Natural Language Processing with Deep Learning

CS224N/Ling284

John Hewitt

Lecture 8: Self-Attention and Transformers

Adapted from slides by Anna Goldie, John Hewitt

Lecture Plan

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

Reminders:

Extra details are in the brand new lecture notes, **wooooo!**

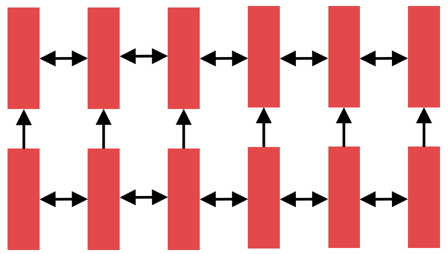
Assignment 4 due a week from today!

Final project proposal out tonight, due Tuesday, Feb 14 at 4:30PM PST!

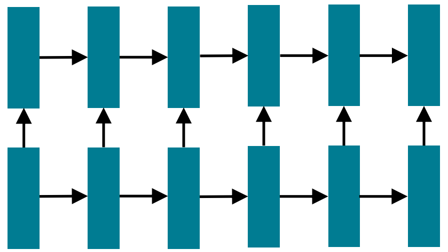
Please try to hand in the project proposal on time; we want to get you feedback quickly!

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

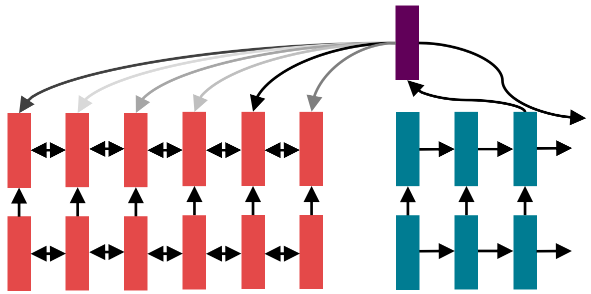
- Circa 2016, the de facto strategy in NLP is to **encode** sentences with a bidirectional LSTM: (for example, the source sentence in a translation)



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

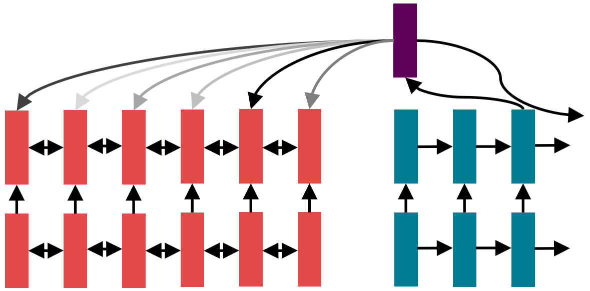


- Use attention to allow flexible access to memory



Today: Same goals, different building blocks

- Last week, 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.



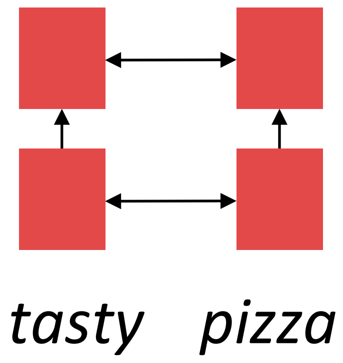
2014-2017ish
Recurrence

Lots of trial
and error

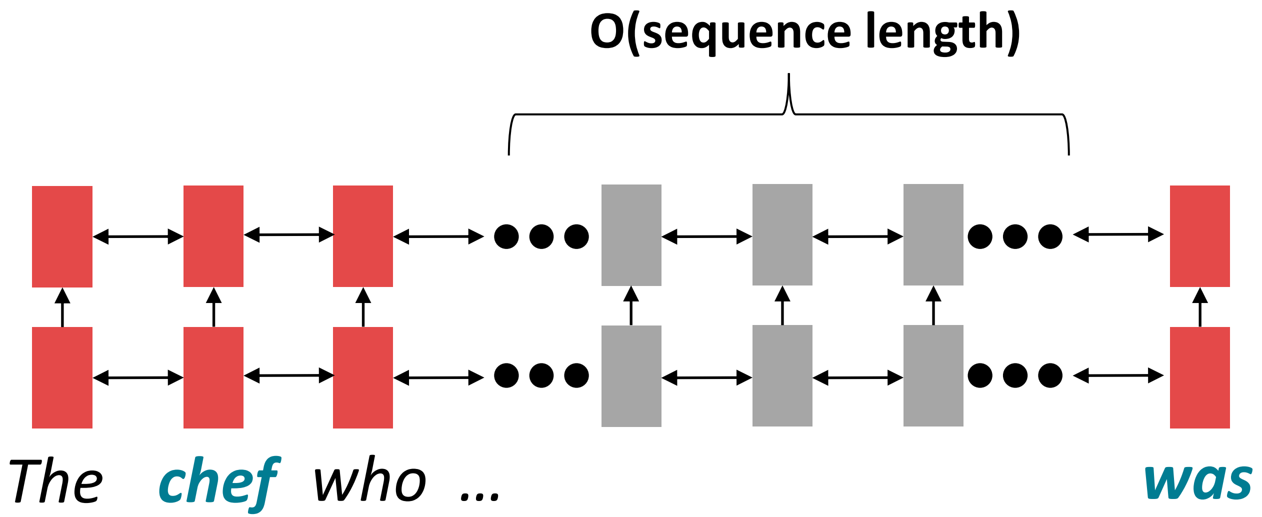
2021
??????

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

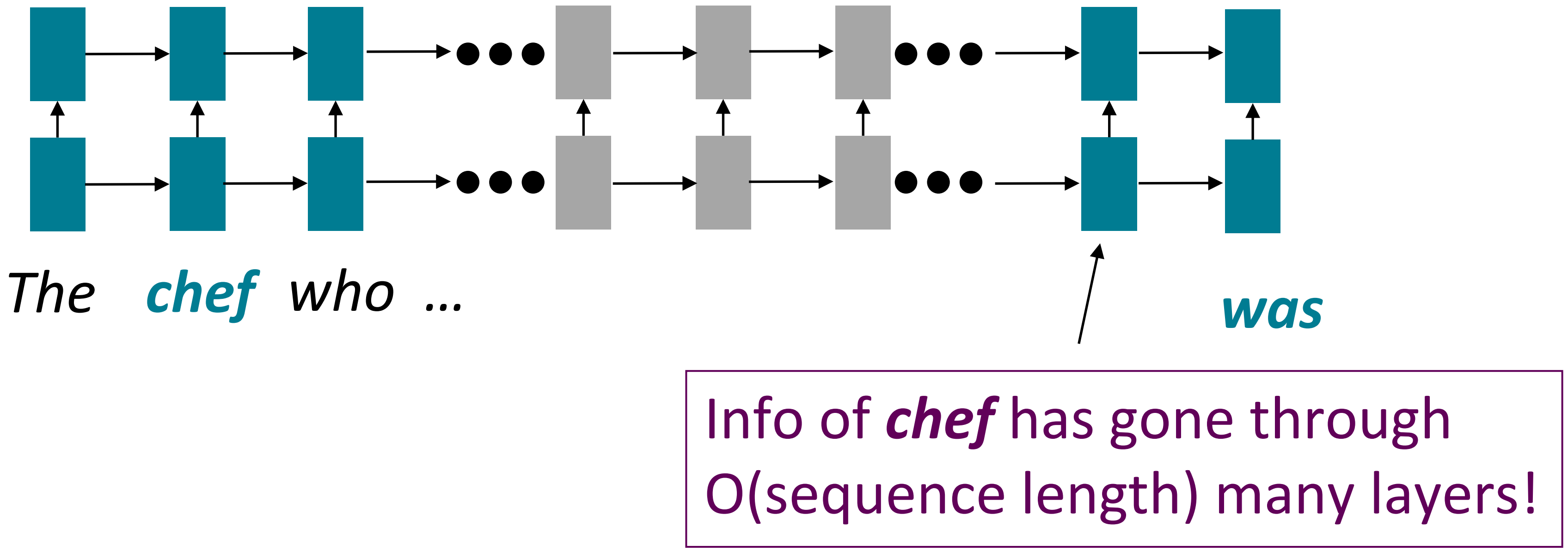


- **Problem:** RNNs take **O(sequence length)** steps for distant word pairs to interact.



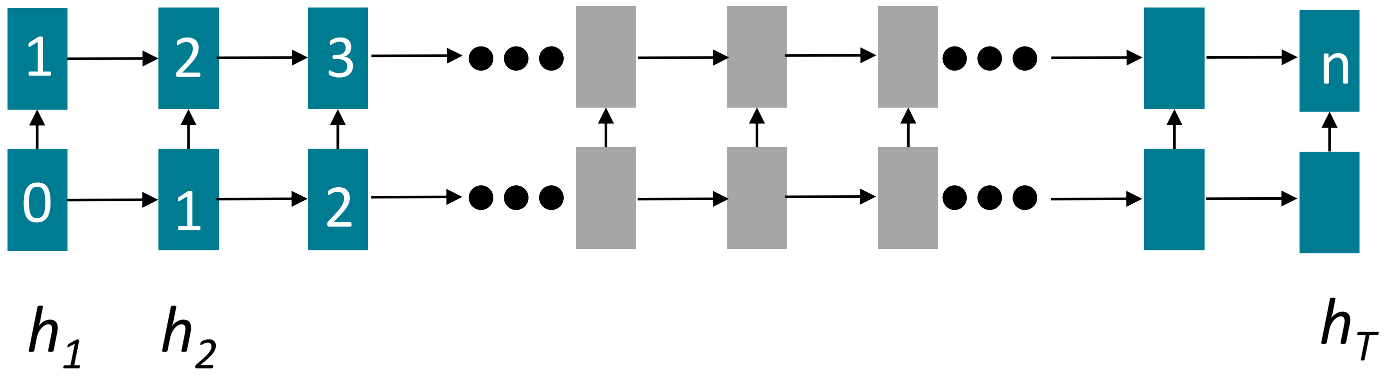
Issues with recurrent models: **Linear interaction distance**

- **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**

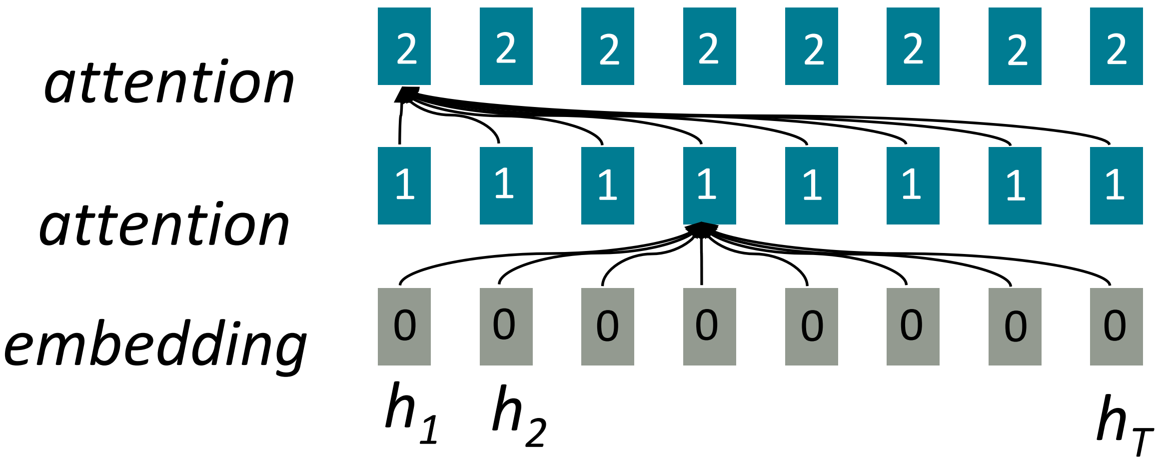
- Forward and backward passes have **O(sequence length)** unparallelizable operations
  - GPUs can perform a bunch of independent computations at once!
  - But future RNN hidden states can't be computed in full before past RNN hidden states have been computed
  - Inhibits training on very large datasets!



