

بینایی کامپیوتر
فصل چهارم
ترنسفورمرها

Transformers

محمد جواد فدائی اسلام

فروردین ۱۴۰۴

Deep Learning Elements

- Two major architectural elements in deep learning:
 - Convolutional Layers
 - Recurrence Layers
- But now we have another element: Attention Layers.
- It was first revealed in the paper "Attention is All You Need" by Vaswani et al. (2017)
- In attention networks, the Query, Key, and Value vectors is created **NOT** by convolution, **NOT** by recurrence, but by matrix multiplication.
- Transformers handle long-range dependencies efficiently.
- Transformers enable parallel training on large datasets.

Transformer

- Transformers handle long-range dependencies efficiently.
- Transformers enable parallel training on large datasets
- Popular transformers
 - BERT → understanding tasks
 - GPT → text generation
 - T5 → text-to-text learning



Attention

”The animal didn't cross the street because it was too tired.”

- What does “it” in this sentence refer to?
- Is it referring to the street or to the animal?
- It’s a simple question to a human, but not as simple to an algorithm.
- When the model is processing the word “it”, self-attention allows it to associate “it” with “animal”.

Two forms of attention

○ Self-attention

A word looks at other words in the **same sentence** to understand context.

Example: “The **animal** didn’t cross the street because **it** was tired.”

○ Cross-attention

It discovers what parts of a sentence in the source language are relevant to the production of each word in the target language.

The **animal** didn’t cross the street because **it** was tired



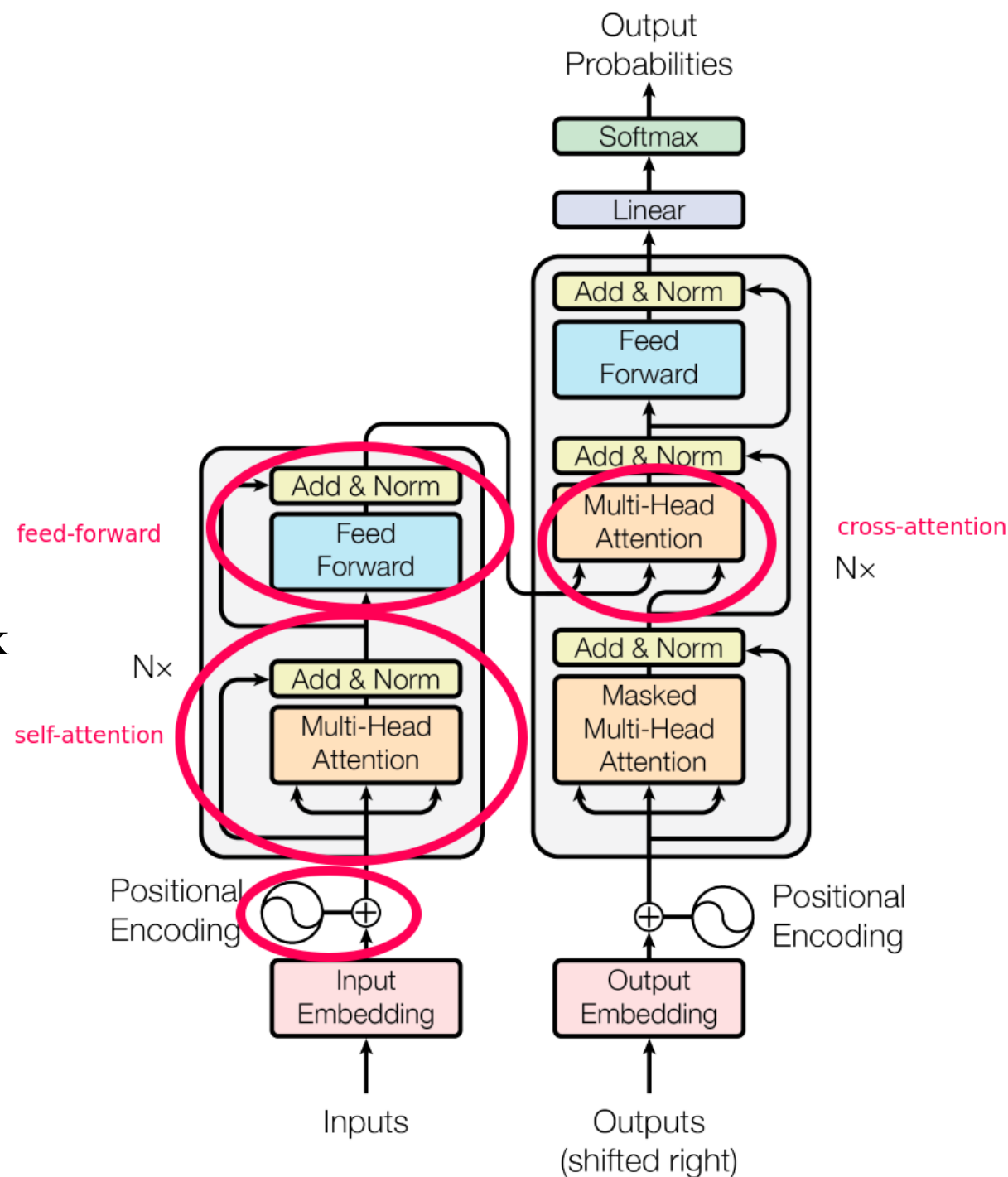
حيوان به دليل خستگي از خيابان عبور نکرد.

Self-attention vs. Cross-attention

Feature	Self-Attention	Cross-Attention
Scope	Single sequence	Two sequences (e.g., encoder-decoder)
Usage	Extracting relationships within the same sequence	Aligning relationships across sequences
Applications	Text classification, language modeling	Machine translation, summarization
Example	BERT	GPT

Transformer Architecture

- Input Embedding
- Positional Encoding
- Multi-Head Attention
- Feed Forward Network
- Output Layer



Attention

Queries, Keys, and Values

- In the simplest form of a database, It is a collection of keys (k) and values (v).
- For instance, our database D might consist of tuples
 {("Zhang", "Aston"),
 ("Lipton", "Zachary"),
 ("Li", "Mu"),
 ("Smola", "Alex"),
 ("Hu", "Rachel"),
 ("Werness", "Brent")}
 with the **last name** being the key and the **first name** being the value.
- With the exact query q ="Li", It returns the value "Mu".
- If ("Li", "Mu") is not a record in D , there will be no valid answer.
 for approximate matches, it retrieves "Zachary".
- This leads to one of the most exciting concepts: *the attention mechanism*.

Attention Pooling

- Inputs:

$D \stackrel{\text{def}}{=} \{(\mathbf{k}_1, \mathbf{v}_1), \dots, (\mathbf{k}_m, \mathbf{v}_m)\}$: a database of m tuples of *keys* and *values*.

$\mathbf{q}$: a query

- Operation: attention over D

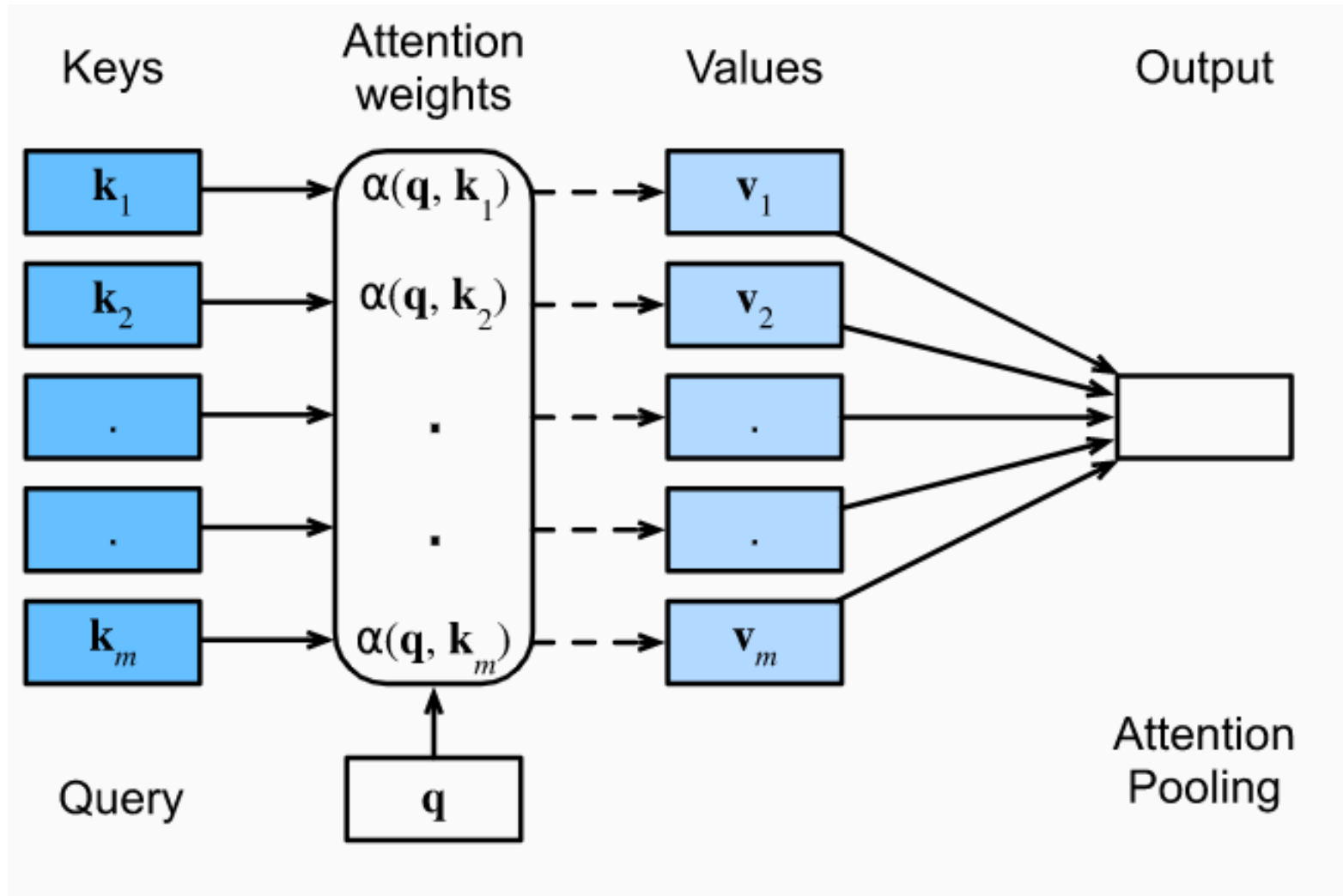
$$\text{Attention}(\mathbf{q}, D) \stackrel{\text{def}}{=} \sum_{i=1}^m \alpha(\mathbf{q}, \mathbf{k}_i) \mathbf{v}_i$$

$\alpha(\mathbf{q}, \mathbf{k}_i) \in R$, scalar attention weight

- Outputs: It generates a linear combination of values contained in the database.

