MACHINE LEARNING

MEI/1



University of Beira Interior,
Department of Informatics

Hugo Pedro Proença,
hugomcp@di.ubi.pt, 2026/27

Machine Learning

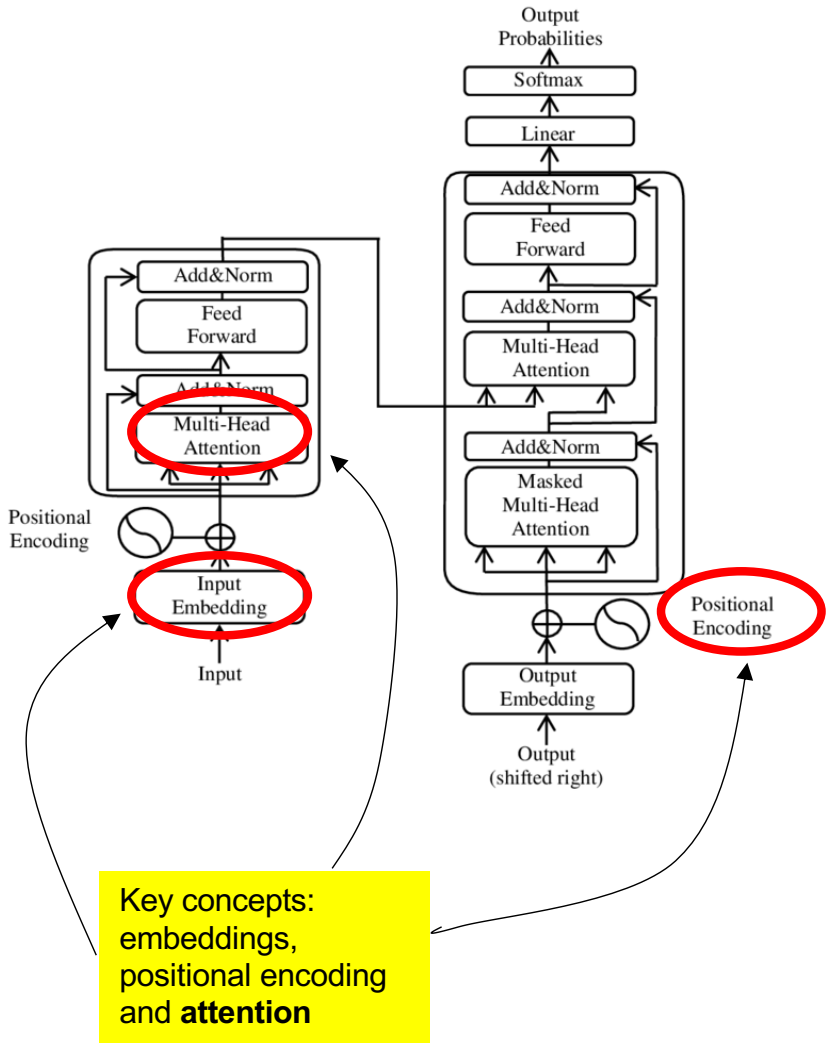
Week 11

Syllabus

- Transformers

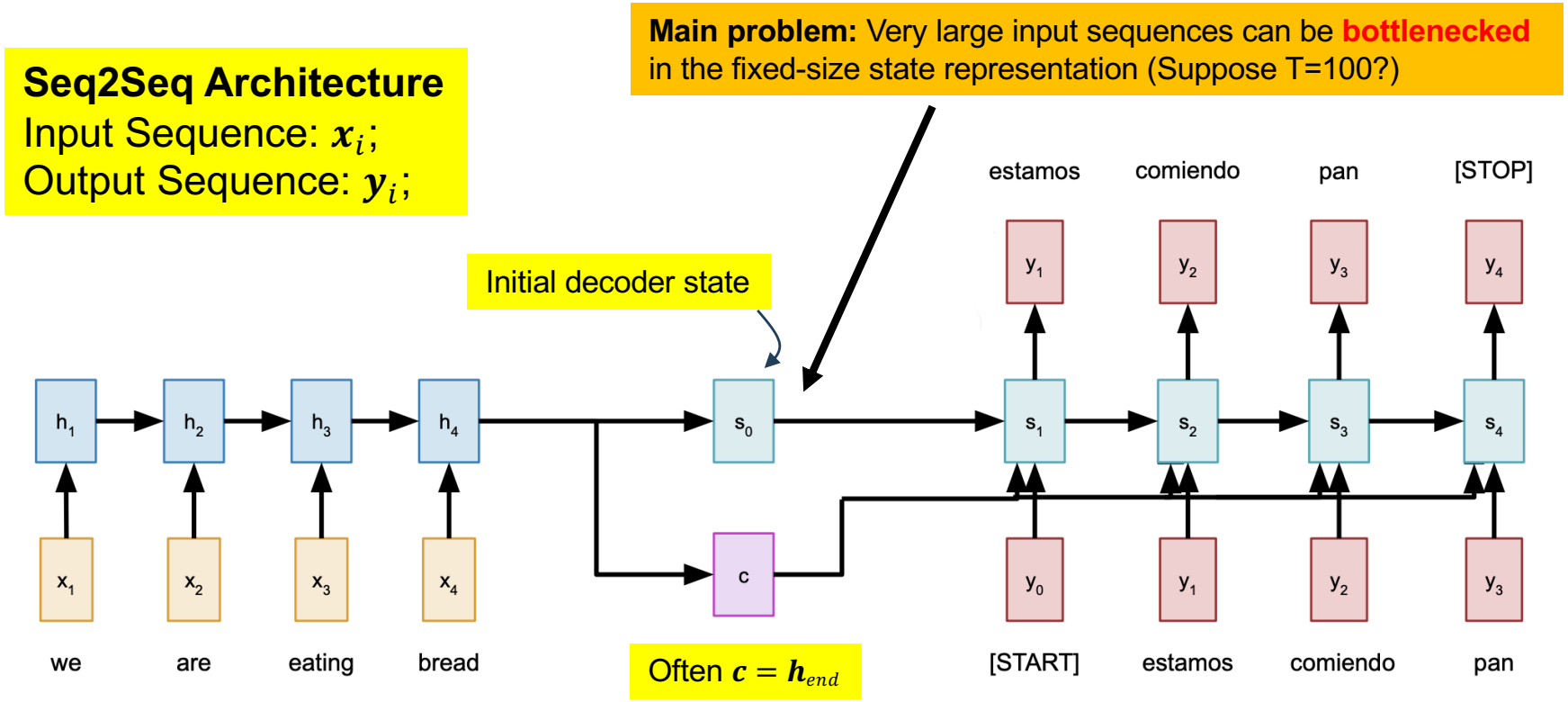
Attention and Transformers

- The Transformer architecture was proposed in the paper entitled "*Attention is All You Need*"
- As of March 2024, this paper had over 111,000 citations from peers
- It was responsible for expanding the 2014 attention mechanism (originally proposed by Bahdanau et. al.) into the Transformer architecture.
- The paper is **considered the founding document for modern artificial intelligence**, as transformers became the main architecture of large language (and vision) models (e.g., Chat GPT).



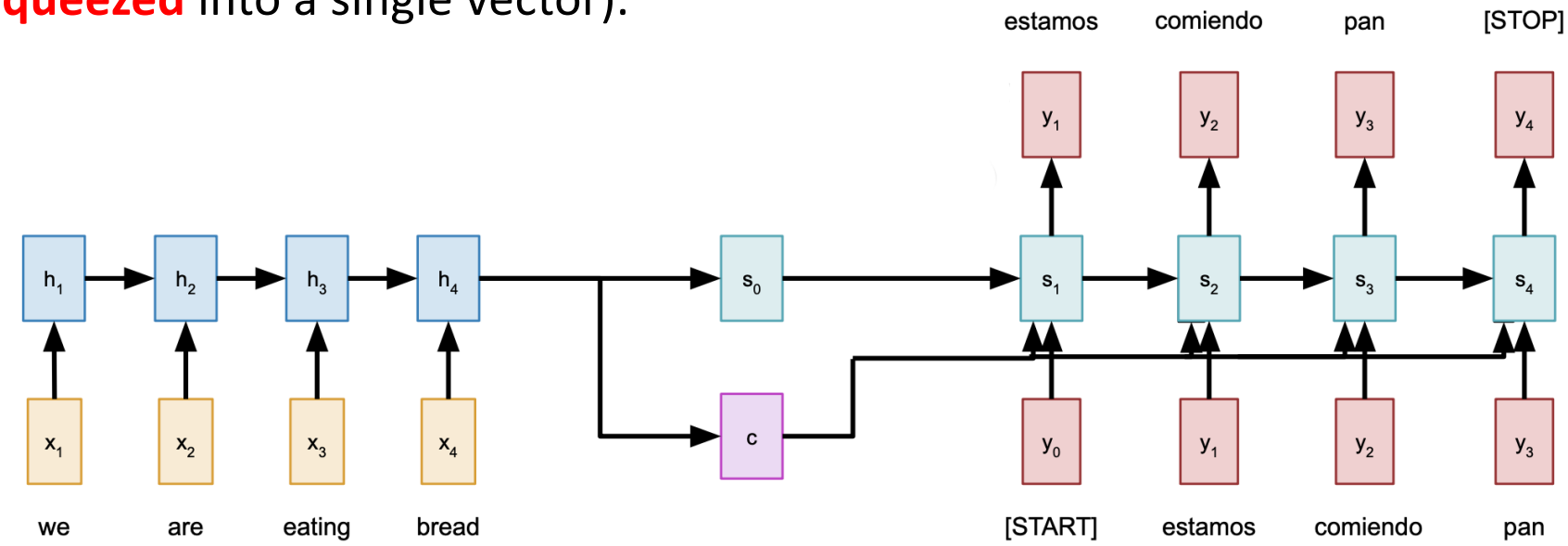
Attention and Transformers

- It was originally proposed for "Machine translation" purposes, i.e., sequence-to-sequence tasks.
- The focus was on improving Seq2seq techniques for machine translation, but even in their paper the authors saw the potential for other tasks like "*question answering*" and for what is now called multimodal Generative AI.



