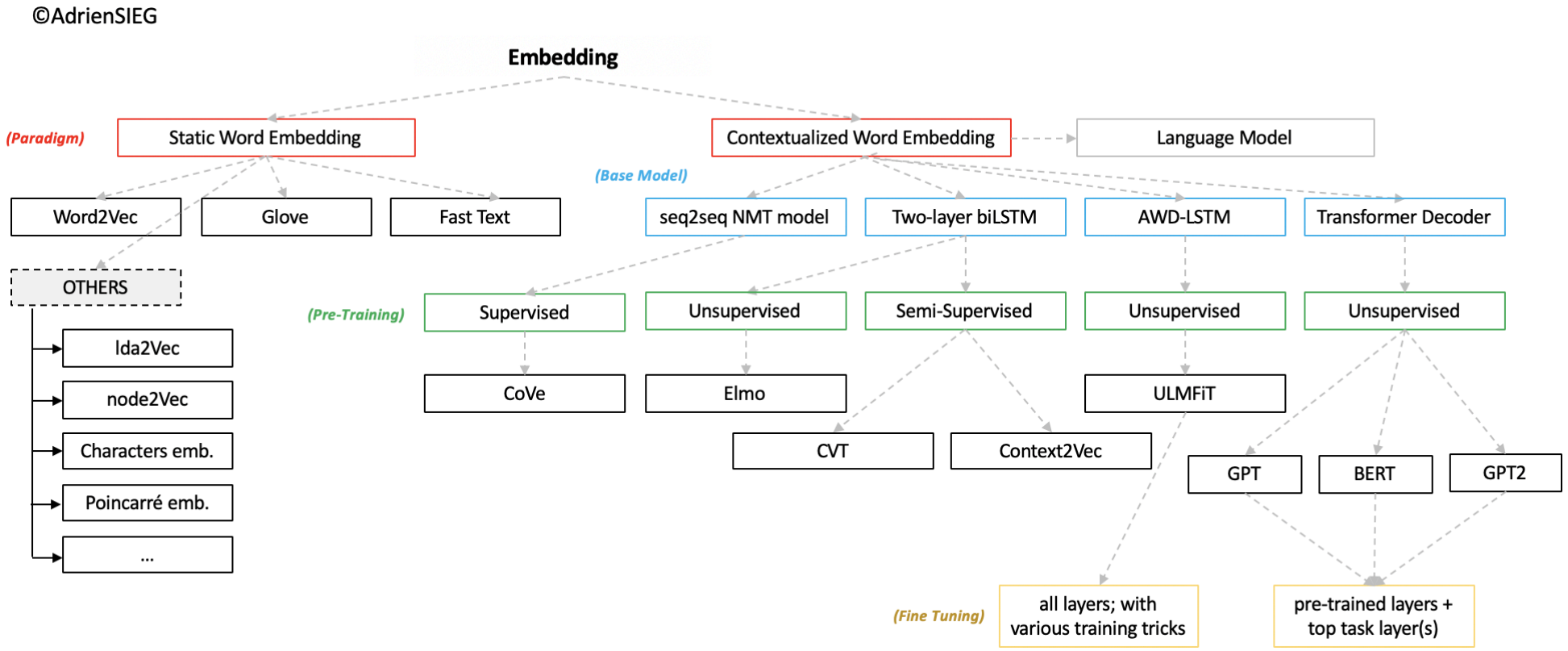
From Pre-trained Word Embeddings TO Pre-trained Language Models - Focus on Transformers

based on https://towardsdatascience.com/from-pre-trained-word-embeddings-to-pre-trained-language-models-focus-on-bert-343815627598

The Illustrated Transformers (https://jalammar.github.io/illustrated-transformer/)



Static Word Embedding

- Skip-Gram & CBOW (aka Word2Vec)
- Glove
- fastText

- Exotic: Lda2Vec, Node2Vec, Characters Embeddings, CNN embeddings, ...
- Poincaré Embeddings to learn hierarchical representation

Contextualized (Dynamic) Word Embedding (LM)

- CoVe (Contextualized Word-Embeddings)
- CVT (Cross-View Training)
- ELMO (Embeddings from Language Models)
- ULMFiT (Universal Language Model Fine-tuning)
- BERT (Bidirectional Encoder Representations from Transformers)
- GPT & GPT-2 (Generative Pre-Training)
- Transformer XL (meaning extra long)
- XLNet (Generalized Autoregressive Pre-training)
- ENRIE (Enhanced Representation through kNowledge IntEgration)
- (FlairEmbeddings (Contextual String Embeddings for Sequence Labelling))

What is a good model?

Best models would be able to capture 4 components:

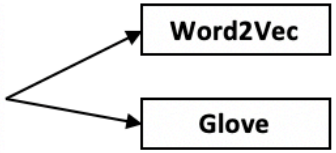
- Lexical approach (relating to the words or vocabulary of a language)
- Syntactic approach (the arrangement of words and phrases to create well-formed sentences in a language -> grammar)
- Semantic approach (relating to meaning in language -> stretch the words beyond, understanding ambiguities)
- Pragmatic approach (relating proximity between words and documents)

What are the main families of models?

1

These pre-trained embeddings specify a function which maps elements v in a (word) vocabulary V to vectors, $h \in \mathbb{R}^d$, that is:

$$f_{\text{vocab}} : v \rightarrow h$$



2

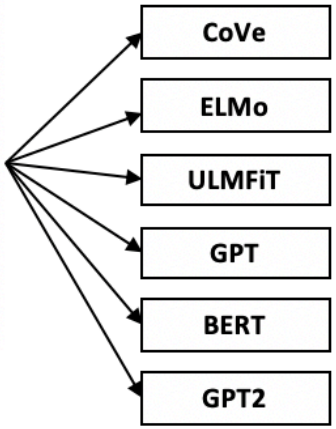
Subword-informed methods like FastText build on the intuition that the literal character sequences of language often are indicative of compositional information about the meaning of the word. These methods map from tuples of vocabulary item v and character sequence $(c_1, \ldots, c_t)$ to vectors:

$$f_{\text{subword}} : (v, (c_1, \ldots, c_t)) \rightarrow h$$

3

Contextual representations of language are functions that have yet a more expressive type signature; they leverage the intuition that the meaning of a particular word in a particular text depends not only on the identity of a word, but also on the words that surround it at that moment. Let $(w_1, w_2, .., w_N)$ be a text (say, a sentence) where each $w_i \in V$ is a word. A contextual representation of language is a function on the whole text, which assigns a vector representation to each word in the sequence:

$$f_{\text{contextual}} : (w_1, \ldots, w_N) \rightarrow (h_1, \ldots, h_N)$$



These models are trained on objectives that are something along the lines of **language modeling**

