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

Embedded Learning

TRANSFORMERS

04-02-2025

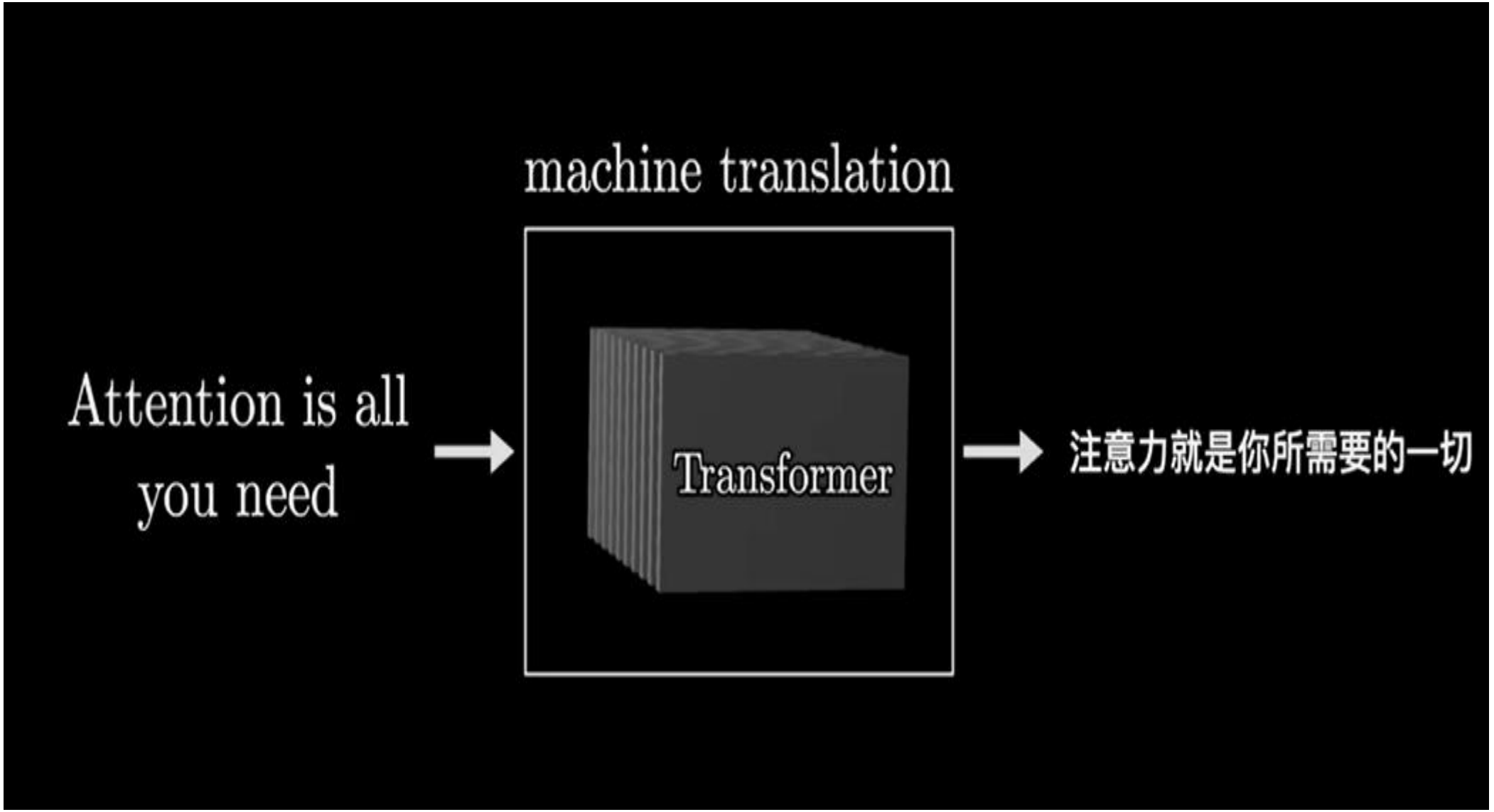
By

Asha.M

Assistant professor

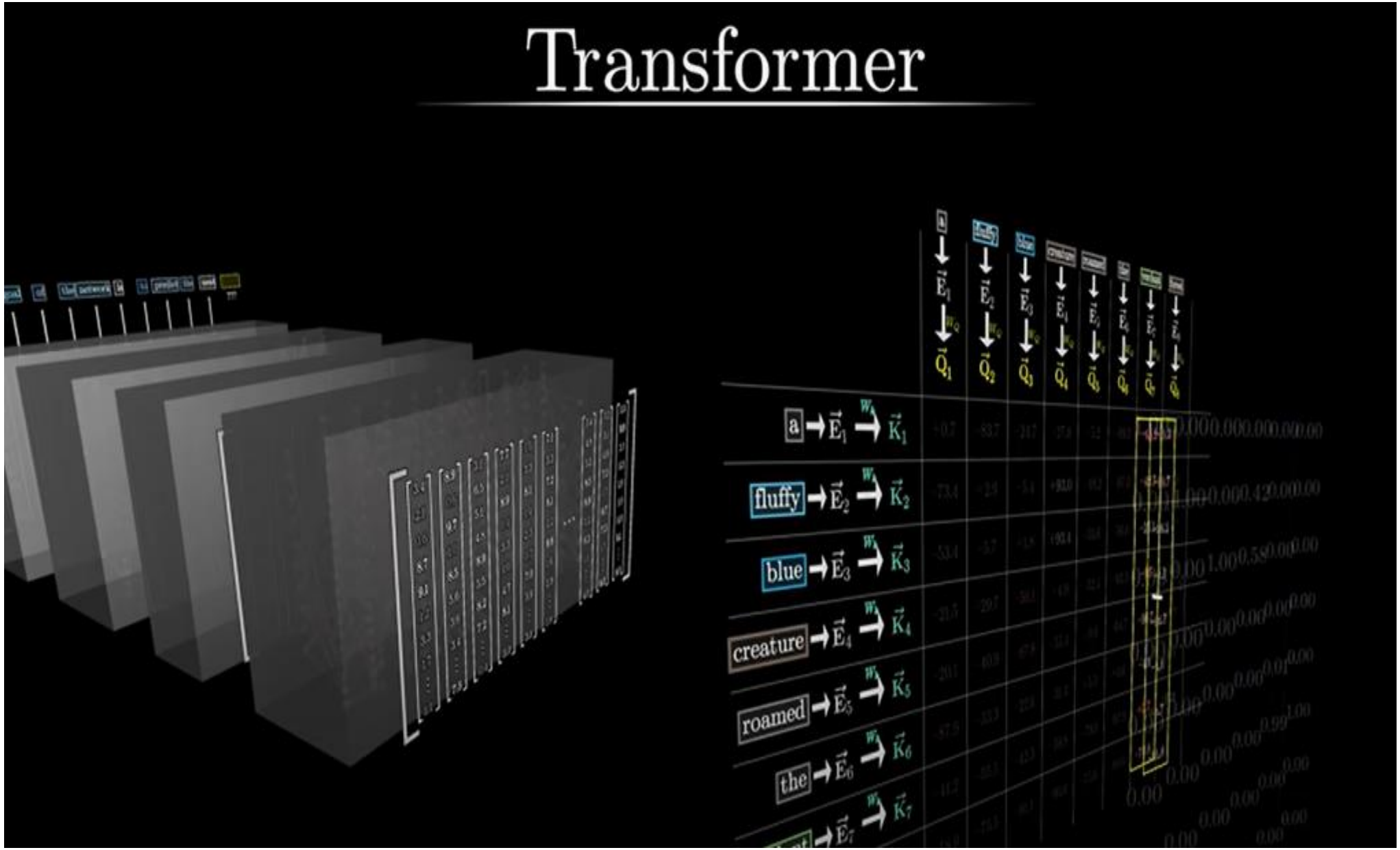
CSE(AI&ML)

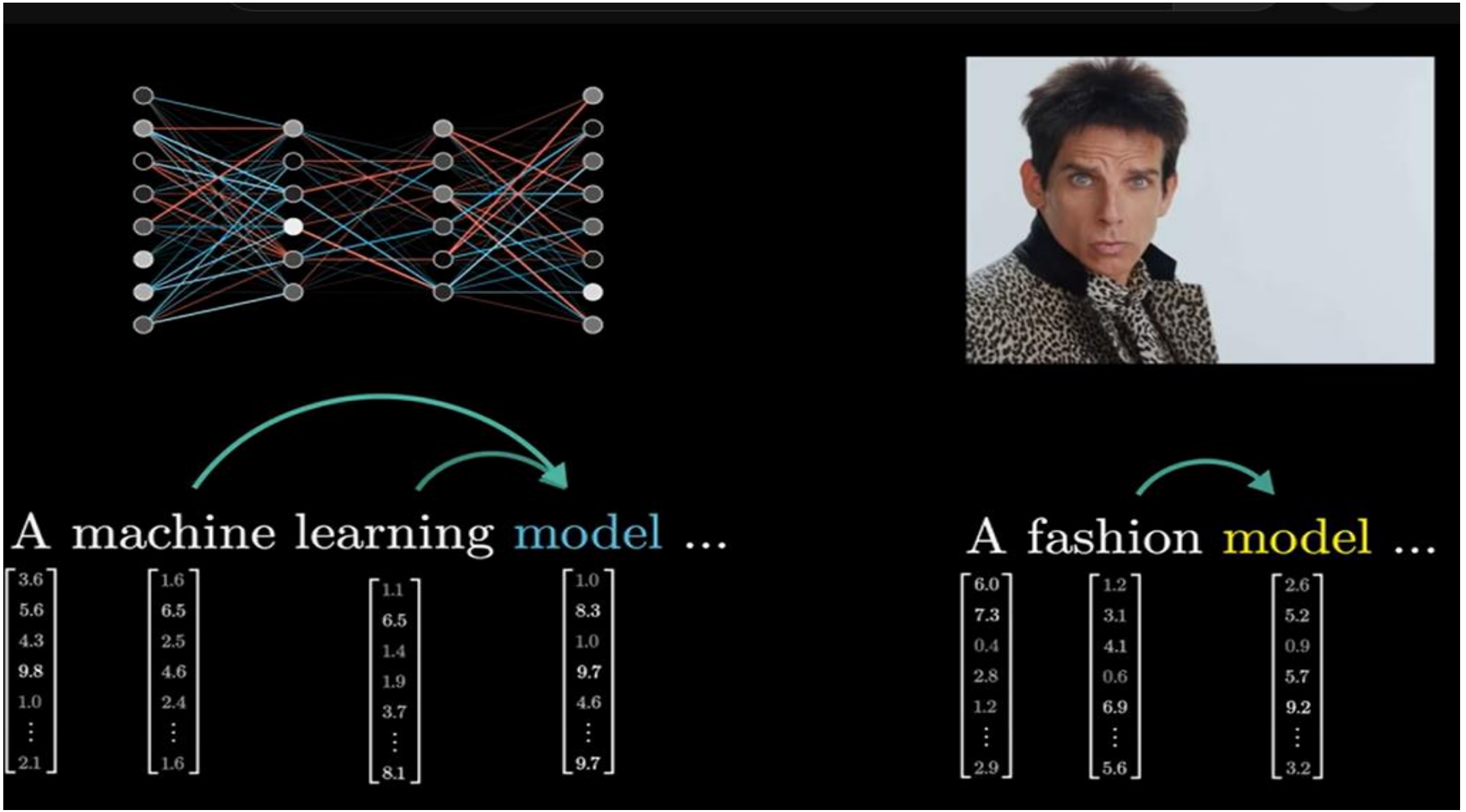
KMIT

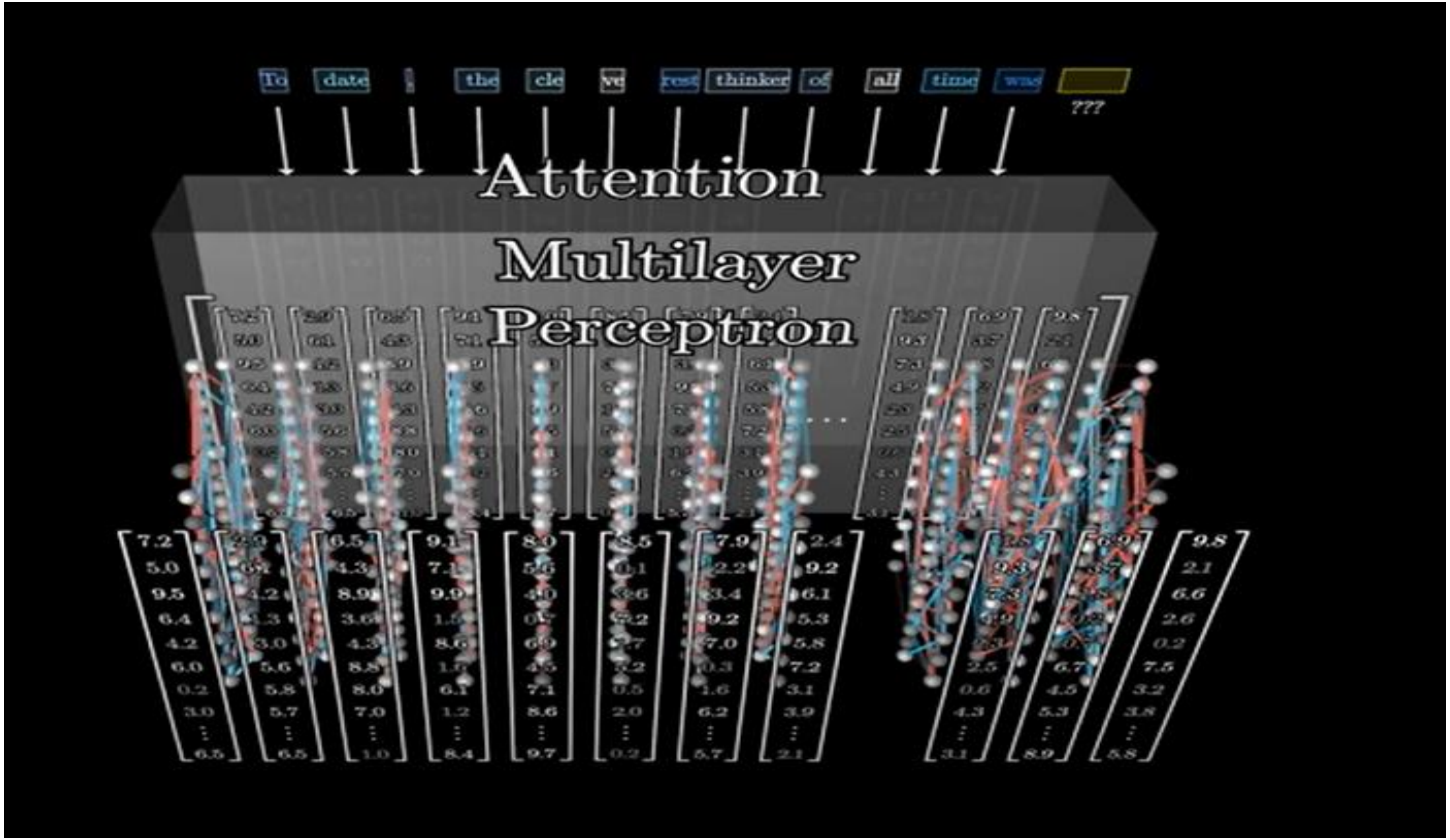

machine translation
Attention is all you need
Transformer
注意力就是你所需要的一切

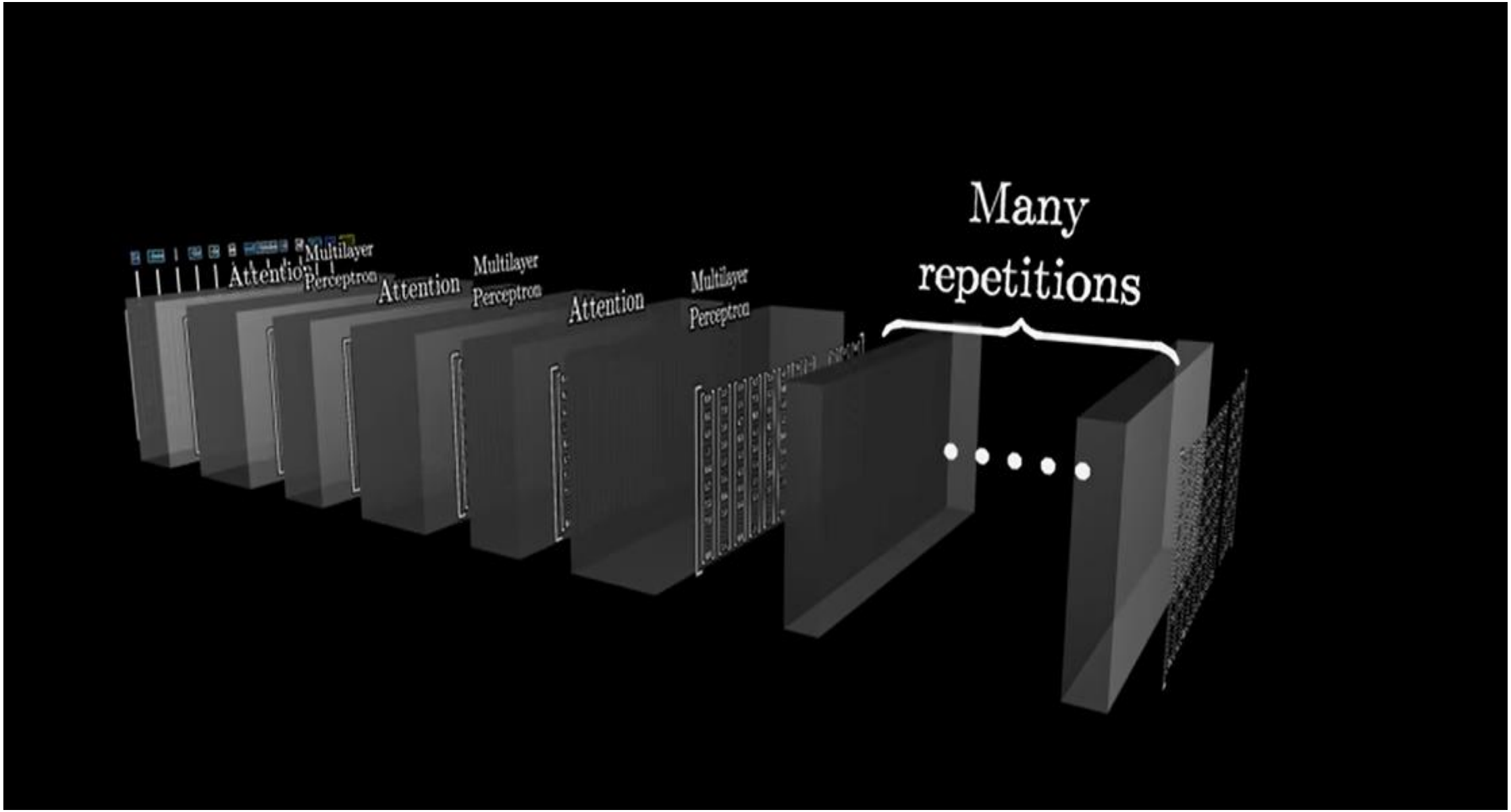
What is a Transformer?

A **Transformer** is a deep learning model introduced in **"Attention Is All You Need" (Vaswani et al., 2017)**. It **replaces recurrent layers (RNNs, LSTMs)** with **self-attention mechanisms**, allowing for **parallel processing** and **long-range dependencies**.


Transformer
a
fluffy
blue
creature
roamed
the


A machine learning model ...
A fashion model ...


To
date
the
thinker
of
all
time
???
Attention
Multilayer
Perceptron


Attention
Multilayer Perceptron
Attention
Multilayer Perceptron
Attention
Multilayer Perceptron
Many repetitions

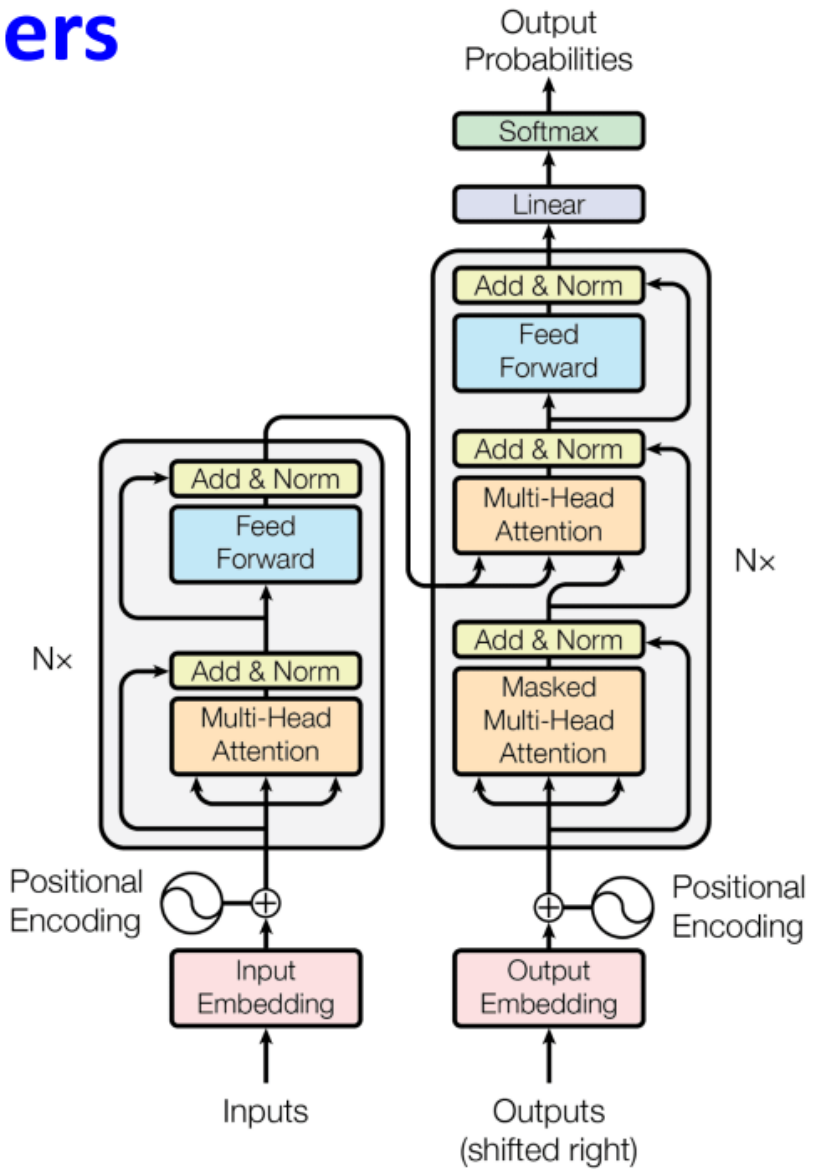
Attention Mechanism

Assign attention weight to each word, to know how much "attention" the model should pay to each word (i.e., for each word, the network learns a "context")

Transformers

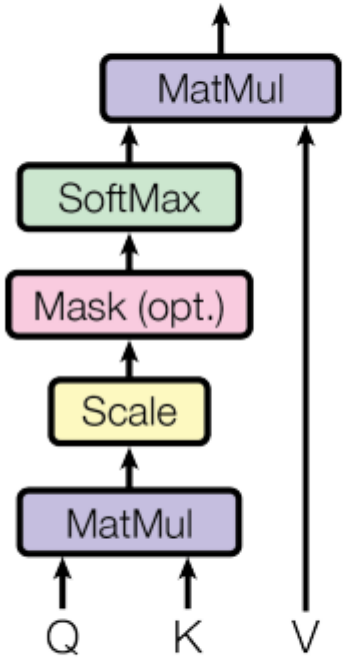
- Tokenizaton
- Input Embeddings
- Position Encodings
- Residuals
- Query
- Key
- Value
- Add & Norm
- Encoder
- Decoder

- Attention
- Self Attention
- Multi Head Attention
- Masked Attention
- Encoder Decoder Attention
- Output Probabilities / Logits
- Softmax
- Encoder-Decoder models
- Decoder only models

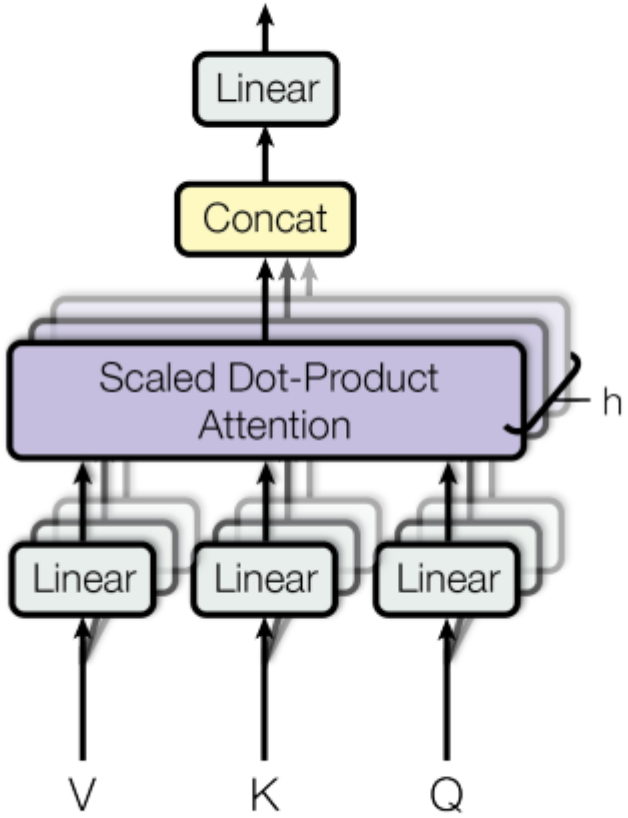


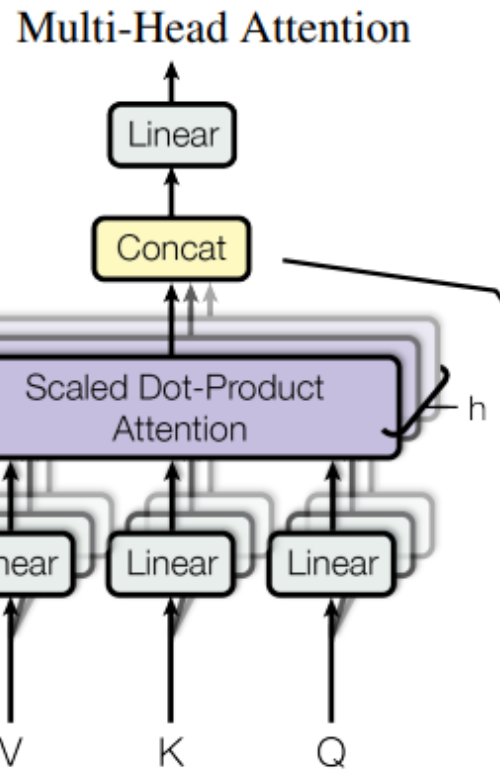
Kmit

Scaled Dot-Product Attention



Multi-Head Attention





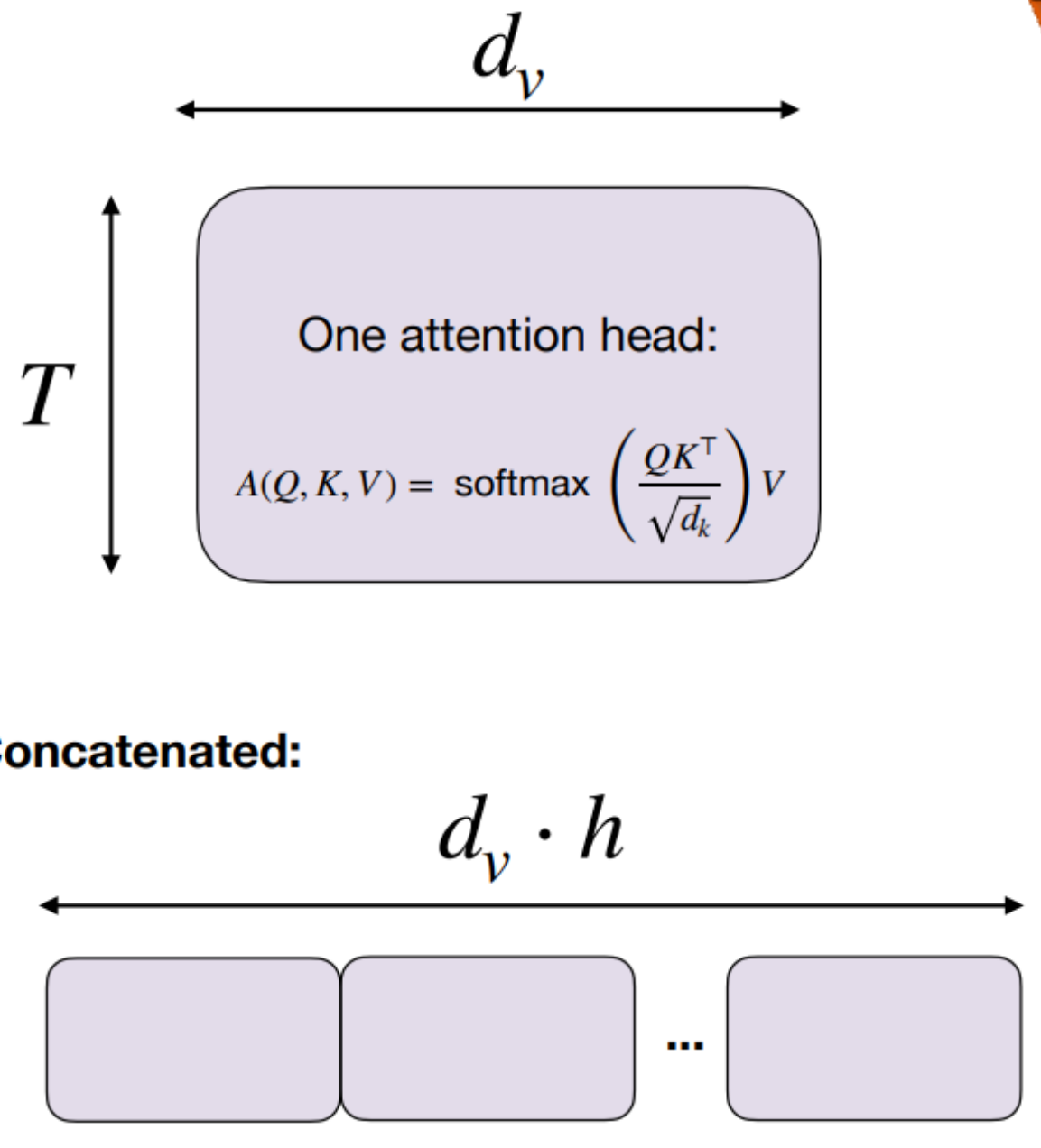
Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L. and Polosukhin, I., 2017. Attention Is All You Need.

Input sequence dim.
in original transformer:

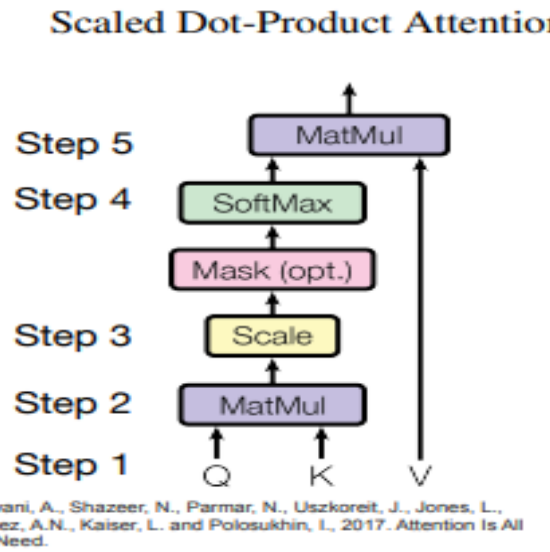
$$T \times d_e = T \times 512$$

and

$$d_v = 512/h = 64$$

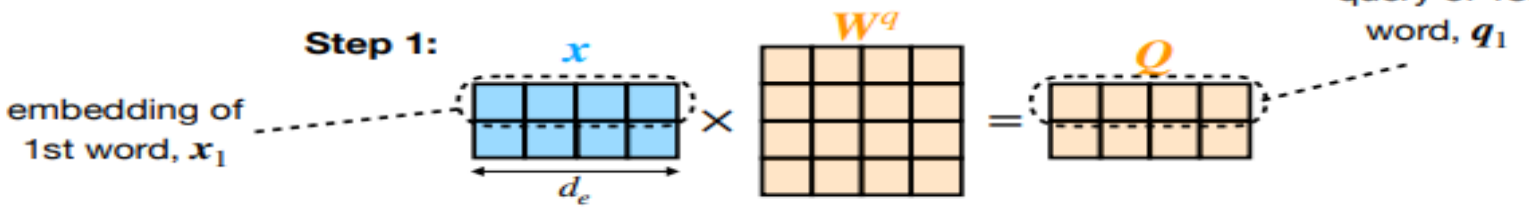


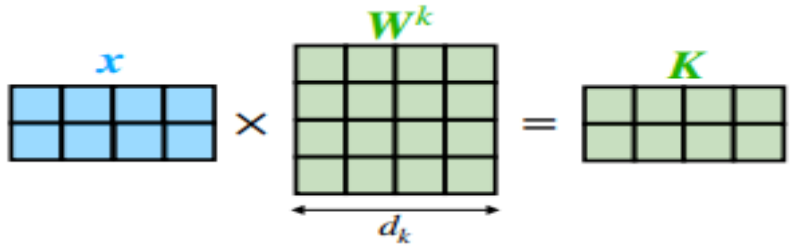
Scaled Dot-Product Attention Recap

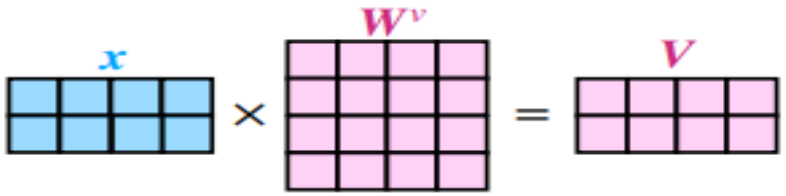


Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L. and Polosukhin, I., 2017. Attention Is All You Need.

