Recurrent Neural Networks

DEEP LEARNING COURSE, FALL 2025

Practicalities

FEEDBACK ON ASSIGNMENT 1

Most of you did really well ☺

(at least that's what the TA's have said)

If you haven't gotten feedback from your TA yet, I think it is fair to ask them (politely!) when they think they will get there.

FEEDBACK ON ASSIGNMENT 1

Two things which were pointed out:

1:

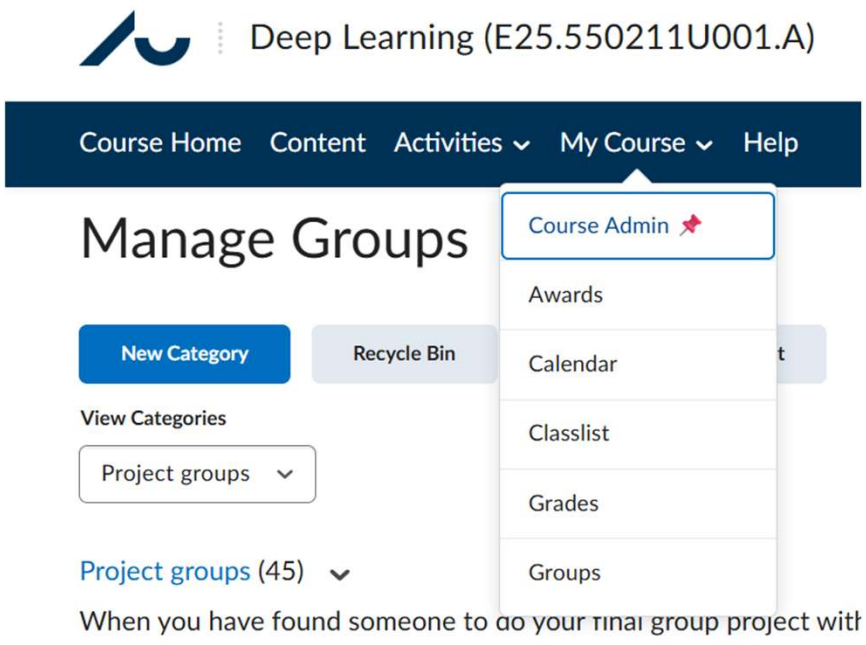
Train split ≠ validation split ≠ test split

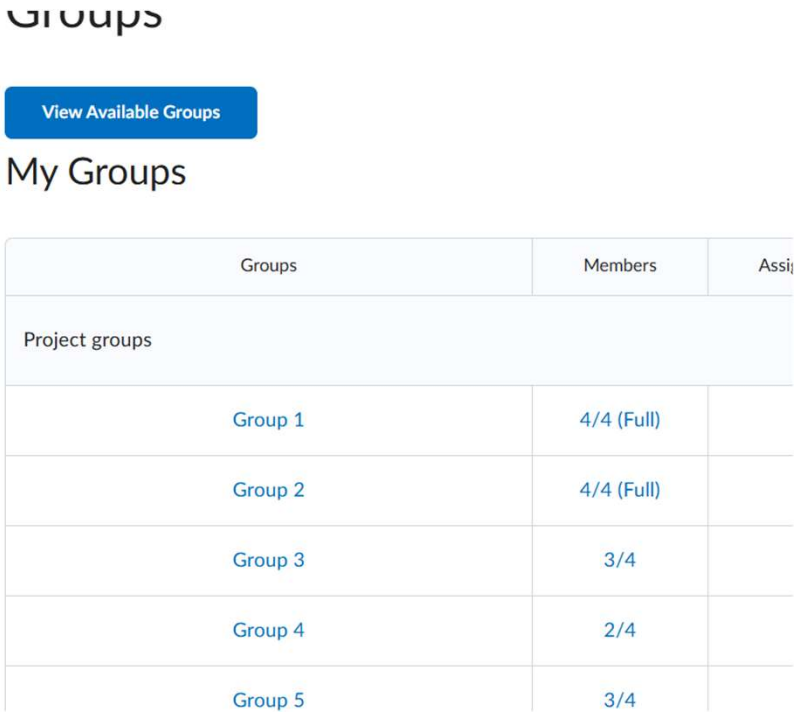
2: People may be using GAI more than they're admitting?

- Examples: can't count neurons in the plot? <- copilot makes this error
- Lots of code, no explanation for what it does

At the exam, we have to be really vigilant about people 'outsourcing' the work. That means inconsistent or missing explanations are going to impact the grade a lot.

Group signup





Plan

Sequence problems

RNN's

Attention

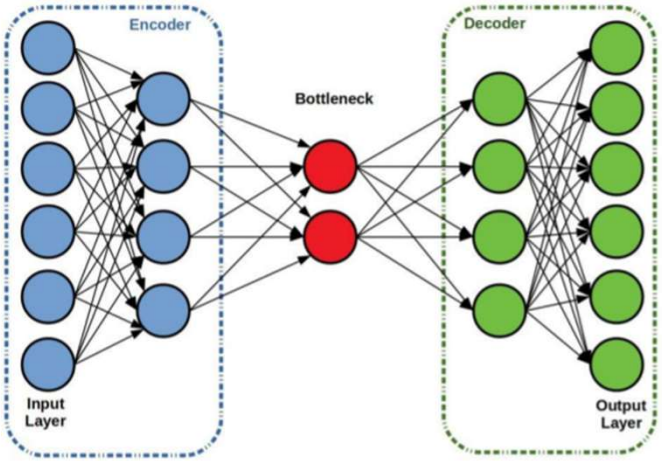
Transformers

Return of the RNN:

Reading & exercises

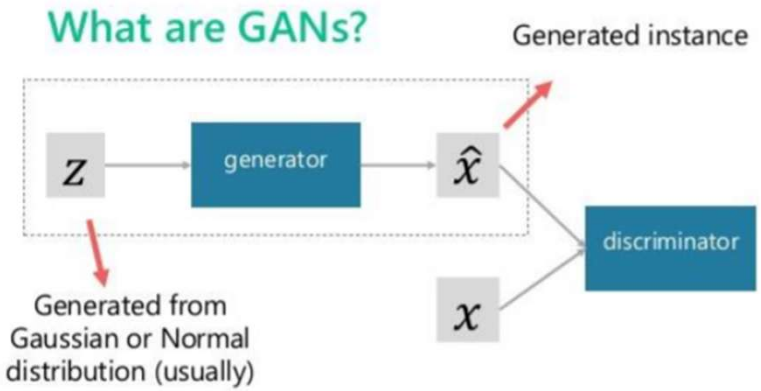
Last week:

Autoencoders:



Variational autoencoders:

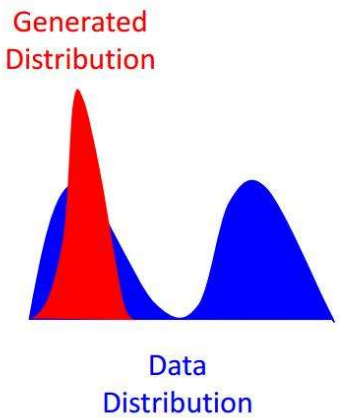
Generative adversarial networks



Quiz:

What is ‘mode collapse’?

Mode Collapse



Sequence models

TEXT, MUSIC, TIME SERIES!

Sequence models

Examples:
One-to-one: what we have been looking at until now
Many-to-many: machine translation
Many-to-one: sentiment classification (IMDB reviews)
One-to-many: music generation

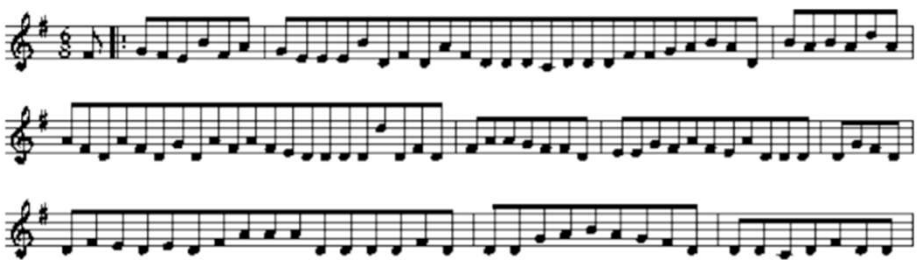
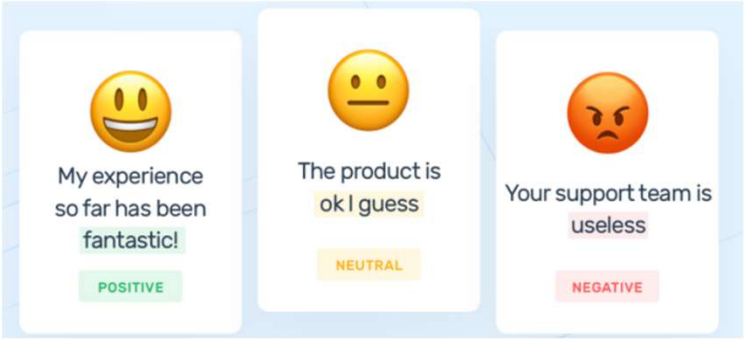


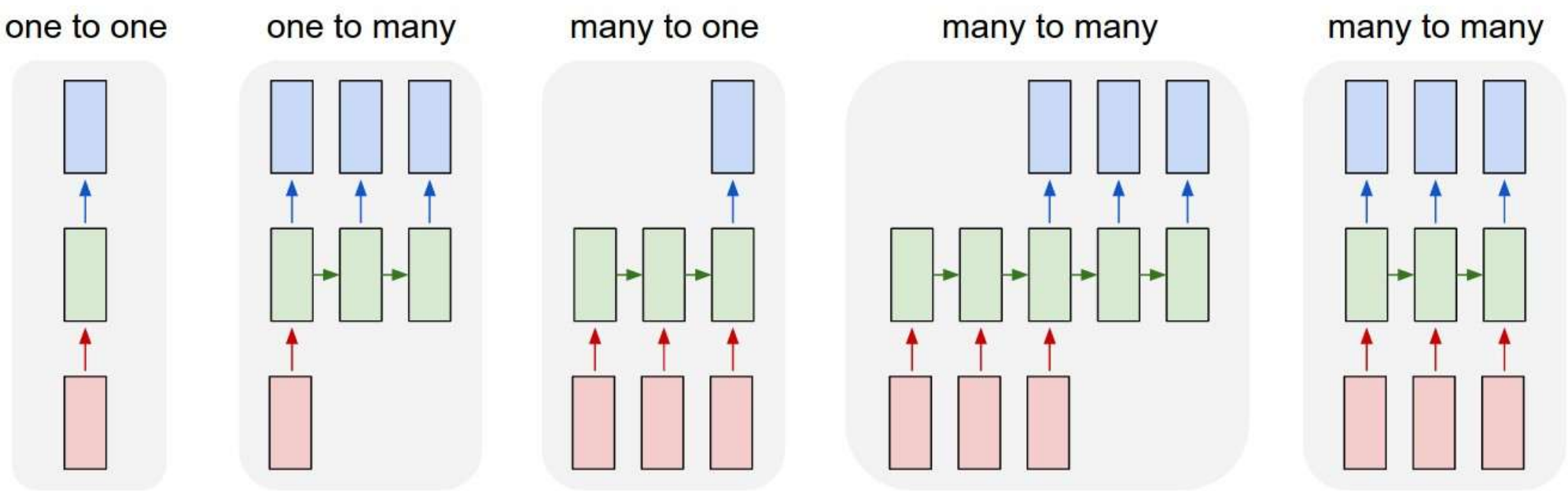
Figure 9: A Sample Music Generated by Char-RNN Model



Source: https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1174/reports/2762076.pdf

Sequence models

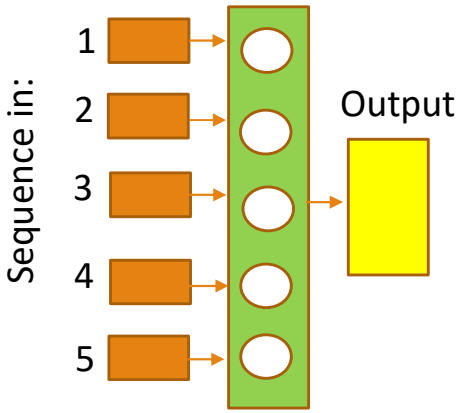
Different visualization – note that the two last approaches are very different:



Source: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

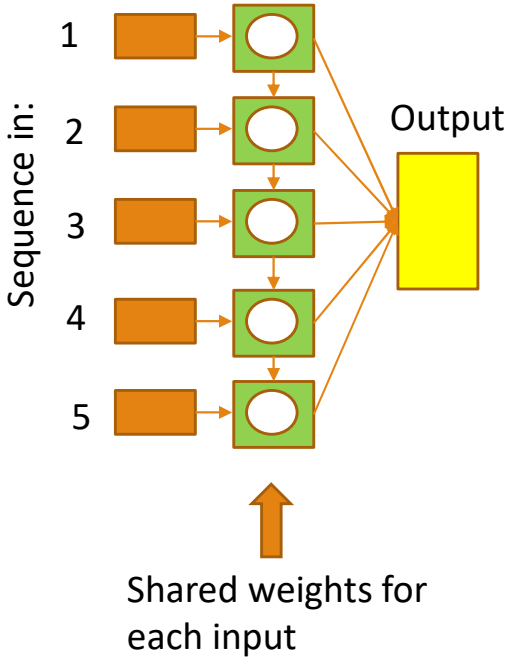
Recurrent Neural Networks (RNN)

Imagine a network which takes a sequence of objects as input:



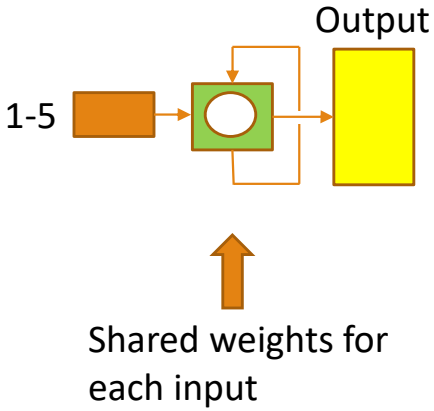
If our sequence is long, this can become a very large network, and we are not using the symmetries between identical inputs.

