# Natural Language Processing and Large Language Models

Corso di Laurea Magistrale in Ingegneria Informatica


Lesson 9

# Transformers I

**Nicola Capuano and Antonio Greco**

**DIEM – University of Salerno**

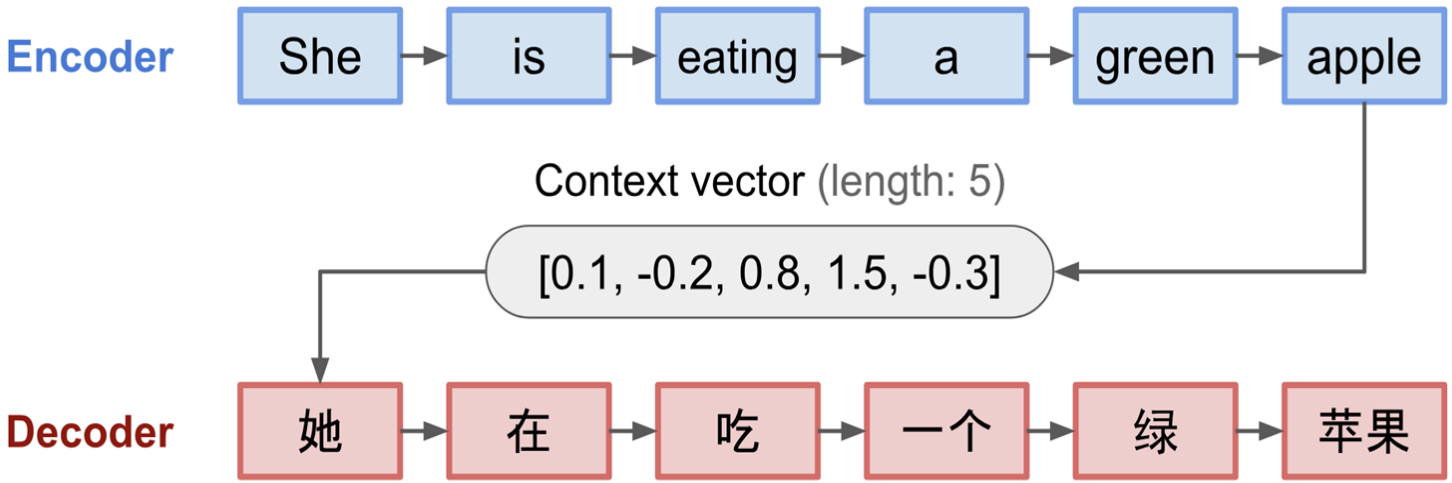
# Outline

- Limitations of RNNs
- Transformer
- Transformer's Input
- Self Attention

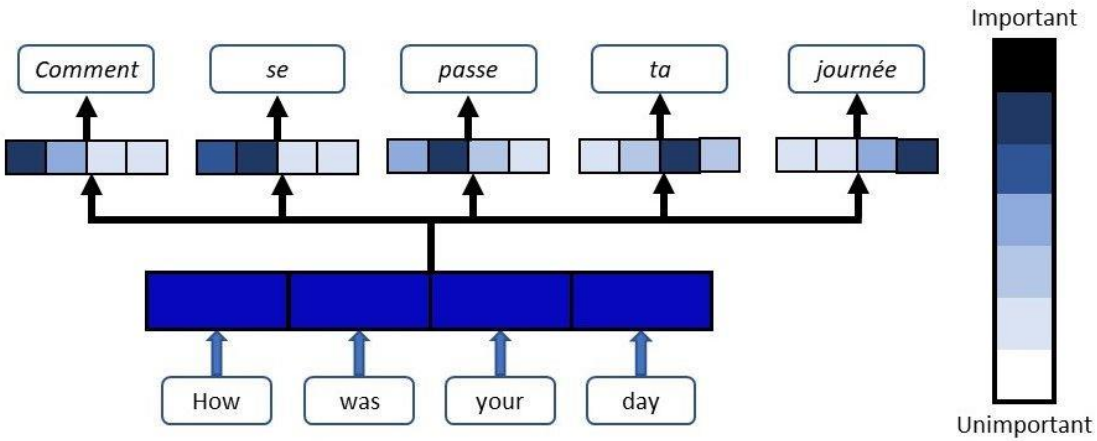
# Limitations of RNNs

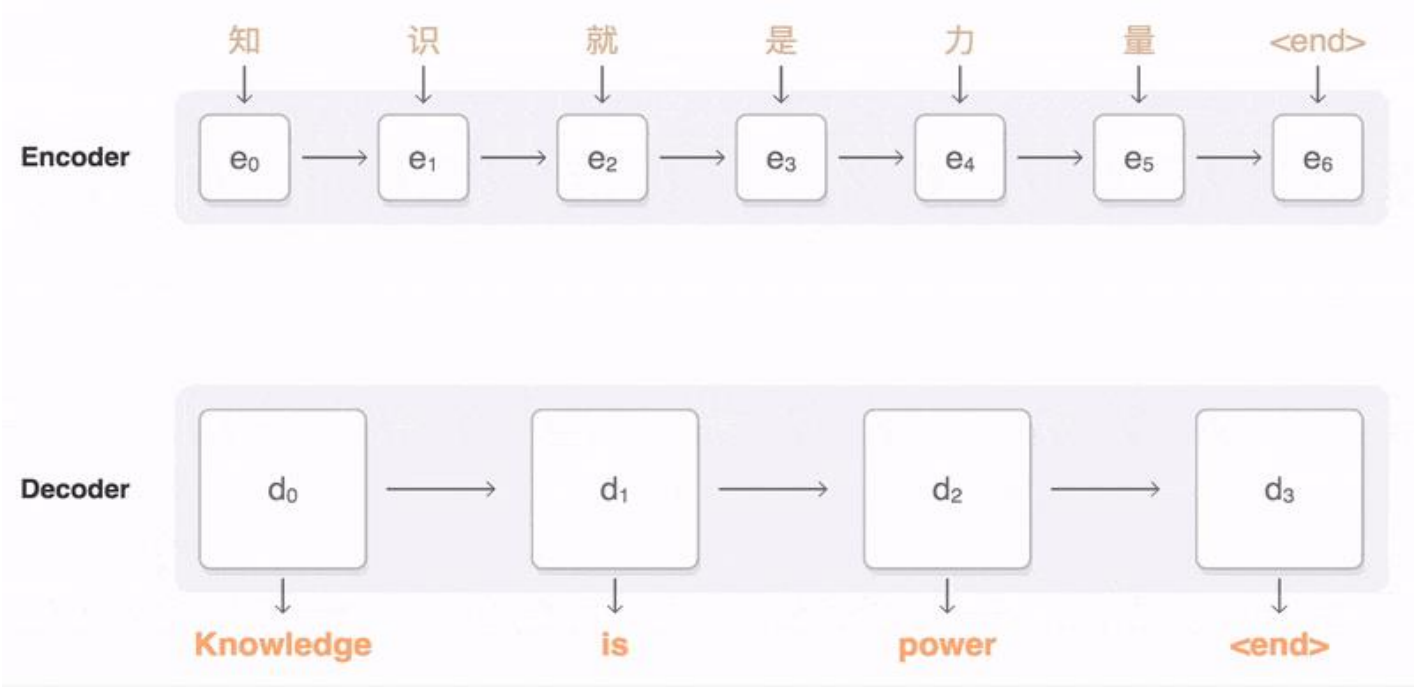
# Limitations of RNNs

- RNNs lack of long-term memory (enc-dec models)
- RNNs are extremely slow to train (for long series)
- RNNs suffer of the vanishing gradient problem



