9. Transformer

LING-581-Natural Language Processing 1

Instructor: Hakyung Sung

October 21, 2025

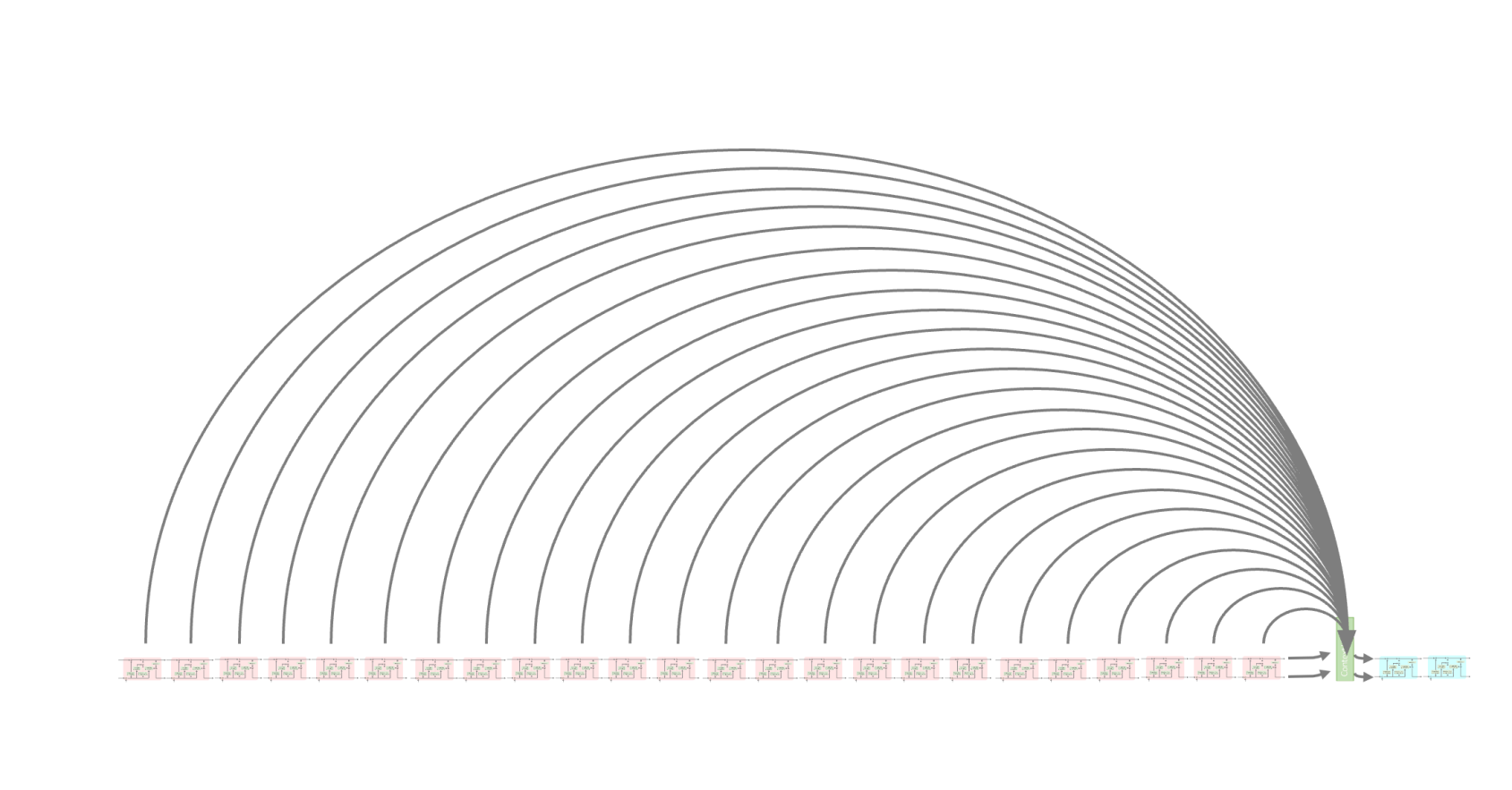
*Acknowledgment: These course slides are based on materials from CS224N @ Stanford University; Dr. Kilho Shin @ Kyocera

Table of contents

Review: Attention

Seq2Seq: Problem

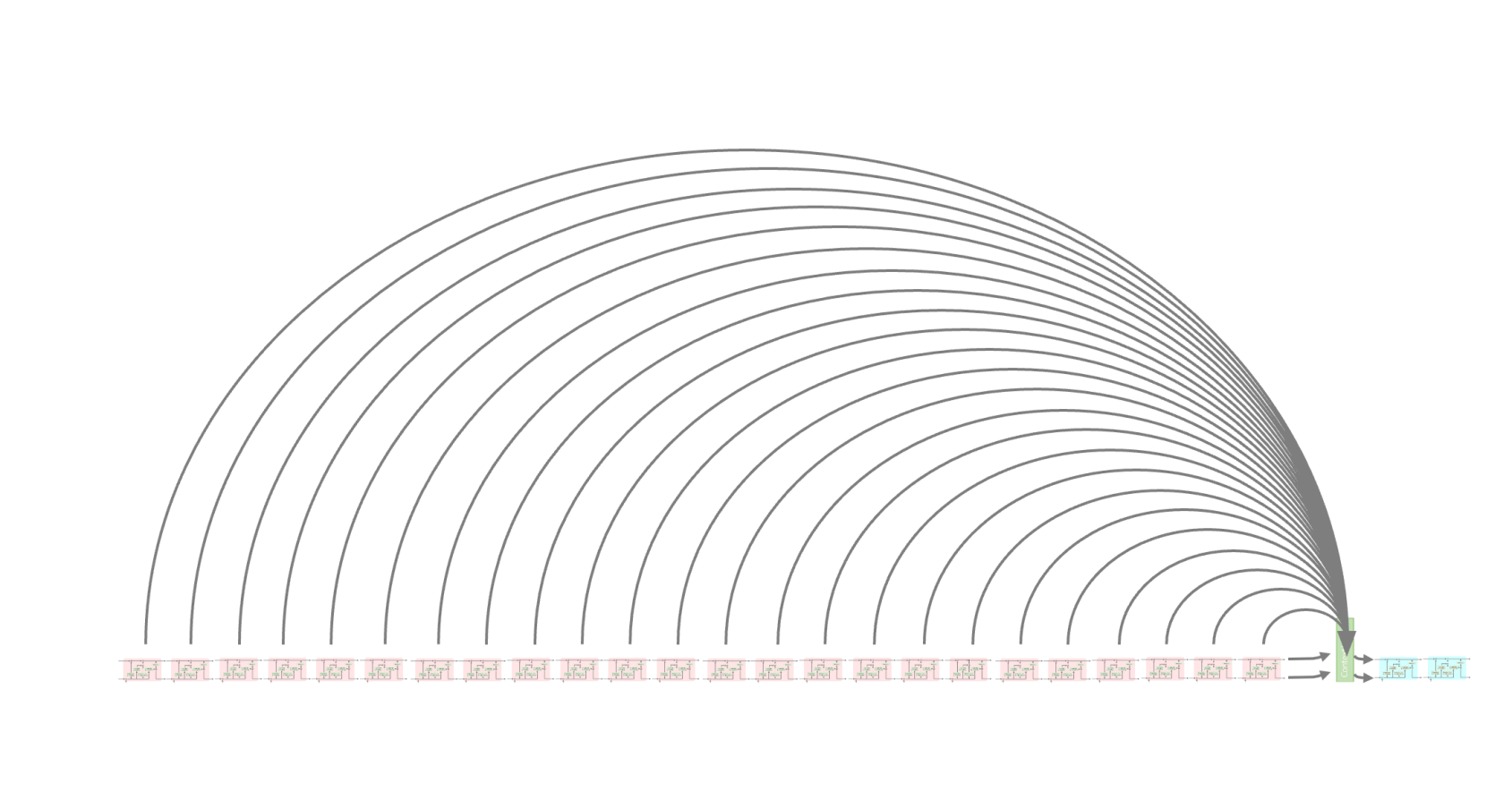
It becomes quite difficult to pack all the information of the input sequence into a fixed-length context vector.



This is called **Bottleneck** problem.

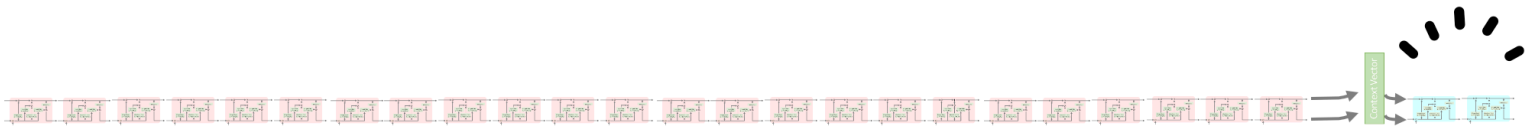
Seq2Seq: Problem

The **attention** mechanism was introduced to address this problem.



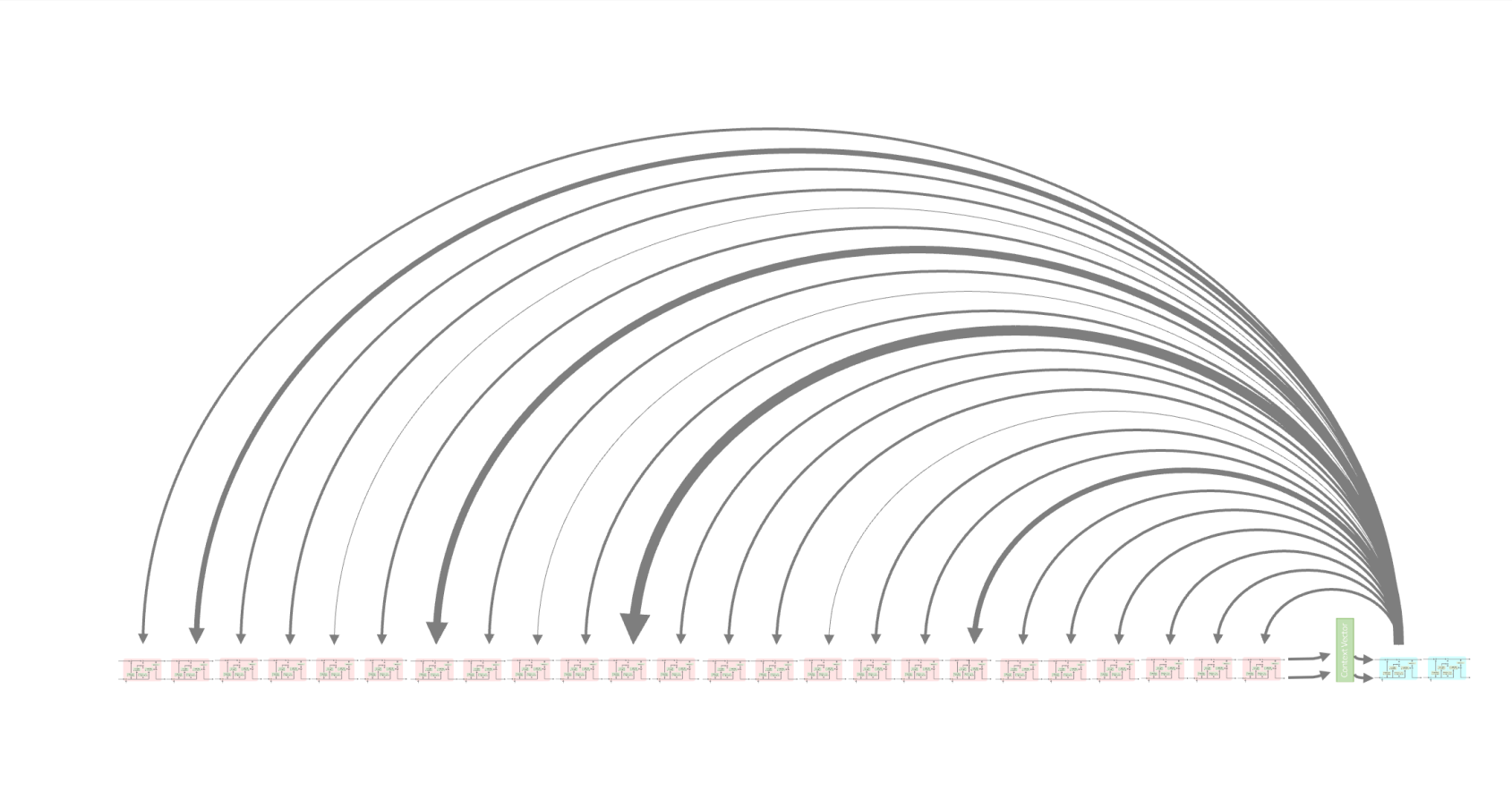
Seq2Seq: Problem

When the decoder generates each word in the output sequence, the attention mechanism



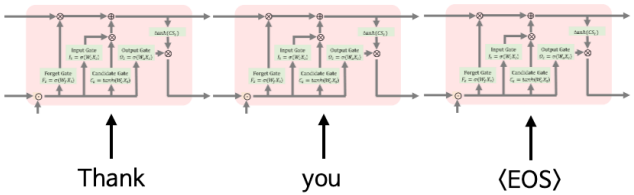
Seq2Seq: Problem

is an algorithm that makes it “attend” to which parts of the input sequence are important.



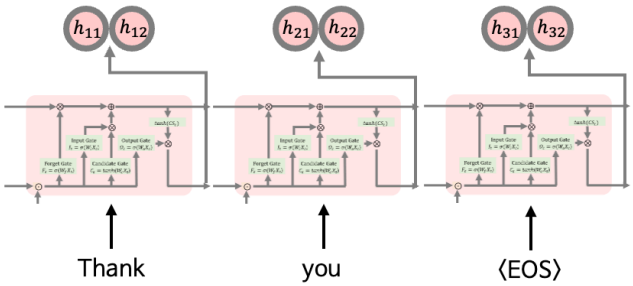
Attention

Suppose the input sequence comes in like this:



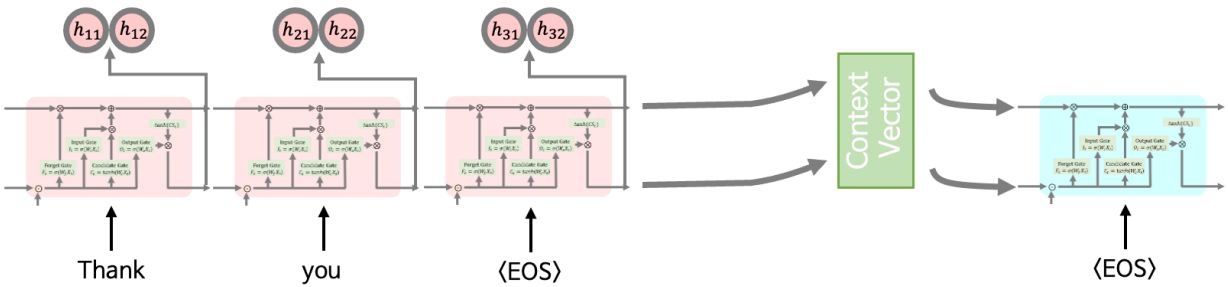
Attention

We store the hidden state for each input word separately.



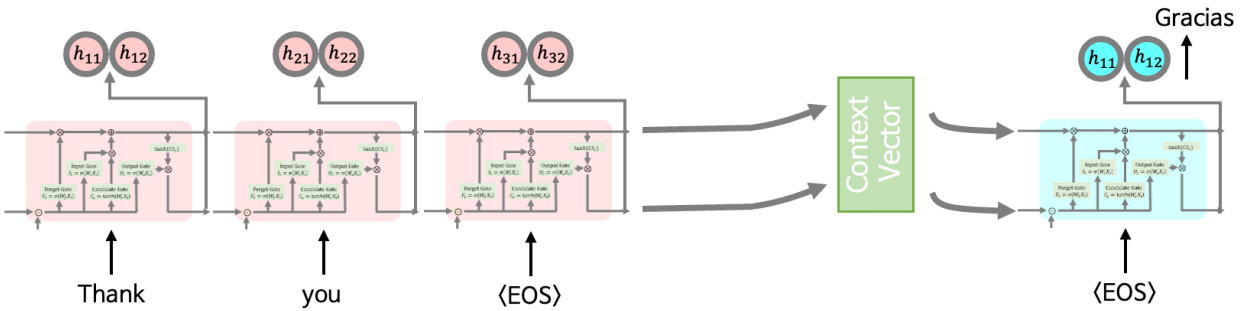
Attention

We build a context vector and feed it to the decoder,



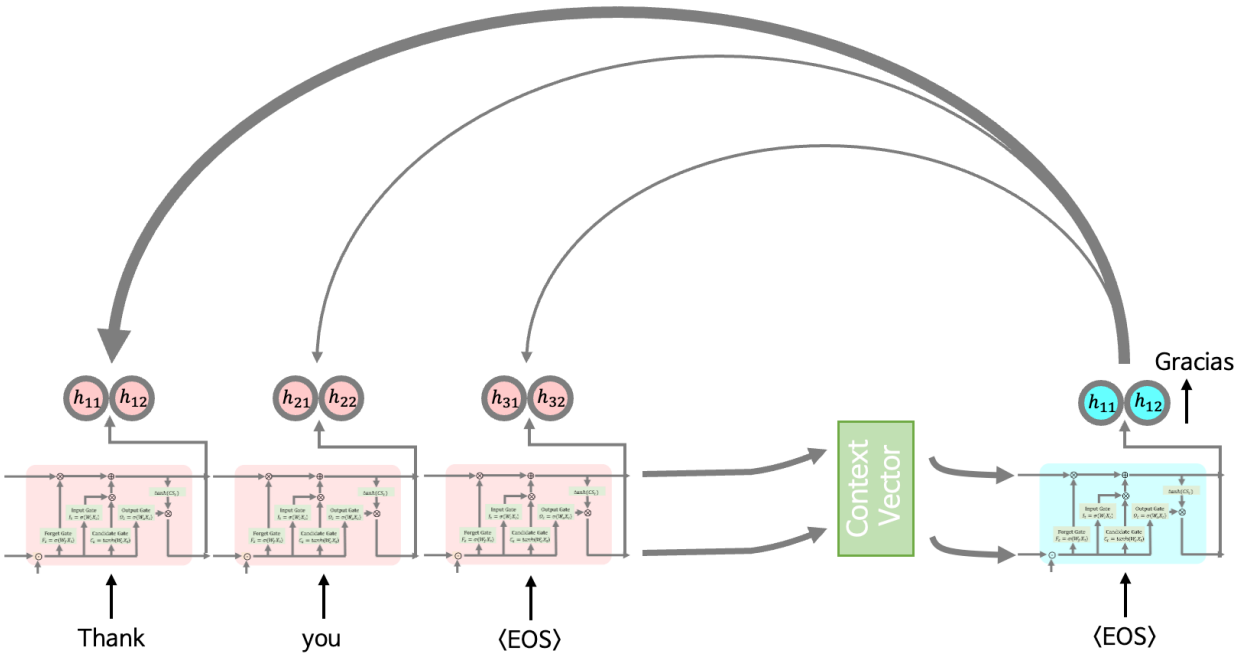
Attention

and obtain the decoder's hidden state and output as follows.



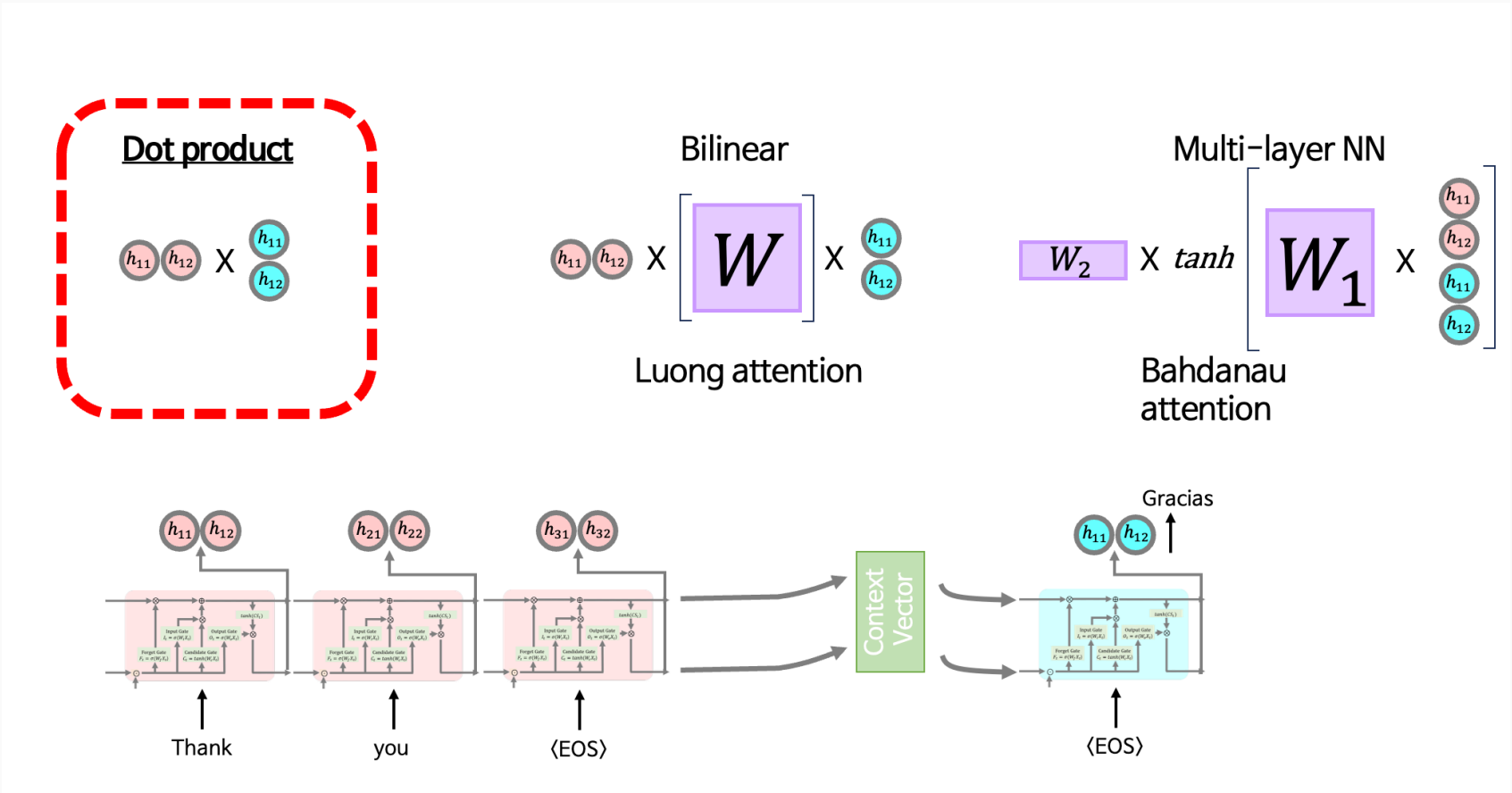
Attention

The **similarity** between two vectors works as a measure that determines the relationship between two states.



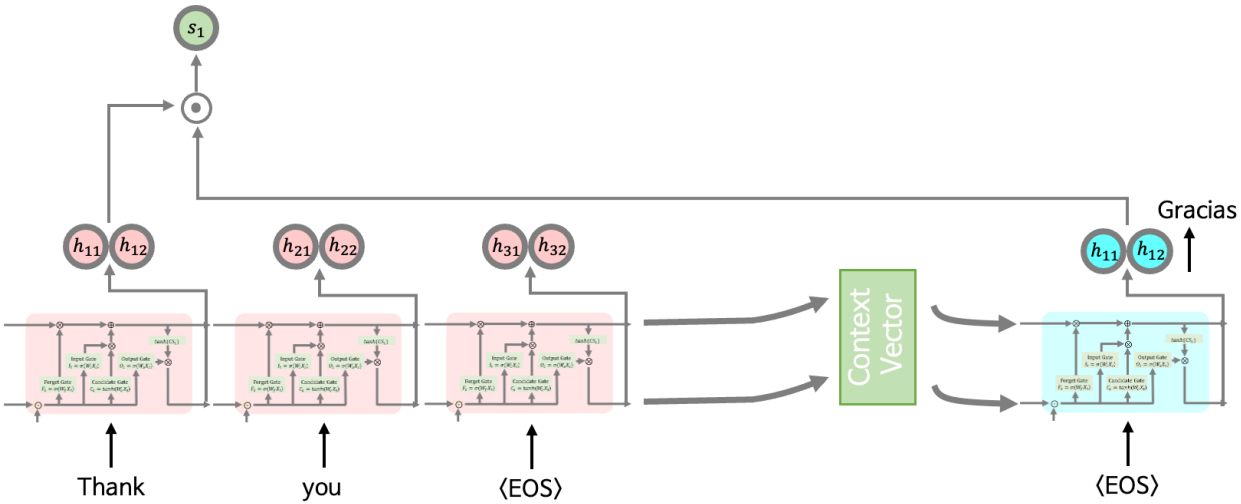
Attention

We focused on the simplest method: the dot product.



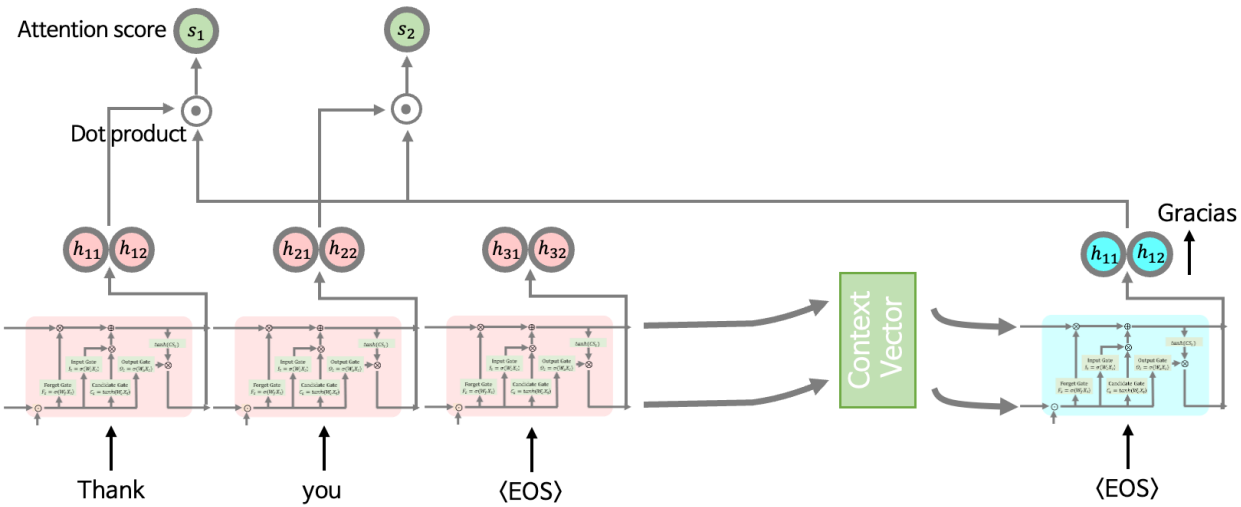
Attention

For example:



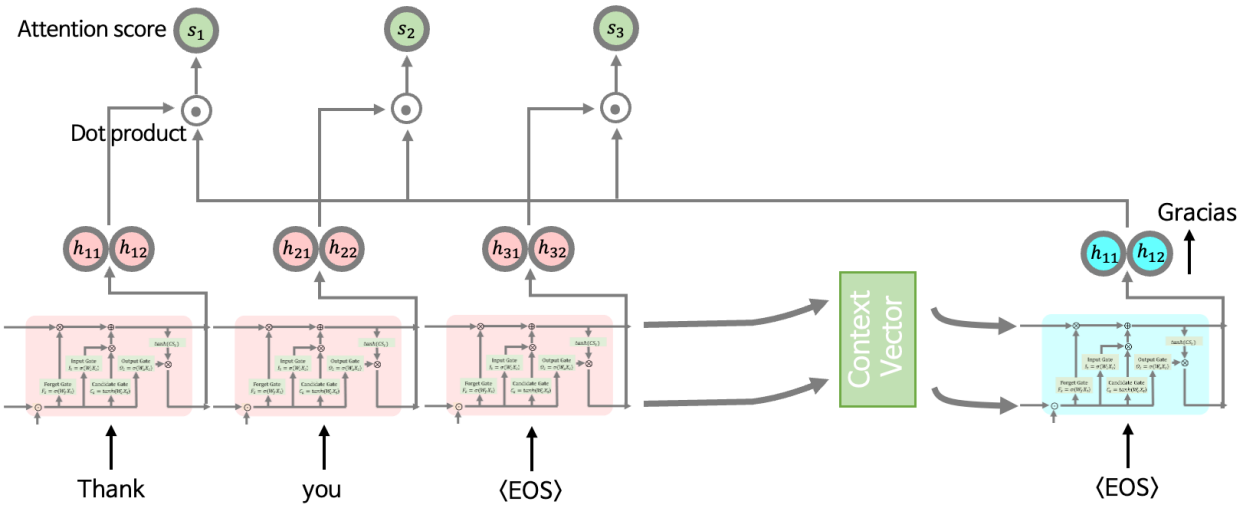
Attention

For example:



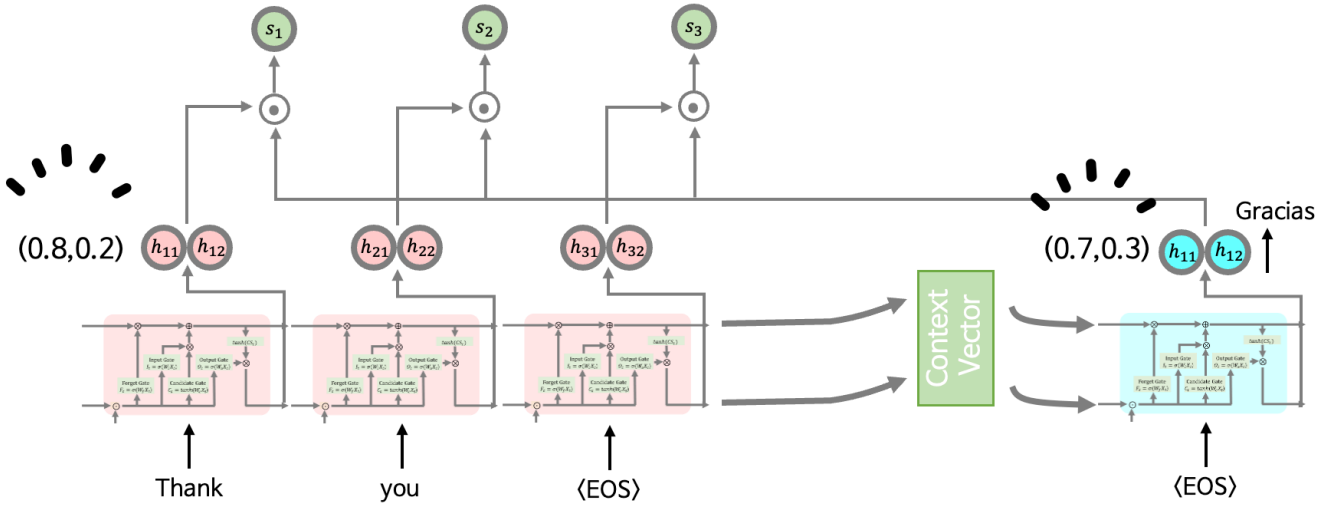
Attention

For example:



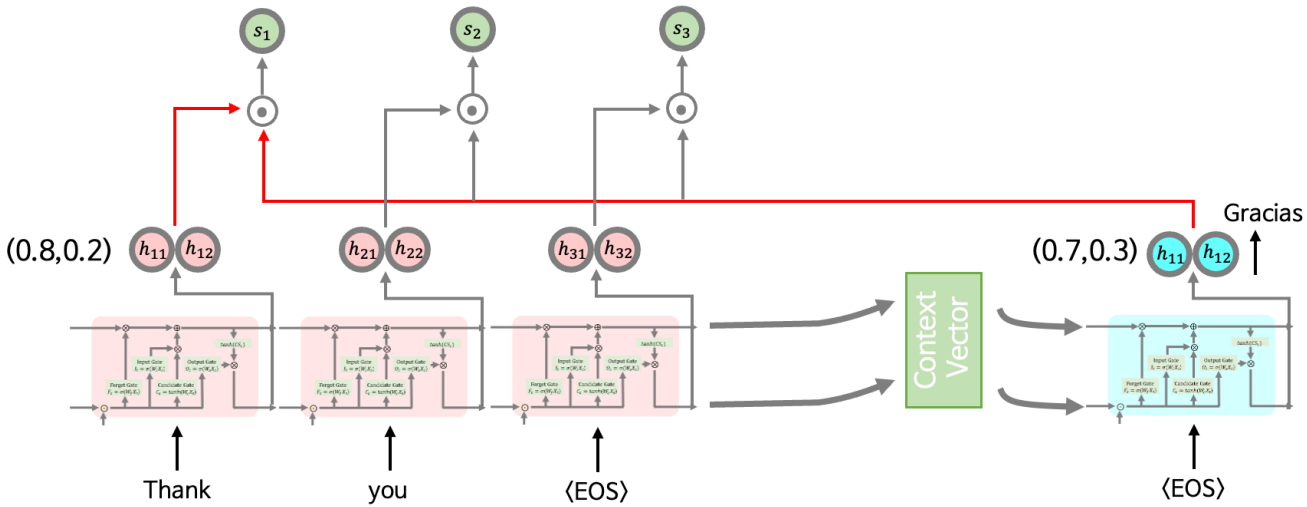
Attention

Suppose the encoder hidden state is (0.8, 0.2) and the decoder hidden state is (0.7, 0.3).



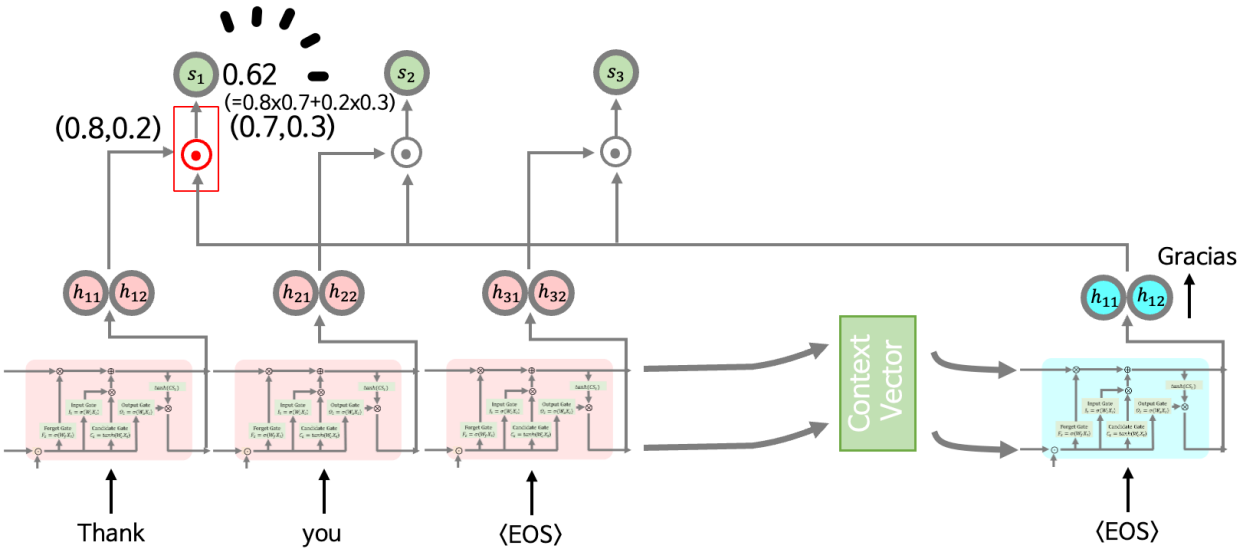
Attention

Suppose the encoder hidden state is (0.8, 0.2) and the decoder hidden state is (0.7, 0.3).



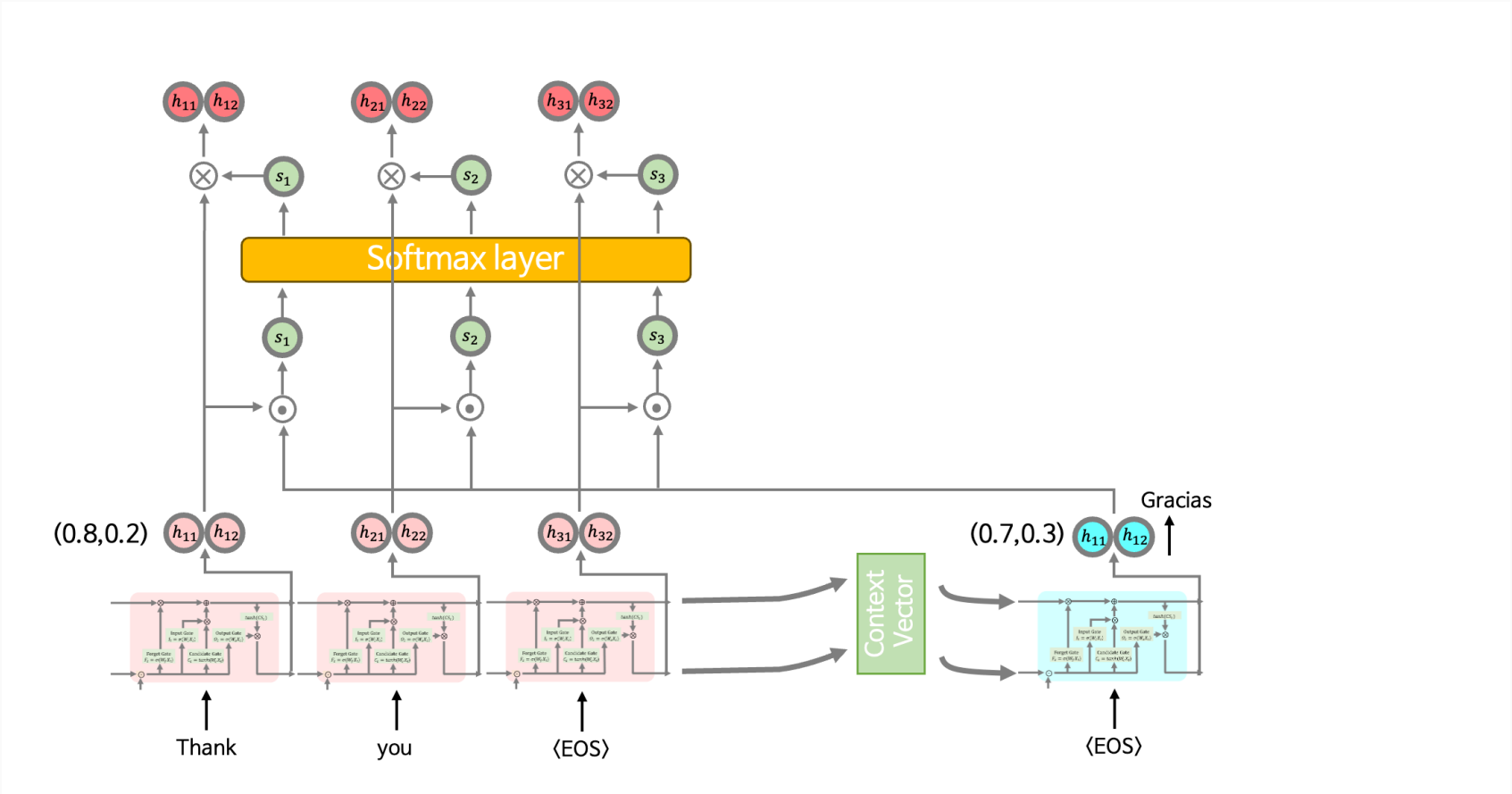
Attention

Then the attention score s1 becomes 0.62 through the following dot-product computation.



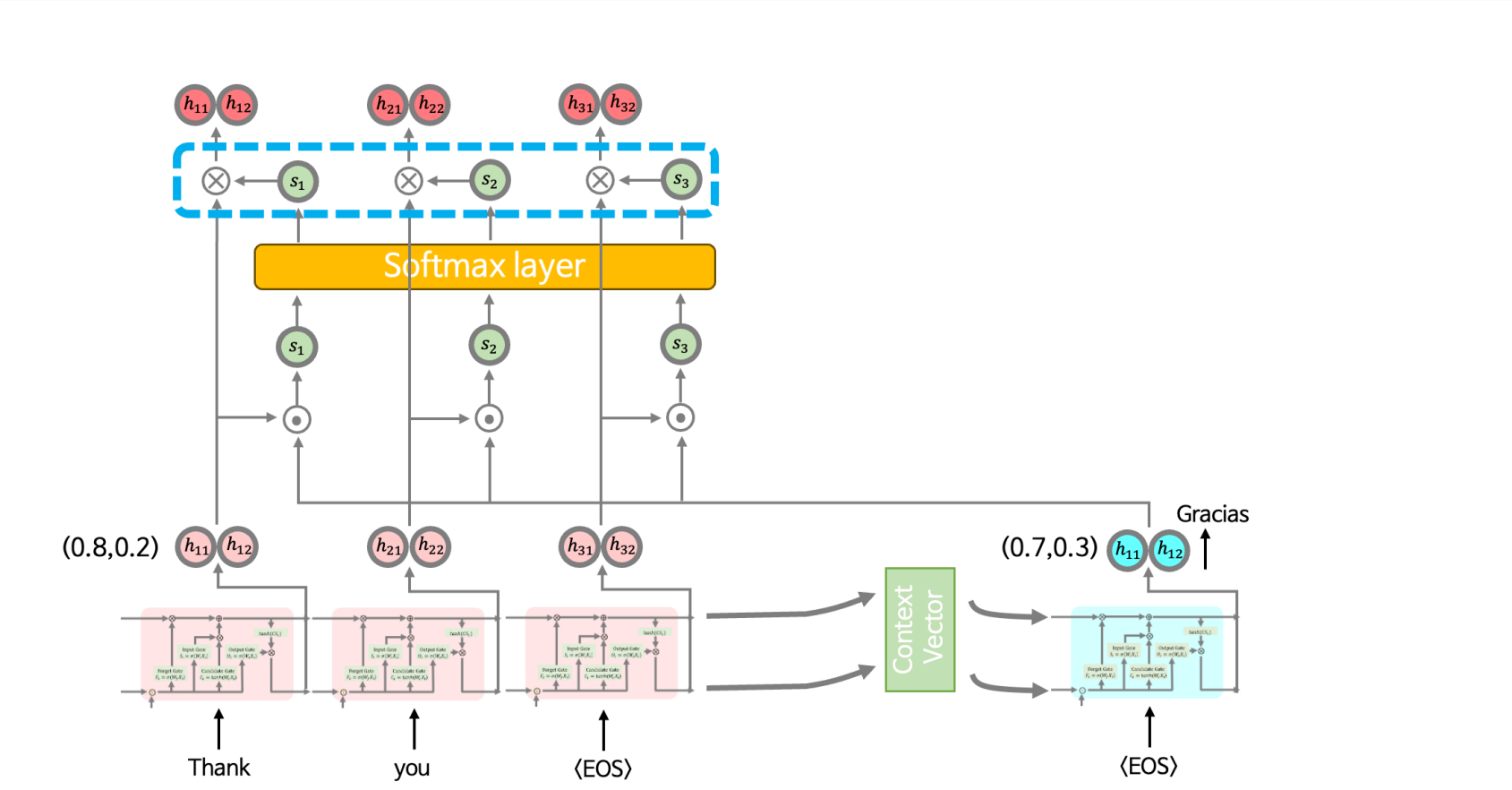
Attention

Next, we compute the **softmax** of each attention score



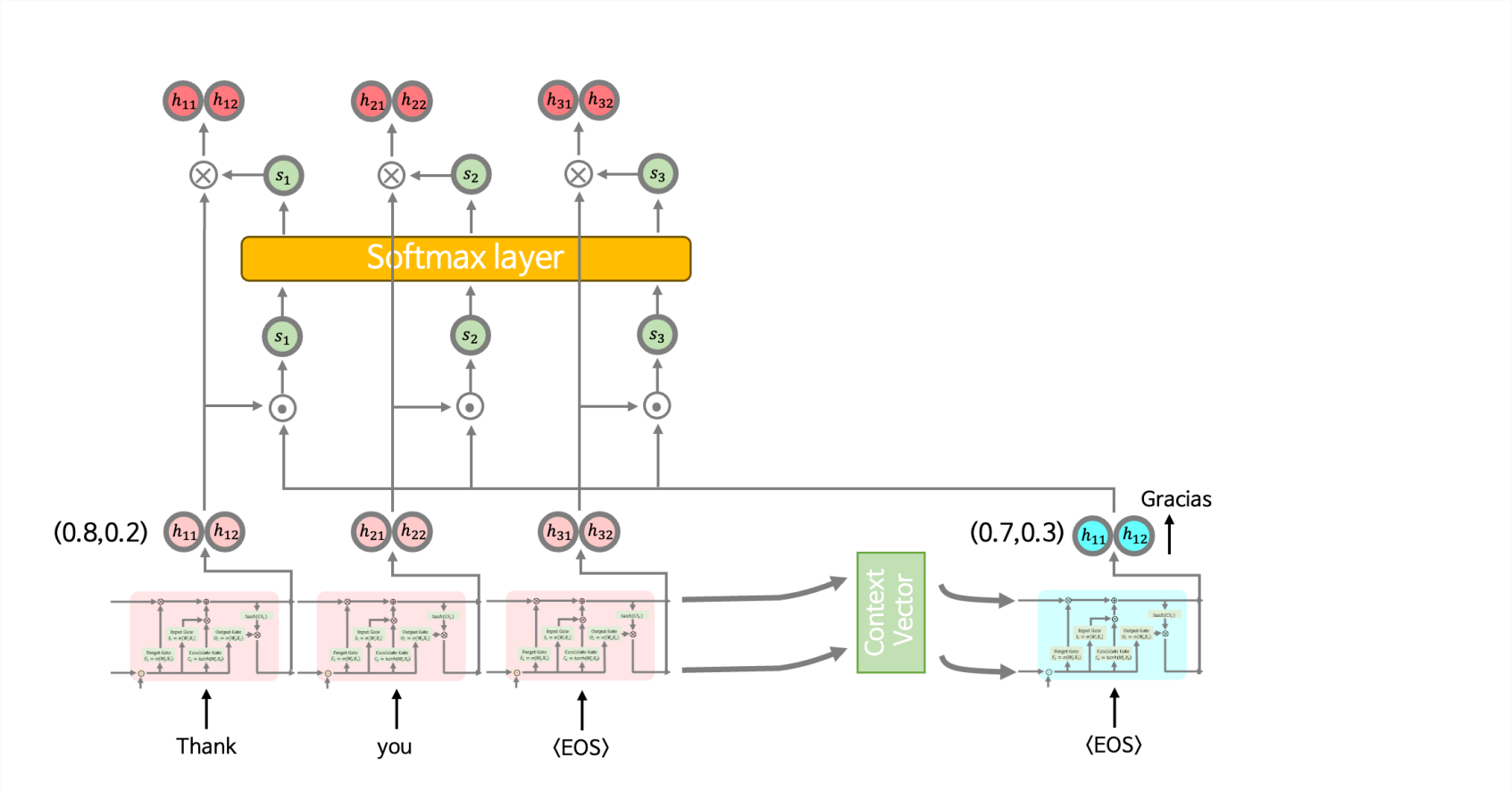
Attention

to convert the attention scores into a **probability distribution** and **normalize** them.



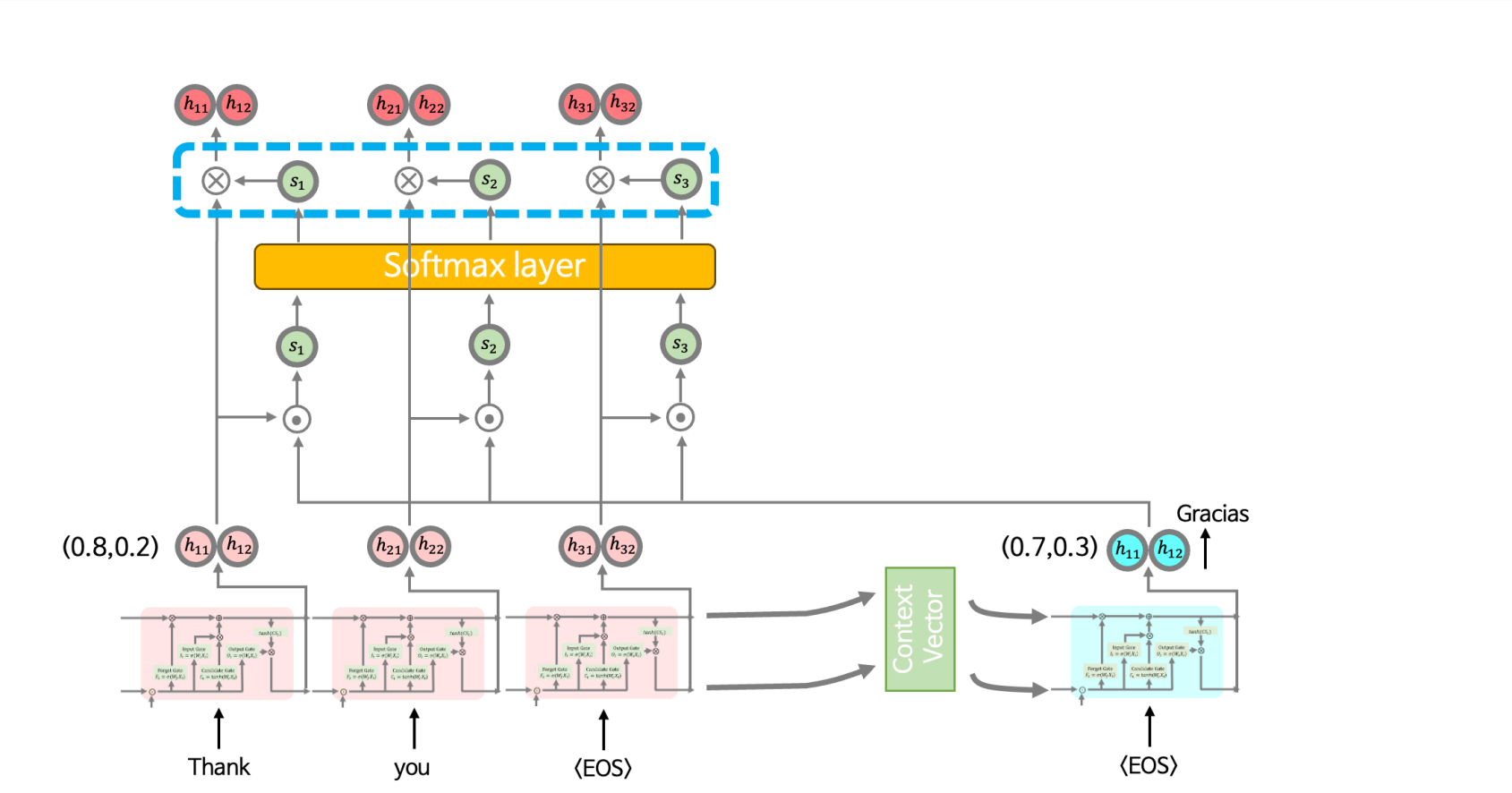
Attention

Next, we multiply each attention score by its corresponding input hidden state.