- Attention* derives from the fact that the operation pays particular attention to the terms for which the weight α is significant.

Attention Pooling



Scaled Dot-Product Attention

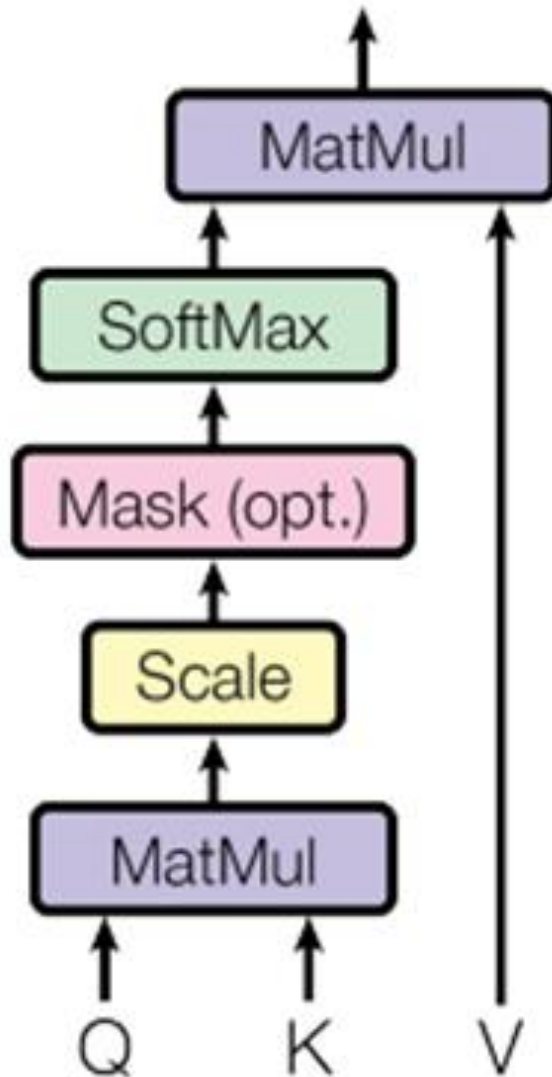
- query $\mathbf{q} \in R^d$ and the key $\mathbf{k}_i \in R^d$ are random variables with zero mean and unit variance.
- The dot product between both vectors has zero mean and a variance of d . To ensure that the variance of the dot product remains one, we rescale the dot-product by $1/\sqrt{d}$:

$$a(\mathbf{q}, \mathbf{k}_i) = \mathbf{q}^T \mathbf{k}_i / \sqrt{d}$$

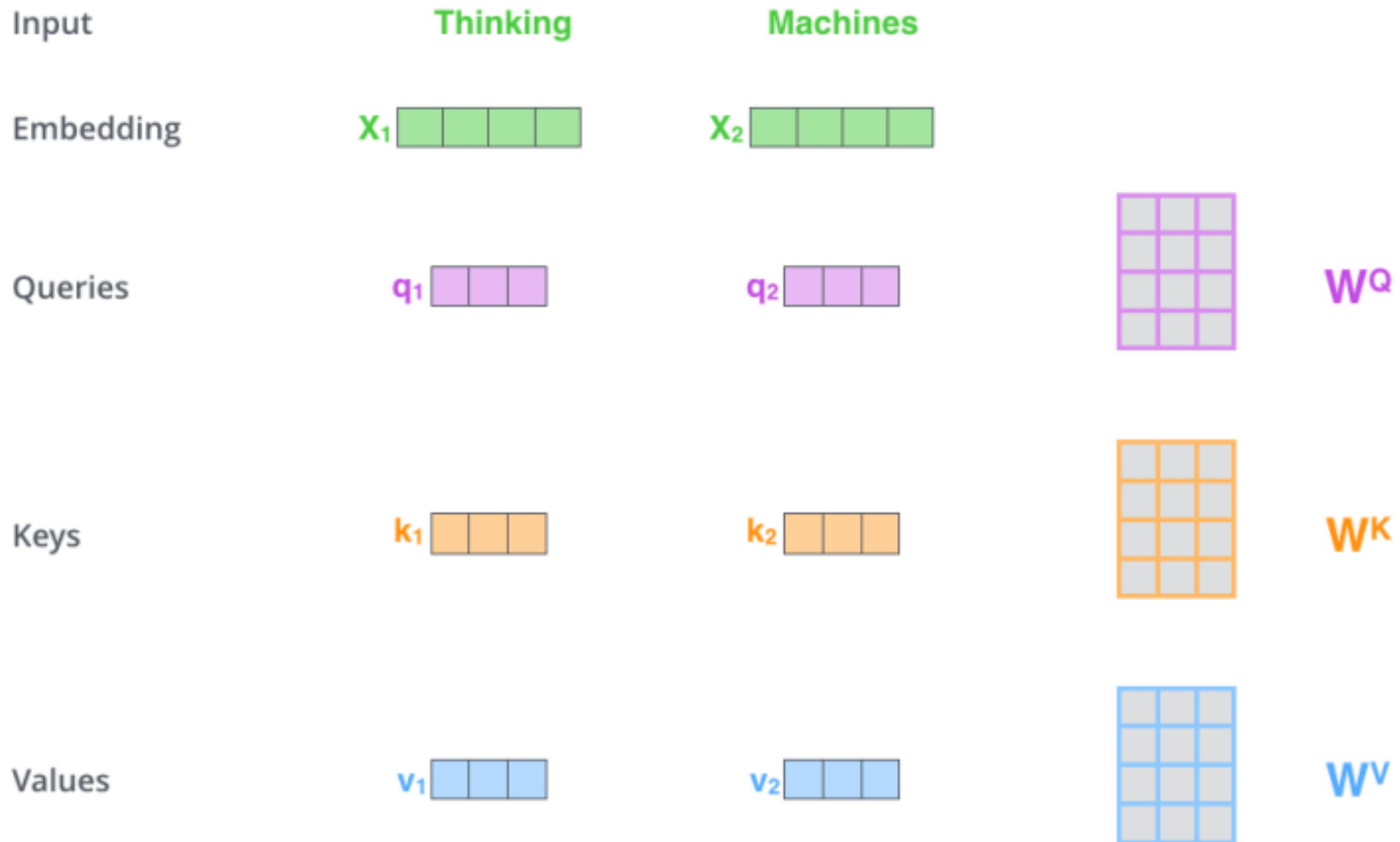
- Note that attention weights α still need normalizing. We can simplify this further by using the softmax operation:

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \text{softmax}(a(\mathbf{q}, \mathbf{k}_i)) = \frac{\exp(\frac{\mathbf{q}^T \mathbf{k}_i}{\sqrt{d}})}{\sum_j \exp(\frac{\mathbf{q}^T \mathbf{k}_j}{\sqrt{d}})}$$

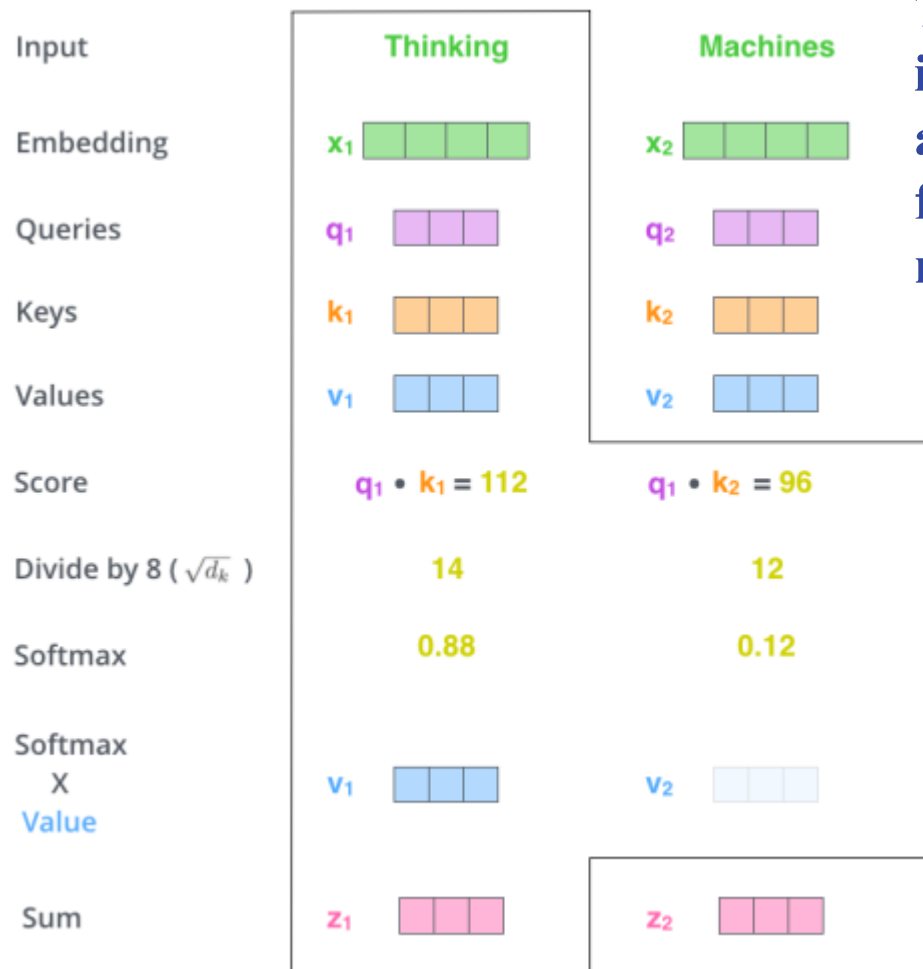
Scaled Dot-Product Attention



creating "query", "key", and "value" projection of each word in the input sentence



Self-attention Calculations



The resulting vector is one we can send along to the feed-forward neural network.

