The Startup · Follow publication

A simple overview of RNN, LSTM and Attention Mechanism

13 min read · Jan 30, 2021

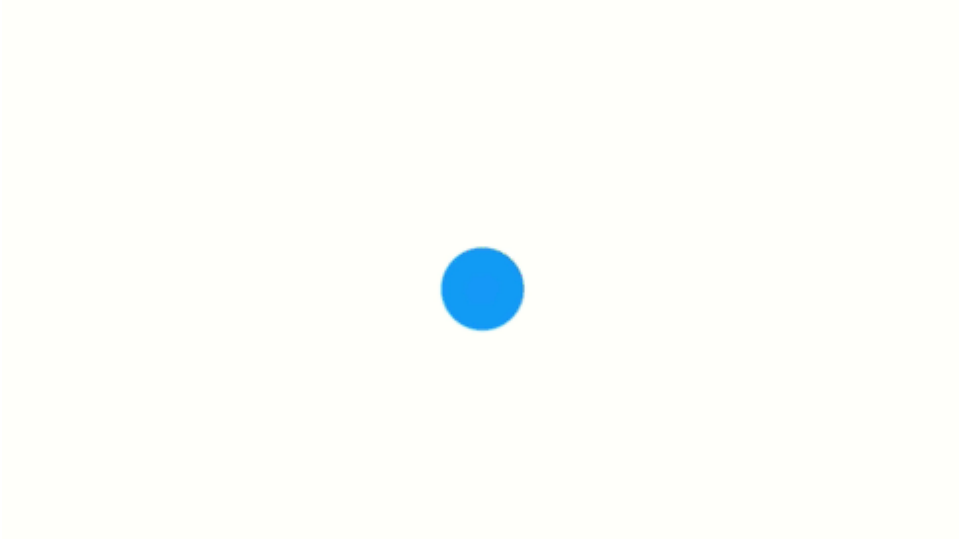
Manu Chauhan Follow

Listen Share

Recurrent Neural Networks, Long Short Term Memory and the famous Attention based approach explained

When you delve into the text of a book, you read in the logical order of chapter and pages and for a good reason. The ideas you form, the train of thoughts, it's all dependent on what you have understood and ***retained*** up to a given point in the book. This *persistence* or the ability to have some *memory and pay attention* helps us to develop understanding of concepts and the world around us, allowing us to think, use our knowledge, to write, solve problems as well as innovate.

There is an inherent notion of progress with steps, with passage of time, for sequential data. *Sequential memory is a mechanism that makes it easier for our brain to recognize sequence patterns.*



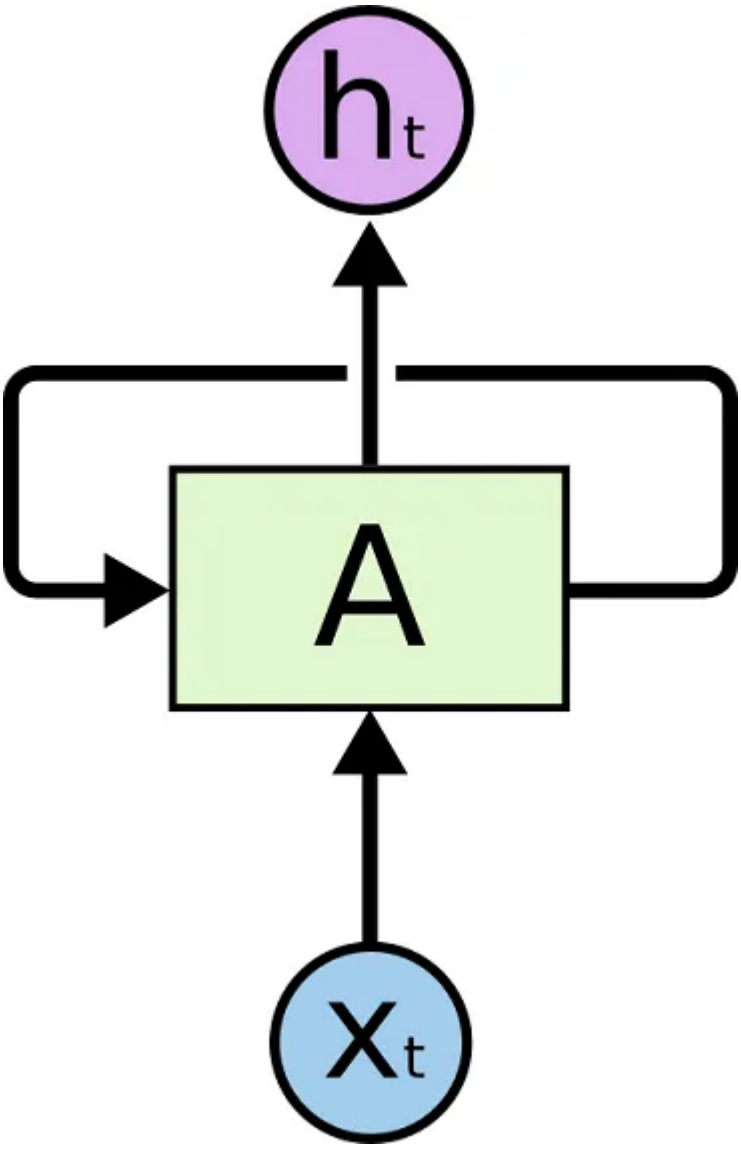
Traditional **Neural Networks**, despite being good at what they do, lack this intuitive innate tendency for *persistence*. How would a simple feed forward Neural Network read a sentence and pass on previously gathered information in order to completely *understand and relate sequence* of incoming data? It cannot (mind it, training NN for weights is not same as persistence of information for next step).

. . .

RNNs to the rescue

> *There's something magical about Recurrent Neural Networks.*
>
> *— Andrej Karpathy*

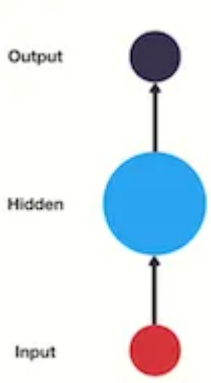
Recurrent Neural Networks address this drawback of vanilla NNs with a simple yet elegant mechanism and are great at modeling sequential data.



rolled RNN

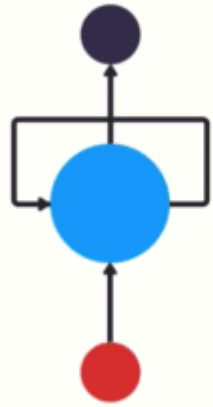
Does RNN look weird to you? Let me explain and remove the confusion.

Take a simple feed forward Neural Network first, shown below. It has the input coming in, red dot, to the hidden layer, blue dot, which results in black dot output.



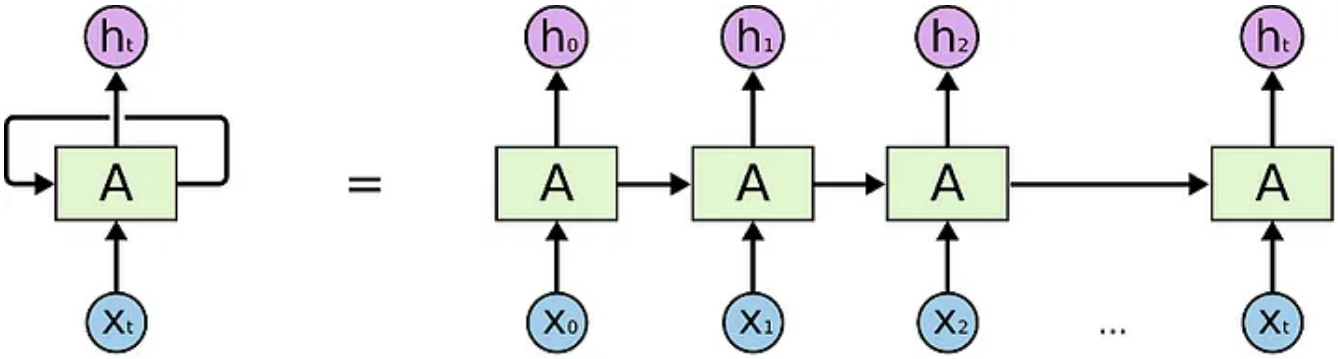
A simple NN

An RNN feeds it's output to itself at next time-step, forming a loop, passing down much needed information.



RNN feeding hidden state value to itself

To better understand the flow, look at the unrolled version below, where each RNN has different input (token in a sequence) and output at each time step.



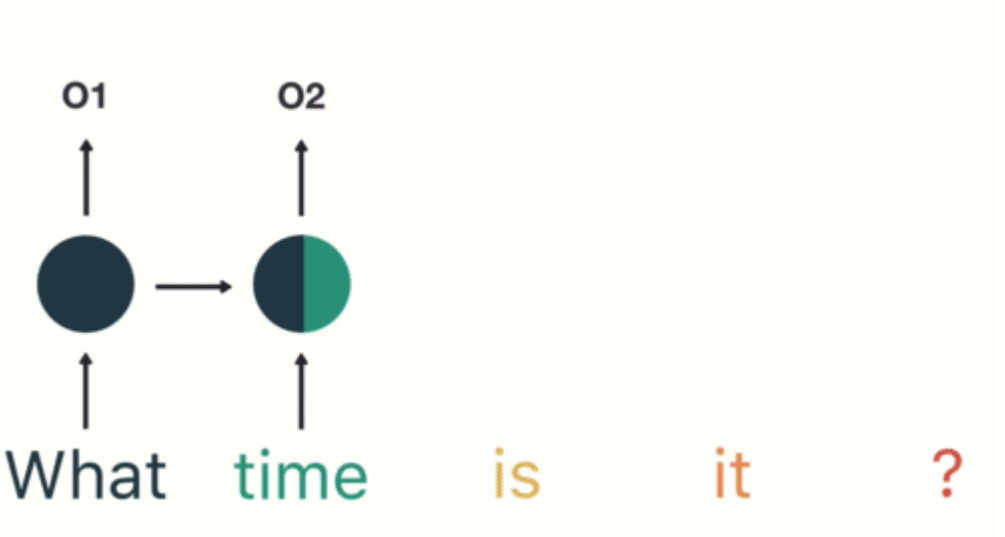
Unrolled RNN, from time step **0** to **t**

The NN ***A*** takes in input at each time step while giving output ***h*** and passing information to itself for next incoming input ***t+1*** step. The incoming sequential data is encoded using RNN first before being utilized to determine the intent/action via another feed forward network for decision.

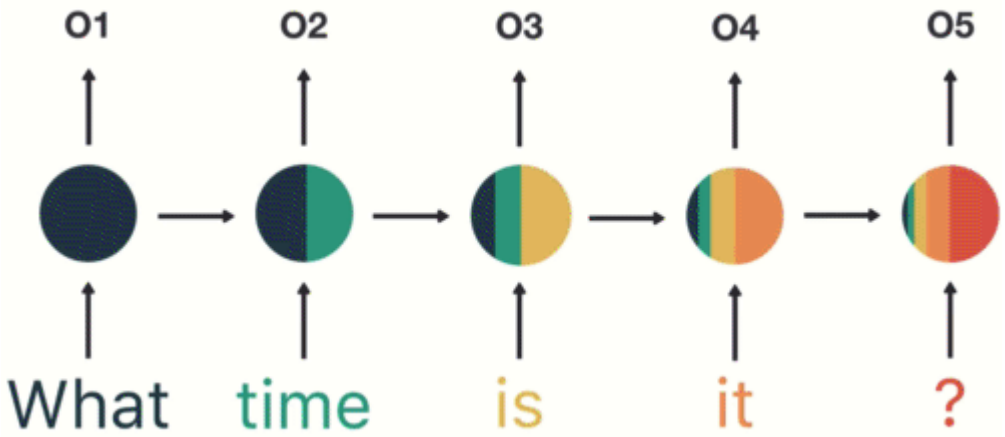
A single RNN unrolling with time steps

RNNs have become the go-to NNs to be used for various tasks involving notion of sequential data, such as: speech recognition, language modeling, translation, image captioning etc.

Let's say we ask a question to your in-house developed AI Assistant named Piri (or whatever *'iri'* you prefer), "what time is it?", here we try to break the sequence and color code it.