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!

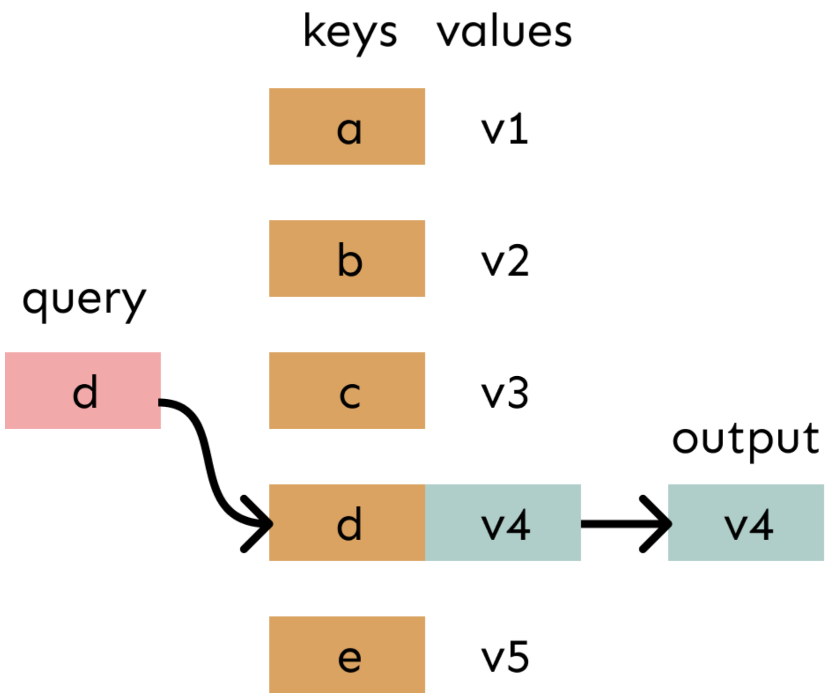


All words attend to all words in previous layer; most arrows here are omitted

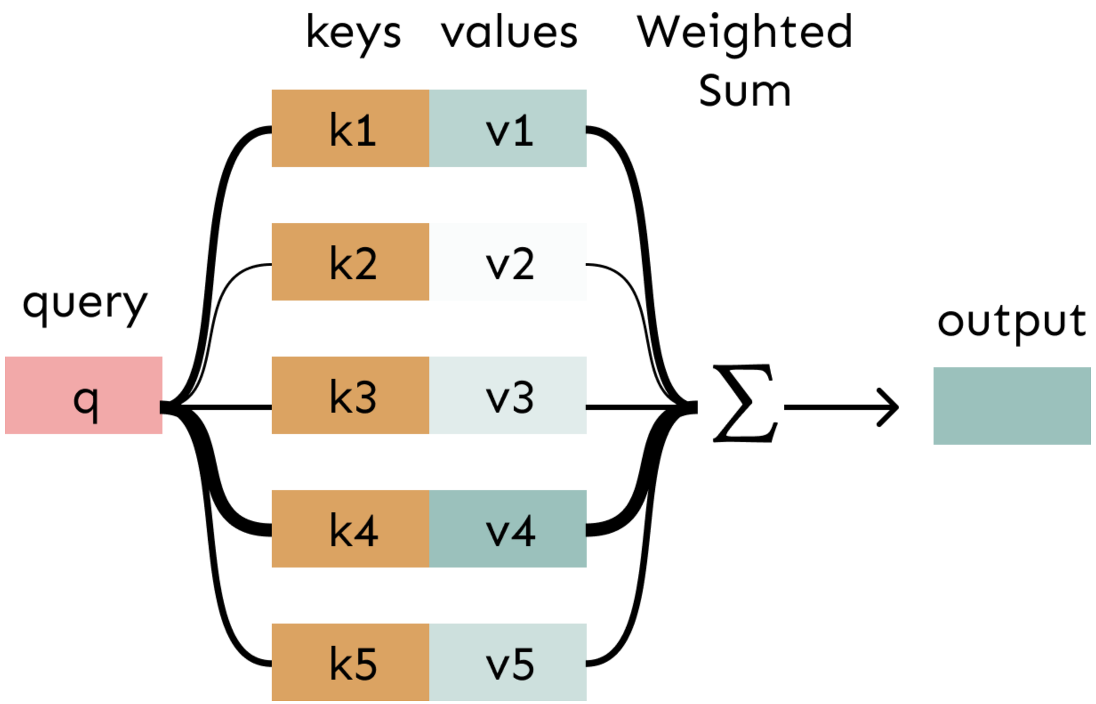
Attention as a soft, averaging lookup table

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

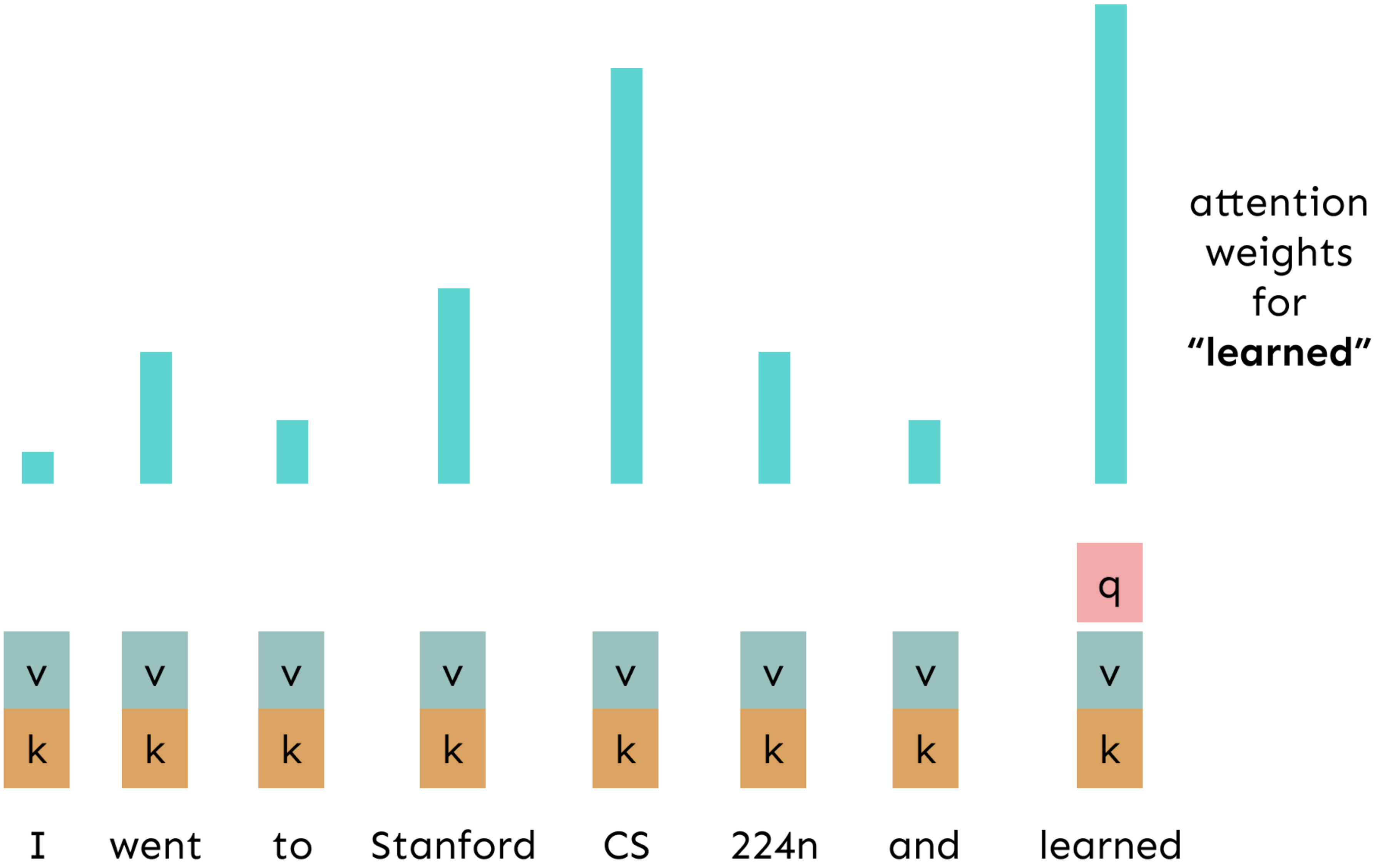
In a **lookup table**, we have a table of **keys** that map to **values**. The **query** matches one of the keys, returning its value.



