

SOS Video Presentation

Large Language Models

By Koushik

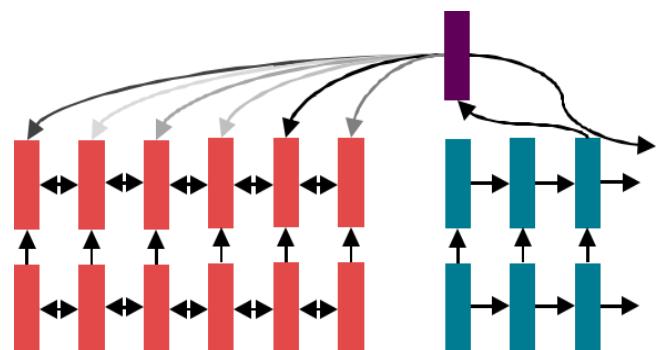
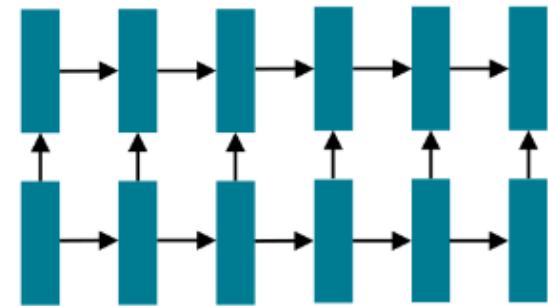
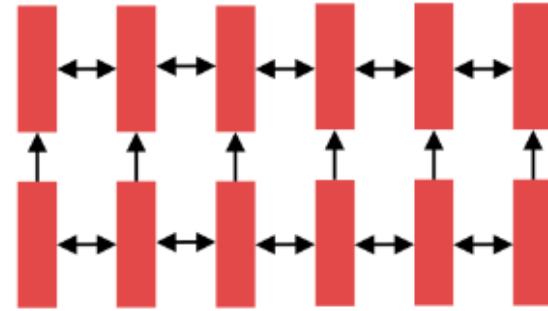
Topic : Self attention and Transformers

Outline :

1. From recurrence (RNN) to attention-based NLP models
2. Introducing the Transformer model and its architecture

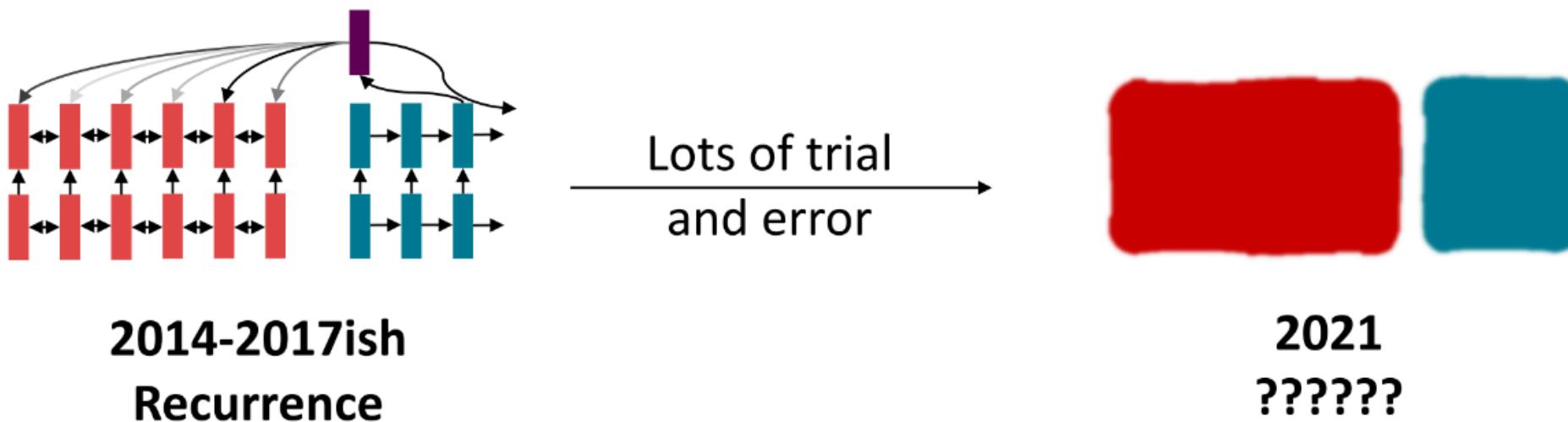
Before Transformers : recurrent models for (most) NLP!

- Circa 2016, the de facto strategy in NLP is to **encode** sentences with a bidirectional LSTM:
(for example, the source sentence in a translation)
- Define your output (parse, sentence, summary) as a sequence, and use an LSTM to generate it.
- Use attention to allow flexible access to memory



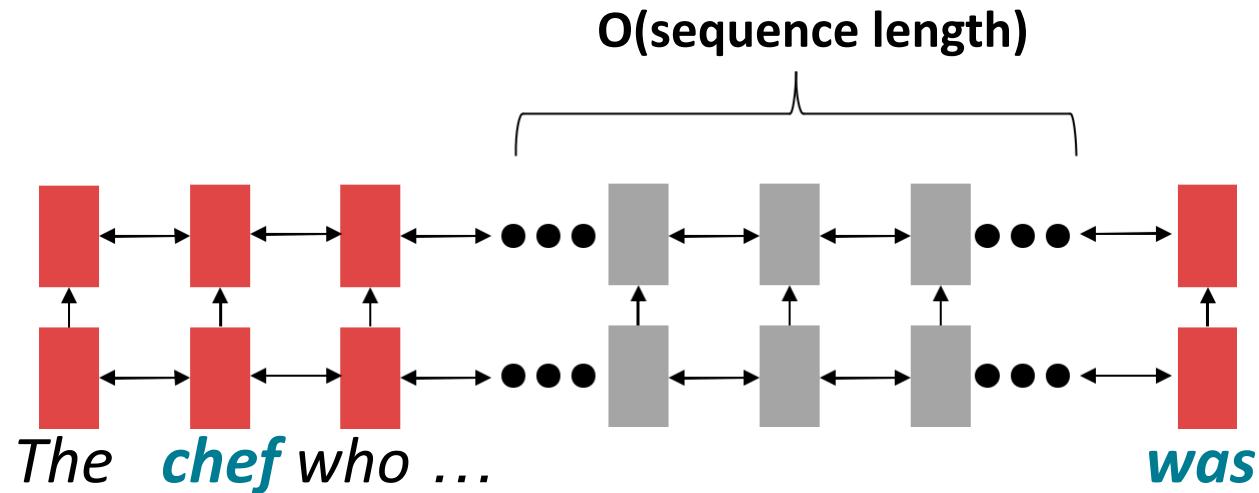
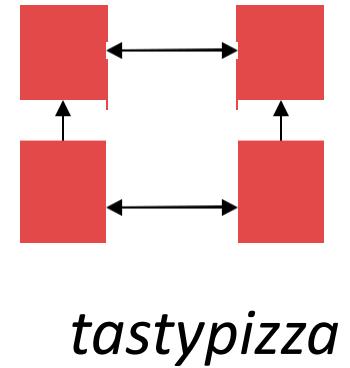
Same goals, different building blocks

- we're **not** trying to motivate entirely new ways of looking at problems (like Machine Translation)
- Instead, we're trying to find the best **building blocks** to plug into our models and enable broad progress.



