/informatik & security

/fh/// st.pölten

DMML

Studiengang: Informatik & Security

Vortragender: Alexander Adrowitzer, Bernward Asprion, Thomas Delissen

Overview – RNN part

Mo	Tu	We	Thu	Fr	Sa	Su
18.11	19.11		21.11			
25.11						

Datum	Hours	Location	Content
Montag: 18.11.2024	5	A.3.11	Theoretical Foundation RNNs.
Dienstag: 19.11.2024	8	A.3.10	Dealing with long sequences, LSTM, GRU
Donnerstag: 21.11.2024	7	Online	Remaining topics, work on assignment
Montag: 25.11.2024	2	A.2.11	Assignment handin verbal sessions

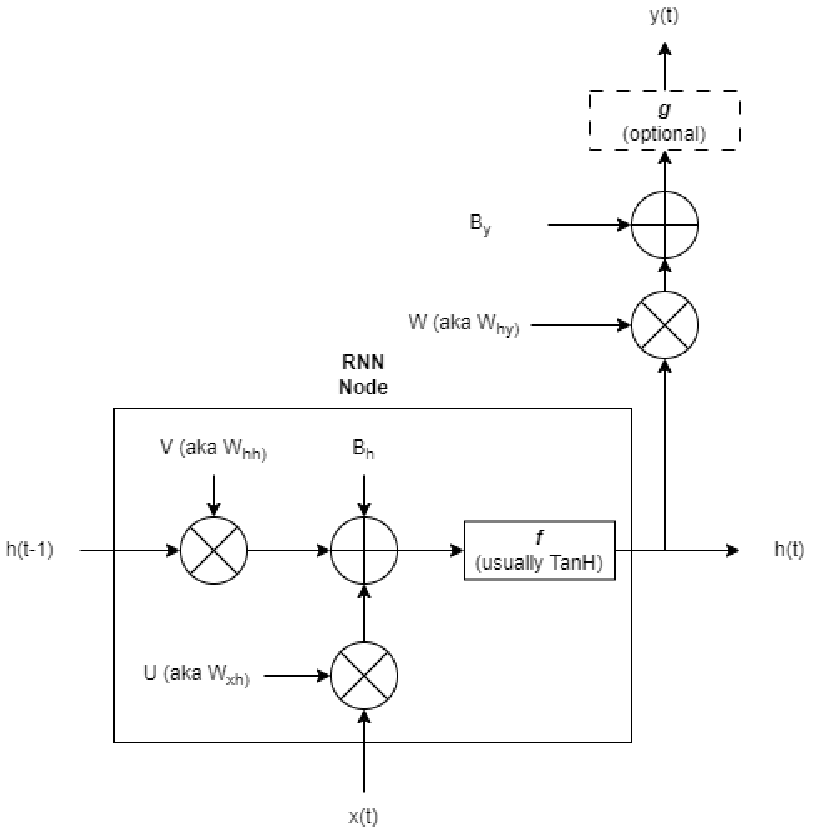
The challenge of long sequences

RNNs are great!

/informatik & security

In theory, RNNs can handle sequences of any lenght

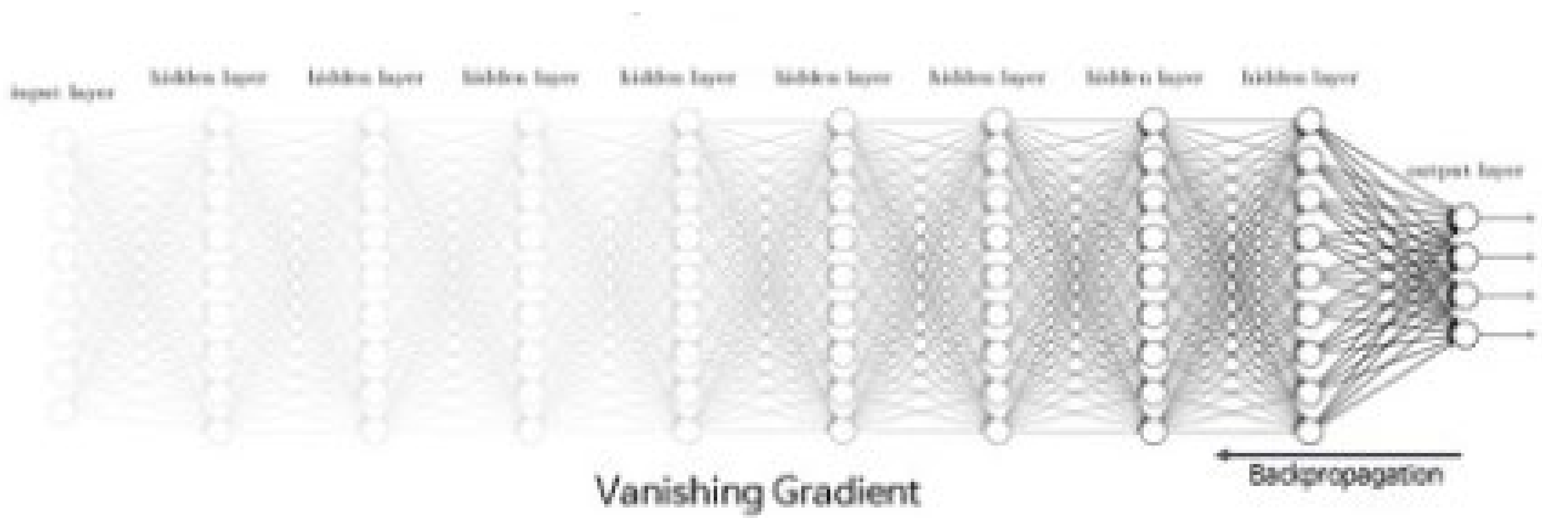
In practice, not so much



Big computational graphs

Both regular neural nets and RNNs can become very large

- Large amount of layers
- Large sequences, resulting in a very large computational graph



https://medium.com/@amanatulla1606/vanishing-gradient-problem-in-deep-learning-understanding-intuition-and-solutions-da90ef4ecb54

Backprop through time

With large sequences, Backprop through time does multiple multiplications with the Weight Matrix for the hidden state (V)

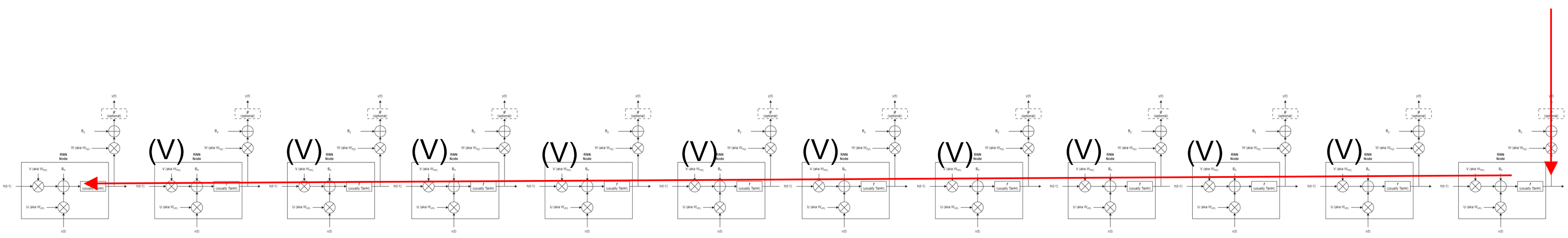
Basically, we can have two situations:

V < 1: Value becomes close to zero

0.7*0.7*0.7*0.7…

V > 1: Value becomes very large

1.3*1.3*1.3*1.3…



Backprop through time

The effect of this is that for the earlier steps in the network, the weights:
- Are updated very little, or
- Wildly oscillate

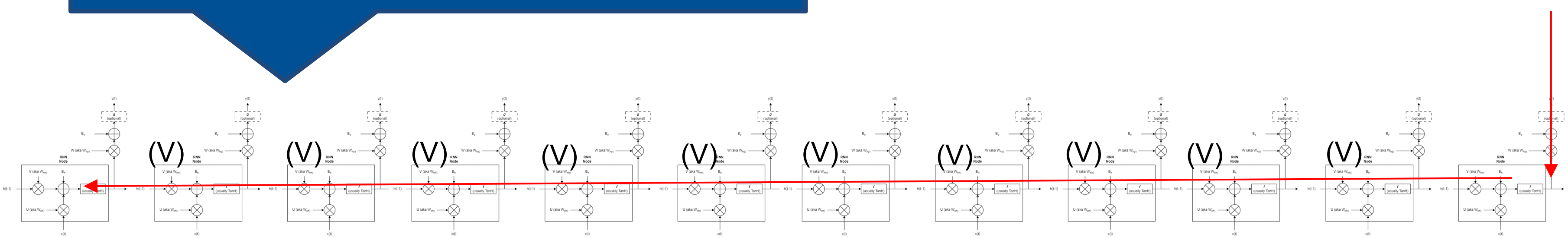
This effect becomes greater the farther back we go in time, meaning that earlier timesteps contribute very little to the learning of the net.

V < 1: Value becomes close to zero

0.7*0.7*0.7*0.7…

V > 1: Value becomes very large

1.3*1.3*1.3*1.3…

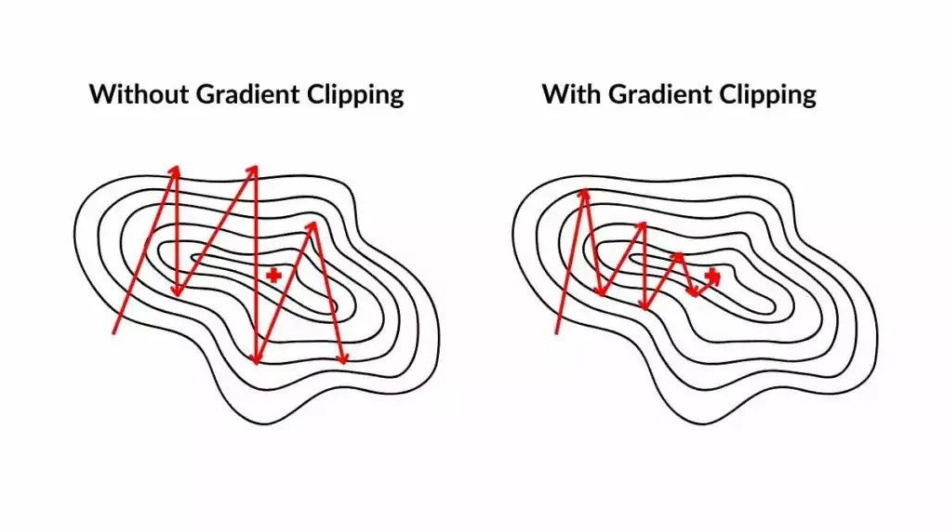


Dealing with exploding gradients - Gradient clipping

/informatik & security /fh/// st.pölten

The exploding gradient problem can cause your model to become worse during training, because it overshoots the minimum

A simple approach is to „clip“ the gradients when they become too large, basically scaling them when to go over a certain threshold



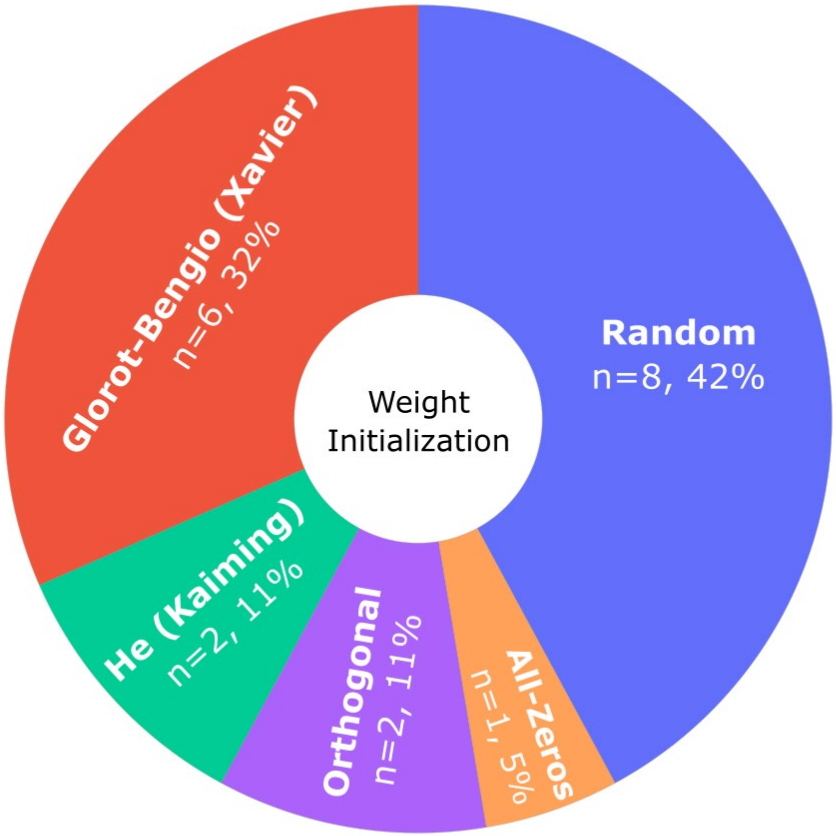
https://spotintelligence.com/2023/12/06/exploding-gradient-problem/

Dealing with Vanishing gradients

A common cause for the vanishing gradient problem can be poor choice of initial weights.

