Natural Language Processing

CS 3216/UG, AI 5203/PG

Week-9

Multi-head attention, Transformers

Mahindra™ University

Global Thinkers. Engaged Leaders.

Recap

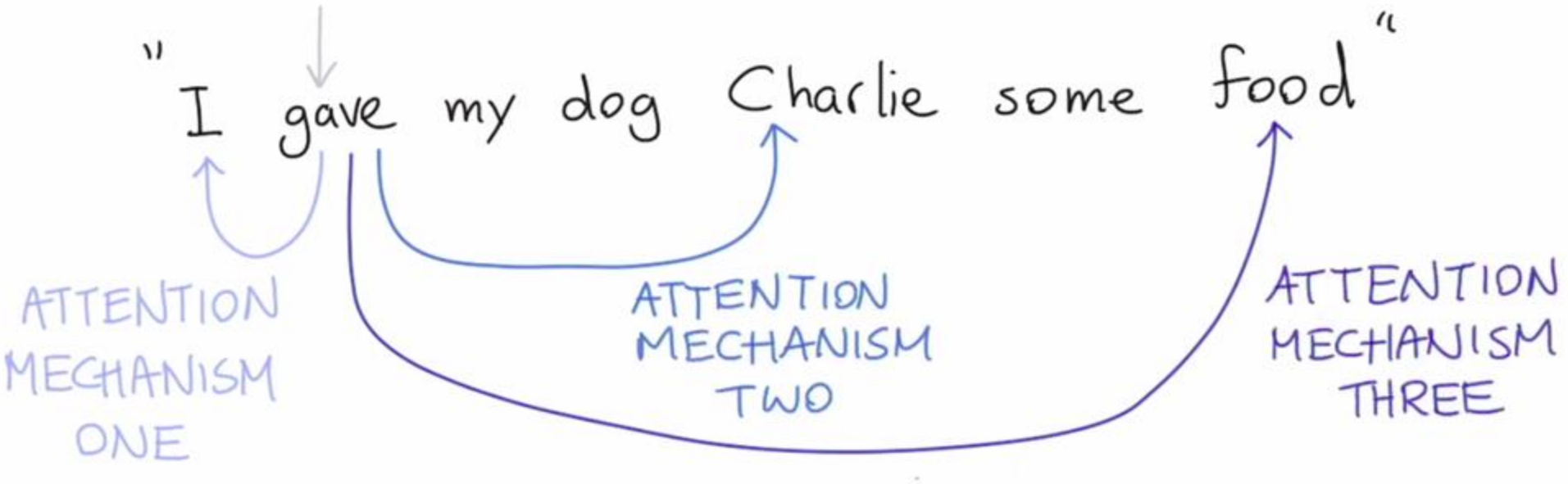
- Language modeling
- Recurrent Neural Network and Implementation
- Applications of Recurrent Neural Network
- Language modeling using Long Short-term Memory
- Sequence to Sequence learning
- Attention

So Far covered.....Intuition behind Attention

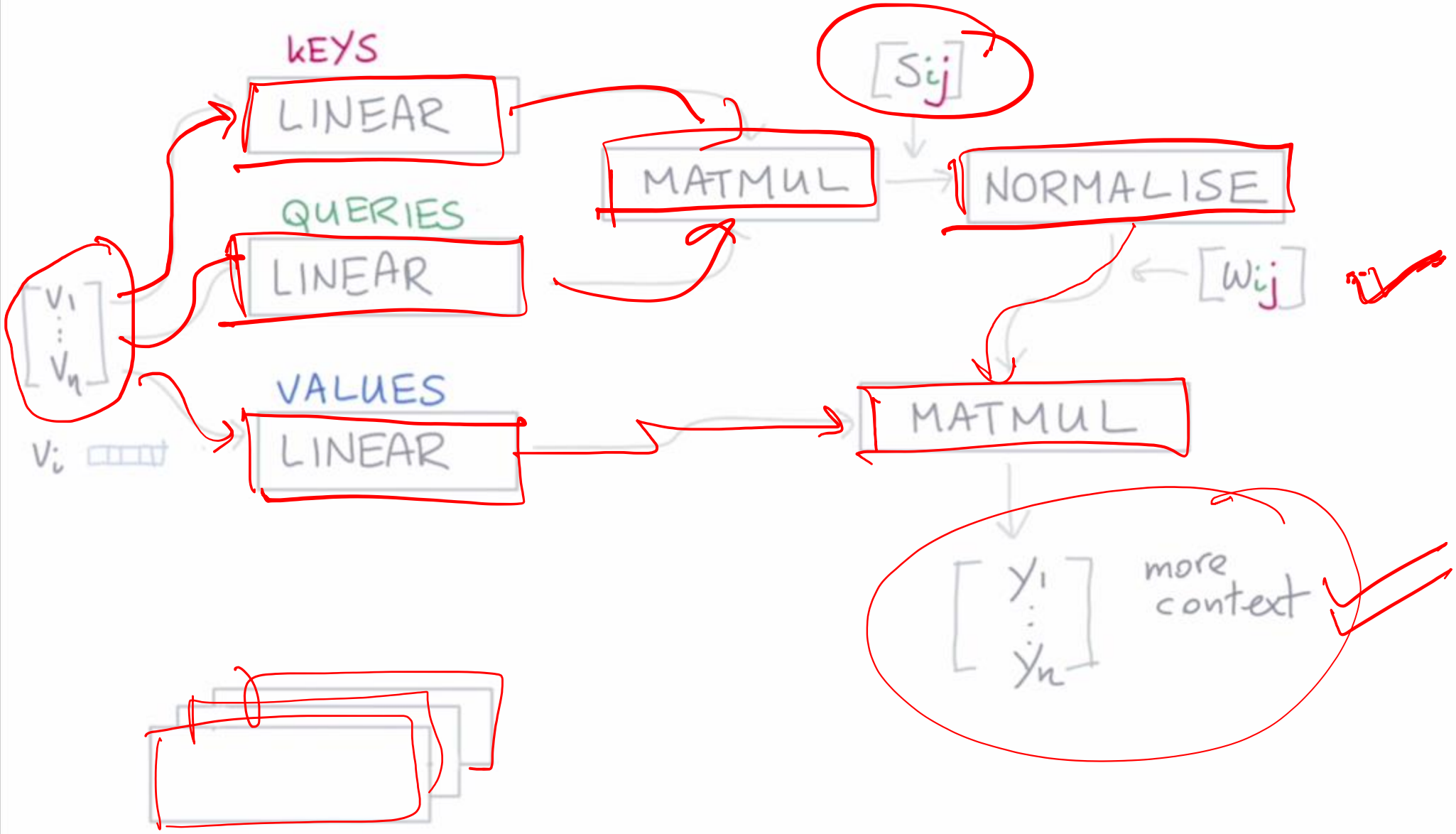
- The weighted sum is a selective summary of the information contained in the values, where the query determines which values to focus on.
- Attention is a way to obtain a fixed-size representation of an arbitrary set of representations (the values), dependent on some other representation (the query)

https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1244/slides/cs224n-2024-lecture08-transformers.pdf

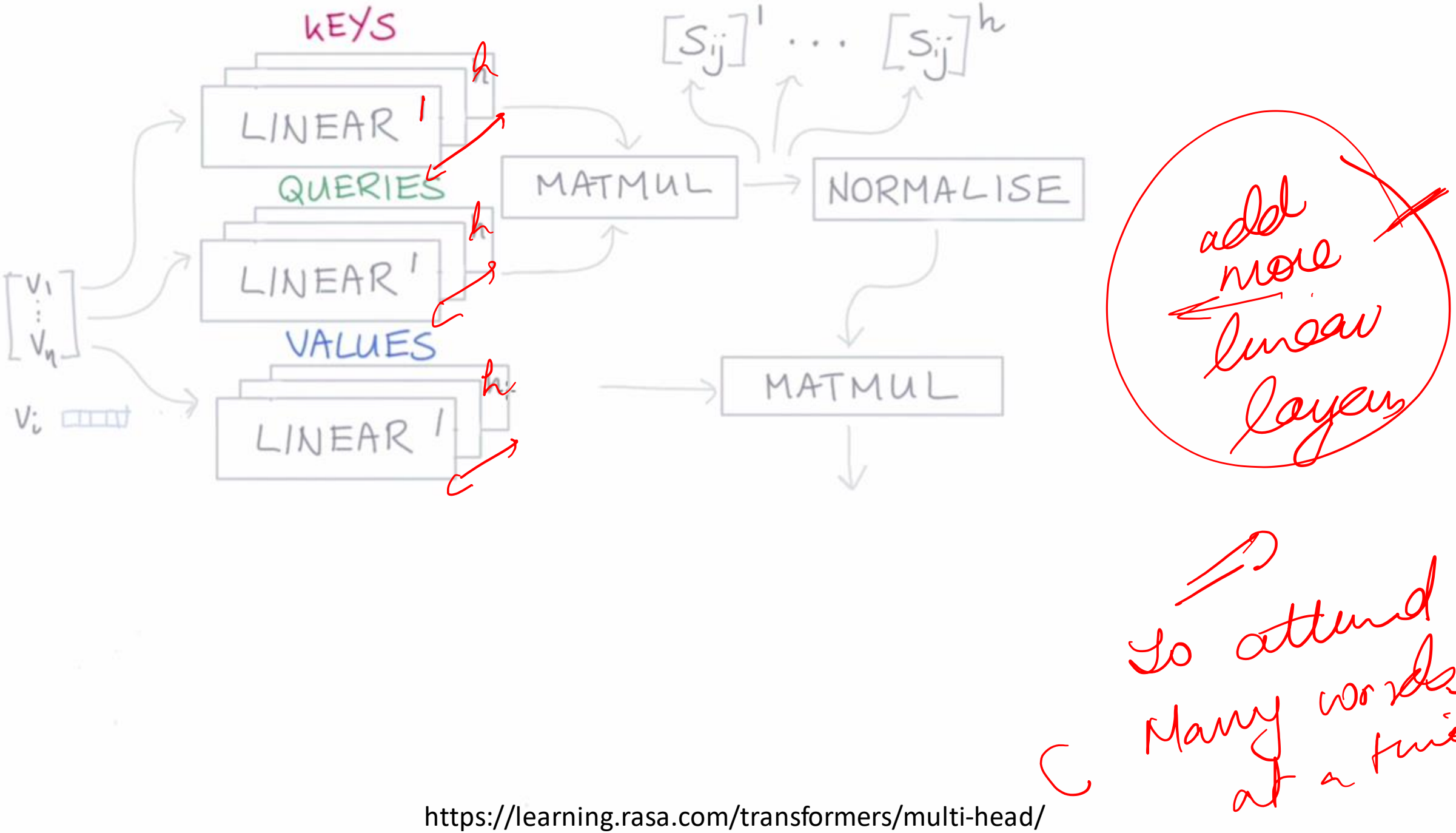
Do we have enough attention?



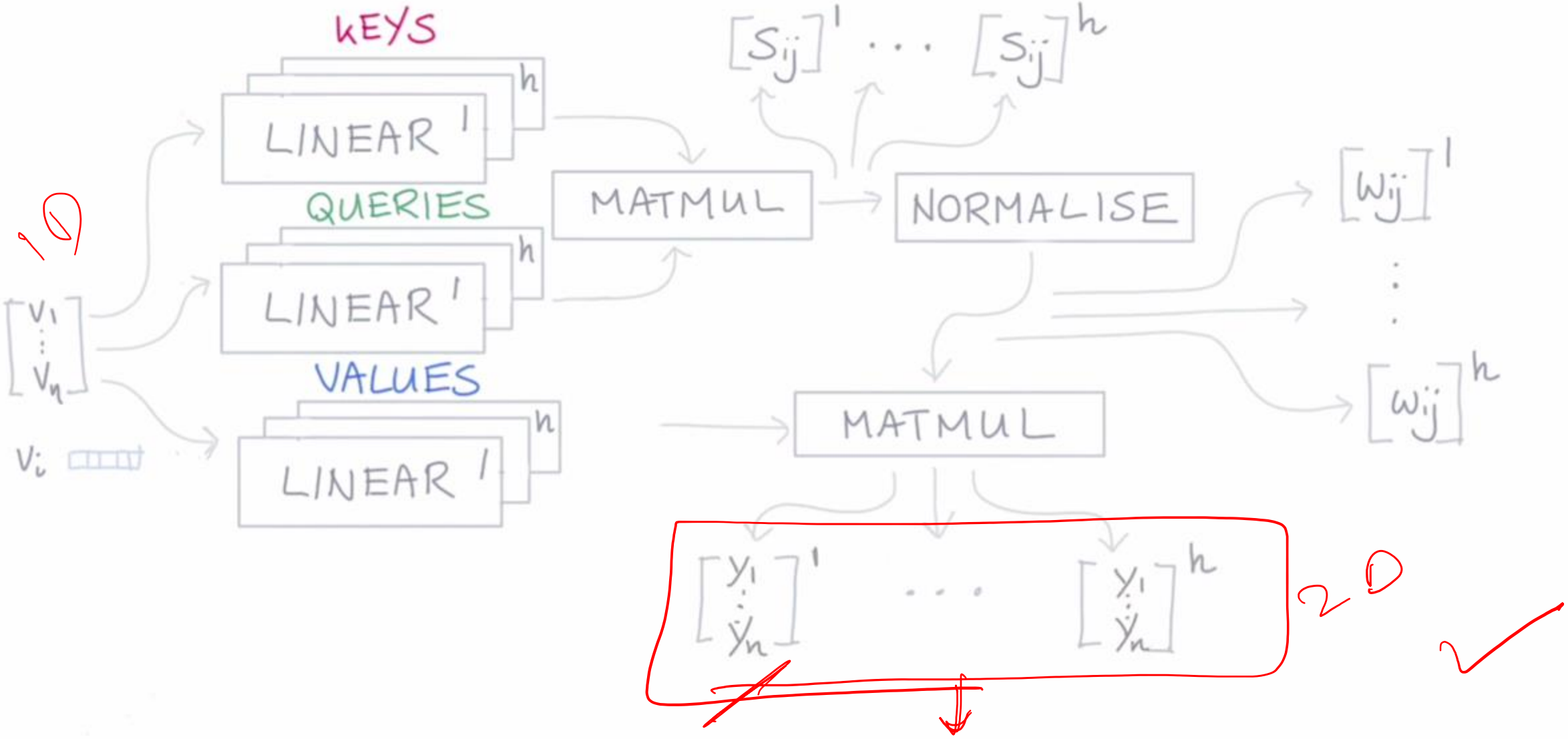
https://learning.rasa.com/transformers/multi-head/



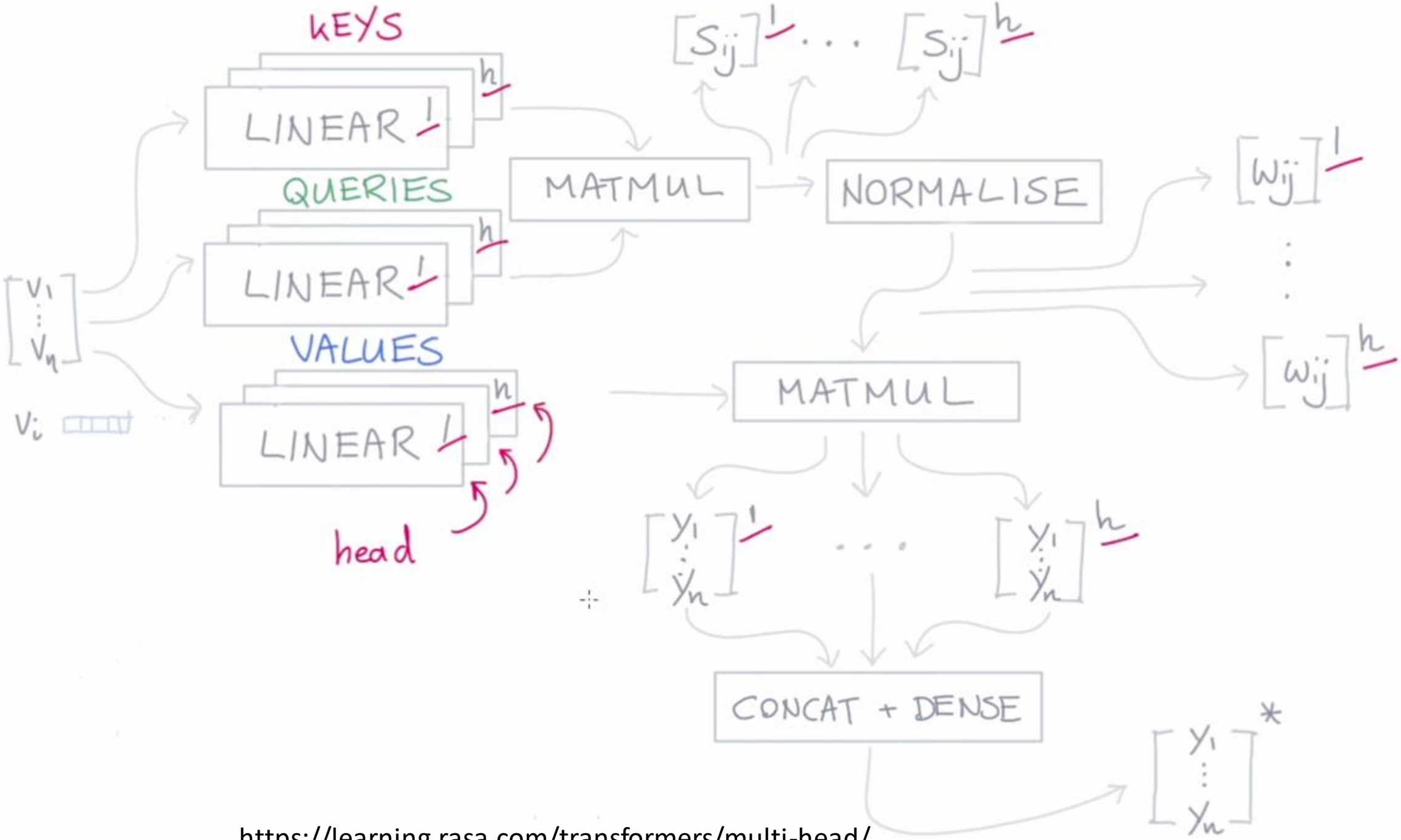
https://learning.rasa.com/transformers/multi-head/



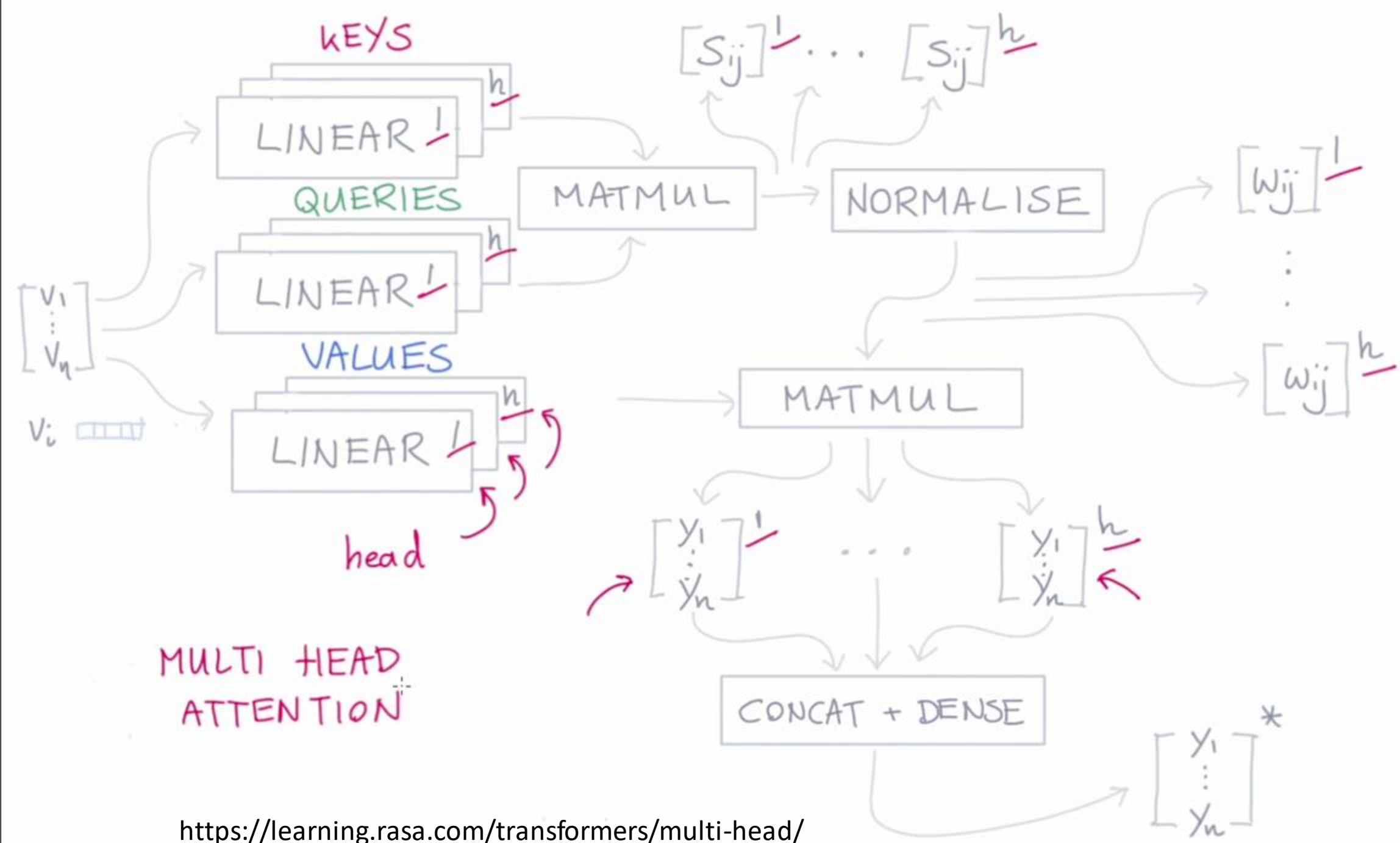
https://learning.rasa.com/transformers/multi-head/



