



# Transformer Architecture

CSCI 601-771 (NLP: Self-Supervised Models)

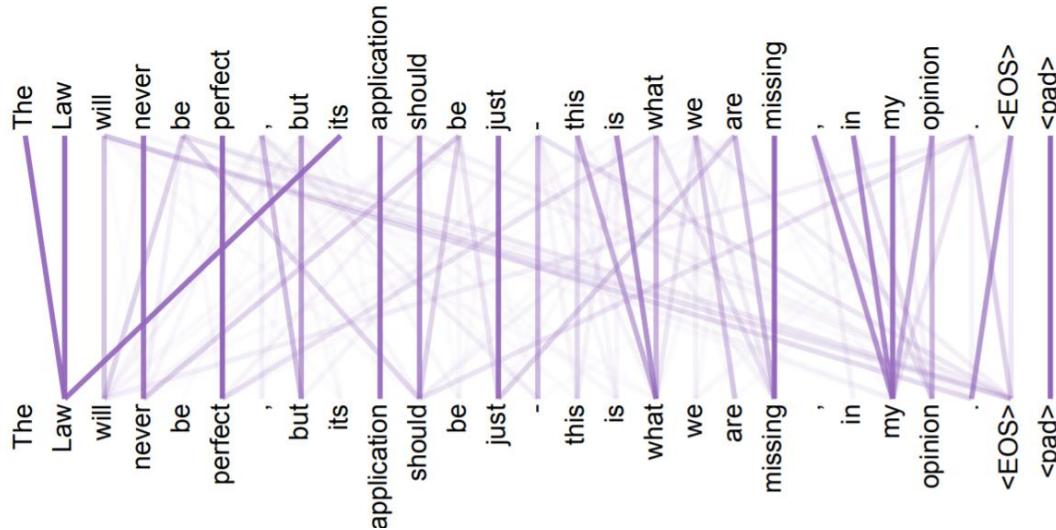
<https://self-supervised.cs.jhu.edu/fa2025/>

# Self-Attention Module

# Attention

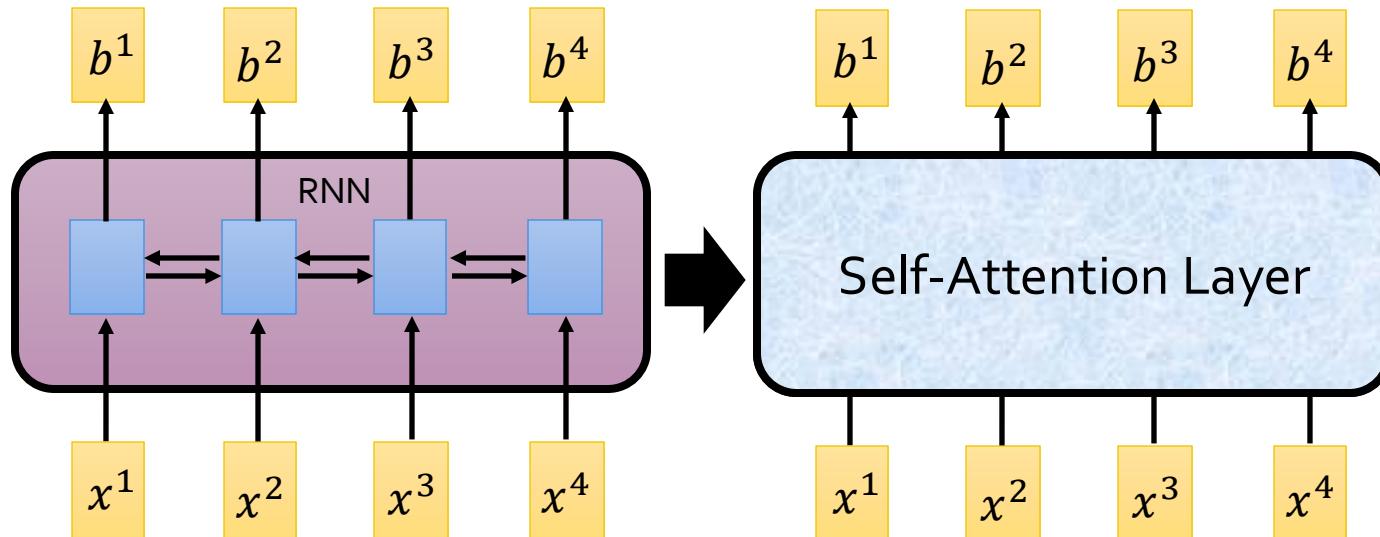


- Core idea: build a mechanism to **focus ("attend")** on a particular part of the context.
- How can this overcome the "long-range dependencies" problem in RNNs? By allowing the model to **directly "look"** at all tokens and decide which one is useful.



# Self-Attention

- $b^i$  is obtained based on the whole input sequence.
- can be parallelly computed.



Idea: replace any thing done by RNN with **self-attention**.

# Defining Self-Attention

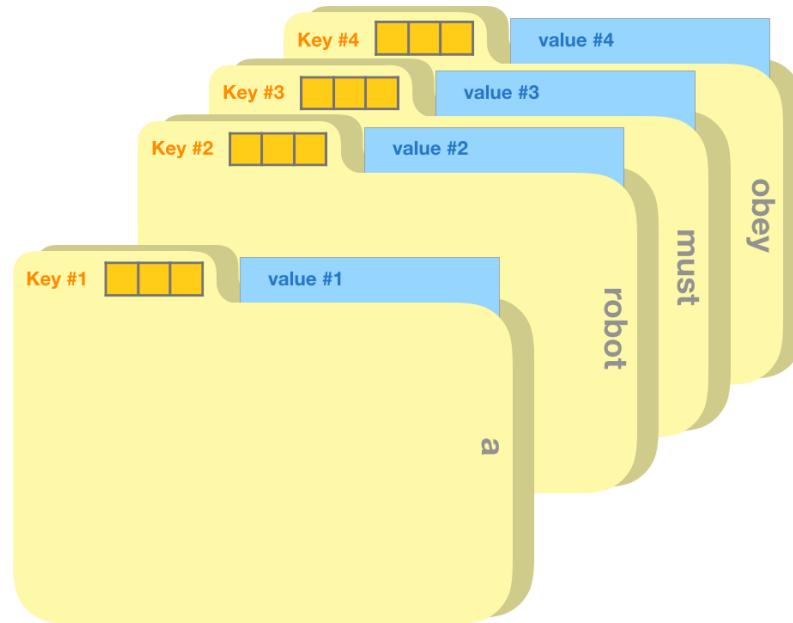
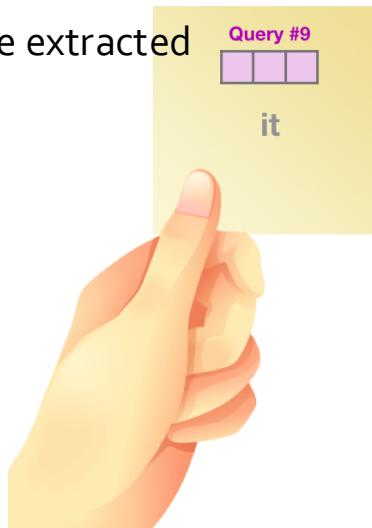
---

- Terminology:
  - **Query**: to match others
  - **Key**: to be matched
  - **Value**: information to be extracted

# Defining Self-Attention

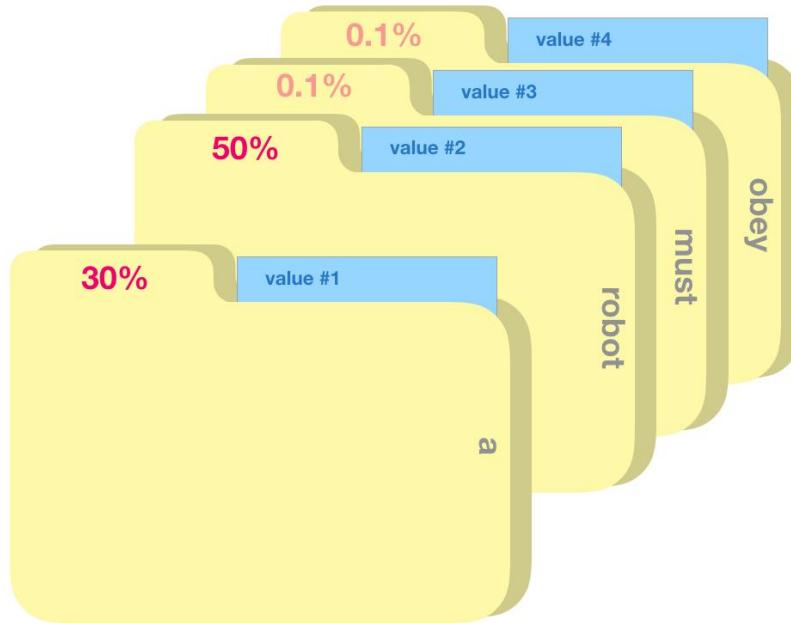
An analogy ....

- Terminology:
  - **Query**: to match others
  - **Key**: to be matched
  - **Value**: information to be extracted



# Defining Self-Attention

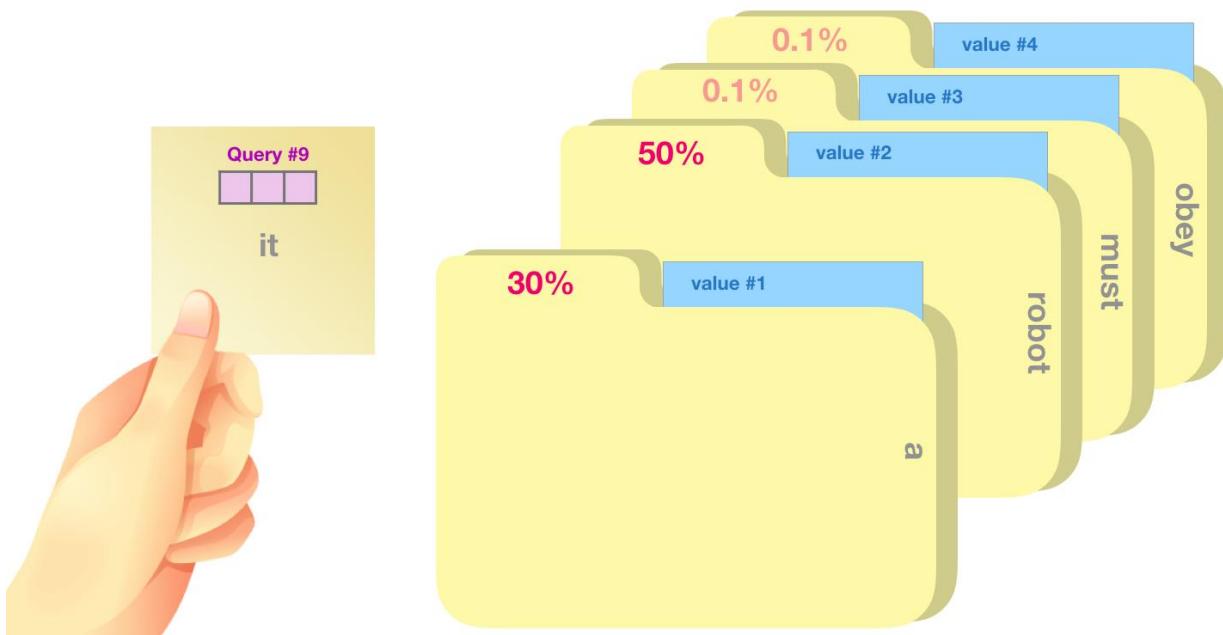
- Terminology:
  - **Query**: to match others
  - **Key**: to be matched
  - **Value**: information to be extracted



$q$ : query (to match others)  
 $q_i = x_i W^q$

$k$ : key (to be matched)  
 $k_i = x_i W^k$

$v$ : value (information to be extracted)  
 $v_i = x_i W^v$



$q$ : query (to match others)

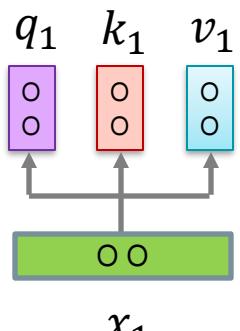
$$q_i = x_i W^q$$

$k$ : key (to be matched)

$$k_i = x_i W^k$$

$v$ : value (information to be extracted)

$$v_i = x_i W^v$$



$x_1$

The

$q$ : query (to match others)

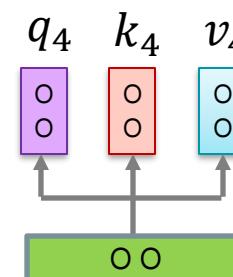
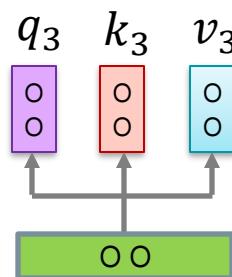
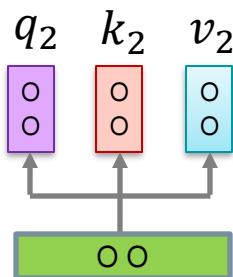
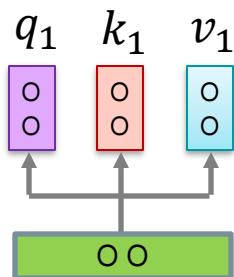
$$q_i = x_i W^q$$

$k$ : key (to be matched)

$$k_i = x_i W^k$$

$v$ : value (information to be extracted)

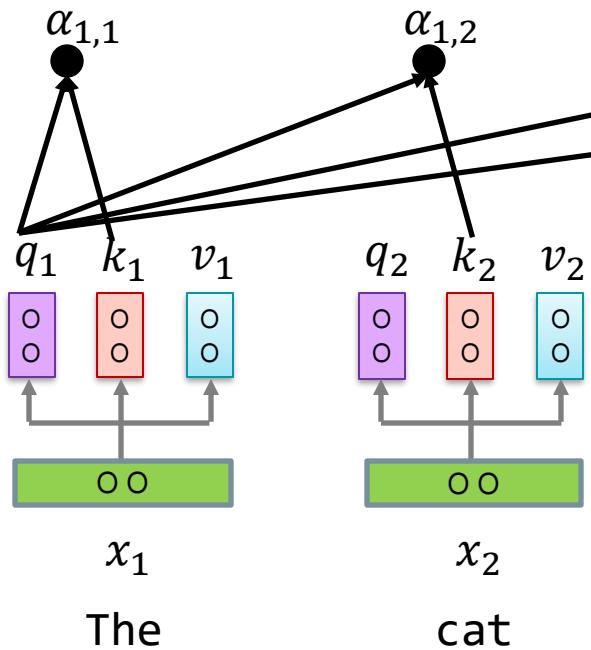
$$v_i = x_i W^v$$



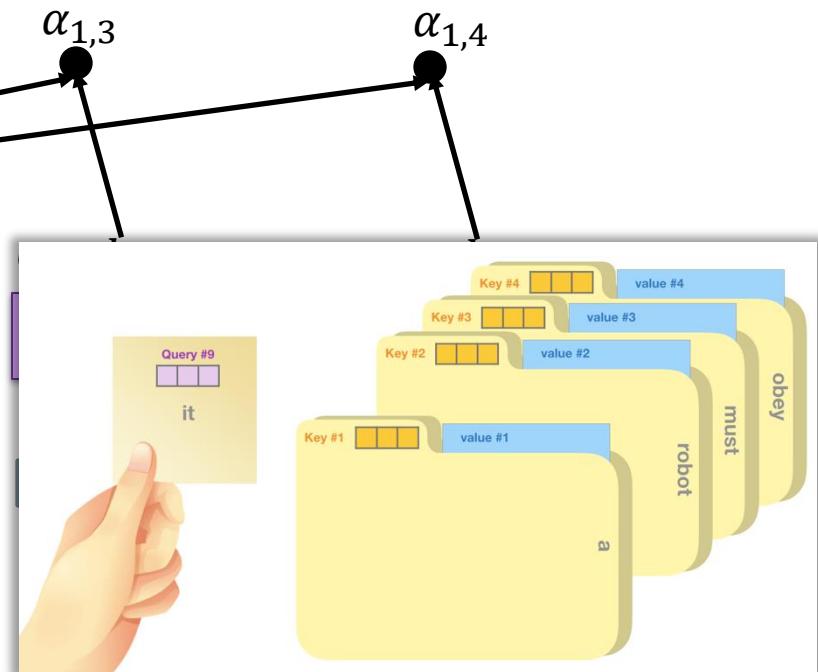
$$\alpha_{1,i} = \frac{q^1 \cdot k^i}{\sqrt{d}}$$

Scaled dot product

How much  
should "The"  
attend to other  
positions?

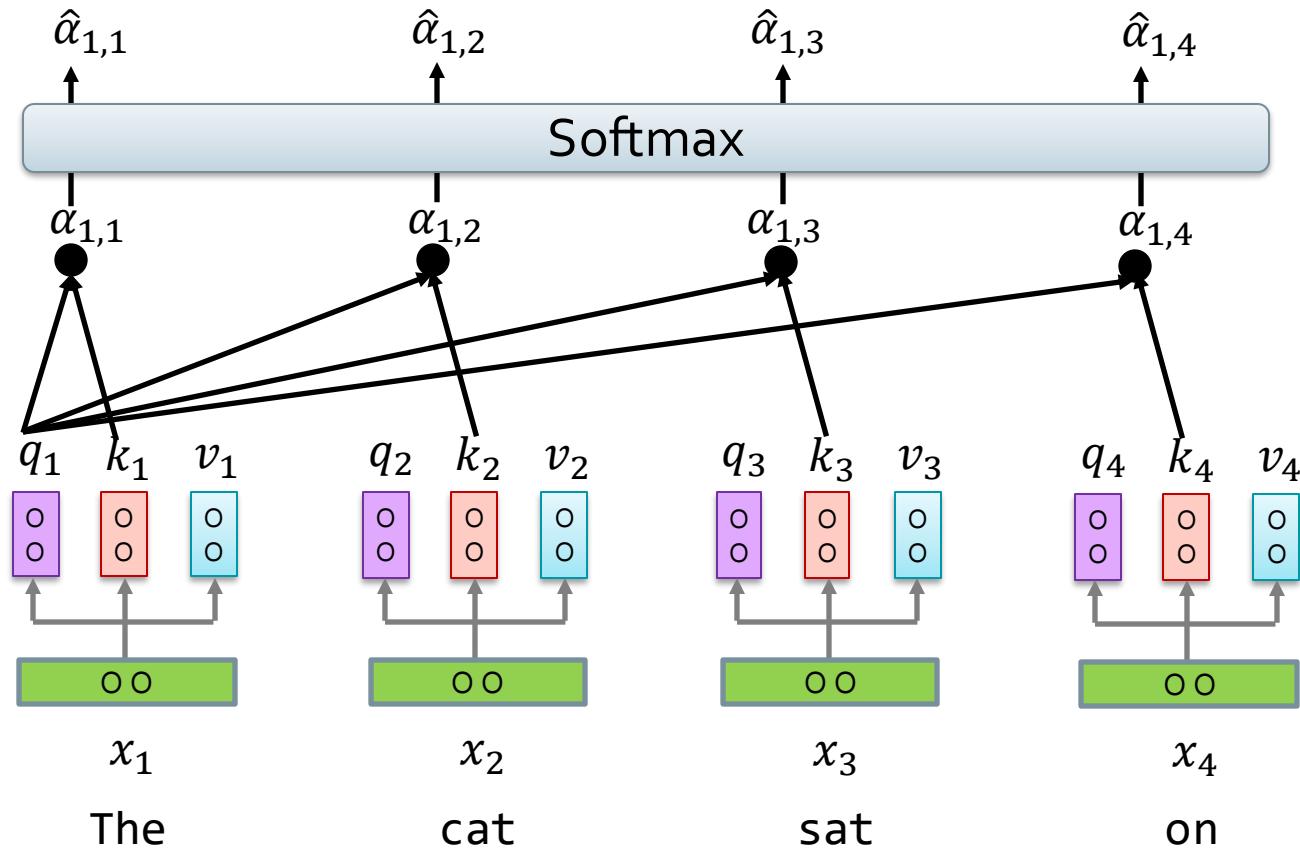


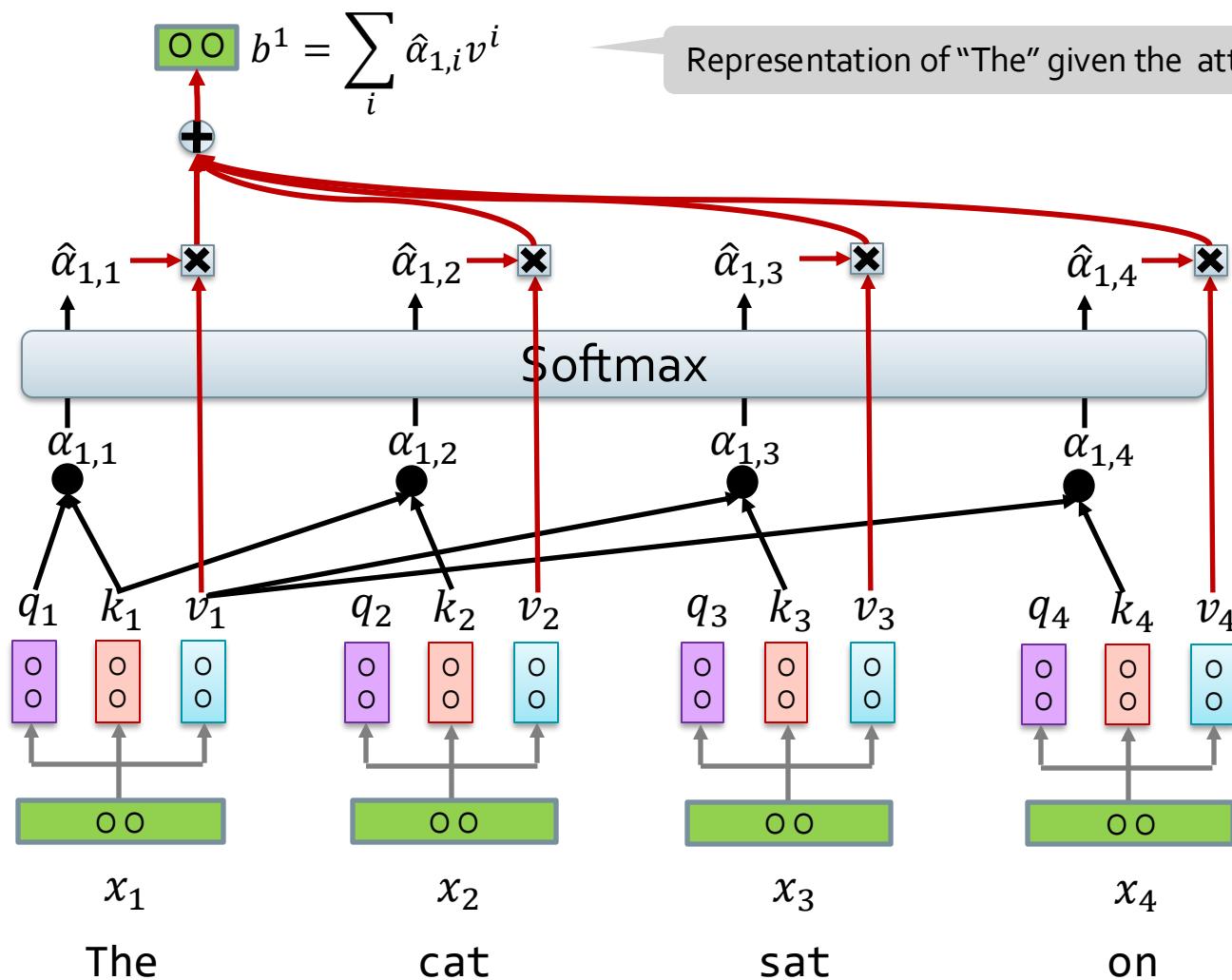
*q*: query (to match others)  
*k*: key (to be matched)  
*v*: value (information to be extracted)



$$\sigma(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

How much  
should "The"  
attend to other  
positions?





