



北京航空航天大學
BEIHANG UNIVERSITY

Large Language Models

School of Artificial Intelligence

Instructor

Si Liu, Lei Sha, Rui She, Jinyang Guo

Preliminaries

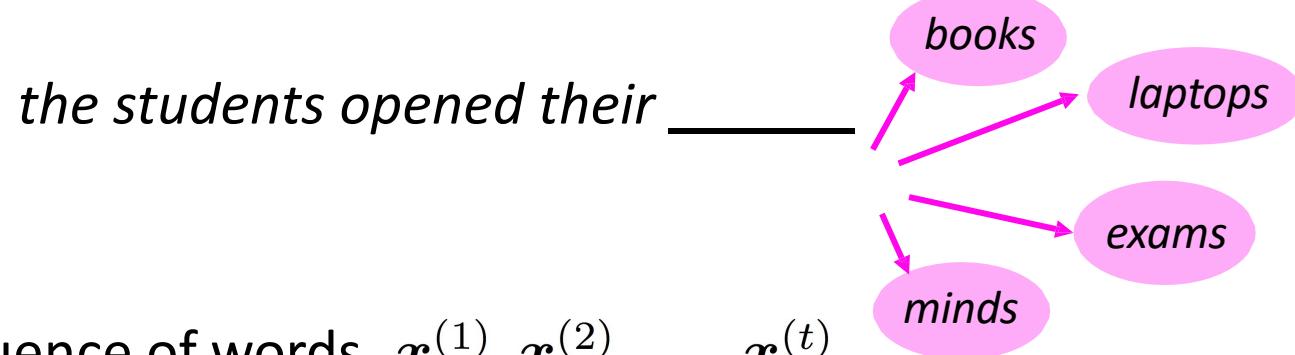
- This course:
 - 32 Class Hours, 2 credits
 - Instructor: Si Liu, Lei Sha, Rui She, Jinyang Guo
 - Score: (Regular Performance) 60% + Course Project(40%)
- Regular Performance= Attendance + Paper Reading Presentation
- Curriculum:
 - Lei Sha: LLM basics(LM, Transformers, PLM, LLM...) week 1~4
 - Rui She: Multi-modal LLMs week 5~8
 - Si Liu: Embodied Intelligence week 9~12
 - Jinyang Guo: Efficiency LLMs week 13~16
- Website: <https://shalei120.github.io/course/LLM/LLM.html>



Language Models

Language Modeling

- **Language Modeling** is the task of predicting what word comes next



- More formally: given a sequence of words $x^{(1)}, x^{(2)}, \dots, x^{(t)}$, compute the probability distribution of the next word $x^{(t+1)}$:

$$P(x^{(t+1)} | x^{(t)}, \dots, x^{(1)})$$

where $x^{(t+1)}$ can be any word in the vocabulary $V = \{w_1, \dots, w_{|V|}\}$

- A system that does this is called a **Language Model**

Language Modeling

- You can also think of a Language Model as a system that **assigns a probability to a piece of text**
- For example, if we have some text $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(T)}$, then the probability of this text (according to the Language Model) is:

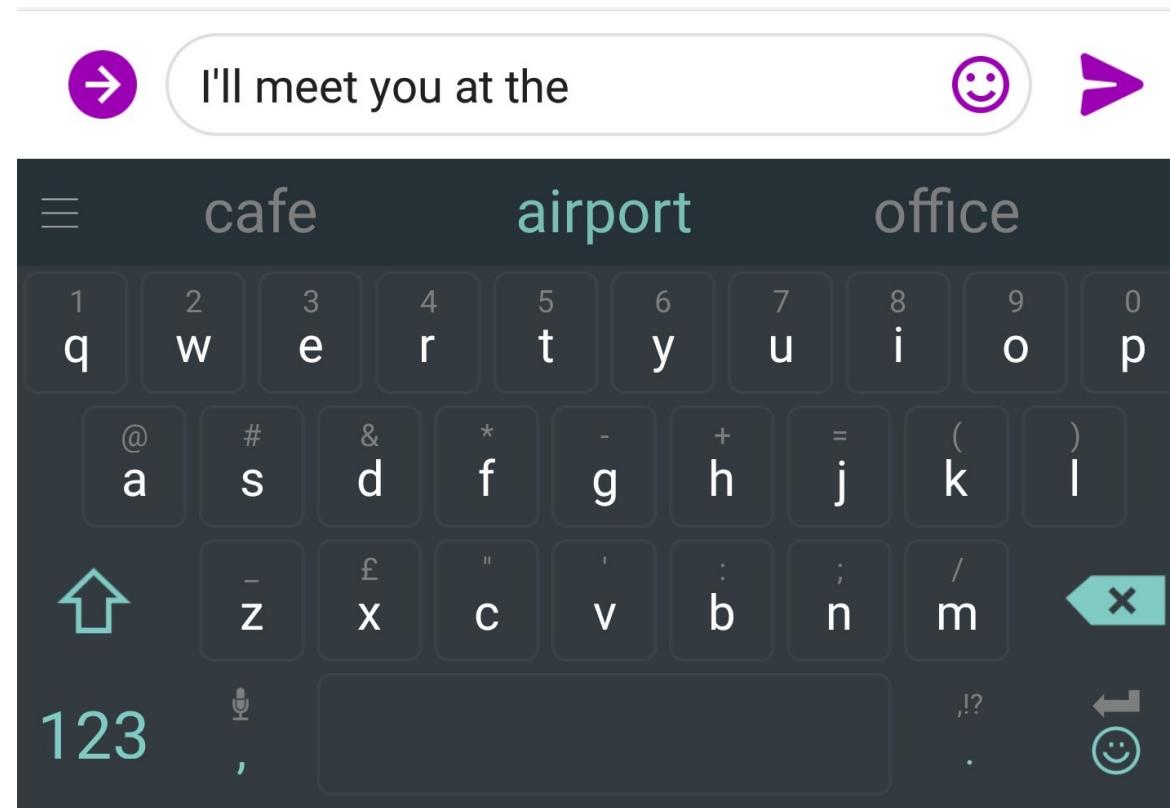
$$P(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(T)}) = P(\mathbf{x}^{(1)}) \times P(\mathbf{x}^{(2)} | \mathbf{x}^{(1)}) \times \dots \times P(\mathbf{x}^{(T)} | \mathbf{x}^{(T-1)}, \dots, \mathbf{x}^{(1)})$$

$$= \prod_{t=1}^T P(\mathbf{x}^{(t)} | \mathbf{x}^{(t-1)}, \dots, \mathbf{x}^{(1)})$$

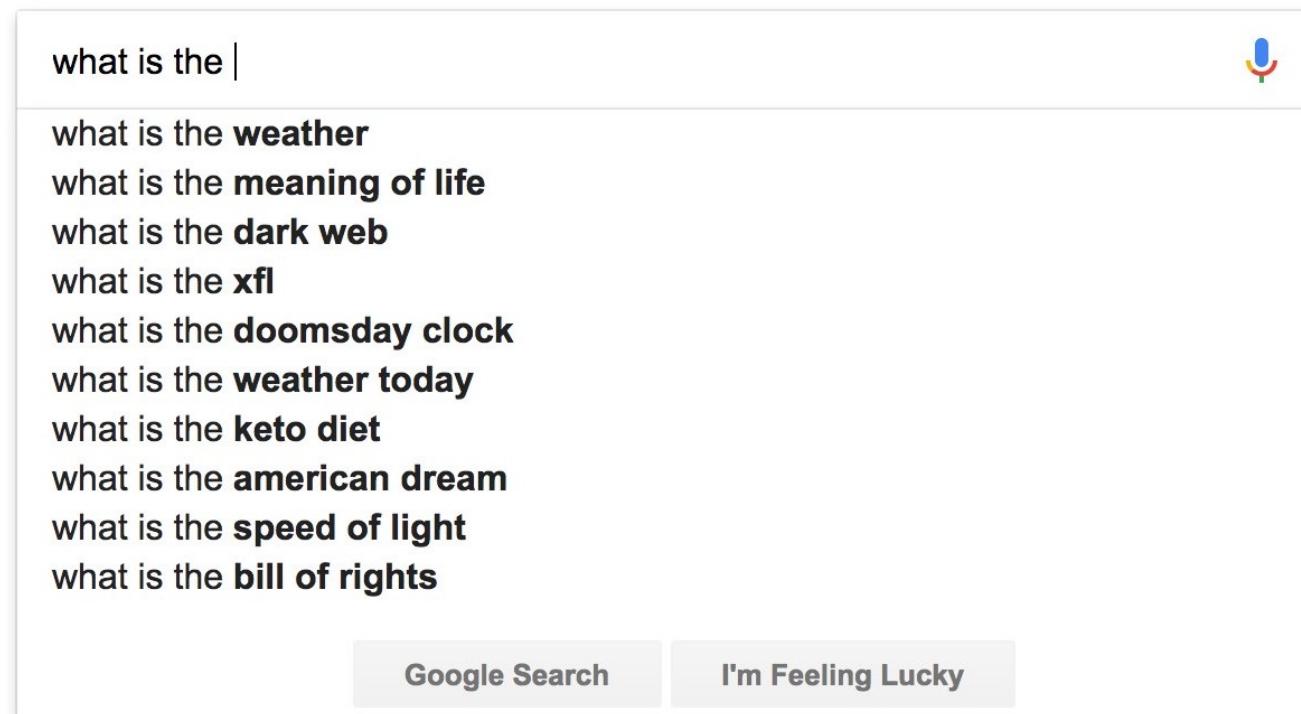


This is what our LM provides

You use Language Models every day!



You use Language Models every day!



A screenshot of a Google search interface. The search bar at the top contains the text "what is the |". To the right of the search bar is a microphone icon with three colored dots (blue, red, and green). Below the search bar is a list of suggested search queries, each preceded by a small blue dot. The suggestions are:

- what is the **weather**
- what is the **meaning of life**
- what is the **dark web**
- what is the **xfl**
- what is the **doomsday clock**
- what is the **weather today**
- what is the **keto diet**
- what is the **american dream**
- what is the **speed of light**
- what is the **bill of rights**

At the bottom of the interface are two buttons: "Google Search" on the left and "I'm Feeling Lucky" on the right.

n-gram Language Models

the students opened their _____

- **Question:** How to learn a Language Model?
- **Answer** (pre- Deep Learning): learn an *n*-gram Language Model!
- **Definition:** An *n*-gram is a chunk of *n* consecutive words.
 - **unigrams:** “the”, “students”, “opened”, “their”
 - **bigrams:** “the students”, “students opened”, “opened their”
 - **trigrams:** “the students opened”, “students opened their”
 - **four-grams:** “the students opened their”
- **Idea:** Collect statistics about how frequent different n-grams are and use these to predict next word.

n-gram Language Models

- First we make a **Markov assumption**: $x^{(t+1)}$ depends only on the preceding $n-1$ words

$$P(x^{(t+1)} | x^{(t)}, \dots, x^{(1)}) = P(x^{(t+1)} | \underbrace{x^{(t)}, \dots, x^{(t-n+2)}}_{n-1 \text{ words}}) \quad (\text{assumption})$$

$$\begin{aligned} \text{prob of a n-gram} &\rightarrow P(x^{(t+1)}, x^{(t)}, \dots, x^{(t-n+2)}) \\ \text{prob of a (n-1)-gram} &\rightarrow P(x^{(t)}, \dots, x^{(t-n+2)}) \end{aligned} \quad (\text{definition of conditional prob})$$

- Question:** How do we get these n -gram and $(n-1)$ -gram probabilities?
- Answer:** By **counting** them in some large corpus of text!

$$\approx \frac{\text{count}(x^{(t+1)}, x^{(t)}, \dots, x^{(t-n+2)})}{\text{count}(x^{(t)}, \dots, x^{(t-n+2)})} \quad (\text{statistical approximation})$$

n-gram Language Models: Example

Suppose we are learning a **4-gram** Language Model.

~~as the proctor started the clock, the students opened their~~ _____
discard _____
condition on this _____

$$P(\mathbf{w}|\text{students opened their}) = \frac{\text{count(students opened their } \mathbf{w}\text{)}}{\text{count(students opened their)}}$$

For example, suppose that in the corpus:

- “students opened their” occurred 1000 times
 - “students opened their *books*” occurred 400 times
 - $\rightarrow P(\text{books} \mid \text{students opened their}) = 0.4$
 - “students opened their *exams*” occurred 100 times
 - $\rightarrow P(\text{exams} \mid \text{students opened their}) = 0.1$

- Should we have discarded the “proctor” context?

Sparsity Problems with n-gram Language Models

Sparsity Problem 1

Problem: What if “*students opened their w*” never occurred in data? Then w has probability 0!

(Partial) Solution: Add small δ to the count for every $w \in V$. This is called *smoothing*.

$$P(w|\text{students opened their}) = \frac{\text{count(students opened their } w\text{)}}{\text{count(students opened their)}}$$

Sparsity Problem 2

Problem: What if “*students opened their*” never occurred in data? Then we can’t calculate probability for *any* w !

(Partial) Solution: Just condition on “*opened their*” instead. This is called *backoff*.

Note: Increasing n makes sparsity problems worse.
Typically, we can’t have n bigger than 5.

Storage Problems with n-gram Language Models

Storage: Need to store count for all n -grams you saw in the corpus.

$$P(\mathbf{w}|\text{students opened their}) = \frac{\text{count(students opened their } \mathbf{w})}{\text{count(students opened their)}}$$

Increasing n or increasing corpus increases model size!

n-gram Language Models in practice

- You can build a simple trigram Language Model over a 1.7 million word corpus (Reuters) in a few seconds on your laptop*

today the _____

Business and financial news

get probability distribution

company	0.153
bank	0.153
price	0.077
italian	0.039
emirate	0.039
...	

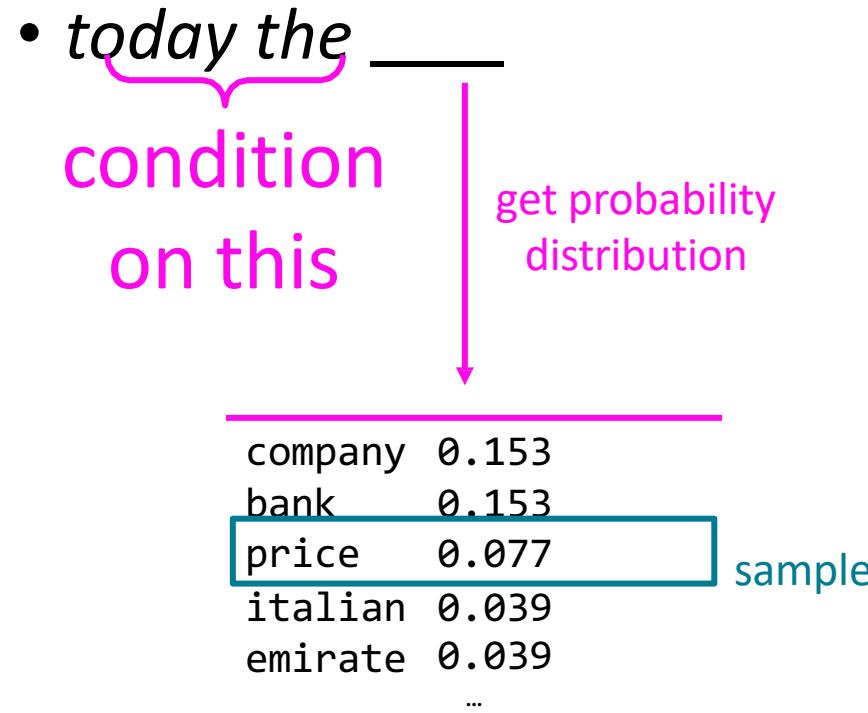
Sparsity problem:
not much granularity
in the probability
distribution

Otherwise, seems reasonable!

* Try for yourself: <https://nlpforhackers.io/language-models/>

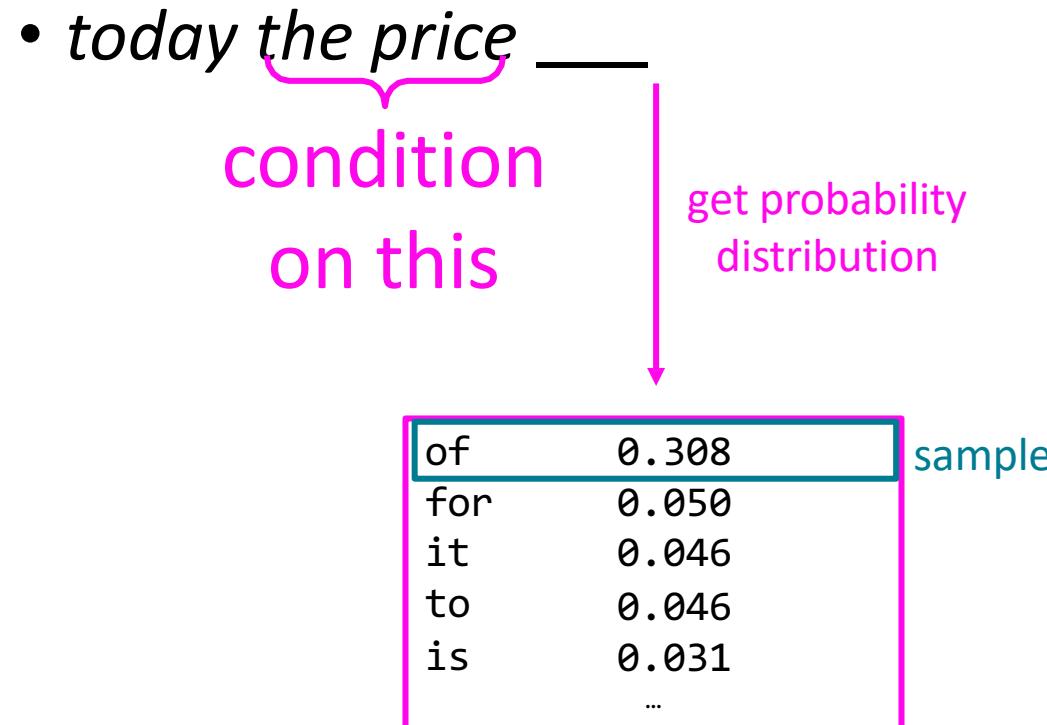
Generating text with a n-gram Language Model

- You can also use a Language Model to generate text



Generating text with a n-gram Language Model

- You can also use a Language Model to generate text



Generating text with a n-gram Language Model

- You can also use a Language Model to generate text

• *today the price of __*
condition
on this

get probability distribution

the	0.072
18	0.043
oil	0.043
its	0.036
gold	0.018
	...

sample

Generating text with a n-gram Language Model

You can also use a Language Model to **generate text**

*today the price of gold per ton , while production of shoe
lasts and shoe industry , the bank intervened just after it
considered and rejected an imf demand to rebuild depleted
european stocks , sept 30 end primary 76 cts a share .*

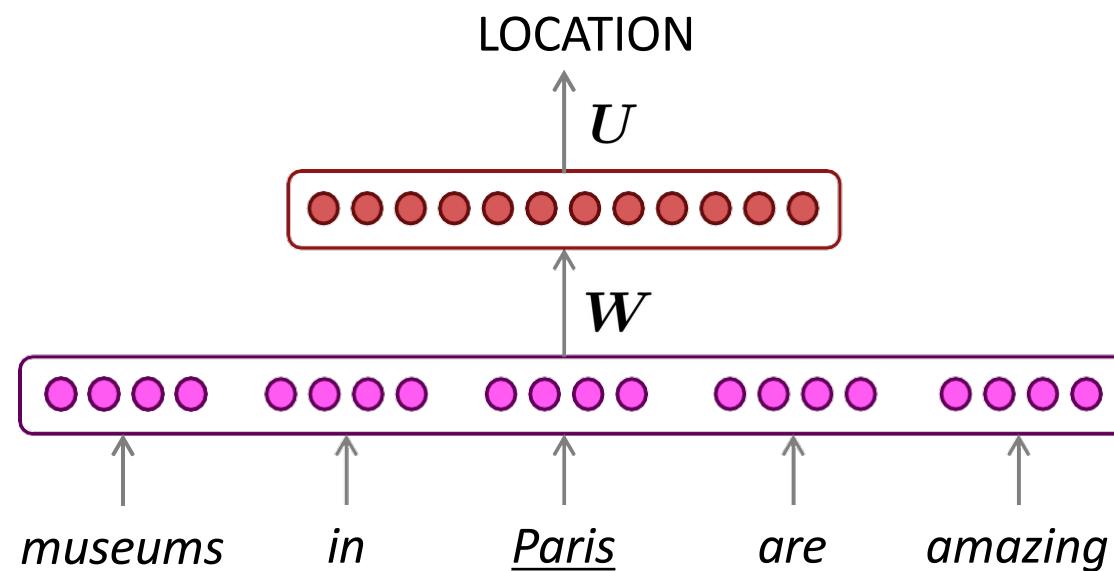
Surprisingly grammatical!

...but **incoherent**. We need to consider more than
three words at a time if we want to model language well.

But increasing n worsens sparsity problem,
and increases model size...

How to build a *neural* Language Model?