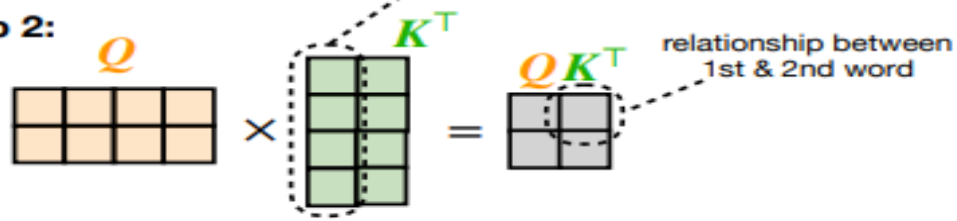
Step 1:



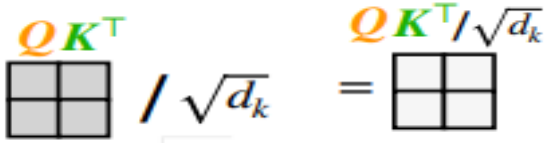




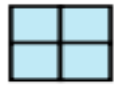
Step 2:



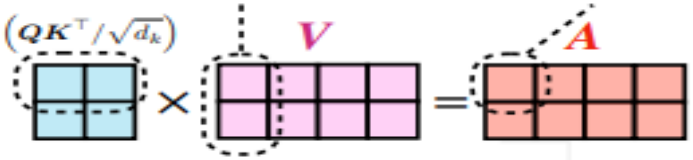
Step 3:

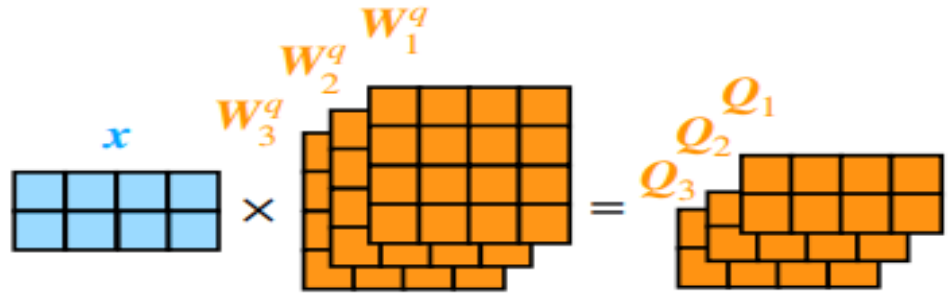


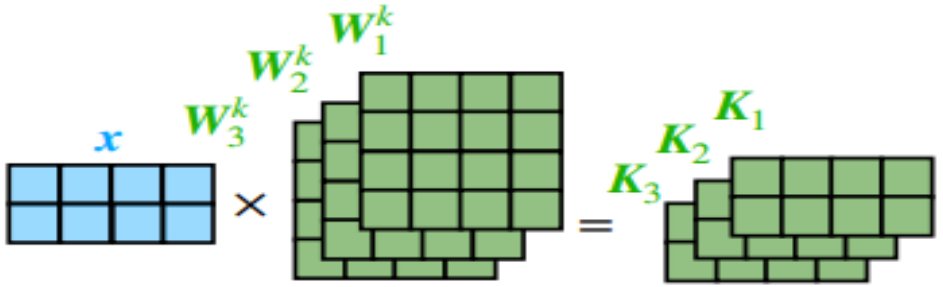
Step 4: $\text{softmax}\left(QK^\top/\sqrt{d_k}\right) =$

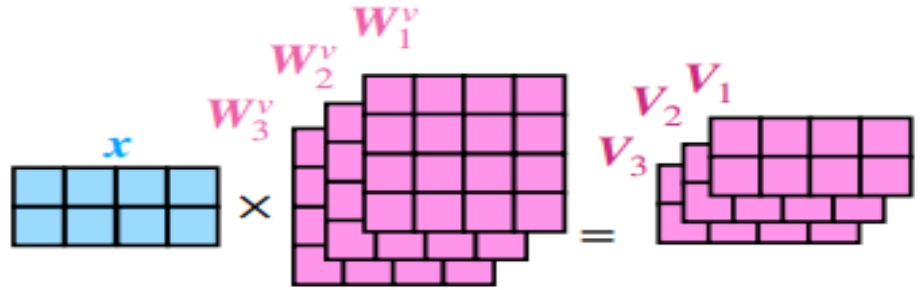


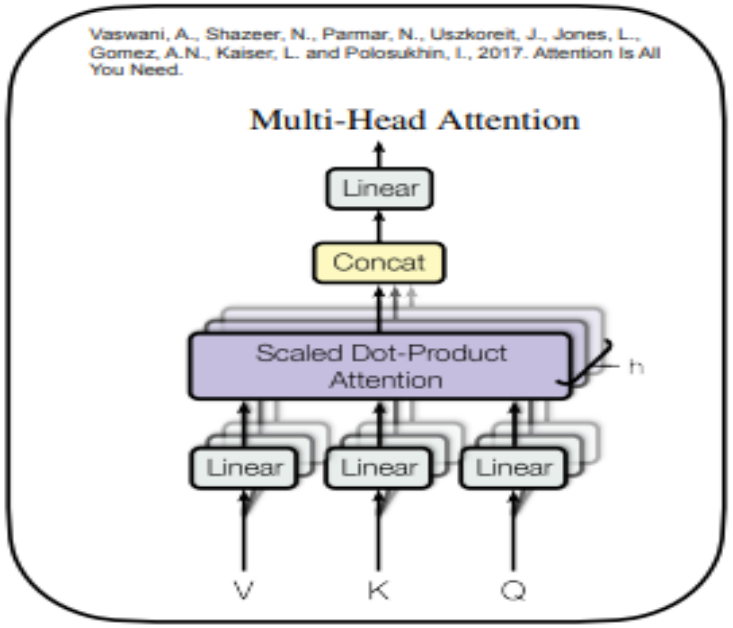
Step 5:

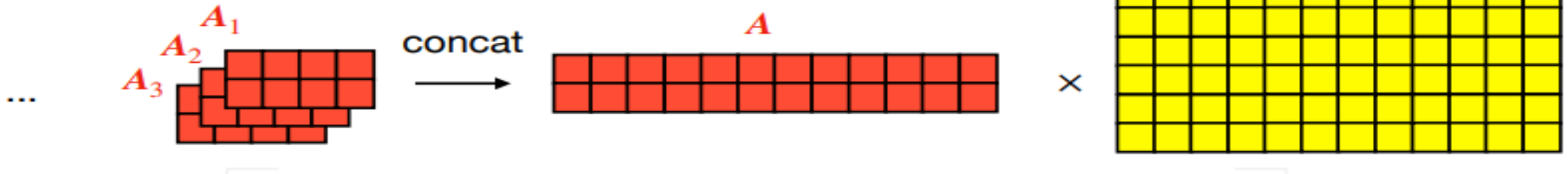



x
W_1^q
W_2^q
W_3^q
Q_1
Q_2
Q_3

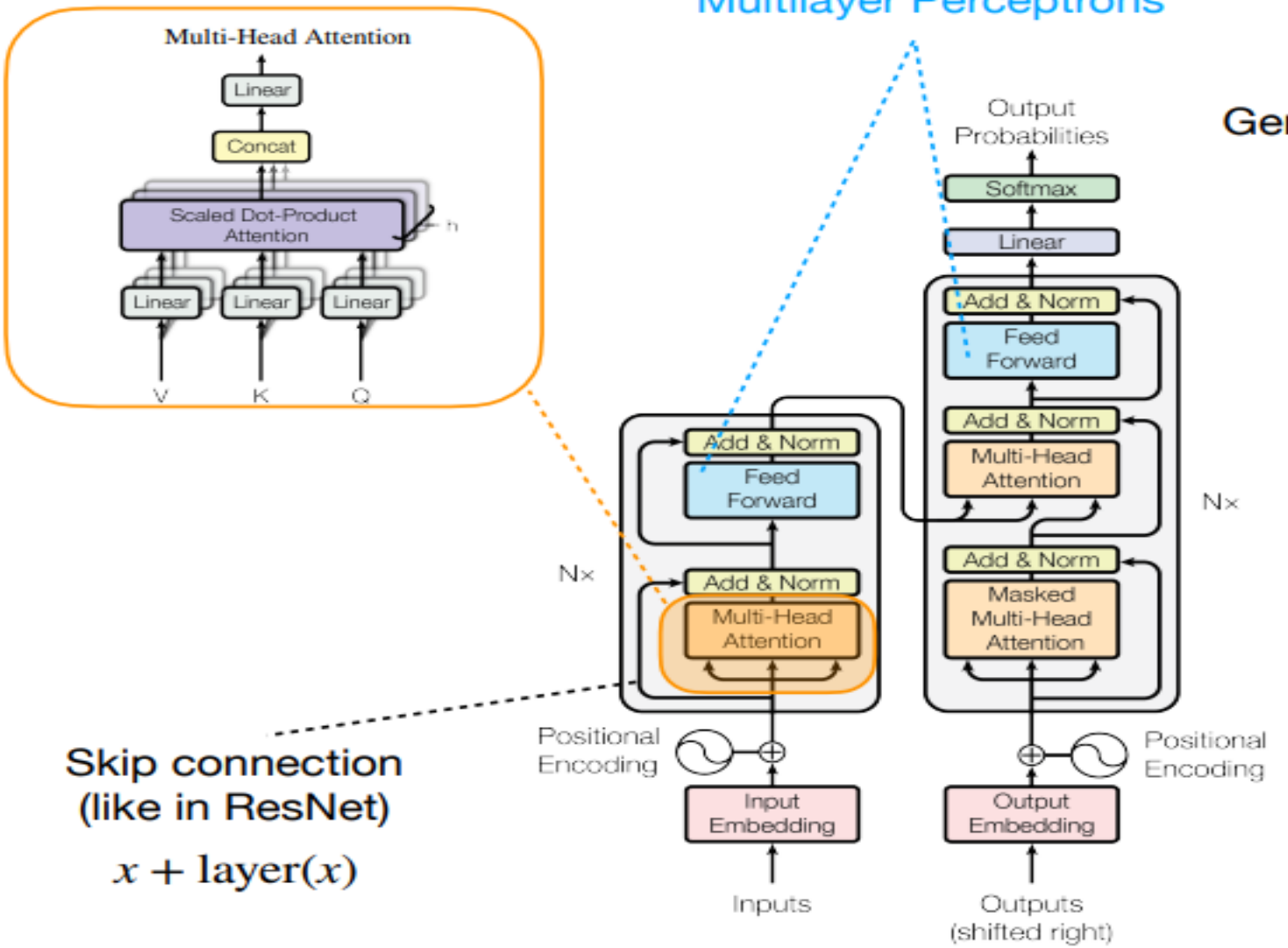

x
W_1^k
W_2^k
W_3^k
K_1
K_2
K_3


x
W_1^v
W_2^v
W_3^v
V_1
V_2
V_3


Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L. and Polosukhin, I., 2017. Attention Is All You Need.
Multi-Head Attention
Linear
Concat
Scaled Dot-Product Attention
h
Linear
Linear
Linear
V
K
Q


...
A_1
A_2
A_3
concat
A
W_o

Multilayer Perceptrons



Generates output words one at a time

Skip connection (like in ResNet)

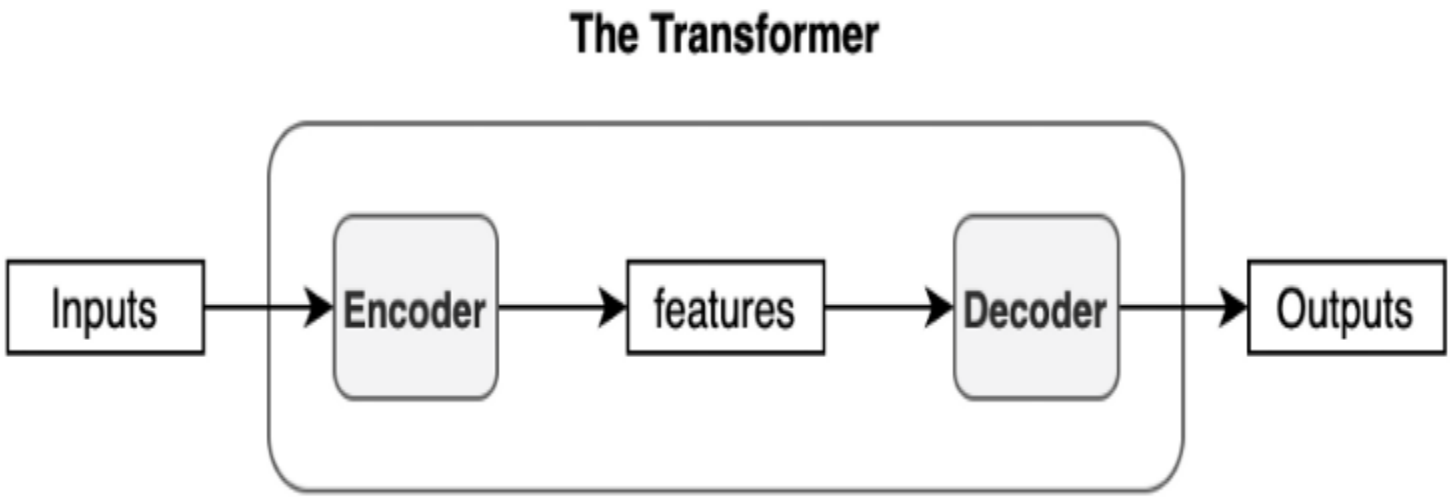
$$x + \text{layer}(x)$$

Figure 1: The Transformer - model architecture.

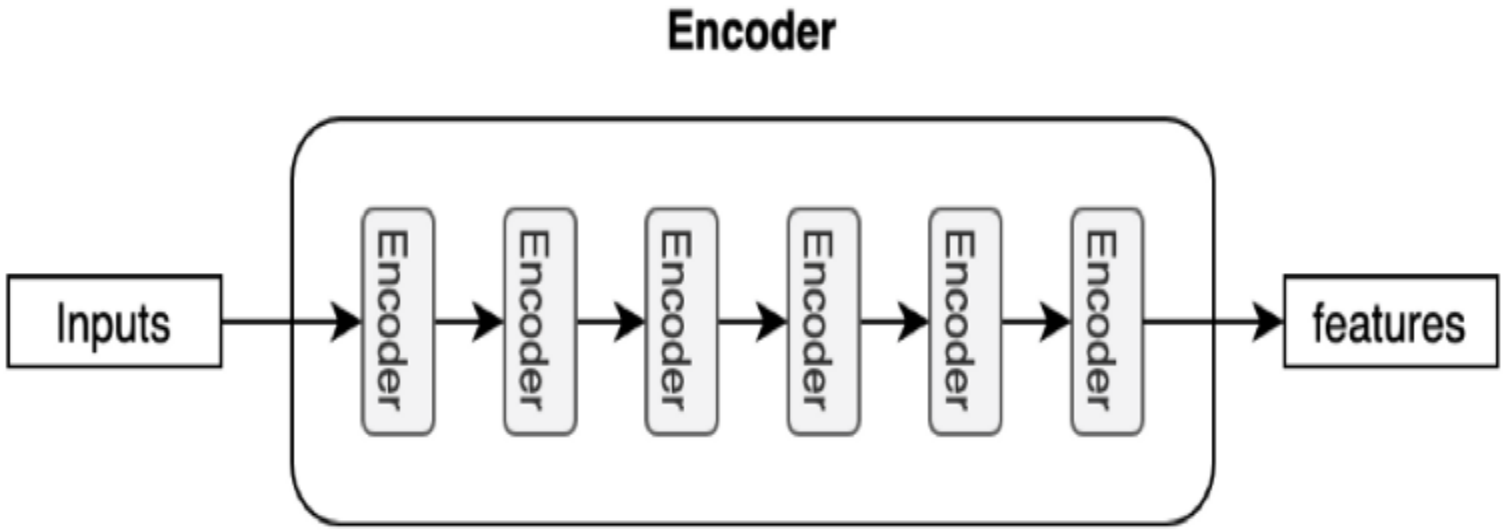
Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L. and Polosukhin, I., 2017. Attention Is All You Need.

The original transformer published in the paper is a neural machine translation model. For example, we can train it to translate English into French sentences.

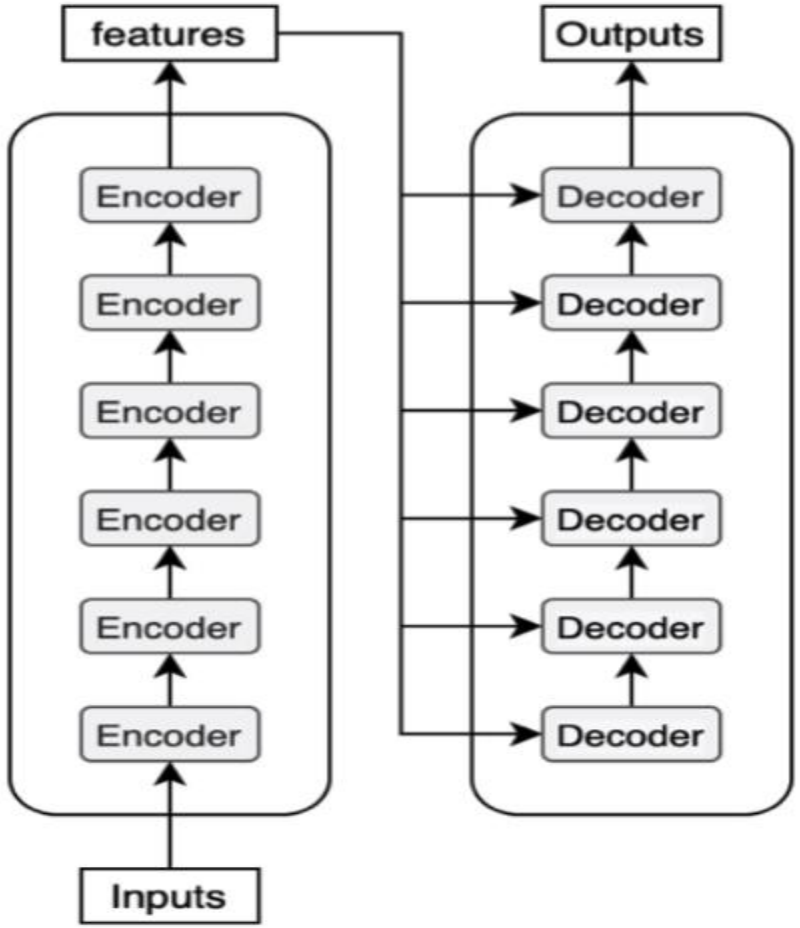
The transformer uses an encoder-decoder architecture. The encoder extracts features from an input sentence, and the decoder uses the features to produce an output sentence (translation).



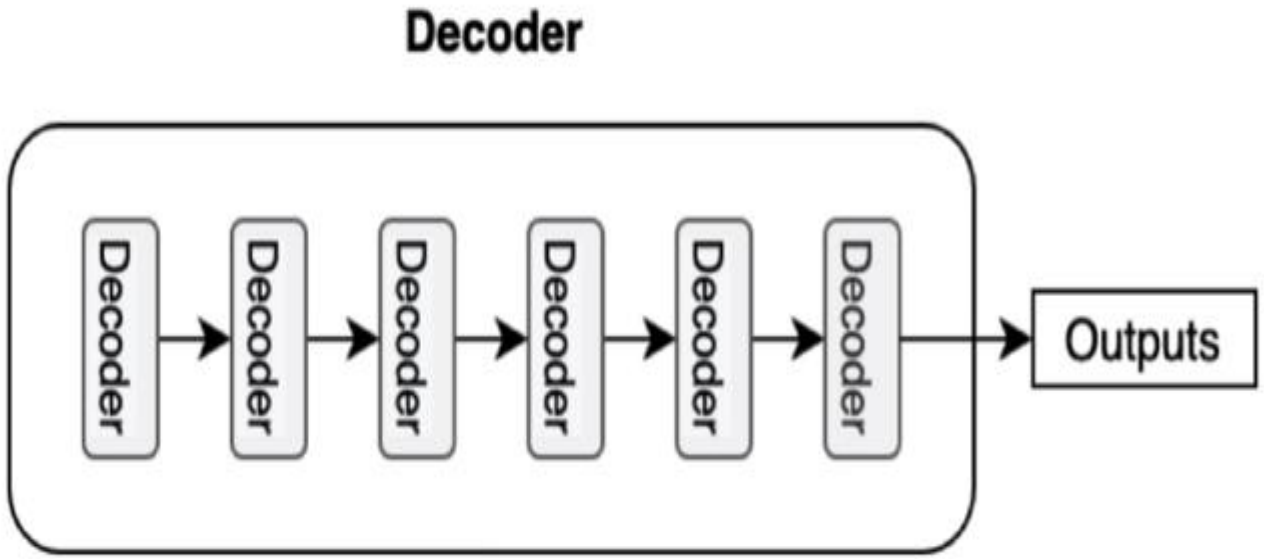
The encoder in the transformer consists of multiple encoder blocks. An input sentence goes through the encoder blocks, and the output of the last encoder block becomes the input features to the decoder.



Each decoder block receives the features from the encoder. If we draw the encoder and the decoder vertically, the whole picture looks like the diagram from the paper.

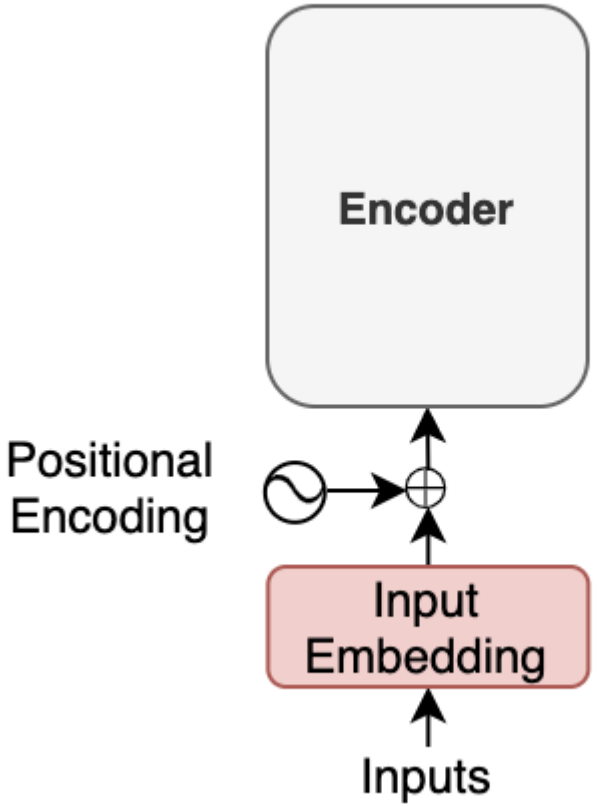


The decoder also consists of multiple decoder blocks.



Input Embedding and Positional Encoding

Like any neural translation model, we often tokenize an input sentence into distinct elements (tokens). A tokenized sentence is a fixed-length sequence.



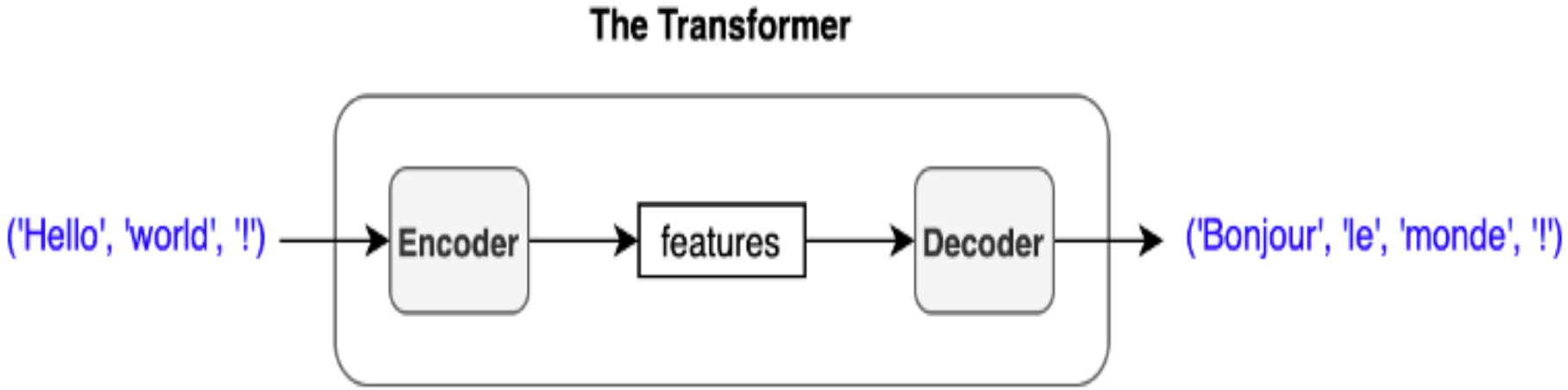
- with the transformer, we inject **positional encoding** into each embedding so that the model can know word positions without recurrence.

- The input to the transformer is not the characters of the input text but a sequence of embedding vectors.

- Each vector represents the semantics and position of a token. The encoder performs linear algebra operations on those vectors to extract the contexts for each token from the entire sentence and enrich the embedding vectors with helpful information for the target task through the multiple encoder blocks.

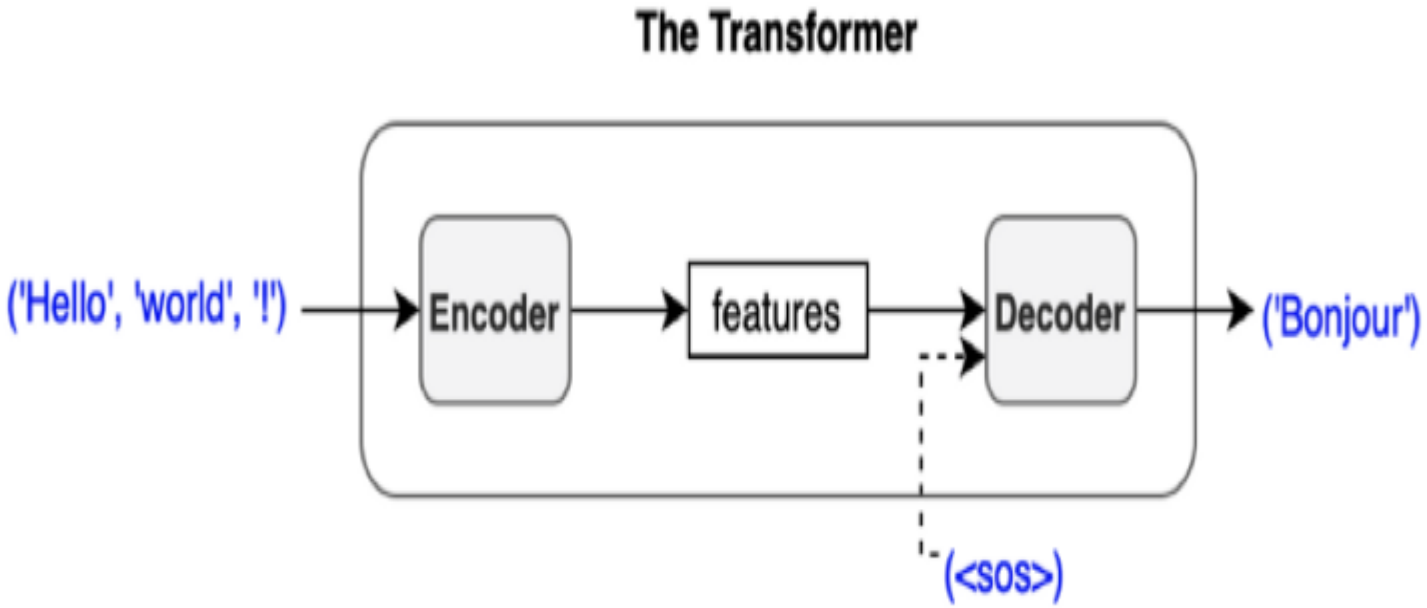
Softmax and Output Probabilities

The decoder uses input features from the encoder to generate an output sentence. The input features are nothing but enriched embedding vectors.



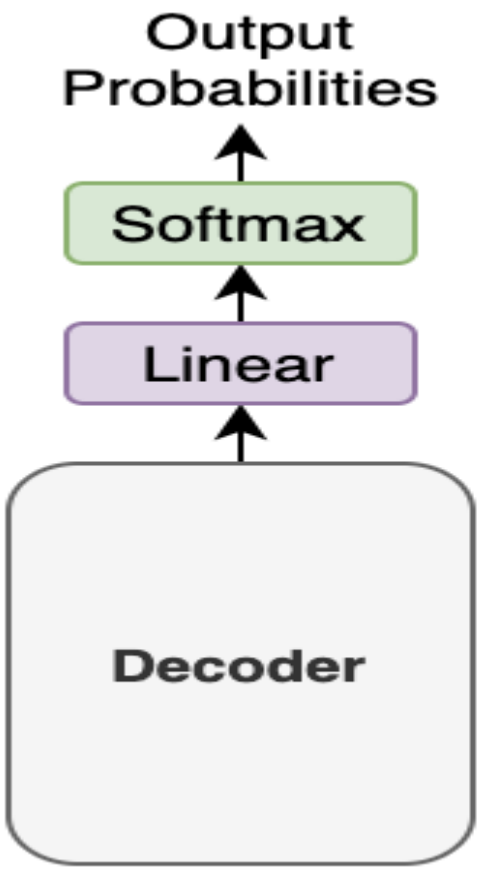
The decoder outputs one token at a time. An output token becomes the subsequent input to the decoder. In other words, a previous output from the decoder becomes the last part of the next input to the decoder. This kind of processing is called “**auto-regressive**”—a typical pattern for generating sequential outputs and not specific to the transformer. It allows a model to generate an output sentence of different lengths than the input.

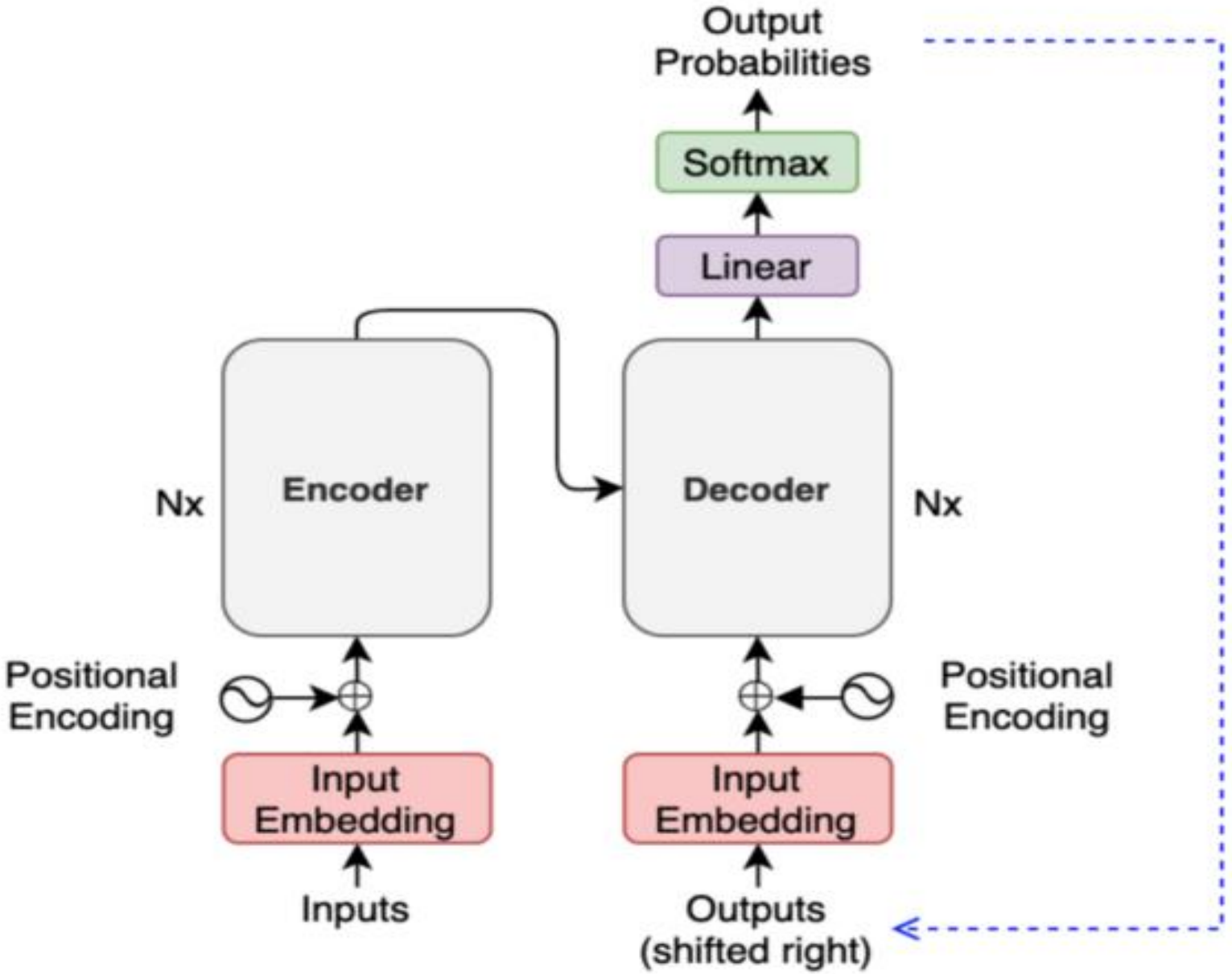
However, we have no previous output at the beginning of a translation. So, we pass the start-of-sentence marker <SOS> to the decoder to initiate the translation.



