Transformers

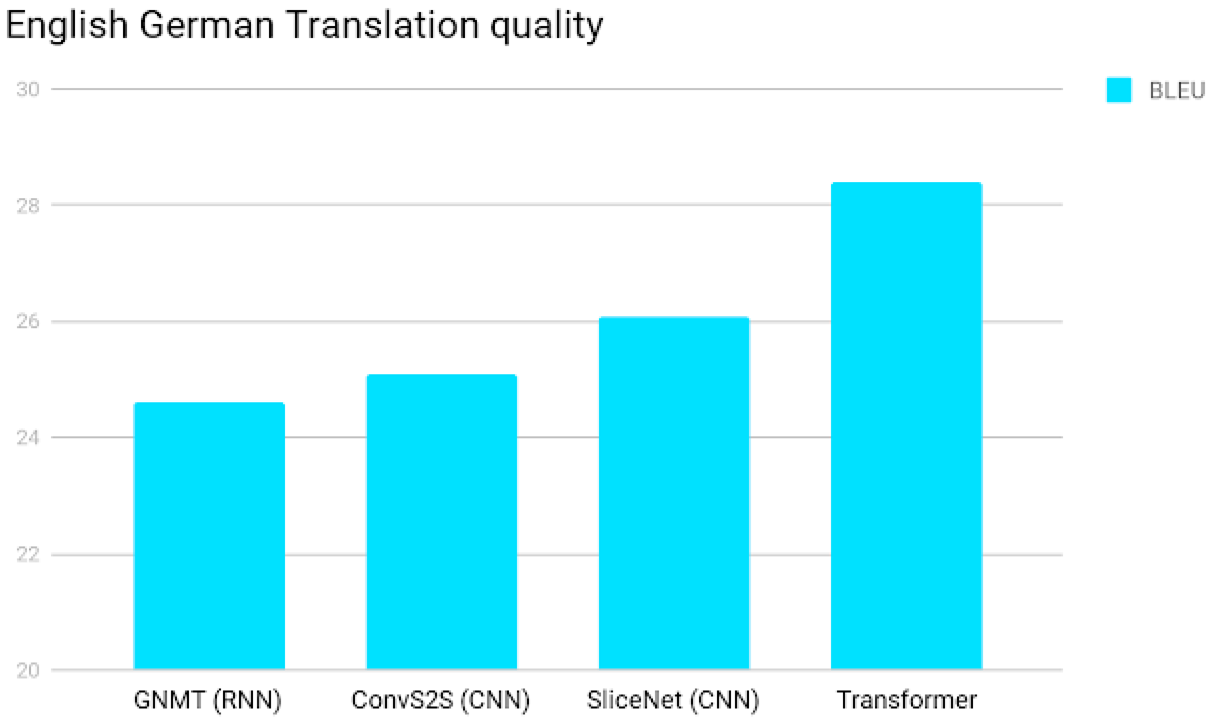
Dr Mehreen Alam

https://jalammar.github.io/illustrated-transformer/

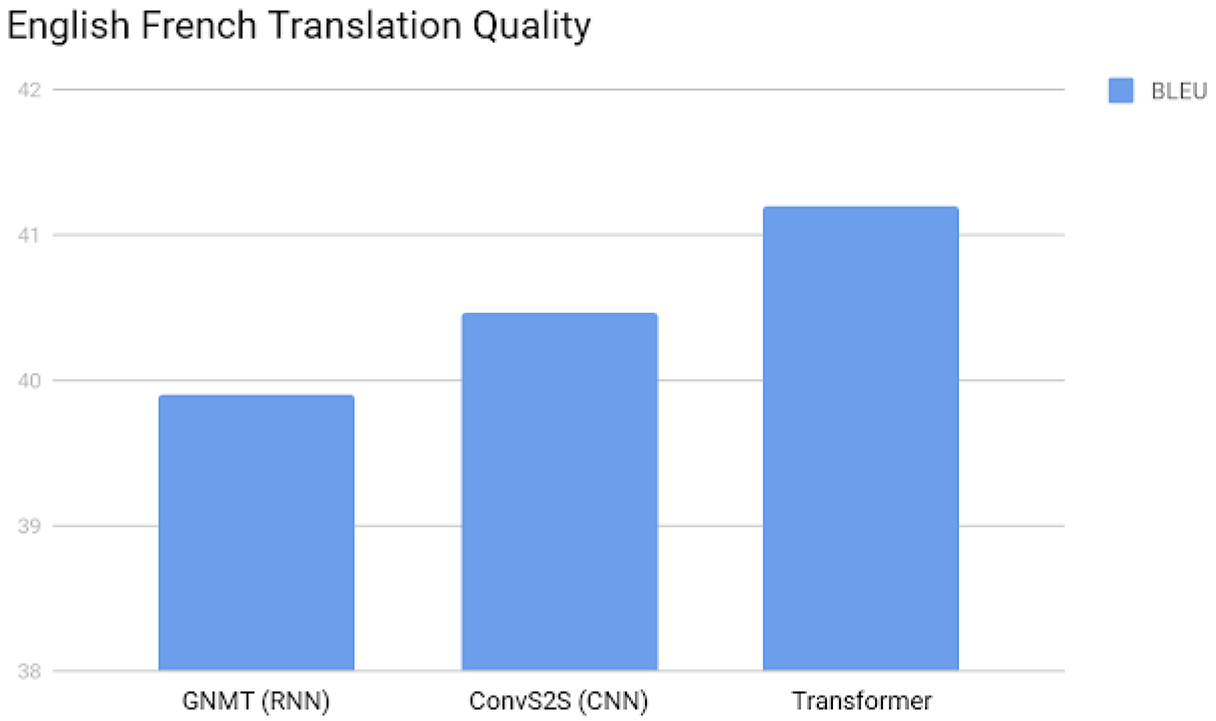
https://machinelearningmastery.com/a-gentle-introduction-to-positional-encoding-in-transformer-models-part-1/

Challenges of RNNs

- Long range dependencies
- Parallelization



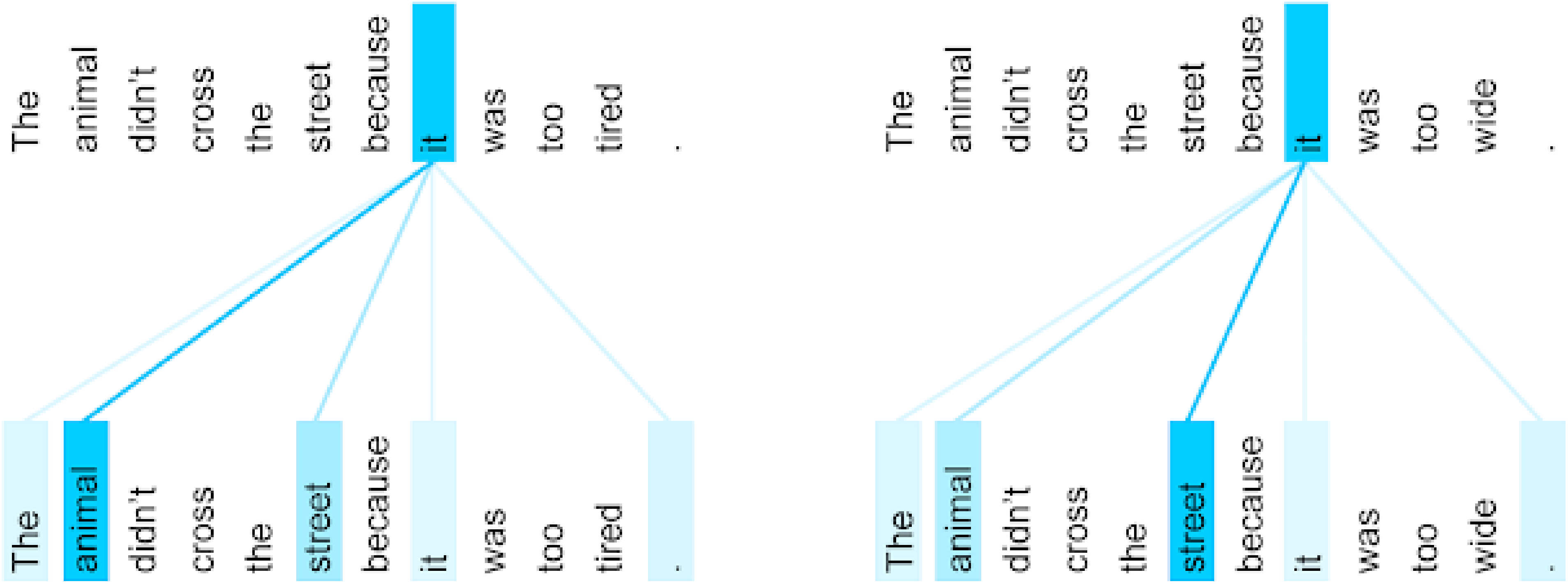
BLEU scores (higher is better) of single models on the standard WMT newstest2014 English to German translation benchmark.

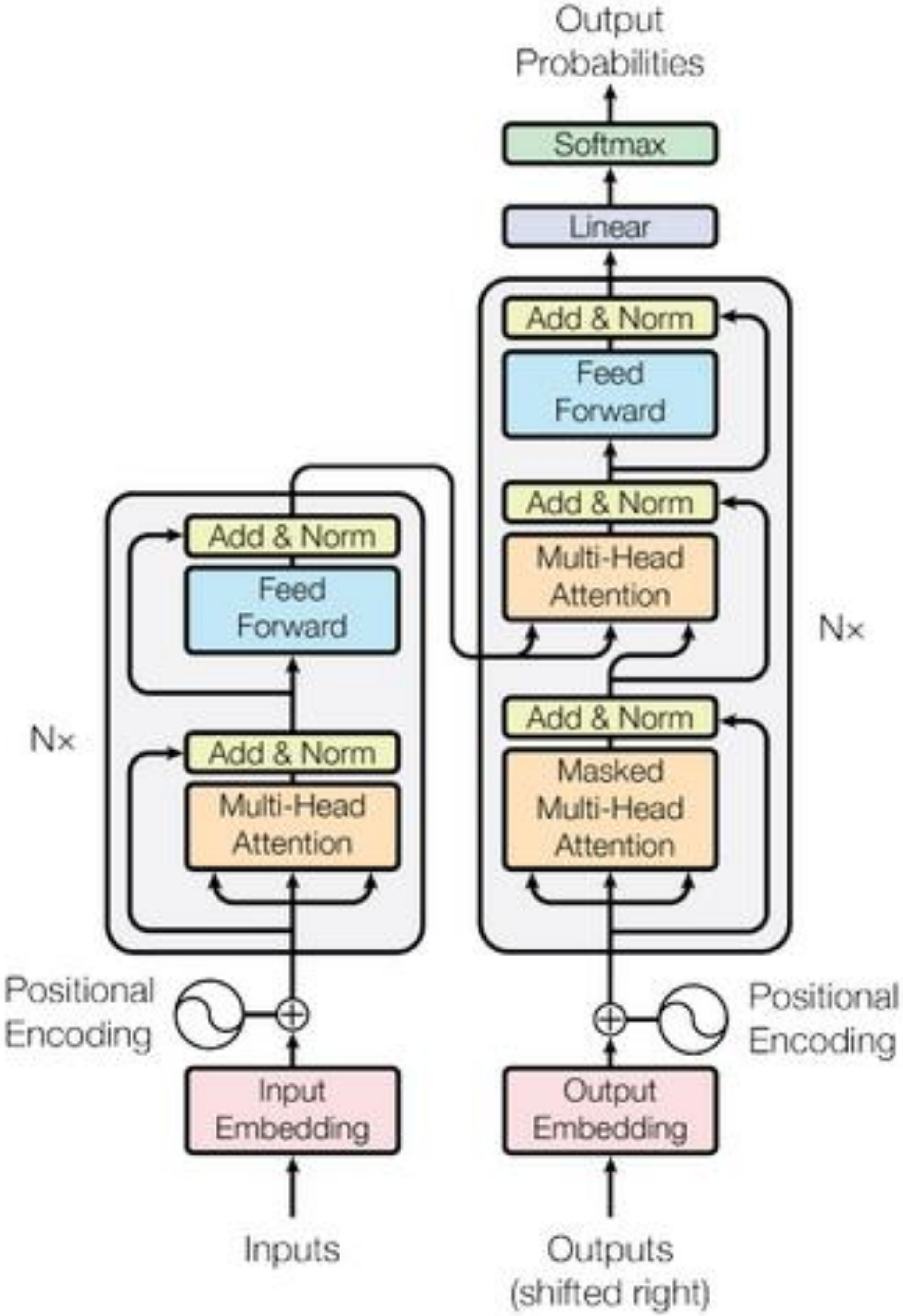


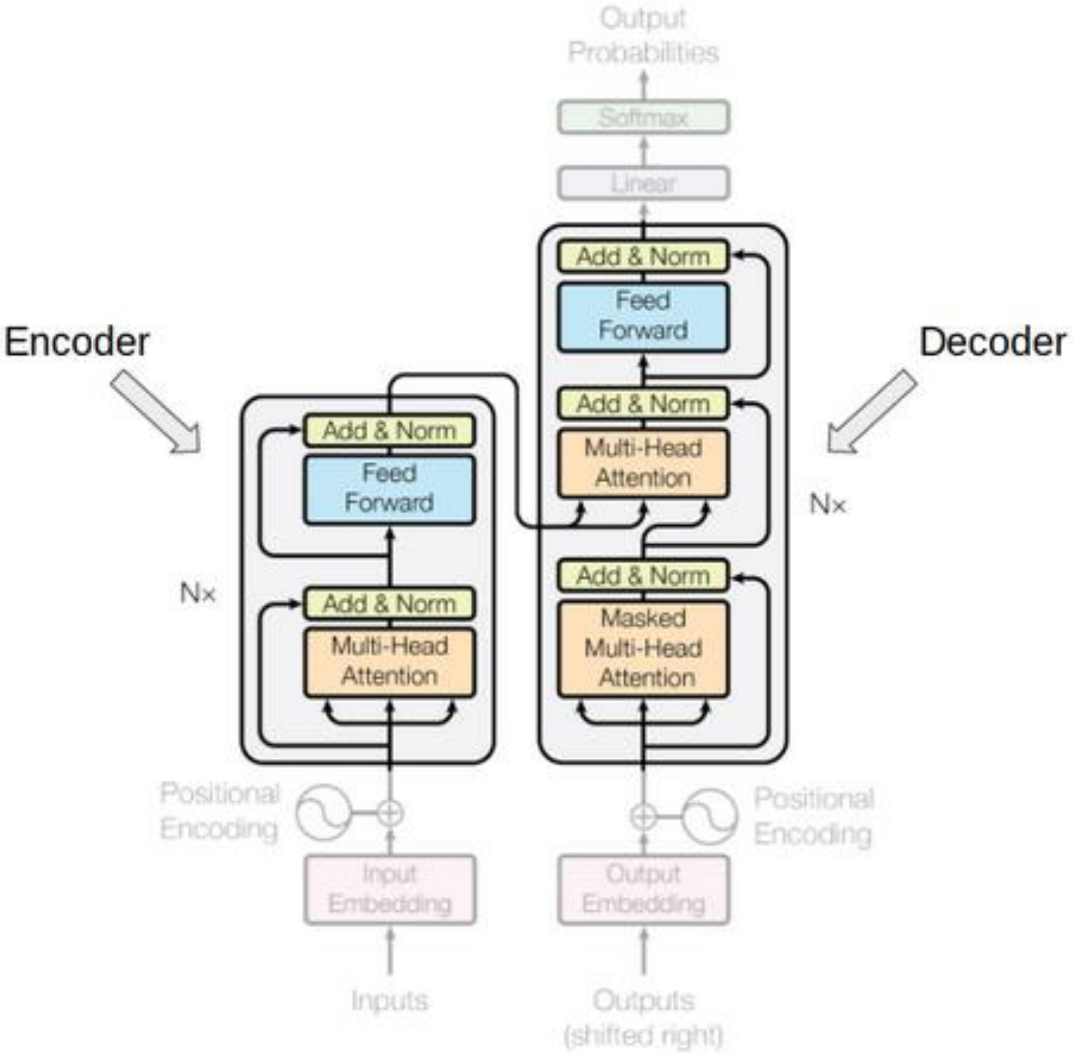
BLEU scores (higher is better) of single models on the standard WMT newstest2014 English to French translation benchmark.