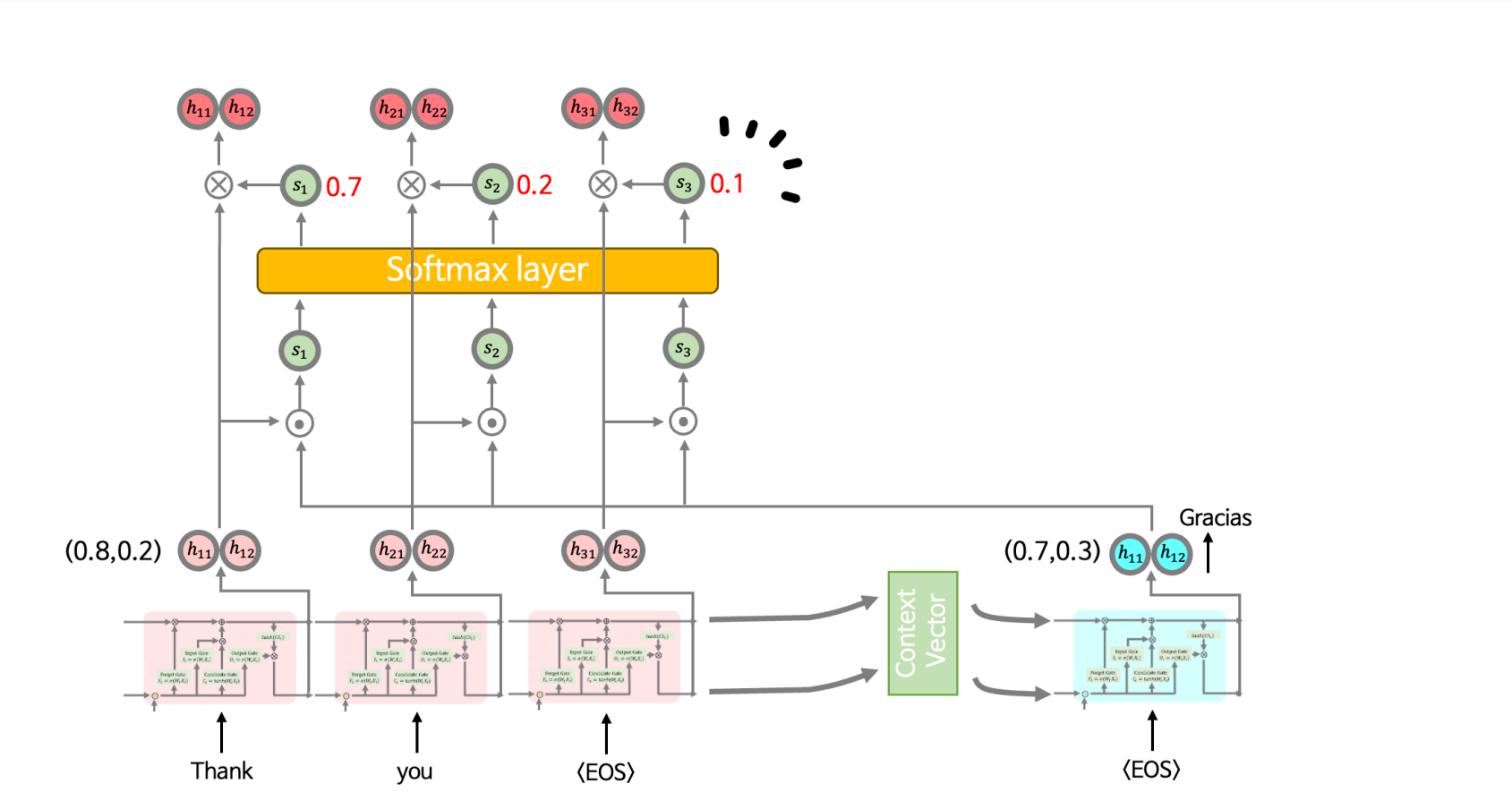
Attention

The multiplication has the effect of **amplifying** the input hidden states according to their attention weights.



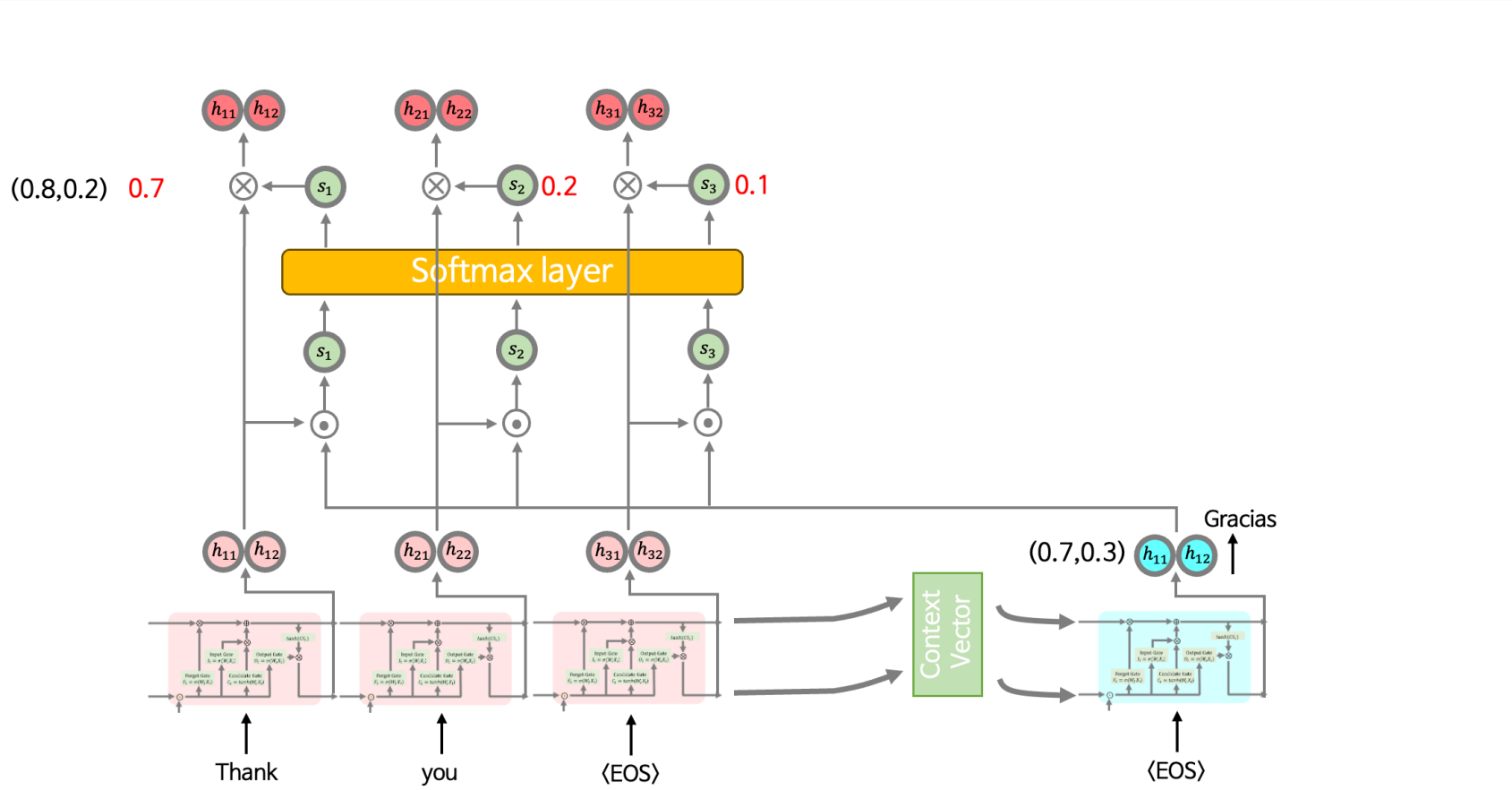
Attention

For example, if the attention scores after the softmax layer are as follows—



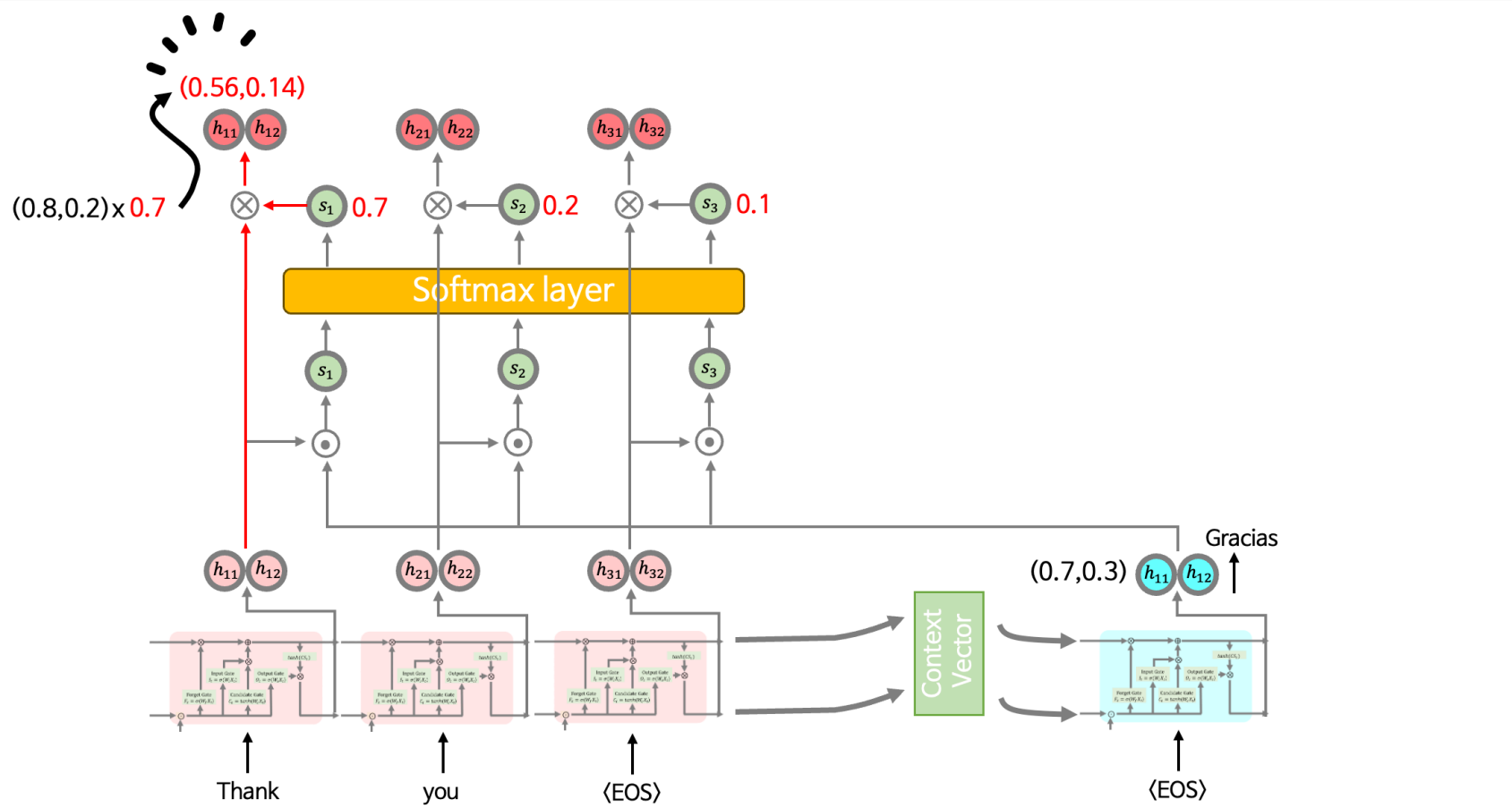
Attention

For example, if the attention scores after the softmax layer are as follows—



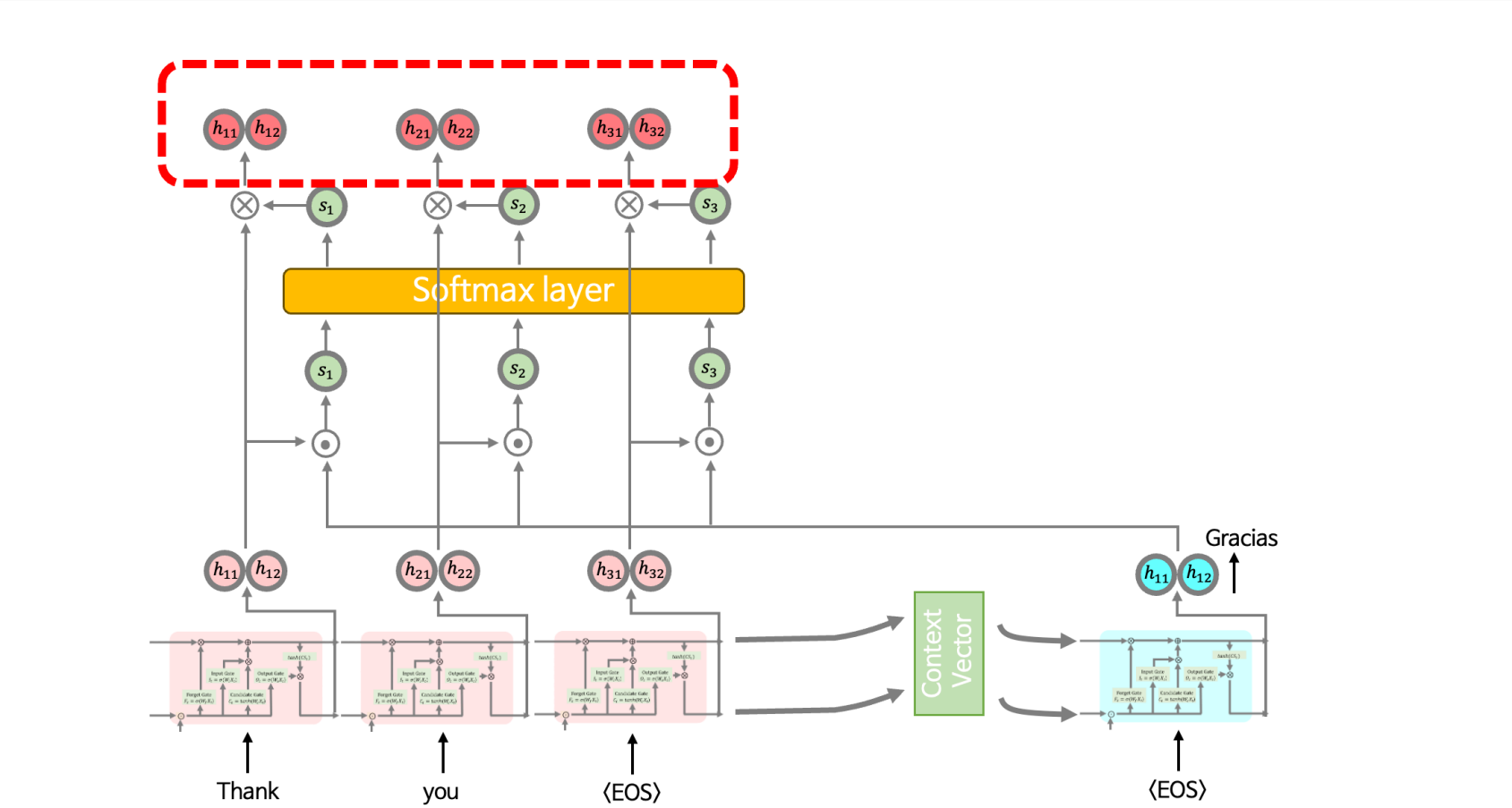
Attention

Then the input state (0.8, 0.2) multiplied by 0.7 becomes (0.56, 0.14).



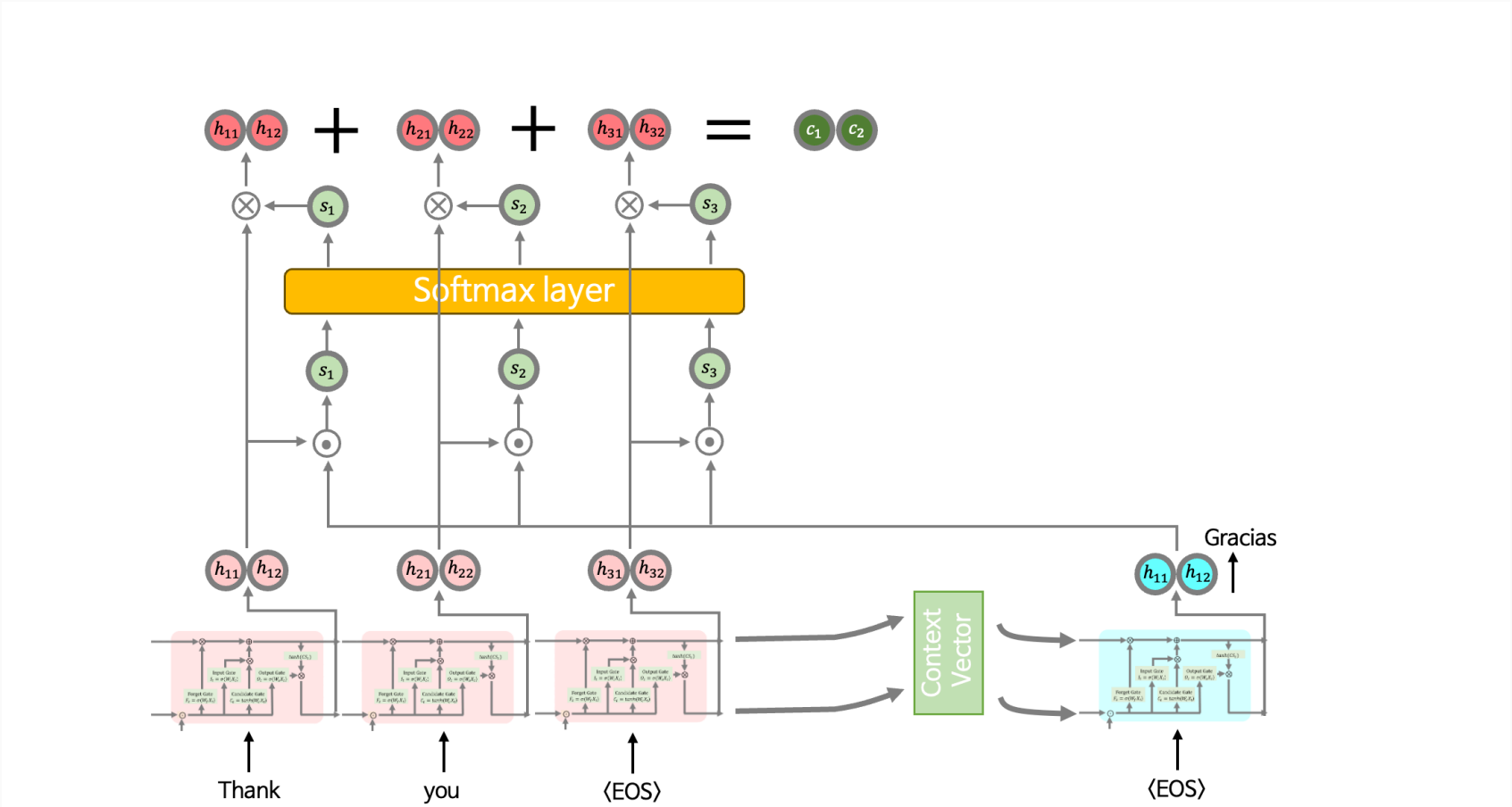
Attention

Now, take these **attention-weighted** input hidden states,



Attention

sum them up, and you obtain a **new context vector**.



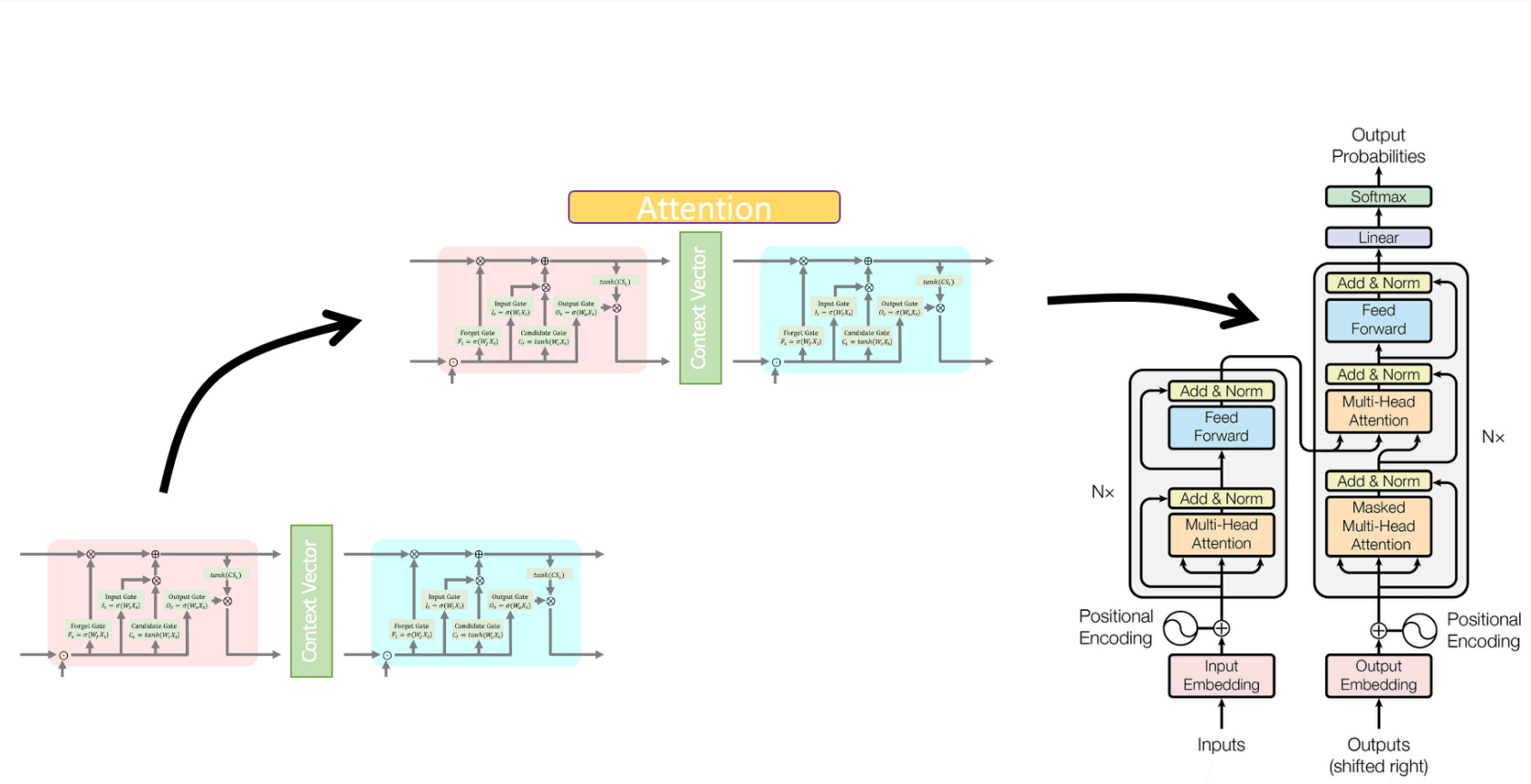
Attention is a *general* deep learning technique

- Attention has become the powerful, flexible, general way pointer and memory manipulation in deep learning models. (A new idea from 2010).

The transformer model

Recall

Eventually led to the development of the **Transformer** model.



Attention in transformers

Multi-head attentions
`https://www.youtube.com/watch?v=eMlx5fFNoYc` Any hints from the video clip?

RNNs: Long-dependency problem

- RNNs process input **sequentially**, one token at a time, passing information through hidden states.

RNNs: Long-dependency problem

- RNNs process input **sequentially**, one token at a time, passing information through hidden states.
- We discussed the long-dependency problem.

RNNs: Long-dependency problem

- RNNs process input **sequentially**, one token at a time, passing information through hidden states.
- We discussed the long-dependency problem.
- Somewhat solved through attention algorithms?

RNNs: Lack of parallelizability

- RNNs process input step by step — each hidden state depends on the previous one.

RNNs: Lack of parallelizability

- RNNs process input step by step — each hidden state depends on the previous one.
- This means both the forward and backward passes require O(sequence length) sequential operations.

RNNs: Lack of parallelizability

- RNNs process input step by step — each hidden state depends on the previous one.
- This means both the forward and backward passes require O(sequence length) sequential operations.
- GPUs are great at performing many independent computations in parallel, but RNNs don’t allow this because future states can’t be computed until past states are done.

RNNs: Lack of parallelizability

- RNNs process input step by step — each hidden state depends on the previous one.
- This means both the forward and backward passes require O(sequence length) sequential operations.
- GPUs are great at performing many independent computations in parallel, but RNNs don’t allow this because future states can’t be computed until past states are done.
- As a result, training RNNs on very large datasets becomes **slow and inefficient**.

RNNs+Attention!

- Seq2Seq (RNN) models: Attention connects the decoder to the encoder — each decoder step selectively focuses on encoder hidden states.

  (Cross-attention only; still sequential)

RNNs+Attention!

- Seq2Seq (RNN) models: Attention connects the decoder to the encoder — each decoder step selectively focuses on encoder hidden states.

  (Cross-attention only; still sequential)
- In Transformers, attention also occurs **within a single sentence** — all words attend to all words in the previous layer.

  (Self-attention + Cross-attention)

RNNs+Attention!

- Seq2Seq (RNN) models: Attention connects the decoder to the encoder — each decoder step selectively focuses on encoder hidden states.

  (Cross-attention only; still sequential)
- In Transformers, attention also occurs **within a single sentence** — all words attend to all words in the previous layer.

  (Self-attention + Cross-attention)
- This means every word can interact with every other word directly — no need to wait for sequential computation.

RNNs+Attention!

- Seq2Seq (RNN) models: Attention connects the decoder to the encoder — each decoder step selectively focuses on encoder hidden states.
  (Cross-attention only; still sequential)
- In Transformers, attention also occurs **within a single sentence** — all words attend to all words in the previous layer.
  (Self-attention + Cross-attention)
- This means every word can interact with every other word directly — no need to wait for sequential computation.
- As a result, Transformers overcome both long-distance dependency and lack of parallelizability.

RNNs+Attention!

- Seq2Seq (RNN) models: Attention connects the decoder to the encoder — each decoder step selectively focuses on encoder hidden states.

  (Cross-attention only; still sequential)
- In Transformers, attention also occurs **within a single sentence** — all words attend to all words in the previous layer.

  (Self-attention + Cross-attention)
- This means every word can interact with every other word directly — no need to wait for sequential computation.
- As a result, Transformers overcome both long-distance dependency and lack of parallelizability.
- *Notes.* This was NOT an entirely new ways of looking NLP problems (e.g., probabilistic language models $\rightarrow$ neural network), but made a huge progress in the field.

Parallelization in Transformers

- **Self-Attention**: All tokens attend to all tokens in the same layer simultaneously.

Parallelization in Transformers

- **Self-Attention**: All tokens attend to all tokens in the same layer simultaneously.
- Unlike RNNs, Transformer layers do not depend on previous hidden states; each token's representation is updated in parallel.

Parallelization in Transformers

- **Self-Attention**: All tokens attend to all tokens in the same layer simultaneously.
- Unlike RNNs, Transformer layers do not depend on previous hidden states; each token's representation is updated in parallel.
- This allows GPUs to perform all attention computations at once using matrix multiplication.

Parallelization in Transformers

- **Self-Attention**: All tokens attend to all tokens in the same layer simultaneously.
- Unlike RNNs, Transformer layers do not depend on previous hidden states; each token's representation is updated in parallel.
- This allows GPUs to perform all attention computations at once using matrix multiplication.
- Training and inference are therefore much faster, especially for long sequences and large datasets.

From Dot Product to Soft Lookup

- Recall how we calculated the **dot product** between the decoder state (query) and each encoder hidden state (key) in the Seq2Seq attention.

From Dot Product to Soft Lookup

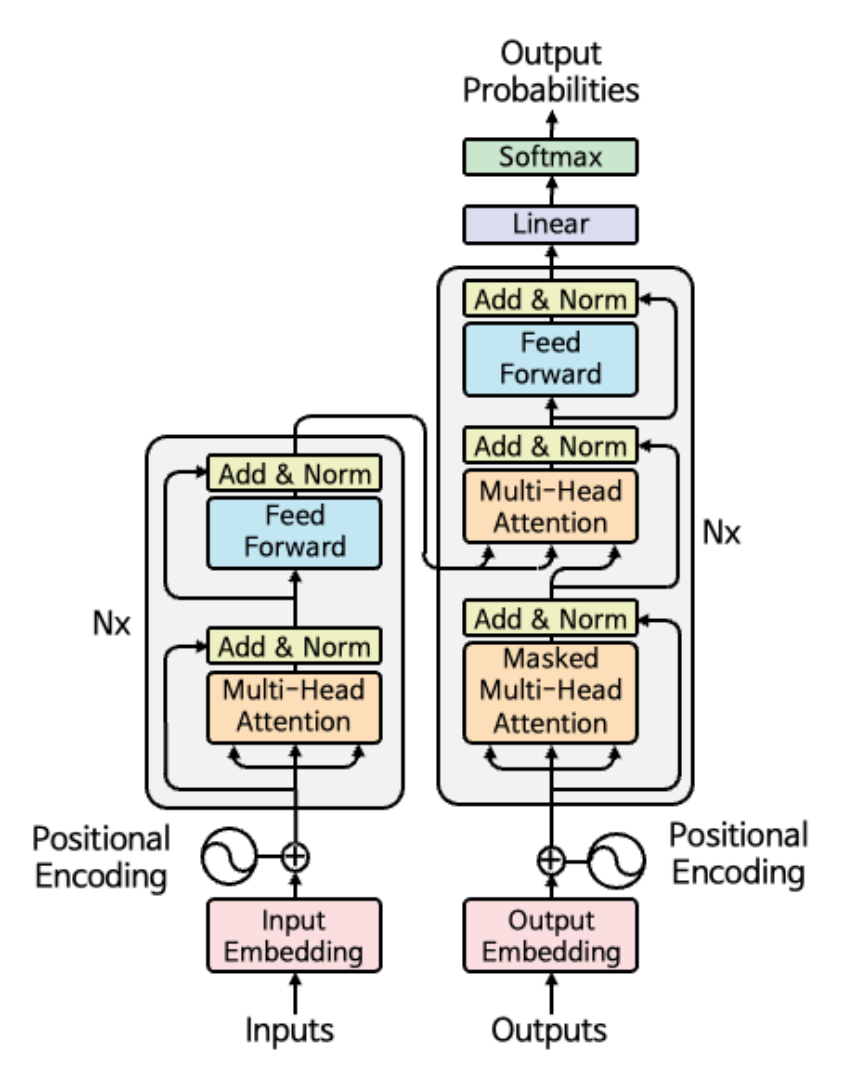
- Recall how we calculated the **dot product** between the decoder state (query) and each encoder hidden state (key) in the Seq2Seq attention.
- That dot product measured how similar the query was to each key — giving us attention weights after softmax.

From Dot Product to Soft Lookup

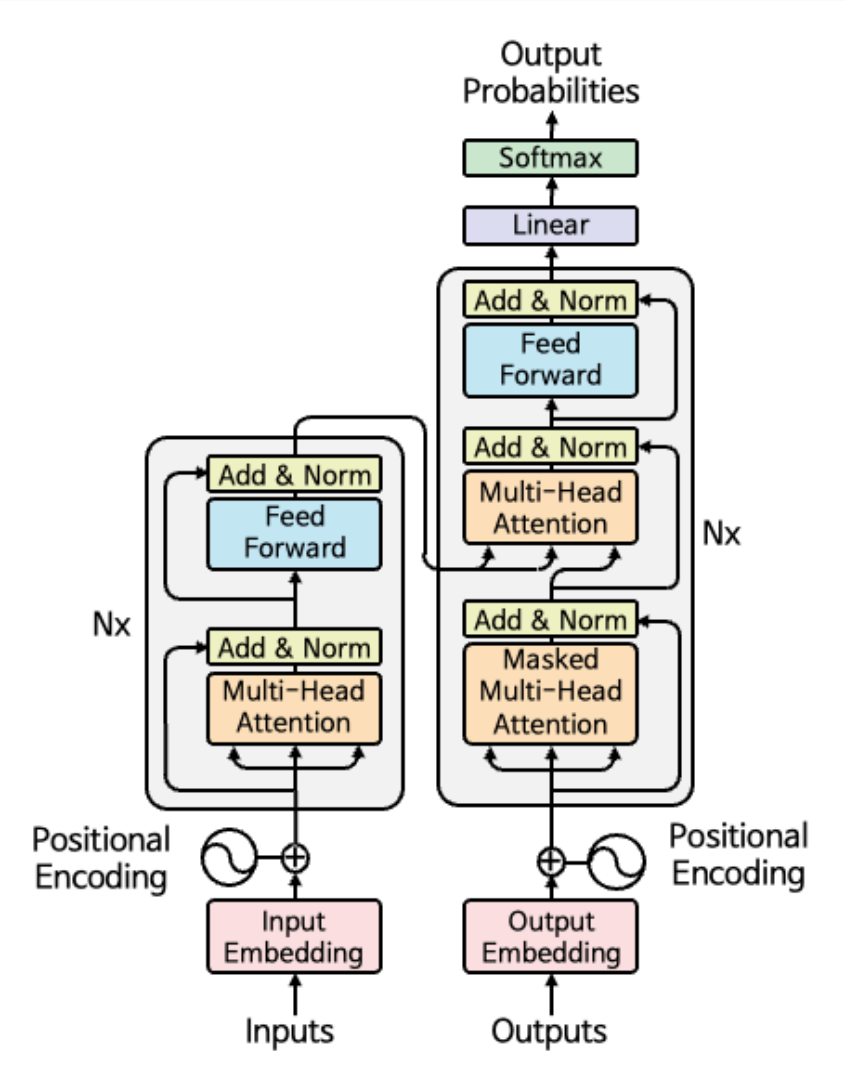
- Recall how we calculated the **dot product** between the decoder state (query) and each encoder hidden state (key) in the Seq2Seq attention.
- That dot product measured how similar the query was to each key — giving us attention weights after softmax.
- In Transformers, the same idea is extended: every token's representation acts as a **query**, looking up information from a set of keys and values (**soft, averaging lookup** in a key–value store) $\Rightarrow$ The dot product attention becomes a **fuzzy retrieval mechanism** that allows each token to access information from all others in parallel.

Understanding the Transformer with example

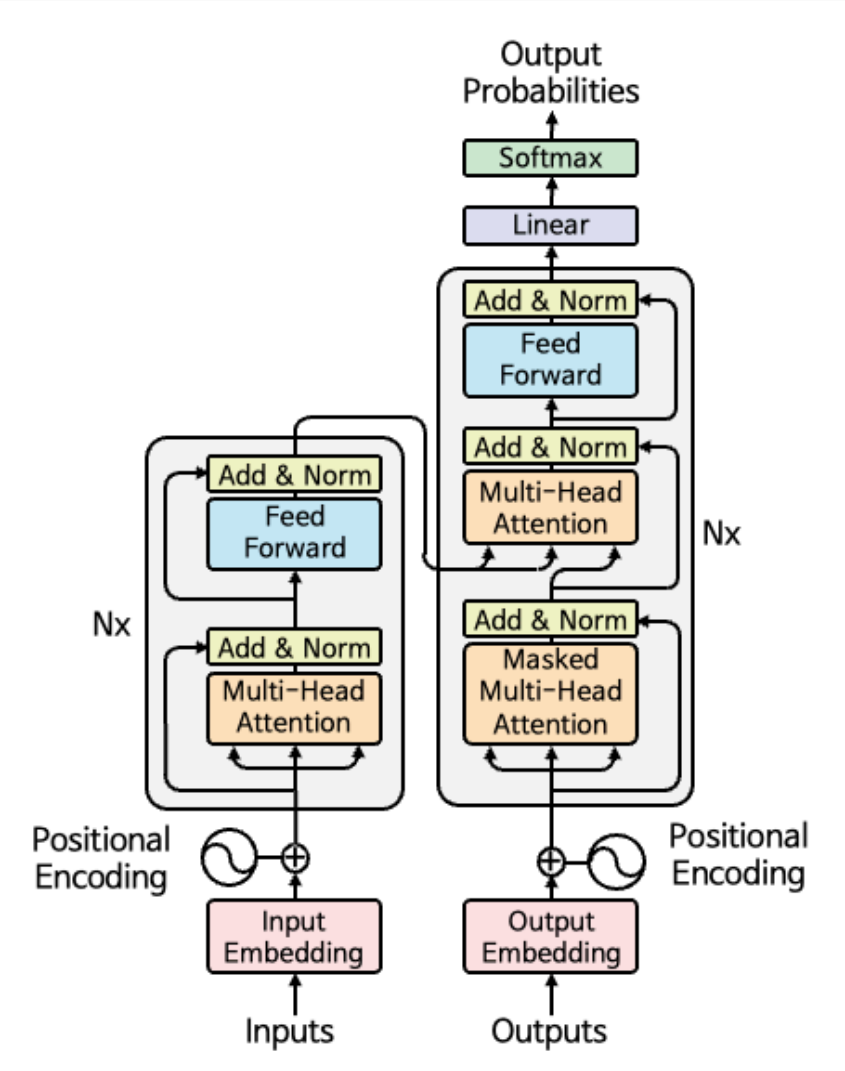
The structure of the Transformer (proposed by Vaswani et al., 2017) looks like this:



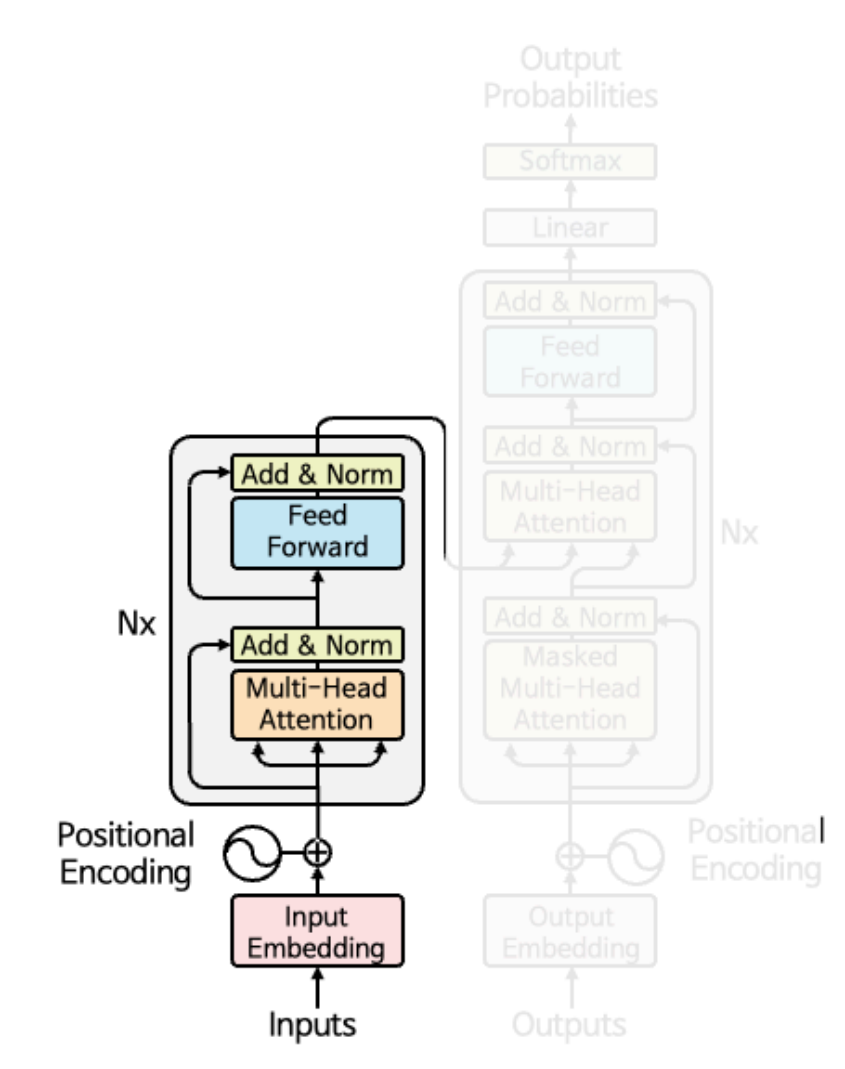
Before diving into the detailed calculations, let's first take a look at the overall structure.



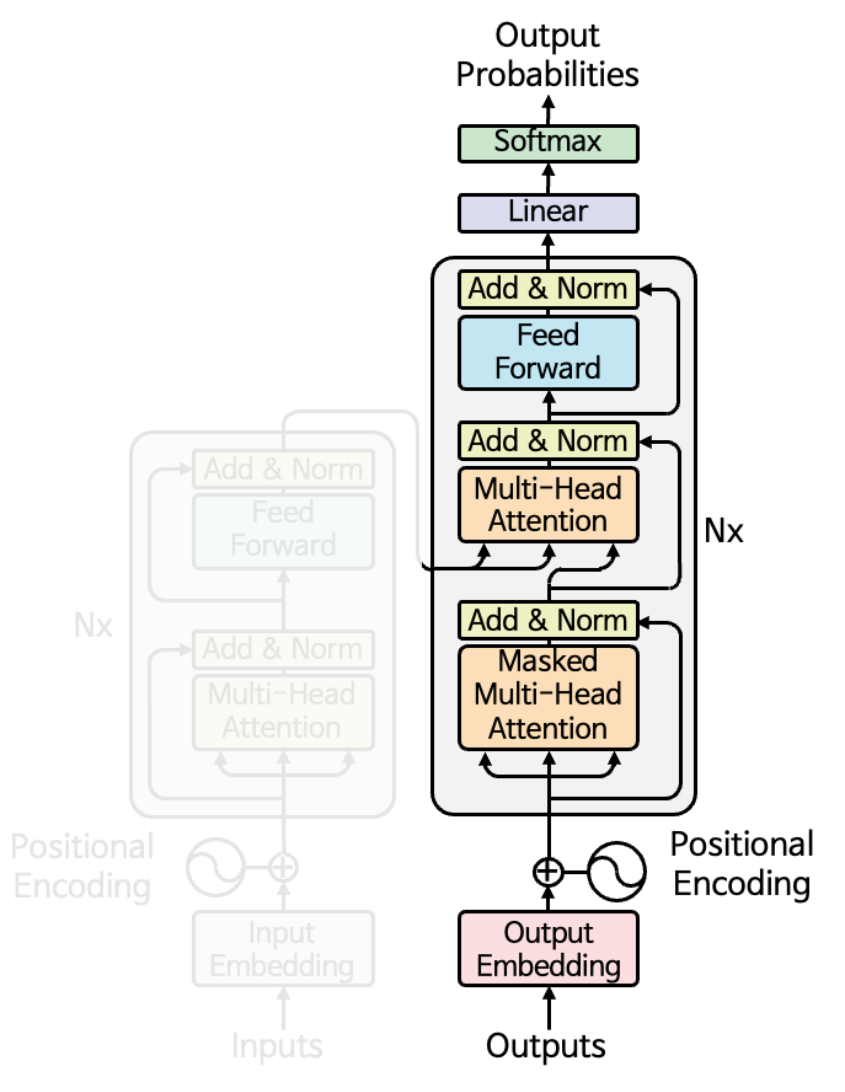
The Transformer model can be broadly divided into two main parts.



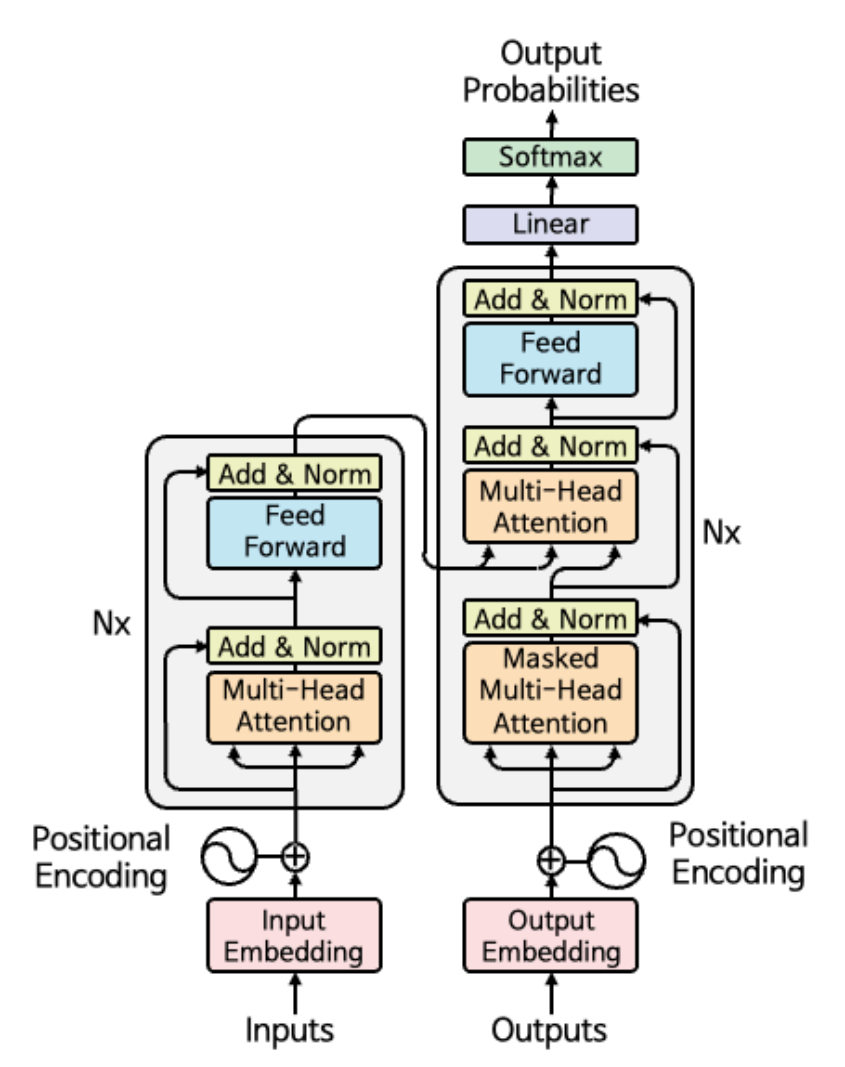
Here we see the **encoder** part,



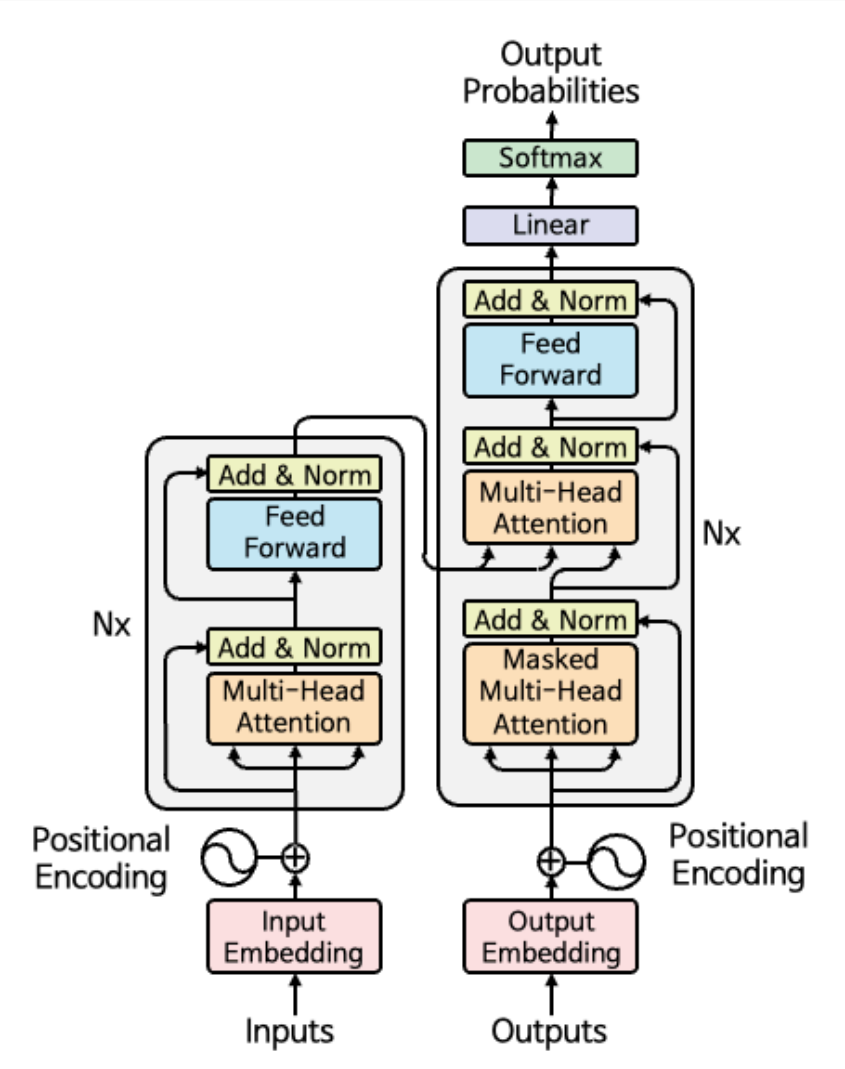
and the **decoder** part.