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*.

wi: One Hot Encoding   encodinng with dimension d in each column

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

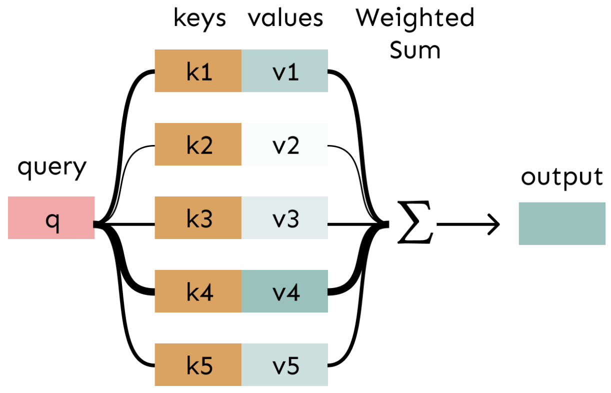
$$\boldsymbol{q}_i = Q\boldsymbol{x}_{\boldsymbol{i}} \text{ (queries)} \qquad \boldsymbol{k}_i = K\boldsymbol{x}_{\boldsymbol{i}} \text{ (keys)} \qquad \boldsymbol{v}_i = V\boldsymbol{x}_{\boldsymbol{i}} \text{ (values)}$$

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**

$$\boldsymbol{p}_i \in \mathbb{R}^d, \text{ for } i \in \{1,2,\dots,n\} \text{ are position vectors}$$

- 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 $\boldsymbol{p}_i$ to our inputs!
- Recall that $\boldsymbol{x}_i$ is the embedding of the word at index i. The positioned embedding is:

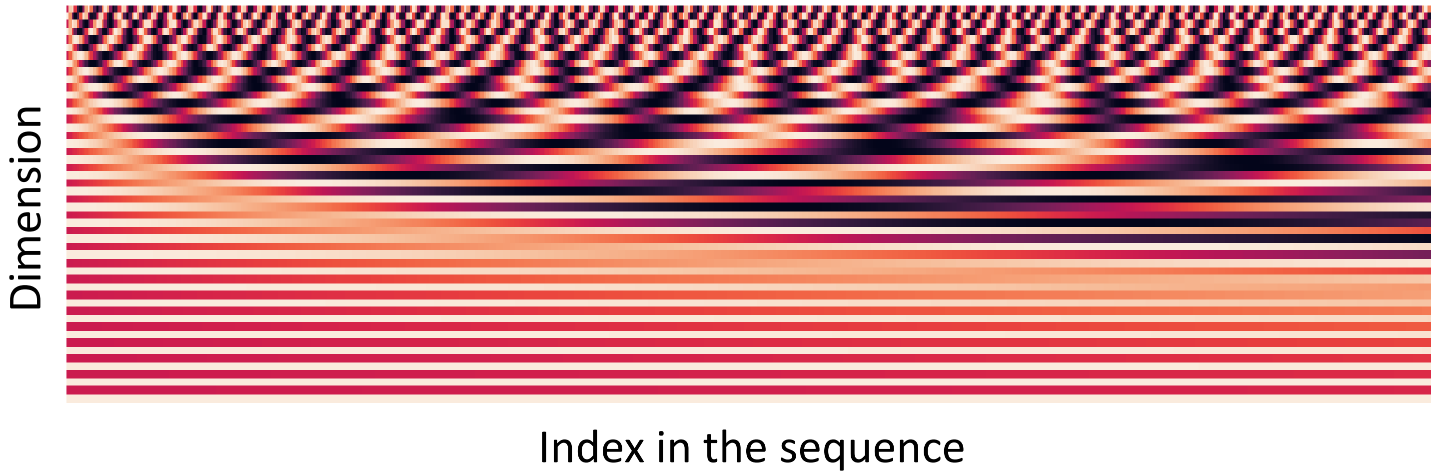
$$\tilde{\boldsymbol{x}}_i = \boldsymbol{x}_i + \boldsymbol{p}_i$$

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 $1, \ldots, n$.
- 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

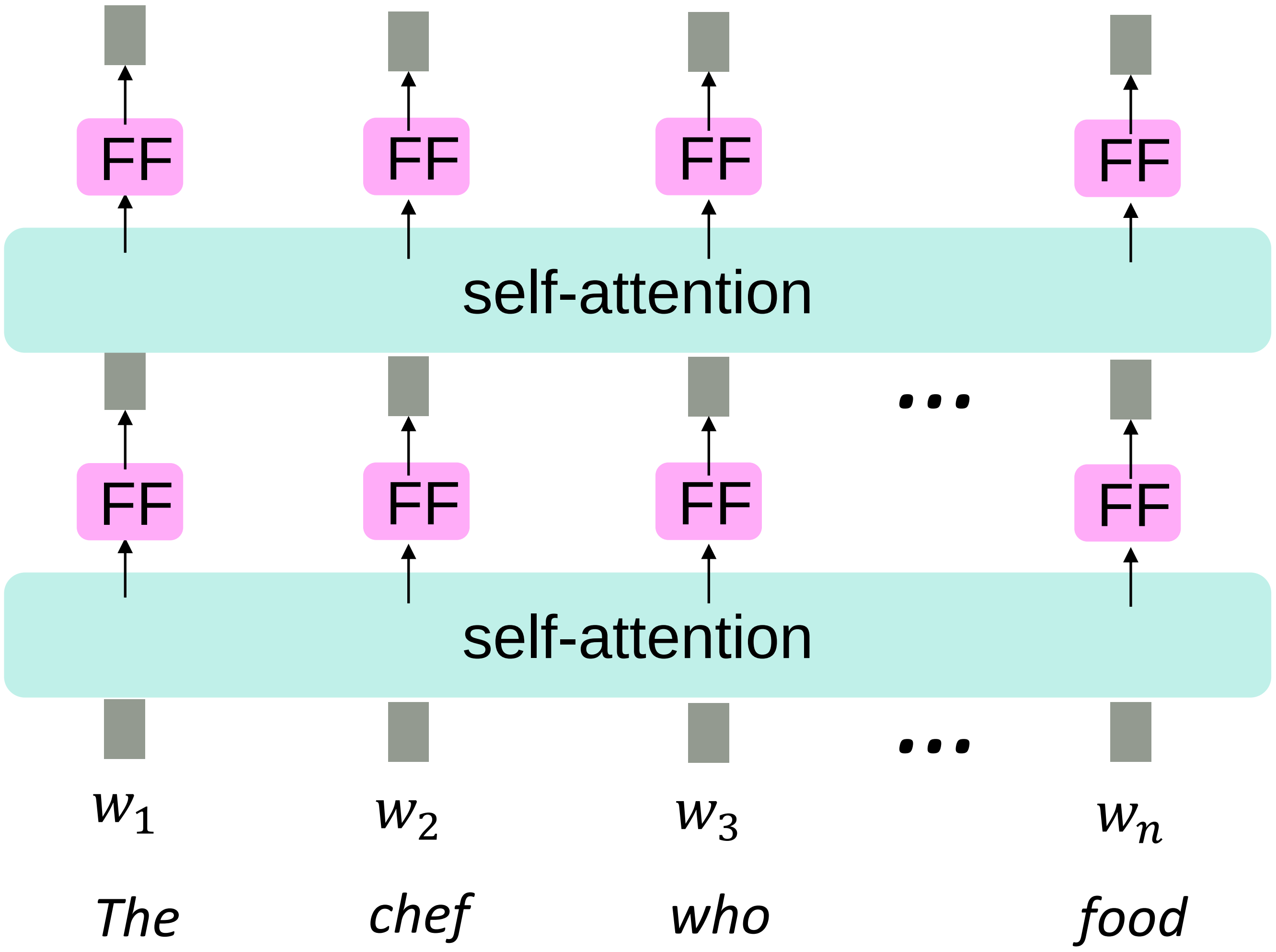
Adding nonlinearities in self-attention

- Note that there are no elementwise nonlinearities in self-attention; stacking more self-attention layers just re-averages **value** vectors (Why? Look at the notes!)

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

Multi Layer Perceptron

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



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

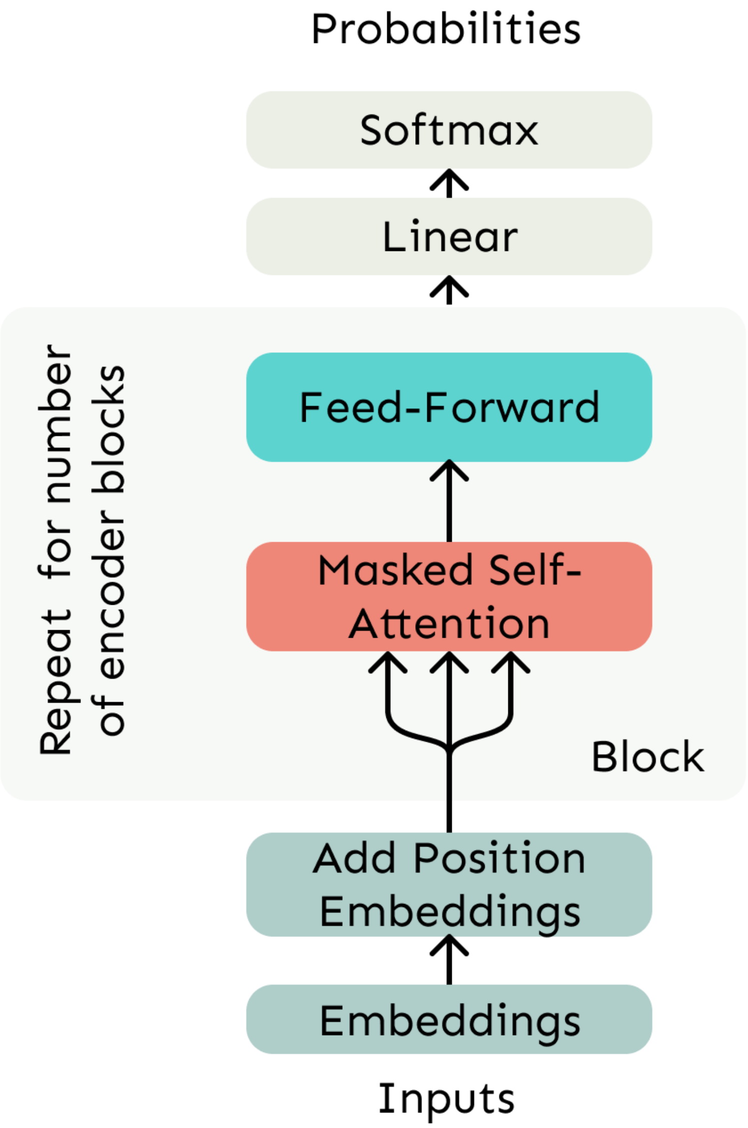
	[START]	The	chef	who
[START]		$-\infty$	$-\infty$	$-\infty$
The			$-\infty$	$-\infty$
chef				$-\infty$
who				

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.