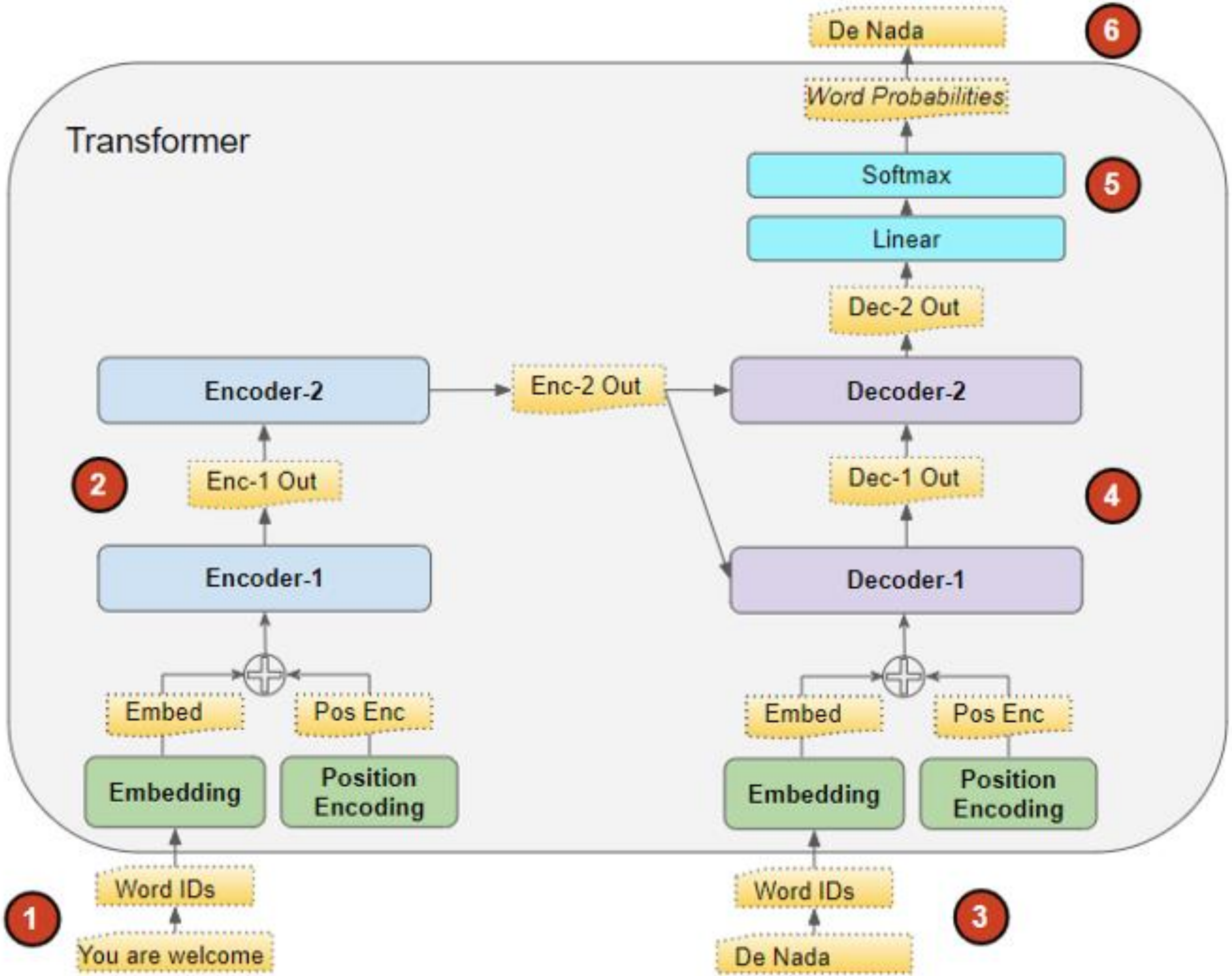
The animal didn't cross the street because it was too tired.
L'animal n'a pas traversé la rue parce qu'il était trop fatigué.

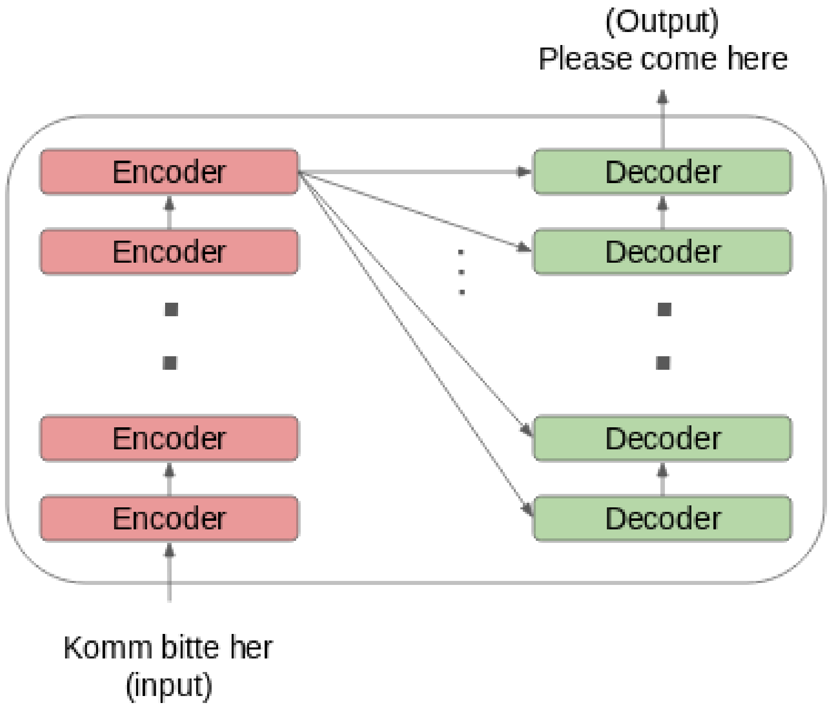
The animal didn't cross the street because it was too wide.
L'animal n'a pas traversé la rue parce qu'elle était trop large.

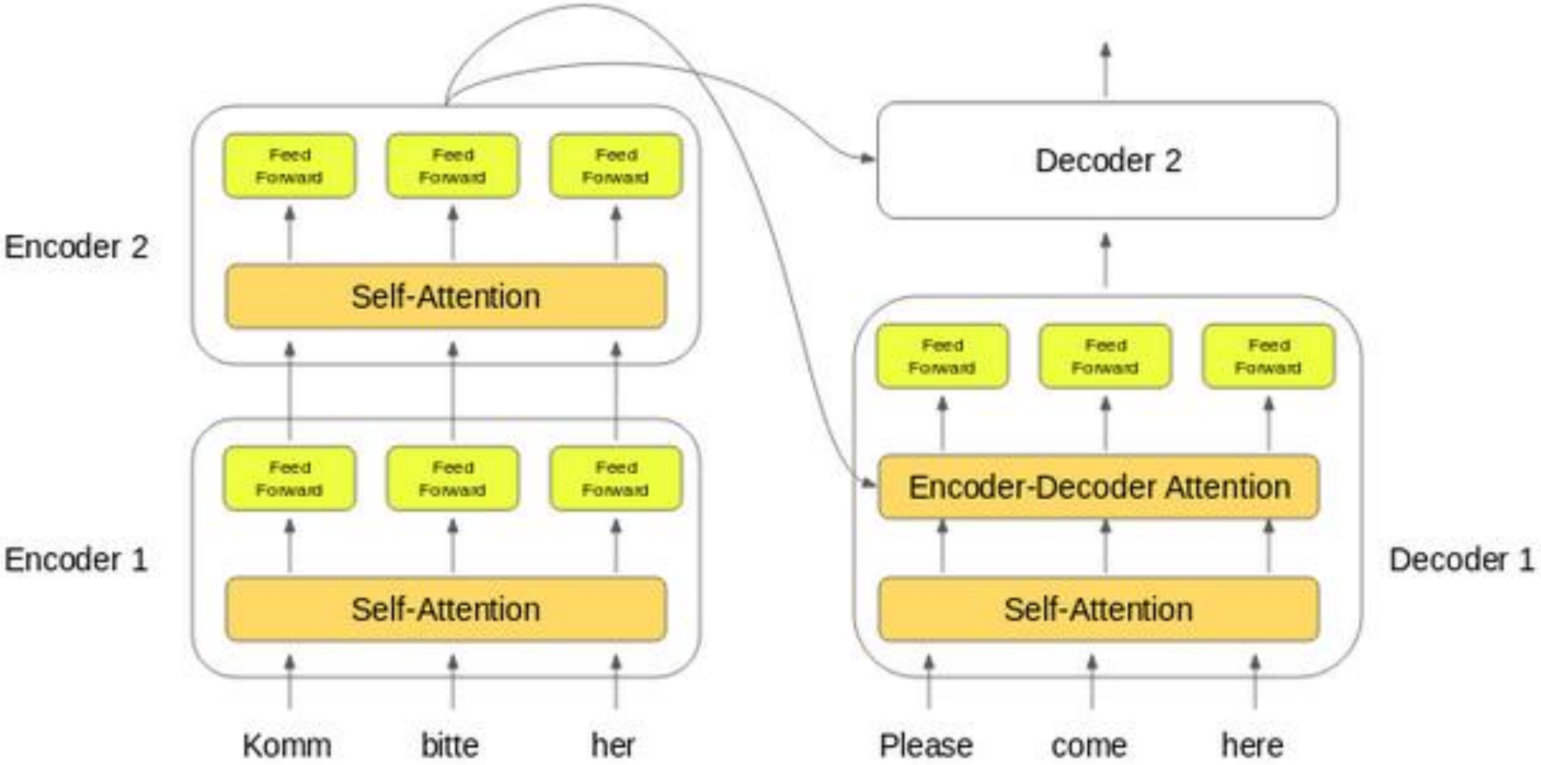

The animal didn't cross the street because it was too tired .
The animal didn't cross the street because it was too tired .
The animal didn't cross the street because it was too wide .
The animal didn't cross the street because it was too wide .


Output
Probabilities
Softmax
Linear
Add & Norm
Feed
Forward
Add & Norm
Multi-Head
Attention
Add & Norm
Masked
Multi-Head
Attention
Nx
Add & Norm
Feed
Forward
Add & Norm
Multi-Head
Attention
Nx
Positional
Encoding
Positional
Encoding
Input
Embedding
Output
Embedding
Inputs
Outputs
(shifted right)


Encoder
Decoder
Output
Probabilities
Softmax
Linear
Add & Norm
Feed
Forward
Add & Norm
Multi-Head
Attention
Add & Norm
Masked
Multi-Head
Attention
Nx
Add & Norm
Feed
Forward
Add & Norm
Multi-Head
Attention
Nx
Positional
Encoding
Positional
Encoding
Input
Embedding
Output
Embedding
Inputs
Outputs
(shifted right)


De Nada
6
Word Probabilities
Transformer
Softmax
5
Linear
Dec-2 Out
Encoder-2
Enc-2 Out
Decoder-2
2
Enc-1 Out
Dec-1 Out
4
Encoder-1
Decoder-1
Embed
Pos Enc
Embed
Pos Enc
Embedding
Position Encoding
Embedding
Position Encoding
Word IDs
Word IDs
1
3
You are welcome
De Nada


(Output)
Please come here
Encoder
Encoder
Encoder
Encoder
Decoder
Decoder
Decoder
Decoder
Komm bitte her
(input)

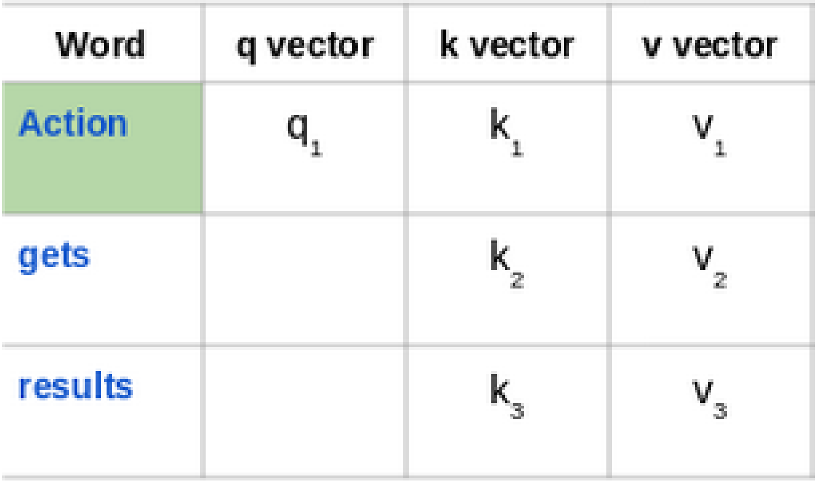

Encoder 2
Feed Forward
Feed Forward
Feed Forward
Self-Attention
Encoder 1
Feed Forward
Feed Forward
Feed Forward
Self-Attention
Komm
bitte
her
Decoder 2
Feed Forward
Feed Forward
Feed Forward
Encoder-Decoder Attention
Self-Attention
Decoder 1
Please
come
here

Self-Attention

- attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence.
- intra-attention

Calculating self-attention

1. create three vectors from each of the encoder’s input vectors:
    1. Query Vector
    2. Key Vector
    3. Value Vector
2. calculate self-attention for every word in the input sequence
    1. Example for “Action gets results.”



Word	q vector	k vector	v vector
Action	q_1	k_1	v_1
gets		k_2	v_2
results		k_3	v_3

Word	q vector	k vector	v vector	score
Action	q_1	k_1	v_1	$q_1 . k_1$
gets		k_2	v_2	$q_1 . k_2$
results		k_3	v_3	$q_1 . k_3$

Word	q vector	k vector	v vector	score	score / 8
Action	q_1	k_1	v_1	$q_1 . k_1$	$q_1 . k_1 / 8$
gets		k_2	v_2	$q_1 . k_2$	$q_1 . k_2 / 8$
results		k_3	v_3	$q_1 . k_3$	$q_1 . k_3 / 8$

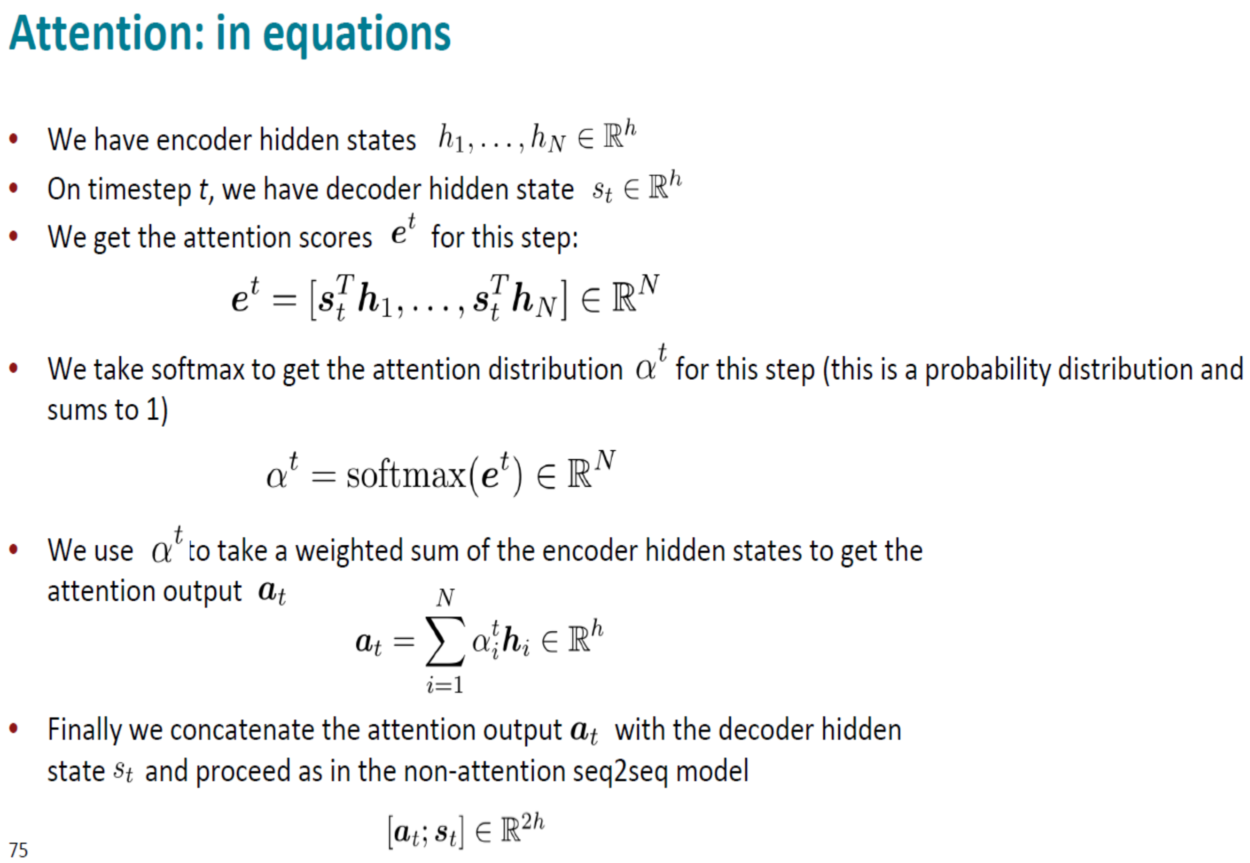
Word	q vector	k vector	v vector	score	score / 8	Softmax
Action	q_1	k_1	v_1	$q_1 . k_1$	$q_1 . k_1 / 8$	x_{11}
gets		k_2	v_2	$q_1 . k_2$	$q_1 . k_2 / 8$	x_{12}
results		k_3	v_3	$q_1 . k_3$	$q_1 . k_3 / 8$	x_{13}

Word	q vector	k vector	v vector	score	score / 8	Softmax	Softmax * v	Sum
Action	q_1	k_1	v_1	$q_1 . k_1$	$q_1 . k_1 / 8$	x_{11}	$x_{11} * v_1$	z_1
gets		k_2	v_2	$q_1 . k_2$	$q_1 . k_2 / 8$	x_{12}	$x_{12} * v_2$	
results		k_3	v_3	$q_1 . k_3$	$q_1 . k_3 / 8$	x_{13}	$x_{13} * v_3$	

Word	q vector	k vector	v vector	score	score / 8	Softmax	Softmax * v	Sum#
Action		k_1	v_1	$q_2 . k_1$	$q_2 . k_1 / 8$	x_{21}	$x_{21} * v_1$	
gets	q_2	k_2	v_2	$q_2 . k_2$	$q_2 . k_2 / 8$	x_{22}	$x_{22} * v_2$	z_2
results		k_3	v_3	$q_2 . k_3$	$q_2 . k_3 / 8$	x_{23}	$x_{23} * v_3$	

Word	q vector	k vector	v vector	score	score / 8	Softmax	Softmax * v	Sum#
Action		k_1	v_1	$q_3 . k_1$	$q_3 . k_1 / 8$	x_{31}	$x_{31} * v_1$	
gets		k_2	v_2	$q_3 . k_2$	$q_3 . k_2 / 8$	x_{32}	$x_{32} * v_2$	
results	q_3	k_3	v_3	$q_3 . k_3$	$q_3 . k_3 / 8$	x_{33}	$x_{33} * v_3$	z_3

Recap of Attention from Seq2Seq



Attention: in equations

- We have encoder hidden states $h_1, \ldots, h_N \in \mathbb{R}^h$
- On timestep *t*, we have decoder hidden state $s_t \in \mathbb{R}^h$
- We get the attention scores $\boldsymbol{e}^t$ for this step:

$$\boldsymbol{e}^t = [\boldsymbol{s}_t^T \boldsymbol{h}_1, \ldots, \boldsymbol{s}_t^T \boldsymbol{h}_N] \in \mathbb{R}^N$$