Masked Softmax Operation

- we need to be able to deal with sequences of **different lengths**.
- Since transformer models process sequences in parallel, all input sequences in a batch need to have the same length.
- To achieve this, shorter sequences are padded with special tokens called "**dummy tokens**".

Batch Matrix Multiplication

- We often think in mini-batches for efficiency, such as computing attention for n queries and m key-value pairs, where queries and keys are of length d and values are of length v . The scaled dot-product attention of queries $\mathbf{Q} \in R^{n \times d}$, keys $\mathbf{K} \in R^{m \times d}$, and values $\mathbf{V} \in R^{m \times v}$ thus can be written as

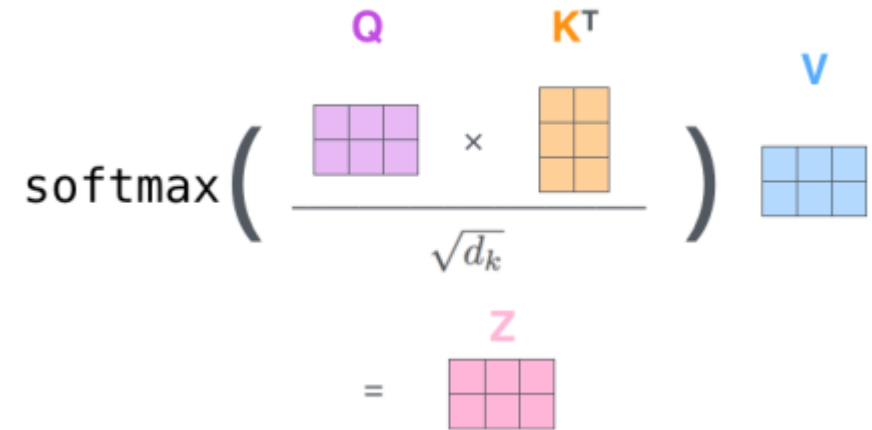
$$\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}}\right)\mathbf{V} \in R^{n \times v}$$

Matrix Calculation of Self-Attention

$$\mathbf{X} \times \mathbf{W}^Q = \mathbf{Q}$$


$$\mathbf{X} \times \mathbf{W}^K = \mathbf{K}$$


$$\mathbf{X} \times \mathbf{W}^V = \mathbf{V}$$


$$\text{softmax}\left(\frac{\mathbf{Q} \times \mathbf{K}^T}{\sqrt{d_k}}\right) \times \mathbf{V} = \mathbf{Z}$$


Matrix Calculation: More Explain

○ Input Tokens and Embeddings:

- The input tokens are first converted into dense numerical vectors using an embedding layer.

○ Linear Transformation:

- Each embedding vector is passed through a linear transformation using the query weight matrix:

$$Q_i = X_i W_q$$

- The transformation matches the model's hidden size. For instance, if the embedding vector has 512 dimensions and the model uses a hidden size of 768, the weight matrix W_q adjust it.

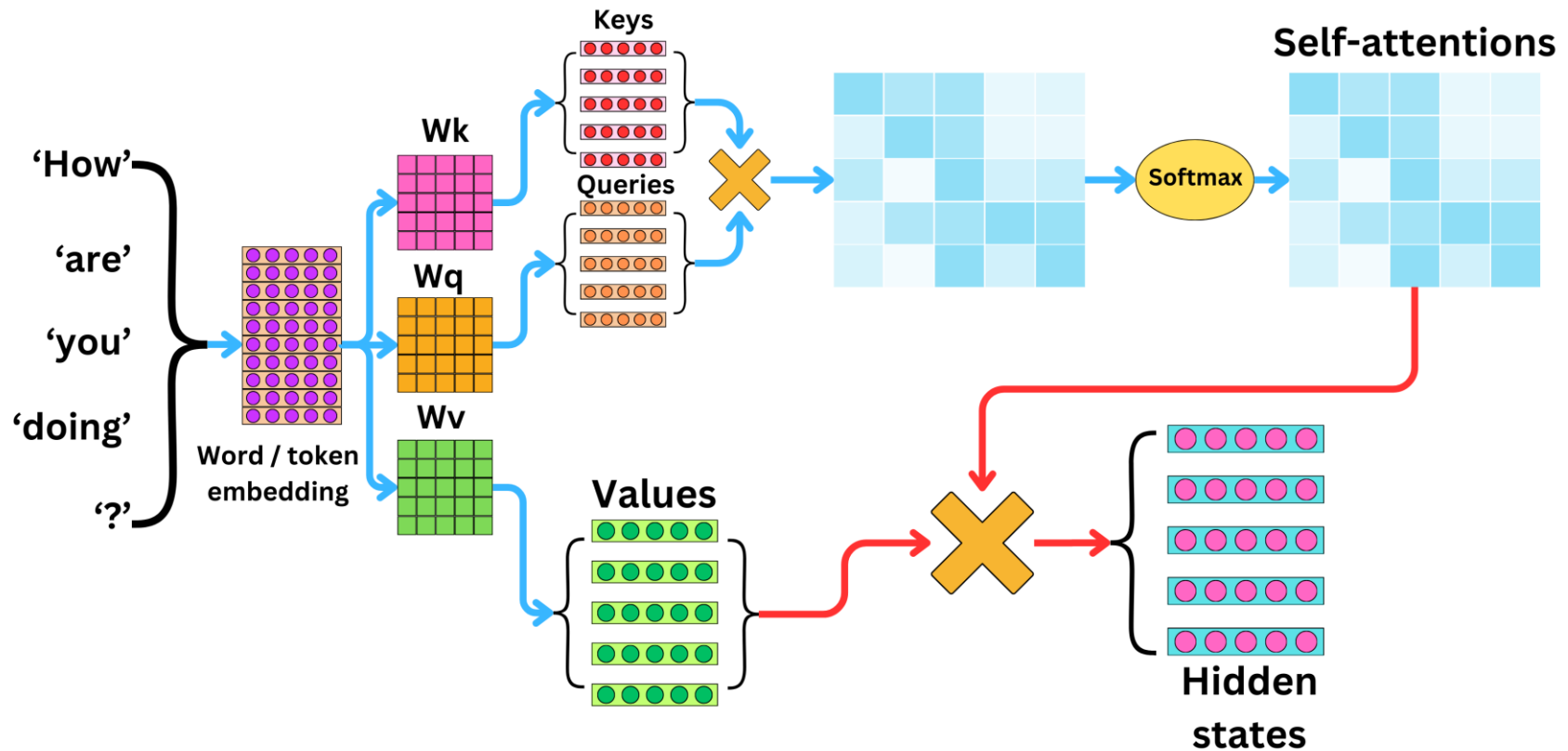
○ Attention Mechanism:

- The attention score between token Q_i and key K_j determines how much this token should focus on token K_j .

○ Softmax and Weighting:

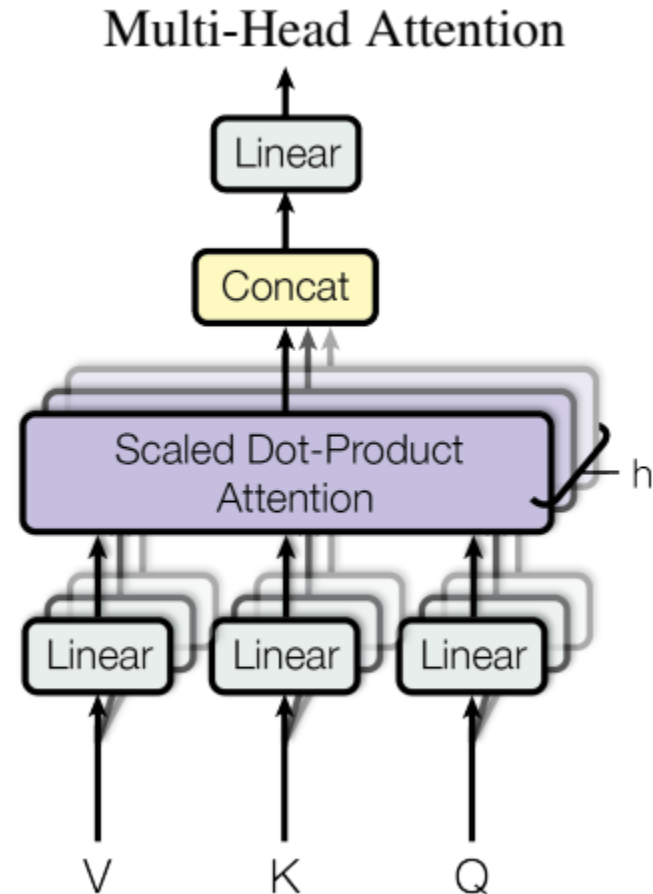
- The scores are passed through a softmax function to normalize them into probabilities and then are used to weight the corresponding values V_i .

Scaled Dot-Product Attention



Multi-Head Attention

- What I have described in the last is referred to as a **Single-Headed Attention**.
- Single-headed attention is not sufficiently rich in its representational power for capturing all the needed inter-word dependencies in a sentence.



