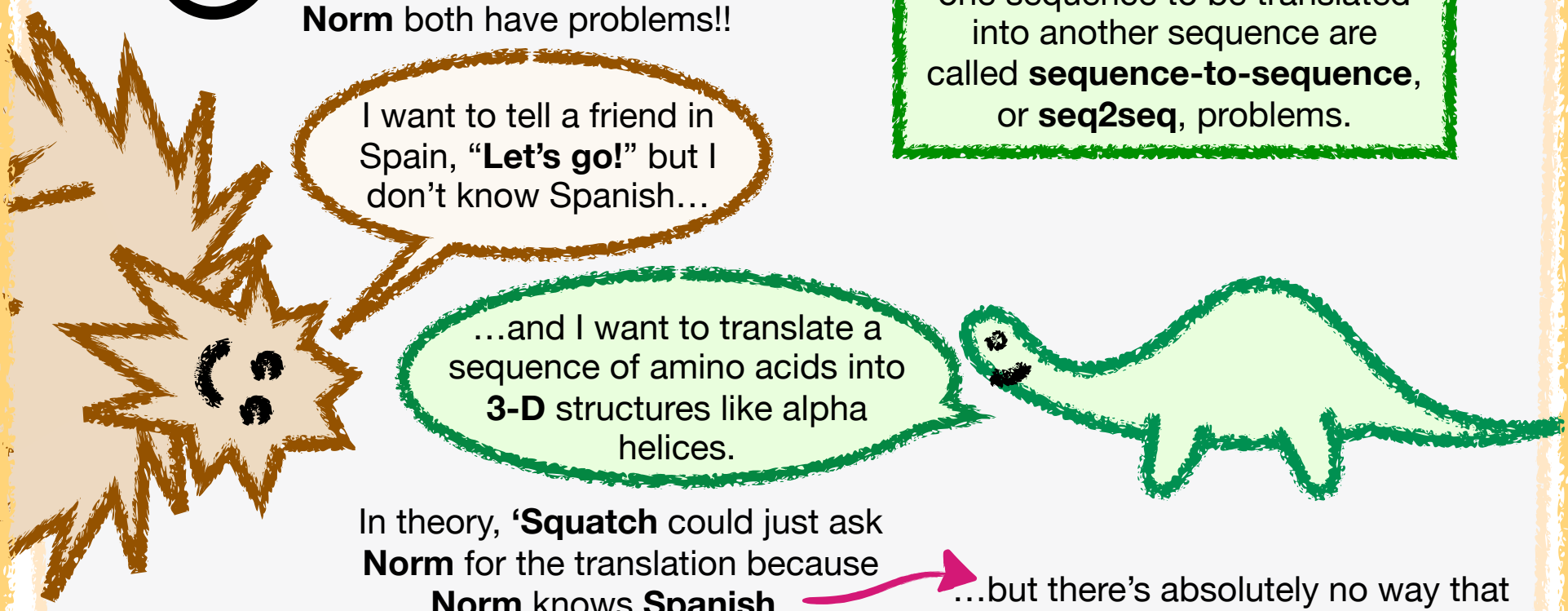
Language Translation with Seq2seq and Encoder-Decoder Models!!!

Encoder-Decoder Models: Main Ideas

1 **A Problem:** In the introduction to this chapter, **'Squatch** and **Norm** both have problems!!

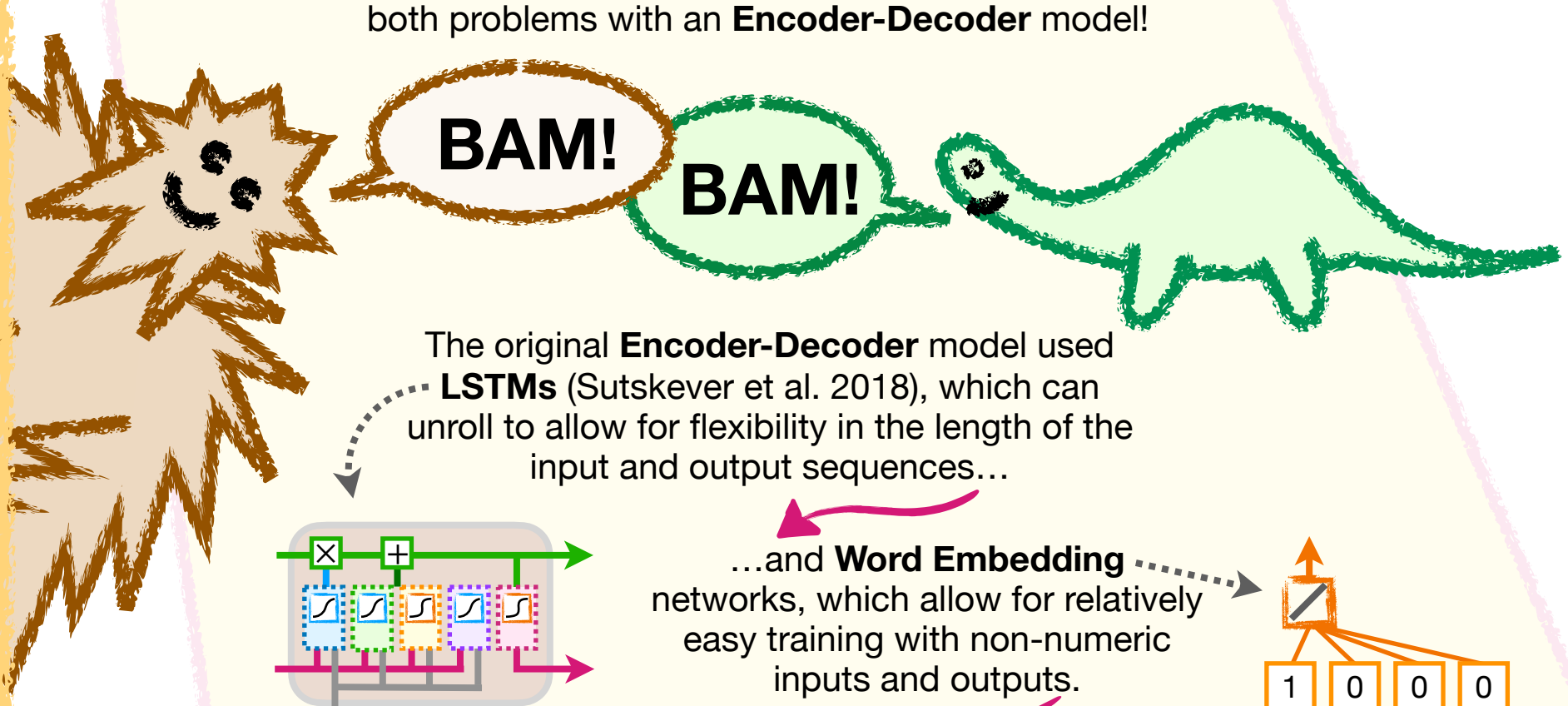
NOTE: Problems that require one sequence to be translated into another sequence are called **sequence-to-sequence**, or **seq2seq**, problems.



In theory, **'Squatch** could just ask **Norm** for the translation because **Norm** knows **Spanish**…

…but there's absolutely no way that **'Squatch** could help out **Norm**, so what can we do???

2 **A Solution:** Because both **'Squatch** and **Norm** have sequences of one type of thing that need to be translated into sequences of another type of thing, we can solve both problems with an **Encoder-Decoder** model!



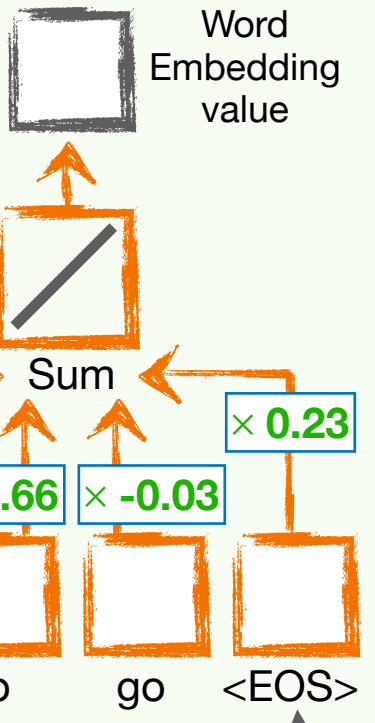
The original **Encoder-Decoder** model used **LSTMs** (Sutskever et al. 2018), which can unroll to allow for flexibility in the length of the input and output sequences…

…and **Word Embedding** networks, which allow for relatively easy training with non-numeric inputs and outputs.

To illustrate how to solve a **sequence-to-sequence** problem using a **seq2seq Encoder-Decoder** model, let's create one that translates English phrases into Spanish phrases.

Let's go! → ¡Vamos!

Encoder-Decoder Models: Details



1 Since we're building an **Encoder-Decoder** model to translate English phrases into Spanish, the input to our model will be the simple English phrase **Let's go**.

However, because we we can't just jam words into a neural network, an Encoder-Decoder model uses a Word Embedding network to convert the words into numbers.

In this super simple example, the English vocabulary for the Encoder part of our Encoder-Decoder model only has **3** words, **Let's**, **to**, and **go**. These three words allow us to use the phrases **Let's go** and **to go** as inputs to the model.

The vocabulary also has the **<EOS>** symbol, which stands for **End of Sentence** or **End of Sequence**.

In this example, we're just creating **1** embedding value per word or symbol. However, in practice, people often create hundreds and sometimes even thousands of embedding values per word or symbol.

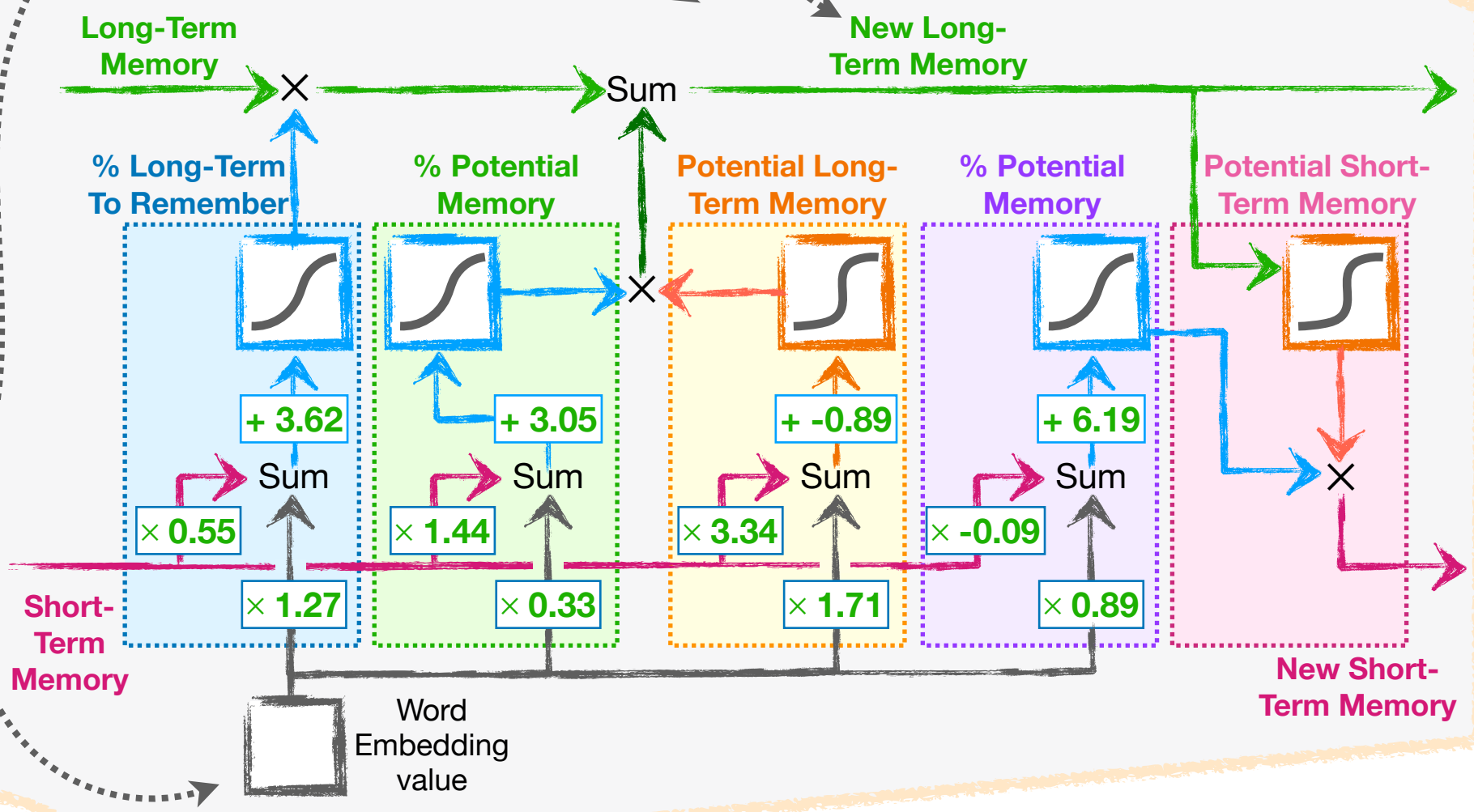
TERMINOLOGY ALERT!!!