- We take softmax to get the attention distribution α^t for this step (this is a probability distribution and sums to 1)

$$\alpha^t = \text{softmax}(\boldsymbol{e}^t) \in \mathbb{R}^N$$

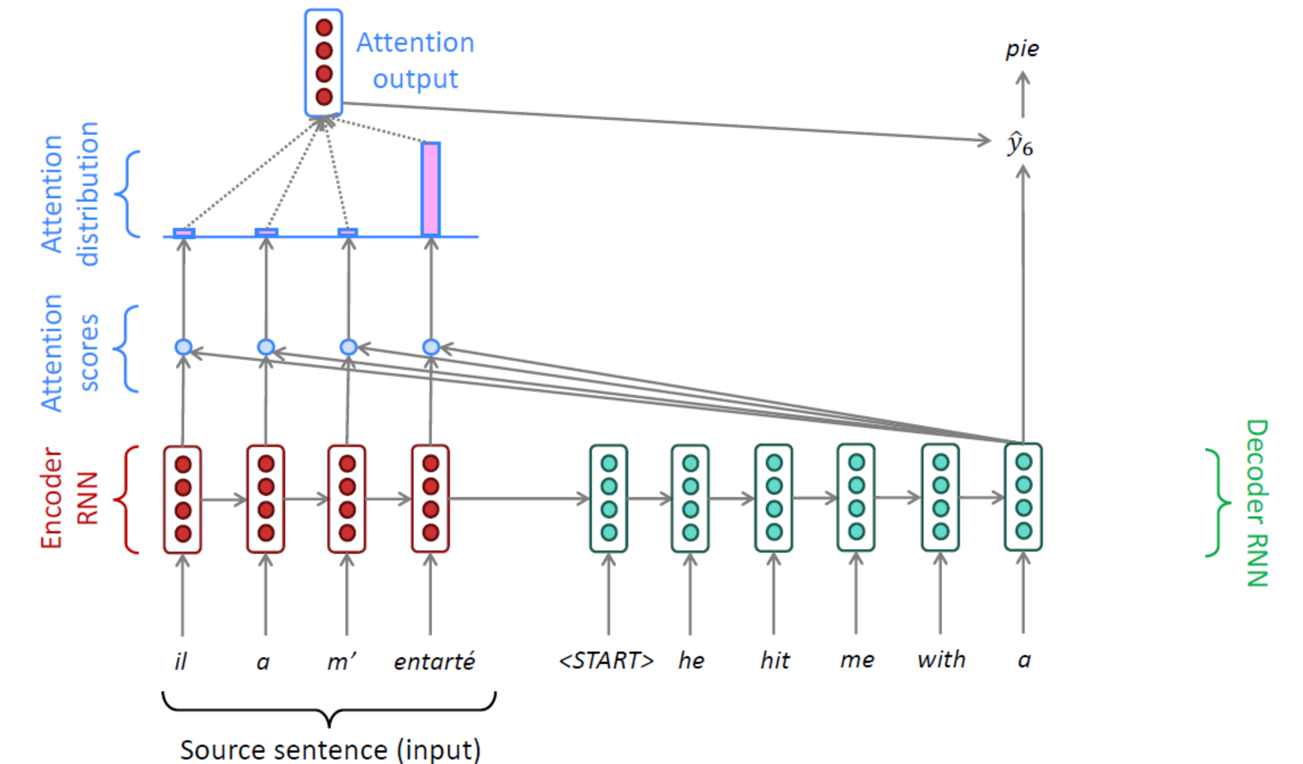
- We use α^t to take a weighted sum of the encoder hidden states to get the attention output $\boldsymbol{a}_t$

$$\boldsymbol{a}_t = \sum_{i=1}^{N} \alpha_i^t \boldsymbol{h}_i \in \mathbb{R}^h$$

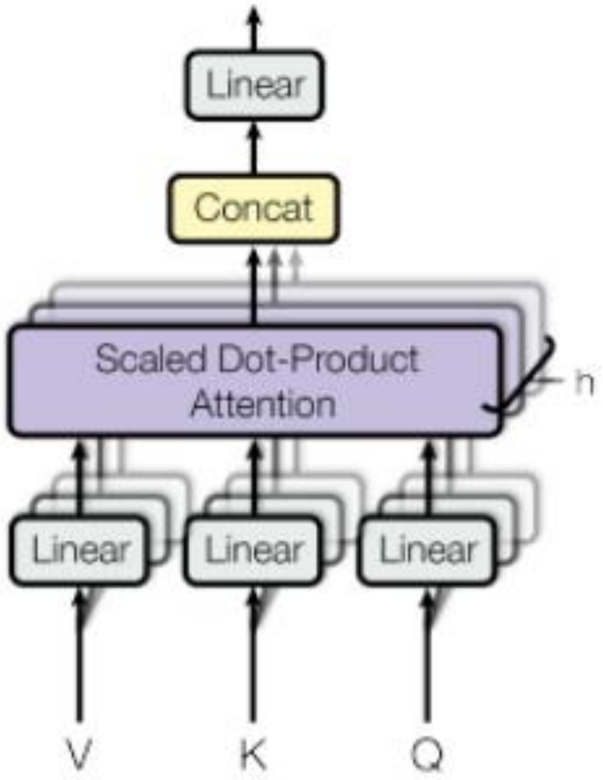
- Finally we concatenate the attention output $\boldsymbol{a}_t$ with the decoder hidden state s_t and proceed as in the non-attention seq2seq model

$$[\boldsymbol{a}_t; \boldsymbol{s}_t] \in \mathbb{R}^{2h}$$

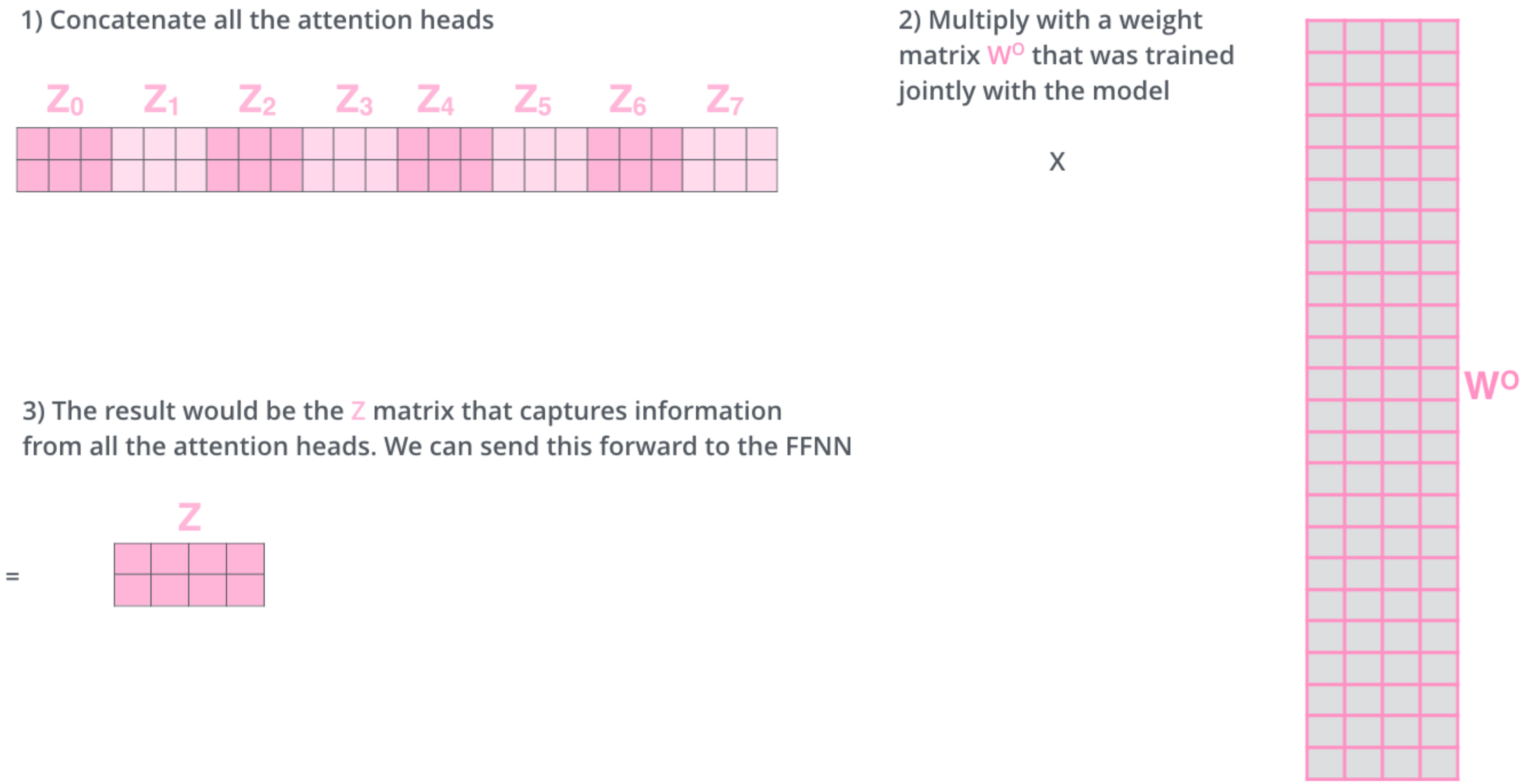
ence-to-sequence with attention



Multi-headed Attention



Multi-Head Attention



1) Concatenate all the attention heads

2) Multiply with a weight matrix W^O that was trained jointly with the model

3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN

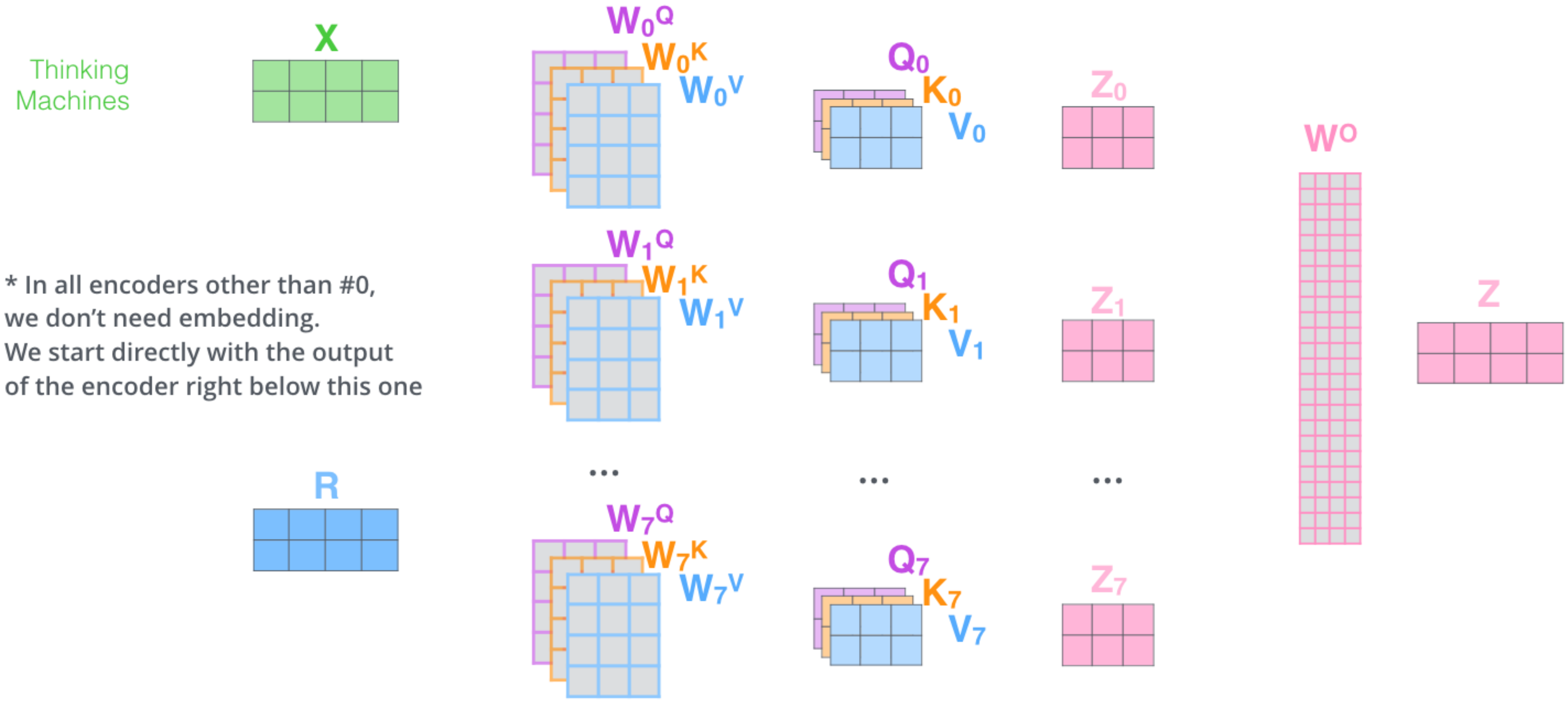
1) This is our input sentence*

2) We embed each word*

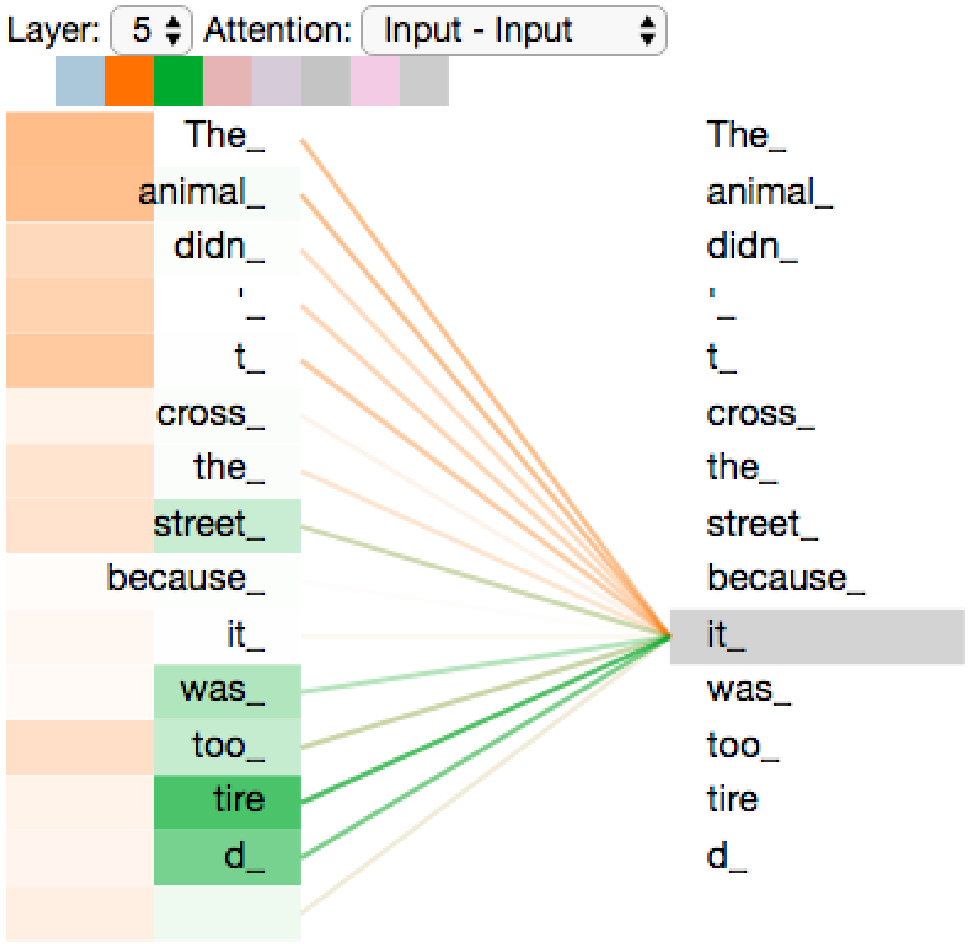
3) Split into 8 heads. We multiply X or R with weight matrices

4) Calculate attention using the resulting Q/K/V matrices

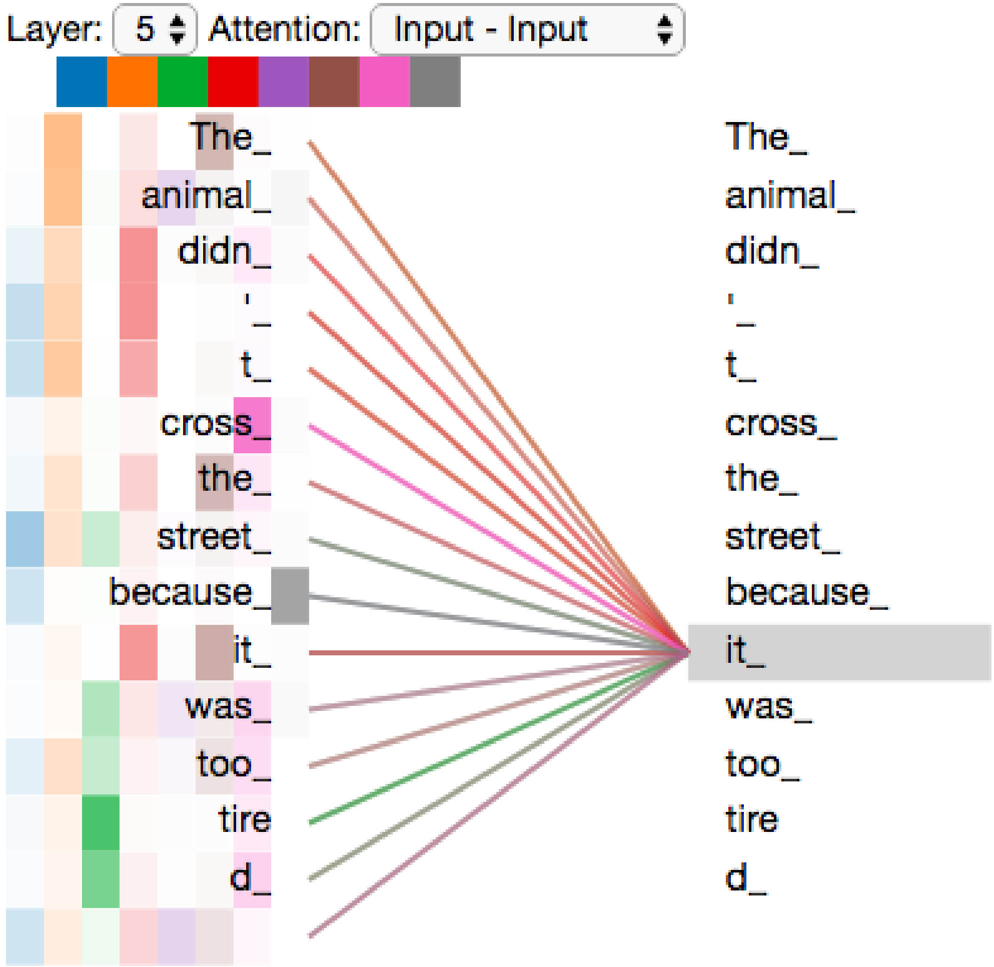
5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer



* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one

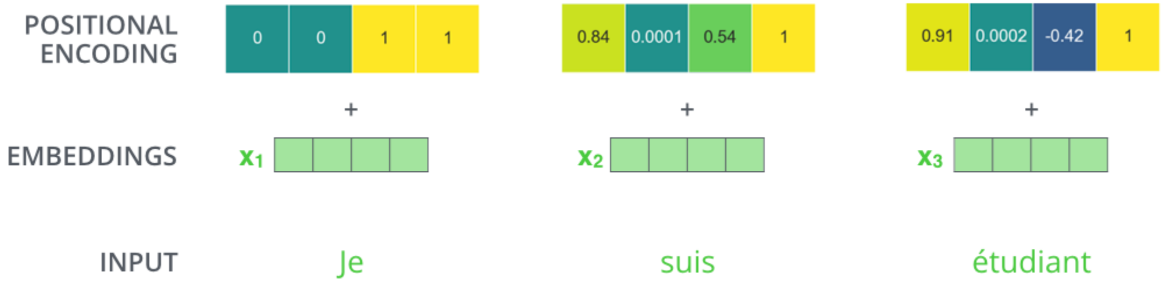


Layer: 5 Attention: Input - Input



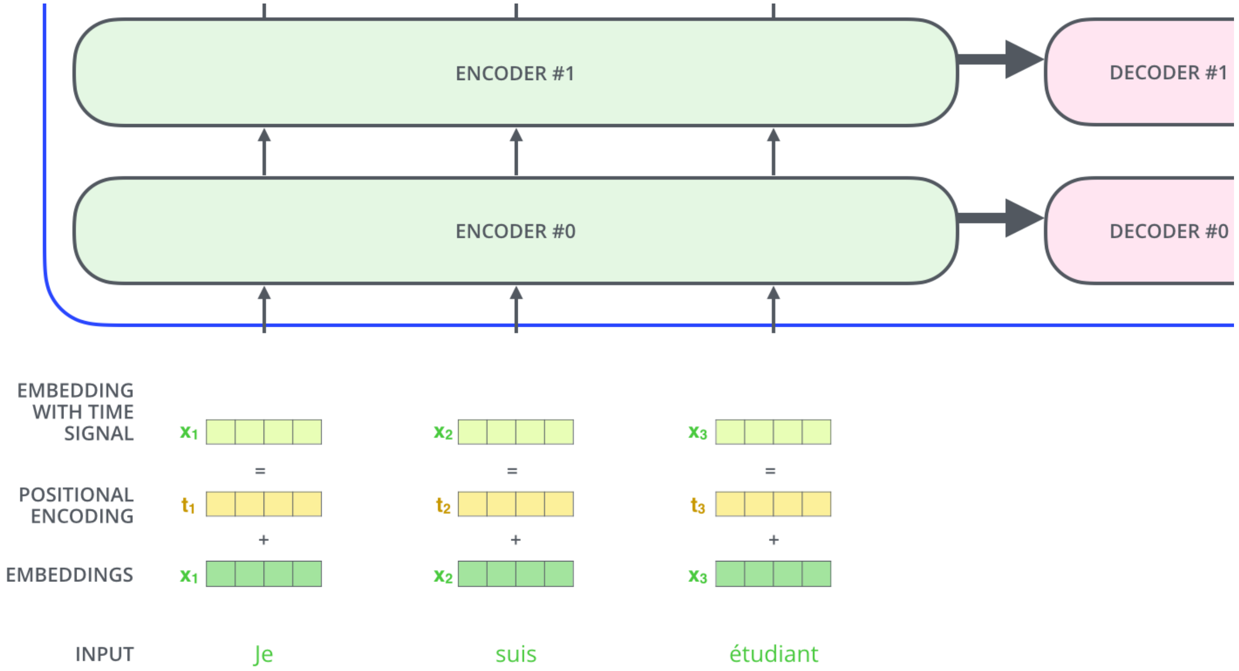
Layer: 5 Attention: Input - Input

Positional Encoding and Embedding



POSITIONAL ENCODING

	Je	suis	étudiant
POSITIONAL ENCODING	0 0 1 1	0.84 0.0001 0.54 1	0.91 0.0002 -0.42 1
	+	+	+
EMBEDDINGS	x_1	x_2	x_3
INPUT	Je	suis	étudiant