Multi-Head Attention

Mathematical Representation

- Given a query $\mathbf{q} \in R^{d_q}$, a key $\mathbf{k} \in R^{d_k}$, and a value $\mathbf{v} \in R^{d_v}$, each attention head $h_i (i = 1, \dots, h)$ is computed as

$$\mathbf{h}_i = f(\mathbf{W}_i^{(q)} \mathbf{q}, \mathbf{W}_i^{(k)} \mathbf{k}, \mathbf{W}_i^{(v)} \mathbf{v}) \in R^{P_v},$$

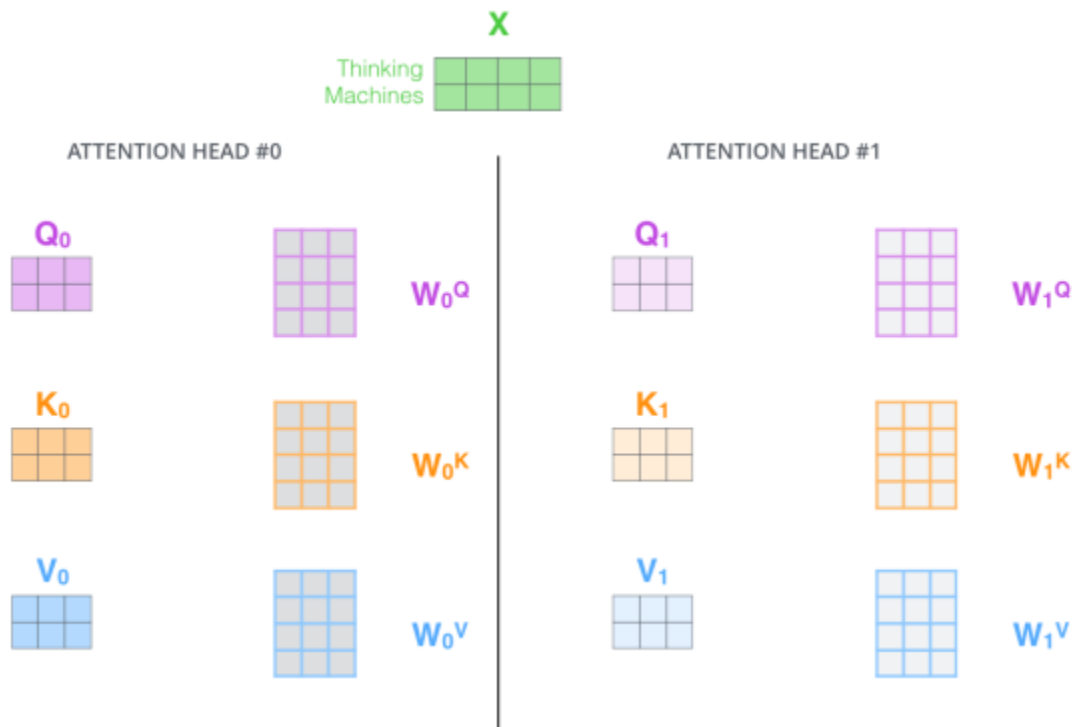
Where $\mathbf{W}$ s are learnable parameters and f is attention pooling.

- The multi-head attention output is another linear transformation via learnable parameters $\mathbf{W}_o \in R^{P_o \times h P_v}$ of the concatenation of h heads:

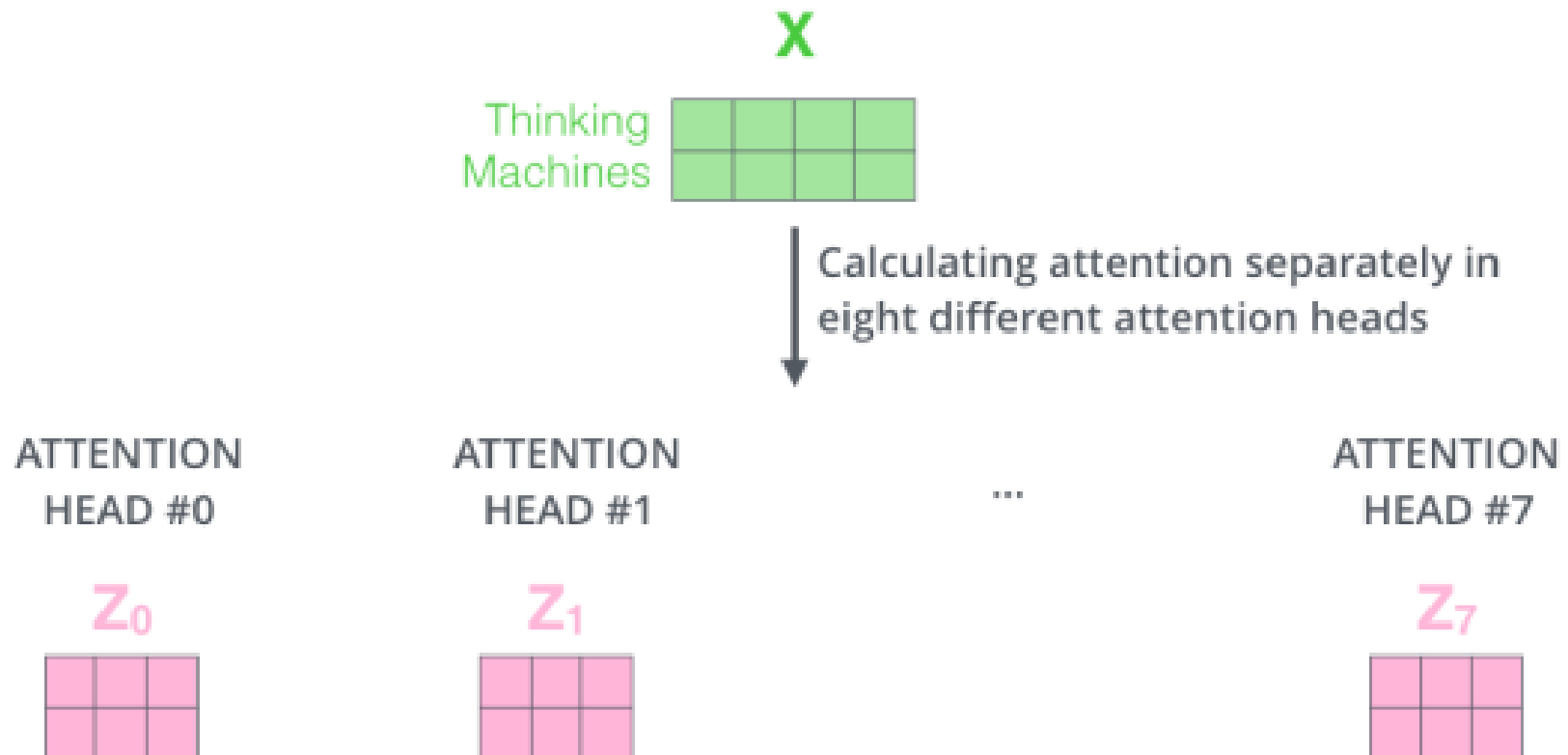
$$\mathbf{W}_o \begin{bmatrix} h_1 \\ \vdots \\ h_n \end{bmatrix} \in R^{P_o}.$$

- Based on this design, each head may attend to different parts of the input. More sophisticated functions than the simple weighted average can be expressed.

Multi-Head Attention

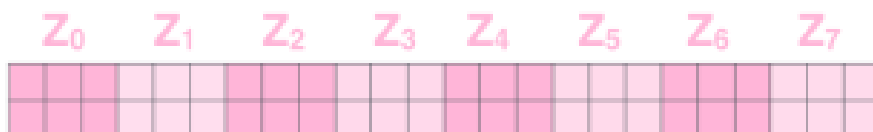


Multi-Head Attention ...



Multi-Head Attention

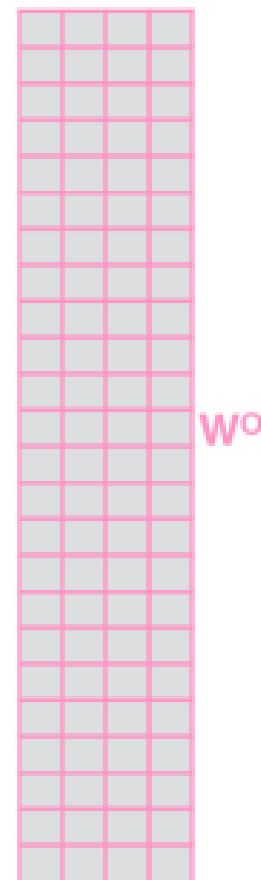
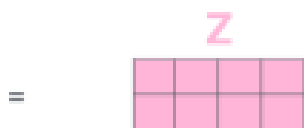
1) Concatenate all the attention heads



2) Multiply with a weight matrix W^O that was trained jointly with the model

$\times$

3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN



Multi-Head Attention

1) This is our input sentence*

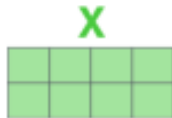
2) We embed each word*

3) Split into 8 heads. We multiply X or R with weight matrices

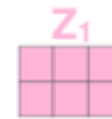
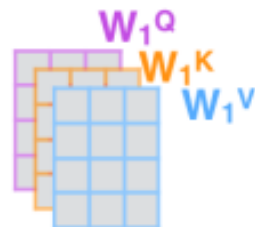
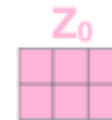
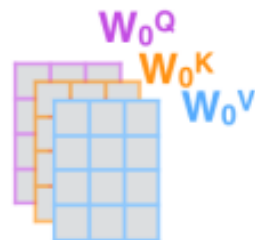
4) Calculate attention using the resulting $Q/K/V$ matrices

5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer

Thinking
Machines



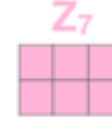
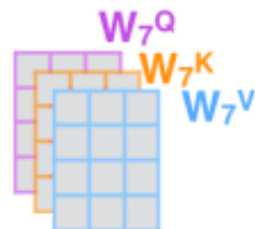
* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one



...

...

...



Positional Encoding

- RNNs recurrently process tokens of a sequence one by one; self-attention operates in parallel.
- **Note:** Self-attention by itself does not preserve the order of the sequence.
- **Positional encoding** is an approach for this purpose. It represents the order of tokens to the model as an additional input associated with each token.