https://learning.rasa.com/transformers/multi-head/



https://learning.rasa.com/transformers/multi-head/



https://learning.rasa.com/transformers/multi-head/

Multi-head attention

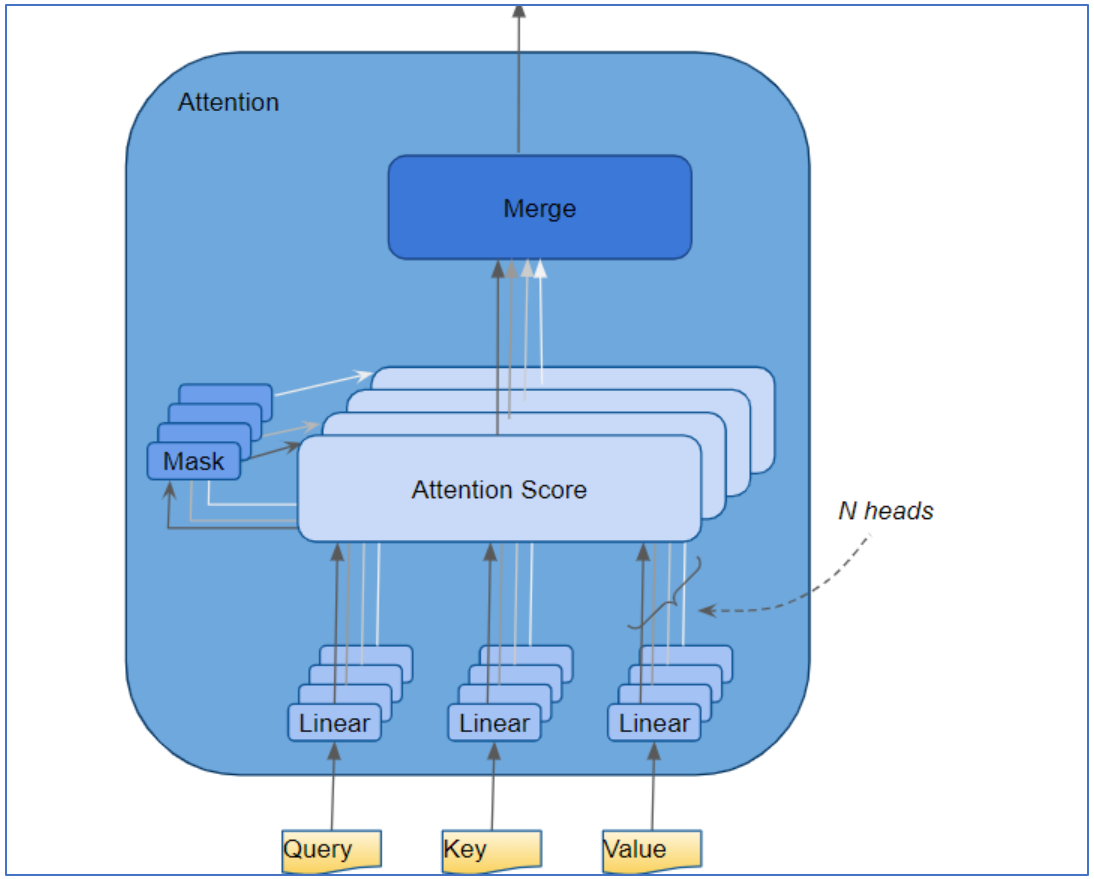
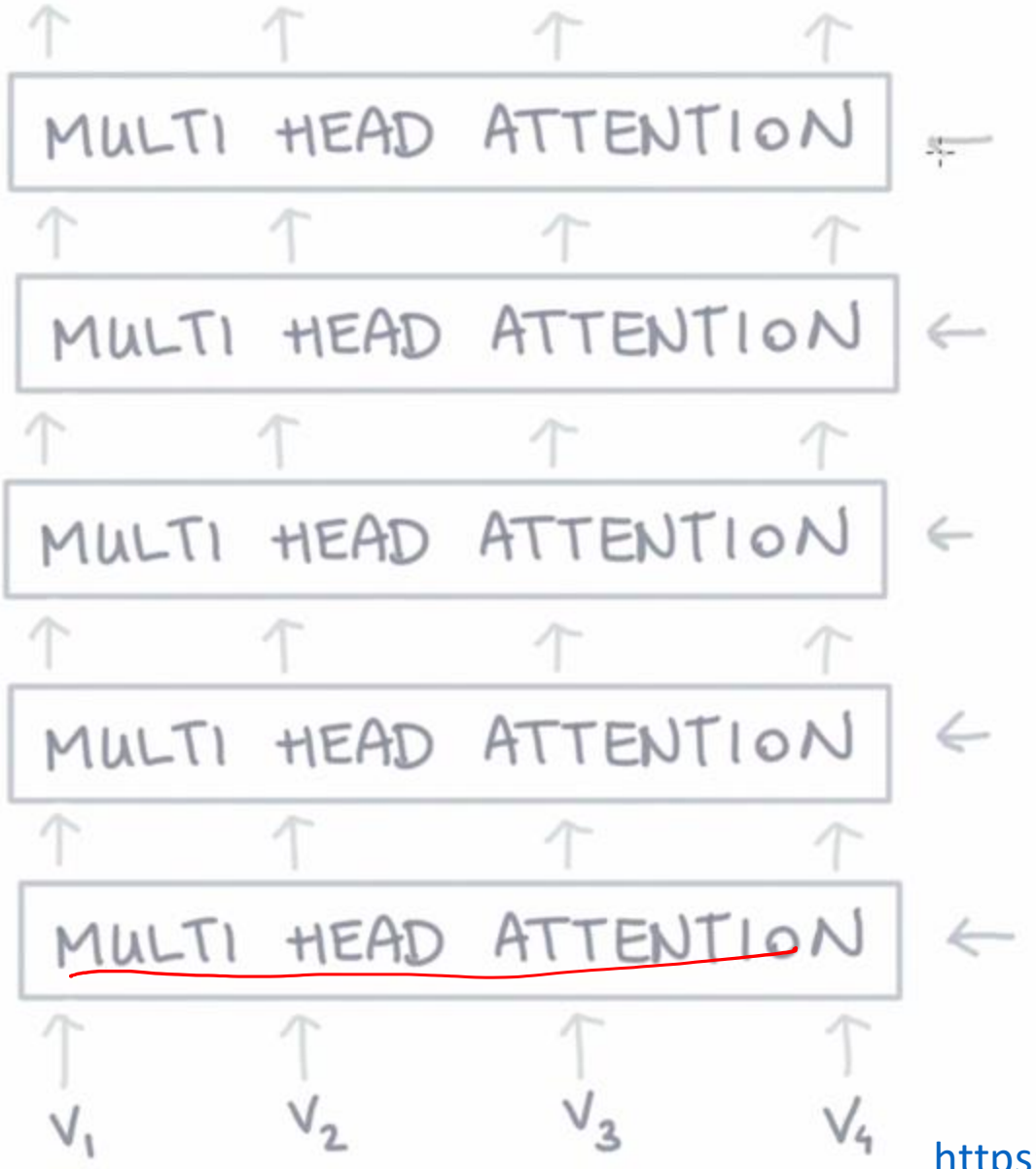
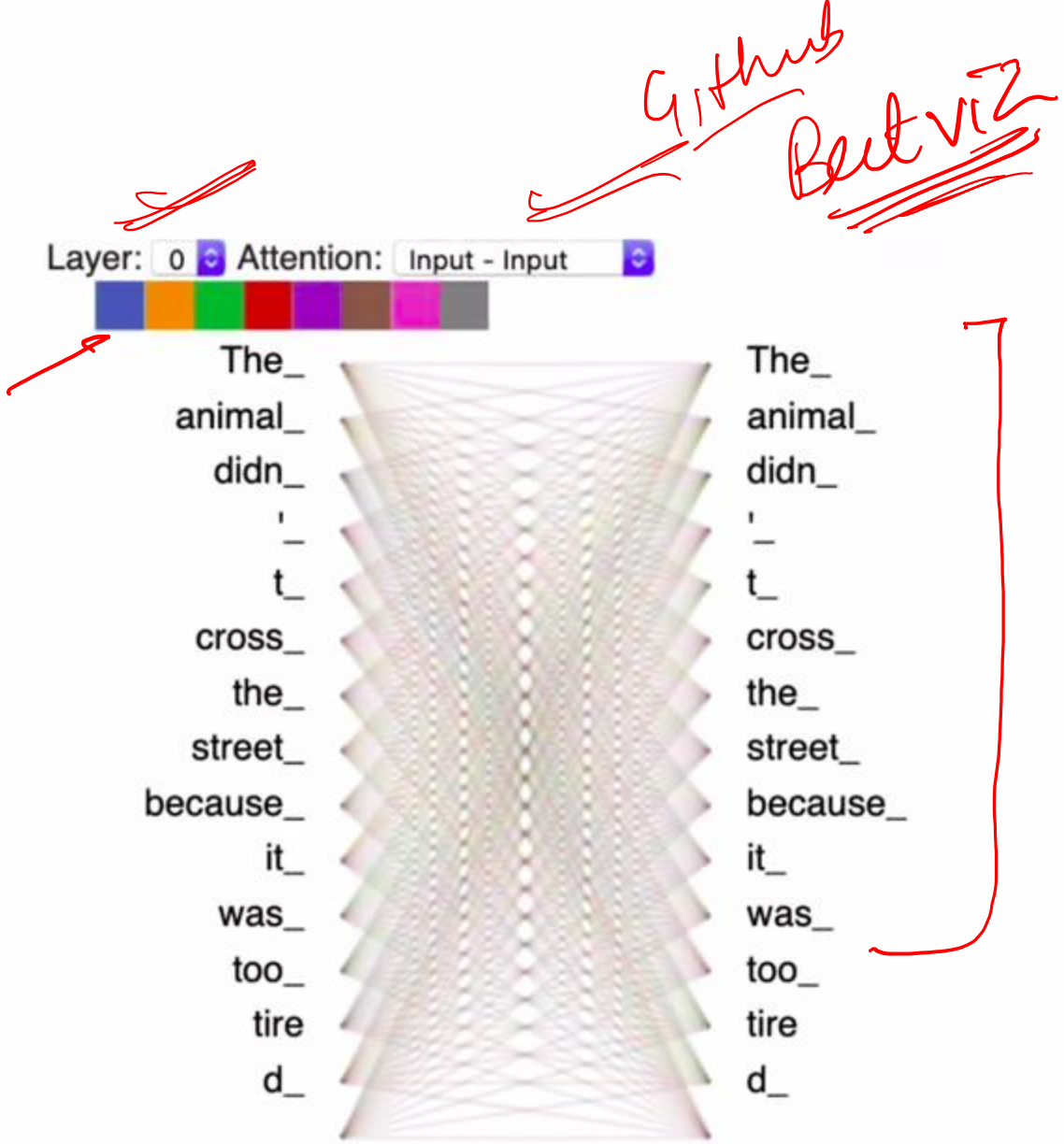
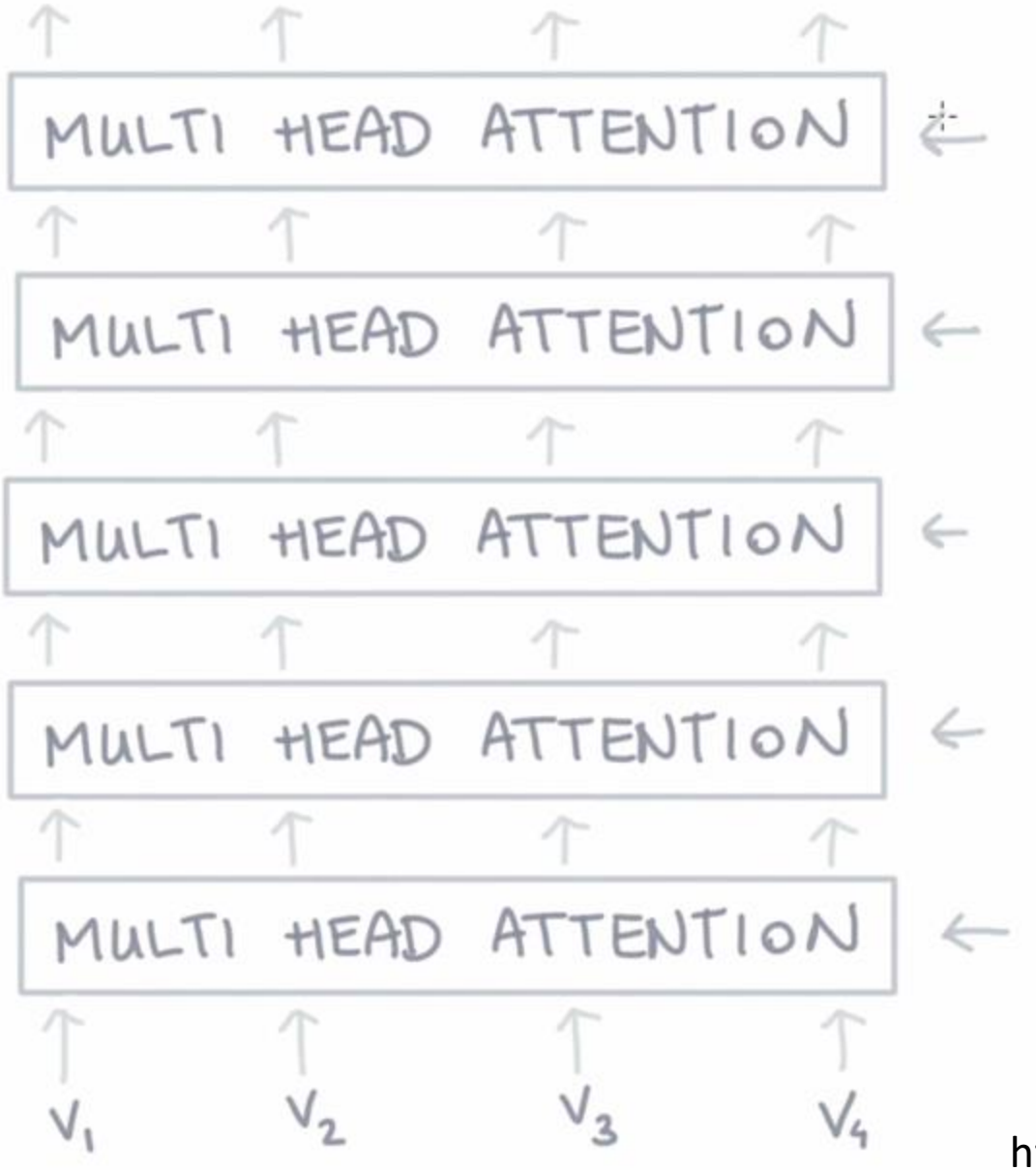


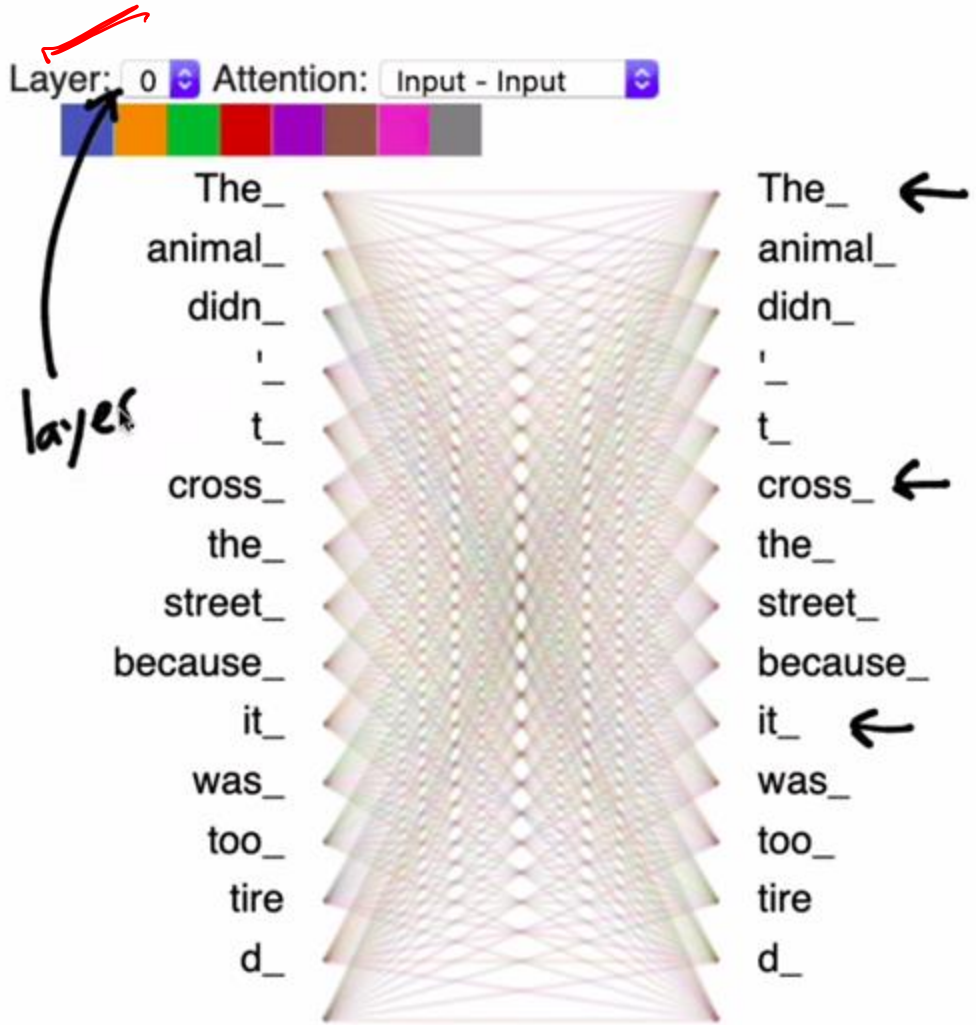
Image source:
https://towardsdatascience.com/transformers-explained-visually-part-3-multi-head-attention-deep-dive-1c1ff1024853



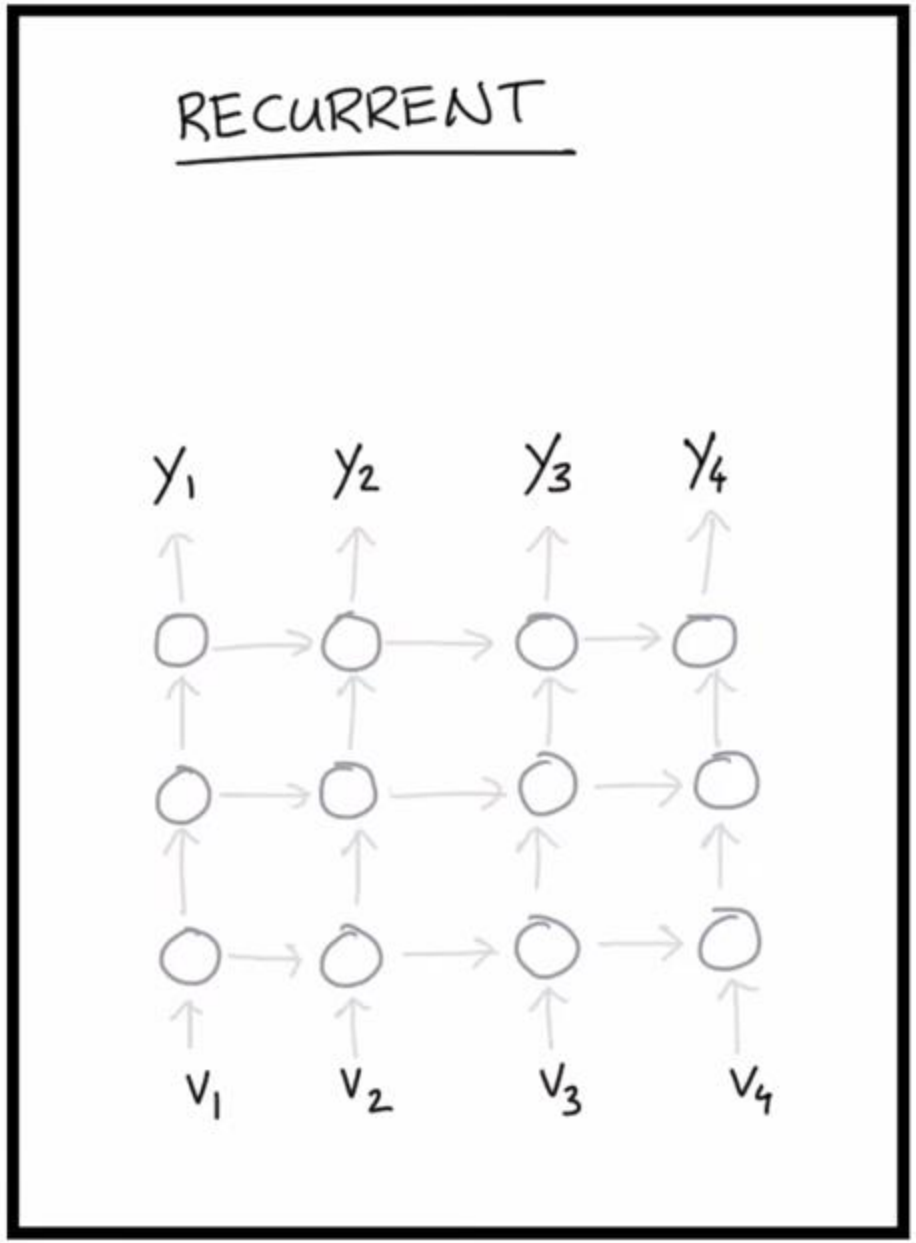


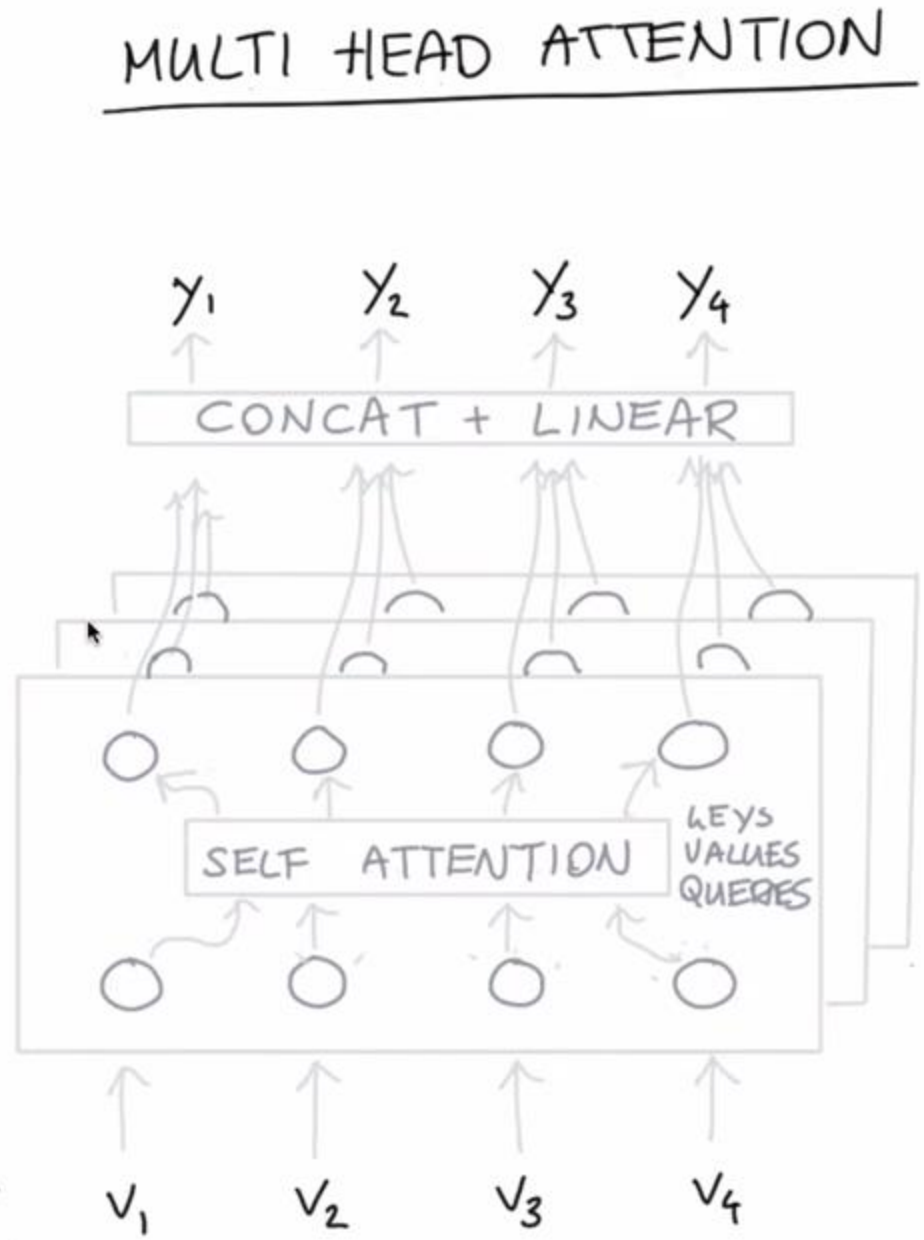
https://learning.rasa.com/transformers/multi-head/





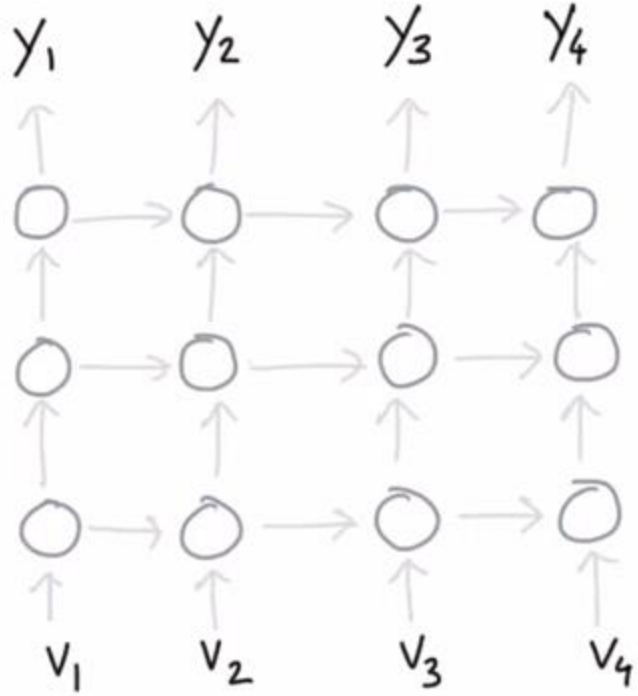
https://learning.rasa.com/transformers/multi-head/



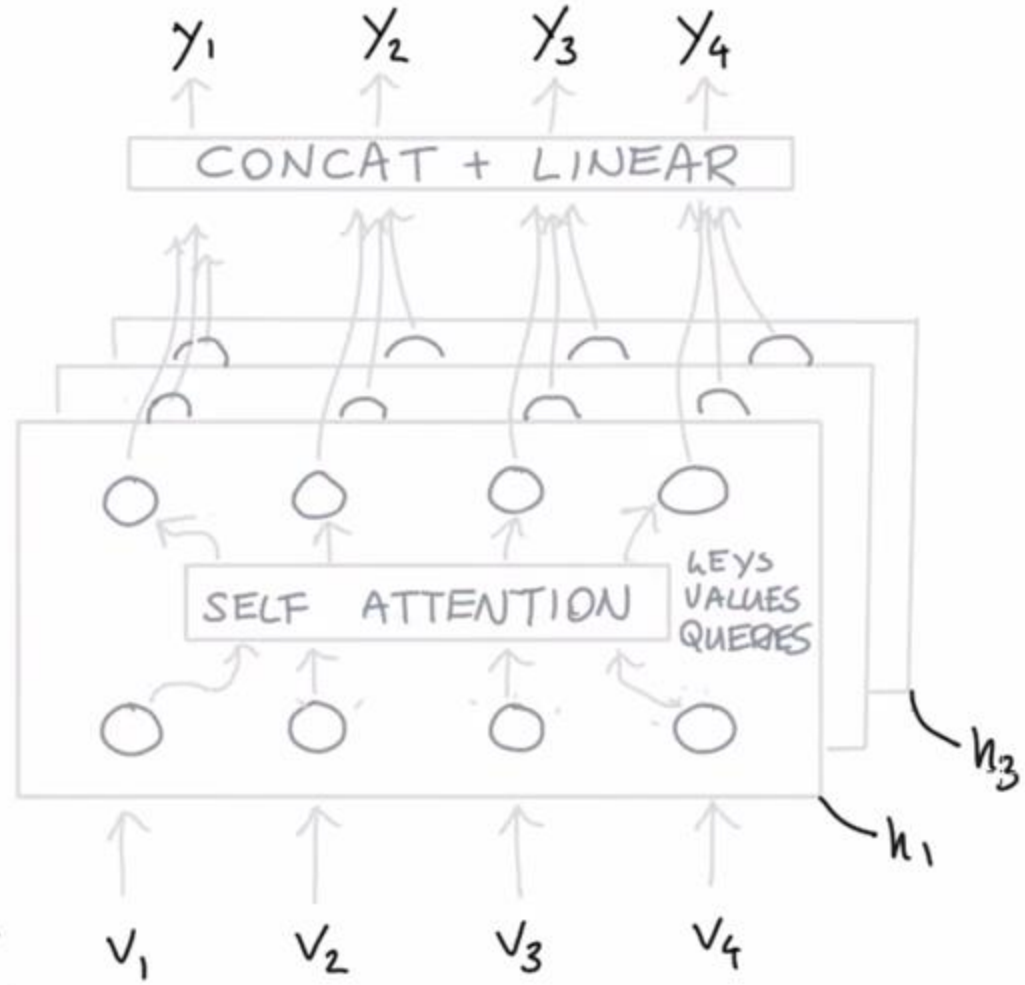


https://learning.rasa.com/transformers/multi-head/

RECURRENT

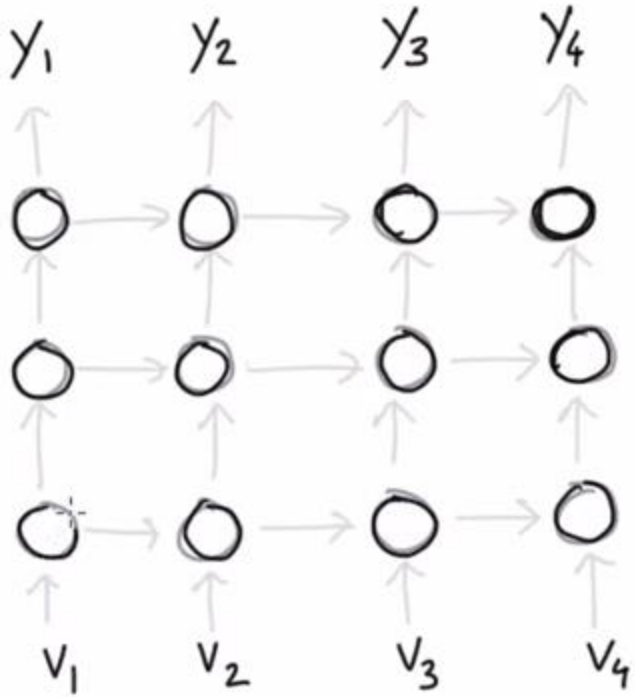


MULTI HEAD ATTENTION

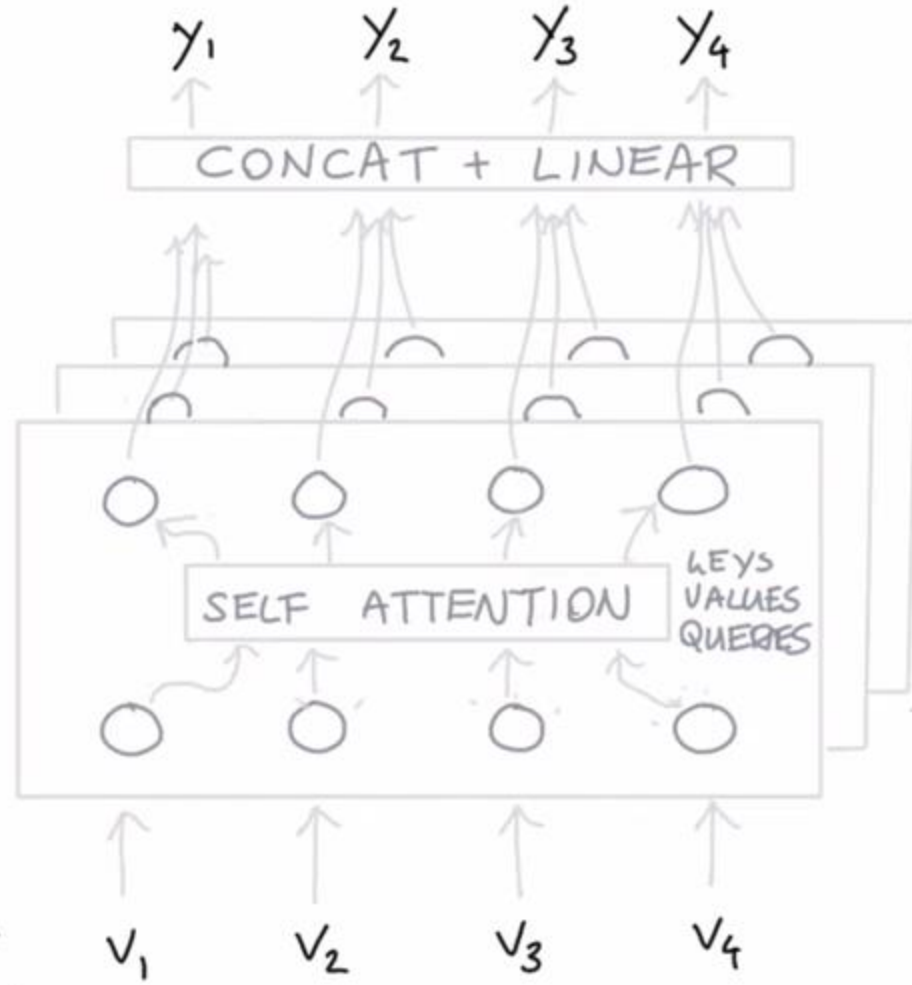


https://learning.rasa.com/transformers/multi-head/

RECURRENT



MULTI HEAD ATTENTION



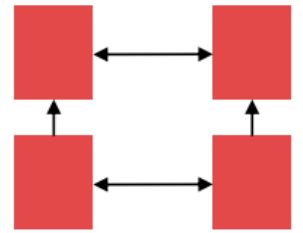
https://learning.rasa.com/transformers/multi-head/

Issues with Recurrent models: Linear interaction distance

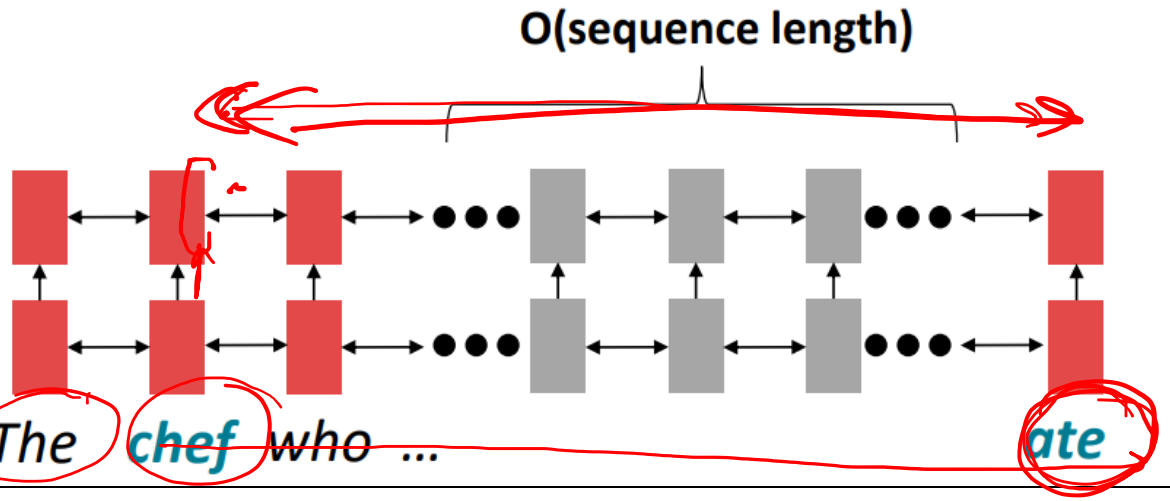
RNNs are unrolled "left-to-right".