ENCODER #1

ENCODER #0

DECODER #1

DECODER #0

EMBEDDING WITH TIME SIGNAL x_1 x_2 x_3

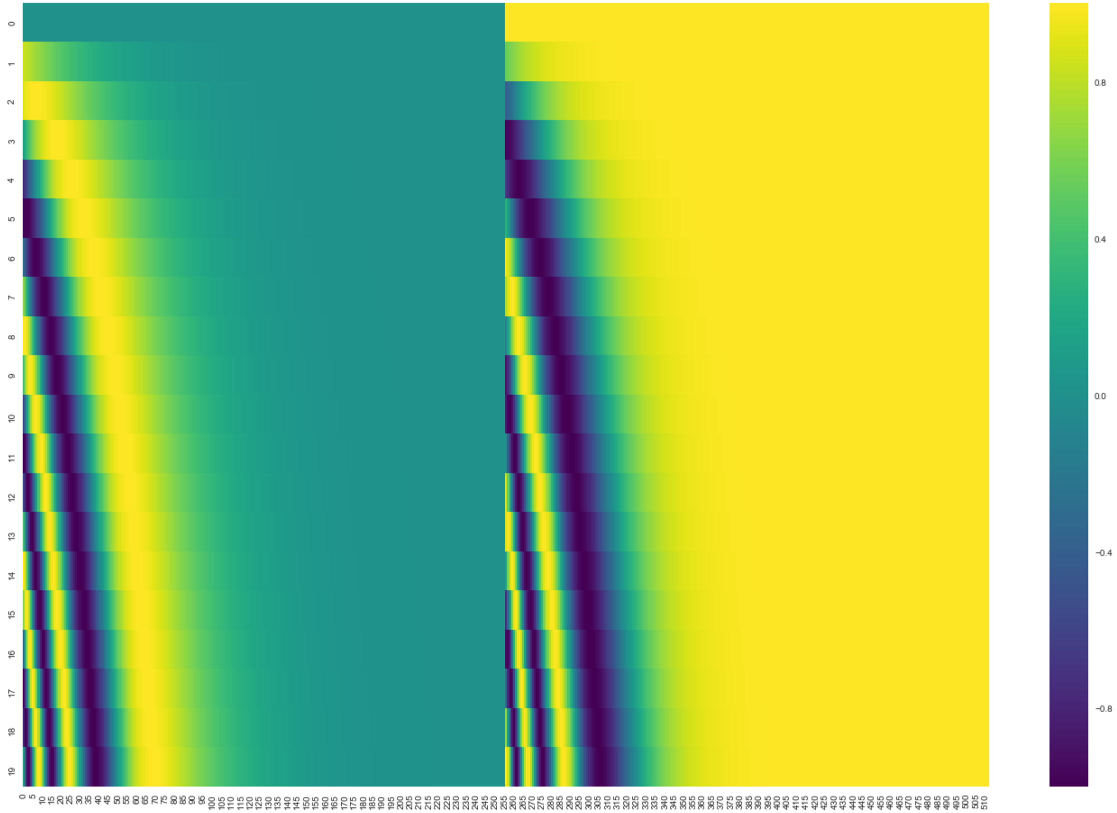
= = =

POSITIONAL ENCODING t_1 t_2 t_3

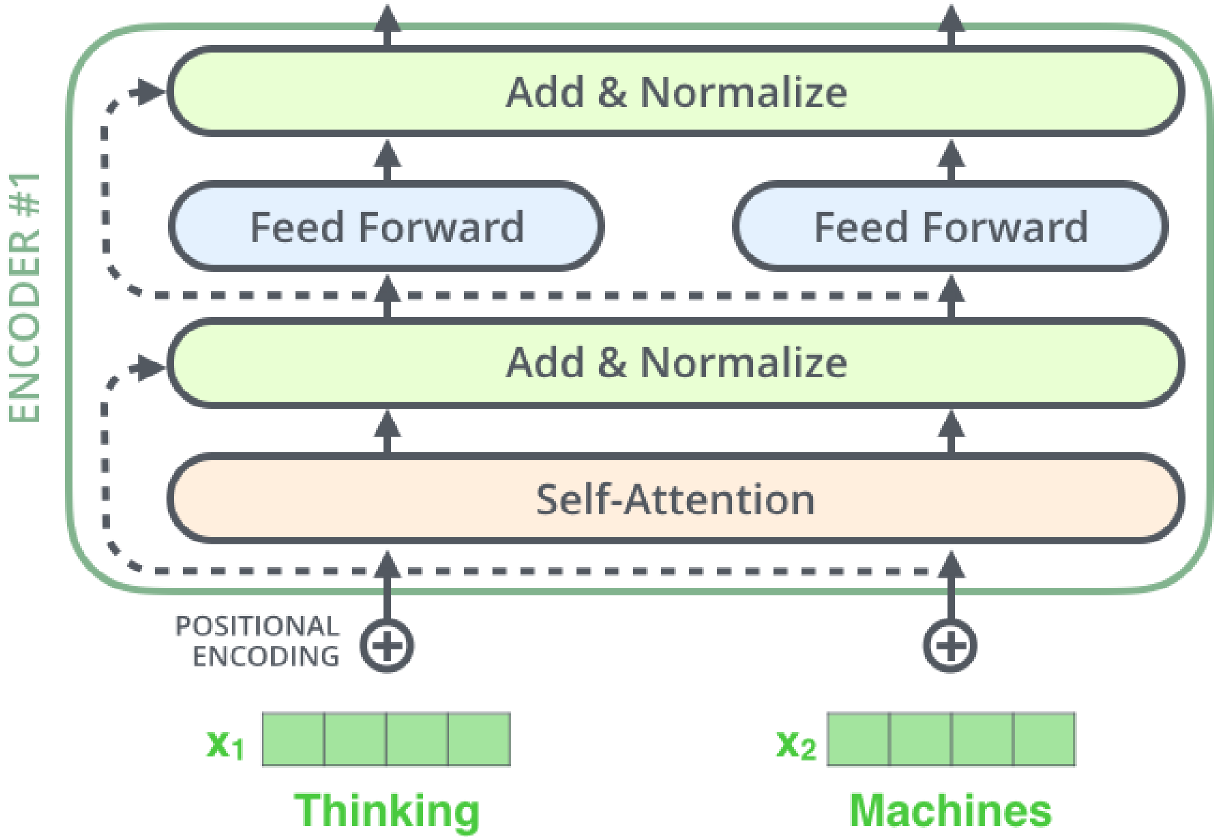
+ + +

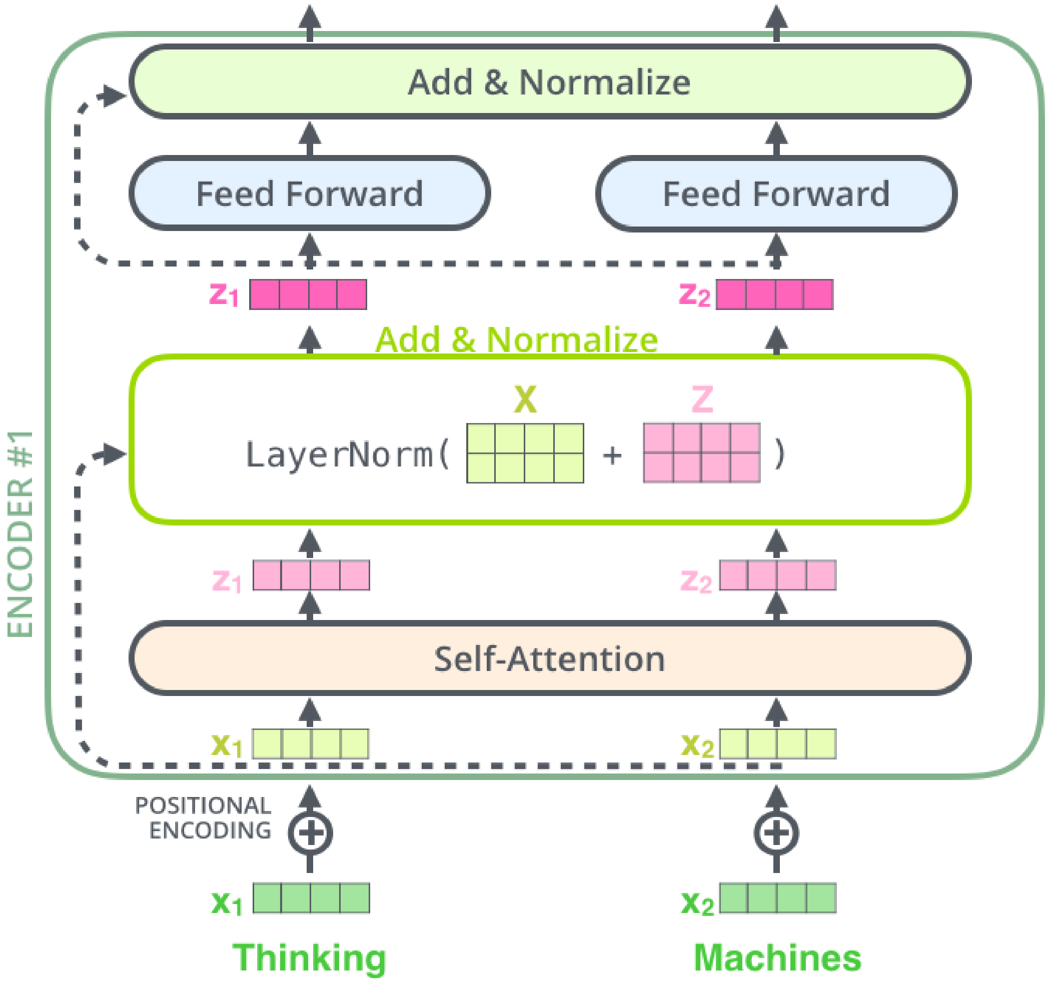
EMBEDDINGS x_1 x_2 x_3

INPUT Je suis étudiant

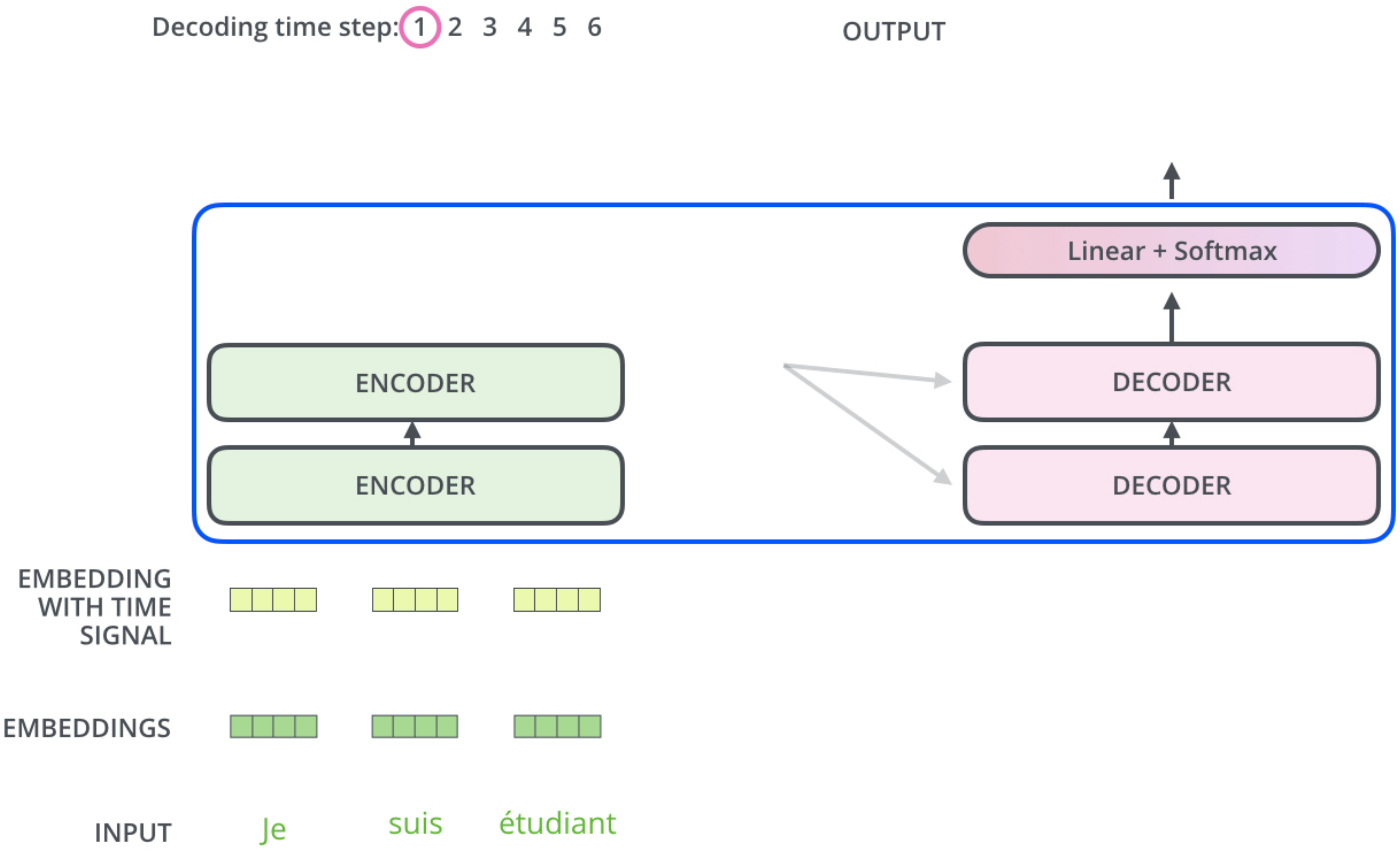


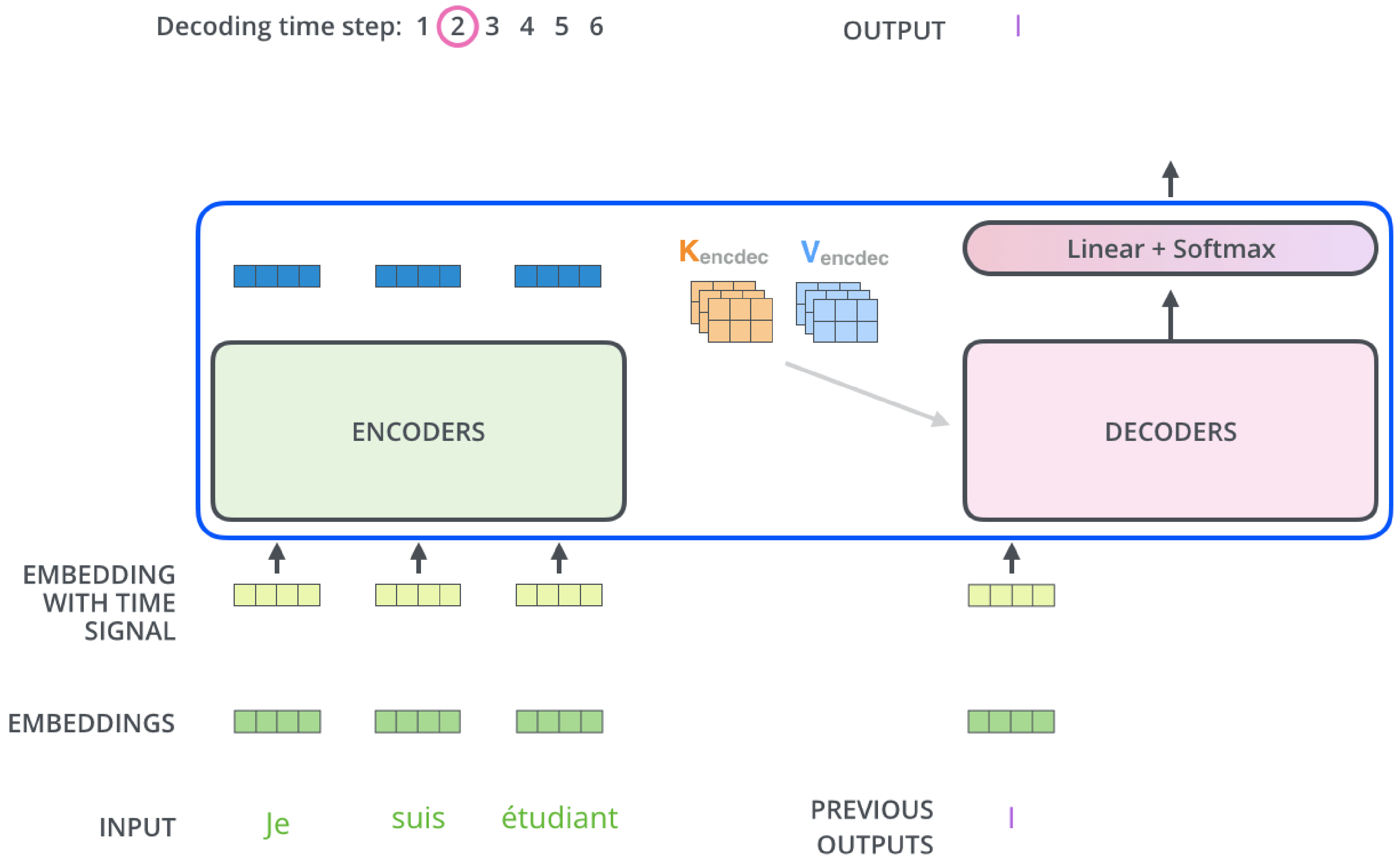
Residuals

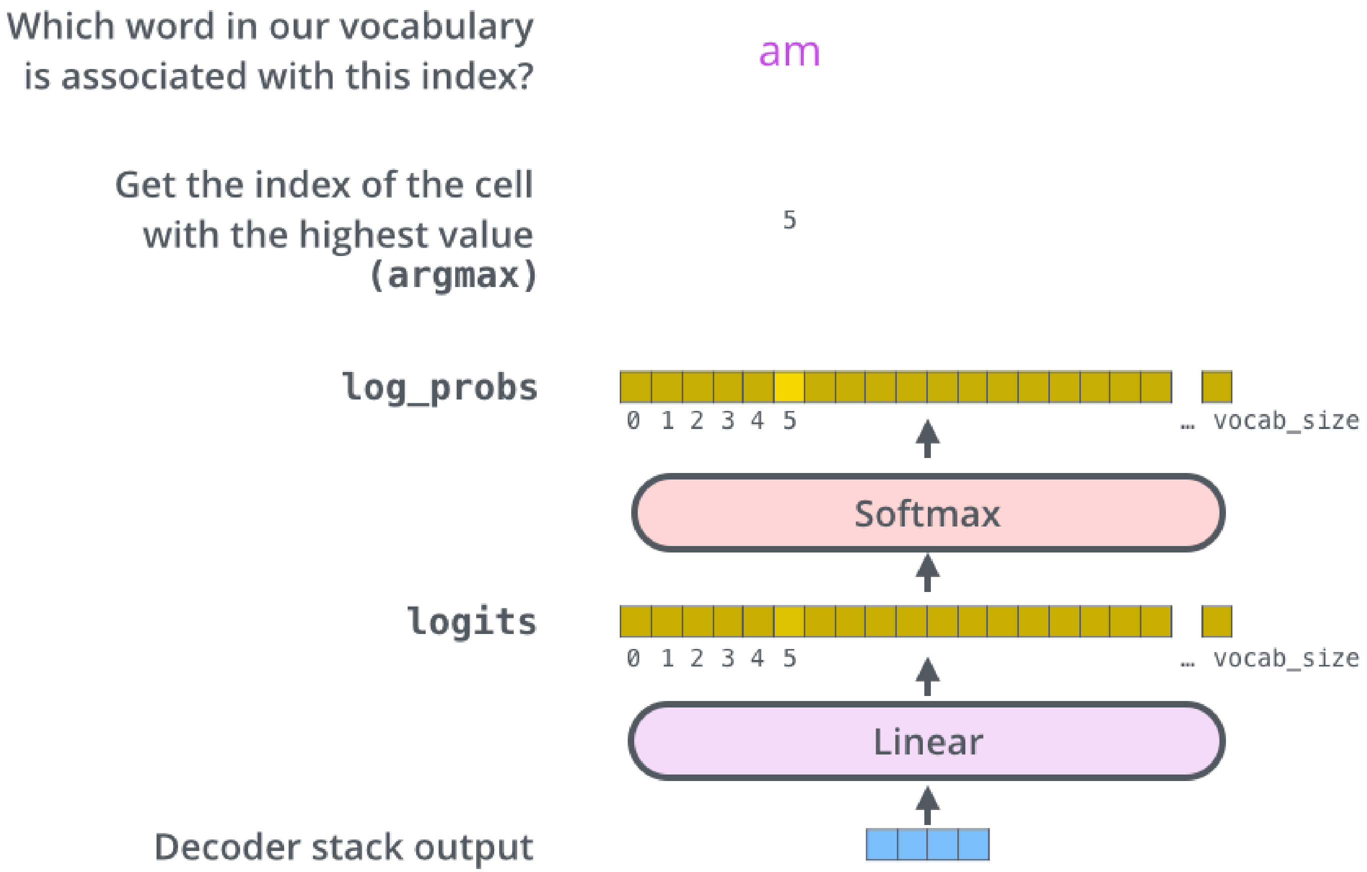

ENCODER #1
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Self-Attention
POSITIONAL ENCODING
x_1
x_2
Thinking
Machines


ENCODER #1
Add & Normalize
Feed Forward
Feed Forward
z_1
z_2
Add & Normalize
X
Z
LayerNorm(+)
z_1
z_2
Self-Attention
x_1
x_2
POSITIONAL ENCODING
x_1
x_2
Thinking
Machines

Decoder


Decoding time step: 1 2 3 4 5 6
OUTPUT
Linear + Softmax
ENCODER
ENCODER
DECODER
DECODER
EMBEDDING WITH TIME SIGNAL
EMBEDDINGS
INPUT
Je
suis
étudiant


Decoding time step: 1 2 3 4 5 6
OUTPUT
I
Kencdec
Vencdec
Linear + Softmax
ENCODERS
DECODERS
EMBEDDING WITH TIME SIGNAL
EMBEDDINGS
INPUT
Je
suis
étudiant
PREVIOUS OUTPUTS
I



Which word in our vocabulary is associated with this index?

am

Get the index of the cell with the highest value **(argmax)**

5

`log_probs`

`0 1 2 3 4 5 … vocab_size`

Softmax

`logits`

`0 1 2 3 4 5 … vocab_size`

Linear

Decoder stack output

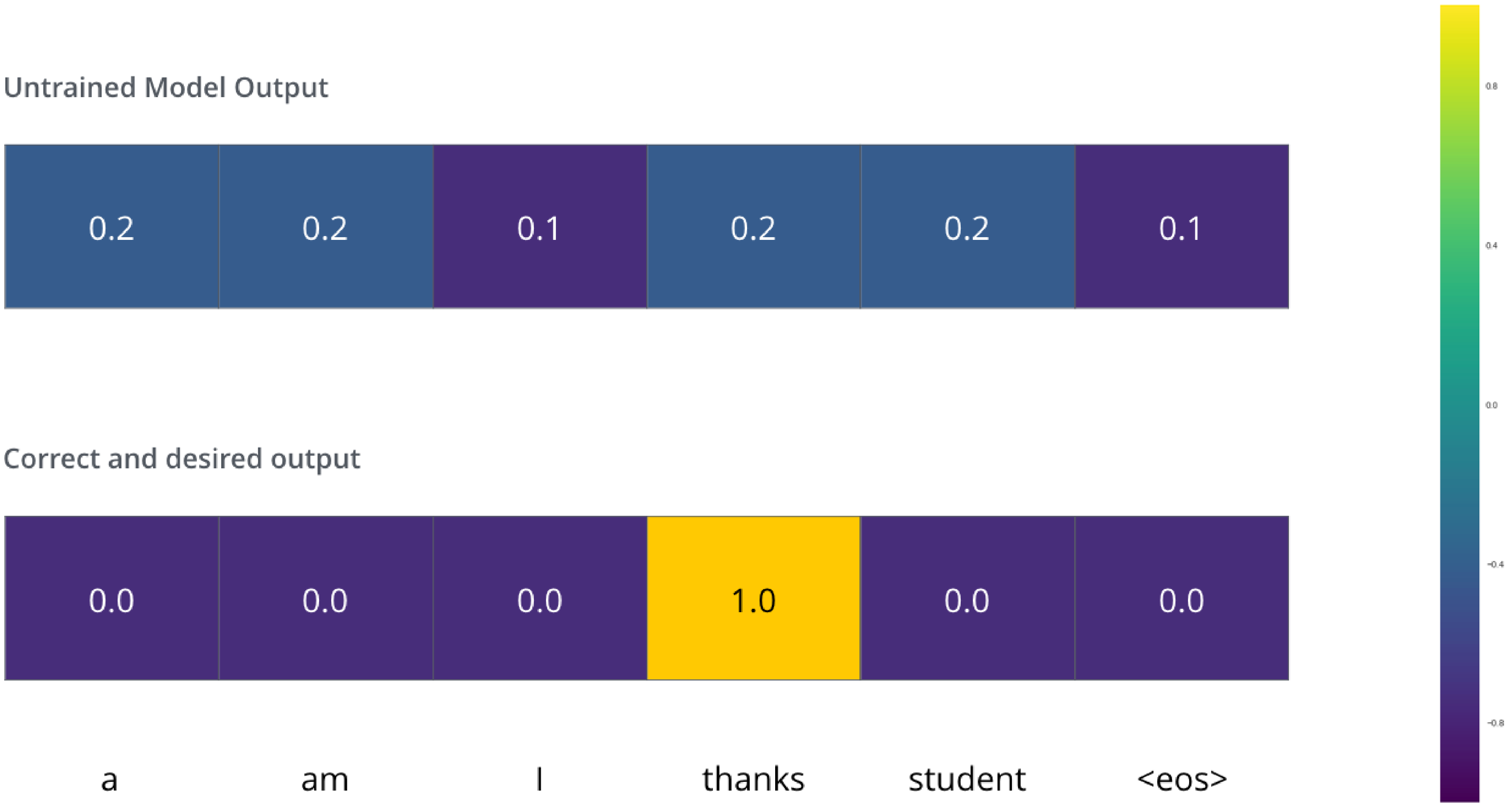
Training Details

Output Vocabulary

WORD	a	am	I	thanks	student	<eos>
INDEX	0	1	2	3	4	5

One-hot encoding of the word “am”

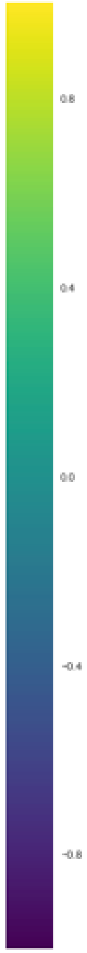
0.0	1.0	0.0	0.0	0.0	0.0

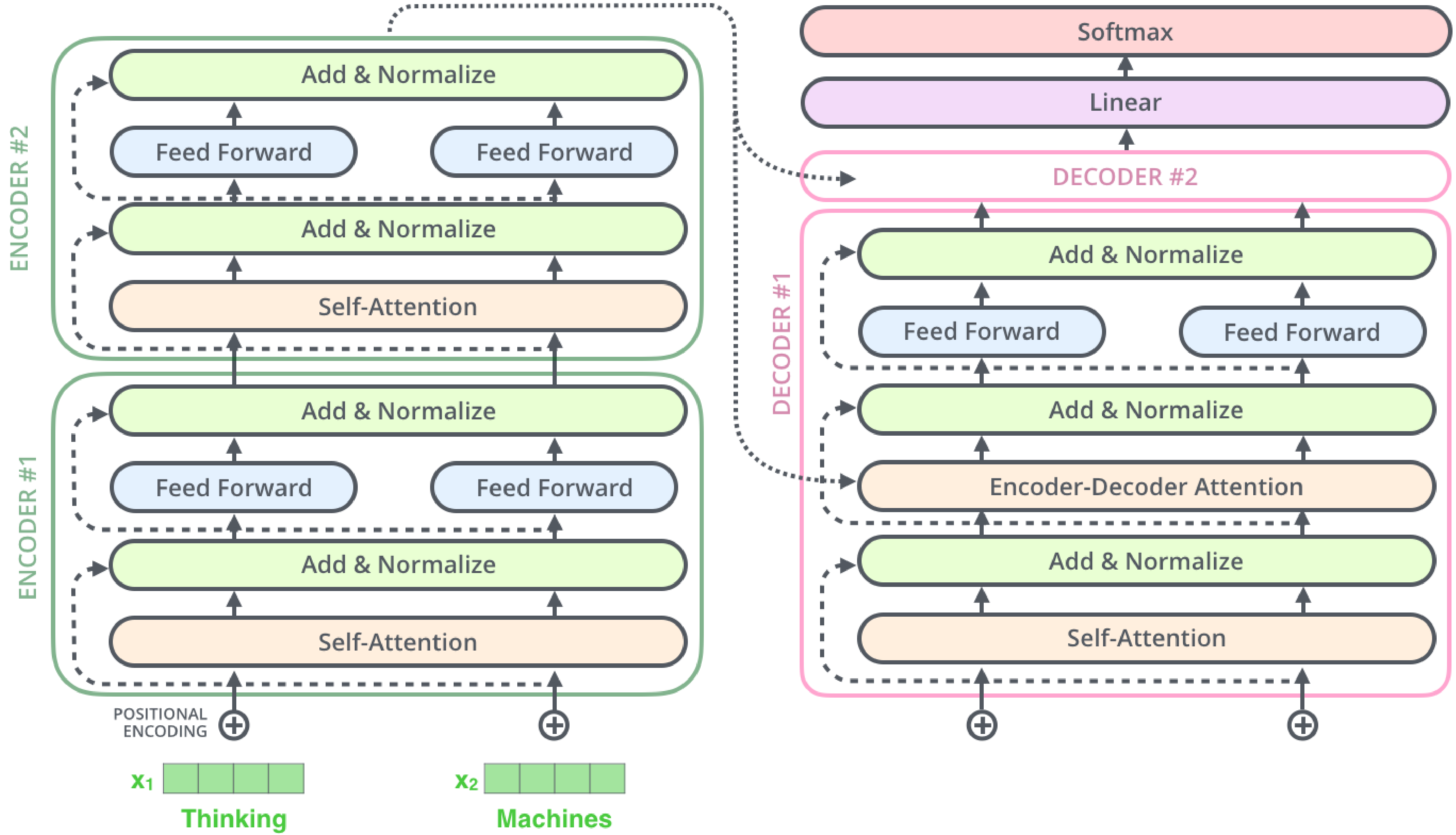


	a	am	I	thanks	student	<eos>
Untrained Model Output	0.2	0.2	0.1	0.2	0.2	0.1
Correct and desired output	0.0	0.0	0.0	1.0	0.0	0.0

