

Foundations of NLP

Lecture-7

Multi-head attention, Transformers

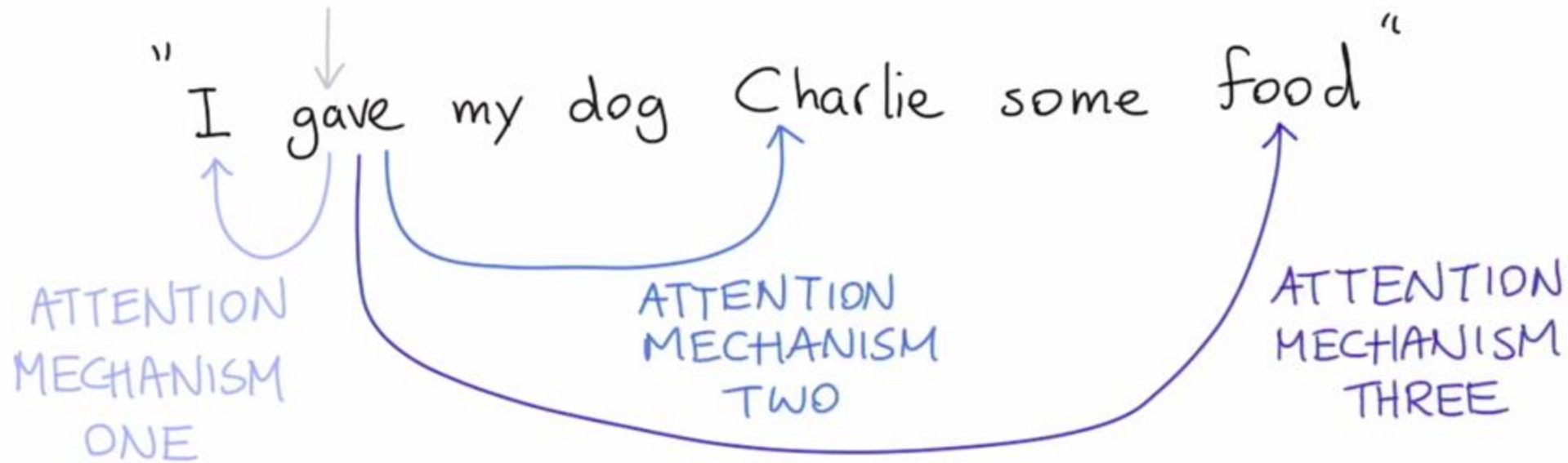
Recap

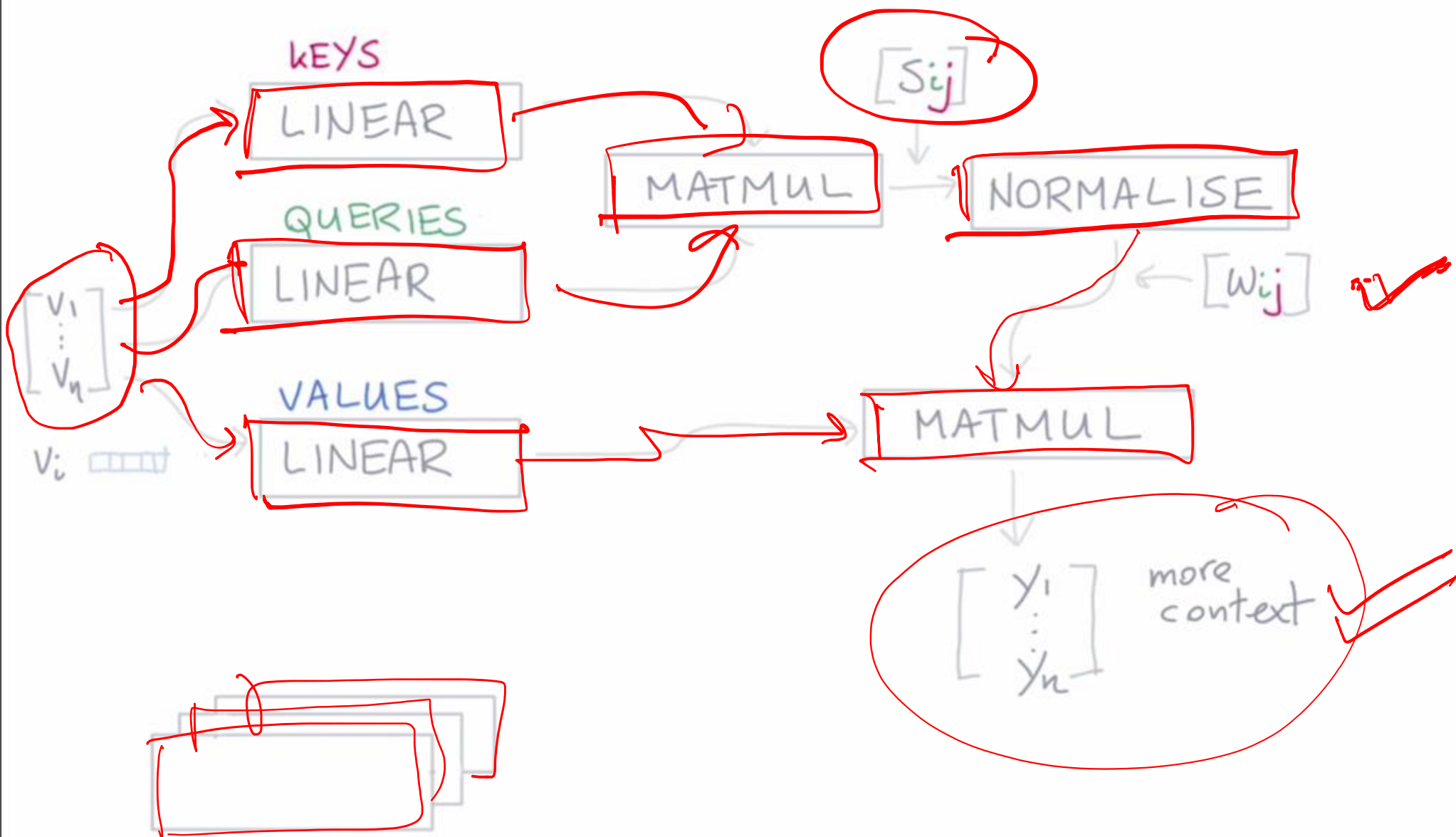
- Language modeling
- Recurrent Neural Network and Implementation
- Applications of Recurrent Neural Network
- Language modeling using Long Short-term Memory
- Sequence to Sequence learning
- Attention

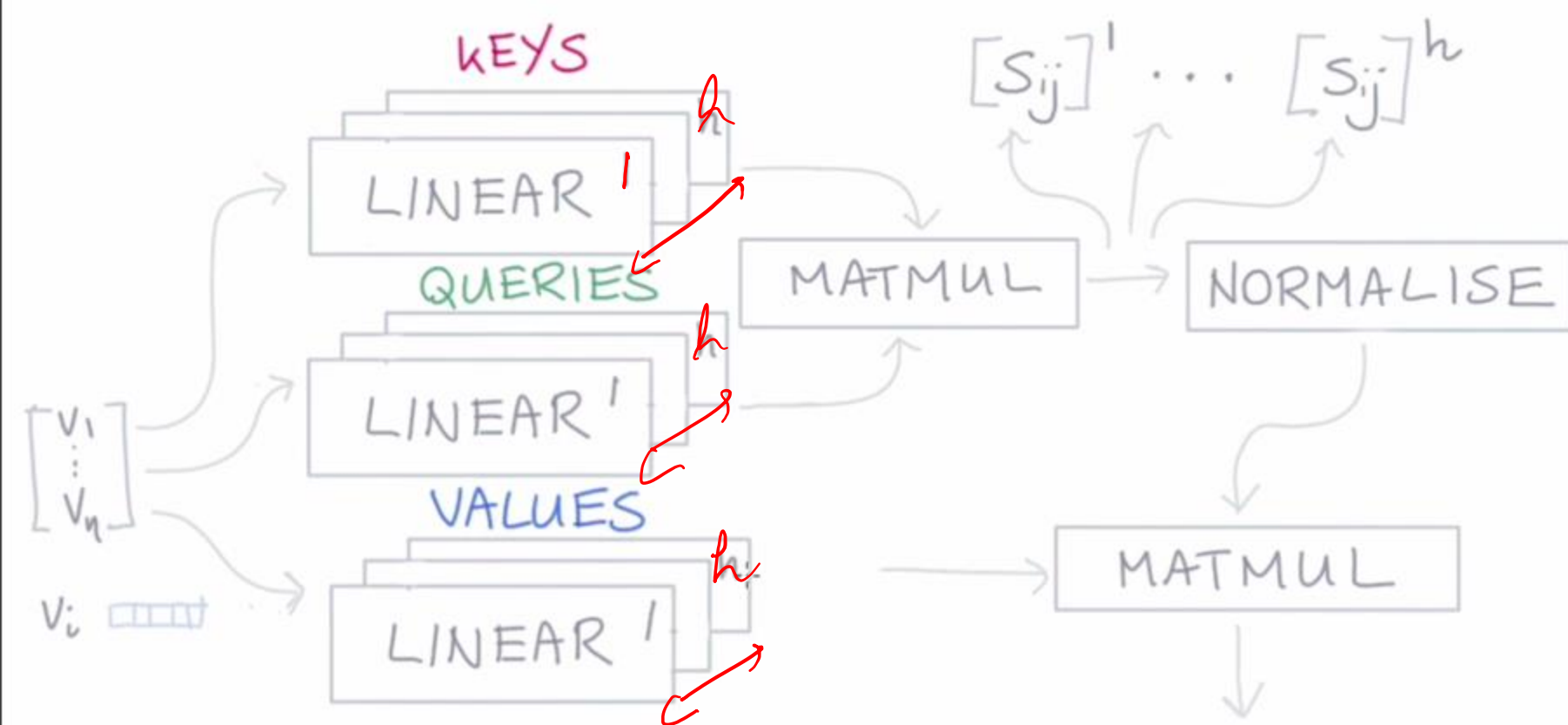
So Far covered.....Intuition behind Attention

- The weighted sum is a **selective summary** of the information contained in **the values**, where the query determines which values to focus on.
- Attention is a way to obtain a **fixed-size representation of an arbitrary set of representations (the values)**, dependent on some **other representation (the query)**

Do we have enough attention?

