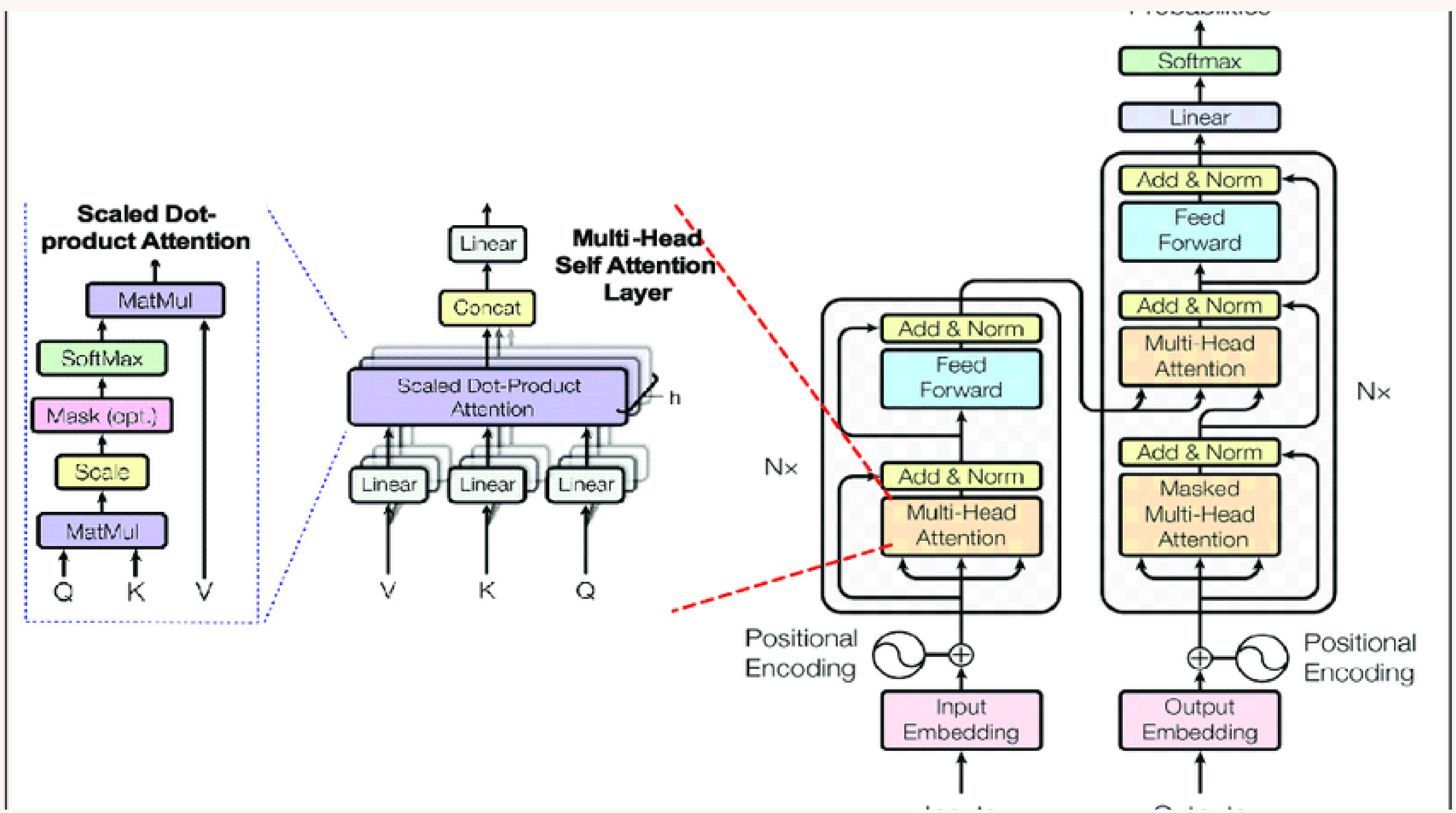
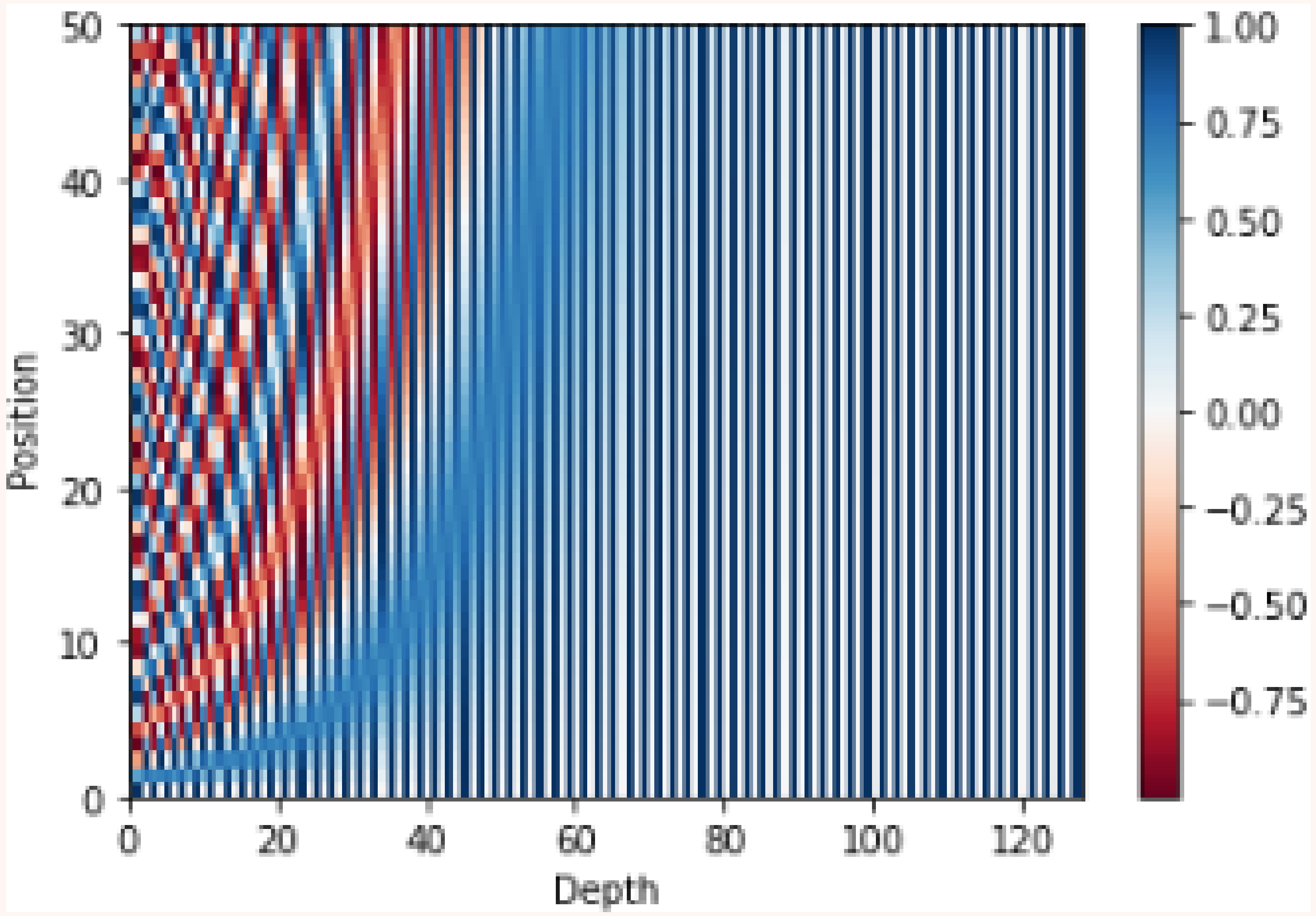
Transformer

OpenAI GPT-3

DON'T FORGET

Architecture





Positional Embeddings

CONTEXT

Unlike Traditional RNN models, Transformers alone can't distinguish meanings behind words when they vary in different locations. Therefore... such positional informations must be embedded in the word vector which is tackled by Positional Embeddings.

Example

Mom told Tim to eat

vs.

