



# C EURO<sup>2</sup>

Mastering Transformers: From Building Blocks to Real-World Applications

Transformers by Prof.Dr.Tuğba Taşkaya Temizel

12 Sept 2023

# Self Attention (Intra Attention) Networks

## Transformers

- The **Transformer** (Vaswani et al., 2017) is an architecture that uses attention for information flow: “Attention is all you need”
- It was originally designed for machine translation and has two parts:
  - an **encoder** that “summarizes” an input sentence
  - a **decoder** (a conditional LM) that generates an output, based on the input

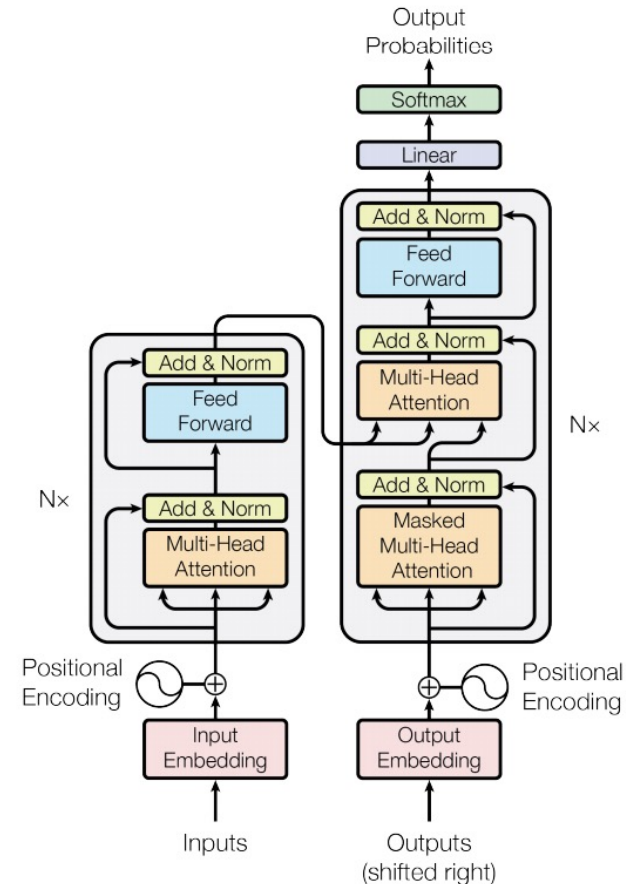


Figure 1: The Transformer - model architecture.

# Self Attention (Intra Attention) Networks

## Transformers

- It is an approach to sequence processing that eliminates recurrent connections and returns to architectures reminiscent of the fully connected networks.
- Transformers map sequences of input vectors  $(x_1, \dots, x_n)$  to sequences of output vectors  $(y_1, \dots, y_n)$  of the same length.



# Self Attention (Intra Attention) Networks

## Transformers

- Transformers are made up of stacks of network layers consisting of simple linear layers, feedforward networks, and custom connections around them.
- The key innovation of transformers is the use of **self-attention layers**.

# Vanilla Transformer: Embedding

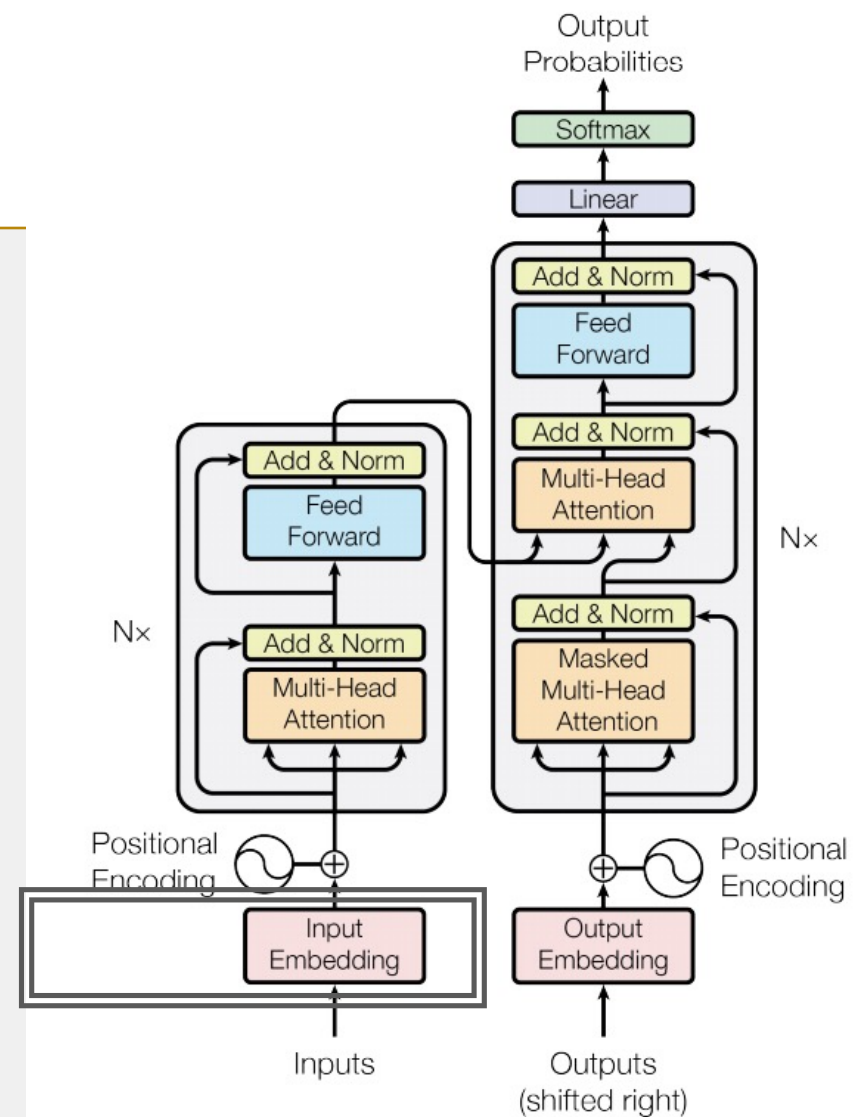


Figure 1: The Transformer - model architecture.

# Input and Output Word Embedding

- They use learned embeddings to convert the input tokens and output tokens to vectors of dimension  $d_{model}$ .
- They use the usual learned linear transformation and softmax function to convert the decoder output to predicted next-token probabilities.

# Vanilla Transformer: Positional Encoding

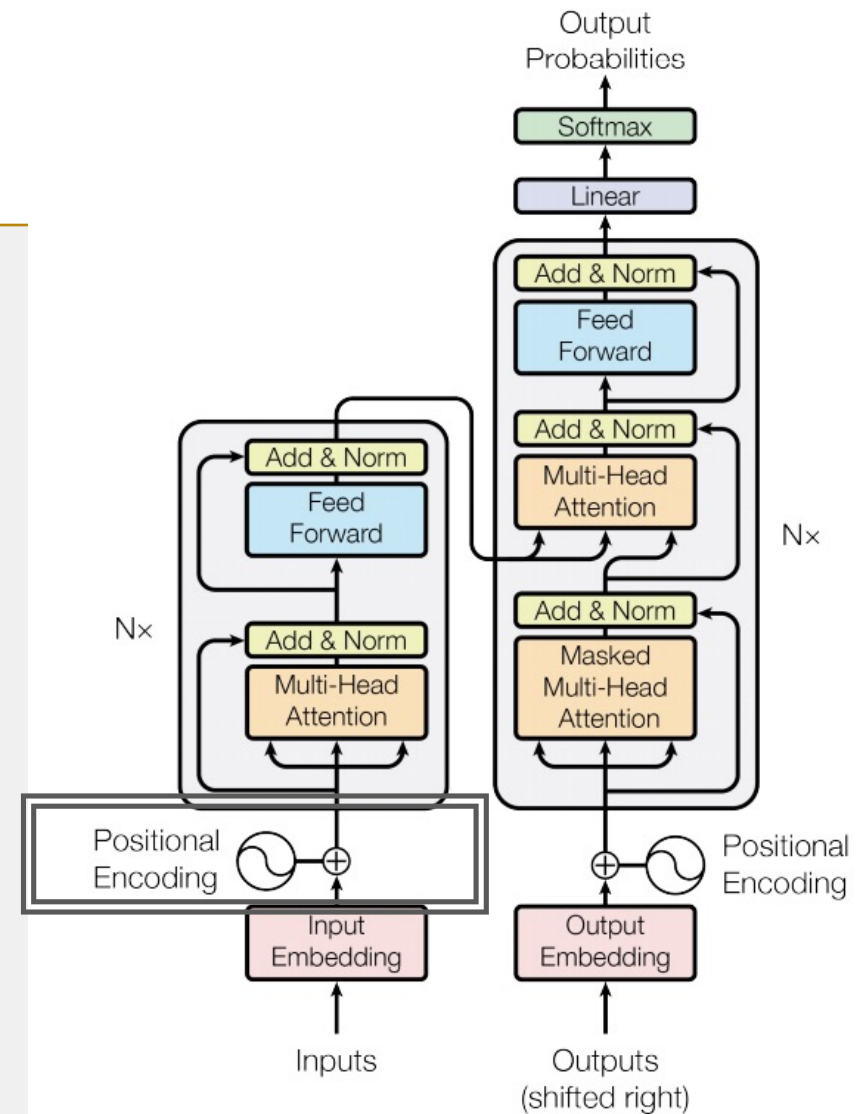


Figure 1: The Transformer - model architecture.



# Positional Embeddings

- The order in the RNN is inherently encoded in the model itself.
- This is not true for transformers.
  - If you scramble the order of inputs in the attention computation illustrated earlier you get exactly the same answer.
  - To address this issue, Transformer inputs are combined with positional embeddings specific to each position in an input sequence.



# Positional Embeddings

- The positional encodings have the same dimension  $d_{model}$  as the embeddings, so that the two can be summed:

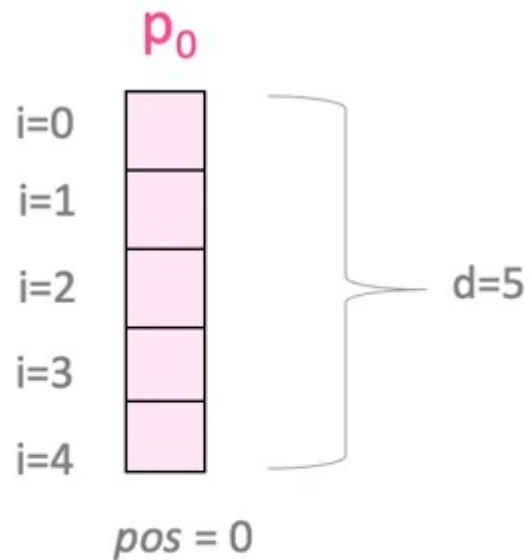
$$\begin{aligned} PE_{(pos, 2i)} &= \sin(pos/10000^{2i/d_{model}}) \\ PE_{(pos, 2i+1)} &= \cos(pos/10000^{2i/d_{model}}) \end{aligned}$$

- where  $pos$  is the position and  $i$  is the dimension. That is, each dimension of the positional encoding corresponds to a sinusoid.

