Transformer Neural Network, Chat GPT foundation, Clearly Explained !

(By Josh Starmer)

Transformer for fast : translation

Word embedding : Using a relatively simple neural network that has one input for every word and symbol in the vocabulary that you want to use

Ex: the sentence to translate to Spanish "Let's go"

→ 1,87 and 0,09 represent the word "Let's"

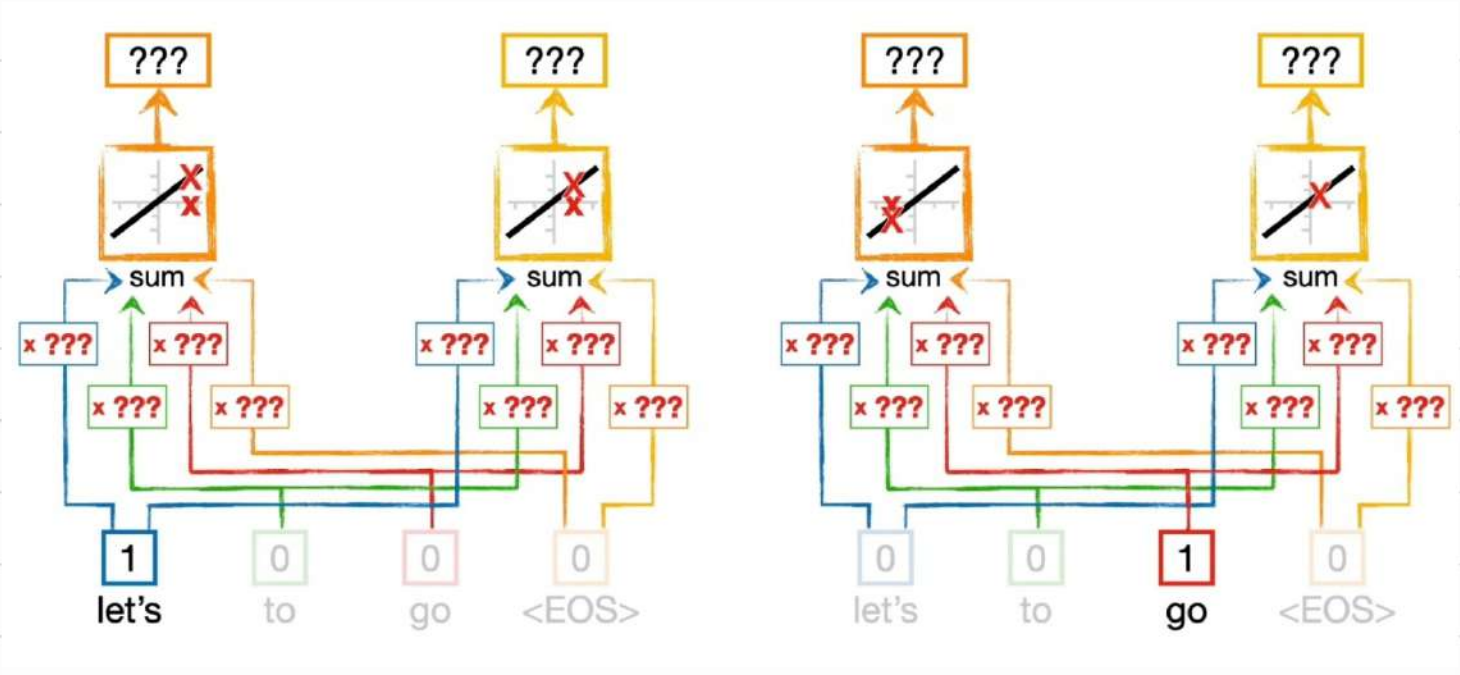
→ -0,78 and 0,27 " " " "go"

Important : We reuse the same word embedding network for

each input word or symbol. In other words, the weights in the network for "Let's" are the exact same as the weights in the network for "go".

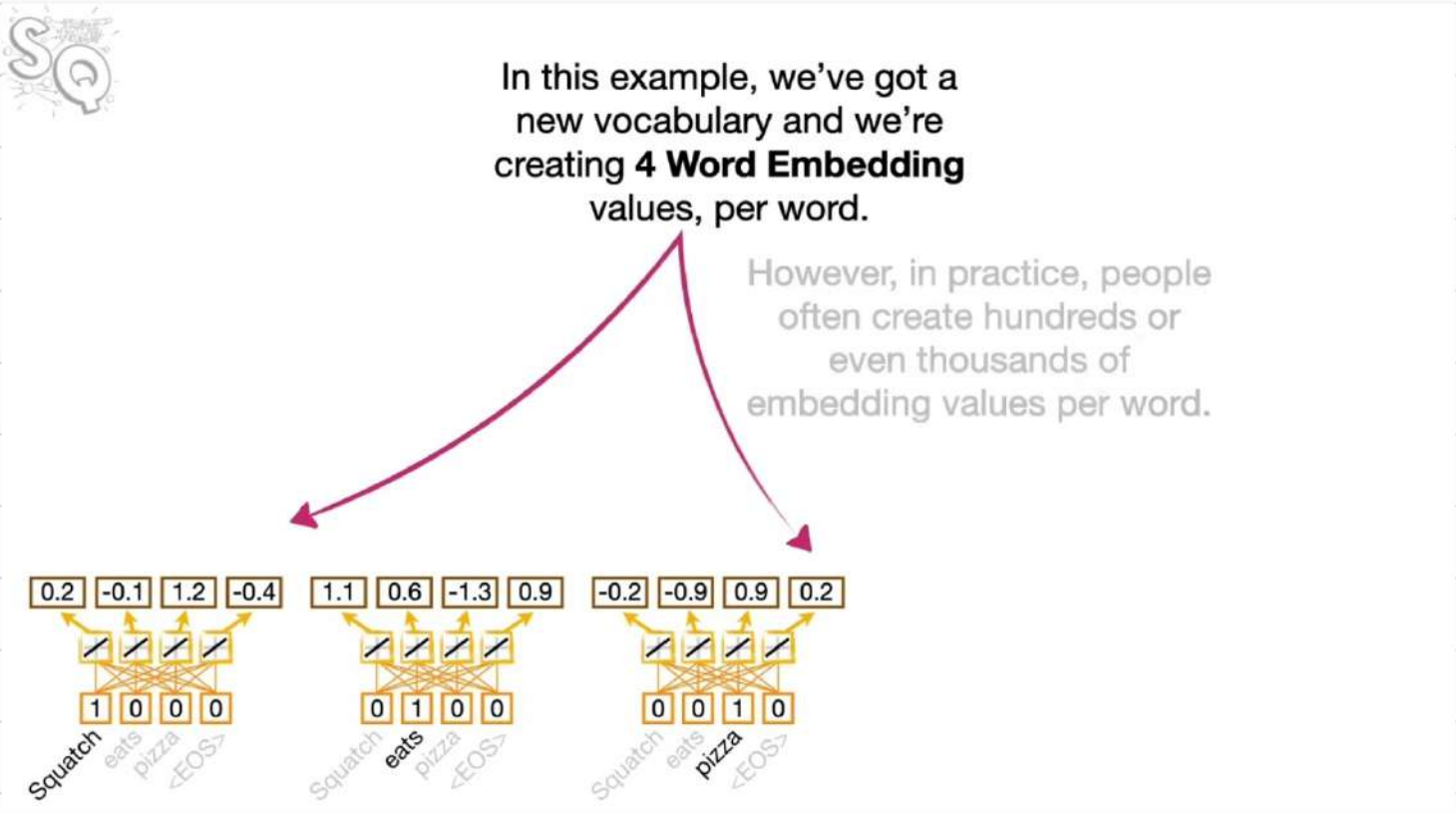
- Weight are determined using **Backpropagation**

Illustration: Determining the weights values through Backpropagation when Training the transformer network with English phrases and known Spanish translations

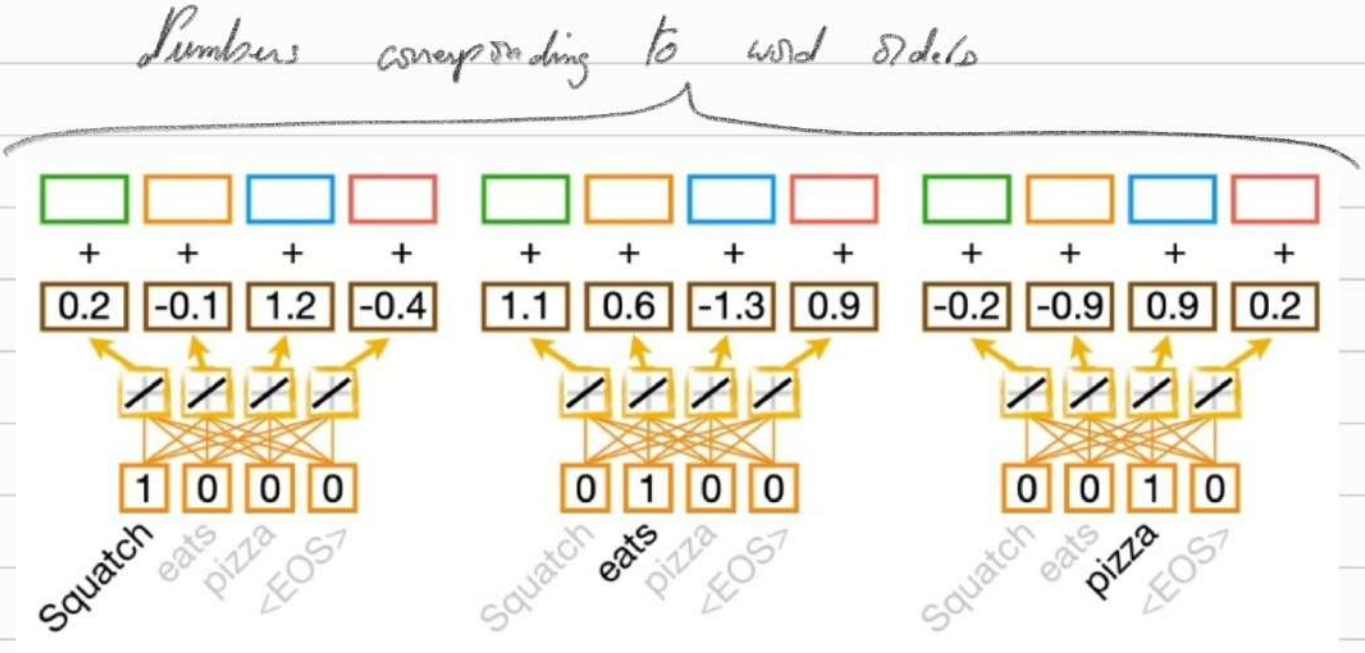


In the example above, we have 2 word embedding values per word.

Word Order: Order is important, so we must do positional encoding w/ one popular method



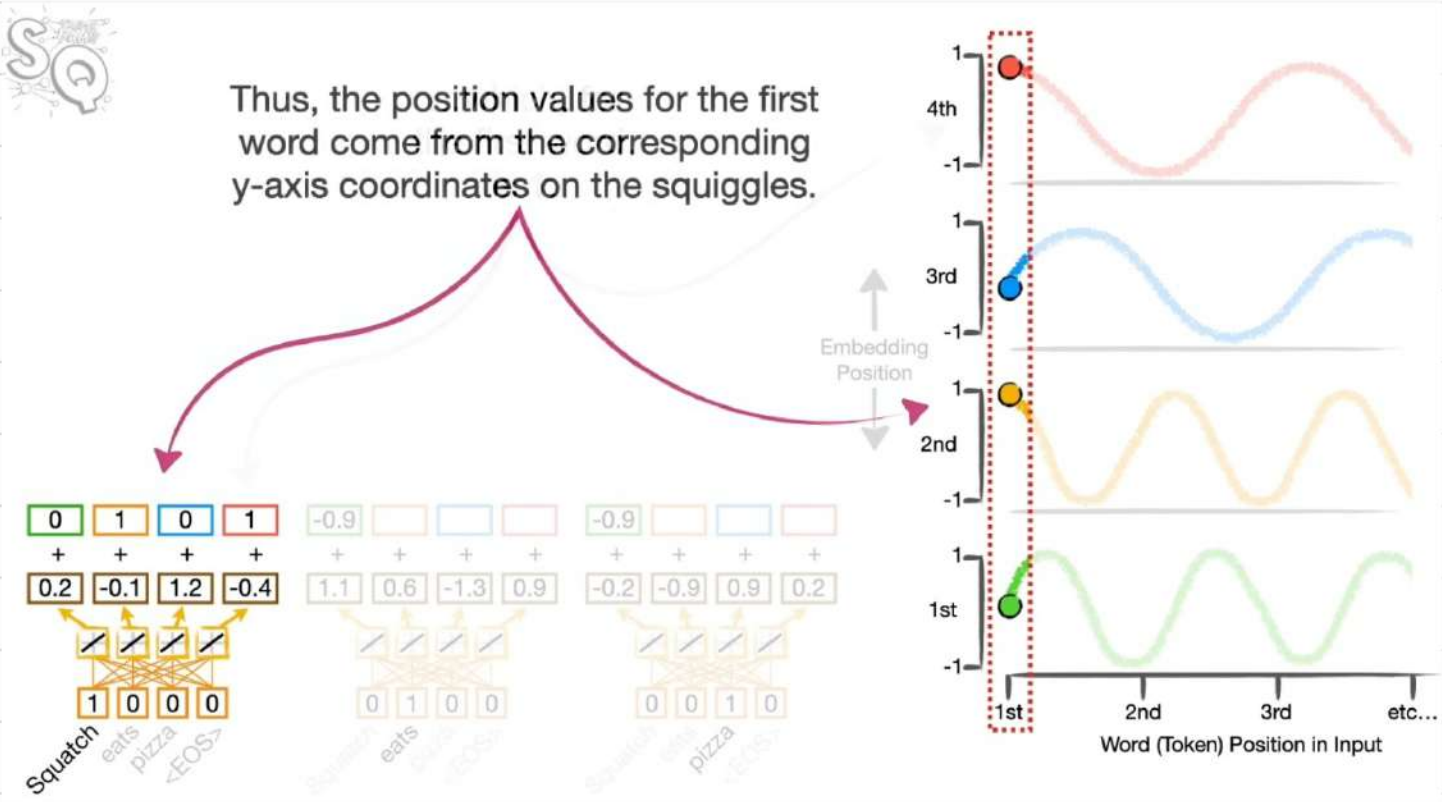
The we add a set of numbers that correspond to word order to the embedding values for each word



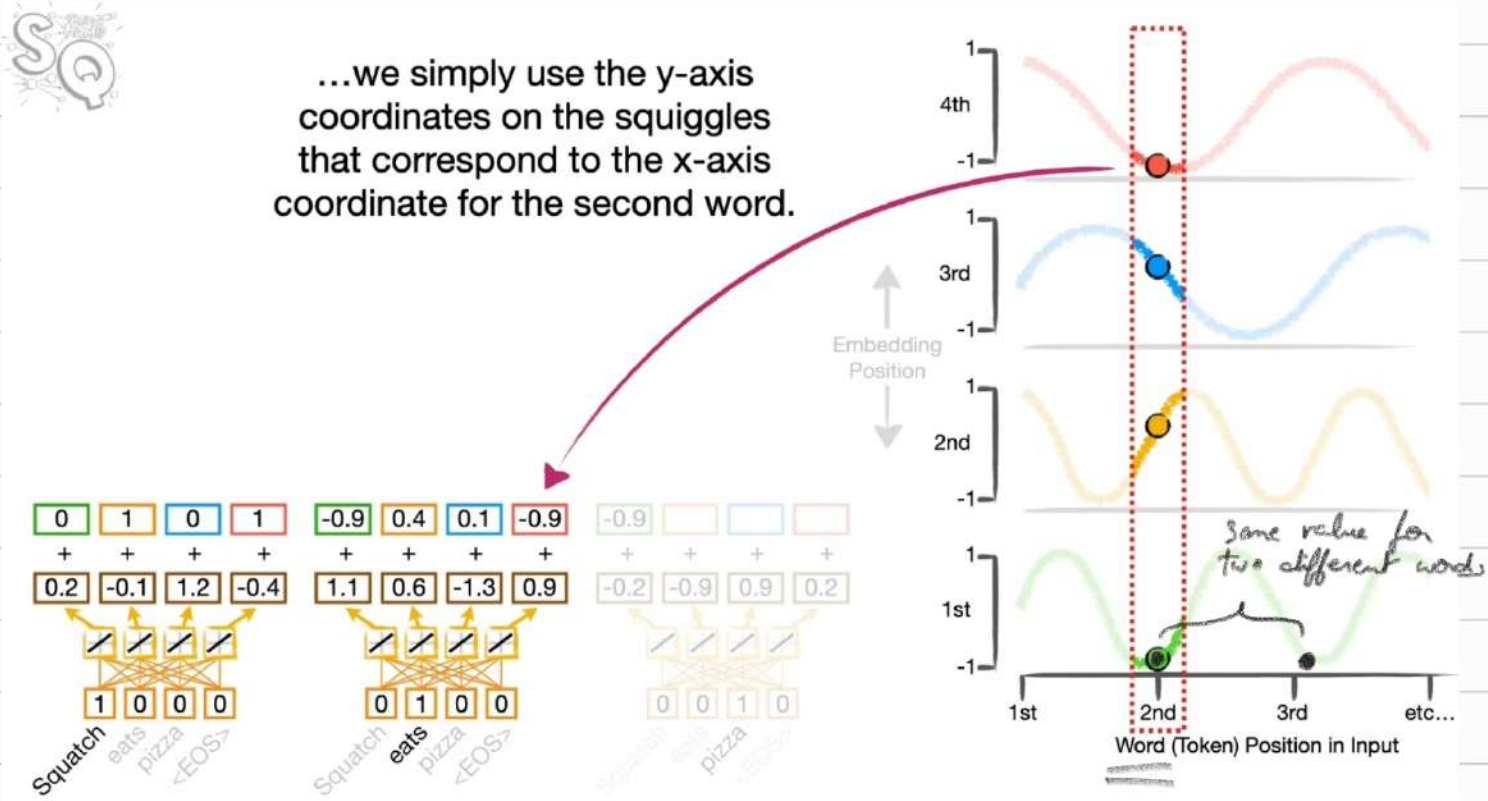
The numbers that represent the word order come from a sequence of alternating Sine an Cosine squiggles.

Illustration:

- Position values for the 1st word:

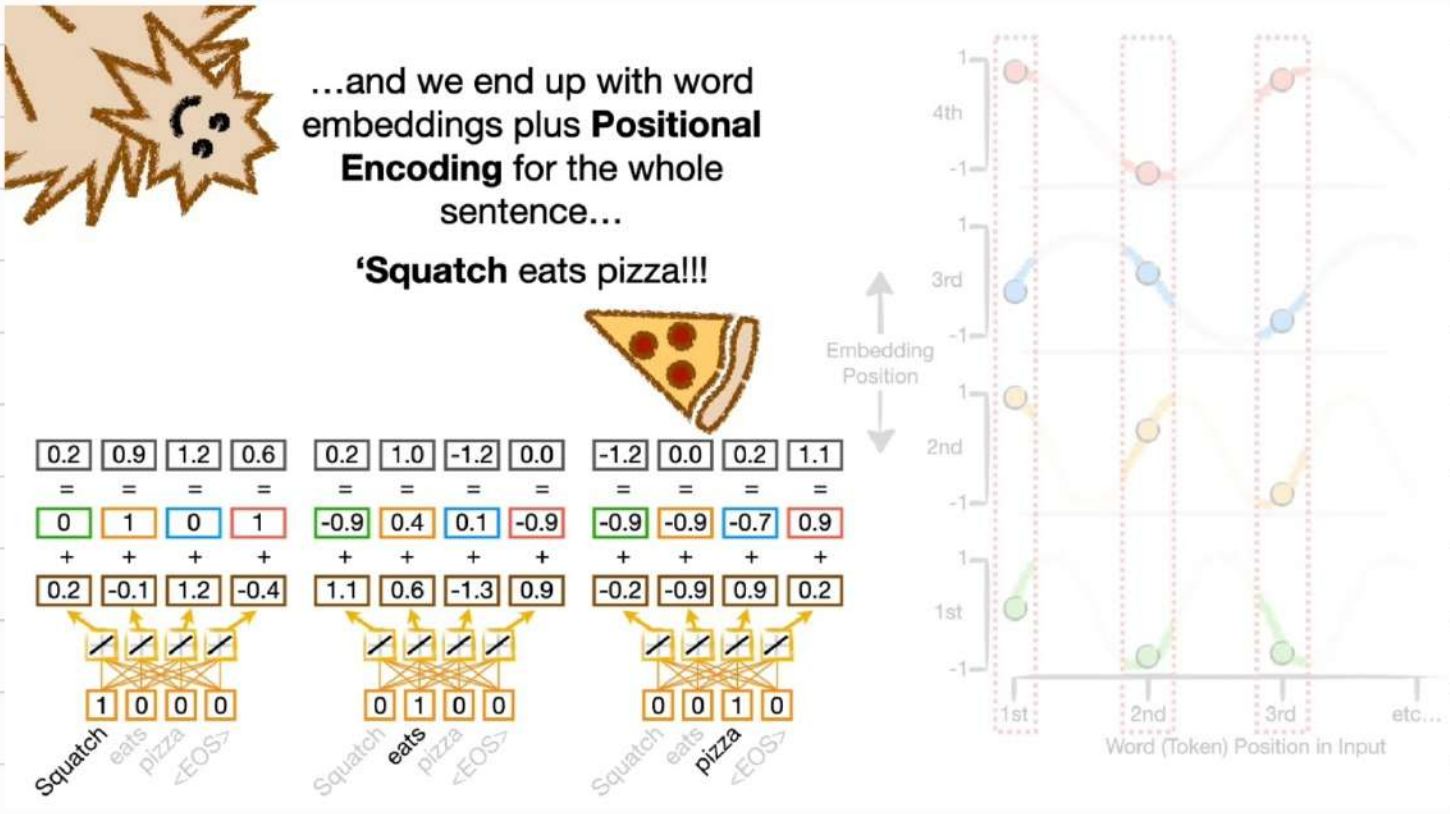


- Position values for the 2nd word:



... and so on for the 3rd and the last word of the sequence.

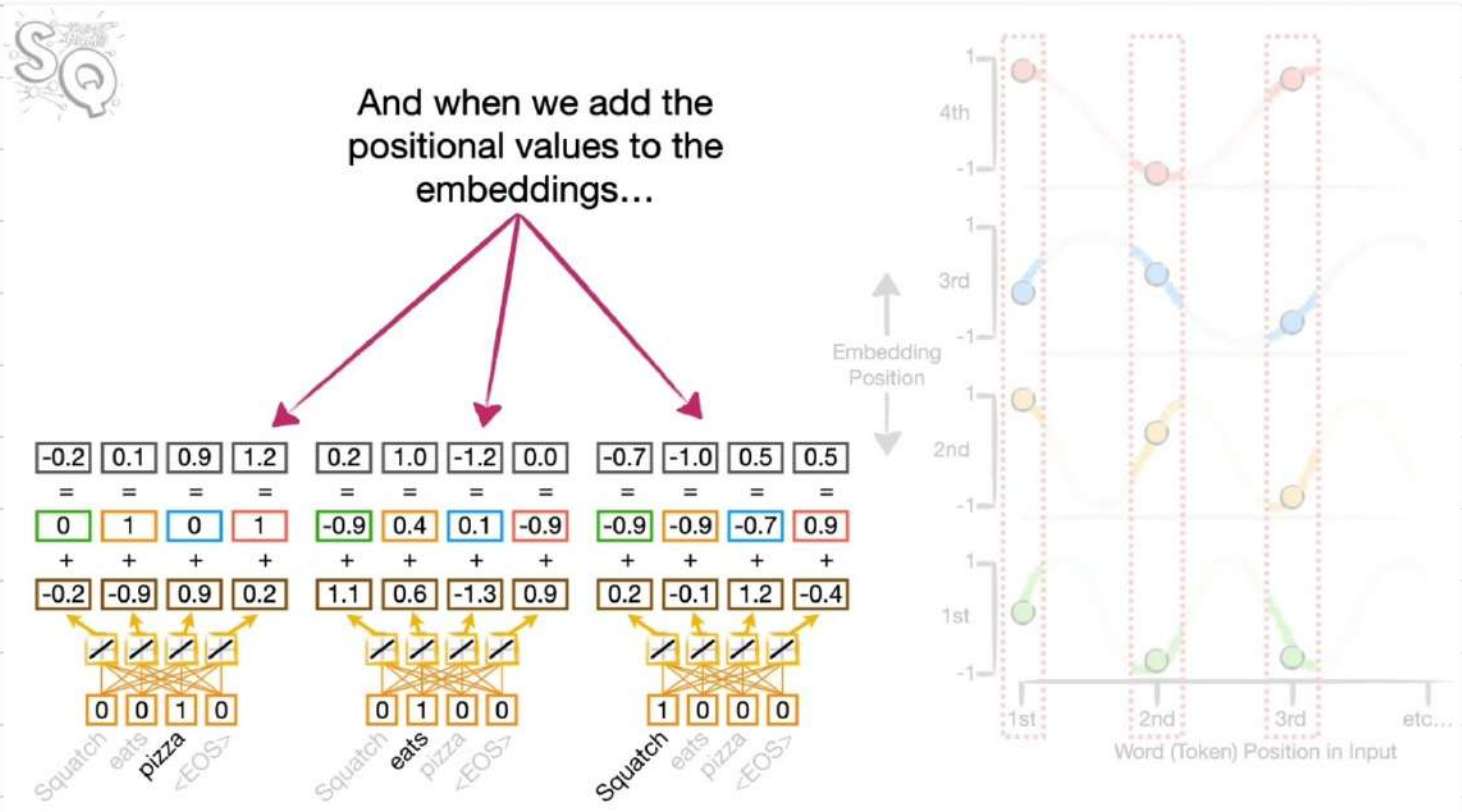
Note : Because the Sine and Cosine squiggels are repetitive, it's possible that two words might get the same position, or y-axis values. (Ex: the 2nd and 3rd words both got -0.9 for the first position value). However, because the squiggels get wider for larger embedding positions, and the more embedding values we have, then the wider squiggels get, then, even with a repeat value here and there, we end up with a unique sequence of position values for each word.



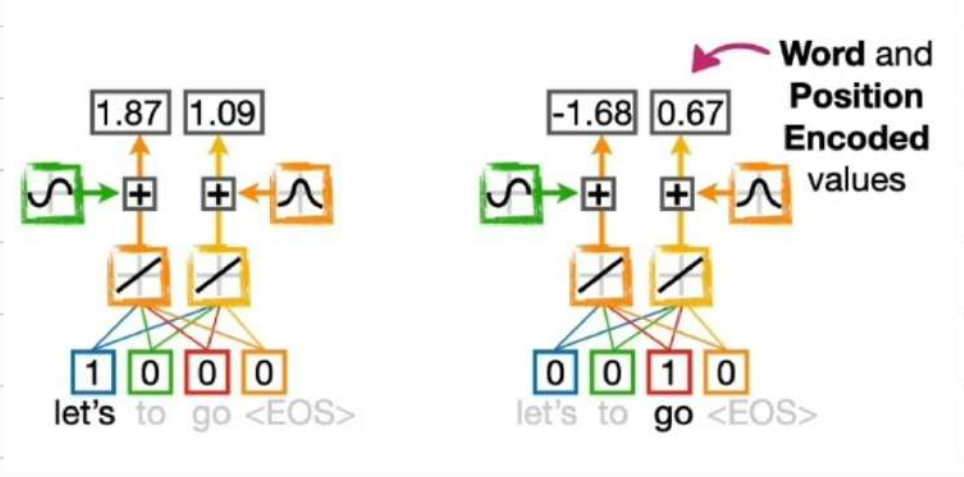
Note : If we had the sentence "pizza eats Squatch"
- The embeddings for the 1st and 3rd words get swapped
- But the positional values of the 1st, the 2nd and the 3rd words stay the same.

→ So we end up with new Positional Encoding for the 1st and 3rd words.