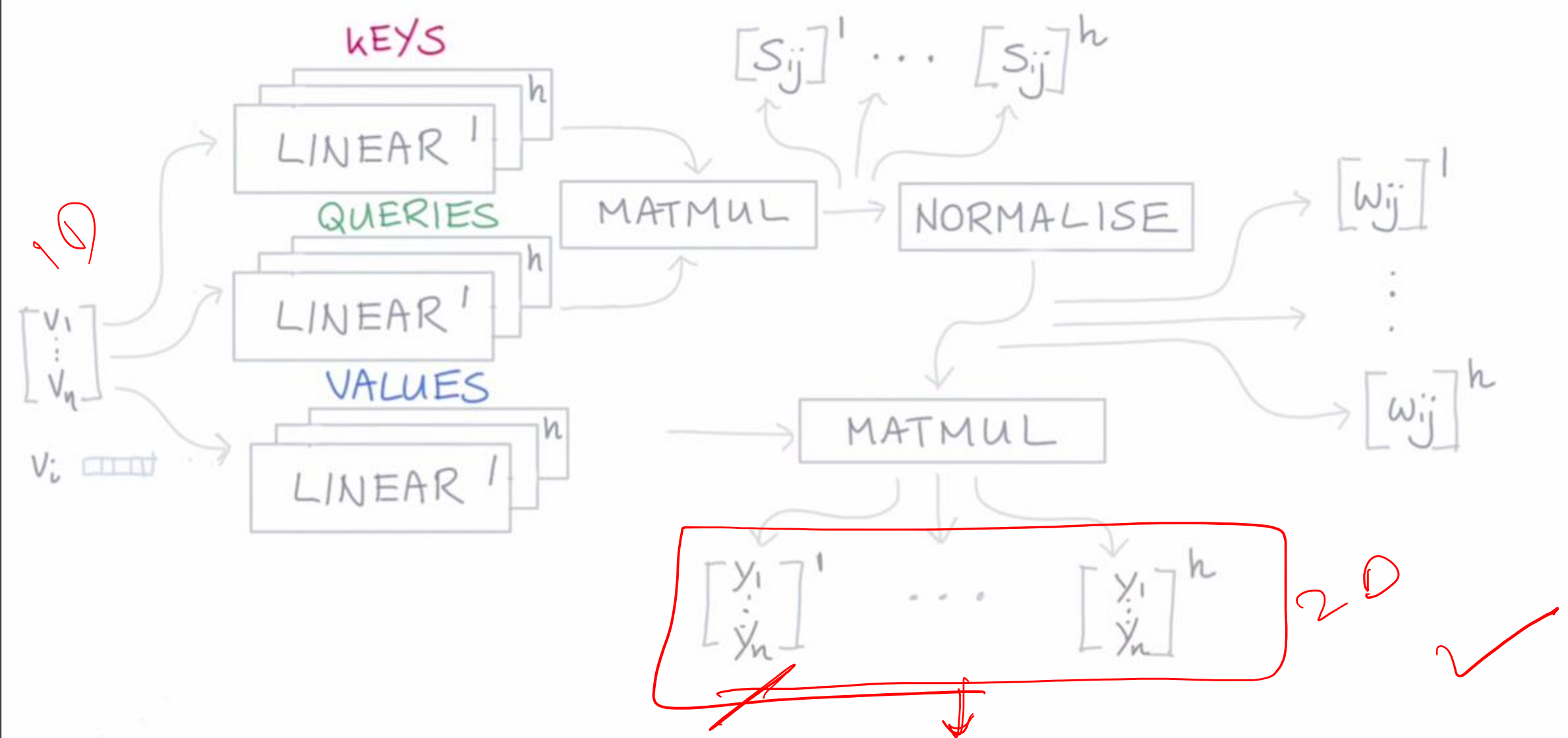


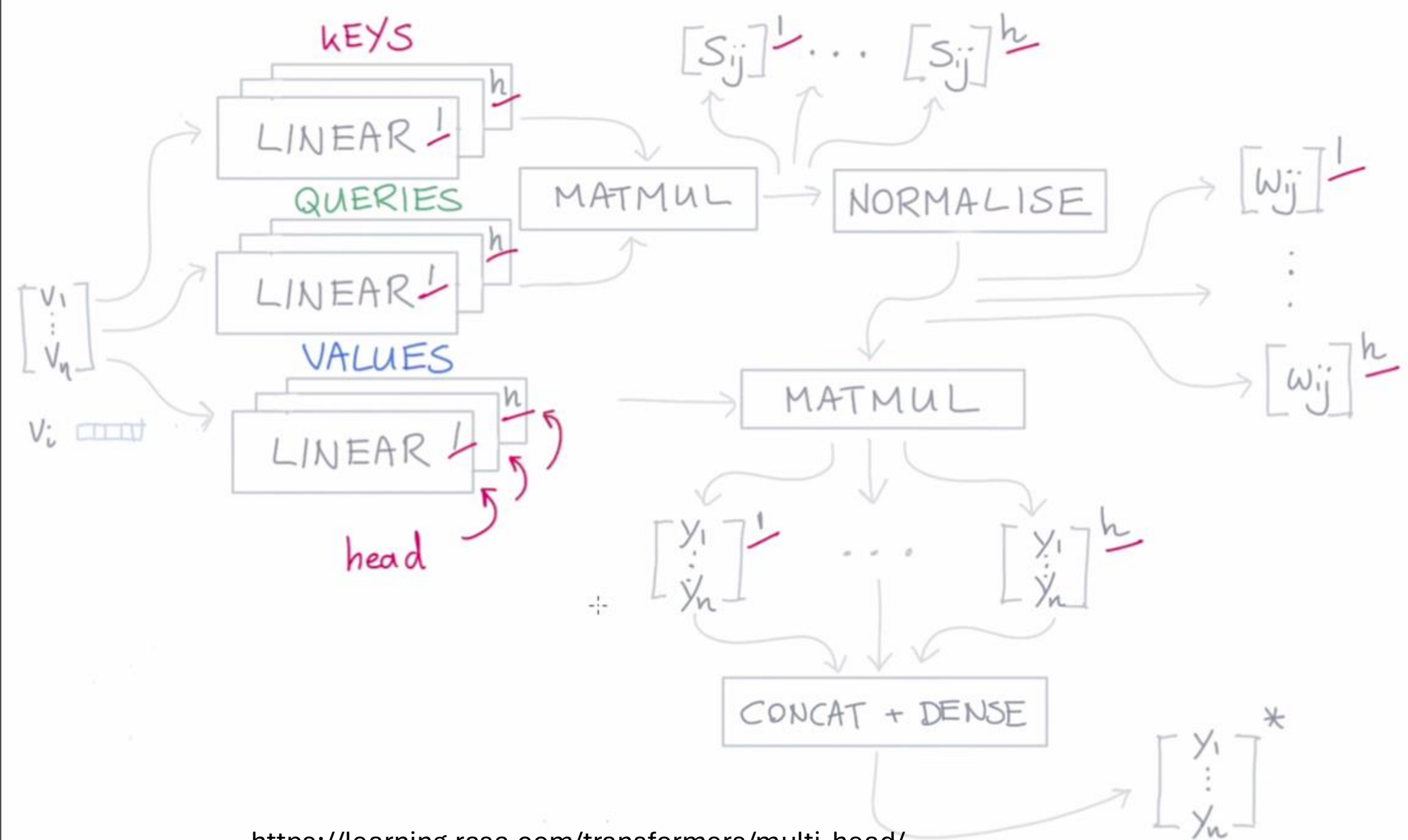


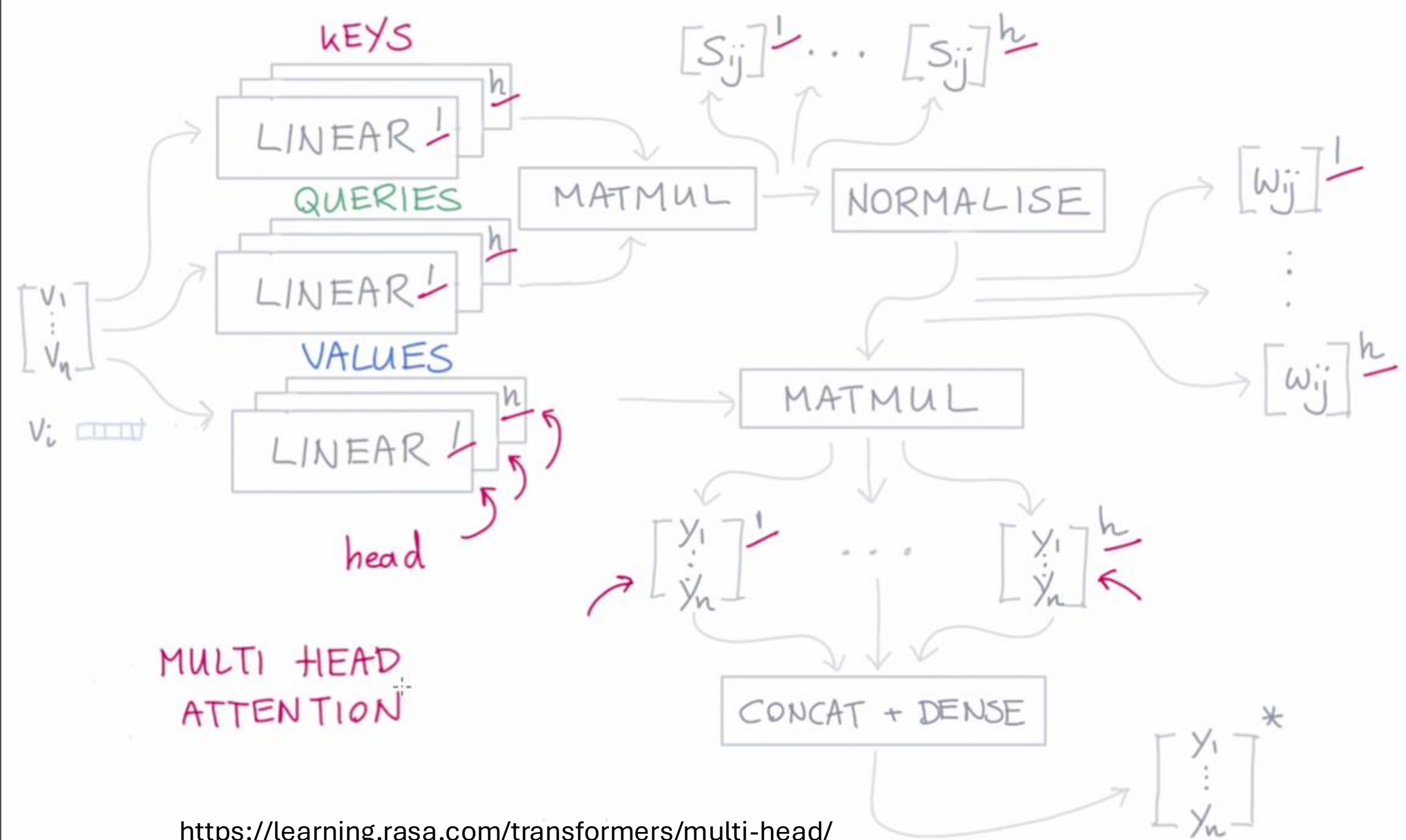


add more linear layers

so attend many words at a time







Multi-head attention

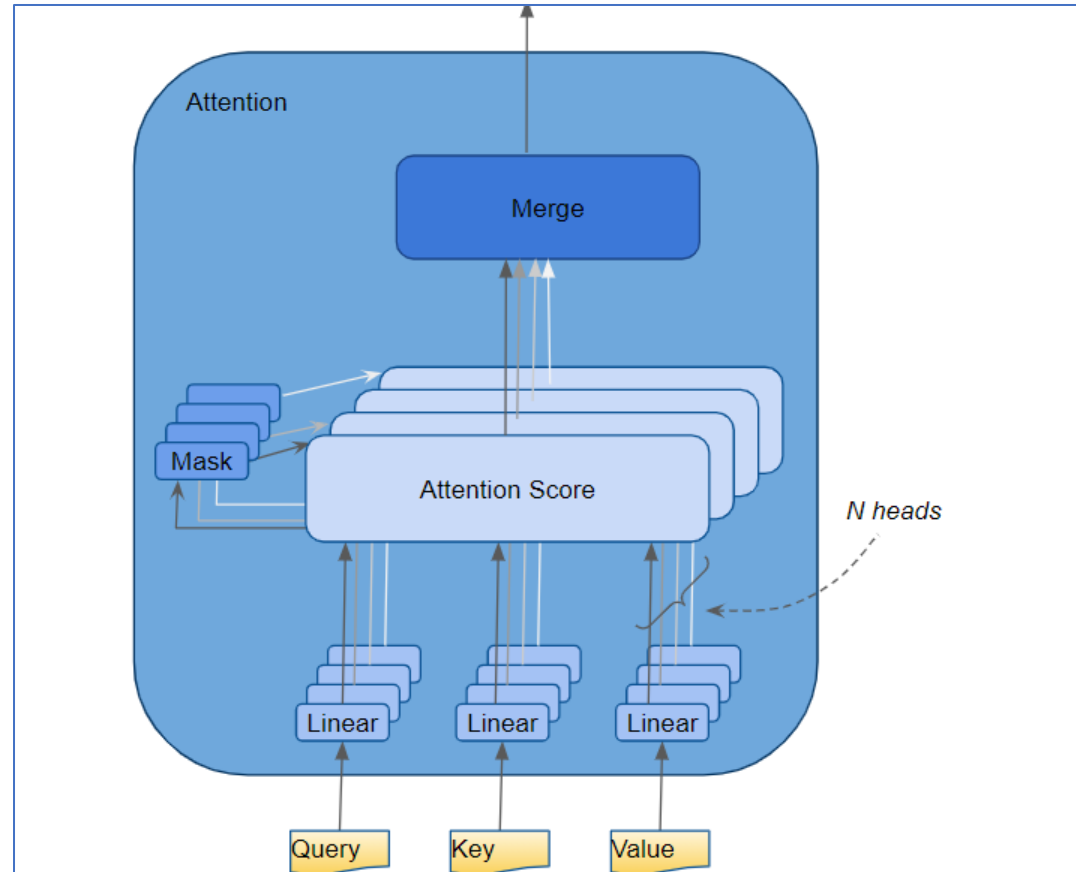
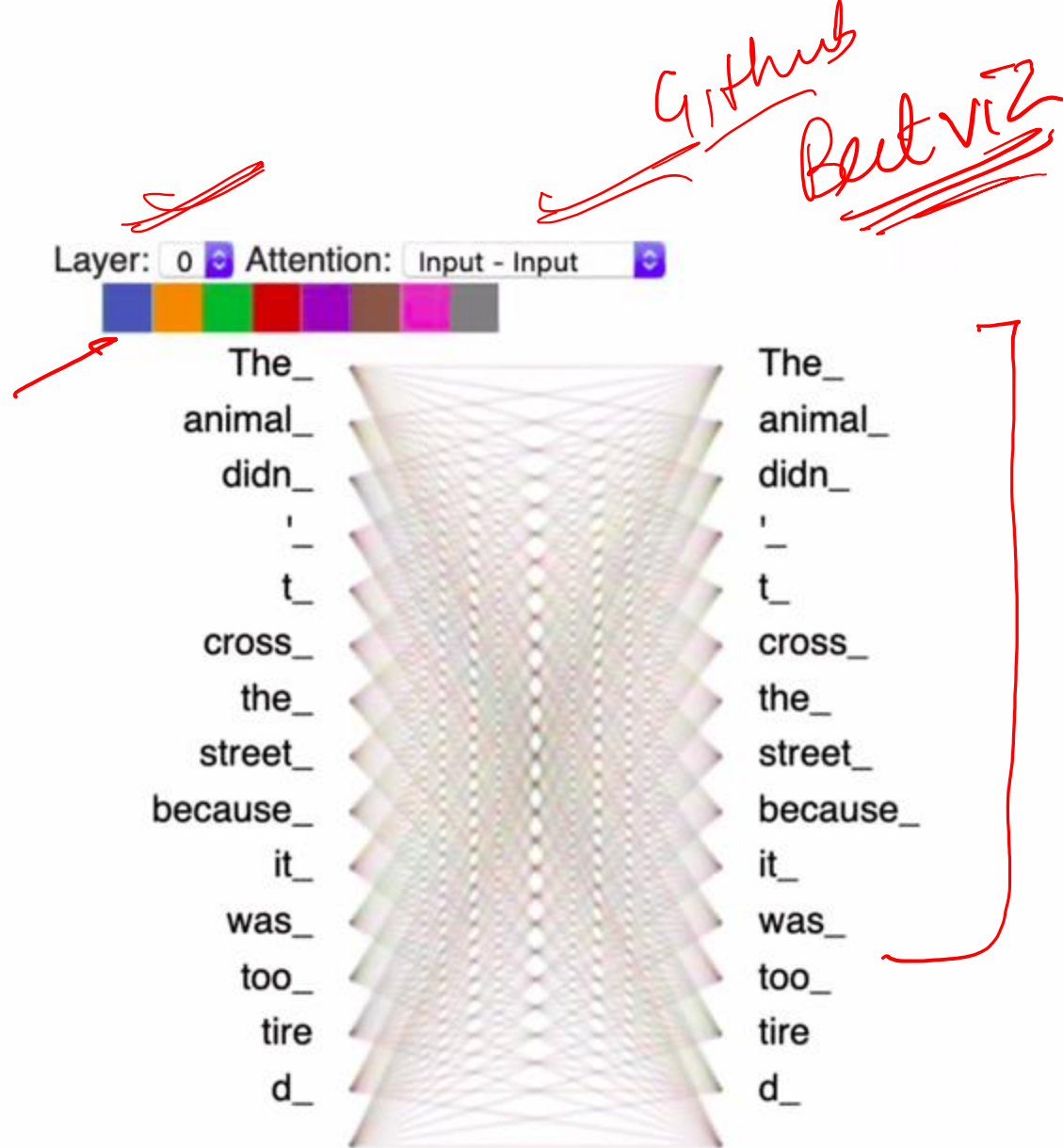
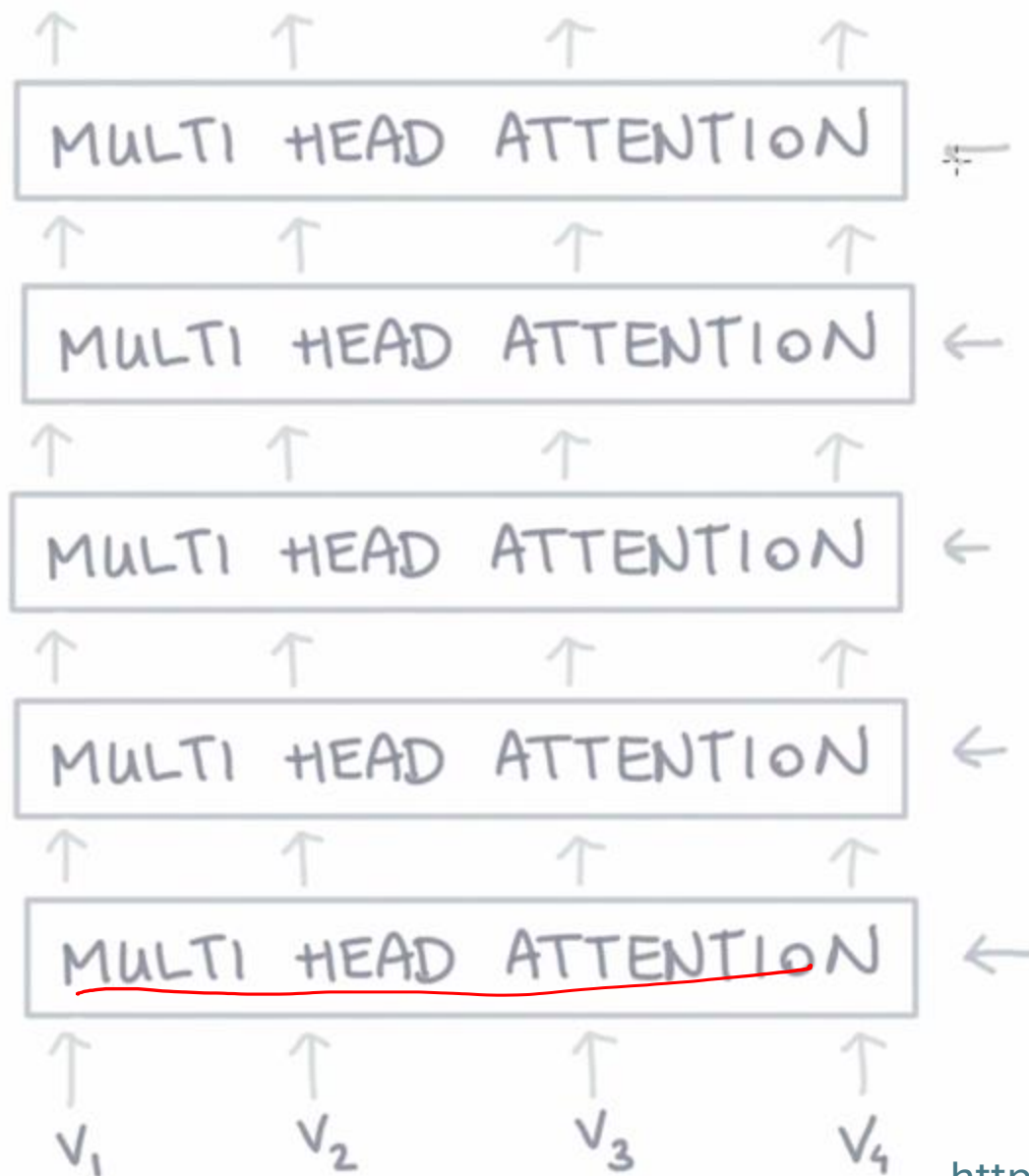
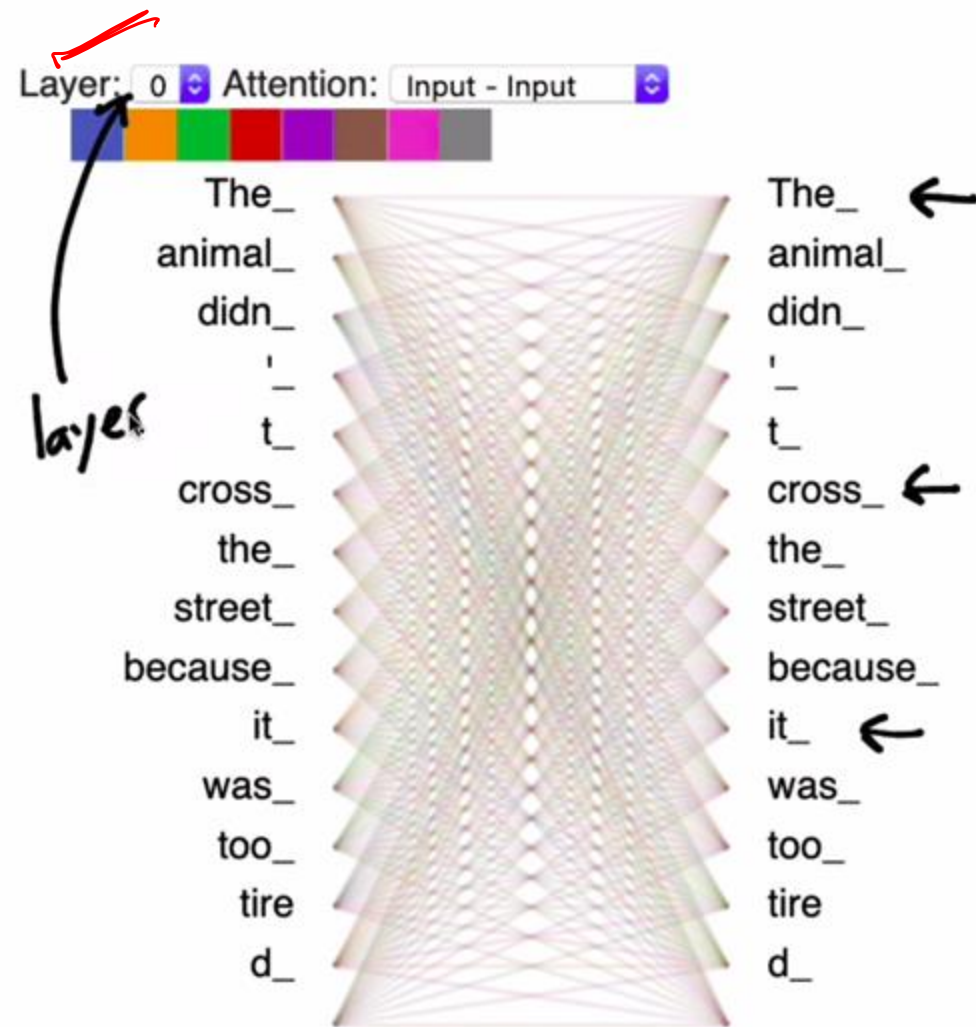
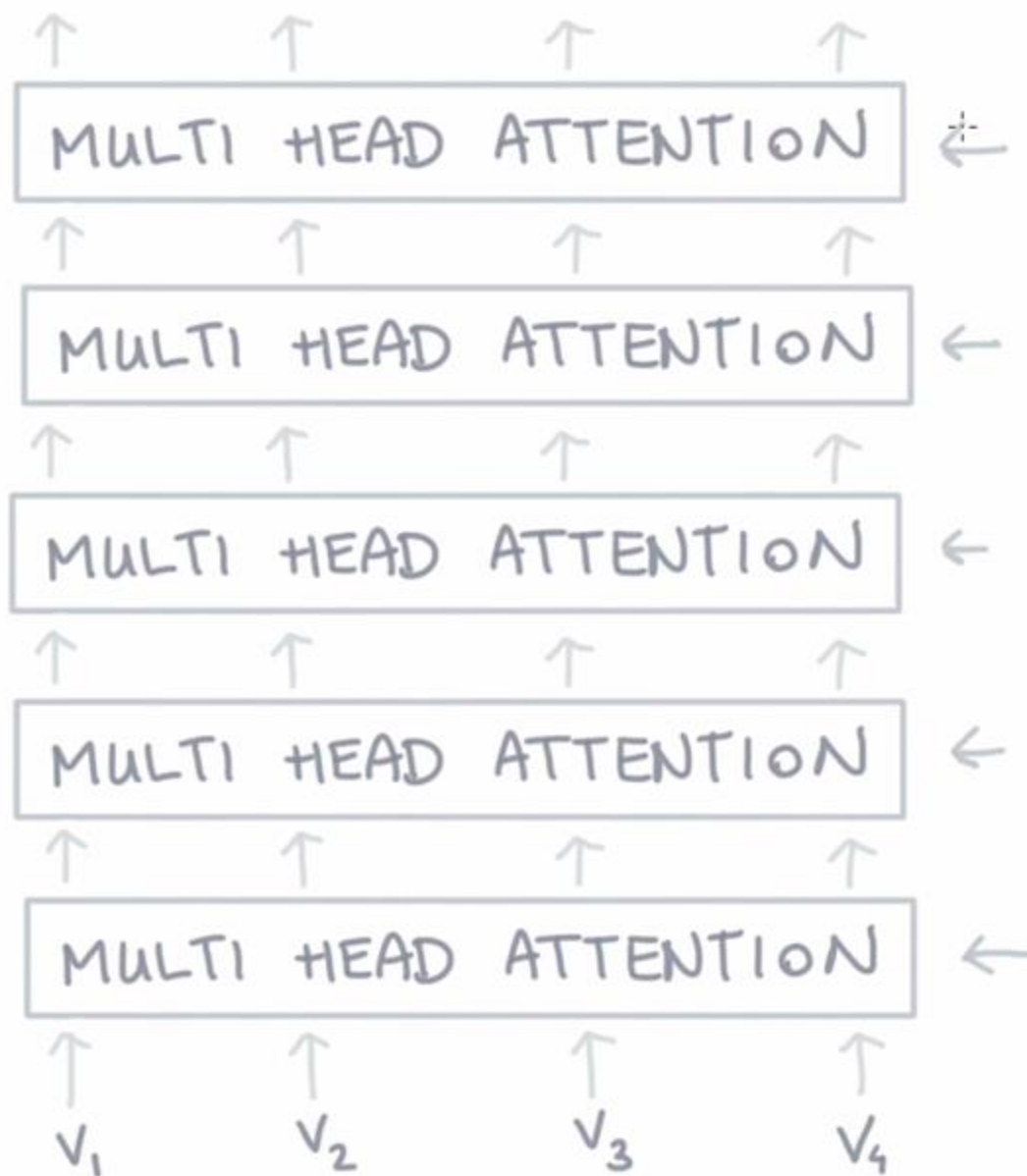


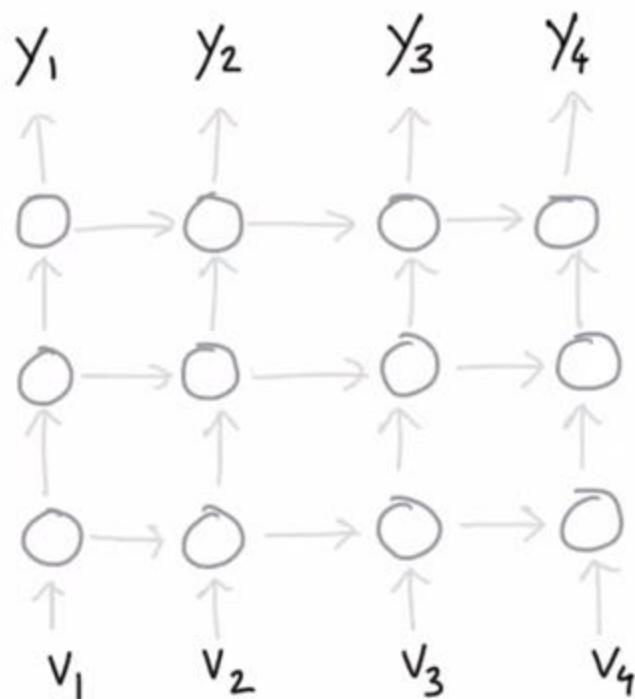
Image source:

<https://towardsdatascience.com/transformers-explained-visually-part-3-multi-head-attention-deep-dive->

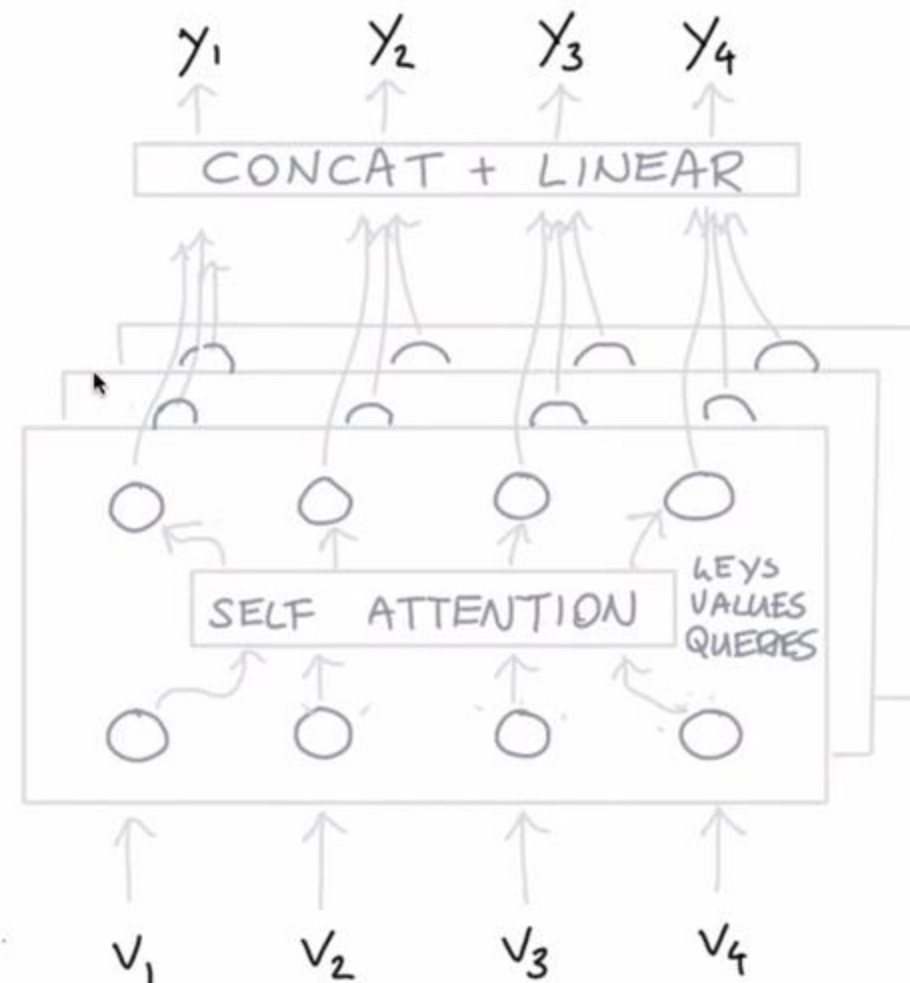




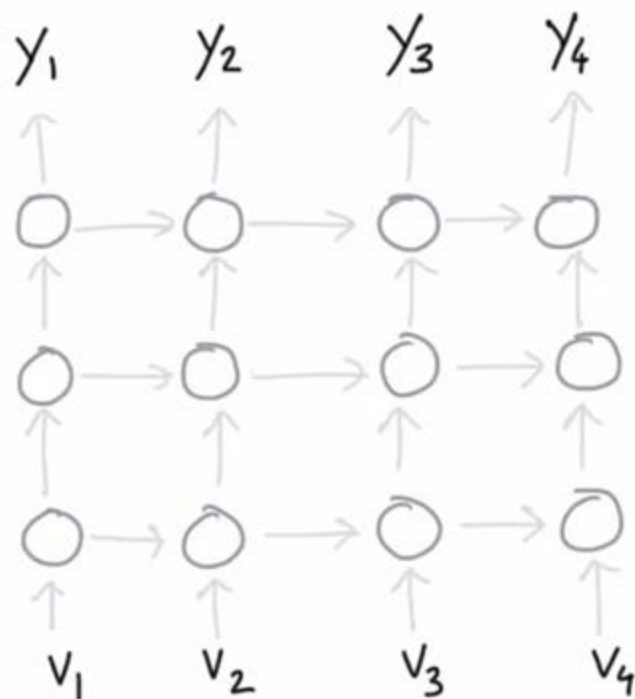
RECURRENT



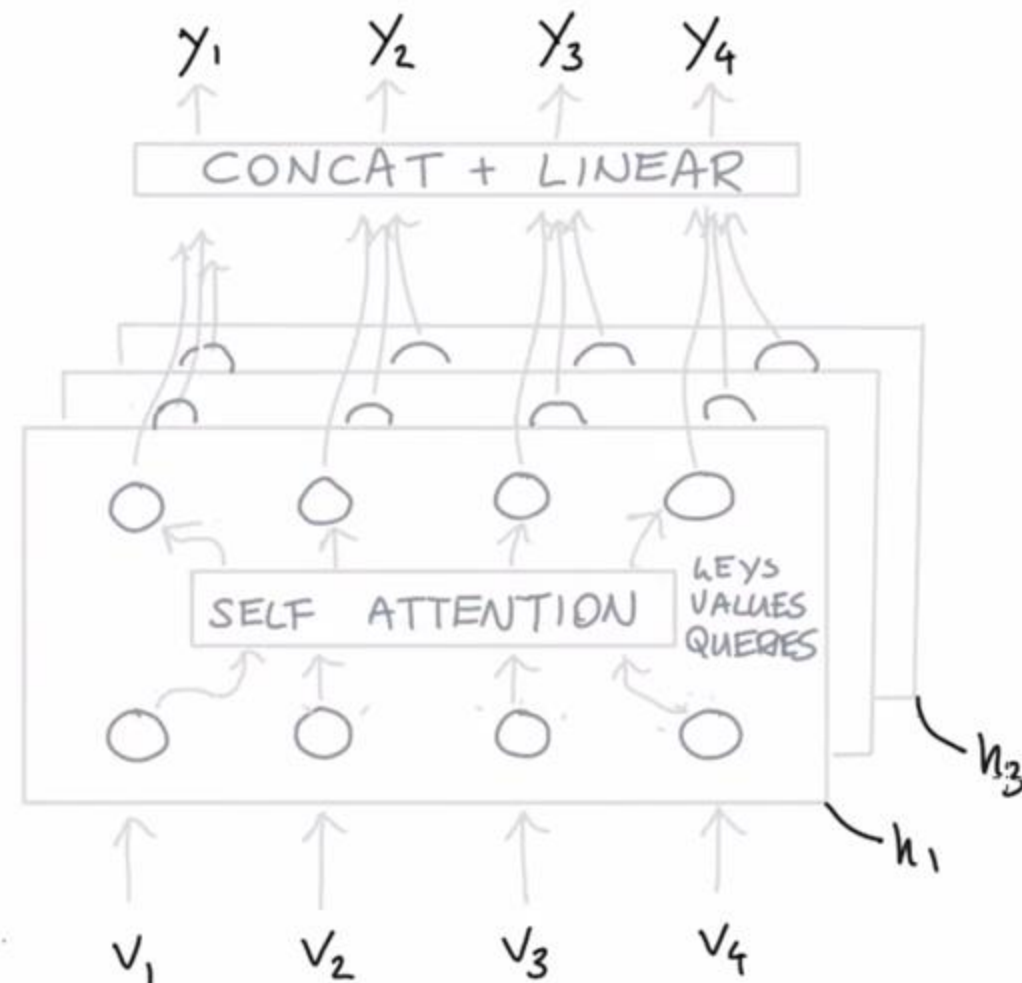
MULTI HEAD ATTENTION



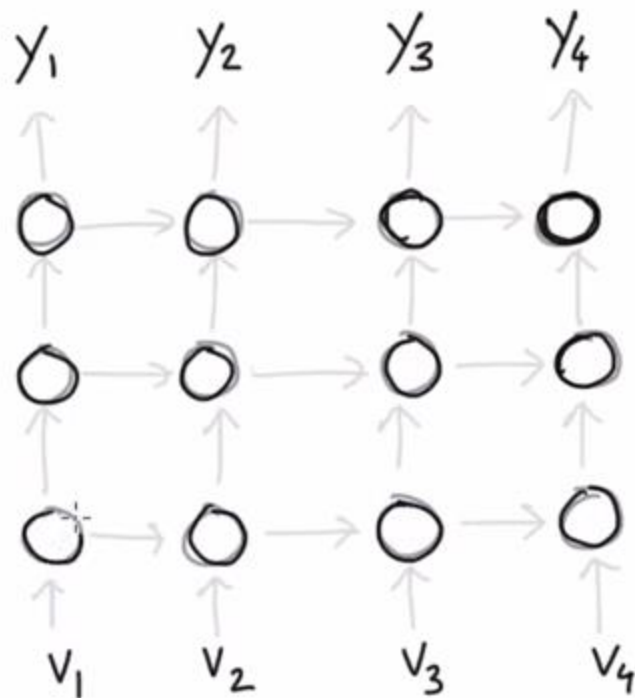
RECURRENT



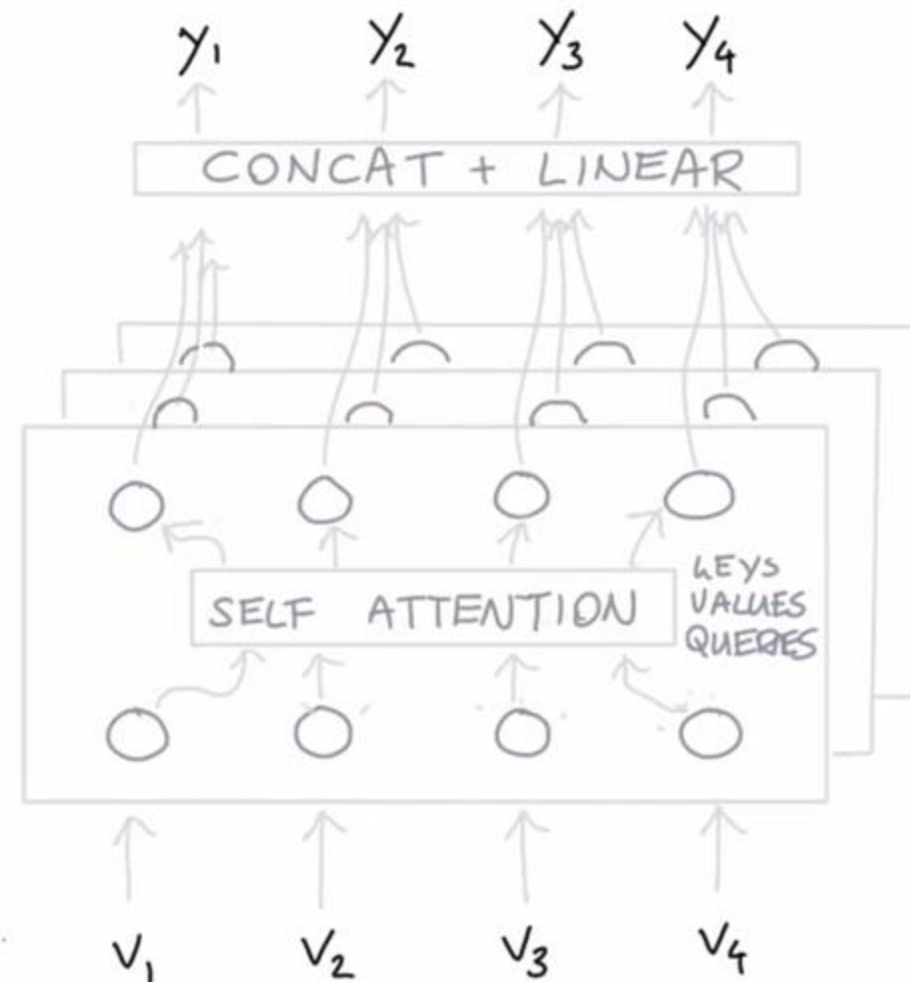
MULTI HEAD ATTENTION



RECURRENT



MULTI HEAD ATTENTION

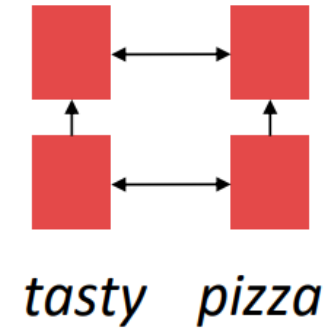


Issues with Recurrent models: Linear interaction distance

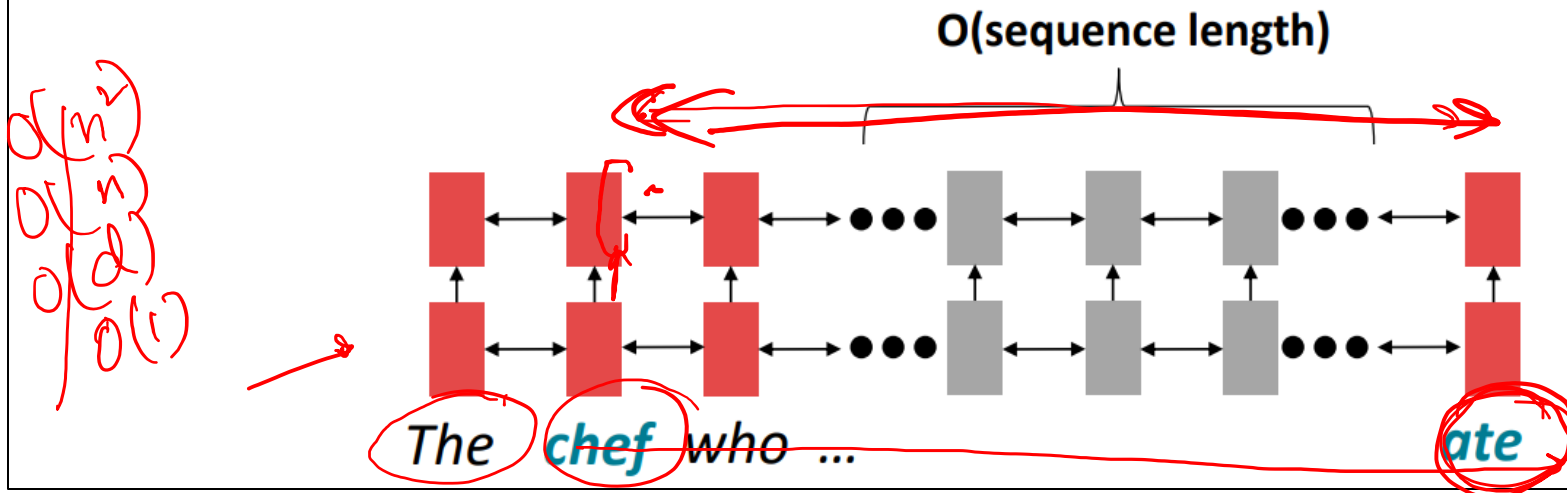
RNNs are unrolled “left-to-right”.

It encodes linear locality: a useful heuristic!

- Nearby words often affect each other’s meanings



Problem: RNNs take $O(\text{sequence length})$ steps for distant word pairs to interact.