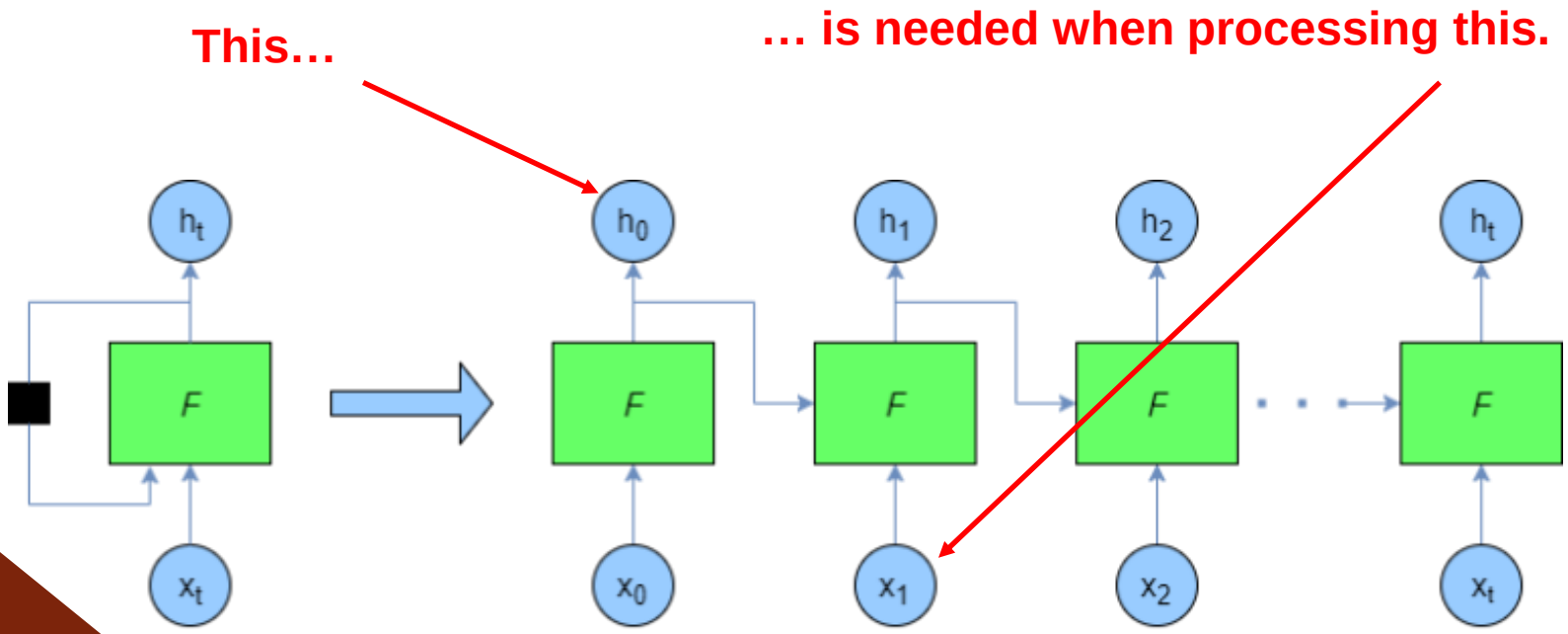
# RNNs lack of long-term memory



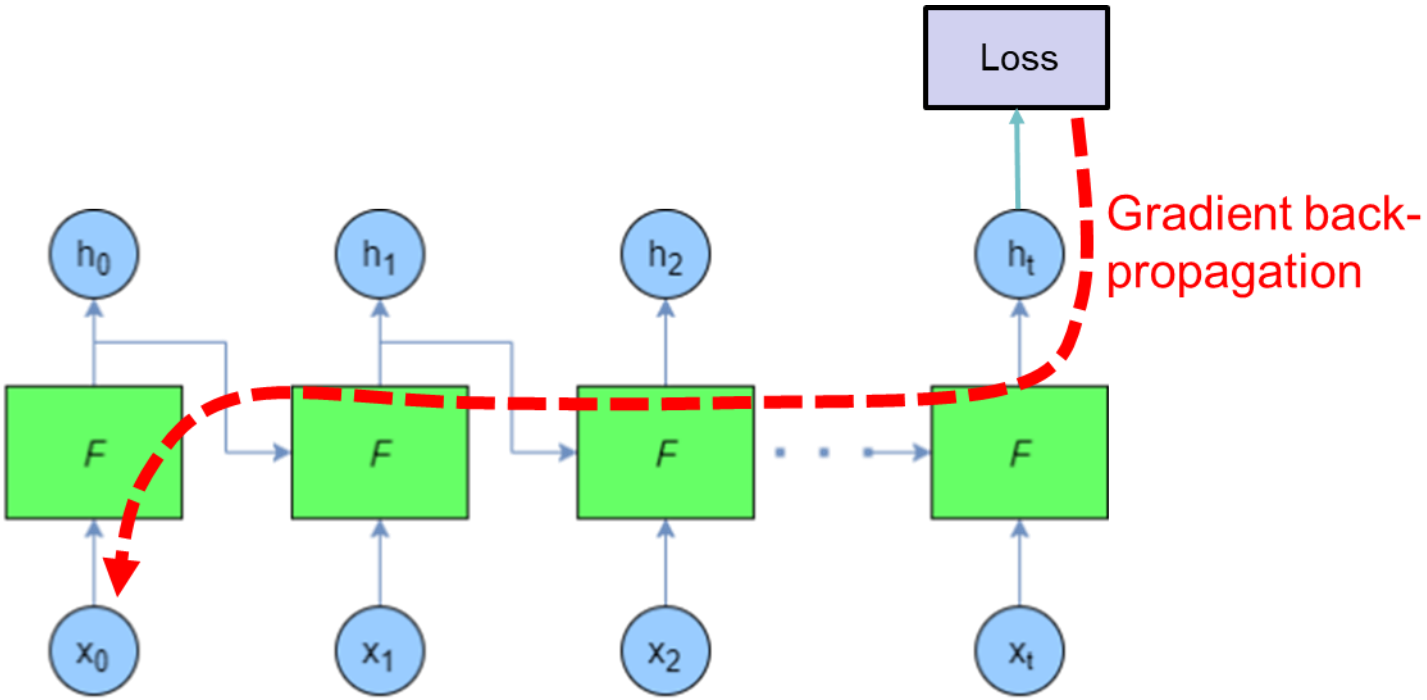
# RNNs are extremely slow to train

- Processing is inherently sequential
- The network can not start processing $x_i$ until it has finished with $x_{i-1}$
- Thus the network cannot exploit the massive parallelism available in a modern GPU!



# RNNs suffer of the vanishing gradient problem

- The other problem is related to Vanishing gradient/exploding gradient
- During backpropagation through time (BPTT) the same function F is traversed many times



# RNNs suffer of the vanishing gradient problem

- The other problem is related to Vanishing gradient/exploding gradient
- During backpropagation through time (BPTT) the same function F is traversed many times
- $$\frac{\partial Loss}{\partial h_0} = \frac{\partial Loss}{\partial h_1} \cdot \frac{\partial F}{\partial h_0} = \frac{\partial Loss}{\partial h_2} \cdot \frac{\partial F}{\partial h_1} \cdot \frac{\partial F}{\partial h_0} = \frac{\partial Loss}{\partial h_3} \cdot \frac{\partial F}{\partial h_2} \cdot \frac{\partial F}{\partial h_1} \cdot \frac{\partial F}{\partial h_0}$$
$$= \frac{\partial Loss}{\partial h_4} \cdot \frac{\partial F}{\partial h_3} \cdot \frac{\partial F}{\partial h_2} \cdot \frac{\partial F}{\partial h_1} \cdot \frac{\partial F}{\partial h_0} = \dots$$

**The derivatives of F are multiplied several times by themselves…**

# RNNs suffer of the vanishing gradient problem

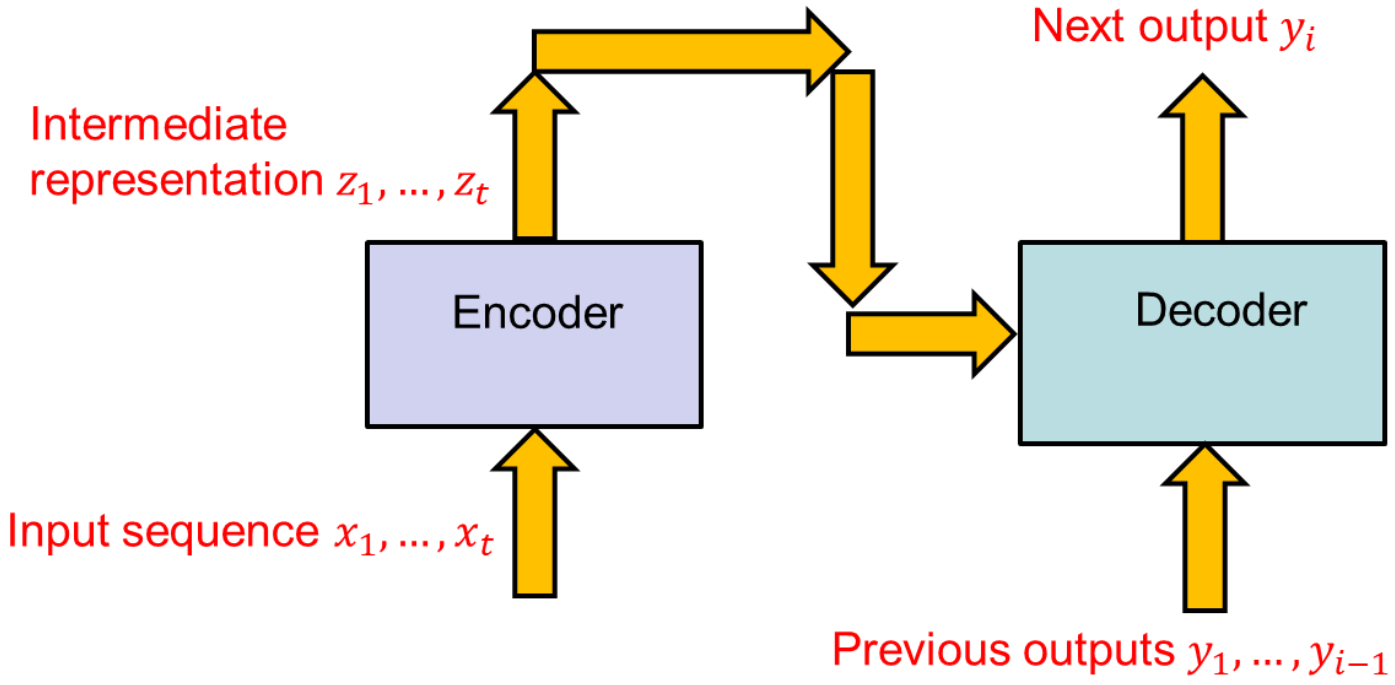
- The other problem is related to Vanishing gradient/exploding gradient
- During backpropagation through time (BPTT) the same function F is traversed many times
- So, if the absolute value of the derivatives of F is small, in this process it will become smaller and smaller… (vanishing gradient)
- … and if it is large, it will be come larger and larger (exploding gradient), causing problems to the stability of the algorithm
- Note: the problem is caused by the fact that we are traversing many times (sequence length) the same layer

# Transformer

# Transformer

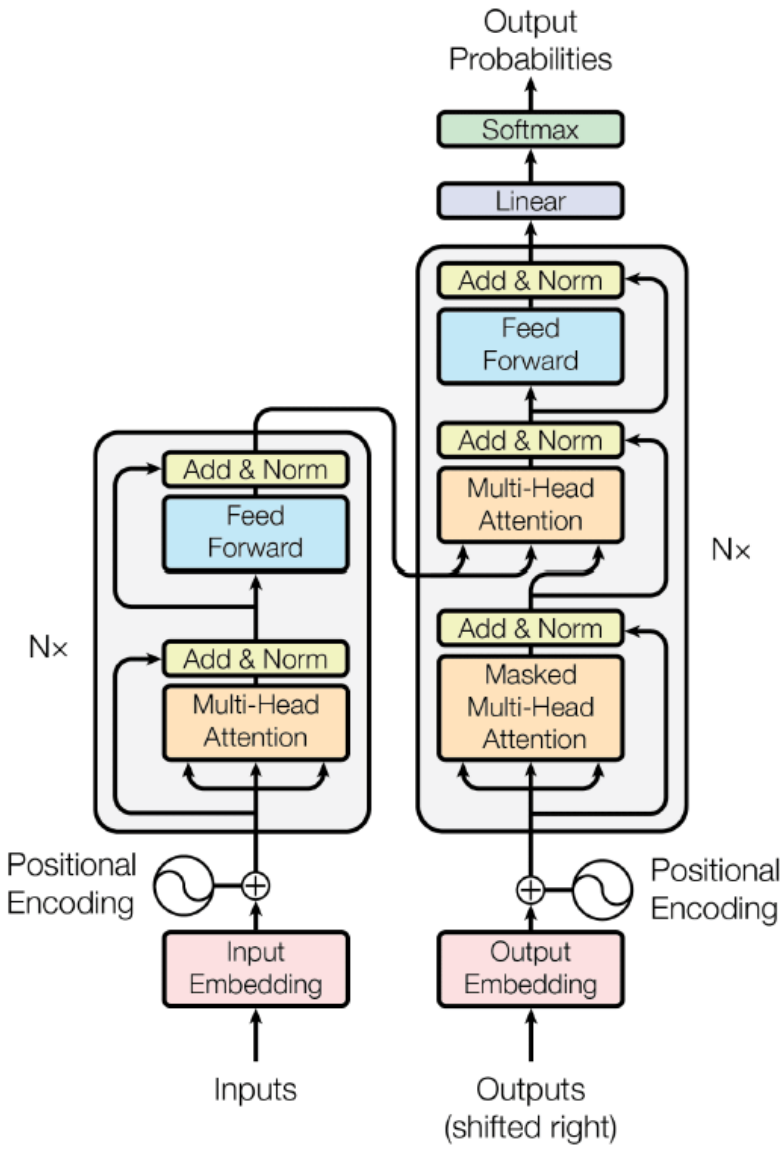
- In 2017, a group of researchers at Google Brain proposed an alternative model for processing sequential data
- In this model, the elements of the sequence can be processed in parallel
- The number of layers traversed does not depend on the length of the sequence (so, no problems with the gradient)
- The model was introduced for language translation (sequence to sequence with different lengths); so, it was called Transformer.
- Subsets of the model can be used for the other sequence processing tasks

