ENGINEERING
W Booth School of Engineering
Practice and Technology

McMaster University

Computational Natural Language Processing

Self-Attention and Transformers

Hamidreza Mahyar
mahyarh@mcmaster.ca

Lecture Plan

1. From recurrence (RNN) to attention-based NLP models
2. The Transformer model
3. Great results with Transformers

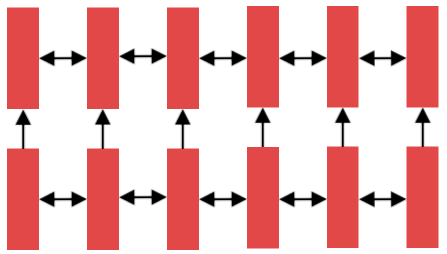
Reminders:

Assignment 1 is due today!

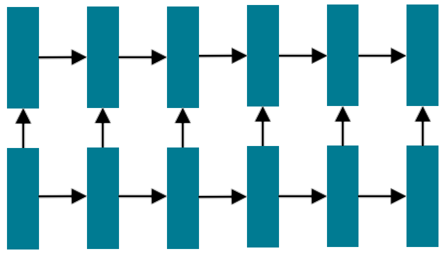
Assignment 2 will be out today!

As of last lecture: recurrent models for (most) NLP!

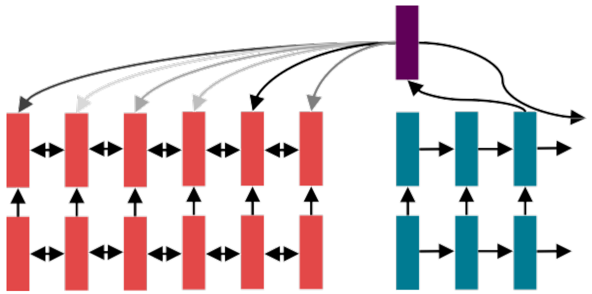
- Circa 2016, the de facto strategy in NLP is to **encode** sentences with a bidirectional LSTM: (for example, the source sentence in a translation)



- Define your output (parse, sentence, summary) as a sequence, and use an LSTM to generate it.

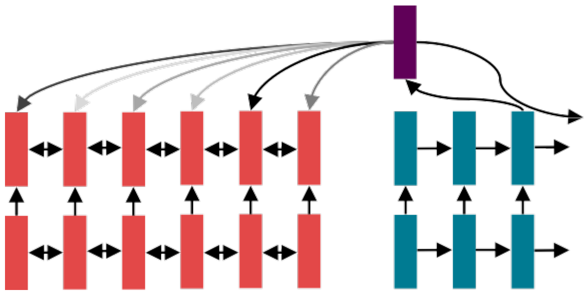


- Use attention to allow flexible access to memory



Today: Same goals, different building blocks

- So far, we learned about sequence-to-sequence problems and encoder-decoder models.
- 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.



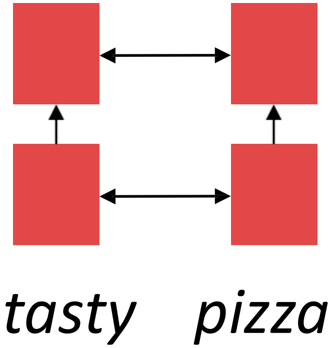
Lots of trial and error

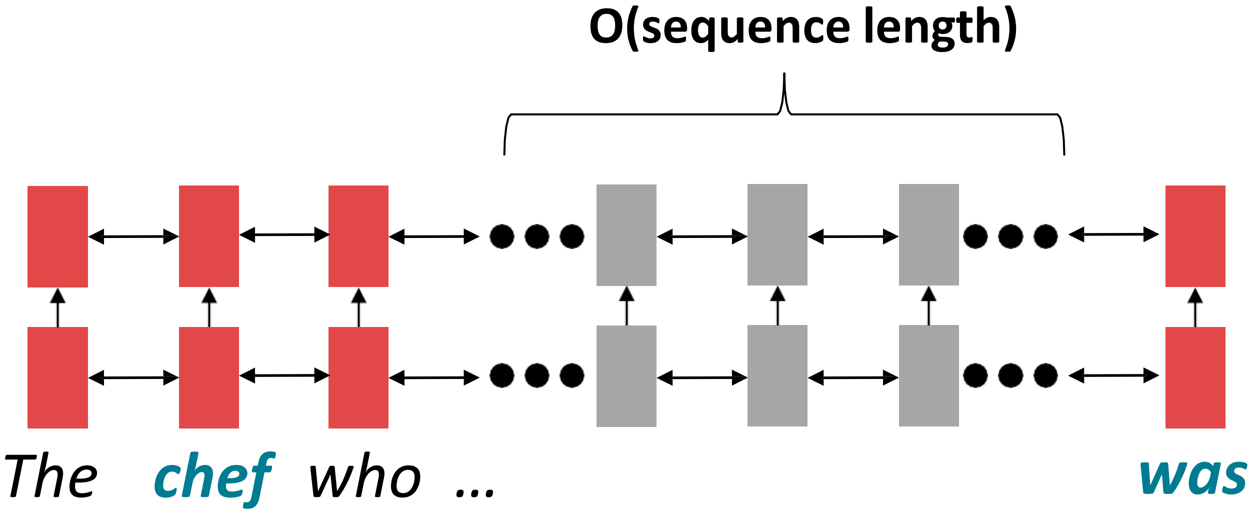
2014-2017ish
Recurrence

2021
??????

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

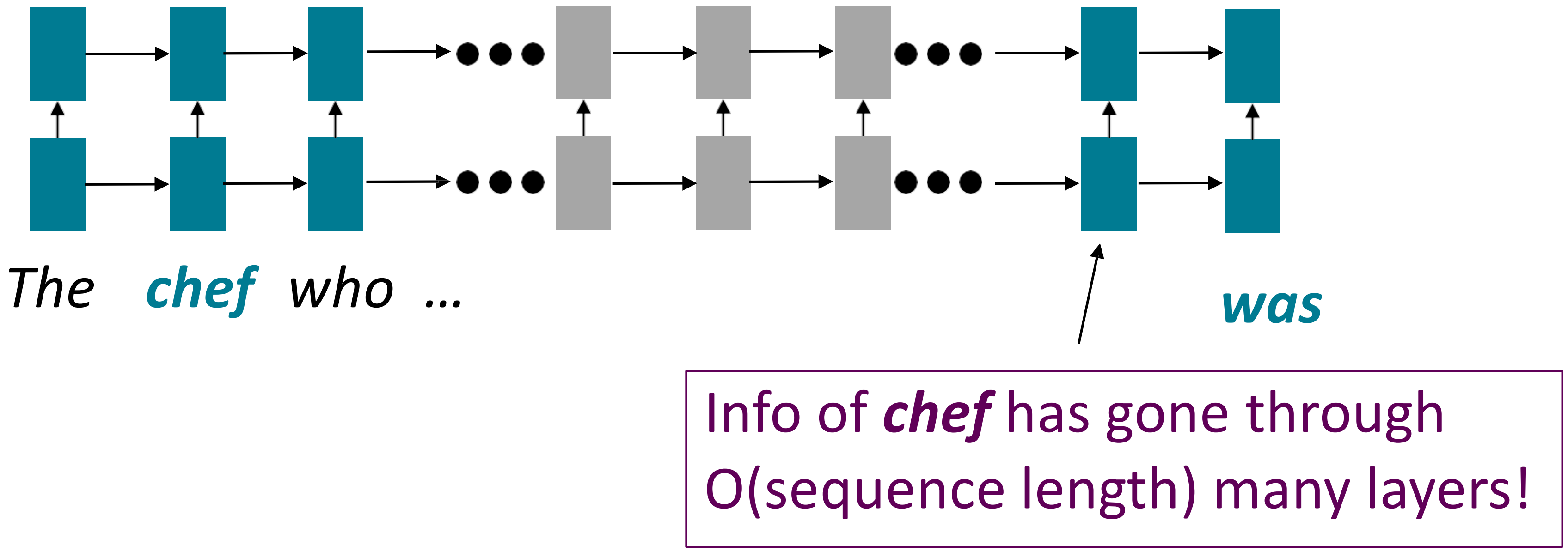
- RNNs are unrolled “left-to-right”.
- This encodes linear locality: a useful heuristic!
  - Nearby words often affect each other’s meanings
- **Problem:** RNNs take **O(sequence length)** steps for distant word pairs to interact.