Tim told Mom to eat

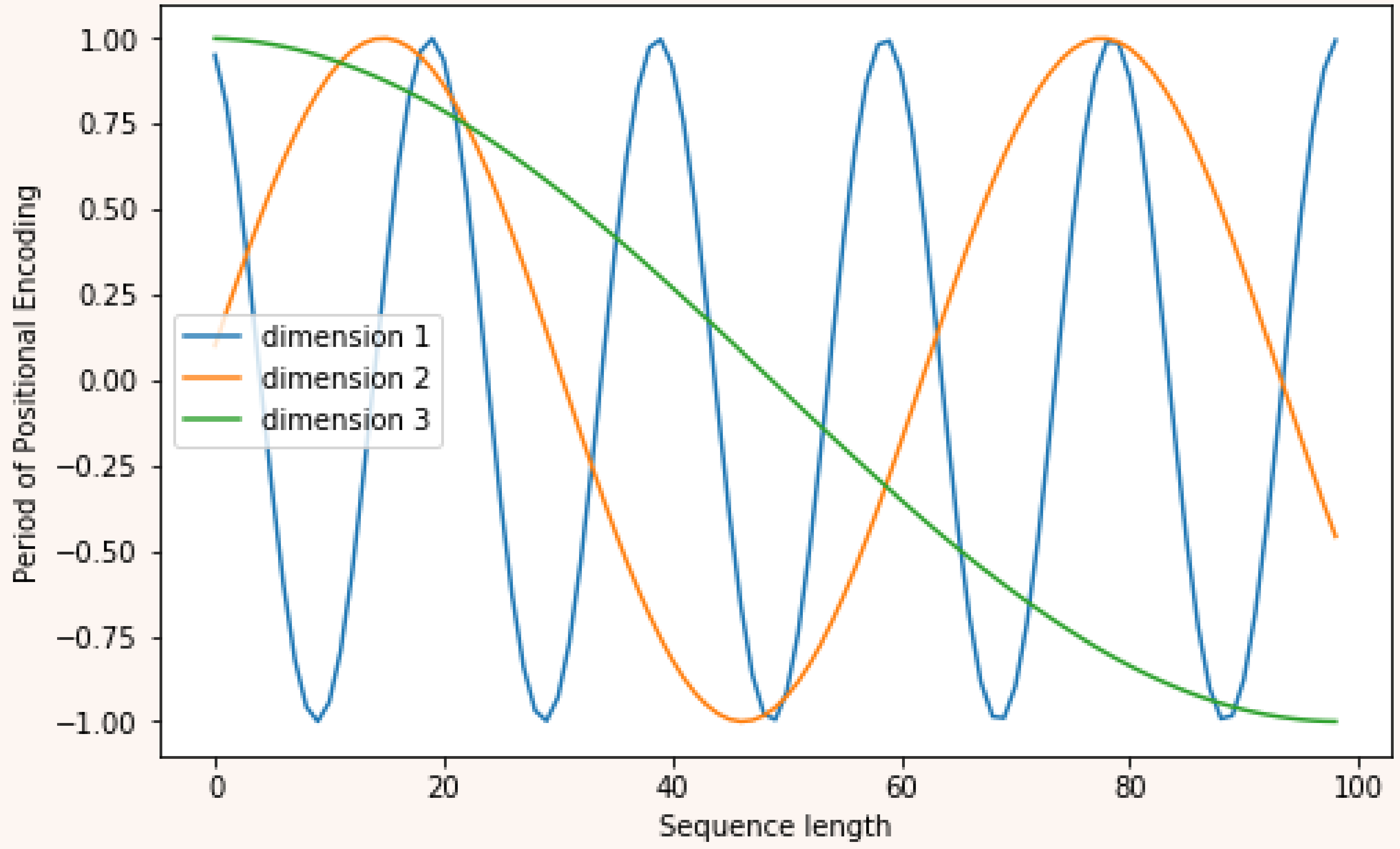
HOW IT WORKS?

- For even word indexes, use Sin
- For Odd word indexes, use Cos

The Goal is that for every different word, PE always give different PE embeddings for variation between words due to locations

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

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

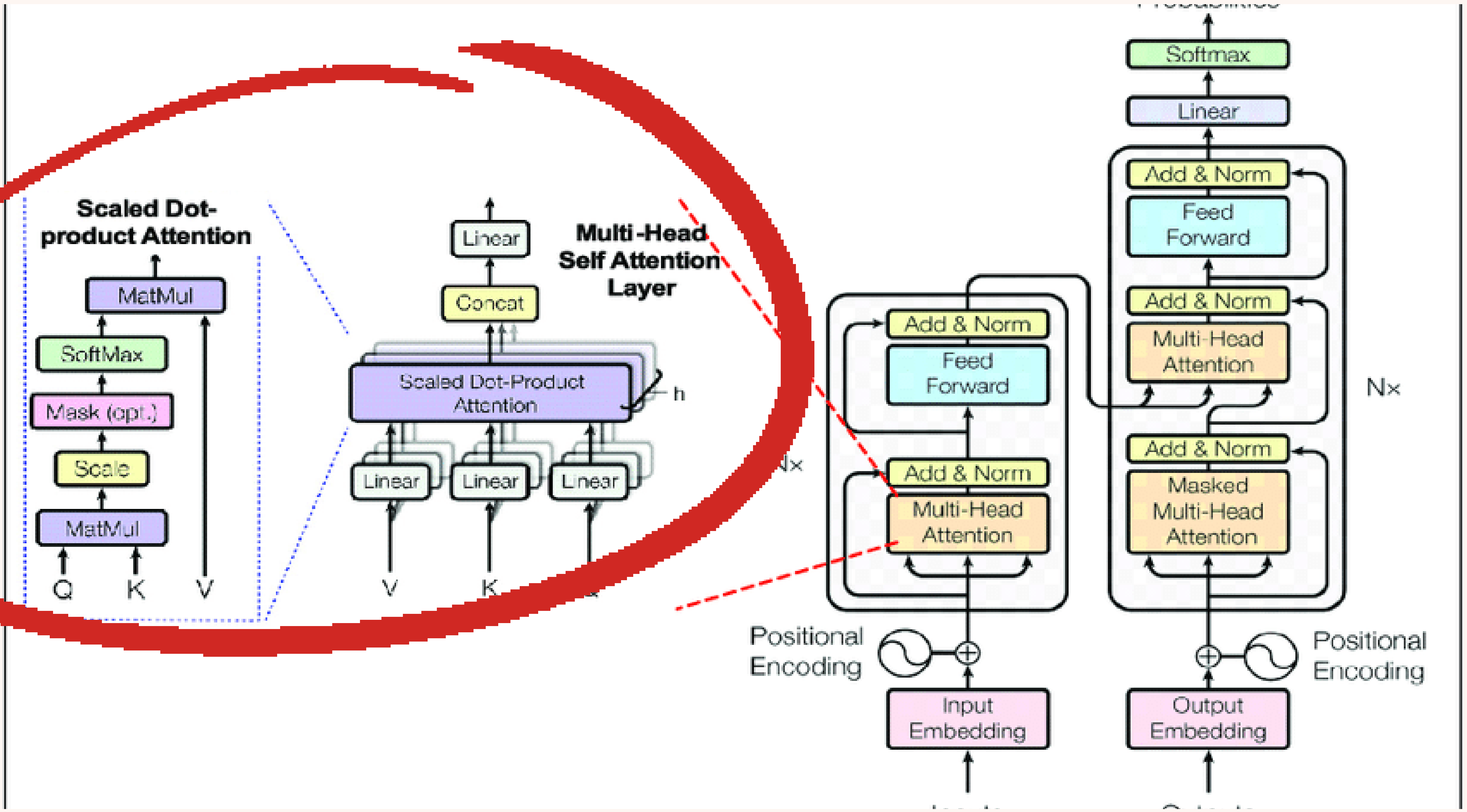


You can see that the PE Embeddings Wobble, such that every word given different locations will always have a variated embedding

NOTE: Despite being same word, they still have differentiated Embeddings due to their locations

(1)PE:	[0,0,1]	[0.84,0.001,0.54]	[0.91,-0.001,0.24]
(2)Embedding:	[1,0,0]	[1,0,0]	[1,0,0]
(3)Input:	Apple	Apple	Apple
(1 + 2)Final Embedding:	[1,0,1]	[1.84,0.001,0.54]	[1.91,-0.001,0.24]

Multi-Head Attention



Note:

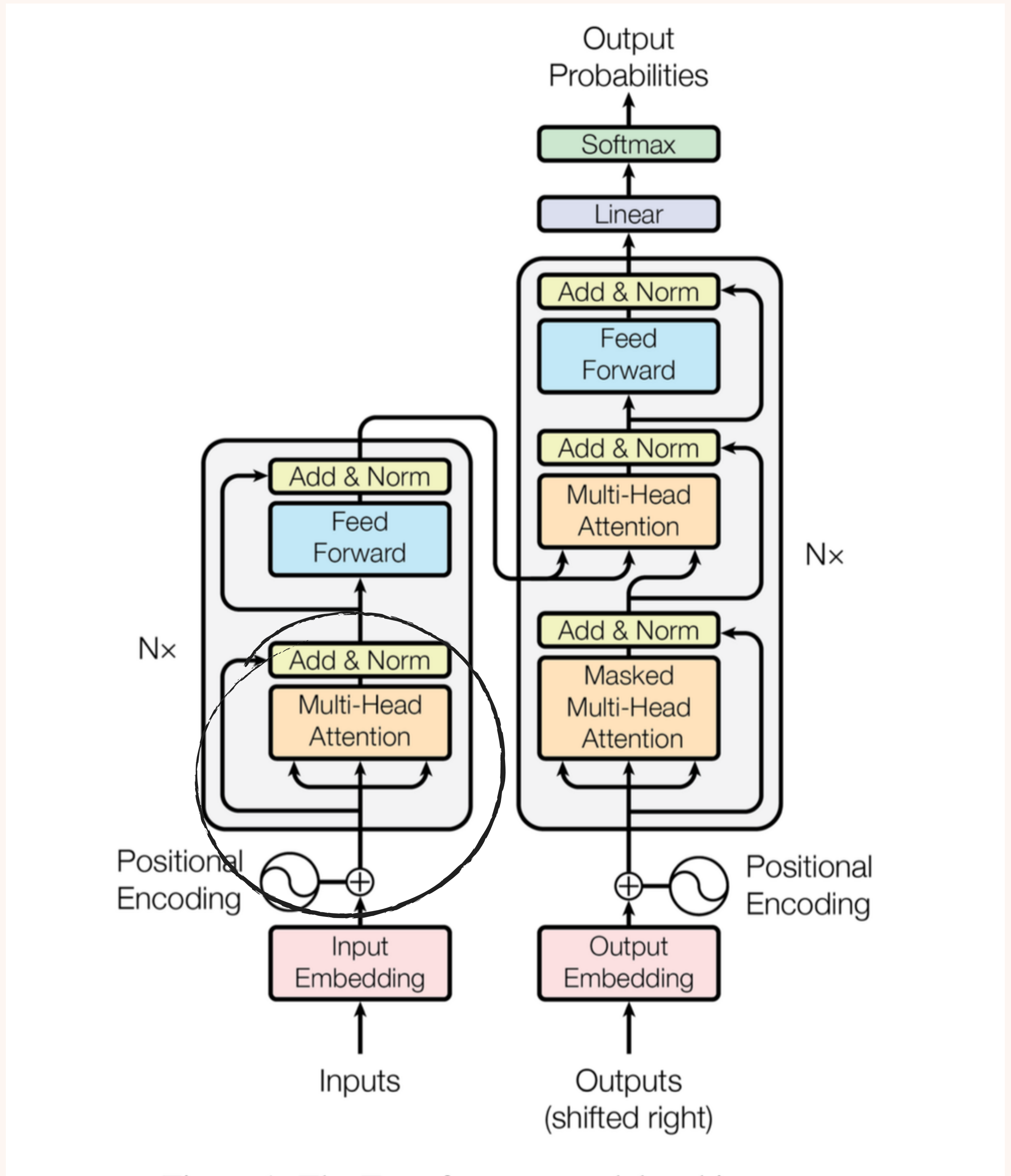
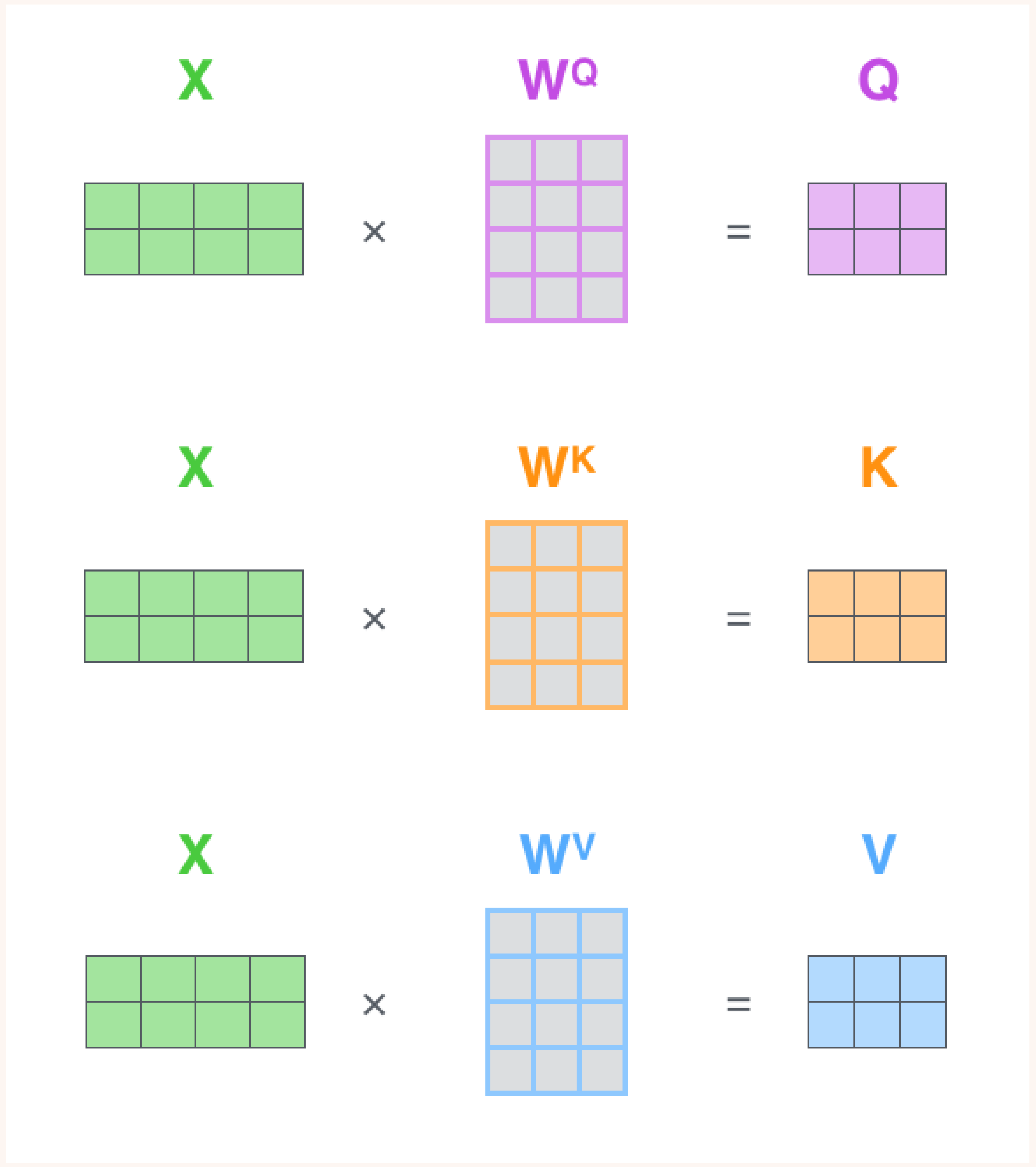


Figure 1: The Transformer - model architecture.

Slides 8 - 12 will cover **Multi-Head Attention** by going over the section circled on the left hand side, to block