Trained Model Outputs

Output Vocabulary:	a	am	I	thanks	student	<eos>
position #1	0.01	0.02	0.93	0.01	0.03	0.01
position #2	0.01	0.8	0.1	0.05	0.01	0.03
position #3	0.99	0.001	0.001	0.001	0.002	0.001
position #4	0.001	0.002	0.001	0.02	0.94	0.01
position #5	0.01	0.01	0.001	0.001	0.001	0.98
	a	am	I	thanks	student	<eos>





Softmax
Linear
DECODER #2
DECODER #1
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Encoder-Decoder Attention
Add & Normalize
Self-Attention
ENCODER #2
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Self-Attention
ENCODER #1
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Self-Attention
POSITIONAL ENCODING
x1
x2
Thinking
Machines

Limitations of Transformers

- Attention can only deal with fixed-length text strings:
  - The text has to be split into a certain number of segments before being fed into the system as input.
- This chunking of text causes context fragmentation:
  - E.g., if a sentence is split from the middle, then a significant amount of context is lost.
  - In other words, the text is split without respecting the sentence or any other semantic boundary

Transformers and Self-Attention: Mathematical Example

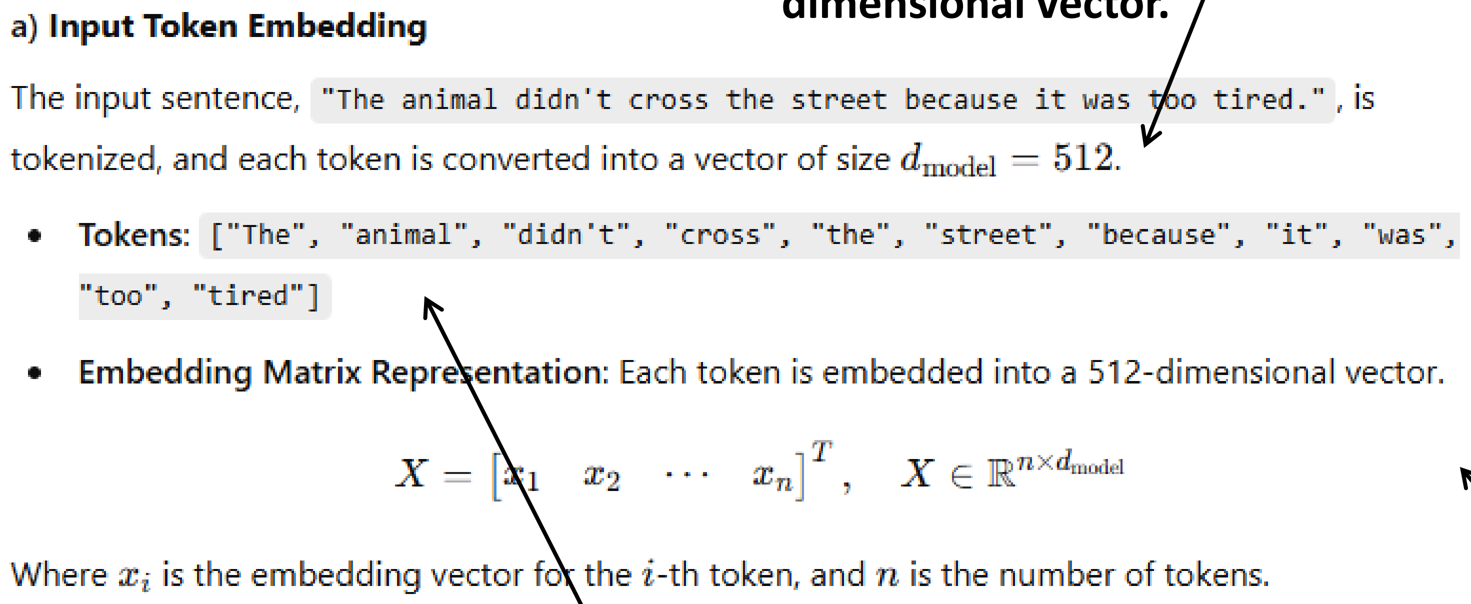
- Input Embedding and Positional Encoding
- Multi-Head Self-Attention (Encoder)
- Feed Forward Network (Encoder)
- Masked Multi-Head Attention (Decoder)Encoder-Decoder Attention
- Final Linear Layer and Softmax

Transformers and Self-Attention: Mathematical Example

- For the given sentence
  - Input: "The animal didn't cross the street because it was too tired.“
  - Output: "L'animal n'a pas traversé la rue parce qu'il était trop fatigué."

Step 1: Input Embedding and Positional Encoding

Each word or token in the sentence gets **its own 512-dimensional vector.**



a) Input Token Embedding

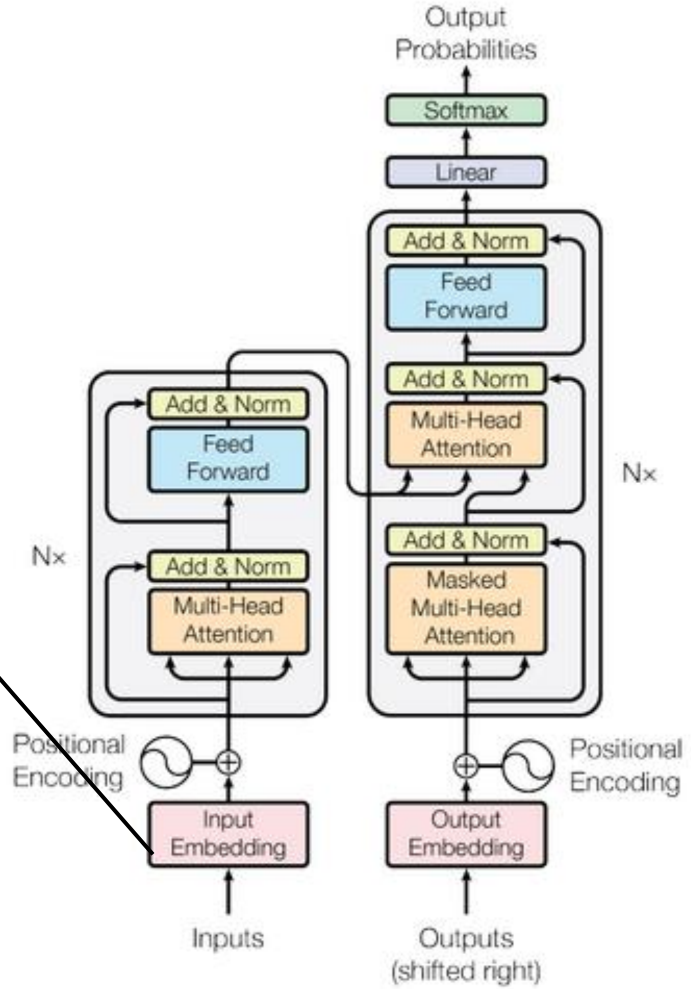
The input sentence, `"The animal didn't cross the street because it was too tired."`, is tokenized, and each token is converted into a vector of size $d_{\text{model}} = 512$.

- **Tokens:** `["The", "animal", "didn't", "cross", "the", "street", "because", "it", "was", "too", "tired"]`
- **Embedding Matrix Representation:** Each token is embedded into a 512-dimensional vector.

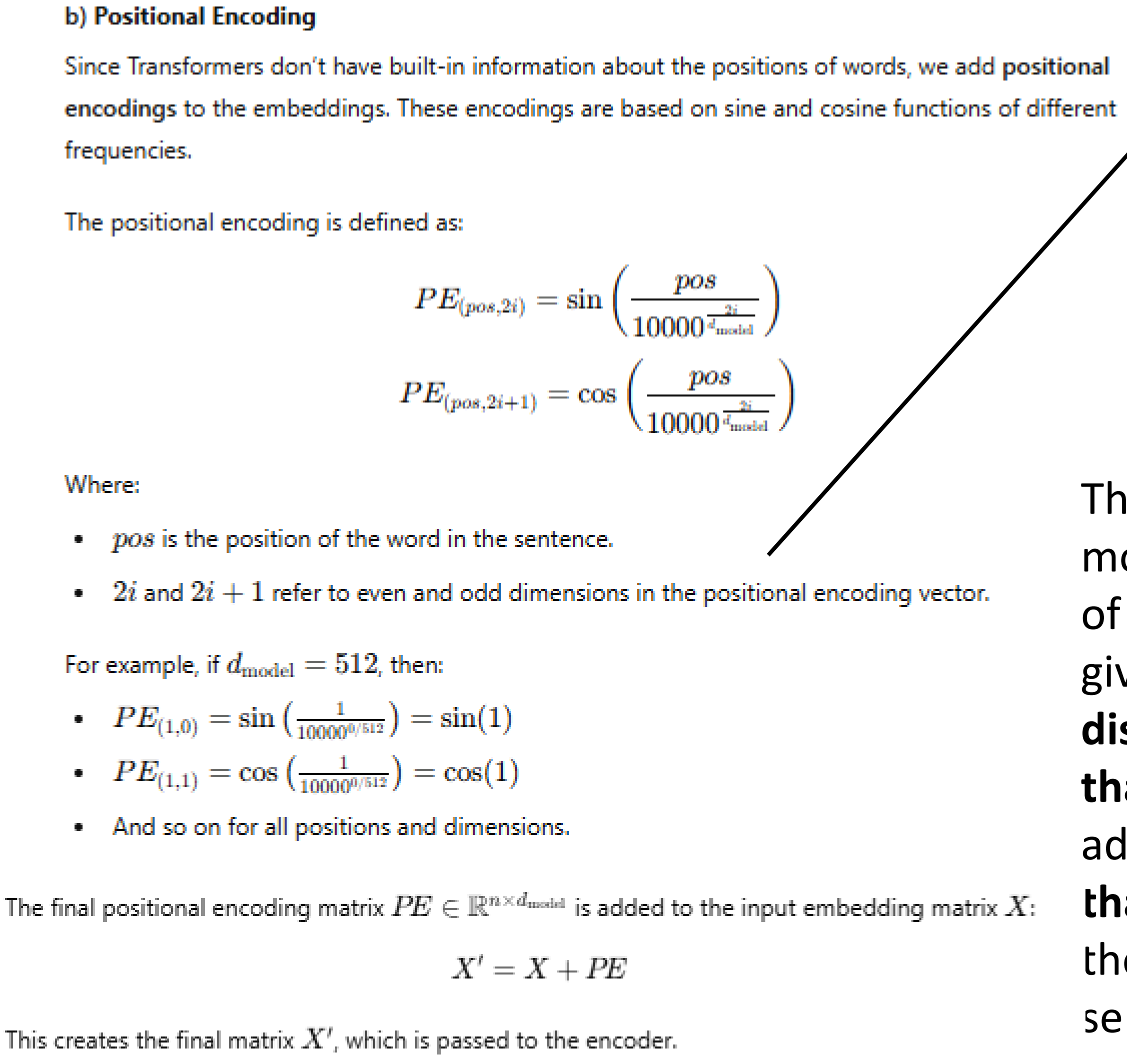
$$X = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}^T, \quad X \in \mathbb{R}^{n \times d_{\text{model}}}$$

Where x_i is the embedding vector for the i-th token, and n is the number of tokens.

So, if the sentence has 10 tokens, you will have 10 vectors, each of size 512.



Step 1: Input Embedding and Positional Encoding



b) Positional Encoding

Since Transformers don't have built-in information about the positions of words, we add **positional encodings** to the embeddings. These encodings are based on sine and cosine functions of different frequencies.

The positional encoding is defined as:

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

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

Where:

- pos is the position of the word in the sentence.
- $2i$ and $2i+1$ refer to even and odd dimensions in the positional encoding vector.

For example, if $d_{\text{model}} = 512$, then:

- $PE_{(1,0)} = \sin\left(\frac{1}{10000^{0/512}}\right) = \sin(1)$
- $PE_{(1,1)} = \cos\left(\frac{1}{10000^{0/512}}\right) = \cos(1)$
- And so on for all positions and dimensions.

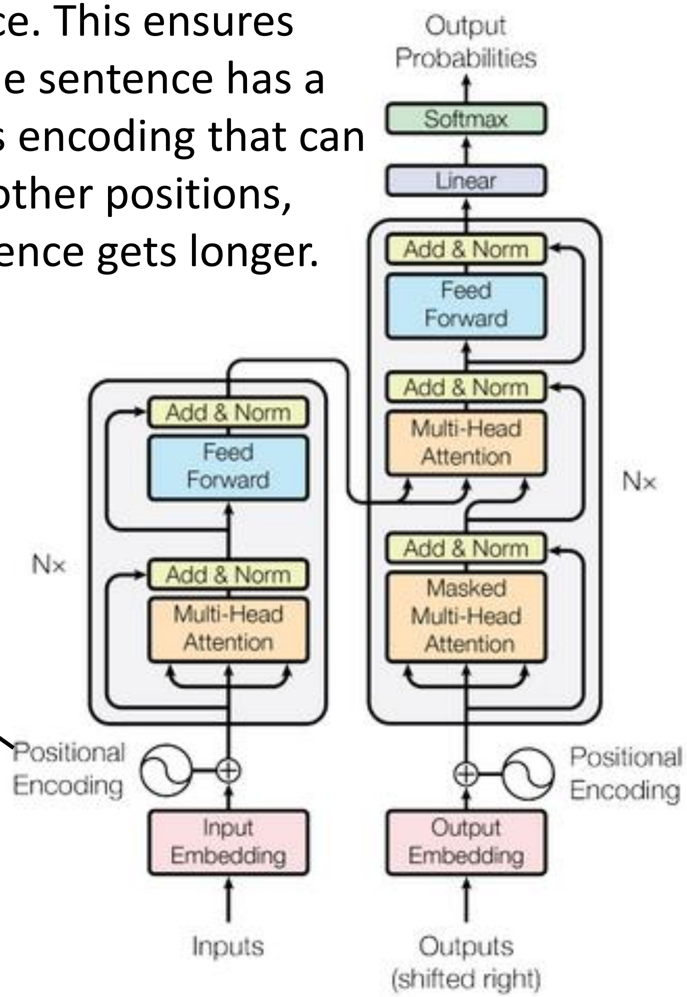
The final positional encoding matrix $PE \in \mathbb{R}^{n \times d_{\text{model}}}$ is added to the input embedding matrix X:

$$X' = X + PE$$

This creates the final matrix X', which is passed to the encoder.

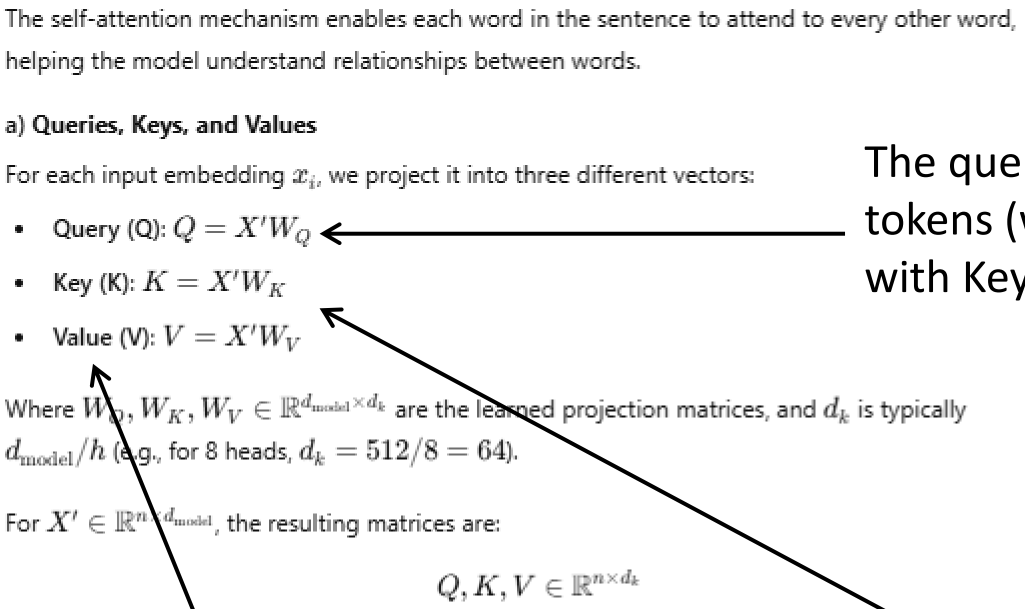
By alternating between sine and cosine functions, the model can create a **rich and unique representation** for each position in the sequence. This ensures that each position in the sentence has a distinct and continuous encoding that can be distinguished from other positions, even as the input sequence gets longer.

The even-odd split allows the model to handle a wider range of positions effectively and gives it the ability to **distinguish between positions that are close together** (like adjacent words) and **positions that are far apart** (words at the beginning and end of a sentence).



2. Multi-Head Self-Attention (Encoder)

https://aibyhand.substack.com/p/11-can-you-calculate-self-attention



The self-attention mechanism enables each word in the sentence to attend to every other word, helping the model understand relationships between words.

a) Queries, Keys, and Values

For each input embedding x_i, we project it into three different vectors:

- **Query (Q):** $Q = X'W_Q$
- **Key (K):** $K = X'W_K$
- **Value (V):** $V = X'W_V$

Where $W_Q, W_K, W_V \in \mathbb{R}^{d_{\text{model}} \times d_k}$ are the learned projection matrices, and d_k is typically d_{model}/h (e.g., for 8 heads, $d_k = 512/8 = 64$).