This is why weight initialisation techniques are often employed to counteract it.

Note that none of these approaches can completely solve the issue



[RNN-LSTM: From applications to modeling techniques and beyond—Systematic review]() - 2023

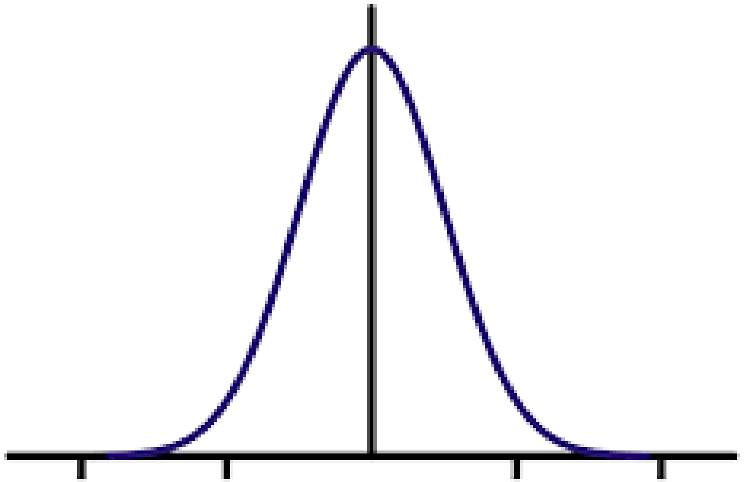
Weight initialisation

/informatik & security /fh/// st.pölten

The „classic“ weight initialisation scheme is random sampling

- Small random values in the range [-0.3, 0.3]
- Small random values in the range [0, 1]
- Small random values in the range [-1, 1]

https://machinelearningmastery.com/weight-initialization-for-deep-learning-neural-networks/



Often, the values are sampled from a gaussian (aka normal) distribution

Weight initialisation – Sigmoid & TanH

/informatik & security /fh/// st.pölten

When using the Sigmoid or TanH activation functions, the standard approach (currently) is to use **Xavier initialization**

Other names are:

- Glorot initialization
- Glorot-Bengio initialization
- Normalized Xavier initialization

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right]$$

[Understanding the difficulty of training deep feedforward neural networks – 2010 - Xavier Glorot, Yoshua Bengio]

- Weights are sampled from U
- U is the uniform distribution
- nj is the number of nodes in previous layer
- Nj+1 is the number of nodes in current layer

This method is supposed to help stabilize learning by reducing vanishing gradients (somewhat)

This seems to be the way Keras does it by default

Weight initialisation – ReLU

/informatik & security /fh/// st.pölten

Xavier initialization does not work that well with ReLU activation function.

The (currently) standard approach for ReLU is to use
He initialization

(or Kaiming initialization)

For the ReLU activation function, this has nowadays become the standard way of initializing weights

Biases are typically initialized to 0.

$$w_l \sim \mathcal{N}(0, 2/n_l)$$

[Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification – 2015, Kaiming He et. al.]

- Weights are sampled from N
- N is the normal distribution
- nl is the number of nodes in current layer

Other methods

Of course, other approaches exist to deal with vanishing/exploding gradients:

- Batch normalisation
- Dropout layers
- Using leaky ReLU
- L2 Regularisation

But the problem persists for RNNs…

They cannot seem to remember information from long age

[Long Short-term Memory, 1997, Sepp Hochreiter, Jürgen Schmidhuber]

LSTMs

They might be less complicated then you think

LSTM sind sequentiell

Sentiment analysis

Let us say we want to do sentiment analysis, to determine if a text is negative or positive. Let us say we have this sentence:

„I really hated that movie.
On a completely unrelated note, I went hiking last week, saw a beautiful deer and made a photograph of it.
But that didn‘t change the way I felt about that movie.“

Sentiment analysis

Let us say we want to do sentiment analysis, to determine if a text is negative or positive. Let us say we have this sentence:

„I really hated that movie.
On a completely unrelated note, I went hiking last week, saw a beautiful deer and made a photograph of it.
But that didn‘t change the way I felt about that movie.“

The green part is highly relevant, the red part is not.

→ A regular RNN will try to encode ALL information, regardless if it is important or not for the task at hand

Long Short-term Memory

The main idea of LSTM is that we enable an RNN cell to:

1. Decide what to forget
2. Decide what to remember
3. Decide what to output

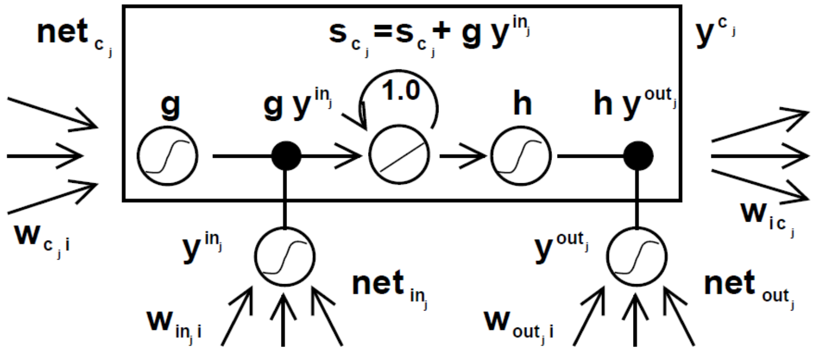


Figure 1: *Architecture of memory cell c_j (the box) and its gate units in_j, out_j. The self-recurrent connection (with weight 1.0) indicates feedback with a delay of 1 time step. It builds the basis of the "constant error carrousel" CEC. The gate units open and close access to CEC. See text and appendix A.1 for details.*

[Long Short-term Memory, 1997, Sepp Hochreiter, Jürgen Schmidhuber]

Long Short-term Memory

/informatik & security

The main idea of LSTM is that we enable an RNN cell to:

1. Decide what to forget
2. Decide what to remember
3. Decide what to output

In order to do this, we need a second "hidden" state:

Neuer State (steht für context?)

C: for step 1 and 2 (cell state)

H: for step 3 (like before)

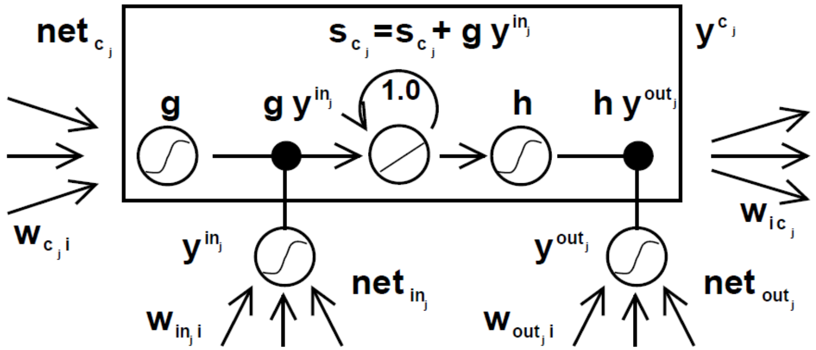


Figure 1: *Architecture of memory cell c_j (the box) and its gate units in_j, out_j. The self-recurrent connection (with weight 1.0) indicates feedback with a delay of 1 time step. It builds the basis of the "constant error carrousel" CEC. The gate units open and close access to CEC. See text and appendix A.1 for details.*

[Long Short-term Memory, 1997, Sepp Hochreiter, Jürgen Schmidhuber]

Mathematical notation

/informatik & security /fh/// st.pölten

Vanilla RNN

$$\boxed{h_t = \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)}$$

LSTM

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Element-wise Multiplication

Screenshot from lecture from Justin Johnson

Gates in LSTM

dotproduct = Matrix multiplication
point multiplication = elementwise multiplication
```
-> Matrix | A |  o  | C |    =>  | A*C |
          | B |  o  | D |        | B*D |
```

Where a „regular“ RNN only has one gate (the tanH), the LSTM has 4 gates, here in order of execution during a forward pass:

f: <u>forget gate</u>: Decide what to forget from cell state c
i: <u>input gate</u>: Decide which parts of the cell state c we want to update
g: <u>candidate gate</u>: What we would like to write to the cell (the original gate)
o: <u>output gate</u>: decides how much we should update the hidden state h, which is also outputted to the „outside world“

LSTM

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

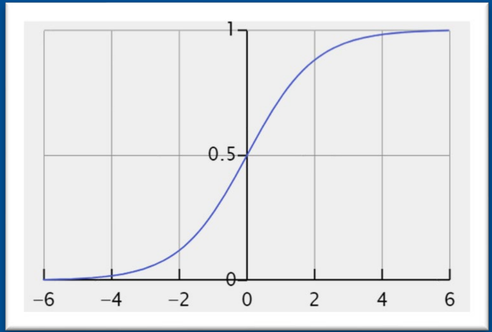
$$h_t = o \odot \tanh(c_t)$$

[Based on lecture from Justin Johnson]

Gates in LSTM

We use Sigmoids because they output between 0 and 1, which you can interpret as a kind of „boolean“ decision:

Forget or not forget
Write or not write
Output or not output



LSTM

$$\begin{pmatrix} f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

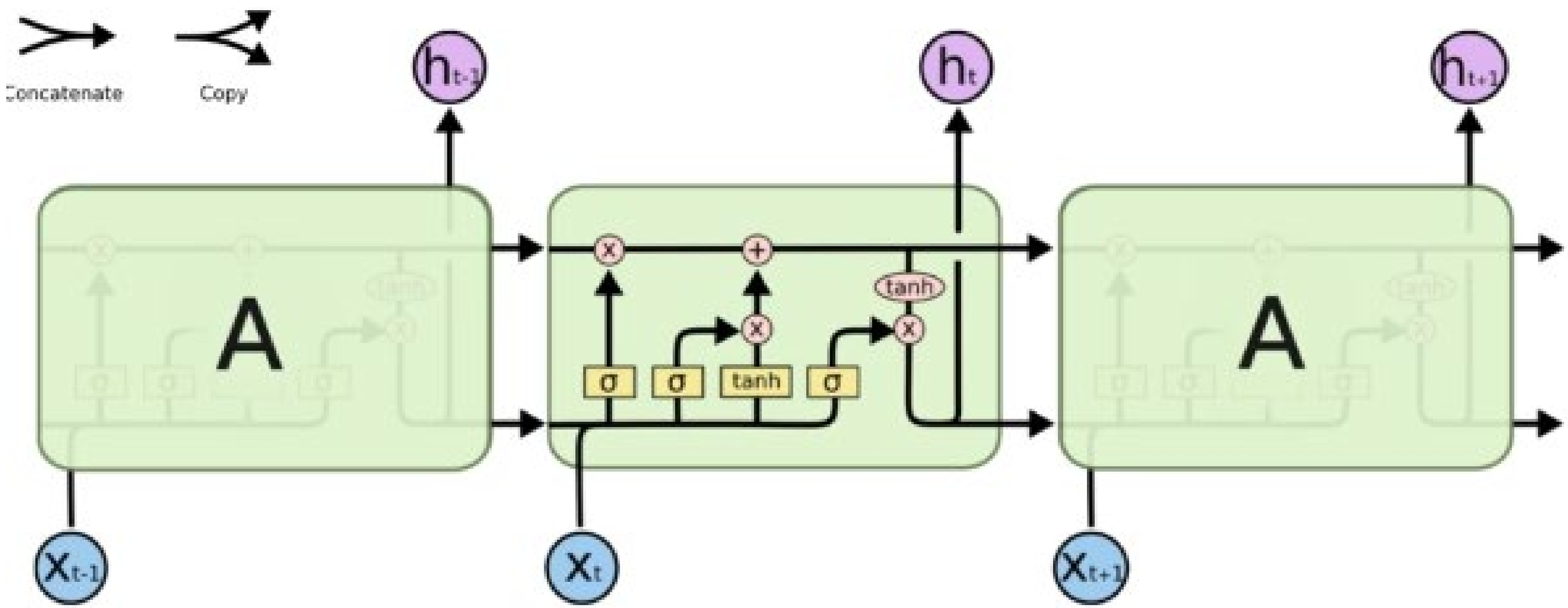
$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

[Based on lecture from Justin Johnson]