The decoder uses multiple decoder blocks to enrich <SOS> with the contextual information from the input features. In other words, the decoder transforms the embedding vector <SOS> into a vector containing information helpful to generating the first translated word (token).

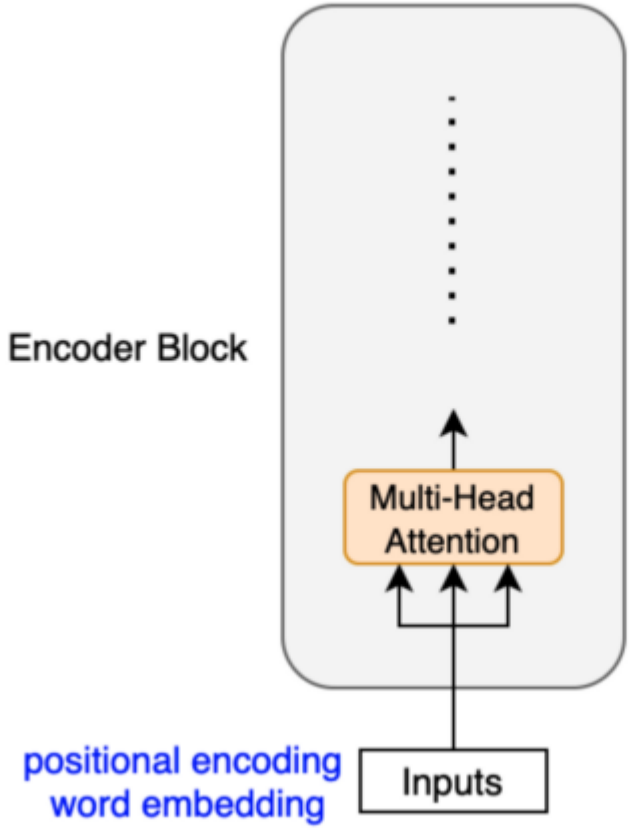
Then, the output vector from the decoder goes through a linear transformation that changes the dimension of the vector from the embedding vector size (512) into the size of vocabulary (say, 10,000). The softmax layer further converts the vector into 10,000 probabilities.


Output
Probabilities
Softmax
Linear
Decoder

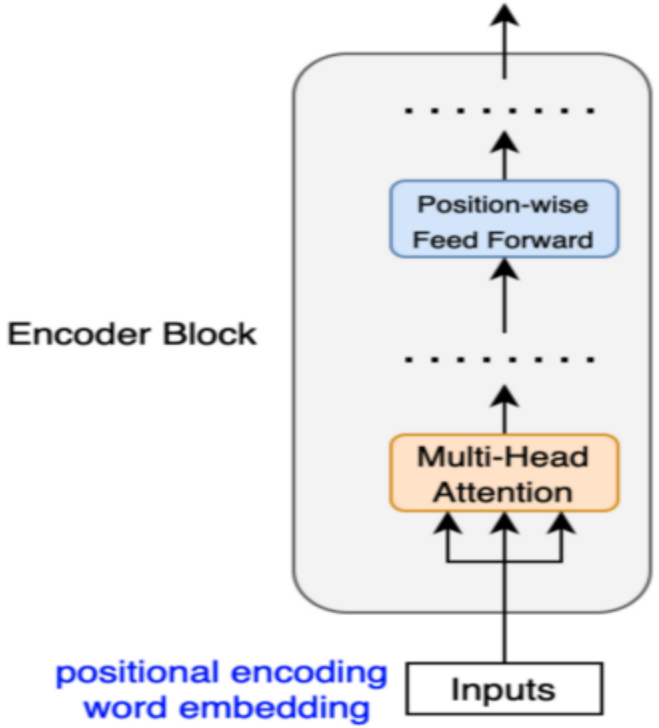

Output Probabilities
Softmax
Linear
Nx
Encoder
Decoder
Nx
Positional Encoding
Positional Encoding
Input Embedding
Input Embedding
Inputs
Outputs (shifted right)

Encoder Block Internals

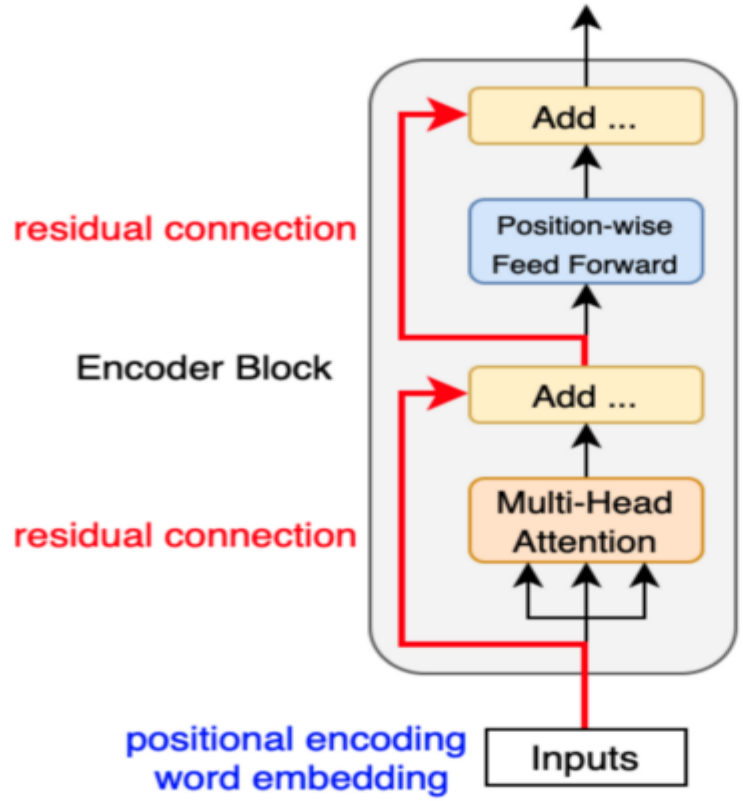
The encoder block uses the **self-attention mechanism** to enrich each token (embedding vector) with contextual information from the whole sentence.



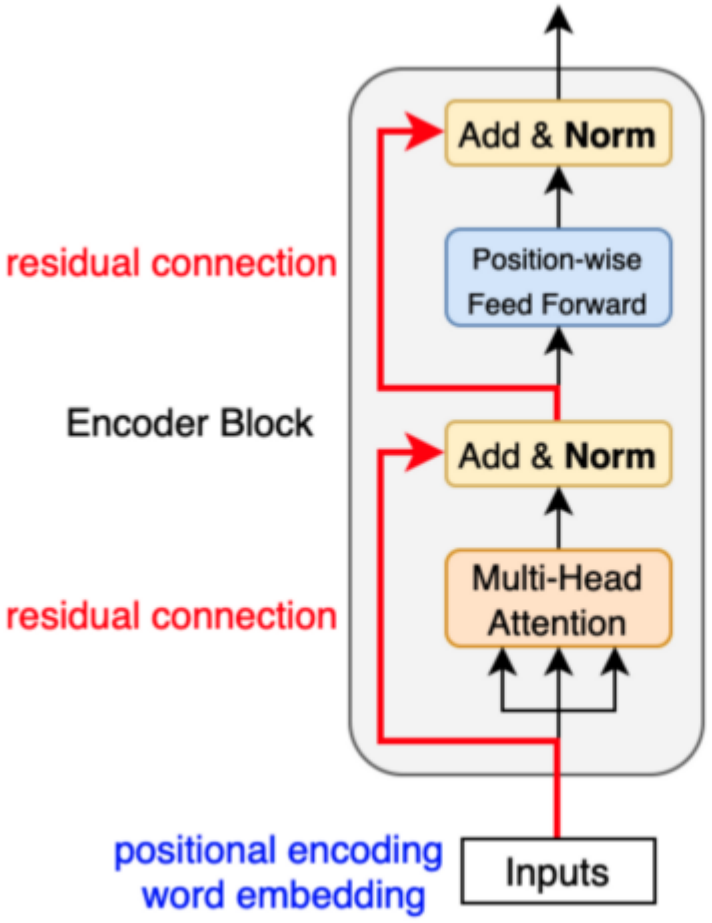
The **position-wise feed-forward network** (FFN) has a linear layer, ReLU, and another linear layer, which processes each embedding vector independently with identical weights. So, each embedding vector (with contextual information from the multi-head attention) goes through the position-wise feed-forward layer for further transformation.



Residual connections carry over the previous embeddings to the subsequent layers. As such, the encoder blocks enrich the embedding vectors with additional information obtained from the multi-head self-attention calculations and position-wise feed-forward networks.



After each residual connection, there is a **layer normalization**:

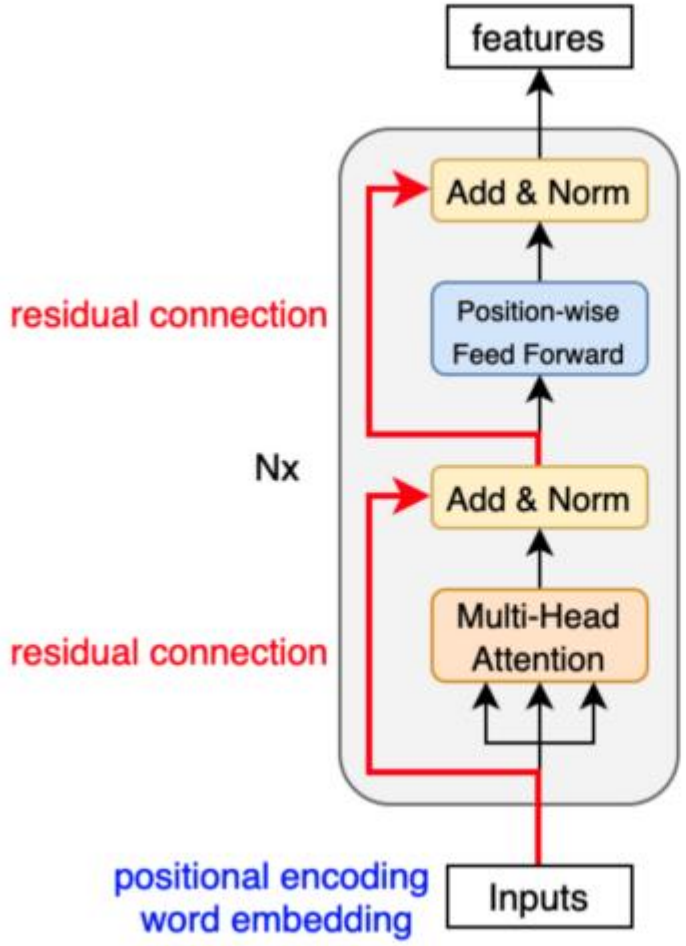


Like batch normalization, layer normalization aims to reduce the effect of covariant shift.

In other words, it prevents the mean and standard deviation of embedding vector elements from moving around, which makes training unstable and slow (i.e., we can’t make the learning rate big enough).

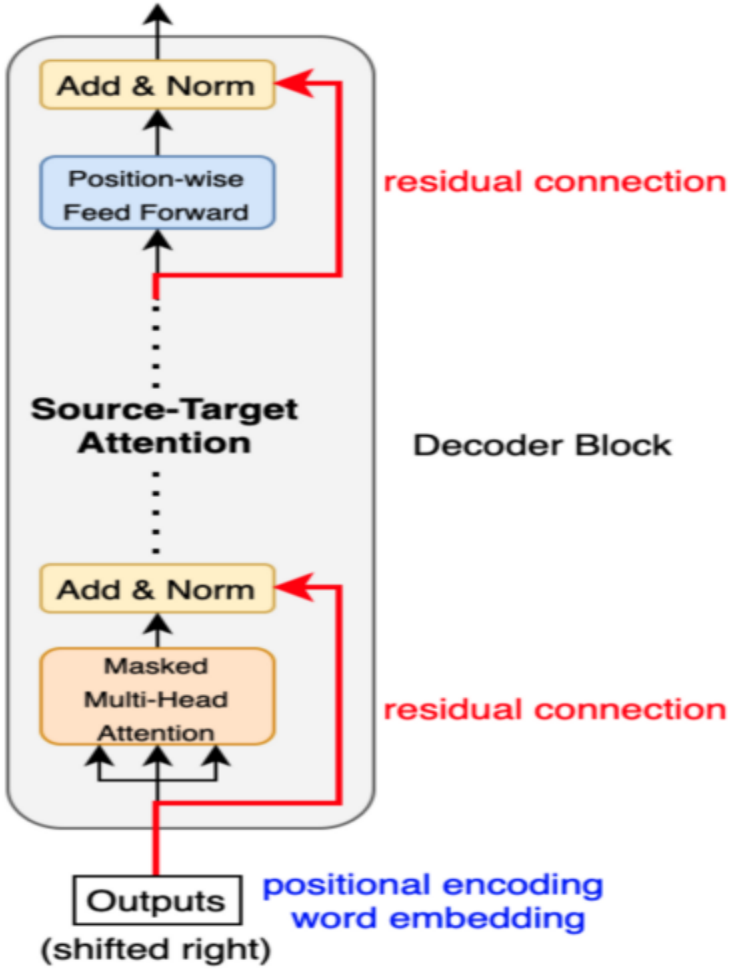
Unlike batch normalization, layer normalization works at each embedding vector (not at the batch level).

The transformer uses six stacked encoder blocks. The outputs from the last encoder block become the input features for the decoder.



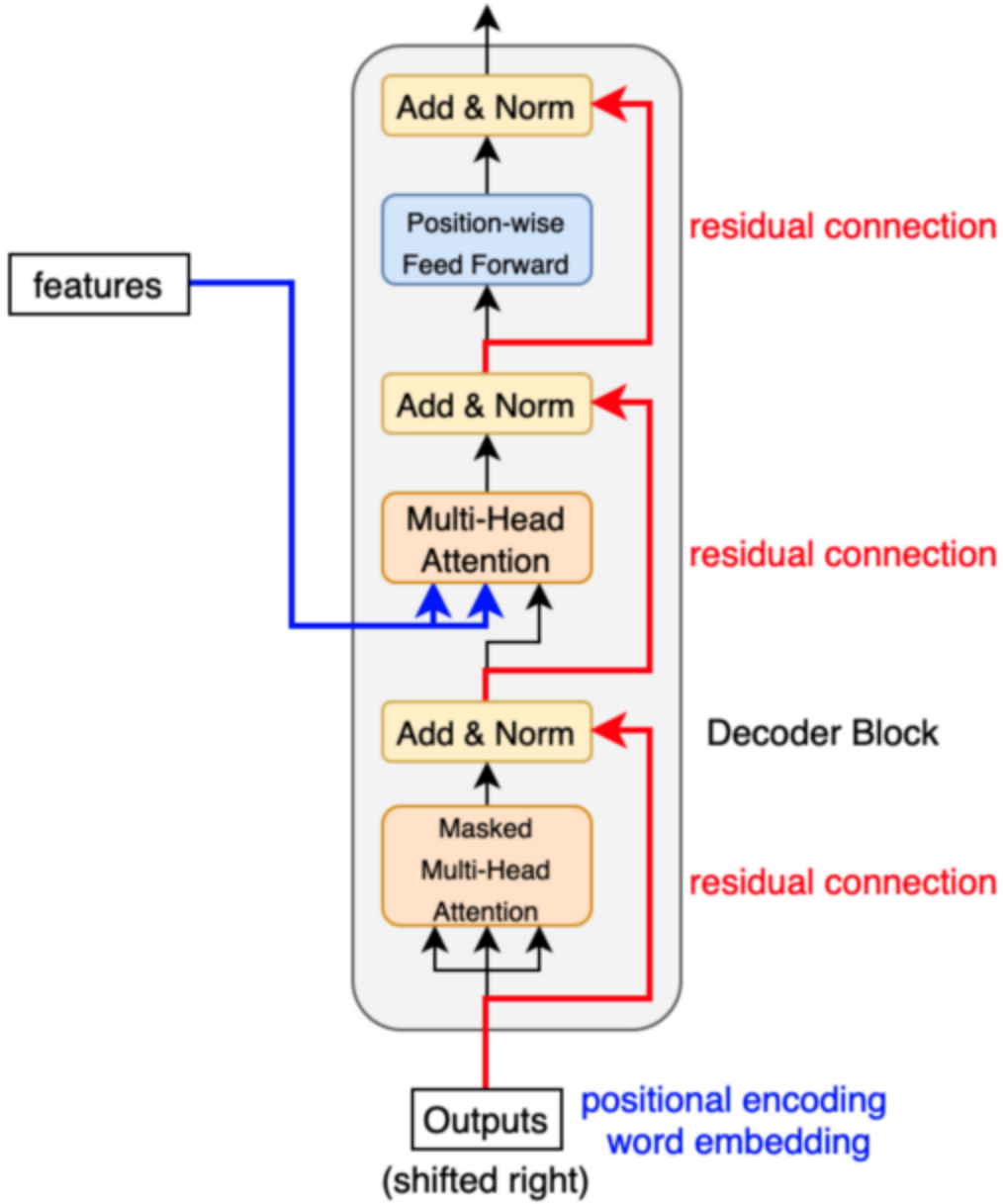
Decoder Block Internals

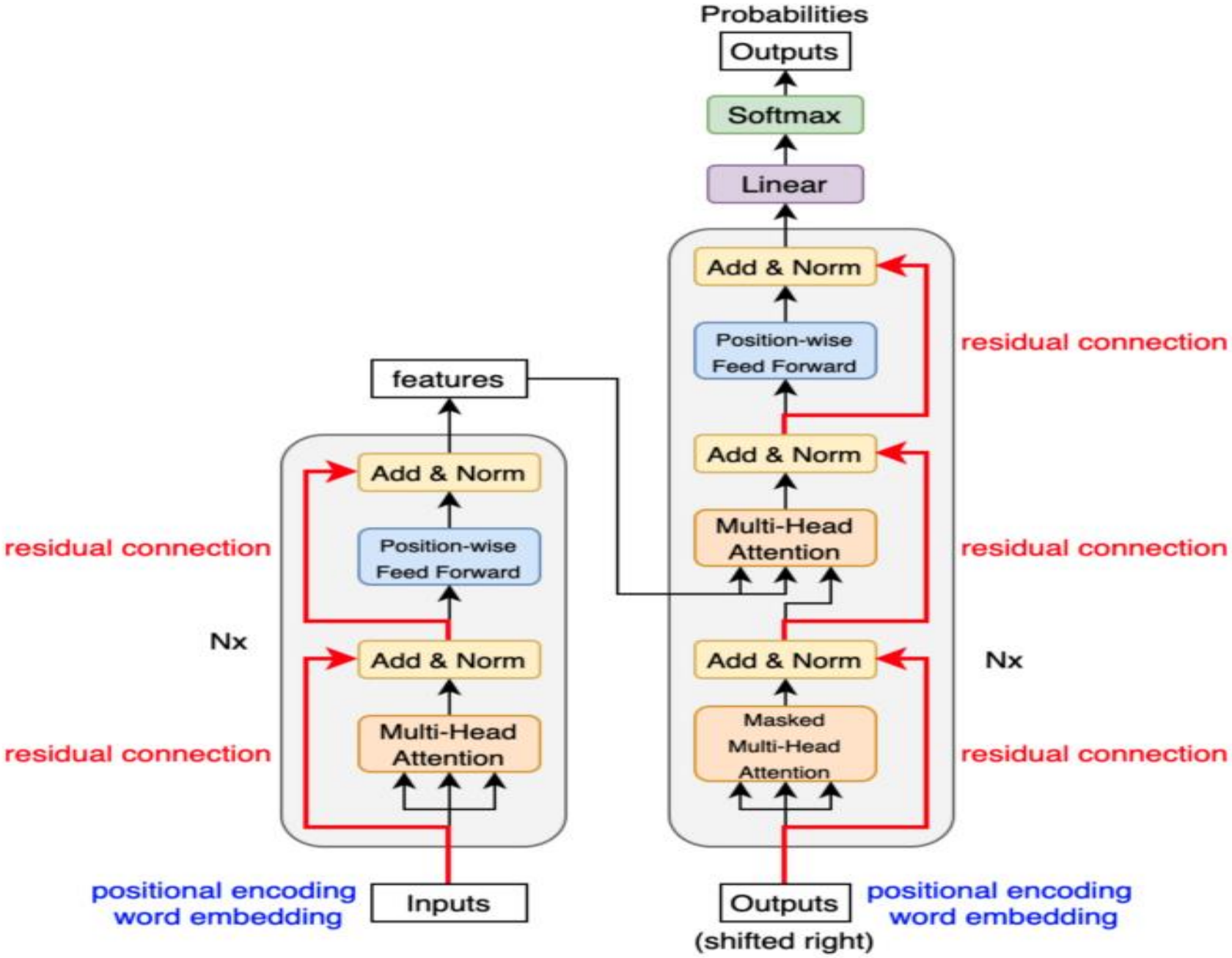
The decoder block is similar to the encoder block, except it calculates the source-target attention.

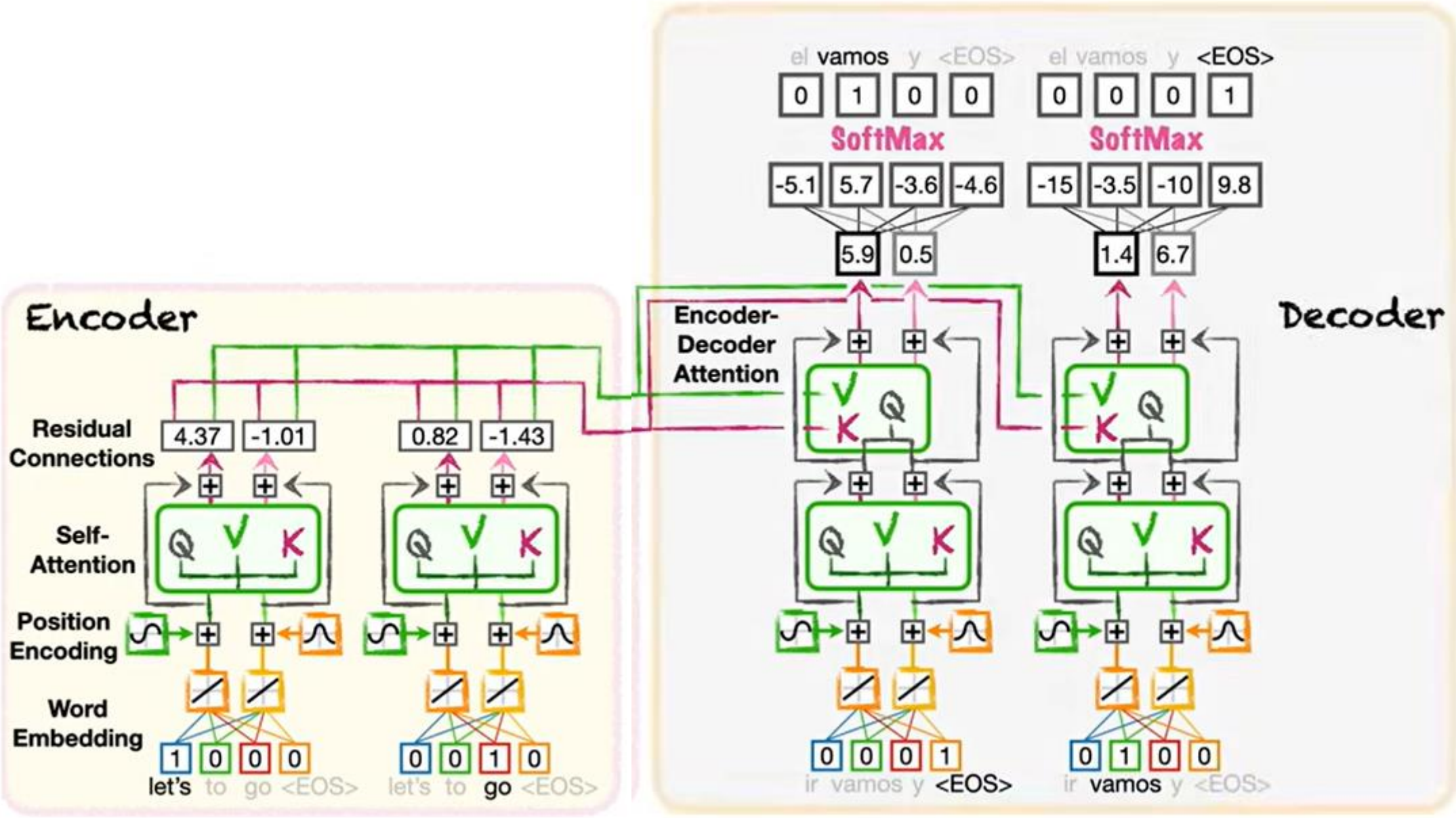


The input to the decoder is an output shifted right, which becomes a sequence of embeddings with positional encoding. So, we can think of the decoder block as another encoder generating enriched embeddings useful for translation outputs.

Masked multi-head attention means the multi-head attention receives inputs with masks so that the attention mechanism does not use information from the hidden (masked) positions. The paper mentions that they used the mask inside the attention calculation by setting attention scores to negative infinity (or a very large negative number). The softmax within the attention mechanisms effectively assigns zero probability to masked positions.


Add & Norm
Position-wise
Feed Forward
features
Add & Norm
Multi-Head
Attention
Add & Norm
Masked
Multi-Head
Attention
Outputs
(shifted right)
residual connection
residual connection
Decoder Block
residual connection
positional encoding
word embedding


Probabilities
Outputs
Softmax
Linear
Add & Norm
Position-wise Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
residual connection
residual connection
residual connection
Nx
Nx
features
Add & Norm
Position-wise Feed Forward
Add & Norm
Multi-Head Attention
residual connection
residual connection
residual connection
positional encoding
word embedding
Inputs
Outputs
(shifted right)
positional encoding
word embedding

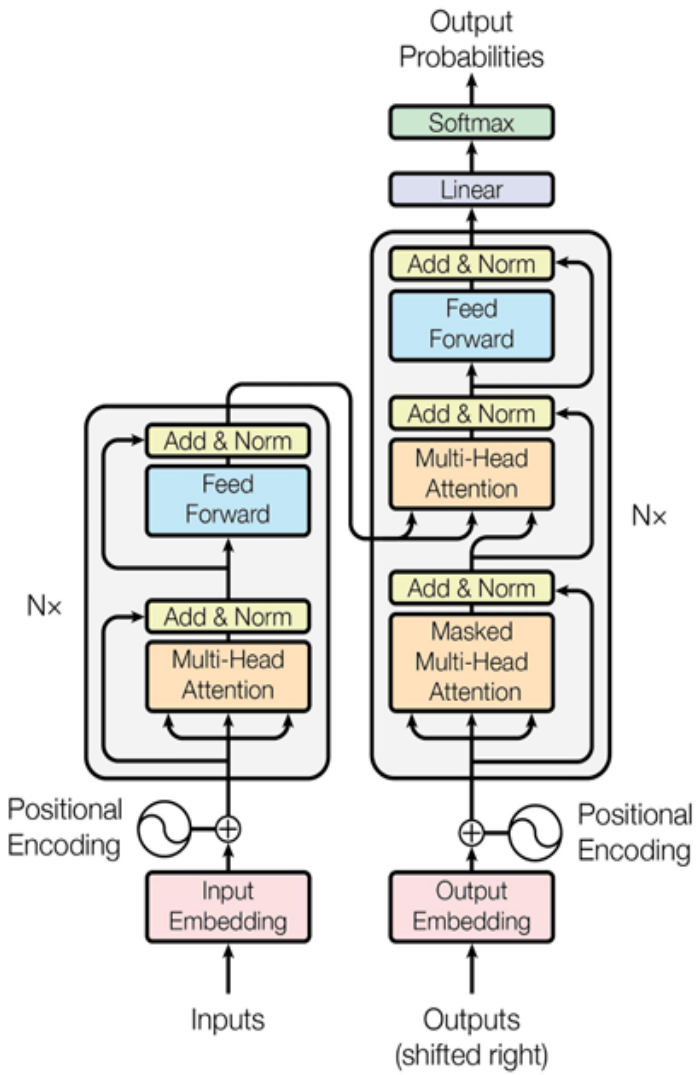

Encoder
Residual Connections
4.37
-1.01
0.82
-1.43
Self-Attention
Q
V
K
Position Encoding
Word Embedding
1 0 0 0
let's to go <EOS>
0 0 1 0
let's to go <EOS>
Encoder-Decoder Attention
Decoder
el vamos y <EOS>
0 1 0 0
SoftMax
-5.1 5.7 -3.6 -4.6
5.9 0.5
el vamos y <EOS>
0 0 0 1
SoftMax
-15 -3.5 -10 9.8
1.4 6.7
0 0 0 1
ir vamos y <EOS>
0 1 0 0
ir vamos y <EOS>

Language translation example

Inputs

Processing Inputs

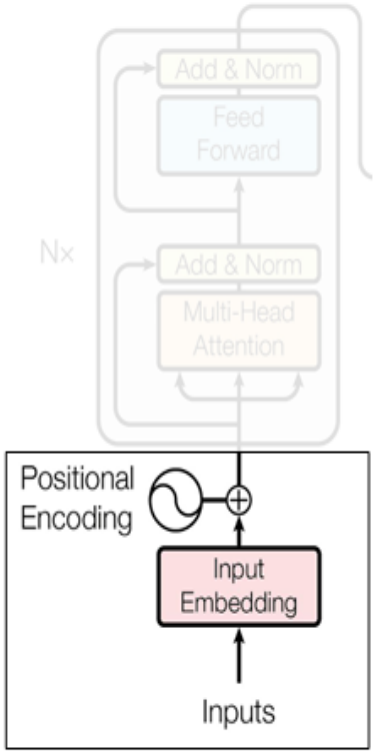
Inputs
I ate an apple



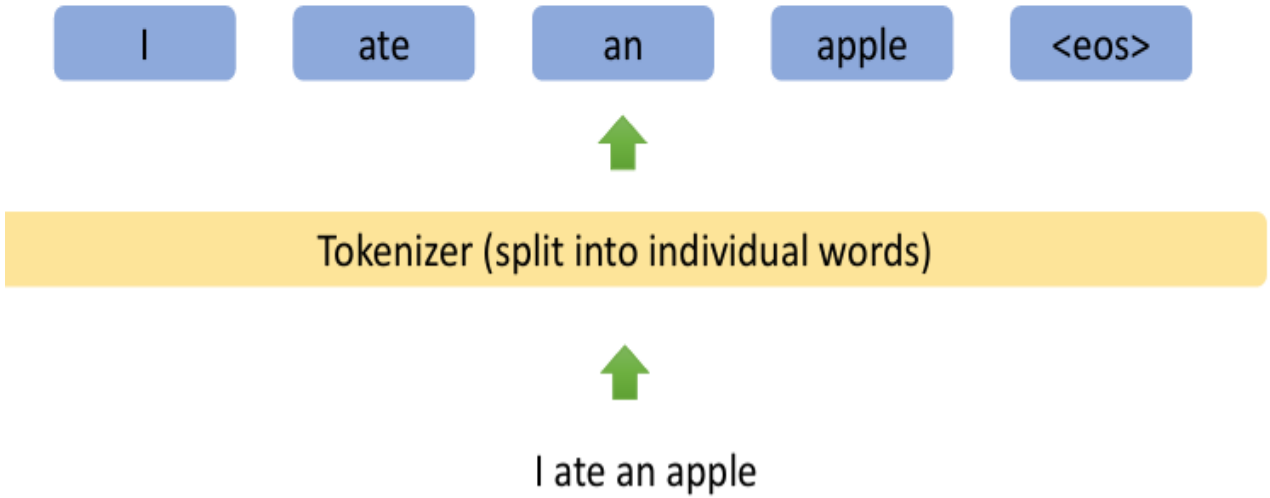
Tokenization

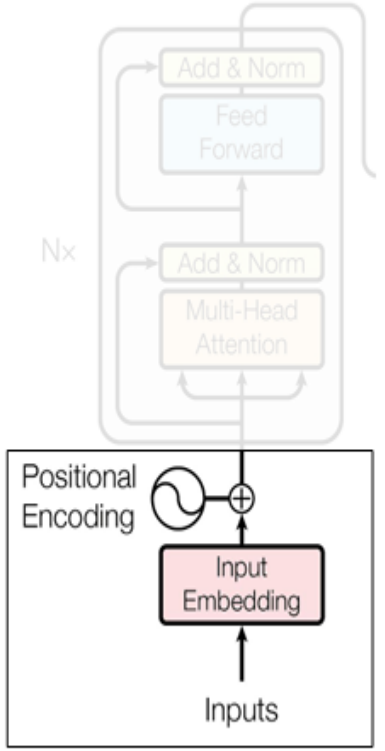
Tokenizer (split into individual words)