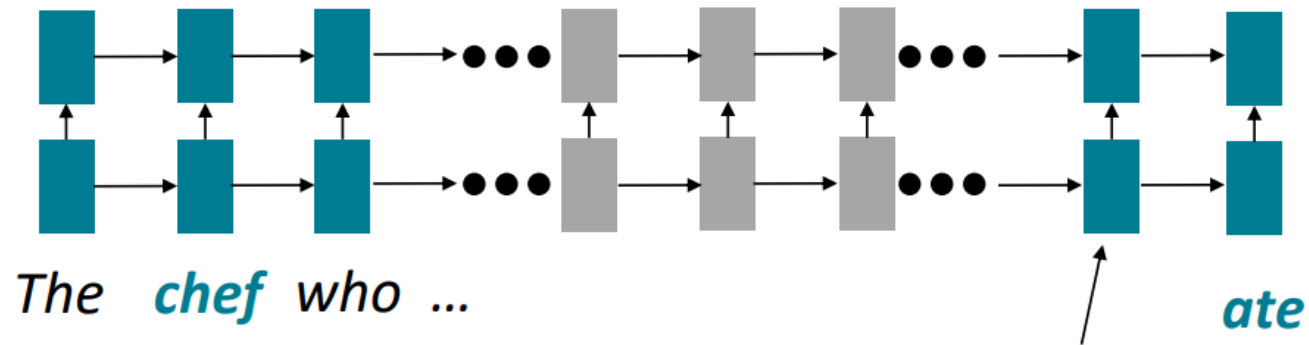


$O(n^2)$ → sequence length
(chef, ate)

Issues with Recurrent models: Linear interaction distance

$O(\text{sequence length})$ steps for distant word pairs to interact means:

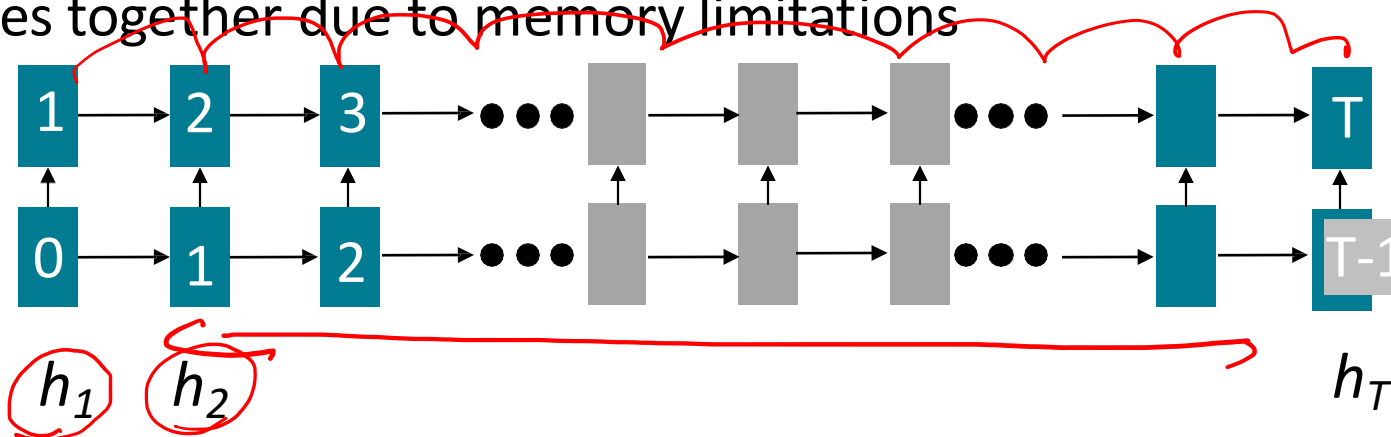
- Hard to learn long-distance dependencies (because gradient problems!)
- Linear order of words is “baked in”; we already know sequential structure ↗
doesn't tell the whole story...



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

Lack of Parallelizability in RNN

- Forward and backward passes have **$O(\text{seq length})$** 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

tion

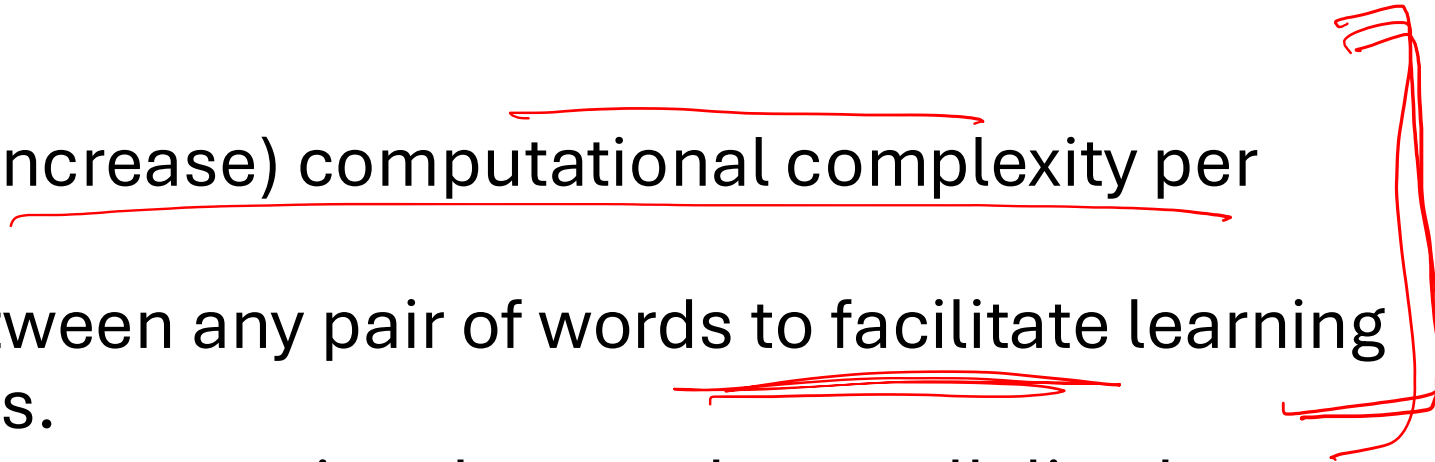
[illegible]

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



Transformer architecture

The Transformers authors had 3 desiderata when designing this architecture:

1. Minimize (or at least not increase) computational complexity per layer.
 2. Minimize path length between any pair of words to facilitate learning of long-range dependencies.
 3. Maximize the amount of computation that can be parallelized.
- 

Computational Complexity per layer

When sequence length (n) $\ll$ representation dimension (d), complexity per layer is lower for a Transformer compared to the recurrent models we've learned about so far.

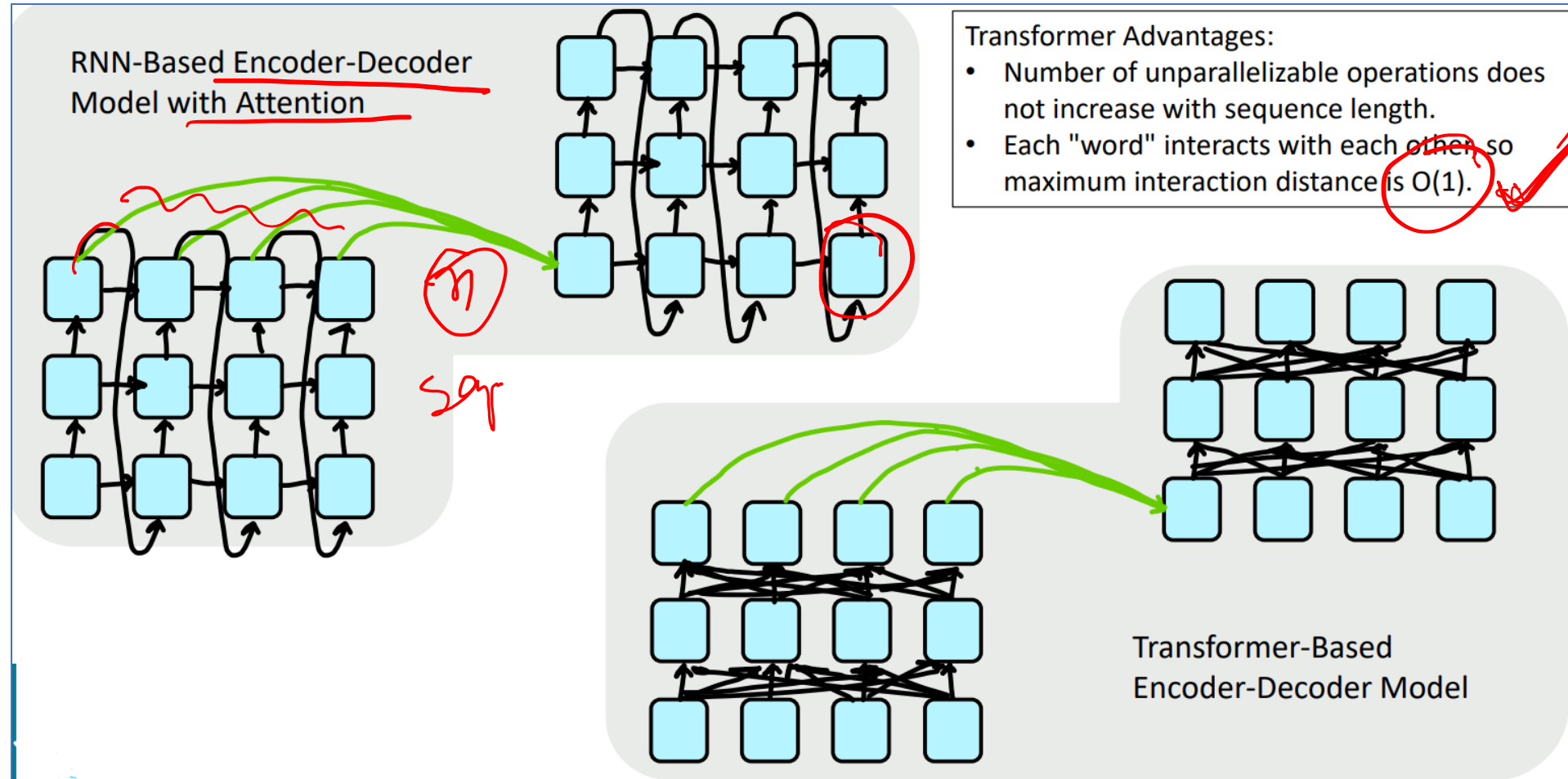
Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(\log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Table 1 of the Transformer paper.

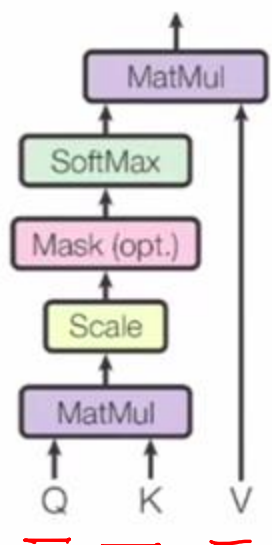
d = vector dimension
 n = sequence length

Path Length minimization and Parallelization



Transformer

Scaled Dot-Product Attention



Multi-Head Attention

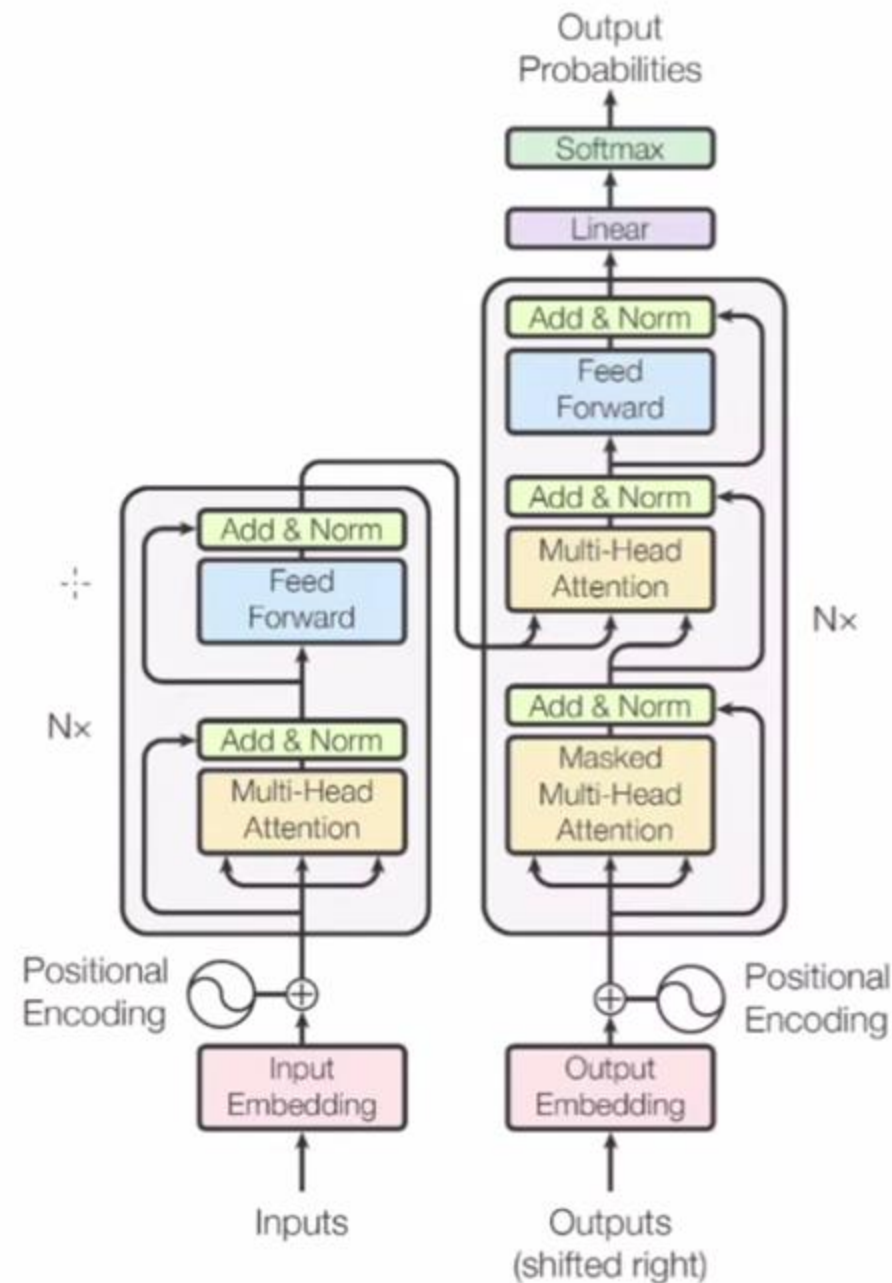
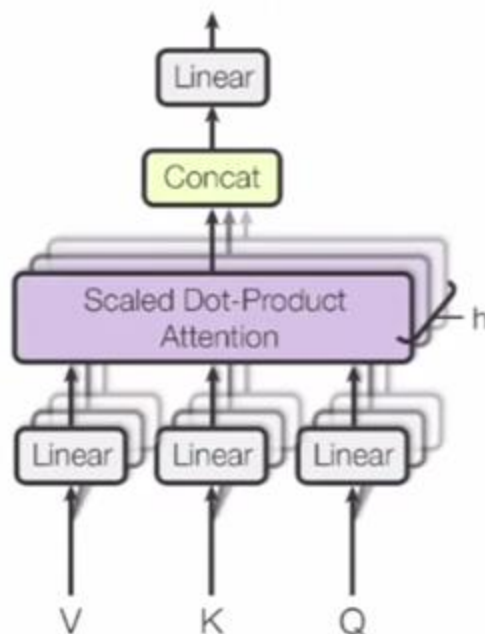
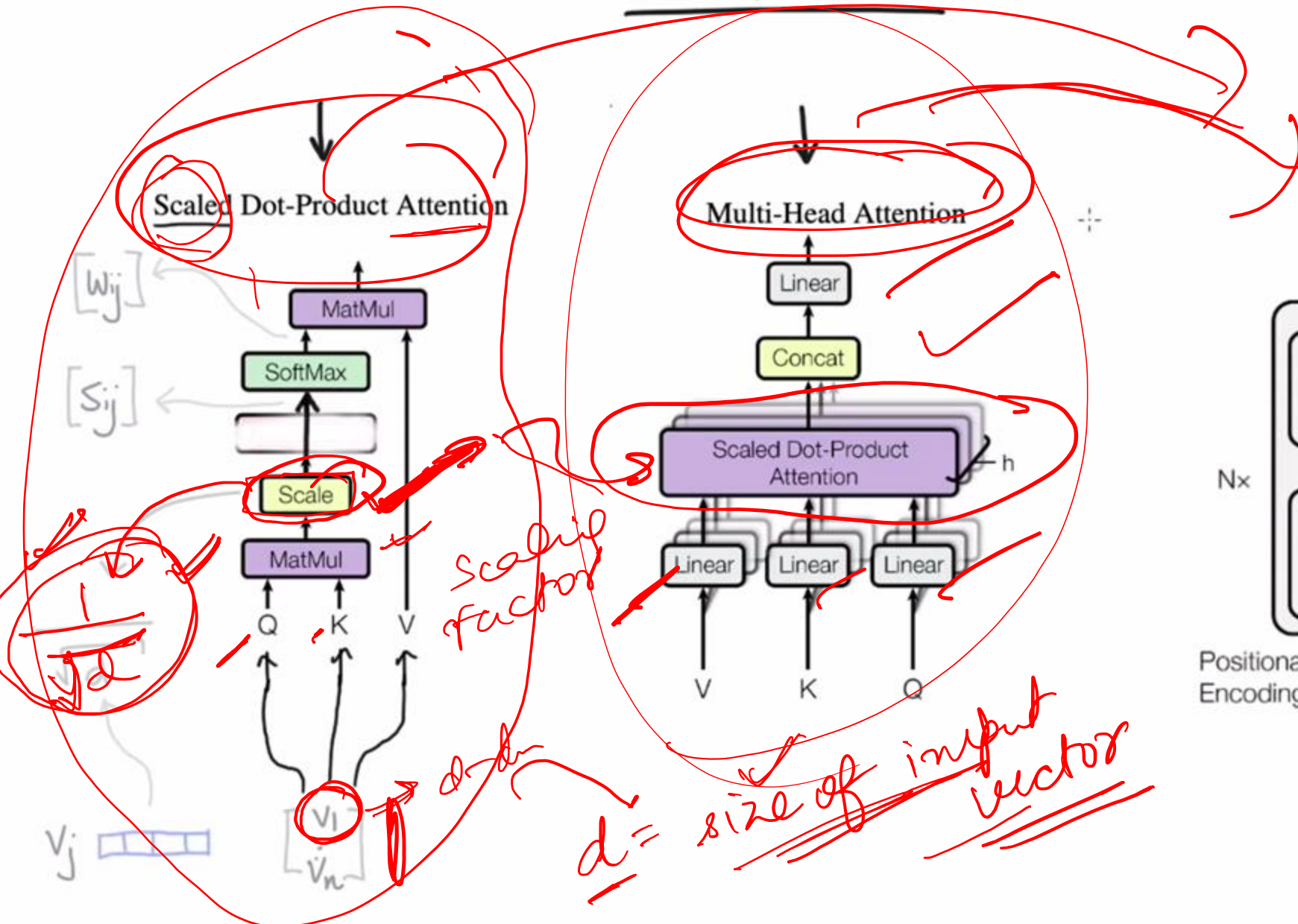


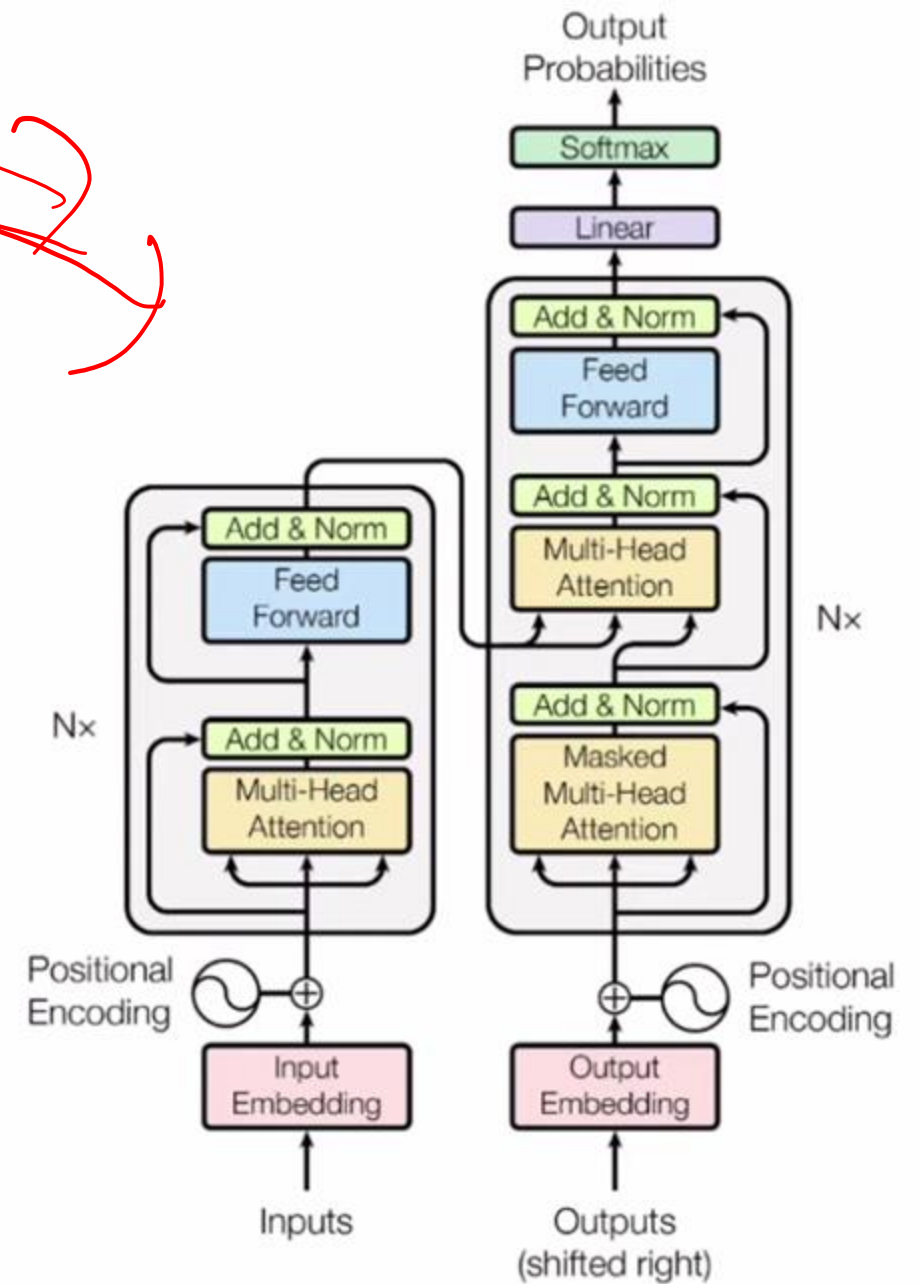
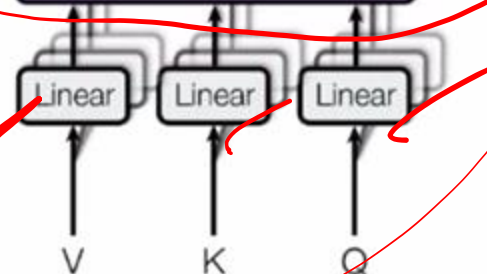
Figure 1: The Transformer - model architecture.

Transformer

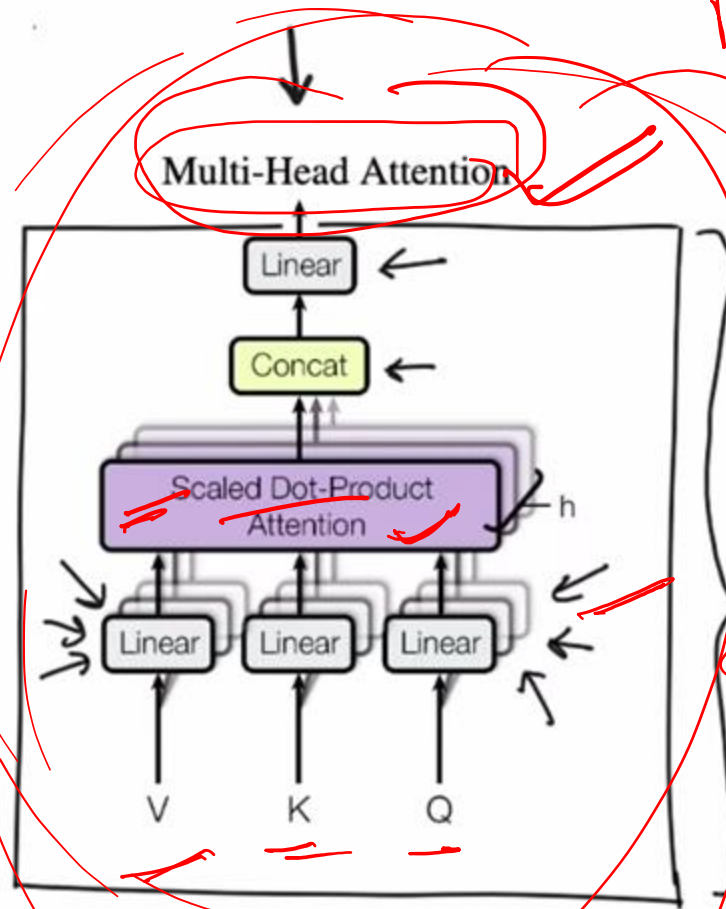
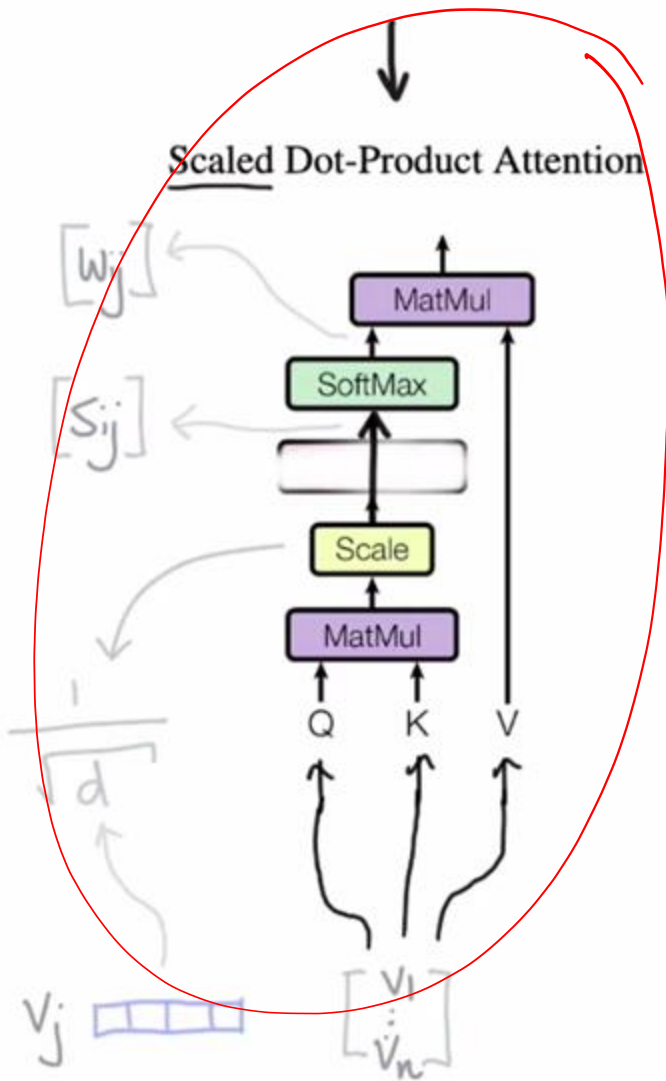


Multi-Head Attention

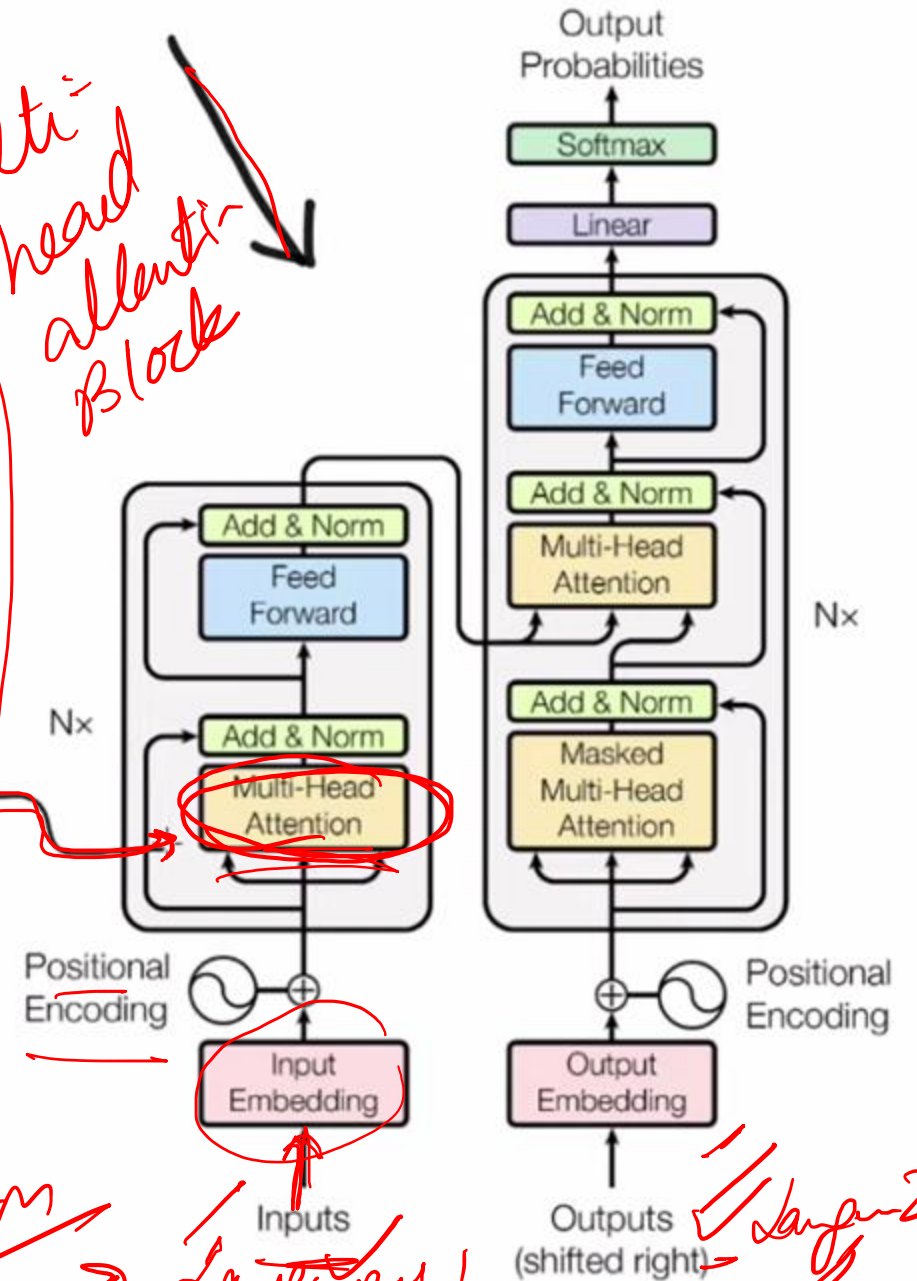
Scaled Dot-Product Attention



Transformer



Multi-head attention block



[Neural Machine Translation]

Layer 1
Layer 2

Figure 1: The Transformer - model architecture.