Positional Encoding

- Input: $\mathbf{X} \in R^{n \times d}$ d -dimensional embeddings for n tokens of a sequence.

- Positional encoding: $\mathbf{P} \in R^{n \times d}$

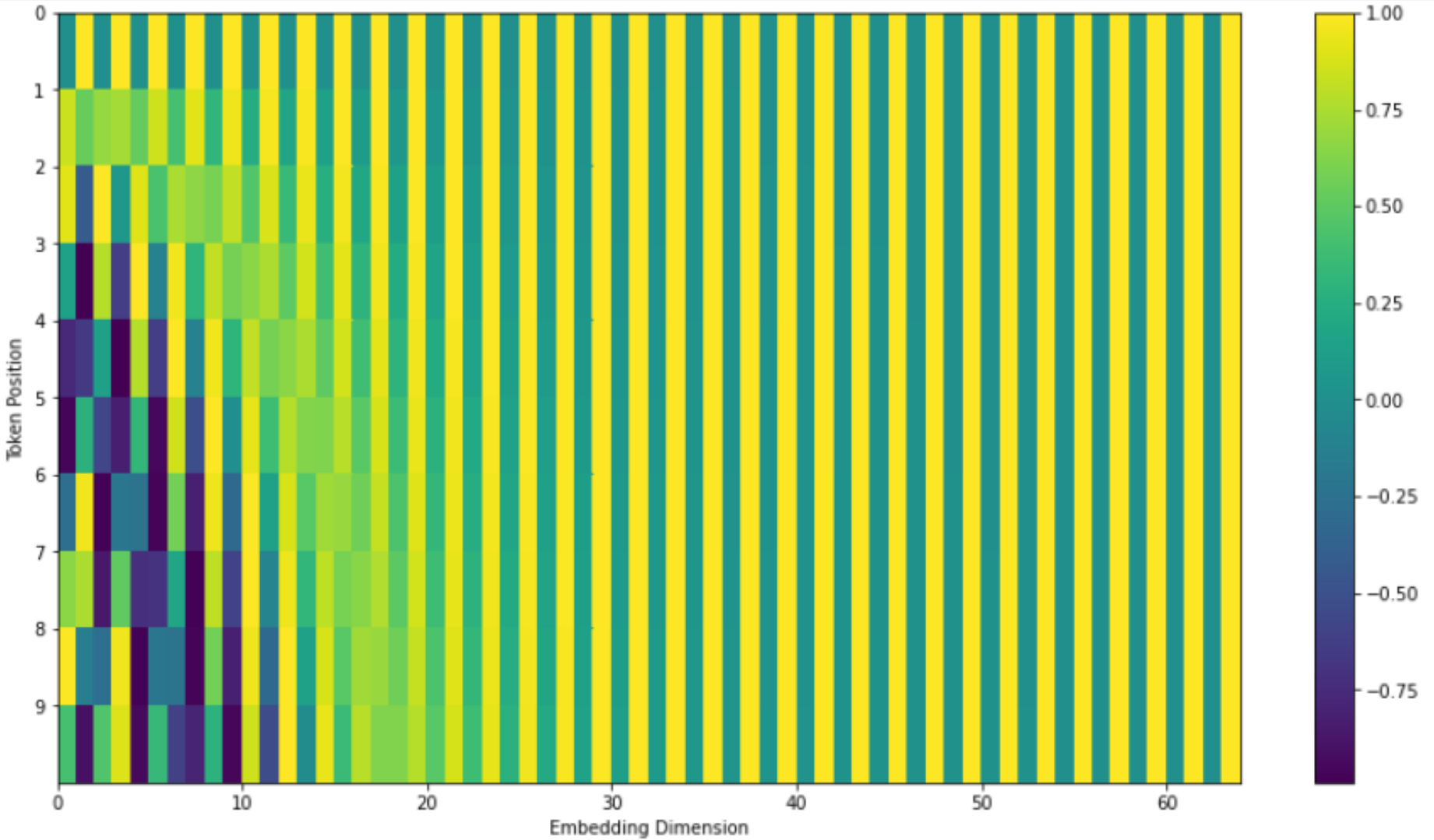
element on the i^{th} row and the $(2j)^{th}$ or the $(2j + 1)^{th}$ column is

$$P_{i,2j} = \sin\left(\frac{i}{10000^{2j/d}}\right)$$

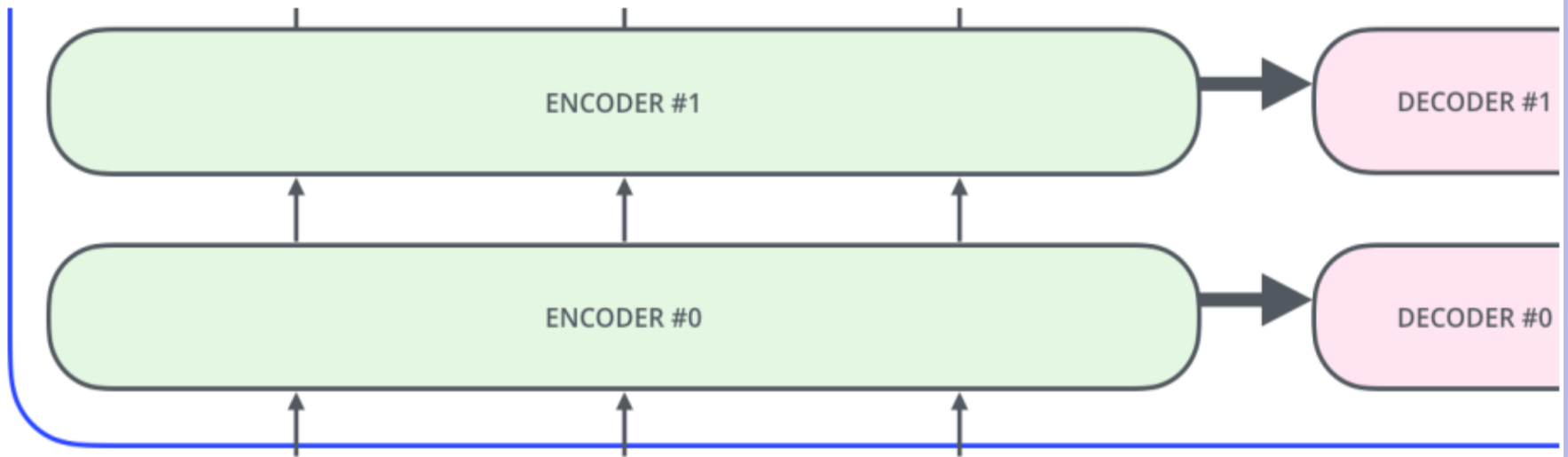
$$P_{i,2j+1} = \cos\left(\frac{i}{10000^{2j/d}}\right)$$