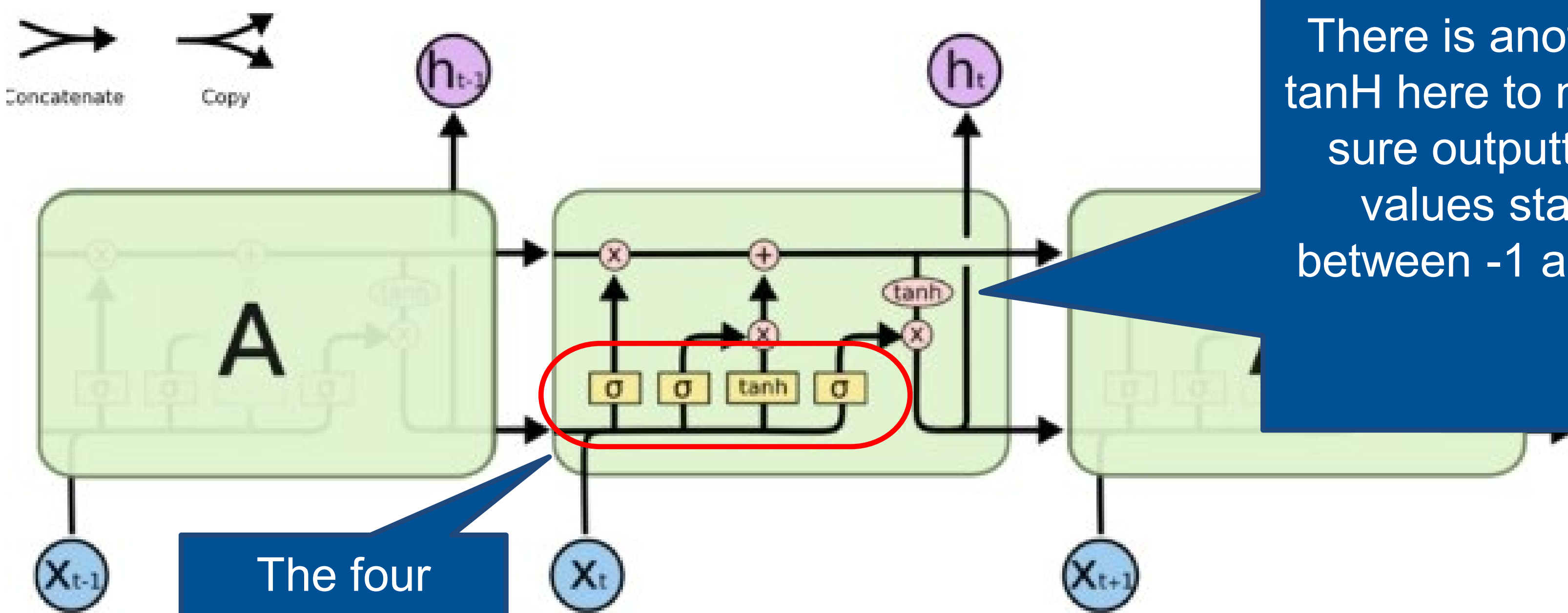
Cell visualisations

/informatik & security



[From MIT Lecture Lex Fridman]

Cell visualisations

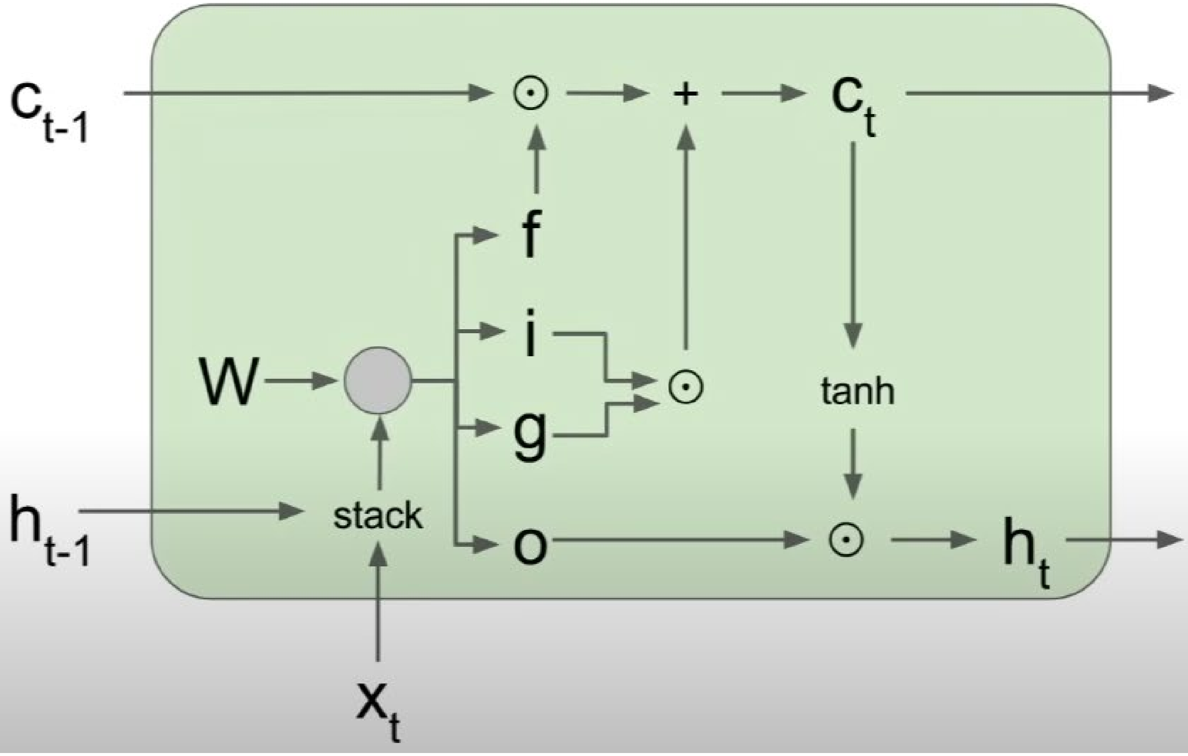


The four gates

There is another tanH here to make sure outputted values stay between -1 and 1

[From MIT Lecture Lex Fridman]

Cell visualisations



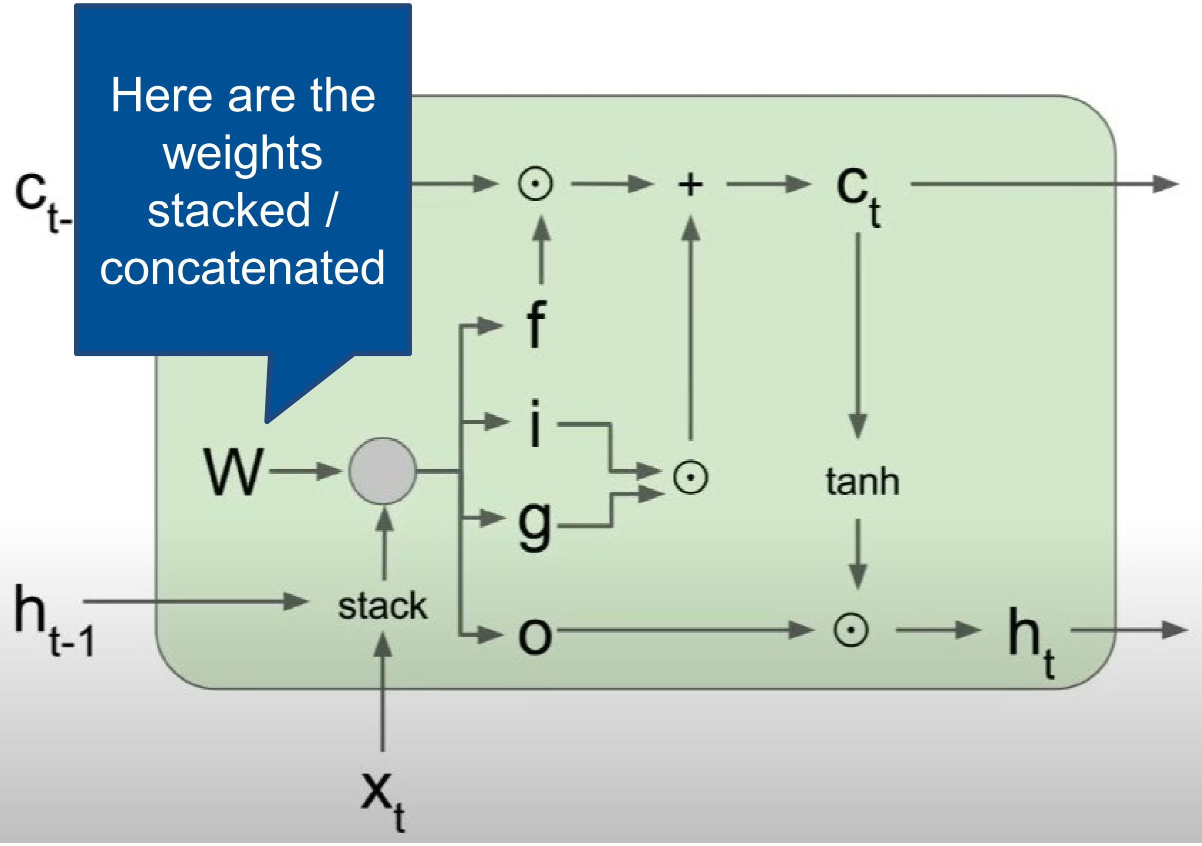
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Screenshot from lecture from Justin Johnson

Cell visualisations



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

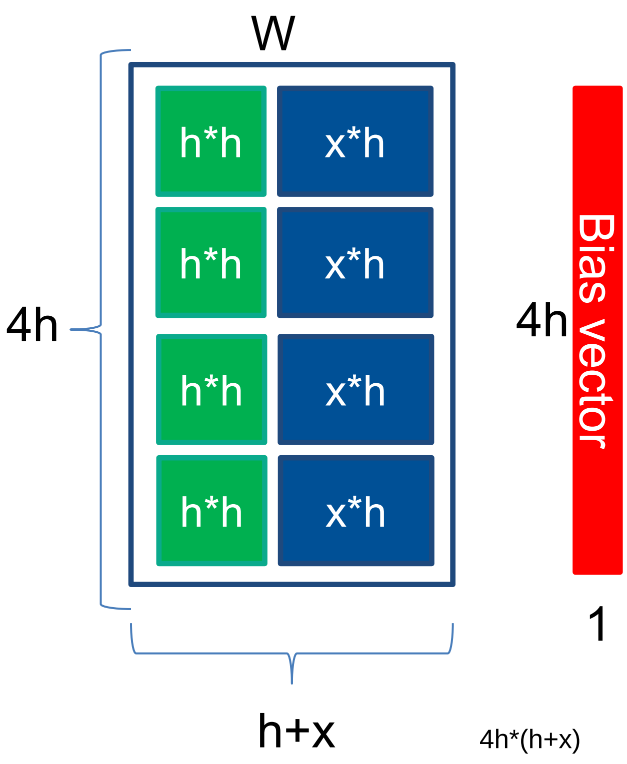
$$c_t = f \odot c_{t-1} + i \odot g$$

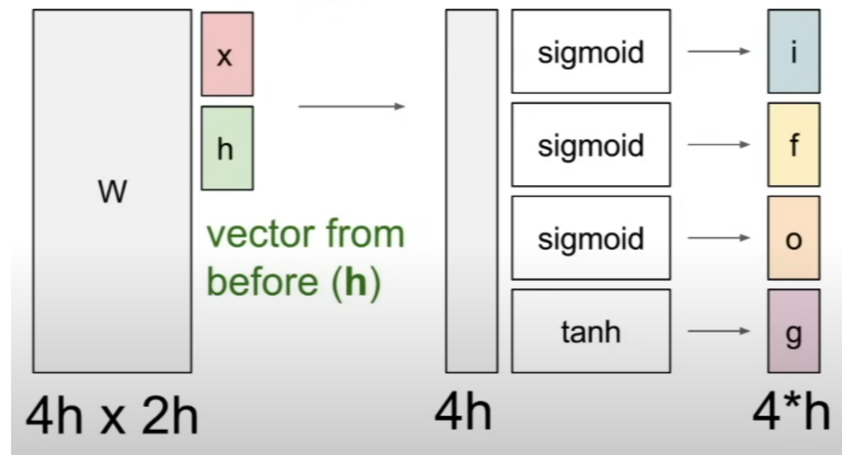
$$h_t = o \odot \tanh(c_t)$$

Screenshot from lecture from Justin Johnson

Weight matrix

Size of the weight matrix depends on the size of h and x





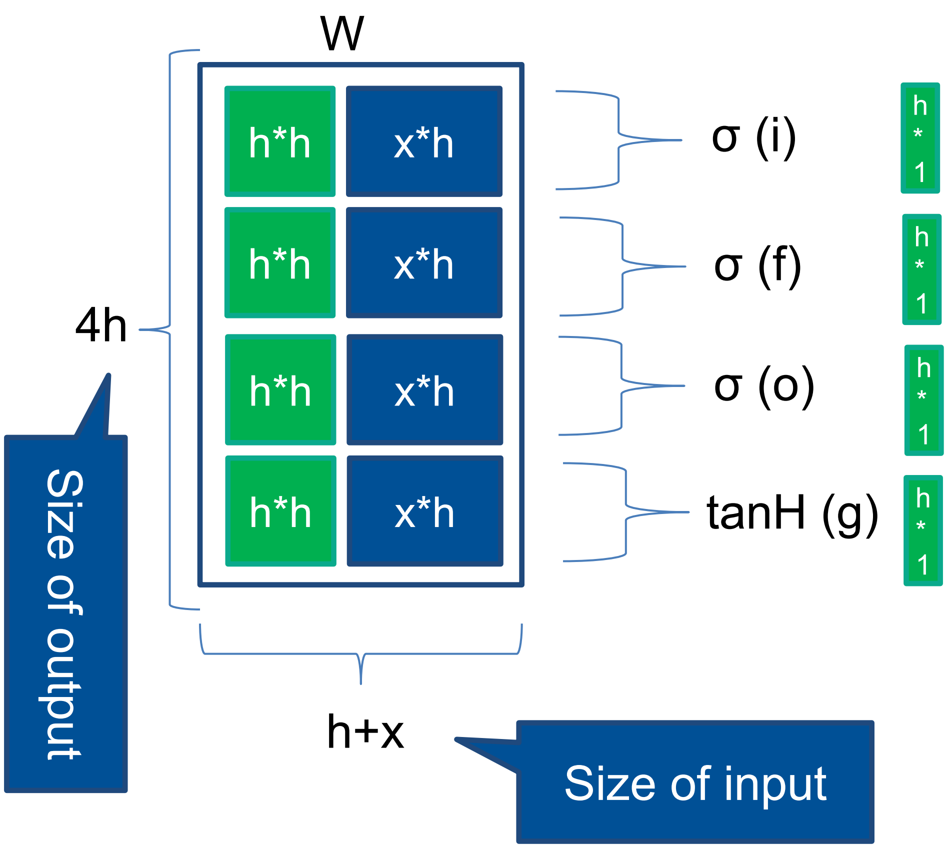
[Screenshot from lecture from Justin Johnson]()