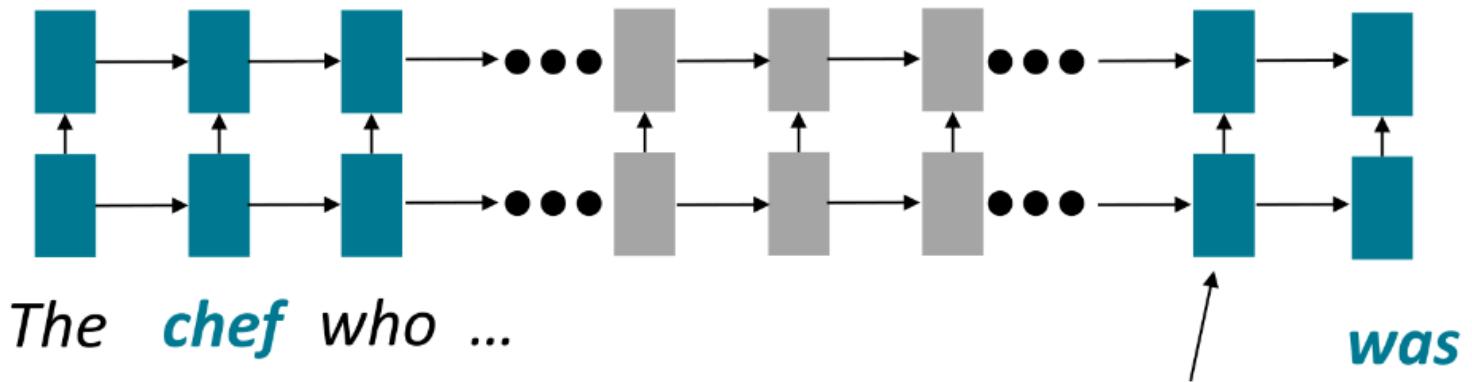
Issues with recurrent models: Linear interaction distance

- RNNs are unrolled “left-to-right”.
- This encodes linear locality: a useful heuristic!
 - Nearby words often affect each other’s meanings
- **Problem:** RNNs take $O(\text{sequence length})$ steps for distant word pairs to interact.



Issues with recurrent models: Linear interaction distance

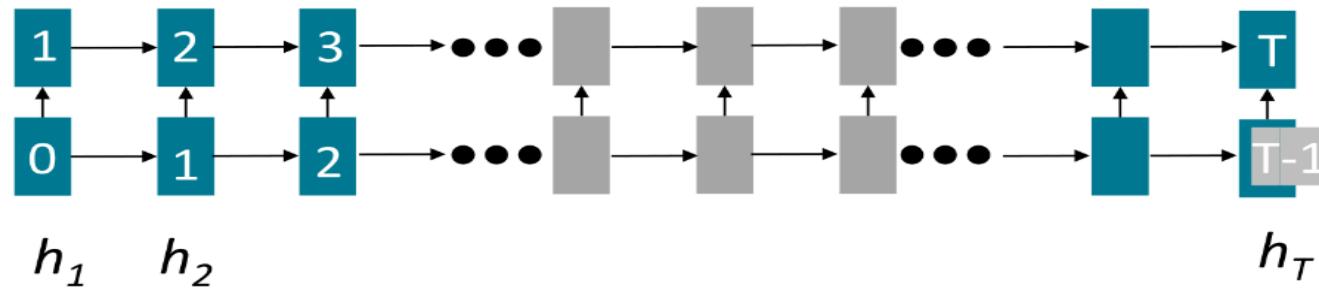
- **O(sequence length)** steps for distant word pairs to interact means:
 - Hard to learn long-distance dependencies (because gradient problems!)
 - Linear order of words is “baked in”; we already know linear order isn’t the right way to think about sentences...



Info of *chef* has gone through
 $O(\text{sequence length})$ many layers!

Issues with recurrent models: Lack of parallelizability

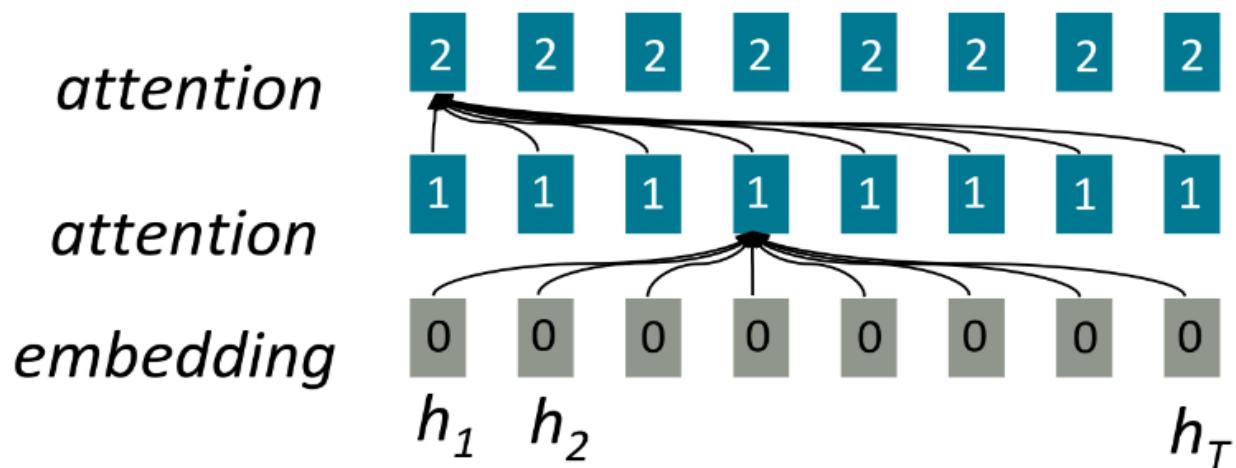
- Forward and backward passes have **O(sequence length)** unparallelizable operations
 - GPUs can perform a bunch of independent computations at once!
 - But future RNN hidden states can't be computed in full before past RNN hidden states have been computed
 - Inhibits training on very large datasets!



Numbers indicate min # of steps before a state can be computed

If not recurrence, then what? How about attention?

- **Attention** treats each word's representation as a **query** to access and incorporate information from **a set of values**.
- Number of unparallelizable operations does not increase sequence length.
- Maximum interaction distance: $O(1)$, since all words interact at every layer!



All words attend to all words in previous layer; most arrows here are omitted

Self-Attention

- Recall: Attention operates on **queries**, **keys**, and **values**.
 - We have some **queries** q_1, q_2, \dots, q_T . Each query is $q_i \in \mathbb{R}^d$
 - We have some **keys** k_1, k_2, \dots, k_T . Each key is $k_i \in \mathbb{R}^d$
 - We have some **values** v_1, v_2, \dots, v_T . Each value is $v_i \in \mathbb{R}^d$
- In **self-attention**, the queries, keys, and values are drawn from the same source.
 - For example, if the output of the previous layer is x_1, \dots, x_T , (one vec per word) we could let $v_i = k_i = q_i = x_i$ (that is, use the same vectors for all of them!)
- The (dot product) self-attention operation is as follows:

$$e_{ij} = q_i^\top k_j$$

Compute **key-query** affinities

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{j'} \exp(e_{ij'})}$$

Compute attention weights from affinities

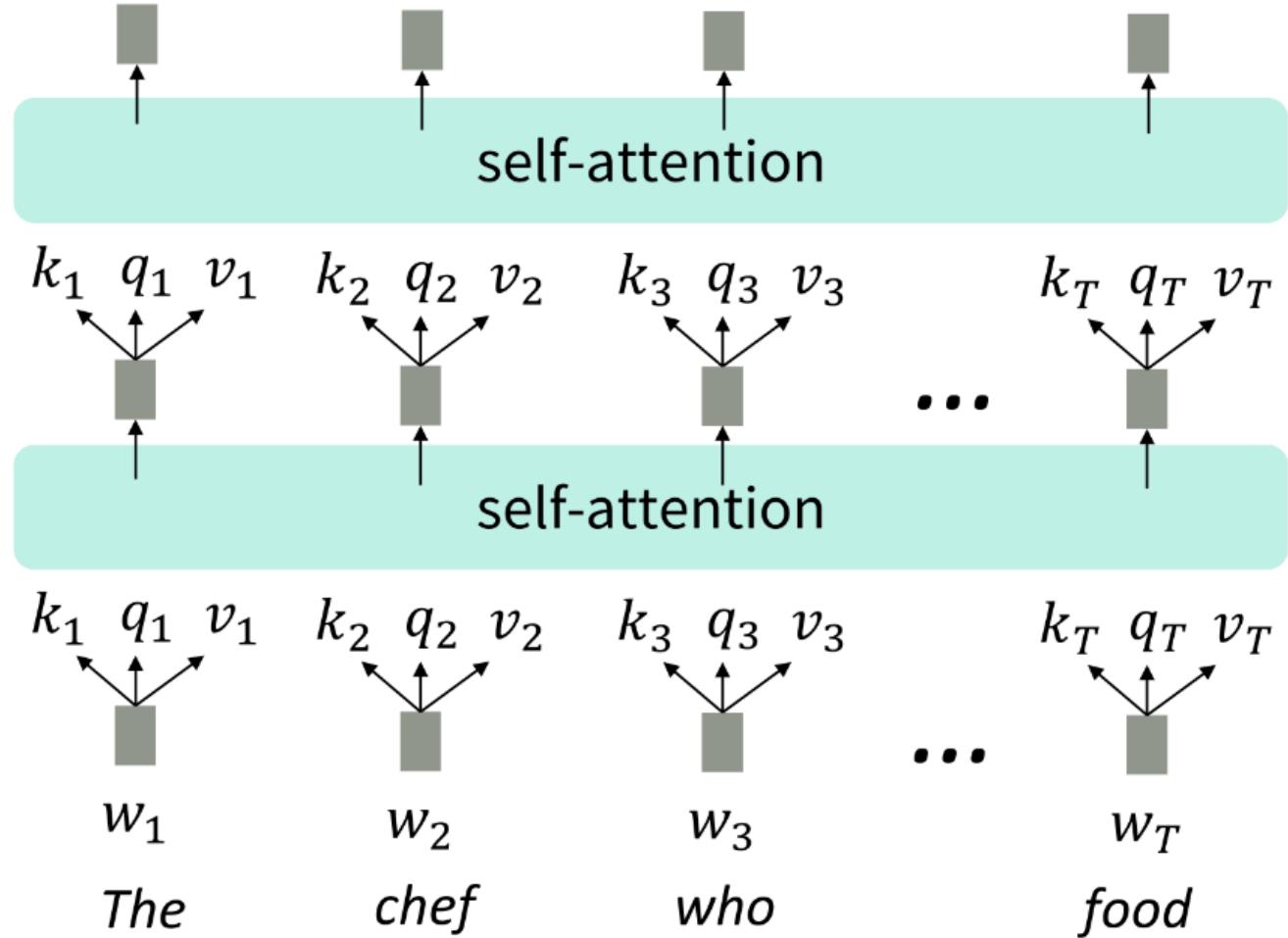
$$\text{output}_i = \sum_j \alpha_{ij} v_j$$

Compute outputs as weighted sum of **values**

The number of queries can differ from the number of keys and values in practice.

Self-attention as an NLP building block

- In the diagram at the right, we have stacked self-attention blocks, like we might stack LSTM layers.
- Can self-attention be a drop-in replacement for recurrence?
- No. It has a few issues, which we'll go through.
- First, self-attention is an operation on **sets**. It has no inherent notion of order.



Self-attention doesn't know the order of its inputs.

Barriers and solutions for Self-Attention as a building block

Barriers

- Doesn't have an inherent notion of order!



Solutions

Fixing the first self-attention problem: sequence order

- Since self-attention doesn't build in order information, we need to encode the order of the sentence in our keys, queries, and values.
- Consider representing each **sequence index** as a **vector**

$p_i \in \mathbb{R}^d$, for $i \in \{1, 2, \dots, T\}$ are position vectors

- Don't worry about what the p_i are made of yet!
- Easy to incorporate this info into our self-attention block: just add the p_i to our inputs!
- Let $\tilde{v}_i, \tilde{k}_i, \tilde{q}_i$ be our old values, keys, and queries.

$$\begin{aligned} v_i &= \tilde{v}_i + p_i \\ q_i &= \tilde{q}_i + p_i \\ k_i &= \tilde{k}_i + p_i \end{aligned}$$

In deep self-attention networks, we do this at the first layer! You could concatenate them as well, but people mostly just add...

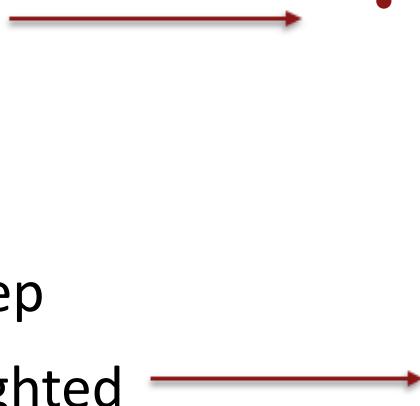
Position representation vectors

- **Sinusoidal position representations:** concatenate sinusoidal functions of varying periods.
- Learned absolute position representations: Let all p_i be learnable parameters!
Learn a matrix $p \in \mathbb{R}^{d \times T}$, and let each p_i be a column of that matrix!
- Sometimes people try more flexible representations of position:
 - Relative linear position attention
 - Dependency syntax-based position

Barriers and solutions for Self-Attention as a building block

Barriers

- Doesn't have an inherent notion of order!
- No nonlinearities for deep learning! It's all just weighted averages