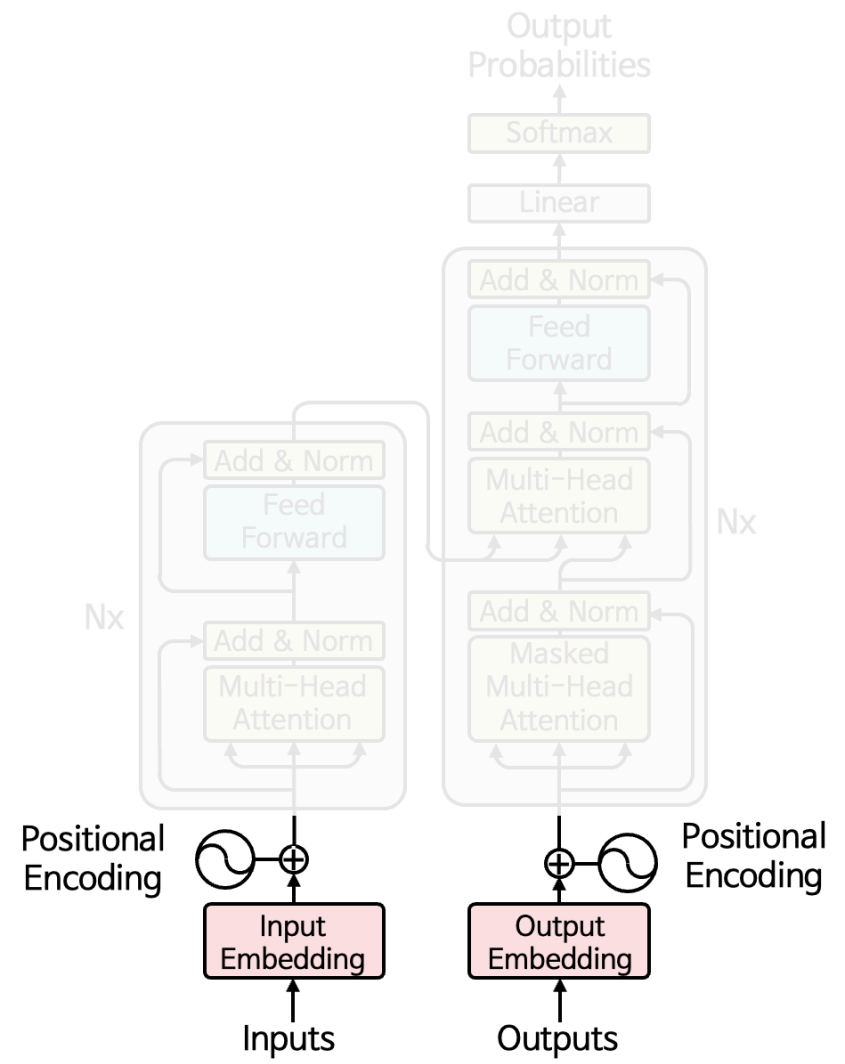
If we look closely at the entire Transformer model,



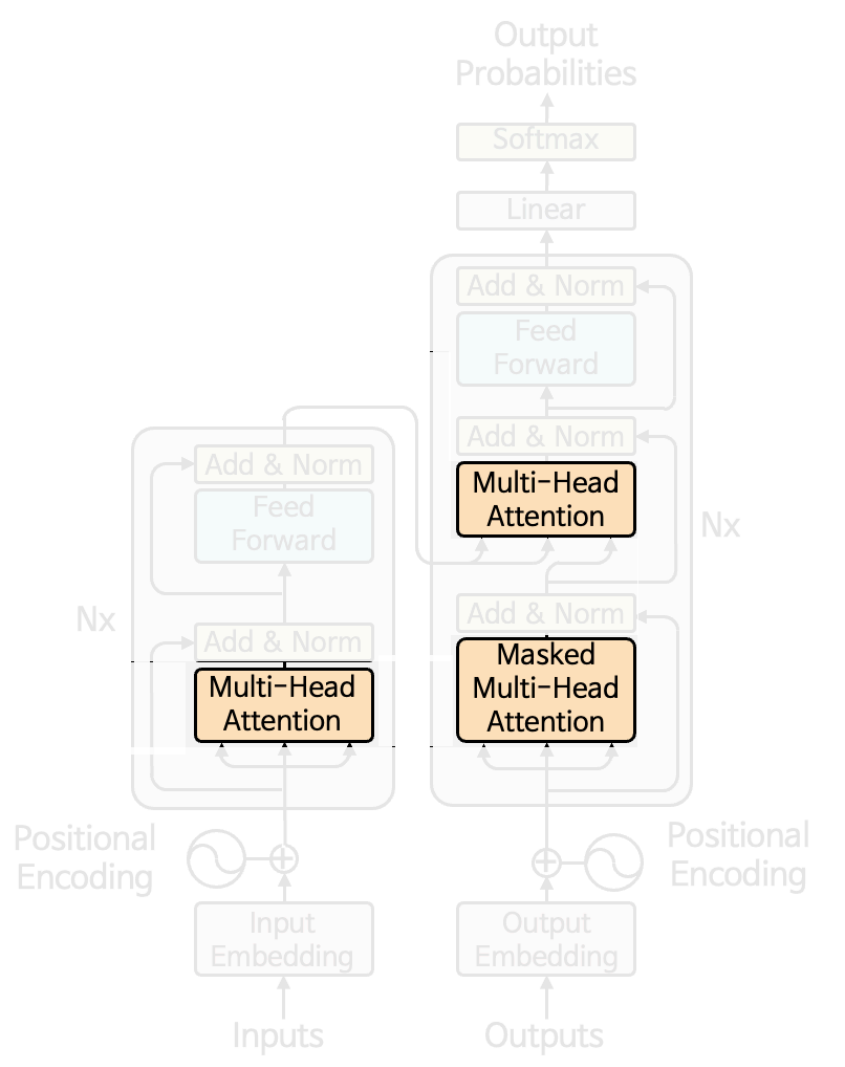
we can see that the same kinds of blocks are **repeatedly stacked**.



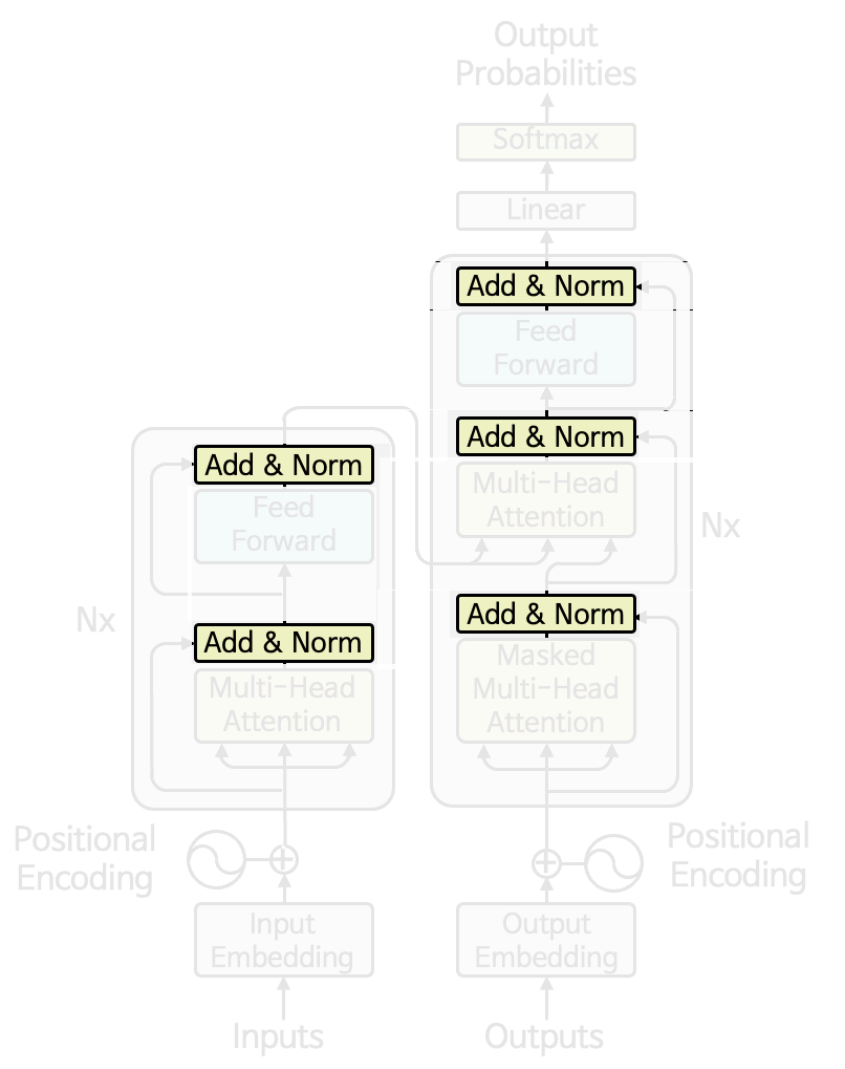
Embedding layer



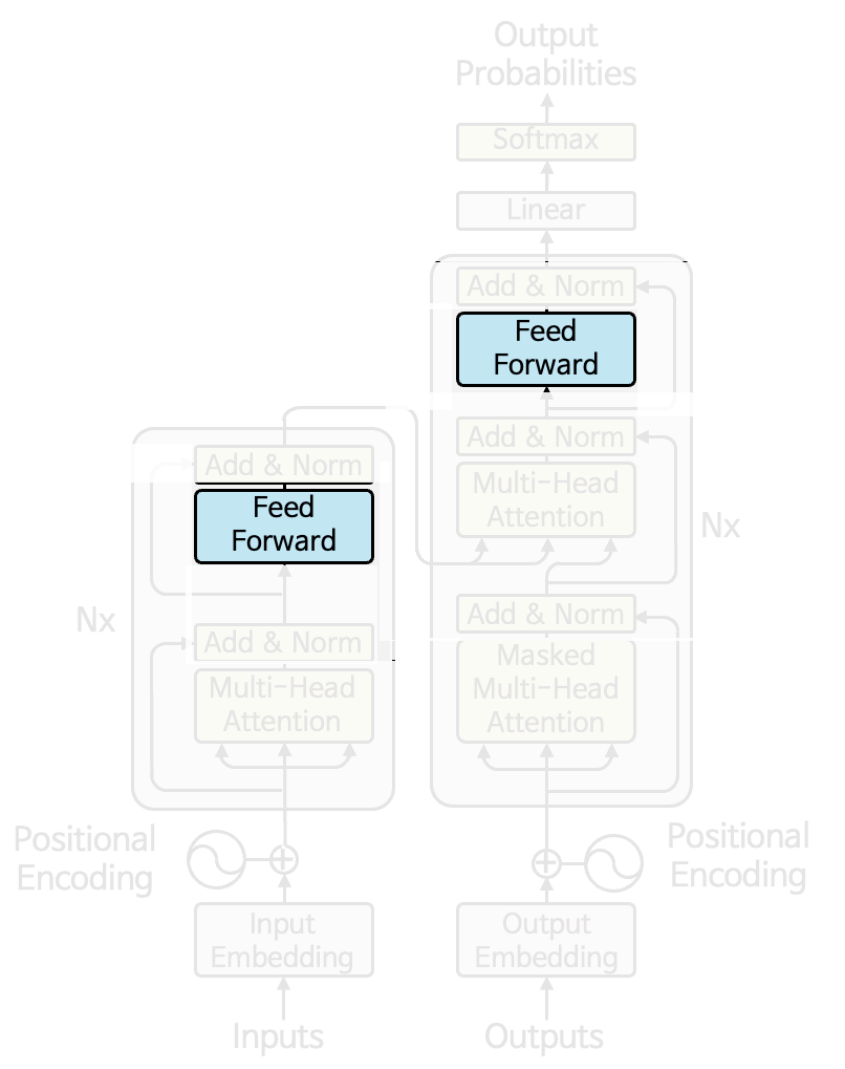
Multi-head attention



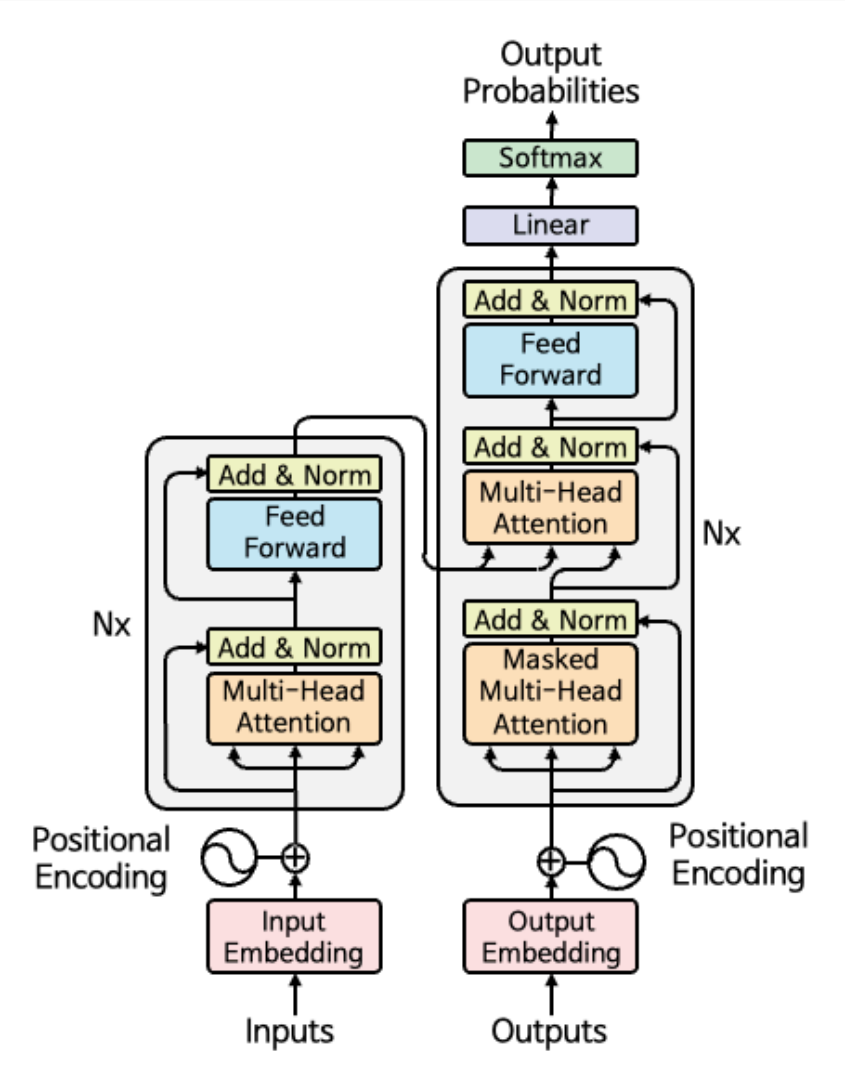
Add & Norm layer



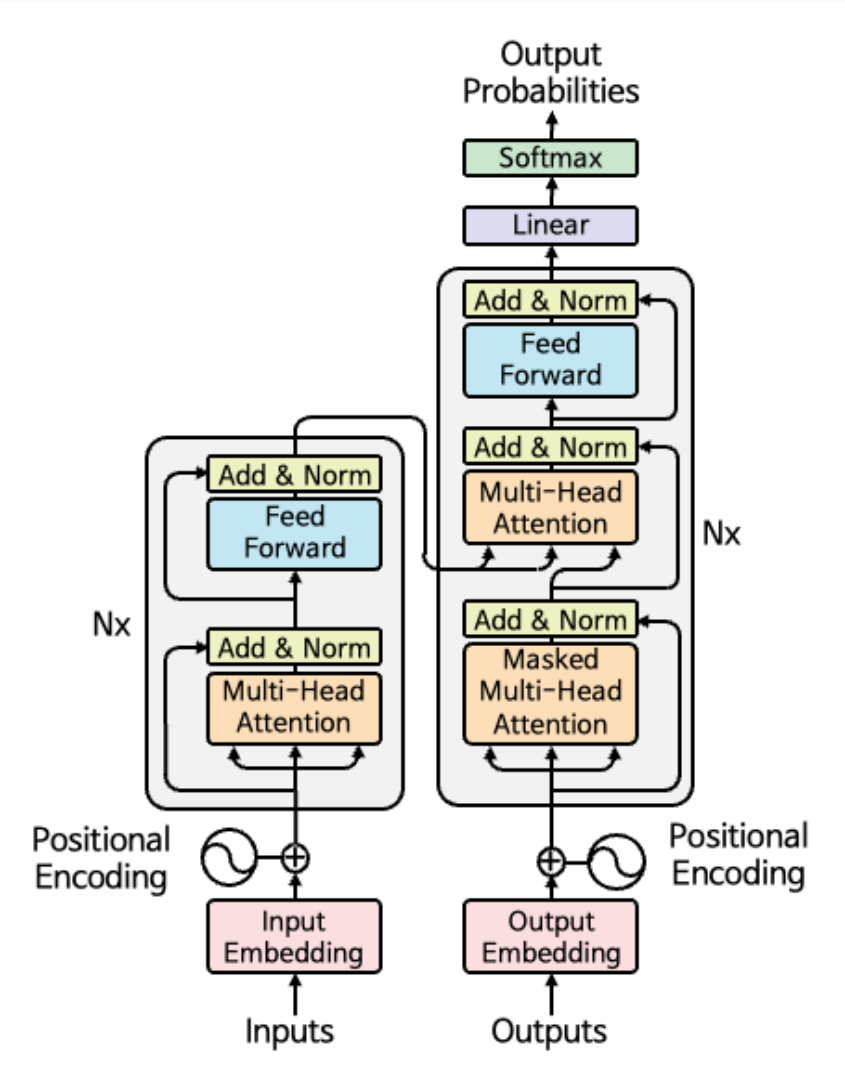
and also the Feed-Forward layer.



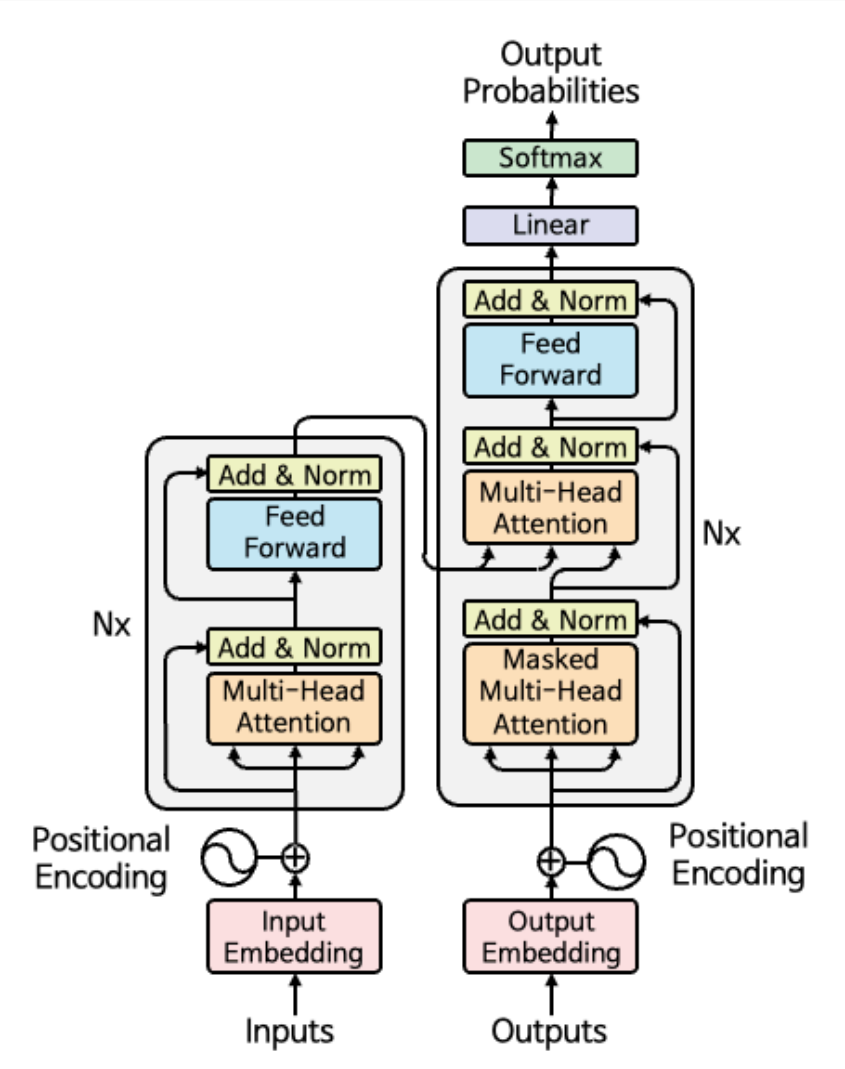
So, even though the Transformer may look complicated at first glance,



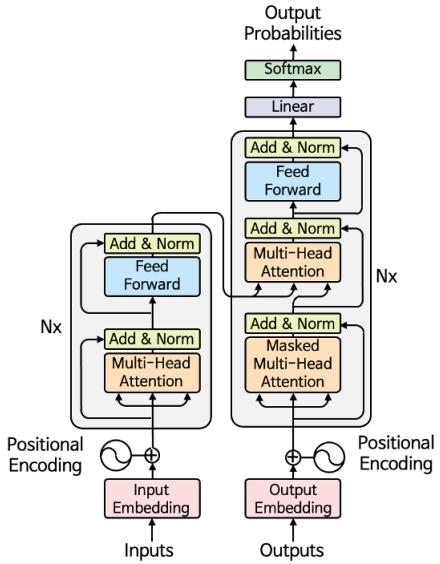
it’s actually made up of a few components that are repeatedly stacked.



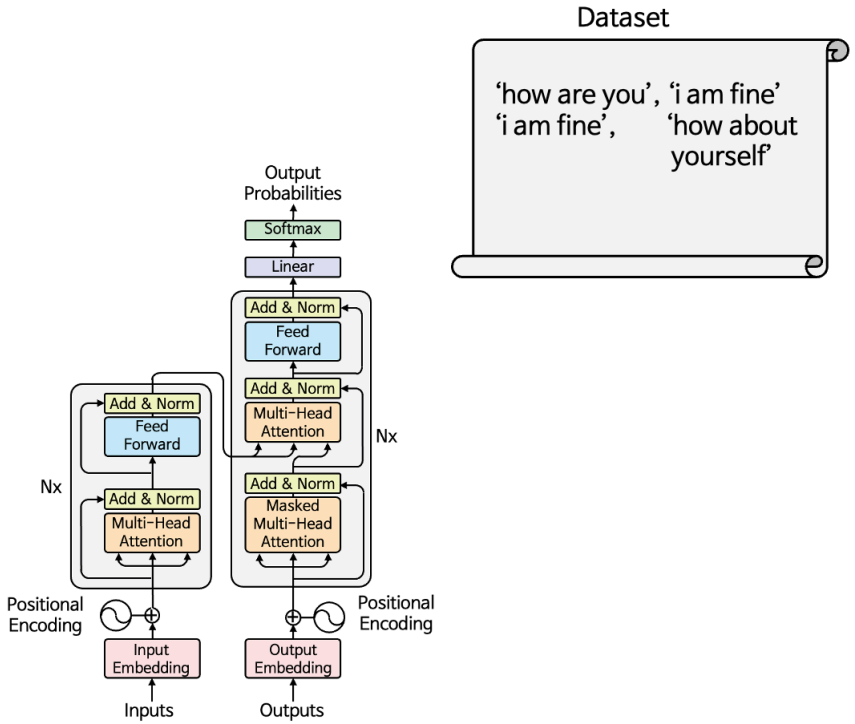
By examining each part step by step, we can fully understand how the model works.



From now on, let’s use a **simple example** to explore how the Transformer learns.



To train a Transformer model, the very first step is to **create a dataset**.



Next, we extract all the words from the dataset to build a vocabulary.

Dataset

'how are you', 'i am fine'
'i am fine', 'how about yourself'

Vocab List

⟨SOS⟩:
⟨EOS⟩:
⟨PAD⟩:
am:
fine:
how:
i:
about:
are:
you:
yourself:

Then, we assign each word (or token) a unique number (index) so that the model can process it as input data.

Dataset

'how are you', 'i am fine'
'i am fine', 'how about yourself'

Vocab List

⟨SOS⟩:	0
⟨EOS⟩:	1
⟨PAD⟩:	2
am:	3
fine:	4
how:	5
i:	6
about:	7
are:	8
you:	9
yourself:	10

We assign each word (token) a unique index to convert it into a form that the model can handle.

Dataset

'how are you', 'i am fine'
'i am fine', 'how about yourself'

Vocab List

⟨SOS⟩:	0
⟨EOS⟩:	1
⟨PAD⟩:	2
am:	3
fine:	4
how:	5
i:	6
about:	7
are:	8
you:	9
yourself:	10

Suppose that our task is to make a model that can do such a Q&A

'How are you?' ➔ *'I am fine'*

For simplicity, let’s assume a very small number of tokens when assigning these indices.

Dataset

‘how are you’, ‘i am fine’
‘i am fine’, ‘how about yourself’

Vocab List

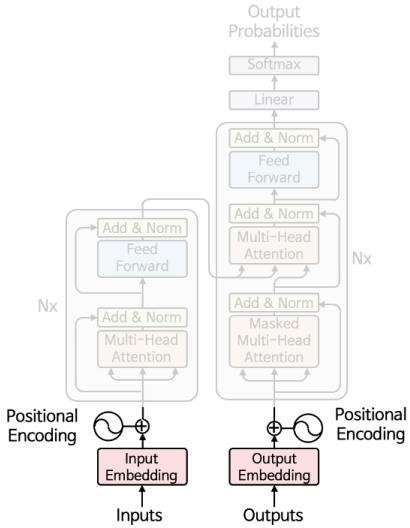
⟨SOS⟩:	0
⟨EOS⟩:	1
⟨PAD⟩:	2
am:	3
fine:	4
how:	5
i:	6
about:	7
are:	8
you:	9
yourself:	10

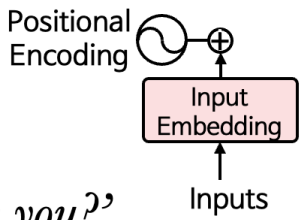
Suppose that our task is to make a model that can do such a Q&A

‘*How are you?*’ ➔ ‘*<SOS> I am*’

[5, 8, 9] ➔ [0, 6, 3]

When an input sentence like this comes in, the first step is word embedding.

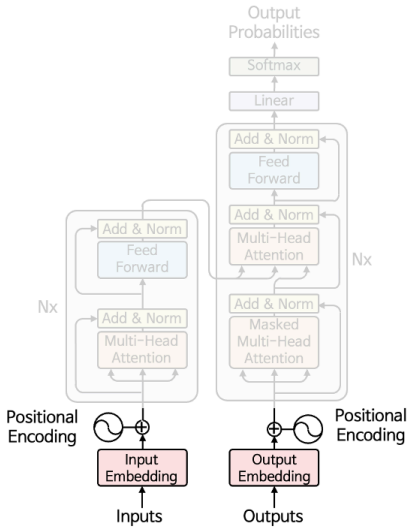


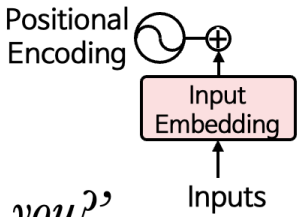


‘How are you?’

[5, 8, 9]

The input embedding block takes input tokens like [5, 8, 9] and

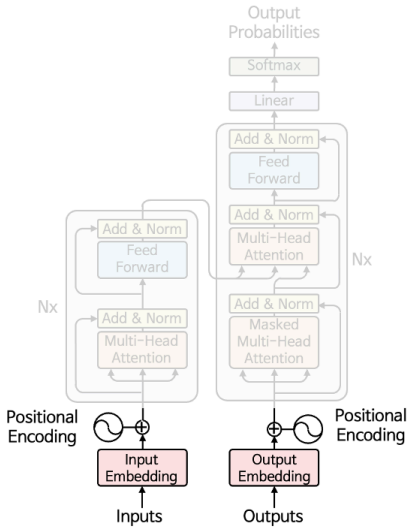




‘How are you?’

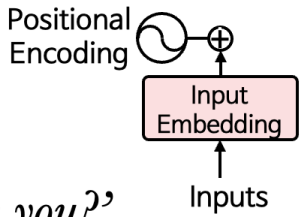
[5, 8, 9]

outputs the corresponding embedding vectors for each word.



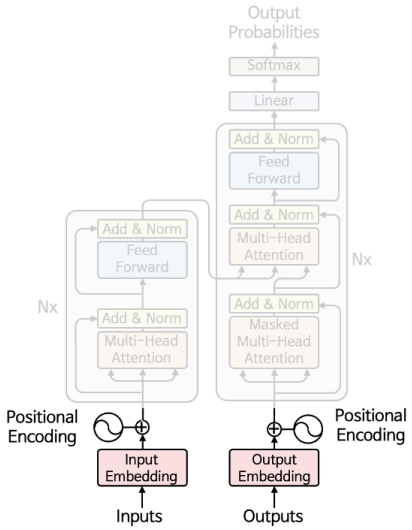
‘How are you?’

[5, 8, 9]



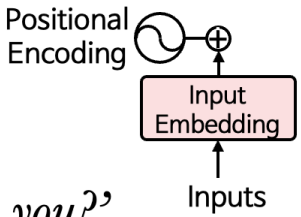
5, how	-0.25	0.86	-1.38	-0.87	-0.22	1.72
8, are	-0.91	-0.66	2.22	0.52	0.35	-0.20
9, you	-1.05	1.28	0.15	0.23	0.06	0.43

The embedding layer compresses, for example, 11 words into



⟨SOS⟩:	0
⟨EOS⟩:	1
⟨PAD⟩:	2
am:	3
fine:	4
how:	5
i:	6
about:	7
are:	8
you:	9
yourself:	10

11

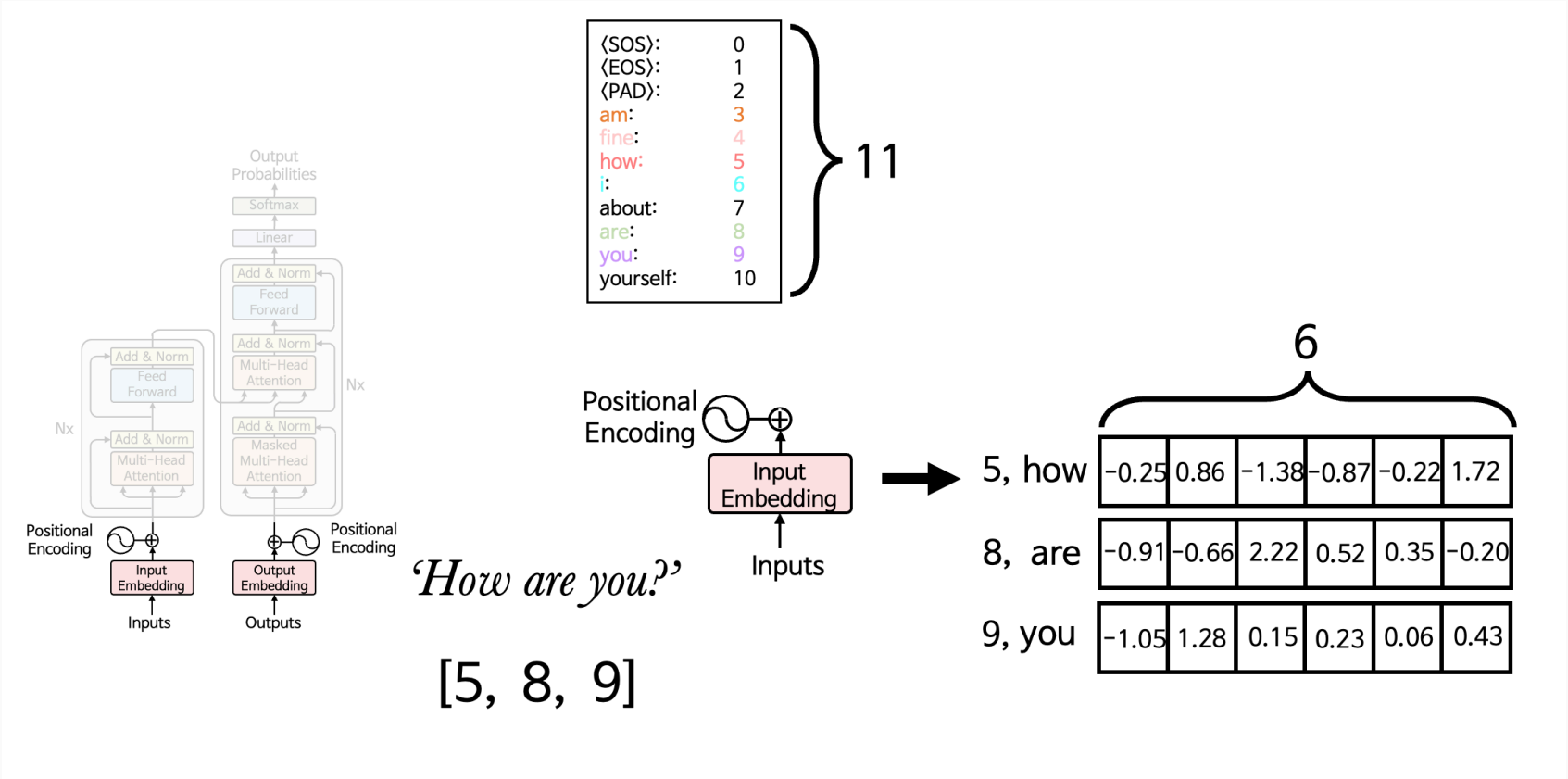


'How are you?'

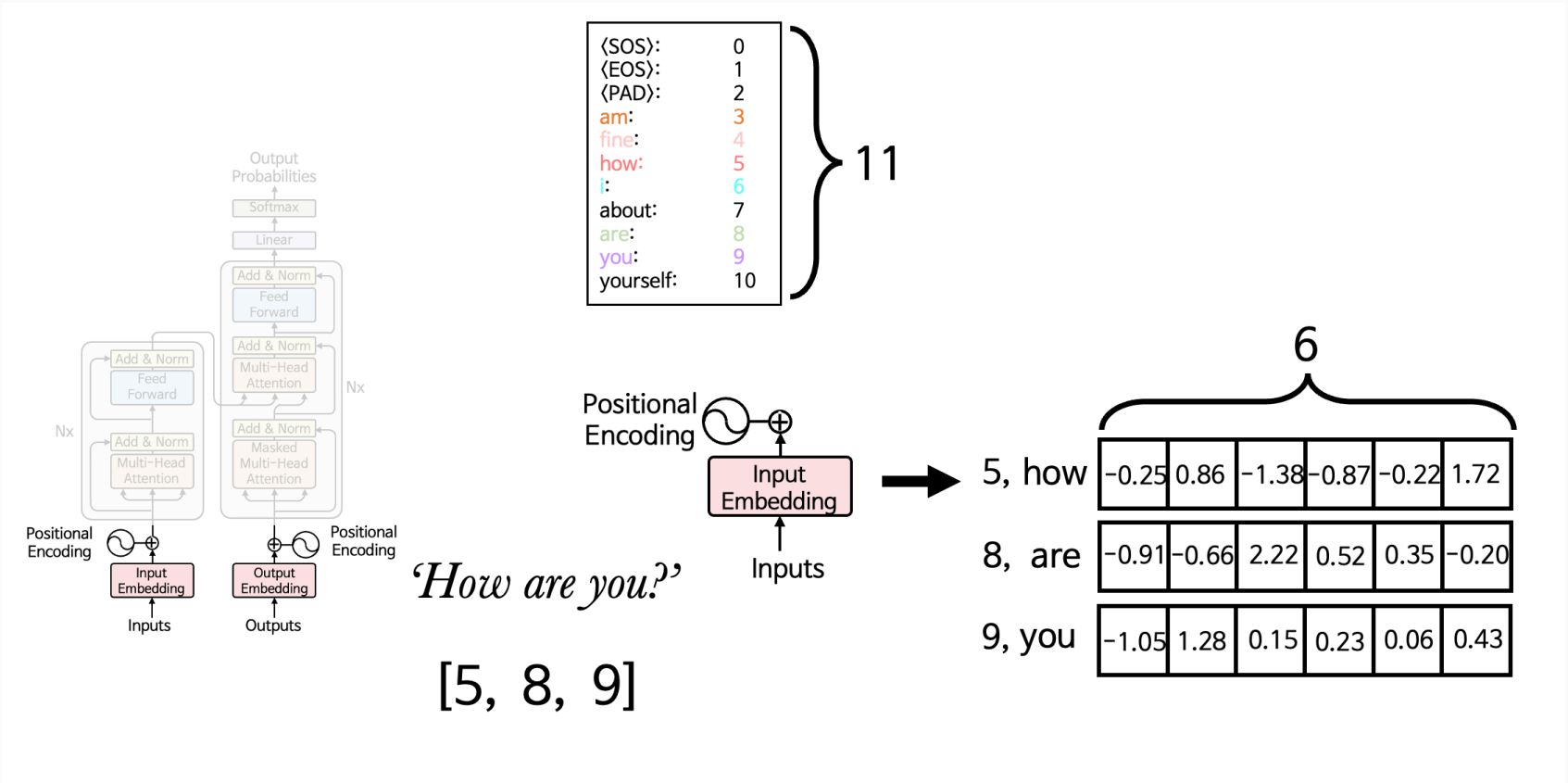
[5, 8, 9]

5, how	-0.25	0.86	-1.38	-0.87	-0.22	1.72
8, are	-0.91	-0.66	2.22	0.52	0.35	-0.20
9, you	-1.05	1.28	0.15	0.23	0.06	0.43

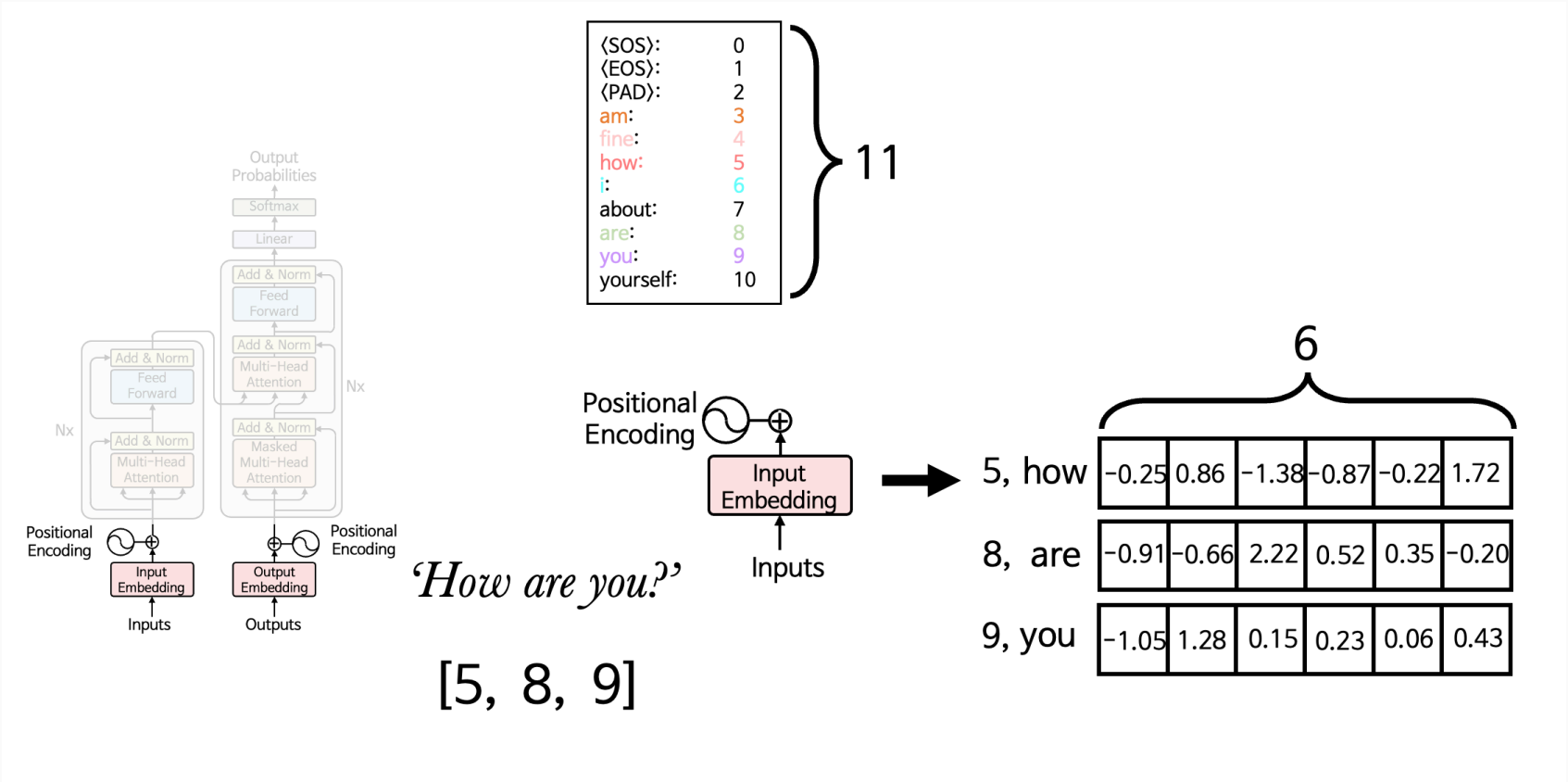
dense vectors of length 6.



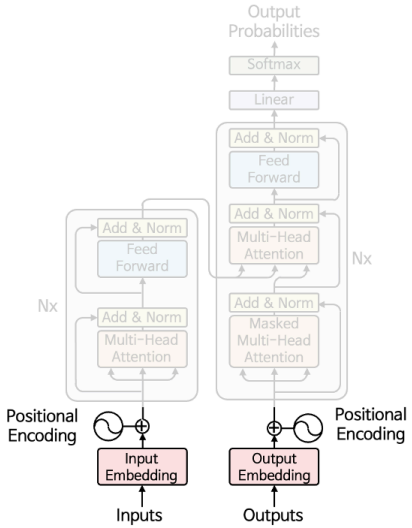
- In the original paper, 37,000 words were reduced to 512-dimensional embeddings. (So, it doesn't have to be 6 as in our example.)

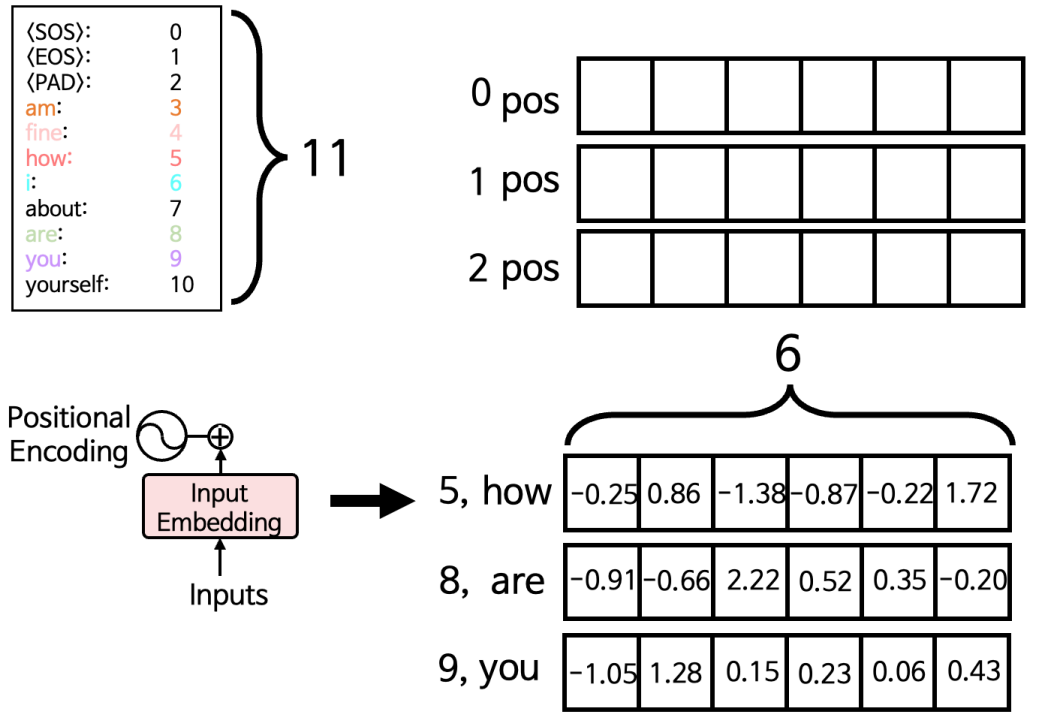


- In the original paper, 37,000 words were reduced to 512-dimensional embeddings. (So, it doesn't have to be 6 as in our example.)
- It's important to understand that this dimensional embeddings allow the model to efficiently process and represent a large number of words internally.



Next is **positional** encoding. The Transformer uses a unique way to encode the position of words within a sentence.

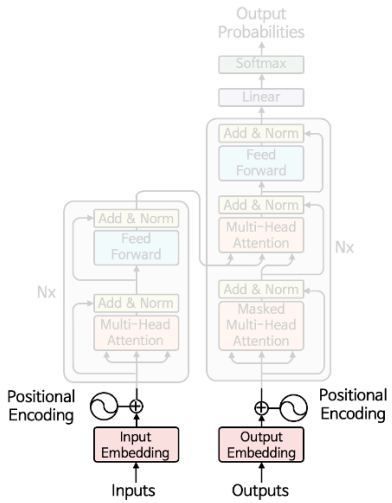


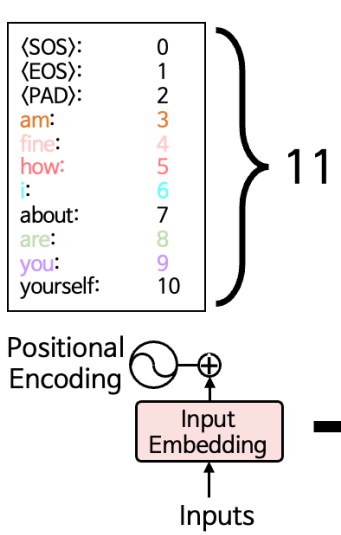


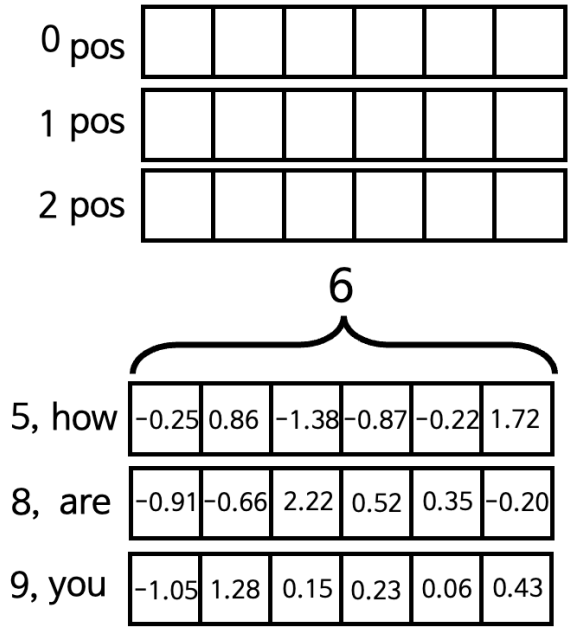
Why do we need this?

Why do we need this? Self-attention alone does not capture word order (e.g., RNNs); it treats inputs as a set, not a sequence.

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





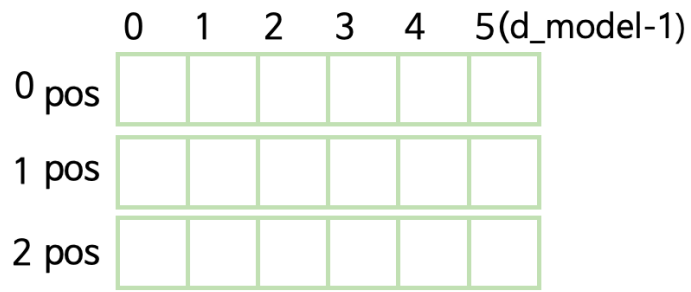


We encode the position of each word based on the following formulas.

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

Let’s take a closer look at these formulas.

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

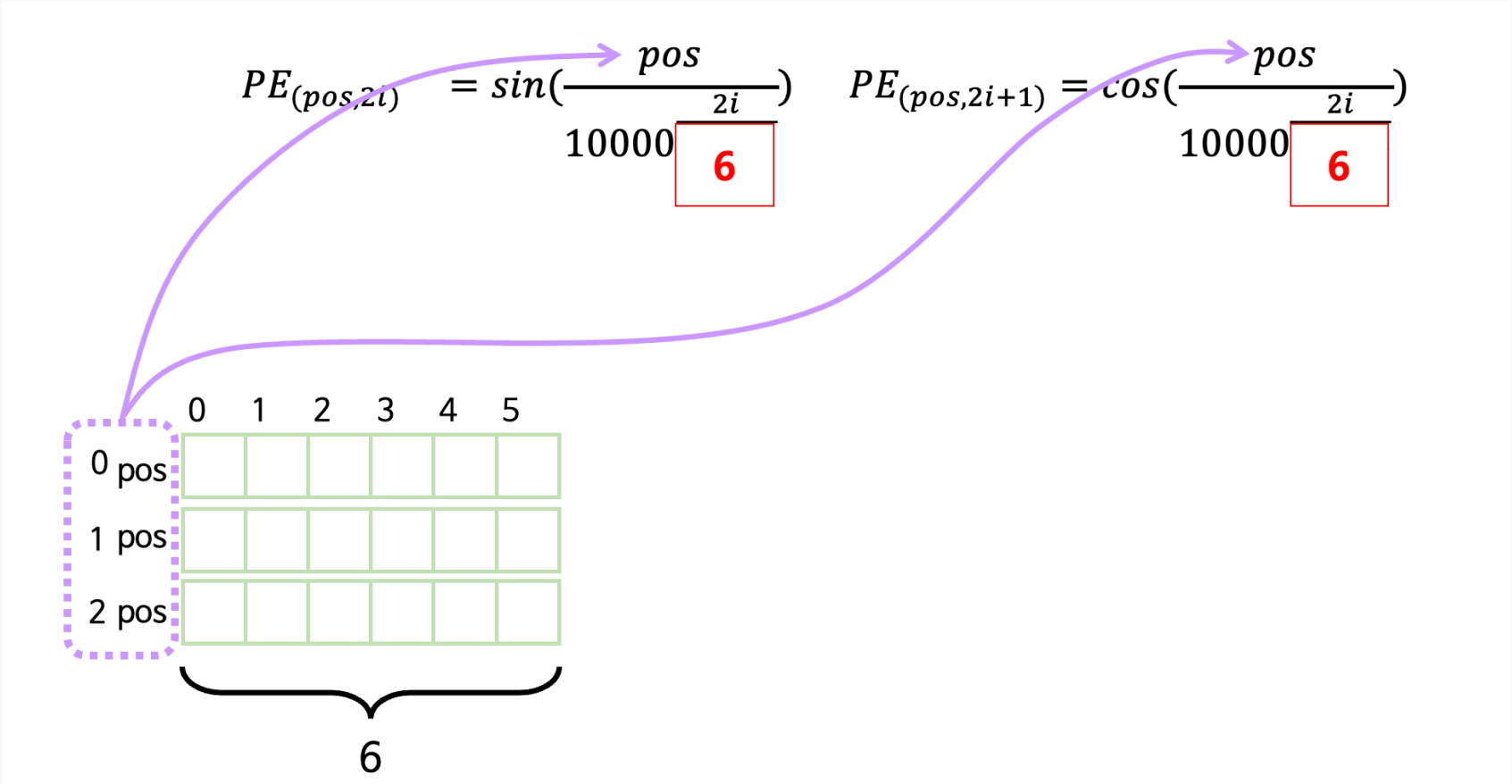


In this example, the value of d_{model} is 6 (the dimensionality of the model).