# Positional Embeddings

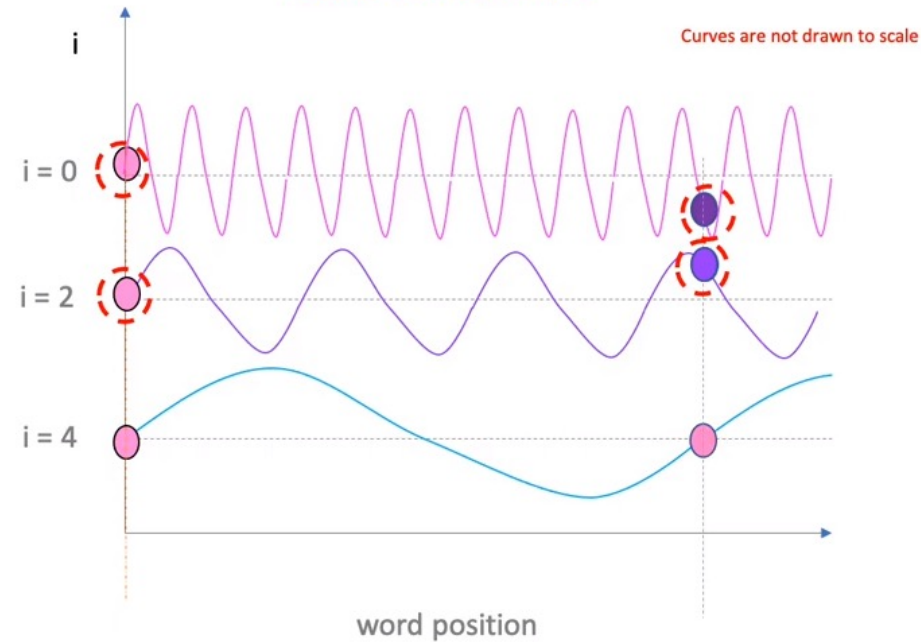
How does it work?

- We work with frequencies!



$$PE_{(pos, 2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

Position Embeddings



We generate a different sinusoidal curve for each index.

We locate each pos on each sinusoidal curve.

# Positional Embeddings

How does it work?

- A toy example:

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

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

$$PE(0,0) = \sin\left(\frac{0}{1000^0}\right) = \sin(0) = 0$$

$$PE(0,1) = \cos\left(\frac{0}{1000^0}\right) = \cos(0) = 1$$

Sequence	Index of token, $k$	Positional Encoding Matrix with $d=4$ , $n=100$			
		$i=0$	$i=0$	$i=1$	$i=1$
I	0	$P_{00}=\sin(0)$ = 0	$P_{01}=\cos(0)$ = 1	$P_{02}=\sin(0)$ = 0	$P_{03}=\cos(0)$ = 1
am	1	$P_{10}=\sin(1/1)$ = 0.84	$P_{11}=\cos(1/1)$ = 0.54	$P_{12}=\sin(1/10)$ = 0.10	$P_{13}=\cos(1/10)$ = 1.0
a	2	$P_{20}=\sin(2/1)$ = 0.91	$P_{21}=\cos(2/1)$ = -0.42	$P_{22}=\sin(2/10)$ = 0.20	$P_{23}=\cos(2/10)$ = 0.98
Robot	3	$P_{30}=\sin(3/1)$ = 0.14	$P_{31}=\cos(3/1)$ = -0.99	$P_{32}=\sin(3/10)$ = 0.30	$P_{33}=\cos(3/10)$ = 0.96

Positional Encoding Matrix for the sequence 'I am a robot'

# Positional Embeddings

In Python

```
def position_encode(pos):  
    input_sequence_length=4  
    pe=[0]*input_sequence_length  
    d_model=4  
    n=100  
    for i in range(0,input_sequence_length,2):  
        pe[i]=math.sin(pos/(n**((2*i)/d_model)))  
        pe[i+1]=math.cos(pos/(n**((2*i)/d_model)))  
    return pe
```

# Positional Embeddings

In Python

```
from scipy import spatial
```

```
pos0=position_encode(0)
print(f'pos0:{pos0}')
pos1=position_encode(1)
print(f'pos1:{pos1}')
pos2=position_encode(2)
print(f'pos2:{pos2}')
pos3=position_encode(3)
print(f'pos3:{pos3}')
result_1=1-spatial.distance.cosine(pos0,pos1)
result_2=1-spatial.distance.cosine(pos0,pos2)
result_3=1-spatial.distance.cosine(pos0,pos3)
print(f'Similarity between position 0 and 1 is {result_1}')
print(f'Similarity between position 0 and 2 is {result_2}')
print(f'Similarity between position 0 and 3 is {result_3}')
```

```
pos0:[0.0, 1.0, 0.0, 1.0]
pos1:[0.8414709848078965, 0.5403023058681398, 0.009999833334166664, 0.9999500004166653]
pos2:[0.9092974268256817, -0.4161468365471424, 0.01999866669333308, 0.9998000066665778]
pos3:[0.1411200080598672, -0.9899924966004454, 0.02999550020249566, 0.9995500337489875]
Similarity between position 0 and 1 is 0.7701261531424025
Similarity between position 0 and 2 is 0.2918265850597177
Similarity between position 0 and 3 is 0.004778768574271064
```

# Vanilla Transformer: Encoder and Decoder Stacks

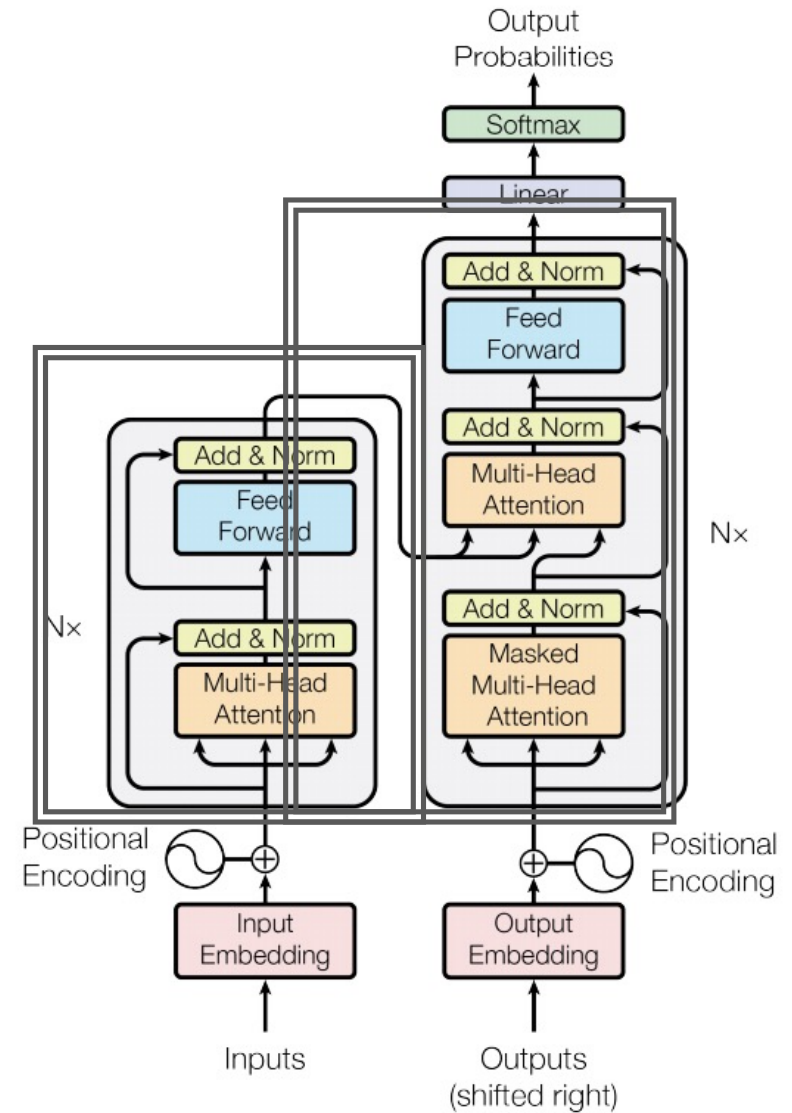
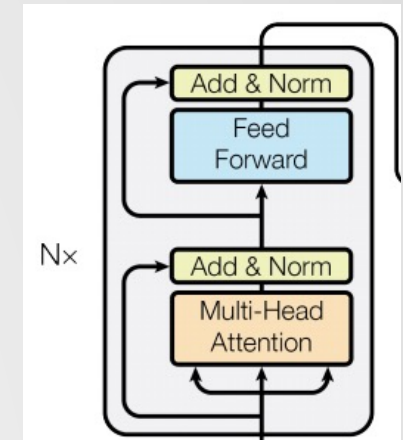


Figure 1: The Transformer - model architecture.

# Encoder Stacks : Summary

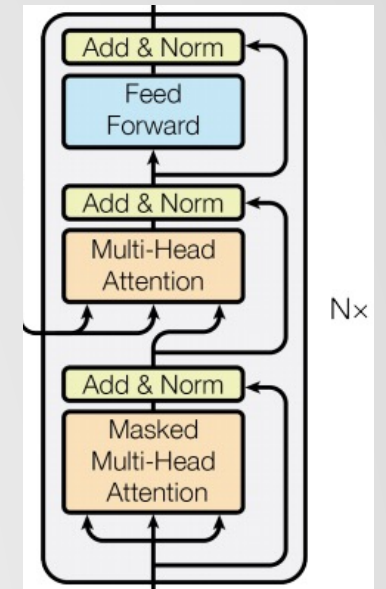
- The encoder is composed of a stack of  $N = 6$  identical layers.
- Each layer has two sub-layers:
  - The first is a multi-head self-attention mechanism,
  - The second is a simple, positionwise fully connected feed-forward network.
- They employ a residual connection around each of the two sub-layers, followed by layer normalization. That is, the output of each sub-layer is  $\text{LayerNorm}(x + \text{Sublayer}(x))$ , where  $\text{Sublayer}(x)$  is the function implemented by the sub-layer itself.
- To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension  $d_{\text{model}} = 512$ .





# Decoder Stacks: Summary

- The decoder is also composed of a stack of  $N = 6$  identical layers
- The decoder also has a third sub-layer, which performs multi-head attention over the output of the encoder stack.
- They employ residual connections around each of the sub-layers, followed by layer normalization.
- They also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions.
  - This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position  $i$  can depend only on the known outputs at positions less than  $i$ .