This diagram might be slightly confusing:
Here the assumption is that **h** has same size as **x**

Weight matrix

/informatik & security /fh/// st.pölten

Size of the weight matrix depends on the size of h and x

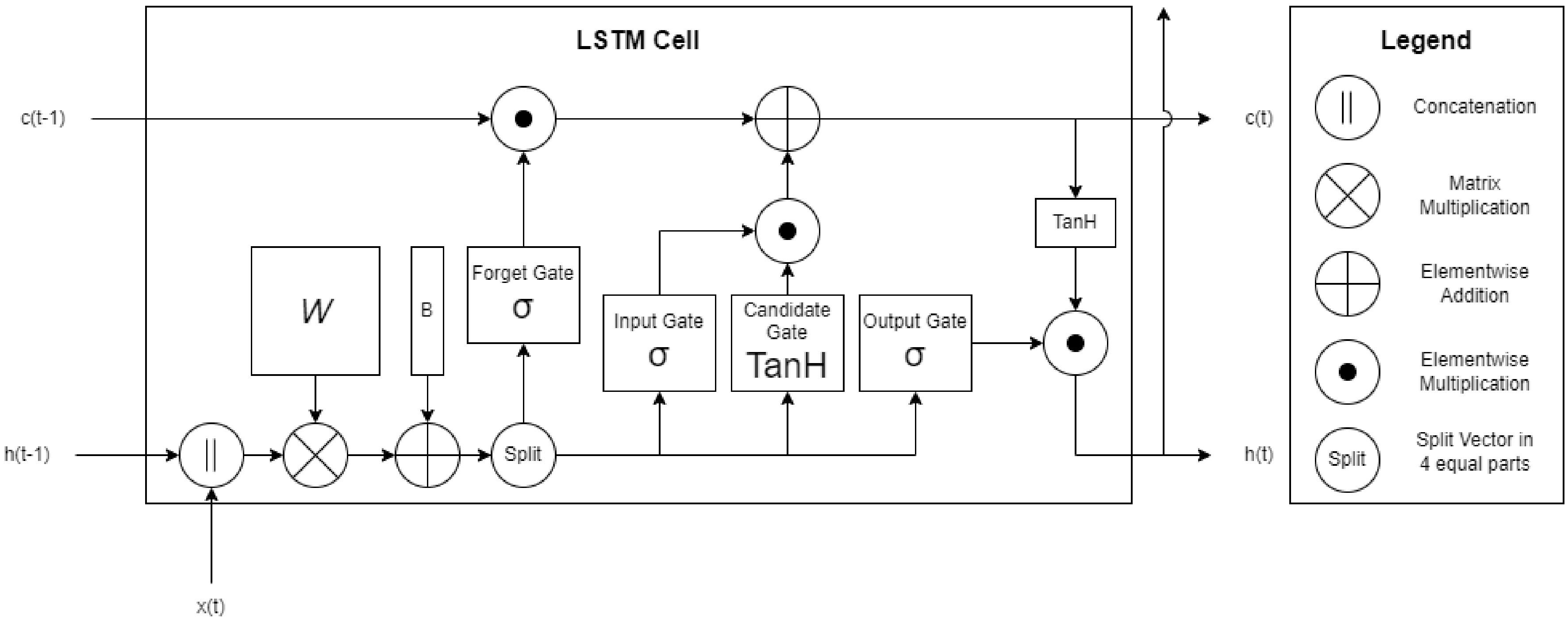


So the input vectors h and x are multiplied by these weight matrices, which results in 4 vectors of size h that are fed into the activation functions (gates)

So only a PART of the output vector is fed into each gate.

My version of a drawing

/informatik & security /fh/// st.pölten

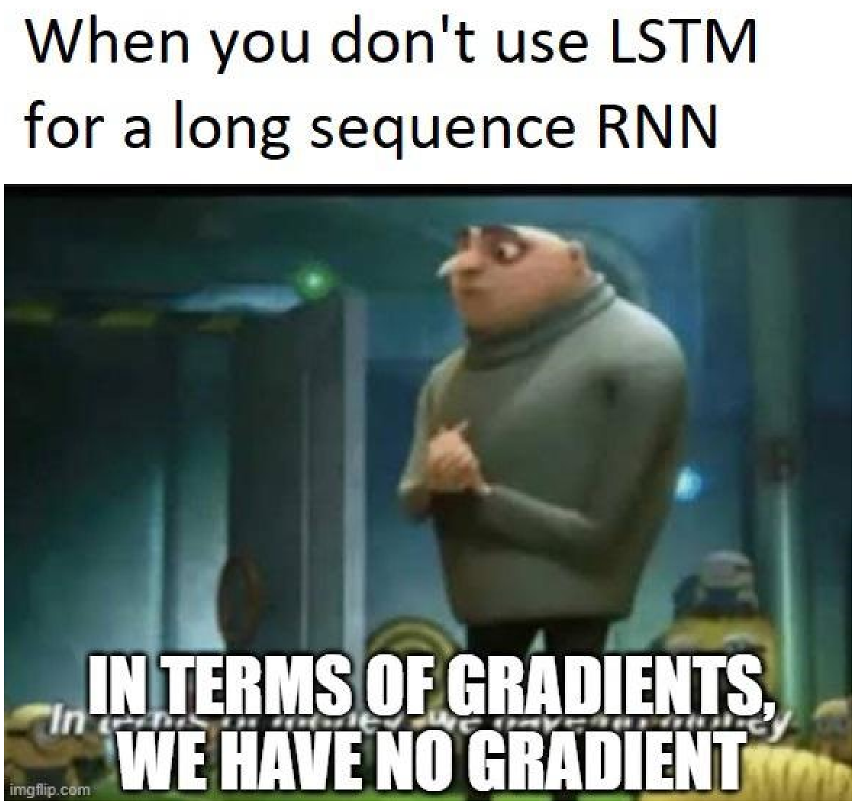


ein element pro time schritt rein

How to use

Congratulations, now you know how a LSTM Cell works!

- Usage of the cell is exactly the same as for a regular RNN
  - LSTM is considered a type of RNN (the most common one)
- Except that it can handle longer sequences much better
- And that it is more computationally expensive



Gated Recurrent Units

LSTMs are kind of complicated

Can‘t we come up with a RNN variant that is:

- Simpler to understand
- Less compute intensive
- But still works well with vanishing gradients?

Meet the **Gated Recurrent Unit**

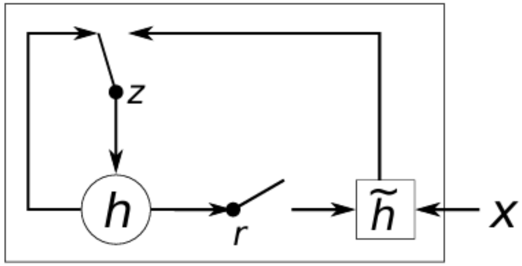
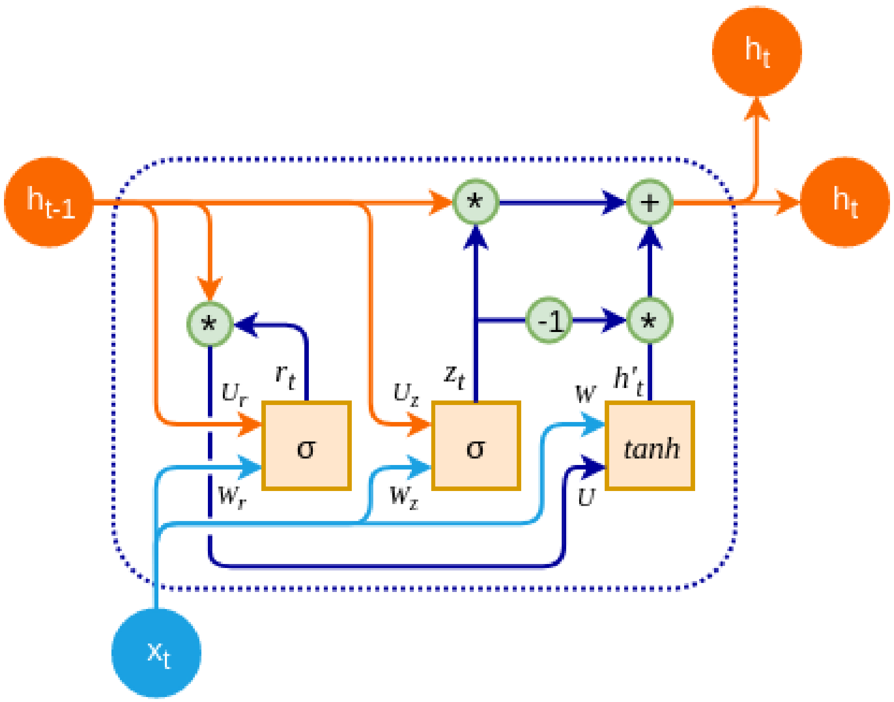


Figure 2: An illustration of the proposed hidden activation function. The update gate z selects whether the hidden state is to be updated with a new hidden state $\tilde{h}$. The reset gate r decides whether the previous hidden state is ignored. See Eqs. (5)–(8) for the detailed equations of r, z, h and $\tilde{h}$.

Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation – 2014 – Cho et. al.