Attention and Transformers

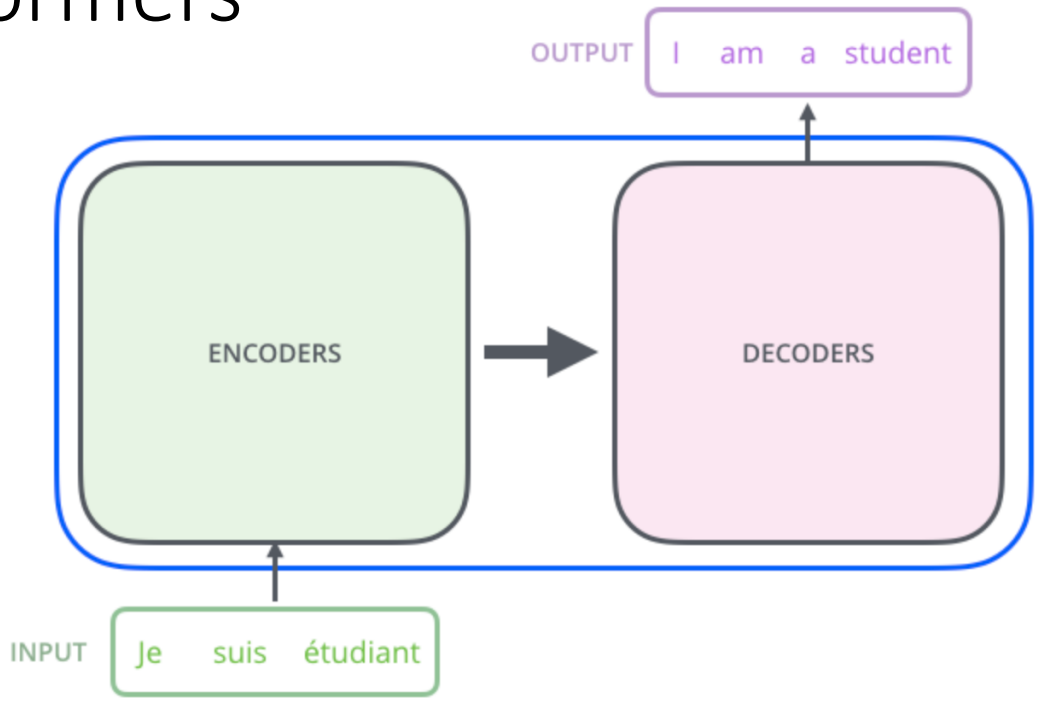
- Using this architecture, the encoder must encapsulate the entire input into a fixed-size vector that is passed to the decoder.
- With **Attention**, the complete input sentences aren't required to be encoded into a single vector. Instead, the decoder attends to different elements in the input sentence at each step of output generation.
- The previous generation of recurrent models had long paths between input and output words. For a 50-word sentence, the decoder had to recall information from 50 steps ago for the first word (and that data had to be **squeezed** into a single vector).



Attention and Transformers

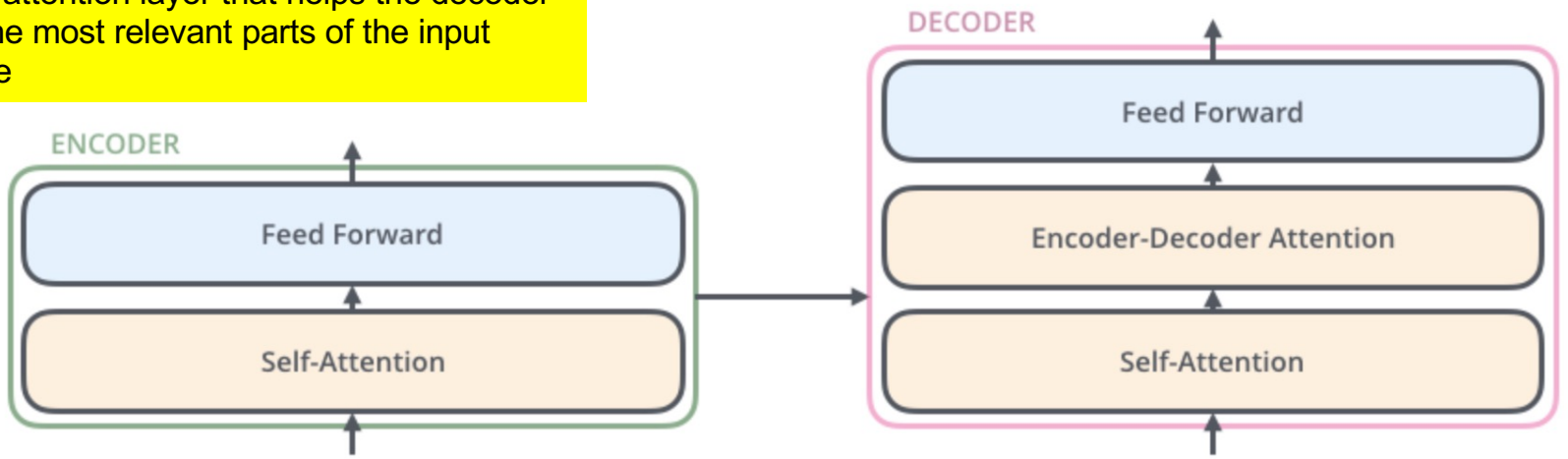
General Architecture:
Transformers share the encoder/decoder architecture, placing a stack of elements in each part of the pipeline (E/D).

The original implementation used a stack of 6 elements at each side.



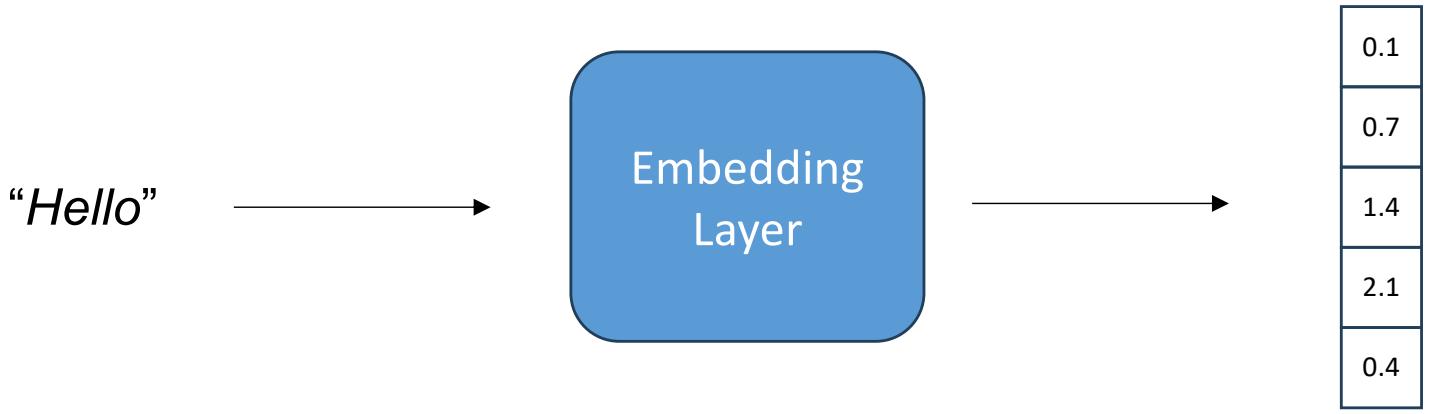
Each **Encoder** is divided into two parts: a **Self-Attention** layer followed by a **Feed-forward** (**Dense**) one.

The **Decoder** has a similar structure, but also uses an attention layer that helps the decoder to find the most relevant parts of the input sentence