# Vanilla Transformer: Attention Layer

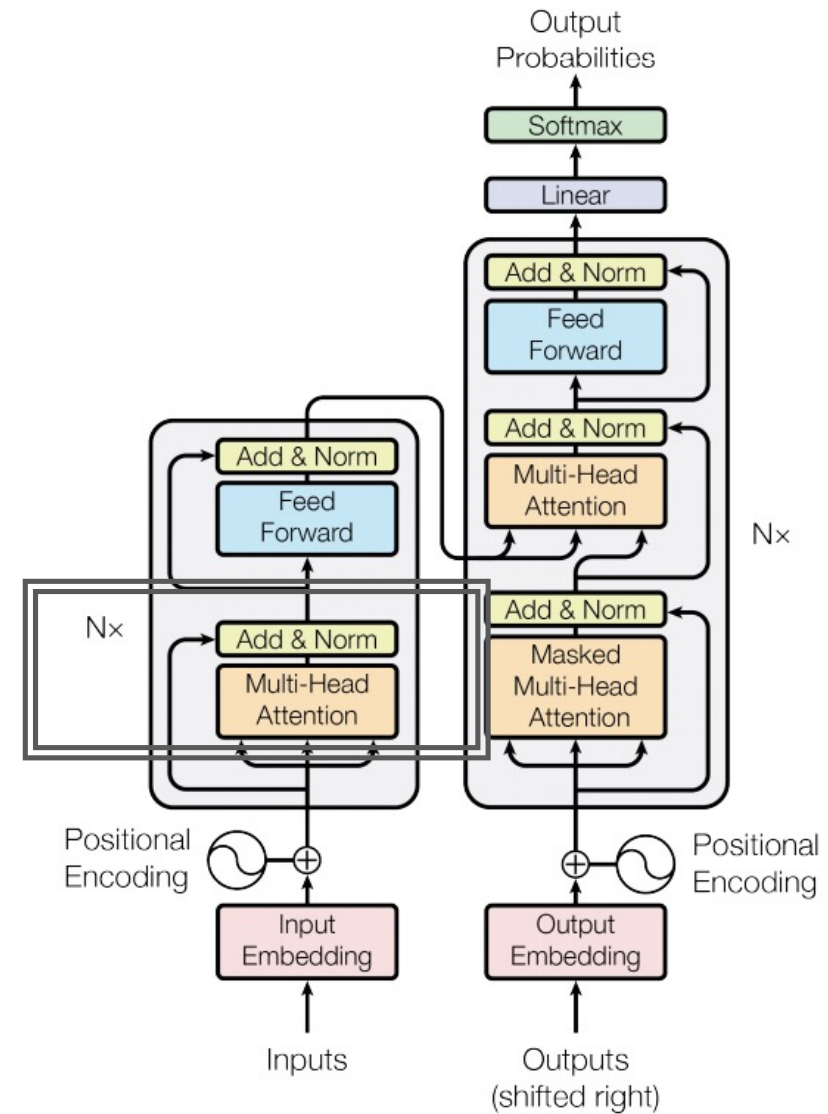


Figure 1: The Transformer - model architecture.

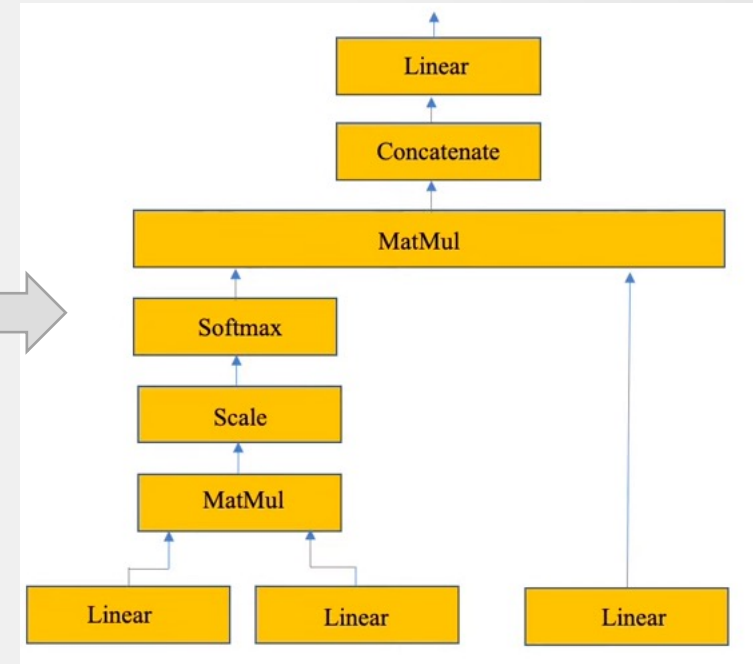
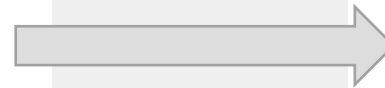
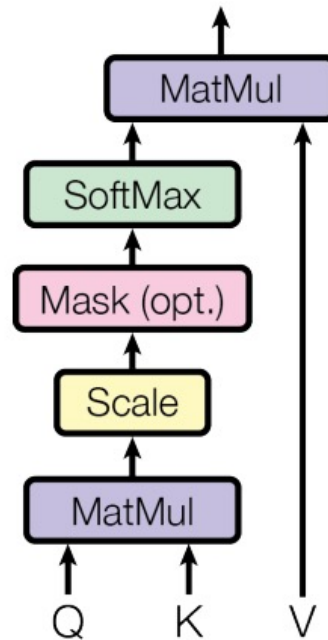
# Self Attention Layers

- Self-attention allows a network to directly extract and use information from arbitrarily large contexts without the need to pass it through intermediate recurrent connections as in RNNs.
- When processing each item in the input, the model has access to all the inputs up to and including the one under consideration, but no access to information about inputs beyond the current one.
  - ensures that we can use this approach to create language models and use them for autoregressive generation,
- The computation performed for each item is independent of all the other computations:
  - means that we can easily parallelize both forward inference and training of such models.

# Self Attention Layers

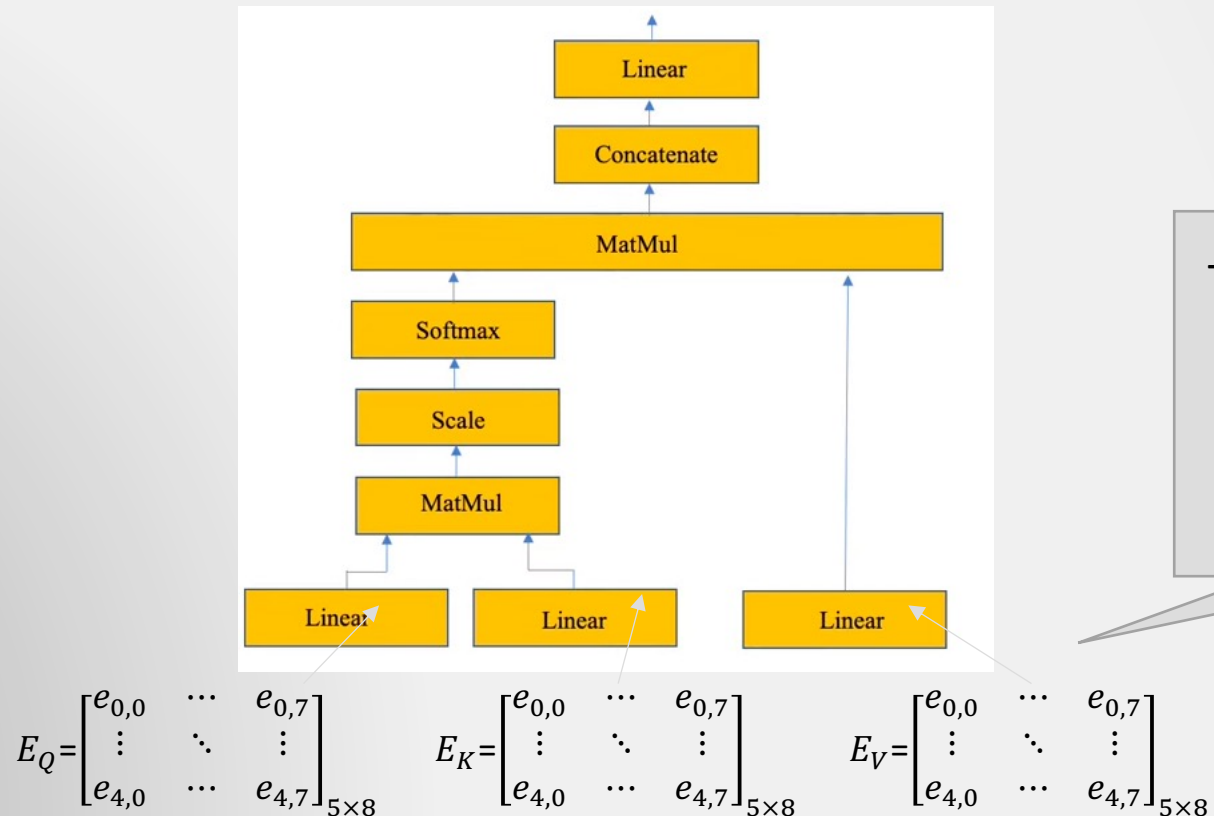
- K, Q and V all connect to a linear layer.

## Scaled Dot-Product Attention



# Self Attention Layers

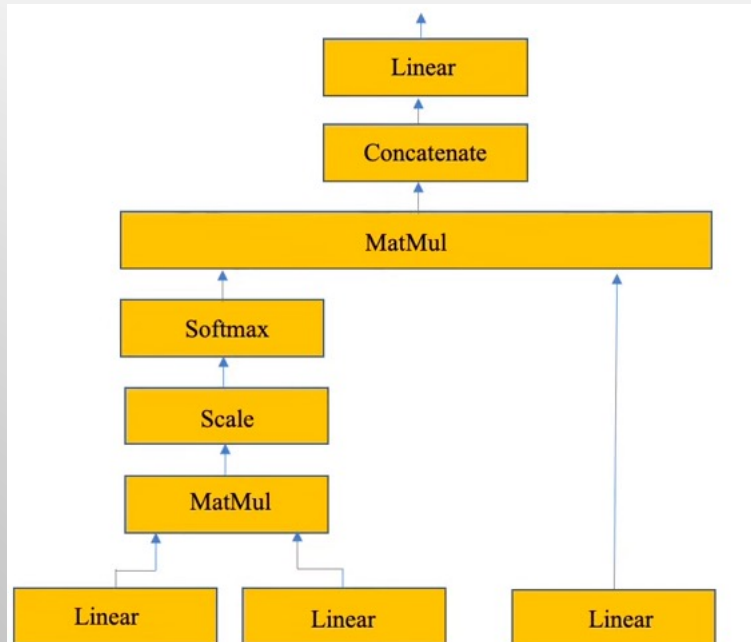
- Consider the example sentence S: “I watched the movie Transformers.”
  - the length of the sentence: 5, the embedding dimension: 8



These input vectors ( $E_Q$ ,  $E_K$ ,  $E_V$ ) are same and include embedding+positional embedding.