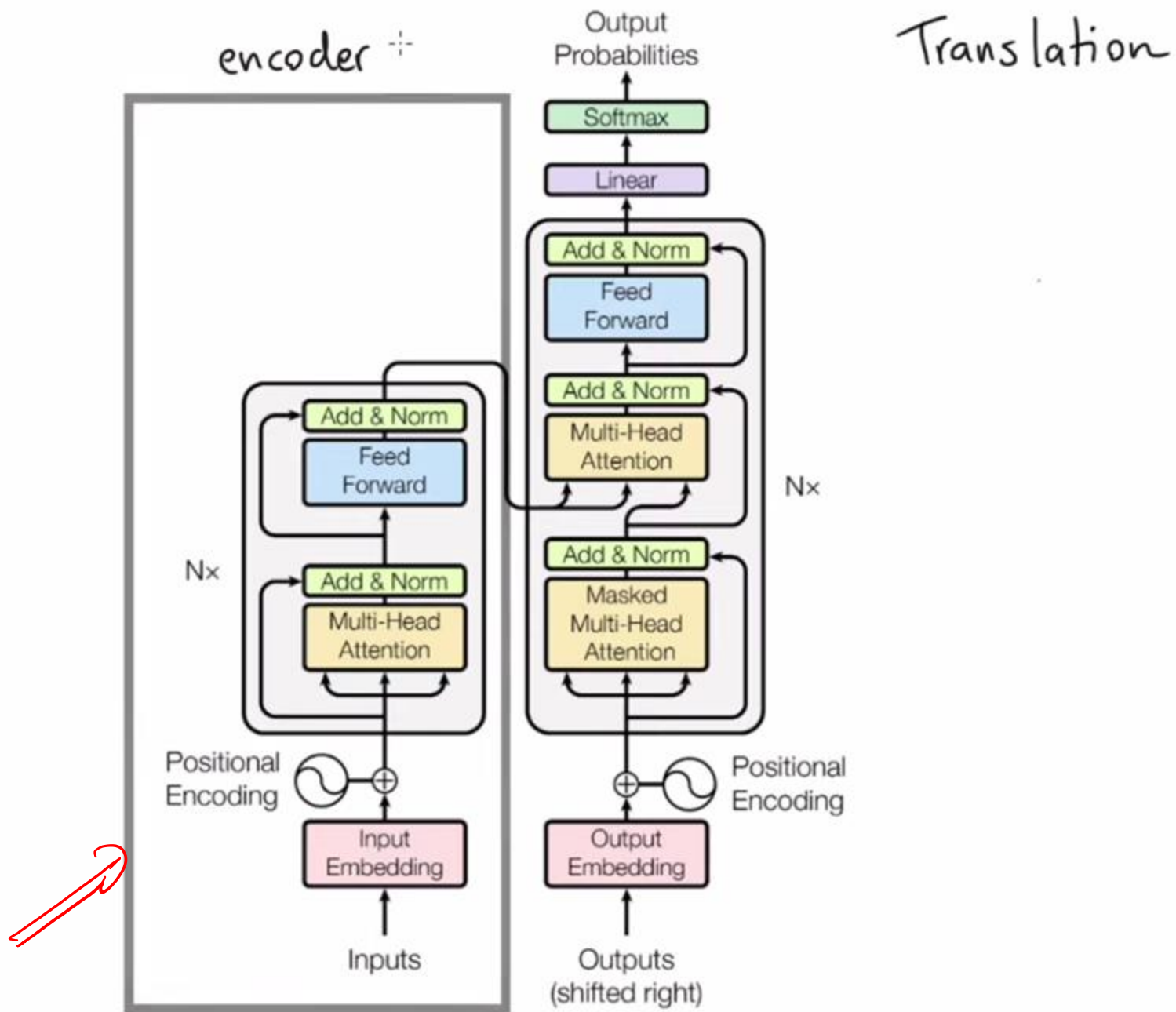
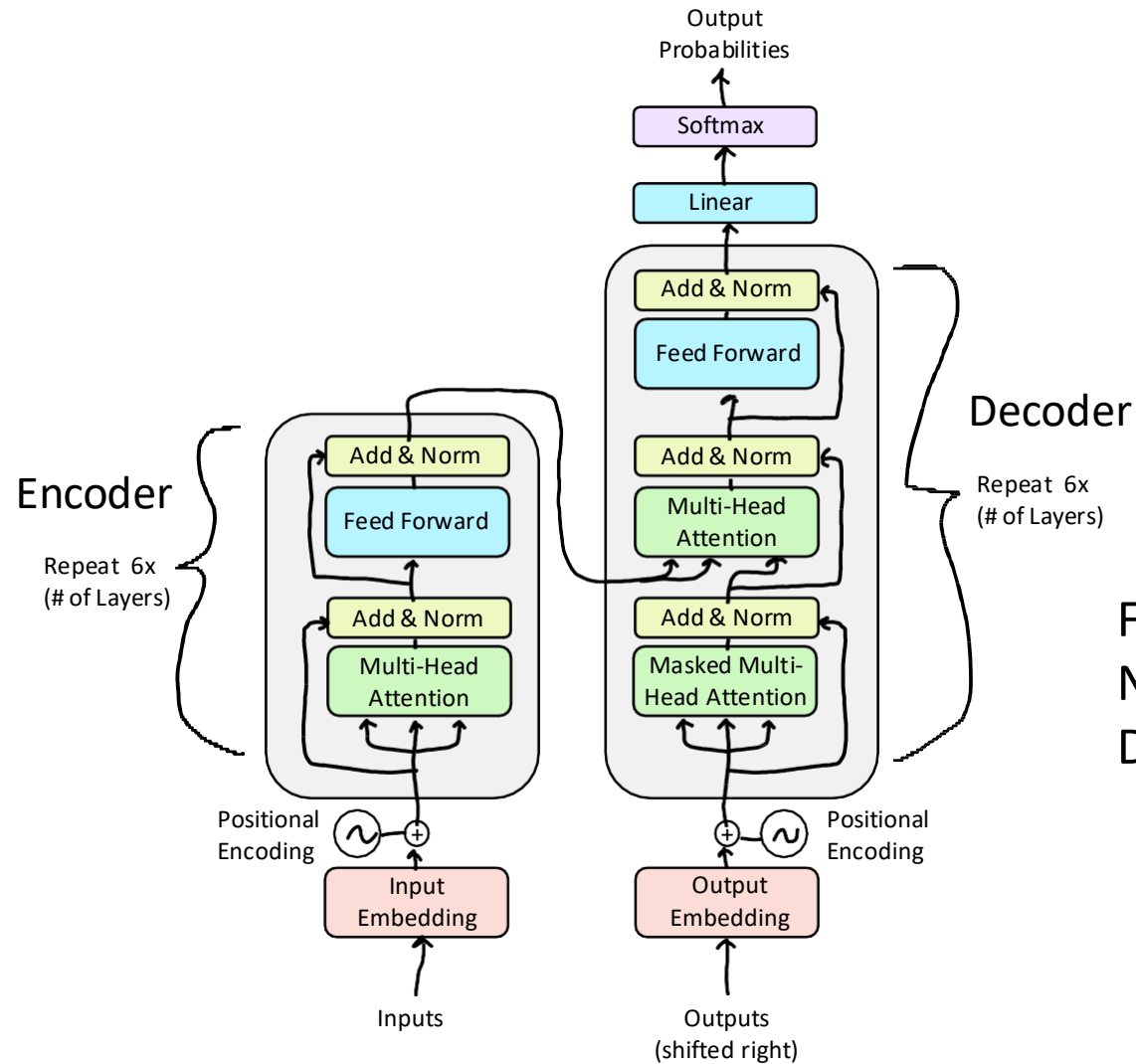


Figure 1: The Transformer - model architecture.

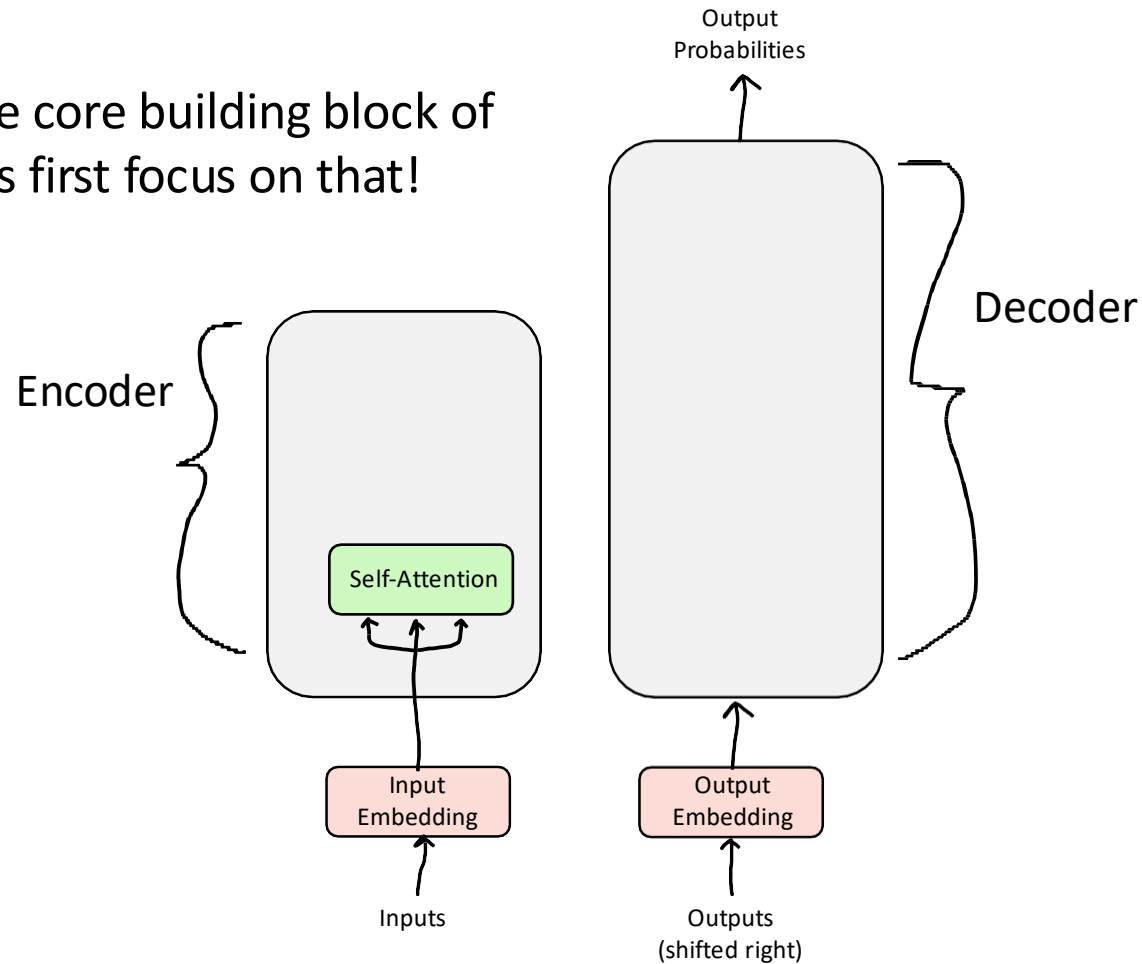
The Transformer Encoder-Decoder [Vaswani et al., 2017]



First, we will talk about the Encoder!
Next, we will go through the
Decoder (which is quite similar)!

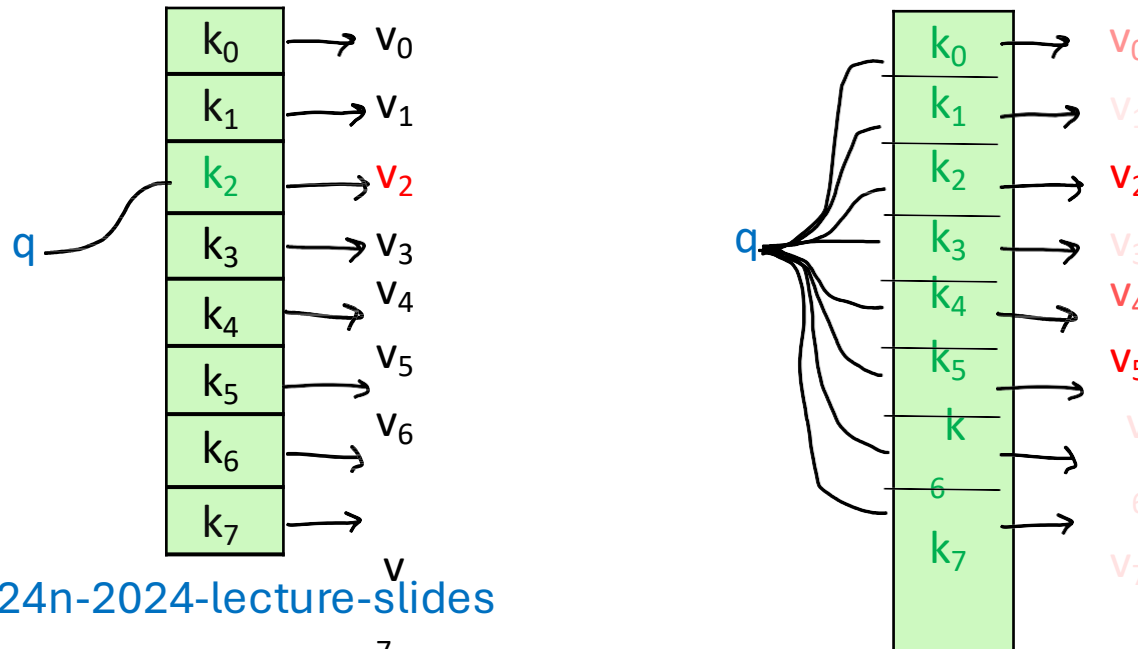
Encoder: Self-Attention

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 x_i , 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**.

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

Query Keys

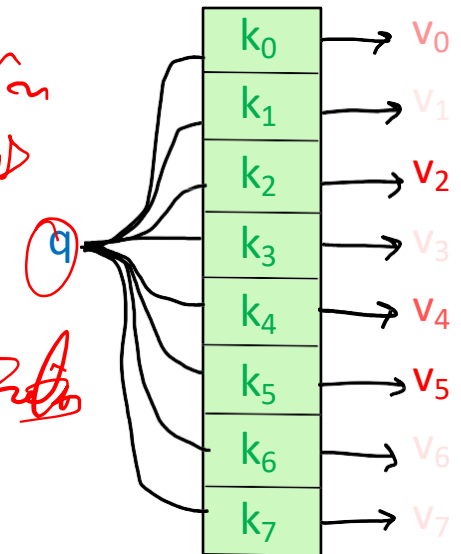
- Step 3: Take the softmax to normalize attention scores.

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

Normalized

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

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



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.

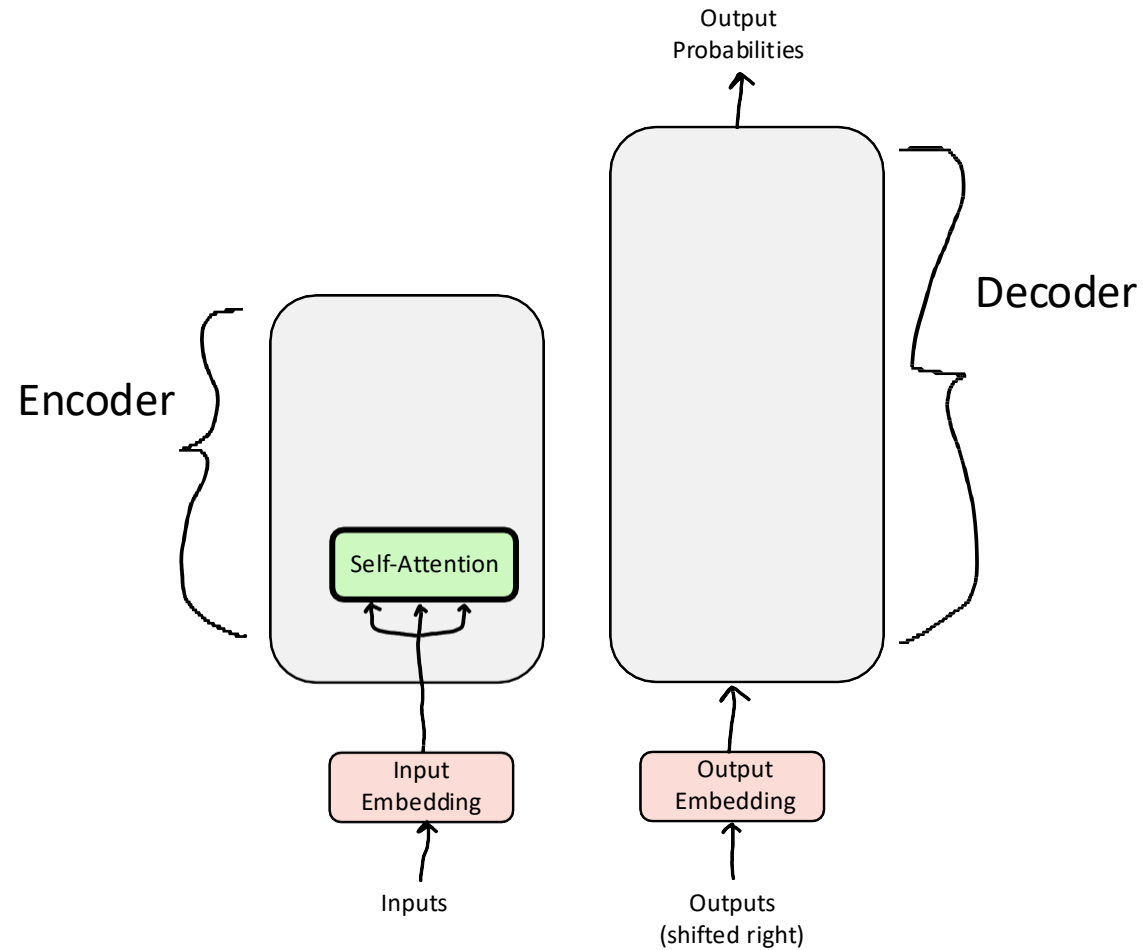
$$A = \text{softmax}(E)$$

- 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!



Attention is All you need!!



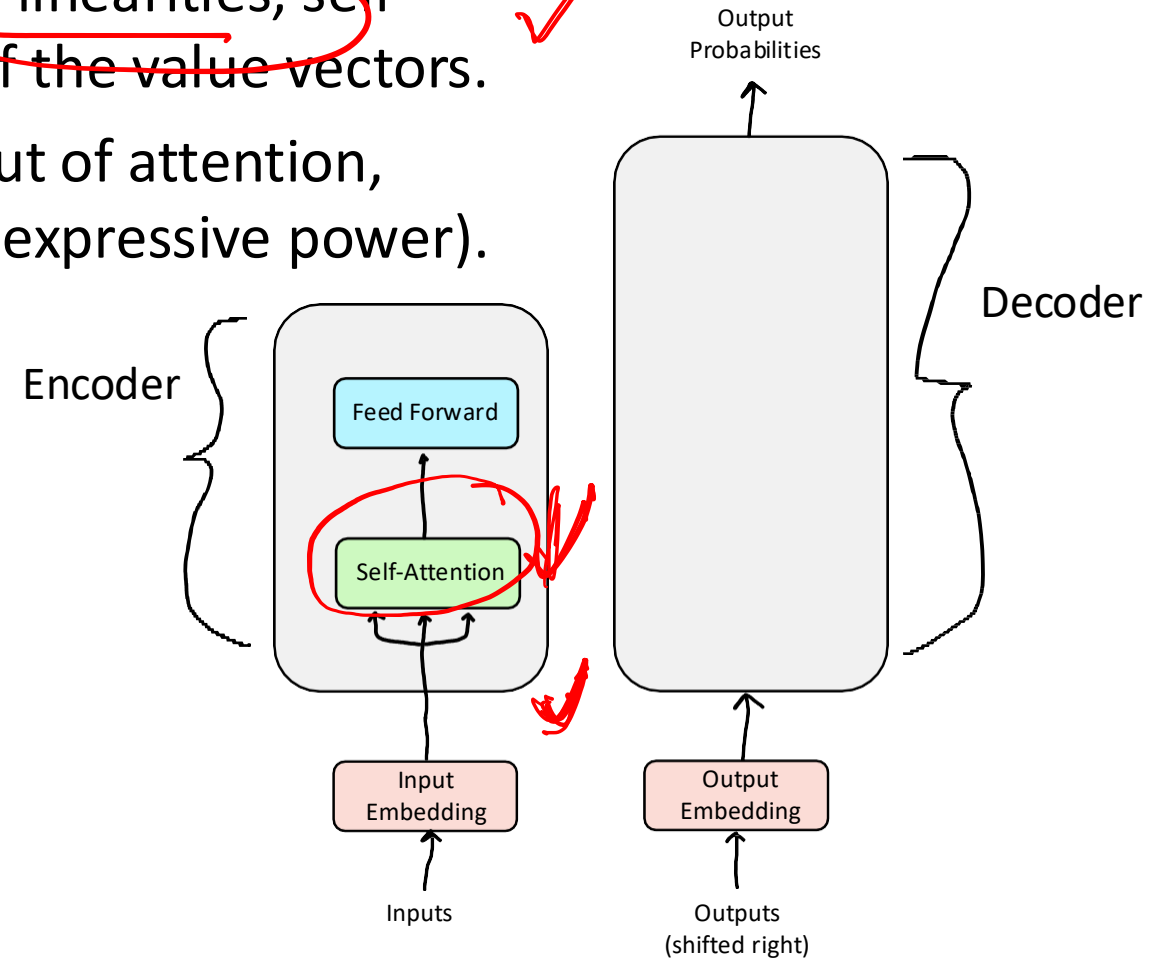
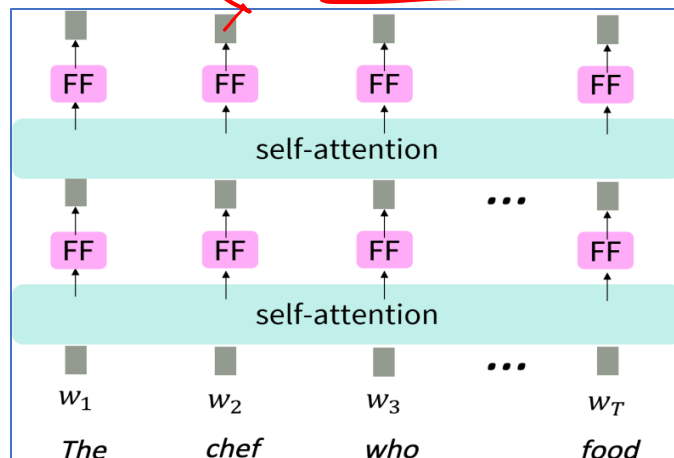
But attention isn't quite all you need!

Problem: Since there are no element-wise non-linearities, self-attention is simply performing a re-averaging of the value vectors.

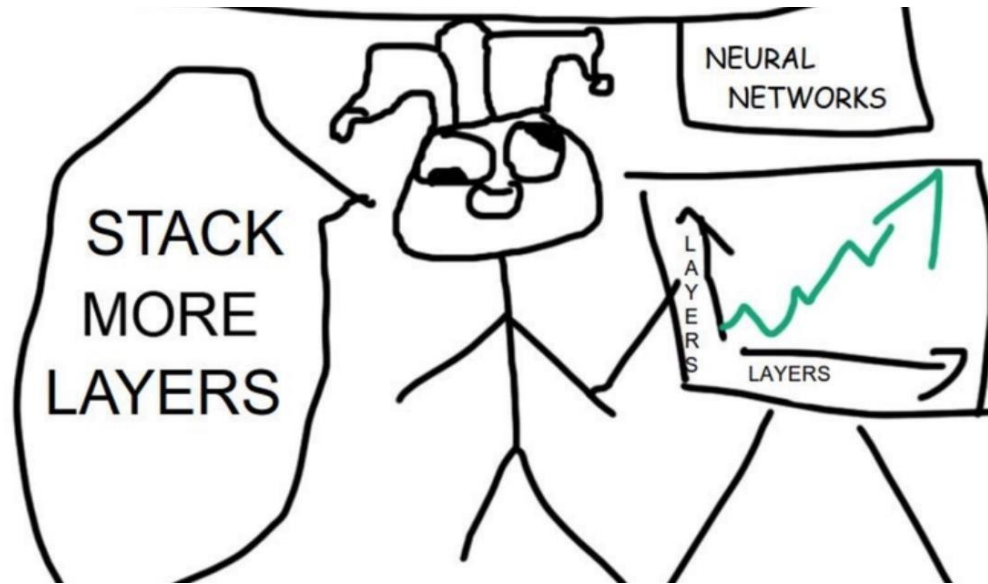
Easy fix: Apply a feedforward layer to the output of attention, providing non-linear activation (and additional expressive power).

