

Self-Attention and Transformers

Self-Attention and Transformers

Outline

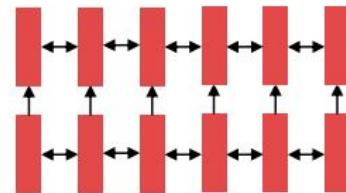
1. From recurrence (RNN) to attention-based NLP models
2. Introducing the Transformer model
3. Great results with Transformers
4. Drawbacks and variants of Transformers

Outline

1. From recurrence (RNN) to attention-based NLP models
2. Introducing the Transformer model
3. Great results with Transformers
4. Drawbacks and variants of Transformers

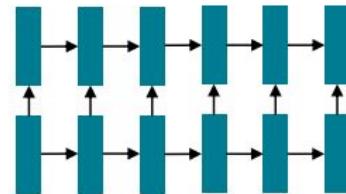
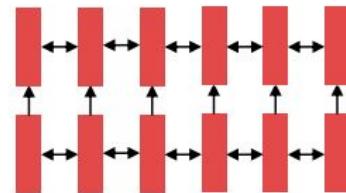
Recurrent models for (most) NLP

- Circa 2016, the de facto strategy in NLP is to **encode** sentences with a bidirectional LSTM:
(for example, the source sentence in a translation)



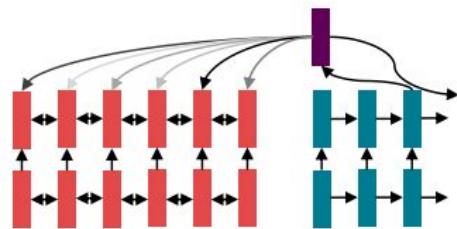
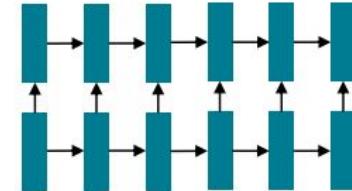
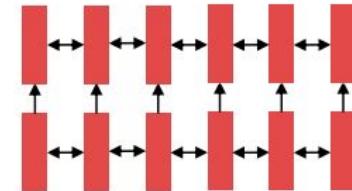
Recurrent models for (most) NLP

- 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.



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

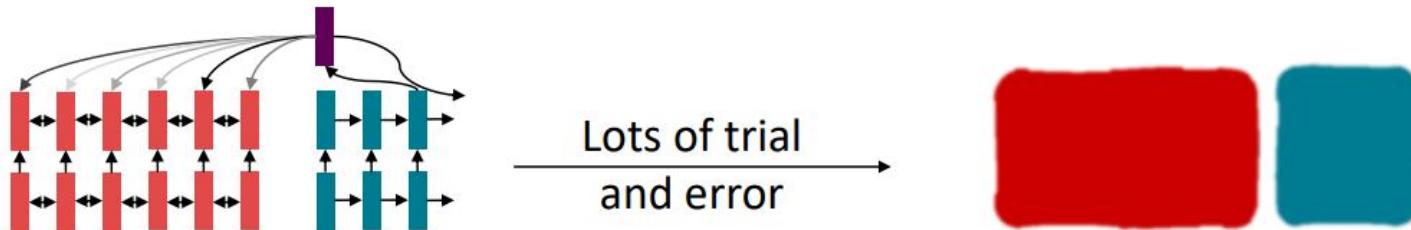


Today: Same goals, different building blocks

- Today, we're **not** trying to motivate entirely new ways of looking at problems (like Machine Translation)

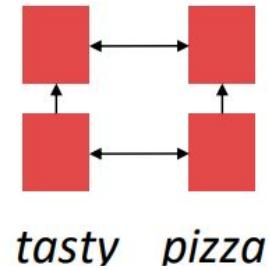
Today: Same goals, different building blocks

- Today, 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.



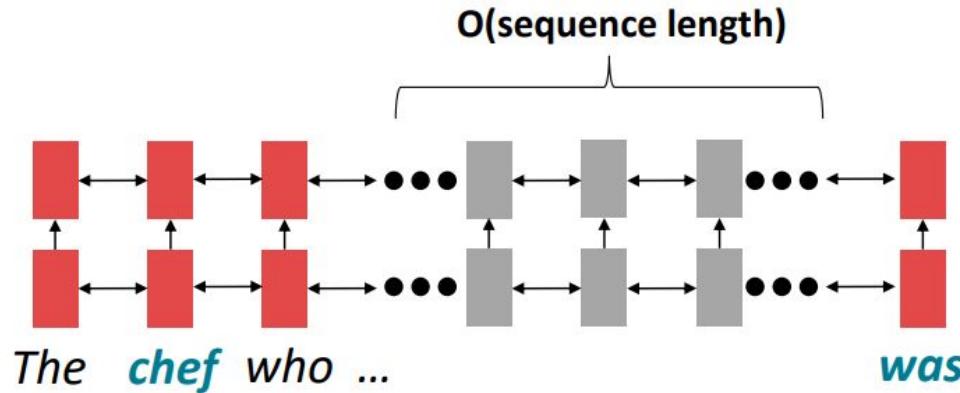
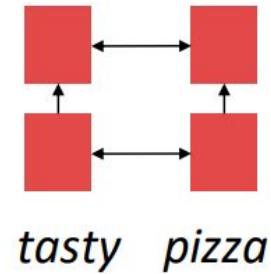
Issues with recurrent models: Linear interaction distance

- RNNs are unrolled “left-to-right”.
- This encodes linear locality: a useful heuristic!
 - Nearby words often affect each other’s meanings



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.

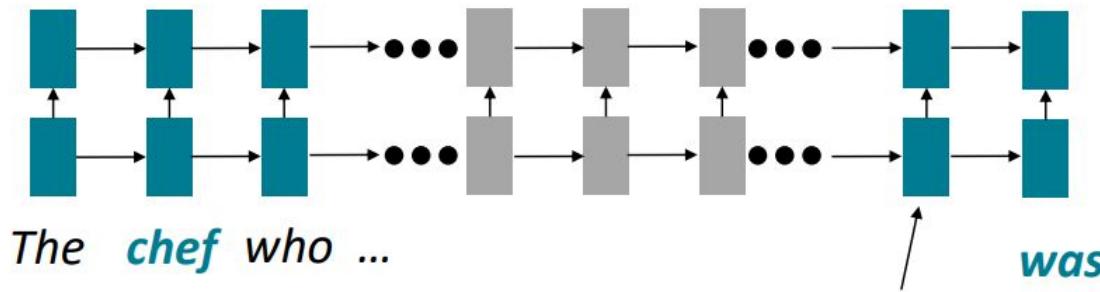


Issues with recurrent models: **Linear interaction distance**

- **O(sequence length)** steps for distant word pairs to interact means:
 - Hard to learn long-distance dependencies (because gradient problems!)

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(sequence length) many layers!

Issues with recurrent models: **Lack of parallelizability**

- Forward and backward passes have **O(sequence length)** unparallelizable operations

Issues with recurrent models: **Lack of parallelizability**

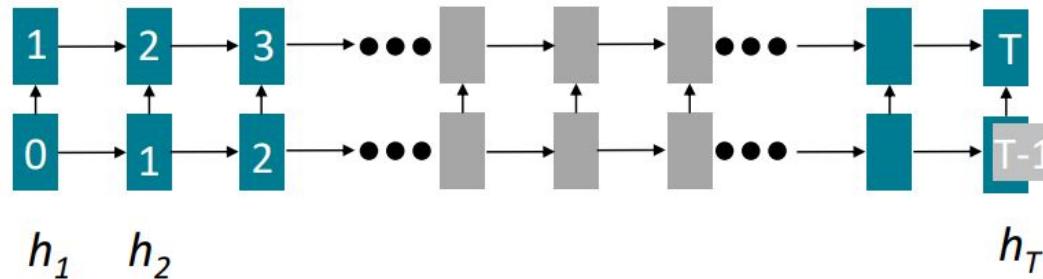
- Forward and backward passes have **O(sequence length)** unparallelizable operations
 - GPUs can perform a bunch of independent computations at once!

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

Issues with recurrent models: Lack of parallelizability

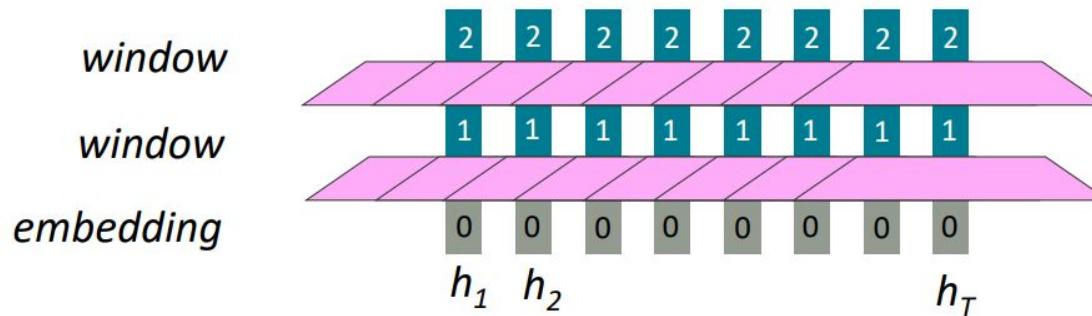
- Forward and backward passes have **O(sequence length)** unparallelizable operations
 - GPUs can perform a bunch of independent computations at once!
 - But future RNN hidden states can't be computed in full before past RNN hidden states have been computedInhibits training on very large datasets!



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

If not recurrence, then what? How about word windows?

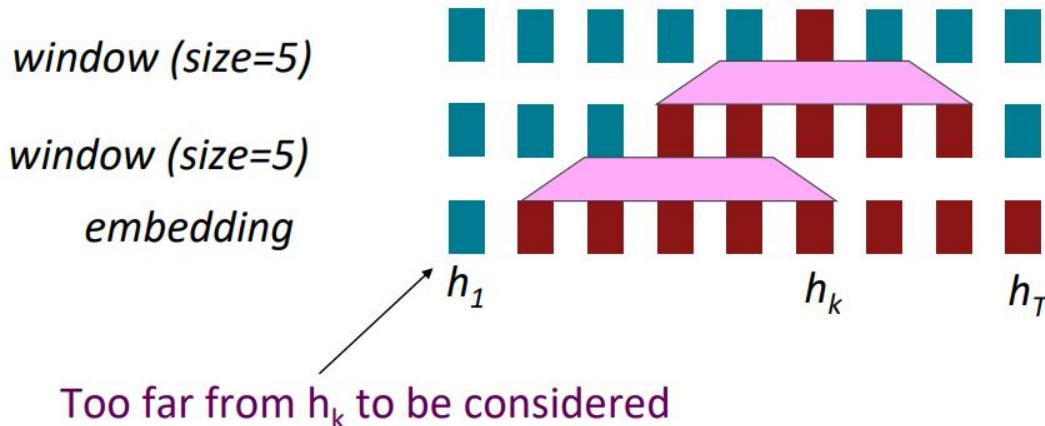
- Word window models aggregate local contexts
 - Number of unparallelizable operations does not increase sequence length!