# Transformer



# Transformer

- Input
  - Tokenization
  - Input embedding
  - Positional encoding
- Encoder
  - Attention
  - Query
  - Key
  - Value
  - Self Attention
  - Multi-head Attention
  - Add & Norm
  - Feedforward
- Decoder
  - Masked attention
  - Encoder decoder attention
- Output



# Input

# Tokenization

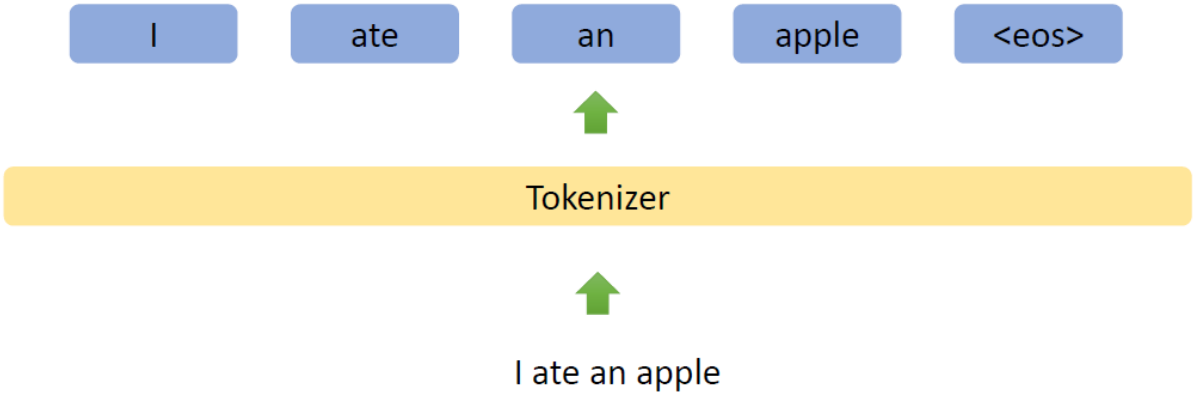
- Representation of text with a set of tokens
- Each token is encoded with a unique id

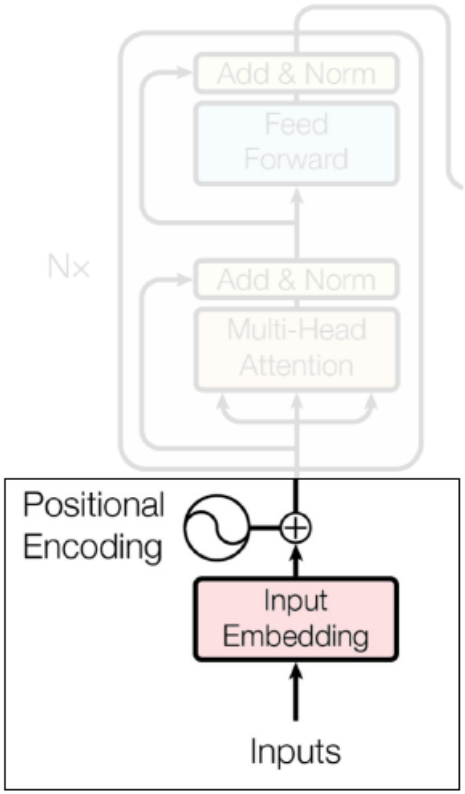
| Tokenization method | Tokens | Token count | Vocab size |
|---|---|---|---|
| Sentence | 'The moon, Earth's only natural satellite, has been a subject of fascination and wonder for thousands of years.' | 1 | # sentences in doc |
| Word | 'The', 'moon,', "Earth's", 'only', 'natural', 'satellite,', 'has', 'been', 'a', 'subject', 'of', 'fascination', 'and', 'wonder', 'for', 'thousands', 'of', 'years.' | 18 | 171K (English[1]) |
| Sub-word | 'The', 'moon', ',', 'Earth', "'", 's', 'on', 'ly', 'n', 'atur', 'al', 's', 'ate', 'll', 'it', 'e', ',', 'has', 'been', 'a', 'subject', 'of', 'fascinat', 'ion', 'and', 'w', 'on', 'd', 'er', 'for', 'th', 'ous', 'and', 's', 'of', 'y', 'ears', '.' | 37 | (varies) |
| Character | 'T', 'h', 'e', ' ', 'm', 'o', 'o', 'n', ',', ' ', 'E', 'a', 'r', 't', 'h', "'", 's', ' ', 'o', 'n', 'l', 'y', ' ', 'n', 'a', 't', 'u', 'r', 'a', 'l', ' ', 's', 'a', 't', 'e', 'l', 'l', 'i', 't', 'e', ',', ' ', 'h', 'a', 's', ' ', 'b', 'e', 'e', 'n', ' ', 'a', ' ', 's', 'u', 'b', 'j', 'e', 'c', 't', ' ', 'o', 'f', ' ', 'f', 'a', 's', 'c', 'i', 'n', 'a', 't', 'i', 'o', 'n', ' ', 'a', 'n', 'd', ' ', 'w', 'o', 'n', 'd', 'e', 'r', ' ', 'f', 'o', 'r', ' ', 't', 'h', 'o', 'u', 's', 'a', 'n', 'd', 's', ' ', 'o', 'f', ' ', 'y', 'e', 'a', 'r', 's', '.' | 110 | 52 + punctuation (English) |