Equation for Feed Forward Layer

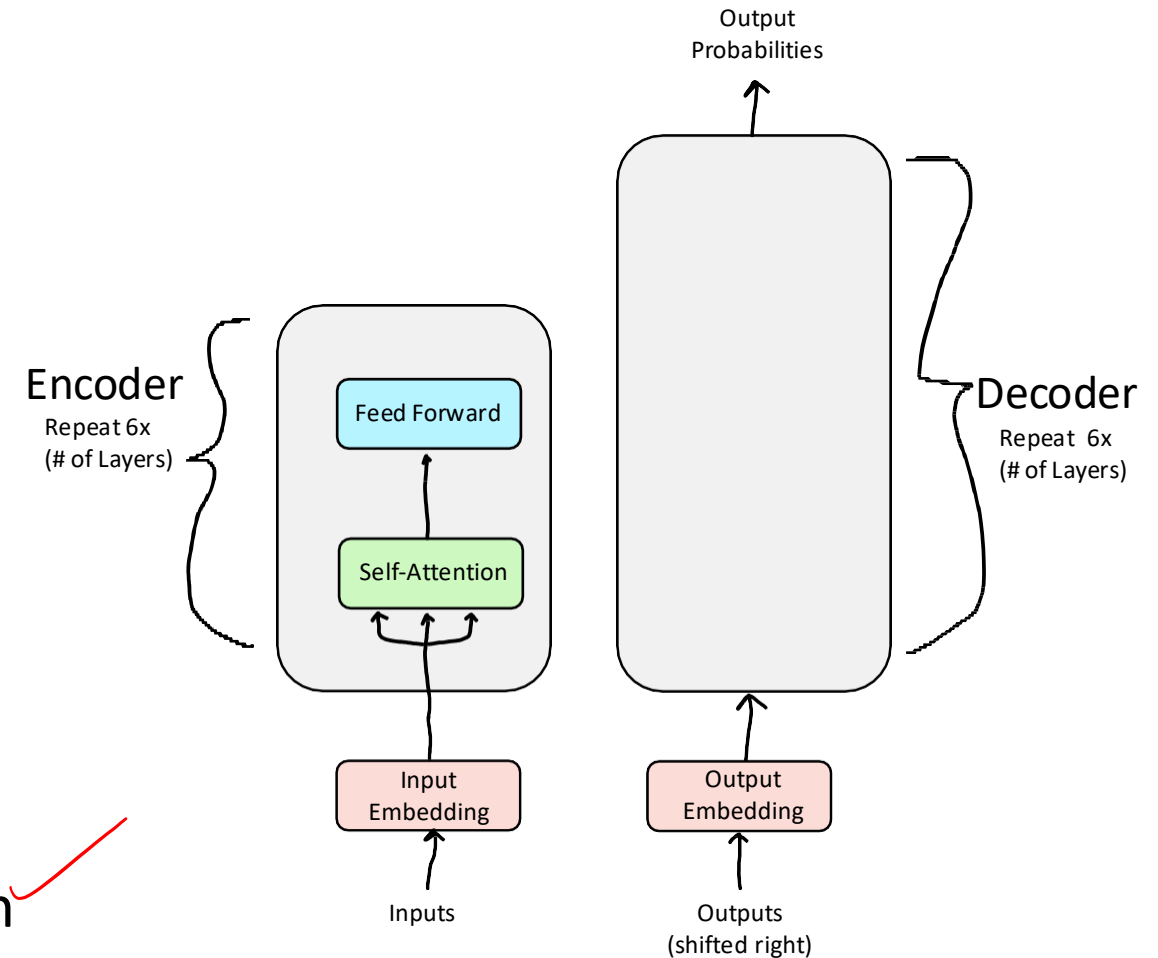
$$m_i = MLP(\text{output}_i) \\ = W_2 * \text{ReLU}(W_1 \times \text{output}_i + b_1) + b_2$$



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 ✓



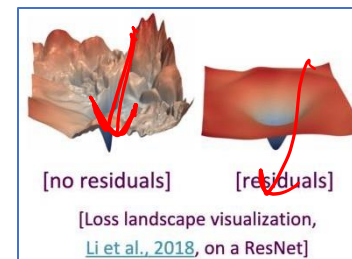
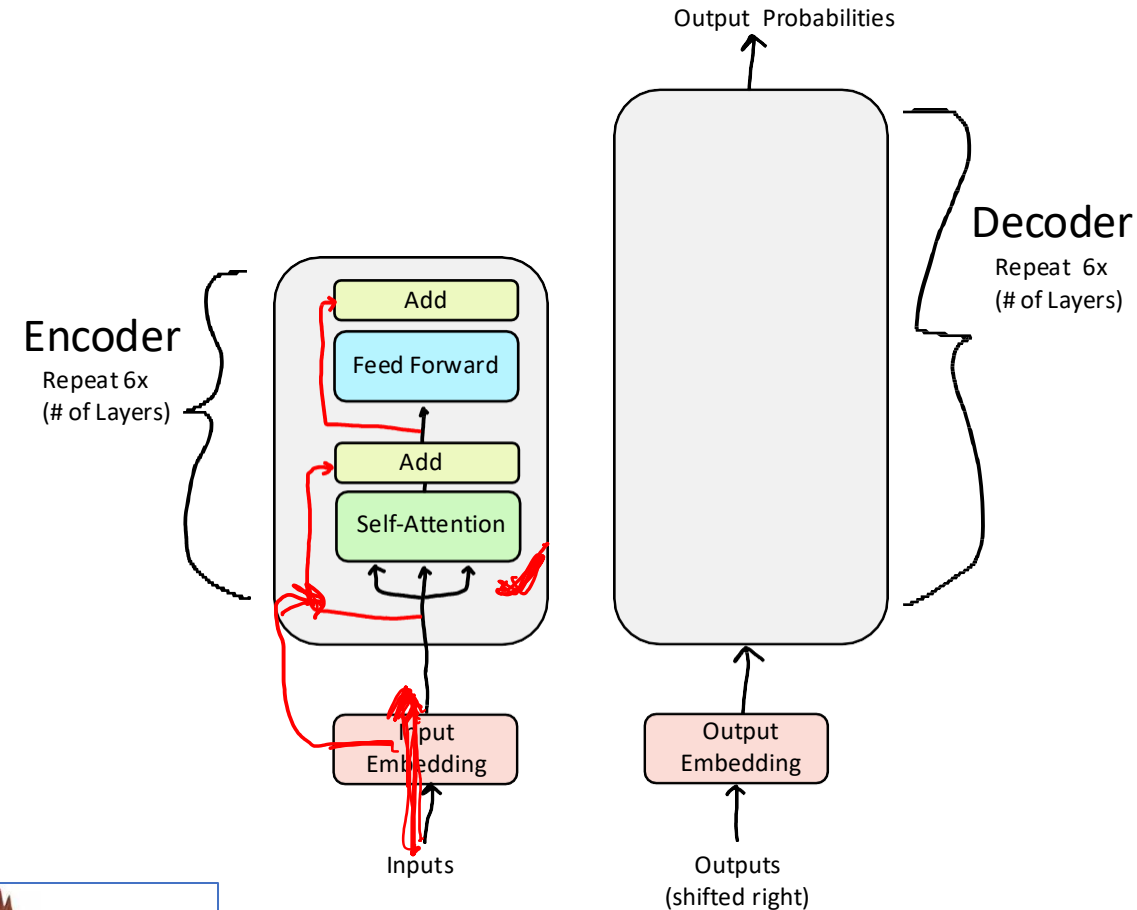
Training Trick #1: Residual Connections [He et al., 2016]

- 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!

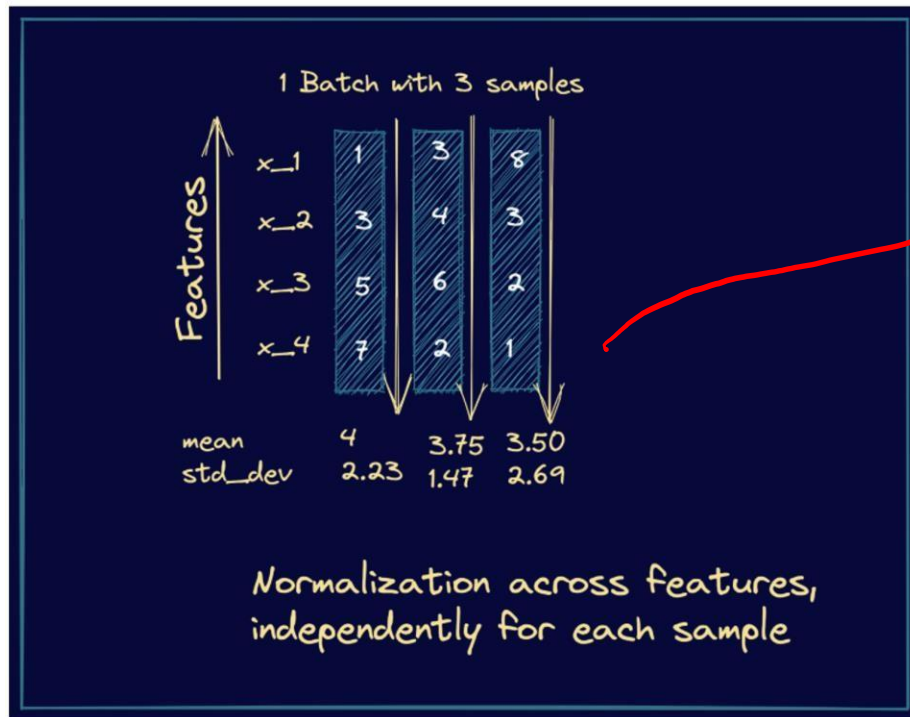
$$x_\ell = F(x_{\ell-1}) + x_{\ell-1}$$

- 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!



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

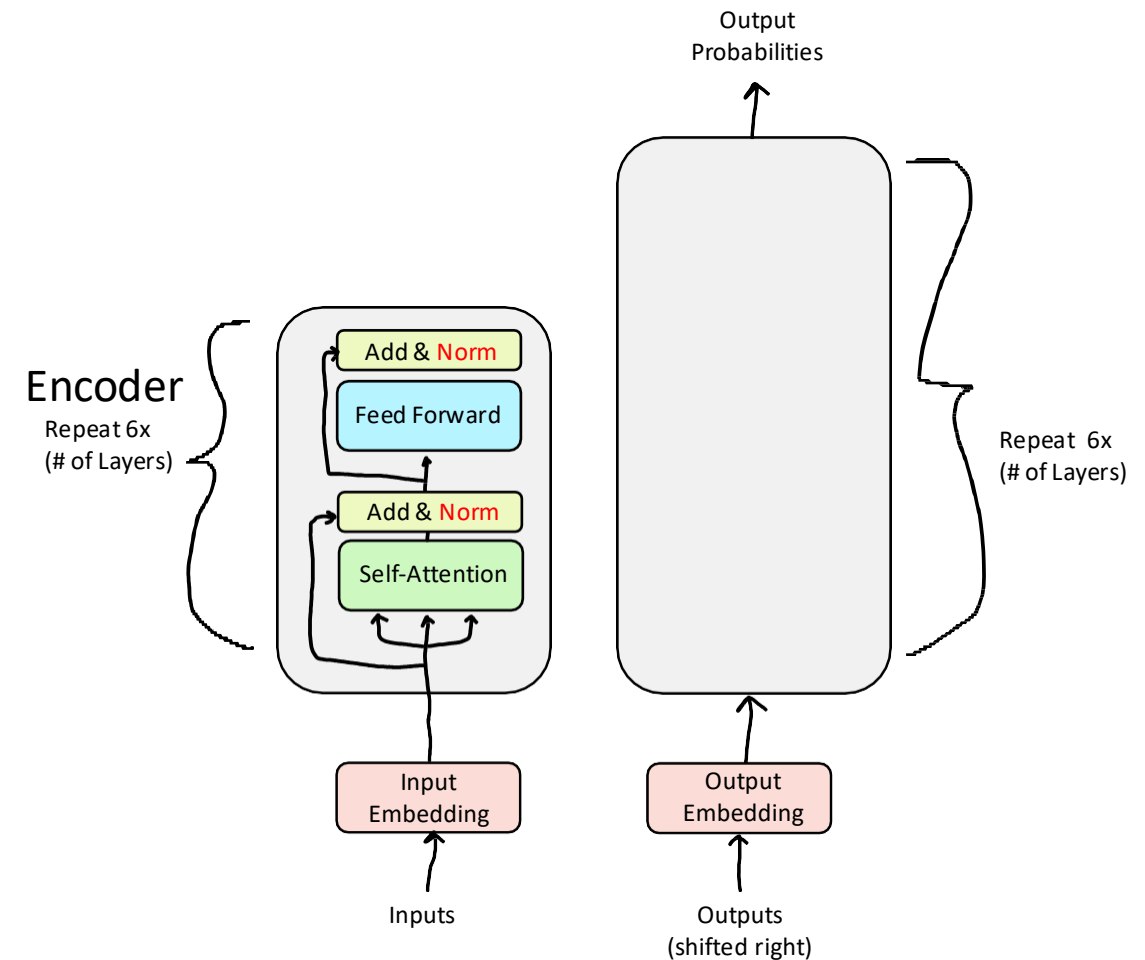


An Example of How LayerNorm Works (Image by Bala Priya C, Pinecone)

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

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

Hidden layers



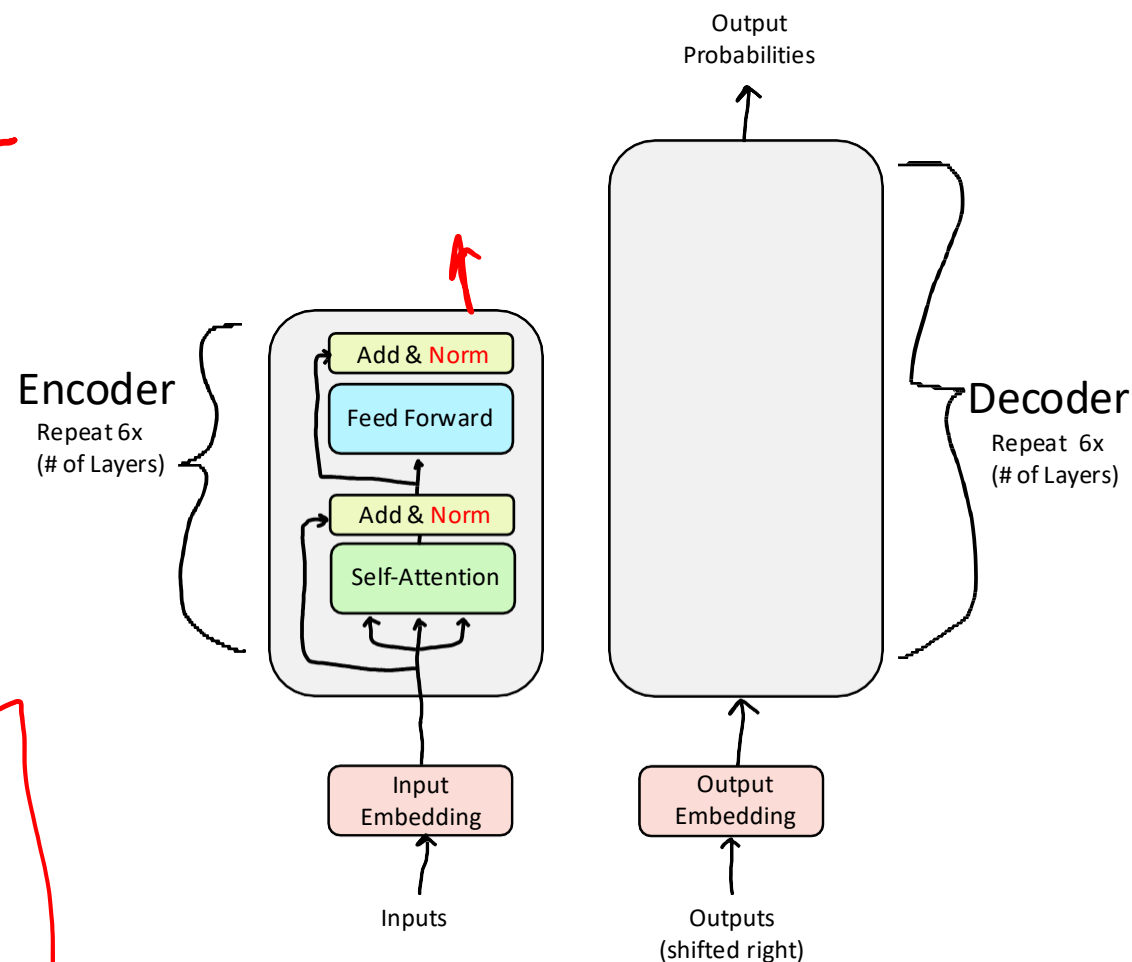
Training Trick #2: Layer Normalization [Ba et al., 2016]

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

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

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

Mean ✓
variance ✓
SD ✓



Training Trick #3: Scaled Dot Product Attention

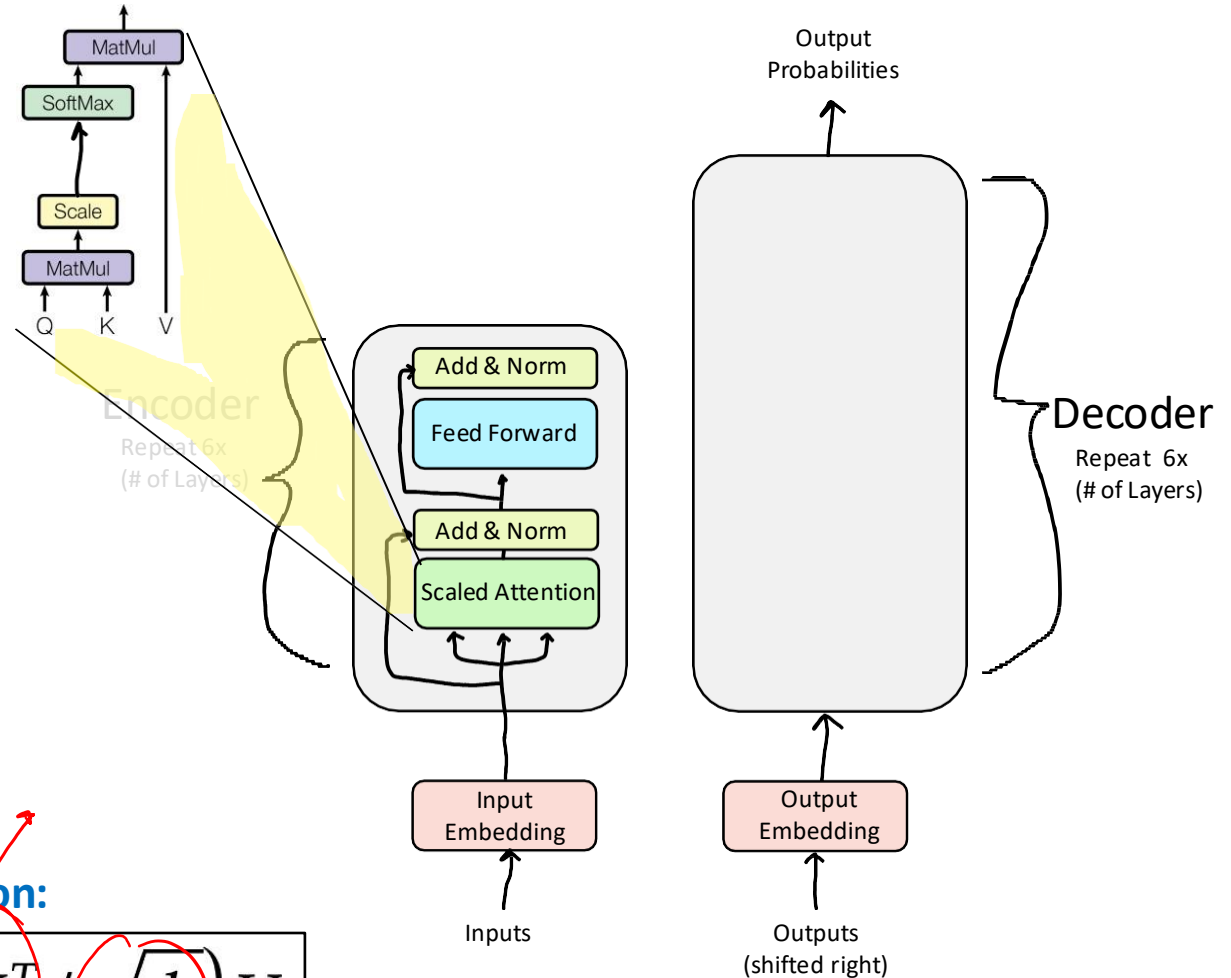
- After LayerNorm, the mean and variance of vector elements is 0 and 1, respectively. (Yay!)
- However, the dot product still tends to take on extreme values, as its variance scales with dimensionality d_k

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:

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

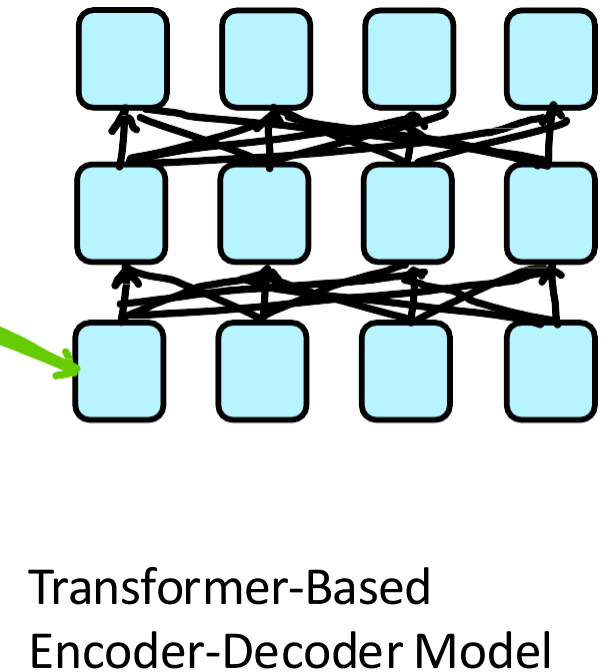
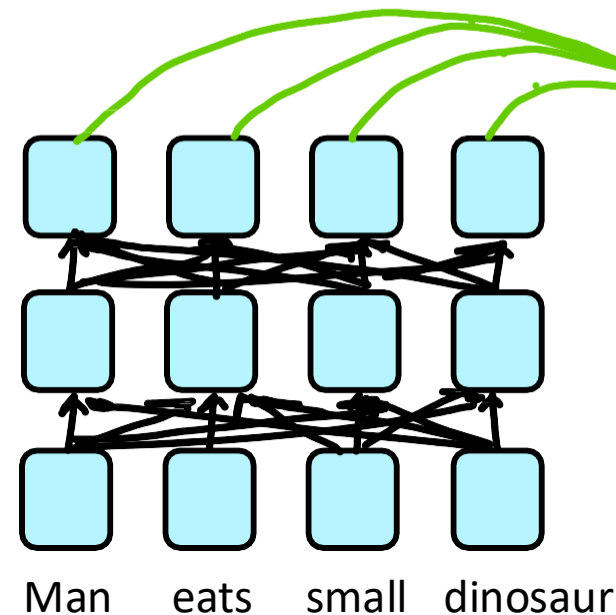


Almost done with Encoder?
What is the Major Problem here?
Can you spot it?

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

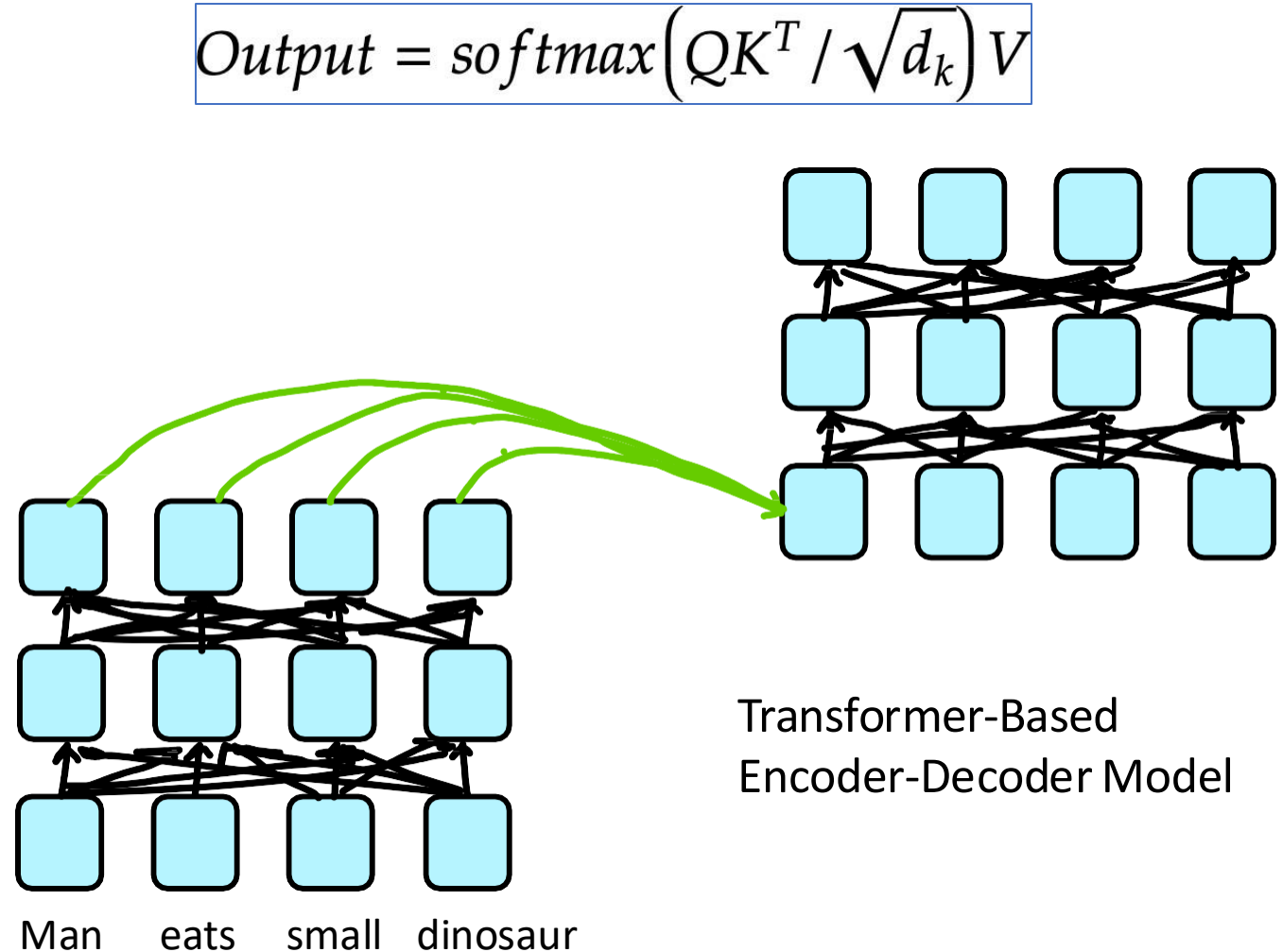
Consider this sentence:

- **"Man eats small dinosaur."**

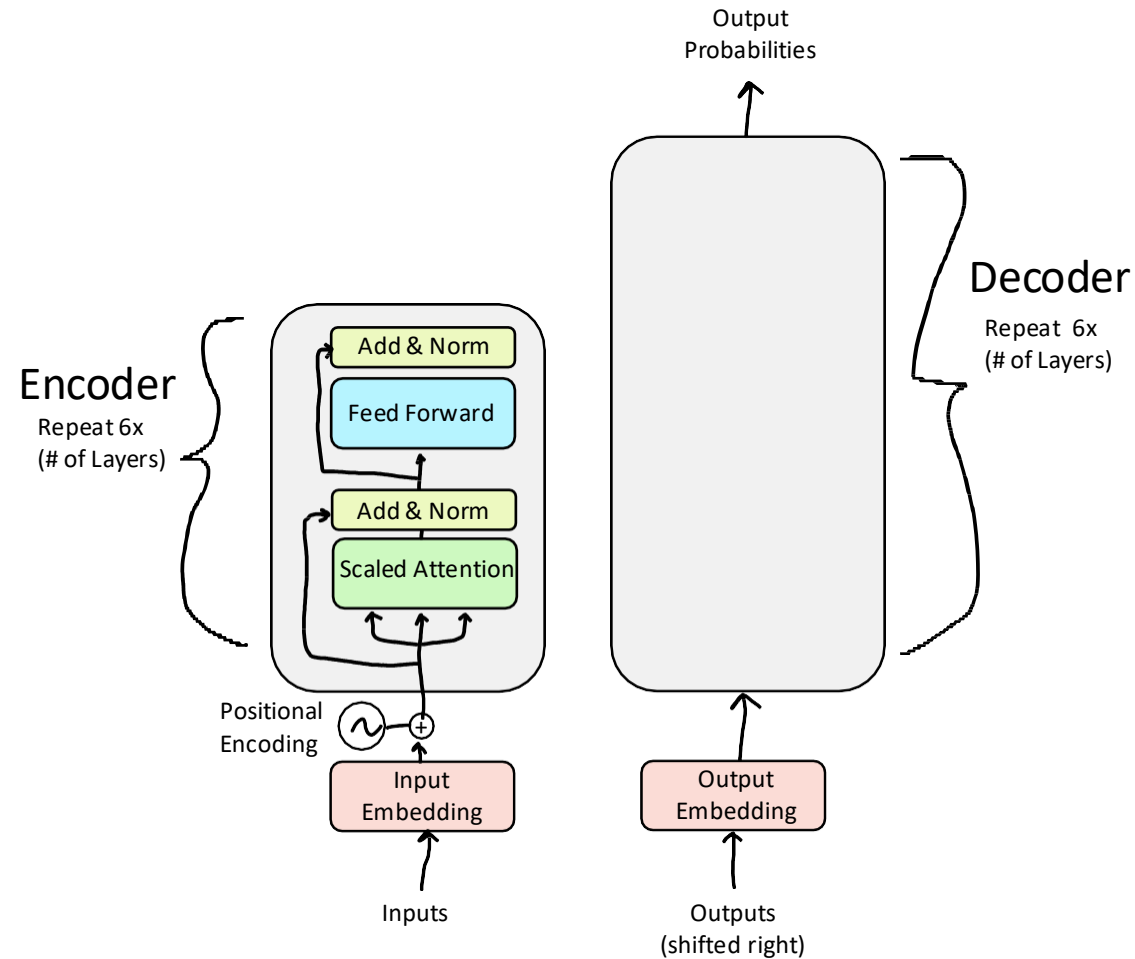


Major issue!