- Recall the Language Modeling task:
 - Input: sequence of words $x^{(1)}, x^{(2)}, \dots, x^{(t)}$
 - Output: prob dist of the next word $P(x^{(t+1)} | x^{(t)}, \dots, x^{(1)})$
- How about a [window-based neural model](#)?
 - We saw this applied to Named Entity Recognition in Lecture 3:



A fixed-window neural Language Model

as the proctor started the clock
the students opened their _____

discard

fixed window

A fixed-window neural Language Model

output distribution

$$\hat{y} = \text{softmax}(U\mathbf{h} + \mathbf{b}_2) \in \mathbb{R}^{|V|}$$

hidden layer

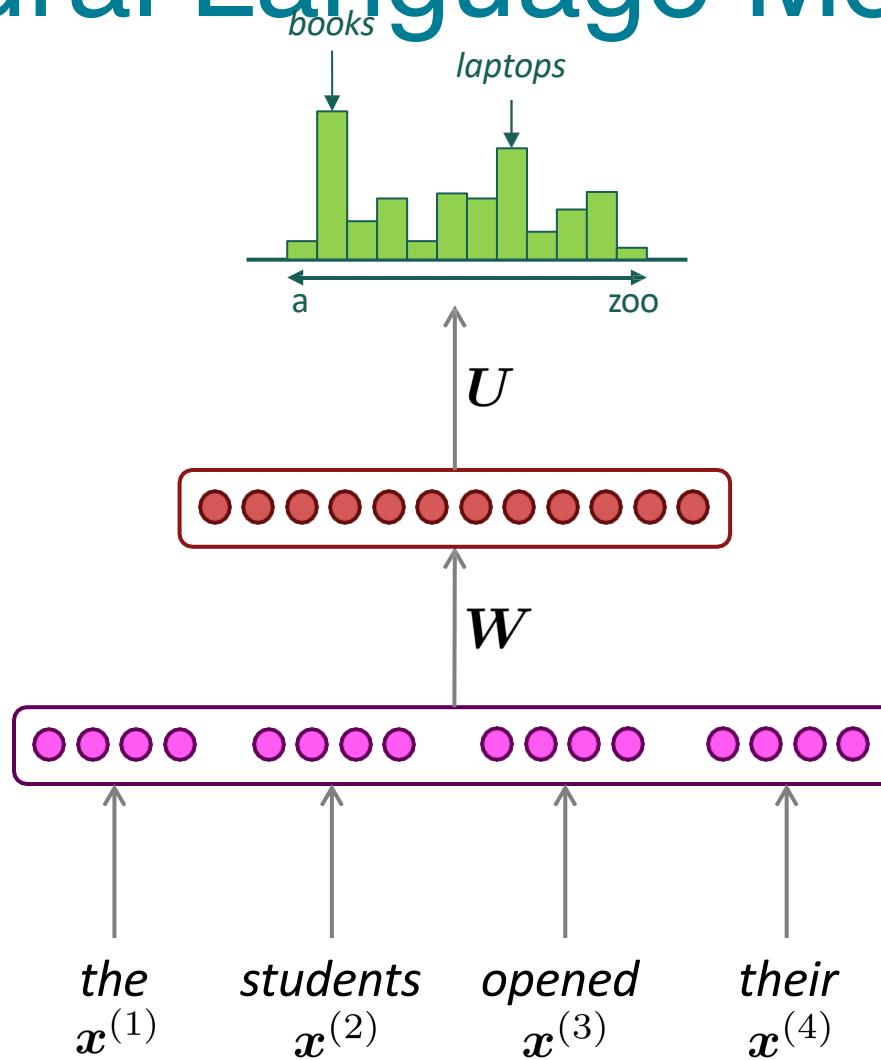
$$\mathbf{h} = f(\mathbf{W}\mathbf{e} + \mathbf{b}_1)$$

concatenated word embeddings

$$\mathbf{e} = [\mathbf{e}^{(1)}; \mathbf{e}^{(2)}; \mathbf{e}^{(3)}; \mathbf{e}^{(4)}]$$

words / one-hot vectors

$$\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)}, \mathbf{x}^{(4)}$$



A fixed-window neural Language Model

Approximately: Y. Bengio, et al. (2000/2003): A Neural Probabilistic Language Model

Improvements over n -gram LM:

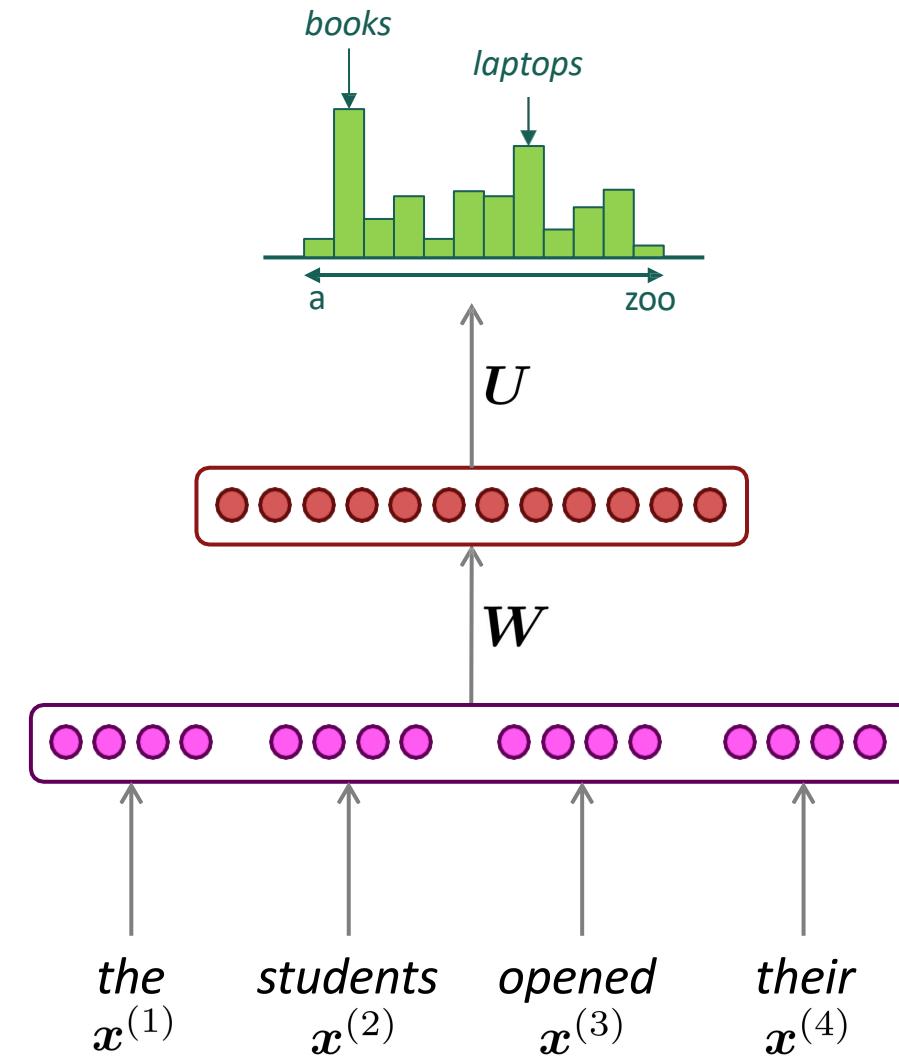
- No sparsity problem
- Don't need to store all observed n -grams

Remaining **problems**:

- Fixed window is **too small**
- Enlarging window enlarges W
- Window can never be large enough!
- $x^{(1)}$ and $x^{(2)}$ are multiplied by completely different weights in W .

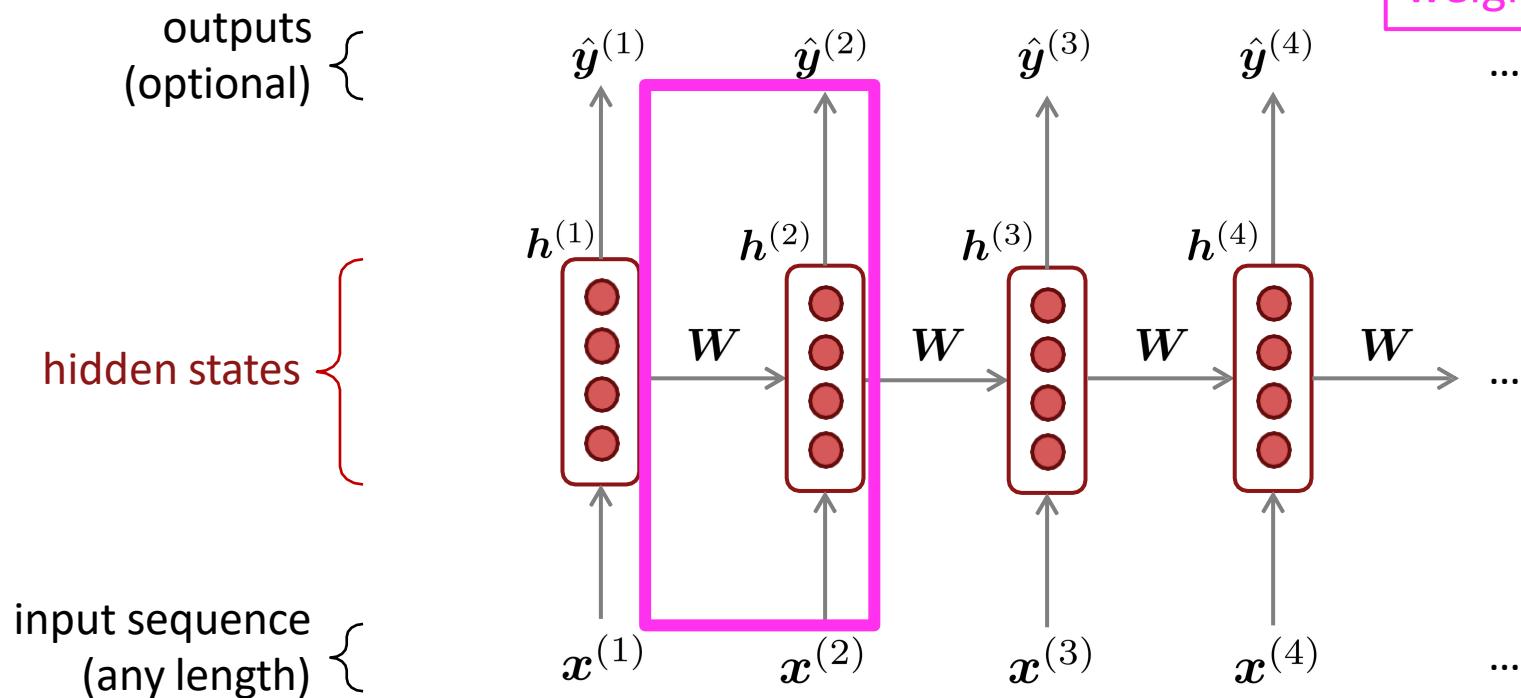
No symmetry in how the inputs are processed.

We need a neural architecture
that can process *any length input*



Recurrent Neural Networks (RNN)

A family of neural architectures



Core idea: Apply the same weights W repeatedly



Transformers

Contents

- Transformers
 - Impact of Transformers on NLP (and ML more broadly)
 - From Recurrence (RNNs) to Attention-Based NLP Models
 - Understanding the Transformer Model
 - Drawbacks and Variants of Transformers
- Pretraining Language Models(PLMs)
 - Subword modeling
 - Motivating model pretraining from word embeddings
 - Model pretraining three ways
 - Decoders
 - Encoders
 - Encoder-Decoders
 - Very large models and in-context learning

Contents

- Transformers
 - Impact of Transformers on NLP (and ML more broadly)
 - From Recurrence (RNNs) to Attention-Based NLP Models
 - Understanding the Transformer Model
 - Drawbacks and Variants of Transformers
- Pretraining Language Models(PLMs)
 - Subword modeling
 - Motivating model pretraining from word embeddings
 - Model pretraining three ways
 - Decoders
 - Encoders
 - Encoder-Decoders
 - Very large models and in-context learning

Transformers: Is Attention All We Need?

Spoiler: Not Quite!

Attention Is All You Need

Ashish Vaswani*
Google Brain
avaswani@google.com

Noam Shazeer*
Google Brain
noam@google.com

Niki Parmar*
Google Research
nikip@google.com

Jakob Uszkoreit*
Google Research
usz@google.com

Llion Jones*
Google Research
llion@google.com

Aidan N. Gomez* †
University of Toronto
aidan@cs.toronto.edu

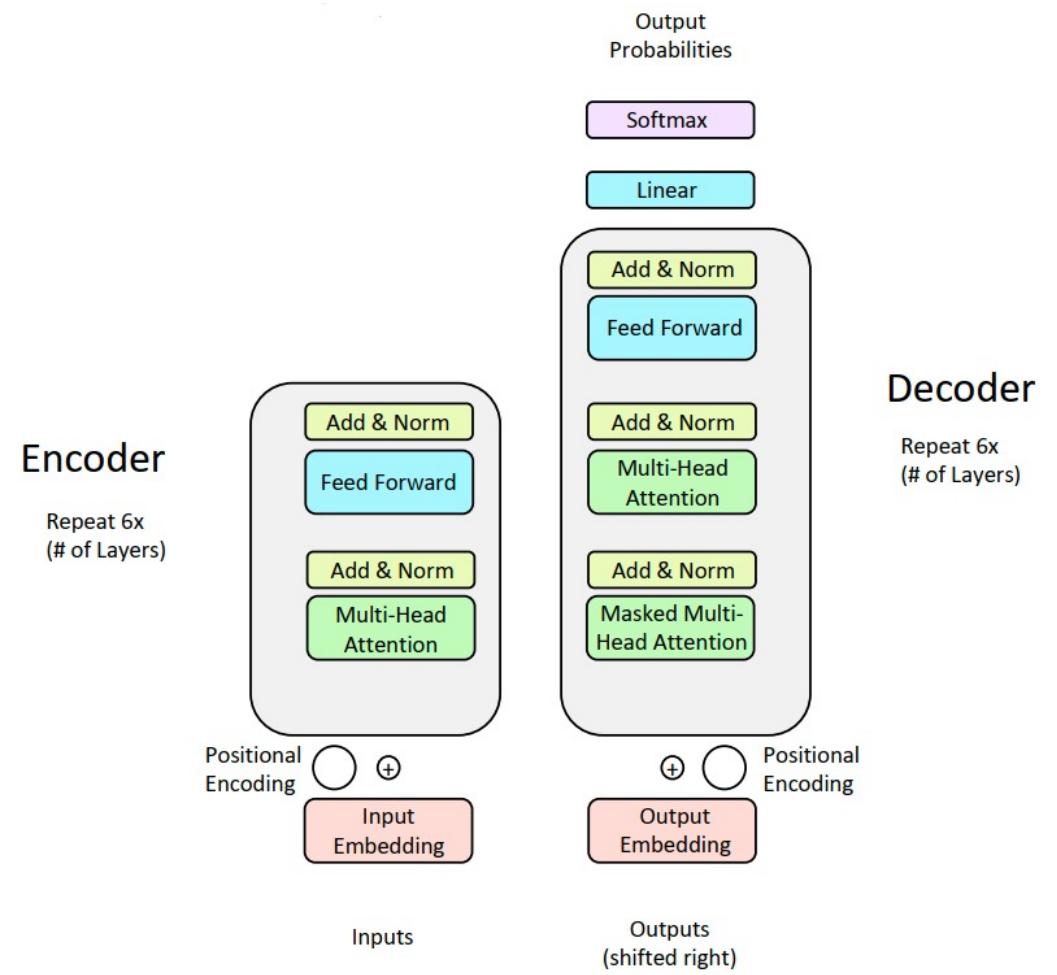
Łukasz Kaiser*
Google Brain
lukaszkaiser@google.com

Illia Polosukhin* ‡
illia.polosukhin@gmail.com

Transformers Have Revolutionized the Field of NLP



Courtesy of Paramount Pictures



Great Results with Transformers: Machine Translation

- First, Machine Translation results 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		$3.3 \cdot 10^{18}$
Transformer (big)	28.4	41.8		$2.3 \cdot 10^{19}$

Great Results with Transformers: Document Generation

- Next, document generation!
- (For perplexity, lower is better; for ROUGE-L, higher is better.)

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 from last week!

Transformers dominating across the board.

Preview: Great Results with (Pre-Trained) Transformers

Before too long, most Transformers results also incorporate **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:



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

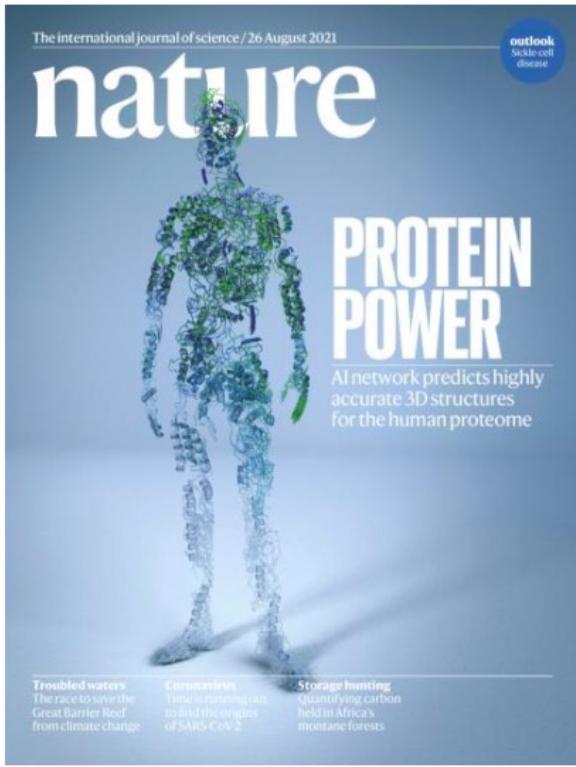
Rank Name	Model	URL Score
1 DeBERTa Team - Microsoft	DeBERTa / TuringNLv4	90.8
2 HFL iFLYTEK	MacALBERT + DKM	90.7
+ 3 Alibaba DAMO NLP	StructBERT + TAPT	90.6
+ 4 PING-AN Omni-Sinitic	ALBERT + DAAF + NAS	90.6
5 ERNIE Team - Baidu	ERNIE	90.4
6 T5 Team - Google	T5	90.3

More results Thursday when we discuss pretraining.

[Liu et al., 2018]

Transformers Even Show Promise Outside of NLP

Protein Folding



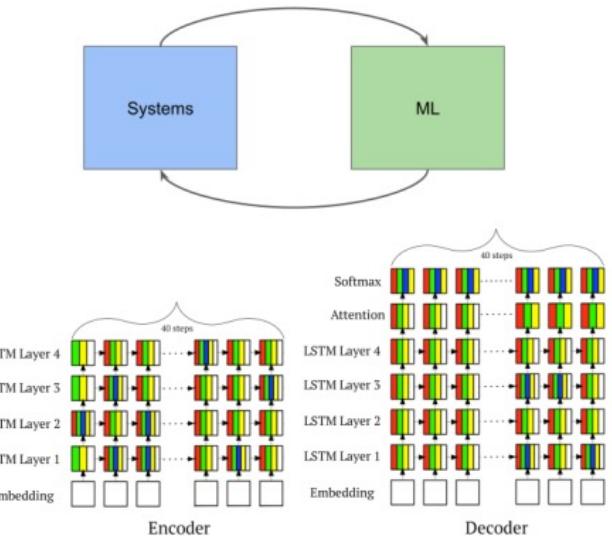
[Jumper et al. 2021] aka AlphaFold2!



Image Classification

[Dosovitskiy et al. 2020]: Vision Transformer (ViT) outperforms ResNet-based baselines with substantially less compute.

	Ours-JFT (ViT-H/14)	Ours-JFT (ViT-L/16)	Ours-I21k (ViT-L/16)	BiT-L (ResNet152x4)	Noisy Student (EfficientNet-L2)
ImageNet	88.55 ± 0.04	87.76 ± 0.03	85.30 ± 0.02	87.54 ± 0.02	88.4 / 88.5*
ImageNet ReaL	90.72 ± 0.05	90.54 ± 0.03	88.62 ± 0.05	90.54	90.55
CIFAR-10	99.50 ± 0.06	99.42 ± 0.03	99.15 ± 0.03	99.37 ± 0.06	—
CIFAR-100	94.55 ± 0.04	93.90 ± 0.05	93.25 ± 0.05	93.51 ± 0.08	—
Oxford-IIIT Pets	97.56 ± 0.03	97.32 ± 0.11	94.67 ± 0.15	96.62 ± 0.23	—
Oxford Flowers-102	99.68 ± 0.02	99.74 ± 0.00	99.61 ± 0.02	99.63 ± 0.03	—
VTAB (19 tasks)	77.63 ± 0.23	76.28 ± 0.46	72.72 ± 0.21	76.29 ± 1.70	—
TPUv3-core-days	2.5k	0.68k	0.23k	9.9k	12.3k



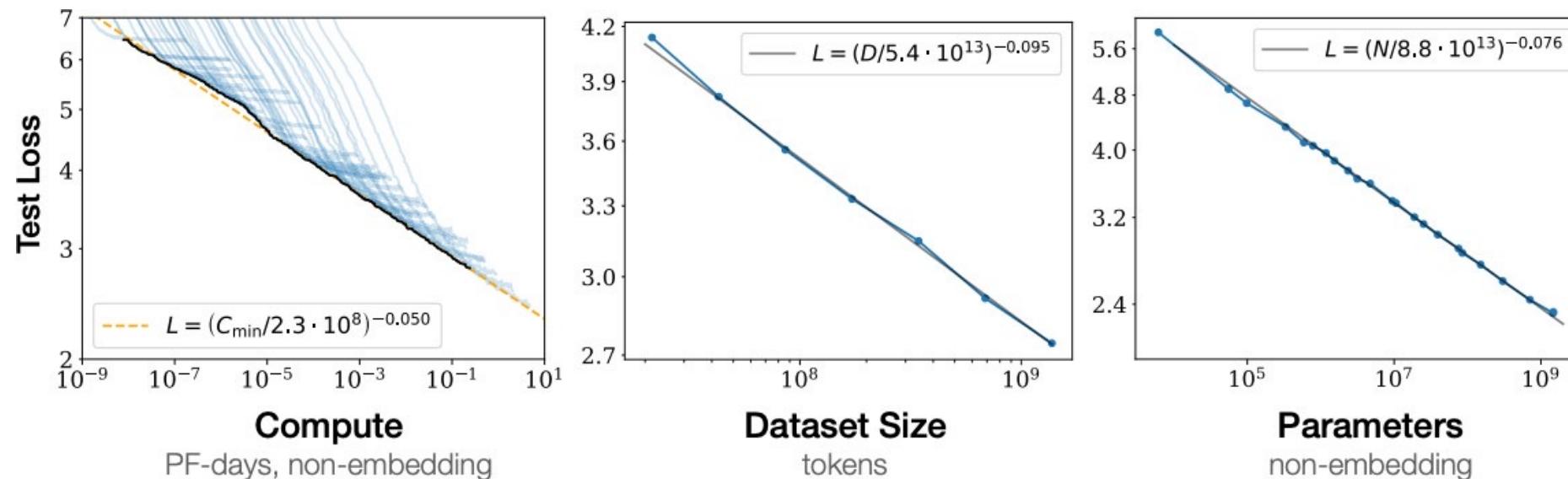
ML for Systems

[Zhou et al. 2020]: A Transformer-based compiler model (GO-one) speeds up a Transformer model!