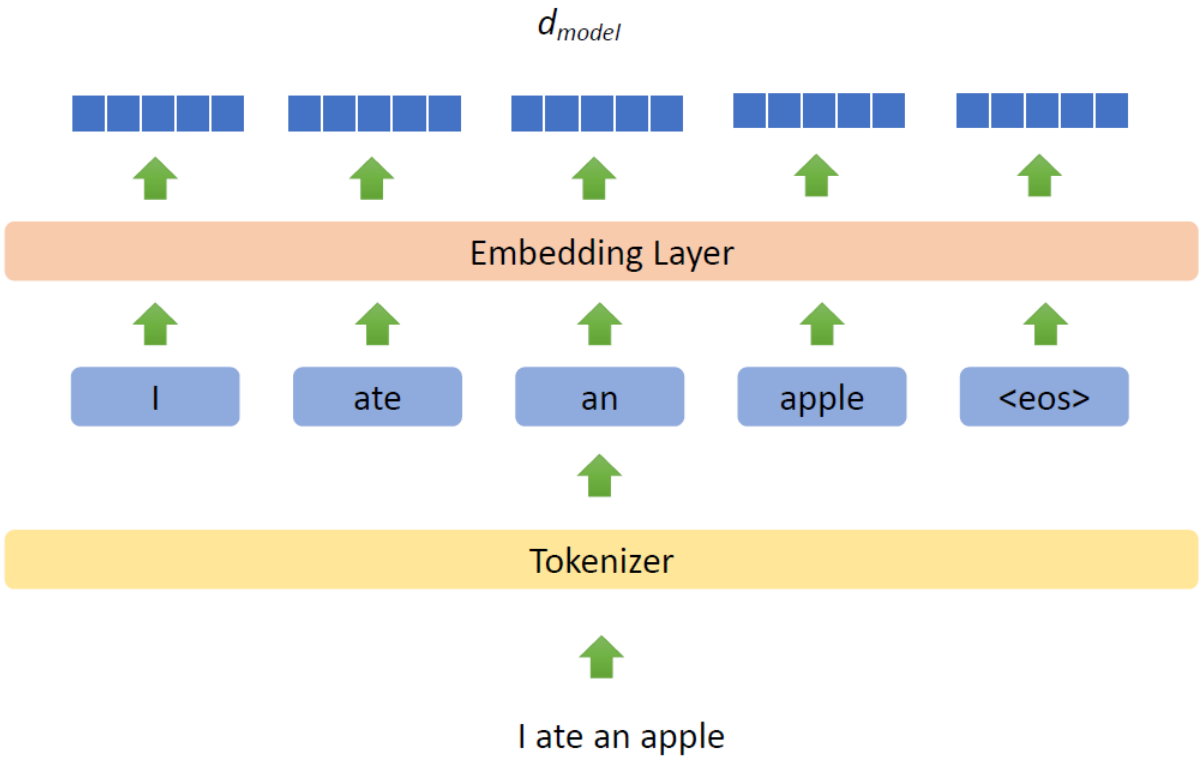
'Source: BBC.com

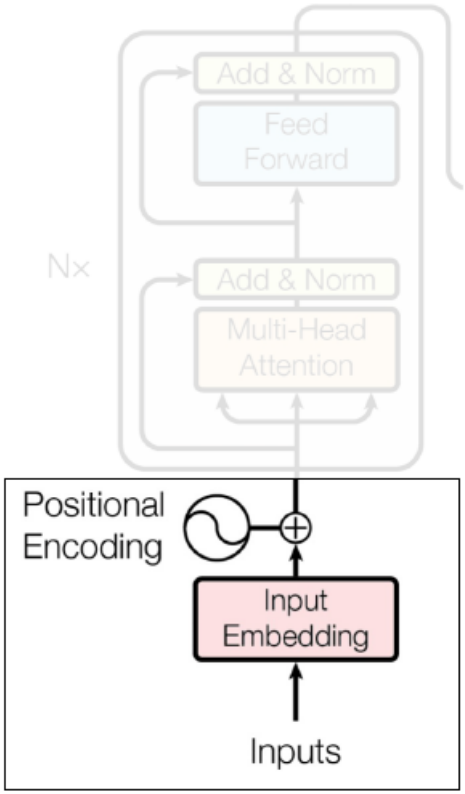
# Tokenization



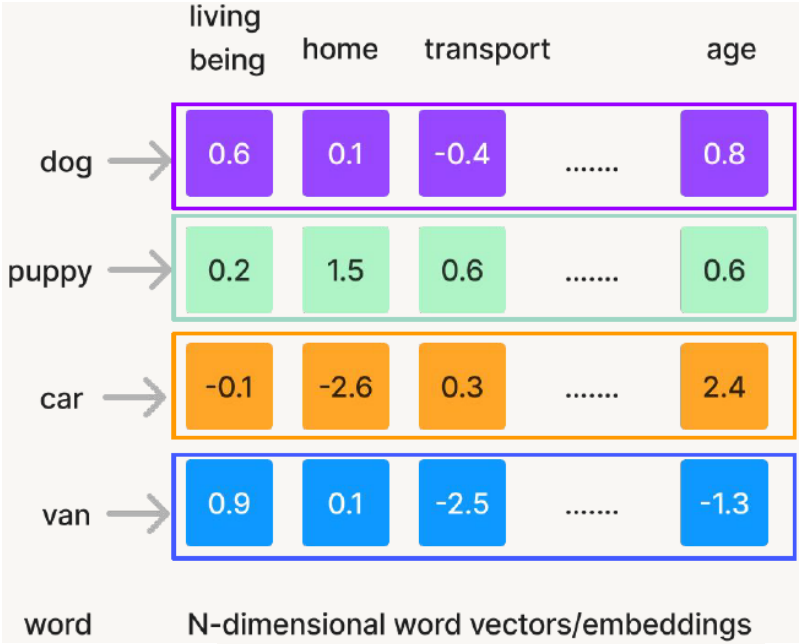
# Input embedding

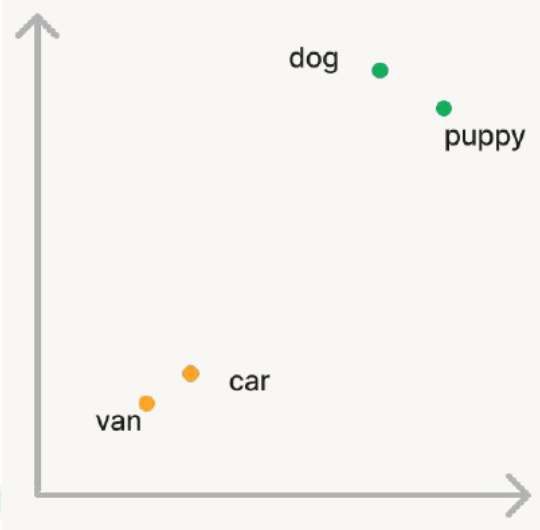


# Input embedding

- Embedding: a representation of a symbol (word, character, sentence) in a distributed low-dimensional space of continuous-valued vectors
- The tokens are projected in a continuous euclidean space
- Correlations among words can be visualized in the embedding space: depending on the task, word embedding can push two words further away or keep them close together.
- Ideally, an embedding captures the semantics of the input by placing semantically similar inputs close together in the embedding space.

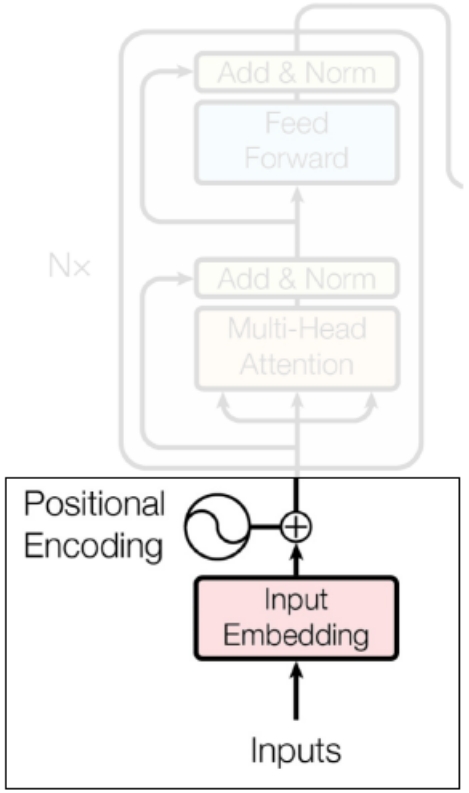


# Positional encoding

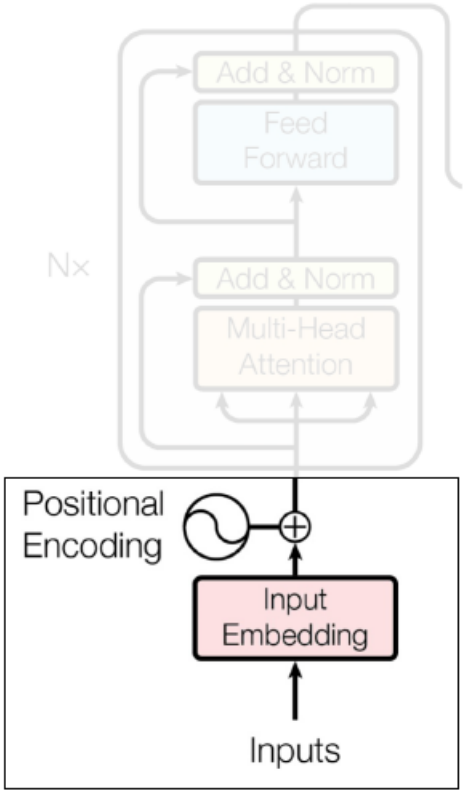
# The importance of order

- Q: With the encoding we have seen so far, is it possible to discriminate between sequences that only differ in the order of the elements?
  - E.g., is it possible to differentiate between "The student is eating an apple" and "An apple is eating the student"?



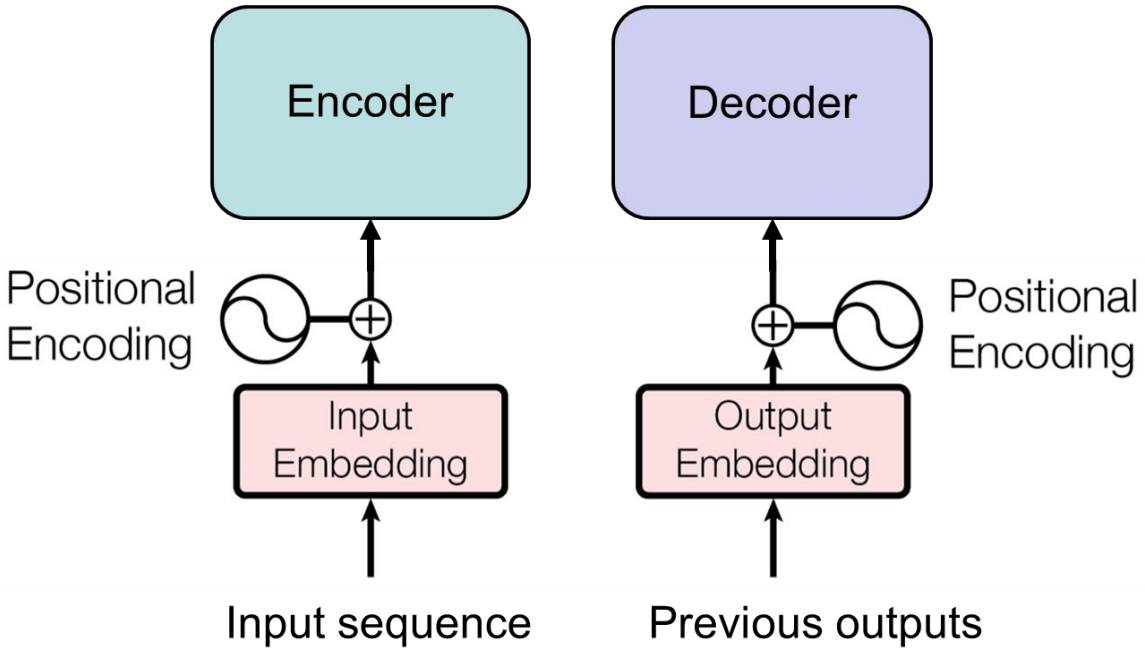
# The importance of order

- Q: With the encoding we have seen so far, is it possible to discriminate between sequences that only differ in the order of the elements?
  - E.g., is it possible to differentiate between "The student is eating an apple" and "An apple is eating the student"?
- A: No, because the output of the attention module does not depend on the order of its keys/values pairs
- So how can we add the information on the order of the sequence elements?



# Positional encoding

- The solution proposed by the authors of the Transformer model is to add a slight perturbation to each element of the sequence, depending on the position within the sequence
- In this way, the same element appearing in different positions would be encoded using slightly different vectors



# Positional encoding

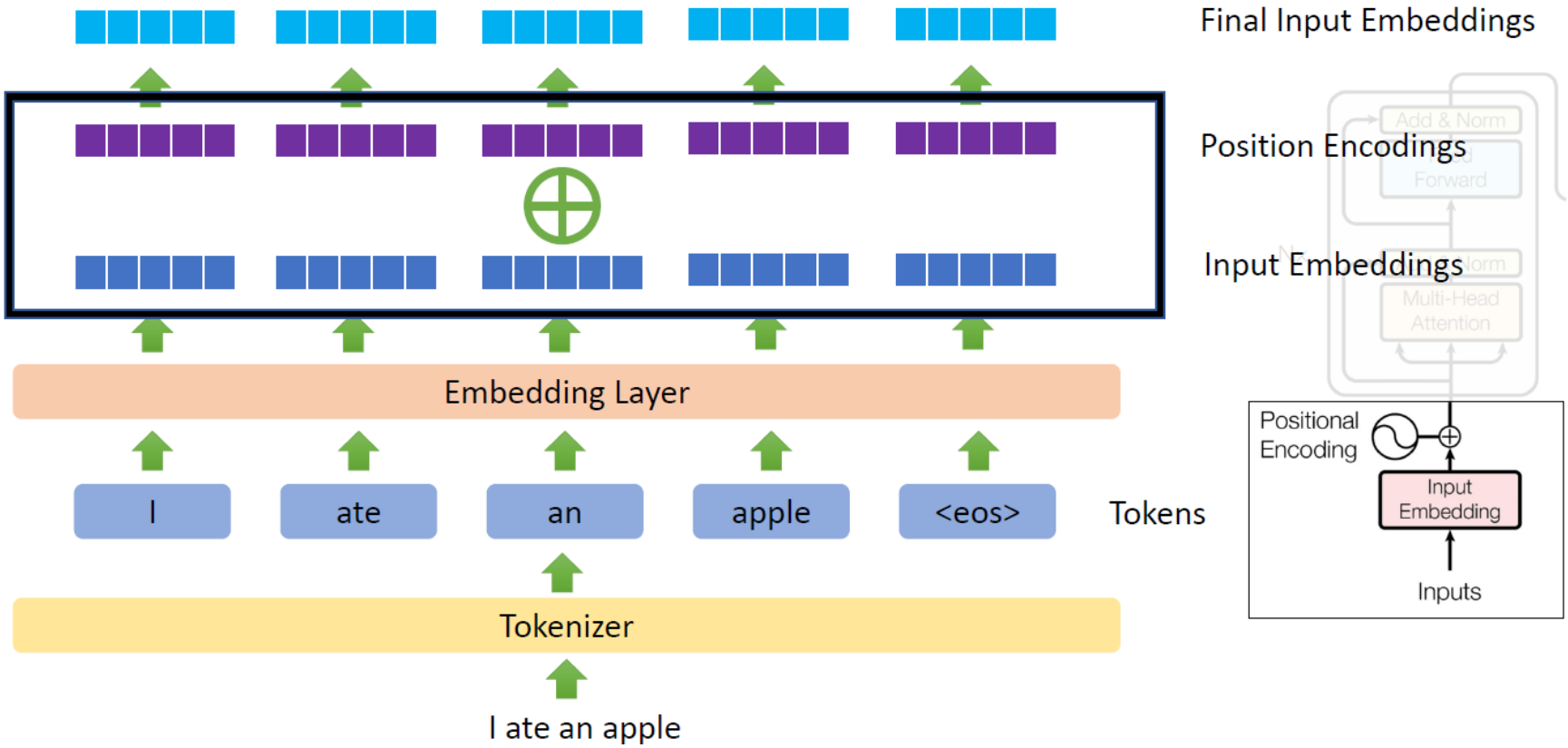
- The position encoding is represented by a set of periodic functions
- In particular, if $d_{model}$ is the size of the embedding, and $pos$ is the position of an element within the sequence, the perturbation to component **i** of the embedding vector representing the element is:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

- The positional encoding is a vector with the same dimension as the input embedding, so it can be added on the input directly.