- Output: $\mathbf{X} + \mathbf{P}$

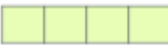
Positional Encoding

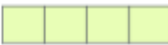


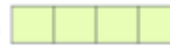
Positional Encoding



EMBEDDING
WITH TIME
SIGNAL

x_1 

x_2 

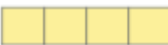
x_3 

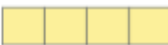
=

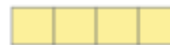
=

=

POSITIONAL
ENCODING

t_1 

t_2 

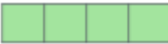
t_3 

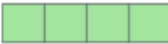
+

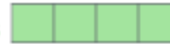
+

+

EMBEDDINGS

x_1 

x_2 

x_3 

INPUT

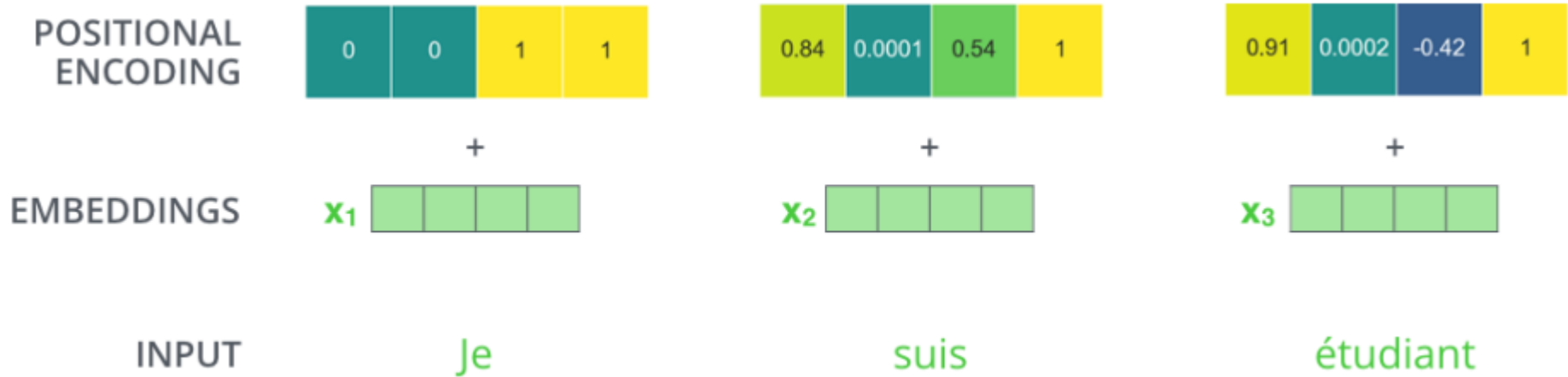
Je

suis

étudiant

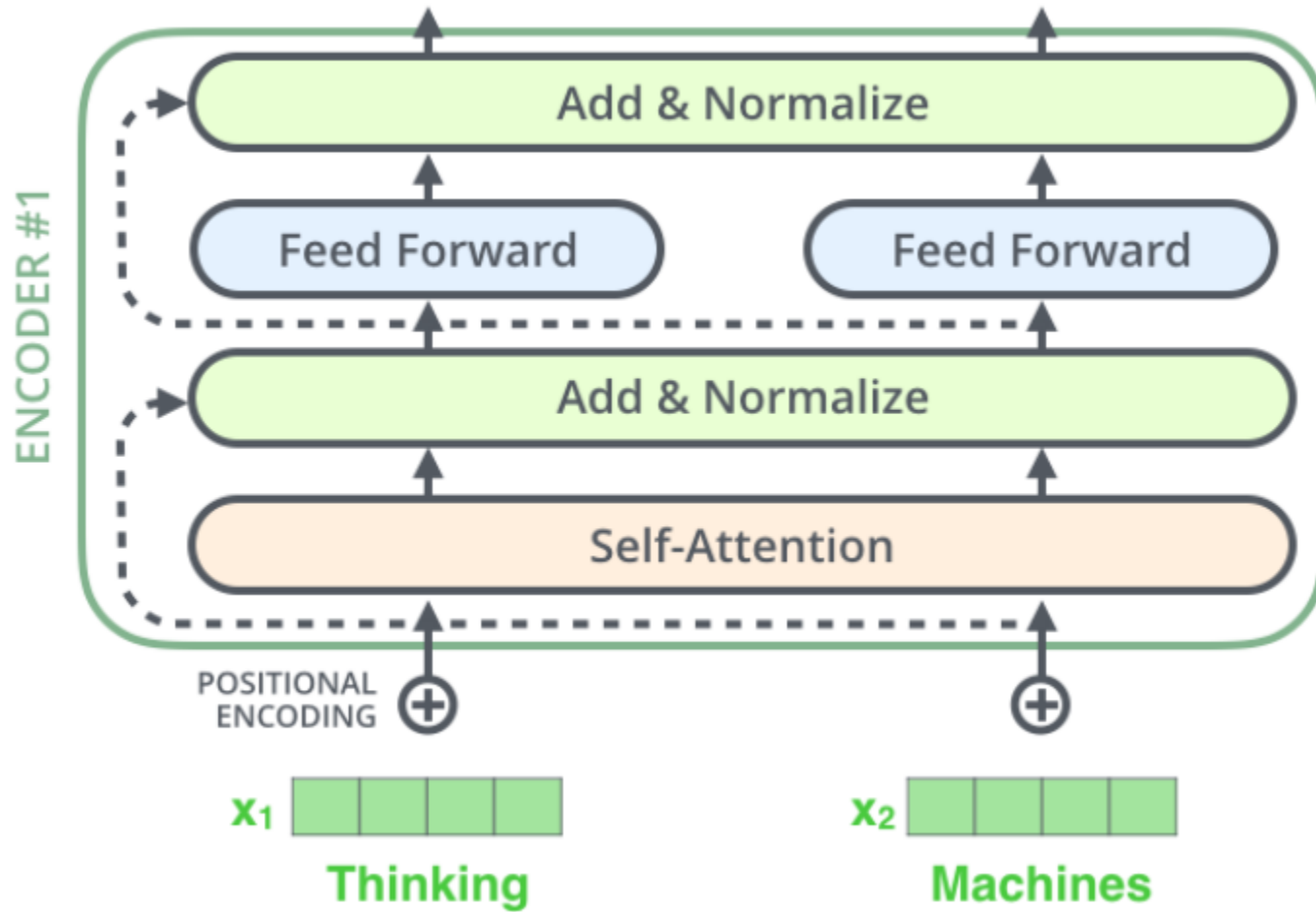
A real example of positional encoding with a toy embedding size of 4

