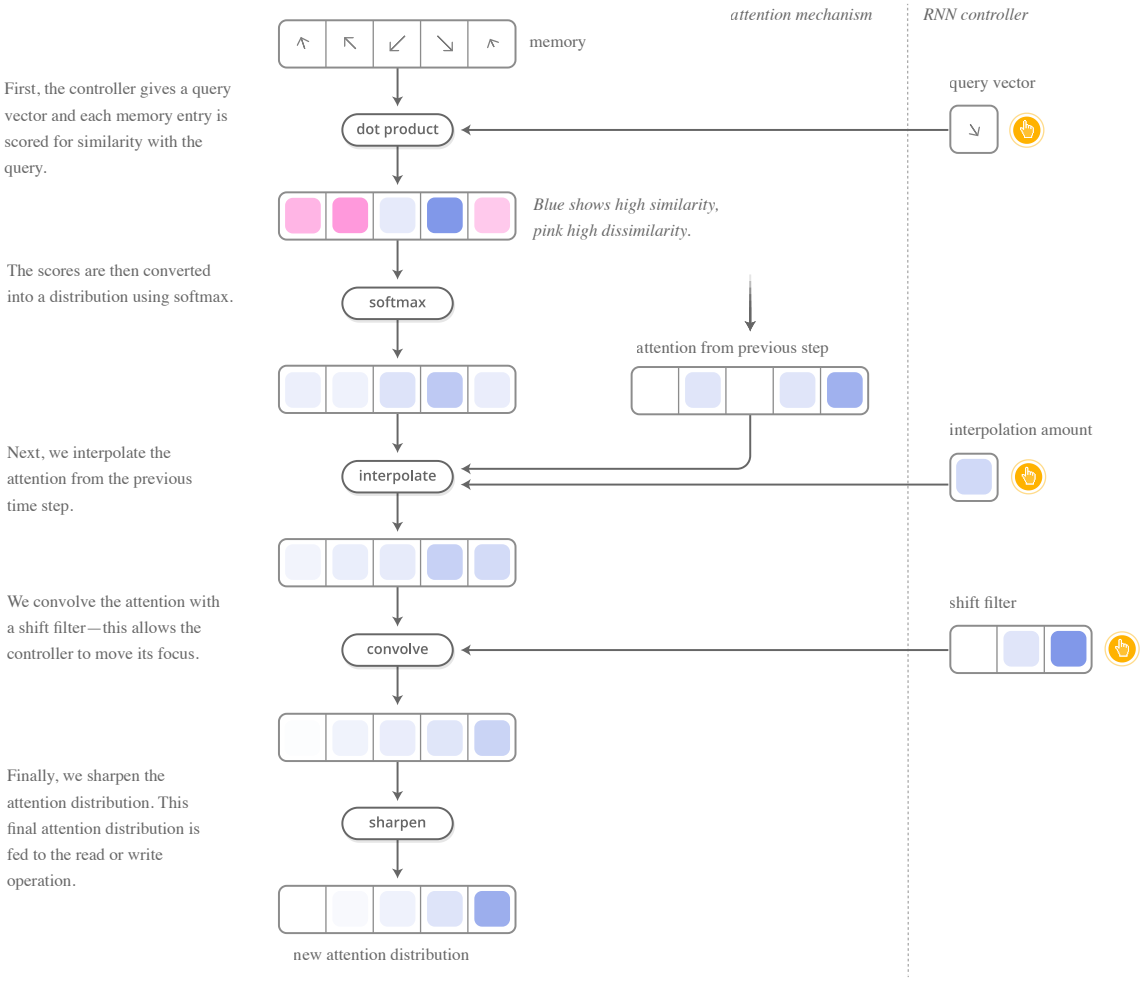


Instead of writing to one location, we write everywhere, just to different extents.