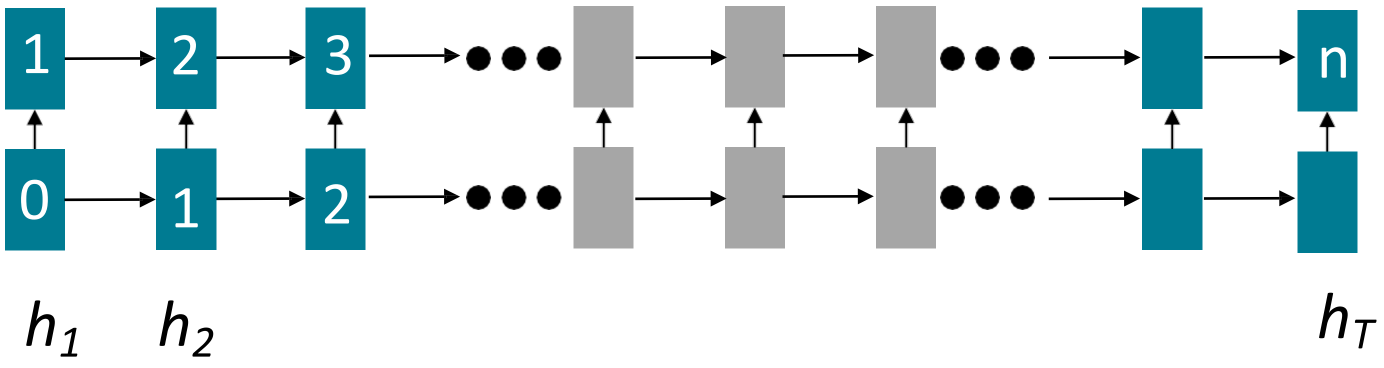
Issues with recurrent models: **Linear interaction distance**

- **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…



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

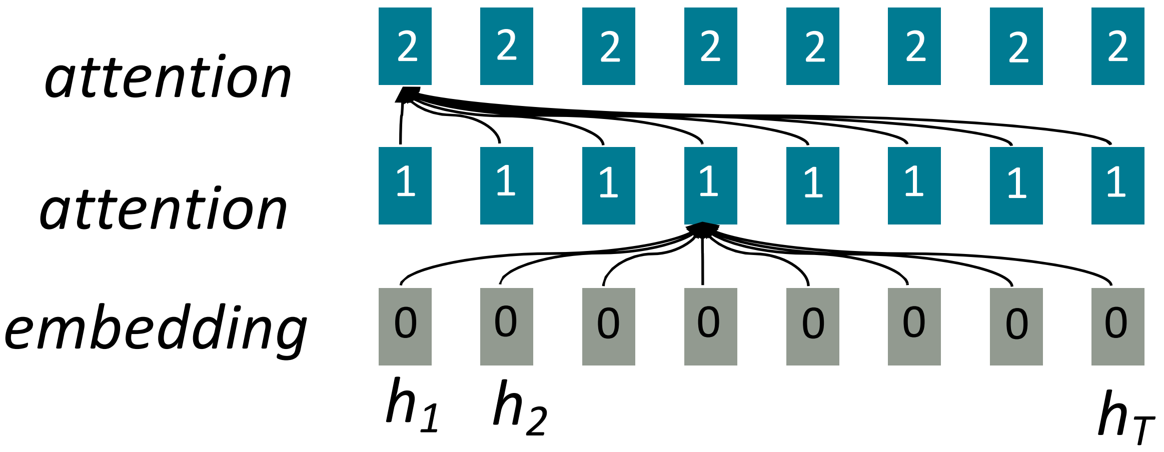
- Forward and backward passes have **O(sequence length)** unparallelizable operations
  - GPUs can perform a bunch of independent computations at once!
  - But future RNN hidden states can't be computed in full before past RNN hidden states have been computed
  - Inhibits training on very large datasets!



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.**
  - 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 with sequence length.
- Maximum interaction distance: O(1), since all words interact at every layer!

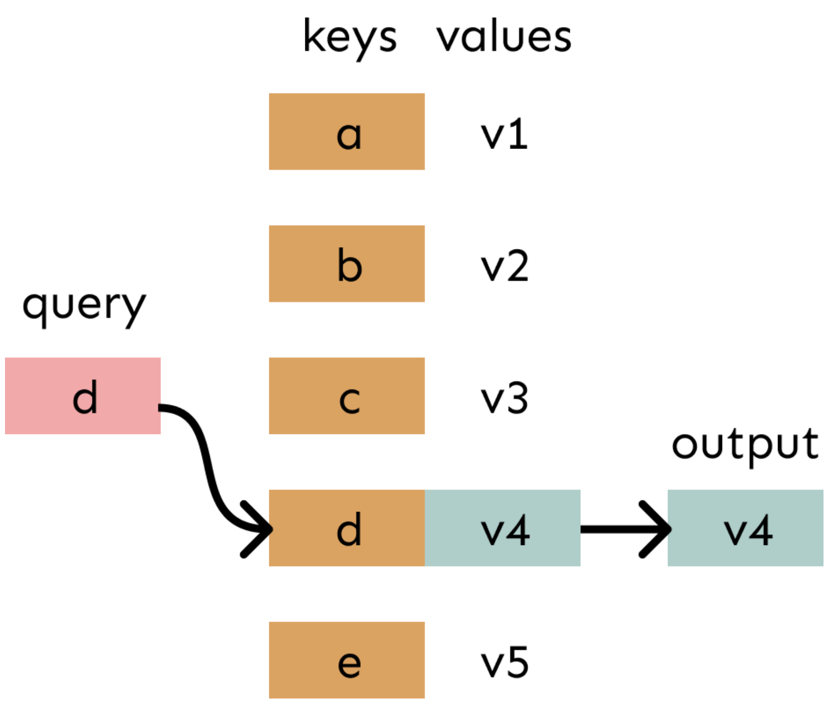


All words attend to all words in previous layer; most arrows here are omitted

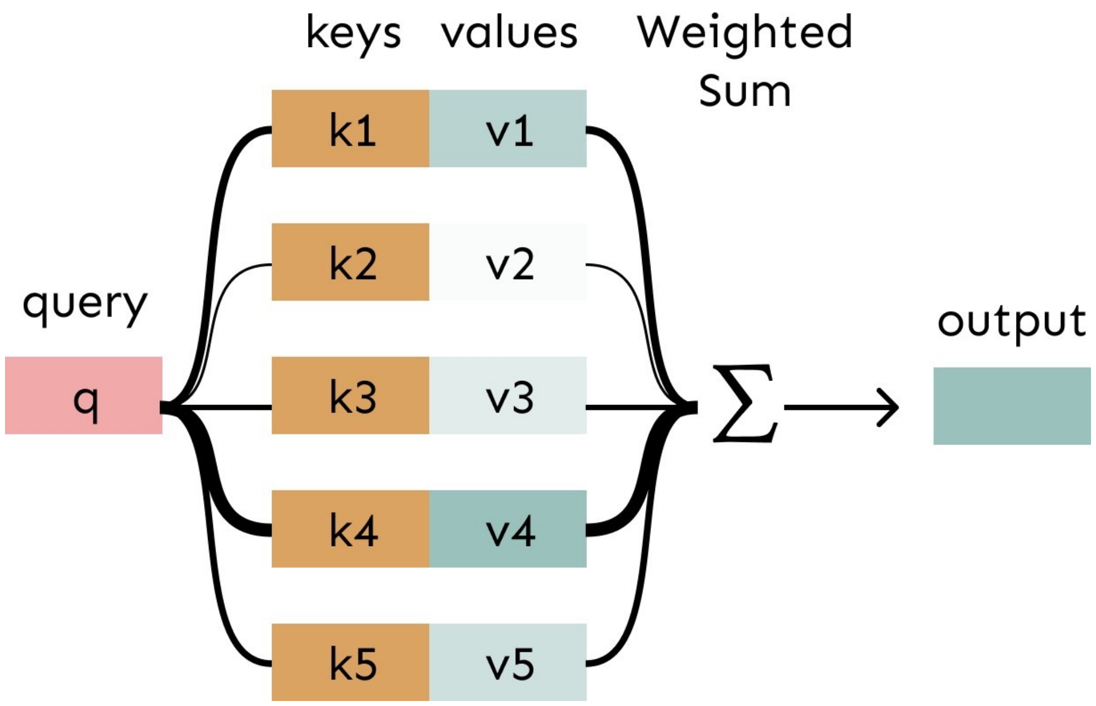
Attention as a soft, averaging lookup table

We can think of **attention** as performing fuzzy lookup in a key-value store.