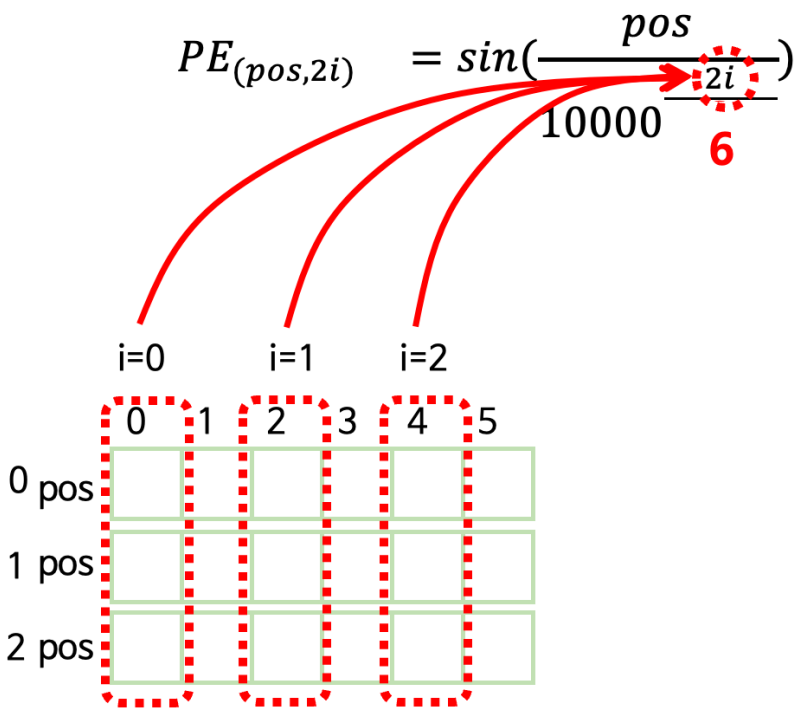
$$PE_{(pos,2i)} = sin(\frac{pos}{10000^{\frac{2i}{6}}}) \qquad PE_{(pos,2i+1)} = cos(\frac{pos}{10000^{\frac{2i}{6}}})$$

	0	1	2	3	4	5(d_model-1)
0 pos						
1 pos						
2 pos						

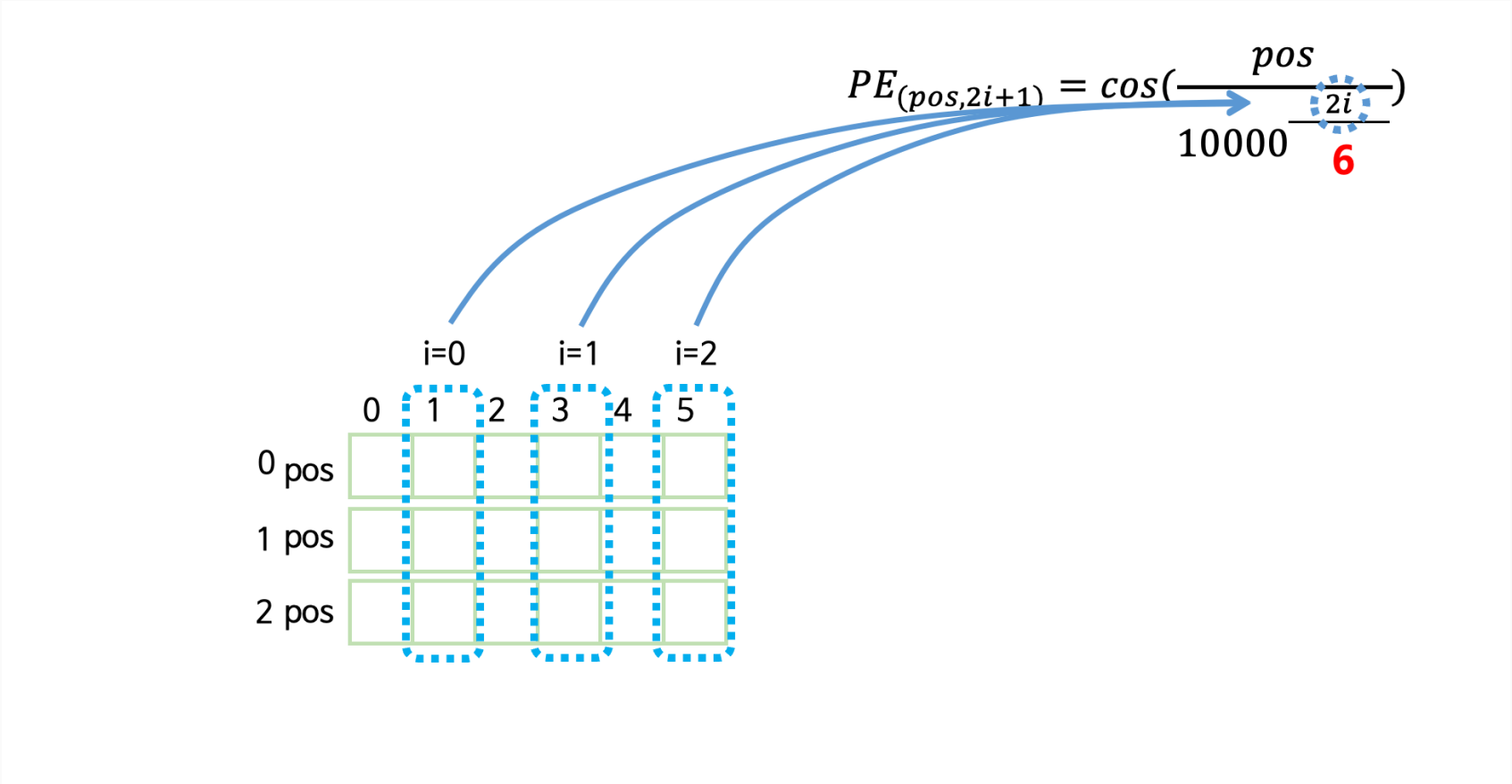
And the variable *pos* takes the values 0, 1, and 2 in order.



i = 0, 1, 2 denotes the dimension index used for even (2i) and odd (2i+1) components. For even dimensions, we apply this formula:



And for odd dimensions, we apply this formula:



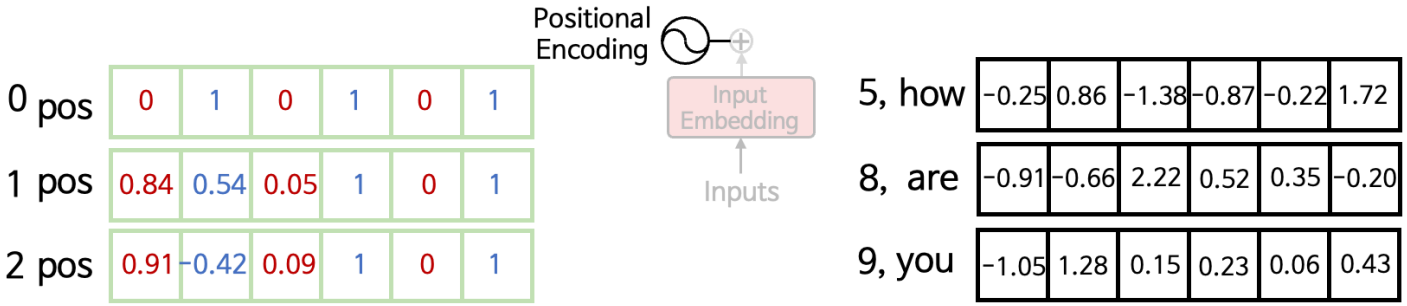
We can now compute the positional encoding values as follows:

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

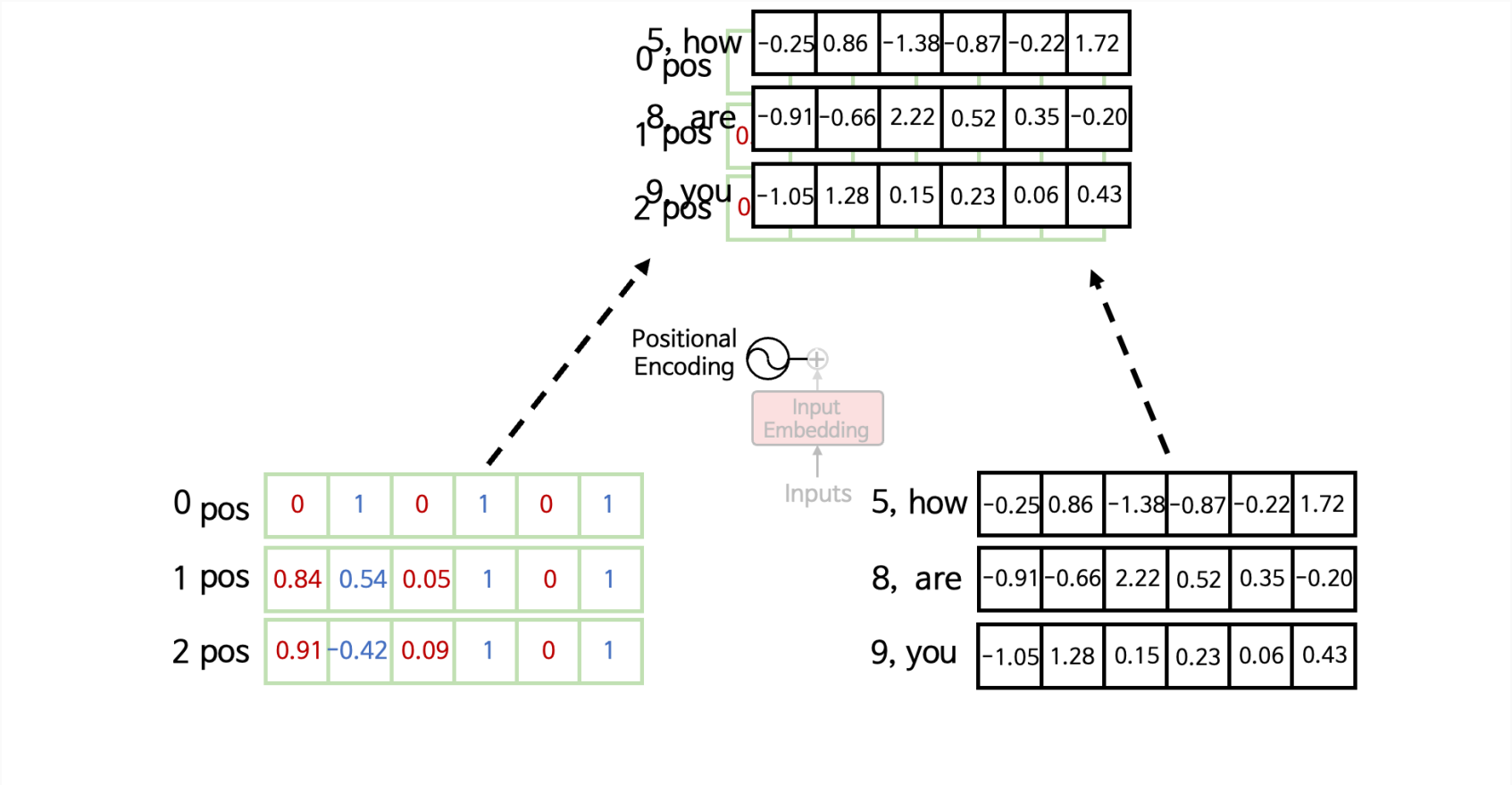
	0	1	2	3	4	5
0 pos	0	1	0	1	0	1
1 pos	0.84	0.54	0.05	1	0	1
2 pos	0.91	-0.42	0.09	1	0	1

6

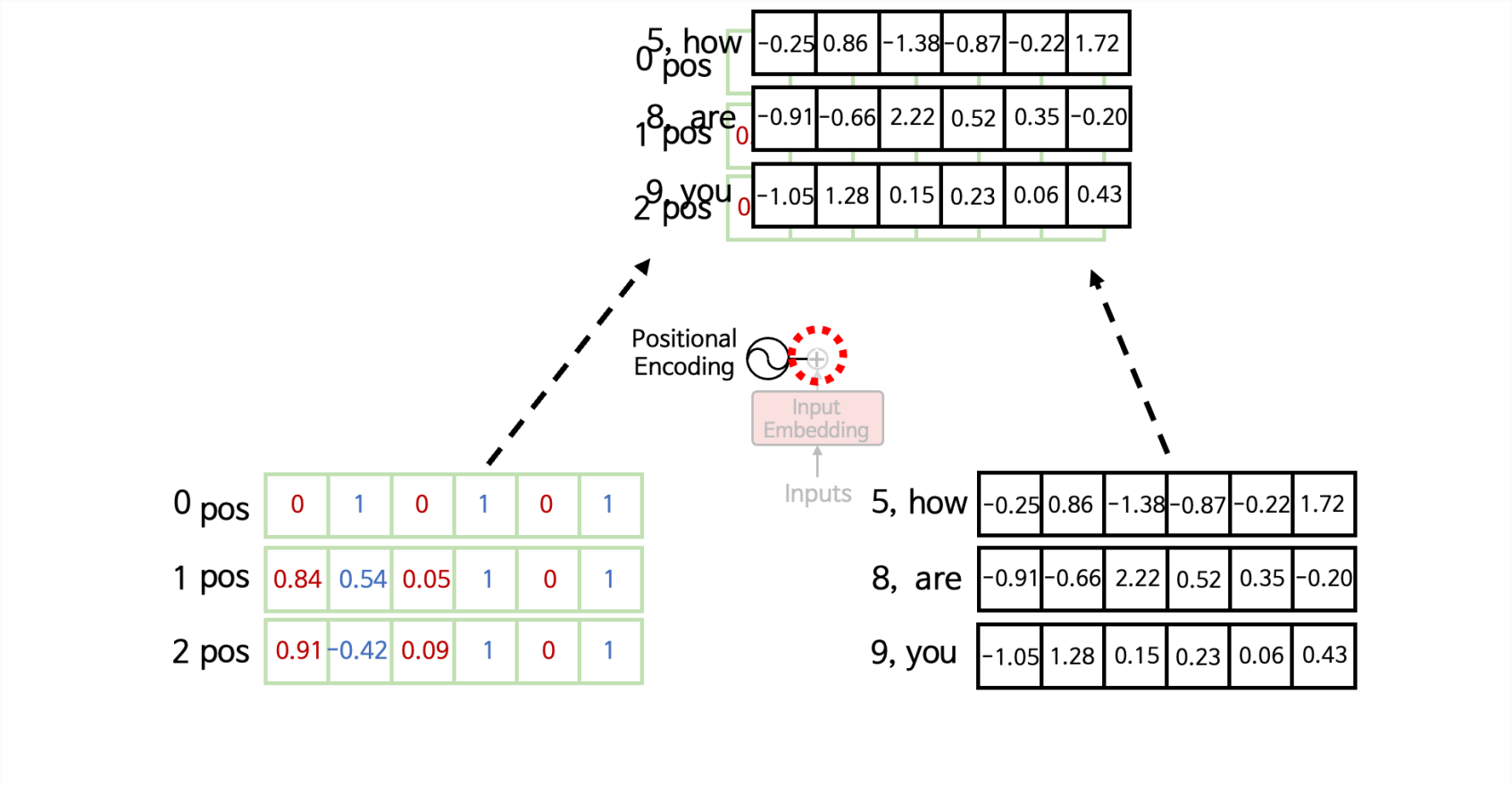
Once we have these positional embeddings, we simply add them to the input embeddings:



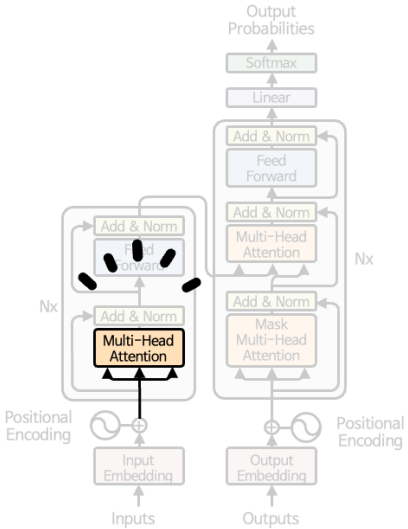
By adding them together, we obtain the combined input + positional embedding vectors.



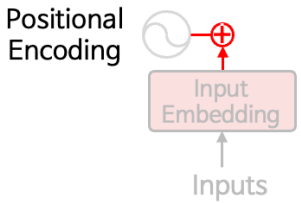
By adding them together, we obtain the combined input + positional embedding vectors.



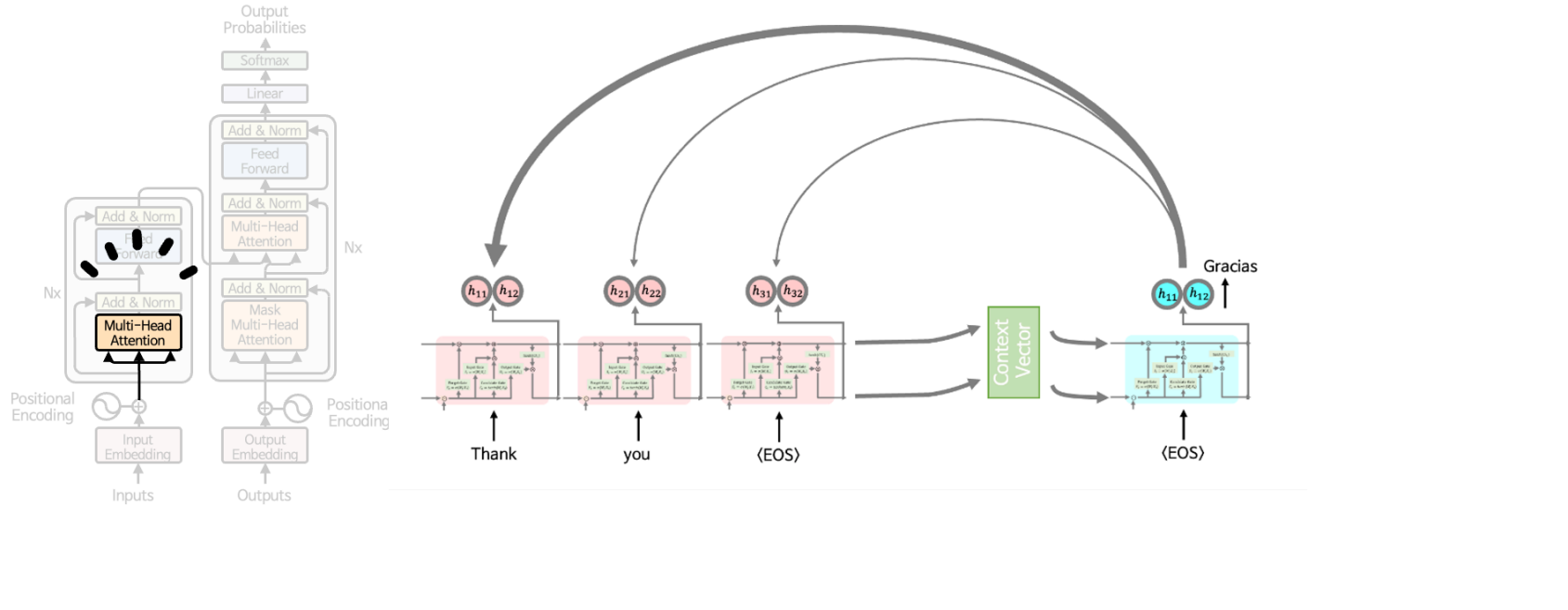
Now, it’s time to feed this combined input–position matrix into the **multi-head attention**, which is the core component of the Transformer.



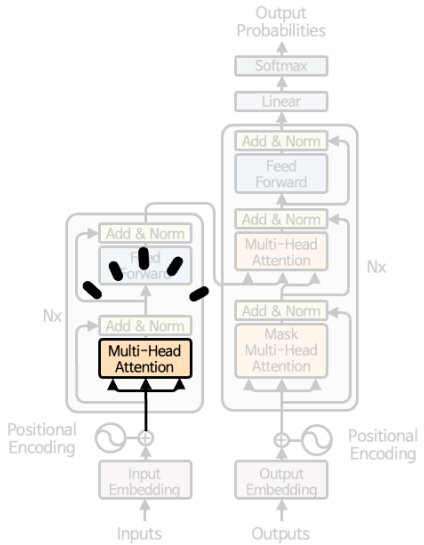
-0.25	1.86	-1.38	0.13	-0.22	2.72
-0.07	-0.12	2.26	1.52	0.35	0.80
-0.15	0.86	0.24	1.23	0.06	1.43

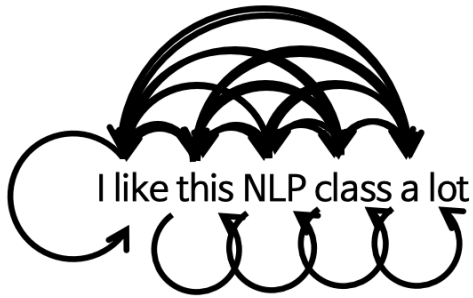


Again, the Transformer's multi-head attention is different from the attention mechanism used in traditional seq2seq models. While seq2seq attention focuses on aligning the input and output sequences,

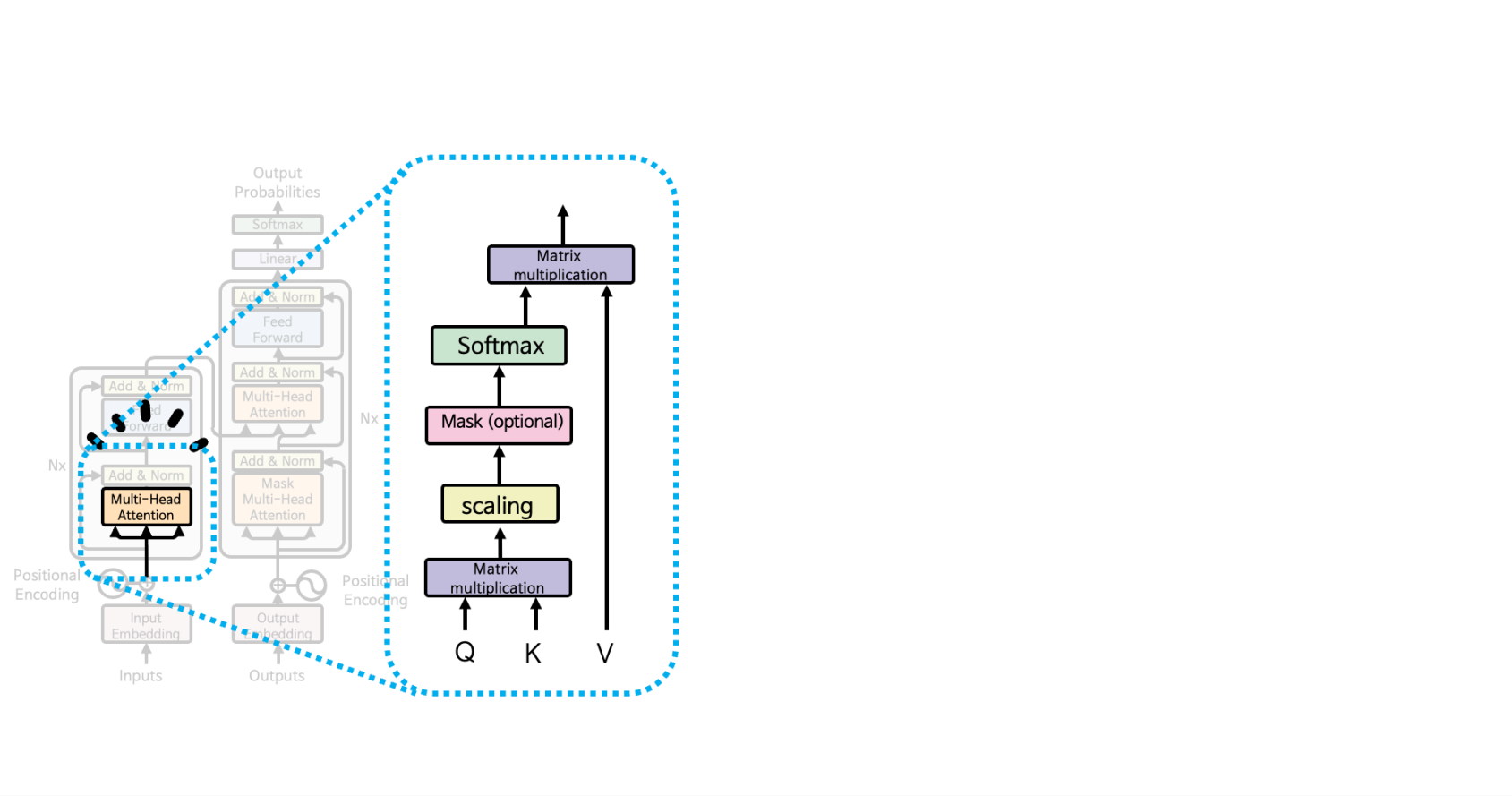


the Transformer’s attention captures the relationships between words within the same input sentence.

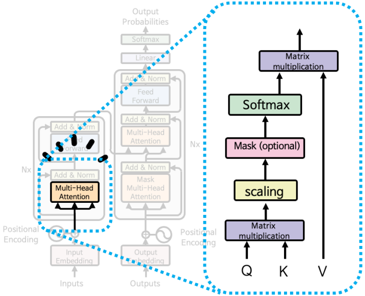




The structure of the multi-head attention mechanism used for self-attention looks like this:



We make three copies of the input + positional encoding matrix.

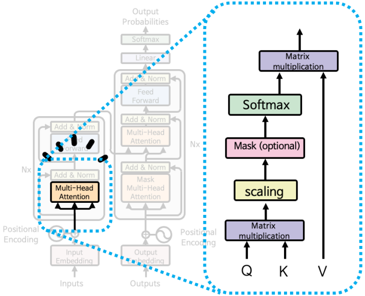


-0.25	1.86	-1.38	0.13	-0.22	2.72
-0.07	-0.12	2.26	1.52	0.35	0.80
-0.15	0.86	0.24	1.23	0.06	1.43

-0.25	1.86	-1.38	0.13	-0.22	2.72
-0.07	-0.12	2.26	1.52	0.35	0.80
-0.15	0.86	0.24	1.23	0.06	1.43

-0.25	1.86	-1.38	0.13	-0.22	2.72
-0.07	-0.12	2.26	1.52	0.35	0.80
-0.15	0.86	0.24	1.23	0.06	1.43

We make three copies of the input + positional encoding matrix.



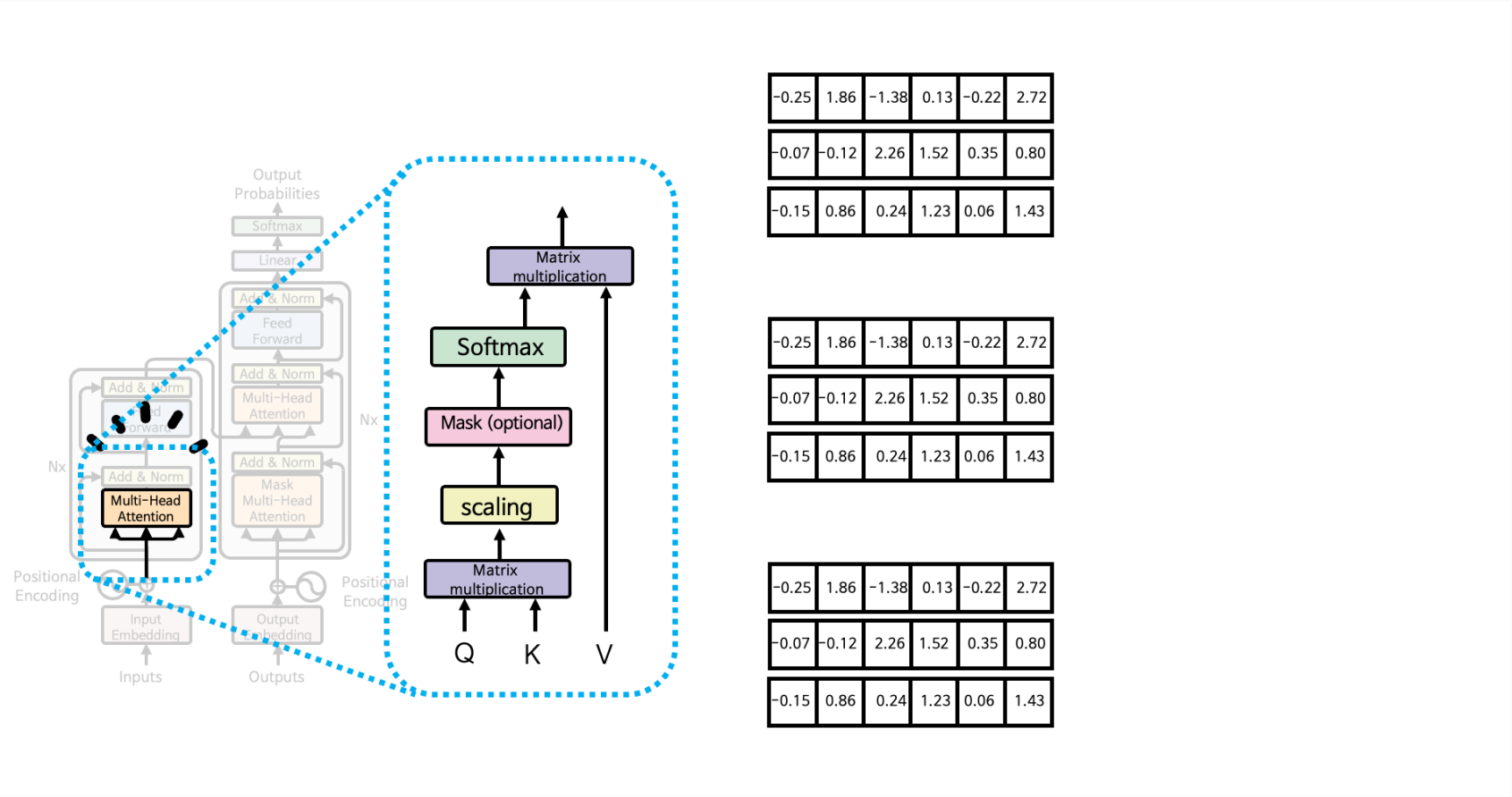
-0.25	1.86	-1.38	0.13	-0.22	2.72
-0.07	-0.12	2.26	1.52	0.35	0.80
-0.15	0.86	0.24	1.23	0.06	1.43

-0.25	1.86	-1.38	0.13	-0.22	2.72
-0.07	-0.12	2.26	1.52	0.35	0.80
-0.15	0.86	0.24	1.23	0.06	1.43

-0.25	1.86	-1.38	0.13	-0.22	2.72
-0.07	-0.12	2.26	1.52	0.35	0.80
-0.15	0.86	0.24	1.23	0.06	1.43

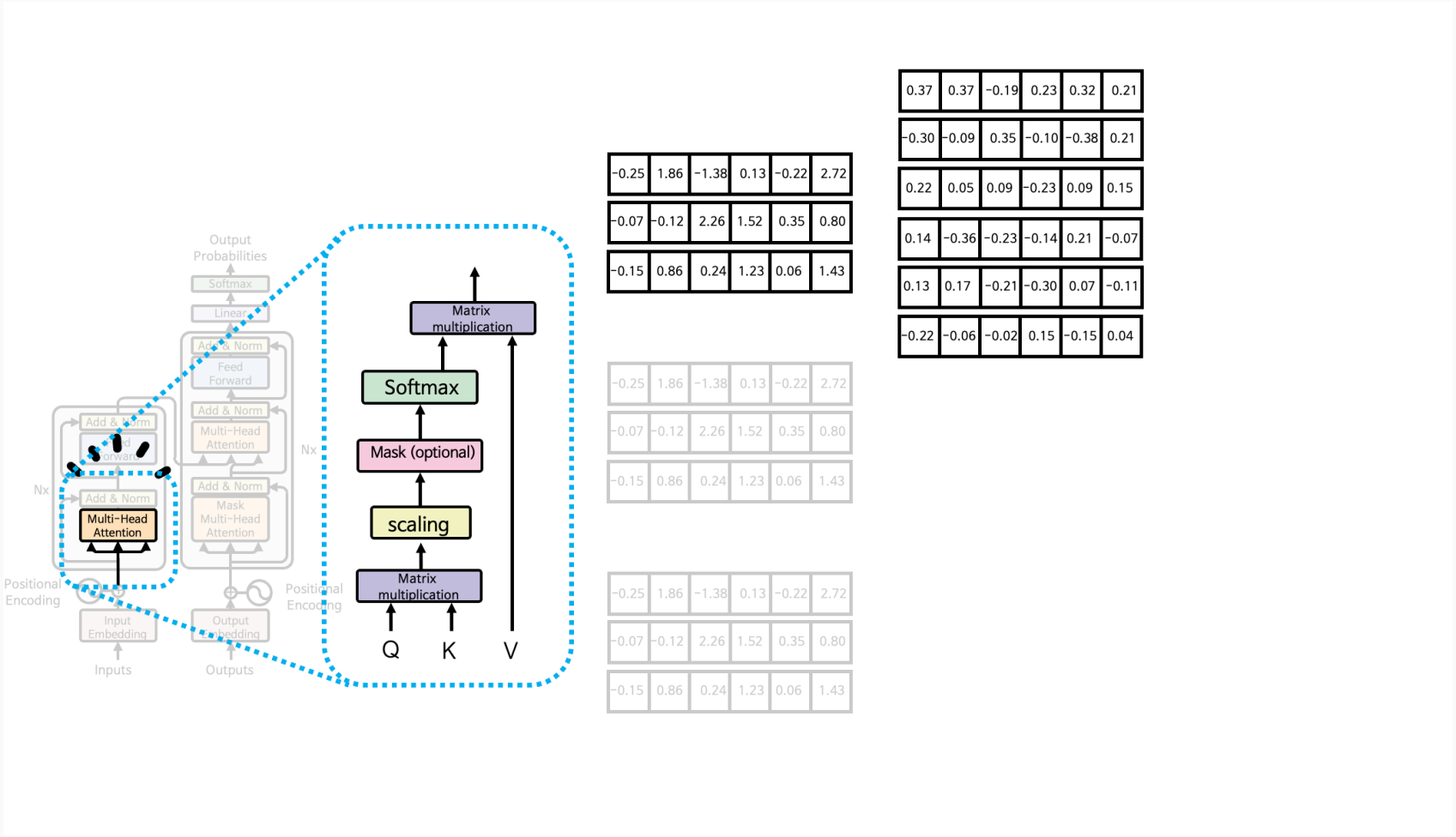
Why?

We make three copies of the input + positional encoding matrix.



Why? This is done to create Query (Q), Key (K), and Value (V) matrices — each representing a different projection of the same input for the attention mechanism.

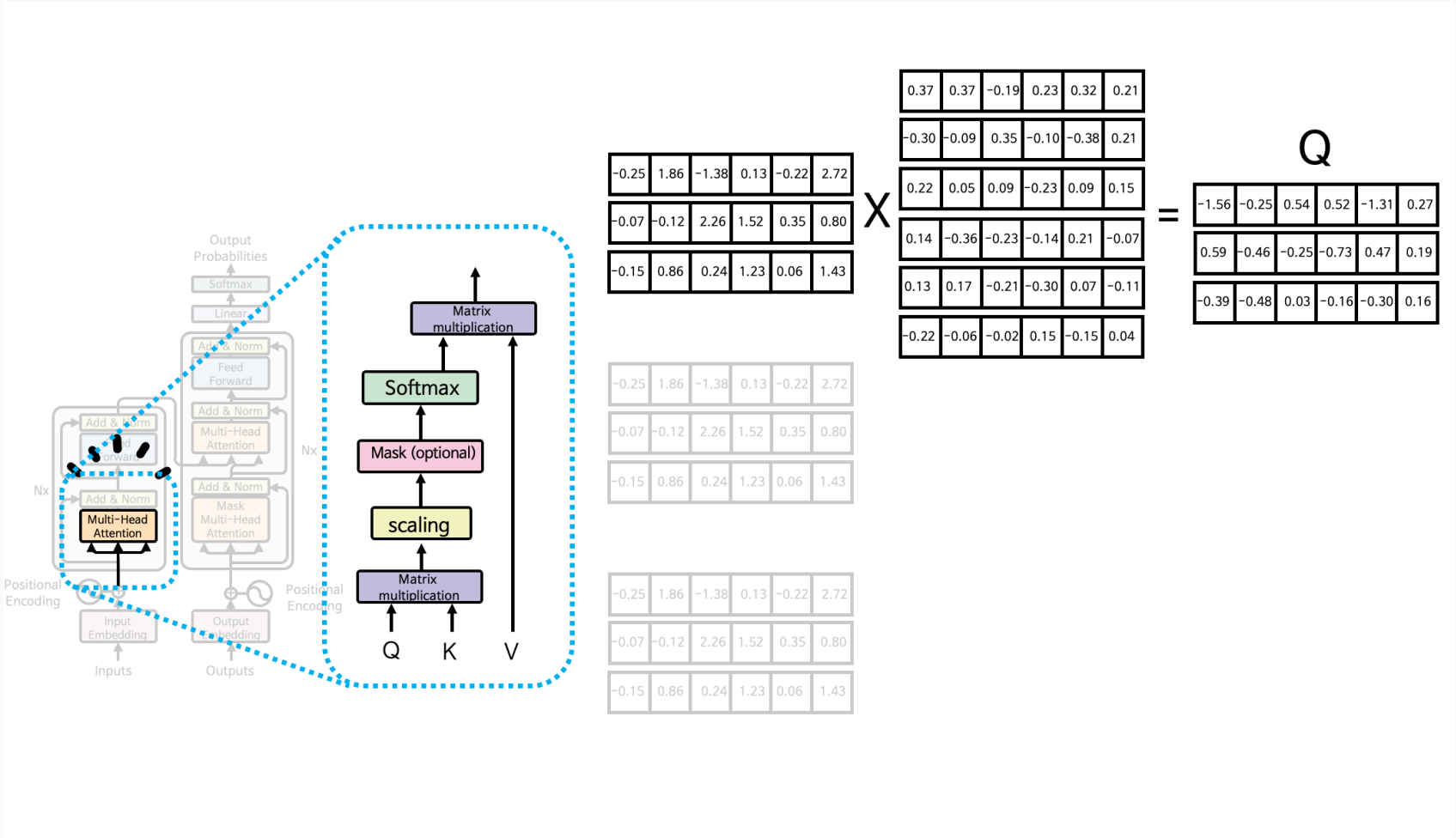
To obtain the Q matrix, we create the following 6×6 weight matrix (randomly initialized).



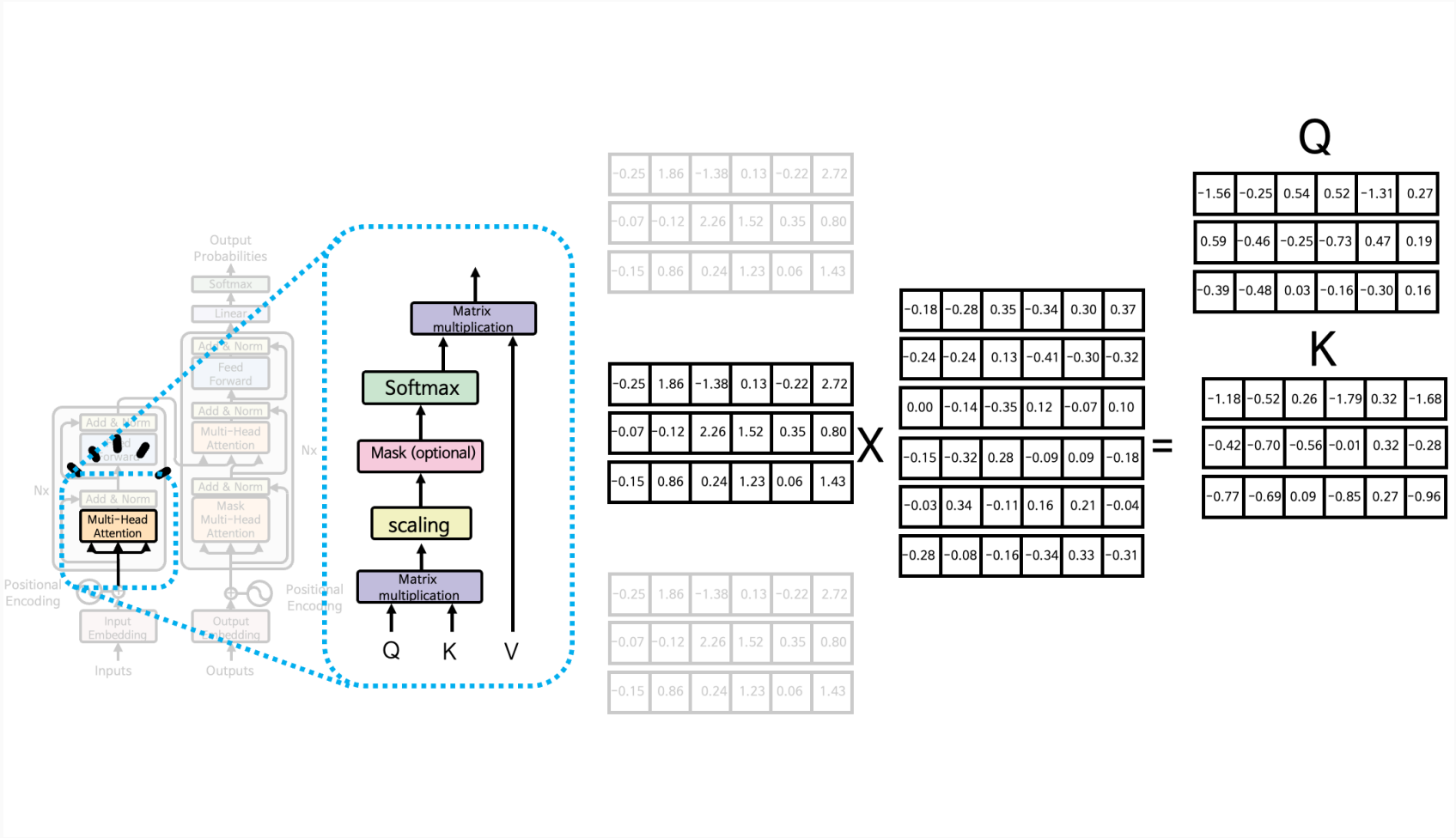
-0.25	1.86	-1.38	0.13	-0.22	2.72
-0.07	-0.12	2.26	1.52	0.35	0.80
-0.15	0.86	0.24	1.23	0.06	1.43

0.37	0.37	-0.19	0.23	0.32	0.21
-0.30	-0.09	0.35	-0.10	-0.38	0.21
0.22	0.05	0.09	-0.23	0.09	0.15
0.14	-0.36	-0.23	-0.14	0.21	-0.07
0.13	0.17	-0.21	-0.30	0.07	-0.11
-0.22	-0.06	-0.02	0.15	-0.15	0.04

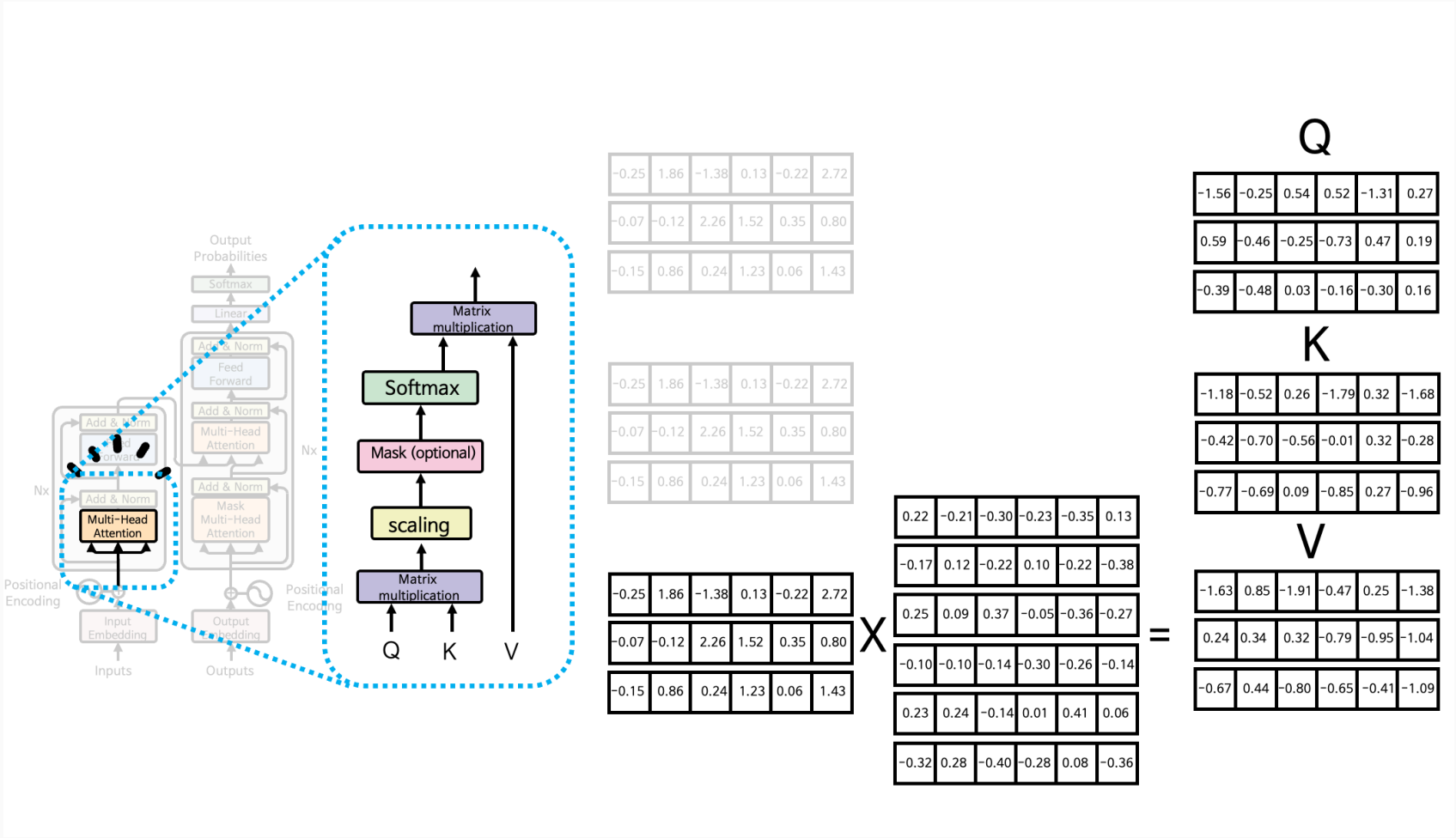
And then we perform matrix multiplication to obtain the Q (Query) matrix.



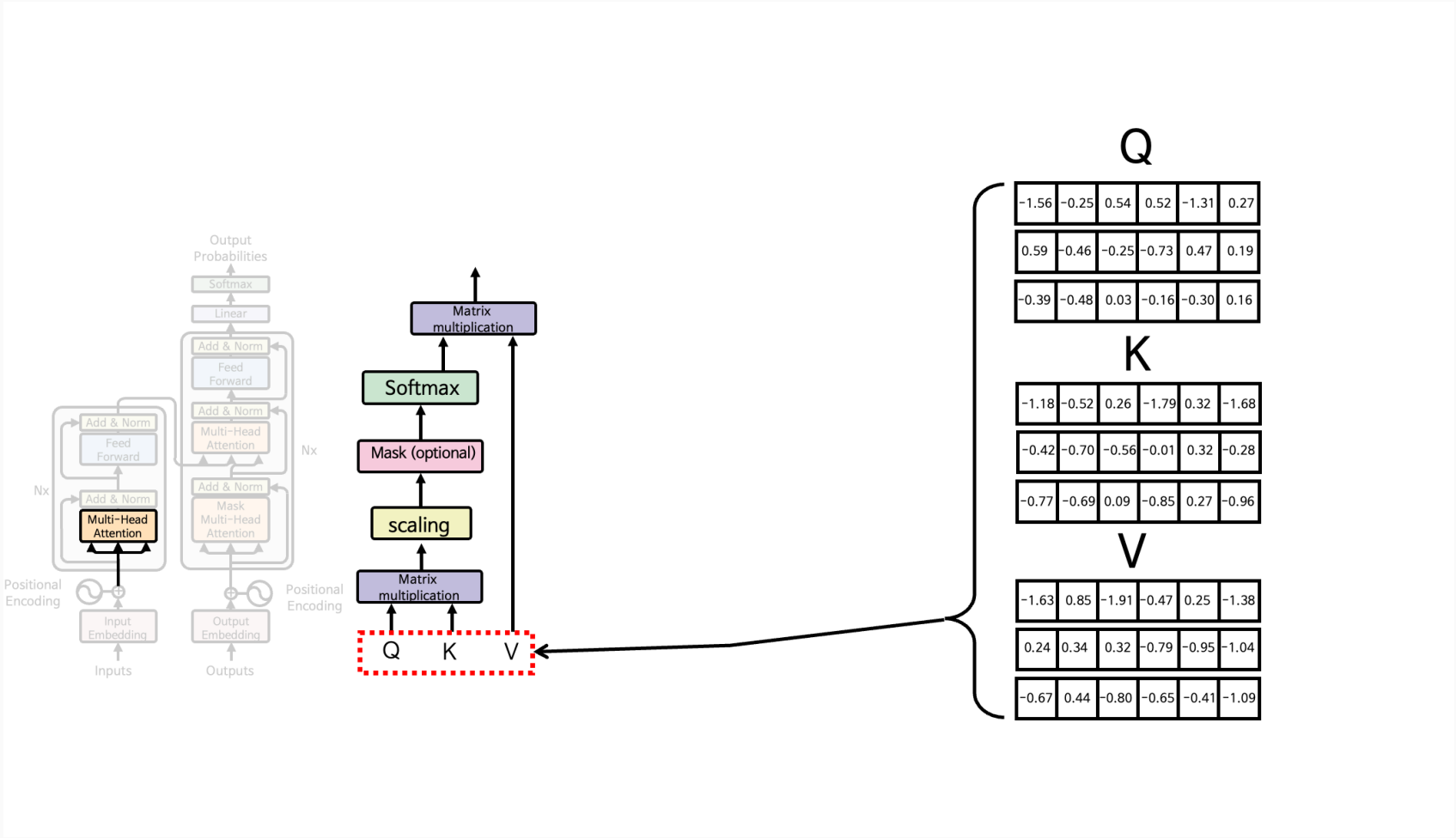
To compute the K (Key) matrix, we create another 6×6 weight matrix (randomly initialized) and multiply it with the input.