Solutions

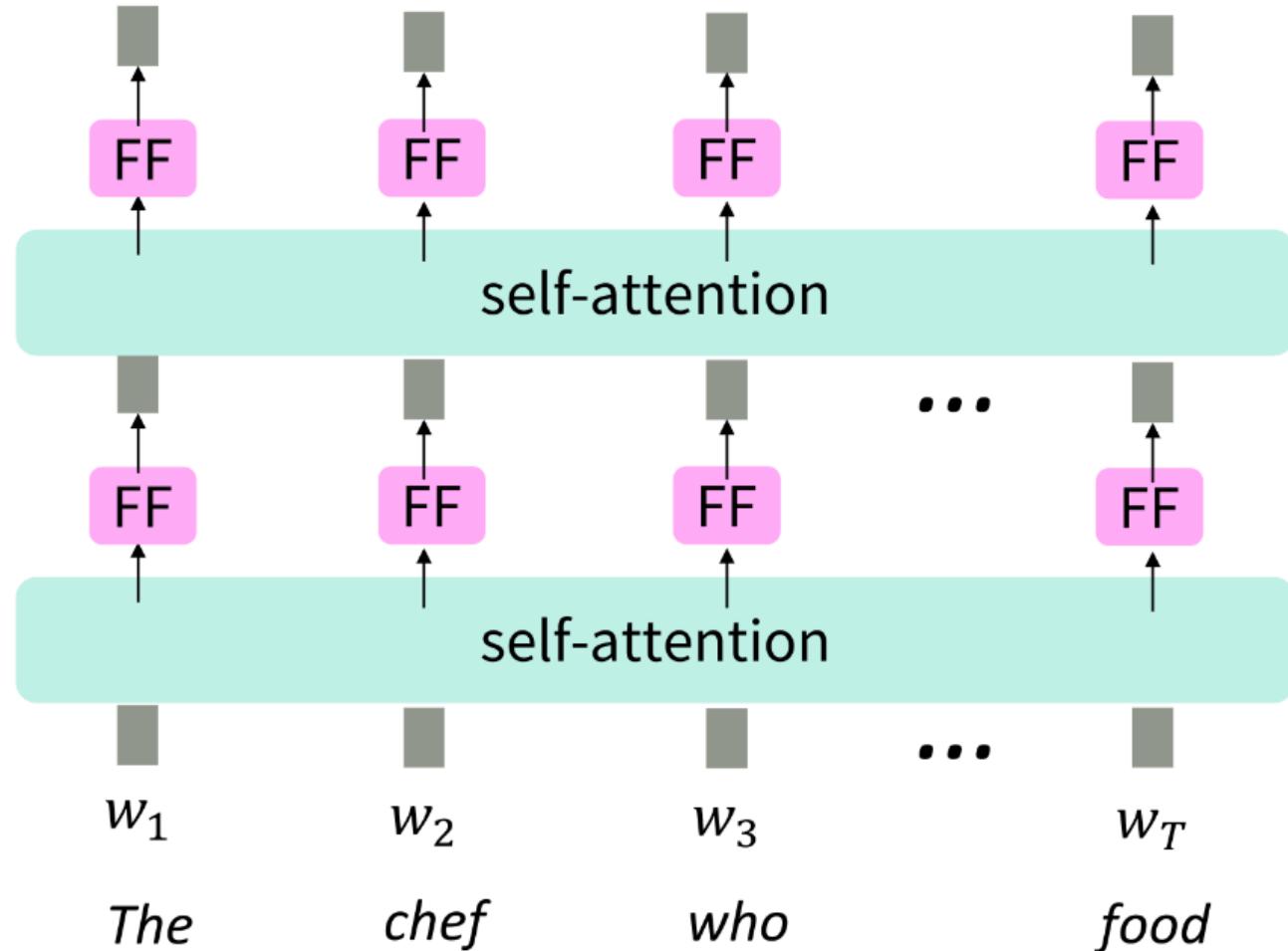
- Add position representations to the inputs

Adding nonlinearities in self-attention

- Note that there are no elementwise nonlinearities in self-attention; stacking more self-attention layers just re-averages **value** vectors
- Easy fix: add a **feed-forward network** to post-process each output vector.

$$m_i = \text{MLP}(\text{output}_i)$$

$$= W_2 * \text{ReLU}(W_1 \times \text{output}_i + b_1) + b_2$$



Barriers and solutions for Self-Attention as a building block

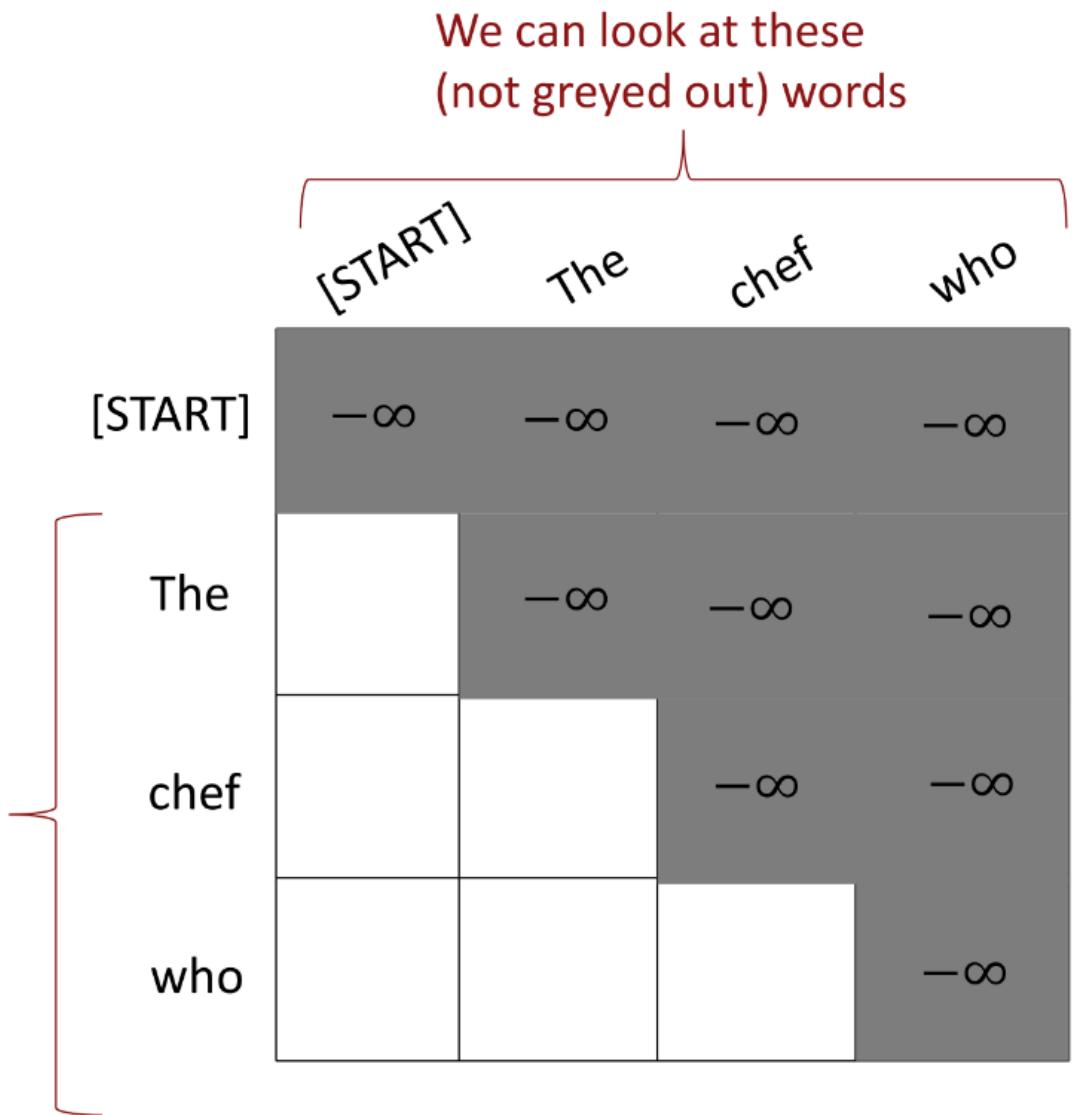
| Barriers | Solutions |
|--|---|
| • Doesn't have an inherent notion of order! | → • Add position representations to the input |
| • No nonlinearities for deep learning magic! It's all just weighted averages | → • Easy fix: apply the same feedforward network to each self-attention output. |
| • Need to ensure we don't "look at the future" when predicting a sequence | → |
| • Like in machine translation • Or language modeling | |

Masking the future in self-attention

- To use self-attention in **decoders**, we need to ensure we can't peek at the future.
- At every timestep, we could change the set of **keys and queries** to include only past words. (Inefficient!)
- To enable parallelization, we **mask out attention** to future words by setting attention scores to $-\infty$.