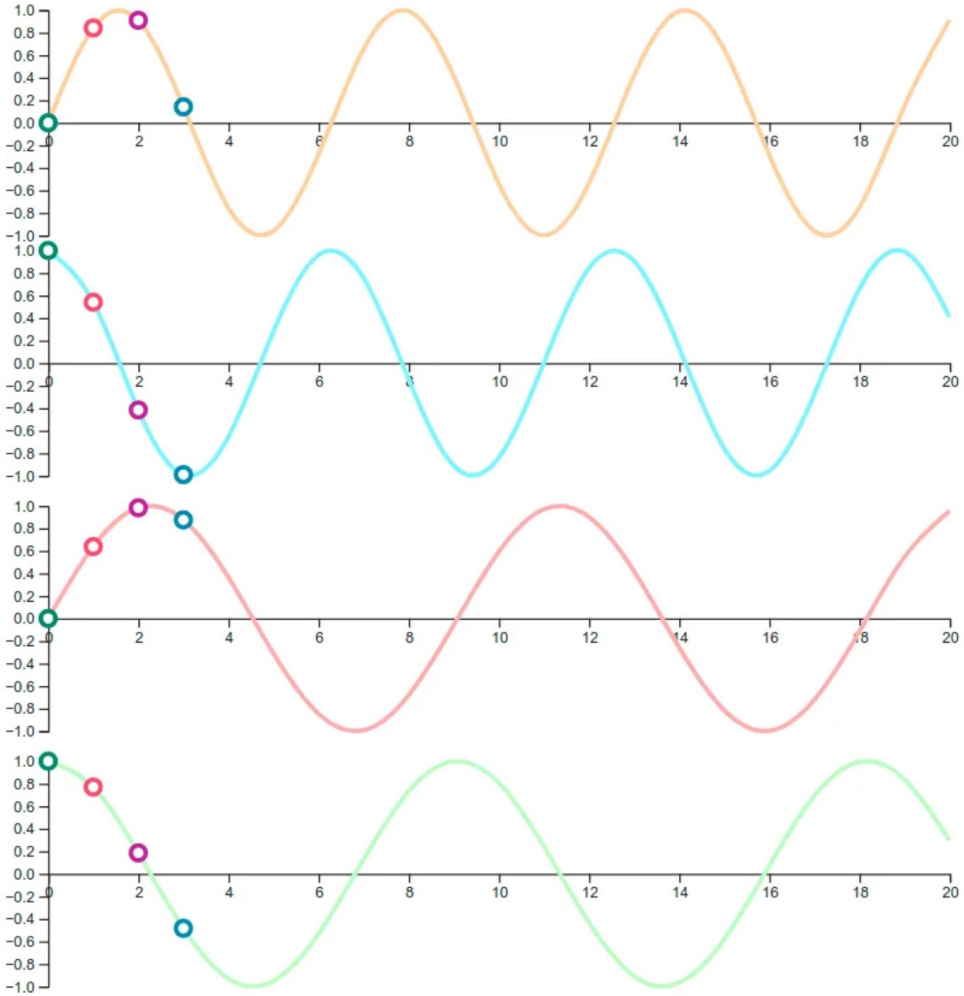
Attention and Transformers - Input Embedding

- The process starts (before feeding the input data to the first Encoder), by obtaining latent representations of the input elements.
- In practice, this first encoder begins by converting input tokens - words or subwords - into vectors using **Embedding layers**.
- These embeddings should capture the semantic meaning of the tokens and convert them into numerical vectors.
  - It is a more sophisticated variant of the "one-hot encoding" previously saw.



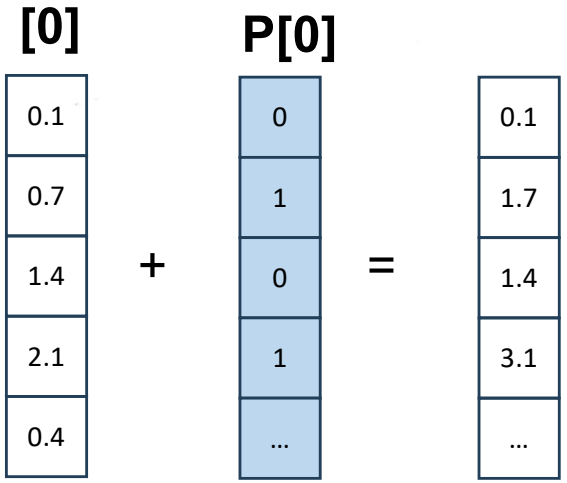
- As Transformers do not have a recurrence mechanism like RNNs, "**Positional encodings"** added to the input embeddings to provide information about the position of each token in the sequence. This allows them to understand the position of each word within the sentence.

Positional Encoding



p0	p1	p2	p3	
0.000	0.841	0.909	0.141	i=0
1.000	0.540	-0.416	-0.990	i=1
0.000	0.638	0.983	0.875	i=2
1.000	0.770	0.186	-0.484	i=3

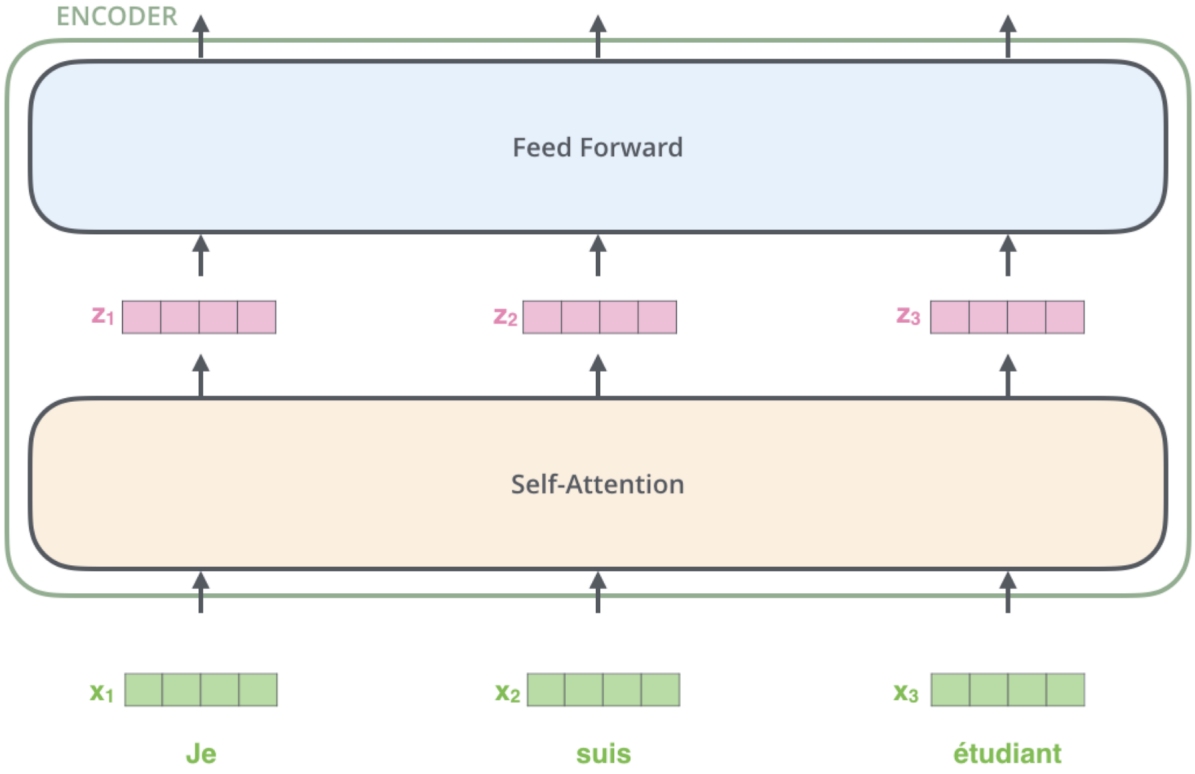
A set of sin() and cos() functions of **different frequencies** are used. This way, each input element is combined (added) to a vector that contains information about the position of the element within the sequence



Embedding with positional context

Attention and Transformers

- Most encoders receive a list of input vectors $\boldsymbol{x}$, each of the size 512.
- After embedding the elements $\boldsymbol{x}_i$, each of them flows through each of the two layers of the encoder.



A key property is that each input element $\boldsymbol{x}_i$ follows an independent path in the network. There are **dependencies** between these paths in the **self-attention layer**. The **feed-forward layer** does not have any dependencies.

Self Attention Mechanism - Encoder

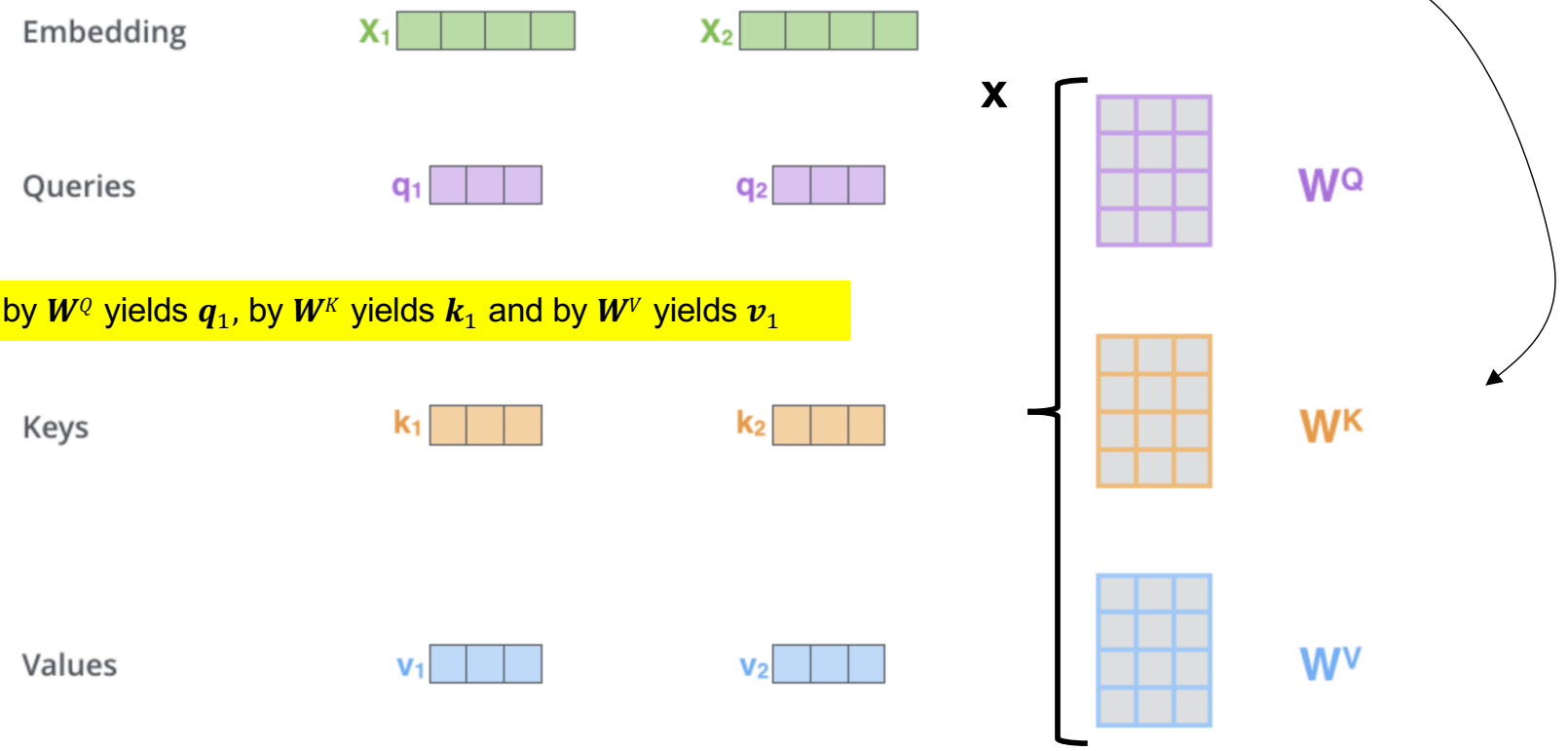
- **Attention** enables the models to relate each element in the input with other elements. For instance, in a given example, the model might learn to connect the element "$\boldsymbol{x}_i$" with "$\boldsymbol{x}_j$".
  - This allows the encoder to focus on different parts of the input sequence as it processes each token
  - It is based on 3 types of vectors: Queries ($\boldsymbol{q}_j$), Keys ($\boldsymbol{k}_j$) and Values ($\boldsymbol{v}_j$)

