Attention Networks

Jerry Ding, Ryan Brigden

Tasks that use attention

How do you do any of the following tasks?

Text → Translation of that text

Image → A caption describing the image

Document, question → Answer that uses the document to answer the question.

Tasks suited for recurrent networks

- “Many to one” architecture
  - When there’s a single classification to be made at the end of sequential input
- “Many to many (streaming)” architecture
  - Each output label is associated with a small segment of the input sequence
  - Outputs generated “on the fly” even without one-to-one correspondence from input to label
- “Generative” architecture
  - The output is itself a sequence, generated from the input sequence
  - Prior belief that the output sequence can only be built after seeing the entire input sequence

Said more simply

Each item in the output sequence must be conditioned on:

- All the past states of the network
- The entire input sequence

In deep learning,

“The network is conditioned on X” ≈

“X, or some encoding of X, is one of the inputs to the neural network”

Being conditioned on something

The outputs of “one-to-one” architectures only depend on the input.

- Without adding noise, the same input gives the same output.

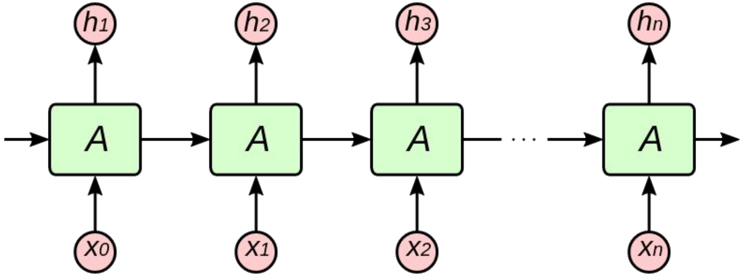
The output of an RNN also depends on all its past states.

- The history of the network is encoded in the hidden vector. Both the input vector and the hidden state vector are used to compute the output and the next hidden state vector.

Later you may learn about Variational Autoencoders, whose behavior can be conditioned on a random number (passed as one of the inputs)

Encoding of the input sequence

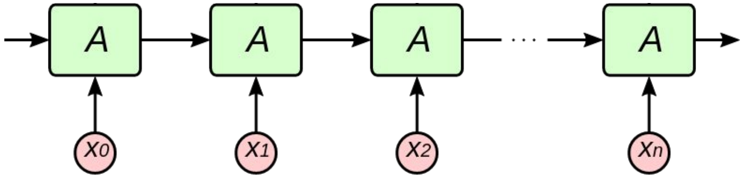
The typical RNN structure uses the hidden state to store meaningful information about the history of both inputs and outputs.



The network is trained to produce the correct output sequence. In doing so, it learns how to best store past information with the hidden state.

Encoding of the input sequence

If we removed the output segment of the RNN, then the hidden state only encodes information about the history of inputs. The last hidden state is an encoding of the entire input sequence!



Note that we have no way to train or use this architecture in isolation.

The network can be trained such that the final hidden state vector is maximally "useful" to a downstream network, which uses that vector to produce outputs.

Using the sequence encoding

In a generative RNN architecture, the output network is also recurrent.

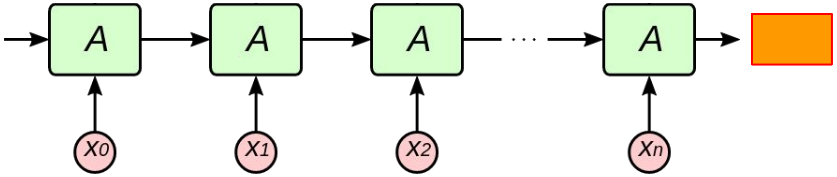
There are many ways to pass in the input sequence. Examples:

1. Pass in an encoding of the input sequence at every time step.
2. Pass an encoding of the input sequence at the first time step, and some placeholder vector (like all zeros) for the remaining time steps.
3. Initialize the hidden vector of the output RNN with an encoded input sequence
4. Pass an encoded input sequence through a feed-forward network, and use the output of that to initialize the hidden vector of the output RNN