- 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!**



Solution: Inject Order Information through Positional Encodings!



Fixing the first self-attention problem: sequence order

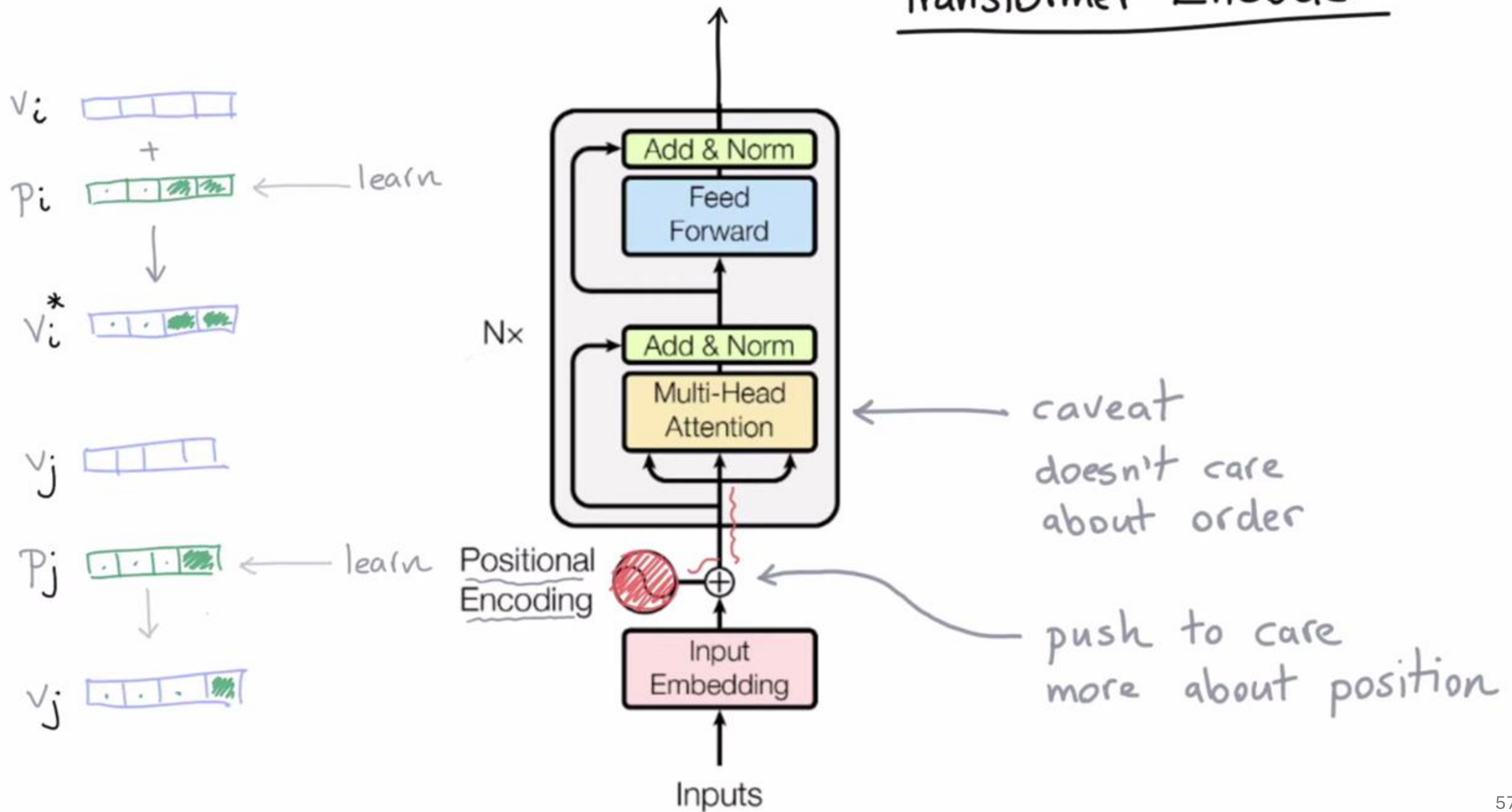
- Since self-attention doesn't build in order information, we need to encode the order of the sentence in our keys, queries, and values.
- Consider representing each **sequence index** as a **vector**

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

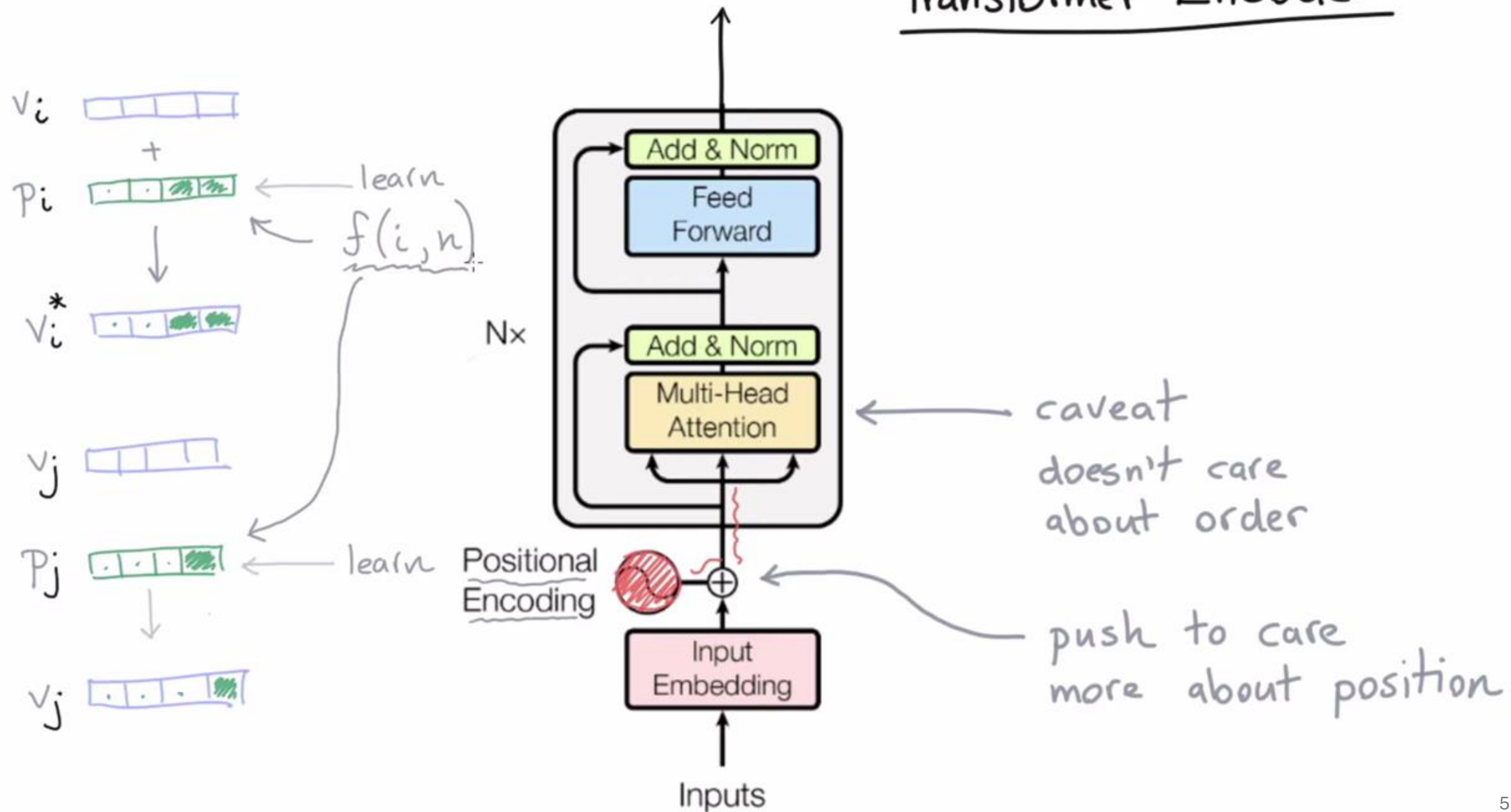
- Don't worry about what the p_i are made of yet!
- Easy to incorporate this info into our self-attention block: just add the p_i to our inputs!

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

Transformer Encoder



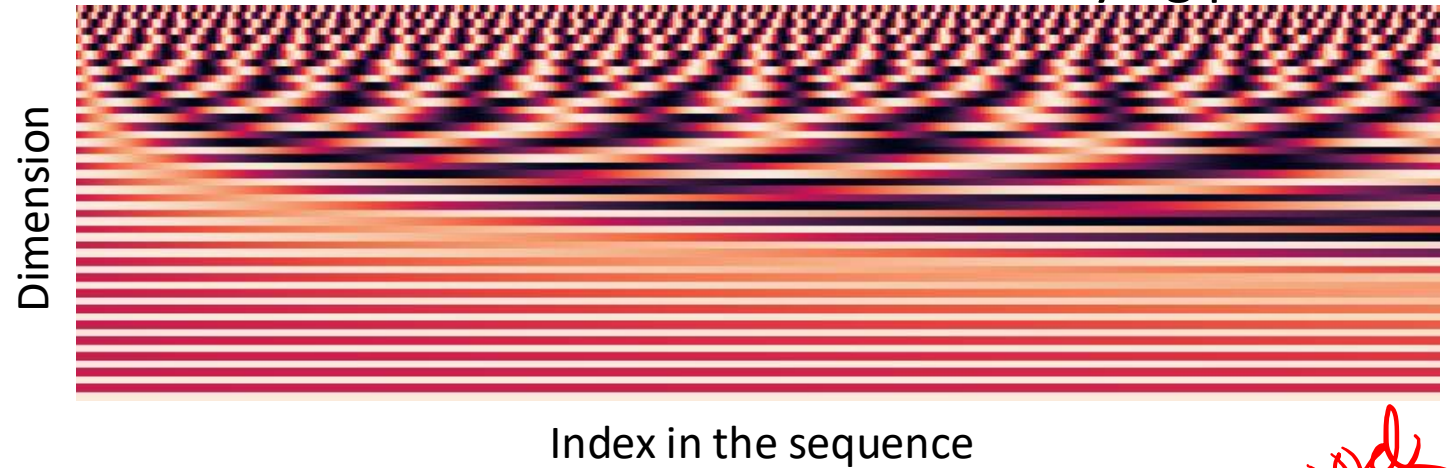
Transformer Encoder



Position representation vectors through sinusoids (original)

✓ Sinusoidal position representations: concatenate sinusoidal functions of varying periods:

$$p_i = \begin{pmatrix} \sin(i/10000^{2*1/d}) \\ \cos(i/10000^{2*1/d}) \\ \vdots \\ \sin(i/10000^{2*d/4}) \\ \cos(i/10000^{2*d/4}) \end{pmatrix}$$



- 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

each word p_i

Extension: Self-Attention w/ Relative Position Encodings

Key Insight: The most salient position information is the relationship (e.g. “cat” is the word before “eat”) between words, rather than their absolute position (e.g. “cat” is word 2).

Original Self-Attention Output:

$$z_i = \sum_{j=1}^n \alpha_{ij} (x_j W^V)$$

where $\alpha_{ij} = \frac{\exp e_{ij}}{\sum_{k=1}^n \exp e_{ik}}$

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

Handwritten red notes and diagrams for the original self-attention:

- A red circle around the z_i equation.
- A red circle around the α_{ij} equation.
- A red circle around the e_{ij} equation.
- A red circle around the $(Q \ K)^T \times V$ expression.
- A red circle around the $\sqrt{d_z}$ term.
- A red arrow pointing from the e_{ij} equation to the α_{ij} equation.
- A red arrow pointing from the α_{ij} equation to the z_i equation.
- A red arrow pointing from the $(Q \ K)^T \times V$ expression to the e_{ij} equation.
- A red arrow pointing from the $\sqrt{d_z}$ term to the e_{ij} equation.

Relation-Aware Self-Attention Output:

$$z_i = \sum_{j=1}^n \alpha_{ij} (x_j W^V + a_{ij}^V)$$

where $\alpha_{ij} = \frac{\exp e_{ij}}{\sum_{k=1}^n \exp e_{ik}}$

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

$$a_{ij}^K = w_{\text{clip}(j-i, k)}^K$$

$$a_{ij}^V = w_{\text{clip}(j-i, k)}^V$$

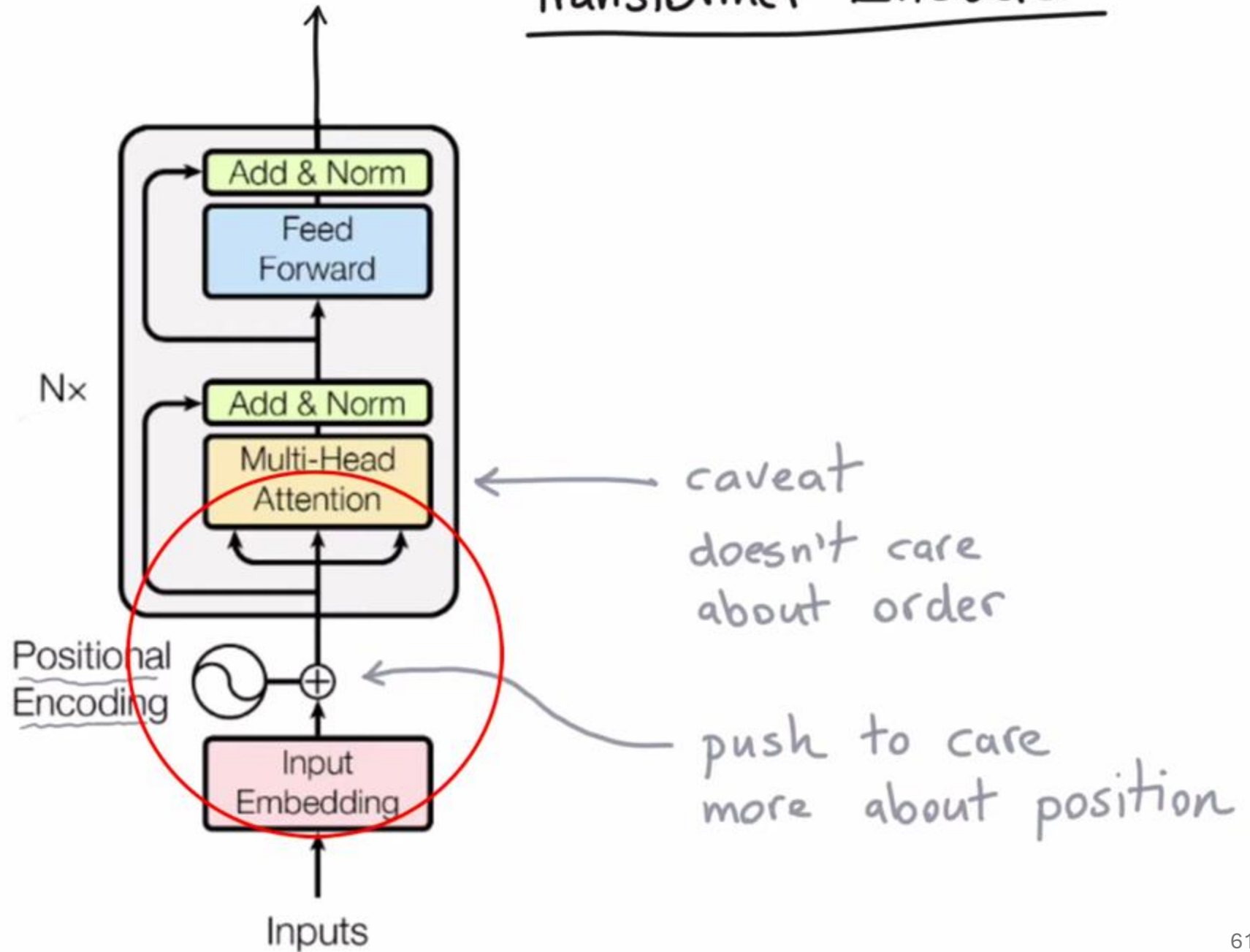
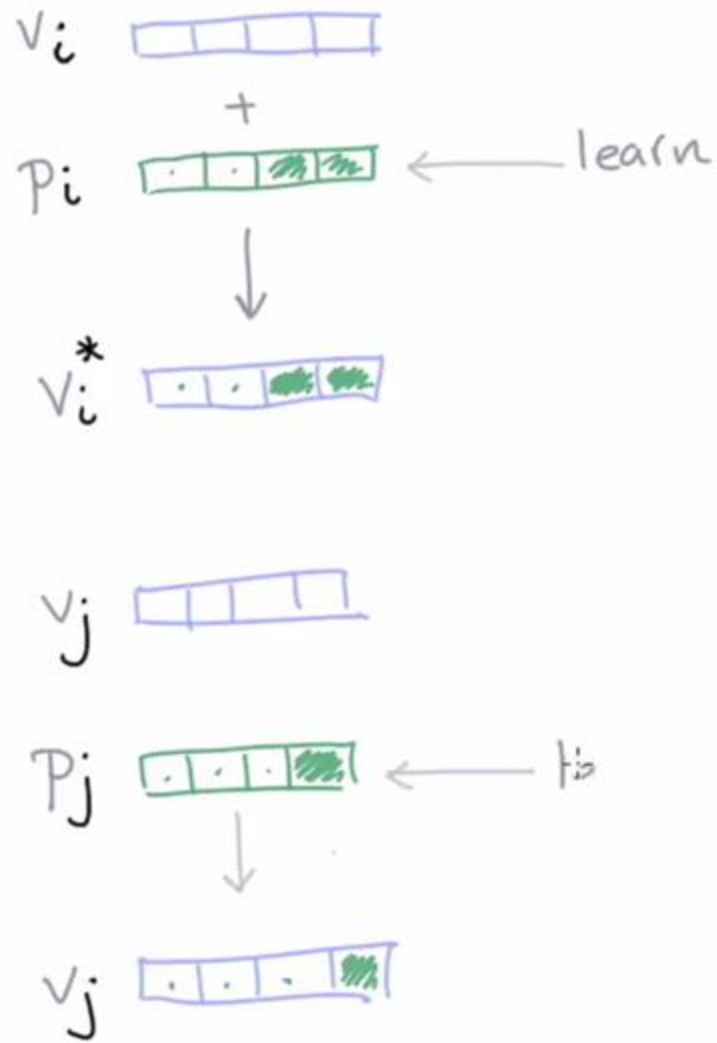
$$\text{clip}(x, k) = \max(-k, \min(k, x))$$

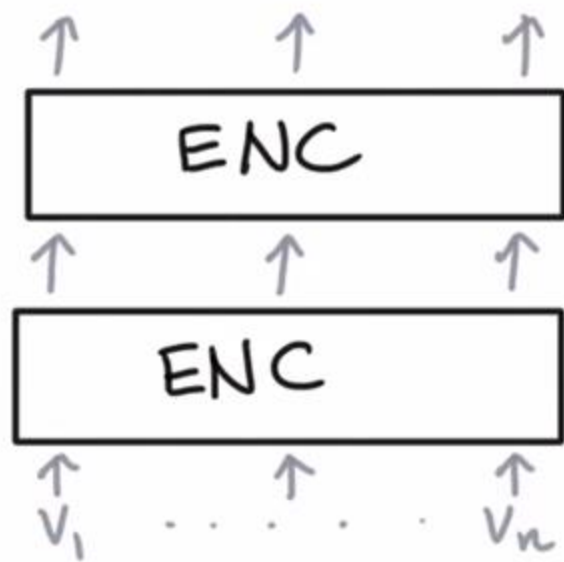
k	EN-DE BLEU
0	12.5
1	25.5
2	25.8
4	25.9
16	25.8
64	25.9
256	25.8

We then learn relative position representations $w^K = (w_{-k}^K, \dots, w_k^K)$ and $w^V = (w_{-k}^V, \dots, w_k^V)$

[Table and Equations From \[Shaw et al., 2018\]](#)

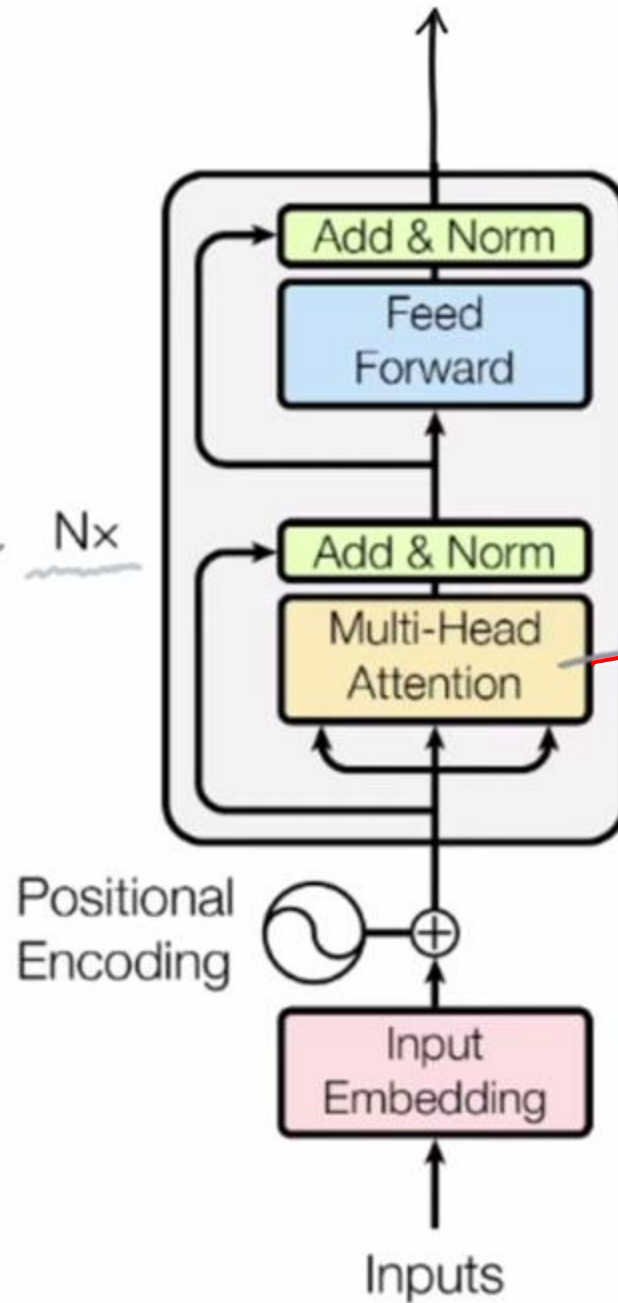
Transformer Encoder





$N \times$

hyper param

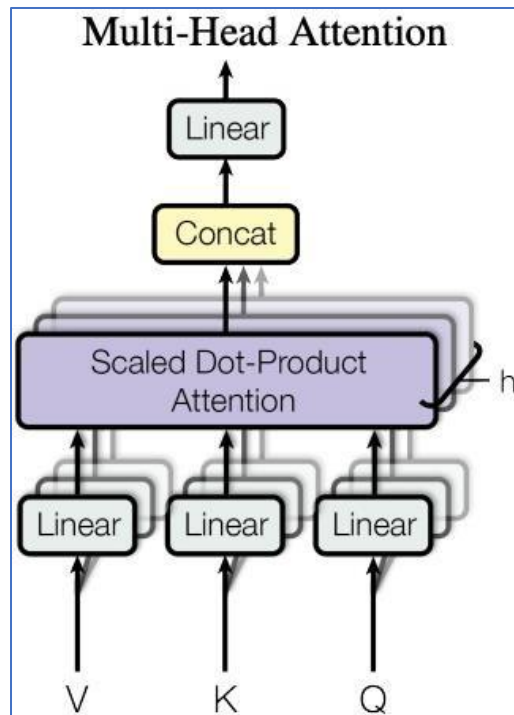


Transformer Encoder

number of heads
hyper param:-

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.