# Self Attention Layers

- Consider the example sentence  $S$ : "I watched the movie Transformers."
  - the length of the sentence: 5, the embedding dimension: 8



Fully connected linear layer

$$E_Q = \begin{bmatrix} e_{0,0} & \cdots & e_{0,7} \\ \vdots & \ddots & \vdots \\ e_{4,0} & \cdots & e_{4,7} \end{bmatrix}_{5 \times 8} \begin{matrix} \text{I} \\ \text{watched} \\ \dots \\ \text{Transformers} \end{matrix} \times \begin{matrix} \boxed{8 \times 2} \end{matrix} = \begin{matrix} \boxed{5 \times 2} \end{matrix} \text{Query}$$
$$E_K = \begin{bmatrix} e_{0,0} & \cdots & e_{0,7} \\ \vdots & \ddots & \vdots \\ e_{4,0} & \cdots & e_{4,7} \end{bmatrix}_{5 \times 8} \times \begin{matrix} \boxed{8 \times 2} \end{matrix} = \begin{matrix} \boxed{5 \times 2} \end{matrix} \text{Key}$$
$$E_V = \begin{bmatrix} e_{0,0} & \cdots & e_{0,7} \\ \vdots & \ddots & \vdots \\ e_{4,0} & \cdots & e_{4,7} \end{bmatrix}_{5 \times 8} \times \begin{matrix} \boxed{8 \times 2} \end{matrix} = \begin{matrix} \boxed{5 \times 2} \end{matrix} \text{Value}$$

Notice that these new vectors (Key, Query, Value) are smaller in dimension than the embedding vector. Their dimensionality is 2, while the embedding and encoder input/output vectors have dimensionality of 8.

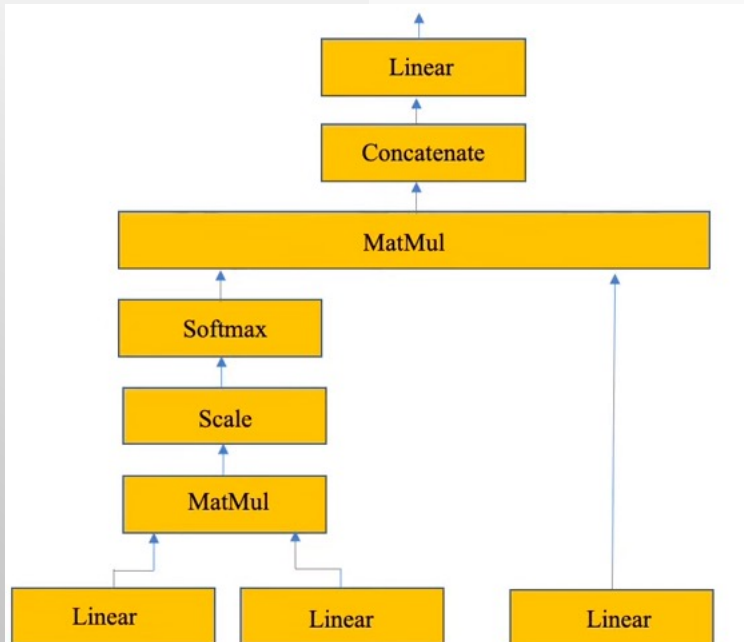
The linear layer here has 2 nodes and each layer has its own weights having a total of  $2 \times 8 = 16$  weights. It is directly connected to the embedding (with pos) layer.

# Self Attention Layers

Generally:

$$\text{Query Size} = \text{Embedding Size} / \text{Number of heads}$$

Let's set number of heads as 4.



Notice that these new vectors (Key, Query, Value) are smaller in dimension than the embedding vector. Their dimensionality is 2, while the embedding and encoder input/output vectors have dimensionality of 8.

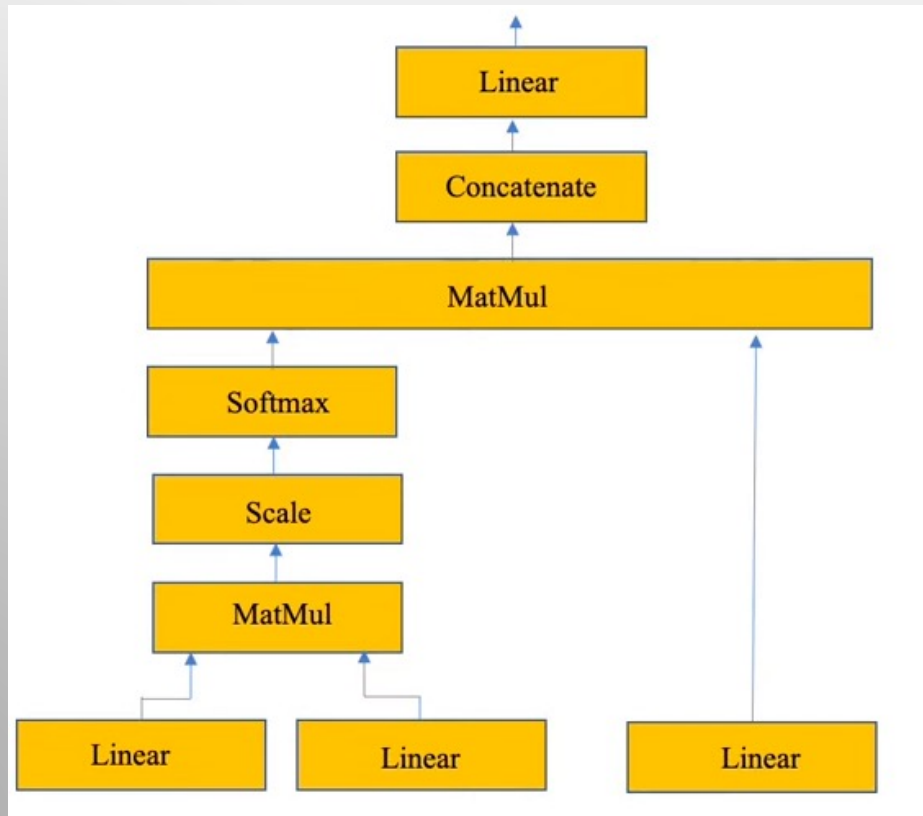
$$\begin{aligned}
 E_Q &= \begin{bmatrix} e_{0,0} & \cdots & e_{0,7} \\ \vdots & \ddots & \vdots \\ e_{4,0} & \cdots & e_{4,7} \end{bmatrix}_{5 \times 8} \quad \text{I watched ... Transformers} \quad \times \quad \begin{bmatrix} \phantom{0} \\ \phantom{0} \\ \phantom{0} \\ \phantom{0} \end{bmatrix}_{8 \times 2} = \begin{bmatrix} \phantom{0} \\ \phantom{0} \\ \phantom{0} \\ \phantom{0} \end{bmatrix}_{5 \times 2} \text{ Query} \\
 E_K &= \begin{bmatrix} e_{0,0} & \cdots & e_{0,7} \\ \vdots & \ddots & \vdots \\ e_{4,0} & \cdots & e_{4,7} \end{bmatrix}_{5 \times 8} \quad \times \quad \begin{bmatrix} \phantom{0} \\ \phantom{0} \\ \phantom{0} \\ \phantom{0} \end{bmatrix}_{8 \times 2} = \begin{bmatrix} \phantom{0} \\ \phantom{0} \\ \phantom{0} \\ \phantom{0} \end{bmatrix}_{5 \times 2} \text{ Key} \\
 E_V &= \begin{bmatrix} e_{0,0} & \cdots & e_{0,7} \\ \vdots & \ddots & \vdots \\ e_{4,0} & \cdots & e_{4,8} \end{bmatrix}_{5 \times 8} \quad \times \quad \begin{bmatrix} \phantom{0} \\ \phantom{0} \\ \phantom{0} \\ \phantom{0} \end{bmatrix}_{8 \times 2} = \begin{bmatrix} \phantom{0} \\ \phantom{0} \\ \phantom{0} \\ \phantom{0} \end{bmatrix}_{5 \times 2} \text{ Value}
 \end{aligned}$$

Fully connected linear layer

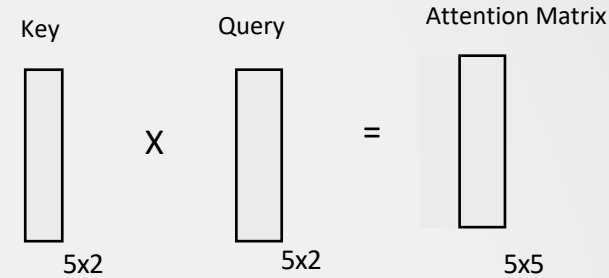
The linear layer here has 2 nodes and each layer has its own weights having a total of  $2 \times 8 = 16$  weights.  
It is directly connected to the embedding (with pos) layer.



# Self Attention Layers



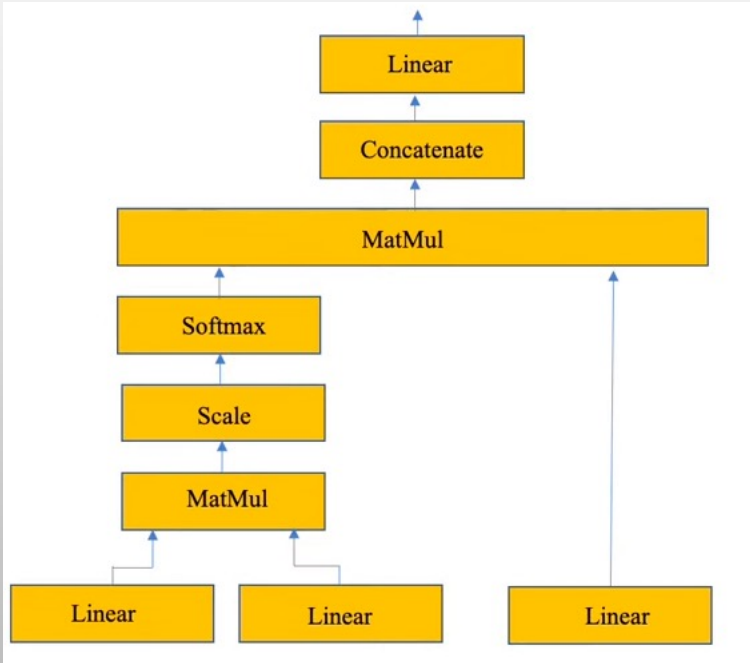
MatMul Step:



	I	watched	the	movie	Transformers
I	98	44	22	24	34
watched	44	95	24	67	68
the	22	24	99	34	34
movie	24	67	34	92	65
Transformers	34	68	34	65	98

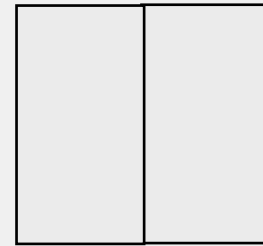
Numbers are randomly selected i.e. co-occurrence number.