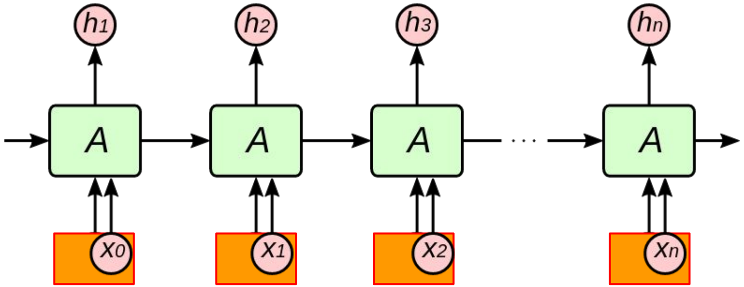
Let’s use the first approach for now.

Network Prototype 1

Produce an encoding of the entire input.



Repeatedly pass the encoding to the output network.



Problems?

We’re using one vector to encode an entire sequence! We’re making a very compressed representation of the input.

That’s fine if we knew exactly what aspects of the input sequence were important (so the vector stores only the important bits and throws out the rest).

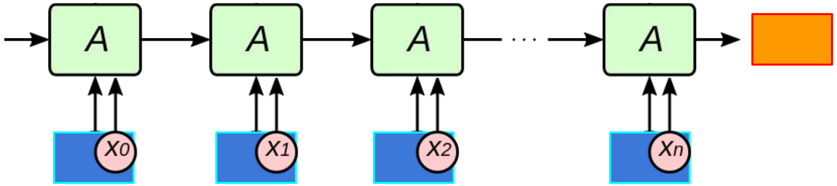
But we’re using the same encoding for all output time steps. Odds are, all parts of the input sequence will be useful at some point.

> E.g. You can’t make a good translation of a sentence if you can’t see some of its words.

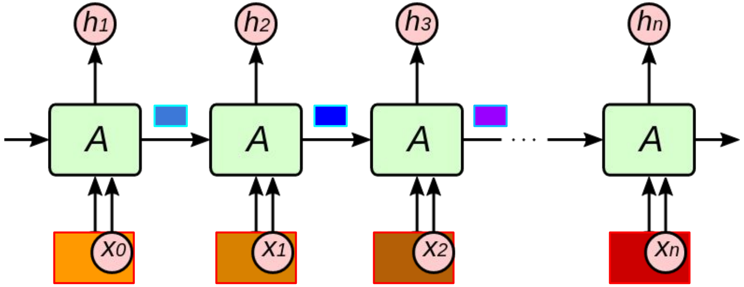
The encoder can’t simply focus on a specific part of the input in this architecture. It must try to describe the entire sequence.

Network Prototype 2

Encode the entire input, conditioned on the hidden state of the output network.



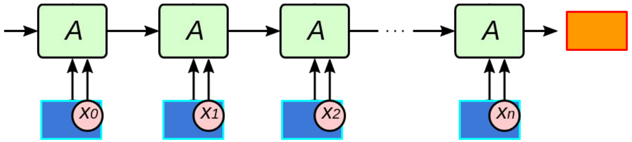
Pass that encoding to the output network to produce the next hidden state & an output. Repeat the above procedure for all time steps.

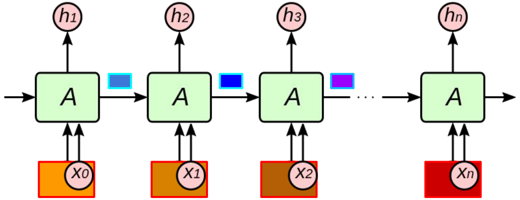


What this approach gives us

The encoding of the input sequence can be different for each time step. The encoding is still lossy, but only needs to contain enough information for a single time step of the output network.