# Self-Attention: Matrix Notation

- Input sequence:  $\mathbf{x} = [x_1, x_2, \dots, x_n] \in \mathbb{R}^{n \times d}$
- Calculate:

$$\mathbf{Q} = \mathbf{x} \mathbf{W}^q \in \mathbb{R}^{n \times d}$$

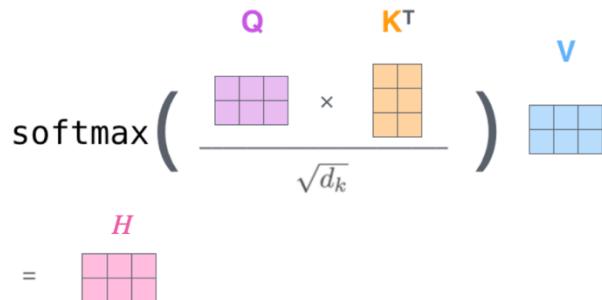
$$\mathbf{K} = \mathbf{x} \mathbf{W}^k \in \mathbb{R}^{n \times d}$$

$$\mathbf{V} = \mathbf{x} \mathbf{W}^v \in \mathbb{R}^{n \times d}$$

- Pairwise similarity matrix between queries and keys:  $\mathbf{Q} \mathbf{K}^T \in \mathbb{R}^{n \times n}$
- Attention output:

$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{\mathbf{Q} \mathbf{K}^T}{\sqrt{d}}\right) \mathbf{V} \in \mathbb{R}^{n \times d}$$

Attention raw scores												
0	-0.08	1.24	0.69	-0.98	1.43	-0.6	0.7	0.16	0.93	1.28	-1.61	-1.1
1	-0.09	-0.0	-0.7	0.06	0.25	0.23	0.26	0.18	0.78	-0.21	-1.01	1.01
2	0.86	1.19	1.59	0.86	-0.13	-0.15	-2.13	-0.98	-0.87	-1.72	1.87	-0.72
3	0.12	-0.03	-0.02	0.88	-0.46	-0.7	0.54	-0.42	-1.89	-0.38	0.04	-0.84
4	0.51	0.17	0.13	-1.64	0.24	-0.02	1.68	-0.36	0.64	0.36	0.27	0.66
5	0.24	-1.44	0.43	0.74	0.96	-1.21	-0.31	1.54	1.66	1.14	0.58	-1.44
6	0.26	-0.1	0.93	0.72	-0.38	1.65	0.47	-0.96	-0.17	-0.9	-1.57	0.22
7	-0.55	0.81	0.71	1.7	-0.8	-1.14	-0.32	1.78	-0.7	-0.04	1.54	0.81
8	0.74	-0.76	-0.44	-0.08	-1.38	-0.13	1.25	-1.37	1.84	0.3	0.57	0.74
9	-0.97	-0.91	0.15	0.35	-0.81	0.11	1.14	-1.52	1.06	1.87	0.5	-0.3
10	1.56	0.9	0.39	1.46	1.44	-1.05	0.9	-0.73	0.36	-0.67	-0.62	-0.43
11	0.32	0.74	0.44	-0.1	1.19	0.83	0.29	2.06	0.51	-0.26	1.51	0.11



# Q: Why do we need $\sqrt{d}$ ?

$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}}\right)\mathbf{V}$$

# Q: Does this also handle batching?

$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}}\right)\mathbf{V}$$

# Self-Attention: Thinking about batching

- Suppose our data comes in batches of size  $b$ , i.e.,  $\mathbf{x} \in \mathbb{R}^{b \times n \times d}$
- Self-attention's matrix form extends to this batched data. Let's verify:
- Calculate the keys, queries and values, per-batch:

$$\mathbf{Q} = \mathbf{x} \mathbf{W}^q \in \mathbb{R}^{b \times n \times d}$$

$$\mathbf{K} = \mathbf{x} \mathbf{W}^k \in \mathbb{R}^{b \times n \times d}$$

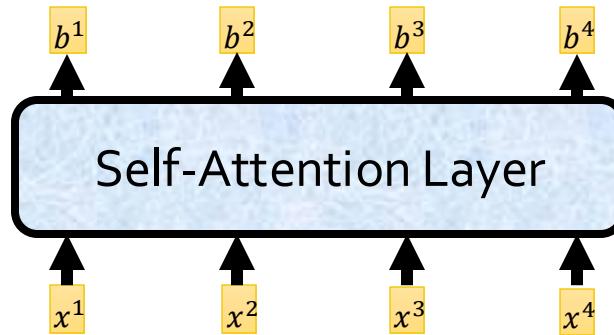
$$\mathbf{V} = \mathbf{x} \mathbf{W}^v \in \mathbb{R}^{b \times n \times d}$$

- Pairwise similarity matrix between queries and keys, per batch:  $\mathbf{Q} \mathbf{K}^T \in \mathbb{R}^{b \times n \times n}$
- Attention output for the whole batched data:

$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{\mathbf{Q} \mathbf{K}^T}{\sqrt{d}}\right) \mathbf{V} \in \mathbb{R}^{b \times n \times d}$$

# Recap: Self-Attention

- **Attention** is a powerful mechanism to create context-aware representations
- A way to focus on select parts of the input



- Better at maintaining **long-distance dependencies** in the context.

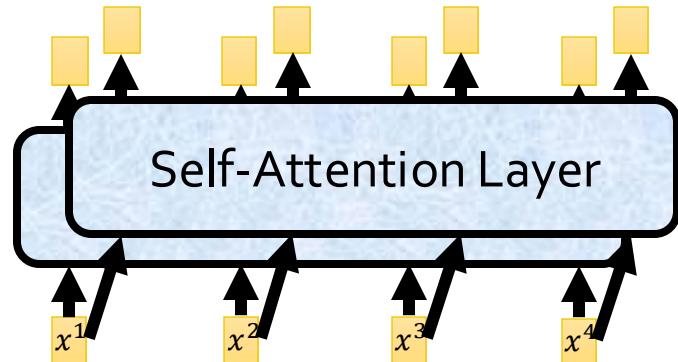
# Multi-Headed Self-Attention

- **Issue:** Each head has limited capacity, e.g., it cannot attend to too many things at the same time.



- **Main idea:** Allows model to jointly attend to information from different representation subspaces (like ensembling).

- **Multiple parallel attention layers.**
  - Each attention layer has its own parameters.
  - Concatenate the results and run them through a linear projection.

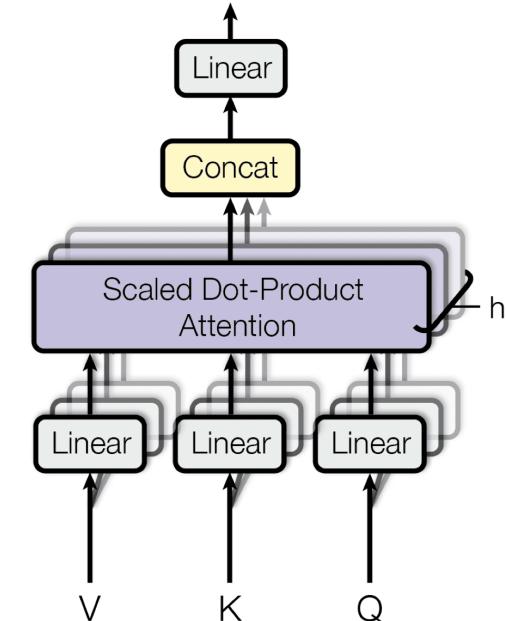


# Multi-Headed Self-Attention

- Just concatenate all the heads and apply an output projection.

$$\begin{aligned}\text{head}_i &= \text{Attention}(\mathbf{x}W_i^q, \mathbf{x}W_i^k, \mathbf{x}W_i^v) \\ \text{MultiHeadedAttention}(\mathbf{x}) &= \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^o\end{aligned}$$

- Previously, we used the following dimensions for **single**-head SA:  
 $W_i^q \in \mathbb{R}^{d \times d}, \quad W_i^k \in \mathbb{R}^{d \times d}, \quad W_i^v \in \mathbb{R}^{d \times d},$
- In practice, we use a reduced dimension for each head.  
 $W_i^q \in \mathbb{R}^{d \times \frac{d}{h}}, \quad W_i^k \in \mathbb{R}^{d \times \frac{d}{h}}, \quad W_i^v \in \mathbb{R}^{d \times \frac{d}{h}}, \quad W^o \in \mathbb{R}^{d \times d}$
- The total computational cost is similar to that of single-head attention with full dimensionality.



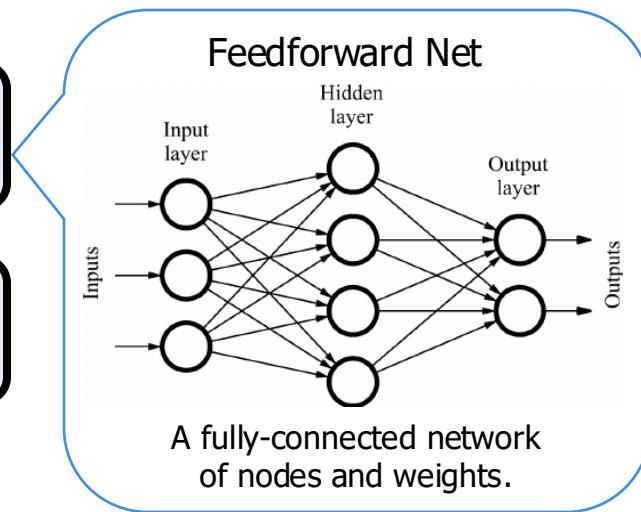
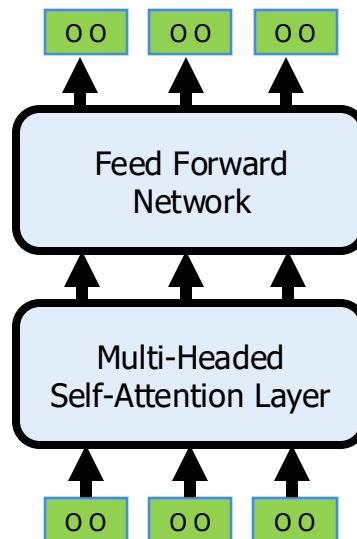
$h$ : number of heads  
 $d$ : feature dimension in output of SA

# Combine with FFN

- Add a **feed-forward network** to add more expressivity.
  - This applies another nonlinearity to the representations (or “post-process” them).

$$\text{FFN}(\mathbf{x}) = f(\mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$$
$$\mathbf{W}_1 \in \mathbb{R}^{d \times d_{\text{ff}}},$$
$$\mathbf{W}_2 \in \mathbb{R}^{d_{\text{ff}} \times d}$$

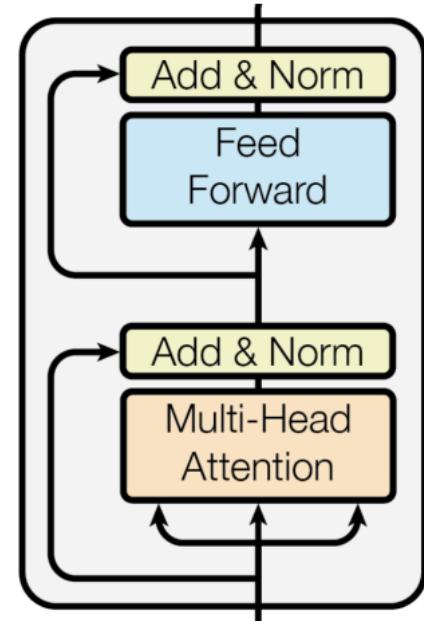
- Usually, the dimensionality of the hidden feedforward layer  $d_{\text{ff}}$  is 2-8 times larger than the input dimension  $d$ .



# How Do We Prevent Vanishing Gradients?

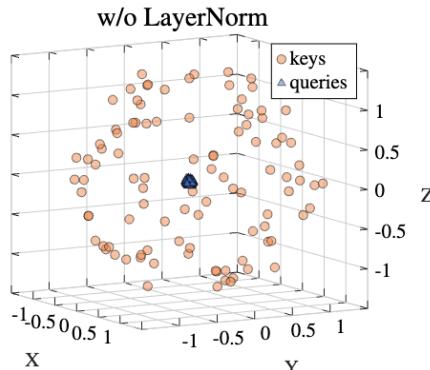
- **Residual connections** let the model “skip” layers
  - These connections are particularly useful for training deep networks
- Use **layer normalization** to stabilize the network and allow for proper gradient flow

$$y = \frac{x - \text{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} * \gamma + \beta$$

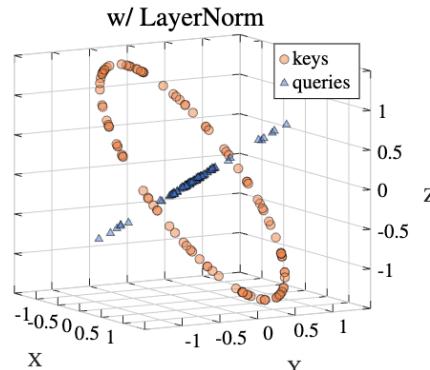


# LayerNorm: A Geometric Interpretation

- LayerNorm is also for expressivity, preventing the “unselectable” keys problem.
- With LayerNorm, we see the following pattern: (Demeter et al., 2020; Grivas et al., 2022)
  - **Queries** are close to **1** vector  $[1, 1, \dots, 1]$ , thus attending to all keys equally, if needed.
  - **Keys** are on a hyper-plane that is orthogonal to the **1** vector, scaled to the same  $\sqrt{d}$  norm.



(a) Without LayerNorm, the model has learned key and query vectors without any apparent geometric structure.



(b) LayerNorm projects the key vectors onto the same hyperplane so that the model can learn to align the queries to be orthogonal to the keys.

# Putting it Together: Self-Attention Block

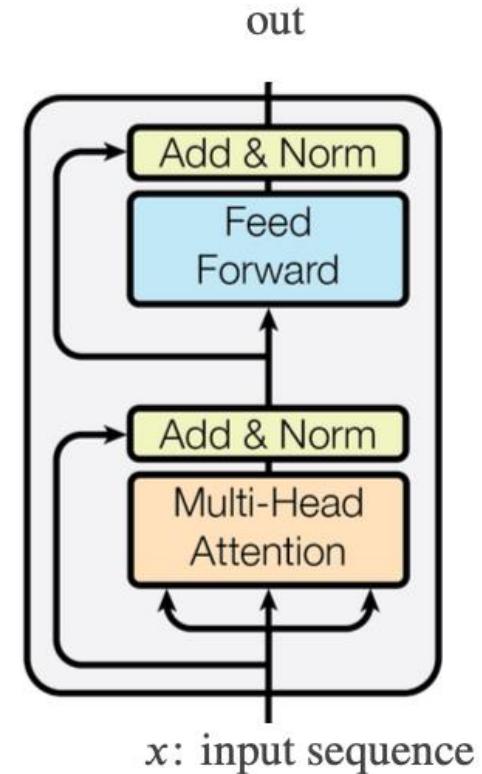
Given input  $\mathbf{x}$ :

$$\tilde{\mathbf{x}} = \text{MultiHeadedAttention}(\mathbf{x}; \mathbf{W}^q, \mathbf{W}^k, \mathbf{W}^v)$$

$$\mathbf{x} = \text{LayerNorm}(\tilde{\mathbf{x}} + \mathbf{x})$$

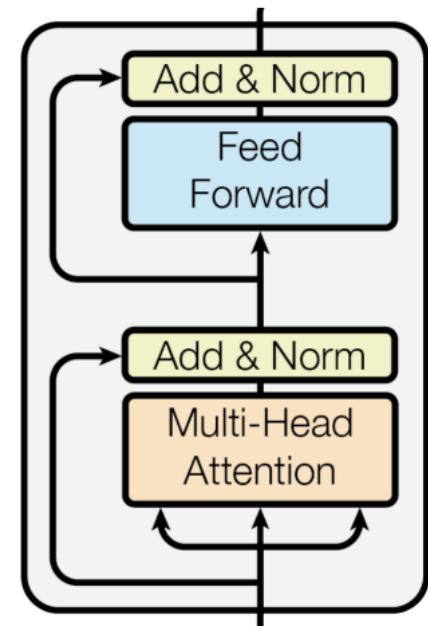
$$\tilde{\mathbf{x}} = \text{FFN}(\mathbf{x}) = f(\mathbf{x}W_1 + b_1)W_2 + b_2$$

$$\text{out} = \text{LayerNorm}(\tilde{\mathbf{x}} + \mathbf{x})$$



# Summary: Self-Attention Block

- **Self-Attention:** A critical building block of modern language models.
  - The idea is to compose meanings of words weighted according some similarity notion.
- **Next:** We will combine self-attention blocks to build various architectures known as Transformer.

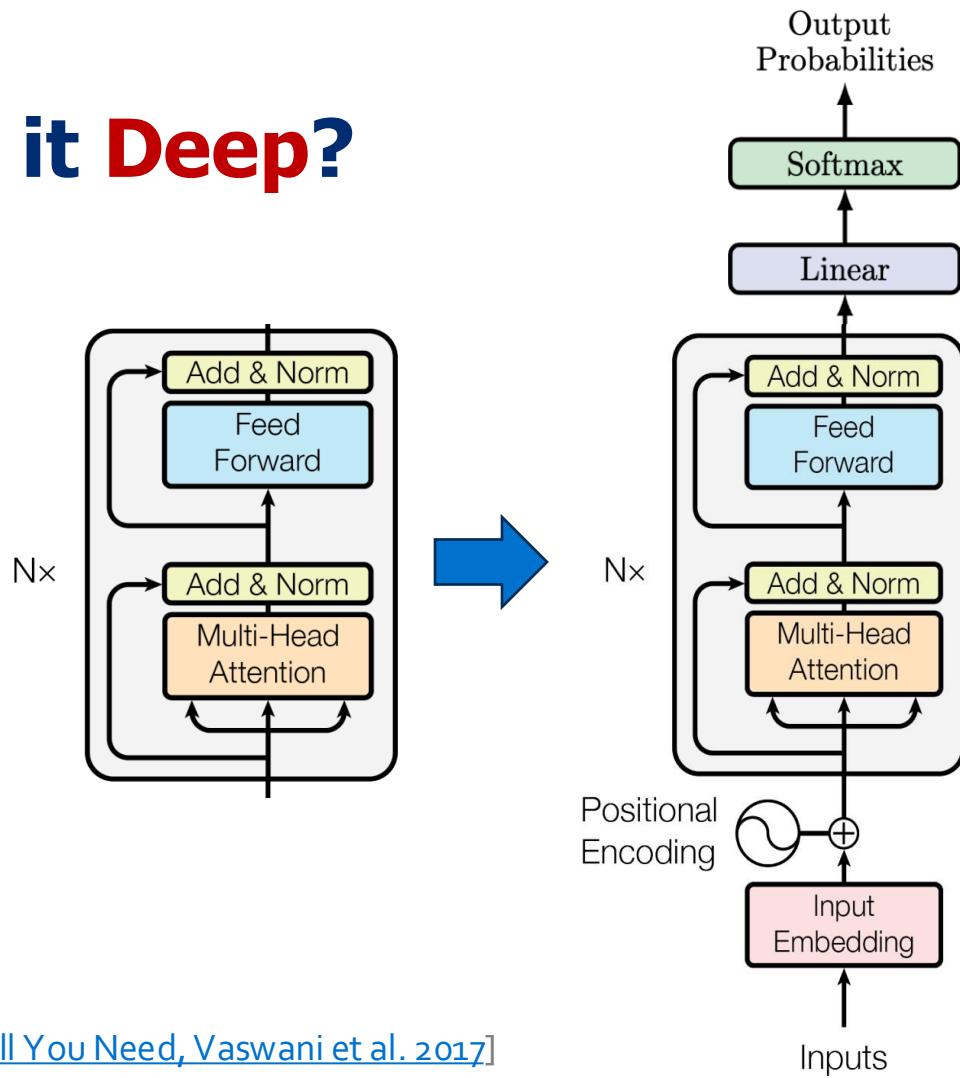
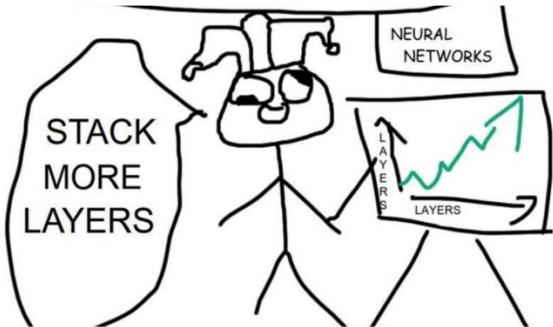




# A Decoder-Only Transformer

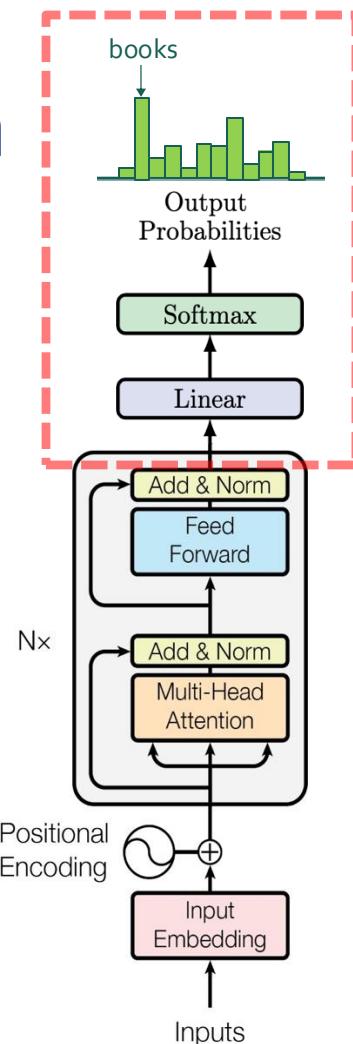
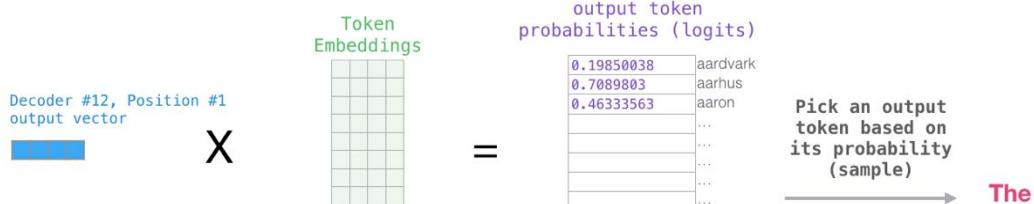
# How Do We Make it Deep?

- Stack more layers!