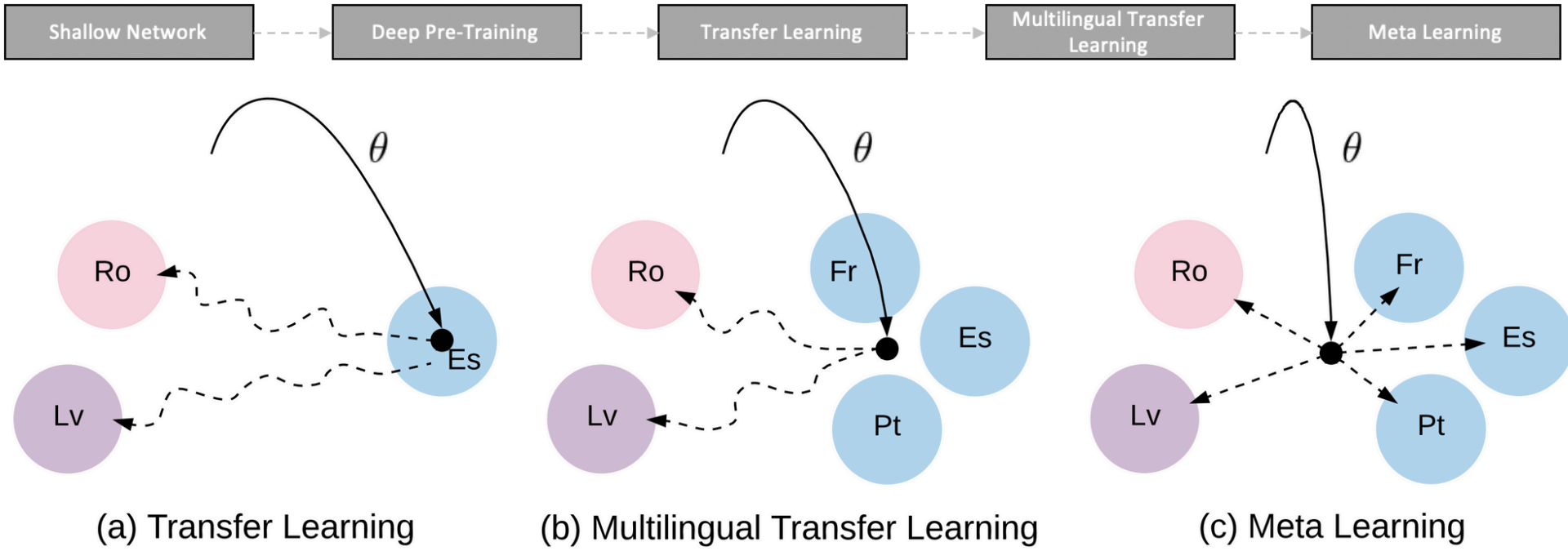
Language modeling is the task of assigning a probability distribution over sequences of words that matches the distribution of a language. Although it sounds formidable, language modeling (i.e. ELMo, BERT, GPT) is essentially just predicting words in a blank. More formally, given a context, a language model predicts the probability of a word occurring in that context.

Why is this method effective? Because this method forces the model to learn how to use information from the entire sentence in deducing what words are missing.

0. Static vs. Dynamic

- Static Word Embeddings fail to capture polysemy. They generate the same embedding for the same word in different contexts.
- Contextualized words embeddings aim at capturing word semantics in different contexts to address the issue of polysemous and the context-dependent nature of words.
- Static Word Embeddings could only leverage off the vector outputs from unsupervised models for downstream tasks — not the unsupervised models themselves.They were mostly shallow models to begin with and were often discarded after training (e.g. word2vec, Glove)
- The output of Contextualized (Dynamic) Word Embedding training is the trained model and vectors — not just vectors.
- Traditional word vectors are shallow representations (a single layer of weights, known as embeddings). They only incorporate previous knowledge in the first layer of the model. The rest of the network still needs to be trained from scratch for a new target task. They fail to capture higher-level information that might be even more useful. Word embeddings are useful in only capturing semantic meanings of words but we also need to understand higher level concepts like anaphora, long-term dependencies, agreement, negation, and many more.

Evolutions



Transfer learning — a technique where instead of training a model from scratch, we use **models pre-trained on a large dataset** and **then fine-tune them for specific natural language tasks.**

Some particularities :

- ULMFiT → Transfer by Fine Tuning
- ELMo → Transfer by Features Extraction
- BERT → Transfer by Attention Extraction

Why using Transfer Learning?

In vision, it has been in practice for some time now, with people using models trained to learn features from the huge ImageNet dataset, and then training it further on smaller data for different tasks.

- Most datasets for text classification (or any other supervised NLP tasks) are rather small. This makes it very difficult to train deep neural networks, as they would tend to overfit on these small training data and not generalize well in practice.

In computer vision, for a couple of years now, the trend is to pre-train any model on the huge ImageNet corpus. This is much better than a random initialization because the model learns general image features and that learning can then be used in any vision task (say captioning, or detection).

In NLP, we trained on a general language modeling (LM) task and then fine tuned on text classification (or other task). This would, in principle, perform well because the model would be able to use its knowledge of the semantics of language acquired from the generative pre-training.

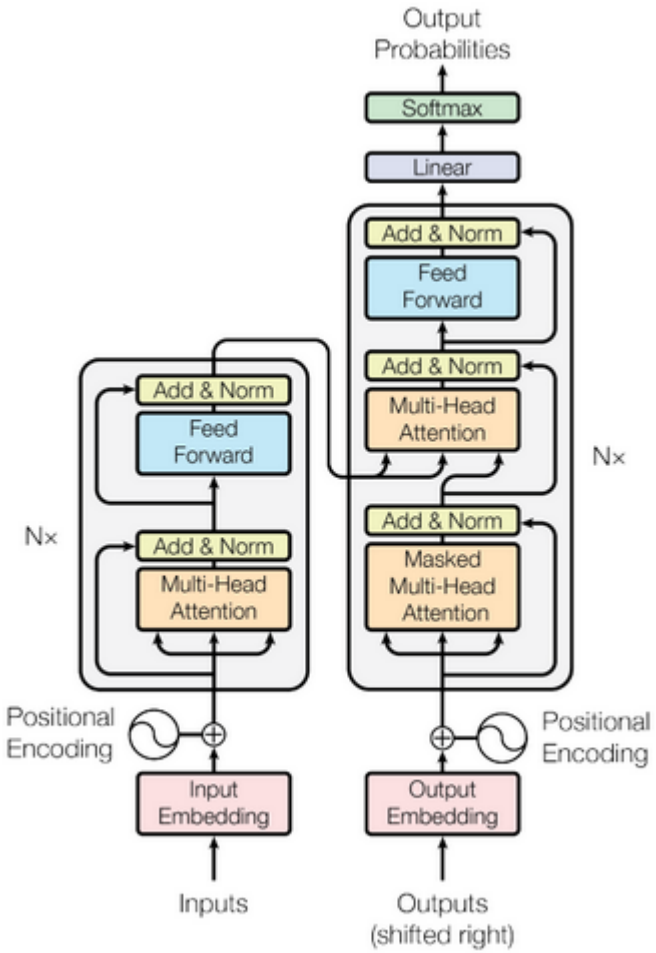
- it is able to capture long-term dependencies in language
- it effectively incorporates hierarchical relations
- it can help the model learn sentiments
- large data corpus is easily available for LM

The Transformer

A model that uses attention to boost the speed with which these models can be trained. The Transformers outperforms the Google Neural Machine Translation model in specific tasks. The biggest benefit, however, comes from how The Transformer lends itself to parallelization. It is in fact Google Cloud's recommendation to use The Transformer as a reference model to use their Cloud TPU offering. So let's try to break the model apart and look at how it functions.