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

If not recurrence, then what? How about word windows?

- Word window models aggregate local contexts
- What about long-distance dependencies?
 - Stacking word window layers allows interaction between farther words
- Maximum Interaction distance = sequence length / window size
 - (But if your sequences are too long, you'll just ignore long-distance context)



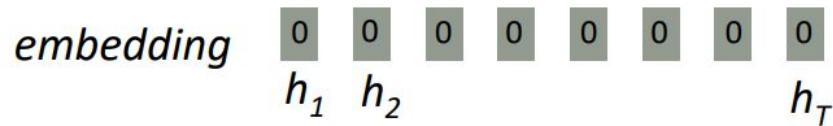
Red states indicate those “visible” to h_k

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**.

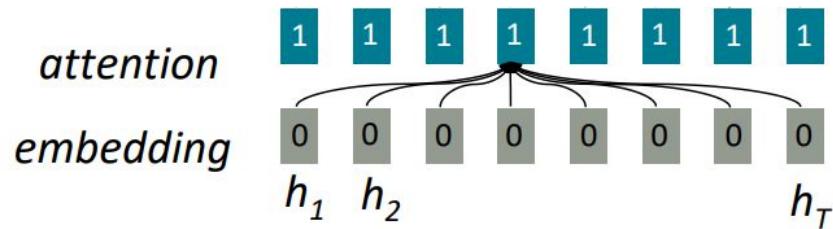
If not recurrence, then what? How about attention?

- **Attention** treats each word's representation as a **query** to access and incorporate information from a **set of values**.
 - We saw attention from the **decoder** to the **encoder**; today we'll think about attention **within a single sentence**.



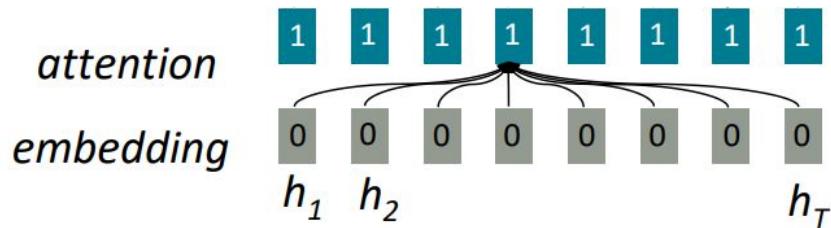
If not recurrence, then what? How about attention?

- **Attention** treats each word's representation as a **query** to access and incorporate information from a **set of values**.
 - We saw attention from the **decoder** to the **encoder**; today we'll think about attention **within a single sentence**.



If not recurrence, then what? How about attention?

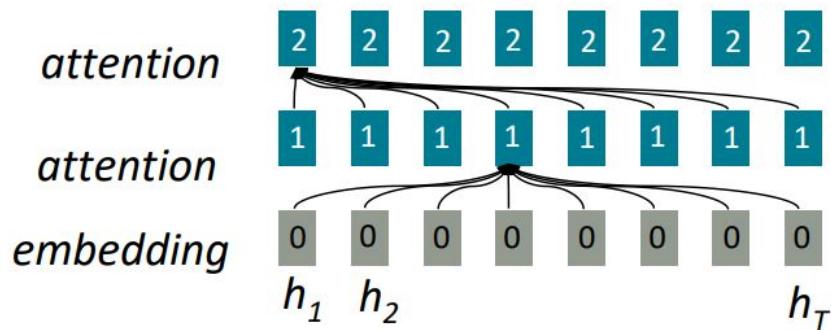
- **Attention** treats each word's representation as a **query** to access and incorporate information from a **set of values**.
 - We saw attention from the **decoder** to the **encoder**; today we'll think about attention **within a single sentence**.



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

If not recurrence, then what? How about attention?

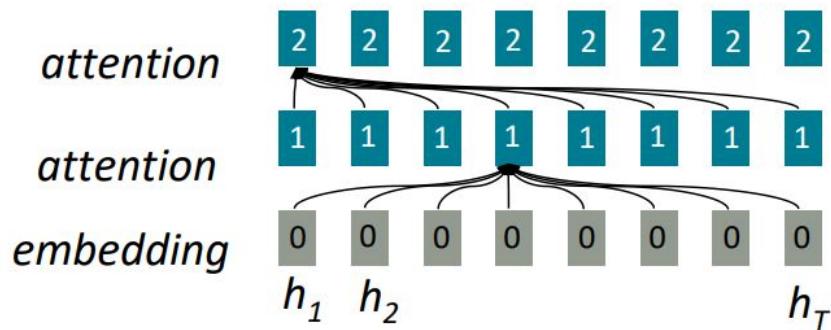
- **Attention** treats each word's representation as a **query** to access and incorporate information from a **set of values**.
 - We saw attention from the **decoder** to the **encoder**; today we'll think about attention **within a single sentence**.



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

If not recurrence, then what? How about attention?

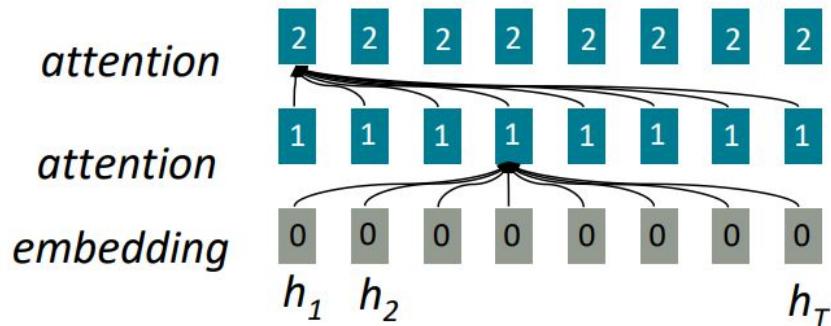
- **Attention** treats each word's representation as a **query** to access and incorporate information from a **set of values**.
 - We saw attention from the **decoder** to the **encoder**; today we'll think about attention **within a single sentence**.
- Number of unparallelizable operations does not increase sequence length.



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

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**.
 - We saw attention from the **decoder** to the **encoder**; today we'll think about attention **within a single sentence**.
- 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**.

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 $v_i \in \mathbb{R}^d$

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 $v_i \in \mathbb{R}^d$

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.

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

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
(softmax)

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

Compute outputs as weighted sum of **values**

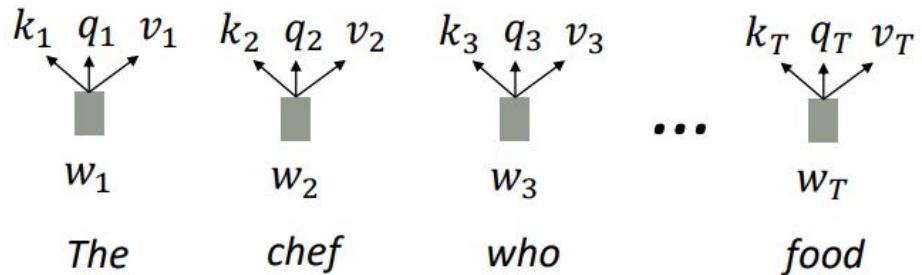
The number of queries can differ from the number of keys and values in practice.

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.

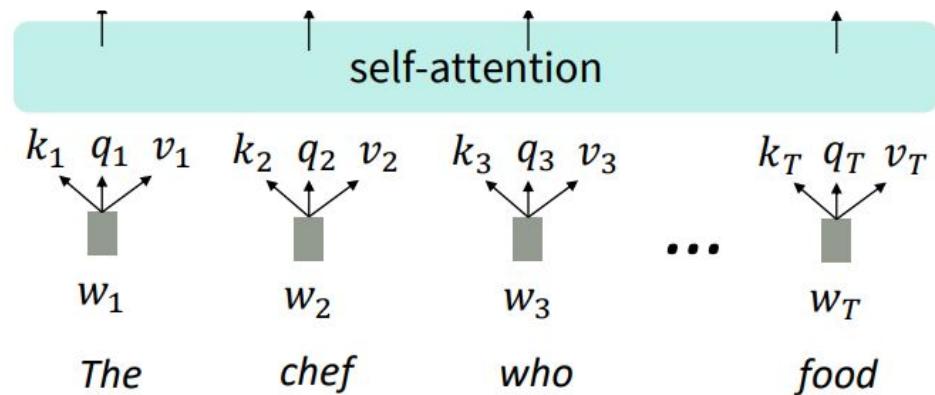
Self-attention as an NLP building block

- In the diagram at the right, we have stacked self-attention blocks, like we might stack LSTM layers.



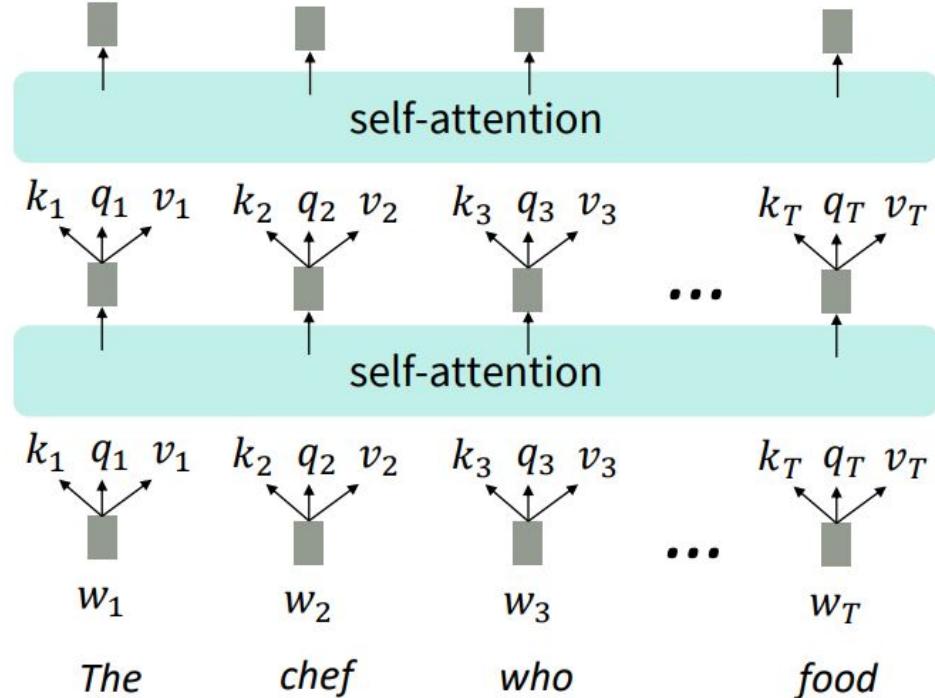
Self-attention as an NLP building block

- In the diagram at the right, we have stacked self-attention blocks, like we might stack LSTM layers.