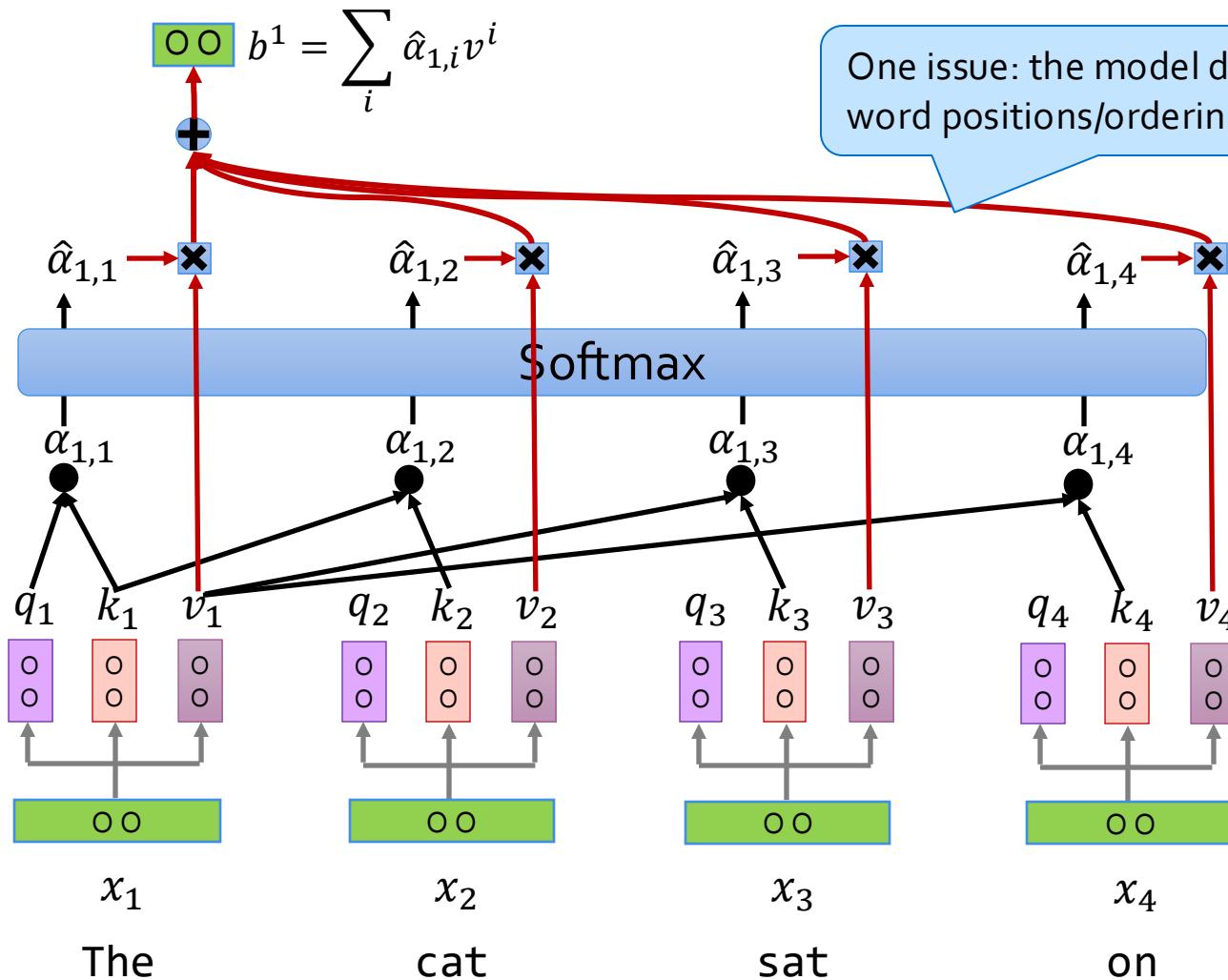


# From Representations to Prediction

- To perform prediction, add a classification head on top of the final layer of the transformer:  $\mathbf{x} \in \mathbb{R}^{n \times d}$ .
  - Where  $n$  is the length of your sequence.
- To obtain logits, we can apply a linear transformation with token embedding matrix  $\mathbf{W}_V \in \mathbb{R}^{V \times d}$ :  
$$\mathbf{x}\mathbf{W}_V^T \in \mathbb{R}^{n \times V}$$
- To obtain probabilities, run this through softmax:  
$$\text{softmax}(\mathbf{x}\mathbf{W}_V^T) \in \mathbb{R}^{n \times V}$$



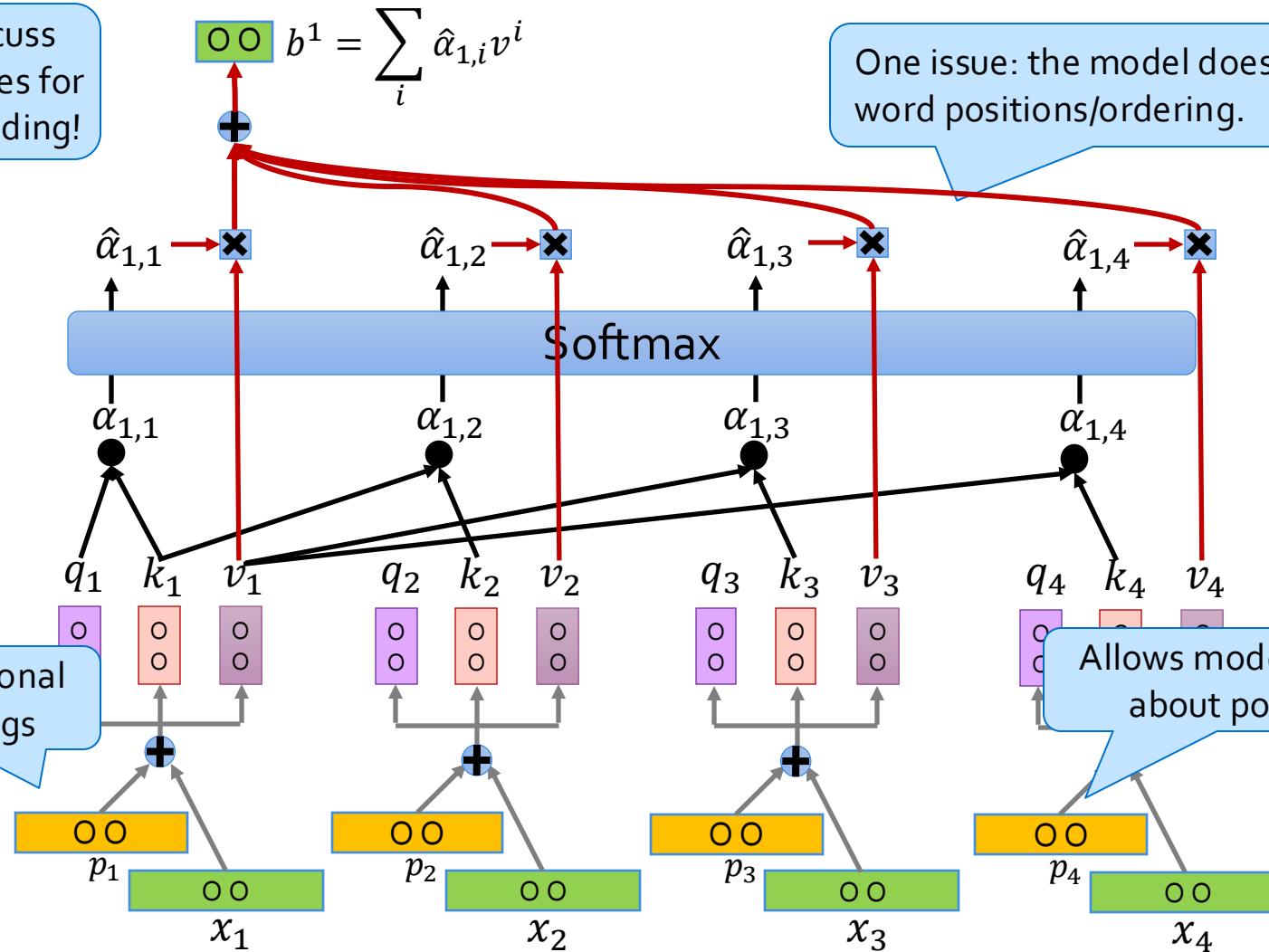
One wrinkle: How does Transformer  
know about word ordering? ...



We will discuss various choices for these embedding!

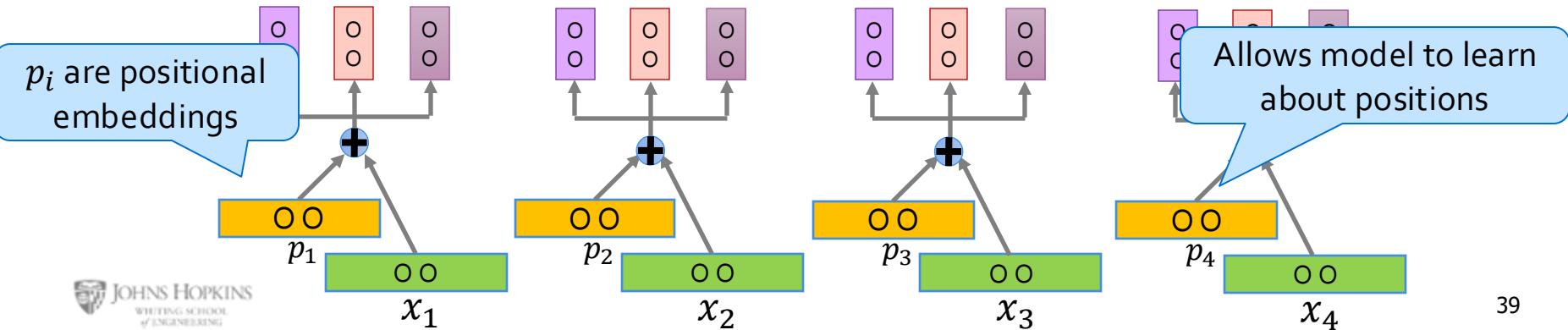
$$b^1 = \sum_i \hat{\alpha}_{1,i} v^i$$

One issue: the model doesn't know word positions/ordering.



# Absolute Positional Embeddings

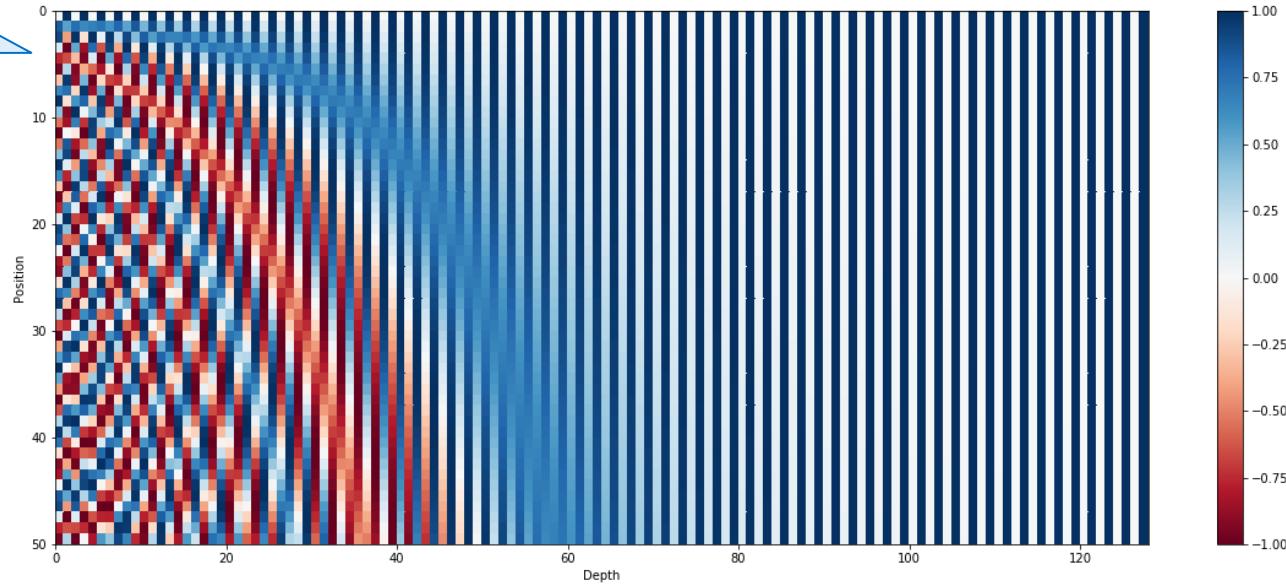
- Why “add”? Why not, say, “concatenate and then project”?
    - “concatenate and then project” would be a more general approach with more trainable parameters.
    - In practice, “sum” works fine that
    - The intuition here is that “summing” forms point clouds of word embedding information around position embeddings unique to each position.



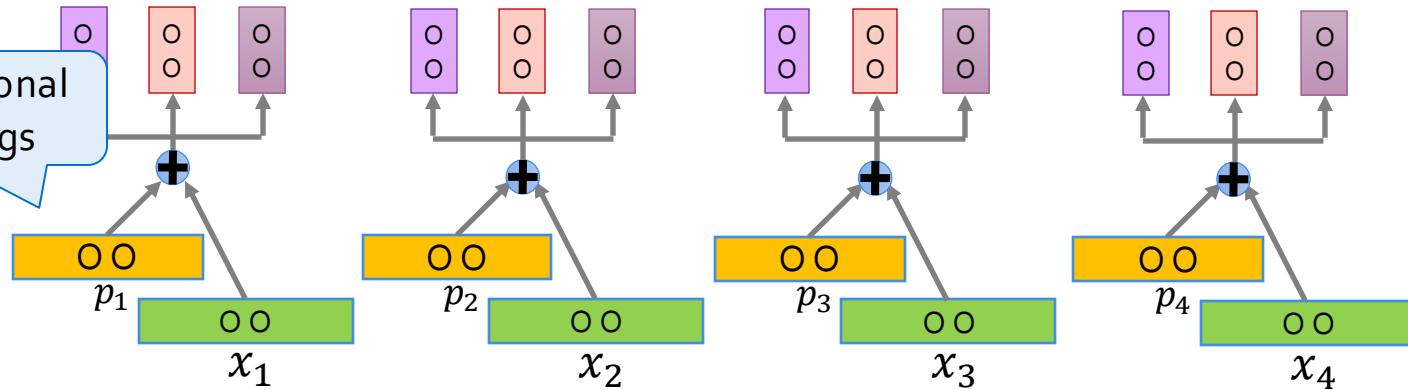
Notice, any pair of rows  
are different!

## Sine/Cosine encoding

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



$p_i$  are positional embeddings



# Recap—positional encoding

---

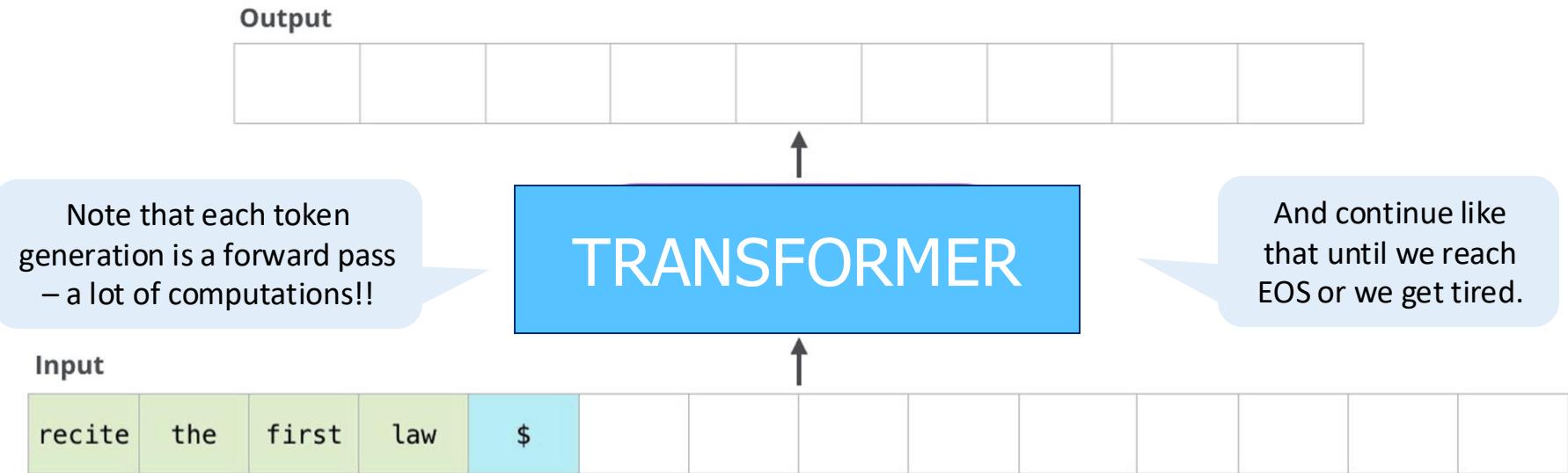
- They inform Transformer about word orderings.
- The original proposal: fixed sine/cosine embeddings added to word embeddings.
  - While this is a reasonable start, later we will see more sophisticated alternatives.
- Next: let's think about running transformer on language data.



# How does inference work?

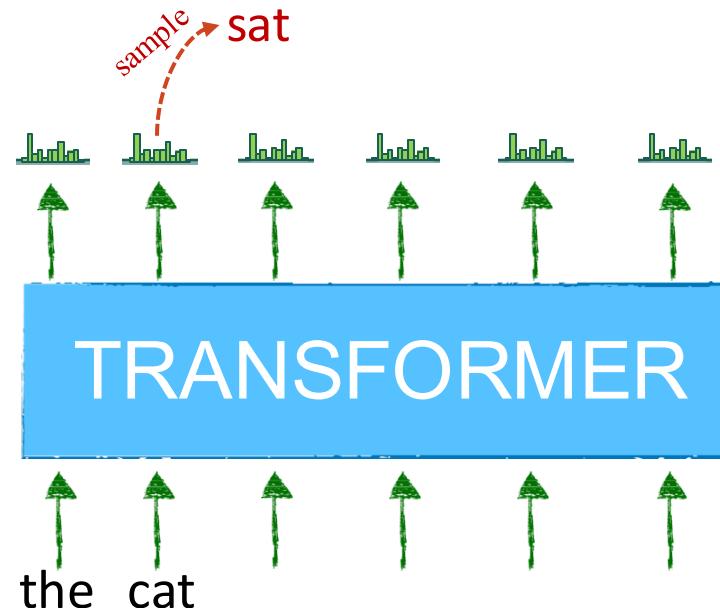


# Generating text via Transformer



# Generating text via Transformer

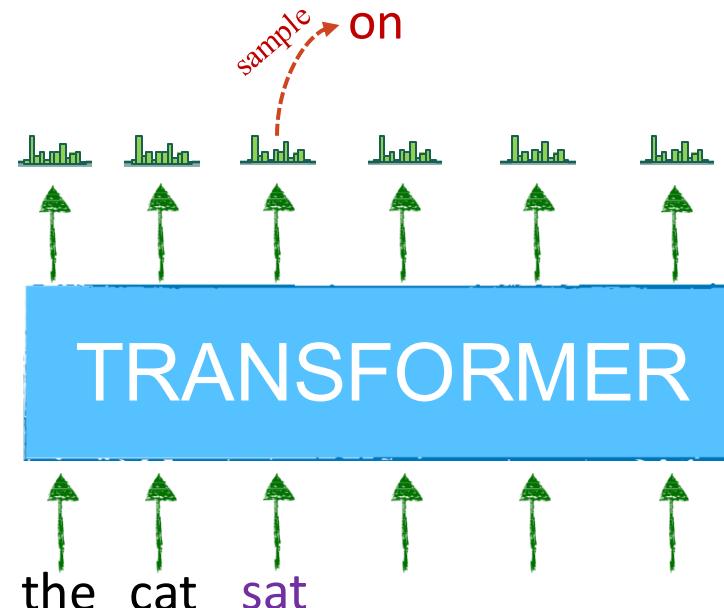
- Use the output of previous step as input to the next step repeatedly



# Generating text via Transformer

- Use the output of previous step as input to the next step repeatedly

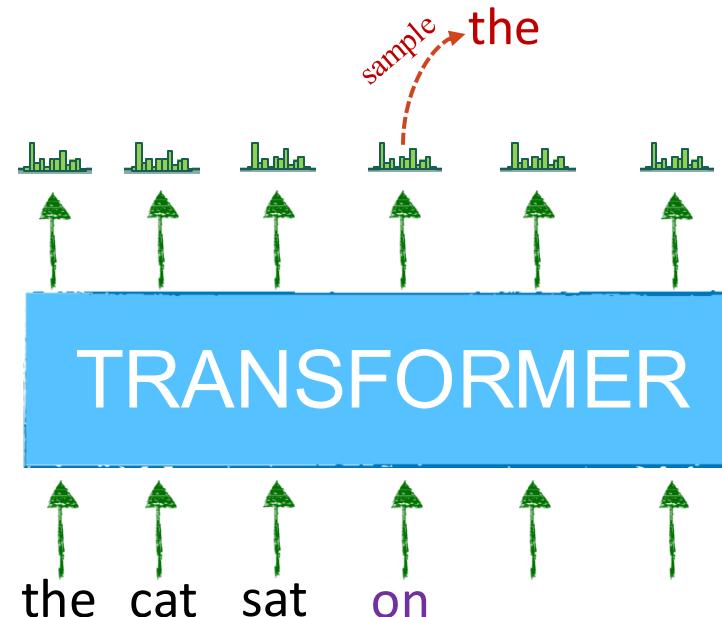
The probabilities get revised upon adding a new token to the input.



# Generating text via Transformer

- Use the output of previous step as input to the next step repeatedly

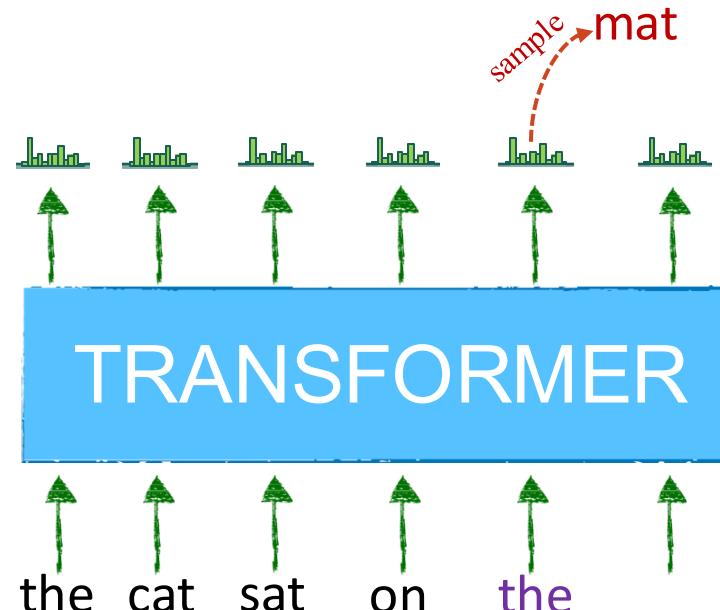
The probabilities get revised upon adding a new token to the input.



# Generating text via Transformer

- Use the output of previous step as input to the next step repeatedly

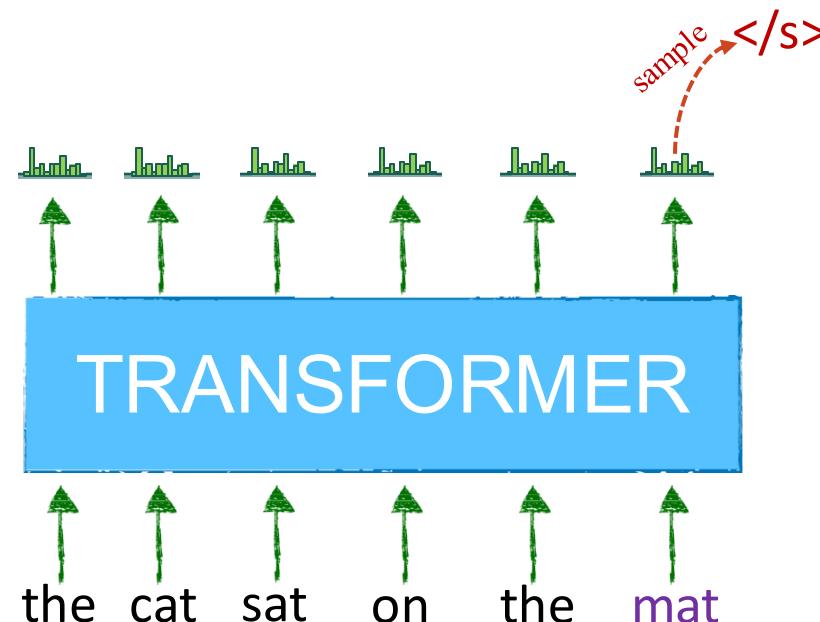
The probabilities get revised upon adding a new token to the input.



# Generating text via Transformer

- Use the output of previous step as input to the next step repeatedly

The probabilities get revised upon adding a new token to the input.





# How does training work?



# Quiz

- What do we optimize when we train a transformer?
  1.  $X$  and  $q$
  2.  $W_k$  and  $W_v$
  3. Softmax function
  4. Sine/cos positional encodings
  5. The embedding matrix
  6.  $W_1$  and  $W_2$  in your FFN
  7.  $W_q$

**Answer:** 2, 6, 7 and 5.

# Training a Transformer Language Model

- **Goal:** Train a Transformer for language modeling (i.e., predicting the next word).
- **Approach:** Train it so that each position is predictor of the next (right) token.
  - We just shift the input to right by one, and use as labels

(gold output)  $Y = \text{cat sat on the mat } </s>$

EOS special token



```
X = text[:, :-1]  
Y = text[:, 1:]
```

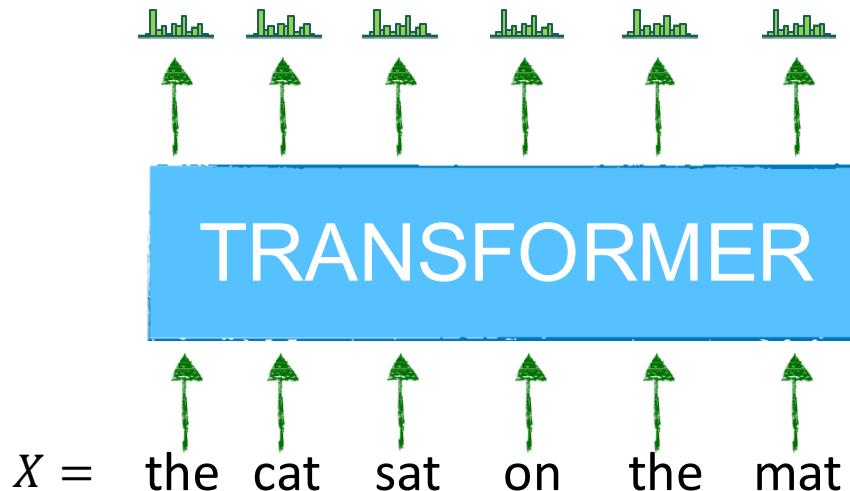
$X = \text{the cat sat on the mat}$

[Slide credit: Arman Cohan]

# Training a Transformer Language Model

- For each position, compute their corresponding **distribution** over the whole vocab.