E.g. If you're translating a sentence and are about to write a noun, you only need to collect information about the nouns in the input to figure out what word to output.

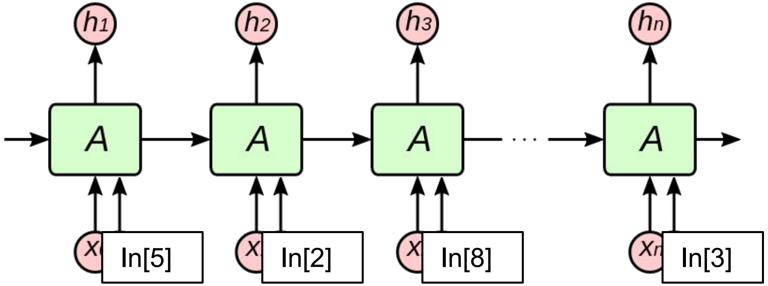




But the network is now really complicated & hairy. We need to pass through the input sequence for every item in the output sequence, which can also get very computationally expensive.

Going simpler

We don’t need complete information about the full input sequence to generate each item of the output sequence. But is it enough if we just passed *a single element of the input* at each time step of the output?



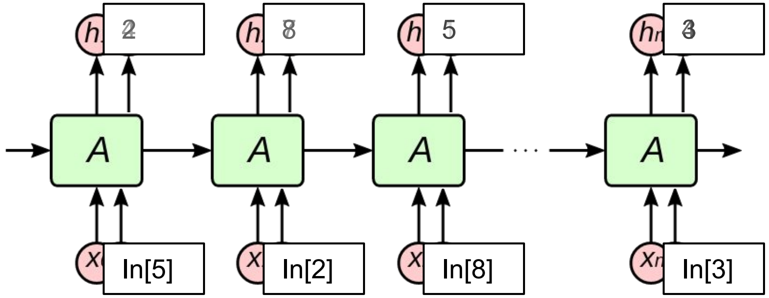
Let’s suppose that this actually works. How do we figure out which input item to choose at each time step?

Network Prototype 3

Assign a “key” to each element of the input sequence.

At each time step, generate a “query”. Then pick the input element whose key matches most closely to the query. Pass this as an input to the output network.

1	2	3	4	...
In[1]	In[2]	In[3]	In[4]	



For example, similarity can be computed with the dot product.

Hard Attention

Prototype 3 is a hard attention mechanism. Instead of computing some kind of encoding, we ***compute an attention*** from the query to the sequence of inputs, and choose an actual item in the input sequence to pass in.

Here we just picked the item with the most similar key. It's also common to use Gumbal noise or random sampling to choose the input item probabilistically.

Pros: We don't need to train an encoder. The vectors that we pass into the output RNN are already known to be meaningful.

Cons: Not differentiable! Also, a single input item might not contain enough information even for a single time step (e.g. one image pixel)

Soft Attention

We’ve already computed the attention from the query to the keys of the input sequence. Instead of taking the most similar element, why not be non-committal and make a “soft” selection?

To be specific, we compute the softmax of the similarity scores to get a sequence of ***attention values***. The attention values will form a probability distribution, a.k.a. the values are positive and sum to 1.

We then find the dot product of the attention values and the input sequence. The result is a (convex) weighted average of the input elements.

Soft Attention

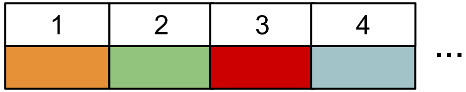
We need to be careful when using soft attention. A weighted average of input items might be nonsensical:

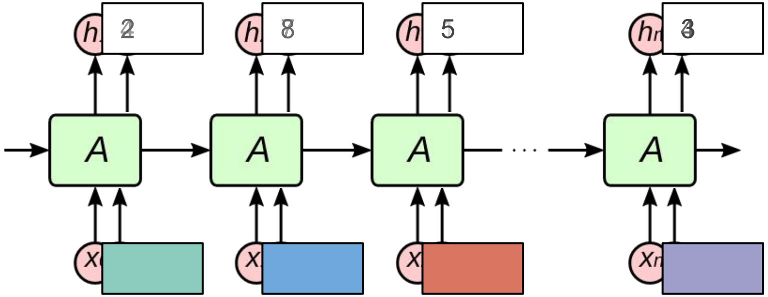
We need to produce an encoding of each individual item. This encoding is designed / learned such that a weighted combination is still meaningful.