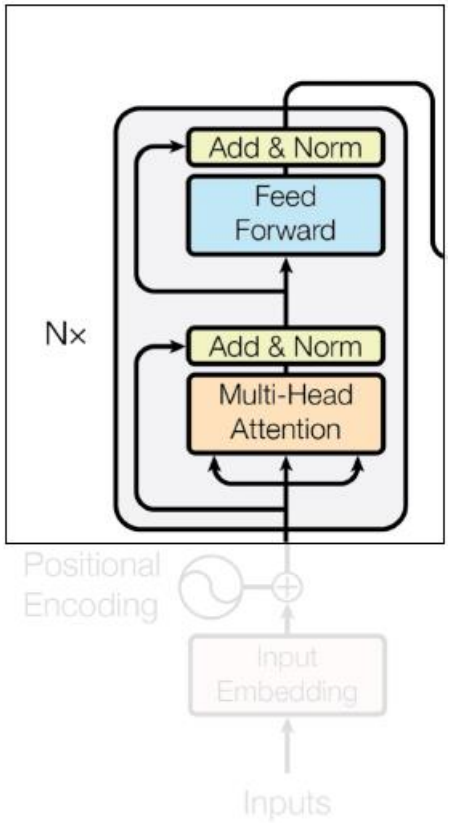
# Positional encoding



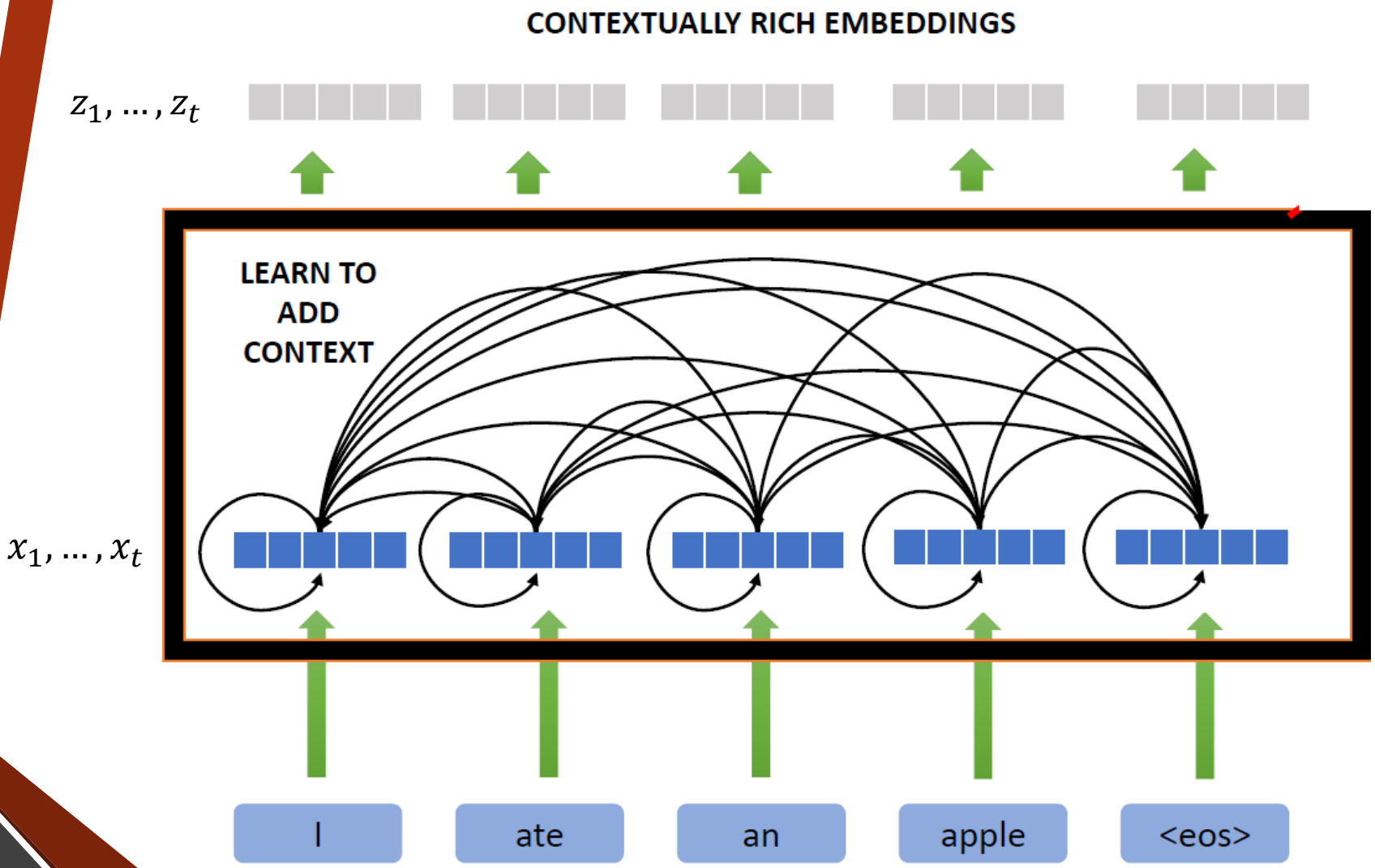
# Encoder

# Encoder

- The **Encoder** transforms an input sequence of vectors $x_1, \dots, x_t$ into an intermediate representation of the same length $z_1, \dots, z_t$
- The vectors $z_1, \dots, z_t$ can be generated in parallel
- Each vector $z_i$ does not depend only on the corresponding $x_i$, but on the whole input sequence $x_1, \dots, x_t$

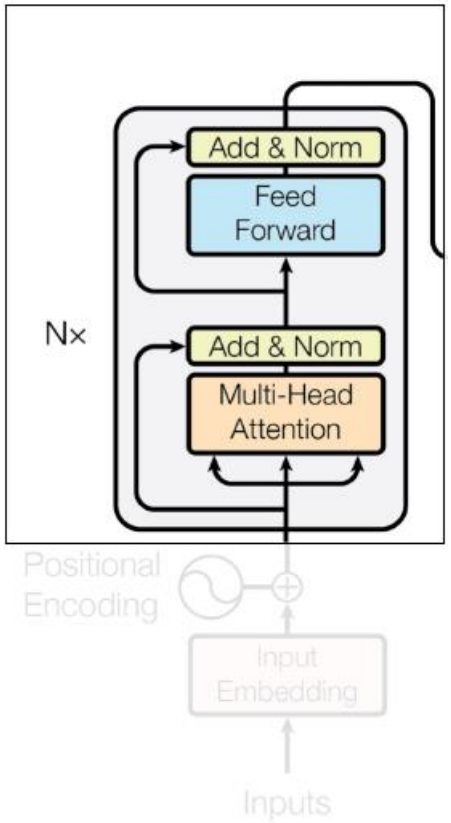


# Encoder


CONTEXTUALLY RICH EMBEDDINGS
$z_1, \ldots, z_t$
LEARN TO ADD CONTEXT
$x_1, \ldots, x_t$
I
ate
an
apple
<eos>

# Encoder

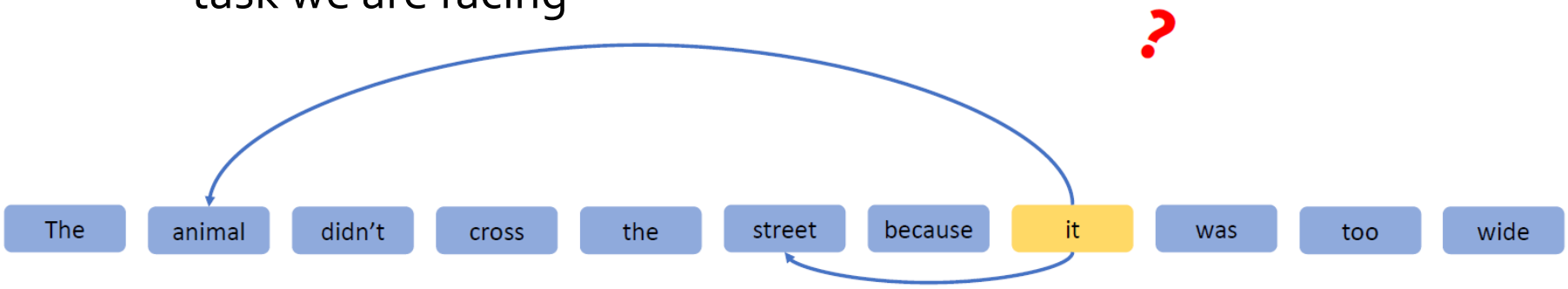
- The encoder is made of a sequence of **encoder blocks** having the same structure
  - The original paper used 6 encoder blocks
- Each encoder block processes a sequence using a combination of the following mechanisms
  - Self-attention: a (multi-headed) attention module where the same vectors are used as Q, K and V
  - A classical feed-forward layer applied separately to each element of the sequence
  - Skip connections
  - Normalization



