Chapter 12

Even Better Language Translation with Transformers!!!

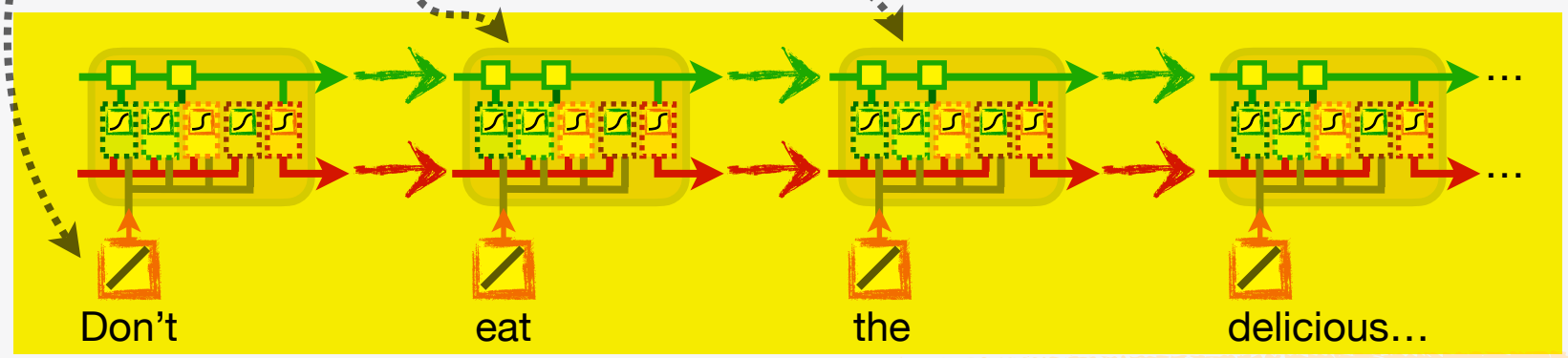
Transformers: Main Ideas

1 **A Problem: Encoder-Decoder models** based on **LSTMs** plus **Attention** work well, but they have to process the input and generate the output sequentially, one token at a time.

For example, we have to encode the first token to calculate the **Long-Term Memories** and **Short-Term Memories** before we can start to encode the second token…

…and we can't start to encode the third token until we've encoded the second token.

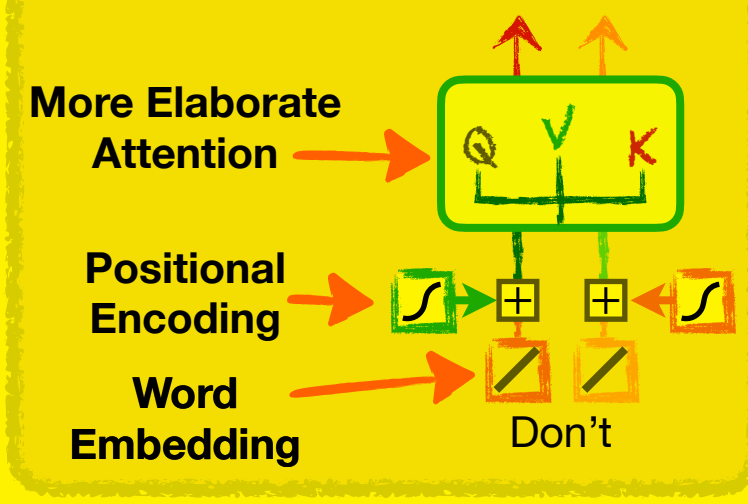
This means training our model on a large dataset with millions of tokens, like the entire Wikipedia, will take forever. Is there something we can do to speed this process up?

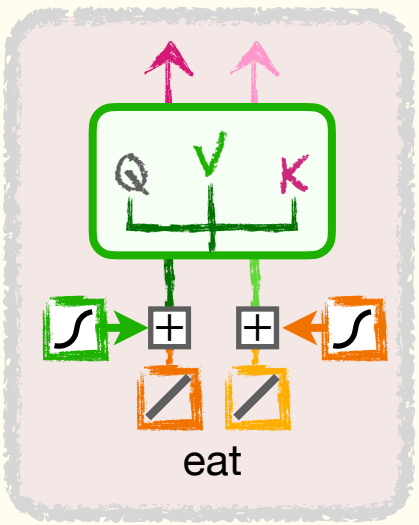


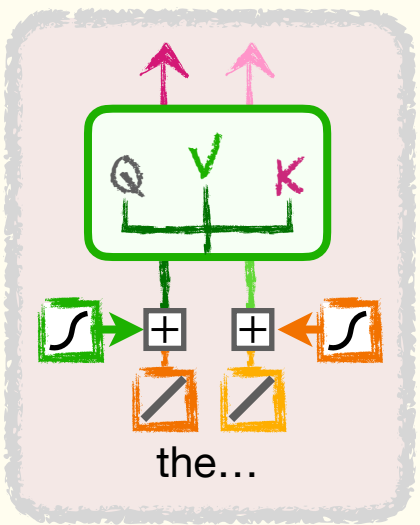
2 **A Solution: Transformers**, which replace **LSTMs** with more elaborate uses of **Attention** and something called **Positional Encoding**, can do all of the math required for training in parallel, rather than sequentially, allowing us to train the model much, much faster on much, much larger datasets.

As a result, **Transformer**-based models are now dominating the field of neural networks and AI and are used for everything from translating one language into another to writing computer programs and even bits of poetry!!!

Essential Parts of Transformers







Transformers: Encoding Details

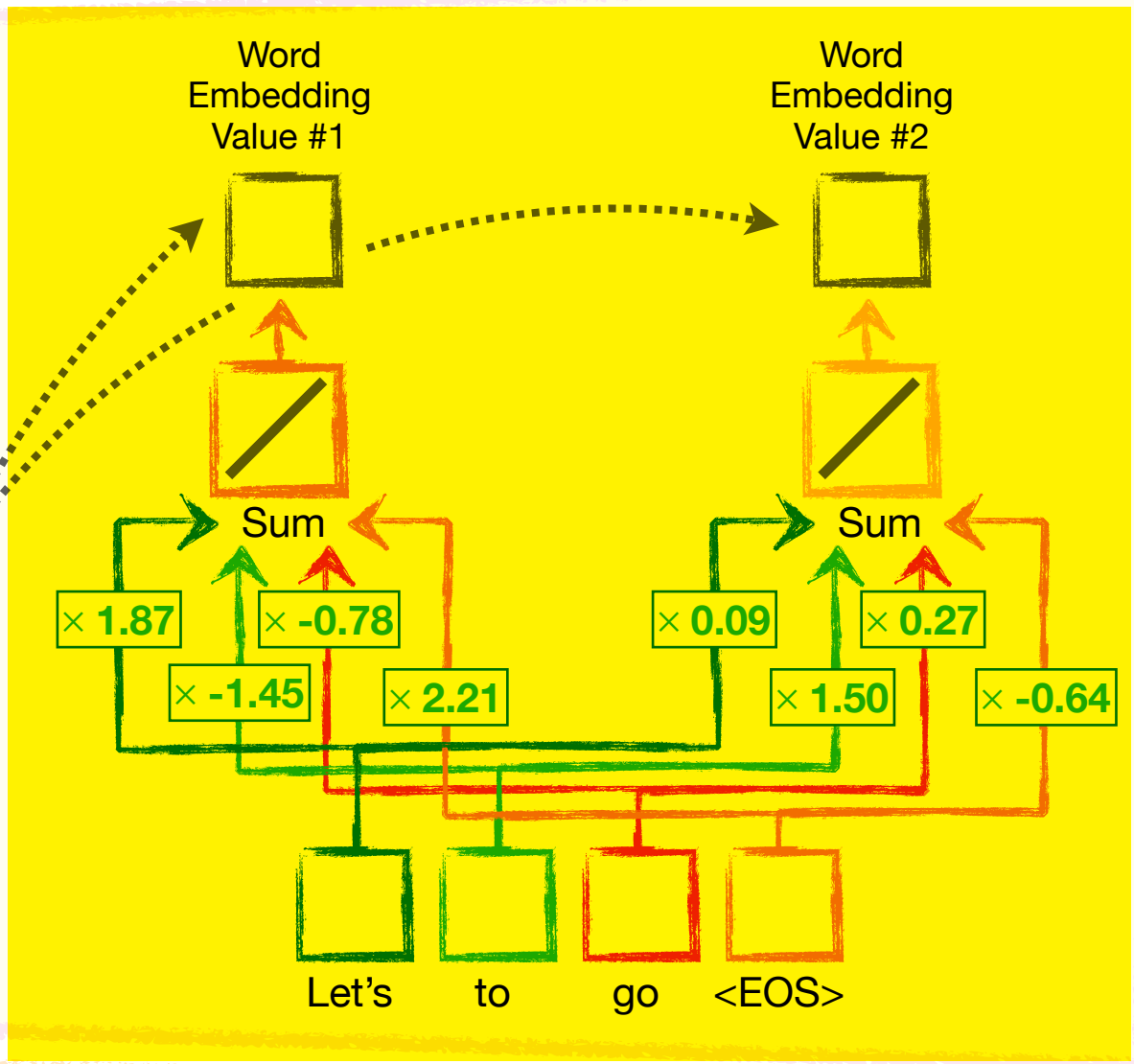
1 To keep things simple, we're going to keep using the same example we used in the last two chapters. We're going to translate the English phrase **Let's go** into the Spanish word **Vamos**.

NOTE: Although we're using a simple example here, Transformers are the basis for **Large Language Models** (**LLMs**). **LLMs** have thousands of token inputs and outputs and millions, if not billions, of weights and biases and are driving the current boom in AI.

ALSO NOTE: Some people find it easier to understand how Transformers work by just looking at the math. If you're one of those people, check out **Appendix G!**

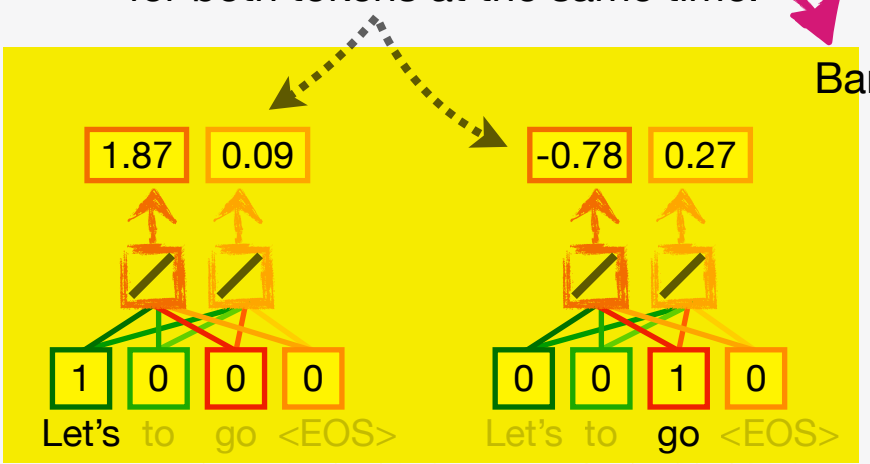
2 Just like with the Encoder-Decoder models that we talked about before, Transformers start with Word Embedding.

However, in this example, to make the math a little more interesting, we'll create **2** embedding values per token instead of just **1**.



3 So, given our input phrase, **Let's go**, we can calculate the Word Embedding values for both tokens at the same time.

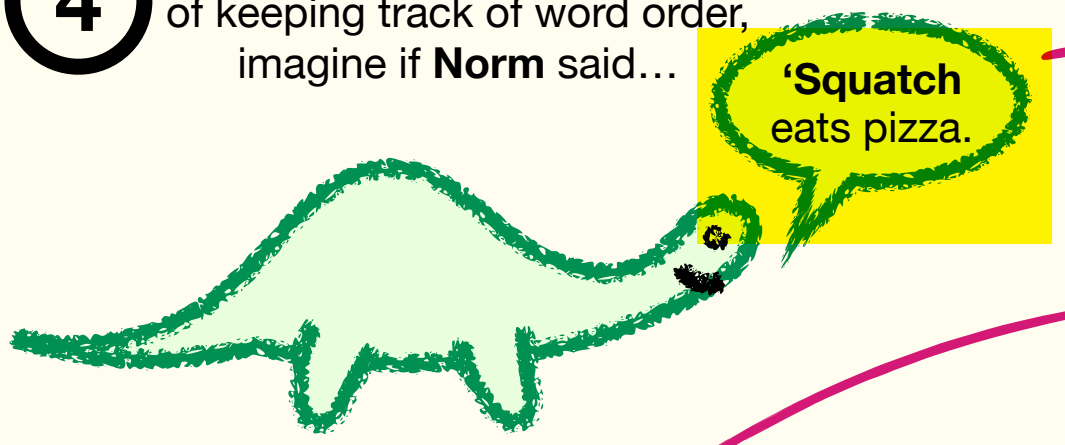
Bam!



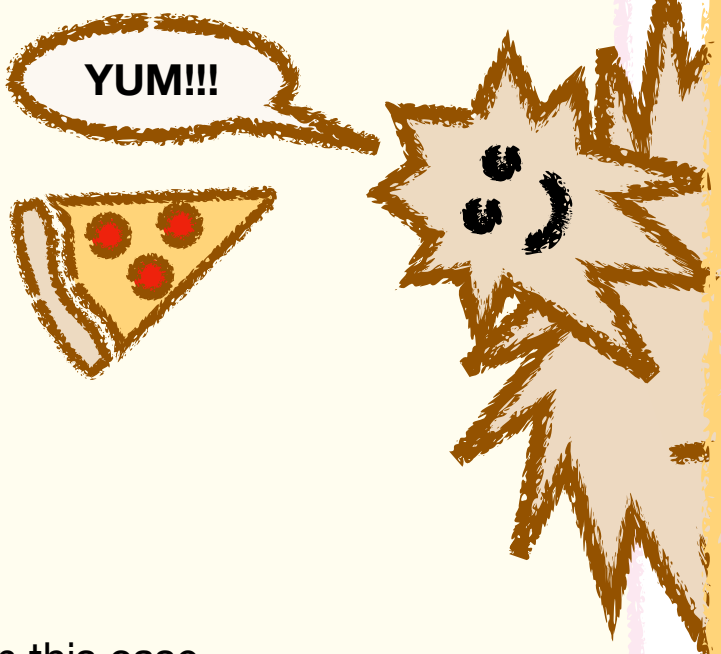
Now let's say a few things about word order.

Transformers: Encoding Details

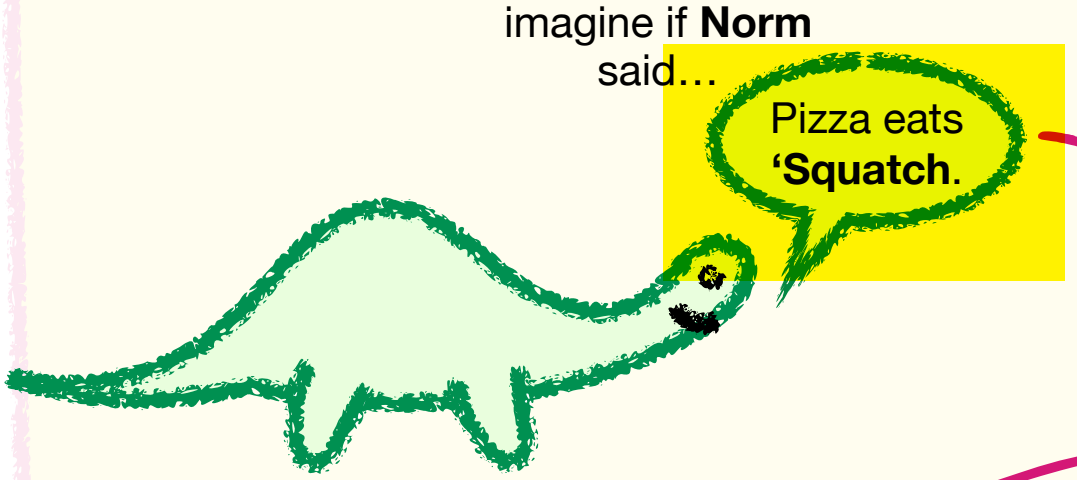
4 To understand the importance of keeping track of word order, imagine if **Norm** said…



In this case, **'Squatch** might then say…



In contrast, imagine if **Norm** said…

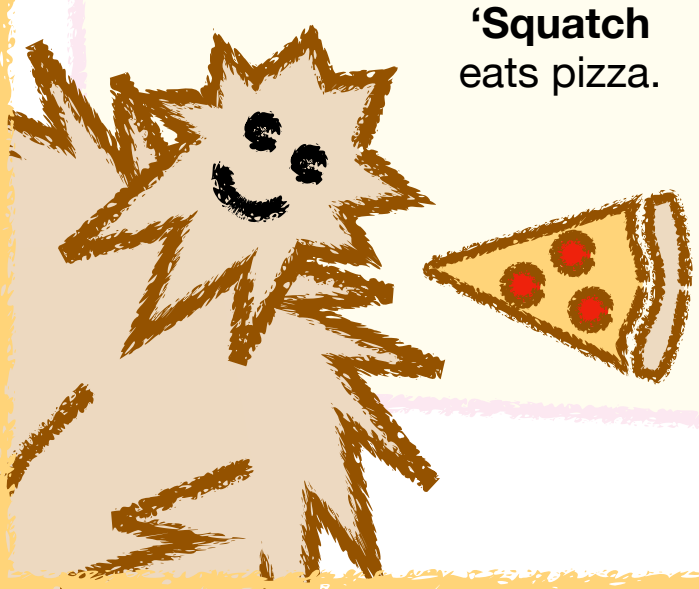


In this case, **'Squatch** might then say…

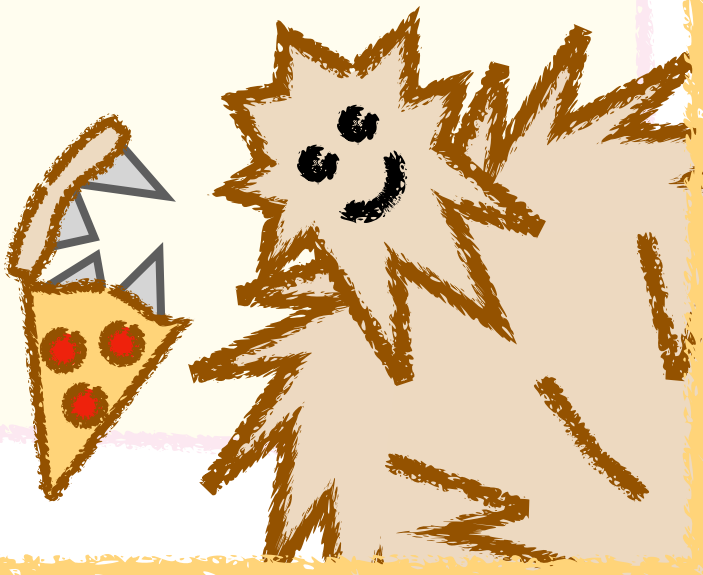
These two phrases…

…use the exact same words, but have very different meanings. So keeping track of word order is super important. Let's learn how Transformers keep track of word order with **Positional Encoding**!

'Squatch eats pizza.



Pizza eats **'Squatch**.



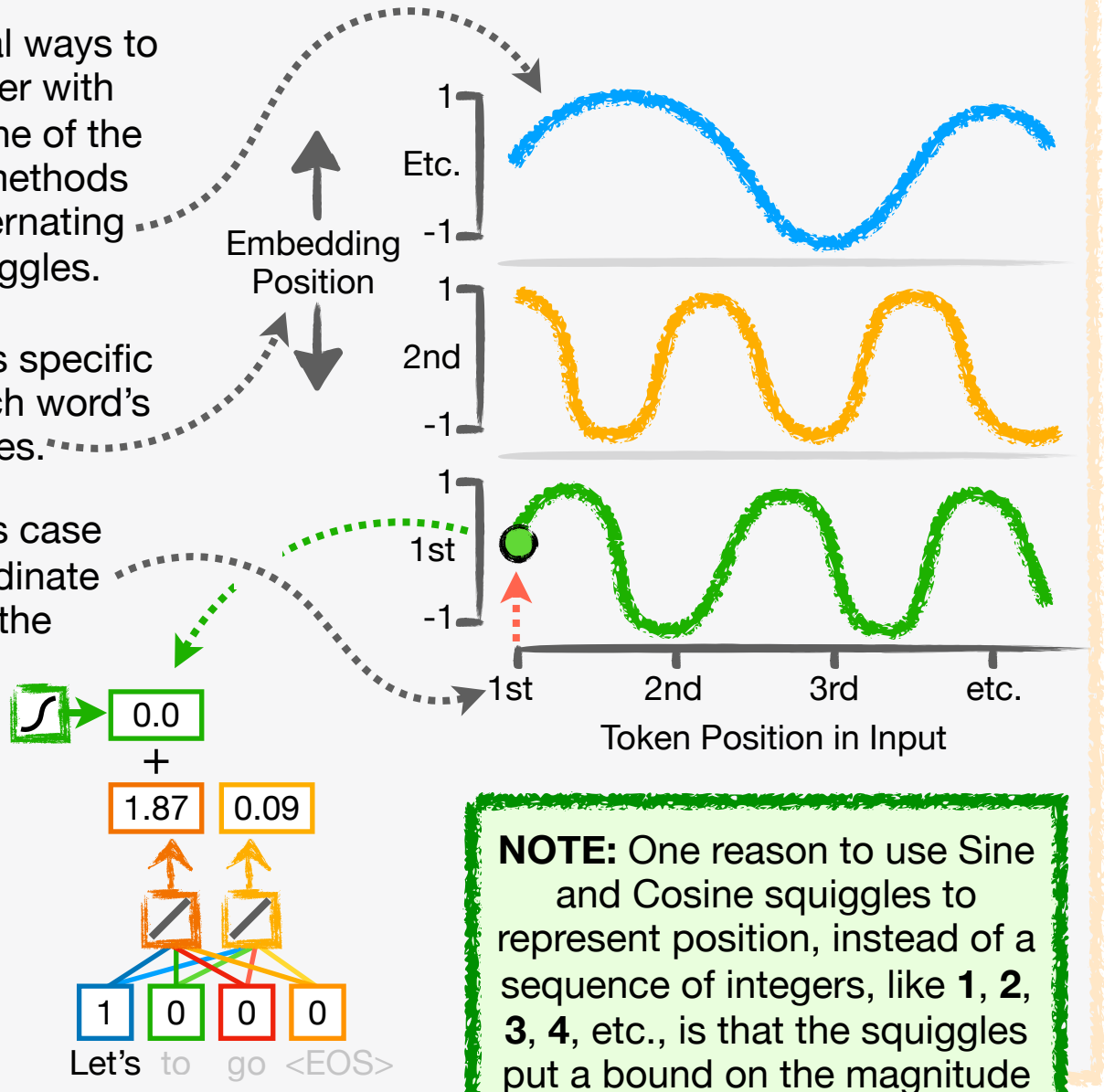
Transformers: Encoding Details

5 Although there are several ways to keep track of word order with **Positional Encoding**, one of the most commonly used methods uses a sequence of alternating **Sine** and **Cosine** squiggles.