Model (#devices)	GO-one (s)	HP (s)	METIS (s)	HDP (s)	Run time speed up over HP / HDP	Search speed up over HDP
2-layer RNNLM (2)	0.173	0.192	0.355	0.191	9.9% / 9.4%	2.95x
4-layer RNNLM (4)	0.210	0.239	0.503	0.251	13.8% / 16.3%	1.76x
8-layer RNNLM (8)	0.320	0.332	0.644	0.764	3.8% / 58.1%	27.8x
2-layer GNMT (2)	0.301	0.384	0.344	0.327	27.6% / 14.3%	30x
4-layer GNMT (4)	0.350	0.469	0.466	0.432	34% / 23.4%	58.8x
8-layer GNMT (8)	0.440	0.562	0.600	0.693	21.7% / 36.5%	7.35x
2-layer Transformer-XL (2)	0.223	0.268	0.37	0.262	20.1% / 17.4%	40x
4-layer Transformer-XL (4)	0.230	0.27	0.40M	0.259	17.4% / 12.6%	26.7x
8-layer Transformer-XL (8)	0.350	0.46	0.60M	0.425	23.9% / 16.7%	16.7x
Inception (2) b64	0.229	0.312	0.60M	0.301	26.6% / 23.9%	13.5x
Inception (2) b64	0.423	0.731	0.60M	0.498	42.1% / 29.3%	21.0x
AmoebaNet (4)	0.394	0.44	0.426	0.418	26.1% / 6.1%	58.8x
2-stack 18-layer WaveNet (2)	0.317	0.376	0.60M	0.354	18.6% / 11.7%	6.67x
4-stack 36-layer WaveNet (4)	0.659	0.988	0.60M	0.721	50% / 9.4%	20x
GEOOMEAN	-	-	-	-	20.5% / 18.2%	15x

Scaling Laws: Are Transformers All We Need?

- With Transformers, language modeling performance improves smoothly as we increase model size, training data, and compute resources.
- This power-law relationship has been observed over multiple orders of magnitude with no sign of slowing!
- If we keep scaling up these models (with no change to the architecture), could they eventually match or exceed human-level performance?



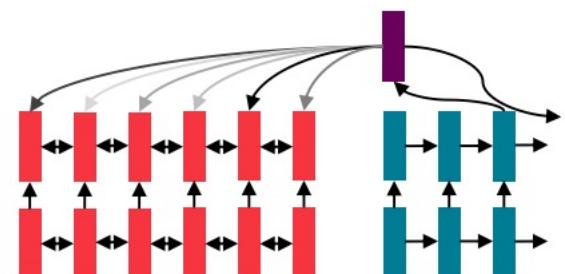
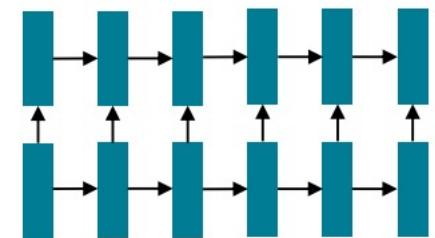
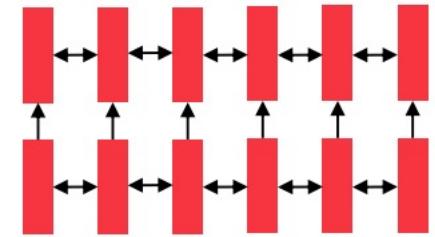
[Kaplan et al., 2020]

Contents

- Transformers
 - Impact of Transformers on NLP (and ML more broadly)
 - **From Recurrence (RNNs) to Attention-Based NLP Models**
 - Understanding the Transformer Model
 - Drawbacks and Variants of Transformers
- Pretraining Language Models(PLMs)
 - Subword modeling
 - Motivating model pretraining from word embeddings
 - Model pretraining three ways
 - Decoders
 - Encoders
 - Encoder-Decoders
 - Very large models and in-context learning

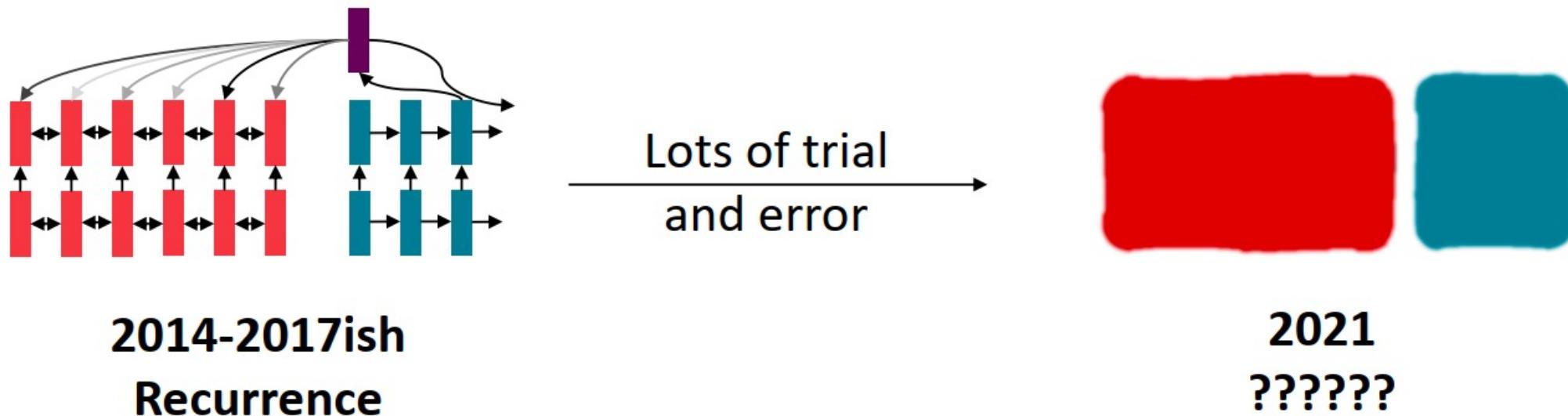
Recurrent models for (most) NLP!

- Since 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



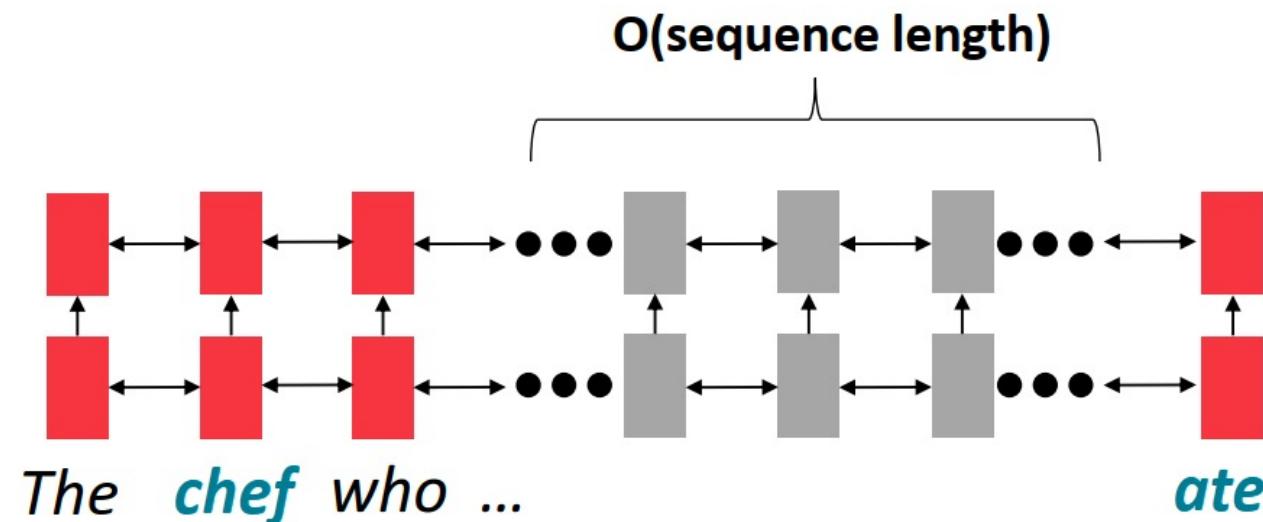
Today: Same goals, different building blocks

- We have 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.



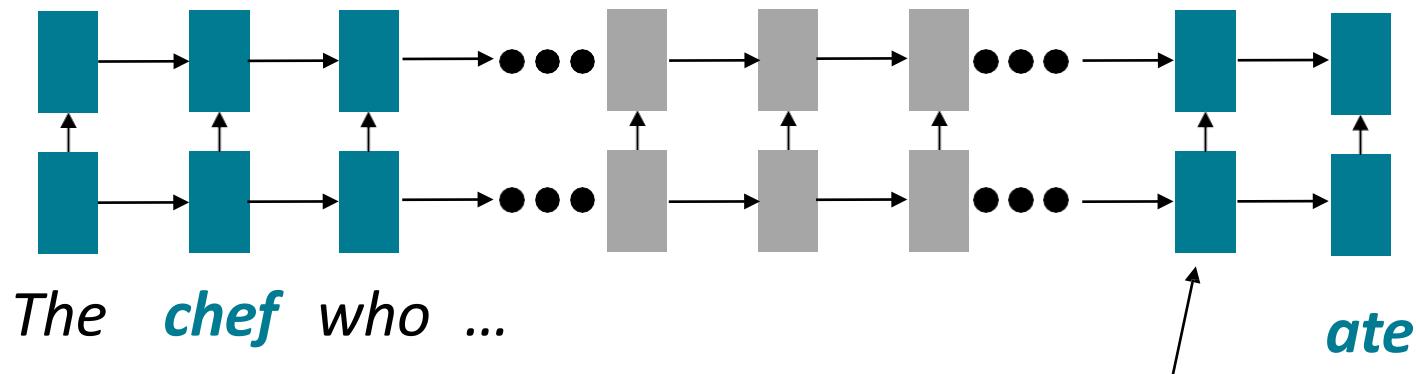
Issues with recurrent models: Linear interaction distance

- RNNs are unrolled “left-to-right”.
- It 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

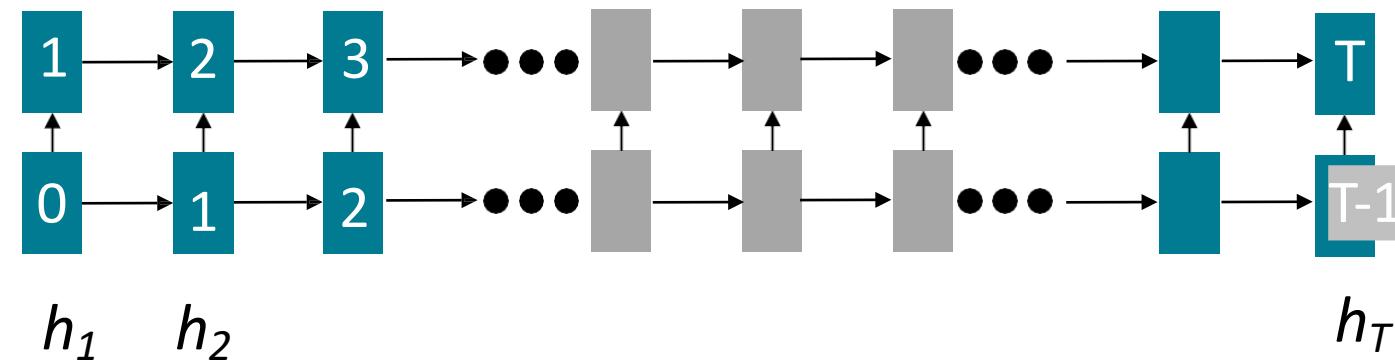
- **O(sequence length)** steps for distant word pairs to interact means:
 - Hard to learn long-distance dependencies (because gradient problems!)
 - Linear order of words is “baked in”; we already know sequential structure doesn't tell the whole story...



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

Issues with recurrent models: Lack of parallelizability

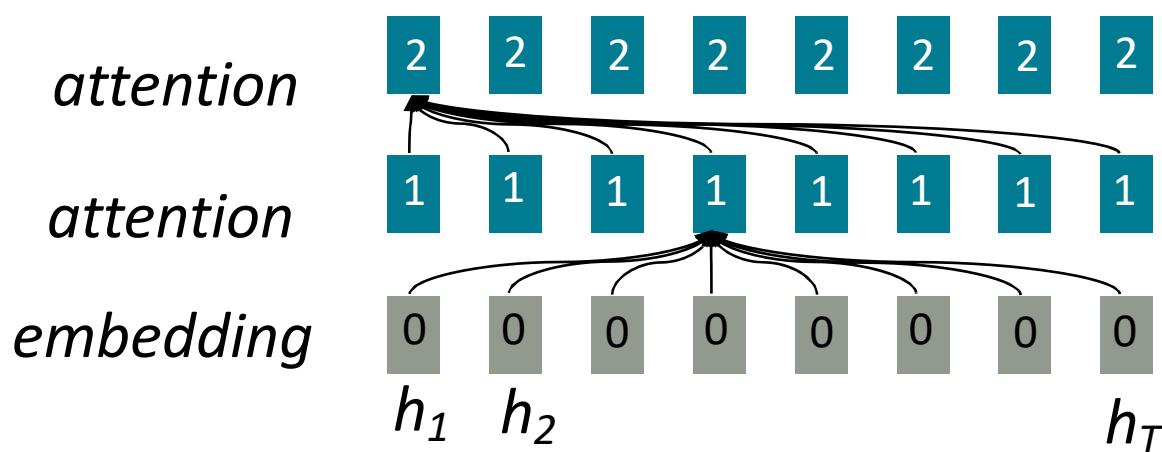
- Forward and backward passes have **O(seq length)** unparallelizable operations
 - GPUs (and TPUs) can perform many independent computations at once!
 - But future RNN hidden states can't be computed in full before past RNN hidden states have been computed
 - Inhibits training on very large datasets!
 - Particularly problematic as sequence length increases, as we can no longer batch many examples together due to memory limitations



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

If not recurrence, then what? How about (self) attention?

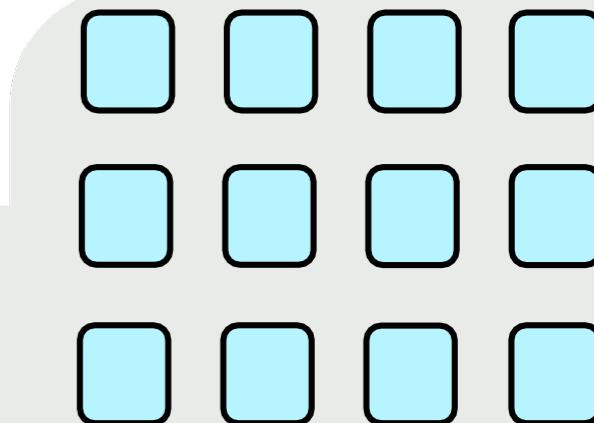
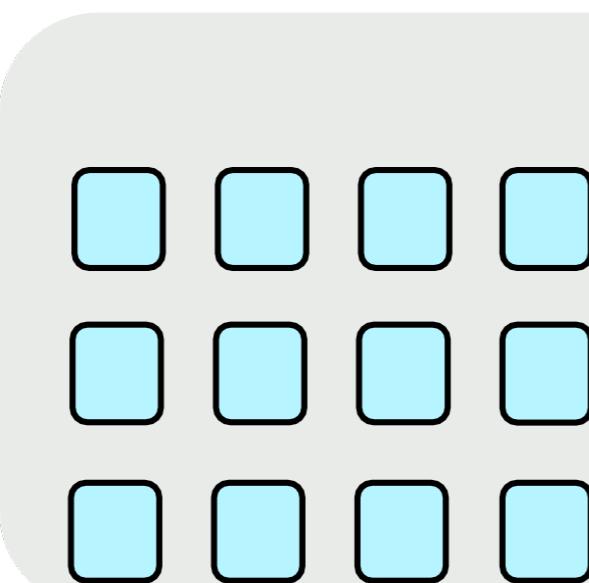
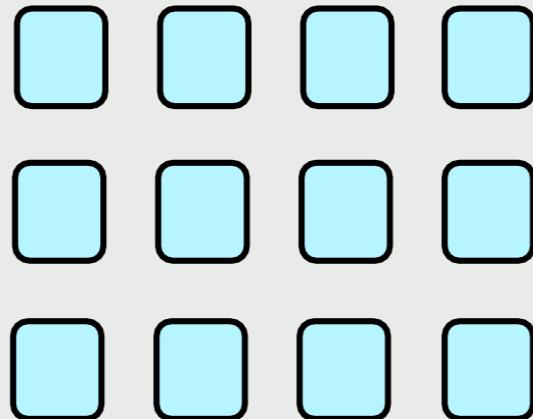
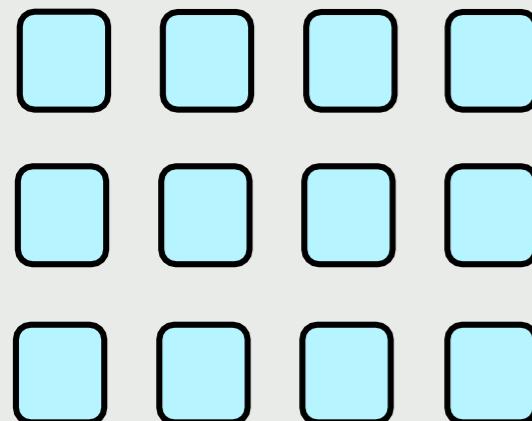
- To recap, **attention** treats each word's representation as a **query** to access and incorporate information from a **set of values**.
 - we saw attention from the **decoder** to the **encoder**;
 - Self-attention** is **encoder-encoder** (or **decoder-decoder**) attention where each word attends to each other word **within the input (or output)**.



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

Computational Dependencies for Recurrence vs. Attention

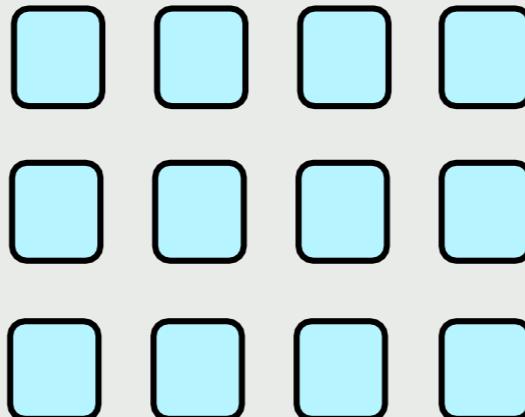
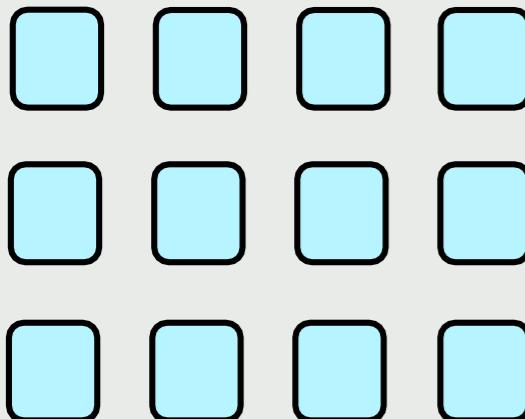
RNN-Based Encoder-Decoder
Model with Attention



Transformer-Based
Encoder-Decoder Model

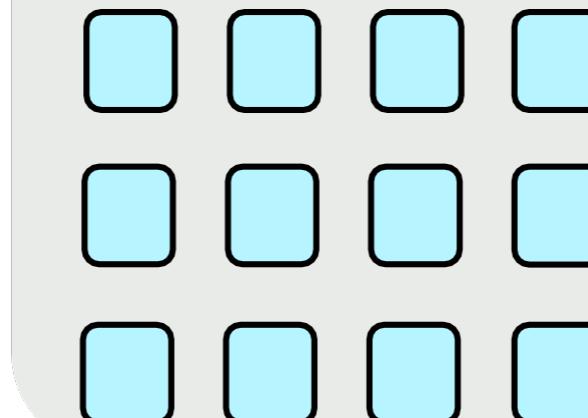
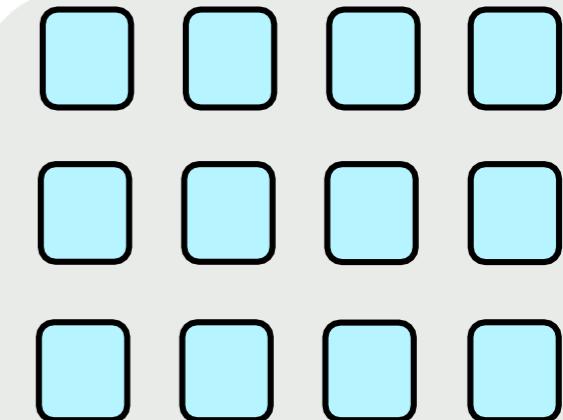
Computational Dependencies for Recurrence vs. Attention

RNN-Based Encoder-Decoder Model with Attention



Transformer Advantages:

- Number of unparallelizable operations does not increase with sequence length.
- Each "word" interacts with each other, so maximum interaction distance: $O(1)$.



