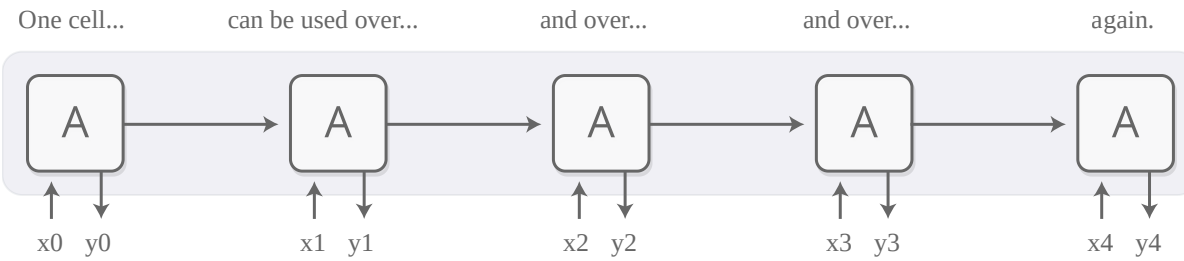
Attention and Augmented Recurrent Neural Networks

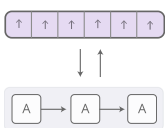
CHRIS OLAH Google Brain
SHAN CARTER Google Brain
Sept. 8 2016
Citation: Olah & Carter, 2016

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!

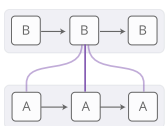


The basic RNN design struggles with longer sequences, but a special variant—"long short-term memory" networks [1]—can even work with these. Such models have been found to be very powerful, achieving remarkable results in many tasks including translation, voice recognition, and image captioning. As a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

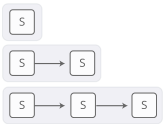
As this has happened, we've seen a growing number of attempts to augment RNNs with new properties. Four directions stand out as particularly exciting:



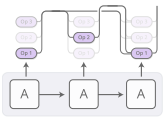
Neural Turing Machines have external memory that they can read and write to.



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



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



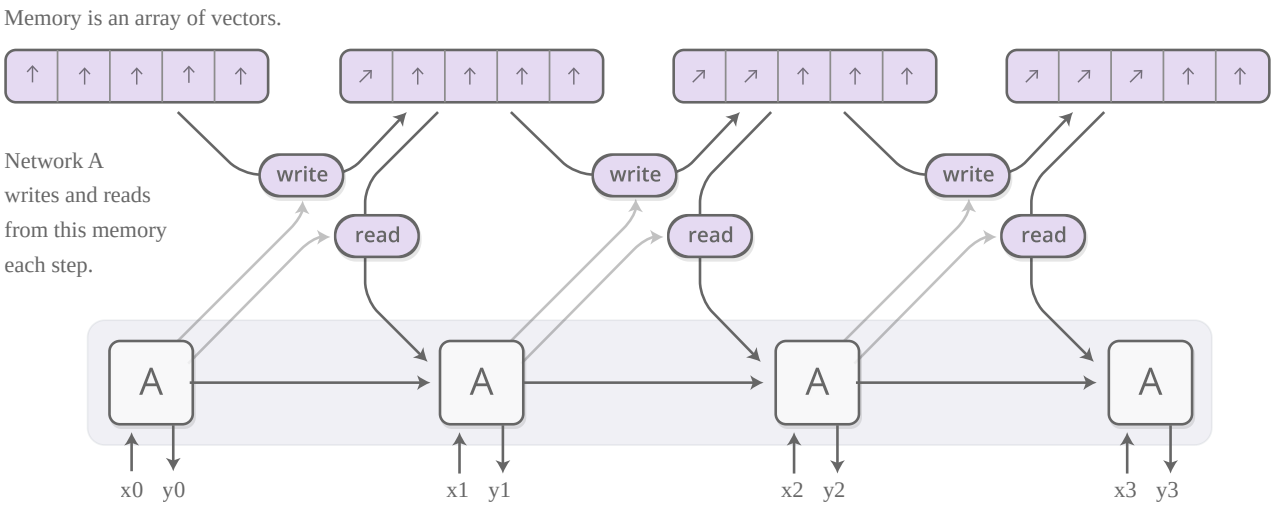
Neural Programmers can call functions, building programs as they run.

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.

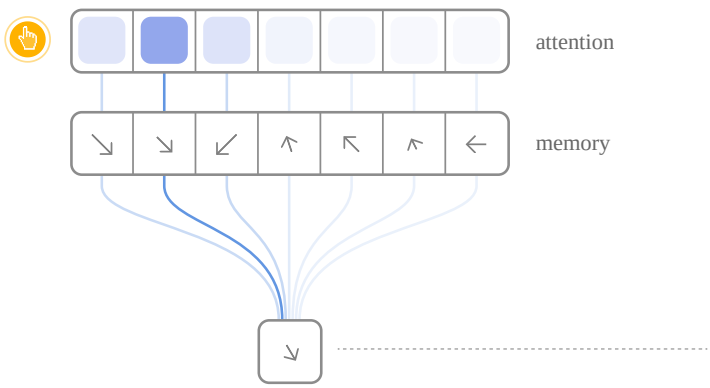
Neural Turing Machines

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.

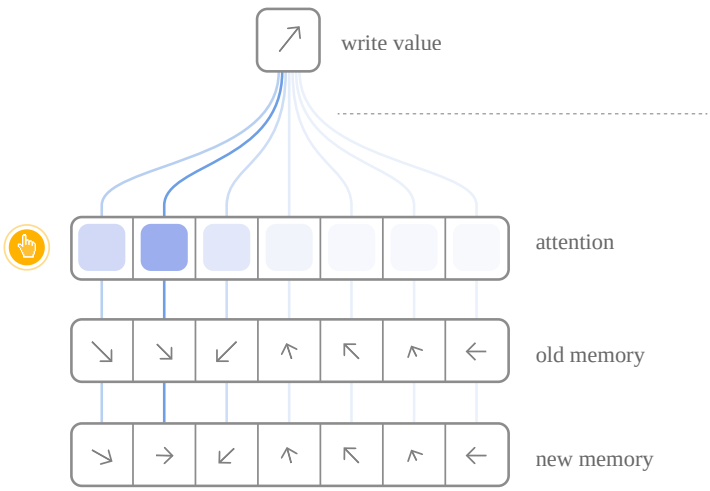


The RNN gives an attention distribution which describe how we spread out the amount we care about different memory positions.