Each squiggle gives us specific position values for each word's embedding values.

The first token, which in this case is **Let's**, has an x-axis coordinate all the way to the left on the **green squiggle**…

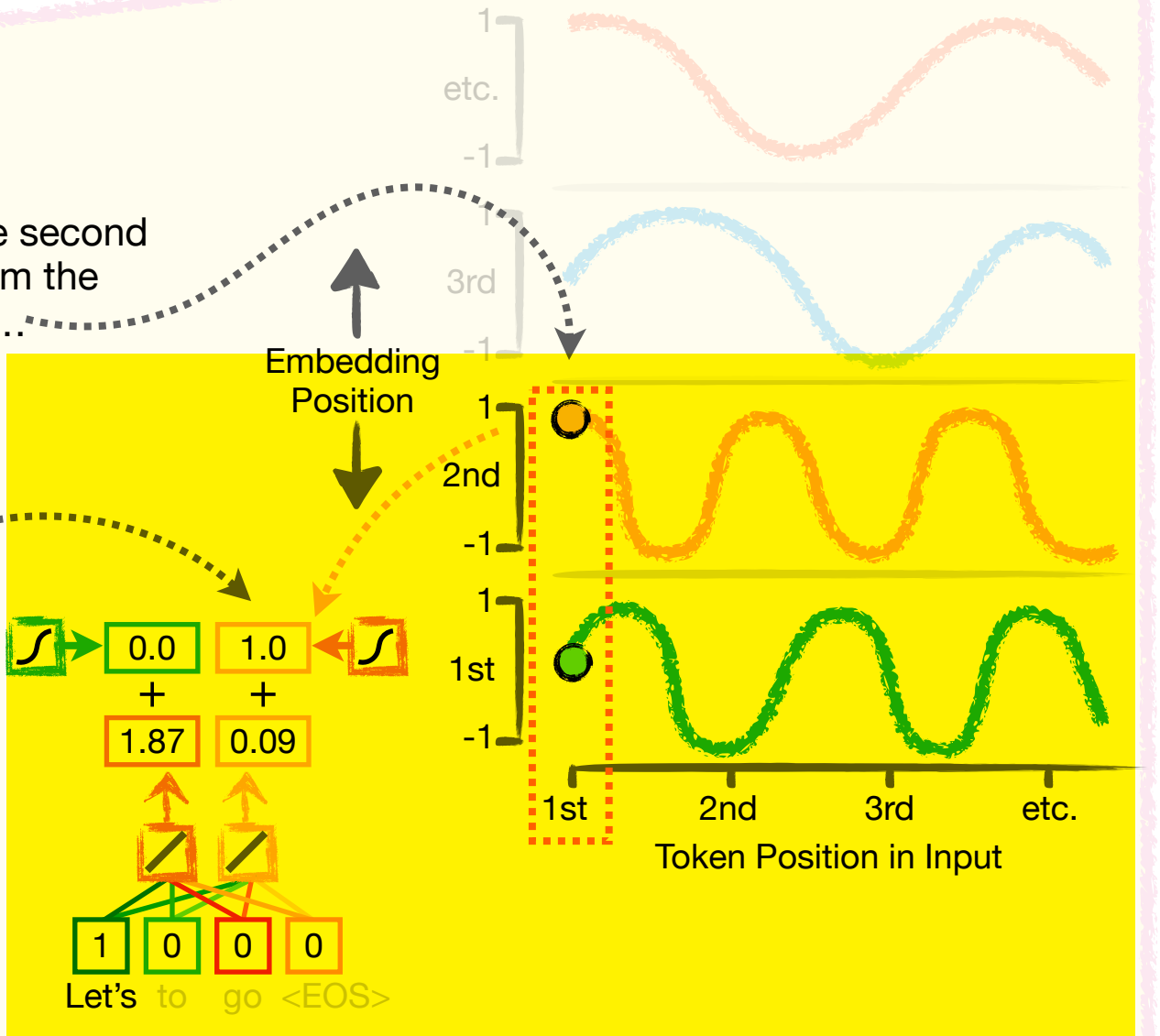
…and the position value for its first embedding is the y-axis coordinate **0**.



NOTE: One reason to use Sine and Cosine squiggles to represent position, instead of a sequence of integers, like **1**, **2**, **3**, **4**, etc., is that the squiggles put a bound on the magnitude of the values; they will always be between **-1** and **1**.

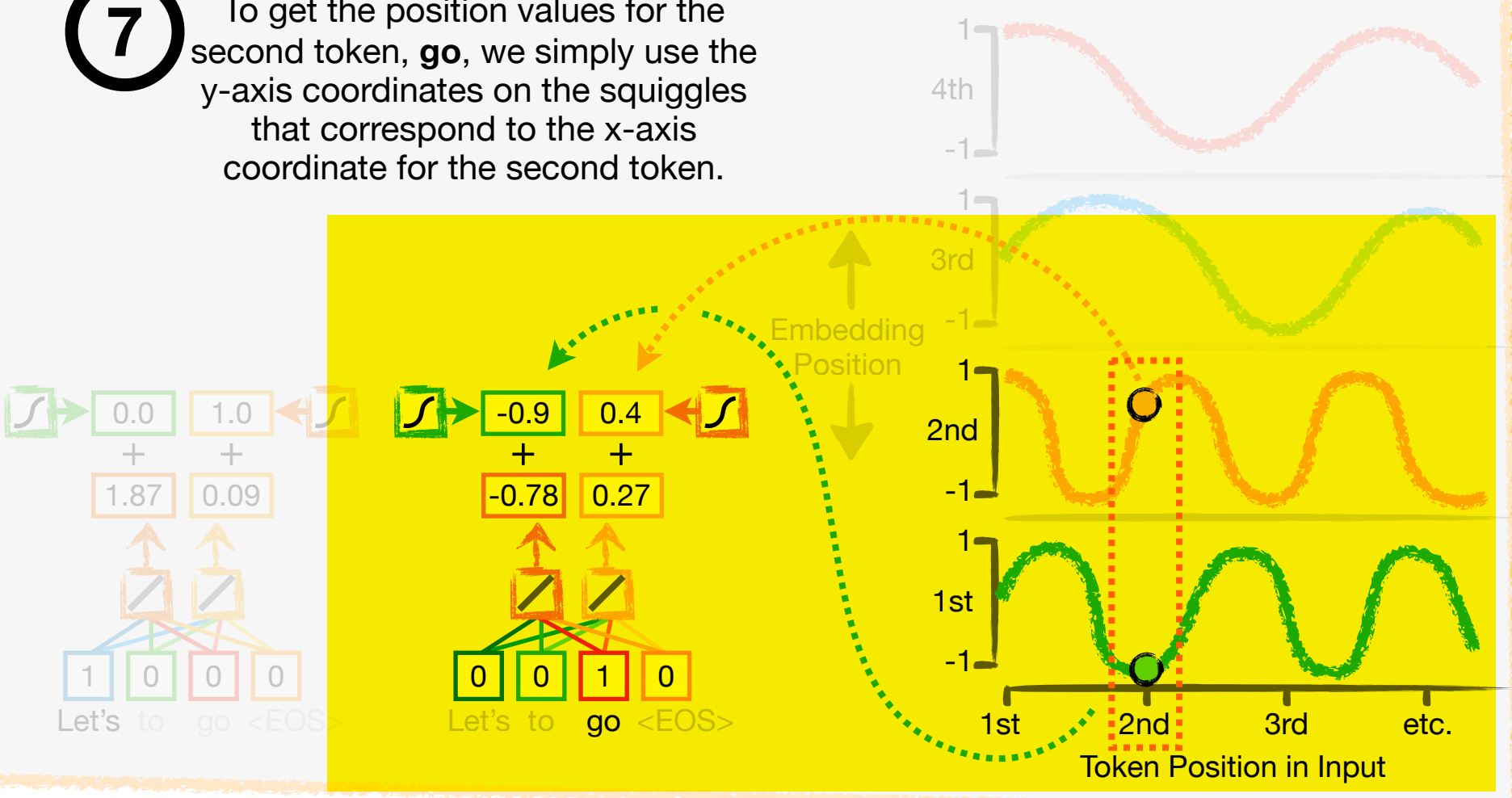
6 The position value for the second embedding comes from the **orange squiggle**…

…and the y-axis coordinate on the **orange squiggle** that corresponds to the first word is **1**.



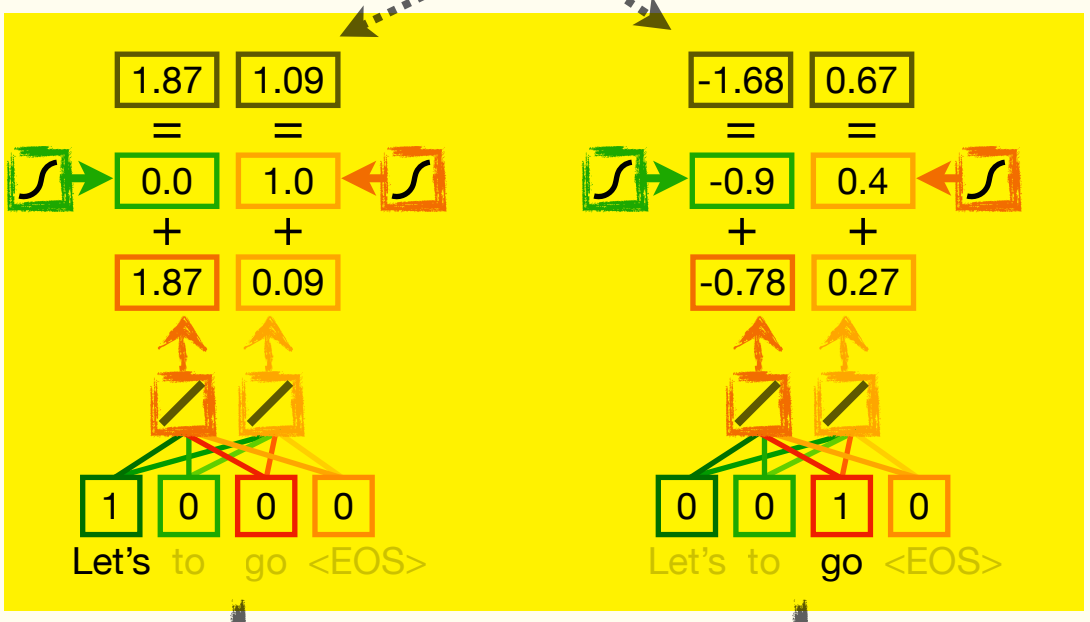
Transformers: Encoding Details

7 To get the position values for the second token, **go**, we simply use the y-axis coordinates on the squiggles that correspond to the x-axis coordinate for the second token.



8 Now we just do the math for both tokens at the same time to get their Positional Encoding…

…and we end up with Word Embeddings plus Positional Encoding for the prompt…



9 From here on out, we'll consolidate the math for Word Embedding and Positional Encoding like this.

Now that we know how to keep track of each word's position, let's talk about how a Transformer uses Attention to keep track of the relationships among words within a phrase.

Transformers: Encoding Details

10 For example, if **'Squatch** said…

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

…then the word **it** could refer to **pizza**…

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

…or it could refer to the word **oven**.

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

I've heard of good-tasting pizza, but never a good-tasting oven!

I know **'Squatch**! That's why it's important that the Transformer correctly associates the word **it** with **pizza**.

11 The good news is that Transformers have something called **Self-Attention**, which is a mechanism to correctly associate the word **it** with the word **pizza**.

In general terms, **Self-Attention** works by seeing how similar each word is to all of the words in the sentence, including itself.

For example, **Self-Attention** calculates Similarity Scores between the first word, **The**, and all of the words in the sentence…

…including itself.

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

Transformers: Encoding Details

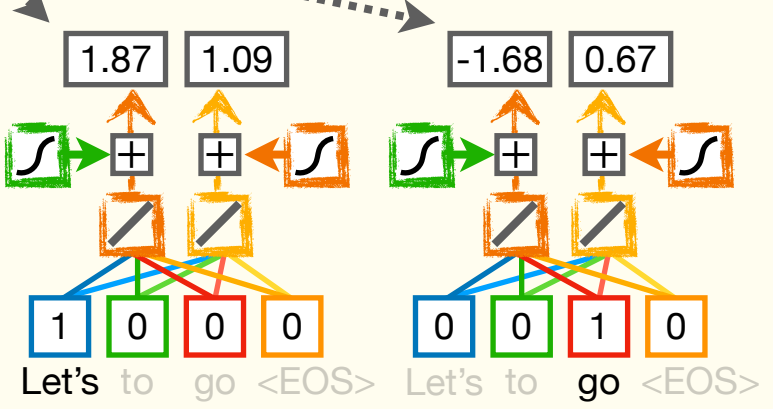
12 Likewise, Self-Attention would calculate the Similarity Scores between **pizza** and all of the words, including itself…

…and, ultimately, Self-Attention would calculate the Similarity Scores between **good** and all of the words, including itself.

13 Once the Similarity Scores are calculated, they're used to determine how the Transformer encodes each word in ways very similar to what we saw in **Chapter 11**.

In other words, if we trained our model on a lot of sentences about **pizza,** and the word **it** was more commonly associated with **pizza** than with **oven**, then the Similarity Score for **pizza** would cause **pizza** to have a larger impact on how the Transformer encodes the word **it**.

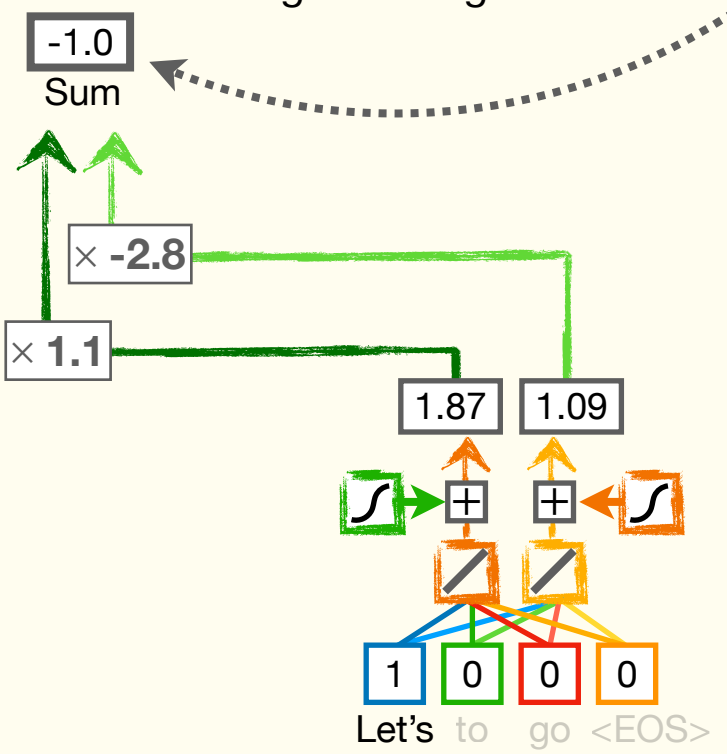
14 In the chapter on Attention, we calculated Similarity Scores directly from the sums of the Word Embeddings and the Positional Encodings.



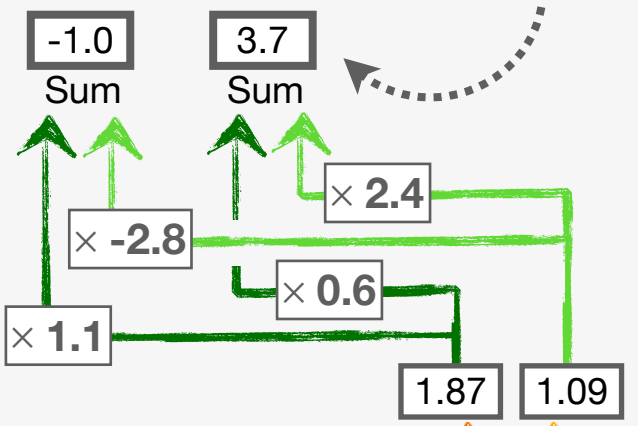
In contrast, Transformers do things a little differently. So let's look at the details!

Transformers: Encoding Details

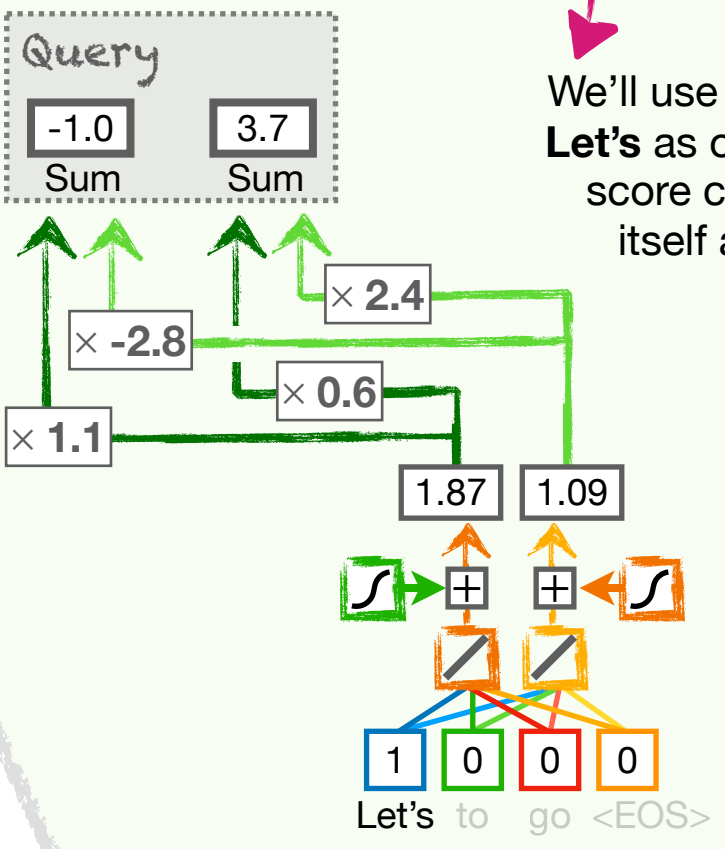
15 Instead of using the encoded values for **Let's** directly, the Transformer multiplies them by a pair of weights and then adds those products together to get a new number.



16 Then, typically, the Transformer will repeat that process with different pairs of weights until we end up with the same number of new values as encodings for each token. In this example, that means we create **2** new numbers.



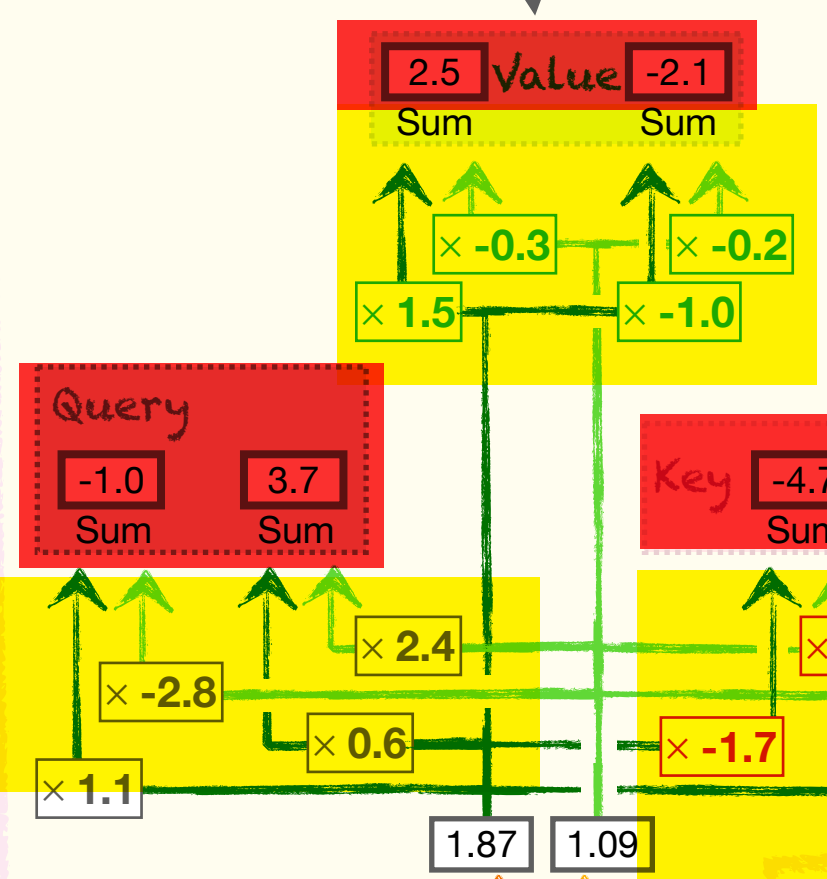
17 In Transformer terminology, these **2** new numbers that represent the word **Let's** are called the **Query** for the word **Let's**.



We'll use the **Query** for the word **Let's** as one part of the similarity score calculations relative to itself and to the word **go**.