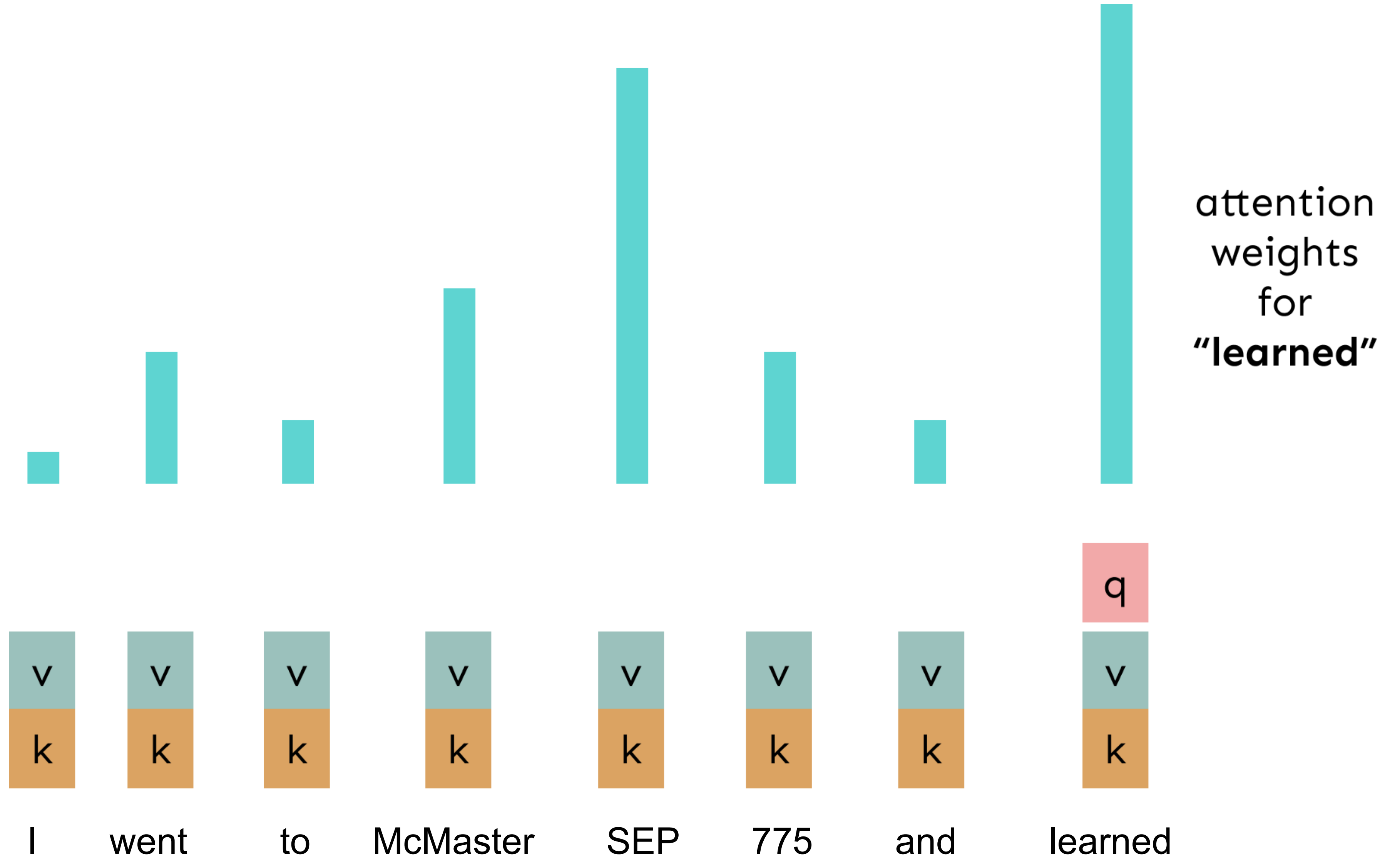
In a **lookup table**, we have a table of keys that map to values. The query matches one of the keys, returning its value.



In **attention**, the query matches all keys *softly*, to a weight between 0 and 1. The keys' values are multiplied by the weights and summed.



Self-Attention Hypothetical Example



Self-Attention: keys, queries, values from the same sequence

Let $\boldsymbol{w}_{1:n}$ be a sequence of words in vocabulary V, like *Zuko made his uncle tea*.

For each $\boldsymbol{w}_i$, let $\boldsymbol{x}_i = E\boldsymbol{w}_{\boldsymbol{i}}$, where $E \in \mathbb{R}^{d\times|V|}$ is an embedding matrix.

1. Transform each word embedding with weight matrices Q, K, V , each in $\mathbb{R}^{d\times d}$

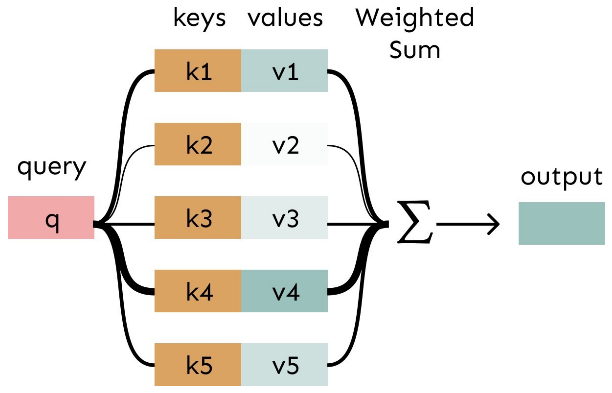
$\boldsymbol{q}_i = Q\boldsymbol{x}_{\boldsymbol{i}}$ (queries) $\quad$ $\boldsymbol{k}_i = K\boldsymbol{x}_{\boldsymbol{i}}$ (keys) $\quad$ $\boldsymbol{v}_i = V\boldsymbol{x}_{\boldsymbol{i}}$ (values)

2. Compute pairwise similarities between keys and queries; normalize with softmax

$$\boldsymbol{e}_{ij} = \boldsymbol{q}_{\boldsymbol{i}}^{\top}\boldsymbol{k}_{\boldsymbol{j}} \qquad \boldsymbol{\alpha}_{ij} = \frac{\exp(\boldsymbol{e}_{ij})}{\sum_{j'}\exp(\boldsymbol{e}_{ij'})}$$

3. Compute output for each word as weighted sum of values

$$\boldsymbol{o}_i = \sum_{\boldsymbol{j}} \boldsymbol{\alpha}_{ij}\,\boldsymbol{v}_i$$



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**

$\boldsymbol{p}_i \in \mathbb{R}^d$, for $i \in \{1,2,\ldots,n\}$ 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 $\boldsymbol{p}_i$ to our inputs!
- Recall that $\boldsymbol{x}_i$ is the embedding of the word at index i. The positioned embedding is:

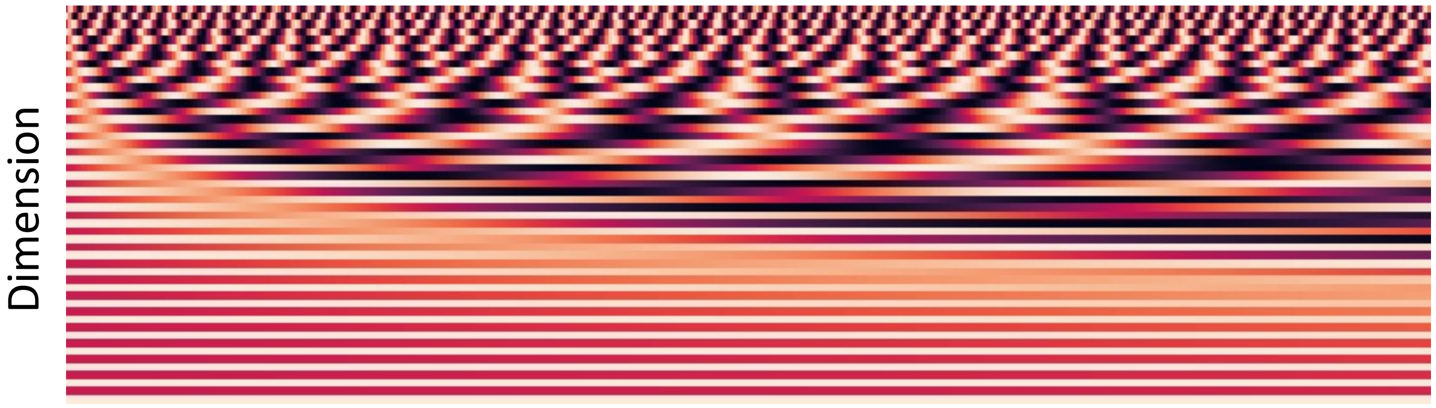
$$\tilde{\boldsymbol{x}}_i = \boldsymbol{x}_i + \boldsymbol{p}_i$$

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:

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



- 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!

Image: https://timodenk.com/blog/linear-relationships-in-the-transformers-positional-encoding/

Position representation vectors learned from scratch

- **Learned absolute position representations:** Let all p_i be learnable parameters!

