

# Natural Language Understanding, Generation, and Machine Translation

## Lecture 8: Transformers

---

Shay Cohen

based on slides by Frank Keller

29 January 2025 (week 3)

School of Informatics

University of Edinburgh

[scohen@inf.ed.ac.uk](mailto:scohen@inf.ed.ac.uk)

The Story so Far

Self-attention

Multi-head Attention

The Transformer

Reading: Vaswani et al. [2017], Bloem [2019].

## The Story so Far

---

# Encoder-Decoder Architecture

When we do MT, we encode a source sentence and then decode it into a target sentence:

**Input:**      Så varför minskar inte vi våra utsläpp?

**Output:**    So why are we not reducing our emissions?

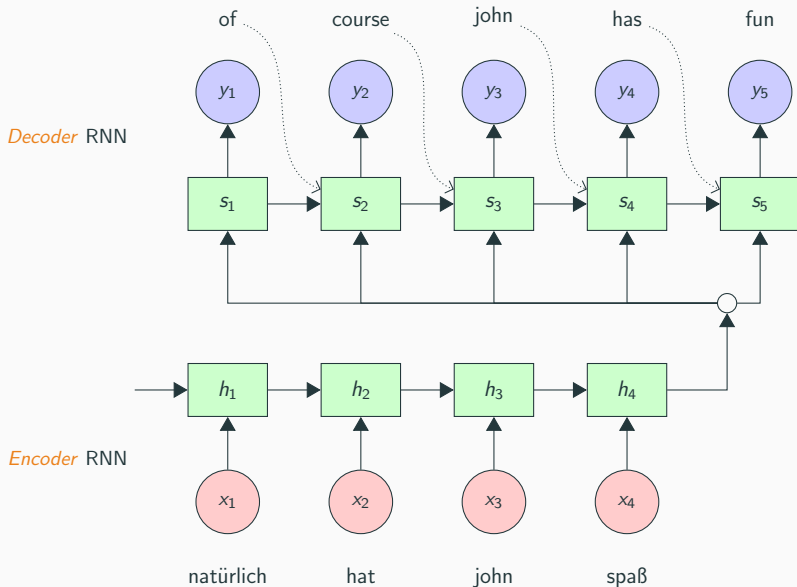
We can model this using an *encoder-decoder* architecture.

Sequence transduction is a common way to formulate NLP tasks:

- question answering
- syntactic and semantic parsing
- generation from a database
- image description

Often RNNs are used for both the encoder and the decoder.

# Encoder-Decoder Architecture



# Encoder-Decoder with Attention

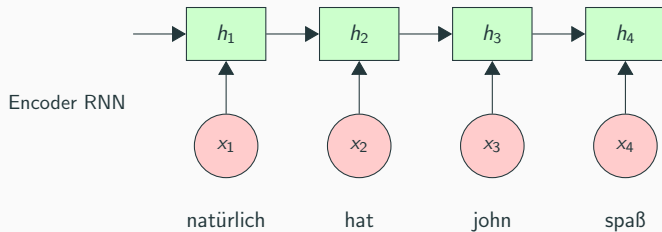
This works pretty well, but:

- the hidden representation is a *bottleneck*: it has to represent a sentence of any length, but its size is fixed;
- RNNs and their variants suffer from vanishing gradients;
- the encoder representation also has a *recency bias*: mostly represents final words of the input;
- target words whose corresponding source words are at the beginning therefore more difficult to generate correctly.

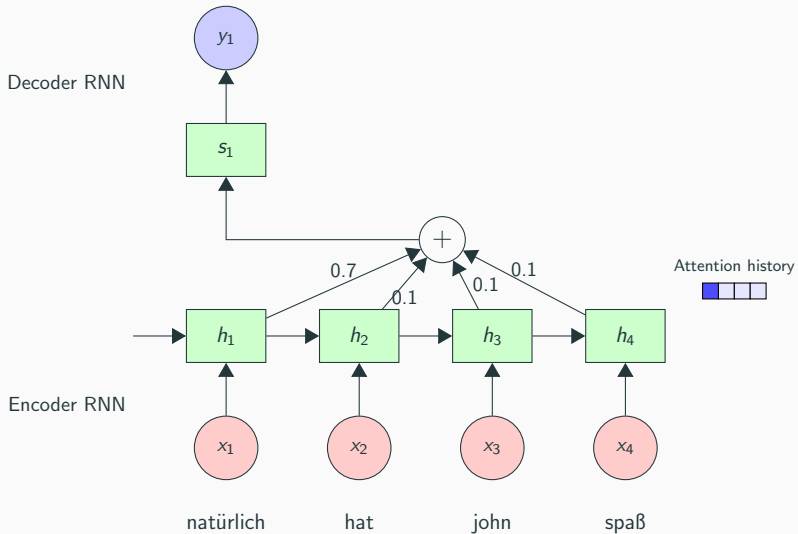
*Solution*: learn which input words are important for the decoder.