Because vocabularies often contain a mix of words, pieces of words, letters, punctuation, and symbols, we refer to the individual elements in a vocabulary as **Tokens**.

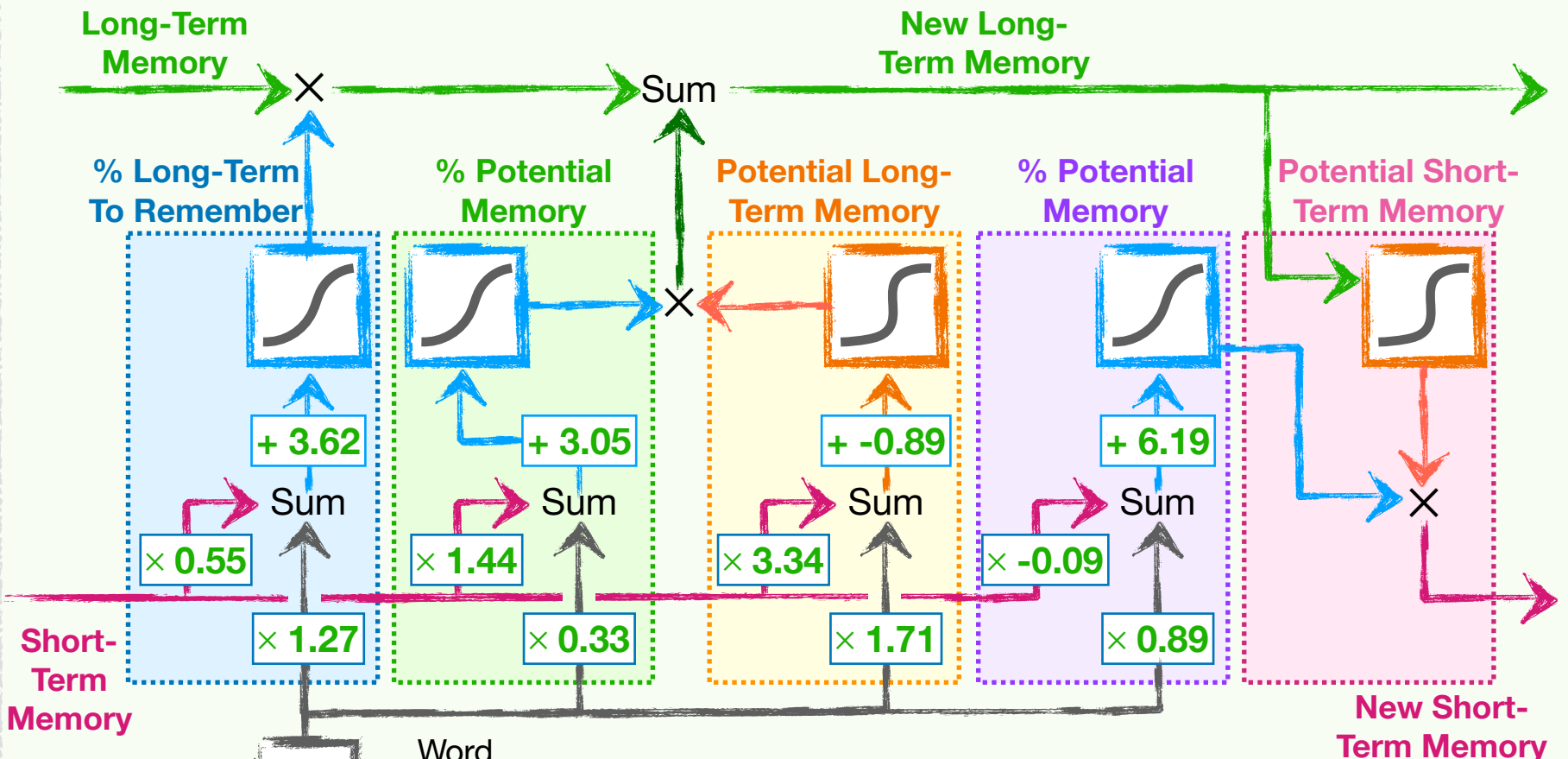
Also, when used as a part of a larger network, a **Word Embedding** network is called a **Word Embedding Layer** or just an **Embedding Layer**.

NOTE: In all of the examples that use Word Embedding, we're intentionally omitting tokens that represent punctuation or alternative capitalization. However, please be aware that these tokens are usually included and used in the same ways described here.

2 Now that we have an Embedding Layer for our input vocabulary, we can plug its output, a Word Embedding value, into the input of an LSTM.

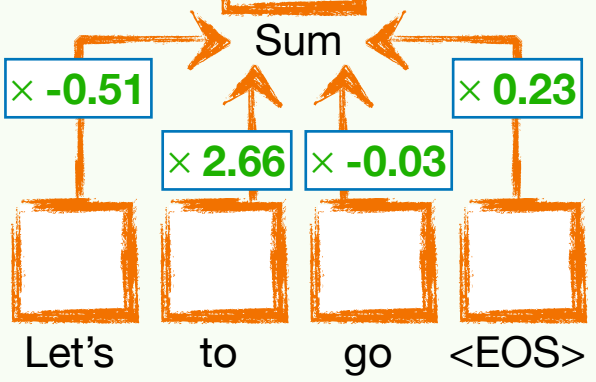


Encoder-Decoder Models: Details



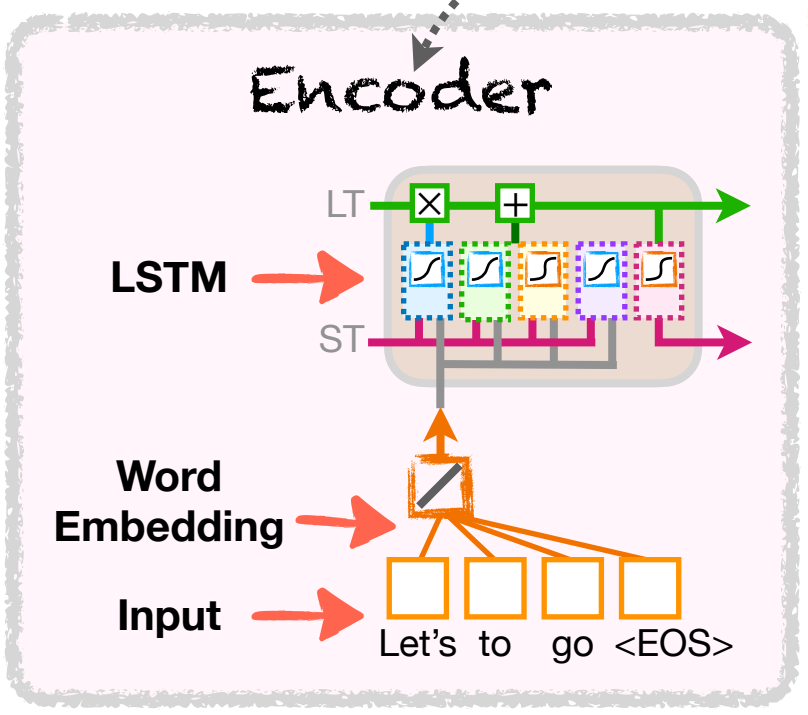
3

Thus, the Encoder part of an Encoder-Decoder model consists of an Embedding Layer for the input vocabulary, which, in this case, is for simple English phrases…



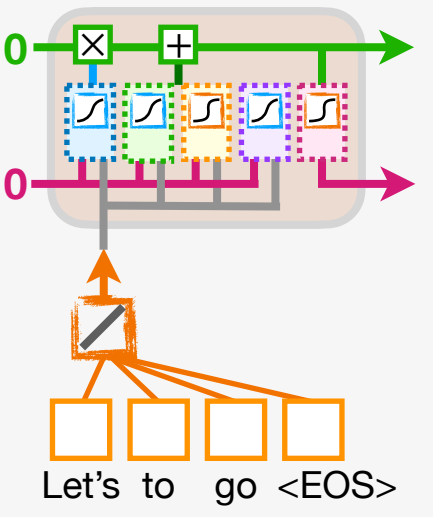
…connected to an LSTM.

Because the Encoder part of our Encoder-Decoder model takes up a lot of space, we'll use this much smaller diagram to represent it from here on out.

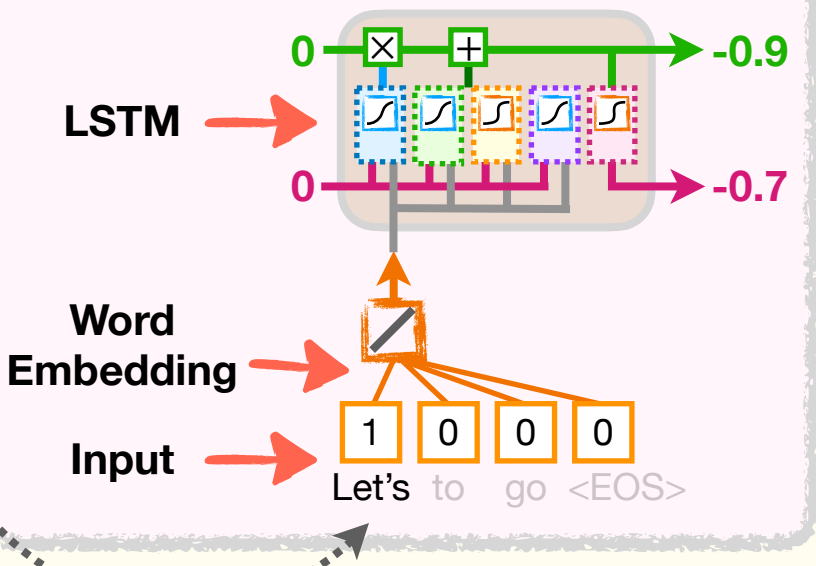


4

Now that we have the Encoder part of the network, the first thing we do is initialize the **Long-Term Memory** and **Short-Term Memory** with **0**.

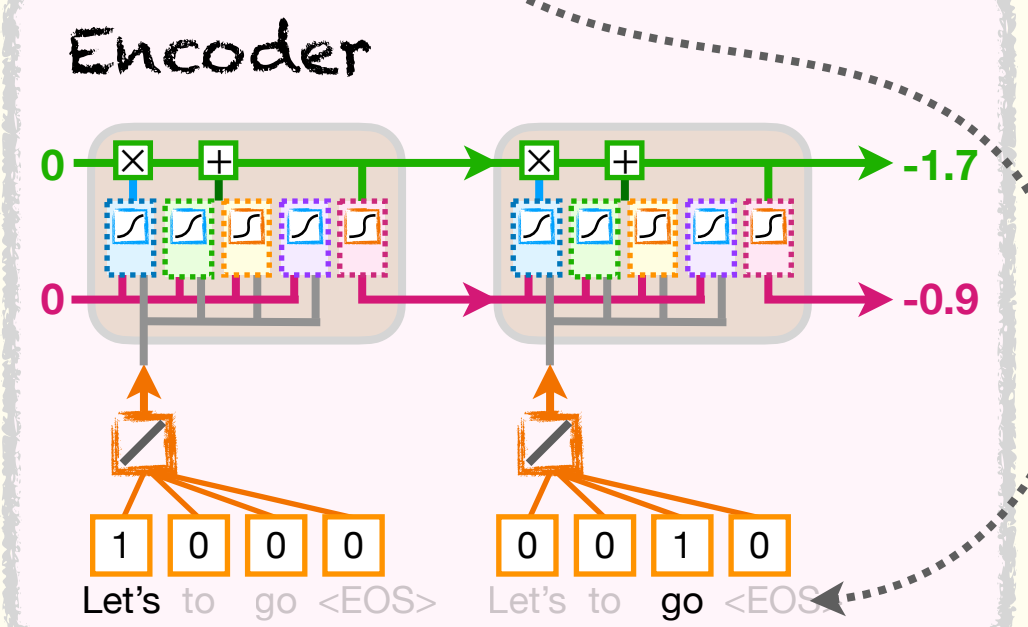


Encoder-Decoder Models: Details



5 Then, the next thing we do is plug a **1** into the input for **Let's** and **0s** for everything else because the first word in the input phrase **Let's go**, is **Let's**…

…and then we unroll the LSTM and the Embedding Layer and put a **1** in the input for **go** and a **0** for everything else.