# Self attention

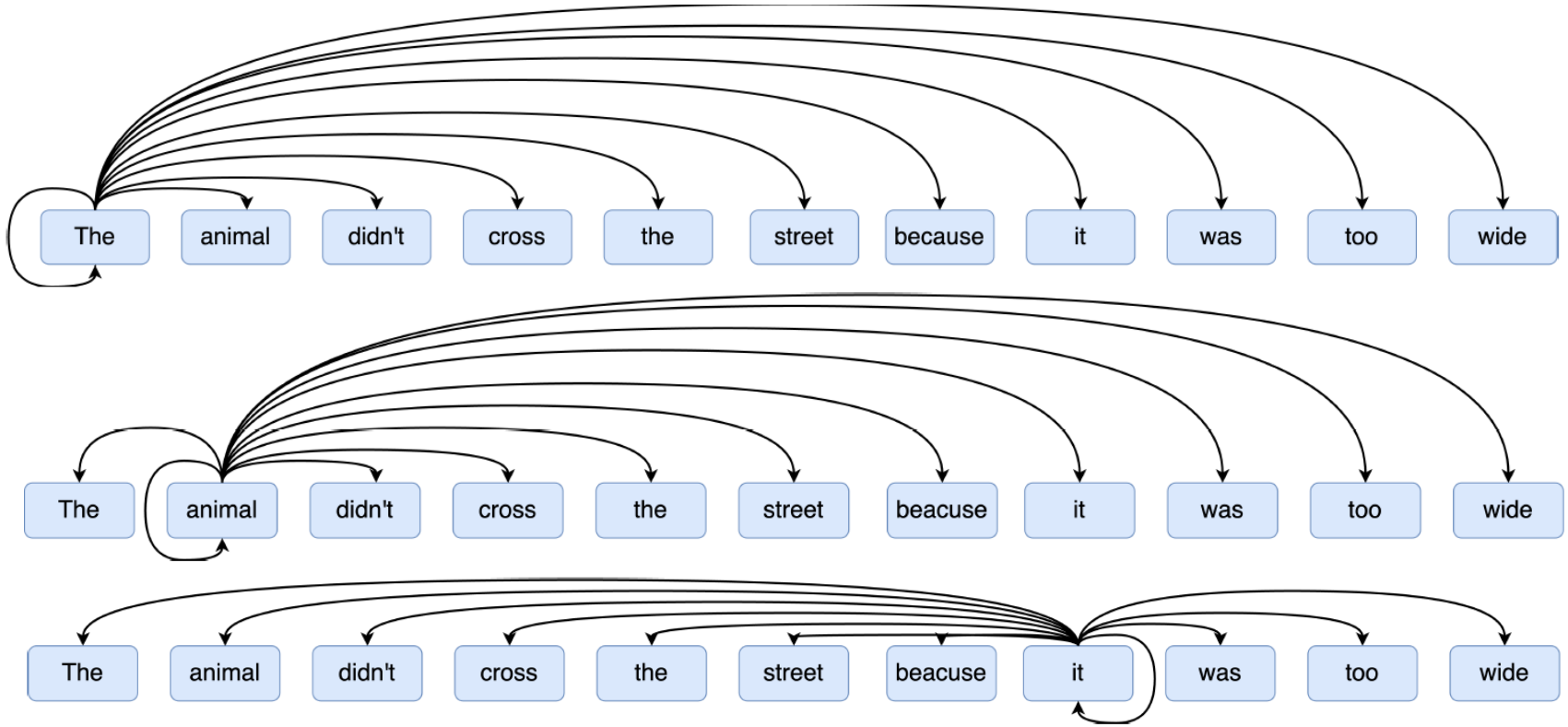
# Self Attention

- Let us consider the sentence:
  - The animal didn't cross the street because it was too wide
- What does it in this sentence refer to?
- Estimating self-attention in this sentence means to find the words that one must consider first to find a better encoding for the word it
- Self-Attention estimate must be learned according to the task we are facing



# Self Attention

- How to compute the attention to give to each input element when encoding the current word?



# Attention

- In order to understand the self attention, we must first introduce its fundamental building block: the attention function
- Informally, an attention function is used when the value to be computed (in this case the embedding of a token in a certain position considering the context of the sentence) depends on a set of other values (in this case other tokens of the sentence), and we want to give each time a different weight (i.e. a different "level of attention") to each of the values (how much each token is important to encode the current token?)
- The attention function depends on three elements, with a terminology inherited from document retrieval: query, key, value

# Attention

- We have an input value $q$ and we want to define some target function $f_T(q)$
- $q$ is called the **query** value in the attention terminology
- In the general case, both $q$ and $f_T(q)$ can be **vectors**
- We want to express $f_T(q)$ as a function of a given set of elements $v_1, \ldots, v_n$
- We want the "attention" given to each $v_i$ to be different depending on $q$
- We assume that for each $v_i$ we have available an additional information $k_i$ that can be used to decide the "attention" to be given to $v_i$
- The elements $v_i$ are called **values** and the $k_i$ are called **keys** in the attention terminology; both the values and the keys can be vectors

# Attention

- A commonly adopted formulation of the problem is to define the target function as:

$$f_T(q) = \alpha(q, k_1) \cdot f_V(v_1) + \cdots + \alpha(q, k_n) \cdot f_V(v_n) =$$

$$= \sum_{i=1}^{n} \alpha(q, k_i) \cdot f_V(v_i)$$

**Attention given to value $v_i$**

# Attention

- A commonly adopted formulation of the problem is to define the target function as:

$$f_T(q) = \sum_{i=1}^{n} \alpha(q, k_i) \cdot f_V(v_i)$$

- $\alpha$ is our **attention function**
  - We want $\alpha$ and $f_V$ to be **learned** by our system
  - Tipically, $\alpha(q, k_i) \in [0,1]$ and $\sum_i \alpha(q, k_i) = 1$
- **Note**: the value of the target function **does not depend on the order** of the key-value pairs $(k_i, v_i)$

# Self Attention

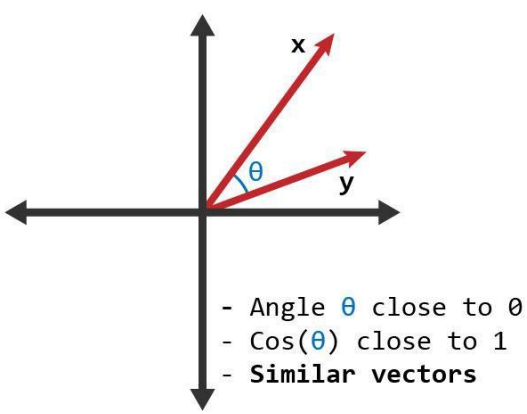
- The Transformer architecture uses a particular definition of the attention function, based on linear vector/matrix operations and the softmax function
- This definition is
  - Differentiable, so it can be learned using Back Propagation
  - Efficient to compute
  - Easy to parallelize, since the attention for several query vectors can be efficiently computed in parallel at the same time

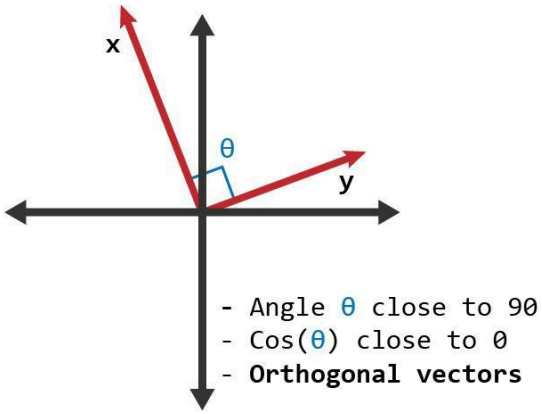
# Self Attention

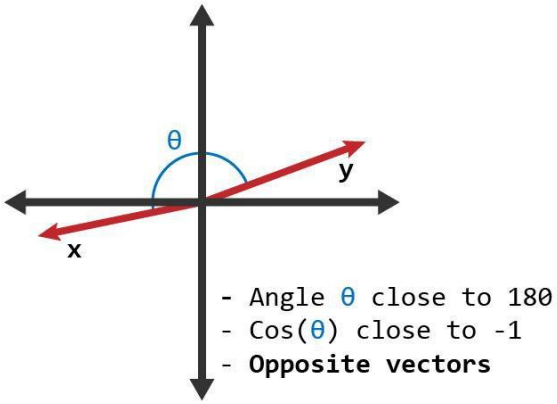
- Input: three matrices Q, K, V
- Q ($m \times d_q$) contains the query vectors (each row is a query)
- K ($n \times d_k$) contains the key vectors (each row is a key)
- V ($n \times d_v$) contains the value vectors (each row is a value)
  - K and V must have the same number of rows

# Self Attention

- If the query and the key are represented by vectors with the same dimensionality, a matching score can be provided by the scaled dot product of the two vectors (cosine similarity)