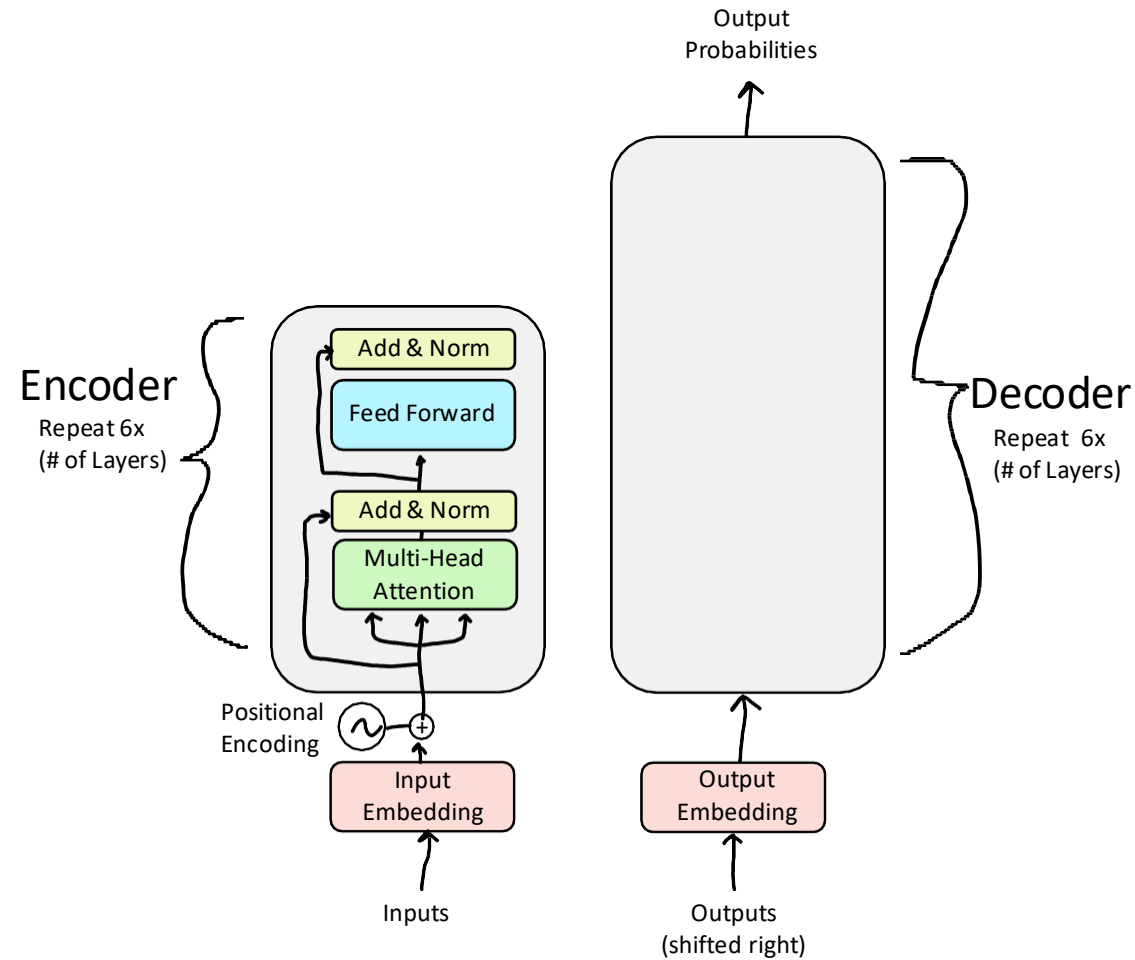


[Vaswani et al. 2017]



Wizards of the Coast, Artist: Todd Lockwood

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



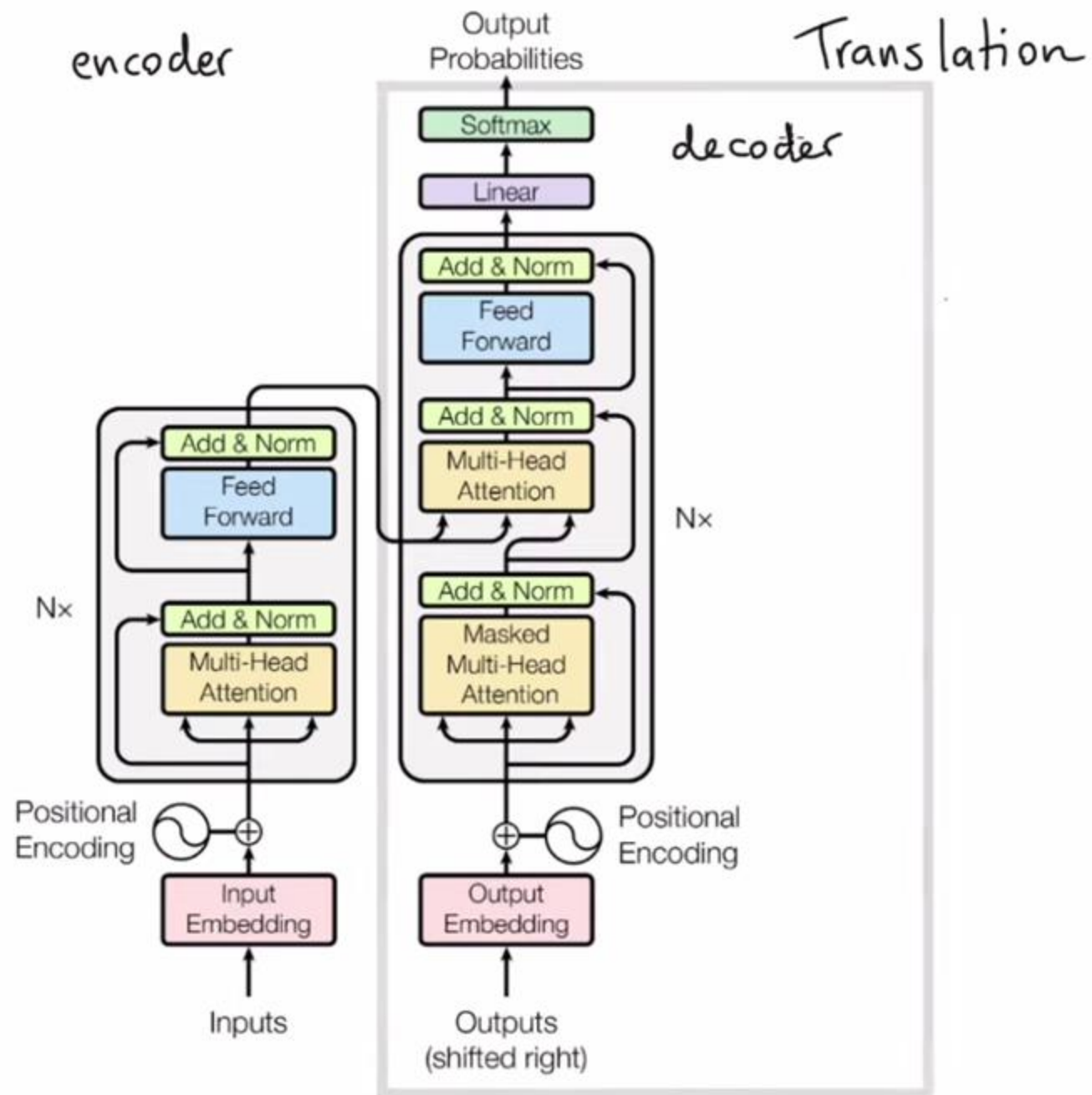


Figure 1: The Transformer - model architecture.

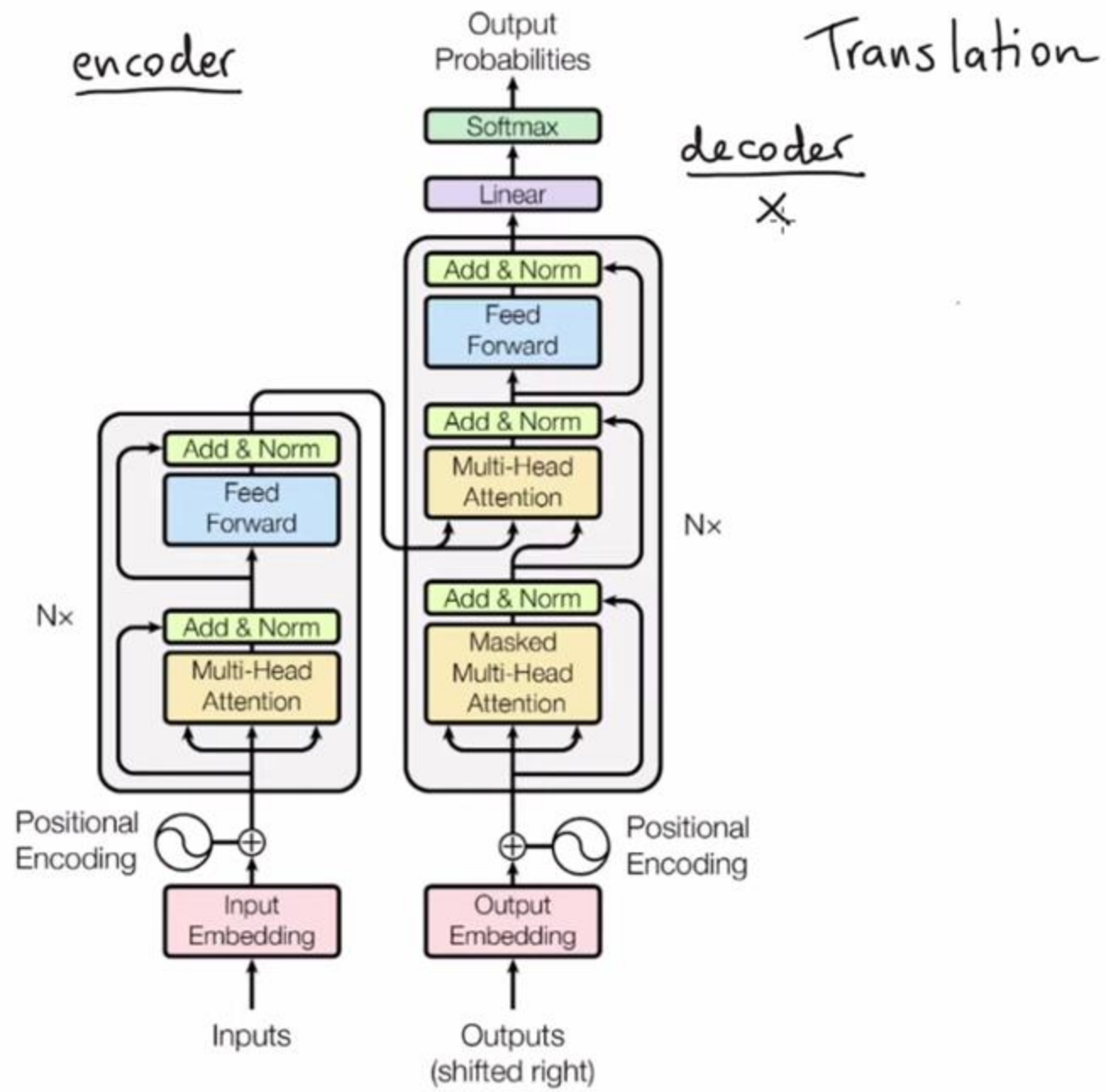
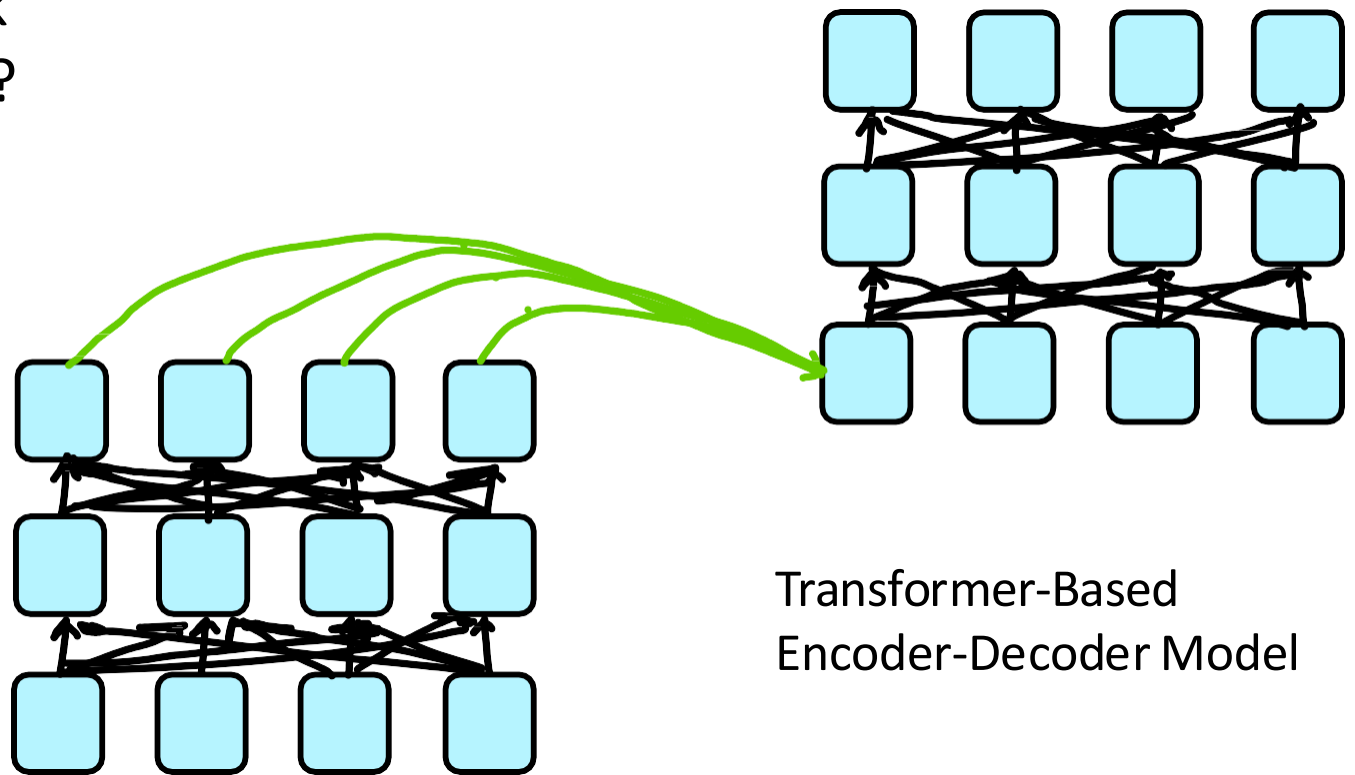


Figure 1: The Transformer - model architecture.

Decoder: Masked Multi-Head Self-Attention

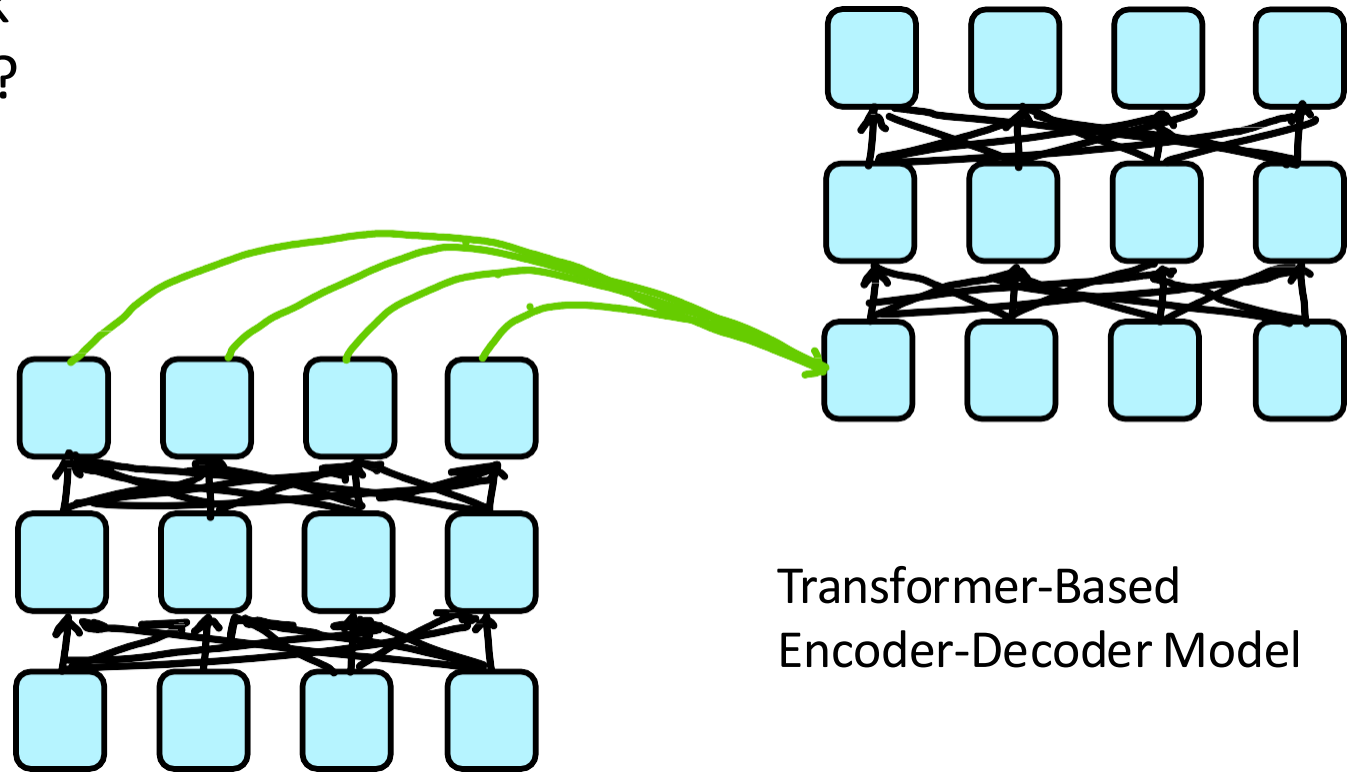
- **Problem:** How do we keep the decoder from “cheating”? If we have a language modeling objective, can't the network just look ahead and "see" the answer?



Transformer-Based
Encoder-Decoder Model

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

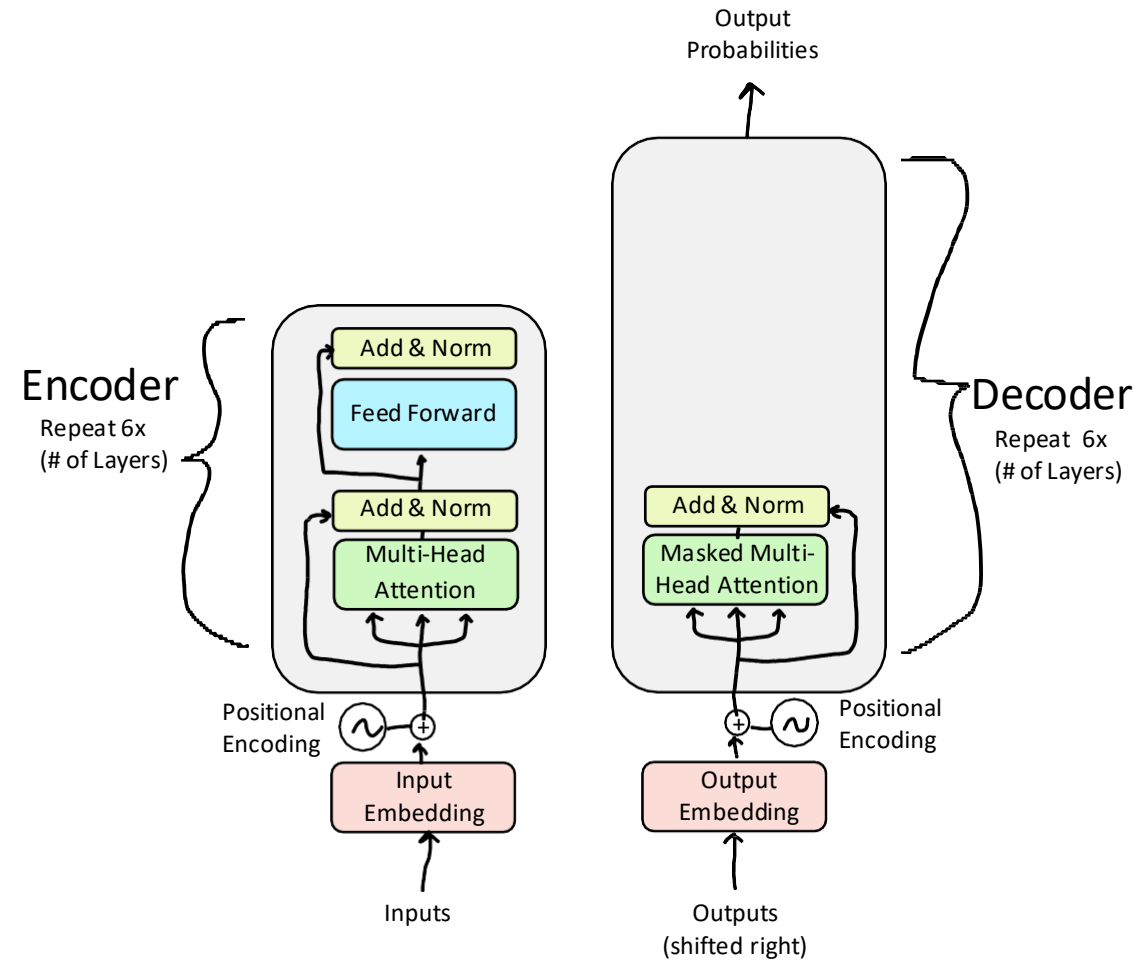
- To use self-attention in **decoders**, we need to ensure we can't peek at the future. ✓
- At every timestep, we could change the set of keys and queries to include only past words. (Inefficient!)
- To enable parallelization, we **mask out attention** to future words by setting attention scores to $-\infty$.

We can look at these
(not greyed out) words

For encoding these words

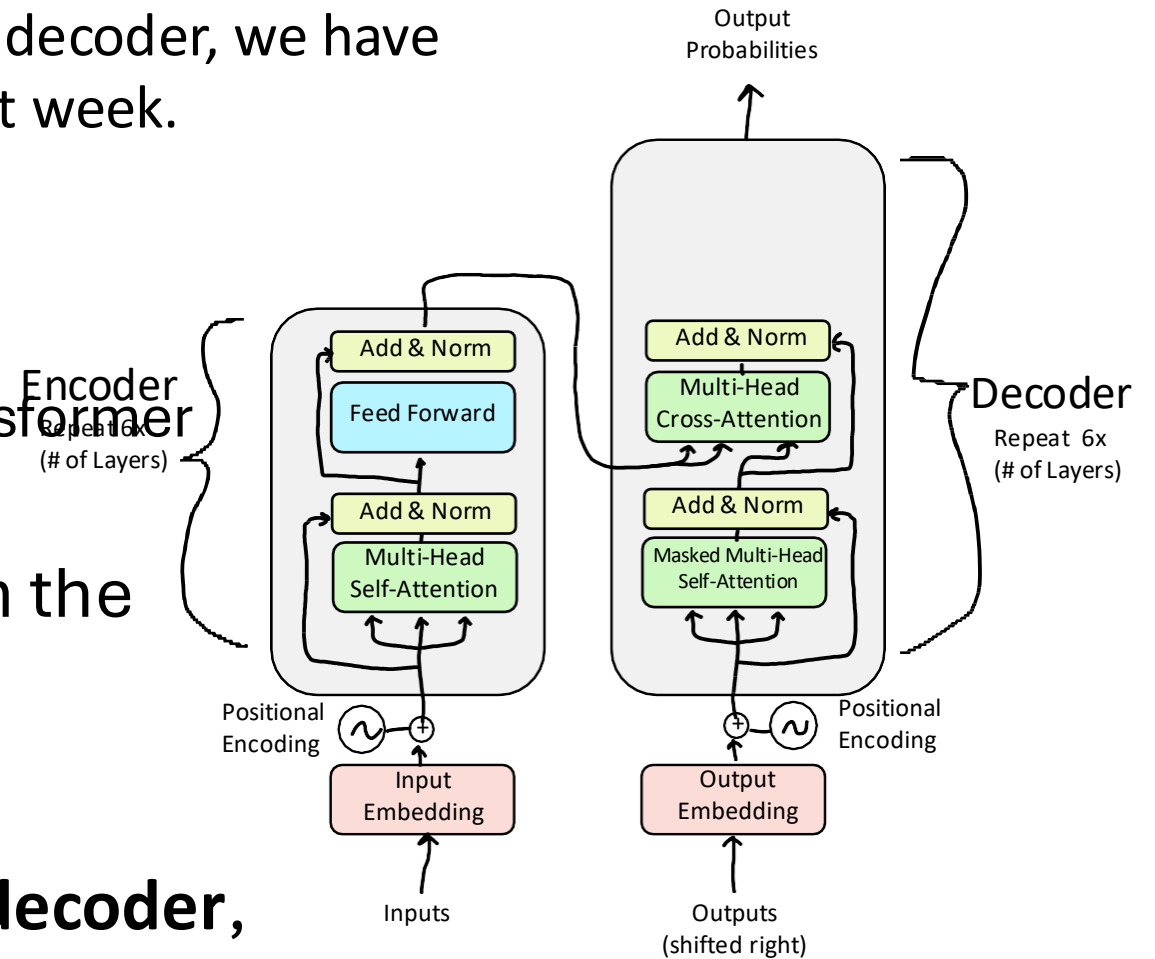
	[START]	The	chef	who
[START]	$-\infty$	$-\infty$	$-\infty$	$-\infty$
The	✓	$-\infty$	$-\infty$	$-\infty$
chef	✓	✓	$-\infty$	$-\infty$
who	✓	✓	✓	$-\infty$

Decoder: Masked Multi-Headed Self-Attention

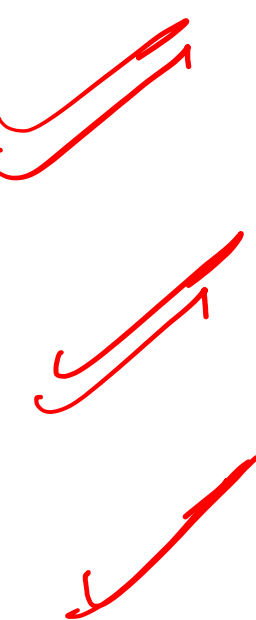
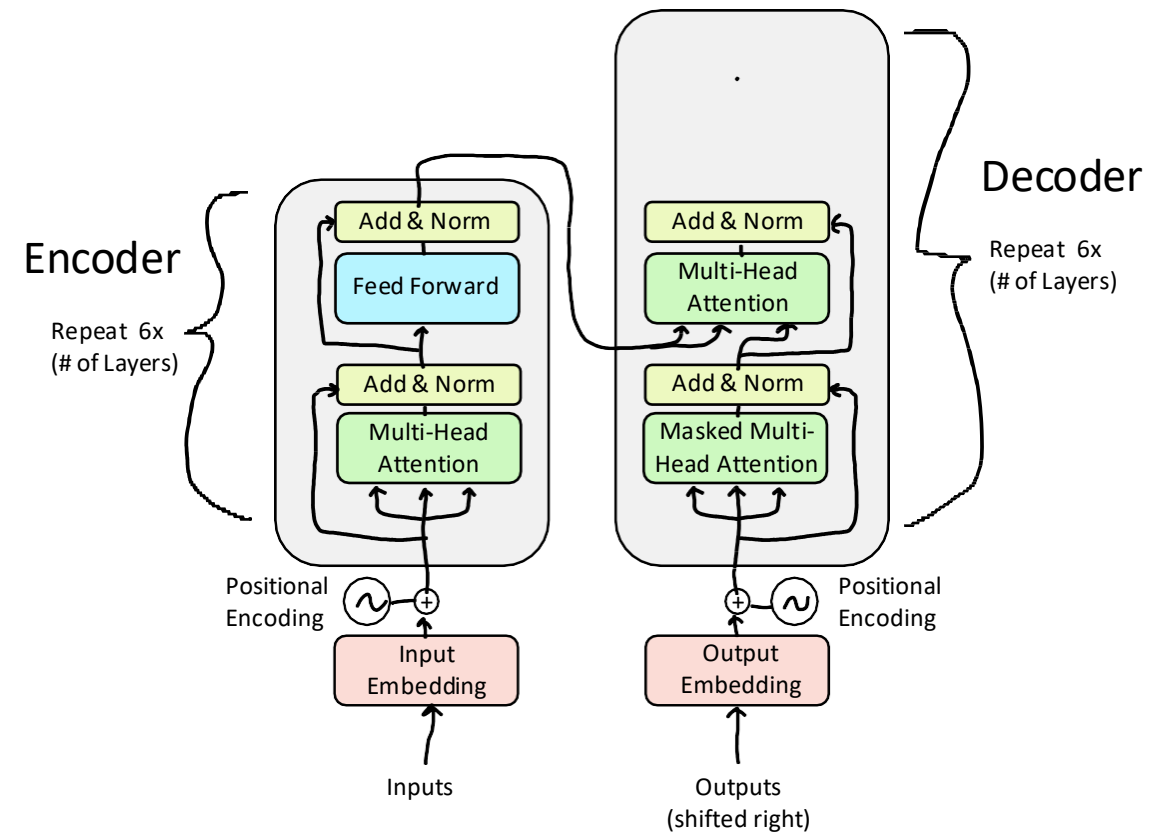


Encoder-Decoder Attention

- 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 what we saw last week.
- Let $h_1, \dots, h_T$ be **output** vectors **from** the
- **Transformer encoder**; $x_i \in \mathbb{R}^d$
- Let $z_1, \dots, z_T$ be input vectors from the **Transformer decoder**, $z_i \in \mathbb{R}^d$
- Then keys and values are drawn from the
- **encoder** (like a memory):
 - $k_i = Kh_i$, $v_i = Vh_i$.
- And the queries are drawn from the **decoder**,
 $q_i = Qz$.