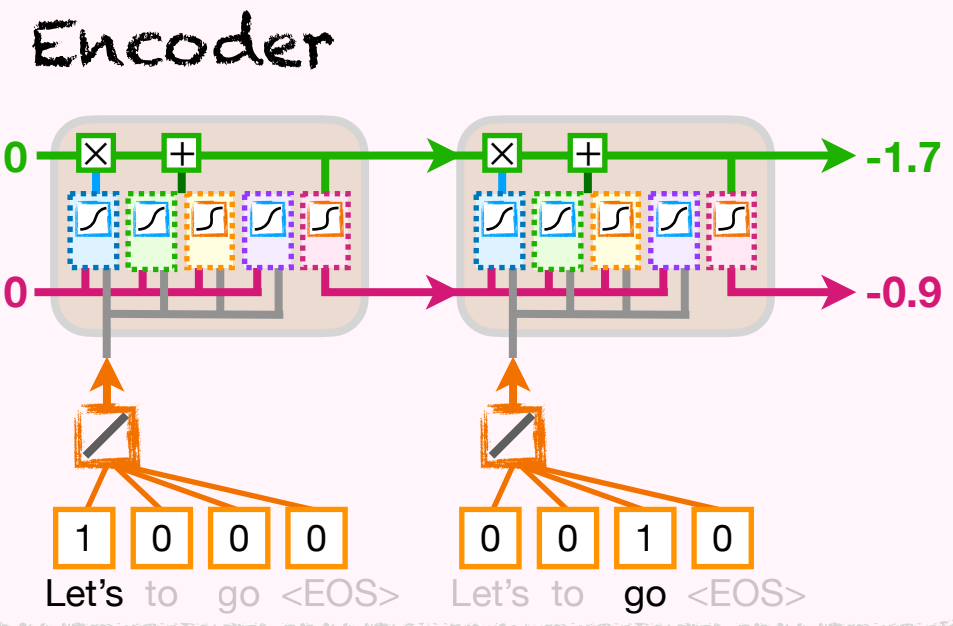


NOTE: To be clear, when we unroll the LSTM and the Embedding Layer, we reuse the exact same weights and biases no matter how many times we unroll them.

In other words, the weights and biases in the LSTM unit and Embedding Layer that we use for the word **Let's** are the exact same weights and biases that we use for the word **go**.

Just a reminder: **Long-Term Memories** are also called **Cell States** and **Short-Term Memories** are also called **Hidden States**.

6 In this example, the Encoder encodes the input phrase, **Let's go**, into a **Long-Term Memory** and a **Short-Term Memory**, **-1.7** and **-0.9**, respectively.



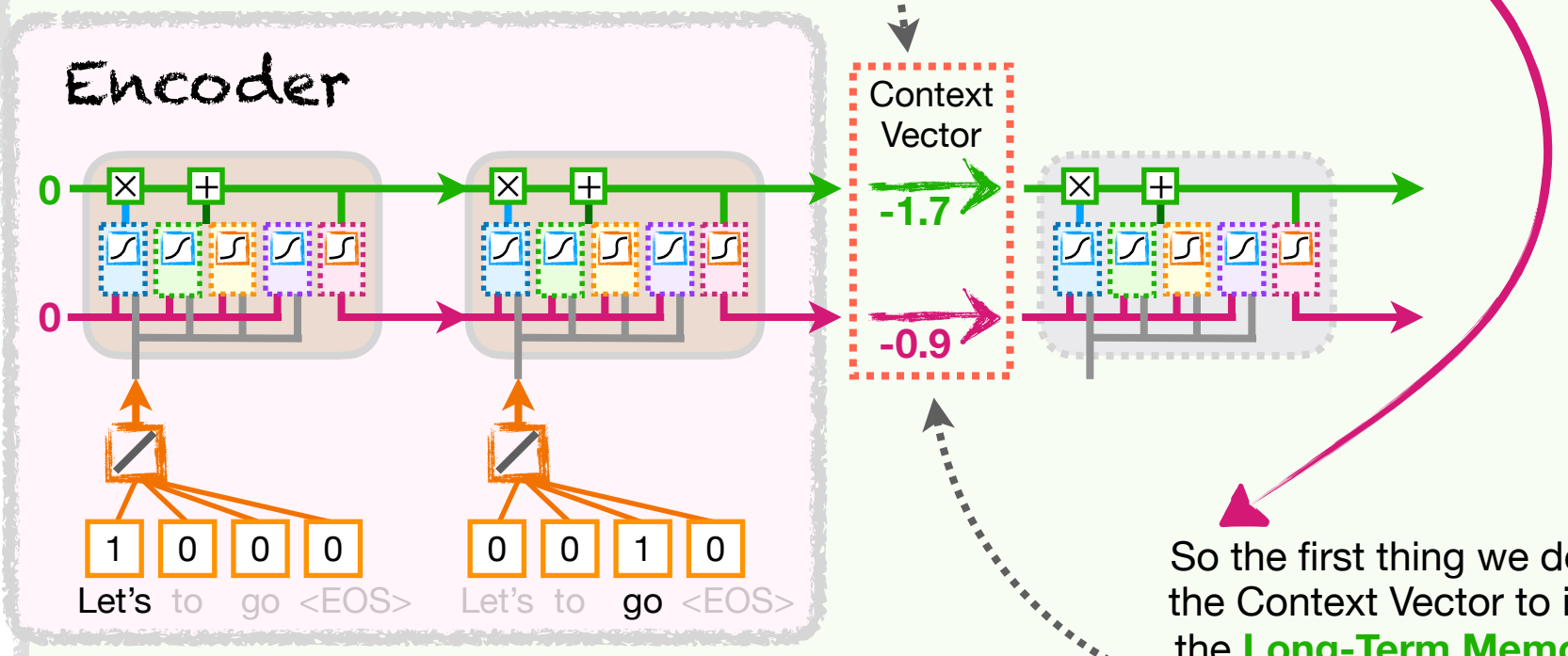
BAM!

TERMINOLOGY ALERT!!!

The last **Long-Term Memory** and **Short-Term Memory** (the **Cell State** and **Hidden State**, respectively) from the LSTM in the Encoder are called the **Context Vector**.

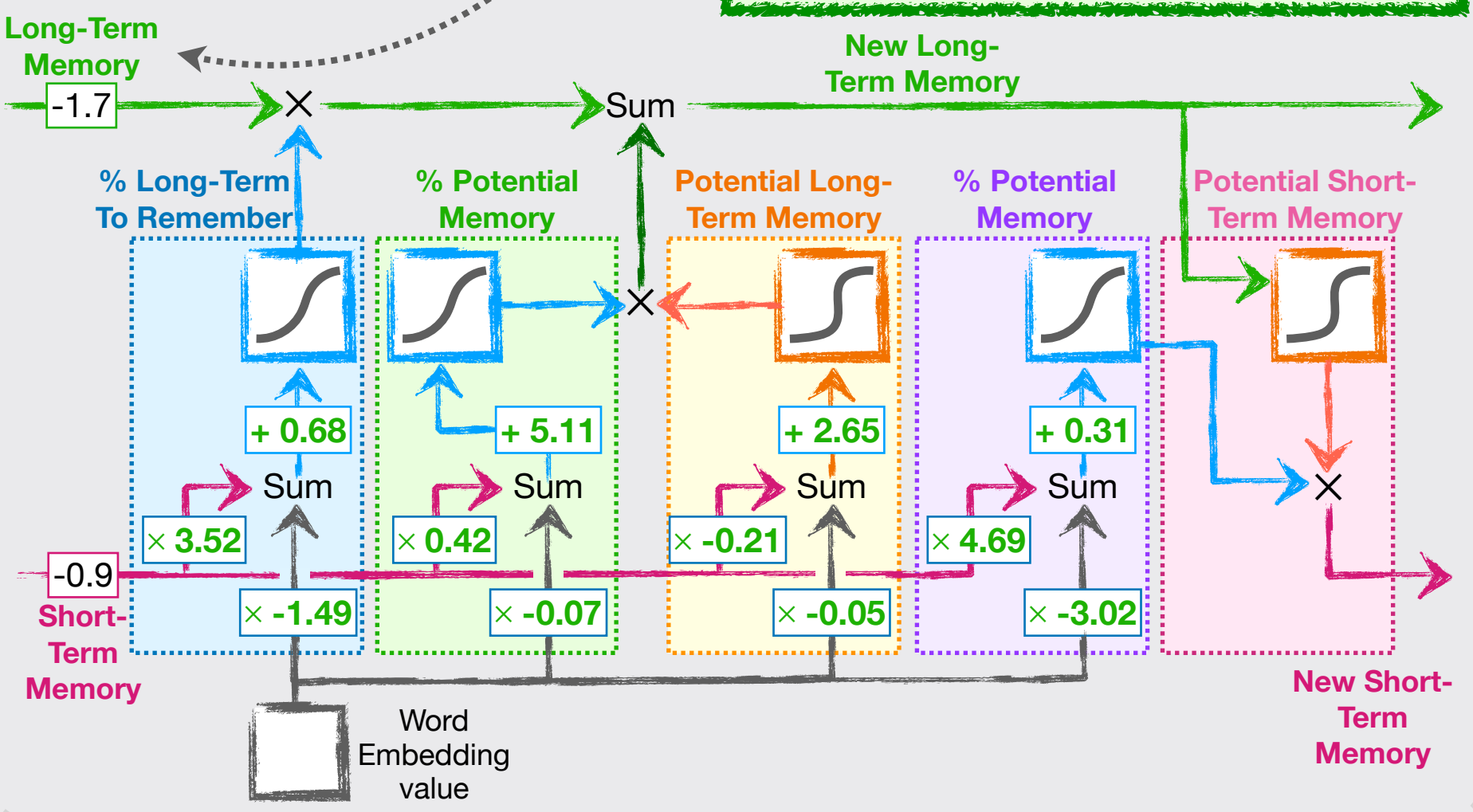
Encoder-Decoder Models: Details

7 Now that we've encoded the input phrase **Let's go** into a Context Vector, we need to decode the Context Vector with a **Decoder**!

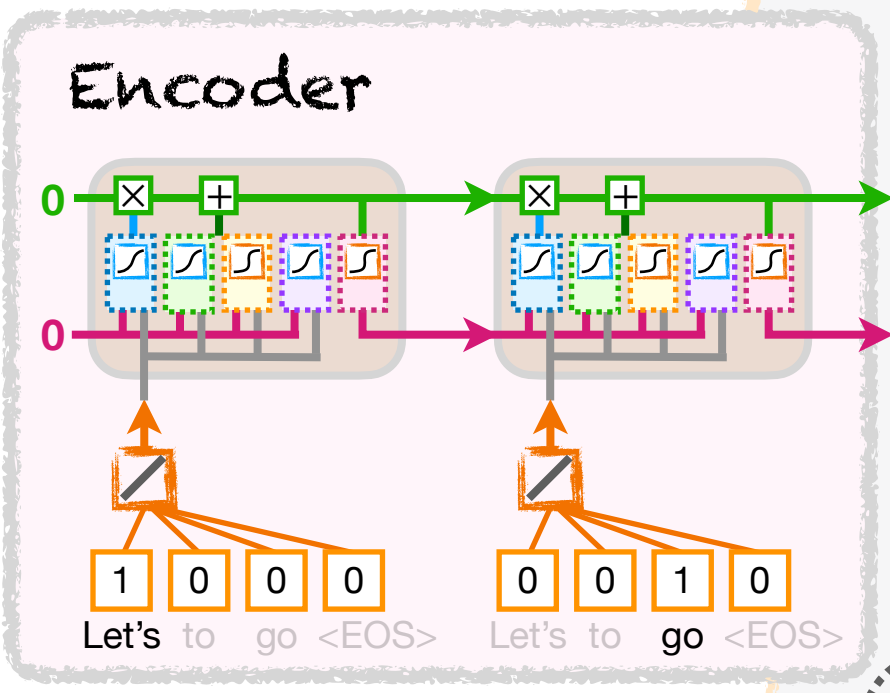


So the first thing we do is use the Context Vector to initialize the **Long-Term Memory** and **Short-Term Memory** (the **Cell State** and **Hidden State**, respectively) in a new LSTM.

