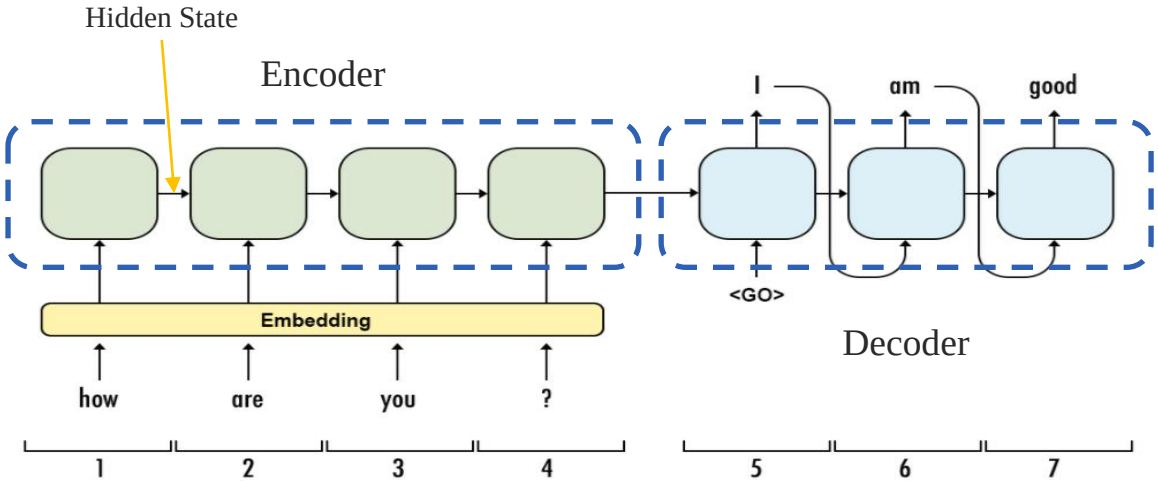
NLP: Advanced Architectures

Yuanzhi Zhu
SiDNN

26/Apr/2022

NLP: Seq2Seq Problems

RNN Encoder Decoder Architecture



Limitations:

- Long Sequences
- Bad Parallelization

For NLP problems like *Question Answering, Language Modeling, or Translation,* RNN Encoder Decoder Architecture are very useful.

As you can see in this picture, where the model is used for a Question Answering system, the tokens in input sequence are first embedded using the embedding technique introduced last week.

For the Encoder part, at each time step, the hidden state from last step and a new token are sent to the encoder and generate a new hidden state.

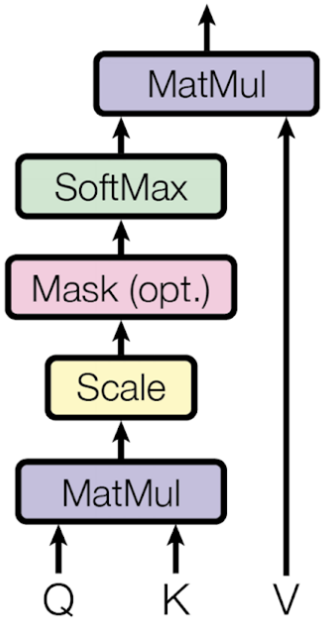
For the decoder, the final hidden state from the encoder and a special start token <GO> are the initial input. In the next steps, the input will be the generated token and the hidden state from last step.

However, there are limitations of this architecture: the information from the very beginning token of a long sequences will get lost.

Also the encoder and decoder have to perform sequentially, result in a significant resource bottleneck on training such a model.

Attention: Better Info Mixer

Scaled Dot-Product Attention



Input: Queries Keys Values (Q K V)

Output: computed as a weighted sum of the values

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

To solve these problems, attention is used to replace the RNN structure, because attention mechanism can better capture the information even for long distance.

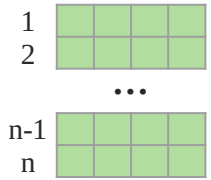
Here is a illustration of scaled dot-product attention. There are also other attention like additive attention, but their main function is to mix information within a sequence.

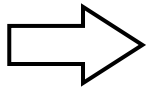
The input is a tuple of Queries Keys Values and the output is a weighted sum of the values as computed in this formula.