Gated Recurrent Unit



https://medium.com/@anishnama20/understanding-gated-recurrent-unit-gru-in-deep-learning-2e54923f3e2

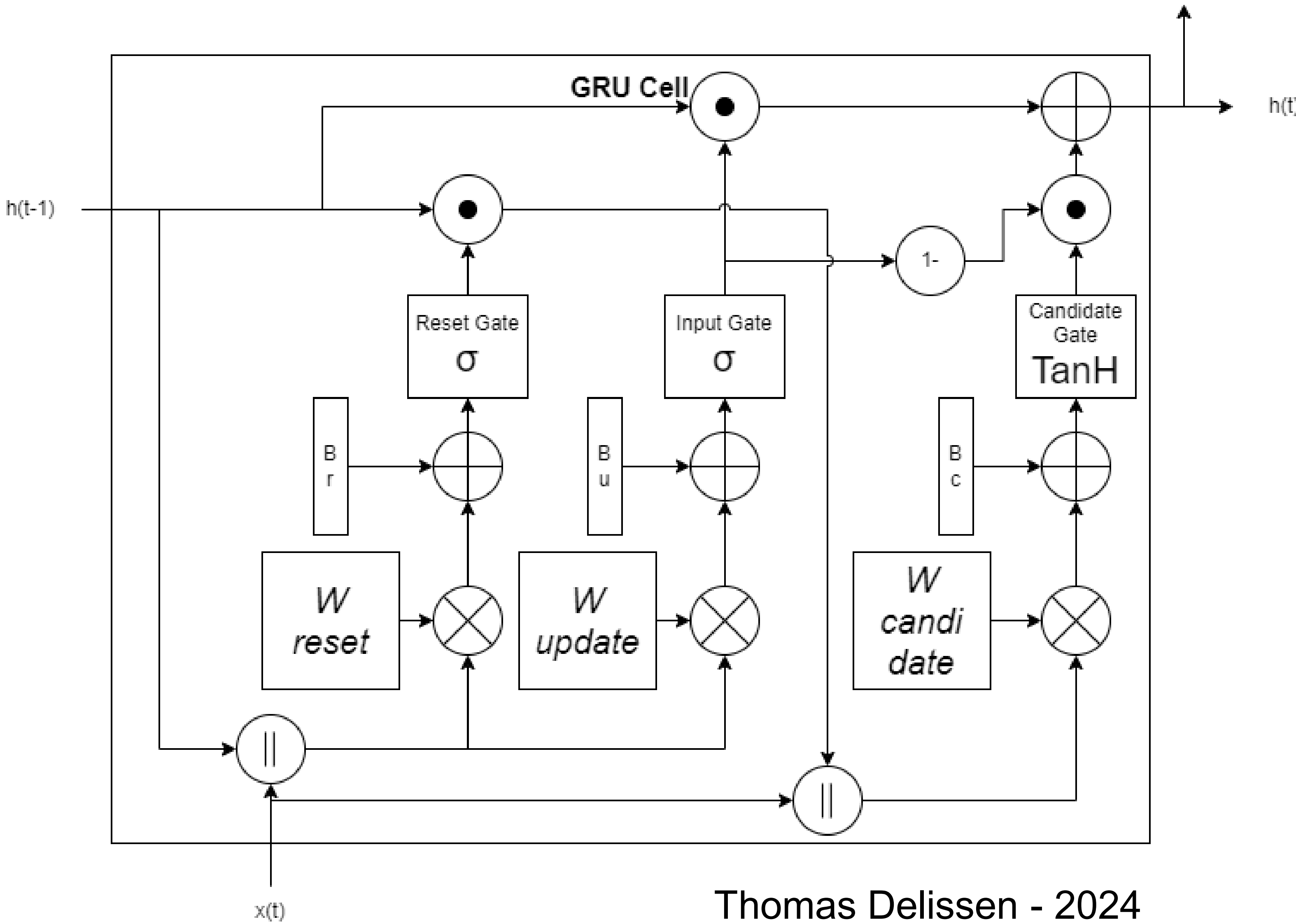
The gated recurrent unit GRU is similar to an LSTM because:

- It also has multiple gates
- Tries to combat vanishing gradient problem

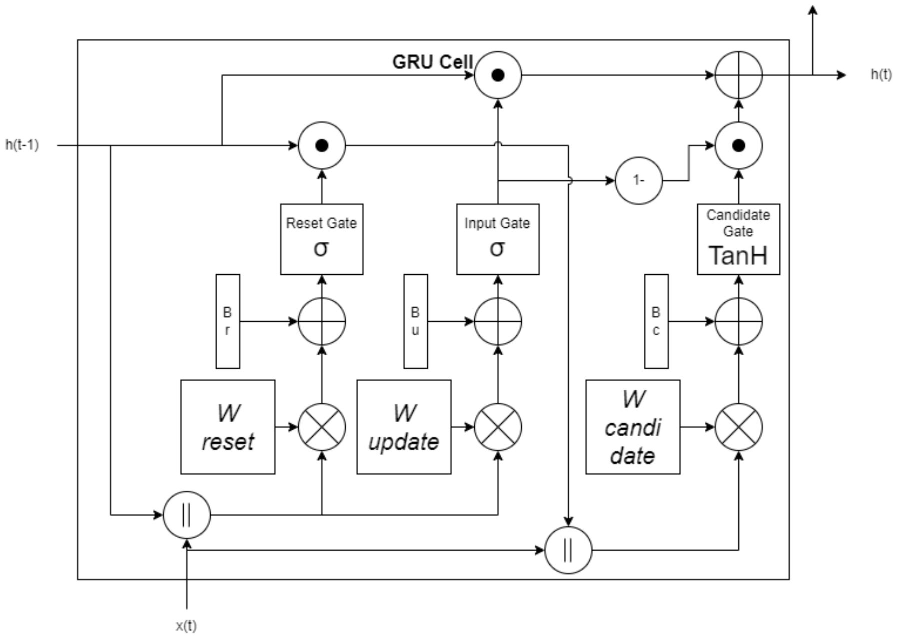
It is different from LSTM because:

- It does not have a state c, uses h for this purpose
  - Thus, also no output gate necessary
- (Forget gate is called reset gate, but works the same)

My variant of a GRU



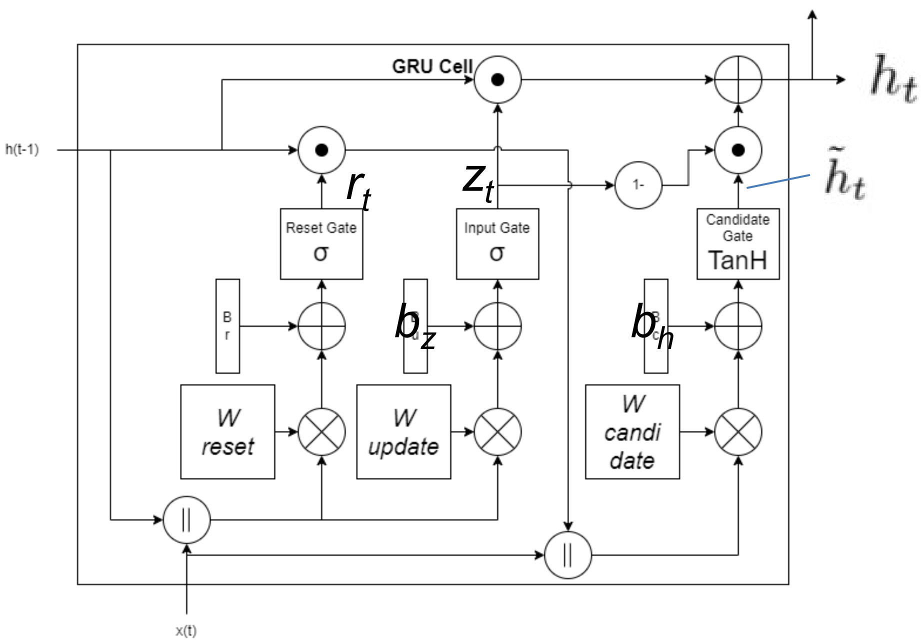
My variant of a GRU



Notes:

- I split the weight matrix in three parts (which I did not do for the LSTM, which is why it might seem more complex
  - Mainly because input for candidate cell is not directly h
- "1-" Operator takes the input value, and deducts it from 1

Mathematically - GRU



$$r_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r)$$

$$z_t = \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z)$$

$$\tilde{h}_t = \tanh(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h)$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t$$

[From lecture from Justin Johnson]

GRU Conclusion

/informatik & security

GRU is theoretically an easier to understand RNN variant than LSTMs

Is it better?

- Maybe? Sometimes?
- It is definitely faster

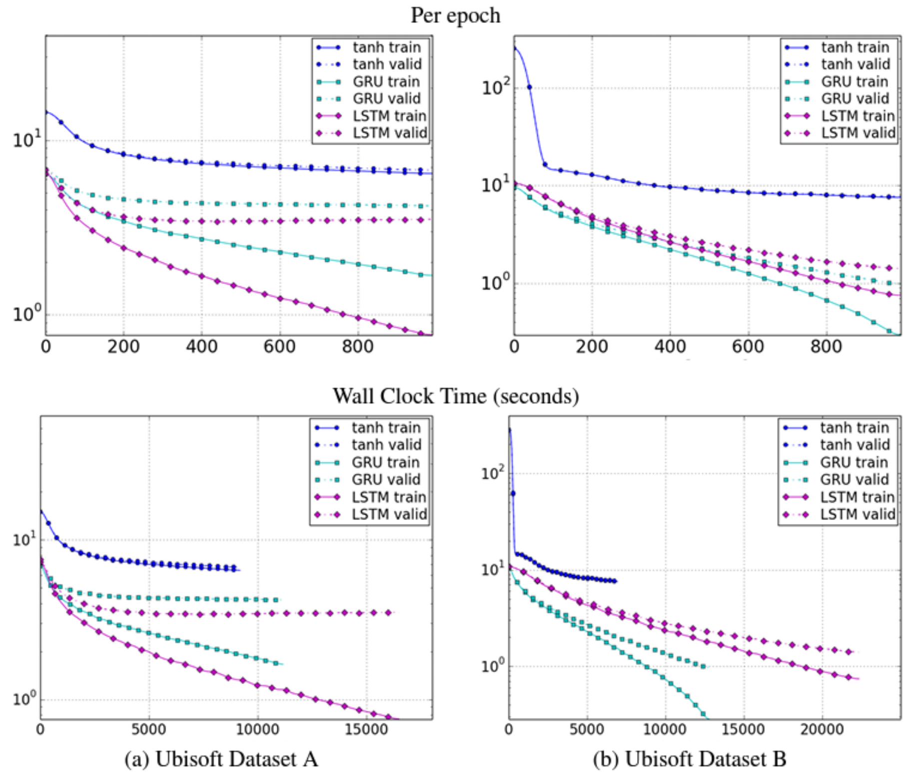


Figure 3: Learning curves for training and validation sets of different types of units with respect to (top) the number of iterations and (bottom) the wall clock time. x-axis is the number of epochs and y-axis corresponds to the negative-log likelihood of the model shown in log-scale.

Image on the right is from paper evaluating the two methods [Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling – 2014 – Chung et. al.]

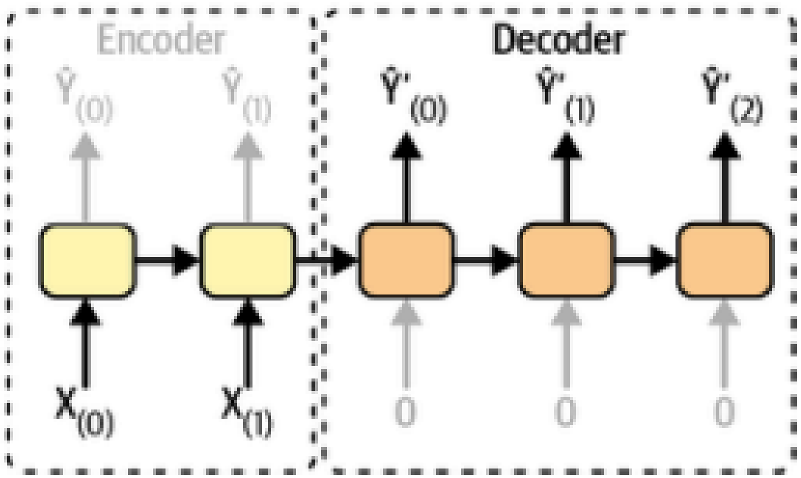
Attention!

Let's make it a bit more complicated ;)

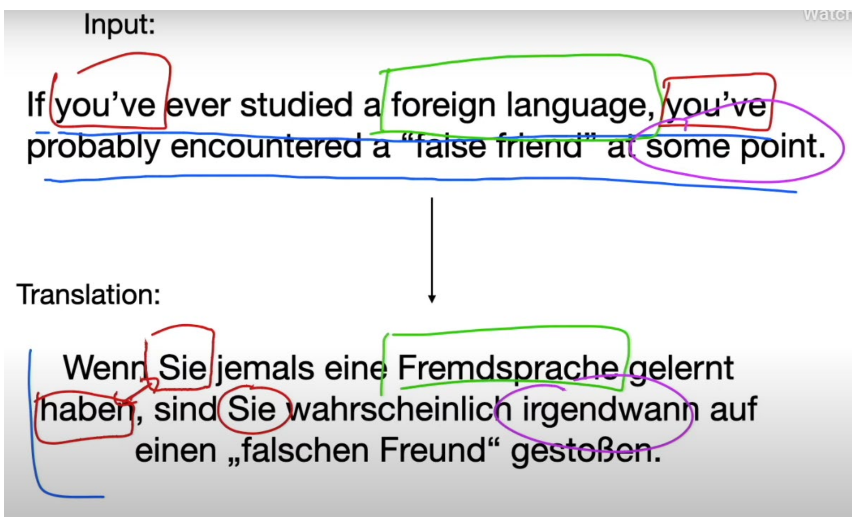
Language Translation

RNNs translate a text by first memorising the entire sentence, then generating the output one word at a time.

