



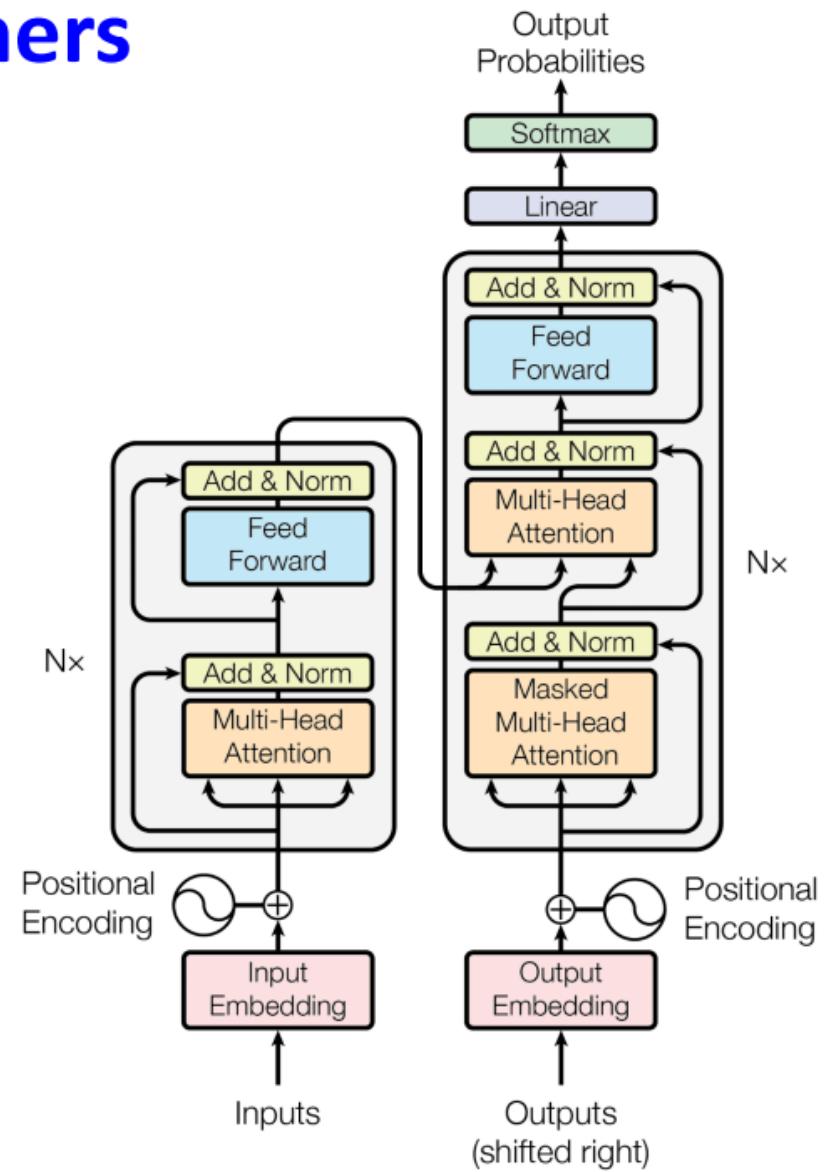
**KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY**  
AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE  
**Narayananaguda, Hyderabad.**

# **Embedded Learning TRANSFORMERS-DECODER 06-02-2025**

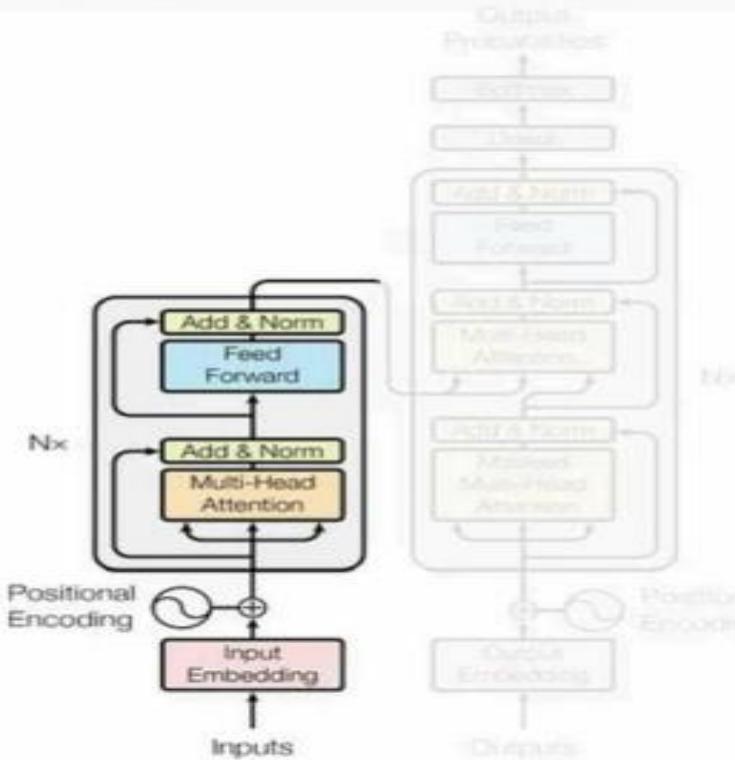
**By  
Asha.M  
Assistant professor  
CSE(AI&ML)  
KMIT**

# Transformers

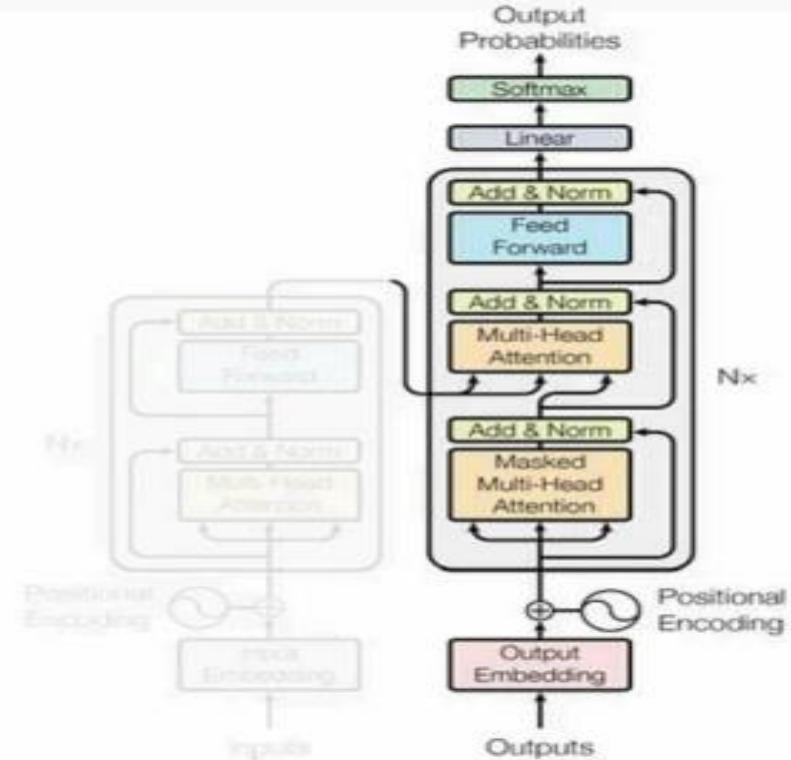
- Tokenization
- Input Embeddings
- Position Encodings
- Residuals
- Query
- Key
- Value
- Add & Norm
- Encoder
- Decoder
- Attention
- Self Attention
- Multi Head Attention
- Masked Attention
- Encoder Decoder Attention
- Output Probabilities / Logits
- Softmax
- Encoder-Decoder models
- Decoder only models



## What we have covered so far



## What we have to cover



Upcoming steps illustration



# Transformer Encoder Block Overview

The **Encoder** is the first part of the Transformer model and is responsible for processing the input sequence and generating a **context-aware representation** that the decoder can use.

Each **Encoder Block** consists of the following two main sub-layers:

1. Multi-Head Self-Attention
2. Feed-Forward Network (FFN)

Each of these layers is followed by **Layer Normalization** and **Residual Connections**.

The encoder does **NOT** have a look-ahead mask because it processes all tokens in **parallel** without restrictions.

## ◆ Input to the Encoder

The encoder takes a **source sequence** ( $x$ ) as input (e.g., a sentence in English) and converts it into a meaningful representation.

### 🚀 Step 1: Input Embedding + Positional Encoding

1. Each token in  $x$  is first converted into a dense vector using an **embedding layer** (like Word2Vec or learned embeddings).
2. Since Transformers do not have recurrence (like RNNs), we add **positional encodings** to retain order information:
  - A **sine-cosine function** is used to generate unique positional vectors for each token.

This results in an **embedded representation of the input**, ready for processing.

## ● 1st Block: Multi-Head Self-Attention

### 📌 Purpose

- Allows the encoder to **attend to all words in the input sequence**, capturing long-range dependencies.

### ⚙️ How it Works?

1. The input ( $x$ ) is passed to a **Self-Attention** mechanism.
2. **Each token attends to all other tokens** in the sequence.
3. **Multi-Head Attention** is used to capture multiple relationships **simultaneously**.

### 🚀 Output of this Block

- A refined **representation of the input sequence**, where each token has context from all other tokens.

## ● 2nd Block: Feed-Forward Network (FFN)

### ➤ Purpose

- Applies **non-linearity** and **transforms attention output** before passing it to the next encoder layer.

### ⚙ How it Works?

1. The self-attention output is passed through **two fully connected layers**:
  - **First layer (Expansion Layer)**: Expands the dimension ( $d_{model} \rightarrow d_{ff}$ ).
  - **ReLU Activation** for non-linearity.
  - **Second layer (Compression Layer)**: Compresses back ( $d_{ff} \rightarrow d_{model}$ ).
2. Helps the model learn **complex representations**.

### 🚀 Output of this Block

- The transformed features are **sent to the next encoder layer**.

## ◆ Additional Components

### ◆ Residual Connections

- Each sub-layer (Self-Attention, FFN) has a **skip connection**:
- Helps in better gradient flow and prevents **vanishing gradients**.

### ◆ Layer Normalization

- Applied after each sub-layer to **stabilize training**.



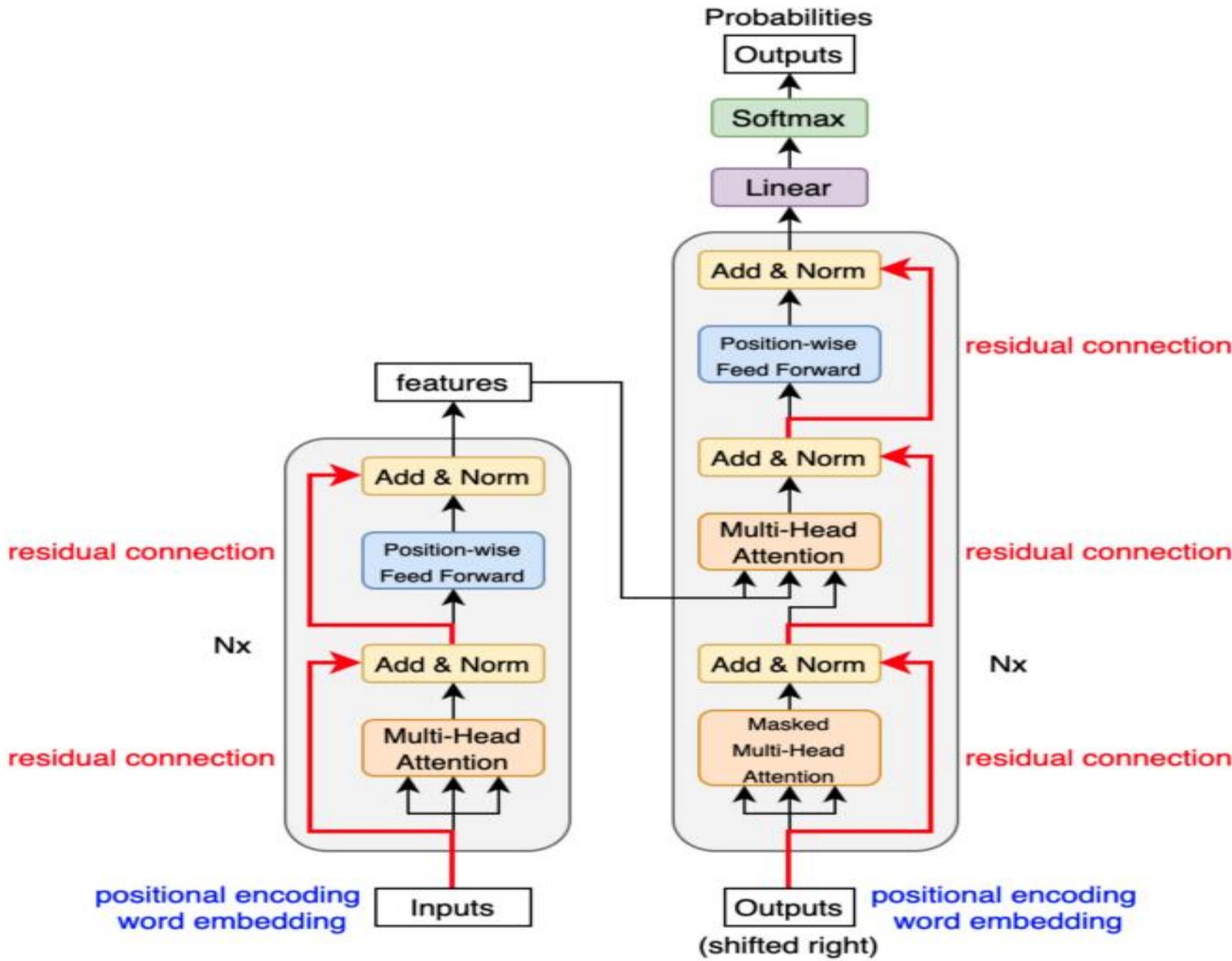
# Final Output of the Encoder

- The encoder block is **stacked N times** (e.g.,  $N = 6$  in the original Transformer).
- The final encoder output ( $z$ ) contains a contextualized representation of the source sequence.
- This output is then passed to the decoder for further processing.

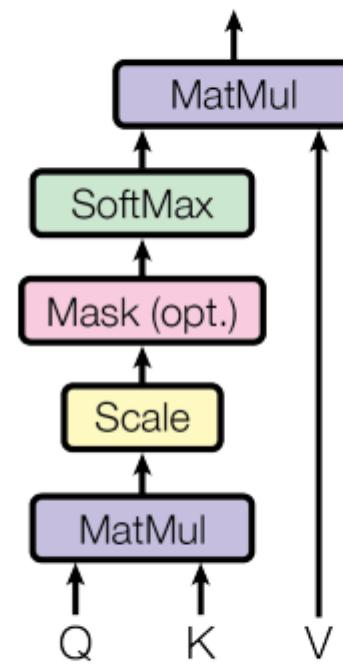


# Summary of Each Block

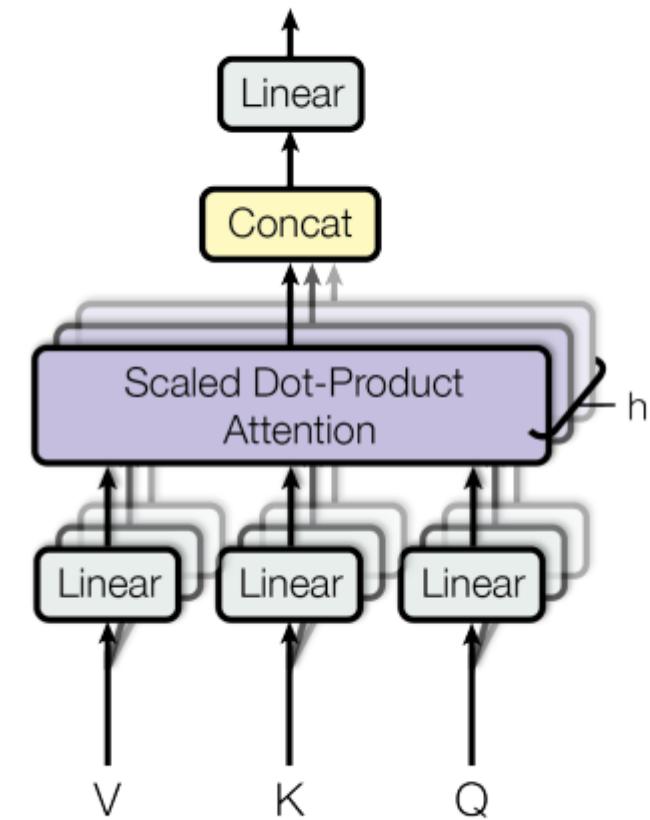
Block	Purpose	Input	Output
Multi-Head Self-Attention	Allows each token to attend to all tokens in the input	Source sequence ( $x$ )	Contextualized representation
Feed-Forward Network (FFN)	Non-linearity and transformation	Self-attention output	Transformed features



## Scaled Dot-Product Attention

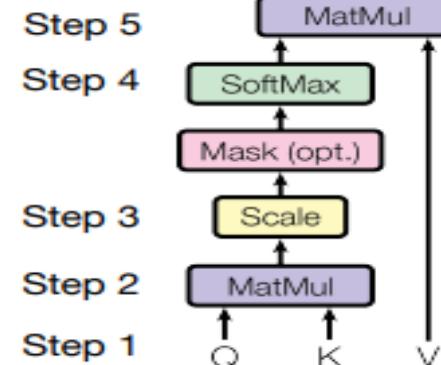


## Multi-Head Attention



# Scaled Dot-Product Attention Recap

## Scaled Dot-Product Attention



Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L. and Polosukhin, I., 2017. Attention Is All You Need.

**Step 2:**

$$Q \times K^T = QK^T$$

key of 1st word,  $k_1$

relationship between 1st & 2nd word

**Step 3:**

$$QK^T / \sqrt{d_k} = \frac{QK^T}{\sqrt{d_k}}$$

embedding of 1st word,  $x_1$

## Step 1:

$$x \times W^q = Q$$

query of 1st word,  $q_1$

$$x \times W^k = K$$

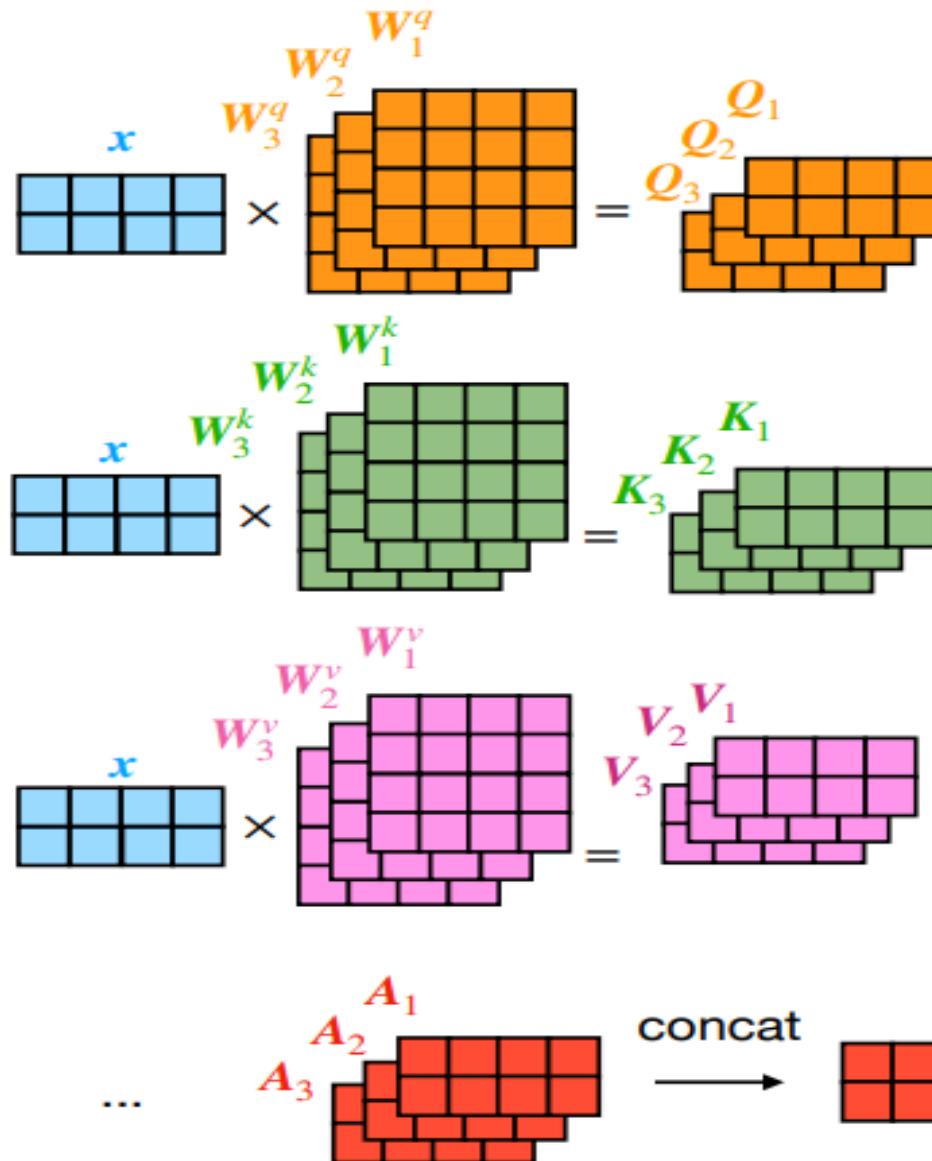
$$x \times W^v = V$$

**Step 4:** softmax  $(QK^T / \sqrt{d_k}) =$

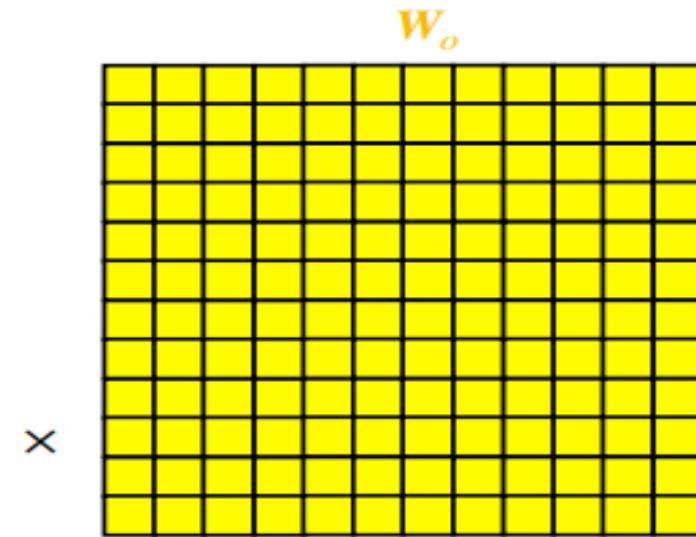
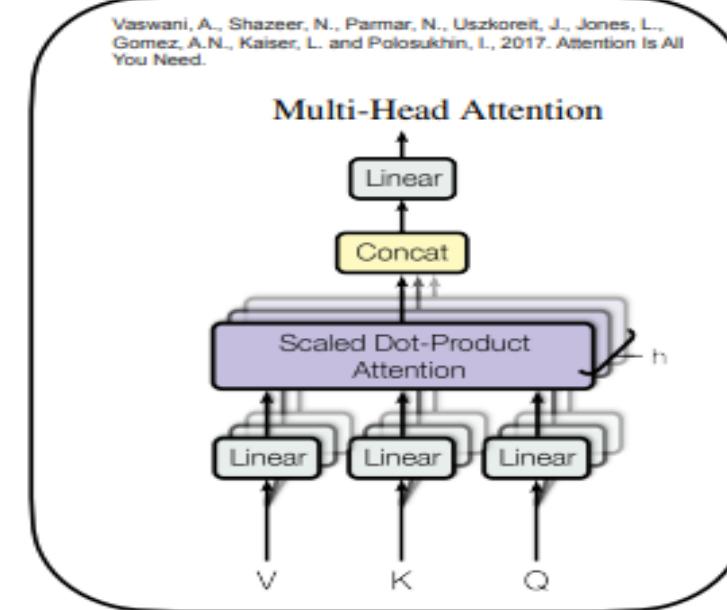
1st value of 1st & 2nd word

softmax  $(QK^T / \sqrt{d_k}) \times V = A$

attentions of 1st word with first value of words



Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L. and Polosukhin, I., 2017. Attention Is All You Need.



# Multilayer Perceptrons

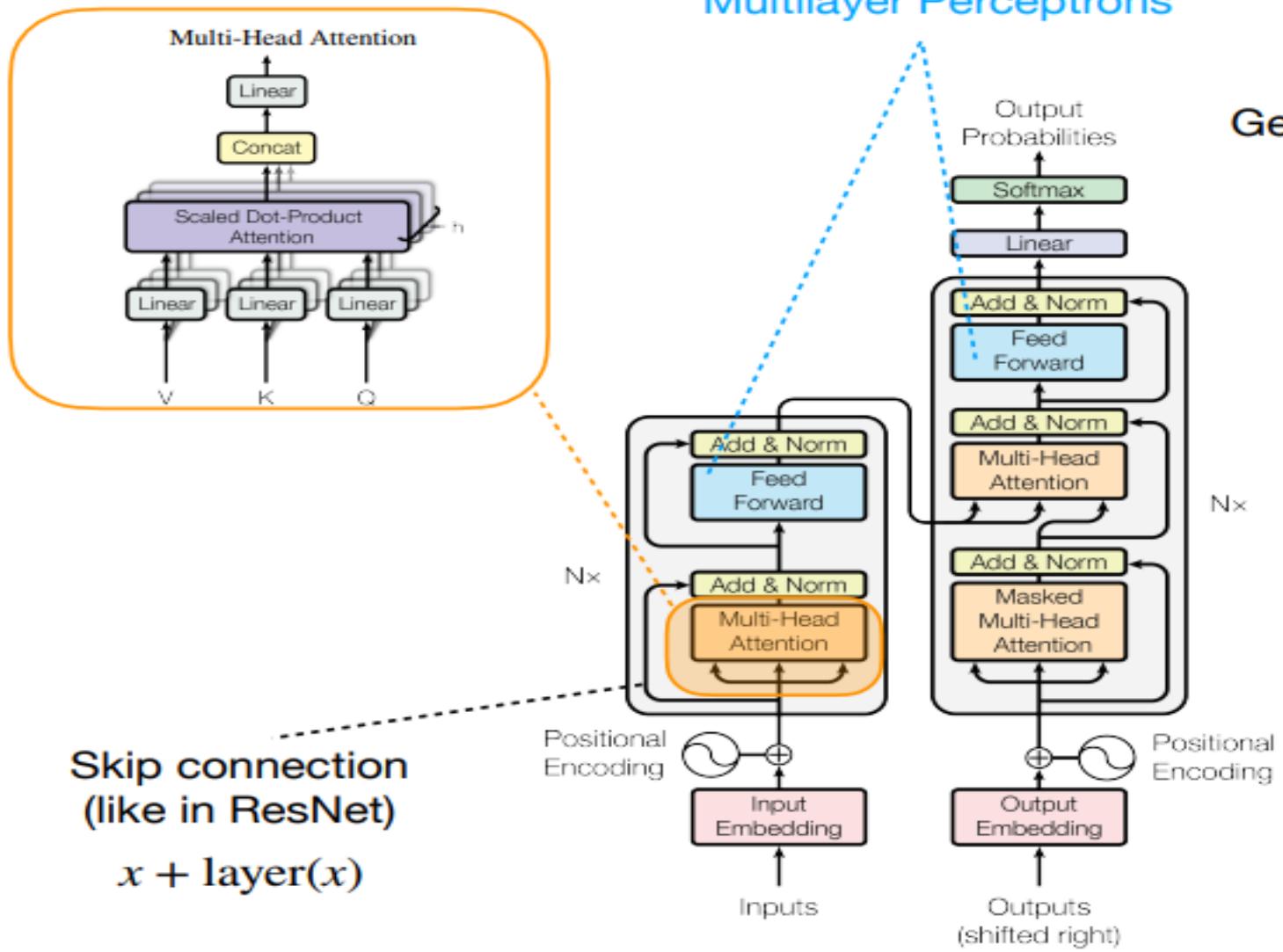


Figure 1: The Transformer - model architecture.

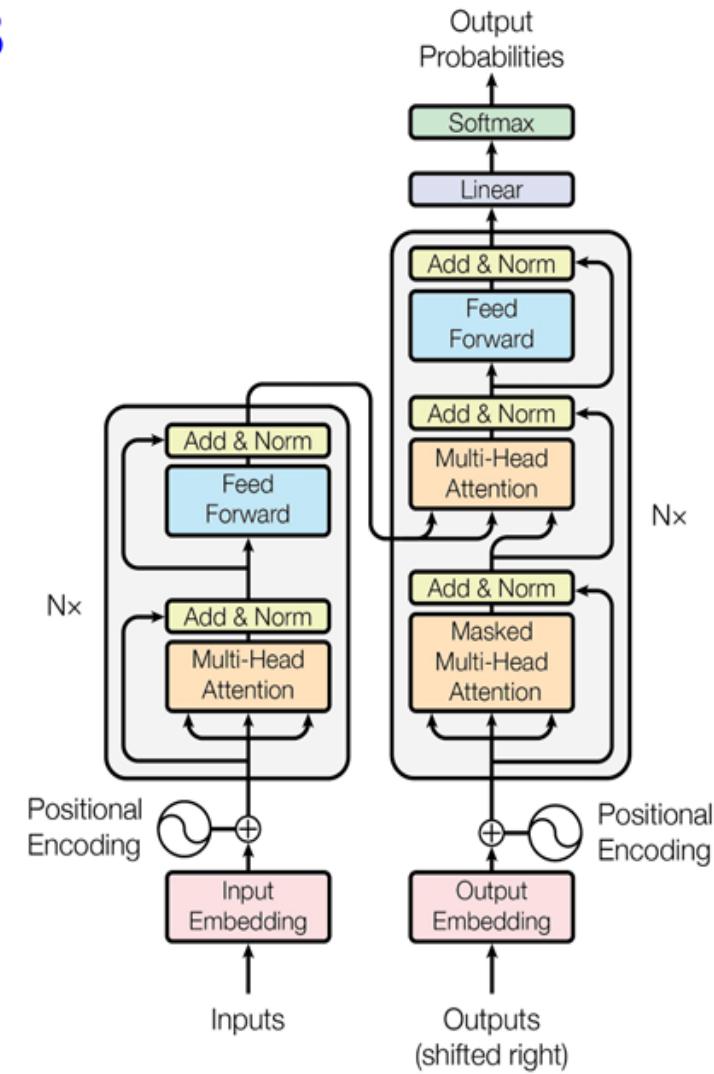
Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L. and Polosukhin, I., 2017. Attention Is All You Need.

# Language translation example

# Inputs

## Processing Inputs

Inputs  
I ate an apple

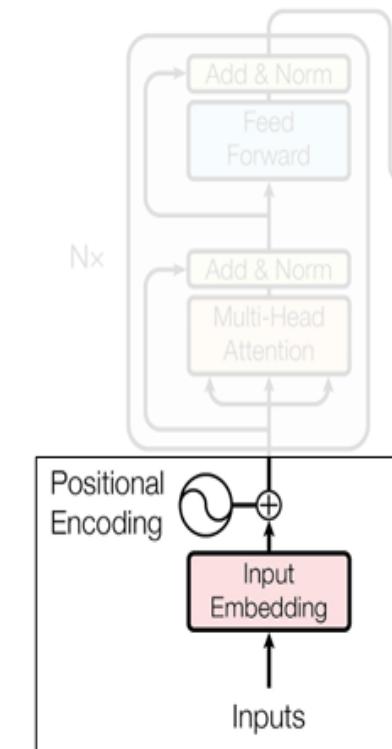


# Tokenization

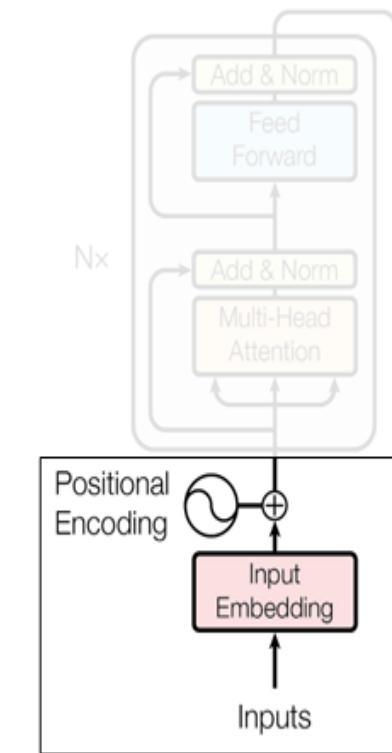
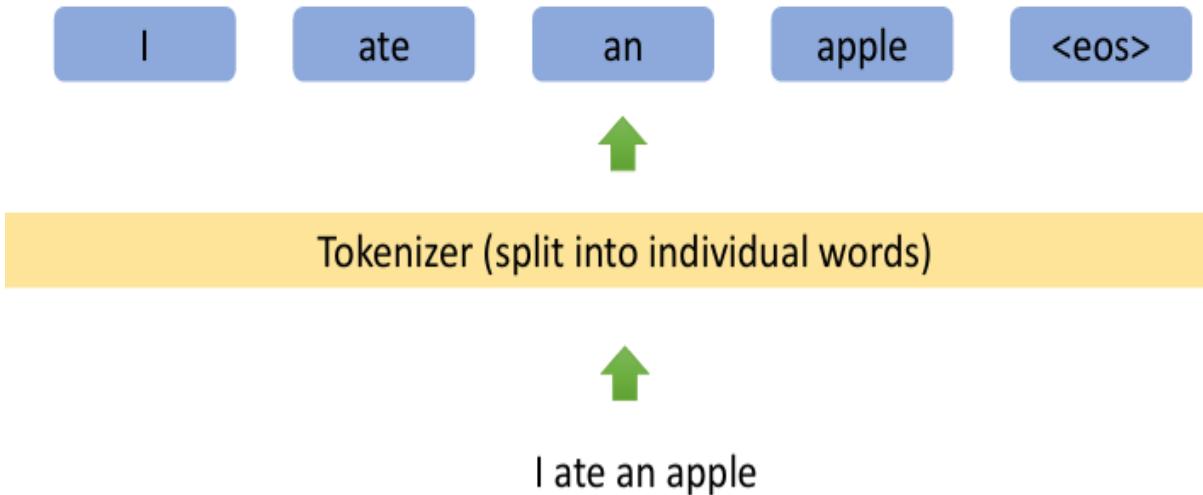
Tokenizer (split into individual words)