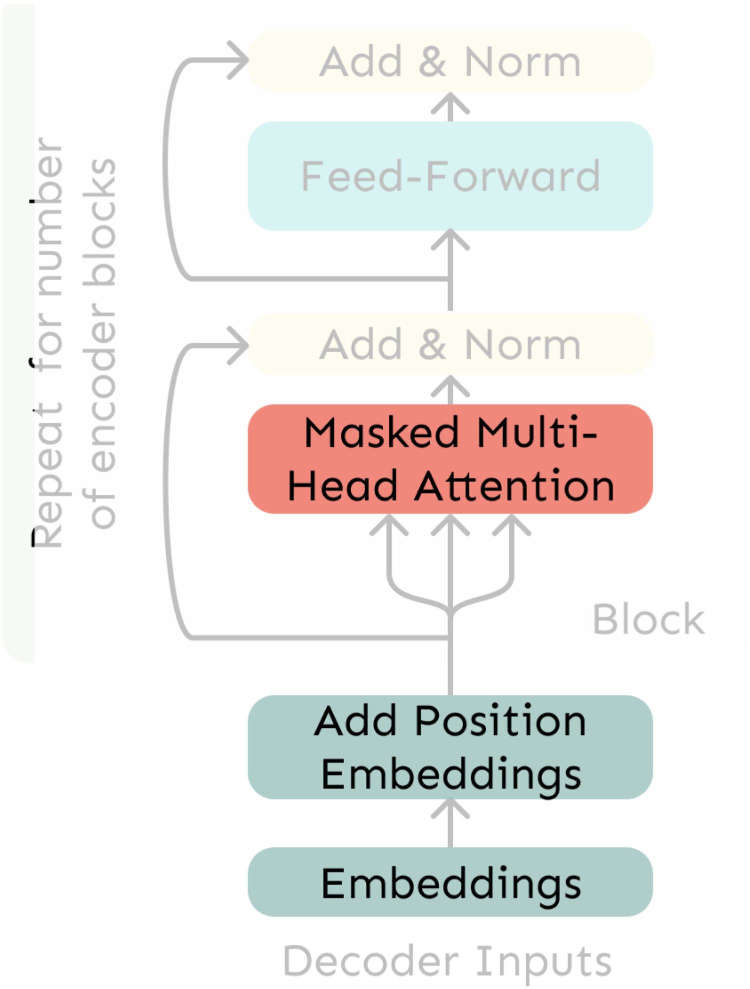


Outline

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

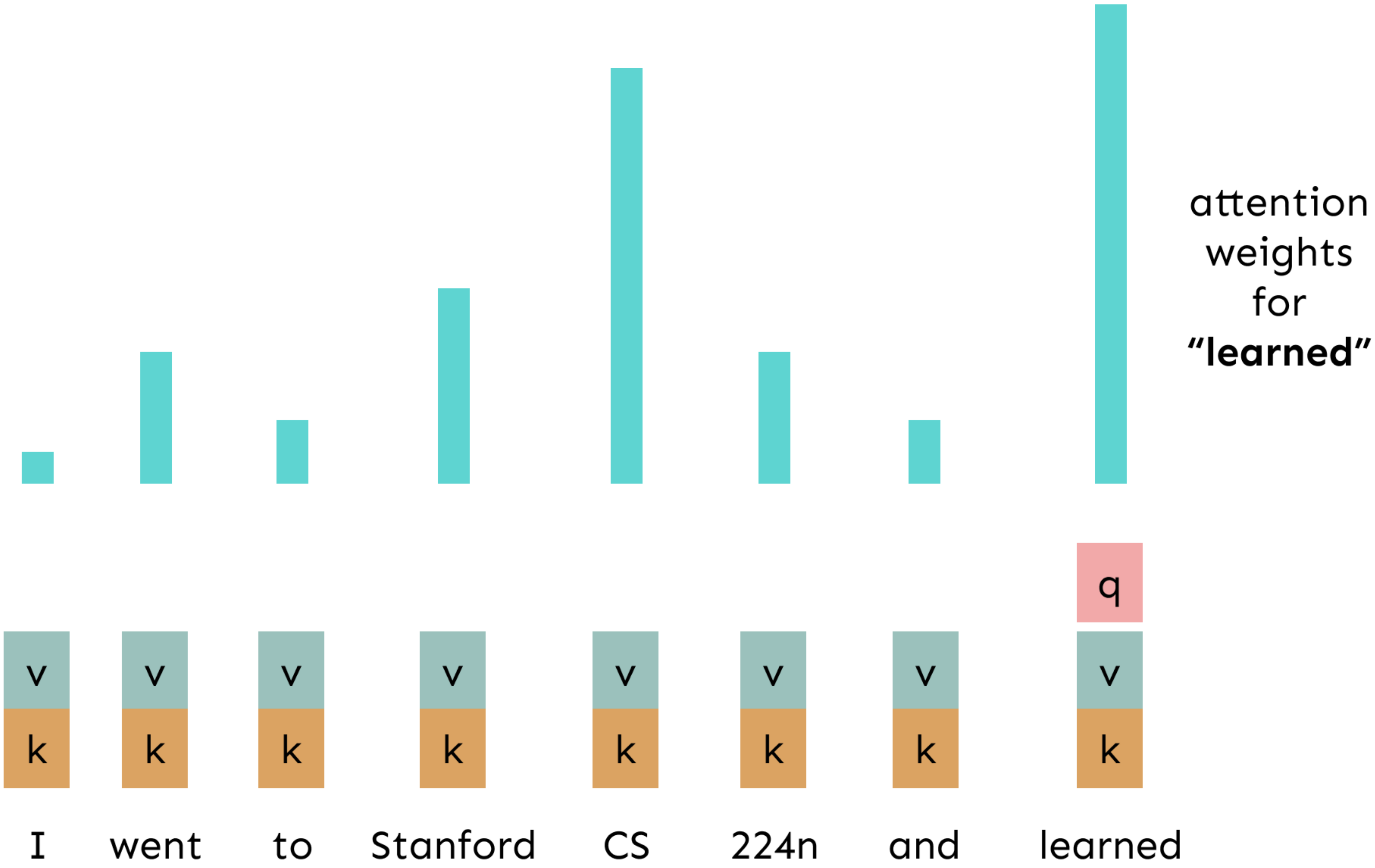
The Transformer Decoder

- 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.**

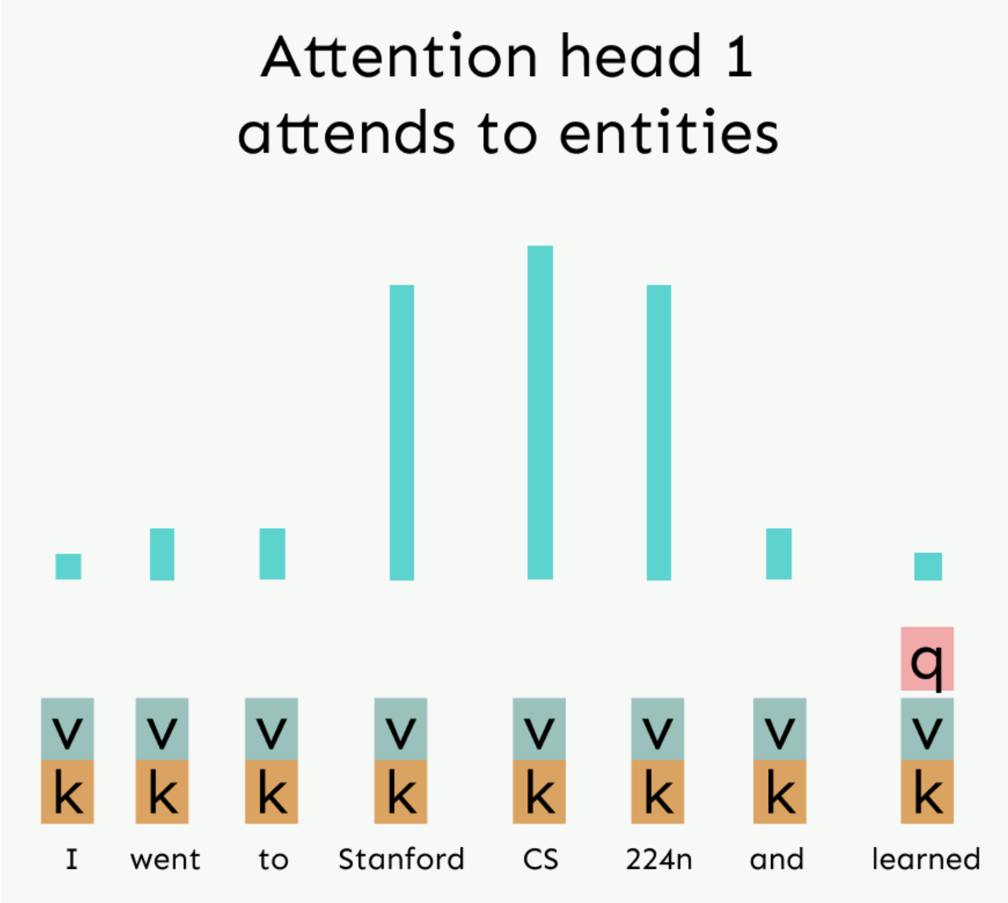


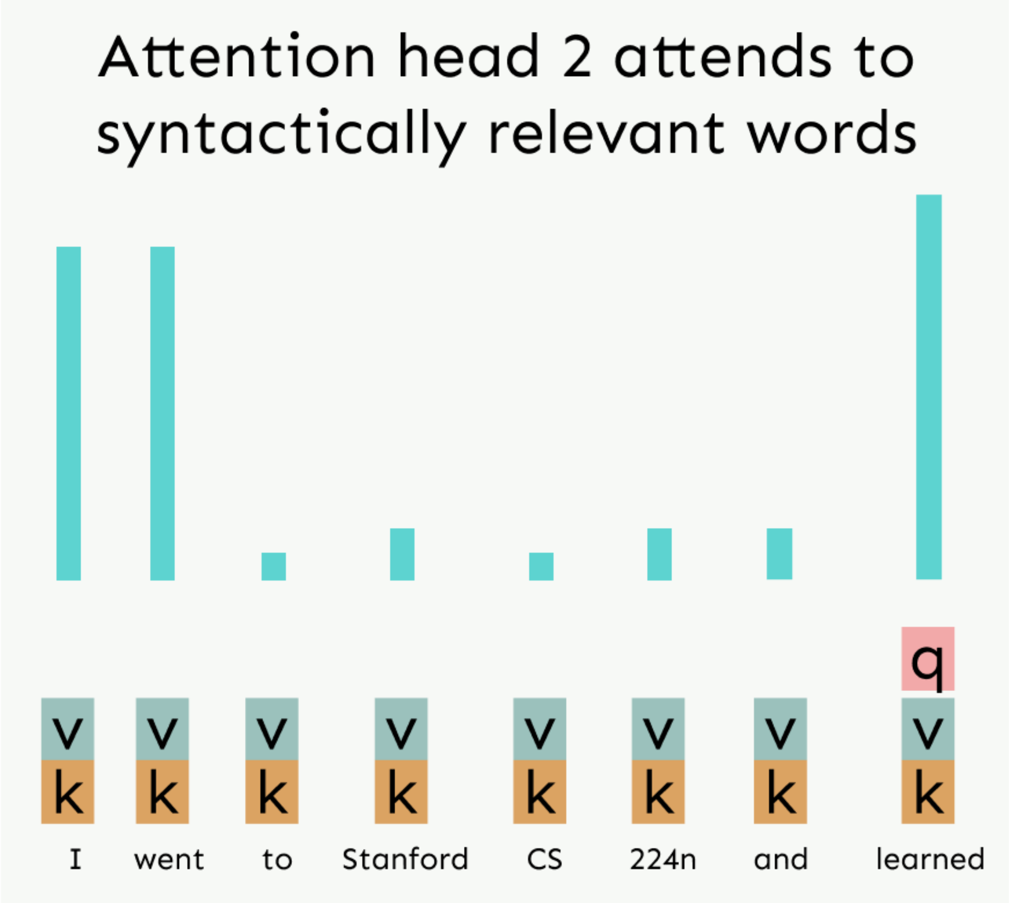
Transformer Decoder

Recall the Self-Attention Hypothetical Example



Hypothetical Example of Multi-Head Attention





I went to Stanford CS 224n and learned

Sequence-Stacked form of Attention

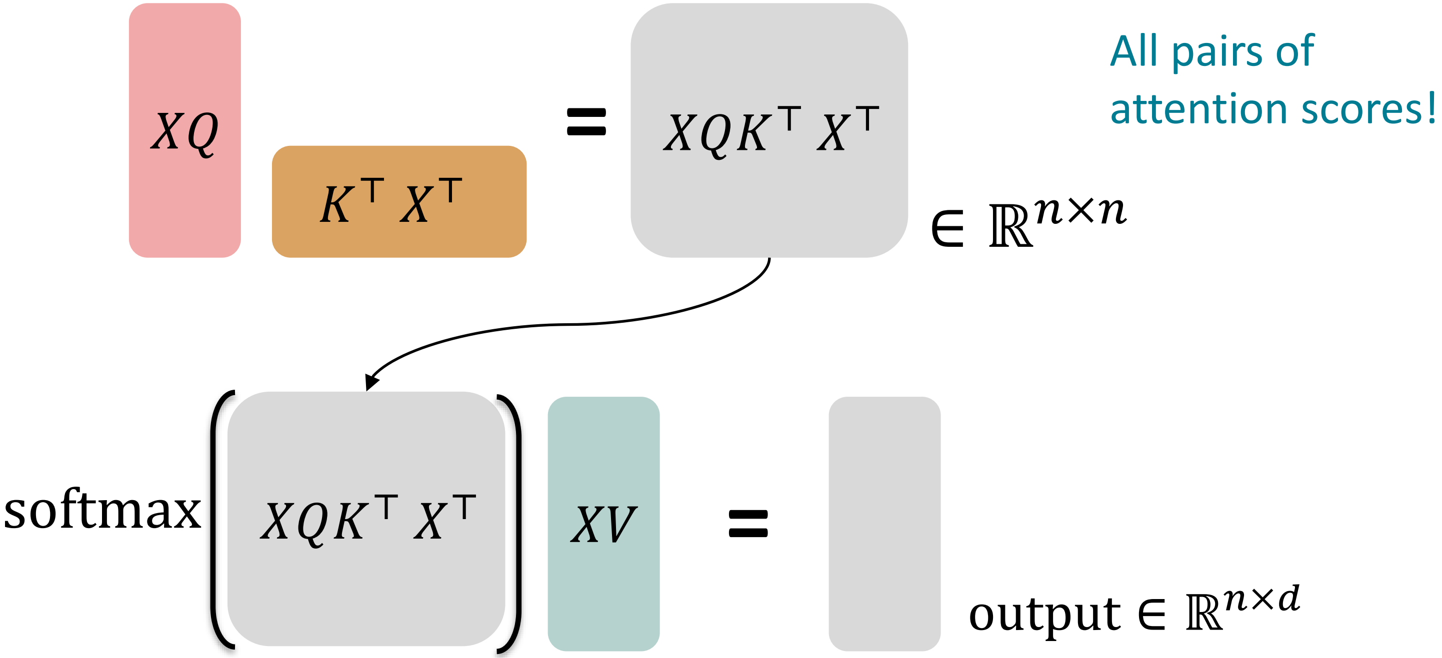
- Let's look at how key-query-value attention is computed, in matrices.
  - Let $X = [x_1; \dots; x_n] \in \mathbb{R}^{n \times d}$ be the concatenation of input vectors. Each row contain input encdings
  - First, note that $XK \in \mathbb{R}^{n \times d}$, $XQ \in \mathbb{R}^{n \times d}$, $XV \in \mathbb{R}^{n \times d}$. XK because row times column

    d*d
  - The output is defined as output $=$ $\mathrm{softmax}(XQ(XK)^\top)XV \in\in \mathbb{R}^{n \times d}$.

(XK)T = KTXT

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:
  - $\text{output}_\ell = \text{softmax}\left(XQ_\ell K_\ell^\top X^\top\right) * XV_\ell$, where $\text{output}_\ell \in \mathbb{R}^{d/h}$

n*d/h

- Then the outputs of all the heads are combined!
  - $\text{output} = [\text{output}_1; \dots; \text{output}_h]Y$, where $Y \in \mathbb{R}^{d \times d}$

n* d

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

Multi-head self-attention is computationally efficient

can have different no of attention heads in different blocks but no reason to have it

Each Attention blocks have different weights

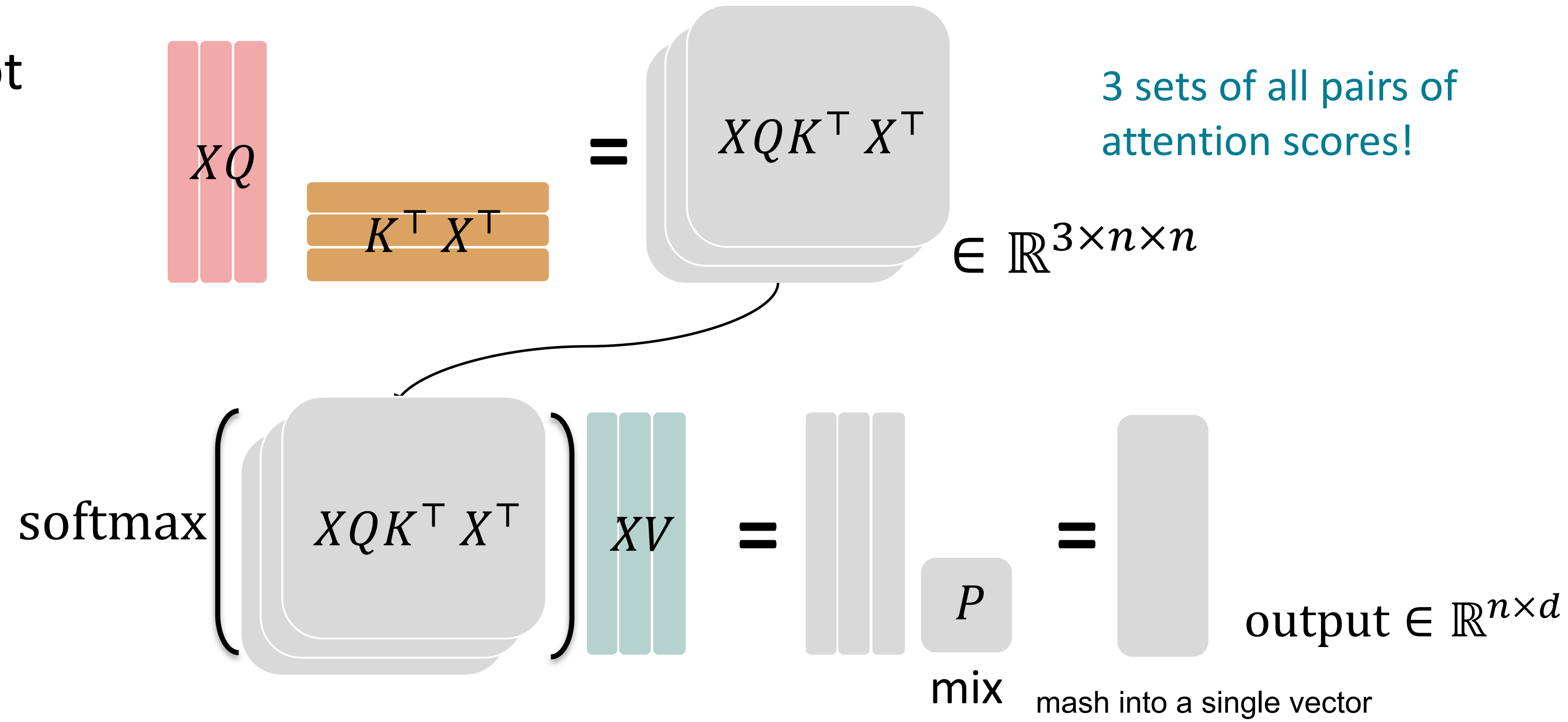
- Even though we compute h many attention heads, it's not really more costly.
  - We compute $XQ \in \mathbb{R}^{n \times d}$, and then reshape to $\mathbb{R}^{n \times h \times d/h}$. (Likewise for XK, XV.)

    3 dimensional vector
  - Then we transpose to $\mathbb{R}^{h \times n \times d/h}$; now the head axis is like a batch axis.

    Transpose in 3 dimension complex, can specify dimensions directly in pytorch
  - 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.