The Transformer was proposed in the paper *Attention is All You Need*. A TensorFlow implementation of it is available as a part of the Tensor2Tensor package. Harvard's NLP group created a guide annotating the paper with PyTorch implementation. In this post, we will attempt to oversimplify things a

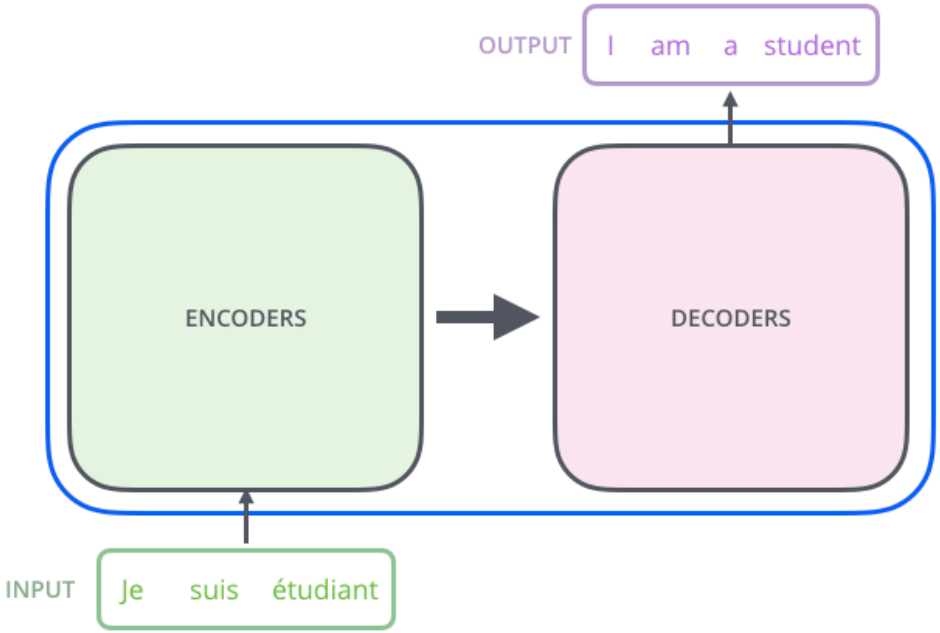
bit and introduce the concepts one by one to hopefully make it easier to understand to people without in-depth knowledge of the subject matter.



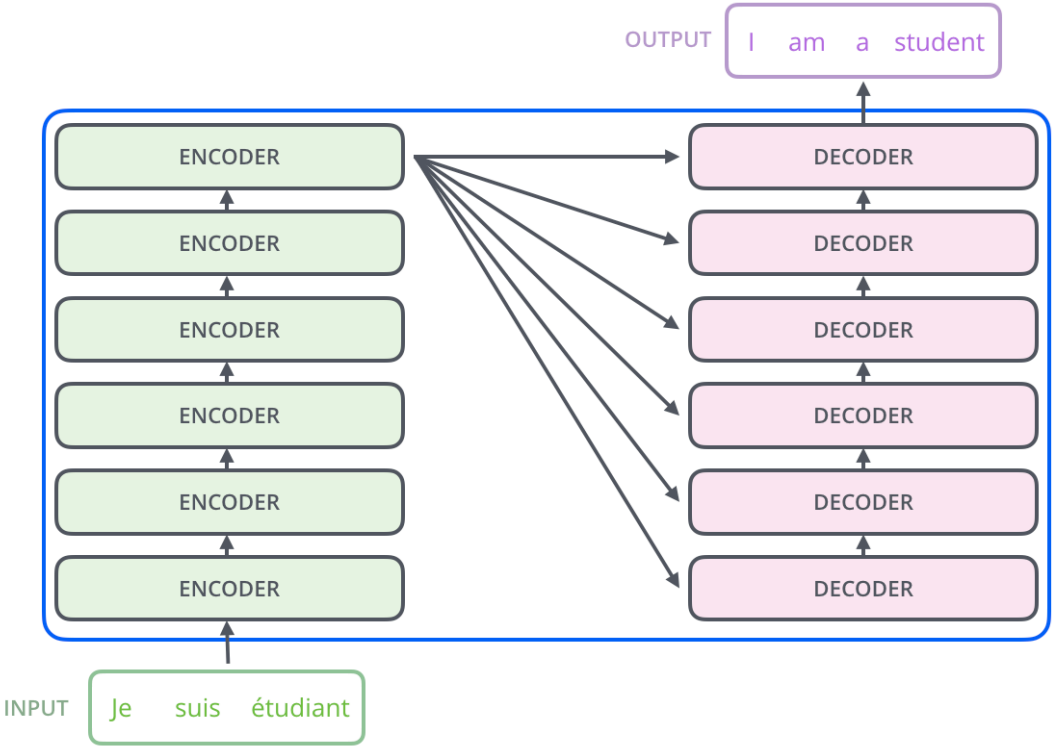
A High-Level Look

Let's begin by looking at the model as a single black box. In a machine translation application, it would take a sentence in one language, and output its translation in another.

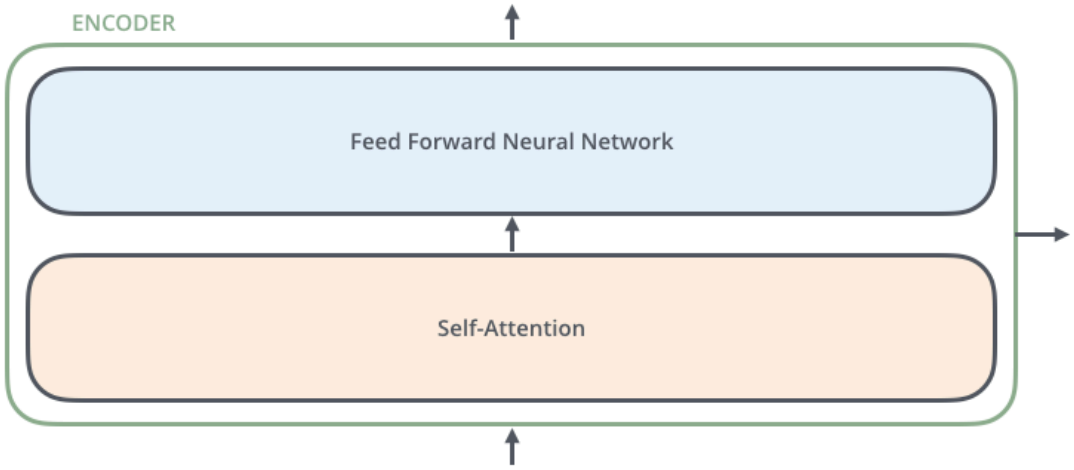
Popping open that Optimus Prime goodness, we see an encoding component, a decoding component, and connections between them.



The encoding component is a stack of encoders (the paper stacks six of them on top of each other – there's nothing magical about the number six, one can definitely experiment with other arrangements). The decoding component is a stack of decoders of the same number.



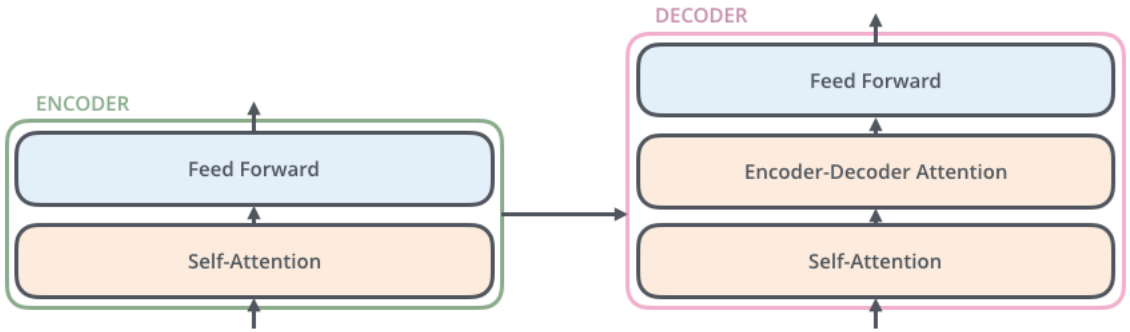
The encoders are all identical in structure (yet they do not share weights). Each one is broken down into two sub-layers:



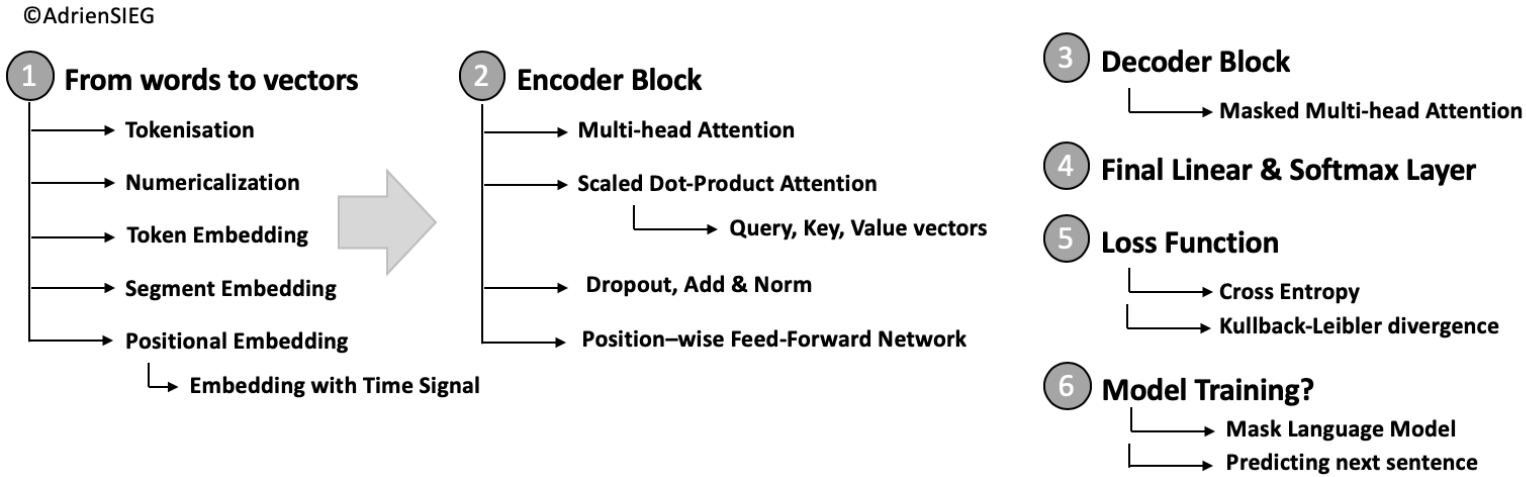
The encoder's inputs first flow through a **self-attention layer** – a layer that helps the encoder look at other words in the input sentence as it encodes a specific word. We'll look closer at self-attention later in the post.

The outputs of the self-attention layer are fed to a feed-forward neural network. The exact same feed-forward network is independently applied to each position.

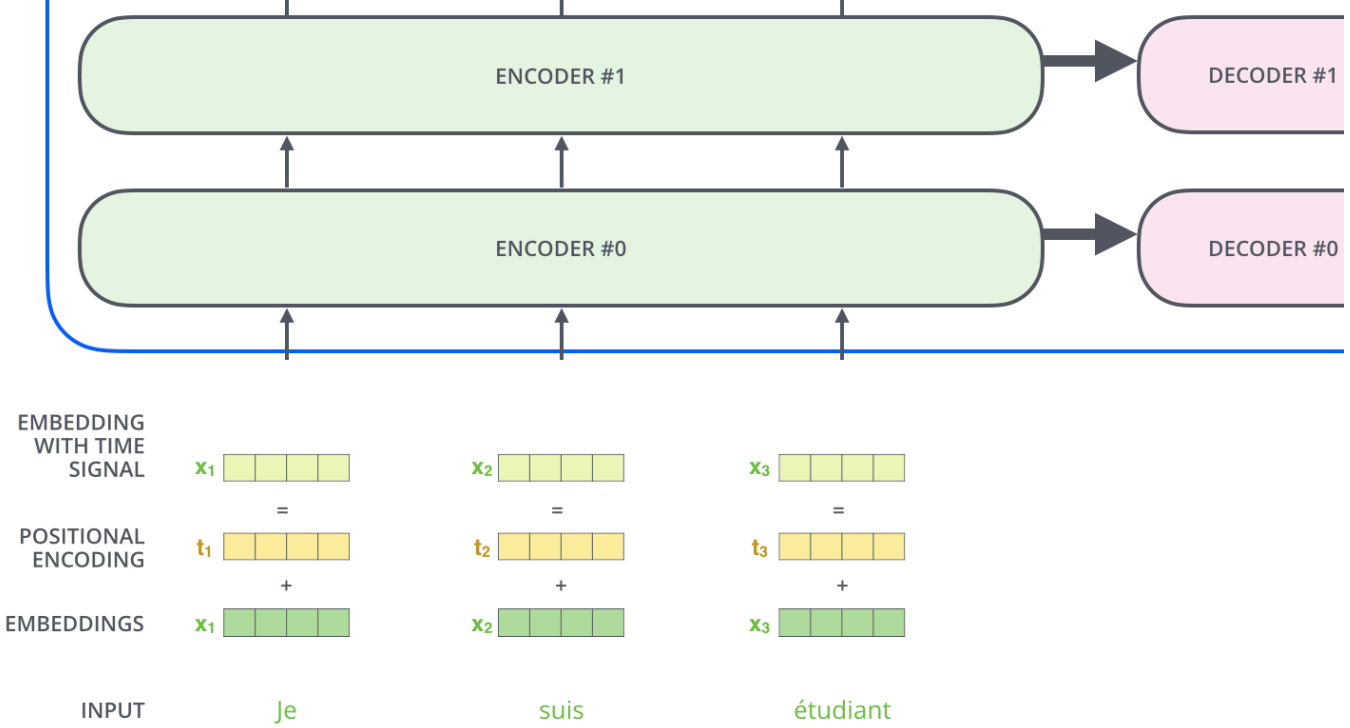
The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence (similar what attention does in seq2seq models).



Job?



From words to Vectors



- **Tokenization** is the task of chopping it up into pieces, called tokens , perhaps at the same time throwing away certain characters, such as punctuation.
- Using **wordpieces**(e.g. playing -> play + ##ing) instead of words. This is effective in reducing the size of the vocabulary and increases the amount of data that is available for each word.
- **Numericalization** aims at mapping each token to a unique integer in the corpus' vocabulary.
- **Token embedding** is the task of get the embedding (i.e. a vector of real numbers) for each word in the sequence. Each word of the sequence is mapped to a emb_dim dimensional vector that the model will learn during training. You can think about it as a vector look-up for each token. The elements of those vectors are treated as model parameters and are optimized with back-propagation just like any other weights.
- **Padding** was used to make the input sequences in a batch have the same length. That is, we increase the length of some of the sequences by adding 'pad' tokens.
- **Positional encoding**:

Recall that the **positional encoding** is designed to help the model learn some notion of **sequences and relative positioning of tokens**. This is crucial for language-based tasks especially here because we are not making use of any traditional recurrent units such as RNN, GRU or LSTM

Intuitively, we aim to be able to **modify the represented meaning of a specific word depending on its position**. We don't want to change the full representation of the word but we want to **modify it a little to encode its position** by adding numbers between [-1,1] using predetermined (non-

learned) sinusoidal functions to the token embeddings. For the rest of the **Encoder**, the word will be **represented slightly differently depending on the position the word is in** (even if it is the same word).