How RNNs work for the tokens of sequence



Final query retrieved as a result of processing the **entire** sequence

Memory: An essential requirement for making Neural Networks smart(er)

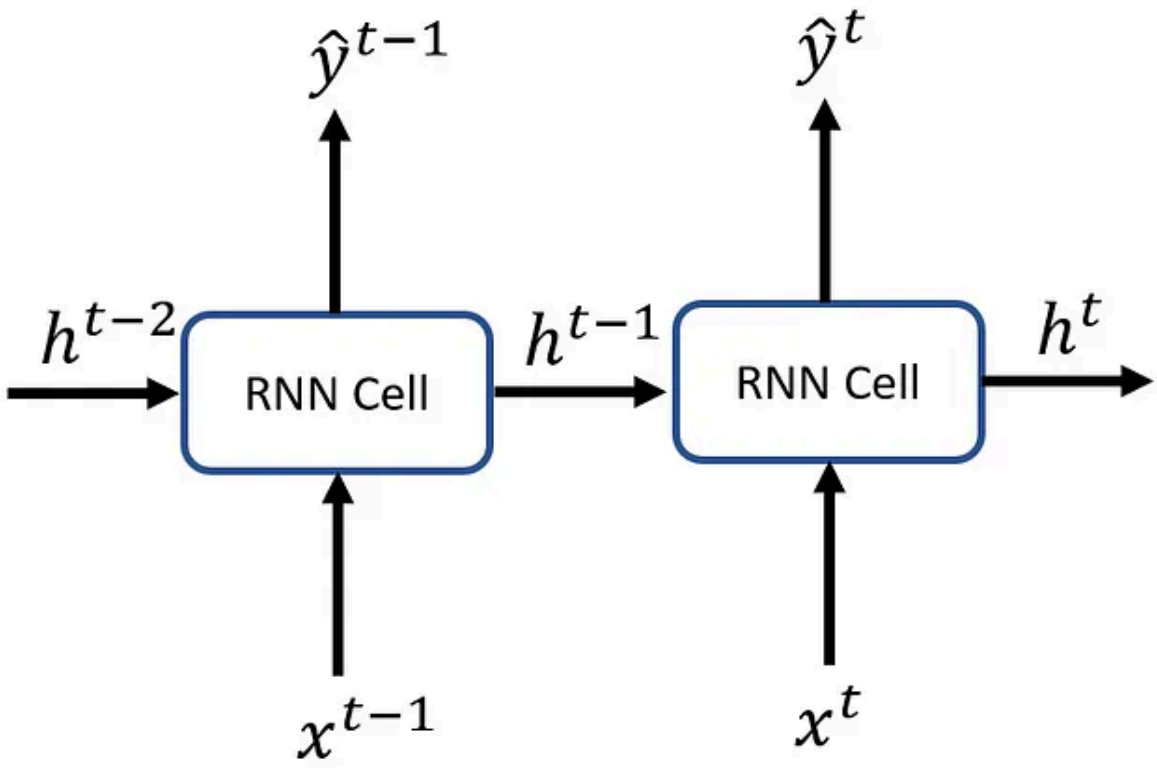
Humans tend to retrieve information from memory, short or long, use current information with it and derive logic to take next action (or impulse/habit, again based on previous experiences).

Similar is the idea to make RNN hold on to previous information or state(s). As the output of a recurrent neuron, at a given time step *t*, is clearly a function of the previous input (or think of it as previous input with accumulated information) till time step *t-1*, one could consider this mechanism as a form of ***memory***. Any part of a neural network that has the notion of preserving state, even partially, across time steps is usually referred to as a ***memory-cell***.

Each recurrent *neuron* has an output as well as a hidden state which is passed to next step *neuron*.

$$h_t = g(h_{t-1}, x_t)$$

hidden state

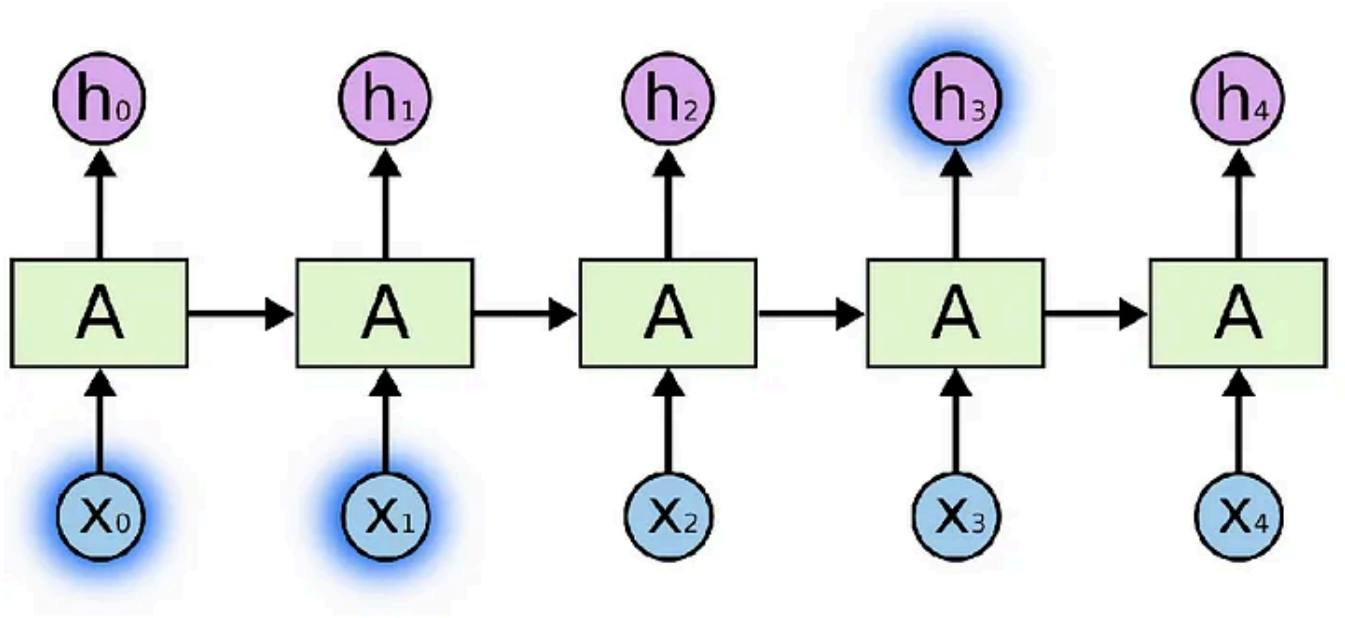


Unrolled RNN with hidden state and output at each time step

. . .

Too good to be true ? Vanishing Gradients and the long dependency

Sometimes the use case requires just immediate information from previous step or only few steps back if not 1. Example: **Clouds are in the *sky*** . If *sky* here is the next word to be predicted using a language model then this becomes an easy task for an RNN as the gap or distance, or better: dependency, between the current step and previous information is small.



Short term dependency: RNNs works like a charm

As you can see the output at *h3* required *X0* and *X1*, which is a ***short-term dependency***, an easy task for simple RNNs. Are RNNs totally infallible then?

What about large sequences and dependency? Theoretically, RNNs are capable of handling such "long-term dependencies."

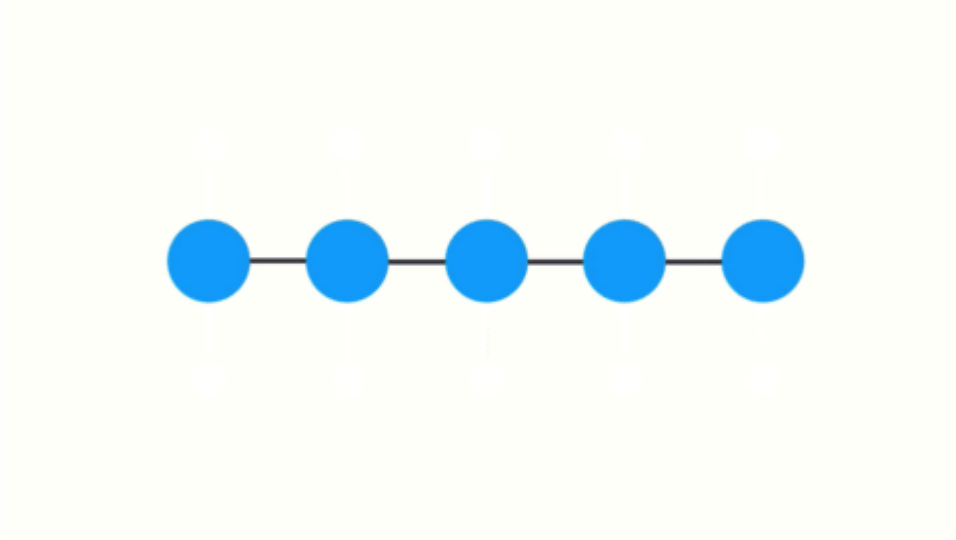
However, in practice, as the dependency gap increases and more of the ***context*** is required, RNNs are unable to learn and would result in defenestration for sure. These issues were analyzed by Hochreiter (1991) and Bengio, et al. (1994).

You can see the changing distribution for colors with each time step, with the recent ones having more area while previous ones reducing further. This beautifully shows the issue with RNNs called as "**short-term memory**".

The age old Vanishing Gradient problem: Major problem that makes training RNNs very slow and inefficient for usage is vanishing gradient issue. The process for a feed forward neural network is as follows: **a)** the forward pass outputs some results **b)** the results are used to compute the loss value **c)** the loss value is used to perform back propagation to calculate the gradients with respect to the weights **d)** these gradients with respect to weights flow backwards to fine tune the weights in order to improve the network's performance.

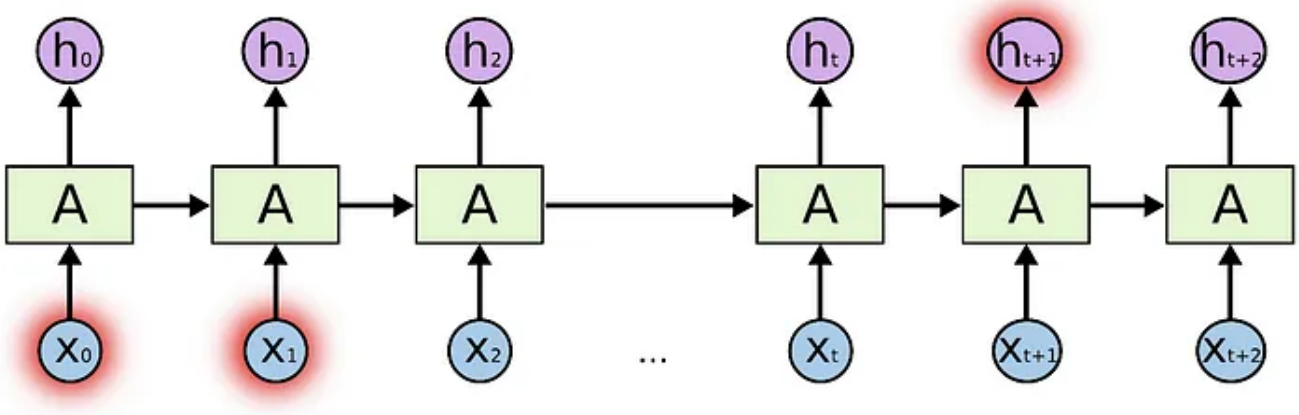
As the manipulation of weights happens according to layer before it, small gradients tends to diminish by large margins after every layer and reach a point where they

are very close to zero hence the learning drops for initial layers and overall effective training slows down.



vanishing gradient problem in NNs during back propagation across layers

Hence, vanishing gradients causes RNN to **NOT** learn the long-range dependencies well across time steps. This means the earlier tokens of a sequence will not be having high importance even if they are crucial to the entire context. Therefore, this inability to learn on long sequences results in short-term memory.(Check out [5] for more on Vanishing Gradient issue)



Long sequence for RNN raising long-term dependency issue, making RNNs struggle to retain all the needed information

This long-term dependency *thing* seems to encumber the usage of RNNs for real world application with decent size sequences. Does that mean RNNs are ostracized forever?

Not quite, as it turns out, a variation of RNNs or let's say a special RNN comes handy for solving the long-term dependency issues encountered with simple RNNs.

. . .