$$e_{ij} = \begin{cases} q_i^T k_j, & j < i \\ -\infty, & j \geq i \end{cases}$$

For encoding
these words



Barriers and solutions for Self-Attention as a building block

| Barriers | Solutions |
|--|---|
| • Doesn't have an inherent notion of order! | → • Add position representations to the inputs |
| • No nonlinearities for deep learning magic! It's all just weighted averages | → • Easy fix: apply the same feedforward network to each self-attention output. |
| • Need to ensure we don't "look at the future" when predicting a sequence <ul style="list-style-type: none">• Like in machine translation• Or language modeling | → • Mask out the future by artificially setting attention weights to 0! |

Necessities for a self-attention building block:

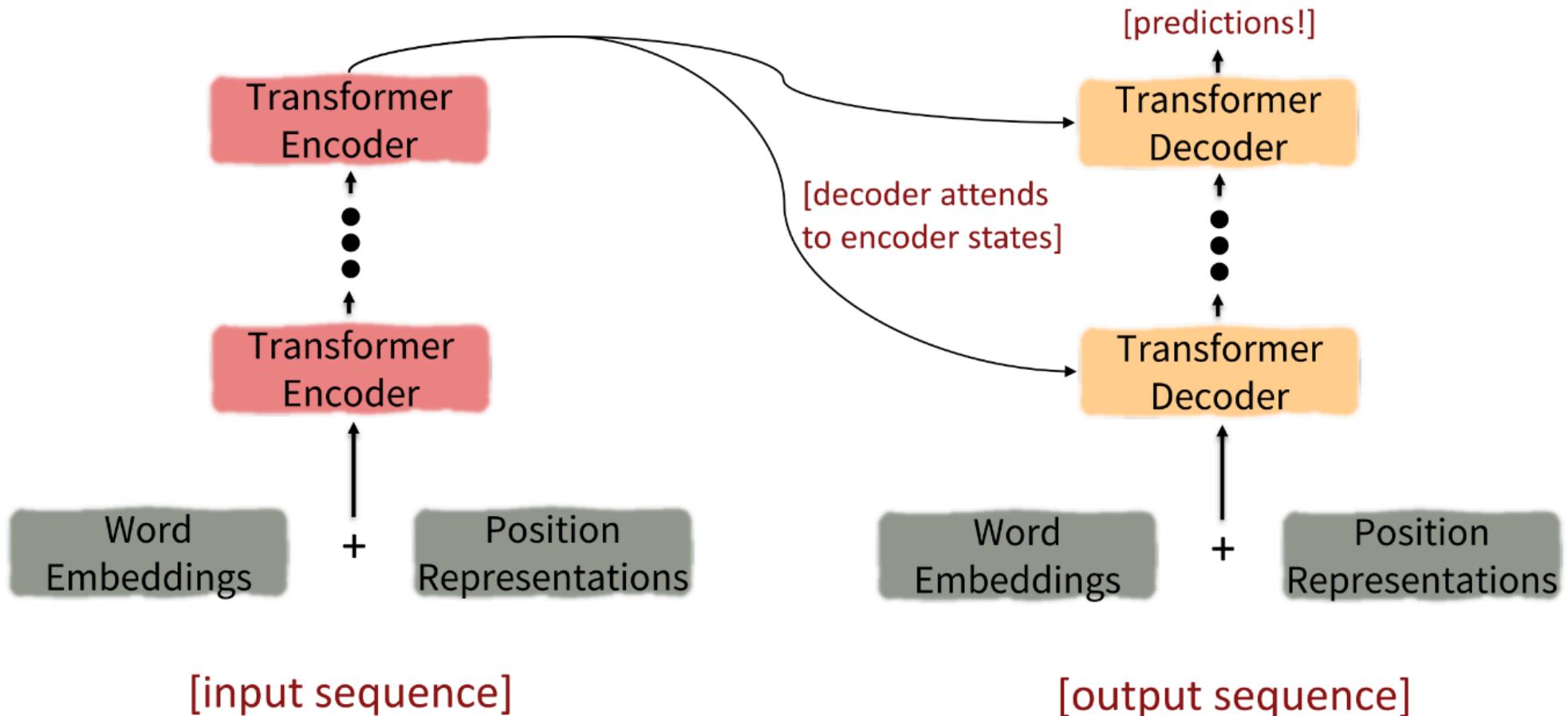
- **Self-attention:**
 - the basis of the method.
- **Position representations:**
 - Specify the sequence order, since self-attention is an unordered function of its inputs.
- **Nonlinearities:**
 - At the output of the self-attention block
 - Frequently implemented as a simple feed-forward network.
- **Masking:**
 - In order to parallelize operations while not looking at the future.
 - Keeps information about the future from “leaking” to the past.
- That’s it! But this is not the **Transformer** model we’ve been hearing about.

Outline

1. From recurrence (RNN) to attention-based NLP models
2. Introducing the Transformer model

The Transformer Encoder-Decoder [Vaswani et al., 2017]

First, let's look at the Transformer Encoder and Decoder Blocks at a high level



The Transformer Encoder-Decoder [Vaswani et al., 2017]

Next, let's look at the Transformer Encoder and Decoder Blocks

What's left in a Transformer Encoder Block that we haven't covered?

1. **Key-query-value attention:** How do we get the k, q, v vectors from a single word embedding?
2. **Multi-headed attention:** Attend to multiple places in a single layer!
3. **Tricks to help with training!**
 1. Residual connections
 2. Layer normalization
 3. Scaling the dot product
 4. These tricks **don't improve** what the model is able to do; they help improve the training process.
Both of these types of modeling improvements are very important!

The Transformer Encoder: Key-Query-Value Attention

- We saw that self-attention is when keys, queries, and values come from the same source. The Transformer does this in a particular way:
 - Let x_1, \dots, x_T be input vectors to the Transformer encoder; $x_i \in \mathbb{R}^d$
- Then keys, queries, values are:
 - $k_i = Kx_i$, where $K \in \mathbb{R}^{d \times d}$ is the key matrix.
 - $q_i = Qx_i$, where $Q \in \mathbb{R}^{d \times d}$ is the query matrix.
 - $v_i = Vx_i$, where $V \in \mathbb{R}^{d \times d}$ is the value matrix.
- These matrices allow *different aspects* of the x vectors to be used/emphasized in each of the three roles.

The Transformer Encoder: Key-Query-Value Attention

- Let's look at how key-query-value attention is computed, in matrices.
 - Let $X = [x_1; \dots; x_T] \in \mathbb{R}^{T \times d}$ be the concatenation of input vectors.
 - First, note that $XK \in \mathbb{R}^{T \times d}$, $XQ \in \mathbb{R}^{T \times d}$, $XV \in \mathbb{R}^{T \times d}$.
 - The output is defined as $\text{output} = \text{softmax}(XQ(XK)^\top) \times XV$.