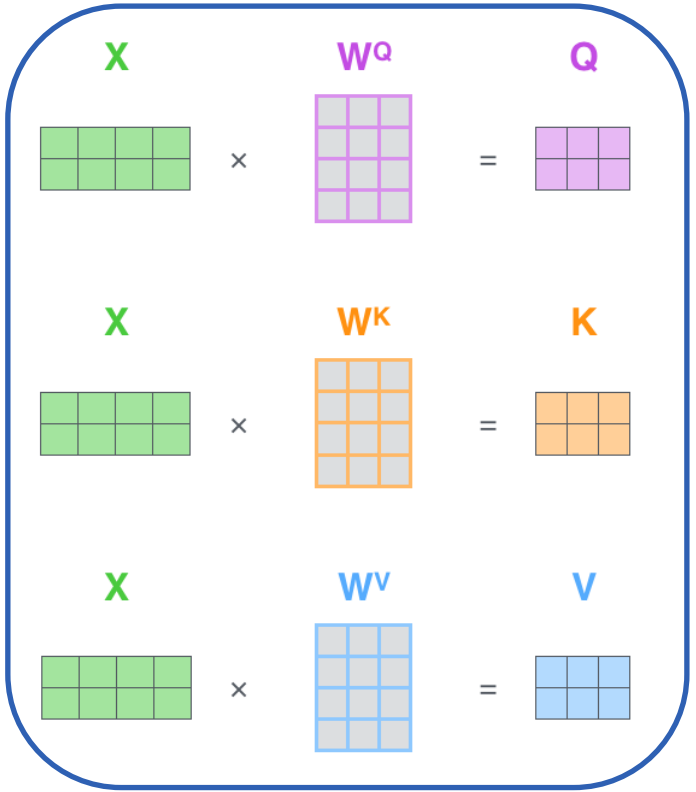
The weight, is a probability distribution from softmax calculated using queries and keys.

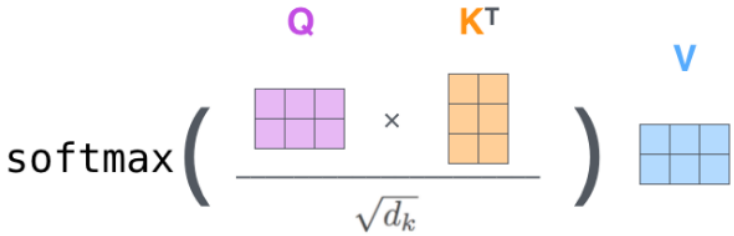
Self-Attention

Input Sequence X:









Now I would like to give you an example of self-attention to help you understand how it is calculated.

First we represent the input sequence as matrix X with size n times d, where n is the length of the sequence and d is the dimension of the embedded tokens.

For self-attention, X is copied 2 times, but all three X are combined with different weight matrix and we get the query, key and value matrix.

These new matrices have the same number of rows but different dimension for each token. Then we can calculated the weighted sum of the values as a new matrix.

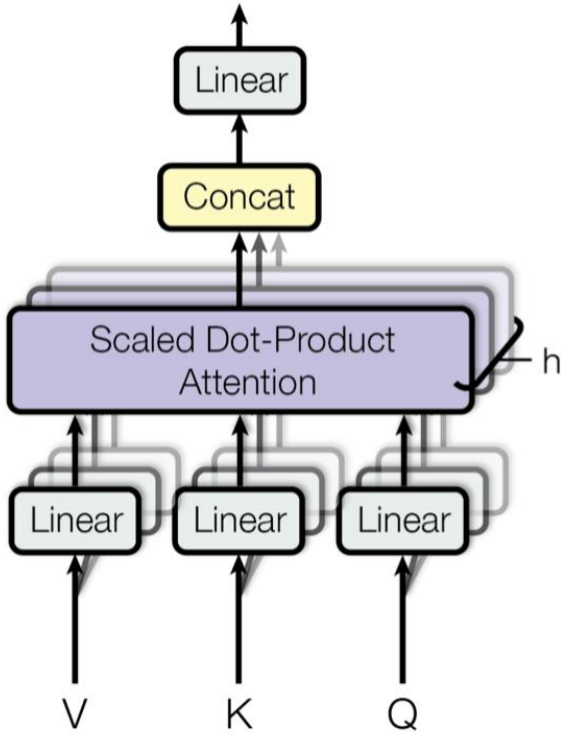
Multi-head Attention ← Multi-channel CNN