All hail the eternal hero: LSTM 😎

Long **S**hort **T**erm **M**emory (LSTM) helps for the "long-term dependency" requirement (or solves short-term memory issue) as the default behavior is to remember long term information.

From RNN to LSTM

Let's say that RNN can remember at least the last step in a sequence of words.

> *Our data set of sentences:*
>
> *Dog scares Cat.*
> *Cat scares Mouse.*
> *Mouse scares Dog.*

When starting out, the RNN's input will be empty token for beginning of each sentence (for words Dog, Cat and Mouse).

RNN model sees each of those words followed by ***scares*** and a **.**(full stop) after that, for each sentence.

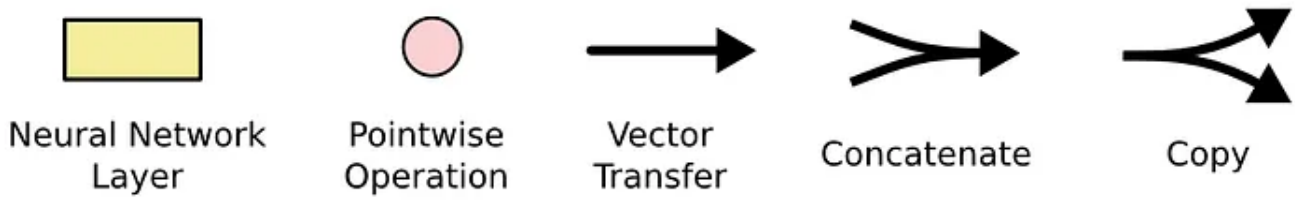
If we try to predict word in sentences, the predictions could take even these form due to ambiguity:

> ***<Empty> Dog scares Dog.***
>
> ***<Empty> Mouse scares Cat.***
>
> ***<Empty> Mouse scares Cat scares Dog.***

Which DOES NOT make any sense as the previous context does not play any role in deciding the next suitable word.

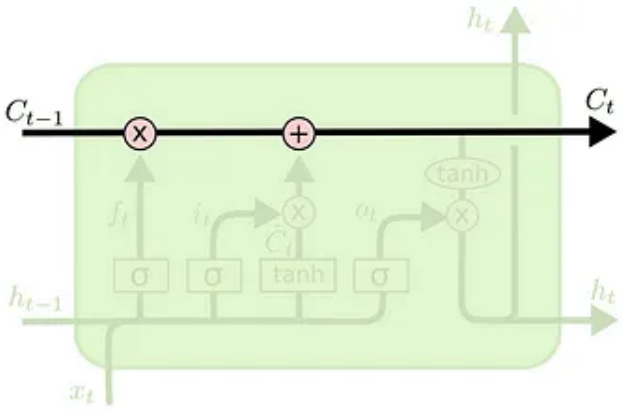
The fundamental LSTM ideas:

First things first: the notations!



Notations used to explain LSTM

The primary component that makes LSTMs rock is the presence of a cell state/vector for each LSTM node, which is passed down to every node in the chain.

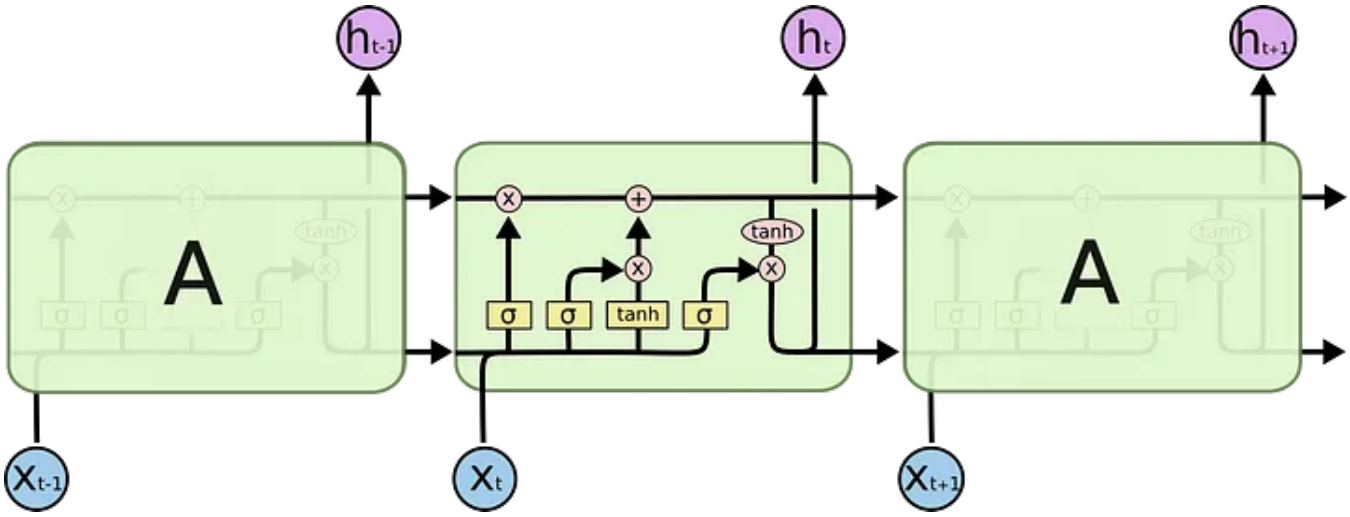


Cell State/Vector running on top of a LSTM cell

This cell state could be modified, if required, with linear mathematical interactions in every node depending upon the learned behavior, regulated by other inner components. In simple words, these inner components, known as gates with activation functions, can add or remove information to the cell state, thus helping in moving forward relevant (sometimes modified) information.

A simple RNN has a simple NN in itself acting as a sole gate for some data manipulations, LSTM, however, has a more intricate inner structure of 4 gates.

NOTE: The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates, making them inherently better than simple RNNs.



LSTM chain of 3 nodes and internal structure depiction

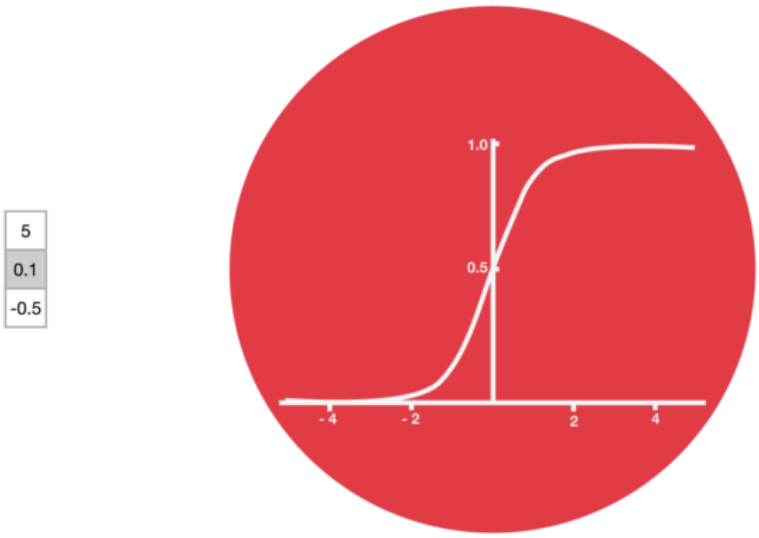
The gates allow information to be optionally altered and let through. For this Sigmoid and Tanh gates are used internally.

. . .

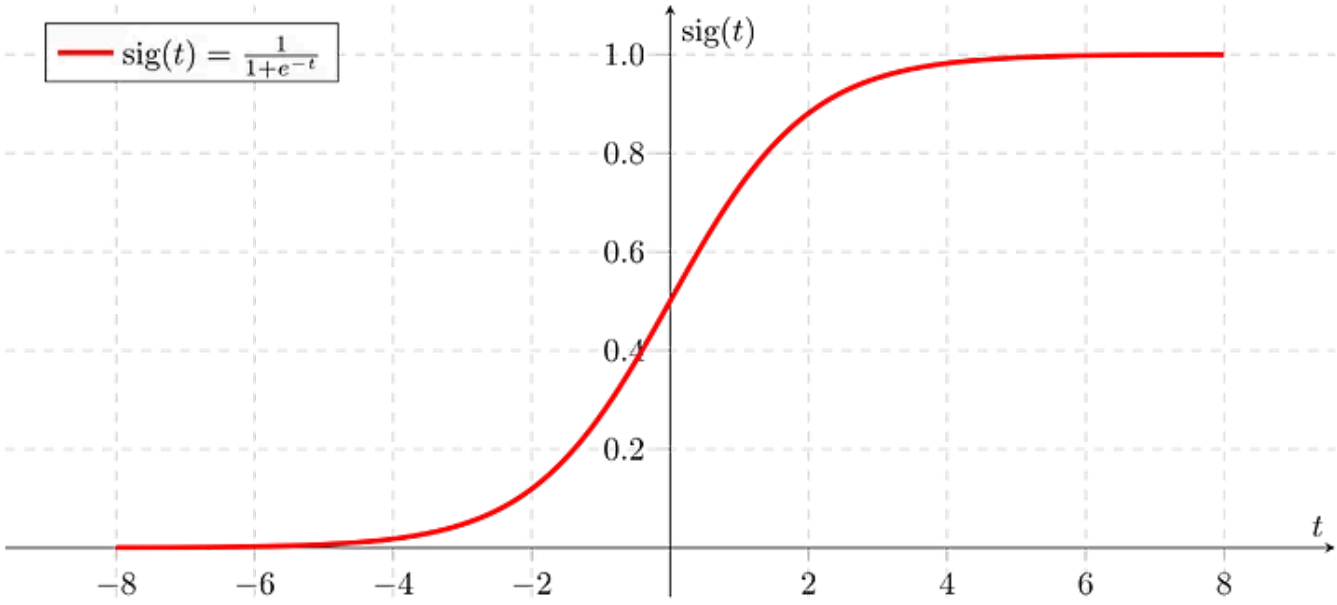
Before we go through step-by-step working of LSTM cell, let's take a look at what Sigmoid and Tanh activation functions are:

Sigmoid activation:

The sigmoid helps to squash the incoming values between 0 and 1 ([0, 1]). This is used in conjugation of other component to stop(if 0) or allow(1) incoming information.

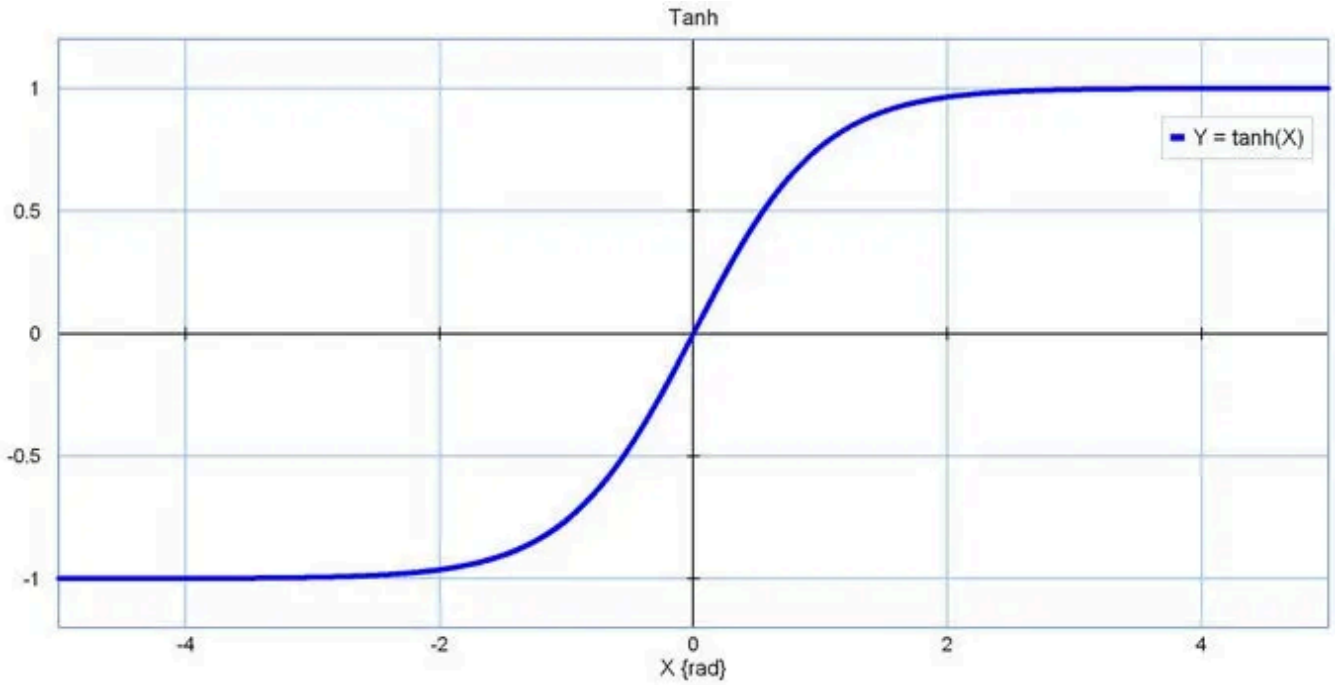


Sigmoid squashing the input values



Tanh activation:

Tanh on other hand helps to compress the values in range **-1 to +1.**



This helps to add or remove data with -1 and +1 respectively.