Network Prototype 4

Encode each element of the input sequence.

For each time step, compute an attention on this sequence. Fuse the input items using the computed attention values. Pass this value to the output RNN.





Simplifications

The query can just be the hidden state of the output RNN.
The keys can just be the encoded input items themselves.

This works because you can store a large amount of information in a vector without affecting its behavior under the similarity metric very much.

```
In [47]: x
Out[47]:
array([0.00056069, 0.00025621, 0.00094021, 0.00034668, 0.00092293,
       0.00072075, 0.00077331, 0.00087205, 0.00053207, 0.00054651,
       0.00005004, 0.00019434, 0.0001579 , 0.00024429, 0.00083791,
       0.00059451, 0.00034378, 1.        , 0.        , 0.        ])

In [48]: y
Out[48]:
array([0.00007009, 0.00013716, 0.00097105, 0.00004687, 0.00077888,
       0.00019543, 0.00025575, 0.00033477, 0.00000329, 0.00086016,
       0.00030442, 0.00039481, 0.0000476 , 0.00088459, 0.00026804,
       0.00083782, 0.00037836, 1.        , 0.        , 0.        ])

In [49]: x.dot(y)
Out[49]: 1.0000039932546048
```

Mathematical representation

We have *n* inputs of *k*-dimensional items. Our hidden states are *d*-dimensional.

V: Input sequence (*n* x *k* matrix)

E: Encoded input sequence (*n* x *d* matrix)

$\mathbf{h}_t$: Output RNN hidden state at time *t* (*d*-vector)

$\mathbf{a}_t$: Attention values at output *t* (*n*-vector)

$$\mathbf{E} = f_{encoder}(\mathbf{V})$$

$$\mathbf{a}_t = \text{softmax}(\mathbf{E}\mathbf{h}_t)$$

$$\mathbf{h}_{t+1} = f_{output}(\mathbf{E}^T\mathbf{a}_t, \mathbf{h}_t)$$

All you need is attention

You can compute an attention from one input element to all the other elements in the input sequence → Encoder can be implemented using attention.

You can compute an attention from the hidden state at one time step to the hidden state at all past time steps → Decoder can be implemented using attention.

Of course, you can use attention to condition the decoder on the encoded inputs.

∴ You can make a network using nothing but attention.

https://arxiv.org/abs/1706.03762

Hard Attention Revisited

Soft attention is not human attention. Human perception does not process an entire scene and *then* attend to what is important for a given task.

We focus our attention selectively and acquire information that may help direct our attention further and eventually solve the task at hand.

- Reduce computational complexity (CNNs scale linearly with # of pixels)
- Significantly reduce the complexity of the task by only focusing on what is useful (in theory)

Hard Attention Revisited

We stated before that hard attention is desirable because we can directly access the most useful piece of information at a given time step (in theory)

We threw our hands up because it was non-differentiable.

Can we formulate hard attention as a sequential decision problem?