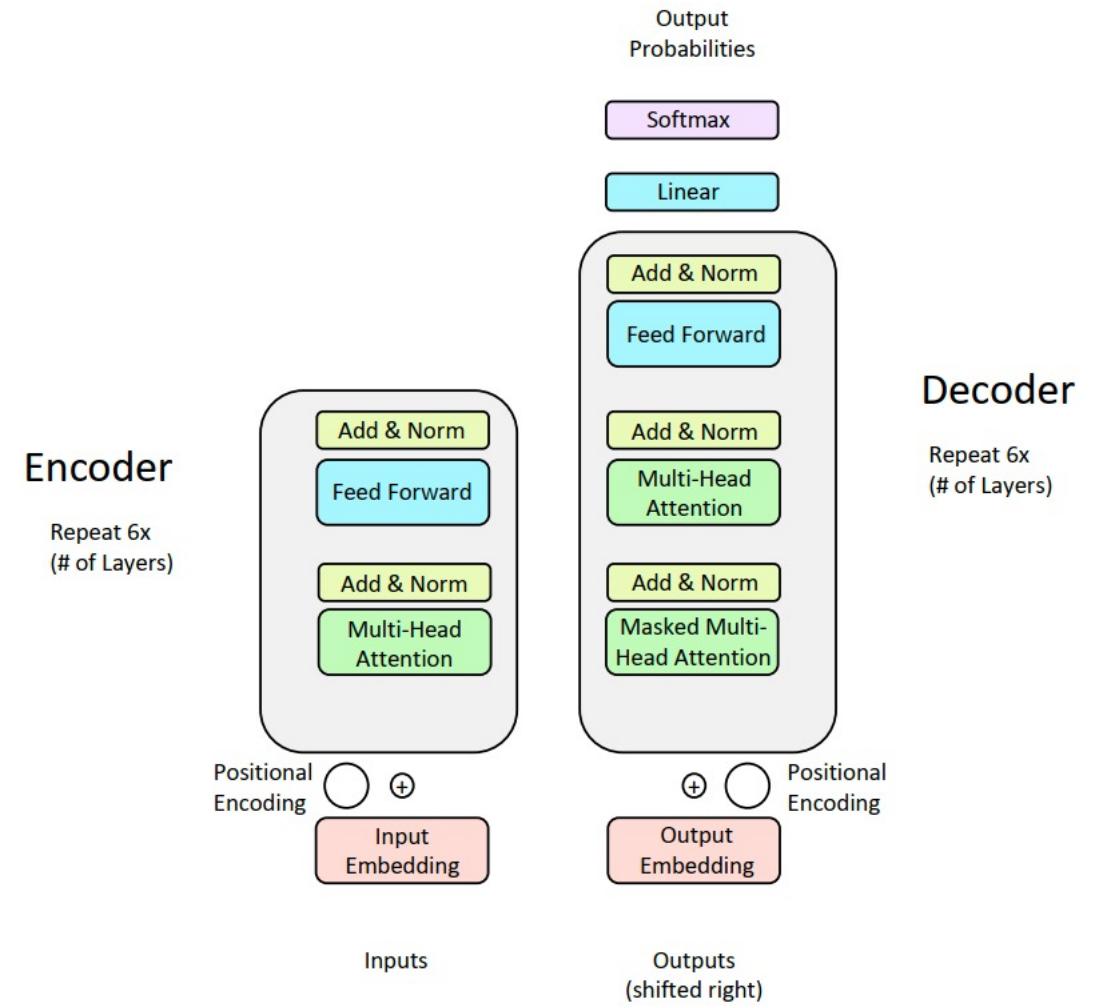
Transformer-Based Encoder-Decoder Model

Contents

- Transformers
 - Impact of Transformers on NLP (and ML more broadly)
 - From Recurrence (RNNs) to Attention-Based NLP Models
 - **Understanding the Transformer Model**
 - Drawbacks and Variants of Transformers
- Pretraining Language Models(PLMs)
 - Subword modeling
 - Motivating model pretraining from word embeddings
 - Model pretraining three ways
 - Decoders
 - Encoders
 - Encoder-Decoders
 - Very large models and in-context learning

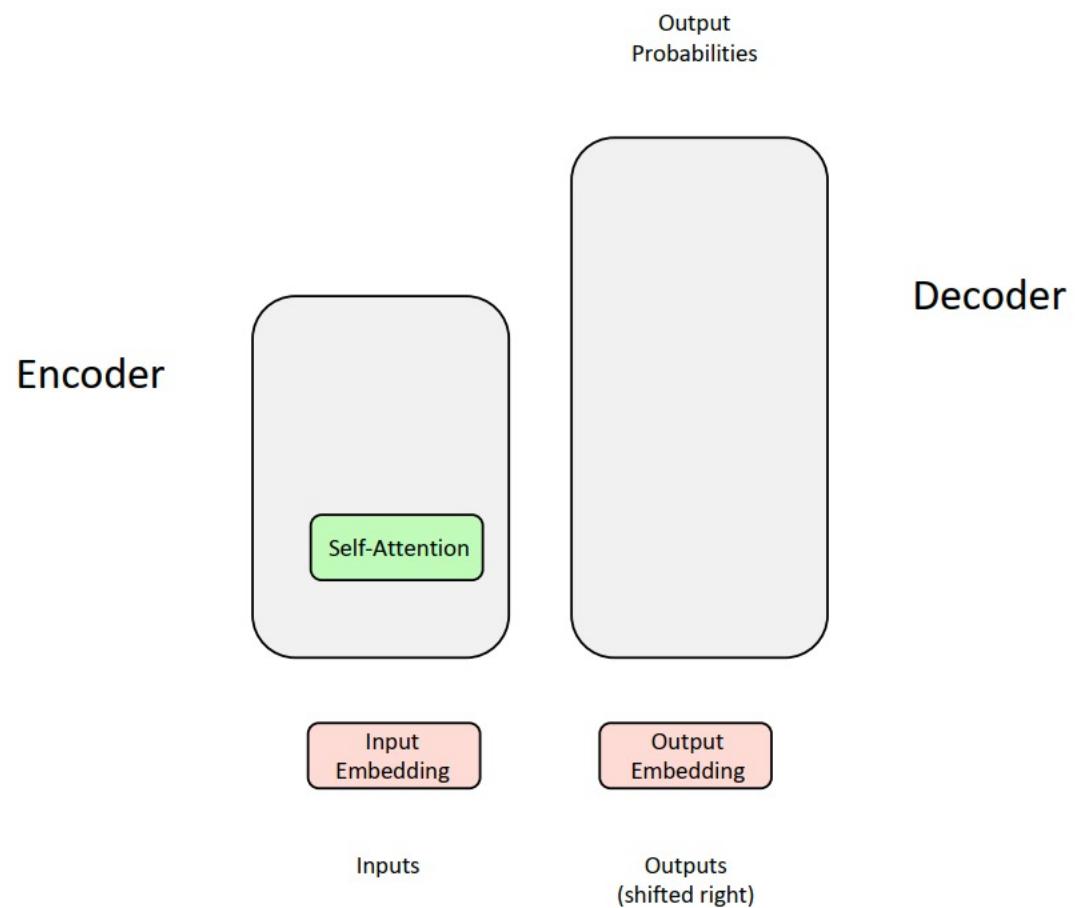
The Transformer Encoder-Decoder [Vaswani et al., 2017]

- In this section, you will learn exactly how the Transformer architecture works:
 - First, we will talk about the Encoder!
 - Next, we will go through the Decoder (which is quite similar)!



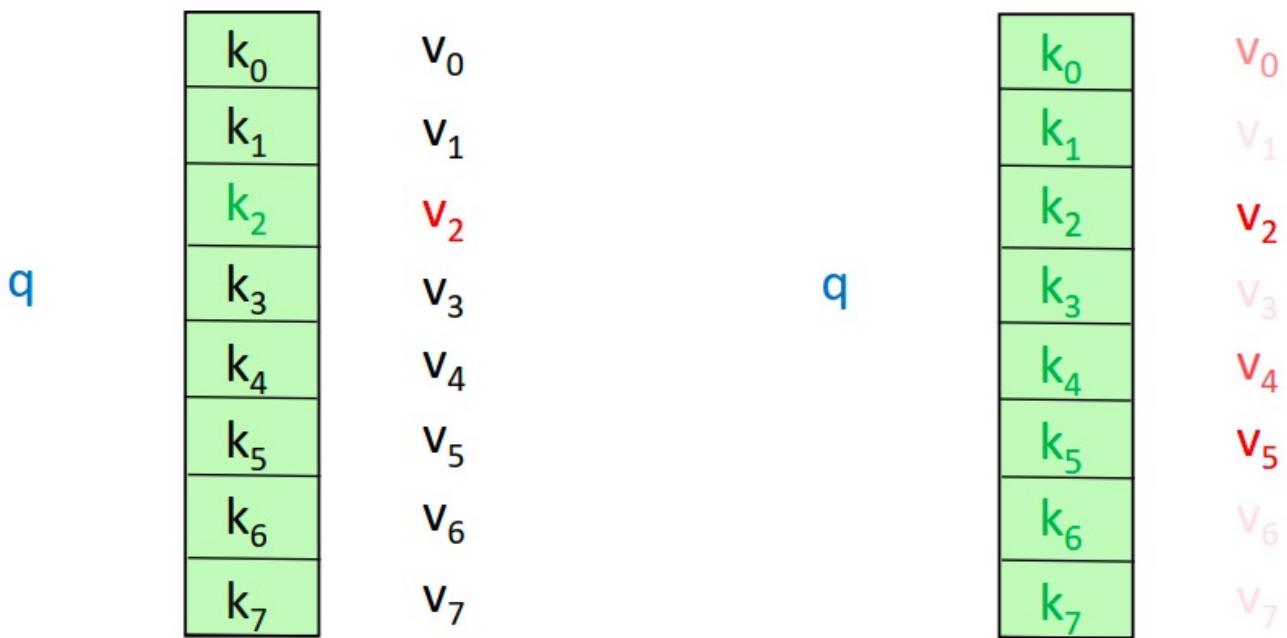
Encoder: Self-Attention

- Self-Attention is the core building block of Transformer, so let's first focus on that!



Intuition for Attention Mechanism

- Let's think of attention as a "fuzzy" or approximate hashtable:
 - To look up a **value**, we compare a **query** against **keys** in a table.
 - In a hashtable (shown on the bottom left):
 - Each **query** (hash) maps to exactly one **key-value** pair.
 - In (self-)attention (shown on the bottom right):
 - Each **query** matches each **key** to varying degrees.
 - We return a sum of **values** weighted by the **query-key** match



Recipe for Self-Attention in the Transformer Encoder

- Step 1: For each word , calculate its **query**, **key**, and **value**.

$$q_i = W^Q x_i \quad k_i = W^K x_i \quad v_i = W^V x_i$$

- Step 2: Calculate attention score between **query** and **keys**.

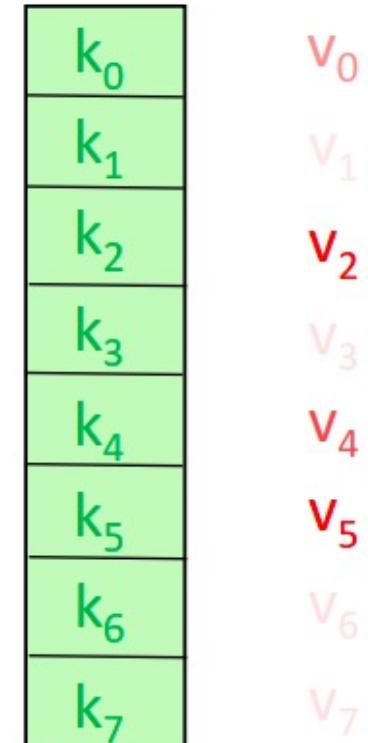
$$e_{ij} = q_i \cdot k_j$$

- Step 3: Take the softmax to normalize attention scores.

$$\alpha_{ij} = softmax(e_{ij}) = \frac{exp(e_{ij})}{\sum_k exp(e_{ik})}$$

- Step 4: Take a weighted sum of **values**

$$Output_i = \sum_j \alpha_{ij} v_j$$



Recipe for (Vectorized) Self-Attention in the Transformer Encoder

- Step 1: With embeddings stacked in X , calculate **queries**, **keys**, and **values**.

$$Q = XW^Q \quad K = XW^K \quad V = XW^V$$

- Step 2: Calculate attention scores between **query** and **keys**.

$$E = QK^T$$

- Step 3: Take the softmax to normalize attention scores.

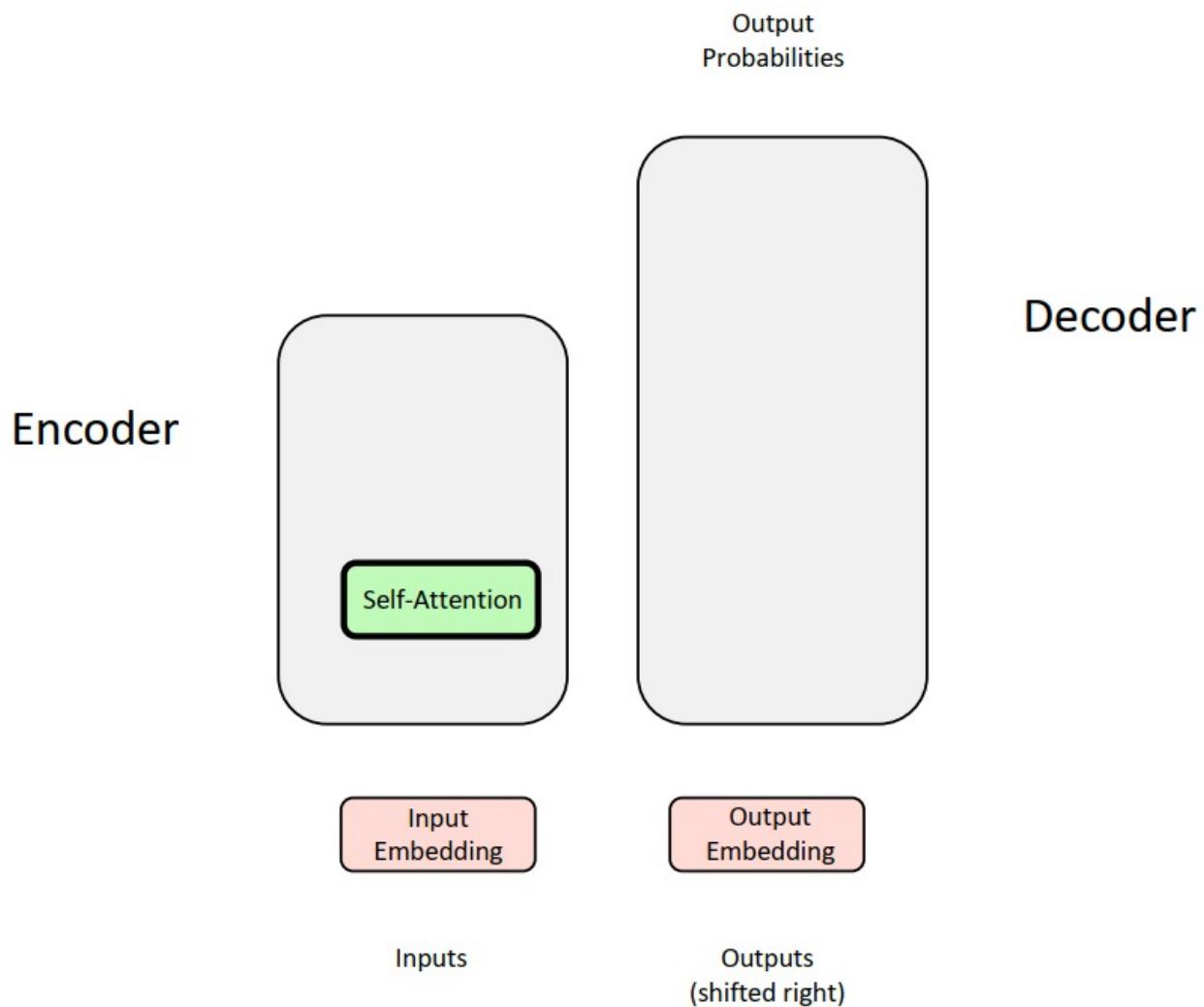
$$A = \text{softmax}(E)$$

- Step 4: Take a weighted sum of **values**

$$\text{Output} = AV$$

$$\text{Output} = \text{softmax}(QK^T)V$$

What We Have So Far: (Encoder) Self-Attention!

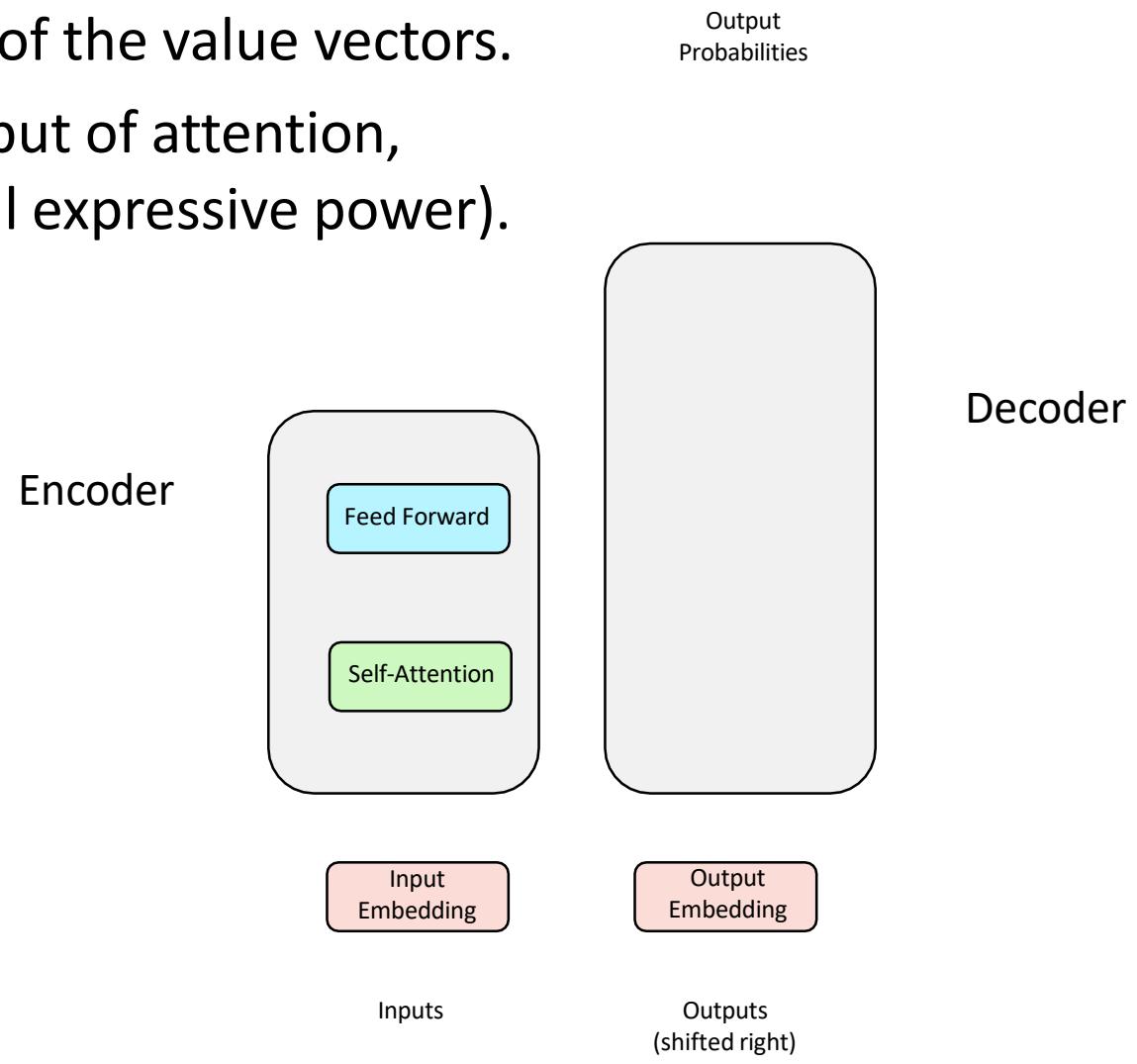
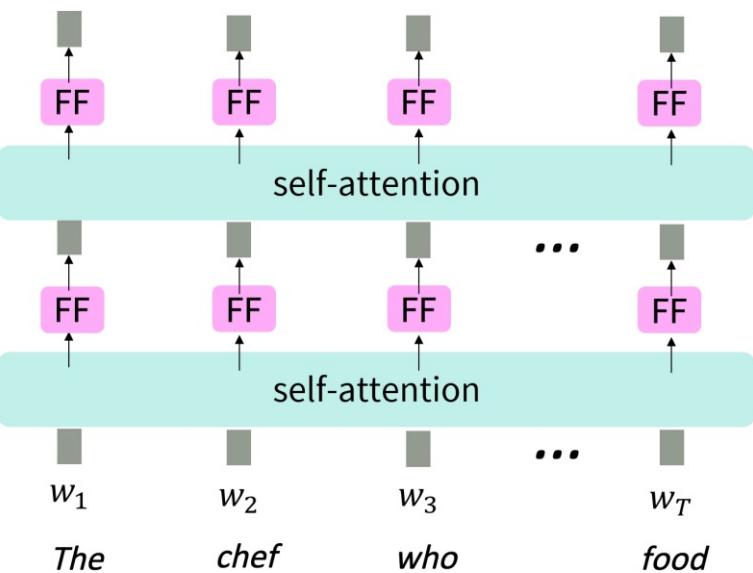


But attention isn't quite all you need!