  Learn a matrix $\boldsymbol{p} \in \mathbb{R}^{d \times n}$*, and let each* $\boldsymbol{p}_i$ *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 $1, \dots, n$.
- 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]

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

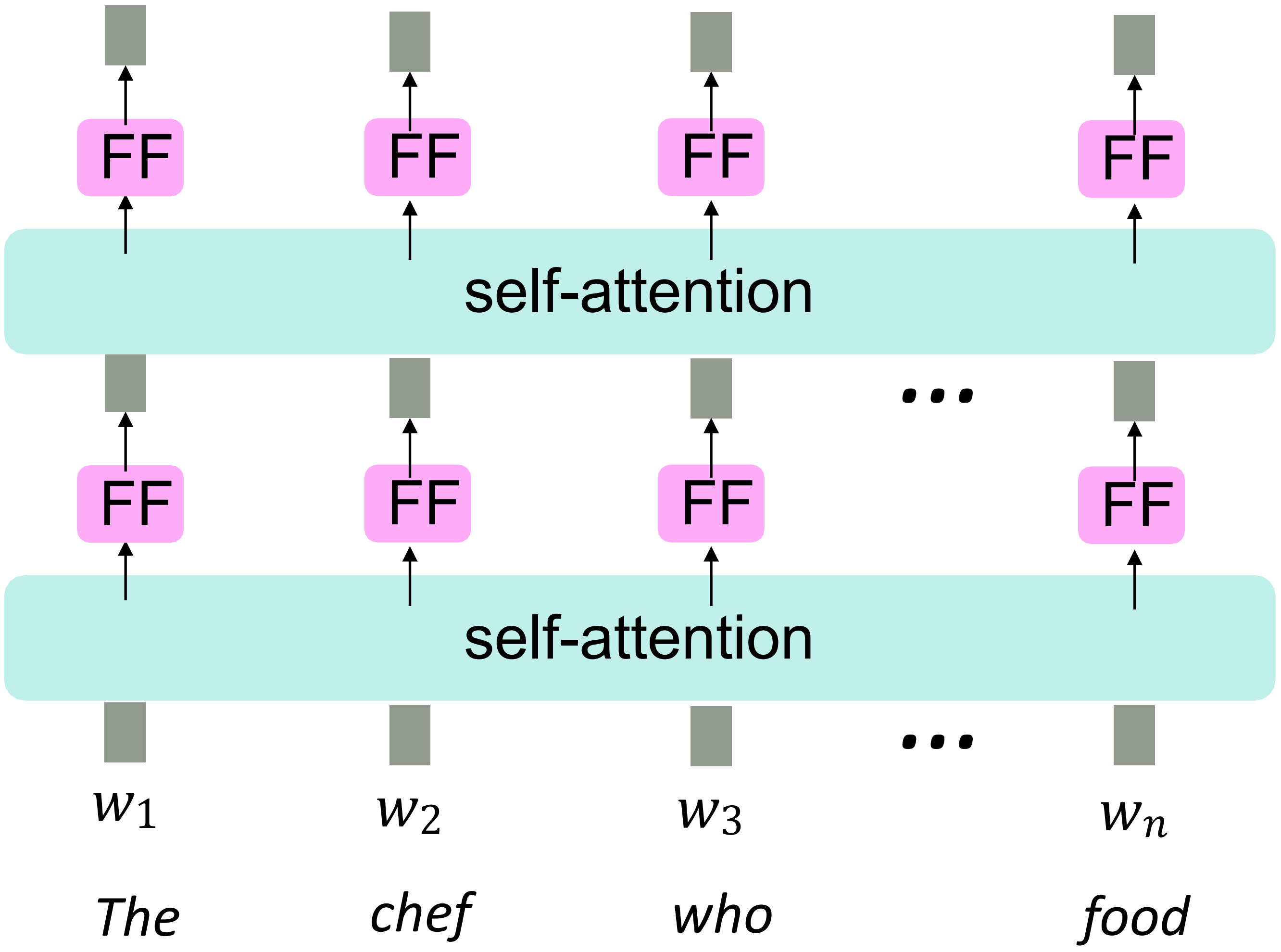
- Add position representations to the inputs

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 (Why?)

- Easy fix: add a **feed-forward network** to post-process each output vector.

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



Intuition: the FF network processes the result of attention

Barriers and solutions for Self-Attention as a building block

Barriers		**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 • 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 $-\infty$.

$$e_{ij} = \begin{cases} q_i^\top k_j, j \leq i \\ -\infty, j > i \end{cases}$$

We can look at these (not greyed out) words

For encoding these words

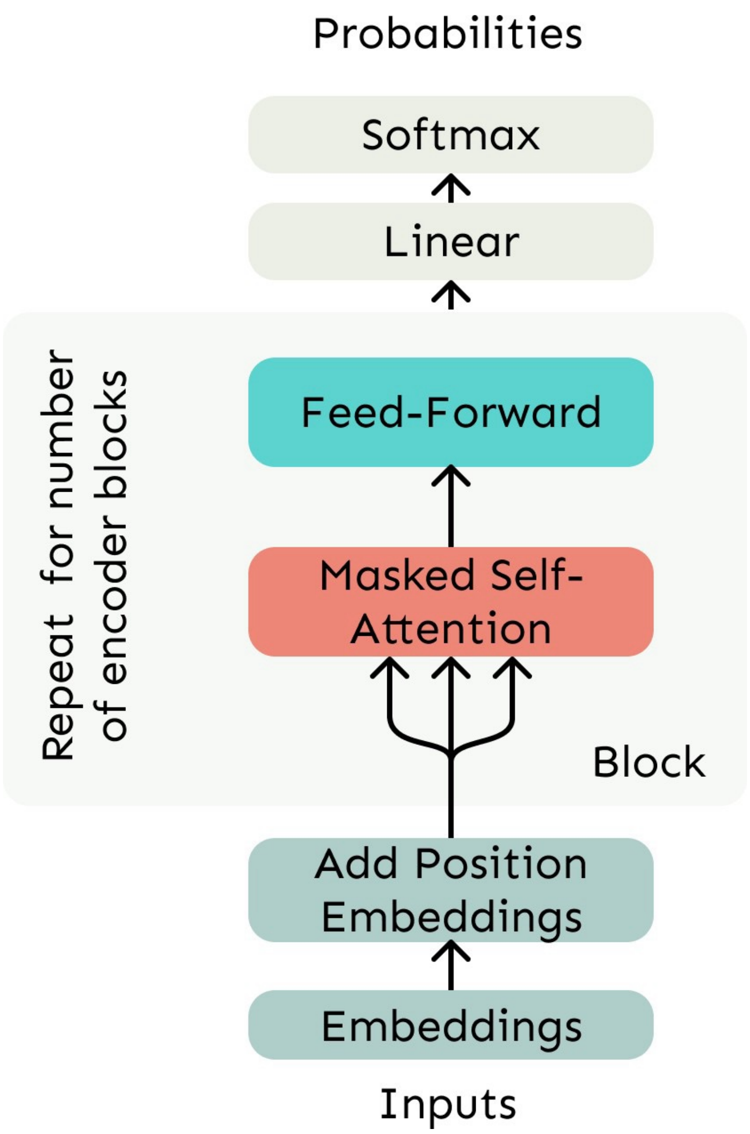
	[START]	The	chef	who
[START]		$-\infty$	$-\infty$	$-\infty$
The			$-\infty$	$-\infty$
chef				$-\infty$
who				

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 • 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.