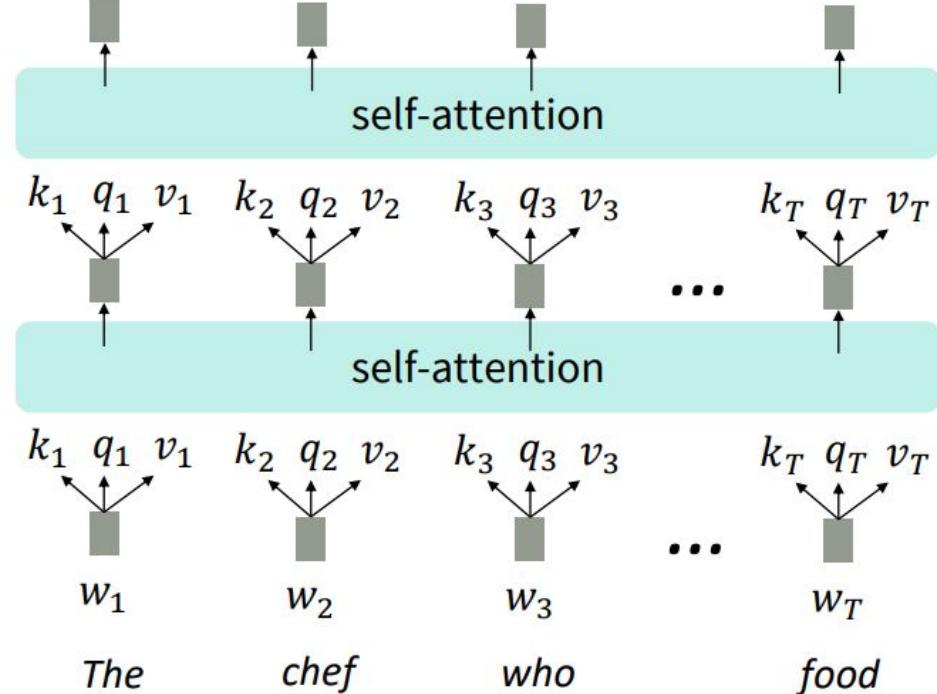
Self-attention as an NLP building block

- In the diagram at the right, we have stacked self-attention blocks, like we might stack LSTM layers.



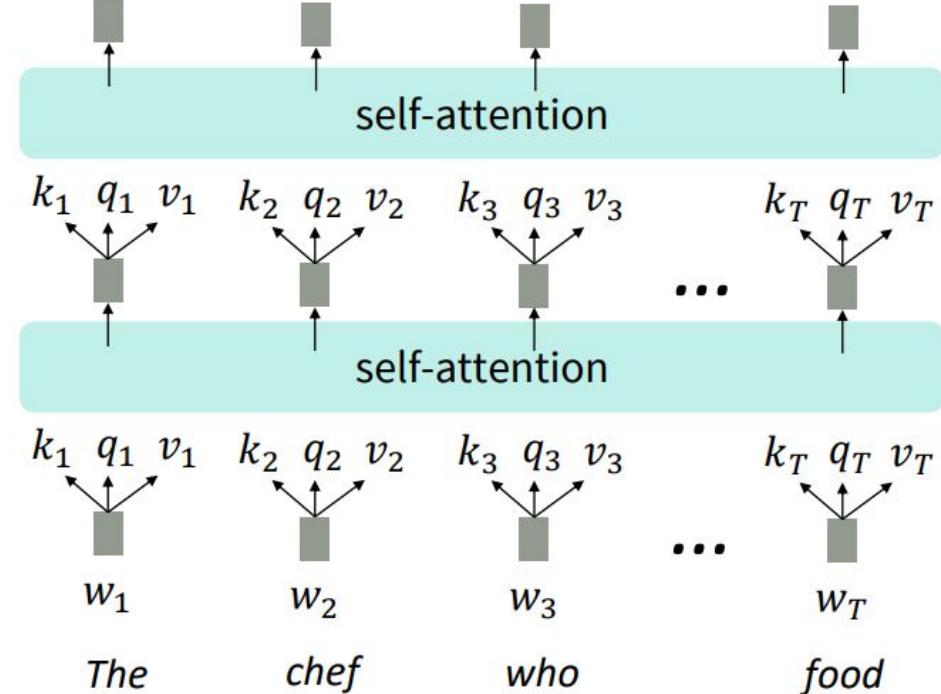
Self-attention as an NLP building block

- 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?



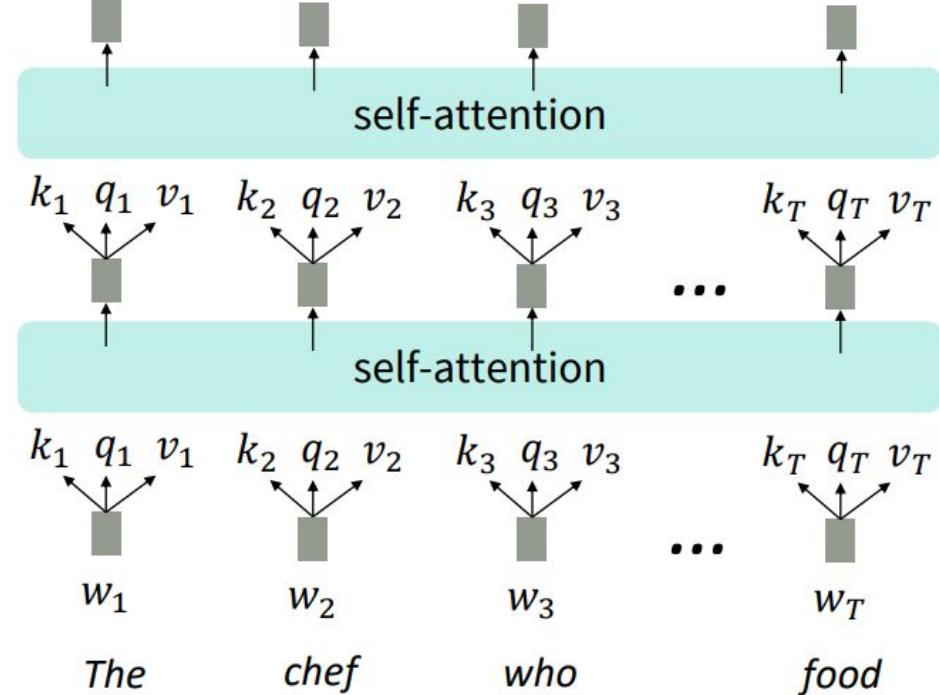
Self-attention as an NLP building block

- 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.



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.

Issues with recurrent models: Linear interaction distance

Barriers

- Doesn't have an inherent notion of order!



Solutions

Fixing the first self-attention problem: **sequence order**

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.

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

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!

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**

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

- 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!

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.

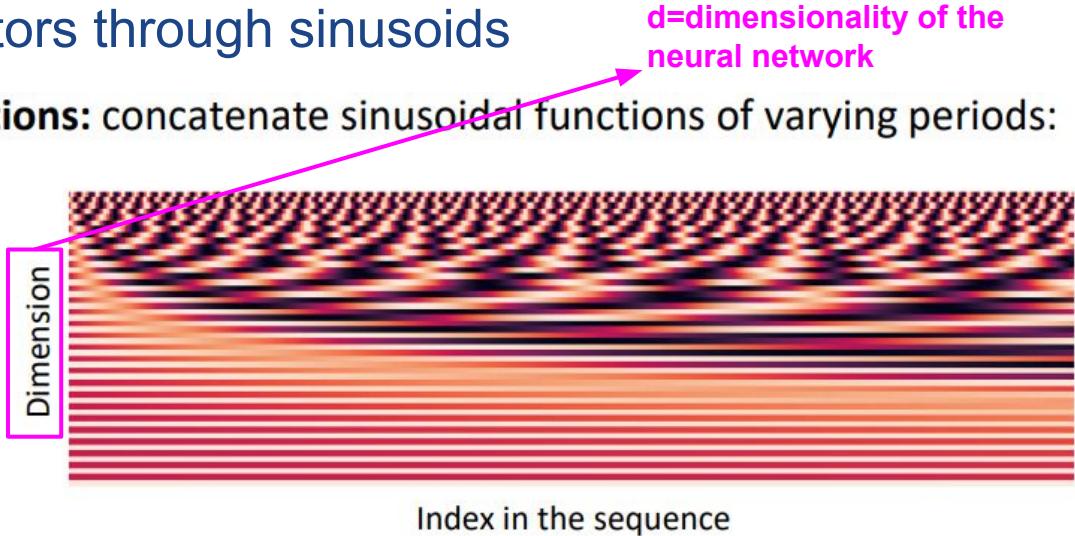
$$\begin{aligned} v_i &= \tilde{v}_i + p_i \\ q_i &= \tilde{q}_i + p_i \\ k_i &= \tilde{k}_i + p_i \end{aligned}$$

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 through sinusoids

- **Sinusoidal position representations:** concatenate sinusoidal functions of varying periods:

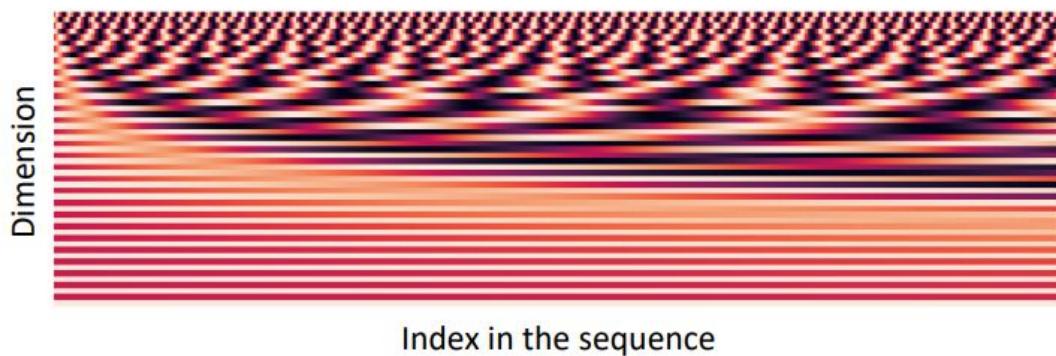
$$p_i = \begin{pmatrix} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \vdots \\ \vdots \\ \sin(i/10000^{2*\frac{d}{2}/d}) \\ \cos(i/10000^{2*\frac{d}{2}/d}) \end{pmatrix}$$