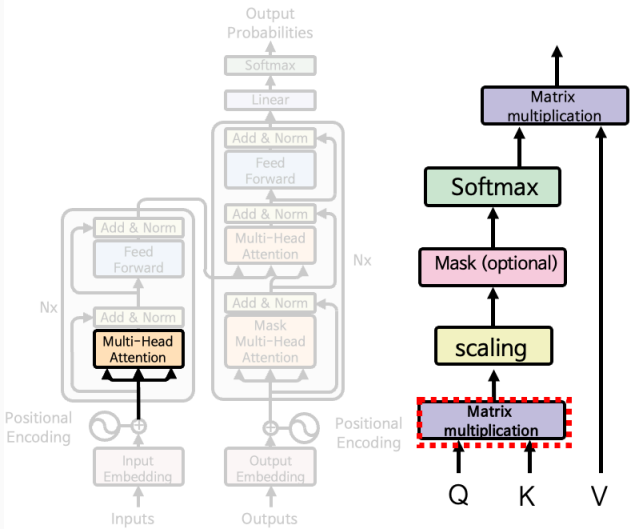
Using the same process, we perform another matrix multiplication to obtain the V (Value) matrix.



Now, we have the three inputs of the multi-head attention layer — Q (Query), K (Key), and V (Value).



Next, we perform the matrix multiplication between Q and K. The formula for this operation is as follows:



$$Q \cdot K^T$$

Q

-1.56	-0.25	0.54	0.52	-1.31	0.27
0.59	-0.46	-0.25	-0.73	0.47	0.19
-0.39	-0.48	0.03	-0.16	-0.30	0.16

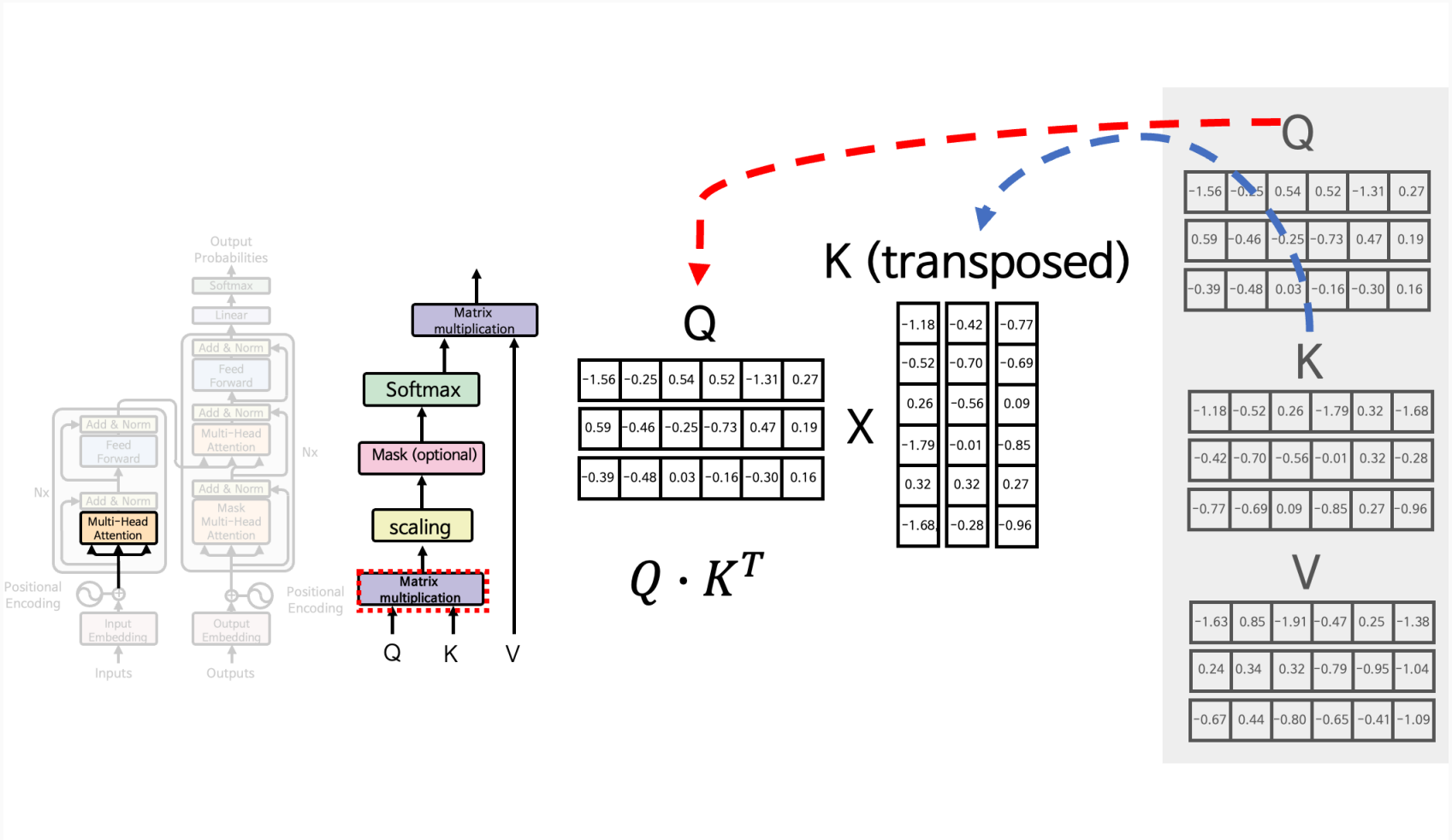
K

-1.18	-0.52	0.26	-1.79	0.32	-1.68
-0.42	-0.70	-0.56	-0.01	0.32	-0.28
-0.77	-0.69	0.09	-0.85	0.27	-0.96

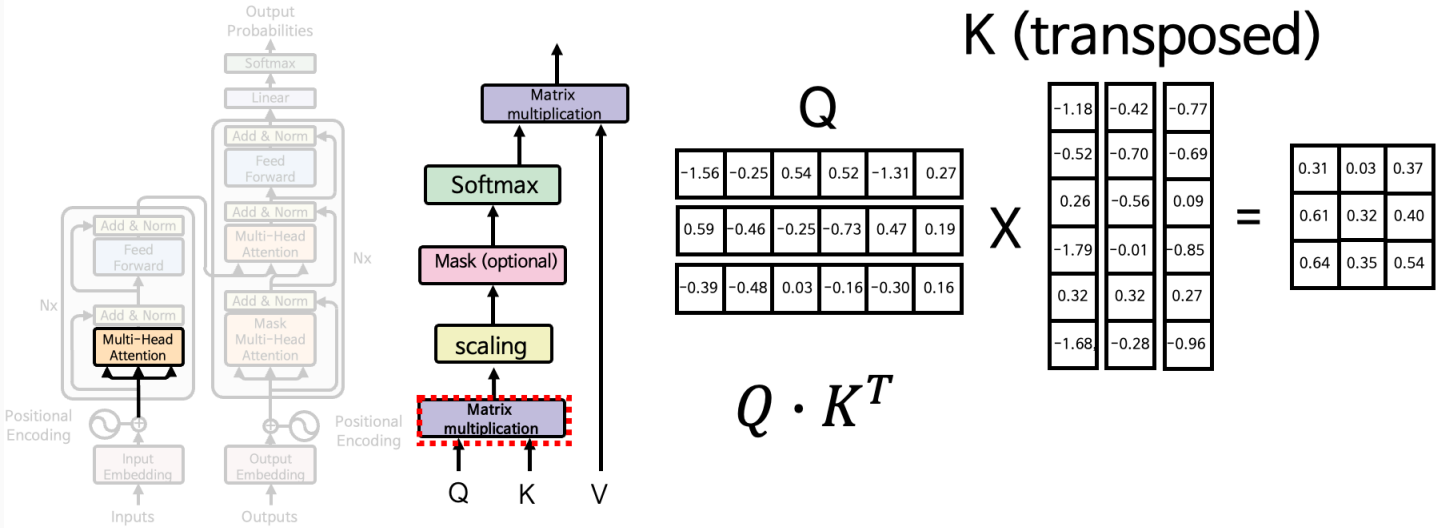
V

-1.63	0.85	-1.91	-0.47	0.25	-1.38
0.24	0.34	0.32	-0.79	-0.95	-1.04
-0.67	0.44	-0.80	-0.65	-0.41	-1.09

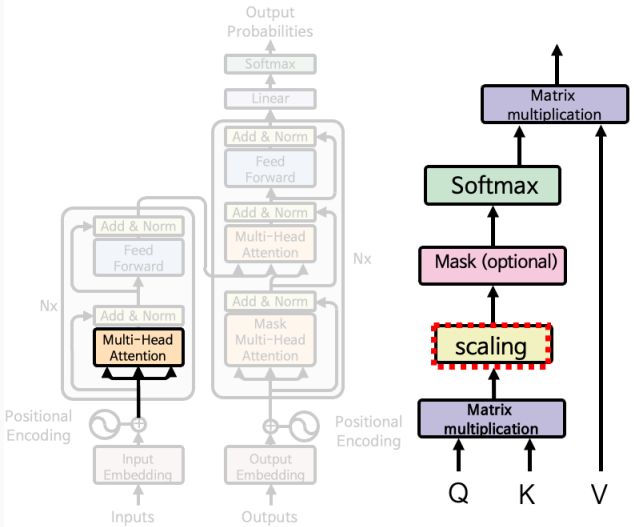
When we plug in the matrix values and calculate,



we can obtain the result as follows:

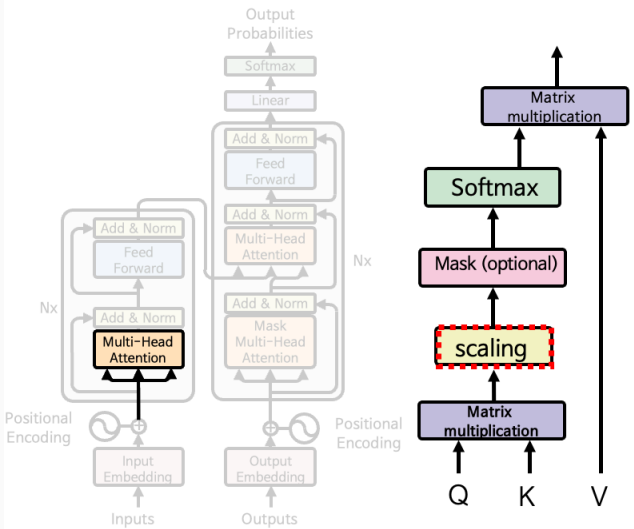


Next, we apply *scaling*, which divides the matrix by $\sqrt{6}$, since in this example the value of d_{model} is 6.



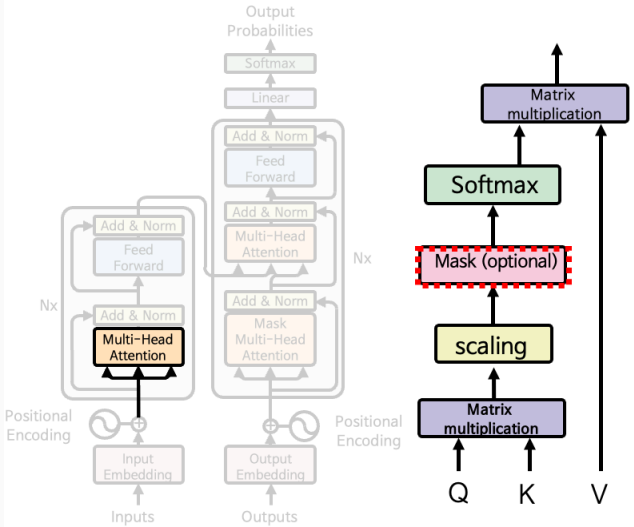
$$Q \cdot K^T = \begin{array}{|c|c|c|}\hline 0.31 & 0.03 & 0.37 \\ \hline 0.61 & 0.32 & 0.40 \\ \hline 0.64 & 0.35 & 0.54 \\ \hline \end{array} \div \sqrt{6}$$

Next, we apply *scaling*, which divides the matrix by $\sqrt{6}$, since in this example the value of d_{model} is 6.



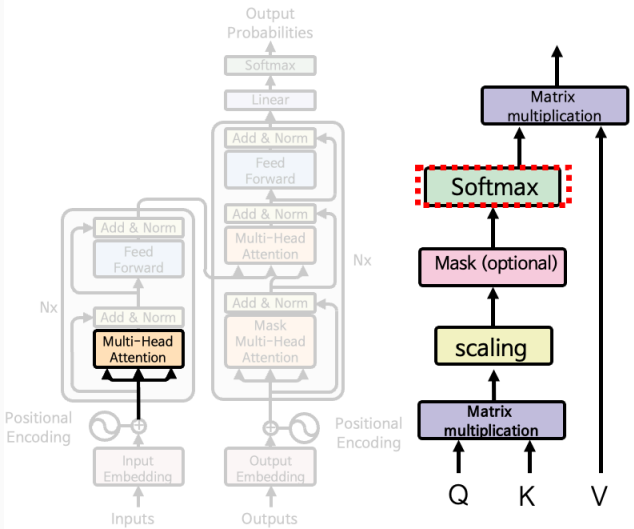
$$Q \cdot K^T = \begin{bmatrix} 0.31 & 0.03 & 0.37 \\ 0.61 & 0.32 & 0.40 \\ 0.64 & 0.35 & 0.54 \end{bmatrix} \div \sqrt{6} = \begin{bmatrix} 0.13 & 0.01 & 0.15 \\ 0.25 & 0.13 & 0.16 \\ 0.26 & 0.14 & 0.22 \end{bmatrix}$$

Since the mask layer is not used in the encoder, we will skip it here.



$$Q \cdot K^T = \begin{bmatrix} 0.31 & 0.03 & 0.37 \\ 0.61 & 0.32 & 0.40 \\ 0.64 & 0.35 & 0.54 \end{bmatrix} \div \sqrt{6} = \begin{bmatrix} 0.13 & 0.01 & 0.15 \\ 0.25 & 0.13 & 0.16 \\ 0.26 & 0.14 & 0.22 \end{bmatrix}$$

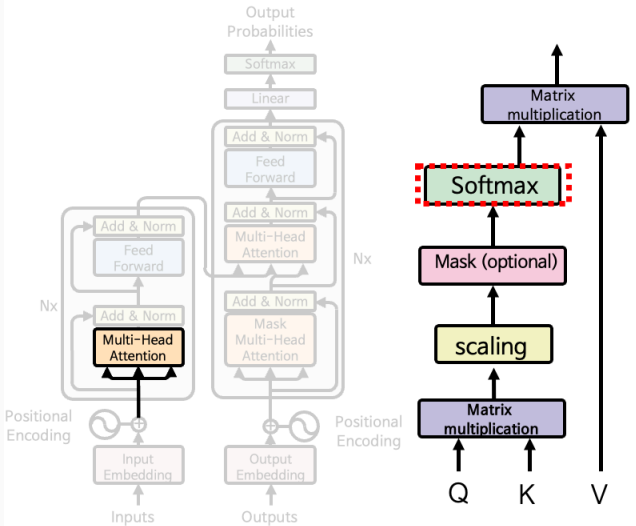
The next softmax layer converts the matrix values into probabilities.



$$softmax\left(\begin{bmatrix} 0.13 & 0.01 & 0.15 \\ 0.25 & 0.13 & 0.16 \\ 0.26 & 0.14 & 0.22 \end{bmatrix}\right) = \begin{bmatrix} 0.34 & 0.31 & 0.35 \\ 0.36 & 0.32 & 0.33 \\ 0.35 & 0.31 & 0.34 \end{bmatrix}$$

$$Q \cdot K^T = \begin{bmatrix} 0.31 & 0.03 & 0.37 \\ 0.61 & 0.32 & 0.40 \\ 0.64 & 0.35 & 0.54 \end{bmatrix} \div \sqrt{6} = \begin{bmatrix} 0.13 & 0.01 & 0.15 \\ 0.25 & 0.13 & 0.16 \\ 0.26 & 0.14 & 0.22 \end{bmatrix}$$

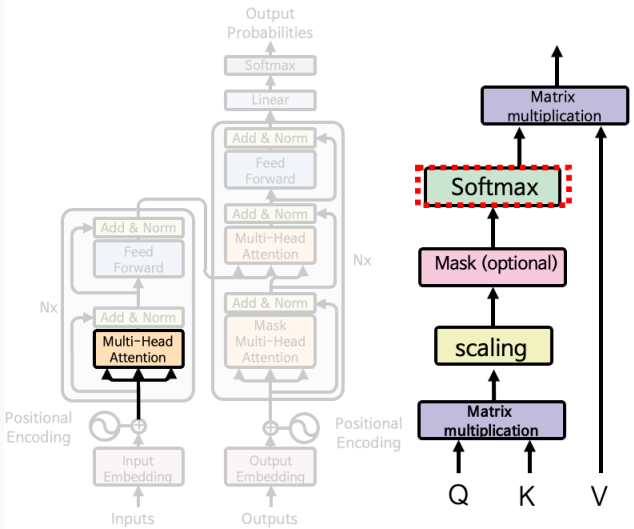
This 3×3 matrix represents the self-attention weights.



$$softmax\left(\begin{bmatrix} 0.13 & 0.01 & 0.15 \\ 0.25 & 0.13 & 0.16 \\ 0.26 & 0.14 & 0.22 \end{bmatrix}\right) =$$

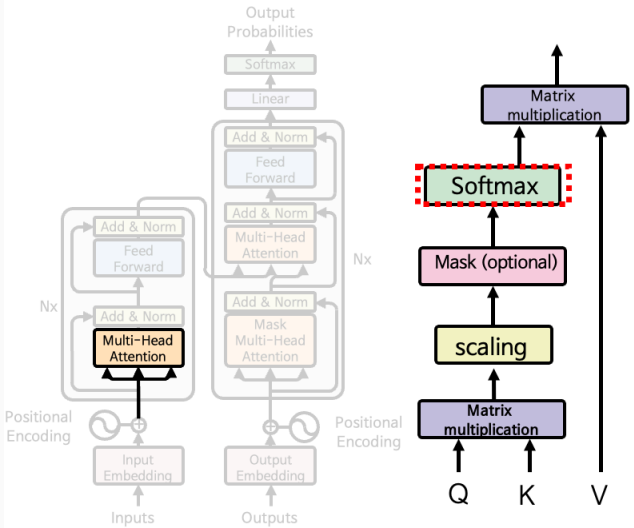
	how	are	you
how	0.34	0.31	0.35
are	0.36	0.32	0.33
you	0.35	0.31	0.34

This matrix numerically shows how each word in the input is related to every other word.



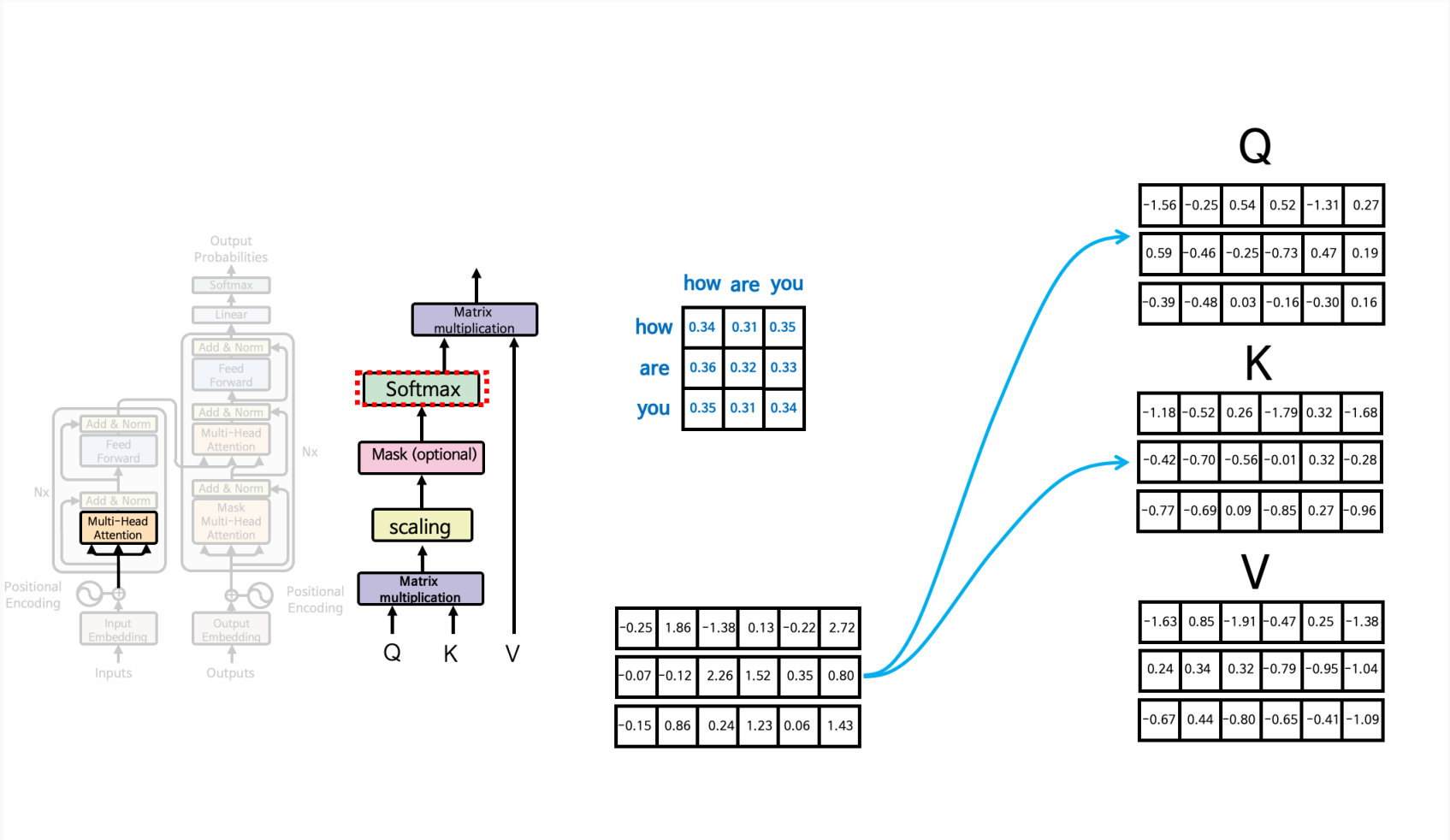
	how	are	you
how	0.34	0.31	0.35
are	0.36	0.32	0.33
you	0.35	0.31	0.34

Thus, word pairs with higher relevance receive higher attention values, while those with lower relevance receive smaller values — as the model learns these relationships.

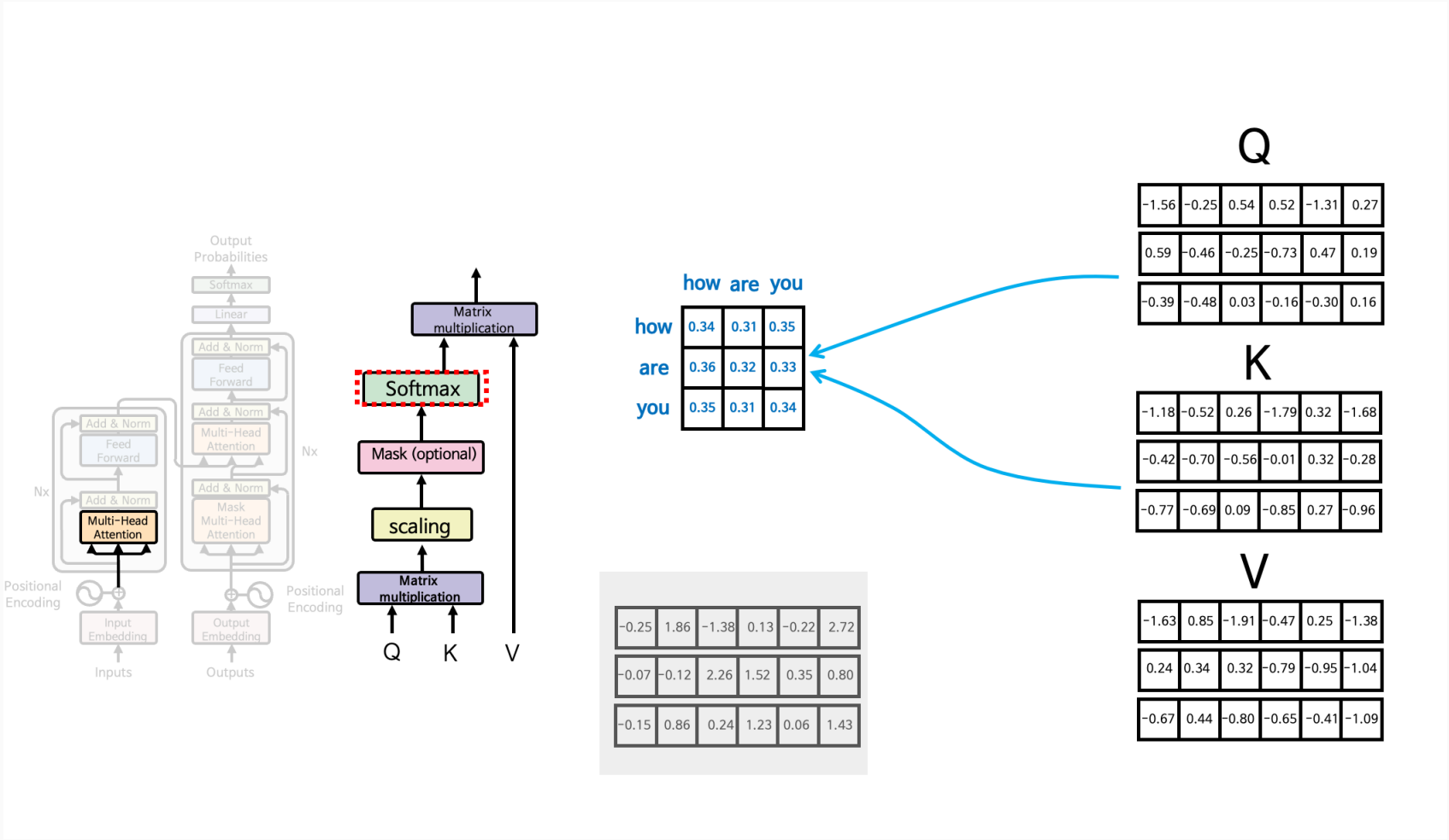


	how	are	you
how	0.34	0.31	0.35
are	0.36	0.32	0.33
you	0.35	0.31	0.34

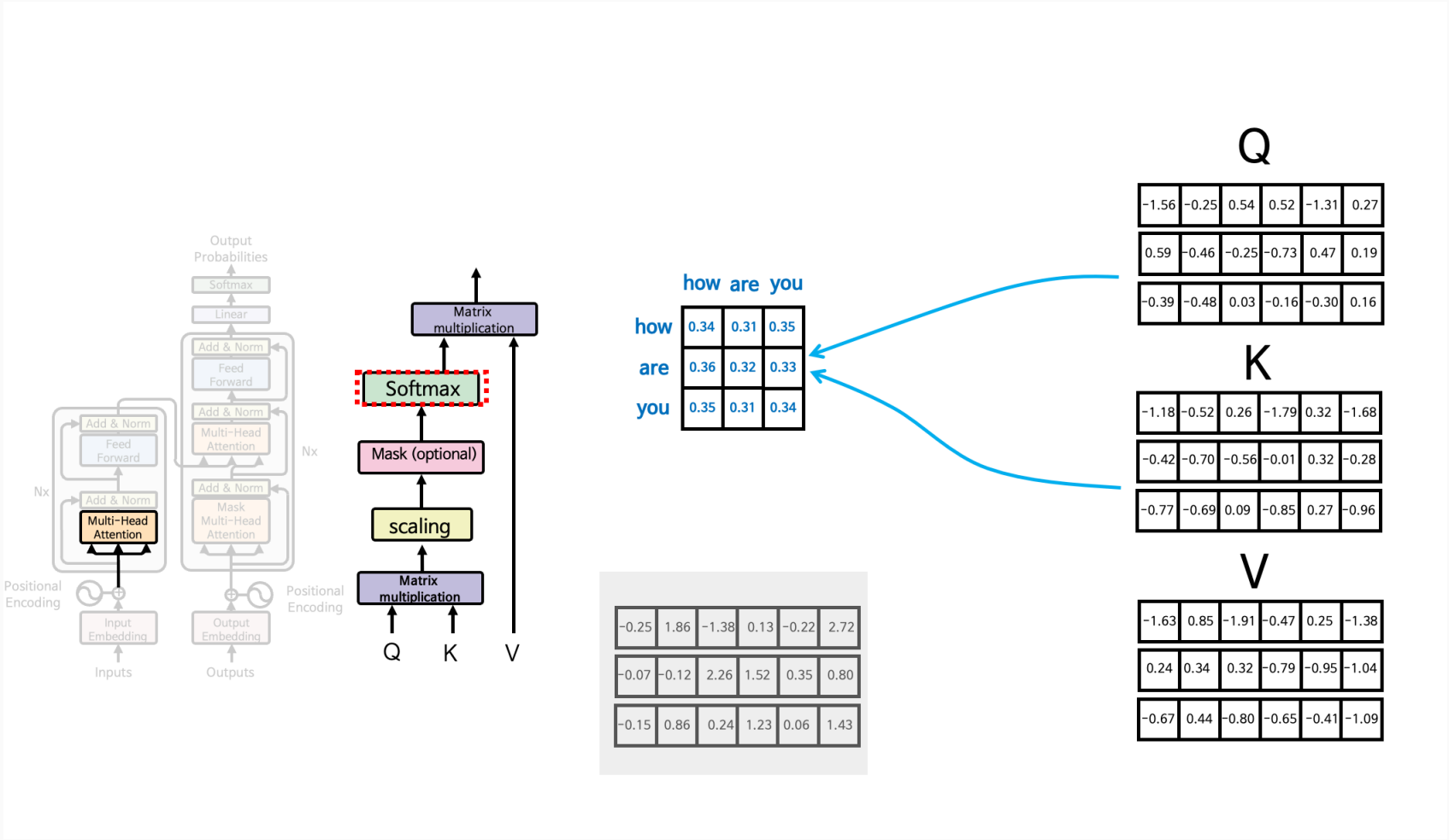
While *traditional* attention models focused on relationships between the input sequence and output sequence, self-attention takes the same input matrix and feeds it into two separate networks to produce the Q and K matrices.



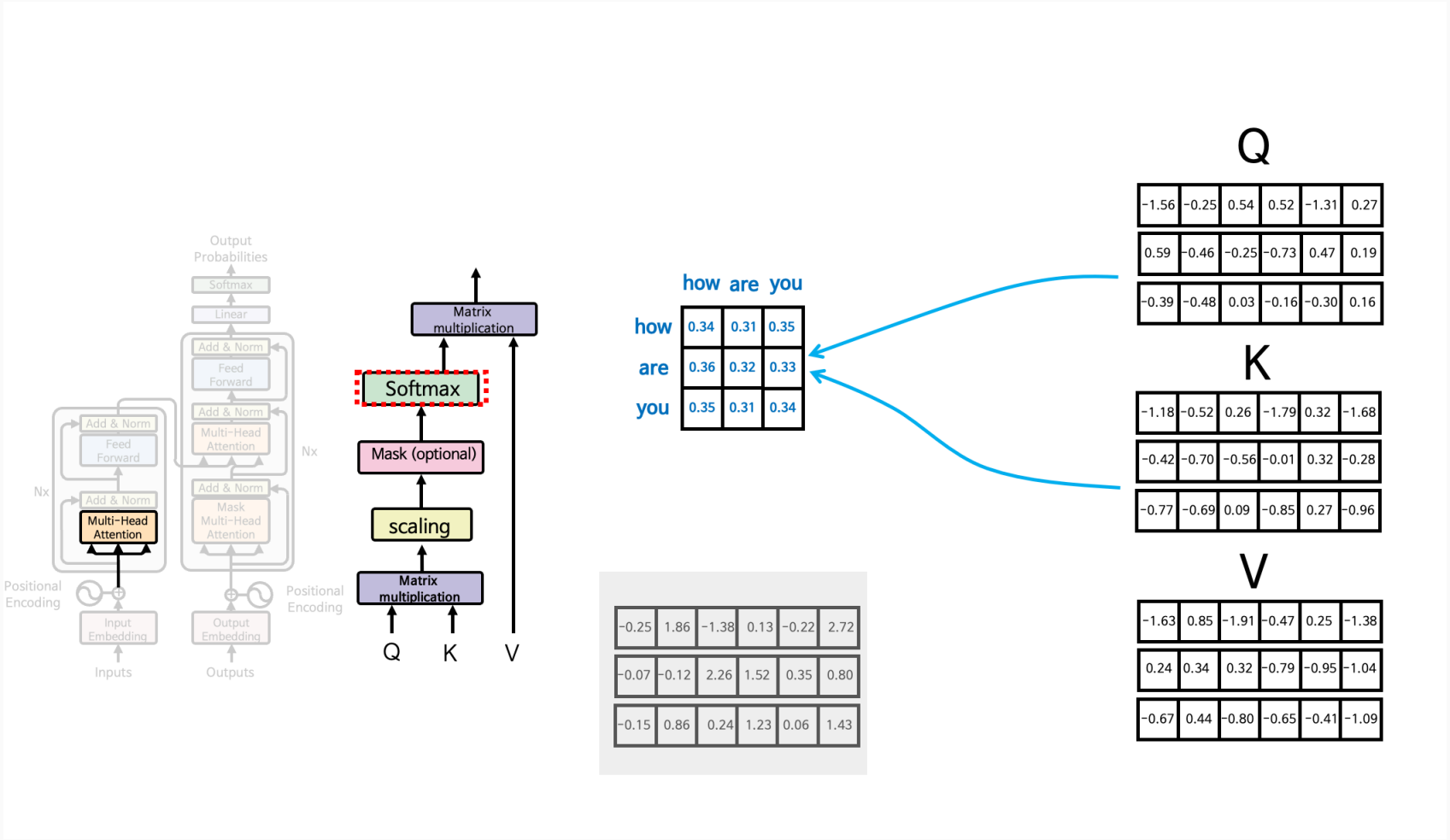
Amazingly, by simply multiplying these two matrices, the model can represent the correlations between words within a sentence.



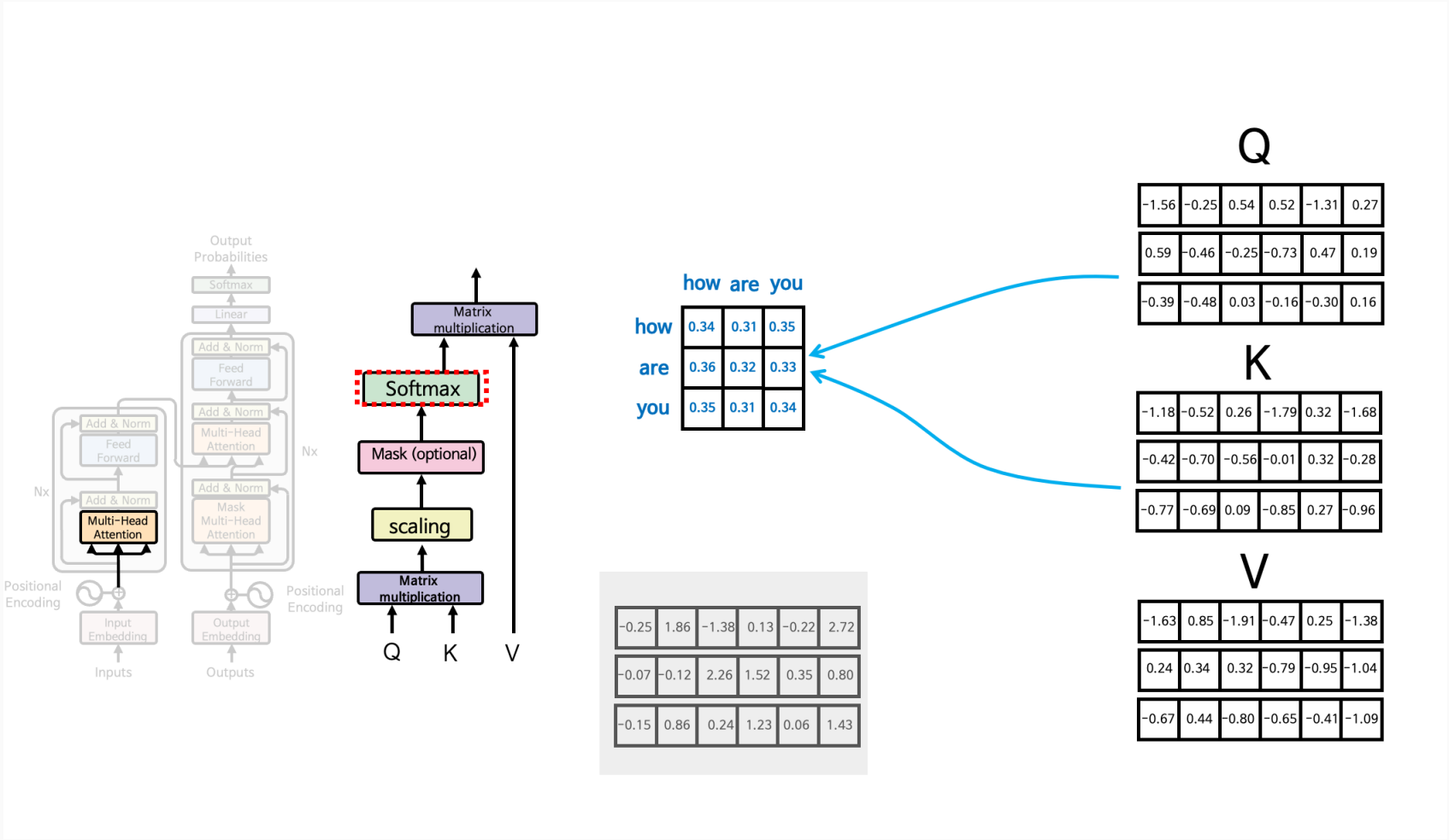
Because the model processes all words in parallel rather than sequentially, it achieves much faster computation.



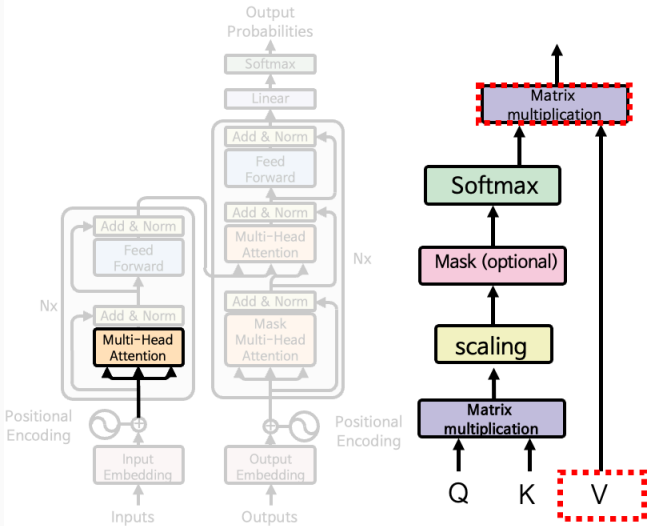
Even for long sentences, the model can calculate the attention between all pairs of words without bias or loss of information.



This self-attention mechanism is the core structure that made today's large language models (LLMs) possible.



So, what about the final matrix multiplication?



	how	are	you
how	0.34	0.31	0.35
are	0.36	0.32	0.33
you	0.35	0.31	0.34

Q

-1.56	-0.25	0.54	0.52	-1.31	0.27
0.59	-0.46	-0.25	-0.73	0.47	0.19
-0.39	-0.48	0.03	-0.16	-0.30	0.16

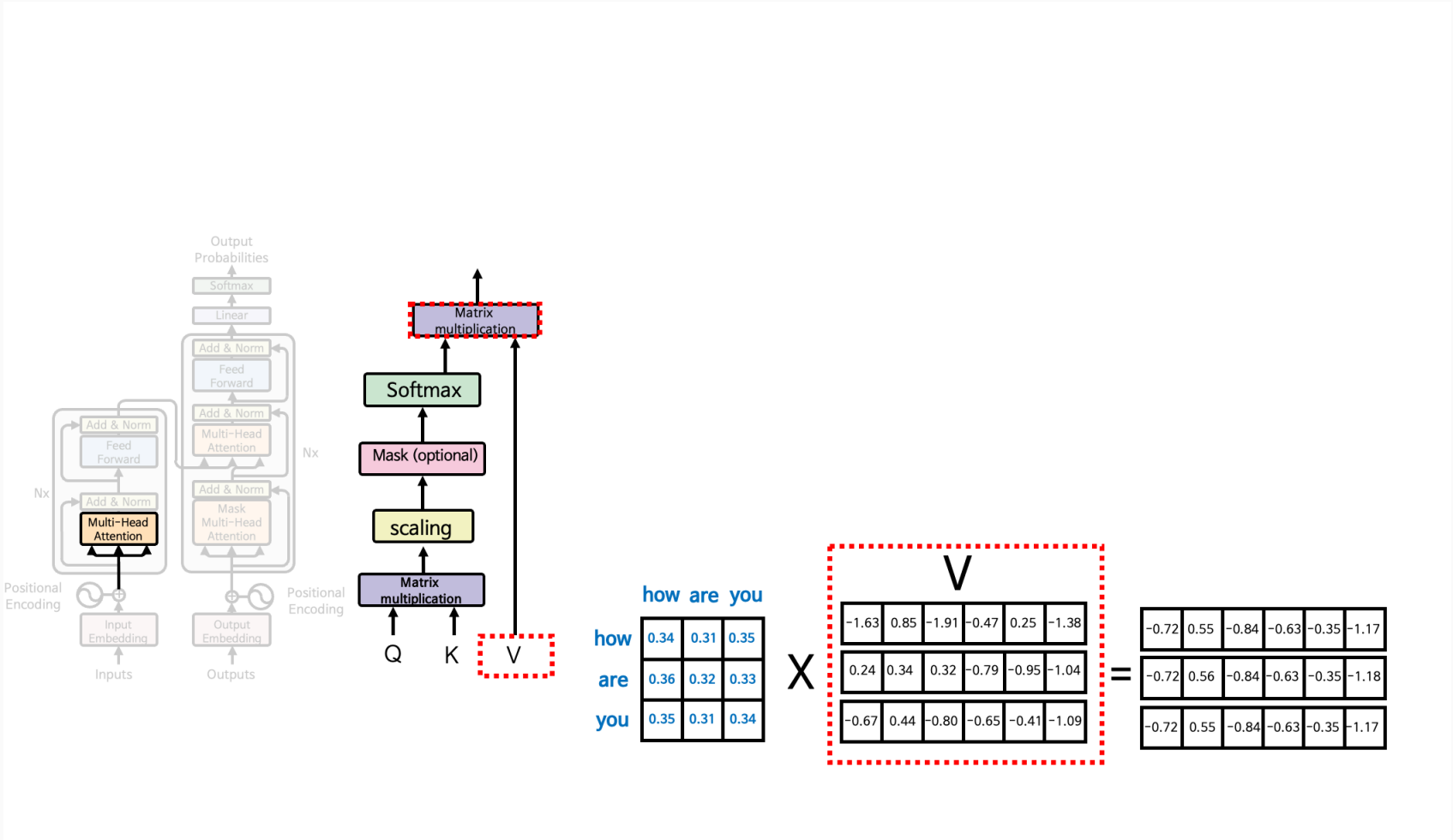
K

-1.18	-0.52	0.26	-1.79	0.32	-1.68
-0.42	-0.70	-0.56	-0.01	0.32	-0.28
-0.77	-0.69	0.09	-0.85	0.27	-0.96

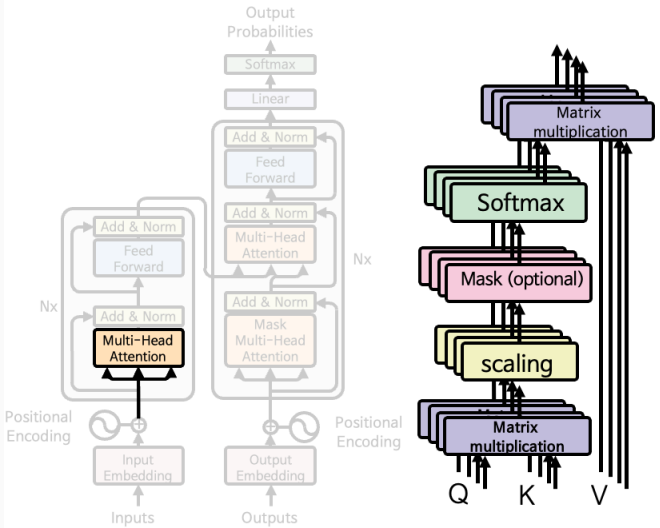
V

-1.63	0.85	-1.91	-0.47	0.25	-1.38
0.24	0.34	0.32	-0.79	-0.95	-1.04
-0.67	0.44	-0.80	-0.65	-0.41	-1.09

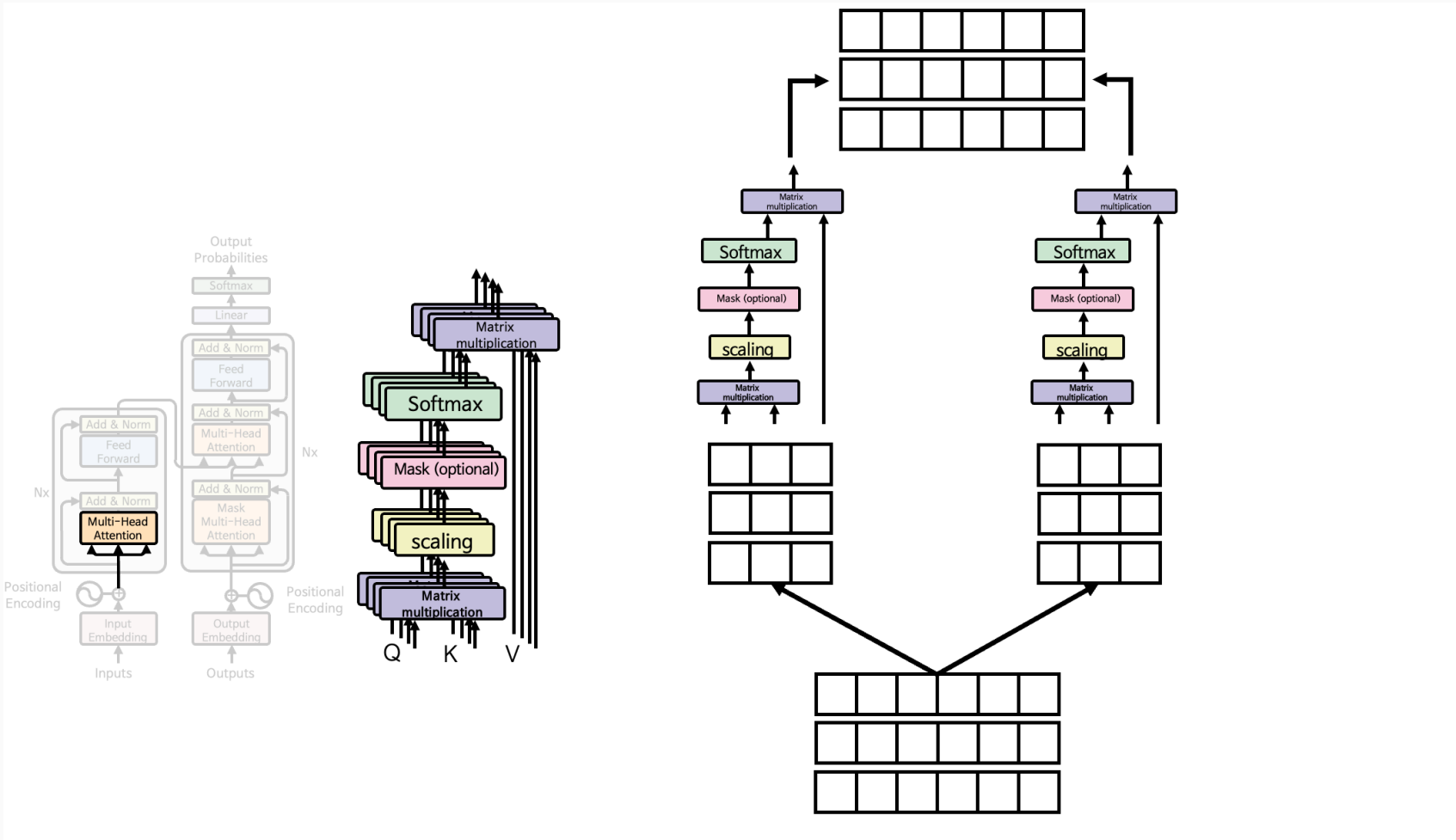
We multiply this by the V matrix to obtain the final output — a self-attended embedding that combines (1) input, (2) positional, and (3) attention information.



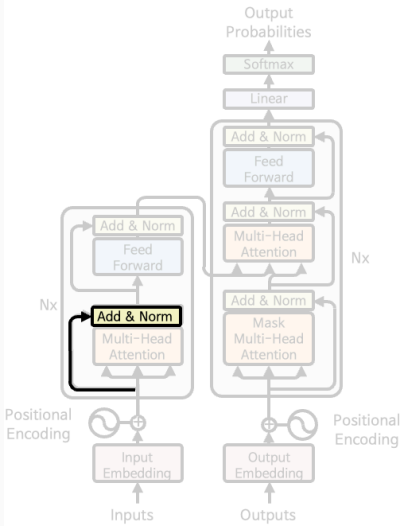
So far, we've assumed a single-head attention for simplicity, but in reality, the Transformer computes multi-head attention — the original paper uses 8 heads.