I ate an apple

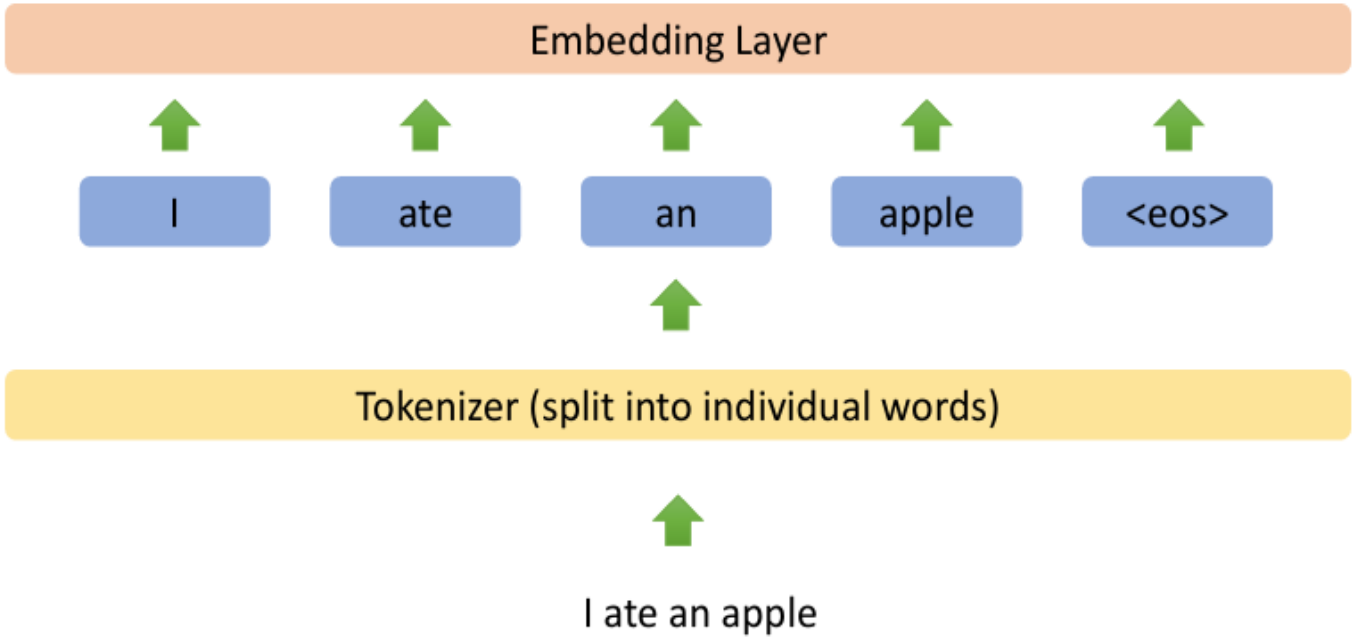


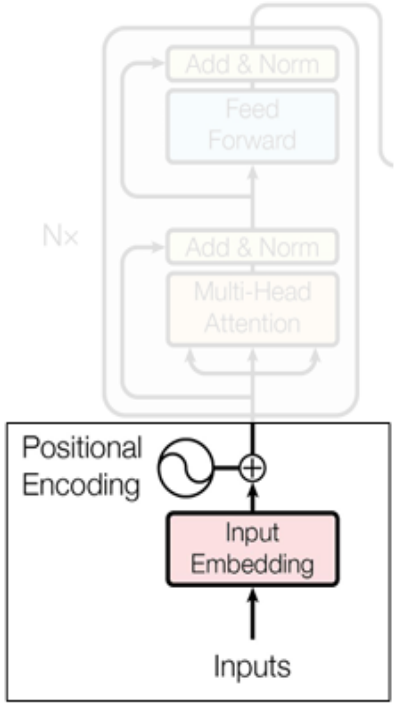
Tokenization





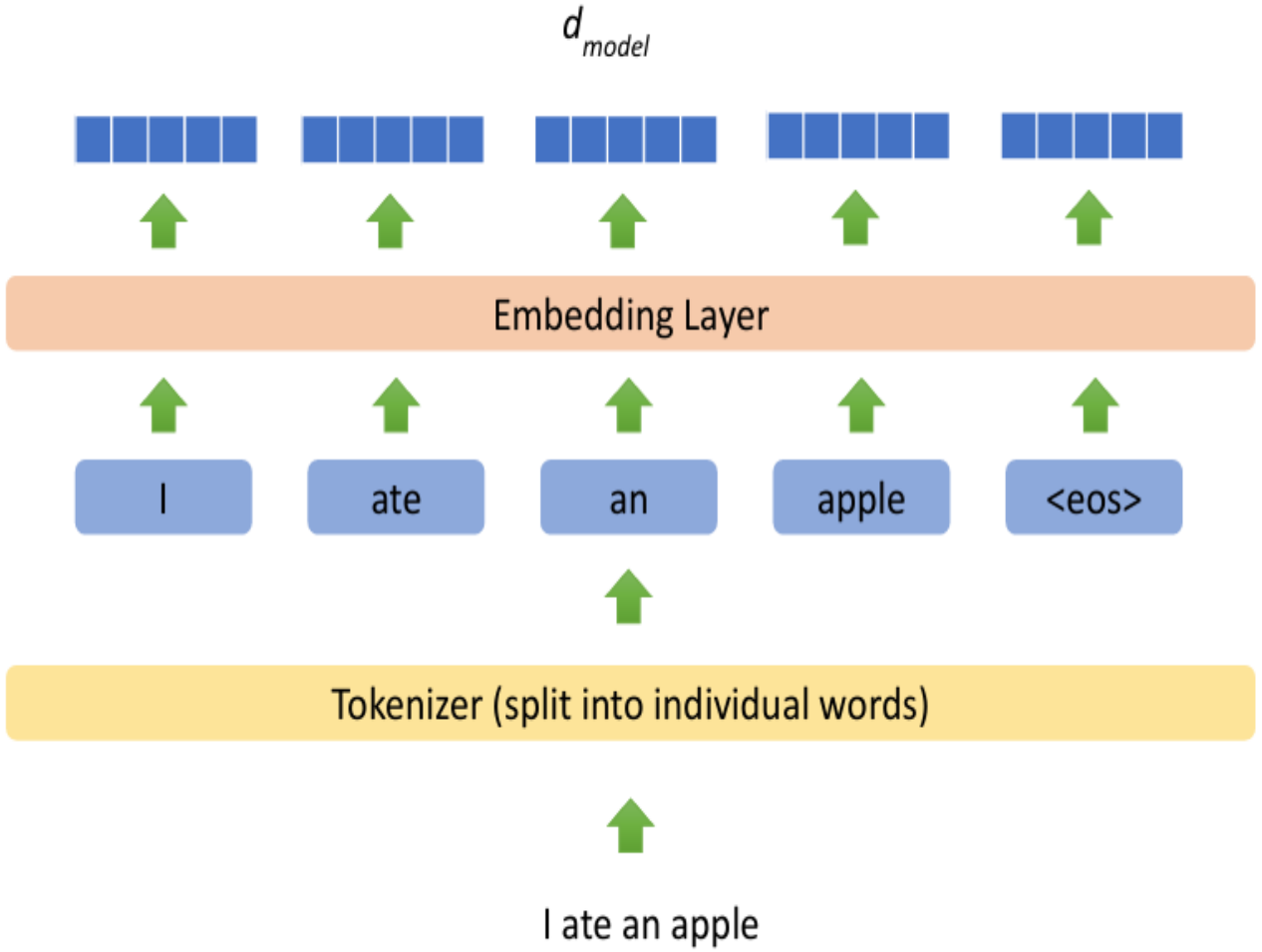
Input Embeddings

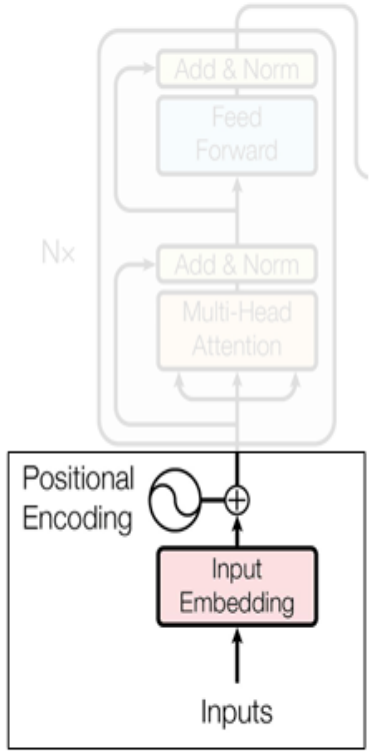




Generate Input Embeddings

Input Embeddings

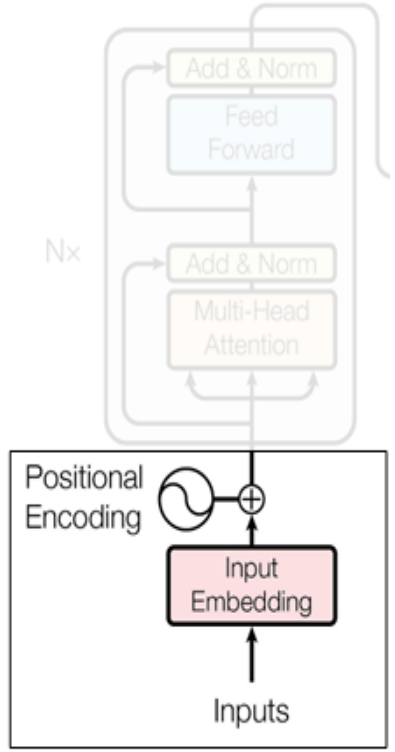




Generate Input Embeddings

Position Encodings

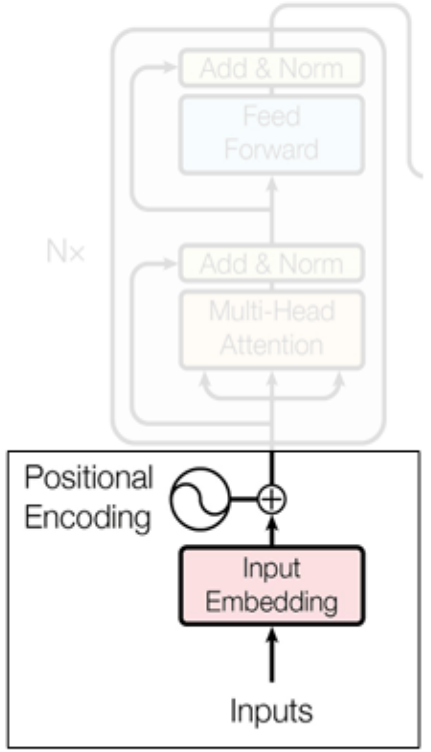
I	ate	an	apple	<eos>



Position Encodings

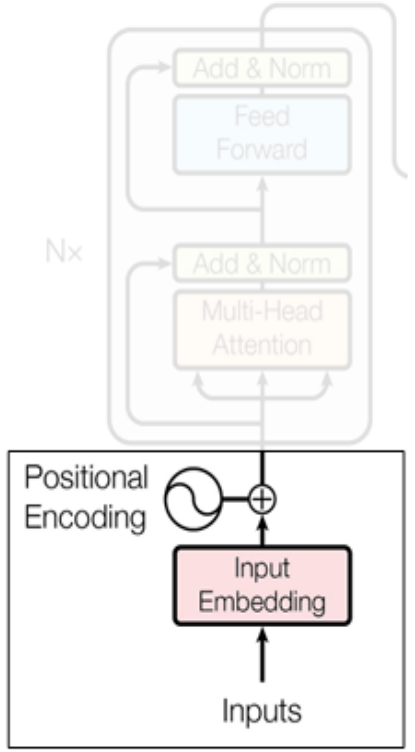
I	ate	an	apple	<eos>

apple	ate	an	I	<eos>



Position Encodings

Requirements for Positional Encodings???



Position Encoding

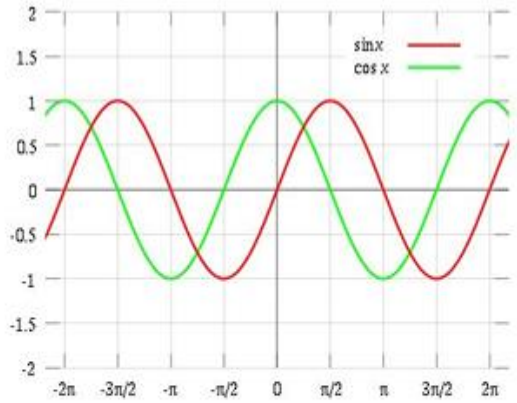
Requirements for Position Encodings

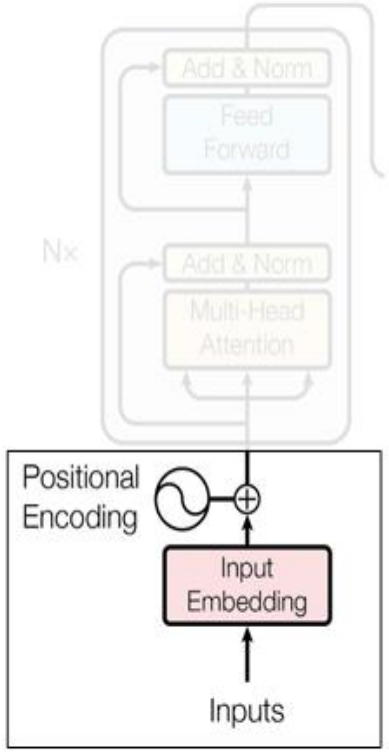
- Some representation of time? (like **seq2seq**?)
- Should be unique for each position
- Bounded

Actual Candidates

*sine(**g(t)**)*

*cosine(**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**.

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

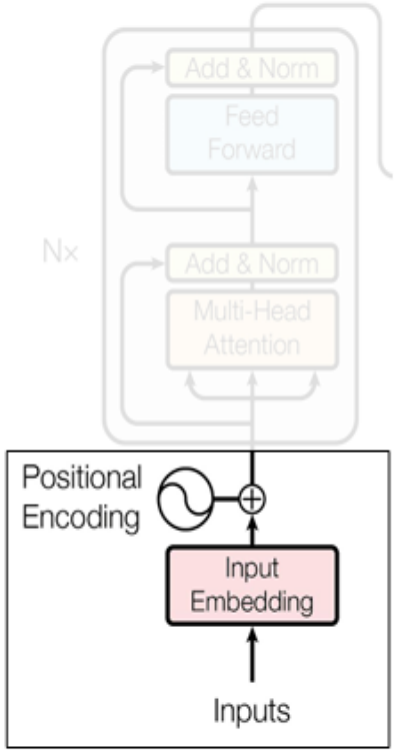
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

pos -> idx of the token in input sentence

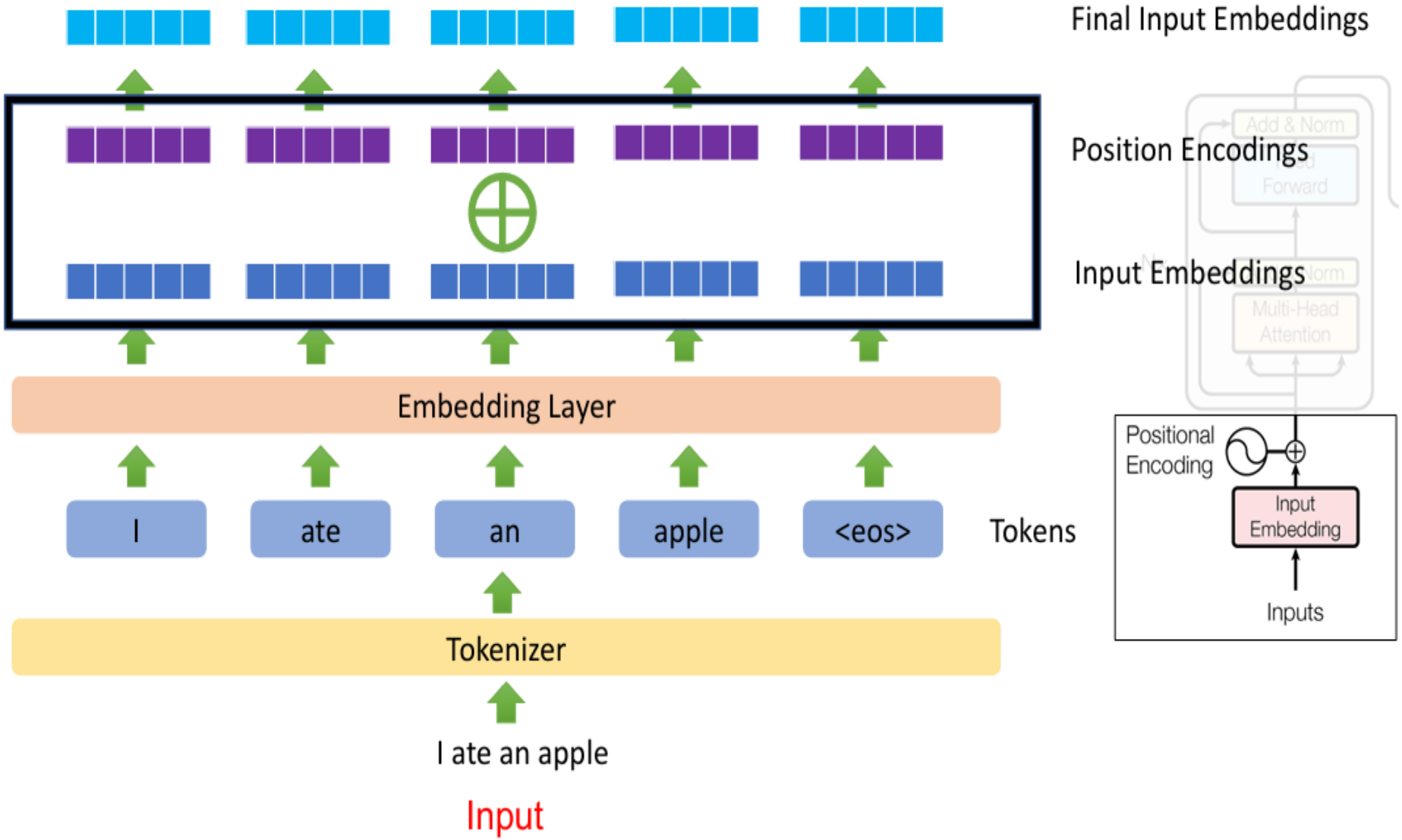
i -> i^{th} dimension out of d

d model -> embedding dimension of each token

Different calculations for odd and even embedding indices

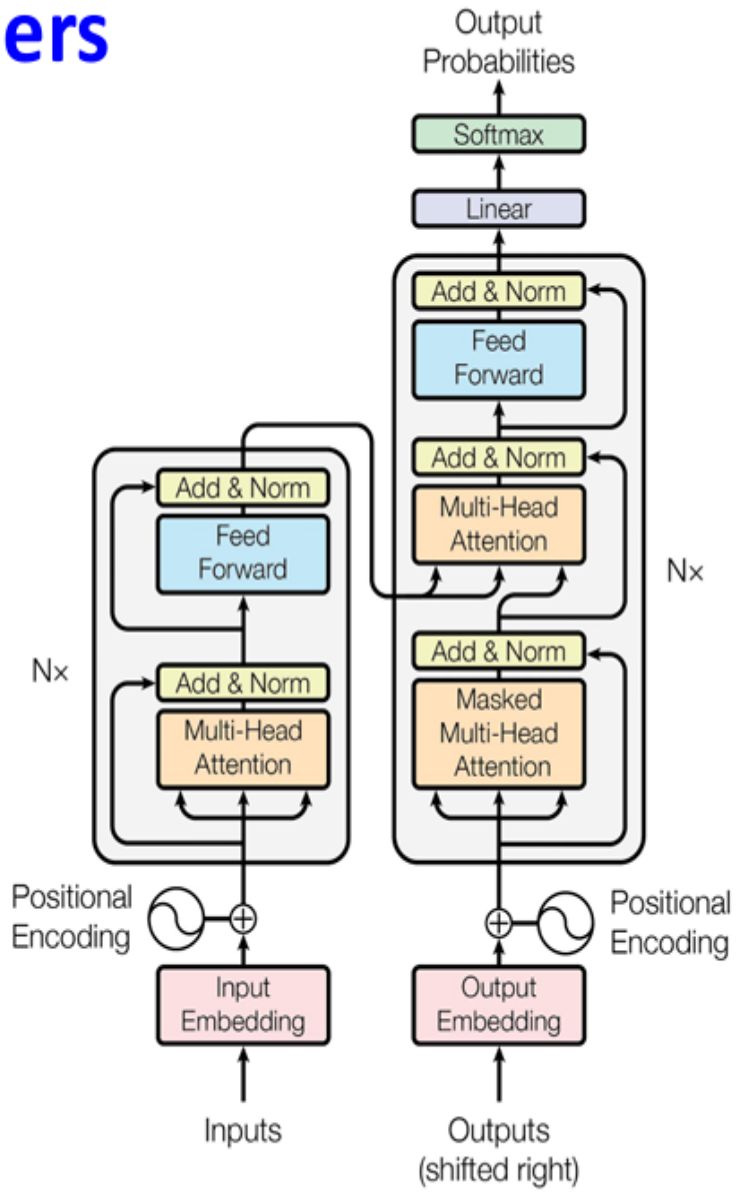


Position Encoding



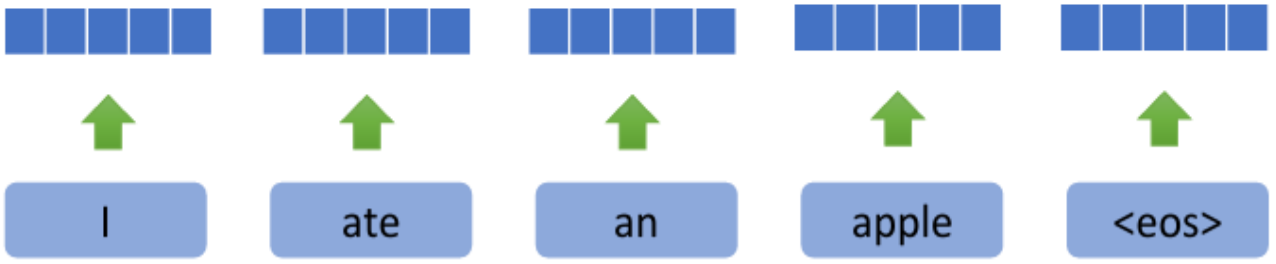
Transformers

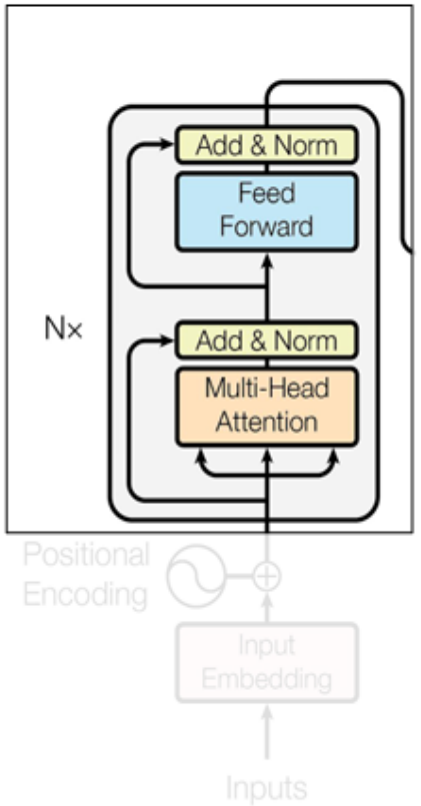
- ✔ Tokenization
- ✔ Input Embeddings
- ✔ Position Encodings
- Query, Key, & Value
- Attention
- Self Attention
- Multi-Head Attention
- Feed Forward
- Add & Norm
- Encoders
- Masked Attention
- Encoder Decoder Attention
- Linear
- Softmax
- Decoders
- Encoder-Decoder Models



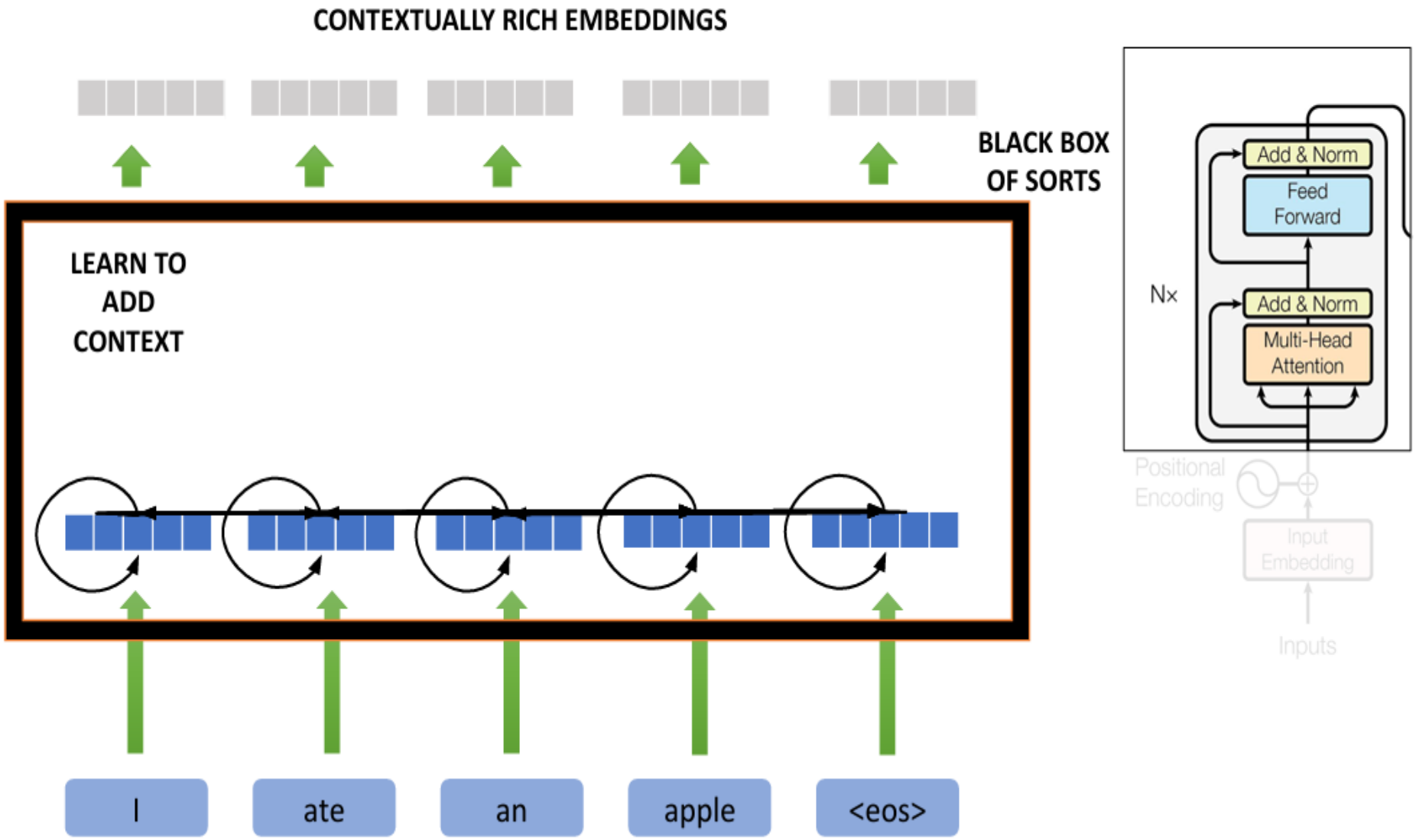
Encoder

WHERE IS THE CONTEXT ?



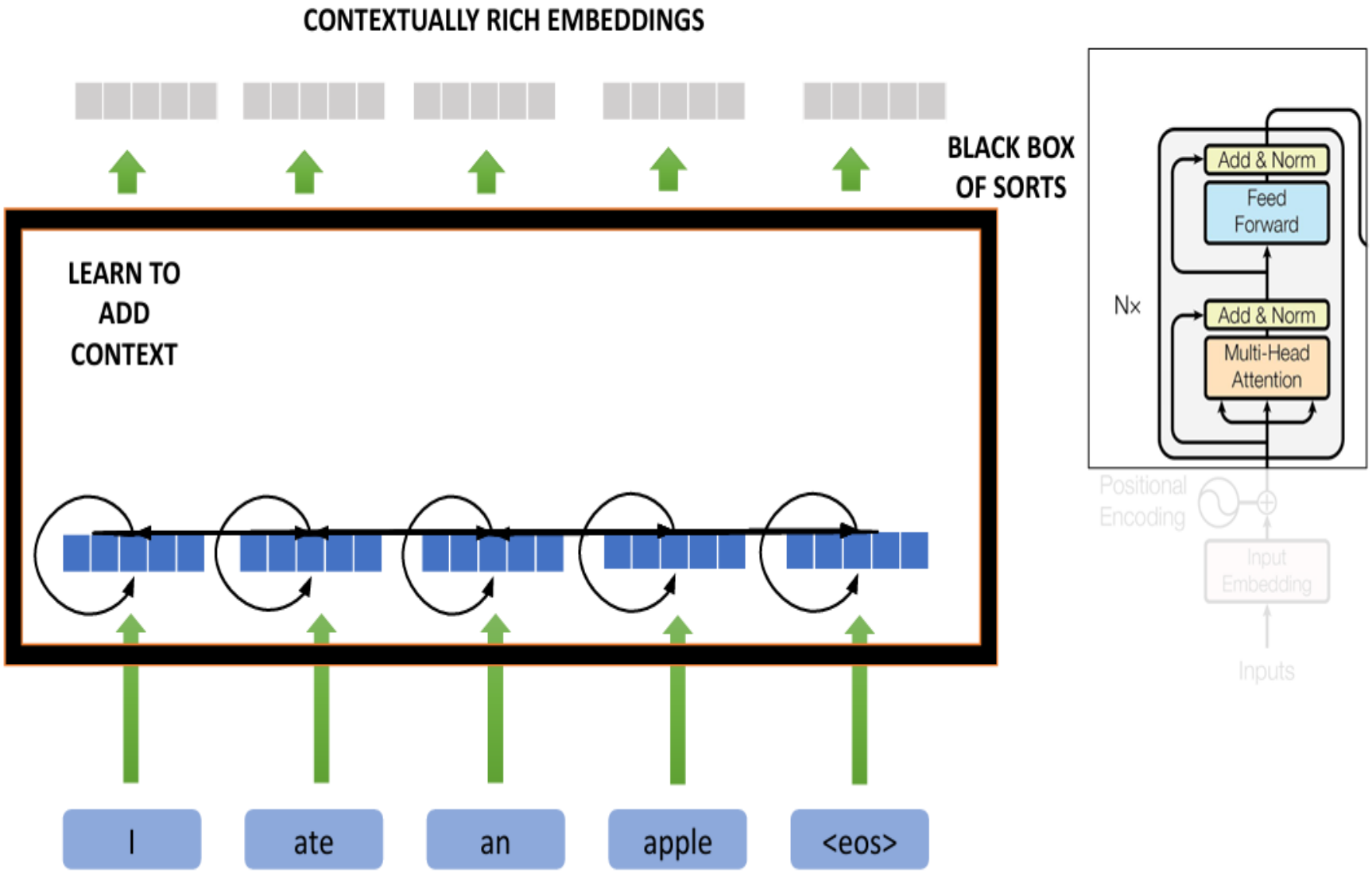


Encoder


CONTEXTUALLY RICH EMBEDDINGS
BLACK BOX OF SORTS
LEARN TO ADD CONTEXT
I
ate
an
apple
<eos>
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Positional Encoding
Input Embedding
Inputs

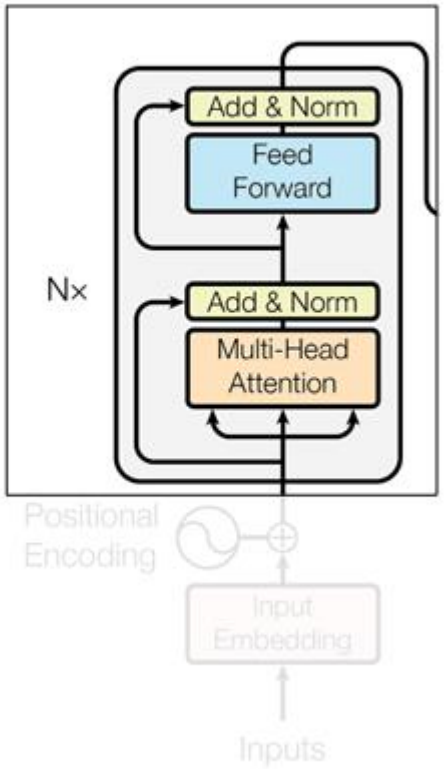
Encoder

$\alpha_{[ij]}$?



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

- 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

1. Search for info

Key

1. Interacts directly with Queries
2. Distinguishes one object from another
3. Identify which object is the most relevant and by how much

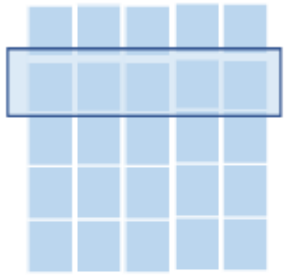
Value

1. Actual details of the object
2. More fine grained

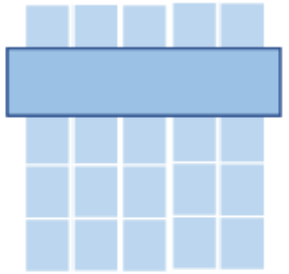
Attention

Query

Key Value Store



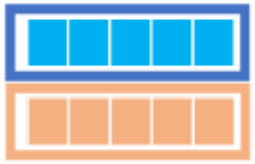
Key



Value

Attention

Parallelizable !!!