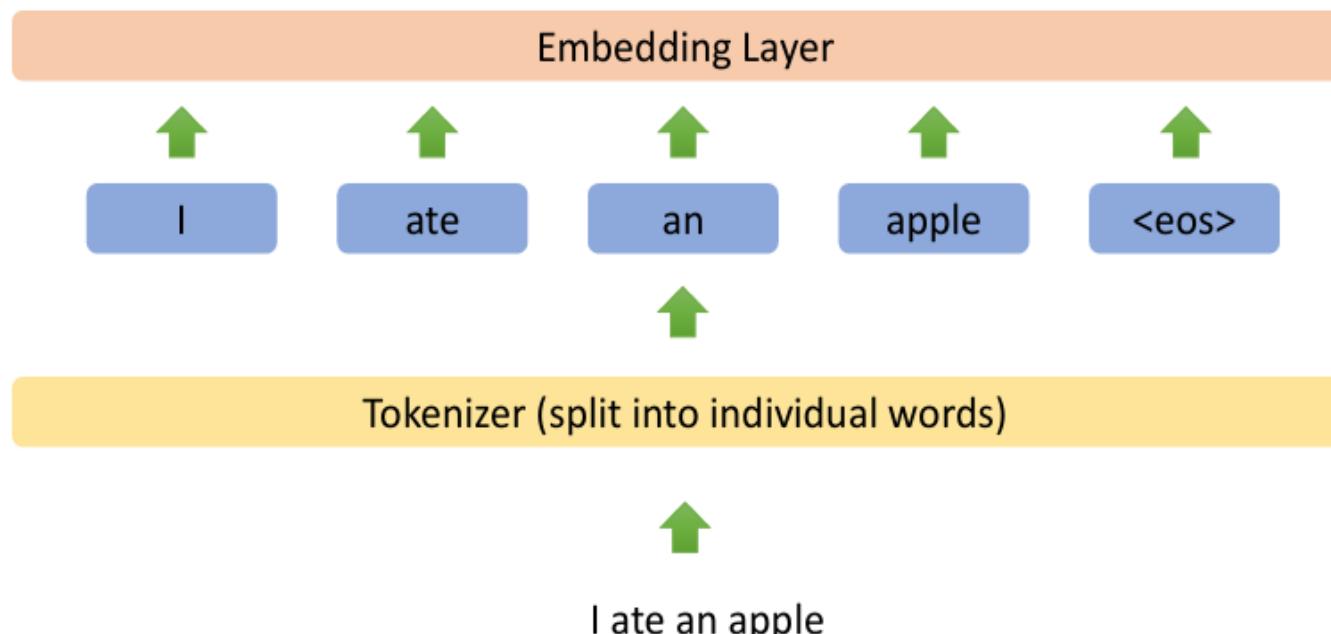
I ate an apple



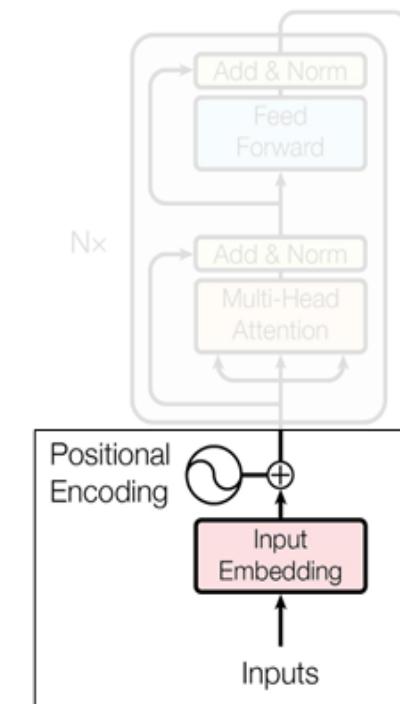
# Tokenization



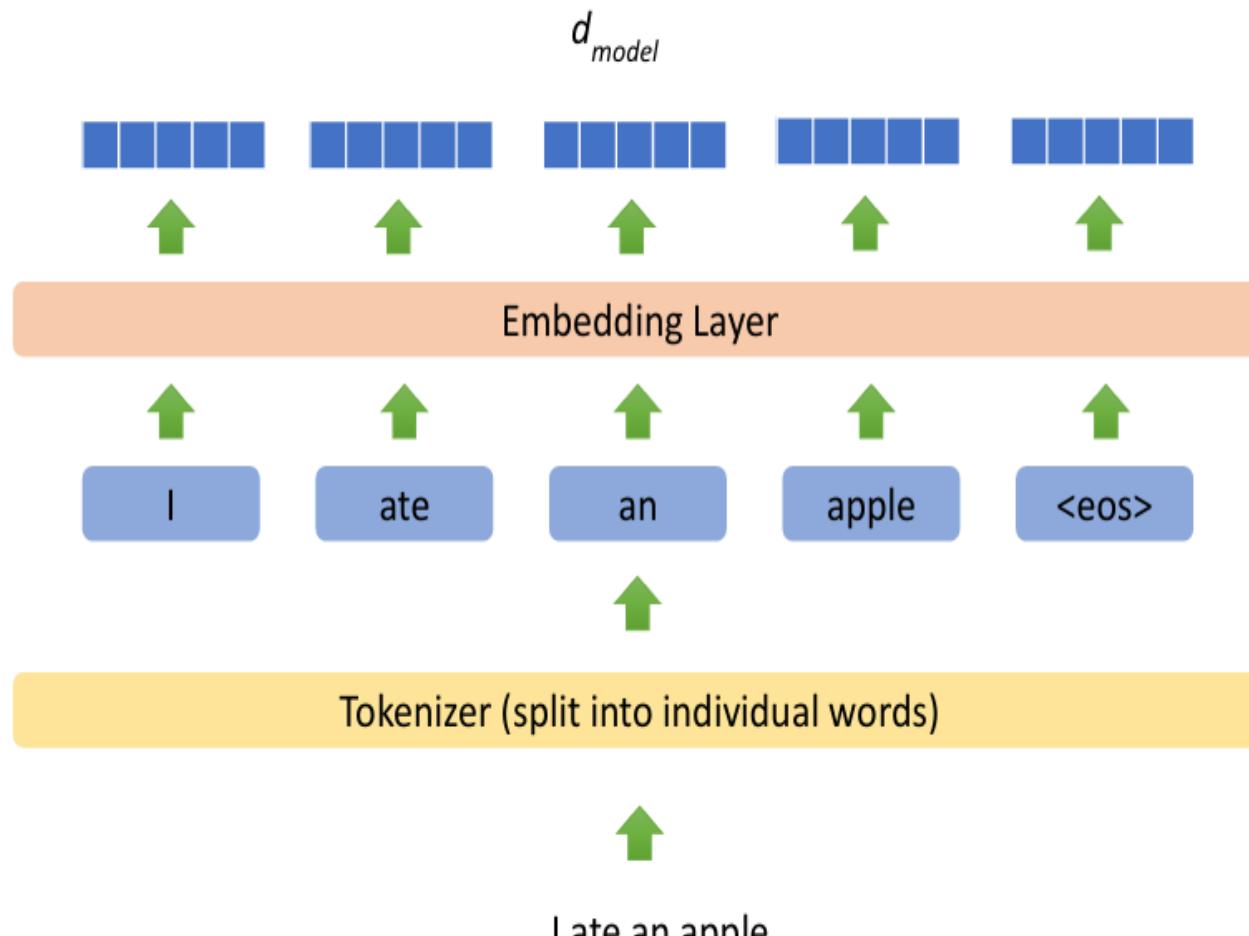
# Input Embeddings



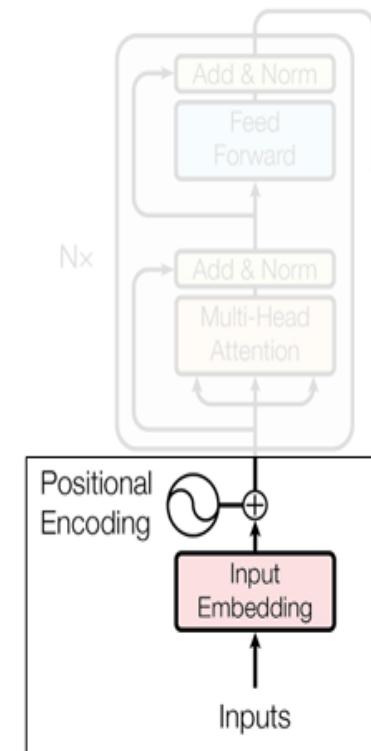
Generate Input Embeddings



# Input Embeddings

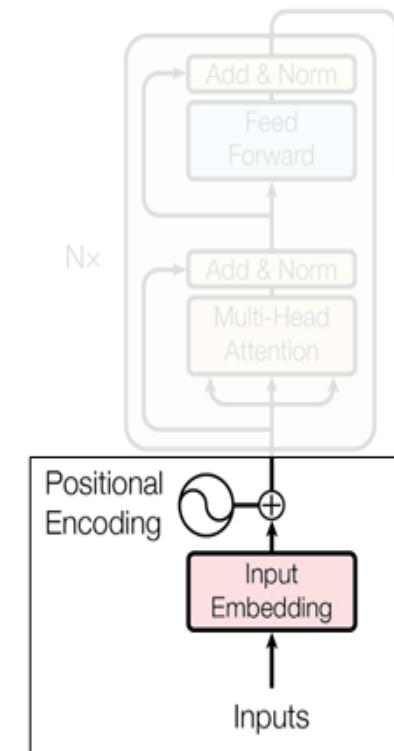


Generate Input Embeddings



# Position Encodings

I ate an apple <eos>

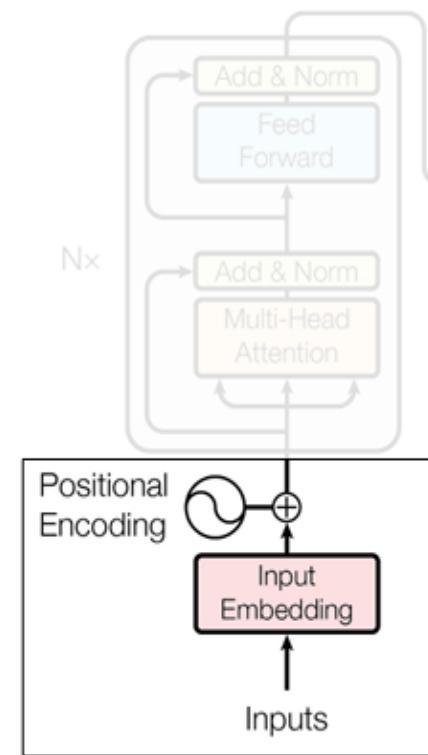


# Position Encodings

I ate an apple <eos>

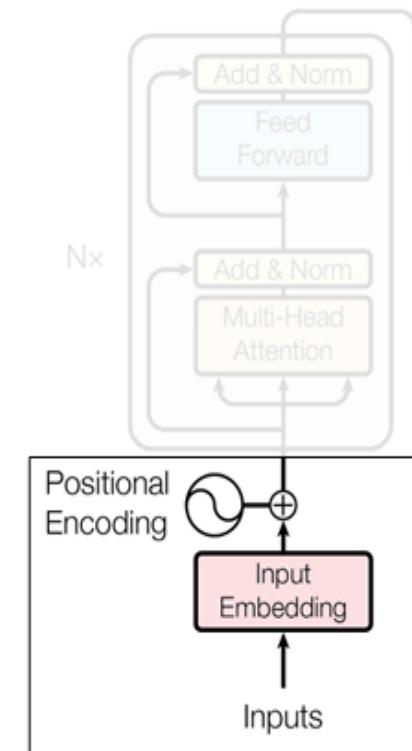


apple ate an I <eos>



# Position Encodings

Requirements for Positional Encodings???



# Position Encoding

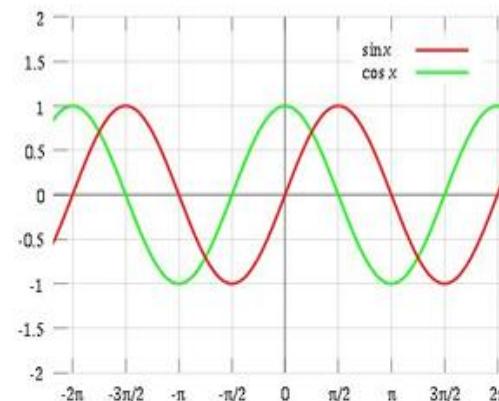
## Requirements for Position Encodings

- Some representation of time? (like seq2seq?)
- Should be unique for each position
- Bounded

## Actual Candidates

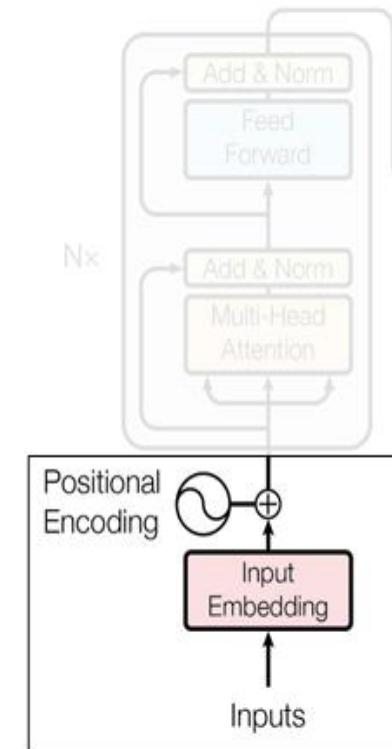
$\sin(g(t))$

$\cos(g(t))$



## Requirements for $g(t)$

- Must have same dimensions as input embeddings
- Must produce overall unique encodings



# Position Encoding

For each position, an embedded input is moved the same distance but at a different angle. **Inputs that are close to each other in the sequence have similar perturbations, but inputs that are far apart are perturbed in different directions.**

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

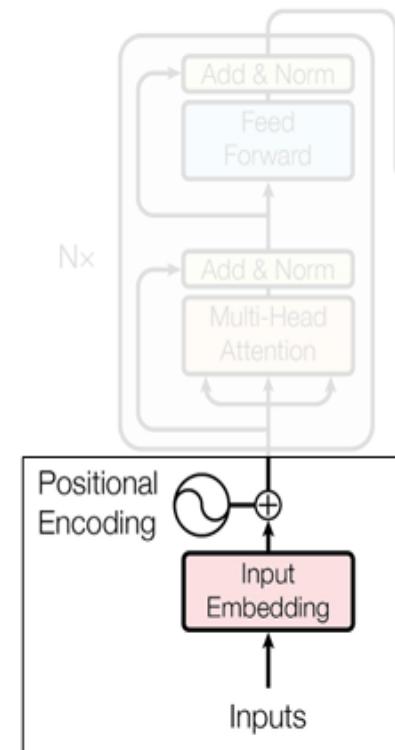
$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

**pos** -> idx of the token in input sentence

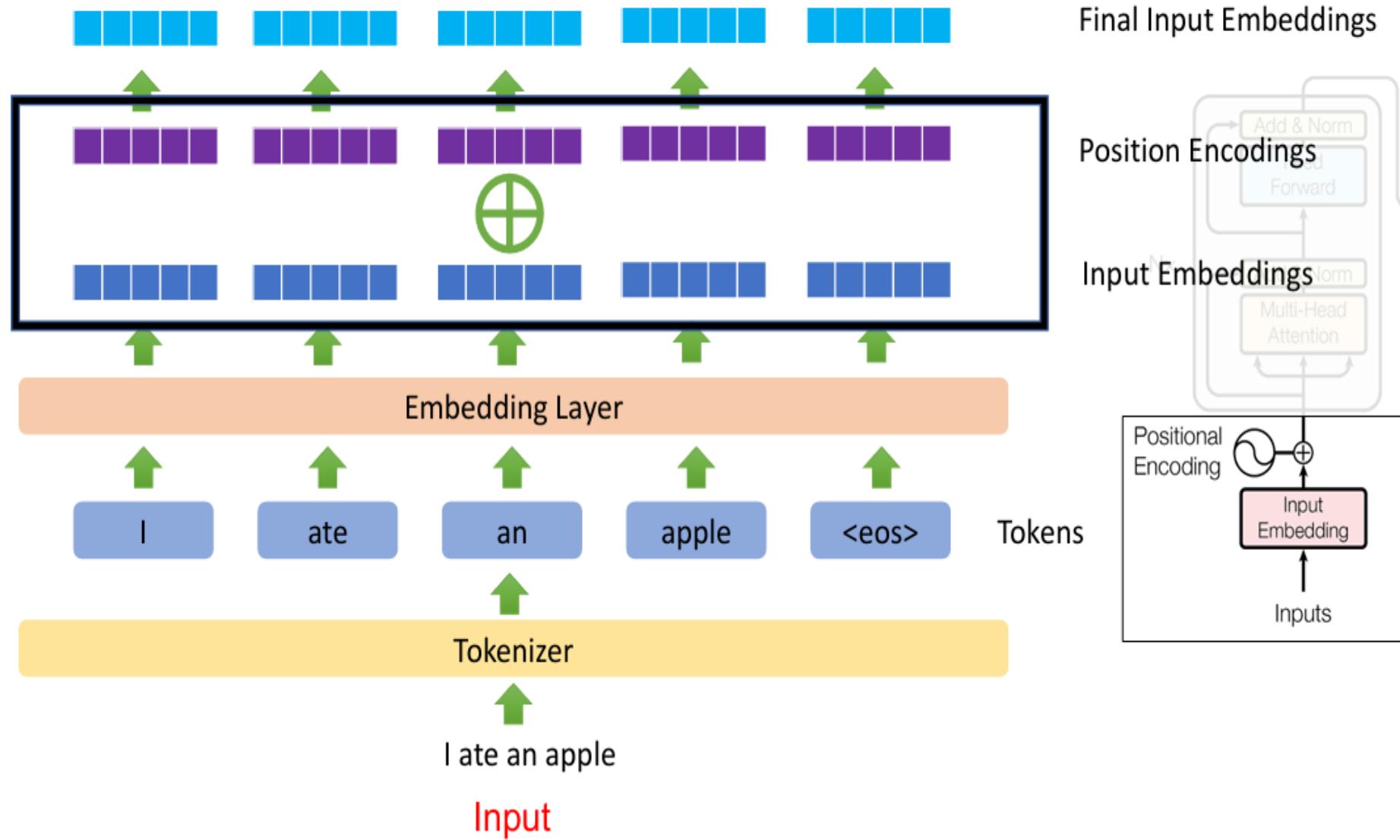
**i** -> i<sup>th</sup> dimension out of d

**d model** -> embedding dimension of each token

Different calculations for odd and even embedding indices



# Position Encoding



# Transformers

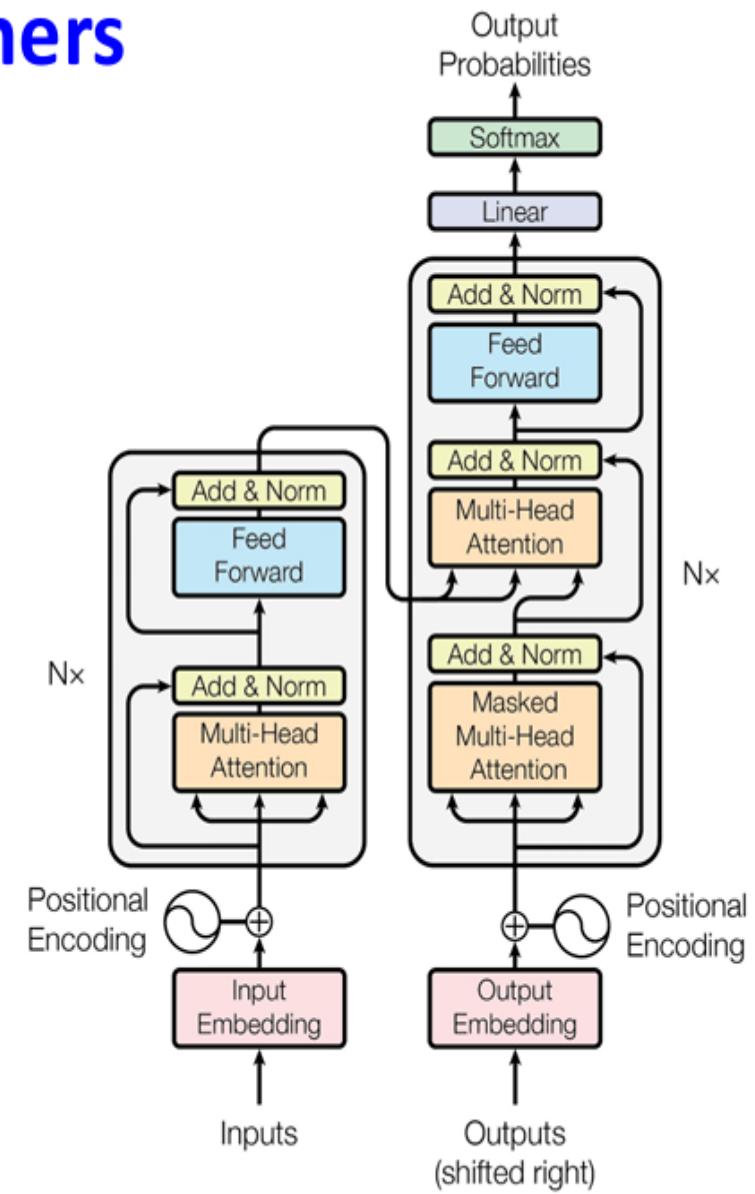
## ✓ Tokenization

## ✓ Input Embeddings

## ✓ Position Encodings

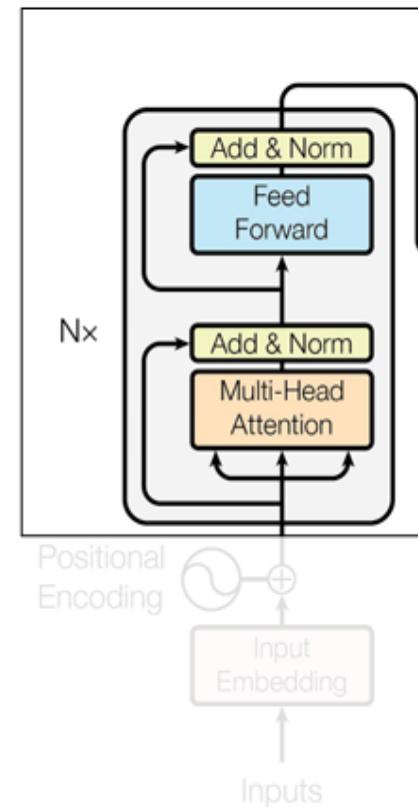
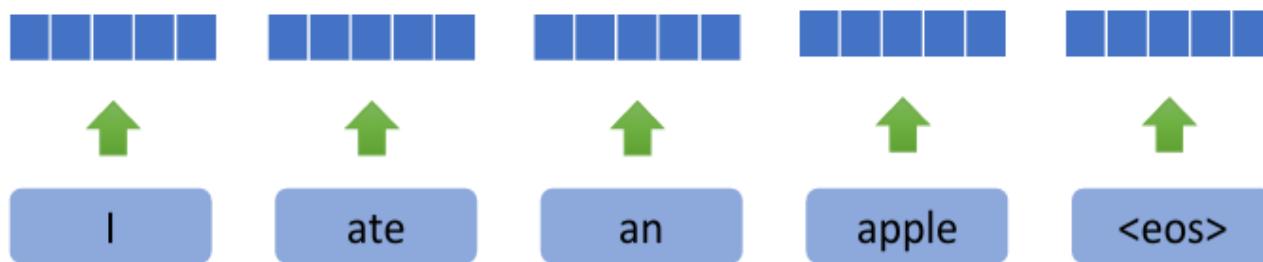
- Query, Key, & Value
- Attention
- Self Attention
- Multi-Head Attention
- Feed Forward
- Add & Norm
- Encoders

- Masked Attention
- Encoder Decoder Attention
- Linear
- Softmax
- Decoders
- Encoder-Decoder Models

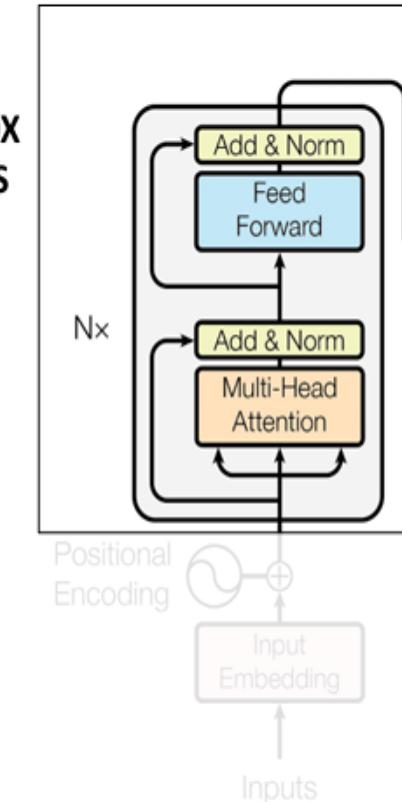
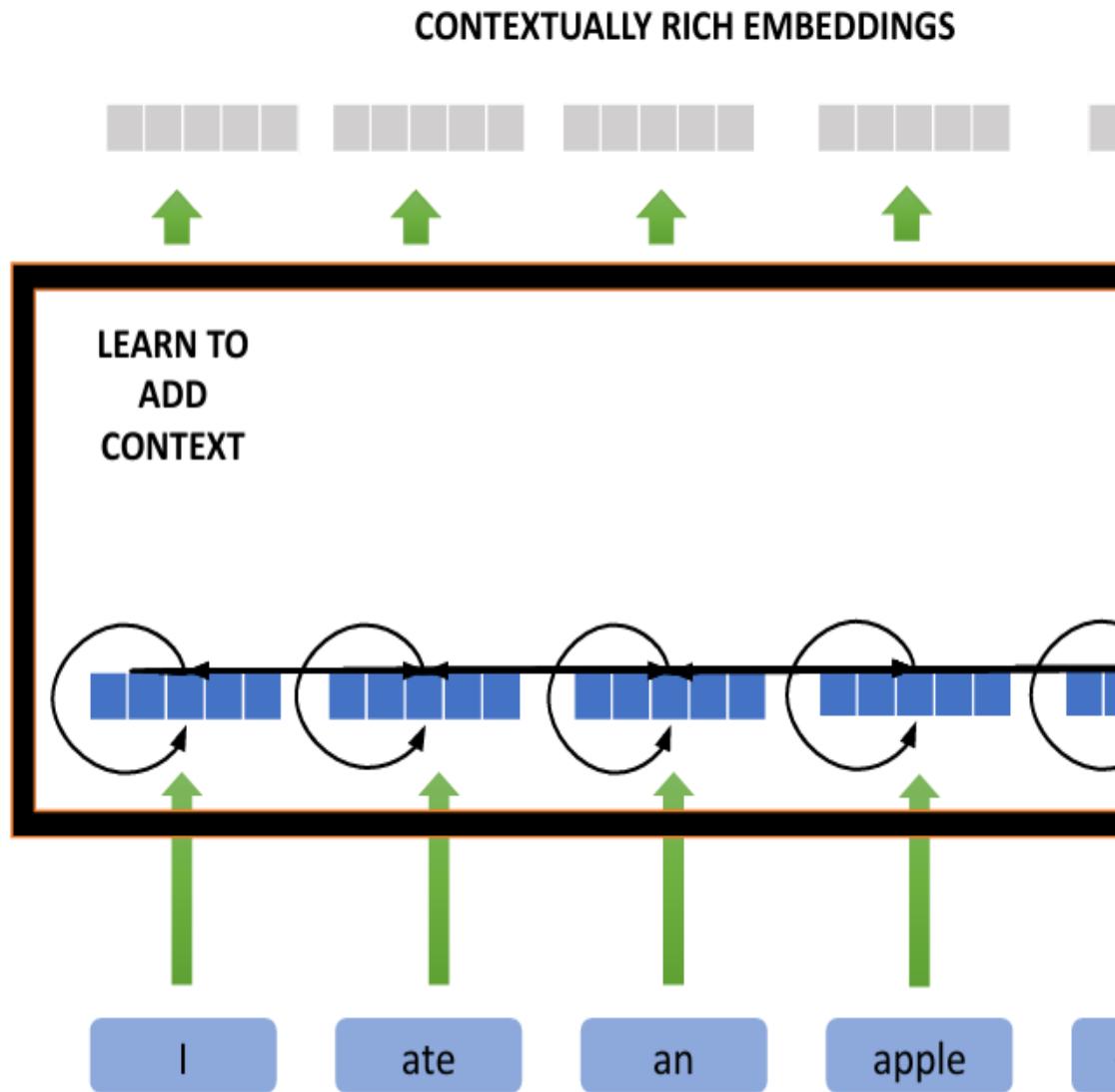


# Encoder

WHERE IS THE  
CONTEXT ?

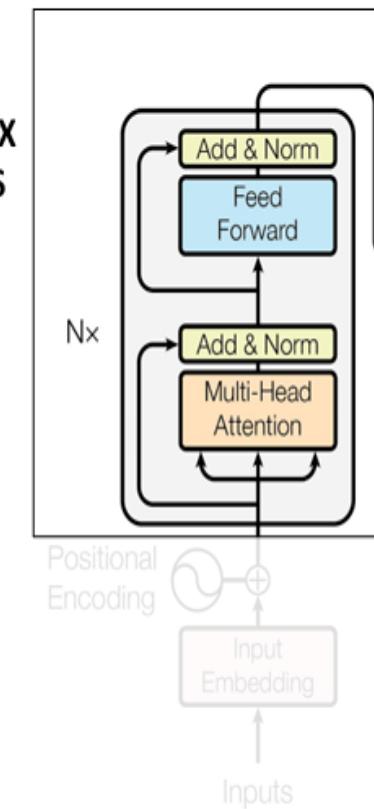
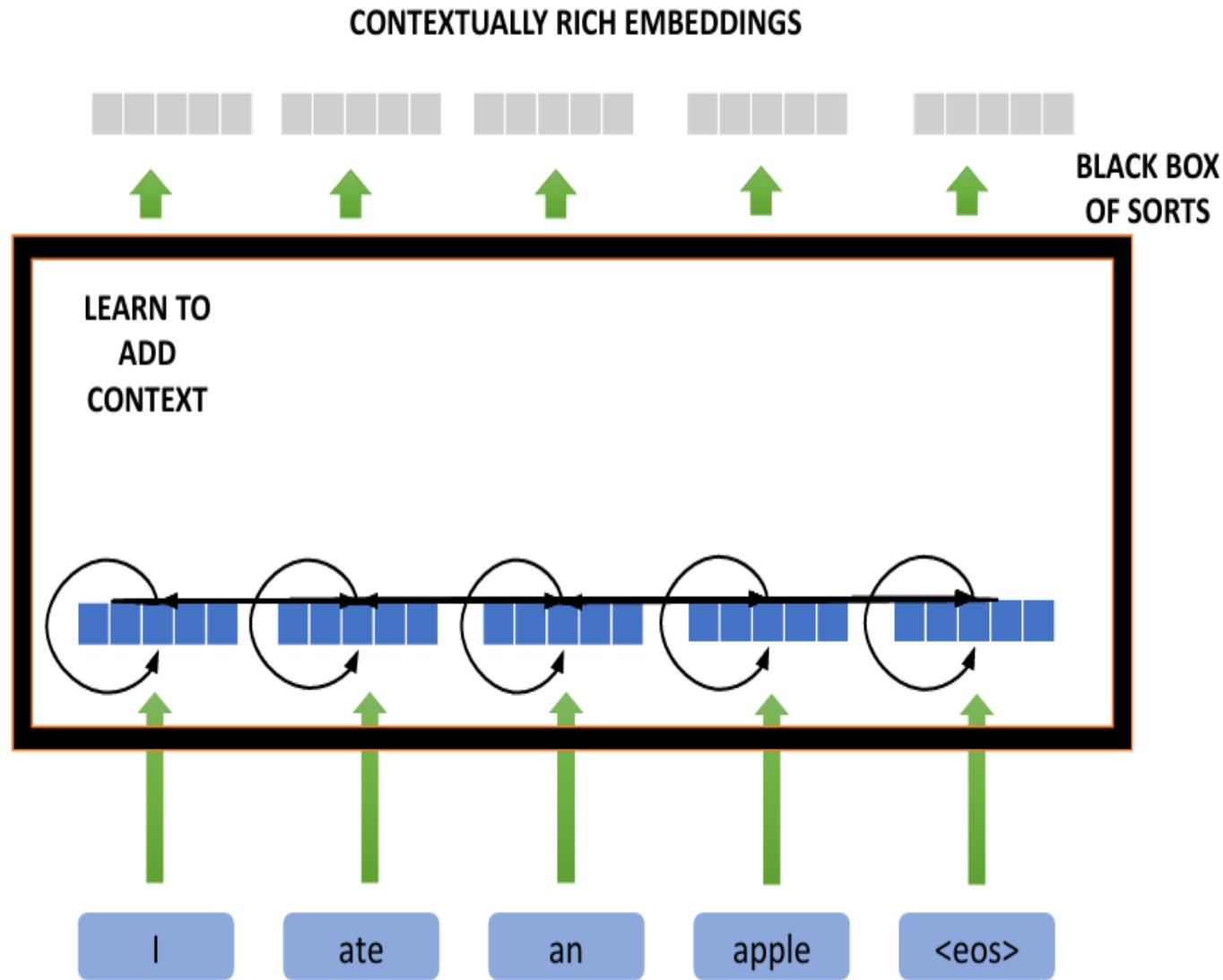


# Encoder



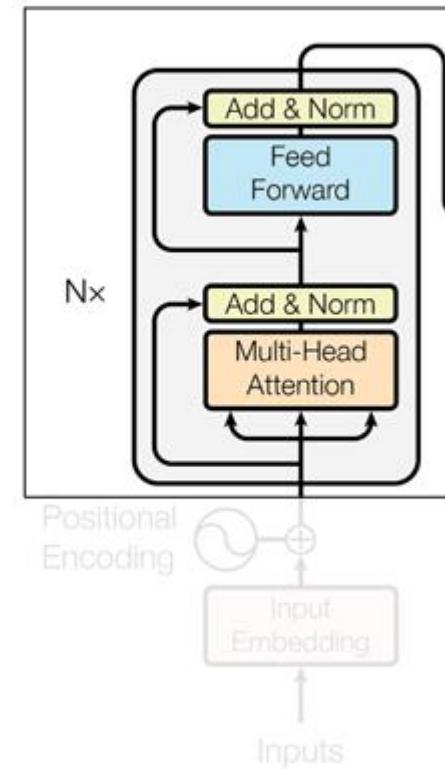
# Encoder

$\alpha_{[ij]}$  ?



$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

- Query
- Key
- Value



# Query, Key & Value

{Query: "Order details of order\_104"}  
OR  
{Query: "Order details of order\_106"}

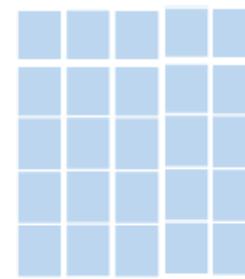
```
{"order_100": {"items": "a1", "delivery_date": "a2", ...}},  
 {"order_101": {"items": "b1", "delivery_date": "b2", ...}},  
 {"order_102": {"items": "c1", "delivery_date": "c2", ...}},  
 {"order_103": {"items": "d1", "delivery_date": "d2", ...}},  
 {"order_104": {"items": "e1", "delivery_date": "e2", ...}},  
 {"order_105": {"items": "f1", "delivery_date": "f2", ...}},  
 {"order_106": {"items": "g1", "delivery_date": "g2", ...}},  
 {"order_107": {"items": "h1", "delivery_date": "h2", ...}},  
 {"order_108": {"items": "i1", "delivery_date": "i2", ...}},  
 {"order_109": {"items": "j1", "delivery_date": "j2", ...}},  
 {"order_110": {"items": "k1", "delivery_date": "k2", ...}}
```

Query	Key	Value
1. Search for info	1. Interacts directly with Queries 2. Distinguishes one object from another 3. Identify which object is the most relevant and by how much	1. Actual details of the object 2. More fine grained

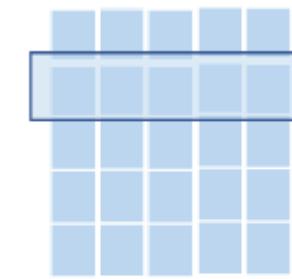
# Attention



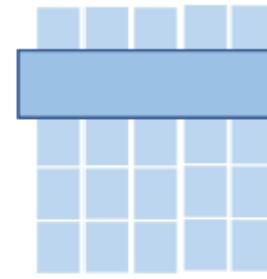
Query



Key Value  
Store



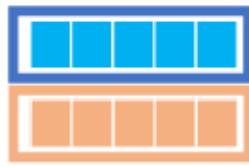
Key



Value

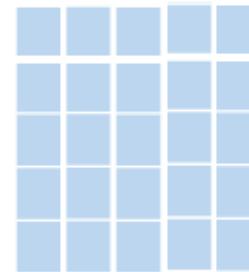
# Attention

Parallelizable !!!