It encodes linear locality: a useful heuristic!

- Nearby words often affect each other's meanings



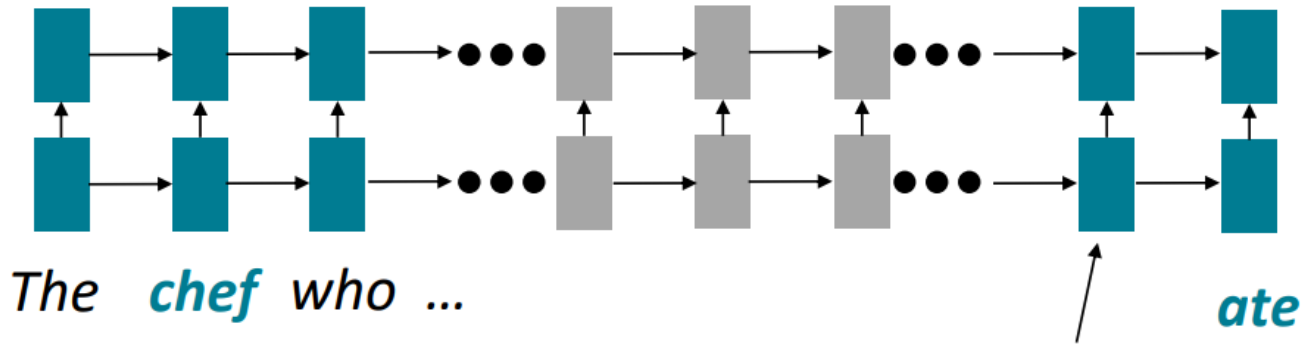
Problem: RNNs take **O(sequence length)** steps for distant word pairs to interact.



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...

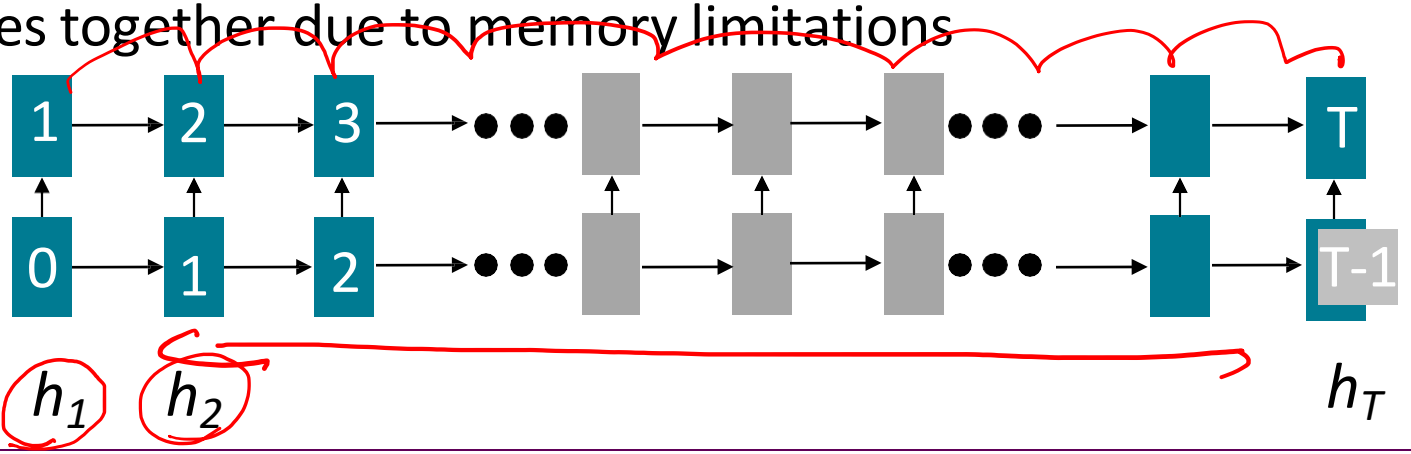


Info of ***chef*** has gone through O(sequence length) many layers!

Adapted from cs224n-2024-lecture08-transformers.pdf (stanford.edu)

Lack of Parallelizability in RNN

- 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

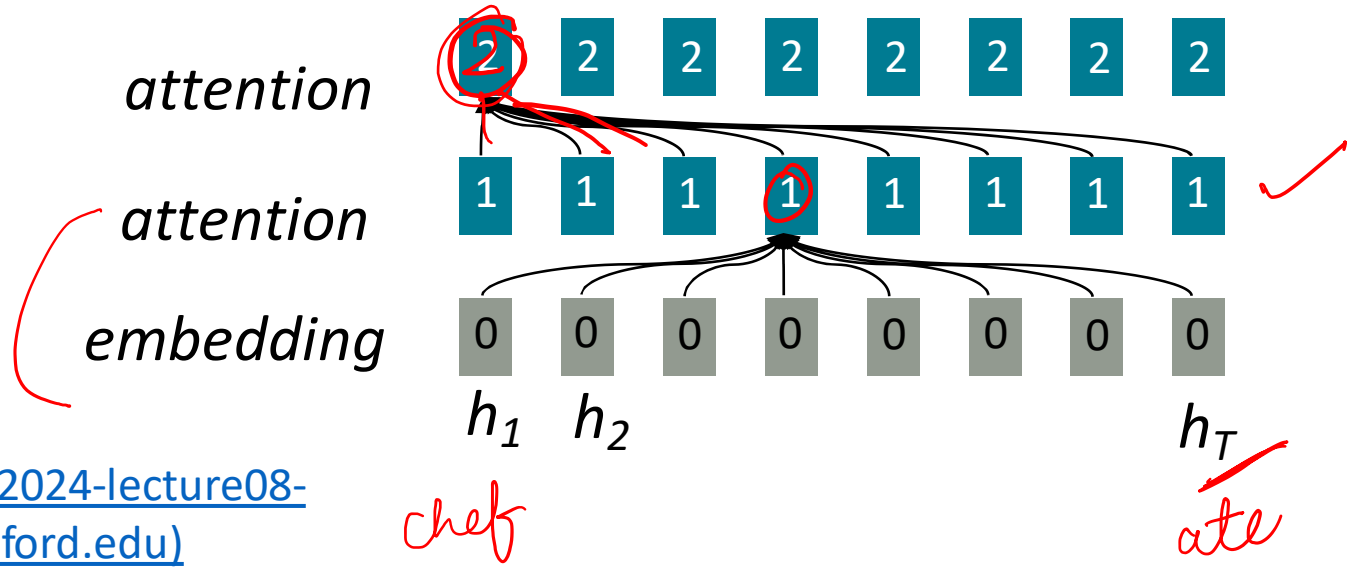
Adapted from cs224n-2024-lecture08-transformers.pdf (stanford.edu)

High-Level Architecture: Transformer is all about

(Self) Attention

To recap, **attention** treats each word's representation as a **query** to access and incorporate information from **a set of values.**

- Last lecture, we saw attention from the **decoder** to the **encoder** in a recurrent sequence-to-sequence model
- **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

[Adapted from cs224n-2024-lecture08-transformers.pdf (stanford.edu)]

Transformer architecture

The Transformers authors had 3 desirata 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.

Adapted from cs224n-2024-lecture08-transformers.pdf (stanford.edu)

Computational Complexity per layer

When sequence length (n) << 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.

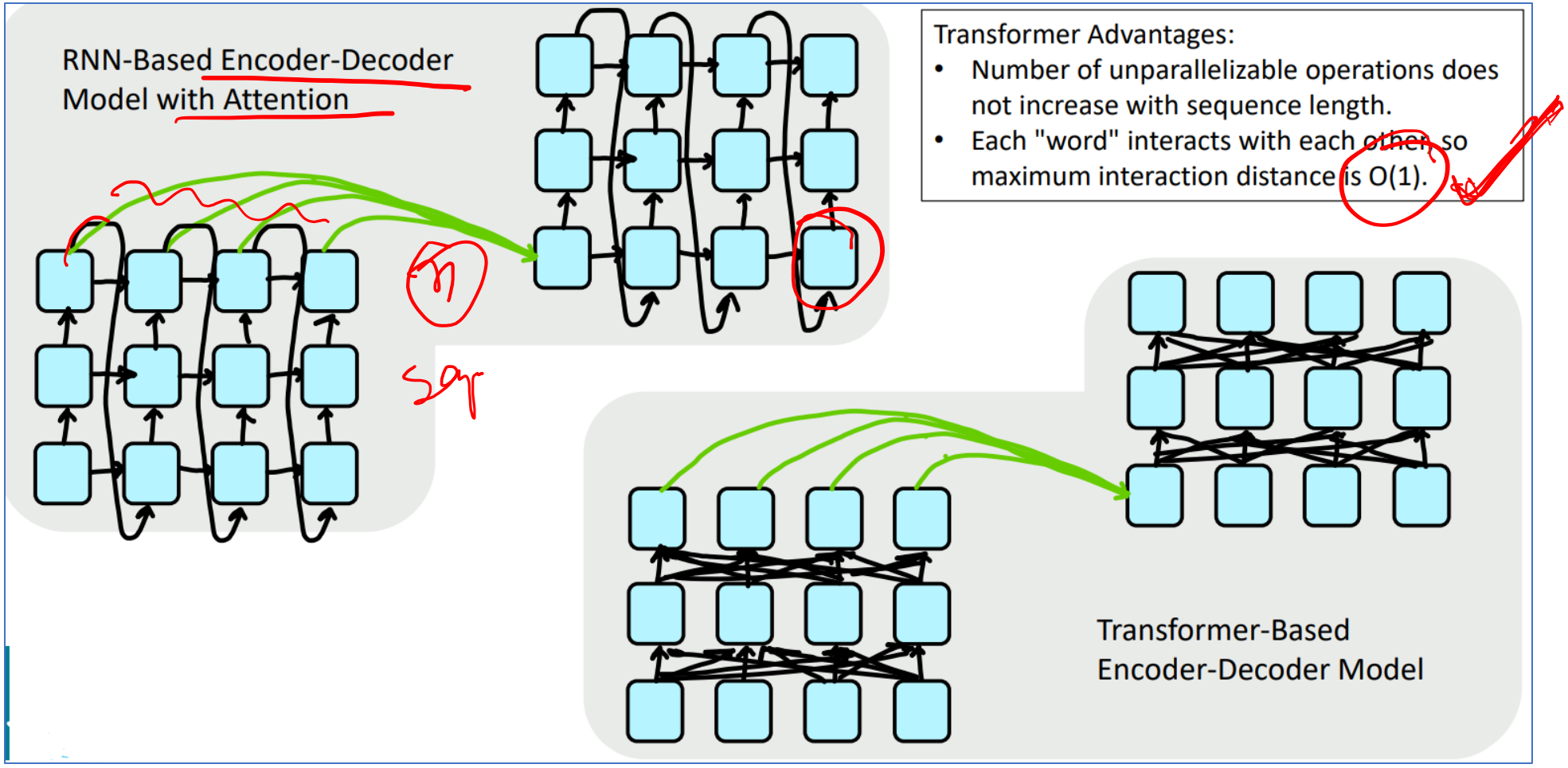
Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Table 1 of the Transformer paper.

d = vector representation dimension
n = sequence length

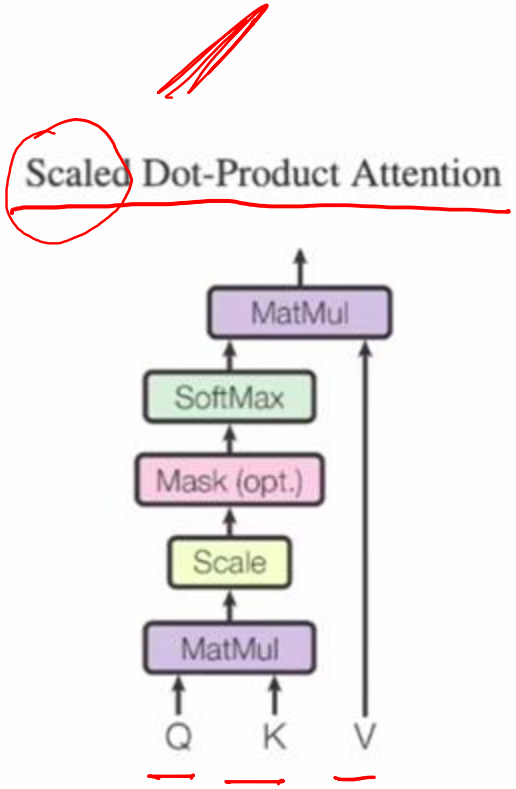
https://arxiv.org/abs/1706.03762

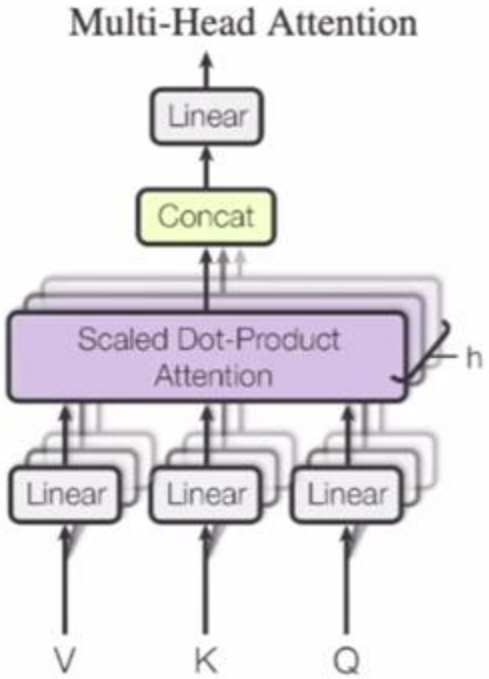
Path Length minimization and Parallelization



Adapted from cs224n-2024-lecture-slides

Transformer





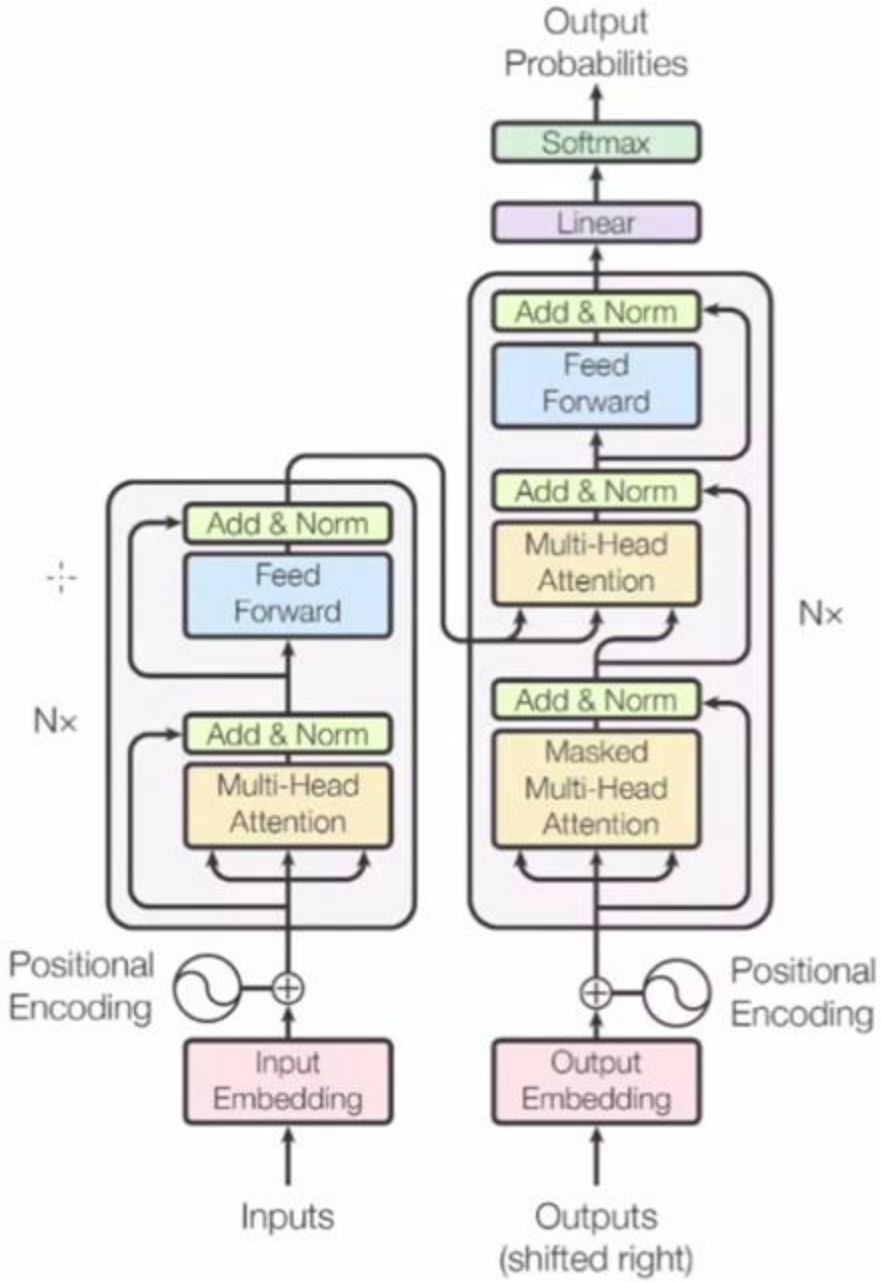
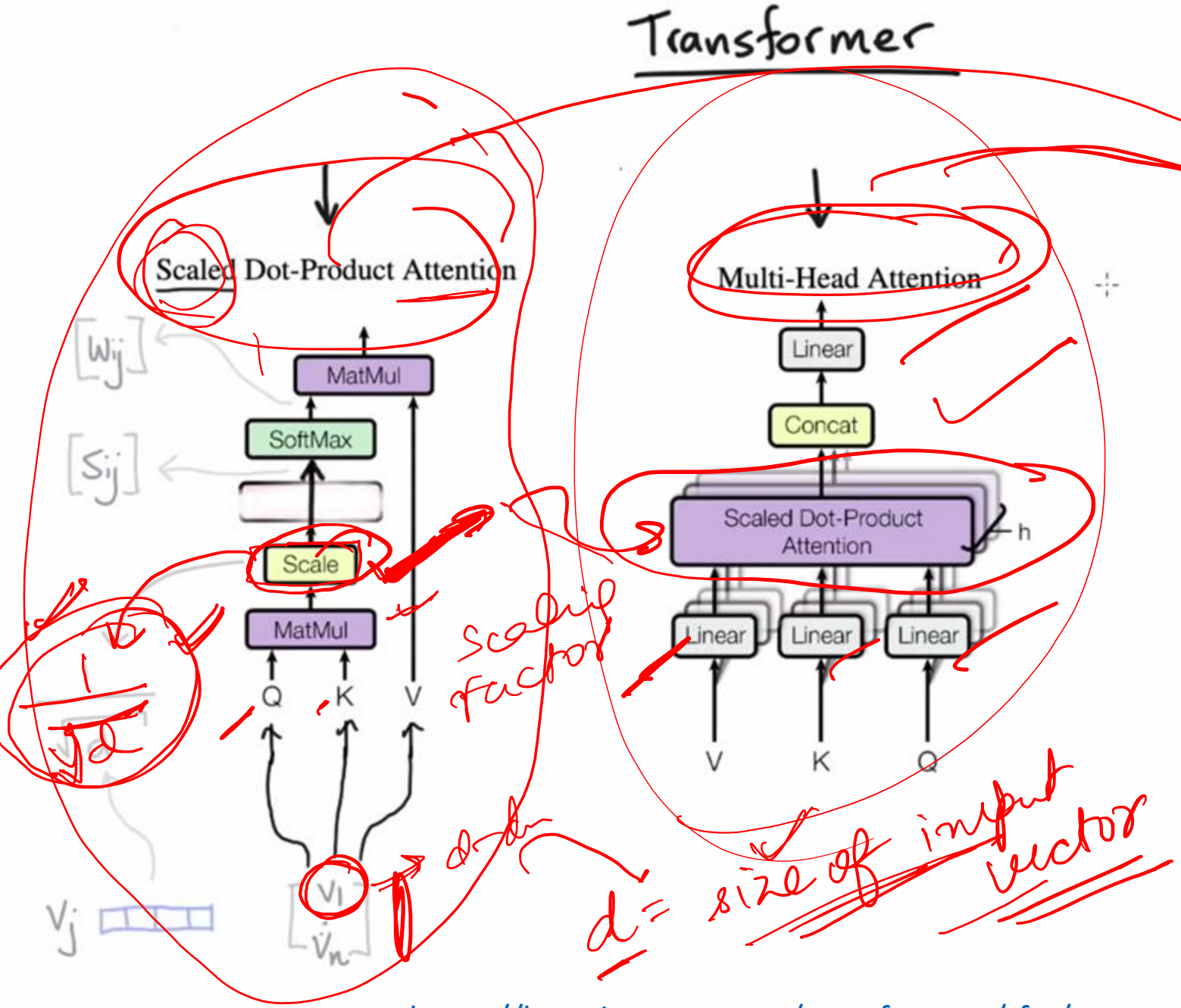


Figure 1: The Transformer - model architecture.

https://learning.rasa.com/transformers/tfm/

Transformer



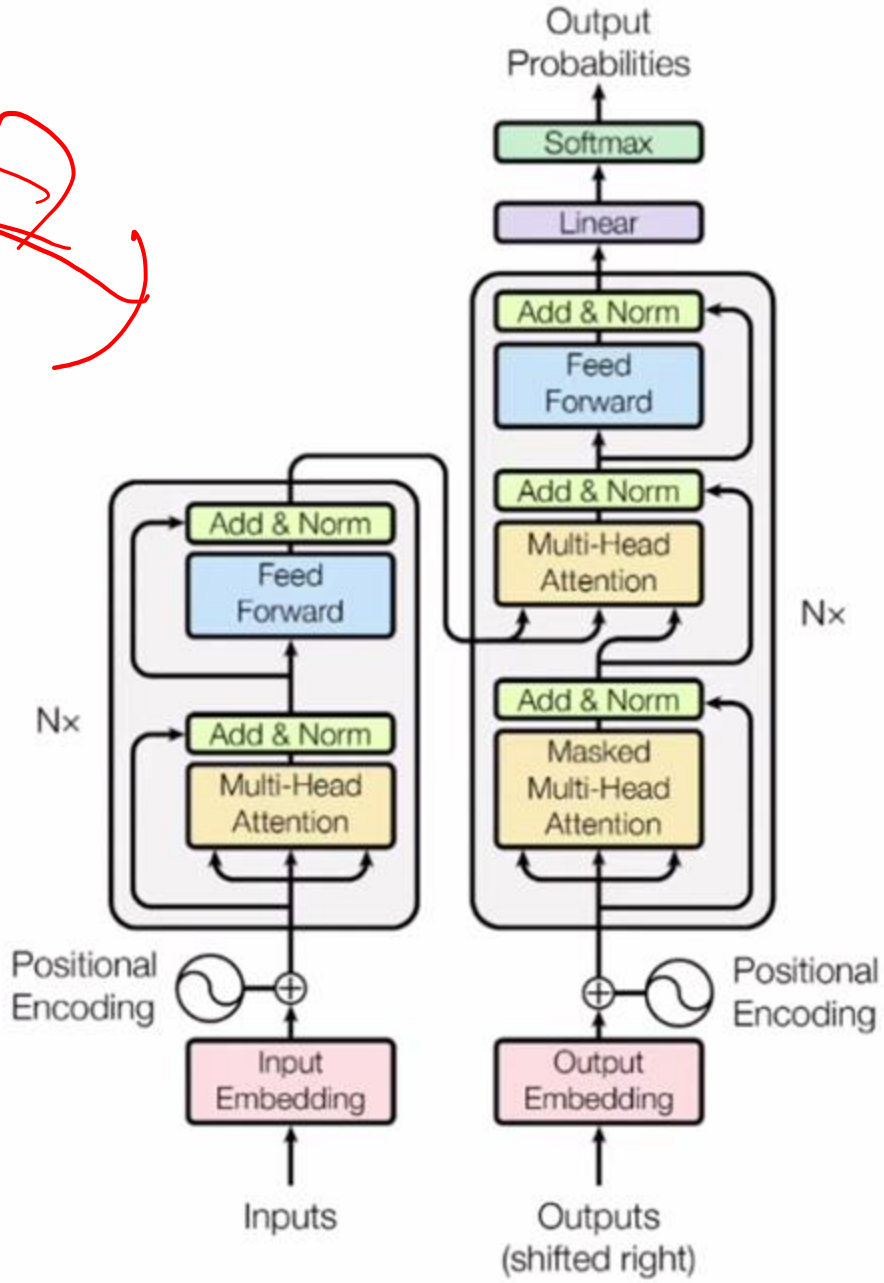
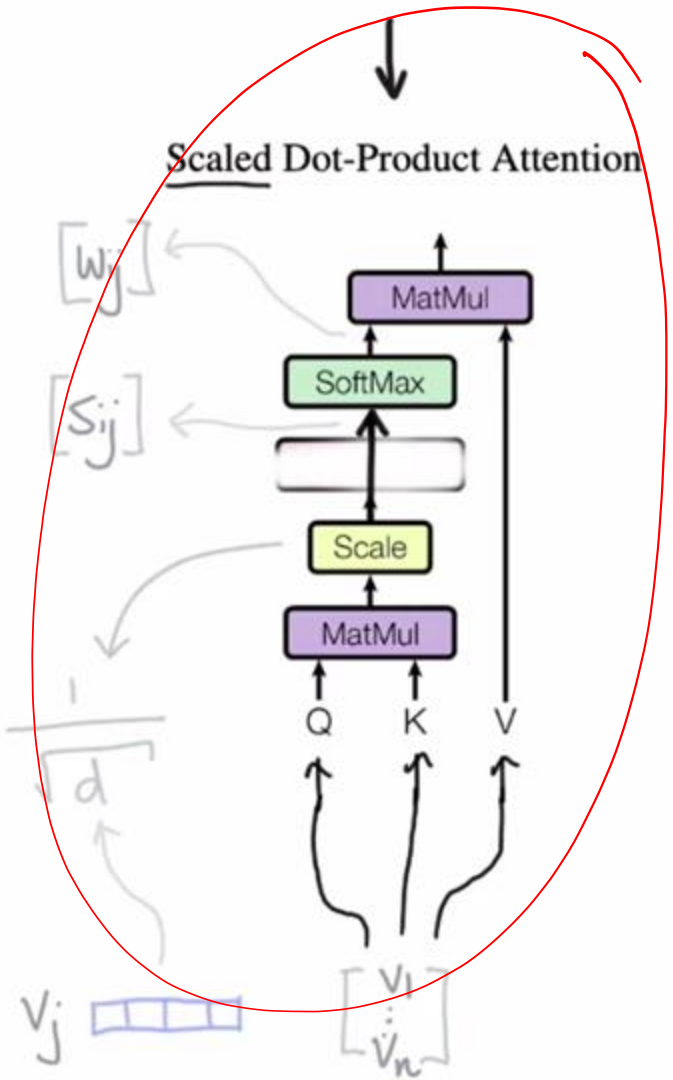
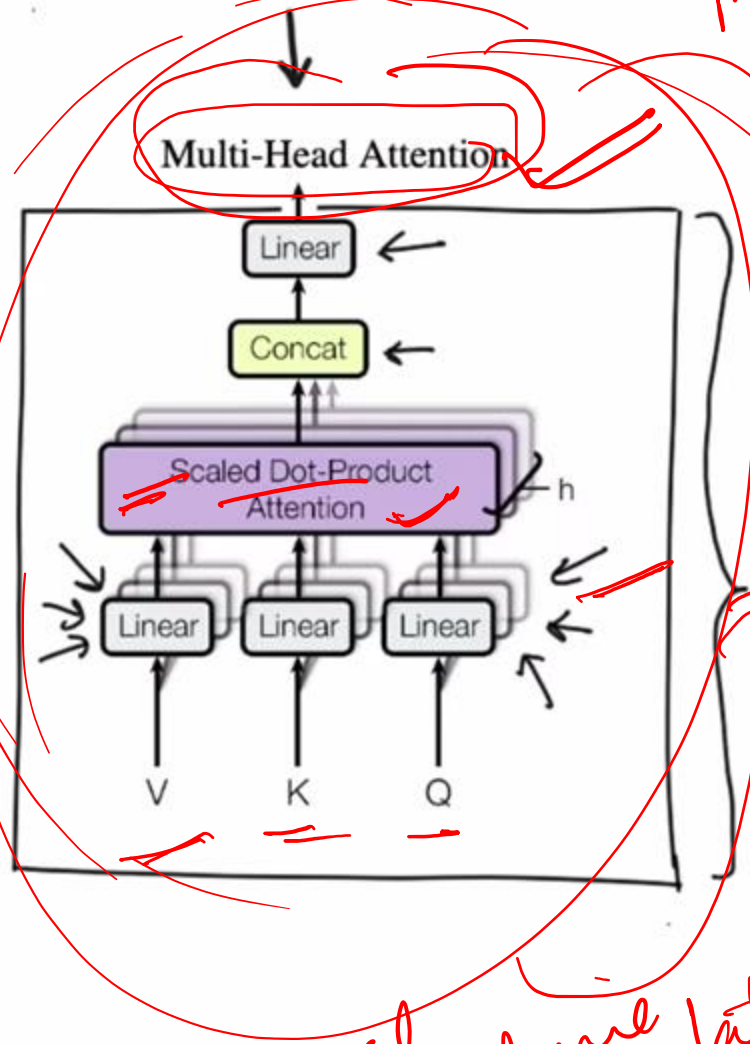