These activation functions also help to curb the outputs which otherwise would just explode after successive multiplications along the chain.

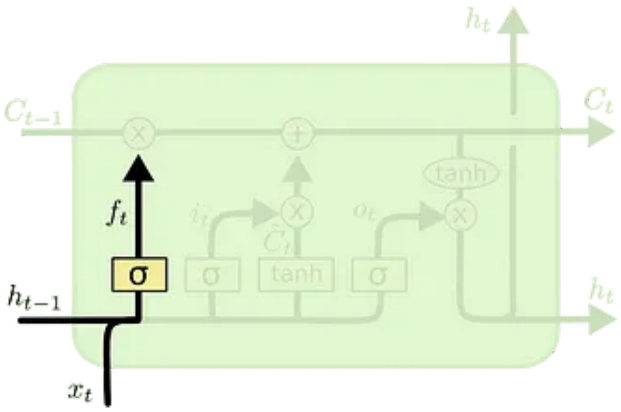
LSTM inner workings 🧐

Step 1: To decide what to *keep* and what to *FORGET*

First step is to decide what all should be forgotten from the cell state. To solve this a Sigmoid is used in ***forget gate layer***. As explained previously a Sigmoid helps to

output values in closed range [0,1]. Here 0 means to 'remove completely' while 1 means to 'keep information' as it is.

The forget gate looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

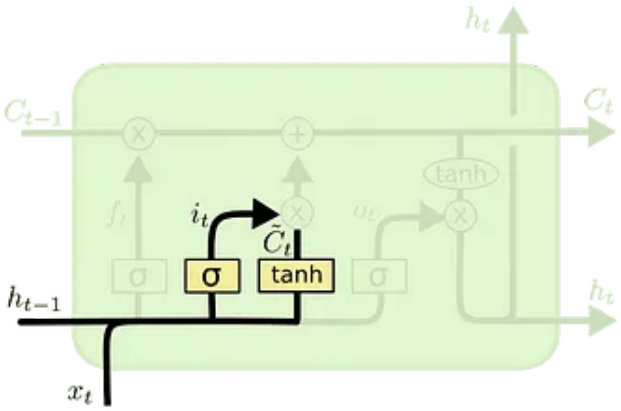
Forget gate layer

Step 2: What new information to *add* back

Second step involves deciding what new data should be added back to the cell state. This involves 2 parts and are used in conjunction.

a) Sigmoid gate layer deciding what exactly to change (with 0 to 1 range)

b) Tanh gate layer creating candidate values that could be added to the cell state (with -1 to +1 range)

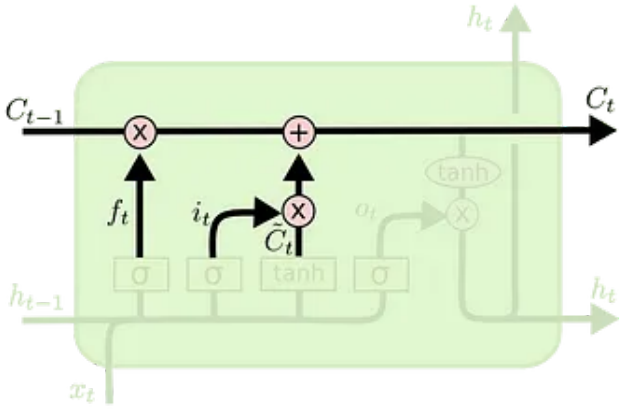


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Sigmoid and Tanh deciding what to manipulate and by how much at input gate layer

Step 3: To actually update the cell state with information from steps 1 and 2

This step includes manipulating the incoming cell state C_{t-1} to reflect the decisions from forget (step 1) and input gate (step 2) layers with help of point-wise multiplication and addition respectively.

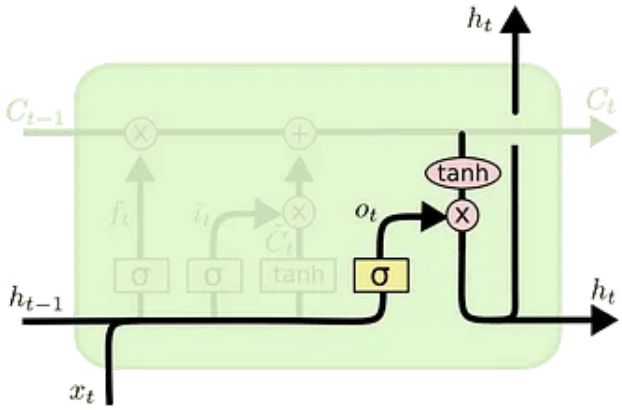


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Carrying out the calculations to actually update cell state

Here f_t is for point-wise multiplication with C_{t-1} to *forget* information from context vector and $i_t * C_t$ is new candidate values scaled by how much they needs to be updated.

Step 4: What's the output



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

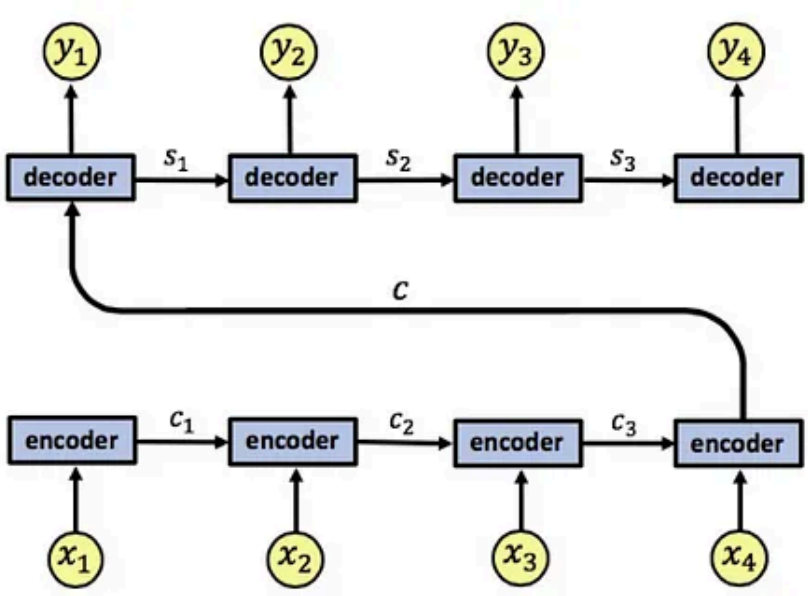
Output gate calculations

The output is a modified version of the cell state. For this sigmoid decides what part needs to be modified, which is multiplied with output after Tanh (used for scaling), resulting in hidden state output.

. . .

RNNs and LSTMs have been used widely for processing and making sense of sequential data. The usual approach is the encoder-decoder architecture for seq2seq tasks.

> ***Sequence-to-sequence (seq2seq)*** *models in NLP are used to convert sequences of Type A to sequences of Type B. For example, translation of English sentences to German sentences is a sequence-to-sequence task.*

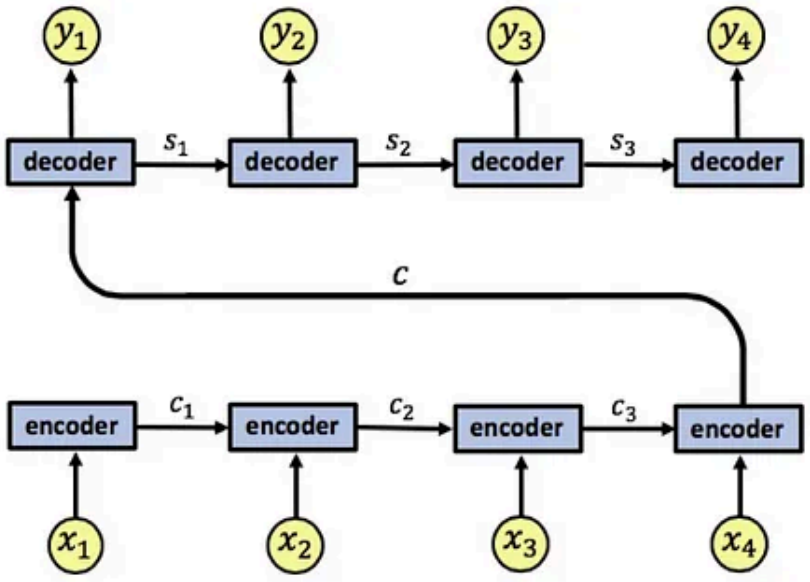


The encoder has an input sequence *x1, x2, x3, x4*. We denote the encoder states by *c1, c2, c3*. The encoder outputs a single output vector ***c*** which is passed as input to the decoder. Like the encoder, the decoder is also a single-layered RNN, we denote the decoder states by *s1, s2, s3* and the network's output by *y1, y2, y3, y4*.

. . .

Attention Mechanism: A superhero! But why? 🤔

The *primary question* to (re)consider — what exactly is being demanded from RNNs and LSTMs (in encoder-decoder architecture): *current node or state has access to information for whole input seen so far* (i.e. the information flowing from t_0 till t_{-1} is available in some modified/partial form for state at time step *t*), **therefore**, the final state of RNN (better to say encoder) must hold information for the entire input sequence.



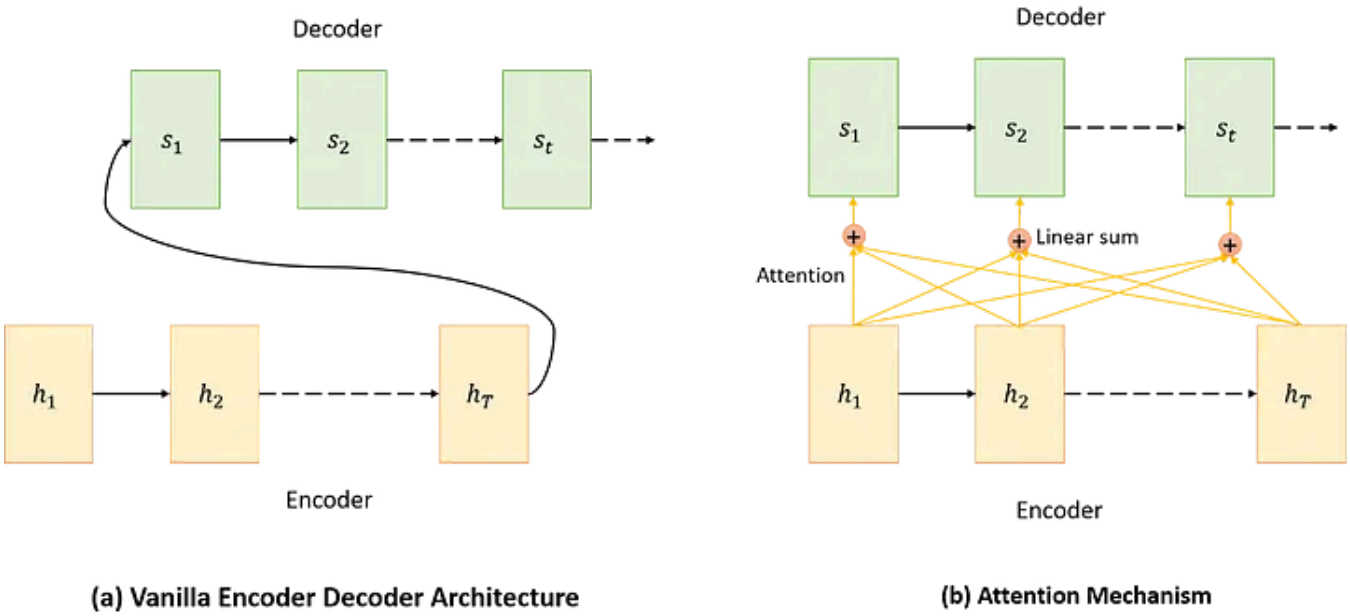
Simple representation of traditional Encoder-Decoder architecture using RNN/LSTM

A major drawback with this architecture lies in the fact that the encoding step needs to represent the entire input sequence ***x1, x2, x3, x4*** as a single vector ***c****, which can cause information loss* as all information needs to be *compressed into* ***c***. **Moreover**, the decoder needs to decipher the passed information from this single vector only, *a highly complex task in itself indeed.*

And this is exactly where upper bound on model's understanding potential as well as performance sets in quickly as this may be ***too much to ask*** from simple encoders and decoders for sequential processing of long input sequences.