We can try to use the symmetries, by focusing on a single element, and reusing:



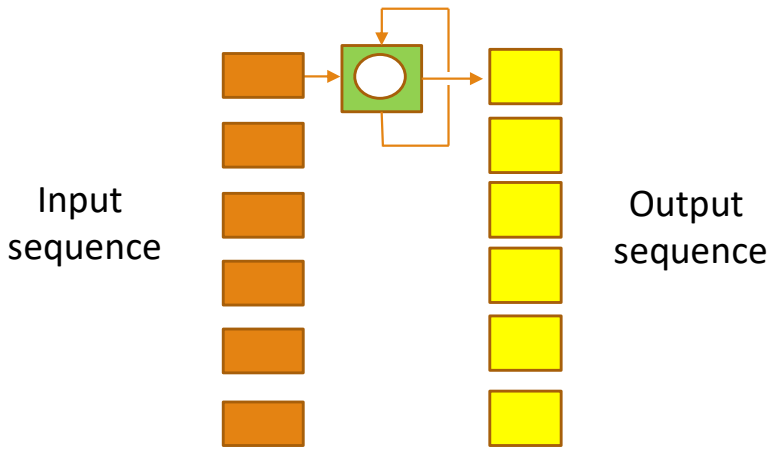
These are the same

Simplifying further:



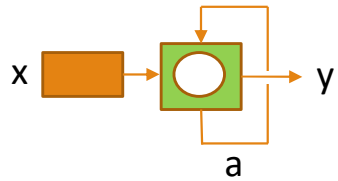
Recurrent Neural Networks (RNN)

Please notice that it is the same piece of code analysing each input consecutively:



(Also, the state from the previous call is input to the next call)

Recurrent Neural Networks (RNN)

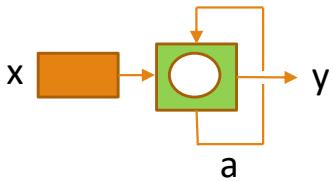


One of the simplest sets of equations used to govern this system is:

$$a_i = g_a(\overbrace{W_{ax}x_i}^{\text{Input}} + \overbrace{W_{aa}a_{i-1}}^{\text{Recurrence}} + \overbrace{b_a}^{\text{Bias}})$$
$$y_i = g_y(W_{ya}a_i + b_y)$$

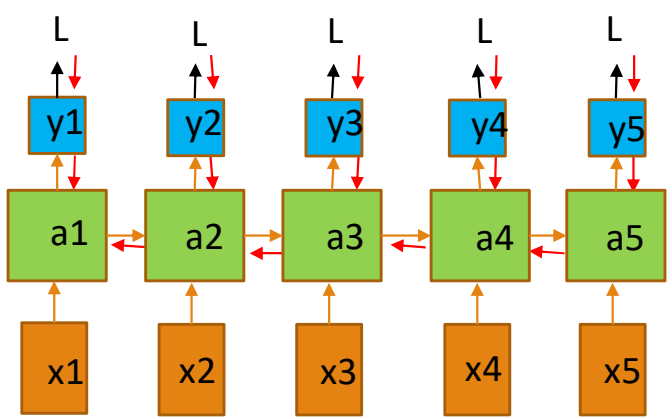
Where the *g*'s are activation functions. For the first input, a_{i-1} is undefined. The most common initiation of a_{i-1} is simply a vector of zeros.

Backpropagation through time



Question: How should we implement backpropagation?

Consider again the rolled-out RNN:



Add losses

And backpropagate them

We can write out the gradient of the loss for a specific step:

$$\frac{\partial L_3}{\partial W_{aa}} = \frac{\partial L_3}{\partial y_3}\frac{\partial y_3}{\partial a_3}\frac{\partial a_3}{\partial W_{aa}} = \frac{\partial L_3}{\partial y_3}\frac{\partial y_3}{\partial a_3}\sum_{i\leq 3}\frac{\partial a_3}{\partial a_i}\frac{\partial a_i}{\partial W_{aa}}$$

In the second step, we use that a_3 is a function of all previous a_i as well as w_{aa}. $\frac{\partial a_3}{\partial a_i}$ can itself be expanded using the chain rule:

$$\frac{\partial L_3}{\partial W_{aa}} = \frac{\partial L_3}{\partial y_3}\frac{\partial y_3}{\partial a_3}\sum_{i=0}^{3}\prod_{j=i-1}^{3}\left(\frac{\partial a_j}{\partial a_{j-1}}\right)\frac{\partial a_i}{\partial W_{aa}}$$

(continues next slide)

Sources: https://github.com/go2carter/nn-learn/blob/master/grad-deriv-tex/rnn-grad-deriv.pdf, http://www.wildml.com/2015/10/recurrent-neural-networks-tutorial-part-3-backpropagation-through-time-and-vanishing-gradients/

Backpropagation through time

In other words, we find:

$$\frac{\partial L_k}{\partial W_{aa}} = \frac{\partial L_k}{\partial y_k}\frac{\partial y_k}{\partial a_k}\sum_{i=0}^{k}\prod_{j=i-1}^{k}\left(\frac{\partial a_j}{\partial a_{j-1}}\right)\frac{\partial a_i}{\partial W_{aa}}$$

As you see, when we 'roll out' the RNN layer and treat it as a regular network, the approach is the same as always.

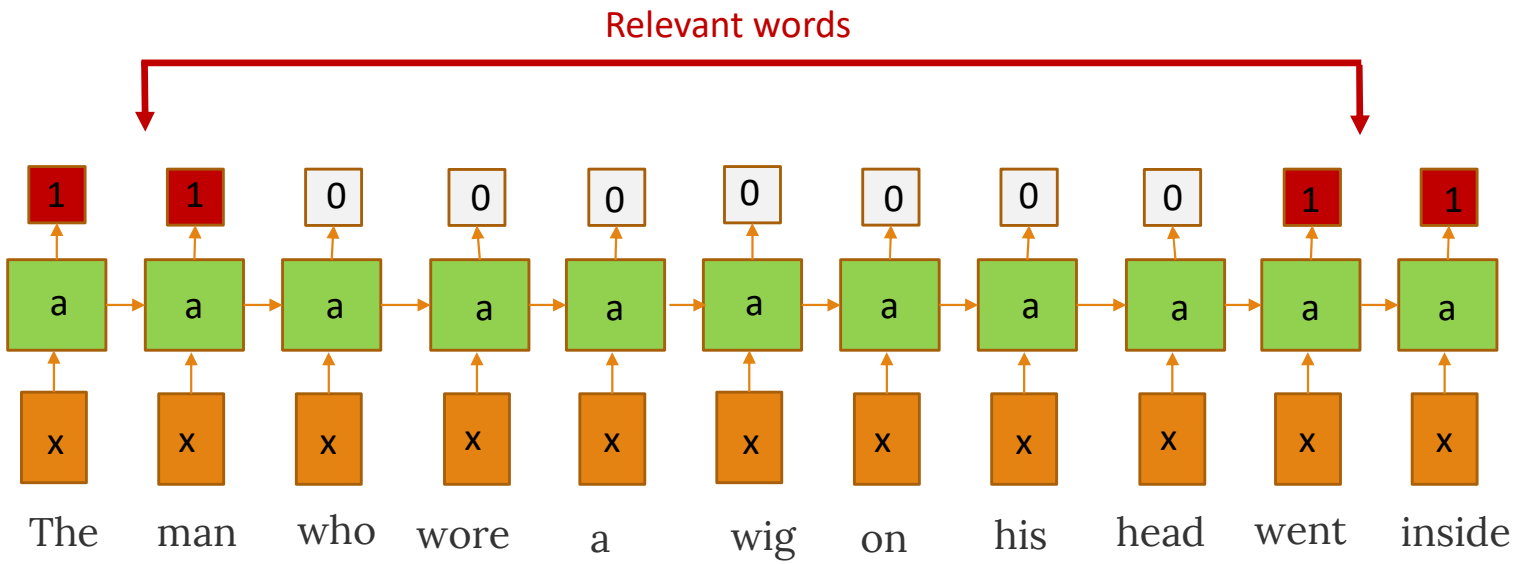
However, hidden inside the rather complicated expression above, are two sets of possible issues, related to the 'depth' of the network (*k*):

- Exploding gradients
- Vanishing gradients

Exploding gradients:
As we are potentially propagating through very many 'layers', all of which have very similar gradients, it's possible to have a gradient which grows exponentially in size (also, we are 'adding' losses as we go along). We call this an 'exploding' gradient, and it can ruin our training if it comes out of range (so it becomes 'NaN'). We can fix it by 'clipping' the gradient (making sure the abs value does not exceed some limit).

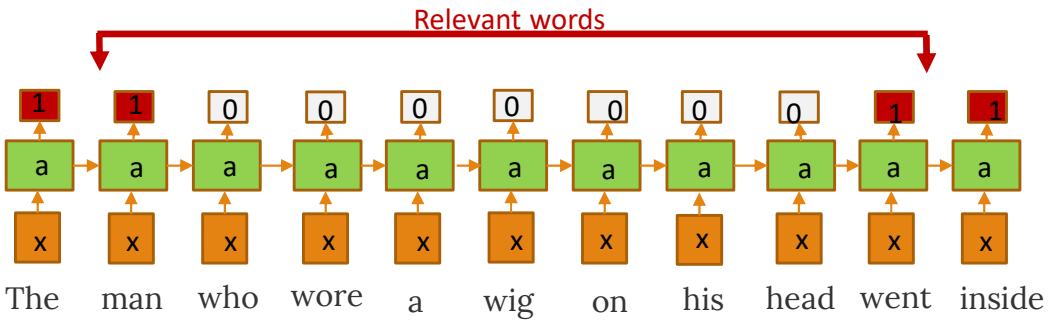
Vanishing gradients

The opposite situation can also be a problem:
When long inputs are used, it is possible that the network has to learn very long range connections.
Example: for the input '*The man who wore a wig on his head went inside*', it can be difficult to determine who or what went inside (there are 7 irrelevant words in the middle).



Nonlinear gates

A simple solution to the vanishing gradients, as we have discussed previously, is to use **RELU** activations instead of either tanh or sigmoids, which alleviates some of the problems.



However, there exist more elaborate solutions by increasing the complexity of the network. We will now look at the most important of these:

We can solve the vanishing gradients by making the response of the network 'more variable' – by introducing gates:

$$\Gamma(x) = \begin{cases} 1 \text{ for } x > c \\ 0 \text{ otherwise} \end{cases}$$

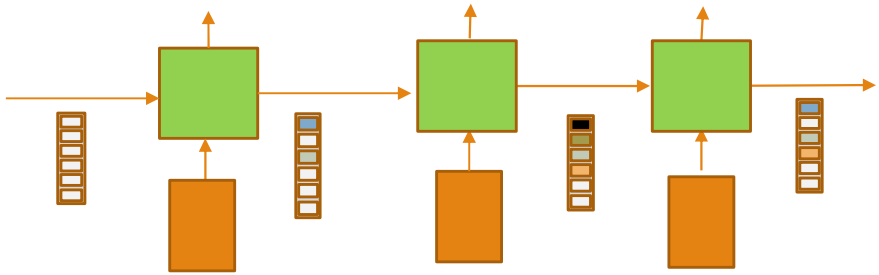
The idea is that if $a_i = \Gamma(x) a_{i-1}$, then it is both possible to transfer the gradient through many layers, while also resetting gradient information when needed.

Nonlinear gates: each cell is gated individually

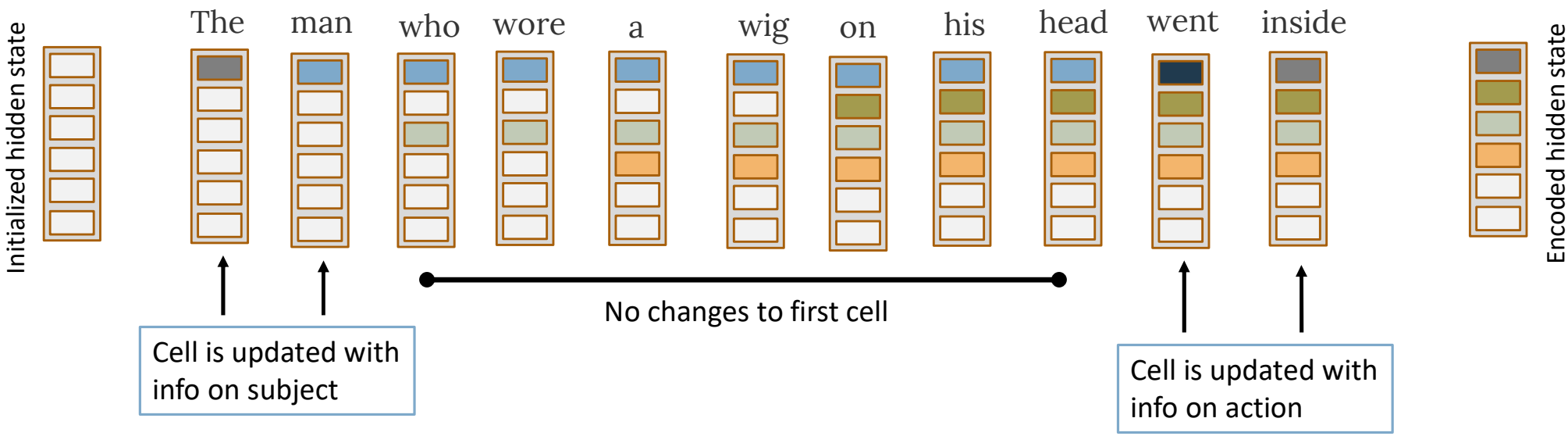
In the following, let this represent the hidden state (a vector) of an RNN unit:

We can populate the vector with information, which changes the appearance of the (abstract) cell containers inside:

Thus:



Nonlinear updating with gates:



With a non-linear gating mechanism, the contribution to the first cell in the hidden state vector is:

$$h(1) = \mathbf{1} \cdot \boldsymbol{f}(\boldsymbol{'The'}) + \mathbf{1} \cdot \boldsymbol{f}(\boldsymbol{'man'}) + 0 \cdot f('who') + 0 \cdot f('wore') + 0 \cdot f('a') + 0 \cdot f('wig') + 0 \cdot f('on') + 0 \cdot f('his') + 0 \cdot f('head') + \mathbf{1} \cdot \boldsymbol{f}(\boldsymbol{'went'}) + \mathbf{1} \cdot \boldsymbol{f}(\boldsymbol{'inside'})$$

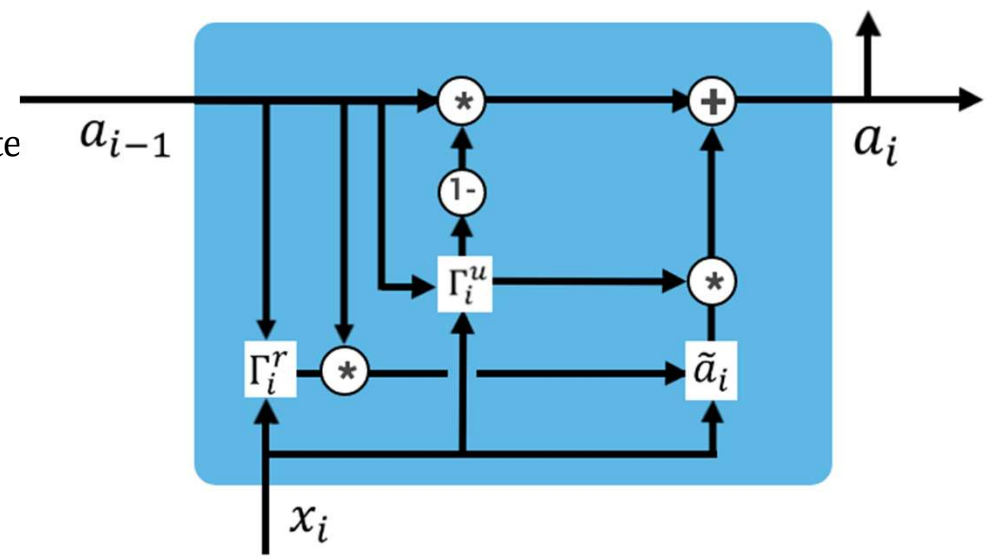
The idea is that h(1) depends strongly on a few, relevant terms, and not at all on the rest. To achieve this, we need a non-linearity between layers that efficiently turns individual contributions on or off. This also simplifies the gradient immensely.

This can be achieved with a **sigmoid** function, which is the standard gating nonlinearity.

Gated Recurrent Unit (GRU)

The GRU was introduced in 2015, as a simplification of the LSTM (next slide). As such, it is somewhat easier to describe, and makes for a good starting point:

$\Gamma_{\mathrm{i}}^{\mathrm{r}} = \sigma(W_{rx}x_i + W_{ra}a_{i-1} + b_r)$: reset gate
$\tilde{a}_i = \tanh(W_{ax}x_i + W_{aa}(\Gamma_i^r * a_{i-1}) + b_a)$: candidate
$\Gamma_{\mathrm{i}}^{\mathrm{u}} = \sigma(W_{ux}x_i + W_{ua}a_{i-1} + b_u)$: update gate
$a_i = (1 - \Gamma_i^u) * a_{i-1} + \Gamma_i^u * \tilde{a}_i$: activation / hidden state



In the above, '$*$' is an element-wise product (giving us the per-element gating).
σ is a sigmoid function.
Note how, depending on Γ_i^u, $a_i \approx a_{i-1}$ (as discussed on previous slide).