Figure 1: The Transformer - model architecture.

https://learning.rasa.com/transformers/tfm/

Transformer





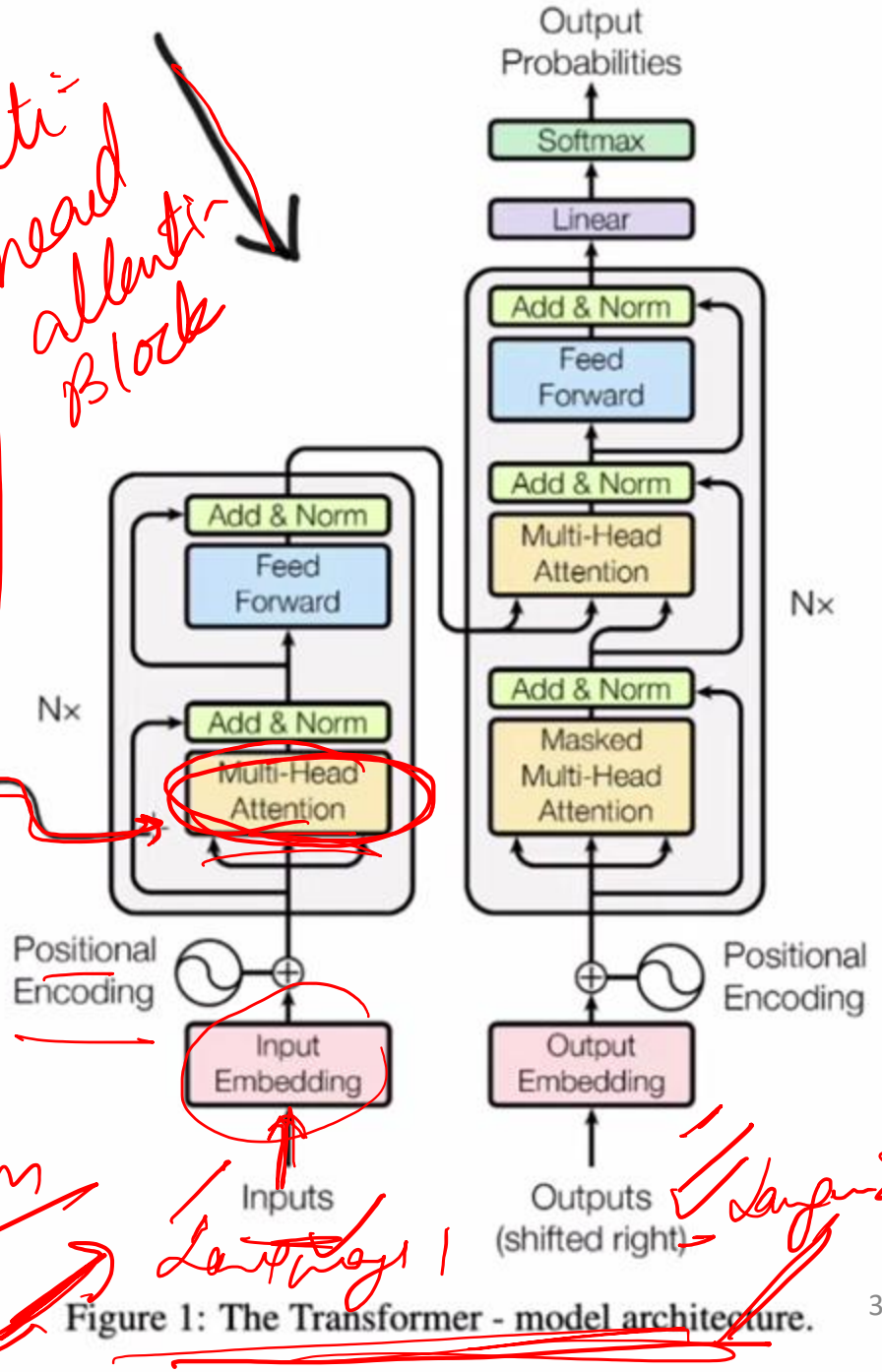


Figure 1: The Transformer - model architecture.

https://learning.rasa.com/transformers/tfm/

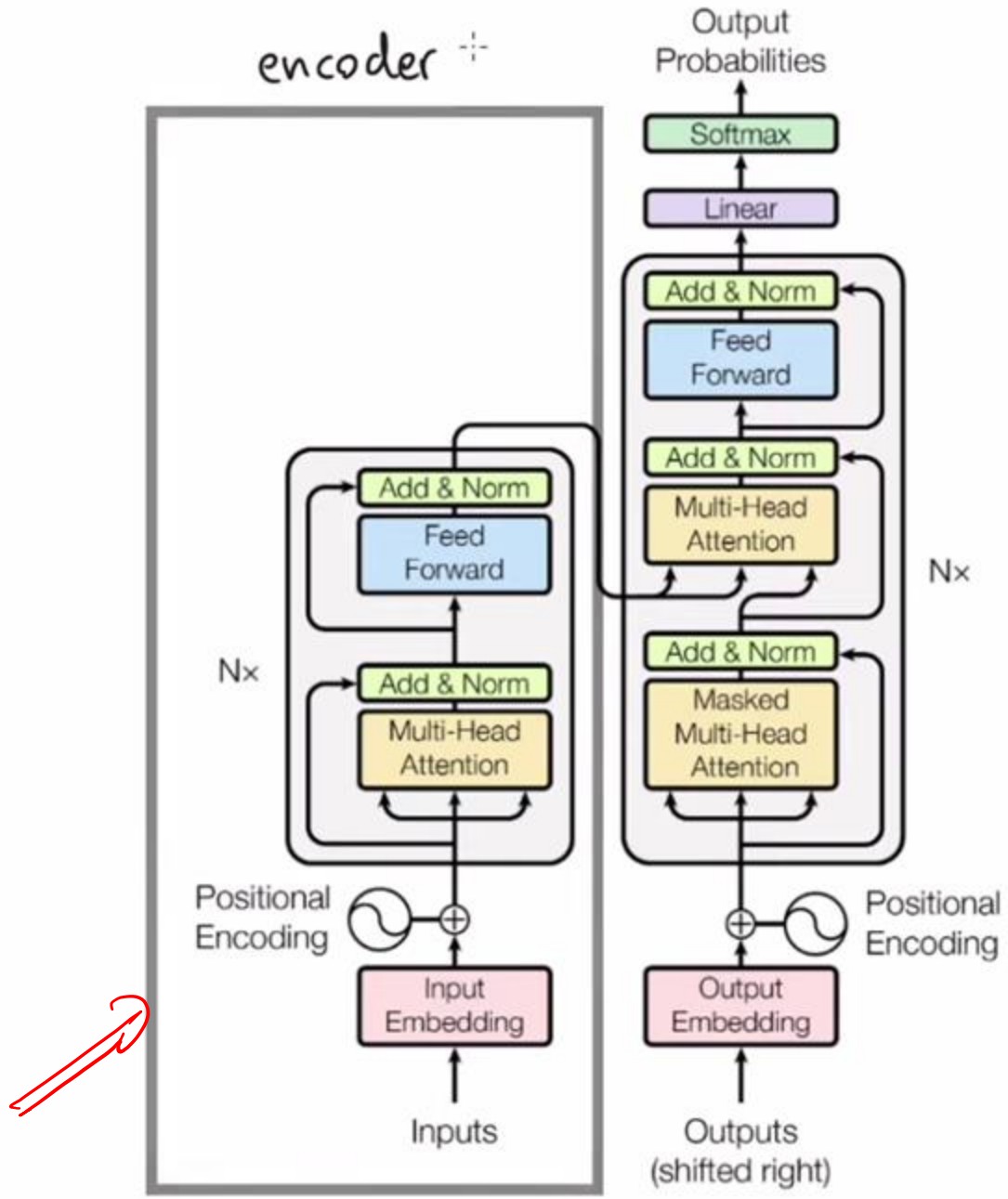
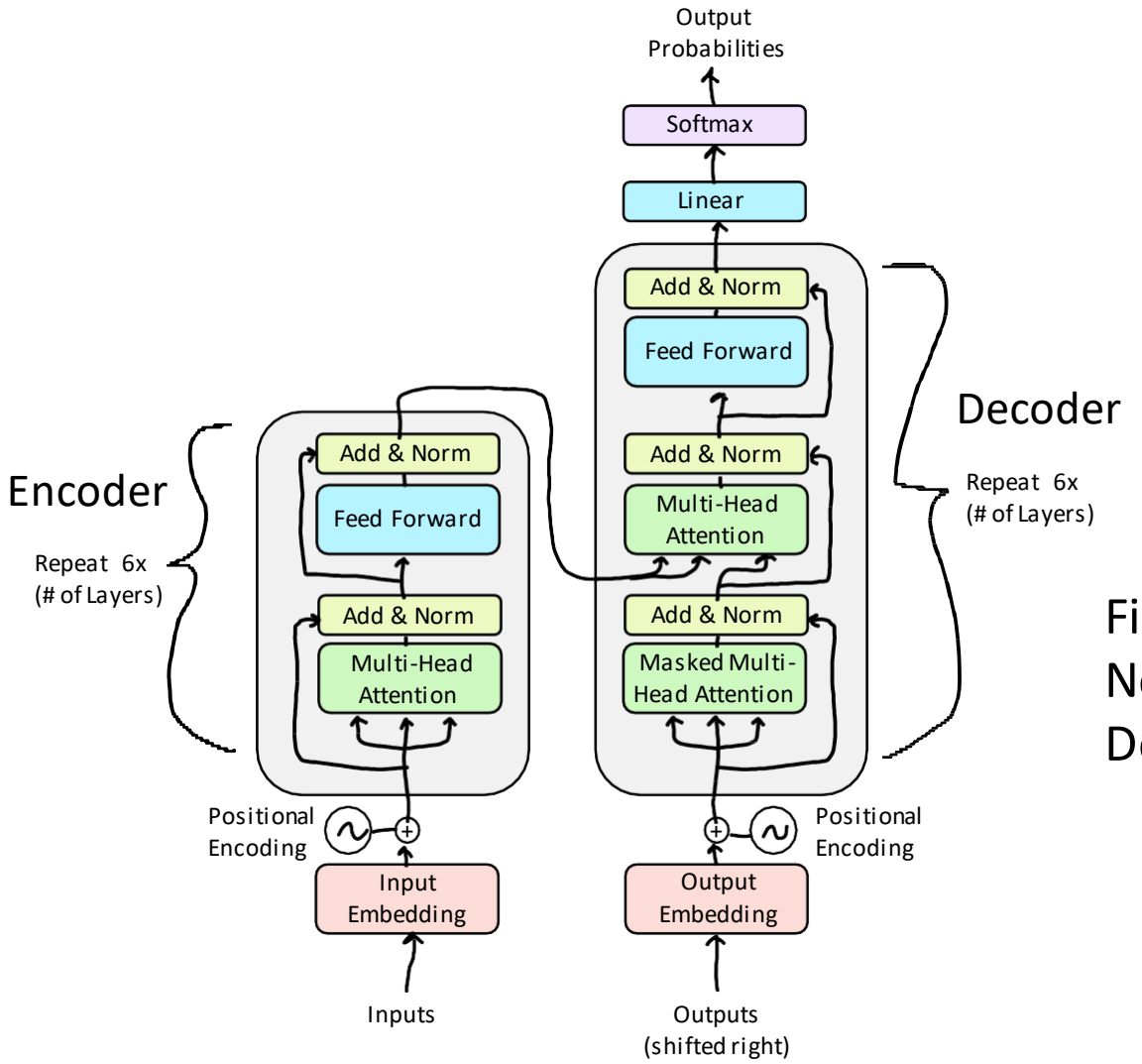


Figure 1: The Transformer - model architecture.

Translation

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

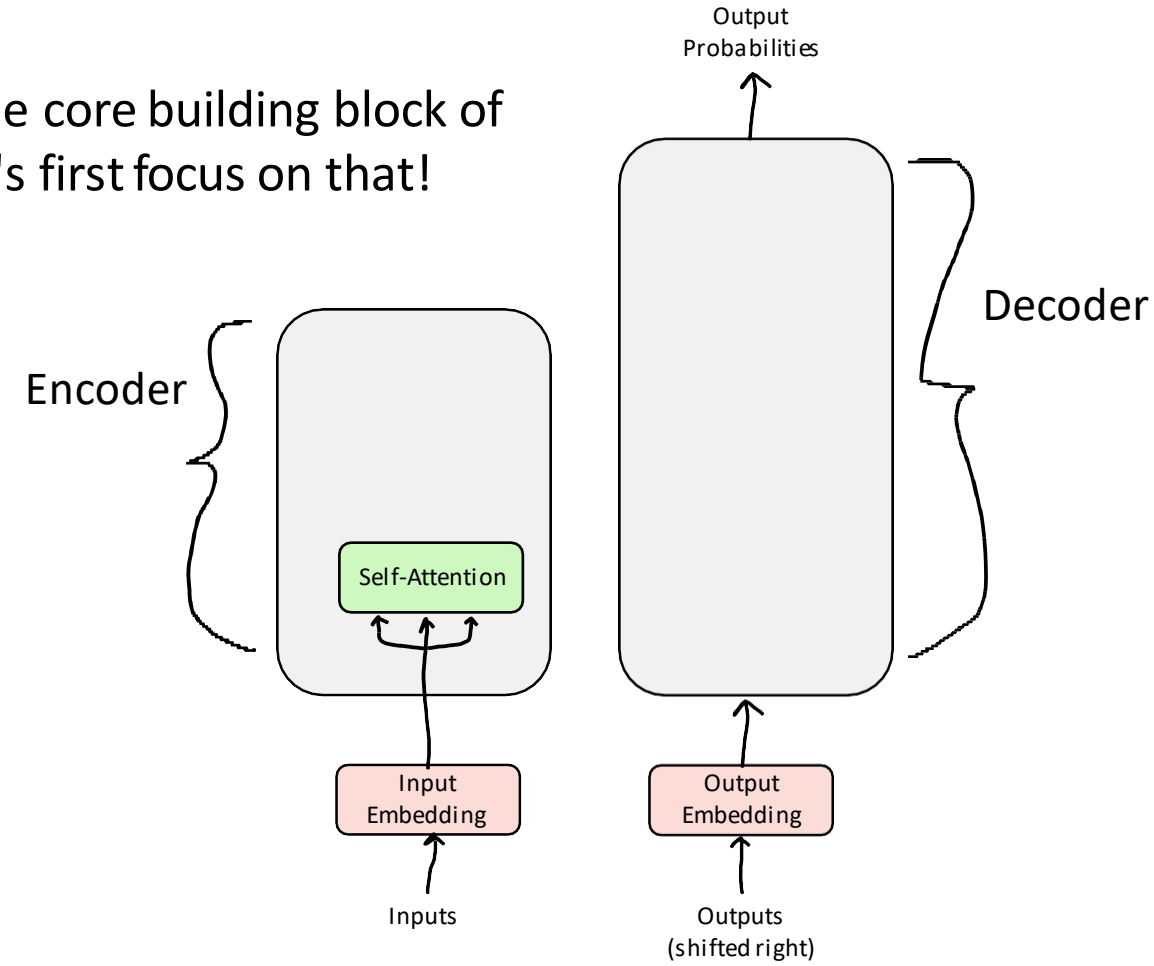


First, we will talk about the Encoder!
Next, we will go through the Decoder (which is quite similar)!

Adapted from cs224n-2024-lecture-slides

Encoder: Self-Attention

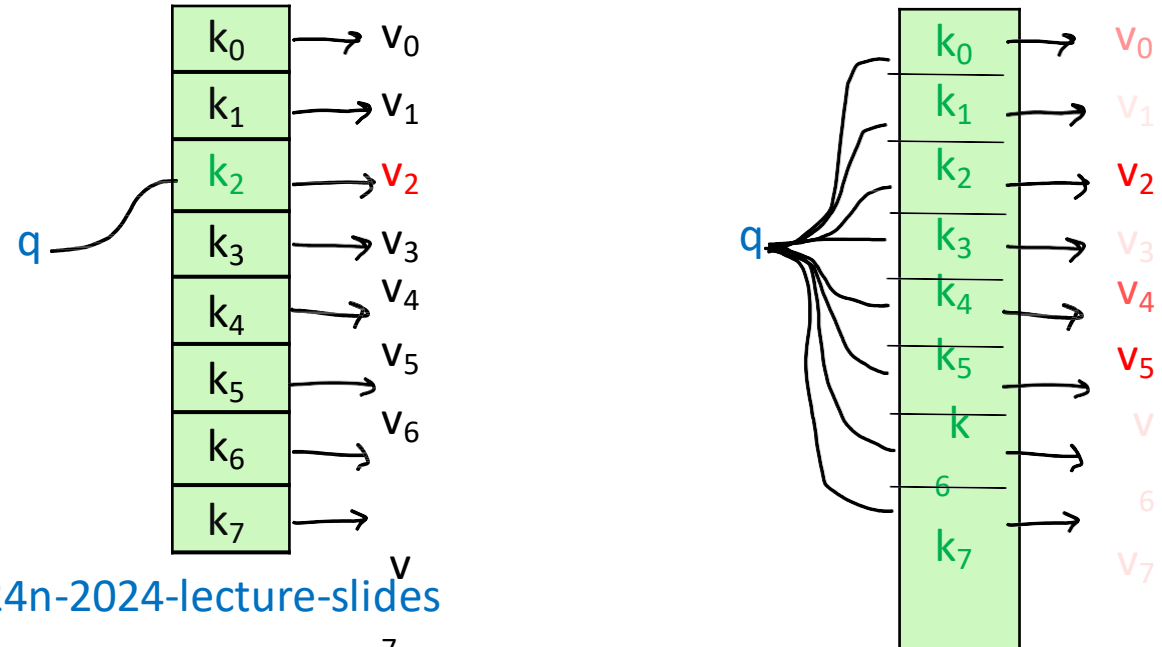
Self-Attention is the core building block of Transformer, so let's first focus on that!



Adapted from cs224n-2024-lecture-slides

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.



Adapted from cs224n-2024-lecture-slides

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**.

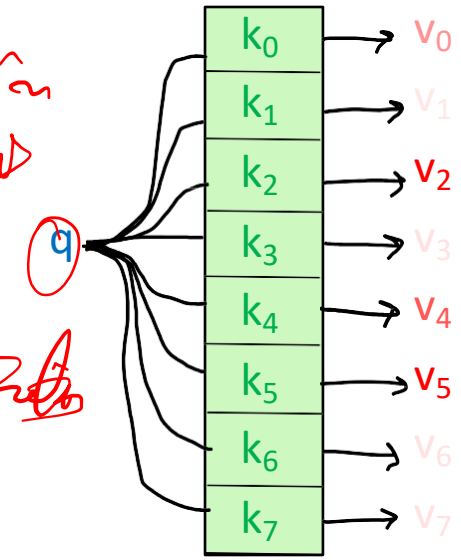
$$e_{ij} = q_i \cdot k_j$$

- 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**.

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



Adapted from cs224n-2024-lecture-slides

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.

$$A = softmax(E)$$

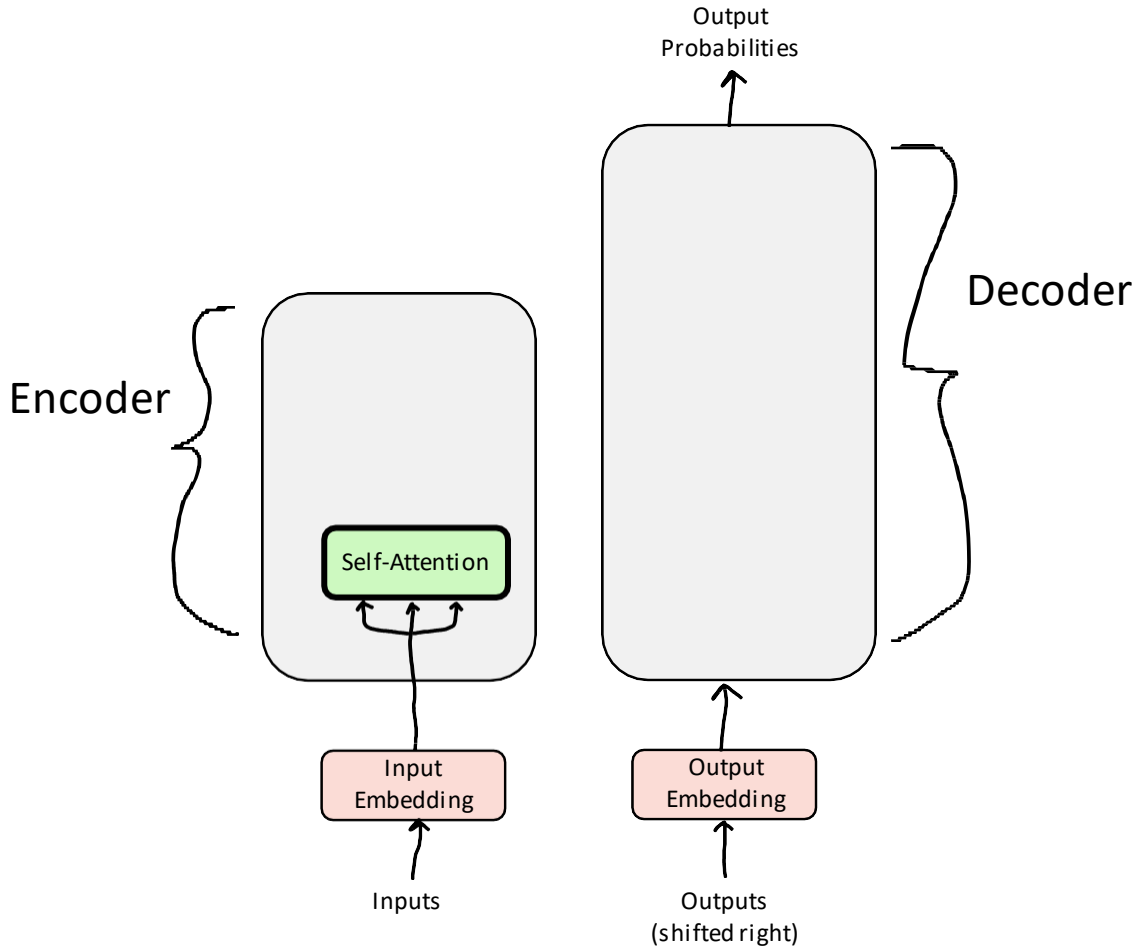
- Step 4: Take a weighted sum of **values**.

$$Output = AV$$

$$Output = softmax\left(QK^T\right)V$$

Adapted from cs224n-2024-lecture-slides

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



Adapted from cs224n-2024-lecture-slides

Attention is All you need!!

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

https://arxiv.org/pdf/1706.03762

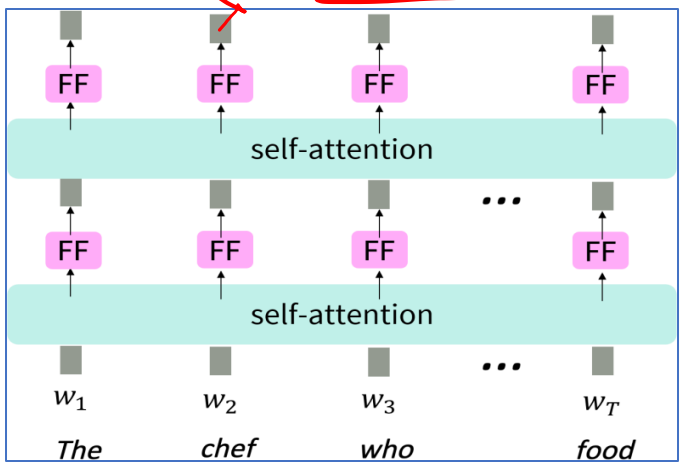
But attention isn't quite all you need!

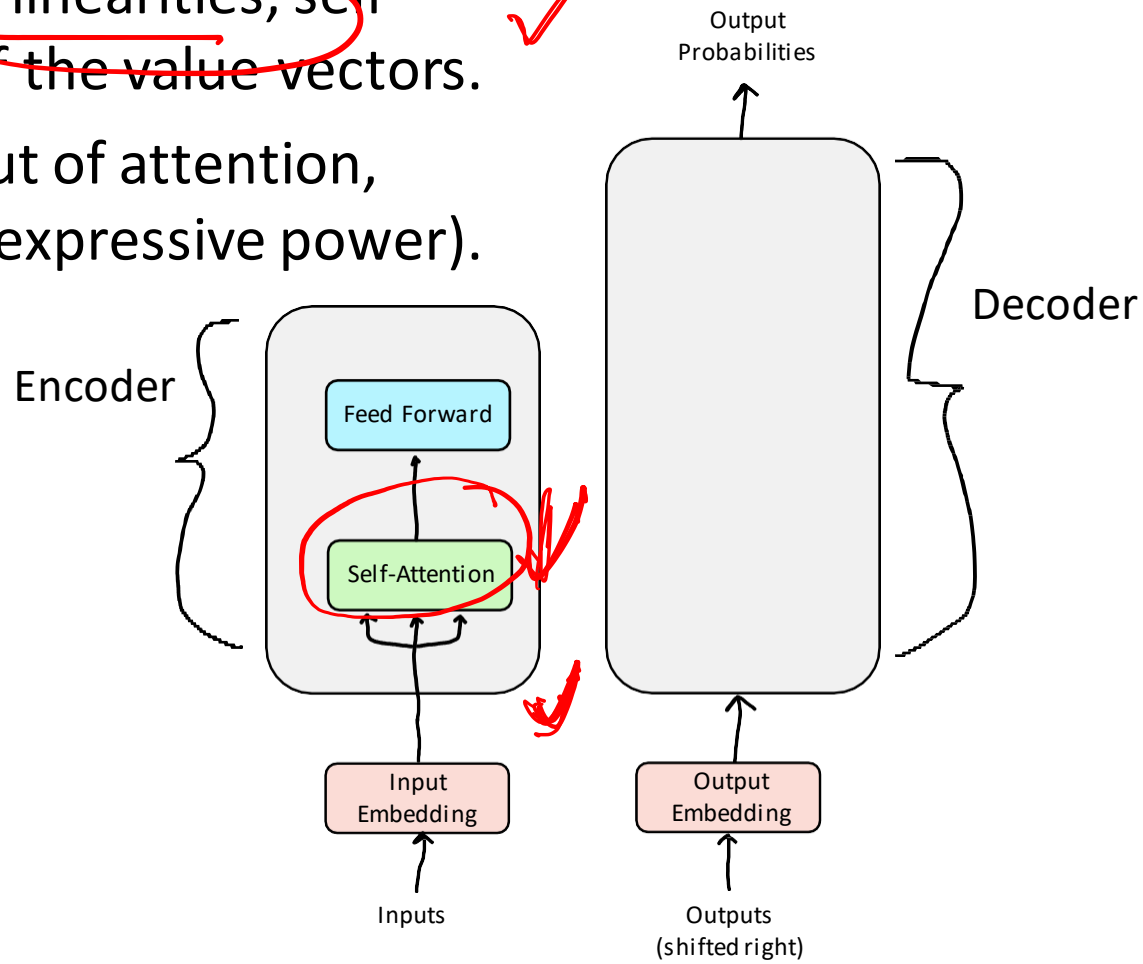
Problem: Since there are no element-wise non-linearities, self-attention is simply performing a re-averaging of the value vectors.