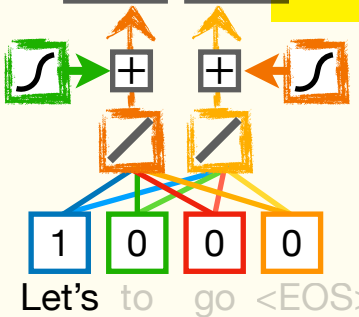
Transformers: Encoding Details

18 But before we do those similarity calculations, the Transformer uses another bunch of weights to create a set of numbers that are called the **Value** for the word **Let's**…



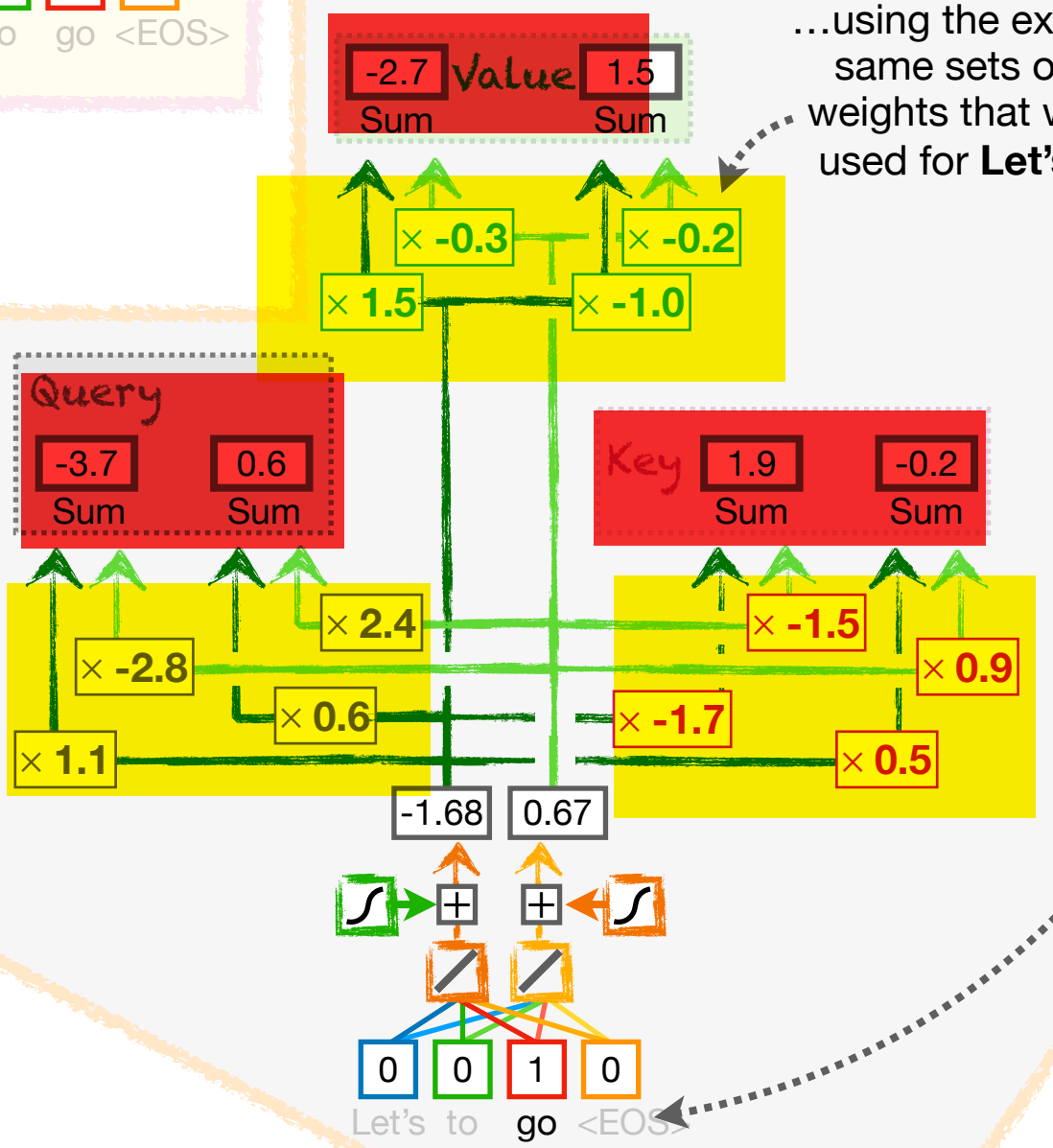
…and a third bunch of weights to create a set of numbers called the **Key** for the word **Let's**.

The **Query**, **Key**, and **Value** terminology comes from **Databases**. The **Query** is our question we ask the database, the **Keys** represent the items in the database, and the **Value** represents what the database returns when it matches the **Query** with a **Key**.



19 Simultaneously, the Transformer creates a **Query**, **Key**, and **Value** for the second token, **go**…

…using the exact same sets of weights that we used for **Let's**.

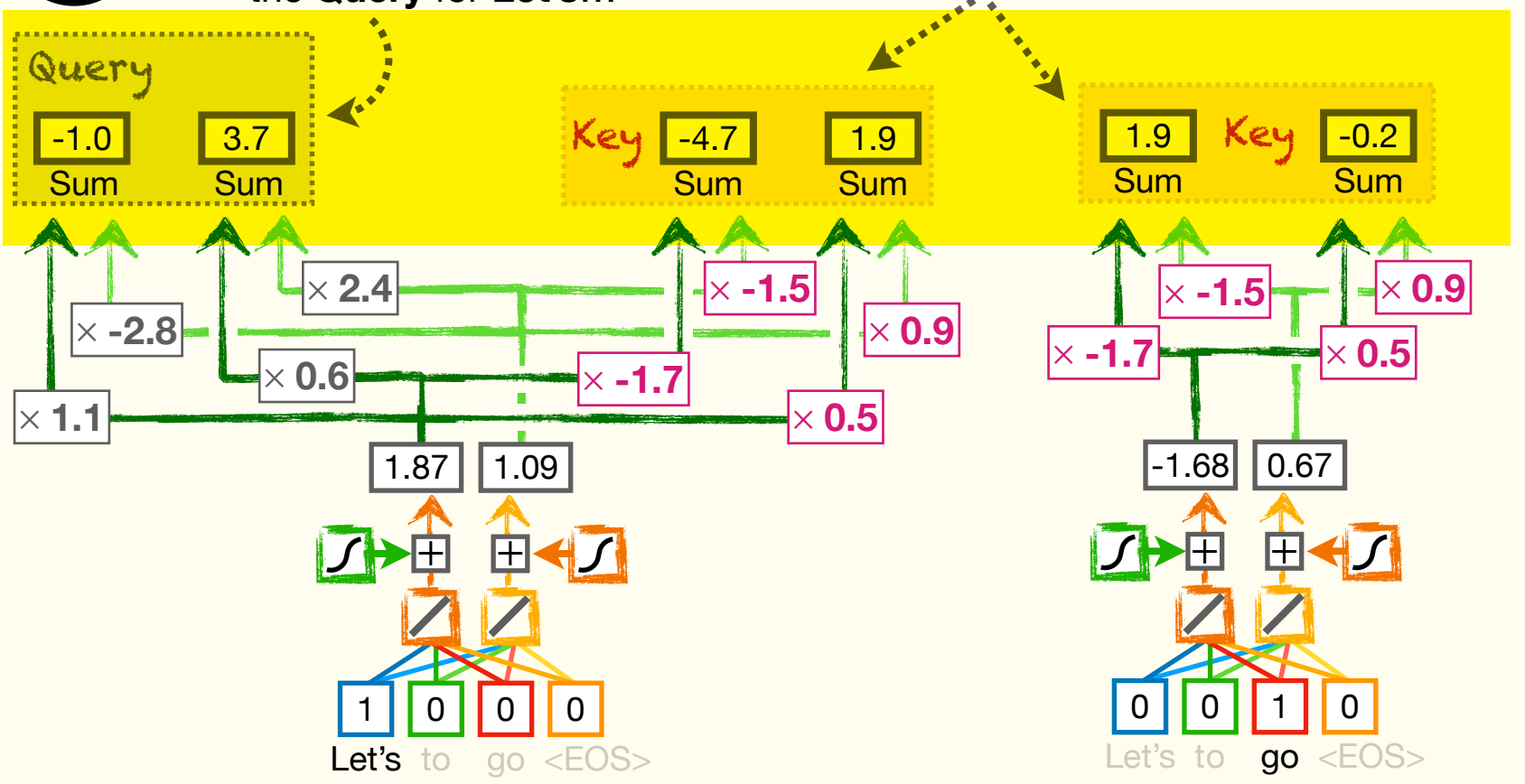


Transformers: Encoding Details

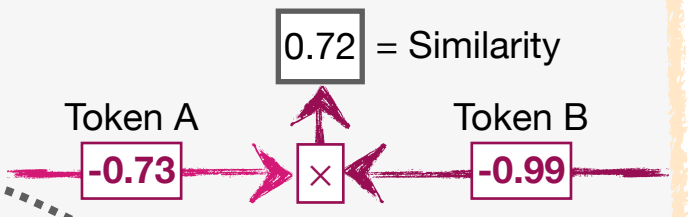
20 Now, the first step in calculating the Attention value for **Let's** uses the **Query** for **Let's**…

…and the **Keys** for **Let's** and **go**…

…to calculate similarities.



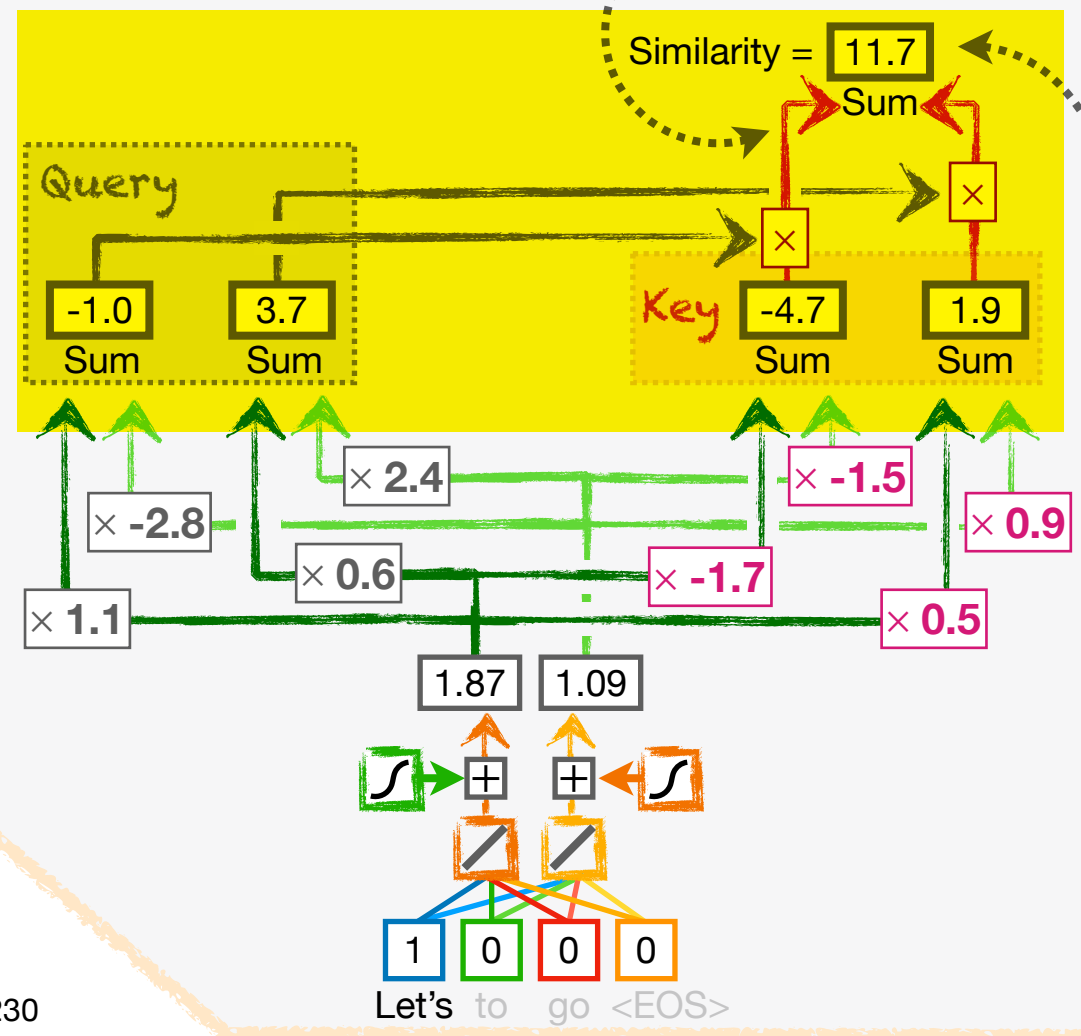
21 In **Chapter 11**, when we first introduced **Attention**, we only had one number representing each token, and we calculated the similarity between two tokens by just multiplying them together.



In this example, we have **2** numbers per token, and we can calculate similarity by multiplying the pairs of numbers together and then adding the products.

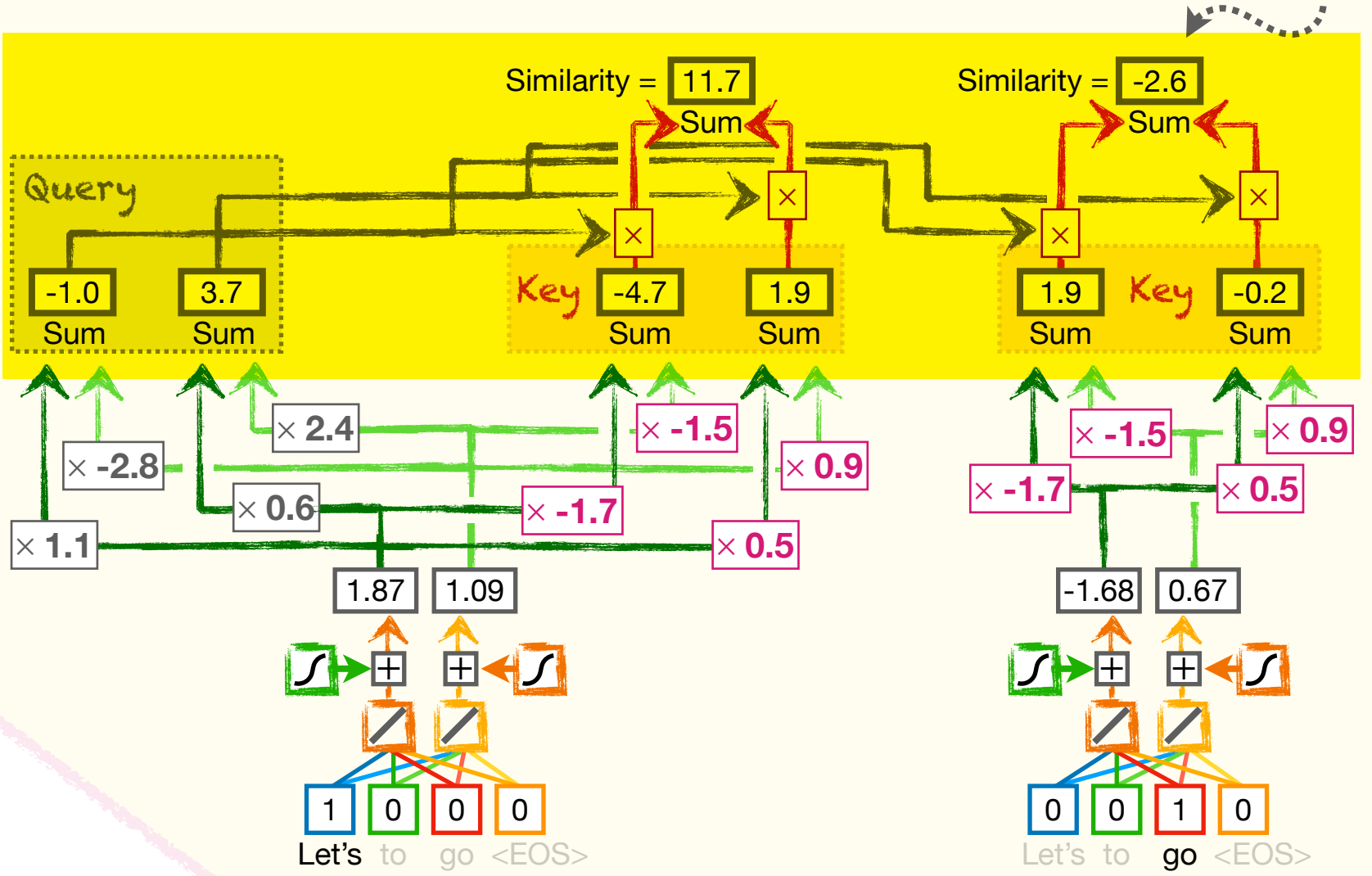
When we do calculate similarity this way, it's called the **Dot Product Similarity**.

Thus, using the **Query** and **Key** for **Let's**, the **Dot Product Similarity** with itself is **11.7**.



Transformers: Details

22 Likewise, using the **Query** for **Let's** and **Key** for **go**, the Dot Product Similarity is **-2.6**.



23

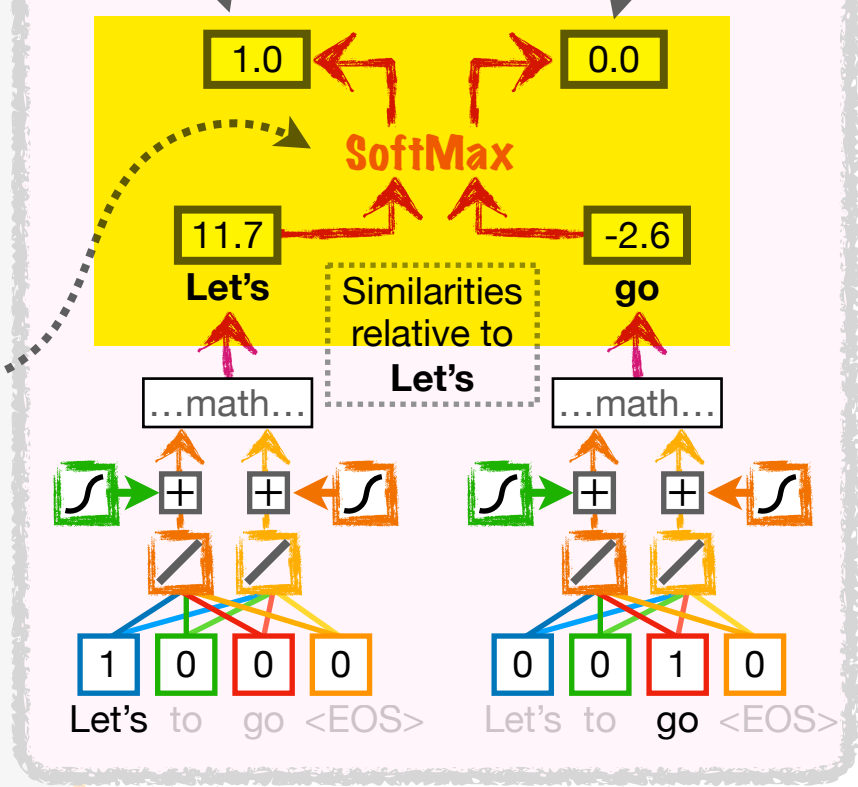
When we calculate the similarities relative to **Let's** by using its **Query**, we see that **Let's** is much more similar to itself (similarity = **11.7**) than it is to the word **go** (similarity = **-2.6**), and that means we want **Let's** to have more influence on its encoding than the word **go**.

And, just like we saw in **Chapter 11** on **Attention**, we determine how much influence each token has on the encoding by first running the similarity scores through the SoftMax function.

Remember: 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**.

24 In this case, because **Let's** is so much more similar to itself than the word **go**, we'll use **100%** of the word **Let's** to encode **Let's**…

…and we'll use **0**% of the word **go** to encode the word **Let's**.