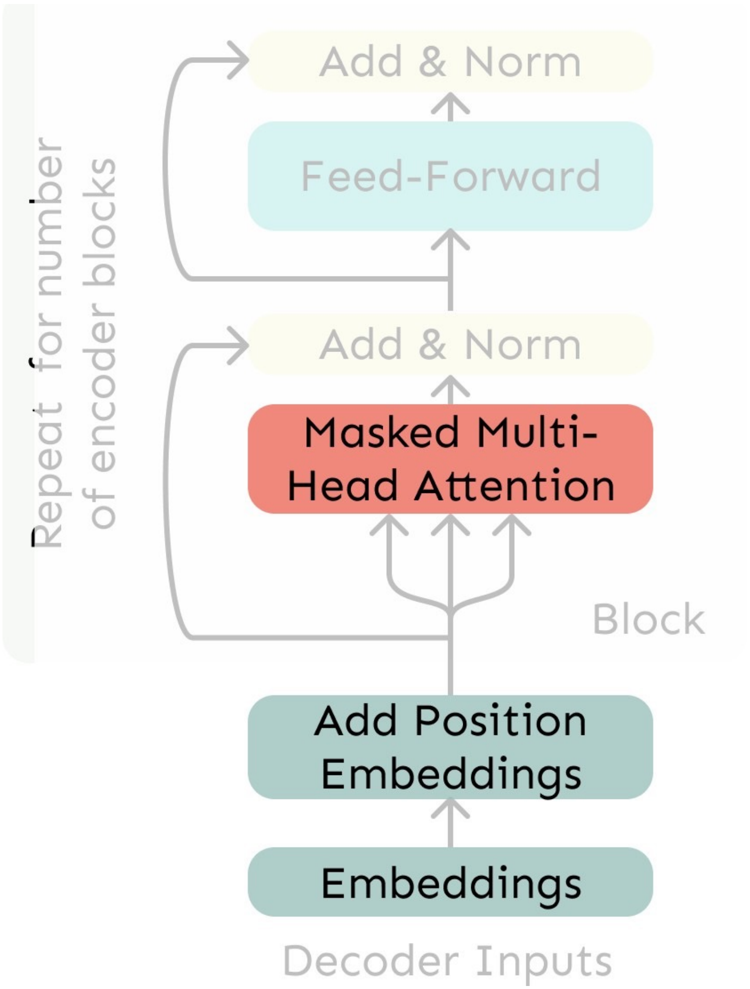


Outline

1. From recurrence (RNN) to attention-based NLP models
2. The Transformer model
3. Great results with Transformers

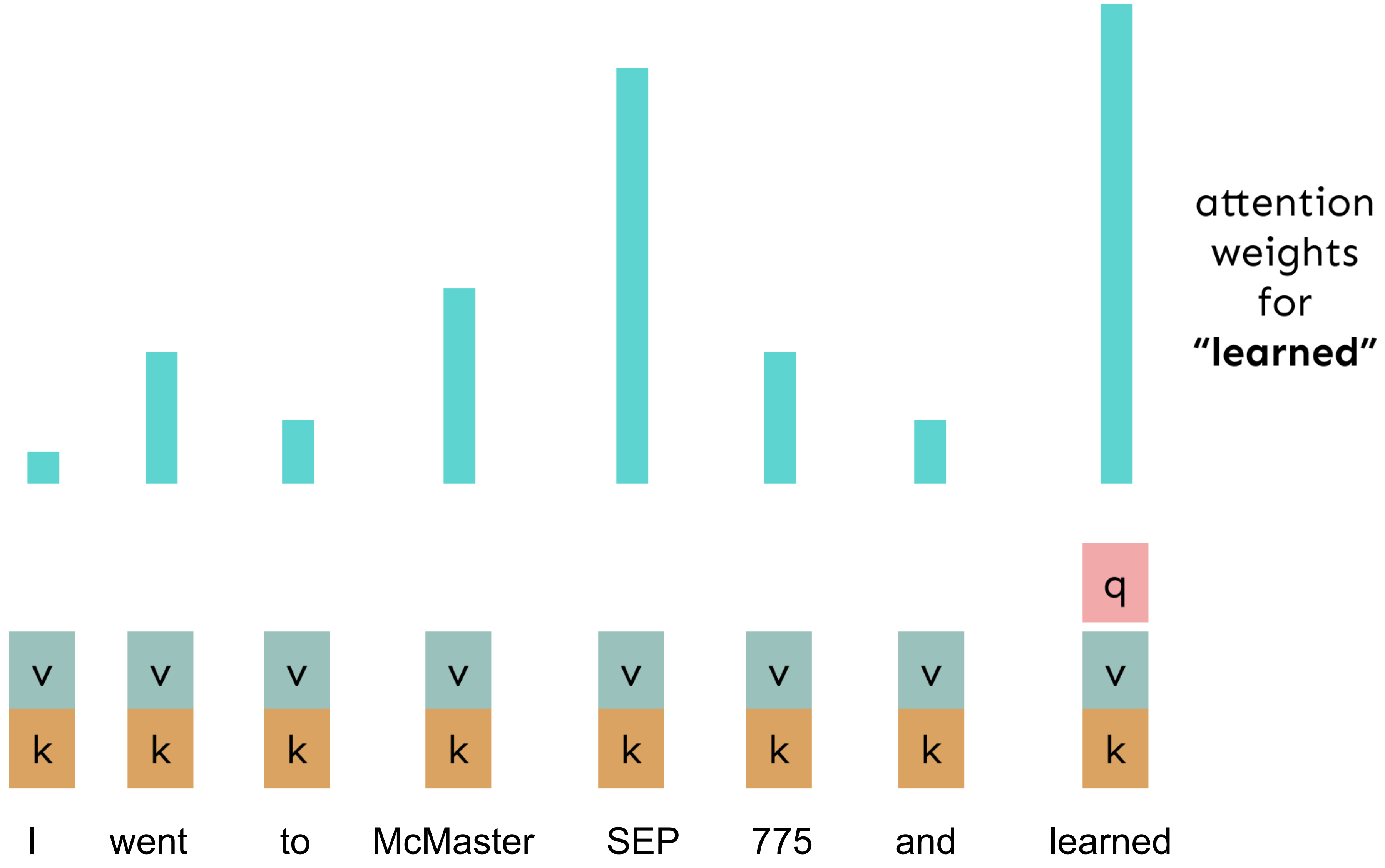
The Transformer Decoder

- A Transformer decoder is how we'll build systems like **language models**.
- It's a lot like our minimal self-attention architecture, but with a few more components.
- The embeddings and position embeddings are identical.
- We'll next replace our self-attention with **multi-head self-attention.**