# Encoder-Decoder with Attention

Decoder RNN

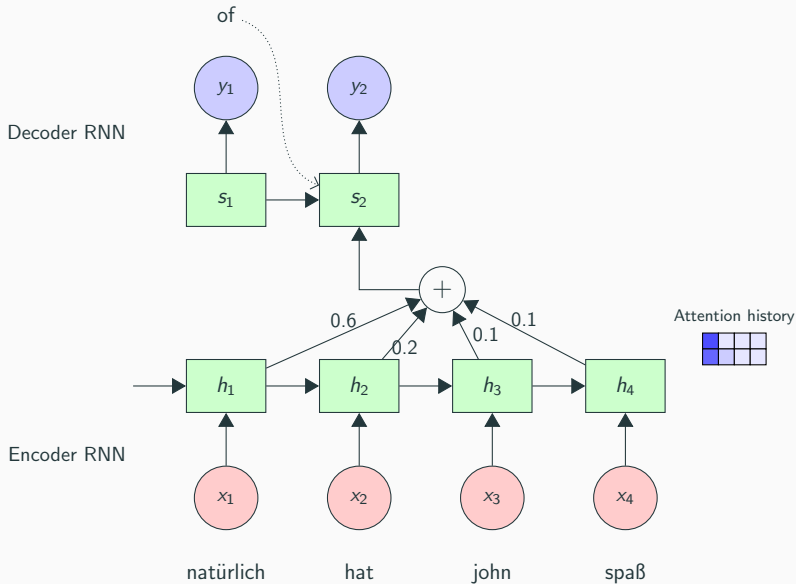


# Encoder-Decoder with Attention

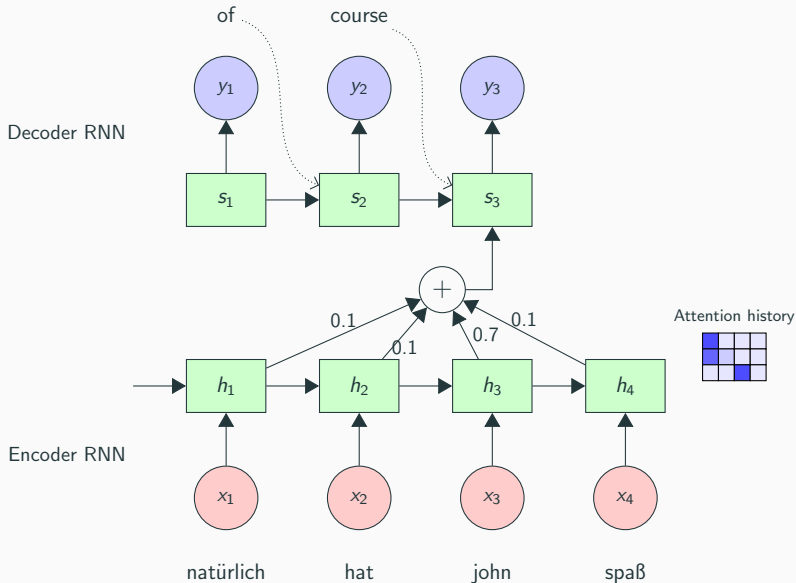




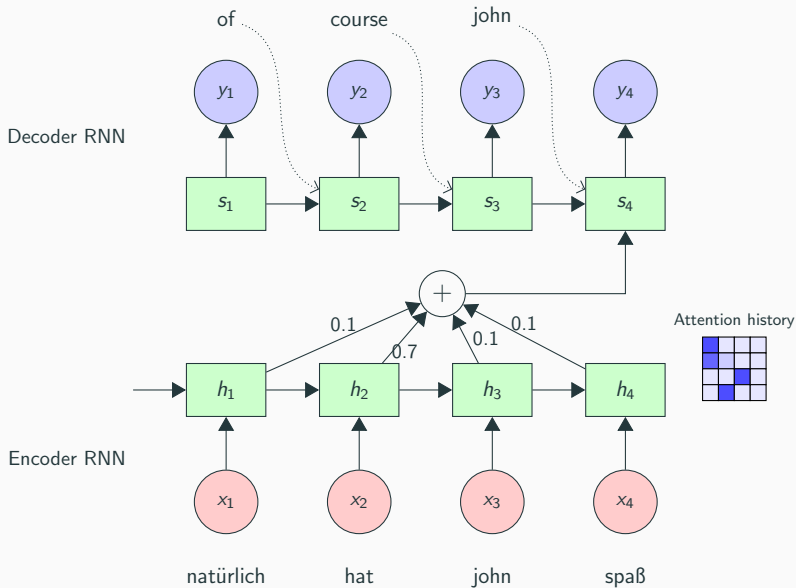
# Encoder-Decoder with Attention



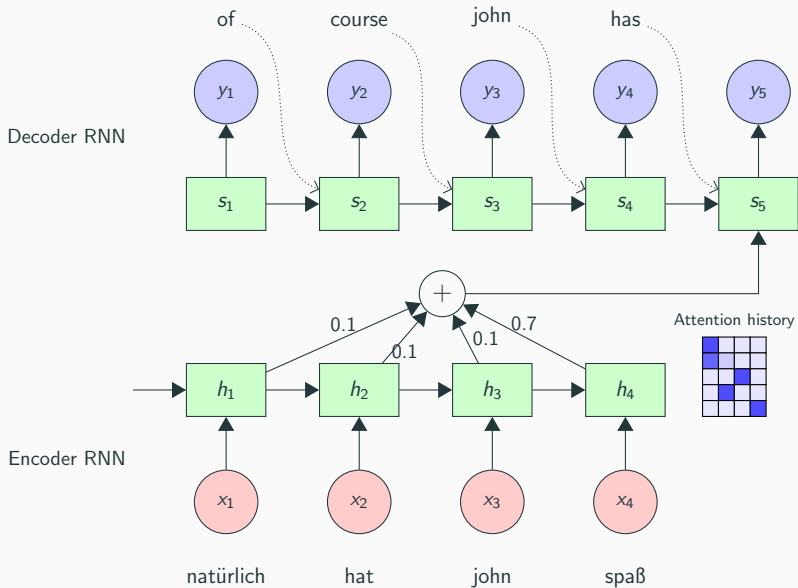