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.

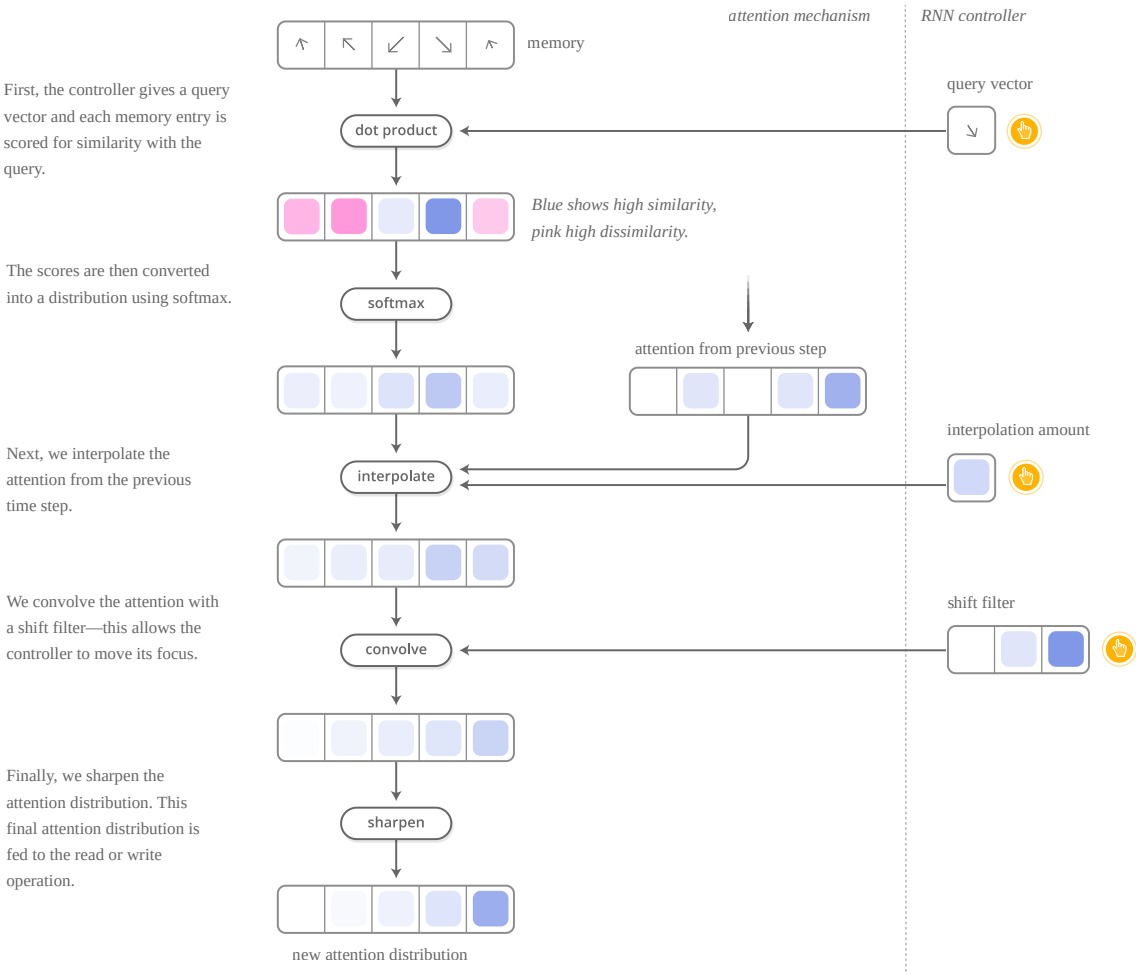


Instead of writing to one location, we write everywhere, just to different extents.

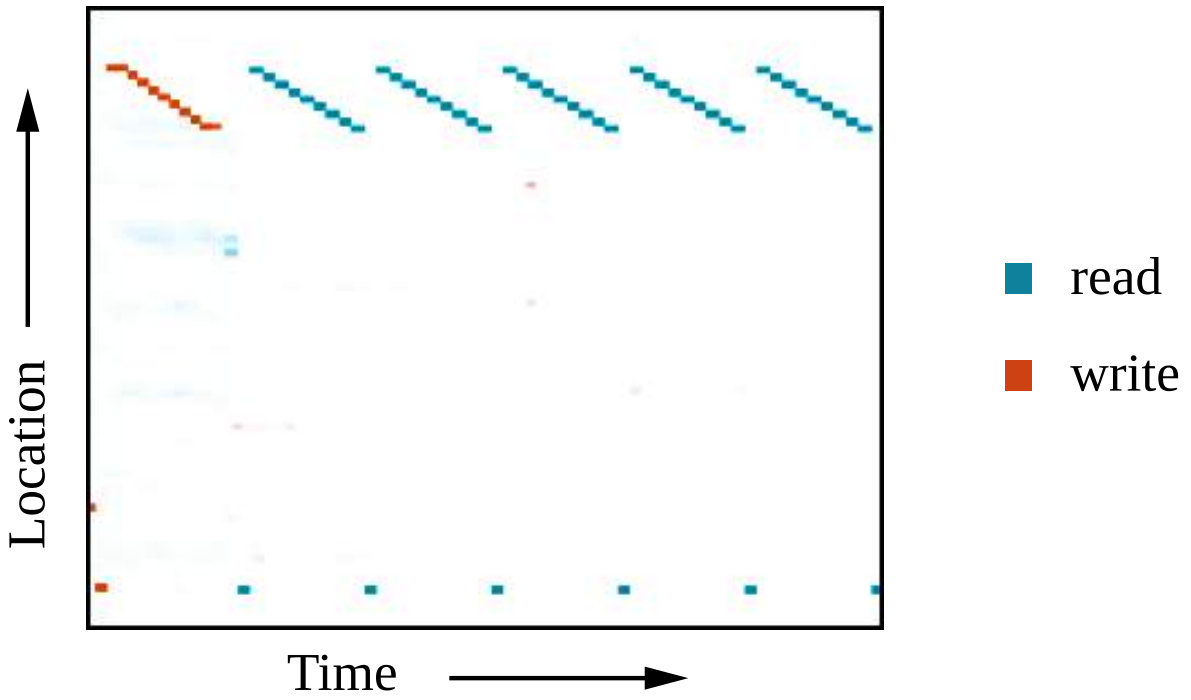
The RNN gives an attention distribution, describing how much we should change each memory position towards the write value.