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

Equation for Feed Forward Layer

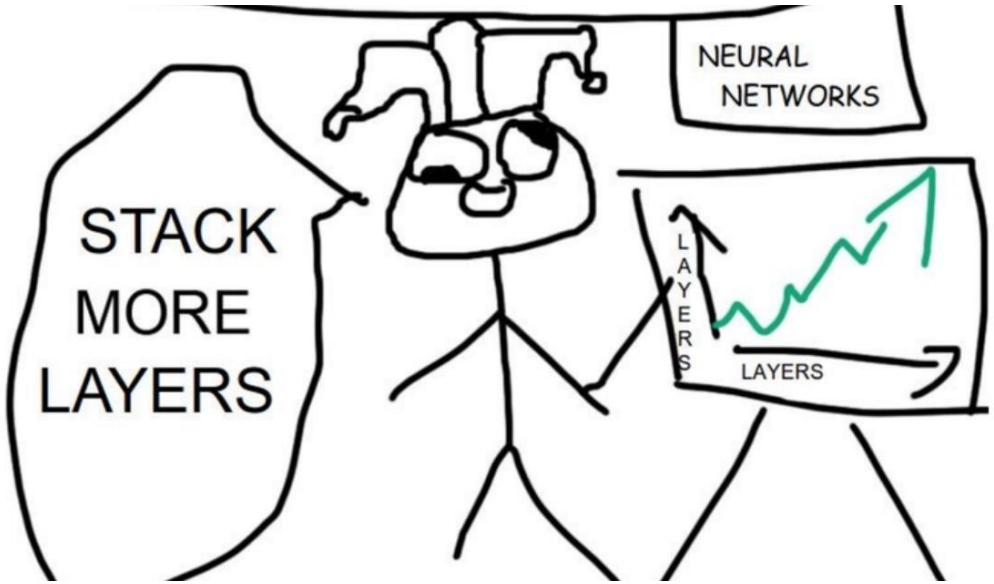
$$m_i = MLP(\text{output}_i)$$
$$= W_2 * \text{ReLU}(W_1 \times \text{output}_i + b_1) + b_2$$

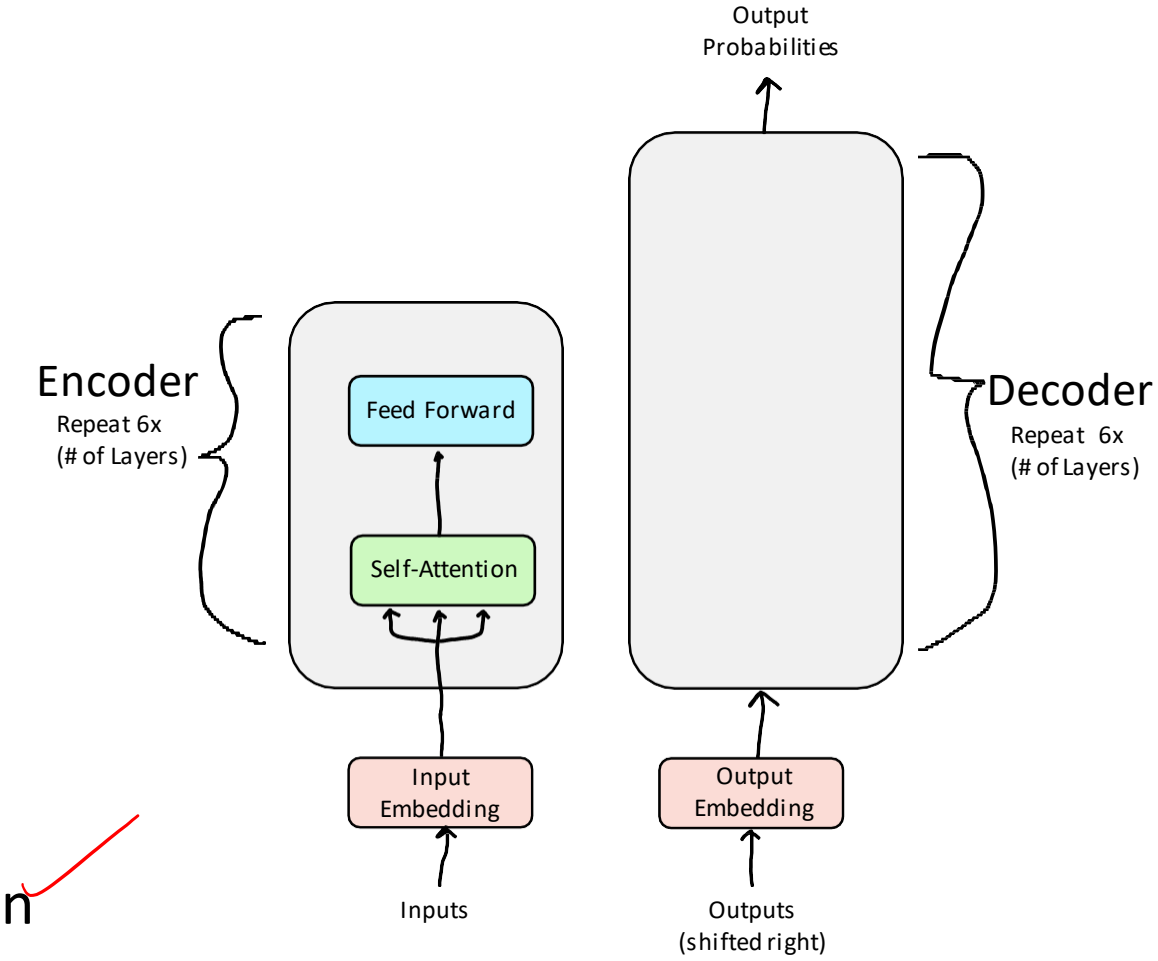




Adapted from cs224n-2024-lecture-slides

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

Adapted from cs224n-2024-lecture-slides

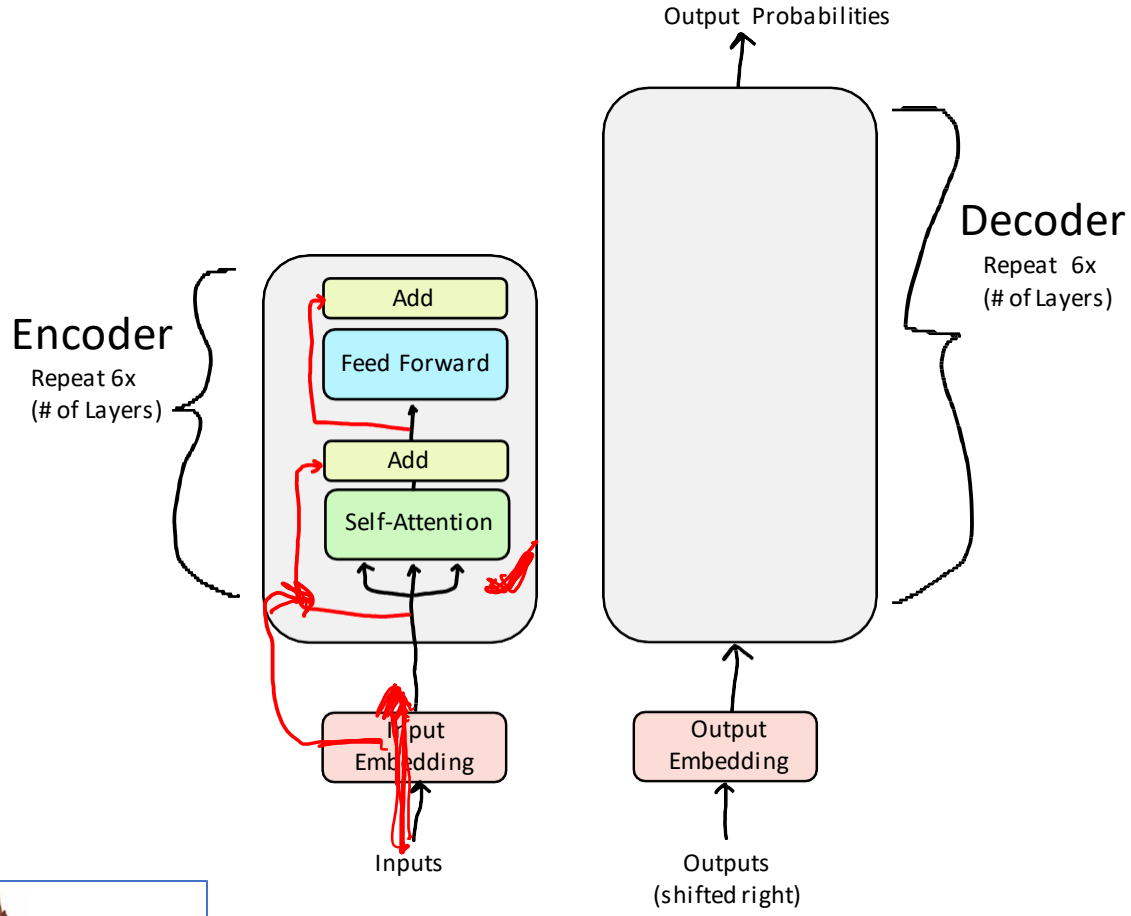
Training Trick #1: Residual Connections [He et al., 2016]

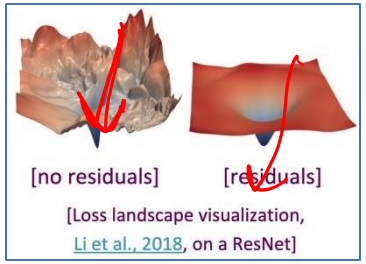
- Residual connections are a simple but powerful technique from computer vision.
- Deep networks are surprisingly bad at learning the identity function!
- Therefore, directly passing "raw" embeddings to the next layer can actually be very helpful!

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

- **This prevents the network from "forgetting" or distorting important information as it is processed by many layers.**

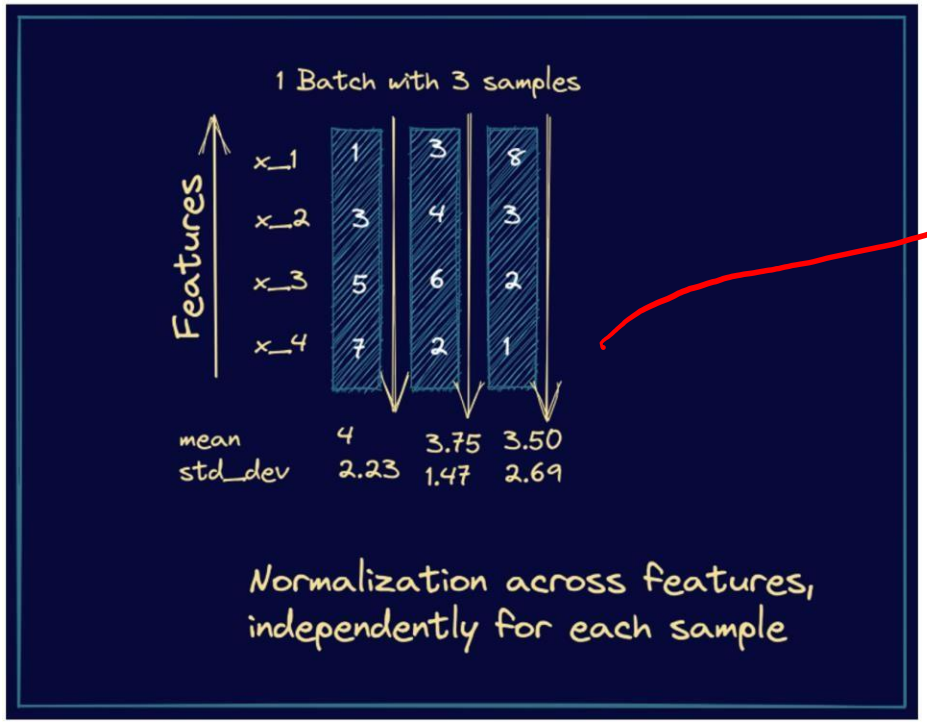
Residual connections are also thought to smooth the loss landscape and make training easier!





Adapted from cs224n-2024-lecture-slides

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

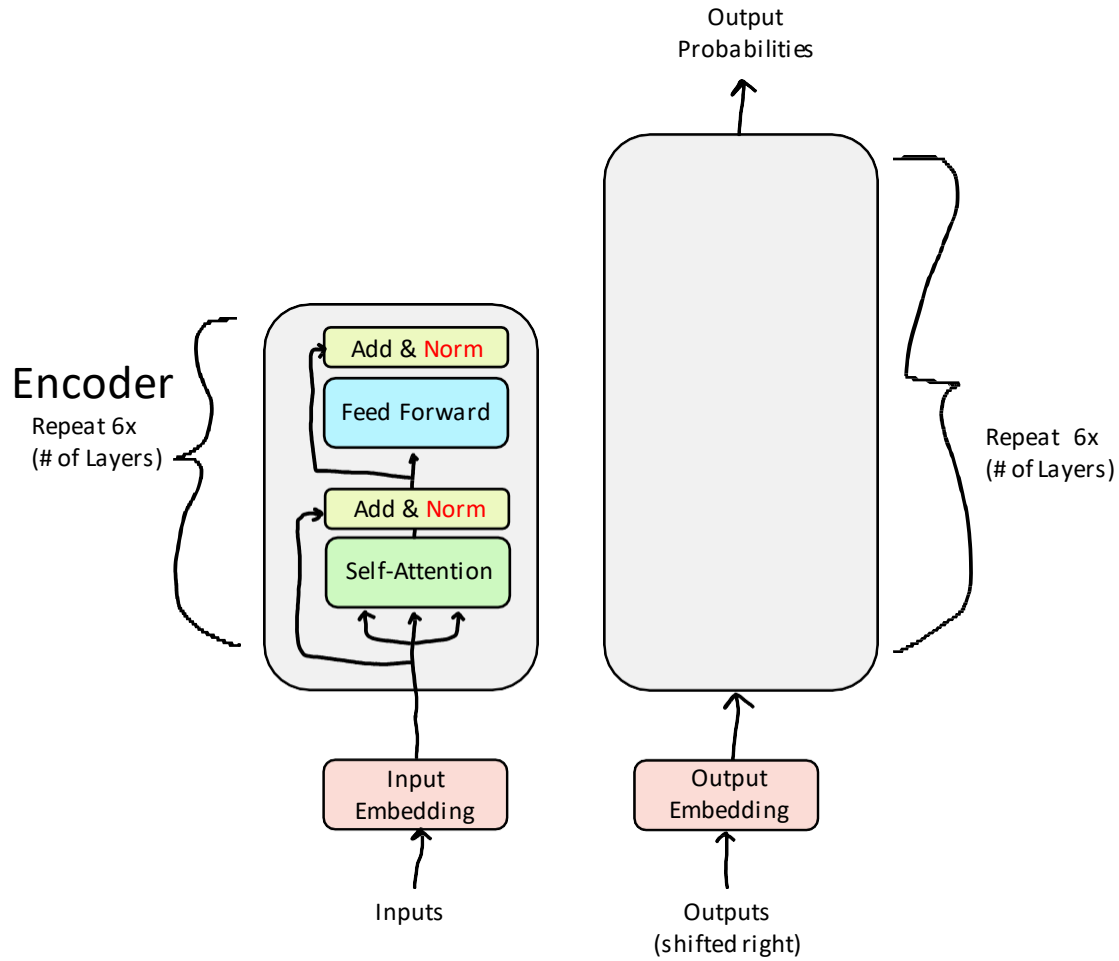


An Example of How LayerNorm Works (Image by Bala Priya C, Pinecone)

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

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

Hidden layers



Adapted from cs224n-2024-lecture-slides

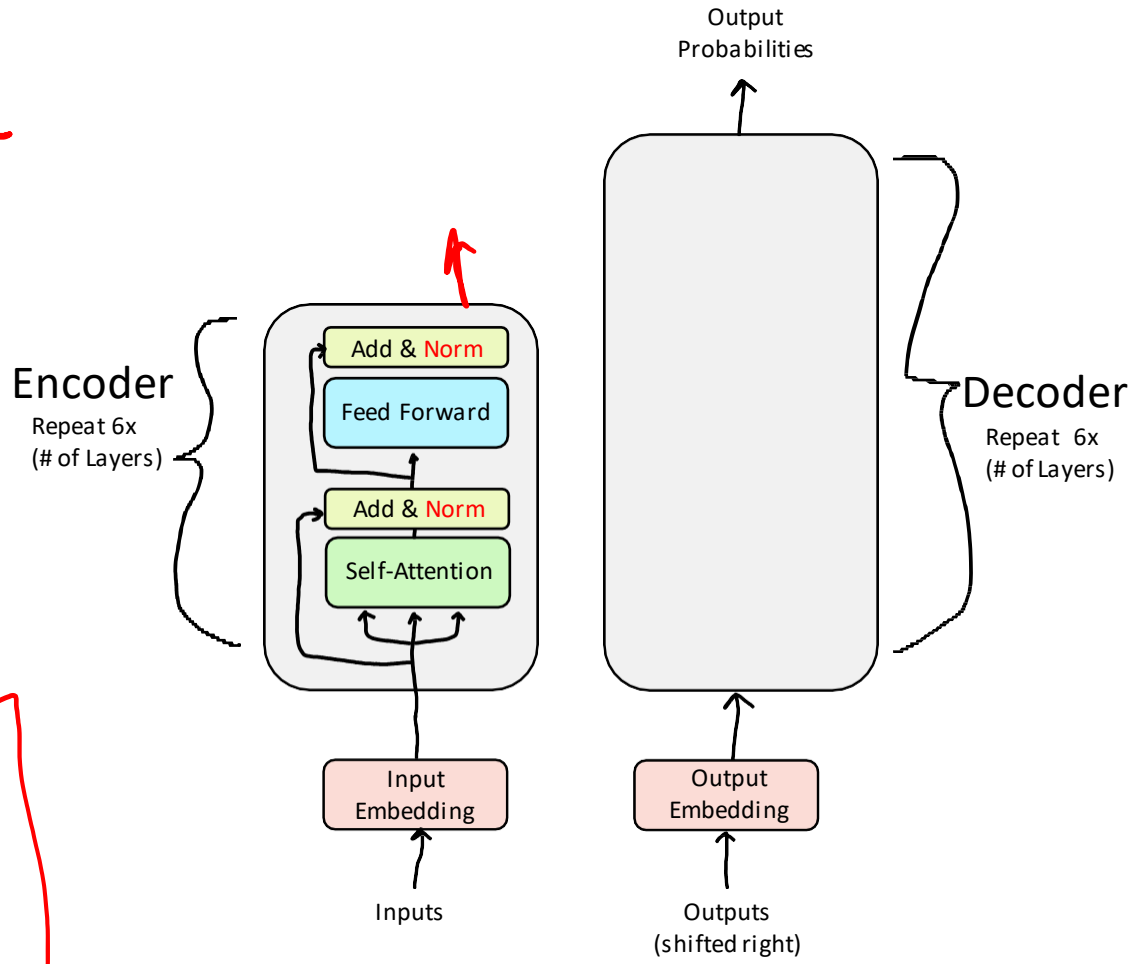
Training Trick #2: Layer Normalization [Ba et al., 2016]

- **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**.

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

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

Mean ✓ Variance ✓ SD



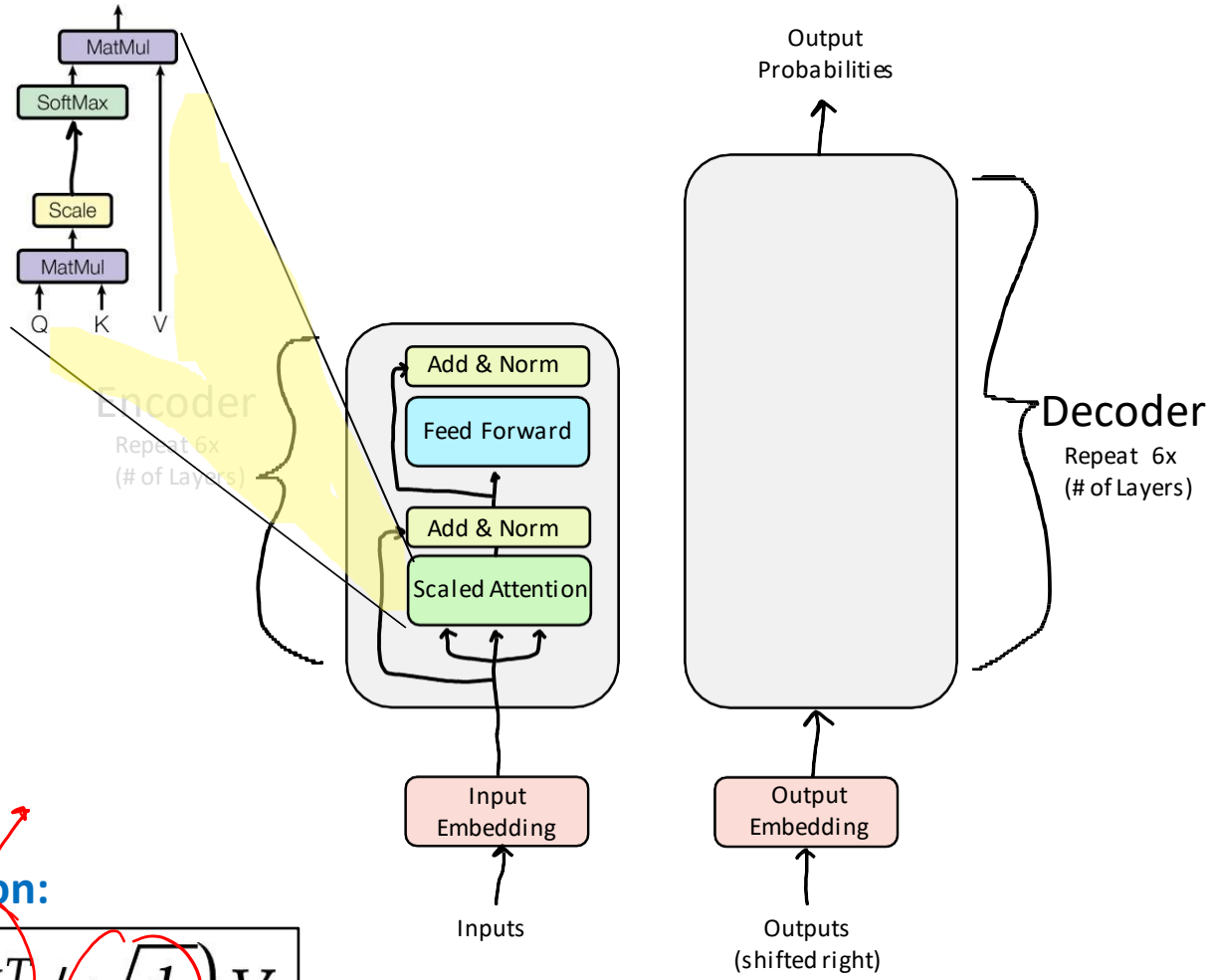
Adapted from cs224n-2024-lecture-slides

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:

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

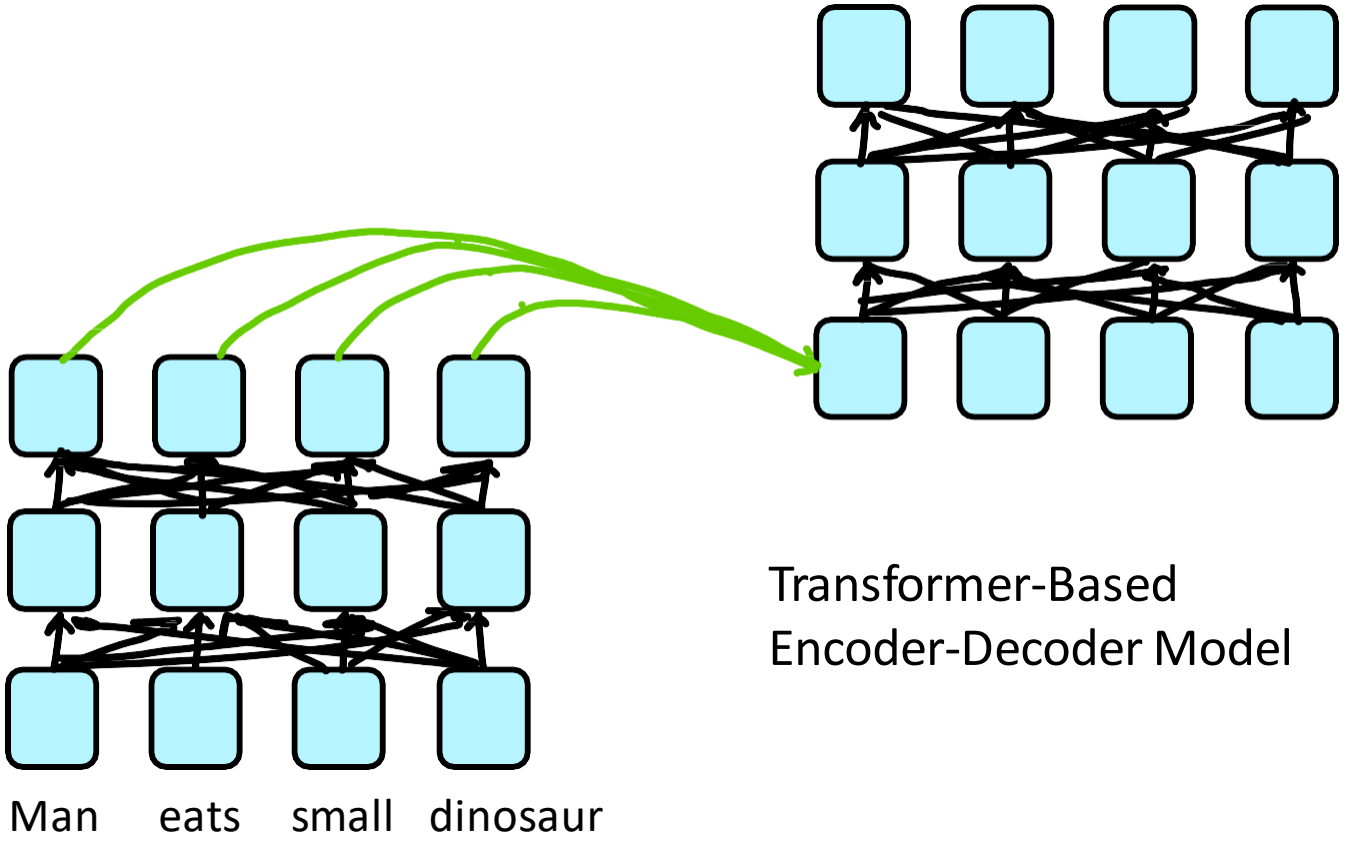
Adapted from cs224n-2024-lecture-slides

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."**



Transformer-Based
Encoder-Decoder Model

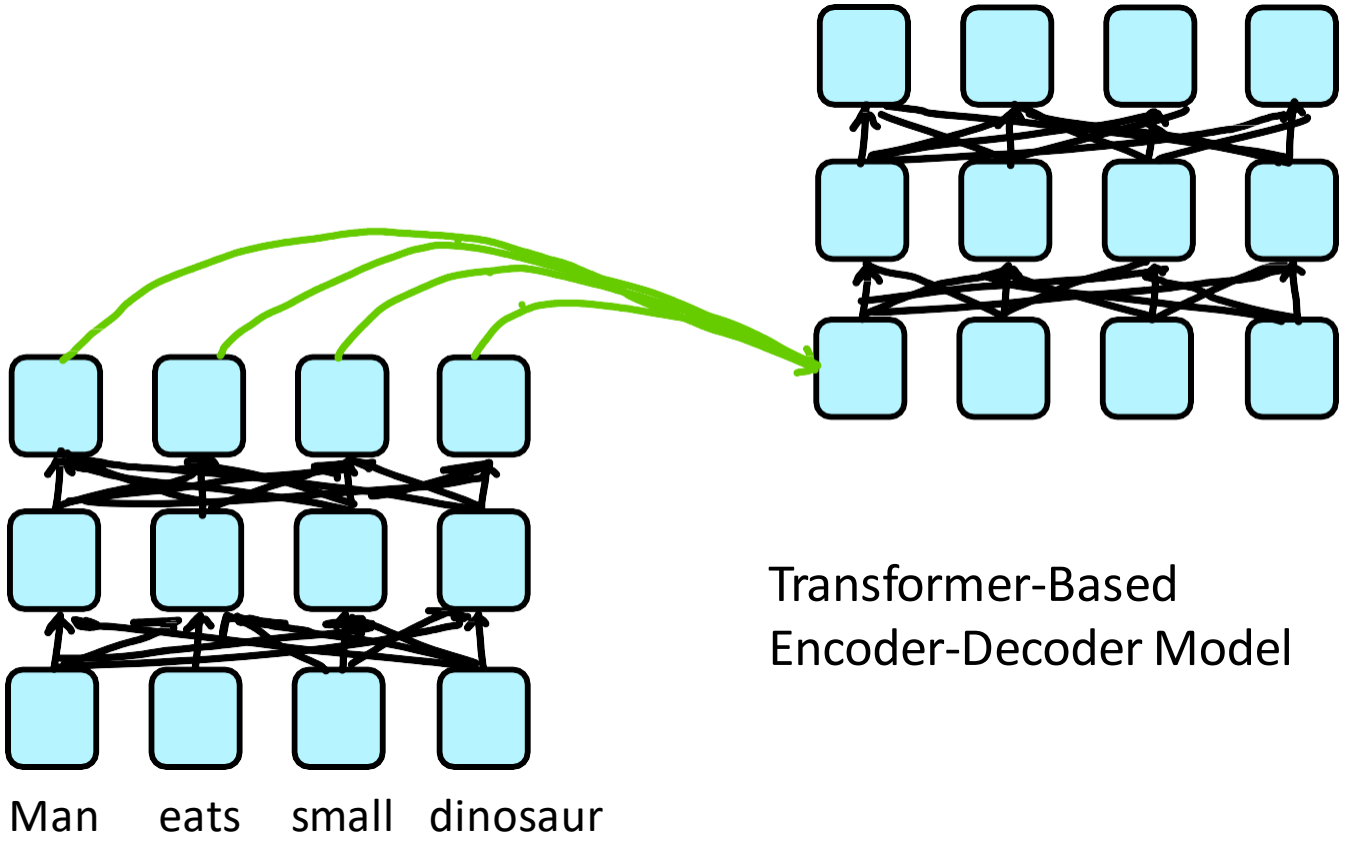
Adapted from cs224n-2024-lecture-slides

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!**

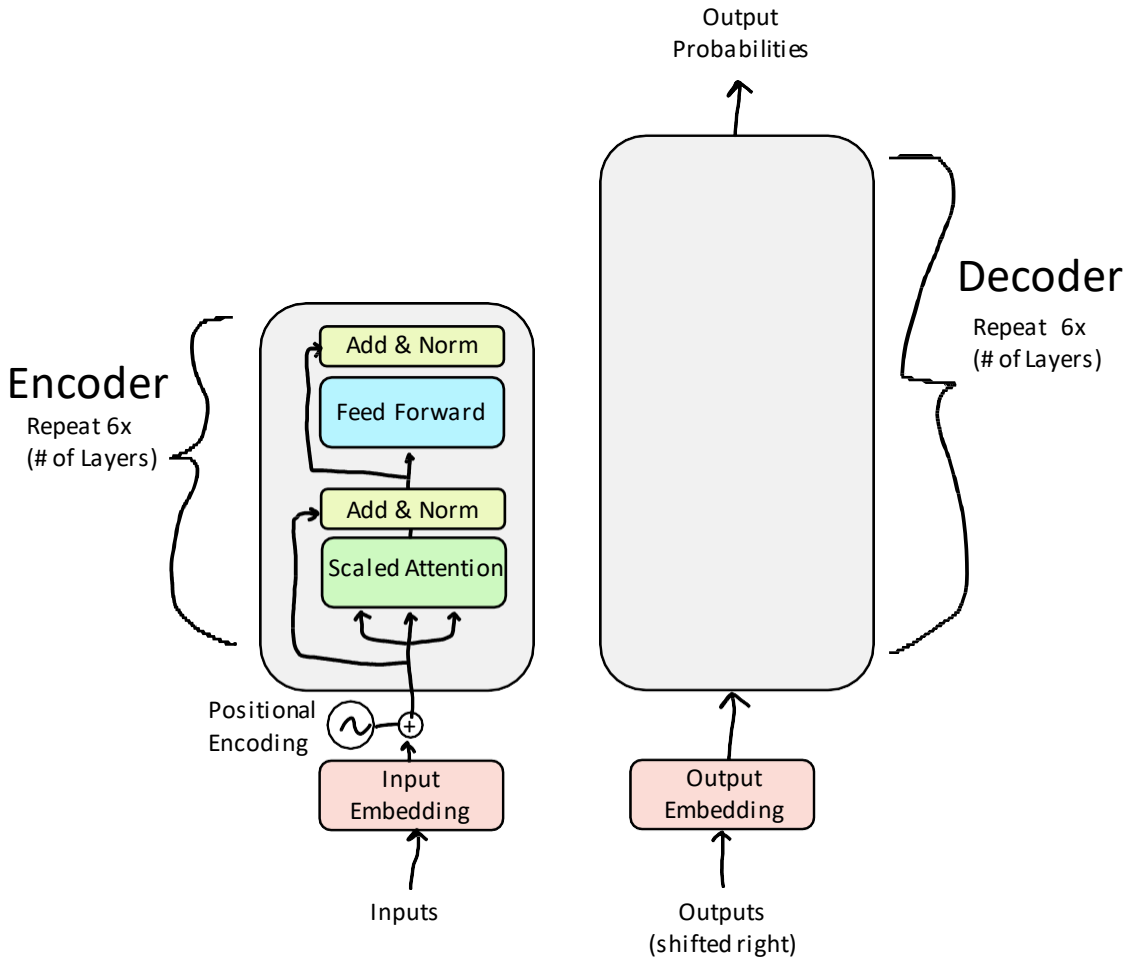
$$Output = softmax\left(QK^T / \sqrt{d_k}\right)V$$



Transformer-Based Encoder-Decoder Model

Adapted from cs224n-2024-lecture-slides

Solution: Inject Order Information through Positional Encodings!



Adapted from cs224n-2024-lecture-slides

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

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

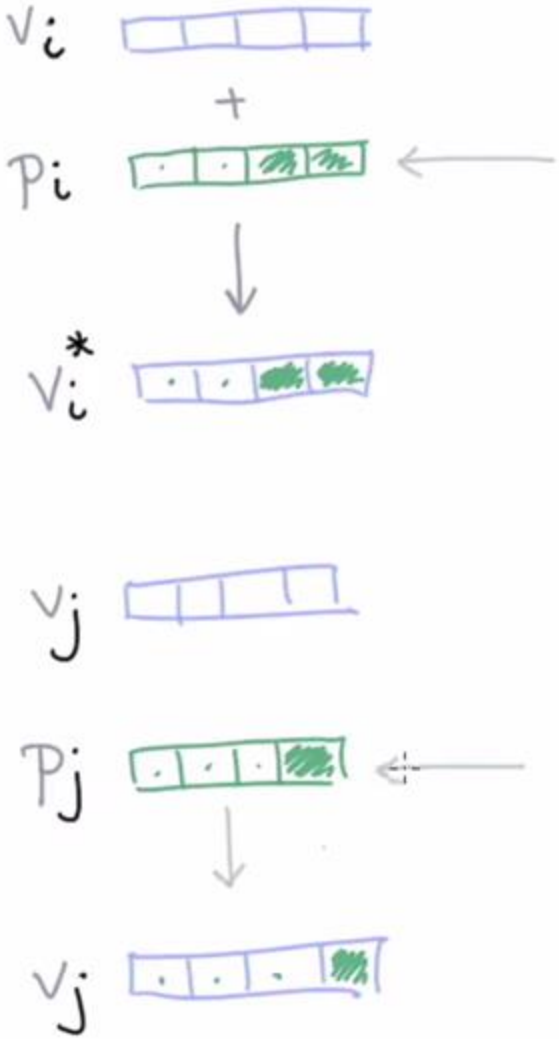
$$p_i \in \mathbb{R}^d \text{, for } i \in \{1,2, \dots, T\} \text{ are position vectors}$$

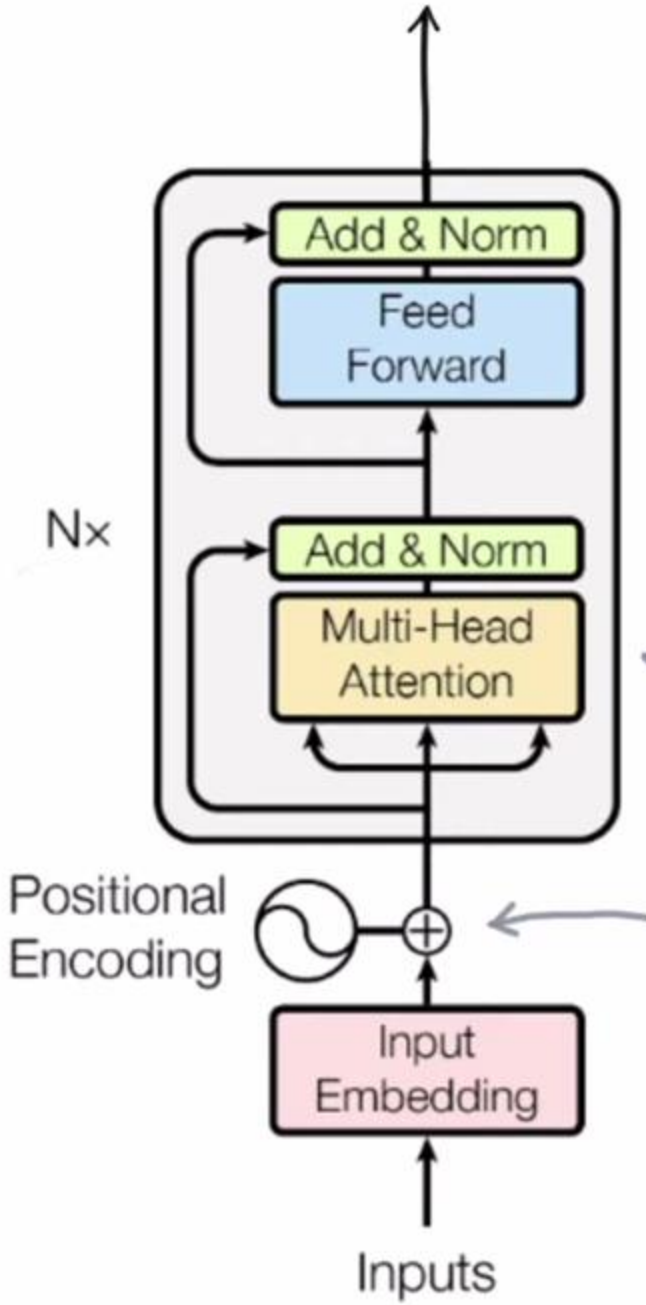
- Don't worry about what the p_i are made of yet!
- Easy to incorporate this info into our self-attention block: just add the p_i to our inputs!

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

Adapted from cs224n-2024-lecture-slides

Transformer Encoder

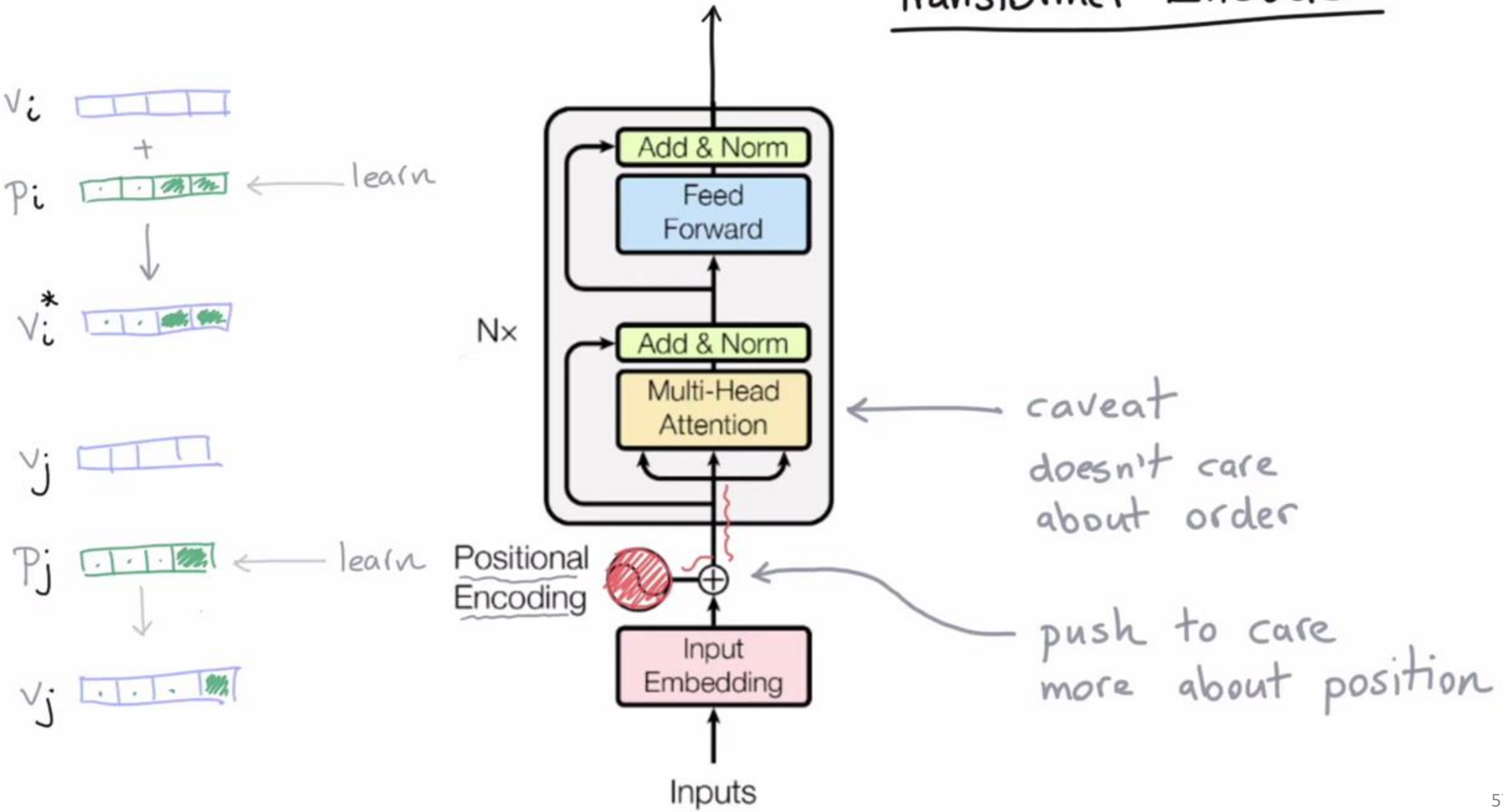




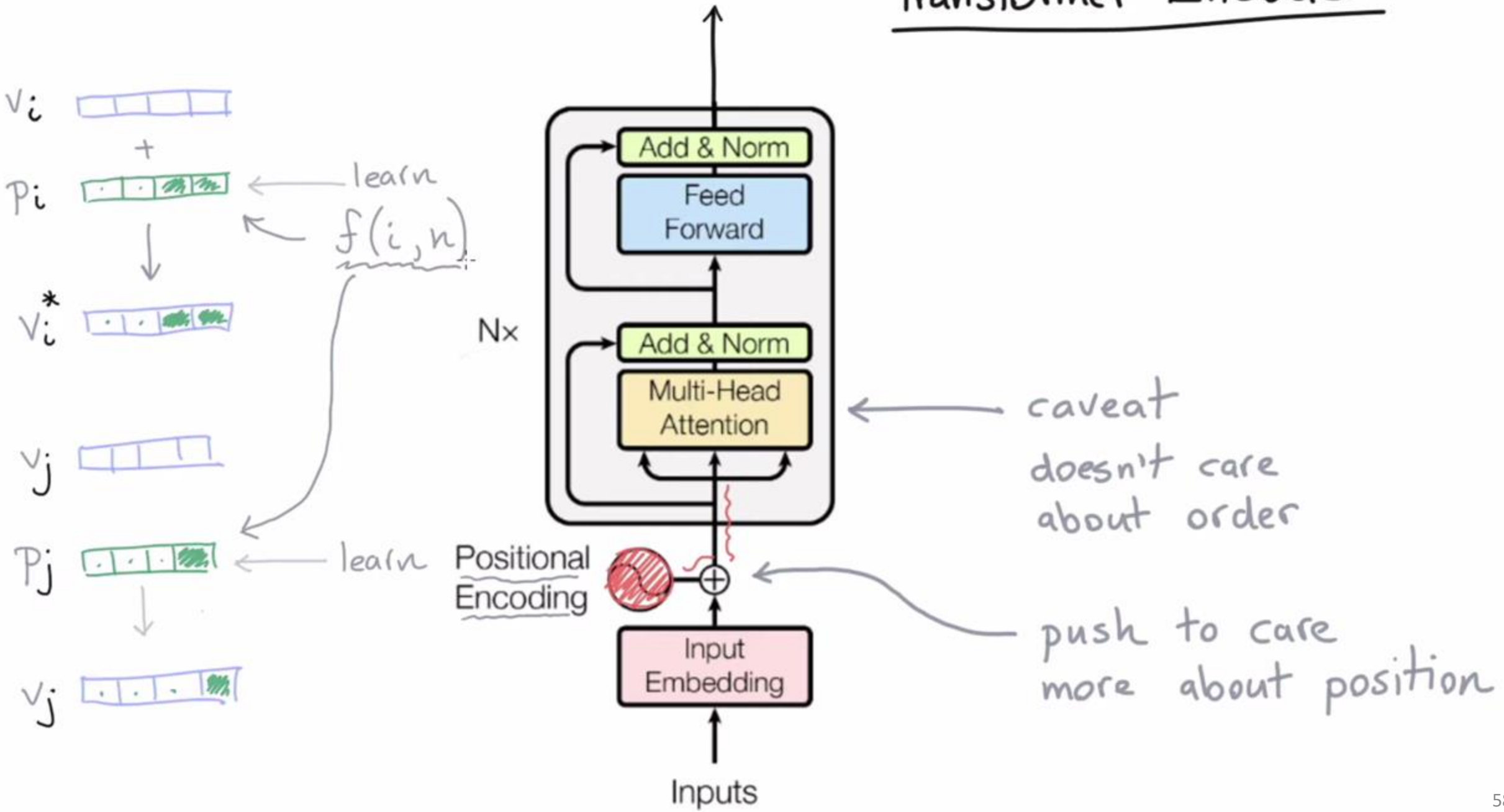
caveat
doesn't care
about order

push to care
more about position

Transformer Encoder



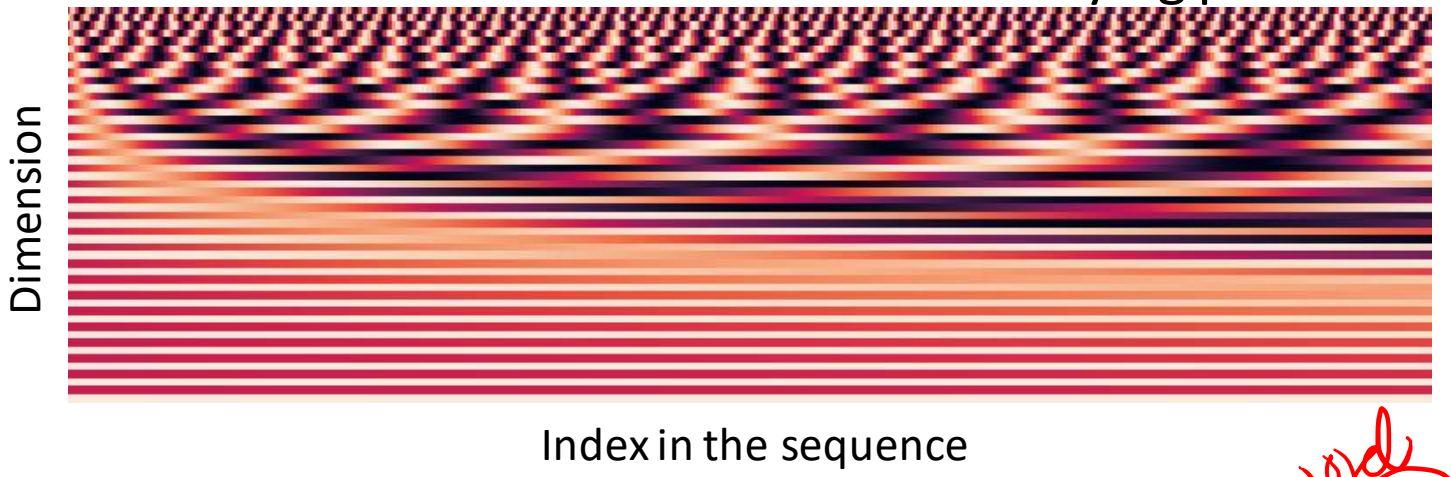
Transformer Encoder



Position representation vectors through sinusoids (original)

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

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



- 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

Adapted from cs224n-2024-lecture-slides Image: https://timodenk.com/blog/linear-relationships-in-the-transformers-positional-encoding/

Extension: Self-Attention w/ Relative Position Encodings

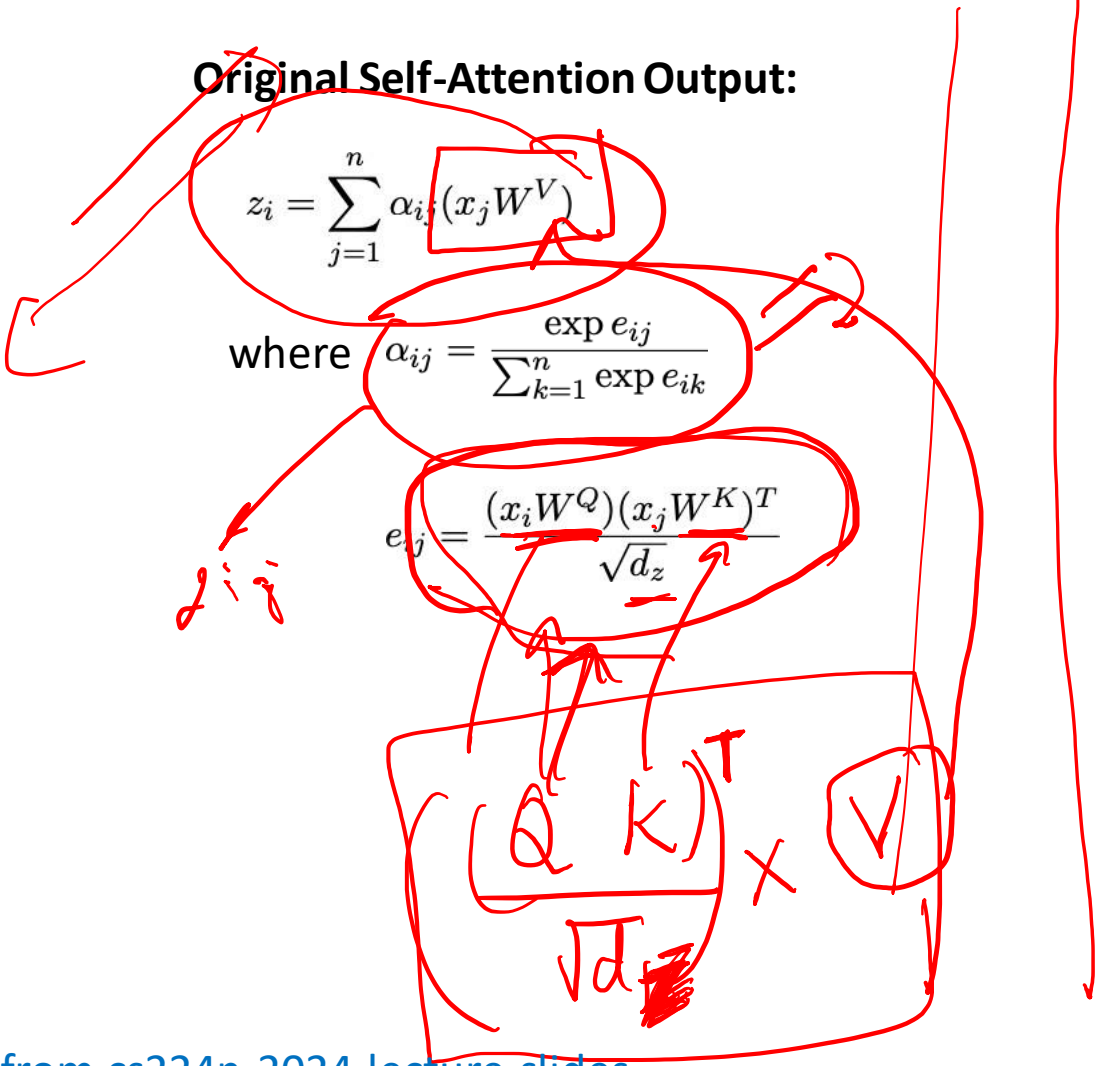
Key Insight: The most salient position information is the relationship (e.g. "cat" is the word before "eat") between words, rather than their absolute position (e.g. "cat" is word 2).