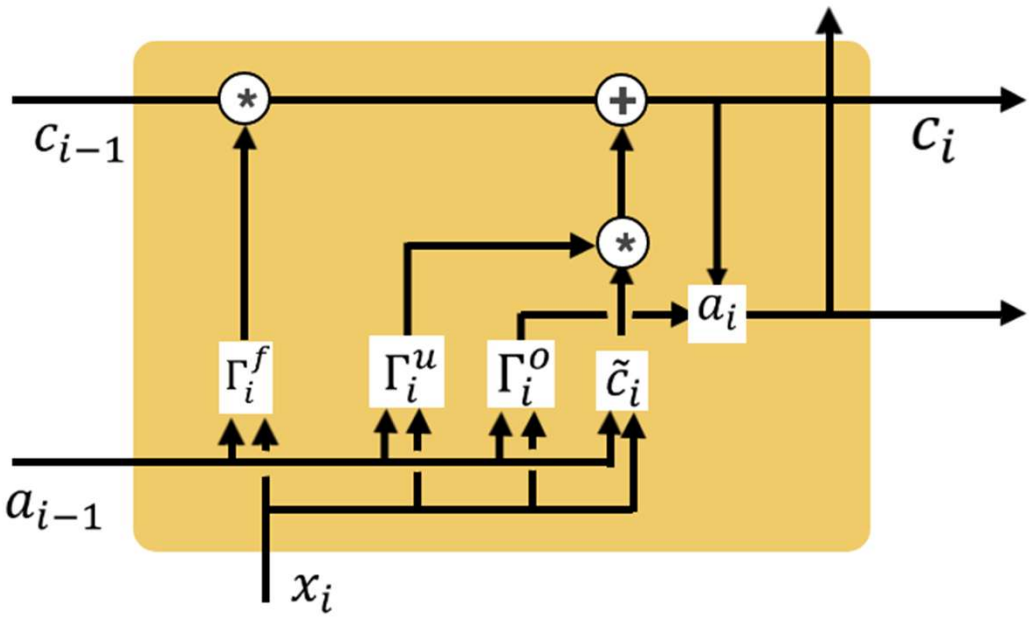
Note that in some presentations, gates have different letters, and 'a' is often 'h' or 'c'.

Source: https://pytorch.org/docs/stable/generated/torch.nn.GRU.html

Long Short Term Memory (LSTM)

The LSTM was introduced in 1997, and has an extra gate compared to the GRU cell:

$c_i = \Gamma_{\mathrm{i}}^{\mathrm{f}} * c_{i-1} + \Gamma_i^u * \tilde{c}_i$: cell state
$a_i = \Gamma_{\mathrm{i}}^{\mathrm{o}} * \tanh(c_i)$: hidden state
$\tilde{c}_i = \tanh(W_{cx}x_i + W_{ca}a_{i-1} + b_c)$: candidate
$\Gamma_{\mathrm{i}}^{\mathrm{o}} = \sigma(W_{ox}x_i + W_{oa}a_{i-1} + b_o)$: output gate
$\Gamma_{\mathrm{i}}^{\mathrm{u}} = \sigma(W_{ux}x_i + W_{ua}a_{i-1} + b_u)$: update gate
$\Gamma_{\mathrm{i}}^{\mathrm{f}} = \sigma\left(W_{fx}x_i + W_{fa}a_{i-1} + b_f\right)$: forget gate



Please note that the gates are still applied element wise ($*$). Again, in some presentations, gates have different letters, and 'a' can be 'h', for 'hidden state'.

Here, the 'cell state', c_i, can be carried unchanged quite far between layers, depending on gate values.

I see people using both a and c as the outputs of the LSTM.

Source: https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html

Comparison

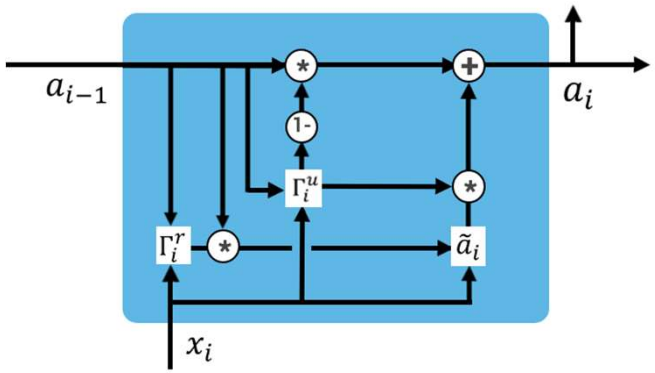
GRU:

$\Gamma_{\mathrm{i}}^{\mathrm{r}} = \sigma(W_{rx}x_i + W_{ra}a_{i-1} + b_r)$: reset gate

$\Gamma_{\mathrm{i}}^{\mathrm{u}} = \sigma(W_{ux}x_i + W_{ua}a_{i-1} + b_u)$: update gate

$\tilde{a}_i = \tanh(W_{ax}x_i + W_{aa}(\Gamma_i^r * a_{i-1}) + b_a)$: candidate

$a_i = (1 - \Gamma_i^u) * a_{i-1} + \Gamma_i^u * \tilde{a}_i$



LSTM:

$\Gamma_{\mathrm{i}}^{\mathrm{o}} = \sigma(W_{ox}x_i + W_{oa}a_{i-1} + b_o)$: output gate

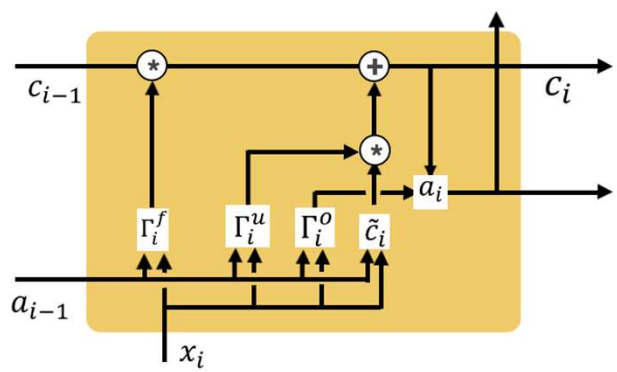
$\Gamma_{\mathrm{i}}^{\mathrm{u}} = \sigma(W_{ux}x_i + W_{ua}a_{i-1} + b_u)$: update gate

$\Gamma_{\mathrm{i}}^{\mathrm{f}} = \sigma\left(W_{fx}x_i + W_{fa}a_{i-1} + b_f\right)$: forget gate

$\tilde{c}_i = \tanh(W_{cx}x_i + W_{ca}a_{i-1} + b_c)$: candidate

$c_i = \Gamma_{\mathrm{i}}^{\mathrm{f}} * c_{i-1} + \Gamma_i^u * \tilde{c}_i$: cell state

$a_i = \Gamma_{\mathrm{i}}^{\mathrm{o}} * \tanh(c_i)$

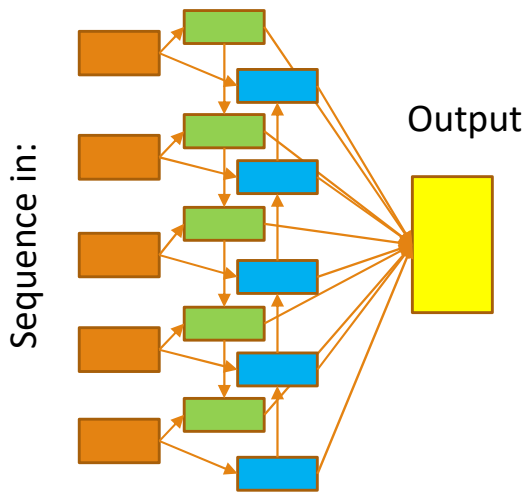


Given the lower number of parameters, GRU cells have a reputation for converging quicker. On the other hand, LSTM cells are a very established part of the literature, and are still the most widely used.

Bi-RNN & multilayer

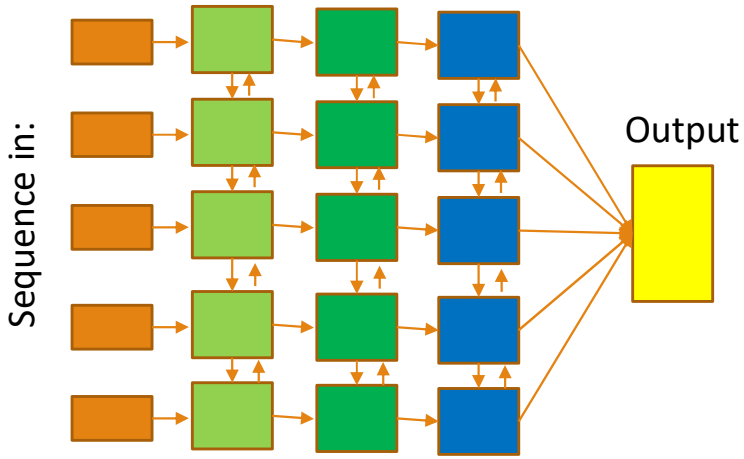
We still have two important extensions of RNN's:

Bi-RNN:



This solves the common situation where information should flow both ways along the input sequence. Notice that 'up' and 'down' don't share weights, and do not rely on each other's outputs.

Multilayer-RNN:



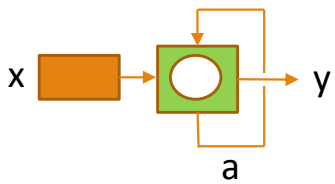
Just as you saw with deep CNN networks, different layers may work on different levels of abstraction.

Note on dropout: if you have multiple layers, pytorch has an inbuilt command for placing dropout between layers. **This does not work if there is only one layer.**
If you have only one layer and want to include dropout yourself, make sure to put it **after** the RNN layer.

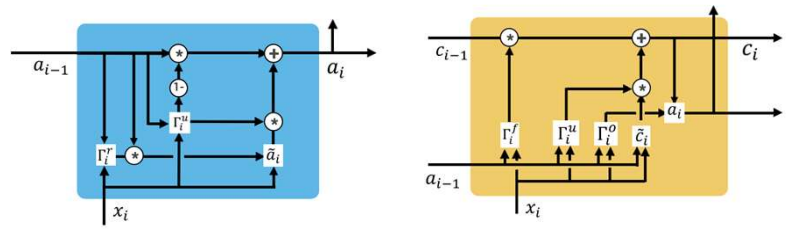
Discussion of RNN dropout: https://adriangcoder.medium.com/a-review-of-dropout-as-applied-to-rnns-72e79ecd5b7b

Summing up our progress:

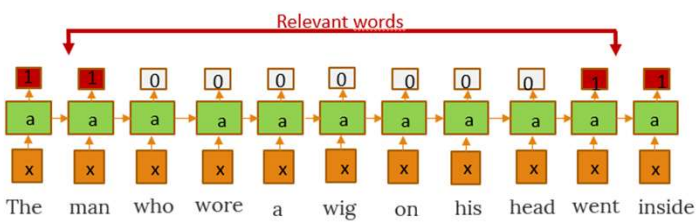
Reuse of small units:



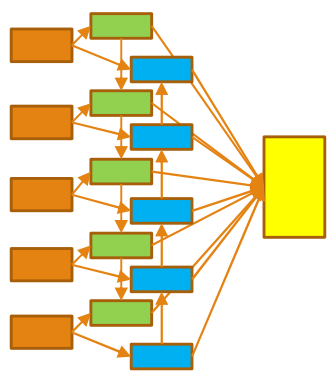
Gated structures:



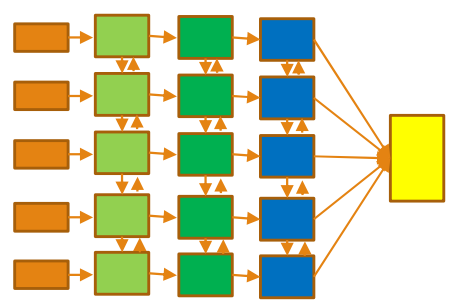
Long range relations:



Bidirectional layers:



Stacked layers:



Break

Quiz

Assume a single GRU cell with input size 1 and hidden dimension (size of a) 1.
Let

$$b_r = b_u = b_a = 0$$
$$W_{rx} = W_{ux} = W_{ax} = W_{aa} = 1$$
$$W_{ra} = 10, W_{ua} = 0$$

If $a_{i-1} = 3, x_i = -3$, then what are $\tilde{a}_i$ and a_i?

1. $\tilde{a}_i \approx 0$ and $a_i \approx 1$
2. $\tilde{a}_i \approx 0$ and $a_i \approx 3$
3. $\tilde{a}_i \approx 3$ and $a_i \approx 1$
4. $\tilde{a}_i \approx 0$ and $a_i \approx 0$

GRU equations:

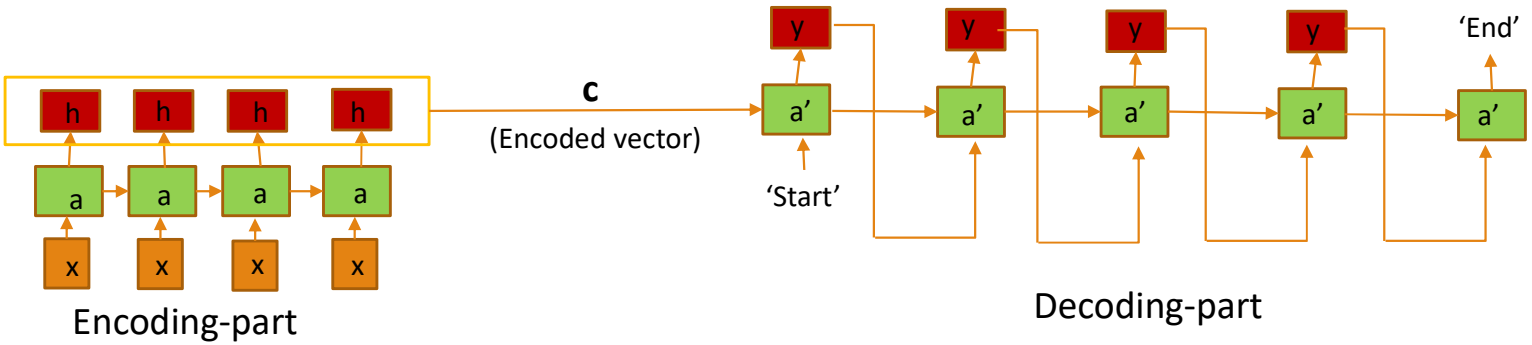
$\Gamma_{\mathrm{i}}^{\mathrm{r}} = \sigma(W_{rx}x_i + W_{ra}a_{i-1} + b_r)$: reset gate
$\Gamma_{\mathrm{i}}^{\mathrm{u}} = \sigma(W_{ux}x_i + W_{ua}a_{i-1} + b_u)$: update gate
$\tilde{a}_i = \tanh(W_{ax}x_i + W_{aa}(\Gamma_i^r * a_{i-1}) + b_a)$: candidate
$a_i = (1 - \Gamma_i^u) * a_{i-1} + \Gamma_i^u * \tilde{a}_i$

$\Gamma_{\mathrm{i}}^{\mathrm{r}} = \sigma(-3 + 10 * 3) = \sigma(27) \approx 1$
$\Gamma_{\mathrm{i}}^{\mathrm{u}} = \sigma(-3) \approx 0$
$\tilde{a}_i \approx \tanh(-3 + 1 * 3) \approx 0$
$a_i \approx (1 - 0) * 3 + 0 * 0 = 3$

$$(\tilde{a}_i = 0, a_i = 2.86)$$

Architecture: Encoder-decoder

A common occurrence, in particular when working with text, is to have inputs and outputs of different lengths and where all parts of the input can influence any given portion of output.
A common approach then is to 'encode' the entire input sequence into an abstract vector, and then pass that vector to a 'decoder' which outputs the correct sequence:

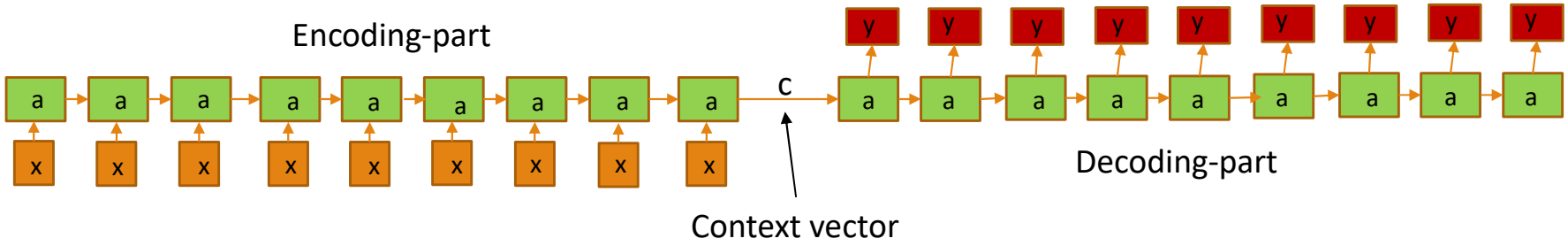


There are then different approaches for how the $h_1, h_2, \ldots, h_T$ are condensed into c. A very common choice is to simply let $c = h_T$, and then use a sufficiently high hidden dimension for the encoder cells.