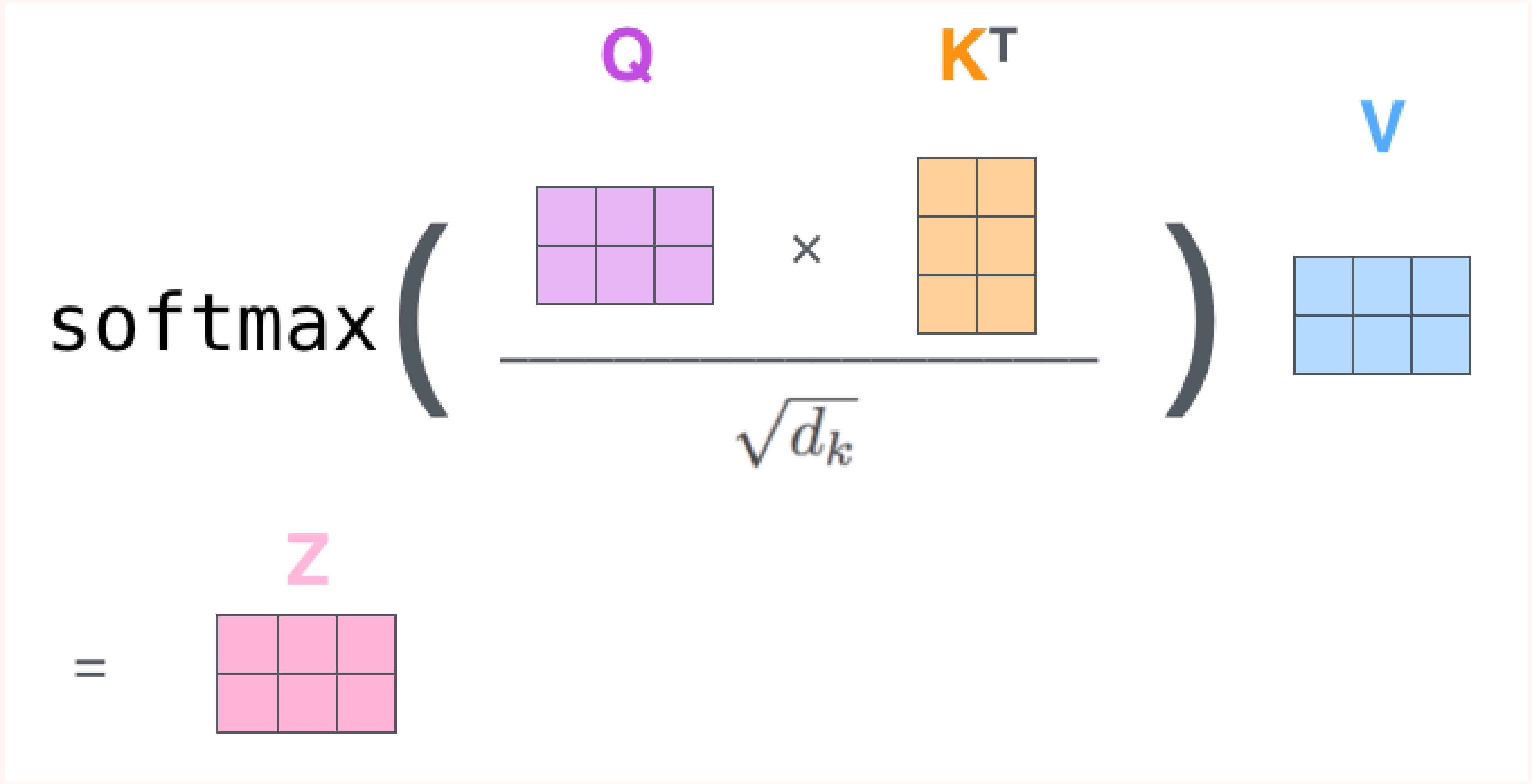
[Add & Norm]

How to Get Q, K, V



1. "X" represents the final embedding after you passed the input into embedding layer and have it added with the positional embeddings
2. Clone "X" three times
3. Let wQ, wK, wV (3 mini NeuralNets) multiply each cloned X
4. Get "Q", "K", "V" and save an another cloned primitive "X" (for future Add & Norm)

Output of Multi-Head Attention: Z



1. Say we have 8 Attention Heads, this computation will occur 8 times
2. Say the num_embed was 32, each attention head gets 4 dimensions to compute from (32/8 = 4)
3. dk is basically the 4 from 2.

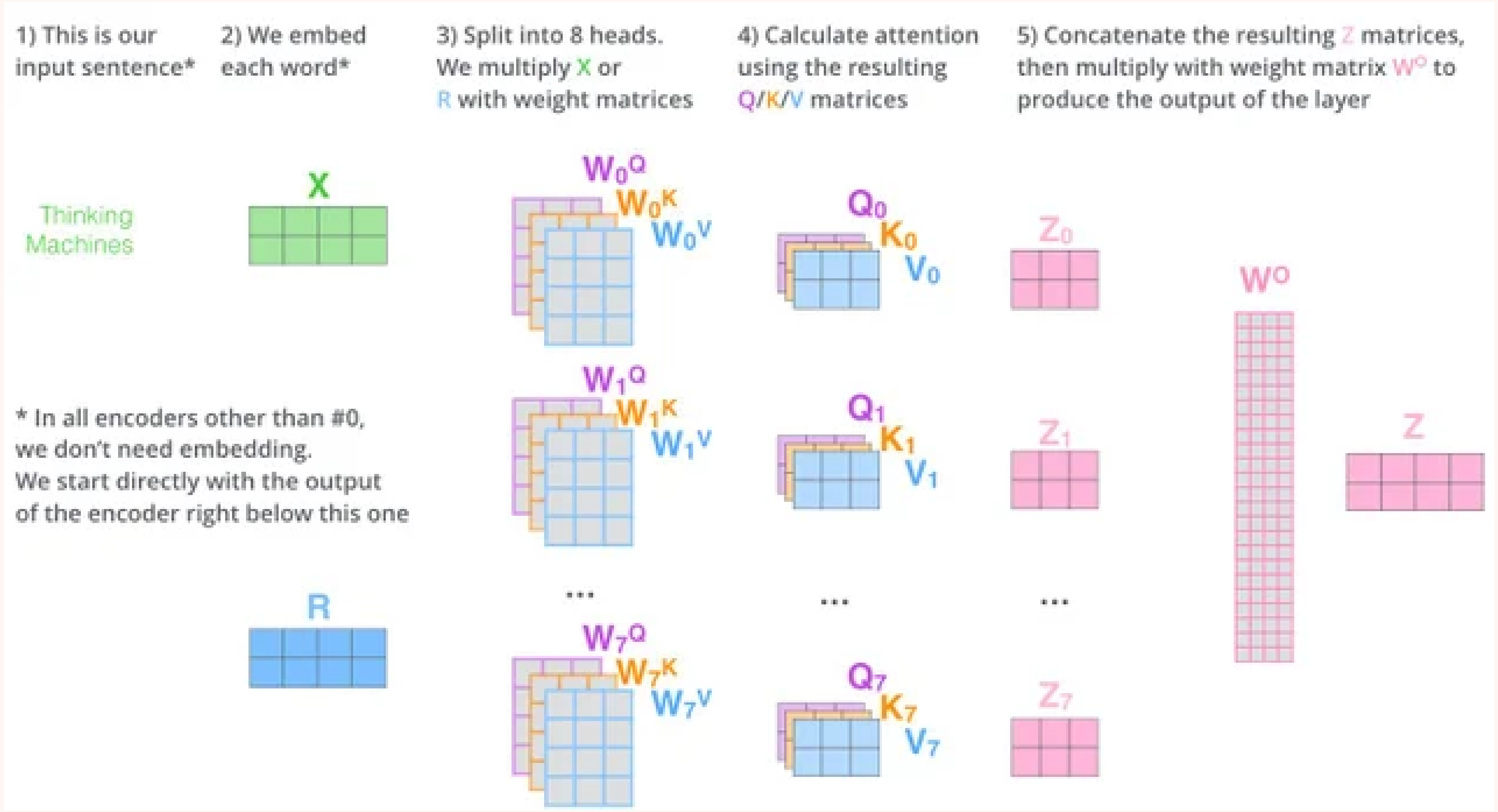
Softmax convert values into proabilities, because they will sum to 1

Pad mask

1. Before computing Softmax from slides 9, we will apply a pad mask to the input. By doing so, we aim to make each word's attention to padding tokens ZERO. (So Softmax will not compute it)
2. It's Okay to have padding token compute attention to different words so they learn how they themselves relate to other words

	a	b	<PAD>
a	a*a	a*b	MASK
b	b*a	b*b	MASK
<PAD>	P*a	P*b	MASK

Note: each Z(i) represent outputs from slides 9.


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
Thinking Machines
X
R
* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one
W0Q W0K W0V
W1Q W1K W1V
W7Q W7K W7V
Q0 K0 V0
Q1 K1 V1
Q7 K7 V7
Z0
Z1
Z7
WO
Z

Last but not least: Add & Norm

Final Output = Layer Norm(X + Z)

Note: The X comes from the cloned primitive X during slides

Note: The Z is the same final output Z from the last slide

Congratulations!

That was the end of Multi-Head Attention! QUESTIONS?