> *Attention Mechanism for sequence modelling was first used in the paper: Neural Machine Translation by jointly learning to align and translate, Bengio et. al. ICLR 2015. Although the notion of 'attention' wasn't as famous back then and the word has been sparingly used within the paper itself.*
>
> ***As explained in the paper*** *[1]*
>
> *A potential issue with this encoder–decoder approach is that a neural network needs to be able to compress all the necessary information of a source sentence into a fixed-length vector. This may make it difficult for the neural network to cope with long sentences, especially those that are longer than the sentences in the training corpus.*

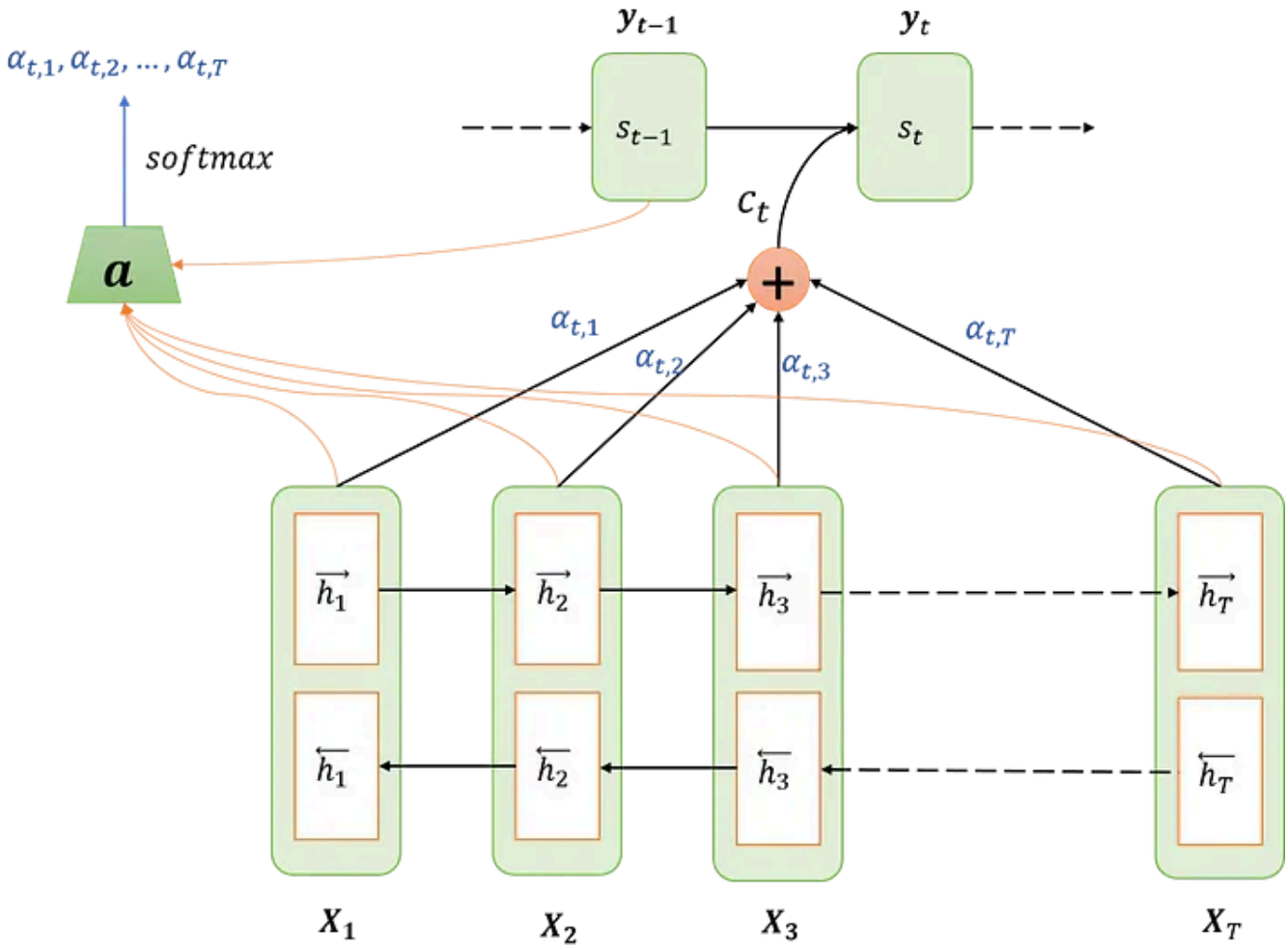
Attention mechanism helps to look at all hidden states from encoder sequence for making predictions unlike vanilla Encoder-Decoder approach.



Simple vs Attention based encoder-decoder architectures

The difference in preceding image explained: *In a simple Encoder-Decoder* architecture the decoder is supposed to start making predictions by looking only at the *final output* of the encoder step which has *condensed information. On the other hand,* ***attention*** based architecture ***attends*** every hidden state from each encoder node at every time step and *then* makes predictions after deciding which one is *more informative.*

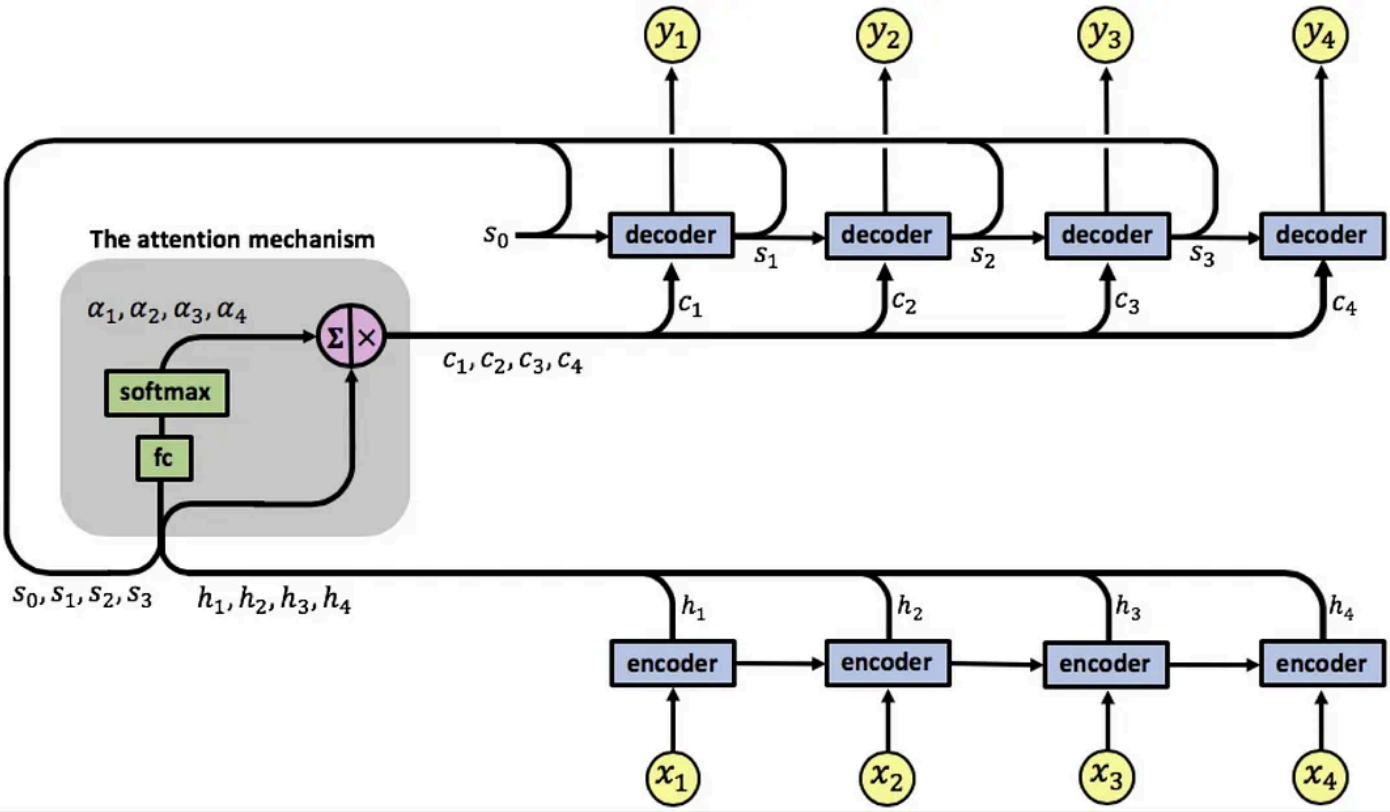
> ***Wait !!***
>
> *But how does it decide which states are more or less useful for every prediction at every time step of decoder?*
>
> ***Solution:*** *Use a Neural Network to "learn" which hidden encoder states to "attend" to and by how much.*



Attention Mechanism architecture overview

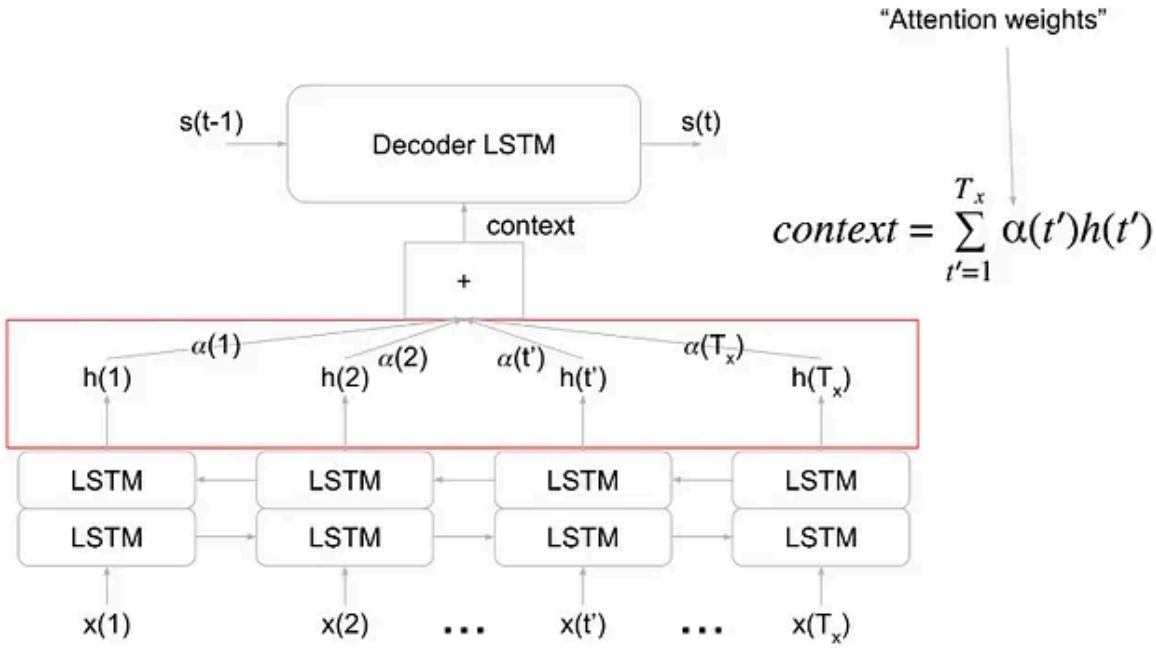
All steps in Attention mechanism:

1. Encoding
2. Computing Attention weights
3. Creating context vector
4. Decoding / Translation



Another simplified version of complete Attention based architecture

Calculating attention weights:

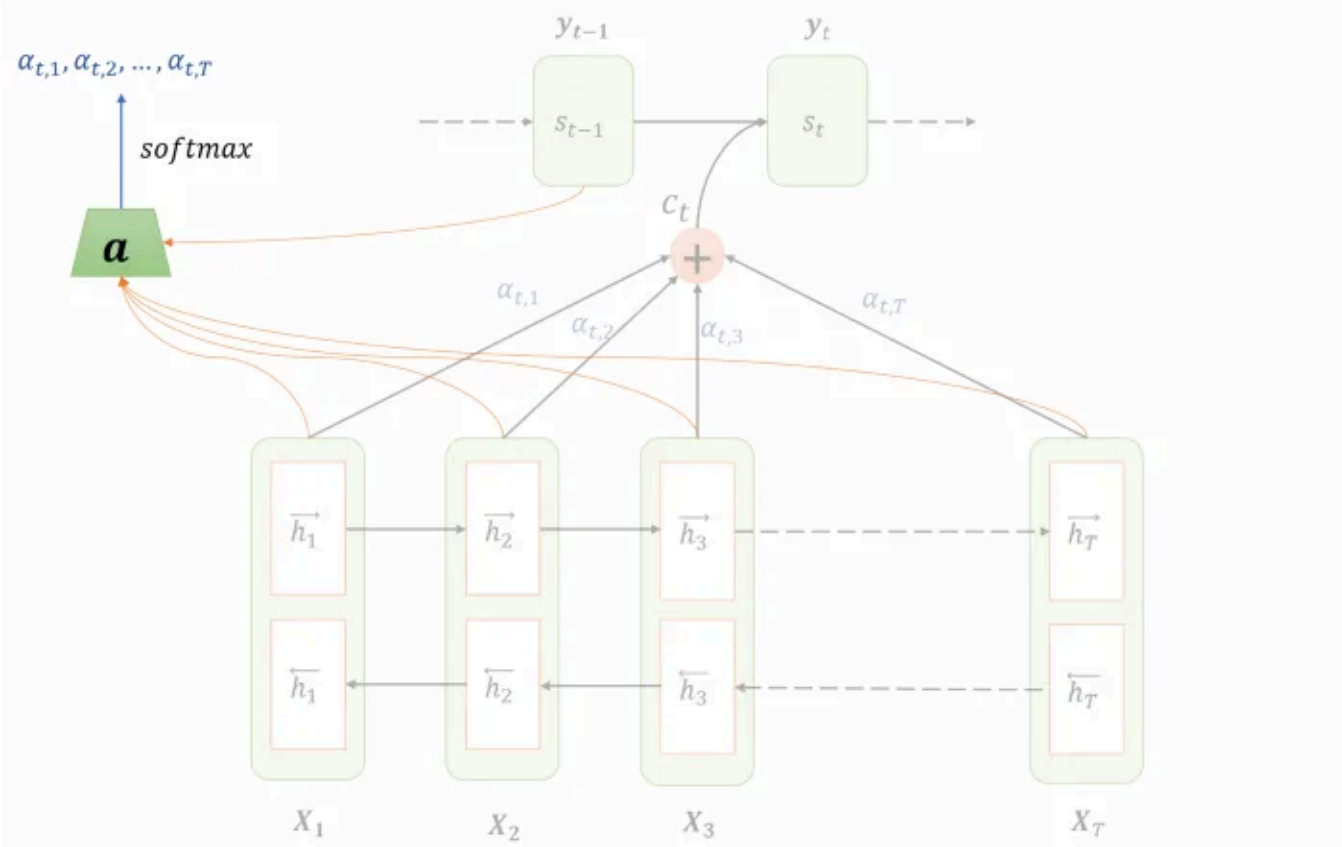


Bi-directional LSTMs used in encoder sequence along with attention weights

$$\alpha_{t'} = NeuralNet([s_{t-1}, h_{t'}]),\ t' = 1 \ldots T_x$$

$$context = \sum_{t'=1}^{T_x} \alpha(t')h(t')$$

Calculating attention weights and creating the context vector using those attention values with encoder state outputs



Isolating calculation of attention weights for better understanding. The output from each hidden state is input to 'attention' NN for every time step along with previous decoder predicted output

The input to attention block is the output from each encoder unit as well as previous time step decoder output. For each time step ***t*** at the decoder, the level/amount of attention to be paid to the hidden encoder unit h_j is denoted by α_{tj} and calculated as a function of both h_j and previous hidden state of decoder s_{t-1}:

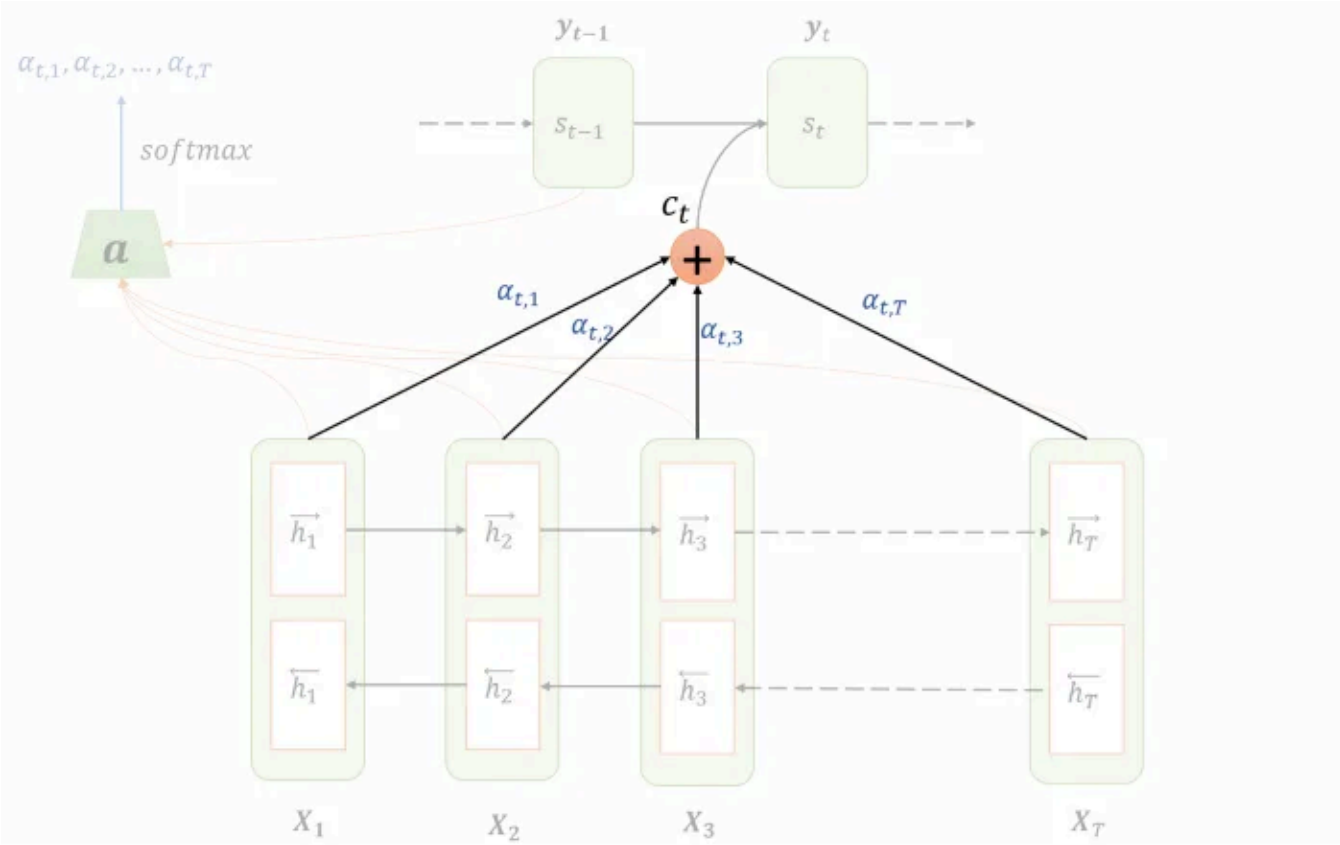
This NN uses a ***Softmax*** unit at the end to normalize the outgoing attention values hence the sum of all values=1

$$e_{tj} = \boldsymbol{a}(h_j, s_{t-1}), \forall j \in [1, T]$$

$$\alpha_{tj} = \frac{\exp(e_{tj})}{\sum_{k=1}^{T} \exp(e_{tk})}$$

Remember: The attention Neural Network will output attention values at each time step for the decoder and takes in output from previous decoder step. *The first time step at decoder will have* S_0 *as empty vector*. Thus the output/prediction from decoder step which goes as input to attention NN keeps changing for each time step and so does the decision for attention and the attention values.

Creating context vector using attention weights:



Context vector formed using hidden state output from each time step along with corresponding attention value

Simply put, the context vector is a simple linear combination of the hidden values $\mathbf{h_j}$ weighted by the attention values $\boldsymbol{\alpha_{tj}}$:

$$c_t = \sum_{j=1}^{T} \alpha_{tj} h_j$$

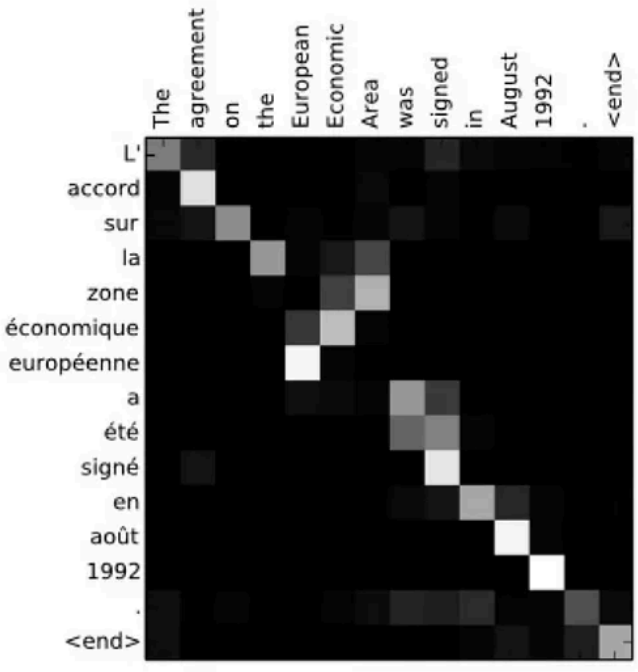
Context Vector

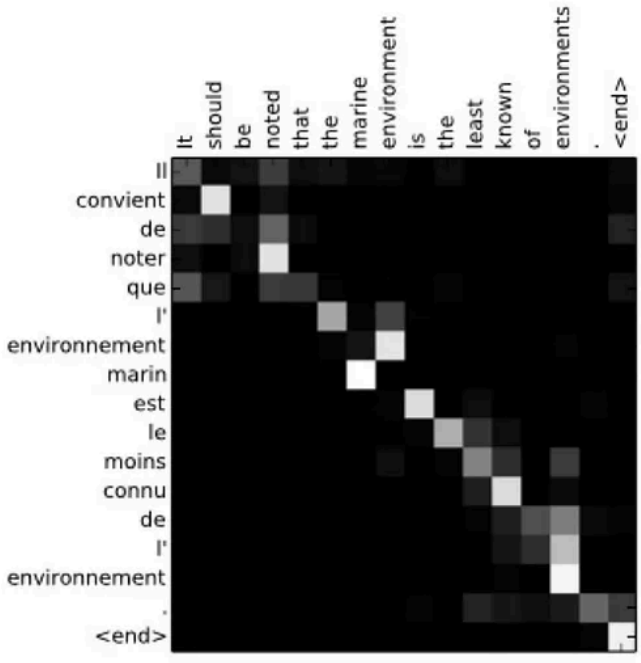
Final step is the decoding step and using the context vector. $\mathbf{c_t}$ is used along with the previous hidden state of the decoder s_{t-1} to compute the new hidden state and output of the decoder: $\mathbf{s_t}$ and $\mathbf{y_t}$.

. . .

Below is an example given in [1], the **English-French machine translation** task. Below are two alignments found by the attention RNN. The x-axis and y-axis of each plot correspond to the words in the source sentence (English) and the generated translation (French), respectively.