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

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

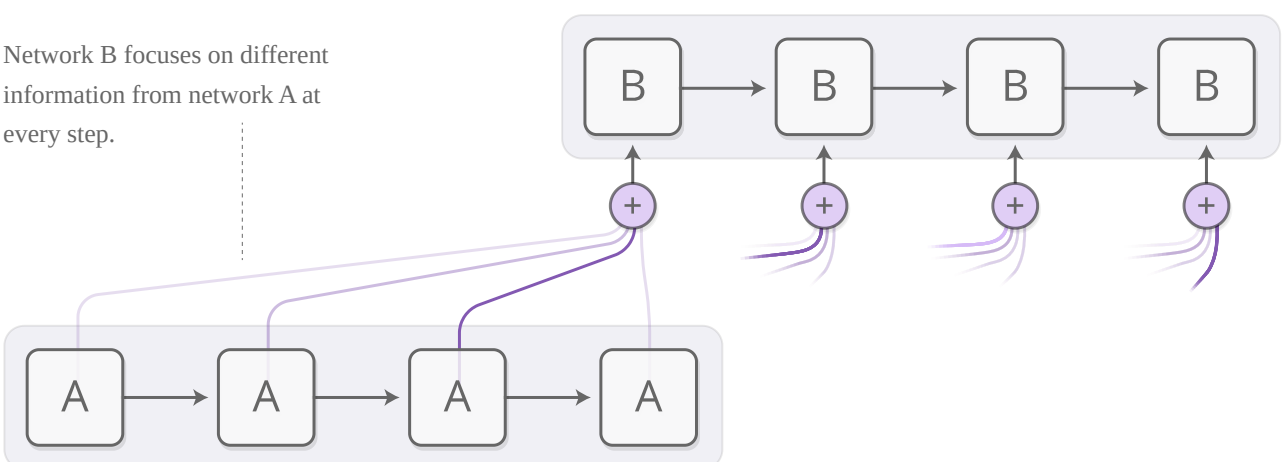
There are a number of open source implementations of these models. Open source implementations of the Neural Turing Machine include Taehoon Kim's (TensorFlow), Shawn Tan's (Theano), Fumin's (Go), Kai Sheng Tai's (Torch), and Snip's (Lasagne). Code for the Neural GPU publication was open sourced and put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

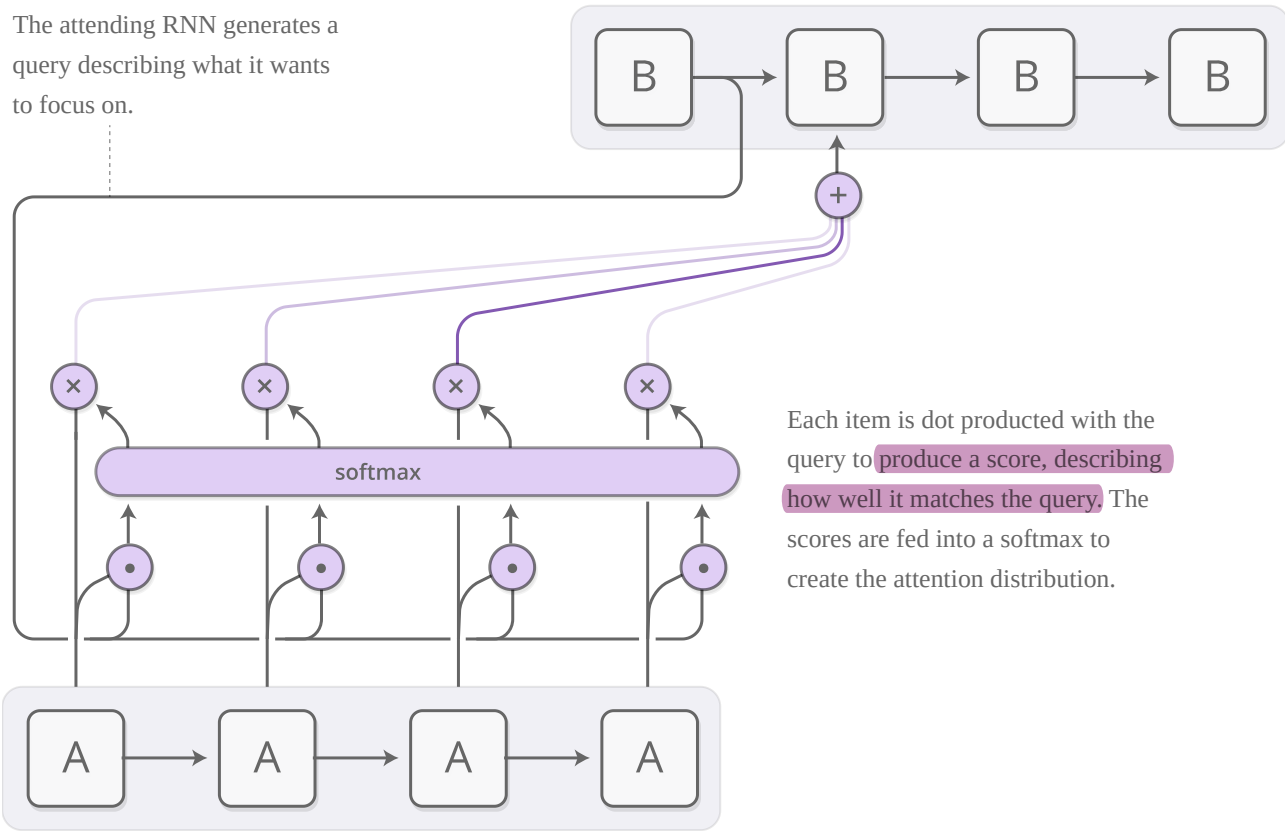
When I'm translating a sentence, I pay special attention to the word I'm presently translating. When I'm transcribing an audio recording, I listen carefully to the segment I'm actively writing down. And if you ask me to describe the room I'm sitting in, I'll glance around at the objects I'm describing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