Encoder Layer

Encoder = Bunch of Encoder Layers stacked on top of one another

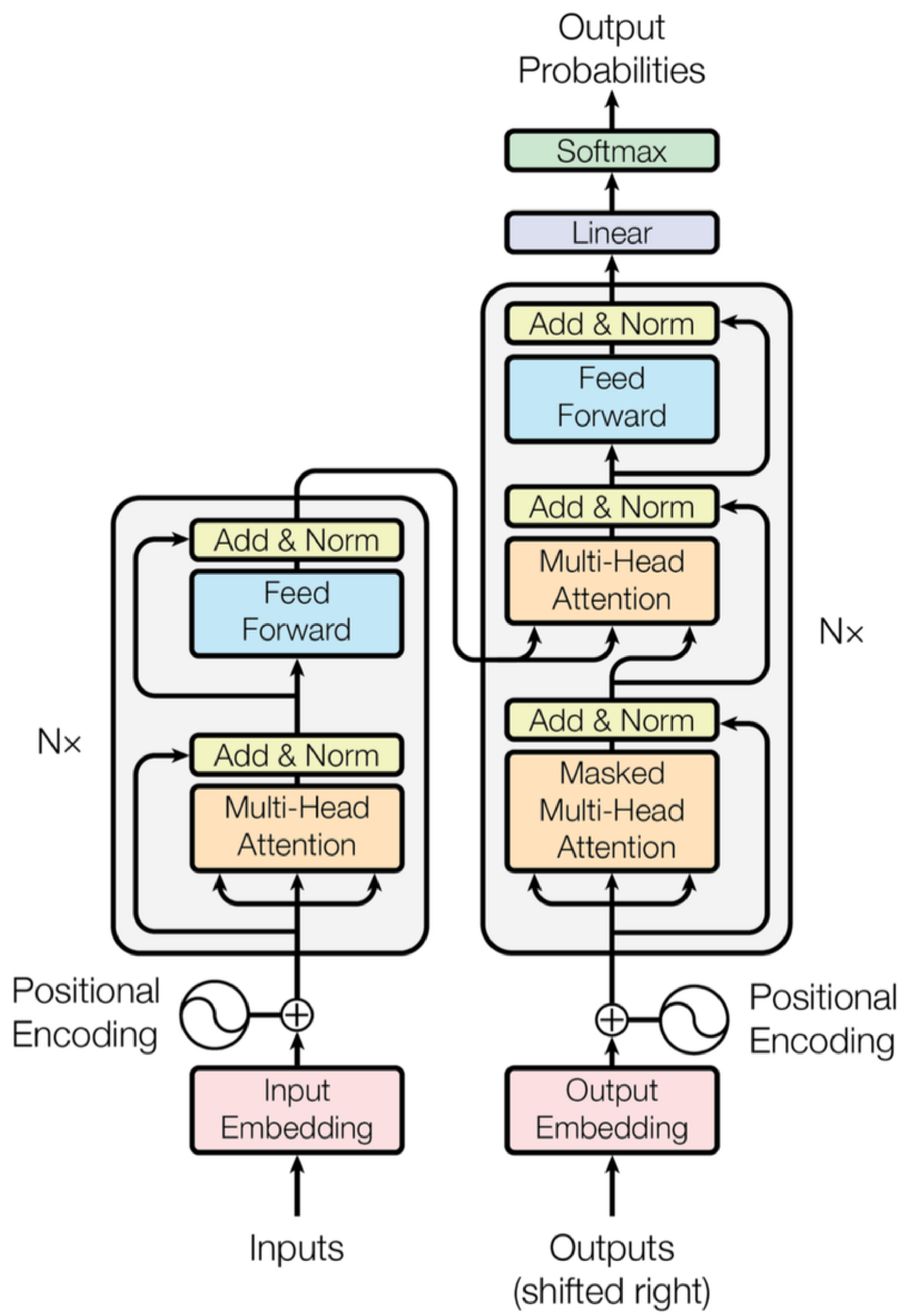


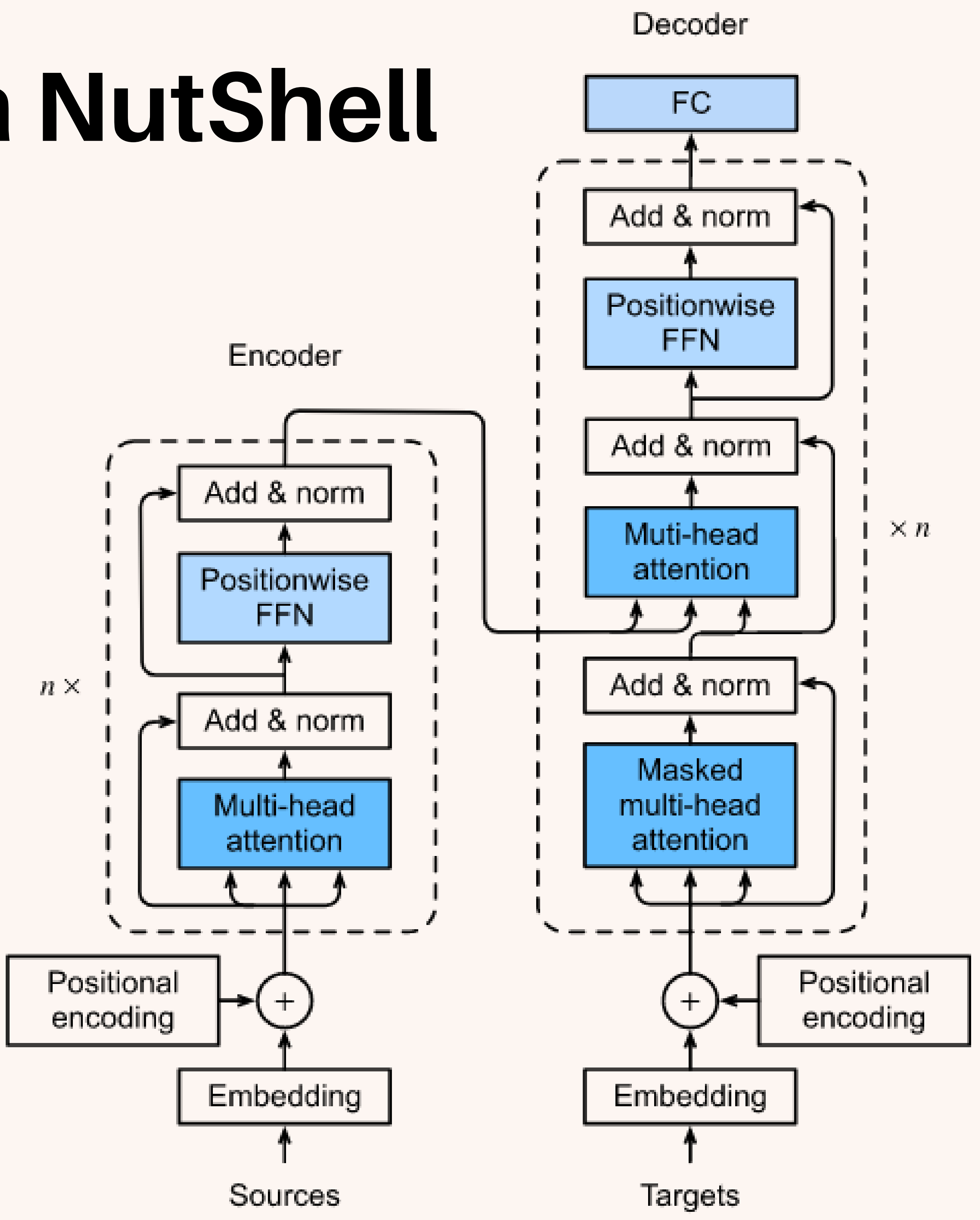
Figure 1: The Transformer - model architecture.

Encoder Layer in a NutShell

1. Let input word tokens go through an embedding layer. Add it with the positional embedding and that will be the Encoder inputs. (Shape should look like: [batch_size, src_maxlen, num_embedding])

(If this ain't the first Encoder Layer, input will just be the last Encoder Layer output)

1. Compute the Multi-Head Attention as demonstrated from slides 7 - 12
2. Save a cloned copy of the output Z from Multi-Head Attention.
3. Then feedforward it to a fully connected neural network.
4. Concat output from 5. with the cloned_Z from 3.
5. Apply Layer Norm. And we're DONE!