Encoder must be able to use the fact that **some words are in a given position** while, in the same sequence, other words are in other specific positions. That is, we want the network to able to **understand relative positions** and **not only absolute ones.**

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

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

Here the i denotes the vector index we are looking at, pos denotes the token, and d_{model} denotes a fixed constant representing the dimension of the input embeddings.
Ok let's break it down further.

The sinuosidal functions chosen by the authors allow positions to be represented as linear combinations of each other and thus allow the network to learn relative relationships between the token positions.

1 **Input** ----> *"Hello, how are you?"*

2 **Tokenisation** ----> *["Hello", ",", "how", "are", "you", "?"]*

3 **Numericalization** ----> *["Hello", ",", "how", "are", "you", "?"] → [34, 90, 15, 684, 55, 193]*

4 **Look-up** ---->

$$
\begin{aligned}
34 &\rightarrow E[34] = [123.4, 0.32, \cdots, 94, 32] \\
90 &\rightarrow E[90] = [83, 34, \cdots, 77, 19] \\
15 &\rightarrow E[15] = [0.2, 50, \cdots, 33, 30] \\
684 &\rightarrow E[684] = [289, 432.98, \cdots, 150, 92] \\
55 &\rightarrow E[55] = [80, 46, \cdots, 23, 32] \\
193 &\rightarrow E[193] = [41, 21, \cdots, 74, 33]
\end{aligned}
$$

Each word of the sequence is mapped to a **emb_dim dimensional vector** that the model will **learn during training**. The elements of those vectors are treated as **model parameters** and are optimized with **back-propagation** just like any other weights.

emb_dim dimensional vector → In paper ***"Attention is all you need"*** it is **512 dimensions**

5 **Stacking** ---->

$$
\begin{array}{c}
Hello \\ , \\ how \\ are \\ you \\ ?
\end{array}
\begin{pmatrix}
123.4 & 0.32 & \cdots & 94 & 32 \\
83 & 34 & \cdots & 77 & 19 \\
0.2 & 50 & \cdots & 33 & 30 \\
289 & 432.98 & \cdots & 150 & 92 \\
80 & 46 & \cdots & 23 & 32 \\
41 & 21 & \cdots & 74 & 33
\end{pmatrix}
$$

input_length

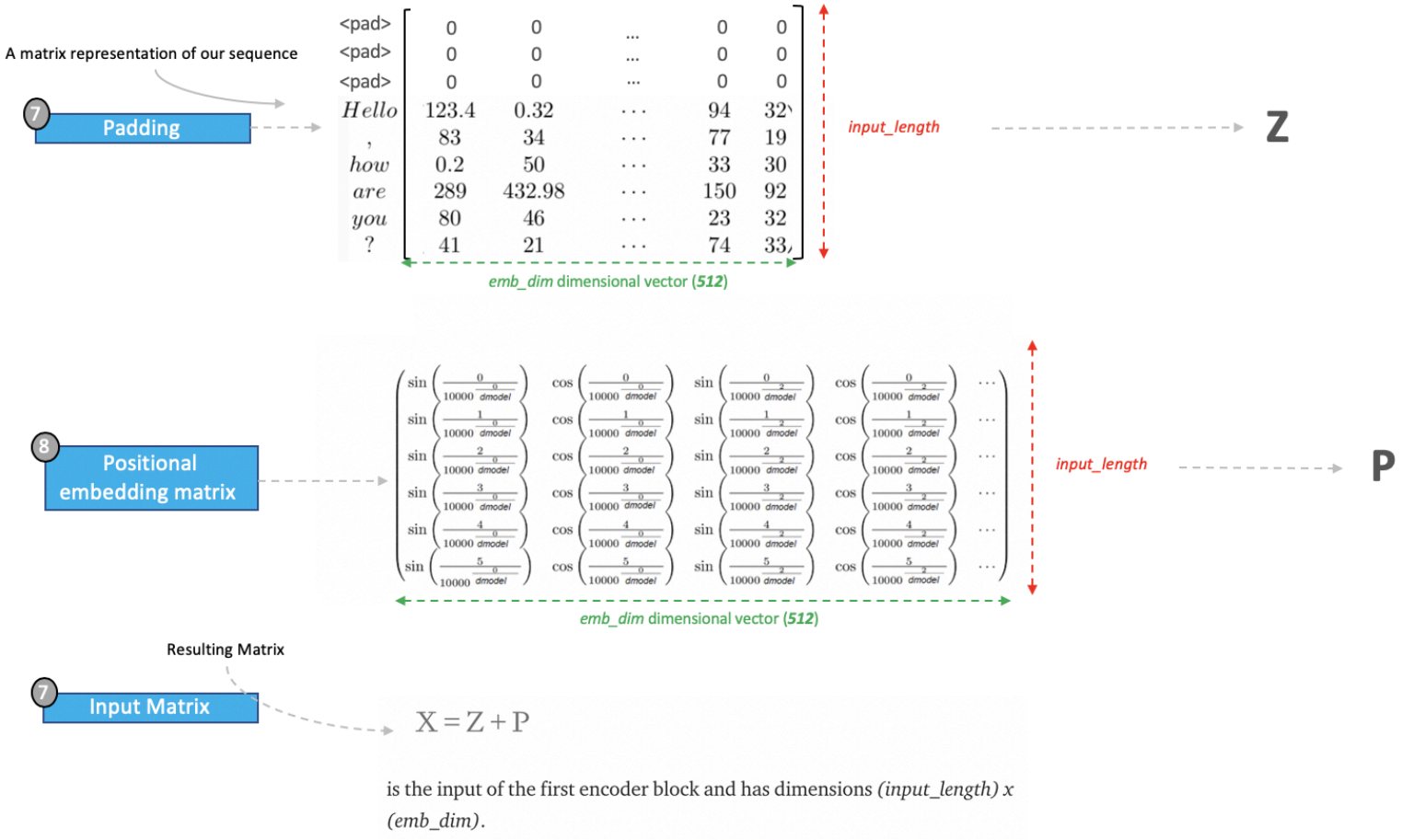
emb_dim dimensional vector (***512***)

6 **Padding** → **512** tokens per sequence (**2** concatenated sentences) will be used and there are **256** sequences per **batch**

if the input_length was set to 9 ----> *["<pad>", "<pad>", "<pad>", "Hello", ",", "how", "are", "you", "?"]*

↓

[5, 5, 5, 34, 90, 15, 684, 55, 193]



Positional embeddings could be understood as the **distance between different words in the sequence**. The intuition here is that adding these values to the embeddings provides **meaningful distances between the embedding vectors once they're projected into Q/K/V vectors and during dot-product attention.**

Problem

The multi-head attention network **cannot** naturally **make use of** the position of the words in the input sequence.
The output of the multi-head attention network would be **the same** for the same sentences in different order.

Sol **Positional Encoding**

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

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

$if\ shape\ of\ enc, dec = [T, d_{model}]$ **pos: position of the word**

$\rightarrow then\ pos \in [0, T),\ i \in [0, d_{model})$ **i: Element i in** d_{model}

Advantage:

PE[pos+k] can be represented as a linear function of **PE[pos]**, so the relative positon between different embeddings can be easily inferred

$$sin(\alpha+\beta) = sin\alpha cos\beta + cos\alpha sin\beta$$

$$cos(\alpha+\beta) = cos\alpha cos\beta - sin\alpha sin\beta$$

$\because$ **Trigonometric Periodicity**

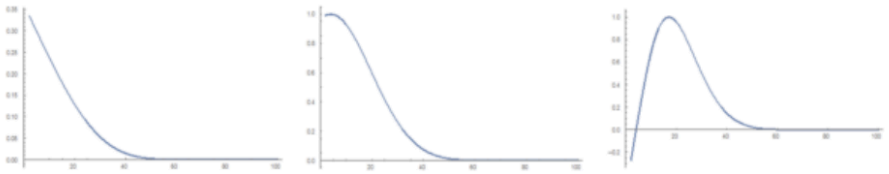
$\therefore$ **Learned the relation between relative and absolute position**

Then Word embedding + positional encoding

(sum, concat...)