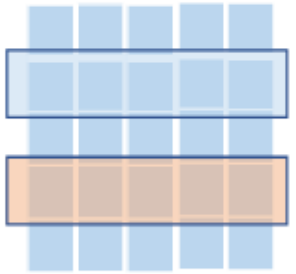


Query

Q

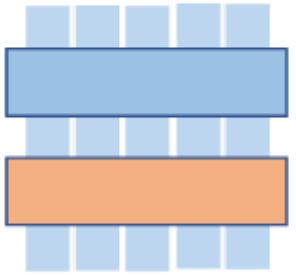
Key Value Store

QK^T



Key

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



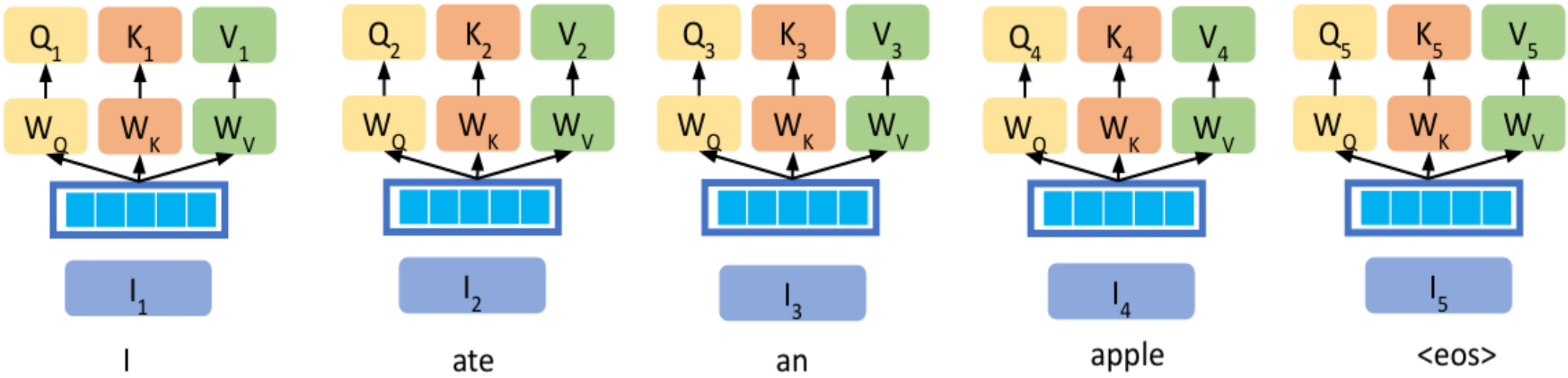
Value

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

Kmit

Dimensions across QKV have been dropped for brevity

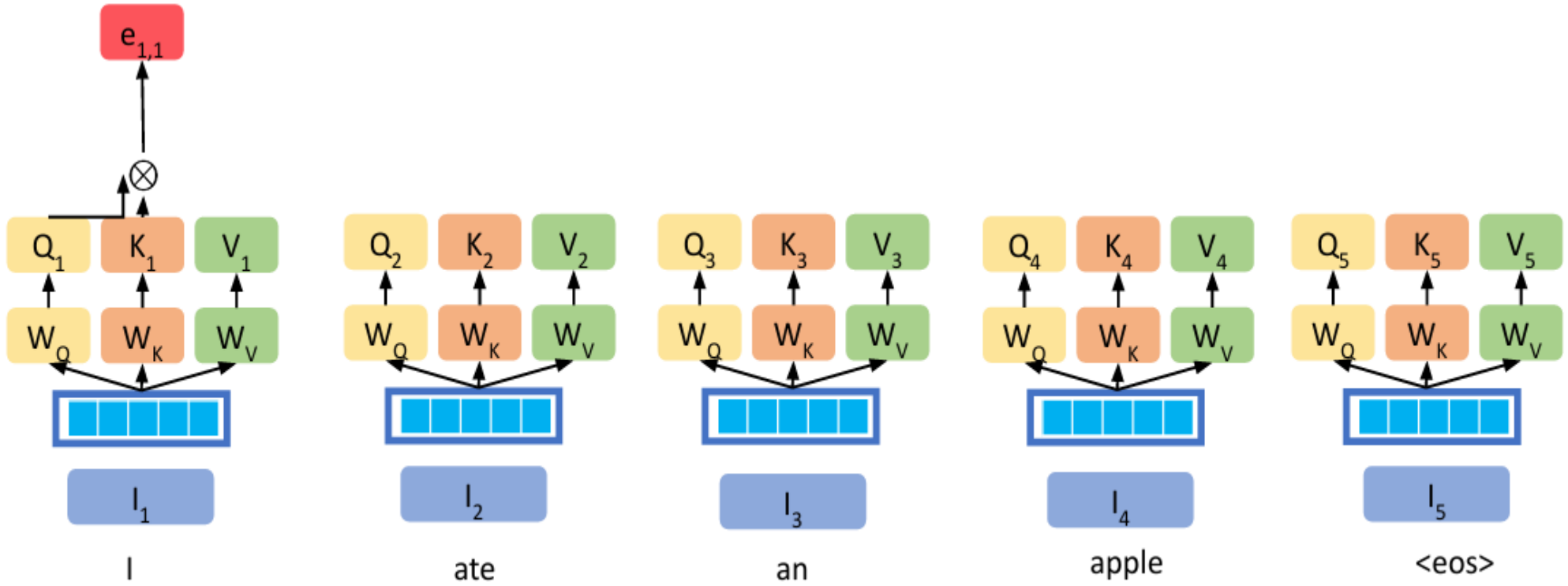
Attention



Kmit

Dimensions across QKV have been dropped for brevity

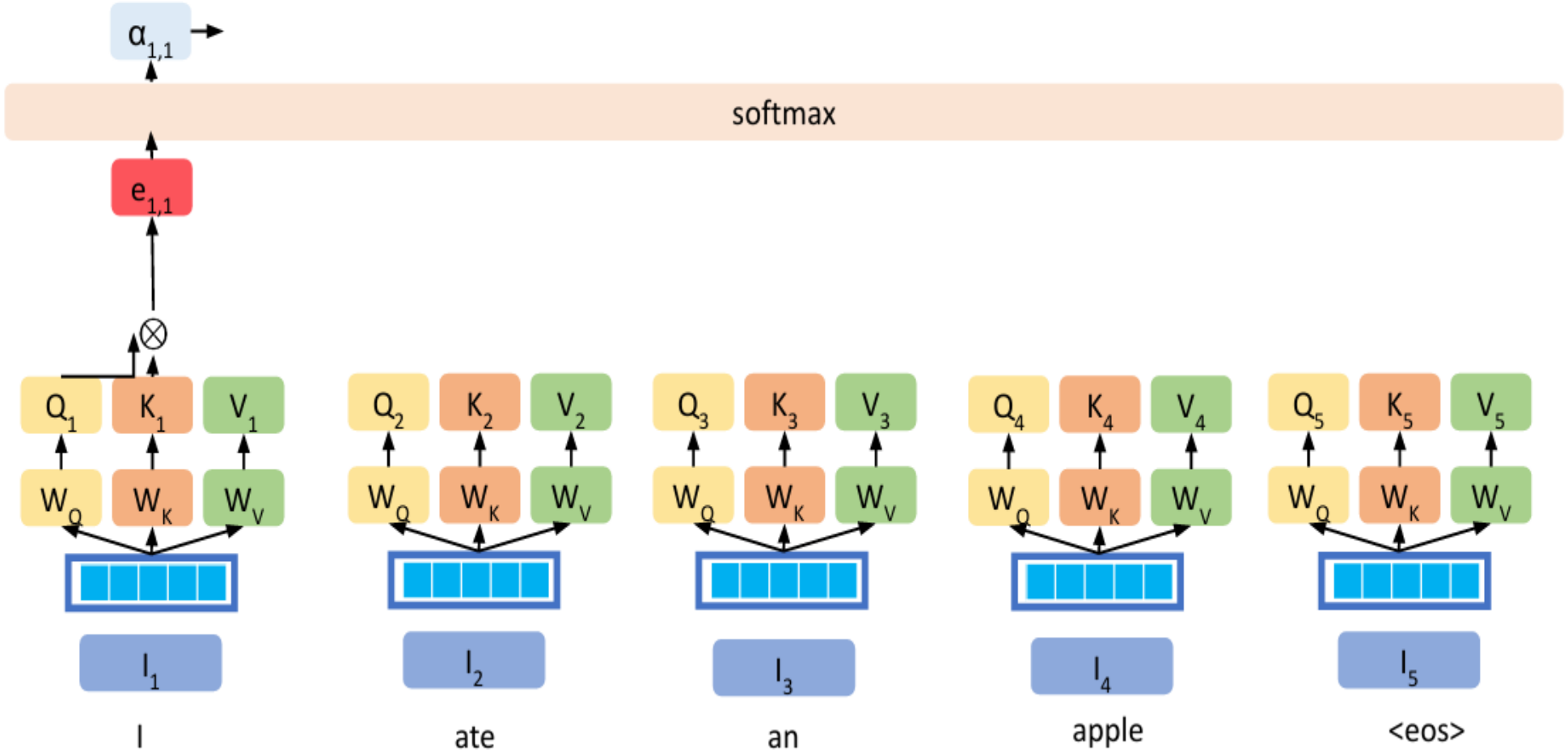
Attention



Kmit

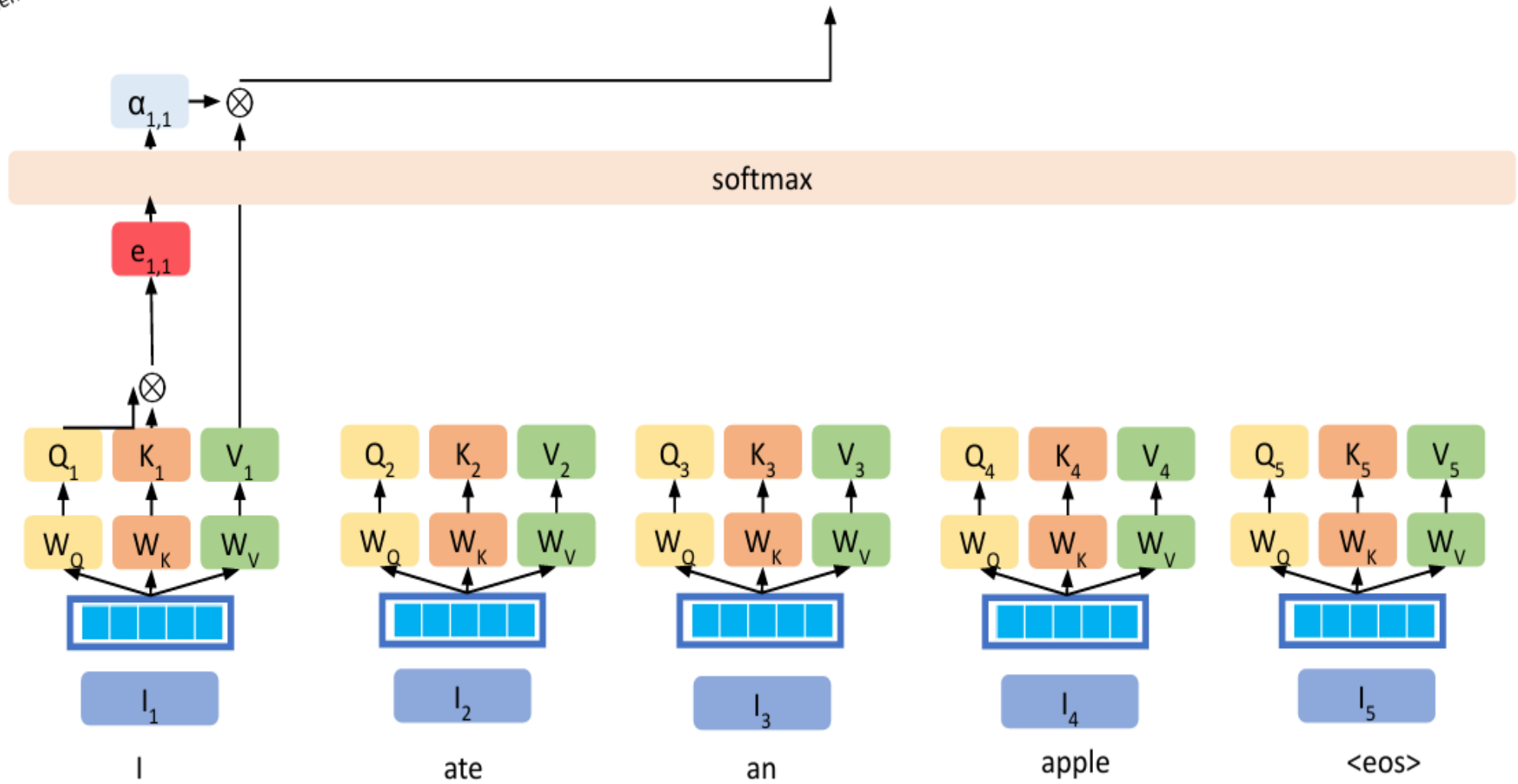
Dimensions across QKV have been dropped for brevity

Attention



Attention

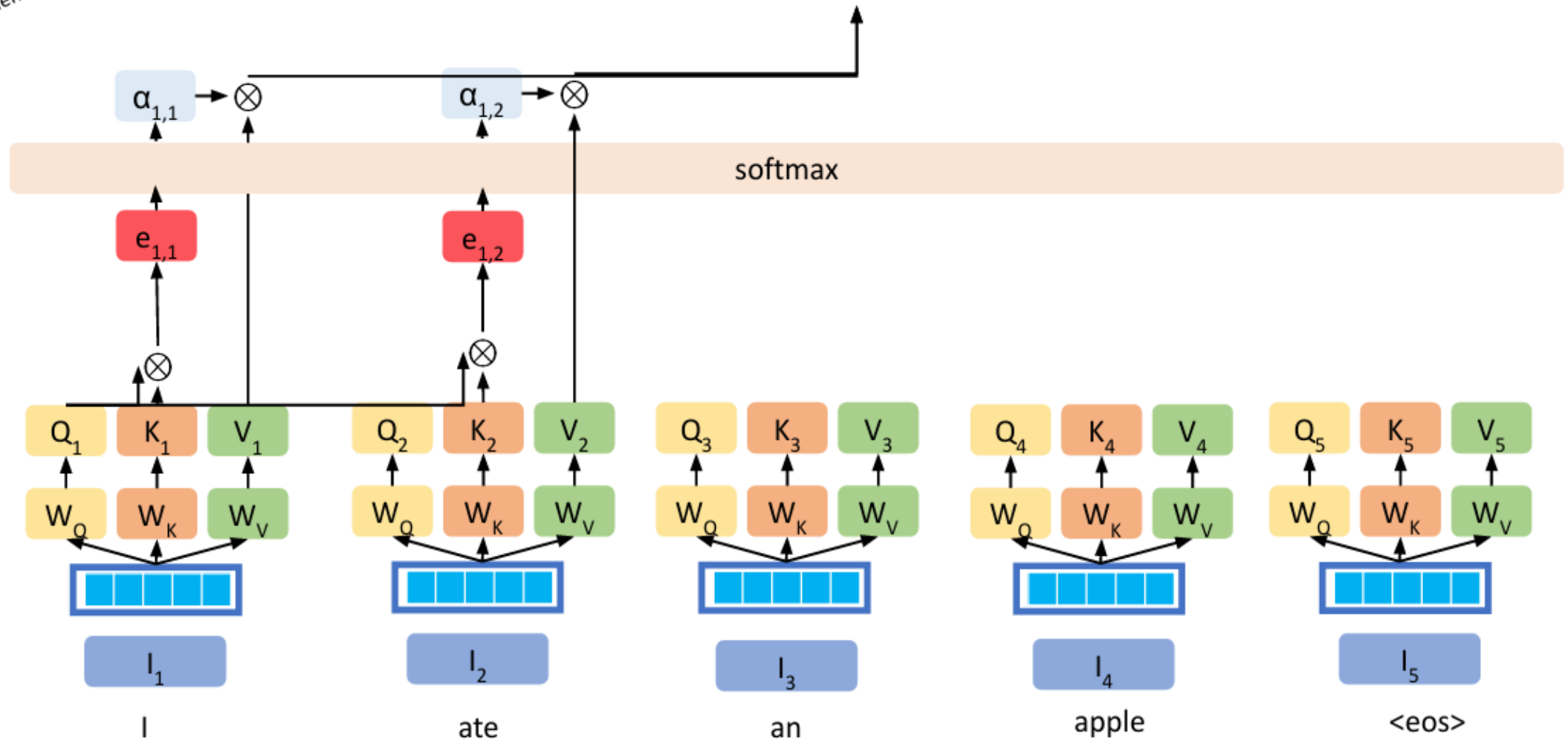
Dimensions across QKV have been dropped for brevity



Kmit

Dimensions across QKV have been dropped for brevity

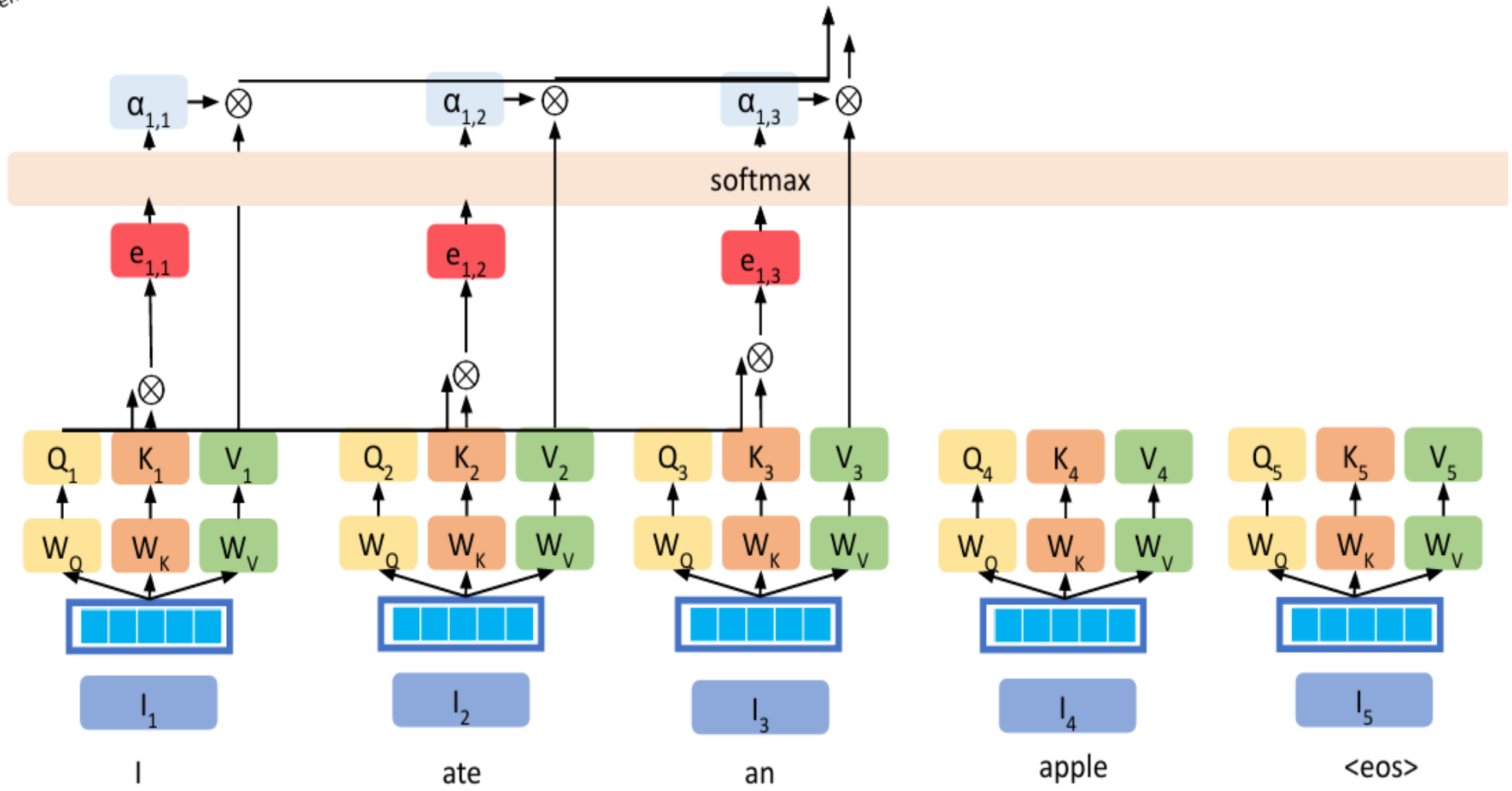
Attention



Kmit

Dimensions across QKV have been dropped for brevity

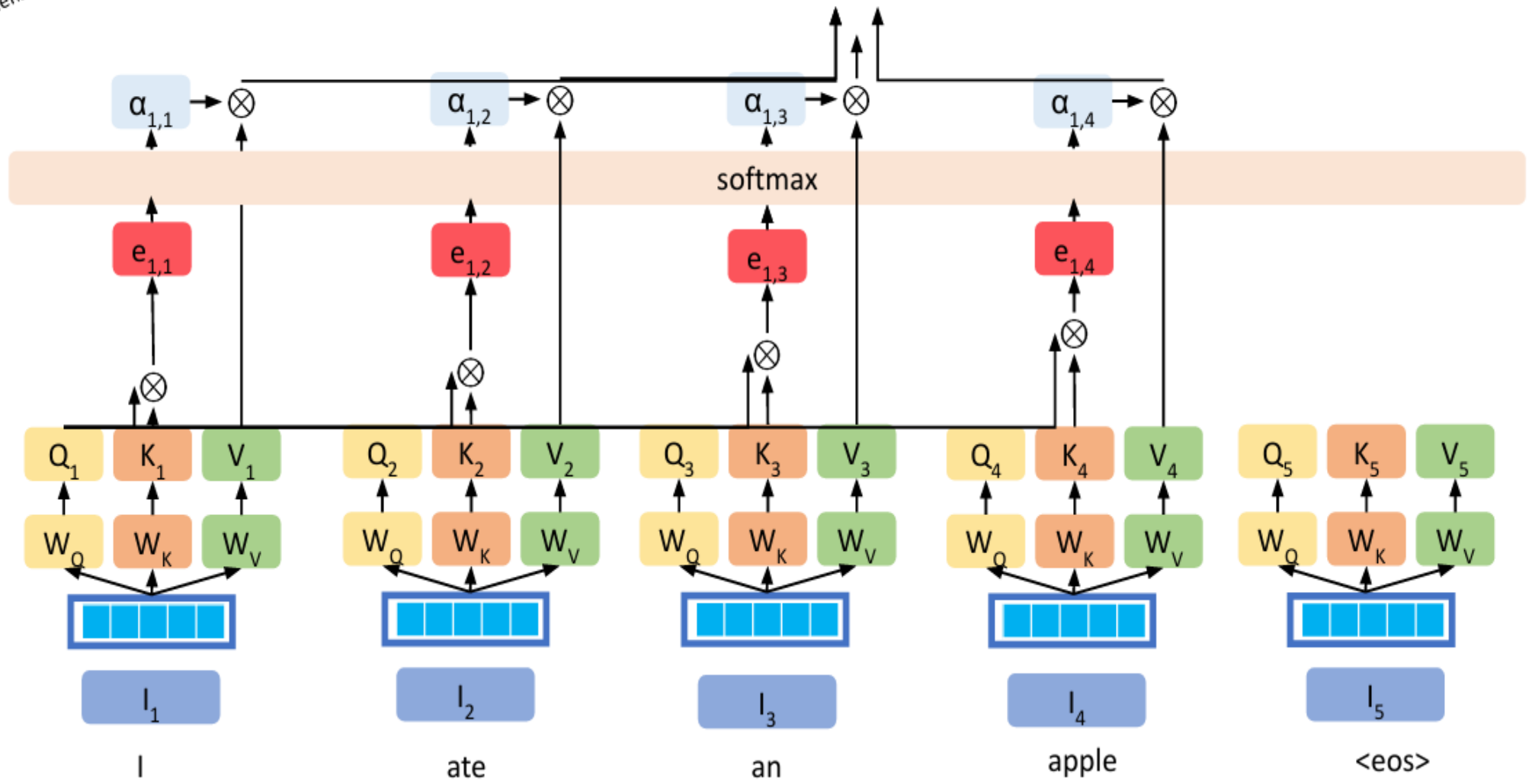
Attention



Kmit

Dimensions across QKV have been dropped for brevity

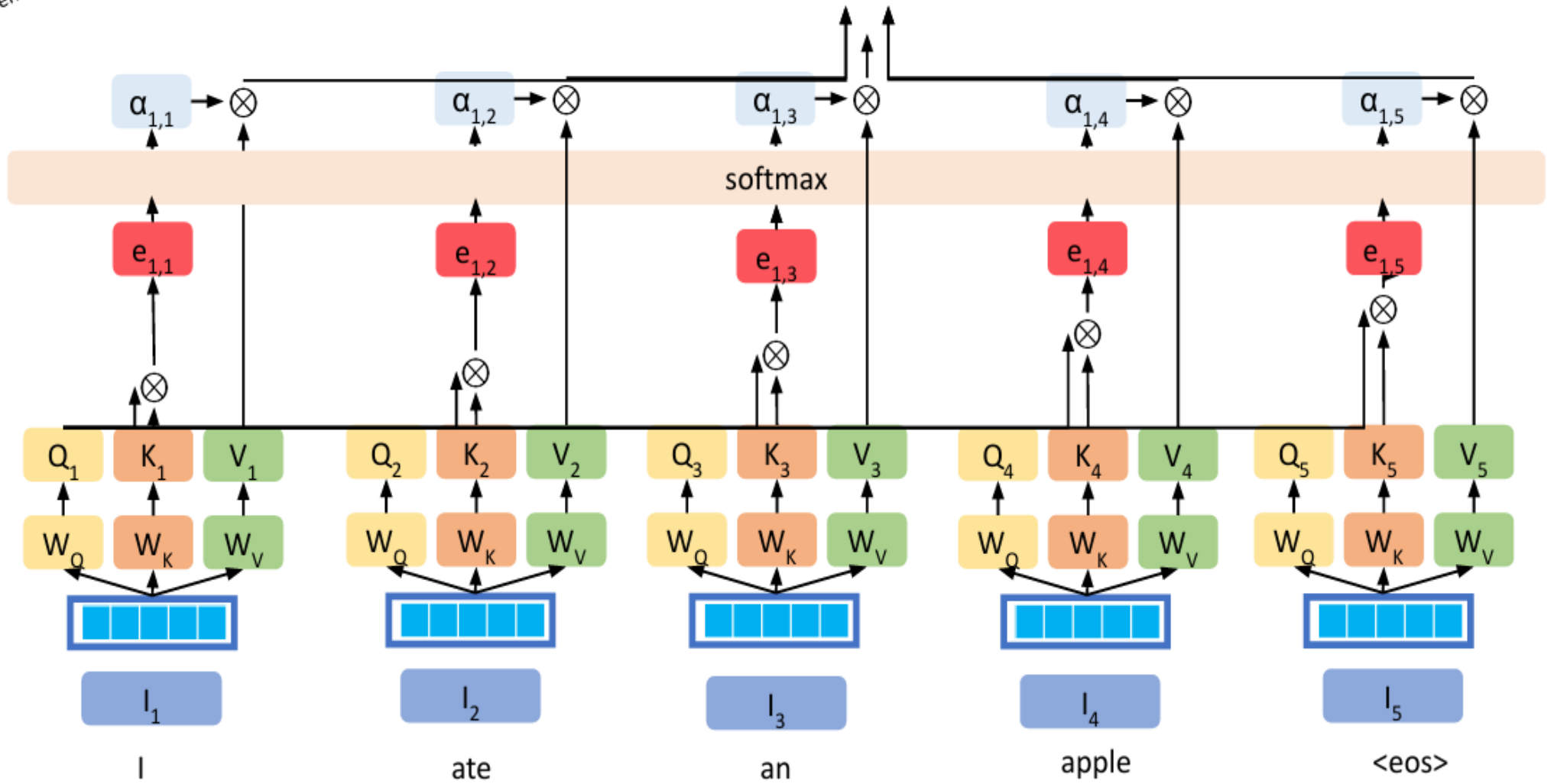
Attention



Kmit

Dimensions across QKV have been dropped for brevity

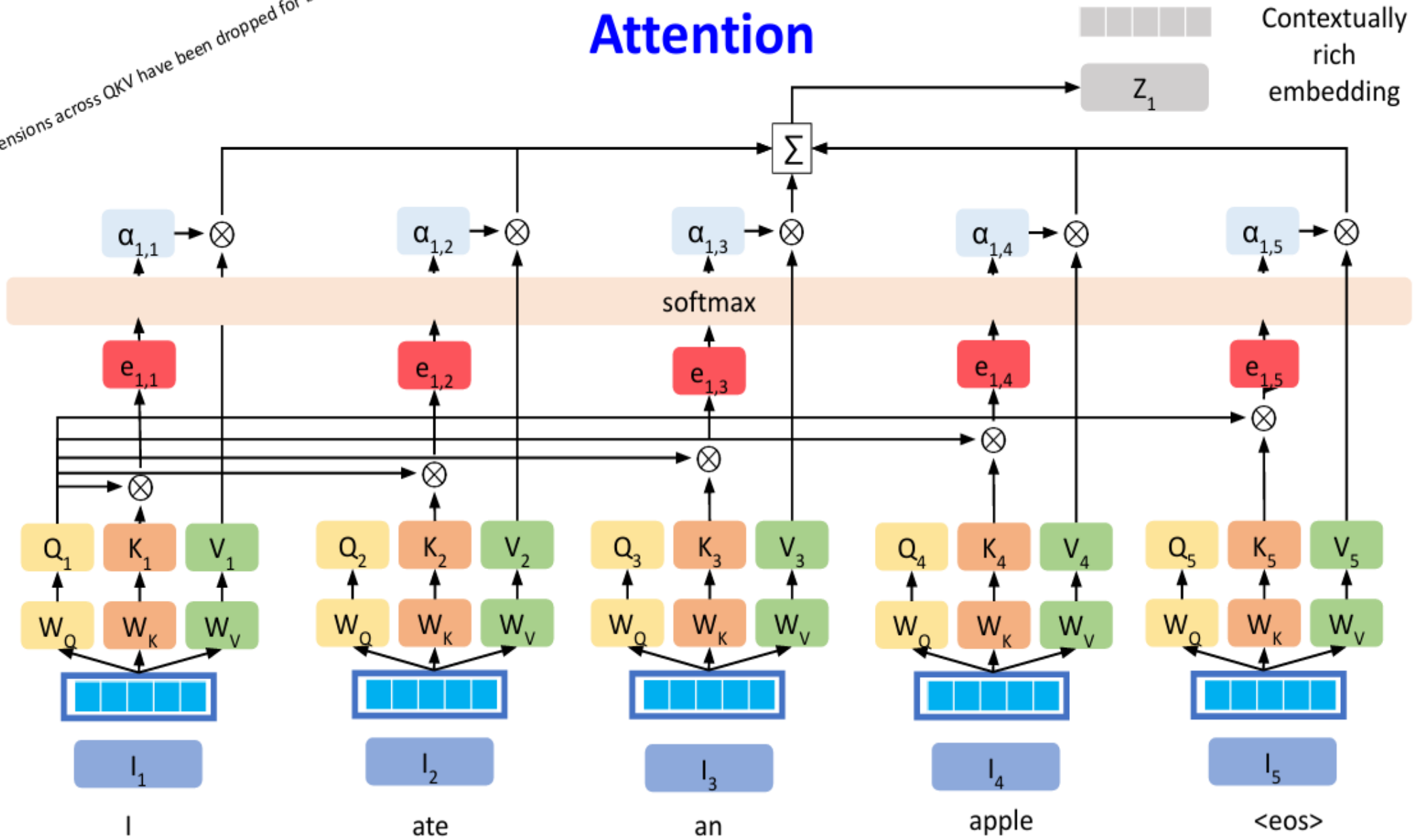
Attention



Kmit

Attention

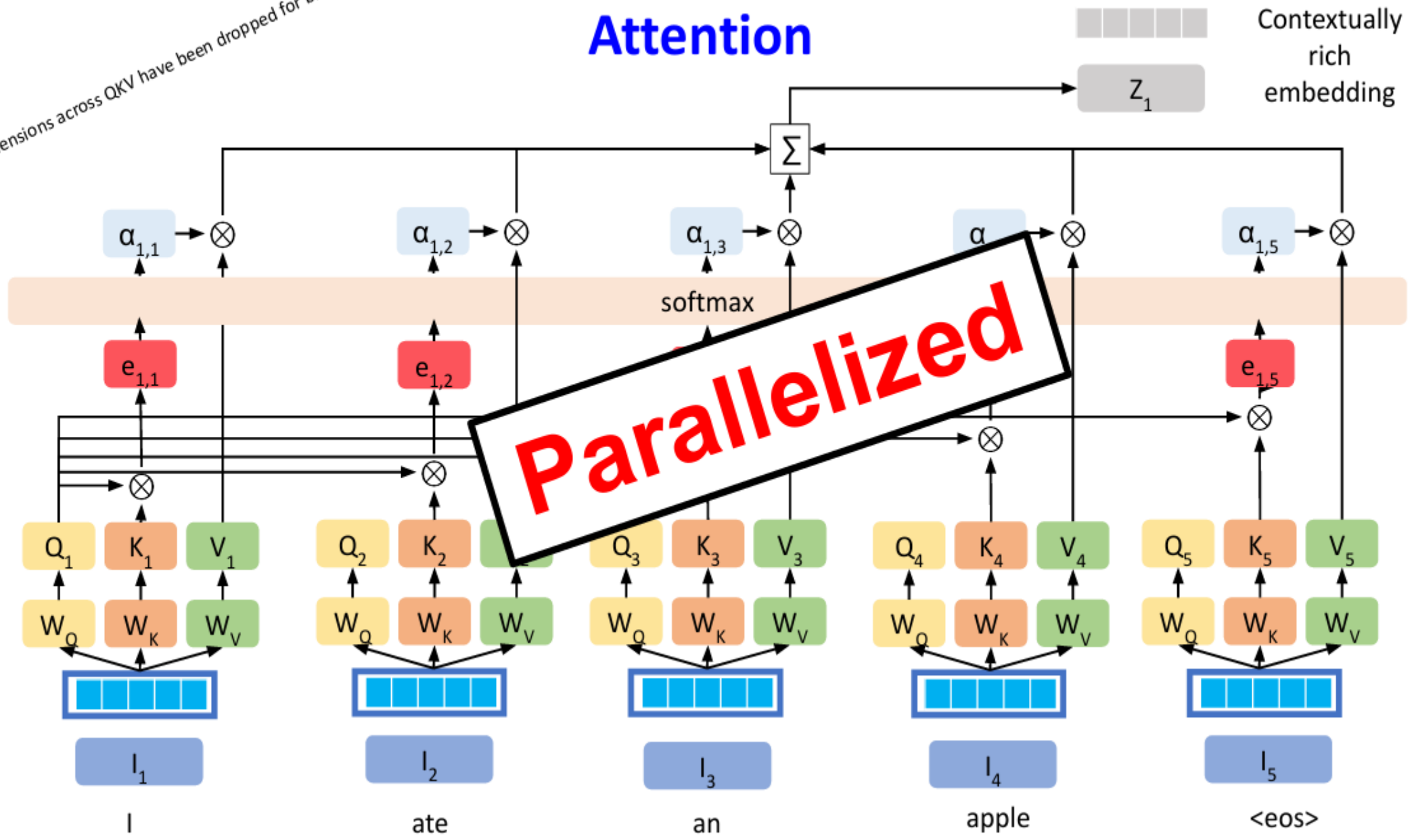
Dimensions across QKV have been dropped for brevity