Original Self-Attention Output:

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

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

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



Relation-Aware Self-Attention Output:

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

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

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

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

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

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

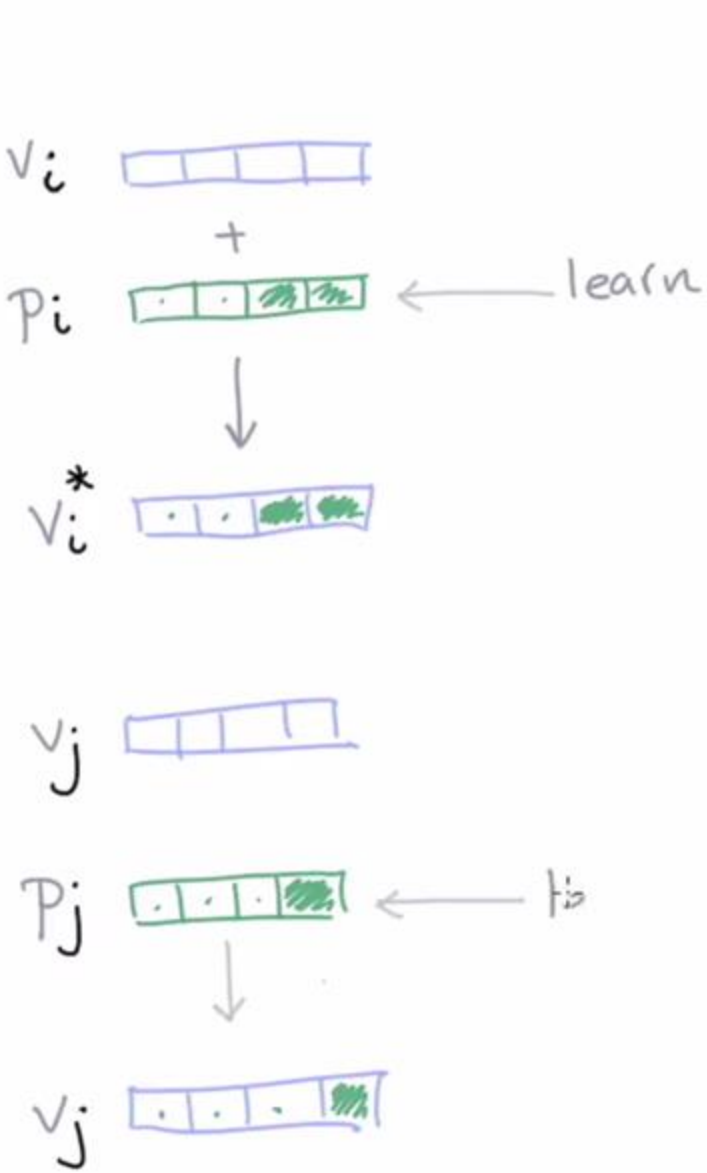
We then learn relative position representations $w^K = (w_{-k}^K, \ldots, w_k^K)$ and $w^V = (w_{-k}^V, \ldots, w_k^V)$

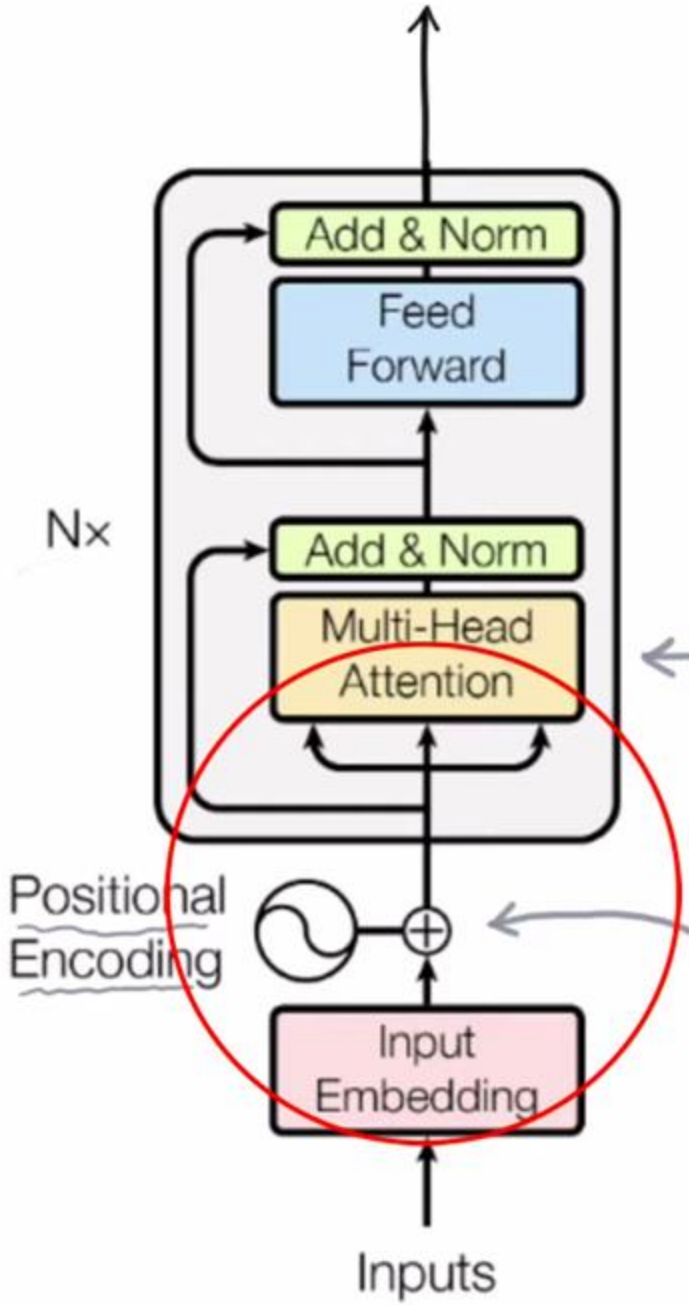
k	EN-DE BLEU
0	12.5
1	25.5
2	25.8
4	25.9
16	25.8
64	25.9
256	25.8

Table and Equations From [Shaw et al., 2018]

Adapted from cs224n-2024-lecture-slides

Transformer Encoder

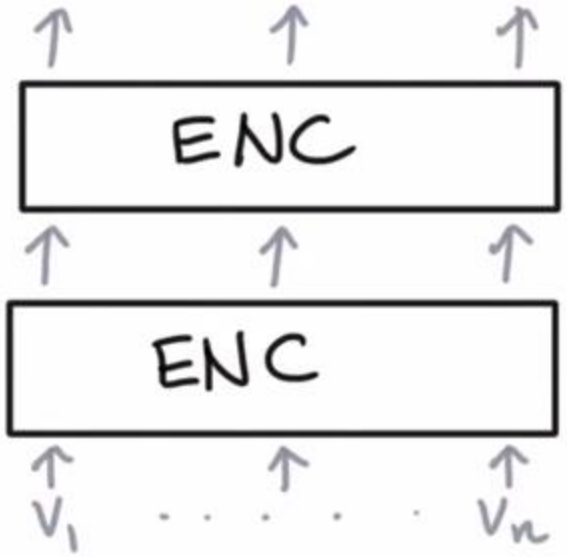




caveat
doesn't care about order

push to care more about position

Transformer Encoder

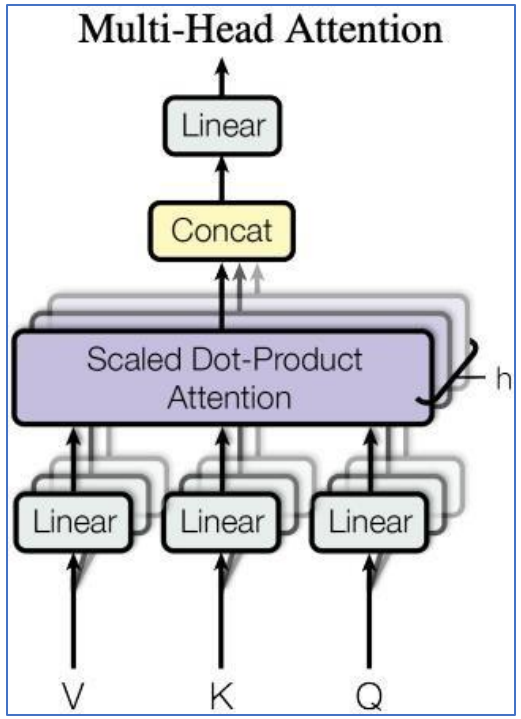


hyper param

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.

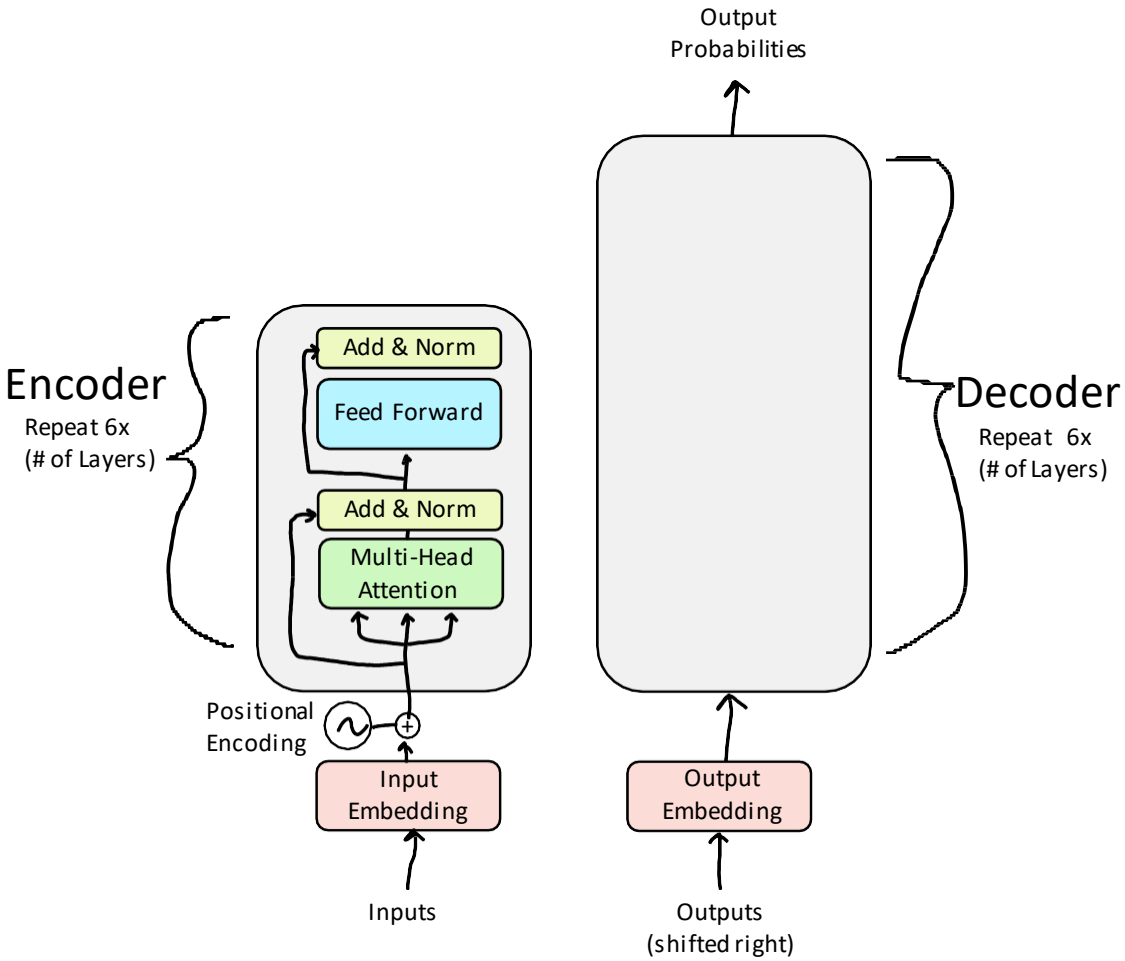


[Vaswani et al. 2017]

Wizards of the Coast, Artist: Todd Lockwood

Adapted from cs224n-2024-lecture-slides

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



Adapted from cs224n-2024-lecture-slides

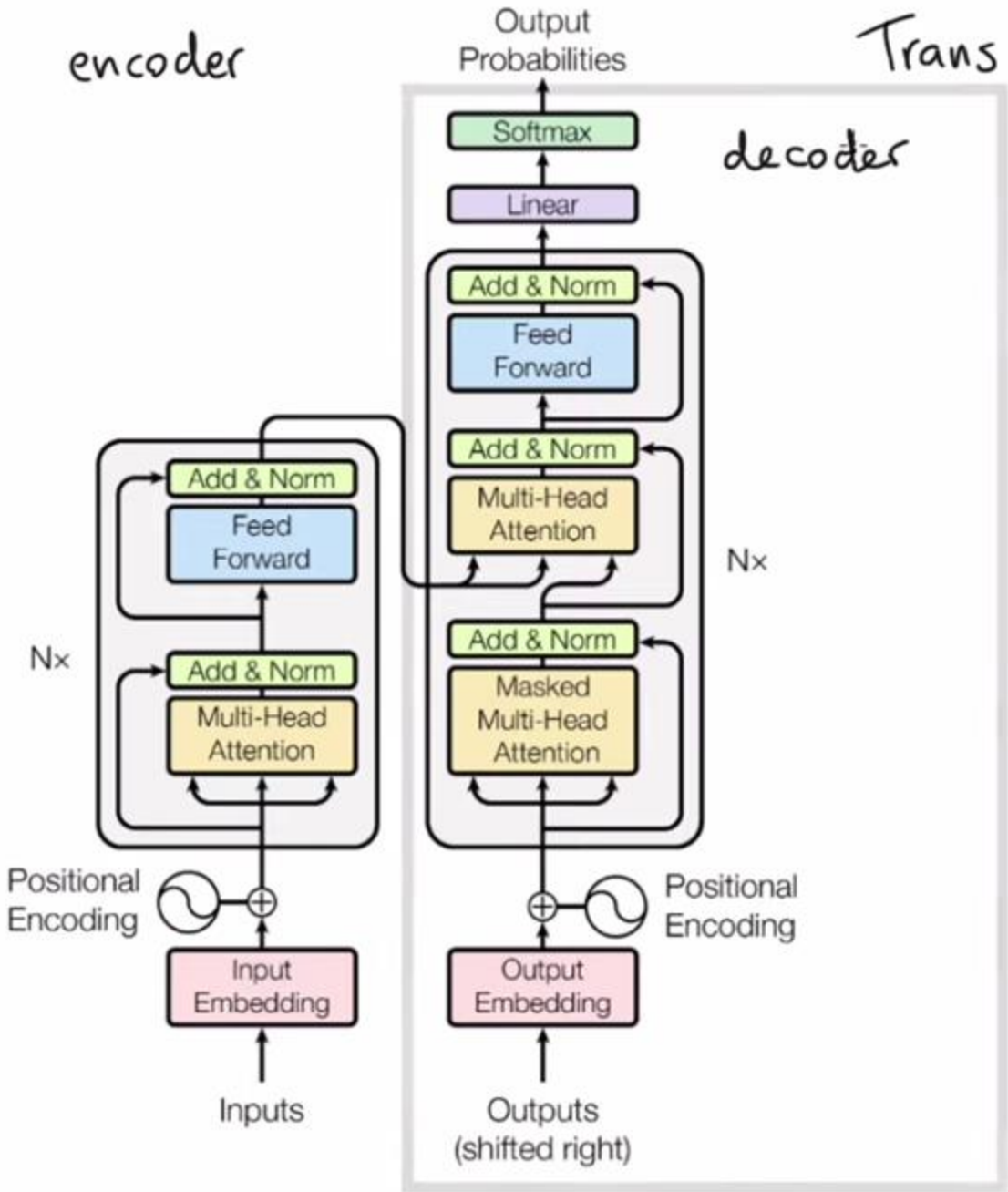


Figure 1: The Transformer - model architecture.

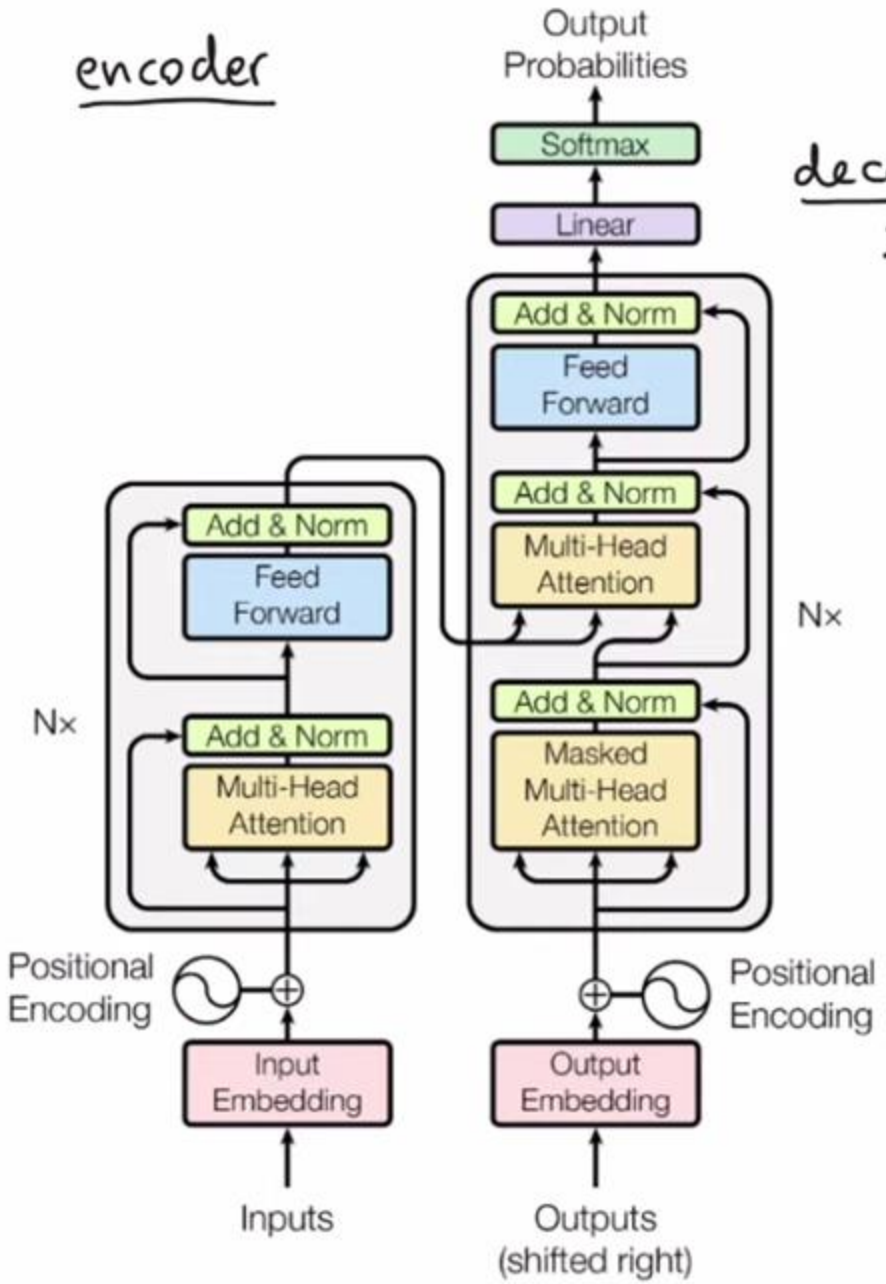
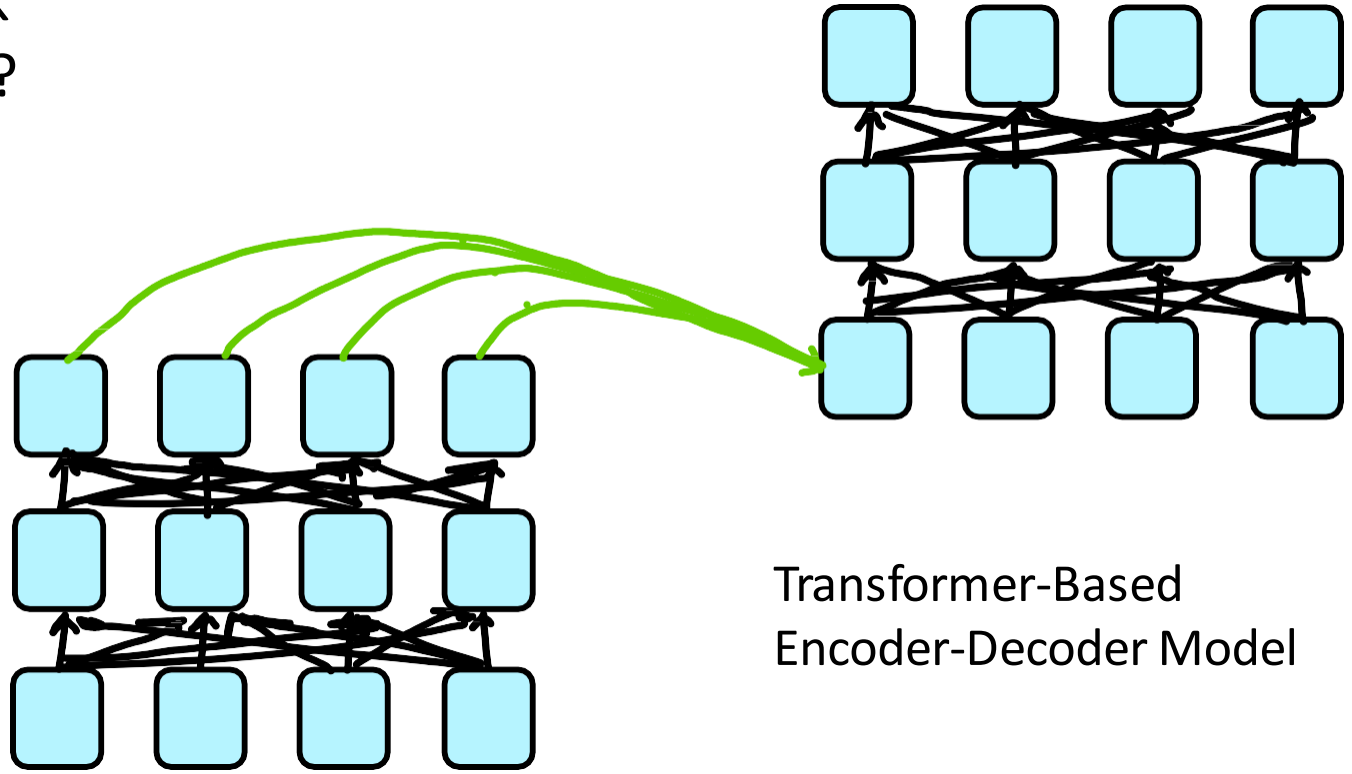


Figure 1: The Transformer - model architecture.

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?

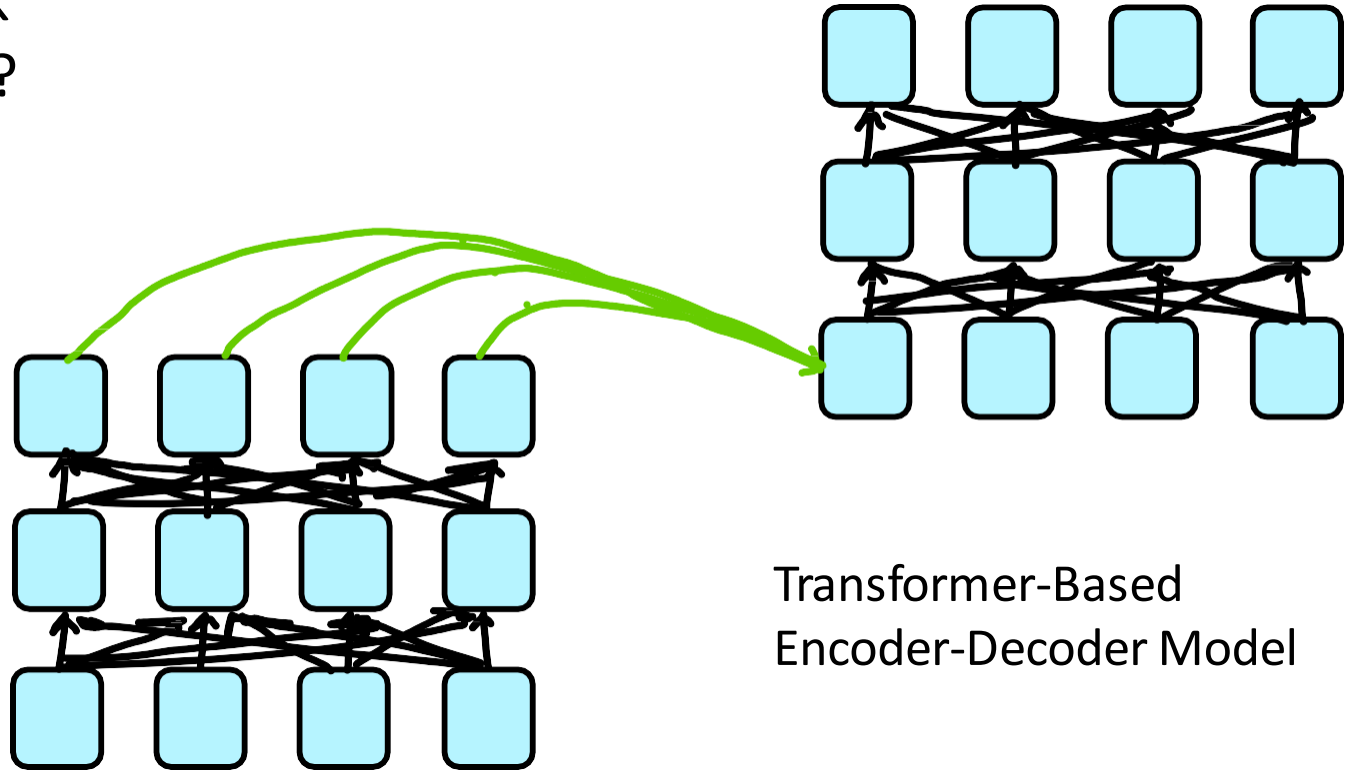


Transformer-Based Encoder-Decoder Model

Adapted from cs224n-2024-lecture-slides

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.



Transformer-Based Encoder-Decoder Model

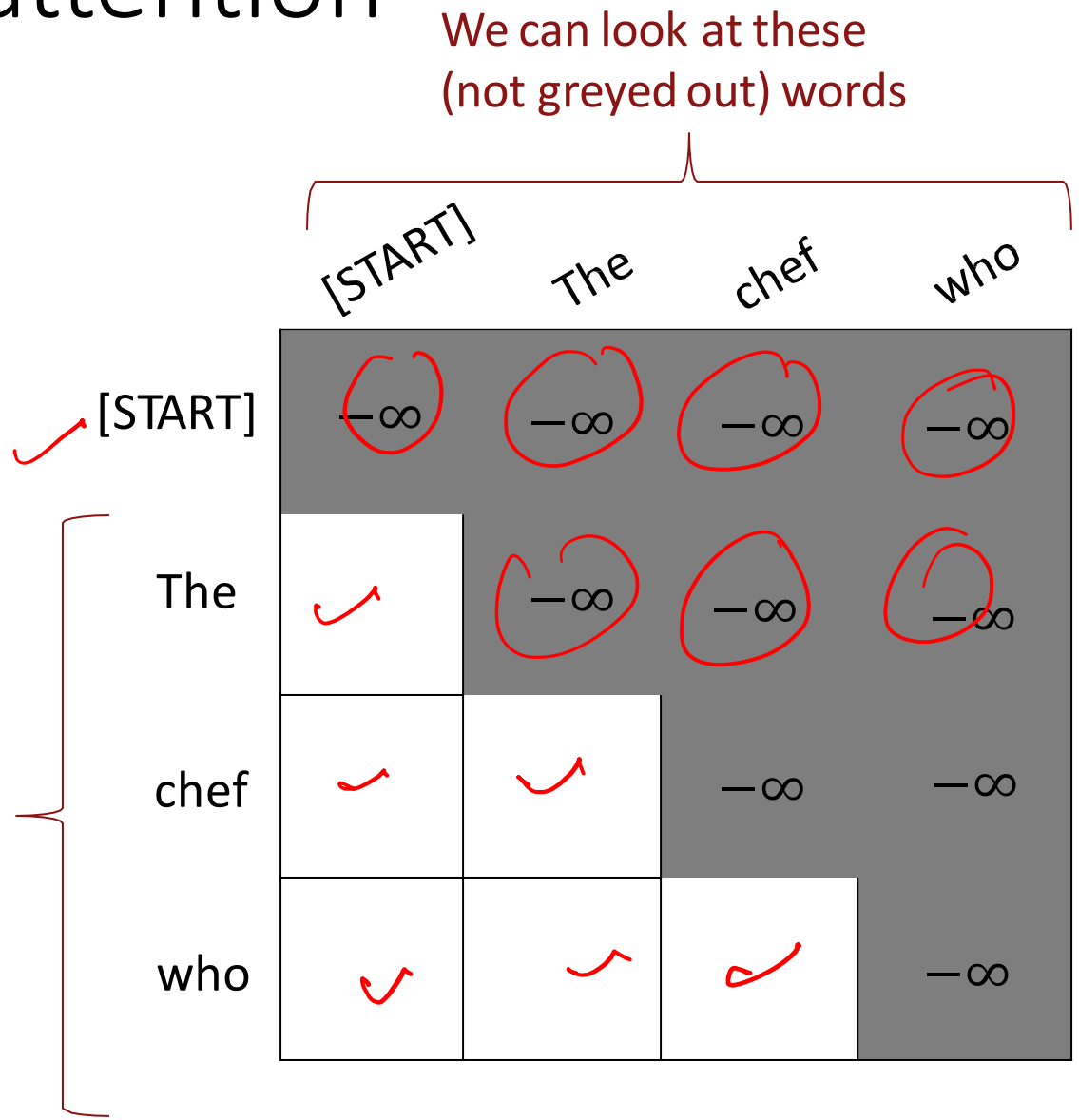
Adapted from cs224n-2024-lecture-slides

Masking the future in self-attention

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

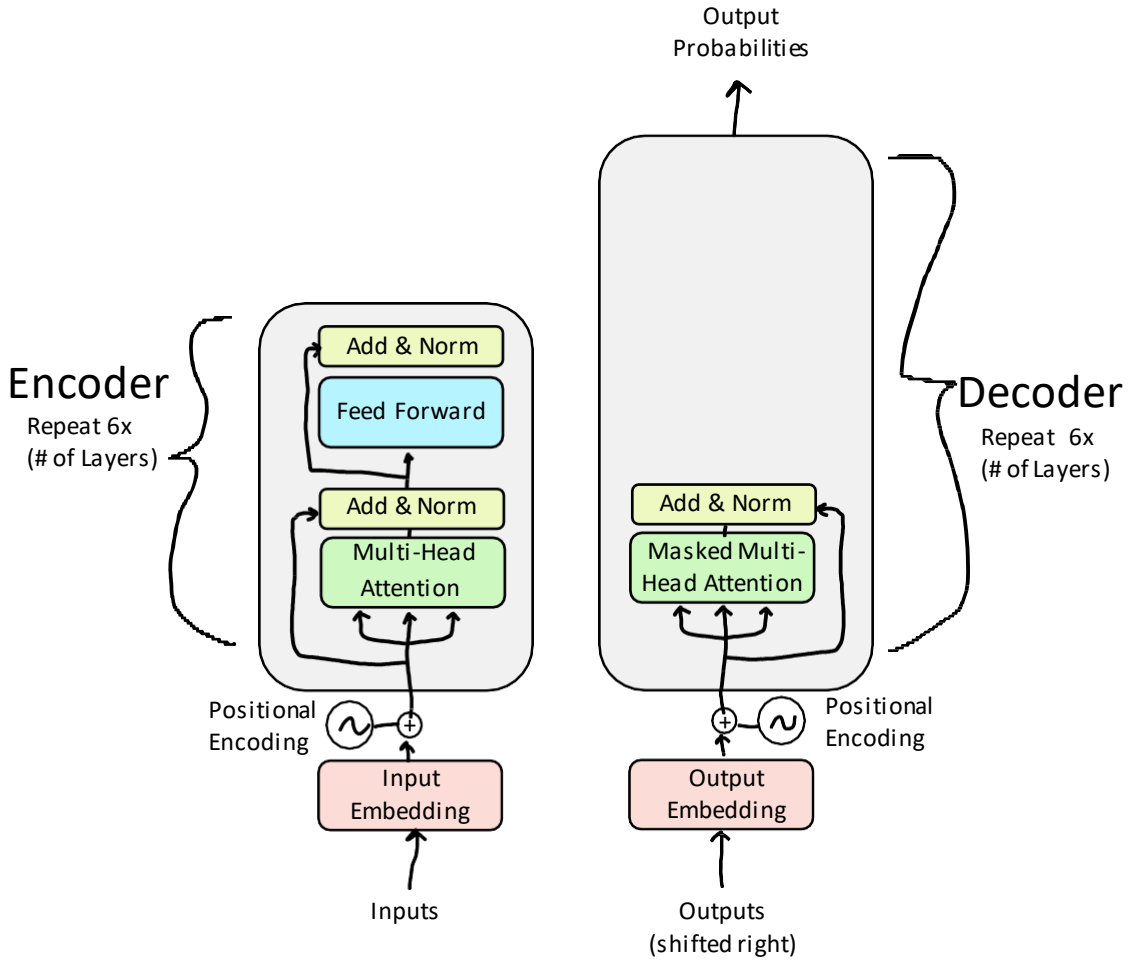
- At every timestep, we could change the set of **keys and queries** to include only past words. (Inefficient!)