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.

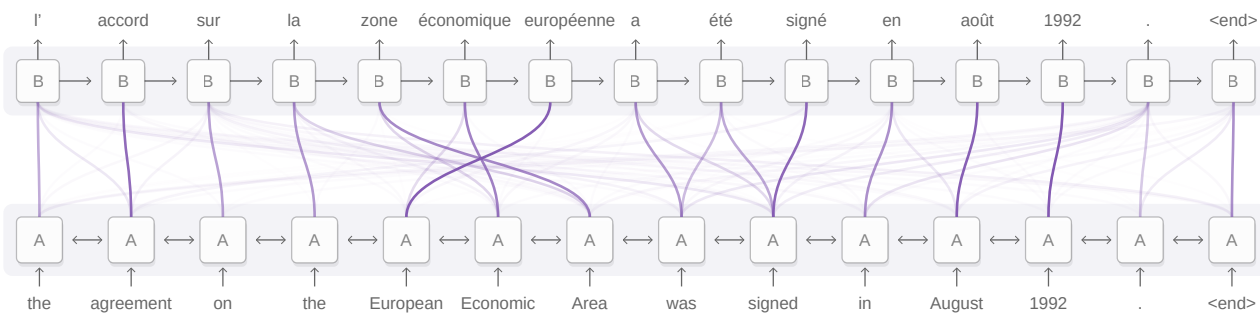


Diagram derived from Fig. 3 of Bahdanau, *et al.* 2014

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.

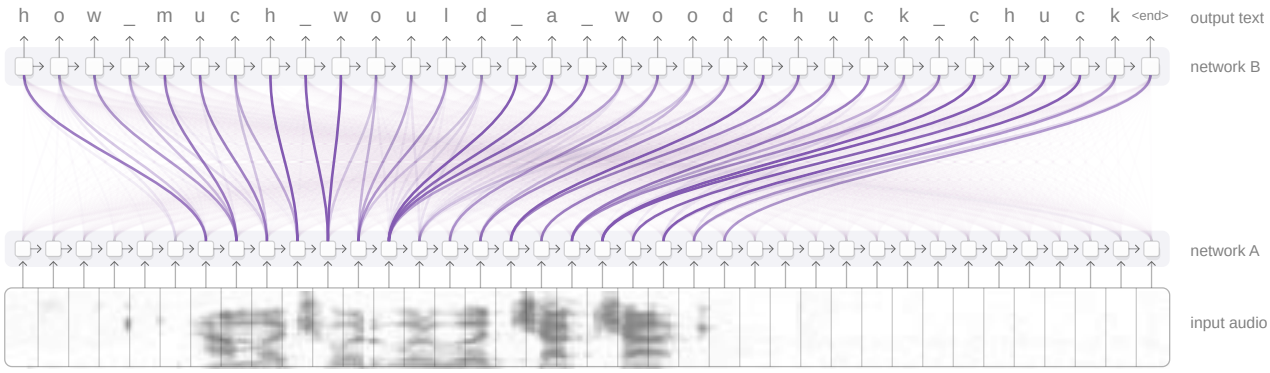


Figure derived from Chan, *et al.* 2015

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:

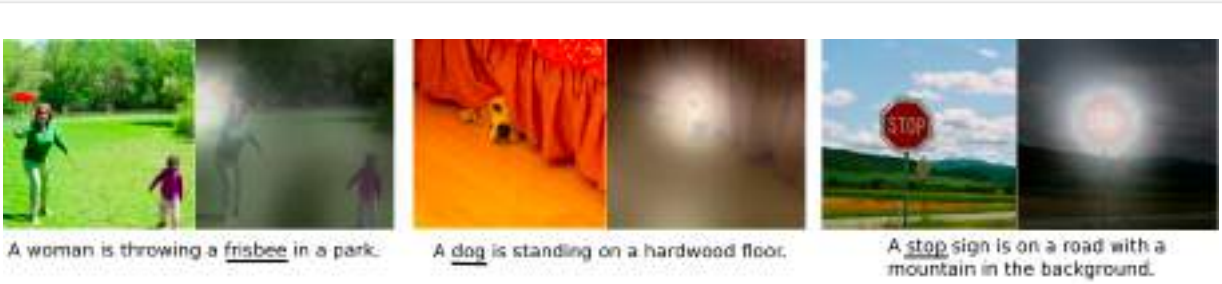


Figure from [3]