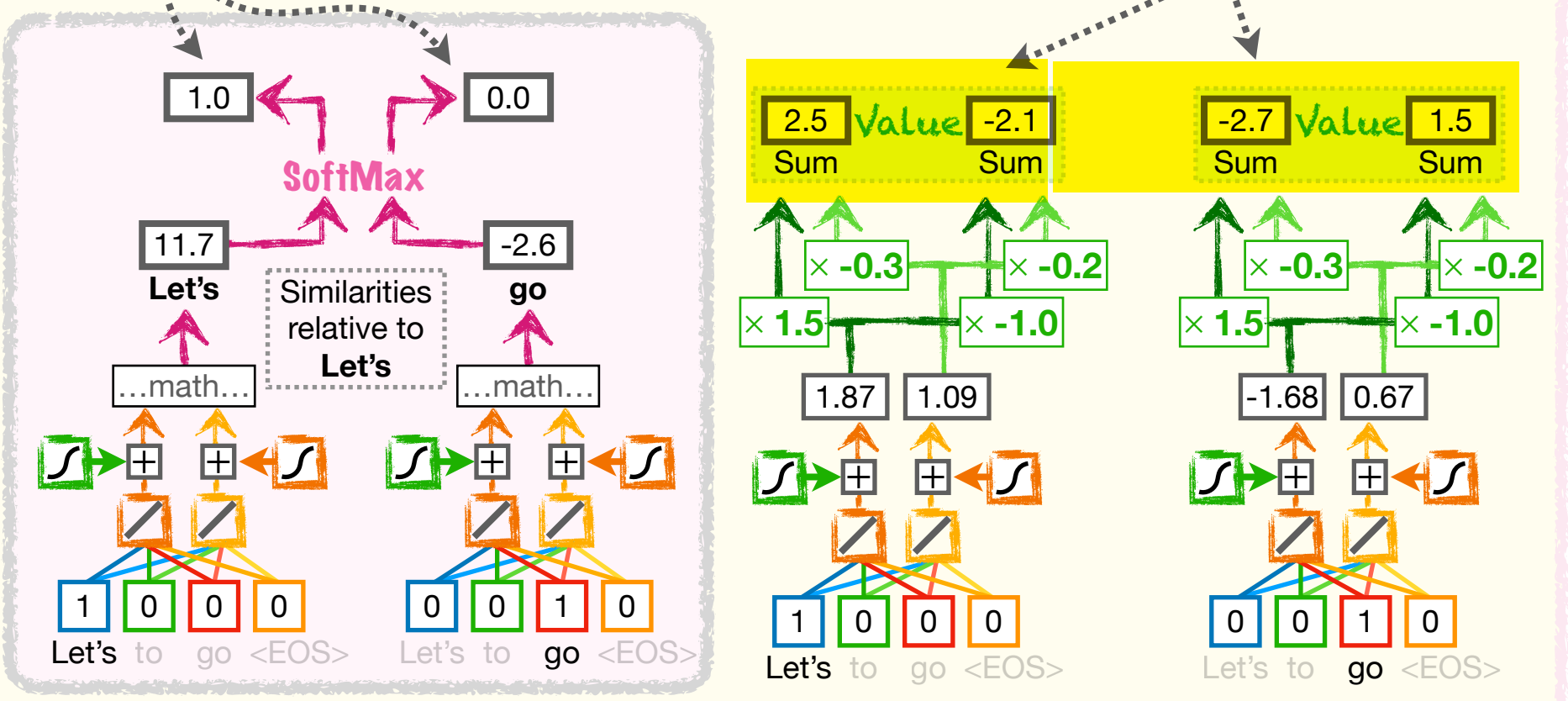


Transformers: Encoding Details

25

Now that we know what percentage of each token should be used to encode **Let's**…

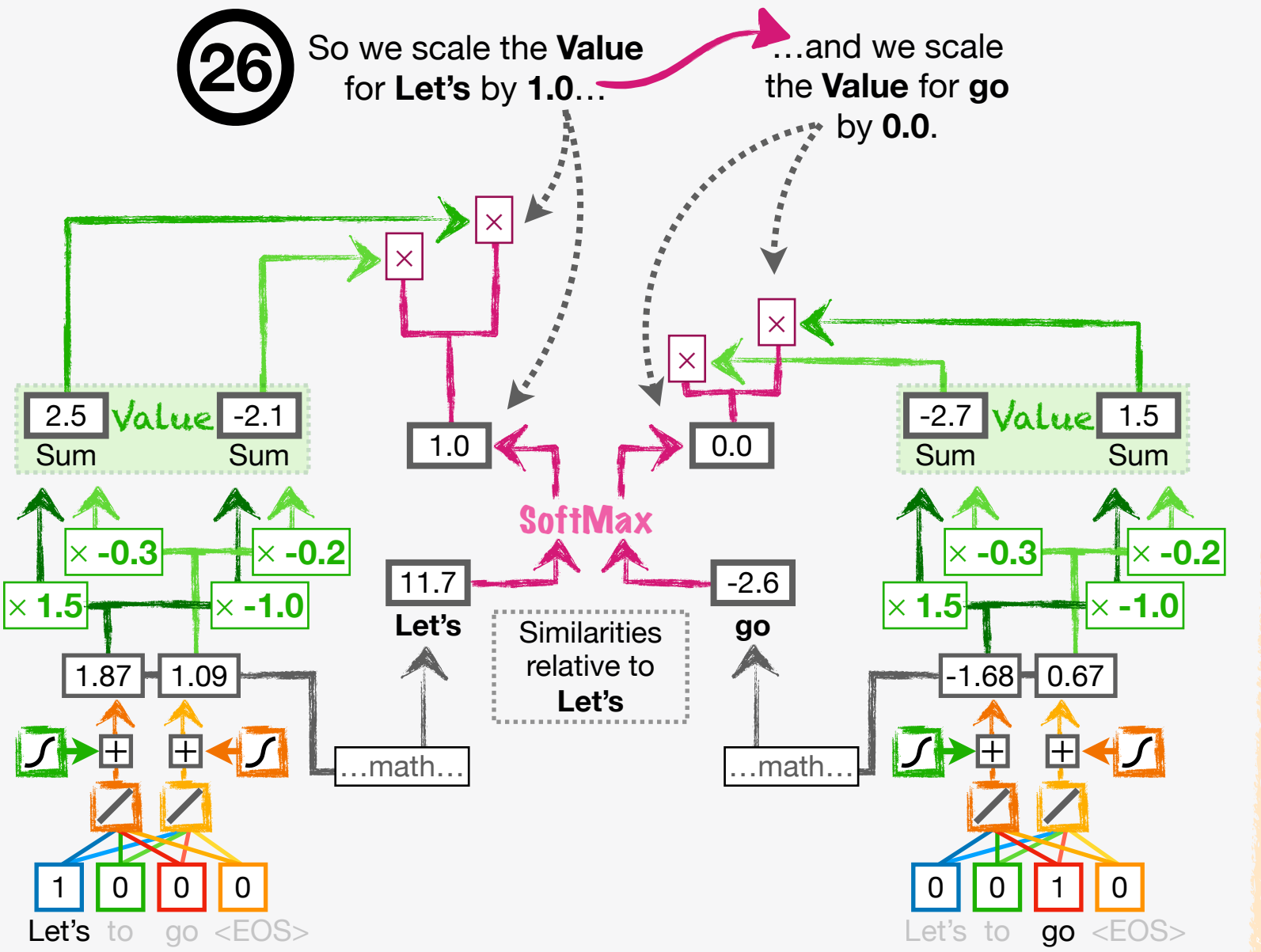
…we use those percentages to scale the **Values** that we calculated for each token earlier.



26

So we scale the **Value** for **Let's** by **1.0**…

…and we scale the **Value** for **go** by **0.0**.

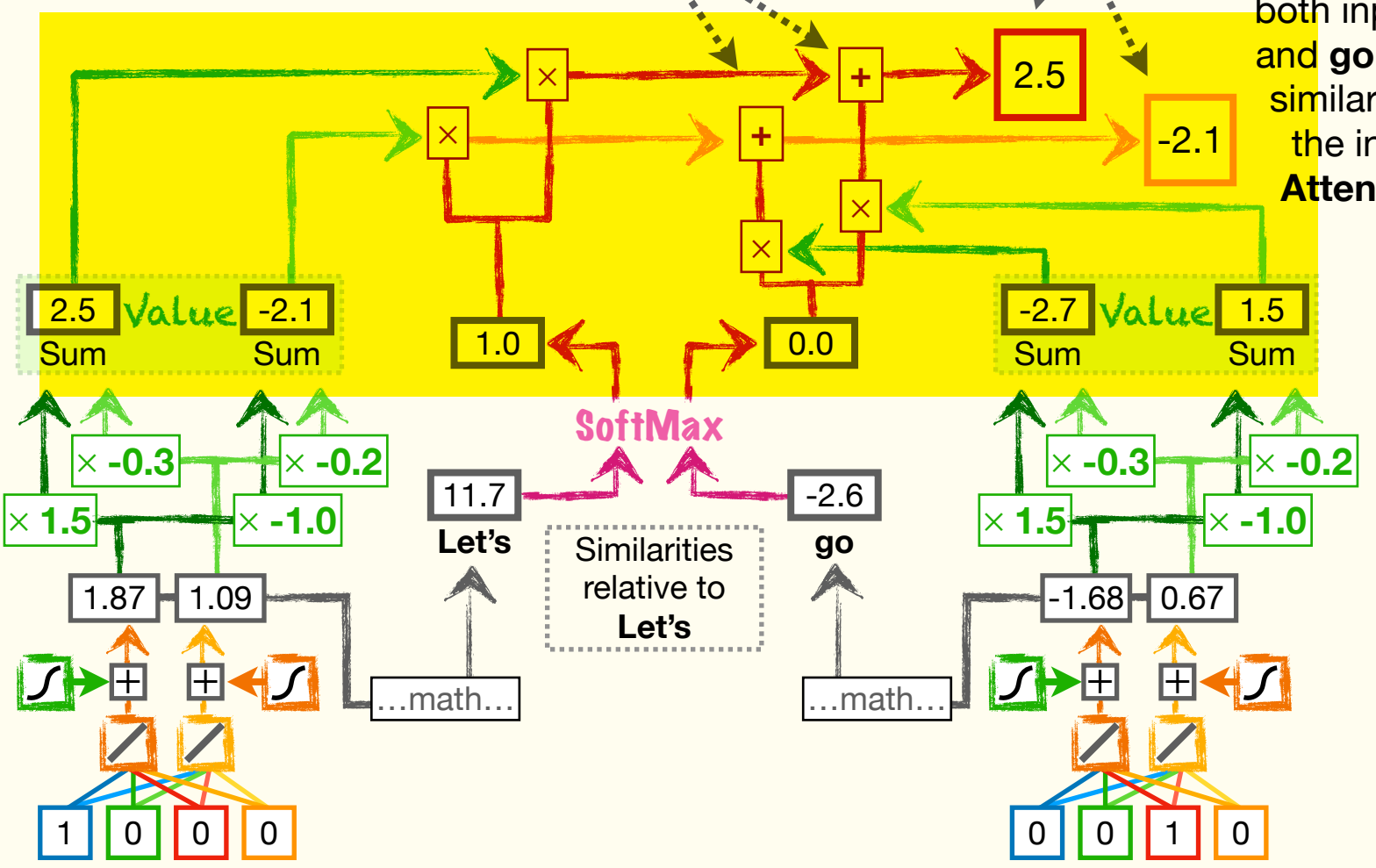


Transformers: Encoding Details

27 Lastly, we add the scaled **Values** together…

…and these sums…

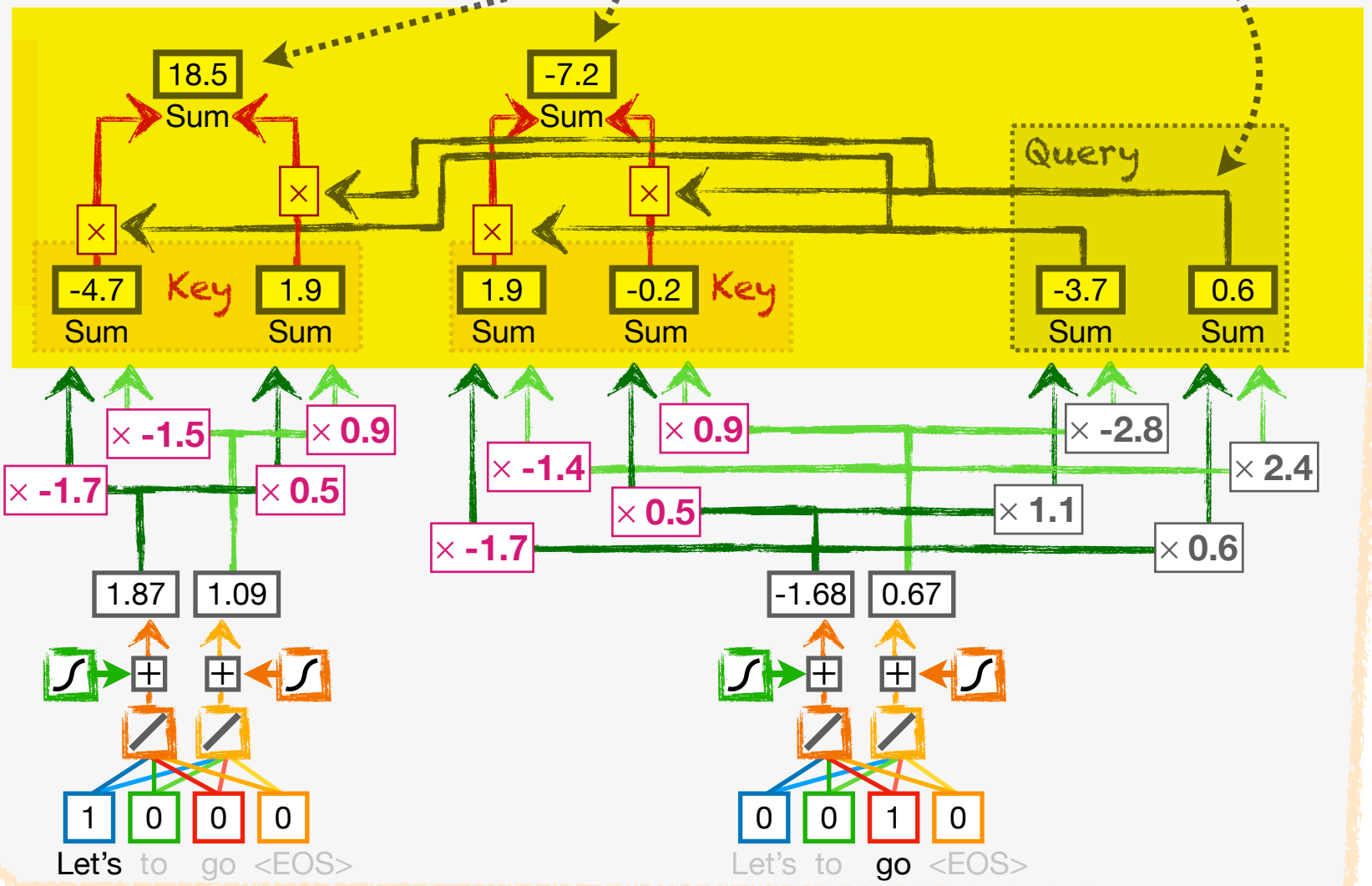
…which combine separate encodings for both input words, **Let's** and **go**, relative to their similarity to **Let's**, are the individual **Self-Attention** values for **Let's**.



Bam!

Let's to go <EOS>

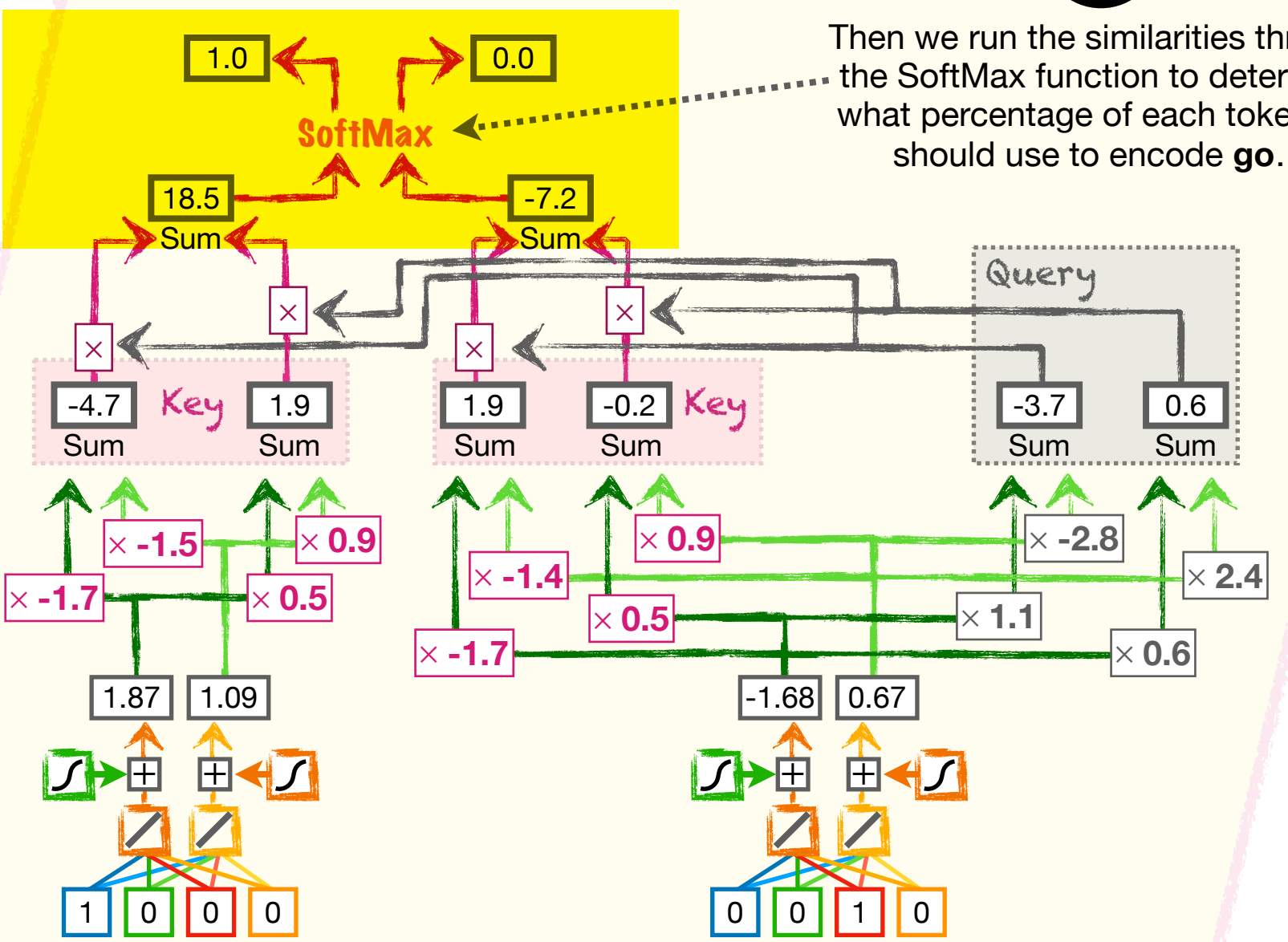
28 Now, to calculate the individual Self-Attention values for **go**, we calculate similarities using the **Query** for **go** and the **Keys** for **Let's** and **go**.

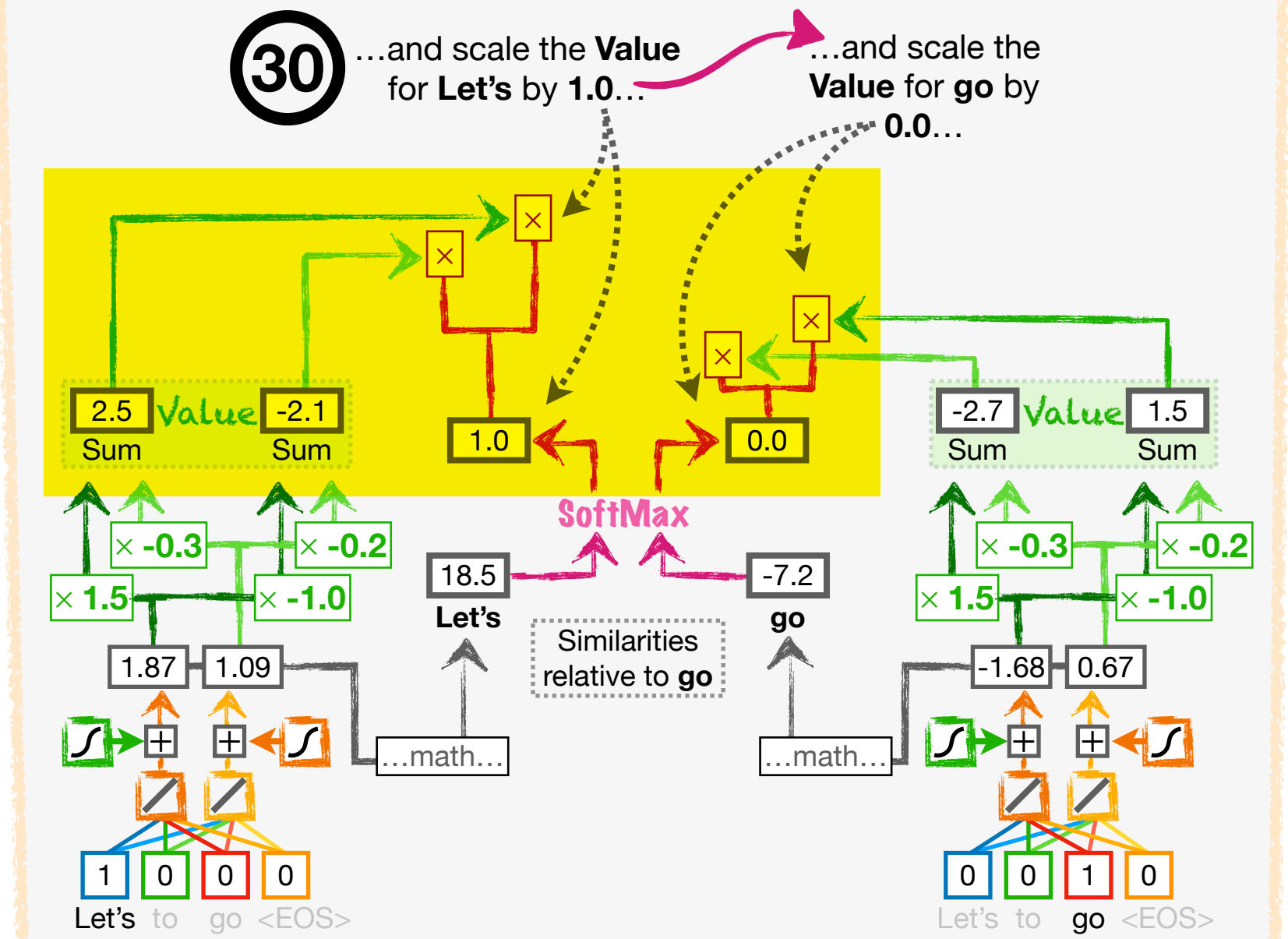


Transformers: Encoding Details

29

Then we run the similarities through the SoftMax function to determine what percentage of each token we should use to encode **go**…



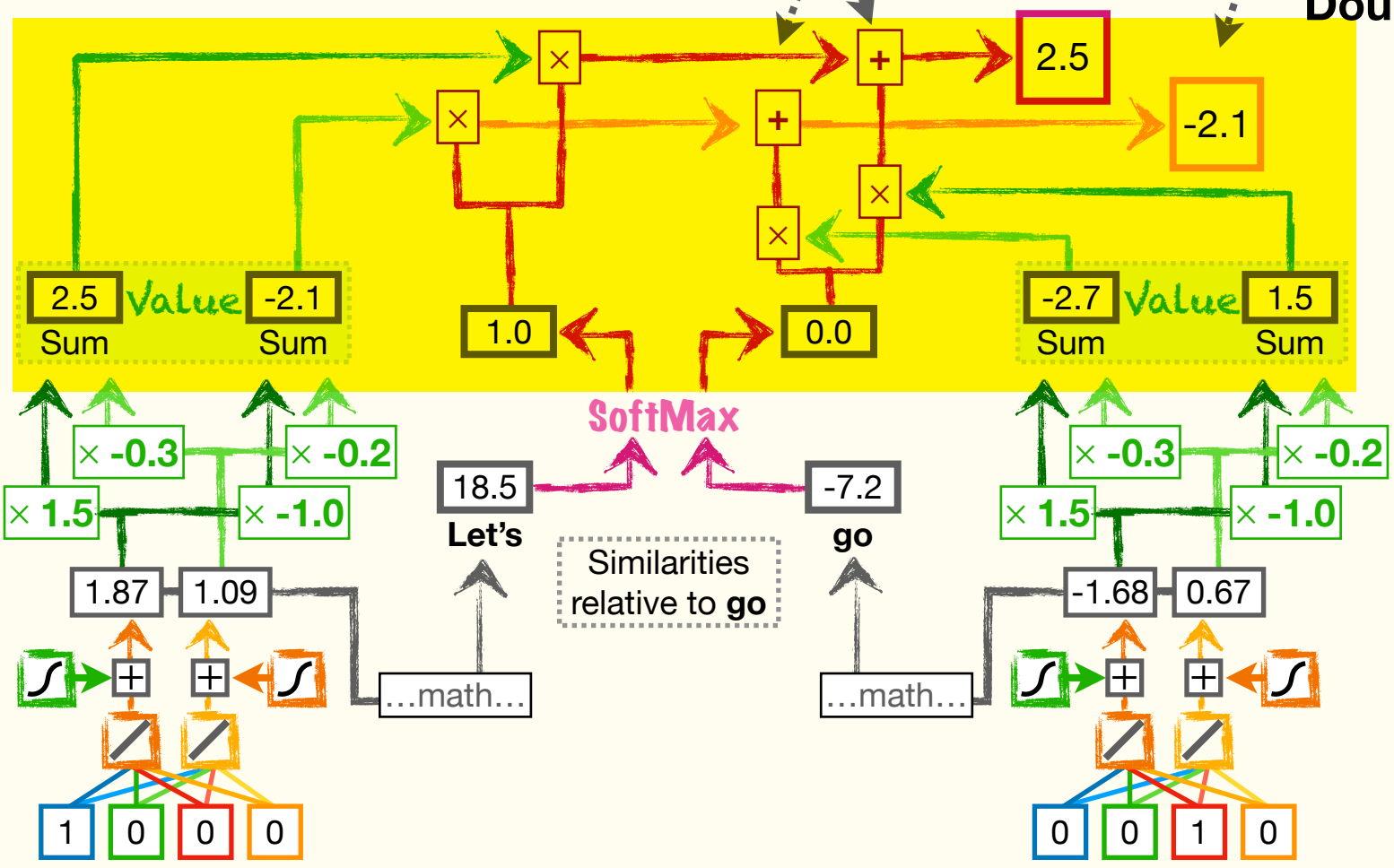


Transformers: Encoding Details

31 …and add the scaled values together…

…to get the individual Self-Attention values for **go**.