- This is not how humans do it



The way humans translate a text



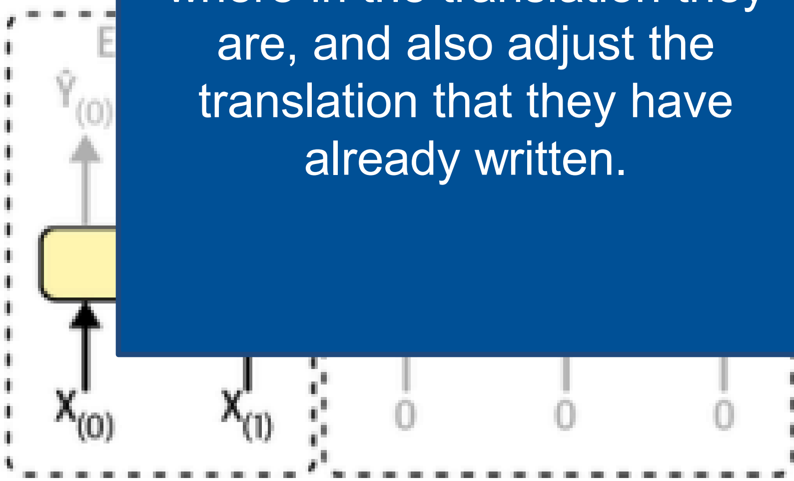
[Sebastian Raschka – Attention Mechanism explained]

Language Translation

/informatik & security

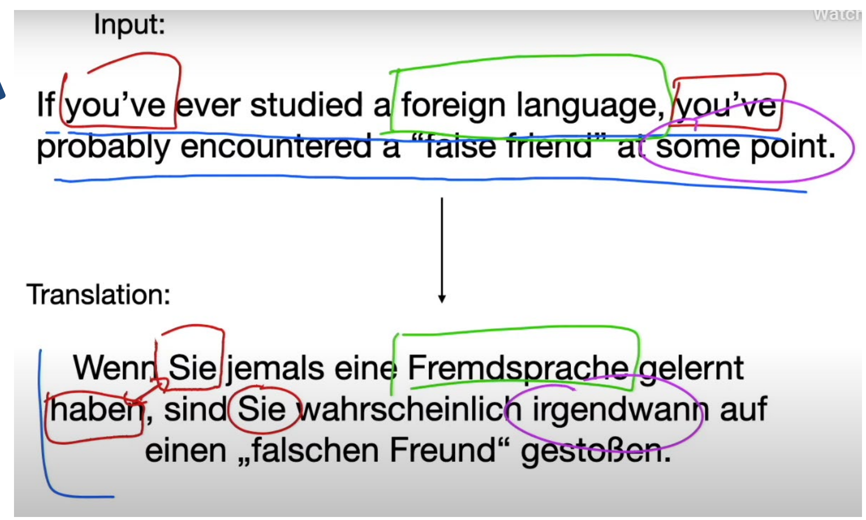
RN
me
the
wo

-



Humans repeatedly read the original sentence, <u>focus on single words</u> depending on where in the translation they are, and also adjust the translation that they have already written.

The way humans translate a text



[Sebastian Raschka – Attention Mechanism explained]()

Attention mechanism

/informatik & security

What if:

- We could give our decoder RNN access to the input entire sentence
- We could teach our decoder <u>on which words to focus</u>

Meet the Attention mechanism!

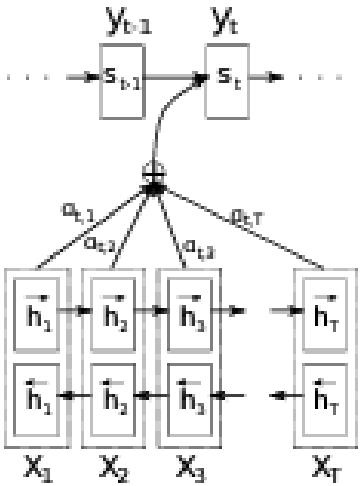


Figure 1: The graphical illustration of the proposed model trying to generate the t-th target word y_t given a source sentence $(x_1, x_2, \ldots, x_T)$.

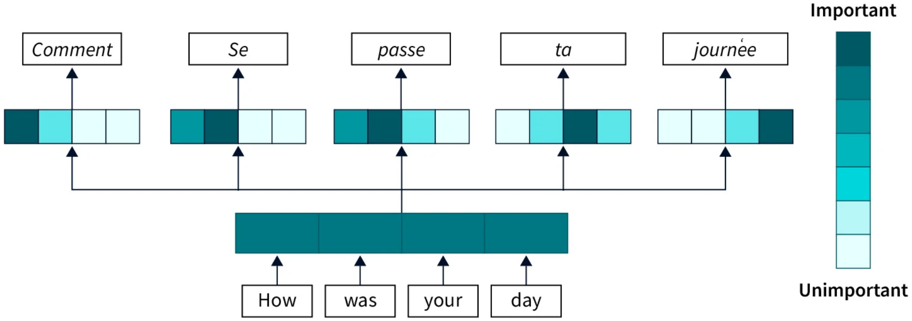
[Neural Machine Translation by Jointly Learning to Align and Translate – 2014 - Bahdanau, Cho, Bengio]

General Idea

The general idea of the attention mechanism is:

- Provide information about the entire input sequence to the decoder – at each step of the decoding (not just at the start)
- Depending on the state of the decoder, we calculate which words in the input sequence the decoder should pay attention to

And this will result in better performance over longer sequences

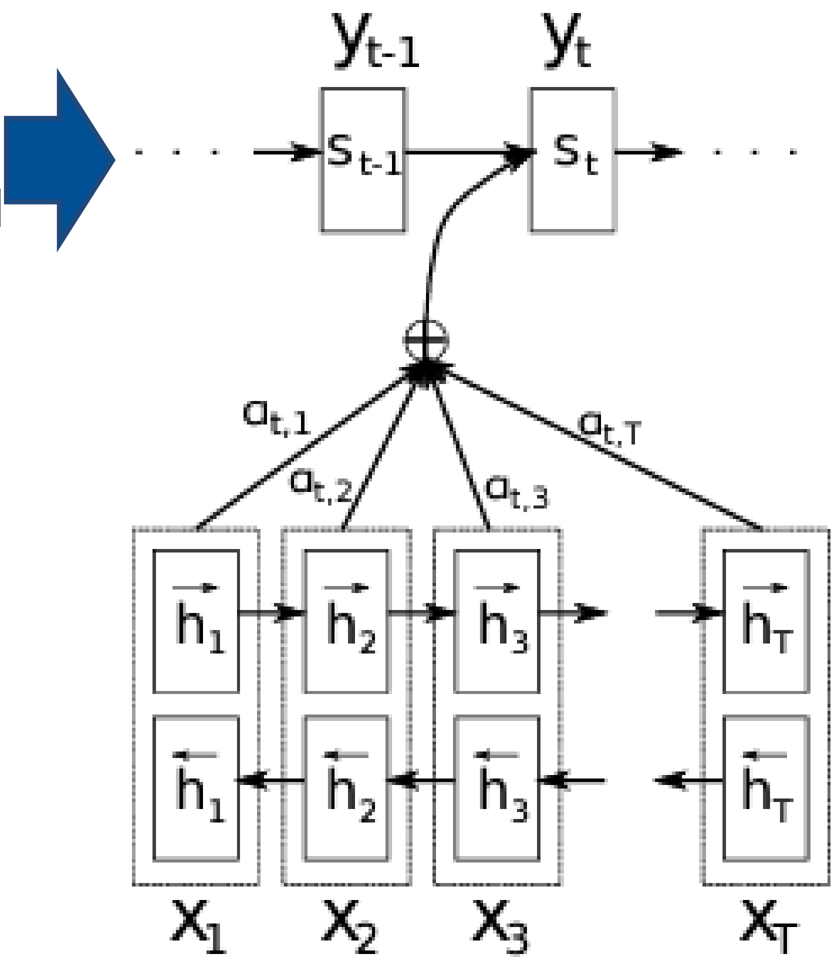


https://www.scaler.com/topics/deep-learning/attention-mechanism-deep-learning/

Slightly different architecture

In general, the model is still an encoder – decoder architecture

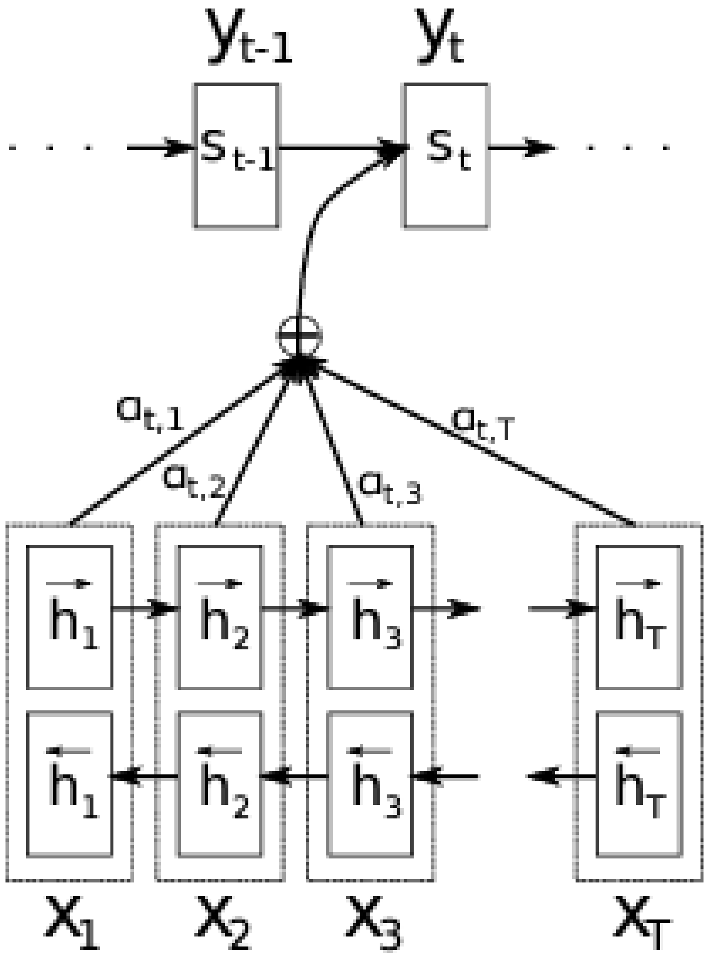
- The **decoder** is a regular RNN that generates an output sequence
- It has a hidden state denoted as "S(t)" which represents "what has been written so far"



Slightly different architecture

/informatik & security /fh/// st.pölten

- The **encoder** (below) is a bi-directional RNN, that processes the input sequence in two directions
- It outputs a value at each “encoder-timestep”

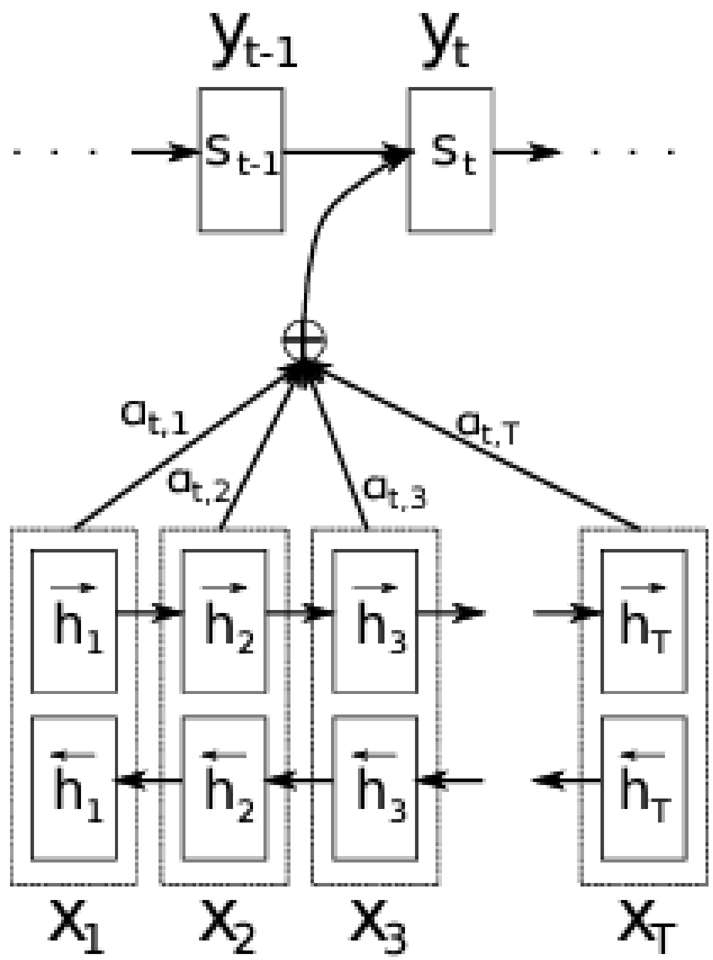


Slightly different architecture

- The output from each timestep is multiplied with so called “attention weights”, then all the multiplied values are summed up and provided to the decoder mechanism.

Here comes the magic part:

The attention weights are <u>different</u> for each decoder-timestep!



Slightly different architecture

/informatik & security /fh/// st.pölten

- The output from each timestep is multiplied with so called “attention weights”, then all the multiplied values are summed up and provided to the decode mechanism.

Here comes the magic part:

The attention weights are <u>different</u> for each decoder-timestep!