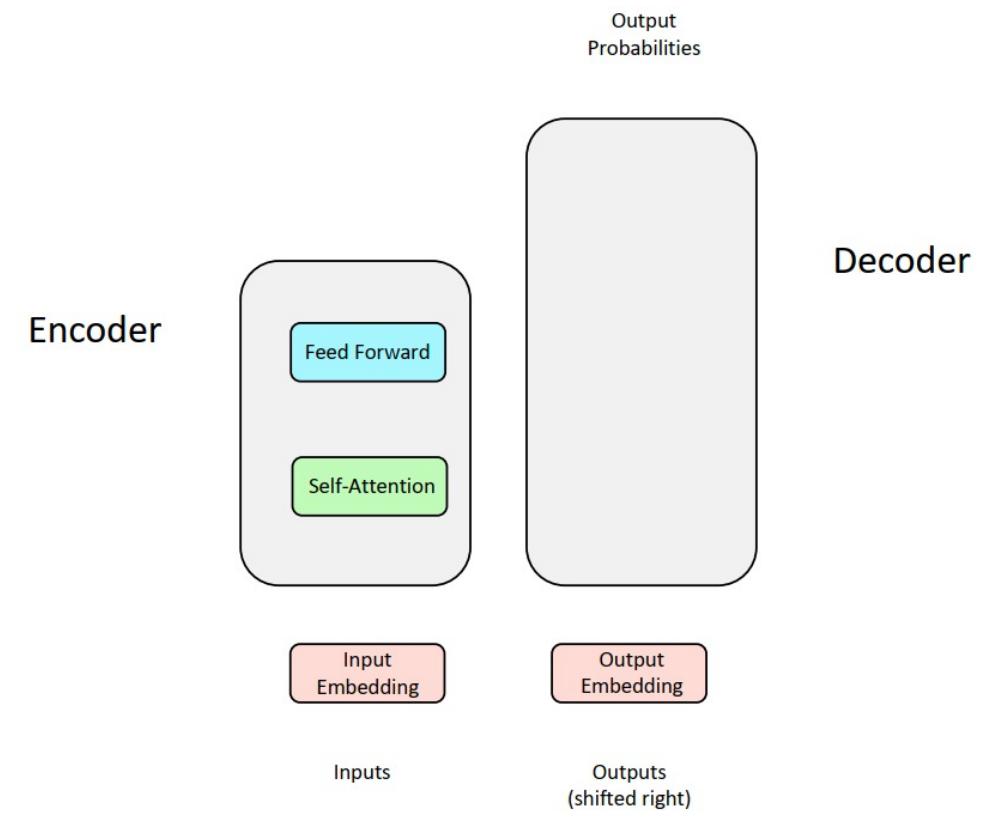
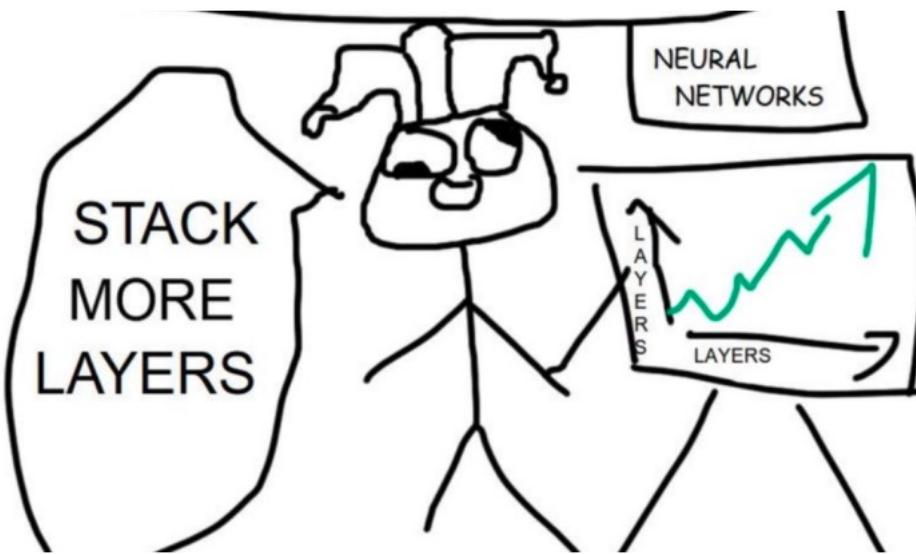
- **Problem:** Since there are no element-wise non-linearities, self-attention is simply performing a re-averaging of the value vectors.
- **Easy fix:** Apply a feedforward layer to the output of attention, providing non-linear activation (and additional expressive power).

Equation for Feed Forward Layer

$$\begin{aligned} m_i &= \text{MLP}(\text{output}_i) \\ &= W_2 * \text{ReLU}(W_1 \times \text{output}_i + b_1) + b_2 \end{aligned}$$



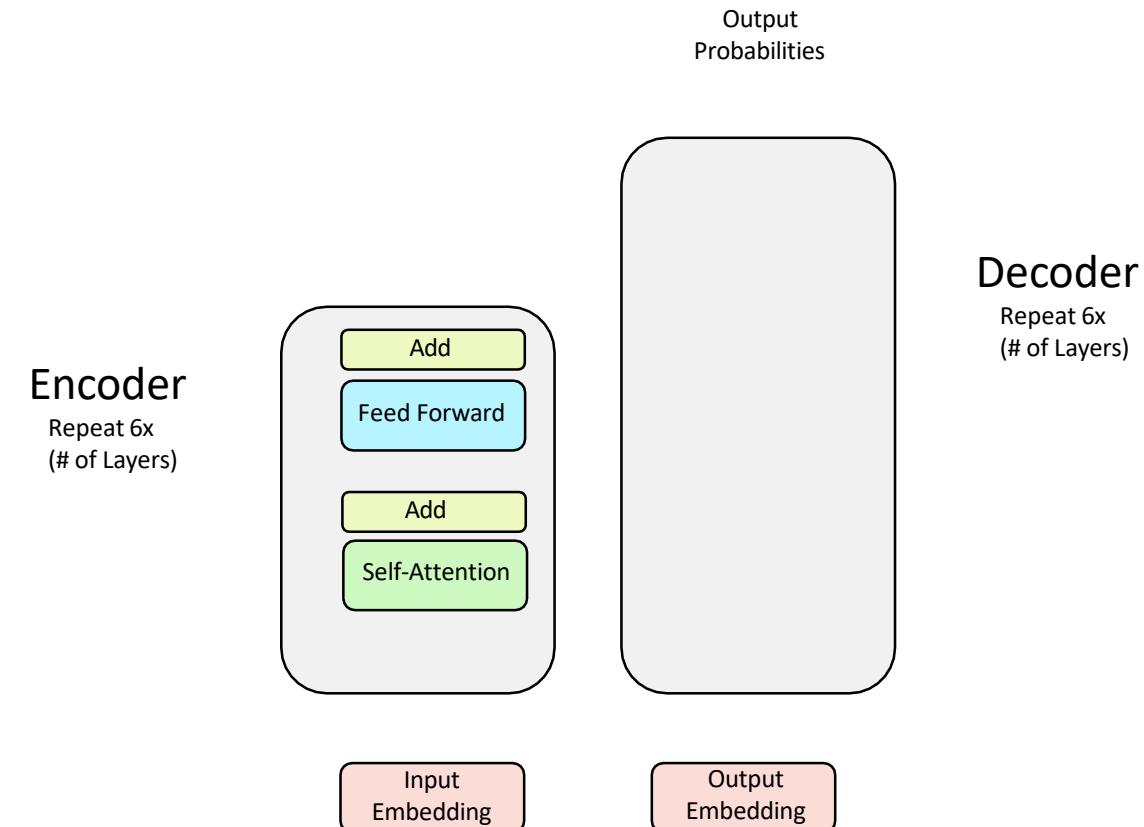
But how do we make this work for deep networks?



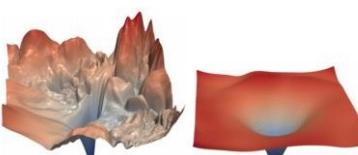
- Training Trick #1: Residual Connections
- Training Trick #2: LayerNorm
- Training Trick #3: Scaled Dot Product Attention

Training Trick #1: Residual Connections [He et al., 2016]

- Residual connections are a simple but powerful technique from computer vision.
- Deep networks are surprisingly bad at learning the identity function!
- Therefore, directly passing "raw" embeddings to the next layer can actually be very helpful!
$$x_\ell = F(x_{\ell-1}) + x_{\ell-1}$$
- This prevents the network from "forgetting" or distorting important information as it is processed by many layers.



Residual connections are also thought to smooth the loss landscape and make training easier!



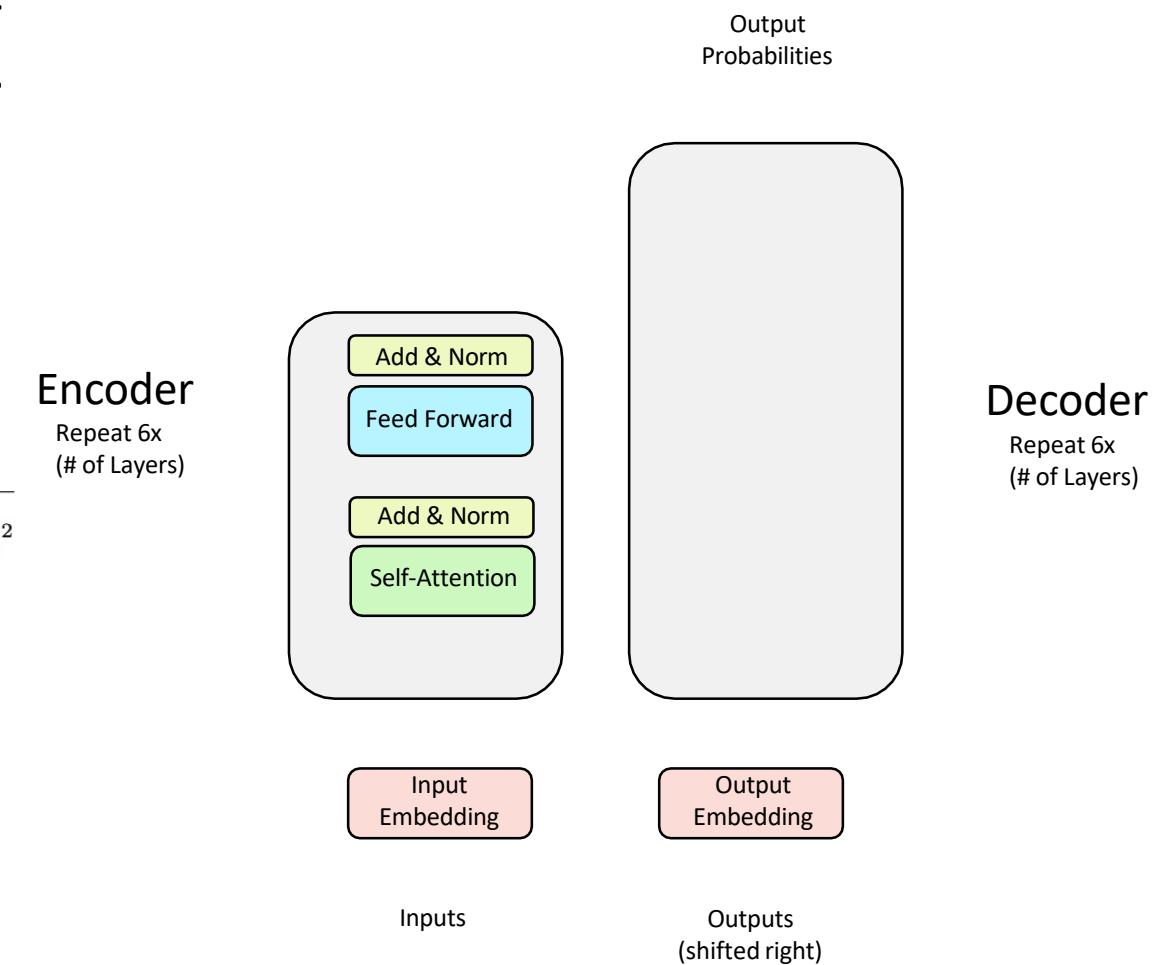
[no residuals] [residuals]
[Loss landscape visualization, Li et al., 2018, on a ResNet]

Training Trick #2: Layer Normalization [Ba et al., 2016]

- **Problem:** Difficult to train the parameters of a given layer because its input from the layer beneath keeps shifting.
- **Solution:** Reduce uninformative variation by **normalizing** to zero mean and standard deviation of one within each **layer**.

$$\text{Mean: } \mu^l = \frac{1}{H} \sum_{i=1}^H a_i^l \quad \text{Standard Deviation: } \sigma^l = \sqrt{\frac{1}{H} \sum_{i=1}^H (a_i^l - \mu^l)^2}$$

$$x'^{\ell'} = \frac{x^{\ell'} - \mu^{\ell'}}{\sigma^{\ell'} + \epsilon}$$

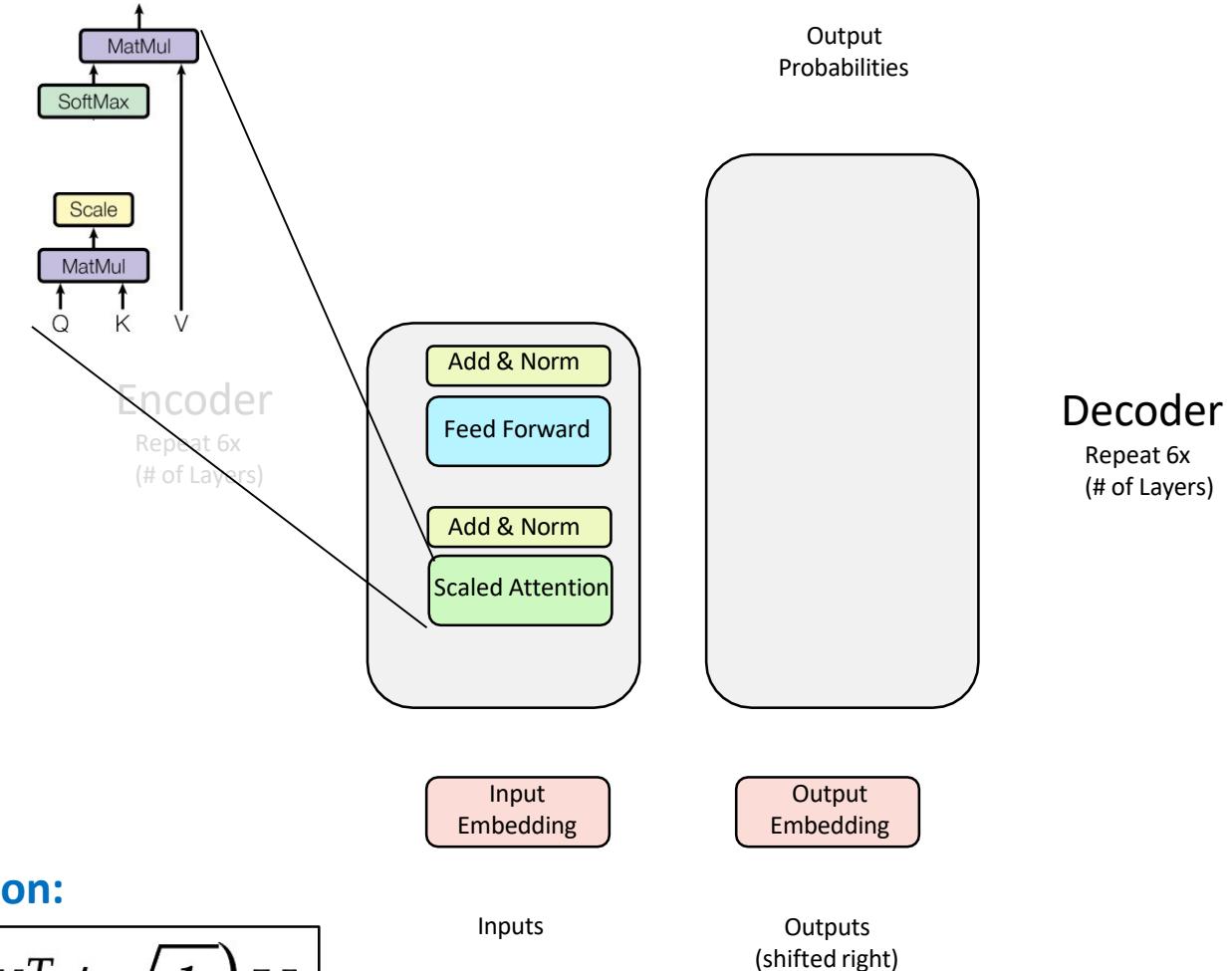


Training Trick #3: Scaled Dot Product Attention

- After LayerNorm, the mean and variance of vector elements is 0 and 1, respectively. (Yay!)
- However, the dot product still tends to take on extreme values, as its variance scales with dimensionality d_k

Quick Statistics Review:

- Mean of sum = sum of means = $d_k * 0 = 0$
- Variance of sum = sum of variances = $d_k * 1 = d_k$
- To set the variance to 1, simply divide by $\sqrt{d_k}$!



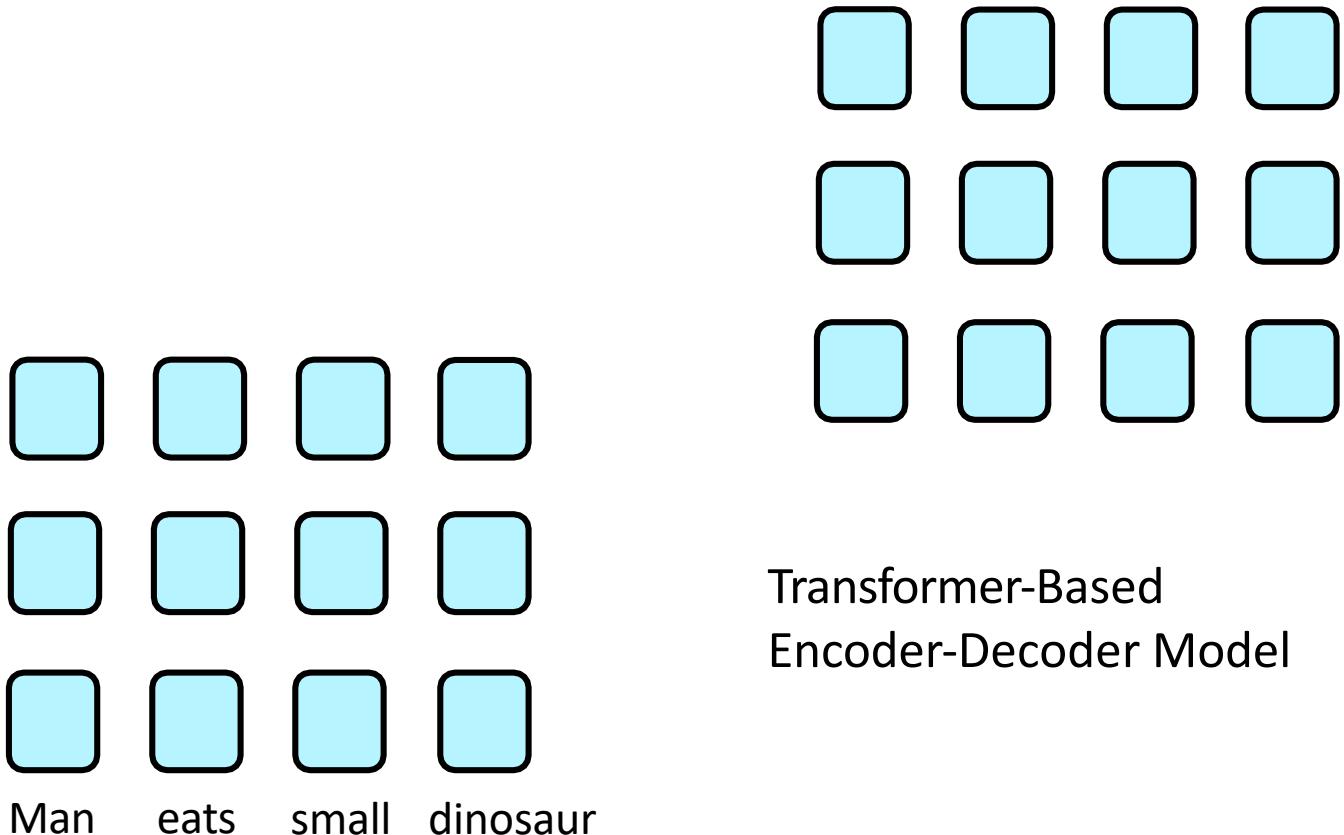
Updated Self-Attention Equation:

$$Output = \text{softmax}\left(QK^T / \sqrt{d_k}\right)V$$

Major issue!

- We're almost done with the Encoder, but we have a major problem! Has anyone spotted it?
- Consider this sentence:
 - "Man eats small dinosaur."