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

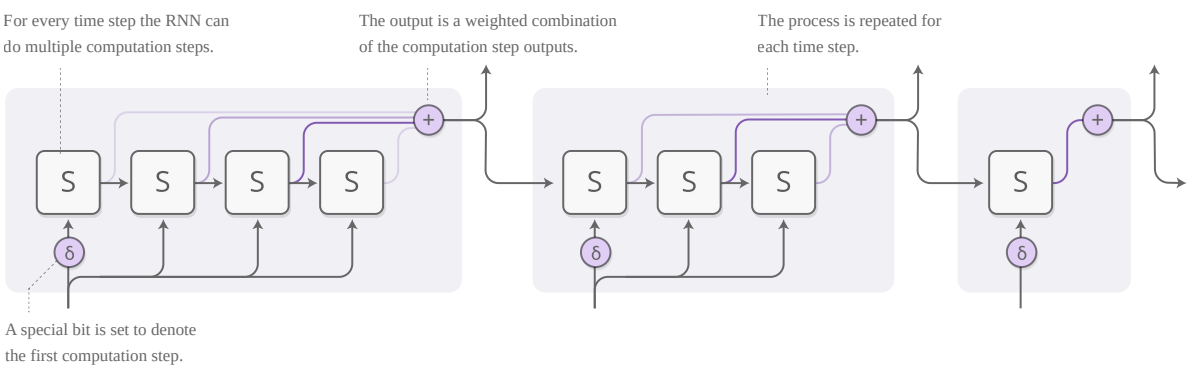
Attentional interfaces have been found to be an extremely general and powerful technique, and are becoming increasingly widespread.

Adaptive Computation Time

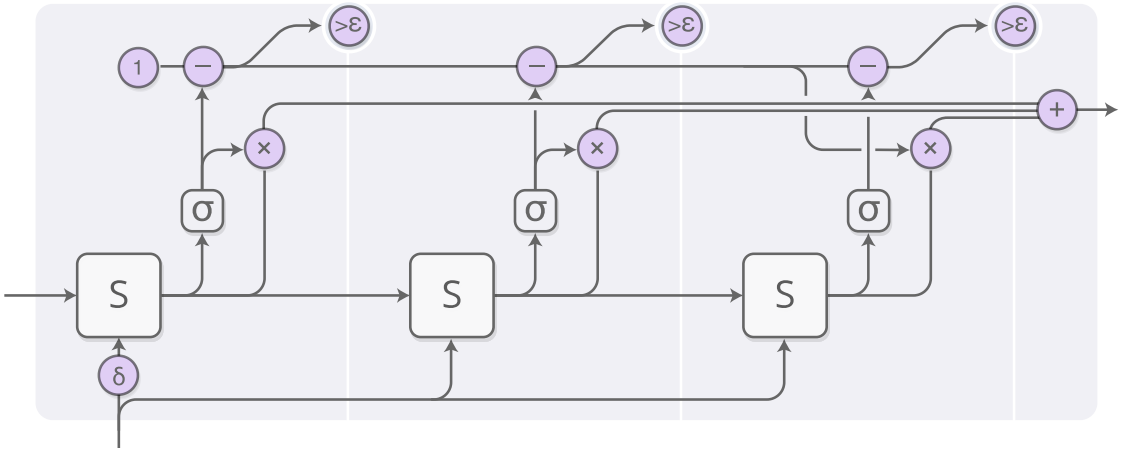
Standard RNNs do the same amount of computation for each time step. This seems unintuitive. Surely, one should think more when things are hard? It also limits RNNs to doing $O(n)$ operations for a list of length n.

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.

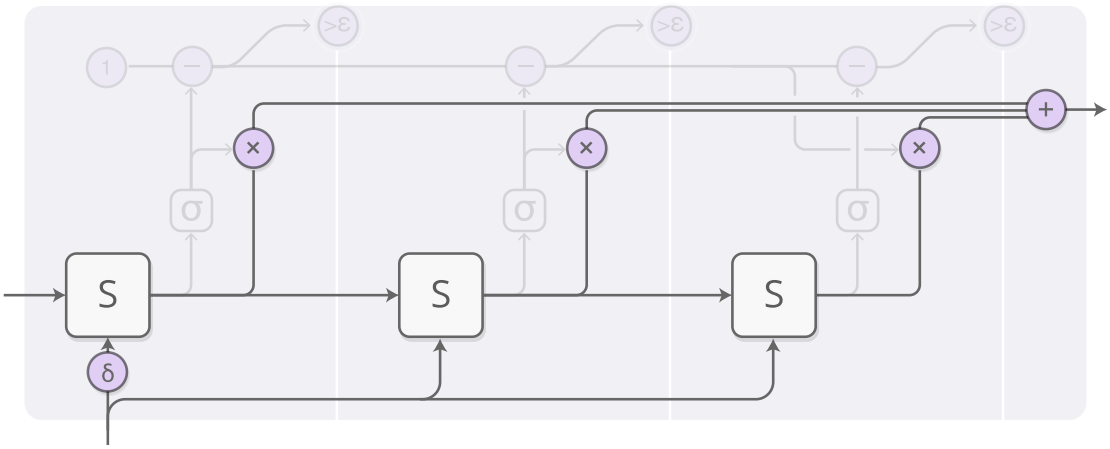
In order for the network to learn how many steps to do, we want the number of steps to be differentiable. We achieve this with the same trick we used before: instead of deciding to run for a discrete number of steps, we have an attention distribution over the number of steps to run. The output is a weighted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



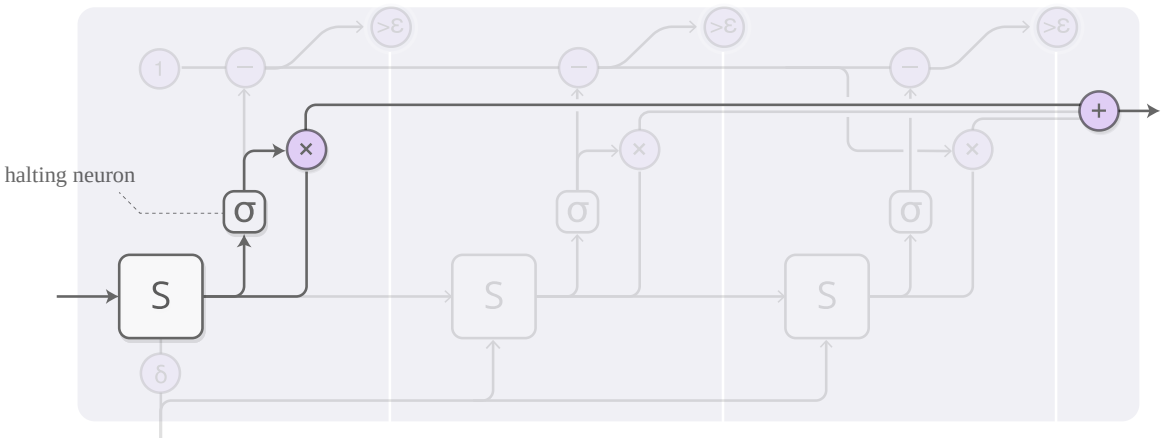
There are a few more details, which were left out in the previous diagram. Here's a complete diagram of a time step with three computation steps.