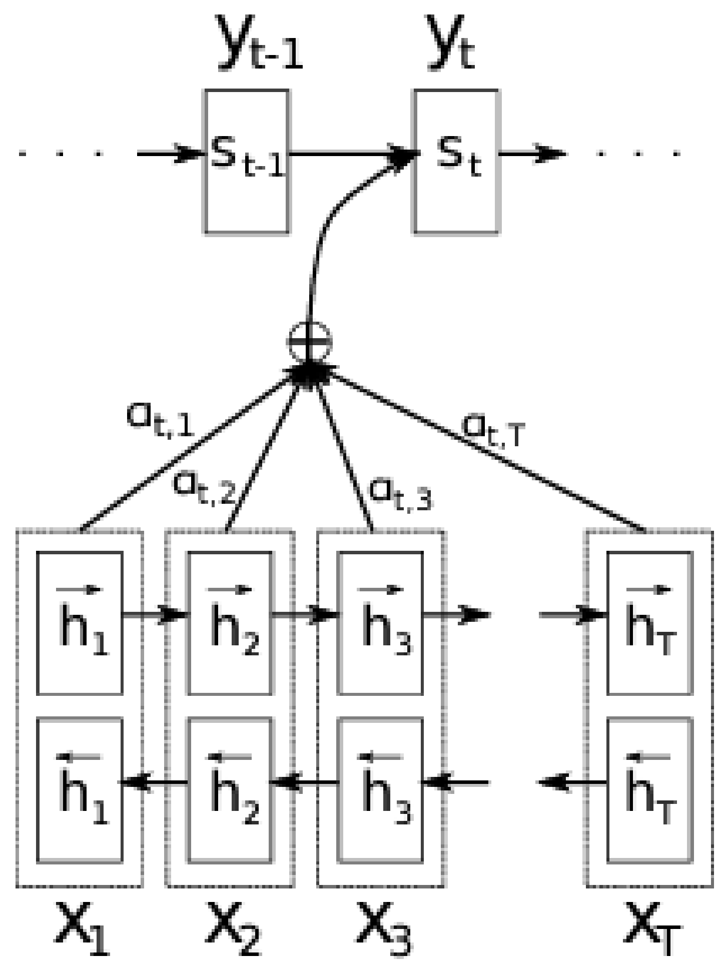
When I first heard this, my mind was blown, as this seems impossible

Calculating the attention

The attention should be:

- Different for each timestep of the encoder
- Different for each timestep of the encoder

We apply it for each combination:

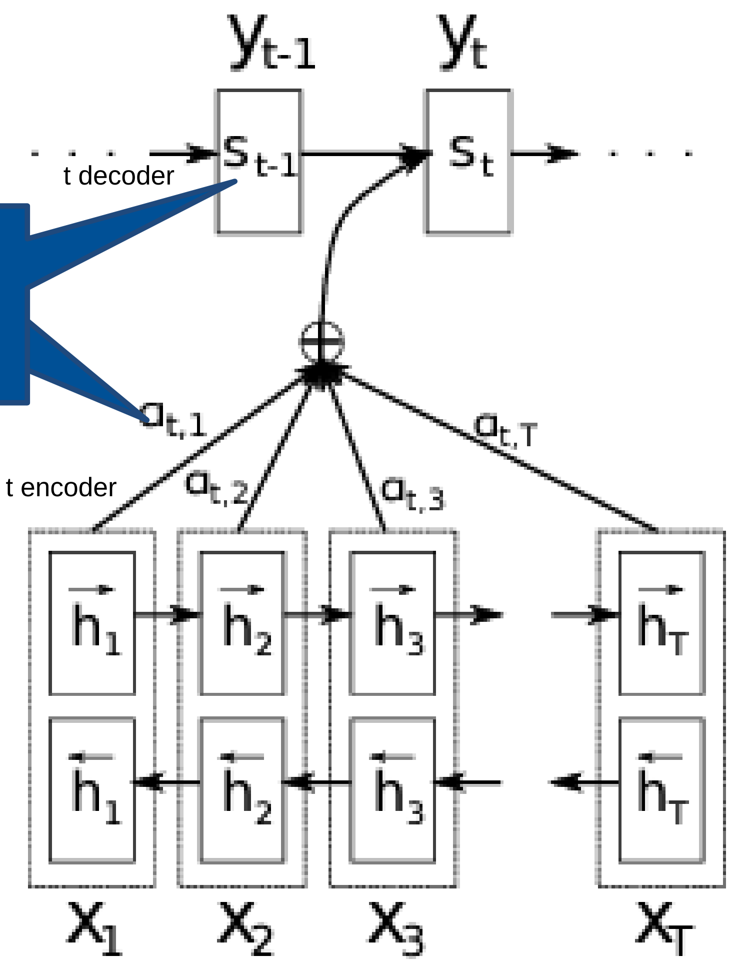


Calculating the attention

The attention should be:

- Different for each ti
  the encoder
- Different for each ti
  the encoder

These are not the same „t“!!!



We apply it for each combination:

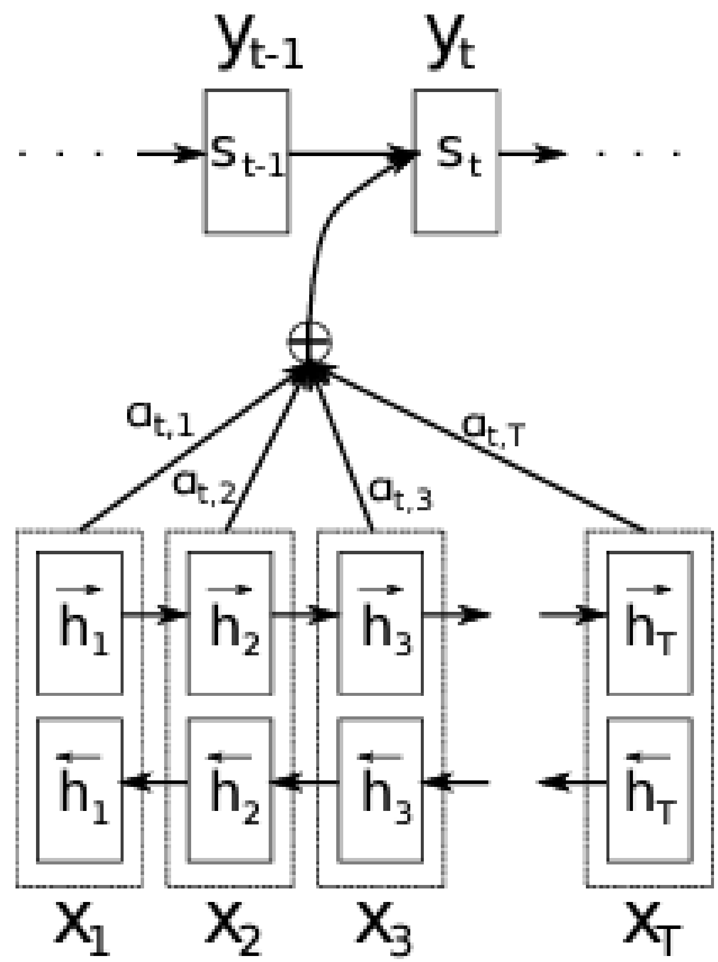
s(t) → Attention „Scorer“ Mechanism → a(t,1)

h(1) → Attention „Scorer“ Mechanism

Calculating the attention

So we calculate the „attention score“ for each combination of s and h. The function could be

- A simple dot product
- A small neural net

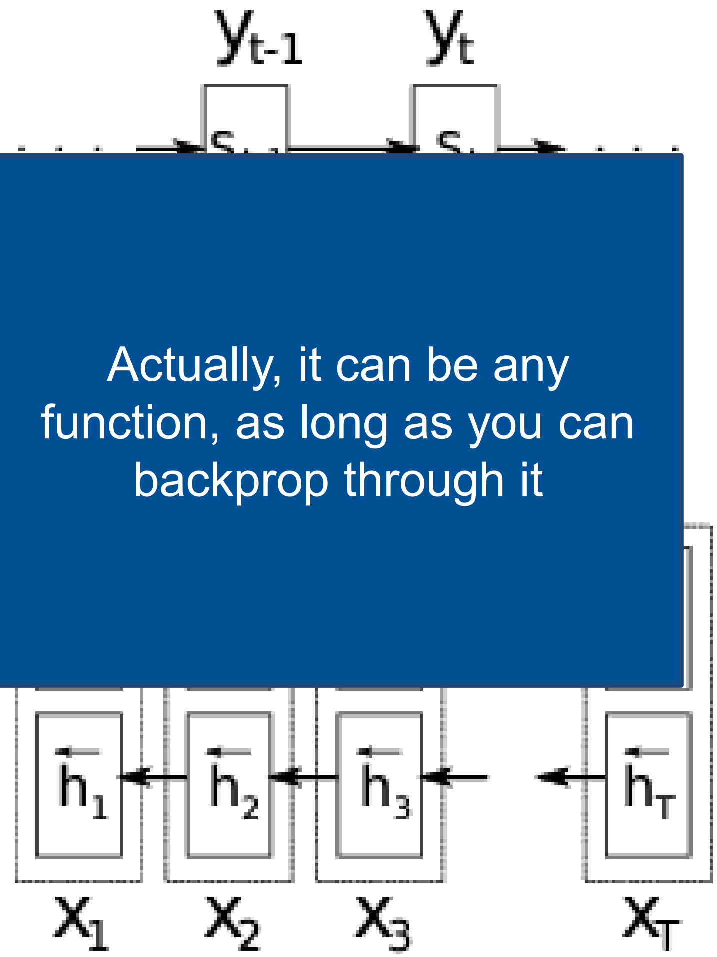


Calculating the attention

So we calculate the „attention score“ for each combination of s and h. The function could be

- A simple dot product
- A small neural net

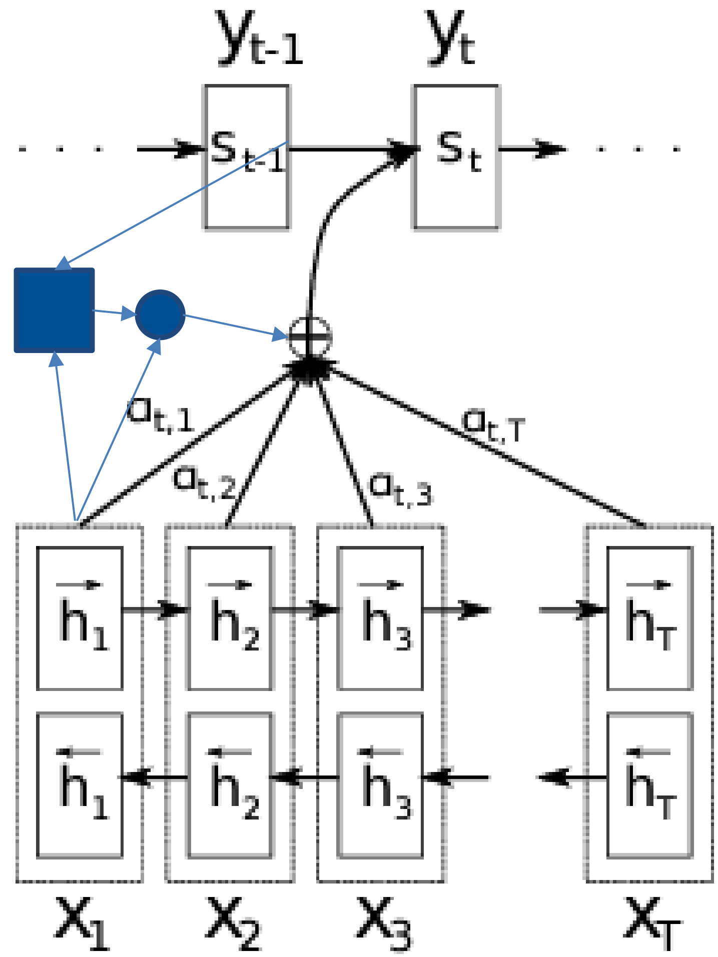
Actually, it can be any function, as long as you can backprop through it



Calculating the attention

So this image is „simplified“

There is actually more going on here



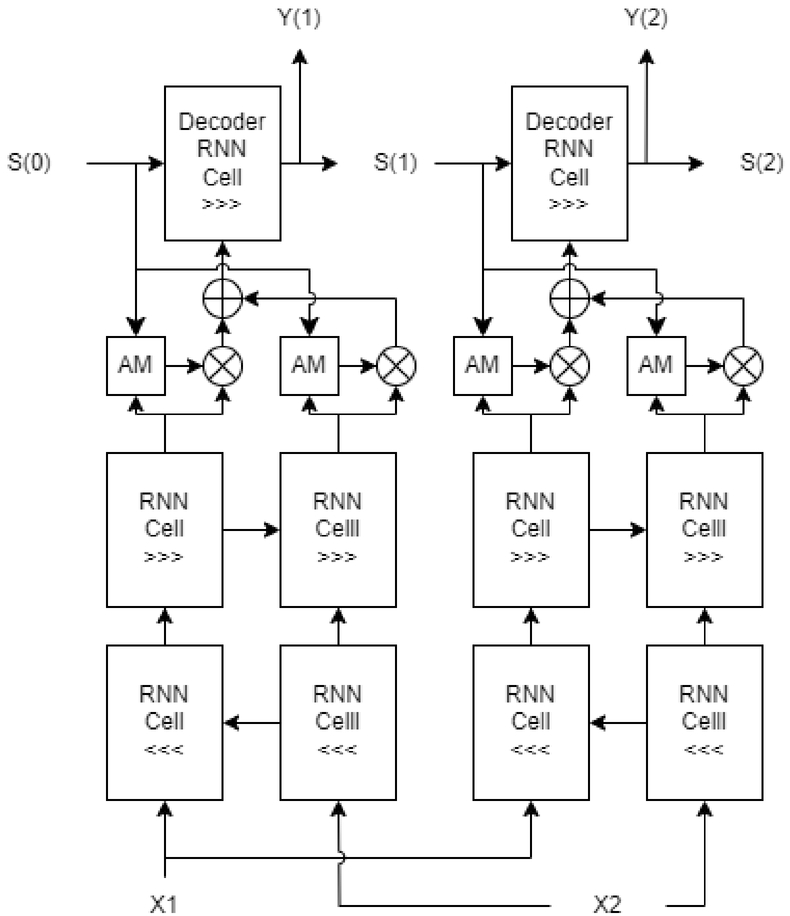
s(t) → Attention „Scorer“ Mechanism → a(t,1)

h(1) → Attention „Scorer“ Mechanism

How to draw a computational graph for this?