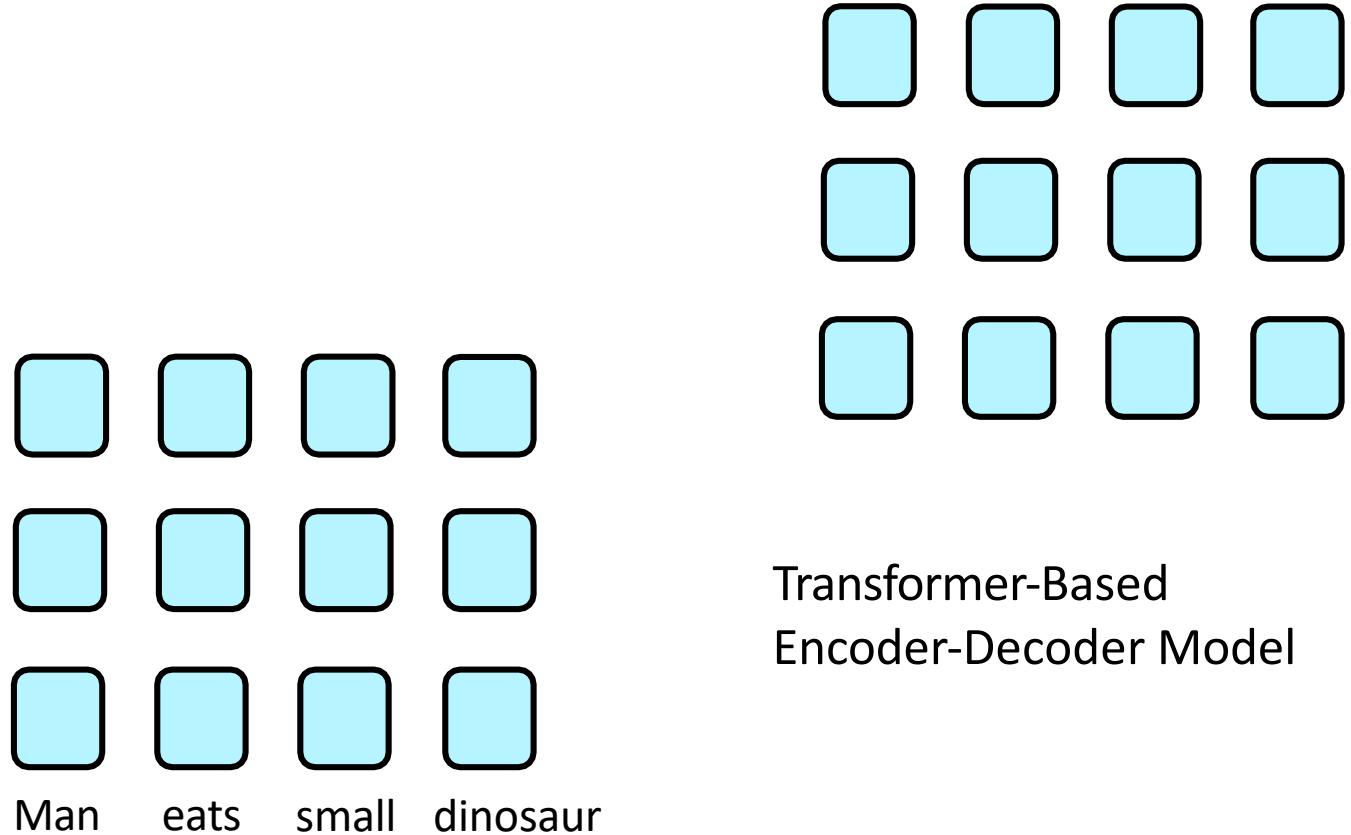
$$\text{Output} = \text{softmax}\left(QK^T / \sqrt{d_k}\right)V$$



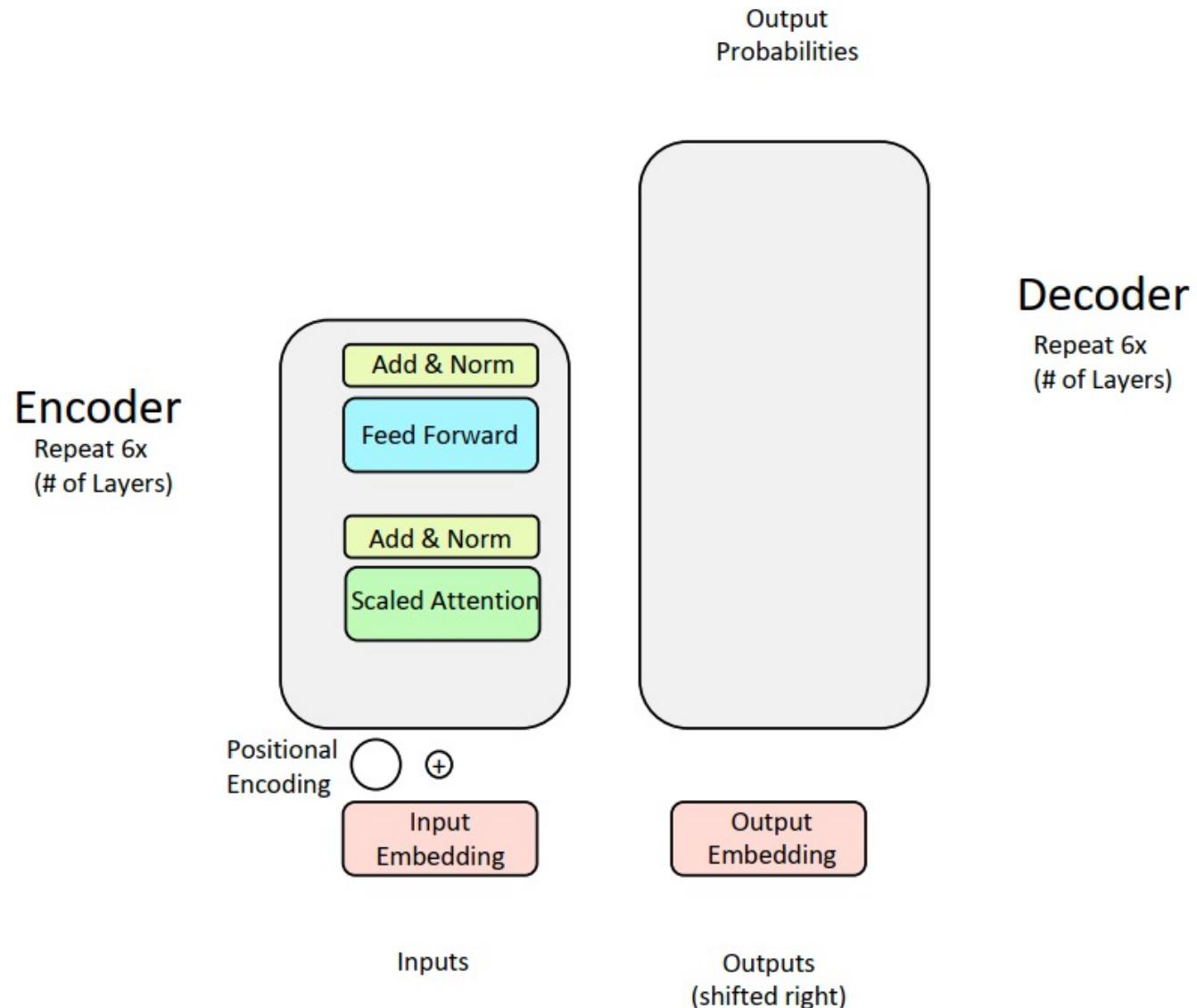
Major issue!

- We're almost done with the Encoder, but we have a major problem! Has anyone spotted it?
- Consider this sentence:
 - "Man eats small dinosaur."
- Wait a minute, order doesn't impact the network at all!
- This seems wrong given that word order does have meaning in many languages, including English!

$$\text{Output} = \text{softmax}\left(QK^T / \sqrt{d_k}\right)V$$



Solution: Inject Order Information through Positional Encodings!



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.

$$v_i = \tilde{v}_i + p_i$$

$$q_i = \tilde{q}_i + p_i$$

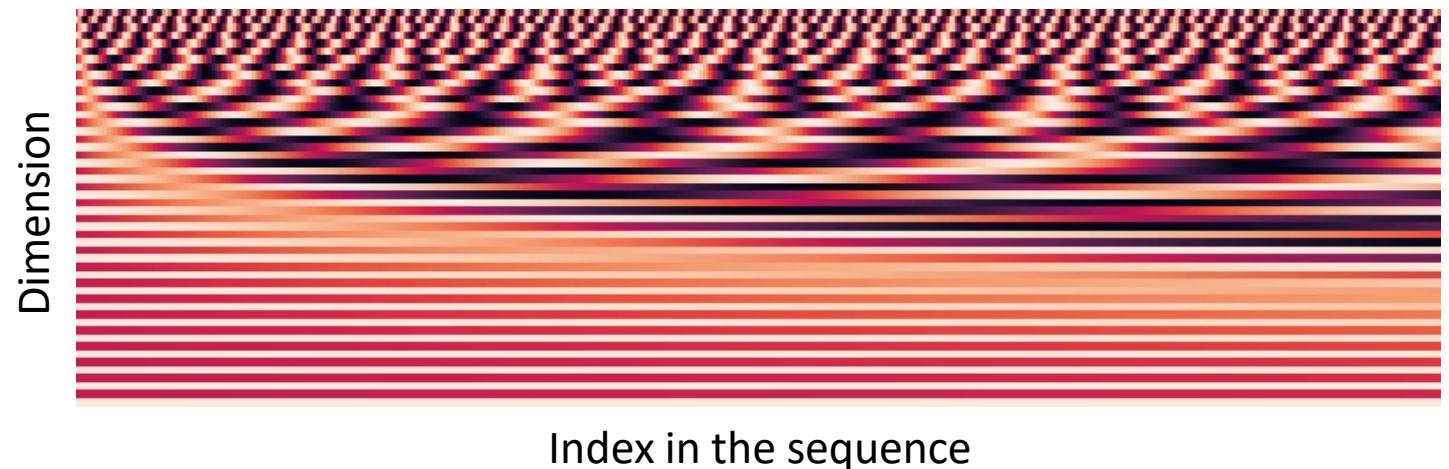
$$k_i = \tilde{k}_i + p_i$$

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:

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



- 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

- **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 [\[Wang et al., 2019\]](#)

Common, modern position embeddings - RoPE

High level thought process: a *relative* position embedding should be some $f(x,i)$ s.t.

$$\langle f(x,i) f(y,j) \rangle = g(x,y, i-j)$$

That is, the attention function *only* gets to depend on the relative position (i-j). How do existing embeddings not fulfill this goal?

- **Sine:** Has various cross-terms that are not relative
- **Absolute:**

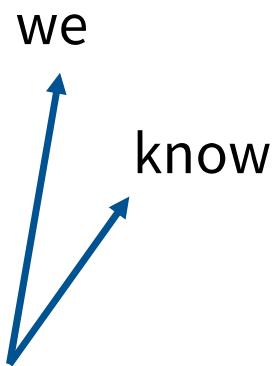
$$e_{ij} = \frac{x_i W^Q (x_j W^K + a_{ij}^K)^T}{\sqrt{d_z}}$$

is not an inner product

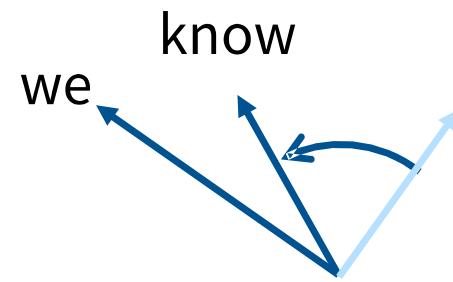
RoPE – Embedding via rotation

How can we solve this problem?

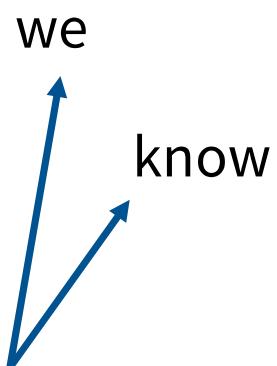
- We want our embeddings to be invariant to absolute position
- We know that inner products are invariant to arbitrary rotation.



Position independent
embedding

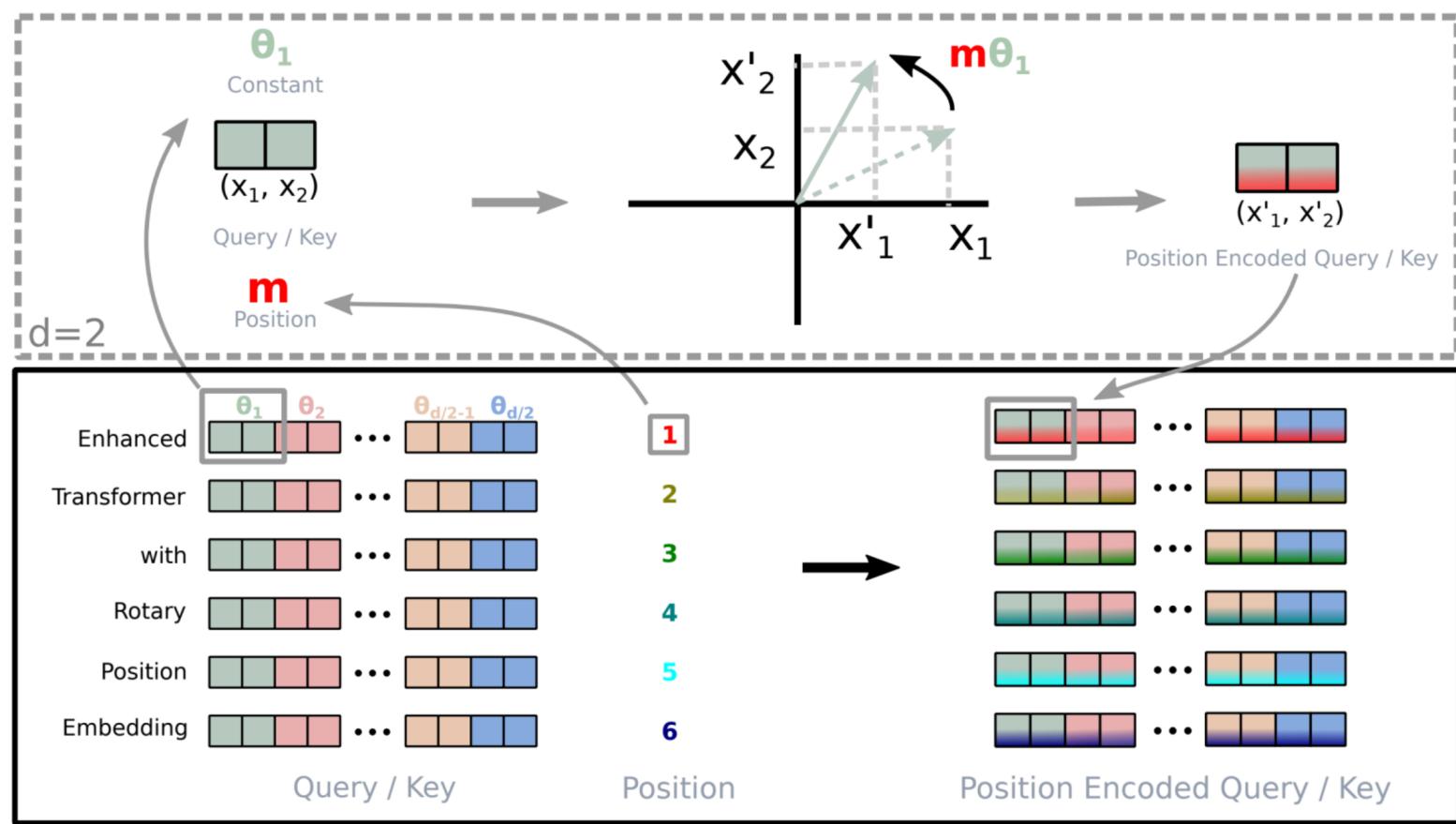


Embedding
“of course we know”
Rotate by ‘2 positions’



Embedding
“we know that”
Rotate by ‘0 positions’

RoPE – From 2 to many dimensions



[Su et al 2021]

Just pair up the coordinates and rotate them in 2d (motivation: complex numbers)

Properties

- Long-term Decay
 - Same with Transformer, choose $\theta_i = 10000^{-\frac{2i}{d}}$.
 - When relative distance increases, inner product will decrease
 - This means that tokens that are farther apart have lower similarity.
- RoPE with Linear Attention
 - By leveraging the property that the **norm remains unchanged after rotating the hidden states**, RoPE can be combined with linear attention.
 - In other words, this means RoPE can be integrated with advancements from Efficient Transformer research.
 - Recall: T5-bias requires the full attention matrix, and therefore cannot be combined with linear attention.

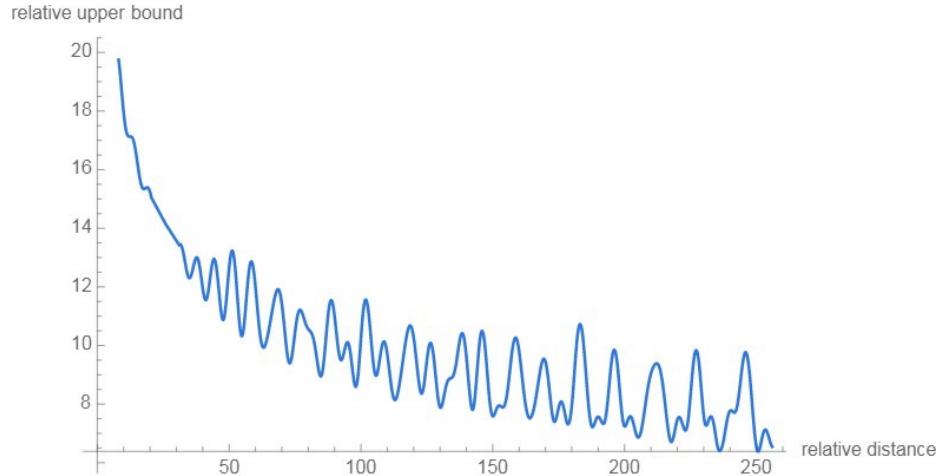
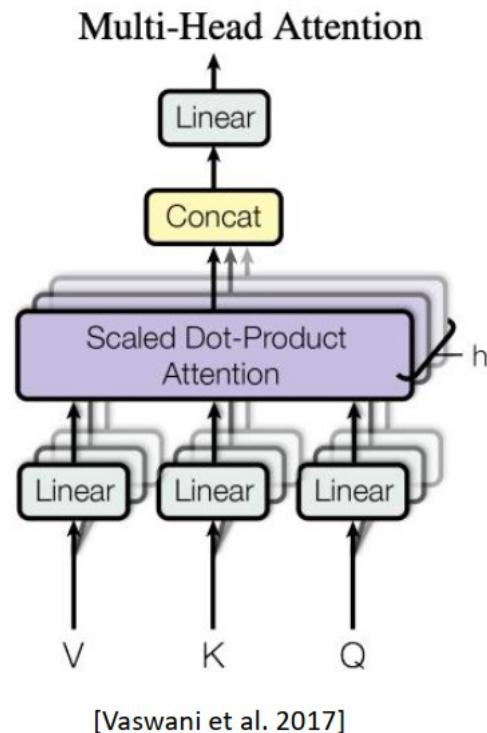


Figure 2: Long-term decay of RoPE.

Multi-Headed Self-Attention: k heads are better than 1!