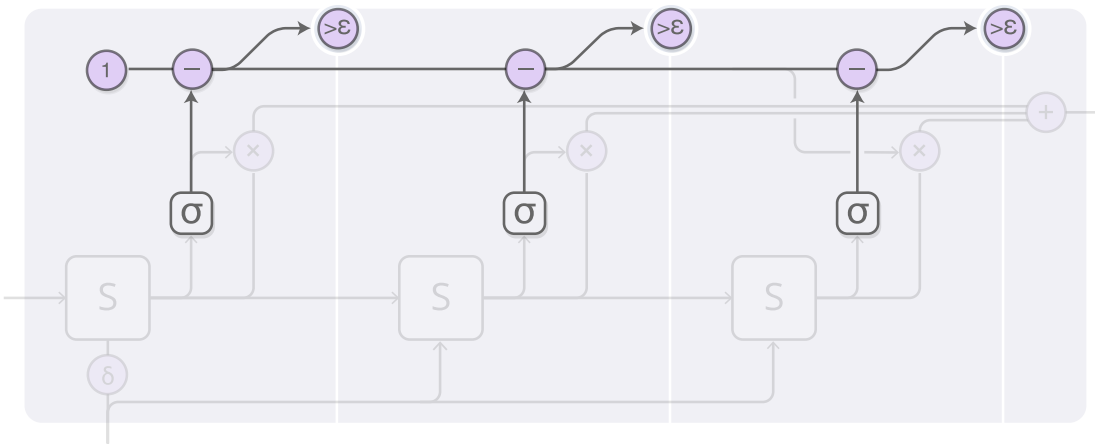
That’s a bit complicated, so let’s work through it step by step. At a high-level, we’re still running the RNN and outputting a weighted combination of the states:



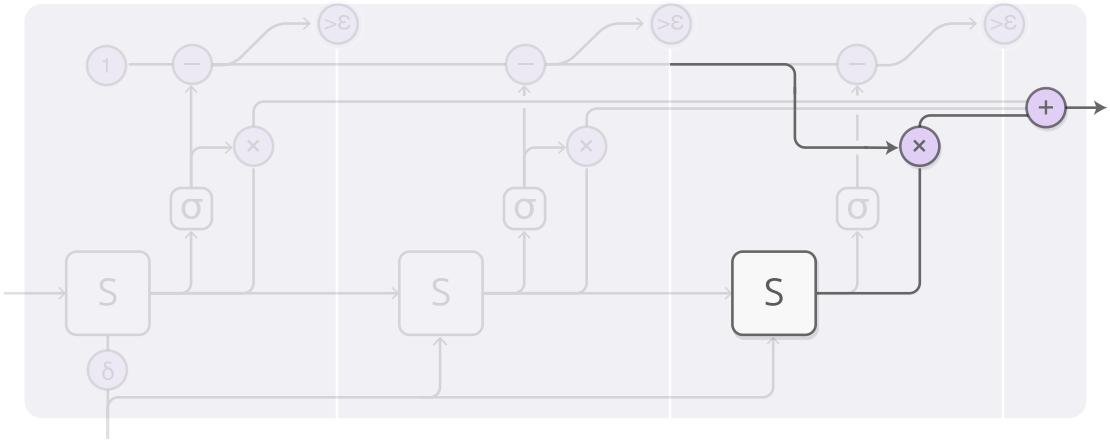
The weight for each step is determined by a “halting neuron.” It’s a sigmoid neuron that looks at the RNN state and gives a halting weight, which we can think of as the probability that we should stop at that step.



We have a total budget for the halting weights of 1, so we track that budget along the top. When it gets to less than epsilon, we stop.



When we stop, might have some left over halting budget because we stop when it gets to less than epsilon. What should we do with it? Technically, it's being given to future steps but we don't want to compute those, so we attribute it to the last step.



When training Adaptive Computation Time models, one adds a "ponder cost" term to the cost function. This penalizes the model for the amount of computation it uses. The bigger you make this term, the more it will trade-off performance for lowering compute time.

Adaptive Computation Time is a very new idea, but we believe that it, along with similar ideas, will be very important.

Code

The only open source implementation of Adaptive Computation Time at the moment seems to be Mark Neumann's (TensorFlow).

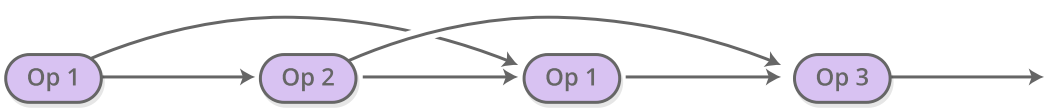
Neural Programmer

Neural nets are excellent at many tasks, but they also struggle to do some basic things like arithmetic, which are trivial in normal approaches to computing. It would be really nice to have a way to fuse neural nets with normal programming, and get the best of both worlds.