# Encoder-Decoder with Attention



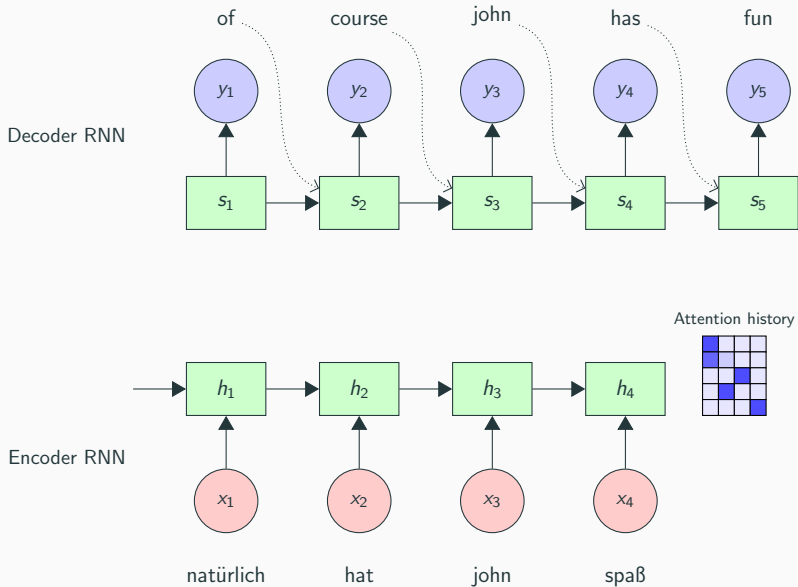
# Encoder-Decoder with Attention



# Encoder-Decoder with Attention



# Encoder-Decoder with Attention



# Encoder-Decoder with Attention

In the decoder softmax, the *context* is now a *weighted average* of source hidden state vectors:

$$P(y_i \mid y_1, \dots, y_{i-1}, x_1, \dots, x_{|x|}) = \text{softmax}(\mathbf{W} \text{concat}(\mathbf{s}_i, \mathbf{c}_i) + \mathbf{b})$$

$$\mathbf{c}_i = \sum_{j=1}^{|x|} \alpha_{ij} \mathbf{h}_j$$

$\alpha_i$  is a *distribution* over elements of  $x$ . In its simplest form, we can compute it as *dot product attention*:

$$a_{ij} = \mathbf{s}_i \cdot \mathbf{h}_j$$

$$\alpha_i = \text{softmax}(\mathbf{a}_i)$$

# Encoder-Decoder with Attention

Observation:

- Attention is a powerful mechanism; maybe we can use it to simplify the encoder and the decoder?
- Removing the RNNs would make our model a lot more efficient and scalable (larger models, more data).

# Encoder-Decoder with Attention

Observation:

- Attention is a powerful mechanism; maybe we can use it simplify the encoder and the decoder?
- Removing the RNNs would make our model a lot more efficient and scalable (larger models, more data).

But first, we need to make attention more complicated:

- We use two types of attention: *self-attention* and *multihead attention*.
- We split up the attention computation into *key*, *value*, and *query* vectors.



# Self-attention

---

# Self-attention

The original Vaswani et al. [2017] paper is a bit impenetrable. We will instead follow the tutorial by Bloem [2019].

Self-attention is what we get when compute attention over the input sequence. Let  $\mathbf{x}_1, \dots, \mathbf{x}_t$  be the input vectors and  $\mathbf{y}_1, \dots, \mathbf{y}_t$  be the output vectors:

$$\mathbf{y}_i = \sum_j w_{ij} \mathbf{x}_j$$

We now compute the attention weight as the dot-product of each input token with every other token.

$$w'_{ij} = \mathbf{x}_i \cdot \mathbf{x}_j$$
$$\mathbf{w}_i = \text{softmax}(\mathbf{w}'_i)$$

Note that the attention weight is now called  $\mathbf{w}'$  and the attention distribution  $\mathbf{w}$  (rather than  $\mathbf{a}$  and  $\alpha$ ).

# Self-attention

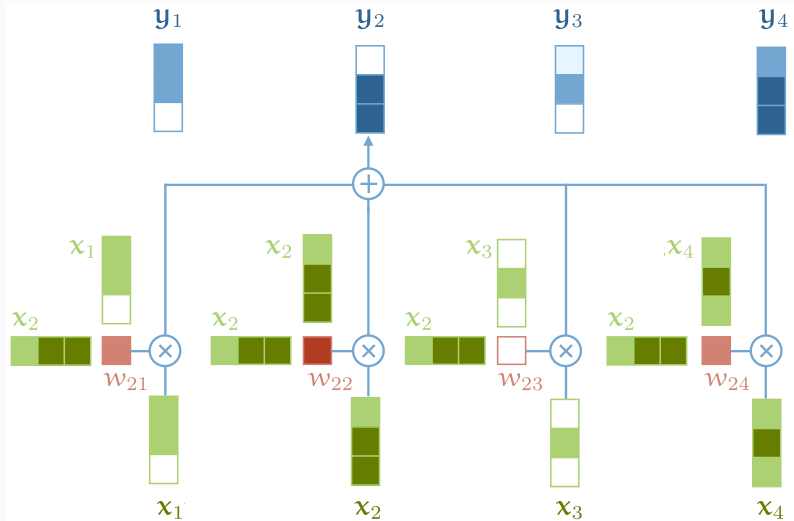


Figure from [Bloem \[2019\]](#).

# Self-attention

Why is self-attention useful?

- The dot product returns large values when the two vectors are similar. The softmax normalizes the resulting vectors.
- The output  $\mathbf{y}_i$  is the weighted sum of all input vectors, weighted by their similarity with input  $\mathbf{x}_i$ .
- We have no trainable parameters! We're relying on the input vectors  $\mathbf{x}_j$  being good representations for our task.
- The input is a *position invariant*, i.e., we have no way to represent word order (unlike in an RNN).

# Self-attention

Why is self-attention useful?