Illustration for the sequence "pizza eats Squatch" :

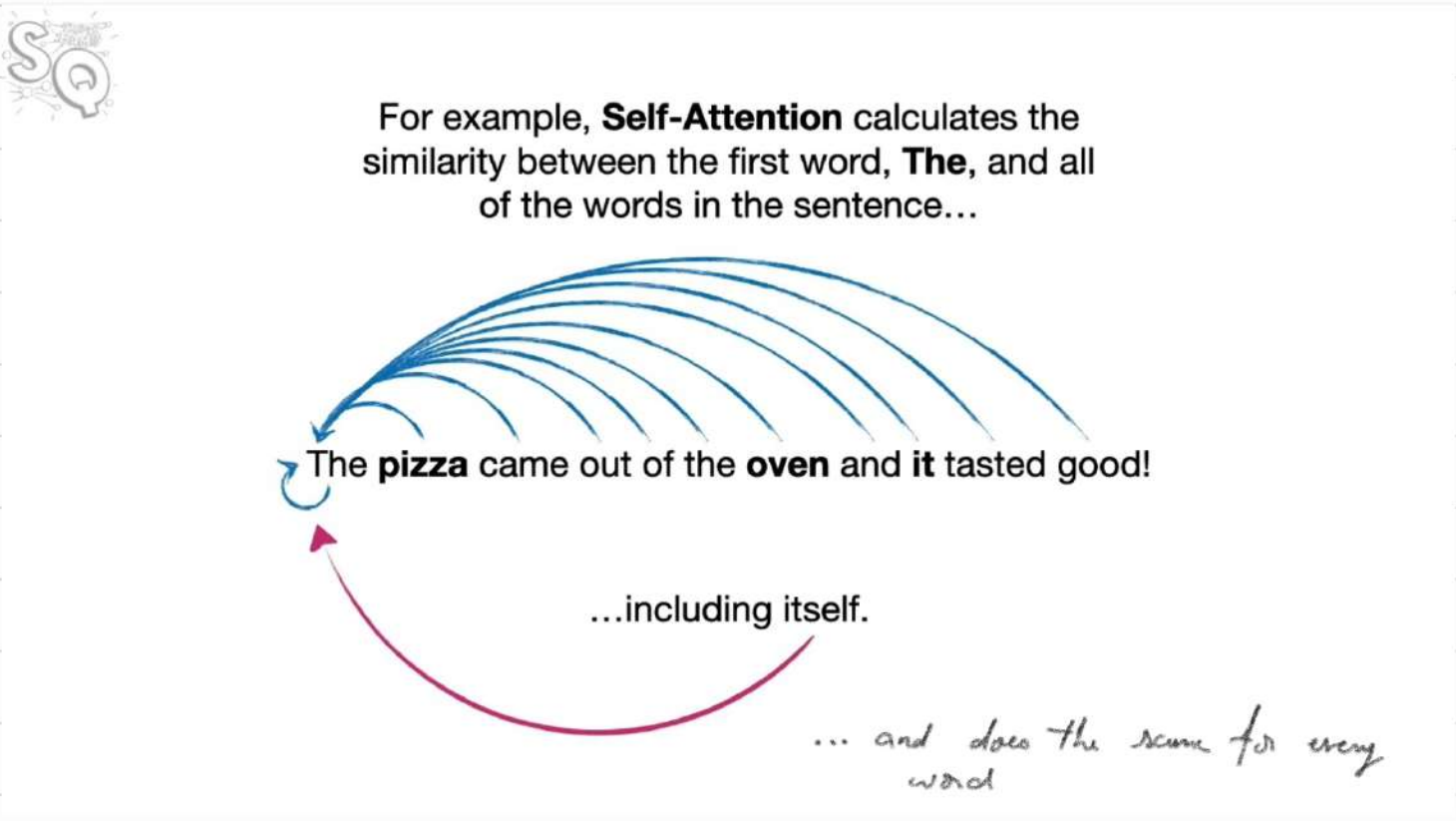


Back to the main example: "Let's go"

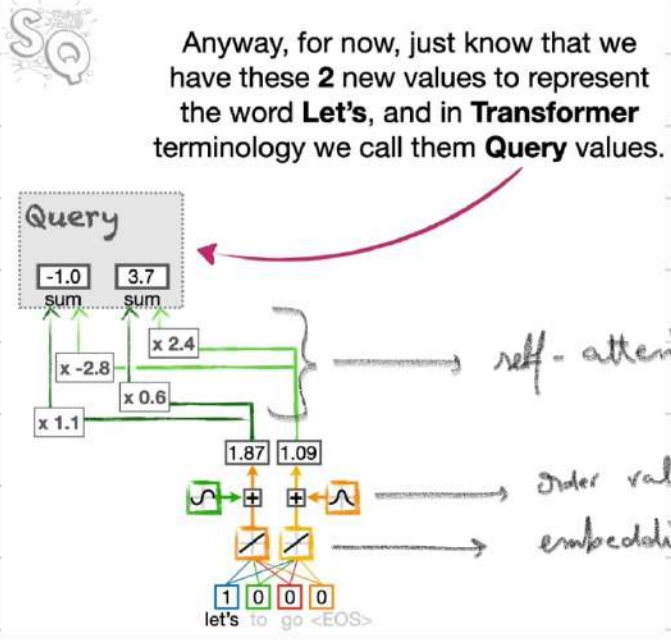


Self-Attention: how a transformer keeps track of the relationships among words

"The pizza came out of the oven and it tasted good!"



- Once the similarities are calculated, they are used to determine how the Transformer encodes each word.
- For example, if you looked at a lot of sentences about pizza and the word it was more commonly associated with pizza than oven, then the similarity score for pizza will cause it to have larger impact on how the word it is encoded by the Transformer.



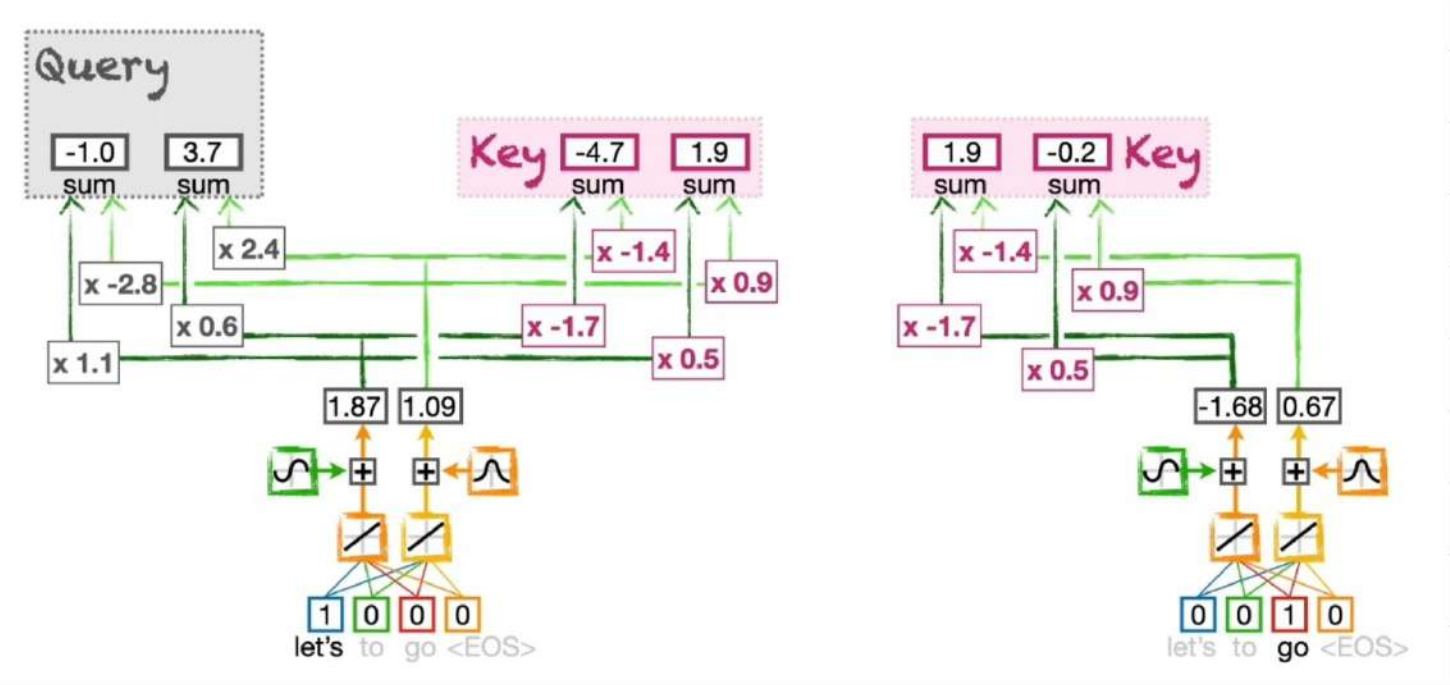
$1{,}87 \times 1{,}1 + 1{,}09 \times -2{,}8 = -1$

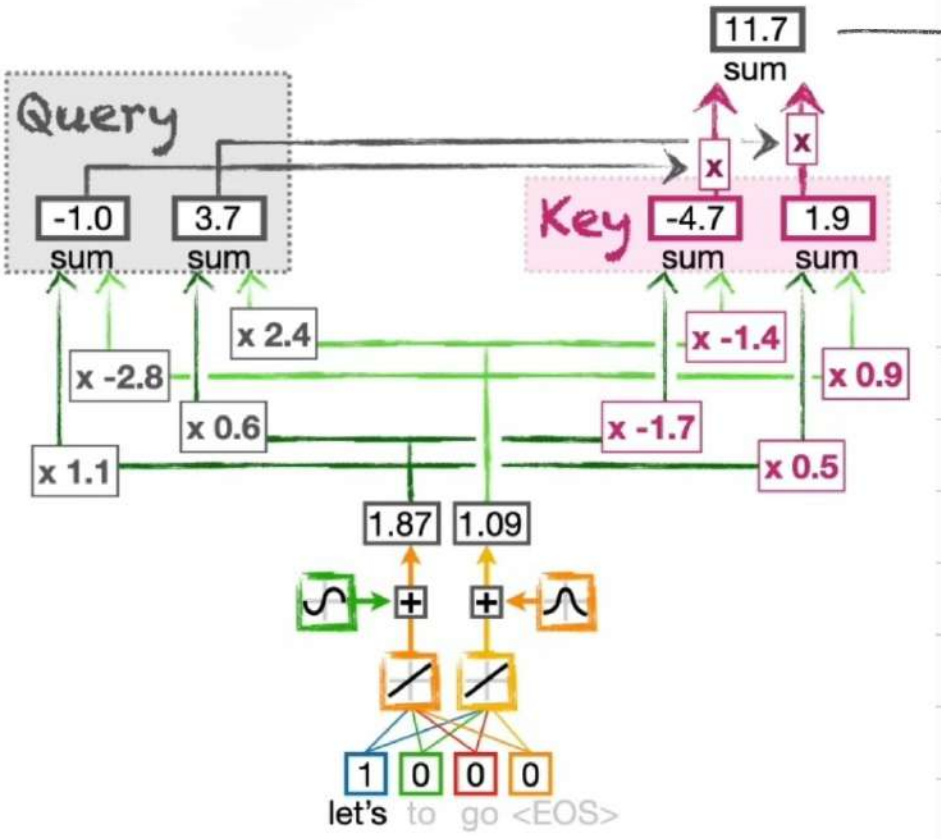
$1{,}87 \times 0{,}6 + 1{,}09 \times 2{,}4 = 3{,}7$

} Query Values

- After obtaining the Query values for the word "Let's", we use them to calculate the similarity between itself and the word go. And we do this by creating 2 new values, just like we did for the Query, to represent the word Let's, and we create 2 new values to represent the word go. Both sets of new values are called Key values, and we use them to calculate similarities with the Query for let's.

- One way to calculate the similarities between the Query and the Keys is to calculate a Dot product.

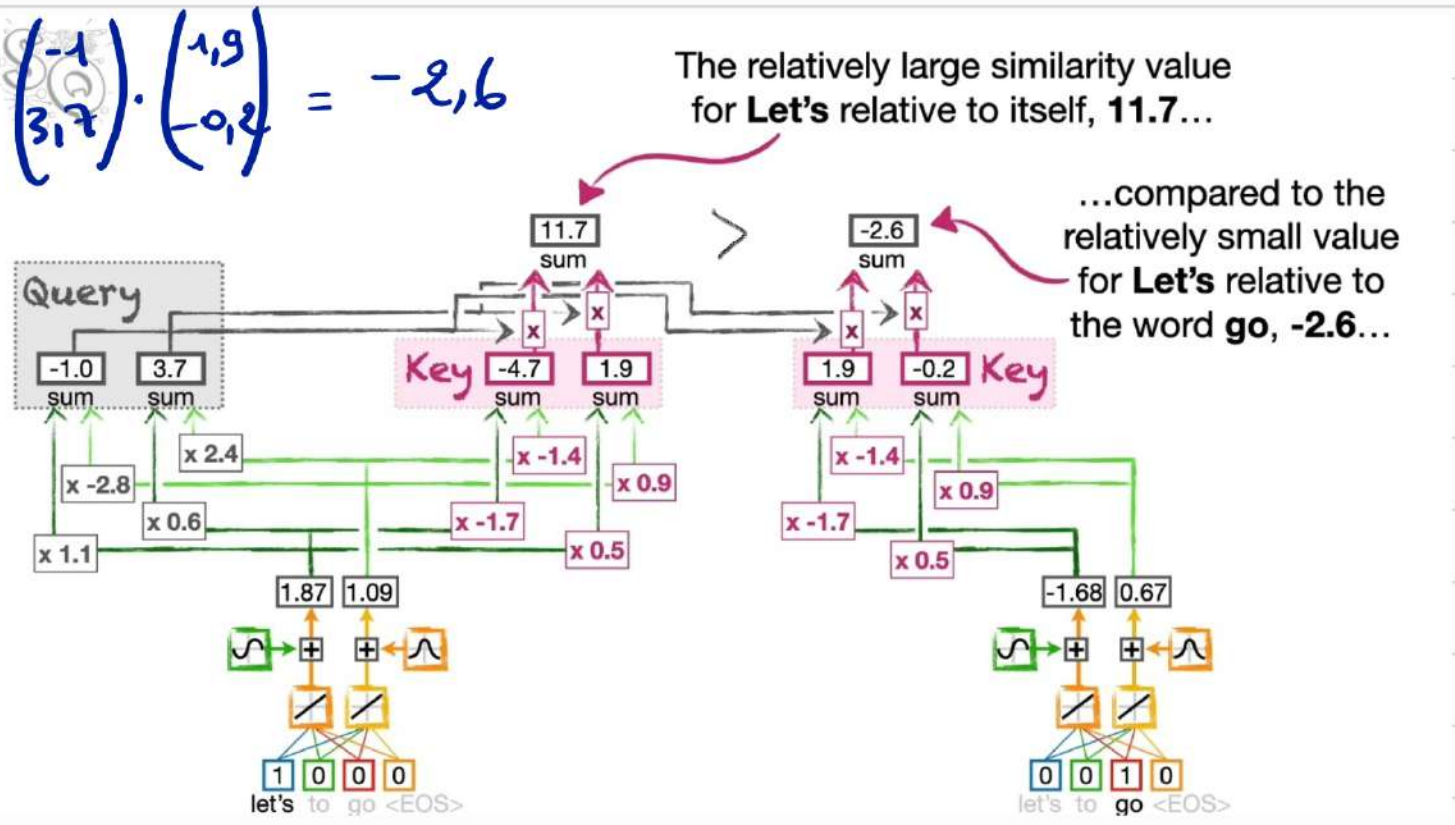




→ the dot product (represents the similarity between the Query for Let's and the Key for go)

$$\begin{pmatrix} -1 \\ 3,7 \end{pmatrix} \cdot \begin{pmatrix} -4,7 \\ 1,9 \end{pmatrix} = 11,7$$

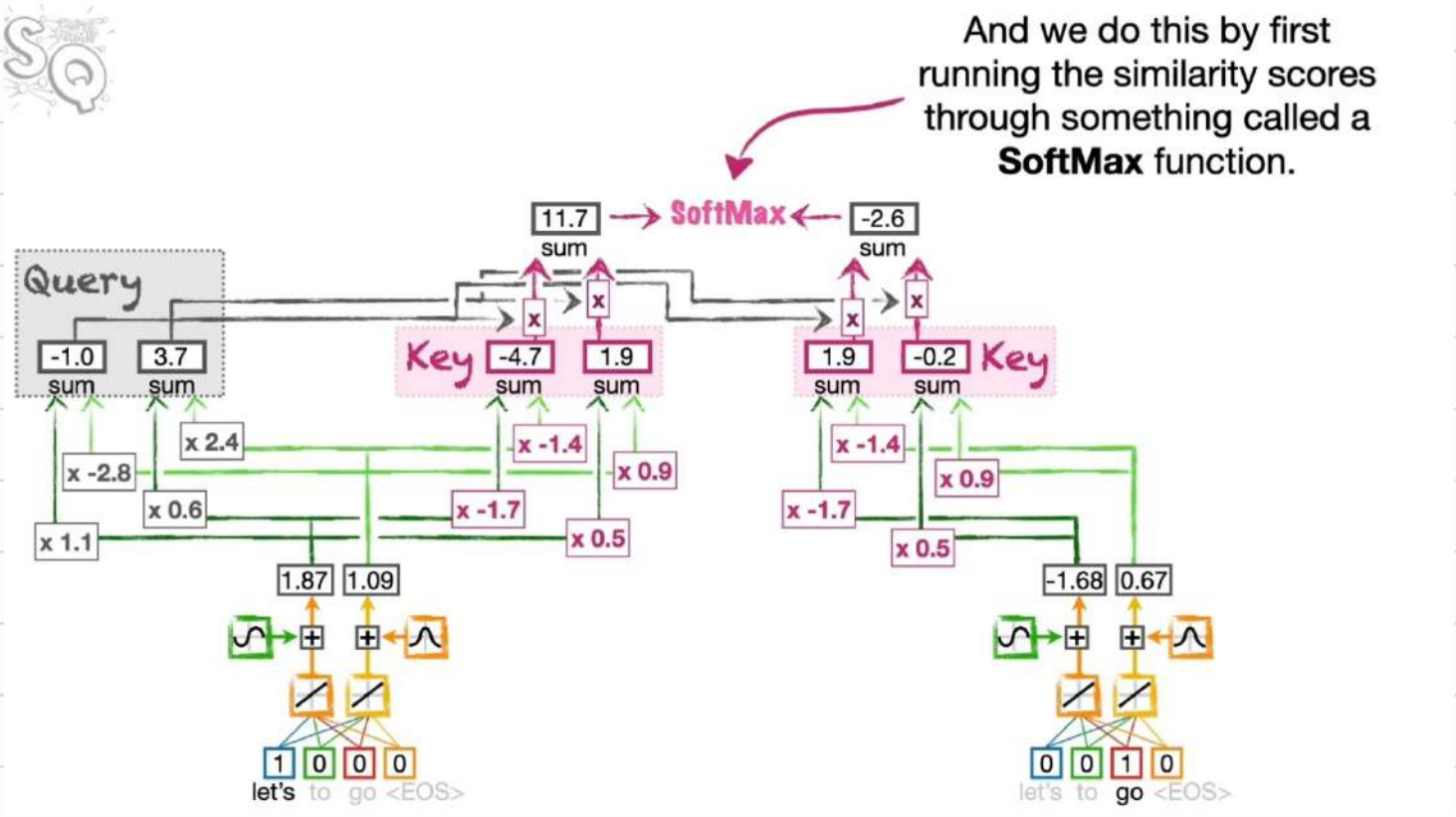
$$\begin{pmatrix} -1 \\ 3,7 \end{pmatrix} \cdot \begin{pmatrix} 1,9 \\ -0,2 \end{pmatrix} = -2,6$$



Tells us that Let's is much more similar to itself than it is to the word go

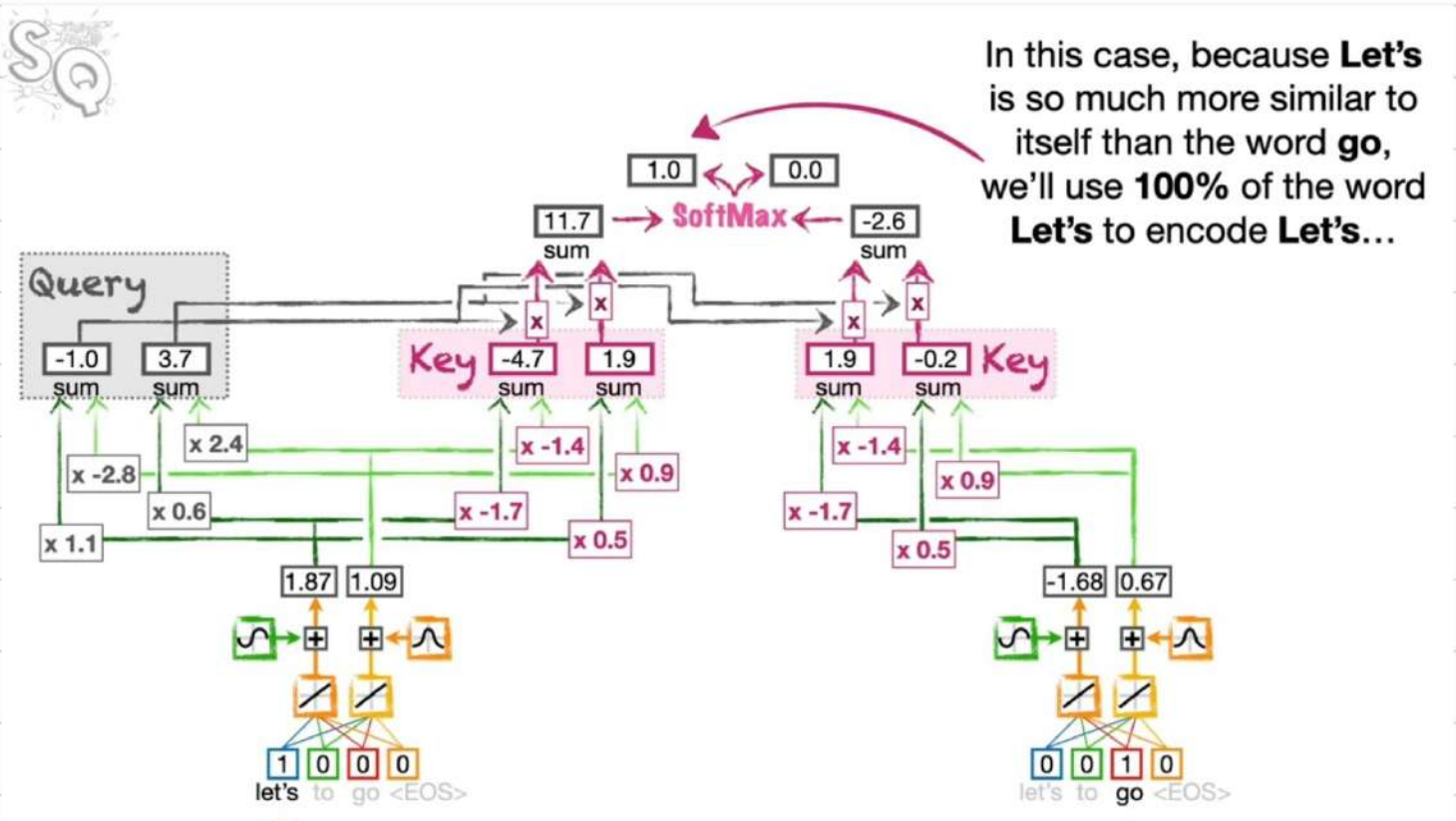
Note: Check cosine similarity too

Since 11,7 > -2,6, we want Let's to have more influence on its encoding than go:



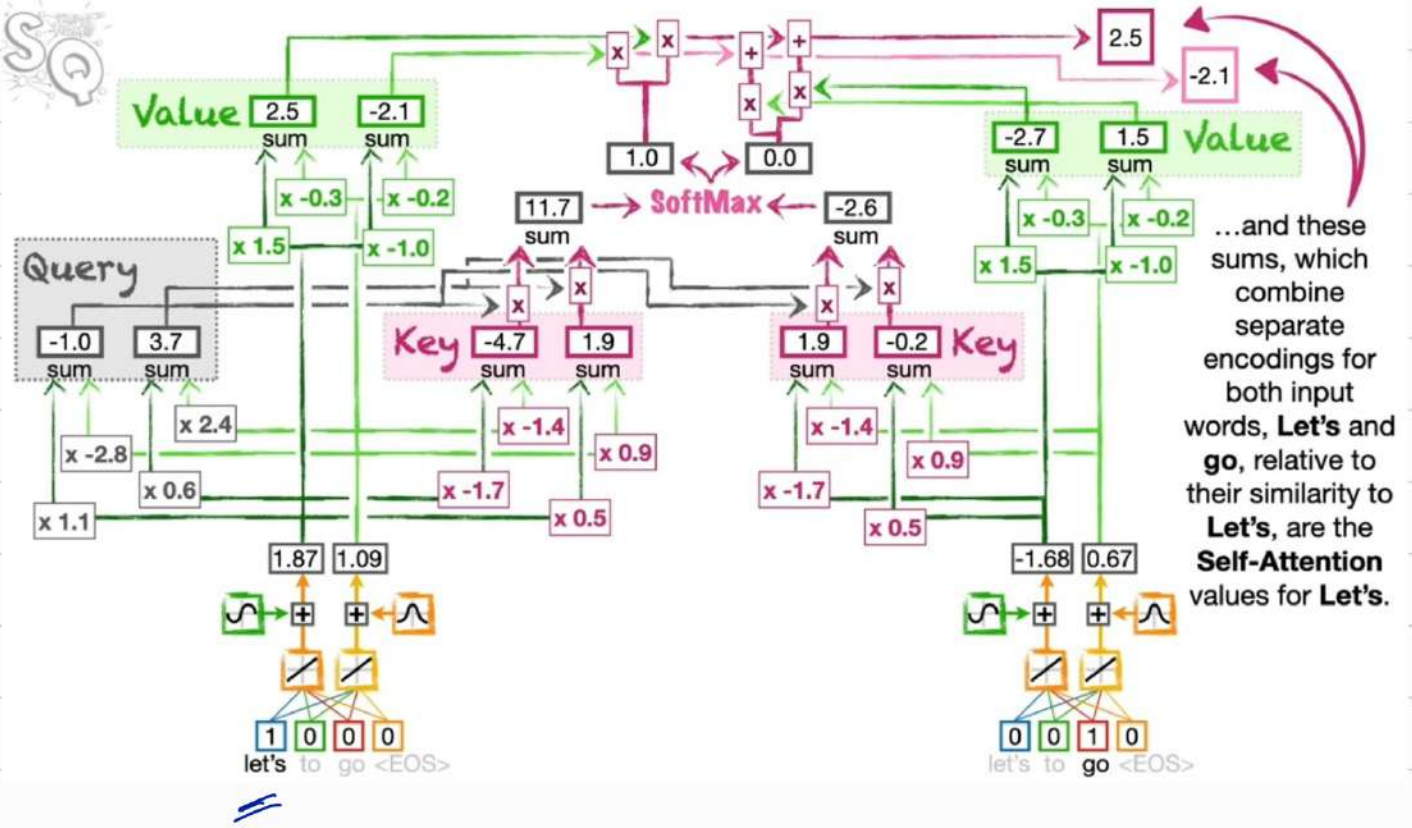
The main idea of a **SoftMax** function is that it preserves the order of the input values, from low to high, and translates them into numbers between **0** and **1** that add up to **1**.

So we can think of the output of the **SoftMax** function as a way to determine what percentage of each input word we should use to encode the word **Let's**.

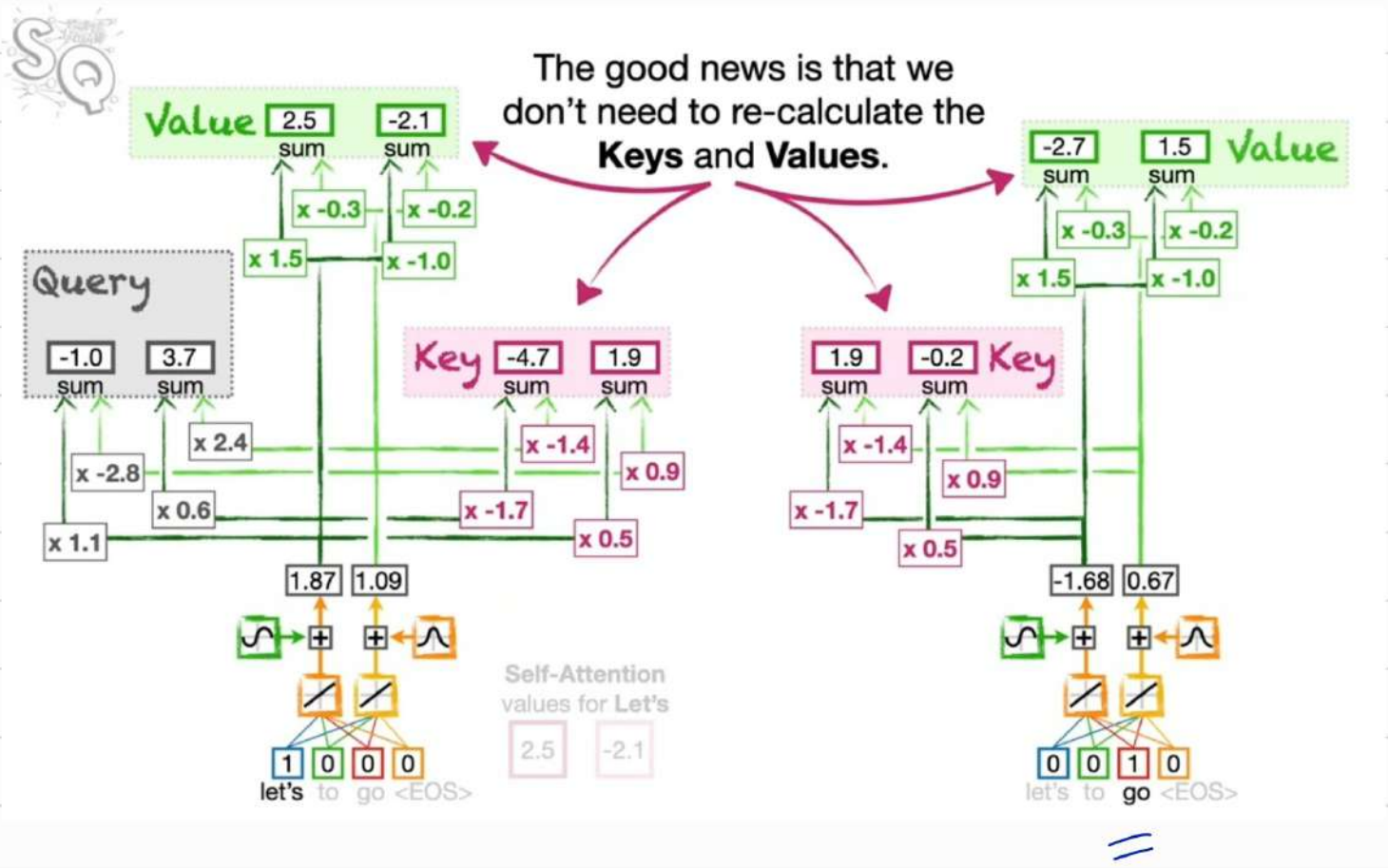


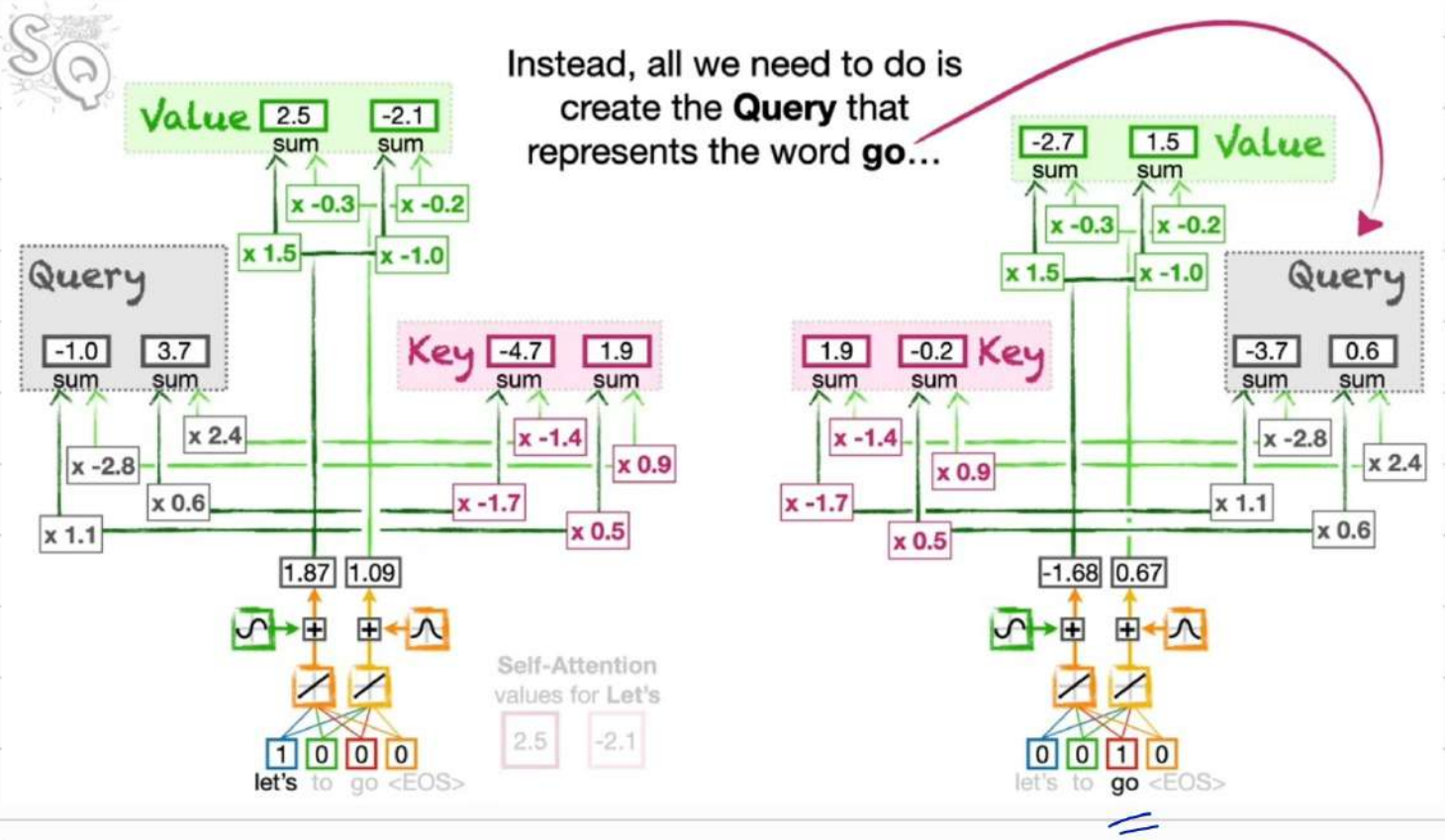
...and **0%** of the word **go** to encode the word **Let's**.