Each pixel shows the weight αij of the j-th source word and the i-th target word, in grayscale (0: black, 1: white).





Here you can clearly see how the attention mechanism allows the RNN to focus on a small (and also sometimes different parts with different "attention" levels simultaneously) of the input sentence when performing the translation. Pay attention to the larger attention values (the white pixels) connecting corresponding parts of the English and French sentences. **This approach allowed authors to achieve great results on the experiments.**

. . .

Will next publish on Transformer architecture (introduced by Google in “Attention is all you need” and used in their BERT architecture as well as in GPT by OpenAI), which is our ultimate goal to utilize attention mechanism, as it shook up the NLP and now vision AI domain.

Thanks for reading.

. . .

References:

1. https://arxiv.org/pdf/1409.0473.pdf
2. https://colah.github.io/posts/2015-08-Understanding-LSTMs/
3. https://shashank7-iitd.medium.com/understanding-attention-mechanism-35ff53fc328e
4. https://medium.com/datadriveninvestor/attention-in-rnns-321fbcd64f05
5. https://towardsdatascience.com/illustrated-guide-to-recurrent-neural-networks-79e5eb8049c9

Deep Learning | Attention Mechanism | Artificial Intelligence | Machine Learning | Recurrent Neural Network

Follow

Written by Manu Chauhan

72 followers · 201 following

A curious being, Python & ML Engineer, interested in AI and other technological advancements

Responses (8)

Write a response

What are your thoughts?

Saksham Shahu
Aug 11, 2023

A very good article indeed

2 Reply

Samarth Agarwal
Mar 17, 2024

A very good article. Really liked the animations and the way it was explained.

1 Reply

Alaazuhd
Mar 27, 2022

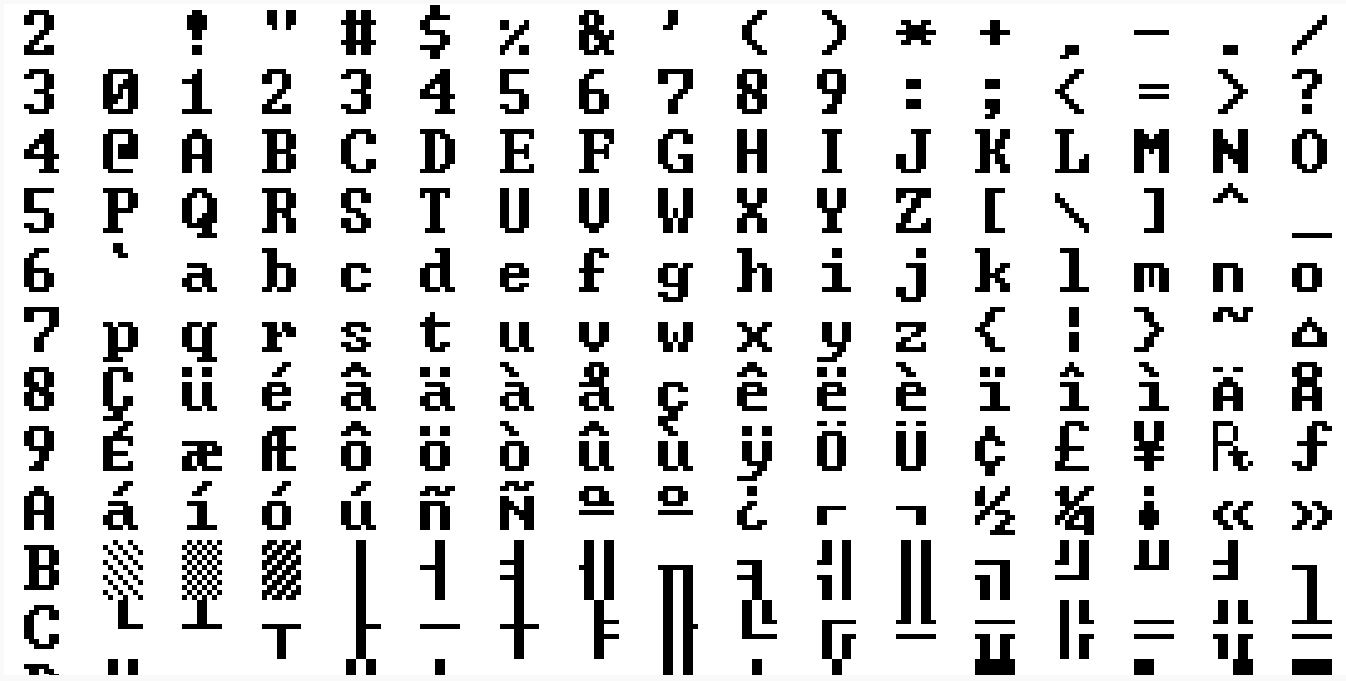
> The incoming sequential data is encoded using RNN first before being utilized to determine the intent/action via another feed forward network for decision

Can't understand this point

1 Reply

See all responses

More from Manu Chauhan and The Startup



Manu Chauhan

Taming and training LLM Tokenization

Getting on LLM tokenization while building a Hindi tokenizer from scratch with Byte Pair Encoding. (for purely experimentation purposes)...

Sep 6, 2024

In The Startup by Divad Sanders

4 Boring Startup Ideas Screaming to Be Built (and How to Build Them)

Build the unsexy thing. Charge for outcomes. Repeat.

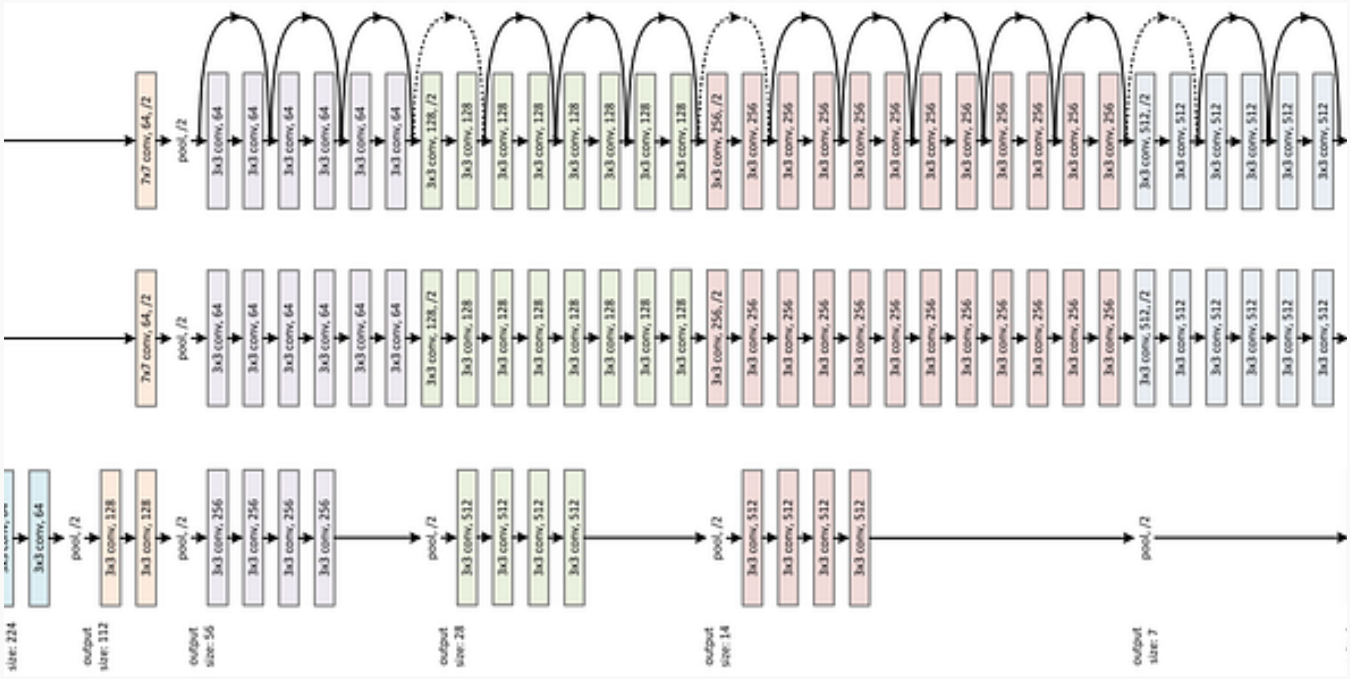
Apr 8 4.1K 131

In The Startup by Divad Sanders

How to Build a Boring AI Startup That Quietly Hits $10M ARR

Ugly processes = untapped margins

Apr 12 2.1K 54



Manu Chauhan

What are deep Residual Networks or Why ResNets are important ?

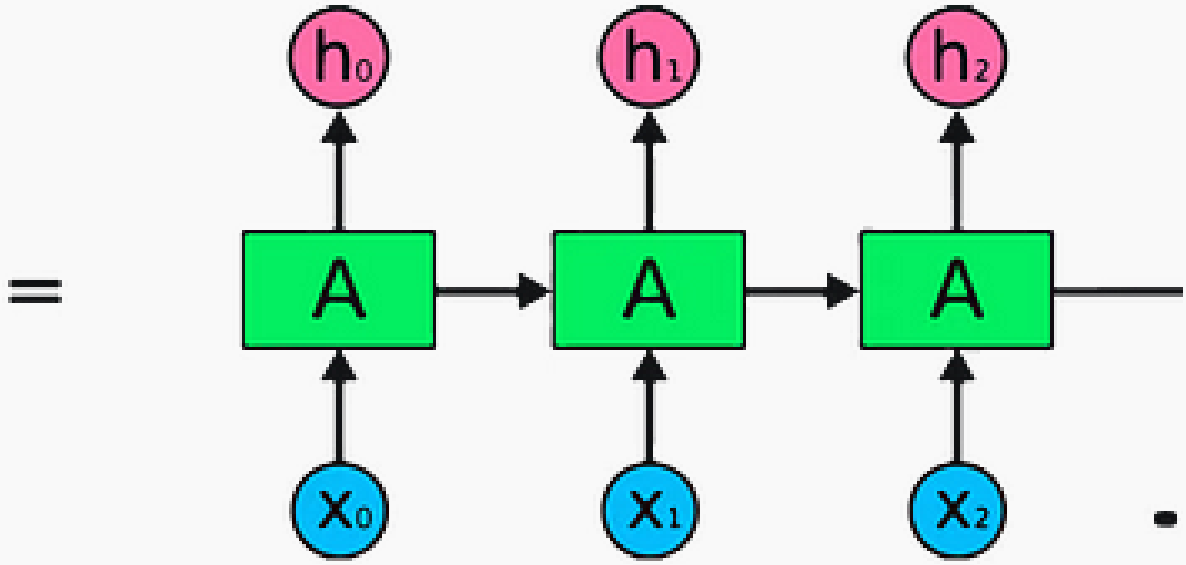
A research paper summary and code.