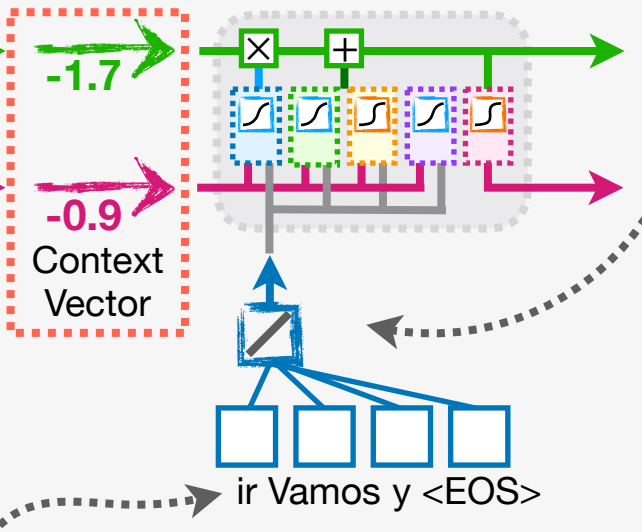
NOTE: To be clear, the LSTM in the **Decoder** part of the Encoder-Decoder model is different from the LSTM in the **Encoder** part of the Encoder-Decoder model and has its own separate weights and biases.



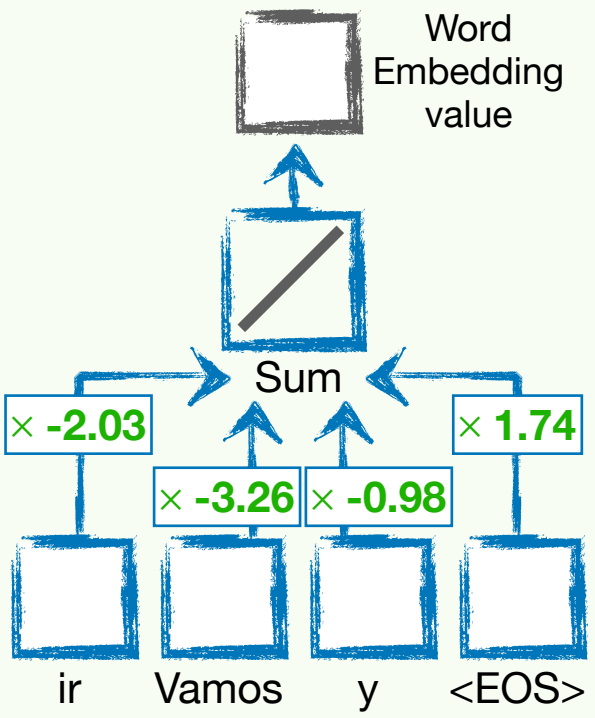
Encoder-Decoder Models: Details



8 Just like in the Encoder, the input to the Decoder LSTM comes from an Embedding Layer.

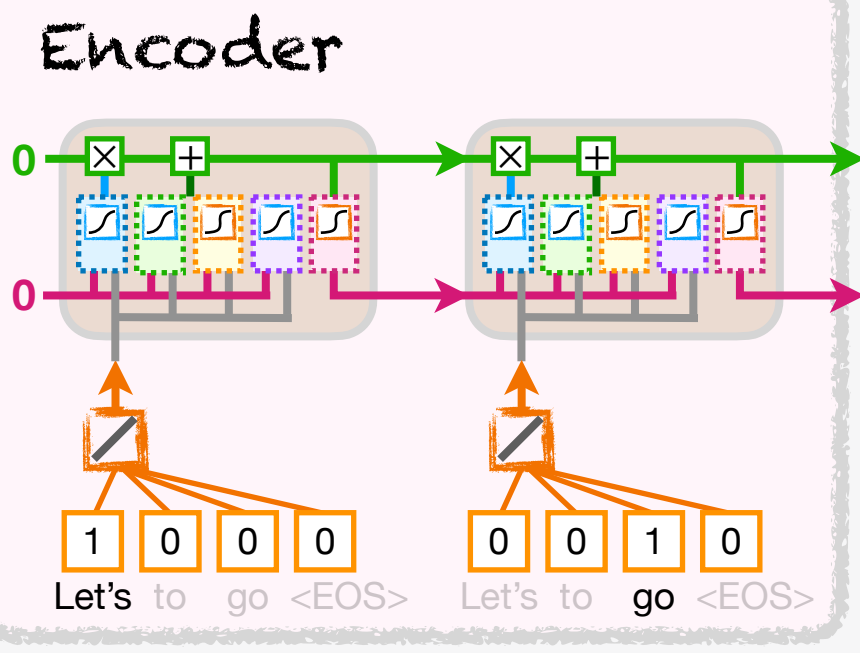


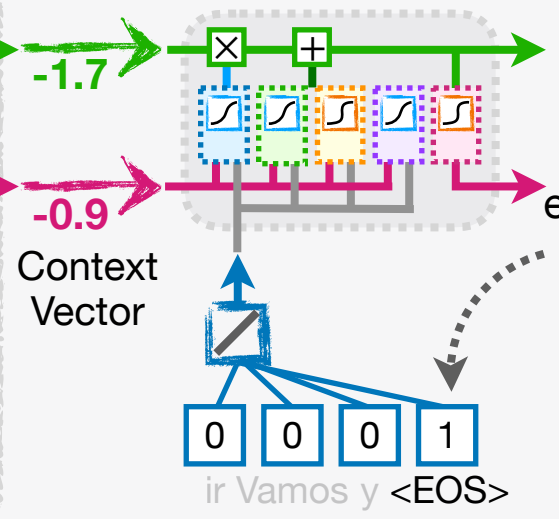
However, now the Embedding Layer creates embedding values for the Spanish words, **ir** (to go), **Vamos** (Let's go), **y** (and), and the **<EOS>** (end of sequence) symbol. Thus, the decoder can translate the English phrases **to go** to **ir** and **Let's go** to **Vamos**. The Spanish word **y** will, ultimately, show that the Transformer can also ignore tokens.



9 Specifically, the Embedding Layer in this Decoder has these Embedding values.

10 Now, because we just finished encoding the English phrase, **Let's go** into the Context Vector…





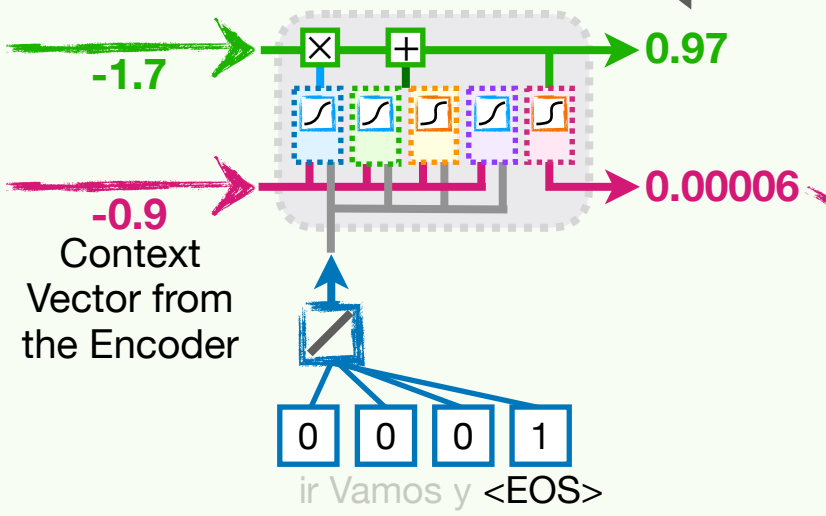
…we initialize the Decoder with the embedding value for the **<EOS>** (end of sequence) token.

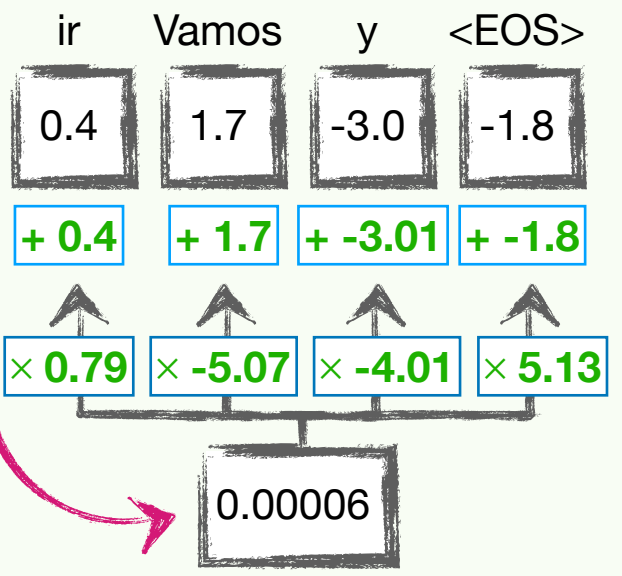
We're using the **<EOS>** token to start the decoding because that's what was used in the first Encoder-Decoder model (Sutskever et al., 2014), and it allows us to keep this example as simple as possible. However, sometimes you'll see people start the decoding with an **<SOS>** token, which stands for **Start of Sentence** or **Start of Sequence**, and which makes more sense to me.

Encoder-Decoder Models: Details

11 Anyway, the Decoder does the math to calculate the **Long-Term Memory** and **Short-Term Memory**…

…and the **Short-Term Memory** is transformed by additional weights and biases in something called a **Fully Connected Layer**.

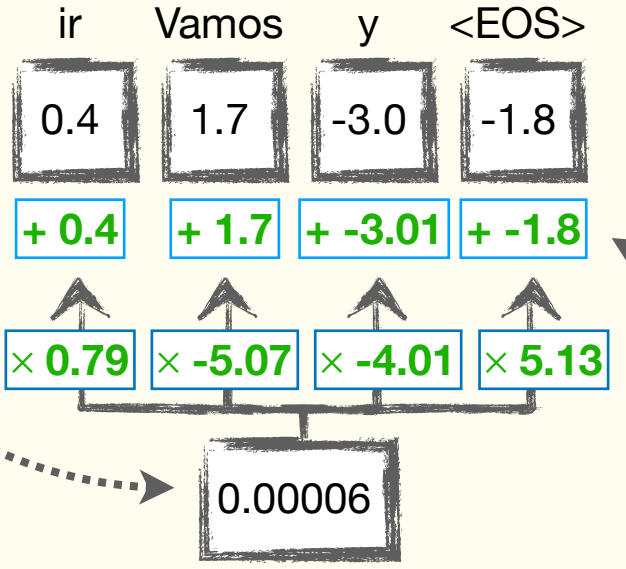




TERMINOLOGY ALERT!!! A **Fully Connected Layer** is just another name for a basic neural network without a hidden layer.

12 This Fully Connected Layer has **1** input for the **Short-Term Memory**…

…and **4** outputs, one for each token in the Spanish vocabulary…

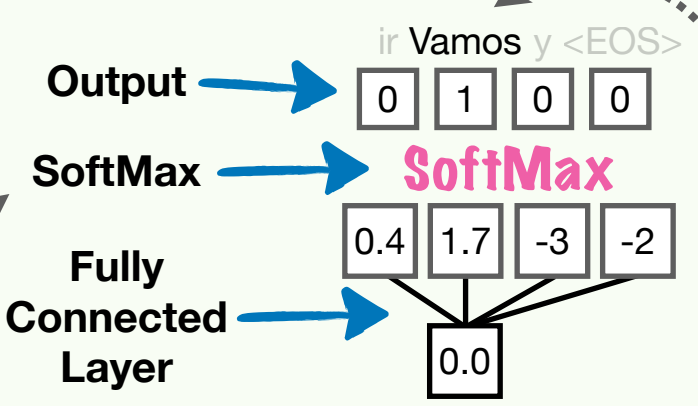


…and, in between, we have connections from the input and the outputs with weights and biases.

13 After the Fully Connected Layer, we run the values through the SoftMax function and use those outputs to determine the first token generated by the Decoder.

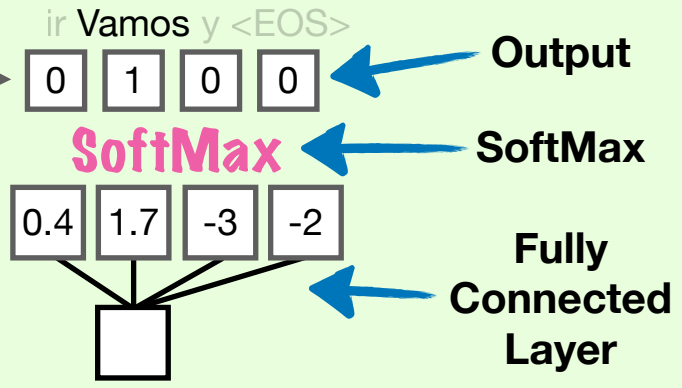
In this case, we will use the output with the highest value to determine what token the Decoder should generate. Thus **Vamos**, is the first word generated by the Decoder.

BAM!