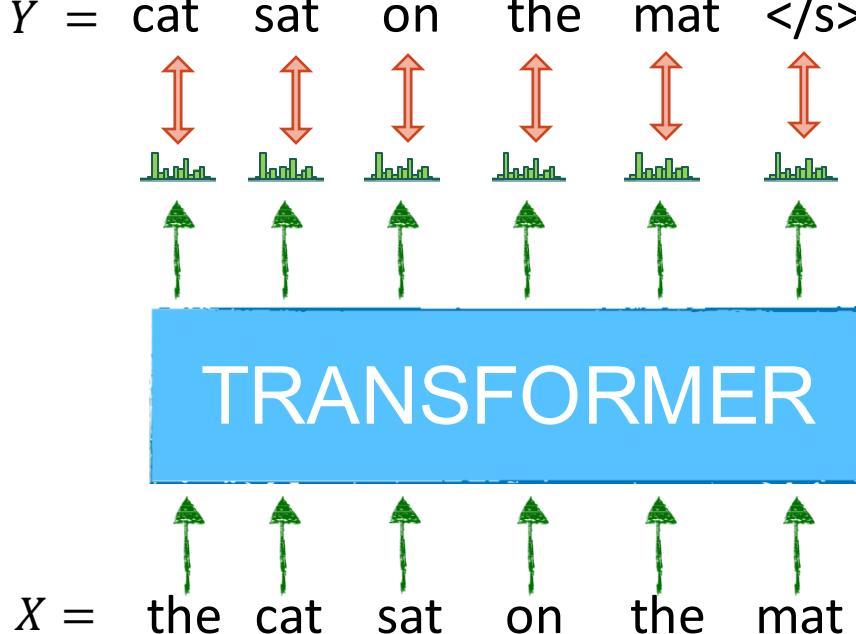
(gold output)  $Y = \text{cat sat on the mat } </s>$



# Training a Transformer Language Model

- For each position, compute the **loss** between the distribution and the gold output label.

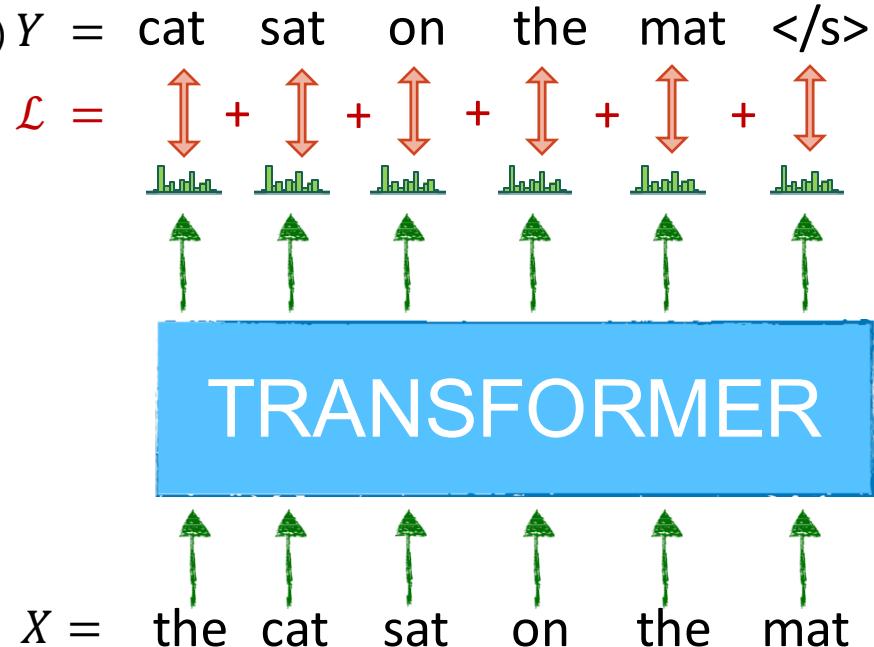
(gold output)  $Y = \text{cat sat on the mat } </s>$



# Training a Transformer Language Model

- Sum the position-wise loss values to obtain a **global loss**.

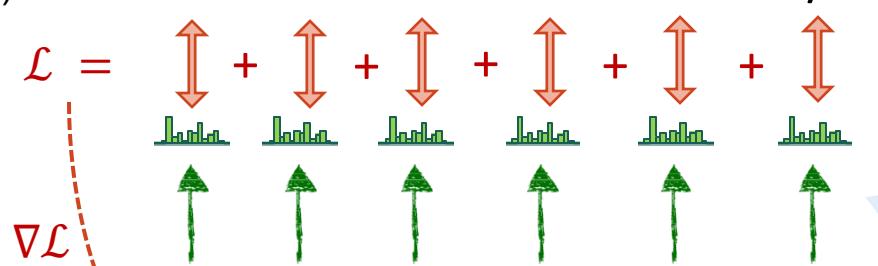
(gold output)  $Y = \text{cat sat on the mat } </s>$



# Training a Transformer Language Model

- Using this loss, do **Backprop** and **update** the Transformer parameters.

(gold output)  $Y = \text{cat sat on the mat } </s>$

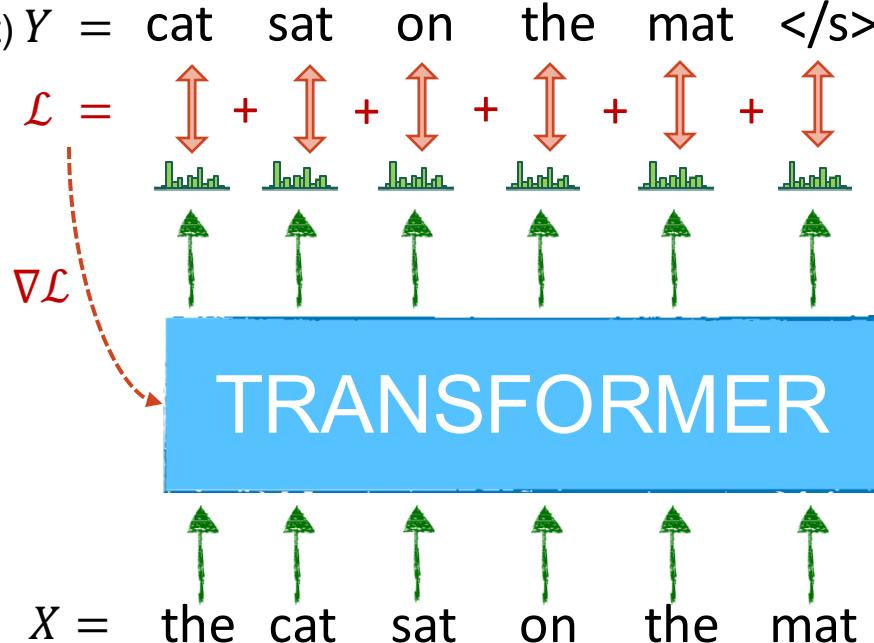


Well, this is not quite right 😊  
...  
what is the problem with this?

# Training a Transformer Language Model

- The model would solve the task by **copying** the next token to output (data leakage).
  - Does **not** learn anything useful

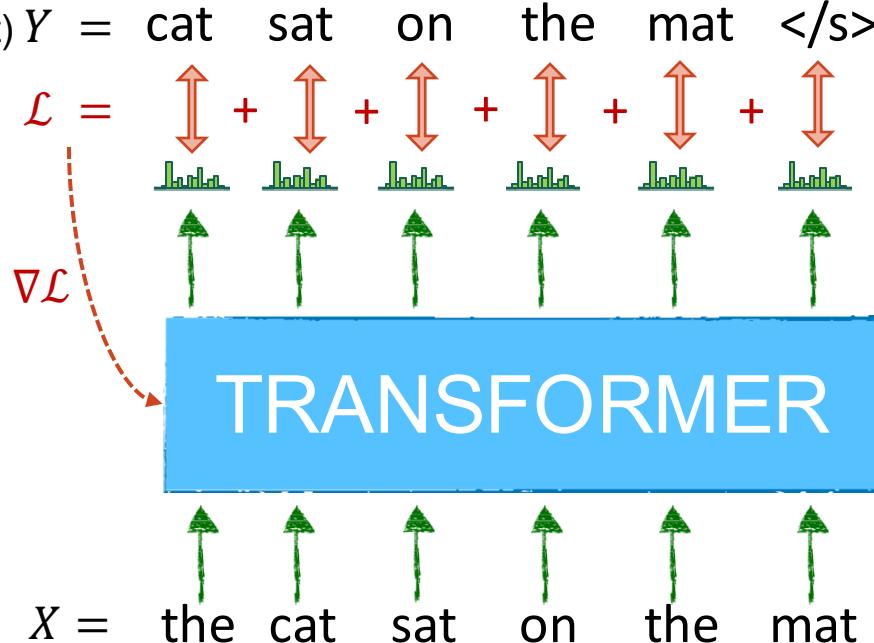
(gold output)  $Y = \text{cat sat on the mat } </s>$



# Training a Transformer Language Model

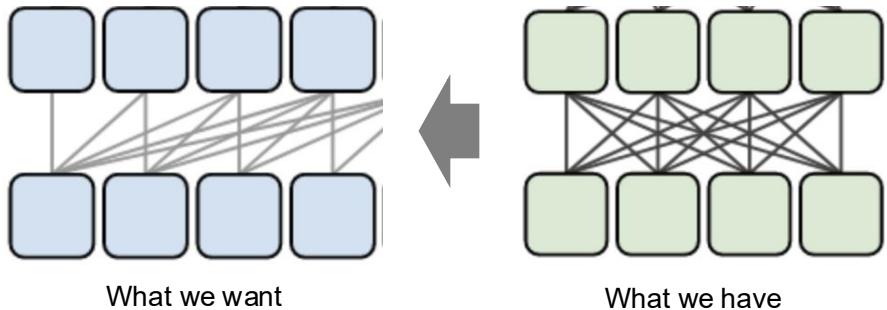
- We need to **prevent information leakage** from future tokens! How?

(gold output)  $Y = \text{cat sat on the mat } </s>$

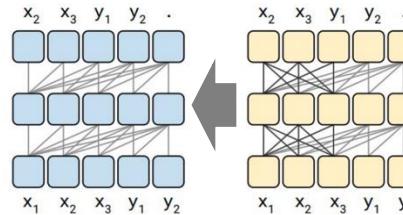


# Attention mask

Attention raw scores												
0	-0.08	1.24	0.69	-0.98	1.43	-0.6	0.7	0.16	0.93	1.28	-1.61	-1.1
1	-0.09	-0.0	-0.7	0.06	0.25	0.23	0.26	0.18	0.78	-0.21	-1.01	1.01
2	0.86	1.19	1.59	0.86	-0.13	-0.15	<b>-2.13</b>	-0.98	-0.87	-1.72	1.87	-0.72
3	0.12	-0.03	-0.02	0.88	-0.46	-0.7	0.54	-0.42	<b>-1.89</b>	-0.38	0.04	-0.84
4	0.51	0.17	0.13	<b>-1.64</b>	0.24	-0.02	1.68	-0.36	0.64	0.36	0.27	0.66
5	0.24	<b>-1.44</b>	0.43	0.74	0.96	<b>-1.21</b>	-0.31	1.54	1.66	1.14	0.58	<b>-1.44</b>
6	0.26	-0.1	0.93	0.72	-0.38	1.65	0.47	<b>-0.96</b>	-0.17	-0.9	<b>-1.57</b>	0.22
7	-0.55	0.81	0.71	1.7	-0.8	-1.14	-0.32	1.78	-0.7	-0.04	1.54	0.81
8	0.74	<b>-0.76</b>	-0.44	-0.08	<b>-1.38</b>	-0.13	1.25	<b>-1.37</b>	1.84	0.3	0.57	0.74
9	-0.97	<b>-0.91</b>	0.15	0.35	-0.81	0.11	1.14	<b>-1.52</b>	1.06	1.87	0.5	-0.3
10	1.56	0.9	0.39	1.46	1.44	<b>-1.05</b>	0.9	-0.73	0.36	-0.67	-0.62	-0.43
11	0.32	0.74	0.44	-0.1	1.19	0.83	0.29	2.06	0.51	-0.26	1.51	0.11



# Attention mask



Attention raw scores

	1	2	3	4	5	6	7	8	9	10	11	12
0	-0.08	1.24	0.69	-0.98	1.43	-0.6	0.7	0.16	0.93	1.28	-1.61	-1.1
1	-0.09	-0.0	-0.7	0.06	0.25	0.23	0.26	0.18	0.78	-0.21	-1.01	1.01
2	0.86	1.19	1.59	0.86	-0.13	-0.15	-2.13	-0.98	-0.87	-1.72	1.87	-0.72
3	0.12	-0.03	-0.02	0.88	-0.46	-0.7	0.54	-0.42	-1.89	-0.38	0.04	-0.84
4	0.51	0.17	0.13	-1.64	0.24	-0.02	1.68	-0.36	0.64	0.36	0.27	0.66
5	0.24	-1.44	0.43	0.74	0.96	-1.21	-0.31	1.54	1.66	1.14	0.58	-1.44
6	0.26	-0.1	0.93	0.72	-0.38	1.65	0.47	-0.96	-0.17	-0.9	-1.57	0.22
7	-0.55	0.81	0.71	1.7	-0.8	-1.14	-0.32	1.78	-0.7	-0.04	1.54	0.81
8	0.74	-0.76	-0.44	-0.08	-1.38	-0.13	1.25	-1.37	1.84	0.3	0.57	0.74
9	-0.97	-0.91	0.15	0.35	-0.81	0.11	1.14	-1.52	1.06	1.87	0.5	-0.3
10	1.56	0.9	0.39	1.46	1.44	-1.05	0.9	-0.73	0.36	-0.67	-0.62	-0.43
11	0.32	0.74	0.44	-0.1	1.19	0.83	0.29	2.06	0.51	-0.26	1.51	0.11

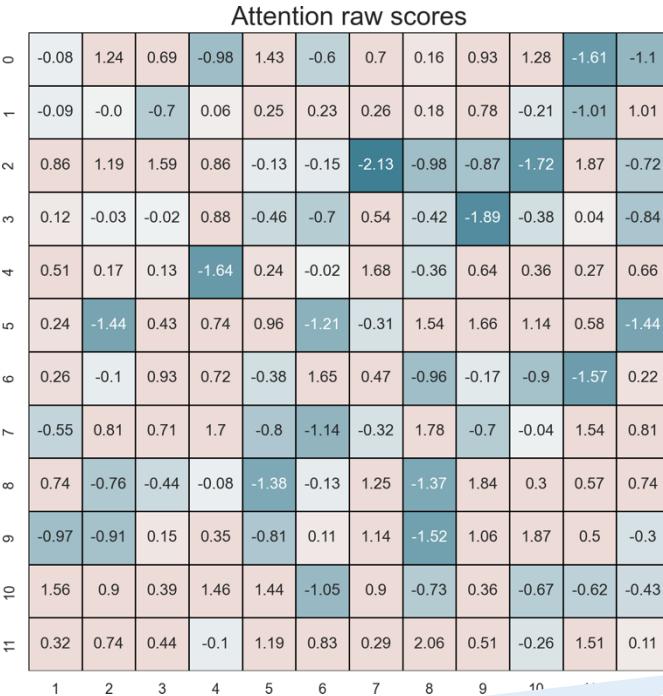
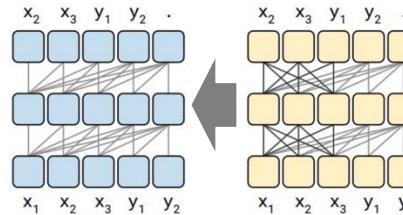
Attention mask

	0	1	2	3	4	5	6	7	8	9	10	11
0	1.0	-inf										
1	1.0	1.0	-inf									
2	1.0	1.0	1.0	-inf								
3	1.0	1.0	1.0	1.0	-inf							
4	1.0	1.0	1.0	1.0	1.0	-inf						
5	1.0	1.0	1.0	1.0	1.0	1.0	-inf	-inf	-inf	-inf	-inf	-inf
6	1.0	1.0	1.0	1.0	1.0	1.0	1.0	-inf	-inf	-inf	-inf	-inf
7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	-inf	-inf	-inf	-inf
8	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	-inf	-inf	-inf
9	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	-inf	-inf
10	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	-inf
11	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	-inf

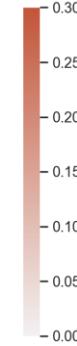


large negative numbers,  
which leads to  $\text{softmax}(-\infty) \approx 0$

# Attention mask

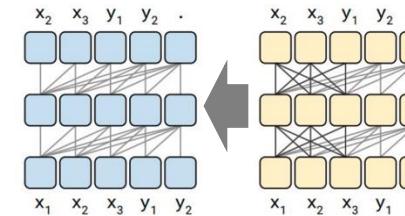


X  
Element-wise  
product



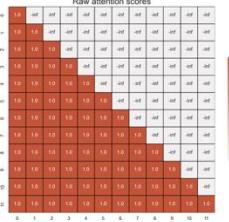
Note matrix operations is quite fast in GPUs.

# Attention mask



Attention raw scores												
	1	2	3	4	5	6	7	8	9	10	11	12
1	-0.08	0.34	0.09	0.38	1.43	-0.1	0.7	0.16	0.00	1.08	1.01	-1.11
2	-0.09	-0.3	-0.7	0.06	0.25	0.29	0.26	0.18	0.78	-0.21	1.01	1.01
3	-0.00	1.19	0.90	0.90	-0.13	0.15	-0.11	0.01	-0.05	-0.72	1.07	-0.76
4	0.12	-0.03	-0.42	0.88	-0.46	-0.1	0.54	-0.42	1.00	-0.38	0.94	-0.84
5	0.51	0.17	0.51	0.34	0.24	0.02	1.68	0.36	0.04	0.36	0.27	0.00
6	0.24	-1.04	0.40	0.78	0.98	1.1	-0.31	1.54	1.00	1.14	0.98	-1.04
7	0.26	-0.1	0.60	0.72	-0.38	1.05	0.47	0.96	-0.1	-0.5	1.01	0.23
8	0.85	0.91	0.71	1.7	-0.8	1.16	-0.32	1.78	-0.7	-0.54	1.04	0.81
9	0.74	-0.76	-0.48	0.08	-0.36	-0.13	0.13	0.37	1.00	0.3	0.97	0.74
10	0.97	0.91	0.15	0.30	0.81	0.11	1.14	-0.13	1.00	1.07	0.3	-0.23
11	0.32	0.74	0.44	0.1	1.19	0.83	0.29	2.06	0.01	-0.26	1.01	0.11
12	1	2	3	4	5	6	7	8	9	10	11	12

X



=

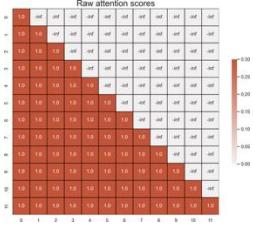
0	1	2	3	4	5	6	7	8	9	10	11	12
-0.08	-inf	-inf	-inf	-inf	-inf	-inf						
-0.09	-0.0	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf
0.86	1.19	1.59	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf
0.12	-0.03	-0.02	0.88	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf
0.51	0.17	0.13	-1.64	0.24	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf
0.24	-1.44	0.43	0.74	0.96	-1.21	-inf	-inf	-inf	-inf	-inf	-inf	-inf
0.26	-0.1	0.93	0.72	-0.38	1.65	0.47	-inf	-inf	-inf	-inf	-inf	-inf
-0.55	0.81	0.71	1.7	-0.8	-1.14	-0.32	1.78	-inf	-inf	-inf	-inf	-inf
0.74	-0.76	-0.44	-0.08	-1.38	-0.13	1.25	-1.37	1.84	-inf	-inf	-inf	-inf
-0.97	-0.91	0.15	0.35	-0.81	0.11	1.14	-1.52	1.06	1.87	-inf	-inf	-inf
1.56	0.9	0.39	1.46	1.44	-1.05	0.9	-0.73	0.36	-0.67	-0.62	-inf	-inf
0.32	0.74	0.44	-0.1	1.19	0.83	0.29	2.06	0.51	-0.26	1.51	0.11	

# Attention mask

The effect is more than just pruning out some of the wirings in self-attention block.

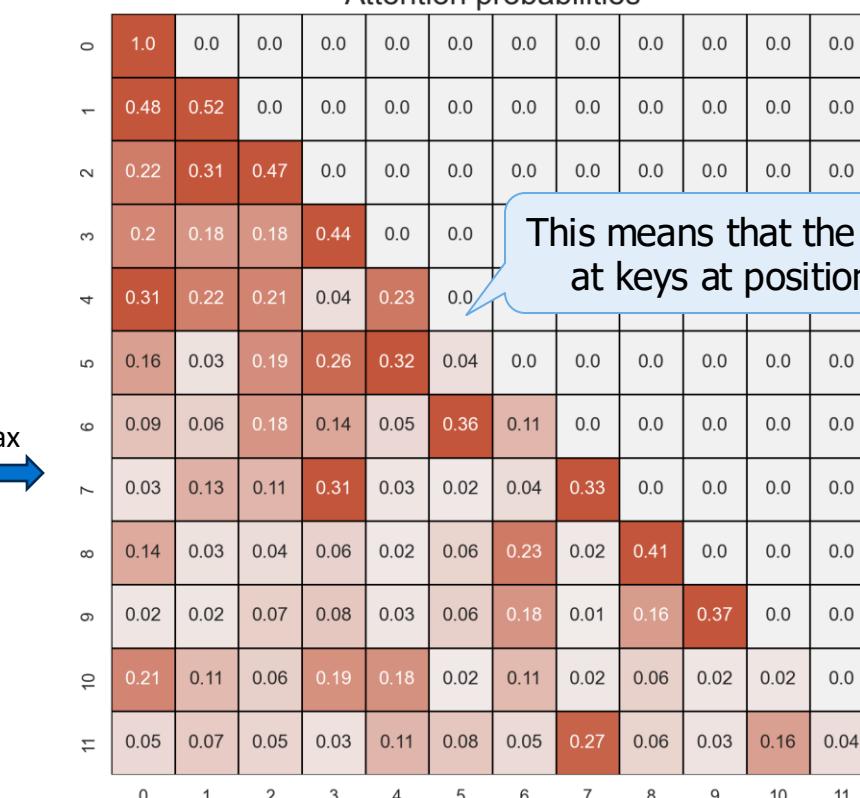
Attention raw scores											
-0.08	1.24	0.98	0.98	1.49	-0.1	0.7	0.26	0.18	0.93	1.08	1.01
-0.09	-0.3	-0.7	0.98	0.98	0.25	0.26	0.18	0.78	-0.21	1.01	1.01
-0.09	1.19	1.59	0.98	-0.15	0.15	1.11	-0.05	-0.05	1.22	1.07	-0.78
-0.12	-0.03	-0.42	0.88	-0.46	-0.13	0.54	-0.42	1.04	-0.38	0.94	-0.84
-0.51	0.17	0.55	0.54	0.24	0.02	1.68	0.36	0.04	0.37	0.06	0.06
-0.24	-1.64	0.61	0.78	0.98	1.25	-0.31	1.54	1.08	1.14	0.98	-1.04
-0.26	-0.1	0.60	0.72	0.28	1.05	0.47	0.96	-0.17	-0.5	1.01	0.22
-0.85	0.91	0.71	1.7	-0.18	0.32	1.78	-0.7	-0.54	1.04	0.81	0.81
0.74	-0.76	-0.48	0.88	-0.38	-0.13	1.25	1.37	1.84	0.3	0.97	0.74
-0.97	0.31	0.15	0.30	0.81	0.11	1.14	-1.13	1.08	1.07	0.93	-0.23
-0.16	0.9	0.39	1.46	1.44	1.25	0.8	0.73	0.38	-0.67	0.62	-0.42
-0.32	0.74	0.44	0.81	1.19	0.83	0.29	2.06	0.01	-0.28	1.51	0.11