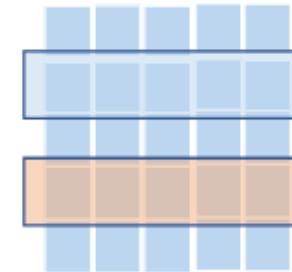
Query

$Q$



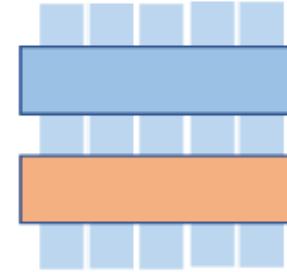
Key Value  
Store

$QK^T$



Key

$softmax(\frac{QK^T}{\sqrt{d}})$

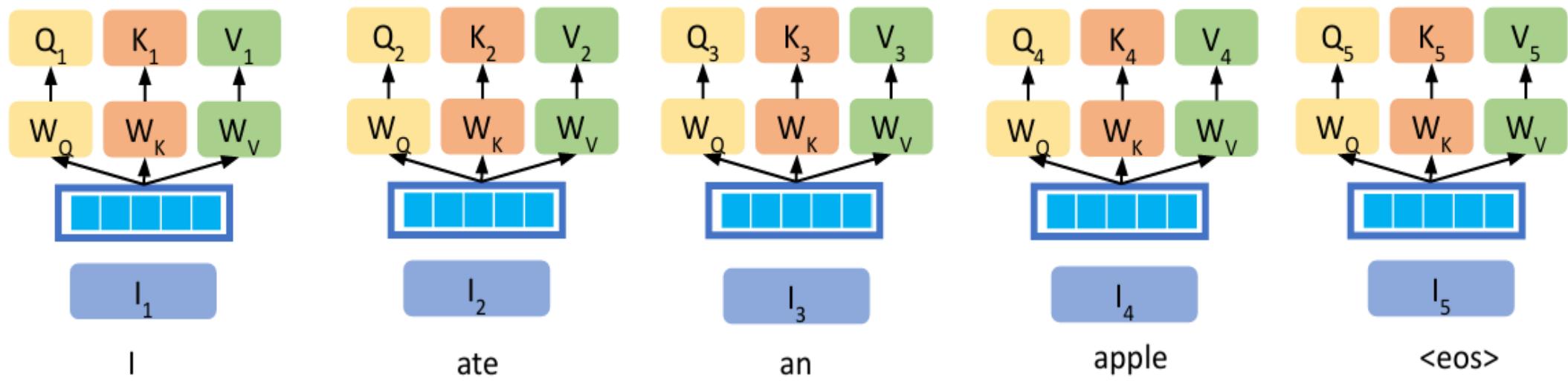


Value

$softmax(\frac{QK^T}{\sqrt{d}})V$

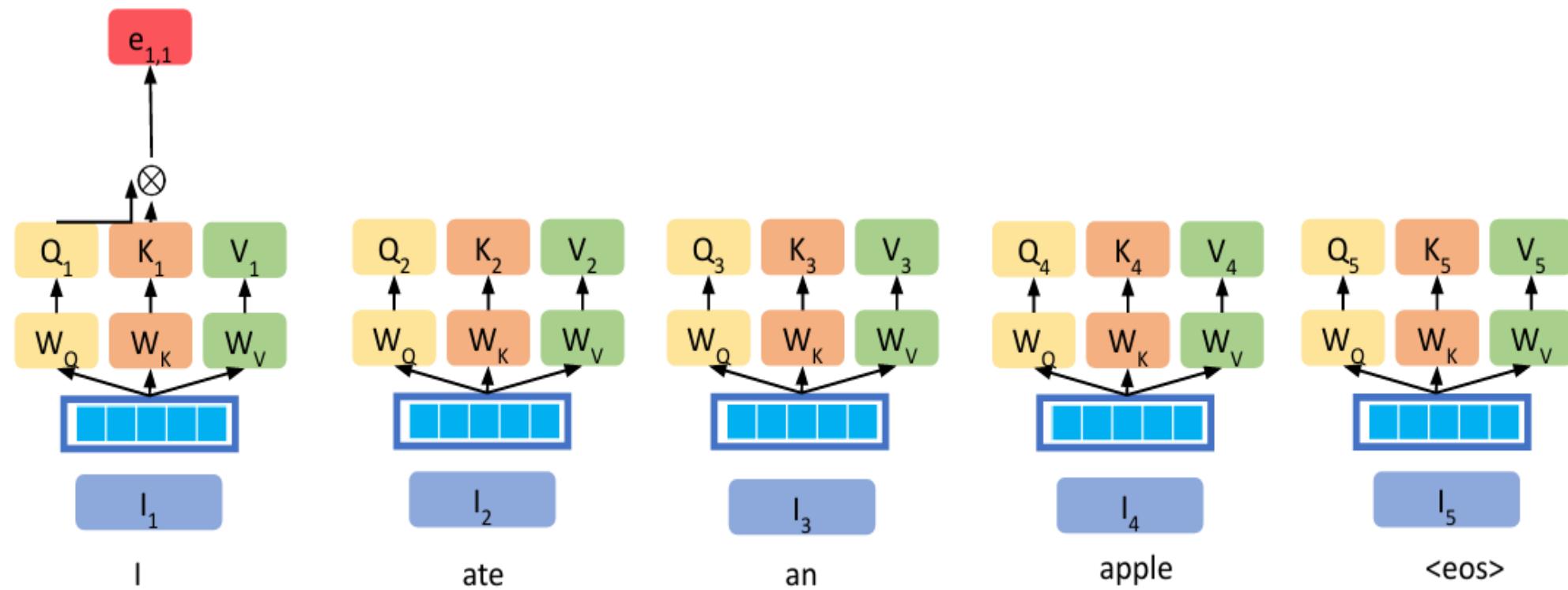
dimensions across QKV have been dropped for brevity

# Attention



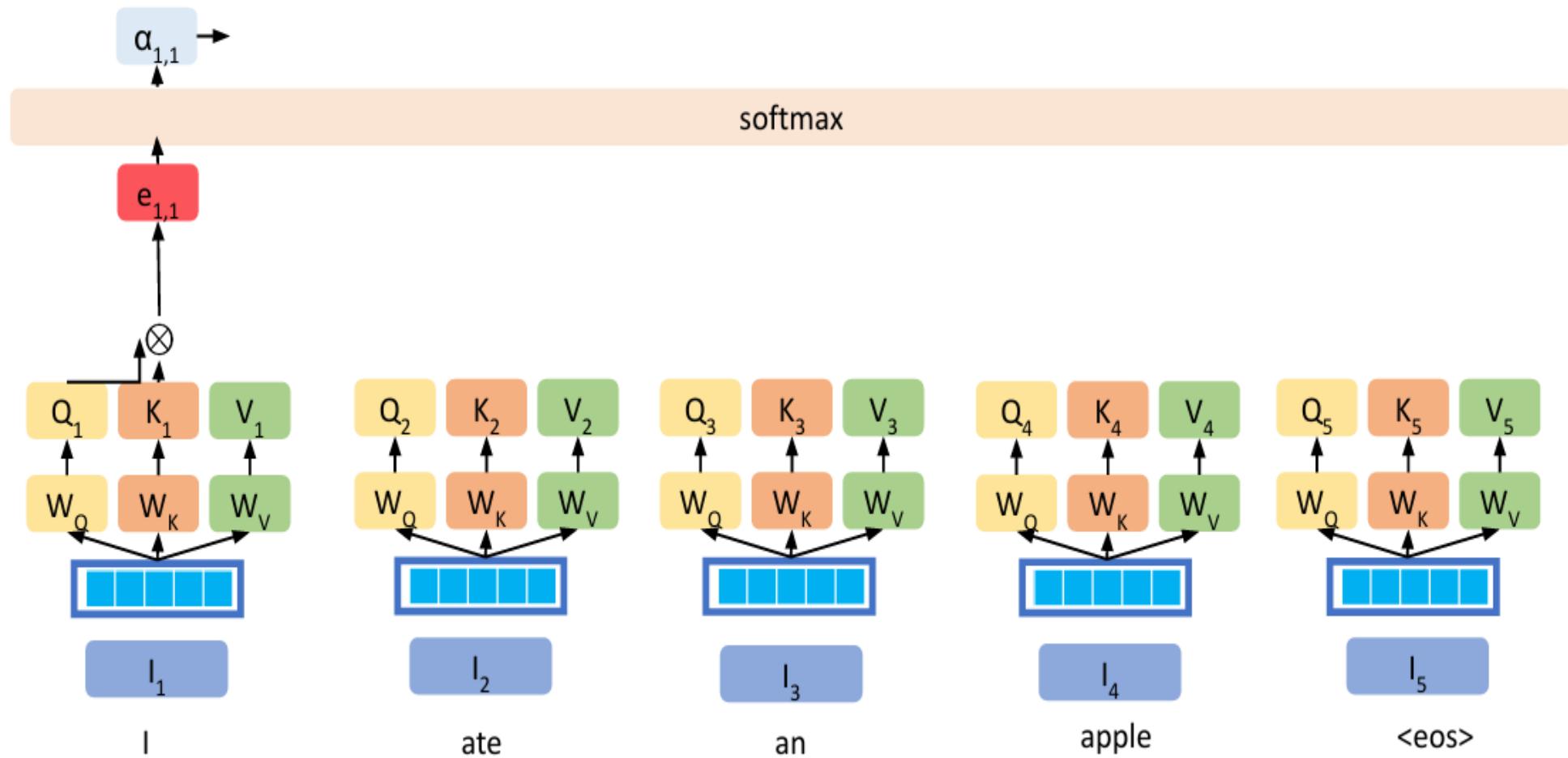
Dimensions across QKV have been dropped for brevity

# Attention



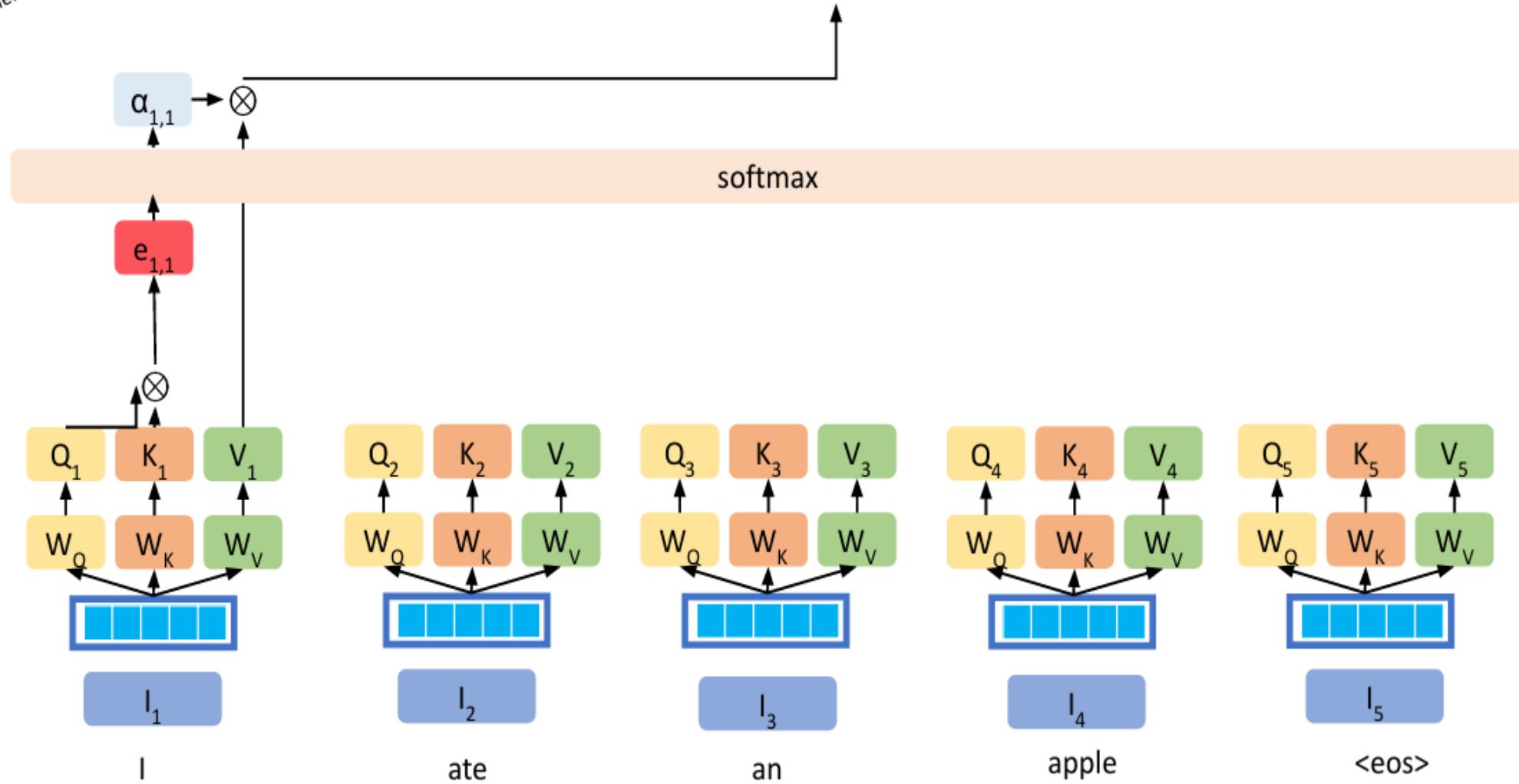
Dimensions across QKV have been dropped for brevity

# Attention



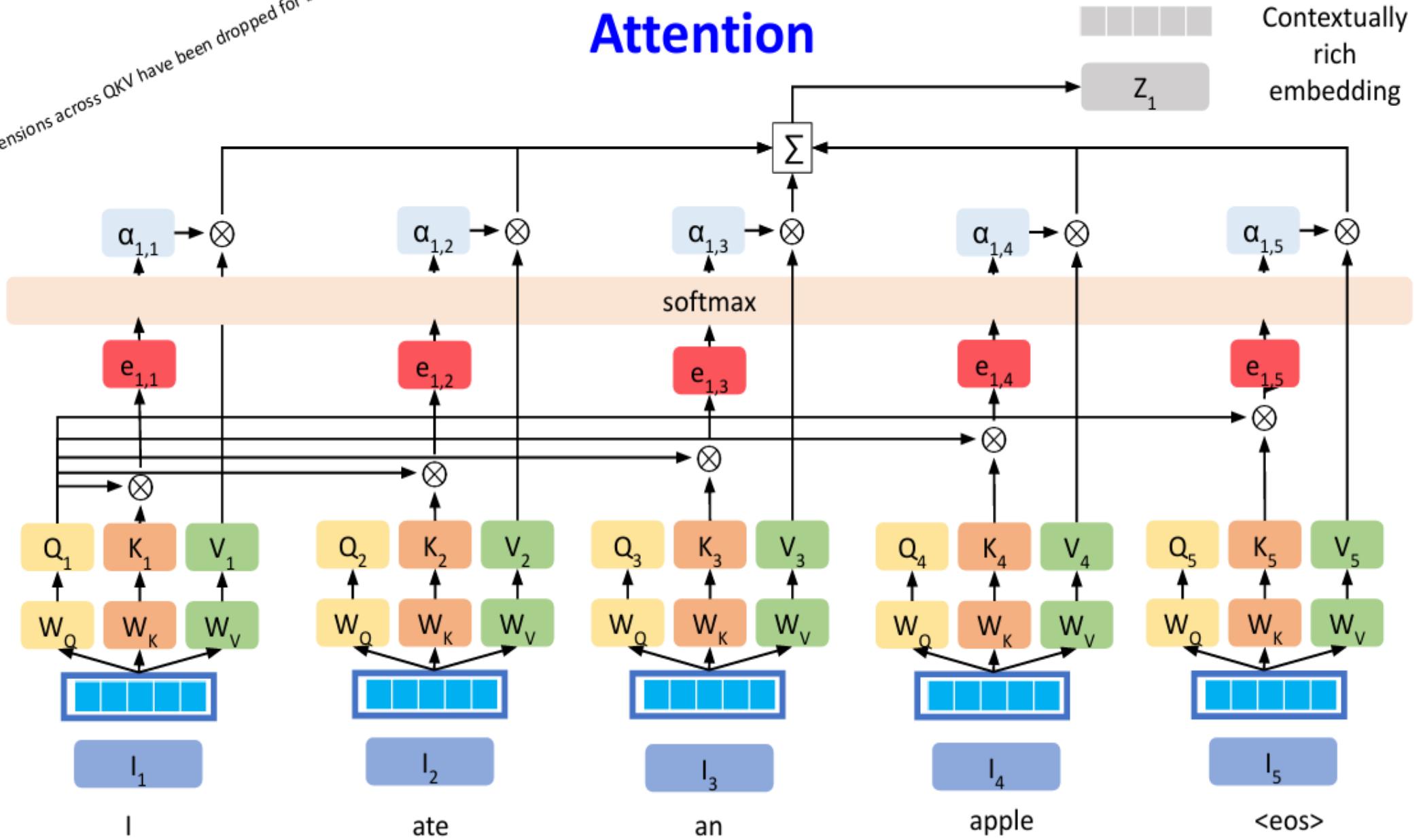
# Attention

Dimensions across QKV have been dropped for brevity



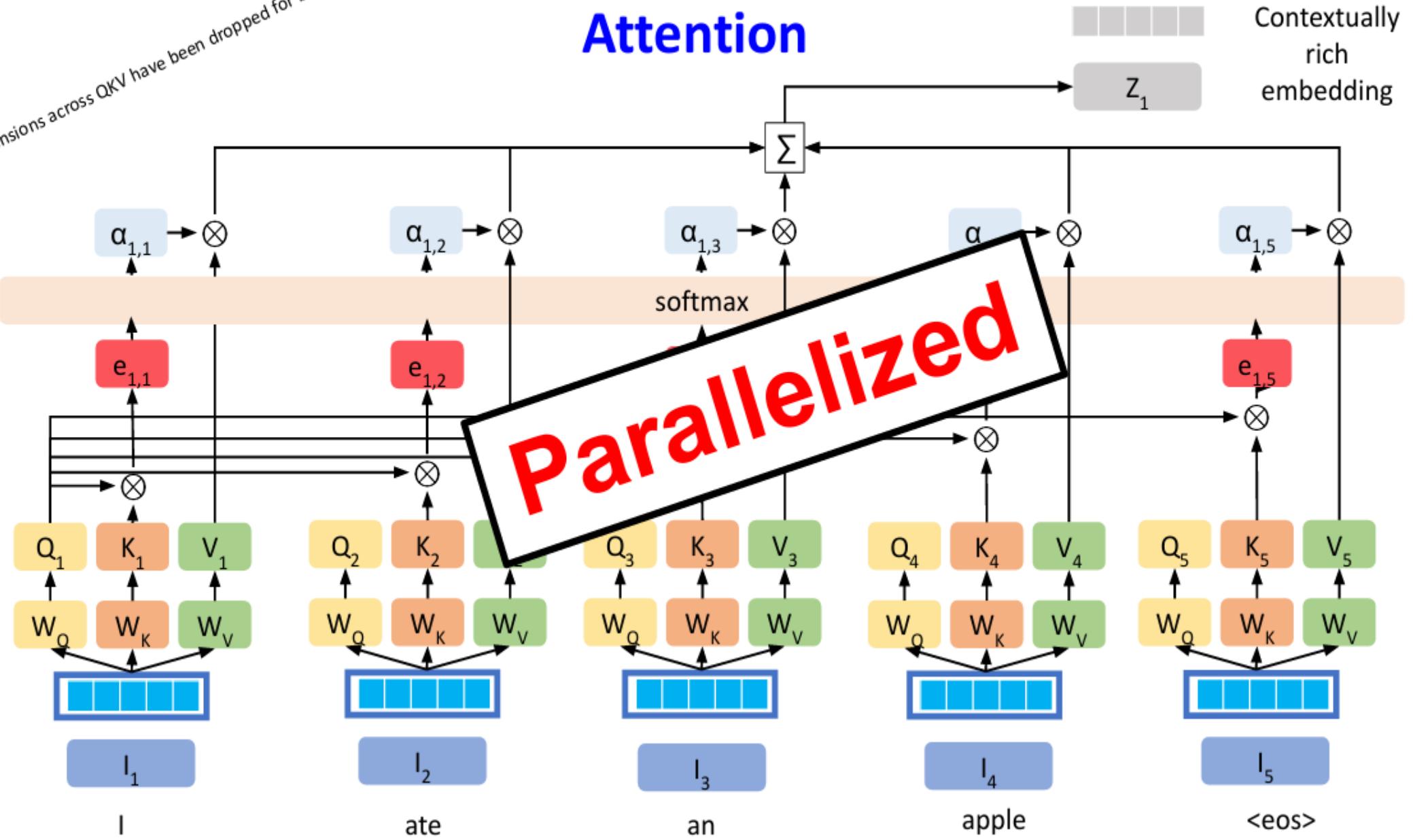
Dimensions across QKV have been dropped for brevity

# Attention



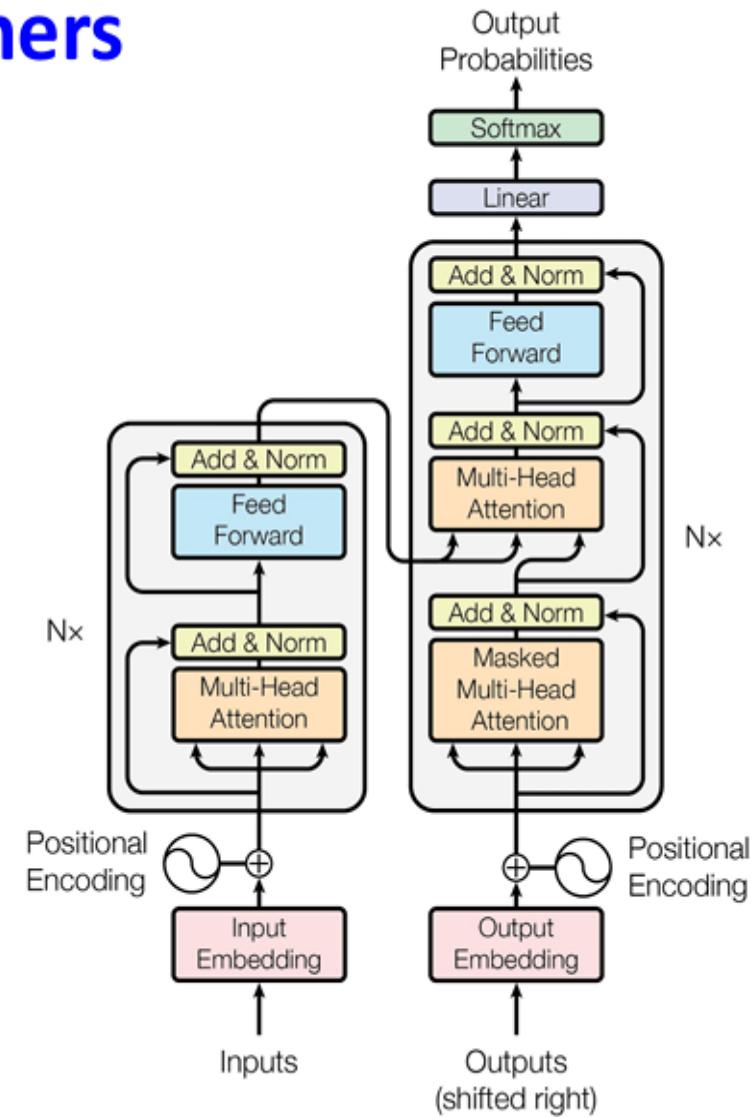
Dimensions across QKV have been dropped for brevity

## Attention



# Transformers

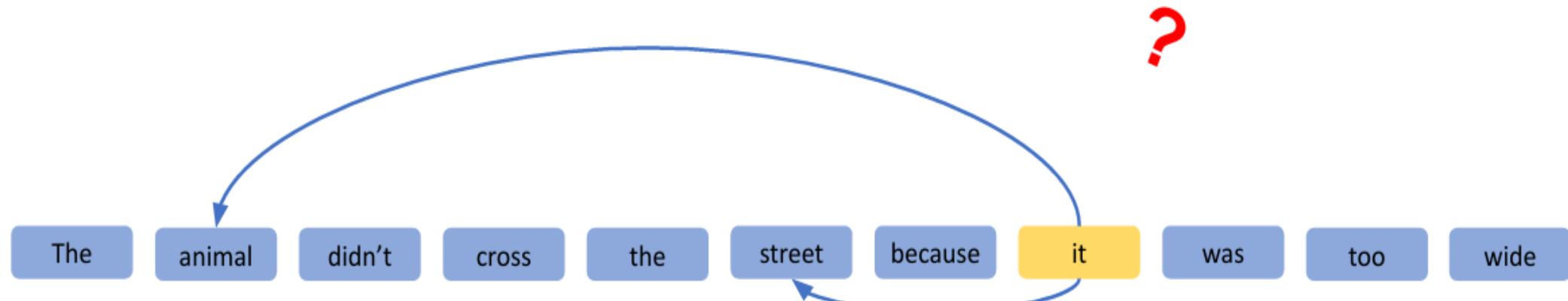
- ✓ Tokenization
- ✓ Input Embeddings
- ✓ Position Encodings
- ✓ Query, Key, & Value
- ✓ Attention
  - Self Attention
  - Multi-Head Attention
  - Feed Forward
  - Add & Norm
  - Encoders
- Masked Attention
- Encoder Decoder Attention
- Linear
- Softmax
- Decoders
- Encoder-Decoder Models



# Self Attention

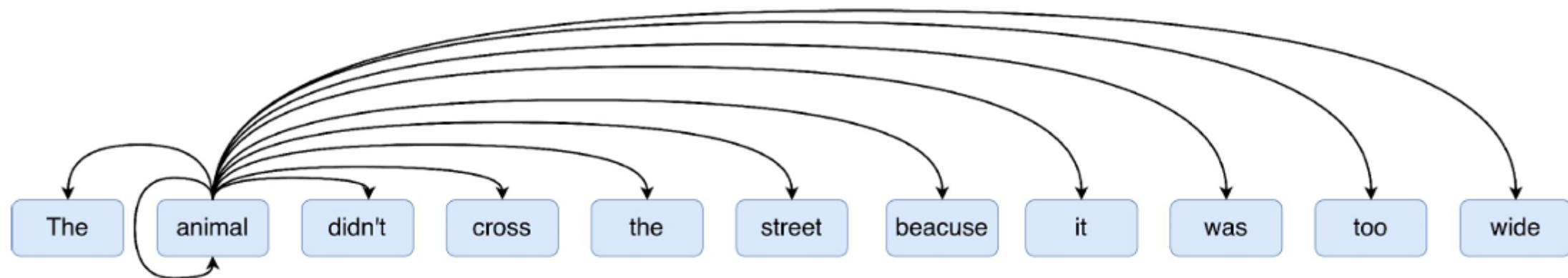
The animal didn't cross the street because it was too wide

# Self Attention

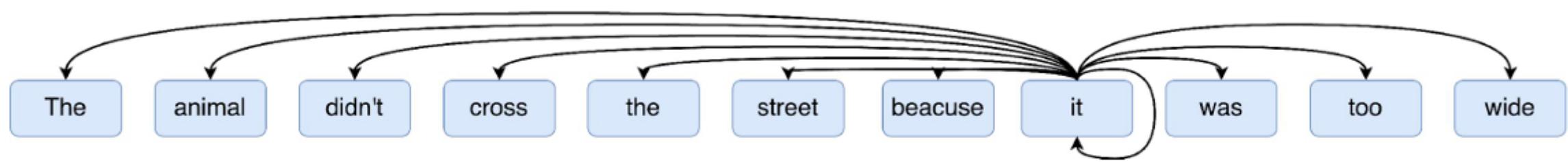


coreference resolution?

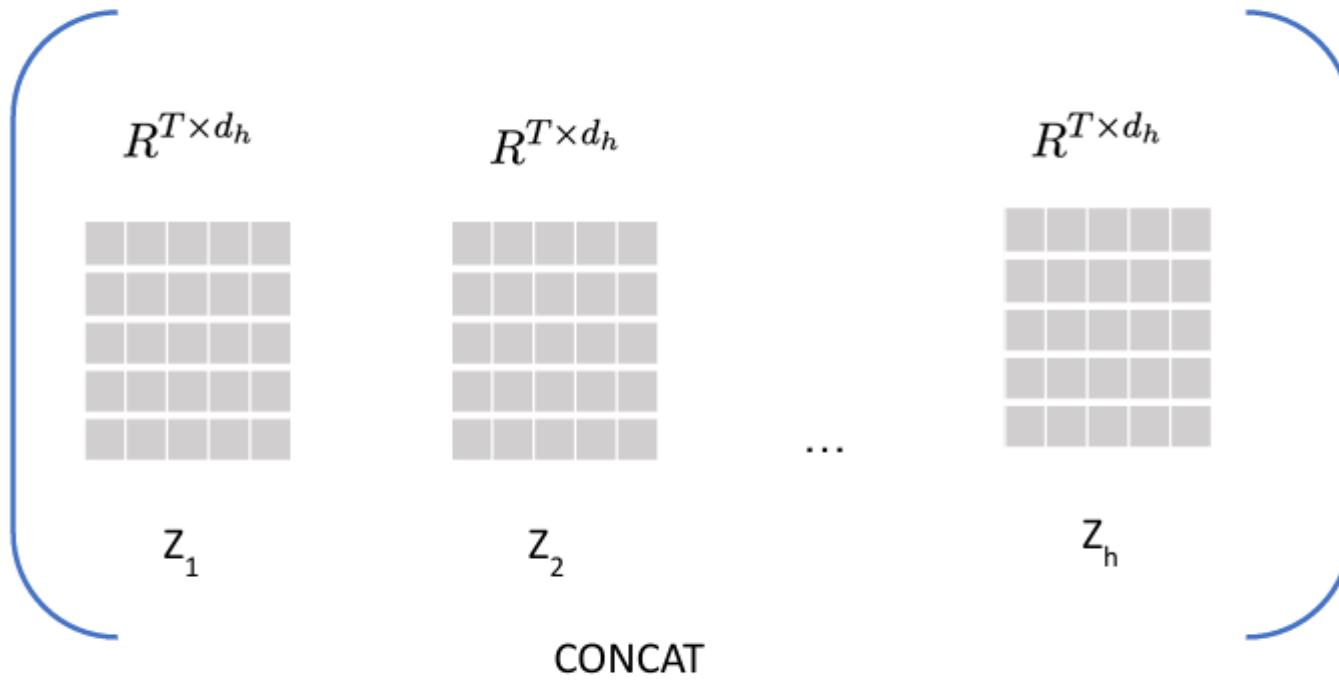
# Self Attention



# Self Attention



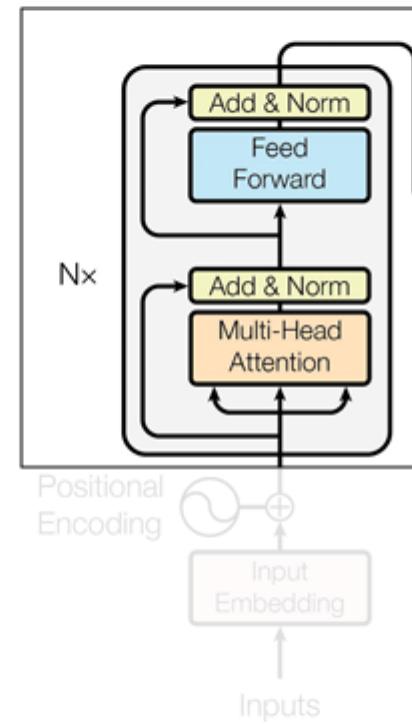
# Multi-Head Attention



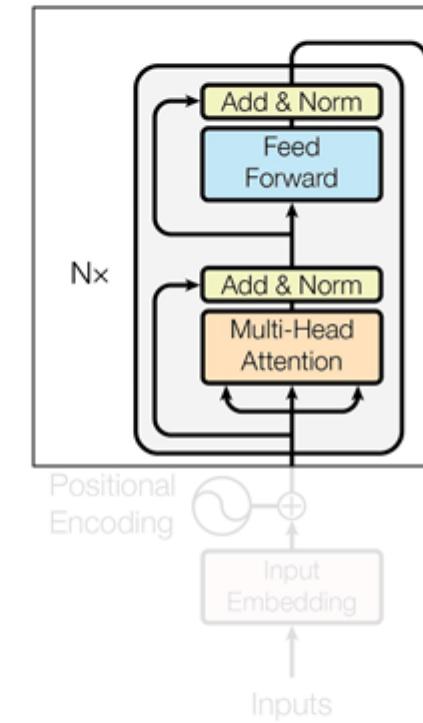
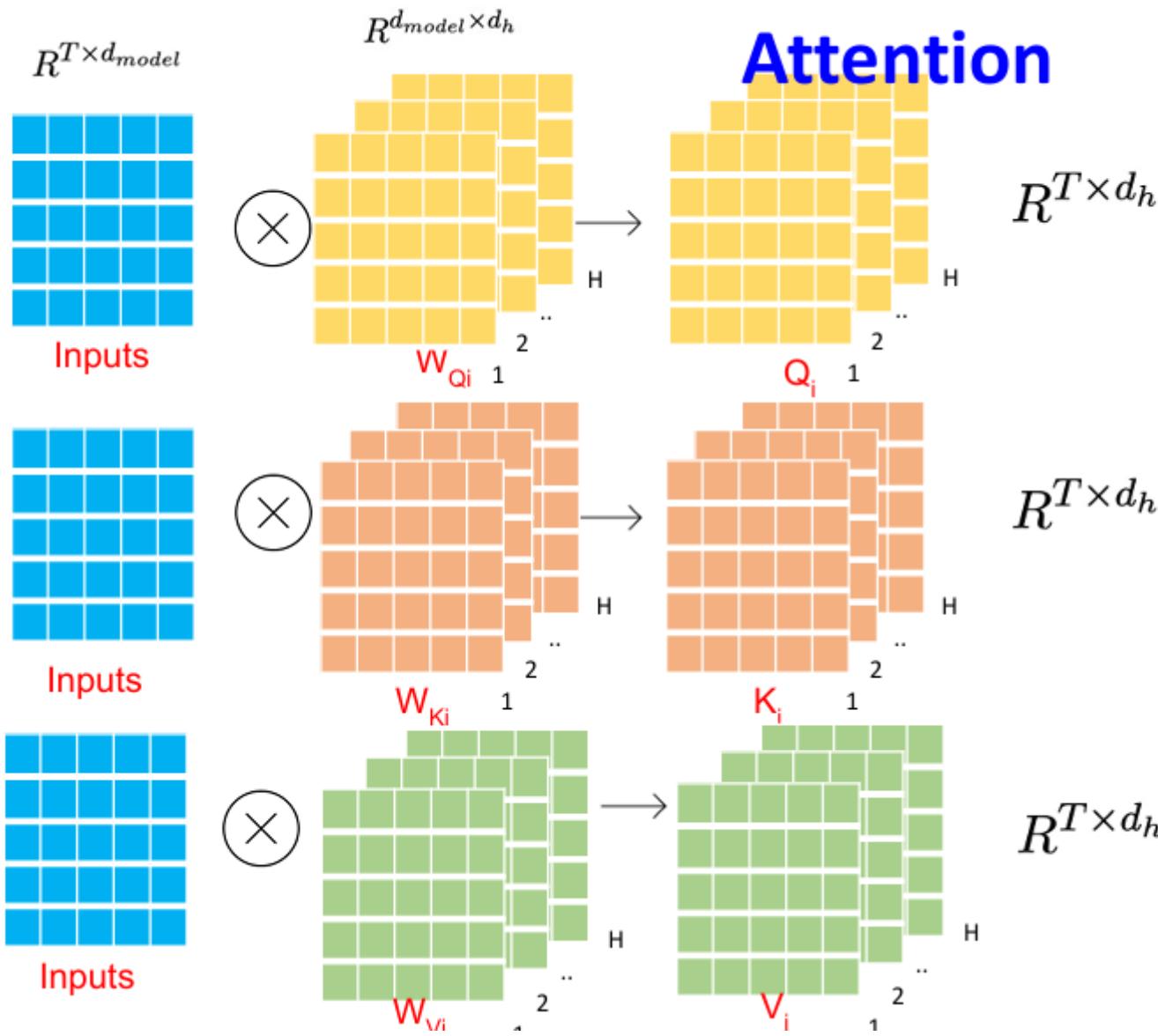
Multi Head Attention : Z