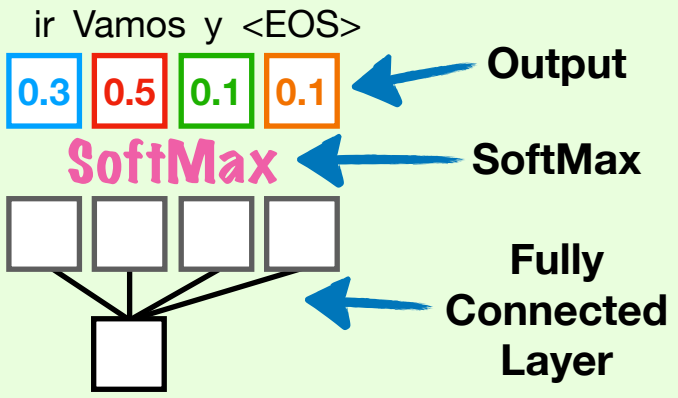


Encoder-Decoder Models: Details

NOTE: In the example we're going through here we are using the output with the highest value to determine what token the Decoder should generate. And when the outputs from the SoftMax function give a **1** for **Vamos** and **0s** for everything else, selecting the token with the highest output value makes sense.

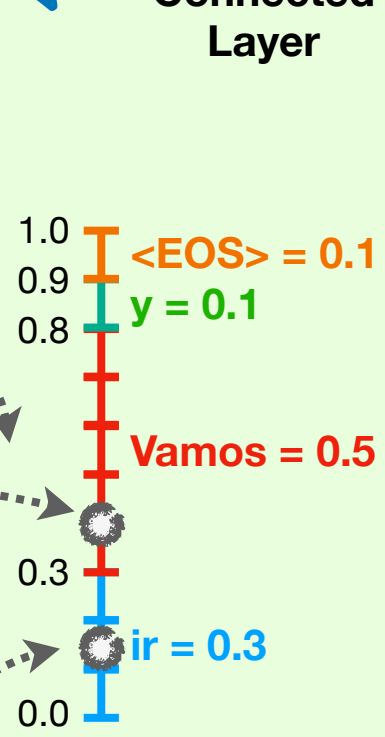


However, it's just as easy to get outputs like these, where no single token gets a **1** and all of the tokens get non-zero values.



In this case, Instead of simply generating the token with the largest output value every single time, the properties of the SoftMax function—that the mutually exclusive output values are always between **0** and **1** (inclusive) and always add up to **1**—allow us to treat the outputs like we would a **Probability Distribution***.

What this means is we could stack the output values so that they create intervals for each token between **0** and **1**…



…and then we could use a random number generator to determine what the next token should be. For example, if the random number generator gives us numbers between **0** and **1**, then we might get **0.42**…

…and because **0.42** falls within the range of values assigned to **Vamos**, the network generates **Vamos**.

Alternatively, the random number generator could give us **0.17**, and in that case, the network generates **ir**.

Using all of the "probabilities" to generate a token means that each token is generated in a way that's proportional to its output value. In other words, tokens with relatively large output values will be selected more frequently than tokens with relatively small output values.

Using this method means that we won't always generate the same token given the same input, and this makes the response generated by the Decoder seem more human and interesting.

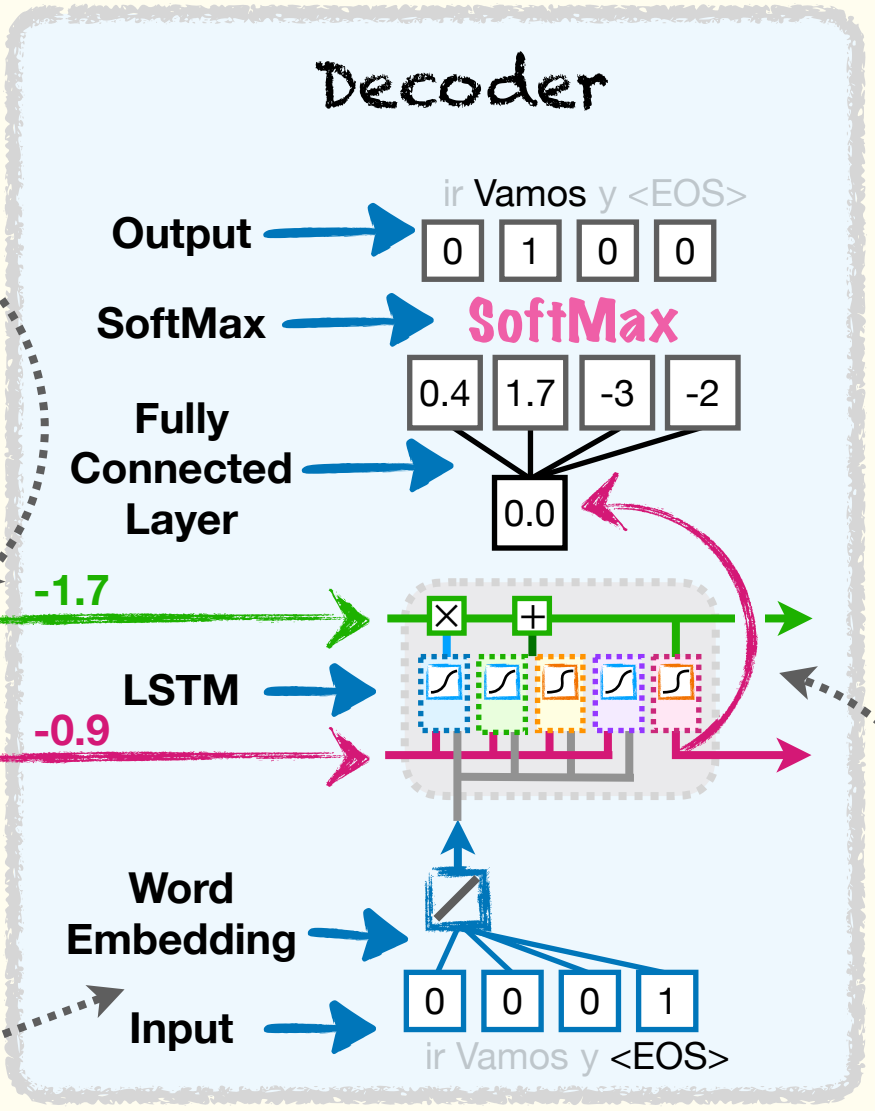
---

*Technically the output from the SoftMax function isn't just *like* a **Probability Distribution**, it *is* a **Probability Distribution** because that's how **Probability Distributions** are defined. However, like we saw in **Chapter 4**, these "probabilities" depend on the random initial values we start with and can be different every single time we optimize the parameters. Thus, I tend to think of them less as true probabilities and put quotes around the word to emphasize their fundamental randomness.

Encoder-Decoder Models: Details

14

To summarize what we've done so far, the Context Vector, which was created by the Encoder from the input phrase **Let's go**, is used to initialize the **Long-Term Memory** and **Short-Term Memory** of an LSTM in the Decoder…



…and the input is initialized with the Embedding value for the **<EOS>** token from an Embedding Layer that contains the output vocabulary…

…and the **Short-Term Memory** from the LSTM is fed into a Fully Connected Layer that goes to the SoftMax function. The outputs from the SoftMax function are then used to determine the first word generated by the Decoder.

All of these parts are what make up the Decoder.

15

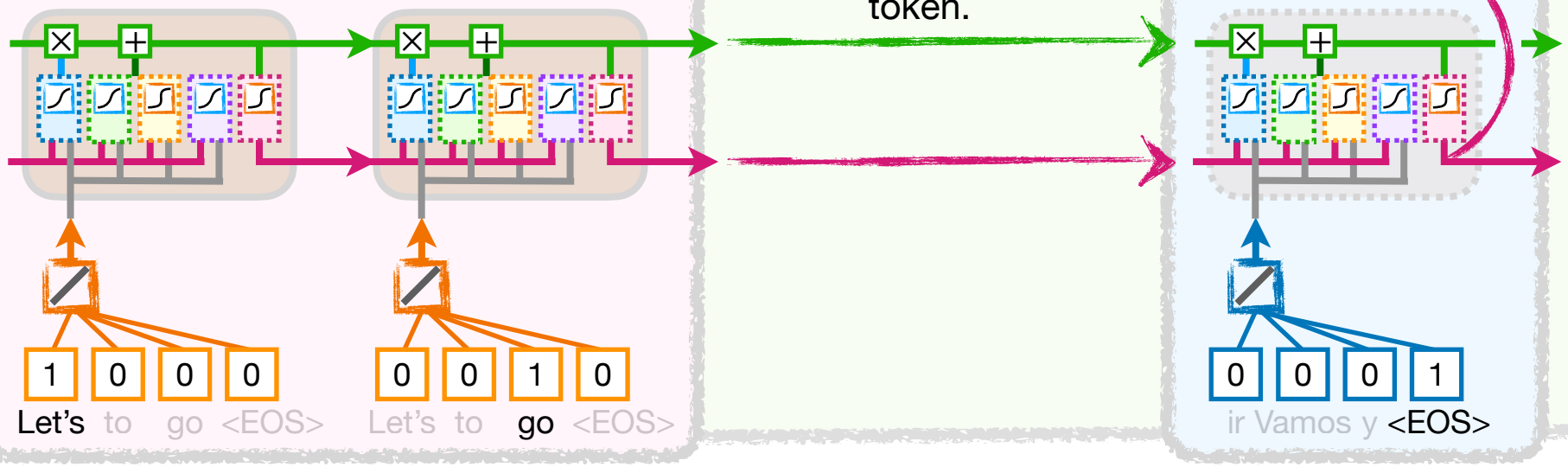
Now we can see an Encoder-Decoder model in action!

The Encoder encodes the input phrase, **Let's go**, into a Context Vector…

…and the Decoder decodes the Context Vector and generates a translation, **Vamos**, which translates to **Let's go** in English.

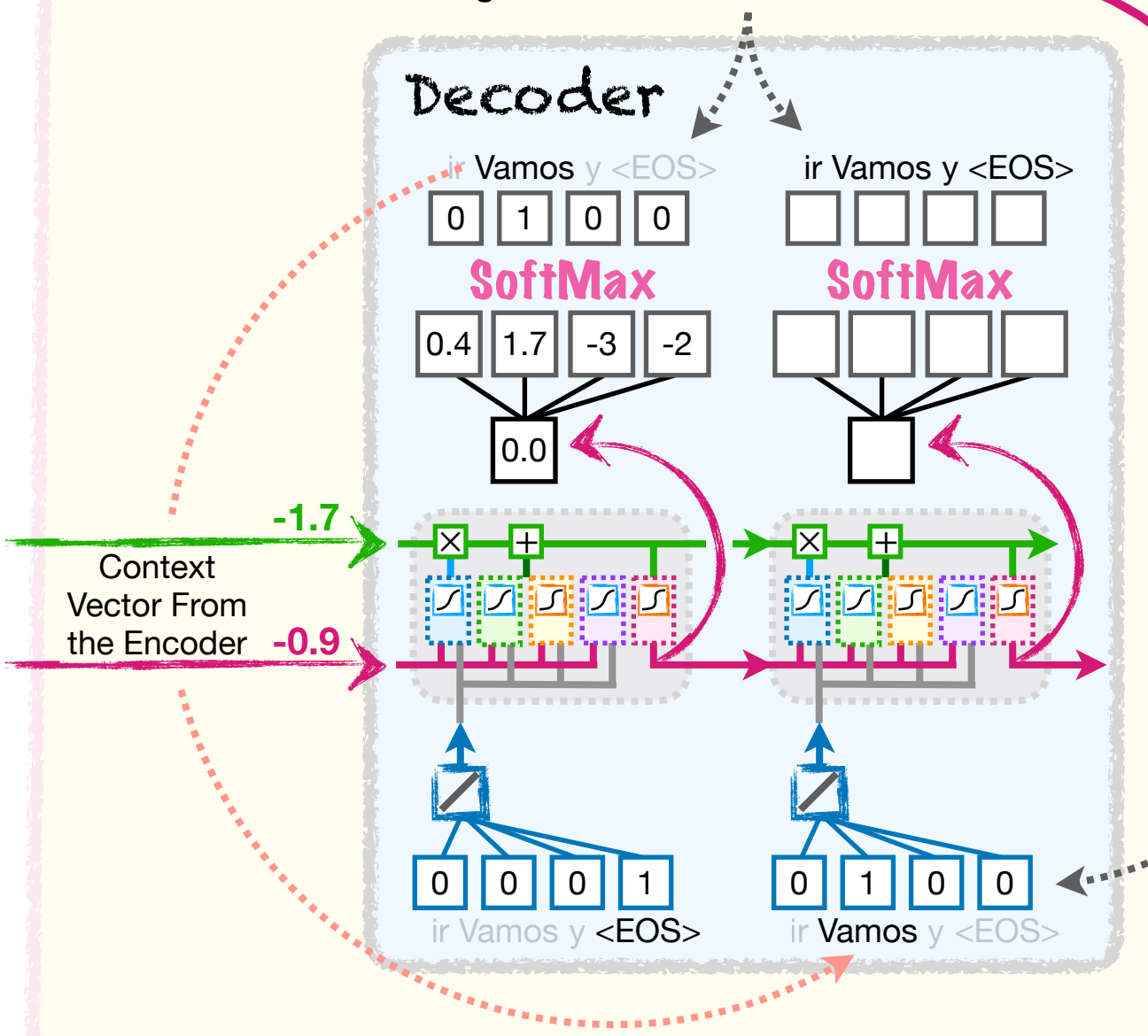
Bam? Not yet.