X



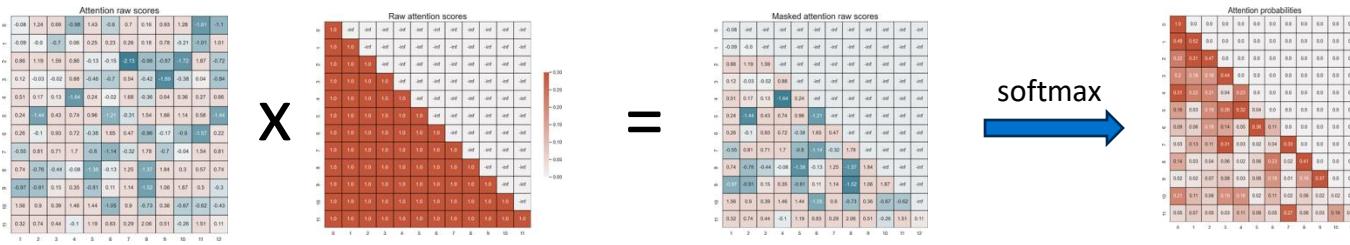
Masked attention raw scores											
-0.08	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf
-0.09	0.0	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf
-0.09	1.19	1.59	inf	inf	inf	inf	inf	inf	inf	inf	inf
-0.12	-0.03	-0.42	0.88	-0.46	-0.13	0.54	-0.42	1.04	-0.38	0.94	-0.84
-0.51	0.17	0.55	0.54	0.24	0.02	1.68	0.36	0.04	0.37	0.06	0.06
-0.24	-1.64	0.61	0.78	0.98	1.25	-0.31	1.54	1.08	1.14	0.98	-1.04
-0.26	-0.1	0.60	0.72	0.28	1.05	0.47	0.96	-0.17	-0.5	1.01	0.22
-0.85	0.91	0.71	1.7	-0.18	0.32	1.78	-0.7	-0.54	1.04	0.81	0.81
0.74	-0.76	-0.48	0.88	-0.38	-0.13	1.25	1.37	1.84	0.3	0.97	0.74
-0.97	0.31	0.15	0.30	0.81	0.11	1.14	-1.13	1.08	1.07	0.93	-0.23
-0.16	0.9	0.39	1.46	1.44	1.06	0.8	0.73	0.38	-0.67	0.62	-0.42
-0.32	0.74	0.44	0.81	1.19	0.83	0.29	2.06	0.01	-0.28	1.51	0.11

softmax



# Attention masking: Why Before Softmax?

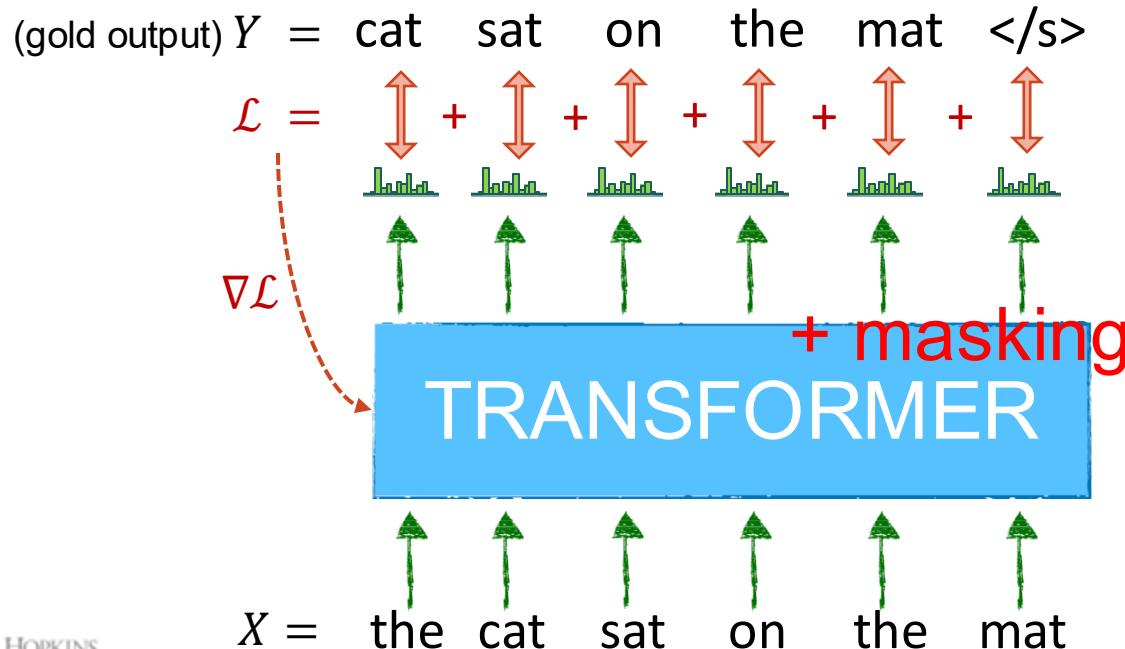
- We applied attention masking before softmax. Why not after?



- Softmax normalizes the scores so that it's a probability. Masking after softmax, would lead to an unnormalized probability distribution.
  - In practice, I (Daniel) suspects applying mask after softmax may just work! ☺

# Training a Transformer Language Model

- We need to **prevent information leakage** from future tokens! How?



# Quiz: attention masking

- What's the point of attention masking during training?
  - A. speeds up training by reducing the number of computations performed.
  - B. prevents accessing future tokens, ensuring that predictions are based only on past and present information.
  - C. selectively highlights important tokens in the input, thereby enhancing attention mechanism.

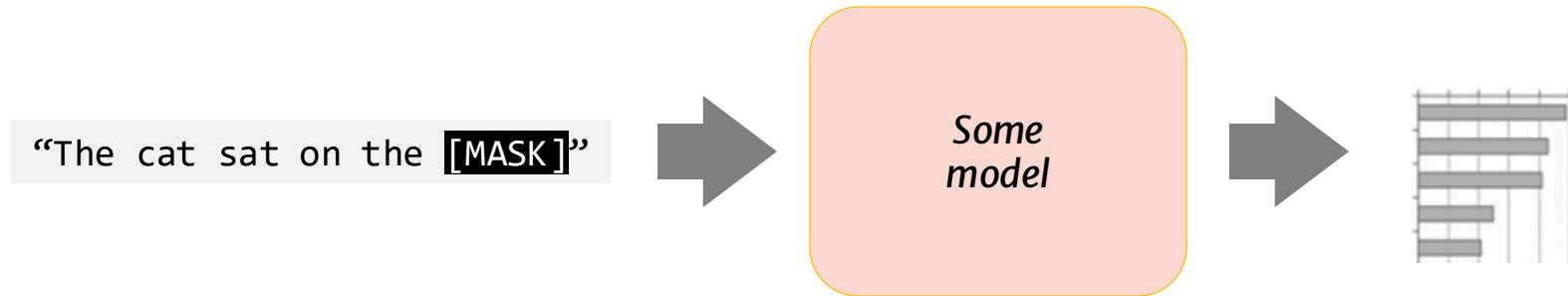


# Transformer Architectural Variants

# Encoder-Decoder Architectures

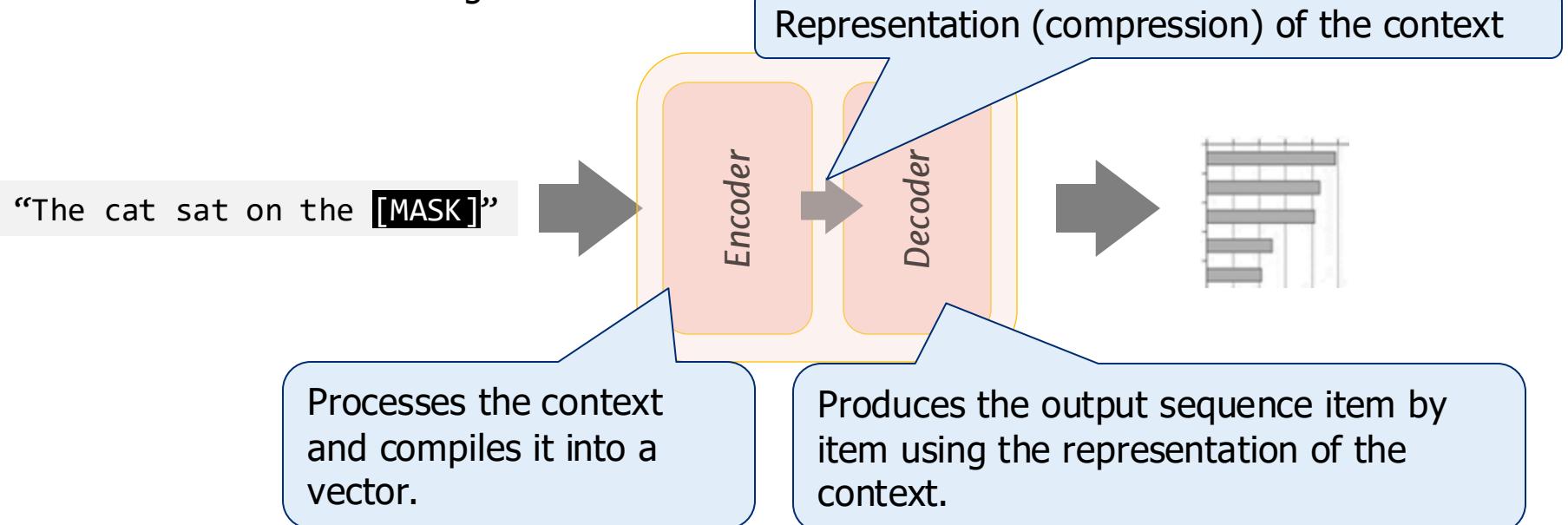
---

- It is useful to think of generative models as two sub-models.



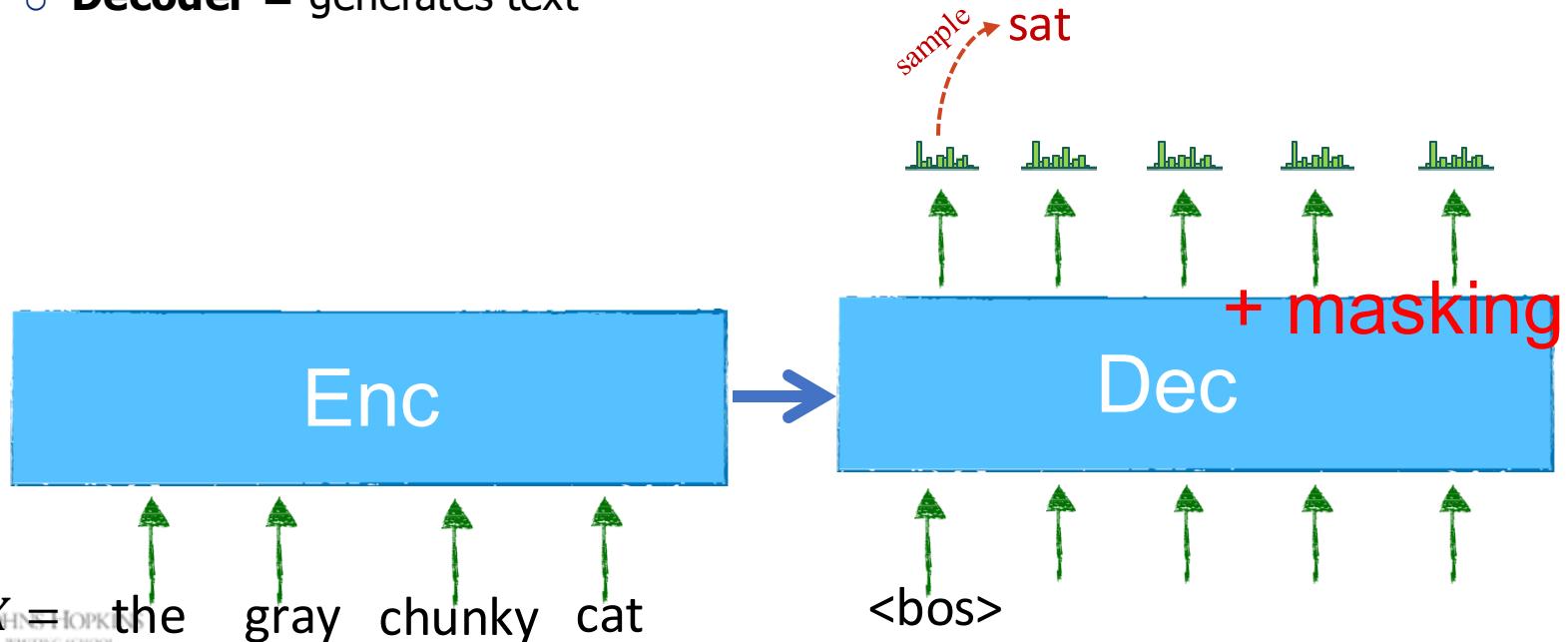
# Encoder-Decoder Architectures

- It is useful to think of generative models as two sub-models



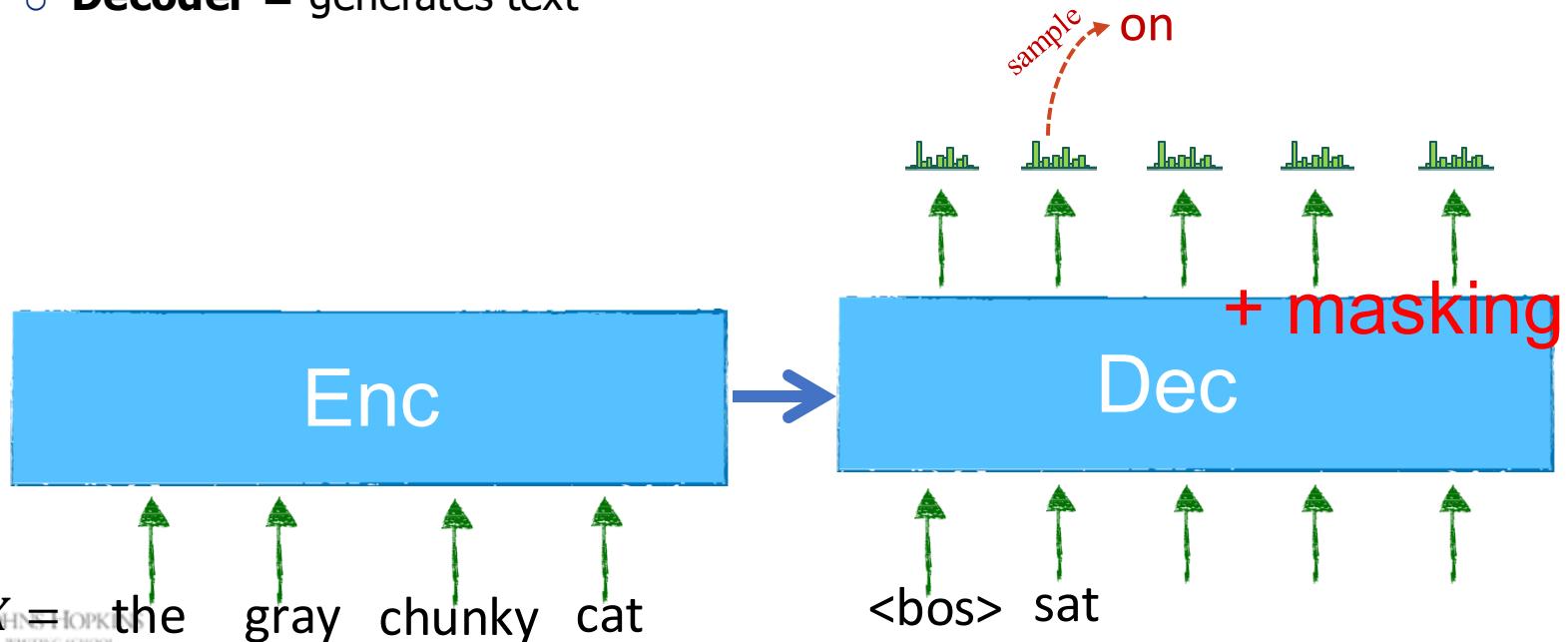
# Enc-Dec work at inference time

- Transformer is two blocks
  - **Encoder** = read or encode the input
  - **Decoder** = generates text



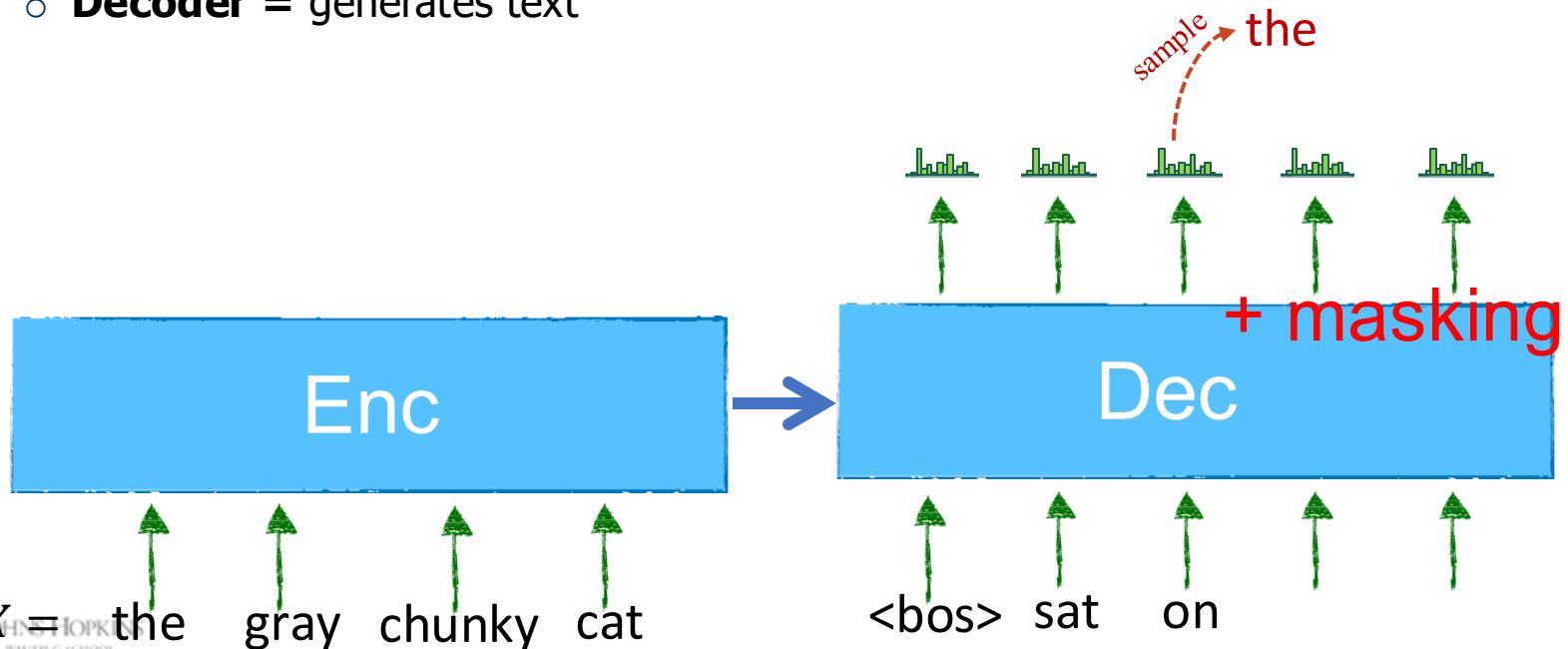
# Enc-Dec work at inference time

- Transformer is two blocks
  - **Encoder** = read or encode the input
  - **Decoder** = generates text



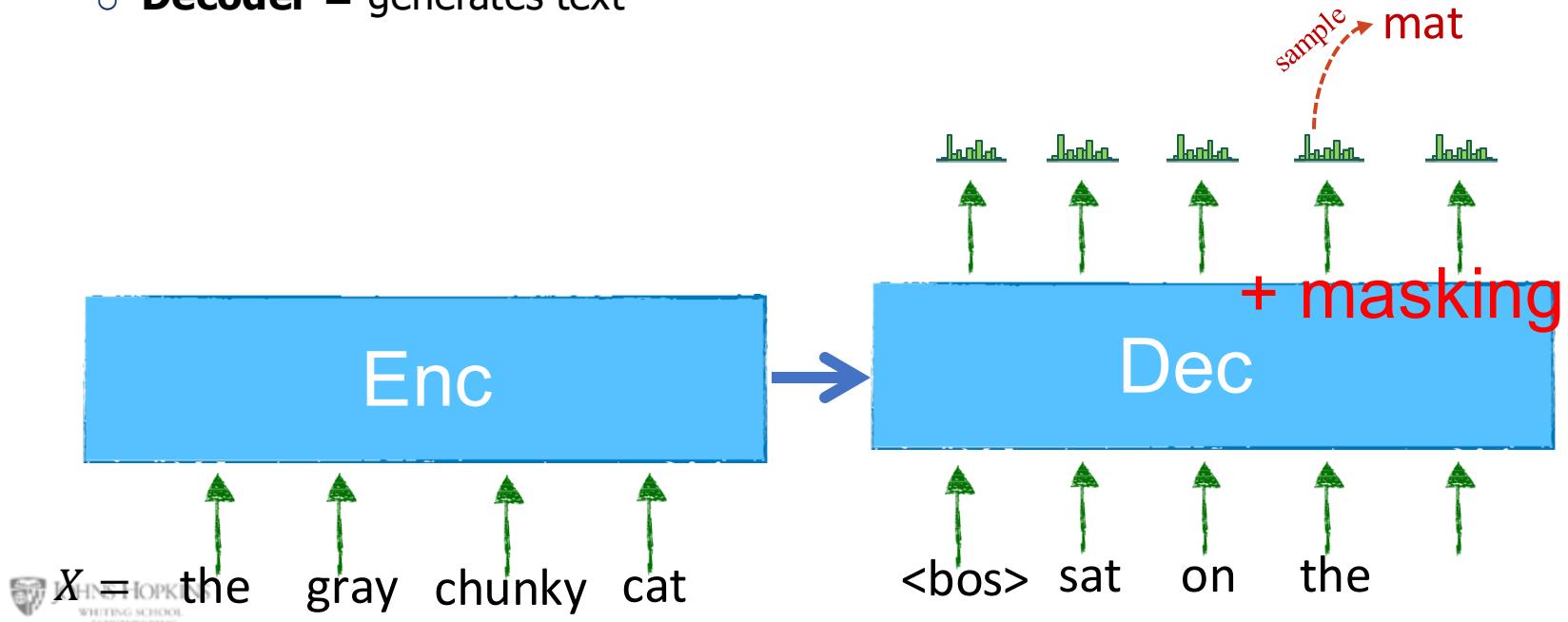
# Enc-Dec work at inference time

- Transformer is two blocks
  - **Encoder** = read or encode the input
  - **Decoder** = generates text



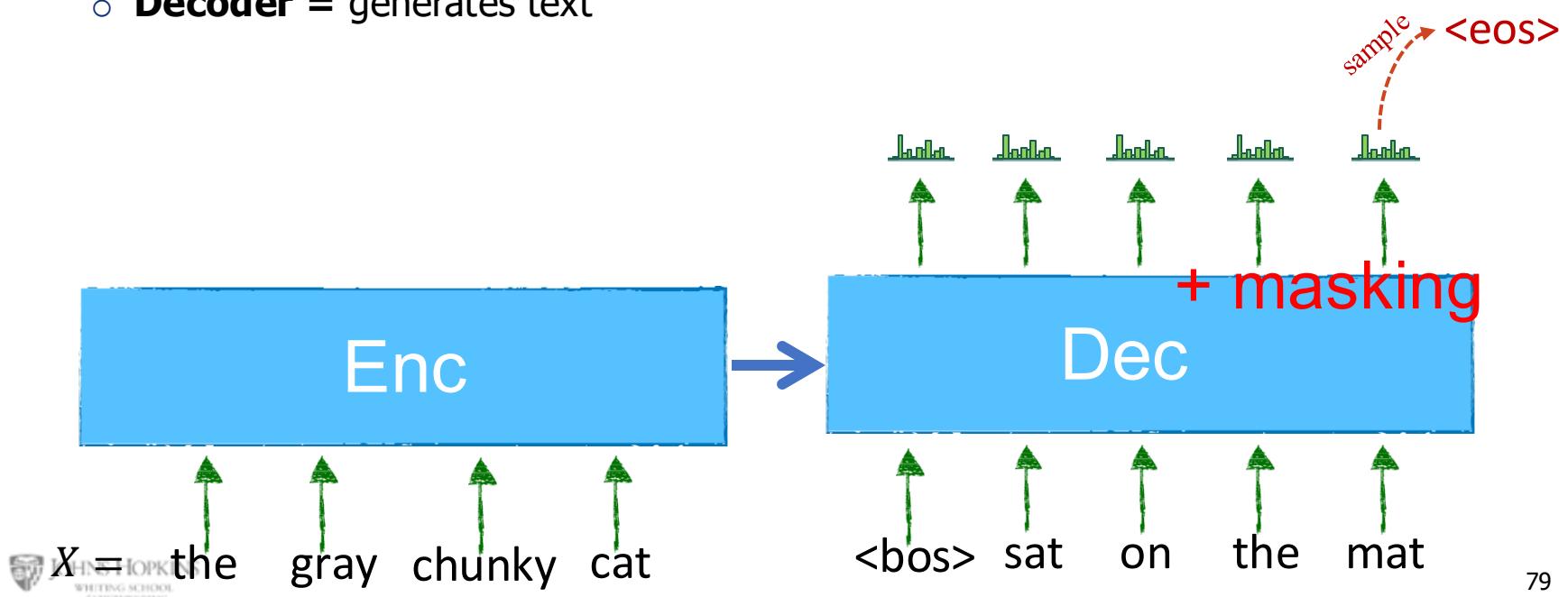
# Enc-Dec work at inference time

- Transformer is two blocks
  - **Encoder** = read or encode the input
  - **Decoder** = generates text