Note : Check out a video about Argmax and Softmax
- We use 2 values for Let's that should multiply by 1
- " " " " go " " " " 0

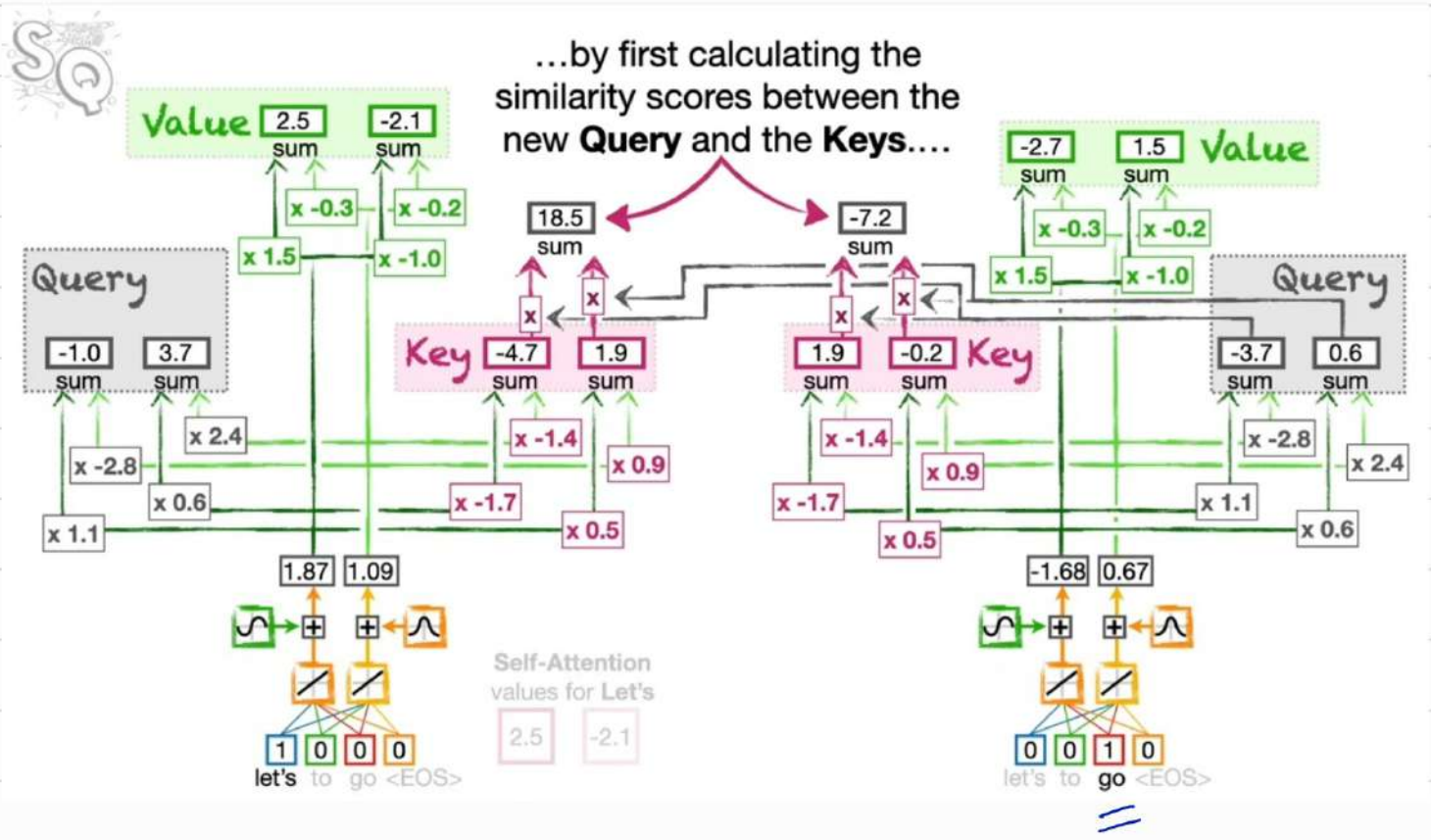


. So 2,5 and -2,1 are the self-attention values for let's
. Let's calculate the self-attention values for go :

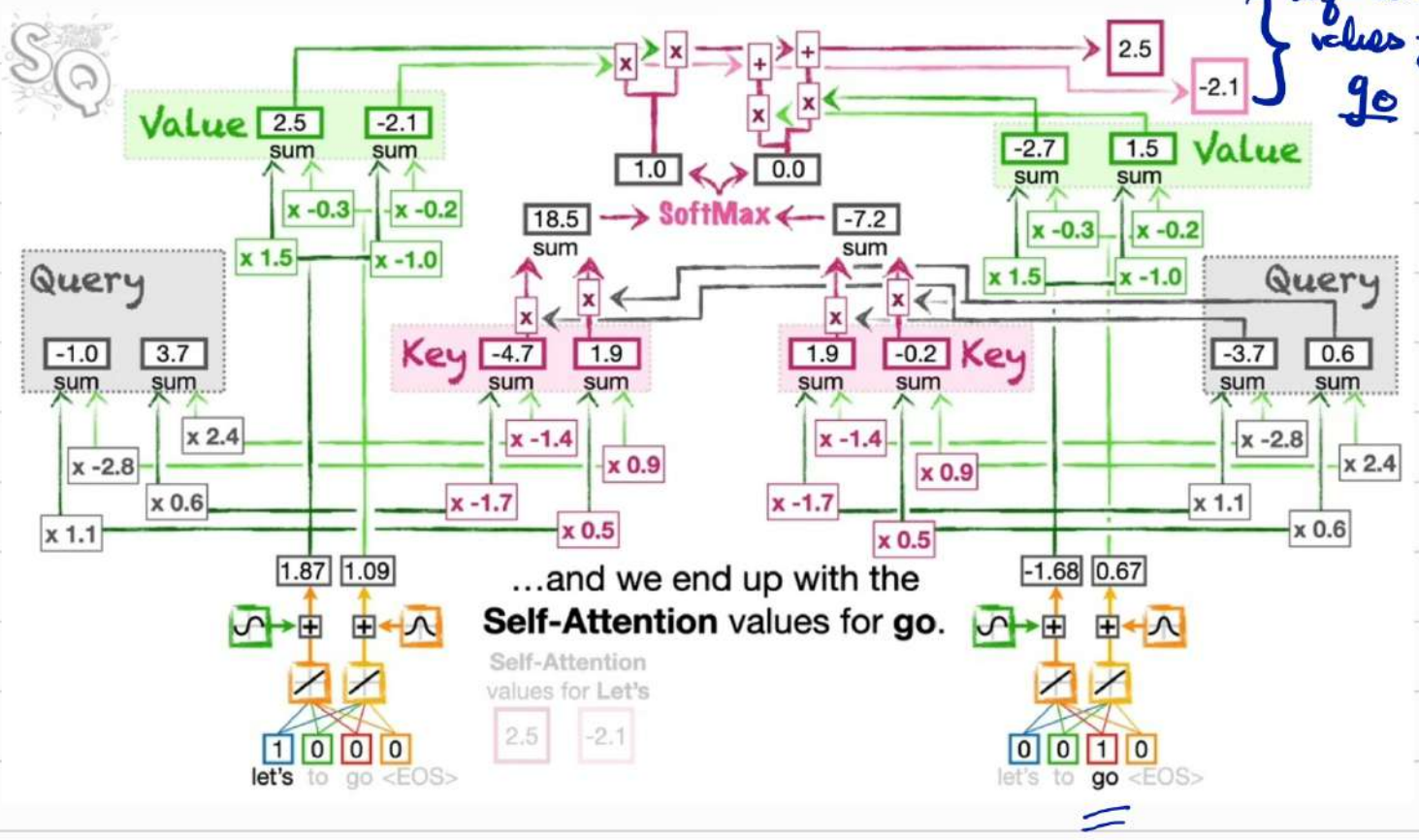




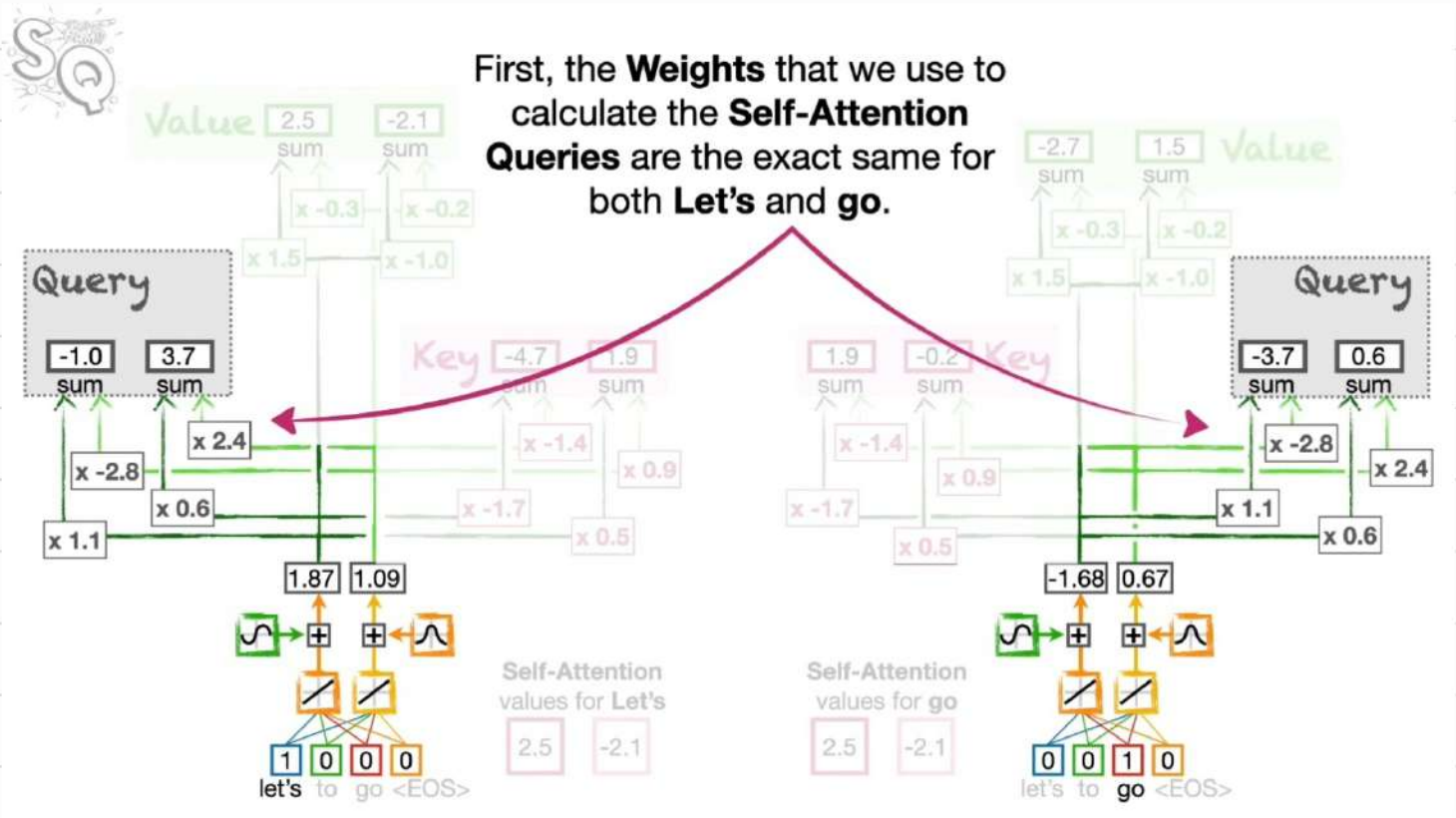
And do the math :



... and run the similarity scores through a Softmax and then scale the values and add them together



Notes about self-attention :

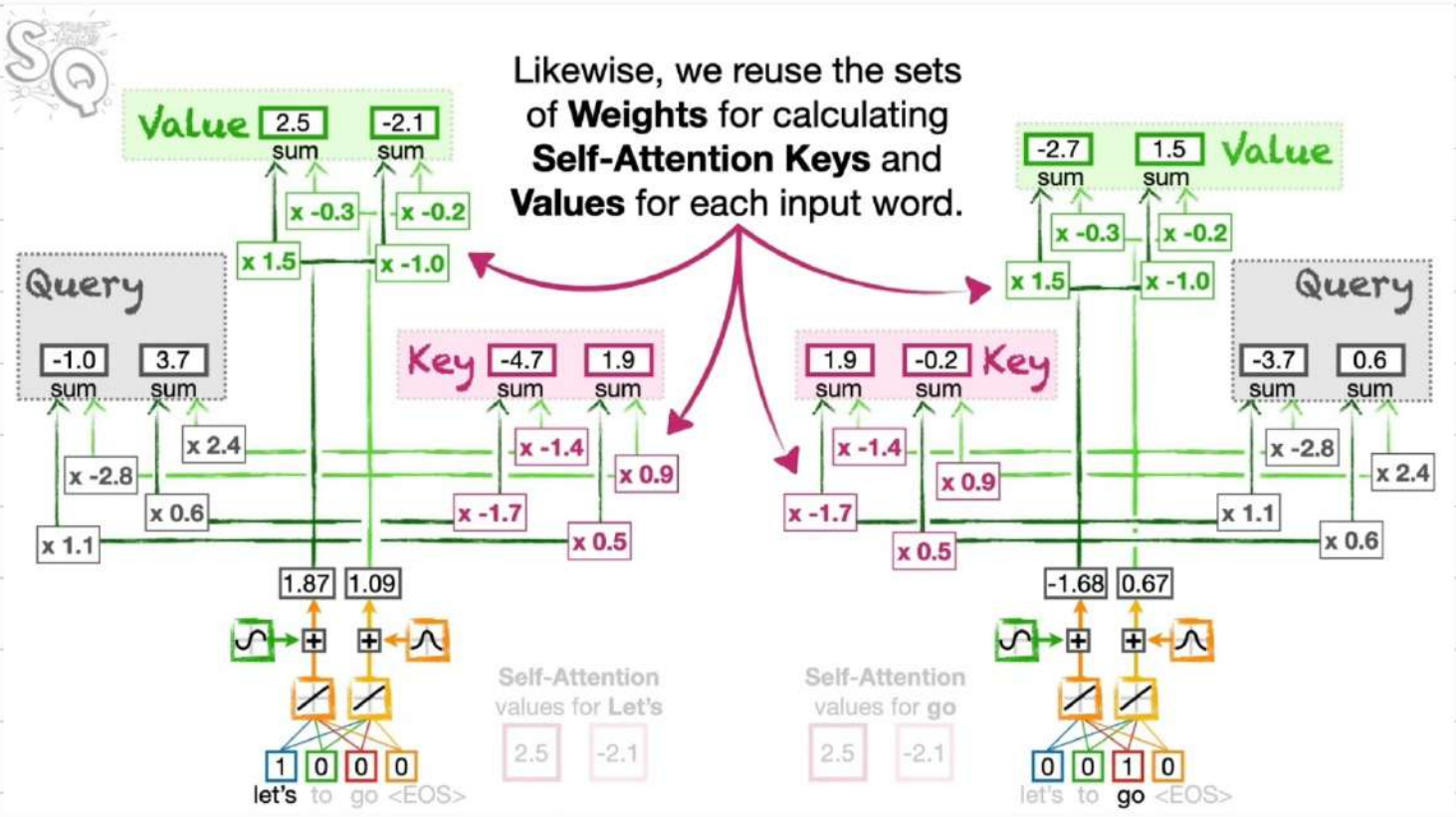


$1{,}87 \times 1{,}1 + 1{,}09 \times -2{,}8 = -1$
$1{,}87 \times 0{,}6 + 1{,}09 \times 2{,}4 = 3{,}7$
} Query values for Let's

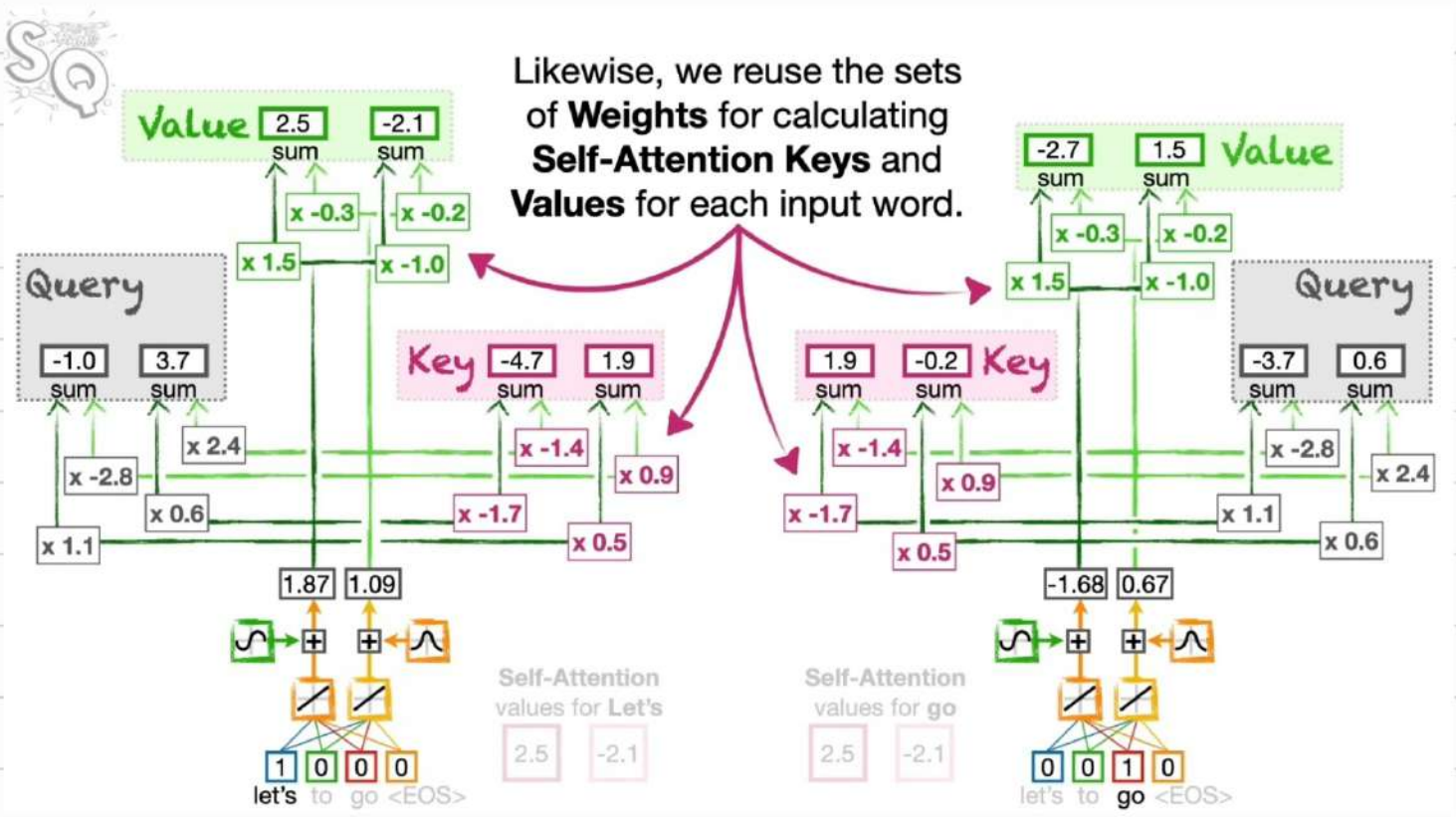
$-1{,}68 \times 1{,}1 + 0{,}67 \times -2{,}8 = -3{,}7$
$-1{,}68 \times 0{,}6 + 0{,}67 \times 2{,}4 = 0{,}6$
} Query values for go

In other words, this example uses one set of weights for calculating Self-Attention Queries, regardless of how many words are in the input.

$\begin{pmatrix} 1{,}2 \\ -2{,}8 \end{pmatrix} \quad \begin{pmatrix} 0{,}6 \\ 2{,}4 \end{pmatrix}$

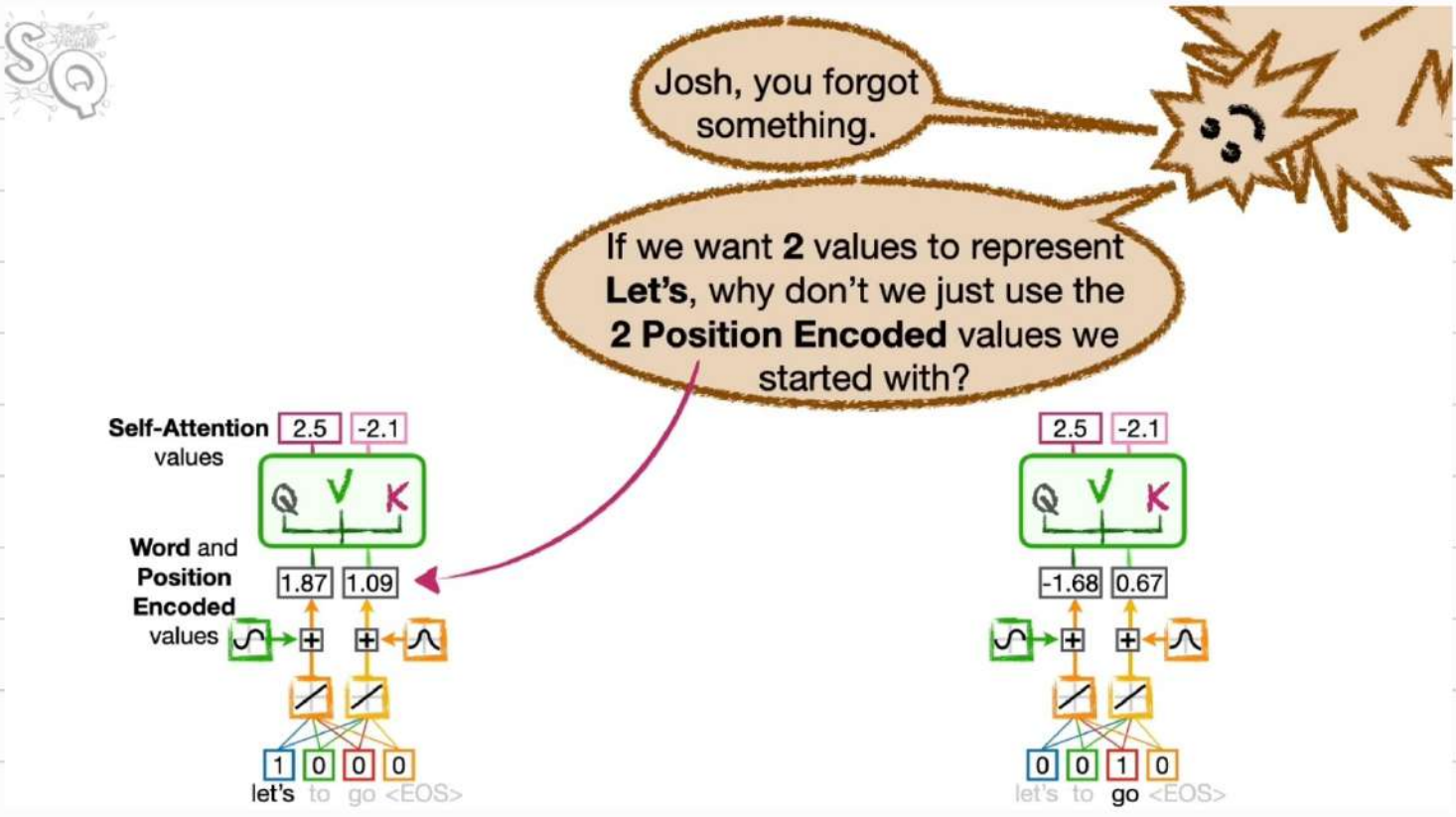


This means that no matter how many words are input into the transformer, we just reuse the same sets of weights for self-attention Queries, Keys and values



In other words, we don't have to calculate the Query, Key and value for the first word first, before moving on to the second word. And because we can do all of the computation at the same time, Transformers can take advantage of parallel computing and run fast.

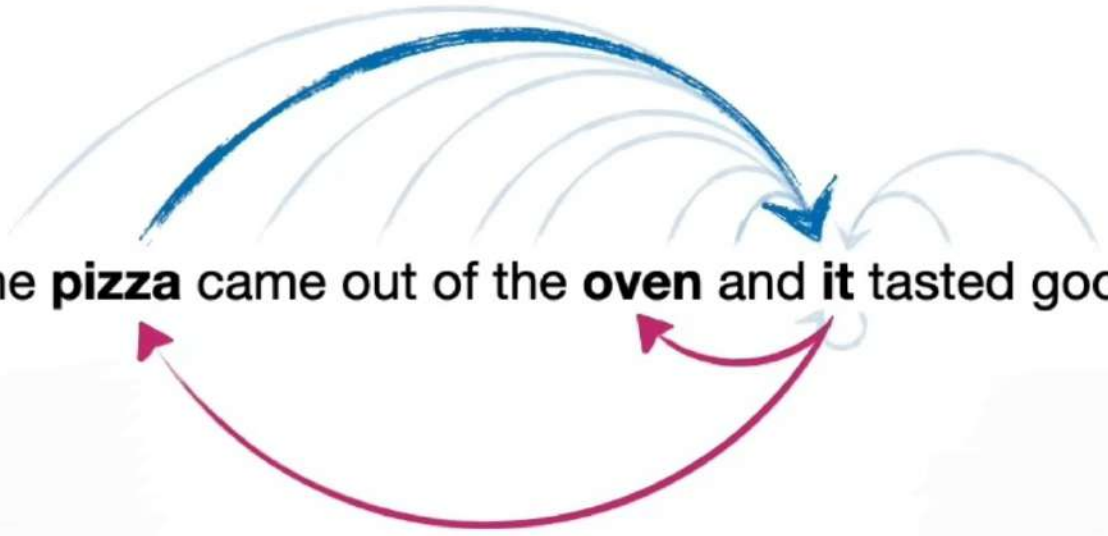
Question :



Answer :

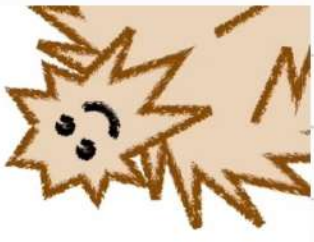
First, the new **Self-Attention** values for each word contain input from all of the other words, and this helps give each word context.

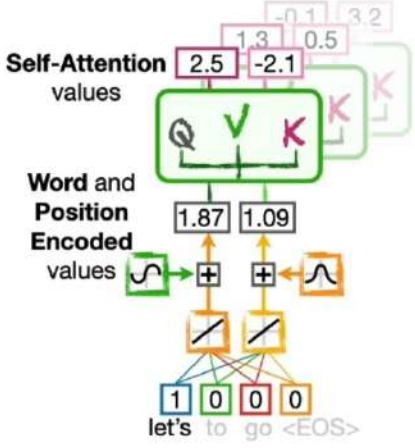
And this can help establish how each word in the input is related to the others.