So far, the translation is correct, but the Decoder doesn't stop until it generates an **<EOS>** token.

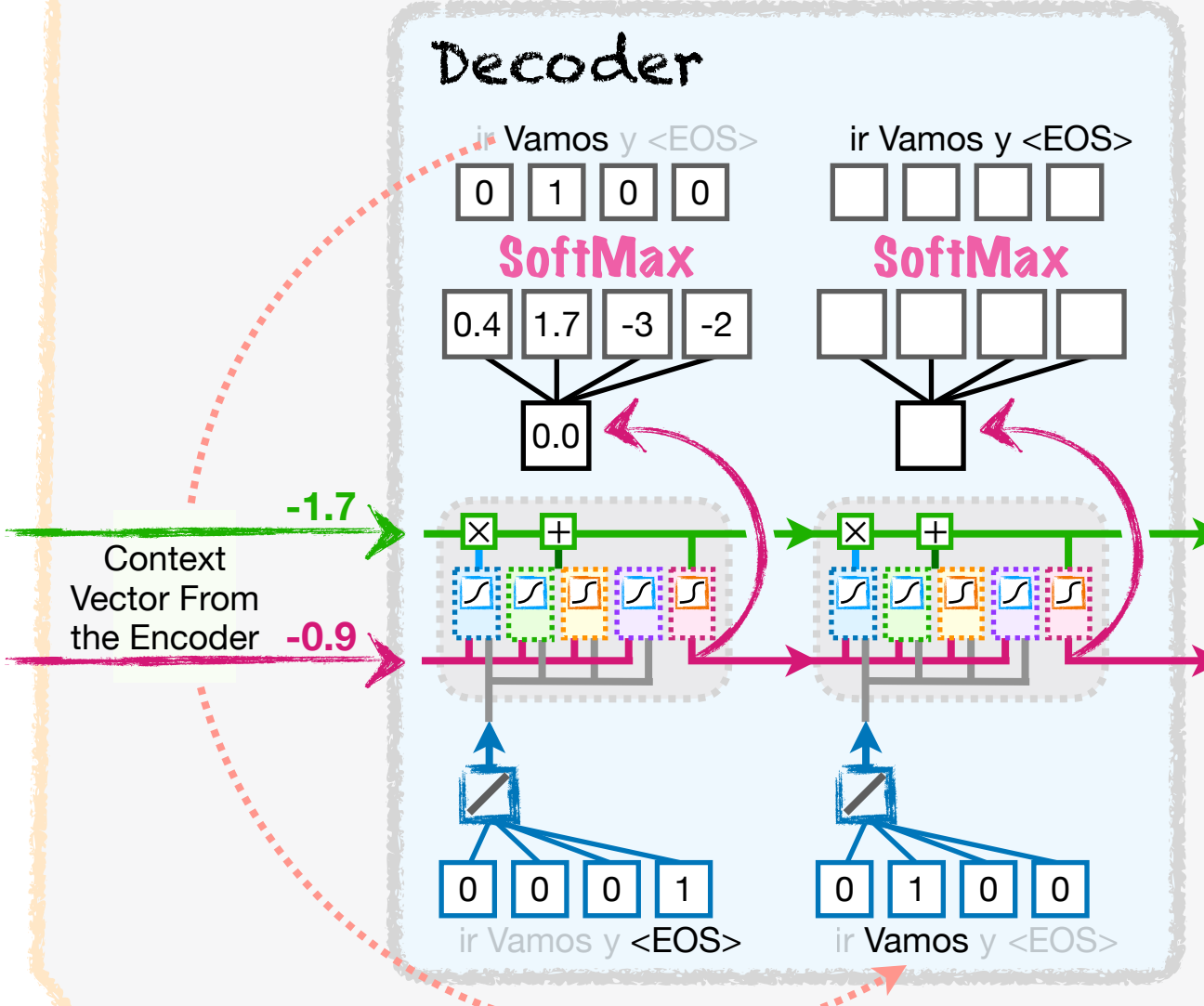


Encoder-Decoder Models: Details

16 Because the Decoder has not yet generated the **<EOS>** token, it's not done decoding, so the next thing we do is unroll the Decoder…



…and plug the last word that the Decoder generated, **Vamos**, into the input of the Decoder's unrolled Embedding Layer.

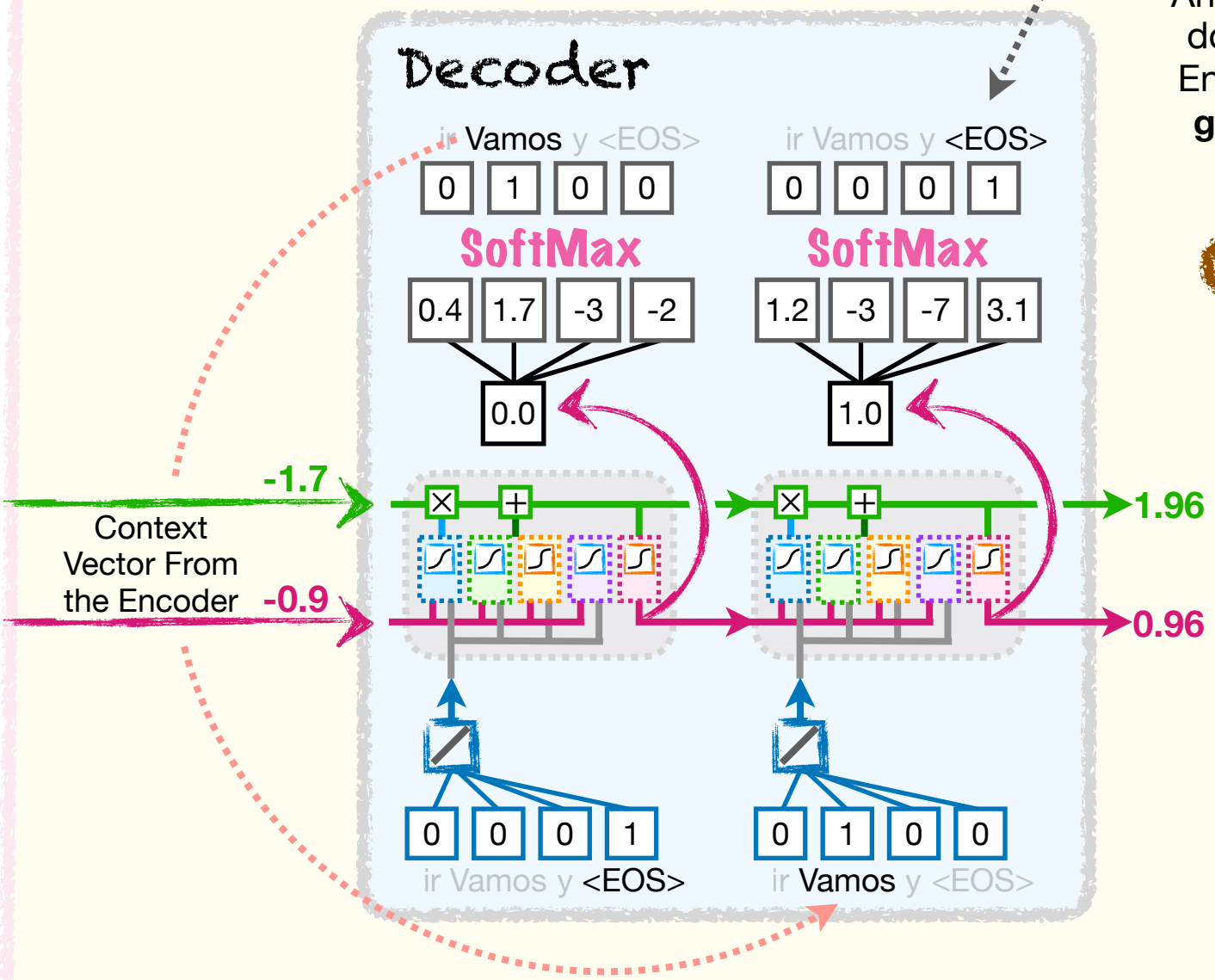


17 Then we do the math in the LSTM to calculate the **Long-Term Memory** and **Short-Term Memory**.

Encoder-Decoder Models: Details

18 Then we run the **Short-Term Memory** through the Fully Connected Layer and the SoftMax function to generate the second token, which turns out to be **<EOS>**.

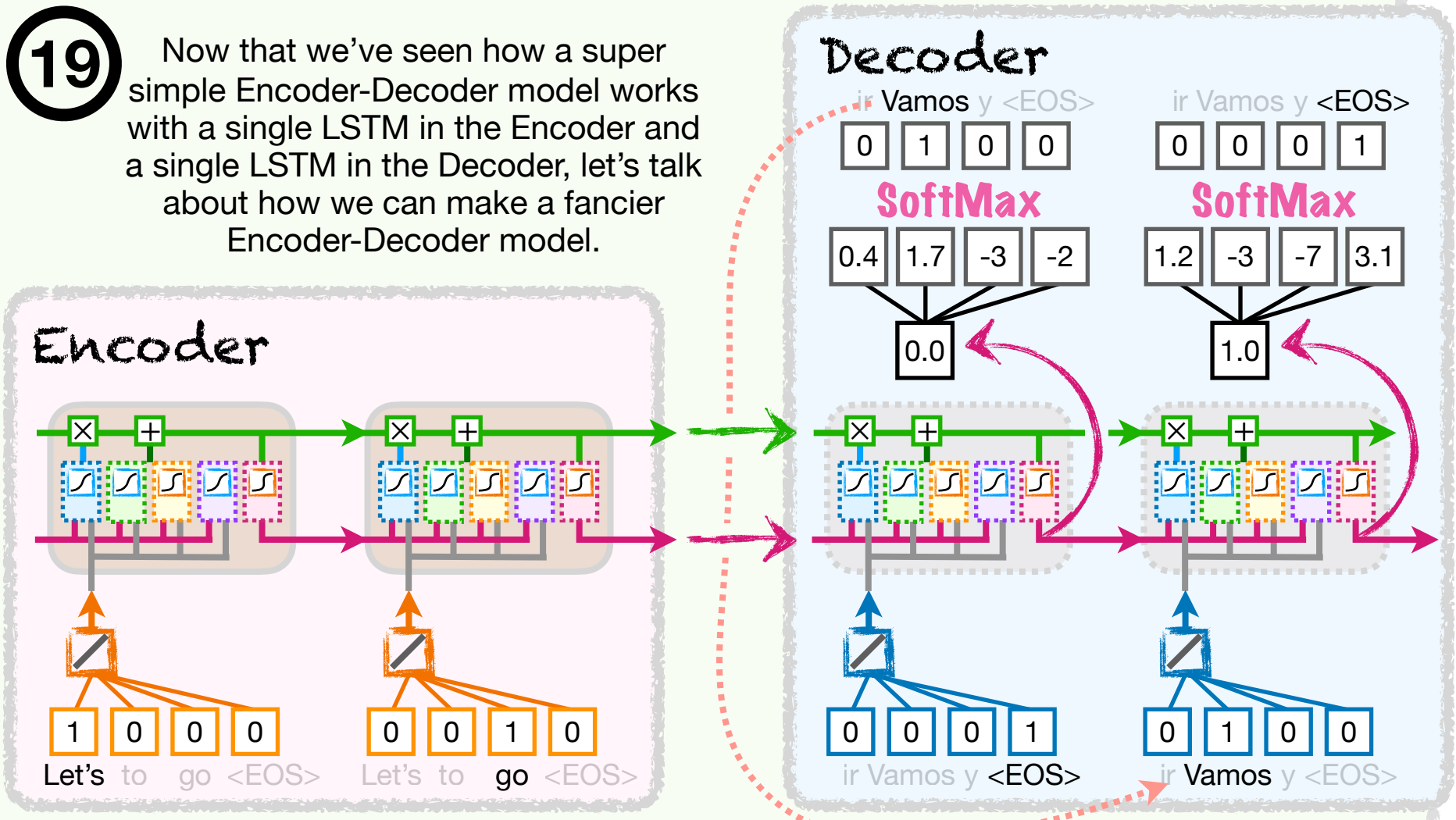
And that means we're done translating the English phrase, **Let's go**, into the correct Spanish phrase.