Double Bam!!

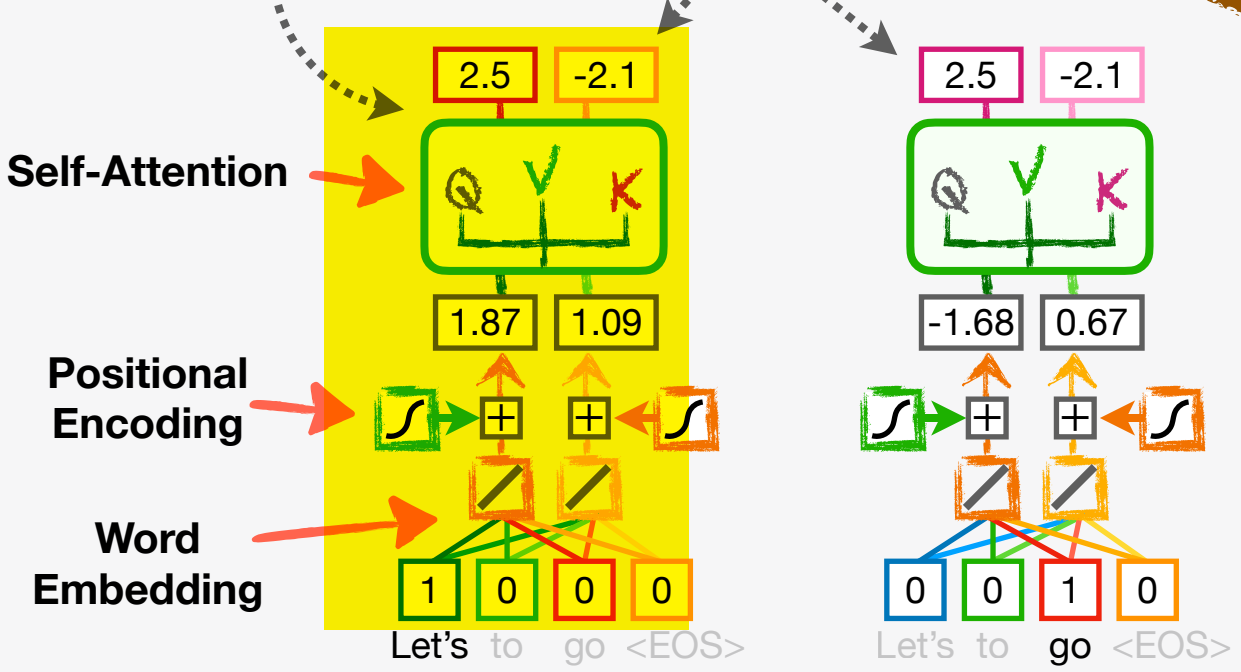


Let's to go <EOS>

Let's to go <EOS>

32 In summary, Transformers create **Queries**, **Keys**, and **Values** for each token to calculate **Self-Attention.**

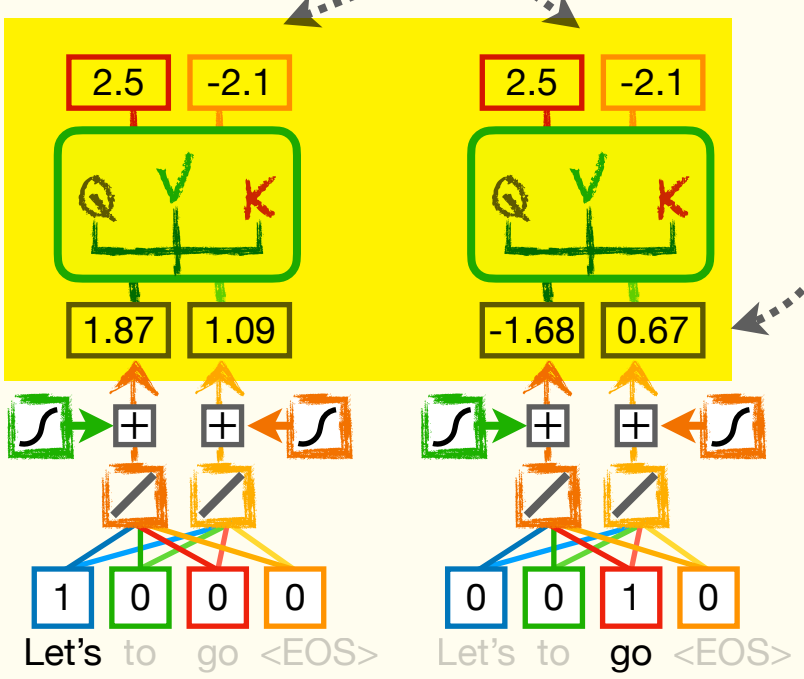
Whenever I finish reading a summary, I like to take a break and eat a snack! It helps me digest the information.



NOTE: You may have noticed that the individual Self-Attention values for both tokens are the same. That's not a typo. That's just how the math worked out with this Transformer that was trained on an incredibly small dataset. However, if we had more tokens and more data, we'd probably get more interesting Self-Attention values for each token.

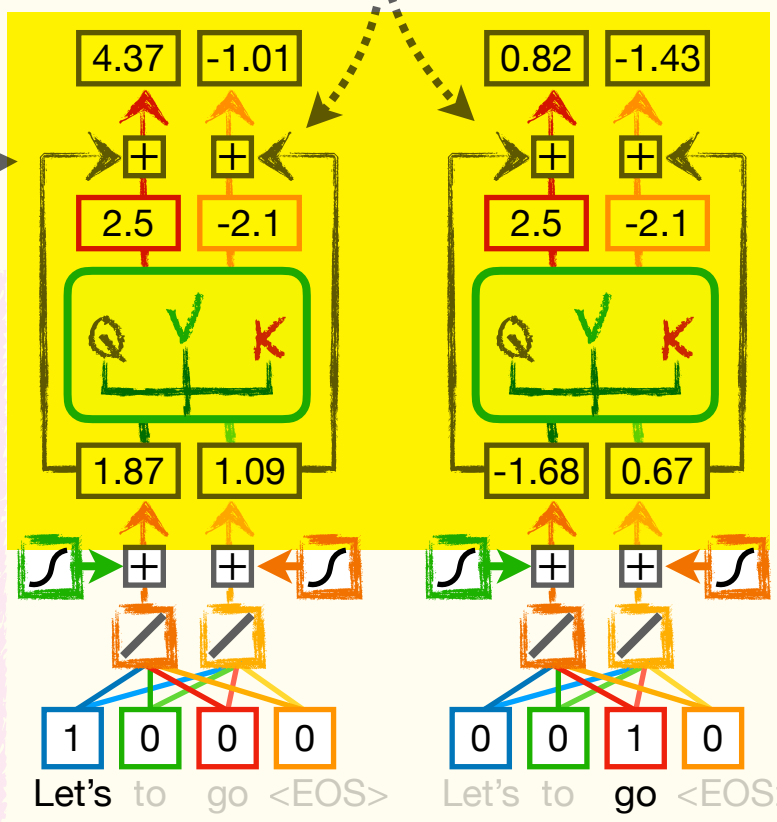
Transformers: Encoding Details

33 After calculating Self-Attention for each input token…

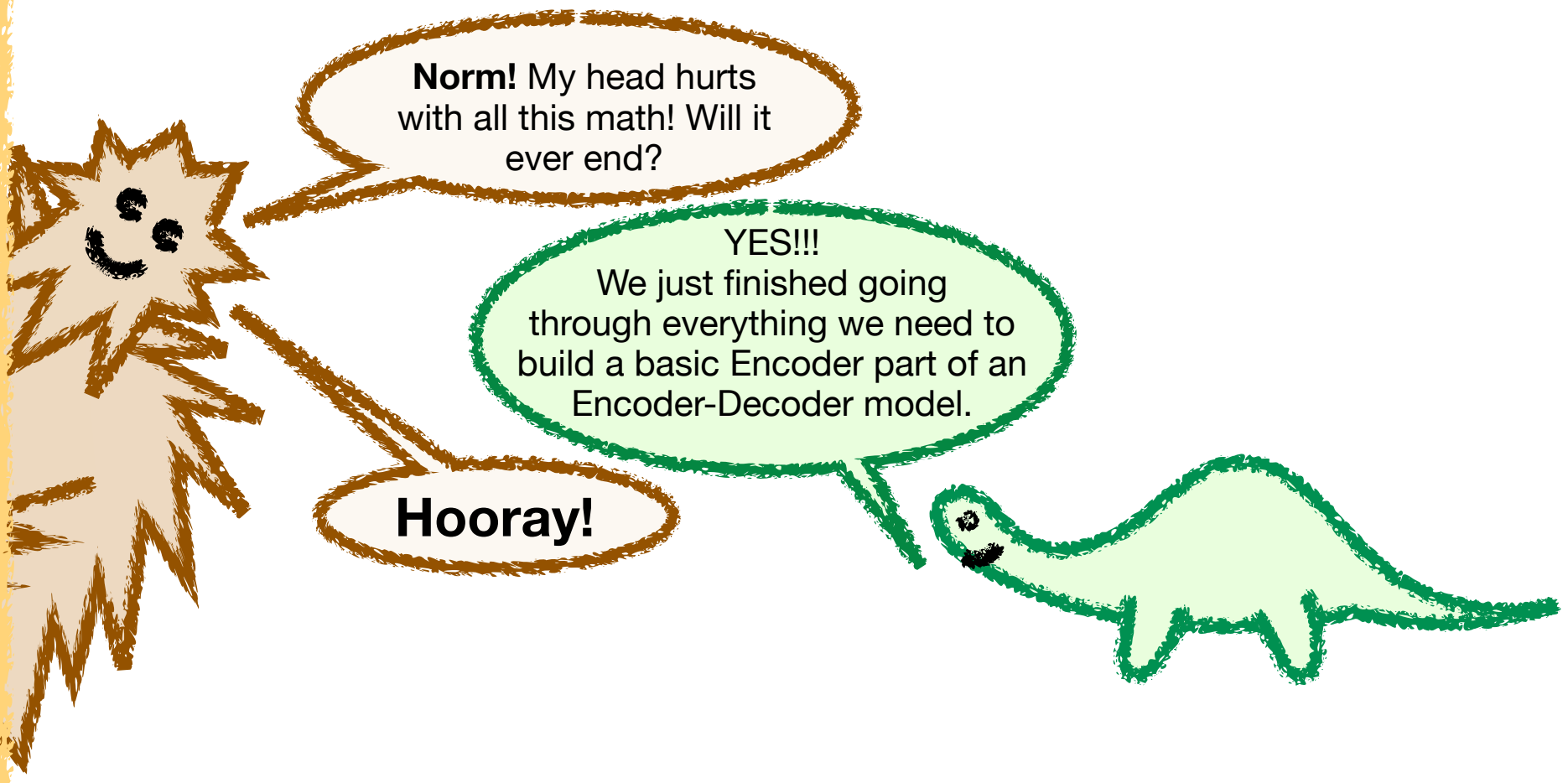


…the next thing we do is take the sum of the Word Embedding and Position Encoded values…

…and add them to the individual Self-Attention values.

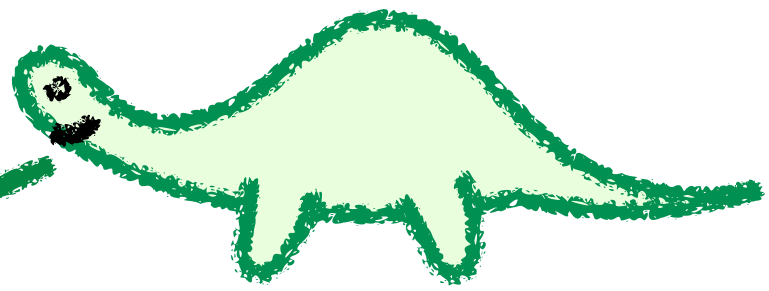


34 These bypasses are called **Residual Connections**, and they make training complex neural networks easier by allowing the Self-Attention layer to establish relationships among the input words without having to also preserve the Word Embedding and Positional Encoding information.



Transformers: Encoding Details

Norm! I'm glad we're done with the Encoder, but I've already forgotten the main ideas!

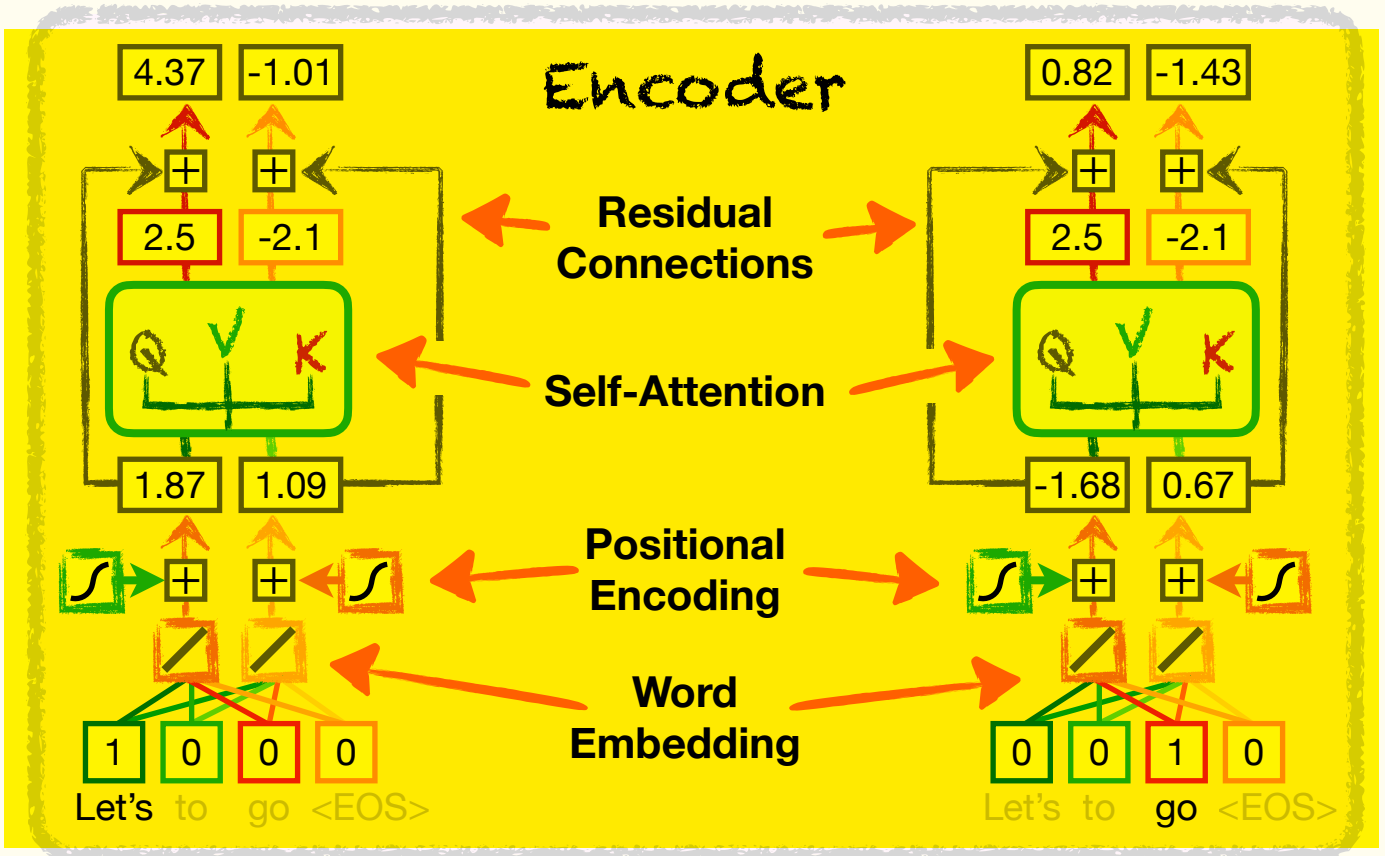


Don't worry **'Squatch**, the next step is a summary of all of the main ideas!

35 Now let's summarize the main ideas behind each step we took to build the Encoder part of our Transformer.

Word Embedding turns tokens into numbers, Positional Encoding keeps track of the order of the tokens, Self-Attention adds the relationships among the tokens, and Residual Connections make training relatively easy. Lastly, each step can be done in parallel.

BAM! Now let's learn how the Decoder works!



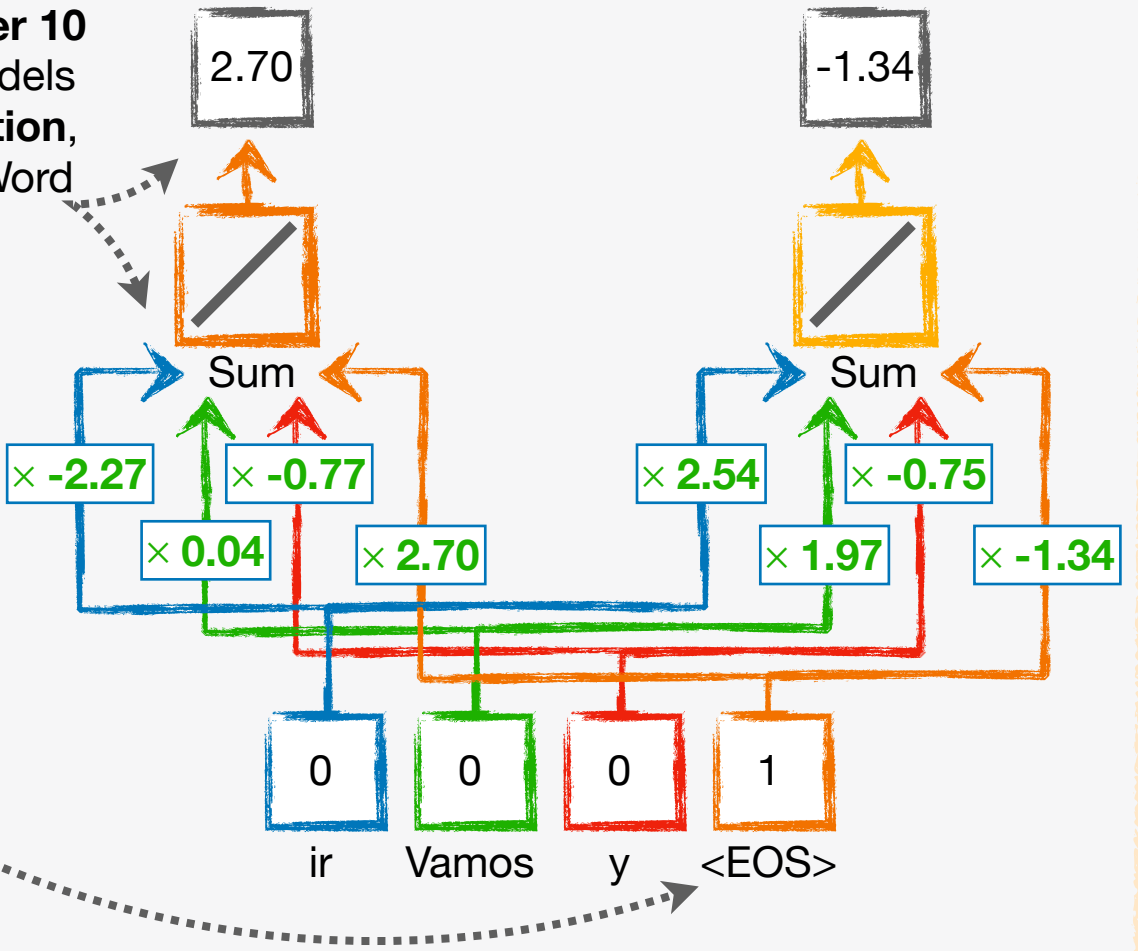
NOTE: There are lots of extra things we can add to a Transformer, and we'll talk about those at the end of this chapter. For now, however, we'll start talking about the Decoder part of our Transformer.