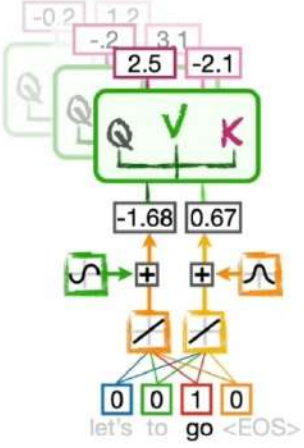


Also, if we think of this unit, with its weights for calculating Queries, Keys and values as self-attention cell, then in order to correctly establish how words are related in complicated sentences and paragraphs, we can create a stack of self-attention cells each with its own sets of weights, that we apply to the position encoded values of each word, to capture different relationships among the words.

In the manuscript that first described **Transformers**, they stacked **8 Self-Attention** cells, and they called this **Multi-Head Attention**.

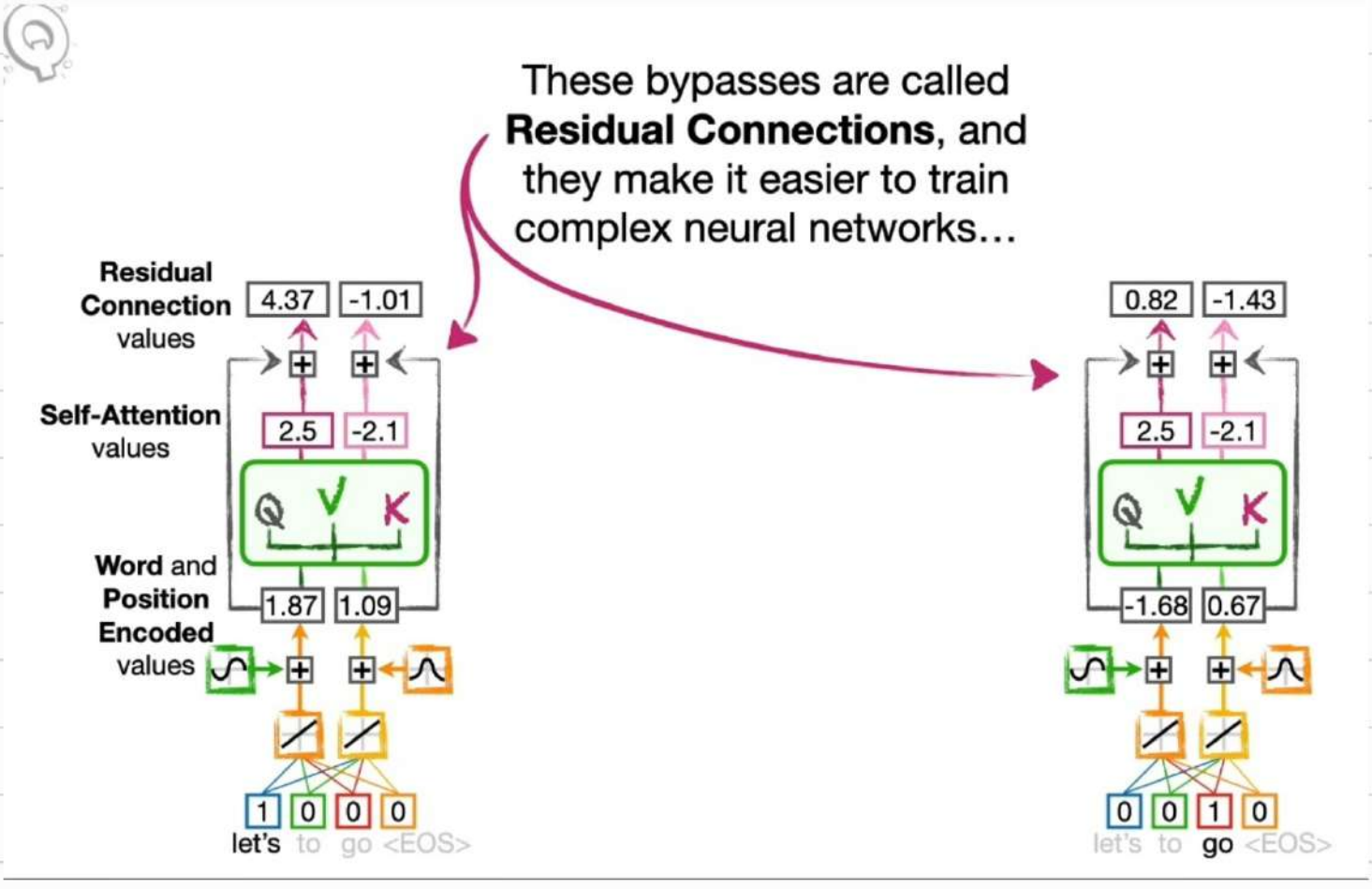






Why 8 ? (Still a relevant question)

Going back to our example with only one self-attention cell, there is one more thing we need to do to encode the input : we take the position encoded values and add them to the self attent° values

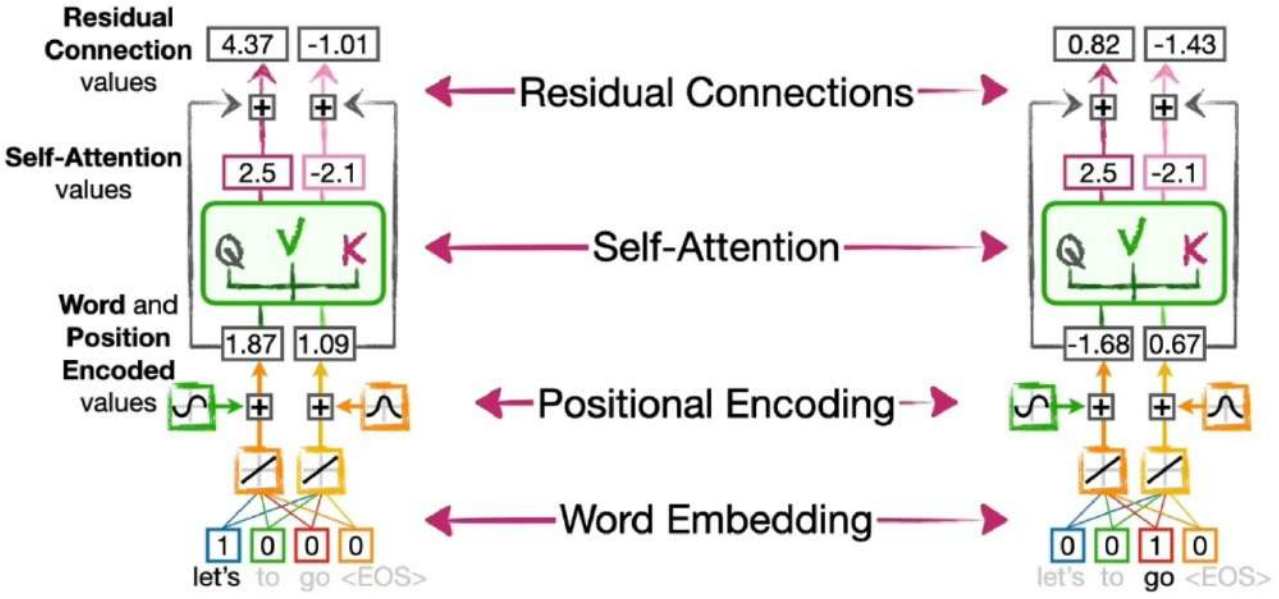


…by allowing the **Self-Attention** layer to establish relationships among the input words without having to also preserve the **Word Embedding** and **Position Encoding** information.

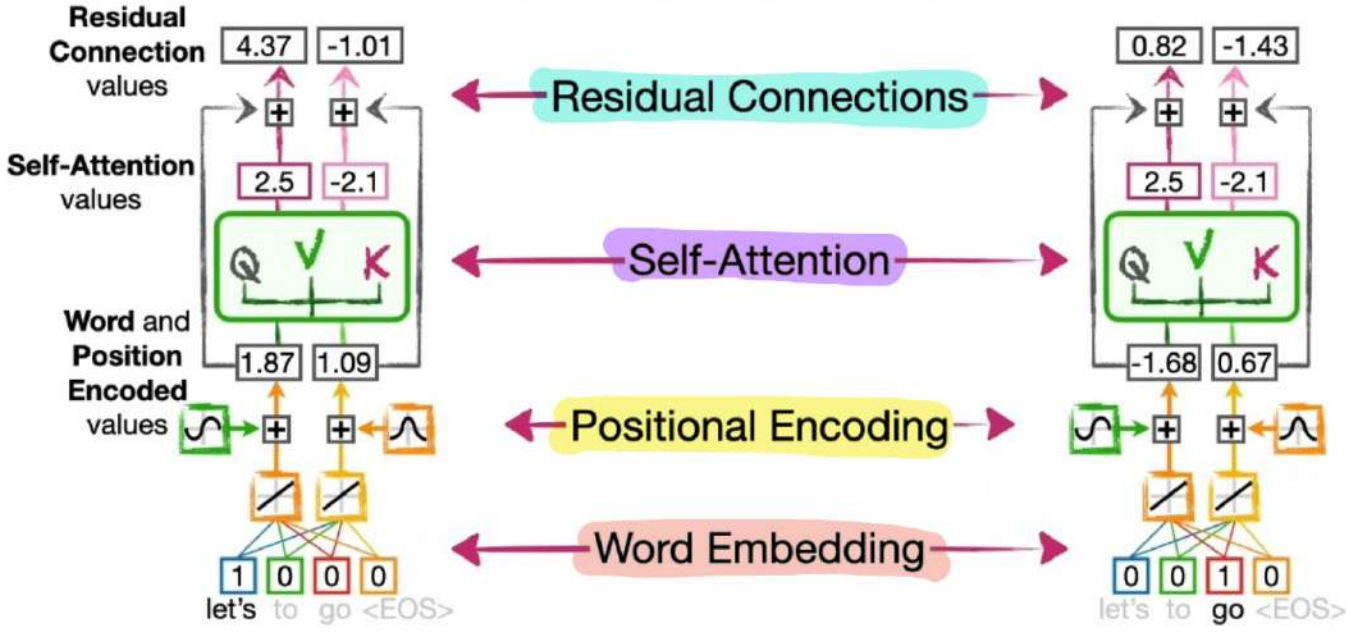
bam.

End of this simple transformer's encoding !

NOTE: This simple **Transformer** only contains the parts required for encoding the input.

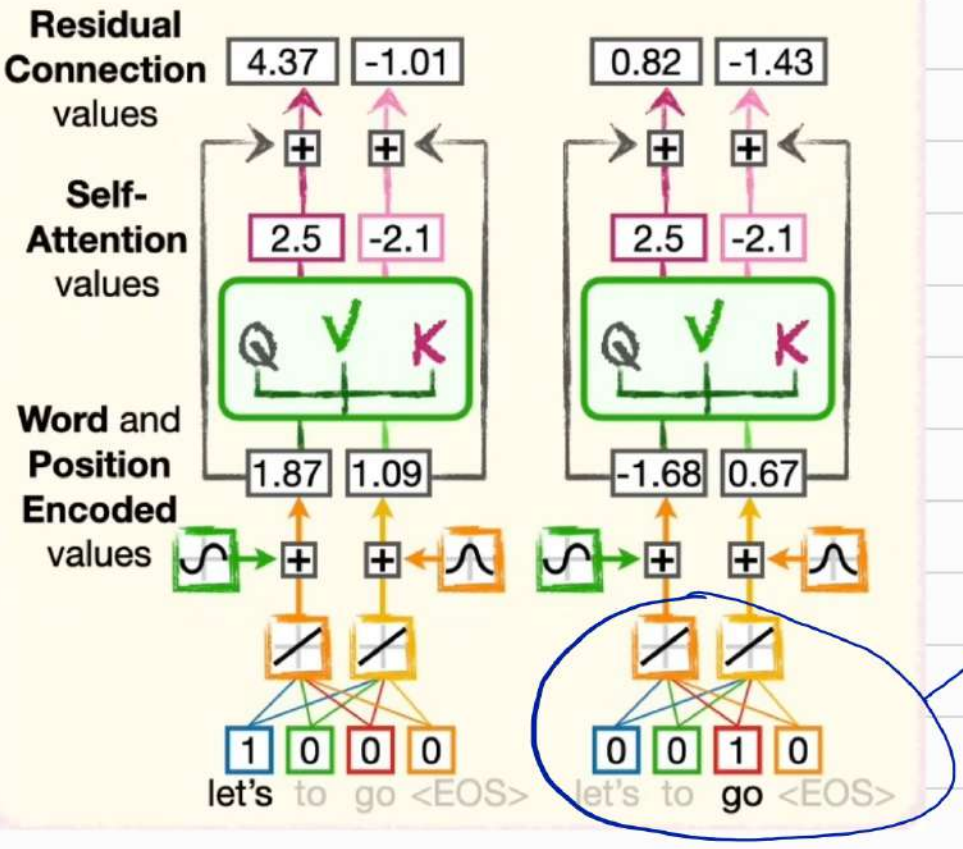


These **4** features allow the **Transformer** to: encode words into numbers, encode the positions of the words, encode the relationships among the words, and relatively easily and quickly train in parallel.



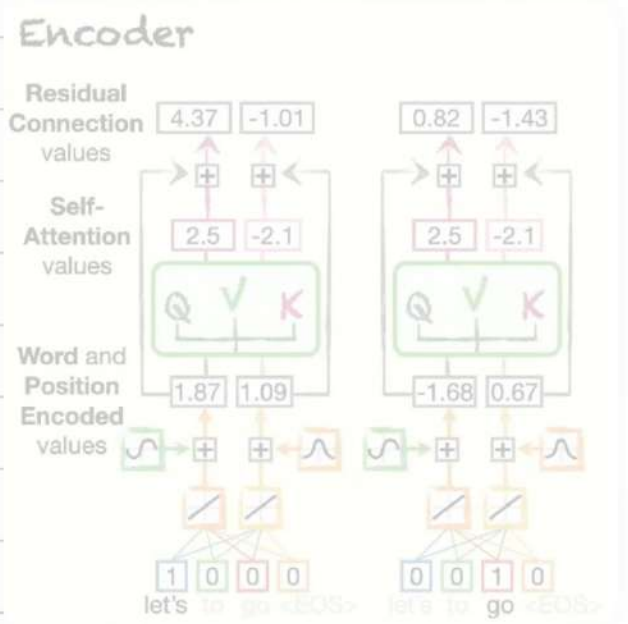
That was the Encoder part, now for the Decoder (into Spanish)

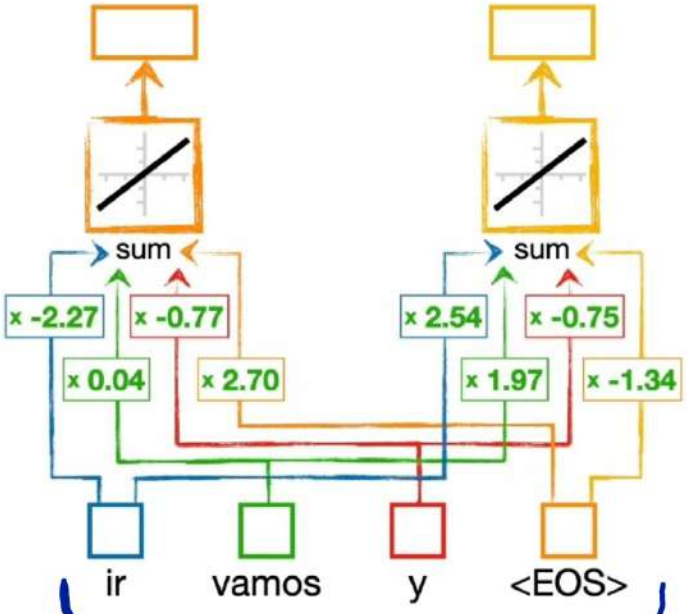
Encoder



Decoder also starts with word embedding

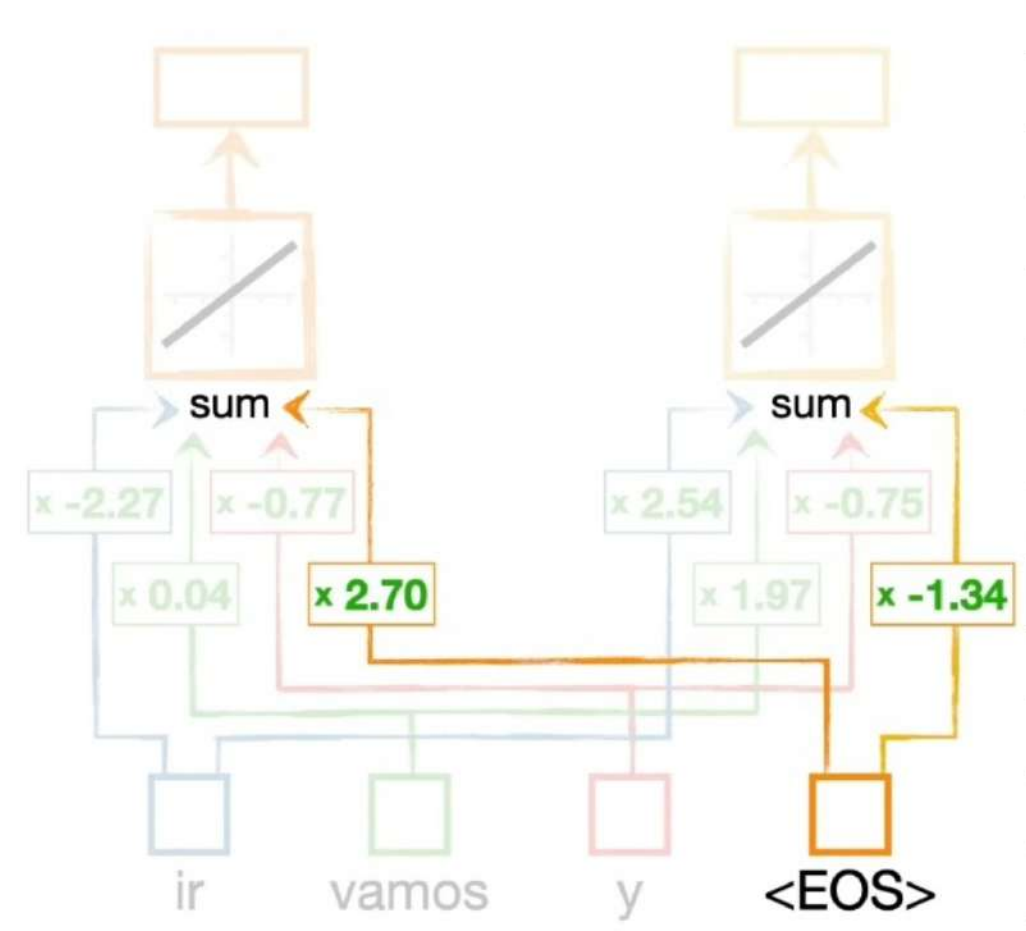
However, this time we create embedding values for the output vocabulary, which consists of the Spanish words...





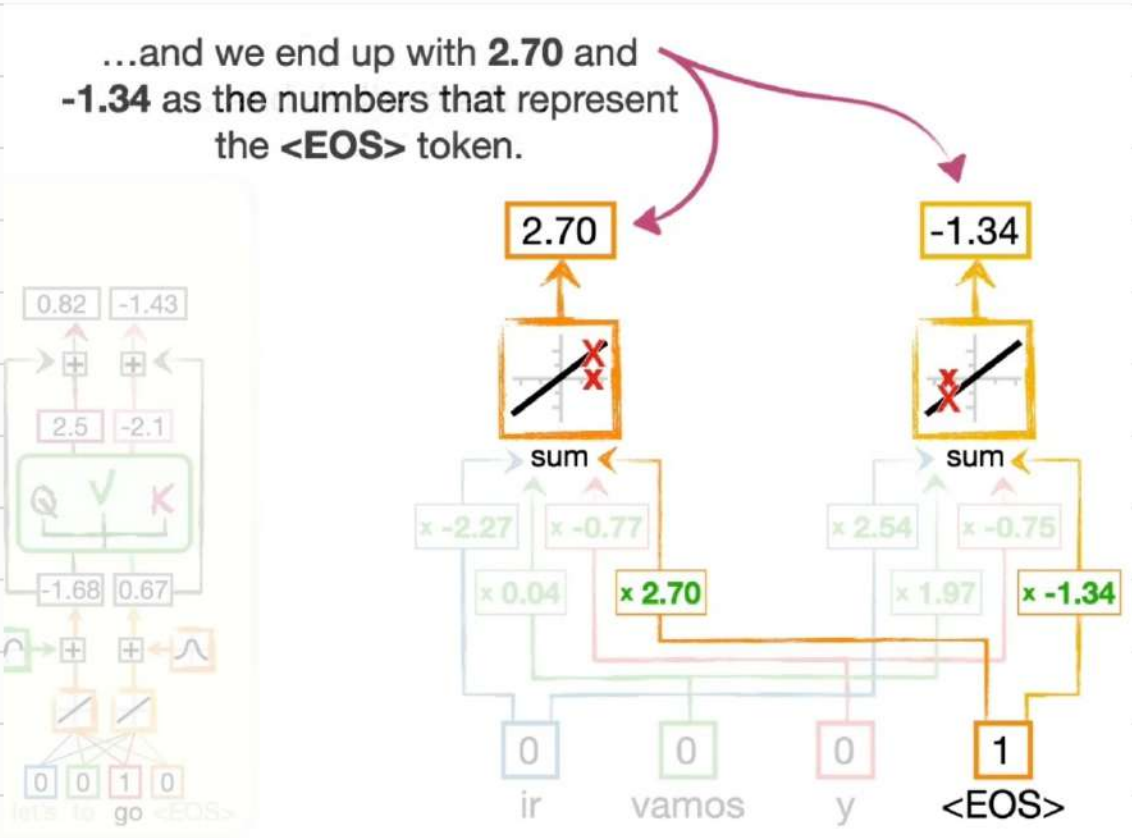
Spanish vocabulary

Now, because we just finished encoding the English sentence Let's go, the decoder starts with embedding values for the <EOS> (end of sequence) token.
In this case, we're using the <EOS> token to start the decoding because that is a common way to initialize the process of decoding the encoded input sentence.

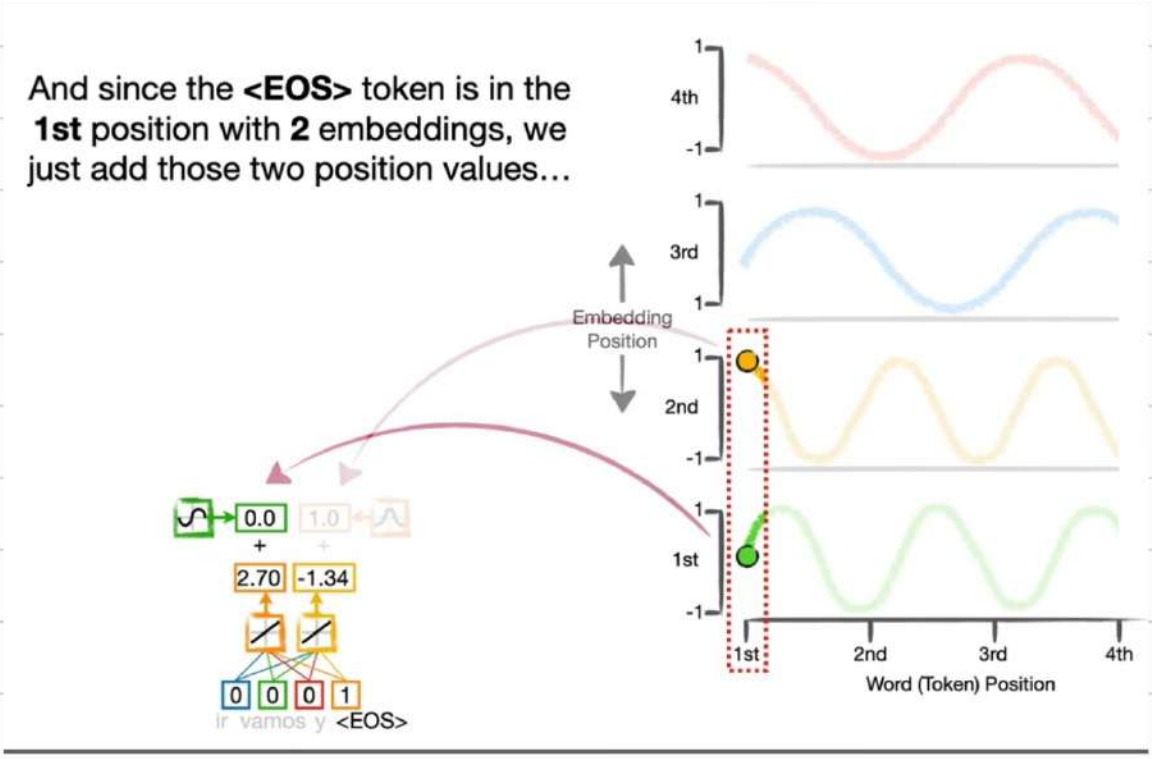


However, sometimes you see people use <SOS> for Start of Sentence, or Start of Sequence, to initialize the process.

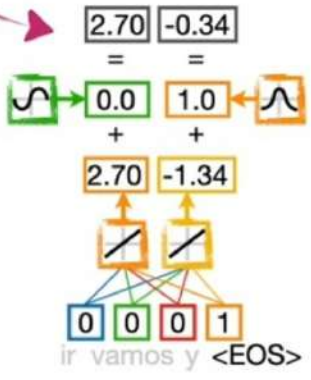
So we plug in 1 for <EOS> and 0 for everything else and do the math:

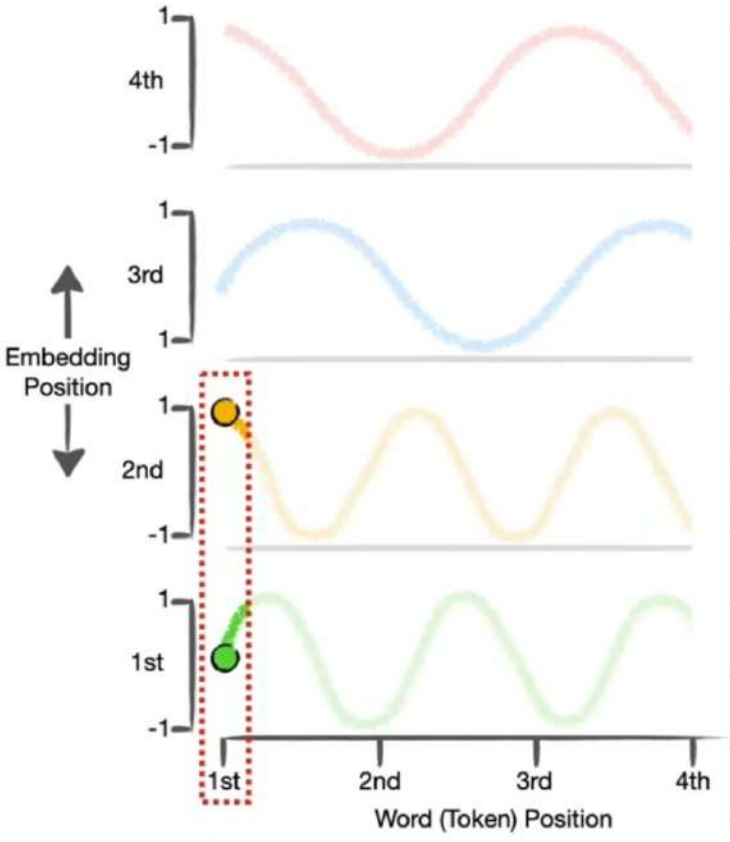


Then the positional Encoding with the exact same Sine and Cosine squiggles that we used when we encoded the input.



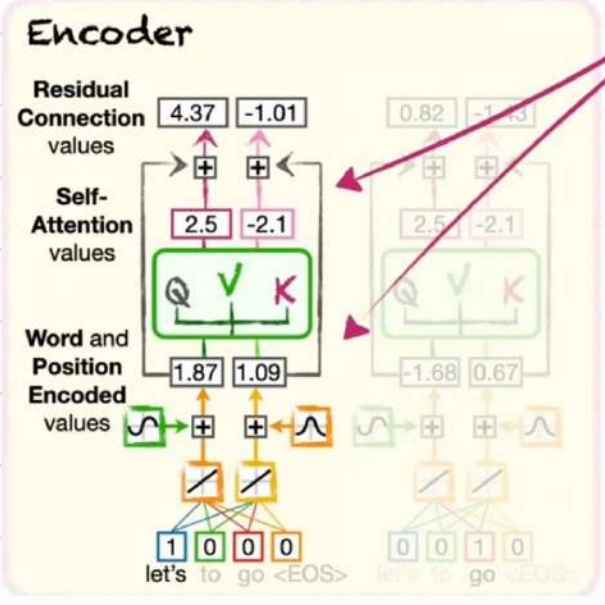
…and we get **2.70** and **-0.34** as the **Position** and **Word Embedding** values representing the **<EOS>** token.

