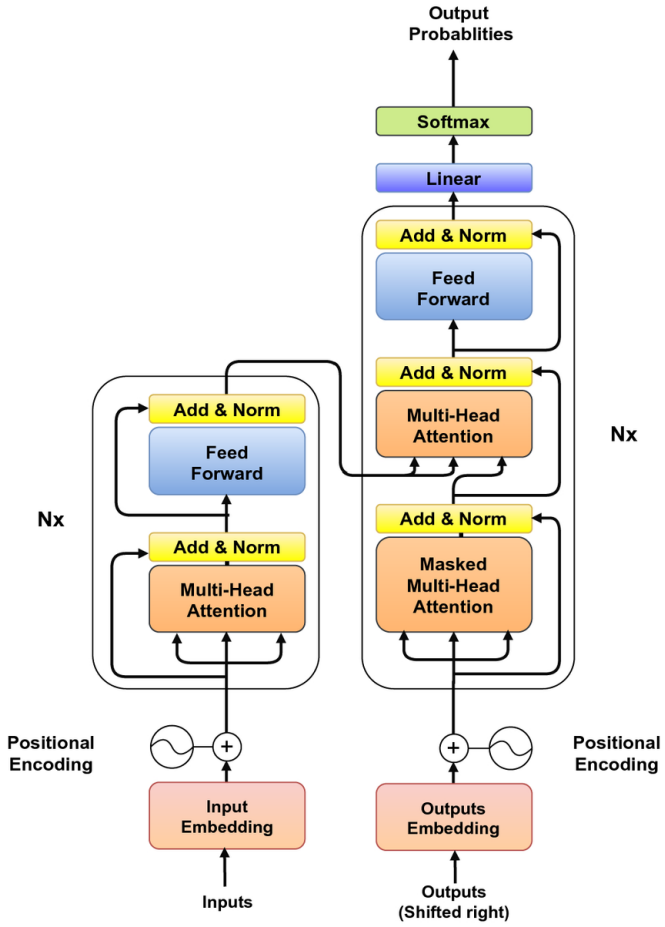
TRANSFORMERS

Transformers are a neural network architecture developed to solve the problem of sequence transduction, or neural machine translation. That means any task that transforms an input sequence to an output sequence. This includes speech recognition, text-to-speech transformation, etc..

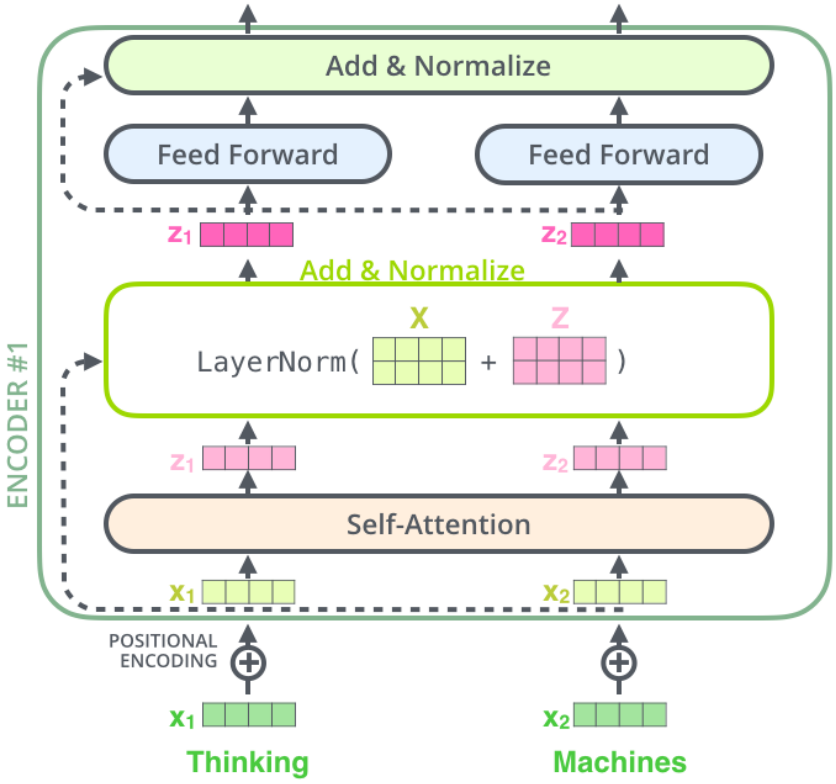
The transformer models are based on ideas of complex recurrent or convolutional neural networks that include an encoder and a decoder connected through an attention mechanism. Transformers are different from traditional RNNs because they avoid recursion in order to allow parallel computation (to reduce training time) and also to reduce drops in performance due to long dependencies.