- The dot product returns large values when the two vectors are similar. The softmax normalizes the resulting vectors.
- The output  $\mathbf{y}_i$  is the weighted sum of all input vectors, weighted by their similarity with input  $\mathbf{x}_i$ .
- We have no trainable parameters! We're relying on the input vectors  $\mathbf{x}_j$  being good representations for our task.
- The input is a *position invariant*, i.e., we have no way to represent word order (unlike in an RNN).

Example: represent words as embeddings:

word sequence:            the, cat, walks, on, the, street

input (embeddings):     $\mathbf{x}_{the}$ ,  $\mathbf{x}_{cat}$ ,  $\mathbf{x}_{walks}$ ,  $\mathbf{x}_{on}$ ,  $\mathbf{x}_{the}$ ,  $\mathbf{x}_{street}$

output:                     $\mathbf{y}_{the}$ ,  $\mathbf{y}_{cat}$ ,  $\mathbf{y}_{walks}$ ,  $\mathbf{y}_{on}$ ,  $\mathbf{y}_{the}$ ,  $\mathbf{y}_{street}$

## More Advanced Self-attention

Every input vector  $\mathbf{x}_i$  is used in three ways in self-attention:

- *Query*: compare  $\mathbf{x}_i$  to every other vector to compute attention weights for its own output  $\mathbf{y}_i$ .
- *Key*: compare  $\mathbf{x}_i$  to every other vector to compute attention weights for the other outputs  $\mathbf{y}_j$ .
- *Value*: use  $\mathbf{x}_i$  in the weighted sum to compute every output vector based on these weights.

We can make attention more flexible by assuming separate, trainable weights for each of these roles: the  $k \times k$  weight matrices  $W_q$ ,  $W_k$ , and  $W_v$  ( $k$ : dimensionality of  $\mathbf{x}$  and  $\mathbf{y}$ ).

## More Advanced Self-attention

Now we compute attention as follows:

$$\mathbf{q}_i = W_q \mathbf{x}_i \quad \mathbf{k}_i = W_k \mathbf{x}_i \quad \mathbf{v}_i = W_v \mathbf{x}_i$$

$$w'_{ij} = \mathbf{q}_i \cdot \mathbf{k}_j$$

$$\mathbf{w}_i = \text{softmax}(\mathbf{w}'_i)$$

$$\mathbf{y}_i = \sum_j w_{ij} \mathbf{v}_j$$

Now the self-attention layer has parameters that allow it to modify the incoming vectors to suit the three roles they must play.

# More Advanced Self-attention

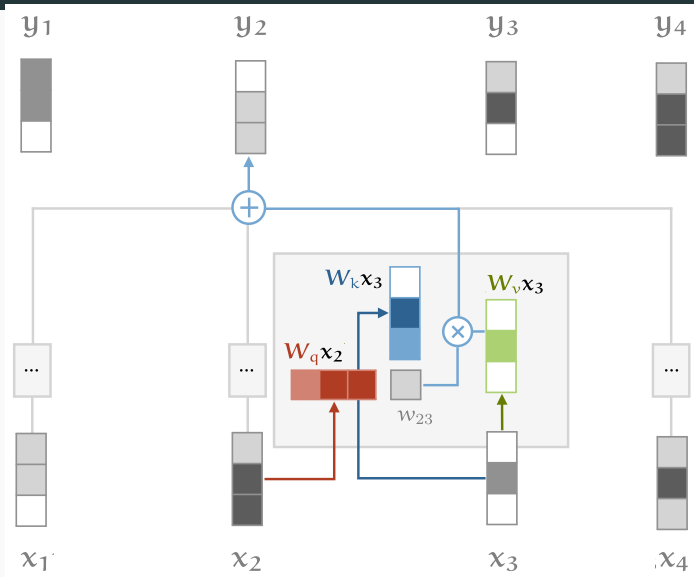


Figure from [Bloem \[2019\]](#).