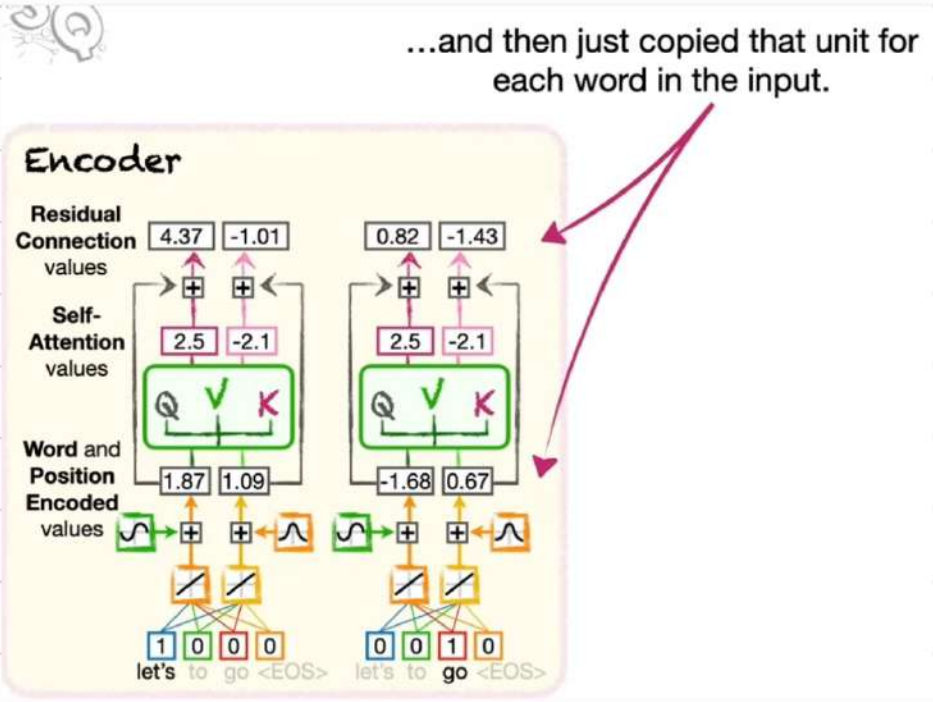


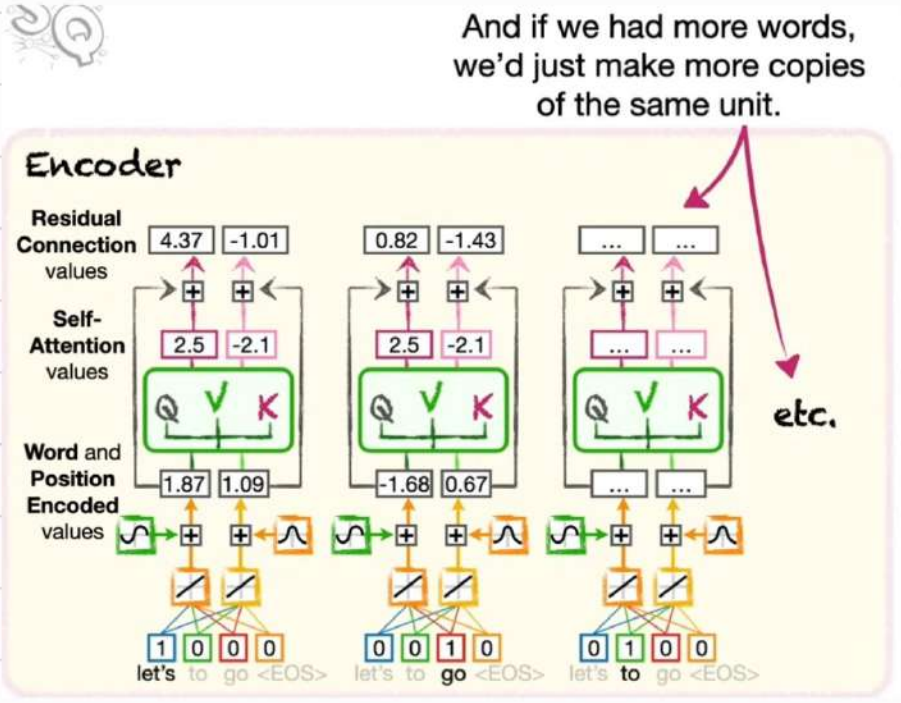


Key concept from the Encoder part:

One key concept from earlier was that we created a single unit to process an input word…





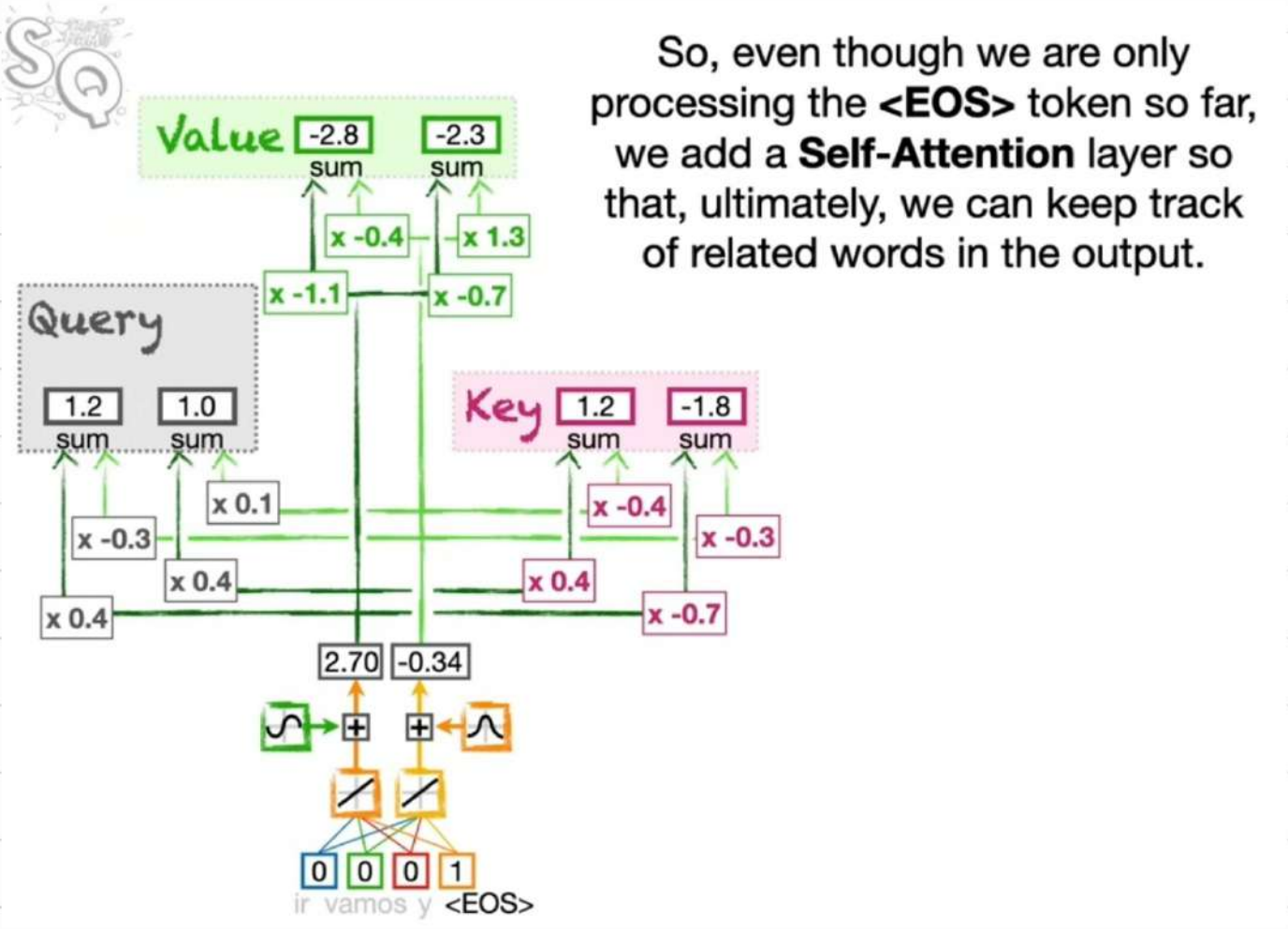


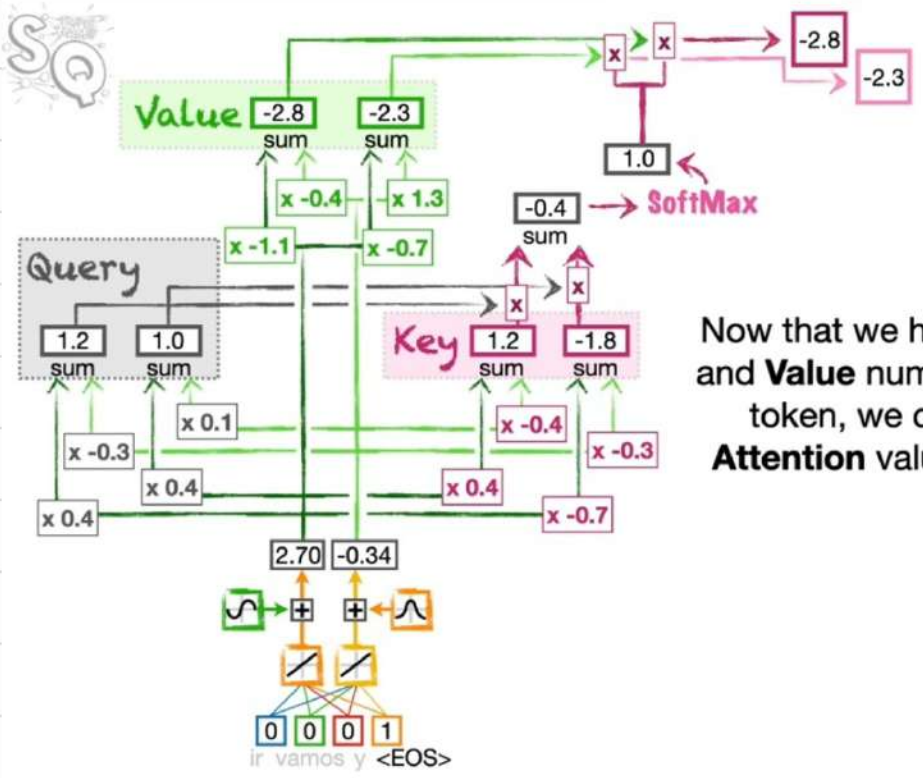
By creating a single unit that can be copied for each input word, the transformer can do all of the computation for each word in the input at the same time.
For example, we can calculate the word embeddings on different processors at the same time, and then add the positional encoding at the same time, and then calculate Queries, Keys and Values at the same time, and, once that is done, we can calculate the self attention values

at the same time, and, lastly, we can calculate the residual connections at the same time.

⇒ Doing all the computations at the same time, rather than doing them sequentially for each word, means we can process a lot of words relatively quickly on a chip with a lot of computing cores, like a GPU (graphics processing unit), or on multiple chips in the cloud.

⇒ Likewise, when we decode and translate the input, we want a single unit that we can copy for each translated word for the same reasons - we want to do the math quickly.



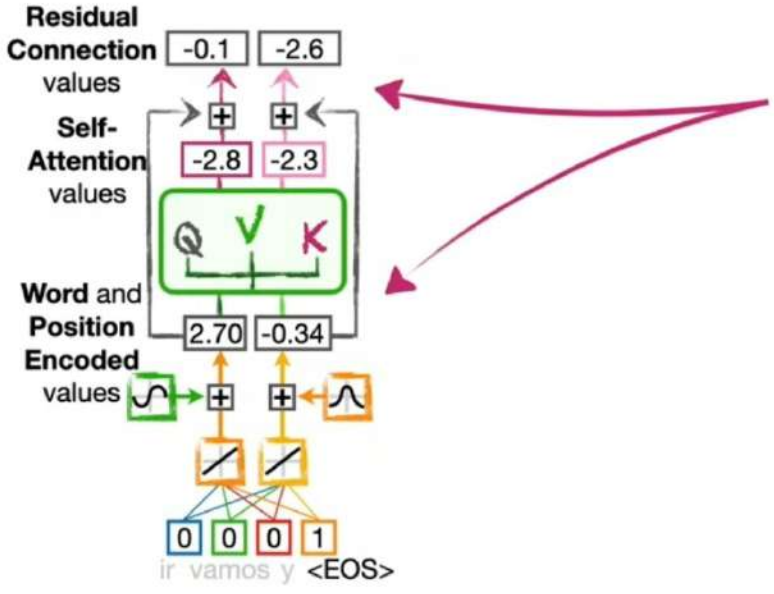


} self attention values for <EOS>

Now that we have the **Query**, **Key** and **Value** numbers for the **<EOS>** token, we calculate its **Self-Attention** values just like before.

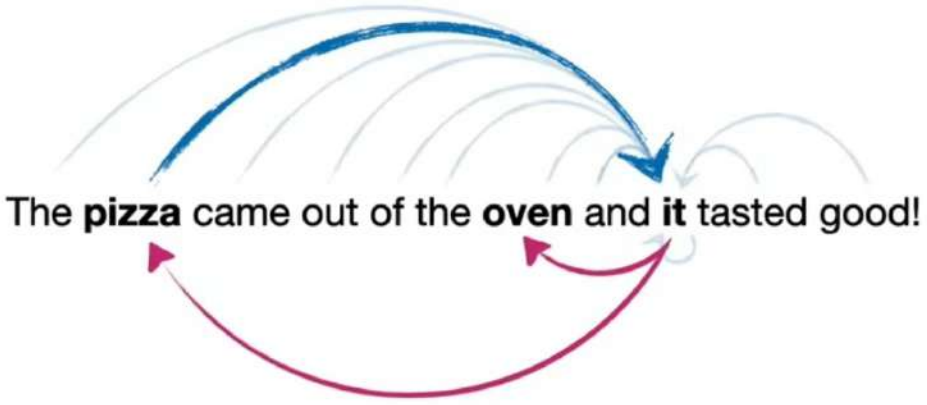
Note : The sets of weights we used to calculate the decoder's self-attention query, key, and value are <u>different</u> from the sets we used in the encoder.

Now let's consolidate the math ...



...and add **Residual Connections**, just like before.

Now, so far we've talked about how **Self-Attention** helps the **Transformer** keep track of how words are related *within* a sentence...



However, since we're translating a sentence, we also need to keep track of the relationships *between* the input sentence and the output.

Example :

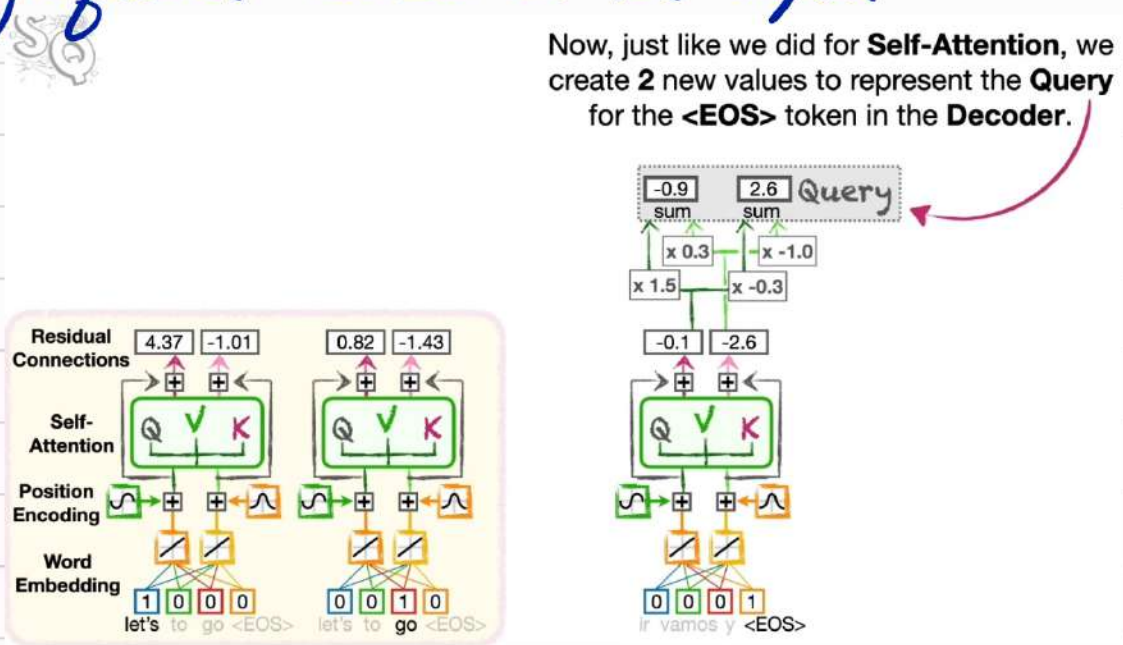
Input sentence : Don't eat the delicious looking and smelling pizza.

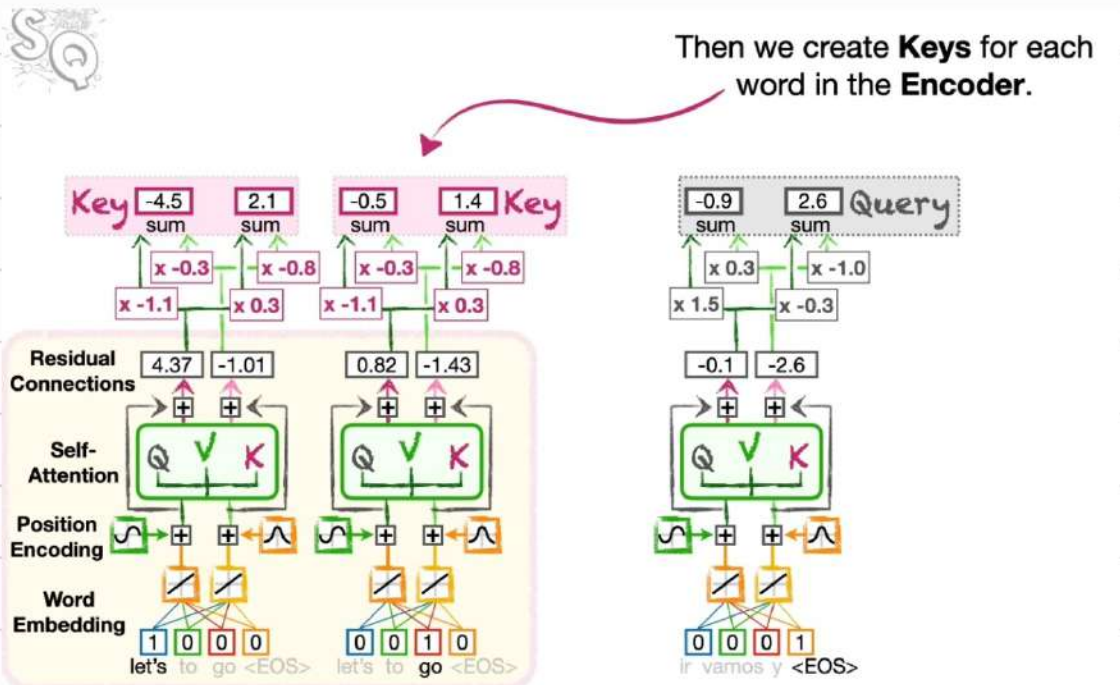
↳ very important to keep track of it

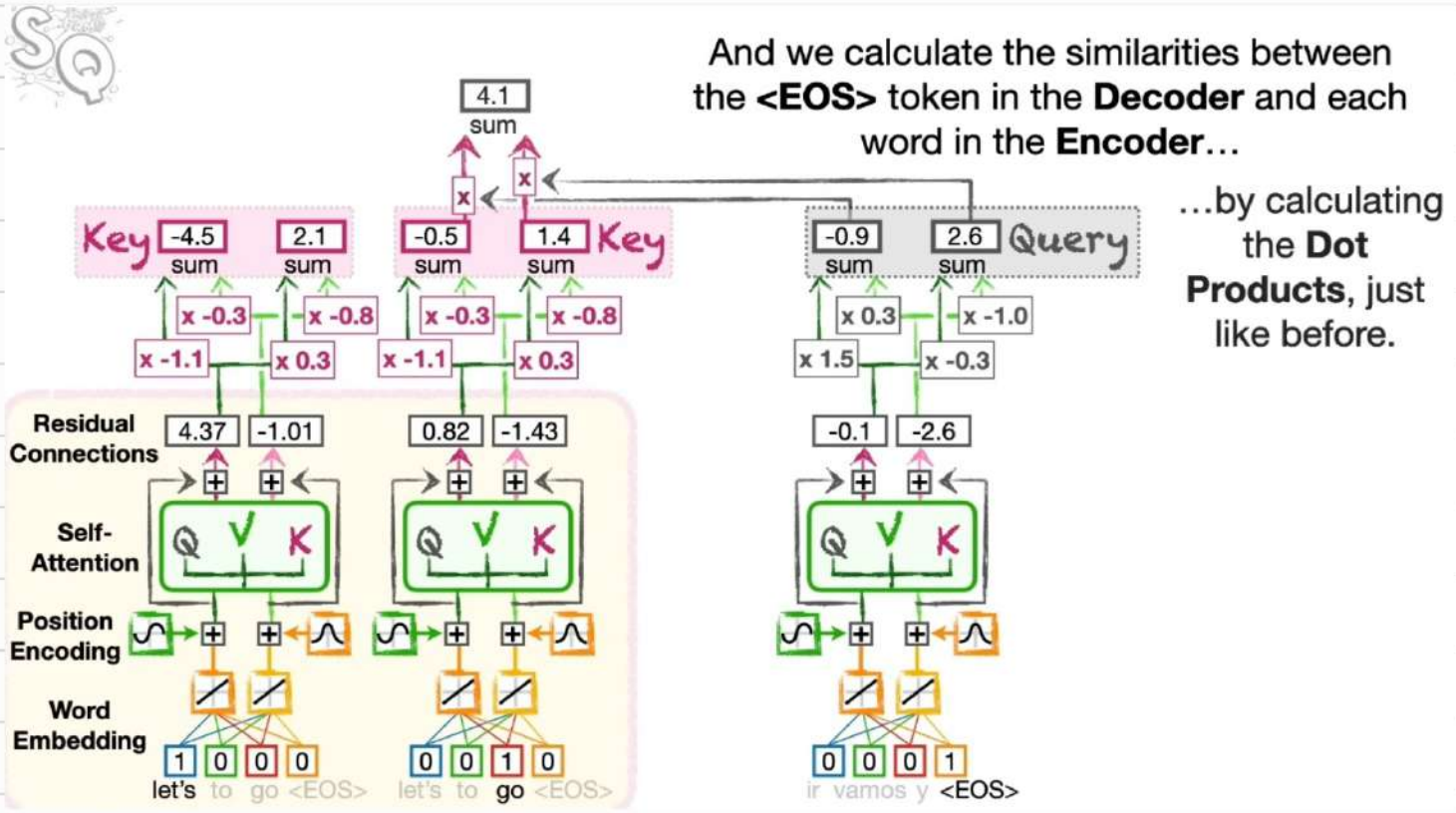
(otherwise we'll end end up with: Eat the delicious looking and smelling pizza)

⇒ It's so important of the decoder to keep track of the significant words in the input

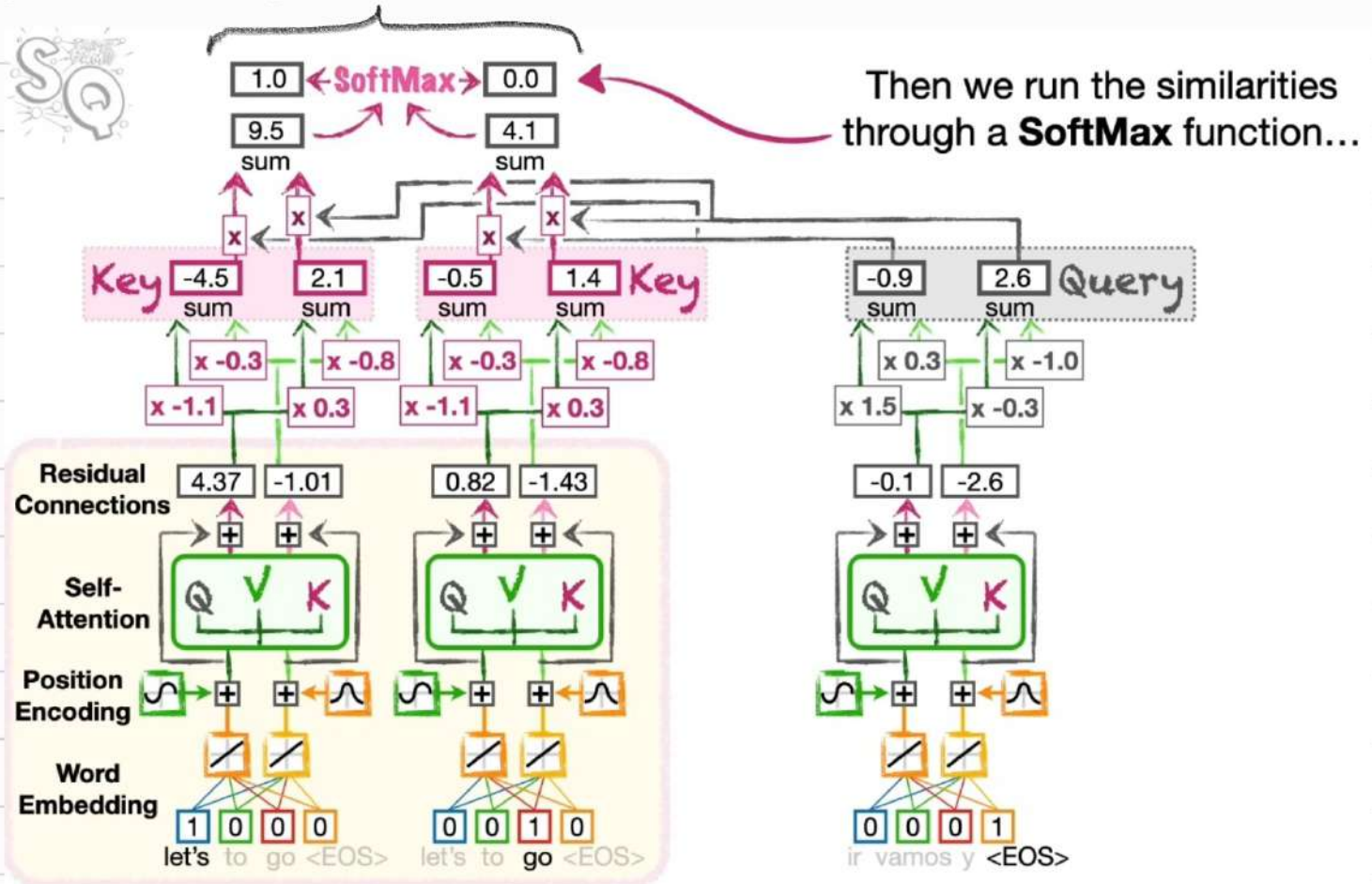
⇒ The main idea of the encoder-decoder attention is to allow the decoder to keep track of the significant words in the input