Scaled Dot Product [Vaswani et al., 2017]

- **"Scaled Dot Product"** attention aids in 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:

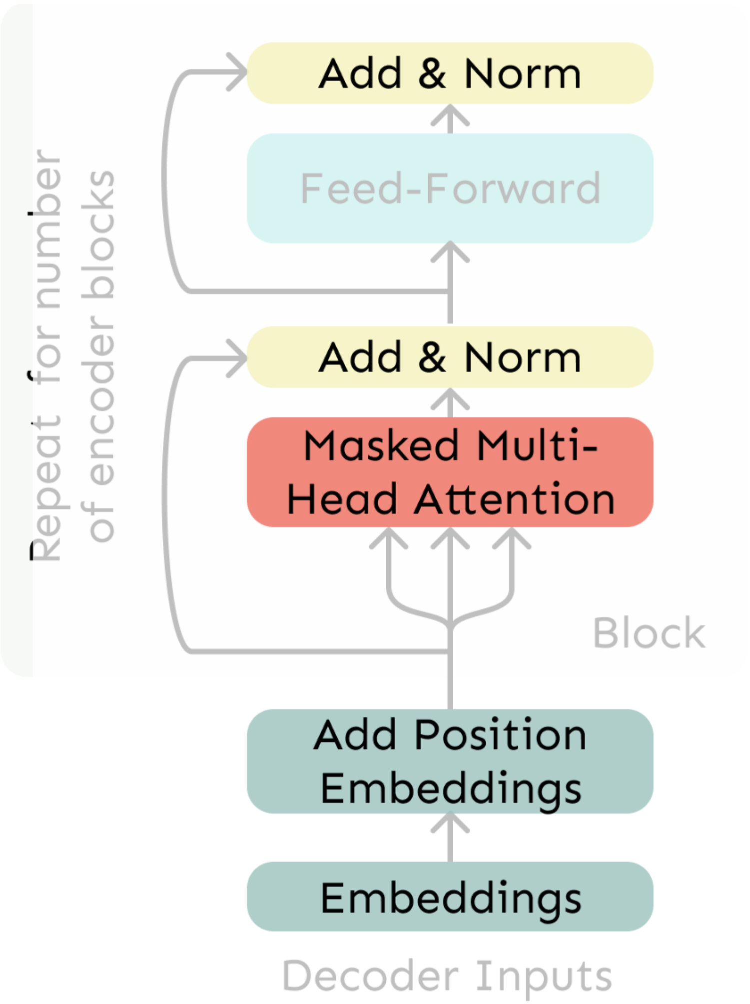
$$\text{output}_\ell = \text{softmax}\left(XQ_\ell K_\ell^\top X^\top\right) * XV_\ell$$

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

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

The Transformer 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"

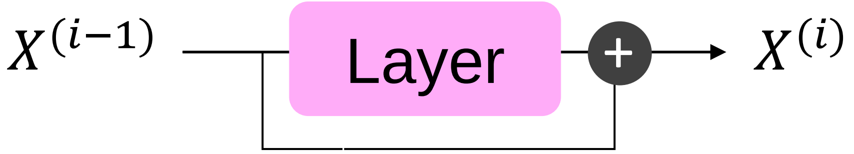


Transformer Decoder

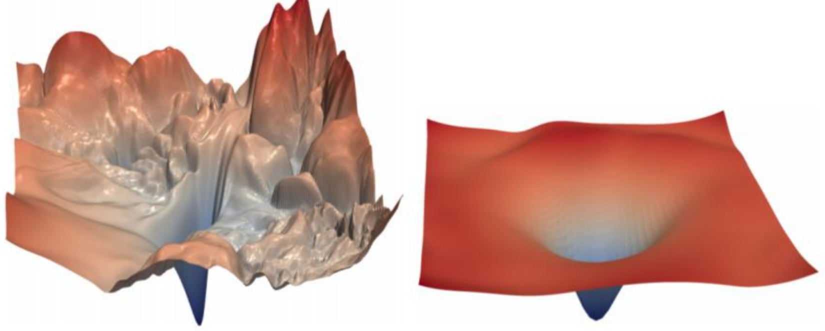
The Transformer Encoder: **Residual connections** [He et al., 2016]

- **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)



too manu local maxima and minima



if vanishing gradient through this layer we can learn at least the gradients from previous layers

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

residual connection is idetity, so goes towards identity

[no residuals] [residuals]

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

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

In the context of transformers, layer normalization plays a crucial role in stabilizing the outputs of various layers (e.g., attention layers, feed-forward neural networks within the transformer blocks) by ensuring that the distribution of activations remains consistent across different layers and training steps.
This helps in speeding up training and improving the overall performance of the model.

- **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.

prevents outputs from getting too big or small

- Let $\mu = \sum_{j=1}^{d} x_j$; this is the mean; $\mu \in \mathbb{R}$.

mean for single word

- 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:

gama and beta are learnable parameters

See Batch Normalization: Normalizes entire batch

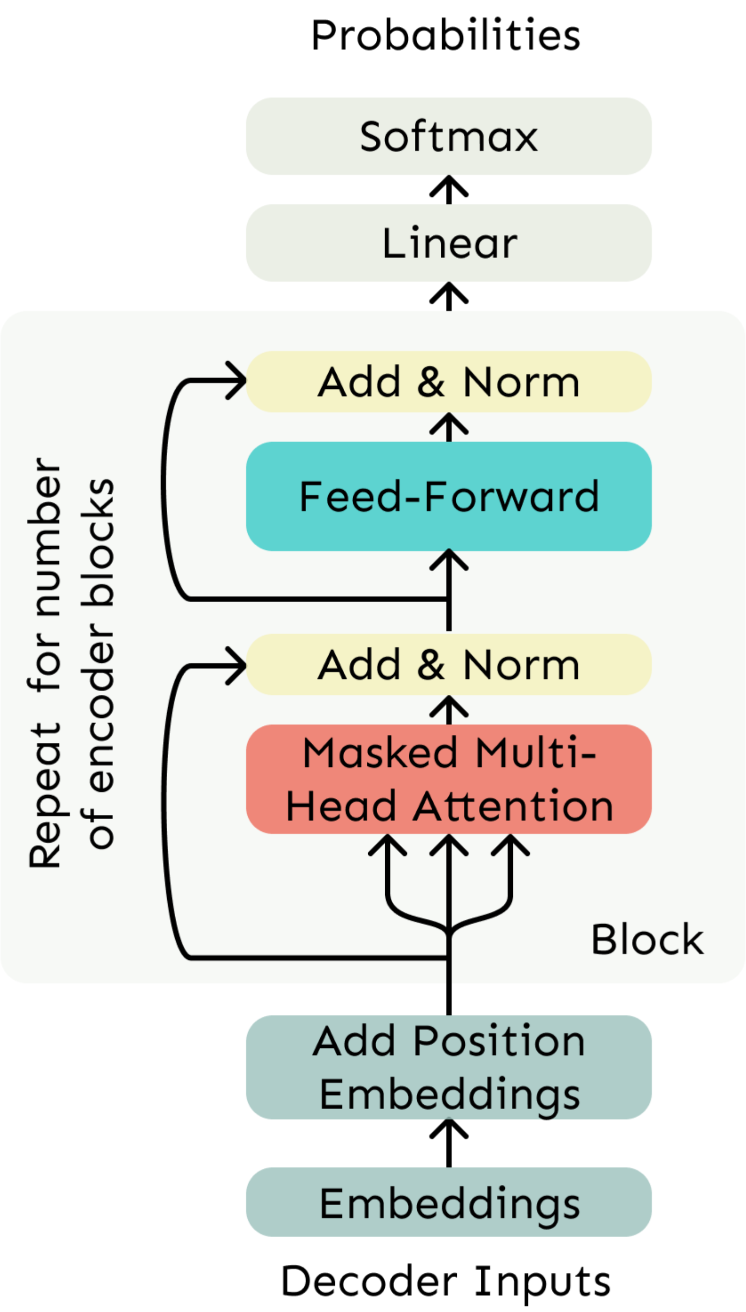
$$\text{output} = \frac{x - \mu}{\sqrt{\sigma} + \epsilon} * \gamma + \beta$$

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.