# Self Attention Layers



## Scale Step:

Attention Matrix, divided each by  $\sqrt{d_{model}}$ , in our case  $\sqrt{8}$

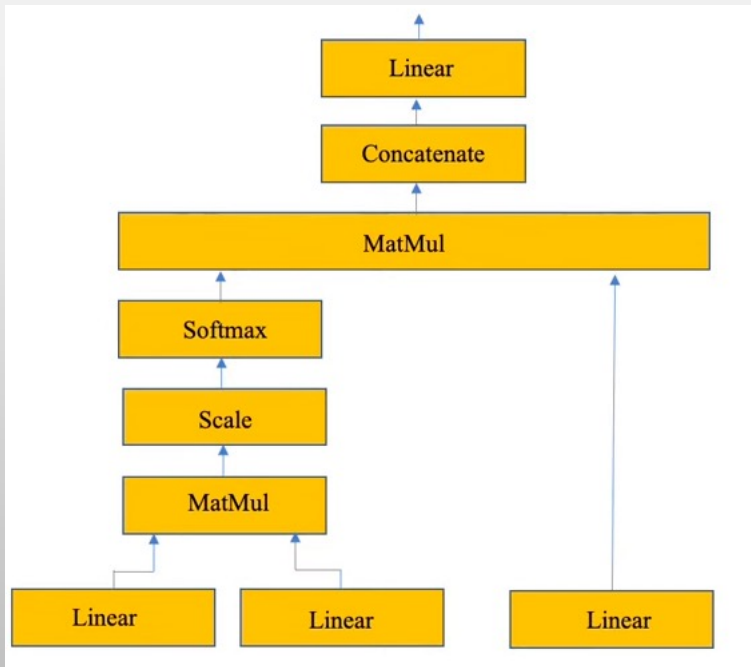


5x5

They have stated that for large values of  $d_{model}$ , the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients. To counteract this effect, they scale the dot products by  $\sqrt{d_{model}}$

	I	watched	the	movie	Transformers
I	34.65	15.56	7.78	8.49	12.02
watched	15.56	33.59	8.49	23.69	24.04
the	7.78	8.49	35.00	12.02	12.02
movie	8.49	23.69	12.02	32.53	22.98
Transformers	12.02	24.04	12.02	22.98	34.65

# Self Attention Layers

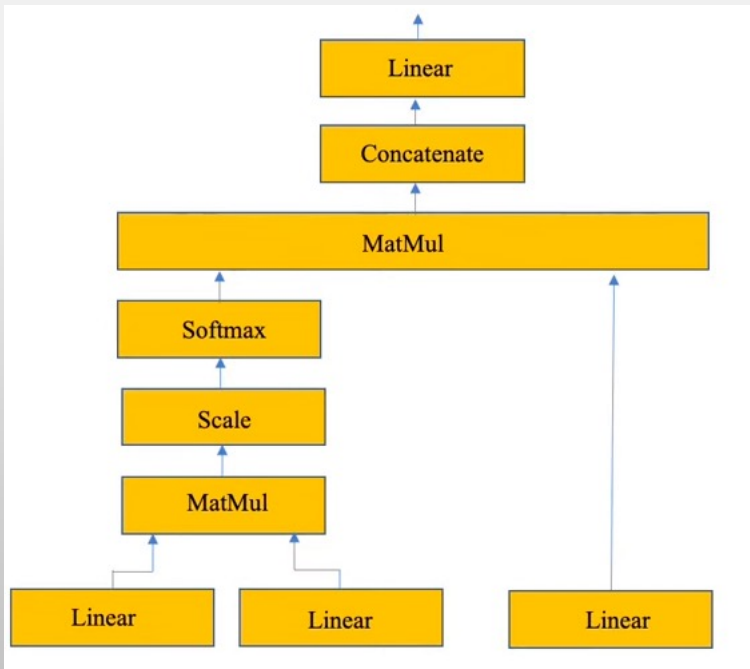


**Softmax Step:**

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

	I	watched	the	movie	Transformers
I	1.00	0.00	0.00	0.00	0.00
watched	0.00	1.00	0.00	0.00	0.00
the	0.00	0.00	1.00	0.00	0.00
movie	0.00	0.00	0.00	1.00	0.00
Transformers	0.00	0.00	0.00	0.00	1.00

# Self Attention Layers



## MatMul Step with Value matrix:

We have the scaled attention matrix showing which words are related with which.

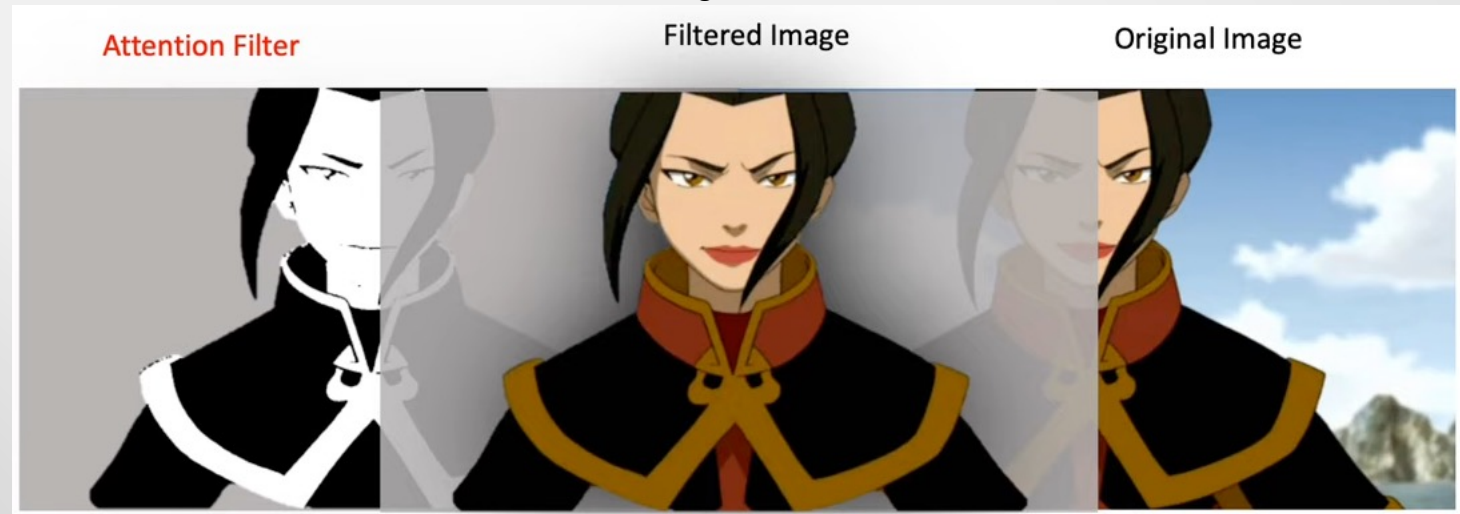
It is like a weight matrix which needs to be multiplied with  $V$  matrix to get the values from.

$$\begin{array}{ccc} \text{Attention Matrix} & & V \\ \begin{array}{|c|} \hline \\ \hline \end{array} & \times & \begin{array}{|c|} \hline \\ \hline \end{array} \\ 5 \times 5 & & 5 \times 2 \\ & & = \\ & & \begin{array}{|c|} \hline \\ \hline \end{array} \\ & & 5 \times 2 \\ & & \text{MatMul Result} \end{array}$$

# Self Attention Layers

## MatMul Step with Value matrix:

Intuition from images:



Attention matrix  
obtained using the  
following:

$$\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

Filtered Image: *Attention Matrix* x *V*

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

Image used to create *K*, *Q*, *V*

# Self Attention Layers

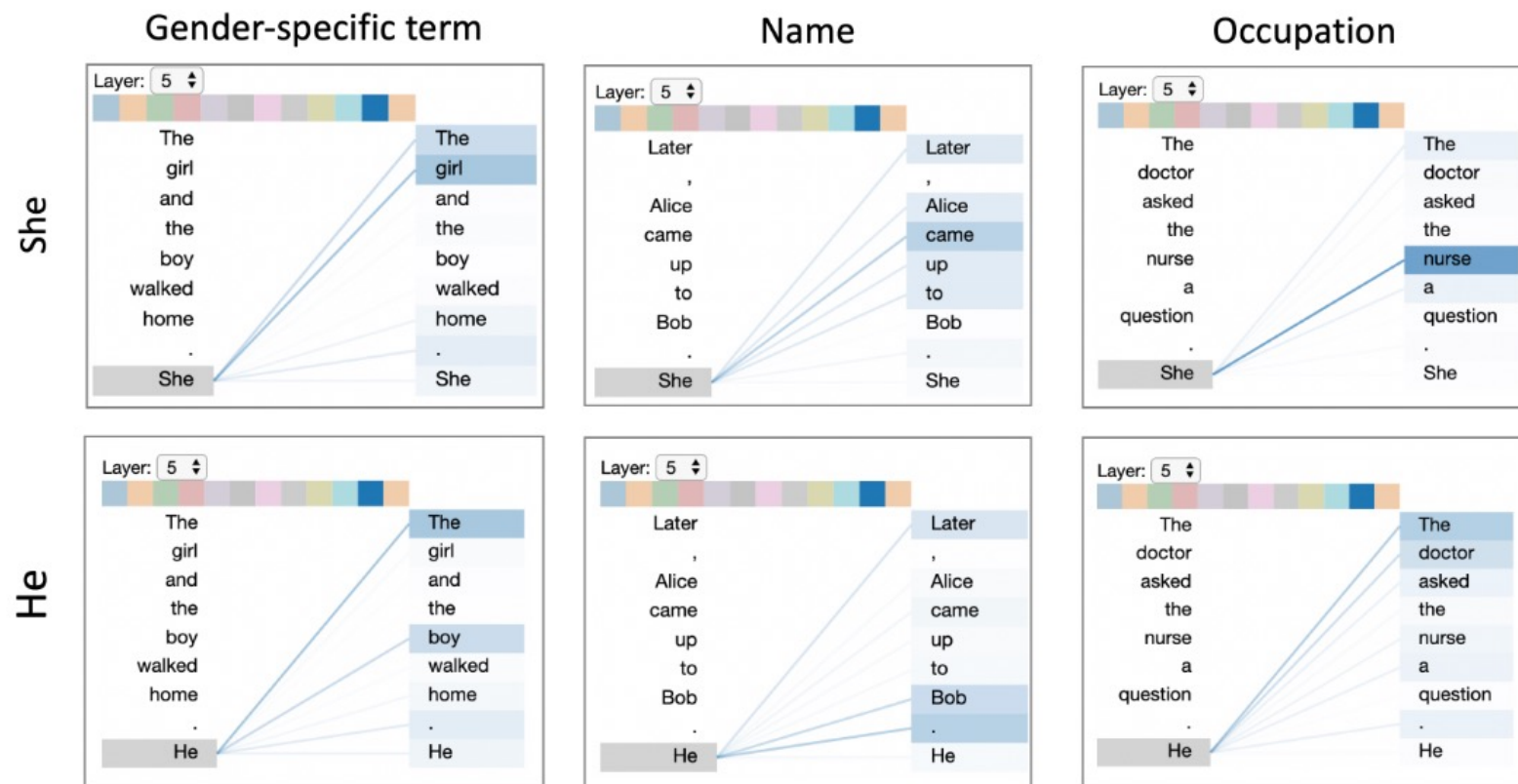


Figure 4: Attention pattern in GPT-2 related to coreference resolution suggests the model may encode gender bias.