- High-Level Idea: Let's perform self-attention multiple times in parallel and combine the results



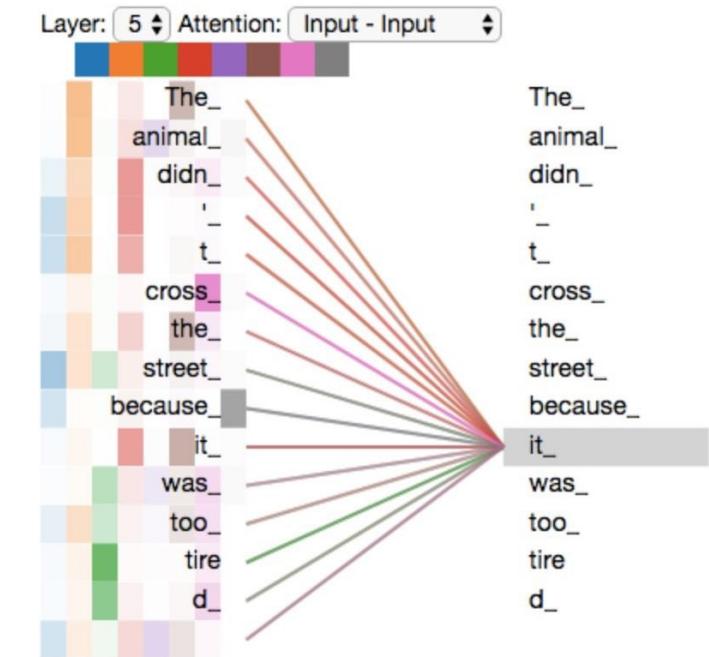
[Vaswani et al. 2017]



Wizards of the Coast, Artist: Todd Lockwood

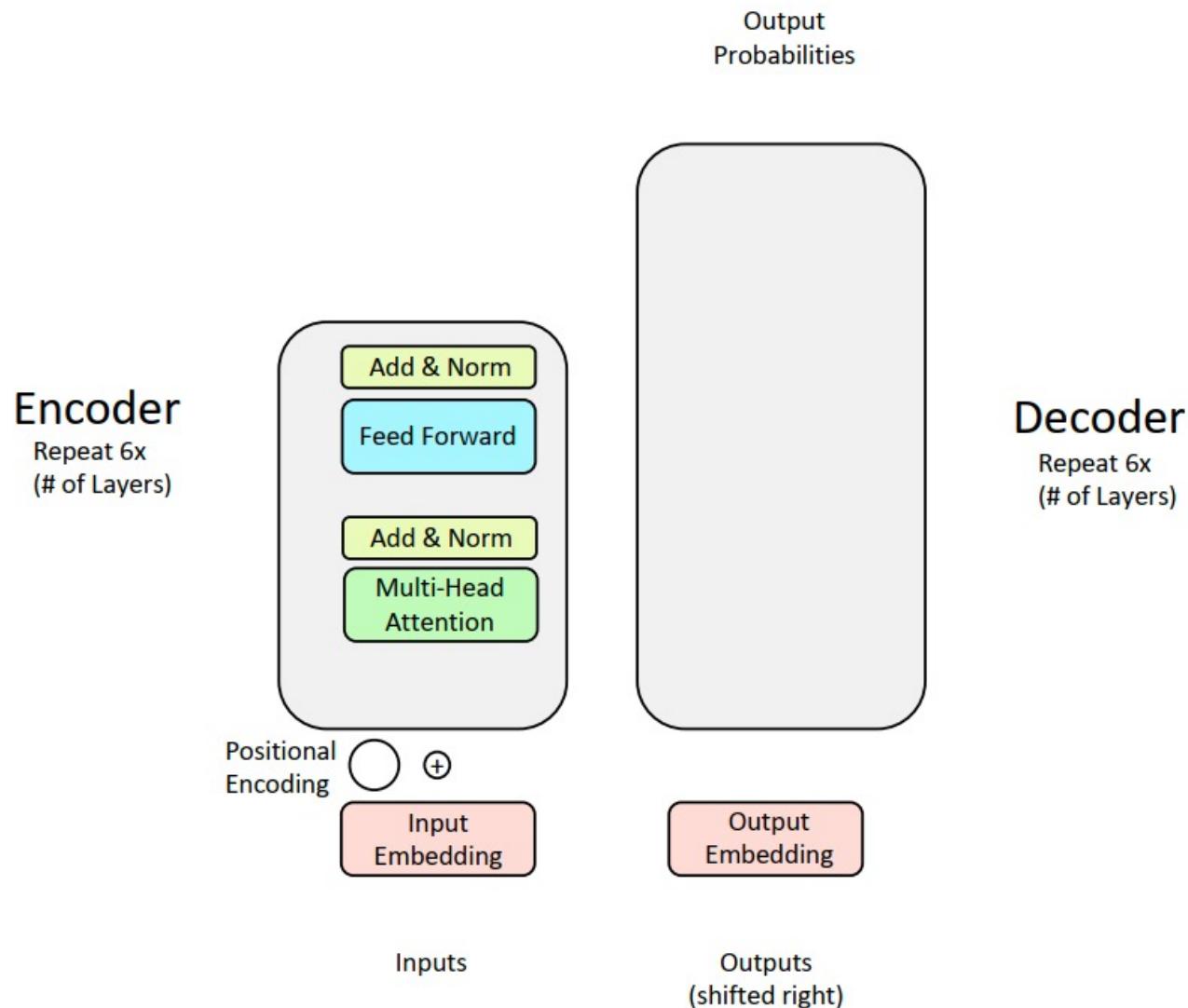
The Transformer Encoder: Multi-headed Self-Attention

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



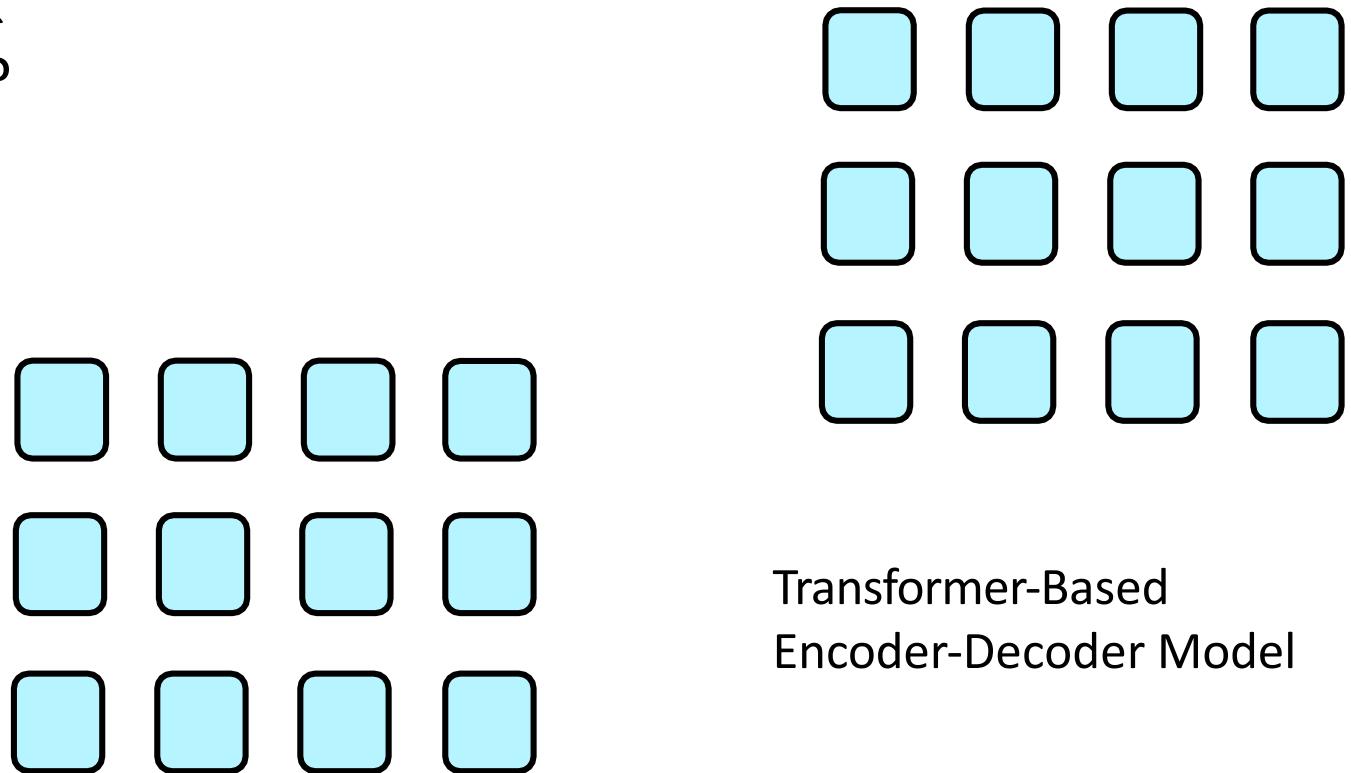
Credit to <https://jalammar.github.io/illustrated-transformer/>

Yay, we've completed the Encoder! Time for the Decoder...



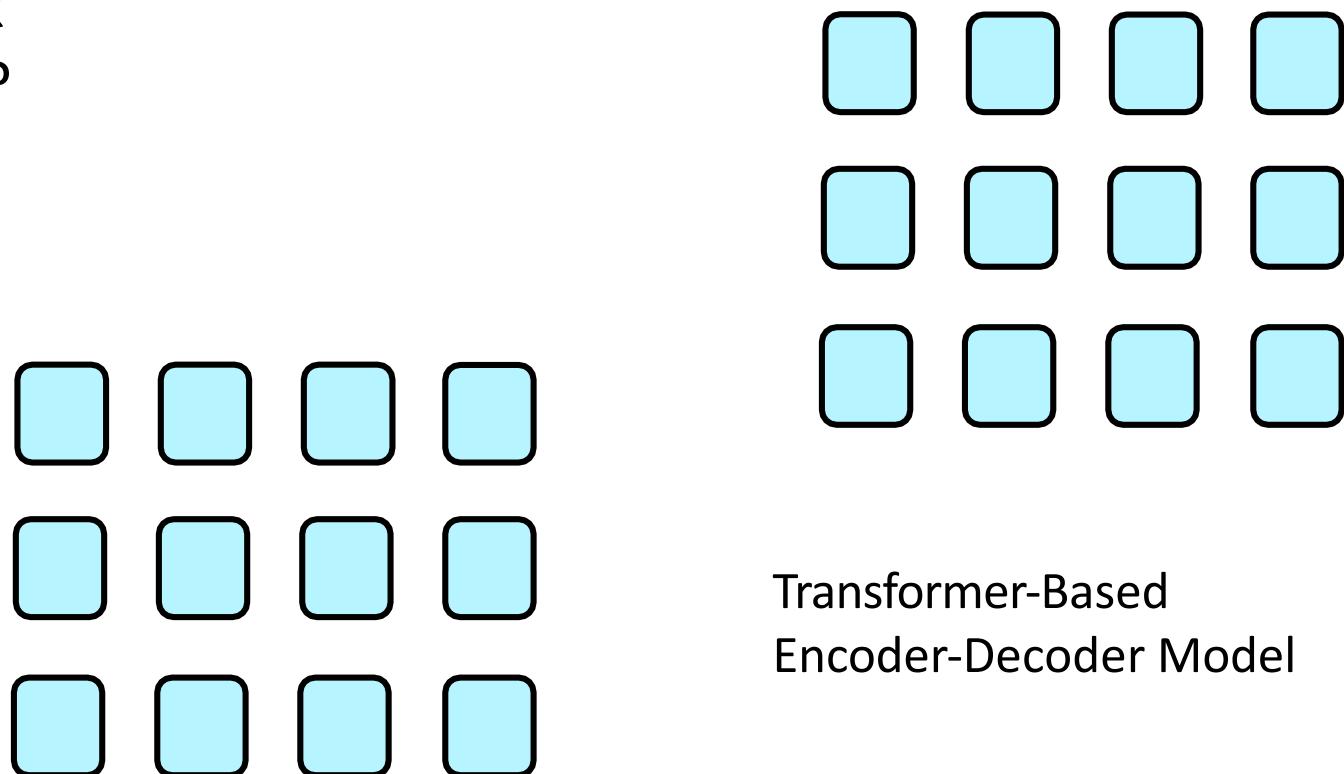
Decoder: Masked Multi-Head Self-Attention

- **Problem:** How do we keep the decoder from cheating? If we have a language modeling objective, can't the network just look ahead and "see" the answer?



Decoder: Masked Multi-Head Self-Attention

- **Problem:** How do we keep the decoder from "cheating"? If we have a language modeling objective, can't the network just look ahead and "see" the answer?
- **Solution:** Masked Multi-Head Attention. At a high-level, we hide (mask) information about future tokens from the model.

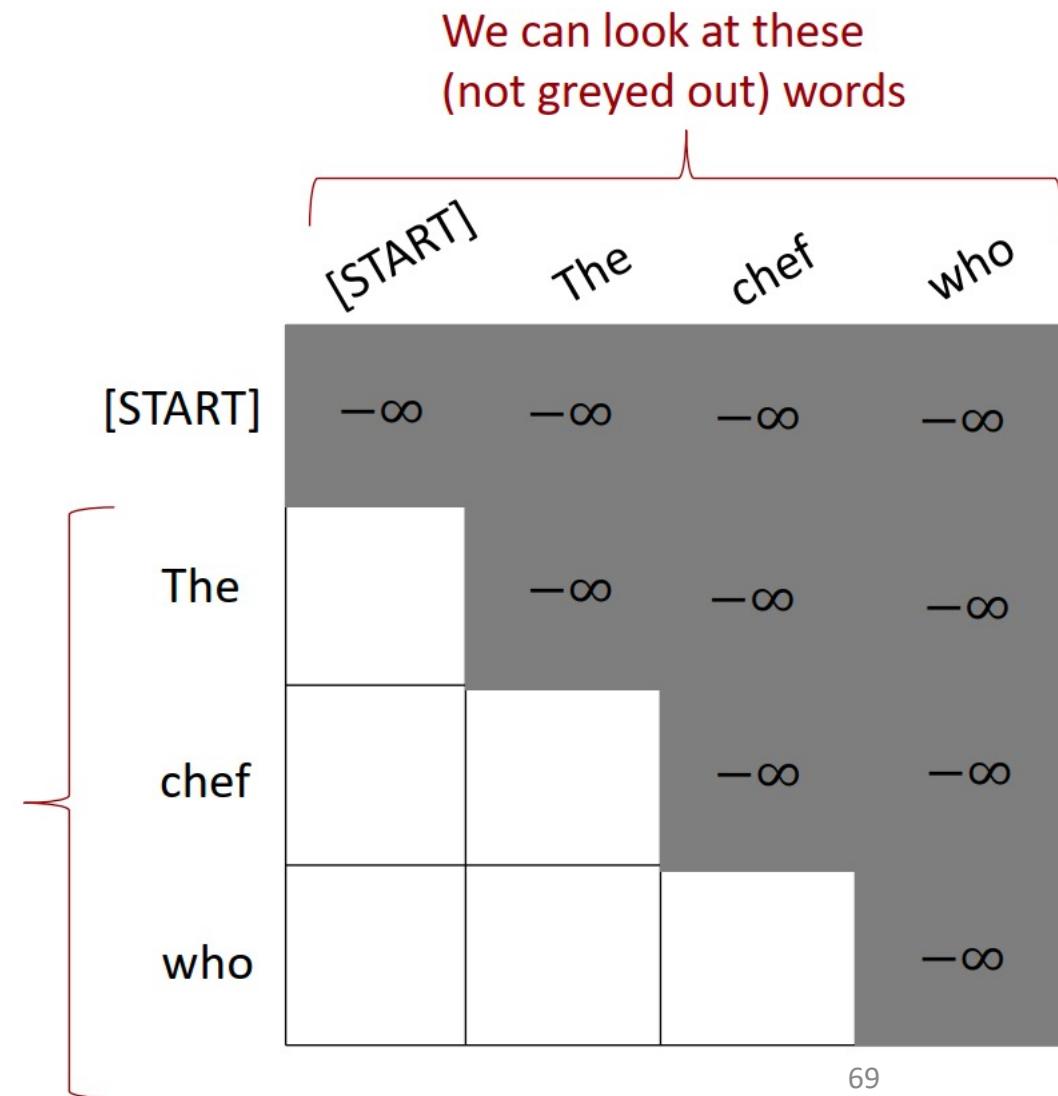


Masking the future in self-attention

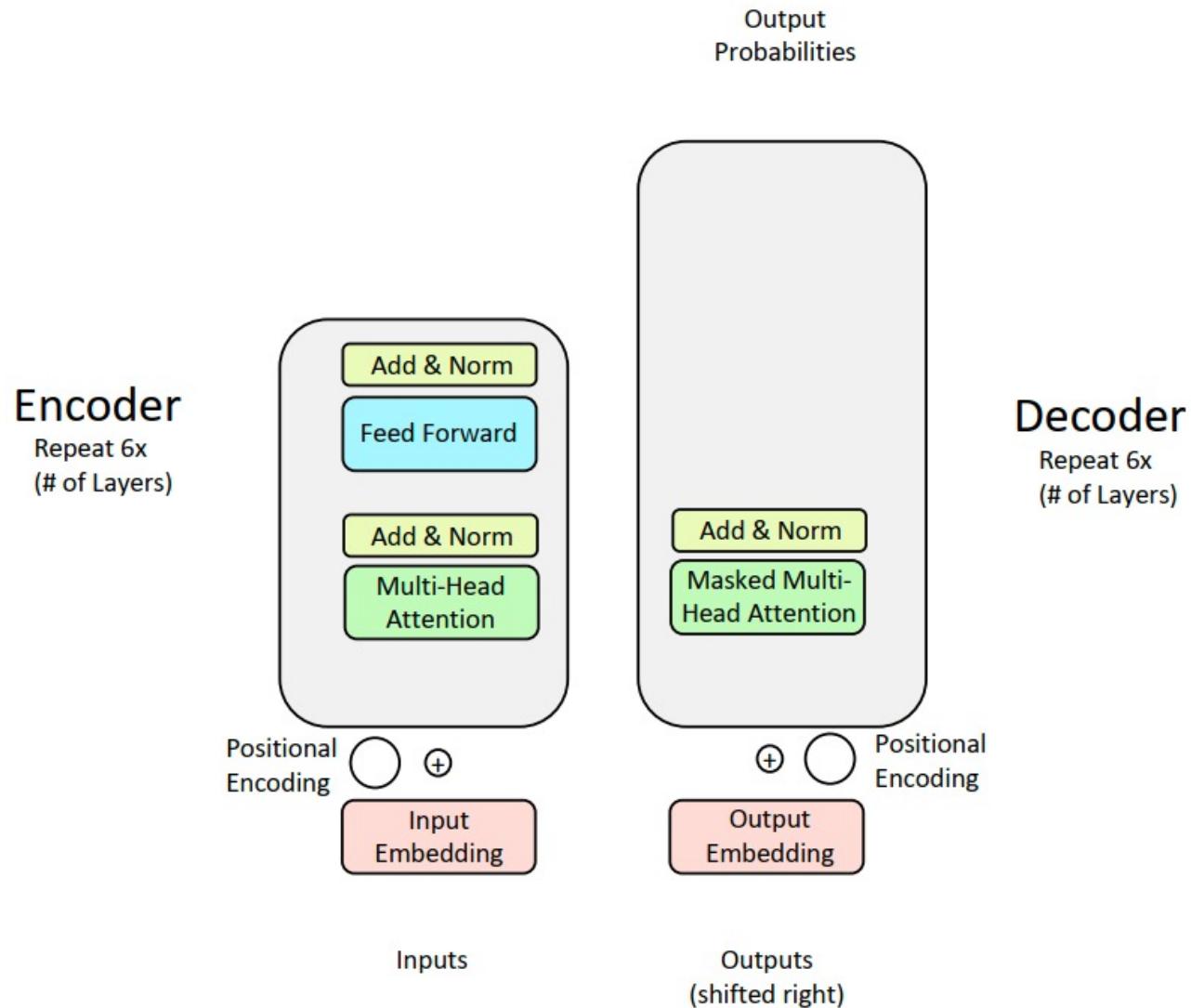
- To use self-attention in decoders, we need to ensure we can't peek at the future.
- To enable parallelization, we mask out attention to future words by setting attention scores to $-\infty$

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

For encoding
these words

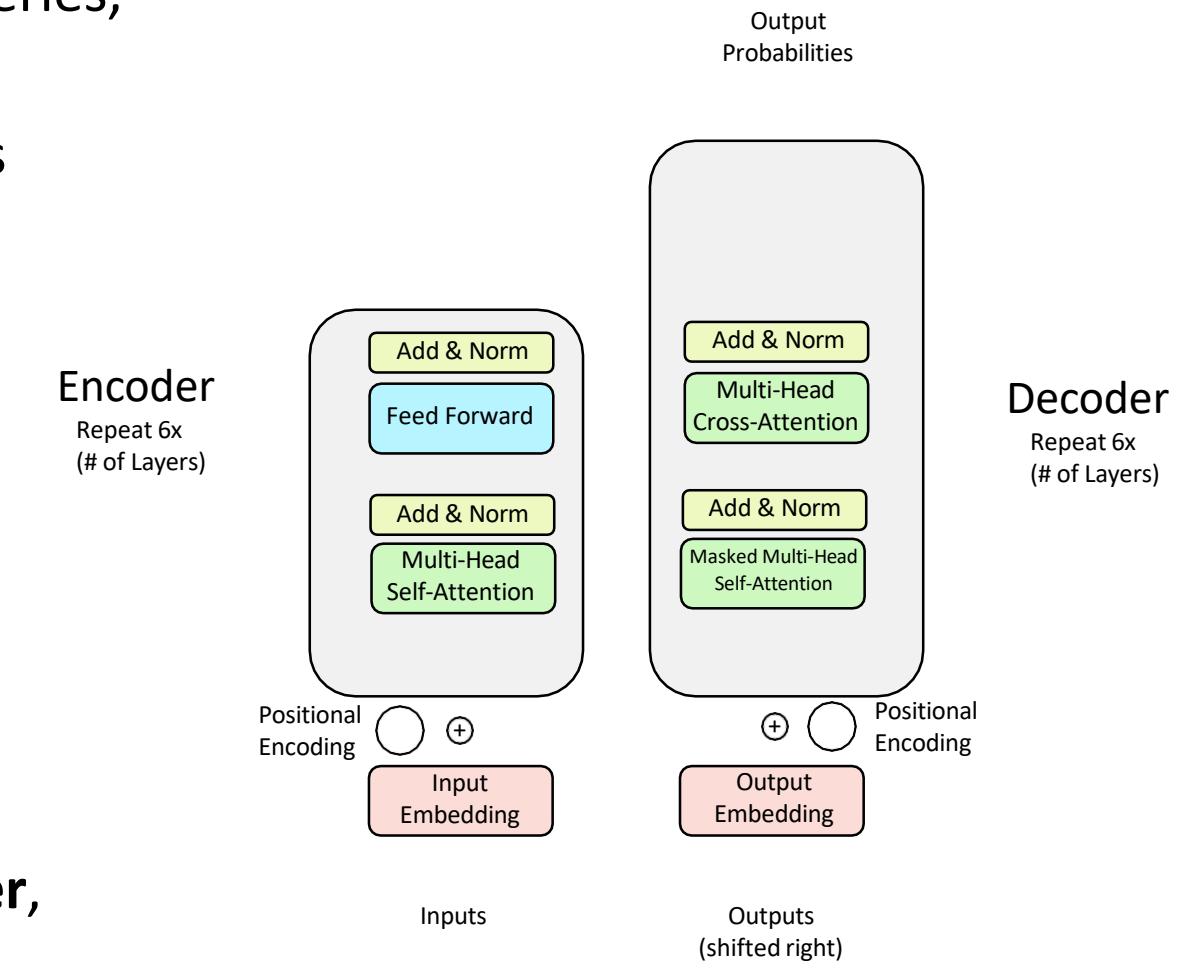


Decoder: Masked Multi-Headed Self-Attention



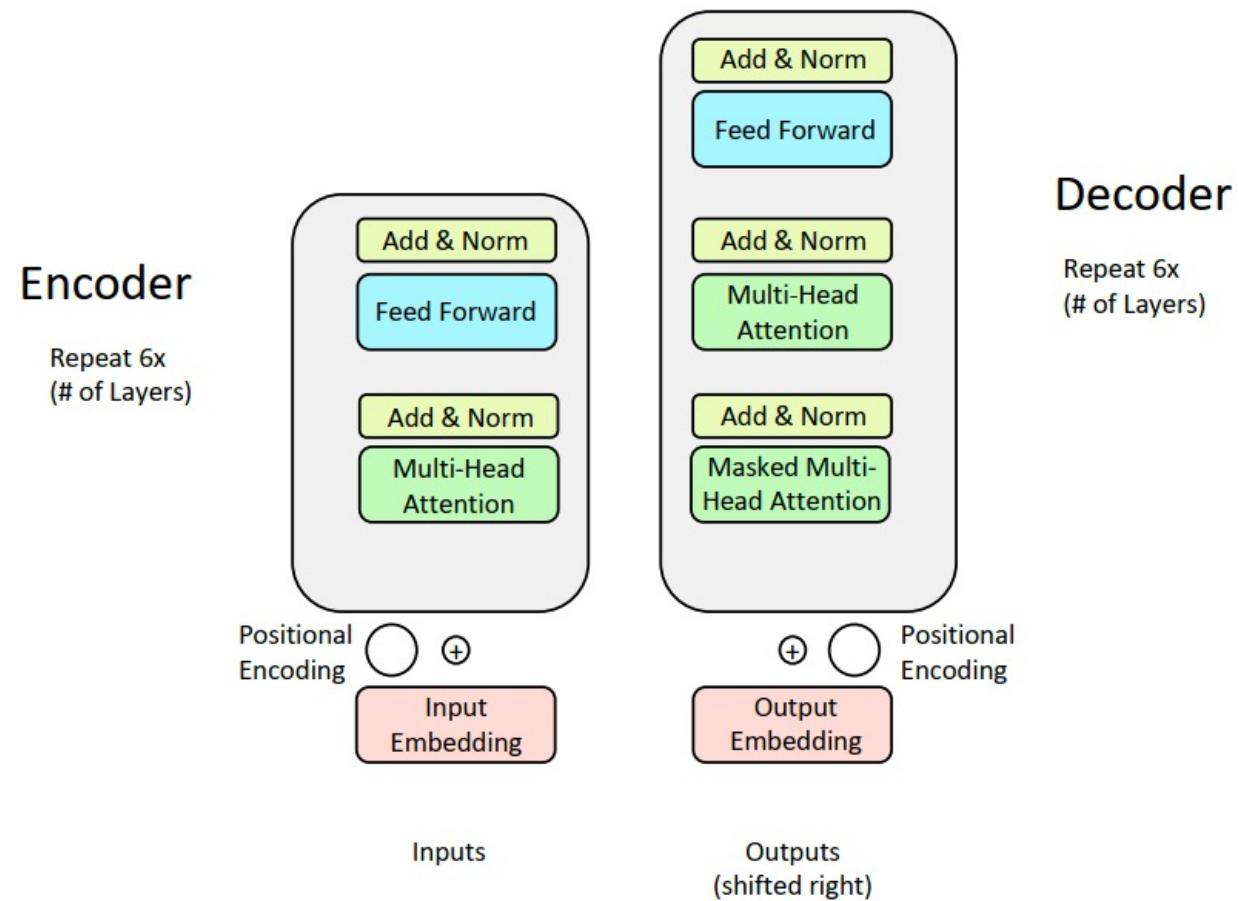
Encoder-Decoder Attention

- 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 encoder.
- 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**, $q_i = Qz_i$.