Architecture of Transformers

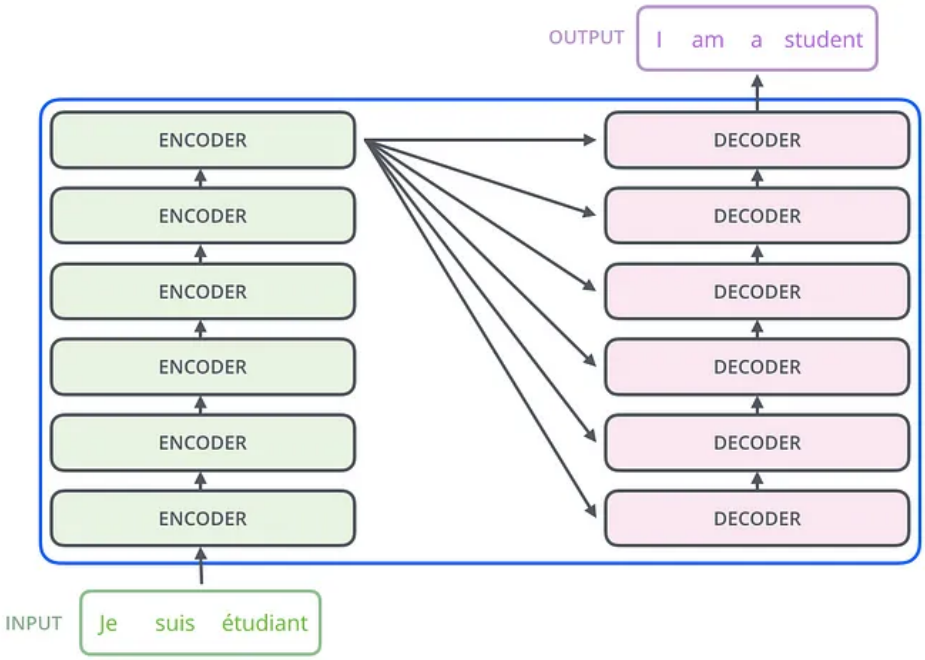


The encoder maps an input sequence of symbol representations (x1, ..., xn) to a sequence of continuous representations z = (z1, ..., zn). Given z, the decoder then generates an output sequence (y1, ..., ym) of symbols one element at a time. At each step the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next. The Transformer follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder.

Encoder: The encoder comprises a stack of N = 6 identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position wise, fully connected feed-forward network. We employ a residual connection around each of the two sub-layers, followed by layer normalization. That is, the output of each sub-layer is LayerNorm(x + Sublayer(x)), where Sublayer(x) is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of the same dimension. Here d_model = 512.



Decoder: The decoder is also composed of a stack of N = 6 identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with the fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.



There are different types of transformer models :

- Encoder-only models: Good for tasks that require understanding of the input, such as sentence classification and named entity recognition. Eg : BERT (also called *auto-encoding* Transformer models)
- Decoder-only models: Good for generative tasks such as text generation. Eg: GPT (also called *auto-regressive* Transformer models)
- Encoder-decoder models or sequence-to-sequence models: Good for generative tasks that require an input, such as translation or summarization. BART/T5-like (also called *sequence-to-sequence* Transformer models).