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

$$W_i^Q \in \mathcal{R}^{d_{model} \times d_k}$$
$$W_i^K \in \mathcal{R}^{d_{model} \times d_k}$$
$$W_i^V \in \mathcal{R}^{d_{model} \times d_v}$$
$$W^O \in \mathcal{R}^{hd_v \times d_{model}}$$
$$hd_k = hd_v = d_{model}$$

$$\text{output dimension} = \mathcal{R}^{n \times d_{model}}$$

All heads are initialized randomly to learn different attentions

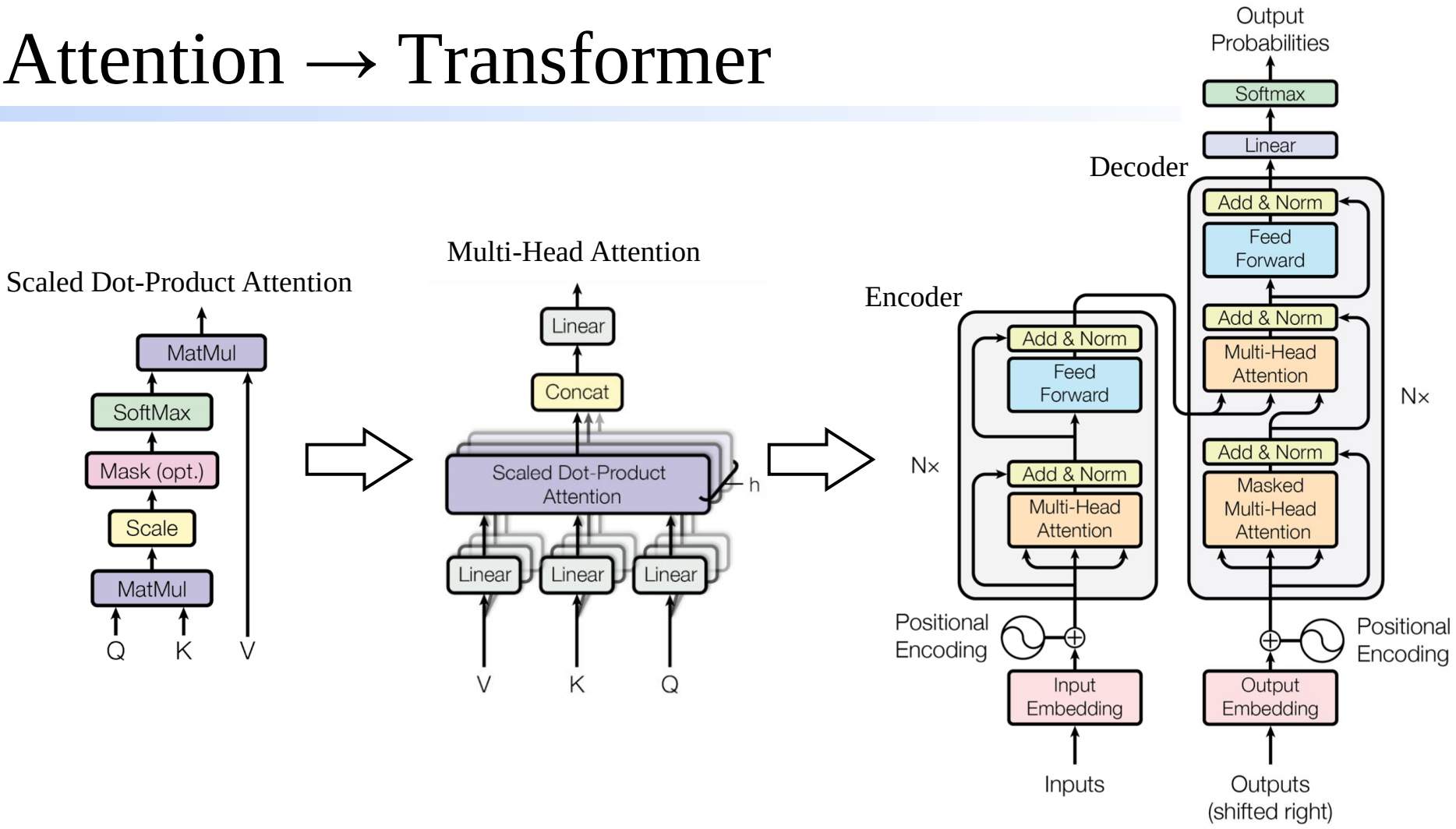


Similar to Multi-channel CNN, Multi-head Attention is proposed to learn different attentions.