Position representation vectors through sinusoids

- **Sinusoidal position representations:** concatenate sinusoidal functions of varying periods:

$$p_i = \begin{pmatrix} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \vdots \\ \sin(i/10000^{2*\frac{d}{2}/d}) \\ \cos(i/10000^{2*\frac{d}{2}/d}) \end{pmatrix}$$

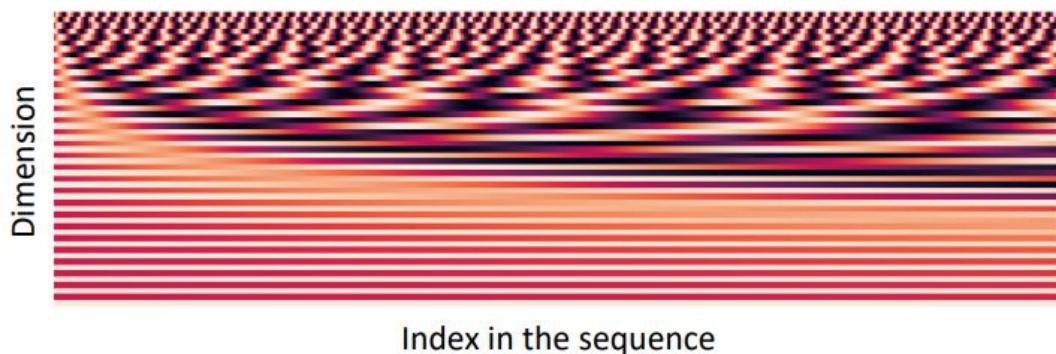


- Pros:
 - Periodicity indicates that maybe “absolute position” isn’t as important
 - Maybe can extrapolate to longer sequences as periods restart!

Position representation vectors through sinusoids

- **Sinusoidal position representations:** concatenate sinusoidal functions of varying periods:

$$p_i = \begin{pmatrix} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \vdots \\ \sin(i/10000^{2*\frac{d}{2}/d}) \\ \cos(i/10000^{2*\frac{d}{2}/d}) \end{pmatrix}$$



- Pros:
 - Periodicity indicates that maybe “absolute position” isn’t as important
 - Maybe can extrapolate to longer sequences as periods restart!
- Cons:
 - Not learnable; also the extrapolation doesn’t really work!

Position representation vectors learned from scratch

Navigation icons: back, forward, search, etc.

Position representation vectors learned from scratch

- **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!

Position representation vectors learned from scratch

- **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!
- Pros:
 - Flexibility: each position gets to be learned to fit the data

Position representation vectors learned from scratch

- **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!
- Pros:
 - Flexibility: each position gets to be learned to fit the data
- Cons:
 - Definitely can't extrapolate to indices outside $1, \dots, T$.
- Most systems use this!

Position representation vectors learned from scratch

- **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!
- Pros:
 - Flexibility: each position gets to be learned to fit the data
- Cons:
 - Definitely can't extrapolate to indices outside $1, \dots, T$.
- Most systems use this!
- Sometimes people try more flexible representations of position:
 - Relative linear position attention [\[Shaw et al., 2018\]](#)
 - Dependency syntax-based position [\[Wang et al., 2019\]](#)

Issues with recurrent models: Linear interaction distance

Barriers

- Doesn't have an inherent notion of order!



Solutions

- Add position representations to the inputs

Issues with recurrent models: Linear interaction distance

Barriers

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



Solutions

- Add position representations to the inputs



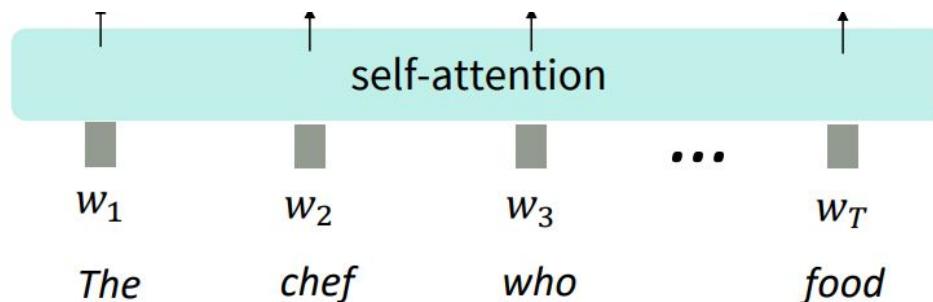
Adding nonlinearities in self-attention

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

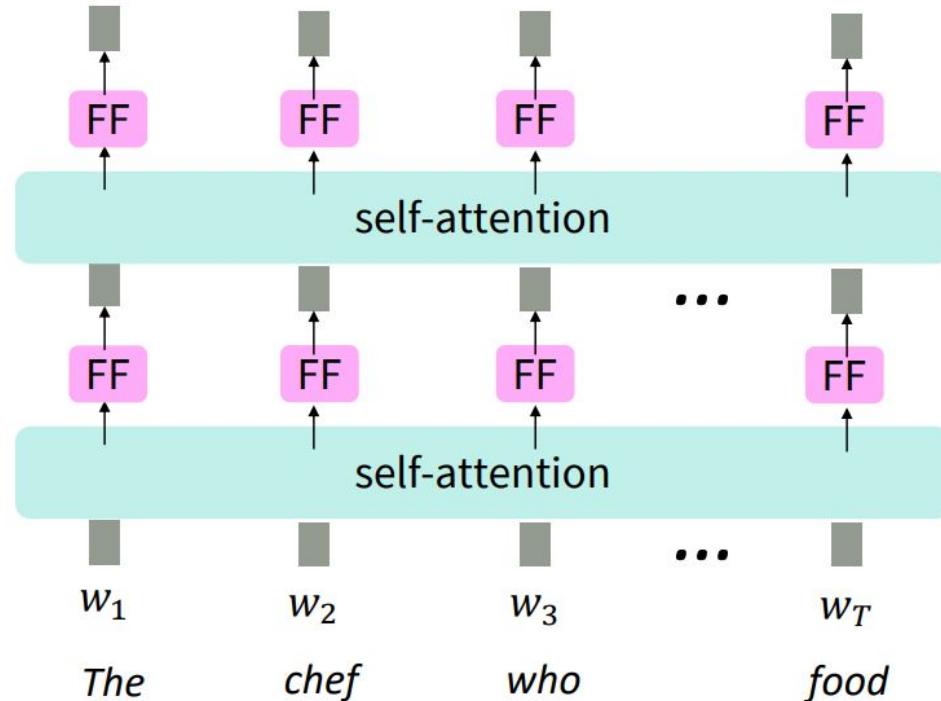
Adding nonlinearities in self-attention

- 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.



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.

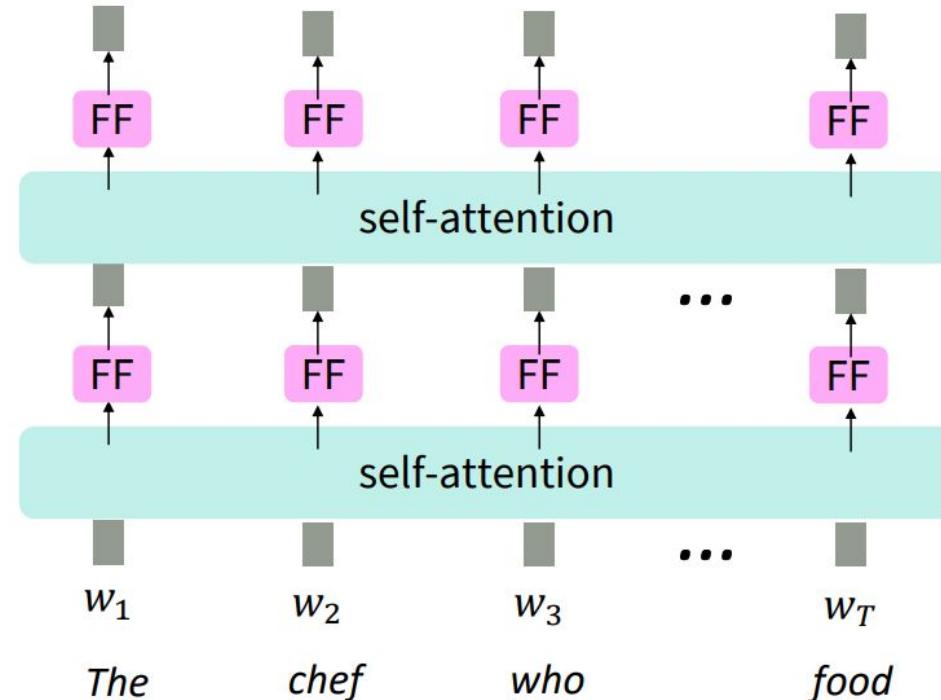


Adding nonlinearities in self-attention

- 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.

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

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

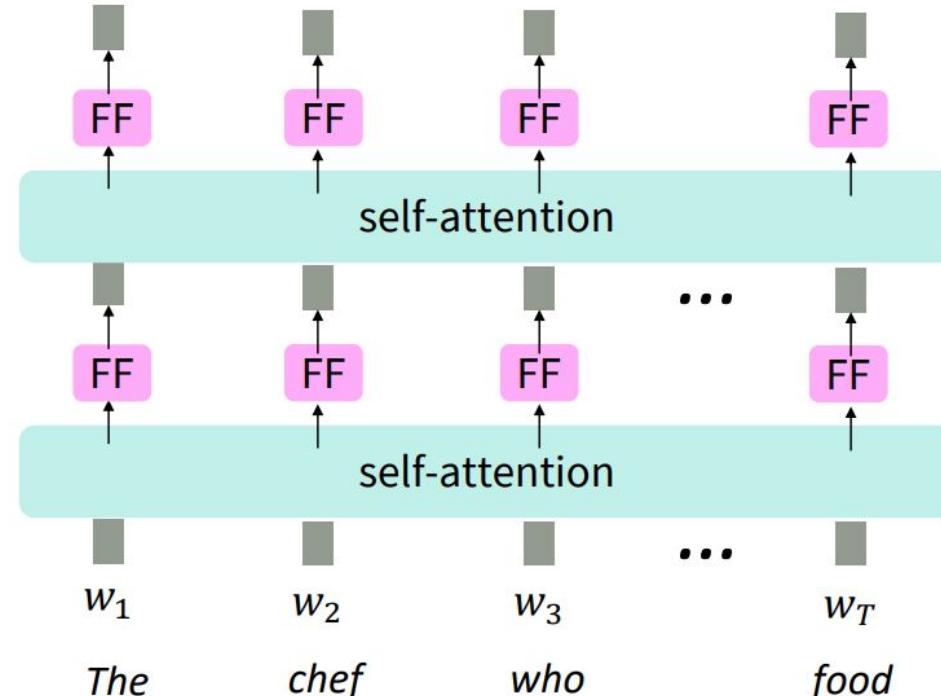


Adding nonlinearities in self-attention

- 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.