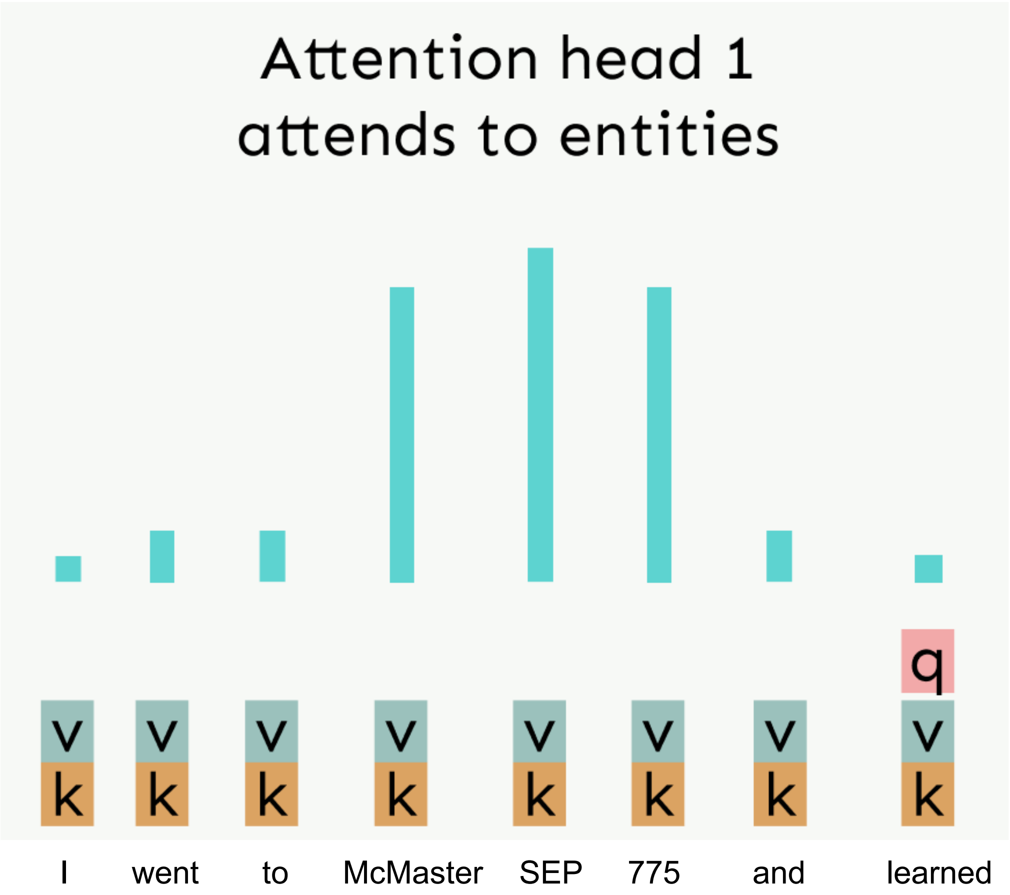


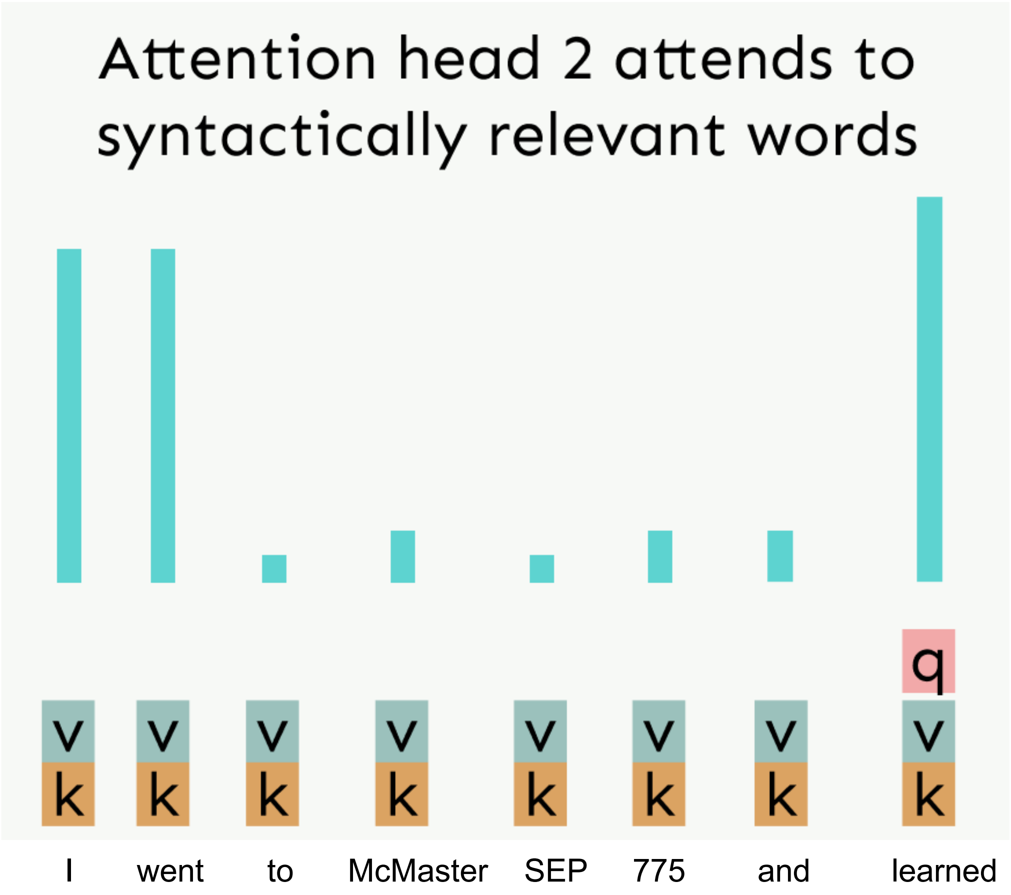
Transformer Decoder

Recall the Self-Attention Hypothetical Example



Hypothetical Example of Multi-Head Attention

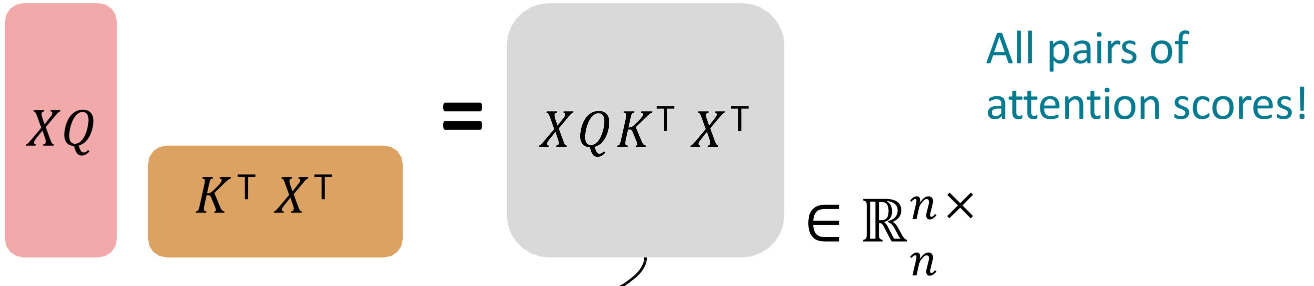




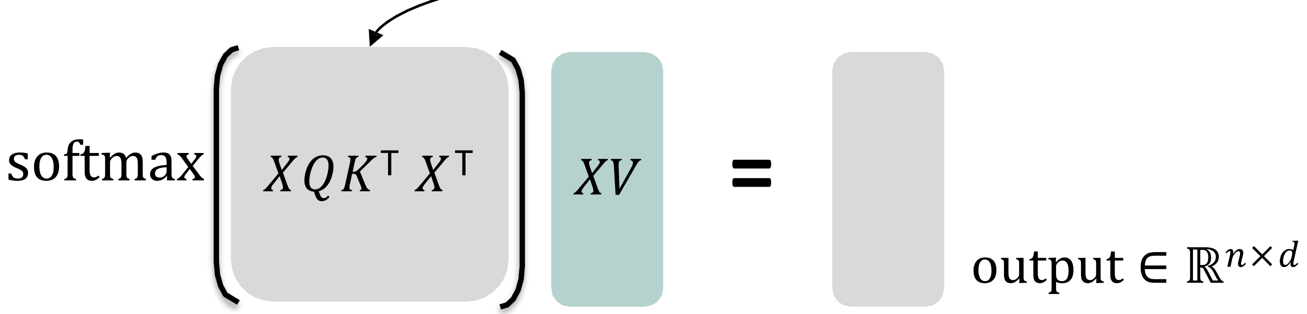
Sequence-Stacked form of Attention

- Let's look at how key-query-value attention is computed, in matrices.
  - Let $X = [x_1; \ldots; x_n] \in \mathbb{R}^{n\times d}$ be the concatenation of input vectors.
  - First, note that $XK \in \mathbb{R}^{n\times d}$, $XQ \in \mathbb{R}^{n\times d}$, $XV \in \mathbb{R}^{n\times d}$.
  - The output is defined as output $=$ $\text{softmax}(XQ(XK)^\top)XV \in\in \mathbb{R}^{n\times d}$.

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.



Multi-headed attention

- What if we want to look in multiple places in the sentence at once?
  - For word i, self-attention "looks" where $x_i^\top Q^\top K x_j$ 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:
  - $\text{output}_\ell = \text{softmax}\left(X Q_\ell K_\ell^\top X^\top\right) * X V_\ell$, where $\text{output}_\ell \in \mathbb{R}^{d/h}$
- Then the outputs of all the heads are combined!
  - $\text{output} = [\text{output}_1; \ldots; \text{output}_h] Y$, where $Y \in \mathbb{R}^{d \times d}$

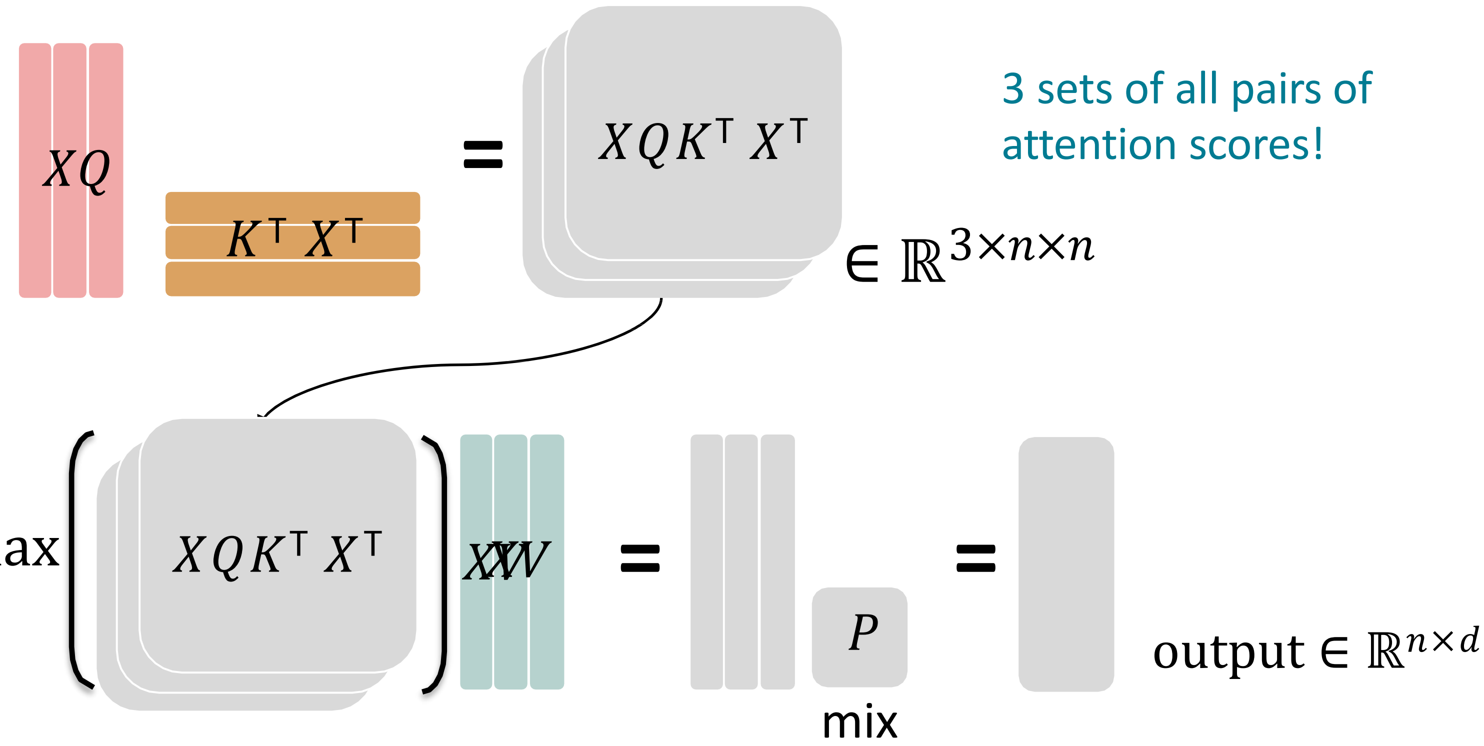
- Each head gets to "look" at different things, and construct value vectors differently.