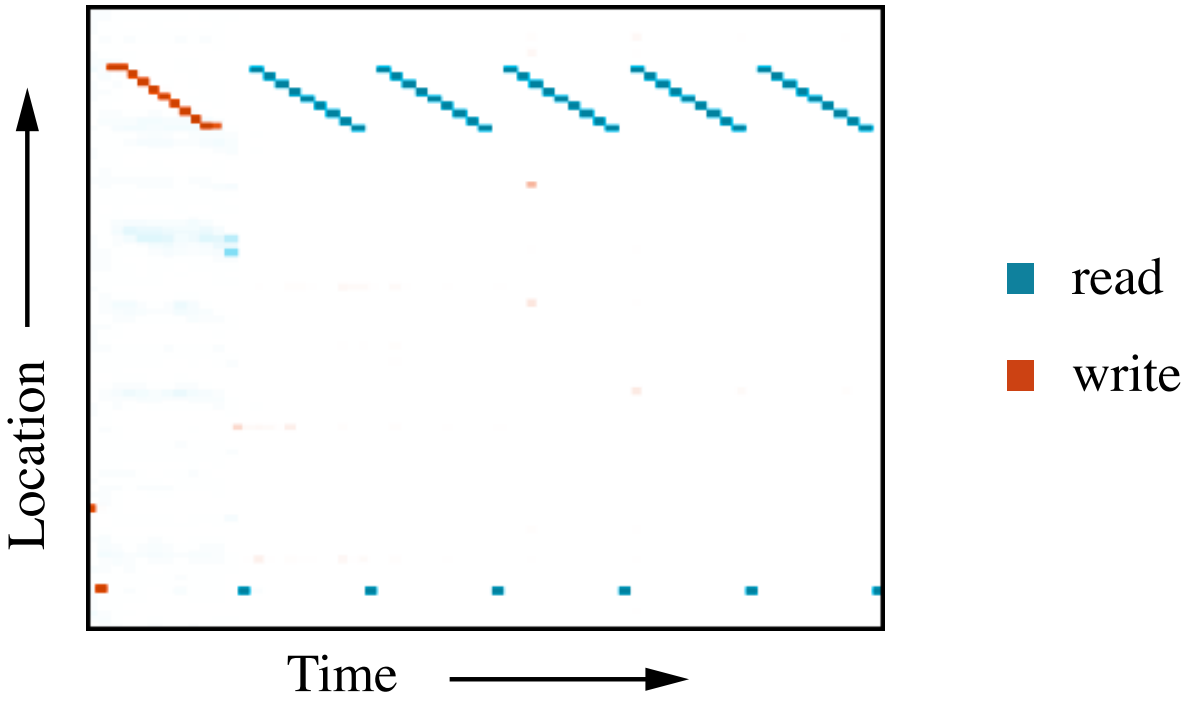
The RNN gives an attention distribution, describing how much we should change each memory position towards the write value.

$$M_i \leftarrow a_i w + (1 - a_i) M_i$$

But how do NTMs decide which positions in memory to focus their attention on? They actually use a combination of two different methods: content-based attention and location-based attention. Content-based attention allows NTMs to search through their memory and focus on places that match what they're looking for, while location-based attention allows relative movement in memory, enabling the NTM to loop.



This capability to read and write allows NTMs to perform many simple algorithms, previously beyond neural networks. For example, they can learn to store a long sequence in memory, and then loop over it, repeating it back repeatedly. As they do this, we can watch where they read and write, to better understand what they're doing:



See more experiments in [3]. This figure is based on the Repeat Copy experiment.

They can also learn to mimic a lookup table, or even learn to sort numbers (although they kind of cheat)! On the other hand, they still can't do many basic things, like add or multiply numbers.

Since the original NTM paper, there have been a number of exciting papers exploring similar directions. The Neural GPU [4] overcomes the NTM's inability to add and multiply numbers. Zaremba & Sutskever [5] train NTMs using reinforcement learning instead of the differentiable read/writes used by the original. Neural Random Access Machines [6] work based on pointers. Some papers have explored differentiable data structures, like stacks and queues [7, 8]. And memory networks [9, 10] are another approach to attacking similar problems.

In some objective sense, many of the tasks these models can perform—such as learning how to add numbers—aren't that objectively hard. The traditional program synthesis community would eat them for lunch. But neural networks are capable of many other things, and models like the Neural Turing Machine seem to have knocked away a very profound limit on their abilities.

Code

There are a number of open source implementations of these models. Open source implementations of the Neural Turing Machine include Taehoon Kim's (TensorFlow), Shawn Tan's (Theano), Fumin's (Go), Kai Sheng Tai's (Torch), and Snip's (Lasagne). Code for the Neural GPU publication was open sourced and put in the TensorFlow Models repository. Open source implementations of Memory Networks include Facebook's (Torch/Matlab), YerevaNN's (Theano), and Taehoon Kim's (TensorFlow).