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

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



Intuition: the FF network processes the result of attention

Barriers and solutions for Self-Attention as a building block

Barriers

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



Solutions

- Add position representations to the inputs
- Easy fix: apply the same feedforward network to each self-attention output.

Barriers and solutions for Self-Attention as a building block

Barriers

- Doesn't have an inherent notion of order!
- No nonlinearities for deep learning magic! It's all just weighted averages
- Need to ensure we don't "look at the future" when predicting a sequence
 - Like in machine translation
 - Or language modeling



Solutions

- Add position representations to the inputs
- Easy fix: apply the same feedforward network to each self-attention output.



Masking the future in self-attention

- To use self-attention in **decoders**, we need to ensure we can't peek at the future.

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 $-\infty$.

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

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

[START]

We can look at these
(not greyed out) words

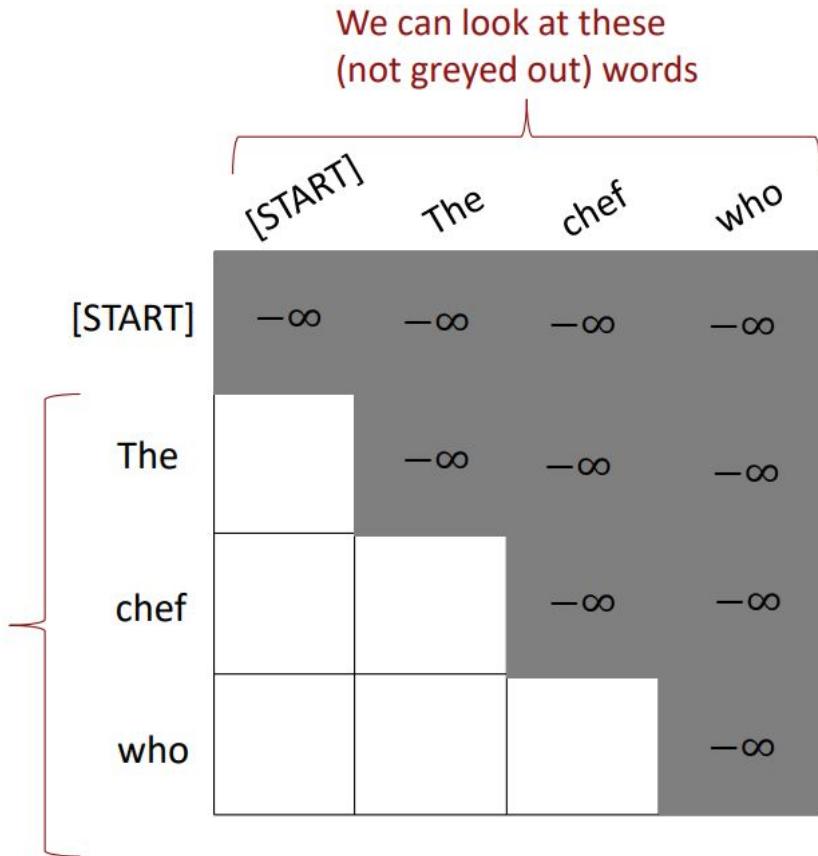
[START] The chef who

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

- Doesn't have an inherent notion of order!
- No nonlinearities for deep learning magic! It's all just weighted averages
- Need to ensure we don't "look at the future" when predicting a sequence
 - Like in machine translation
 - Or language modeling



Solutions

- Add position representations to the inputs
- Easy fix: apply the same feedforward network to each self-attention output.
- Mask out the future by artificially setting attention weights to 0!

Necessities for a self-attention building block:

:S

Necessities for a self-attention building block:

- **Self-attention:**
 - the basis of the method.

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.

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.

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.

Necessities for a self-attention building block:

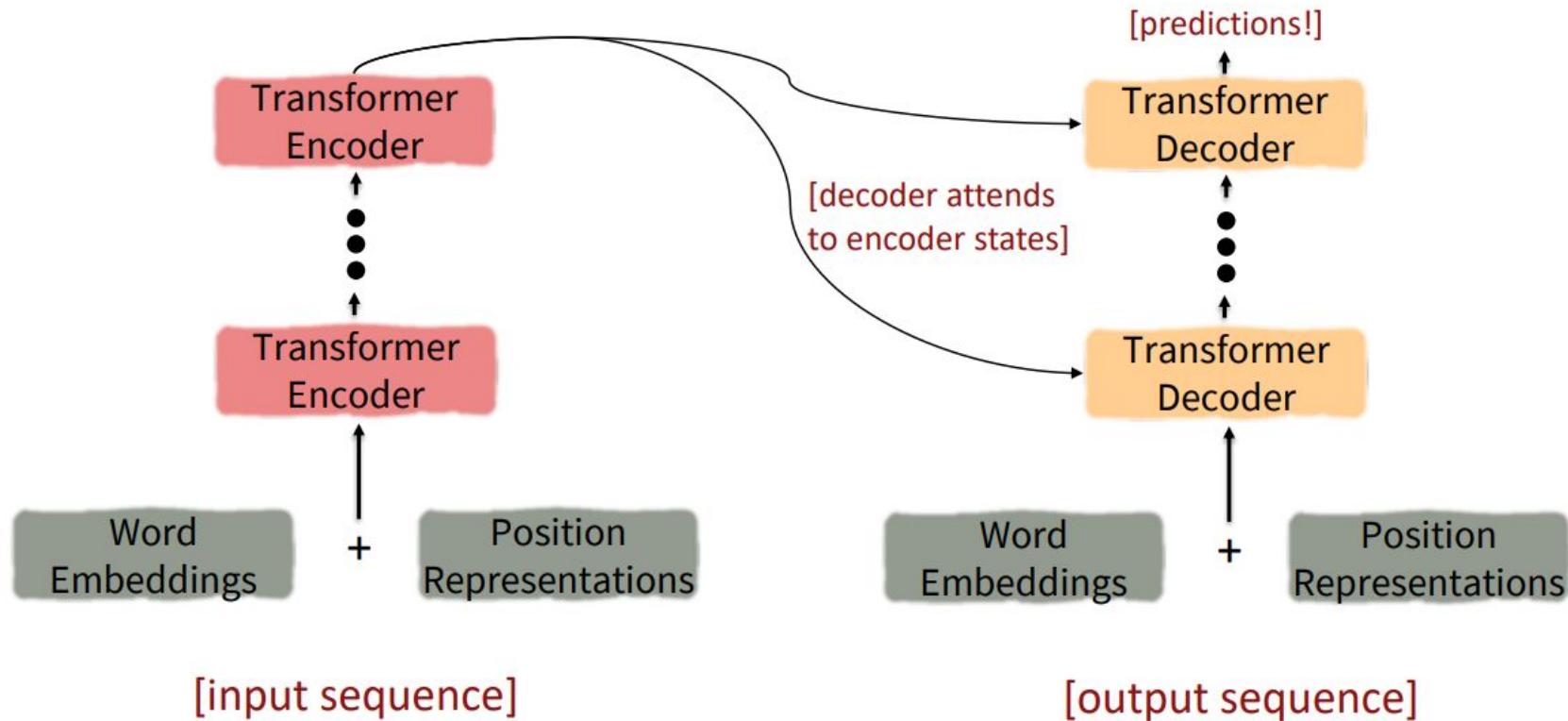
- **Self-attention:**
 - the basis of the method.
- **Position representations:**
 - Specify the sequence order, since self-attention is an unordered function of its inputs.
- **Nonlinearities:**
 - At the output of the self-attention block
 - Frequently implemented as a simple feed-forward network.
- **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
3. Great results with Transformers
4. Drawbacks and variants of Transformers

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?

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?

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!

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!**

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

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:

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; $x_i \in \mathbb{R}^d$

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.

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.

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.

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 $XK \in \mathbb{R}^{T \times d}$, $XQ \in \mathbb{R}^{T \times d}$, $XV \in \mathbb{R}^{T \times d}$.

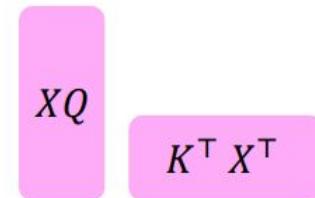
The Transformer Encoder: Key-Query-Value Attention

- 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 $\text{output} = \text{softmax}(XQ(XK)^\top) \times XV$.

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 output = $\text{softmax}(XQ(XK)^\top) \times XV$.

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



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 output = $\text{softmax}(XQ(XK)^\top) \times XV$.

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

$$XQ \quad K^\top X^\top = XQK^\top X^\top \in \mathbb{R}^{T \times T}$$

All pairs of attention scores!

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 output = $\text{softmax}(XQ(XK)^\top) \times XV$.

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

$$\begin{array}{ccc} XQ & K^\top X^\top & = XQK^\top X^\top \\ & & \text{All pairs of attention scores!} \end{array}$$

$\in \mathbb{R}^{T \times T}$

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

$$\text{softmax} \left(\begin{matrix} XQK^\top X^\top \\ \vdots \end{matrix} \right) \times XV$$

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 output = $\text{softmax}(XQ(XK)^\top) \times XV$.

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

$$\begin{array}{ccc} XQ & K^\top X^\top & = XQK^\top X^\top \\ & & \text{All pairs of attention scores!} \end{array}$$

$\in \mathbb{R}^{T \times T}$

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

$$\text{softmax} \left(\begin{array}{c} XQK^\top X^\top \end{array} \right) XV = \text{output} \in \mathbb{R}^{T \times d}$$

The Transformer Encoder: **Multi-headed attention**

The Transformer Encoder: **Multi-headed attention**

- What if we want to look in multiple places in the sentence at once?

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?

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

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 .