- To enable parallelization, we **mask out attention** to future words by setting attention scores to $-\infty$.



Adapted from cs224n-2024-lecture-slides

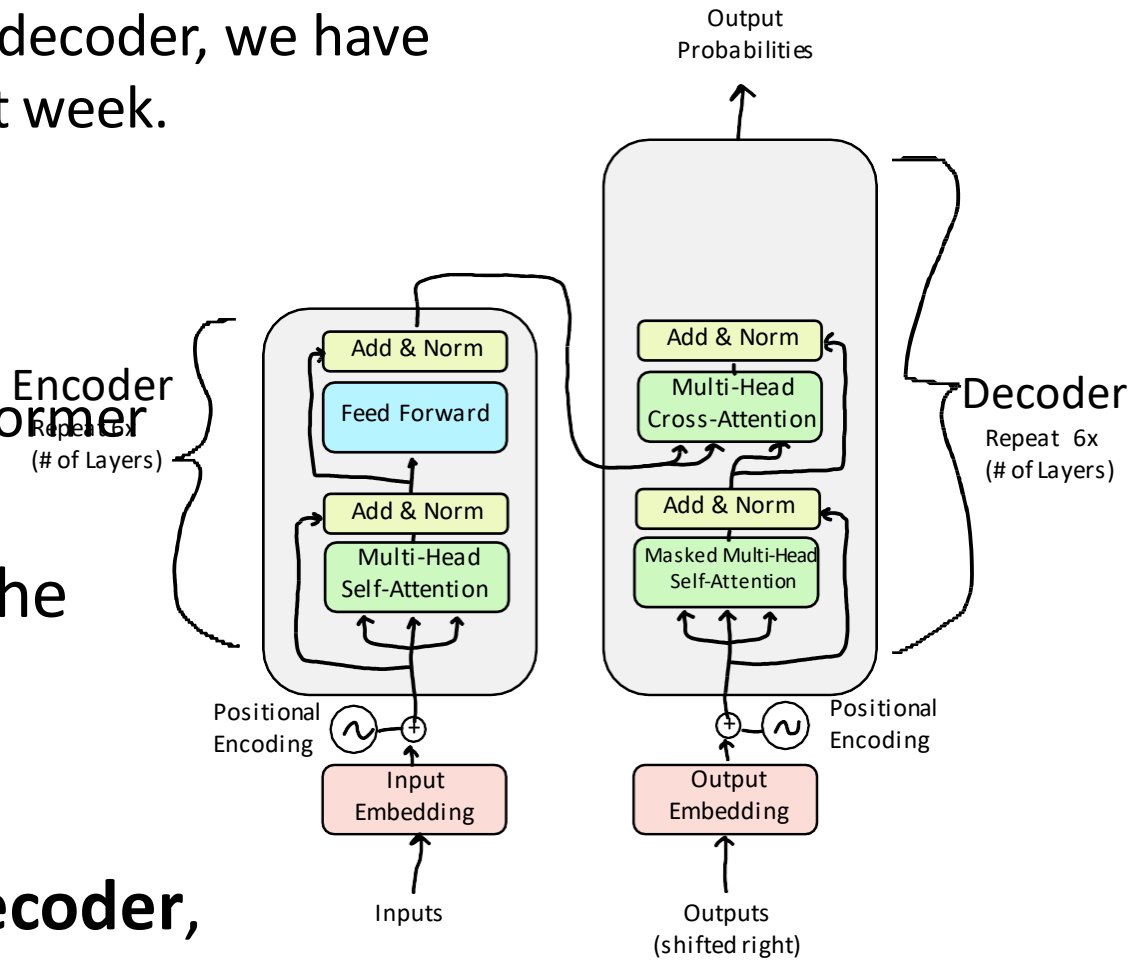
Decoder: Masked Multi-Headed Self-Attention



Adapted from cs224n-2024-lecture-slides

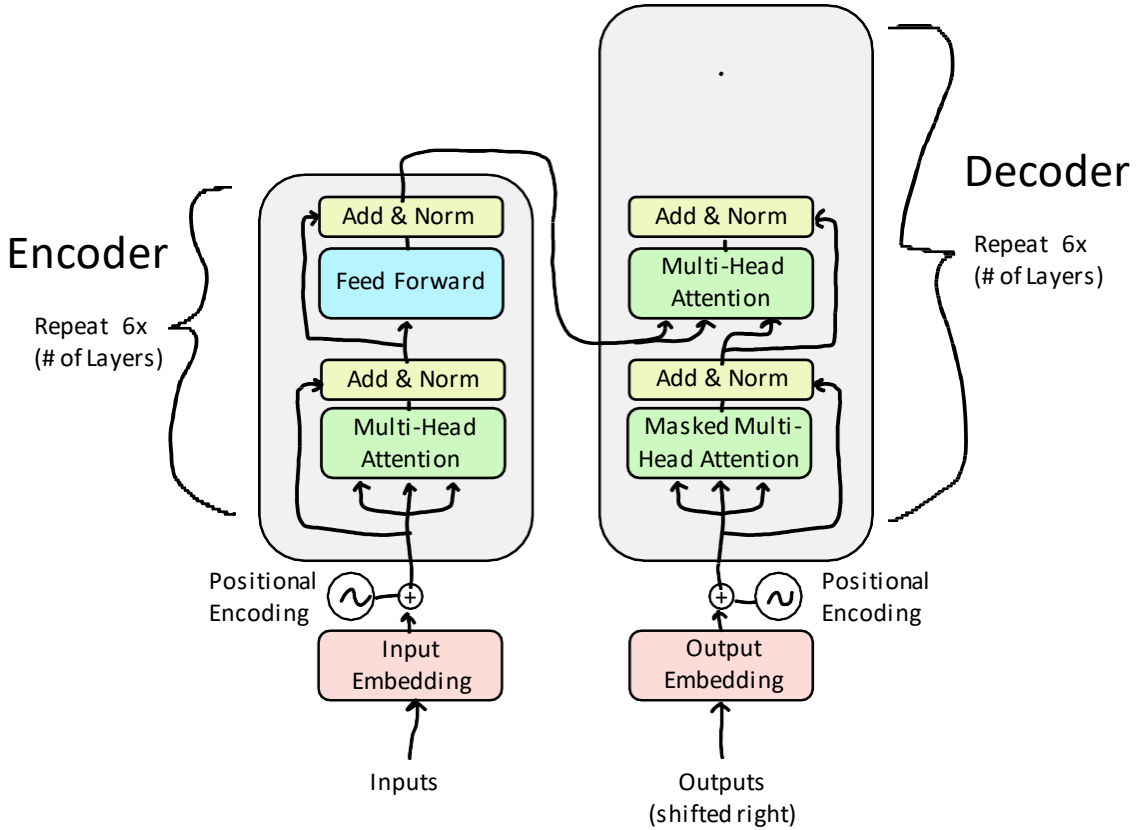
Encoder-Decoder Attention

- 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 $h_1, \ldots, h_T$ be **output** vectors **from** the Transformer **encoder**; $x_i \in \mathbb{R}^d$
- Let $z_1, \ldots, z_T$ 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**, $q_i = Qz$.



Adapted from cs224n-2024-lecture-slides

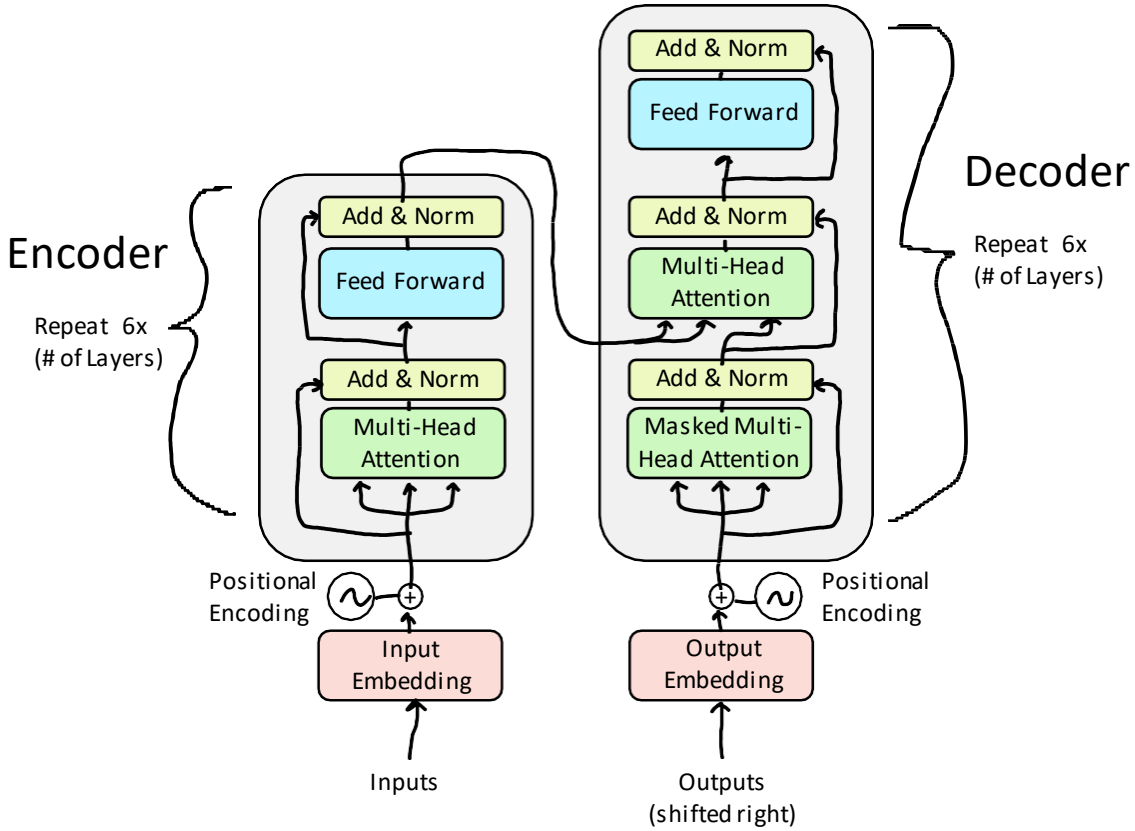
Decoder: Finishing touches!



Adapted from cs224n-2024-lecture-slides

Decoder: Finishing touches!

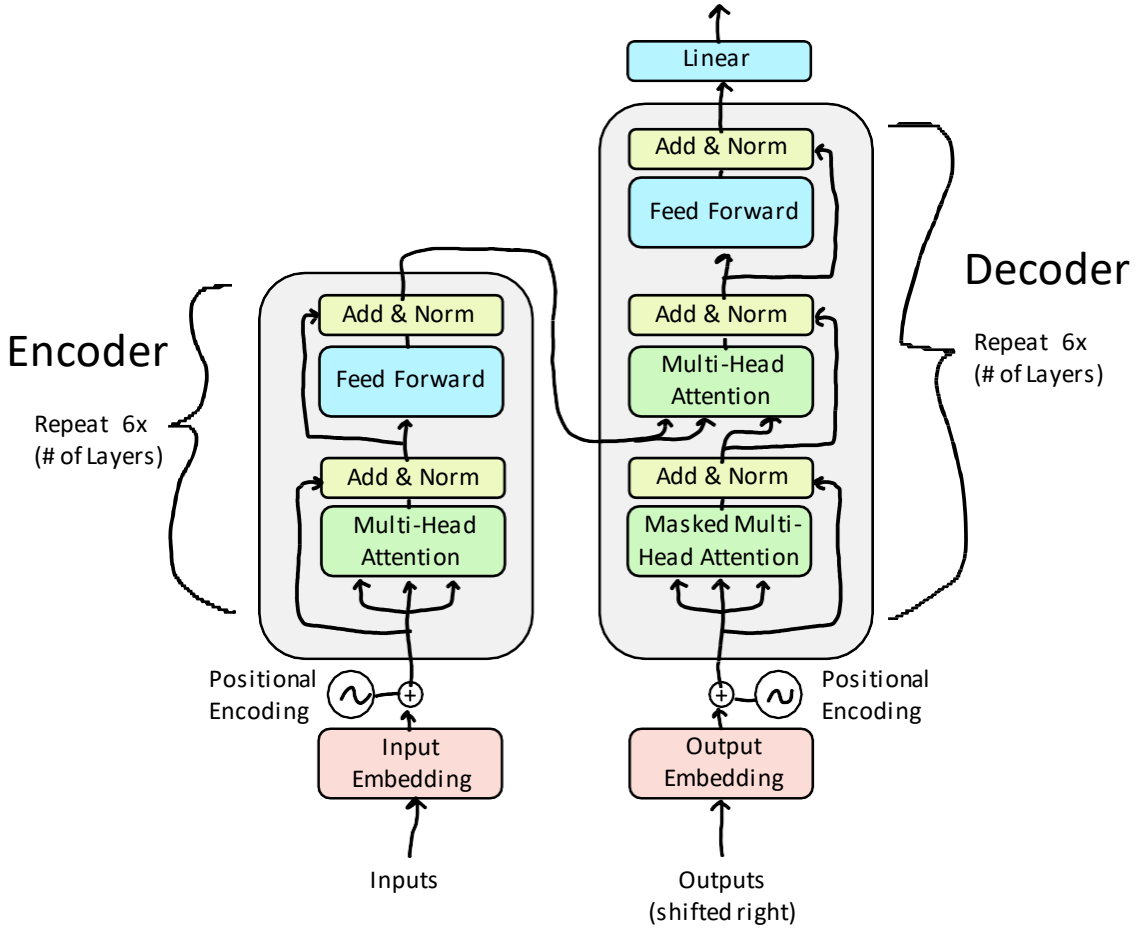
- Add a feed forward layer (with residual connections and layer norm)



Adapted from cs224n-2024-lecture-slides

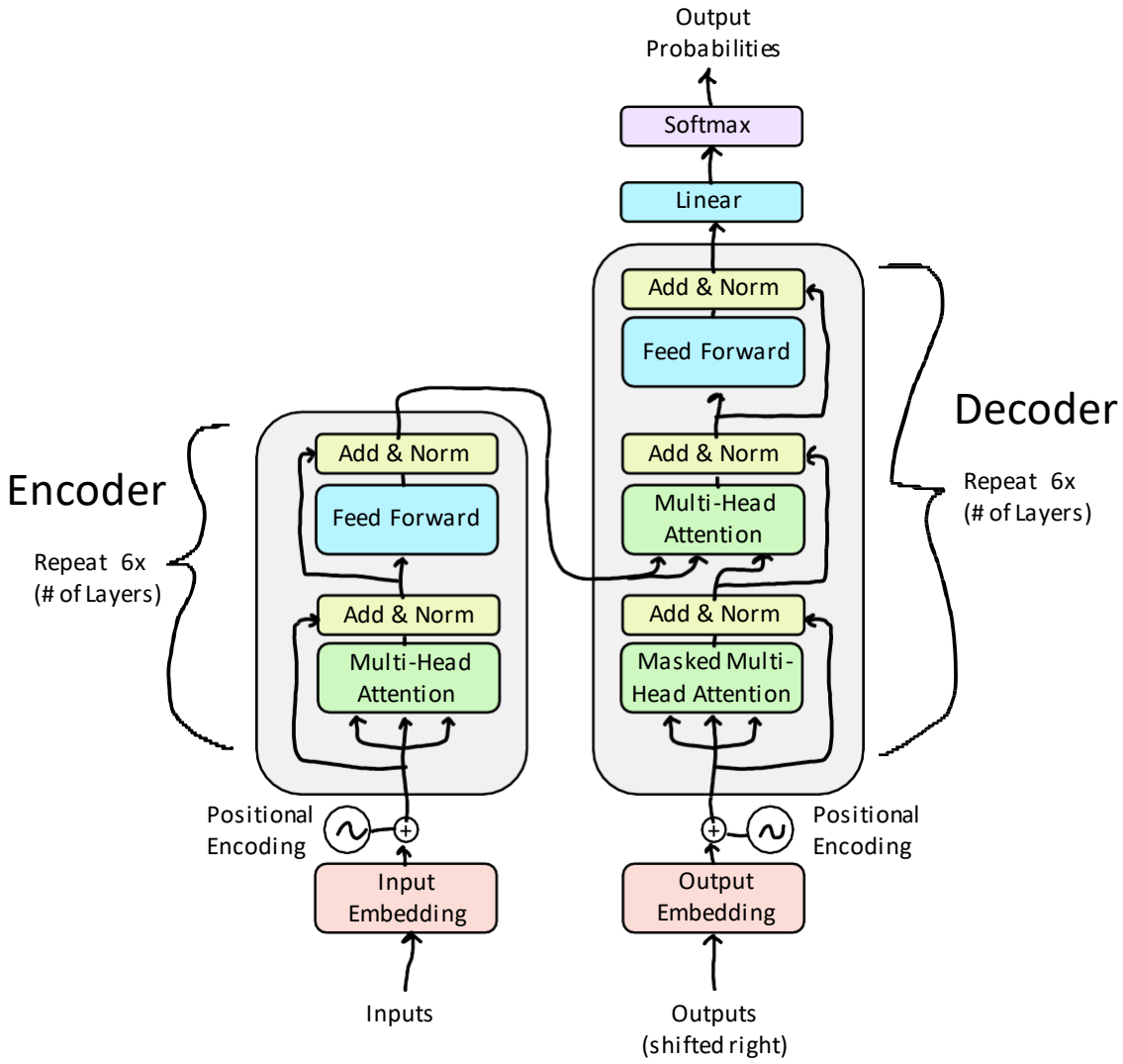
Decoder: Finishing touches!

- 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)

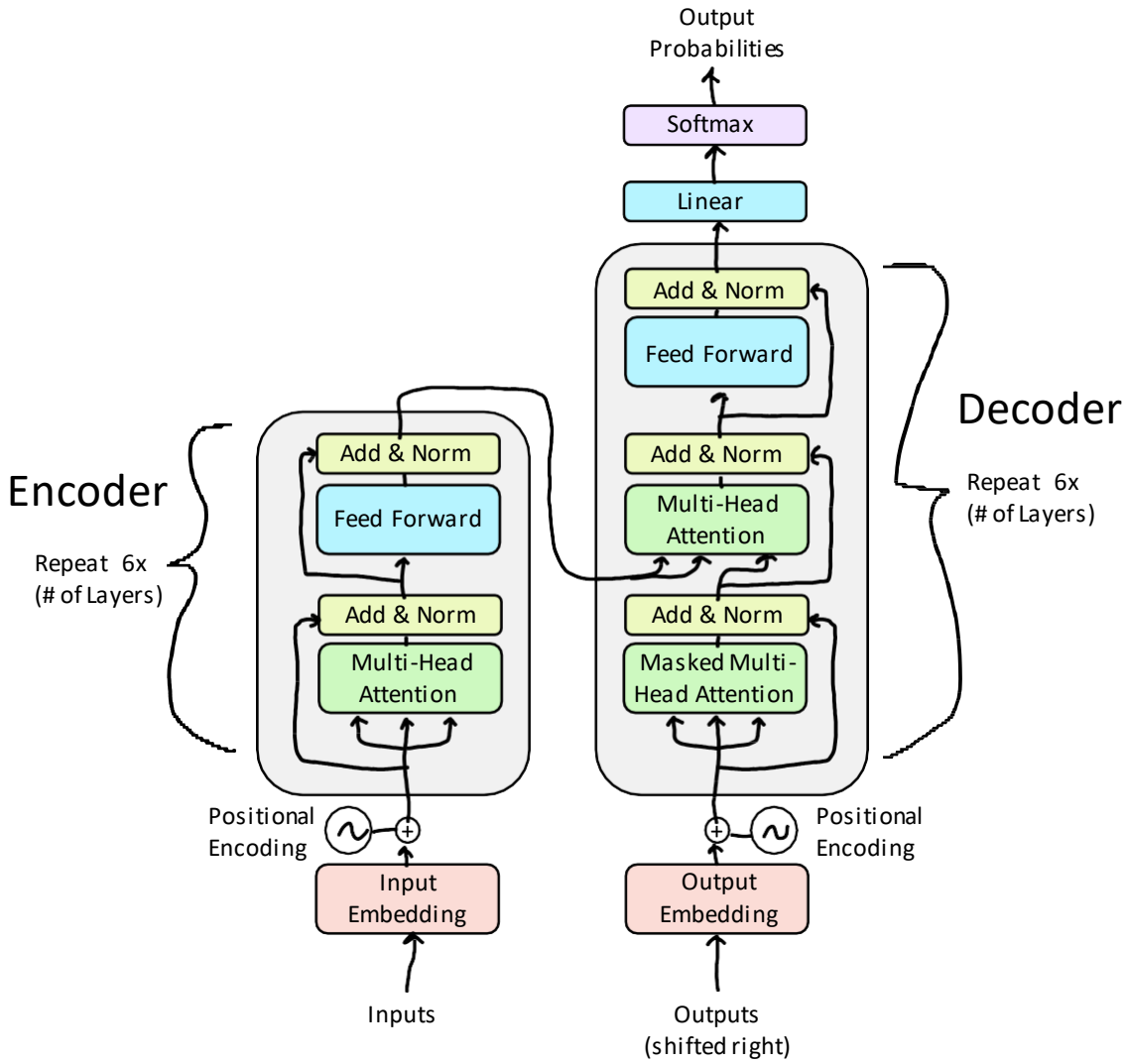


Decoder: Finishing touches!

- 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



Adapted from cs224n-2024-lecture-slides

What would we like to fix about the Transformer?

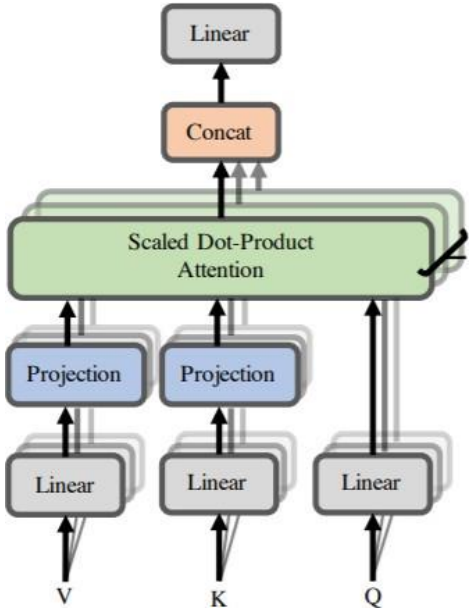
- **Quadratic compute in self-attention (today)**:
  - Computing all pairs of interactions means our computation grows **quadratically** with the sequence length!
  - For recurrent models, it only grew linearly!
- **Position representations**:
  - Are simple absolute indices the best we can do to represent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

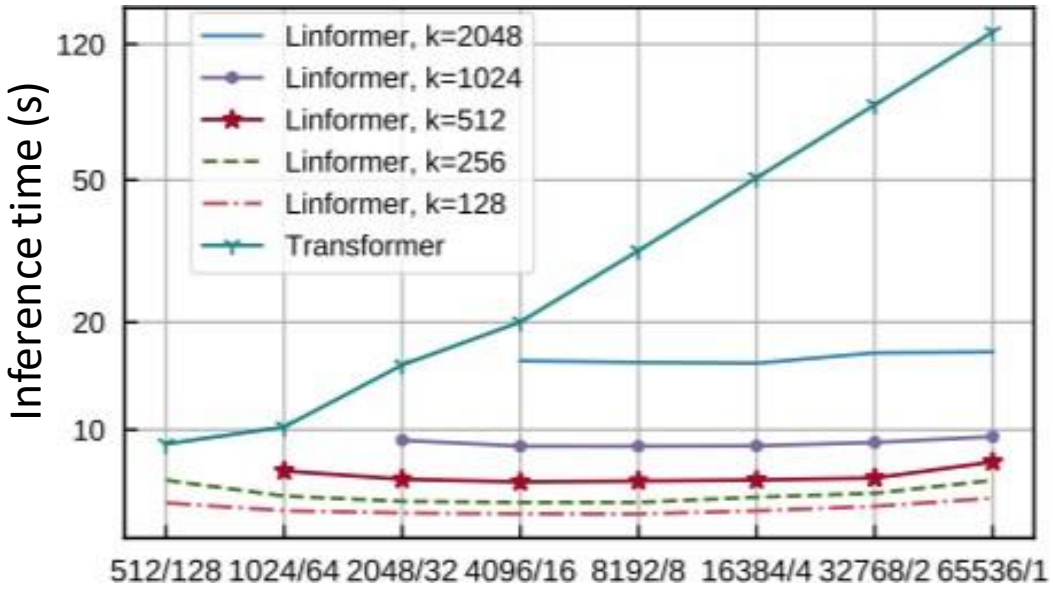
Adapted from cs224n-2024-lecture-slides

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



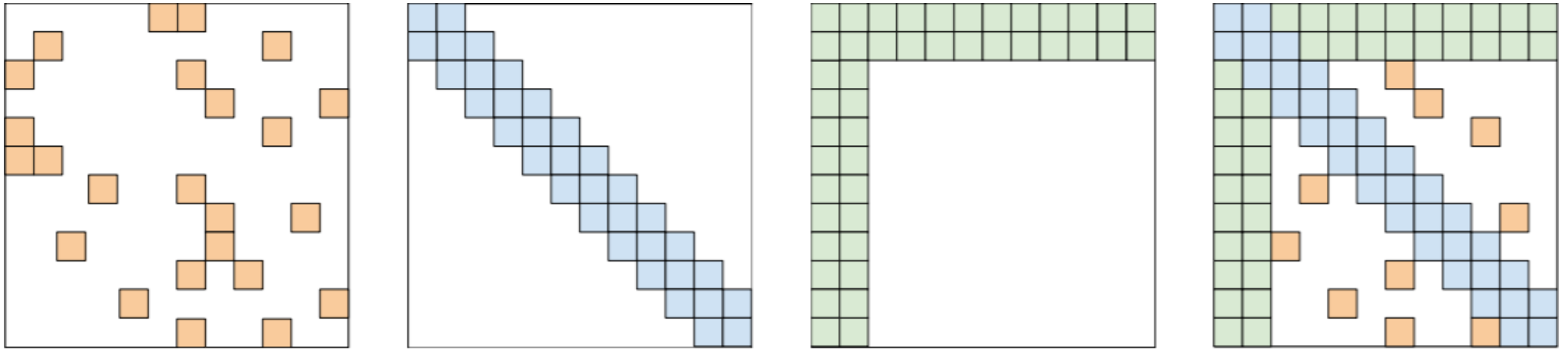


Adapted from cs224n-2024-lecture-slides

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

Adapted from cs224n-2024-lecture-slides

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**	24.34	**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**
Rezero	223M	11.1T	3.51	2.262 ± 0.003	1.939	61.69	15.64	20.90	26.37
Rezero + LayerNorm	223M	11.1T	3.26	2.223 ± 0.006	1.858	70.42	17.58	23.02	26.29
Rezero + RMS Norm	223M	11.1T	3.34	2.221 ± 0.009	1.875	70.33	17.32	23.02	26.19
Fixup	223M	11.1T	2.95	2.382 ± 0.012	2.067	58.56	14.42	23.02	26.31
24 layers, $d_{ff} = 1536$, $H = 6$	224M	11.1T	3.33	2.200 ± 0.007	1.843	**74.89**	17.75	**25.13**	**26.89**
18 layers, $d_{ff} = 2048$, $H = 8$	223M	11.1T	3.38	2.185 ± 0.005	**1.831**	**76.45**	16.83	**24.34**	**27.10**
8 layers, $d_{ff} = 4608$, $H = 18$	223M	11.1T	3.69	2.190 ± 0.005	1.847	**74.58**	17.69	**23.28**	**26.85**
6 layers, $d_{ff} = 6144$, $H = 24$	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**
Untied 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
Evolved 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	2.148 ± 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**	**26.19**	**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†

Adapted from cs224n-2024-lecture-slides

Summary

- Yay, you now understand Transformers!
- Good luck on Class activity!
- Remember to work on your project presentatic

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/

- 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