Positional Encoding

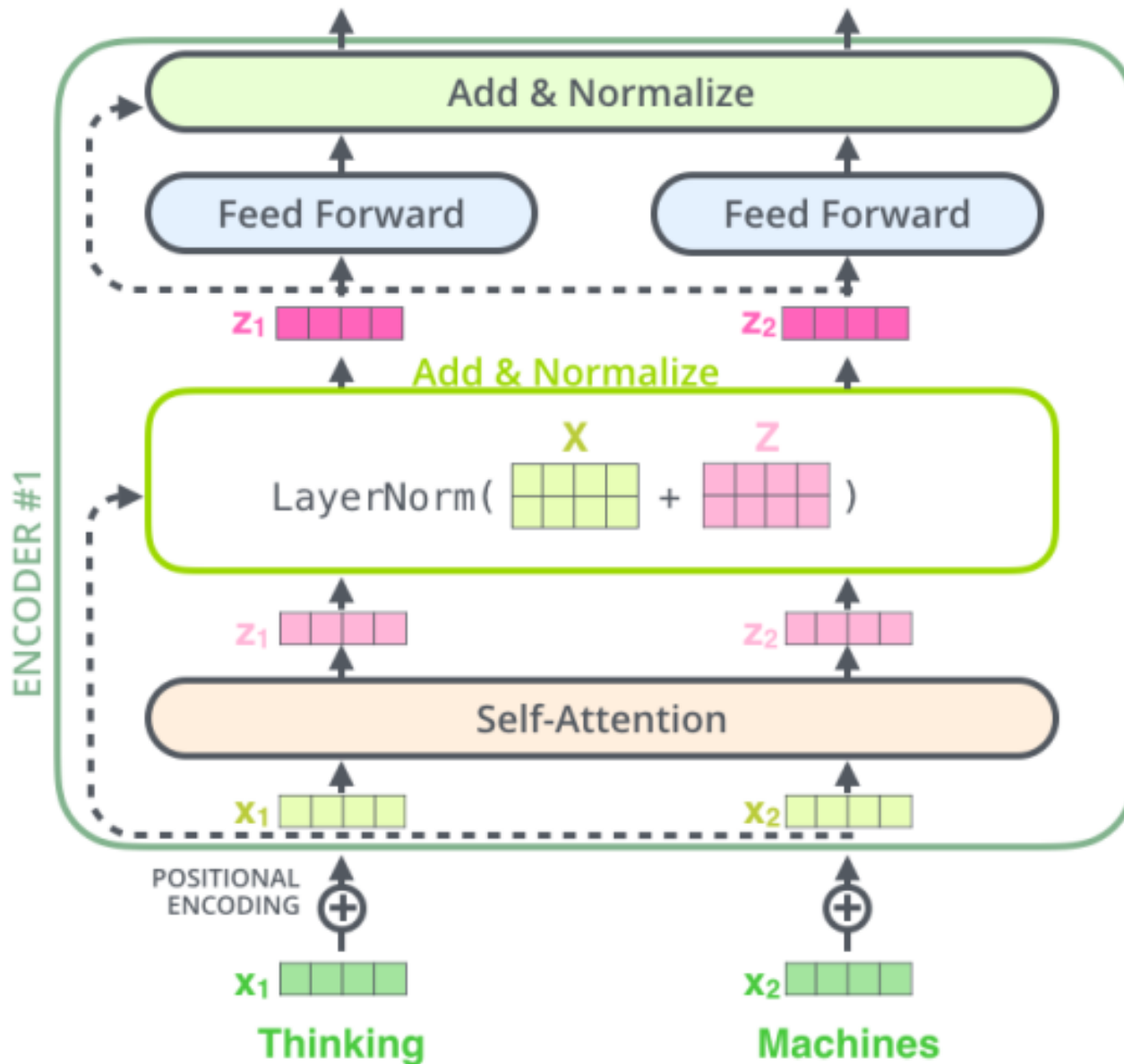


A real example of positional encoding with a toy embedding size of 4

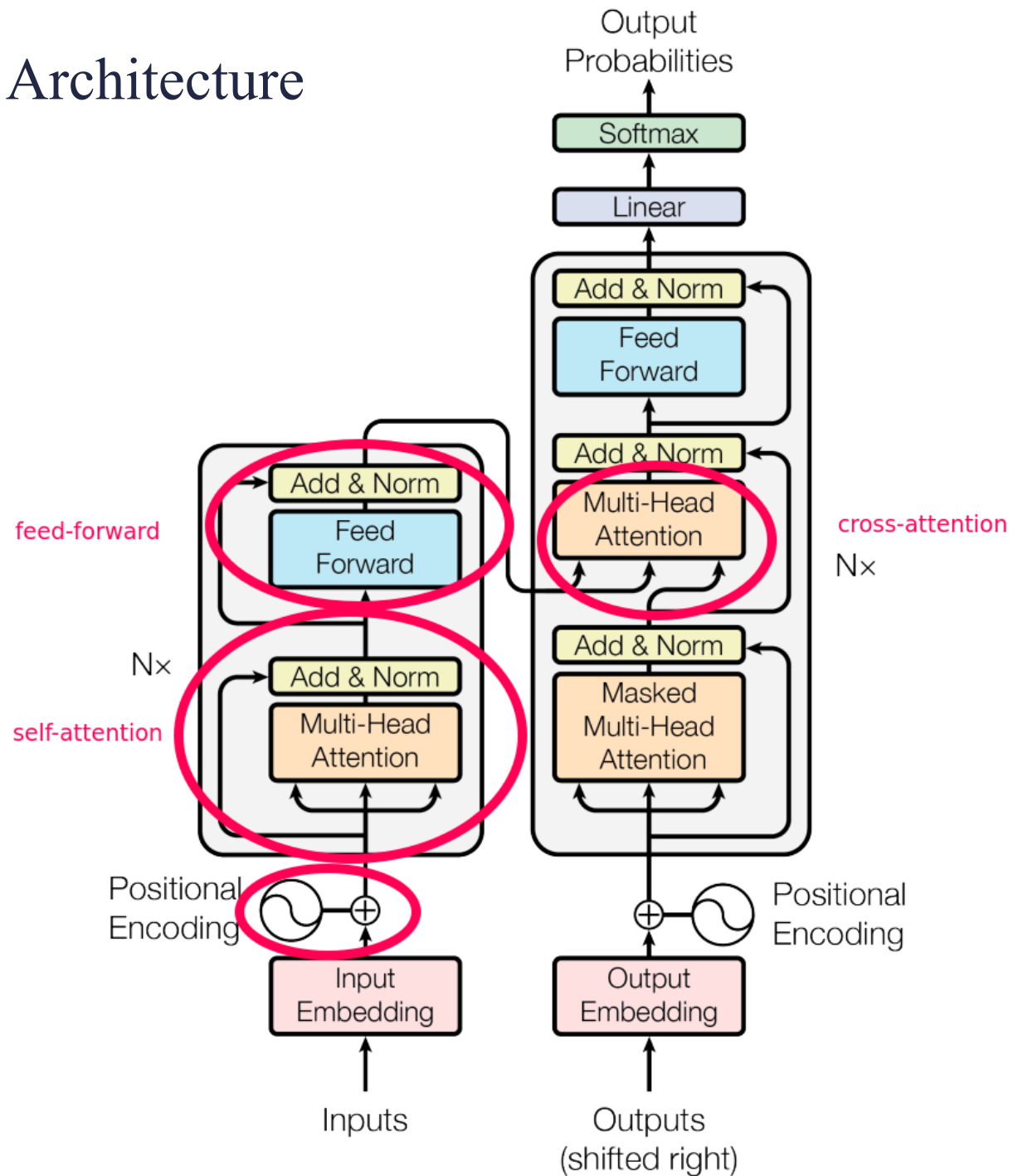
Encoder



Encoder – Residual



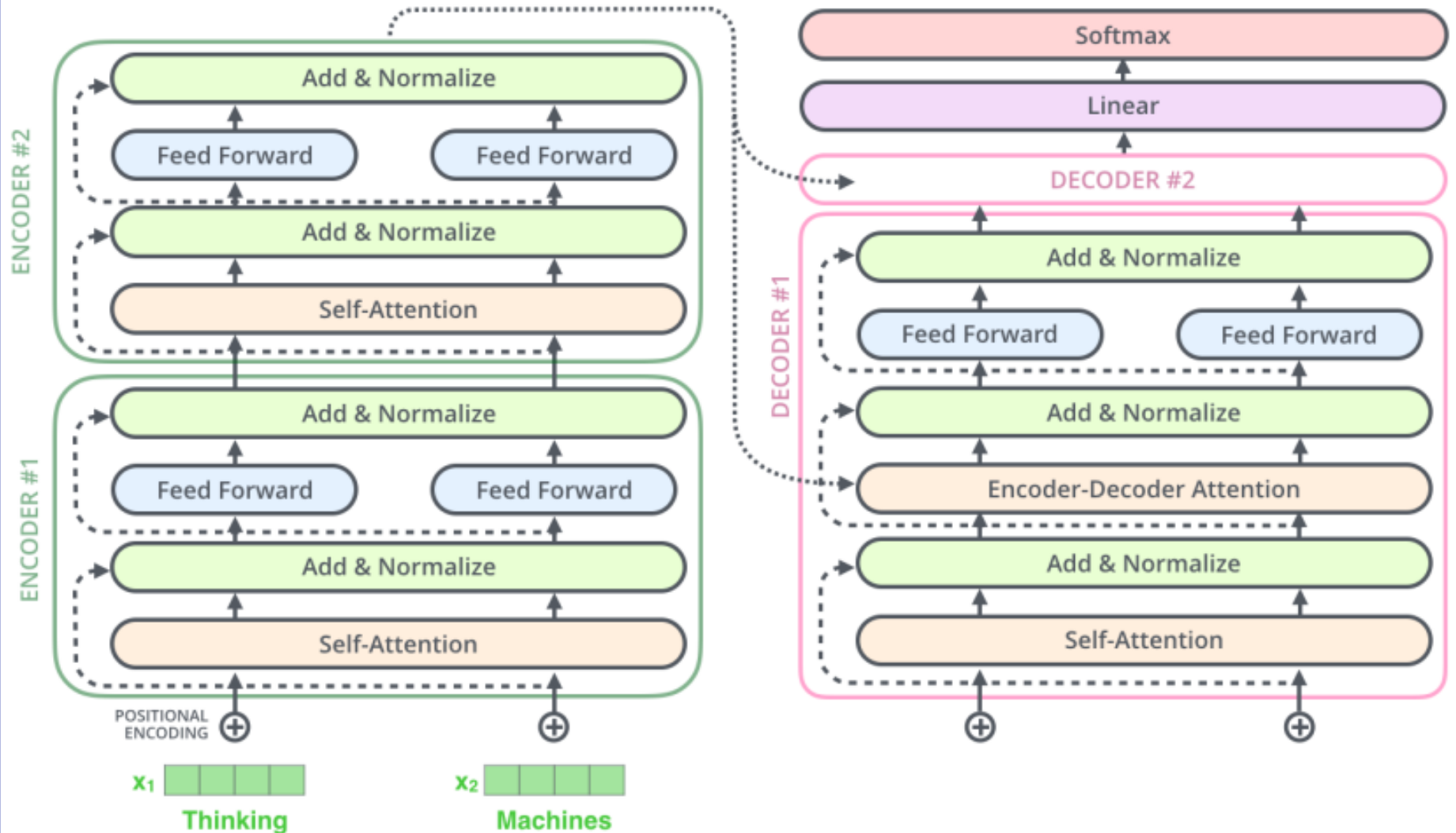
Transformer Architecture



Main Learnable Matrices in Transformers

Component	Learnable Parameters
Embedding	E
Attention	W^K, W^Q, W^V
Multi-head Attention	W^O
Feed-forward	W_1, W_2
Positional embedding	<i>Usually fixed (Optional)</i>
LayerNorm	γ : Scale parameter β : Bias parameter

Encoder - Decoder



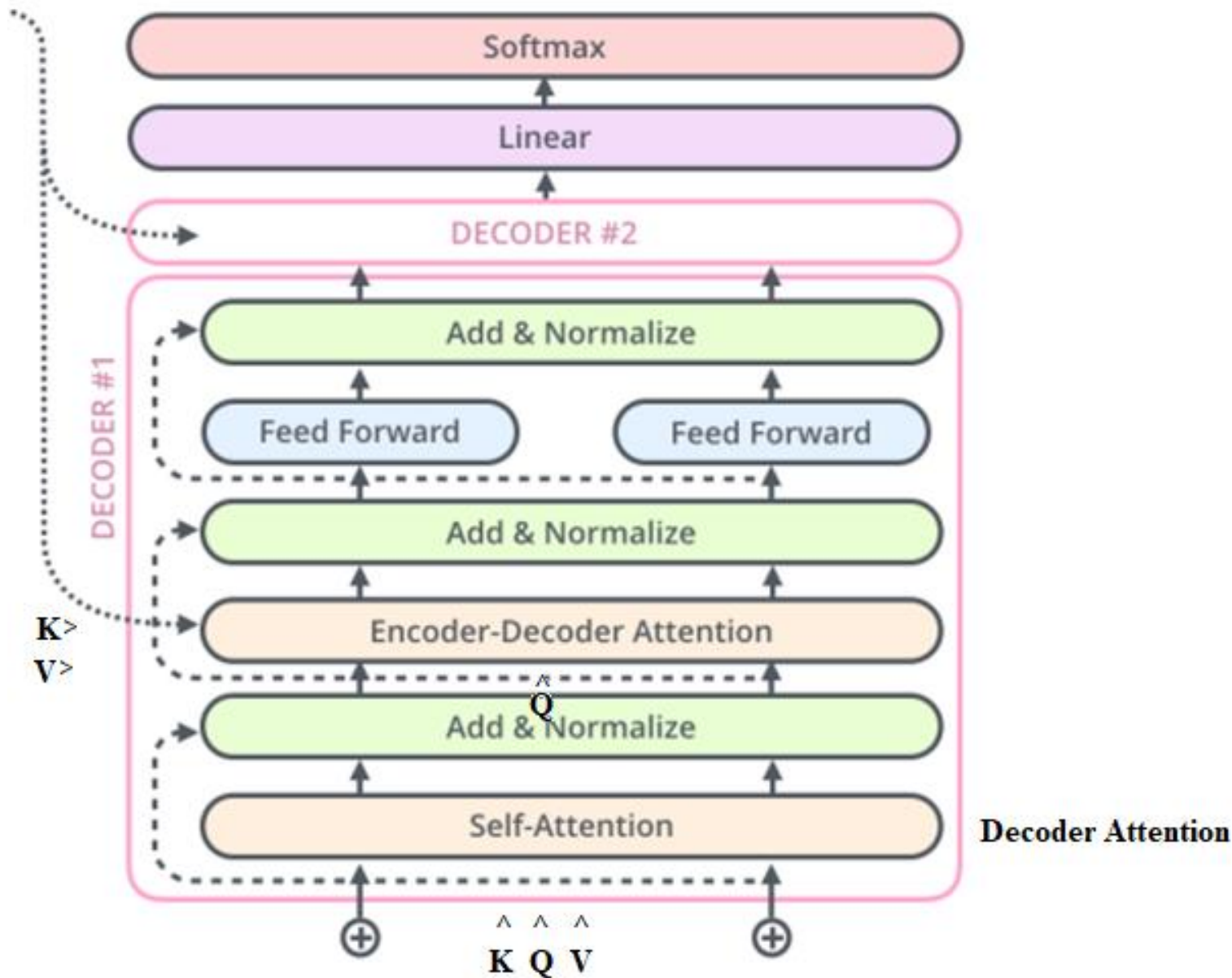
Transformer architecture - Encoder

- It is a stack of multiple identical layers.
- Each layer has two sublayers:
 - 1) multi-head self-attention pooling
 - 2) a position wise feed-forward network (MLP).
- In the encoder, queries, keys, and values are all from the outputs of the previous encoder layer.
- Inspired by the ResNet, a residual connection is employed around both sublayers.
- The addition from the residual connection is immediately followed by **layer normalization**.
- It outputs a d -dimensional vector representation for **each position** of the input sequence.