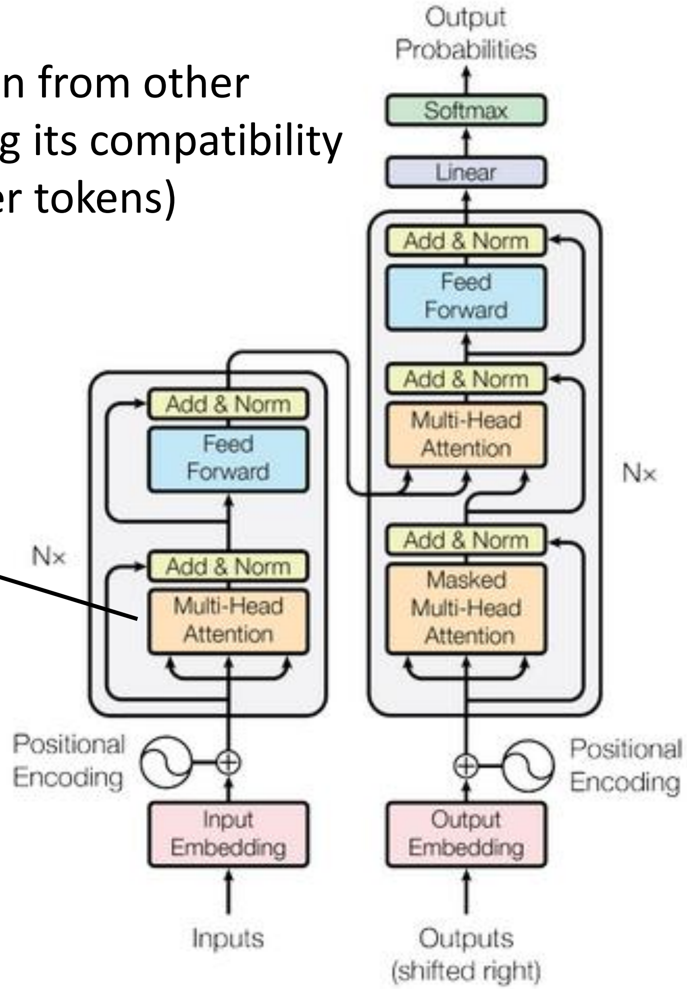
For $X' \in \mathbb{R}^{n \times d_{\text{model}}}$, the resulting matrices are:

$$Q, K, V \in \mathbb{R}^{n \times d_k}$$

The query is used to find relevant information from other tokens (words) in the sequence by computing its compatibility with Keys (which are representations of other tokens)

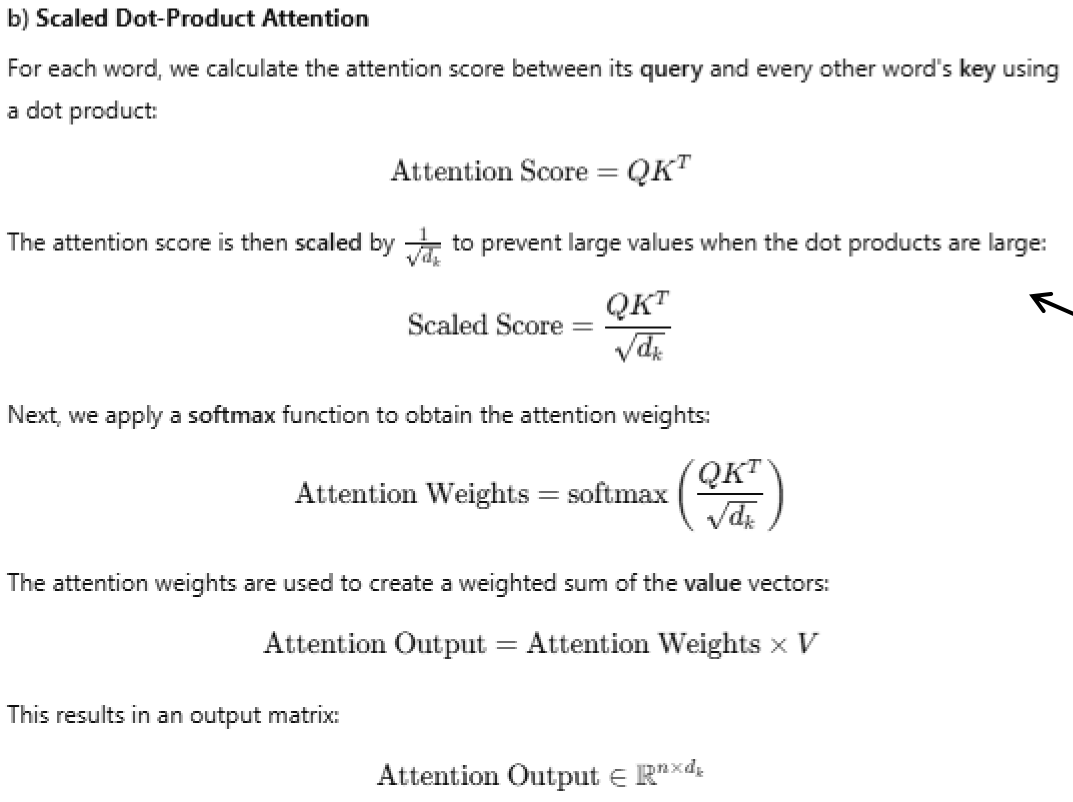
The Value vector contains the word’s representation and is used to compute the final weighted output after attention scores are applied. The Values are what get "returned" after the attention process decides which tokens to focus on.

Each word in the sequence has a Key, which is compared to Queries from other tokens to compute attention scores. The Key holds information about a word and helps decide how important this word is when processing the current word (Query).



2. Multi-Head Self-Attention (Encoder)

https://aibyhand.substack.com/p/11-can-you-calculate-self-attention



b) Scaled Dot-Product Attention

For each word, we calculate the attention score between its **query** and every other word's **key** using a dot product:

$$\text{Attention Score} = QK^T$$

The attention score is then **scaled** by $\frac{1}{\sqrt{d_k}}$ to prevent large values when the dot products are large:

$$\text{Scaled Score} = \frac{QK^T}{\sqrt{d_k}}$$

Next, we apply a **softmax** function to obtain the attention weights:

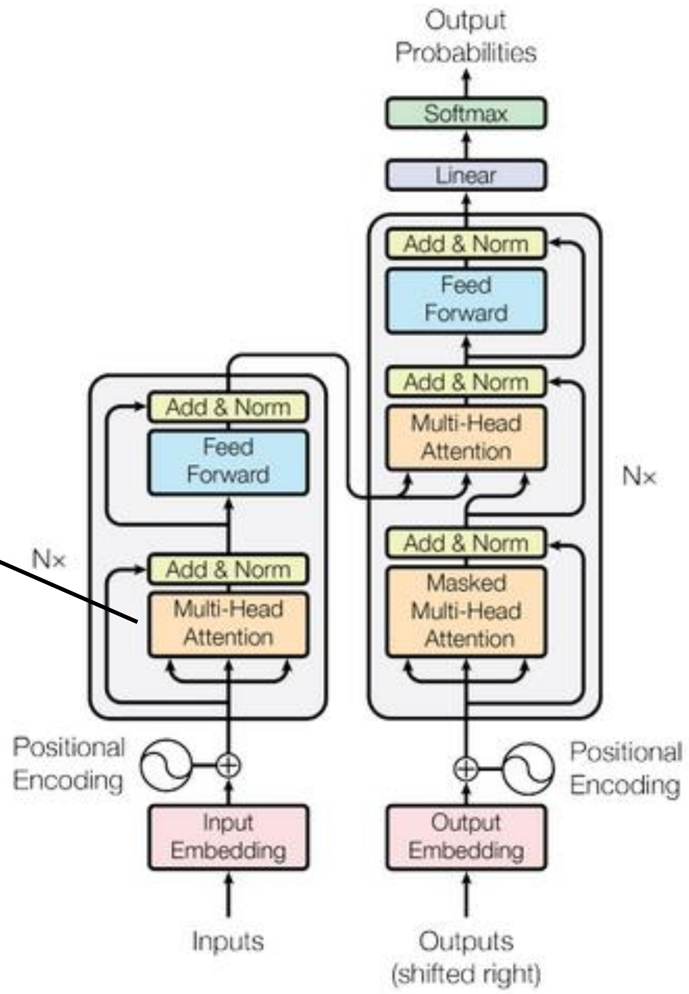
$$\text{Attention Weights} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

The attention weights are used to create a weighted sum of the **value** vectors:

$$\text{Attention Output} = \text{Attention Weights} \times V$$

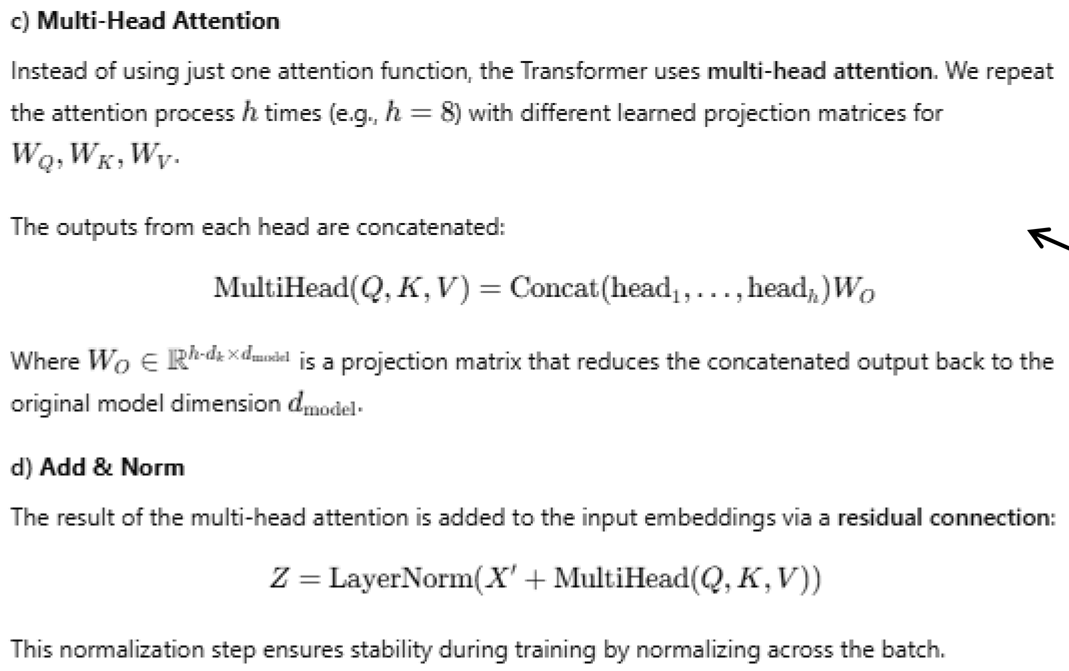
This results in an output matrix:

$$\text{Attention Output} \in \mathbb{R}^{n \times d_k}$$



2. Multi-Head Self-Attention (Encoder)

https://aibyhand.substack.com/p/11-can-you-calculate-self-attention



c) **Multi-Head Attention**

Instead of using just one attention function, the Transformer uses **multi-head attention**. We repeat the attention process h times (e.g., $h = 8$) with different learned projection matrices for W_Q, W_K, W_V.

The outputs from each head are concatenated:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W_O$$

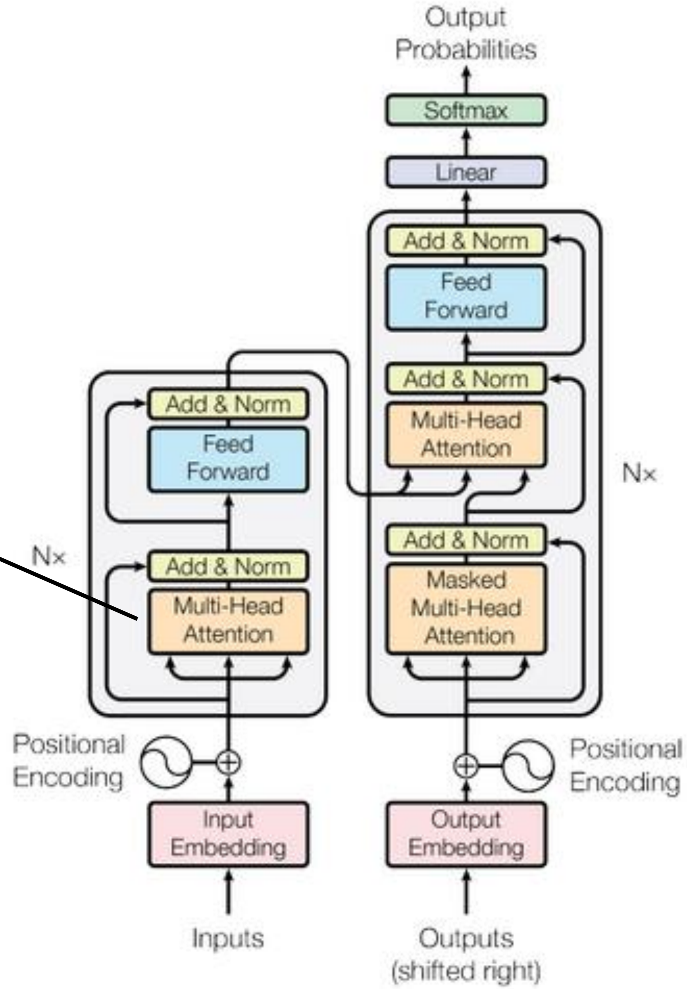
Where $W_O \in \mathbb{R}^{h \cdot d_k \times d_{\text{model}}}$ is a projection matrix that reduces the concatenated output back to the original model dimension d_{model}.

d) **Add & Norm**

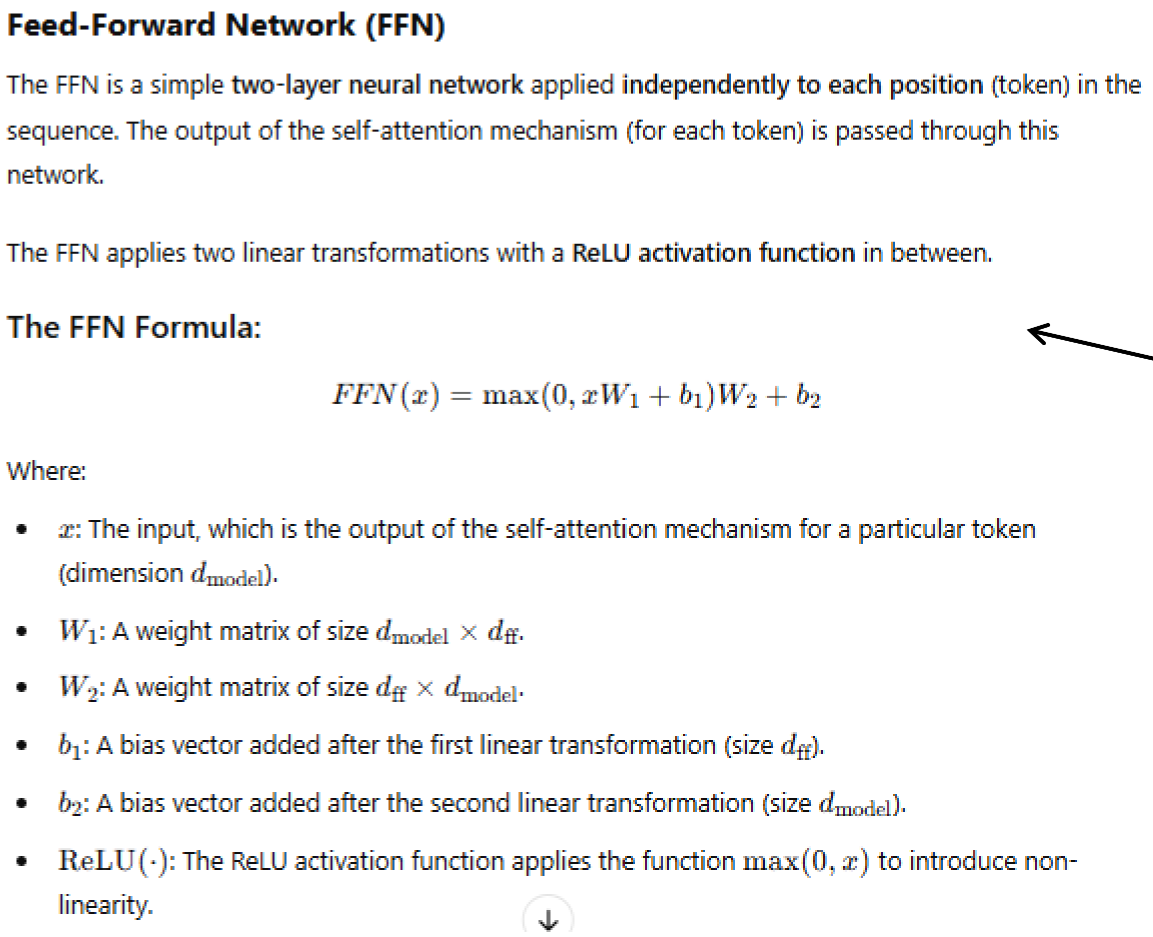
The result of the multi-head attention is added to the input embeddings via a **residual connection:**

$$Z = \text{LayerNorm}(X' + \text{MultiHead}(Q, K, V))$$

This normalization step ensures stability during training by normalizing across the batch.



3. FFN (Encoder)



Feed-Forward Network (FFN)

The FFN is a simple **two-layer neural network** applied **independently to each position** (token) in the sequence. The output of the self-attention mechanism (for each token) is passed through this network.

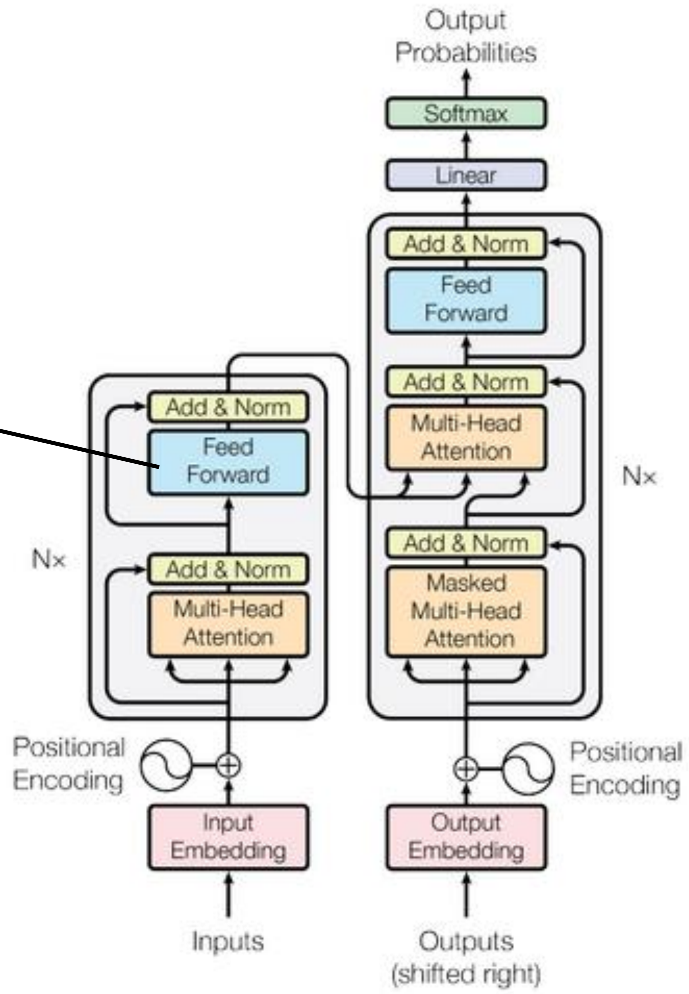
The FFN applies two linear transformations with a **ReLU activation function** in between.