Each head is a set of weight matrices acting on the input Value, Key and Query matrices.

These weight matrices are Linear transform to lower dimension, then we concatenate these attentions and use another linear layer to get an attention with the same dimension of result from simple attention.

Attention → Transformer



Given this attention mechanism, which solves the problem in RNN, we want to use this attention module to construct a new encoder-decoder module.

As you can see in the pictures, on the right side is the so called transformer architecture

And there are three places multi-head attention is used in the transformer architecture:

- Encoder multi-head self attention(left)
- Decoder masked multi-head self attention(lower right)
- Encoder-Decoder multi-head attention(upper right)

Actually, they are in charge of mixing info in input sequence, mixing info in output sequence and mixing info between input and output sequence

Positional Encoding: order of the sequence

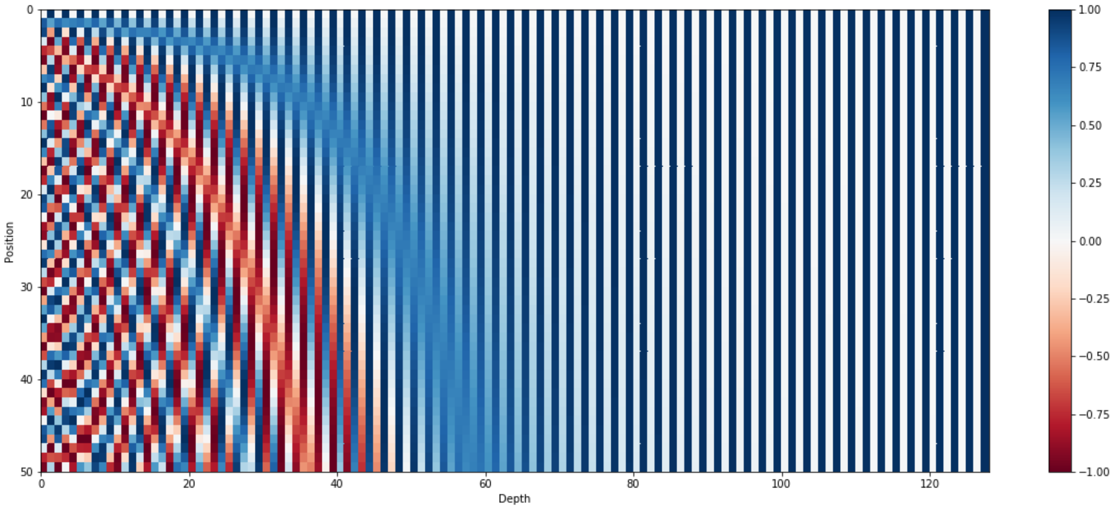
Binary encoding:

0 :	0	0	0	0	8 :	1	0	0	0
1 :	0	0	0	1	9 :	1	0	0	1
2 :	0	0	1	0	2 :	1	0	1	0
3 :	0	0	1	1	11 :	1	0	1	1
4 :	0	1	0	0	12 :	1	1	0	0
5 :	0	1	0	1	13 :	1	1	0	1
6 :	0	1	1	0	14 :	1	1	1	0
7 :	0	1	1	1	15 :	1	1	1	1

From *Attention is all you need*:

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

pos: position
i: dimension



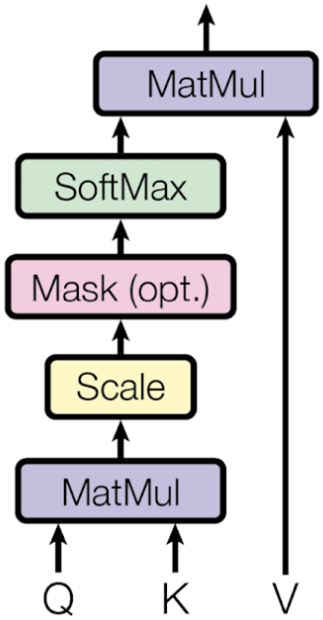
Since transformer is independent of recurrence or convolution, in order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position

- It should output a unique encoding for each time-step (word's position in a sentence)
- Distance between any two time-steps should be consistent across sentences with different lengths.
- Allow the model to easily learn to attend by relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of $PE_{pos.}$