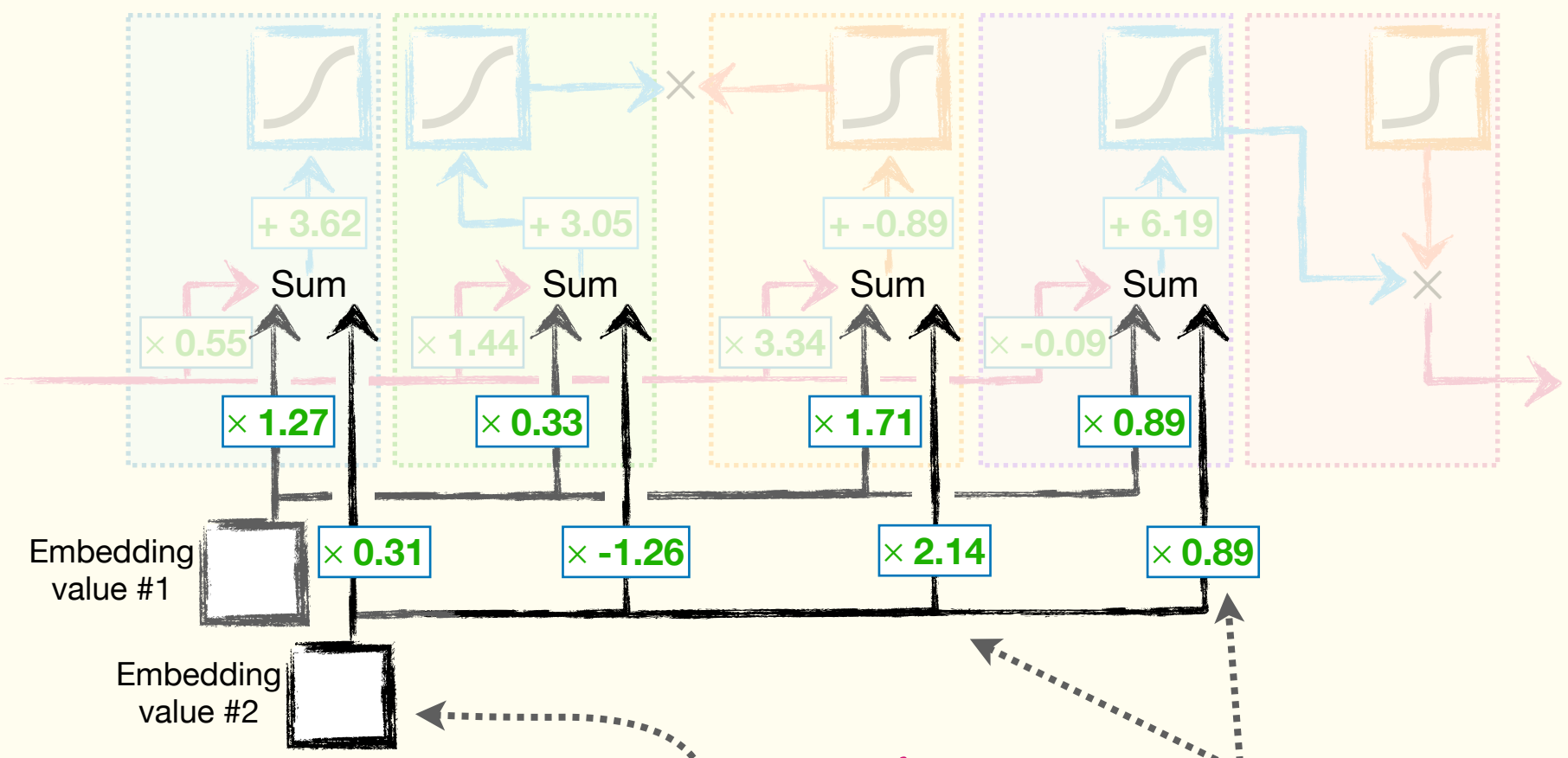
DOUBLE BAM!!!

Encoder-Decoder Models: Details

19 Now that we've seen how a super simple Encoder-Decoder model works with a single LSTM in the Encoder and a single LSTM in the Decoder, let's talk about how we can make a fancier Encoder-Decoder model.



20 First, we can create more than one Embedding value per token. For example, these Embedding Layers create **2** values per token. However, it's not unheard of to create **1,000** or more Embedding values per token!



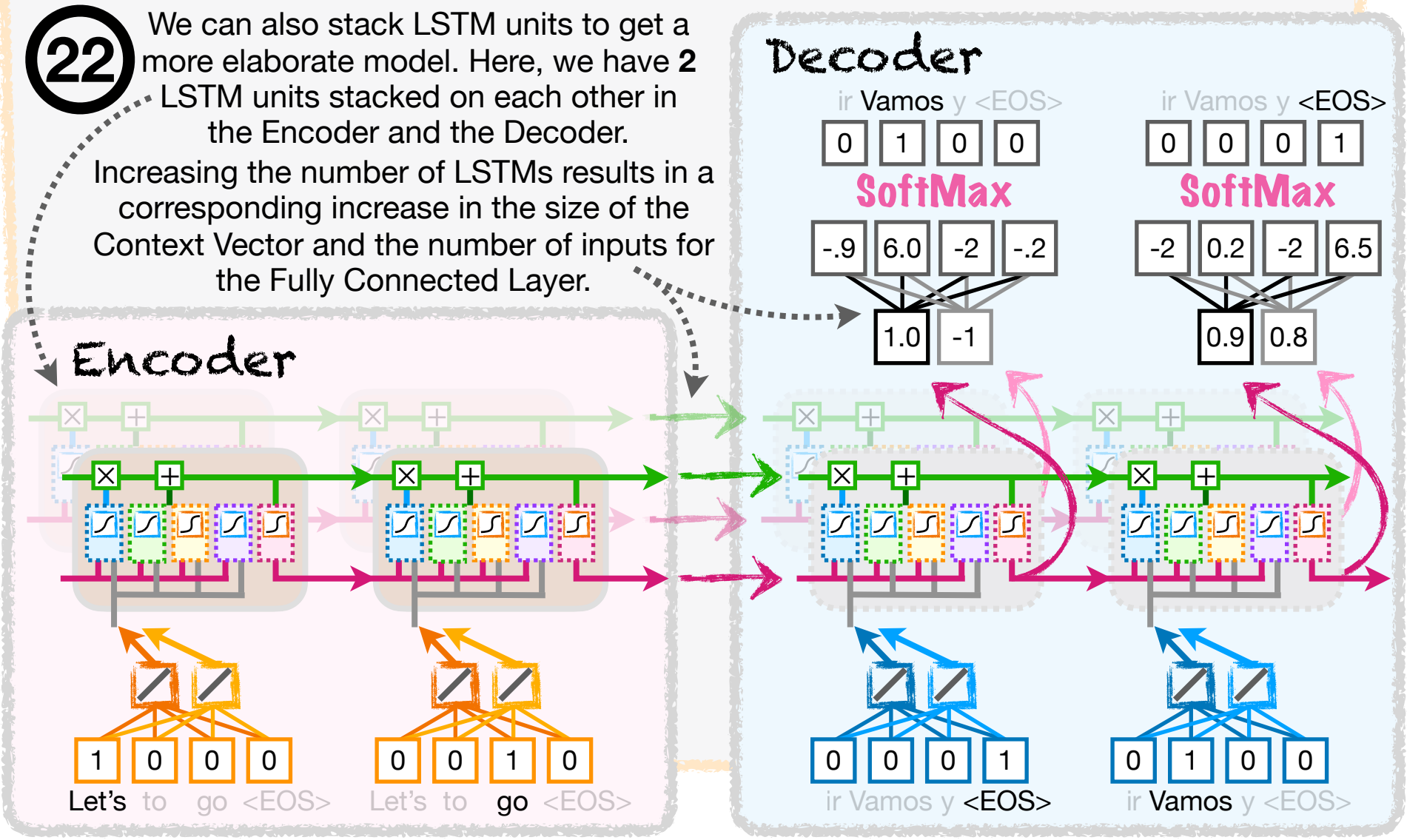
21 If we have **2** Embedding values, then we have to add a second input for the second value…

…and additional weights on those new connections to each part of the LSTM.

Encoder-Decoder Models: Details

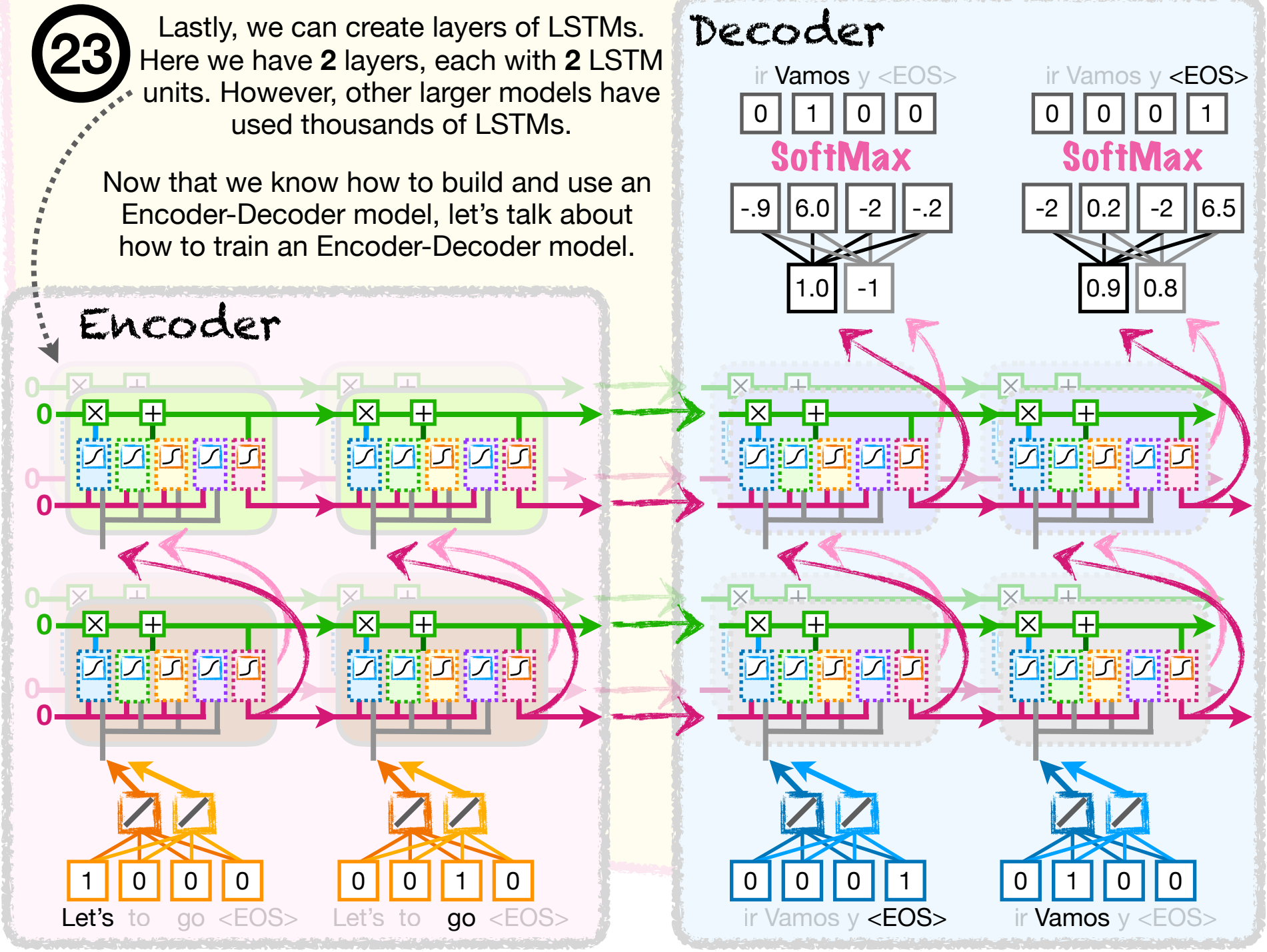
22 We can also stack LSTM units to get a more elaborate model. Here, we have **2** LSTM units stacked on each other in the Encoder and the Decoder.

Increasing the number of LSTMs results in a corresponding increase in the size of the Context Vector and the number of inputs for the Fully Connected Layer.



23 Lastly, we can create layers of LSTMs. Here we have **2** layers, each with **2** LSTM units. However, other larger models have used thousands of LSTMs.