Intuition: *Imagine a library. We have a specific question (**query**). Books on the shelves have titles on their spines (**keys**) that suggest their content. We compare your query to these titles to decide how relevant each book is, and how much attention to give each book. Finally, we can get the information (**value**) from the relevant books to answer our question.*

- Attention is about how much *weight* the query word (e.g., $\boldsymbol{q}$) should give each word in the sentence (e.g., $\boldsymbol{k}_1$, $\boldsymbol{k}_2$,…). This is obtained via a dot product between the query and all the keys.
  - The dot product measures how similar two vectors are.
  - If the dot product between a query-key pair is high, we pay more attention to it.
  - These dot products then go through a *softmax* which makes the attention scores (across all keys) sum to 1

Self Attention Mechanism - Encoder

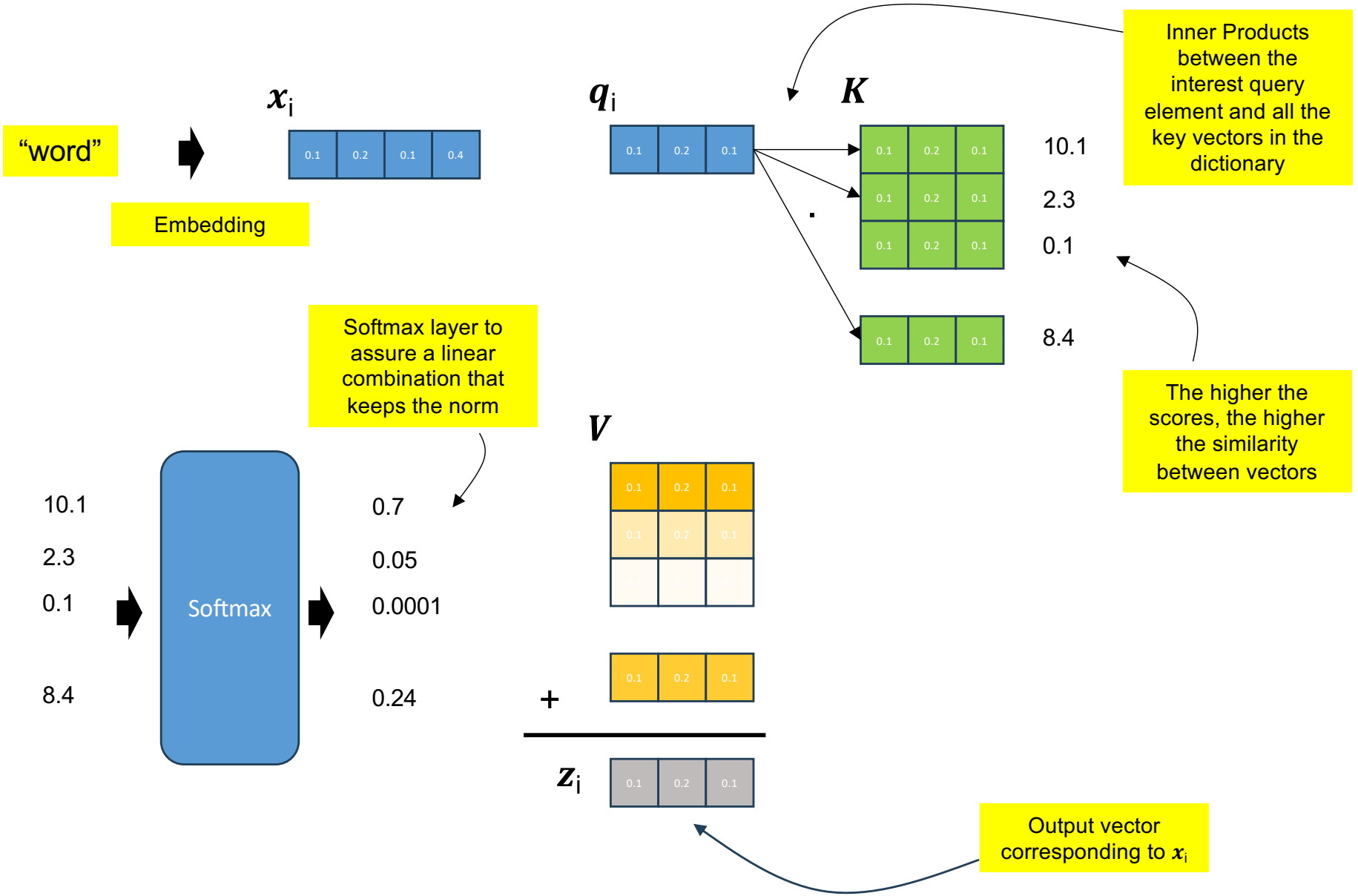
- We start by obtaining 3 vectors for each input element:
  - The **Query**, **Key** and **Value**. They are all created by multiplying the embedding by three matrices (the only ones trained during the learning process).



Multiplying $\boldsymbol{x}_1$ by $\boldsymbol{W}^Q$ yields $\boldsymbol{q}_1$, by $\boldsymbol{W}^K$ yields $\boldsymbol{k}_1$ and by $\boldsymbol{W}^V$ yields $\boldsymbol{v}_1$

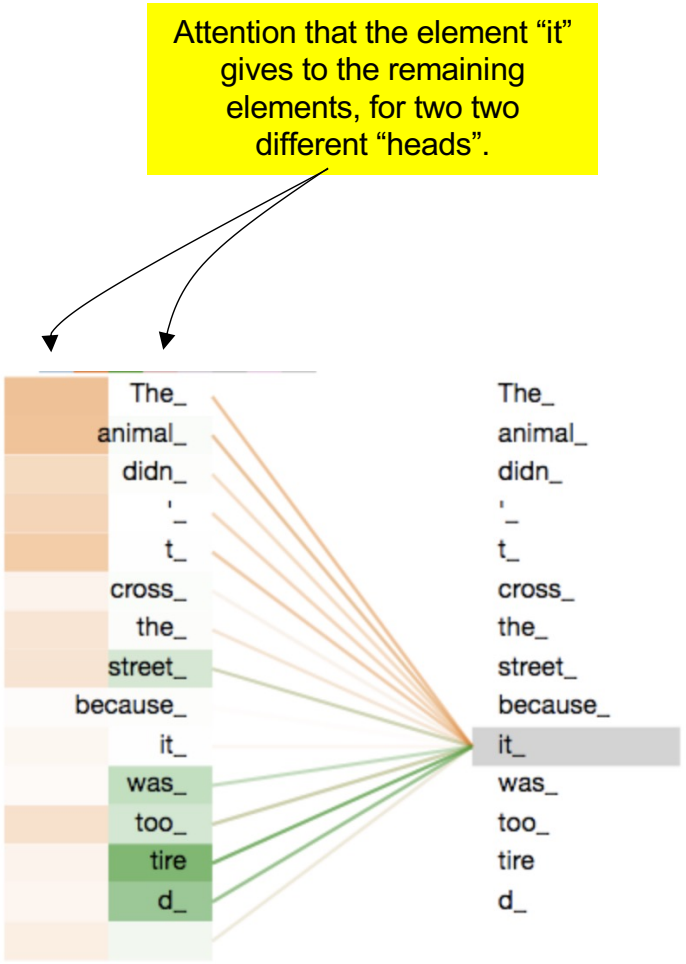
Typically, the dimensionality of the query, key and value vectors is smaller than the dimensionality of the embedding (e.g., 64 << 512)