Kmit

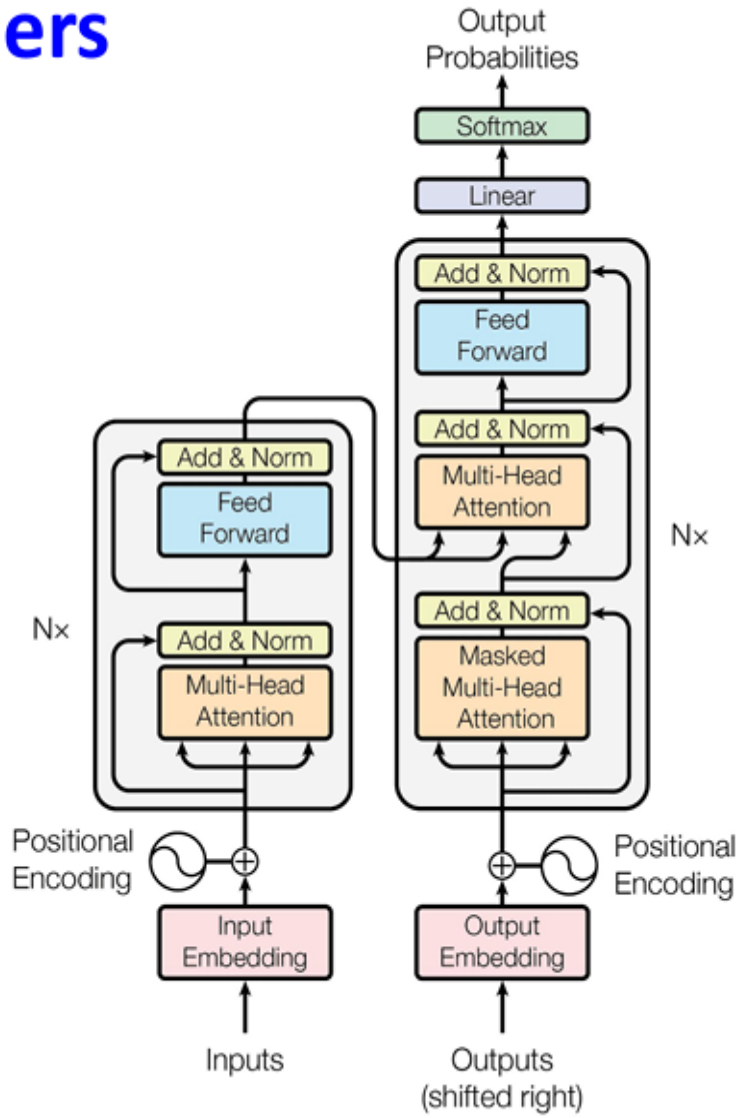
Attention

Dimensions across QKV have been dropped for brevity



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



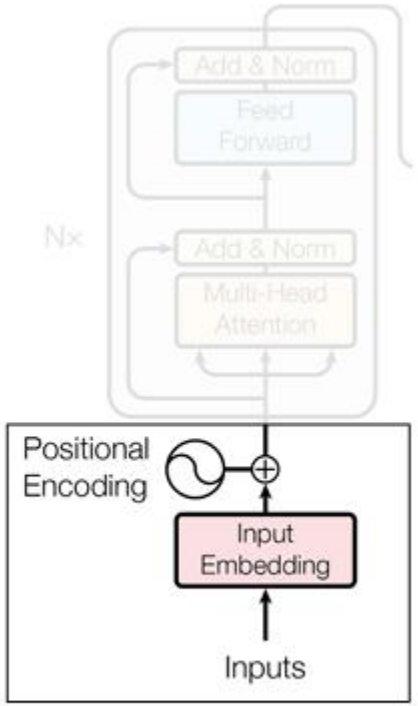
Which of the following are true about attention?

a. To calculate attention weights for input $\mathbf{I}_2$, you would use key $\mathbf{k}_2$, and all queries

b. To calculate attention weights for input $\mathbf{I}_2$, you would use query $\mathbf{q}_2$, and all keys

c. We scale the QK^T product to bring attention weights in the range of [0,1]

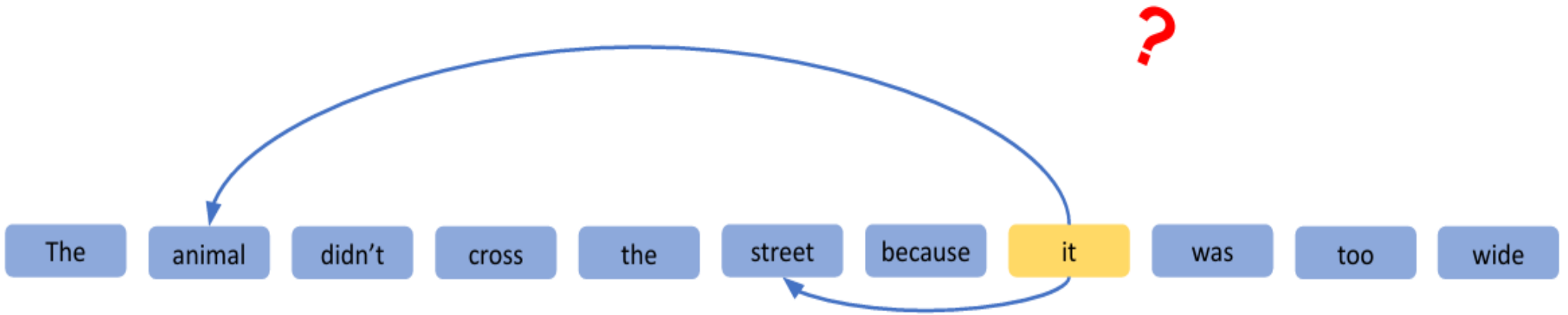
d. We scale the QK^T product to allow for numerical stability



Self Attention

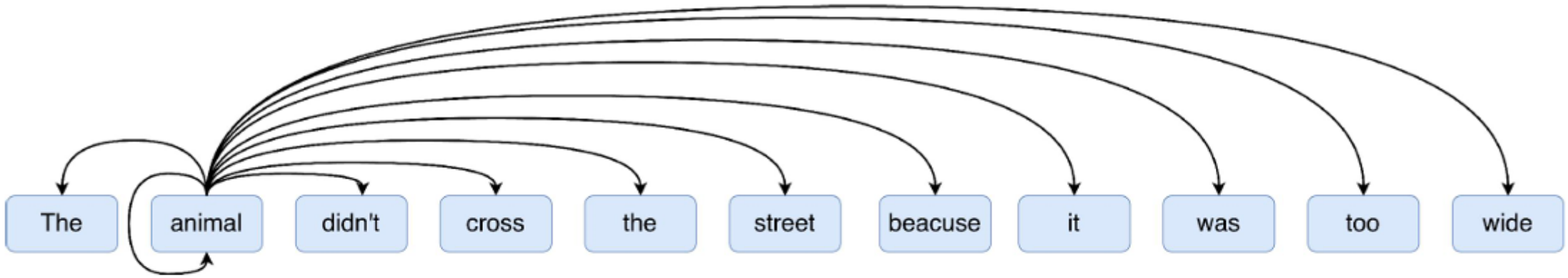
The	animal	didn’t	cross	the	street	because	it	was	too	wide

Self Attention

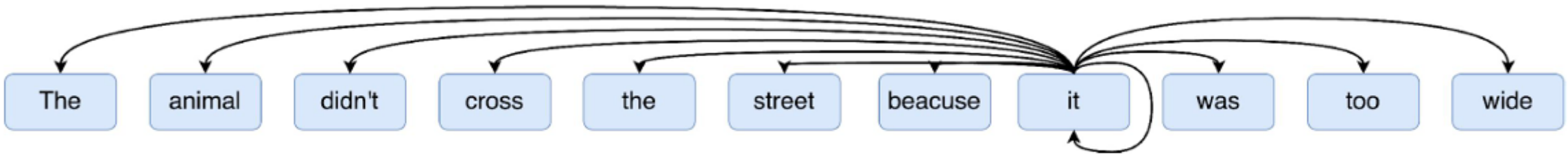


coreference resolution?

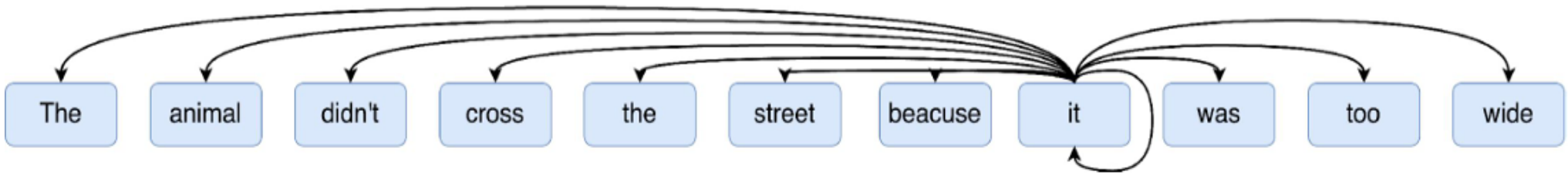
Self Attention



Self Attention



Self Attention

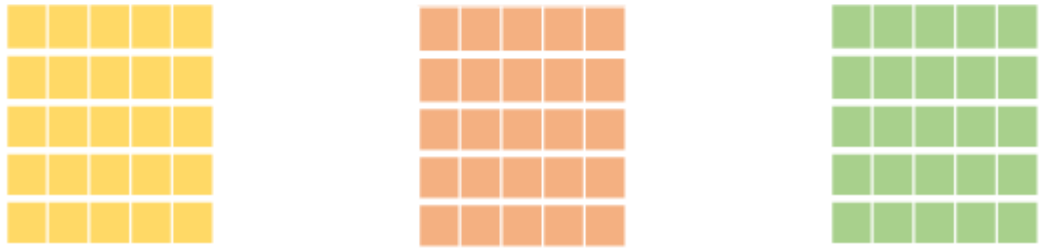


SELF

Query Inputs = Key Inputs = Value Inputs

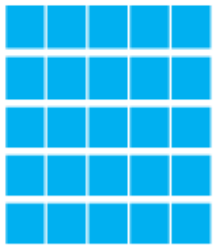
Self Attention

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

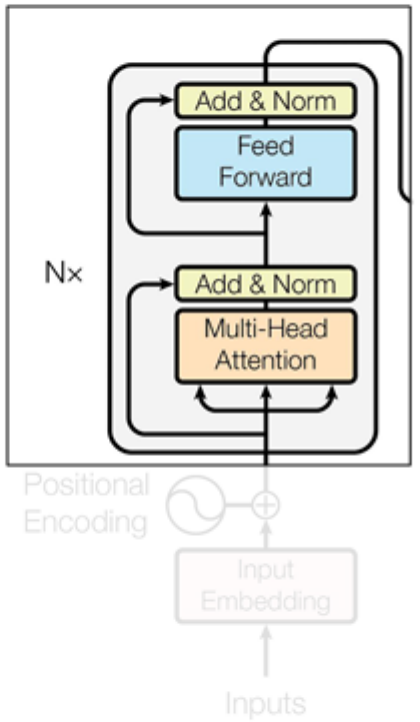


W_Q W_K W_V

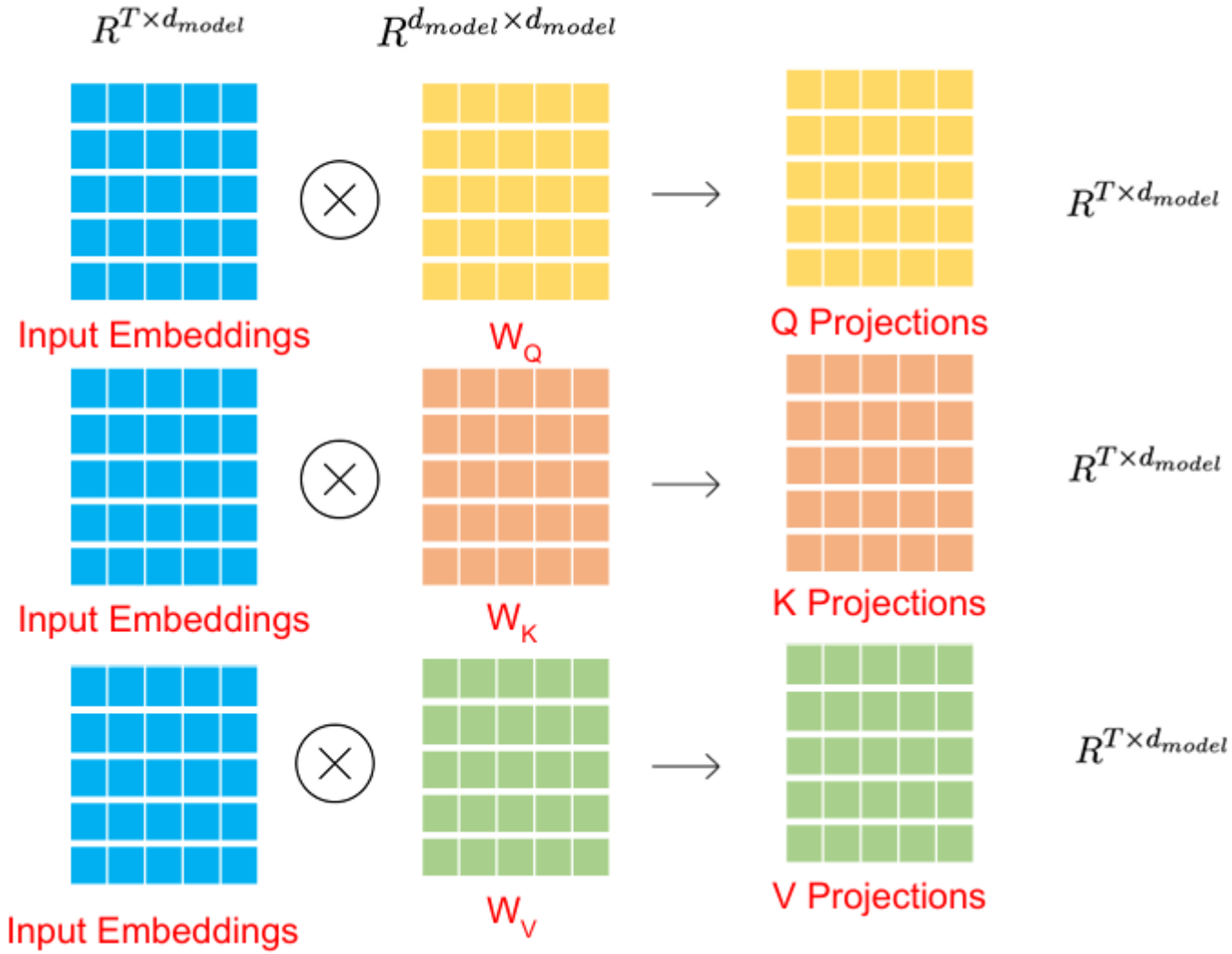
$R^{T \times d_{model}}$

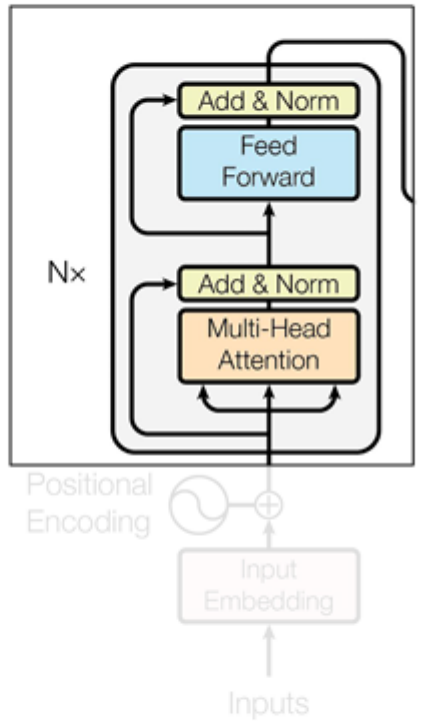


Input Embeddings



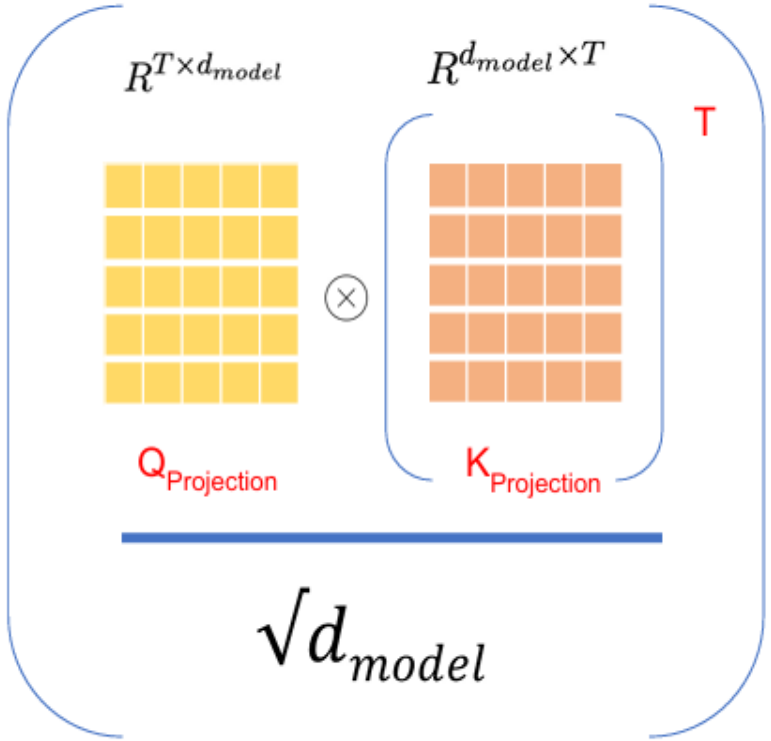
Self Attention



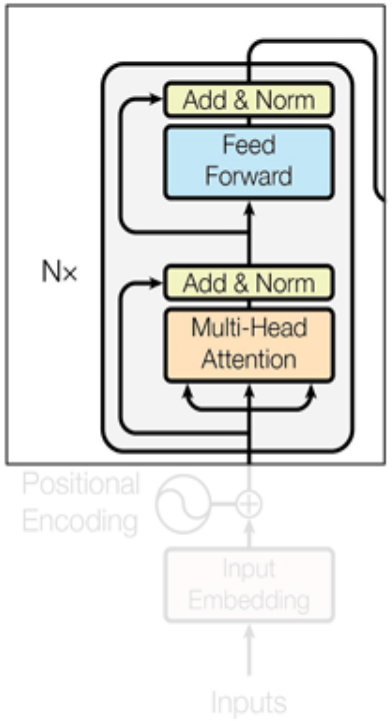


Self Attention

softmax



$R^{T \times T}$



Self Attention

$O(T^2 \times d_{model})$

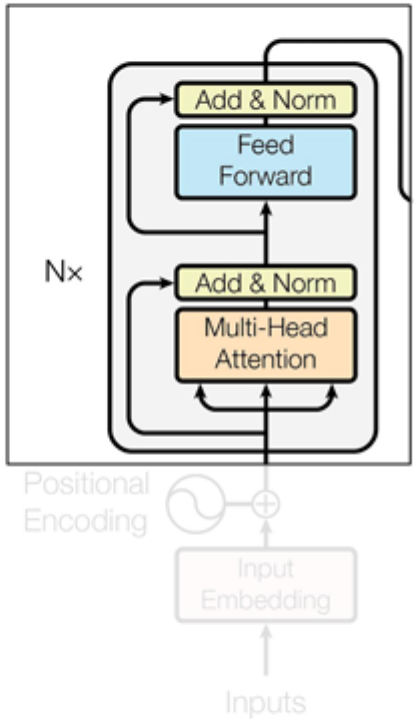
softmax

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

$Q_{Projection}$ $\quad$ $K_{Projection}$

$$\frac{}{\sqrt{d_{model}}}$$

$R^{T \times T}$



Self Attention

$O(T^2 \times d_{model})$

softmax

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

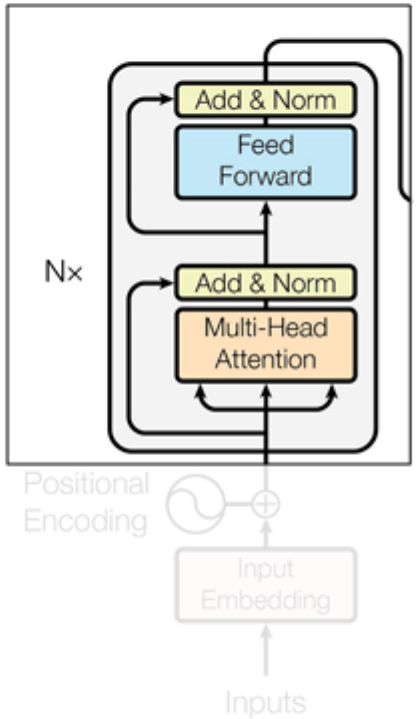
$Q_{Projection}$ $K_{Projection}$

$\sqrt{d_{model}}$

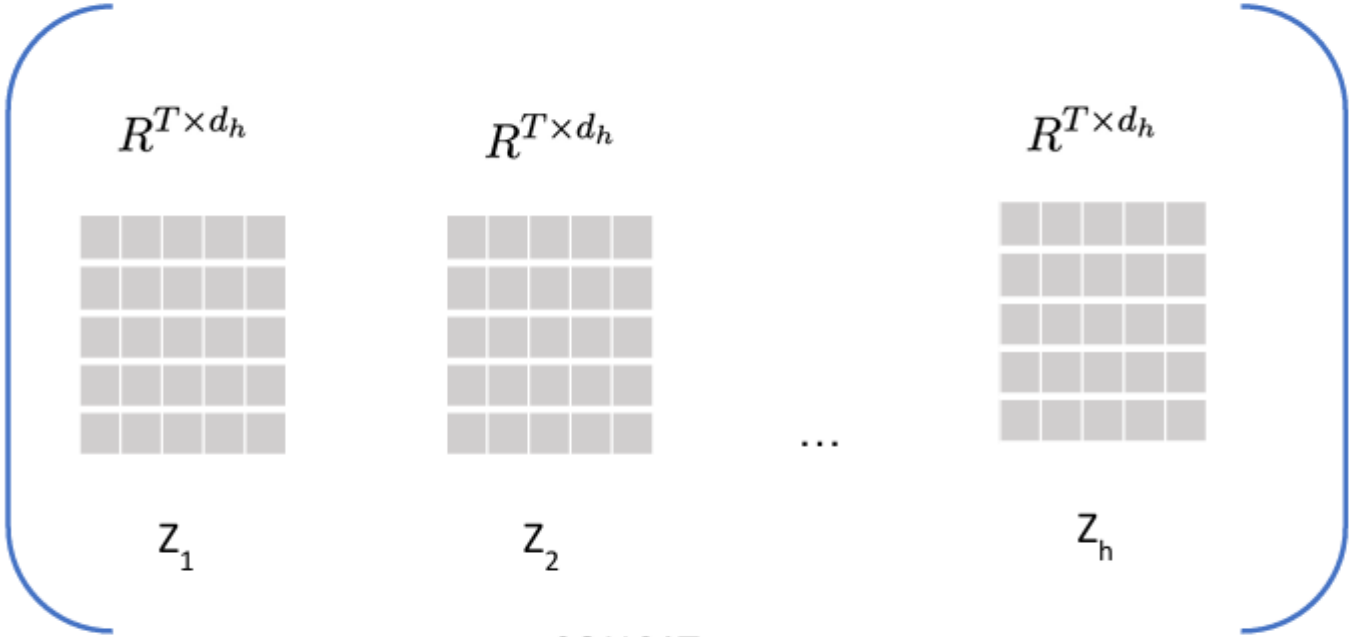
$R^{T \times T}$

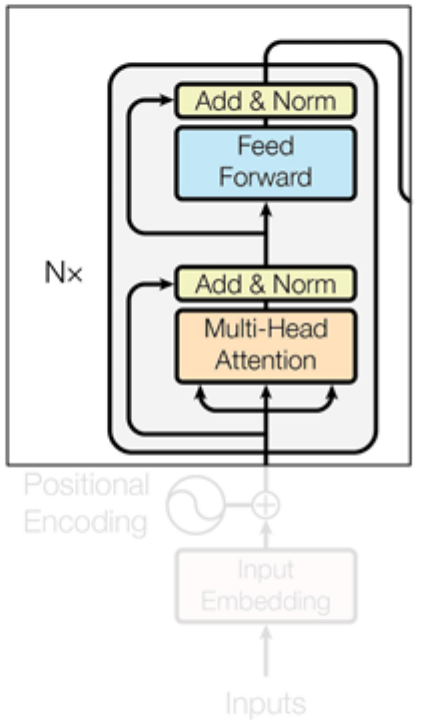
$R^{T \times d_{model}}$

$V_{Projection}$



Multi-Head Attention



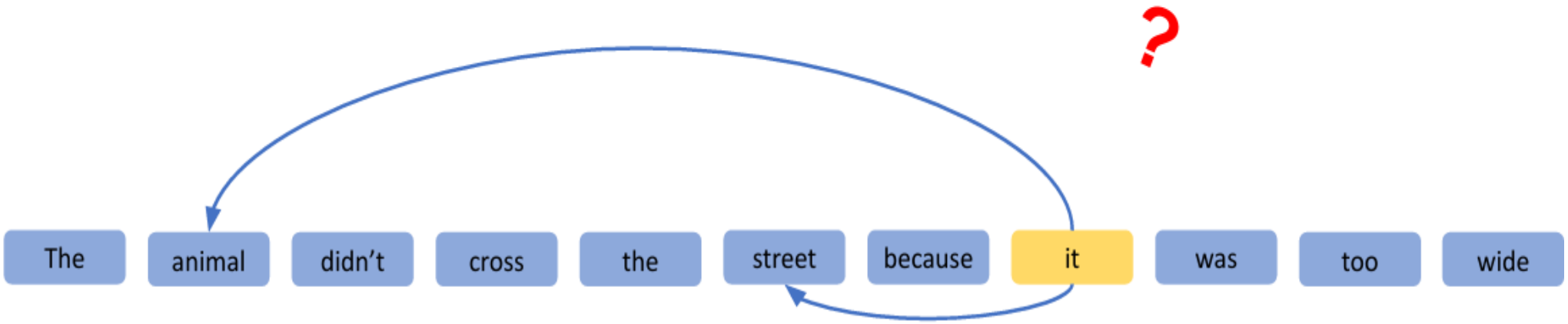


Multi Head Attention : Z

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

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

Self Attention



Sentence boundaries ?

Coreference resolution ✓

Context ?

Semantic relationships ?

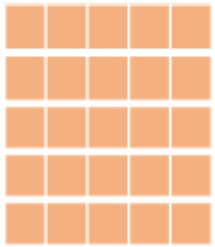
Part of Speech ?

Comparisons ?