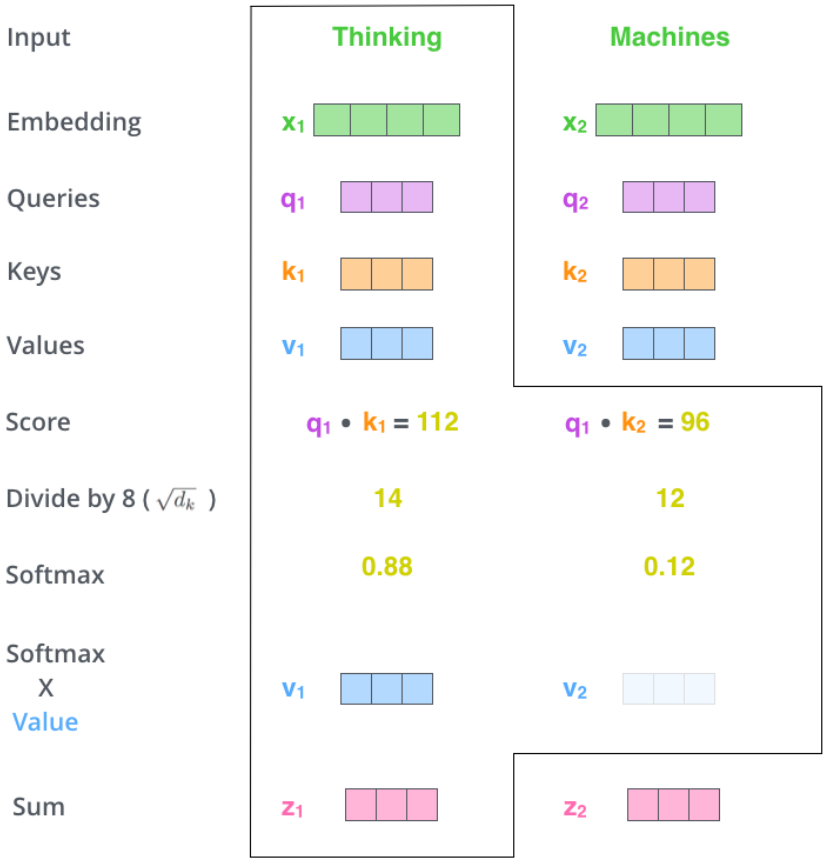
Attention

The transformer building blocks are scaled dot-product attention units. For each attention unit, the transformer model learns three weight matrices; the query weights W_Q, the key weights W_K, and the value weights W_V. For each token i, the input word embedding X_i is multiplied with each of the three weight matrices to produce a query vector q_i = X_i * W_Q, a key vector k_i = X_i * W_K and a value vector v_i = X_i * W_V.

Attention weights are calculated using the query and key vectors: the attention weight a_i j
from token i to token j is the dot product between q_i and k_j . The attention weights are divided by the square root of the dimension of the key vectors, which stabilizes gradients during training, and then passed through a softmax which normalizes the weights.

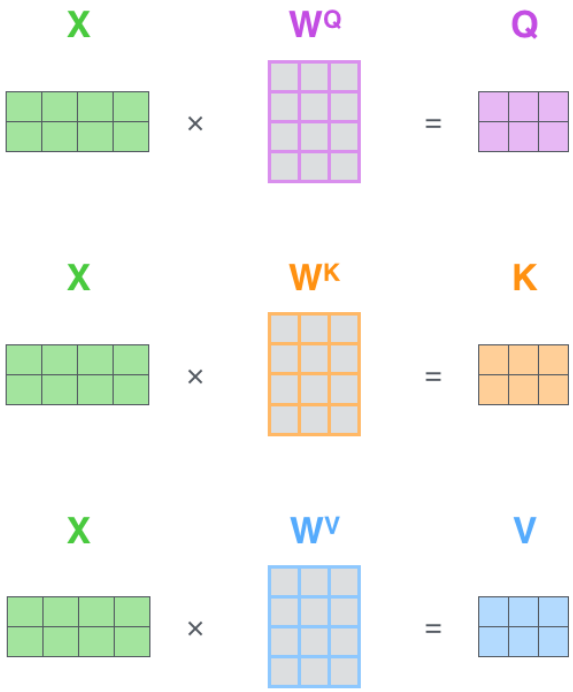
$$\text{Attention}(Q, K, V) = \text{softmax}(Q*K.T \sqrt{} dk) * V$$

Illustration considering one input at a time:



The two most commonly used attention functions are additive attention, and dot-product (multiplicative) attention. Dot-product attention is identical to our algorithm, except for the scaling factor of √ 1 dk . Additive attention computes the compatibility function using a feed-forward network with a single hidden layer. While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code. While for small values of dk the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of dk. We suspect that for large values of dk, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients 4 . To counteract this effect, we scale the dot products by 1/ √dk.