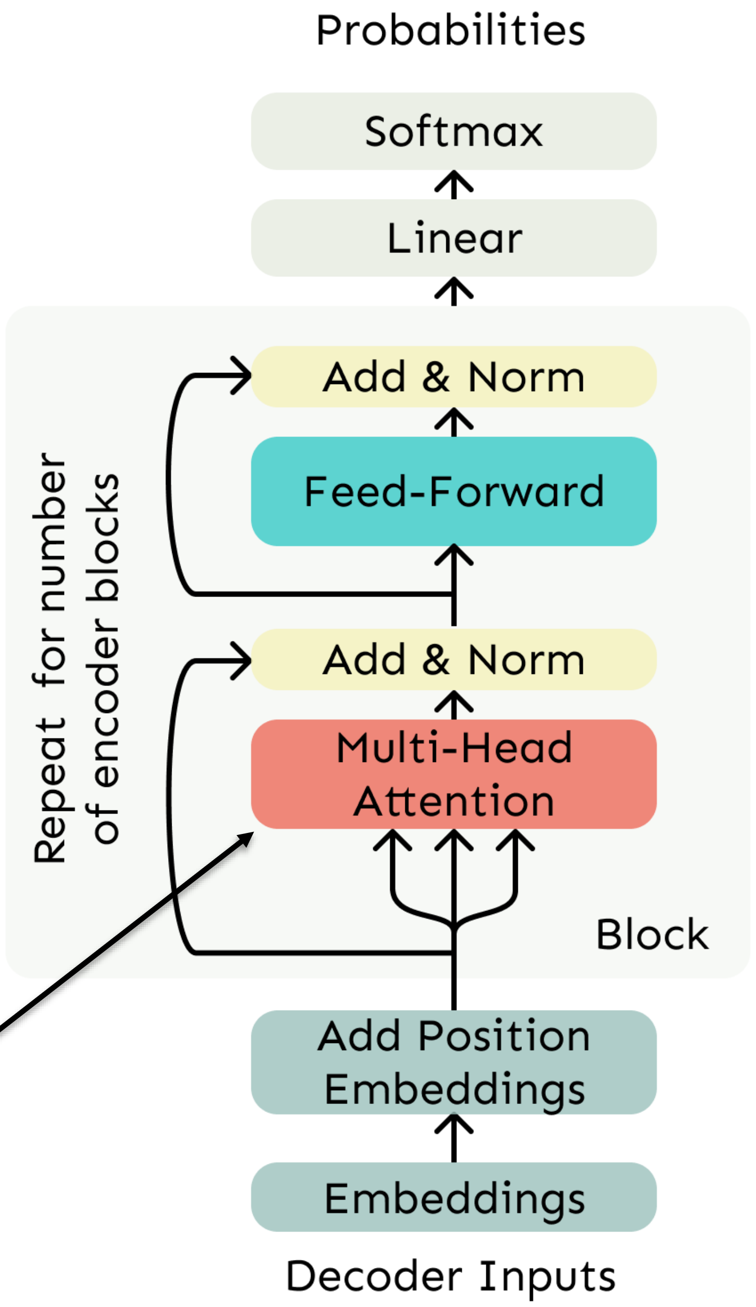


Variable length handled through padding.
padding self attention also -infinity

The Transformer Encoder

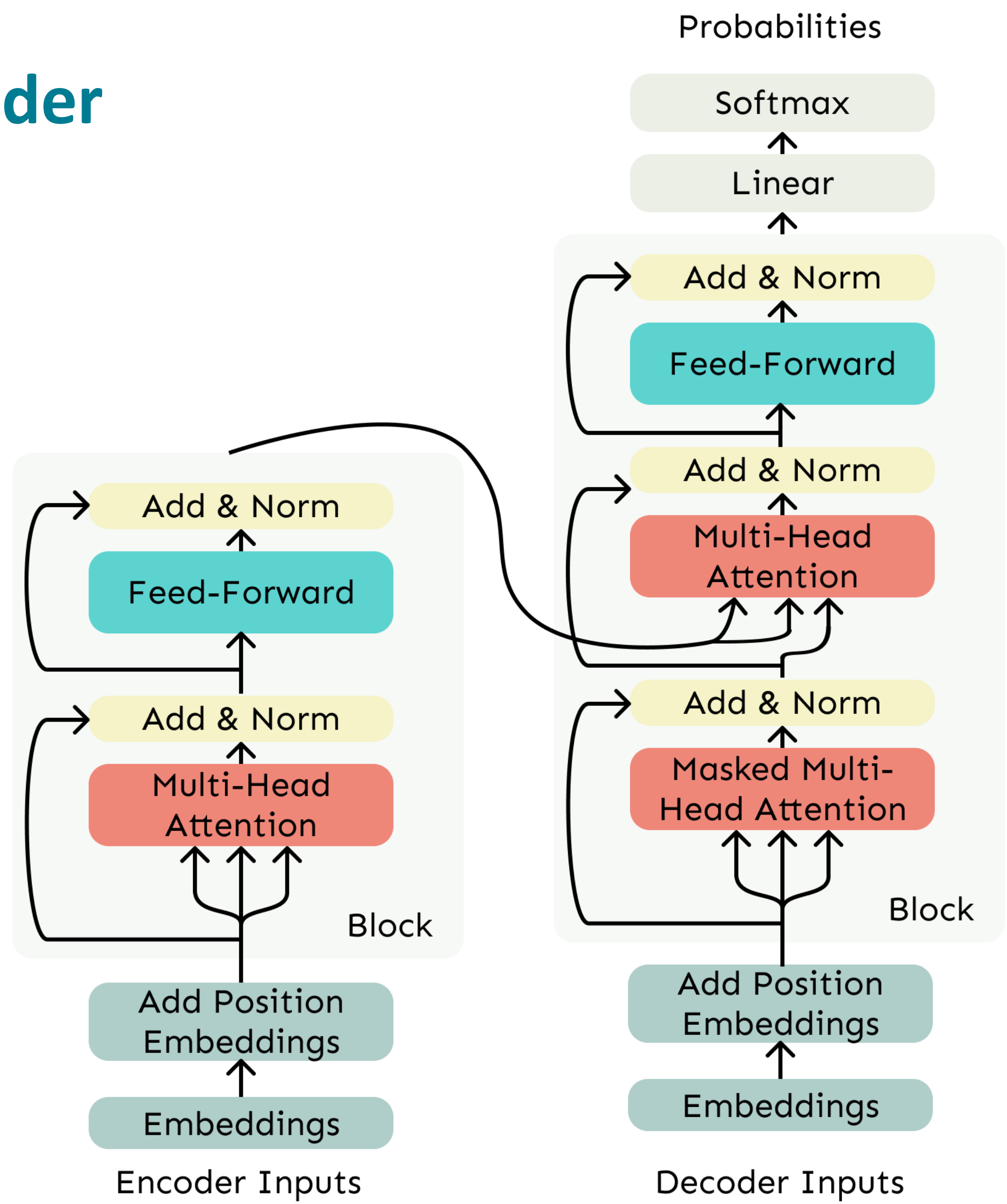
- The Transformer Decoder constrains to **unidirectional context**, as for **language models.**
- What if we want **bidirectional context**, like in a bidirectional RNN?
- This is the Transformer Encoder. The only difference is that we **remove the masking** in the self-attention.

No Masking!



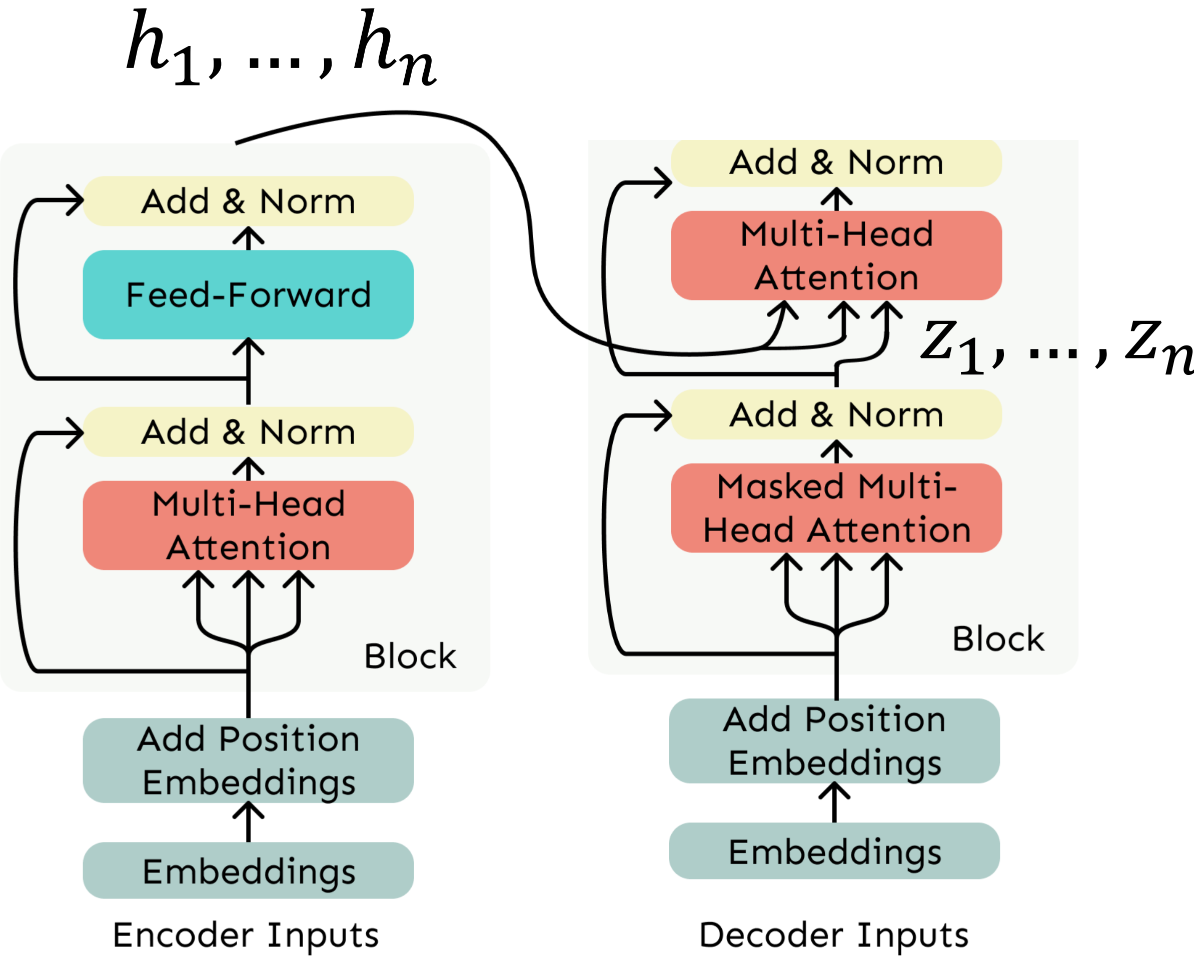
The Transformer Encoder-Decoder

- 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.



Cross-attention (details)

- We saw that self-attention is when keys, queries, and values come from the same source.
- In the decoder, we have attention that looks more like what we saw last week.
- Let $h_1, \ldots, h_n$ be **output** vectors **from** the Transformer **encoder**; $x_i \in \mathbb{R}^d$
- Let $z_1, \ldots, z_n$ 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):
  - $k_i = K h_i, v_i = V h_i$.
- And the queries are drawn from the **decoder**, $q_i = Q z_i$.



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**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer(base model)	27.3	38.1	3.3*10^18	
Transformer(big)	28.4	41.8	2.2*10^19	

Not just better BLEU scored but also more efficient to train

 [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**, a method we'll go over on Thursday.

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]

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?

- **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

- One of the benefits of self-attention over recurrence was that it's highly parallelizable.
- However, its total number of operations grows as $O(n^2 d)$, where n is the sequence length, and d is the dimensionality.

$$XQ \; K^\top X^\top = XQK^\top X^\top \in \mathbb{R}^{n\times n}$$

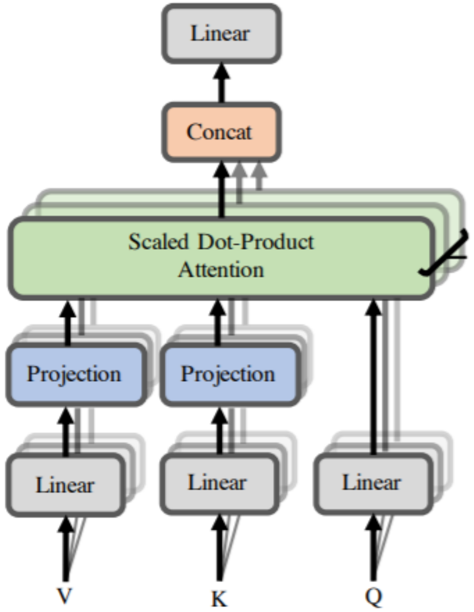
Need to compute all pairs of interactions! $O(n^2 d)$