Self Attention Mechanism - Encoder



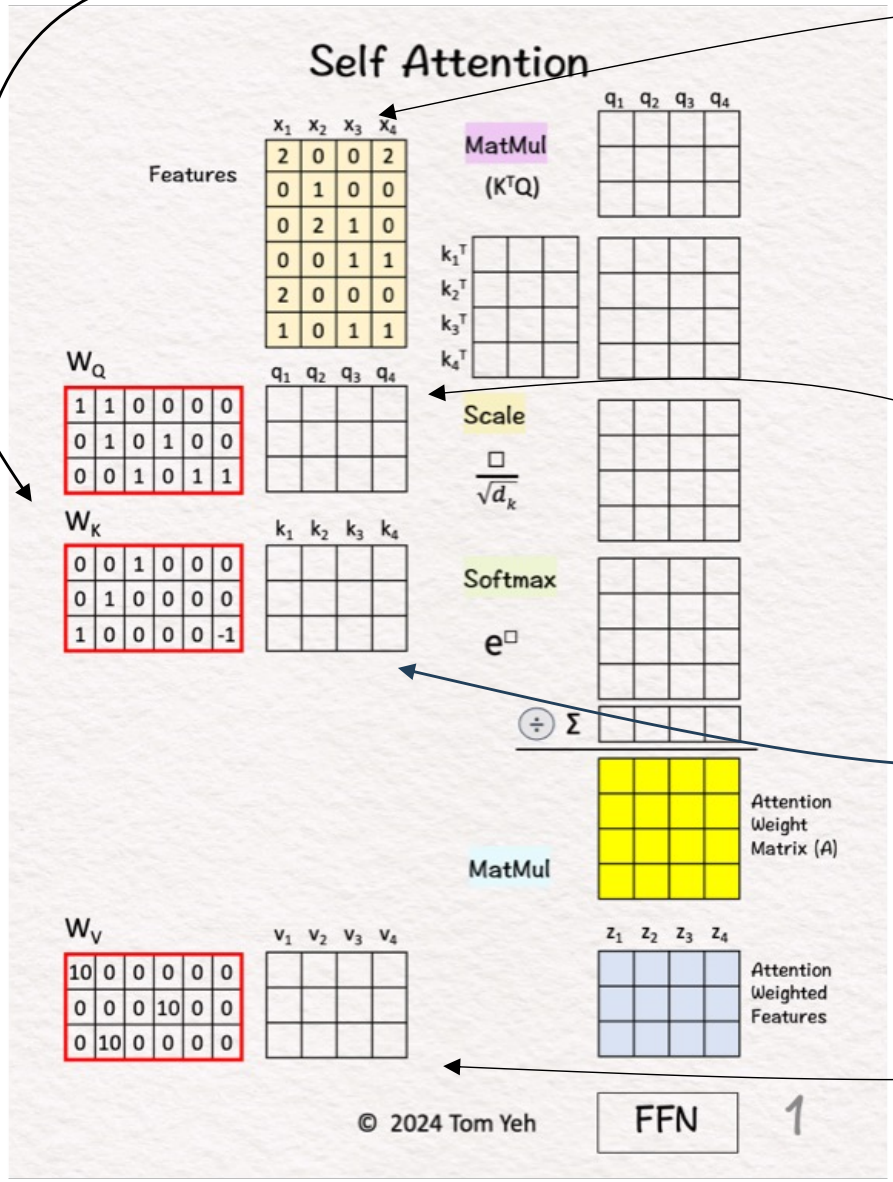
Self Attention Mechanism - Encoder

- Next, the inner product between the query $\boldsymbol{q}_i$, and all the key elements ($\boldsymbol{k}_1$,… $\boldsymbol{k}_n$) measures the similarity of the query with respect to every other element ($\boldsymbol{q}_i.\boldsymbol{k}_j$)
  - Normalizing and applying a *softmax* for all products gives us *how much* of the corresponding value vector should be used in the final sum to obtain the output vector $\boldsymbol{z}_i$.
  - Formally, this step yields the parameters of a linear combination between all the vectors, that will be used to represent the input $\boldsymbol{x}_i$.
- The resulting vector is sent to the feed-forward layer.
- The output of the final encoder layer is a set of vectors, each representing the input sequence with a rich contextual understanding. This output is then used as the input for the decoder in a Transformer model.



❤️ This is the “heart” of the Attention mechanism. To **replace** a representation of the input element by a **linear combination** of the other elements in the space, depending on the similarity/importance of each one with respect to the input. ❤️

Self Attention Mechanism – Matrix Form Example



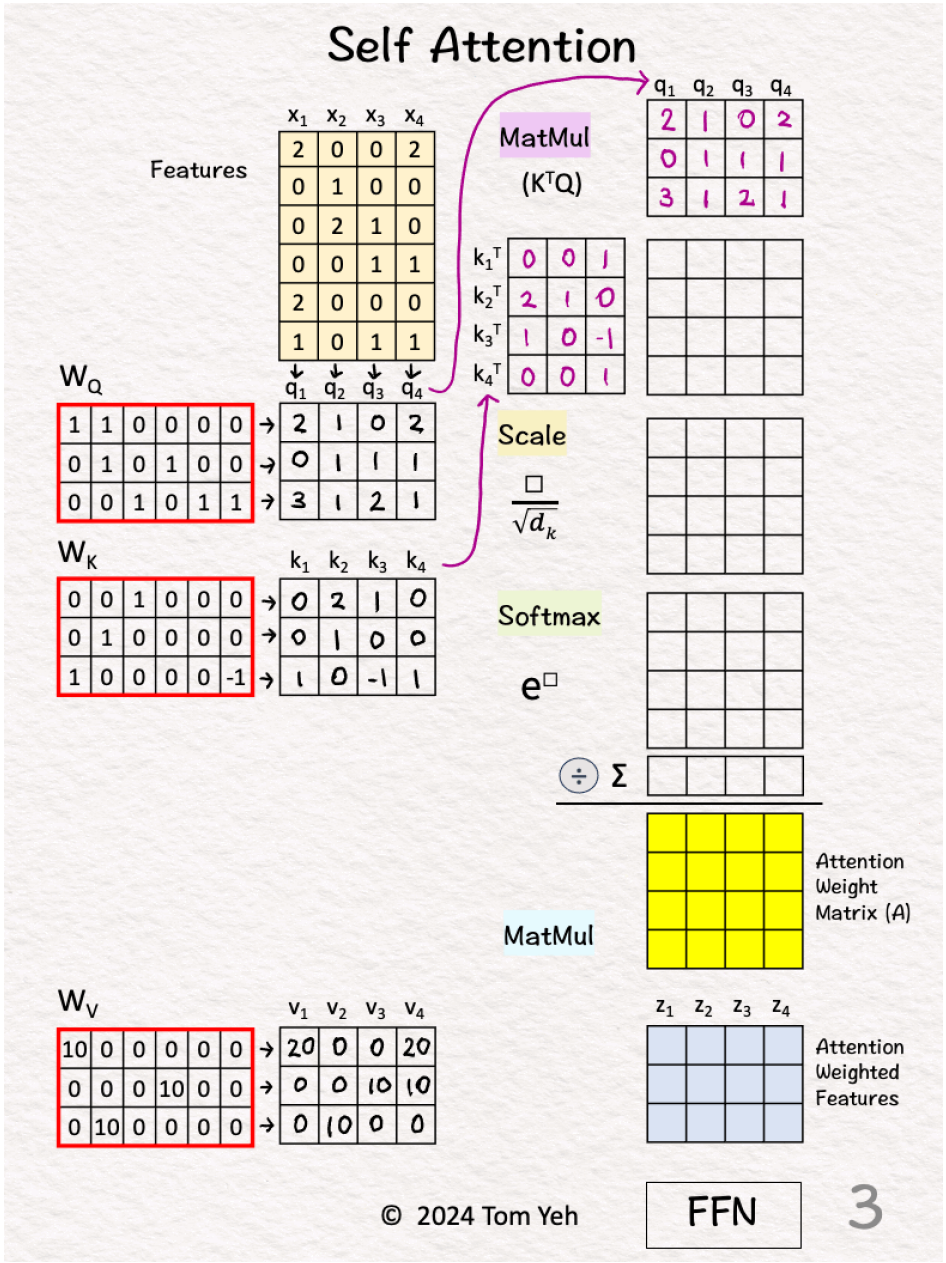
Input: Four 6D vectors x_i

(Suppose that at the current iteration, the Query, Quey and Value matrices have these values)

Step 1. Obtain the query, key and value representations (by multiplying the input vectors by the corresponding matrices)

Credits: https://www.linkedin.com/posts/tom-yeh_deeplearning-transformer-neuralnetwork-activity-7153234215709360129-9J8y/

Self Attention Mechanism – Matrix Form Example



Step 2. Multiply K^T and Q
This is equivalent to taking the dot product between every pair of query and key vectors.