# Self Attention Layers

## Multi-head Attention

- The different words in a sentence can relate to each other in many different ways simultaneously.
  - For example, distinct syntactic, semantic, and discourse relationships can hold between verbs and their arguments in a sentence.
- Transformers address this issue with multihead self attention layers.
- These are sets of self-attention layers, called heads, that reside in parallel layers at the same depth in a model, each with its own set of parameters.



# Self Attention Layers

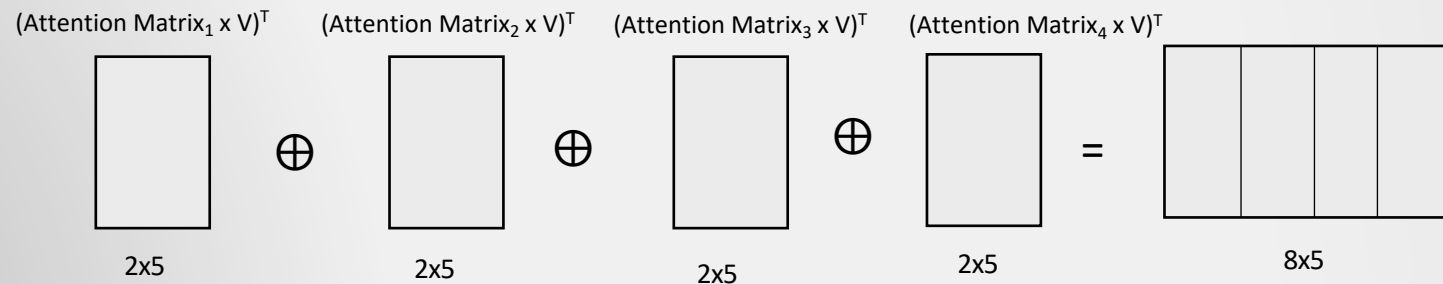
## Multi-head Attention

- Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

where  $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$

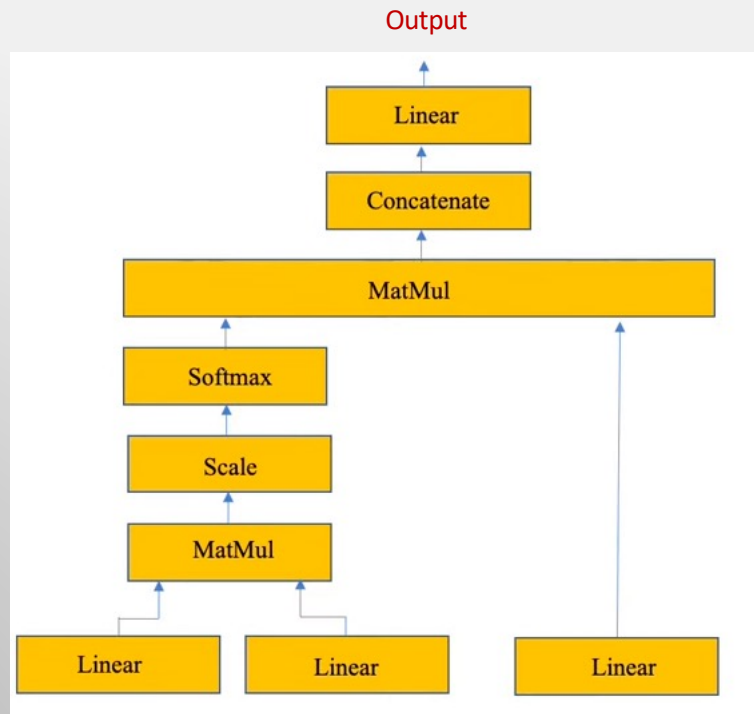
Where the projections are parameter matrices  $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$ ,  $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$ ,  $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$  and  $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$ .



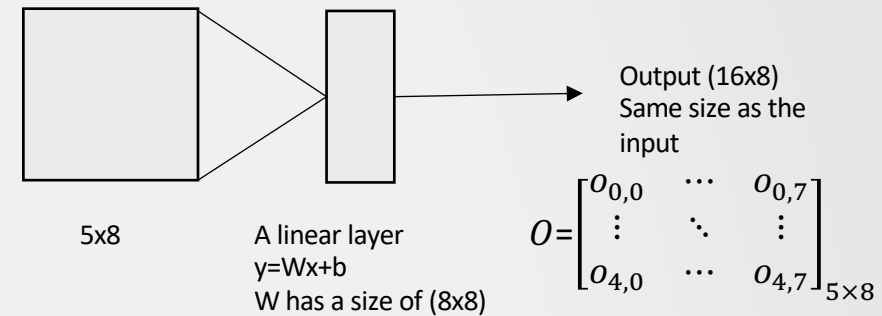
We have four heads.

# Self Attention Layers

## Output:



Linear layer at the end shrinks the size of the concatenated matrix.  
We don't want to grow it with attention layers.



# Vanilla Transformer: Residual Connection

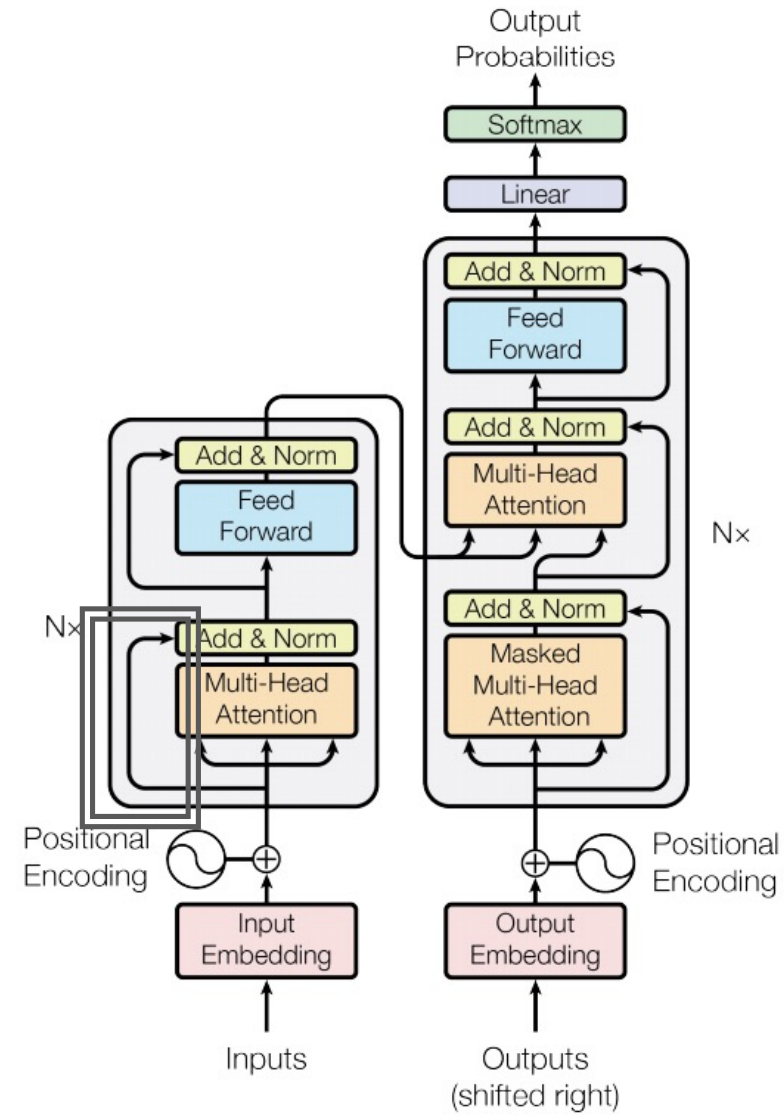
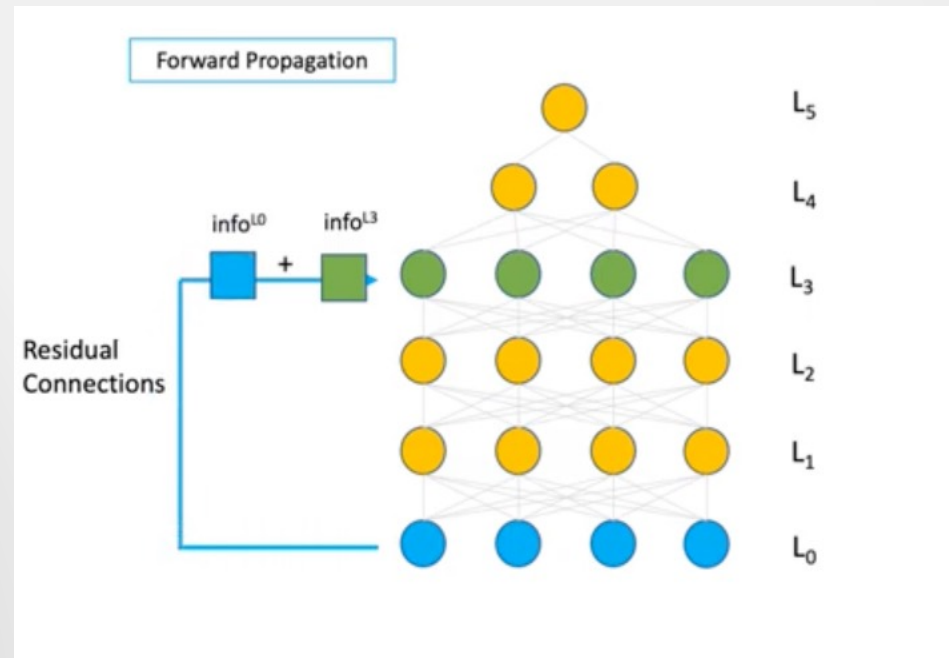


Figure 1: The Transformer - model architecture.

# Residual Connection

- Information loss at the last hidden layers can be solved with so-called high-ways aka residual connections.