The FFN Formula:

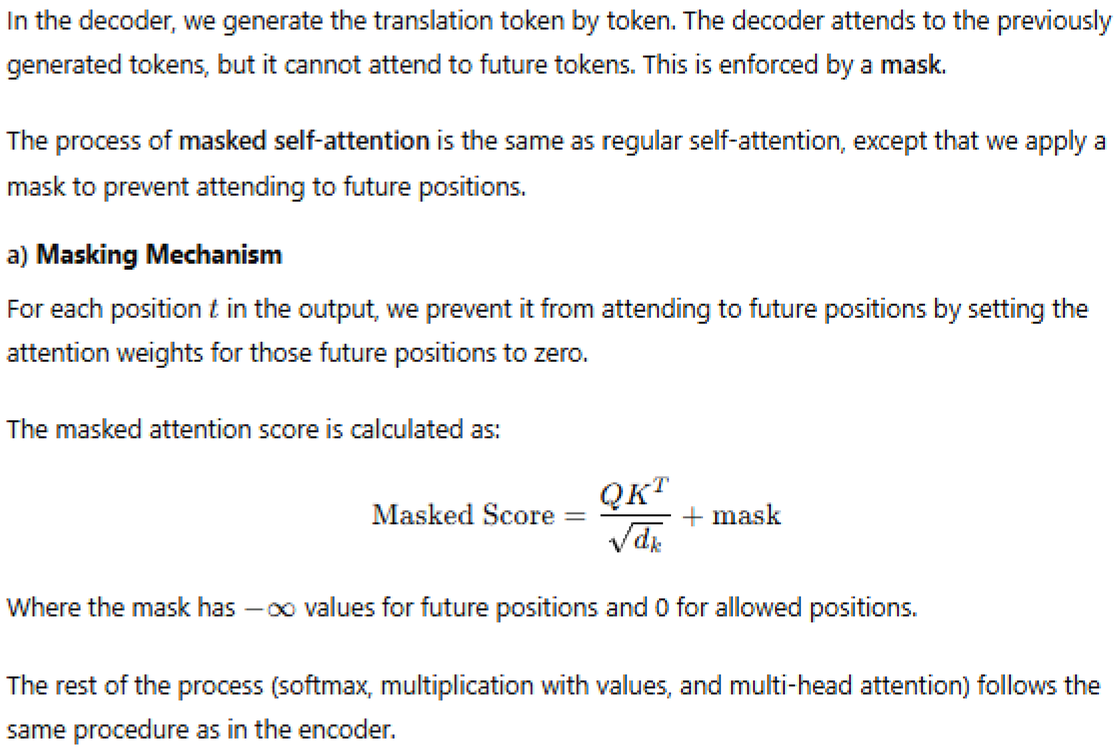
$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Where:

- x: The input, which is the output of the self-attention mechanism for a particular token (dimension d_{model}).
- W_1: A weight matrix of size $d_{\text{model}} \times d_{\text{ff}}$.
- W_2: A weight matrix of size $d_{\text{ff}} \times d_{\text{model}}$.
- b_1: A bias vector added after the first linear transformation (size d_{ff}).
- b_2: A bias vector added after the second linear transformation (size d_{model}).
- $\text{ReLU}(\cdot)$: The ReLU activation function applies the function $\max(0, x)$ to introduce non-linearity.



4. Masked Multi-Head Attention (Decoder)



In the decoder, we generate the translation token by token. The decoder attends to the previously generated tokens, but it cannot attend to future tokens. This is enforced by a **mask**.

The process of **masked self-attention** is the same as regular self-attention, except that we apply a mask to prevent attending to future positions.

a) **Masking Mechanism**

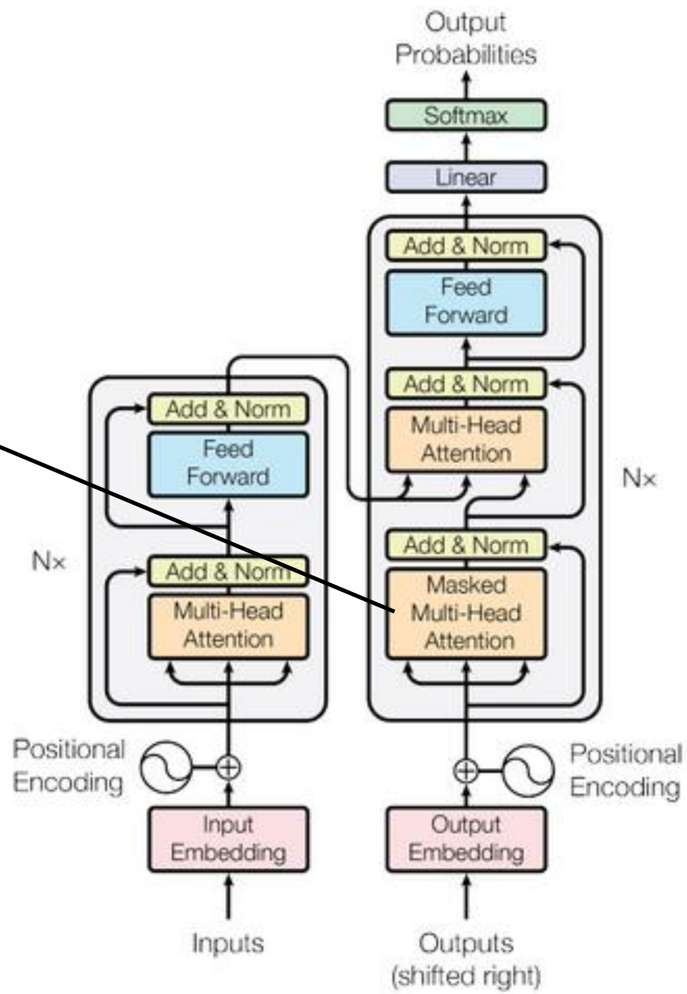
For each position t in the output, we prevent it from attending to future positions by setting the attention weights for those future positions to zero.

The masked attention score is calculated as:

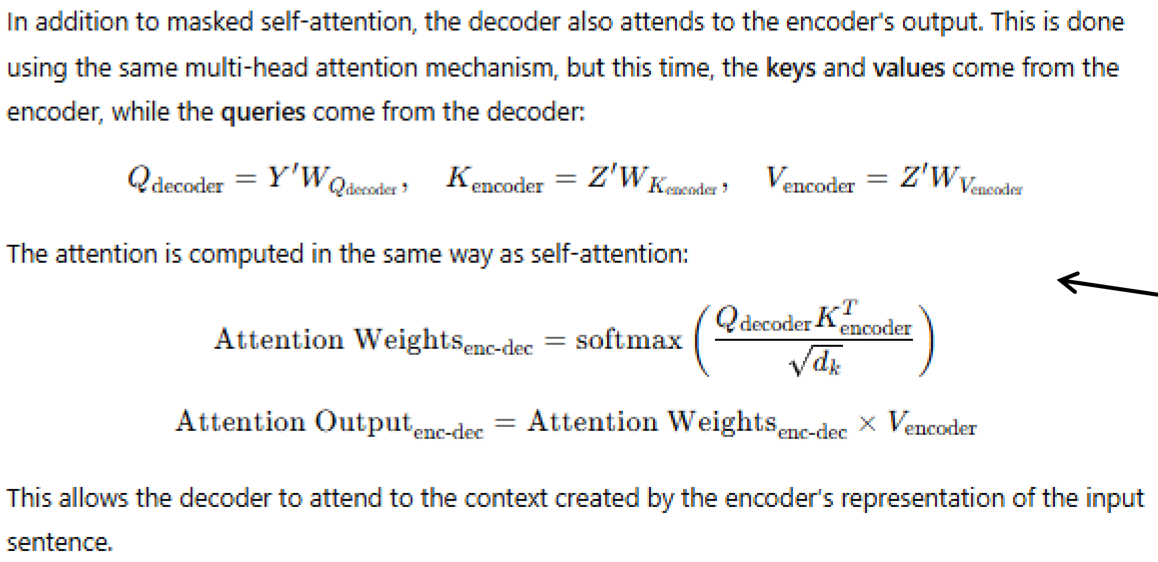
$$\text{Masked Score} = \frac{QK^T}{\sqrt{d_k}} + \text{mask}$$

Where the mask has $-\infty$ values for future positions and 0 for allowed positions.

The rest of the process (softmax, multiplication with values, and multi-head attention) follows the same procedure as in the encoder.



5. Encoder-Decoder Attention



In addition to masked self-attention, the decoder also attends to the encoder's output. This is done using the same multi-head attention mechanism, but this time, the **keys** and **values** come from the encoder, while the **queries** come from the decoder:

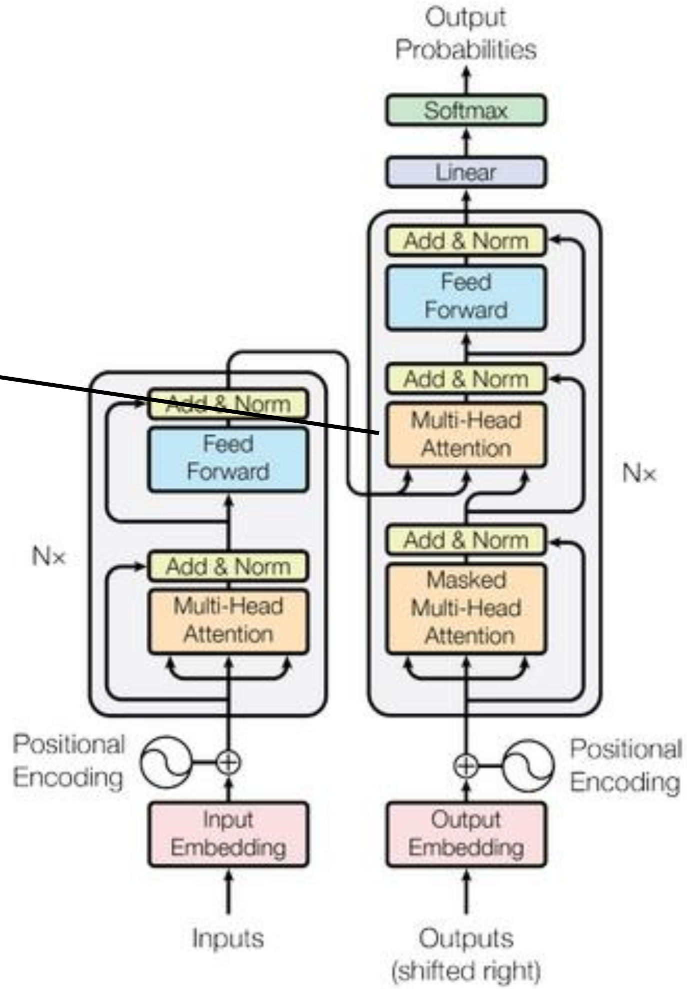
$$Q_{\text{decoder}} = Y'W_{Q_{\text{decoder}}}, \quad K_{\text{encoder}} = Z'W_{K_{\text{encoder}}}, \quad V_{\text{encoder}} = Z'W_{V_{\text{encoder}}}$$

The attention is computed in the same way as self-attention:

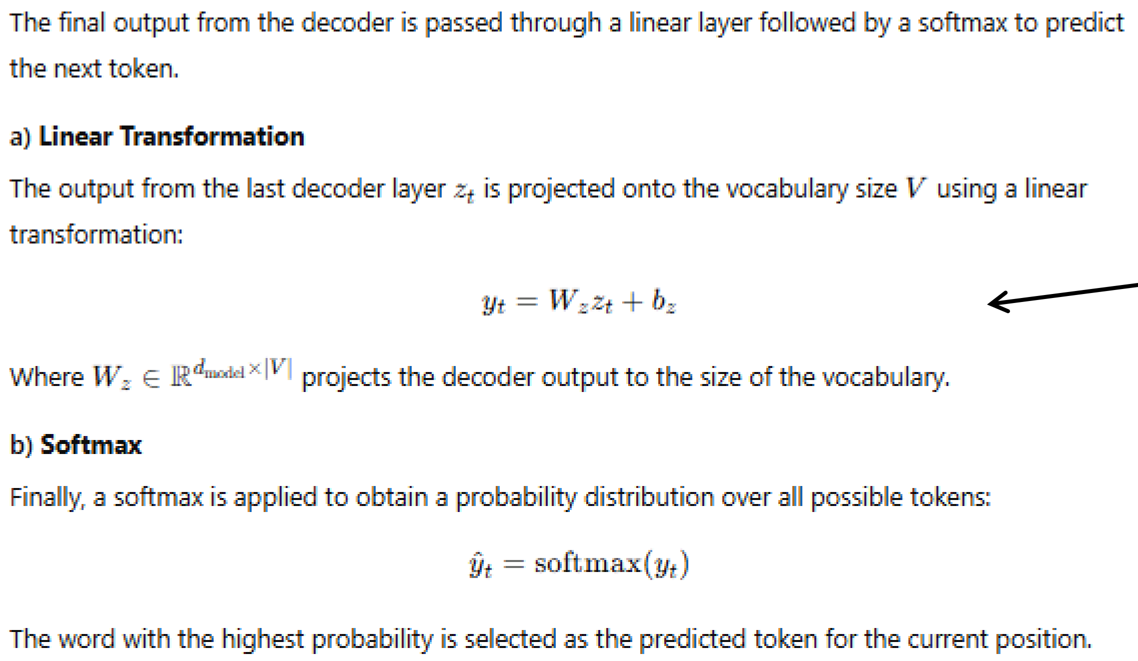
$$\text{Attention Weights}_{\text{enc-dec}} = \text{softmax}\left(\frac{Q_{\text{decoder}}K_{\text{encoder}}^T}{\sqrt{d_k}}\right)$$

$$\text{Attention Output}_{\text{enc-dec}} = \text{Attention Weights}_{\text{enc-dec}} \times V_{\text{encoder}}$$

This allows the decoder to attend to the context created by the encoder's representation of the input sentence.



6. Final Linear Layer and Softmax



The final output from the decoder is passed through a linear layer followed by a softmax to predict the next token.

a) **Linear Transformation**

The output from the last decoder layer z_t is projected onto the vocabulary size V using a linear transformation:

$$y_t = W_z z_t + b_z$$

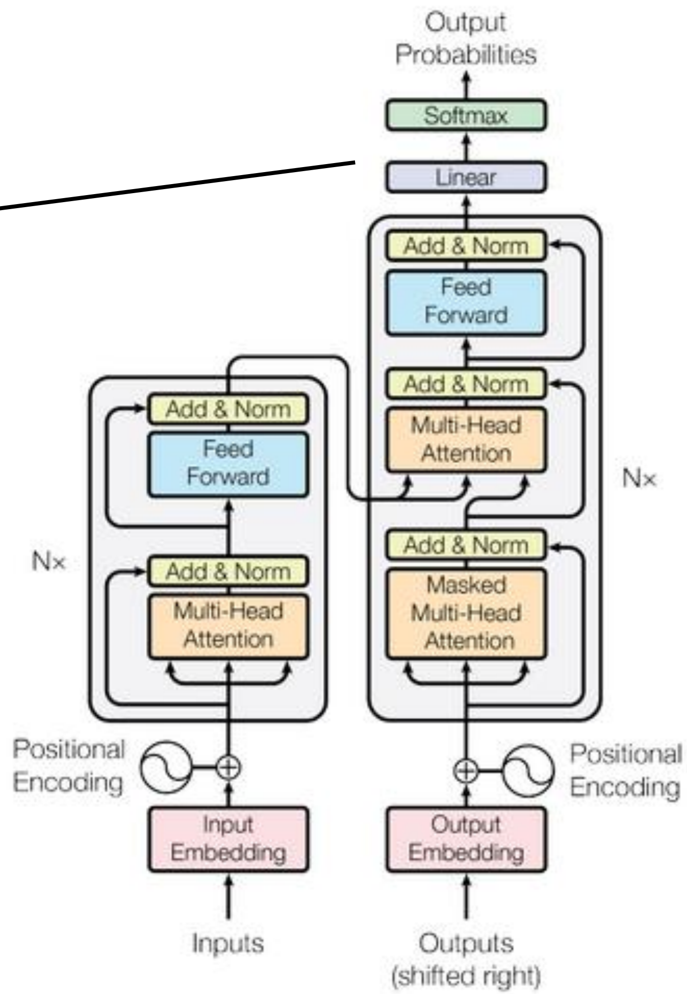
Where $W_z \in \mathbb{R}^{d_{\text{model}} \times |V|}$ projects the decoder output to the size of the vocabulary.

b) **Softmax**

Finally, a softmax is applied to obtain a probability distribution over all possible tokens:

$$\hat{y}_t = \text{softmax}(y_t)$$

The word with the highest probability is selected as the predicted token for the current position.



Useful Links

- Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems. 2017.
- https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html
- https://www.analyticsvidhya.com/blog/2019/06/understanding-transformers-nlp-state-of-the-art-models/
- https://jalammar.github.io/illustrated-transformer/
- Sample Code Links:
  - https://github.com/tensorflow/tensor2tensor/blob/master/README.md#walkthrough
  - https://colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb#scrollTo=s19ucTii_wYb