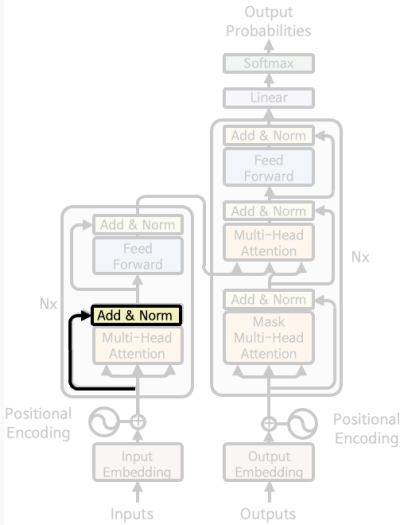
When using *multiple* heads, the model splits the dimensions of Q, K, and V according to the number of heads, performs self-attention separately for each head, then **concatenates** the resulting matrices and passes them through a fully connected layer to produce the final output of multi-head attention.



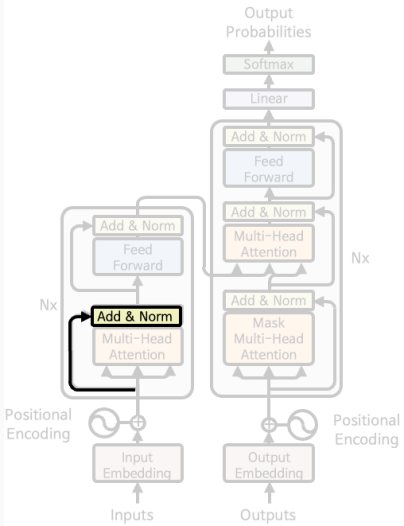
Next, we move on to the **addition and normalization** layer.



The **addition** step adds the multi-head output matrix to the initial input + positional embedding.

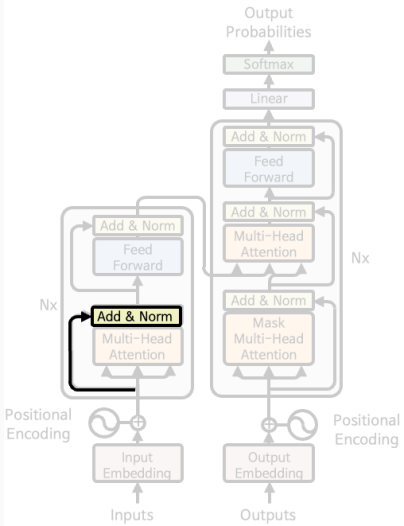


Let’s assume that the resulting matrix looks like this:

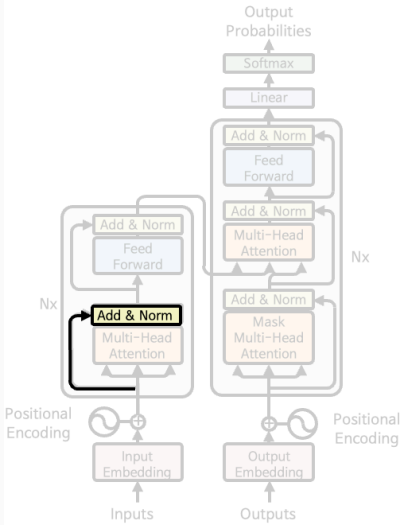


-0.59	1.99	-0.86	0.38	0.18	3.10
-0.41	0.02	2.79	1.77	0.76	1.19
-0.48	0.99	0.77	1.48	0.47	1.81

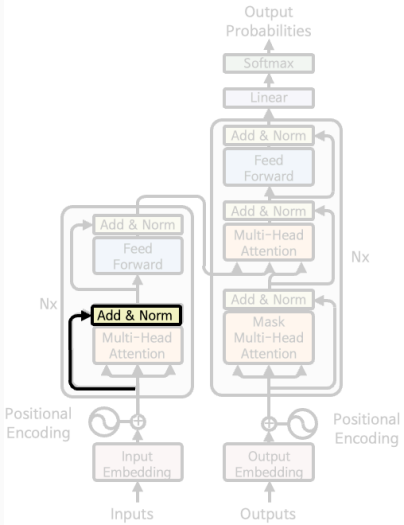
Next, for the normalization step, we first calculate the mean and standard deviation for each column.



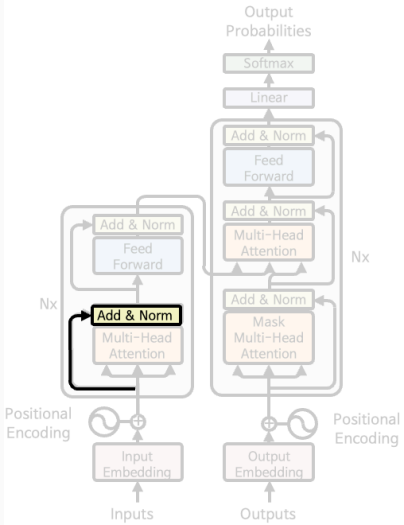
						M	SD
-0.59	1.99	-0.86	0.38	0.18	3.10	0.70	1.41
-0.41	0.02	2.79	1.77	0.76	1.19		
-0.48	0.99	0.77	1.48	0.47	1.81		



							M	SD
-0.59	1.99	-0.86	0.38	0.18	3.10	→	0.70	1.41
-0.41	0.02	2.79	1.77	0.76	1.19	→	1.02	1.07
-0.48	0.99	0.77	1.48	0.47	1.81			

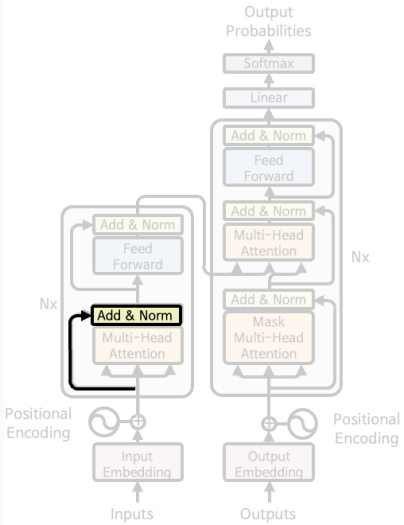


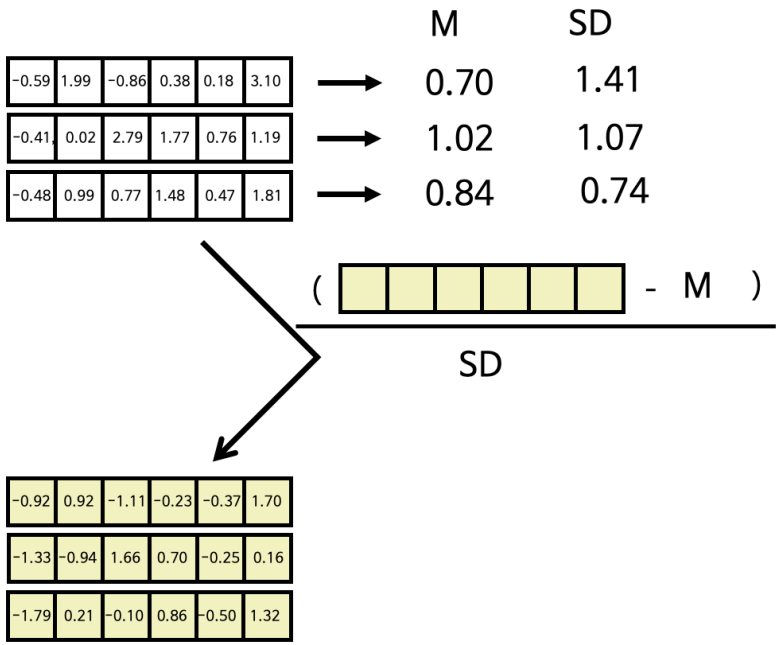
							M	SD
-0.59	1.99	-0.86	0.38	0.18	3.10	→	0.70	1.41
-0.41	0.02	2.79	1.77	0.76	1.19	→	1.02	1.07
-0.48	0.99	0.77	1.48	0.47	1.81	→	0.84	0.74



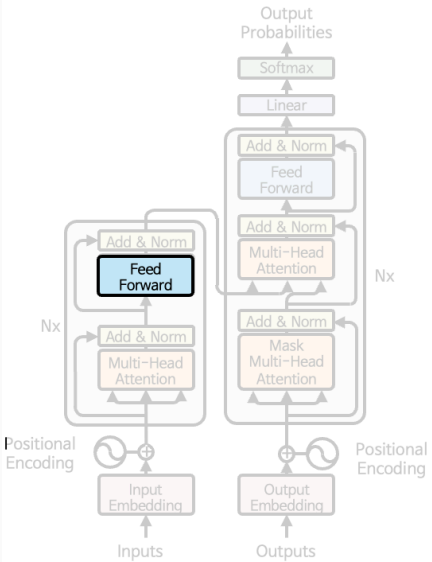
							M	SD
-0.59	1.99	-0.86	0.38	0.18	3.10	→	0.70	1.41
-0.41	0.02	2.79	1.77	0.76	1.19	→	1.02	1.07
-0.48	0.99	0.77	1.48	0.47	1.81	→	0.84	0.74

$$\frac{(\square\square\square\square\square\square - M)}{SD}$$


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Mask Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs


M
SD
-0.59 1.99 -0.86 0.38 0.18 3.10
0.70
1.41
-0.41 0.02 2.79 1.77 0.76 1.19
1.02
1.07
-0.48 0.99 0.77 1.48 0.47 1.81
0.84
0.74
(- M)
SD
-0.92 0.92 -1.11 -0.23 -0.37 1.70
-1.33 -0.94 1.66 0.70 -0.25 0.16
-1.79 0.21 -0.10 0.86 -0.50 1.32

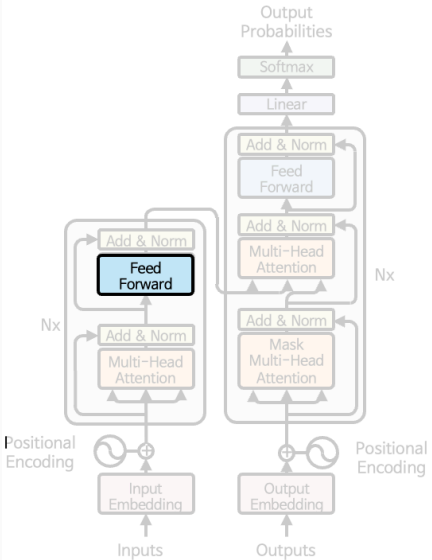
Next is the **Feed-Forward** layer.



-0.92	0.92	-1.11	-0.23	-0.37	1.70
-1.33	-0.94	1.66	0.70	-0.25	0.16
-1.79	0.21	-0.10	0.86	-0.50	1.32

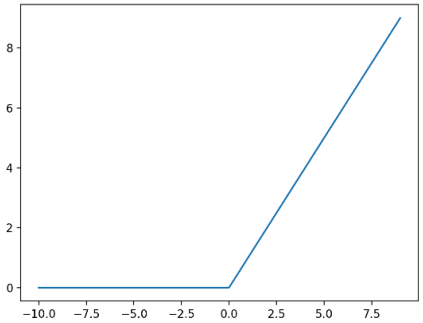
The Feed-Forward layer is a simple neural network consisting of **two layers** and using **ReLU()** as the activation function.

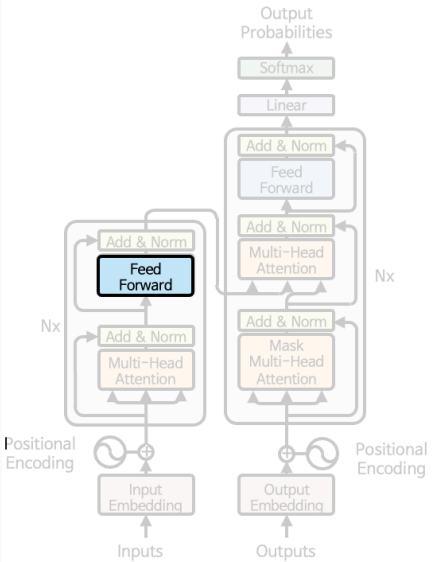
$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2$$



-0.92	0.92	-1.11	-0.23	-0.37	1.70
-1.33	-0.94	1.66	0.70	-0.25	0.16
-1.79	0.21	-0.10	0.86	-0.50	1.32

ReLU function





-0.92	0.92	-1.11	-0.23	-0.37	1.70
-1.33	-0.94	1.66	0.70	-0.25	0.16
-1.79	0.21	-0.10	0.86	-0.50	1.32

X

W_1

0.31	0.20	-0.35	-0.07	-0.13	0.23
0.15	0.34	-0.38	-0.29	-0.30	-0.28
-0.28	0.21	0.17	-0.27	-0.07	0.21
-0.40	0.10	-0.20	0.14	-0.20	0.19
-0.33	-0.00	-0.08	-0.12	-0.12	0.29
0.30	-0.31	-0.23	0.25	-0.39	-0.31

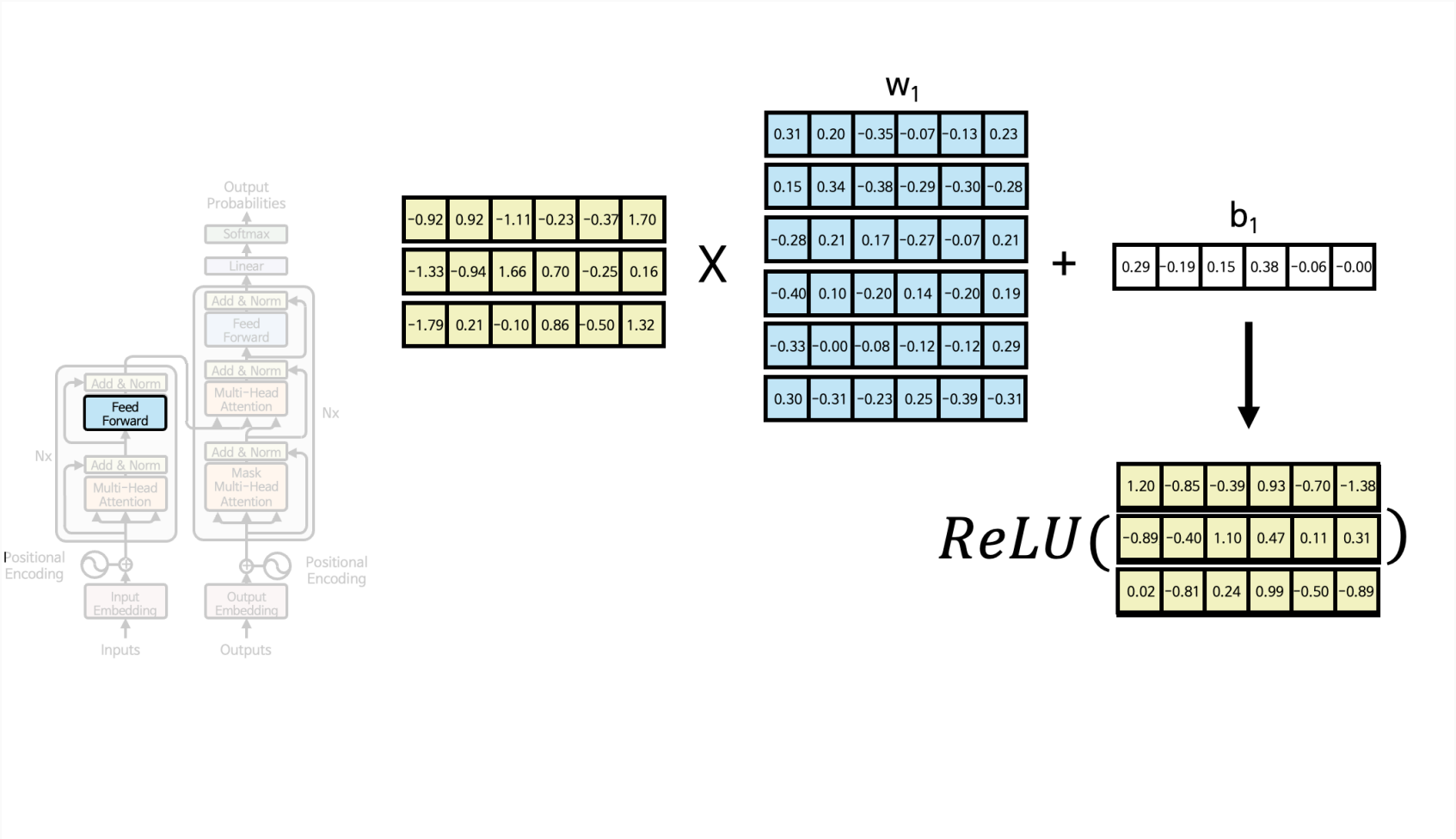
+

b_1

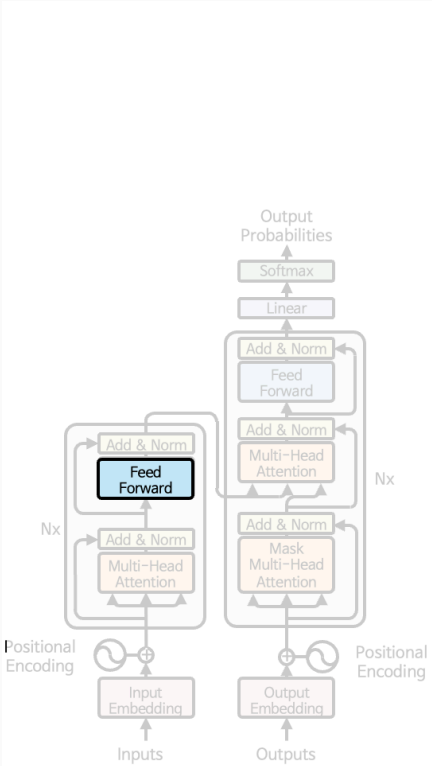
0.29	-0.19	0.15	0.38	-0.06	-0.00

1.20	-0.85	-0.39	0.93	-0.70	-1.38
-0.89	-0.40	1.10	0.47	0.11	0.31
0.02	-0.81	0.24	0.99	-0.50	-0.89

Applying the ReLU activation function...



Applying the ReLU activation function...



-0.92	0.92	-1.11	-0.23	-0.37	1.70
-1.33	-0.94	1.66	0.70	-0.25	0.16
-1.79	0.21	-0.10	0.86	-0.50	1.32

X

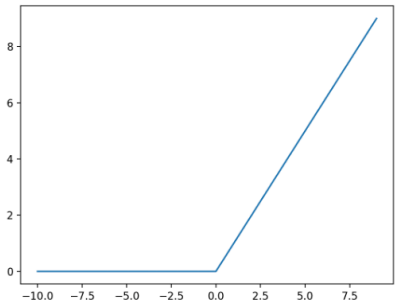
w_1

0.31	0.20	-0.35	-0.07	-0.13	0.23
0.15	0.34	-0.38	-0.29	-0.30	-0.28
-0.28	0.21	0.17	-0.27	-0.07	0.21
-0.40	0.10	-0.20	0.14	-0.20	0.19
-0.33	-0.00	-0.08	-0.12	-0.12	0.29
0.30	-0.31	-0.23	0.25	-0.39	-0.31

+

b_1

0.29	-0.19	0.15	0.38	-0.06	-0.00

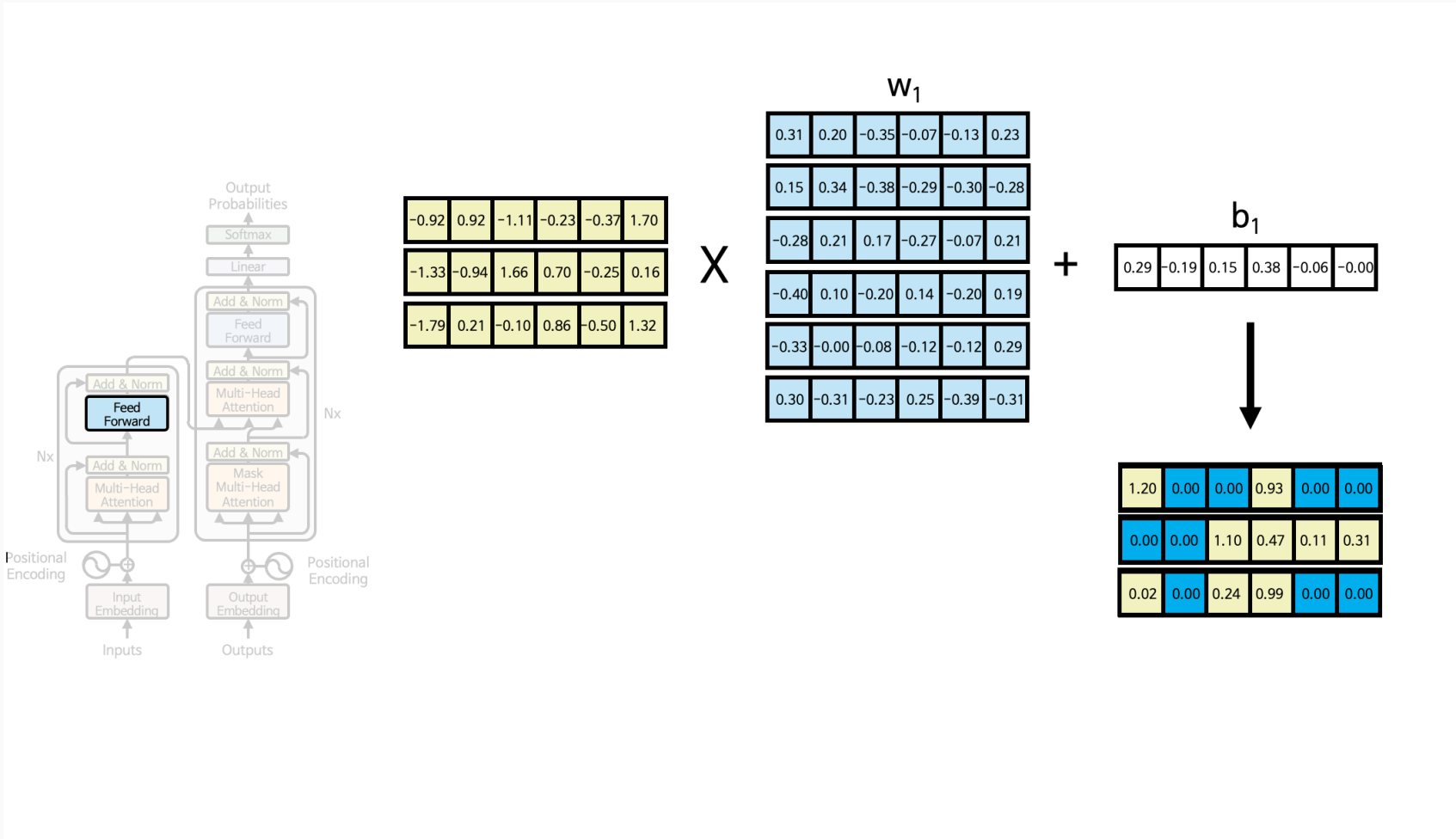


$ReLU($

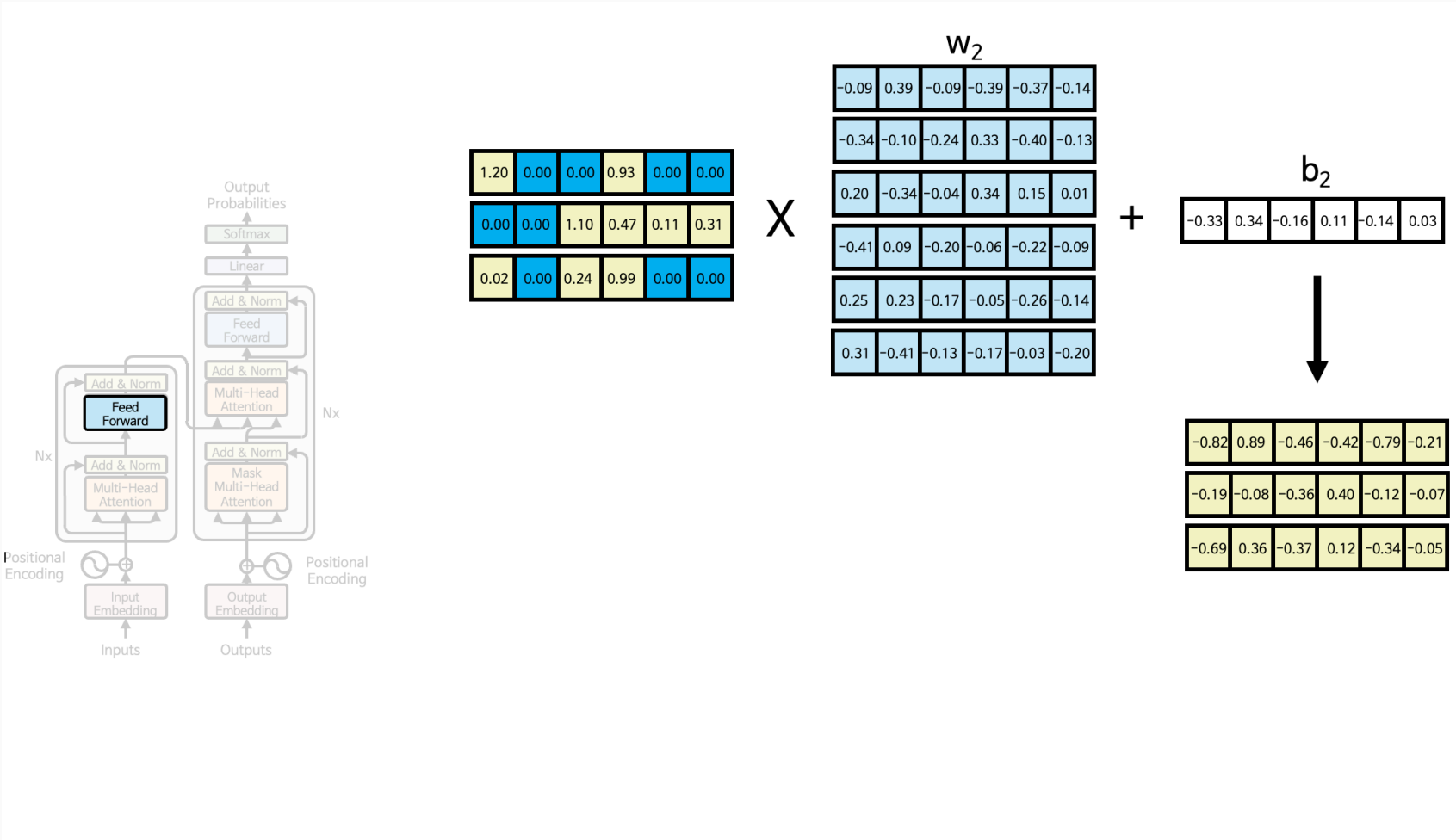
1.20	-0.85	-0.39	0.93	-0.70	-1.38
-0.89	-0.40	1.10	0.47	0.11	0.31
0.02	-0.81	0.24	0.99	-0.50	-0.89

$)$

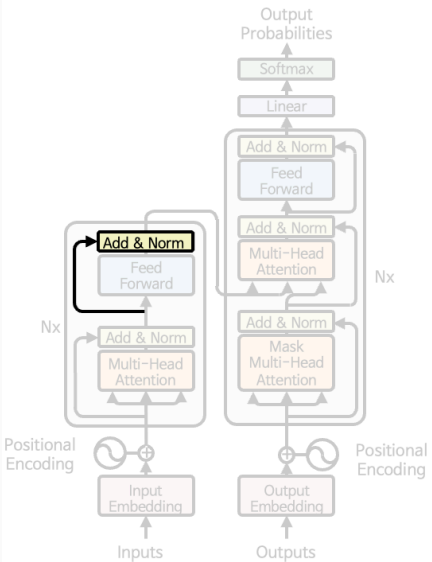
Negative values become 0.



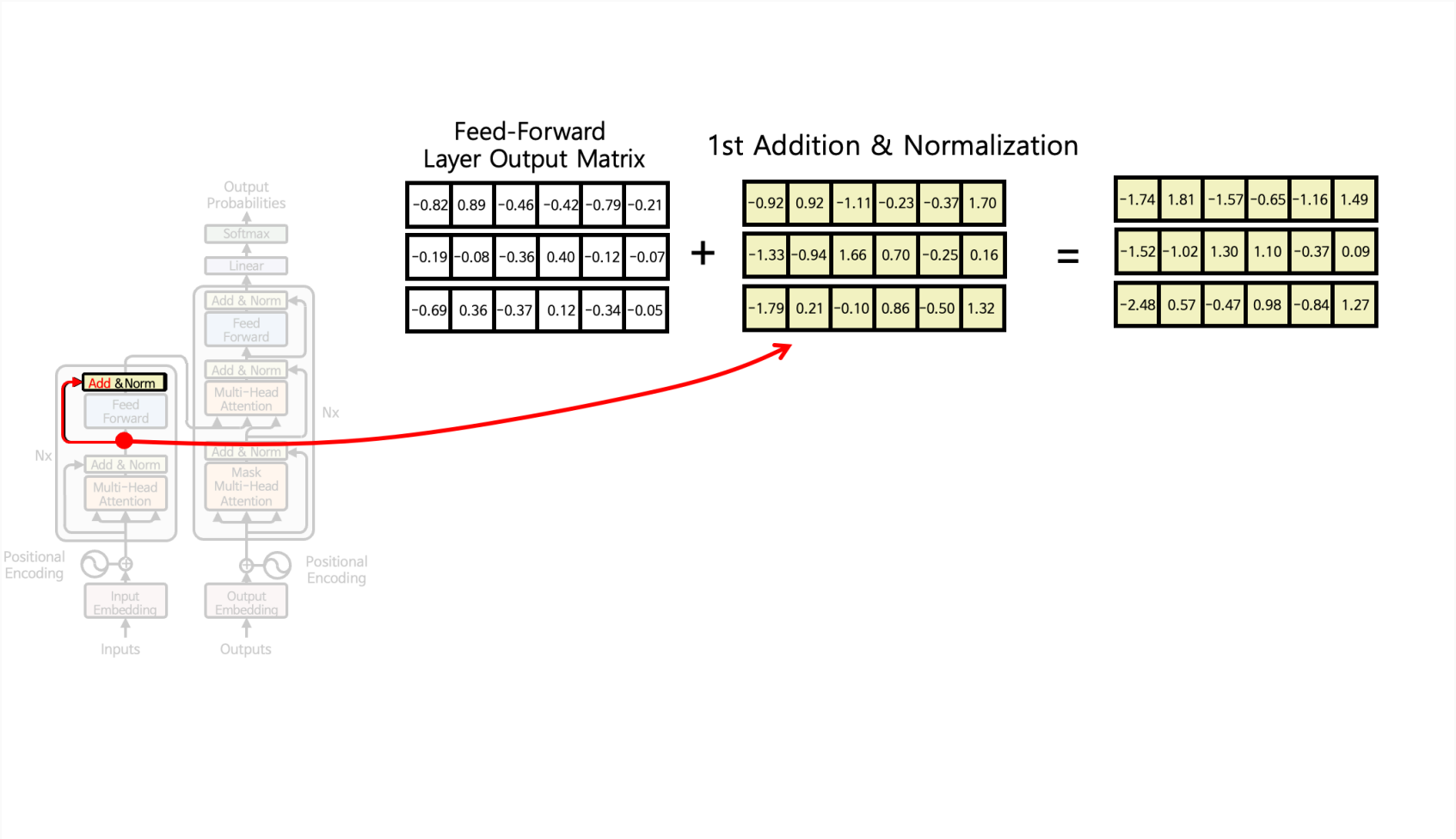
Then, by multiplying with the second layer's weights and biases, we obtain the final output of the Feed-Forward layer. The purpose of this layer is to increase **non-linearity**, allowing the network to better process and distinguish complex patterns (recall multi-layer perceptrons!)



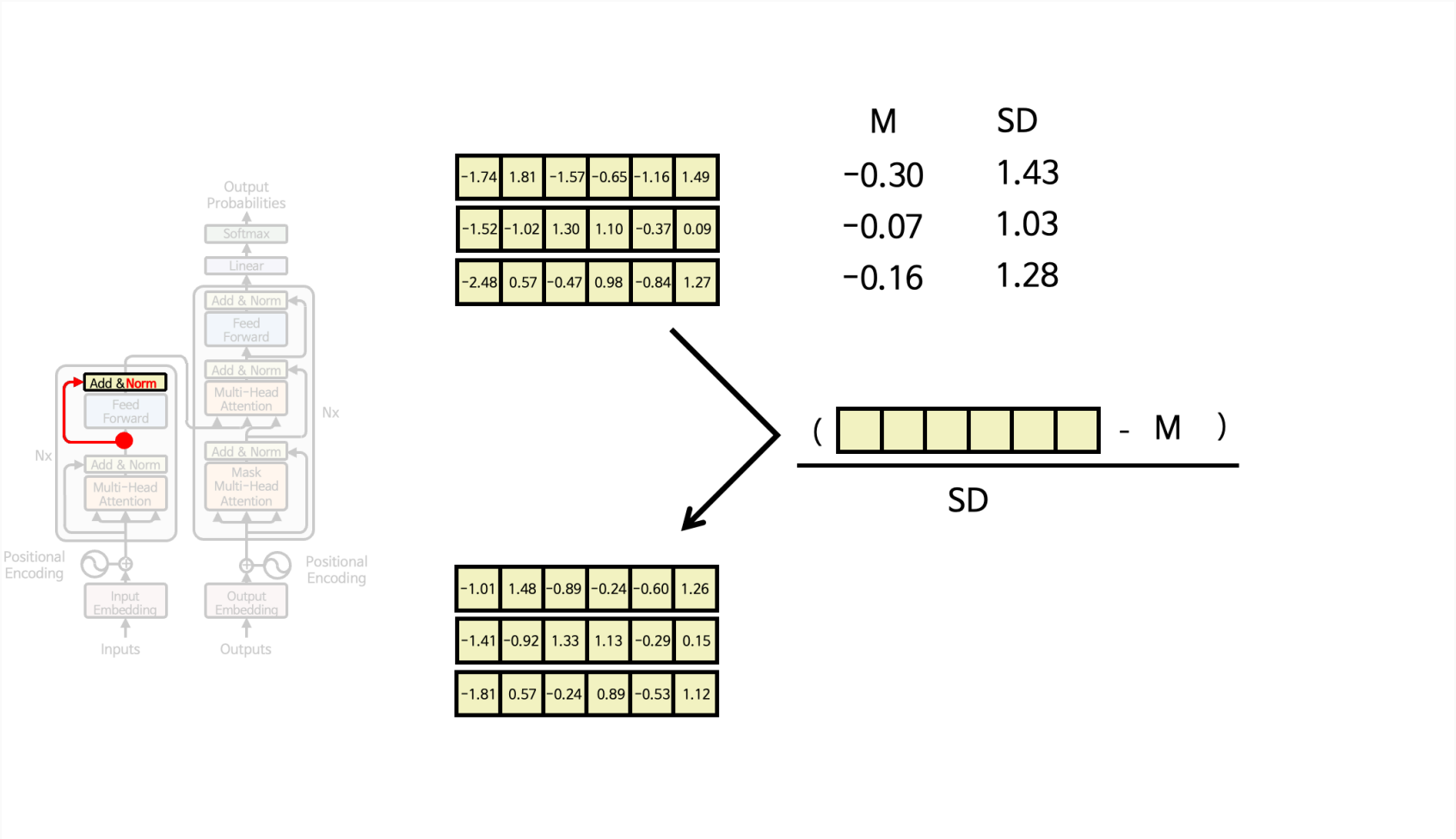
Next, we repeat the same **Add & Normalize** process.



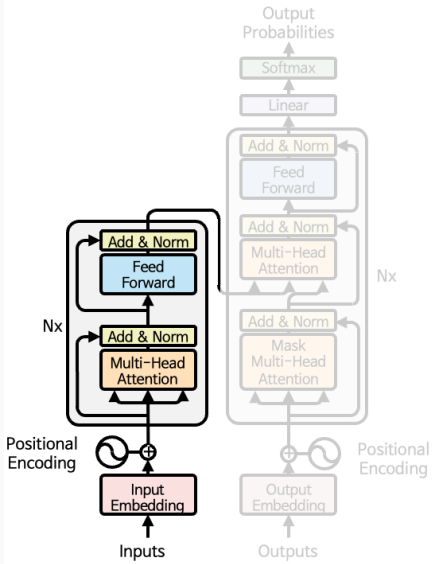
As before, we first calculate the sum of the two matrices.



Then, we normalize the matrix again:

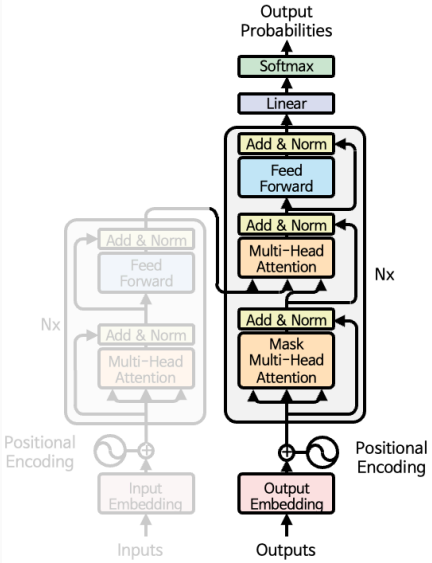


This resulting matrix is the **final output of the encoder.**

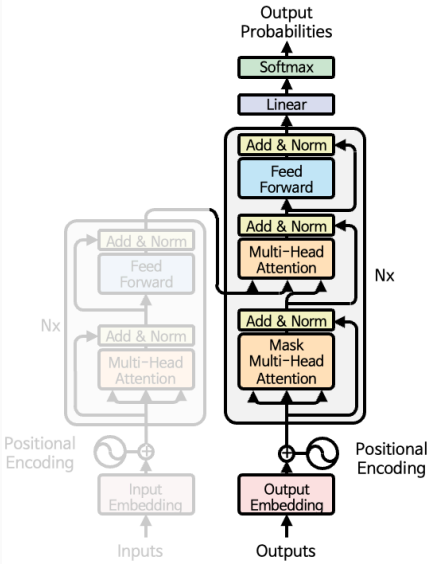


-1.01	1.48	-0.89	-0.24	-0.60	1.26
-1.41	-0.92	1.33	1.13	-0.29	0.15
-1.81	0.57	-0.24	0.89	-0.53	1.12

Now, let’s move on to the **Decoder**.



Just like the encoder, the decoder first performs **word encoding** and **positional encoding** of the output (target) sequence.

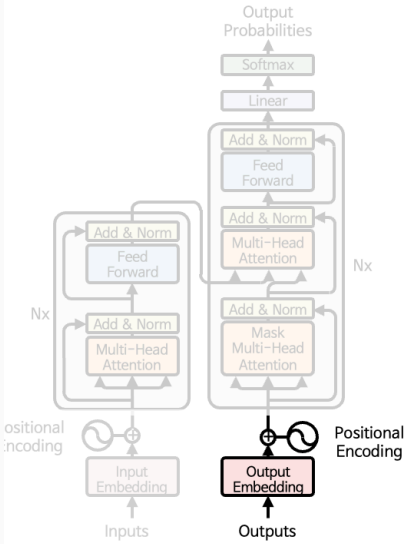


Decoder input: [⟨sos⟩, I, am] → Index: [0, 6, 3]

Decoder output target : [I, am, fine] → Index : [6, 3, 4]

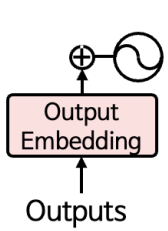
The process of word encoding and positional encoding is identical to that in the encoder. In fact, the positional encoding values can be reused from the encoder.

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



0	1	2	3	4	5	
0	1	0	1	0	1	0 pos
0.84	0.54	0.05	1	0	1	1 pos
0.91	-0.42	0.09	1	0	1	2 pos

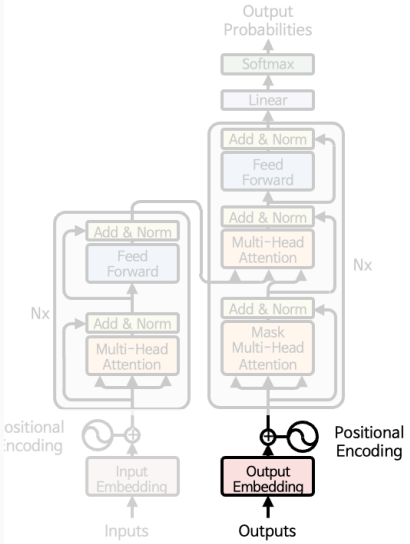
6



6

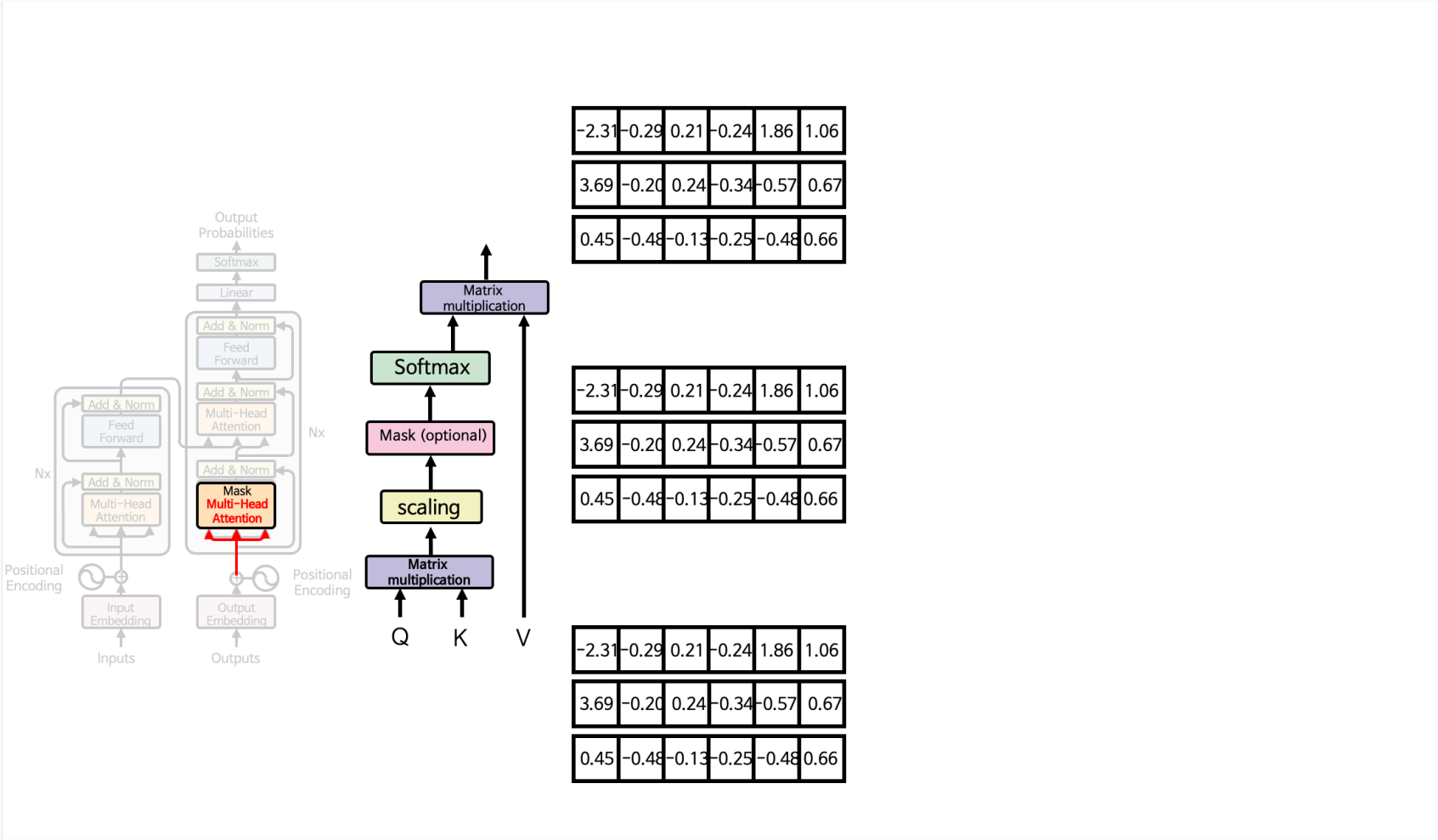
0, ⟨sos⟩	-2.31	-1.29	0.21	-1.24	1.86	0.06
6, I	2.85	-0.74	0.20	-1.34	-0.57	-0.33
3, am	-0.46	-0.06	-0.22	-1.25	-0.49	-0.34

Here’s how it looks:

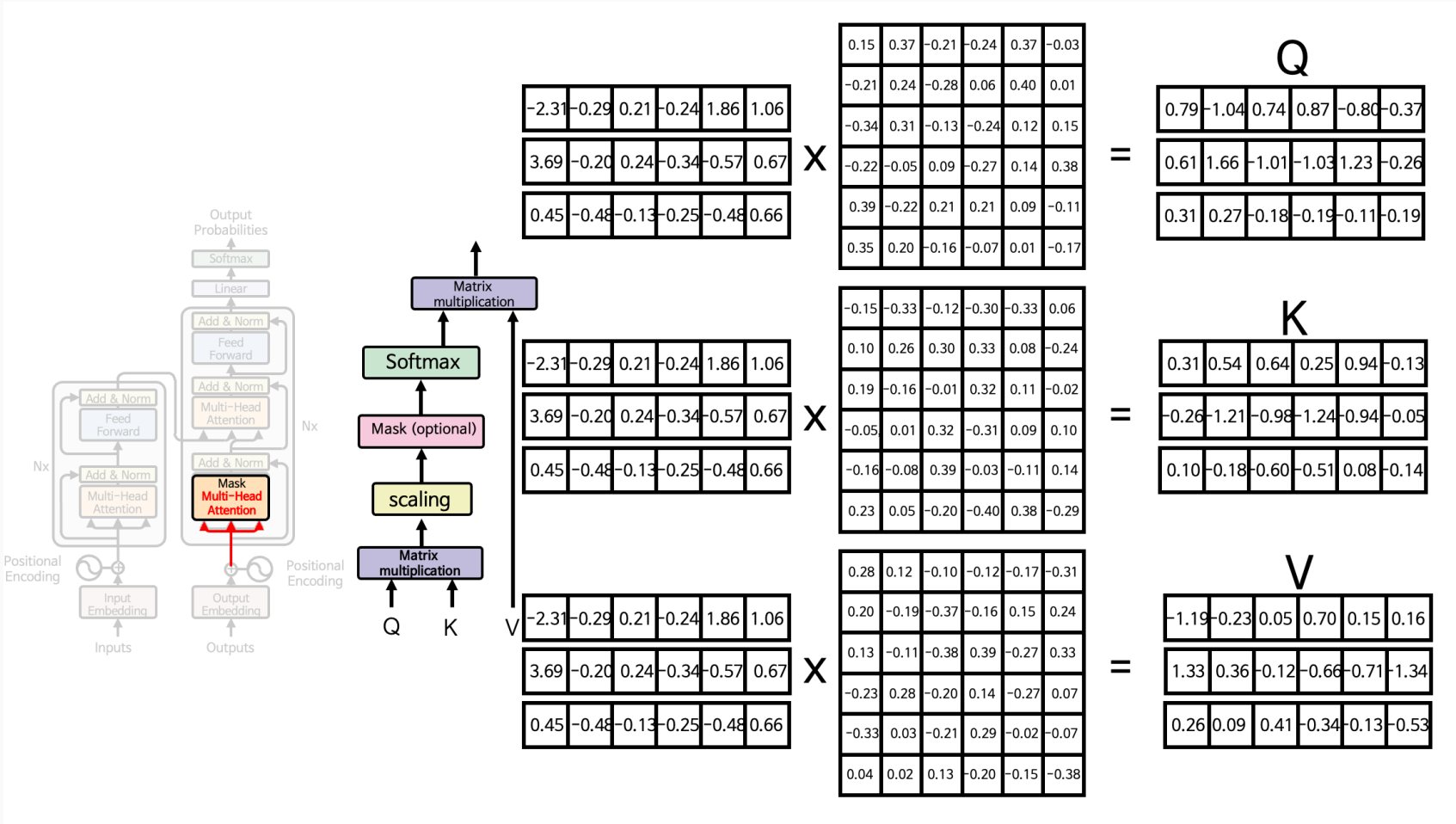


0	-2.31	-1.29	0.21	-1.24	1.86	0.06
0.	2.85	-0.74	0.20	-1.34	-0.57	-0.33
0.	-0.46	-0.06	-0.22	-1.25	-0.49	-0.34

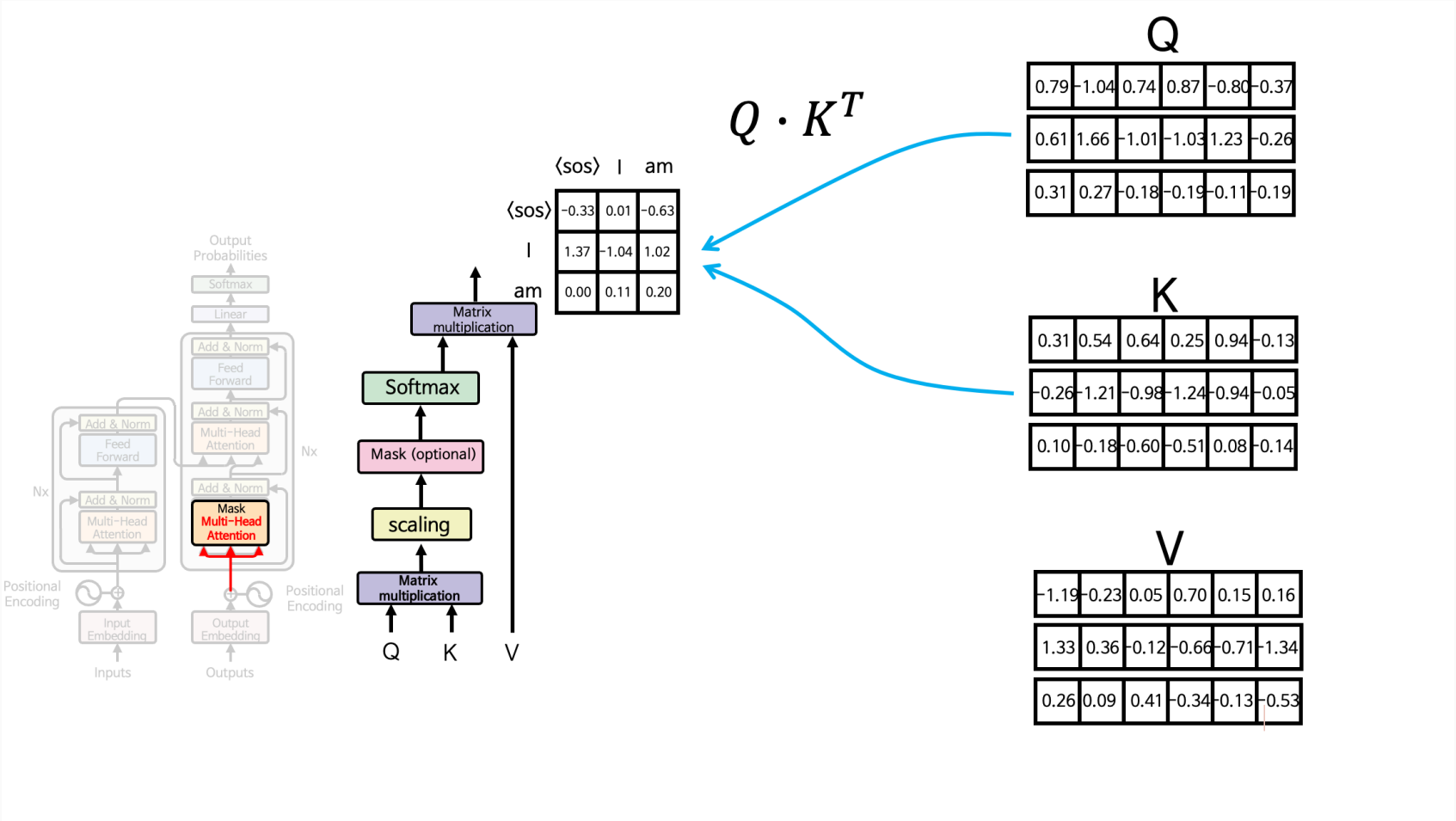
The **Masked Multi-Head Attention** operation in the decoder is almost the same as in the encoder.