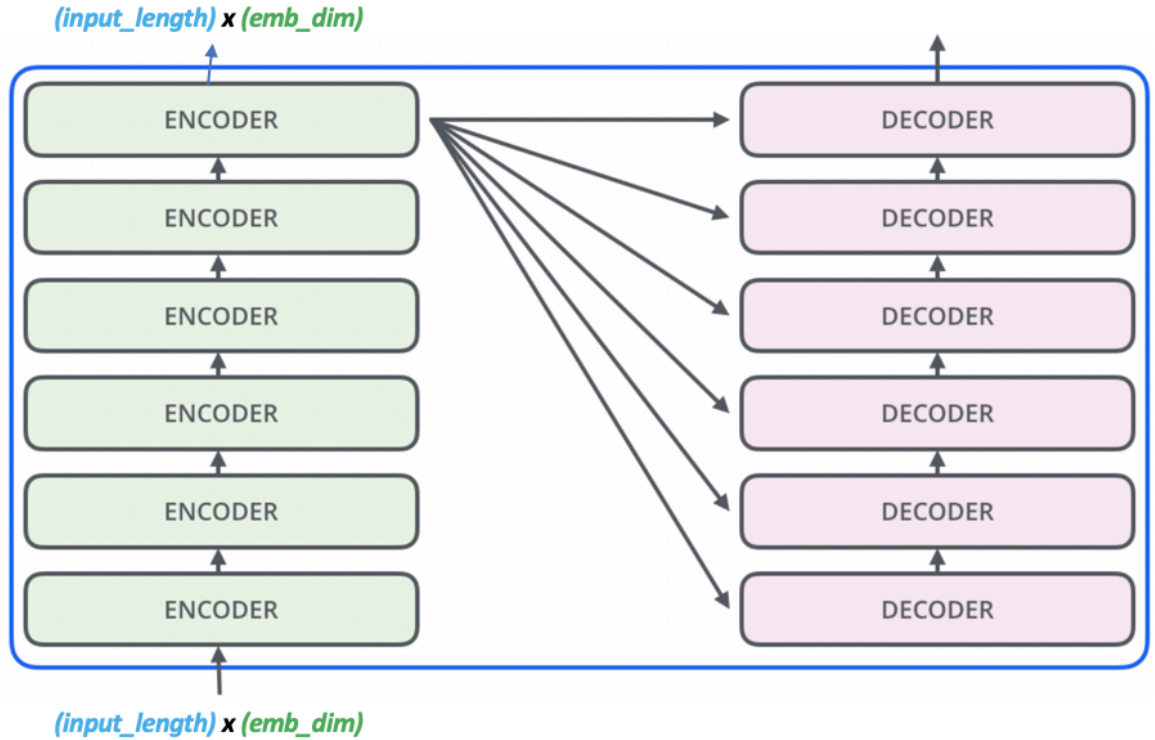
$if\ d_{model} = 512:$

pos=1 pos=5 pos=10



$\longrightarrow$ **Three plots are the same by doing linear transformation**

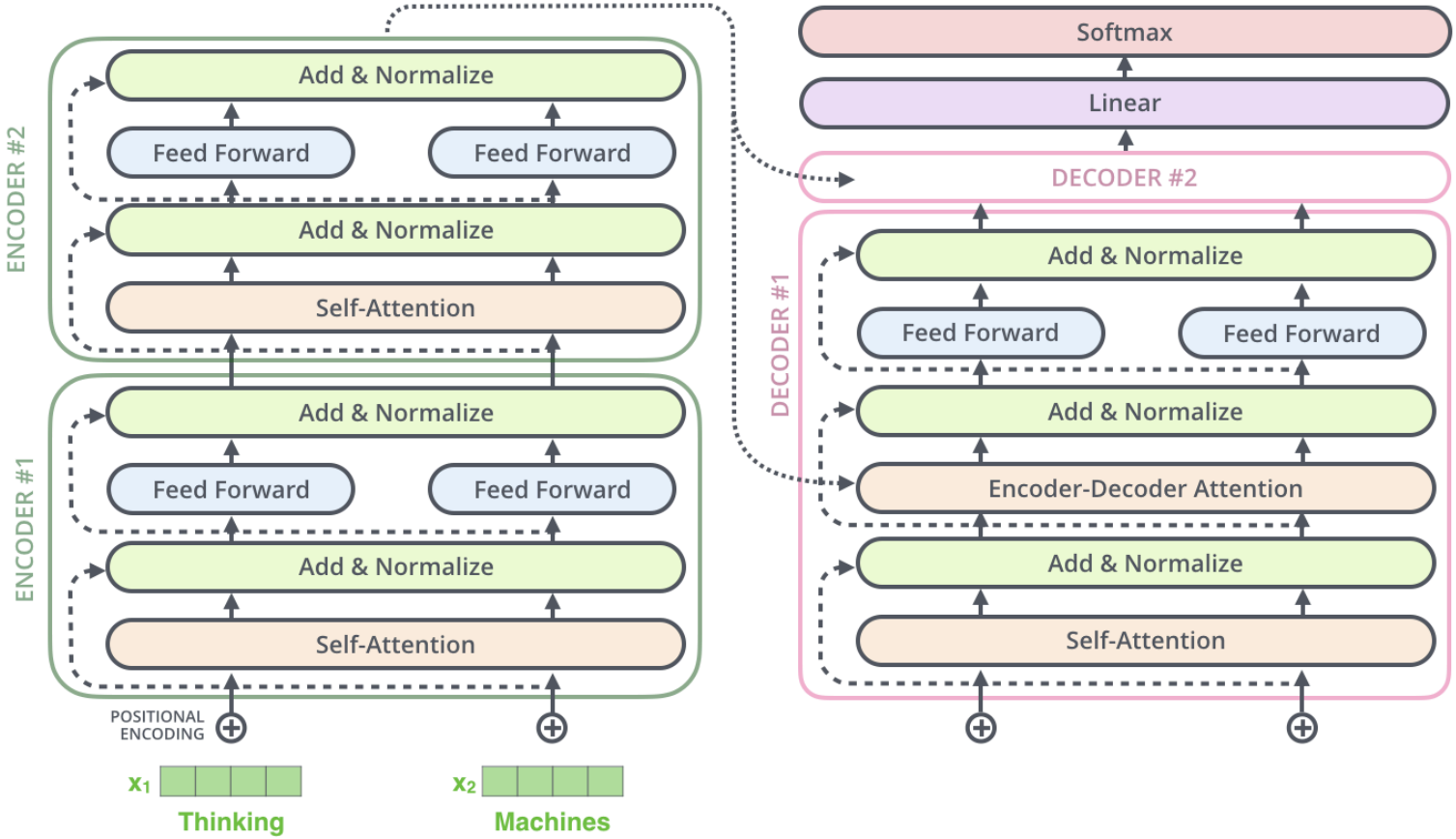
Encoder Blocks



A total of N encoder blocks are chained together to generate the **Encoder's** output.