Transformer architecture - Decoder

- It is also a stack of multiple identical layers with residual connections and layer normalizations.
- Besides the two sublayers described in the encoder, the decoder inserts a third sublayer, known as the encoder-decoder attention, between these two.
- So, It has three sublayers:
 - 1) decoder self-attention,
 - 2) encoder-decoder attention,
 - 3) position wise feed-forward networks.
- In the encoder-decoder attention, **queries** are from the outputs of the previous decoder layer, and the **keys and values** are from the transformer encoder outputs.
- In the decoder self-attention, **queries, keys, and values** are all from the outputs of the previous decoder layer.
- However, each position in the decoder is allowed to only attend to all positions in the decoder up to that position. This **masked** attention preserves the **auto-regressive** property, ensuring that the prediction only depends on those output tokens that have been generated.

Transformer architecture - Decoder



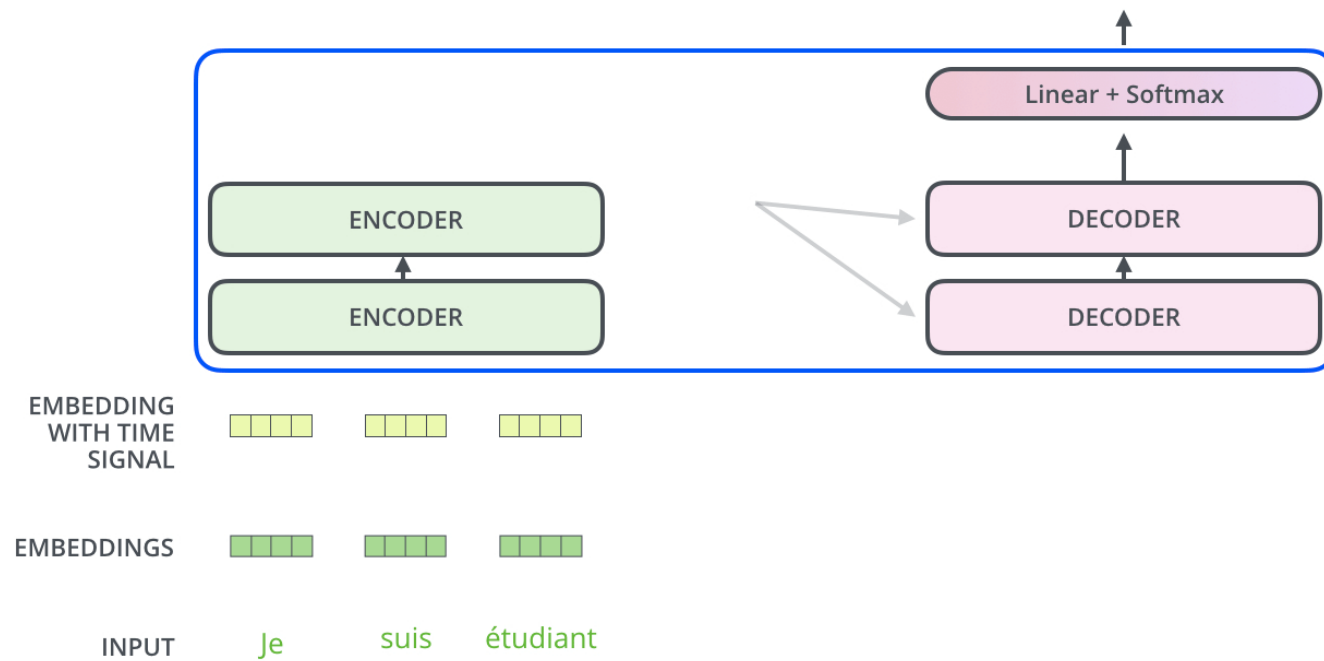
Transformer architecture – Decoder (2)

- In **training step**, tokens at all the positions (time steps) of the output sequence are known.
- In **prediction (test)**, the output sequence is generated token by token; thus, at any decoder time step, only the generated tokens can be used in the decoder self-attention.

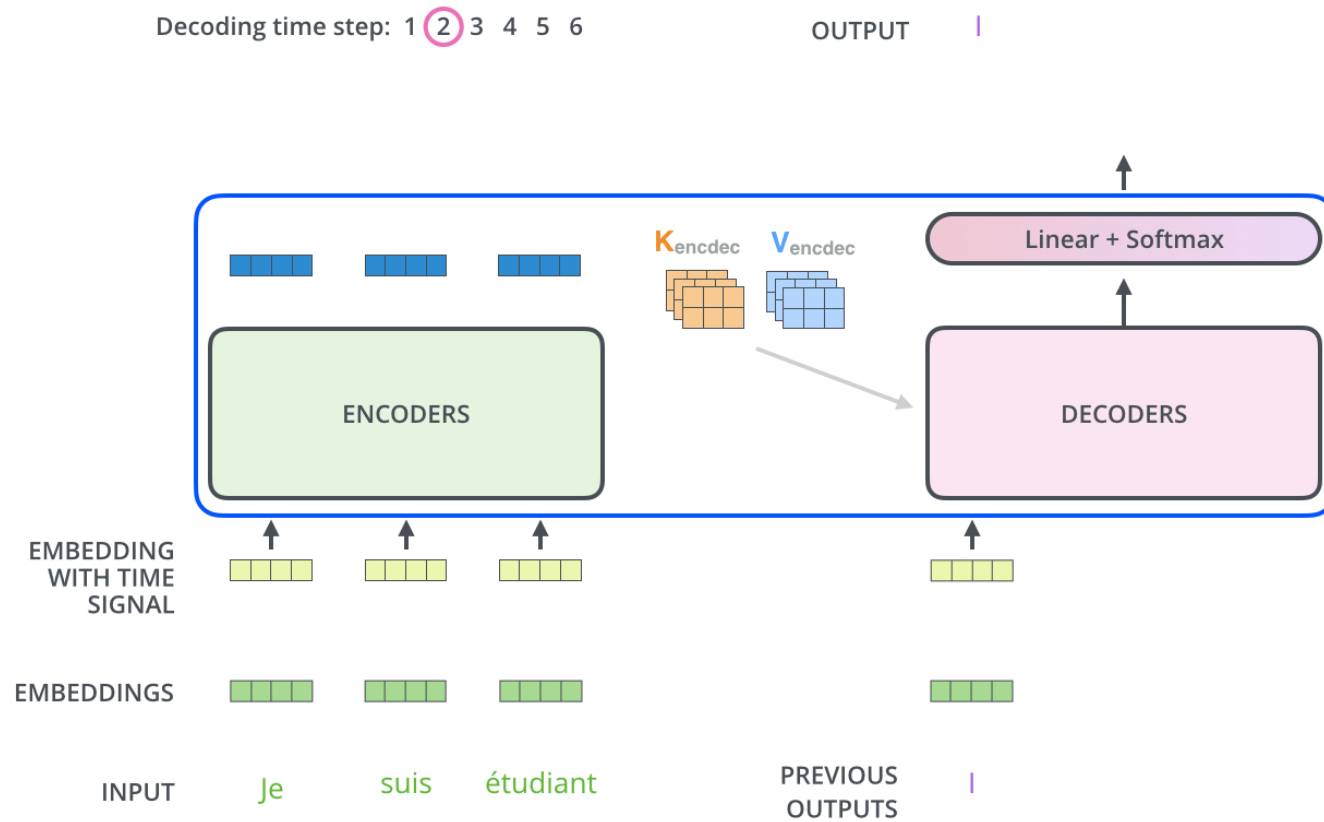
Decoder

Decoding time step: 1 2 3 4 5 6

OUTPUT



Decoder



Decoder

- The self attention layers in the decoder operate in a slightly different way than the one in the encoder.
- In the decoder, the self-attention layer is 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.

The Final Linear and Softmax Layer

- The decoder stack outputs a vector of floats. How do we turn that into a word? That's the job of the final Linear layer which is followed by a Softmax Layer.
- The **Linear layer is a simple fully connected neural network** that projects the vector produced by the stack of decoders, into a much, much larger vector called a logits vector.
- Let's assume that our model knows 10,000 unique English words (our model's "output vocabulary"). This would make the logits vector 10,000 cells wide – each cell corresponding to the score of a unique word. That is how we interpret the output of the model followed by the Linear layer.
- The softmax layer then turns those scores into probabilities (all positive, all add up to 1.0). The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step.

The Final Linear and Softmax Layer

Which word in our vocabulary
is associated with this index?

Get the index of the cell
with the highest value
(**argmax**)

am

5

log_probs



Softmax

logits



Linear

Decoder stack output



Simple self-attention code

```
import torch
import torch.nn as nn
import torch.nn.functional as F

# Example input
# 3 words, embedding size = 4
x = torch.tensor([
    [1.0, 0.0, 1.0, 0.0], # word 1
    [0.0, 2.0, 0.0, 2.0], # word 2
    [1.0, 1.0, 1.0, 1.0]  # word 3
])

# Self-Attention Module
class SelfAttention(nn.Module):
    def __init__(self, embed_size):
        super().__init__()

        # Learnable matrices
        self.W_Q = nn.Linear(embed_size, embed_size, bias=False)
        self.W_K = nn.Linear(embed_size, embed_size, bias=False)
        self.W_V = nn.Linear(embed_size, embed_size, bias=False)
```

Simple self-attention code ...

```
def forward(self, x):  
  
    # Create Query, Key, Value  
    Q = self.W_Q(x)  
    K = self.W_K(x)  
    V = self.W_V(x)  
  
    # Compute attention scores  
    scores = torch.matmul(Q, K.T)  
  
    # Scale scores  
    d_k = K.shape[-1]  
    scaled_scores = scores / (d_k ** 0.5)  
  
    # Softmax  
    attention_weights = F.softmax(scaled_scores, dim=-1)  
  
    # Weighted sum  
    output = torch.matmul(attention_weights, V)  
  
    return output, attention_weights
```

Simple self-attention code ...

```
# Create model
model = SelfAttention(embed_size=4)

# Forward pass
output, attention = model(x)

print("Attention Weights:")
print(attention)

print("\nOutput:")
print(output)
```

References

- https://d2l.ai/chapter_attention-mechanisms-and-transformers/index.html
- <https://jalammar.github.io/illustrated-transformer/>