# Vanilla Transformer: Add&Normalization Layer Feedforward Layer

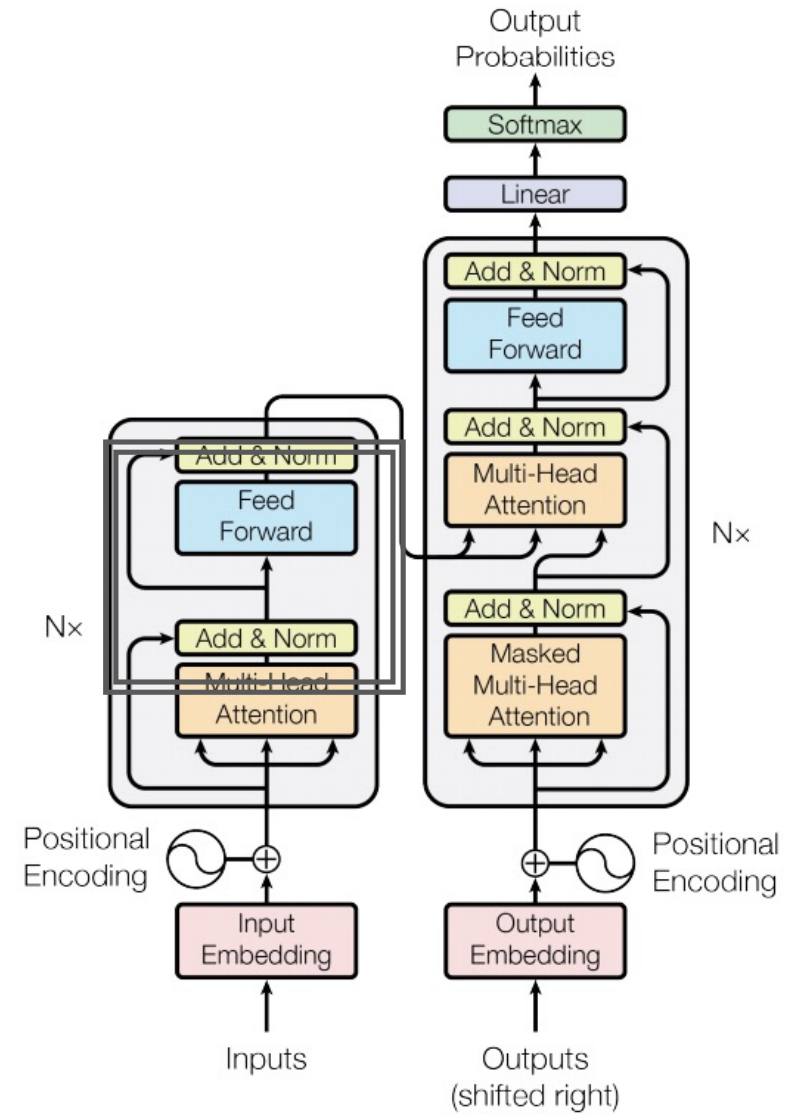


Figure 1: The Transformer - model architecture.

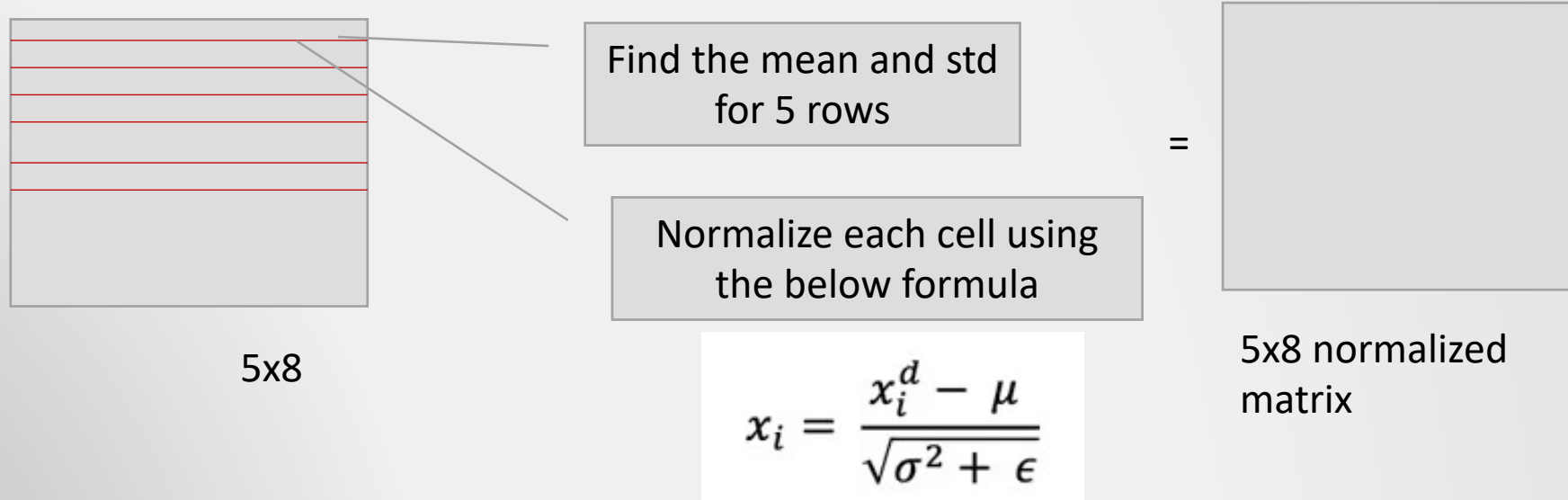
# Add&Normalization Layer

- We simply add the residual connection and the output of multihead attention.

$$\begin{array}{ccc}
 E_Q = \begin{bmatrix} e_{0,0} & \cdots & e_{0,7} \\ \vdots & \ddots & \vdots \\ e_{4,0} & \cdots & e_{4,7} \end{bmatrix}_{5 \times 8} & \begin{array}{l} \text{I} \\ \text{watched} \\ \dots \\ \dots \end{array} & + \quad O = \begin{bmatrix} o_{0,0} & \cdots & o_{0,7} \\ \vdots & \ddots & \vdots \\ o_{4,0} & \cdots & o_{4,7} \end{bmatrix}_{5 \times 8} \\
 \text{Embedding+pos} & & \text{Output of multi-head attention}
 \end{array}
 = \begin{array}{c} \boxed{\phantom{5 \times 8}} \\ 5 \times 8 \end{array}$$

# Add&Normalization Layer

- We simply add the residual connection and the output of multihead attention.

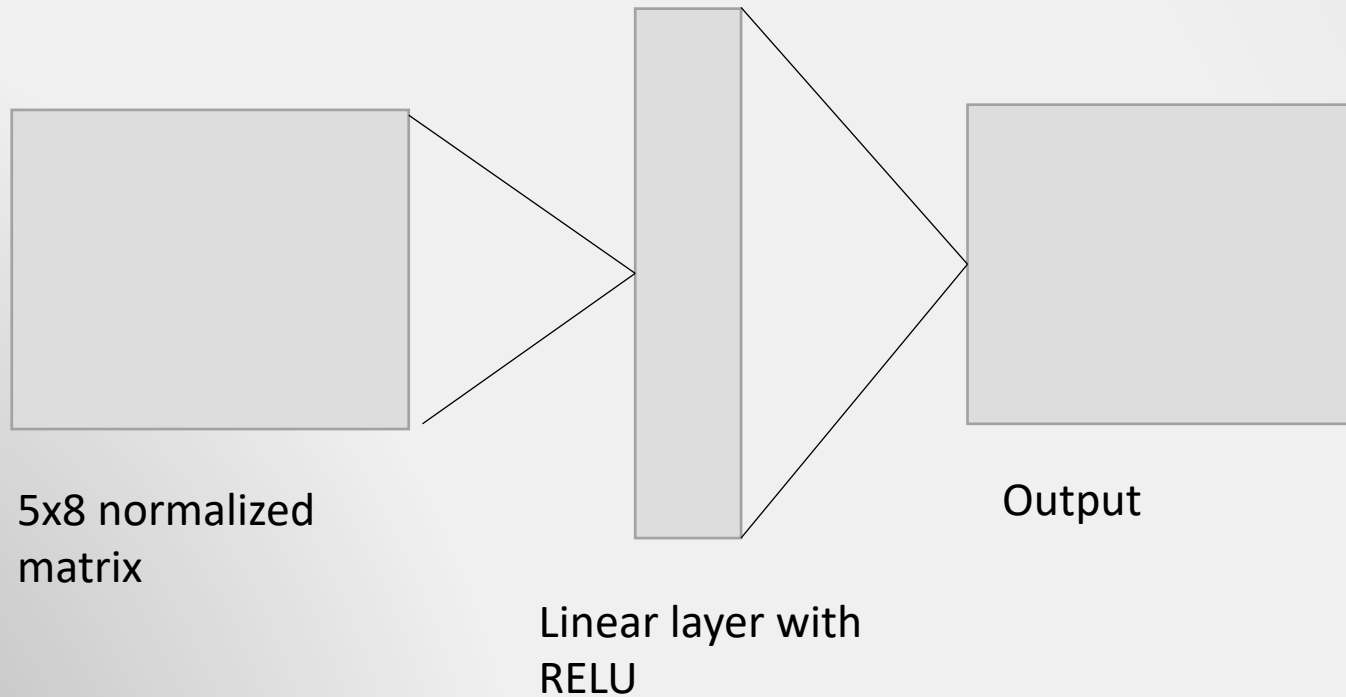


$\epsilon$  can be set as 0.0001 to make the denominator as non zero



# Feedforward Layer

- Linear layers with RELU activation function



# Vanilla Transformer: Decoder

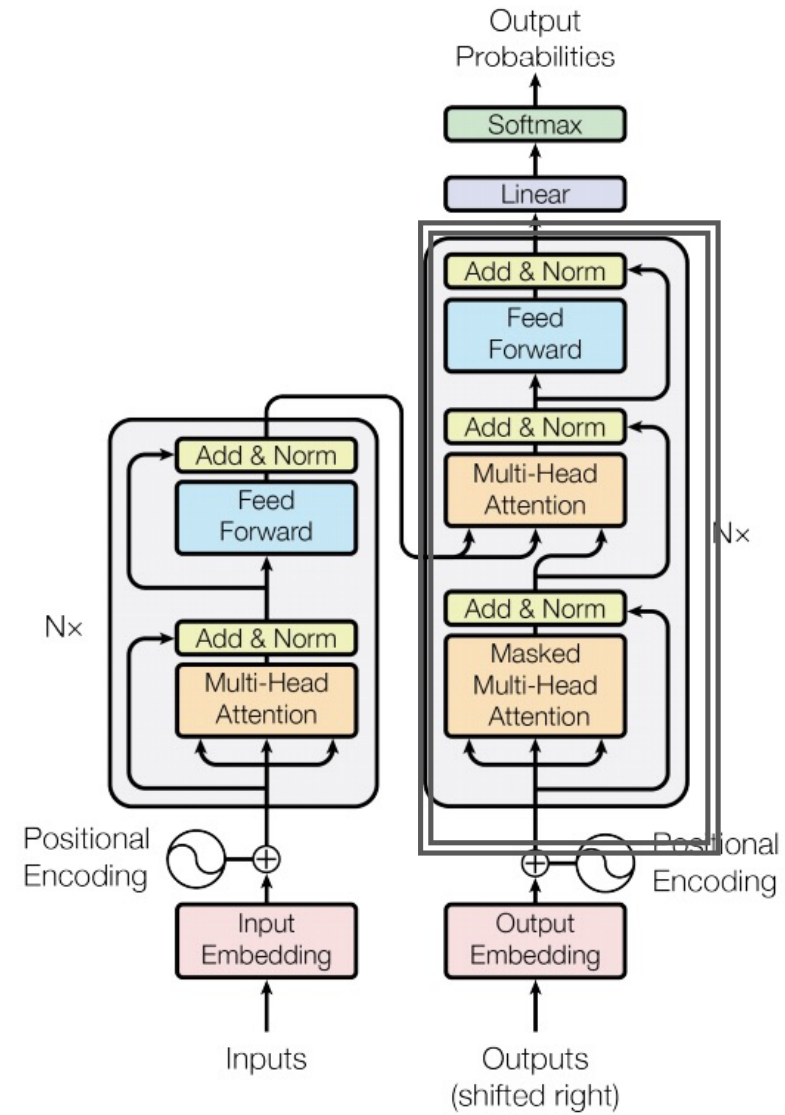
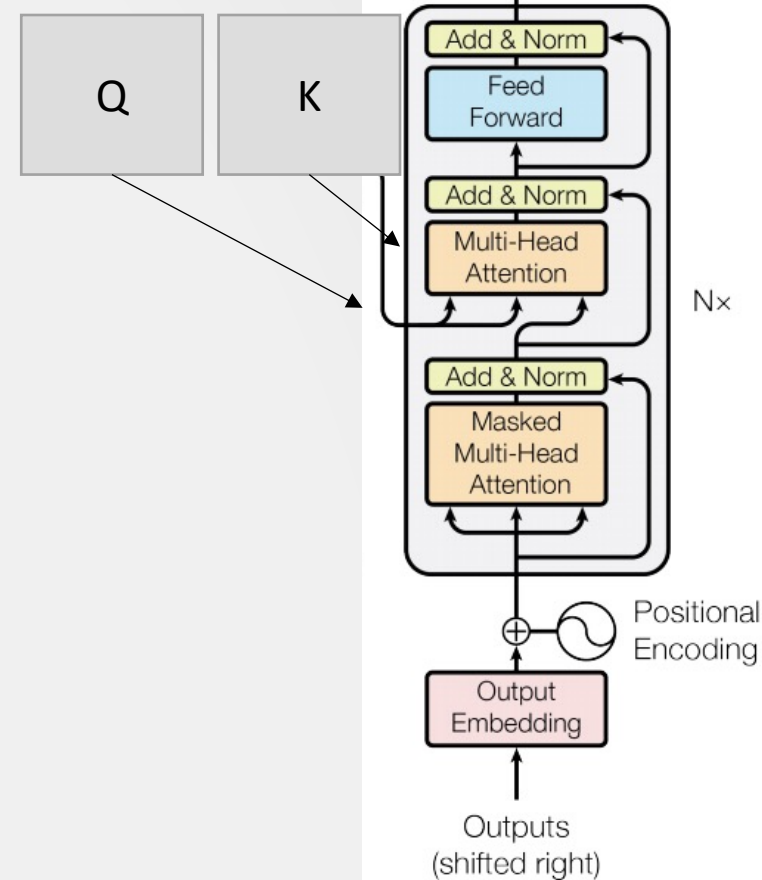