Multi-head self-attention is computationally efficient

- Even though we compute h many attention heads, it's not really more costly.
  - We compute $XQ \in \mathbb{R}^{n \times d}$, and then reshape to $\mathbb{R}^{n \times h \times d/h}$. (Likewise for XK, XV.)
  - Then we transpose to $\mathbb{R}^{h \times n \times d/h}$; now the head axis is like a batch axis.
  - Almost everything else is identical, and the **matrices are the same sizes.**

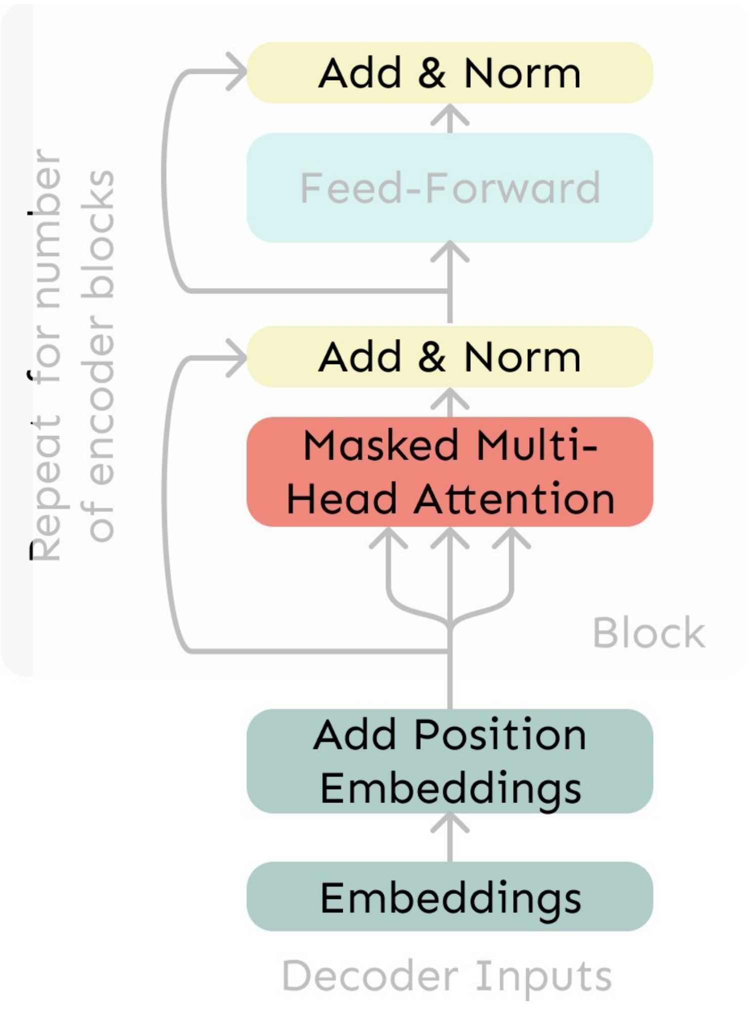
First, take the query-key dot products in one matrix multiplication: $XQ(XK)^{\top}$

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



The Transformer Decoder

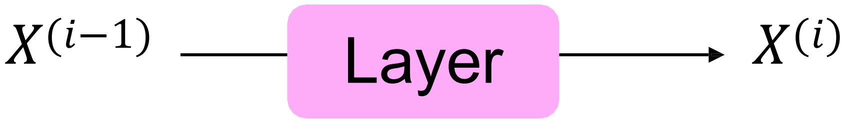
- Now that we've replaced self-attention with multi-head self-attention, we'll go through two **optimization tricks** that end up being :
  - **Residual Connections**
  - **Layer Normalization**
- In most Transformer diagrams, these are often written together as "Add & Norm"



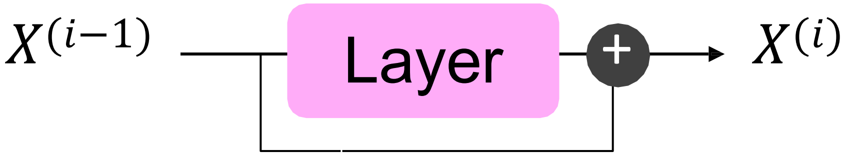
Transformer Decoder

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

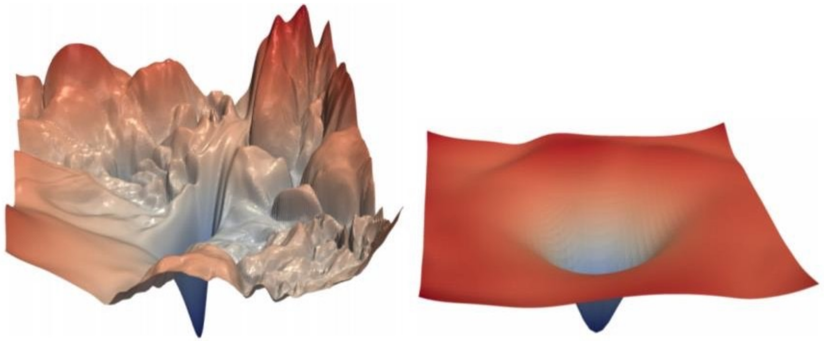
- **Residual connections** are a trick to help models train better.
  - Instead of $X^{(i)} = \text{Layer}(X^{(i-1)})$ (where i represents the layer)



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



  - Gradient is **great** through the residual connection; it's 1!
  - Bias towards the identity function!



[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 $\sigma = \sqrt{\frac{1}{d} \sum_{j=1}^{d} \left(x_j - \mu\right)^2}$; 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:

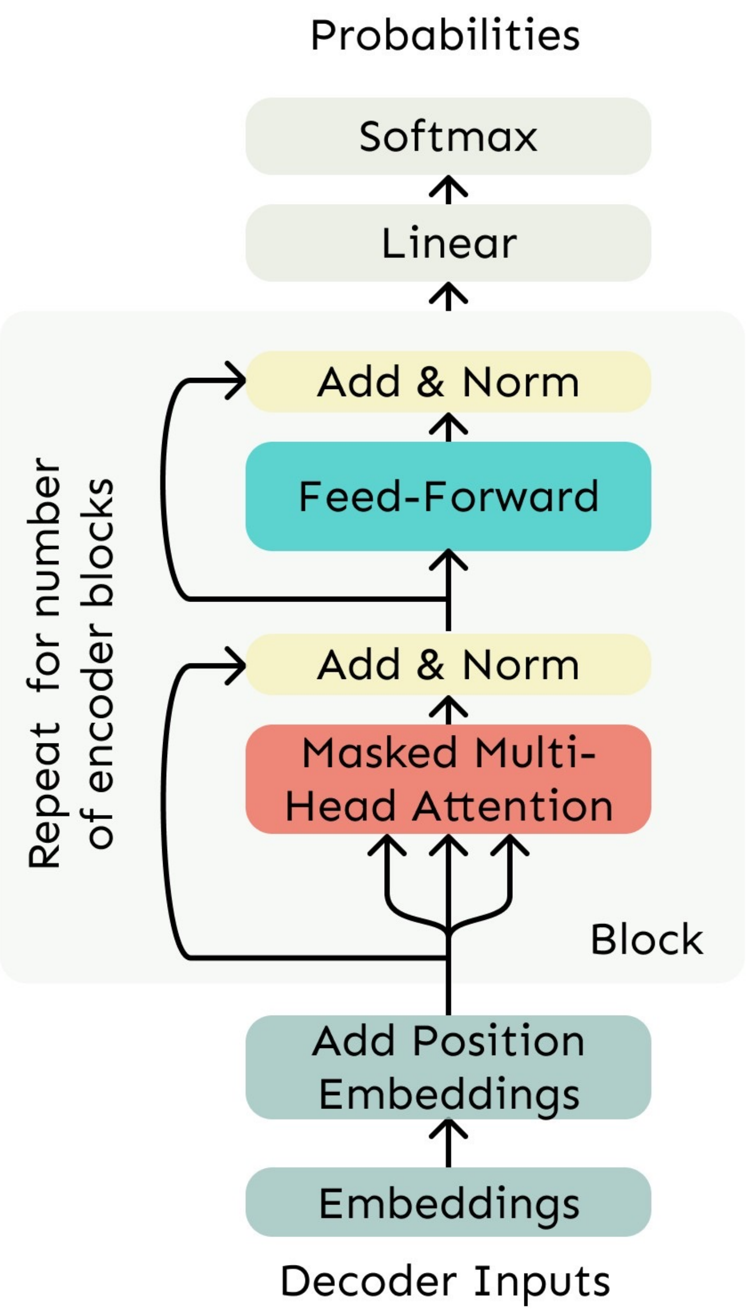
$$\text{output} = \frac{x - \mu}{\sqrt{\sigma} + \epsilon} * \gamma + \beta$$