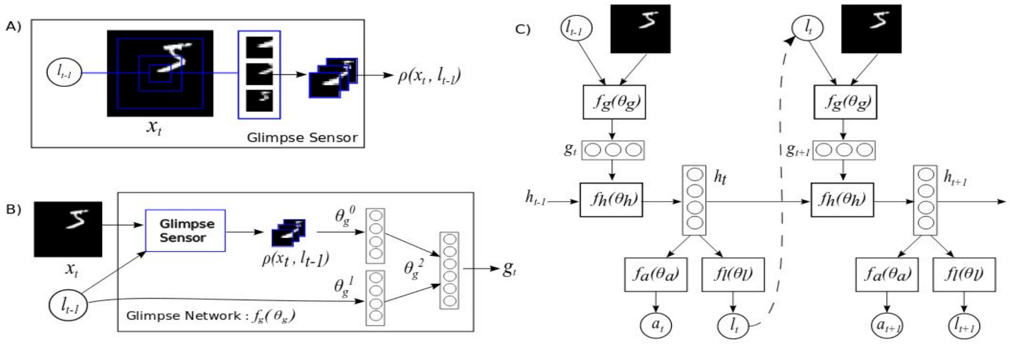
Can we introduce a new method that attends based on past information *and* the direct demands of the task?

Reinforcement Learning for Visual Attention

Formulate the problem as a sequential decision problem.

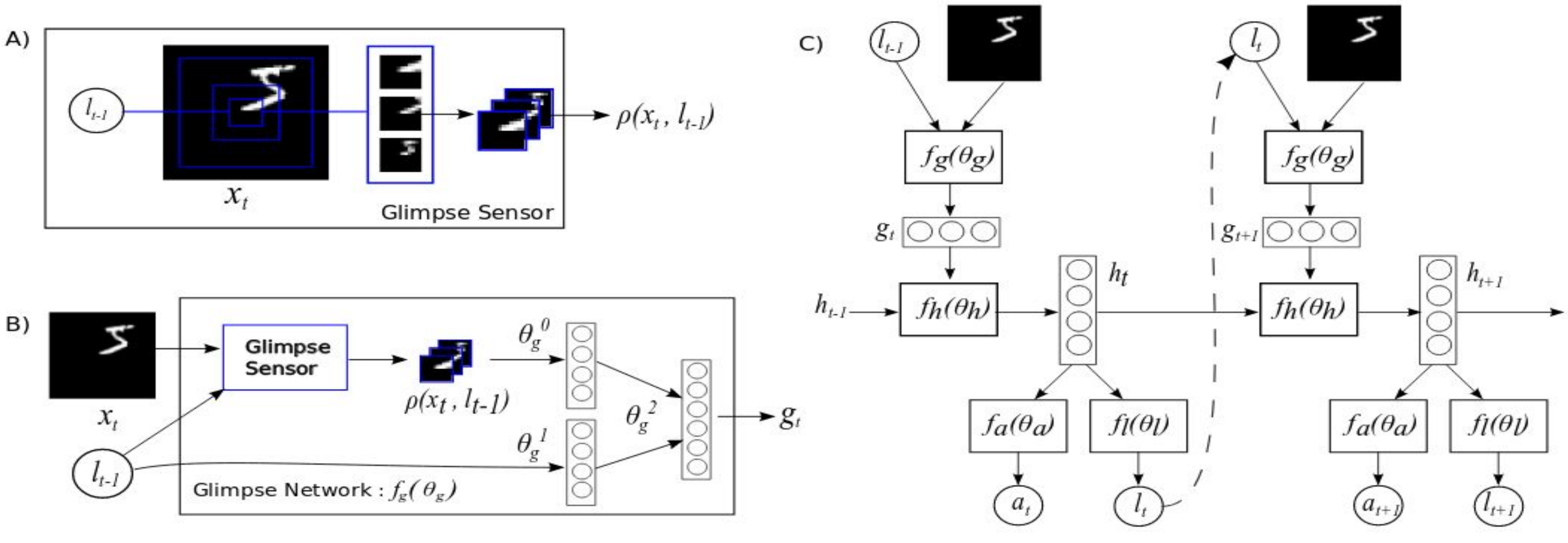
Use (variance reduced) REINFORCE algorithm to learn policy $f_l(\text{theta}_l)$, which chooses the next location for the “Glimpse Sensor.”