First, take the query-key dot products in one matrix multiplication: $XQ(XK)^\top$

Next, softmax, and compute the weighted average with another matrix multiplication.

$$XQ \begin{pmatrix} K^\top X^\top \\ XV \end{pmatrix} = \begin{pmatrix} XQK^\top X^\top \\ XV \end{pmatrix} \in \mathbb{R}^{T \times T}$$

All pairs of attention scores!

softmax

The Transformer Encoder: Multi-headed attention

- What if we want to look in multiple places in the sentence at once?
 - For word i , self-attention “looks” where $x_i^T Q^T K x_j$ is high, but maybe we want to focus on different j for different reasons?
- We’ll define **multiple attention “heads”** through multiple Q,K,V matrices
- Let, $Q_\ell, K_\ell, V_\ell \in \mathbb{R}^{d \times \frac{d}{h}}$, where h is the number of attention heads, and ℓ ranges from 1 to h .
- Each attention head performs attention independently:
 - $\text{output}_\ell = \text{softmax}(X Q_\ell K_\ell^T X^T) * X V_\ell$, where $\text{output}_\ell \in \mathbb{R}^{d/h}$
- Then the outputs of all the heads are combined!
 - $\text{output} = Y[\text{output}_1; \dots; \text{output}_h]$, where $Y \in \mathbb{R}^{d \times d}$
- Each head gets to “look” at different things, and construct value vectors differently.

The Transformer Encoder: Multi-headed attention

- What if we want to look in multiple places in the sentence at once?
 - For word i , self-attention “looks” where $x_i^T Q^T K x_j$ is high, but maybe we want to focus on different j for different reasons?
- We’ll define **multiple attention “heads”** through multiple Q,K,V matrices
- Let, $Q_\ell, K_\ell, V_\ell \in \mathbb{R}^{d \times \frac{d}{h}}$, where h is the number of attention heads, and ℓ ranges from 1 to h .

Single-head attention

(just the query matrix)

$$X \quad Q = XQ$$

Multi-head attention

(just two heads here)

$$X \quad Q_1 \quad Q_2 = XQ_1 \quad XQ_2$$

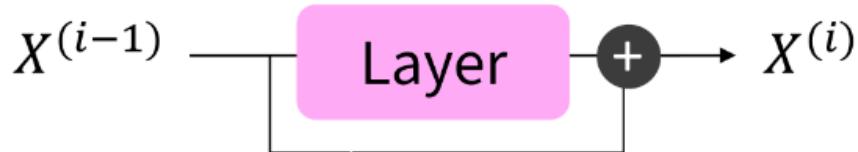
Same amount of computation as single-head self-attention!

The Transformer Encoder: Residual connections [He et al., 2016]

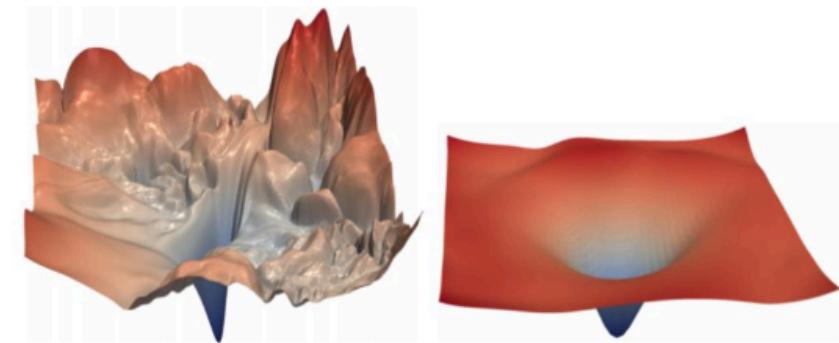
- **Residual connections** are a trick to help models train better.
 - Instead of $X^{(i)} = \text{Layer}(X^{(i-1)})$ (where i represents the layer)



- We let $X^{(i)} = X^{(i-1)} + \text{Layer}(X^{(i-1)})$ (so we only have to learn “the residual” from the previous layer)



- Residual connections are thought to make the loss landscape considerably smoother (thus easier training!)



[no residuals] [residuals]

[Loss landscape visualization,
[Li et al., 2018](#), on a ResNet]

The Transformer Encoder: Layer normalization [Ba et al., 2016]

- Layer normalization is a trick to help models train faster.
- Idea: cut down on uninformative variation in hidden vector values by normalizing to unit mean and standard deviation **within each layer**.
 - LayerNorm's success may be due to its normalizing gradients [[Xu et al., 2019](#)]
- Let $x \in \mathbb{R}^d$ be an individual (word) vector in the model.
- Let $\mu = \sum_{j=1}^d x_j$; this is the mean; $\mu \in \mathbb{R}$.
- Let $\sigma = \sqrt{\frac{1}{d} \sum_{j=1}^d (x_j - \mu)^2}$; this is the standard deviation; $\sigma \in \mathbb{R}$.
- Let $\gamma \in \mathbb{R}^d$ and $\beta \in \mathbb{R}^d$ be learned “gain” and “bias” parameters. (Can omit!)
- Then layer normalization computes:

$$\text{output} = \frac{x - \mu}{\sqrt{\sigma} + \epsilon}$$

Normalize by scalar
mean and variance

The Transformer Encoder: Scaled Dot Product [Vaswani et al., 2017]

- “**Scaled Dot Product**” attention is a final variation to aid in Transformer training.
- When dimensionality d becomes large, dot products between vectors tend to become large.
 - Because of this, inputs to the softmax function can be large, making the gradients small.
- Instead of the self-attention function we’ve seen:

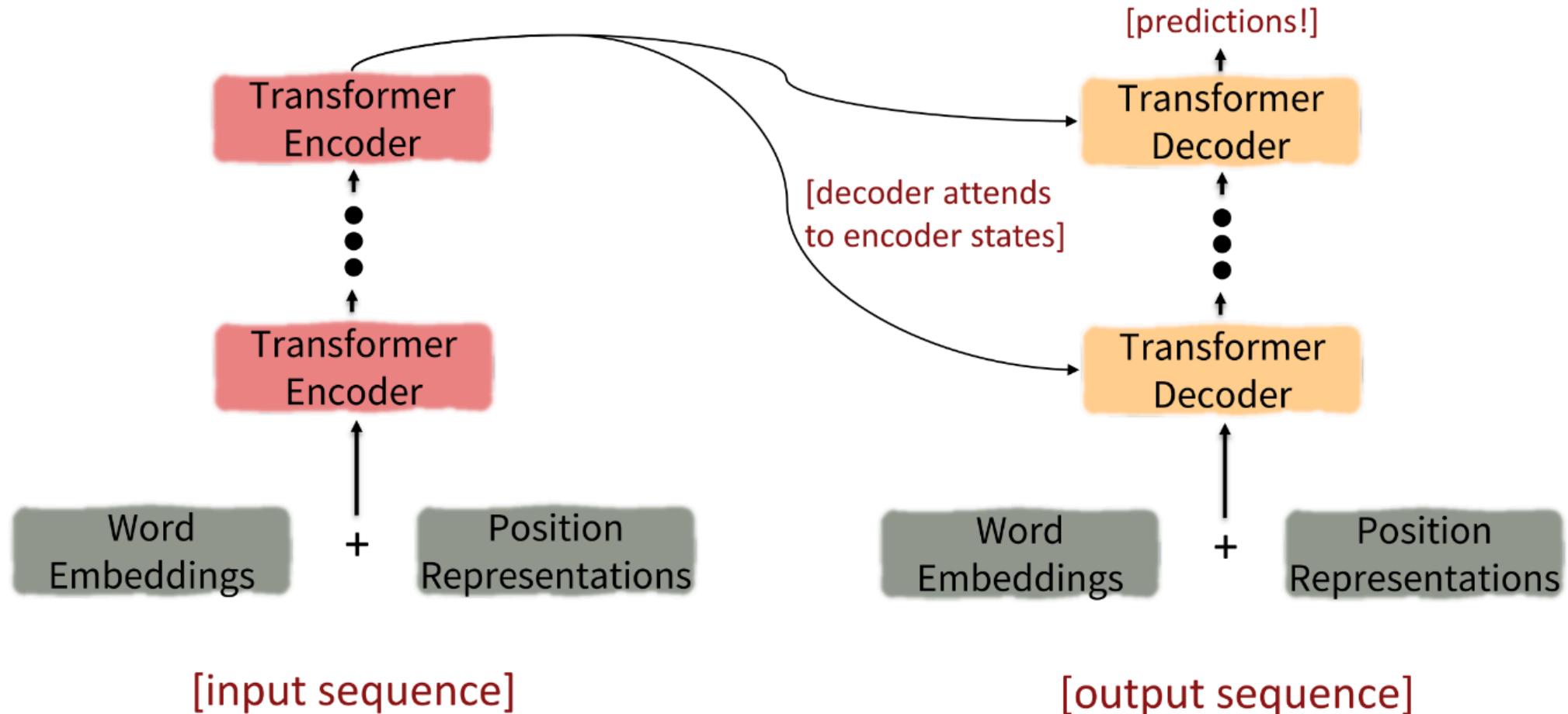
$$\text{output}_\ell = \text{softmax}(XQ_\ell K_\ell^\top X^\top) * XV_\ell$$