Normalize by scalar mean and variance

Modulate by learned elementwise gain and bias

The Transformer Decoder

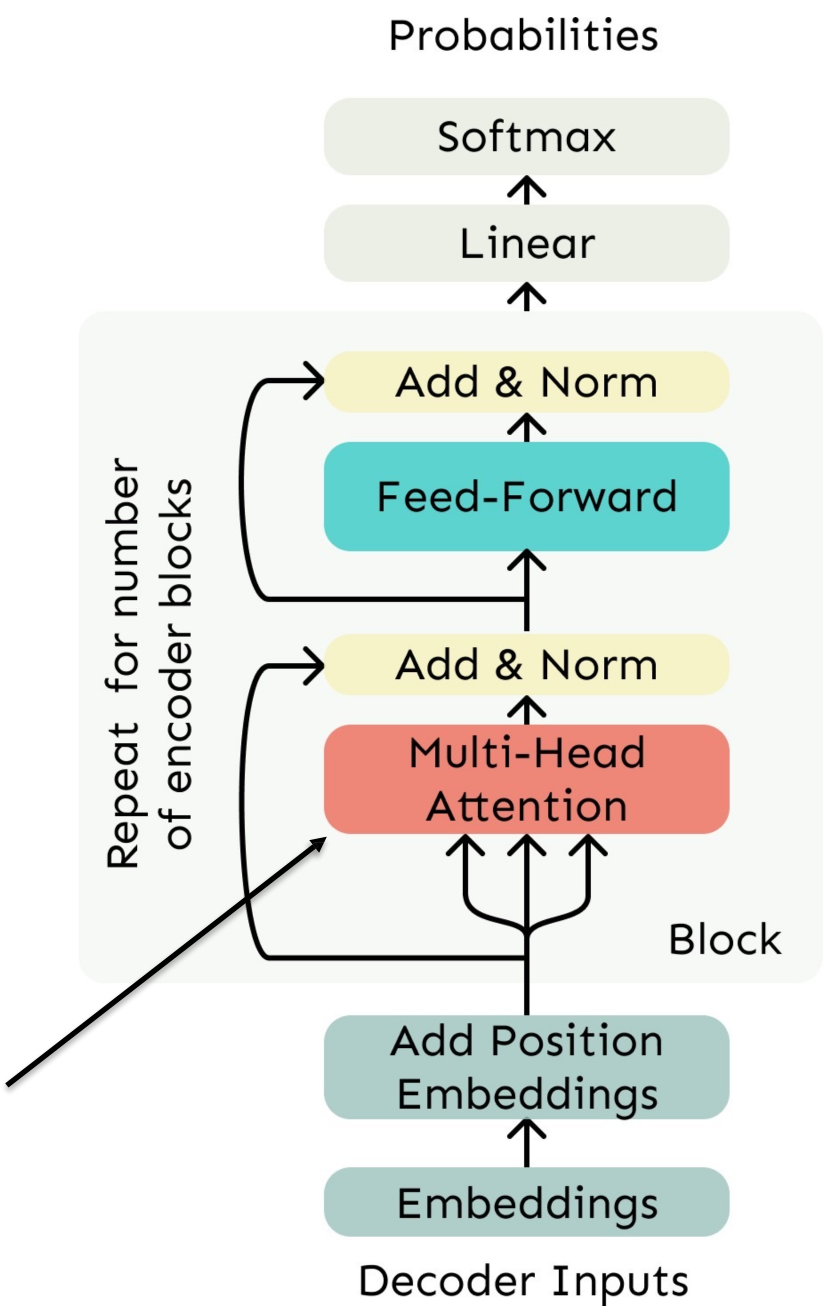
- The Transformer Decoder is a stack of Transformer Decoder **Blocks**.
- Each Block consists of:
  - Self-attention
  - Add & Norm
  - Feed-Forward
  - Add & Norm
- That's it! We've gone through the Transformer Decoder.



The Transformer Encoder

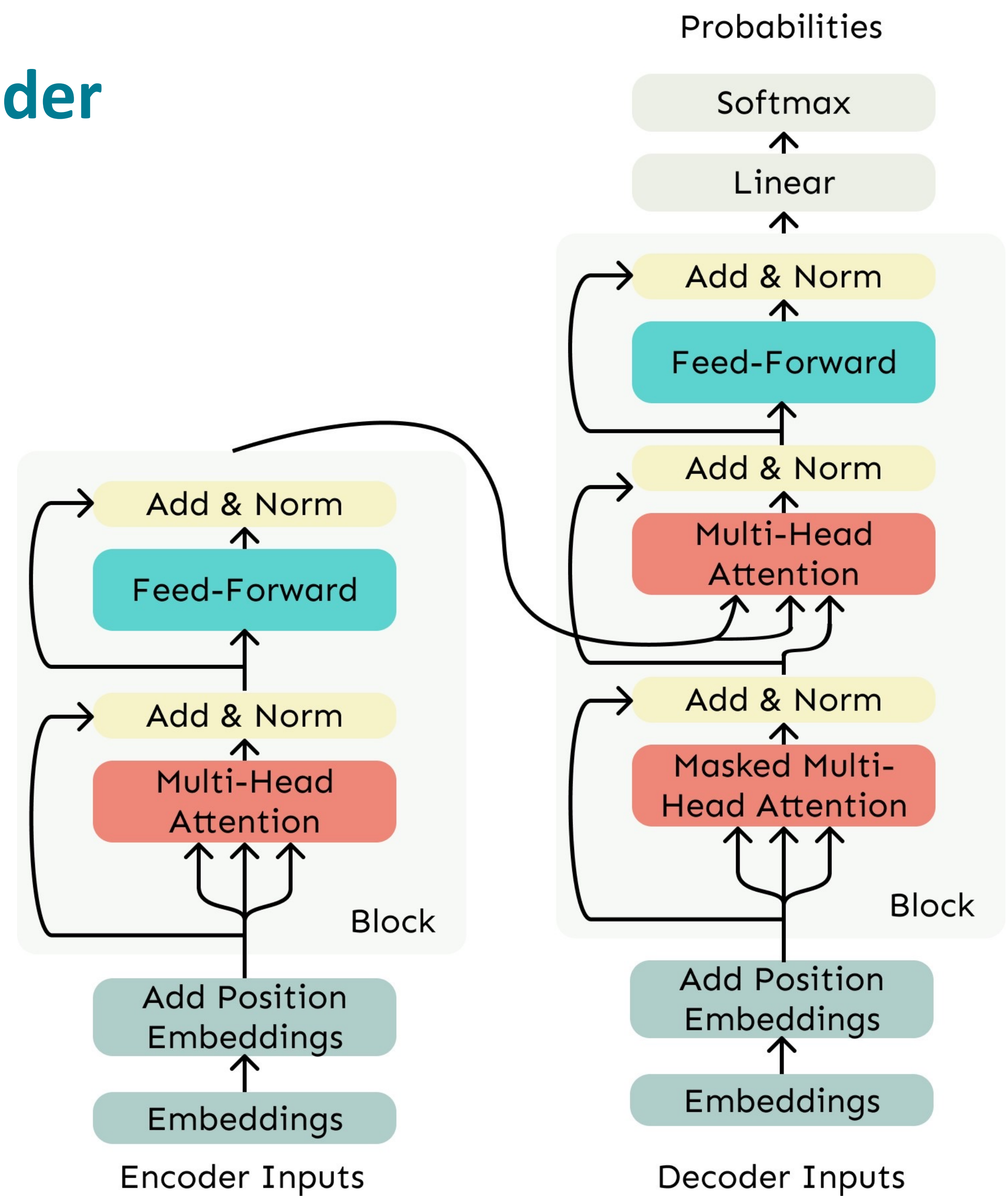
- The Transformer Decoder constrains to **unidirectional context**, as for **language models.**
- What if we want **bidirectional context**, like in a bidirectional RNN?
- This is the Transformer Encoder. The only difference is that we **remove the masking** in the self-attention.

No Masking!



The Transformer Encoder-Decoder

- Recall that in machine translation, we processed the source sentence with a **bidirectional** model and generated the target with a **unidirectional model**.
- For this kind of seq2seq format, we often use a Transformer Encoder-Decoder.
- We use a normal Transformer Encoder.
- Our Transformer Decoder is modified to perform **cross-attention** to the output of the Encoder.



Outline

1. From recurrence (RNN) to attention-based NLP models
2. Introducing the Transformer model
3. Great results with 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**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$

 [Test sets: WMT 2014 English-German and English-French]

[Vaswani et al., 2017]

Great Results with Transformers

Next, document generation!

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

The old standard

Transformers all the way down.

[Liu et al., 2018]; WikiSum dataset

Great Results with Transformers

Before too long, most Transformers results also included **pretraining**.

Transformers' parallelizability allows for efficient pretraining, and have made them the de-facto standard.

On this popular aggregate benchmark, for example:

GLUE

All top models are Transformer (and pretraining)-based.

	Rank	Name	Model	URL	Score
	1	DeBERTa Team - Microsoft	DeBERTa / TuringNLRv4		90.8
	2	HFL iFLYTEK	MacALBERT + DKM		90.7
+	3	Alibaba DAMO NLP	StructBERT + TAPT		90.6
+	4	PING-AN Omni-Sinitic	ALBERT + DAAF + NAS		90.6
	5	ERNIE Team - Baidu	ERNIE		90.4
	6	T5 Team - Google	T5		90.3

More results Thursday when we discuss pretraining.

[Liu et al., 2018]

Questions?