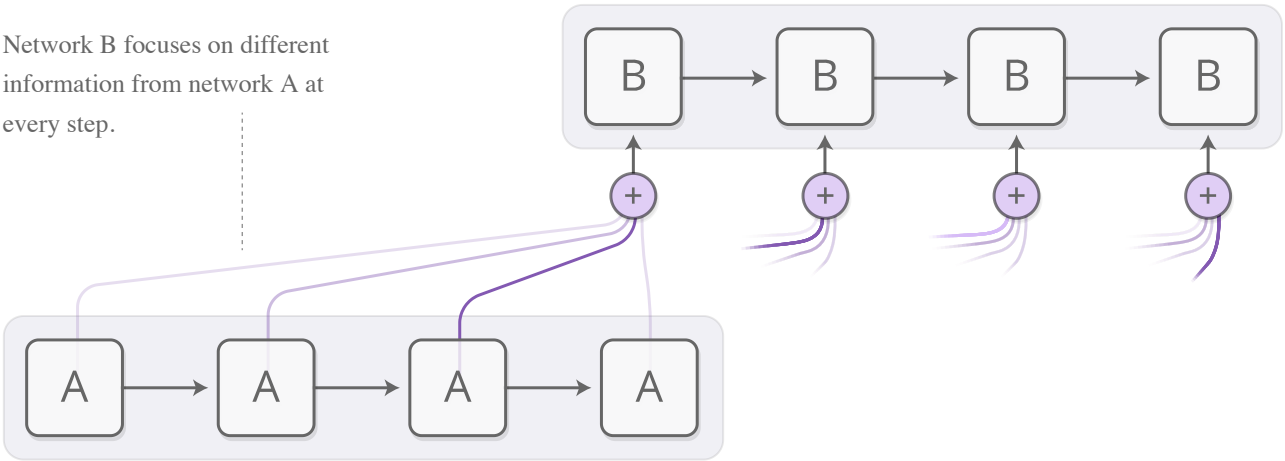
Attentional Interfaces

When I'm translating a sentence, I pay special attention to the word I'm presently translating. When I'm transcribing an audio recording, I listen carefully to the segment I'm actively writing down. And if you ask me to describe the room I'm sitting in, I'll glance around at the objects I'm describing as I do so.

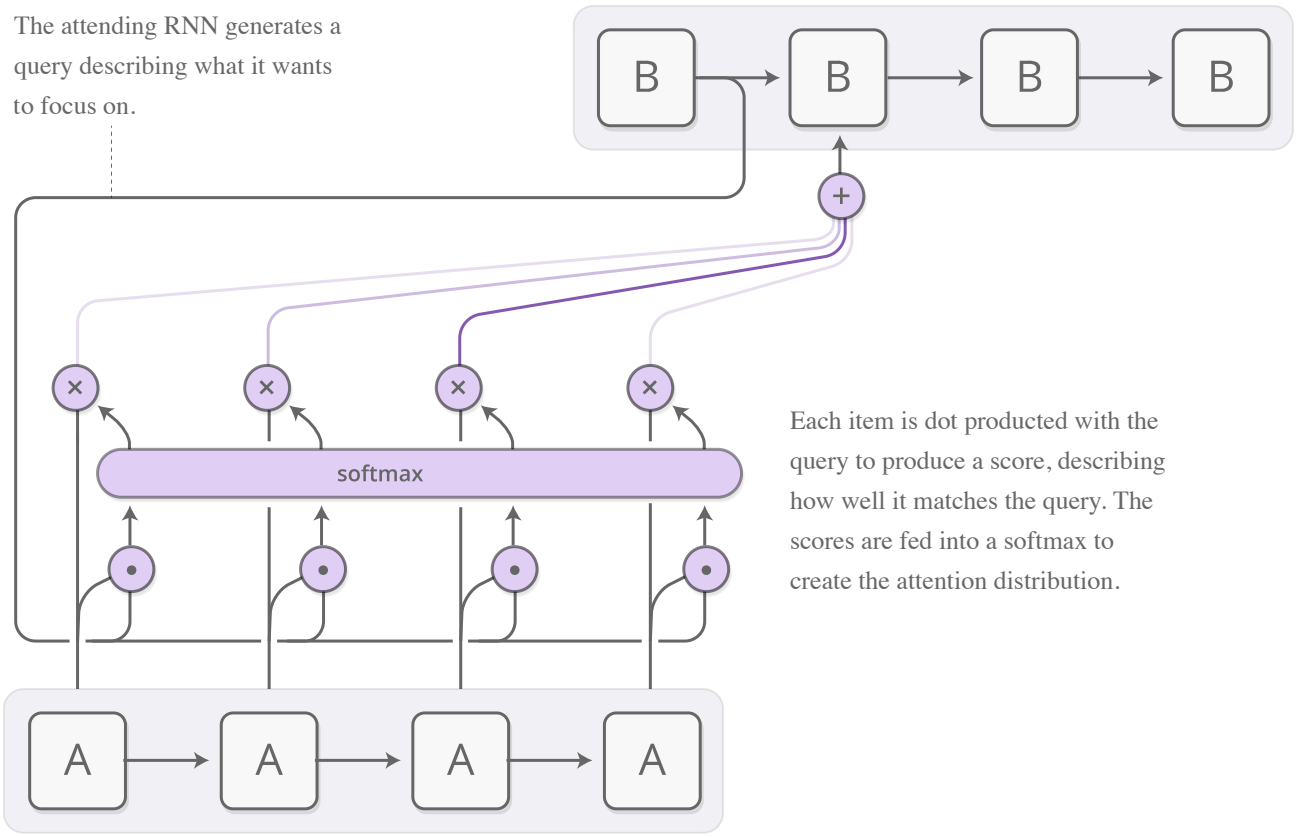
Neural networks can achieve this same behavior using *attention*, focusing on part of a subset of the information they're given. For example, an RNN can attend over the output of another RNN. At every time step, it focuses on different positions in the other RNN.