How is it done using matrices :



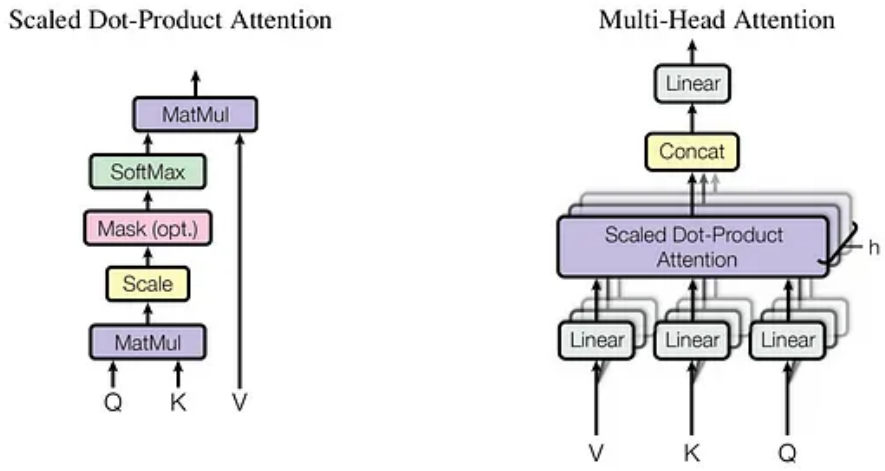


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

Multi-head attention

One set of (W_Q,W_K,W_V) matrices is called an *attention head*, and each layer in a transformer model has multiple attention heads. While each attention head attends to the tokens that are relevant to each token, with multiple attention heads the model can do this for different definitions of "relevance". In addition, the influence field representing relevance can become progressively dilated in successive layers. Many transformer attention heads encode relevance relations that are meaningful to humans. For example, some attention heads can attend mostly to the next word, while others mainly attend from verbs to their direct objects. The computations for each attention head can be performed in parallel, which allows for fast processing. The outputs for the attention layer are concatenated to pass into the feed-forward neural network layers.

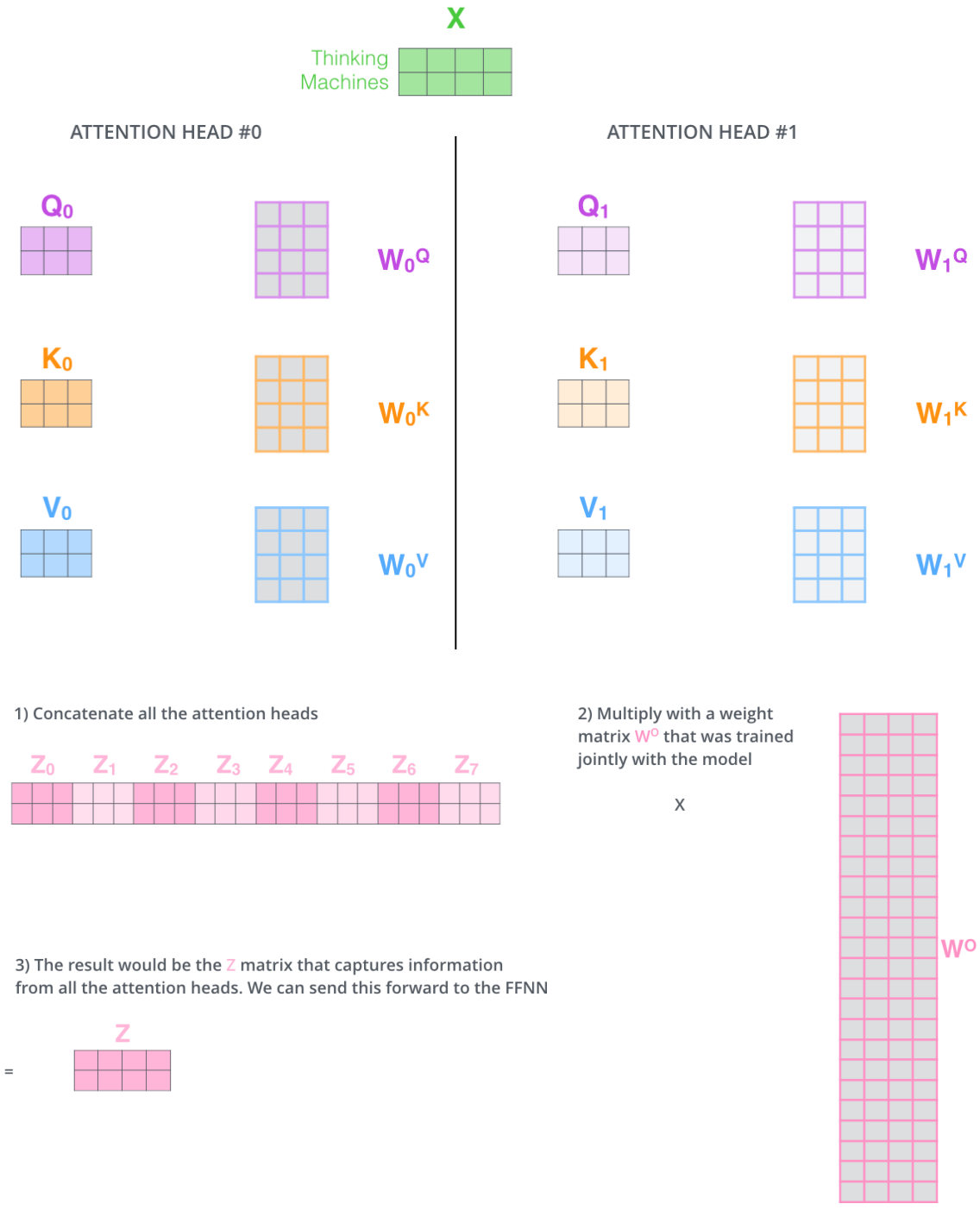
Concretely, let the multiple attention heads be indexed by i, then we have