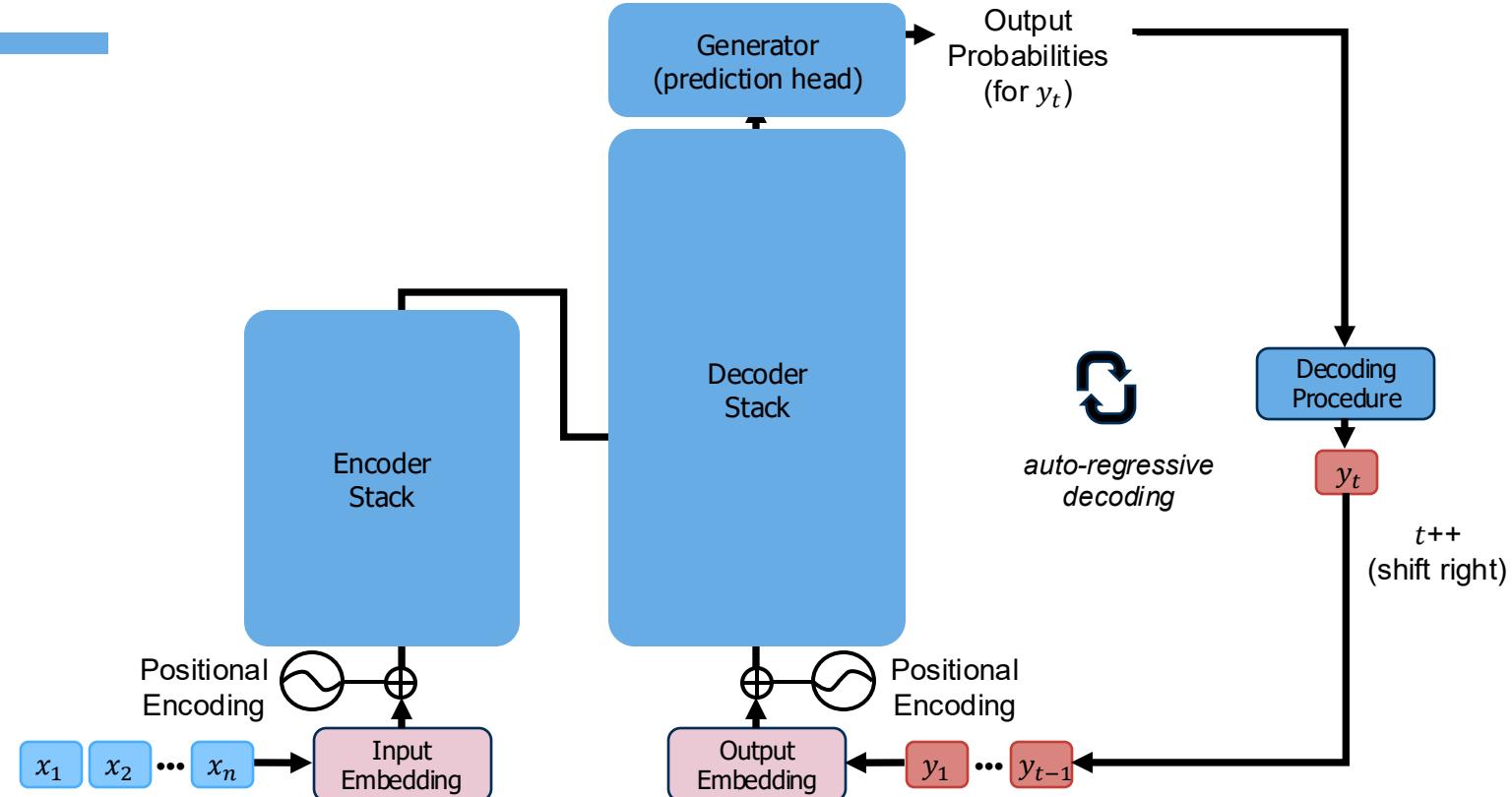
# Enc-Dec work at inference time

- Transformer is two blocks
  - **Encoder** = read or encode the input
  - **Decoder** = generates text



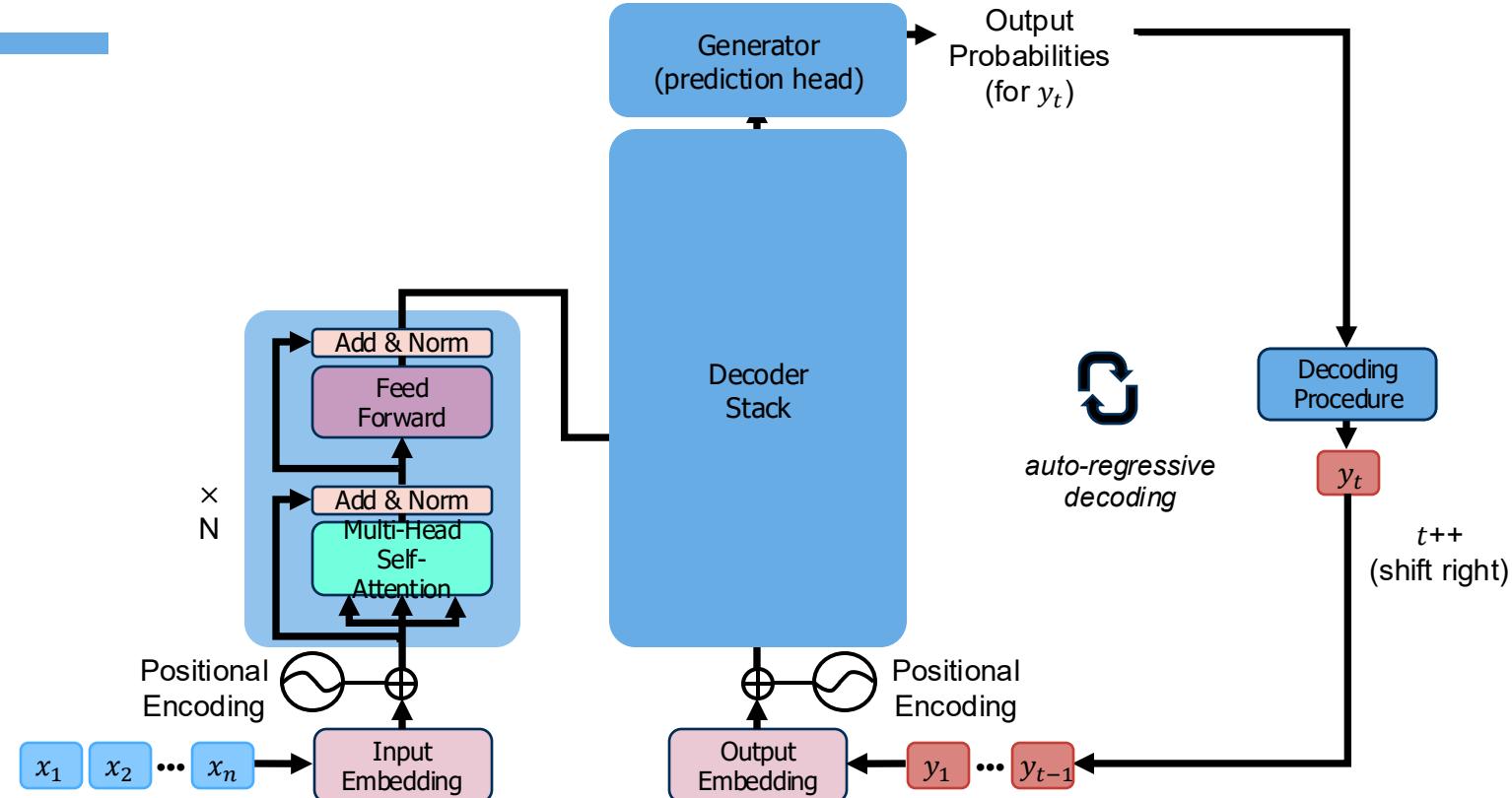
# Transformer

[Vaswani et al. 2017]



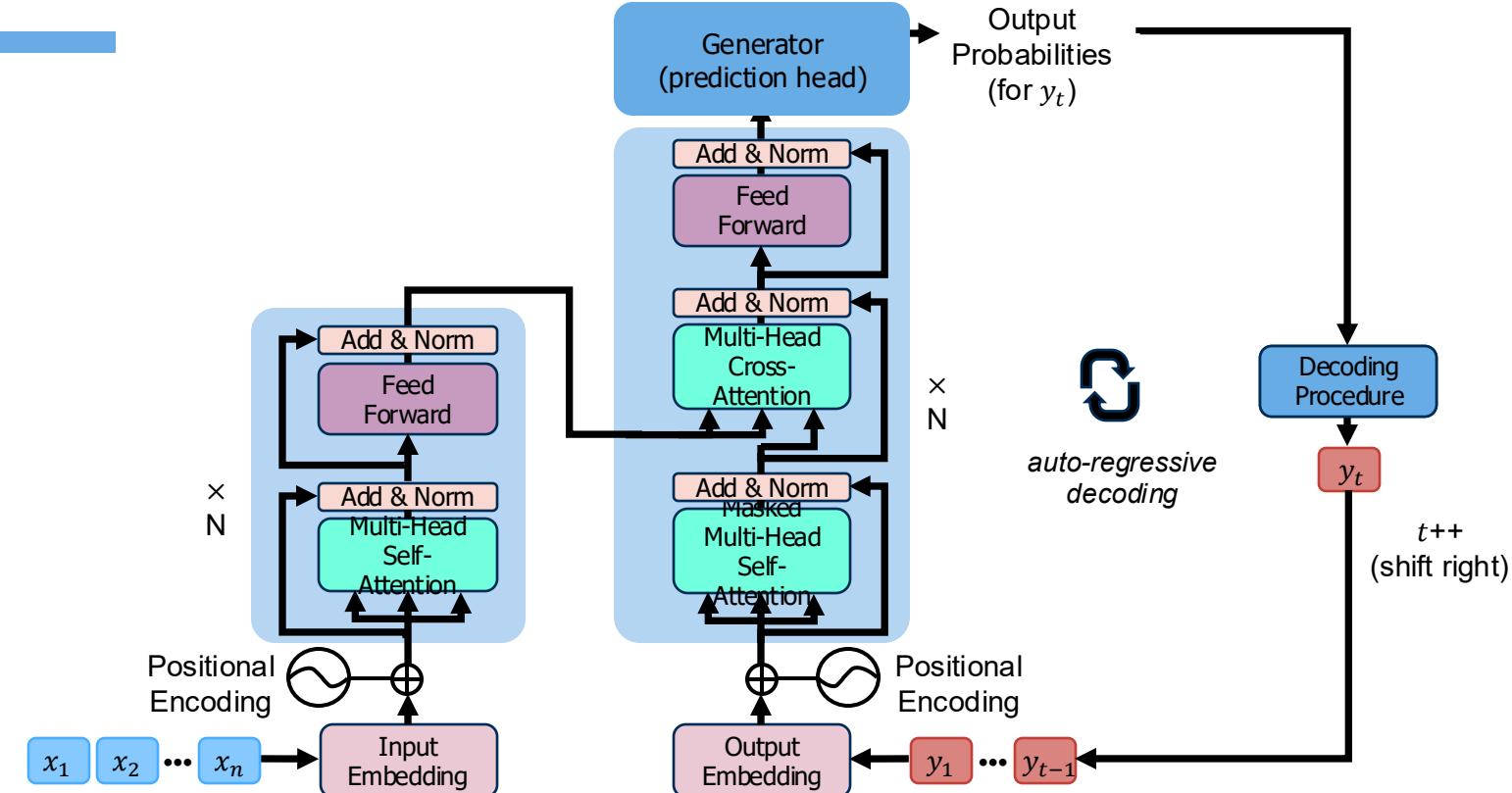
# Transformer

[Vaswani et al. 2017]



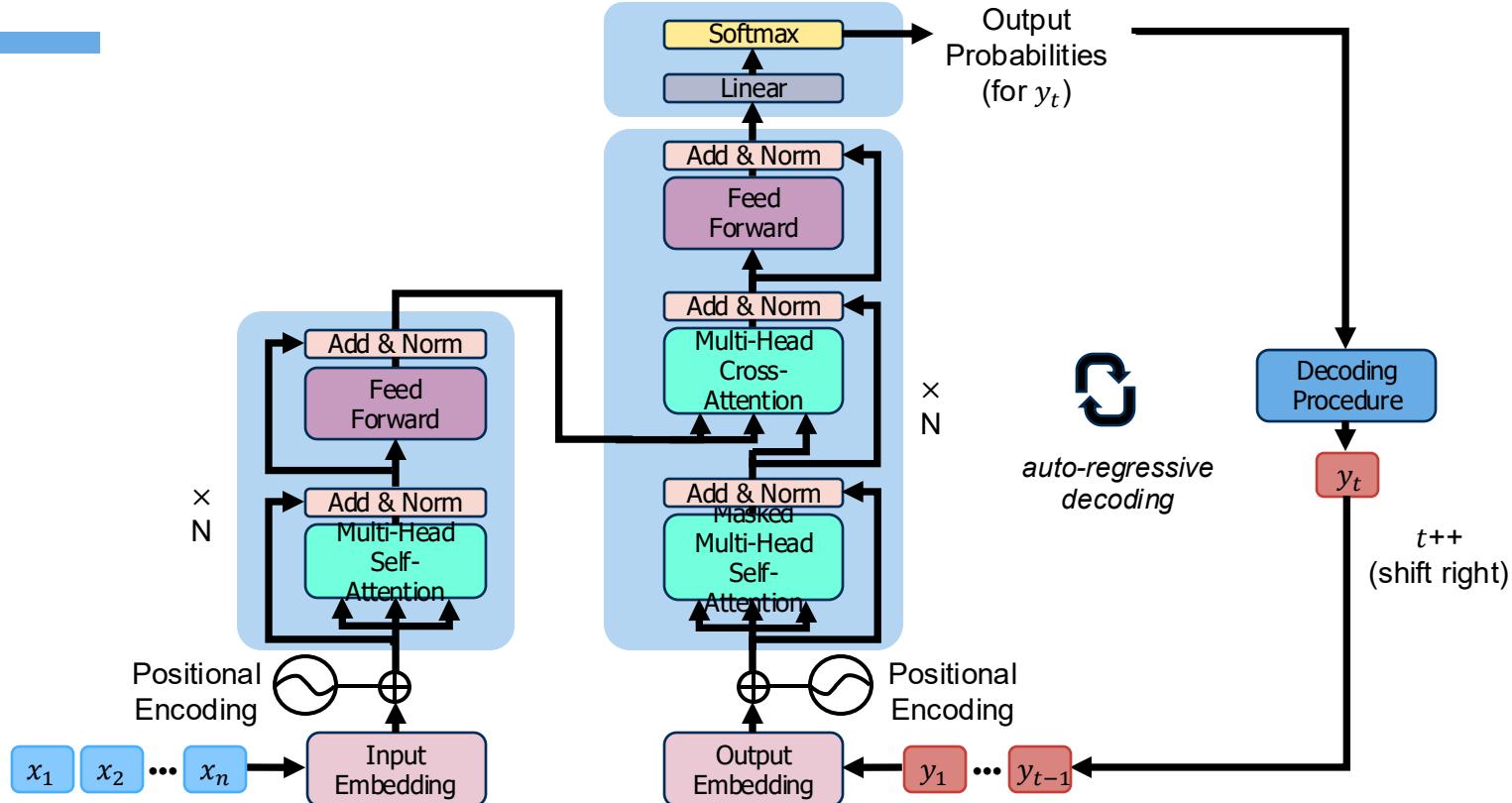
# Transformer

[Vaswani et al. 2017]



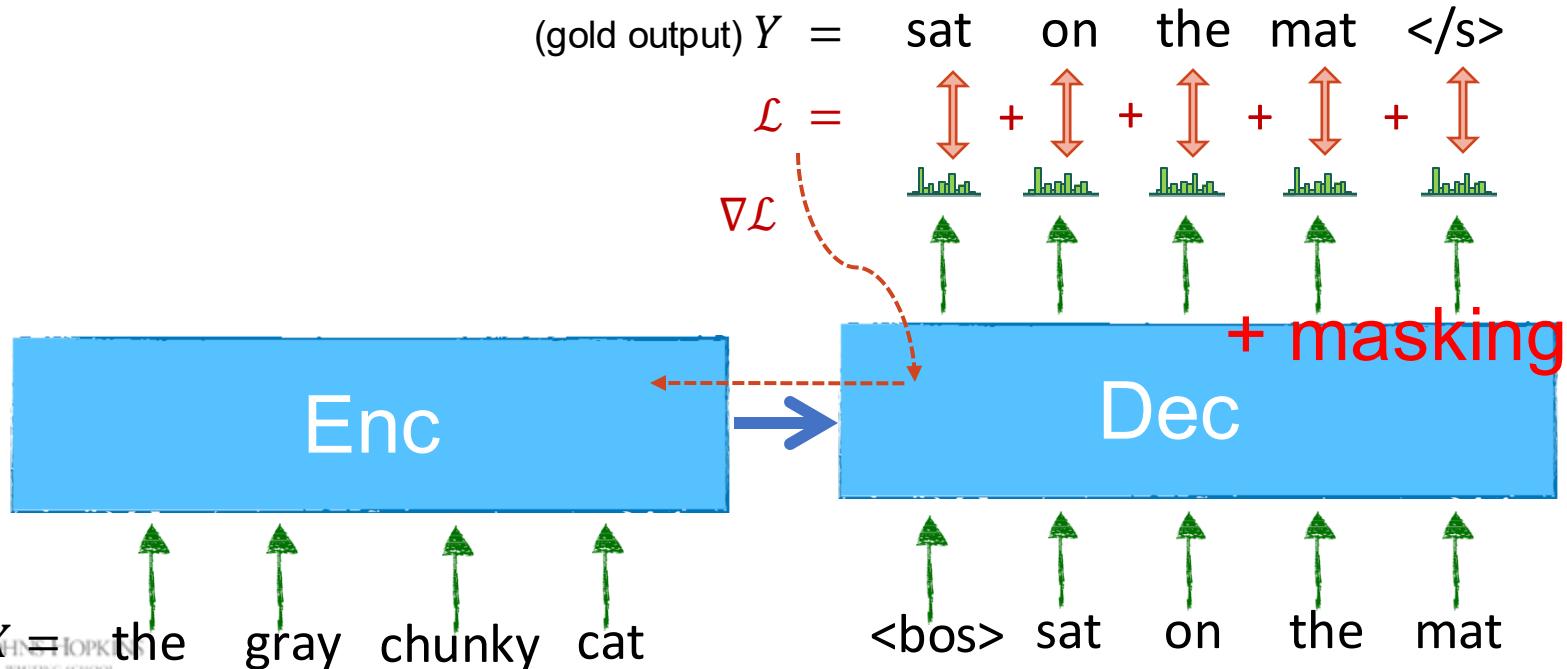
# Transformer

[Vaswani et al. 2017]



# How does training Enc-Dec work?

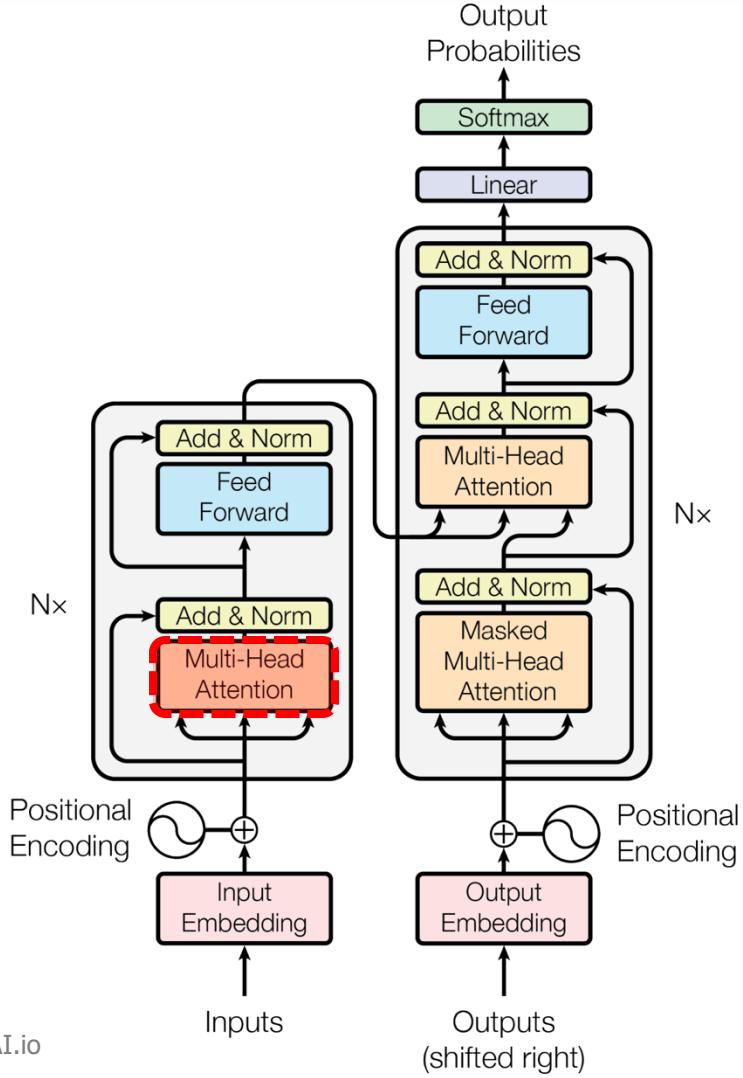
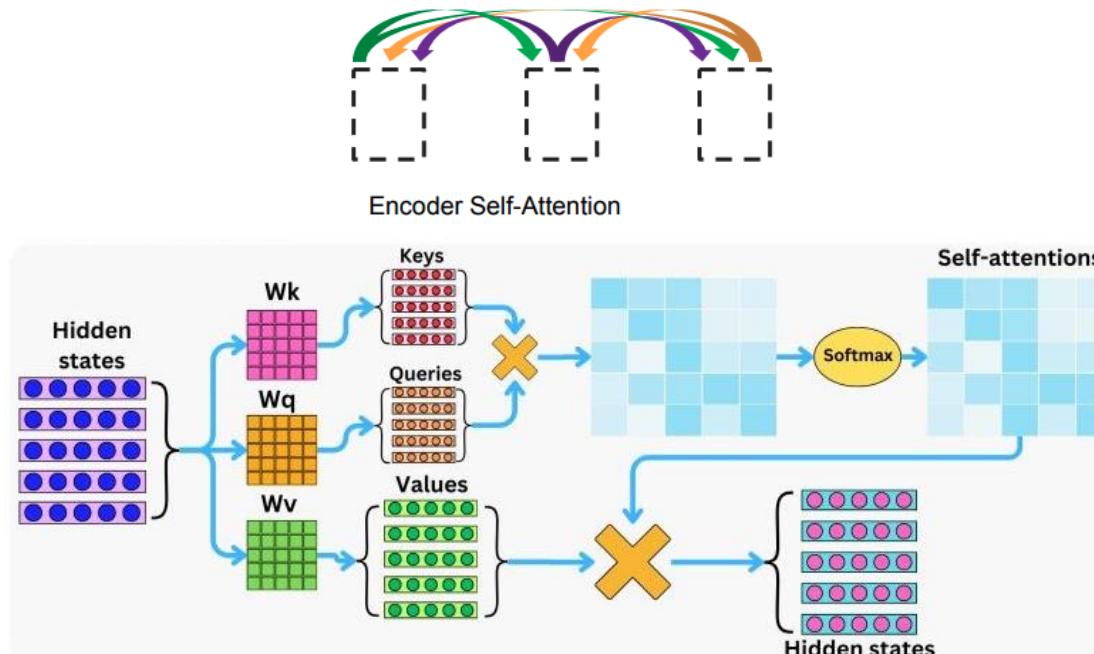
- Same as what we did before for encoder-only models.