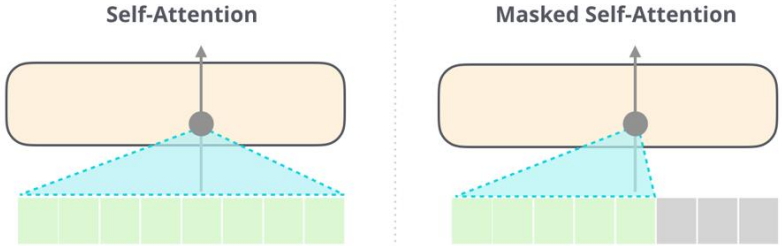
The frequencies are decreasing along the vector dimension. Thus it forms a geometric progression from 2pi to 2000.pi on the wavelengths.

Attention: Masking

Scaled Dot-Product Attention



Masked attention



	-∞	-∞	-∞	-∞
		-∞	-∞	-∞
			-∞	-∞
				-∞

SoftMax

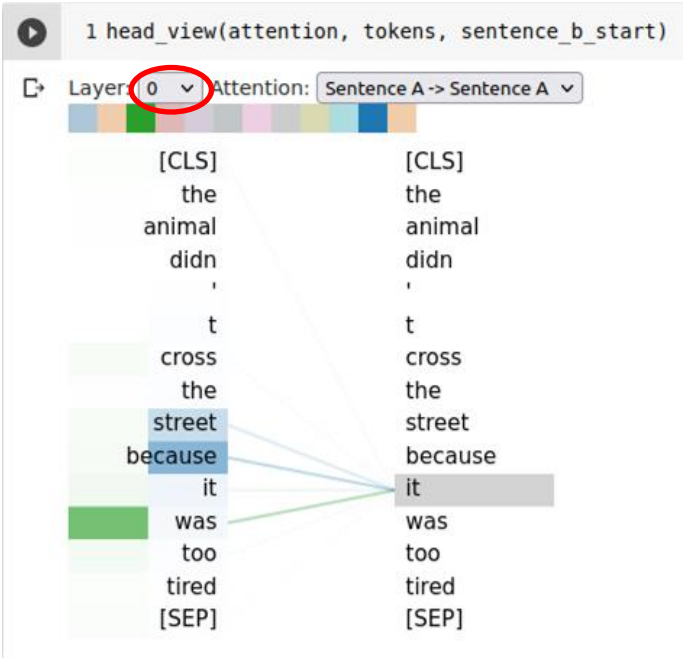
	0	0	0	0
		0	0	0
			0	0
				0

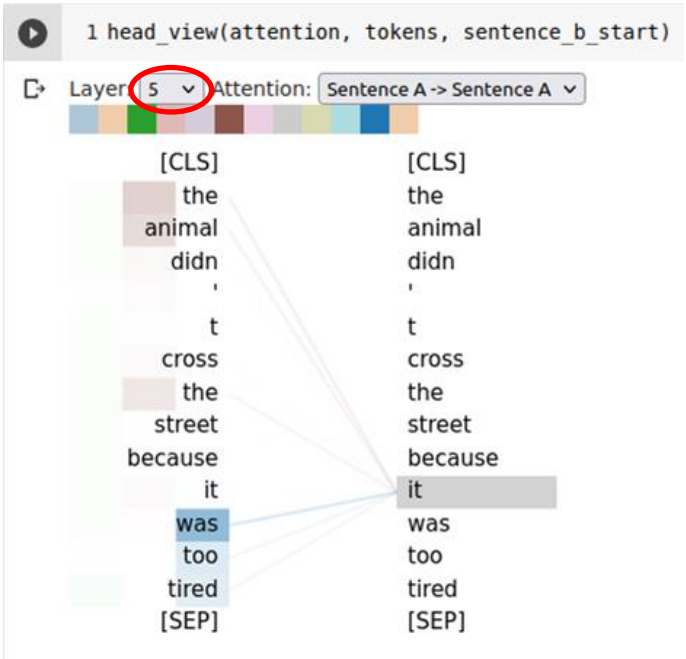
Now we look back to the optional mask sub layer in the attention.

We use this mask sublayer only in the decoder, because the decoder has to output in sequence and couldn’t access to the future tokens.

We mask all the grey element in the product of queries and keys as $-\infty$, then the output probability/weight after softmax is 0, that exactly what we want.