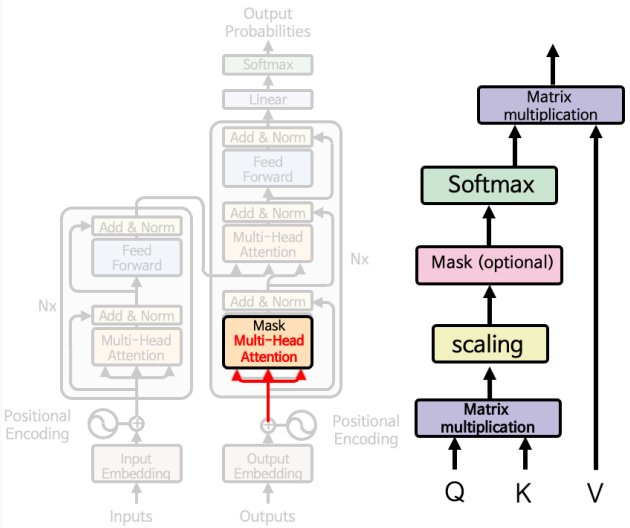
The **Masked Multi-Head Attention** operation in the decoder is almost the same as in the encoder.



We multiply the Q and K matrices to create the attention matrix.



We then scale the matrix by dividing it by $\sqrt{6}$, just as before, so that the range of values changes accordingly.

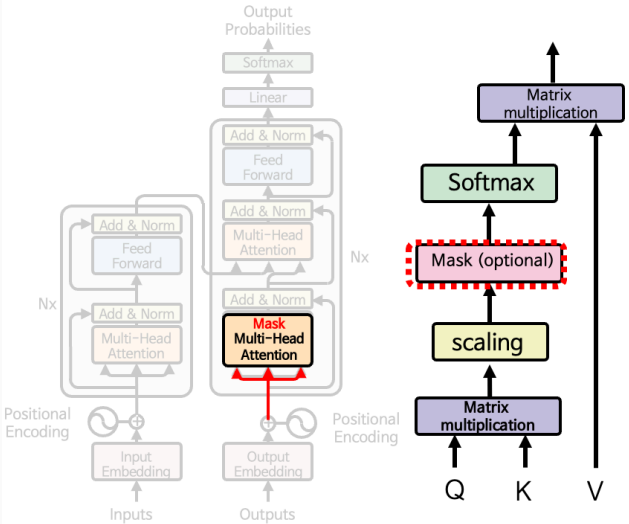


	⟨sos⟩	I	am
⟨sos⟩	-0.33	0.01	-0.63
I	1.37	-1.04	1.02
am	0.00	0.11	0.20

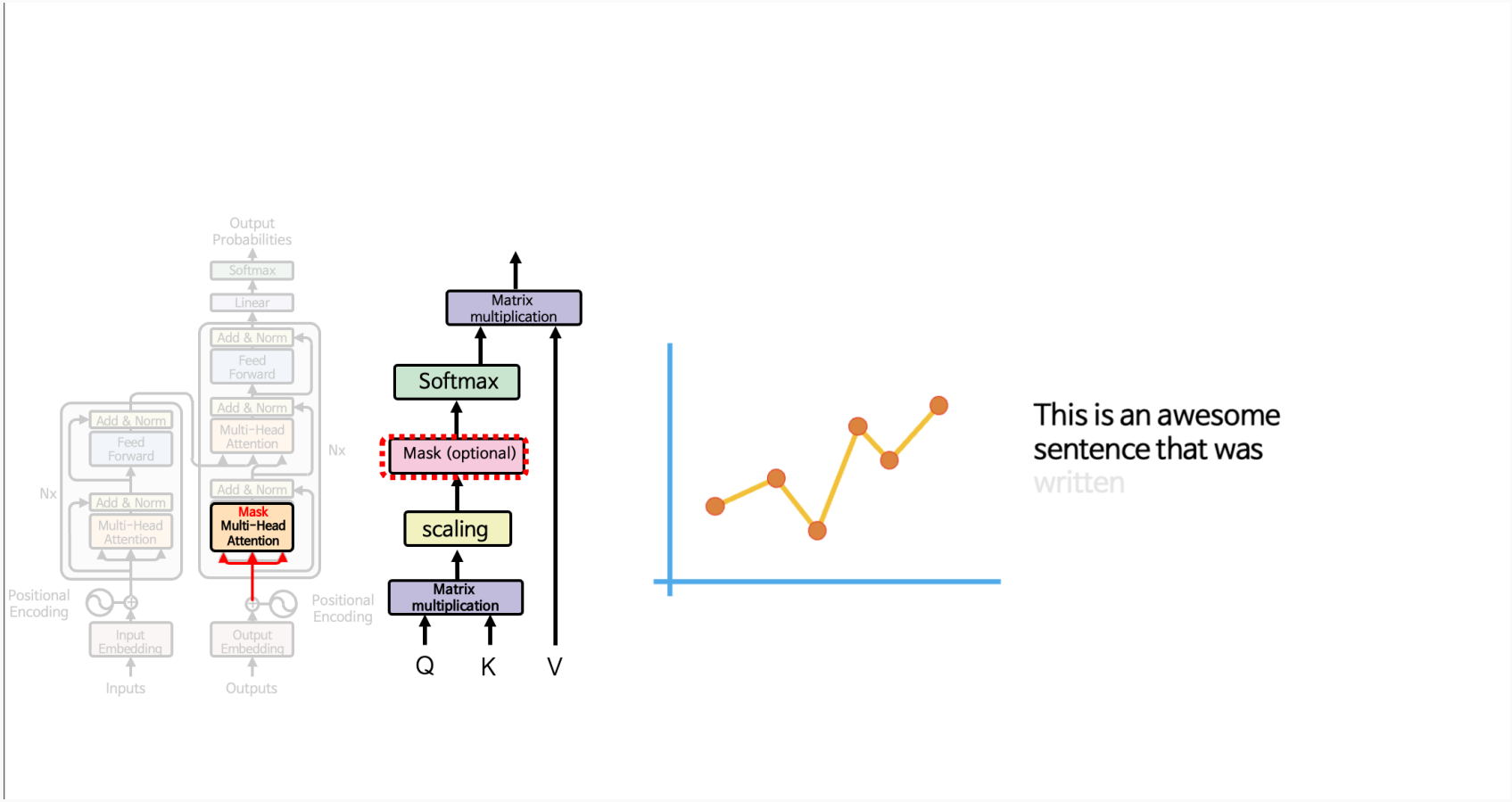
$\div \sqrt{6} =$

-0.13	0.00	-0.26
0.56	-0.43	0.42
0.00	0.04	0.08

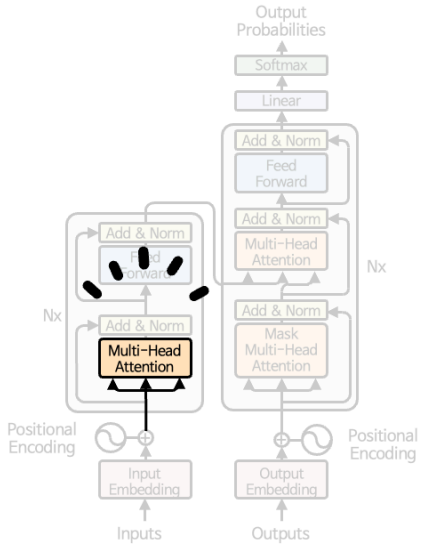
Now, let’s explore the key concept of the decoder’s multi-head attention — **masking**.

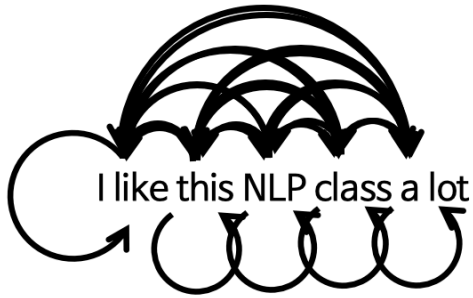


The goal of the transformer decoder is to generate the output word sequence, one token at a time (recall: language modeling).

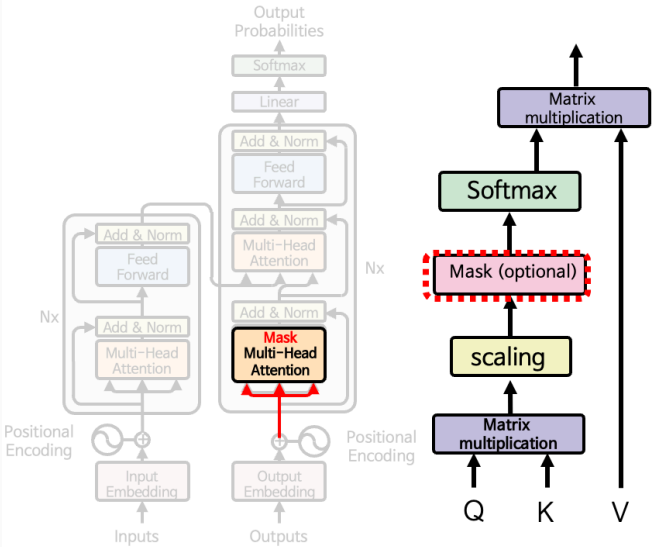


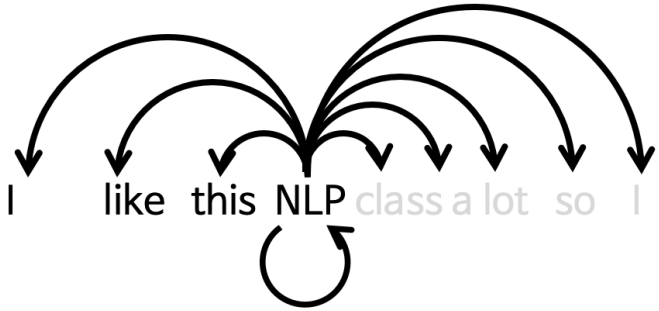
While the encoder needs to consider all tokens in the input sentence to understand the full meaning,



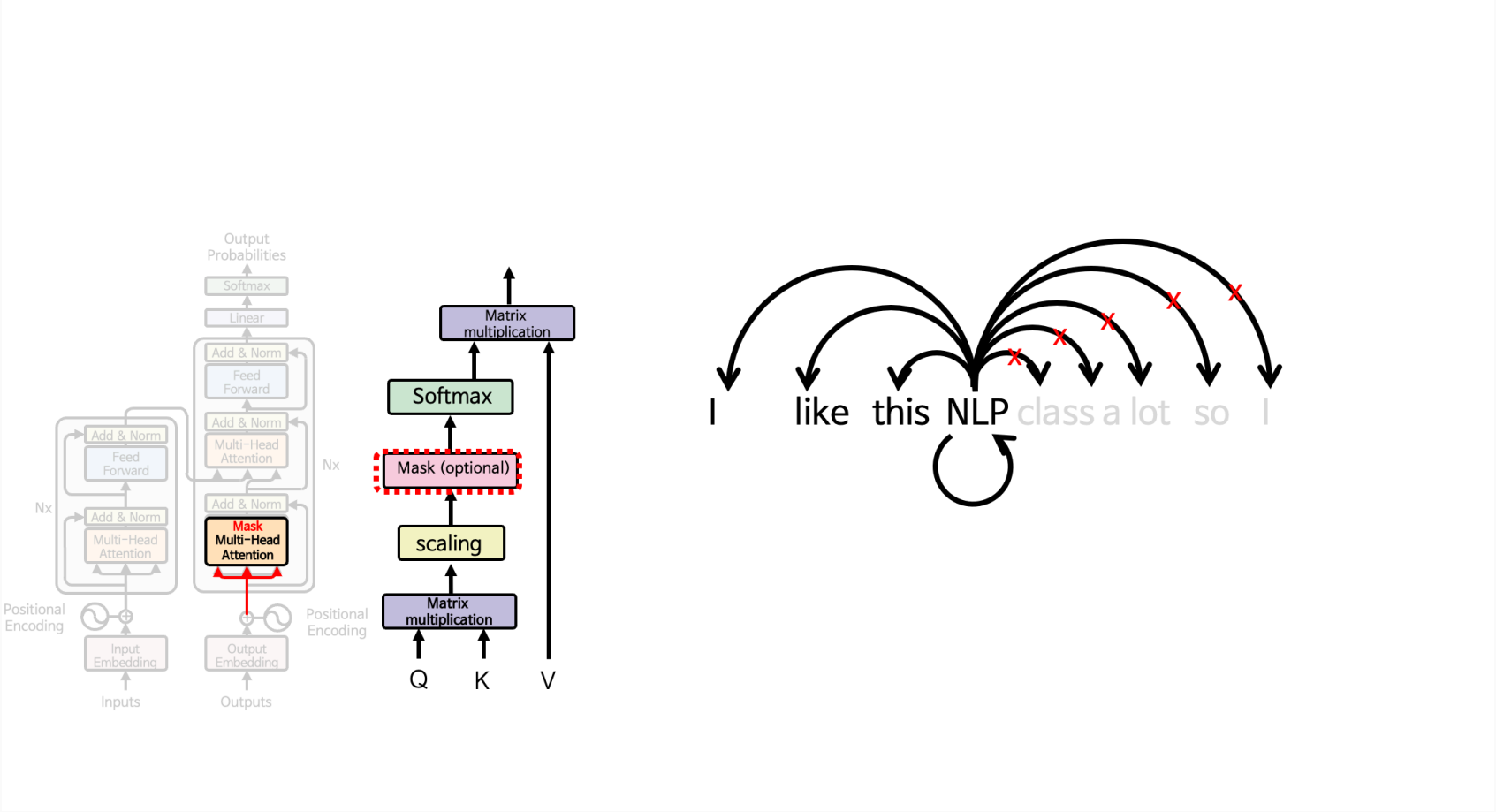


the decoder generates output **one word at a time**.

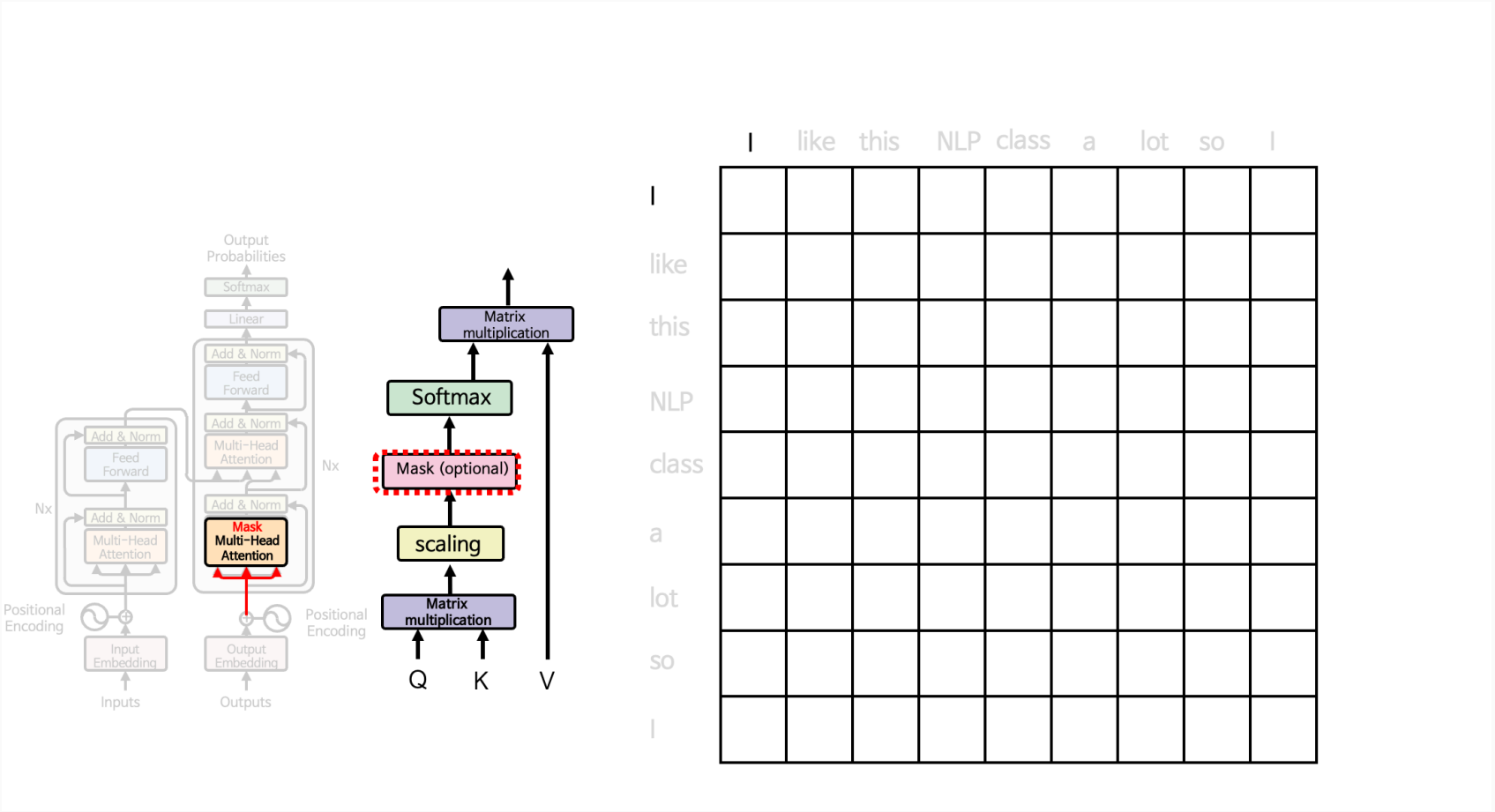




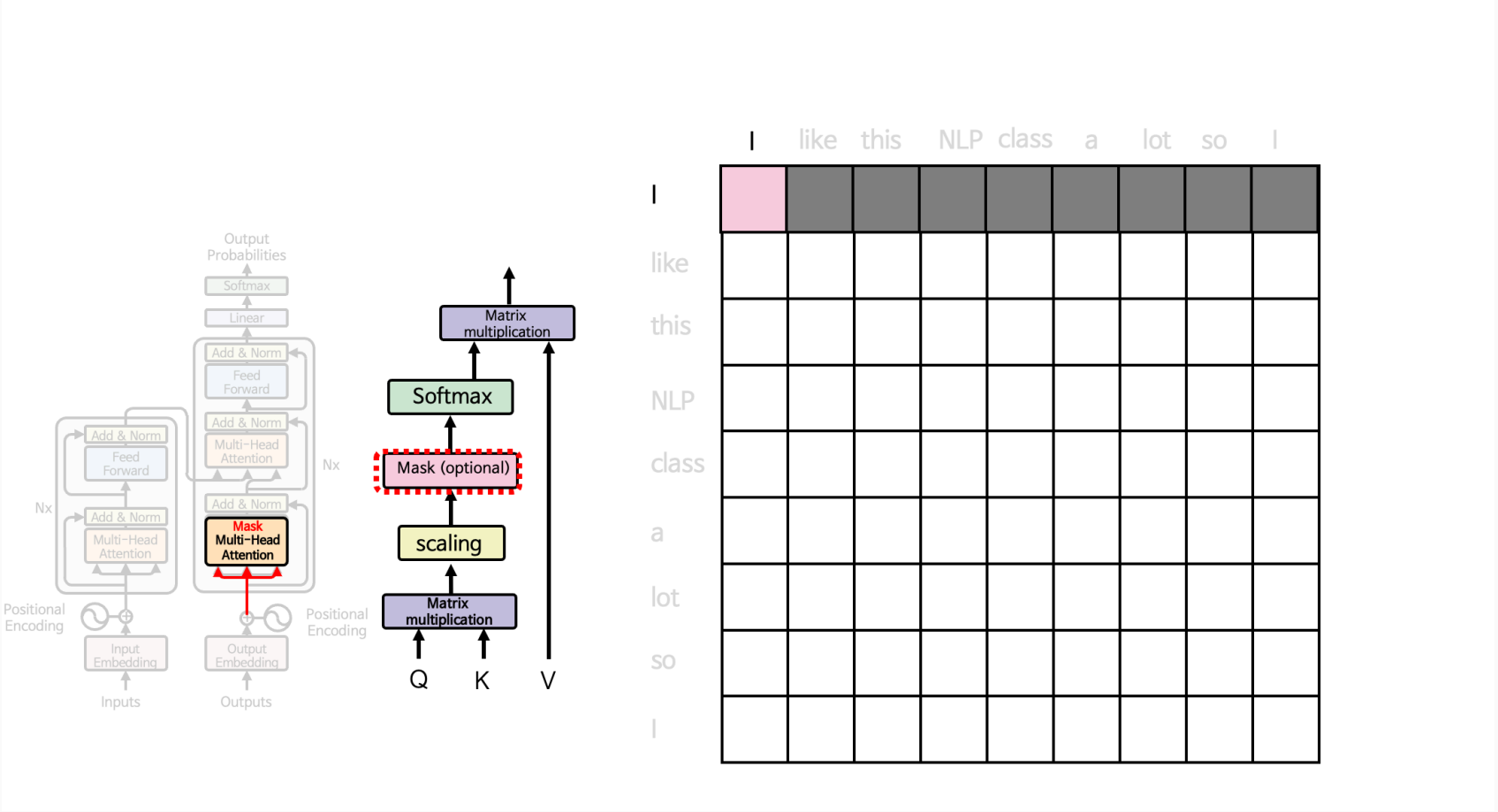
Therefore, it’s natural that the decoder should NOT attend to words that haven’t been generated yet.

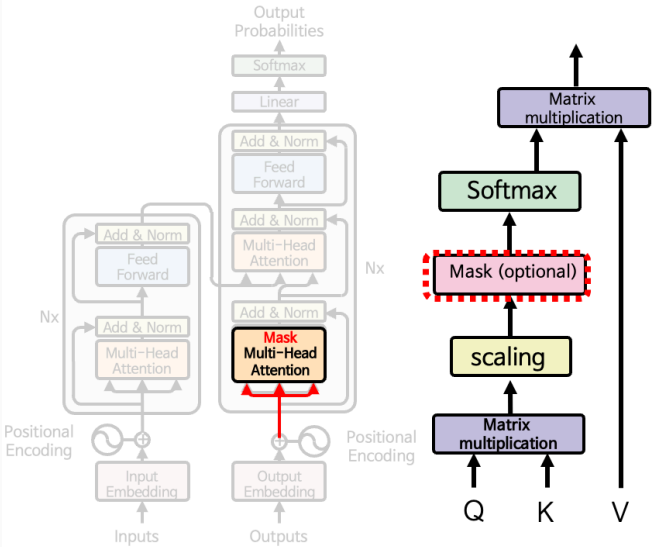


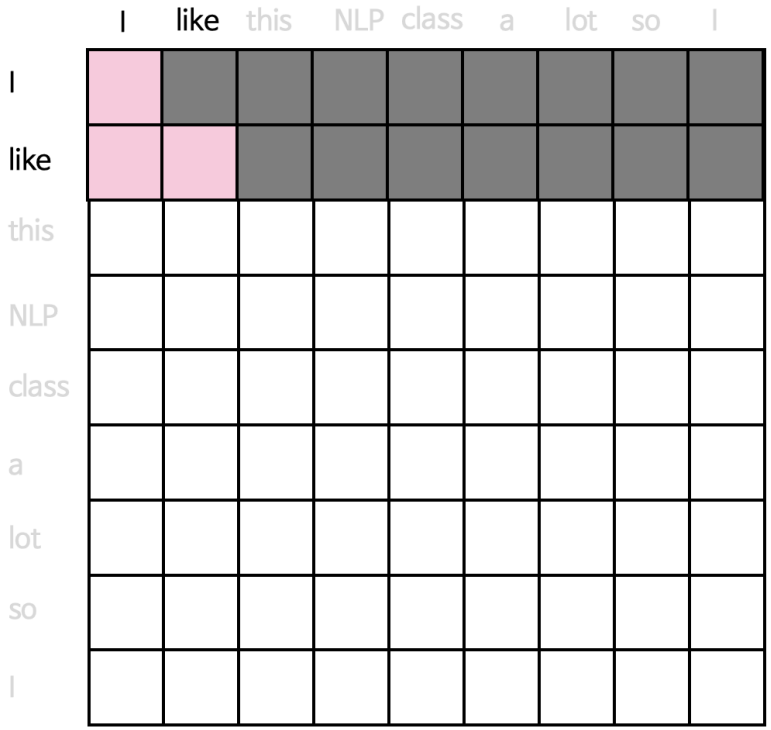
To reflect this characteristic, the decoder applies a masking mechanism during training.

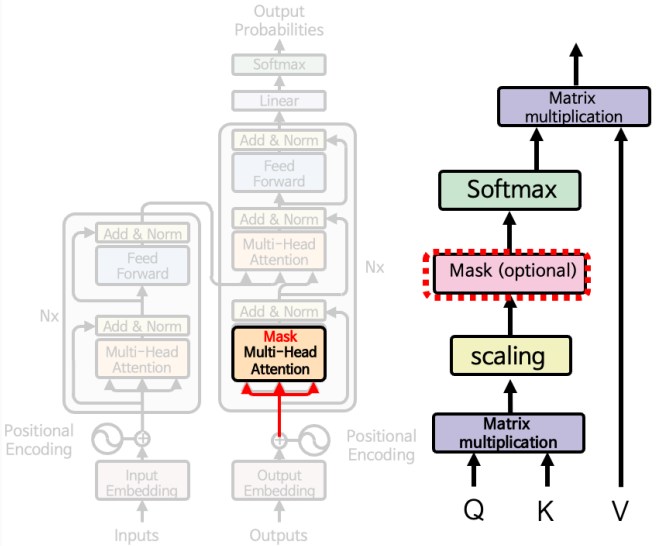


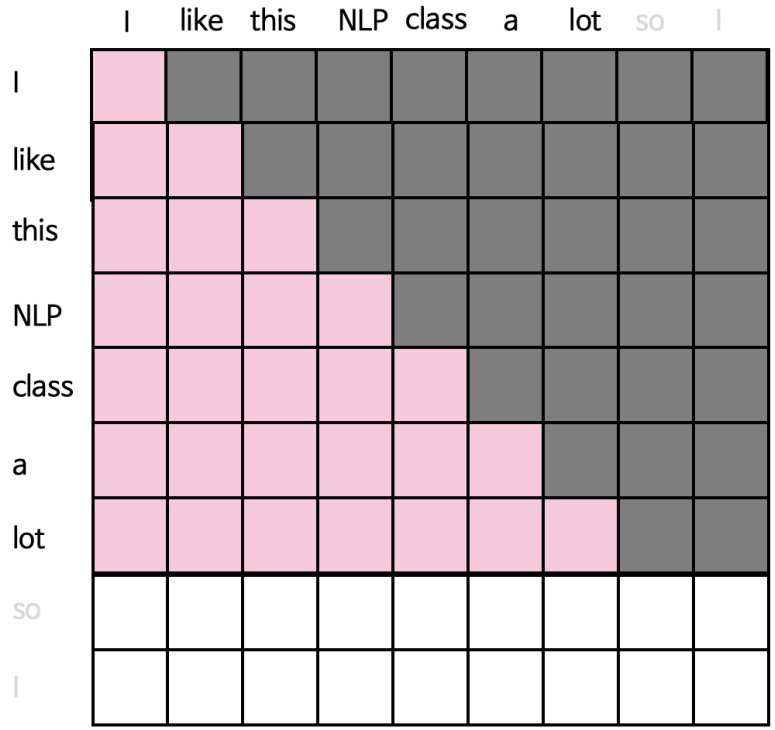
The key idea is to hide future tokens so they do not affect the current prediction.

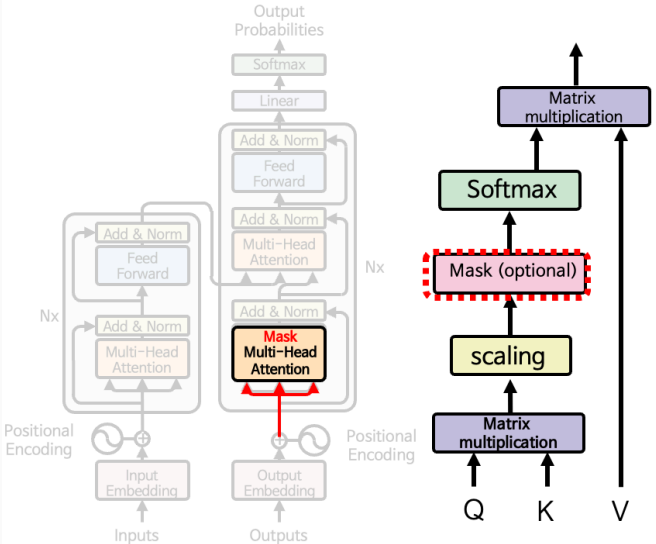


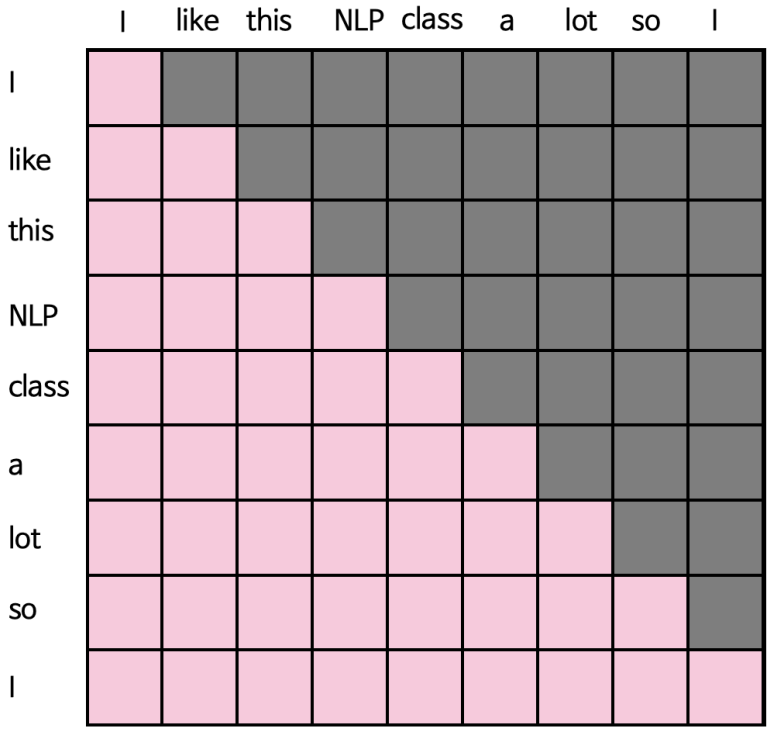

Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Mask Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs
Matrix multiplication
Softmax
Mask (optional)
scaling
Matrix multiplication
Q
K
V


I
like
this
NLP
class
a
lot
so
I
I
like
this
NLP
class
a
lot
so
I

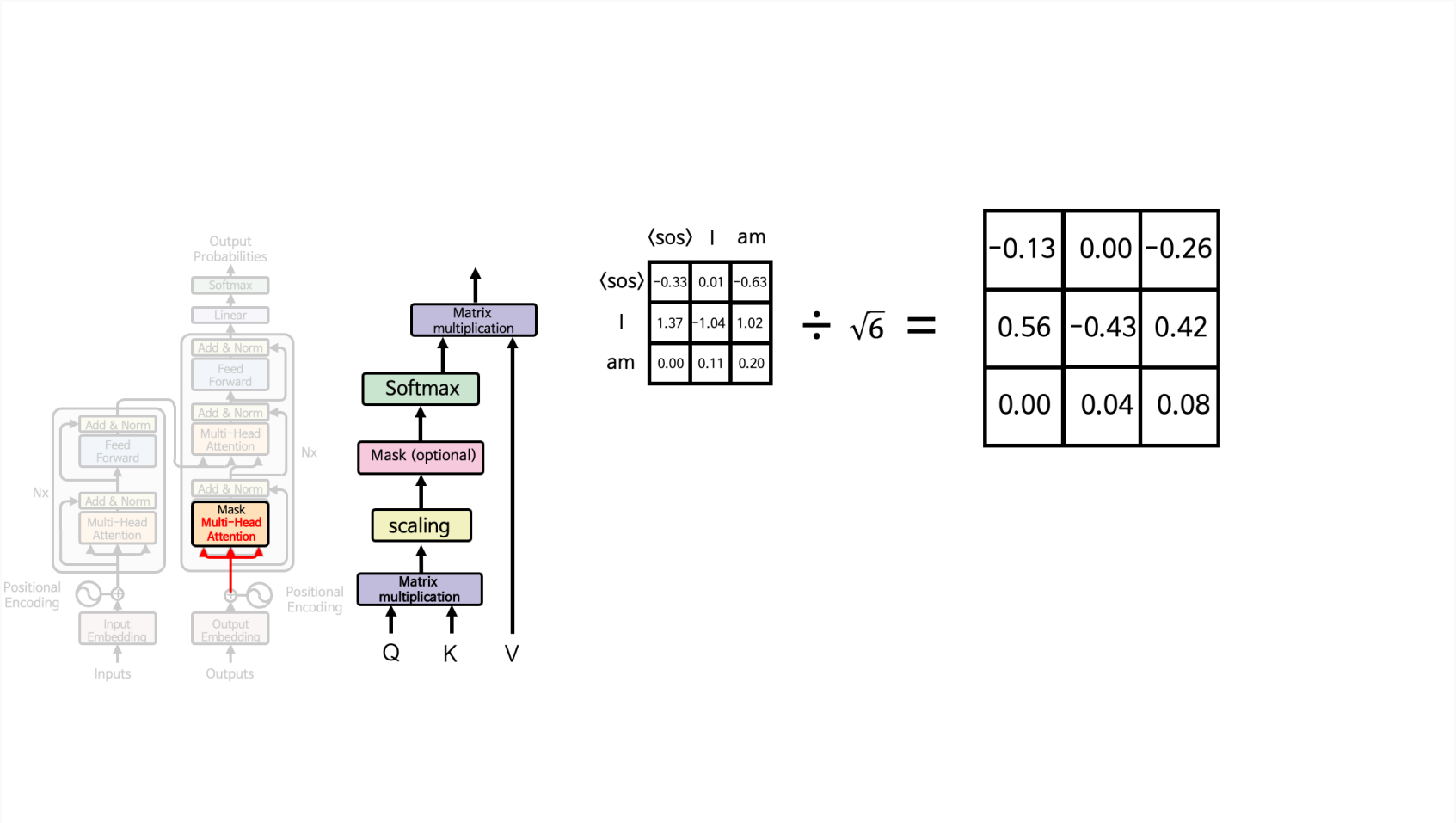

Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Mask Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs
Matrix multiplication
Softmax
Mask (optional)
scaling
Matrix multiplication
Q
K
V


I
like
this
NLP
class
a
lot
so
I
I
like
this
NLP
class
a
lot
so
I

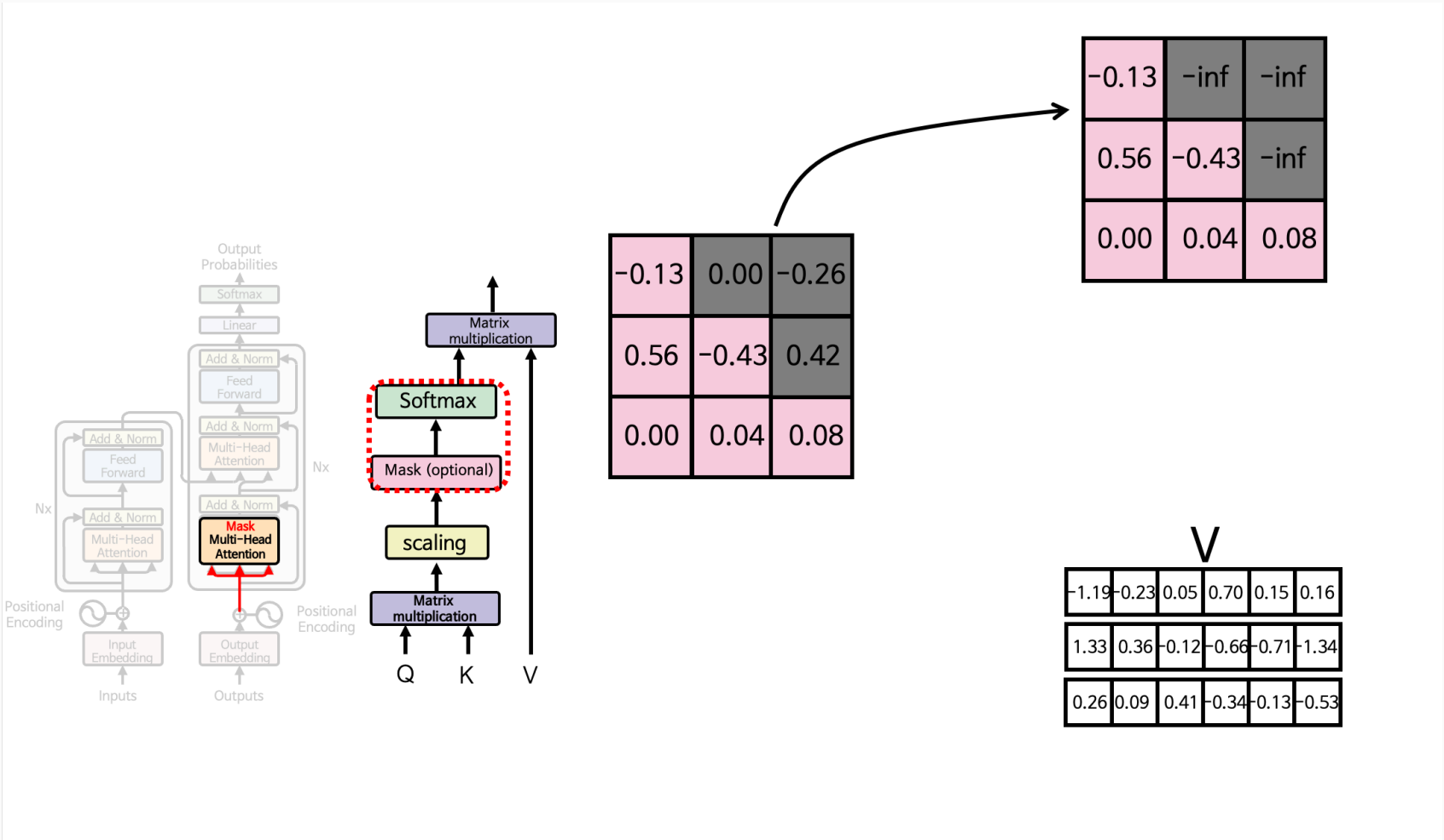

Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Mask Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs
Matrix multiplication
Softmax
Mask (optional)
scaling
Matrix multiplication
Q
K
V


I
like
this
NLP
class
a
lot
so
I

When this masking algorithm is applied to the attention matrix, we get:

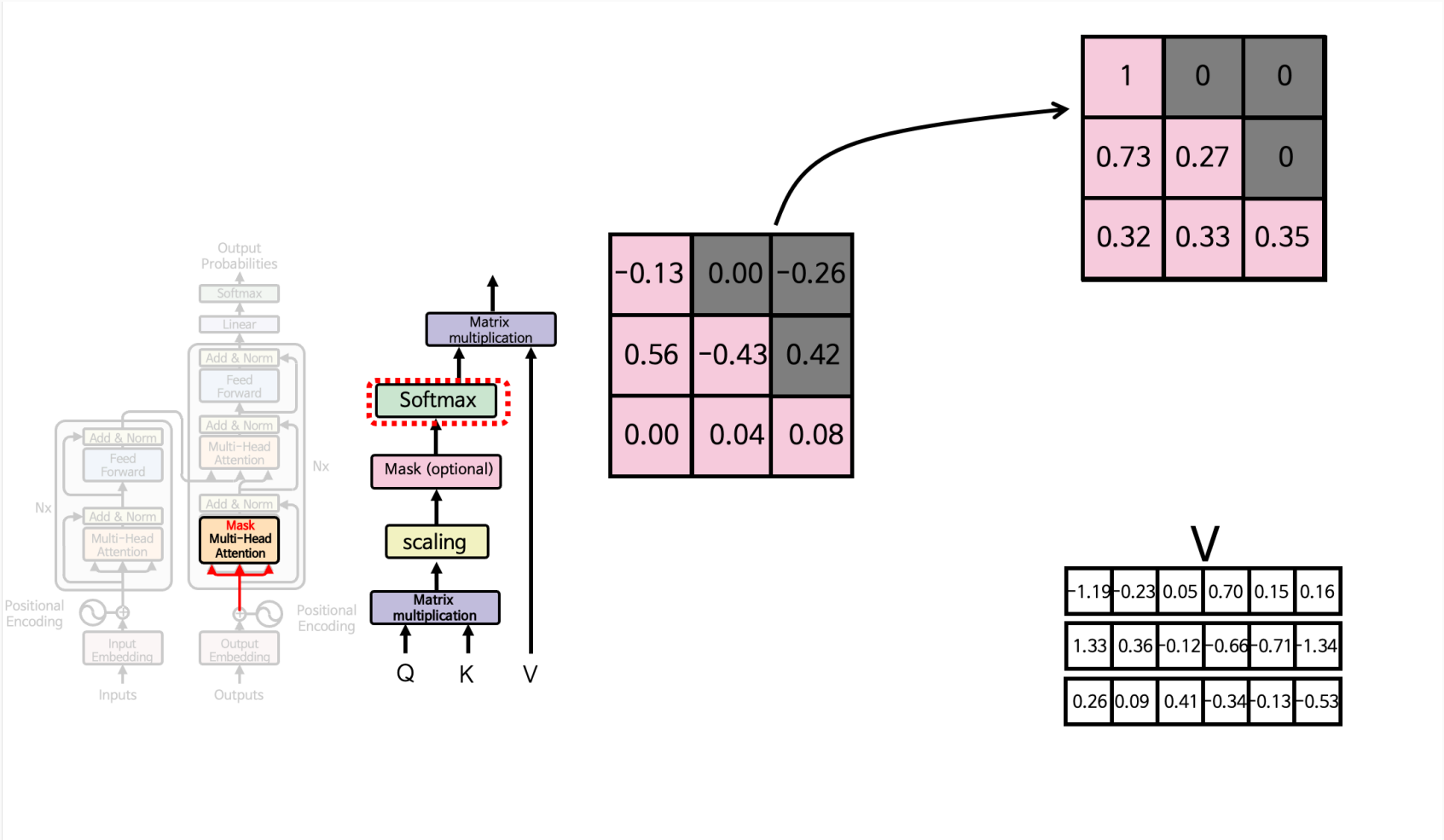


We add *–inf* to the masked positions because, after passing through the softmax layer, *–inf* becomes 0, effectively eliminating attention to those positions.