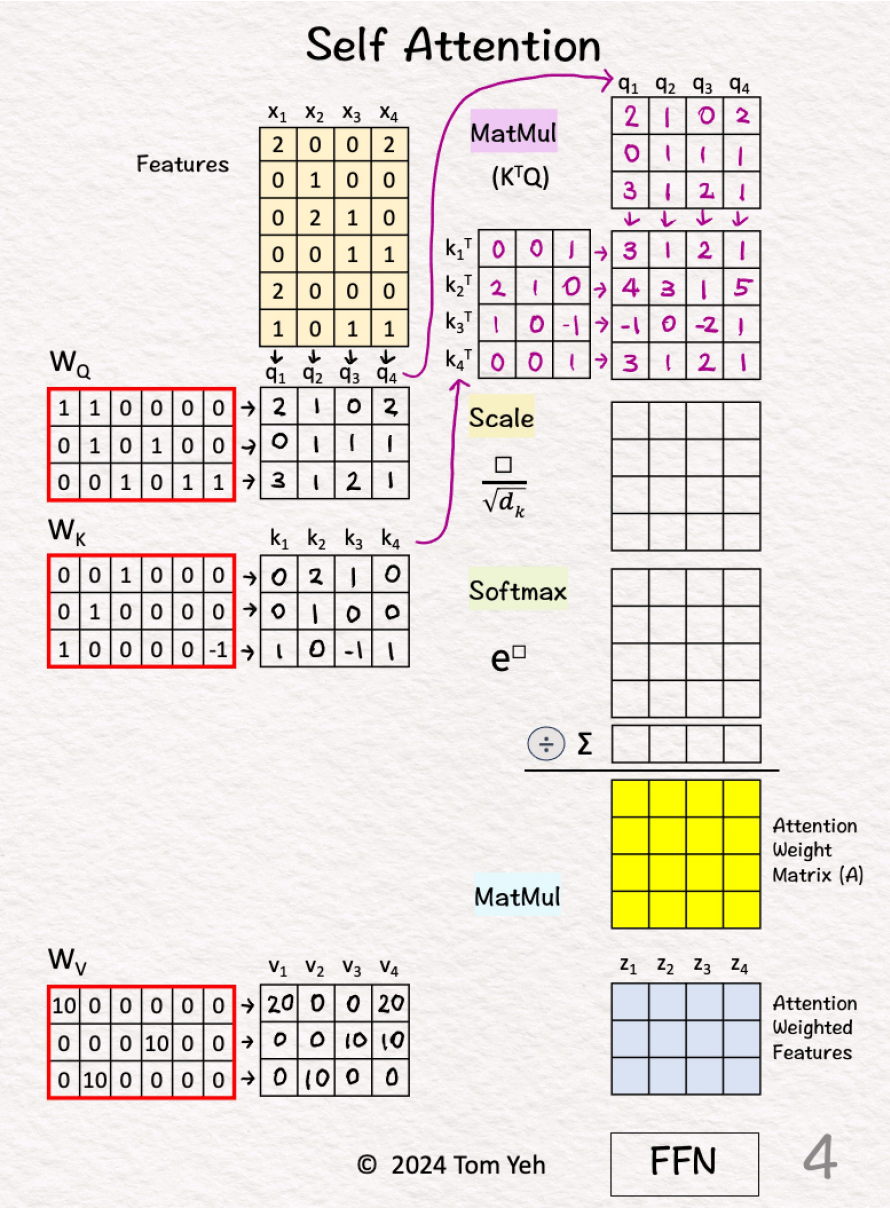
(4x3) x (3x4) = (4x4)

The idea is to use the dot product *as an estimate of the "matching score" between every key-value pair.*

This estimate makes sense because the dot product is the *numerator* of **cosine similarity** between two vectors.

Credits: https://www.linkedin.com/posts/tom-yeh_deeplearning-transformer-neuralnetwork-activity-7153234215709360129-9J8y/

Self Attention Mechanism – Matrix Form Example



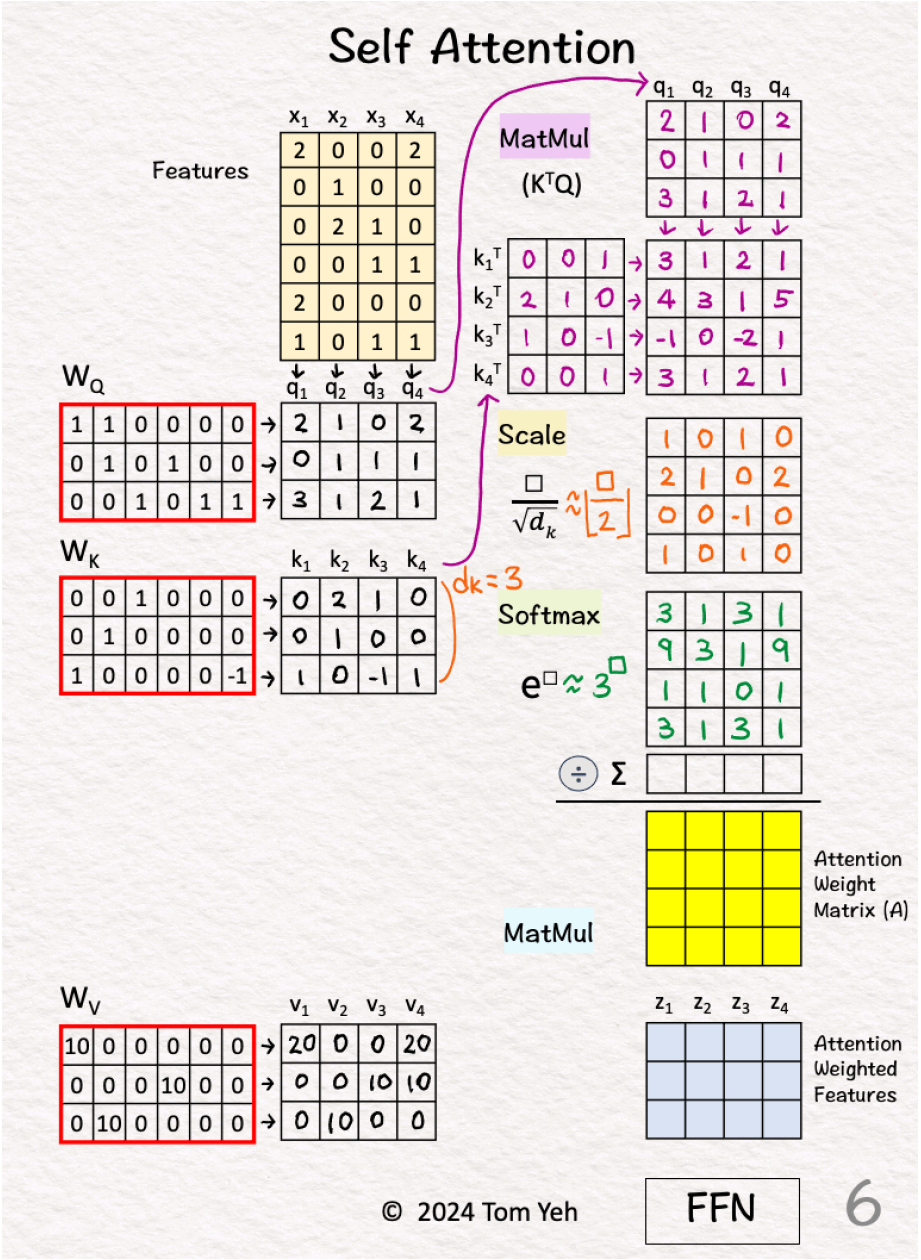
Step 3. Scale each element by the square root of dk, which is the dimension of key vectors (dk=3).

The purpose is to normalize the impact of the dk on matching scores, even if we scale dk to 32, 64, or 128.

To simplify hand calculation, we approximate [□/sqrt(3)] with [floor(□/2)].

Credits: https://www.linkedin.com/posts/tom-yeh_deeplearning-transformer-neuralnetwork-activity-7153234215709360129-9J8y/

Self Attention Mechanism – Matrix Form Example



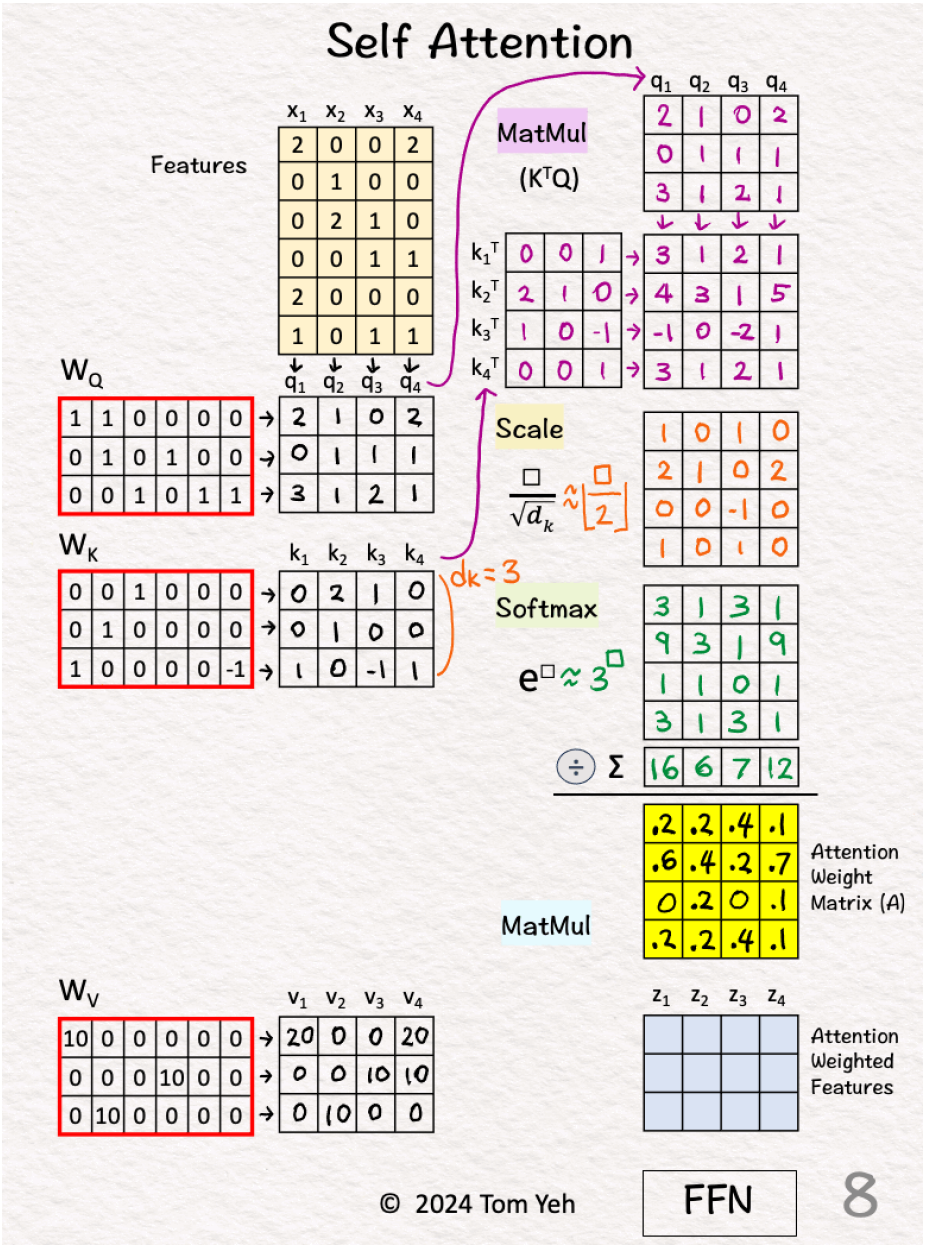
Step 4. Softmax: e^x

- Raise e to the power of the number in each cell

- To simplify hand calculation, we approximate e^□ with 3^□.