We'd like attention to be differentiable, so that we can learn where to focus. To do this, we use the same trick Neural Turing Machines use: we focus everywhere, just to different extents.

Network B focuses on different information from network A at every step.



The attention distribution is usually generated with content-based attention. The attending RNN generates a query describing what it wants to focus on. Each item is dot-producted with the query to produce a score, describing how well it matches the query. The scores are fed into a softmax to create the attention distribution.



One use of attention between RNNs is translation [11]. A traditional sequence-to-sequence model has to boil the entire input down into a single vector and then expands it back out. Attention avoids this by allowing the RNN processing the input to pass along information about each word it sees, and then for the RNN generating the output to focus on words as they become relevant.

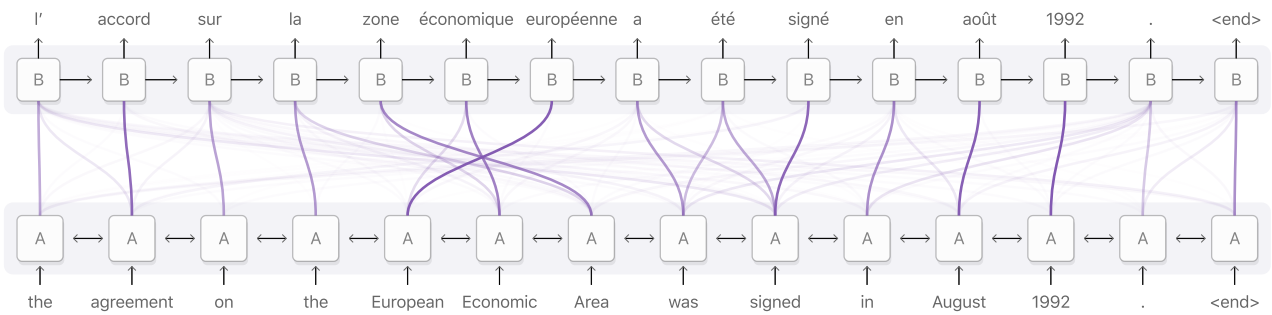


Diagram derived from Fig. 3 of Bahdanau, *et al.* 2014

This kind of attention between RNNs has a number of other applications. It can be used in voice recognition [12], allowing one RNN to process the audio and then have another RNN skim over it, focusing on relevant parts as it generates a transcript.