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 $\text{output}_\ell \in \mathbb{R}^{d/h}$

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 $Y \in \mathbb{R}^{d \times d}$

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)

$$X \quad Q = XQ$$

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)

$$X \begin{matrix} \\ Q \end{matrix} = XQ$$

Multi-head attention

(just two heads here)

$$X \begin{matrix} \\ Q_1 \quad Q_2 \end{matrix} = XQ_1 \quad XQ_2$$

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

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

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

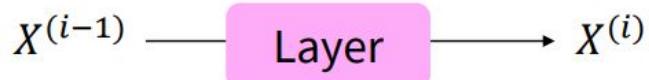
- **Residual connections** are a trick to help models train better.

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)

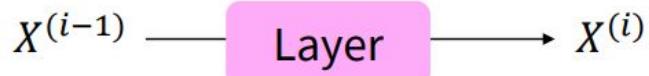
The Transformer Encoder: **Residual connections** [He et al., 2016]

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



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

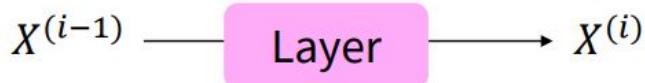
- **Residual connections** are a trick to help models train better.
 - Instead of $X^{(i)} = \text{Layer}(X^{(i-1)})$ (where i represents the layer)



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

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)



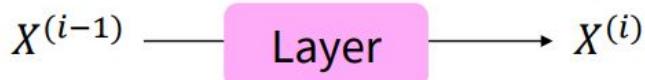
- We let $X^{(i)} = X^{(i-1)} + \text{Layer}(X^{(i-1)})$ (so we only have to learn “the residual” from the previous layer)



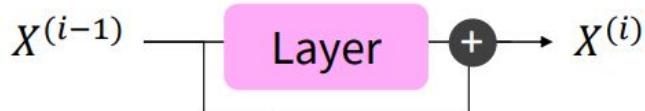
The Transformer Encoder: Residual connections [He et al., 2016]

- **Residual connections** are a trick to help models train better.

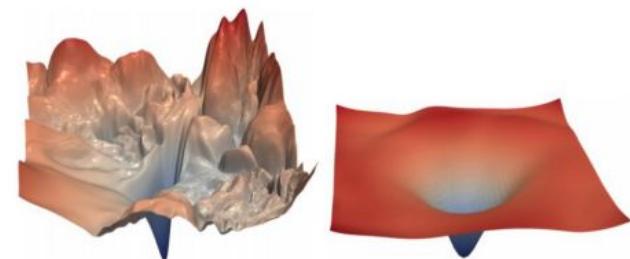
- Instead of $X^{(i)} = \text{Layer}(X^{(i-1)})$ (where i represents the layer)



- 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]

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

- **Layer normalization** is a trick to help models train faster.

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**.

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](#)]

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.

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; $\mu \in \mathbb{R}$.

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; $\sigma \in \mathbb{R}$.

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!)

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:

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

Normalize by scalar
mean and variance

Modulate by learned
elementwise gain and bias

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

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

- “**Scaled Dot Product**” attention is a final variation to aid in Transformer training.

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.

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.

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:

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

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:

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

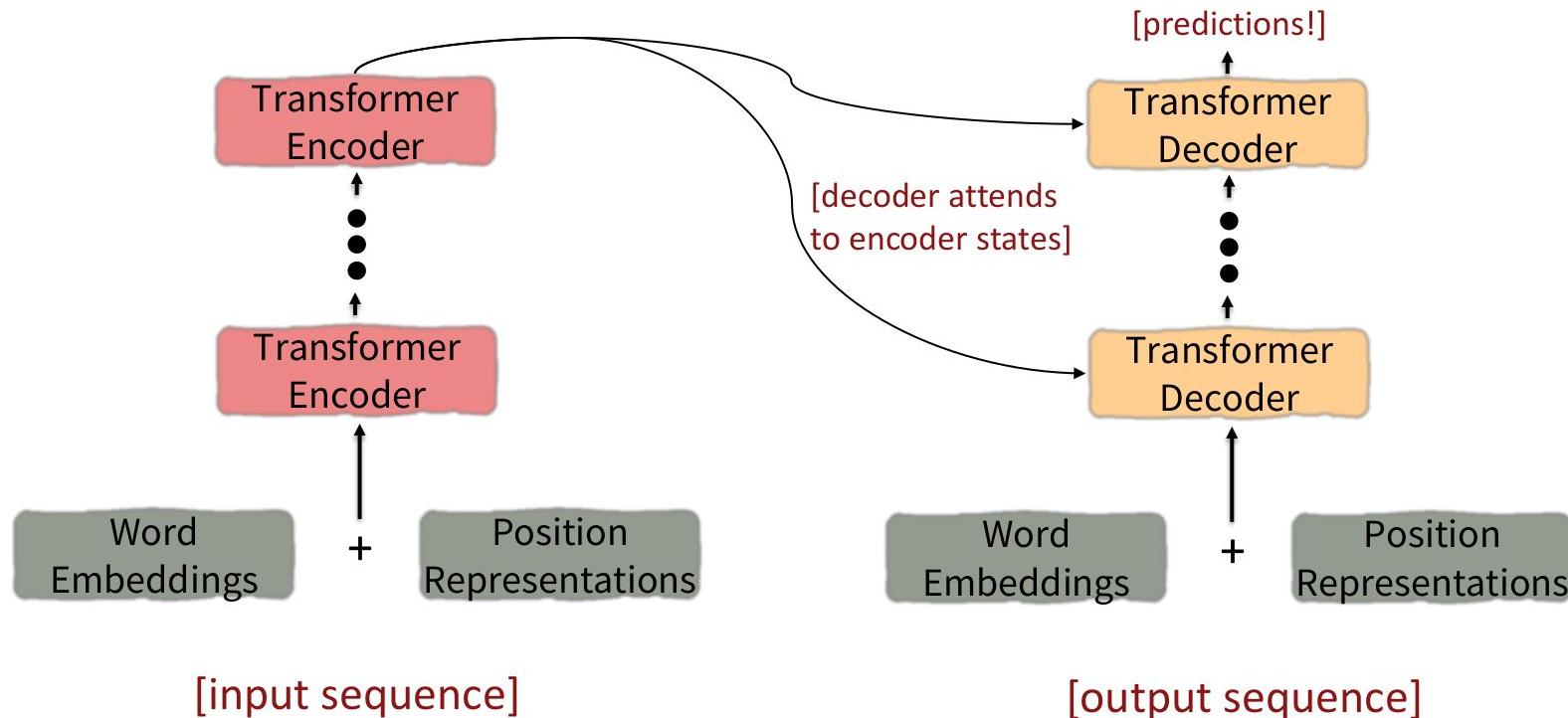
d/h is given by the multi-head attention size

- 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.)

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

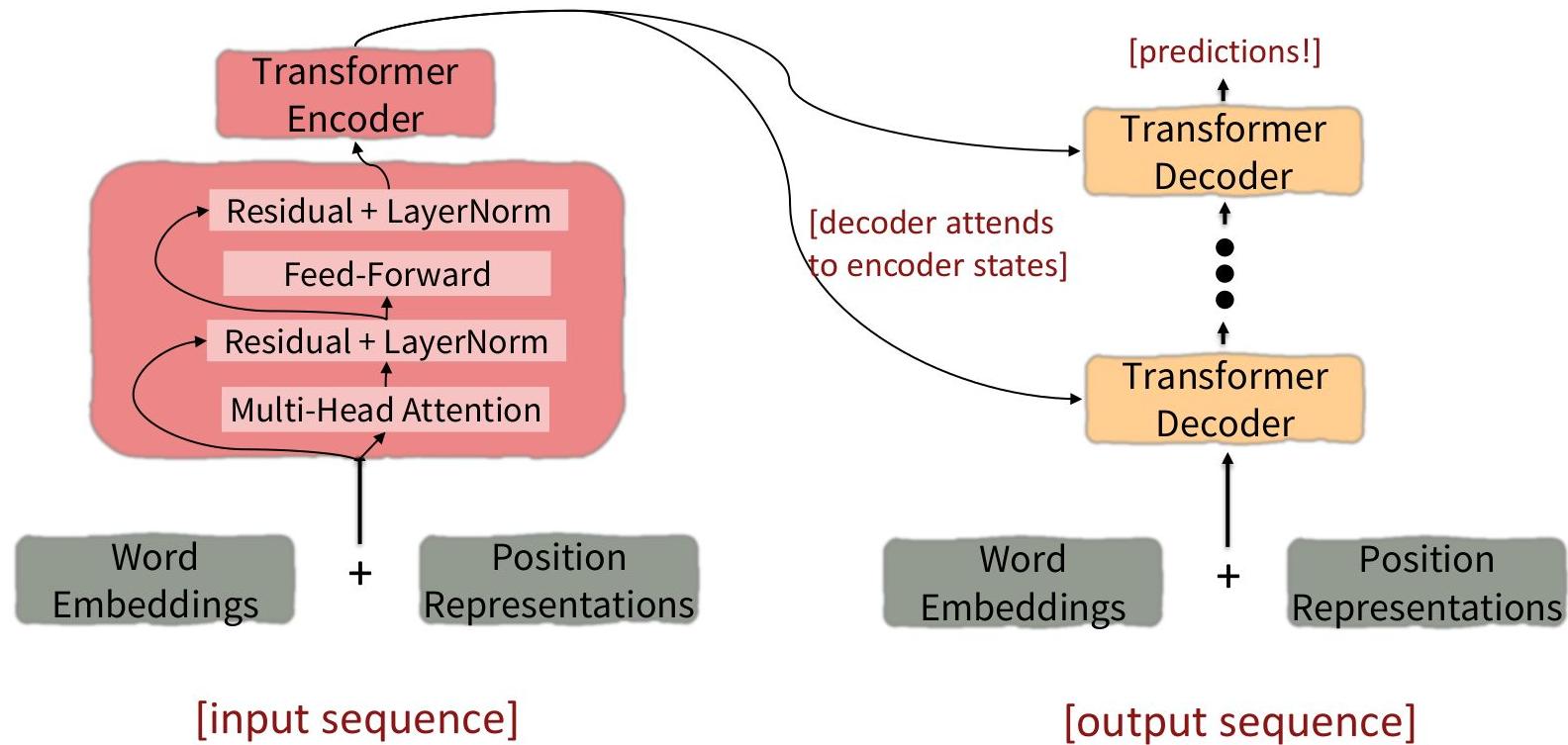
The Transformer Encoder-Decoder [Vaswani et al., 2017]