MultiheadedAttention(Q,K,V) = Concat(Attention(X*WQ_i,X*WK_i, X*WV_i)) * Wo

Where, the matrix X is the concatenation of word embeddings, and the matrices WQ_i, WK_i, WV_i are "projection matrices" owned by individual attention heads and Wo is a final projection matrix owned by the whole multi-headed attention head.

W Q i ∈ R d_model × d_k
W K i ∈ R d_model × d k
WV i ∈ R d_model × d_v
WO ∈ R h*d_v × d_model . h = number of attention heads.

In this work we employ h = 8 parallel attention layers, or heads. For each of these we use dk = dv = dmodel/h = 64. Due to the reduced dimension of each head, the total computational cost is similar to that of single-head attention with full dimensionality.



All these happen in only one encoder. Remember there are 6 of them. The output of the first encoder is input for the second and so on.

Masked attention

The first sublayer of the decoder stack receives the previous output of the decoder stack, augments it with positional information, and implements multi-head self-attention over it. While the encoder is designed to attend to all words in the input sequence regardless of their position in the sequence, the decoder is modified to attend only to the preceding words. Hence, **the prediction for a word at position i can only depend on the known outputs for the words that come before it in the sequence**. In the multi-head attention mechanism (which implements multiple, single attention functions in parallel), this is achieved by introducing a mask over the values produced by the scaled multiplication of matrices, Q and K. This masking is implemented by suppressing the matrix values that would otherwise correspond to illegal connections:

$$\text{mask}(QK^T) = \text{mask}\left(\begin{bmatrix} e_{11} & e_{12} & \dots & e_{1n} \\ e_{21} & e_{22} & \dots & e_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ e_{m1} & e_{m2} & \dots & e_{mn} \end{bmatrix}\right) = \begin{bmatrix} e_{11} & -\infty & \dots & -\infty \\ e_{21} & e_{22} & \dots & -\infty \\ \vdots & \vdots & \ddots & \vdots \\ e_{m1} & e_{m2} & \dots & e_{mn} \end{bmatrix}$$

Position-wise Feed-Forward Networks

In addition to attention sub-layers, each of the layers in our encoder and decoder contains a fully connected feed-forward network, which is applied to each position separately and identically. This consists of two linear transformations with a ReLU activation in between.

$$FFN(x) = \max(0, xW1 + b1)W2 + b2 .$$

The dimensionality of input and output is d_model = 512, and the inner-layer has dimensionality df f = 2048.