$$R^{T \times d_{model}}$$

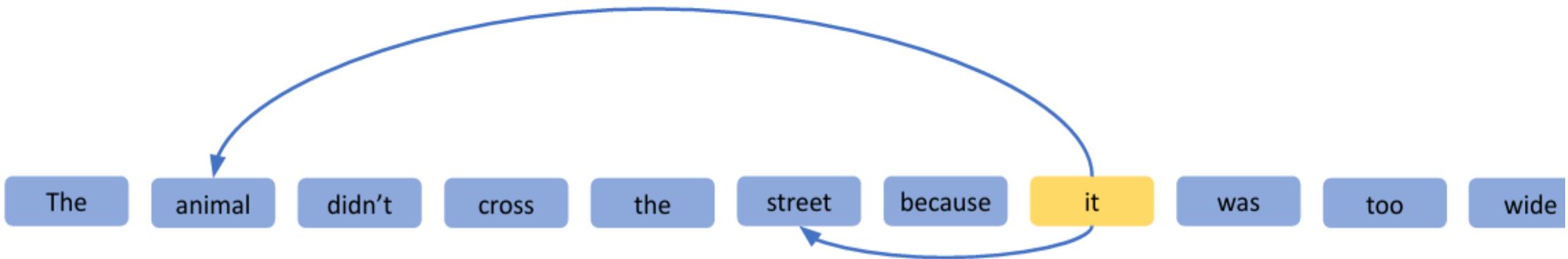
$$d_h = \frac{d_{model}}{h}$$



# Multi-Head Attention



# Multi-Head Attention



Sentence boundaries



Coreference resolution



Context



Semantic relationships



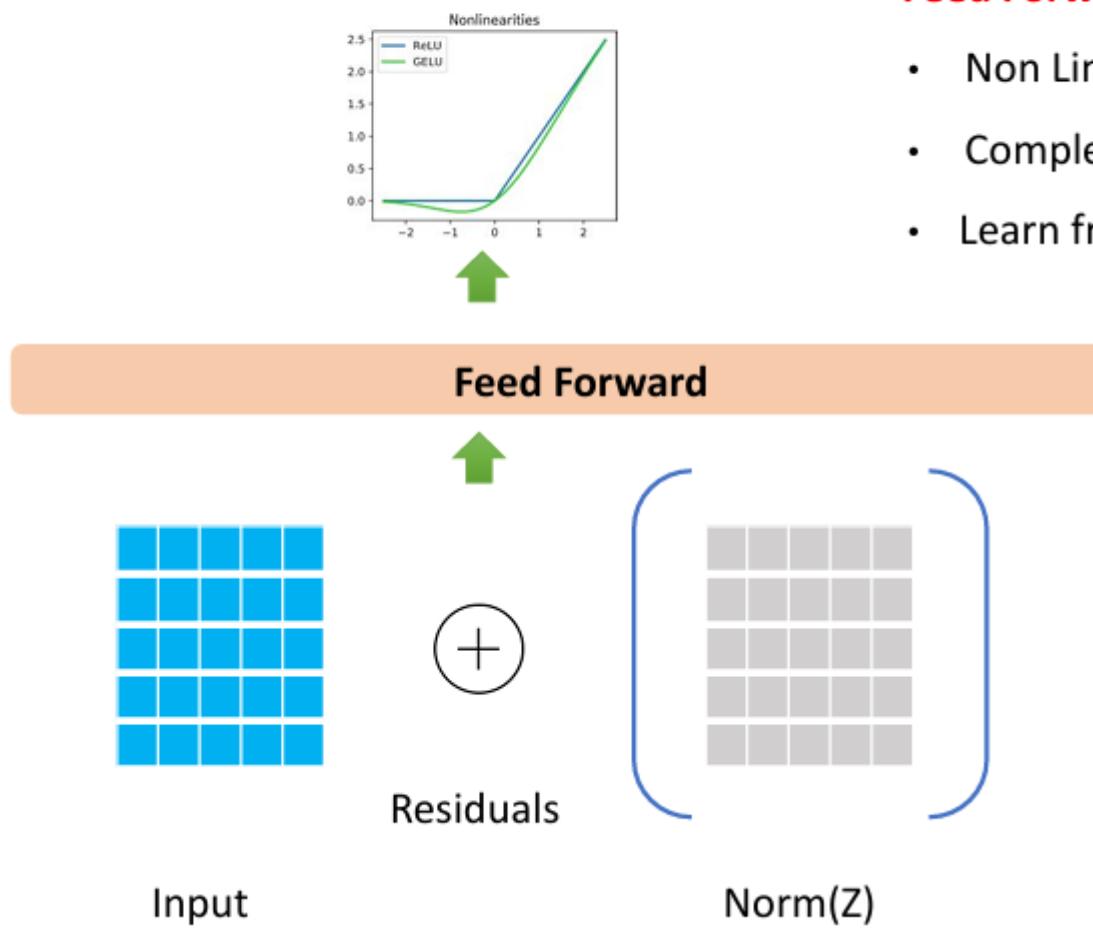
Part of speech



Comparisons

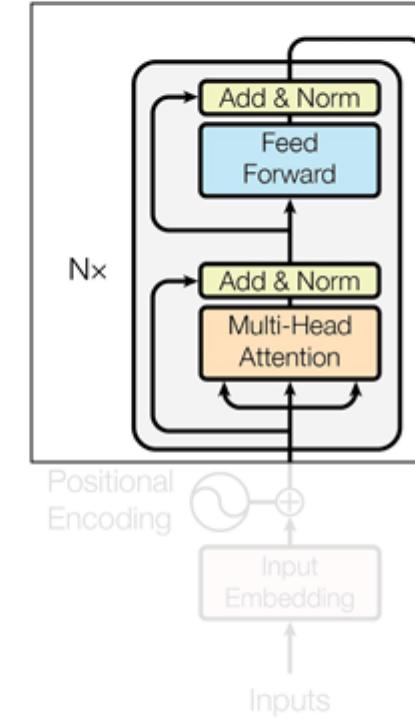


# Feed Forward

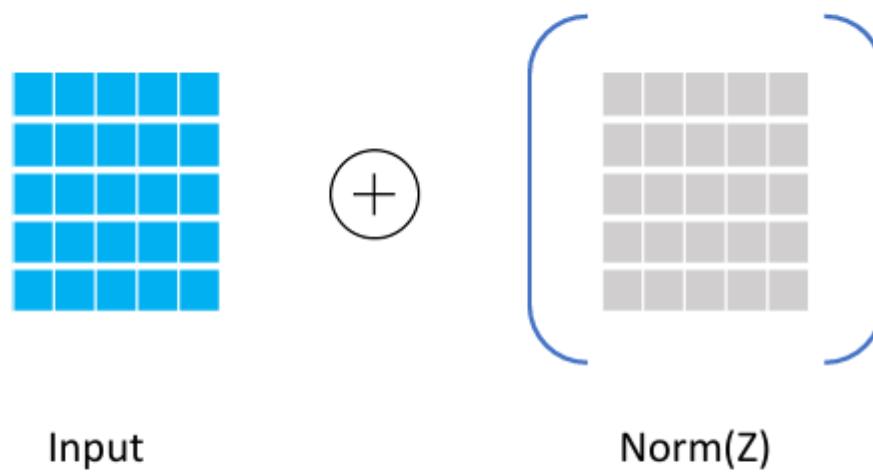


## Feed Forward

- Non Linearity
- Complex Relationships
- Learn from each other



# Add & Norm



## Normalization

Mean 0, Std dev 1

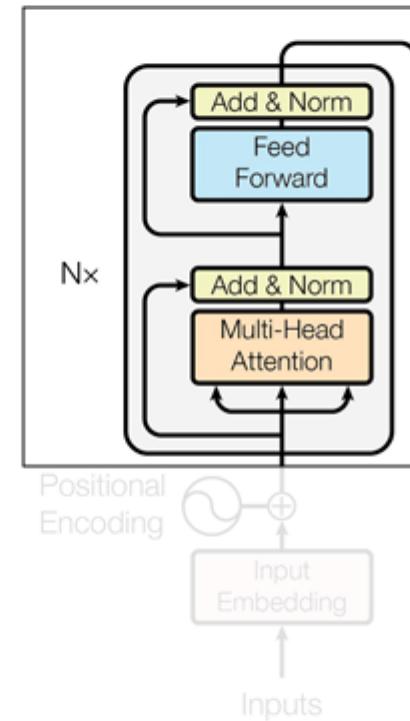
Stabilizes training

Regularization effect

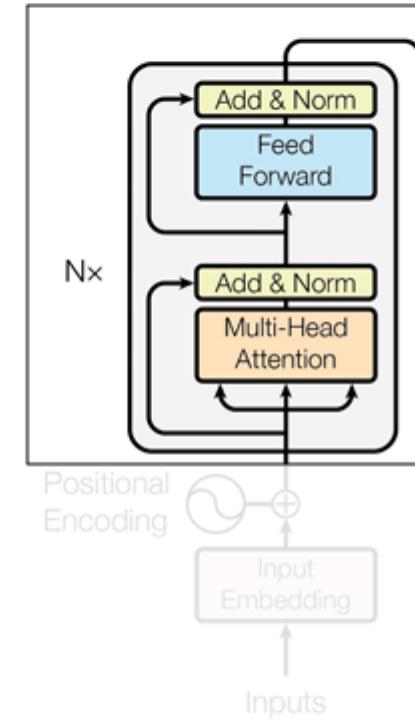
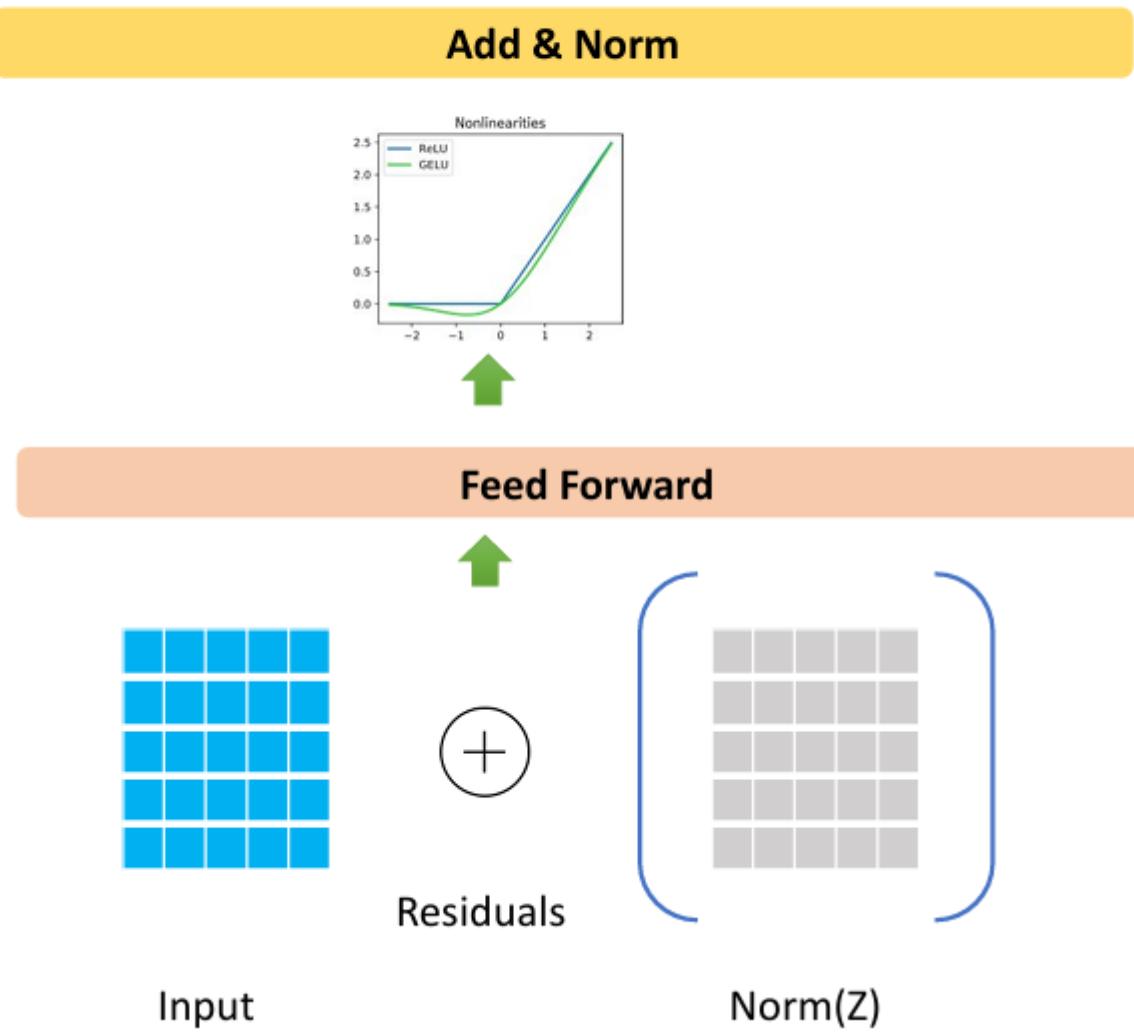
## Add Residuals

Avoid vanishing gradients

Train deeper networks

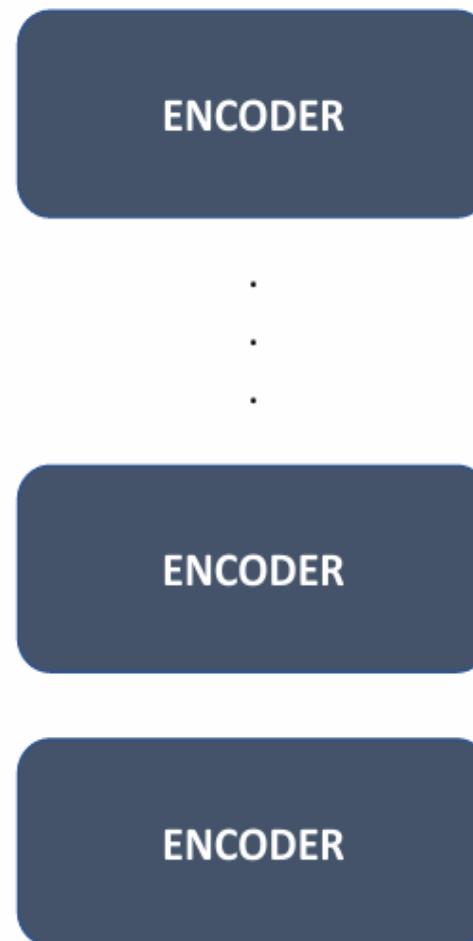


# Add & Norm

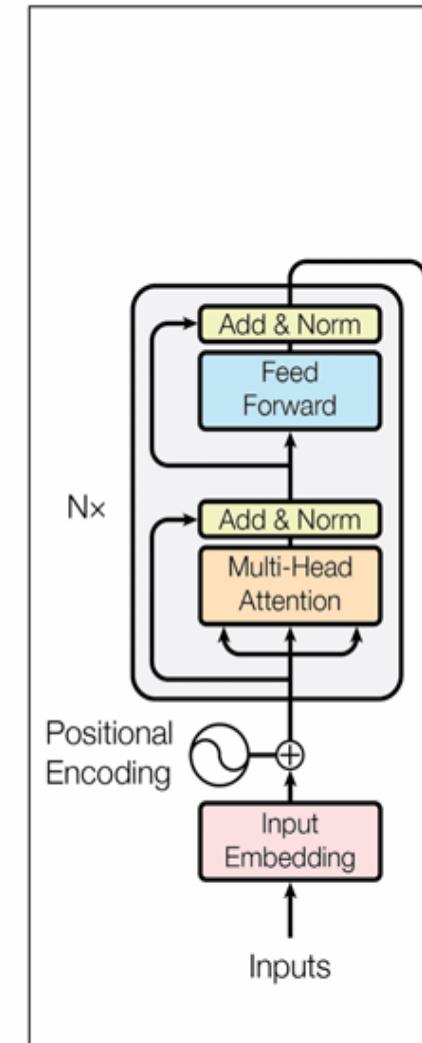


# Encoders

## Encoder

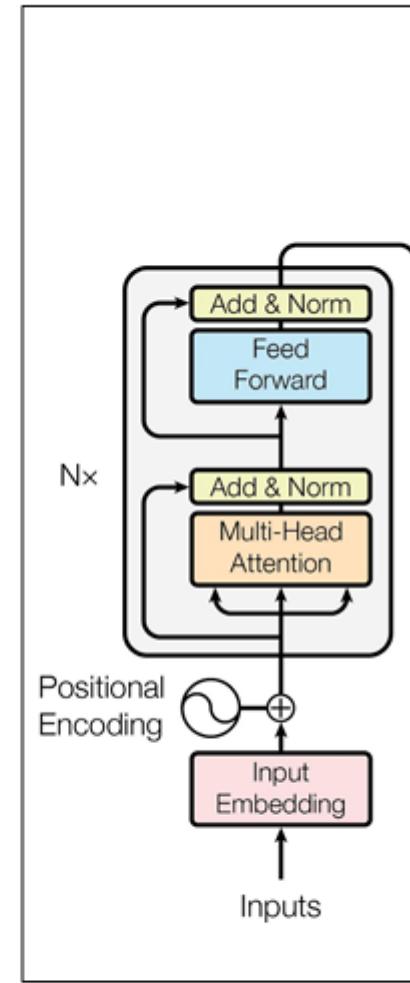


Input to Encoder<sub>i+1</sub>  
↑  
Output from Encoder<sub>i</sub>



# Transformers

- ✓ Tokenization
- ✓ Input Embeddings
- ✓ Position Encodings
- ✓ Query, Key, & Value
- ✓ Attention
- ✓ Self Attention
- ✓ Multi-Head Attention
- ✓ Feed Forward
- ✓ Add & Norm
- ✓ Encoders
  - Masked Attention
  - Encoder Decoder Attention
  - Linear
  - Softmax
  - Decoders
  - Encoder-Decoder Models



# Masked Multi-head attention

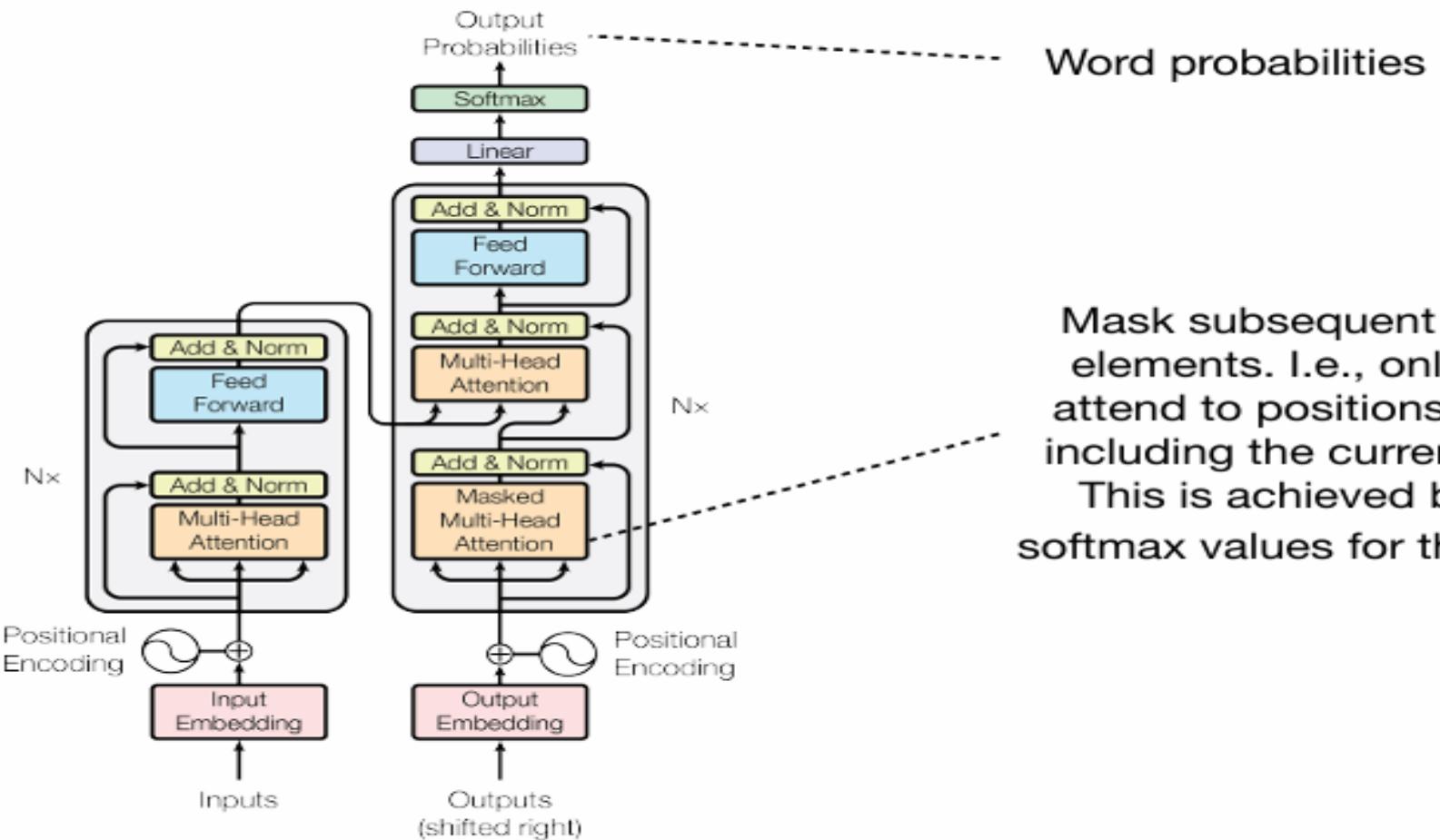
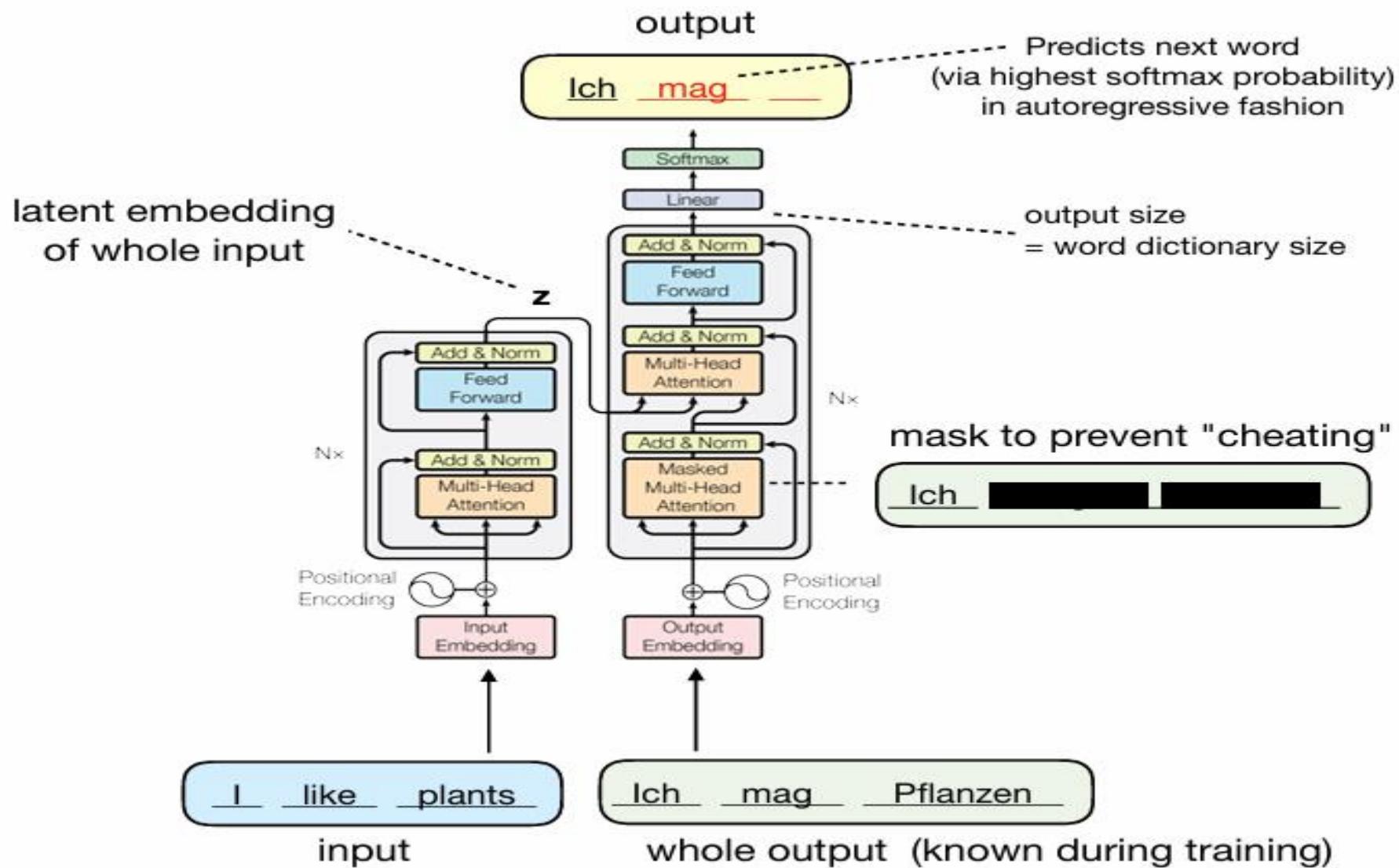
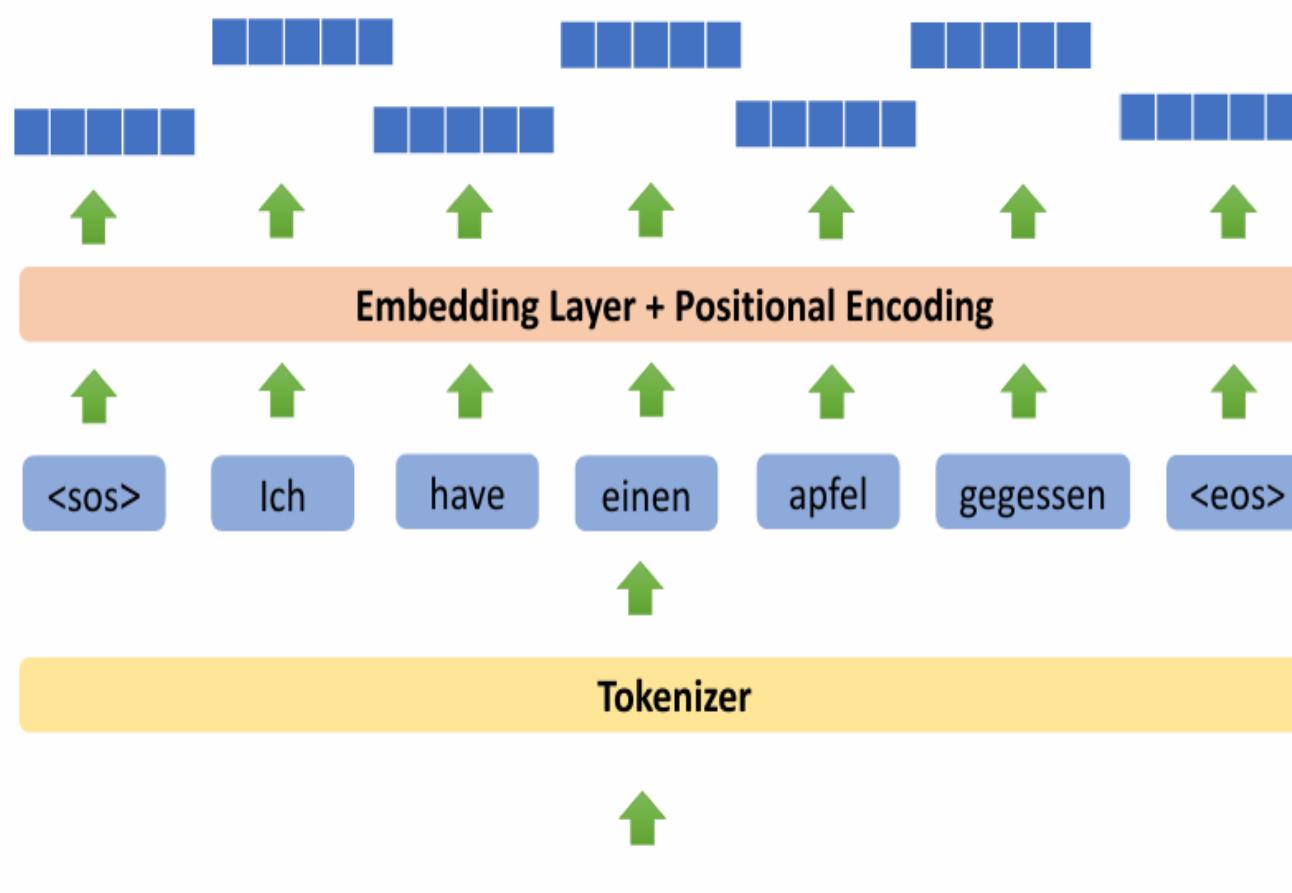


Figure 1: The Transformer - model architecture.

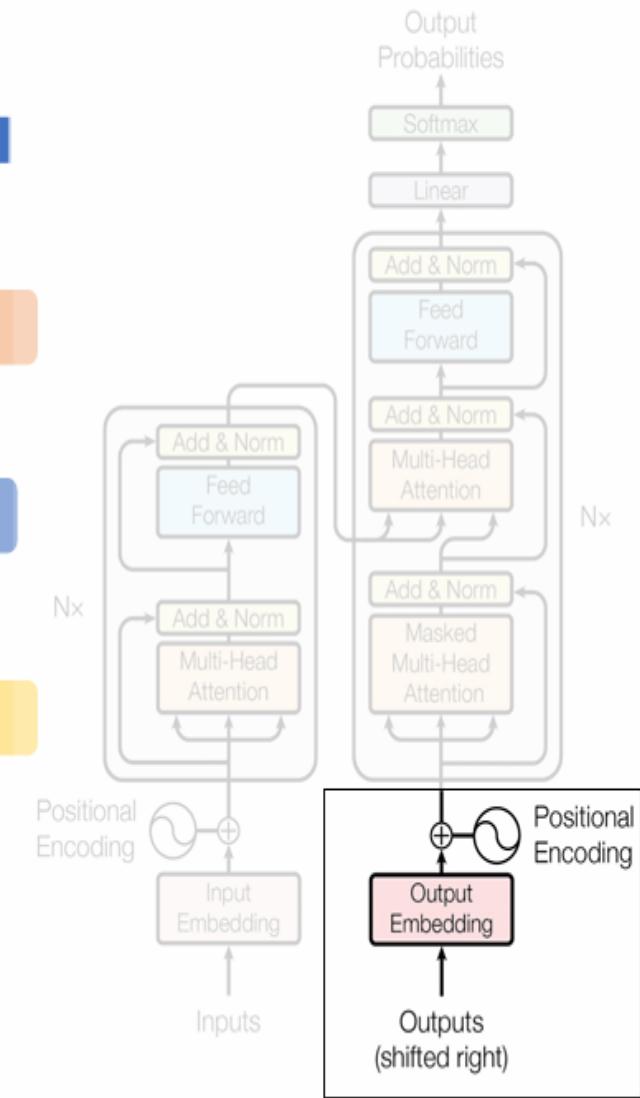
Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L. and Polosukhin, I., 2017. Attention Is All You Need.