Credits: https://www.linkedin.com/posts/tom-yeh_deeplearning-transformer-neuralnetwork-activity-7153234215709360129-9J8y/

Self Attention Mechanism – Matrix Form Example



Step 5. Softmax: e^x

- Raise e to the power of the number in each cell

- To simplify hand calculation, we approximate e^□ with 3^□.

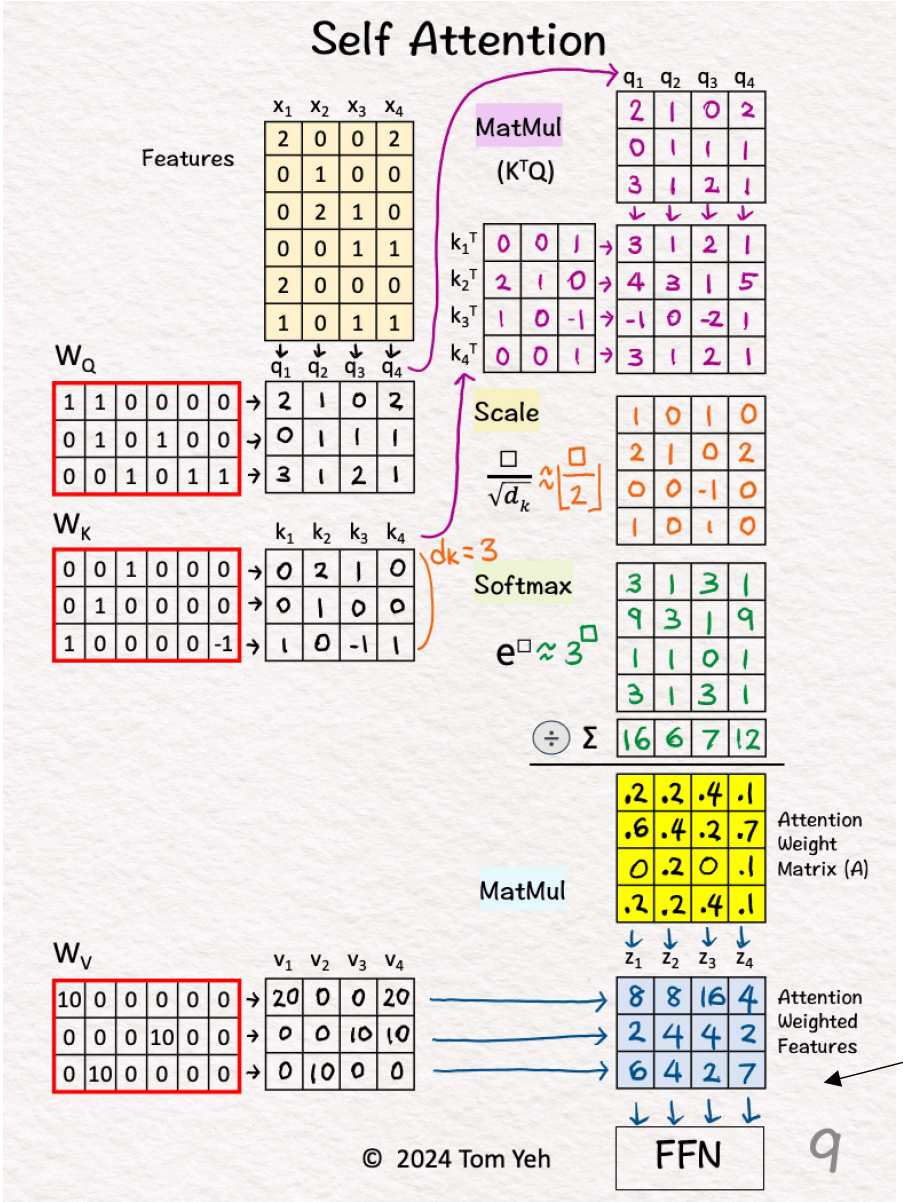
For each column, divide each element by the column sum

The purpose is normalize each column so that the numbers sum to 1. In other words, each column is a probability distribution of attention, and we have four of them.

The result is the Attention Weight Matrix (A) (yellow)

Credits: https://www.linkedin.com/posts/tom-yeh_deeplearning-transformer-neuralnetwork-activity-7153234215709360129-9J8y/

Self Attention Mechanism – Matrix Form Example



Step 7. MatMul
Multiply the value vectors (Vs) with the Attention Weight Matrix (A)

The results are the attention weighted features Zs.

They are fed to the position-wise feed forward network in the next layer.

In:

2	0	0	0	2	1

Out:

8	2	6

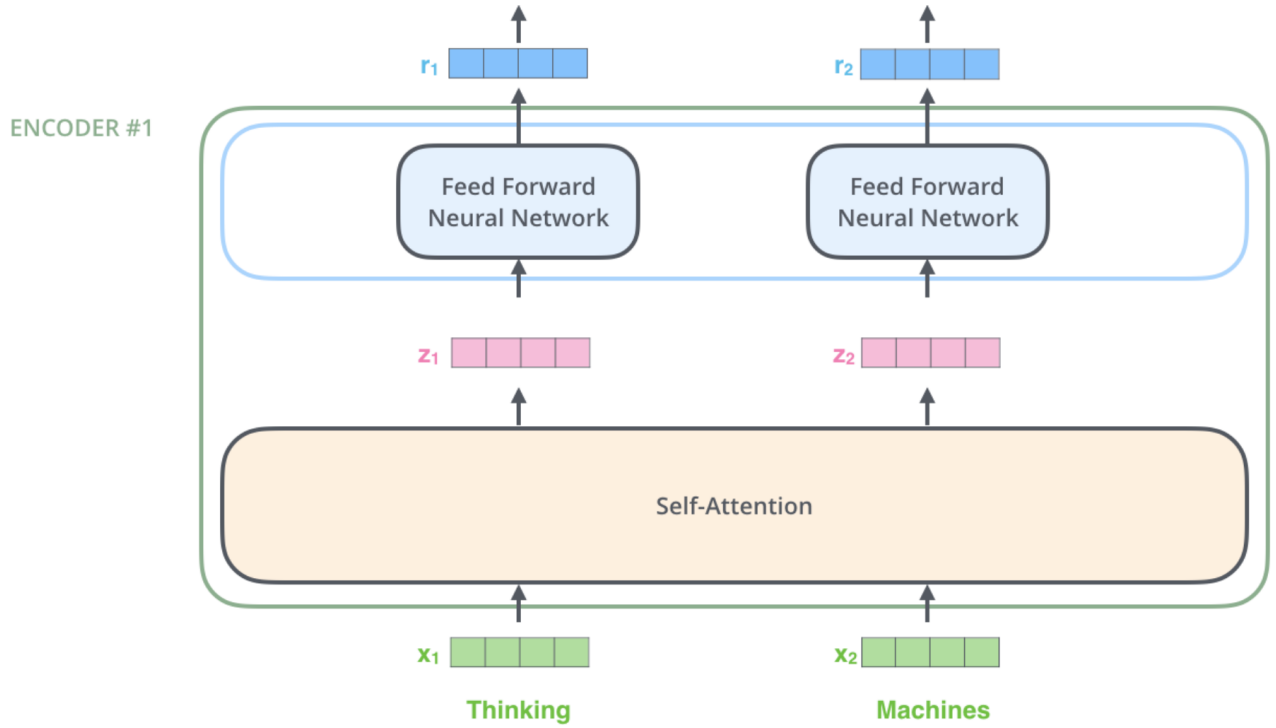
In practice, the first value vector (v_1) pays attention (is replaced by…) to 0.2 v_1 + 0.6 v_2 + 0.2 v_3

This yields one row of the $\boldsymbol{V}_{\text{encdec}}$ matrix (next...)