Reward function can be swapped out for different tasks, making this a flexible approach



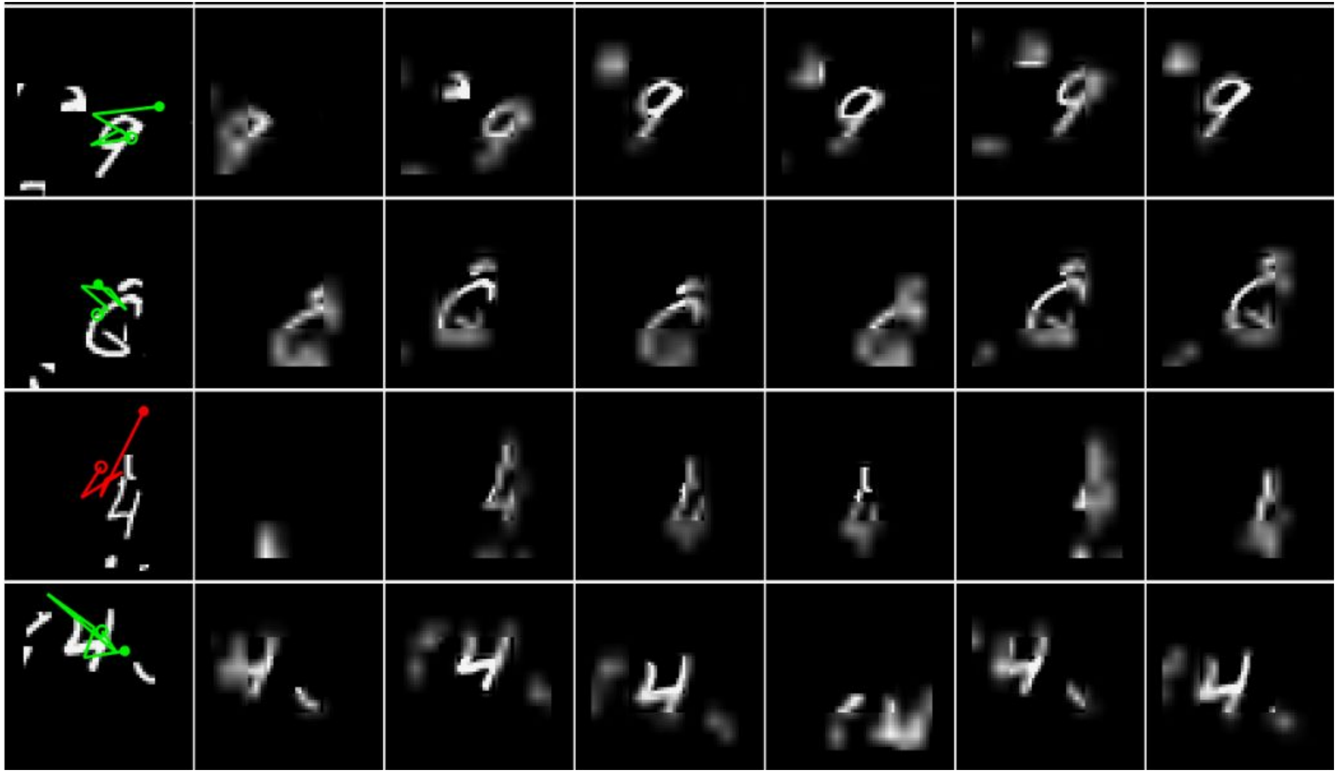
Recurrent Models for Visual Attention https://arxiv.org/abs/1406.6247

Reinforcement Learning for Visual Attention



Recurrent Models for Visual Attention https://arxiv.org/abs/1406.6247

Reinforcement Learning for Visual Attention



Recurrent Models for Visual Attention https://arxiv.org/abs/1406.6247

Jupyter Notebook

Here’s a simple example of an attention network