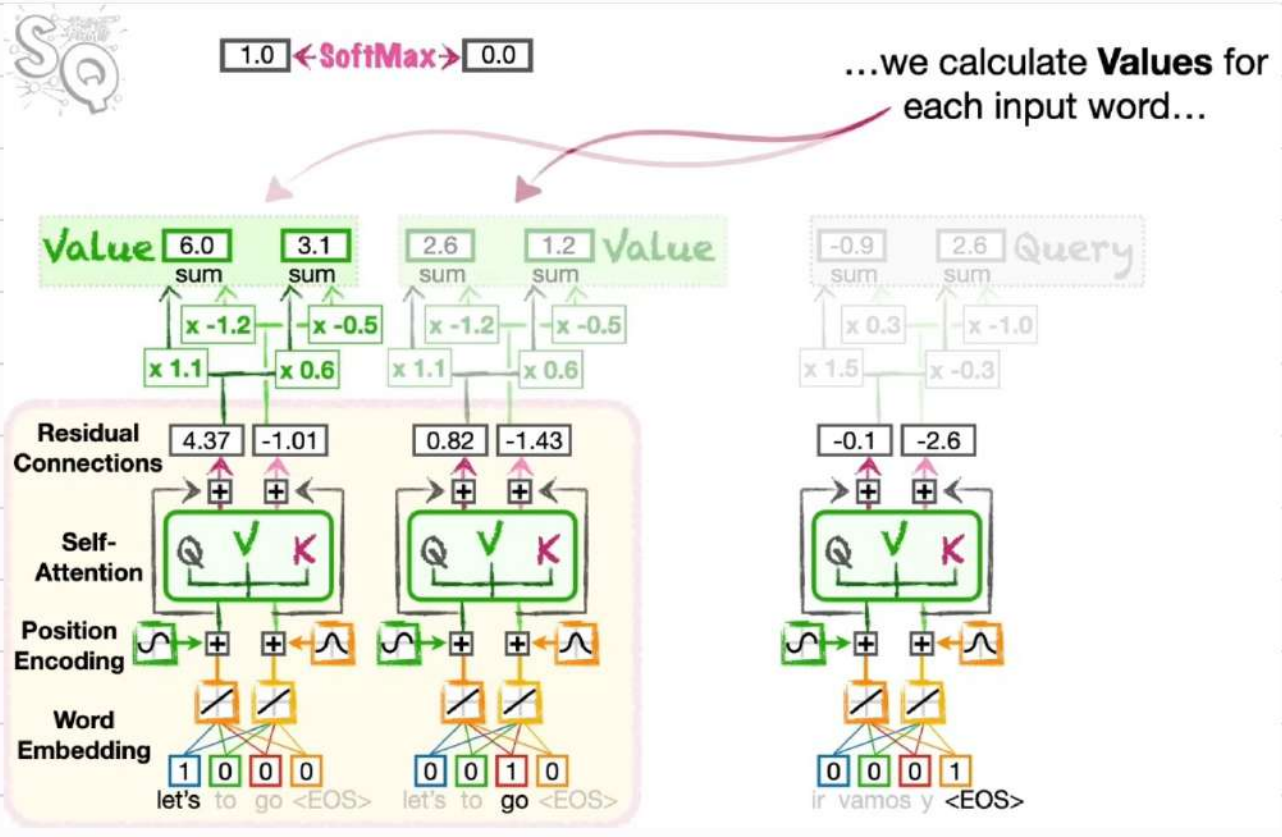


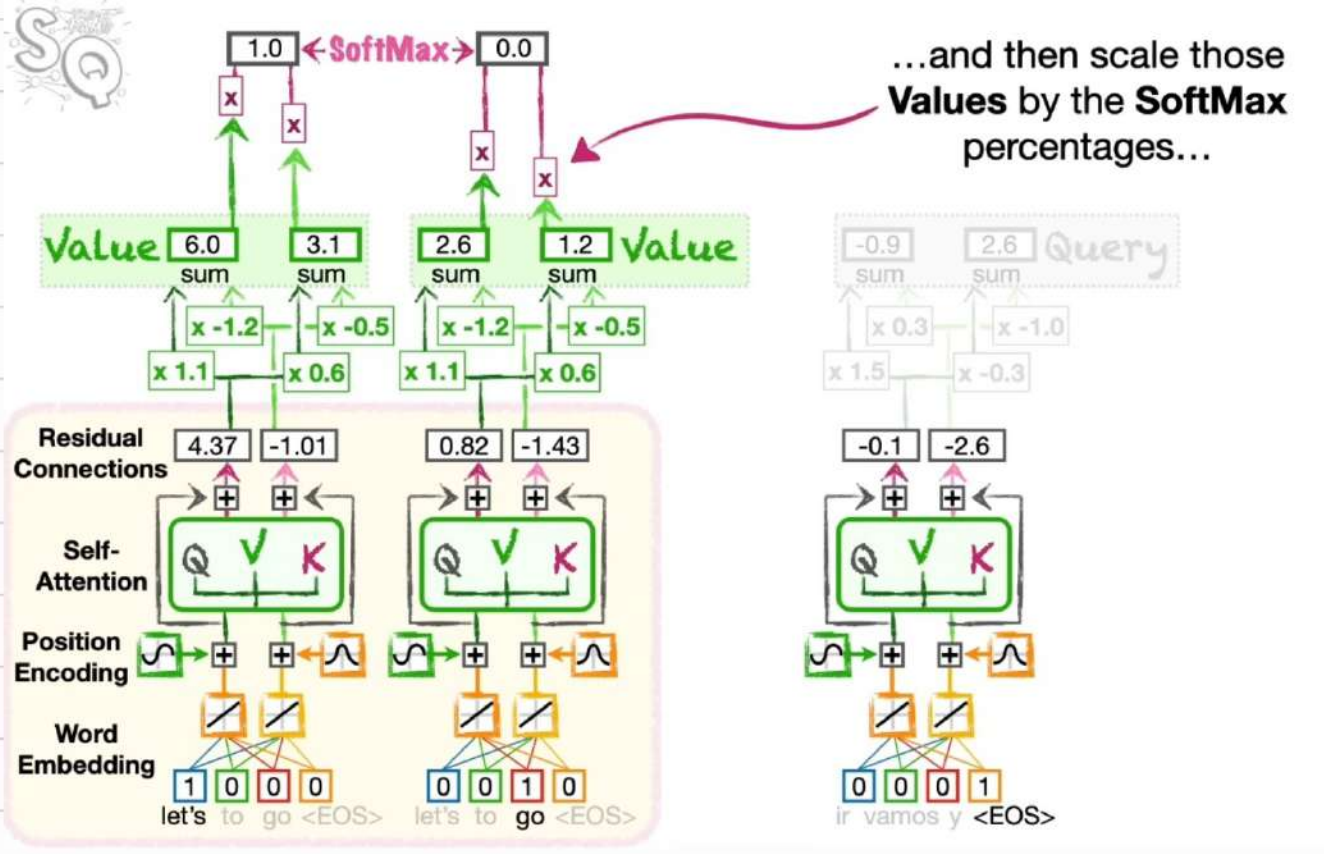


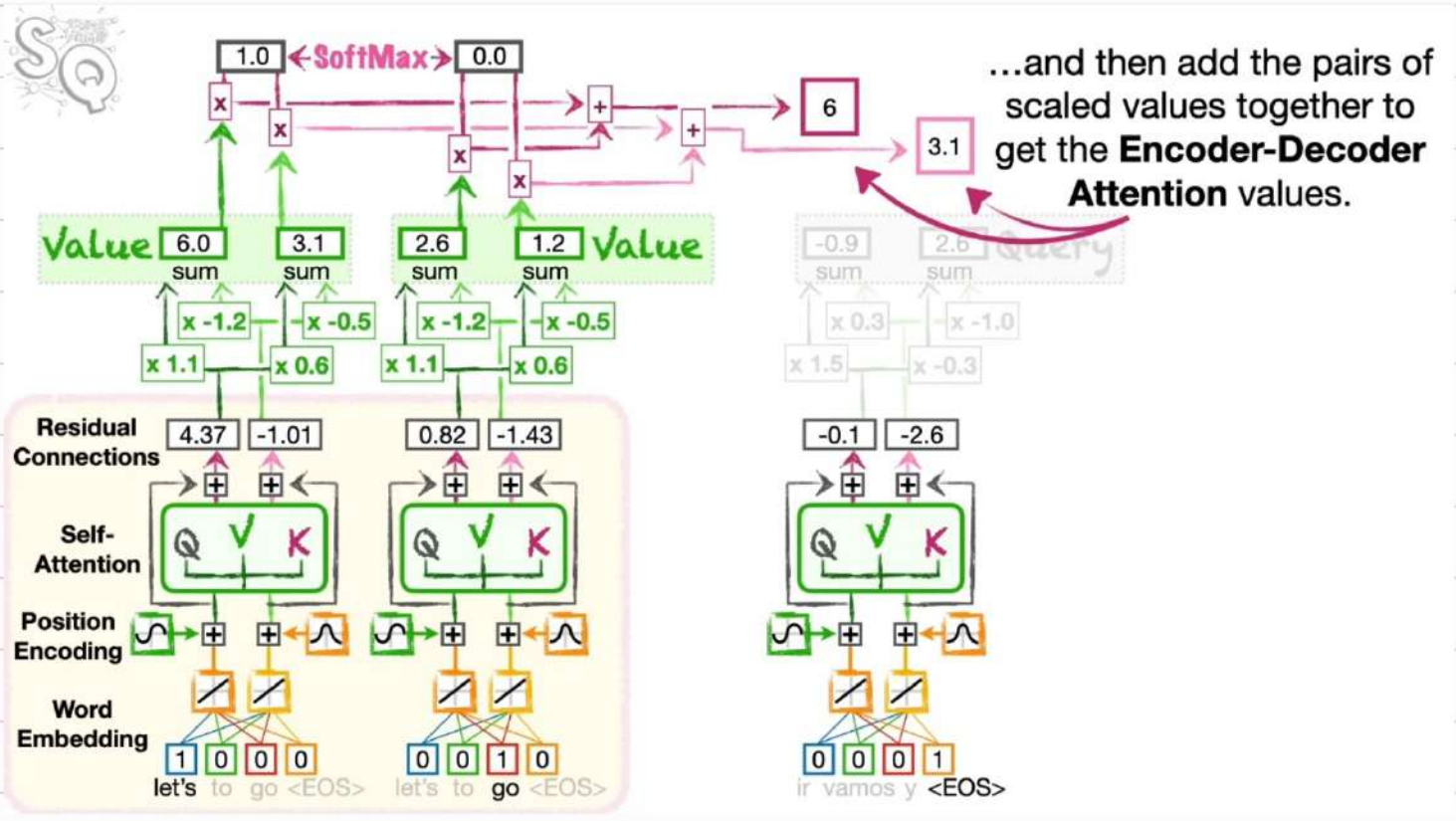
This tells us to use 100% of the first input word, and 0% of the second, when the decoder determines what should be the 1st translated word



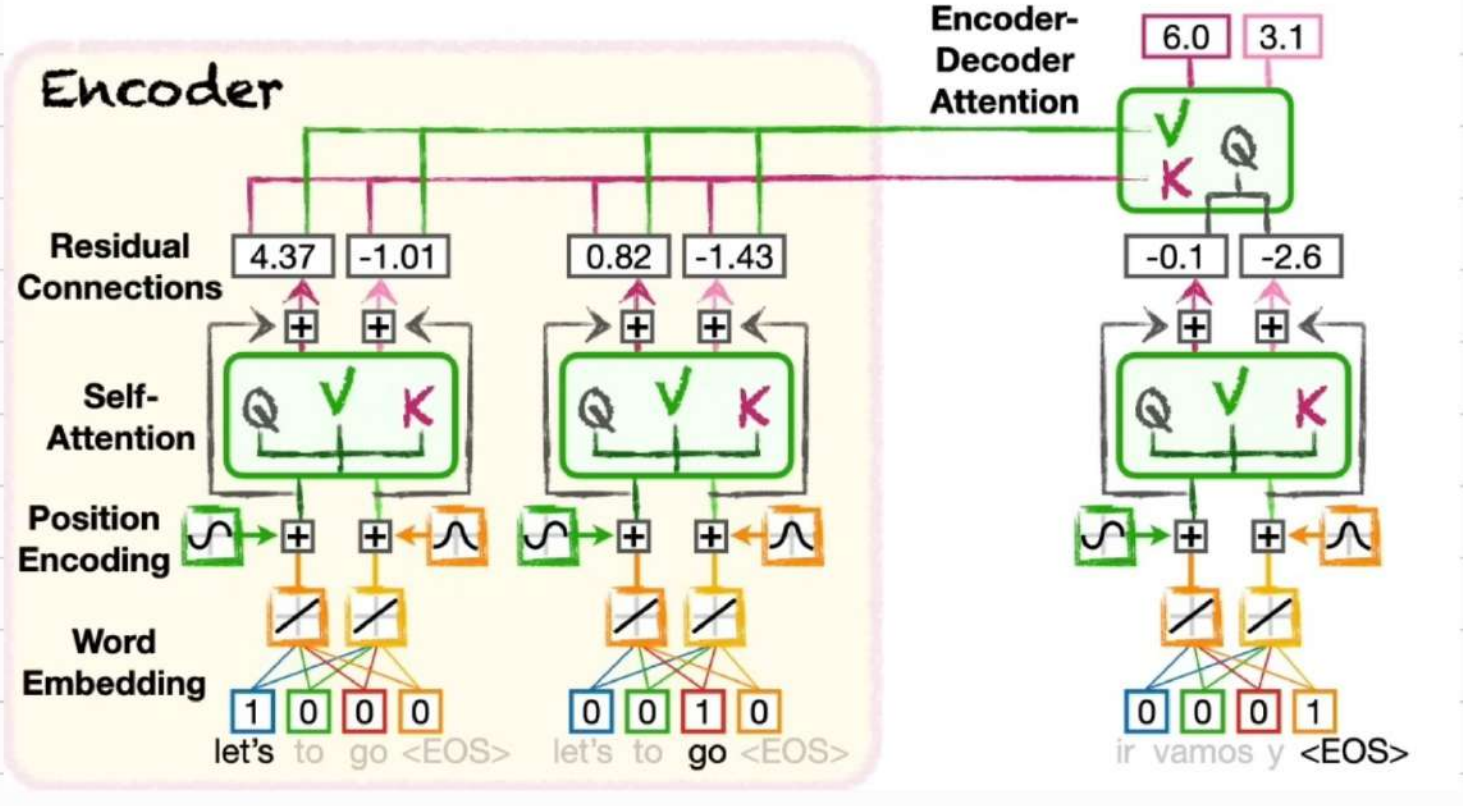
Now that we know what percentange of each input word to use when determining what should be the first translated word, we calculate values for each input word...





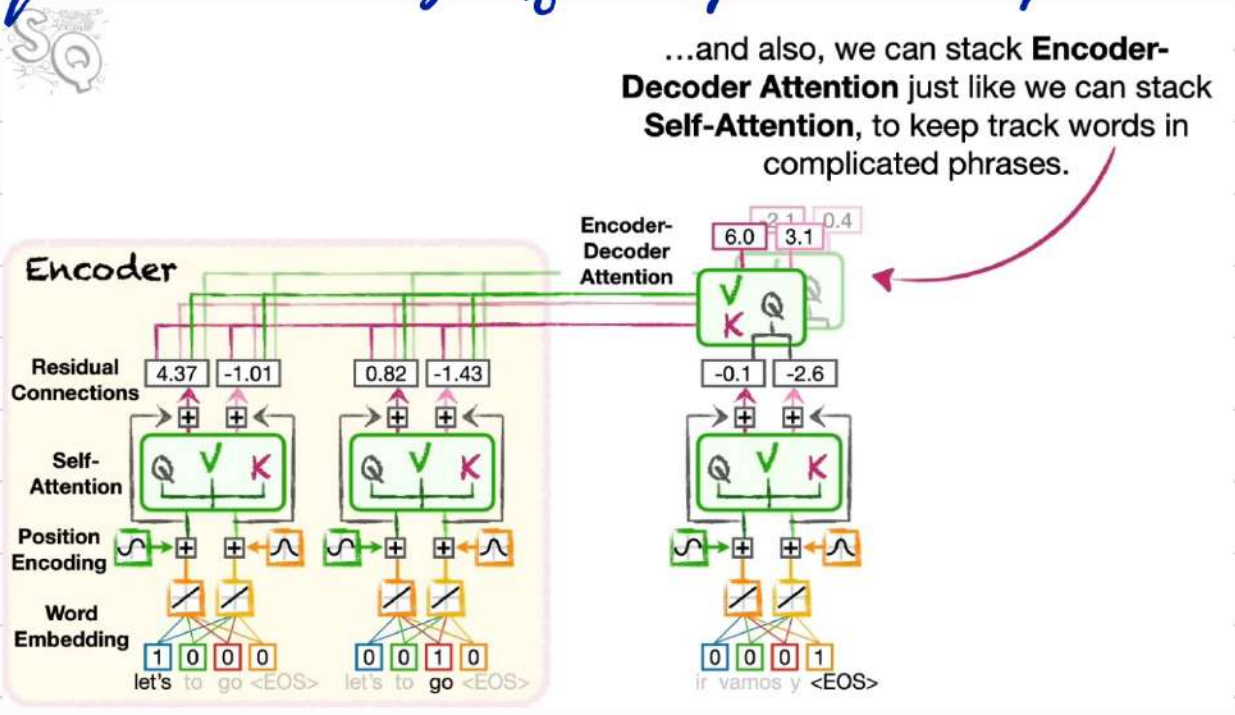


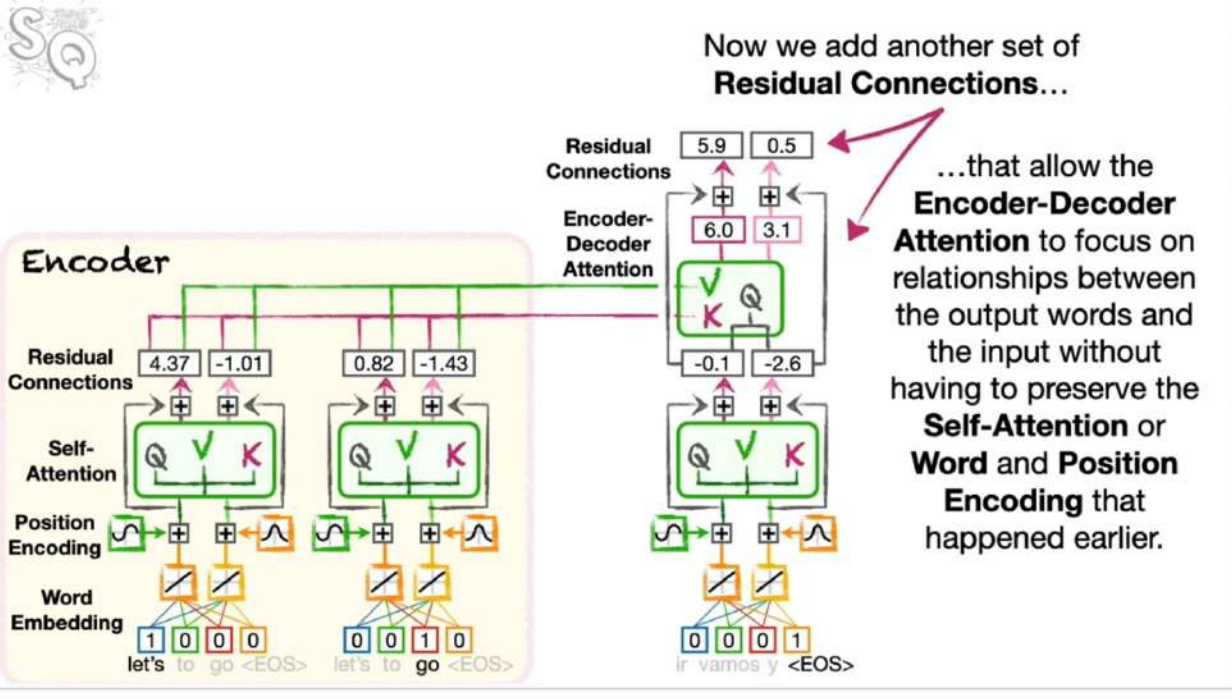
Consolidated encoder-decoder architecture:



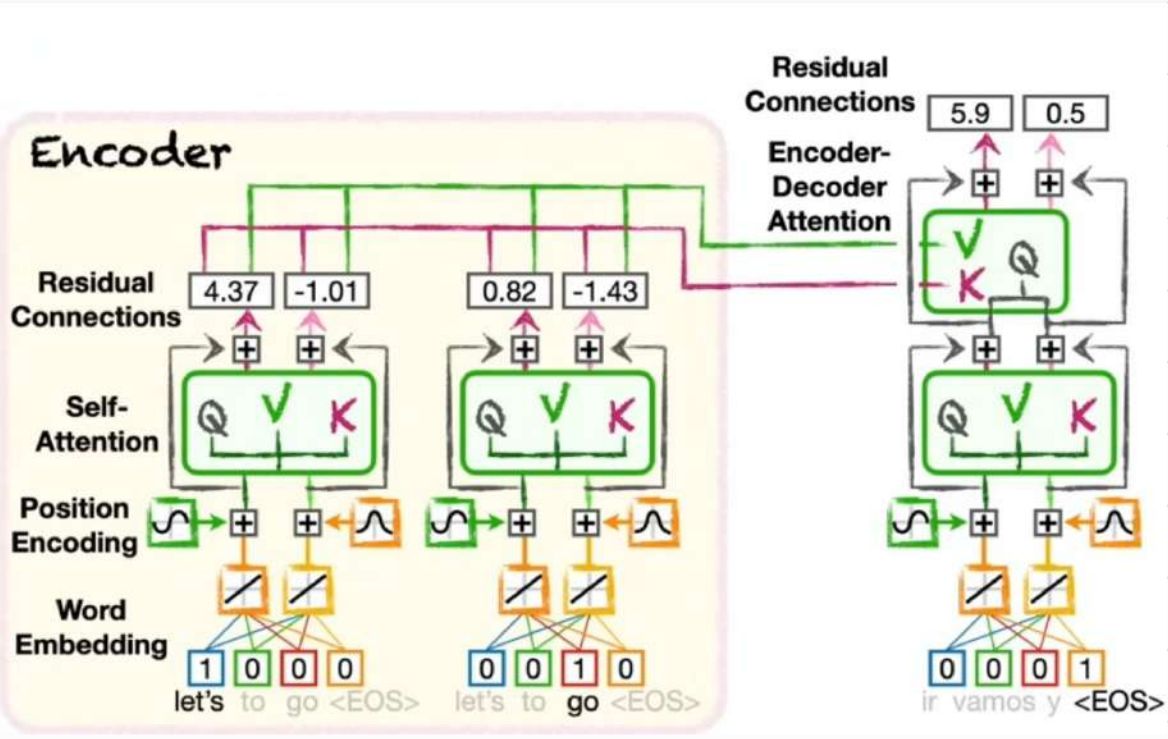
Note: The sets of weights that we use to calculate queries, keys and values for encoder-decoder attention are <u>different</u> from the sets of weights we use for self-attention.

However, just like for self-attention, the sets of weights are copied and reused for each word. This allows the transformer to be flexible with the length of the inputs and outputs.



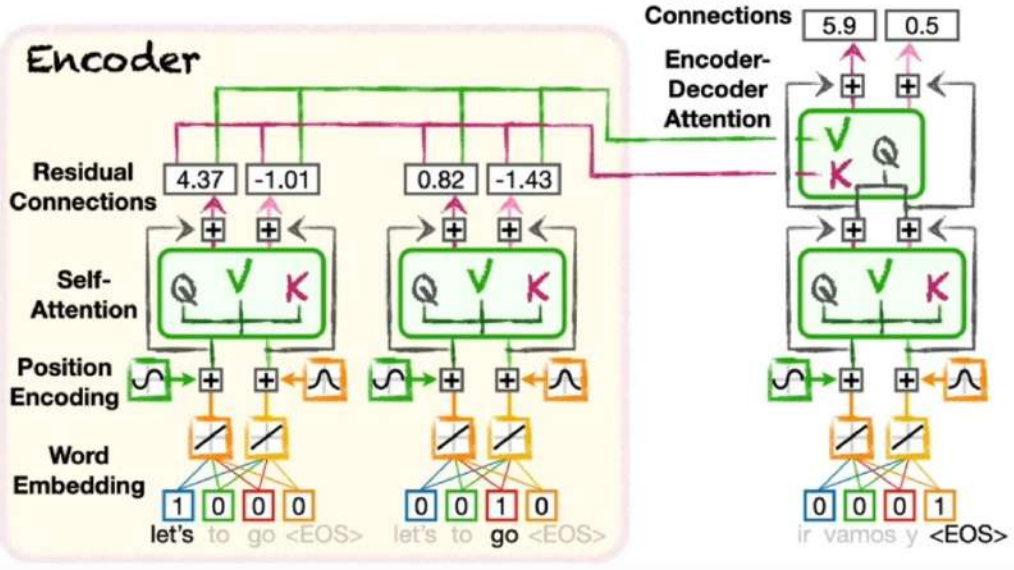


Consolidated architecture:

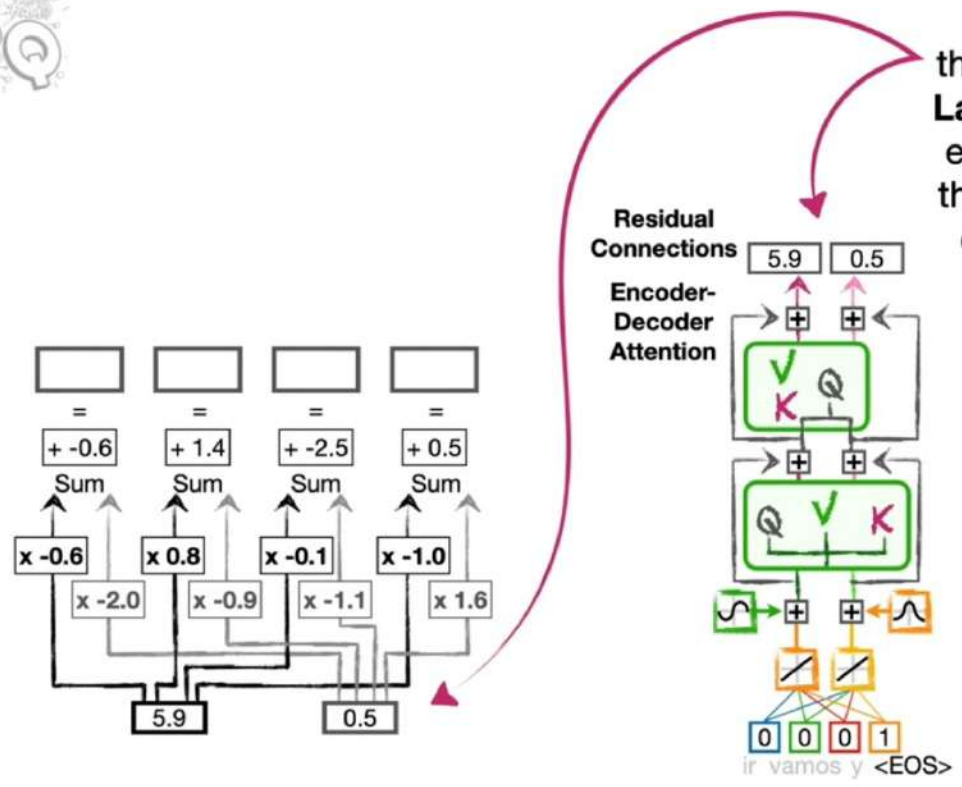


Lastly, we need a way to take these **2** values that represent the **<EOS>** token in the **Decoder**...

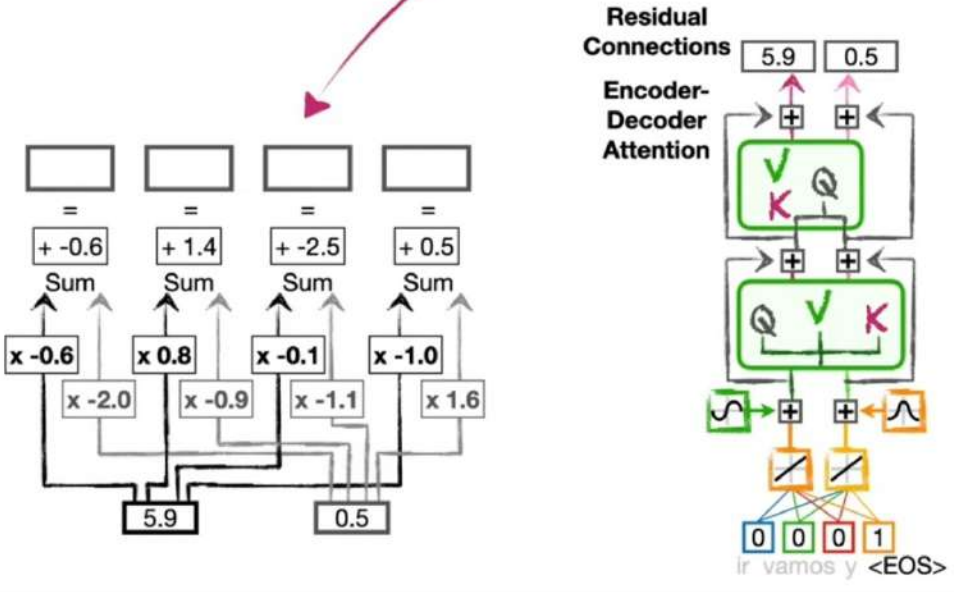
...and select one of the **4** output tokens, **ir**, **vamos**, **y** or **<EOS>**.



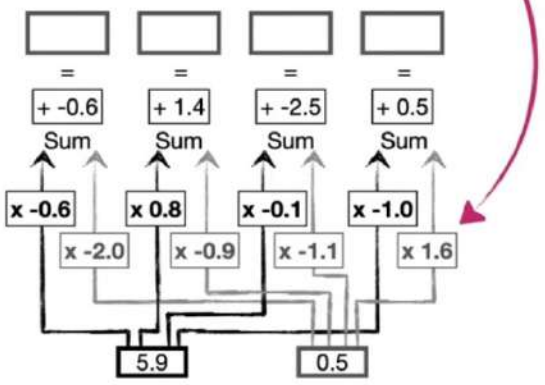
So we run the **2** values through a **Fully Connected Layer** that has one input for each value that represents the current token, so in this case we have **2** inputs…



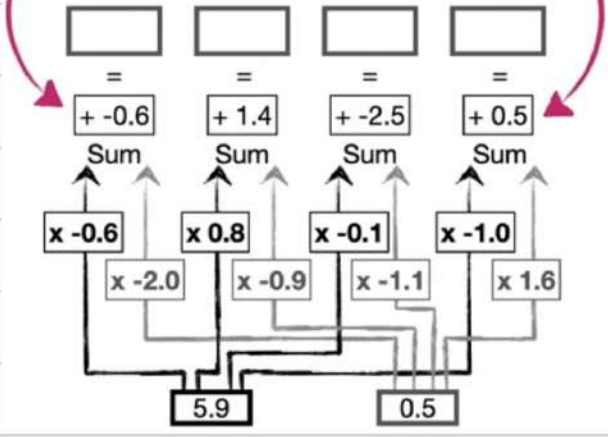
…and one output for each token in the output vocabulary, which in this case means **4** outputs.



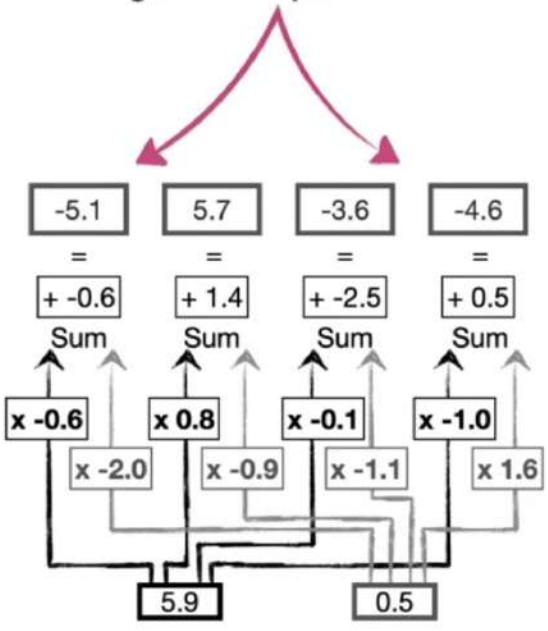
NOTE: A **Fully Connected Layer** is just a simple **Neural Network** with **Weights**, numbers we multiply the inputs by…

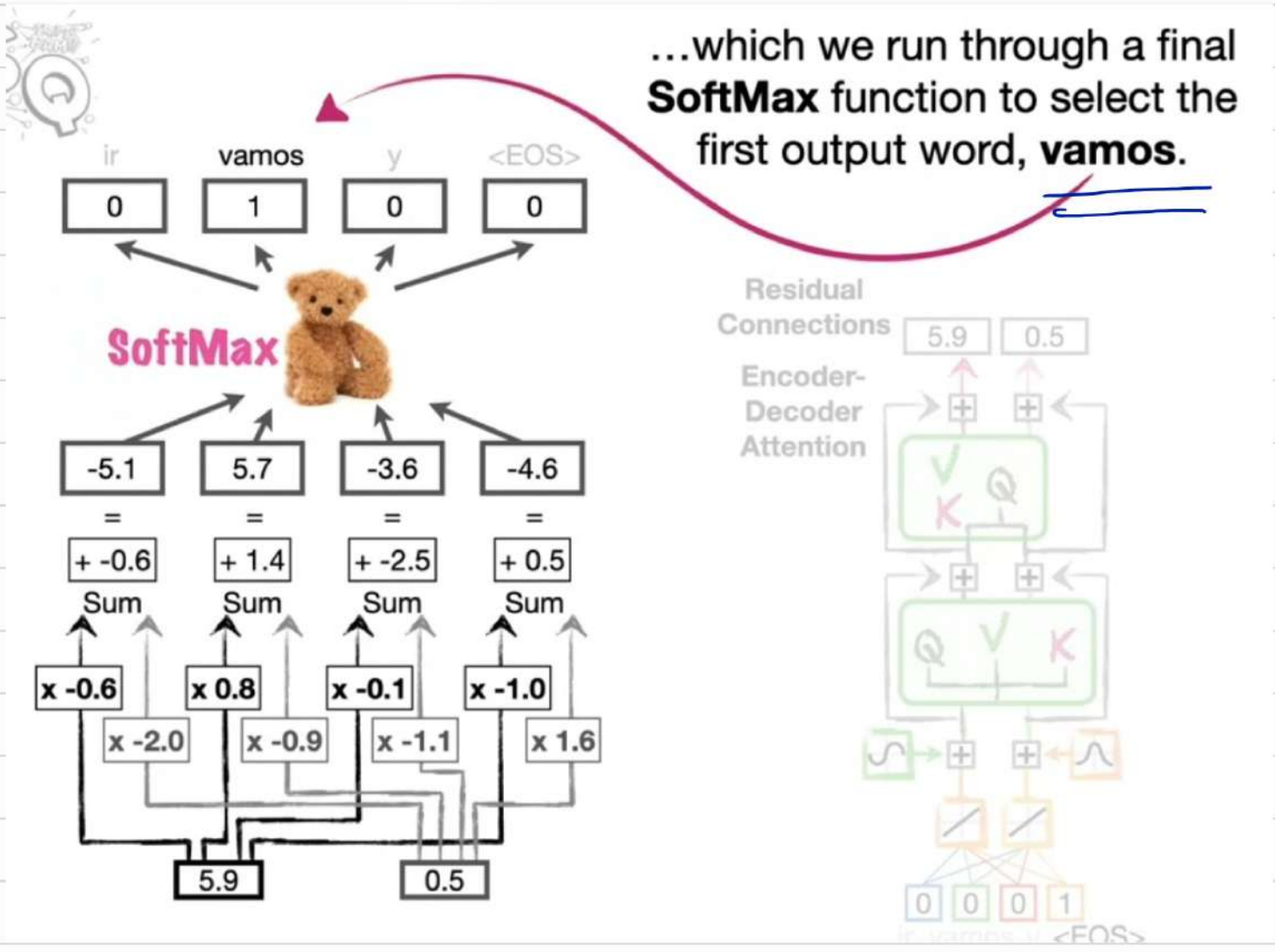


…and **Biases**, numbers we add to the sums of the products.