During training, the encoder functions regularly, while the decoder may be subject to '*teacher forcing*'. This means that instead of forwarding '$y_{pred,i-1}$' (which might be incorrect), instead the correct y_{i-1} from the training set is forwarded. So the decoder can practice all decoding steps simultaneously.

New concept: Attention

Consider an encoder-decoder architecture, with possibly long sequences:



For instance, we could do sentence-to-sentence translation:
Danish:
Der var så dejligt ude på landet; det var sommer, kornet stod gult, havren grøn, høet var rejst i stakke nede i de grønne enge, og der gik storken på sine lange, røde ben og snakkede ægyptisk, for det sprog havde han lært af sin moder.

English:
It was so lovely in the country; it was summer, the wheat was golden, the oats were green, the hay was stacked in the green meadows, and there, the stork was strutting about, on his long, red legs, speaking egyptian, a language his mother had taught him.

New concept: Attention

Danish:
Der var så dejligt ude på landet; det var sommer, kornet stod gult, havren grøn, høet var rejst i stakke nede i de grønne enge, og der gik storken på sine lange, røde ben og snakkede ægyptisk, for det sprog havde han lært af sin moder.

English:
It was so lovely in the country; it was summer, the wheat was golden, the oats were green, the hay was stacked in the green meadows, and there, the stork was strutting about, on his long, red legs, speaking egyptian, because his mother had taught him that.

If a human had to do this translation (as, indeed, this particular human did), s/he would not focus on the entire Danish sentence at once, but instead bit by bit:

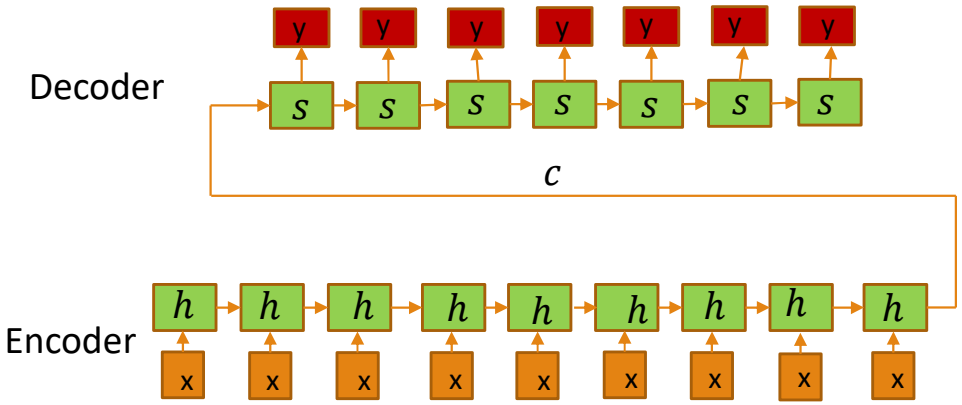
Der var så dejligt ude på landet -> It was so lovely in the country

(etc.)

It turns out that giving neural networks the same capability is a *wonderful* idea.

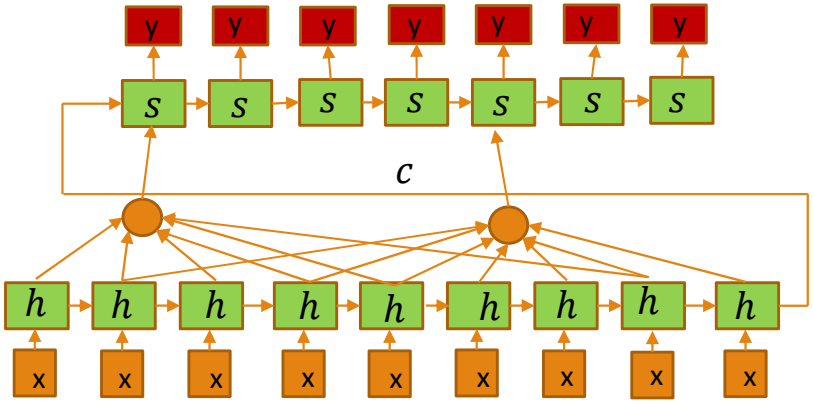
New concept: Attention

Consider the encoder-decoder from before:



In this setup, the encoder has to include context for the entire input sentence into just one context vector. For long inputs, that can be problematic.

So we could give some more direct information to each output node:



In this setup, the decoder nodes each receive both context from the previous node, as well as a weighted average of the hidden states from the encoder cells.

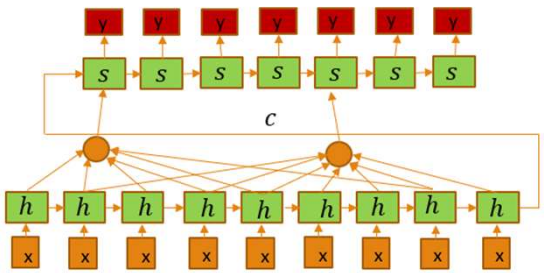
New concept: Attention

Let s_i be the activations in the *i*'th cell in the decoder layer. Let c_i be the additional input vector ('context' vector) to cell *i*, y_i the output from cell *i*, and let h_j be the activations in encoder cell *j*:

Decoder activation $$s_i = f(c_i, y_{i-1}, s_{i-1})$$

Decoder context $$c_i = \sum_j \alpha_{ij}\, h_j$$

Attention weights $$\alpha_{ij} = \frac{e^{a_{ij}}}{\sum_k e^{a_{ik}}} \text{ (softmax over encoders)}$$



The question then becomes how the a_{ij} are calculated?

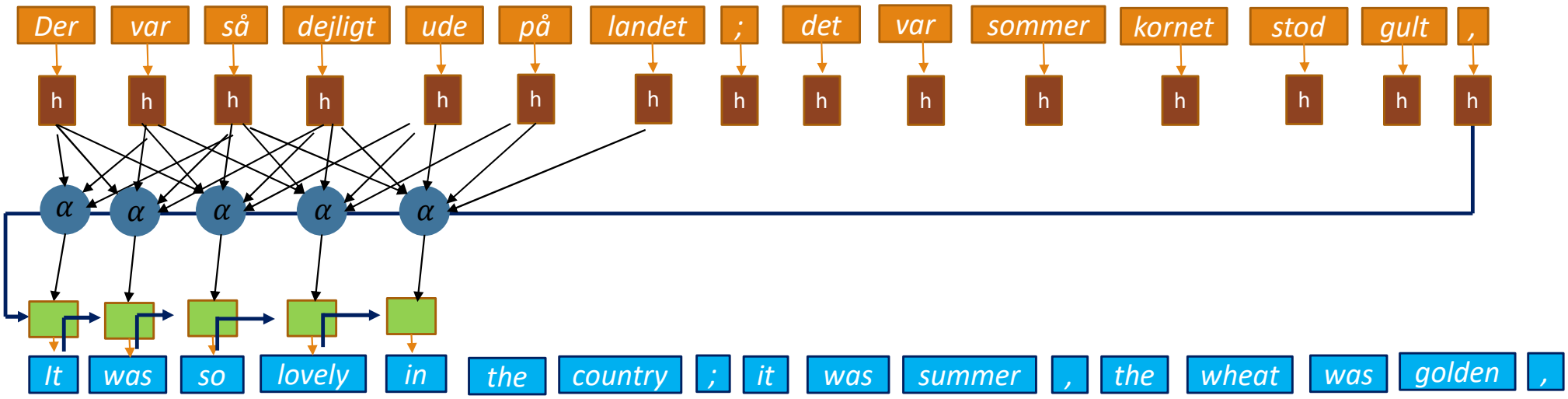
- The most common approach is 'dot product attention': $a_{ij} = s_{i-1} \cdot h_j = s_{i-1}^T h_j$
- Much more elaborate approaches are also possible. In the original paper, a_{ij} is a small feedforward neural network in itself, taking as input s_{i-1}, h_j and outputting a_{ij}: $a_{ij} = v^T * \tanh\left(W * \begin{pmatrix} h_j \\ s_{i-1} \end{pmatrix}\right)$.

The logic in both is that given the state of the previous decoder cell, it is possible to determine which encoder outputs are most relevant to the current task.

You may occasinally see people talk of a_{ij} in terms of the the *alignment function*: $a_{ij} = \text{align}(s_{i-1}, h_j)$.

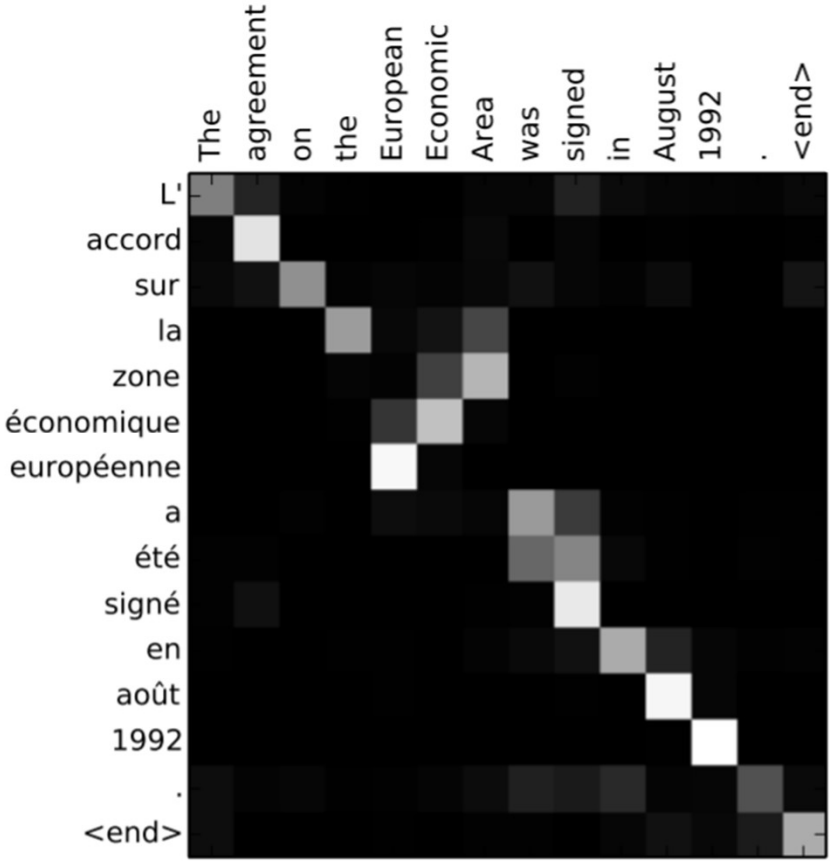
Source: https://arxiv.org/abs/1409.0473

So, going back to our translation example:



Here, α denotes the 'alignment function'.
Note that often, the decoder unit would get both s_{i-1} and y_{i-1} as input.

Example: sentence translation

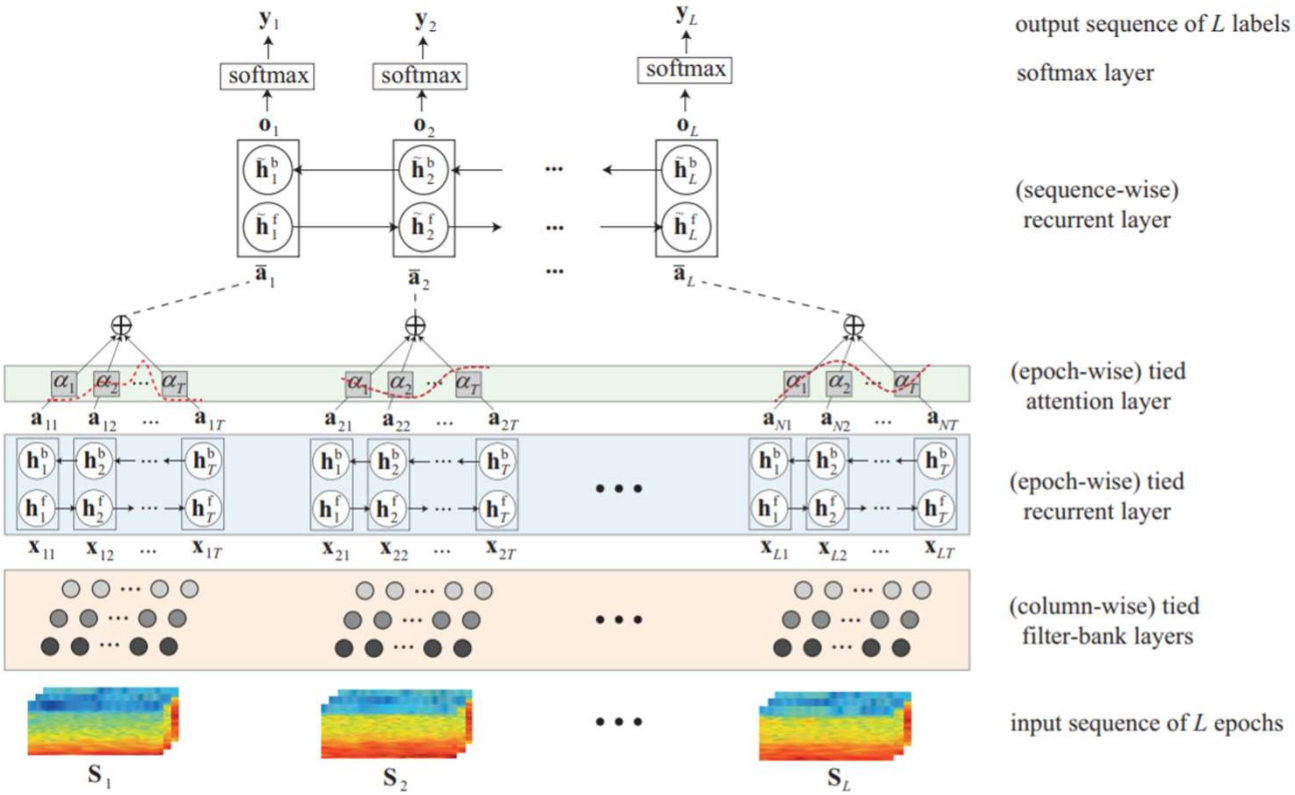


To the left we see an example from the original paper, where an english sentence is translated into a french sentence.
For each word in the output (each row) is shown which input words (columns) were being attended to the most (α_{ij}).