Apr 19, 2019 102 1

See all from Manu Chauhan

See all from The Startup

Recommended from Medium

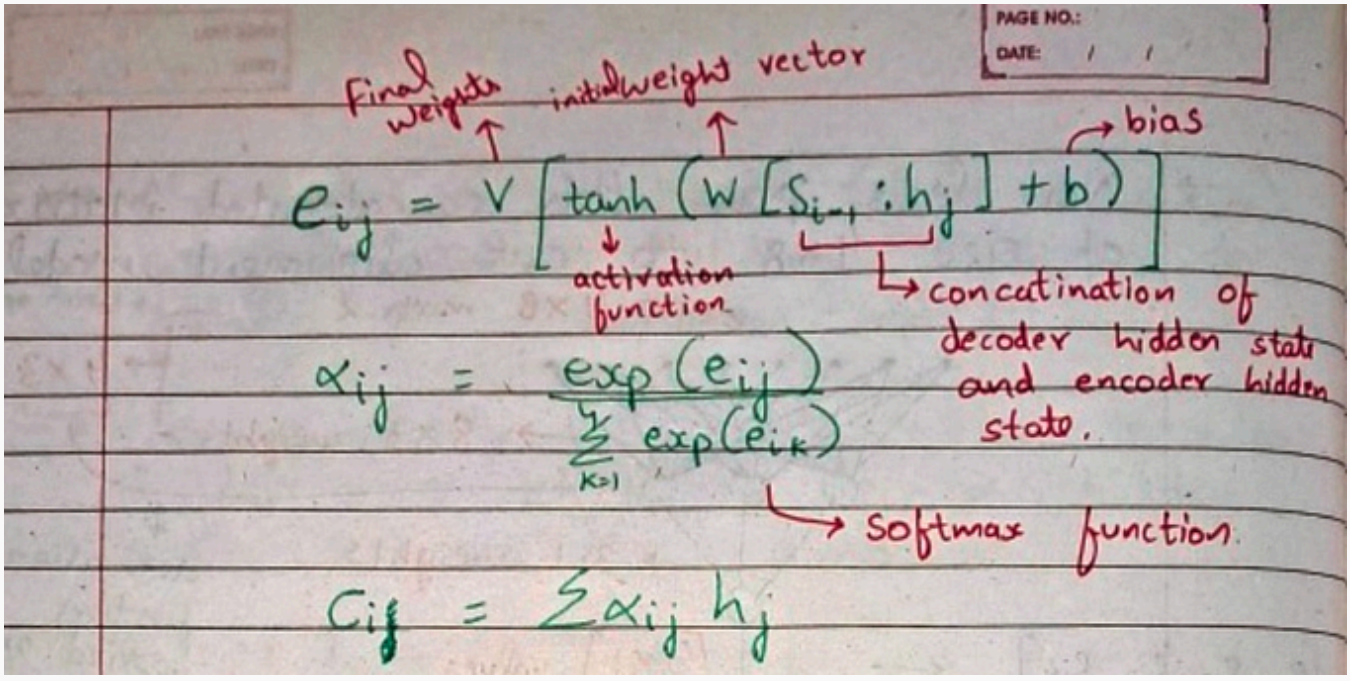


Arjhun Sreedar

Let's build an LSTM from scratch.

This article assumes that you know how basic RNNs work.

Feb 1

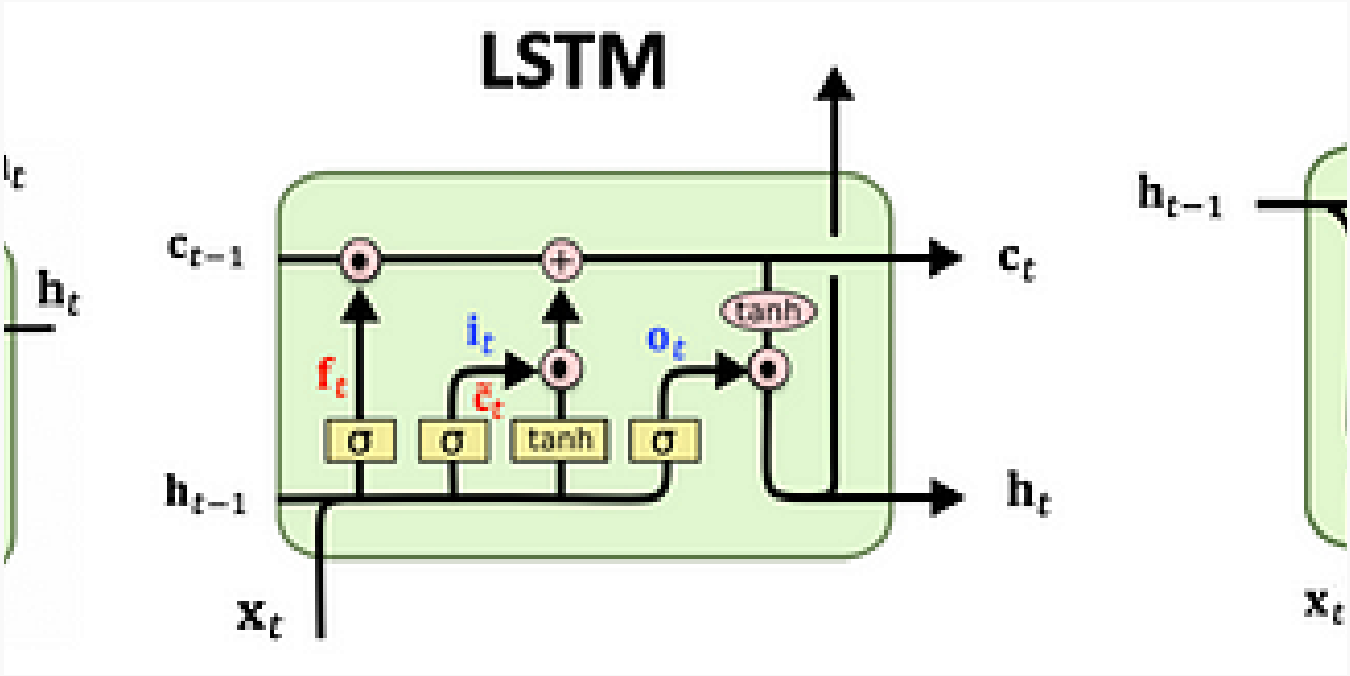


Navdeep Sharma

Bahdanau vs. Luong Attention: Unpacking Two Key Mechanisms in Neural Networks

A Deep Dive into Additive and Multiplicative Attention: How They Work and Why They Matter

Jan 11 4



LM Po

A Journey Through RNN, LSTM, GRU, and Beyond

If you're not a Medium subscriber, click here to read the full article.

Feb 14 21

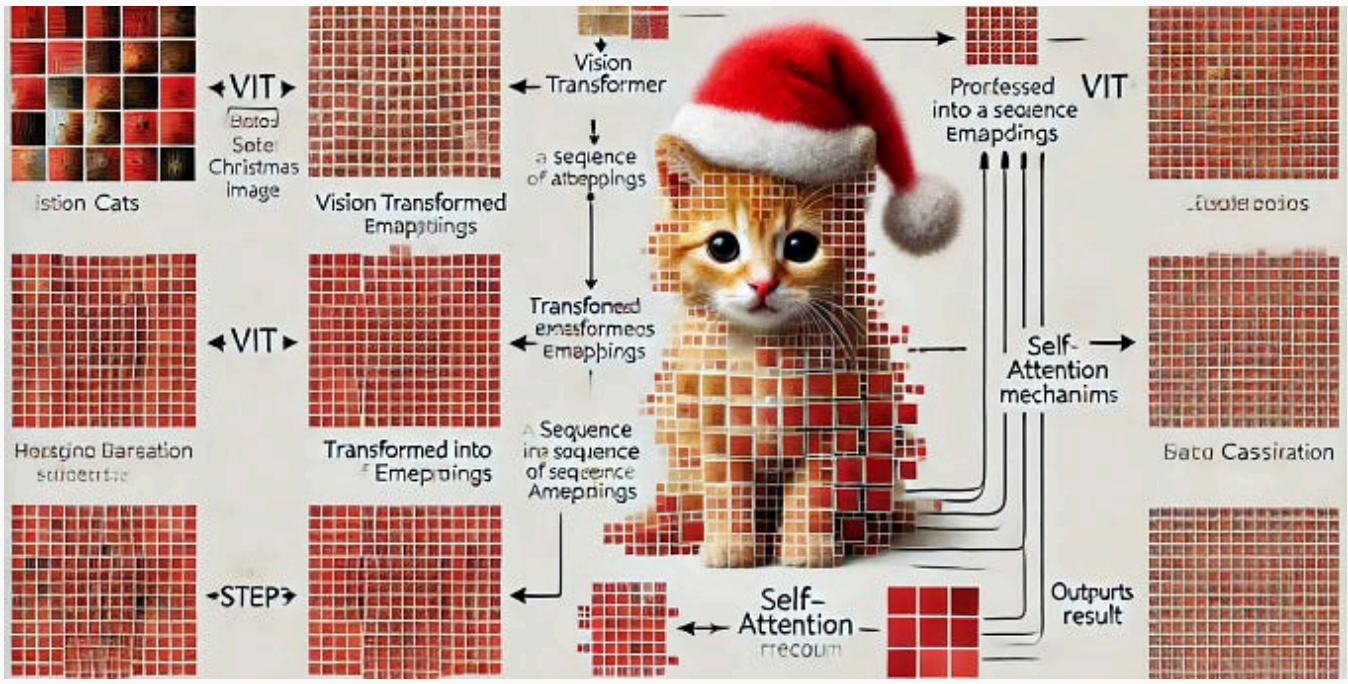
re	Decoder-Only	Encoder-Only	Encoder-Decoder
se	Text generation	Text understanding	Sequence-to-sequence
ionality	Unidirectional	Bidirectional	Encoder: bidirectional, Decode unidirectional
tion anism	Causal masking	Full bidirectional	Cross-attention in the decoder
gths	Coherent text output	Rich contextual representations	Handling complex input-outpu mappings
l Tasks	Autoregressive tasks	Analysis and extraction	Translation, summarization, te to-text

Chris Yan

Understanding Transformer Architectures: Decoder-Only, Encoder-Only, and Encoder-Decoder Models

The Standard Transformer was introduced in the seminal paper “Attention is All You Need” by Vaswani et al. in 2017. The Transformer...

Nov 20, 2024

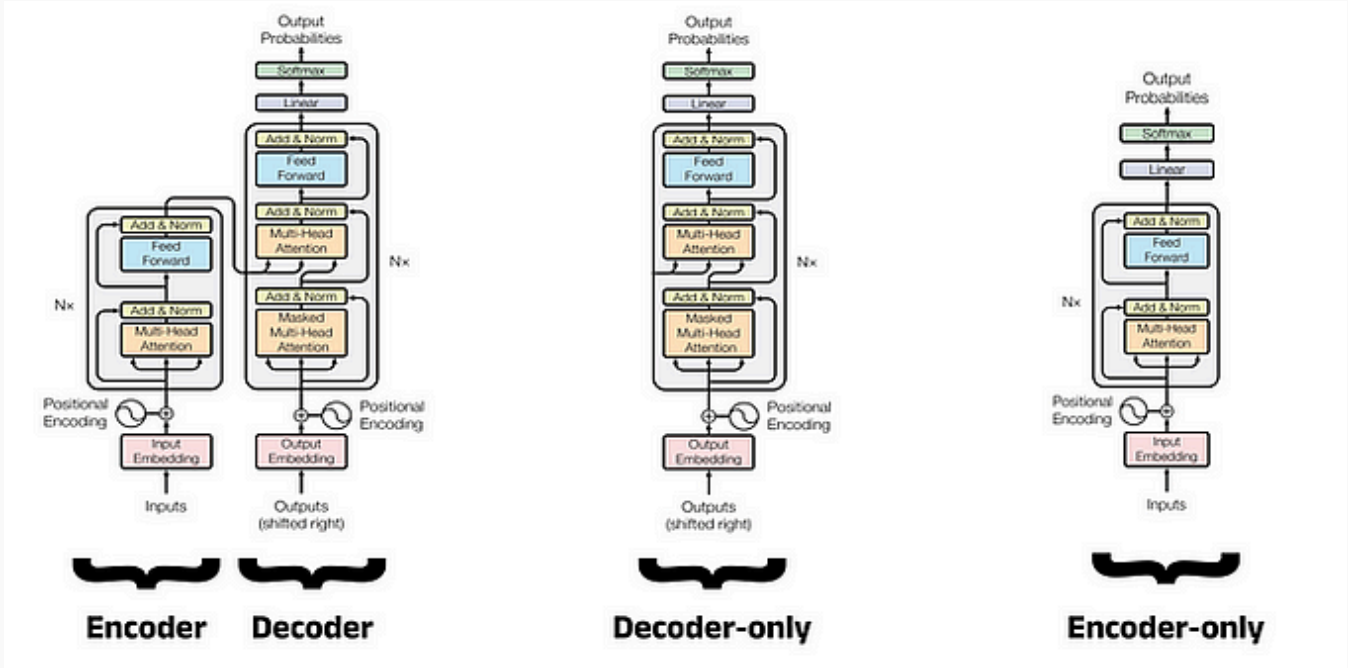


In Tech Spectrum by Aarafat Islam

How Vision Transformers Work?

The Paradigm Shift in Computer Vision

Nov 20, 2024 142 1



Vipra Singh

LLM Architectures Explained: BERT (Part 8)

Deep Dive into the architecture & building real-world applications leveraging NLP Models starting from RNN to Transformer.

Nov 18, 2024 278 2

See more recommendations