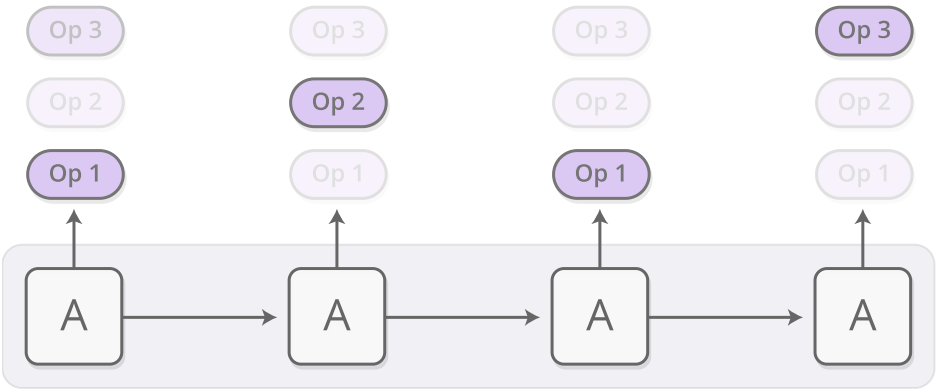
The neural programmer [16] is one approach to this. It learns to create programs in order to solve a task. In fact, it learns to generate such programs *without needing examples of correct programs*. It discovers how to produce programs as a means to the end of accomplishing some task.

The actual model in the paper answers questions about tables by generating SQL-like programs to query the table. However, there are a number of details here that make it a bit complicated, so let's start by imagining a slightly simpler model, which is given an arithmetic expression and generates a program to evaluate it.

The generated program is a sequence of operations. Each operation is defined to operate on the output of past operations. So an operation might be something like "add the output of the operation 2 steps ago and the output of the operation 1 step ago." It's more like a Unix pipe than a program with variables being assigned to and read from.

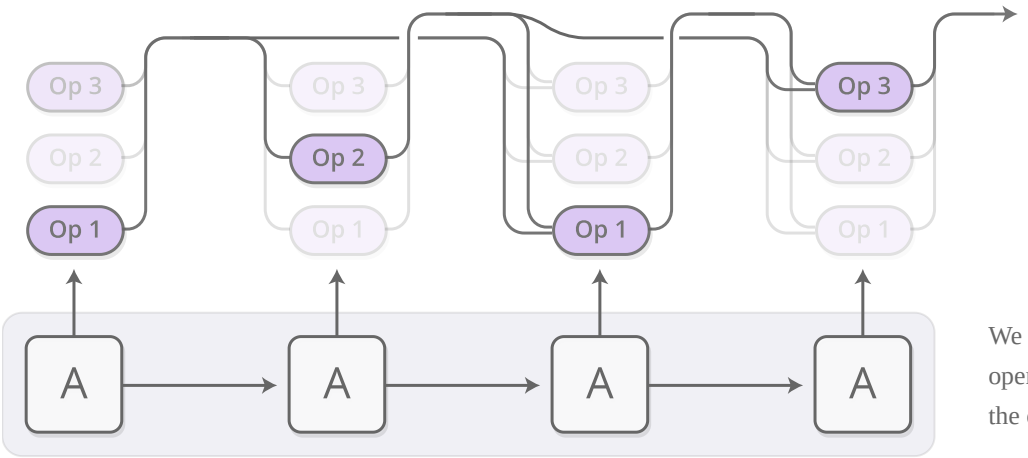


The program is generated one operation at a time by a controller RNN. At each step, the controller RNN outputs a probability distribution for what the next operation should be. For example, we might be pretty sure we want to perform addition at the first time step, then have a hard time deciding whether we should multiply or divide at the second step, and so on...



At each step the controller RNN outputs a probability distribution.

The resulting distribution over operations can now be evaluated. Instead of running a single operation at each step, we do the usual attention trick of running all of them and then average the outputs together, weighted by the probability we ran that operation.



We run all of the operations and average the outputs together.

As long as we can define derivatives through the operations, the program's output is differentiable with respect to the probabilities. We can then define a loss, and train the neural net to produce programs that give the correct answer. In this way, the Neural Programmer learns to produce programs without examples of good programs. The only supervision is the answer the program should produce.

That's the core idea of Neural Programmer, but the version in the paper answers questions about tables, rather than arithmetic expressions. There are a few additional neat tricks:

- **Multiple Types:** Many of the operations in the Neural Programmer deal with types other than scalar numbers. Some operations output selections of table columns or selections of cells. Only outputs of the same type get merged together.

- **Referencing Inputs:** The neural programmer needs to answer questions like "How many cities have a population greater than 1,000,000?" given a table of cities with a population column. To facilitate this, some operations allow the network to reference constants in the question they're answering, or the names of columns. This referencing happens by attention, in the style of pointer networks [17].

The Neural Programmer isn't the only approach to having neural networks generate programs. Another lovely approach is the Neural Programmer-Interpreter [18] which can accomplish a number of very interesting tasks, but requires supervision in the form of correct programs.

We think that this general space, of bridging the gap between more traditional programming and neural networks is extremely important. While the Neural Programmer is clearly not the final solution, we think there are a lot of important lessons to be learned from it.

Code

The more recent version of Neural Programmer for question answering has been open sourced by its authors and is available as a TensorFlow Model. There is also an implementation of the Neural Programmer-Interpreter by Ken Morishita (Keras).

The Big Picture

A human with a piece of paper is, in some sense, much smarter than a human without. A human with mathematical notation can solve problems they otherwise couldn't. Access to computers makes us capable of incredible feats that would otherwise be far beyond us.

In general, it seems like a lot of interesting forms of intelligence are an interaction between the creative heuristic intuition of humans and some more crisp and careful media, like language or equations. Sometimes, the medium is something that physically exists, and stores information for us, prevents us from making mistakes, or does computational heavy lifting. In other cases, the medium is a model in our head that we manipulate. Either way, it seems deeply fundamental to intelligence.

Recent results in machine learning have started to have this flavor, combining the intuition of neural networks with something else. One approach is what one might call "heuristic search." For example, AlphaGo [19] has a model of how Go works and explores how the game could play out guided by neural network intuition. Similarly, DeepMath [20] uses neural networks as intuition for manipulating mathematical expressions. The "augmented RNNs" we've talked about in this article are another approach, where we connect RNNs to engineered media, in order to extend their general capabilities.