In general, attention is considered a very powerful concept, which has seen wide adoption in language processing.

One downside which can be relevant in certain contexts is that the a_{ij} have to be calculated on run-time, and it scales as the number of input words TIMES the number of outputs words. In our Danish-> English example, that means 2162 a_{ij}-values.
(in short, attention can be slow)

Example: SeqSleepNet



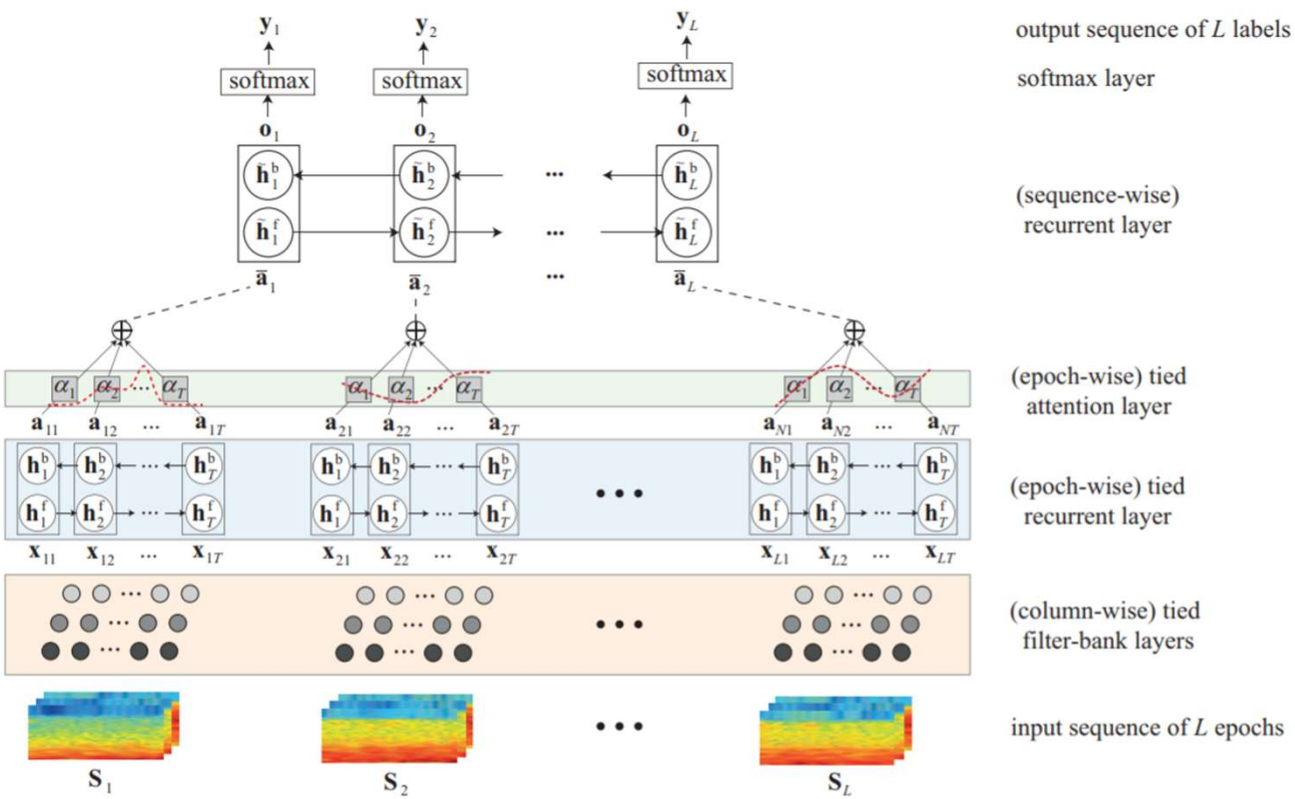
Presented in 2019, SeqSleepNet was among state-of-the-art in automatic sleep scoring (since overtaken by larger models).

It takes as input sequences of L spectrograms. Each spectrogram consists of a sequence of T spectres.

After performing dimension reduction of the spectrograms, they are each fed into a bidirectional layer of GRU cells. The output of these are then fed to an attention layer (more on that next slide). The attention-filtered outputs are then formed into a new sequence, which is again fed into a bidirectional GRU sequence.
The output of that is used to create L outputs, one label for each spectrogram.

Source: https://github.com/pquochuy/SeqSleepNet

Example: SeqSleepNet



In SeqSleepNet, attention is implemented as:

$$a_{ij} = v^T * \tanh\left(W * \begin{pmatrix} h_j^b \\ h_j^f \end{pmatrix} + b\right)$$

This looks very similar to the original implementation, but the interpretation is now somewhat different.

Since the attention weights do not depend on the previous output, the alignment function becomes a 'library' of relevant activation patterns. The actual activations are compared to the library, and the most important ones are condensed into the output.

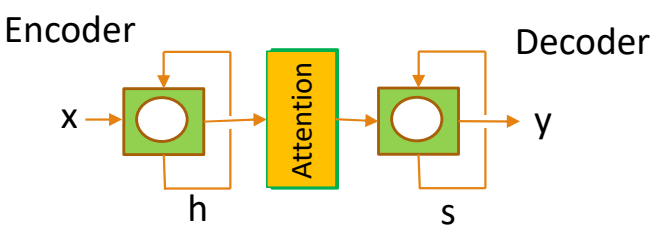
Source: https://github.com/pquochuy/SeqSleepNet (you have to check the source code, the paper has an error in the attention description)

Break

Quiz!

Below is a description of an attention implementation.

1. Cell activations: $s_i = f(c_i, y_{i-1}, s_{i-1})$
2. Alignment function: $a_{ij} = s_{i-1} * h_j + y_{i-1}$
3. Attention weights: $\alpha_{ij} = \frac{e^{a_{ij}}}{\sum_k e^{a_{ik}}}$
4. Attention output: $c_i = \sum_j \alpha_{ij} s_j$



Or is it…?

Pick 1 option:

1. Line 1 is weird
2. Line 2 is weird
3. Line 3 is weird
4. Line 4 is weird
5. Everything is weird
6. Everything is good

The output of the attention layer should always be a linear combination of the outputs from the previous layer (h_j).

New concept:
Transformers

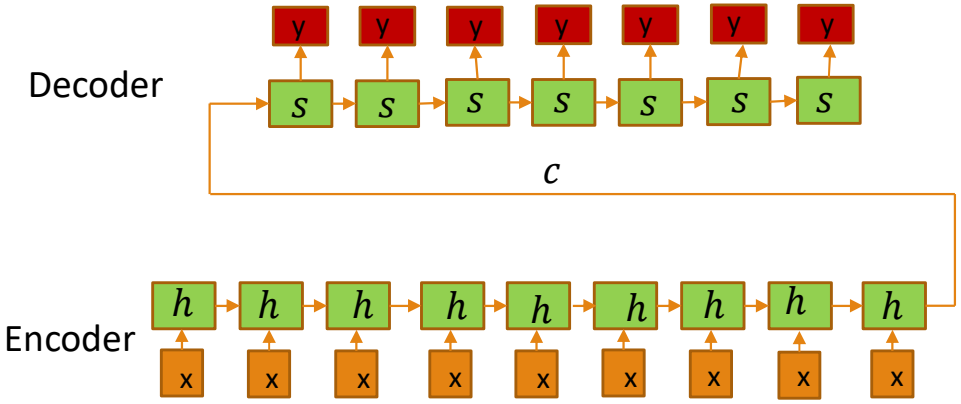
Motivation

RNN architectures have one inherent drawback:

No matter how many processors / graphics cards you have available, the individual steps have to be done sequentially – no parallization possible.

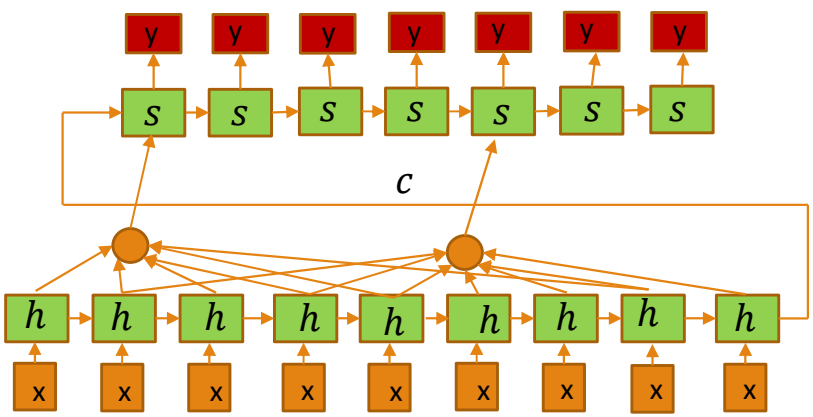
This can be an issue in, for instance, real time text processing. A sentence of 20 words still requires 20 consecutive evaluations.

Note that we are here strictly talking about the deployed, fully trained implementation.



Motivation

On the other hand, while the attention mechanism comes with a high complexity (#input * #outputs) it is fully parallelizable. The question then is, how much work could we offload into the attention mechanism, away from the sequential processing portion?



To do that successfully, it's important to generalize what is going on in attention:

Given a problem description, s_{i-1}, the system compares that with encoded information, $\{h_j\}$ and generates an output, $\sum \alpha_{ij} h_j$.
For a more general system, we can say that when presented with a **Query** (s_{i-1}), this is compared with the contents of a database (described by **Keys**, $\{h_j\}$), and a **Value** is returned ($\sum \alpha_{ij} h_j$). This is, in general terms, how a search engine works.

The Query, Key, Value (Q,K,V) concept is central in the following.

Source: 'Attention is all you need' (https://arxiv.org/abs/1706.03762)

Self attention

Previously, we saw attention defined as:

$$c_i = \sum_j \alpha_{ij}\, h_j$$
$$\alpha_{ij} = \mathrm{softmax}(a_{ij})$$

Here, the a_{ij} defines relevance or similarity between h_j and the decoding task (s_i). The output is then a linear combination of h_j weighted towards relevance.

We can generalize this further, and instead of using h_j we use linearly transformed versions: $\mathrm{h} \rightarrow Wh_j$. This is the essence of '**self attention**':

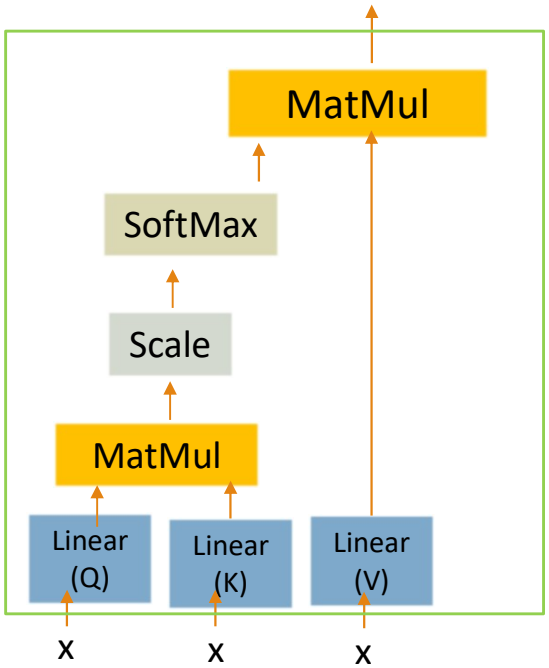
$$\mathrm{z} = \mathrm{softmax}\left(\frac{(W_Q x)^T (W_K x)}{\sqrt{d_k}}\right) W_V x$$

Here, x is the input sequence of dimension $d_k \times n$ and W_Q, W_K are matrices of size $d_{model} \times d_k$ and W_V of size $d_v \times d_k$.
Think of it as 'scaled dot product attention', but the keys and queries are different linear functions of the input.

Source: 'Attention is all you need' (https://arxiv.org/abs/1706.03762)

Multi-head attention

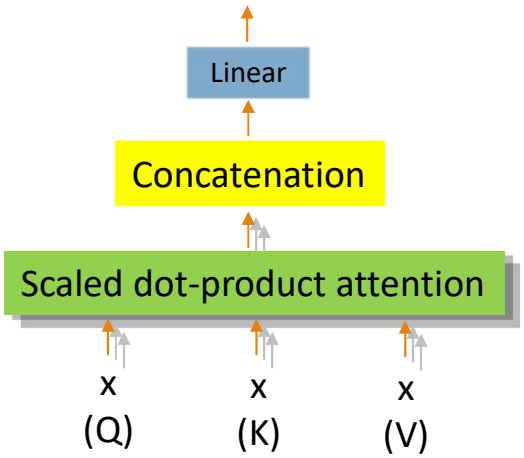
Pictorially, self attention works like this:



A.k.a 'scaled dot-product attention'

Self-attention can be extended to *multi-head attention* by doing everything *h* times:

Multi-head attention:



Meaning different weights for each head:

$$z = \text{softmax}\left(\frac{(W_{Q,i}x)^T(W_{K,i}x)}{\sqrt{d_k}}\right)W_{V,i}x\ , 1 \leq i \leq h$$

Transformers:

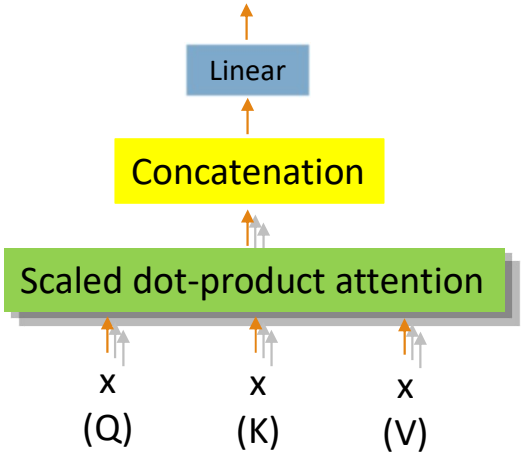
Multi-head attention was introduced in the pivotal paper '*Attention is all you need*', by Vaswani et al 2017.
In that paper, it was part of a larger model, called the **Transformer**.

Since then, many state-of-the-art models in various domains have been presented, all based on portions of the transformer, primarily the multi-head attention part.

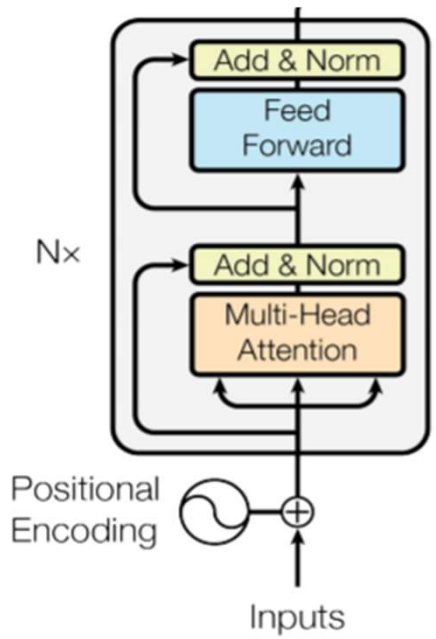
Many of these models are called transformers, or transformer-based, even though they may only have this one ingredient in common.

For the sake of your cultural upbringing, we will now talk through the full transformer architecture, but keep in mind that most 'transformer' models out there do not actually match this description.