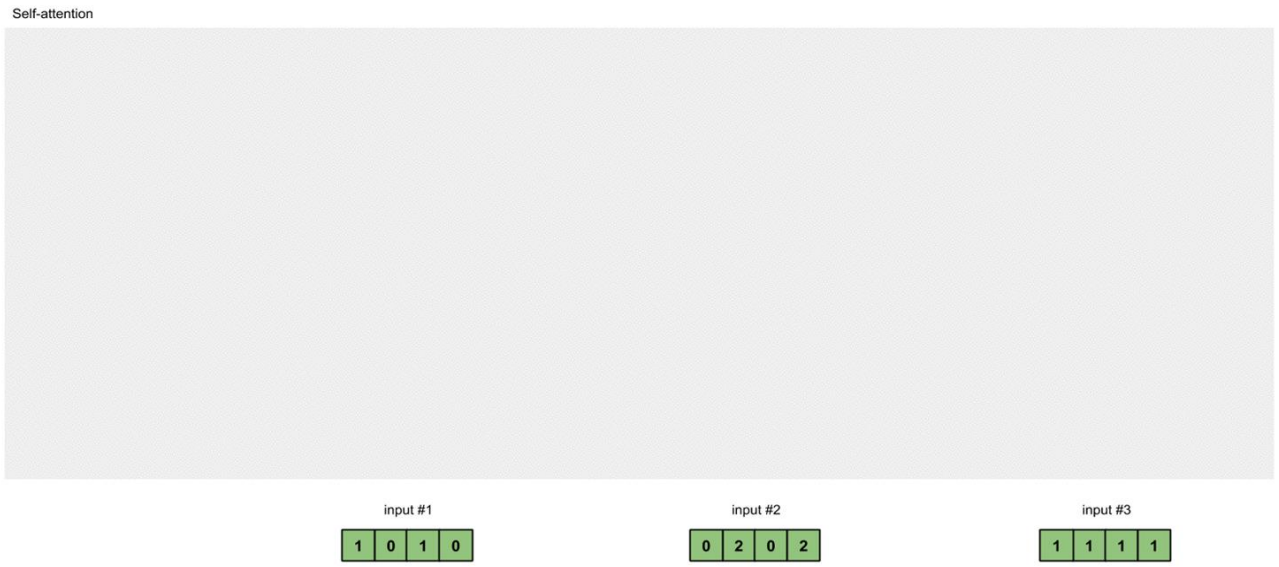


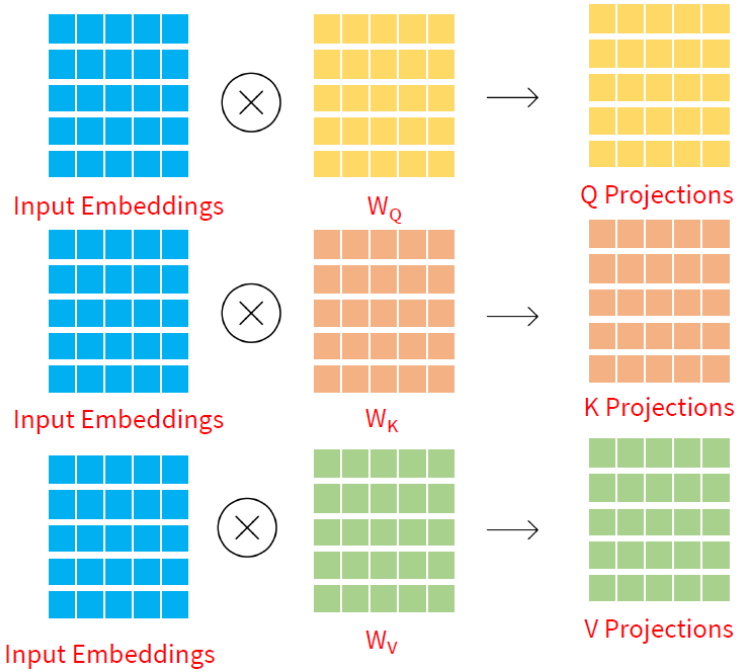
# Self Attention

- Step 0: Each element in the sequence is represented by a numerical vector



# Self Attention

- Step 1: the input matrices are "projected" onto a different subspace, by multiplying them (using row-by-column dot product) by weight matrices




- $Q' = Q \cdot W_Q$
- $K' = K \cdot W_K$
- $V' = V \cdot W_V$

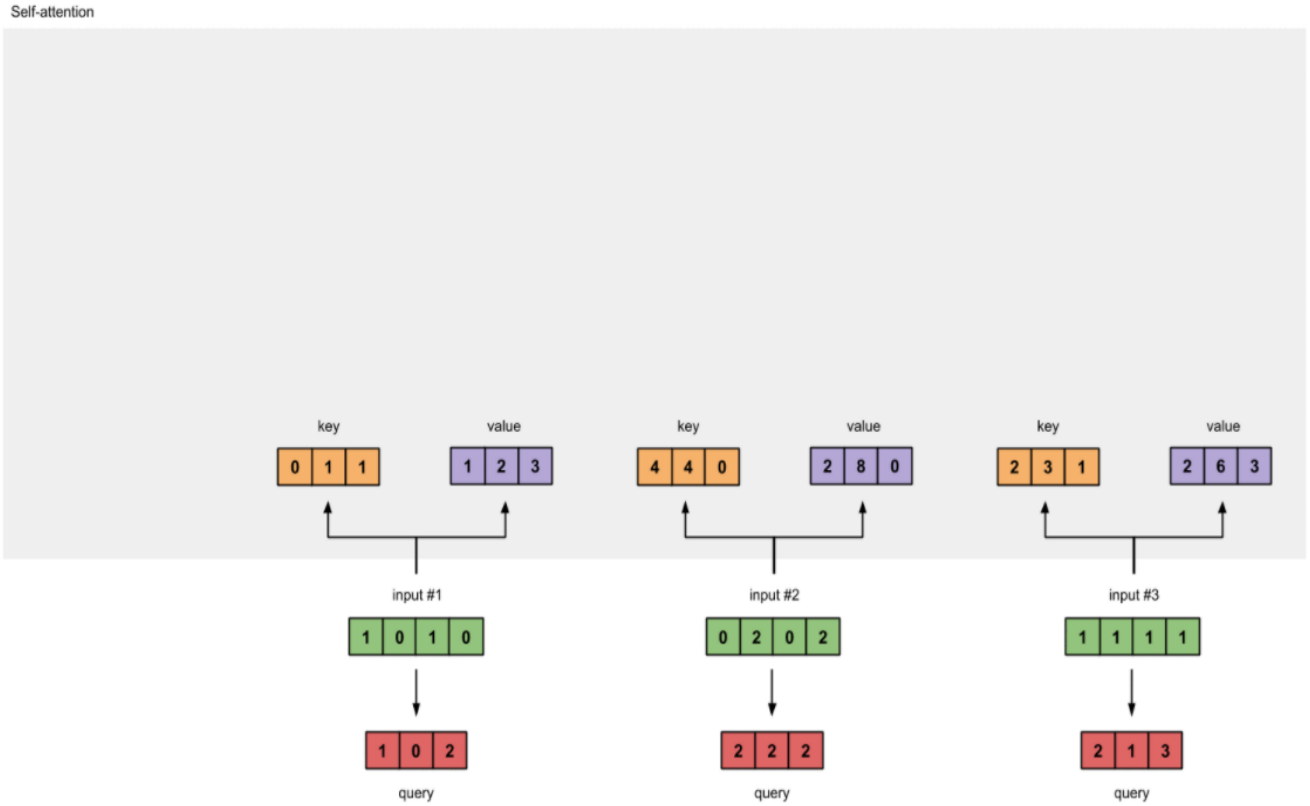
**Note**: $W_Q$ and $W_K$ **must** have the same number of columns

These are the trainable weights:

- $W_Q$ ($d_q$ x $d'_q$)
- $W_K$ ($d_k$ x $d'_k$)
- $W_V$ ($d_v$ x $d'_v$)

# Self Attention

- Step 1: Compute a key (K), a value (V) and a query (Q) as linear function of each element in the sequence.



# Self Attention

- Step 2: the attention matrix A is computed for each position by multiplying Q' and the transpose of K', scaling by $1/\sqrt{d'_k}$ and applying softmax to each of the resulting rows

- $A = softmax\left(\frac{Q' \cdot K'^T}{\sqrt{d'_k}}\right)$

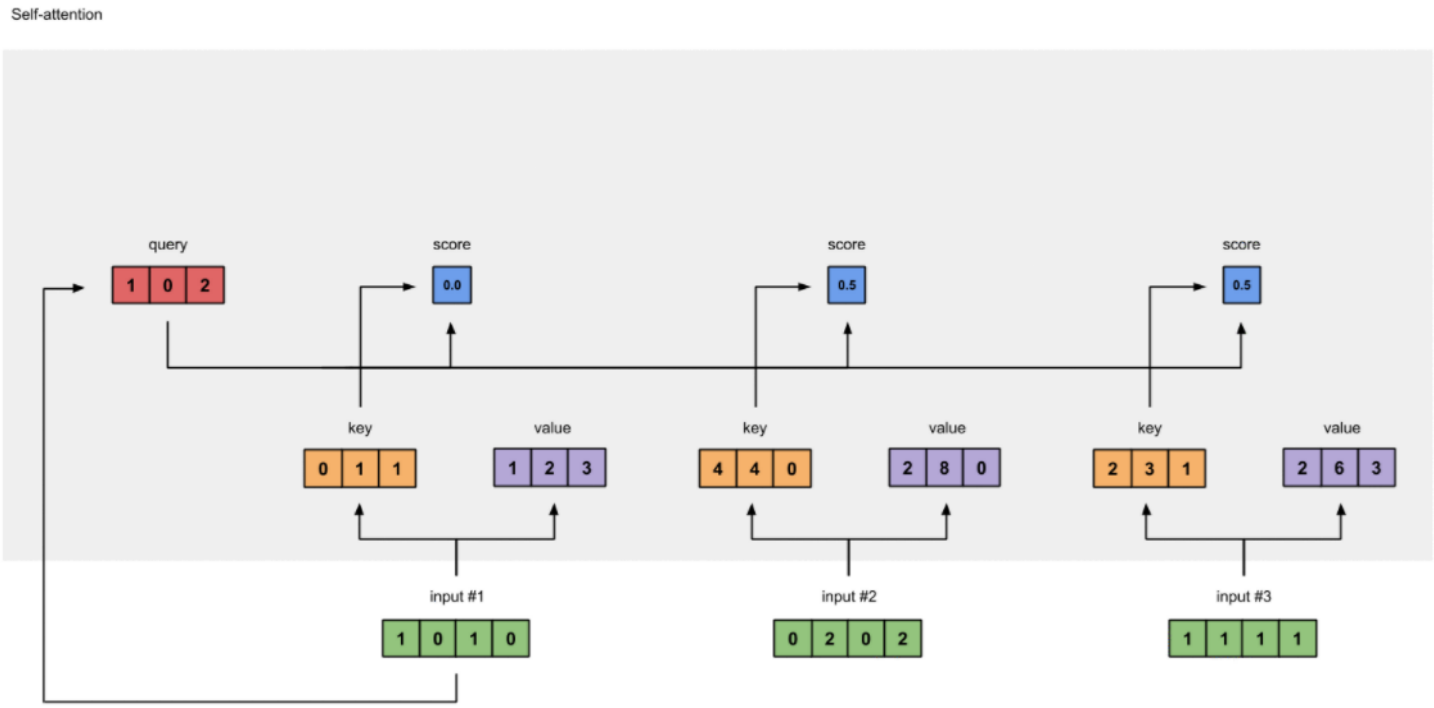
This scaling is used to avoid that the argument of softmax becomes too large with the increase of the dimension $d'_k$