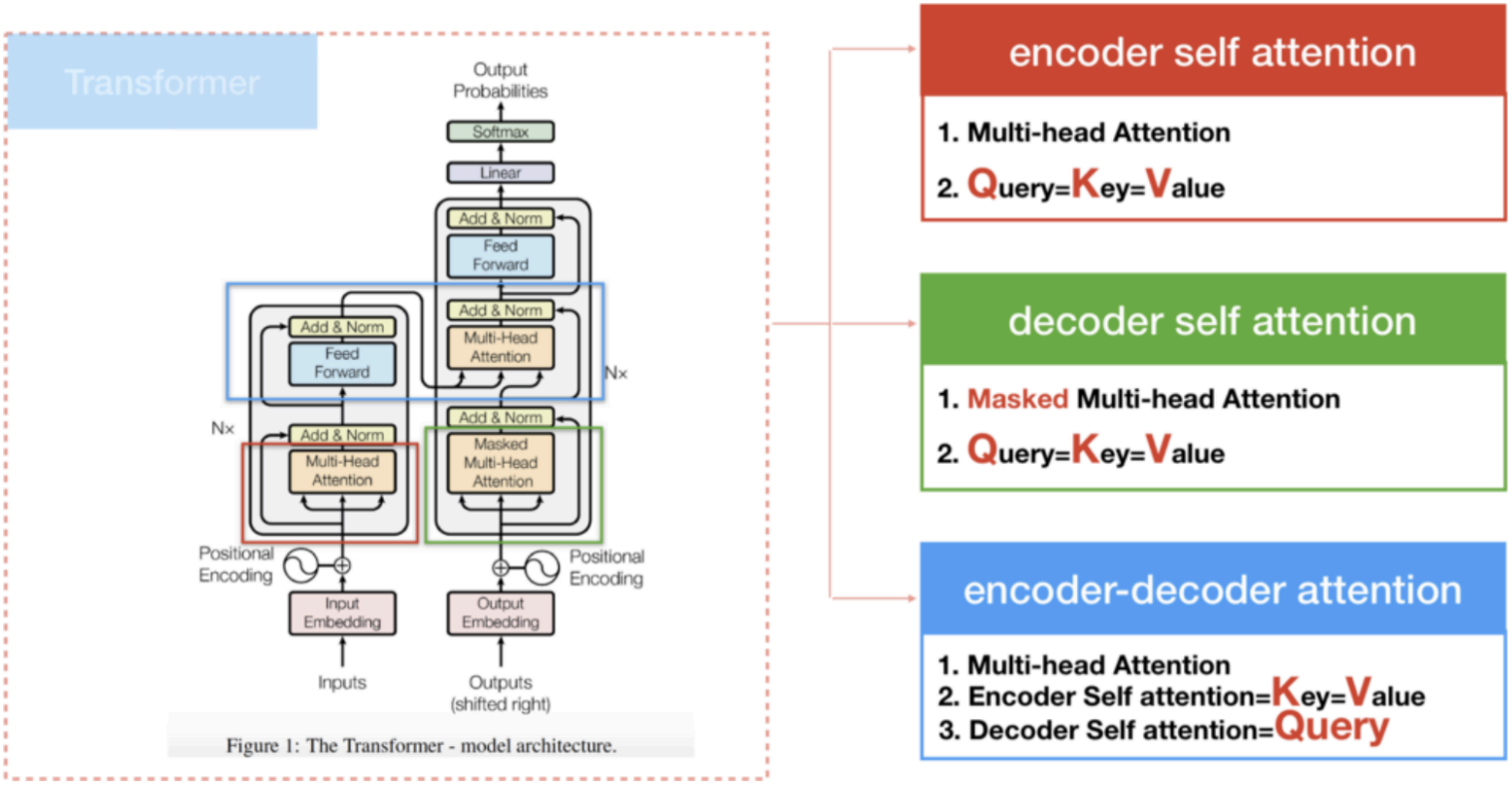
Note: In BERT's experiments, the number of blocks N (or L, as they call it) was chosen to be 12 and 24.

- The dimensions of the **input** and **output** of the encoder block are the same. Hence, it makes sense to **use the output of one encoder block as the input of the next encoder block**.
- A specific block is in charge of **finding relationships between the input representations and encode them** in its output.
- The blocks do not share weights with each other.
- This iterative process through the blocks will help the neural network capture more complex relationships between words in the input sequence.
- The **Transformer** uses **Multi-Head Attention**, which means it computes attention **h different times** with **different weight matrices** and then concatenates the results together.



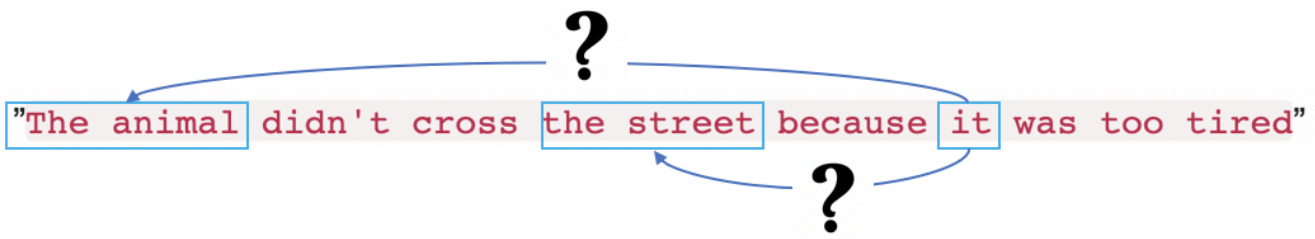
Attention Mechanism

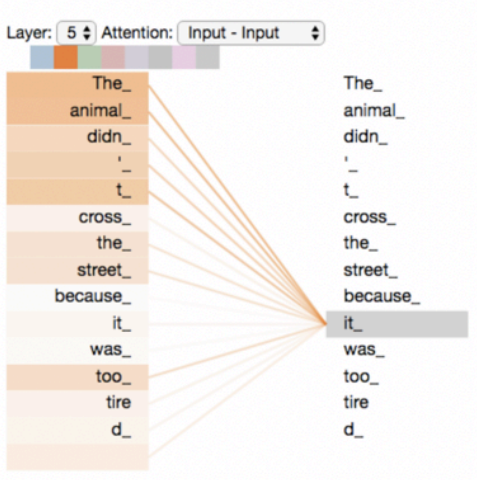
Let's dive into attention mechanism. Note that Multi Head Self Attention is different between **ENCODER block** and **DECODER block**.



A - One Head Self-Attention

A **RNN** maintains a hidden state allows it to incorporate its representation of previous words/vectors it has processed with the current one it's processing. **Self-attention** is the method the Transformer uses to bake the "understanding" of other relevant words into the one we're currently processing.





What does “it” in this sentence refer to? Is it referring to the street or to the animal?