Multi-head attention:



Transformers:



Besides multi-head attention, we also need:

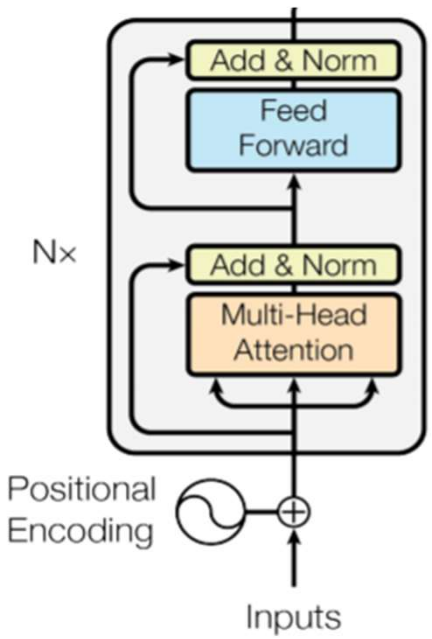
Feed forward: $W_2 \cdot ReLu(W_1 x + b_1) + b_2$

Add & norm: LayerNormalization(x+sublayer(x)). To make this feasible, each head has output dimension $d_v = \frac{d_k}{h}$. The 'x' is a *residual* connection, such that the sublayer only has to output the change.

Positional Encoding: The self-attention erases all information about the order of the input sequence. The positional encoding seeks to remedy this.

Source: 'Attention is all you need' (https://arxiv.org/abs/1706.03762)

Positional Encoding:



In the original paper, to the inputs were added a positional encoding tensor:

$$\mathrm{X} \rightarrow X + PE$$

Which was defined as:

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

Where ‘pos’ runs along the sequence, and ‘i’ runs along input dimensions. Usually, the encoding is only added to the first few (4) dimensions.

Positional Encoding:

Note that positional encoding schemes come in many different flavours, and I do not think the jury is settled on what the best way to do it is.

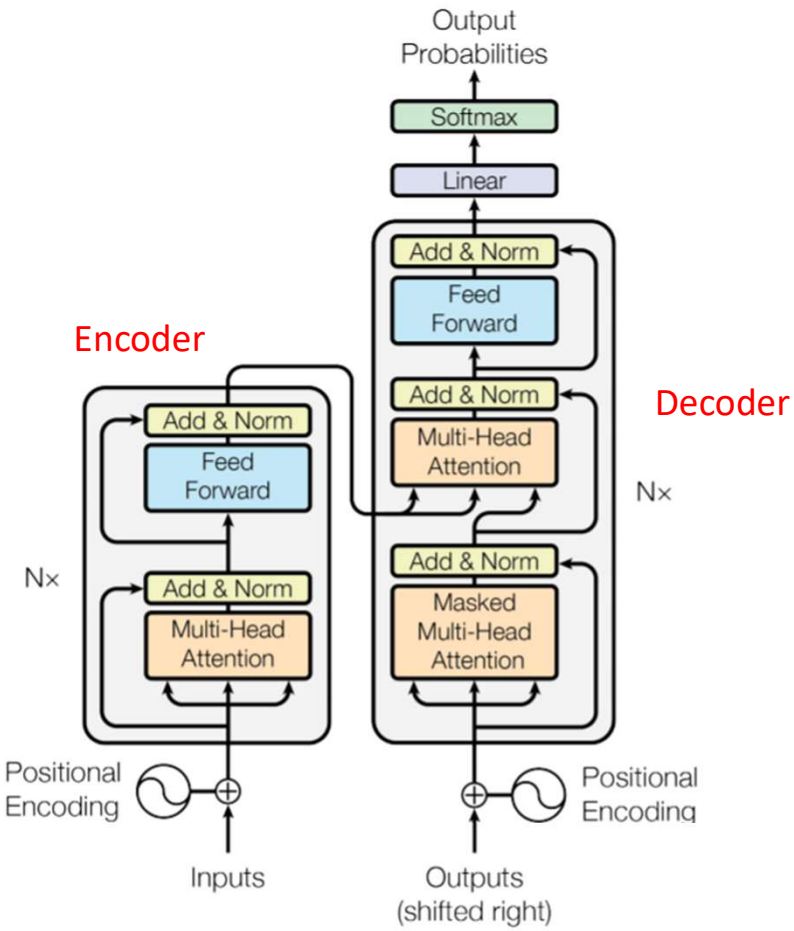
The sin & cos approach is an example of so-called 'absolute encoding', since the values of the encoding at a specific position does not depend on input length. In 'relative encoding', the encoding depends on the relative position (and so relative distances may be easier to calculate).

In some implementations, the positional encoding is learned by the model, and there is some valid discussions about whether additive encoding is really better than concatenating. There is a reasonable discussion of the topic in:
https://towardsdatascience.com/master-positional-encoding-part-i-63c05d90a0c3
https://towardsdatascience.com/master-positional-encoding-part-ii-1cfc4d3e7375

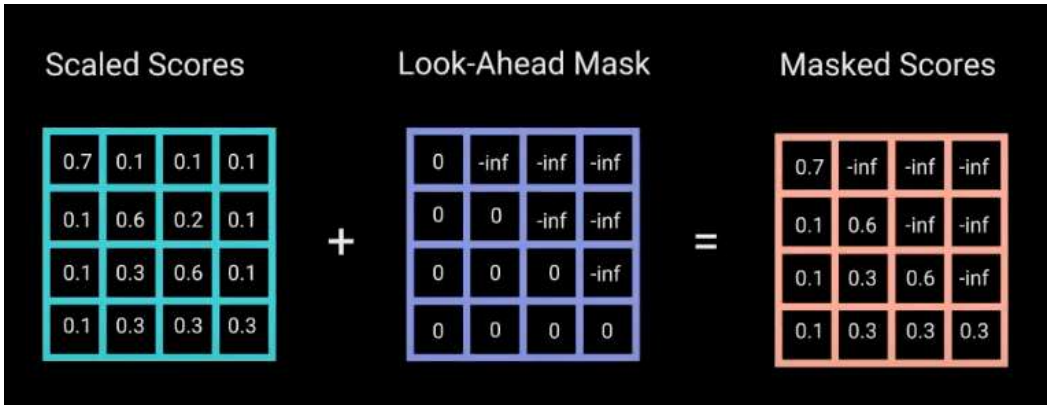
And 'Rethinking Positional Encoding in Language Pre-training', https://arxiv.org/abs/2006.15595

And this textbook has a good chapter on relative positional encodings:
Transformer Positional Encoding & Embeddings

Transformers: encoding & decoding

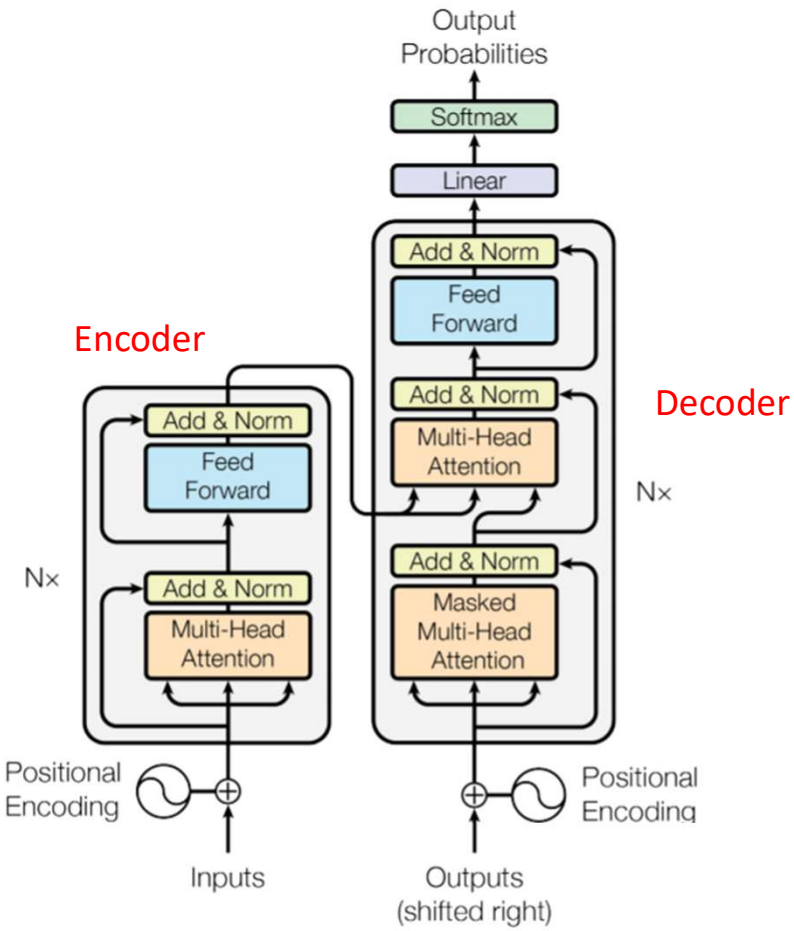


In the encoder and decoder setup, the **encoder** is trained like we have described so far.
The **decoder** has a few extra details. Like with the RNN encoder-decoder, a form of 'teacher forcing' is used in training the decoder. This is implemented by 'masking' the multihead attention, such that when processing the n'th input in the sequence, only the self attention until that point is visible. This is done by setting the 'masked' attention weights to -Inf:



Source: 'Attention is all you need' (https://arxiv.org/abs/1706.03762)

Transformers: encoding & decoding



The intuition is that the encoder describes the structure of the input, while the decoder first determines the structure of the already generated sequence, and then compares to the full structure of the encoded input. Based on that, a suitable next output is generated.
We may think that in the second multi-head attention, the values are from the output space, but we pick among them based on information from the input space.

Due to the masking, the training can be run in parallel for each output (like with teacher forcing).

During prediction, there is no teacher forcing, so instead the decoder is sequentially asked to predict each token in the output, being fed the previous outputs as input (just like RNN's).

Source: 'Attention is all you need' (https://arxiv.org/abs/1706.03762)

Transformers

In the seminal paper, the self-attention cells had h=8 heads each, and encoder and decoder were further stacked in N=6 layers.

Transformers have reputations of training significantly faster than, for instance LSTM's, in large part due to the parallelization. However, they have also occasionally been shown to require larger amounts of data to reach SOTA performance.

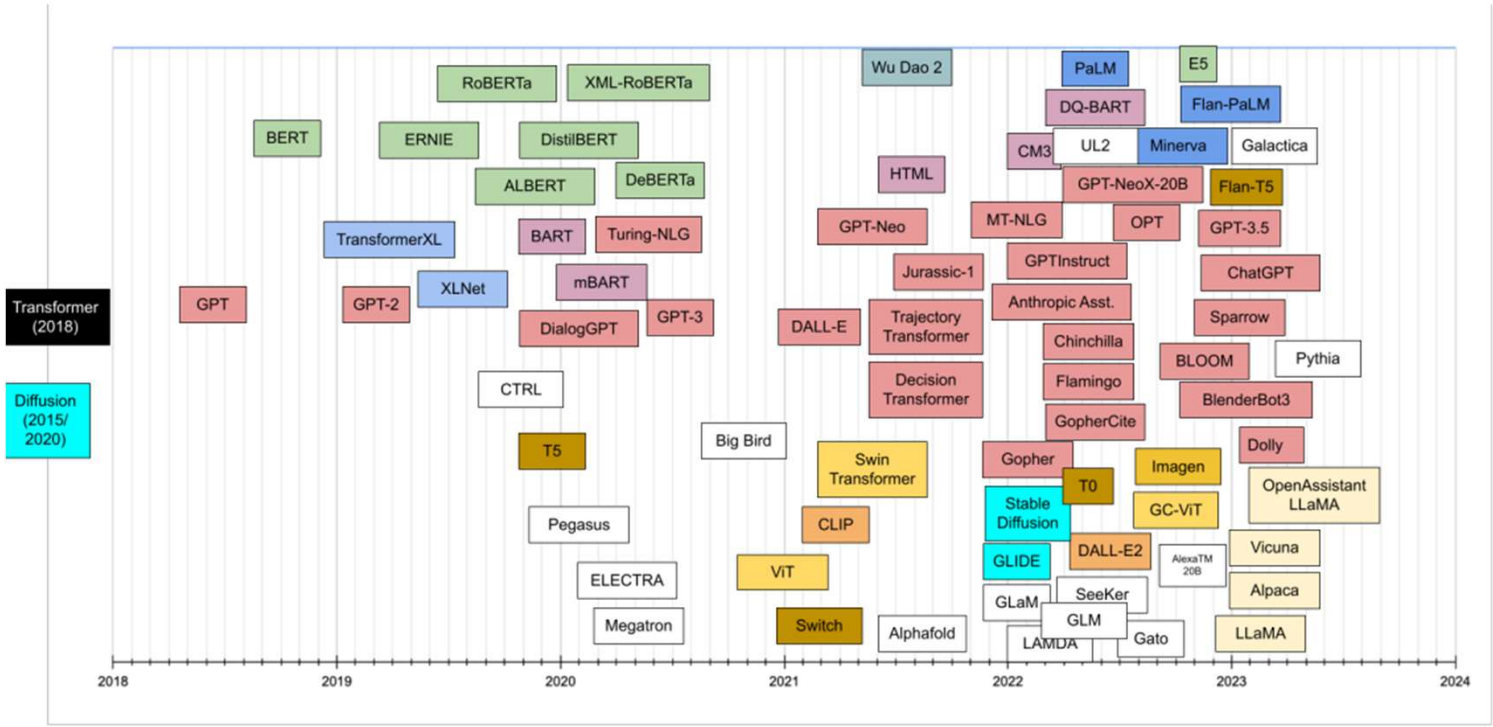
Note that each head in the multihead attention unit projects into a $d_v = d_{model}/h$ dimensional space, where d_{model} is the input dimension. This means that increasing h does not increase the number of parameters, and it makes the residual connections simpler to implement.

Finally: self attention consists of projections into sub spaces. That means it is particularly well-suited for high-dimensional input. Not everybody gets that.