# Transformer

 [Vaswani et al. 2017]

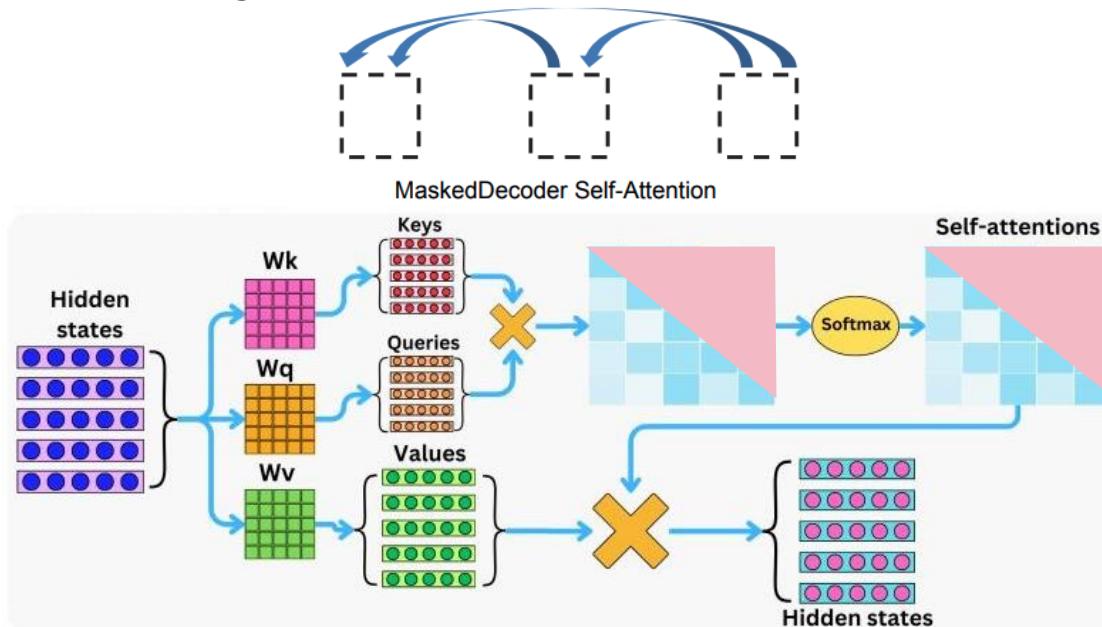
- Computation of **encoder** attends to both sides.



# Transformer

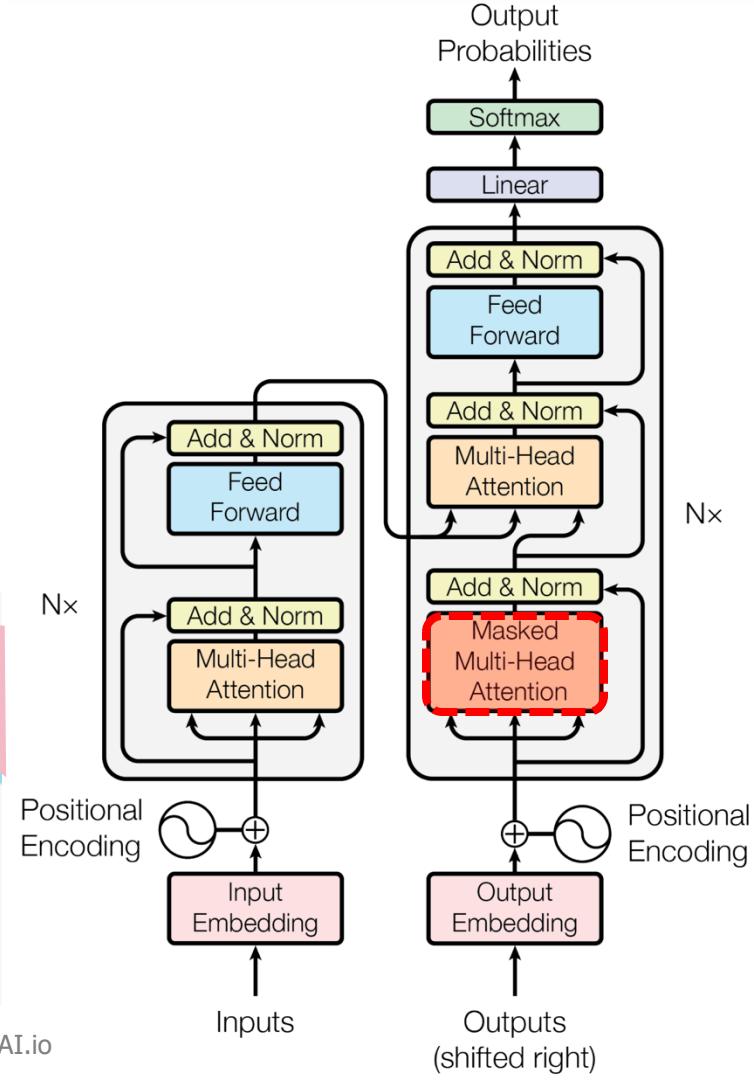
 [Vaswani et al. 2017]

- At any step of **decoder**, it attends to previous computation of **encoder** as well as **decoder's** own generations.



[Attention Is All You Need, Vaswani et al. 2017]

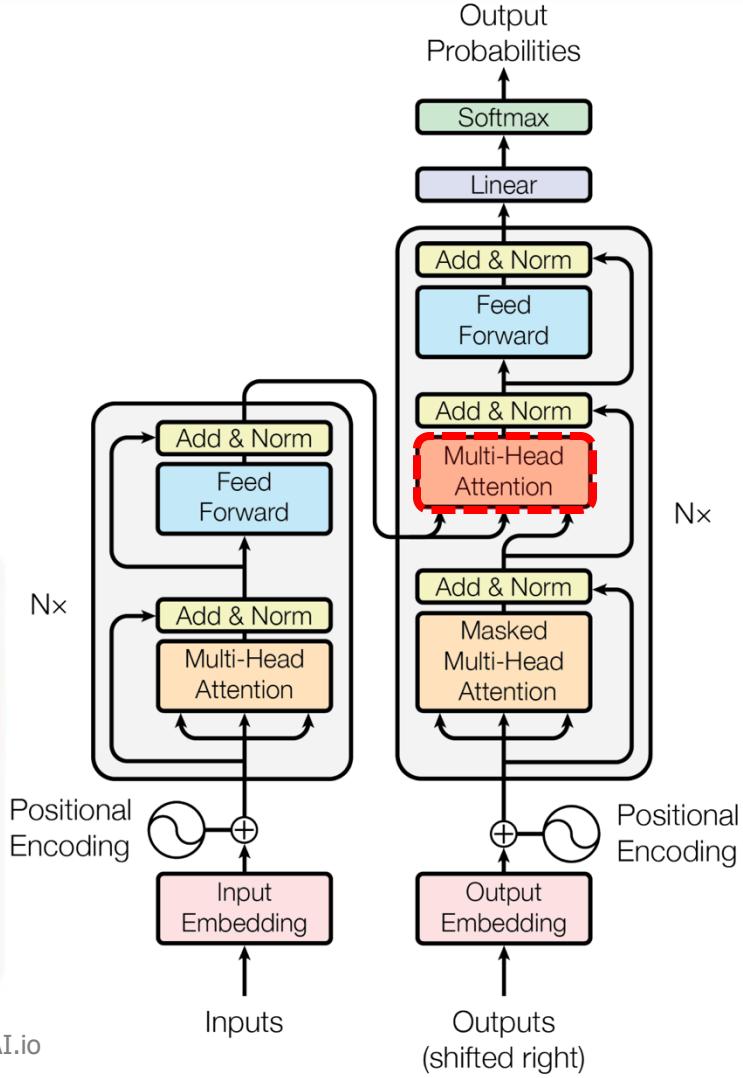
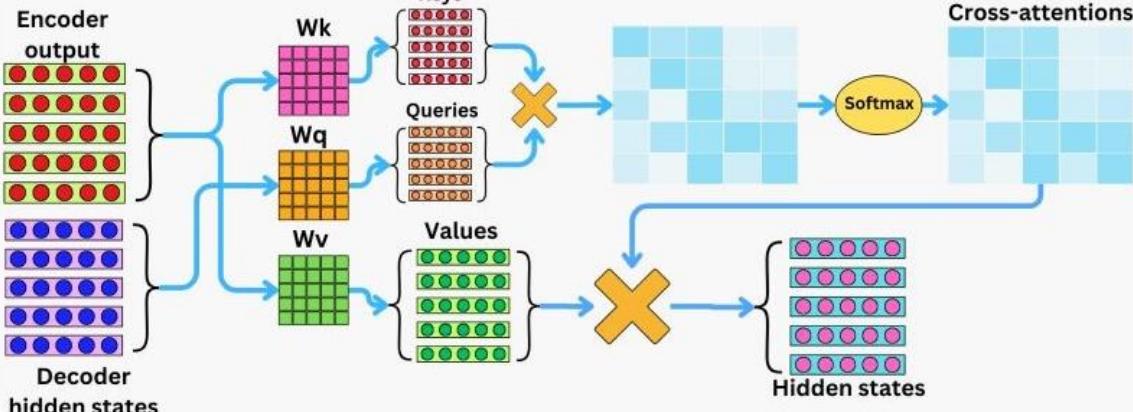
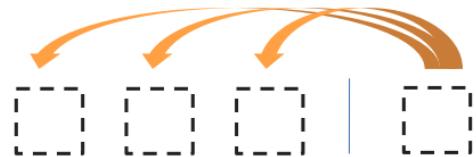
Visualization: theEdgeAI.io



# Transformer

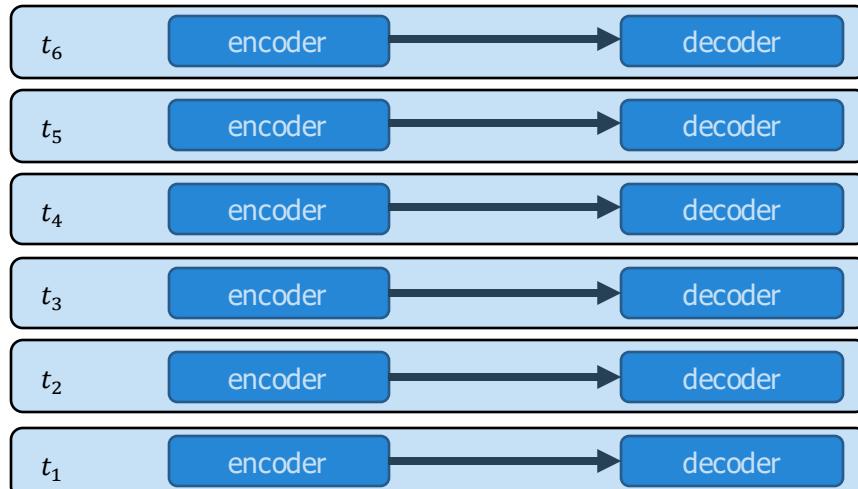
[Vaswani et al. 2017]

- At any step of **decoder**, it attends to previous computation of **encoder**.

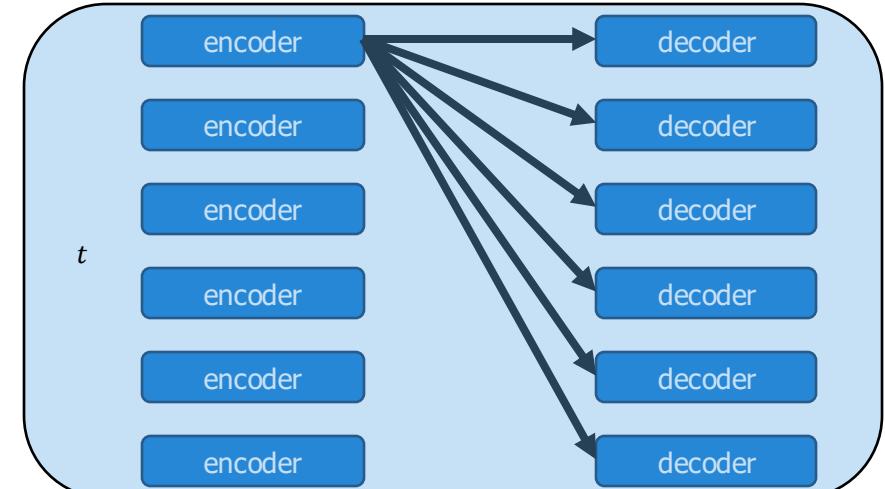


# Quiz: Enc-Dec Connections

- Which best represents encoder-decoder connections? (y-axis denotes the layers)



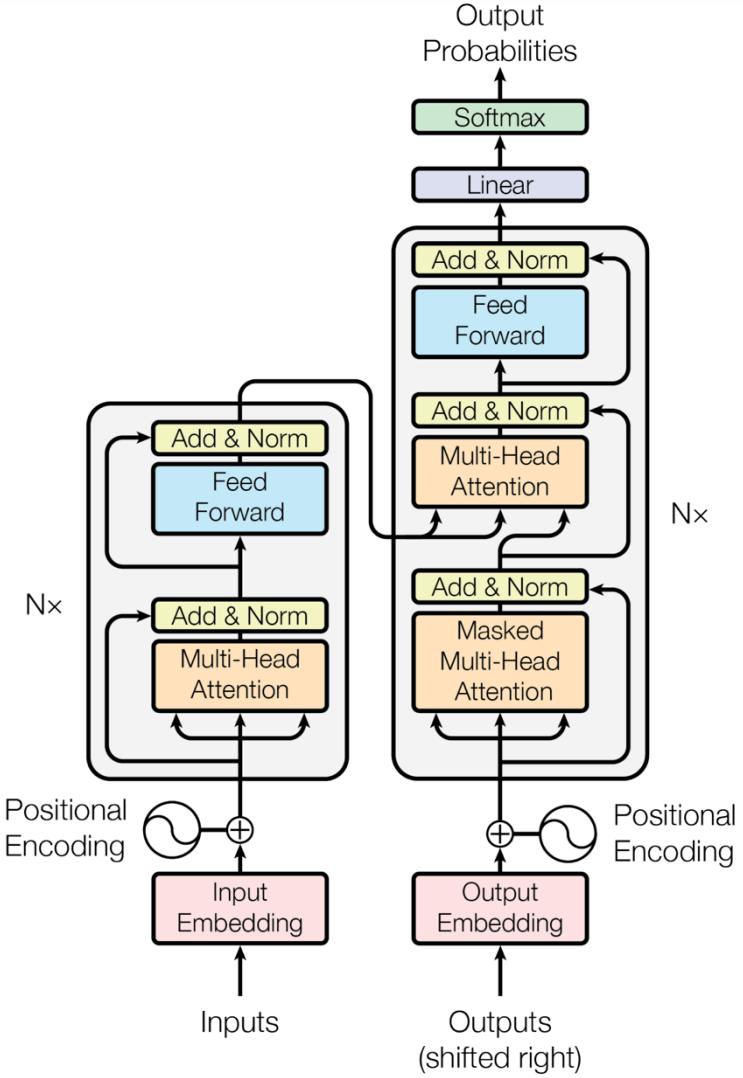
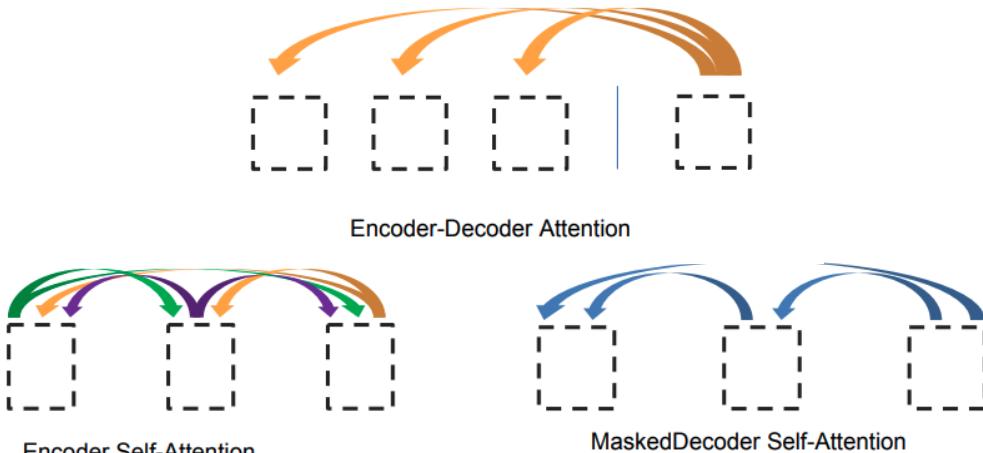
Incorrect



Correct

# Recap: Transformer

- Yaaay we know Transformers now! 🎉
- An **encoder-decoder** architecture
- 3 forms of attention



# Considerations about computational cost in Transformers

# Diversion: Floating-point Ops: FLOPS

---

- Floating point operations per second (FLOPS, flops or flop/s)
- Each FLOP can represent an addition, subtraction, multiplication, or division of floating-point numbers,
- The total FLOP of a model (e.g., Transformer) provides a basic approximation of computational costs associated with that model.

# FLOPS of Matrix Multiplication

- Matrix-vector multiplication are common in Self-Attention (e.g., QKV projection)
  - Requires  $2mn$  ( $2 \times$  matrix size) operations for multiplying  $A \in \mathbb{R}^{m \times n}$  and  $b \in \mathbb{R}^n$
  - (2 because 1 for multiplication, 1 for addition)

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m1} & A_{m2} & \cdots & A_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} A_{11}x_1 + A_{12}x_2 + \cdots + A_{1n}x_n \\ A_{21}x_1 + A_{22}x_2 + \cdots + A_{2n}x_n \\ \vdots \\ A_{m1}x_1 + A_{m2}x_2 + \cdots + A_{mn}x_n \end{bmatrix}$$

# Quiz: thinking about computations

- Consider the following matrix multiplication:

$$A \cdot B, \quad \text{where } A \in \mathbb{R}^{n \times m}, B \in \mathbb{R}^{m \times k}$$

- **Question 1:** Computing  $AB$  involves how many arithmetic operations?

- (also referred to as floating-point operations or FLOPs)

- **Answer:** It's  $O(n \times m \times k)$ .

- (to be a bit more precise, it's  $\approx 2n \times m \times k$  since each element in  $A \cdot B$  requires almost equal num of multiplications and summations.)

- **Question 2:** Computing  $AB$  involves how many memory/IO access?

- **Answer:** It's  $n \times m$  (reading A) +  $m \times k$  (reading B) +  $n \times m$  (writing AB).

# Computations in Self-Attention Block

---

- We are going to count computations and IO access in Transformer computations.
- Note we assume that the full input sequence is given at once (e.g., training time).
- Here is the first step. Given:  $\mathbf{x} \in \mathbb{R}^{b \times n \times d}$ ,  $\mathbf{W}_i^q \in \mathbb{R}^{d \times \frac{d}{m}}$
- Let's think about the following computation:  $\mathbf{x}\mathbf{W}_i^q$
- **Q1:** What is the number of arithmetic operations?  $O(b \times n \times d \times \frac{d}{m})$  for each head
- **Q2:** What about the number of IO?  $O(b \times n \times d + d \times \frac{d}{m} + b \times n \times \frac{d}{m})$  for each head
- **Q3:** What are these quantities for all heads?
  - Number of arithmetic ops:  $O(bnd^2)$
  - Number of IO ops:  $O(2bnd + d^2)$

# Computations in Self-Att

$b$ : batch size,  
 $n$ : sequence length,  
 $m$ : number of heads  
 $d$ : feature dimension in output of SA  
 $d/m$ : feature dimension inside each SA head  
 $d_{\text{ff}} = 4d$ : feature dimension inside FFN

Dimensions	Operation	Computations	IO
$\mathbf{x} \in \mathbb{R}^{b \times n \times d}, \mathbf{W}_i^q \in \mathbb{R}^{d \times \frac{d}{m}}$	$\mathbf{x}\mathbf{W}_i^q, \mathbf{x}\mathbf{W}_i^k, \mathbf{x}\mathbf{W}_i^v$ for $m$ heads	$O(bnd^2)$	$O(d^2 + 2bnd)$
$\mathbf{Q}_i, \mathbf{K}_i \in \mathbb{R}^{b \times n \times \frac{d}{m}}$	$P_i \leftarrow \text{softmax}\left(\frac{\mathbf{Q}_i \mathbf{K}_i^T}{\sqrt{d/m}}\right)$ for $m$ heads	$O(bn^2d)$	$O(2bnd + bmn^2)$
$\mathbf{V}_i \in \mathbb{R}^{b \times n \times \frac{d}{m}}, P_i \in \mathbb{R}^{b \times n \times n}$	$\text{head}_i \leftarrow P_i \mathbf{V}_i$ for $m$ heads	$O(bn^2d)$	$O(2bnd + bmn^2)$
$\mathbf{W}^o \in \mathbb{R}^{d \times d}, \text{head}_i \in \mathbb{R}^{b \times n \times \frac{d}{m}}$	$Y = \text{Concat}(\text{head}_1, \dots, \text{head}_m) \mathbf{W}^o$	$O(bnd^2)$	$O(2bnd + d^2)$
$Y \in \mathbb{R}^{b \times n \times d}, \mathbf{W}_1 \in \mathbb{R}^{d \times d_{\text{ff}}}, \mathbf{W}_2 \in \mathbb{R}^{d_{\text{ff}} \times d}$	$Y = \text{ReLU}(Y \mathbf{W}_1) \mathbf{W}_2$	$O(16bnd^2)$	$O(2bnd + 8d^2)$
---	Total	$O(bnd^2 + bn^2d)$	$O(bnd + bmn^2 + d^2)$

# The bottlenecks

$b$ : batch size,  
 $n$ : sequence length,  
 $d$ : feature dimension in output of SA

- So, in total, we have →
- The quadratic terms are based on  $n$  and  $d$
- $d$  is fixed (part of architecture) but  $n$  changes with input.
- **Bottlenecks #1:** If  $n$  (sequence length)  $\gg d$  (feature dimension), the time and space complexity would be dominated by  $O(n^2)$ .
- **However,** these despite this quadratic dependence these are parallelizable operations which can be computed efficiently in GPUs.

- In comparison, RNNs perform less arithmetic ops but they're not all parallelizable.

- **Bottlenecks #2:** Another potential bottleneck is how fast we can run IO.  
A good way to think about this is using "Arithmetic Intensity" (more on this later)

Computations	IO
$O(bnd^2 + bn^2d)$	$O(bnd + bmn^2 + d^2)$

Layer Type	Complexity per Layer	Sequential Operations
Self-Attention	$O(n^2 \cdot d)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$

[Vaswani et al. 2017]

# Recap

---

- Transformers **computation of a full sequence** is bounded by  $O(bnd^2 + bn^2d)$ .
  - Generally, the quadratic term that depends on seq len  $n$  is more concerning.
- We have not discussed yet how **IO may impose** other limits on this. (in a week)
- Also, the above calculations is for a given sentences.
  - How bad is the computational complexity during the decoding time where we want to generate text one token at a time?

During decoding time, how slow  
is attention computation?