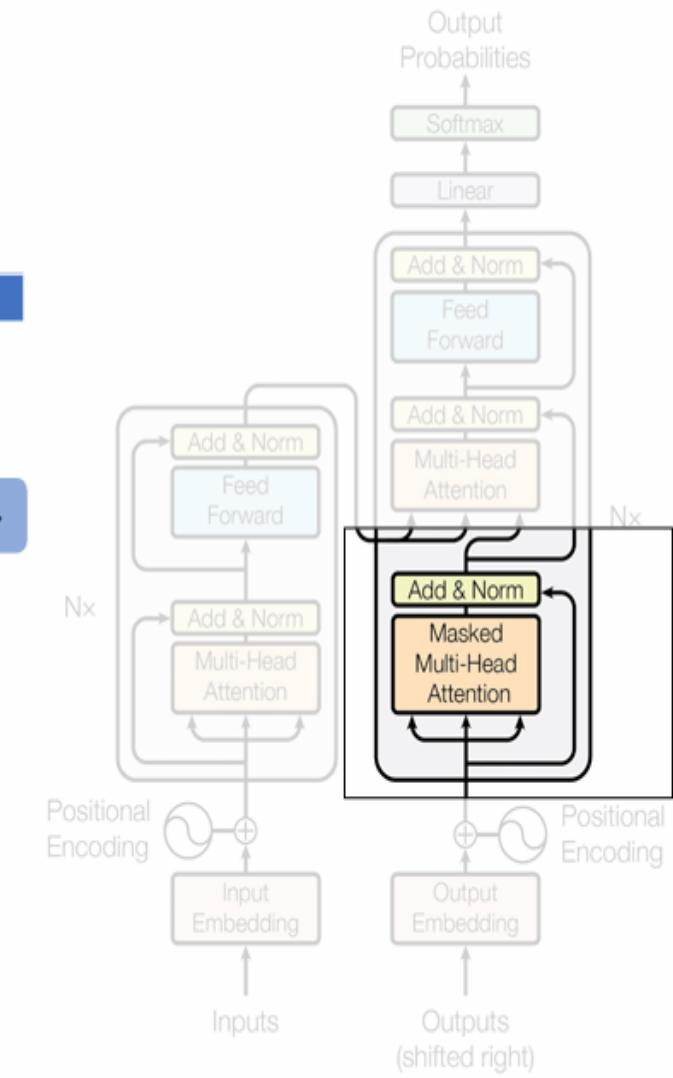
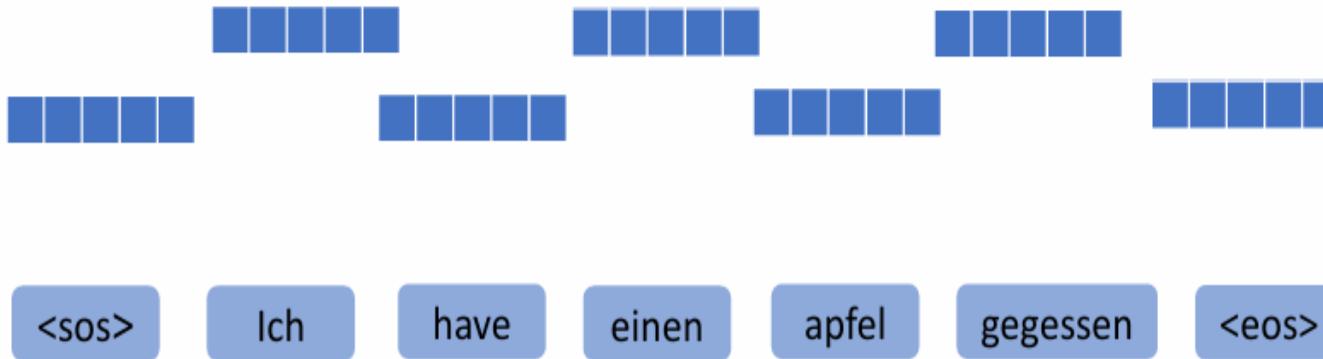
# Targets



Generate Target Emebeddings

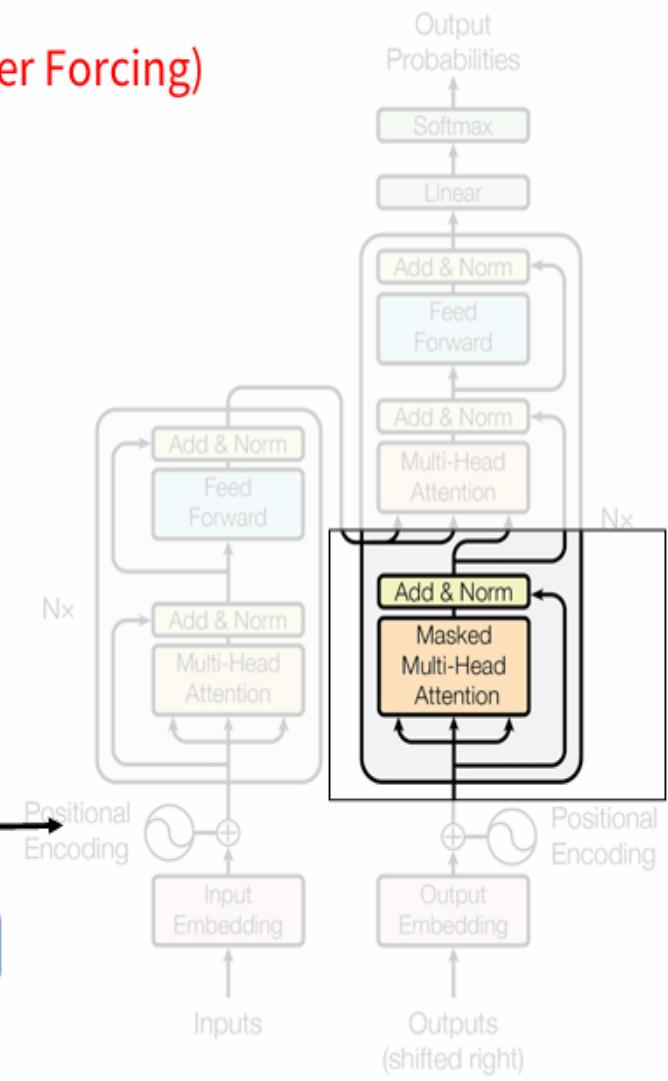
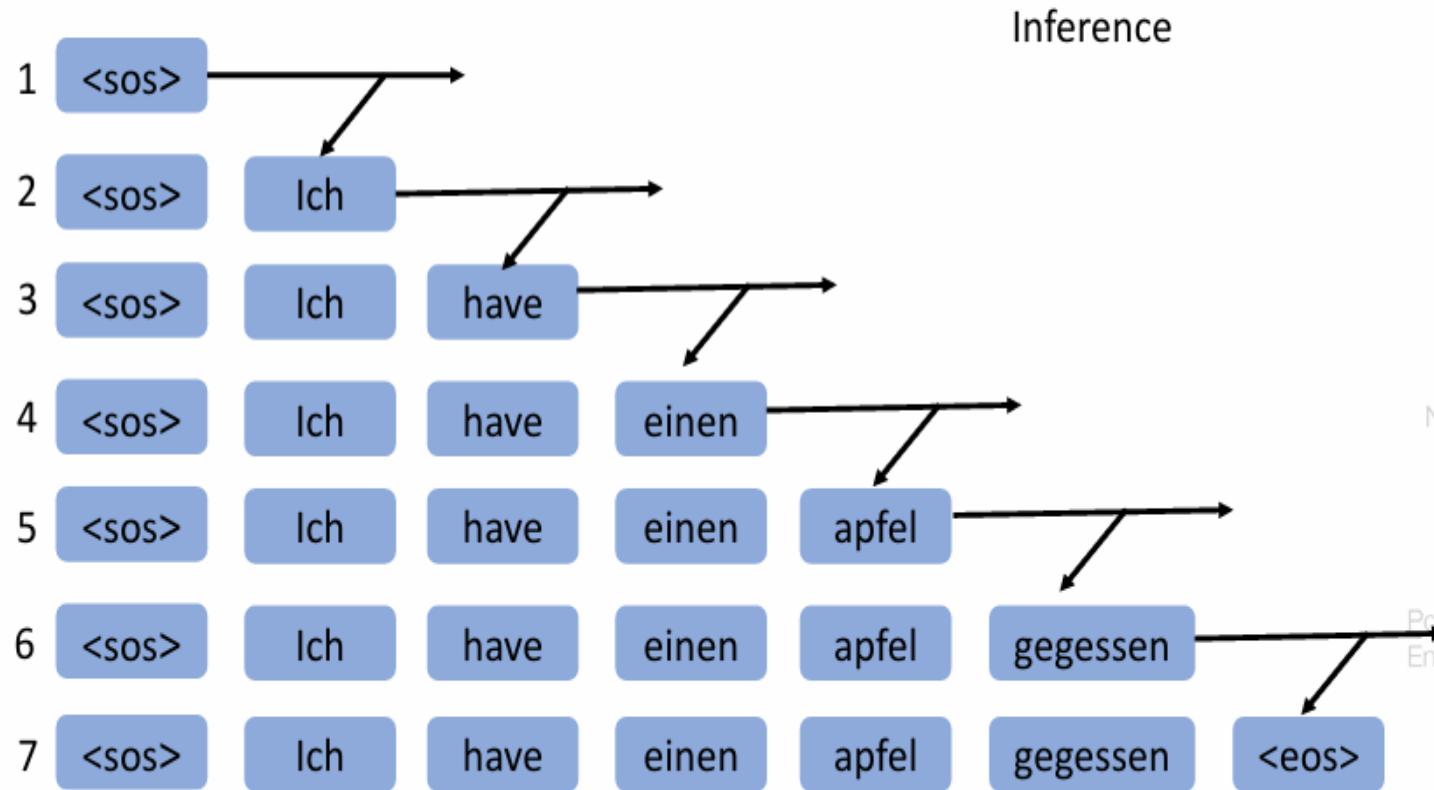


# Masked Multi Head Attention



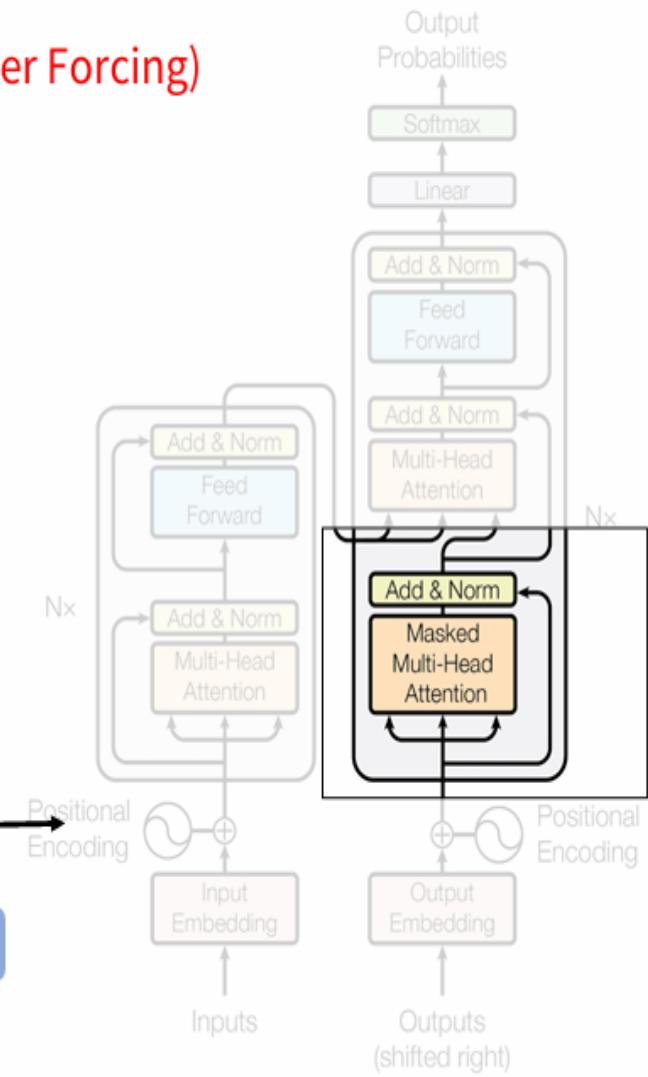
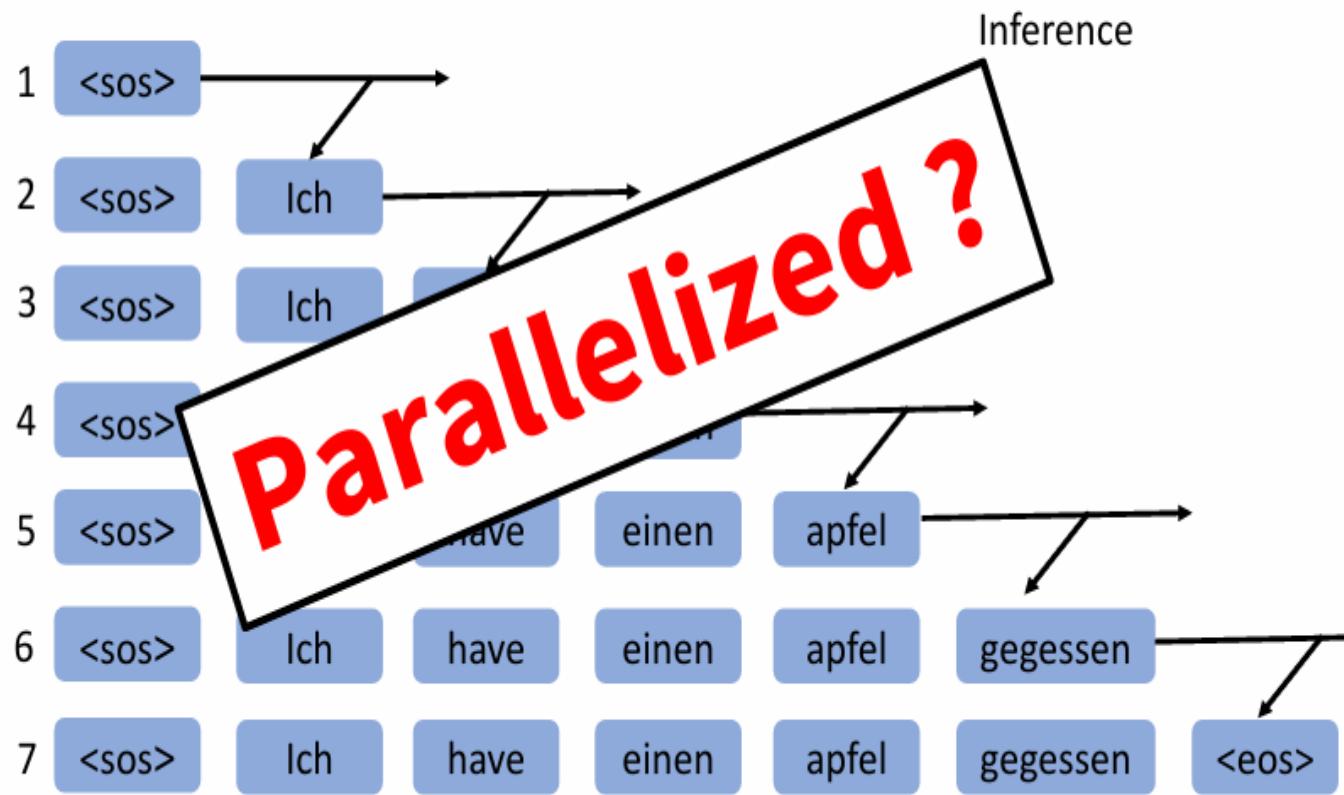
# Masked Multi Head Attention

Decoding step by step (using Teacher Forcing)



# Masked Multi Head Attention

Decoding step by step (using Teacher Forcing)

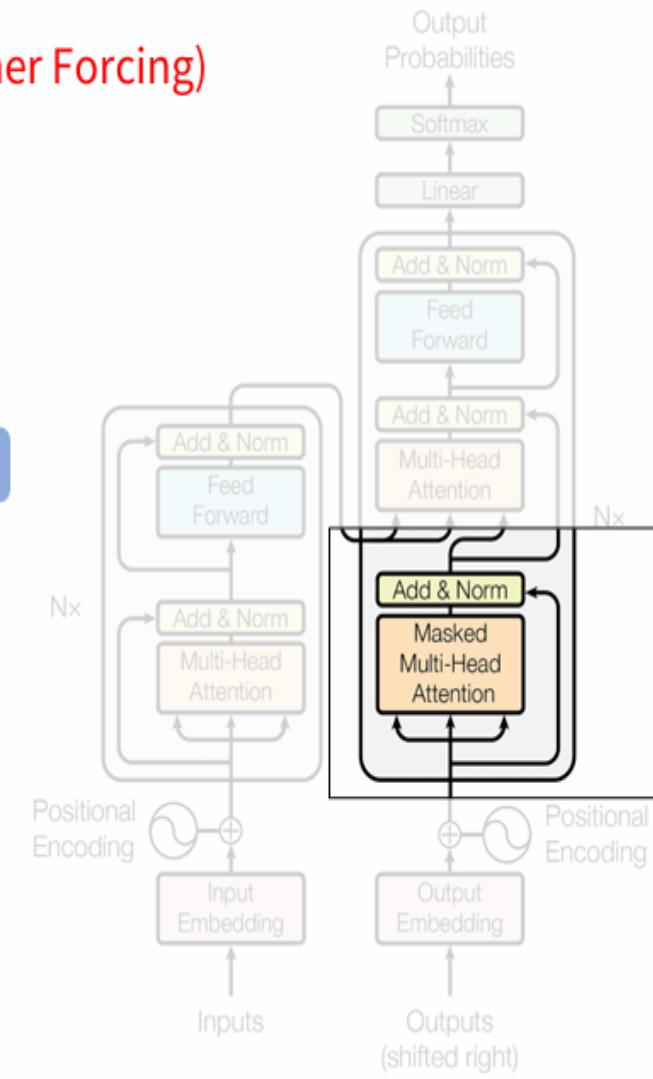


# Masked Multi Head Attention

Decoding step by step (using Teacher Forcing)

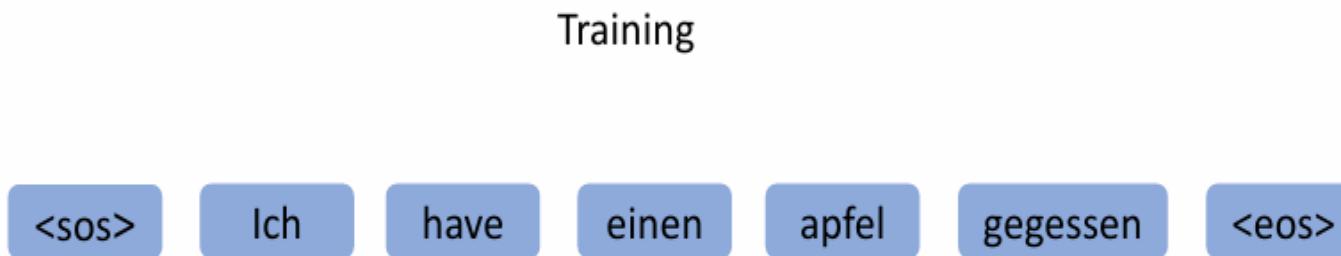
Training

<sos> Ich have einen apfel gegessen <eos>

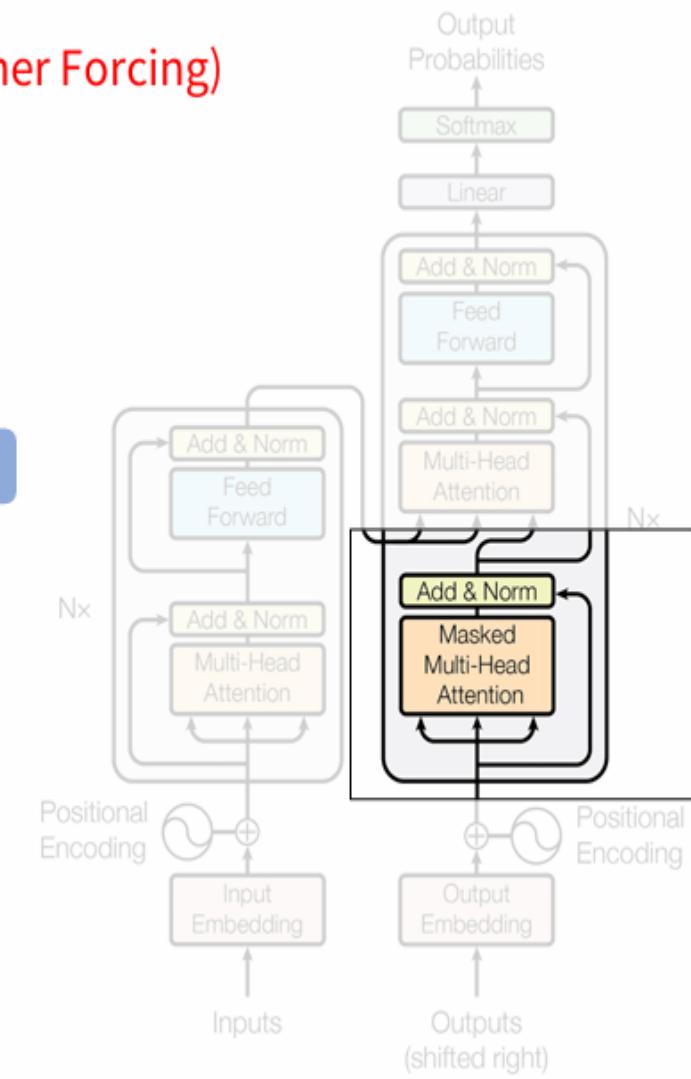


# Masked Multi Head Attention

Decoding step by step (using Teacher Forcing)



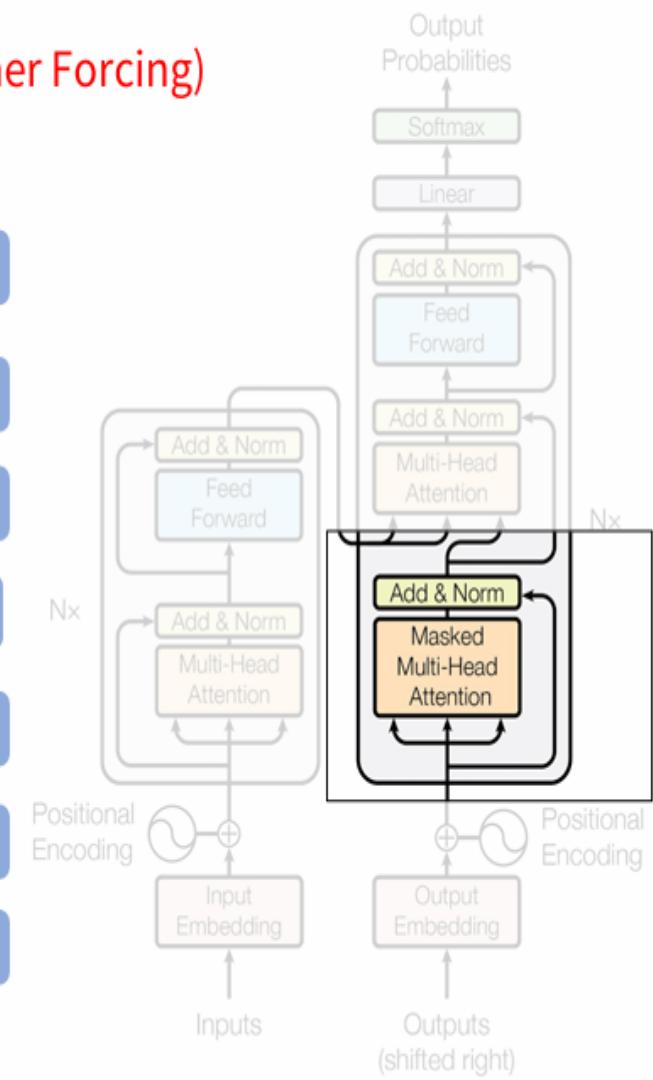
*Outputs at time  $T$  should only pay attention to outputs  
until time  $T-1$*



# Masked Multi Head Attention

Decoding step by step (using Teacher Forcing)

1	<sos>	Ich	have	einen	apfel	gegessen	<eos>
2	<sos>	Ich	have	einen	apfel	gegessen	<eos>
3	<sos>	Ich	have	einen	apfel	gegessen	<eos>
4	<sos>	Ich	have	einen	apfel	gegessen	<eos>
5	<sos>	Ich	have	einen	apfel	gegessen	<eos>
6	<sos>	Ich	have	einen	apfel	gegessen	<eos>
7	<sos>	Ich	have	einen	apfel	gegessen	<eos>

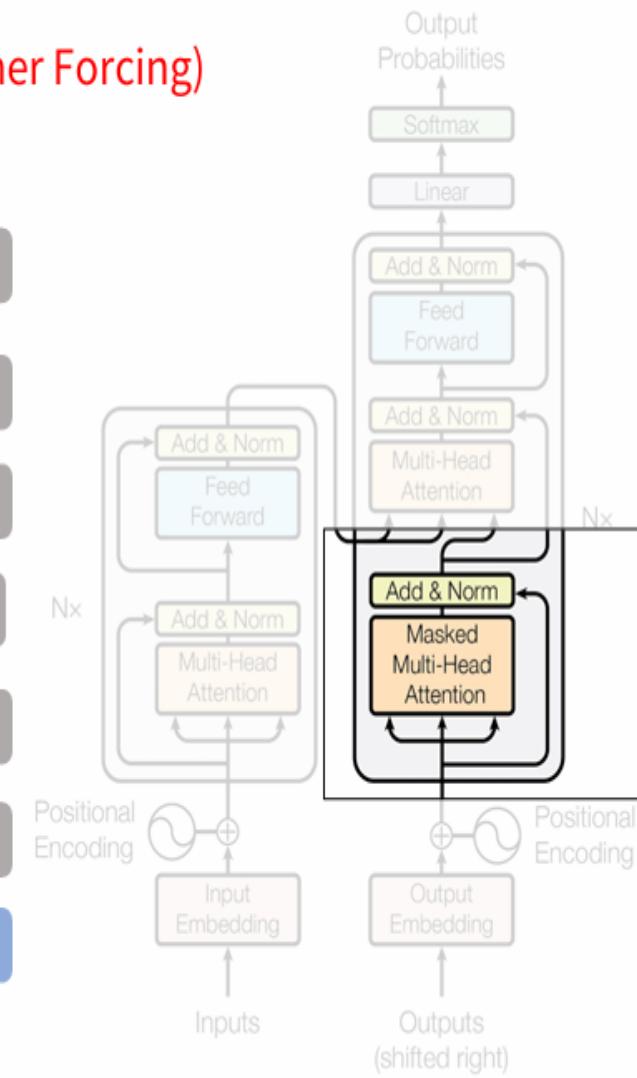


# Masked Multi Head Attention

Decoding step by step (using Teacher Forcing)

1	<sos>	Ich	have	einen	apfel	gegessen	<eos>
2	<sos>	Ich	have	einen	apfel	gegessen	<eos>
3	<sos>	Ich	have	einen	apfel	gegessen	<eos>
4	<sos>	Ich	have	einen	apfel	gegessen	<eos>
5	<sos>	Ich	have	einen	apfel	gegessen	<eos>
6	<sos>	Ich	have	einen	apfel	gegessen	<eos>
7	<sos>	Ich	have	einen	apfel	gegessen	<eos>

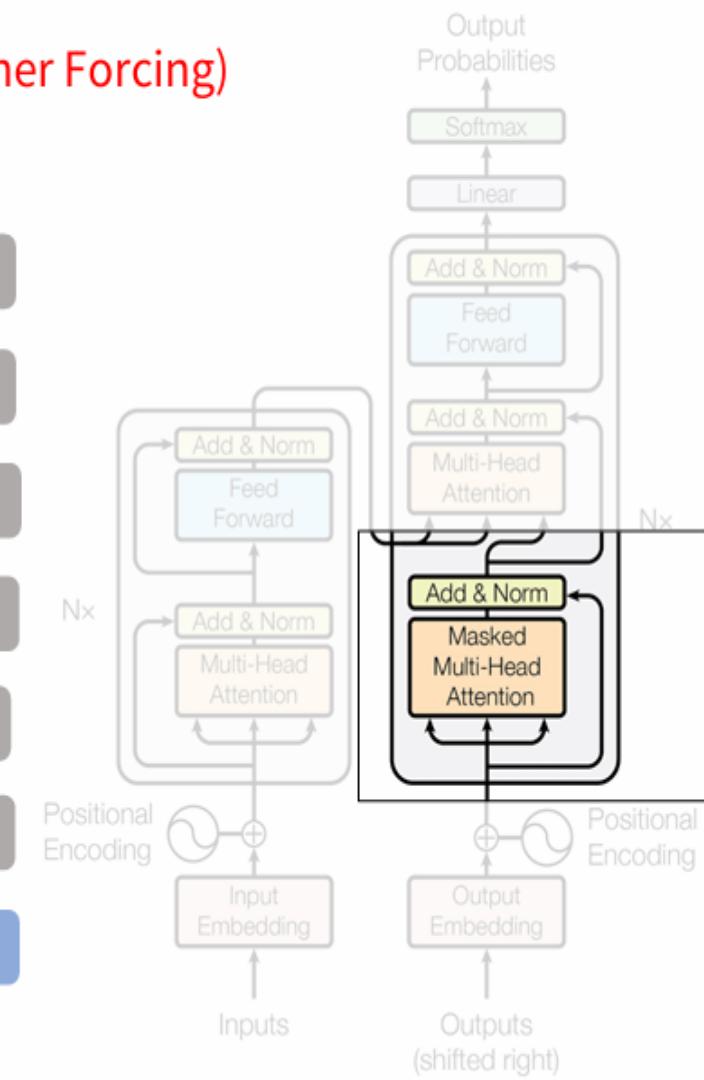
Mask the available attention values ?



# Masked Multi Head Attention

Decoding step by step (using Teacher Forcing)

1	<sos>	- ∞	- ∞	- ∞	- ∞	- ∞	- ∞
2	<sos>	Ich	- ∞	- ∞	- ∞	- ∞	- ∞
3	<sos>	Ich	have	- ∞	- ∞	- ∞	- ∞
4	<sos>	Ich	have	einen	- ∞	- ∞	- ∞
5	<sos>	Ich	have	einen	apfel	- ∞	- ∞
6	<sos>	Ich	have	einen	apfel	gegessen	- ∞
7	<sos>	Ich	have	einen	apfel	gegessen	<eos>

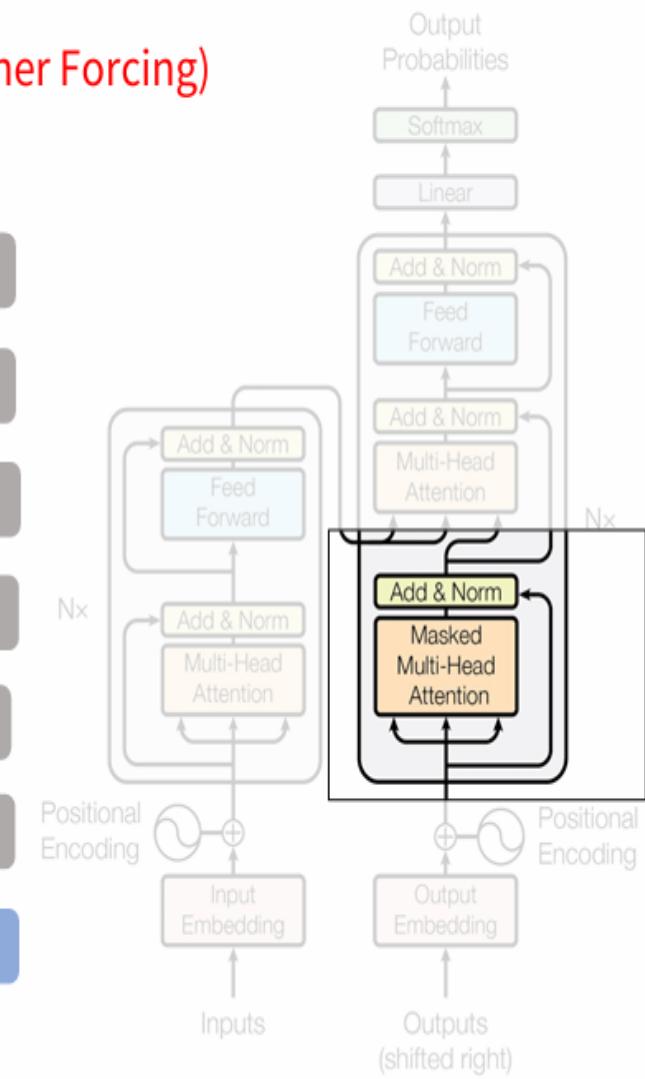


# Masked Multi Head Attention

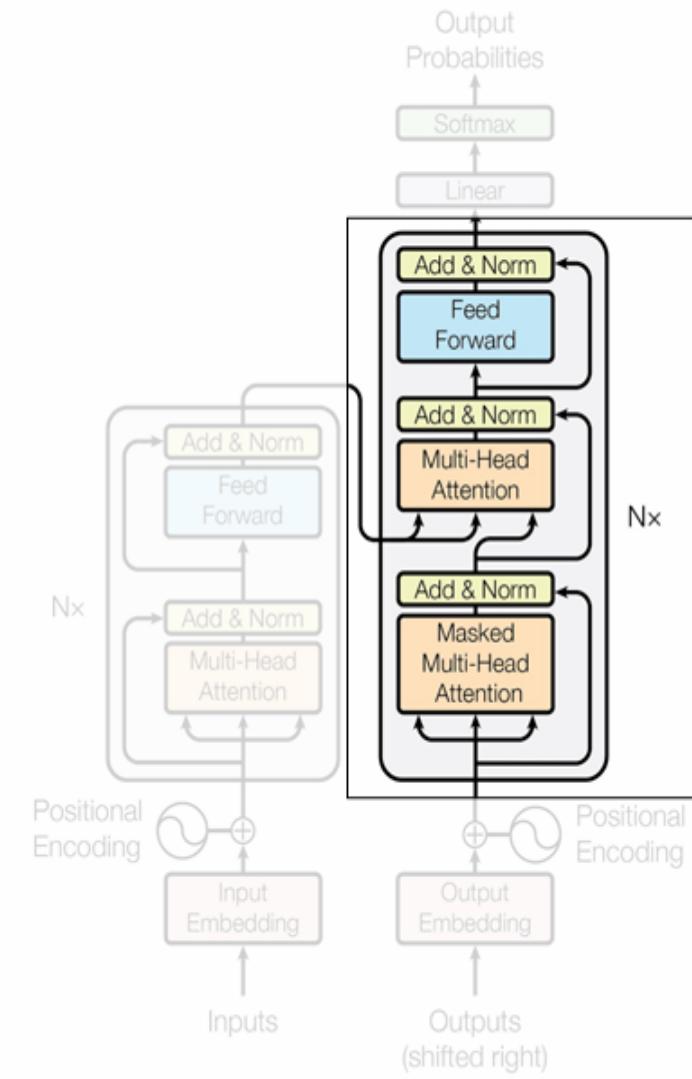
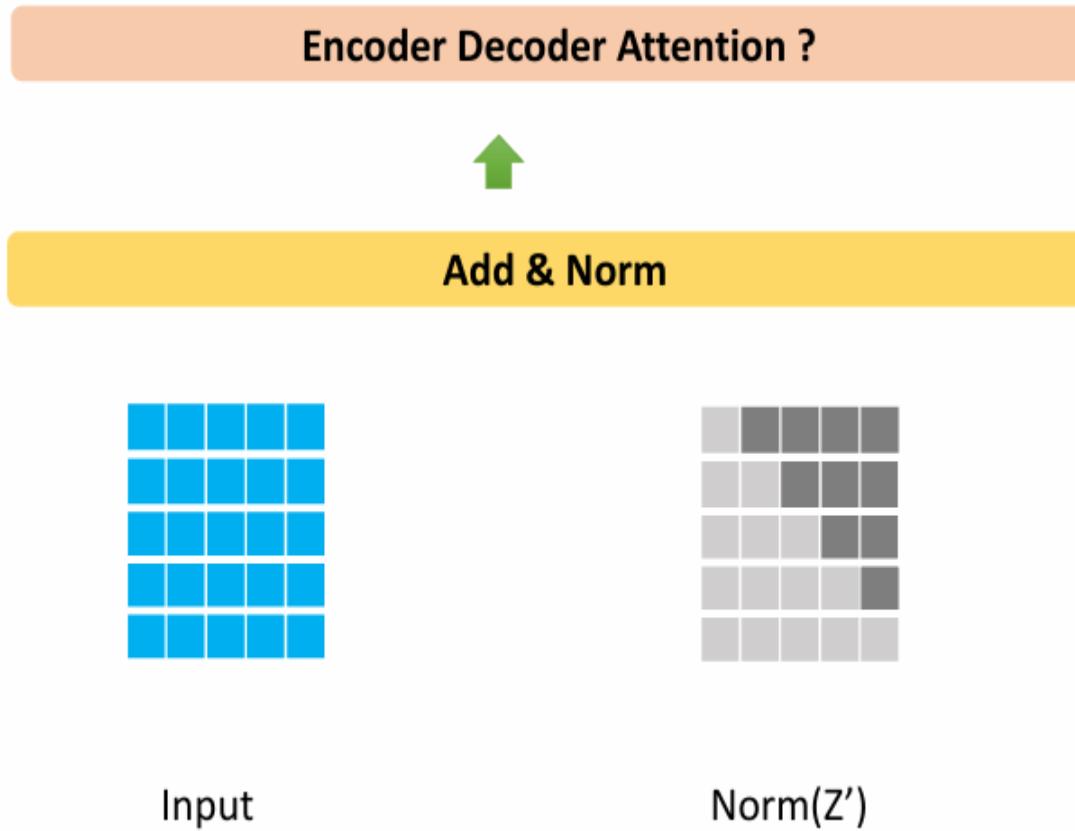
Decoding step by step (using Teacher Forcing)

1	<sos>	- ∞	- ∞	- ∞	- ∞	- ∞	- ∞
2	<sos>	Ich	- ∞	- ∞	- ∞	- ∞	- ∞
3	<sos>	Ich	have	- ∞	- ∞	- ∞	- ∞
4	<sos>	Ich	have	einen	- ∞	- ∞	- ∞
5	<sos>	Ich	have	einen	apfel	- ∞	- ∞
6	<sos>	Ich	have	einen	apfel	gegessen	- ∞
7	<sos>	Ich	have	einen	apfel	gegessen	<eos>