Figure 1: The Transformer - model architecture.

# Decoder

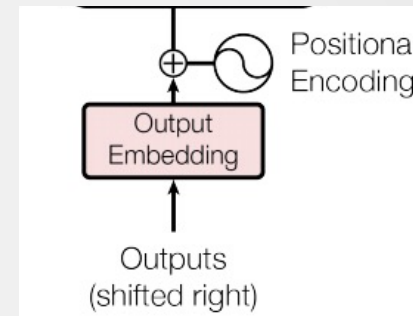
- Decoder layer takes two inputs:
  - The output of the encoder layer
  - The predicted/generated output so far
- On the other hand, encoder layer takes only inputs
- The output of the encoder layer is copied twice to construct the Q and K matrices.



# Decoder

## Output Embedding

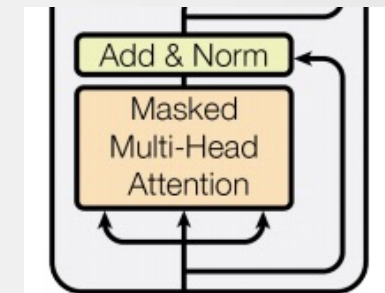
- Step 1:
- The first <s> start of sentence is provided.
- The embedding with its positional embedding is generated.



# Decoder

## Output Embedding

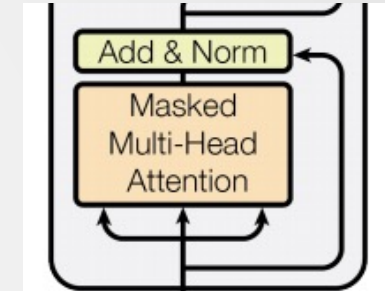
- Step 2:
- During training, the model accepts the input and output as a whole sequence.
- For example:
  - Input: <I watched the movie Transformers.>
  - Output: <Transformers filmini seyrettim.>
- My vocabulary size: 3+2(start and end of sentence tags)



# Decoder

## Output Embedding

- Step 2:
- Although our input is provided to the encoder, we provide the next sequence items one by one via masked multi-head attention.



Input: <I watched the movie Transformers.>

Next sequence: <Transformers filmini seyrettim.>

Next sequence: <Transformers filmini seyrettim.>

Next sequence: <Transformers filmini seyrettim.>

# Decoder

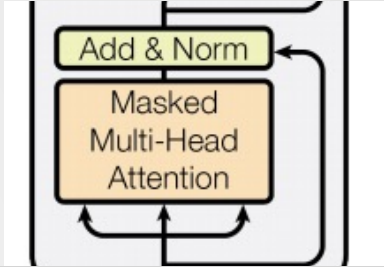
## Output Embedding

- Step 2:
- The outputs in the future time steps are masked via  $-\infty$  values.

<Transformers filmini seyrettim.>

	<s>	Transformers	filmini	seyrettim	</s>	
<s>	32	4	7	4	1	
Transformers	4	40	7	3	3	
filmini	7	7	31	2	4	
seyrettim	4	3	2	49	23	
<s>	1	3	4	23	0	

+



	<s>	Transformers	filmini	seyrettim	</s>
<s>	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$
Transformers	0	0	$-\infty$	$-\infty$	$-\infty$
filmini	0	0	0	$-\infty$	$-\infty$
seyrettim	0	0	0	0	$-\infty$
<s>	0	0	0	0	0

Attention filter

Masked filter

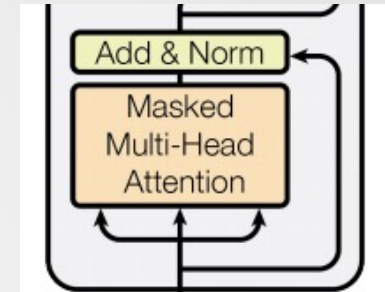
We obtain masked-attention filter at the end.



# Decoder

## Output Embedding

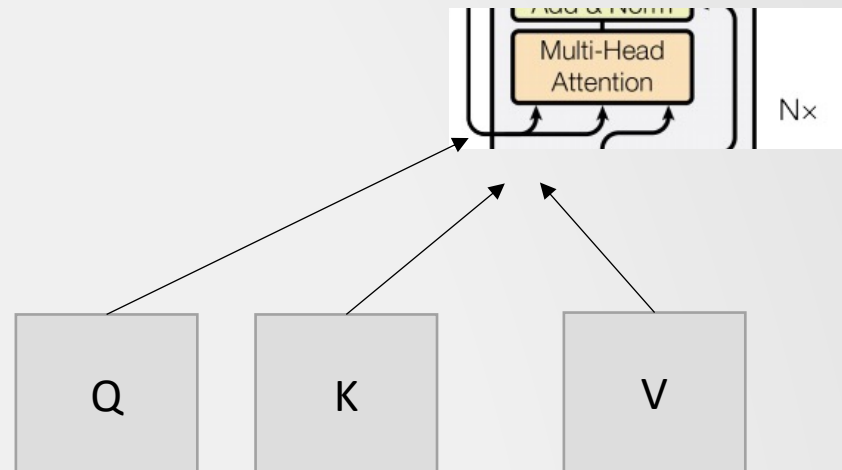
- Step 2:
- The reason of  $-\infty$  values is that when we pass it from softmax, they will become zero for the words in the future.
- Don't forget that we use teacher forcing for training.
- We pass it from add&normalization layer to obtain our V matrix.



# Decoder

## Output Embedding

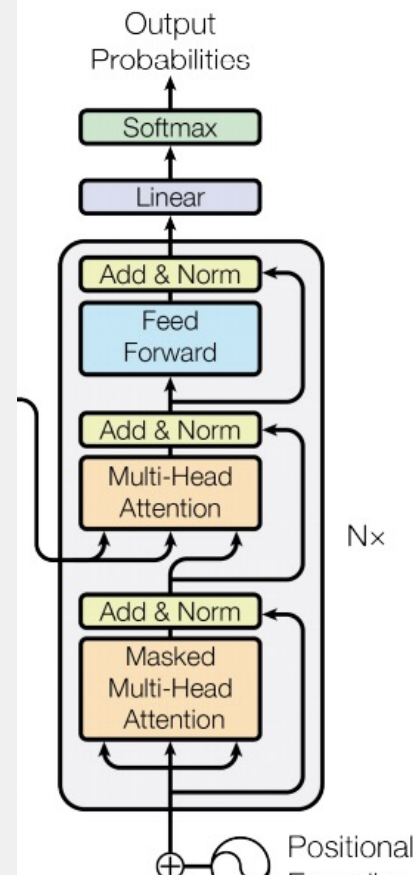
- Step 3:
- This embedding  $V$  together with the previously generated  $Q$  and  $K$  matrices from the encoding layer is fed to the multi-head attention.



# Decoder

## Output Embedding

- Step 4:
- The remaining steps feedforward, add&norm, linear layer and softmax steps are same as in the encoding step.
- We produce the probability of each word in our vocabulary.
- We use cross entropy to calculate the loss.



# Results

## Dataset

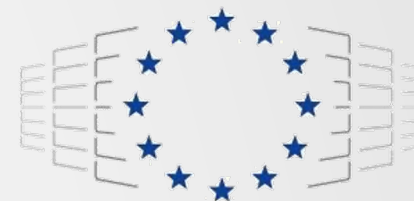
- WMT 2014 English-German dataset consisting of about 4.5 million sentence pairs are used.
- Sentences were encoded using byte-pair encoding, which has a shared source- target vocabulary of about 37000 tokens.
- For English-French, they used the significantly larger WMT 2014 English-French dataset consisting of 36M sentences and split tokens into a 32000 word-piece vocabulary.
- Sentence pairs were batched together by approximate sequence length.
- Each training batch contained a set of sentence pairs containing approximately 25000 source tokens and 25000 target tokens.

# Results

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	<b>41.29</b>	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	<b><math>3.3 \cdot 10^{18}</math></b>	
Transformer (big)	<b>28.4</b>	<b>41.8</b>	$2.3 \cdot 10^{19}$	

# Thanks!



**EuroHPC**  
Joint Undertaking

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 951732. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Germany, Bulgaria, Austria, Croatia, Cyprus, Czech Republic, Denmark, Estonia, Finland, Greece, Hungary, Ireland, Italy, Lithuania, Latvia, Poland, Portugal, Romania, Slovenia, Spain, Sweden, United Kingdom, France, Netherlands, Belgium, Luxembourg, Slovakia, Norway, Switzerland, Turkey, Republic of North Macedonia, Iceland, Montenegro