Decoder Layer

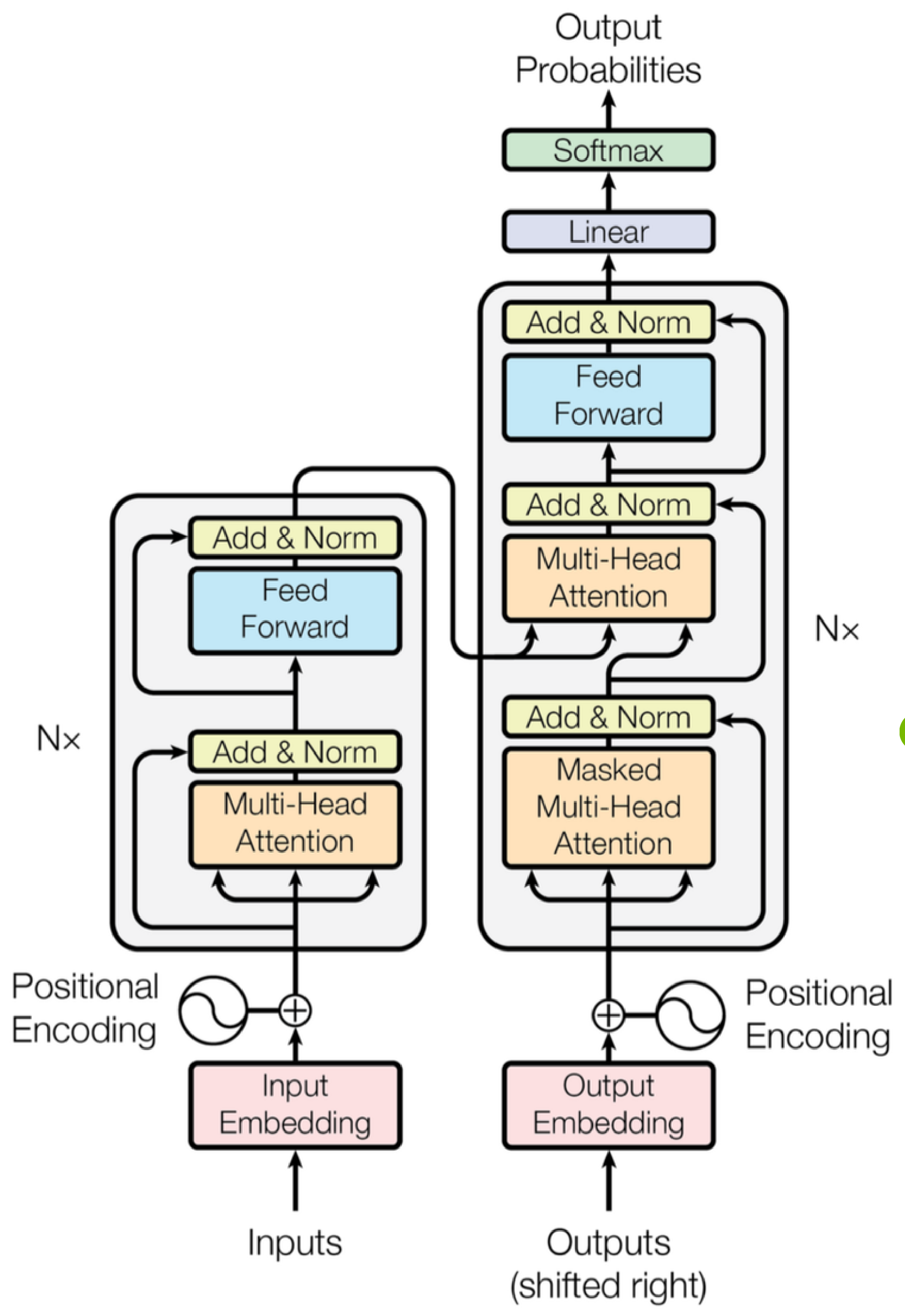
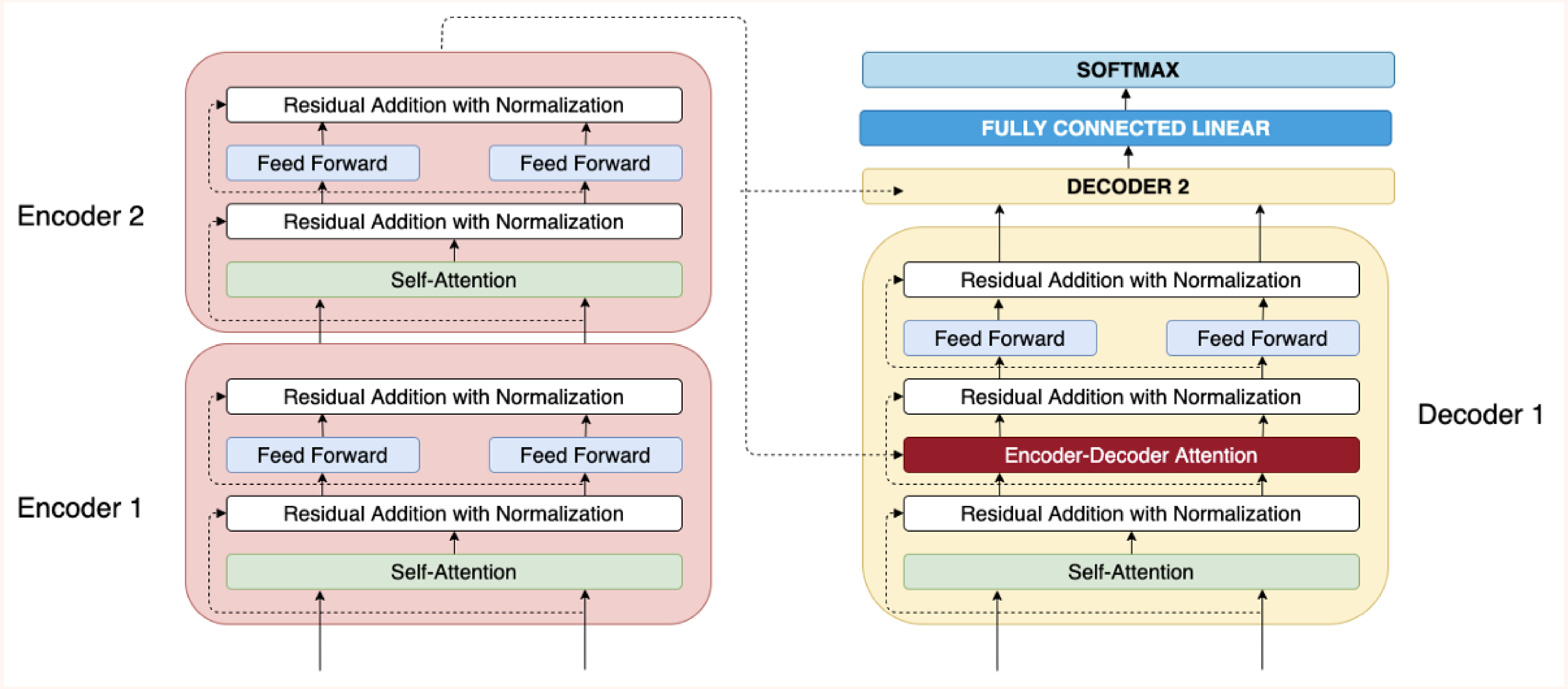


Figure 1: The Transformer - model architecture.

Decoder = Bunch of Decoder Layers stacked on top of one another

You can tell Decoder Layers looks very similar to Encoder Layers, but... they vary slightly



SOFTMAX
FULLY CONNECTED LINEAR
DECODER 2
Encoder 2
Residual Addition with Normalization
Feed Forward
Feed Forward
Residual Addition with Normalization
Self-Attention
Encoder 1
Residual Addition with Normalization
Feed Forward
Feed Forward
Residual Addition with Normalization
Self-Attention
Decoder 1
Residual Addition with Normalization
Feed Forward
Feed Forward
Residual Addition with Normalization
Encoder-Decoder Attention
Residual Addition with Normalization
Self-Attention