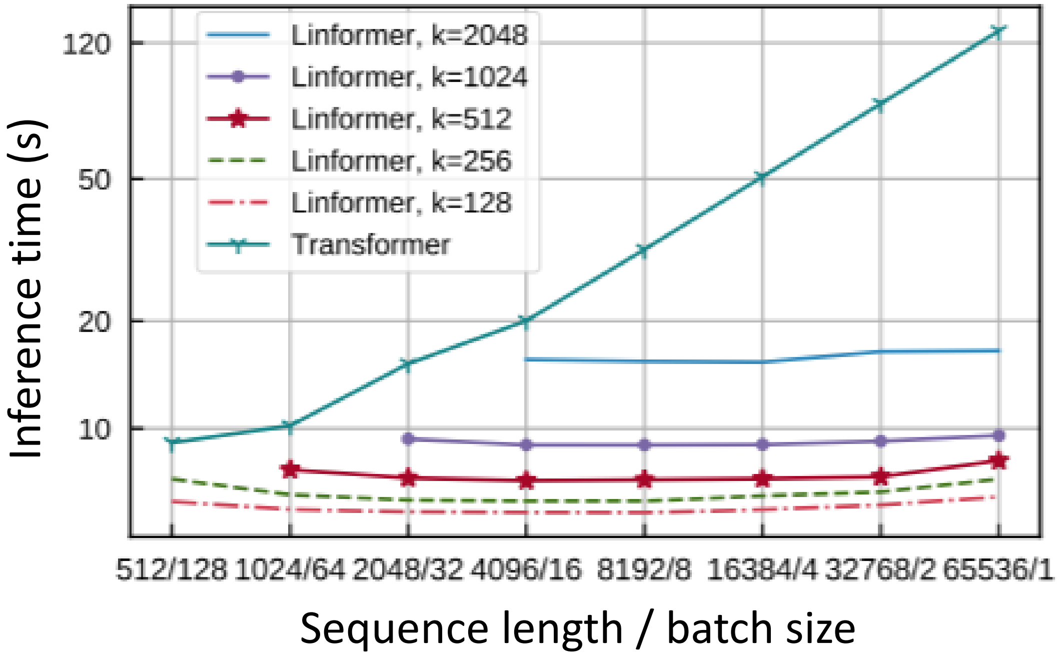
- Think of d as around $\mathbf{1,000}$ (though for large language models it's much larger!).
  - So, for a single (shortish) sentence, $n \le 30; n^2 \le \mathbf{900.}$
  - In practice, we set a bound like $n = 512.$
  - **But what if we'd like $\boldsymbol{n} \ge \mathbf{50,000}$?** For example, to work on long documents?

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





Do we even need to remove the quadratic cost of attention?

- As Transformers grow larger, a larger and larger percent of compute is **outside** the self-attention portion, despit the quadratic cost.
- In practice, **almost no large Transformer language models use anything but the quadratic cost attention we've presented here.**
  - The cheaper methods tend not to work as well at scale.
- So, is there no point in trying to design cheaper alternatives to self-attention?
- Or would we unlock much better models with much longer contexts (>100k tokens?) if we were to do it right?

Do Transformer Modifications Transfer?

- "Surprisingly, we find that most modifications do not meaningfully improve performance."

Model	Params	Ops	Step/s	Early loss	Final loss	SGLUE	XSum	WebQ	WMT EnDe
Vanilla Transformer	223*M*	11.1*T*	3.50	2.182 ± 0.005	1.838	71.66	17.78	23.02	26.62
GeLU	223*M*	11.1*T*	3.58	2.179 ± 0.003	1.838	**75.79**	**17.86**	**25.13**	26.47
Swish	223*M*	11.1*T*	3.62	2.186 ± 0.003	1.847	**73.77**	17.74	**24.34**	**26.75**
ELU	223*M*	11.1*T*	3.56	2.270 ± 0.007	1.932	67.83	16.73	23.02	26.08
GLU	223*M*	11.1*T*	3.59	2.174 ± 0.003	**1.814**	**74.20**	**17.42**	24.34	**27.12**
GeGLU	223*M*	11.1*T*	3.55	2.130 ± 0.006	**1.792**	**75.96**	**18.27**	**24.87**	**26.87**
ReGLU	223*M*	11.1*T*	3.57	2.145 ± 0.004	**1.803**	**76.17**	**18.36**	**24.87**	**27.02**
SeLU	223*M*	11.1*T*	3.55	2.315 ± 0.004	1.948	68.76	16.76	22.75	25.99
SwiGLU	223*M*	11.1*T*	3.53	2.127 ± 0.003	**1.789**	**76.00**	**18.20**	**24.34**	**27.02**
LiGLU	223*M*	11.1*T*	3.59	2.149 ± 0.005	**1.798**	**75.34**	**17.97**	**24.34**	26.53
Sigmoid	223*M*	11.1*T*	3.63	2.291 ± 0.019	1.867	**74.31**	17.51	23.02	26.30
Softplus	223*M*	11.1*T*	3.47	2.207 ± 0.011	1.850	**72.45**	17.65	**24.34**	**26.89**
RMS Norm	223*M*	11.1*T*	3.68	2.167 ± 0.008	**1.821**	**75.45**	**17.94**	**24.07**	**27.14**
Rezero	223*M*	11.1*T*	3.51	2.262 ± 0.003	1.939	61.69	15.64	20.90	26.37
Rezero + LayerNorm	223*M*	11.1*T*	3.26	2.223 ± 0.006	1.858	70.42	17.58	23.02	26.29
Rezero + RMS Norm	223*M*	11.1*T*	3.34	2.221 ± 0.009	1.875	70.33	17.32	23.02	26.19
Fixup	223*M*	11.1*T*	2.95	2.382 ± 0.012	2.067	58.56	14.42	23.02	26.31
24 layers, $d_{ff} = 1536$, $H = 6$	224*M*	11.1*T*	3.33	2.200 ± 0.007	1.843	**74.89**	17.75	**25.13**	**26.89**
18 layers, $d_{ff} = 2048$, $H = 8$	223*M*	11.1*T*	3.38	2.185 ± 0.005	**1.831**	**76.45**	16.83	**24.34**	**27.10**
8 layers, $d_{ff} = 4608$, $H = 18$	223*M*	11.1*T*	3.69	2.190 ± 0.005	1.847	**74.58**	17.69	**23.28**	**26.85**
6 layers, $d_{ff} = 6144$, $H = 24$	223*M*	11.1*T*	3.70	2.201 ± 0.010	1.857	**73.55**	17.59	**24.60**	**26.66**
Block sharing	65*M*	11.1*T*	3.91	2.497 ± 0.037	2.164	64.50	14.53	21.96	25.48
+ Factorized embeddings	45*M*	9.4*T*	4.21	2.631 ± 0.305	2.183	60.84	14.00	19.84	25.27
+ Factorized & shared embeddings	20*M*	9.1*T*	4.37	2.907 ± 0.313	2.385	53.95	11.37	19.84	25.19
Encoder only block sharing	170*M*	11.1*T*	3.68	2.298 ± 0.023	1.929	69.60	16.23	23.02	26.23
Decoder only block sharing	144*M*	11.1*T*	3.70	2.352 ± 0.029	2.082	67.93	16.13	**23.81**	26.08
Factorized Embedding	227*M*	9.4*T*	3.80	2.208 ± 0.006	1.855	70.41	15.92	22.75	26.50
Factorized & shared embeddings	202*M*	9.1*T*	3.92	2.320 ± 0.010	1.952	68.69	16.33	22.22	26.44
Tied encoder/decoder input embeddings	248*M*	11.1*T*	3.55	2.192 ± 0.002	1.840	**71.70**	17.72	**24.34**	26.49
Tied decoder input and output embeddings	248*M*	11.1*T*	3.57	2.187 ± 0.007	**1.827**	**74.86**	17.74	**24.87**	**26.67**
Untied embeddings	273*M*	11.1*T*	3.53	2.195 ± 0.005	**1.834**	**72.99**	17.58	**23.28**	26.48
Adaptive input embeddings	204*M*	9.2*T*	3.55	2.250 ± 0.002	1.899	66.57	16.21	**24.07**	**26.66**
Adaptive softmax	204*M*	9.2*T*	3.60	2.364 ± 0.005	1.982	**72.91**	16.67	21.16	25.56
Adaptive softmax without projection	223*M*	10.8*T*	3.43	2.229 ± 0.009	1.914	**71.82**	17.10	23.02	25.72
Mixture of softmaxes	232*M*	16.3*T*	2.24	2.227 ± 0.017	**1.821**	**76.77**	17.62	22.75	**26.82**
Transparent attention	223*M*	11.1*T*	3.33	2.181 ± 0.014	1.874	54.31	10.40	21.16	**26.80**
Dynamic convolution	257*M*	11.8*T*	2.65	2.403 ± 0.009	2.047	58.30	12.67	21.16	17.03
Lightweight convolution	224*M*	10.4*T*	4.07	2.370 ± 0.010	1.989	63.07	14.86	23.02	24.73
Evolved Transformer	217*M*	9.9*T*	3.09	2.220 ± 0.003	1.863	**73.67**	10.76	**24.07**	26.58
Synthesizer (dense)	224*M*	11.4*T*	3.47	2.334 ± 0.021	1.962	61.03	14.27	16.14	**26.63**
Synthesizer (dense plus)	243*M*	12.6*T*	3.22	2.191 ± 0.010	1.840	**73.98**	16.96	**23.81**	**26.71**
Synthesizer (dense plus alpha)	243*M*	12.6*T*	3.01	2.180 ± 0.007	**1.828**	**74.25**	17.02	**23.28**	26.61
Synthesizer (factorized)	207*M*	10.1*T*	3.94	2.341 ± 0.017	1.968	62.78	15.39	**23.55**	26.42
Synthesizer (random)	254*M*	10.1*T*	4.08	2.326 ± 0.012	2.009	54.27	10.35	19.56	26.44
Synthesizer (random plus)	292*M*	12.0*T*	3.63	2.189 ± 0.004	1.842	**73.32**	17.04	**24.87**	26.43
Synthesizer (random plus alpha)	292*M*	12.0*T*	3.42	2.186 ± 0.007	**1.828**	**75.24**	17.08	**24.08**	26.39
Universal Transformer	84*M*	40.0*T*	0.88	2.406 ± 0.036	2.053	70.13	14.09	19.05	23.91
Mixture of experts	648*M*	11.7*T*	3.20	2.148 ± 0.006	**1.785**	**74.55**	**18.13**	**24.08**	**26.94**
Switch Transformer	1100*M*	11.7*T*	3.18	2.135 ± 0.007	**1.758**	**75.38**	**18.02**	**26.19**	**26.81**
Funnel Transformer	223*M*	1.9*T*	4.30	2.288 ± 0.008	1.918	67.34	16.26	22.75	23.20
Weighted Transformer	280*M*	71.0*T*	0.59	2.378 ± 0.021	1.989	69.04	16.98	23.02	26.30
Product key memory	421*M*	386.6*T*	0.25	2.155 ± 0.003	**1.798**	**75.16**	17.04	**23.55**	**26.73**

Do Transformer Modifications Transfer Across Implementations and Applications?

Sharan Narang* Hyung Won Chung Yi Tay William Fedus
Thibault Fevry† Michael Matena† Karishma Malkan† Noah Fiedel
Noam Shazeer Zhenzhong Lan† Yanqi Zhou Wei Li
Nan Ding Jake Marcus Adam Roberts Colin Raffel†

Parting remarks

- Pretraining on Tuesday!
- Good luck on assignment 4!
- Remember to work on your project proposal!