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



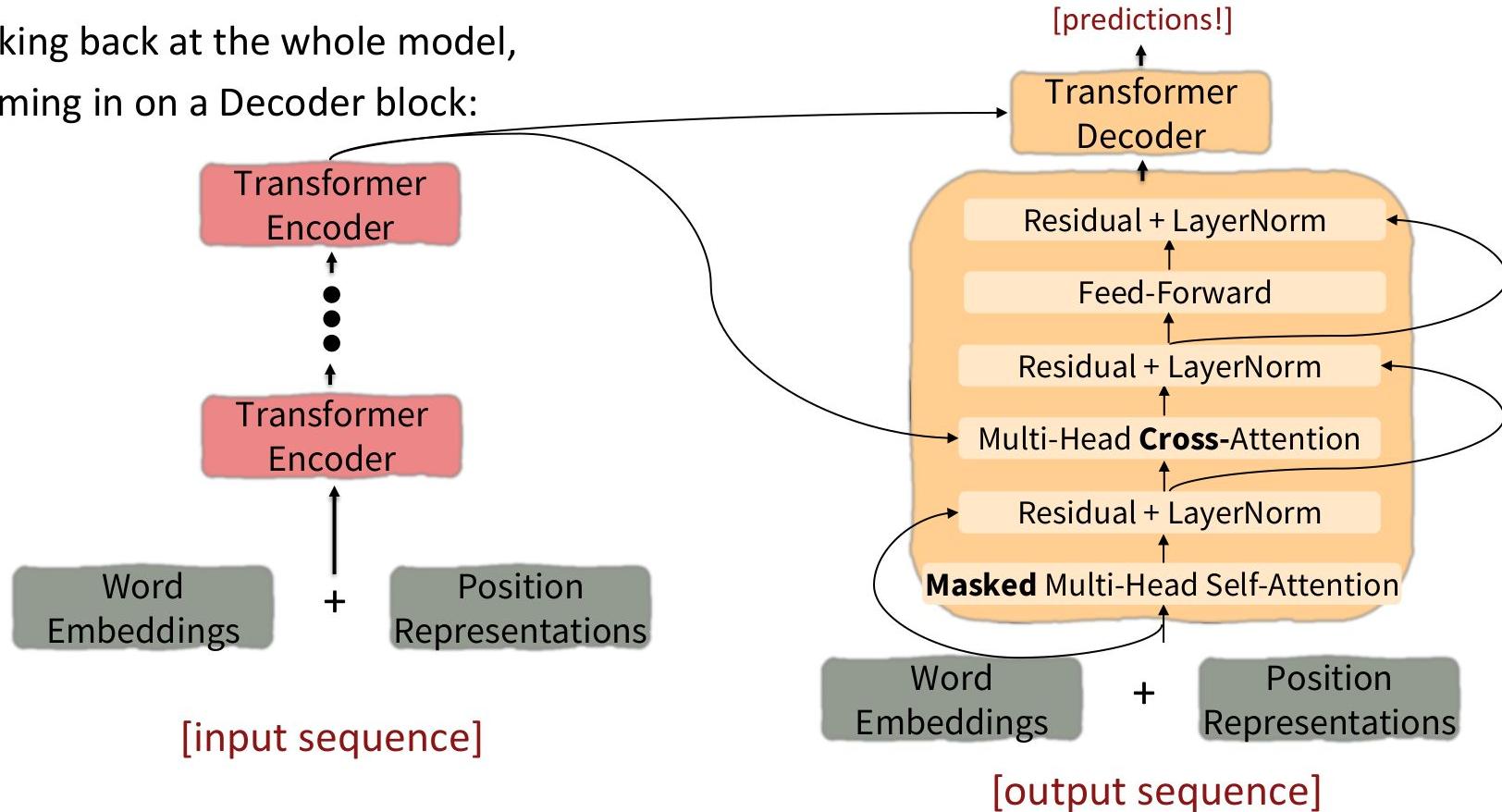
The Transformer Encoder-Decoder [Vaswani et al., 2017]

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



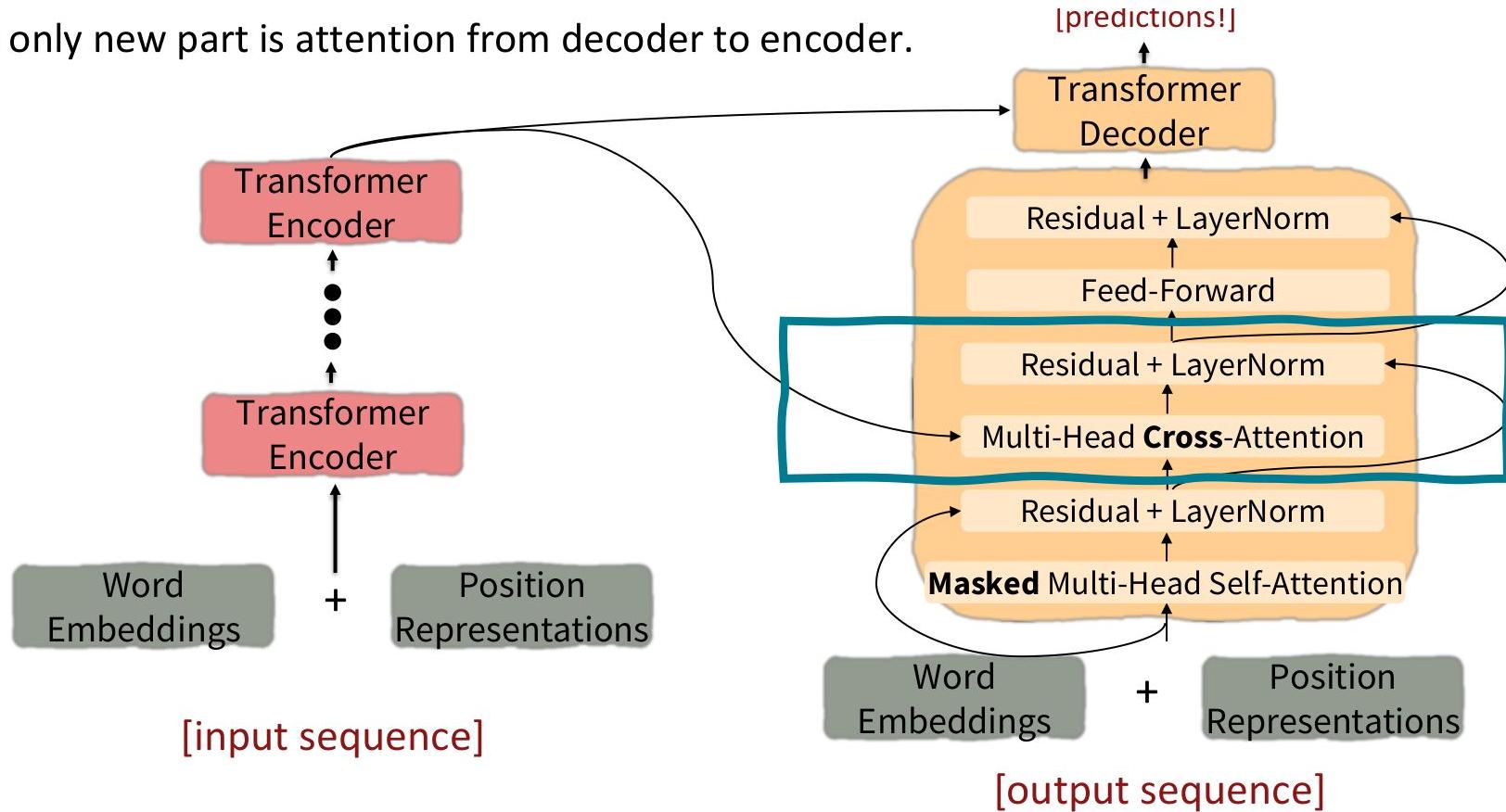
The Transformer Encoder-Decoder [Vaswani et al., 2017]

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.



The Transformer Decoder: **Cross-attention (details)**

The Transformer Decoder: **Cross-attention (details)**

- We saw that self-attention is when keys, queries, and values come from the same source.

The Transformer Decoder: Cross-attention (details)

- We saw that self-attention is when keys, queries, and values come from the same source.
- Let h_1, \dots, h_T be **output** vectors **from** the Transformer **encoder**; $x_i \in \mathbb{R}^d$

The Transformer Decoder: **Cross-attention (details)**

- We saw that self-attention is when keys, queries, and values come from the same source.
- 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**, $z_i \in \mathbb{R}^d$

The Transformer Decoder: **Cross-attention (details)**

- We saw that self-attention is when keys, queries, and values come from the same source.
- 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):
 - $k_i = Kh_i, v_i = Vh_i$.

The Transformer Decoder: Cross-attention (details)

- We saw that self-attention is when keys, queries, and values come from the same source.
- 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 Decoder: **Cross-attention (details)**

The Transformer Decoder: **Cross-attention (details)**

- Let's look at how cross-attention is computed, in matrices.

The Transformer Decoder: **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.

The Transformer Decoder: **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.

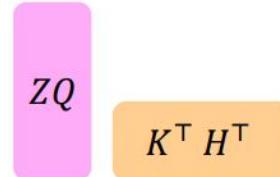
The Transformer Decoder: **Cross-attention (details)**

- 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 $\text{output} = \text{softmax}(ZQ(HK)^\top) \times HV$.

The Transformer Decoder: **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: $ZQ(HK)^\top$



The Transformer Decoder: 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: $ZQ(HK)^\top$

$$\begin{matrix} ZQ \\ K^\top H^\top \end{matrix} = \begin{matrix} ZQK^\top H^\top \\ \in \mathbb{R}^{T \times T} \end{matrix} \quad \text{All pairs of attention scores!}$$

The Transformer Decoder: 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: $ZQ(HK)^\top$

The diagram illustrates the computation of cross-attention scores. It starts with three components: a pink box labeled ZQ , an orange box labeled $K^\top H^\top$, and a brown box labeled $ZQK^\top H^\top \in \mathbb{R}^{T \times T}$. An equals sign connects the first two boxes to the third. To the right of the third box is the text "All pairs of attention scores!". Below this, a curved arrow points from the brown box down to another brown box containing the expression $ZQK^\top H^\top$, which is followed by an orange box labeled HV . This second brown box is enclosed in a large bracket, indicating it is the input to a softmax function. The text "softmax" is written to the left of this bracket.

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

The Transformer Decoder: 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.

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

Outline

1. From recurrence (RNN) to attention-based NLP models
2. Introducing the Transformer model
3. Great results with Transformers
4. Drawbacks and variants of Transformers

Great Results with Transformers

First, Machine Translation from the original Transformers paper!