Softmax is applied to each row separately

A is a (m x n) matrix whose element $a_{ij} = \alpha(q_i, k_j)$

# Self Attention

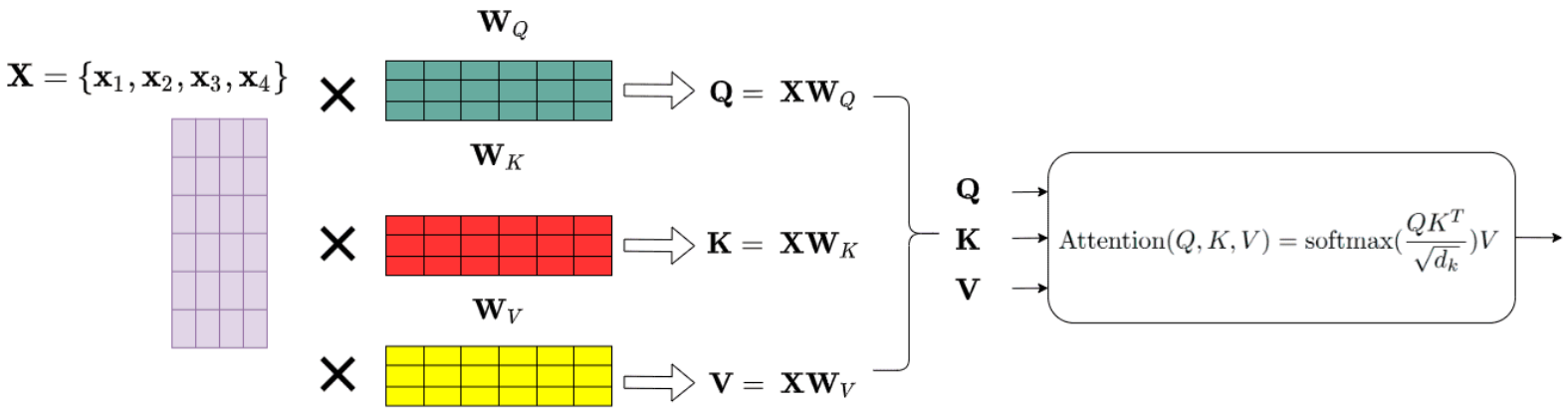
- Step 2: Compute attention score for each position i as a softmax of the scaled dot product of all the keys (*bidirectional self-attention*) with Q*i*



# Self Attention

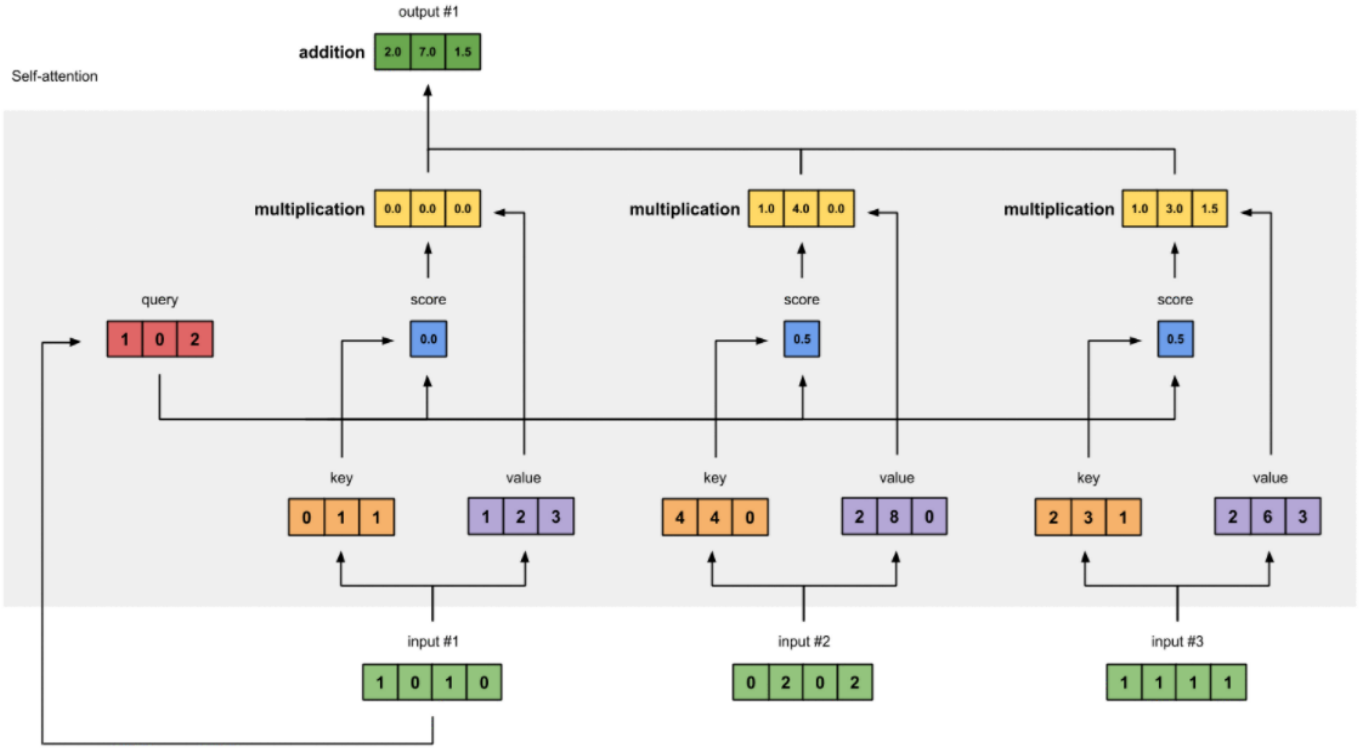
- Final step: the target value is computed by row-by-column multiplication between A and V'
  - $f_T(Q) = A \cdot V'$

The result is a m x d'$_v$ matrix representing the target function computed on the m queries in the input matrix Q.

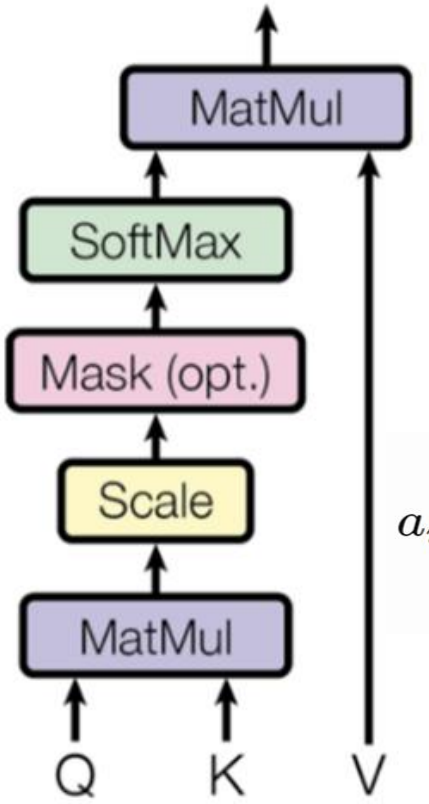


# Self Attention

- Step 3: Output representation for each position I, as a weighted sum of values (each one multiplied by the related attention score)



# Self Attention

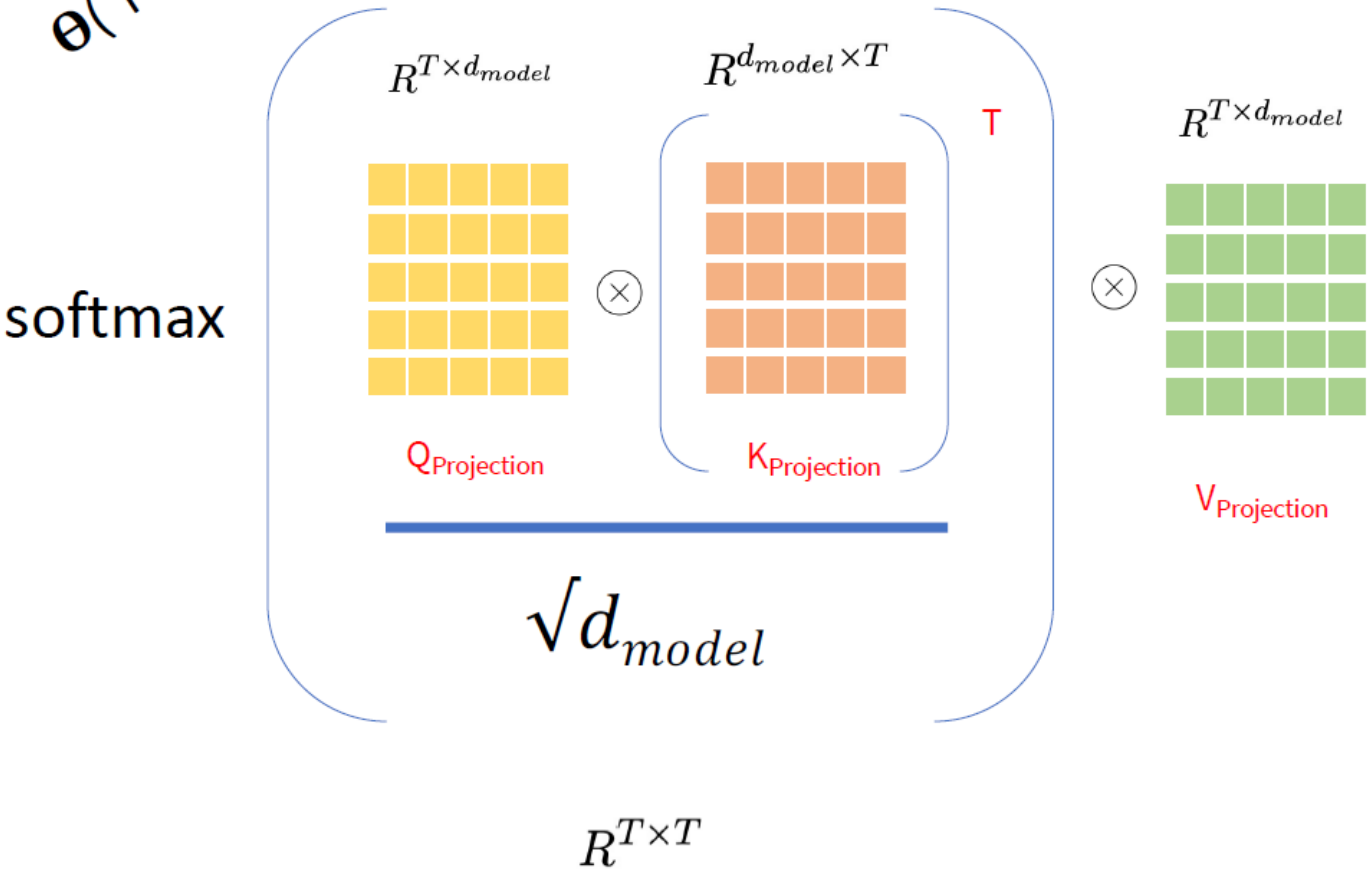



$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}})\mathbf{V}$$

$$a_{ij} = \text{softmax}(\frac{\mathbf{q}_i {\mathbf{k}_j}^\top}{\sqrt{d_k}}) = \frac{\exp(\mathbf{q}_i {\mathbf{k}_j}^\top)}{\sqrt{d_k} \sum_{r \in S_i} \exp(\mathbf{q}_i {\mathbf{k}_r}^\top)}$$

# Self Attention

$\Theta(T^2 \times d_{model})$



# Natural Language Processing and Large Language Models

Corso di Laurea Magistrale in Ingegneria Informatica


Lesson 9

# Transformers I

**Nicola Capuano and Antonio Greco**

**DIEM – University of Salerno**