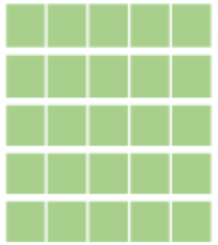
Self Attention

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

W_Q

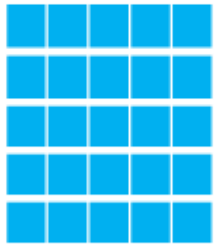


W_K

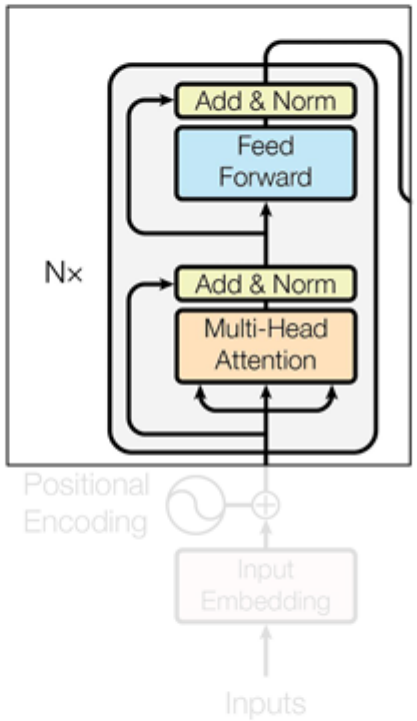


W_v

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

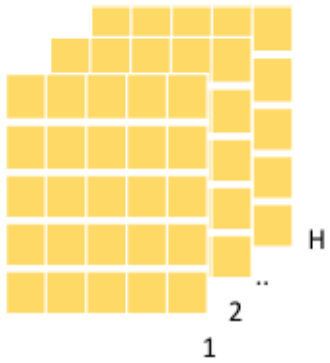


Input Embeddings

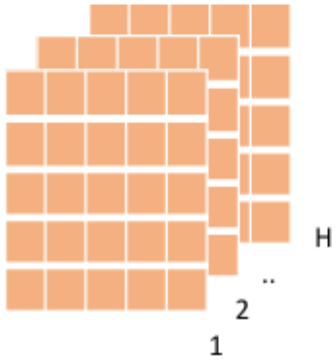


Multi-Head Attention

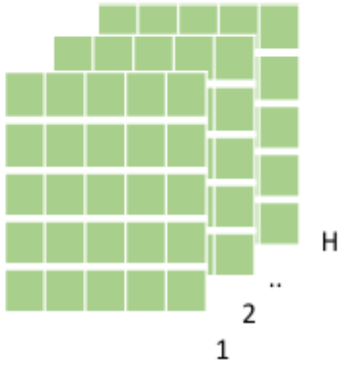
$R^{d_{model} \times d_h}$



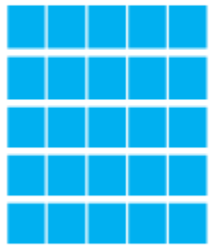
$W_{Q1}, W_{Q2}, \ldots W_{QH},$



$W_{K1}, W_{K2}, \ldots W_{KH},$



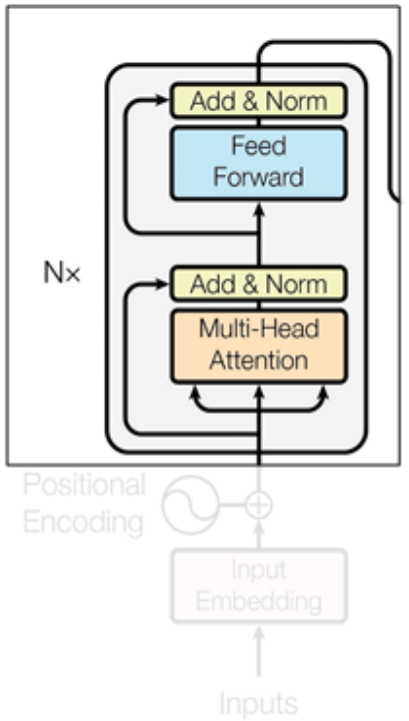
$W_{V1}, W_{V2}, \ldots W_{VH},$



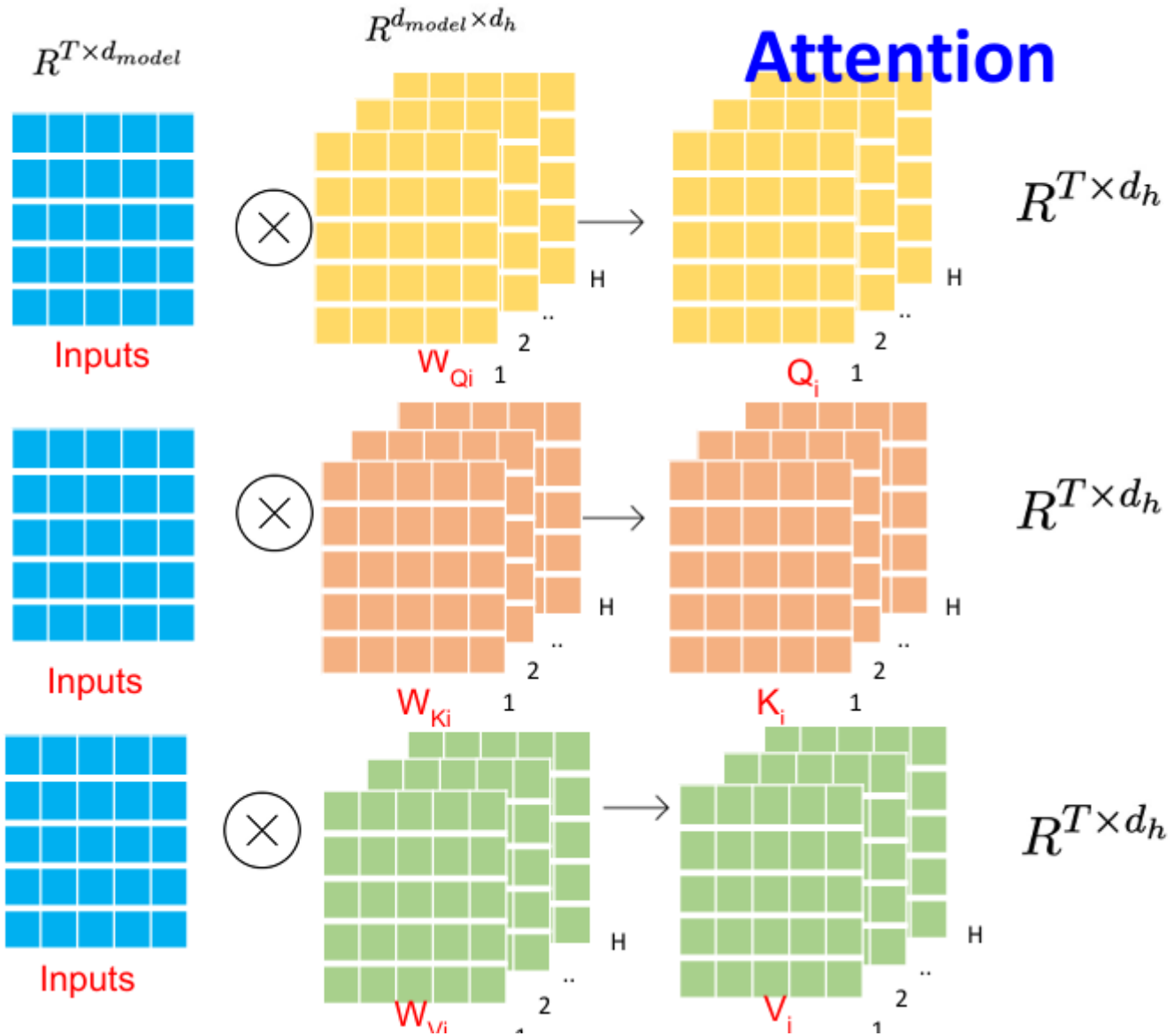
Input Embeddings

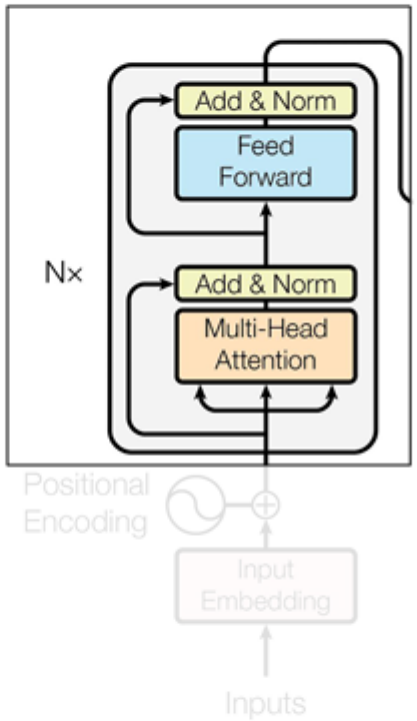
$R^{T \times d_{model}}$

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

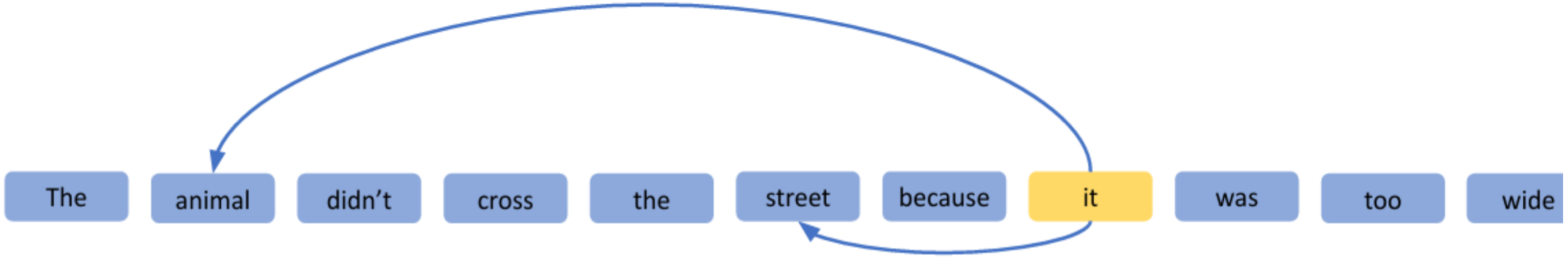


Multi-Head Attention





Multi-Head Attention



Sentence boundaries ✓

Coreference resolution ✓

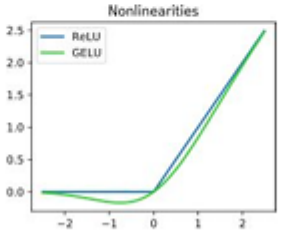
Context ✓

Semantic relationships ✓

Part of speech ✓

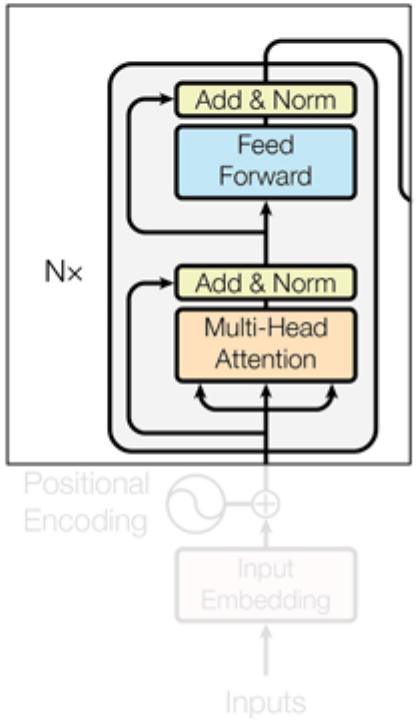
Comparisons ✓

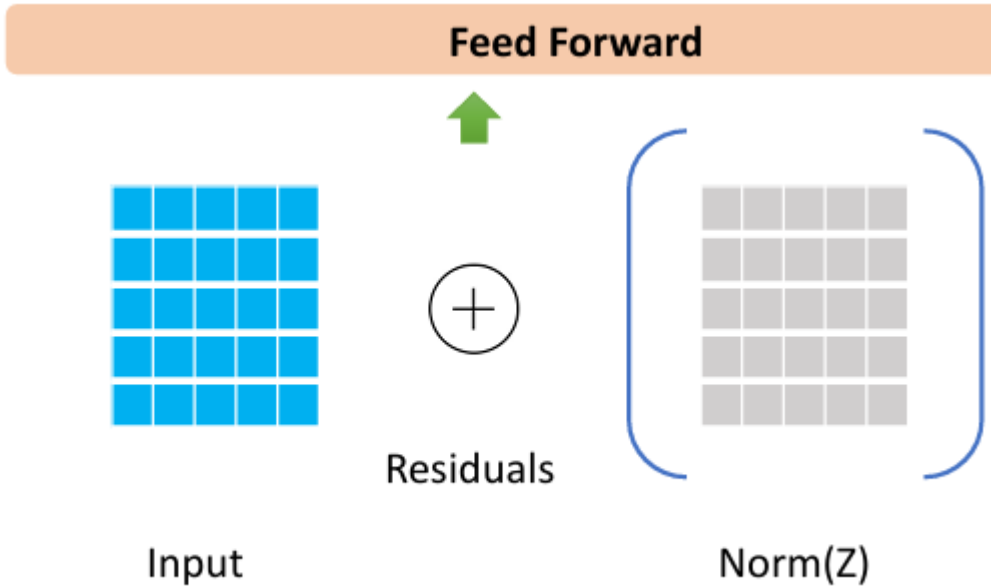
Feed Forward



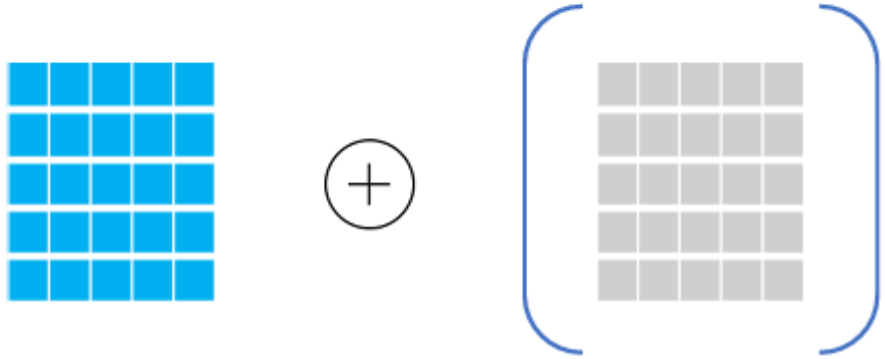
Feed Forward

- Non Linearity
- Complex Relationships
- Learn from each other





Add & Norm



Input Norm(Z)

Normalization

Mean 0, Std dev 1

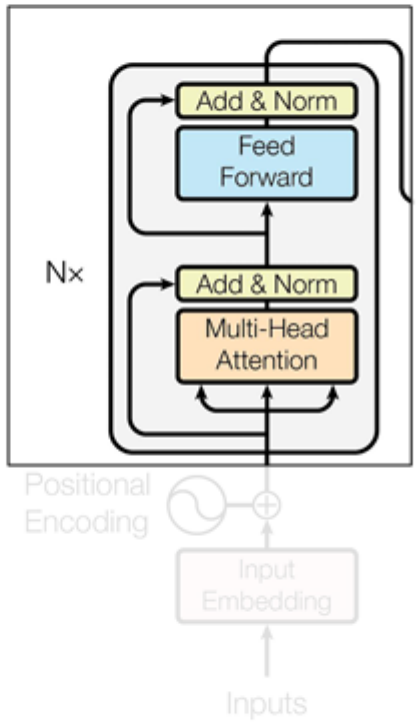
Stabilizes training

Regularization effect

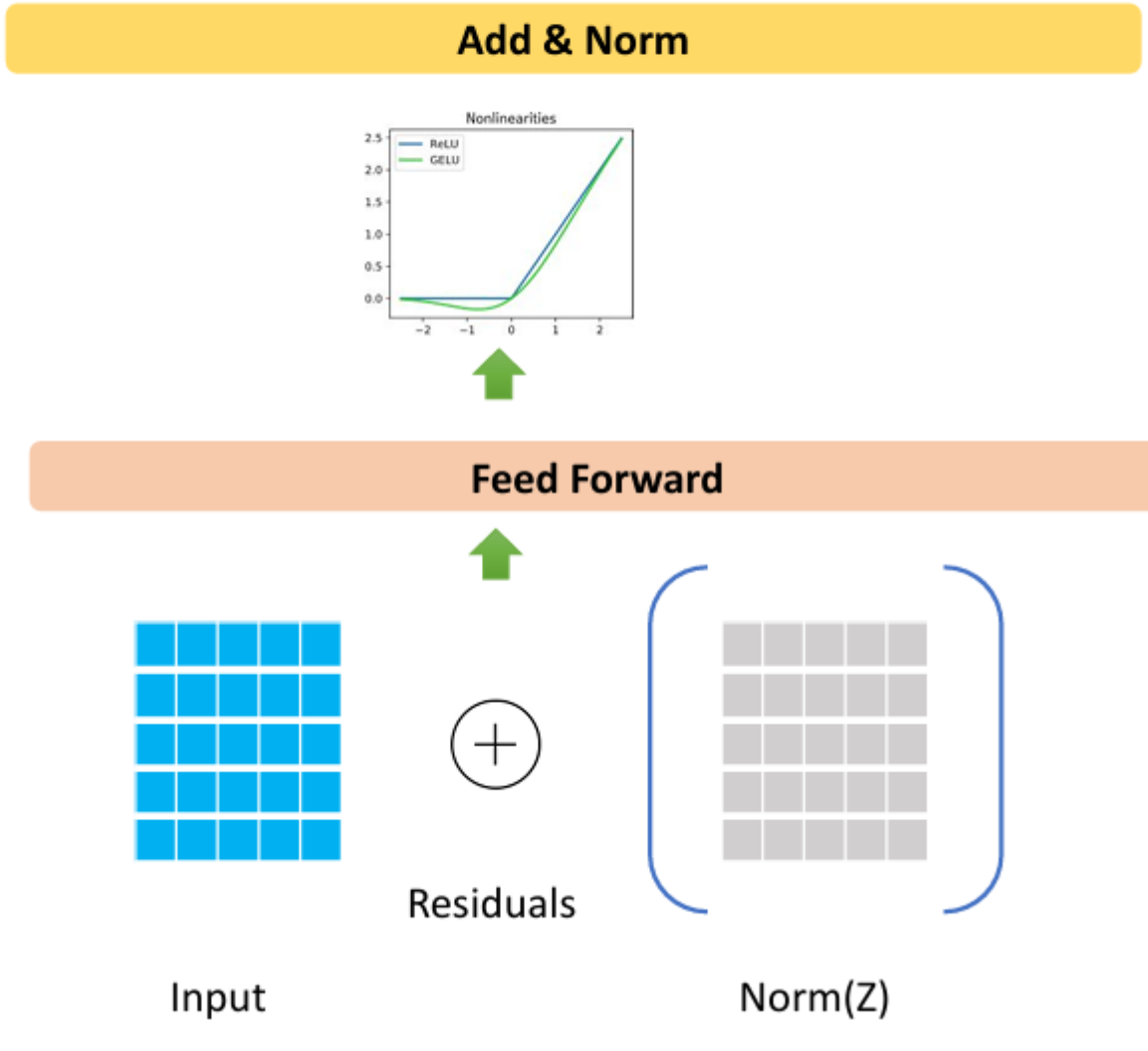
Add Residuals

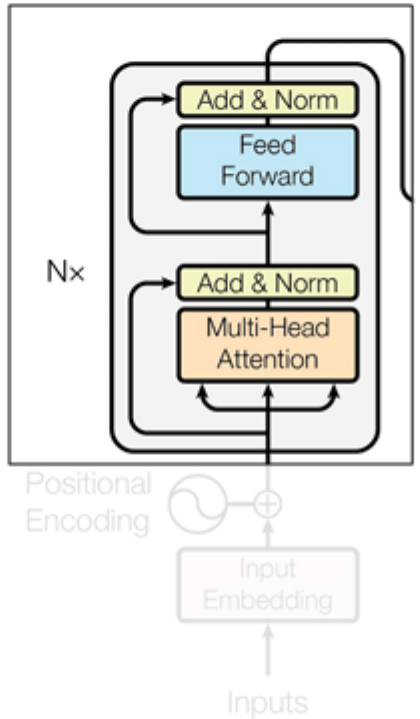
Avoid vanishing gradients

Train deeper networks



Add & Norm

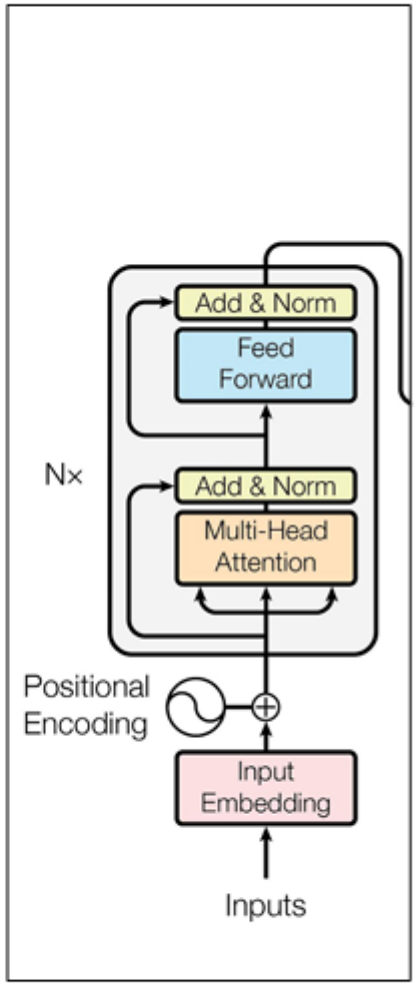

Add & Norm
Nonlinearities
ReLU
GELU
Feed Forward
Residuals
Input
Norm(Z)


Add & Norm
Feed Forward
Nx
Add & Norm
Multi-Head Attention
Positional Encoding
Input Embedding
Inputs

Encoders

Encoder

ENCODER



Encoders

Encoder

ENCODER

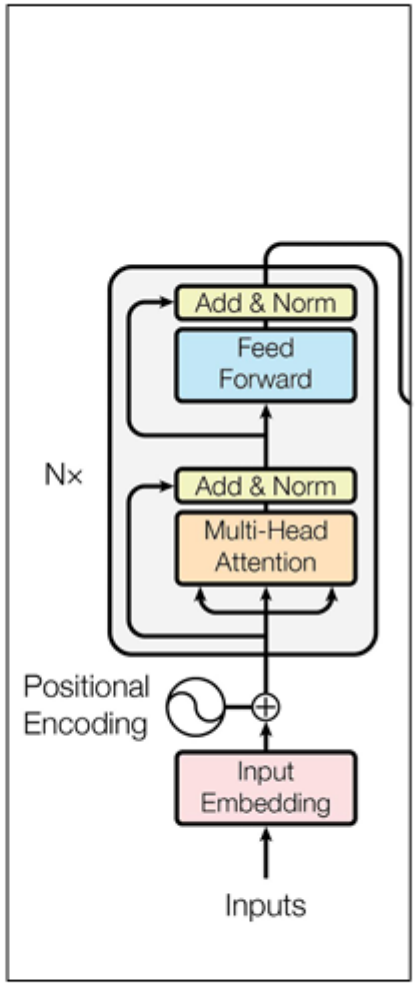
.
.
.

ENCODER

ENCODER

Input to Encoder$_{i+1}$

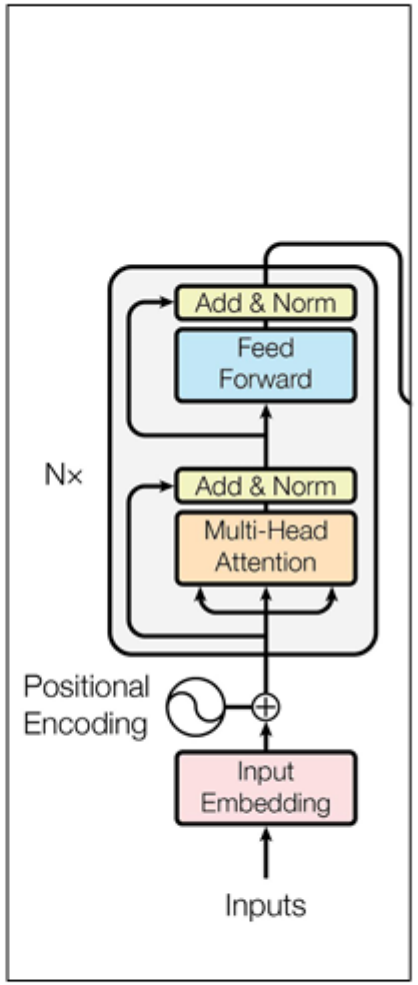
Output from Encoder$_{i}$



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



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

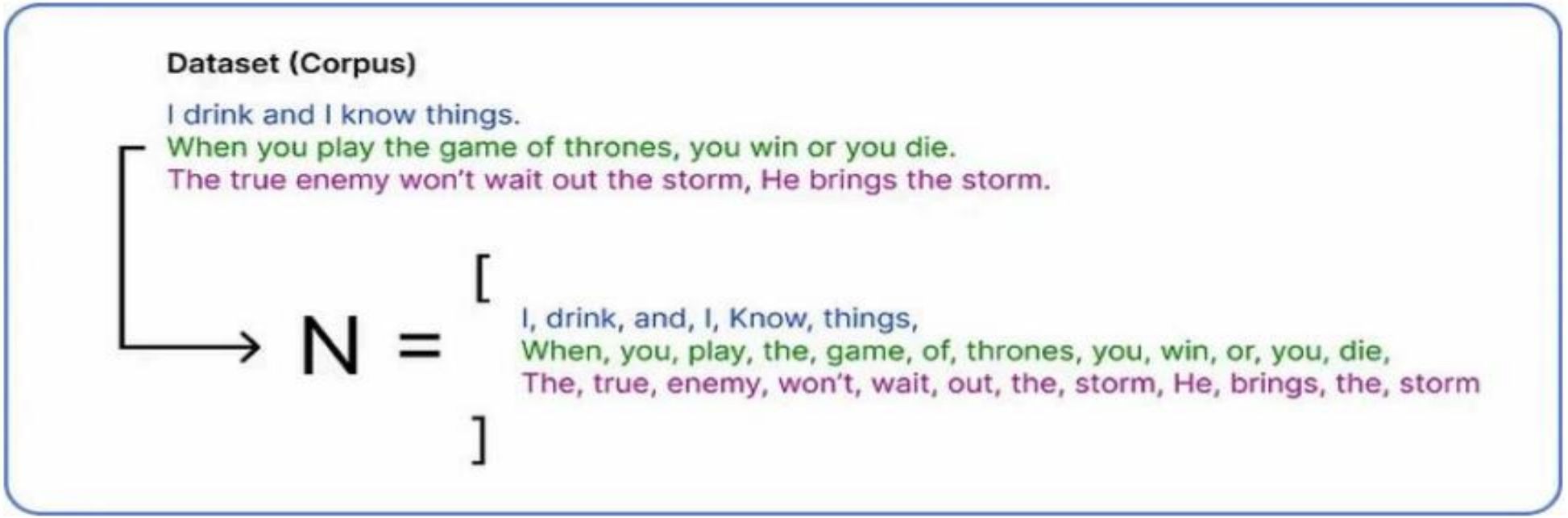
Step 2— Finding Vocab Size

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.

$$vocab\ size = count(set(N))$$

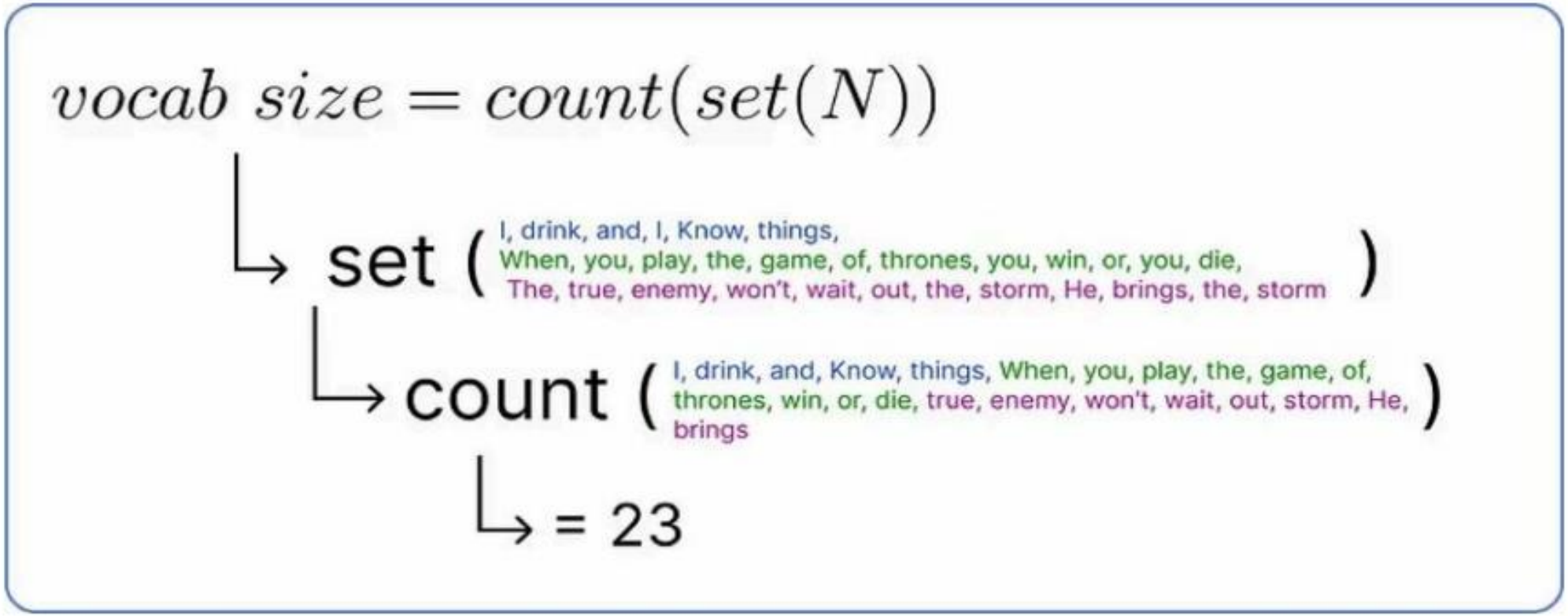
vocab_size formula where N is total number of words

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



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.



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.

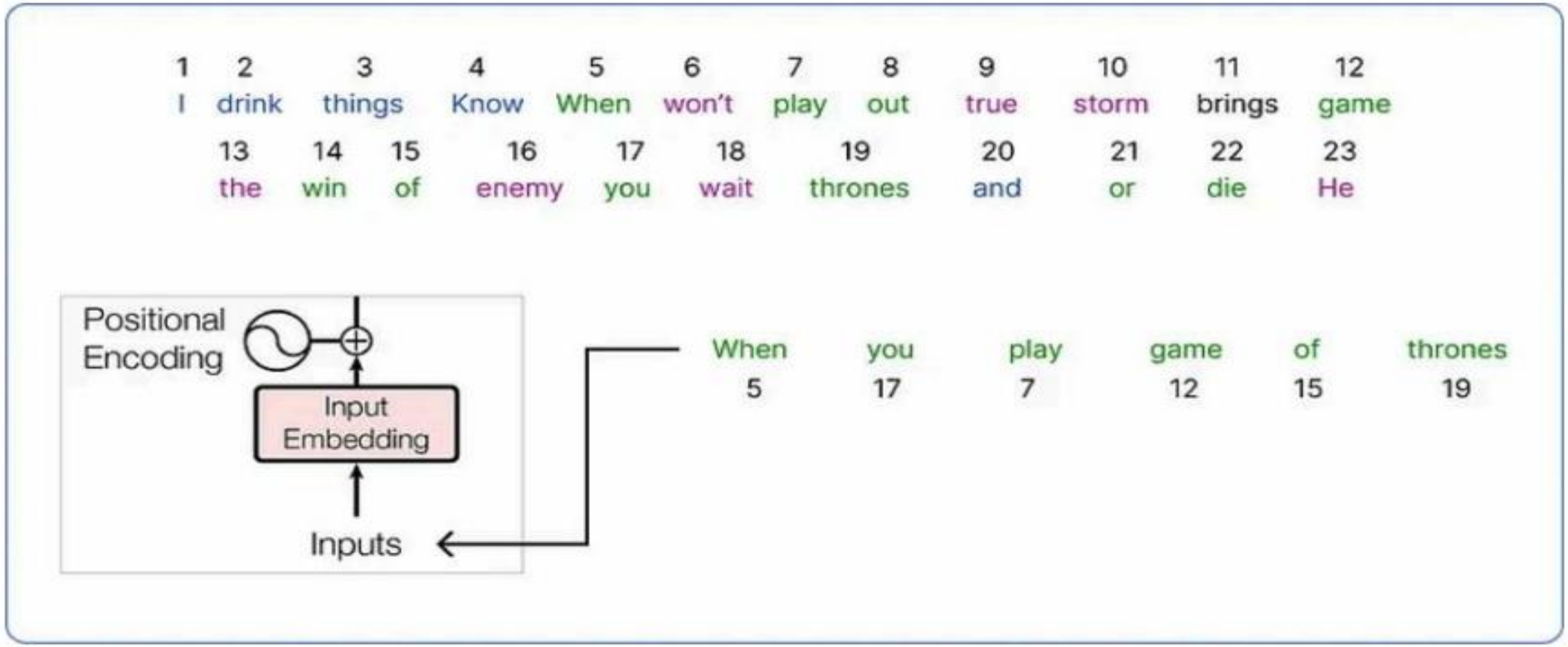
1	2	3	4	5	6	7	8	9	10	11	12
I	drink	things	Know	When	won't	play	out	true	storm	brings	game

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

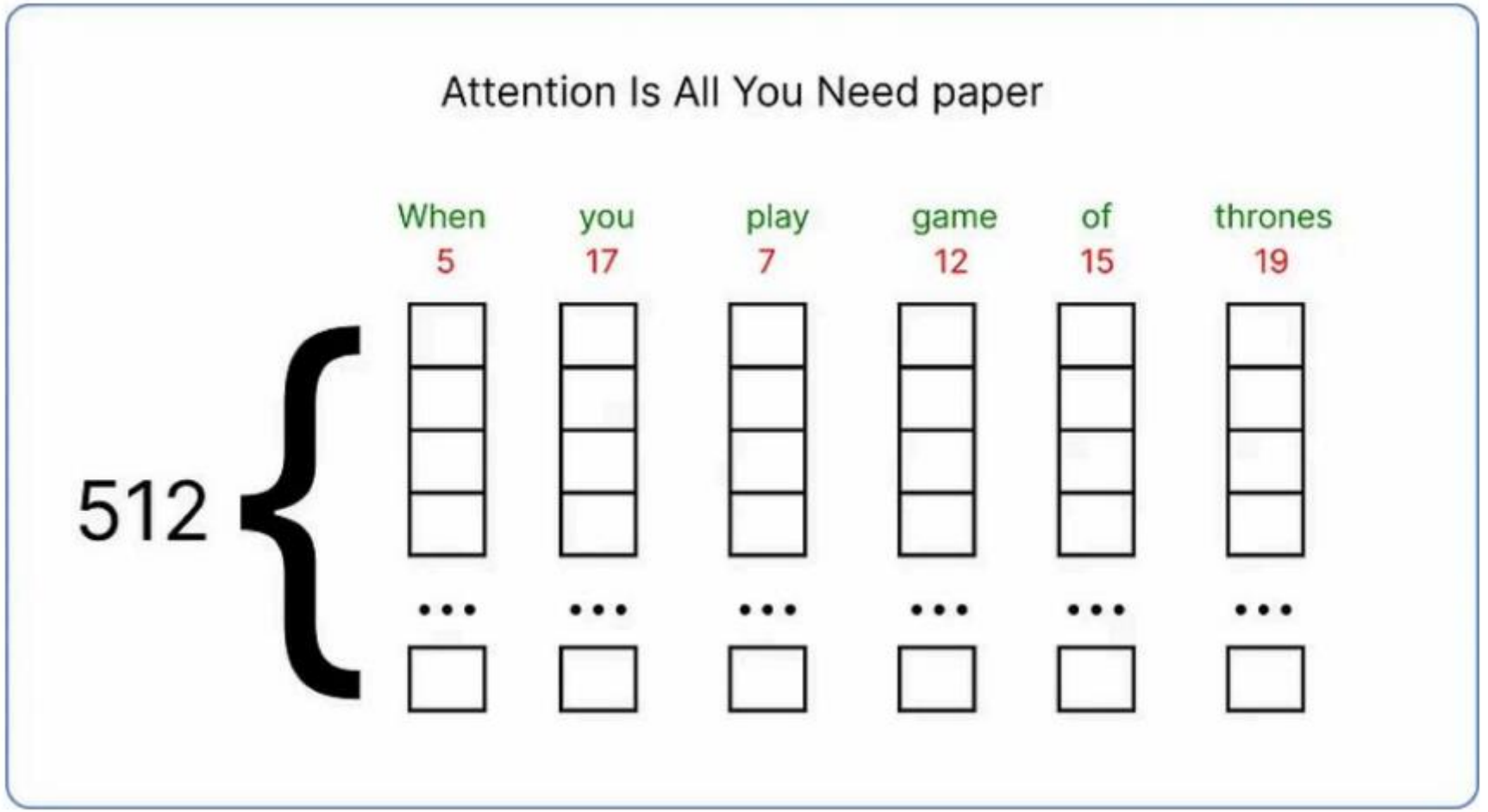
Step 4 — Calculating Embedding

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



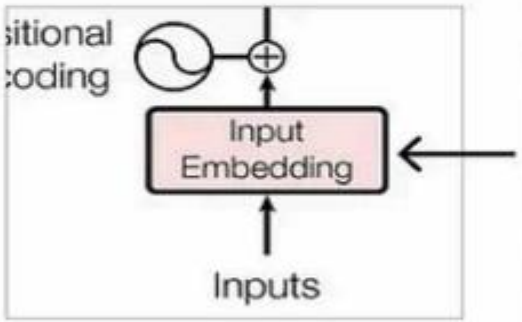
Input sentence for transformer

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.



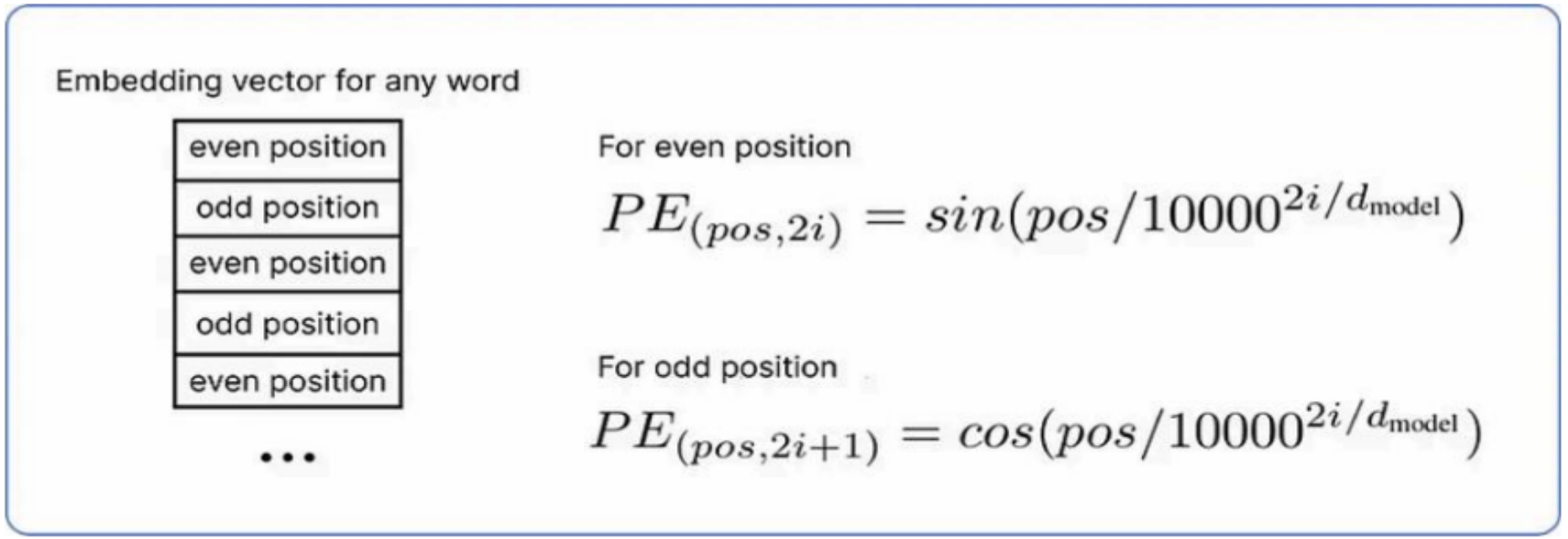
When	you	play	game	of	thrones
5	17	7	12	15	19

e1	e2	e3	e4	e5	e6
0.79	0.38	0.01	0.12	0.88	0.6
0.6	0.12	0.51	0.6	0.41	0.33
0.96	0.06	0.27	0.65	0.79	0.75
0.64	0.79	0.31	0.22	0.62	0.48
0.97	0.9	0.56	0.07	0.5	0.94
0.2	0.74	0.59	0.37	0.7	0.21

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 ith value of that embedding vector for each word.