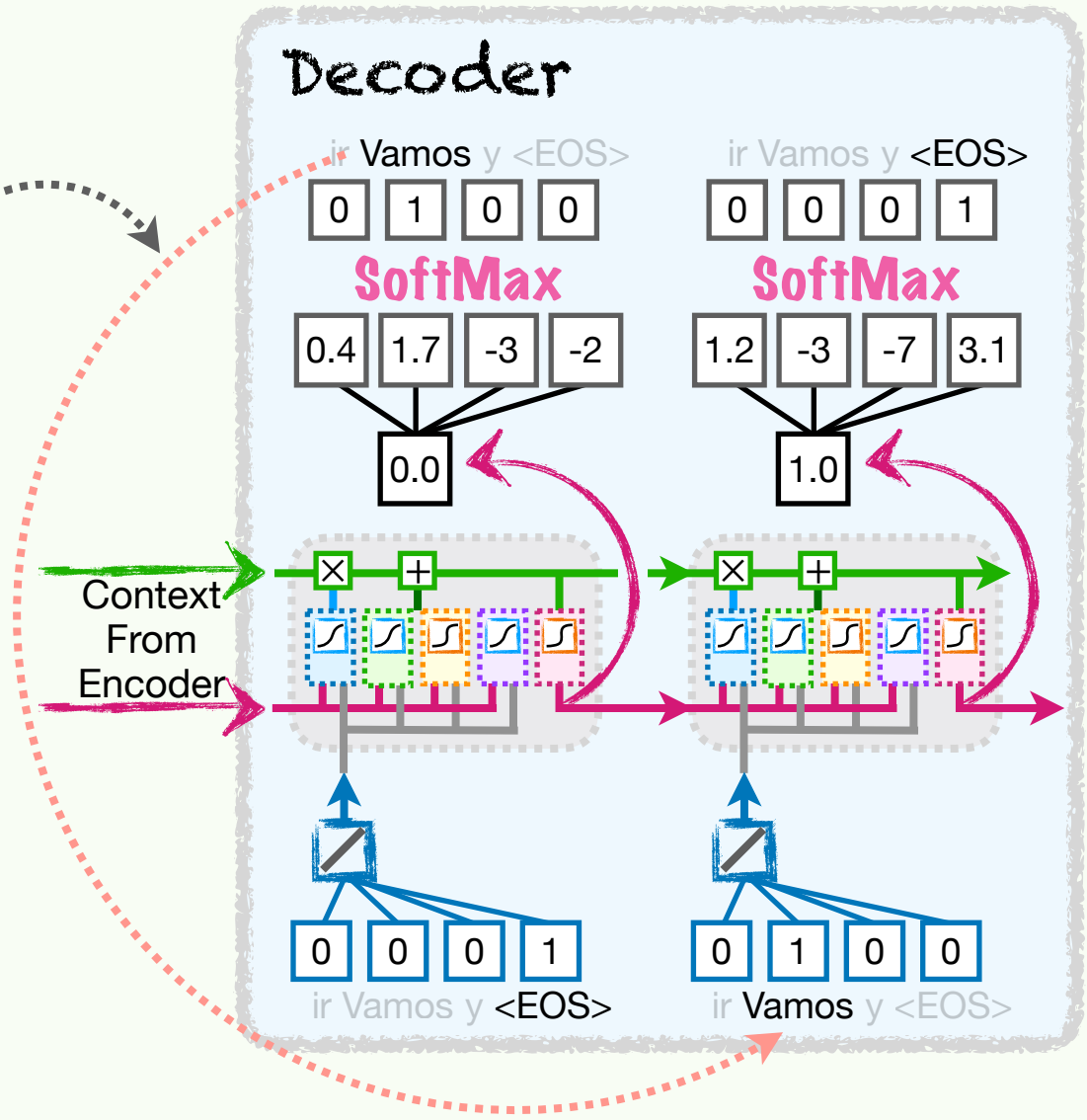
Now that we know how to build and use an Encoder-Decoder model, let's talk about how to train an Encoder-Decoder model.



Encoder-Decoder Models: Details

24 Earlier, when we translated **Let's go** into **Vamos**, we used the token generated by the Decoder, **Vamos**, as the input to the next unrolled Embedding Layer.

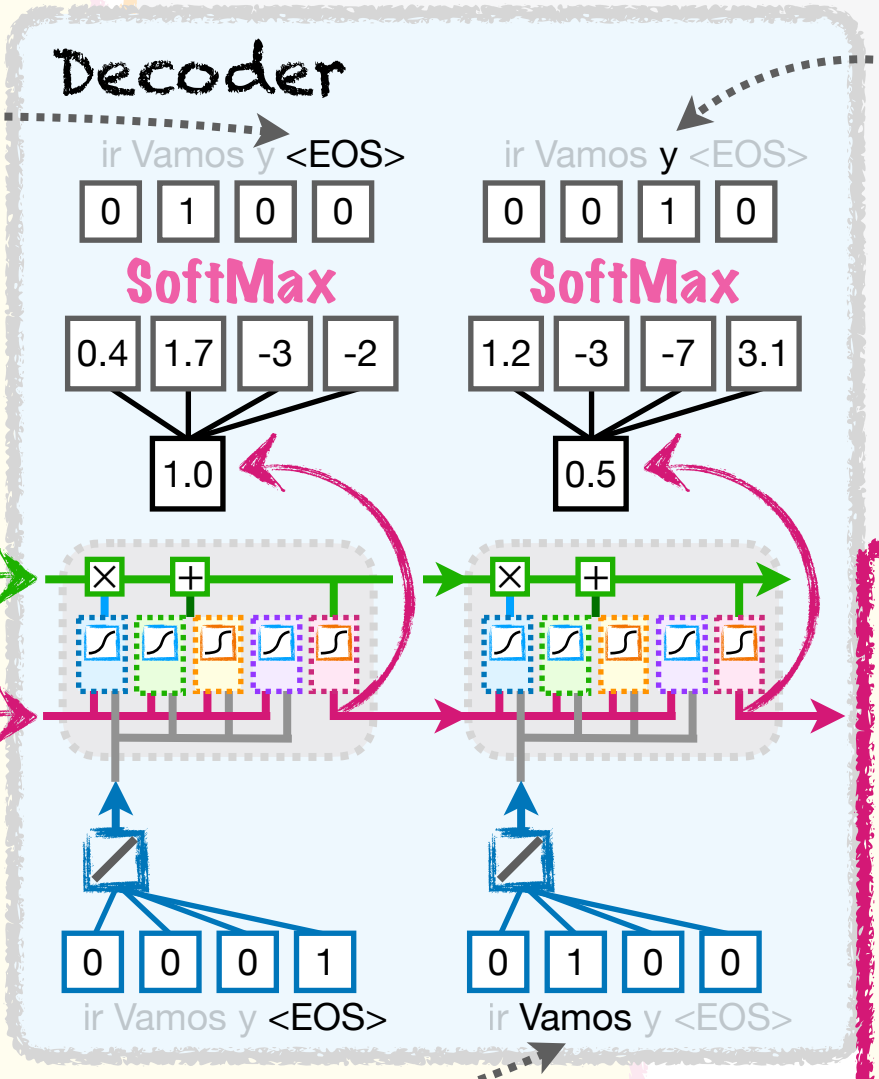
In contrast, when training an Encoder-Decoder, instead of using the generated token as input to the unrolled Decoder, we use the known, correct token.



25 In other words, if the first generated token was **<EOS>**, which, in theory, would mean that we should stop decoding…

…then during training, we would still use **Vamos**, the correct Spanish word, as input to the unrolled Decoder.

26 Also, during training, instead of just predicting tokens until the Decoder predicts the **<EOS>** token, each output phrase ends where the known phase ends.



In other words, even if the second predicted token was the Spanish word **y** instead of the correct token **<EOS>**, then during training, we would still stop generating additional tokens. **BAM!**

Now let's code an Encoder-Decoder model in **PyTorch**!

TERMINOLOGY ALERT!!!

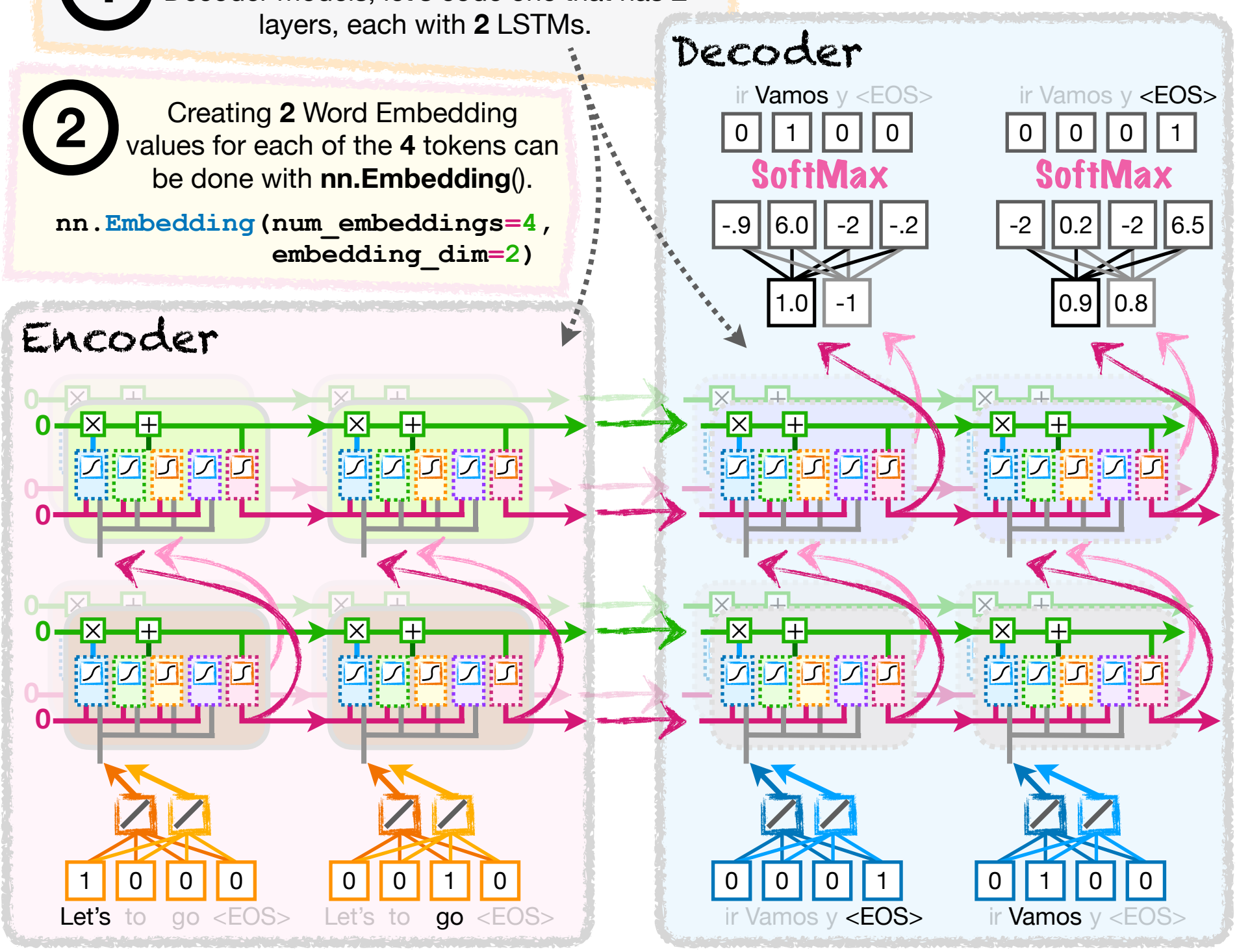
Plugging in the known words and stopping at the known phrase length, rather than using the generated tokens for everything, is called **Teacher Forcing**, which sounds vaguely like what happened to me in school!

Encoder-Decoder Models in PyTorch

1 Now that we know the details of Encoder-Decoder models, let's code one that has **2** layers, each with **2** LSTMs.

2 Creating **2** Word Embedding values for each of the **4** tokens can be done with **nn.Embedding**().

```
nn.Embedding(num_embeddings=4,
             embedding_dim=2)
```



3 We can create the stacks and layers of LSTM units with **nn.LSTM()**.

```
nn.LSTM(input_size=2,
        hidden_size=2,
        num_layers=2)
```

Setting **input_size** to **2** allows us to use both Word Embedding values as inputs.

Setting **hidden_size** to **2** creates a stack of **2** LSTM units that both accept the same input values.

Setting **num_layers** to **2** to creates **2** layers of the stack of LSTM units defined by **hidden_size**.

4 Lastly, we can create the Fully Connected Layer with **nn.Linear()**.

```
nn.Linear(in_features=2,
          out_features=4)
```

5 Scan, click, or tap this QR code to access the **PyTorch** code that builds and trains an Encoder-Decoder model with **nn.Embedding()**, **nn.LSTM()**, and **nn.Linear()**!!!