# Scaling the Dot Product

- The softmax function can be sensitive to very large input values;
- this kills the gradient and can slow down learning or cause it to stop;
- it helps to scale the dot product back to stop the inputs to the softmax function from growing too large:

$$w'_{ij} = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{k}}$$

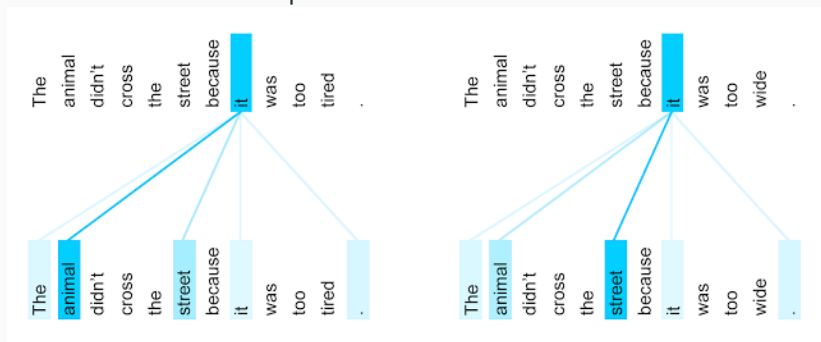
# Multi-head Attention

---

# Multi-head Attention

*Multi-head attention* is able to jointly attend to different parts of the input. If we just have a single head, have to average.

For example, one head could attend to the subject of a verb, another one to its object. Or different heads could attend to different referents of a pronoun.



# Multi-head Attention

We use separate weight matrices for each head:  $W_q^r$ ,  $W_k^r$ ,  $W_v^r$ , where  $r$  is an index over heads.

For the input  $\mathbf{x}_i$ , each attention head now produces a different output  $\mathbf{y}_i^r$ . We concatenate them and pass them through a linear transformation to reduce the dimensions to  $k$ . Two variants:

- *Narrow self-attention*: cut the input into chunks of size  $k/h$  ( $h$ : number of heads), use weight matrices of size  $k/h \times k/h$ .
- *Wide self-attention*: use weight matrices that cover the whole input for each head, i.e., of size  $k \times k$ .

# The Transformer

---

# The Transformer

The multi-head self attention mechanism needs to be integrated into a larger architecture to be useful:

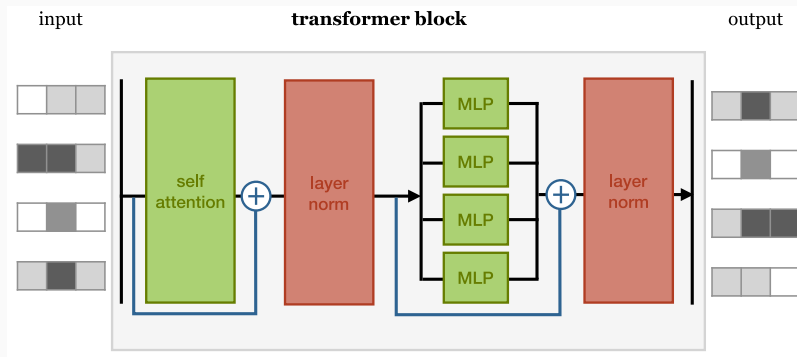


Figure from [Bloem \[2019\]](#).

# The Transformer

- The blue connections are residual connections. They prevent the gradients from getting too large or small.
- Layer normalization makes training faster.
- The feedforward layer applies a single MLP independently to each input vector.

# Example: Movie Classification

Transformer to classify a movie review as positive or negative:

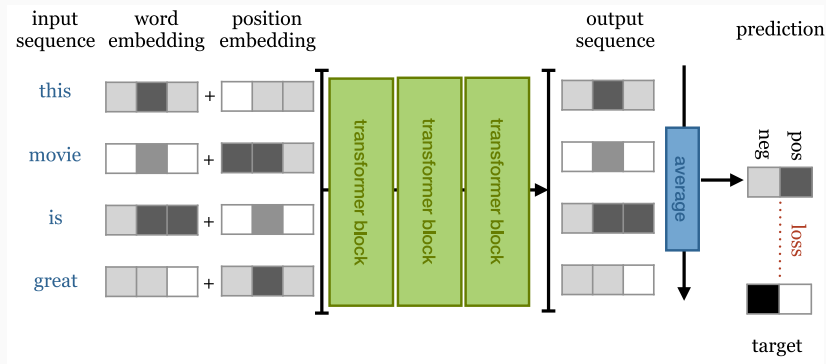


Figure from [Bloem \[2019\]](#).



## Example: Movie Classification

- At the core is a chain for transformer blocks.
- *Input:* represent words as embeddings; these are then combined with position encodings.
- Recall that attention is position invariant, i.e., the transformer cannot directly represent word order.
- *Output:* apply average pooling to the final output sequence, map the result to a softmaxed class vector.

# Position Encodings

We choose a function  $f : \mathbb{N} \rightarrow \mathbb{R}^k$  that maps positions to real valued vectors, and let the network learn how to interpret these.