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	41.29	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$

Great Results with Transformers

Next, document generation!

Model	Test perplexity	ROUGE-L
<i>seq2seq-attention, L = 500</i>	5.04952	12.7
<i>Transformer-ED, L = 500</i>	2.46645	34.2
<i>Transformer-D, L = 4000</i>	2.22216	33.6
<i>Transformer-DMCA, no MoE-layer, L = 11000</i>	2.05159	36.2
<i>Transformer-DMCA, MoE-128, L = 11000</i>	1.92871	37.9
<i>Transformer-DMCA, MoE-256, L = 7500</i>	1.90325	38.8

The old standard

Transformers all the way down.

Outline

1. From recurrence (RNN) to attention-based NLP models
2. Introducing the Transformer model
3. Great results with Transformers
4. Drawbacks and variants of Transformers

What would we like to fix about the Transformer?

What would we like to fix about the Transformer?

- **Quadratic compute in self-attention (today):**

What would we like to fix about the Transformer?

- **Quadratic compute in self-attention (today):**
 - Computing all pairs of interactions means our computation grows **quadratically** with the sequence length!

What would we like to fix about the Transformer?

- **Quadratic compute in self-attention (today):**
 - Computing all pairs of interactions means our computation grows **quadratically** with the sequence length!
 - For recurrent models, it only grew linearly!

What would we like to fix about the Transformer?

- **Quadratic compute in self-attention (today):**
 - Computing all pairs of interactions means our computation grows **quadratically** with the sequence length!
 - For recurrent models, it only grew linearly!
- **Position representations:**

What would we like to fix about the Transformer?

- **Quadratic compute in self-attention (today):**
 - Computing all pairs of interactions means our computation grows **quadratically** with the sequence length!
 - For recurrent models, it only grew linearly!
- **Position representations:**
 - Are simple absolute indices the best we can do to represent position?

What would we like to fix about the Transformer?

- **Quadratic compute in self-attention (today):**
 - Computing all pairs of interactions means our computation grows **quadratically** with the sequence length!
 - For recurrent models, it only grew linearly!
- **Position representations:**
 - Are simple absolute indices the best we can do to represent position?
 - Relative linear position attention [\[Shaw et al., 2018\]](#)

What would we like to fix about the Transformer?

- **Quadratic compute in self-attention (today):**
 - Computing all pairs of interactions means our computation grows **quadratically** with the sequence length!
 - For recurrent models, it only grew linearly!
- **Position representations:**
 - Are simple absolute indices the best we can do to represent position?
 - Relative linear position attention [\[Shaw et al., 2018\]](#)
 - Dependency syntax-based position [\[Wang et al., 2019\]](#)

Quadratic computation as a function of sequence length

Quadratic computation as a function of sequence length

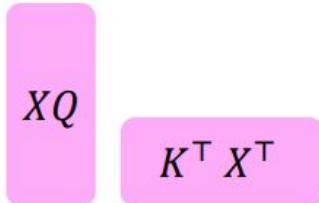
- One of the benefits of self-attention over recurrence was that it's highly parallelizable.

Quadratic computation as a function of sequence length

- One of the benefits of self-attention over recurrence was that it's highly parallelizable.
- However, its total number of operations grows as $O(T^2d)$, where T is the sequence length, and d is the dimensionality.

Quadratic computation as a function of sequence length

- One of the benefits of self-attention over recurrence was that it's highly parallelizable.
- However, its total number of operations grows as $O(T^2d)$, where T is the sequence length, and d is the dimensionality.


$$\begin{matrix} XQ \\ K^T X^T \end{matrix}$$

Quadratic computation as a function of sequence length

- One of the benefits of self-attention over recurrence was that it's highly parallelizable.
- However, its total number of operations grows as $O(T^2d)$, where T is the sequence length, and d is the dimensionality.

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

Need to compute all pairs of interactions!
 $O(T^2d)$

Quadratic computation as a function of sequence length

- One of the benefits of self-attention over recurrence was that it's highly parallelizable.
- However, its total number of operations grows as $O(T^2d)$, where T is the sequence length, and d is the dimensionality.

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

Need to compute all pairs of interactions!
 $O(T^2d)$

- Think of d as around **1,000**.

Quadratic computation as a function of sequence length

- One of the benefits of self-attention over recurrence was that it's highly parallelizable.
- However, its total number of operations grows as $O(T^2d)$, where T is the sequence length, and d is the dimensionality.

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

Need to compute all pairs of interactions!
 $O(T^2d)$

- Think of d as around **1,000**.
 - So, for a single (shortish) sentence, $T \leq 30$; $T^2 \leq \mathbf{900}$.

Quadratic computation as a function of sequence length

- One of the benefits of self-attention over recurrence was that it's highly parallelizable.
- However, its total number of operations grows as $O(T^2d)$, where T is the sequence length, and d is the dimensionality.

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

Need to compute all pairs of interactions!
 $O(T^2d)$

- Think of d as around **1,000**.
 - So, for a single (shortish) sentence, $T \leq 30$; $T^2 \leq \mathbf{900}$.
 - In practice, we set a bound like $T = 512$.

Quadratic computation as a function of sequence length

- One of the benefits of self-attention over recurrence was that it's highly parallelizable.
- However, its total number of operations grows as $O(T^2d)$, where T is the sequence length, and d is the dimensionality.

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

Need to compute all pairs of interactions!
 $O(T^2d)$

- Think of d as around **1,000**.
 - So, for a single (shortish) sentence, $T \leq 30$; $T^2 \leq \mathbf{900}$.
 - In practice, we set a bound like $T = 512$.
 - **But what if we'd like $T \geq 10,000$?** For example, to work on long documents?

Recent work on improving on quadratic self-attention cost

Recent work on improving on quadratic self-attention cost

- Considerable recent work has gone into the question, *Can we build models like Transformers without paying the $O(T^2)$ all-pairs self-attention cost?*

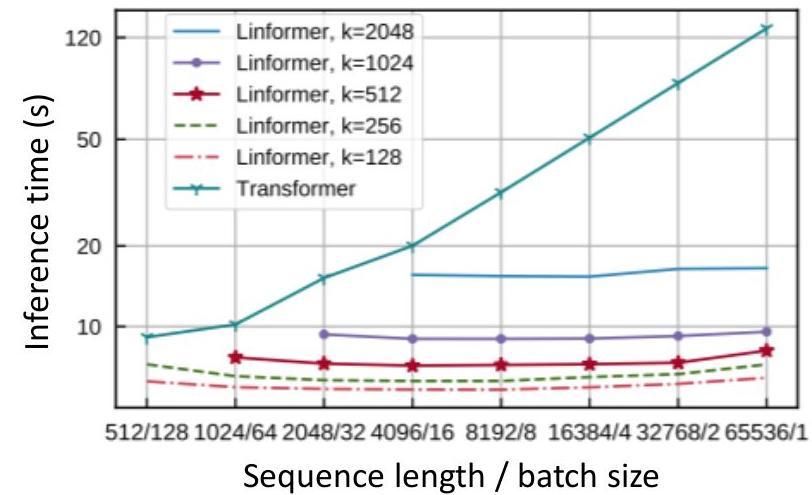
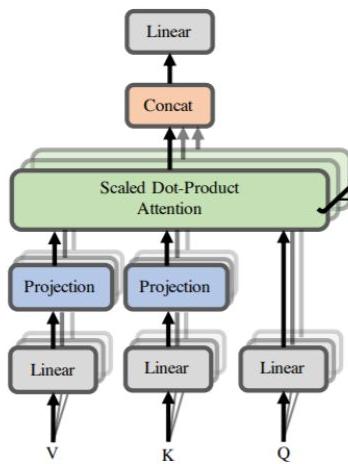
Recent work on improving on quadratic self-attention cost

- Considerable recent work has gone into the question, *Can we build models like Transformers without paying the $O(T^2)$ all-pairs self-attention cost?*
- For example, **Linformer** [[Wang et al., 2020](#)]

Recent work on improving on quadratic self-attention cost

- Considerable recent work has gone into the question, *Can we build models like Transformers without paying the $O(T^2)$ all-pairs self-attention cost?*
- For example, **Linformer** [[Wang et al., 2020](#)]

Key idea: map the sequence length dimension to a lower-dimensional space for values, keys



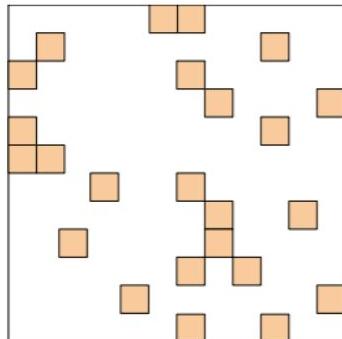
Recent work on improving on quadratic self-attention cost

- Considerable recent work has gone into the question, *Can we build models like Transformers without paying the $O(T^2)$ all-pairs self-attention cost?*
- For example, **BigBird** [[Zaheer et al., 2021](#)]

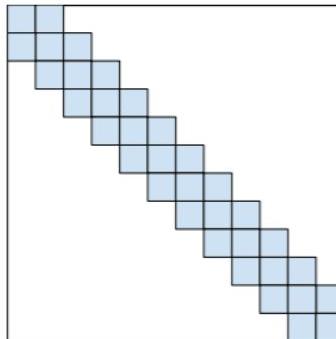
Recent work on improving on quadratic self-attention cost

- Considerable recent work has gone into the question, *Can we build models like Transformers without paying the $O(T^2)$ all-pairs self-attention cost?*
- For example, **BigBird** [\[Zaheer et al., 2021\]](#)

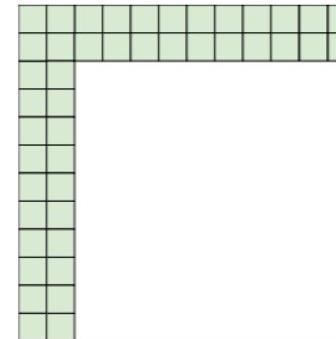
Key idea: replace all-pairs interactions with a family of other interactions, **like local windows, looking at everything, and random interactions.**



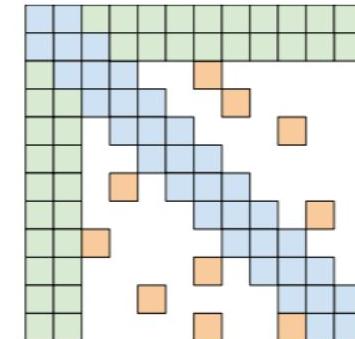
(a) Random attention



(b) Window attention



(c) Global Attention



(d) BIGBIRD