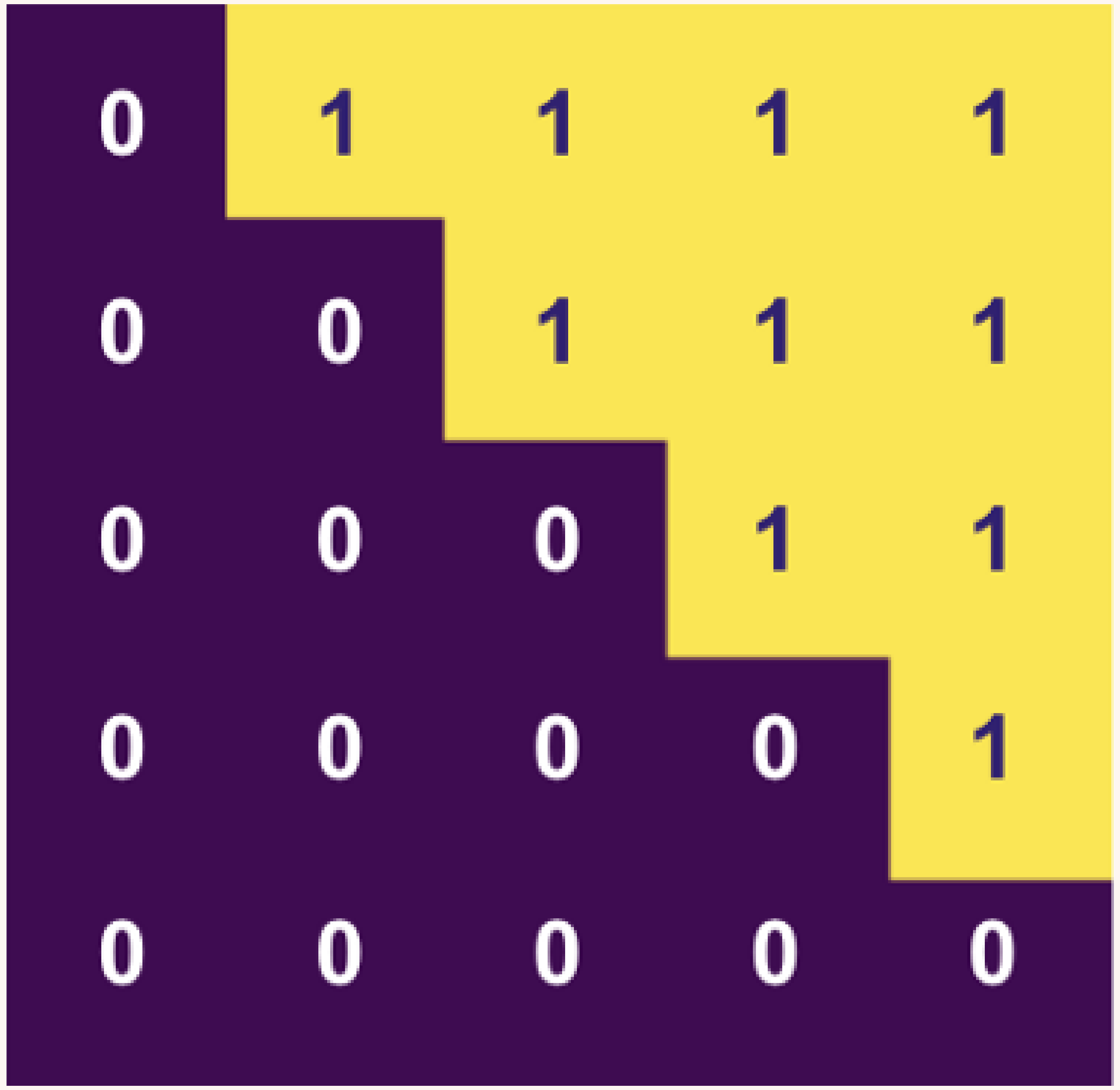
Encoder Layer VS. Decoder Layer

1. The first Decoder Layers takes input from the final embedding of the target sequence, where as the first Encoder Layer takes input from the final embedding of input sequence (final embedding = embedding layer + PE embedding)
2. A Decoder Layer has two Multi-Head Attentions, first comes the Regular Multi-Head Attention, second comes the "Encoder-Decoder Attention"
3. A Decoder Layer's first Multi-Head Attention utilizes both pad mask and tril mask

1) Should be pretty intuitive, we will just cover "tril mask" and "Encoder-Decoder Attention"

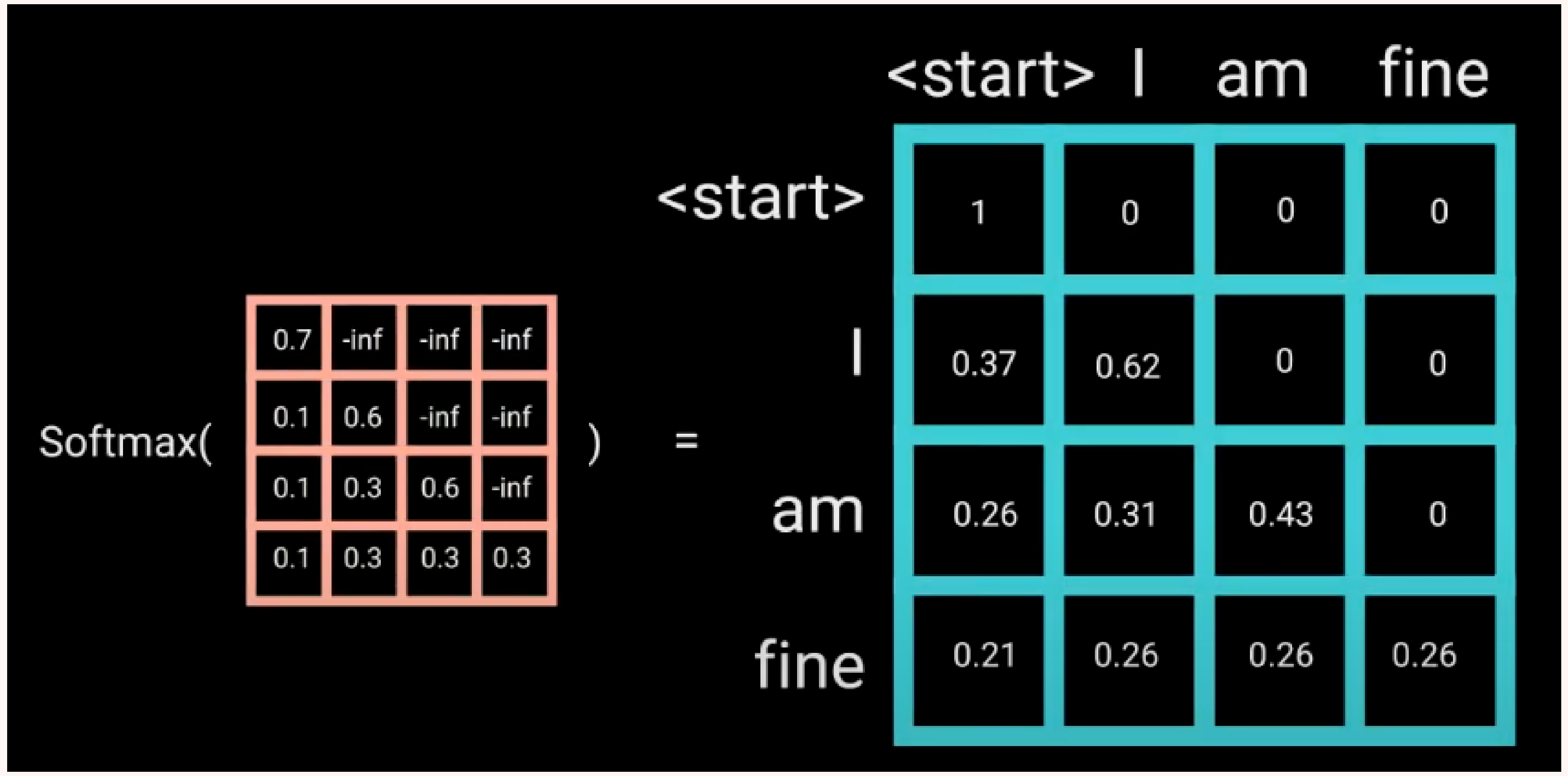
Tril Mask

Recall that Pad Mask aims to prevent model from computing attention of a word to a padding token. Tril Mask serves similar purposes but are slightly different



0	1	1	1	1
0	0	1	1	1
0	0	0	1	1
0	0	0	0	1
0	0	0	0	0

Why?