- We divide the attention scores by $\sqrt{d/h}$, to stop the scores from becoming large just as a function of d/h (The dimensionality divided by the number of heads.)

$$\text{output}_\ell = \text{softmax}\left(\frac{XQ_\ell K_\ell^\top X^\top}{\sqrt{d/h}}\right) * XV_\ell$$

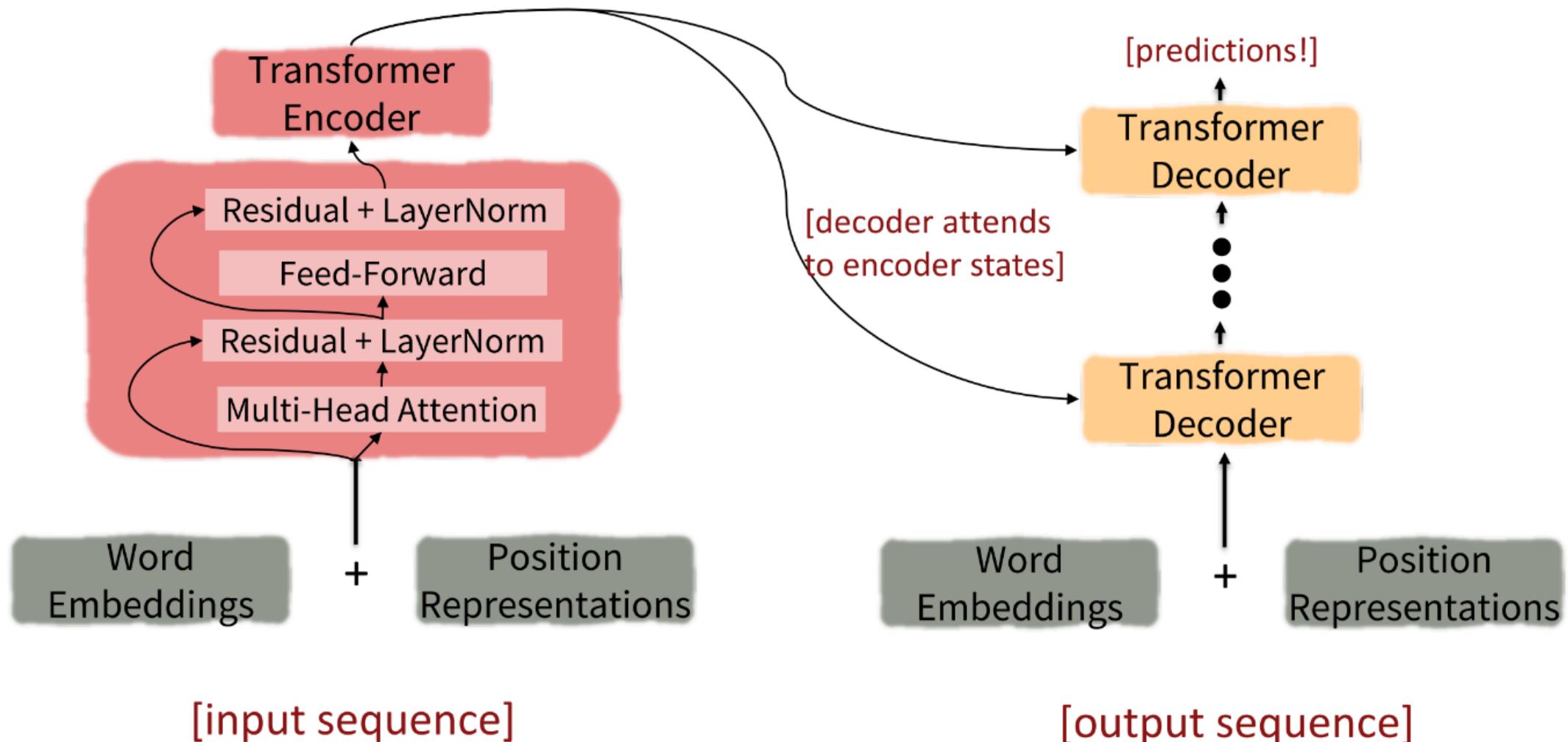
The Transformer Encoder-Decoder [Vaswani et al., 2017]

Looking back at the whole model, zooming in on an Encoder block:



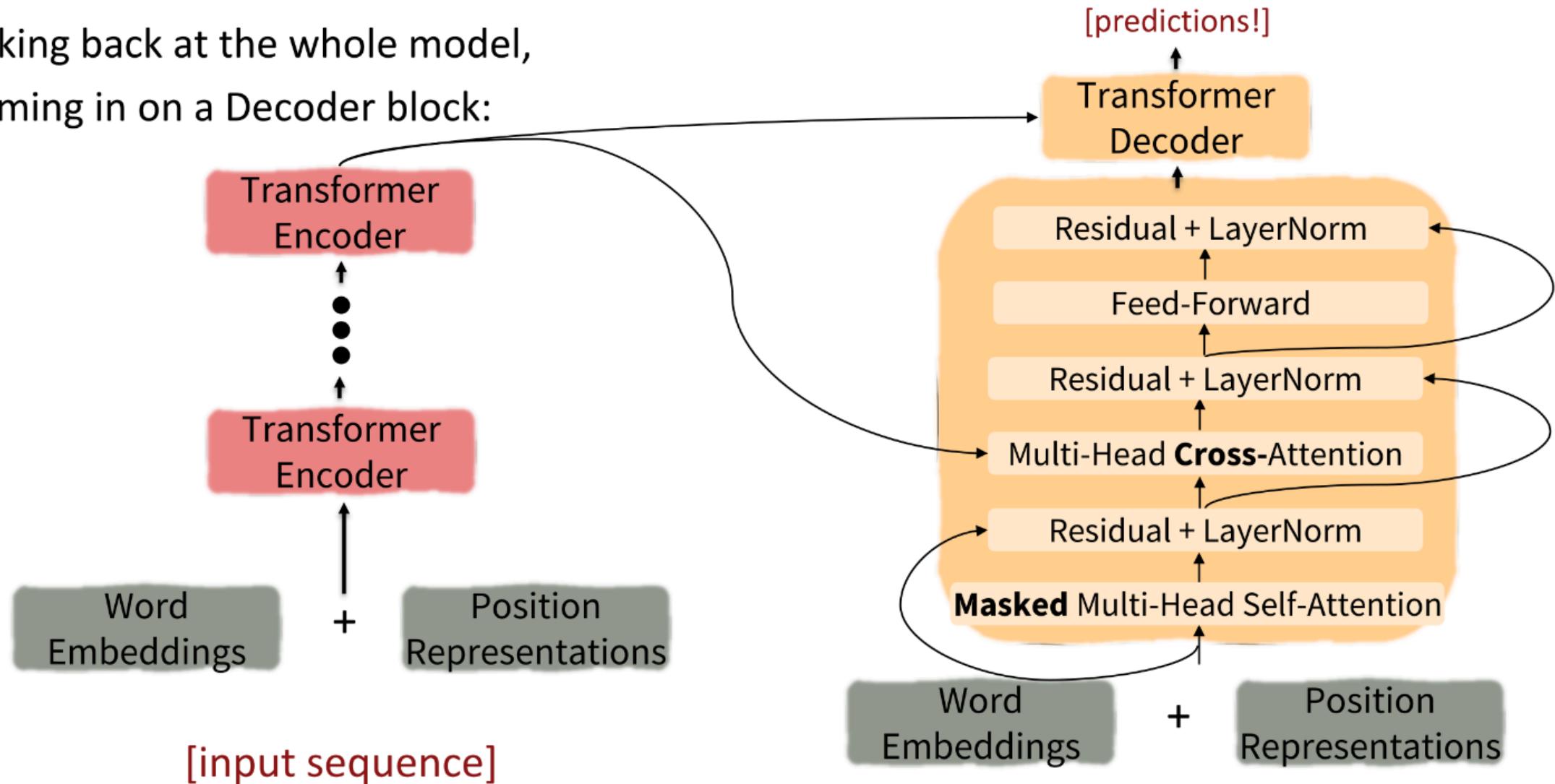
The Transformer Encoder-Decoder [Vaswani et al., 2017]