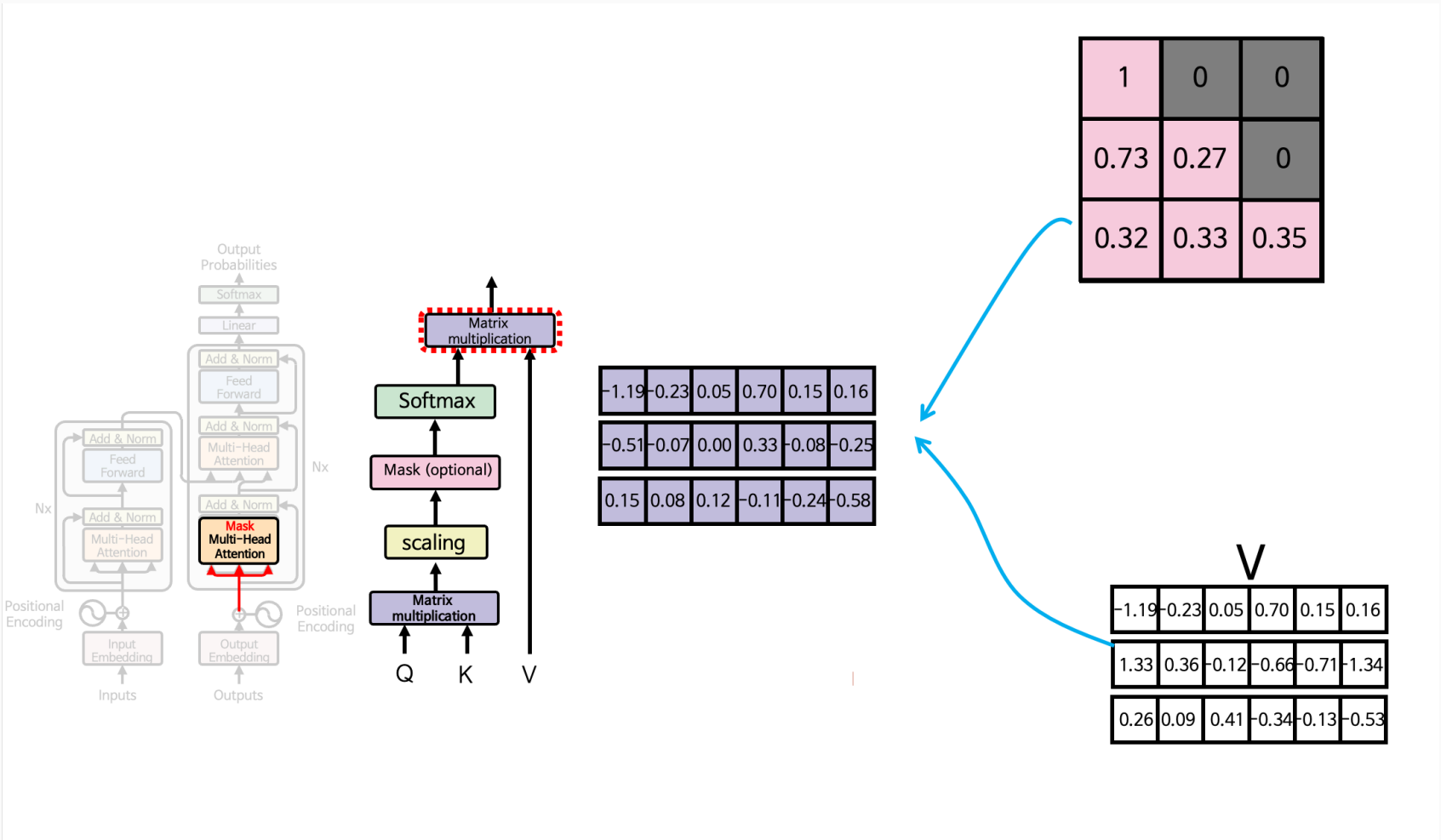


[1] https://www.youtube.com/shorts/SrJN_hpiuAs

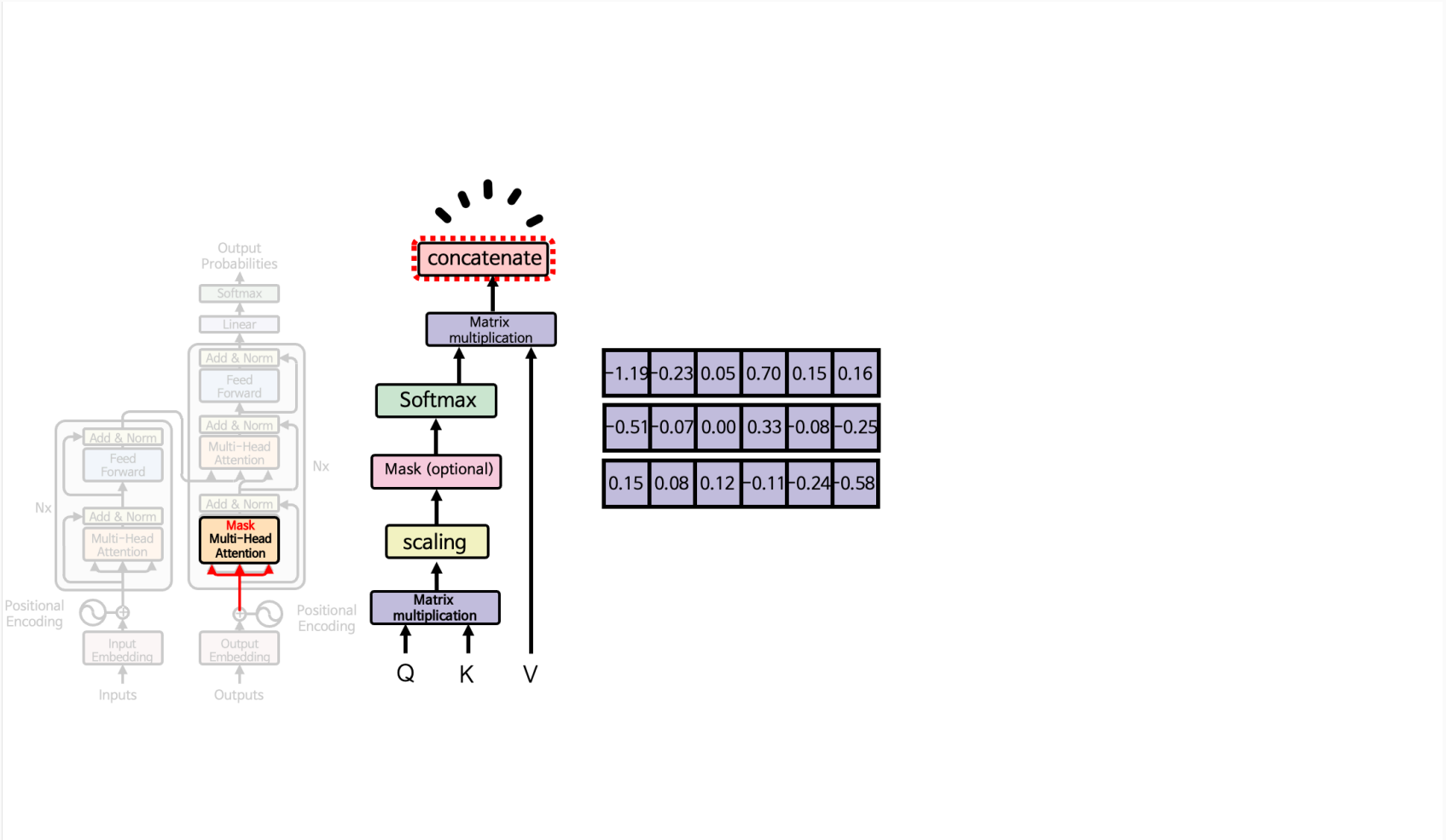
Feeding this matrix into the softmax layer gives us:



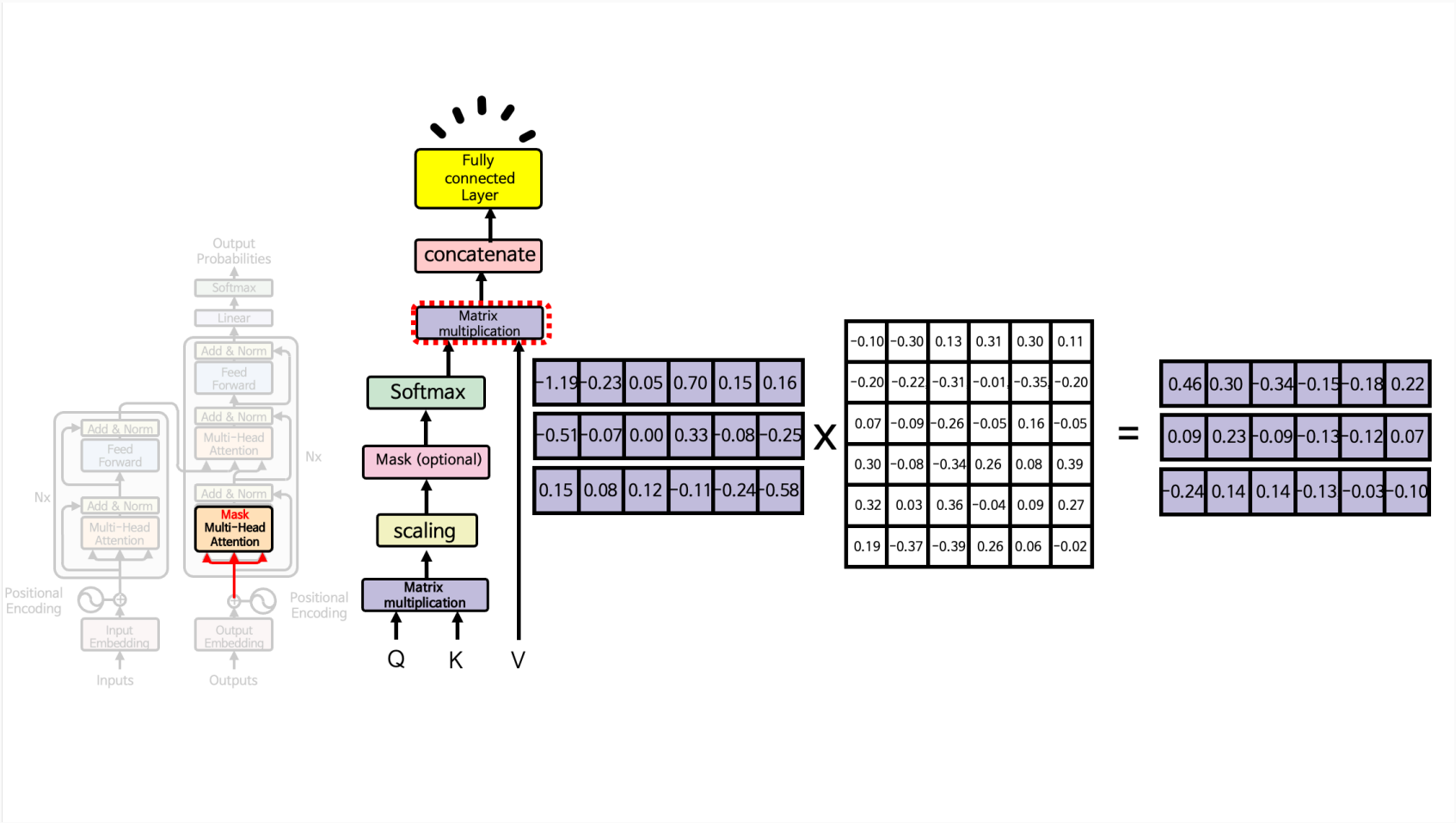
Then we multiply the two matrices as follows:



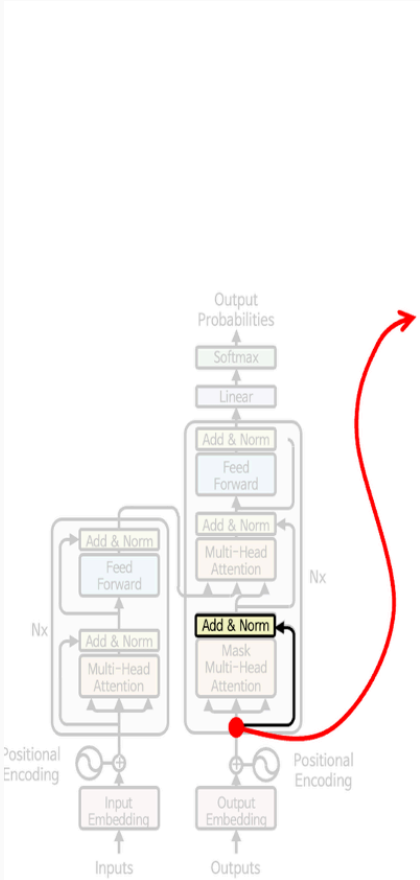
Next, we concatenate the resulting matrices (if needed):



We then multiply again to produce the final matrix of the masked multi-head attention.



We then apply the same Add & Normalize process again.

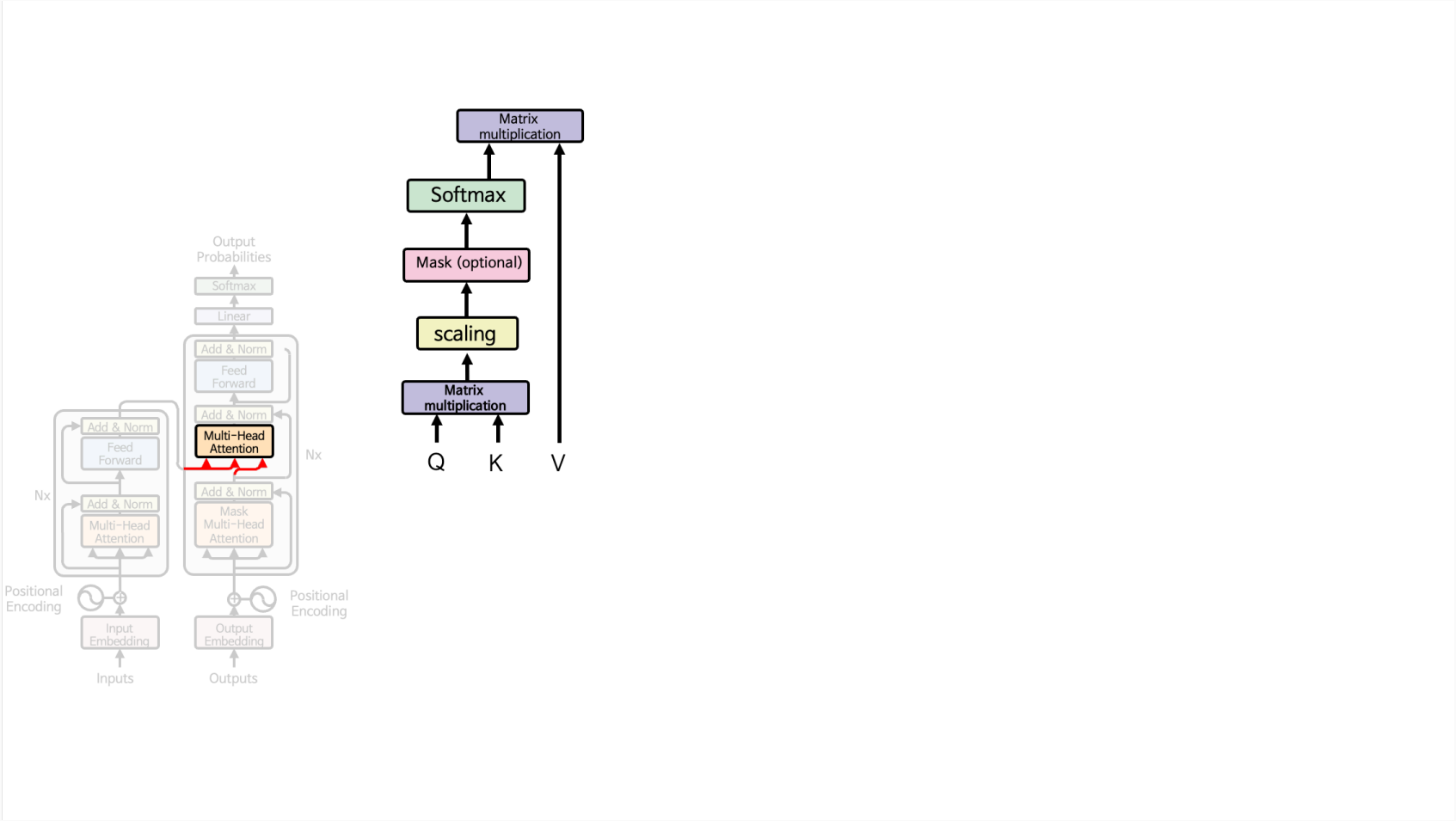


-2.31	-0.29	0.21	-0.24	1.86	1.06	+	0.46	0.30	-0.34	-0.15	-0.18	0.22
3.69	-0.20	0.24	-0.34	-0.57	0.67		0.09	0.23	-0.09	-0.13	-0.12	0.07
0.45	-0.48	-0.13	-0.25	-0.48	0.66		-0.24	0.14	0.14	-0.13	-0.03	-0.10

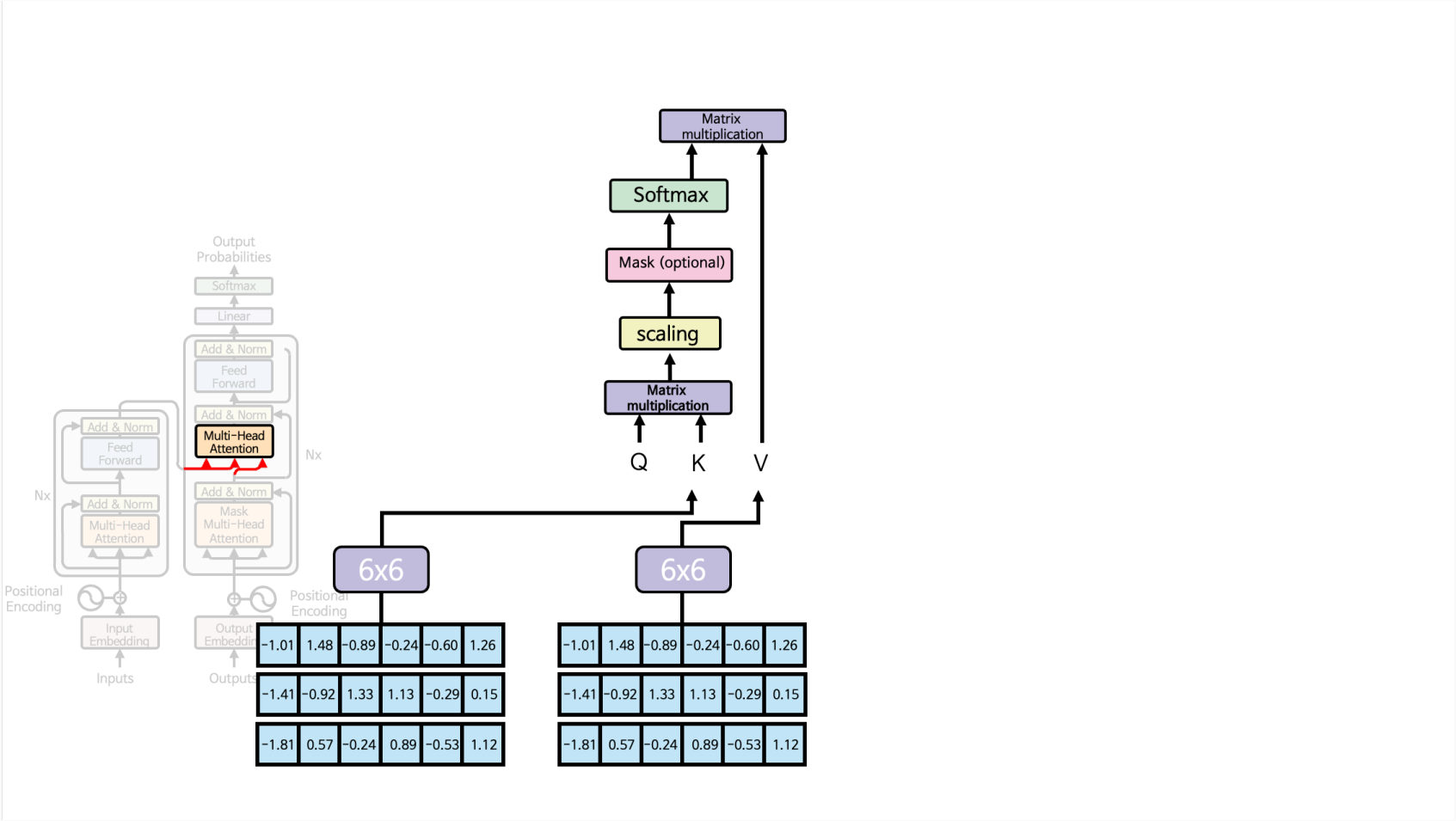
=

$$Norm\left(\begin{array}{cccccc} -1.85 & 0.01 & -0.12 & -0.39 & 1.67 & 1.28 \\ 3.79 & 0.03 & 0.15 & -0.47 & -0.69 & 0.74 \\ 0.21 & -0.33 & 0.01 & -0.38 & -0.51 & 0.56 \end{array}\right) = \begin{array}{cccccc} -1.69 & -0.08 & -0.20 & -0.42 & 1.37 & 1.02 \\ 2.13 & -0.37 & -0.29 & -0.71 & -0.85 & 0.10 \\ 0.76 & -0.70 & 0.23 & -0.82 & -1.16 & 1.69 \end{array}$$

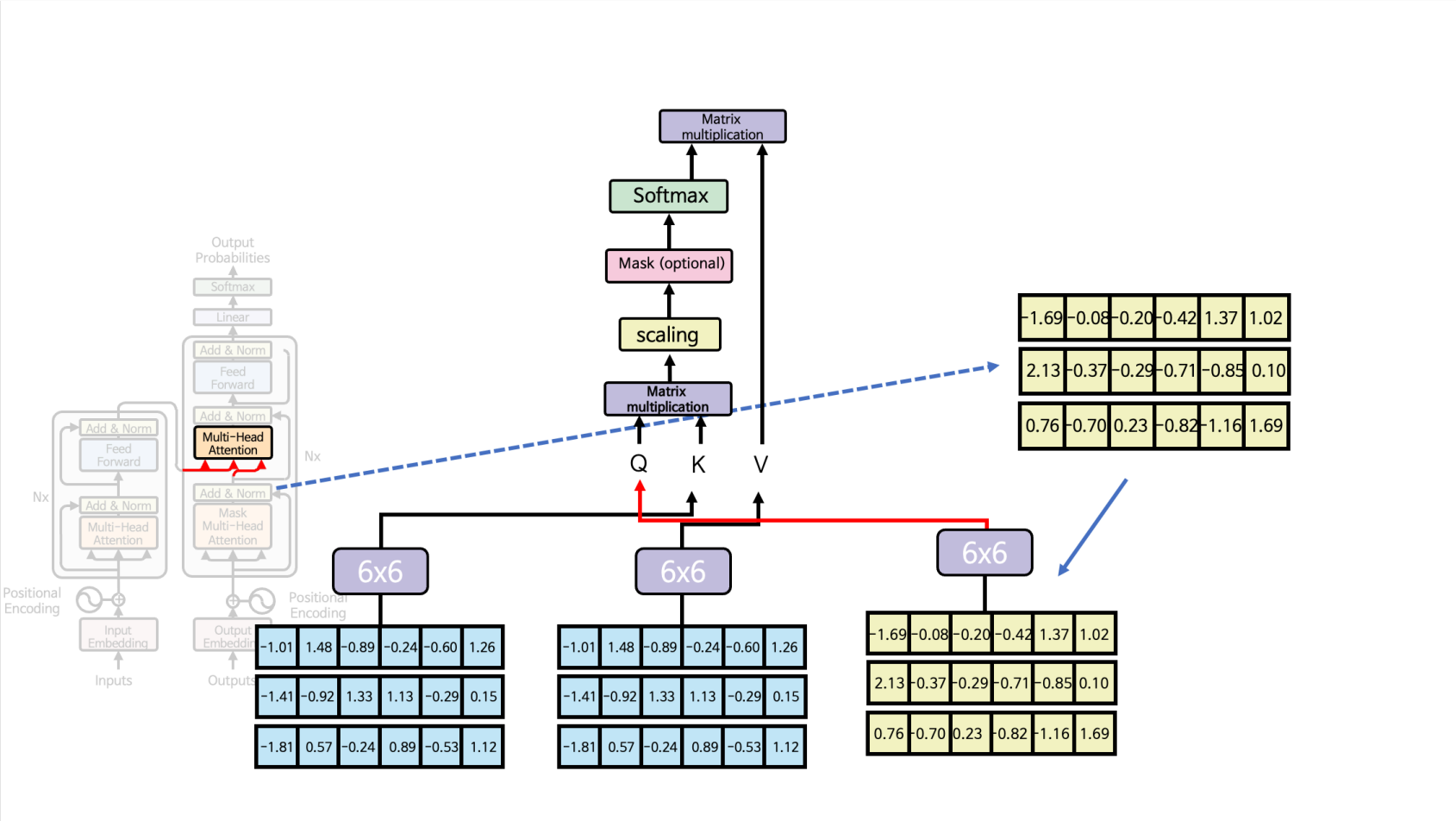
The decoder’s second multi-head attention operates the same way as the encoder’s, except for the inputs.



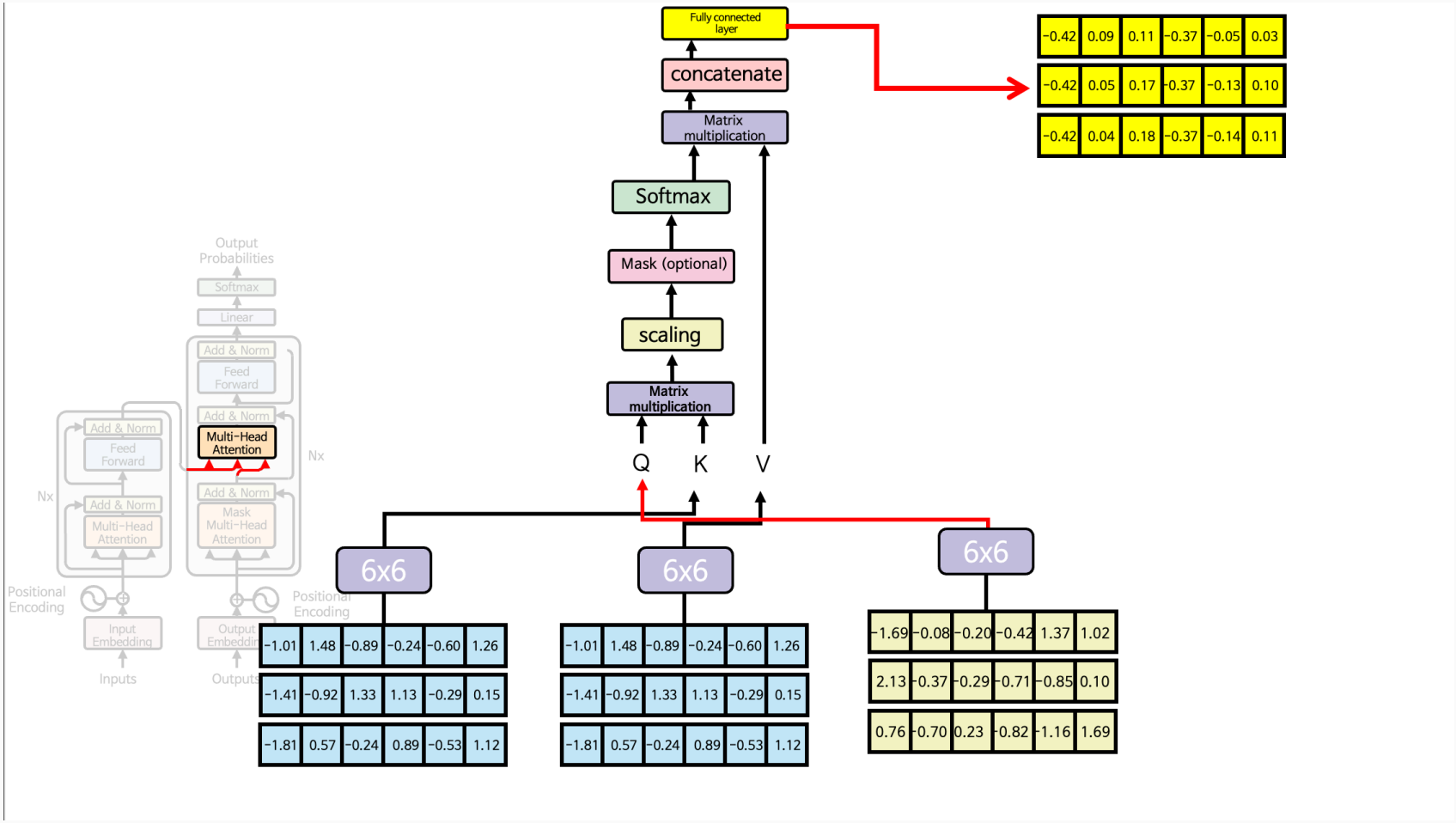
Here, the values of K and V are derived from the **encoder's final output**, multiplied by a 6×6 matrix.



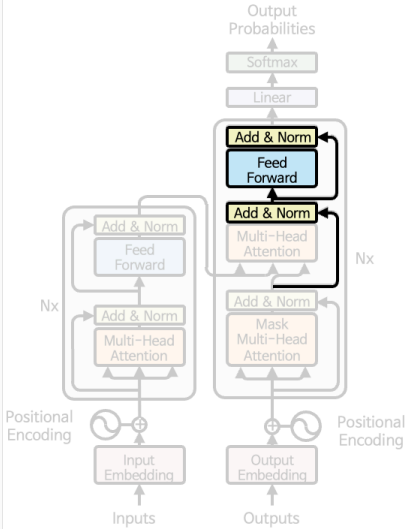
The value of Q comes from the output of the decoder's previous Add & Norm layer. **In other words, the decoder determines which parts of the encoder's output (K, V) to attend to, based on the context it has generated so far (Q)**-similar to attention in RNNs.



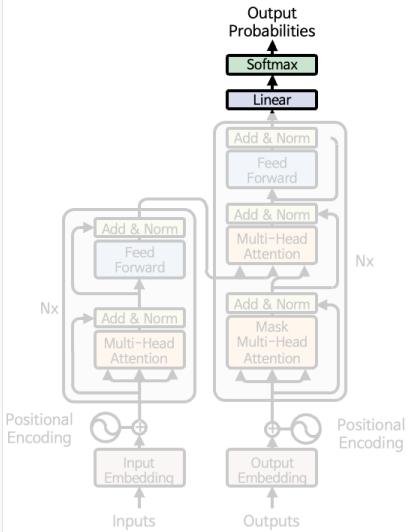
Assume the final output of this multi-head attention looks as follows:



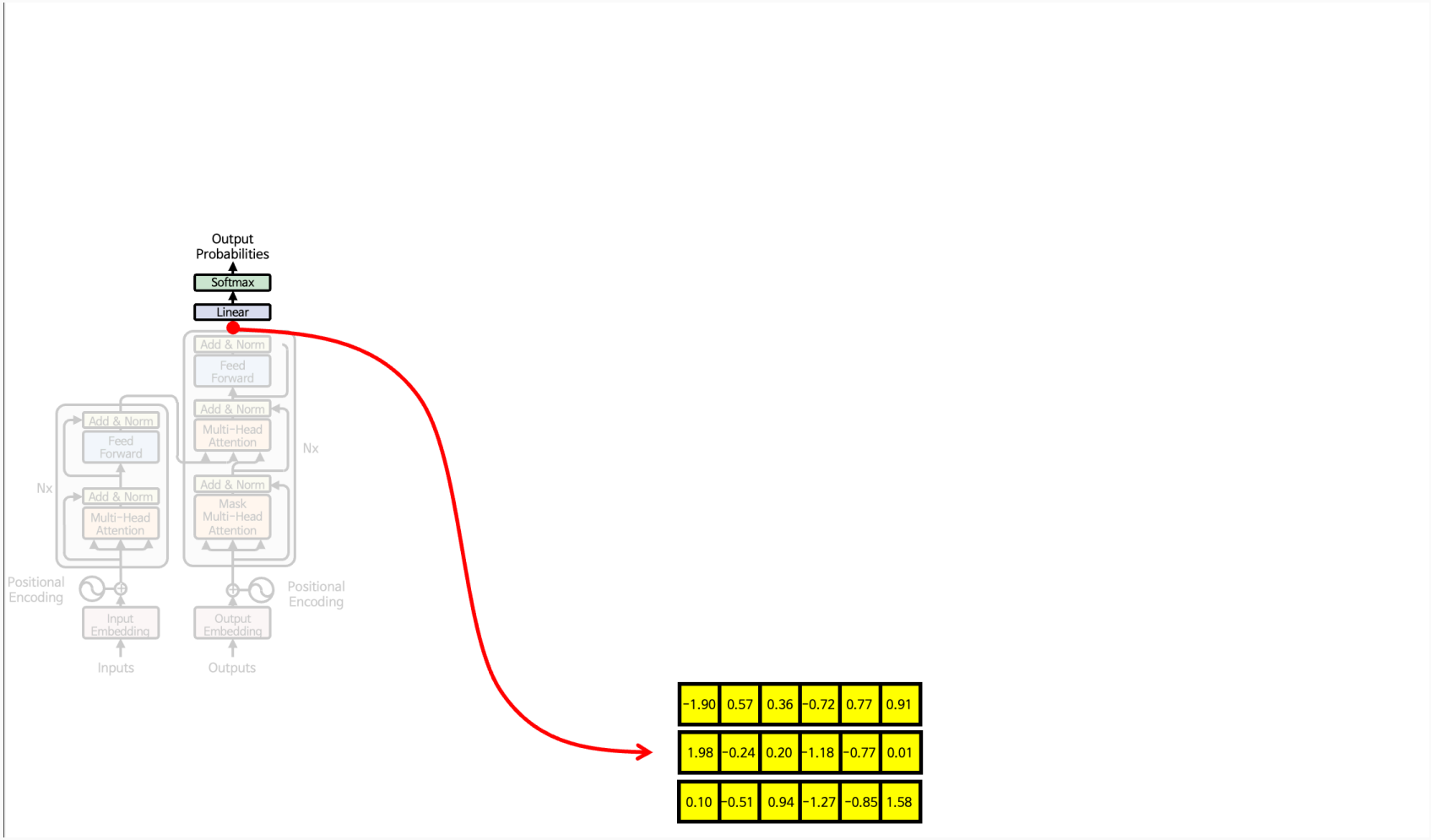
The same computational process is repeated, so details are omitted here.



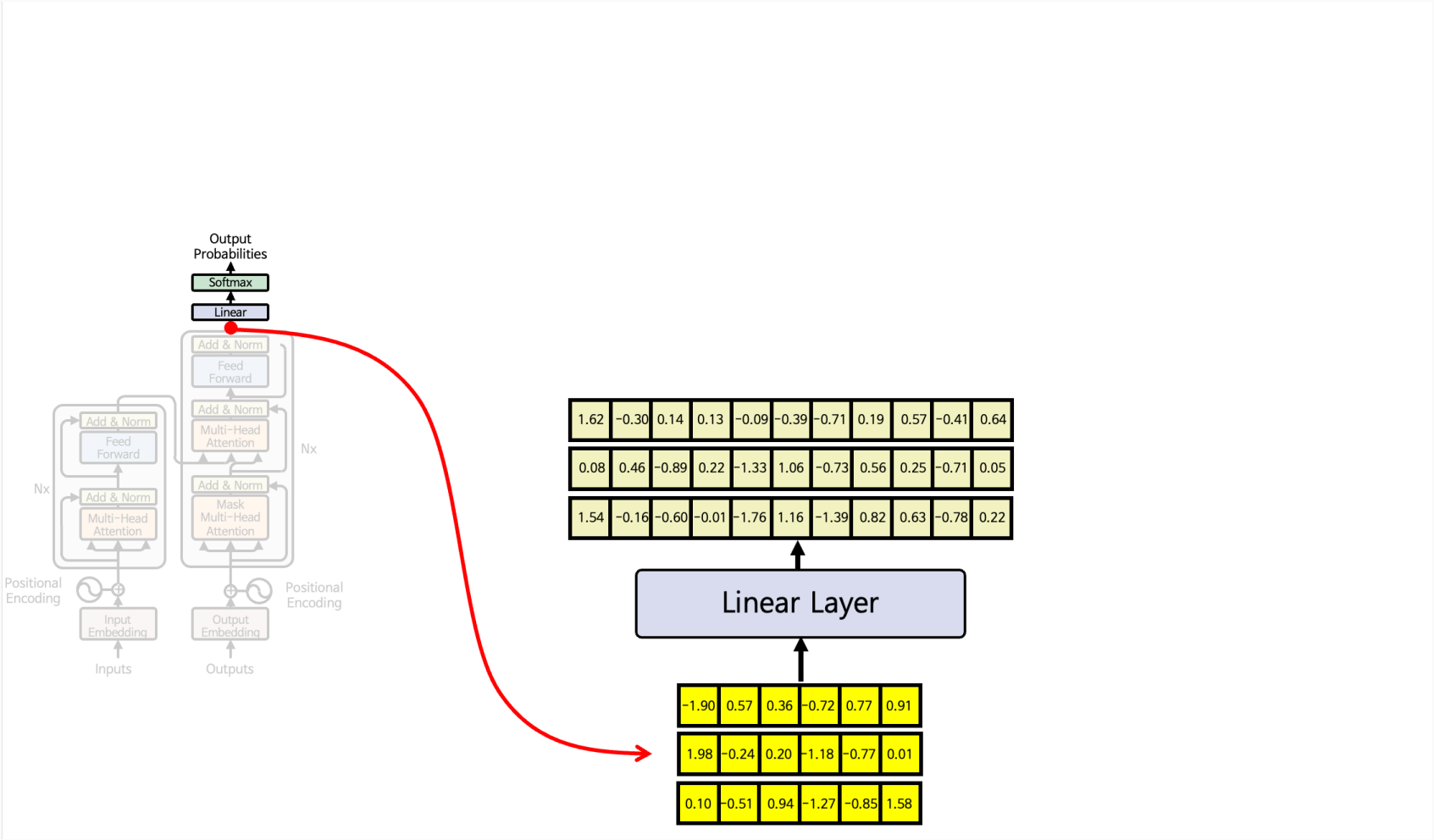
Finally, the last linear and softmax layers produce the final output probabilities.



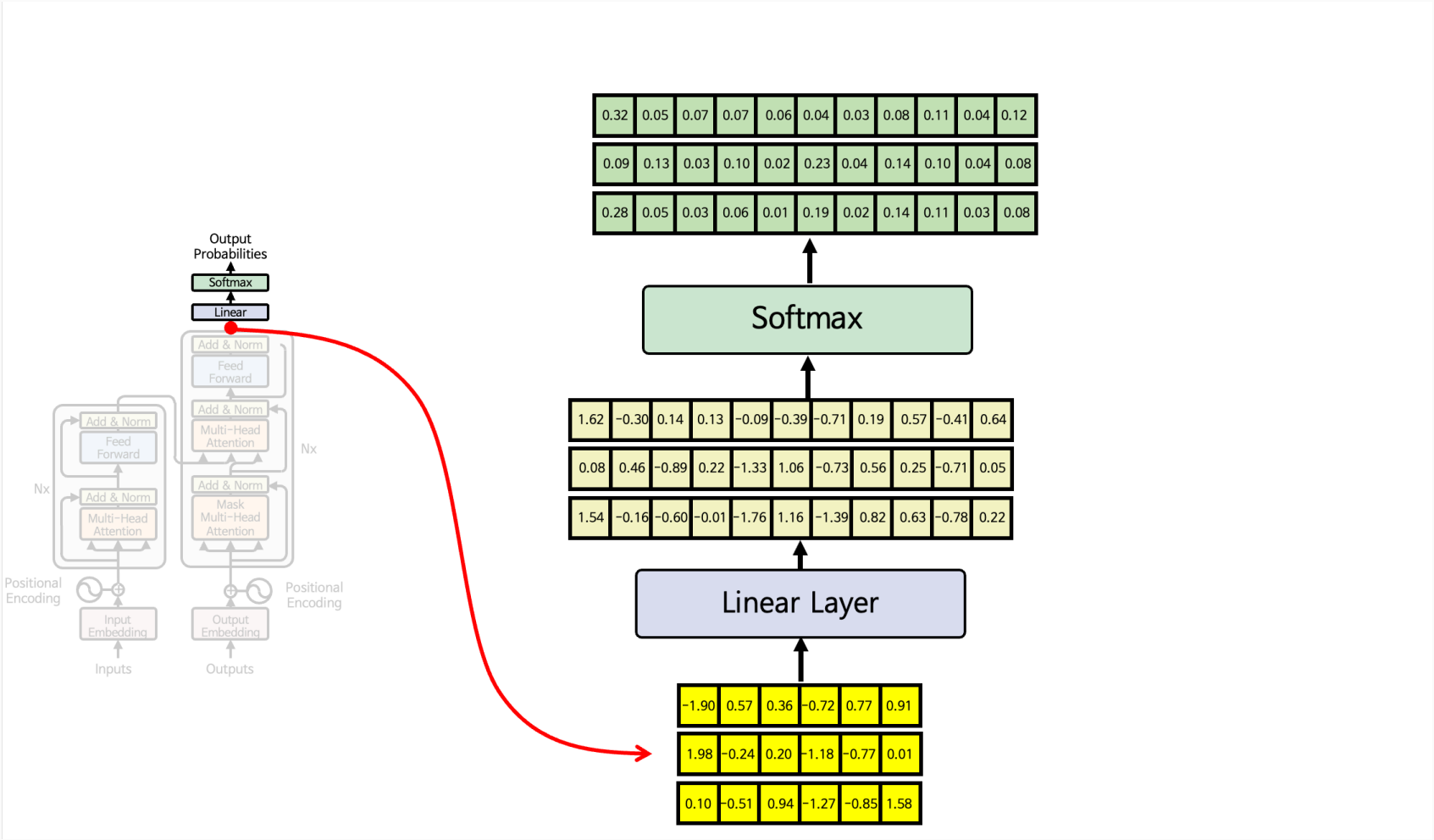
If the decoder output matrix at this stage looks like this:



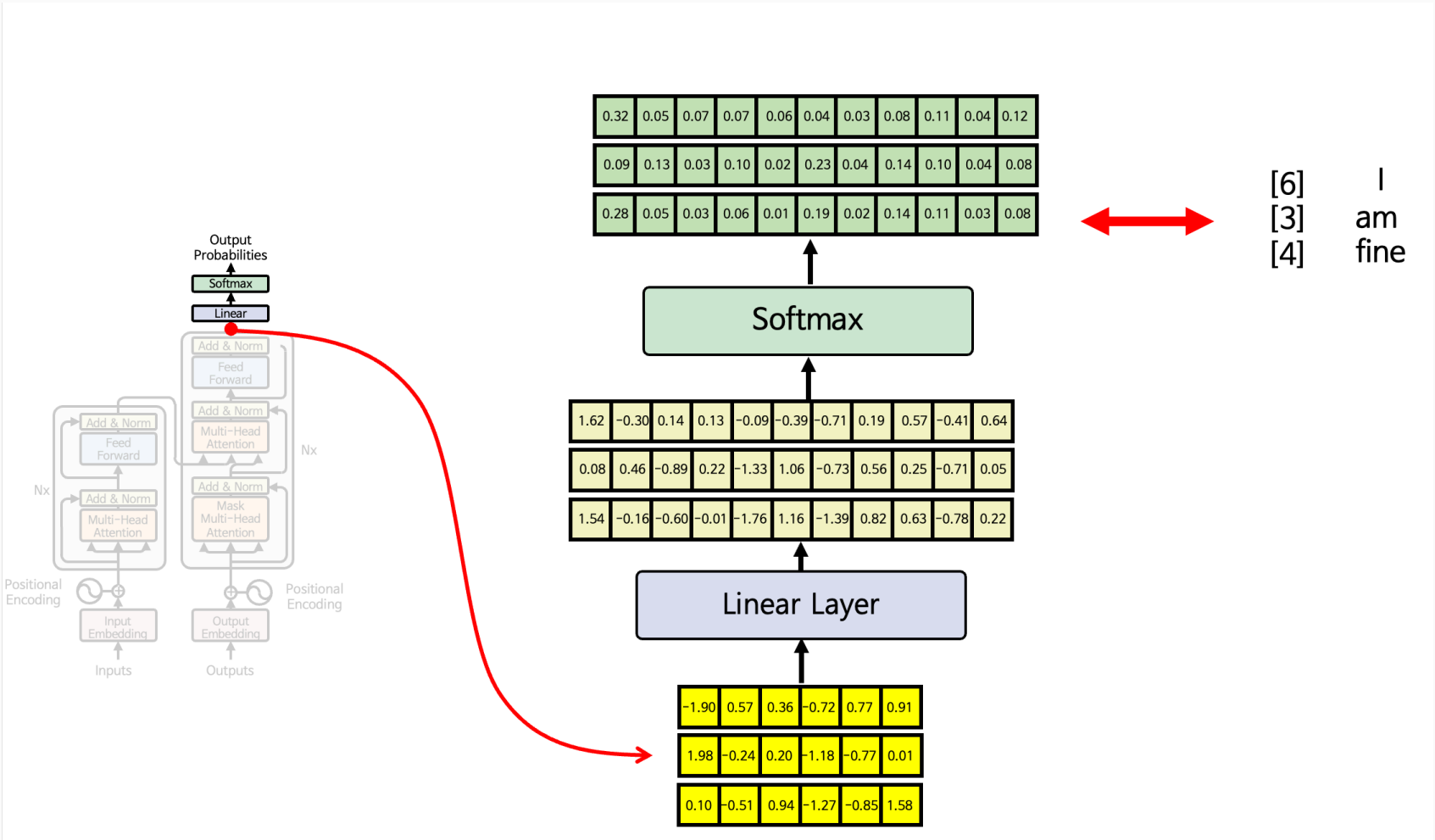
The linear layer projects it back to the full vocabulary size (=11).



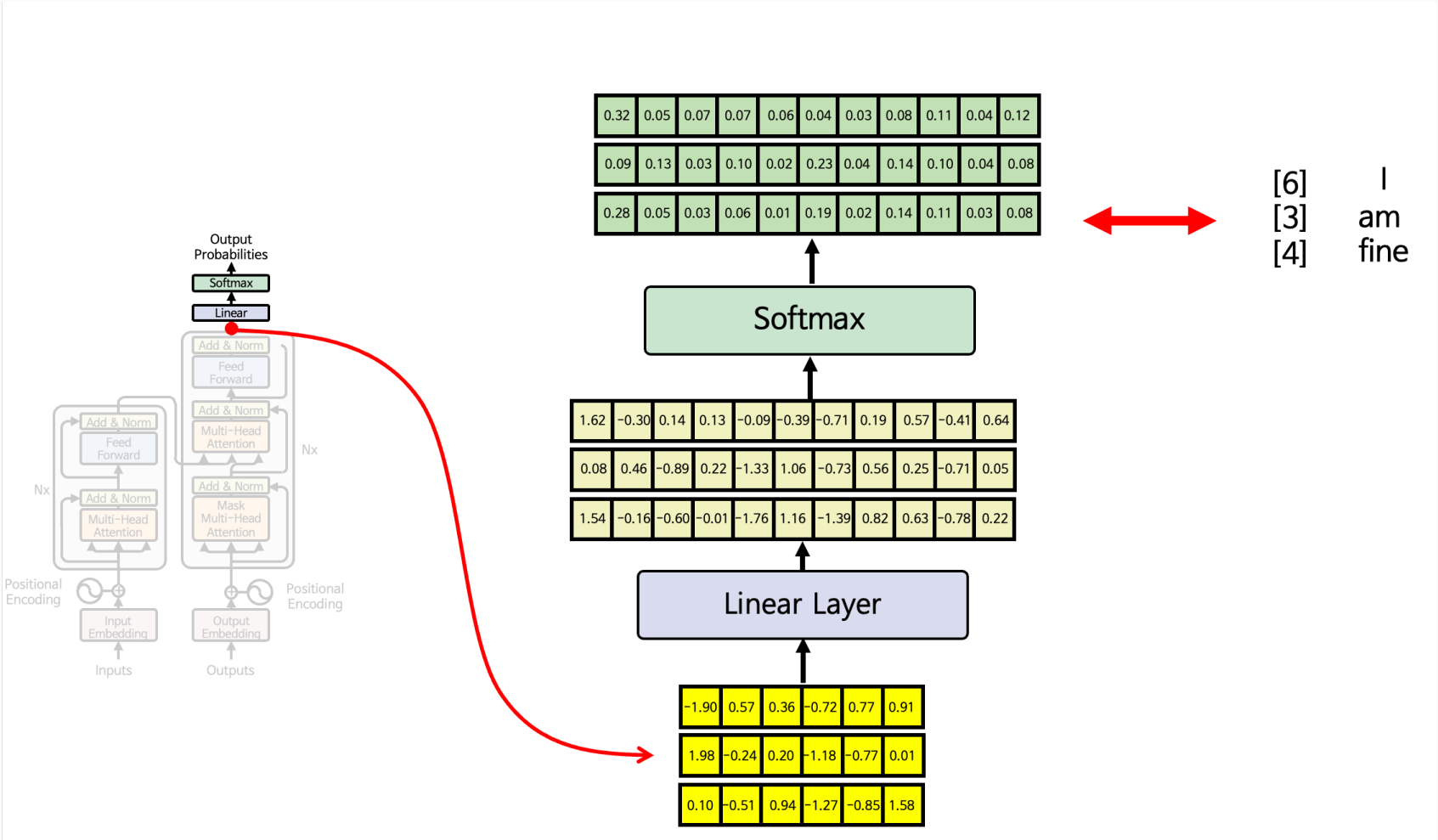
The softmax function then produces the final output probabilities.

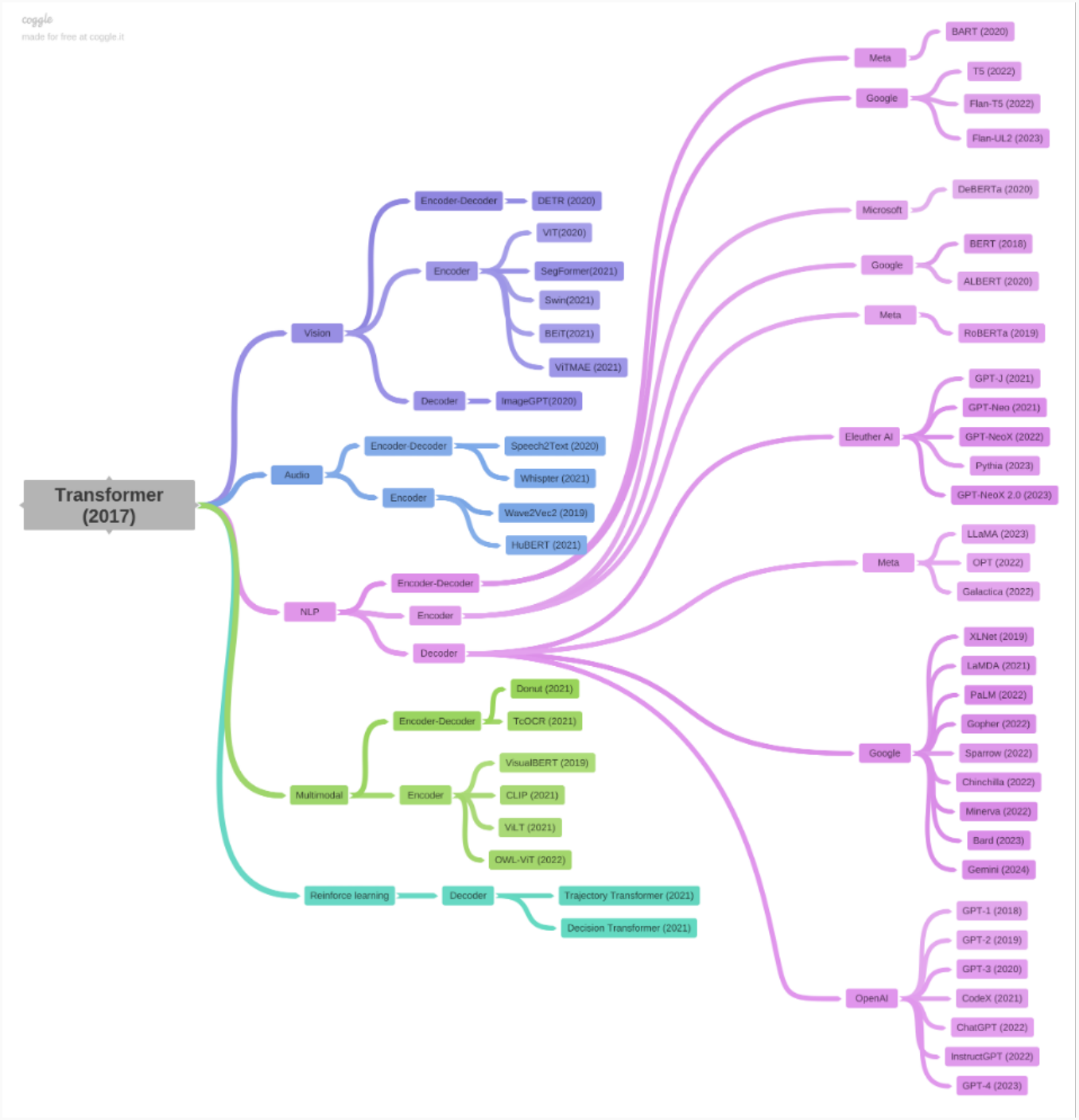


Finally, the predicted outputs are compared with the ground-truth labels (e.g., 6, 3, 4).



Using a **loss function** (e.g., cross-entropy) and **backpropagation**, the model updates all weight parameters across every layer — this is the learning process of the Transformer.




coggle
made for free at coggle.it
Transformer (2017)
Vision
Encoder-Decoder
DETR (2020)
Encoder
ViT(2020)
SegFormer(2021)
Swin(2021)
BEiT(2021)
ViTMAE (2021)
Decoder
ImageGPT(2020)
Audio
Encoder-Decoder
Speech2Text (2020)
Whispter (2021)
Encoder
Wave2Vec2 (2019)
HuBERT (2021)
NLP
Encoder-Decoder
Encoder
Decoder
Multimodal
Encoder-Decoder
Donut (2021)
TrOCR (2021)
Encoder
VisualBERT (2019)
CLIP (2021)
ViLT (2021)
OWL-ViT (2022)
Reinforce learning
Decoder
Trajectory Transformer (2021)
Decision Transformer (2021)
Meta
BART (2020)
Google
T5 (2022)
Flan-T5 (2022)
Flan-UL2 (2023)
Microsoft
DeBERTa (2020)
Google
BERT (2018)
ALBERT (2020)
Meta
RoBERTa (2019)
Eleuther AI
GPT-J (2021)
GPT-Neo (2021)
GPT-NeoX (2022)
Pythia (2023)
GPT-NeoX 2.0 (2023)
Meta
LLaMA (2023)
OPT (2022)
Galactica (2022)
Google
XLNet (2019)
LaMDA (2021)
PaLM (2022)
Gopher (2022)
Sparrow (2022)
Chinchilla (2022)
Minerva (2022)
Bard (2023)
Gemini (2024)
OpenAI
GPT-1 (2018)
GPT-2 (2019)
GPT-3 (2020)
CodeX (2021)
ChatGPT (2022)
InstructGPT (2022)
GPT-4 (2023)

Wrap up

Wrap-up

- We explored the architecture of the Transformer model.

Wrap-up

- We explored the architecture of the Transformer model.
- Do we need to train this model entirely from scratch?

Wrap-up

- We explored the architecture of the Transformer model.
- Do we need to train this model entirely from scratch?
- **No**. We can take advantage of powerful *pre-trained language models* that have already learned general language patterns from massive datasets.

Wrap-up

- We explored the architecture of the Transformer model.
- Do we need to train this model entirely from scratch?
- **No**. We can take advantage of powerful *pre-trained language models* that have already learned general language patterns from massive datasets.
- We will experiment with these models using *Hugging Face* (open-source library) in the lab session.

Attention in transformers

Would be a nice recap:
`https://www.youtube.com/watch?v=eMlx5fFNoYc` Any hints from the video clip?