Source: https://arxiv.org/abs/2105.11043, https://ieeexplore.ieee.org/document/9003750

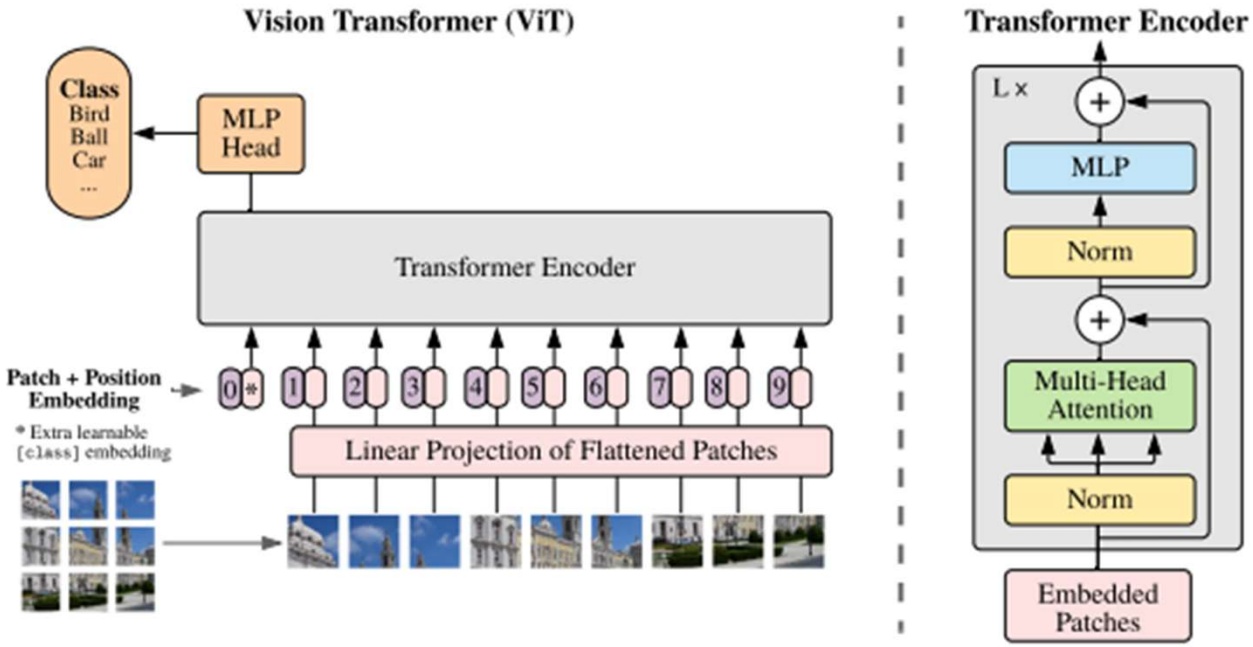
Other flavours

Since 2018, a lot of different models have been presented, all using parts of the transformer:



https://amatriain.net/blog/transformer-models-an-introduction-and-catalog-2d1e9039f376/

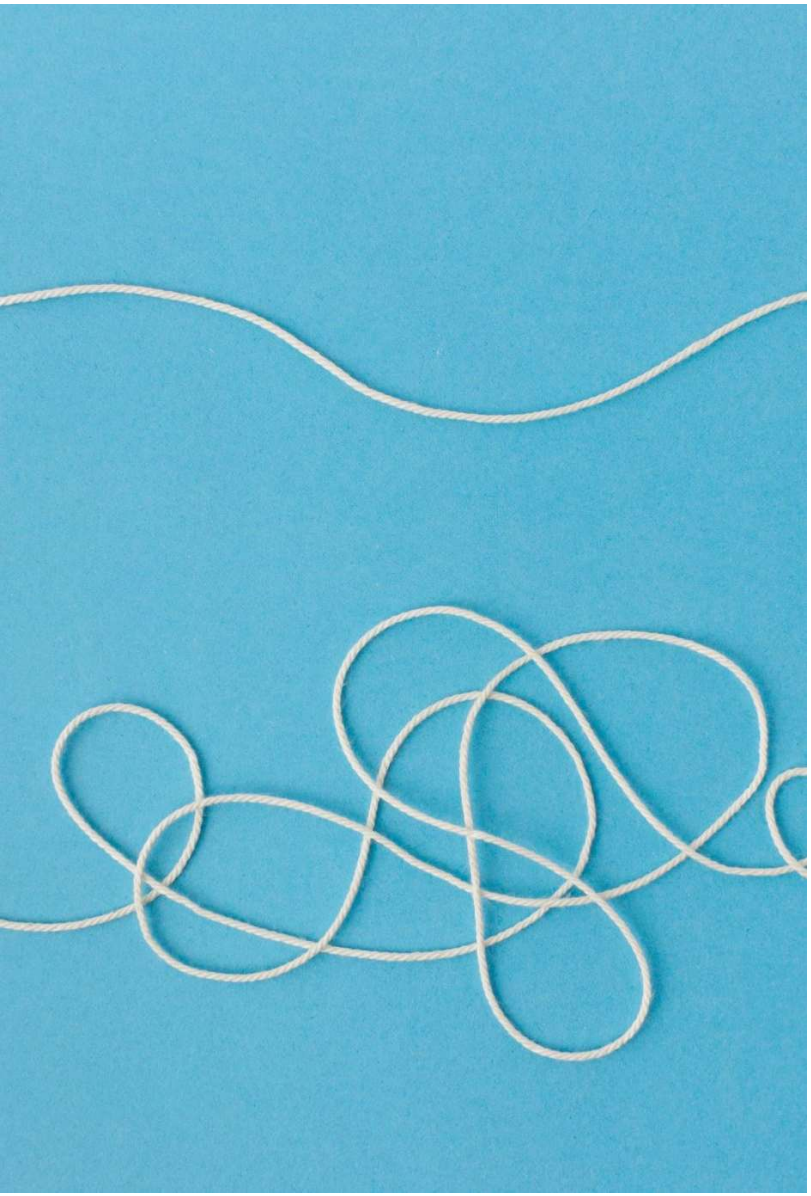
Special mention: Vision transformer



[2010.11929v2] An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale

New developments

THE TRANSFORMER COUNTER-CULTURE



Motivation: really long inputs

Self-attention scales as n^2, with n being the number of input tokens.

The input sizes to the newest LLM's is ever increasing (LLAMA 3.1 accepts up to 128k tokens, Gemini 1.5 up to 1 million!)

The old LSTM & GRU models scale linearly with n.

Could old-school models stage a comeback?

State space models

Consider a simple set of differential equations:

$$\dot{x}[t] = Ax[t] + Bu[t]$$

$$y[t] = Cx[t] + Du[t]$$

We can easily discretize this, by picking a stepsize Δ and performing some calculus to generate 'discretized' matrices:

$$x[t] = \bar{A}x[t-1] + \bar{B}u[t]$$

$$y[t] = \bar{C}x[t]$$

(convention is to set D=0, because D can be modeled as a skip connection separately)

This is super simple. We have removed gates, everything is linear again, and super convenient.

Write it out:

$x[0] = \bar{B}u[0]$

$x[1] = \bar{A}x[0] + \bar{B}u[1] = \bar{A}\bar{B}u[0] + \bar{B}u[1]$

$x[2] = \bar{A}x[1] + \bar{B}u[2] = \bar{A}^2\bar{B}u[0] + \bar{A}\bar{B}u[1] + \bar{B}u[2]$

$x[3] = \bar{A}^3\bar{B}u[0] + \cdots$

This is enough for us to investigate the pattern for y:

$y[0] = \bar{C}x[0] = \bar{C}\bar{B}u[0]$

$y[1] = \bar{C}x[1] = \bar{C}\bar{A}\bar{B}u[0] + \bar{C}\bar{B}u[1]$

$y[2] = \bar{C}x[2] = \bar{C}\bar{A}^2\bar{B}u[0] + \bar{C}\bar{A}\bar{B}u[1] + \bar{C}\bar{B}u[2]$

It turns out that we can describe y[k] as a convolution of **u** using a somewhat complicated kernel:

$$y[k] = \boldsymbol{K}_k * \boldsymbol{u}, \boldsymbol{K_k} = \left[\bar{C}\bar{B}, \bar{C}\bar{A}\bar{B}, \dots, \bar{C}\bar{A}^{\mathrm{k}}\bar{B}\right]$$

The kernel can be calculated using an FFT, and this formulation speeds everything up *a lot*.

This means that the same model can be used with unbounded context, while also parallelized during training!

Variations

The first important paper discussed **S**tructured **S**tate **S**pace **S**equence Models (‘**S4**’ models)

(‘structure’ refers to restrictions on A, speeding up computations)

Then there were a lot of different versions: S4 v2, DSS, S4D (and probably others), leading to:

S5: Simplified State Space Layers for Sequence Modeling

Which had a better way to deal with multidimensional input.

S4: [2111.00396] Efficiently Modeling Long Sequences with Structured State Spaces

Mamba!

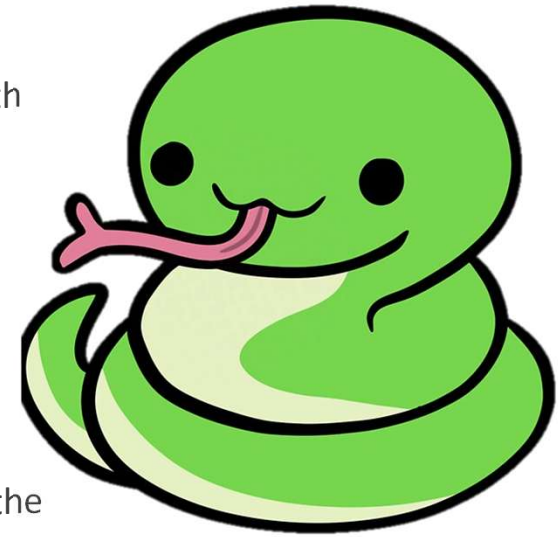
Of course, this is all a little silly, because the whole speedup comes because we have removed the gates. Which leads us to the possibly most popular model yet:

S6: 'Mamba: Linear-Time Sequence Modeling with Selective State Spaces' (S4 with 'selection' and 'scanning' = S6)

Change the formulation again:

$$x[t] = Ax[t-1] + B(u)u[t]$$

$$y[t] = C(u)x[t]$$



This means we can't do the convolution trick anymore, but they retain speed by employing the 'parallel scan' algorithm and being really smart about how to use the GPU memory.

Mamba ‘selection’

From the Mamba paper (beware that u is now called x):

Algorithm 1 SSM (S4)

Input: $x : (\mathsf{B}, \mathsf{L}, \mathsf{D})$
Output: $y : (\mathsf{B}, \mathsf{L}, \mathsf{D})$

$\boldsymbol{A} : (\mathsf{D}, \mathsf{N}) \leftarrow$ Parameter
▷ Represents structured $N \times N$ matrix
$\boldsymbol{B} : (\mathsf{D}, \mathsf{N}) \leftarrow$ Parameter
$\boldsymbol{C} : (\mathsf{D}, \mathsf{N}) \leftarrow$ Parameter
$\Delta : (\mathsf{D}) \leftarrow \tau_\Delta(\text{Parameter})$
$\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}} : (\mathsf{D}, \mathsf{N}) \leftarrow \text{discretize}(\Delta, \boldsymbol{A}, \boldsymbol{B})$
$y \leftarrow \text{SSM}(\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}}, \boldsymbol{C})(x)$
▷ Time-invariant: recurrence or convolution
return y

Algorithm 2 SSM + Selection (S6)

Input: $x : (\mathsf{B}, \mathsf{L}, \mathsf{D})$
Output: $y : (\mathsf{B}, \mathsf{L}, \mathsf{D})$

$\boldsymbol{A} : (\mathsf{D}, \mathsf{N}) \leftarrow$ Parameter
▷ Represents structured $N \times N$ matrix
$\boldsymbol{B} : (\mathsf{B}, \mathsf{L}, \mathsf{N}) \leftarrow s_B(x)$
$\boldsymbol{C} : (\mathsf{B}, \mathsf{L}, \mathsf{N}) \leftarrow s_C(x)$
$\Delta : (\mathsf{B}, \mathsf{L}, \mathsf{D}) \leftarrow \tau_\Delta(\text{Parameter} + s_\Delta(x))$
$\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}} : (\mathsf{B}, \mathsf{L}, \mathsf{D}, \mathsf{N}) \leftarrow \text{discretize}(\Delta, \boldsymbol{A}, \boldsymbol{B})$
$y \leftarrow \text{SSM}(\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}}, \boldsymbol{C})(x)$
▷ Time-varying: recurrence (*scan*) only
return y

From the paper:

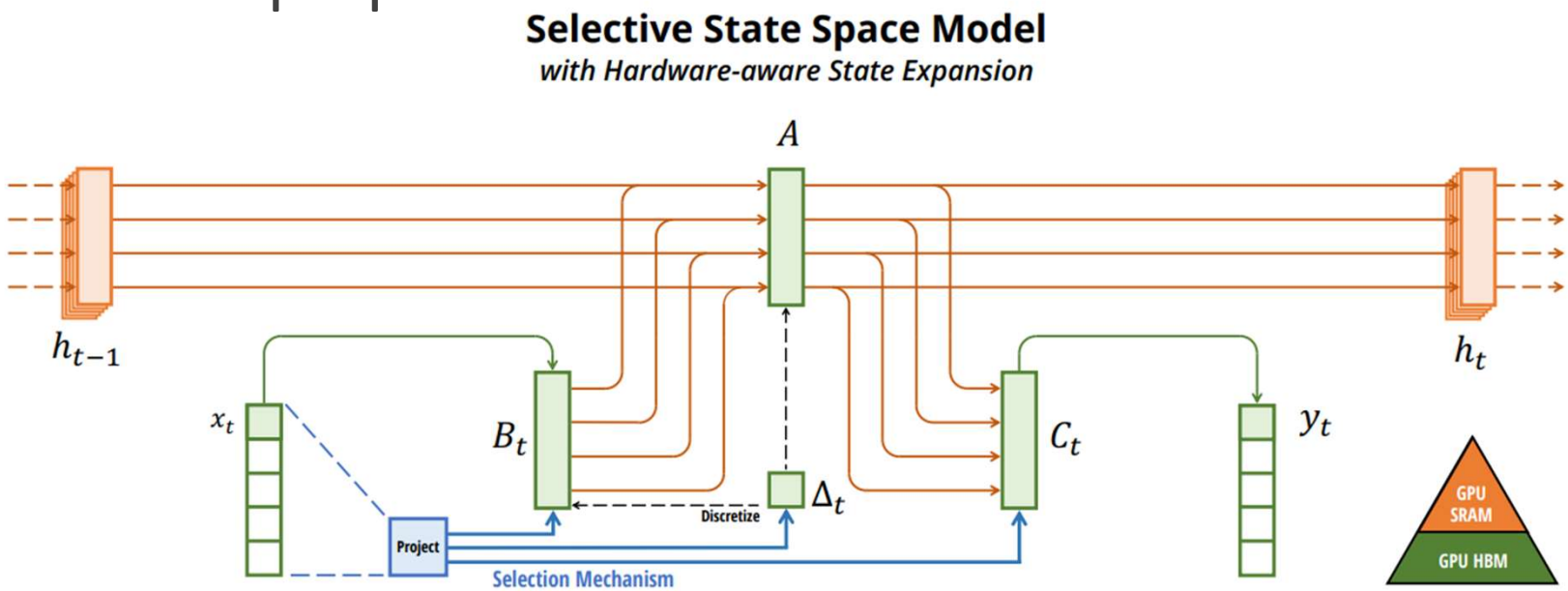


Figure 1: (**Overview.**) Structured SSMs independently map each channel (e.g. $D = 5$) of an input x to output y through a higher dimensional latent state h (e.g. $N = 4$). Prior SSMs avoid materializing this large effective state (DN, times batch size B and sequence length L) through clever alternate computation paths requiring time-invariance: the $(\Delta, \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})$ parameters are constant across time. Our selection mechanism adds back input-dependent dynamics, which also requires a careful hardware-aware algorithm to only materialize the expanded states in more efficient levels of the GPU memory hierarchy.

Take-aways:

A lot of thought goes into how to efficiently use the hardware.

At this time, SSM's (Mamba) seem to beat transformers for tasks requiring a very long context. And they seem to be better with signals that have a continuous origin.

I have not understood why the researchers don't abandon the 'discretization' step.