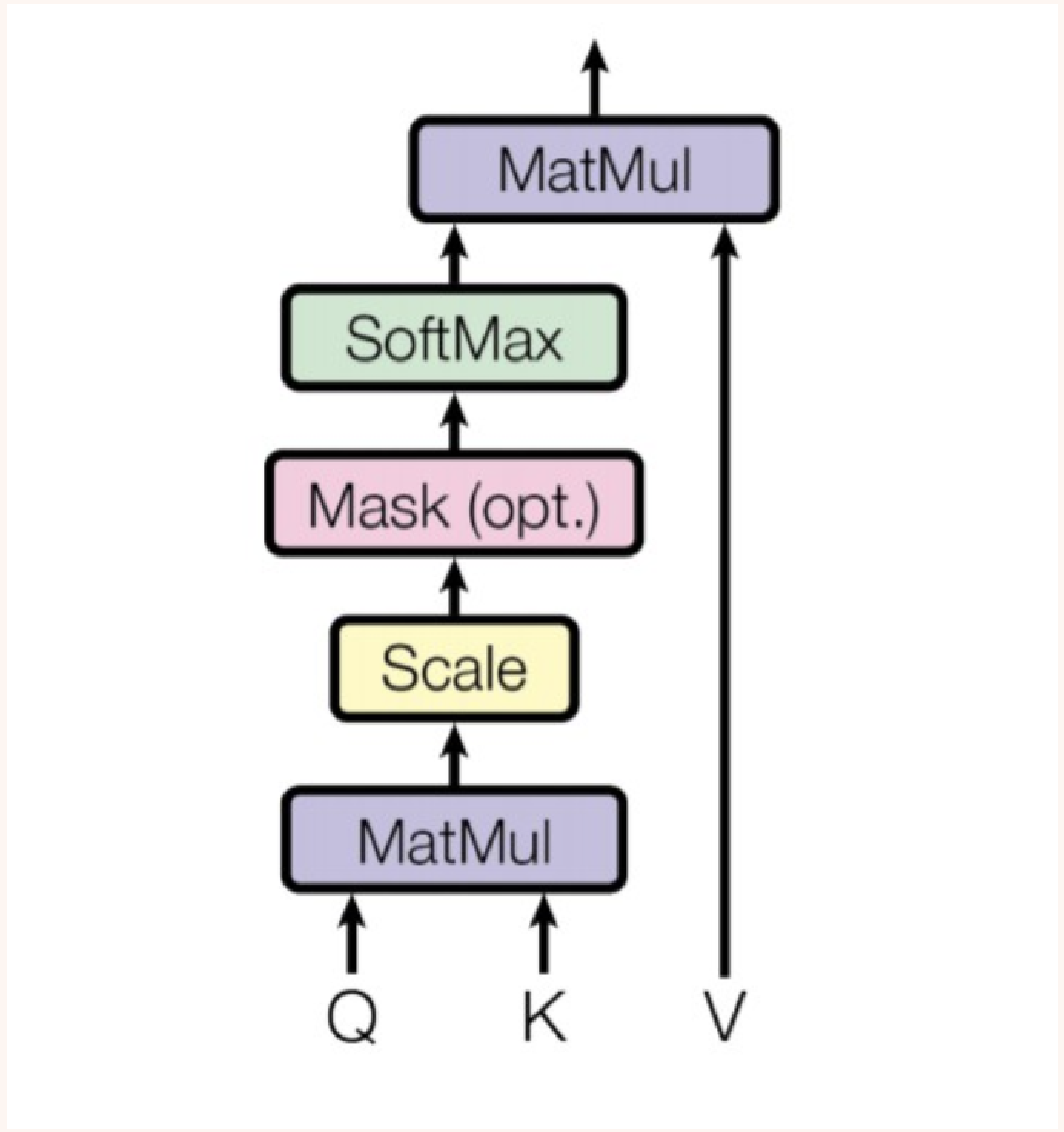


$$\text{Softmax}\left(\begin{array}{cccc} 0.7 & -\text{inf} & -\text{inf} & -\text{inf} \\ 0.1 & 0.6 & -\text{inf} & -\text{inf} \\ 0.1 & 0.3 & 0.6 & -\text{inf} \\ 0.1 & 0.3 & 0.3 & 0.3 \end{array}\right) = \begin{array}{c|cccc} & \text{<start>} & \text{I} & \text{am} & \text{fine} \\ \hline \text{<start>} & 1 & 0 & 0 & 0 \\ \text{I} & 0.37 & 0.62 & 0 & 0 \\ \text{am} & 0.26 & 0.31 & 0.43 & 0 \\ \text{fine} & 0.21 & 0.26 & 0.26 & 0.26 \end{array}$$

Say when we Input "I", the model should only know the infos from the words before "I" such as "<start >" and "I". It shouldn't learn about infos from the words following "I". When we make predictions, we do so word by word and in order and how can we know what information the subsequent words will include? We want our model to predict and not CHEAT. To use this mask, first create a matrix with all zeros in the bottom triangle and all negative infinity in the top triangle.

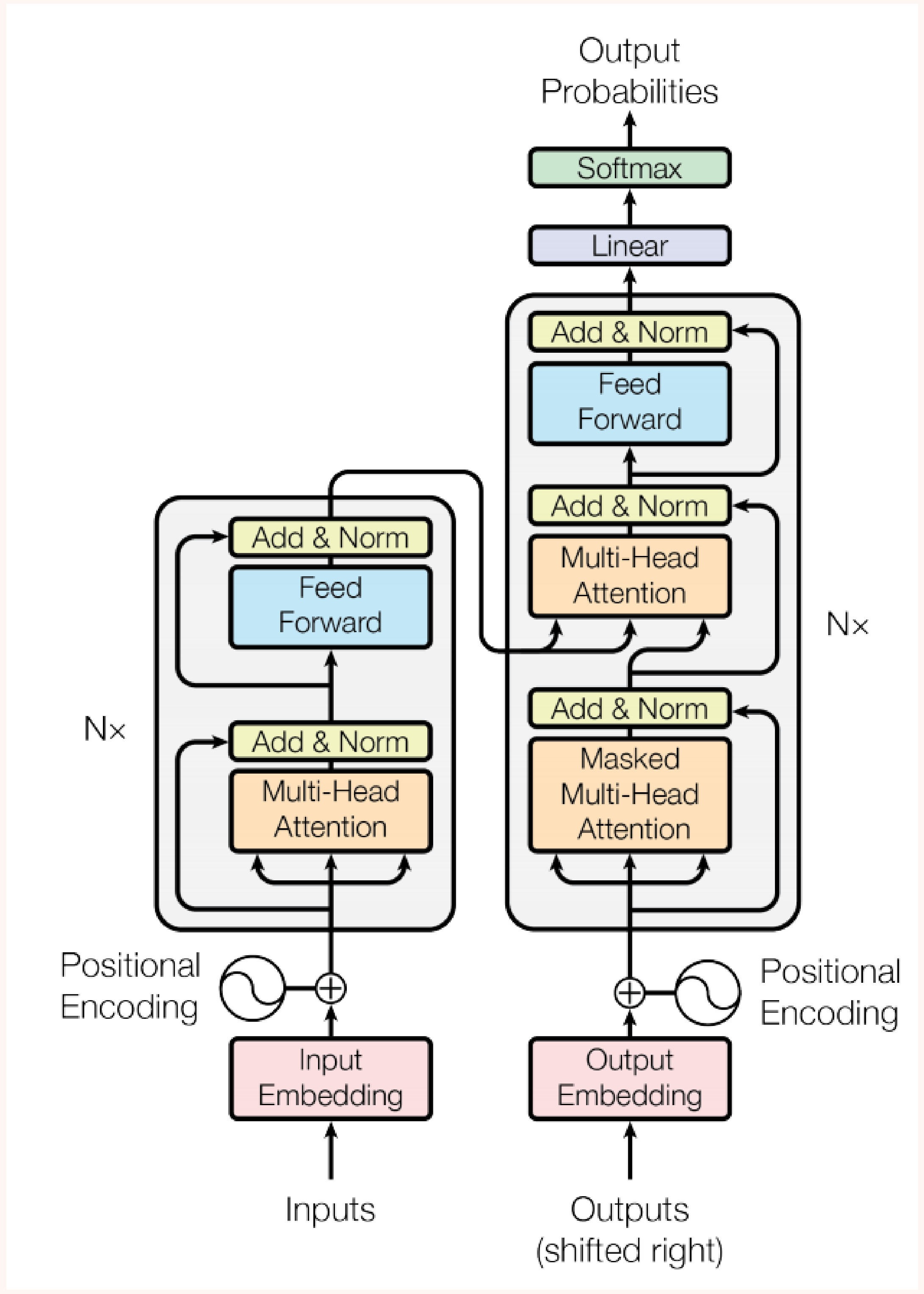
Encoder–Decoder Attention

The only variation in Encoder-Decoder Attention is that the Q comes from the last decoder layer output (or computed target sequence with Wv if this is the first layer) and K, V comes from the Encoder output (Encoder output K & V are the same)



Take a Moment to Digest...

Question Time



CREDITS

1. https://www.gkm0120.cn/transformer/#toc-head-5
2. https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04
3. https://zhuanlan.zhihu.com/p/44121378
4. https://medium.com/geekculture/transformers-231c4a430746
5. https://wmathor.com/index.php/archives/1438/