Transformers: Decoding Details

1 Just like we saw in **Chapter 10** on **Encoder-Decoder** models and **Chapter 11** on **Attention**, the **Decoder** starts with Word Embedding.

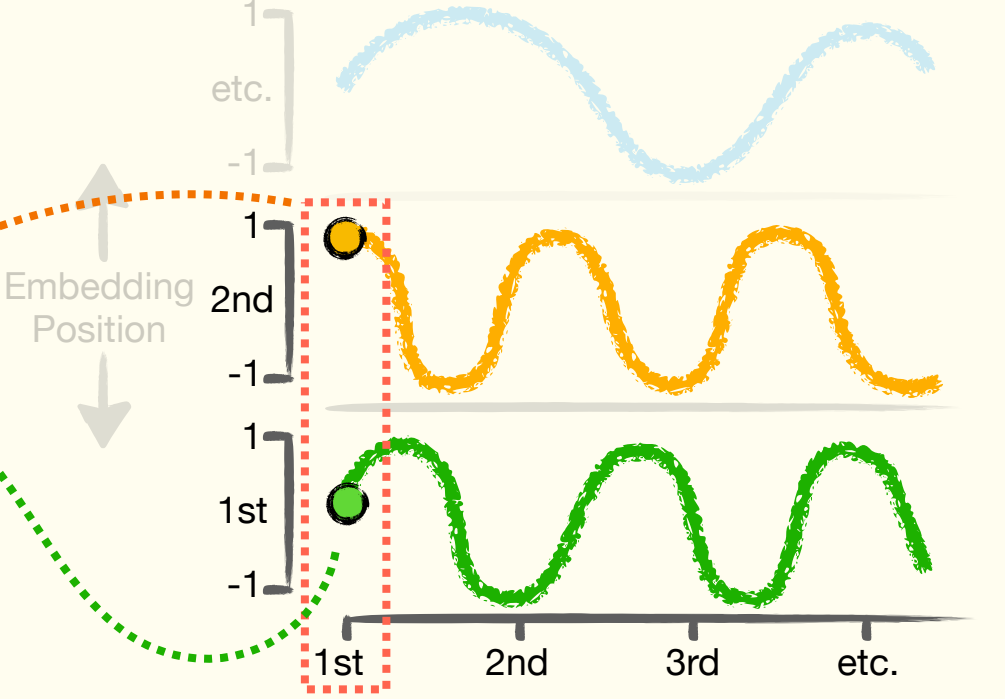
And, just like in those earlier chapters, this Embedding layer has the Spanish tokens, and we'll initialize the Decoder by starting with the **<EOS>** token*.

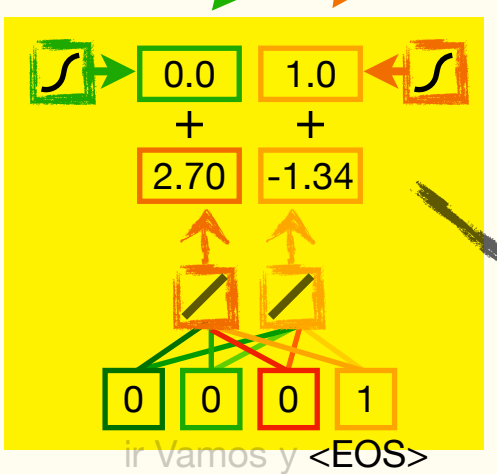
*Or whatever token we define as the one that initializes the Decoder.

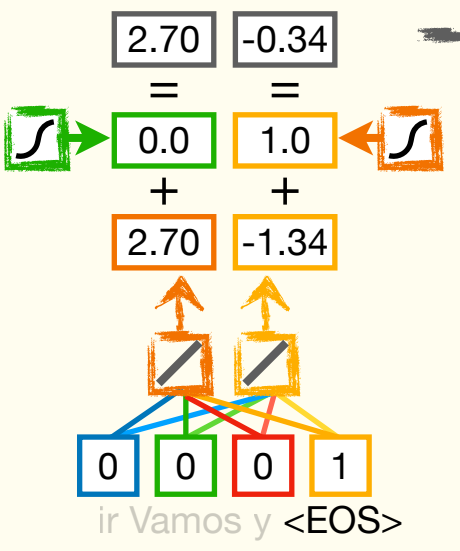


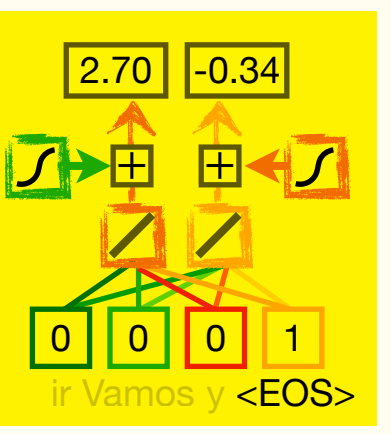
2 Then we add Positional Encoding, just like we did in the Encoder part of the Transformer.

The **<EOS>** gets the **2** embedding values from the first position because it's the first token that we're processing in the Decoder.





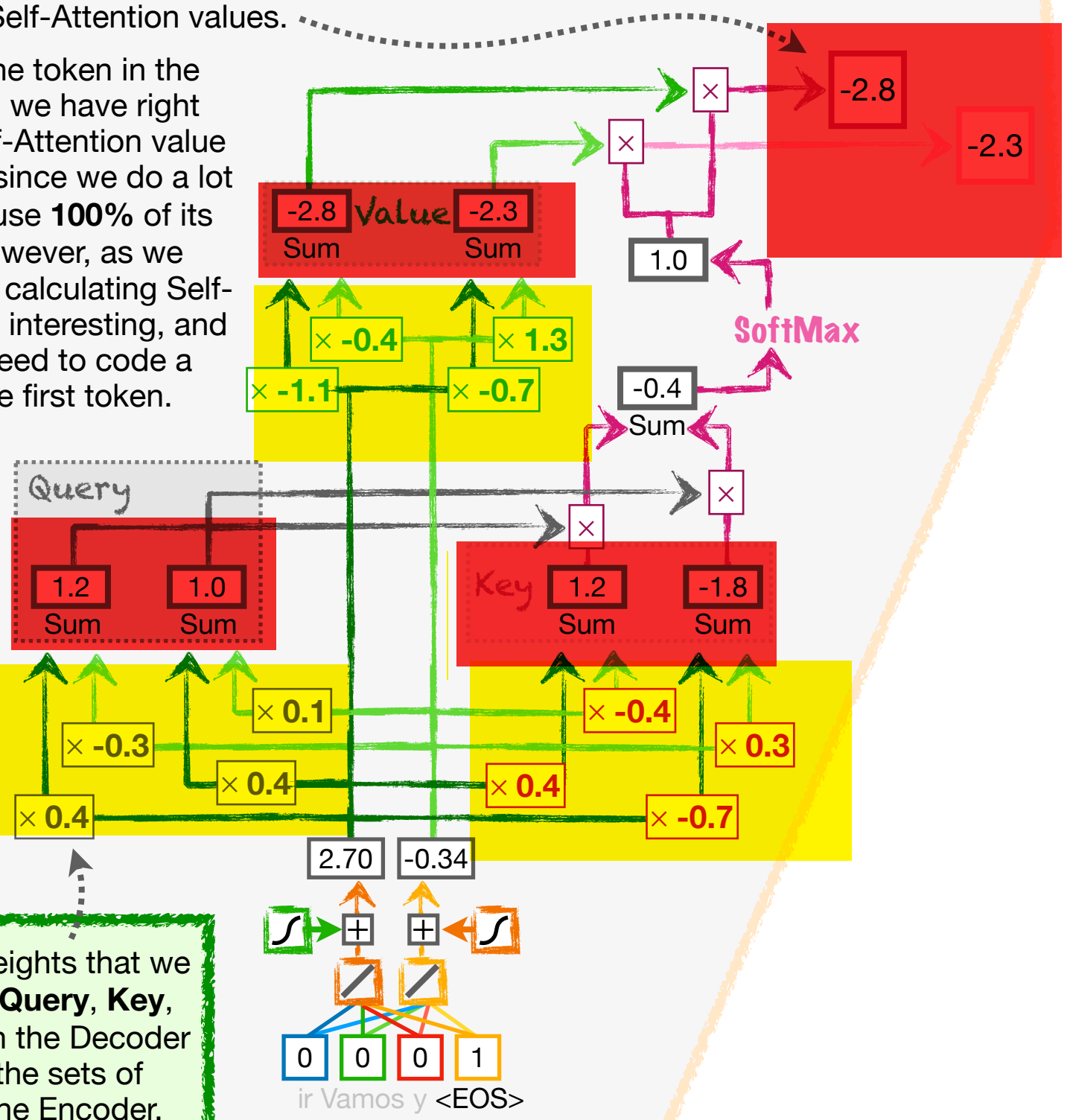




Transformers: Decoding Details

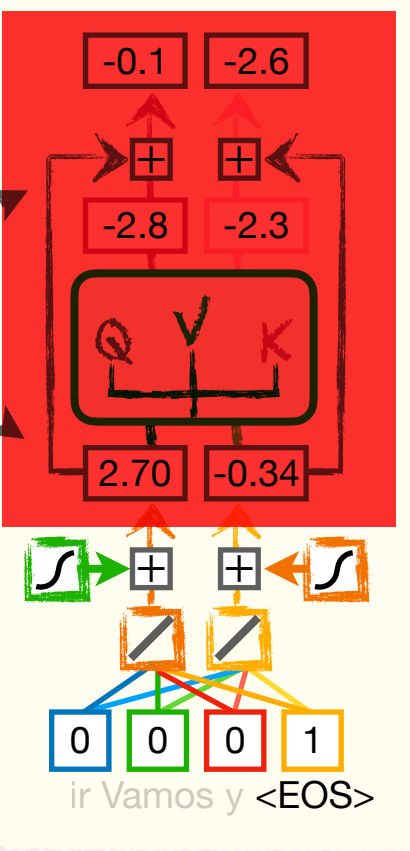
3 And just like we did with the Encoder part of the Transformer, we calculate the individual Self-Attention values.

When there's only one token in the Decoder, which is all we have right now, calculating a Self-Attention value may seem a little silly, since we do a lot of extra math just to use **100%** of its **Value** numbers. However, as we generate more tokens, calculating Self-Attention will get more interesting, and it means we don't need to code a special case for the first token.



NOTE: The sets of weights that we use to calculate the **Query**, **Key**, and **Value** numbers in the Decoder are different from the sets of weights we use in the Encoder.

4 The next step is to add Residual Connections, just like we did in the Encoder part of the Transformer.



Transformers: Decoding Details

5 Now, just like we saw in **Chapter 11** on **Attention**, the Decoder needs to keep track of all of the words in the input in order to do a good job of translating.

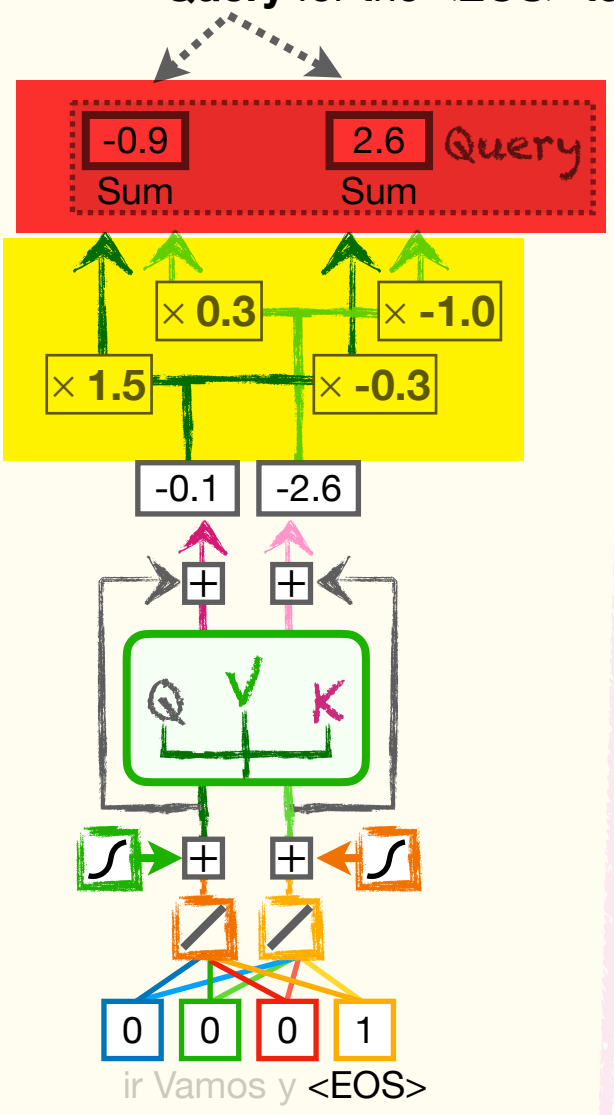
And if the input is a really long sentence, like…

Don't eat the delicious-looking and -smelling pizza.

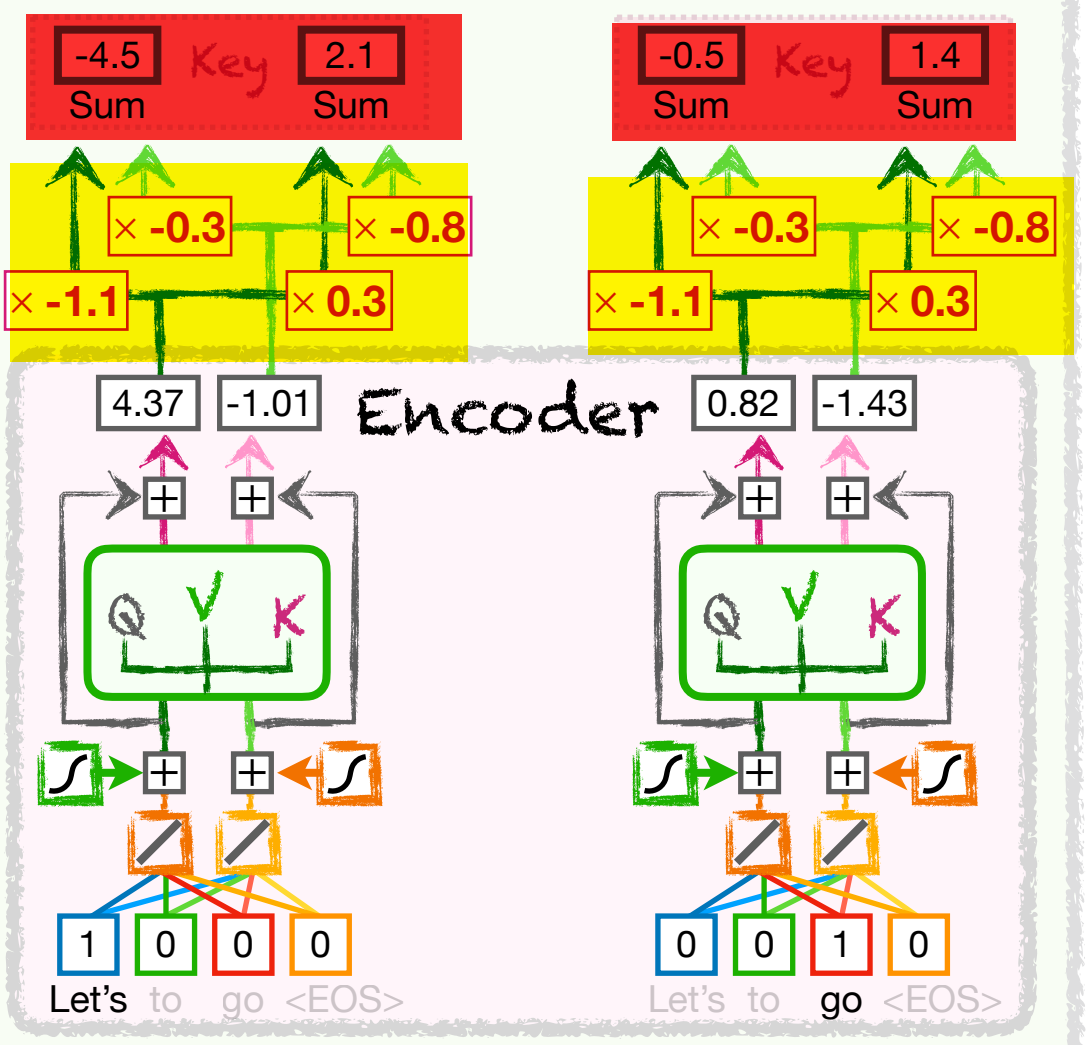
…then it's super important for the Decoder to keep track of the very first word, **Don't**.

If it doesn't, then **Don't eat the delicious-looking and -smelling pizza** turns into **Eat the delicious-looking and -smelling pizza**, which means the exact opposite! So, to keep track of all of the input words, the Decoder uses a type of **Encoder-Decoder Attention** like what we saw in **Chapter 11**.

6 Just like we did for Self-Attention, but now using a set of weights that's just for Encoder-Decoder Attention, we create **2** new values to represent the **Query** for the **<EOS>** token in the Decoder.



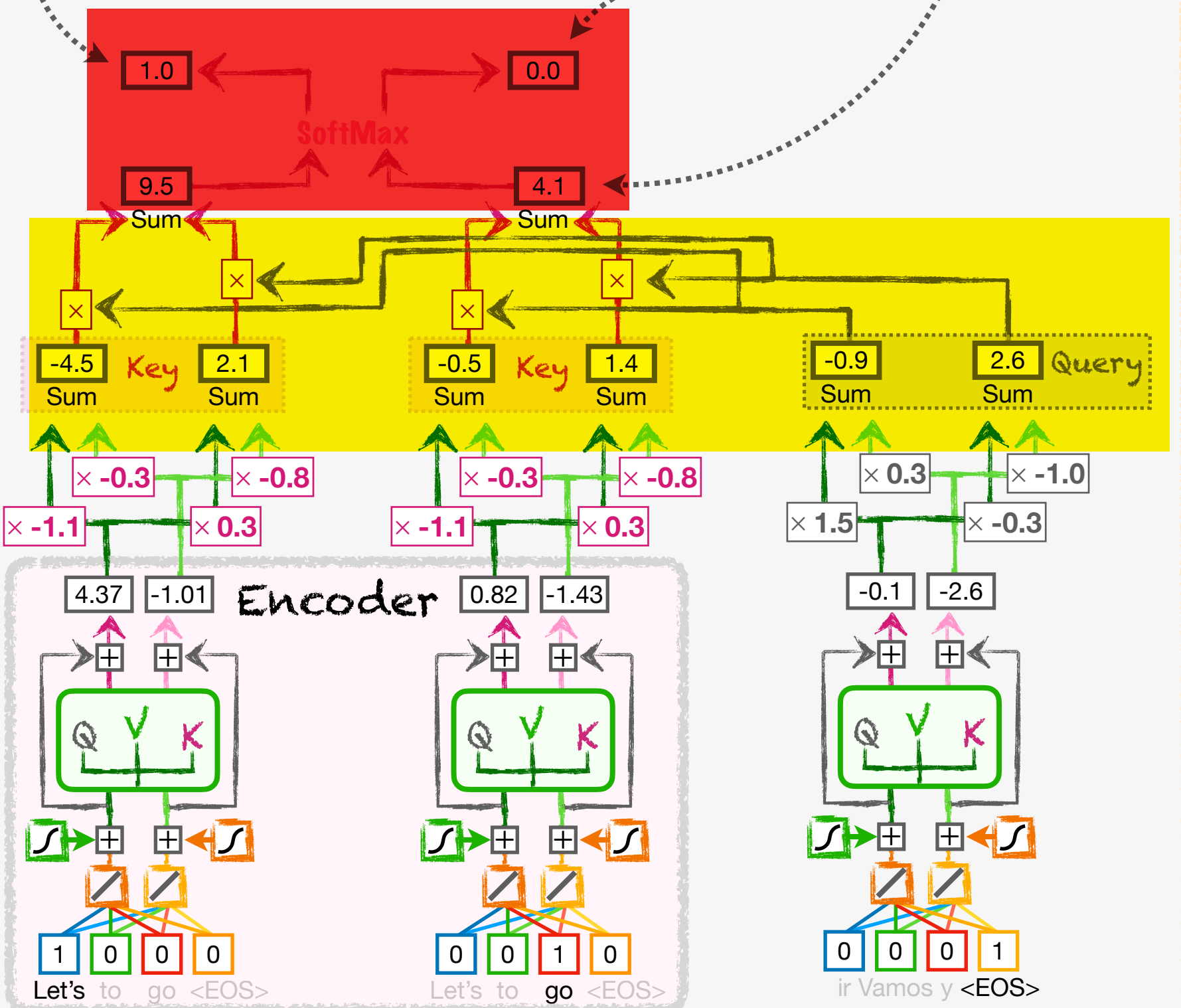
7 And we also use a set of Encoder-Decoder Attention-specific weights to create new **Keys** for **Let's** and **go**, the tokens in the Encoder, using the final values we calculated earlier.



Transformers: Decoding Details

8 Then we calculate the similarities with the **Query** for the **<EOS>** token in the Decoder and the **Keys** for each token in the Encoder by calculating the Dot Products, just like before.

Running the similarities through the SoftMax function tells us to use **100%** of the first input word, **Let's**, and **0%** of the second, **go**, when the Decoder determines what should be the first translated word.



Transformers: Decoding Details

9 Then, using a different set of Encoder-Decoder Attention-specific weights, we create new **Values** for the tokens in the Encoder…

…which are then scaled by the SoftMax percentages…

…and then the pairs of scaled **Values** are added together to get the individual Encoder-Decoder Attention values for the **<EOS>** token in the Decoder.

SoftMax relative to **<EOS>** in the **Decoder**

Let's 1.0 6.0 Value 3.1 Sum Sum

go 0.0 2.6 Value 1.2 Sum Sum

+ 6.0 3.1

Bam!