The final Linear Layers and Softmax Layer

The decoder stack outputs a vector of floats. The Linear layer 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") that it's learned from its training dataset. This would make the logits vector 10,000 cells wide – each cell corresponding to the score of a unique word. 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.

Now that we have learnt about the architecture, we will look into how data is fitted into a transformer model. In this project, we are going to look into two different types of data : text and time series.
For text data , follow these steps :

1. Generate input embeddings from the text. The process of generating embeddings is as follows :
    a. Initially, add a start and an end token to the sentences of both the languages. <START> + sentence + <END> would do the work.
    b. Tokenization: Next, break down the text document into smaller units, called tokens. This can be done using various techniques, such as word-level tokenization, character-level tokenization, or subword-level tokenization.
    c. Vocabulary creation: Next, a vocabulary of unique words is created from the tokens in the text. This vocabulary is used to map each word to a unique integer index.
    d. Word embedding matrix: A word embedding matrix is created, which is a matrix of size (vocab_size x embedding_size), where vocab_size is the size of the vocabulary and embedding_size is the dimensionality of the embedding vectors. The values in the matrix are initialized randomly or pre-trained using techniques such as Word2Vec or GloVe.
    e. Padding: In some cases, the input sequences may have different lengths. To handle this, padding is added to make all sequences of the same length. This is usually done by adding zeros or a special padding token to the end of shorter sequences.

2. Positional Encoding : A positional encoding is a fixed-size vector representation that encapsulates the relative positions of tokens within a target sequence: it provides the transformer model with information about *where* the words are in the input sequence. We add "positional encodings" to the input embeddings at the bottoms of the encoder and decoder stacks. The positional encodings have the same dimension d_model as the embeddings, so that the two can be summed. We use sine and cosine functions of different frequencies to generate positional encodings :

$$\begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \qquad PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

where pos is the position of the word embedding and i is the dimension.

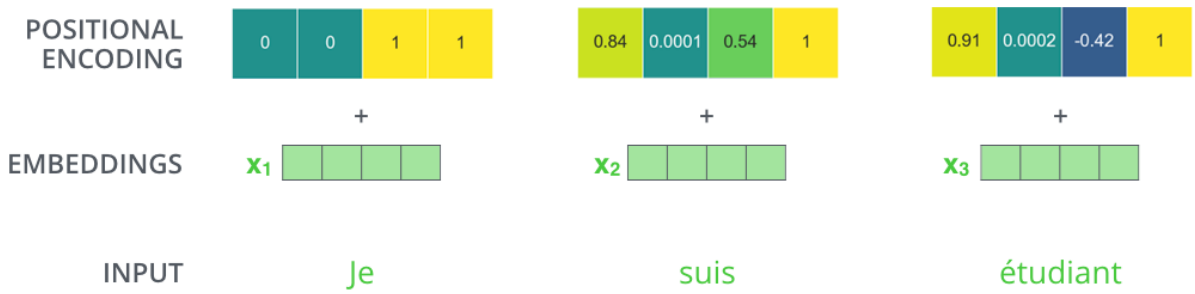
Example Code

d_model = 300 (dimensions of the word embedding.)
pos = position of word in the sentence
j = (2i or 2i +1) = jth element in the embedding vector of the word.
j varies from 1 to 300.
Max_len = number of words in the sentence

```
def get_positional_encoding(max_len, d_model):

    pos_enc = np.zeros((max_len, d_model))

    for pos in range(max_len):
        for i in range(d_model):
            if i % 2 == 0:
                pos_enc[pos, i] = np.sin(pos / (10000 ** (i / d_model)))
            else:
                pos_enc[pos, i] = np.cos(pos / (10000 ** ((i - 1) / d_model)))
    return pos_enc
```



3. The most important step is how the data is passed to the model.
   The input data is supposed to be of the following form (inputs, targets), labels .
   Inputs refer to the X_train or X_test. The sentences of the first language excluding the START token.
   Targets refer to sentences of second language excluding the END token.
   Labels refer to sentences of second language excluding the START token.

   While predicting, the targets are provided one at a time.
   Model.fit() work perfectly but Model.evaluate() and Model.predict() don't. Those functions have to be built manually.

In case of time-series data, we find time2vec embeddings of the data and concatenate it on the data. https://arxiv.org/pdf/1907.05321v1.pdf