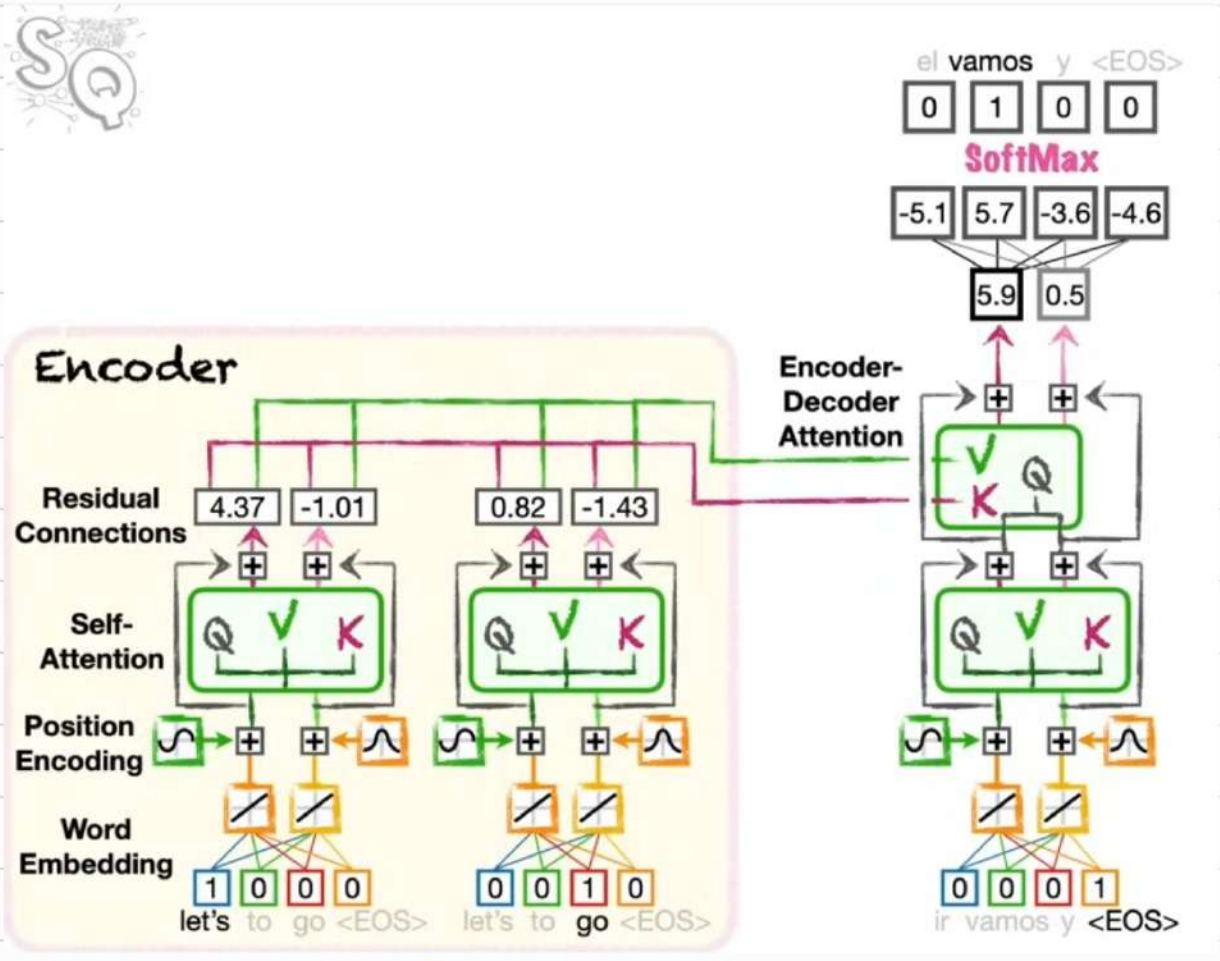
And when we do the math, we get **4** output values…

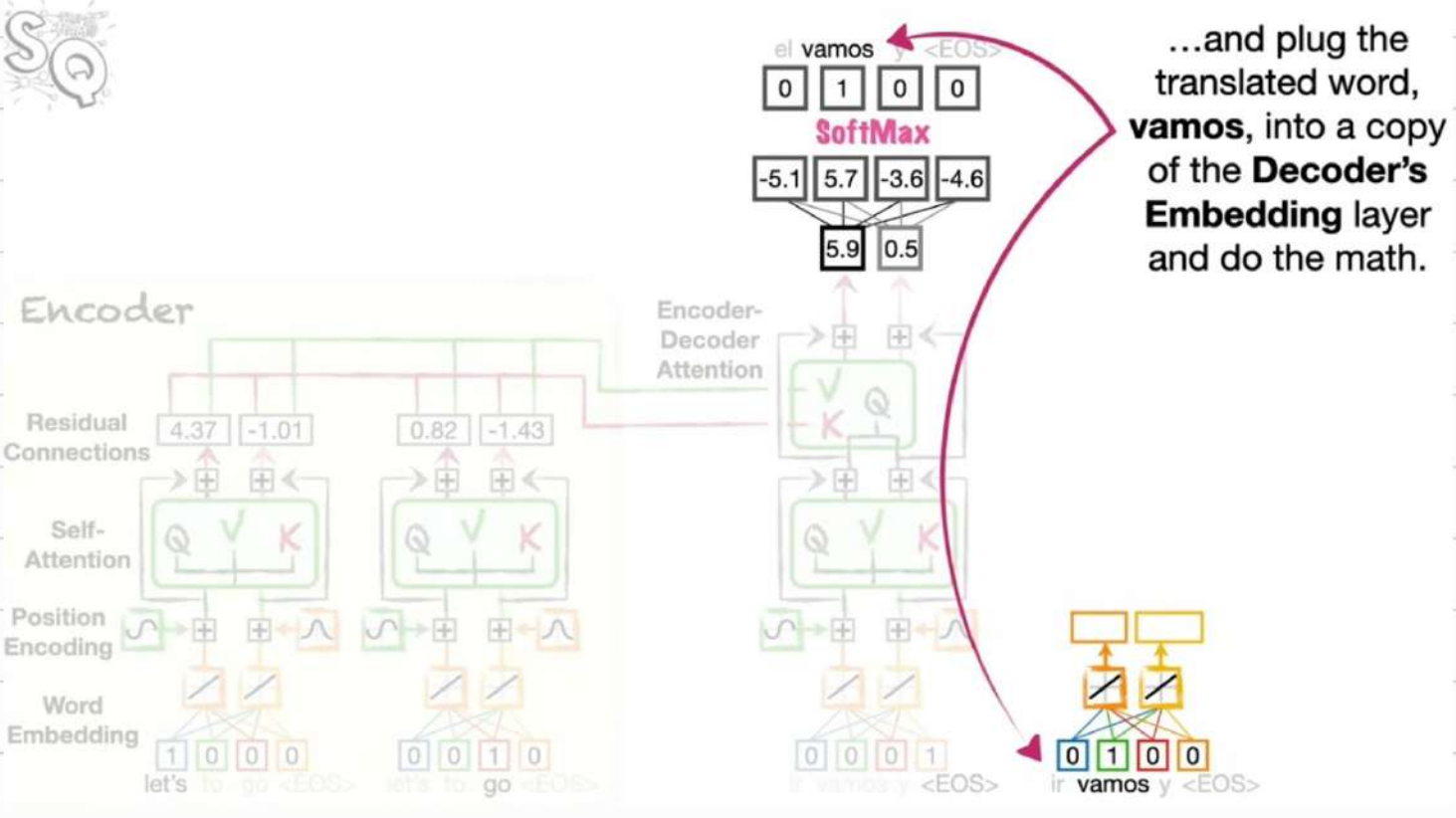




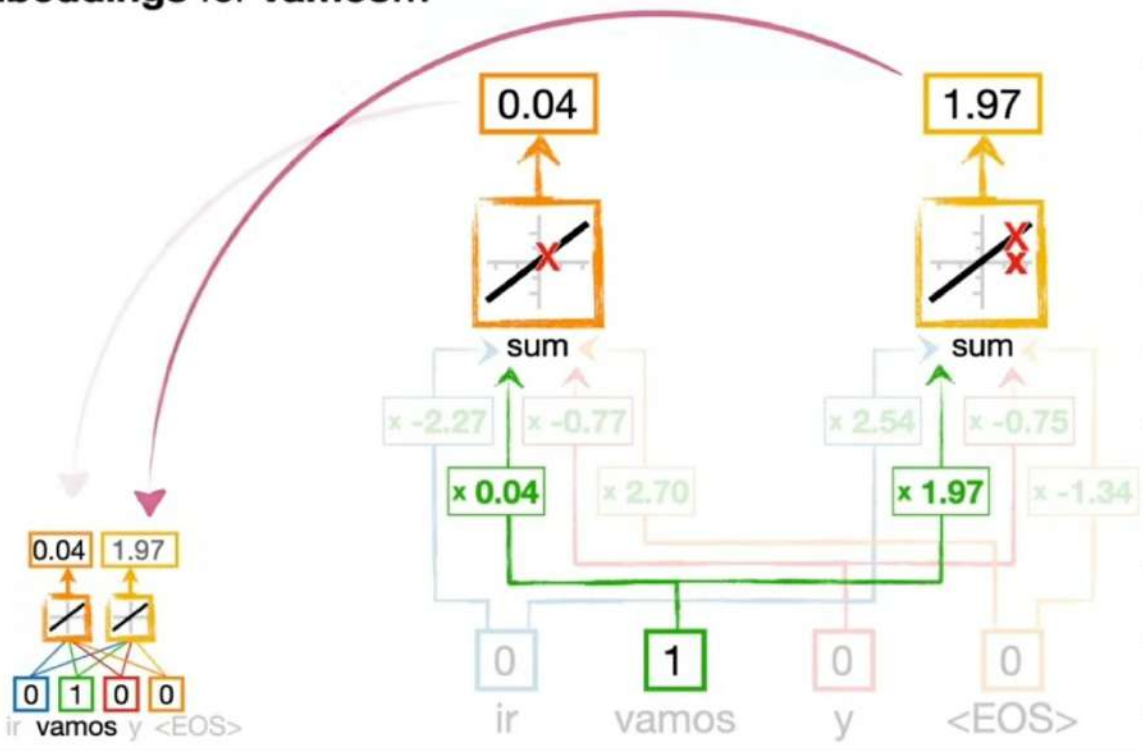
So far, the translation is correct, but the decoder don't stop until it outputs an <EOS> token

Consolidated architecture:

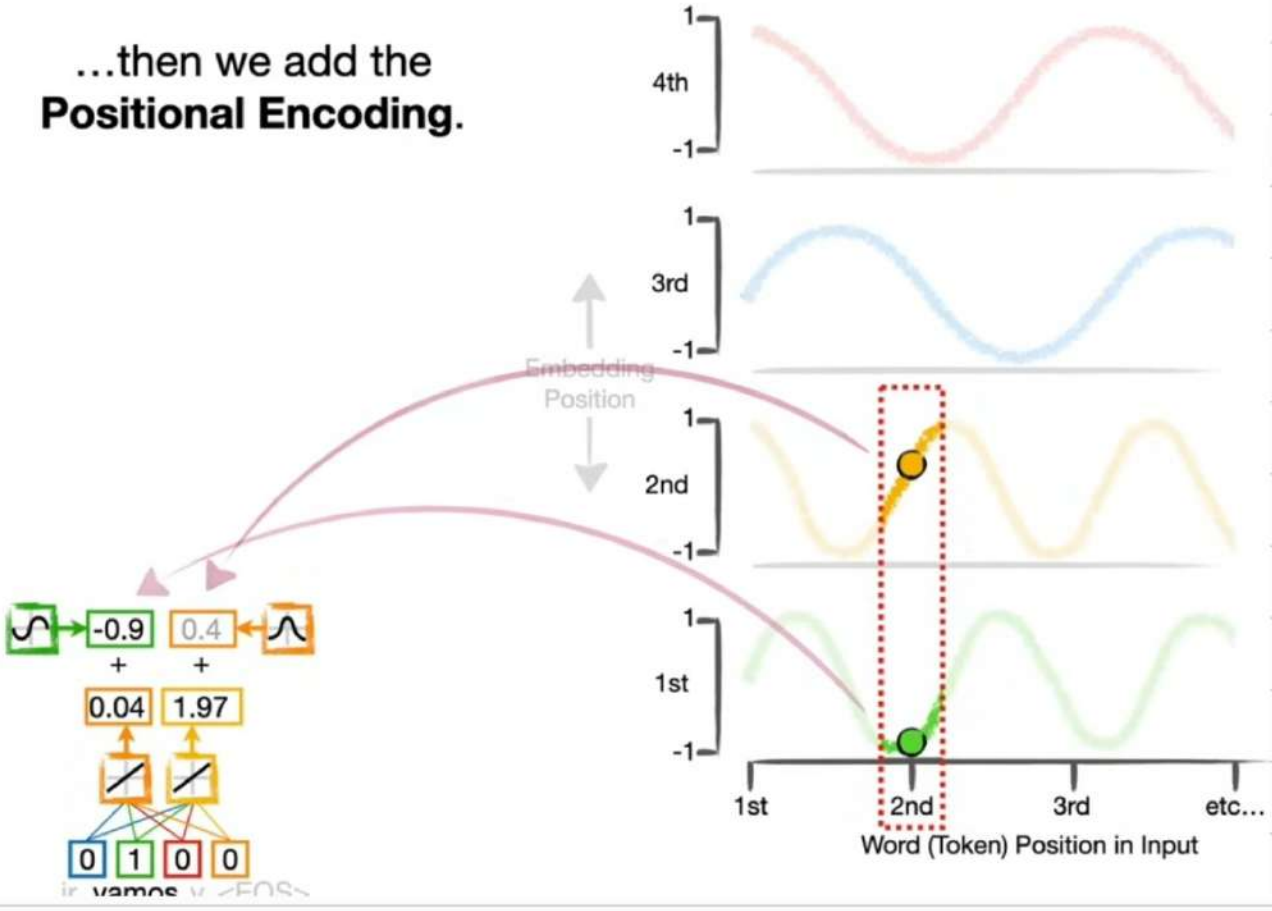

el vamos y <EOS>
0 1 0 0
SoftMax
-5.1 5.7 -3.6 -4.6
5.9 0.5
Encoder
Encoder-Decoder Attention
Residual Connections
4.37 -1.01 0.82 -1.43
Self-Attention
Q V K
Position Encoding
Word Embedding
1 0 0 0
let's to go <EOS>
0 0 1 0
let's to go
0 0 0 1
ir vamos y <EOS>


el vamos y <EOS>
0 1 0 0
SoftMax
-5.1 5.7 -3.6 -4.6
5.9 0.5
…and plug the translated word, **vamos**, into a copy of the **Decoder's Embedding** layer and do the math.
Encoder
Encoder-Decoder Attention
Residual Connections
4.37 -1.01 0.82 -1.43
Self-Attention
Q V K
Position Encoding
Word Embedding
1 0 0 0
let's to go <EOS>
0 0 1 0
go
0 0 0 1
ir vamos y <EOS>
0 1 0 0
ir vamos y <EOS>

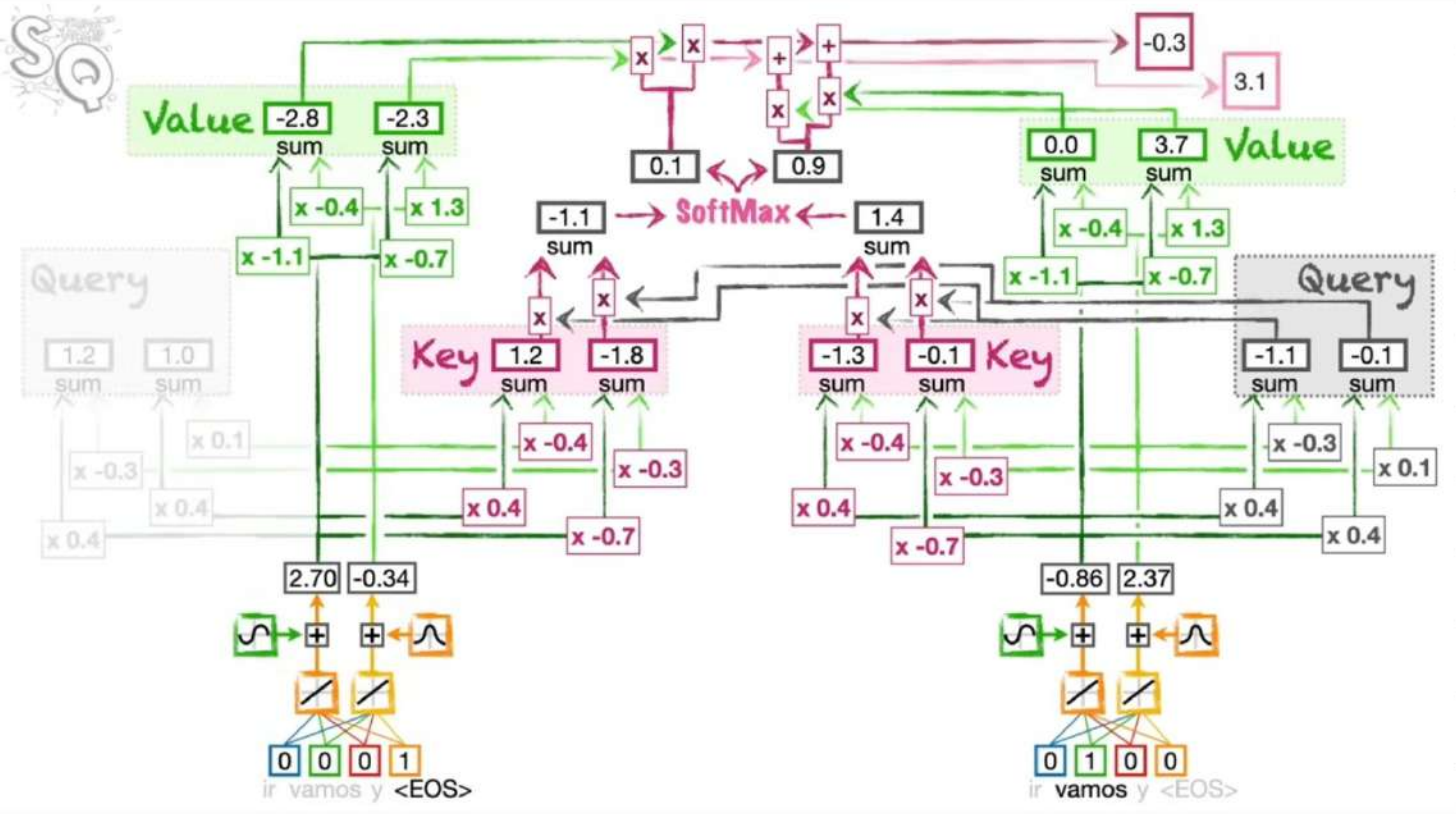
First, we get the **Word Embeddings** for **vamos**...



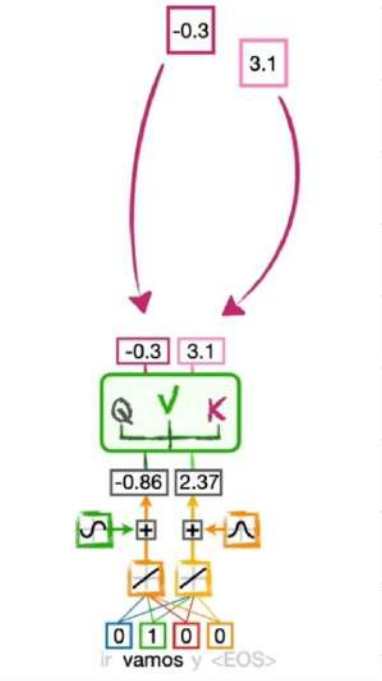
...then we add the **Positional Encoding**.

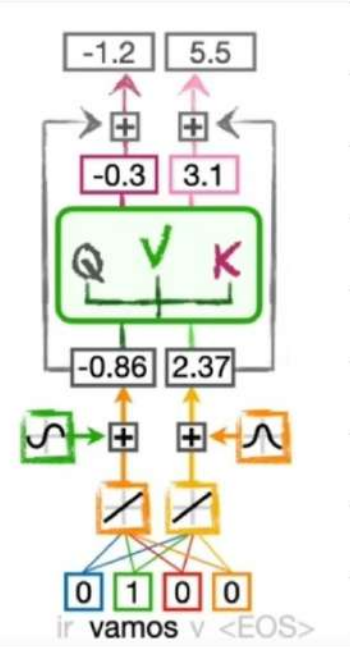


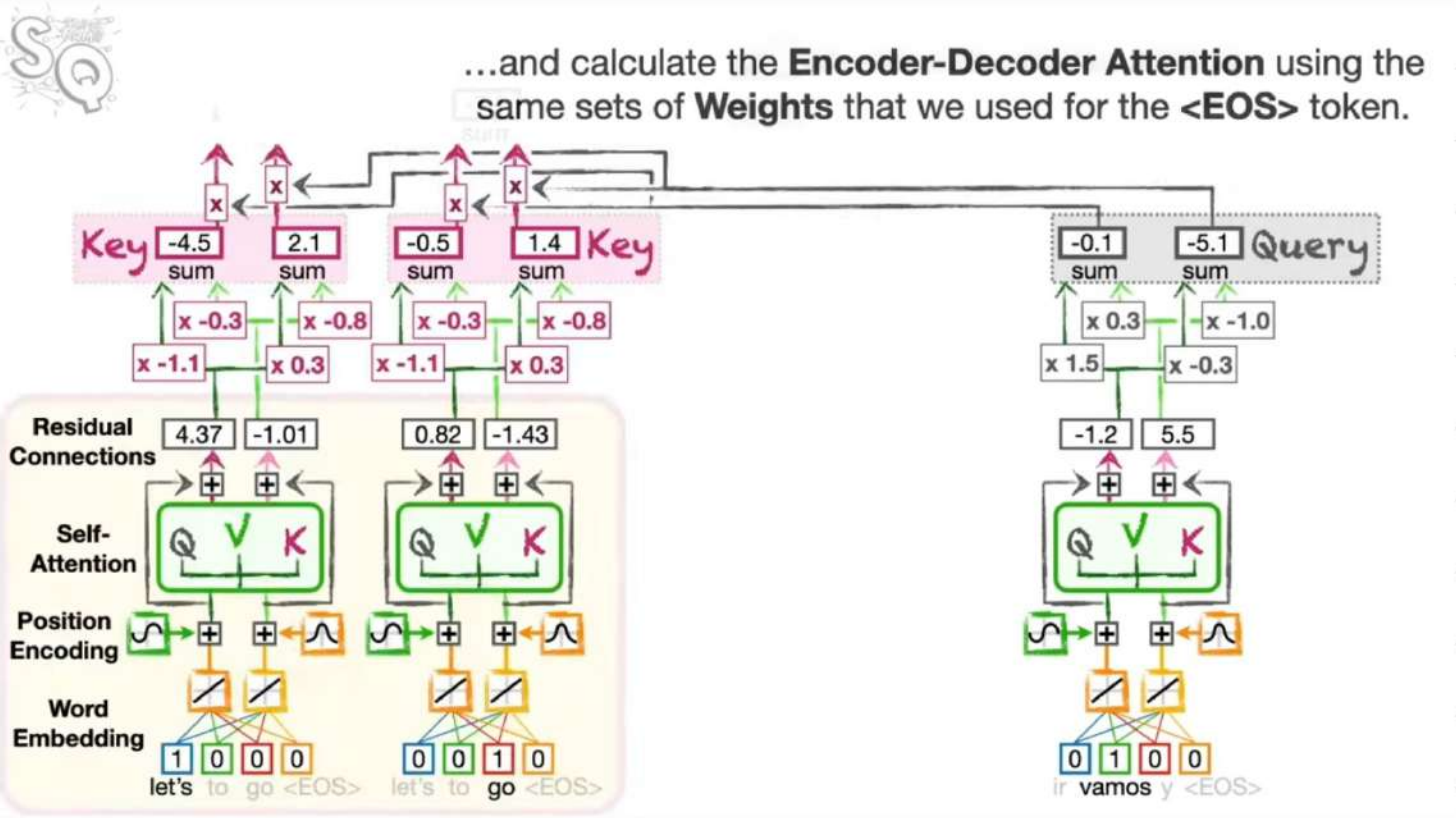
Now we calculate Self-Attention values for vamos using the exact same weights that we used for the <EOS> token.