- 1 input feature X of sequence lenght 2: So two encoder timesteps (X1, X2)
- 1 RNN cell per RNN layer both in encoder and decoder
- 2 decoder timesteps S(1) and S(2)
- AM is the attention mechanism, can be a small neural net

I simplified it of course, because I did not show the inner workings of the RNN Cells here.



Does it work?

Yes!

- Figure on the right is from the original paper, and shows how the “performance” of the Attention RNN stays stable, while regular RNNs get worse when dealing with longer sentences

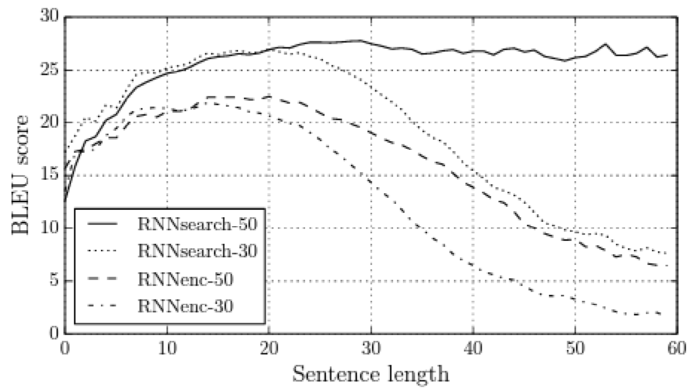


Figure 2: The BLEU scores of the generated translations on the test set with respect to the lengths of the sentences. The results are on the full test set which includes sentences having unknown words to the models.

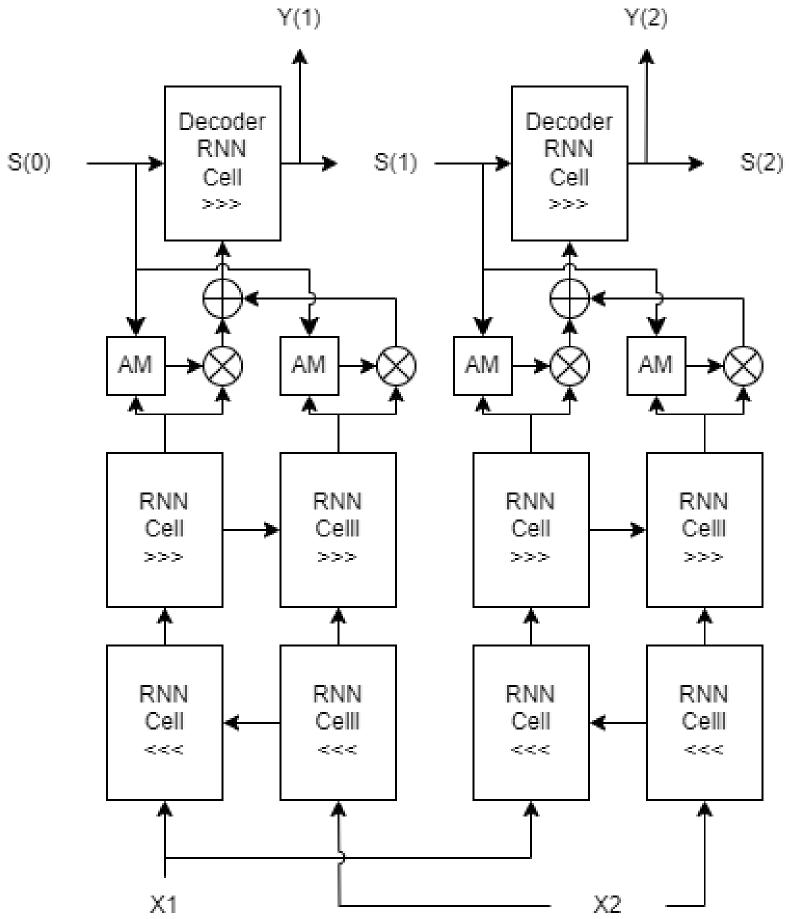
[Neural Machine Translation by Jointly Learning to Align and Translate – 2014 - Bahdanau, Cho, Bengio]

Is it fast?

/informatik & security

NO!

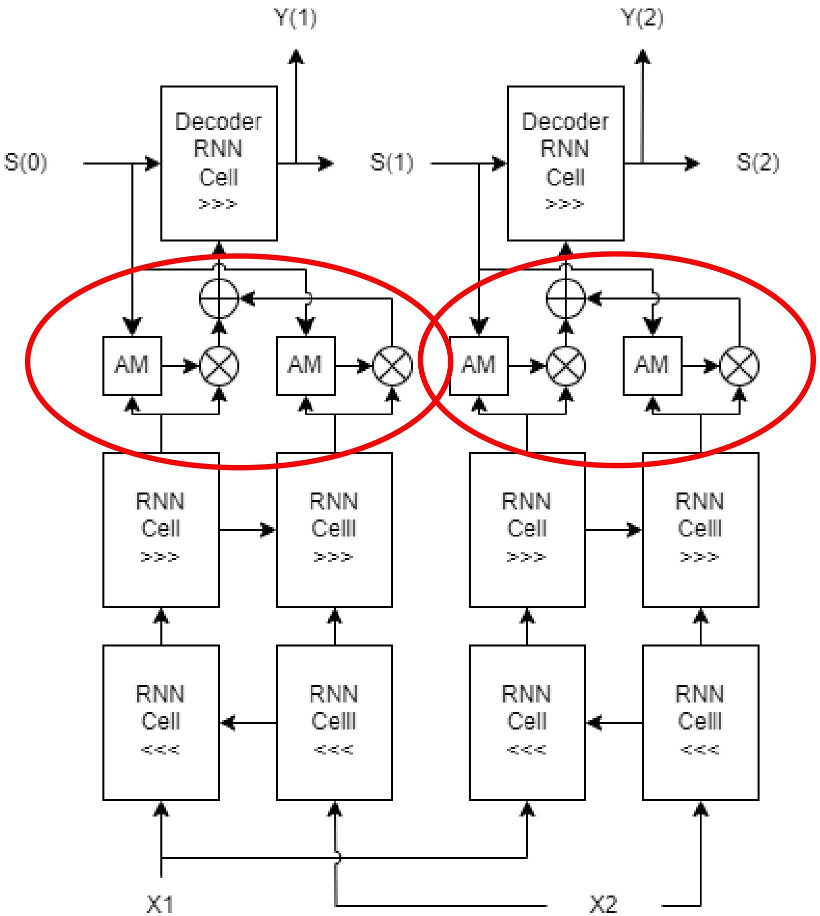
- Performance is bad, because it is basically „a loop in a loop“
- For each step of the decoder sequence, we „search“ through the input sequence 2 times (back and forth)



Attention weights are calculated in parallel

A positive point is that the attention weights can be calculated <u>in parallel</u>, which can be done very fast, if you have sufficient hardware

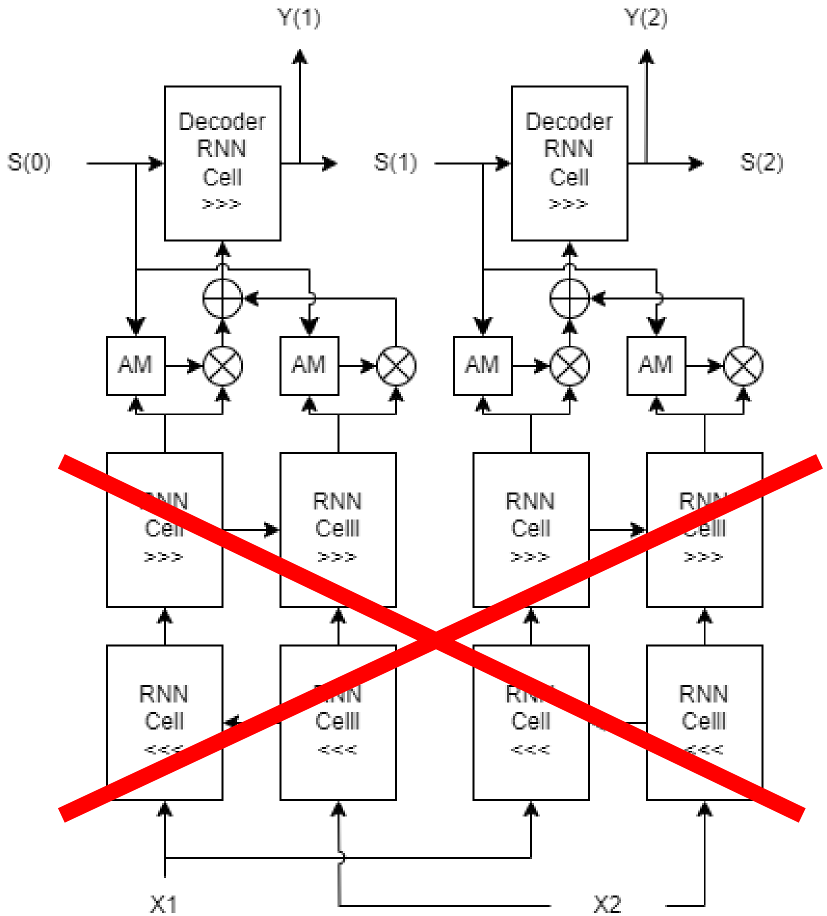
- It is only the RNN parts that are sequentially reading in the input



Attention weights are calculated in parallel

A positive point is that the attention weights can be calculated <u>in parallel</u>, which can be done very fast, if you have sufficient hardware

- It is only the RNN parts that are sequentially reading in the input



If only we could do something about this pesky double looping at the bottom here…

Attention is all you need!

/informatik & security

Let‘s get rid of recurrence althogether!

- The Transformer architecture still takes in sequences of varying length, but processes all input tokens in parallel
- It does not use recurrence (in the input), but relies solely on the attention mechanism to tie the input sequence to the output
- Output is still generated sequentially

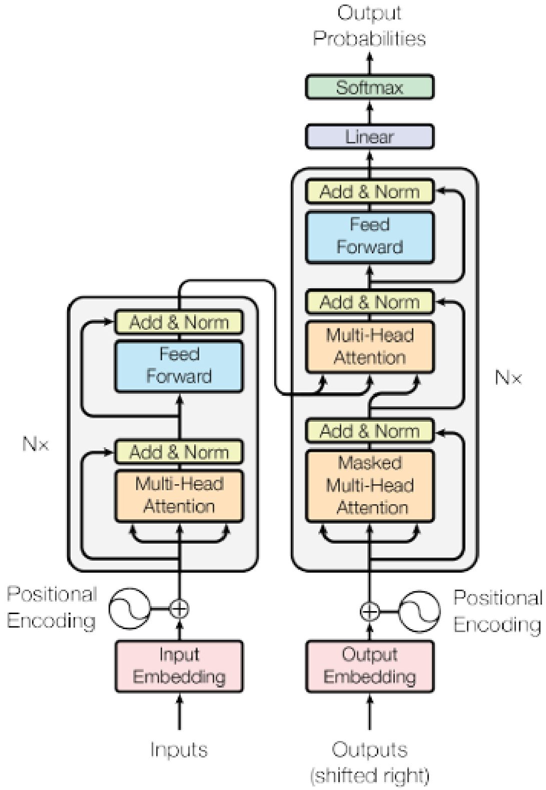


Figure 1: The Transformer - model architecture.

[Attention is all you need – 2017 – Vaswani et. al.]

The end

But that is a story for another time.

Now it is time for bed…

… I mean programming!

Let us implement LSTMs and GRUs in Python!

(generated with AI)