6.0 3.1

× -1.2 × -0.5 × 1.1 × 0.6

4.37 -1.01

Encoder

0.82 -1.43

Q V K

1 0 0 0

Let's to go <EOS>

0 0 1 0

Let's to go <EOS>

-0.1 -2.6

0 0 0 1

ir Vamos y <EOS>

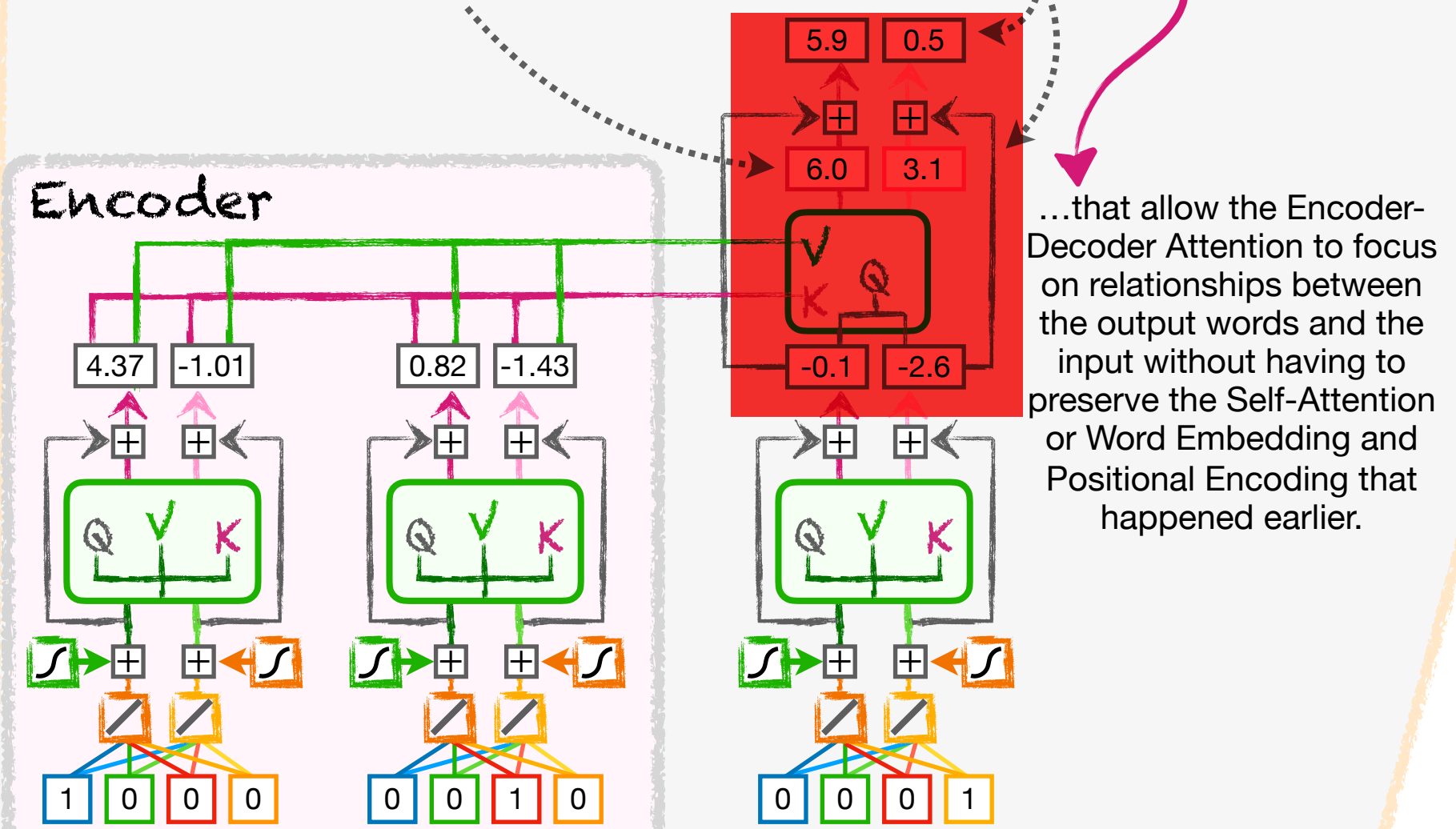
Although there wasn't a summary, we just got through another major step. So I'll eat another snack.

Transformers: Decoding Details

10 Now that we've calculated Encoder-Decoder Attention for the **<EOS>** token…

…we add another set of Residual Connections…

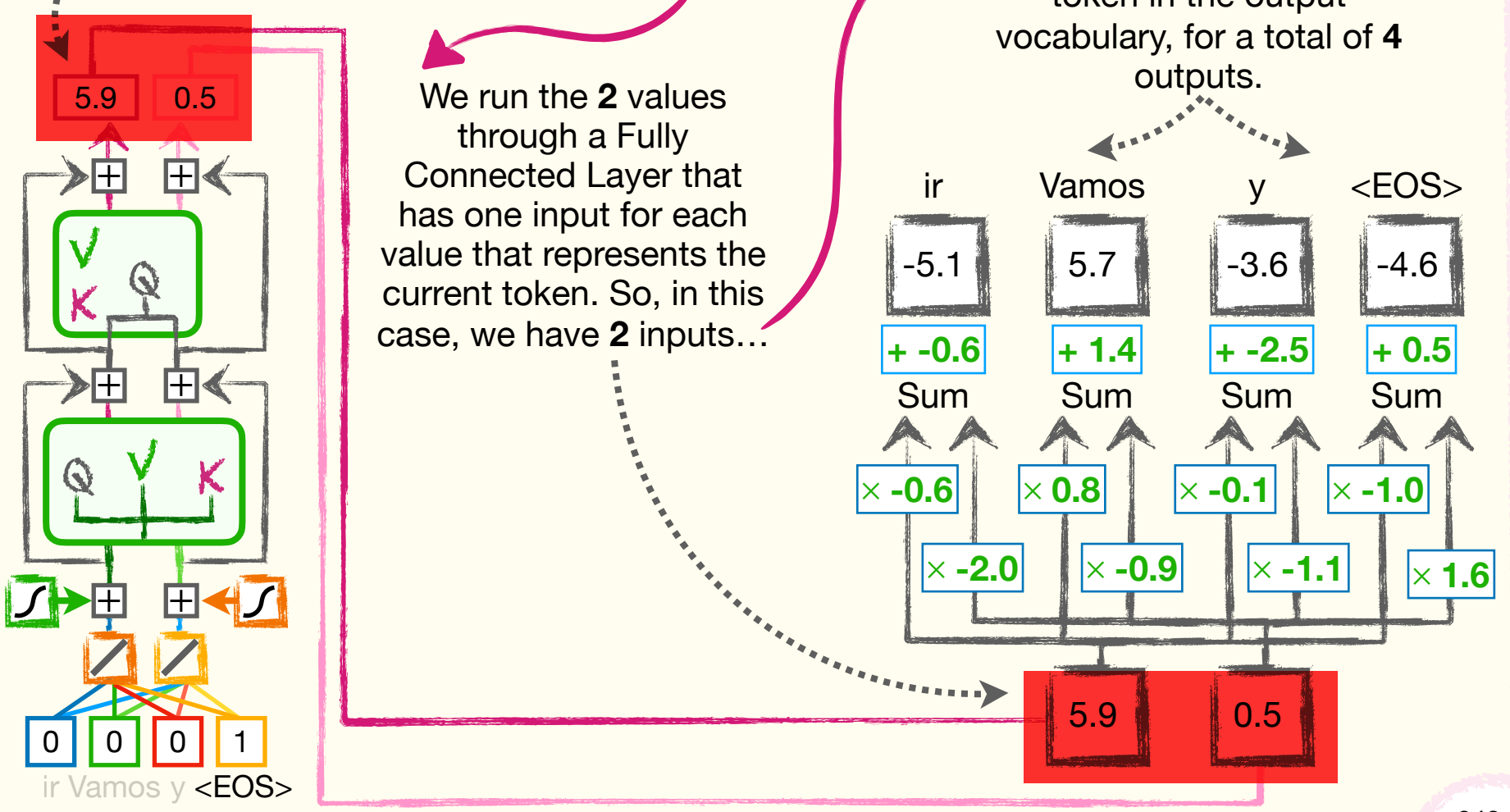
…that allow the Encoder-Decoder Attention to focus on relationships between the output words and the input without having to preserve the Self-Attention or Word Embedding and Positional Encoding that happened earlier.



11 Now we'll use the values that represent the **<EOS>** token in the Decoder, **5.9** and **0.5**, to select one of the **4** output tokens, **ir**, **Vamos**, **y**, or **<EOS>**.

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 **1** output for each token in the output vocabulary, for a total of **4** outputs.

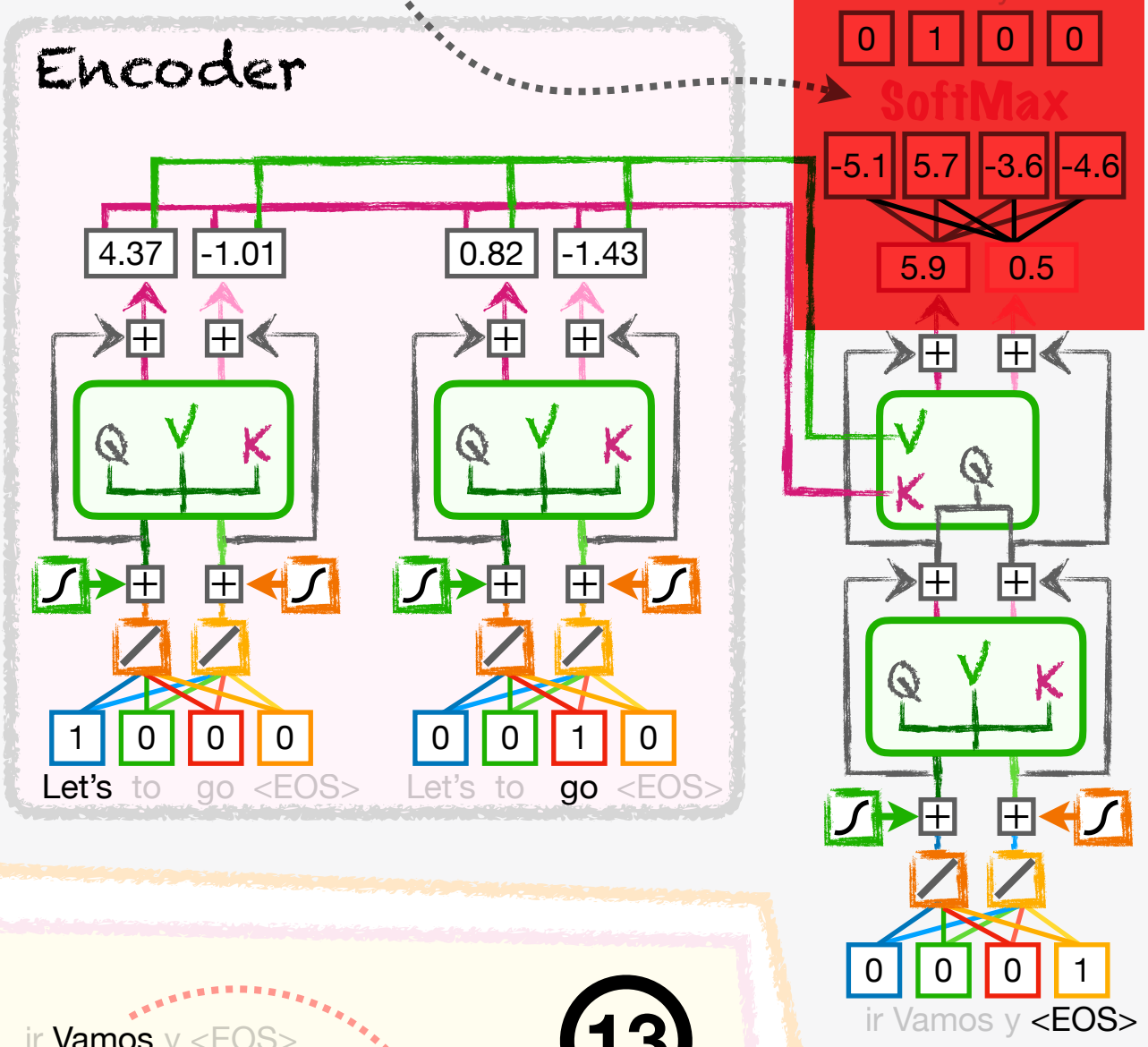


Transformers: Decoding Details

12 Lastly, we run the outputs from the Fully Connected Layer through the SoftMax function…

…to generate the first output token, **Vamos**.

BAM!!!

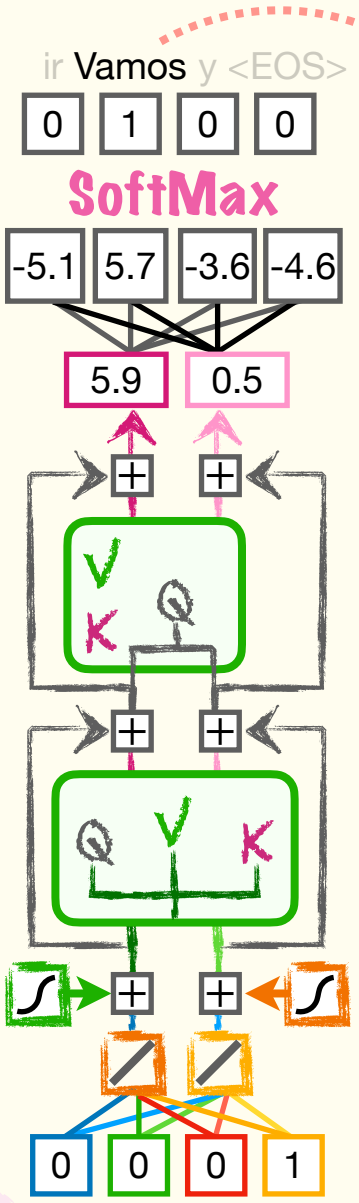


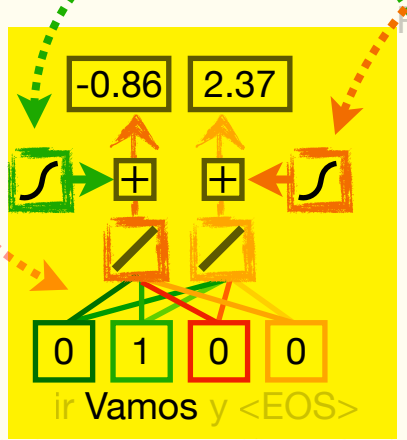
Vamos is the correct translation of the input phrase **Let's go**, but just like we saw with the Encoder-Decoder models in **Chapters 10** and **11**, we're not done generating output tokens until we generate an **<EOS>** token.

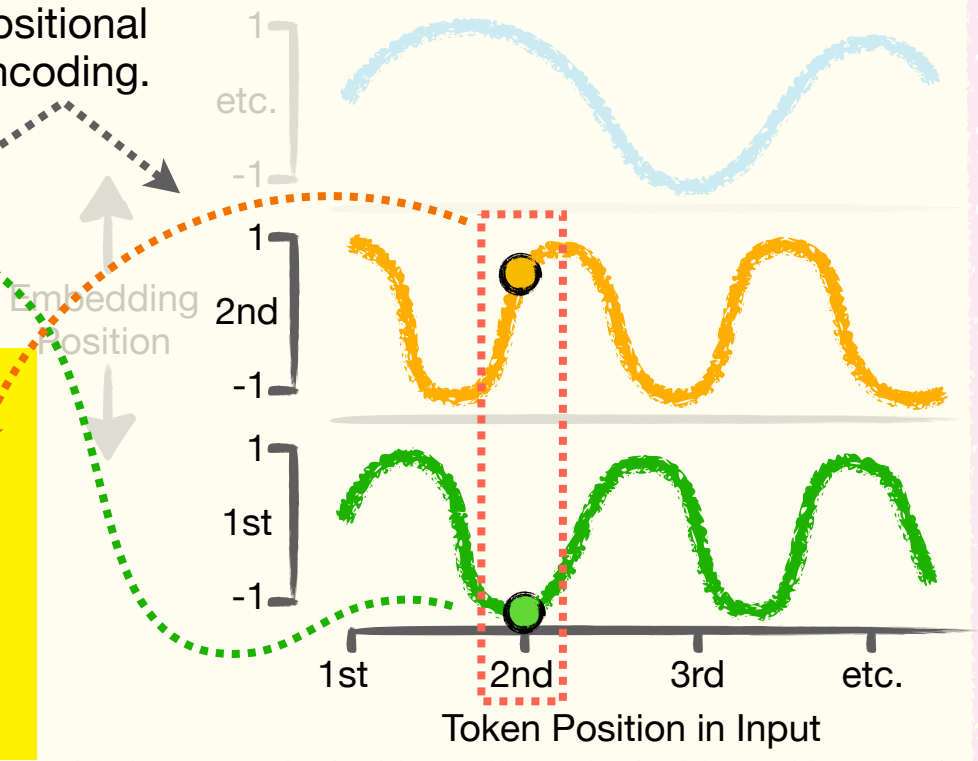
NOTE: Just like we saw in **Chapter 10**, we don't just have to generate the token that has the largest value coming out of the SoftMax function. Instead, we can use the output from the SoftMax function as a probability distribution to randomly select a token to generate.

13 Since we didn't generate the **<EOS>** token, we plug the newly generated token, **Vamos**, into the Decoder's Embedding Layer…

…and add Positional Encoding.