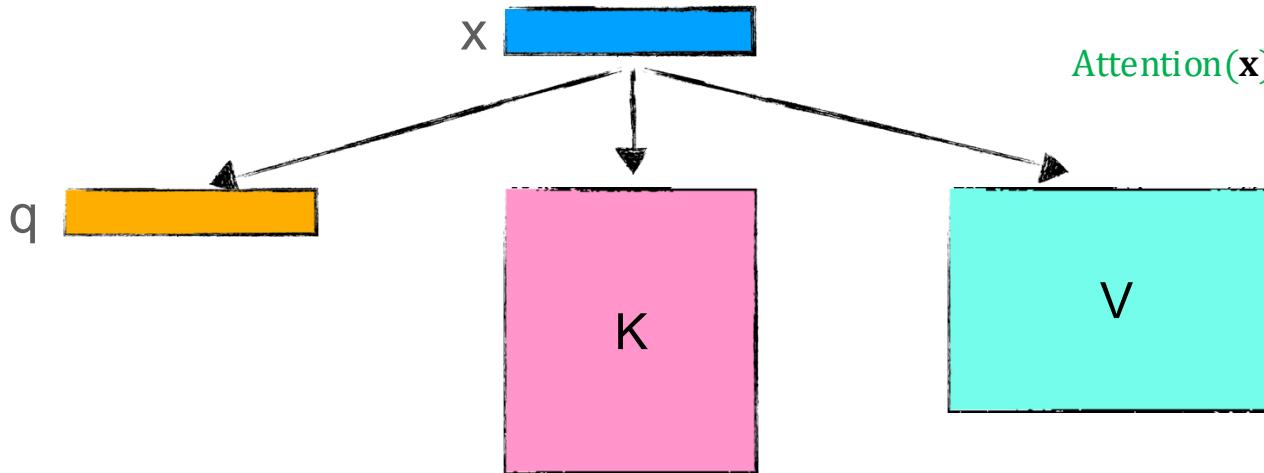
# Self-Attention During Inference

$$Q = \mathbf{x}W^q$$

$$K = \mathbf{x}W^k$$

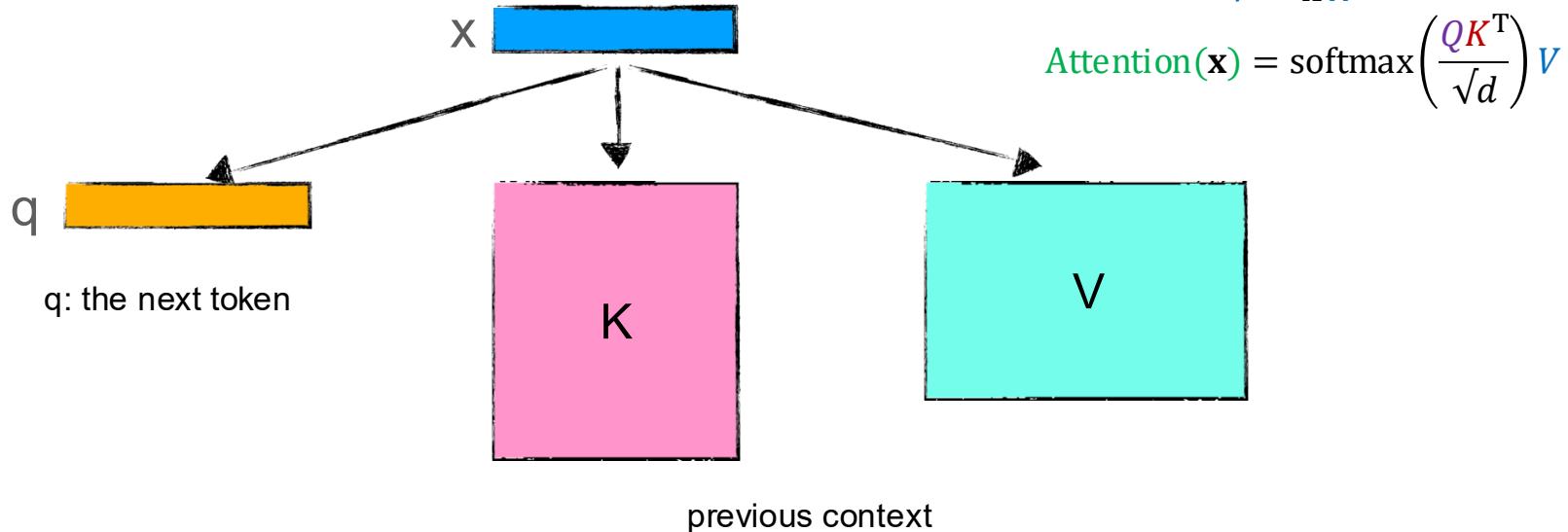
$$V = \mathbf{x}W^v$$

$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$



[Slide credit: Arman Cohan]

# Self-Attention During Inference



[Slide credit: Arman Cohan]

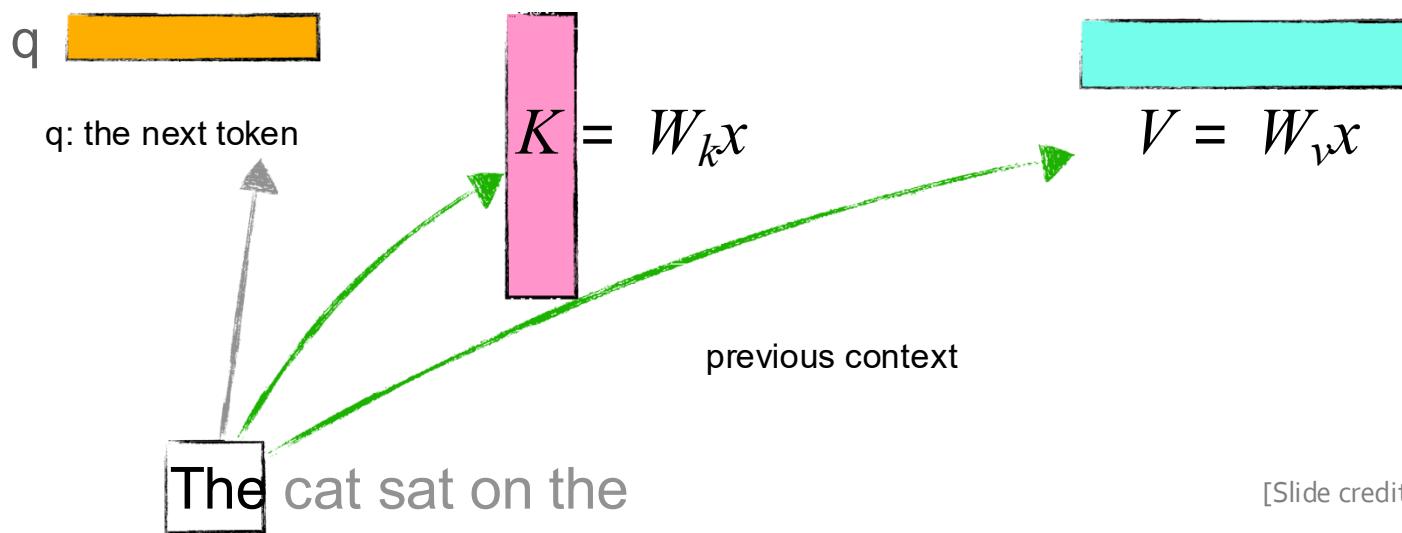
# Self-Attention During Inference

$$Q = \mathbf{x}\mathbf{W}^q$$

$$K = \mathbf{x}\mathbf{W}^k$$

$$V = \mathbf{x}\mathbf{W}^v$$

$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$



[Slide credit: Arman Cohan]

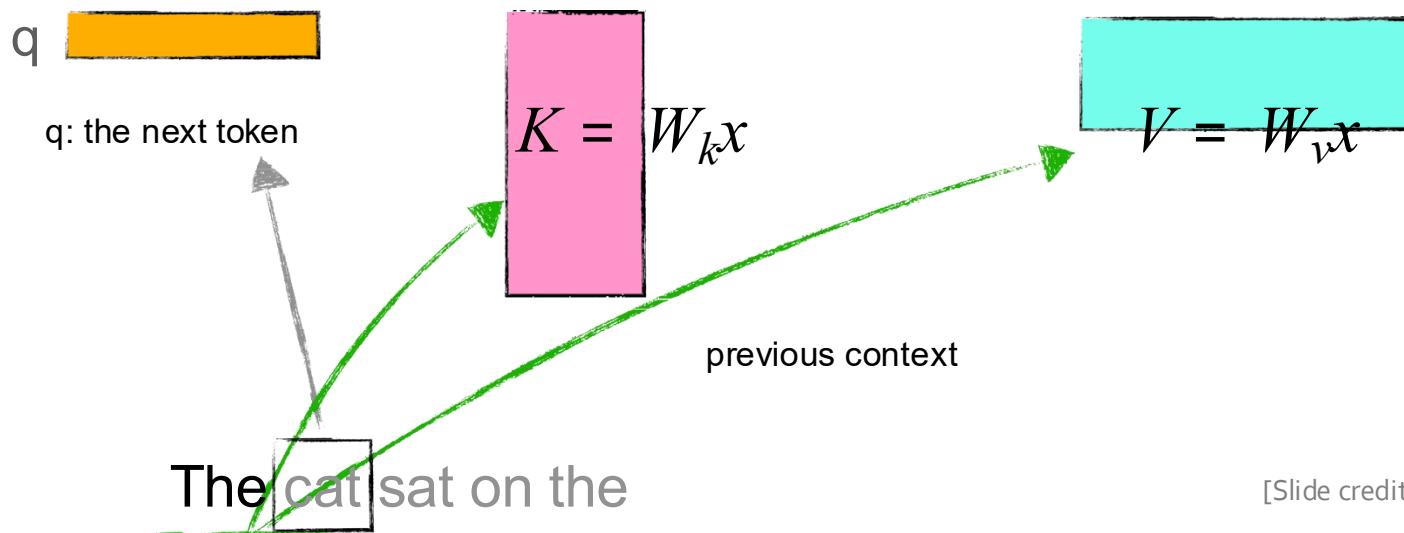
# Self-Attention During Inference

$$Q = \mathbf{x}\mathbf{W}^q$$

$$K = \mathbf{x}\mathbf{W}^k$$

$$V = \mathbf{x}\mathbf{W}^v$$

$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$



[Slide credit: Arman Cohan]

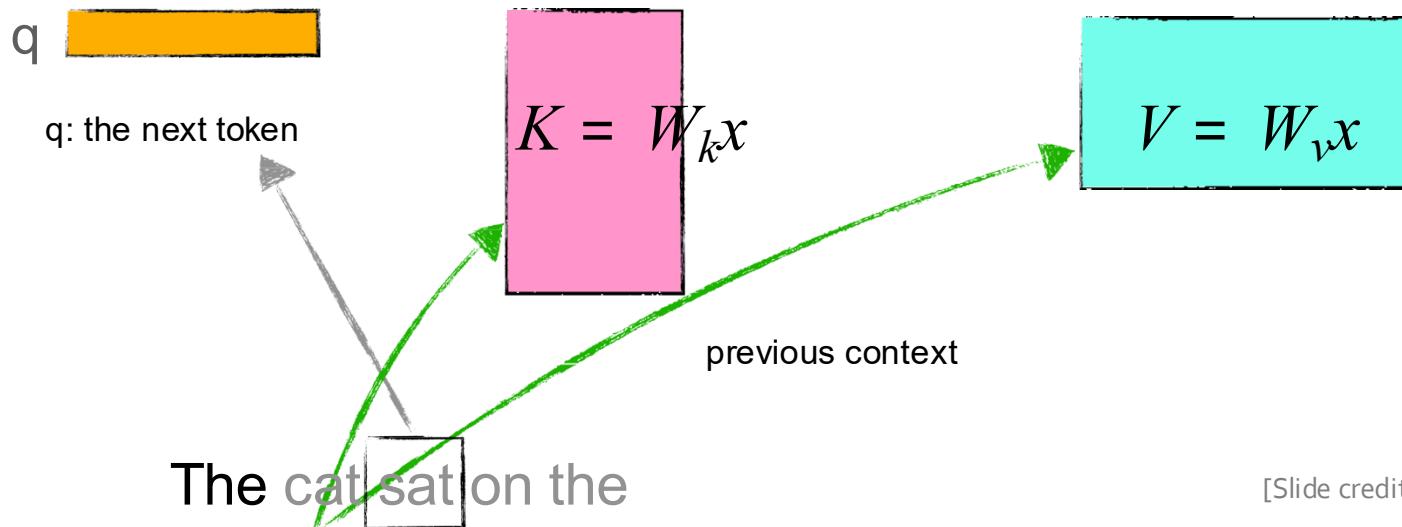
# Self-Attention During Inference

$$Q = \mathbf{x}W^q$$

$$K = \mathbf{x}W^k$$

$$V = \mathbf{x}W^v$$

$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$



[Slide credit: Arman Cohan]

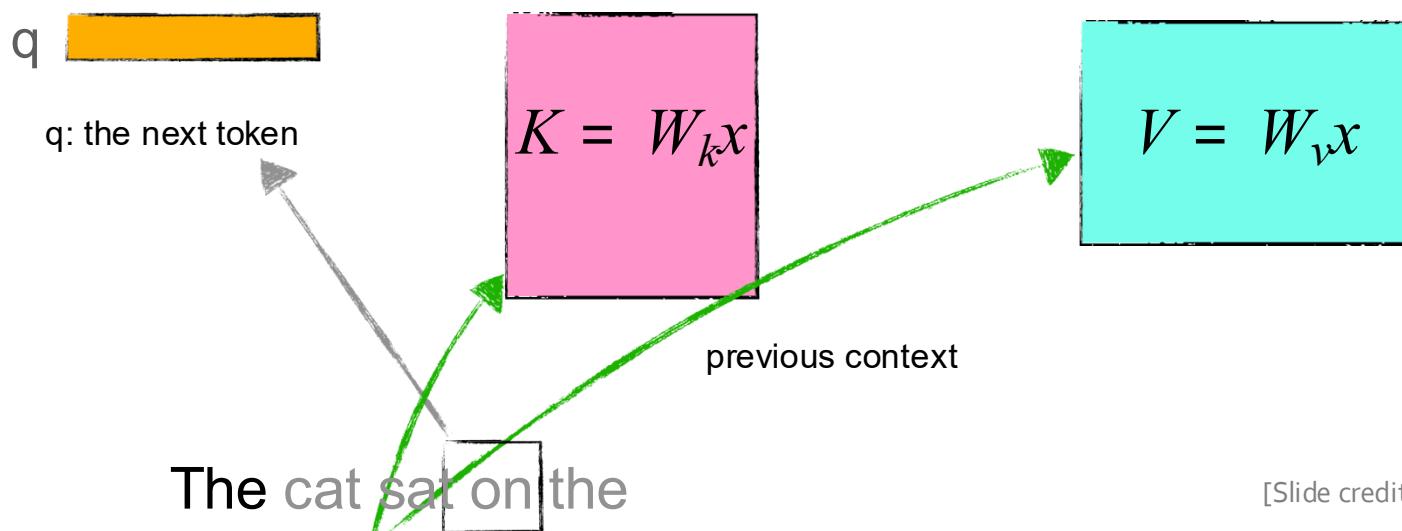
# Self-Attention During Inference

$$Q = \mathbf{x}W^q$$

$$K = \mathbf{x}W^k$$

$$V = \mathbf{x}W^v$$

$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$



[Slide credit: Arman Cohan]

# Self-Attention During Inference

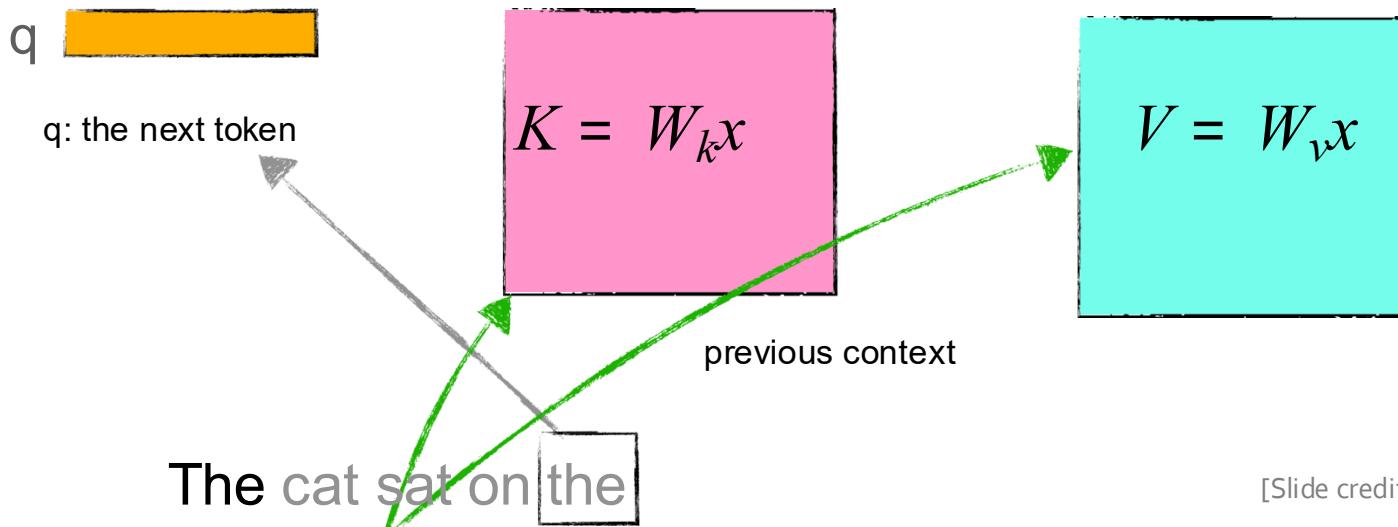
- We are computing the Keys and Values many times!
  - Let's reduce redundancy! 😊

$$Q = \mathbf{x}W^q$$

$$K = \mathbf{x}W^k$$

$$V = \mathbf{x}W^v$$

$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$



[Slide credit: Arman Cohan]

# KV-Cache for reducing inference redundancy

- We are computing the Keys and Values many times!
  - Let's reduce redundancy! 😊

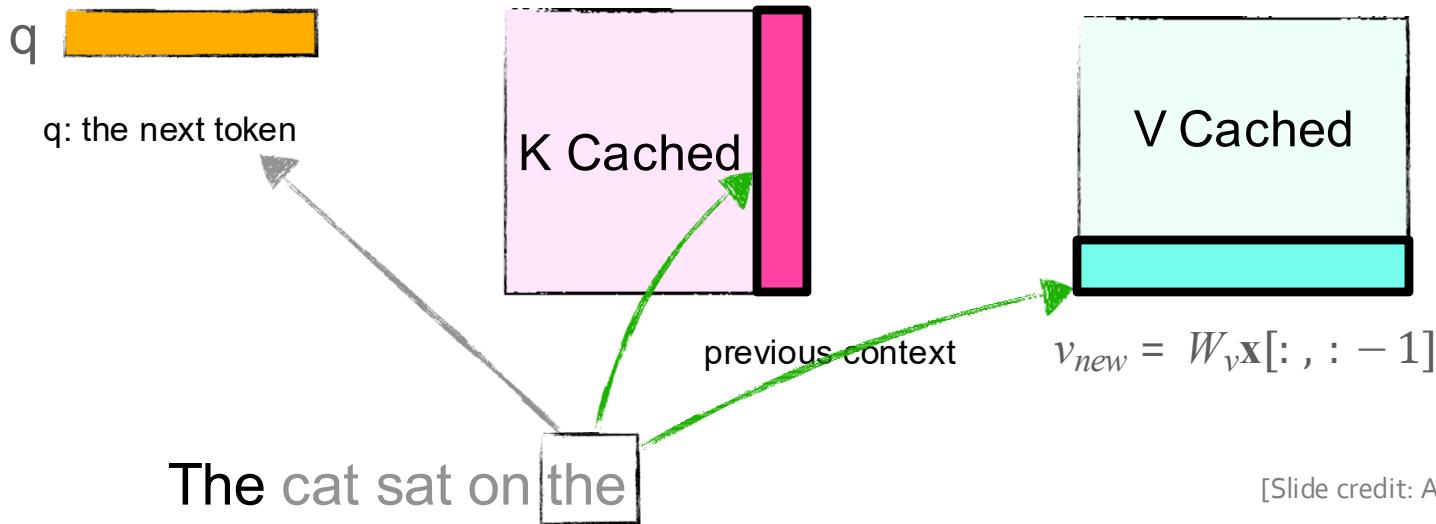
$$k_{new} = W_k \mathbf{x}[:, :, -1]$$

$$Q = \mathbf{x}W^q$$

$$K = \mathbf{x}W^k$$

$$V = \mathbf{x}W^v$$

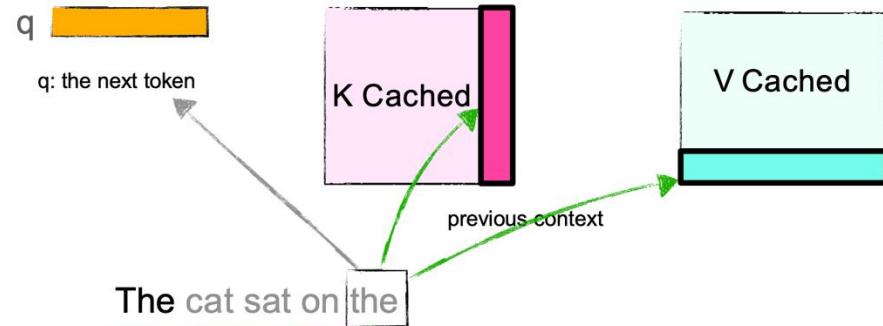
$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$



[Slide credit: Arman Cohan]

# Quiz: KV-Cache

- **Question:** How much **memory** does this KV-cache require?
    - Let's assume, embedding dimension is  $d$ , the length of the sequence seen so far is  $n$ , and your model has  $L$  layers.
1.  $2nL$
  2.  $2dL$
  3.  $ndL$
  4.  $2ndL$



# Recap

---

- To avoid **redundant computations** during decoding time, KV-cache is used to keep track of previous calculations of keys and values.
- But how exactly how costly are these computations?

# Decoding Computations

Notice we're doing this computations for one token

Compared to the previous table (SA for a seq of length  $n$ ), all the cells have one less dependence on  $n$  (e.g.,  $n^2 \rightarrow n$  or  $n \rightarrow 1$ ).

Dimensions	Operation	Computations	IO
$\mathbf{x} \in \mathbb{R}^{b \times 1 \times d}, \mathbf{W}_i^q \in \mathbb{R}^{d \times \frac{d}{m}}$	Query/key computations get combined with KV-cache	$O(bd^2)$	$O(d^2 + 2bd)$
$\mathbf{Q}_i, \mathbf{K}_i \in \mathbb{R}^{b \times 1 \times \frac{d}{m}} + \text{KV-cache}$	$P_i \leftarrow \text{softmax}\left(\frac{\mathbf{Q}_i \mathbf{K}_i^T}{\sqrt{d/m}}\right)$ for $m$ heads	$O(bnd)$	$O(bnm + bnd + bd)$
$\mathbf{V}_i \in \mathbb{R}^{b \times 1 \times \frac{d}{m}}, P_i \in \mathbb{R}^{b \times n \times 1}$	$\text{head}_i \leftarrow P_i \mathbf{V}_i$ for $m$ heads	$O(bnd)$	$O(bnm + bnd + bd)$
$\mathbf{W}^o \in \mathbb{R}^{d \times d}, \text{head}_i \in \mathbb{R}^{b \times 1 \times \frac{d}{m}}$	$Y = \text{Concat}(\text{head}_1, \dots, \text{head}_m) \mathbf{W}^o$	$O(bd^2)$	$O(2bd + d^2)$
$Y \in \mathbb{R}^{b \times 1 \times d}, \mathbf{W}_1 \in \mathbb{R}^{d \times d_{ff}}, \mathbf{W}_2 \in \mathbb{R}^{d_{ff} \times d}$	$Y = \text{ReLU}(Y \mathbf{W}_1) \mathbf{W}_2$	$O(16bd^2)$	$O(2bd + 8d^2)$
---	Total	$O(bd^2 + bnd)$	$O(bmn + bnd + d^2)$

Now the computations (of next token) has linear dependence on seq length.

# Recap: complexity of decoding text

---

- Transformers computation for generating each word is  $O(bd^2 + bnd)$ .
- Remember the complexity of processing a full sequence of length:  $O(bnd^2 + bn^2d)$ .

# Quiz: Enc-Dec inference cost

- Suppose we're using an Enc-Dec model for summarization task. Input are long passages with length  $m$  and the expected (summary length is  $n$ ). The computational complexity of this operation is:
  - $O(n) + O(m)$
  - $O(n) + O(m) + O(nm)$
  - $O(n^2) + O(m^2) + O(nm)$
  - $O(n^2) + O(m^2)$

No, self attention is all-to-all  
and so quadratic.

# Quiz: Enc-Dec inference cost

- Suppose we're using an Enc-Dec model for summarization task. Input are long passages with length  $m$  and the expected (summary length is  $n$ ). The computational complexity of this operation is:
  - $O(n) + O(m)$
  - $O(n) + O(m) + O(nm)$
  - $O(n^2) + O(m^2) + O(nm)$
  - $O(n^2) + O(m^2)$

No, self attention is all-to-all  
and so quadratic in  $m$  and  $n$ .

# Quiz: Enc-Dec inference cost

- Suppose we're using an Enc-Dec model for summarization task. Input are long passages with length  $m$  and the expected (summary length is  $n$ ). The computational complexity of this operation is:
    - $O(n) + O(m)$
    - $O(n) + O(m) + O(nm)$
    - $O(n^2) + O(m^2) + O(nm)$
    - $O(n^2) + O(m^2)$
- No, cross attention is missing.

# Quiz: Enc-Dec inference cost

- Suppose we're using an Enc-Dec model for summarization task. Input are long passages with length  $m$  and the expected (summary length is  $n$ ). The computational complexity of this operation is:
  - $O(n) + O(m)$
  - $O(n) + O(m) + O(nm)$
  - $O(n^2) + O(m^2) + O(nm)$
  - $O(n^2) + O(m^2)$

Yes. The three terms are respectively the Encoder self-attention, Decoder self-attention, and Cross attention.

Though a more accurate term is:  $O(n^2d + nd^2 + m^2d + md^2 + nmd)$  with  $d$  denoting the model embedding dimension.

# Summary: Computational Complexity of Transformers

---

- **Process a sequence at once:** Computation is bounded by  $O(n^2)$ .
- **Processing one token at a time during inference:**
  - **KV-Cache:** To avoid redundant computations during decoding time, KV-cache is used to keep track of previous calculations of keys and values.
  - The computation is bounded by  $O(n)$ .
- Though in all cases, the computations are penalizable (modulo Transformer layers).
- **IO bottlenecks:** we will get to this (next chapter).