Softmax -> - ∞ -> 0



# Encoder Decoder Attention



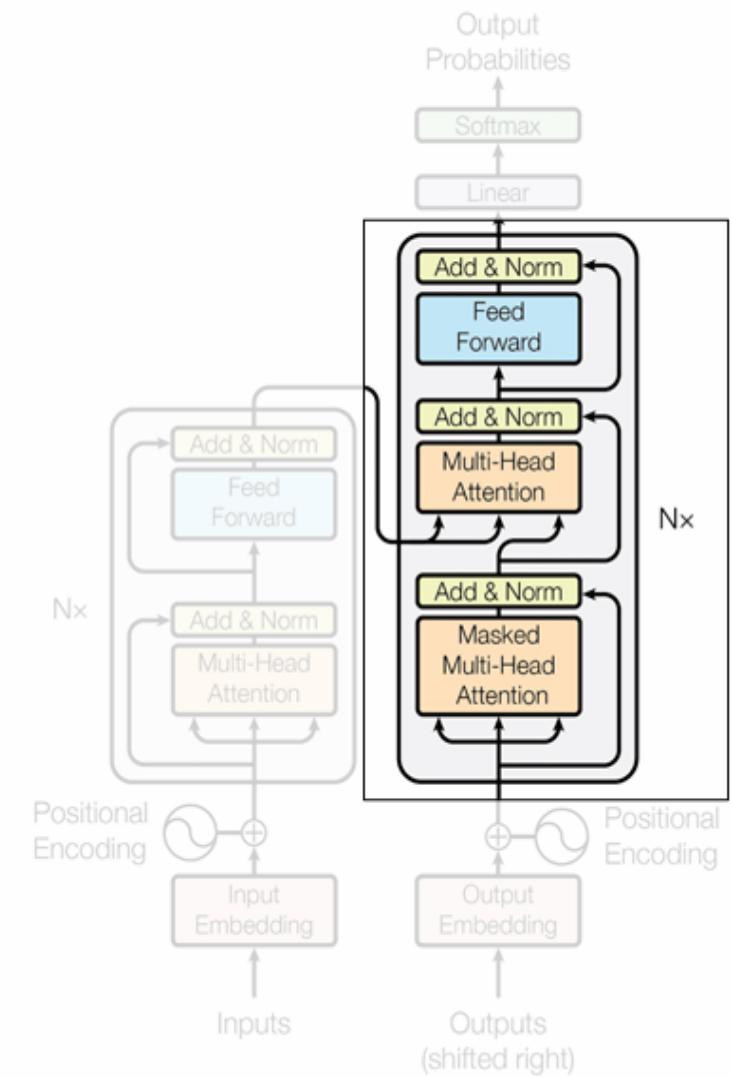
# Encoder Decoder Attention

## Encoder Self Attention

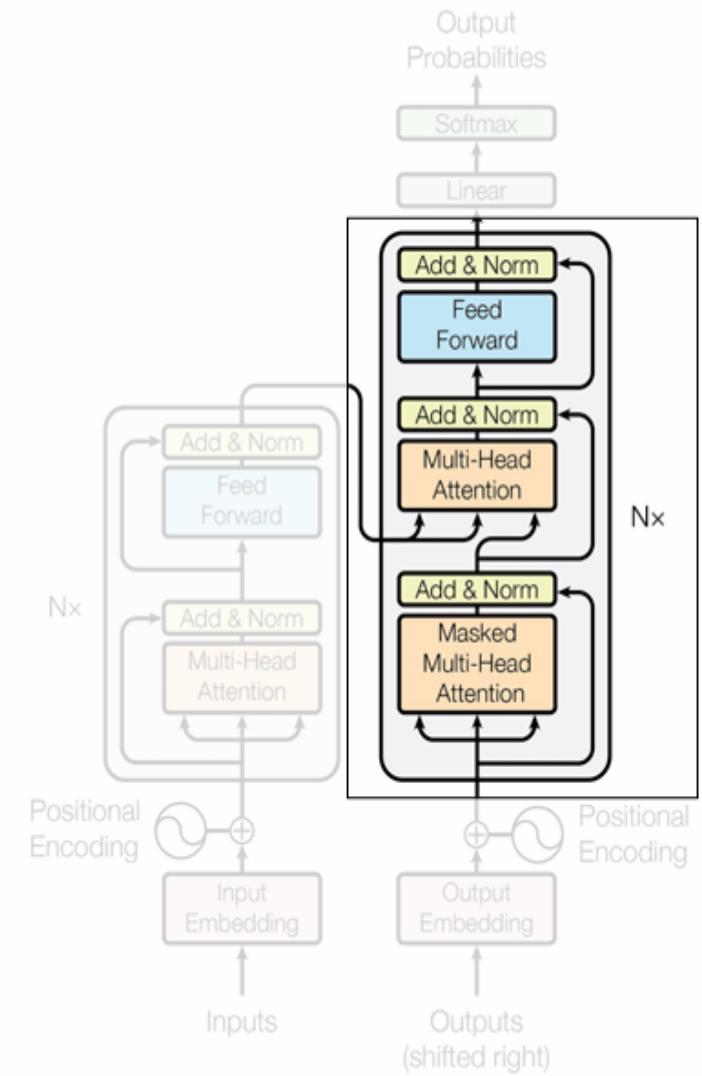
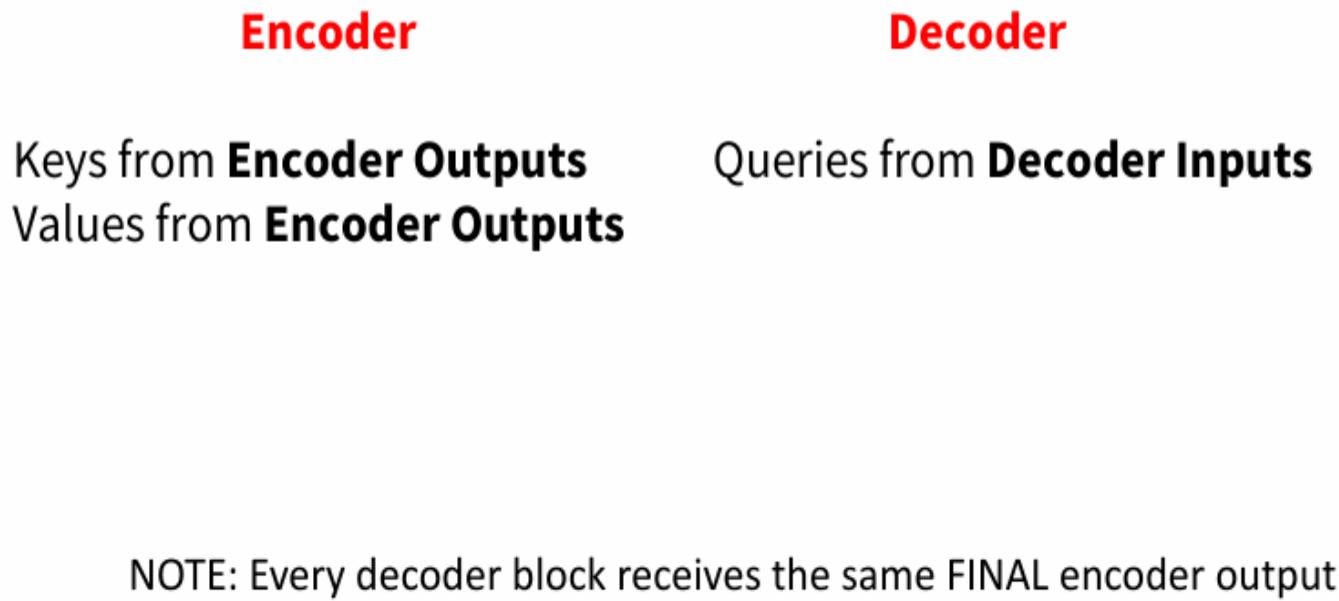
1. Queries from Encoder Inputs
2. Keys from Encoder Inputs
3. Values from Encoder Inputs

## Decoder Masked Self Attention

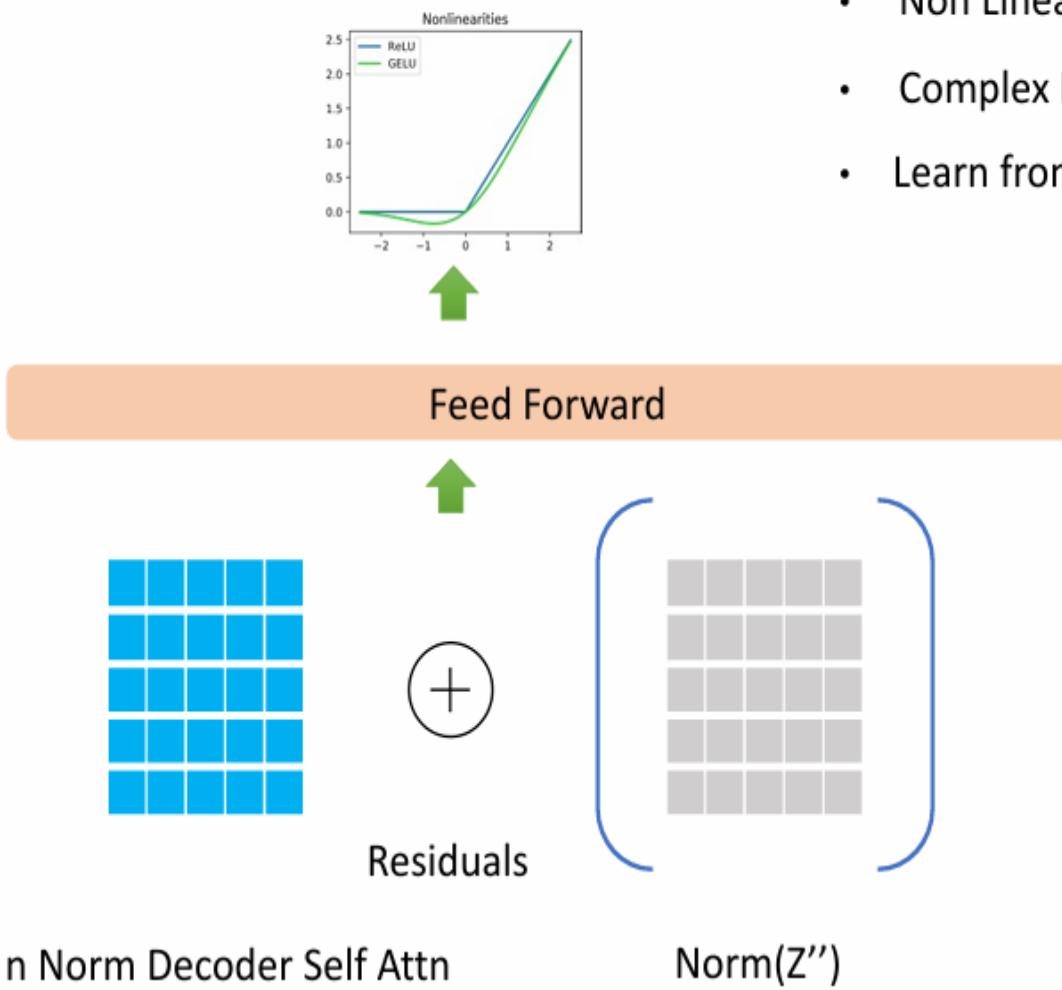
1. Queries from Decoder Inputs
2. Keys from Decoder Inputs
3. Values from Decoder Inputs



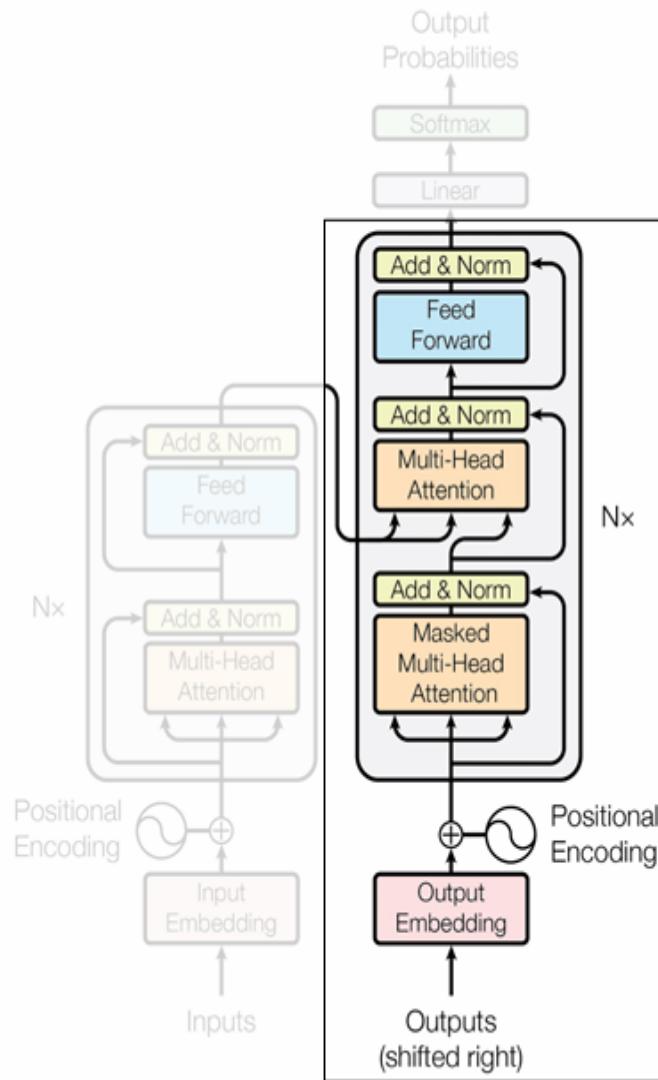
# Encoder Decoder Attention



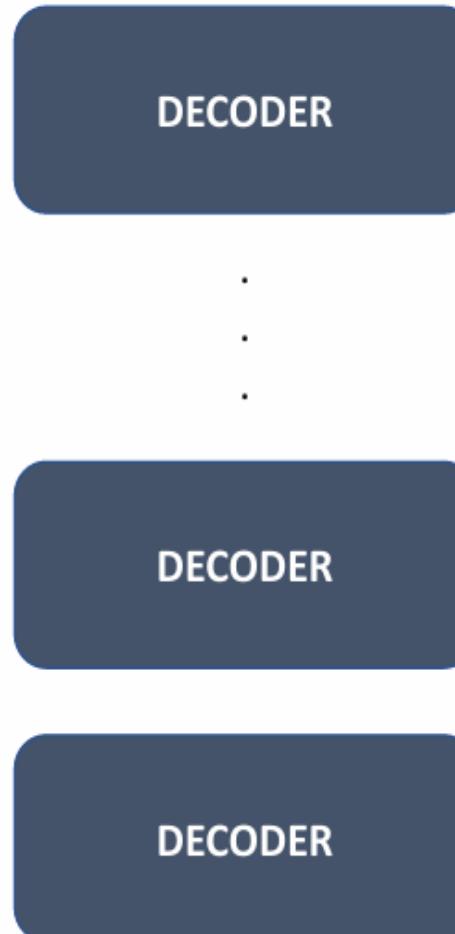
# Encoder Decoder Attention



- Non Linearity
  - Complex Relationships
  - Learn from each other



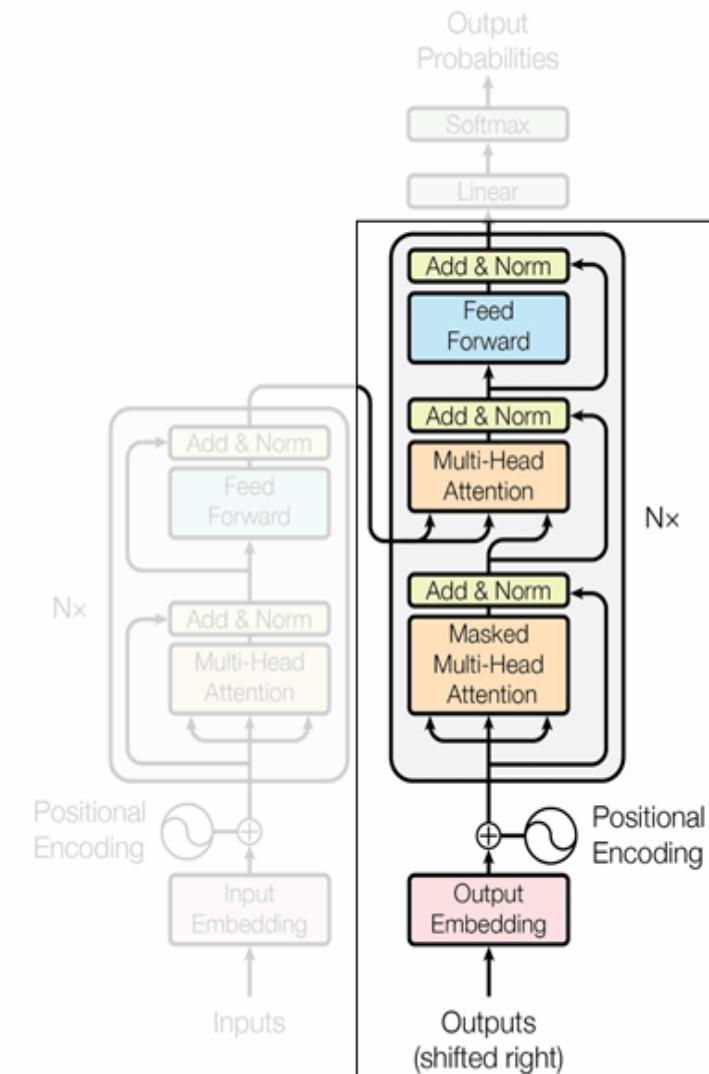
# Decoder



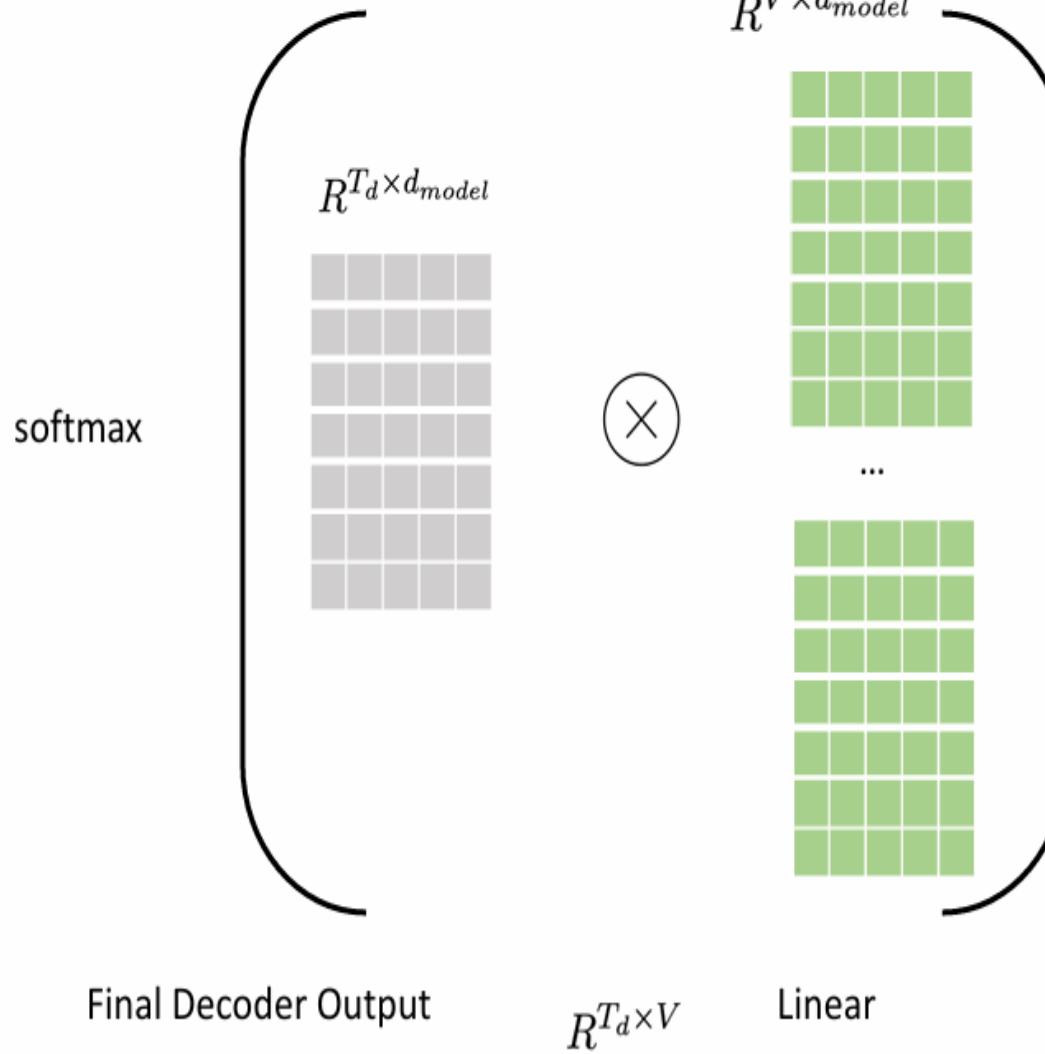
$$R^{T_d \times d_{model}}$$



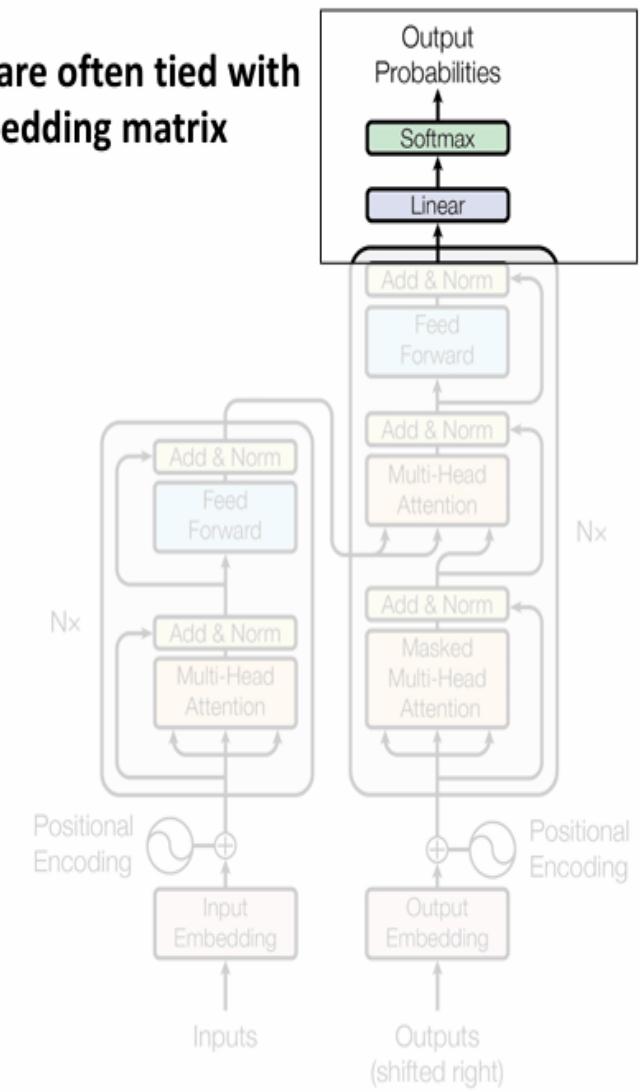
Decoder output



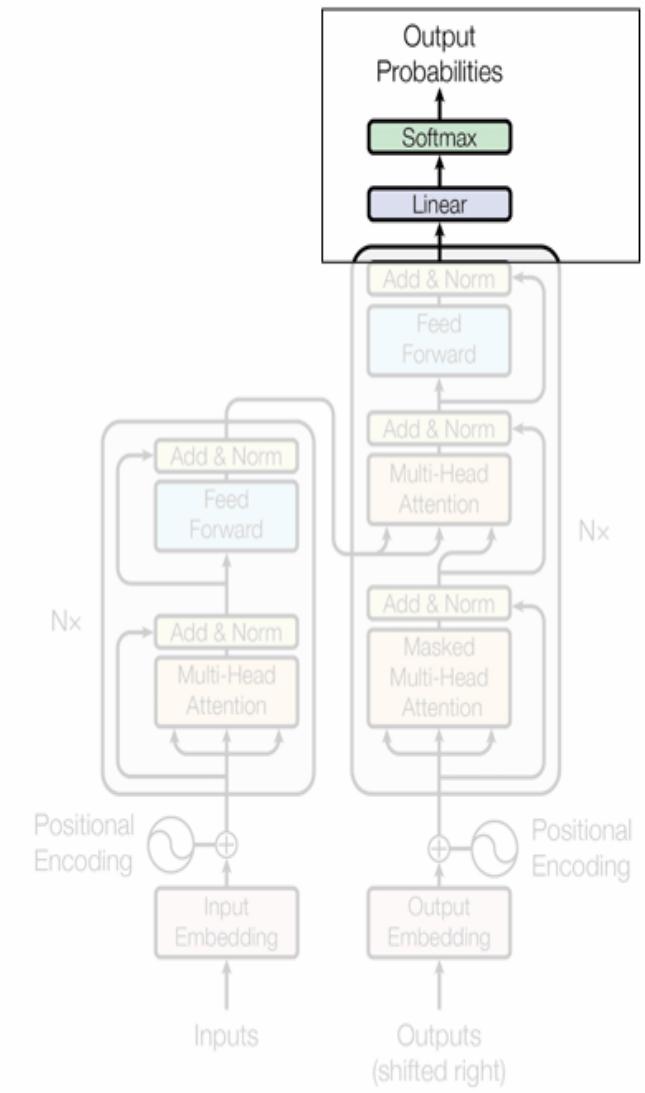
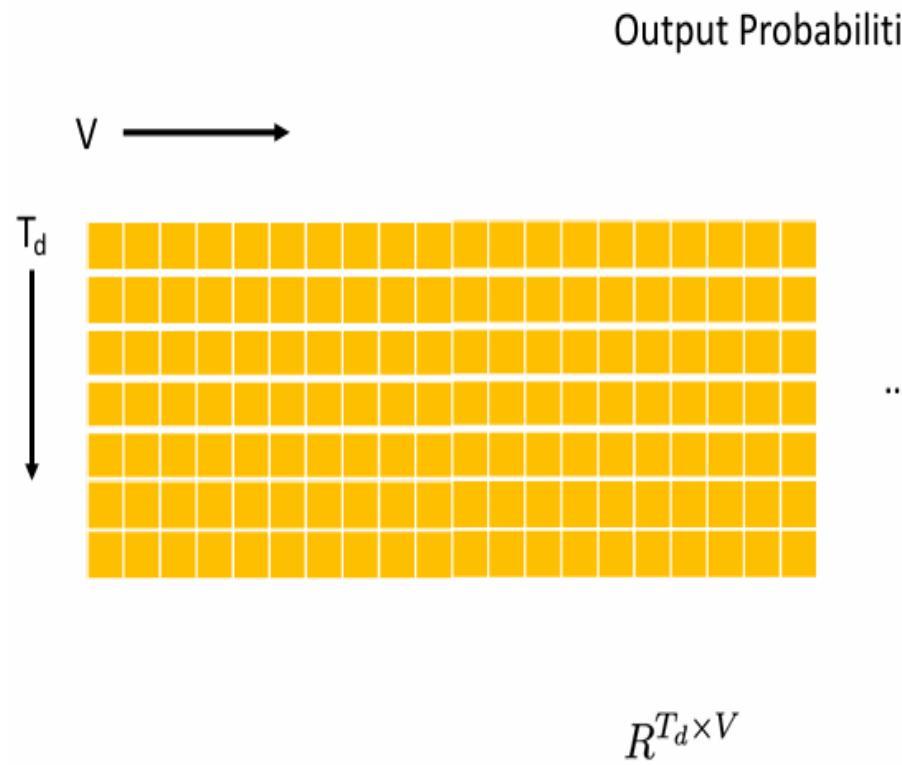
# Linear



Linear weights are often tied with  
input embedding matrix

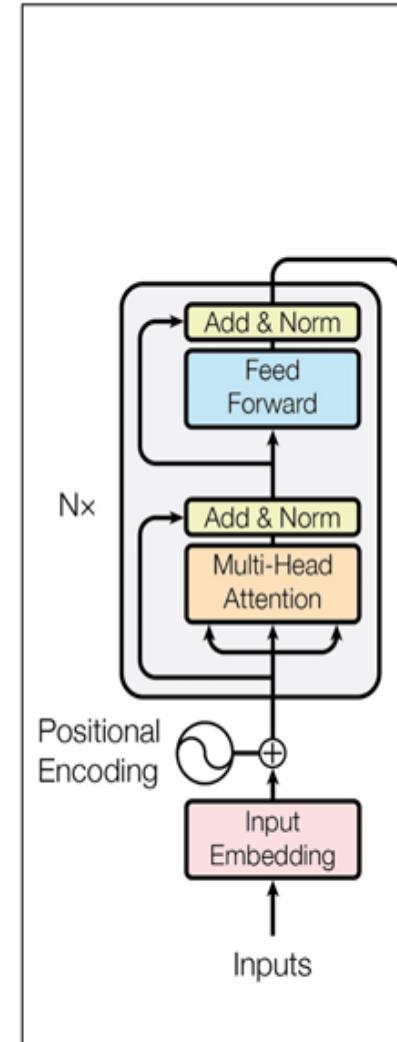


# Softmax



# Transformers

- ✓ Tokenization
- ✓ Input Embeddings
- ✓ Position Encodings
- ✓ Residuals
- ✓ Query
- ✓ Key
- ✓ Value
- ✓ Add & Norm
- ✓ Encoder
- ✓ Decoder
- ✓ Attention
- ✓ Self Attention
- ✓ Multi Head Attention
- ✓ Masked Attention
- ✓ Encoder Decoder Attention
- ✓ Output Probabilities / Logits
- ✓ Softmax
  - Encoder-Decoder models
  - Decoder only models



## Step 1—Defining our Dataset

The dataset used for creating ChatGPT is 570 GB. On the other hand, for our purposes, we will be using a very small dataset to perform numerical calculations visually.

### Dataset (corpus)

I drink and I know things.

When you play the game of thrones, you win or you die.

The true enemy won't wait out the storm, He brings the storm.

Our entire dataset containing only three sentences

## Step 2— Finding Vocab Size

The vocabulary size determines the total number of **unique words** in our dataset. It can be calculated using the below formula, where N is the total number of words in our dataset.

$$\text{vocab size} = \text{count}(\text{set}(N))$$

vocab\_size formula where N is total number of words

In order to find N, we need to break our dataset into individual words.

**Dataset (Corpus)**

I drink and I know things.

When you play the game of thrones, you win or you die.

The true enemy won't wait out the storm, He brings the storm.

$$\rightarrow N = [ \text{I, drink, and, I, Know, things,} \\ \text{When, you, play, the, game, of, thrones, you, win, or, you, die,} \\ \text{The, true, enemy, won't, wait, out, the, storm, He, brings, the, storm} ]$$

calculating variable N

After obtaining N, we perform a set operation to remove duplicates, and then we can count the unique words to determine the vocabulary size.

$$\text{vocab size} = \text{count}(\text{set}(N))$$

↳ set ( I, drink, and, I, Know, things,  
When, you, play, the, game, of, thrones, you, win, or, you, die,  
The, true, enemy, won't, wait, out, the, storm, He, brings, the, storm )

↳ count ( I, drink, and, Know, things, When, you, play, the, game, of,  
thrones, win, or, die, true, enemy, won't, wait, out, storm, He,  
brings )

↳ = 23

finding vocab size

Therefore, the vocabulary size is **23**, as there are 23 unique words in our dataset.

## Step 3 — Encoding

Now, we need to assign a unique number to each unique word.

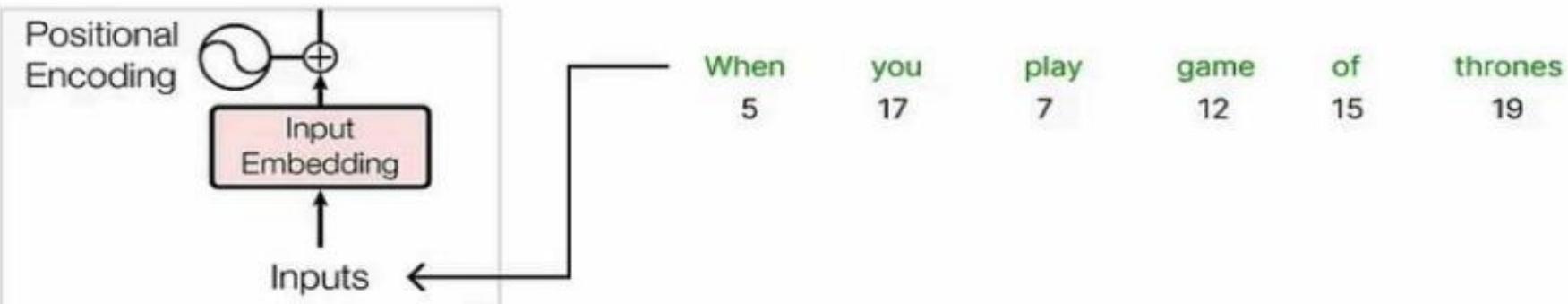
1	2	3	4	5	6	7	8	9	10	11	12
I	drink	things	Know	When	won't	play	out	true	storm	brings	game
13	14	15	16	17	18	19	20	21	22	23	
the	win	of	enemy	you	wait	thrones	and	or	die	He	

encoding our unique words

## Step 4 — Calculating Embedding

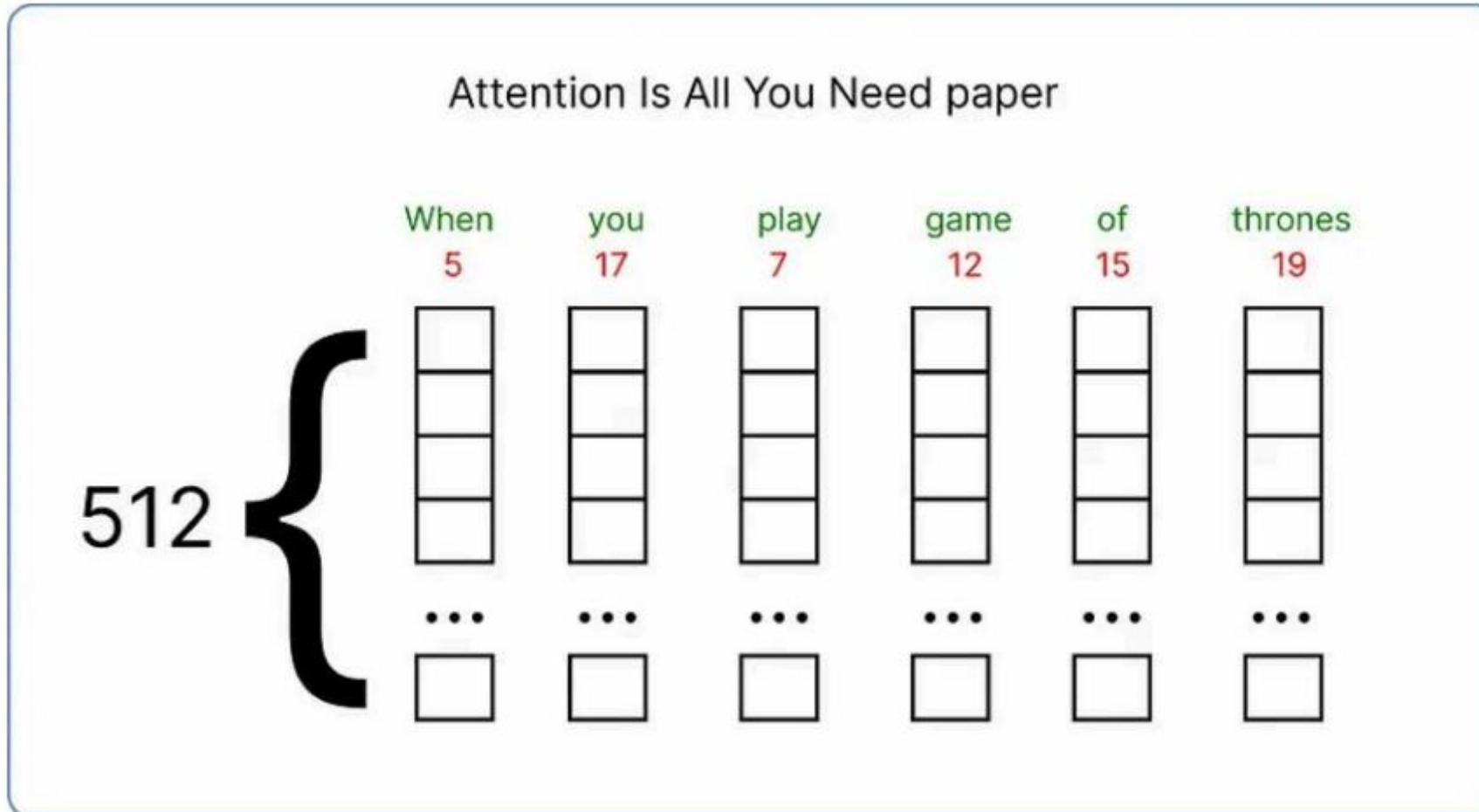
Let's select a sentence from our corpus that will be processed in our transformer architecture.

1	2	3	4	5	6	7	8	9	10	11	12
I	drink	things	Know	When	won't	play	out	true	storm	brings	game
13	14	15	16	17	18	19	20	21	22	23	
the	win	of	enemy	you	wait	thrones	and	or	die	He	



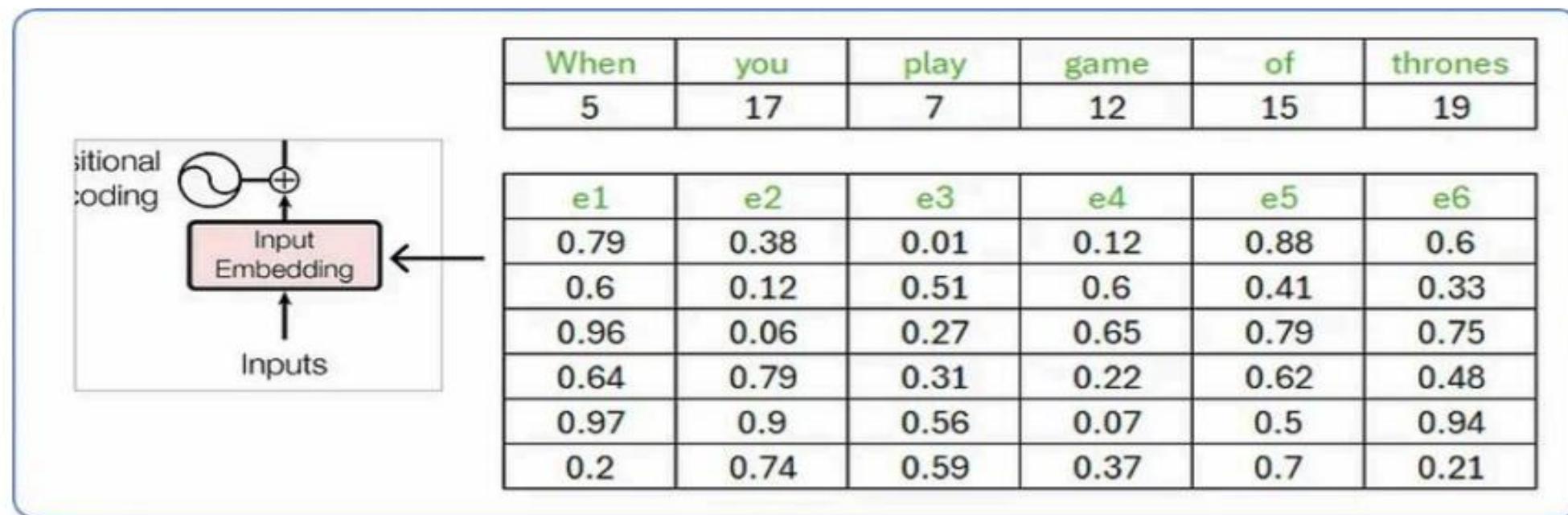
Input sentence for transformer

We have selected our input, and we need to find an embedding vector for it. The original paper uses a **512-dimensional embedding vector** for each input word.