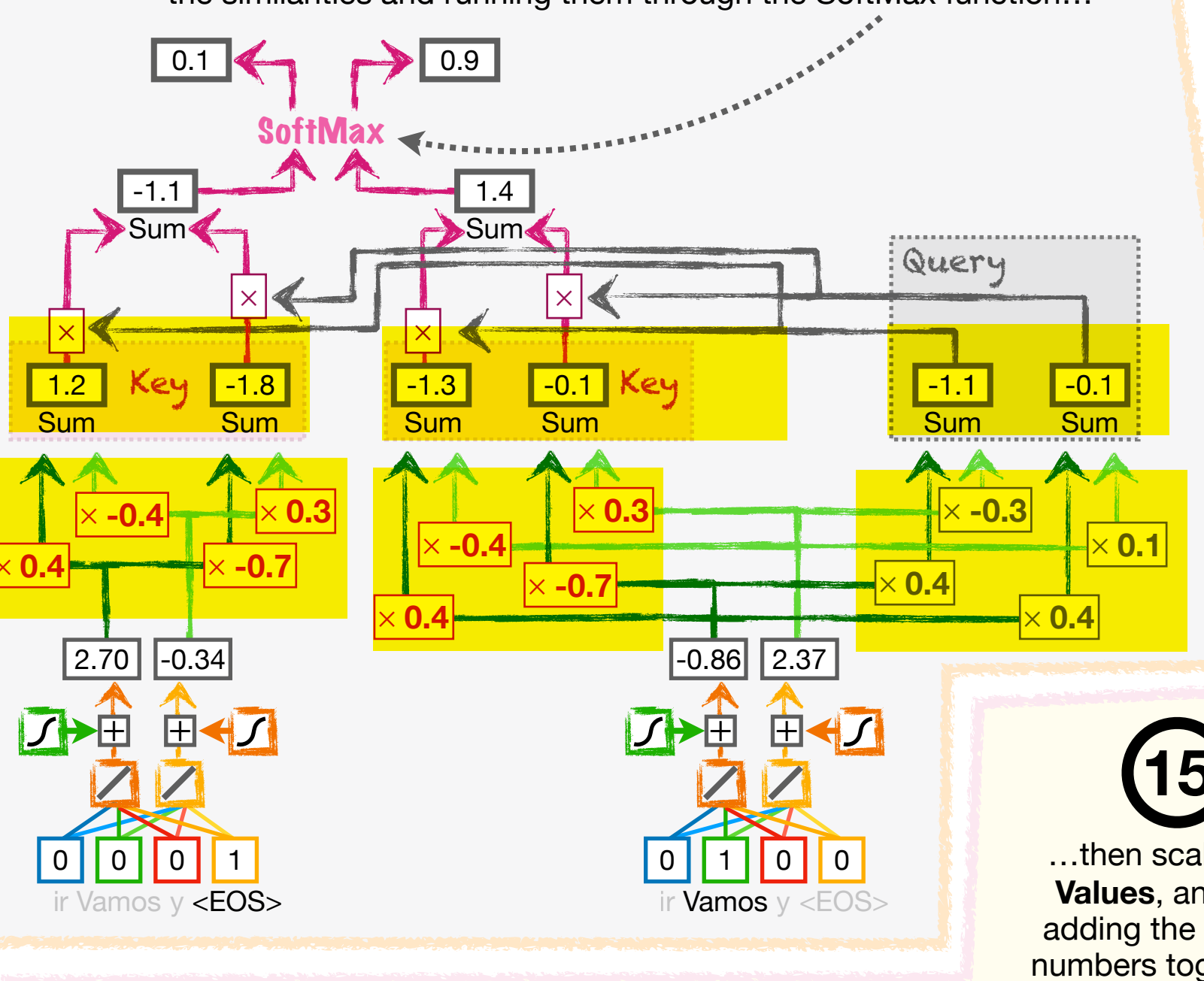




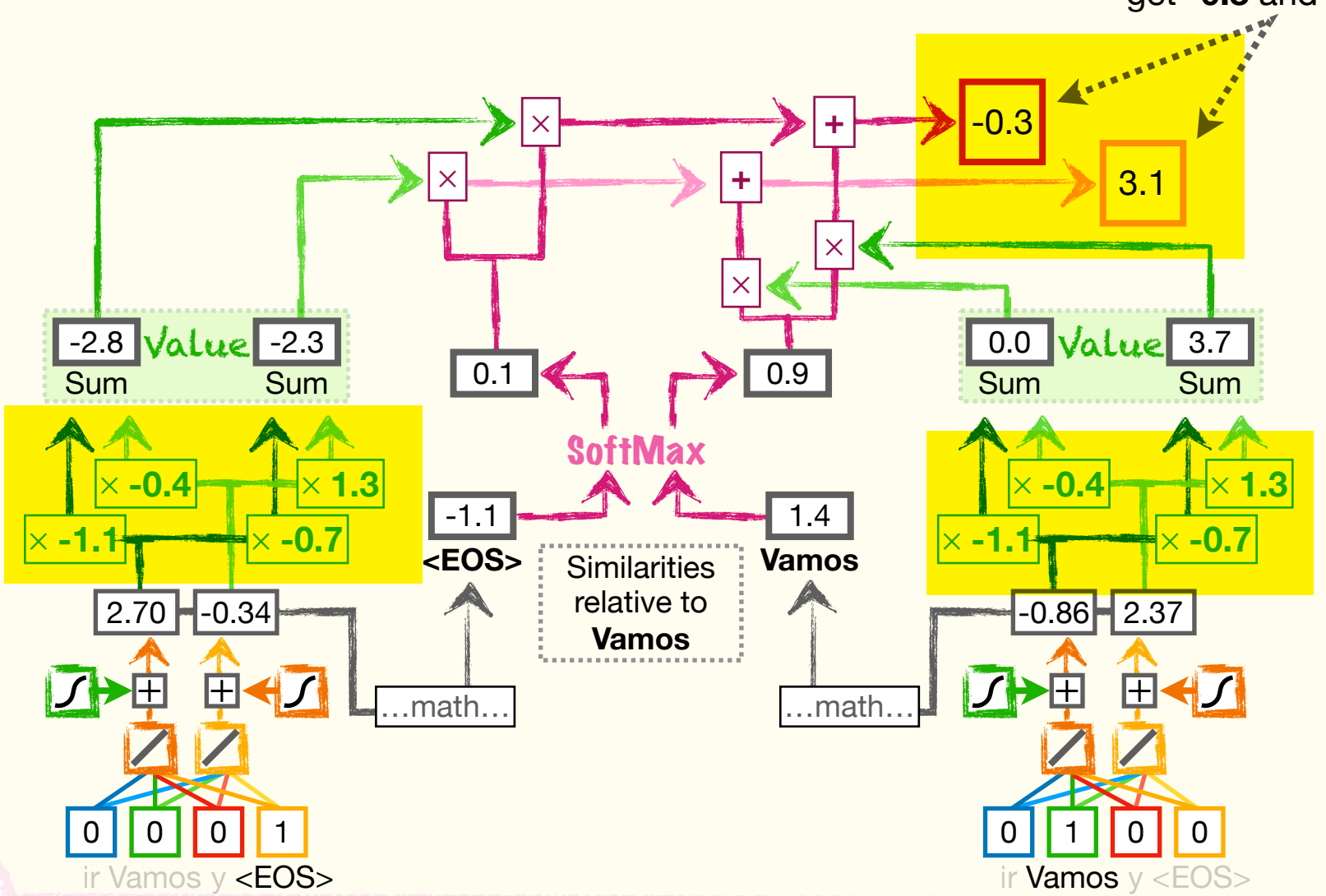


Transformers: Decoding Details

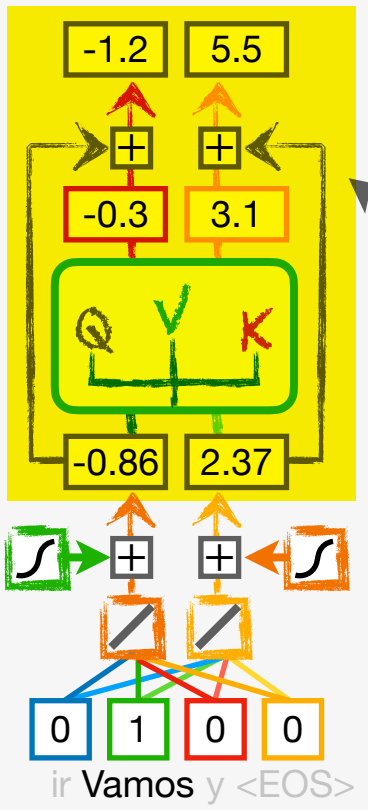
14 Then we calculate Self-Attention between the two tokens in the Decoder, **<EOS>** and **Vamos**, relative to **Vamos**, by first calculating the similarities and running them through the SoftMax function…



15 …then scaling the **Values**, and then adding the pairs of numbers together to get **-0.3** and **3.1**.



Transformers: Decoding Details



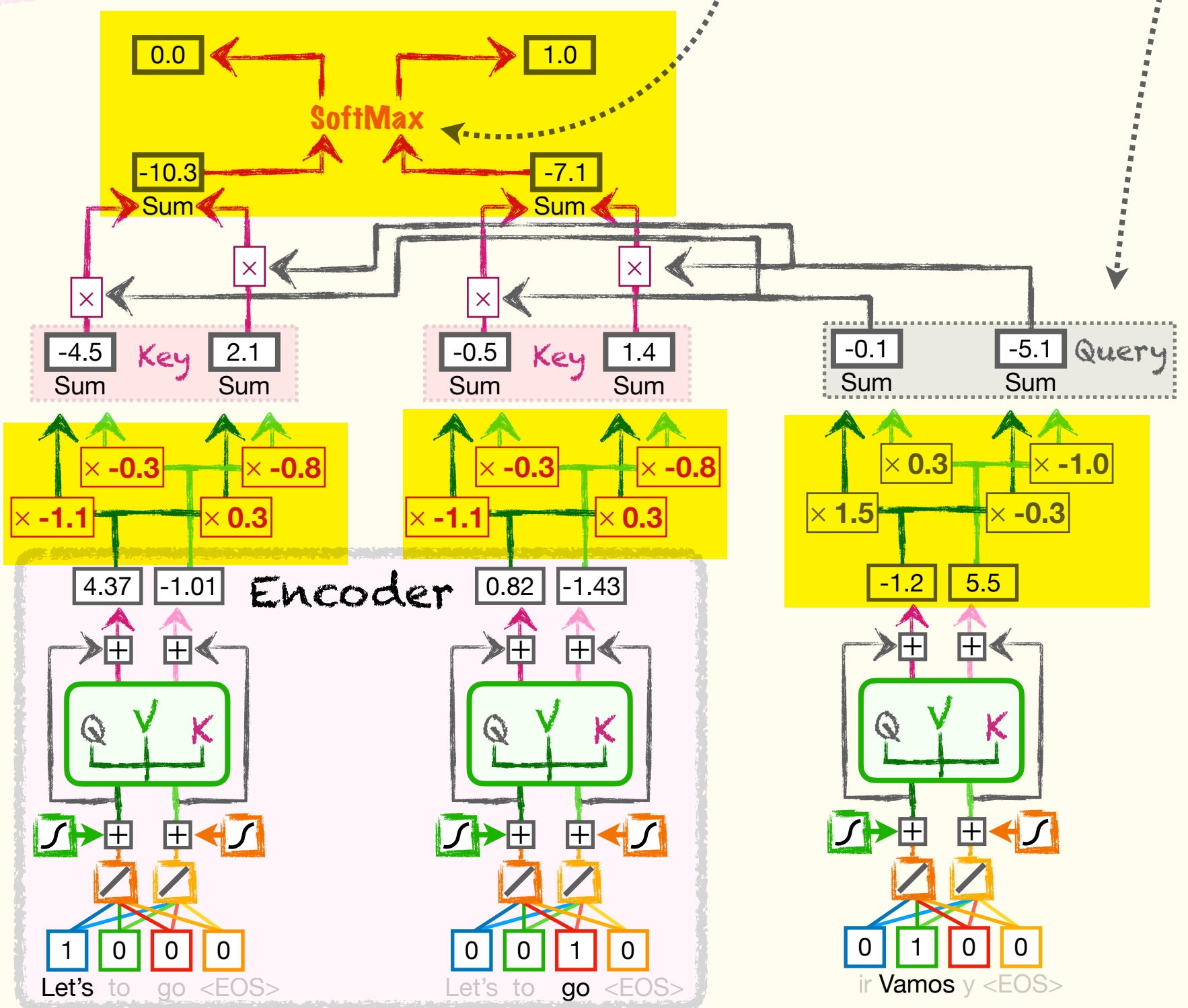
16

Then we add the Residual Connections.

17

Then we calculate the Encoder-Decoder Attention using a **Query** that we calculate from the numbers that represent **Vamos** in the Decoder…

…and the **Keys** for **Let's** and **go** in the Encoder that we calculated before. First, we calculate the similarities. Then we run the similarities through the SoftMax function…



Transformers: Decoding Details

18 Next, we bring back the Encoder-Decoder Attention **Values** that we calculated earlier for **Let's** and **go** and scale them.

Lastly, we add the products together to get the individual Encoder-Decoder Attention values for **Vamos**.

SoftMax relative to **Vamos** in the **Decoder**

Let's

go

0.0

1.0

6.0 Value 3.1

Sum Sum

2.6 Value 1.2

Sum Sum

2.6

1.2

× -1.2 × -0.5 × 1.1 × 0.6

4.37 -1.01

× -1.2 × -0.5 × 1.1 × 0.6

0.82 -1.43

Encoder

V K Q

2.6 1.2

-1.2 5.5

Q V K

1 0 0 0

Let's to go <EOS>

Q V K

0 0 1 0

Let's to go <EOS>

Q V K

0 1 0 0

ir Vamos y <EOS>

NOTE: The weights for calculating the **Values** are the same weights we used when we calculated Encoder-Decoder Attention for the token that initialized the Decoder, **<EOS>**, so the **Values** are also the same.

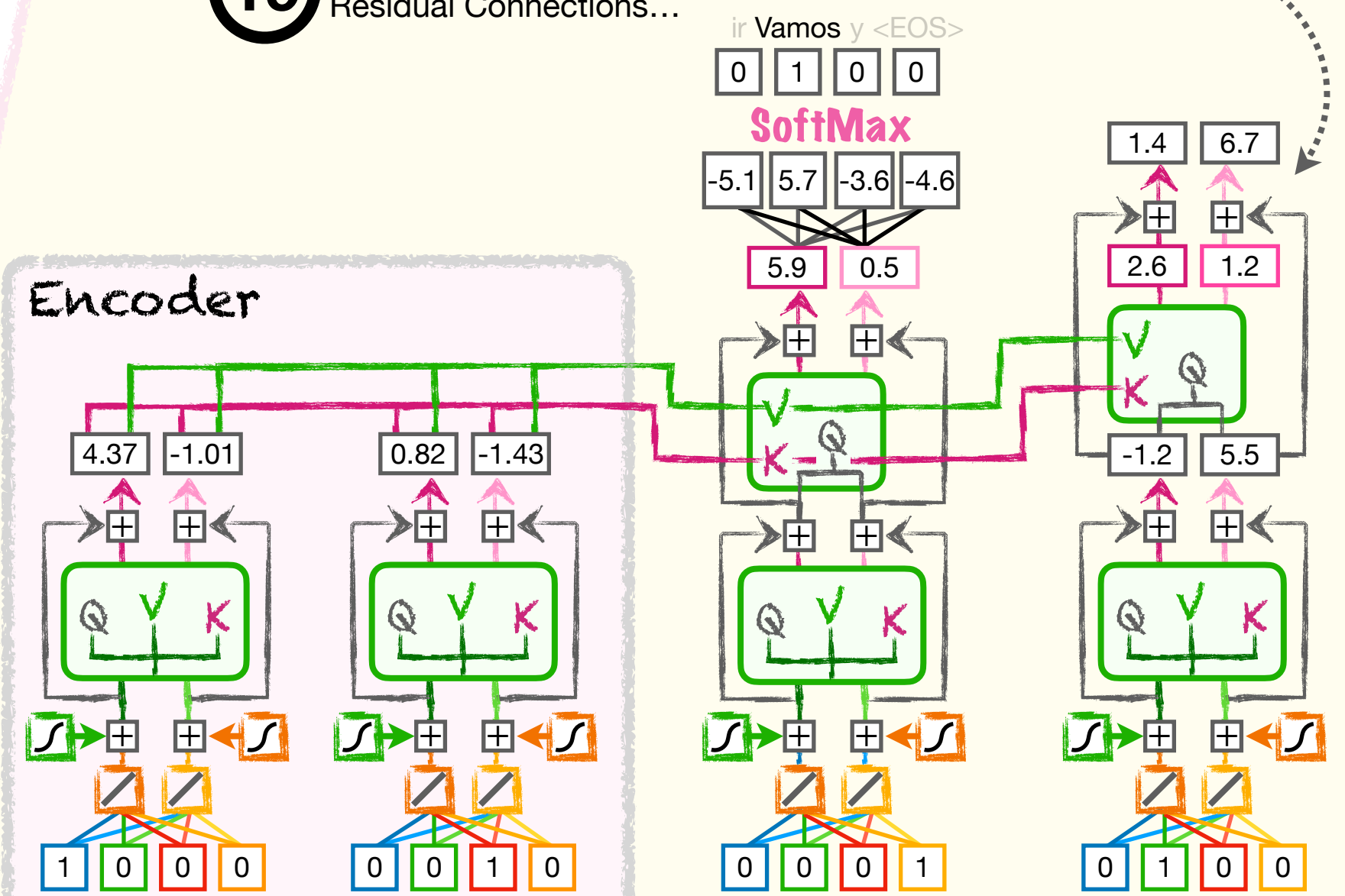
Norm, are we almost done? My head is starting to hurt again.

Bam!

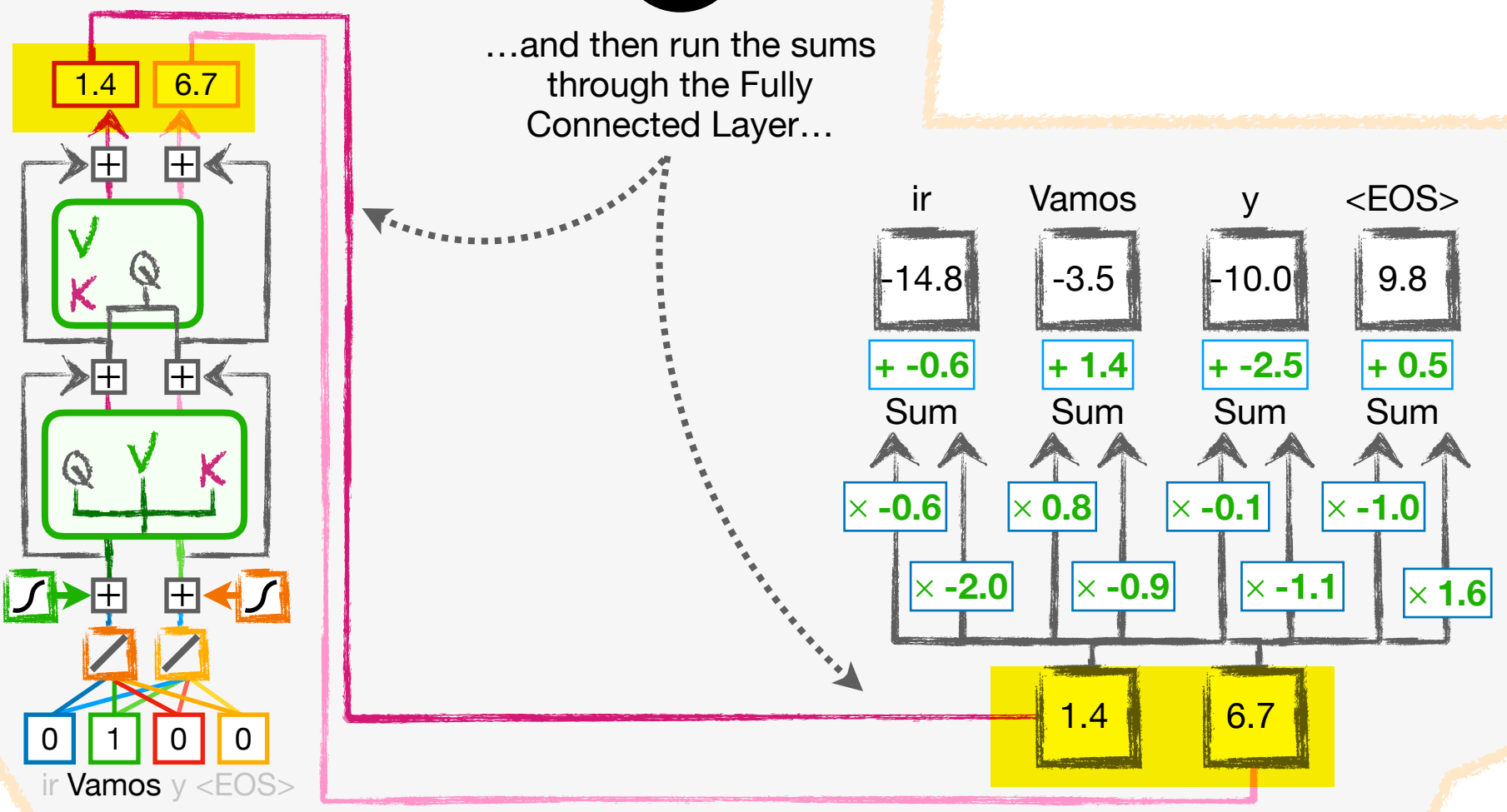
We're almost done, '**Squatch**!

Transformers: Decoding Details

19 Now we add the Residual Connections…



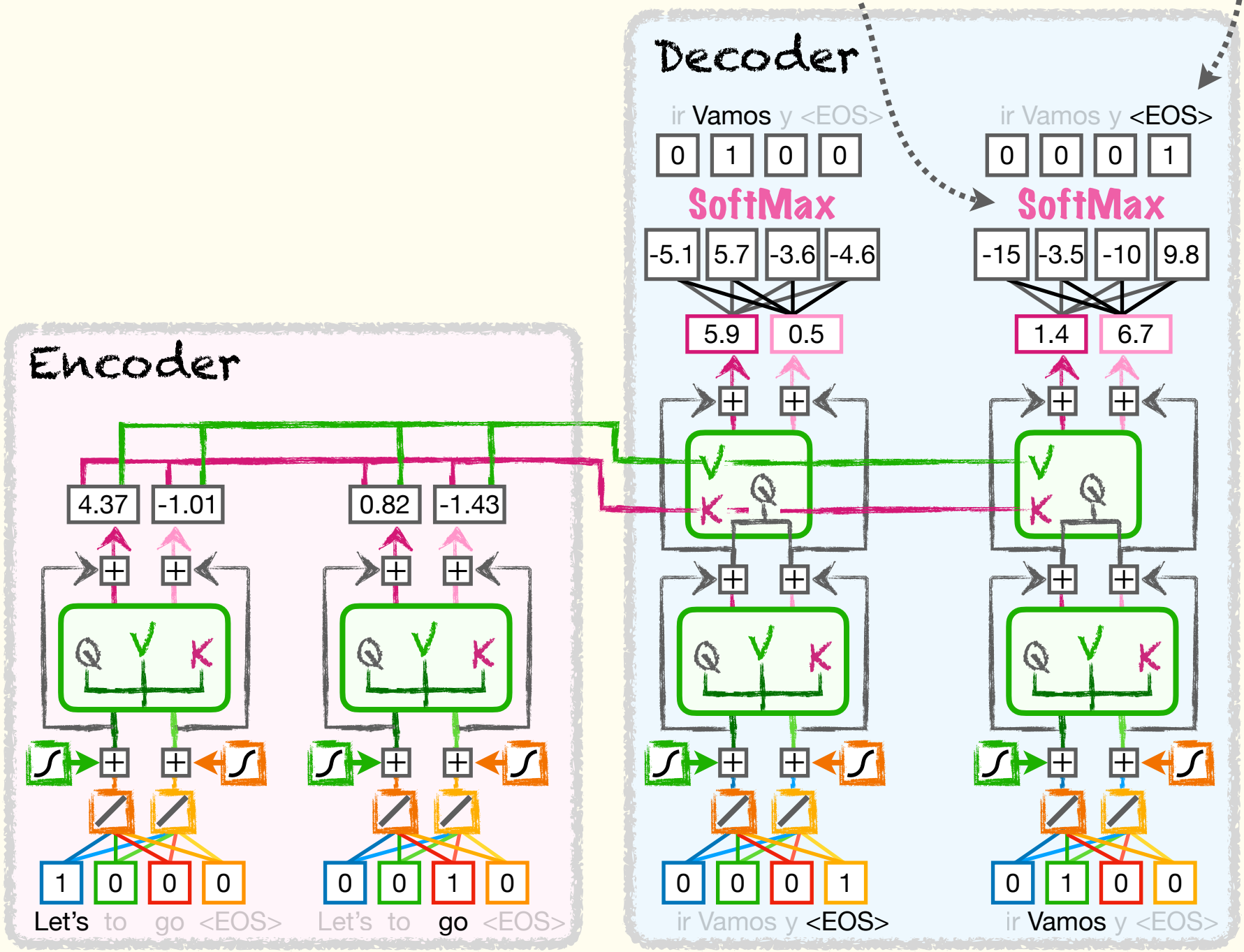
20 …and then run the sums through the Fully Connected Layer…



Transformers: Decoding Details

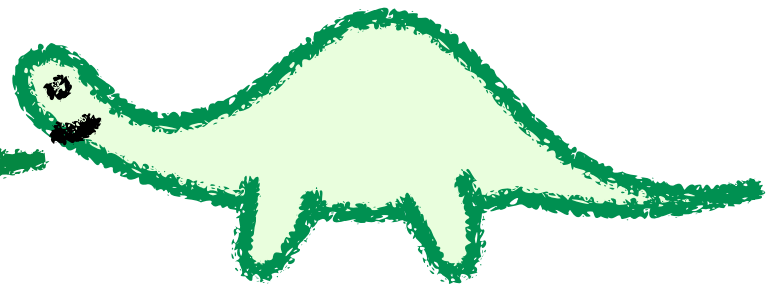
21 Lastly, we run the outputs from the Fully Connected Layer through the SoftMax function to generate the next token…

…and we get the **<EOS>** token, which means we're done generating tokens and we're done decoding.



TRIPLE BAM!!!

Now that we understand the main ideas of how Encoder-Decoder Transformers work, let's talk about a few extra things we can add to them.



Great! But first I'm going to eat some snacks!

NOTE: All of the extra things that can be added to the **Transformers** mentioned in this chapter can also be added to the **Decoder-Only** and **Encoder-Only Transformers** described in **Chapters 13** and **14**.