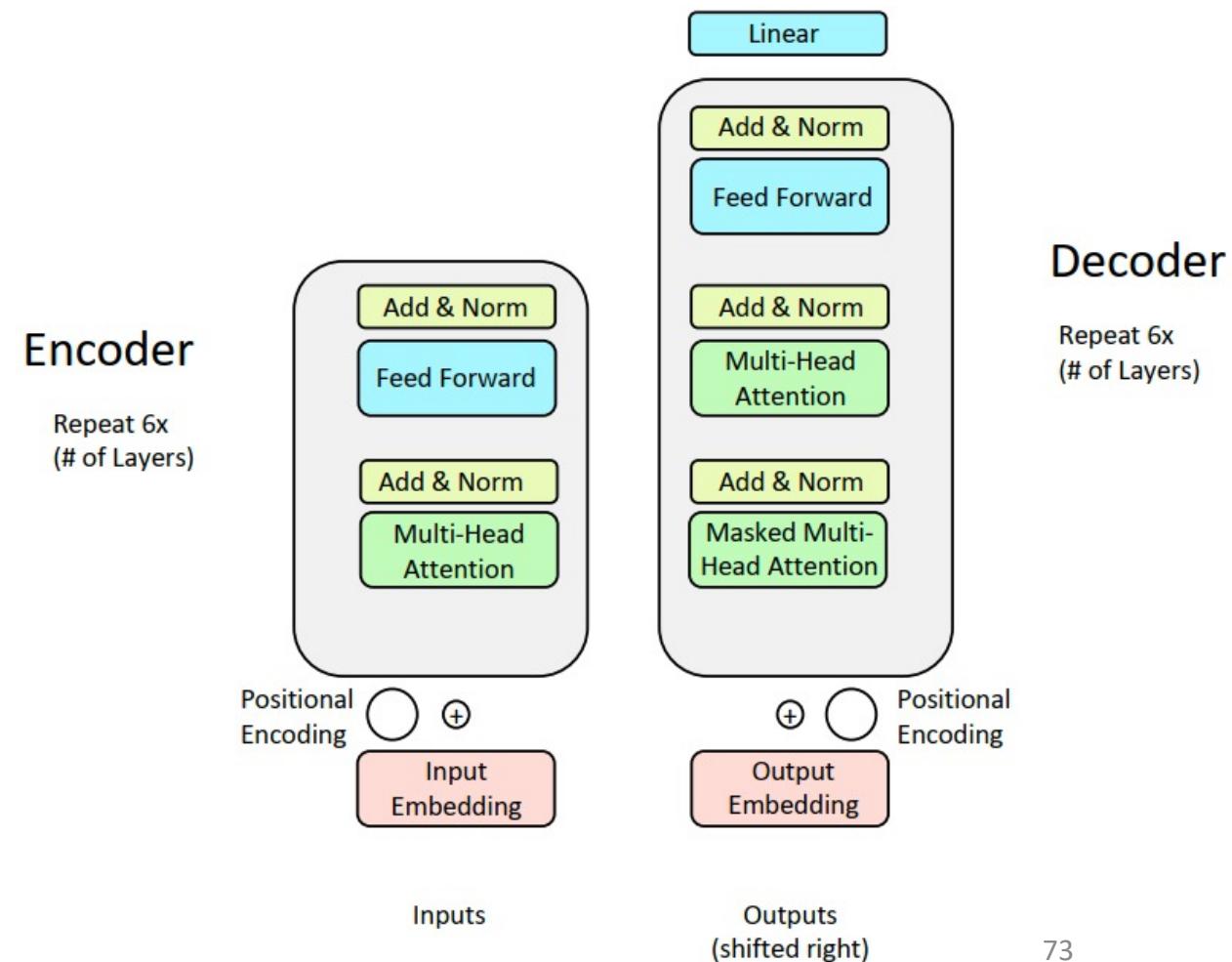
Decoder: Finishing touches!

- Add a feed forward layer (with residual connections and layer norm)



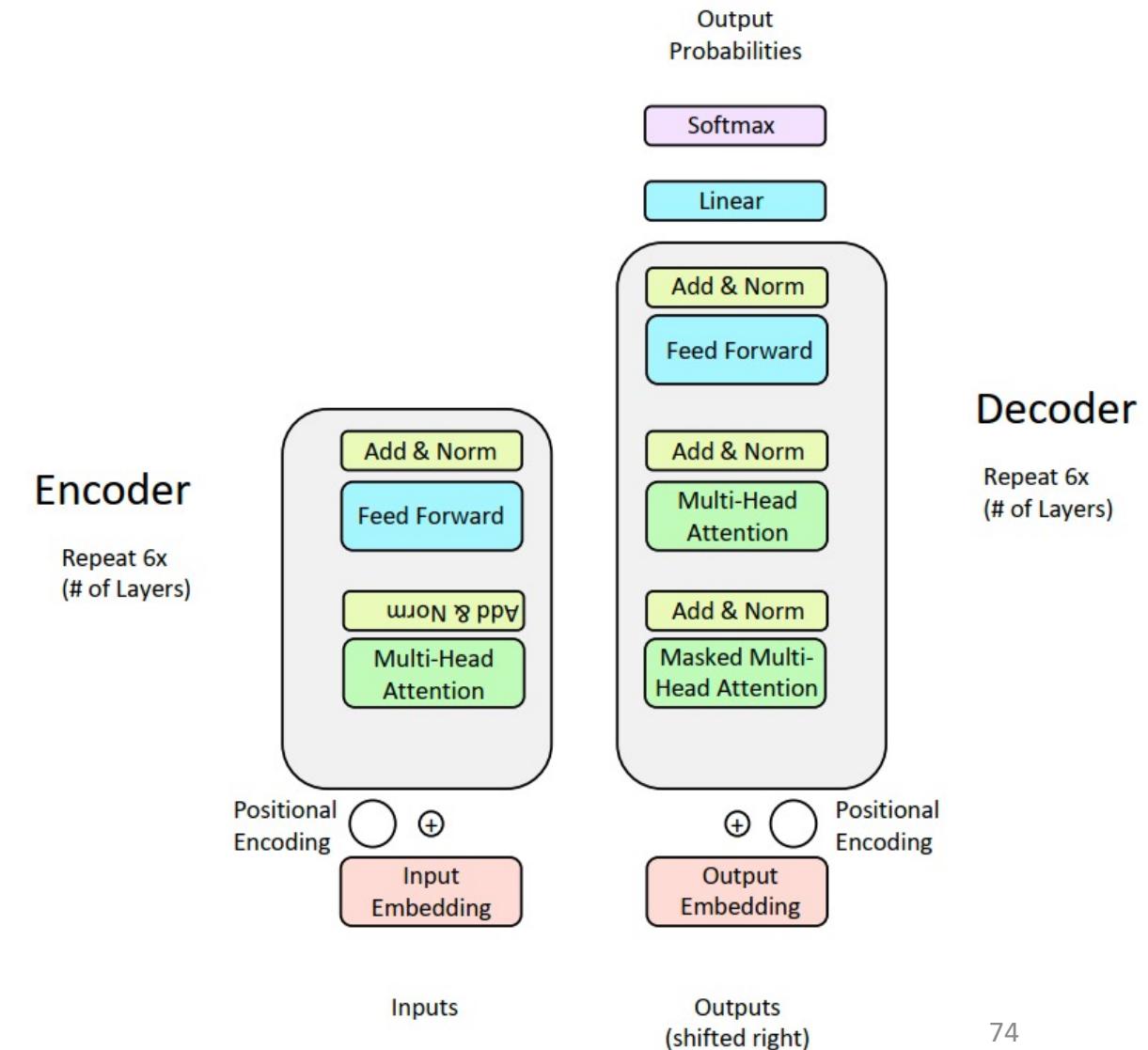
Decoder: Finishing touches!

- Add a feed forward layer (with residual connections and layer norm)
- Add a final linear layer to project the embeddings into a much longer vector of length vocab size (logits)

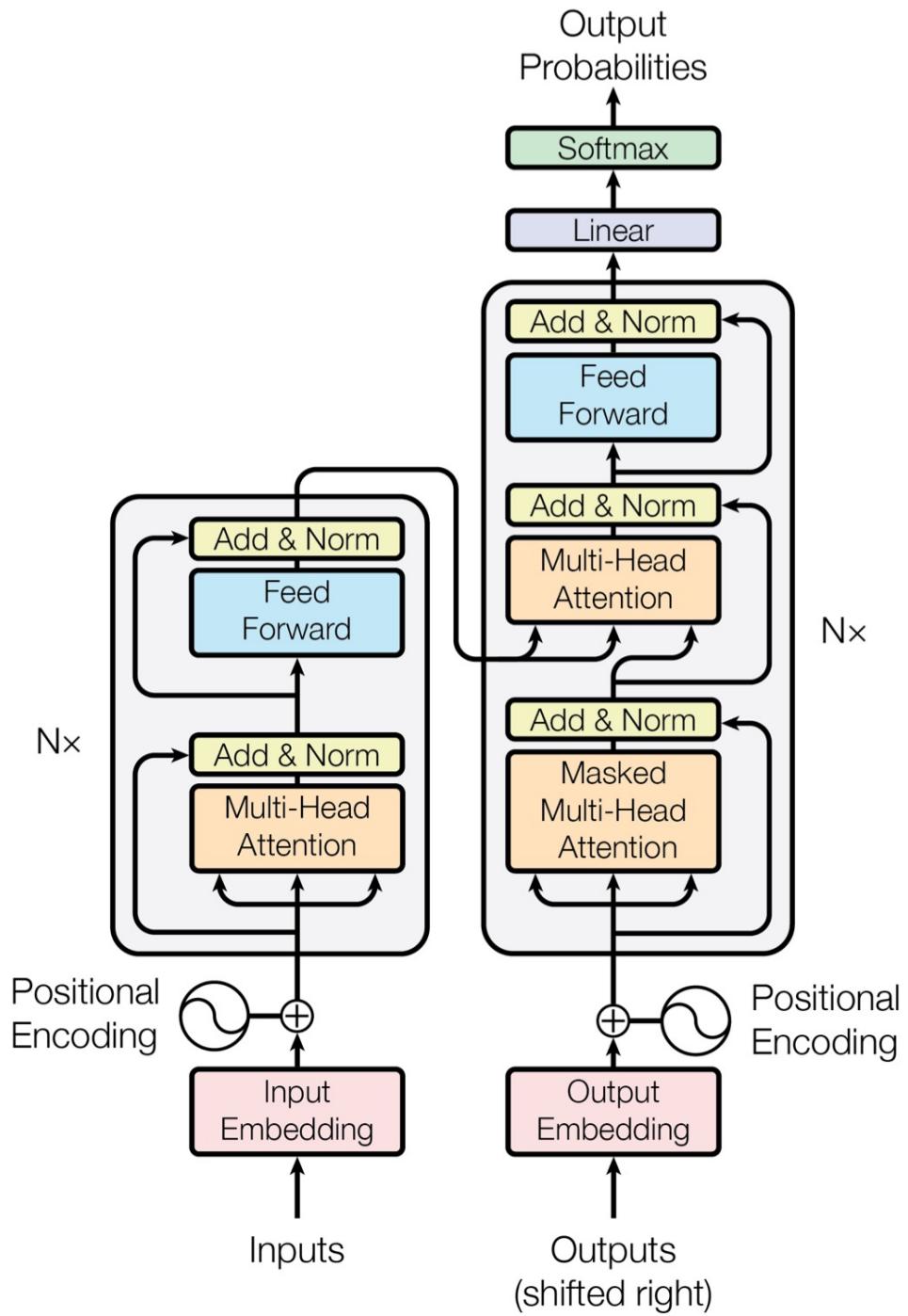


Decoder: Finishing touches!

- Add a feed forward layer (with residual connections and layer norm)
- Add a final linear layer to project the embeddings into a much longer vector of length vocab size (logits)
- Add a final softmax to generate an probability distribution of possible next words!



Recap of Transformer Architecture



Contents

- Transformers
 - Impact of Transformers on NLP (and ML more broadly)
 - From Recurrence (RNNs) to Attention-Based NLP Models
 - Understanding the Transformer Model
 - Drawbacks and Variants of Transformers
- Pretraining Language Models(PLMs)
 - Subword modeling
 - Motivating model pretraining from word embeddings
 - Model pretraining three ways
 - Decoders
 - Encoders
 - Encoder-Decoders
 - Very large models and in-context learning

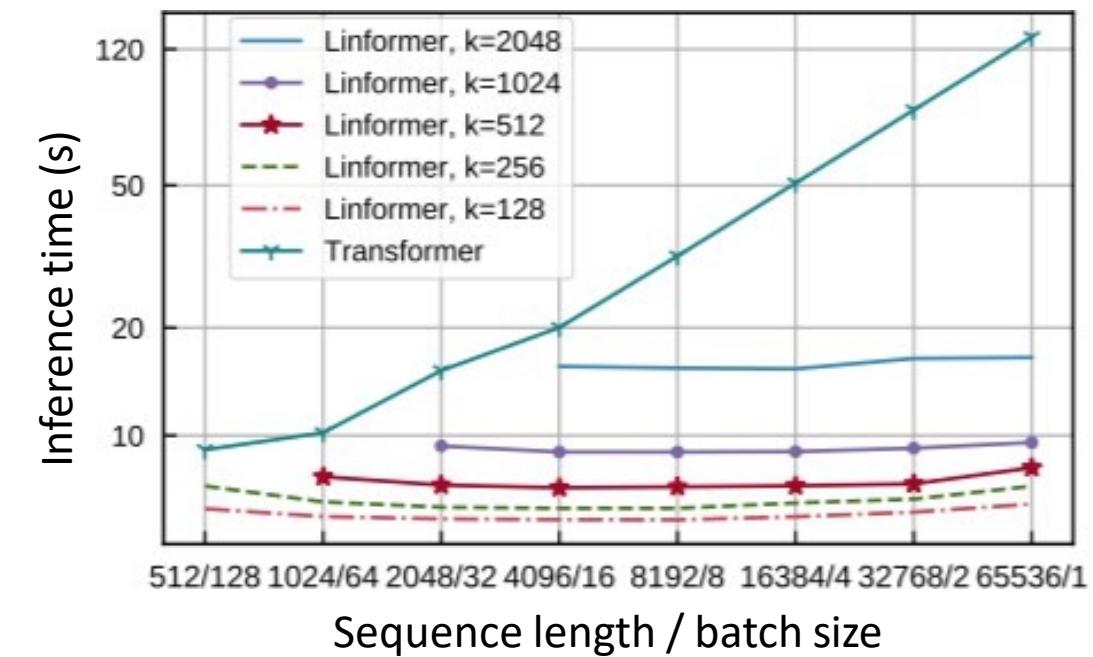
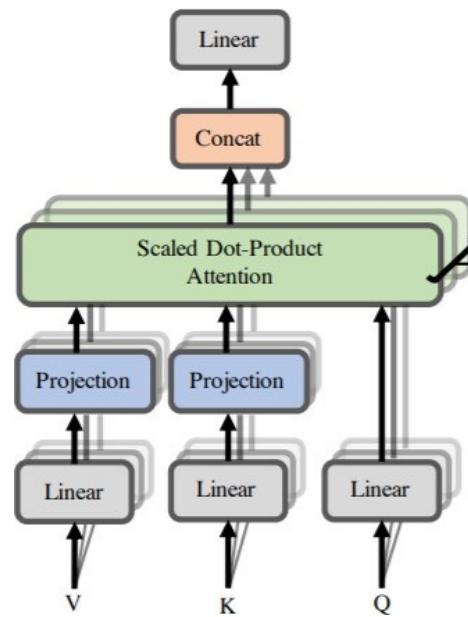
What would we like to fix about the Transformer?

- **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 [\[Wang et al., 2019\]](#)

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]

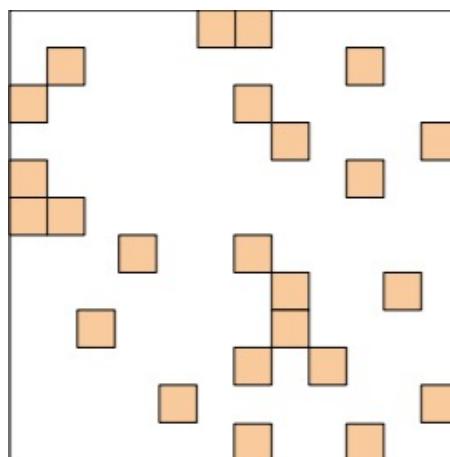
Key idea: map the sequence length dimension to a lower-dimensional space for values, keys



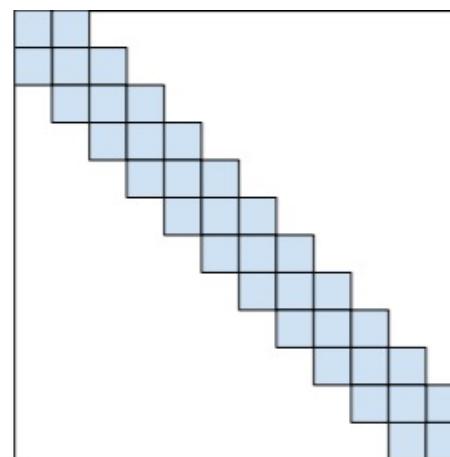
Recent work on improving on quadratic self-attention cost

- 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](#)]

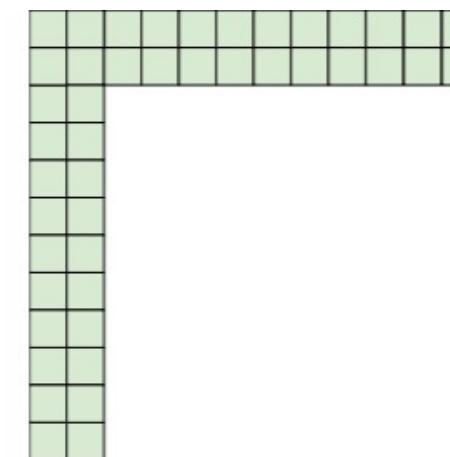
Key idea: replace all-pairs interactions with a family of other interactions, **like local windows, looking at everything, and random interactions.**



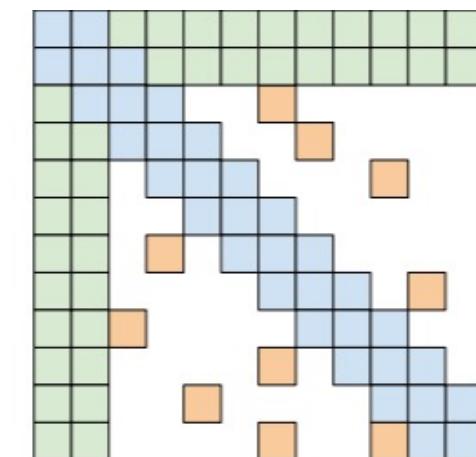
(a) Random attention



(b) Window attention



(c) Global Attention



(d) BIGBIRD

Thank you