Credits: https://www.linkedin.com/posts/tom-yeh_deeplearning-transformer-neuralnetwork-activity-7153234215709360129-9J8y/

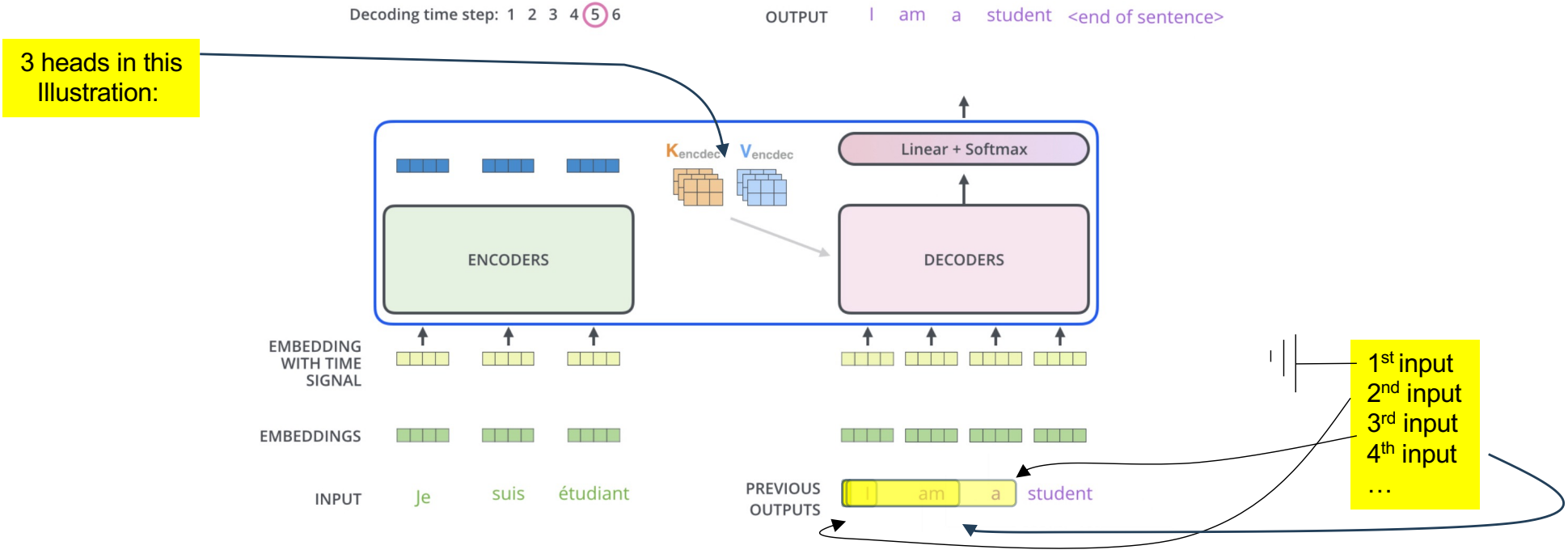
Self Attention Mechanism - Encoder

- The described process allows to map the embedding representations of each input v_i element into the attention vectors z_i.
- The final encoding encoding step consists of passing the z_i elements through a feed forward (dense) layer.



Self Attention Mechanism - Decoder

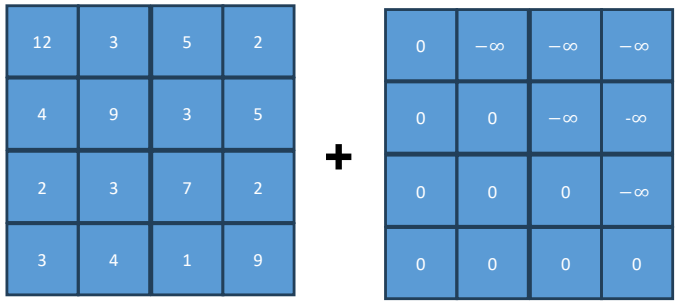
- Using multiple heads, the Value matrices representations (obtained as previously Illustrated) are concatenated into ($\boldsymbol{K}_{\text{encdec}}$ and $\boldsymbol{V}_{\text{encdec}}$) (with as many elements as the number of heads used).
  - They represent the features of the whole input sequence.
  - Are used in the second multi-head attention module of the decoder to relate the input sequence to the masked output of the first multi-head decoder.
- Then, the decoder starts to produce its outputs, until a special element (<END>) indicates that the process must be stopped.
  - During the first iteration, only the "<start>" token is additionally given
  - At each iteration, the set of previous outputs is also given as input.
  - The self attention layers are only allowed to attend to earlier positions in the output sequence. This is done by masking future positions (setting them to " $-\infty$") before the softmax step in the self-attention calculation.



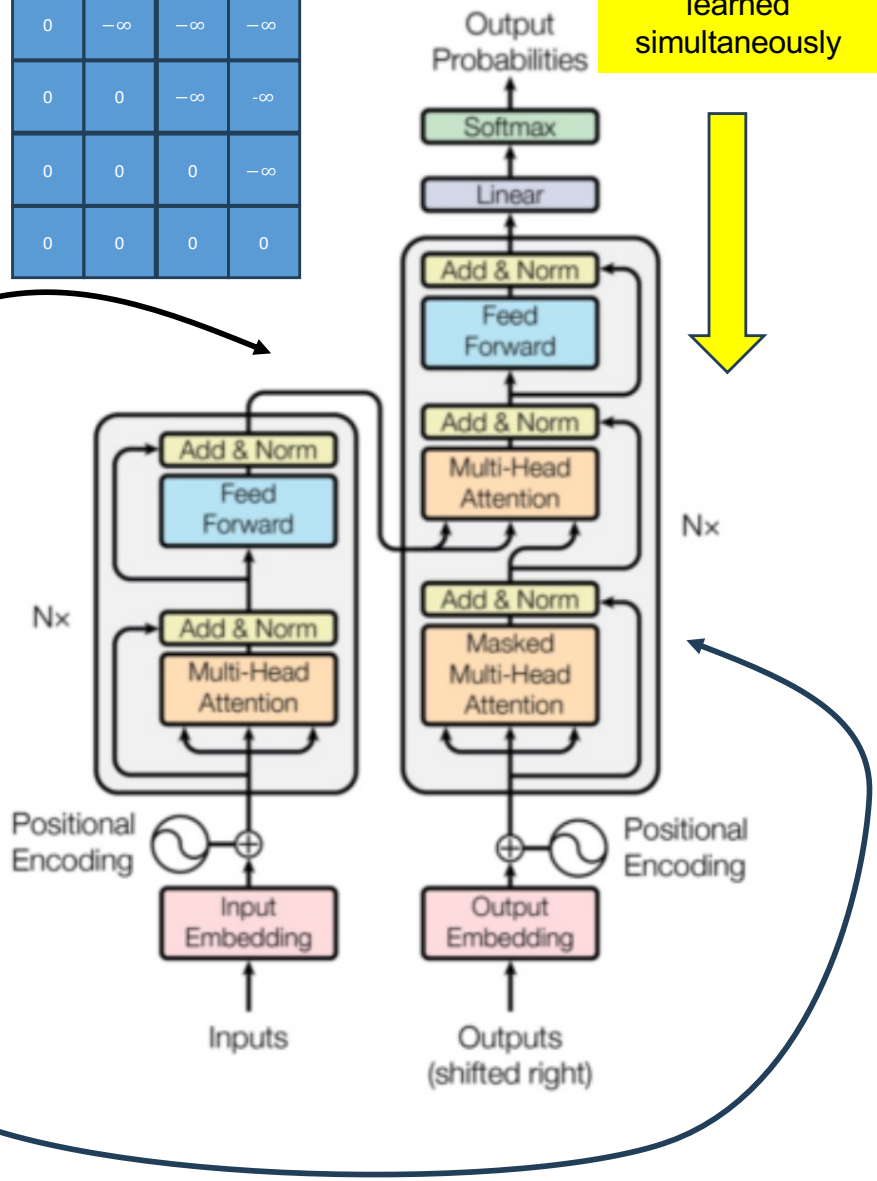
Self Attention Mechanism - Decoder

Gradients flow across the whole network. i.e., all matrices are learned simultaneously

- Note that the decoder has an extra level of complexity. The masked multi-head attention layer, that avoids to pay attention to “future words”

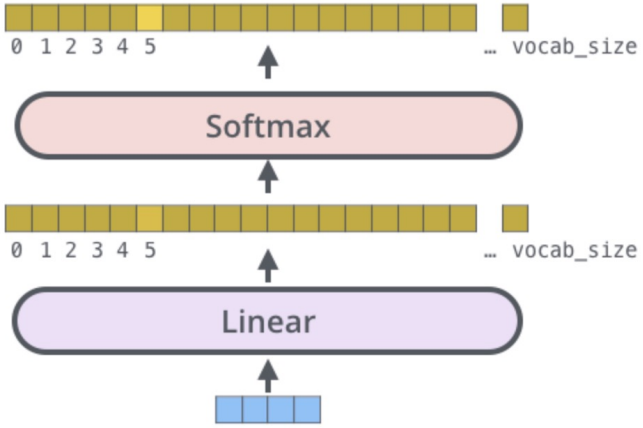


- The output of this layer is then fed to the “Multi-head Attention” layer that uses the Key $\boldsymbol{K}_{\text{encdec}}$ and Value $\boldsymbol{V}_{\text{encdec}}$ outputs given by the encoder.



Self Attention Mechanism - Decoder

- The final part of the decoder works pretty much as a standard “classification” CNN, returning a vector with as many entries as the number of elements in the dictionary. After a “softmax()” layer, the index o the maximum element is found and the corresponding entry in the dictionary returned.



- Real-Life Well-Known Transformers:
  - Google's 2018 release of **BERT**, an open-source natural language processing framework, revolutionized NLP with its unique bidirectional training. Pre-trained on Wikipedia, excels in various NLP tasks, prompting Google to integrate it into its search engine for more natural queries.
  - **LaMDA** (Language Model for Dialogue Applications) is a Transformer-based model developed by Google, designed specifically for conversational tasks, and launched during in 2021. They are designed to generate more natural and contextually relevant responses, enhancing user interactions in various applications.
  - **ChatGPT**, developed by OpenAI, are advanced generative models known for their ability to produce coherent and contextually relevant text. They are suitable for content creation, conversation, language translation, ... GPT's architecture enables it to generate text that closely resembles human writing, making it useful in applications like creative writing, customer support, and even coding assistance.