Positional Embedding formula

When
5

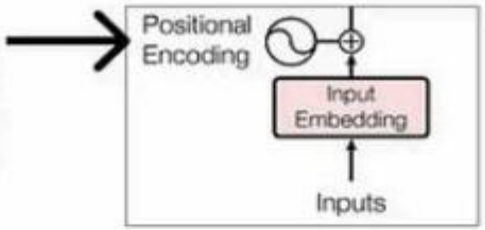
i	e1	Position	Formula	p1
0	0.79	Even	$sin(0/10000^{(2*0/6)})$	0
1	0.6	Odd	$cos(0/10000^{(2*1/6)})$	1
2	0.96	Even	$sin(0/10000^{(2*2/6)})$	0
3	0.64	Odd	$cos(0/10000^{(2*3/6)})$	1
4	0.97	Even	$sin(0/10000^{(2*4/6)})$	0
5	0.2	Odd	$cos(0/10000^{(2*5/6)})$	1
d (dim)	6			
POS	0			

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.

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

Position Embedding Matrix

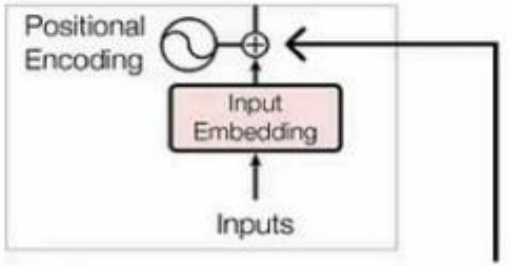
p1	p2	p3	p4	p5	p6
0	0.8415	0.9093	0.1411	-0.7568	-0.9589
1	0.0464	0.9957	0.1388	0.1846	0.9732
0	0.0022	0.0043	0.0065	0.0086	0.0108
1	0.0001	1	0.0003	0.0004	1
0	0	0	0	0	0
1	0	1	0	0	1

+

Word Embedding Matrix

e1	e2	e3	e4	e5	e6
0.79	0.38	0.01	0.12	0.88	0.6
0.6	0.12	0.51	0.6	0.41	0.33
0.96	0.06	0.27	0.65	0.79	0.75
0.64	0.79	0.31	0.22	0.62	0.48
0.97	0.9	0.56	0.07	0.5	0.94
0.2	0.74	0.59	0.37	0.7	0.21

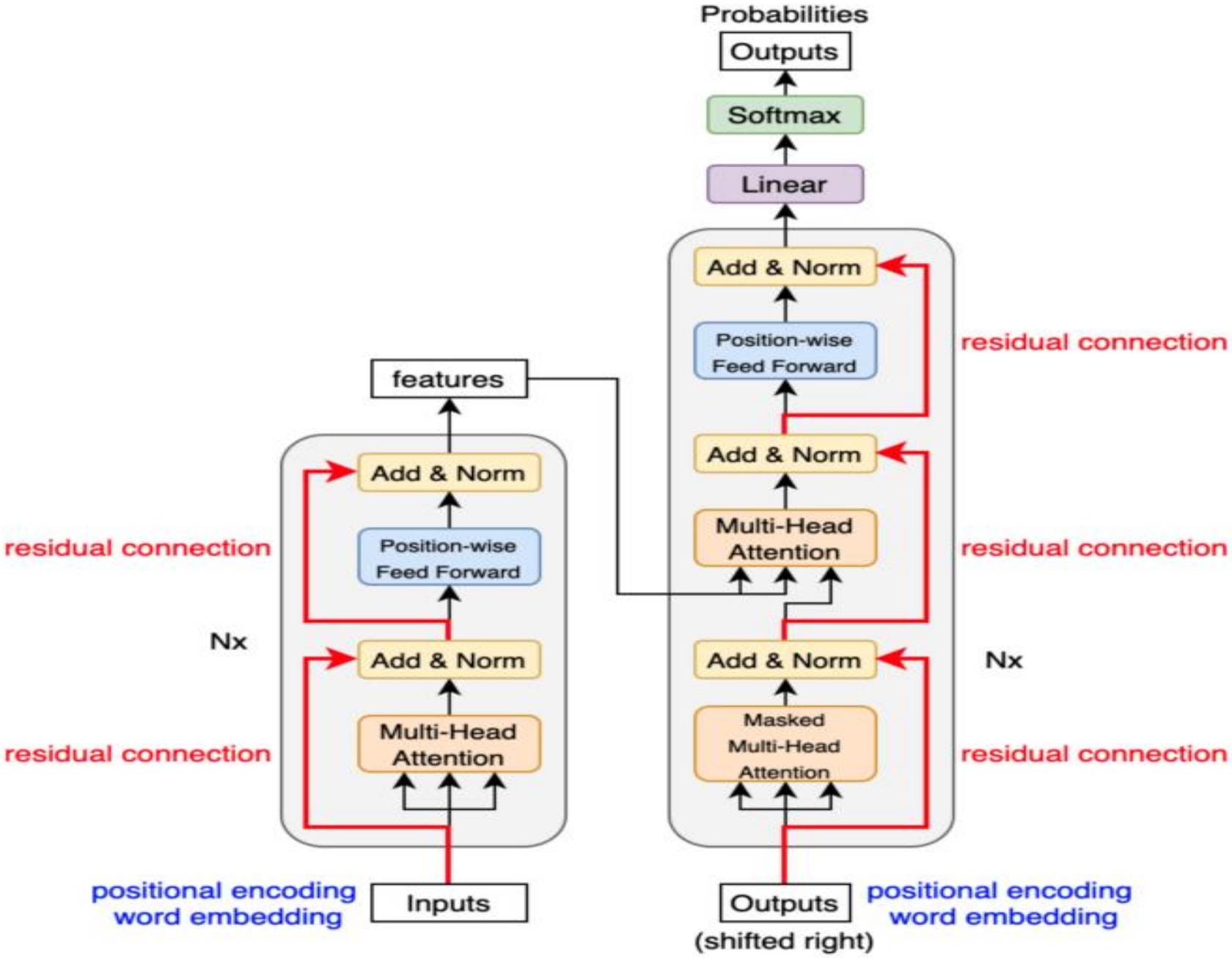
=



ep1	ep2	ep3	ep4	ep5	ep6
0.79	1.22	0.92	0.26	0.12	-0.36
1.6	0.17	1.51	0.74	0.59	1.3
0.96	0.06	0.27	0.66	0.8	0.76
1.64	0.79	1.31	0.22	0.62	1.48
0.97	0.9	0.56	0.07	0.5	0.94
1.2	0.74	1.59	0.37	0.7	1.21

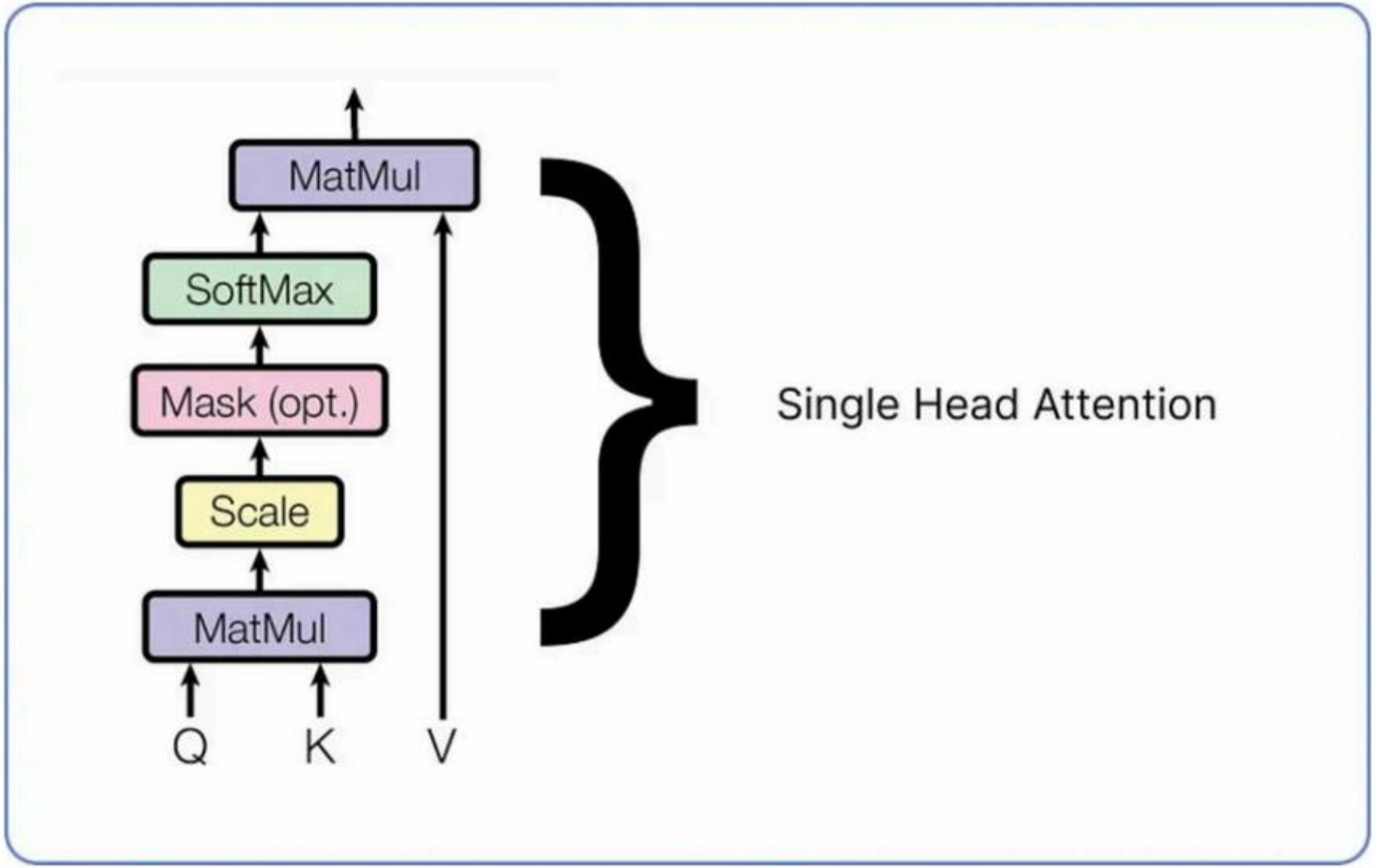
concatenation step

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


Probabilities
Outputs
Softmax
Linear
Add & Norm
Position-wise Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
residual connection
residual connection
residual connection
Nx
Nx
features
Add & Norm
Position-wise Feed Forward
Add & Norm
Multi-Head Attention
residual connection
residual connection
residual connection
positional encoding
word embedding
Inputs
Outputs
(shifted right)
positional encoding
word embedding

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

X

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

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

X

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

X

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

Calculating Key and Value Matrices

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

Query, Key, Value matrices

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

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

X

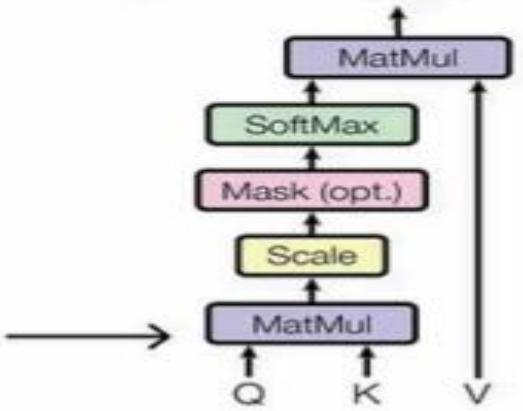
Transpose (Key)

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

4 x 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



matrix multiplication between Query and Key

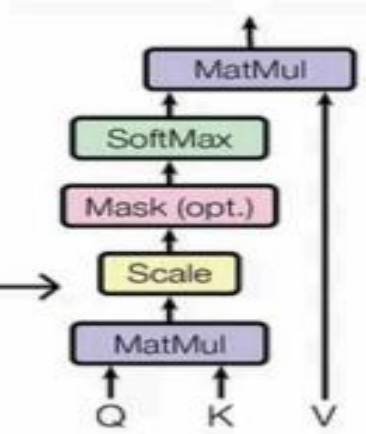
For scaling the resultant matrix, we have to reuse the dimension of our embedding vector, which is 6 .

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

$\sqrt{d_k}$ where d (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

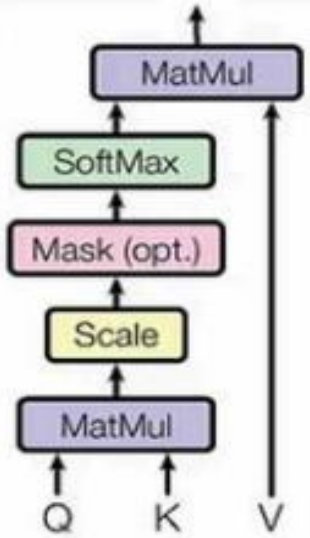
SoftMax

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

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

=

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

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

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

X

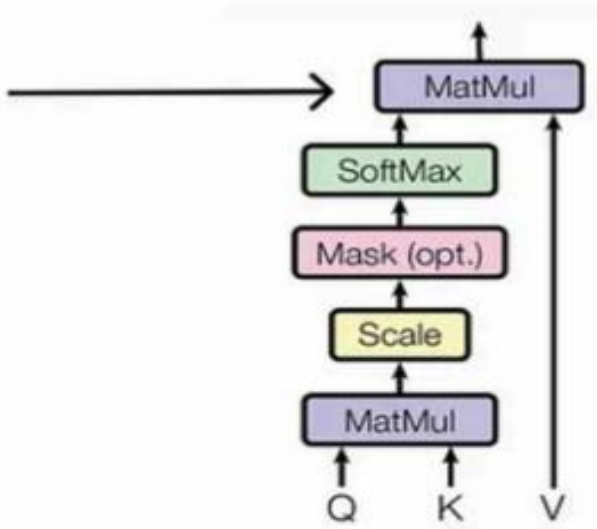
Value

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

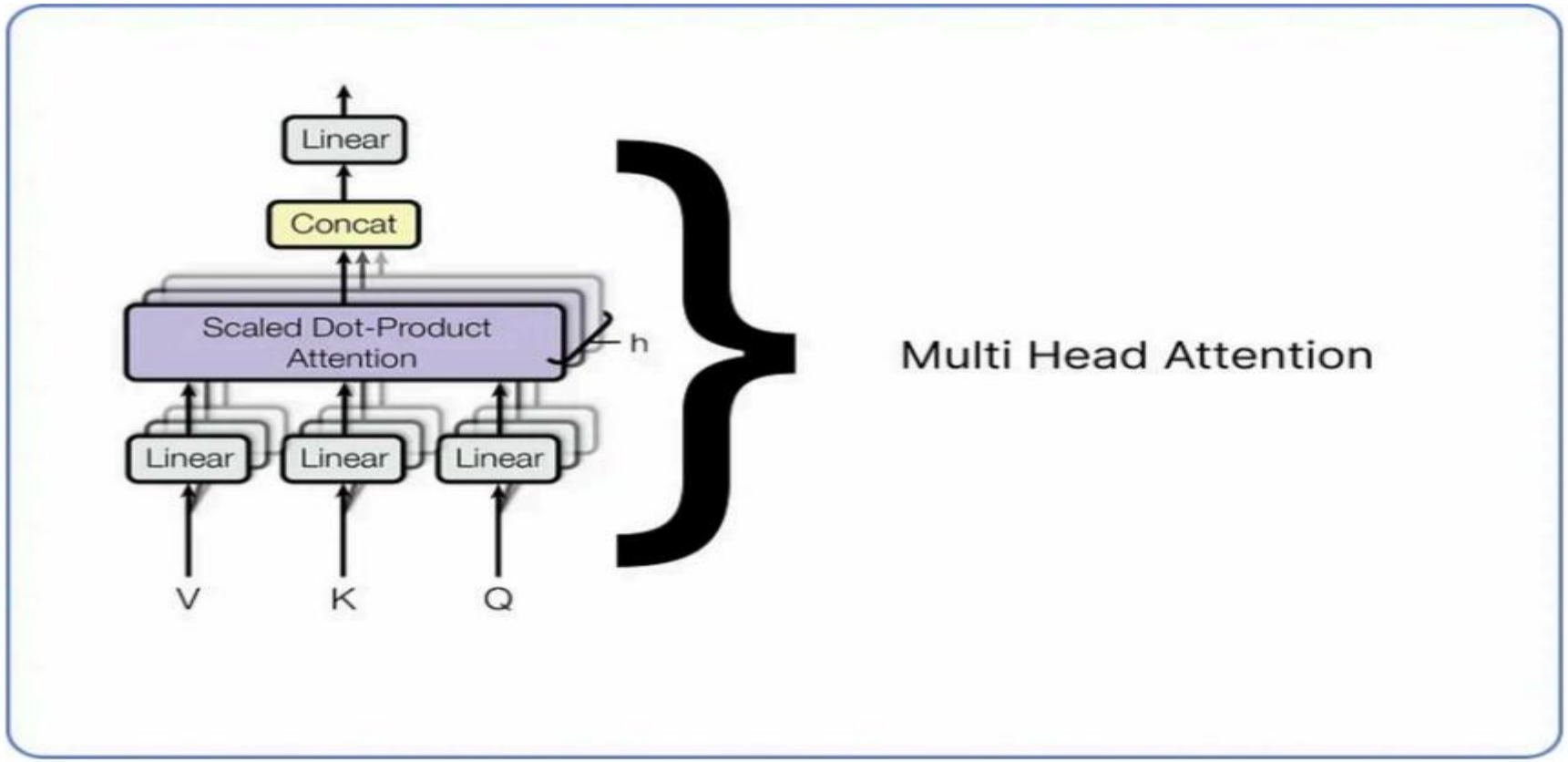
=

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:



Multi Head attention in Transformer

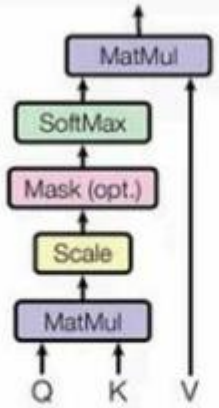
Each single-head attention has three inputs: **query**, **key**, and **value**, and each three have a different set of weights. Once all single-head attentions output their resultant matrices, they will all be concatenated, and the final concatenated matrix is once again transformed linearly by multiplying 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.

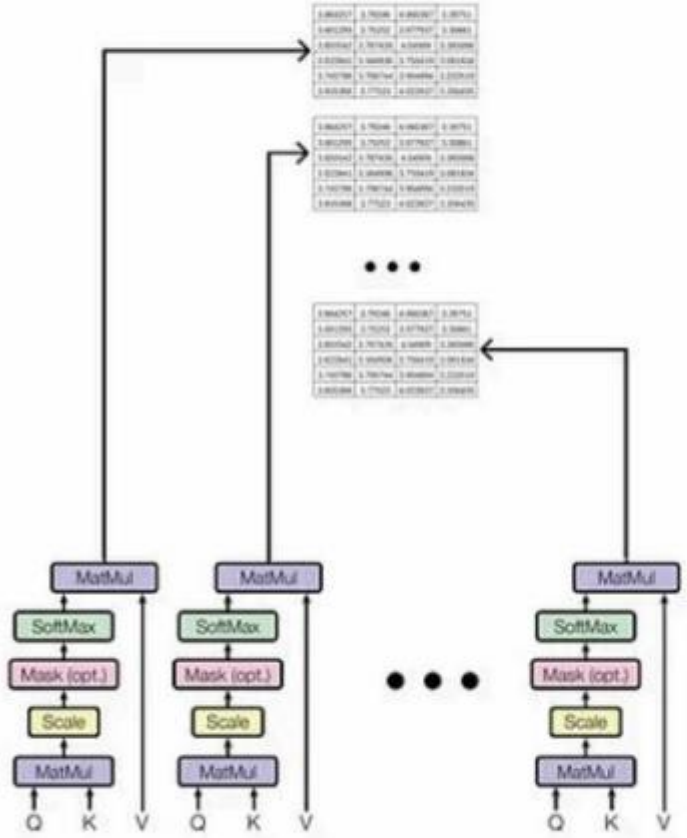
Kmit

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

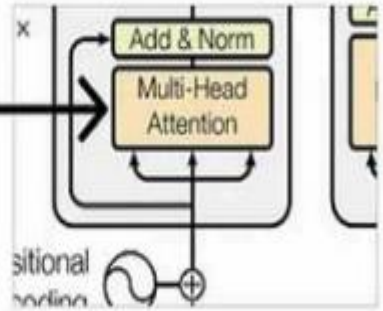


Single Head attention vs Multi Head attention

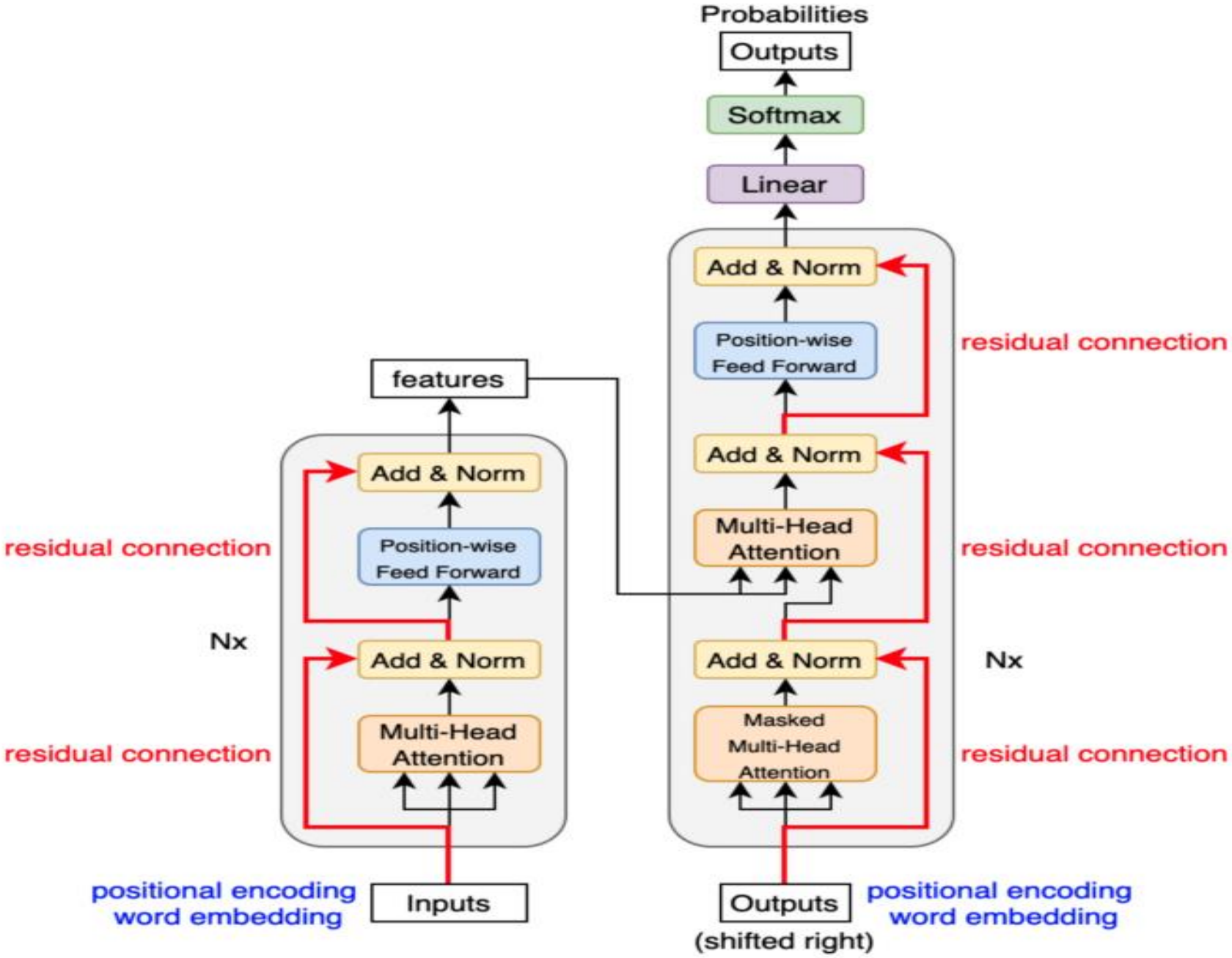
Output of Multi Head attention

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


Probabilities
Outputs
Softmax
Linear
Add & Norm
Position-wise Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
residual connection
residual connection
residual connection
Nx
Nx
features
Add & Norm
Position-wise Feed Forward
Add & Norm
Multi-Head Attention
residual connection
residual connection
residual connection
positional encoding
word embedding
Inputs
Outputs
(shifted right)
positional encoding
word embedding

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.

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

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

Row Wise Implementation

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

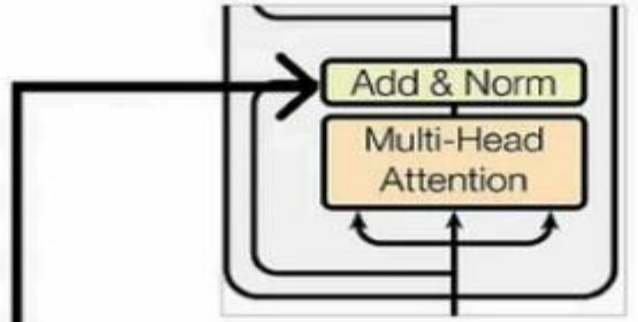
calculating meand and std.

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

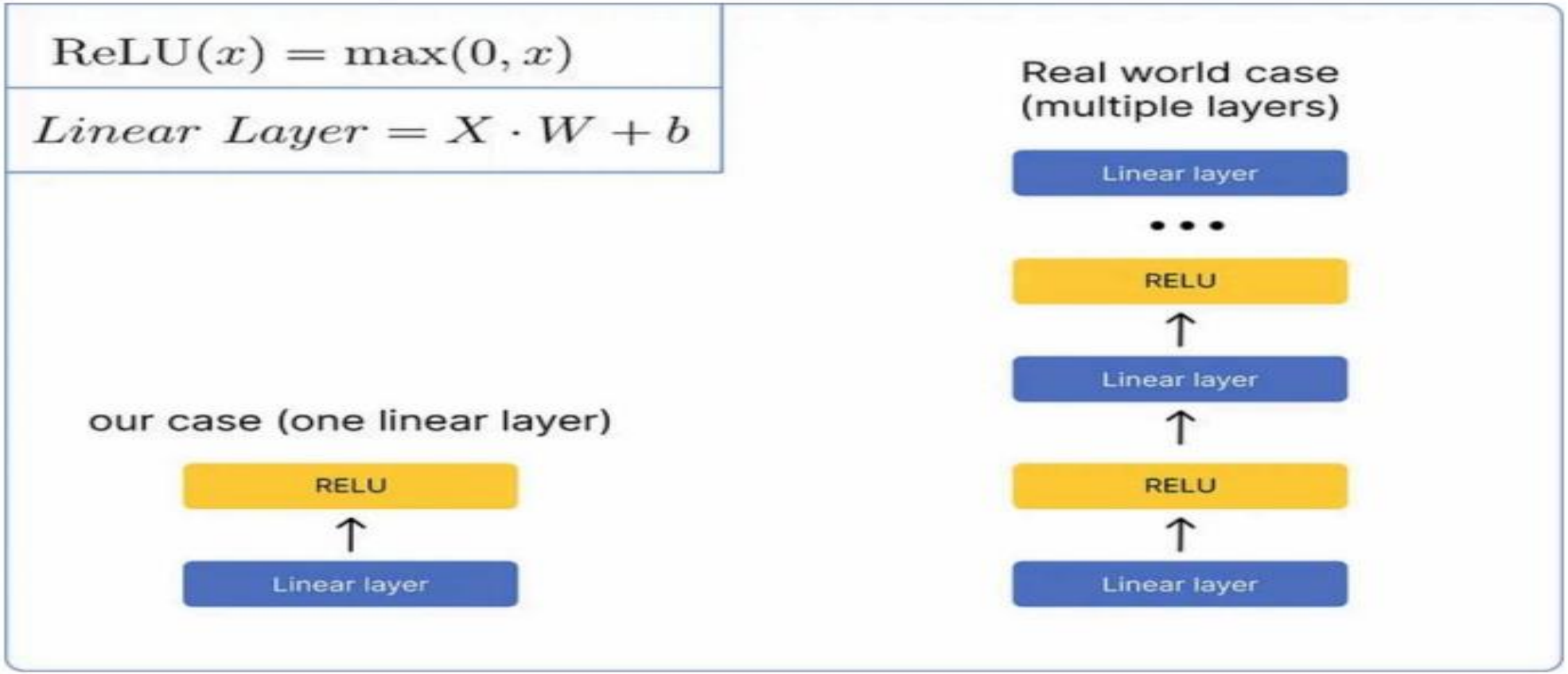
normalizing the resultant matrix

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:

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

$$Linear\ Layer = X \cdot W + b$$



Feed Forward network comparison

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 afrer 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.

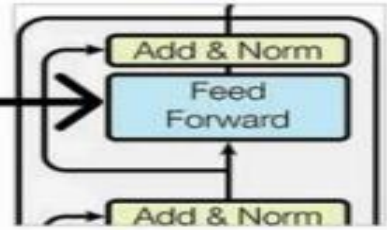
$$\text{ReLU}(x) = \max(0, x)$$

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 x 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



Calculating ReLU Layer

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.

Kmit

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

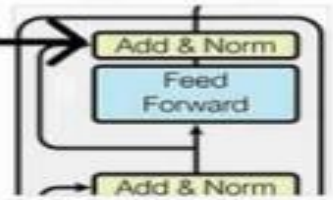
→

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

→



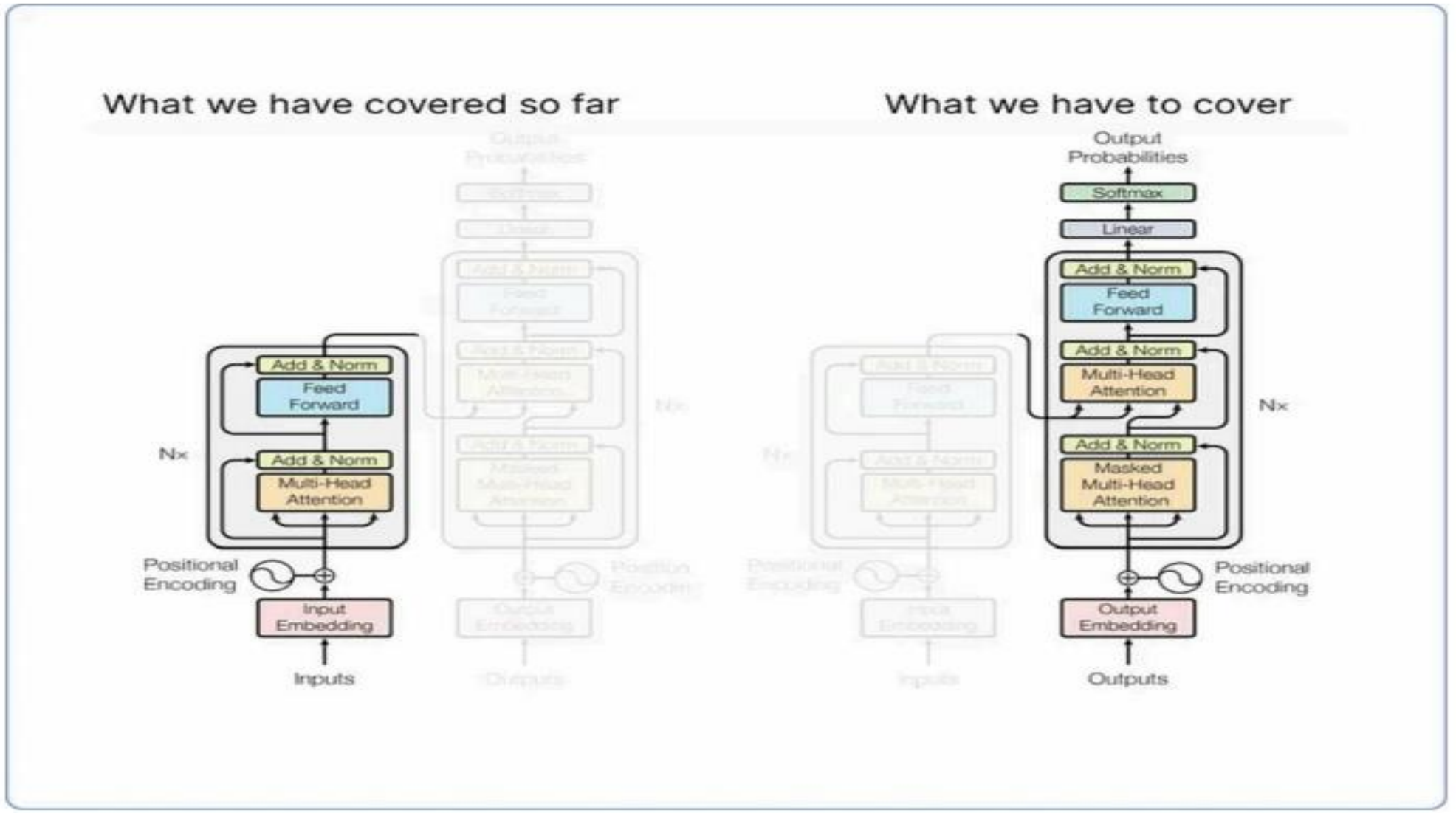
Add and Norm after Feed Forward Network

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



Upcoming steps illustration