Original Paper uses 512 dimension vector

Since, for our case, we need to work with a smaller dimension of embedding vector to visualize how the calculation is taking place. So, we will be using a dimension of 6 for the embedding vector.



Embedding vectors of our input

## Step 5 — Calculating Positional Embedding

Now we need to find positional embeddings for our input. There are two formulas for positional embedding depending on the position of the  $i$ th value of that embedding vector for each word.

Embedding vector for any word

even position
odd position
even position
odd position
even position
...

For even position

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{\text{model}}})$$

For odd position

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{\text{model}}})$$

Positional Embedding formula

When
5

i	e1	Position	Formula	p1
0	0.79	Even	$\sin(0/10000^{(2*0/6)})$	0
1	0.6	Odd	$\cos(0/10000^{(2*1/6)})$	1
2	0.96	Even	$\sin(0/10000^{(2*2/6)})$	0
3	0.64	Odd	$\cos(0/10000^{(2*3/6)})$	1
4	0.97	Even	$\sin(0/10000^{(2*4/6)})$	0
5	0.2	Odd	$\cos(0/10000^{(2*5/6)})$	1

d (dim) 6

POS 0

Positional Embedding for word: When

Similarly, we can calculate positional embedding for all the words in our input sentence.

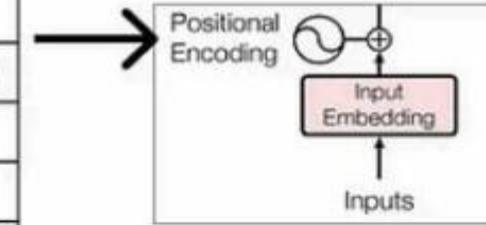
When	you	play	game	of	thrones
5	17	7	12	15	19

i	p1	p2	p3	p4	p5	p6
0	0	0.8415	0.9093	0.1411	-0.7568	-0.9589
1	1	0.0464	0.9957	0.1388	0.1846	0.9732
2	0	0.0022	0.0043	0.0065	0.0086	0.0108
3	1	0.0001	1	0.0003	0.0004	1
4	0	0	0	0	0	0
5	1	0	1	0	0	1

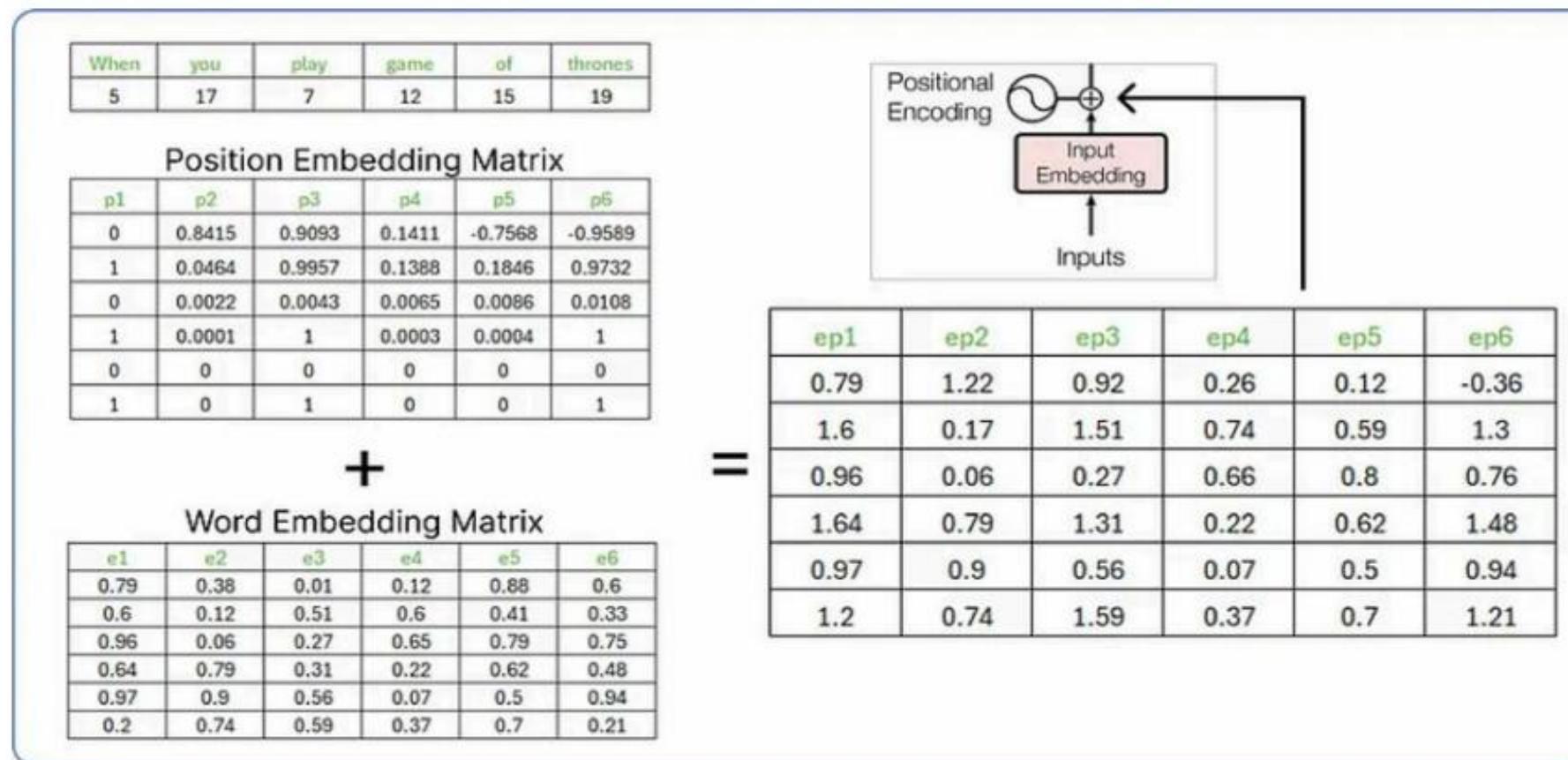
d (dim)	6	6	6	6	6	6
POS	0	1	2	3	4	5



Calculating Positional Embeddings of our input (The calculated values are rounded)

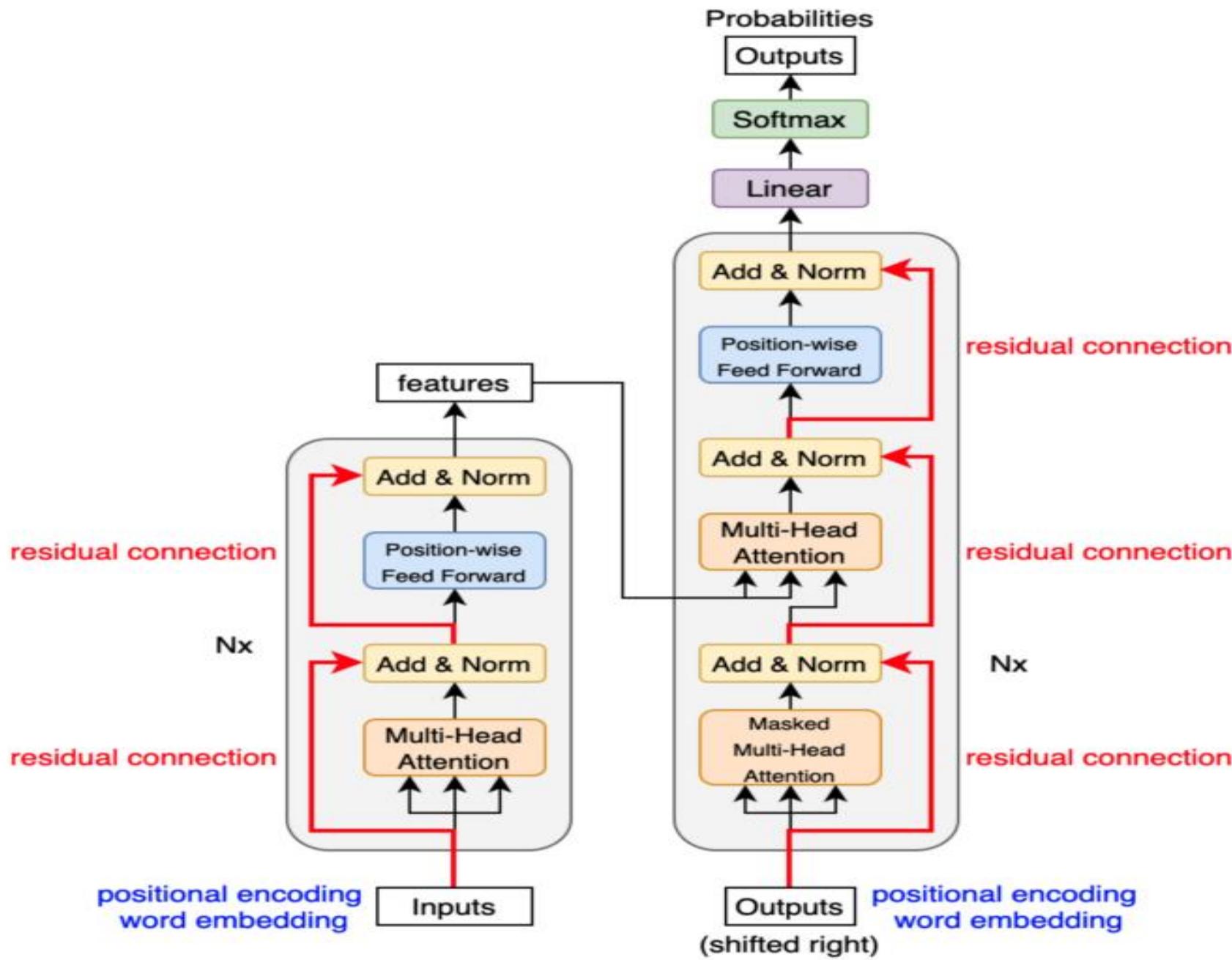
## Step 6 — Concatenating Positional and Word Embeddings

After calculating positional embedding, we need to add word embeddings and positional embeddings.



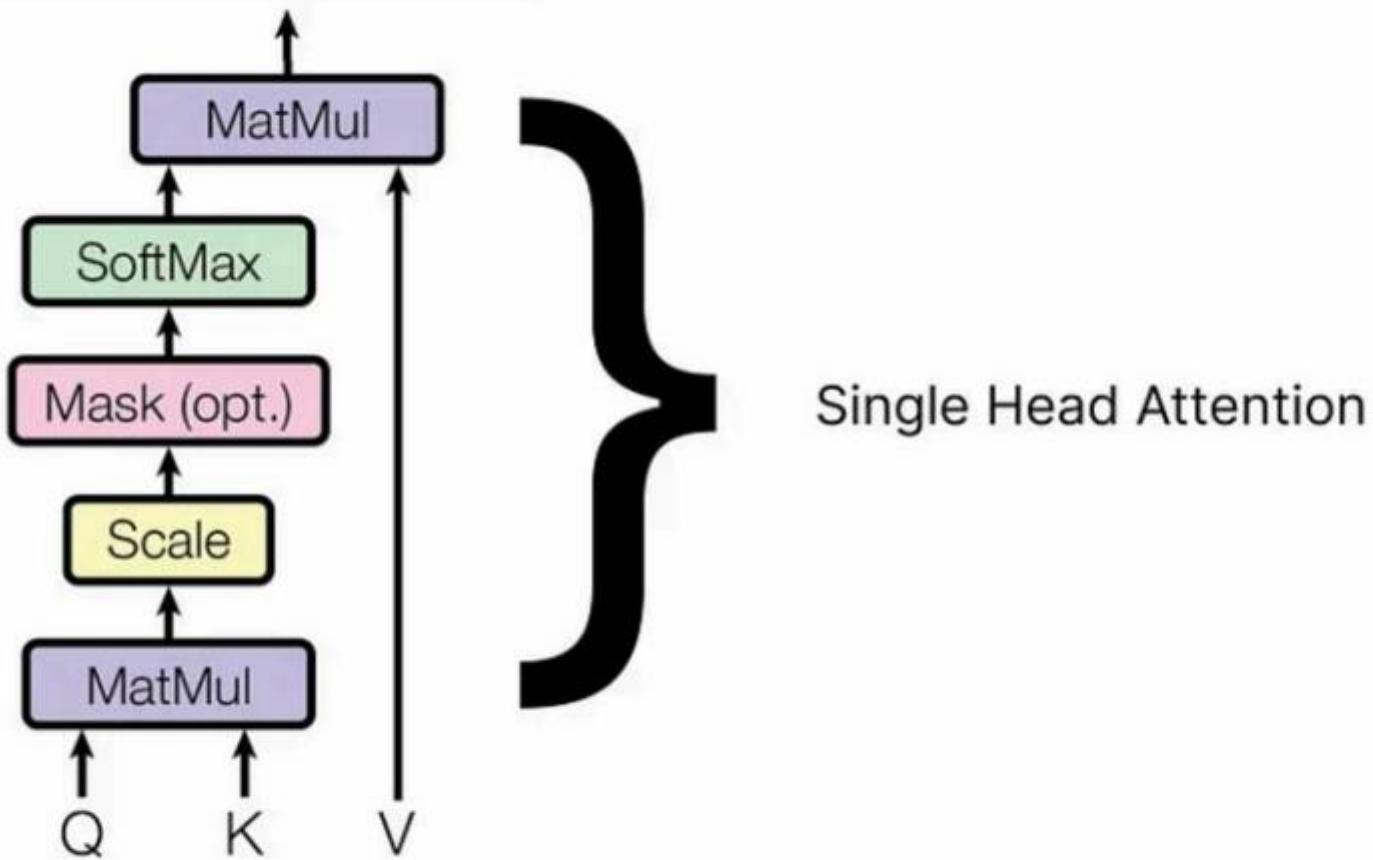
concatenation step

This resultant matrix from combining both matrices (Word embedding matrix and positional embedding matrix) will be considered as an input to the encoder part.



## Step 7 — Multi Head Attention

A multi-head attention is comprised of many single-head attentions. It is up to us how many single heads we need to combine. For example, LLaMA LLM from Meta has used 32 single heads in the encoder architecture. Below is the illustrated diagram of how a single-head attention looks like.



Single Head attention in Transformer

There are three inputs: **query**, **key**, and **value**. Each of these matrices is obtained by multiplying a different set of weights matrix from the **Transpose** of same matrix that we computed earlier by adding the word embedding and positional embedding matrix.

Word Embedding + Positional Embedding

When	0.79	1.6	0.96	1.64	0.97	1.2
you	1.22	0.17	0.06	0.79	0.9	0.74
play	0.92	1.51	0.27	1.31	0.56	1.59
game	0.26	0.74	0.66	0.22	0.07	0.37
of	0.12	0.59	0.8	0.62	0.5	0.7
thrones	-0.36	1.3	0.76	1.48	0.94	1.21

6 x 6

Linear weights for query

0.52	0.45	0.91	0.69
0.05	0.85	0.37	0.83
0.49	0.1	0.56	0.61
0.71	0.64	0.4	0.14
0.76	0.27	0.92	0.67
0.85	0.56	0.57	0.07

X

6 x 4

calculating Query matrix

Similarly, we can compute the **key** and **value** matrices using the same procedure, but the values in the weights matrix must be different for both.

Similarly, we can compute the **key** and **value** matrices using the same procedure, but the values in the weights matrix must be different for both.

Word Embedding + Positional Embedding

When	0.79	1.6	0.96	1.64	0.97	1.2
you	1.22	0.17	0.06	0.79	0.9	0.74
play	0.92	1.51	0.27	1.31	0.56	1.59
game	0.26	0.74	0.66	0.22	0.07	0.37
of	0.12	0.59	0.8	0.62	0.5	0.7
thrones	-0.36	1.3	0.76	1.48	0.94	1.21

6 x 6

Linear weights for key

0.74	0.57	0.21	0.73
0.55	0.16	0.9	0.17
0.25	0.74	0.8	0.98
0.8	0.73	0.2	0.31
0.37	0.96	0.42	0.08
0.28	0.41	0.87	0.86

6 x 4

Word Embedding + Positional Embedding

When	0.79	1.6	0.96	1.64	0.97	1.2
you	1.22	0.17	0.06	0.79	0.9	0.74
play	0.92	1.51	0.27	1.31	0.56	1.59
game	0.26	0.74	0.66	0.22	0.07	0.37
of	0.12	0.59	0.8	0.62	0.5	0.7
thrones	-0.36	1.3	0.76	1.48	0.94	1.21

6 x 6

Linear weights for value

0.62	0.07	0.7	0.95
0.2	0.97	0.61	0.35
0.57	0.8	0.61	0.5
0.67	0.35	0.98	0.54
0.47	0.83	0.34	0.94
0.6	0.69	0.13	0.98

6 x 4

So, after multiplying matrices, the resultant **query**, **key**, and **values** are obtained:

Query

3.88	3.8	4.08	3.42
2.55	1.86	2.77	1.78
3.39	3.6	3.49	2.72
1.02	1.18	1.24	1.3
1.9	1.56	1.88	1.53
3.04	2.9	2.73	2.22

6 x 4

key

3.71	4.04	4.15	3.41
2.18	2.51	1.64	1.93
3.28	3.11	3.65	3.01
1.07	1.13	1.64	1.35
1.49	1.97	2.14	1.81
2.51	3.04	3.45	2.28

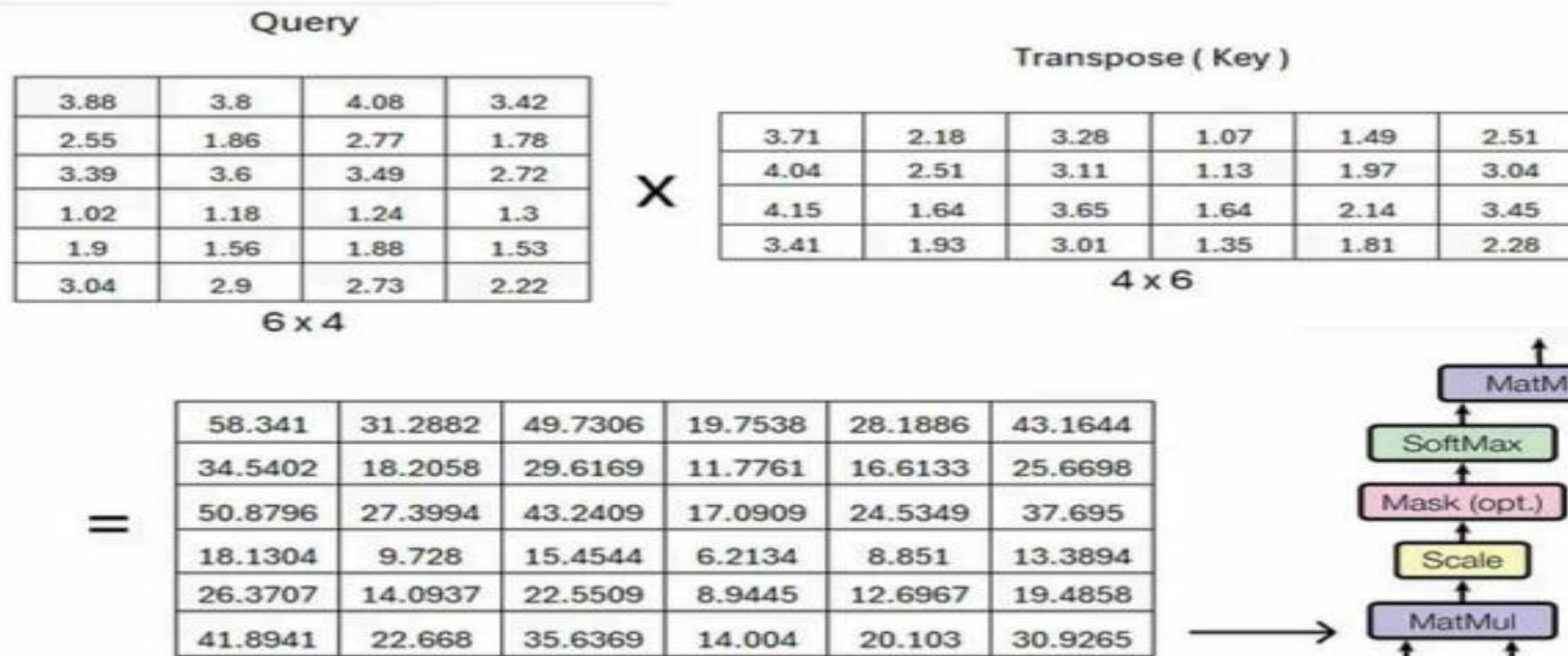
6 x 4

value

3.88	3.8	4.08	3.42
2.55	1.86	2.77	1.78
3.39	3.6	3.49	2.72
1.02	1.18	1.24	1.3
1.9	1.56	1.88	1.53
3.04	2.9	2.73	2.22

6 x 4

Now that we have all three matrices, let's start calculating single-head attention step by step.



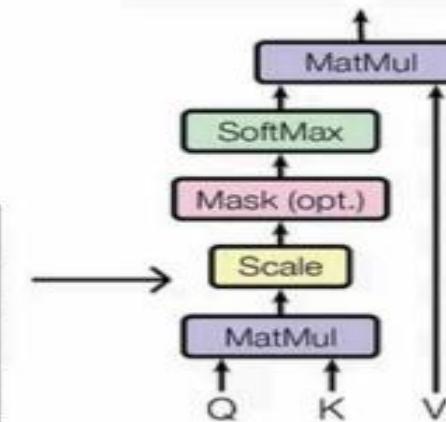
matrix multiplication between Query and Key

For scaling the resultant matrix, we have to reuse the dimension of our embedding vector, which is 6.

58.341	31.2882	49.7306	19.7538	28.1886	43.1644
34.5402	18.2058	29.6169	11.7761	16.6133	25.6698
50.8796	27.3994	43.2409	17.0909	24.5349	37.695
18.1304	9.728	15.4544	6.2134	8.851	13.3894
26.3707	14.0937	22.5509	8.9445	12.6967	19.4858
41.8941	22.668	35.6369	14.004	20.103	30.9265

$$\sqrt{d_k} \quad \text{where } d \text{ (dimension) is 6}$$

23.81721	12.77409	20.30219	8.062904	11.50852	17.62
14.1009	7.434201	12.09231	4.809165	6.781004	10.47973
20.77167	11.186	17.65266	6.976963	10.01433	15.39096
7.401542	3.972256	6.307436	2.535222	3.612997	5.466445
10.76551	5.752218	9.205999	3.64974	5.184753	7.956759
17.10152	9.254989	14.54997	5.715476	8.205791	12.62712



scaling the resultant matrix with dimension 5

23.82	12.77	20.3	8.06	11.51	17.62
14.1	7.43	12.09	4.81	6.78	10.48
20.77	11.19	17.65	6.98	10.01	15.39
7.4	3.97	6.31	2.54	3.61	5.47
10.77	5.75	9.21	3.65	5.18	7.96
17.1	9.25	14.55	5.72	8.21	12.63

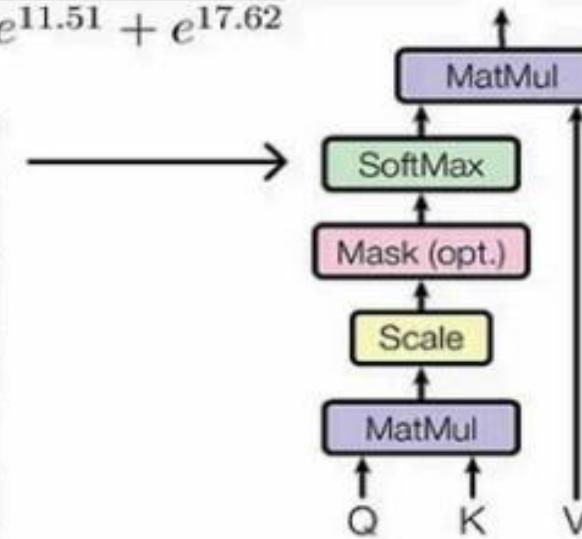
SoftMax

$$s(x_i) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$$

→ softmax(23.82) =  $\frac{e^{23.82}}{e^{23.82} + e^{12.77} + e^{20.3} + e^{8.06} + e^{11.51} + e^{17.62}}$

=

0.9693	0	0.0287	0	0	0.002
0.86	0.0011	0.1152	0.0001	0.0006	0.023
0.9534	0.0001	0.0421	0	0	0.0044
0.6476	0.021	0.2177	0.005	0.0146	0.094
0.7803	0.0052	0.164	0.0006	0.0029	0.047
0.9174	0.0004	0.0716	0	0.0001	0.0105



Applying softmax on resultant matrix

$$\text{softmax}\left(\frac{Q \cdot K^T}{\sqrt{d_k}}\right)$$

0.9693	0	0.0287	0	0	0.002
0.86	0.0011	0.1152	0.0001	0.0006	0.023
0.9534	0.0001	0.0421	0	0	0.0044
0.6476	0.021	0.2177	0.005	0.0146	0.094
0.7803	0.0052	0.164	0.0006	0.0029	0.047
0.9174	0.0004	0.0716	0	0.0001	0.0105

6 x 6

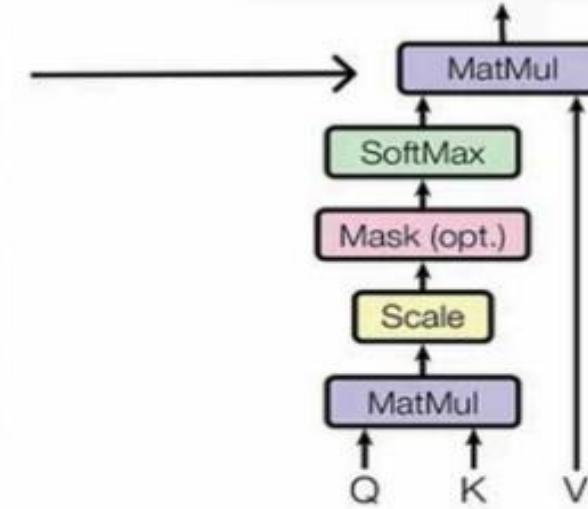
Value

3.88	3.8	4.08	3.42
2.55	1.86	2.77	1.78
3.39	3.6	3.49	2.72
1.02	1.18	1.24	1.3
1.9	1.56	1.88	1.53
3.04	2.9	2.73	2.22

6 x 4

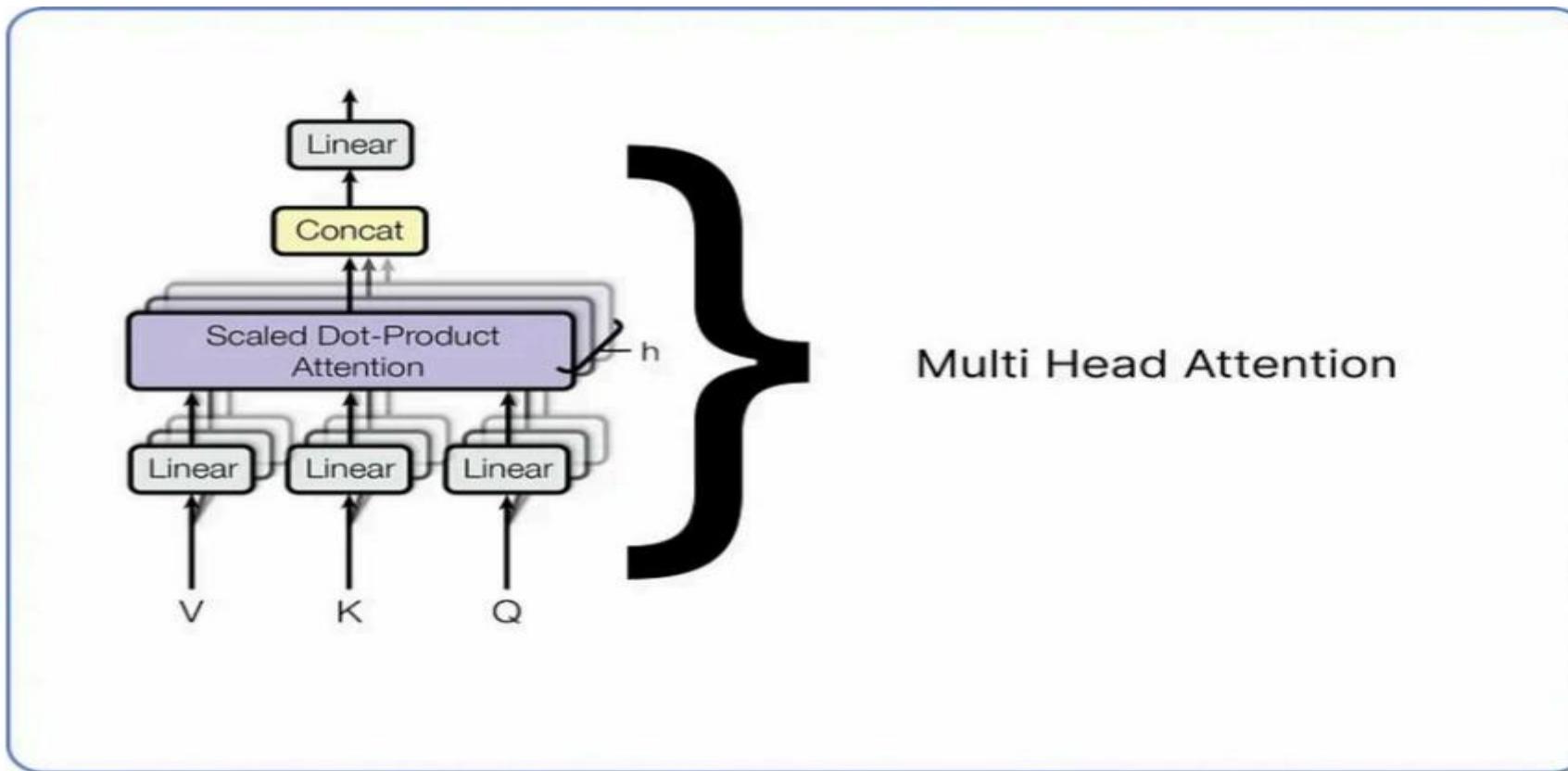
=

3.864257	3.79246	4.060367	3.39751
3.801295	3.75252	3.977937	3.30861
3.855542	3.787426	4.04909	3.385086
3.622841	3.584936	3.750419	3.081834
3.745786	3.706744	3.904894	3.233519
3.835366	3.77523	4.022837	3.356435



calculating the final matrix of single head attention

We have calculated single-head attention, while multi-head attention comprises many single-head attentions, as I stated earlier. Below is a visual of how it looks like:

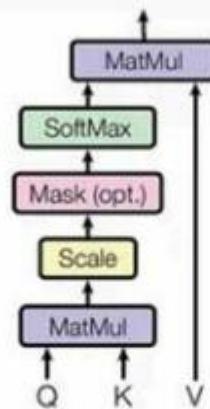


Each single-head attention has three inputs: **query**, **key**, and **value**, and each three have a different set of weights. Once all single-head attentions output their resultant matrices, they will all be concatenated, and the final concatenated matrix is once again transformed linearly by multiplying it with a set of weights matrix initialized with random values, which will later get updated when the transformer starts training.

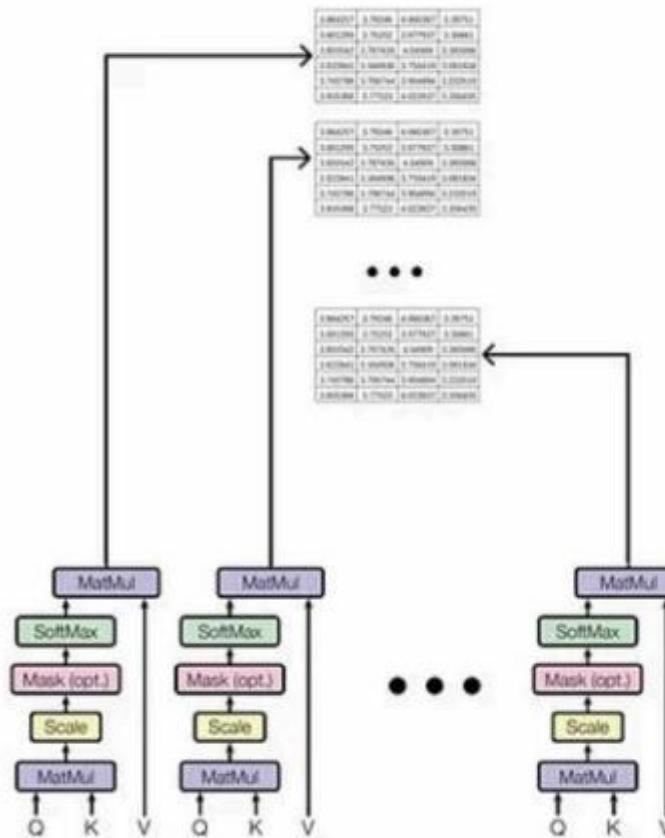
Since, in our case, we are considering a single-head attention, but this is how it looks if we are working with multi-head attention.