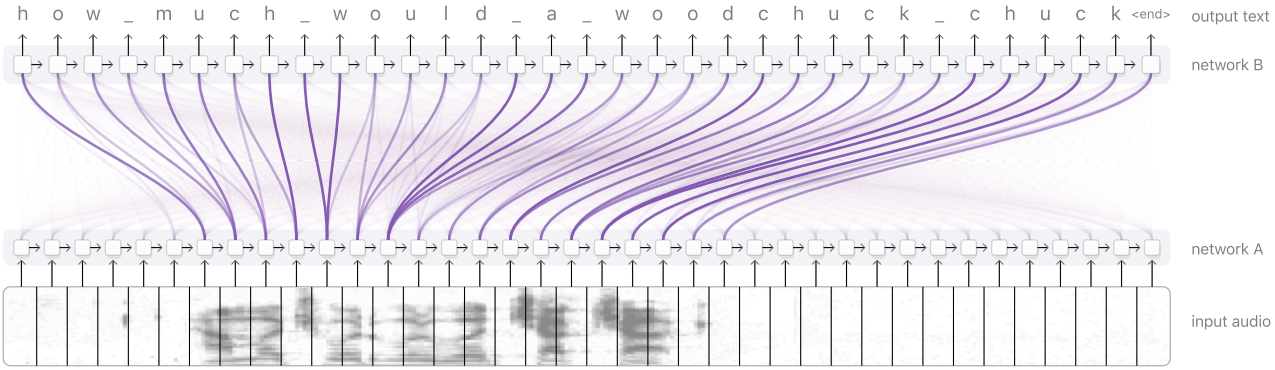


Figure derived from Chan, *et al.* 2015

Other uses of this kind of attention include parsing text [13], where it allows the model to glance at words as it generates the parse tree, and for conversational modeling [14], where it lets the model focus on previous parts of the conversation as it generates its response.

Attention can also be used on the interface between a convolutional neural network and an RNN. This allows the RNN to look at different position of an image every step. One popular use of this kind of attention is for image captioning. First, a conv net processes the image, extracting high-level features. Then an RNN runs, generating a description of the image. As it generates each word in the description, the RNN focuses on the conv net's interpretation of the relevant parts of the image. We can explicitly visualize this:

Figure from [3]

More broadly, attentional interfaces can be used whenever one wants to interface with a neural network that has a repeating structure in its output.

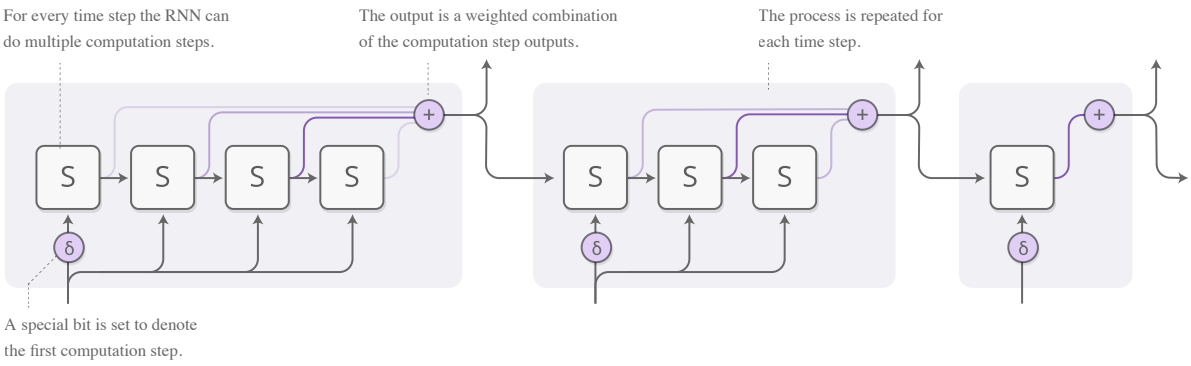
Attentional interfaces have been found to be an extremely general and powerful technique, and are becoming increasingly widespread.

Adaptive Computation Time

Standard RNNs do the same amount of computation for each time step. This seems unintuitive. Surely, one should think more when things are hard? It also limits RNNs to doing $O(n)$ operations for a list of length n.

Adaptive Computation Time [15] is a way for RNNs to do different amounts of computation each step. The big picture idea is simple: allow the RNN to do multiple steps of computation for each time step.

In order for the network to learn how many steps to do, we want the number of steps to be differentiable. We achieve this with the same trick we used before: instead of deciding to run for a discrete number of steps, we have an attention distribution over the number of steps to run. The output is a weighted combination of the outputs of each step.



There are a few more details, which were left out in the previous diagram. Here's a complete diagram of a time step with three computation steps.