The choice of encoding function is a hyperparameter. Vaswani et al. [2017] use:

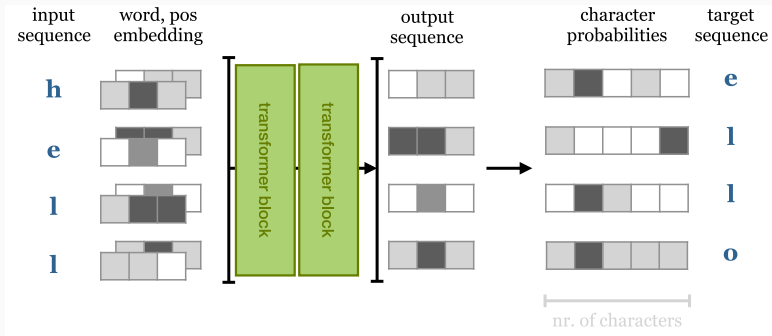
$$PE_{(pos, 2i)} = \sin(pos/10000^{2i/k})$$
$$PE_{(pos, 2i+1)} = \cos(pos/10000^{2i/k})$$

where  $pos$  is the position and  $i$  is an index over dimensions.

The positions encoding is then added to the input embedding (both are of dimensionality  $k$ ).

# Masking

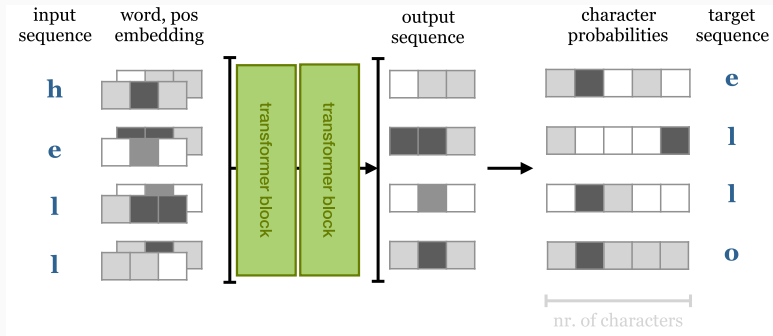
What if we want to *generate text* as an output? Then we need an *autoregressive* model:



For this, we need to ensure that the transformer cannot look forward in the input when generating the output.

# Masking

What if we want to *generate text* as an output? Then we need an *autoregressive* model:

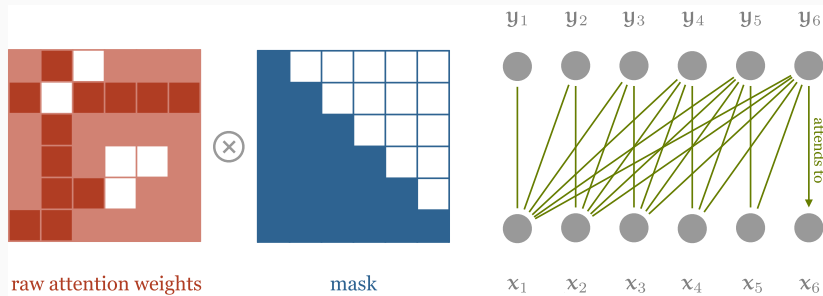


For this, we need to ensure that the transformer cannot look forward in the input when generating the output.

Else it would just generate a copy of the input! *We need masking.*

# Masking

We apply a mask to the matrix of dot products, before the softmax is applied. This disables all elements above the diagonal:



The model can no longer look forward and just copy the upcoming input; it behaves like an RNN!

Figure from [Bloem \[2019\]](#).

# Summary

- The transformer overcomes the problem with recurrent connections (serial bottleneck).
- It can model long-range dependencies using self-attention.
- Position encodings are needed to capture word order.
- Transformer blocks can be stacked to make models more powerful; only limited by compute and memory.
- The transformer has only two sources of non-linearity: the feedforward layer and the softmax in the self-attention.
- Masking makes it possible to use transformers in an autoregressive way for sequence-to-sequence tasks.

Now we have covered most of the key modeling ideas in the course!

**Next class:** Word embeddings with and without transformers.

Peter Bloem. Transformers from Scratch.

<http://www.peterbloem.nl/blog/transformers>, 2019.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008, Red Hook, NY, 2017. Curran Associates.