## Single Head Attention Our Case

3.864257	3.79246	4.060367	3.39751
3.801295	3.75252	3.977937	3.30861
3.855542	3.787426	4.04909	3.385086
3.622841	3.584936	3.750419	3.081834
3.745786	3.706744	3.904894	3.233519
3.835366	3.77523	4.022837	3.356435



## Multi Head Attention (N Heads) Real world Case Concatenation

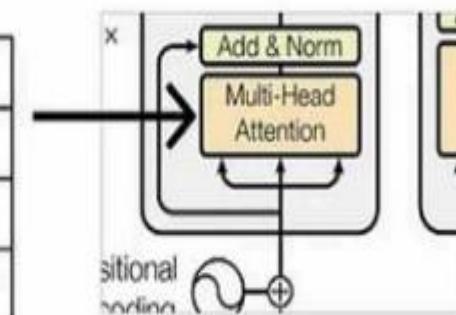


## Single Head attention vs Multi Head attention

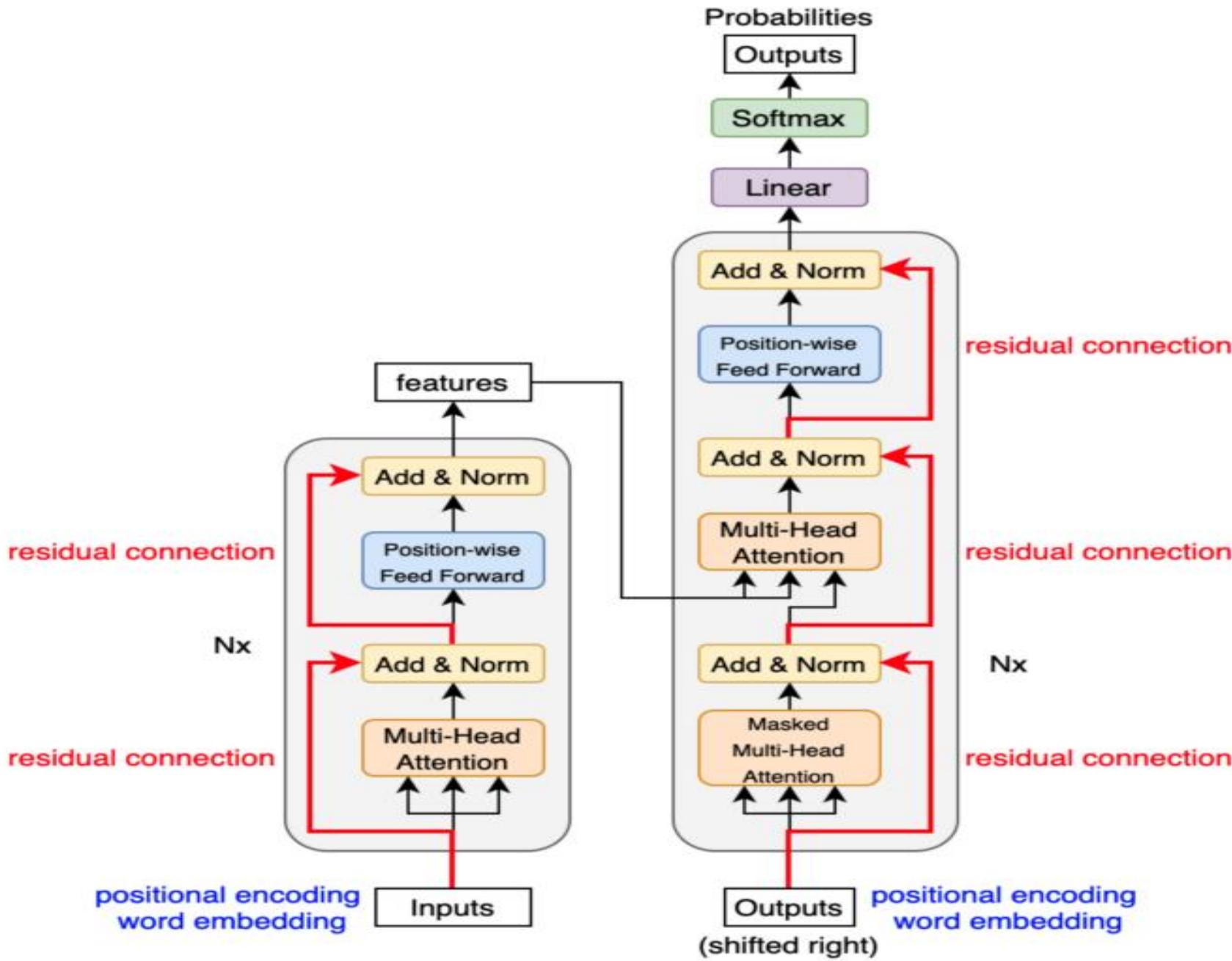
### Output of Multi Head attention

10.84	9.45	7.33	7.8	6.09	7.66
10.65	9.28	7.22	7.67	5.99	7.51
10.83	9.43	7.33	7.79	6.08	7.65
10.08	8.77	6.85	7.25	5.67	7.09
10.48	9.12	7.11	7.54	5.89	7.38
10.77	9.38	7.29	7.75	6.05	7.6

6 x 6



Output matrix of multi head attention



## Step 8 — Adding and Normalizing

Once we obtain the resultant matrix from multi-head attention, we have to add it to our original matrix. Let's do it first.

Word Embedding + Positional Embedding

When	0.79	1.6	0.96	1.64	0.97	1.2
you	1.22	0.17	0.06	0.79	0.9	0.74
play	0.92	1.51	0.27	1.31	0.56	1.59
game	0.26	0.74	0.66	0.22	0.07	0.37
of	0.12	0.59	0.8	0.62	0.5	0.7
thrones	-0.36	1.3	0.76	1.48	0.94	1.21

6 x 6



Output of Multi Head attention

10.84	9.45	7.33	7.8	6.09	7.66
10.65	9.28	7.22	7.67	5.99	7.51
10.83	9.43	7.33	7.79	6.08	7.65
10.08	8.77	6.85	7.25	5.67	7.09
10.48	9.12	7.11	7.54	5.89	7.38
10.77	9.38	7.29	7.75	6.05	7.6

6 x 6

11.63	11.05	8.29	9.44	7.06	8.86
11.87	9.45	7.28	8.46	6.89	8.25
11.75	10.94	7.6	9.1	6.64	9.24
10.34	9.51	7.51	7.47	5.74	7.46
10.6	9.71	7.91	8.16	6.39	8.08
10.41	10.68	8.05	9.23	6.99	8.81

Adding matrices to perform add and norm step

To normalize the above matrix, we need to compute the mean and standard deviation row-wise for each row.

$$\text{mean} = \frac{\sum_{i=1}^N X_i}{N}$$

$$\text{standard dev.} = \sqrt{\frac{\sum_{i=1}^N (X_i - \mu)^2}{N}}$$

Row Wise Implementation

Mean      Standard Deviation

11.63	11.05	8.29	9.44	7.06	8.86
11.87	9.45	7.28	8.46	6.89	8.25
11.75	10.94	7.6	9.1	6.64	9.24
10.34	9.51	7.51	7.47	5.74	7.46
10.6	9.71	7.91	8.16	6.39	8.08
10.41	10.68	8.05	9.23	6.99	8.81

→	9.26	1.57
→	8.56	1.64
→	9.04	1.76
→	7.86	1.51
→	8.37	1.35
→	8.93	1.28

calculating meand and std.

we subtract each value of the matrix by the corresponding row mean and divide it by the corresponding standard deviation.

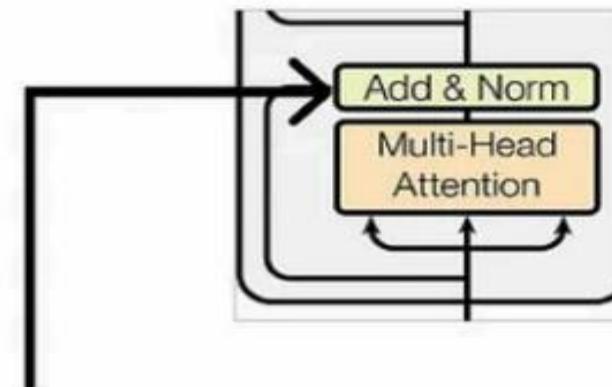
11.63	11.05	8.29	9.44	7.06	8.86
11.87	9.45	7.28	8.46	6.89	8.25
11.75	10.94	7.6	9.1	6.64	9.24
10.34	9.51	7.51	7.47	5.74	7.46
10.6	9.71	7.91	8.16	6.39	8.08
10.41	10.68	8.05	9.23	6.99	8.81

Mean	Std
9.26	1.57
8.56	1.64
9.04	1.76
7.86	1.51
8.37	1.35
8.93	1.28

$$\frac{\text{value} - \text{mean}}{\text{std} + \text{error}} = \frac{11.63 - 9.26}{1.57 + 0.0001}$$



1.51	1.14	-0.62	0.11	-1.4	-0.25
2.02	0.54	-0.78	-0.06	-1.02	-0.19
1.54	1.08	-0.82	0.03	-1.36	0.11
1.64	1.09	-0.23	-0.26	-1.4	-0.26
1.65	0.99	-0.34	-0.16	-1.47	-0.21
1.16	1.37	-0.69	0.23	-1.52	-0.09



normalizing the resultant matrix

## Step 9 — Feed Forward Network

After normalizing the matrix, it will be processed through a feedforward network. We will be using a very basic network that contains only one linear layer and one ReLU activation function layer. This is how it looks like visually:

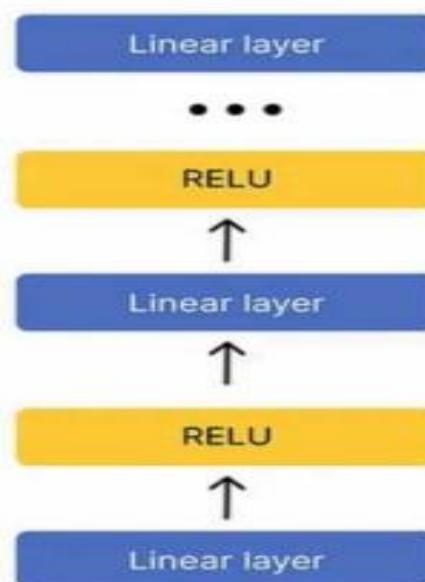
$$\text{ReLU}(x) = \max(0, x)$$

$$\text{Linear Layer} = X \cdot W + b$$

our case (one linear layer)



## Real world case (multiple layers)



First, we need to calculate the linear layer by multiplying our last calculated matrix with a random set of weights matrix, which will be updated when the transformer starts learning, and adding the resultant matrix to a bias matrix that also contains random values.

Matrix after add and norm step

1.51	1.14	-0.62	0.11	-1.4	-0.25
2.02	0.54	-0.78	-0.06	-1.02	-0.19
1.54	1.08	-0.82	0.03	-1.36	0.11
1.64	1.09	-0.23	-0.26	-1.4	-0.26
1.65	0.99	-0.34	-0.16	-1.47	-0.21
1.16	1.37	-0.69	0.23	-1.52	-0.09

6 x 6

X

W

0.5	0.05	0.97	0.22	0.56	0.02
0.17	0.52	0.63	0.48	0.06	0.6
0.53	0.87	0.47	0.1	0.31	0.79
0.83	0.58	0.38	0.09	0.64	0.25
0.81	0.85	0.74	0.35	0.31	0.53
0.25	0.31	0.22	0.77	0.57	0.85

6 x 6

 $X \cdot W$ 

0.49	1.07	0.84	0.14	0.22	0.7
0.24	1.26	1.11	0.12	0.46	0.97
0.53	1.18	-0.82	0.39	0.33	0.59
0.53	0.97	0.98	0.15	0.16	0.52
0.56	1.11	-0.87	0.11	0.2	0.64
0.62	1.02	0.61	0.26	0.14	0.52

6 x 6

+

Bias

b1	b2	b3	b4	b5	b6
0.42	0.18	0.25	0.42	0.35	0.45

=

0.91	1.25	1.09	0.56	0.57	1.15
0.66	1.44	1.36	0.54	0.81	1.42
0.95	1.36	-0.57	0.81	0.68	1.04
0.95	1.15	1.23	0.57	0.51	0.97
0.98	1.29	-0.62	0.53	0.55	1.09
1.04	1.2	0.86	0.68	0.49	0.97

6 x 6

Calculating Linear Layer

After calculating the linear layer, we need to pass it through the ReLU layer and use its formula.

$$\text{ReLU}(x) = \max(0, x)$$

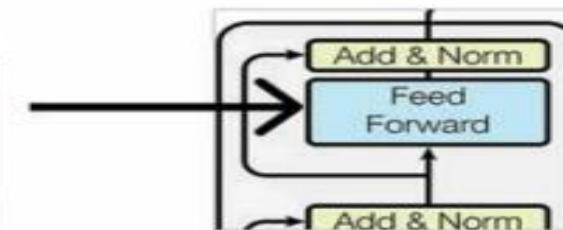
0.91	1.25	1.09	0.56	0.57	1.15
0.66	1.44	1.36	0.54	0.81	1.42
0.95	1.36	-0.57	0.81	0.68	1.04
0.95	1.15	1.23	0.57	0.51	0.97
0.98	1.29	-0.62	0.53	0.55	1.09
1.04	1.2	0.86	0.68	0.49	0.97

6 × 6

→  $\max(0, 0.91)$



0.91	1.25	1.09	0.56	0.57	1.15
0.66	1.44	1.36	0.54	0.81	1.42
0.95	1.36	0	0.81	0.68	1.04
0.95	1.15	1.23	0.57	0.51	0.97
0.98	1.29	0	0.53	0.55	1.09
1.04	1.2	0.86	0.68	0.49	0.97



## **Step 10 — Adding and Normalizing Again**

Once we obtain the resultant matrix from feed forward network, we have to add it to the matrix that is obtained from previous add and norm step, and then normalizing it using the row wise mean and standard deviation.

Matrix from Feed Forward Network

0.91	1.25	1.09	0.56	0.57	1.15
0.66	1.44	1.36	0.54	0.81	1.42
0.95	1.36	0	0.81	0.68	1.04
0.95	1.15	1.23	0.57	0.51	0.97
0.98	1.29	0	0.53	0.55	1.09
1.04	1.2	0.86	0.68	0.49	0.97



Matrix from Previous Add and Norm Step

1.51	1.14	-0.62	0.11	-1.4	-0.25
2.02	0.54	-0.78	-0.06	-1.02	-0.19
1.54	1.08	-0.82	0.03	-1.36	0.11
1.64	1.09	-0.23	-0.26	-1.4	-0.26
1.65	0.99	-0.34	-0.16	-1.47	-0.21
1.16	1.37	-0.69	0.23	-1.52	-0.09

=

2.42	2.39	0.47	0.67	-0.83	0.9
2.68	1.98	0.58	0.48	-0.21	1.23
2.49	2.44	-0.82	0.84	-0.68	1.15
2.59	2.24	1	0.31	-0.89	0.71
2.63	2.28	-0.34	0.37	-0.92	0.88
2.2	2.57	0.17	0.91	-1.03	0.88

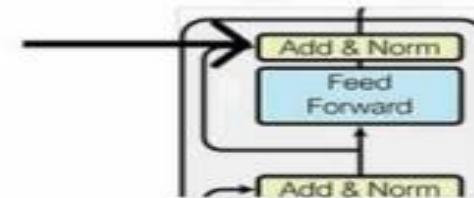


Mean Std

1.0033	1.103534
1.1233	1.214349
0.9033	1.301837
0.9933	1.289055
0.8167	1.306016
0.95	1.320773

=

1.28	1.26	-0.48	-0.3	-1.66	-0.09
1.28	0.71	-0.45	-0.53	-1.1	0.09
1.22	1.18	-1.32	-0.05	-1.22	0.19
1.24	0.97	0.01	-0.53	-1.46	-0.22
1.39	1.12	-0.89	-0.34	-1.33	0.05
0.95	1.23	-0.59	-0.03	-1.5	-0.05



Add and Norm after Feed Forward Network

## 1. Input Embedding & Positional Encoding

- Since Transformers don't process data sequentially, **Positional Encoding** is added to retain order information.

## 2. Multi-Head Attention

- Instead of a single Self-Attention layer, we use **multiple attention heads**, allowing the model to capture various relationships simultaneously.

## 3. Feedforward Network

- A simple MLP applied to each token independently.

## 4. Residual Connections & Layer Normalization

- Helps in stabilizing training and improving convergence.

## Transformer Structure

A Transformer consists of:

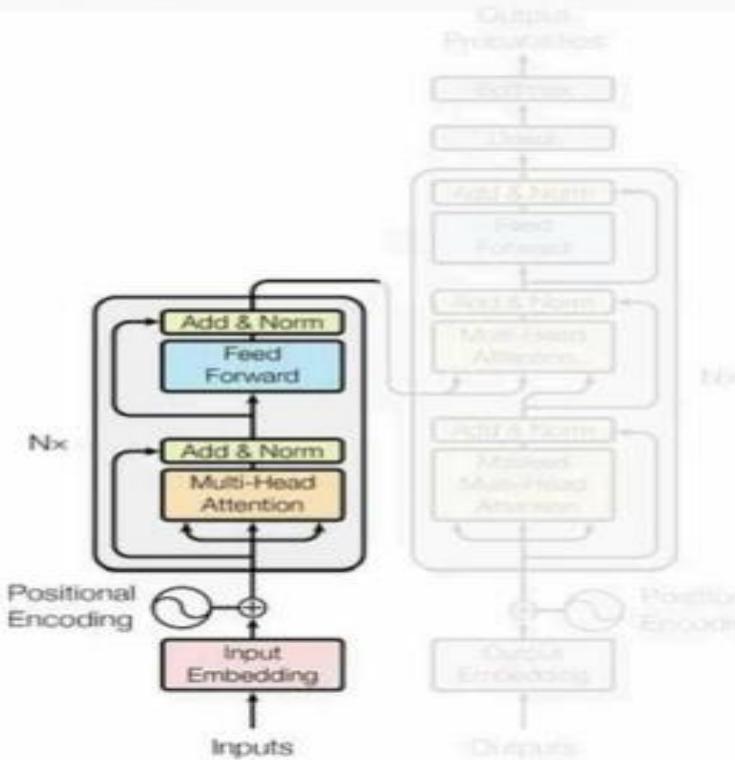
### 1. Encoder (N layers)

- Input embeddings + positional encoding
- Multi-Head Self-Attention
- Feedforward Network
- Layer Normalization & Residual Connections

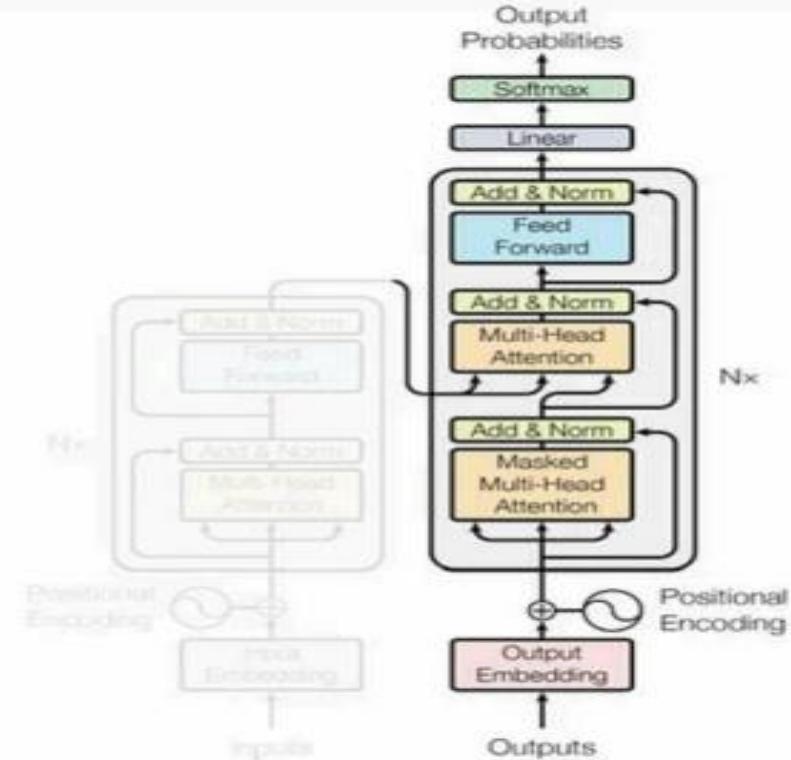
### 2. Decoder (N layers)

- Multi-Head Self-Attention
- Encoder-Decoder Attention
- Feedforward Network
- Layer Normalization & Residual Connections

## What we have covered so far



## What we have to cover

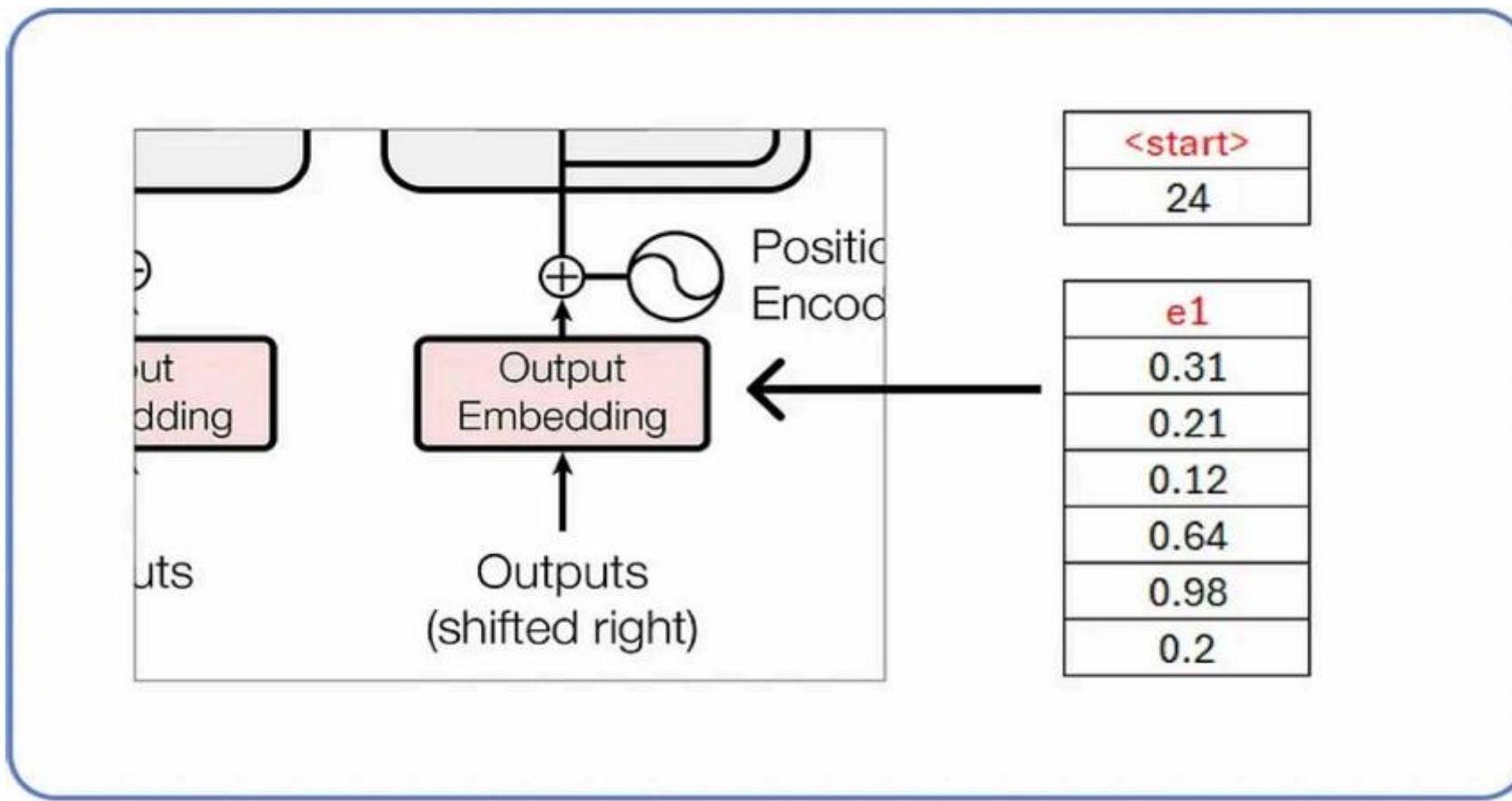


Upcoming steps illustration

Encoder Input → When you play game of thrones  
Decoder Input → <start> you win or you die <end>

input comparison of encoder and decoder

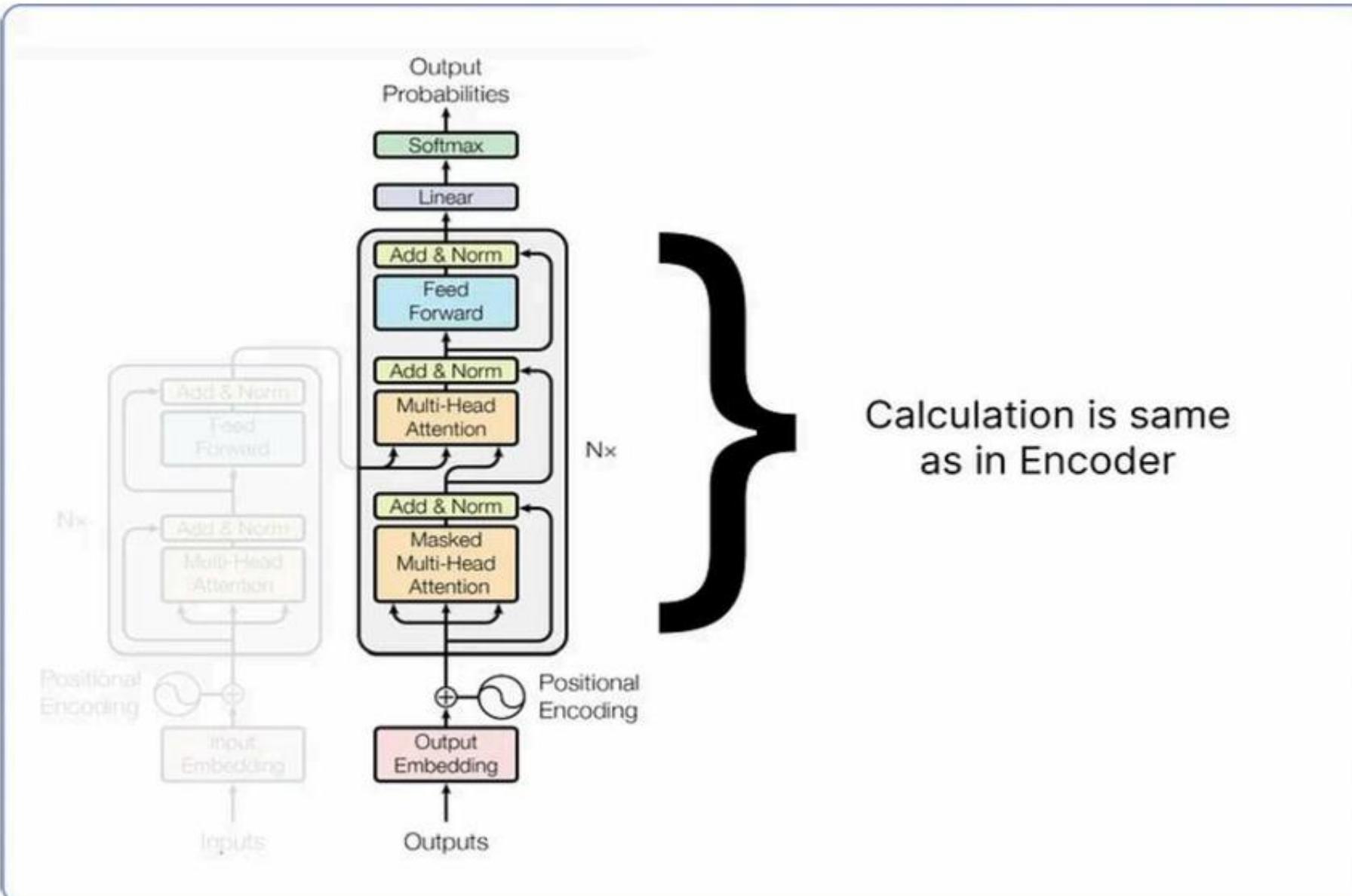
Where <start> and <end> are two new tokens being introduced. Moreover, the decoder takes one token as an input at a time. It means that <start> will be served as an input, and you must be the predicted text for it.



Decoder input <start> word

As we already know, these embeddings are filled with random values, which will later be updated during the training process.

## Step 11 — Decoder Part



## Step 12 — Understanding Mask Multi Head Attention

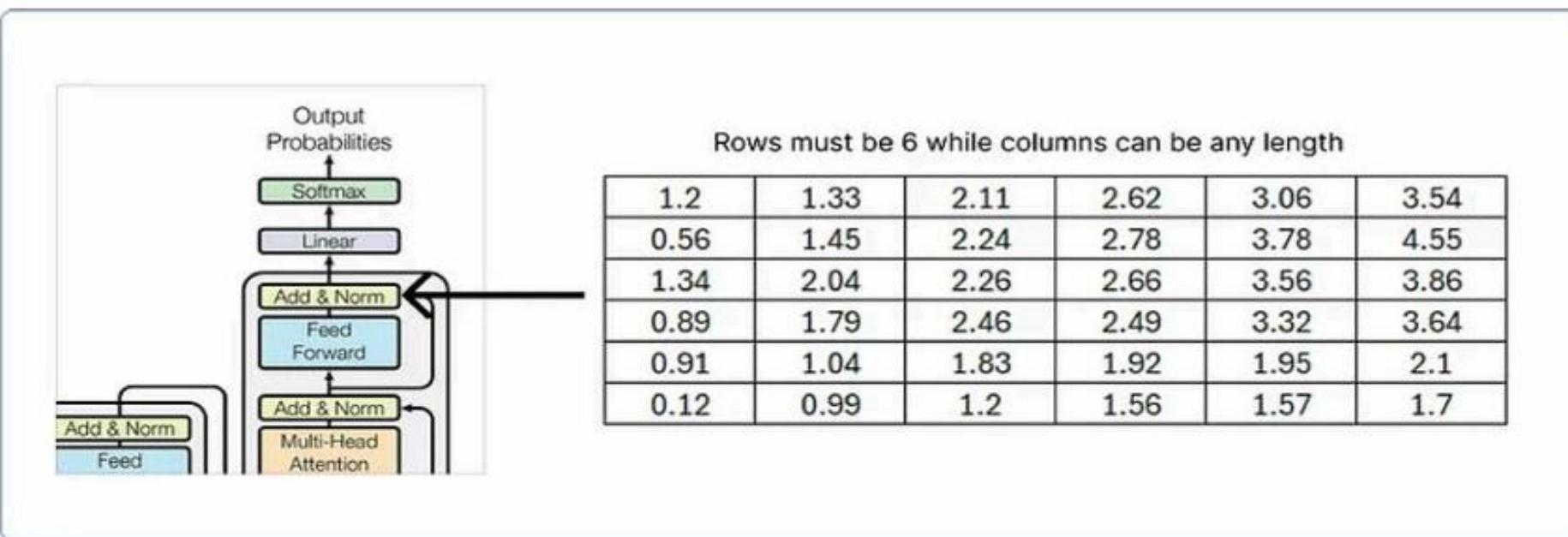
In a Transformer, the masked multi-head attention is like a spotlight that a model uses to focus on different parts of a sentence. It's special because it doesn't let the model cheat by looking at words that come later in the sentence. This helps the model understand and generate sentences step by step, which is important in tasks like talking or translating words into another language.

Now, let's understand the masked multi-head attention components having two heads:

1. **Linear Projections (Query, Key, Value):** Assume the linear projections for each head: **Head 1:**  $Wq1, Wk1, Wv1$  and **Head 2:**  $Wq2, Wk2, Wv2$
2. **Calculate Attention Scores:** For each head, calculate attention scores using the dot product of Query and Key, and apply the mask to prevent attending to future positions.
3. **Apply Softmax:** Apply the softmax function to obtain attention weights.
4. **Weighted Summation (Value):** Multiply the attention weights by the Value to get the weighted sum for each head.
5. **Concatenate and Linear Transformation:** Concatenate the outputs from both heads and apply a linear transformation.

## Step 13 — Calculating the Predicted Word

The output matrix of the last add and norm block of the decoder must contain the same number of rows as the input matrix, while the number of columns can be any. Here, we work with 6.

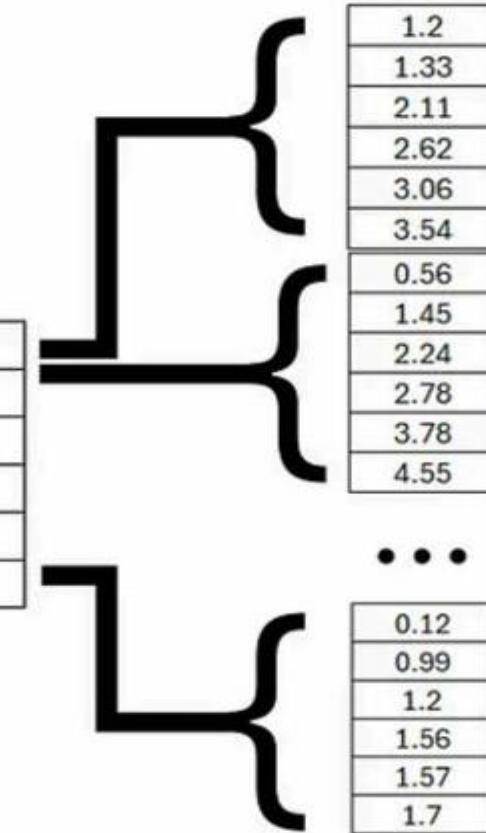


Add and Norm output of decoder

The last **add and norm** block resultant matrix of the decoder must be flattened in order to match it with a linear layer to find the predicted probability of each unique word in our dataset (corpus).

## Flatten the matrix

1.2	1.33	2.11	2.62	3.06	3.54
0.56	1.45	2.24	2.78	3.78	4.55
1.34	2.04	2.26	2.66	3.56	3.86
0.89	1.79	2.46	2.49	3.32	3.64
0.91	1.04	1.83	1.92	1.95	2.1
0.12	0.99	1.2	1.56	1.57	1.7



flattened the last add and norm block matrix

This flattened layer will be passed through a linear layer to compute the **logits** (scores) of each unique word in our dataset.

# Transformer Decoder Block Overview

The **Decoder Block** is a crucial part of the Transformer model, used primarily in **sequence-to-sequence** tasks like machine translation.

Each **Decoder Block** consists of the following **three main sub-layers**:

1. Masked Multi-Head Self-Attention
2. Multi-Head Cross-Attention
3. Feed-Forward Network (FFN)

Each of these layers is followed by **Layer Normalization** and **Residual Connections**.

## ◆ Input to the Decoder

The decoder takes two inputs:

1. **Target Sequence ( $y$ )**: The sequence that we want to generate (e.g., translated sentence).
2. **Encoder Output ( $z$ )**: The output from the Transformer encoder (contains encoded information from the source sequence).

The decoder processes the target sequence in an autoregressive manner, meaning it only looks at past tokens while generating the next one.

## ● 1st Block: Masked Multi-Head Self-Attention

### ❖ Purpose

- Allows the decoder to attend to previous words in the target sequence.
- Ensures that the decoder **cannot peek ahead** (to future tokens) during training.

### ⚙ How it Works?

1. The input **target sequence** ( $y$ ) is **embedded** and **positionally encoded** (same as in the encoder).
2. This sequence is passed into the **Self-Attention** mechanism.
3. A **look-ahead mask** (triangular matrix) is applied to prevent the decoder from seeing future words.

### 💡 Masking Explained

The **look-ahead mask** ensures that for any token at position  $t$ , it can **only attend** to previous tokens ( $t-1$ ,  $t-2$ , etc.), but **not** future ones.

## ● 2nd Block: Multi-Head Cross-Attention

### 📌 Purpose

- Helps the decoder **attend to relevant parts of the encoder output** (i.e., the source sentence).
- This is where the decoder **learns dependencies between the source and target sequences**.

### ⚙️ How it Works?

1. The **queries** ( $Q$ ) come from the **decoder's self-attention output**.
2. The **keys** ( $k$ ) and **values** ( $v$ ) come from the **encoder output** ( $z$ ).
3. Multi-head attention is applied:
  - The decoder **learns which parts of the source sequence are important** for generating the target sequence.

### 🚀 Output of this Block

- The output now contains information **from both the target and source sequences**.

## ● 3rd Block: Feed-Forward Network (FFN)

### 📌 Purpose

- Applies **non-linearity** and **transforms attention output** before passing it to the next decoder layer.

### ⚙️ How it Works?

1. The cross-attention output is passed through **two fully connected layers**:
  - **First layer (Expansion Layer)**: Expands the dimension ( $d_{model} \rightarrow d_{ff}$ ).
  - **ReLU Activation** for non-linearity.
  - **Second layer (Compression Layer)**: Compresses back ( $d_{ff} \rightarrow d_{model}$ ).
2. Helps the model learn **complex representations**.

### 🚀 Output of this Block

- The transformed features are **sent to the next decoder layer** (or final output layer in the last decoder block).

## ◆ Additional Components

### ◆ Residual Connections

- Each sub-layer (Self-Attention, Cross-Attention, FFN) has a skip connection:
- Helps in better gradient flow and prevents vanishing gradients.

### ◆ Layer Normalization

- After each sub-layer, Layer Normalization (LayerNorm) is applied:
- Stabilizes training.



# Final Output of the Decoder Block

- The decoder block is stacked  $N$  times (e.g.,  $N = 6$  in the original Transformer).
- The final decoder output is sent to a linear layer + softmax to generate probabilities over vocabulary tokens.



# Summary of Each Block

Block	Purpose	Input	Output
Masked Multi-Head Self-Attention	Allows decoder to attend only to previous tokens (prevents cheating)	Target sequence ( $Y$ )	Processed representation of $Y$
Multi-Head Cross-Attention	Allows decoder to attend to relevant source tokens	Encoder output ( $Z$ ) + Self-attention output	Context-aware representation
Feed-Forward Network (FFN)	Non-linearity and transformation	Cross-attention output	Final transformed representation