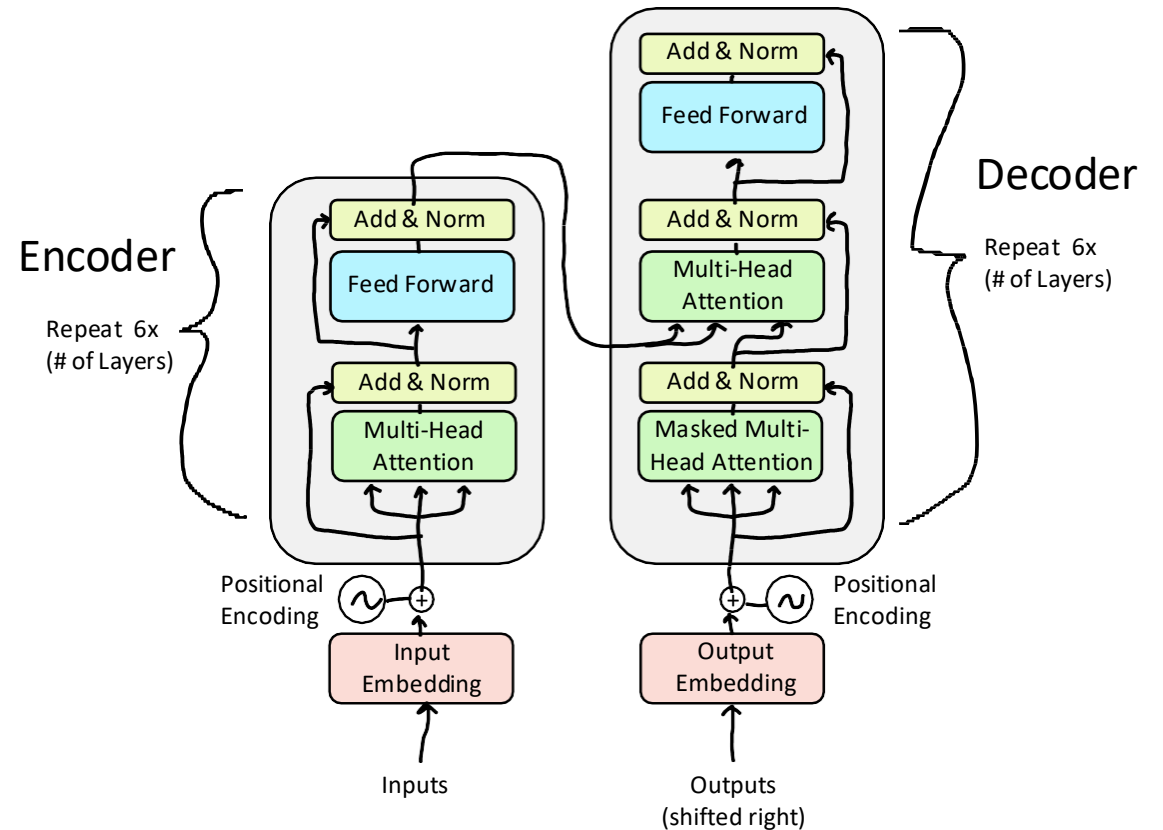


Decoder: Finishing touches!



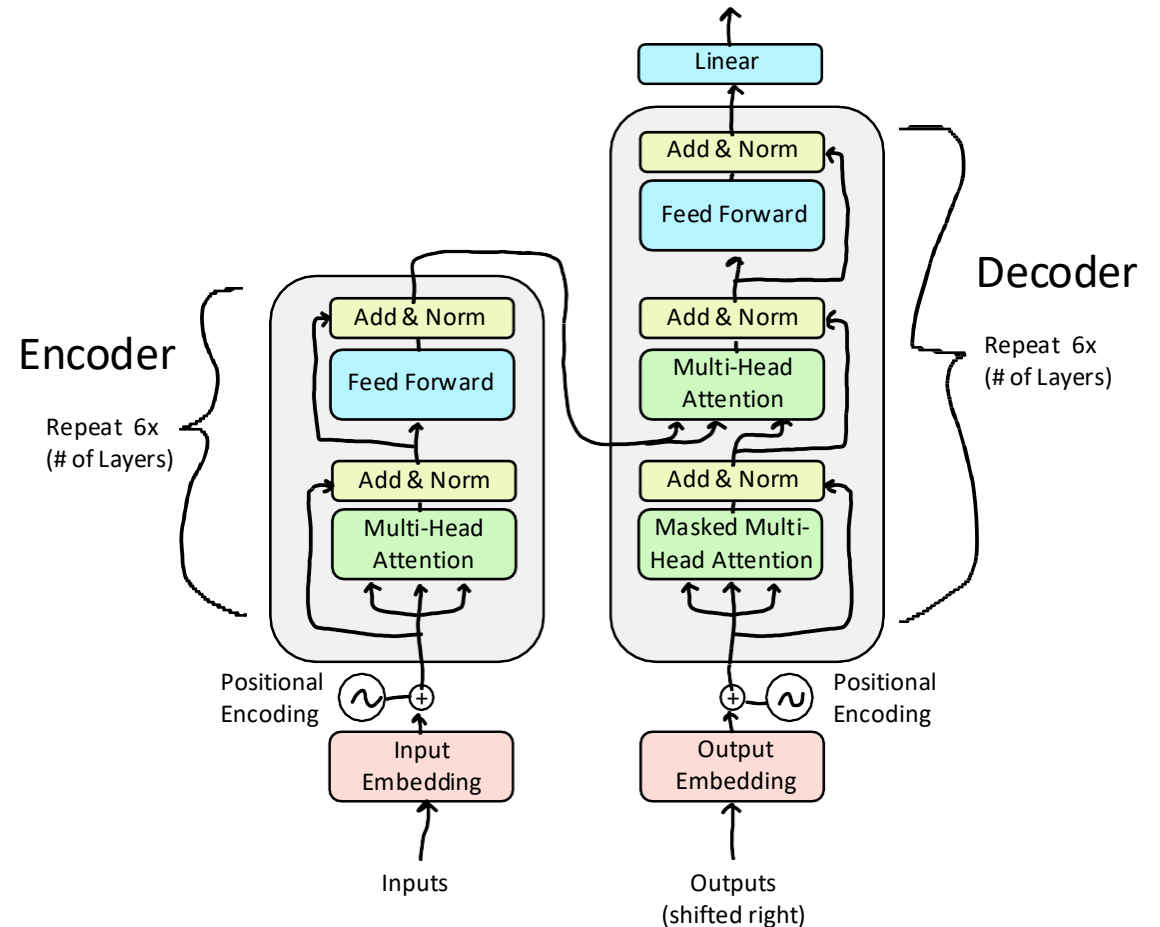
Decoder: Finishing touches!

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



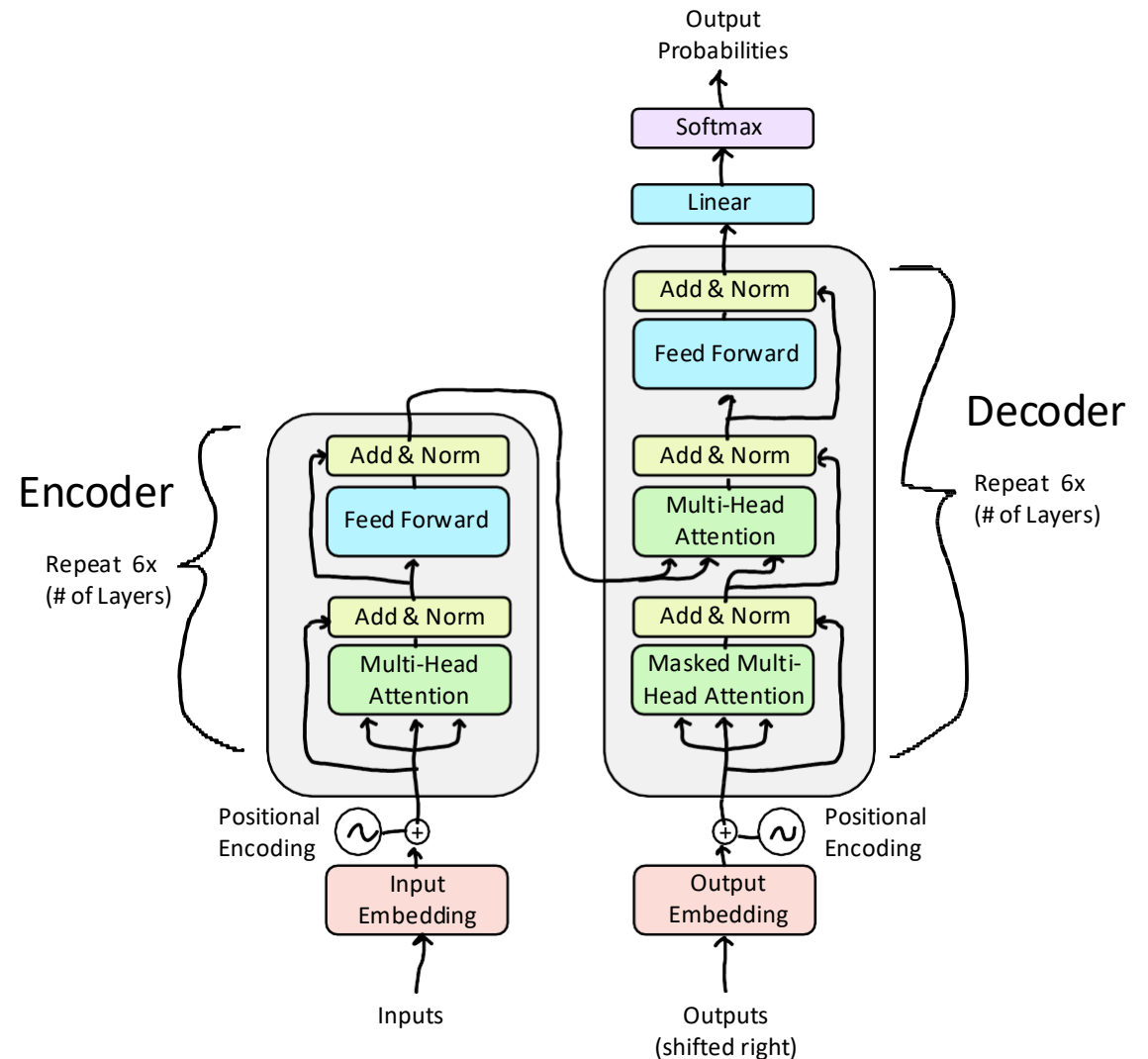
Decoder: Finishing touches!

- 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)

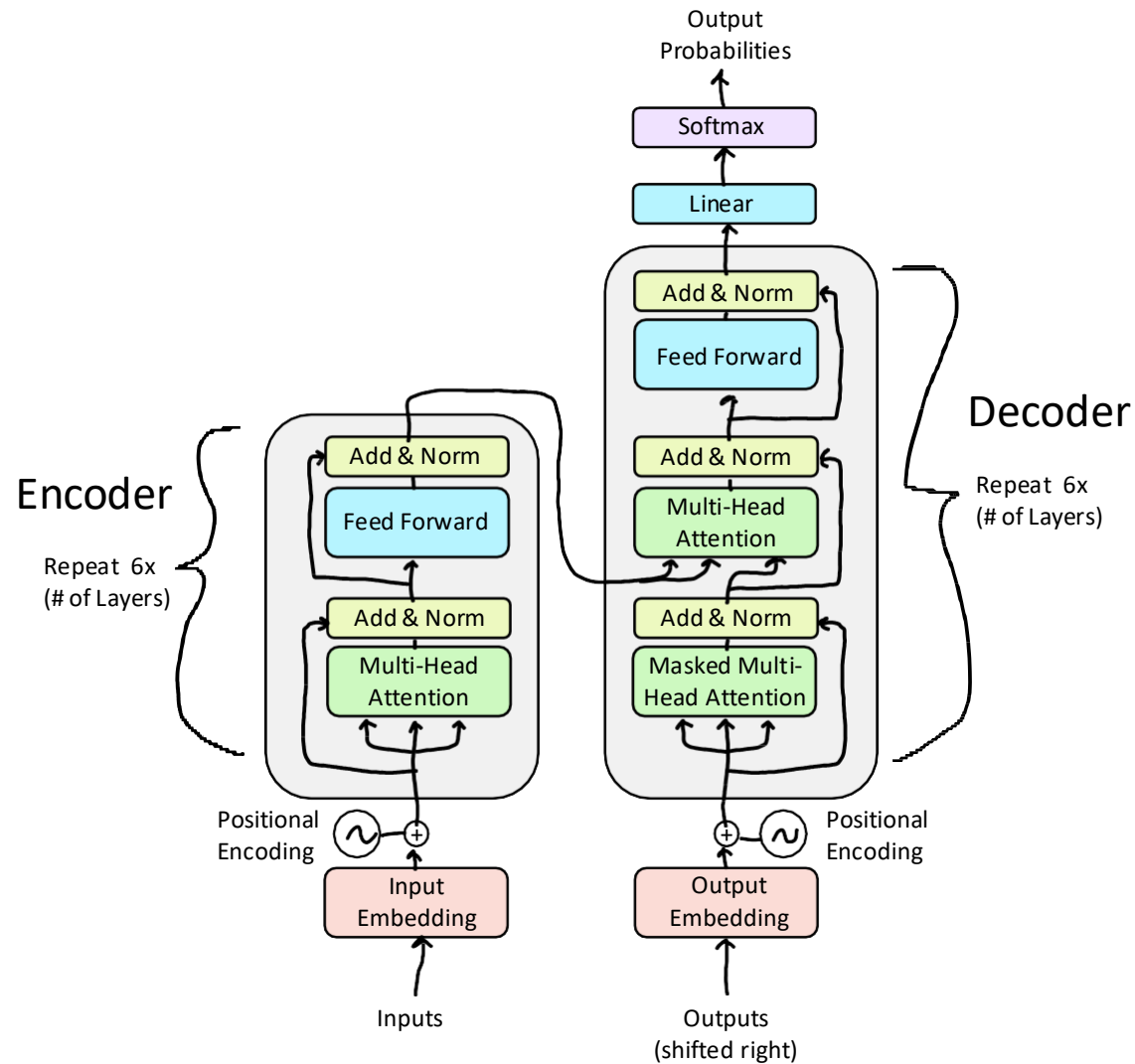


Decoder: Finishing touches!

- 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 a probability distribution of possible next words!



Recap of Transformer Architecture



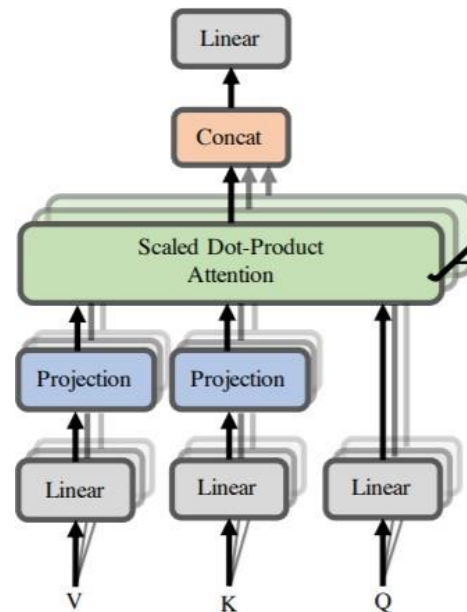
What would we like to fix about the Transformer?

- **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?
 - As we learned: Relative linear position attention [\[Shaw et al., 2018\]](#)
 - Dependency syntax-based position [\[Wang et al., 2019\]](#)
 - Rotary Embeddings [\[Su et al., 2021\]](#)

Recent work on improving on quadratic self-attention cost

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

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

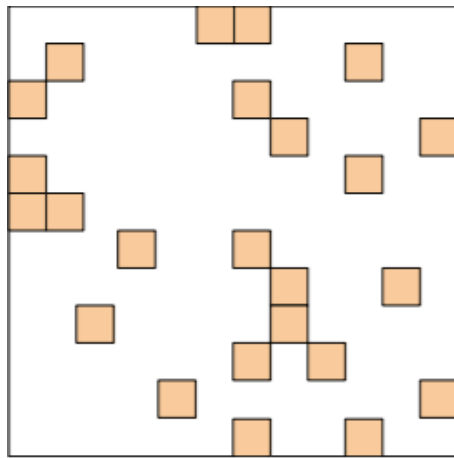


Sequence length / batch size

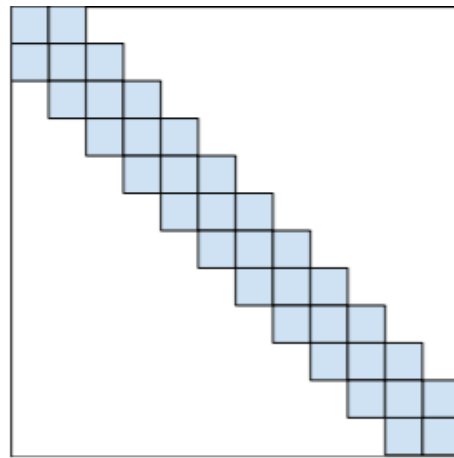
Recent work on improving on quadratic self-attention cost

- 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** [\[Zaheer et al., 2021\]](#)

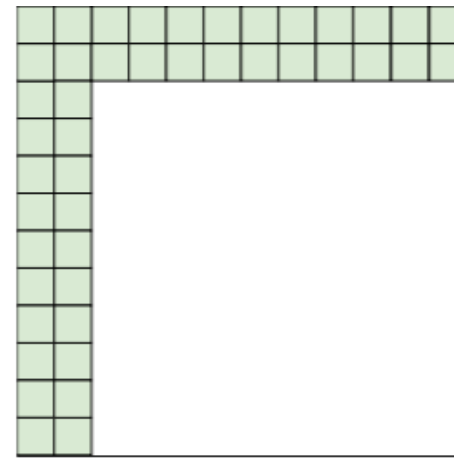
Key idea: replace all-pairs interactions with a family of other interactions, **like local windows, looking at everything, and random interactions.**



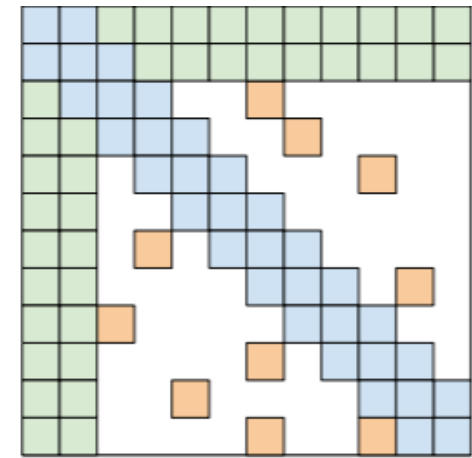
(a) Random attention



(b) Window attention



(c) Global Attention



(d) BIGBIRD

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	223M	11.1T	3.50	2.182 ± 0.005	1.838	71.66	17.78	23.02	26.62
GeLU	223M	11.1T	3.58	2.179 ± 0.003	1.838	75.79	17.86	25.13	26.47
Swish	223M	11.1T	3.62	2.186 ± 0.003	1.847	73.77	17.74	24.34	26.75
ELU	223M	11.1T	3.56	2.270 ± 0.007	1.932	67.83	16.73	23.02	26.08
GLU	223M	11.1T	3.59	2.174 ± 0.003	1.814	74.20	17.42	24.31	27.12
GeGLU	223M	11.1T	3.55	2.130 ± 0.006	1.792	75.96	18.27	24.87	26.87
ReGLU	223M	11.1T	3.57	2.145 ± 0.004	1.803	76.17	18.36	24.87	27.02
Selu	223M	11.1T	3.55	2.315 ± 0.004	1.948	68.76	16.76	22.75	25.99
SwiGLU	223M	11.1T	3.53	2.127 ± 0.003	1.789	76.00	18.20	24.34	27.02
LiGLU	223M	11.1T	3.59	2.149 ± 0.005	1.798	75.34	17.97	24.34	26.53
Sigmoid	223M	11.1T	3.63	2.291 ± 0.019	1.867	74.31	17.51	23.02	26.30
Softplus	223M	11.1T	3.47	2.207 ± 0.011	1.850	72.45	17.65	24.34	26.89
RMS Norm	223M	11.1T	3.68	2.167 ± 0.008	1.821	75.45	17.94	24.07	27.14
Resero	223M	11.1T	3.51	2.262 ± 0.003	1.939	61.69	15.64	20.90	26.37
Resero + LayerNorm	223M	11.1T	3.26	2.223 ± 0.006	1.858	70.42	17.58	23.02	26.29
Resero + RMS Norm	223M	11.1T	3.34	2.221 ± 0.009	1.875	70.33	17.32	23.02	26.19
Fluip	223M	11.1T	2.95	2.382 ± 0.012	2.067	58.56	14.42	23.02	26.31
24 layers, $d_v = 1536, H = 6$	224M	11.1T	3.33	2.200 ± 0.007	1.843	74.89	17.75	25.13	26.89
18 layers, $d_v = 2048, H = 8$	223M	11.1T	3.38	2.185 ± 0.005	1.831	76.45	16.83	24.34	27.10
8 layers, $d_v = 4608, H = 18$	223M	11.1T	3.69	2.190 ± 0.005	1.847	74.58	17.69	23.28	26.85
6 layers, $d_v = 6144, H = 24$	223M	11.1T	3.70	2.201 ± 0.010	1.857	73.55	17.59	24.60	26.66
Block sharing	65M	11.1T	3.91	2.497 ± 0.037	2.164	64.50	14.53	21.96	25.48
+ Factorized embeddings	45M	9.4T	4.21	2.631 ± 0.305	2.183	60.84	14.00	19.84	25.27
+ Factorized & shared embeddings	20M	9.1T	4.37	2.907 ± 0.313	2.385	53.95	11.37	19.84	25.19
Encoder only block sharing	170M	11.1T	3.68	2.298 ± 0.023	1.929	69.60	16.23	23.02	26.23
Decoder only block sharing	144M	11.1T	3.70	2.352 ± 0.029	2.082	67.93	16.13	23.81	26.08
Factorized Embedding	227M	9.4T	3.80	2.208 ± 0.006	1.855	70.41	15.92	22.75	26.50
Factorized & shared embeddings	202M	9.1T	3.92	2.320 ± 0.010	1.952	68.69	16.33	22.22	26.44
Tied encoder/decoder input embeddings	248M	11.1T	3.55	2.192 ± 0.002	1.840	71.70	17.72	24.34	26.49
Tied decoder input and output embeddings	248M	11.1T	3.57	2.187 ± 0.007	1.827	74.86	17.74	24.87	26.67
Unified embeddings	273M	11.1T	3.53	2.195 ± 0.005	1.834	72.99	17.58	23.28	26.48
Adaptive input embeddings	204M	9.2T	3.55	2.250 ± 0.002	1.899	66.57	16.21	24.07	26.66
Adaptive softmax	204M	9.2T	3.60	2.364 ± 0.005	1.982	72.91	16.67	21.16	25.56
Adaptive softmax without projection	223M	10.8T	3.43	2.229 ± 0.009	1.914	71.82	17.10	23.02	25.72
Mixture of softmaxes	232M	16.3T	2.24	2.227 ± 0.017	1.821	76.77	17.62	22.75	26.82
Transparent attention	223M	11.1T	3.33	2.181 ± 0.014	1.874	54.31	10.40	21.16	26.80
Dynamic convolution	257M	11.8T	2.65	2.403 ± 0.009	2.047	58.30	12.67	21.16	17.03
Lightweight convolution	224M	10.4T	4.07	2.370 ± 0.010	1.989	63.07	14.86	23.02	24.73
Enviied Transformer	217M	9.9T	3.09	2.220 ± 0.003	1.863	73.67	10.76	24.07	26.58
Synthesizer (dense)	224M	11.4T	3.47	2.334 ± 0.021	1.962	61.03	14.27	16.14	26.63
Synthesizer (dense plus)	243M	12.6T	3.22	2.191 ± 0.010	1.840	73.98	16.96	23.81	26.71
Synthesizer (dense plus alpha)	243M	12.6T	3.01	2.180 ± 0.007	1.828	74.25	17.02	23.28	26.61
Synthesizer (factorized)	207M	10.1T	3.94	2.341 ± 0.017	1.968	62.78	15.39	23.55	26.42
Synthesizer (random)	254M	10.1T	4.08	2.326 ± 0.012	2.009	54.27	10.35	19.56	26.44
Synthesizer (random plus)	292M	12.0T	3.63	2.189 ± 0.004	1.842	73.32	17.04	24.87	26.43
Synthesizer (random plus alpha)	292M	12.0T	3.42	2.186 ± 0.007	1.828	75.24	17.08	24.08	26.39
Universal Transformer	84M	40.0T	0.88	2.406 ± 0.036	2.053	70.13	14.09	19.05	23.91
Mixture of experts	648M	11.7T	3.20	2.146 ± 0.006	1.785	74.55	18.13	24.08	26.94
Switch Transformer	1100M	11.7T	3.18	2.135 ± 0.007	1.758	75.38	18.02	26.10	26.81
Funnel Transformer	223M	1.9T	4.30	2.288 ± 0.008	1.918	67.34	16.26	22.75	23.20
Weighted Transformer	280M	71.0T	0.59	2.378 ± 0.021	1.989	69.04	16.98	23.02	26.30
Product key memory	421M	386.6T	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†

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

Lukasz Kaiser*
Google Brain
lukaszkaiser@google.com

Illia Polosukhin*[‡]
illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

*Equal contribution. Listing order is random. Jakob proposed replacing RNNs with self-attention and started the effort to evaluate this idea. Ashish, with Illia, designed and implemented the first Transformer models and has been crucially involved in every aspect of this work. Noam proposed scaled dot-product attention, multi-head attention and the parameter-free position representation and became the other person involved in nearly every detail. Niki designed, implemented, tuned and evaluated countless model variants in our original codebase and

The annotated Transformer (Code)

- [The Annotated Transformer \(harvard.edu\)](#)

Summary

- Yay, you now understand Transformers!
- Good luck on Class activity on Transformers!



Acknowledgments

- These slides were adapted from the book SPEECH and LANGUAGE PROCESSING: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition
- Some modifications from CS224N presentations and resources found in the WEB by several scholars.

Reference materials

- <https://vlanc-lab.github.io/mu-nlp-course/teachings/fall2024-AI-schedule.html>
- Lecture notes
- (A) Speech and Language Processing by Daniel Jurafsky and James H. Martin
- (B) Natural Language Processing with Python. (updated edition based on Python 3 and NLTK 3) Steven Bird et al. O'Reilly Media