Looking back at the whole model, zooming in on an Encoder block:



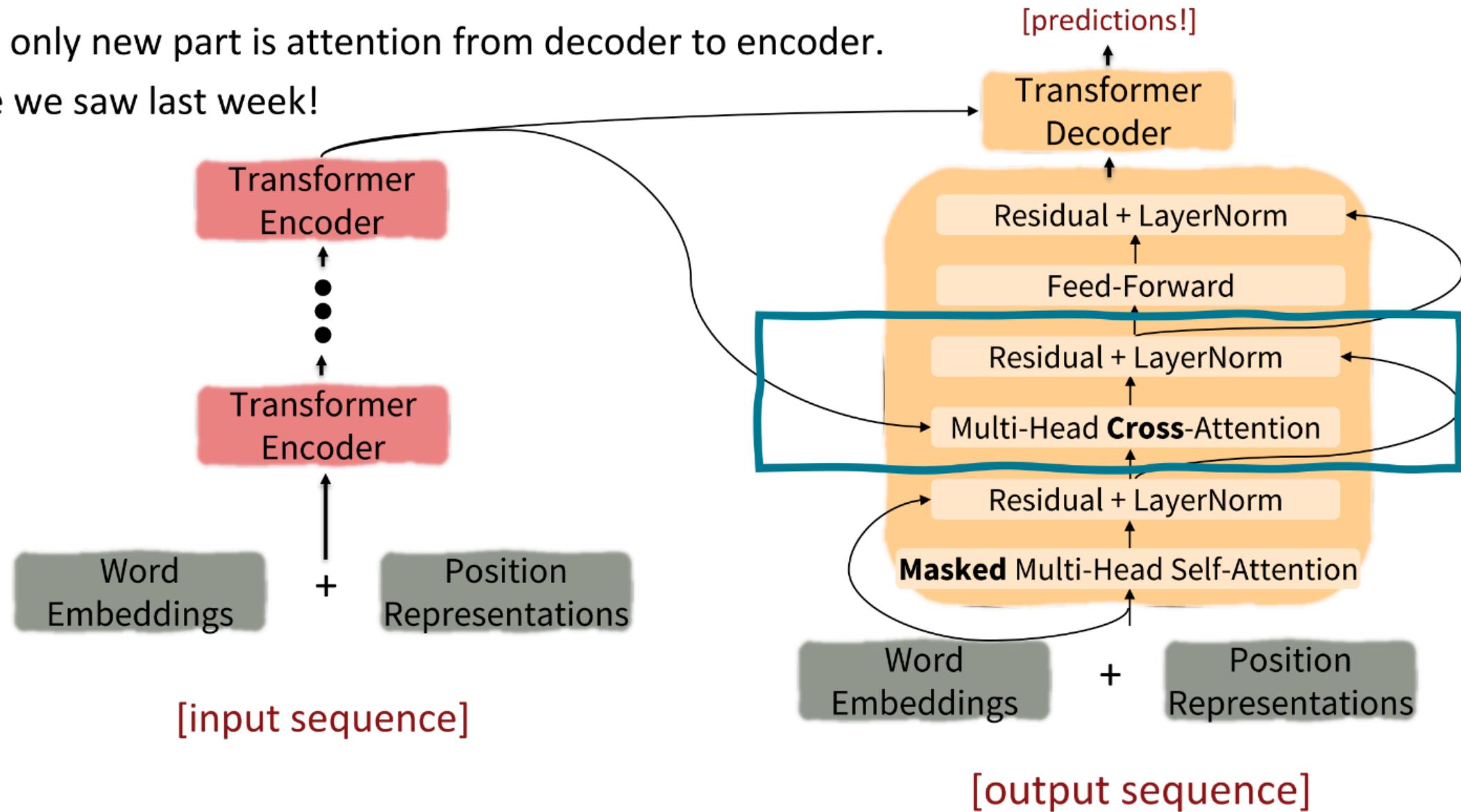
The Transformer Encoder-Decoder [Vaswani et al., 2017]

Looking back at the whole model,
zooming in on a Decoder block:



The Transformer Encoder-Decoder [Vaswani et al., 2017]

The only new part is attention from decoder to encoder.
Like we saw last week!



The Transformer Decoder: Cross-attention (details)

- We saw that self-attention is when keys, queries, and values come from the same source.
- In the decoder, we have attention that looks more like what we saw last week.
- Let h_1, \dots, h_T be **output** vectors **from** the Transformer **encoder**; $x_i \in \mathbb{R}^d$
- Let z_1, \dots, z_T be input vectors from the Transformer **decoder**, $z_i \in \mathbb{R}^d$
- Then keys and values are drawn from the **encoder** (like a memory):
 - $k_i = Kh_i, v_i = Vh_i$.
- And the queries are drawn from the **decoder**, $q_i = Qz_i$.

The Transformer Encoder: Cross-attention (details)

- Let's look at how cross-attention is computed, in matrices.
 - Let $H = [h_1; \dots; h_T] \in \mathbb{R}^{T \times d}$ be the concatenation of encoder vectors.
 - Let $Z = [z_1; \dots; z_T] \in \mathbb{R}^{T \times d}$ be the concatenation of decoder vectors.
 - The output is defined as $\text{output} = \text{softmax}(ZQ(HK)^\top) \times HV$.

First, take the query-key dot products in one matrix multiplication: $ZQ(HK)^\top$

$$ZQ \quad K^\top H^\top = ZQK^\top H^\top \in \mathbb{R}^{T \times T}$$

All pairs of attention scores!

Next, softmax, and compute the weighted average with another matrix multiplication.

$$\text{softmax} \left(ZQK^\top H^\top \right) HV = \text{output} \in \mathbb{R}^{T \times d}$$

Thank you