Self-Attention: Example



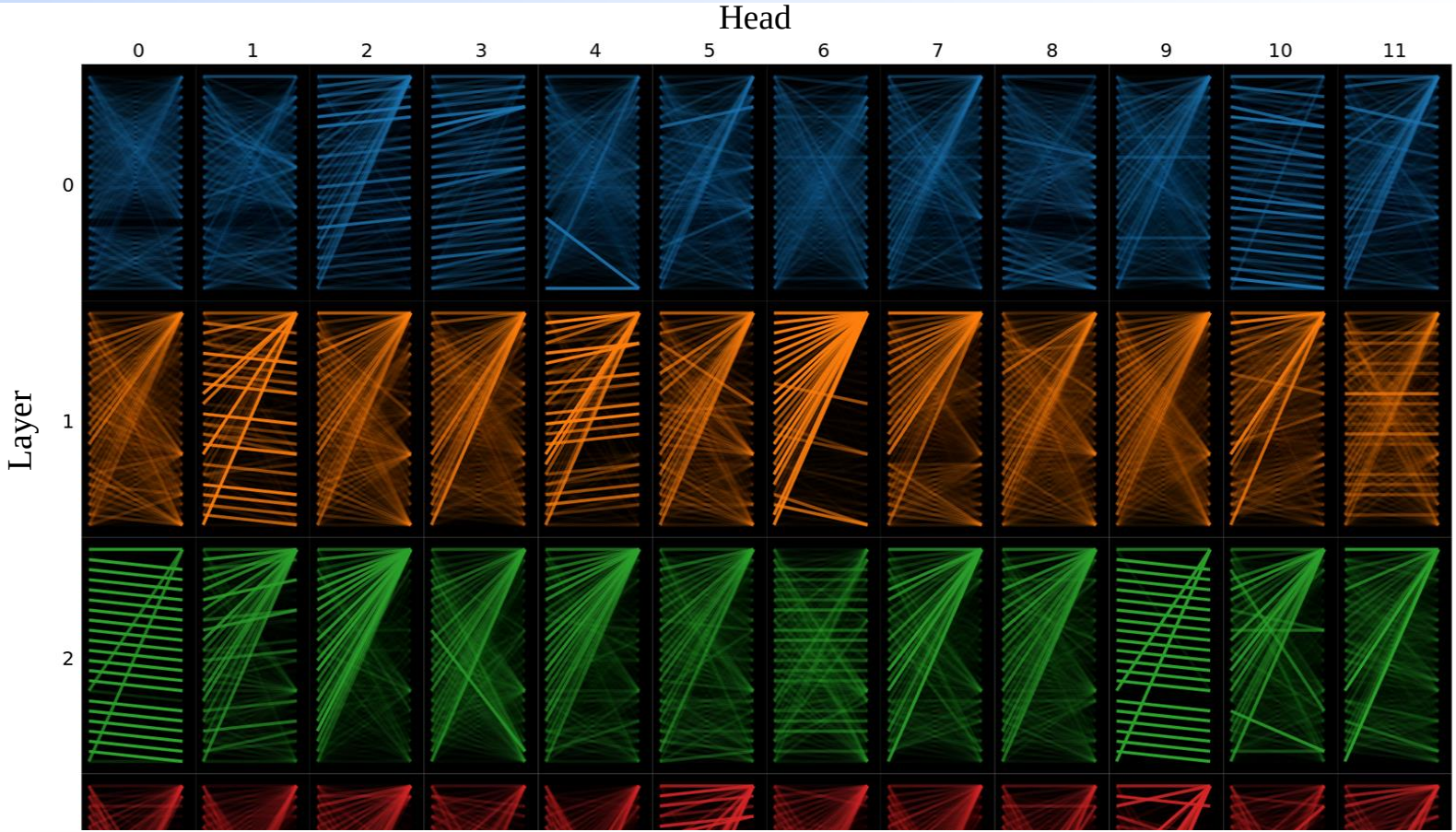


https://colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb
https://github.com/jessevig/bertviz

Now let have a look at a visualized example of multi-head self-attention encoder

Here we can see that in different layers, one word pay different attention to other words

Self-Attention: Example



In this picture, part of all the attentions are visualized

For all 12 heads and the first 3 encoder layers, each head learns different weights.

I believe one can better understand this by comparing to feature maps in CNN