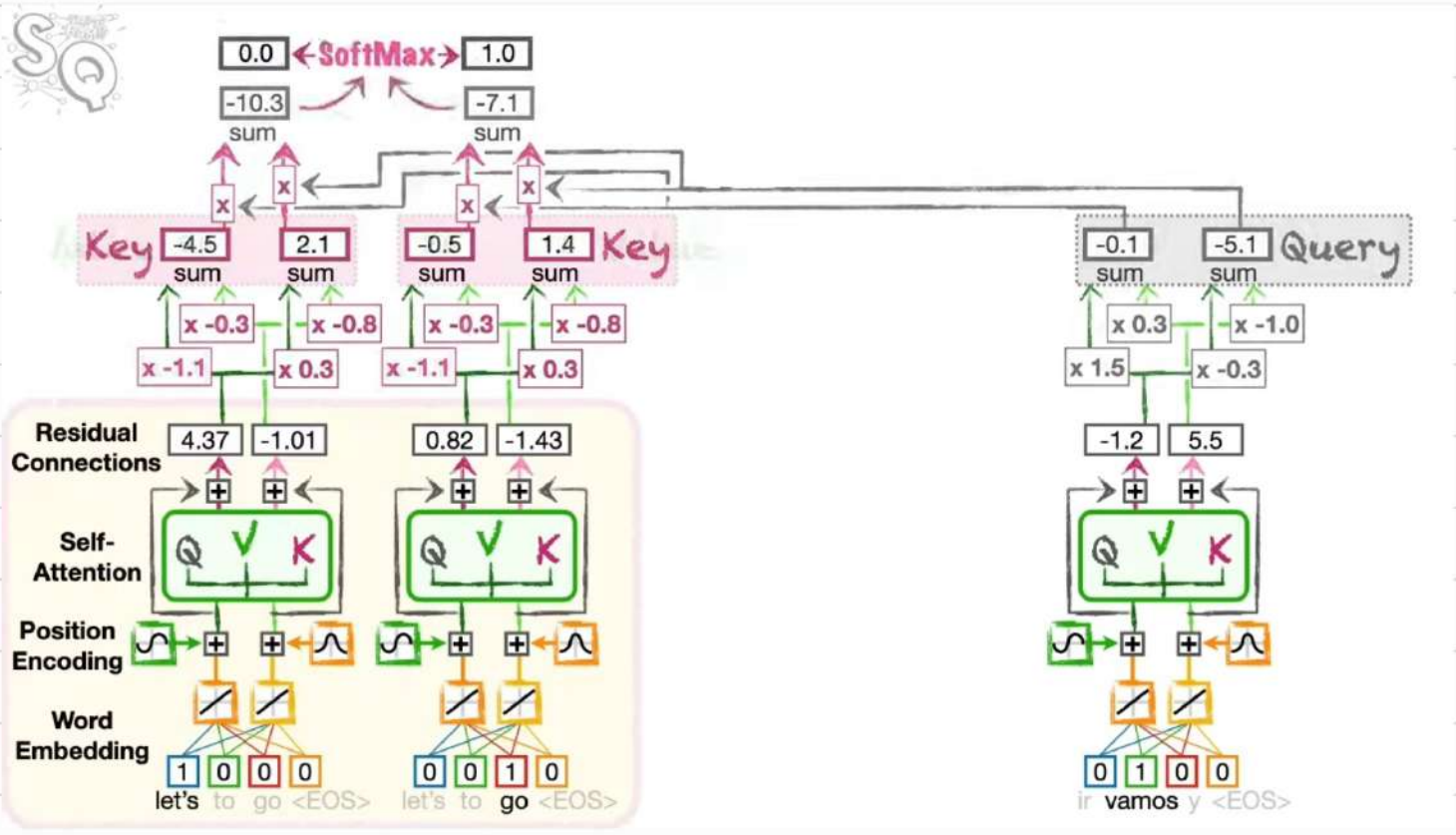


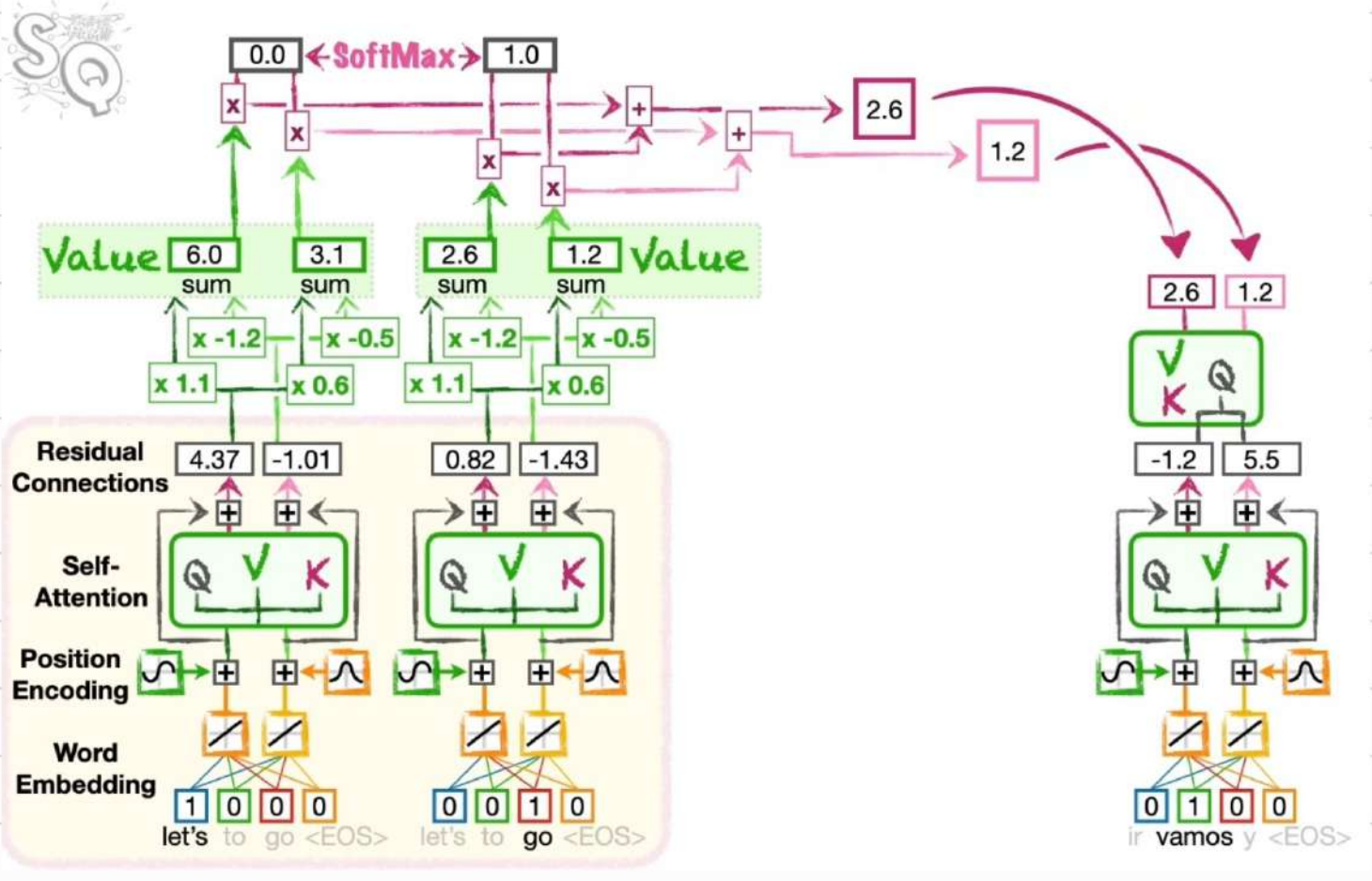
Now we add residual connections:

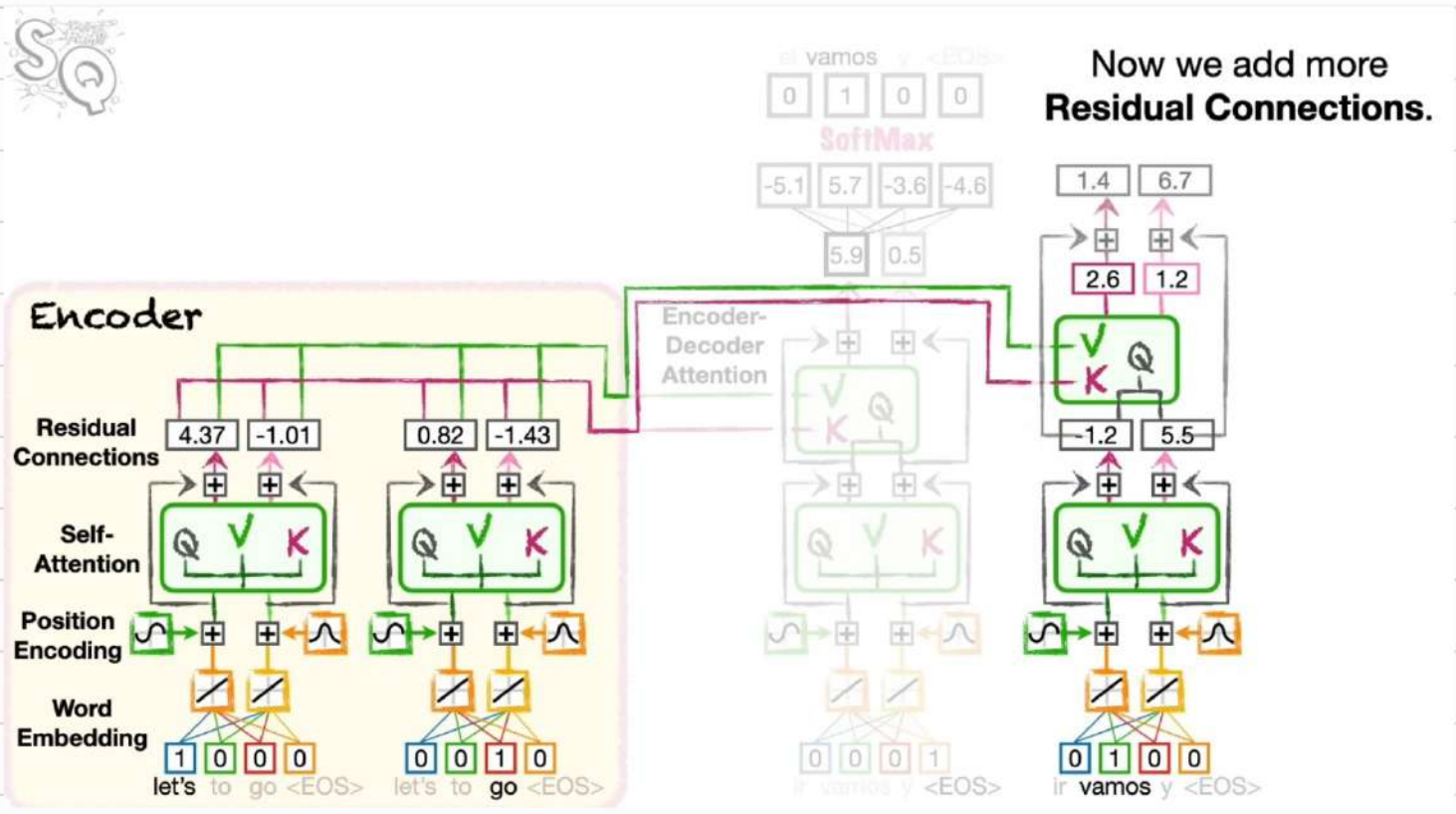



-1.2
5.5
-0.3
3.1
Q
V
K
-0.86
2.37
0
1
0
0
ir
vamos
v
<EOS>

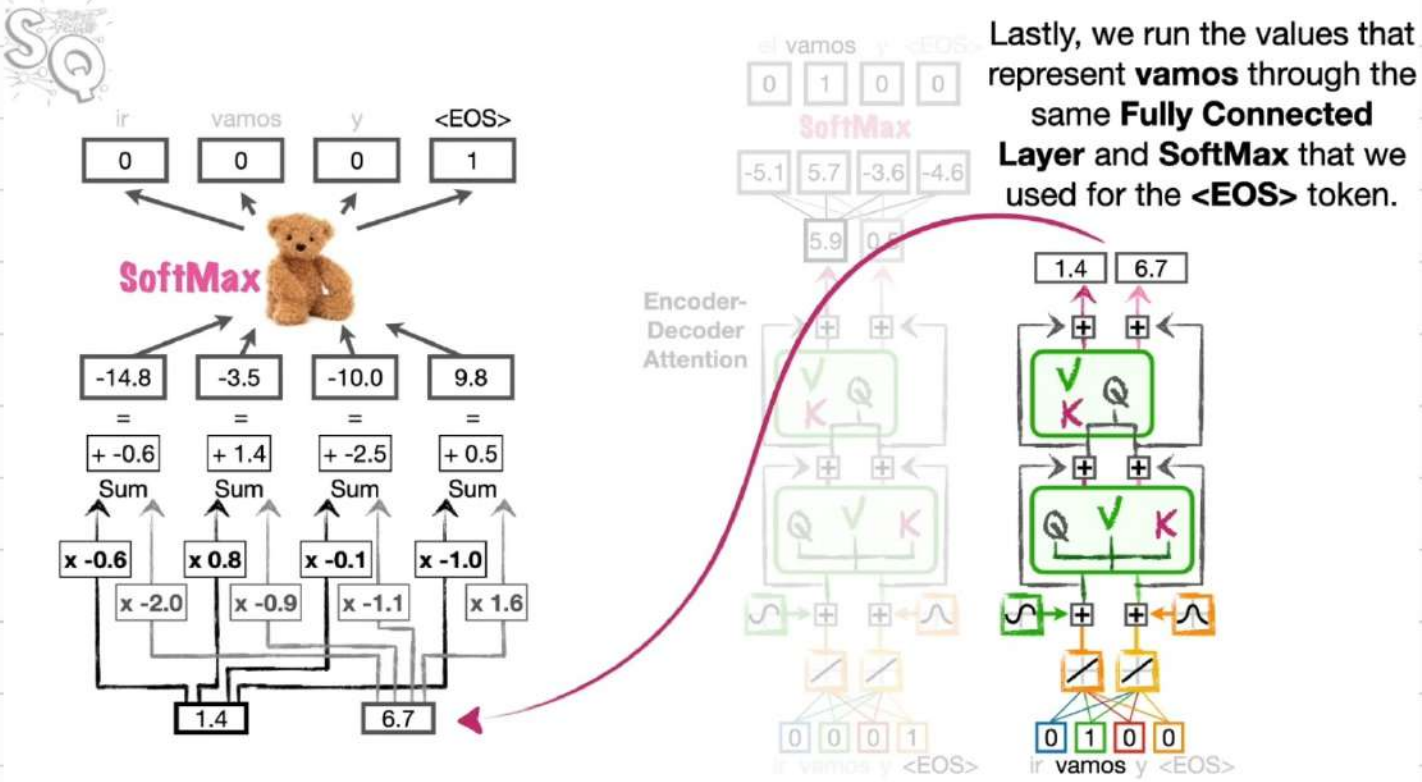

...and calculate the Encoder-Decoder Attention using the same sets of Weights that we used for the <EOS> token.
Key
-4.5
2.1
-0.5
1.4
Key
-0.1
-5.1
Query
sum
x -0.3
x -0.8
x -1.1
x 0.3
x 0.3
x -1.0
x 1.5
x -0.3
Residual Connections
4.37
-1.01
0.82
-1.43
-1.2
5.5
Self-Attention
Position Encoding
Word Embedding
1
0
0
0
let's
to
go
<EOS>
0
1
0
ir
vamos
y

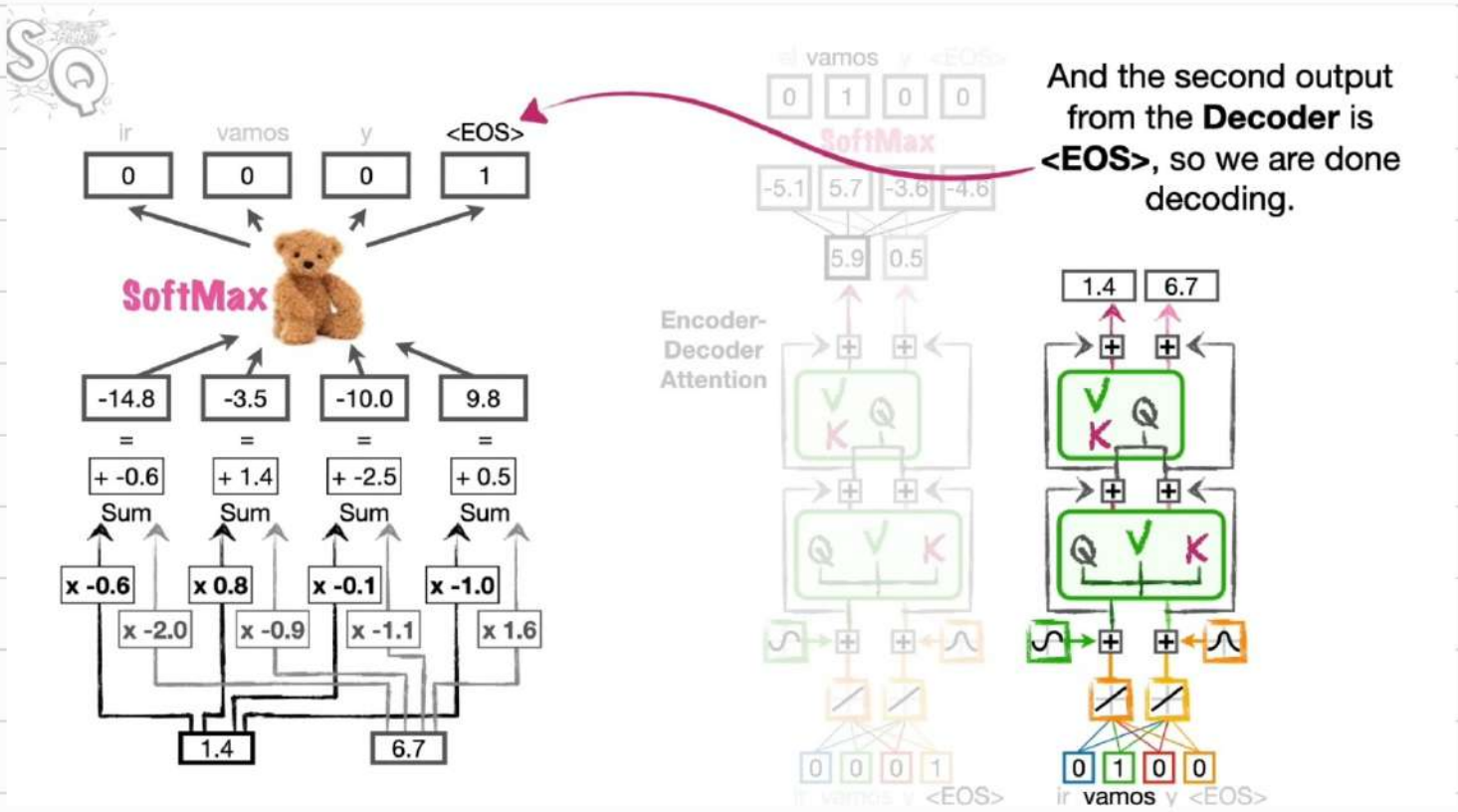

0.0
SoftMax
1.0
-10.3
-7.1
sum
sum
Key
-4.5
2.1
-0.5
1.4
Key
-0.1
-5.1
Query
x -0.3
x -0.8
x -1.1
x 0.3
x 0.3
x -1.0
x 1.5
x -0.3
Residual Connections
4.37
-1.01
0.82
-1.43
-1.2
5.5
Self-Attention
Q
V
K
Position Encoding
Word Embedding
let's to go <EOS>
let's to go <EOS>
ir vamos y <EOS>

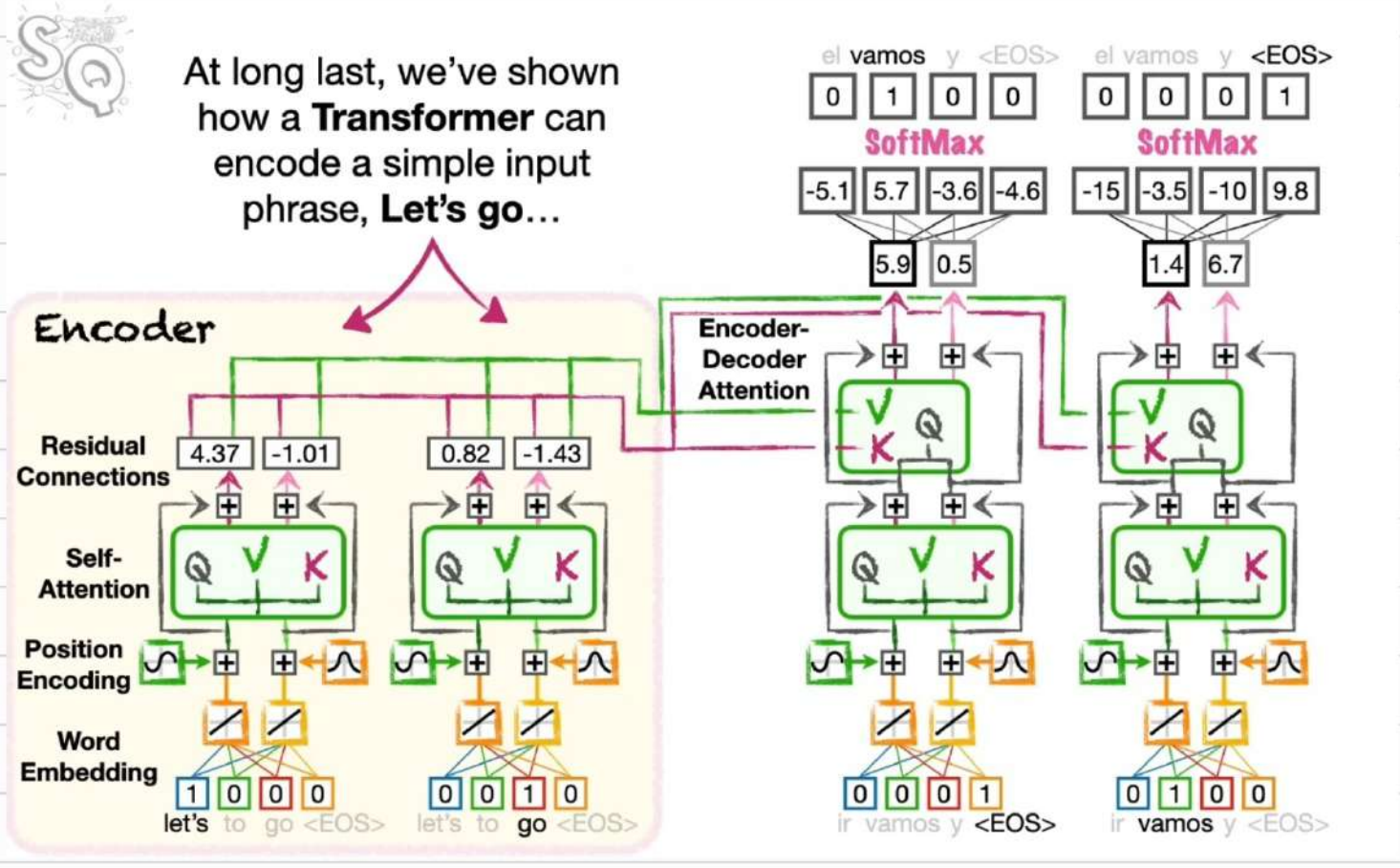

0.0
SoftMax
1.0
2.6
1.2
Value
6.0
3.1
2.6
1.2
Value
x -1.2
x -0.5
x 1.1
x 0.6
Residual Connections
4.37
-1.01
0.82
-1.43
-1.2
5.5
Self-Attention
Position Encoding
Word Embedding
let's to go <EOS>
let's to go <EOS>
ir vamos y <EOS>

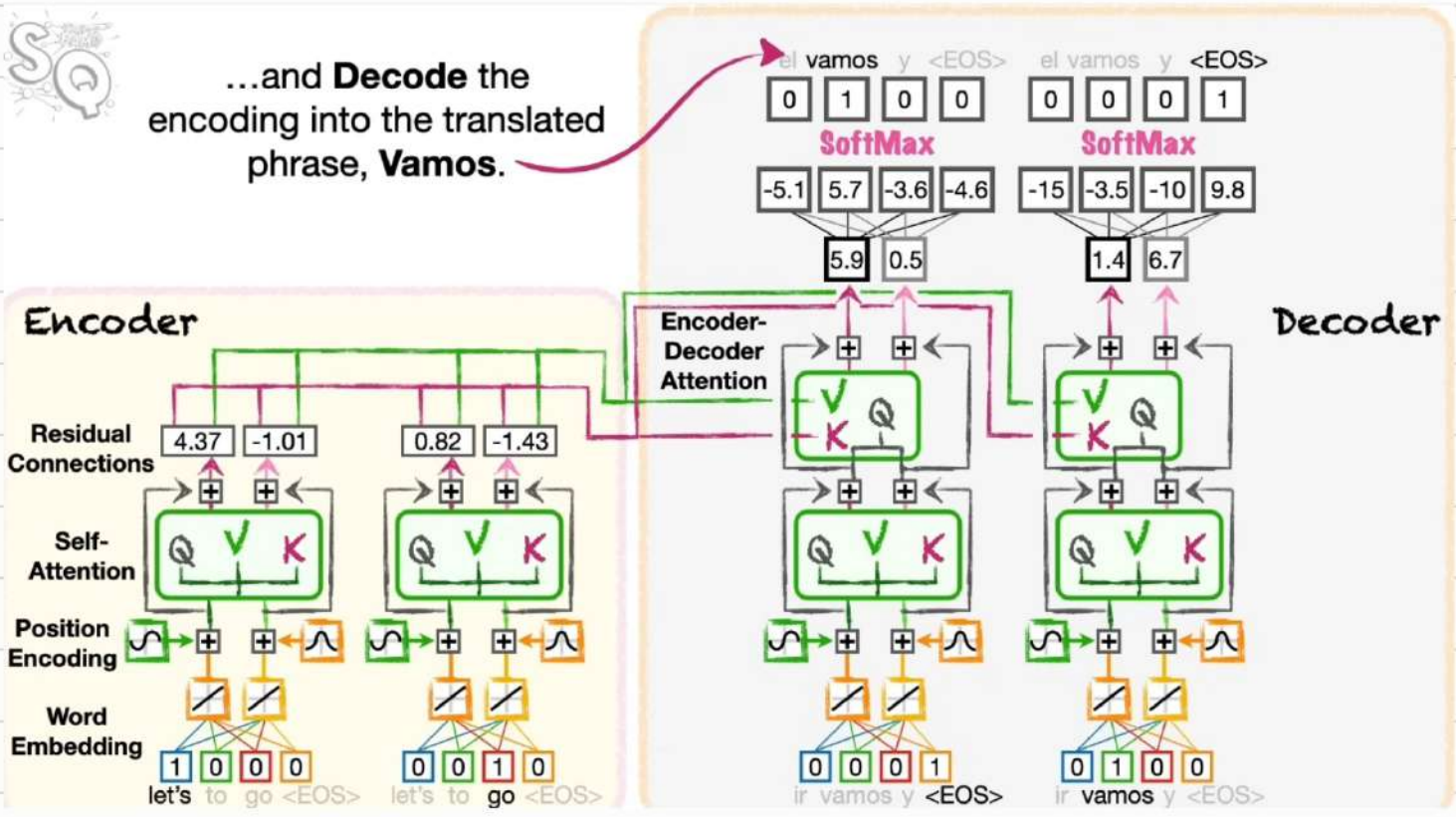


Lastly, we run the values that represent vamos through the same Fully Connected Layer and Soft Max that we used for the <EOS> token




ir vamos y <EOS>
0 0 0 1
SoftMax
-14.8 -3.5 -10.0 9.8
= = = =
+ -0.6 + 1.4 + -2.5 + 0.5
Sum Sum Sum Sum
x -0.6 x 0.8 x -0.1 x -1.0
x -2.0 x -0.9 x -1.1 x 1.6
1.4 6.7
And the second output from the **Decoder** is **<EOS>**, so we are done decoding.
el vamos y <EOS>
0 1 0 0
SoftMax
-5.1 5.7 -3.6 -4.6
5.9 0.5
Encoder-Decoder Attention
1.4 6.7
0 0 0 1
<EOS>
0 1 0 0
ir vamos y <EOS>


At long last, we've shown how a **Transformer** can encode a simple input phrase, **Let's go**...
Encoder
Residual Connections
Self-Attention
Position Encoding
Word Embedding
4.37 -1.01 0.82 -1.43
1 0 0 0
let's to go <EOS>
0 0 1 0
let's to go <EOS>
Encoder-Decoder Attention
el vamos y <EOS>
0 1 0 0
SoftMax
-5.1 5.7 -3.6 -4.6
5.9 0.5
0 0 0 1
ir vamos y <EOS>
el vamos y <EOS>
0 0 0 1
SoftMax
-15 -3.5 -10 9.8
1.4 6.7
0 1 0 0
ir vamos y <EOS>



In summary:

Transformers use:

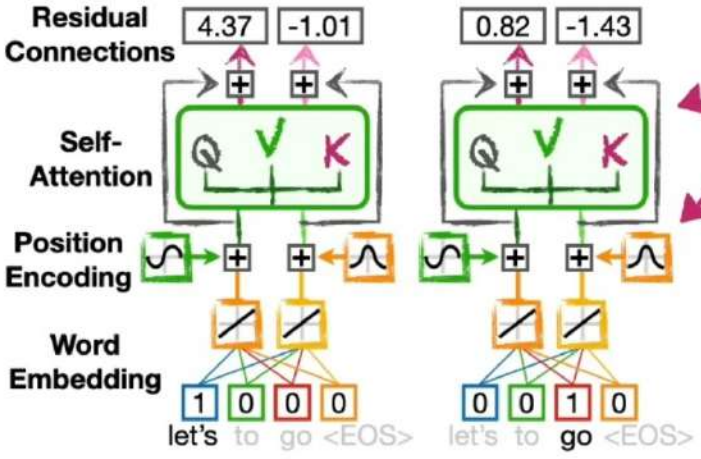
- Word Embedding to convert words into numbers
- Positional Encoding to keep track of word order
- Self-Attention to keep track of word relationships within the input and output phrases
- Encoder-Decoder attention to keep track of things between the input and output phrases to make sure important words in the input are not lost in the translation...
- and Residual Connections to allow each sub-unit, like Self-Attention, to focus on solving just one part of the problem.

Additional Notes:

- In this example, we kept things super simple. However, if we had larger vocabularies, and the original transformer had 37000 tokens, and longer input and output phrases, then, in order to get their model to work, they had to normalize the

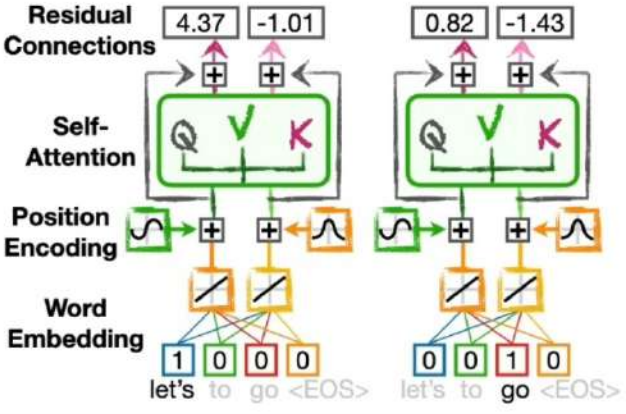
values after every step

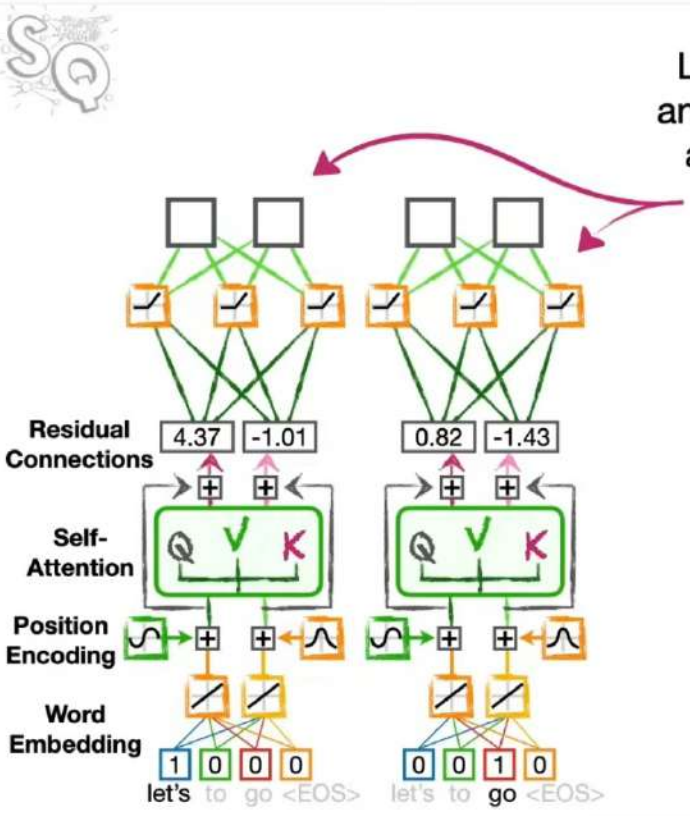
For example, they normalized the values after **Positional Encoding** and after **Self-Attention**, in both the **Encoder** and the **Decoder**.



In the original **Transformer** manuscript, they calculated the similarities with the **Dot-Product** divided by the square root of the number of embedding values per token.

$$\text{Similarity} = \frac{\text{Dot-Product}}{\sqrt{\#\text{ Embedding Values}}}$$





Lastly, to give a **Transformer** more **Weights** and **Biases** to fit to complicated data, you can add additional neural networks with hidden layers to both the **Encoder** and **Decoder**.

bam.

The end