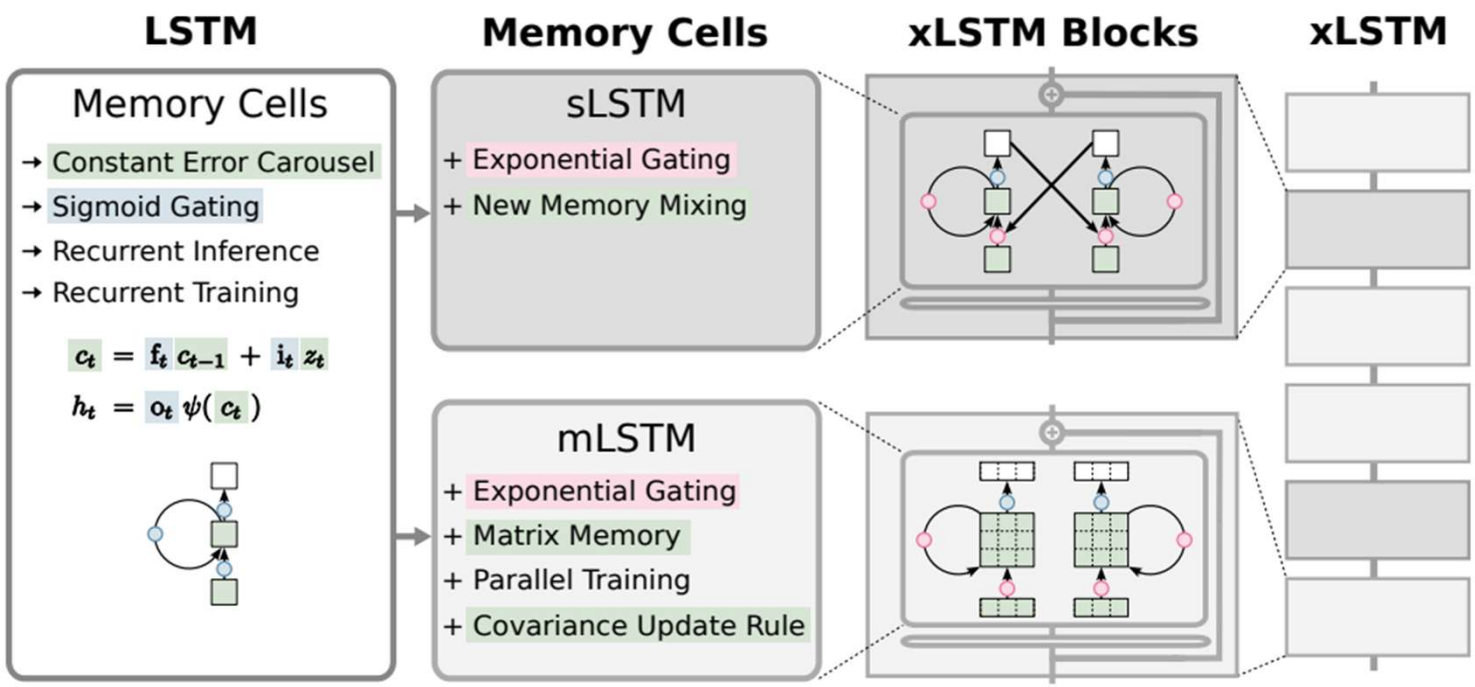
Some recommended reading / viewing:

Video: State Space Models (S4, S5, S6/Mamba) Explained

Blog: Introduction to State Space Models (SSM)

Paper: State Space Models as Foundation Models: A Control Theoretic Overview

xLSTM: ‘extended’ LSTM



Sepp Hochreiter (described LSTM in master’s thesis)

sLSTM block:

$$c_t = \mathrm{f}_t \, c_{t-1} + \mathrm{i}_t \, z_t \qquad \text{cell state} \qquad (8)$$

$$n_t = \mathrm{f}_t \, n_{t-1} + \mathrm{i}_t \qquad \text{normalizer state} \qquad (9)$$

$$h_t = \mathrm{o}_t \, \tilde{h}_t \,, \qquad \tilde{h}_t = c_t \,/\, n_t \qquad \text{hidden state} \qquad (10)$$

$$z_t = \varphi\left(\tilde{z}_t\right) \,, \qquad \tilde{z}_t = \boldsymbol{w}_z^\top \boldsymbol{x}_t + r_z h_{t-1} + b_z \qquad \text{cell input} \qquad (11)$$

$$\mathrm{i}_t = \exp\left(\tilde{\mathrm{i}}_t\right) \,, \qquad \tilde{\mathrm{i}}_t = \boldsymbol{w}_\mathrm{i}^\top \boldsymbol{x}_t + r_\mathrm{i} \, h_{t-1} + b_\mathrm{i} \qquad \text{input gate} \qquad (12)$$

$$\mathrm{f}_t = \sigma\left(\tilde{\mathrm{f}}_t\right) \text{ OR } \exp\left(\tilde{\mathrm{f}}_t\right) \,, \qquad \tilde{\mathrm{f}}_t = \boldsymbol{w}_\mathrm{f}^\top \boldsymbol{x}_t + r_\mathrm{f} \, h_{t-1} + b_\mathrm{f} \qquad \text{forget gate} \qquad (13)$$

$$\mathrm{o}_t = \sigma\left(\tilde{\mathrm{o}}_t\right) \,, \qquad \tilde{\mathrm{o}}_t = \boldsymbol{w}_\mathrm{o}^\top \boldsymbol{x}_t + r_\mathrm{o} \, h_{t-1} + b_\mathrm{o} \qquad \text{output gate} \qquad (14)$$

We broadcast the original LSTM gating techniques, i.e., input- and/or hidden-dependent gating plus bias term, to the new architectures. Exponential activation functions can lead to large values that cause overflows. Therefore, we stabilize gates with an additional state m_t (Milakov & Gimelshein, 2018):

$$m_t = \max\left(\log(\mathrm{f}_t) + m_{t-1}, \log(\mathrm{i}_t)\right) \qquad \text{stabilizer state} \qquad (15)$$

$$\mathrm{i}'_t = \exp\left(\log\left(\mathrm{i}_t\right) - m_t\right) = \exp\left(\tilde{\mathrm{i}}_t - m_t\right) \qquad \text{stabil. input gate} \qquad (16)$$

$$\mathrm{f}'_t = \exp\left(\log\left(\mathrm{f}_t\right) + m_{t-1} - m_t\right) \qquad \text{stabil. forget gate} \qquad (17)$$

mLSTM block:

$$\boldsymbol{C}_t = \mathrm{f}_t \, \boldsymbol{C}_{t-1} + \mathrm{i}_t \, \boldsymbol{v}_t \, \boldsymbol{k}_t^\top \qquad \text{cell state} \quad (19)$$

$$\boldsymbol{n}_t = \mathrm{f}_t \, \boldsymbol{n}_{t-1} + \mathrm{i}_t \, \boldsymbol{k}_t \qquad \text{normalizer state} \quad (20)$$

$$\boldsymbol{h}_t = \mathbf{o}_t \odot \tilde{\boldsymbol{h}}_t \,, \qquad \tilde{\boldsymbol{h}}_t = \boldsymbol{C}_t \, \boldsymbol{q}_t \, / \, \max\left\{ \left| \boldsymbol{n}_t^\top \, \boldsymbol{q}_t \right|, 1 \right\} \qquad \text{hidden state} \quad (21)$$

$$\boldsymbol{q}_t = \boldsymbol{W}_q \, \boldsymbol{x}_t + \boldsymbol{b}_q \qquad \text{query input} \quad (22)$$

$$\boldsymbol{k}_t = \frac{1}{\sqrt{d}} \boldsymbol{W}_k \, \boldsymbol{x}_t + \boldsymbol{b}_k \qquad \text{key input} \quad (23)$$

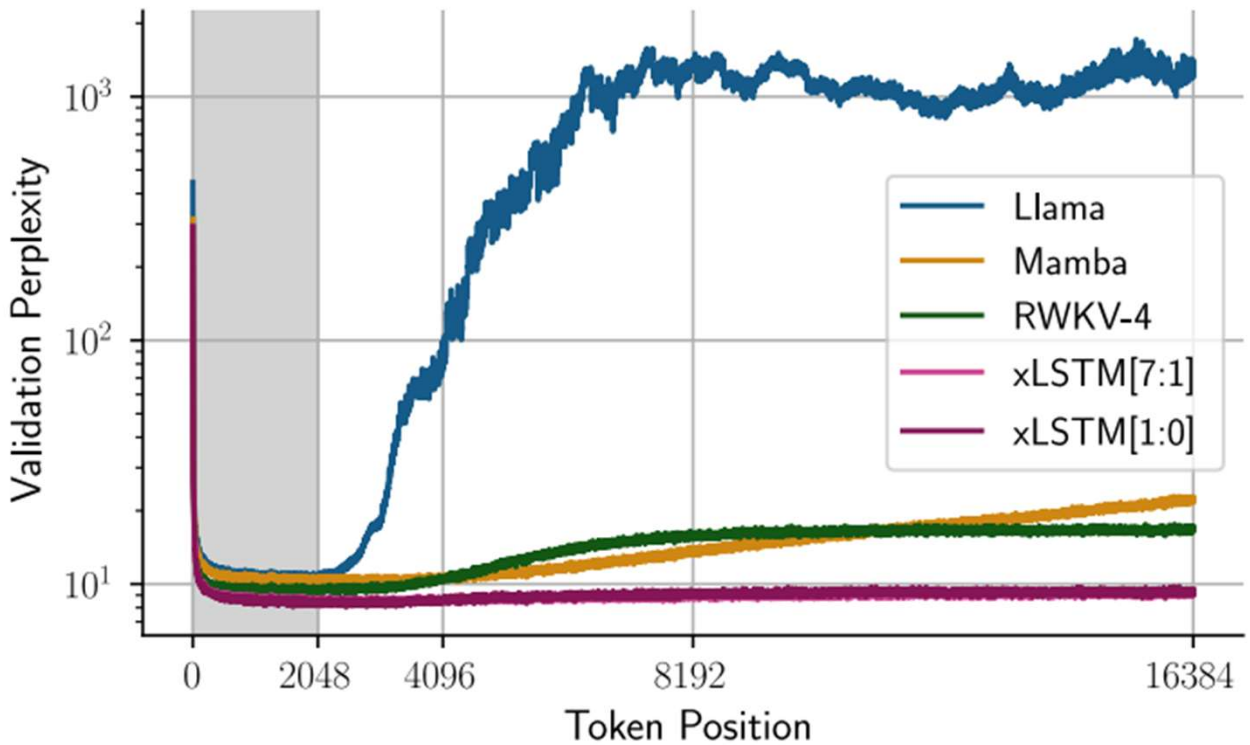
$$\boldsymbol{v}_t = \boldsymbol{W}_v \, \boldsymbol{x}_t + \boldsymbol{b}_v \qquad \text{value input} \quad (24)$$

$$\mathrm{i}_t = \exp\left(\tilde{\mathrm{i}}_t\right) \,, \qquad \tilde{\mathrm{i}}_t = \boldsymbol{w}_\mathrm{i}^\top \, \boldsymbol{x}_t + b_\mathrm{i} \qquad \text{input gate} \quad (25)$$

$$\mathrm{f}_t = \sigma\left(\tilde{\mathrm{f}}_t\right) \text{ OR } \exp\left(\tilde{\mathrm{f}}_t\right) , \quad \tilde{\mathrm{f}}_t = \boldsymbol{w}_\mathrm{f}^\top \, \boldsymbol{x}_t + b_\mathrm{f} \qquad \text{forget gate} \quad (26)$$

$$\mathbf{o}_t = \sigma\left(\tilde{\mathbf{o}}_t\right) \,, \qquad \tilde{\mathbf{o}}_t = \boldsymbol{W}_\mathbf{o} \, \boldsymbol{x}_t + \boldsymbol{b}_\mathbf{o} \qquad \text{output gate} \quad (27)$$

According to their own tests, it works really nice for language tasks:



Exercises & reading

RNNs:

Nice overview of concepts:
https://github.com/afshinea/stanford-cs-230-deep-learning/blob/master/en/cheatsheet-recurrent-neural-networks.pdf

Discussion of exploding and vanishing gradients (written prior to the invention of GRU):
http://proceedings.mlr.press/v28/pascanu13.pdf

Transformer specific:

A somewhat famous blog-post which people keep referring to:
https://jalammar.github.io/illustrated-transformer/

Good, graphical walkthrough of the transformer concept:
https://www.youtube.com/watch?v=dichIcUZfOw
https://www.youtube.com/watch?v=mMa2PmYJlCo
https://www.youtube.com/watch?v=gJ9kaJsE78k

Very good walkthrough by one of the inventors of self-attention, including the rationale behind that and positional encoding:
https://www.youtube.com/watch?v=5vcj8kSwBCY

Project:

Please tell your TA what your project is going to be about. You don't have to have all the details in place, but you should have an idea about the topic.

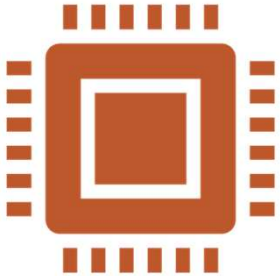
Pitches:

You still have plenty of time to prepare them. But please remember that you have to at some point ☺

The deadline for handing in the project is 11th of December ☺

(and you need to do it through Brightspace)

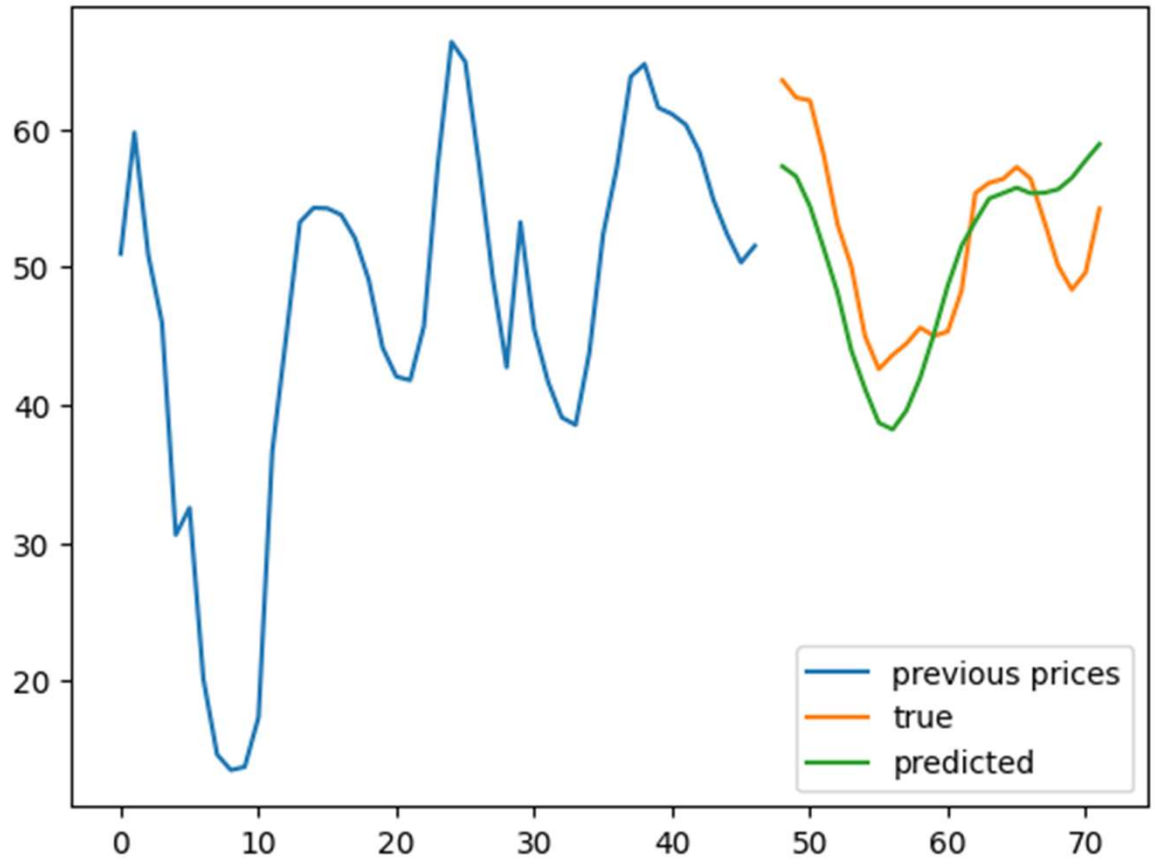
Exercises



GPU exercise

Time series forecasting

Time series forecasting

Using input features from the past 48 hrs, predict electricity pricing for the next 24.

Dataset is prepared for you.