Interacting with media naturally involves making a sequence of taking an action, observing, and taking more actions. This creates a major challenge: how do we learn which actions to take? That sounds like a reinforcement learning problem and we could certainly take that approach. But the reinforcement learning literature is really attacking the hardest version of this problem, and its solutions are hard to use. The wonderful thing about attention is that it gives us an easier way out of this problem by partially taking all actions to varying extents. This works because we can design media—like the NTM memory—to allow fractional actions and to be differentiable. Reinforcement learning has us take a single path, and try to learn from that. Attention takes every direction at a fork, and then merges the paths back together.

A major weaknesses of attention is that we have to take every "action" every step. This causes the computational cost to grow linearly as you do things like increase the amount of memory in a Neural Turing Machine. One thing you could imagine doing is having your attention be sparse, so that you only have to touch some memories. However, it's still challenging because you may want to do things like have your attention depend on the content of the memory, and doing that naively forces you to look at each memory. We've seen some initial attempts to attack this problem, such as [21], but it seems like there's a lot more to be done. If we could really make such sub-linear time attention work, that would be very powerful!

Augmented recurrent neural networks, and the underlying technique of attention, are incredibly exciting. We look forward to seeing what happens next!

Acknowledgments

Thank you to Maithra Raghu, Dario Amodei, Cassandra Xia, Luke Vilnis, Anna Goldie, Jesse Engel, Dan Mané, Natasha Jaques, Emma Pierson and Ian Goodfellow for their feedback and encouragement. We're also very grateful to our team, Google Brain, for being extremely supportive of our project.

References

1. **Understanding LSTM Networks** [link]
   Olah, C., 2015.
2. **Neural Turing Machines** [PDF]
   Graves, A., Wayne, G. and Danihelka, I., 2014. CoRR, Vol abs/1410.5401.
3. **Show, attend and tell: Neural image caption generation with visual attention**
   Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhutdinov, R., Zemel, R.S. and Bengio, Y., 2015. arXiv preprint arXiv:1502.03044, Vol 2(3), pp. 5. CoRR.
4. **Neural GPUs Learn Algorithms** [PDF]
   Kaiser, L. and Sutskever, I., 2015. CoRR, Vol abs/1511.08228.
5. **Reinforcement Learning Neural Turing Machines** [PDF]
   Zaremba, W. and Sutskever, I., 2015. CoRR, Vol abs/1505.00521.
6. **Neural Random-Access Machines** [PDF]
   Kurach, K., Andrychowicz, M. and Sutskever, I., 2015. CoRR, Vol abs/1511.06392.
7. **Learning to Transduce with Unbounded Memory** [PDF]
   Grefenstette, E., Hermann, K.M., Suleyman, M. and Blunsom, P., 2015. Advances in Neural Information Processing Systems 28, pp. 1828—1836. Curran Associates, Inc.
8. **Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets** [PDF]
   Joulin, A. and Mikolov, T., 2015. Advances in Neural Information Processing Systems 28, pp. 190—198. Curran Associates, Inc.
9. **Memory Networks** [PDF]
   Weston, J., Chopra, S. and Bordes, A., 2014. CoRR, Vol abs/1410.3916.
10. **Ask Me Anything: Dynamic Memory Networks for Natural Language Processing** [PDF]
    Kumar, A., Irsoy, O., Su, J., Bradbury, J., English, R., Pierce, B., Ondruska, P., Gulrajani, I. and Socher, R., 2015. CoRR, Vol abs/1506.07285.

11. **Neural machine translation by jointly learning to align and translate**
Bahdanau, D., Cho, K. and Bengio, Y., 2014. arXiv preprint arXiv:1409.0473.
12. **Listen, Attend and Spell** [PDF]
Chan, W., Jaitly, N., Le, Q.V. and Vinyals, O., 2015. CoRR, Vol abs/1508.01211.
13. **Grammar as a foreign language**
Vinyals, O., Kaiser, L., Koo, T., Petrov, S., Sutskever, I. and Hinton, G., 2015. Advances in Neural Information Processing Systems, pp. 2773—2781.
14. **A Neural Conversational Model** [PDF]
Vinyals, O. and Le, Q.V., 2015. CoRR, Vol abs/1506.05869.
15. **Adaptive Computation Time for Recurrent Neural Networks** [PDF]
Graves, A., 2016. CoRR, Vol abs/1603.08983.
16. **Neural Programmer: Inducing Latent Programs with Gradient Descent** [PDF]
Neelakantan, A., Le, Q.V. and Sutskever, I., 2015. CoRR, Vol abs/1511.04834.
17. **Pointer networks**
Vinyals, O., Fortunato, M. and Jaitly, N., 2015. Advances in Neural Information Processing Systems, pp. 2692—2700.
18. **Neural Programmer-Interpreters** [PDF]
Reed, S.E. and Freitas, N.d., 2015. CoRR, Vol abs/1511.06279.
19. **Mastering the game of Go with deep neural networks and tree search** [link]
Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Driessche, G.v.d., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. and Hassabis, D., 2016. Nature, Vol 529(7587), pp. 484—489. Nature Publishing Group. DOI: 10.1038/nature16961
20. **DeepMath - Deep Sequence Models for Premise Selection** [PDF]
Alemi, A.A., Chollet, F., Irving, G., Szegedy, C. and Urban, J., 2016. CoRR, Vol abs/1606.04442.
21. **Learning Efficient Algorithms with Hierarchical Attentive Memory** [PDF]
Andrychowicz, M. and Kurach, K., 2016. CoRR, Vol abs/1602.03218.

Updates and Corrections

View all changes to this article since it was first published. If you see a mistake or want to suggest a change, please create an issue on GitHub.

Citations and Reuse

For attribution in academic contexts, please cite this work as

```
Olah & Carter, "Attention and Augmented Recurrent Neural Networks", Distill, 2016. http://doi.o
```