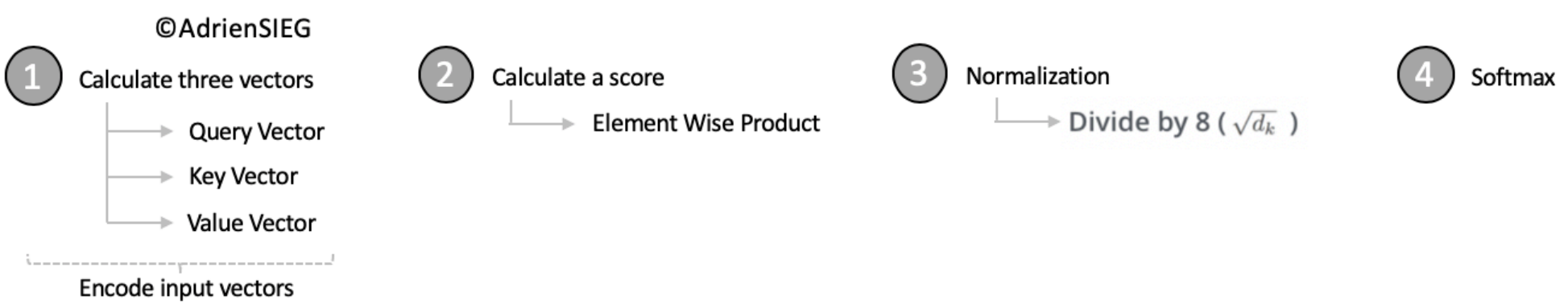
- Create **three vectors** from each of the **encoder’s input vectors**
- For **each word**, we create **a Query vector, a Key vector, and a Value vector
- These vectors are created by **multiplying the embedding** by **three matrices** that we trained during the training process.

Notice that **these new vectors** are **smaller in dimension than the embedding vector**. Their dimensionality is 64, while the embedding and encoder input/output vectors have dimensionality of 512.

Why **dimensionality is 64?** As we must have :

-> Output’s dimension is [length of input sequences] x [dimension of embeddings — 512]

-> We use 8 heads during Multi-head Self-Attention process. The output size of a given self attention vector is [length of input sequences] x [64]. So the concatenated vector resulting from all Multi-head Self-Attention process would be [length of input sequences] x ([64] x [8]) = [length of input sequences] x ([512])



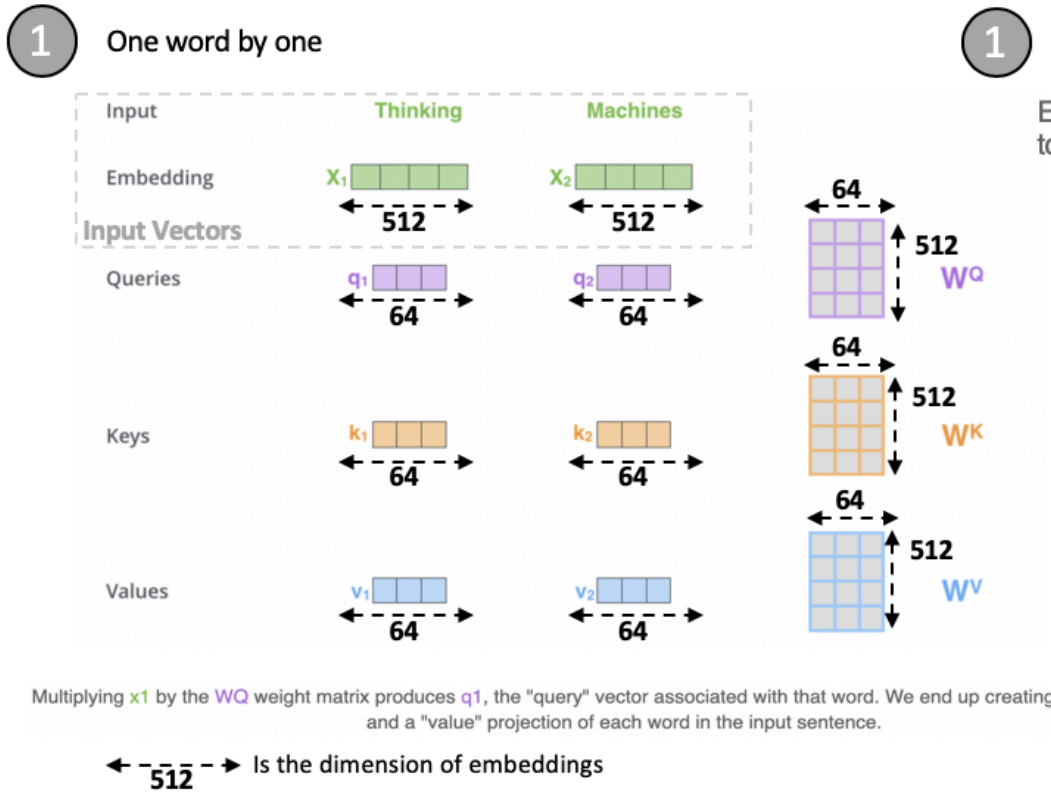
Query q: the query vector q encodes the word/position on the left that is paying attention, i.e. the one that is "querying" the other words. In the example above, the query vector for "the" (the selected word) is highlighted.

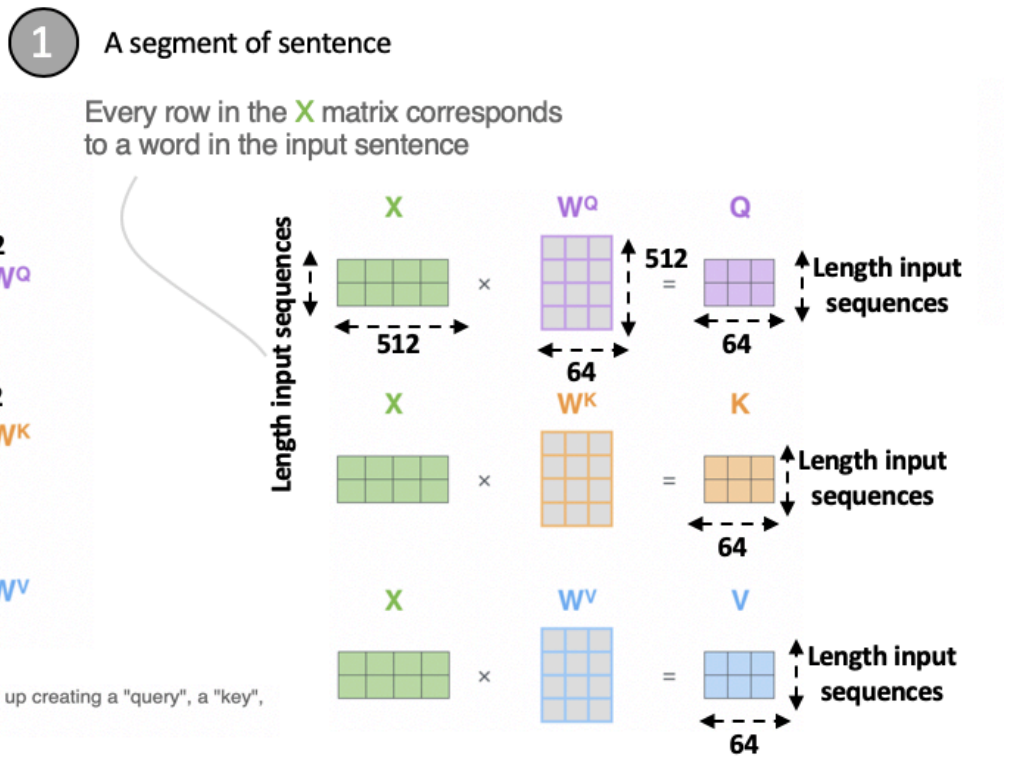
Key k: the key vector k encodes the word on the right to which attention is being paid. The key vector together with the query vector determine the attention score between the respective words, as described below.

q×k (element-wise): the element-wise product of the query vector and a key vector. This product is computed between the selected query vector and each of the key vectors. This is a precursor to the dot product (the sum of the element-wise product) and is included for visualization purposes because it shows how individual elements in the query and key vectors contribute to the dot product.

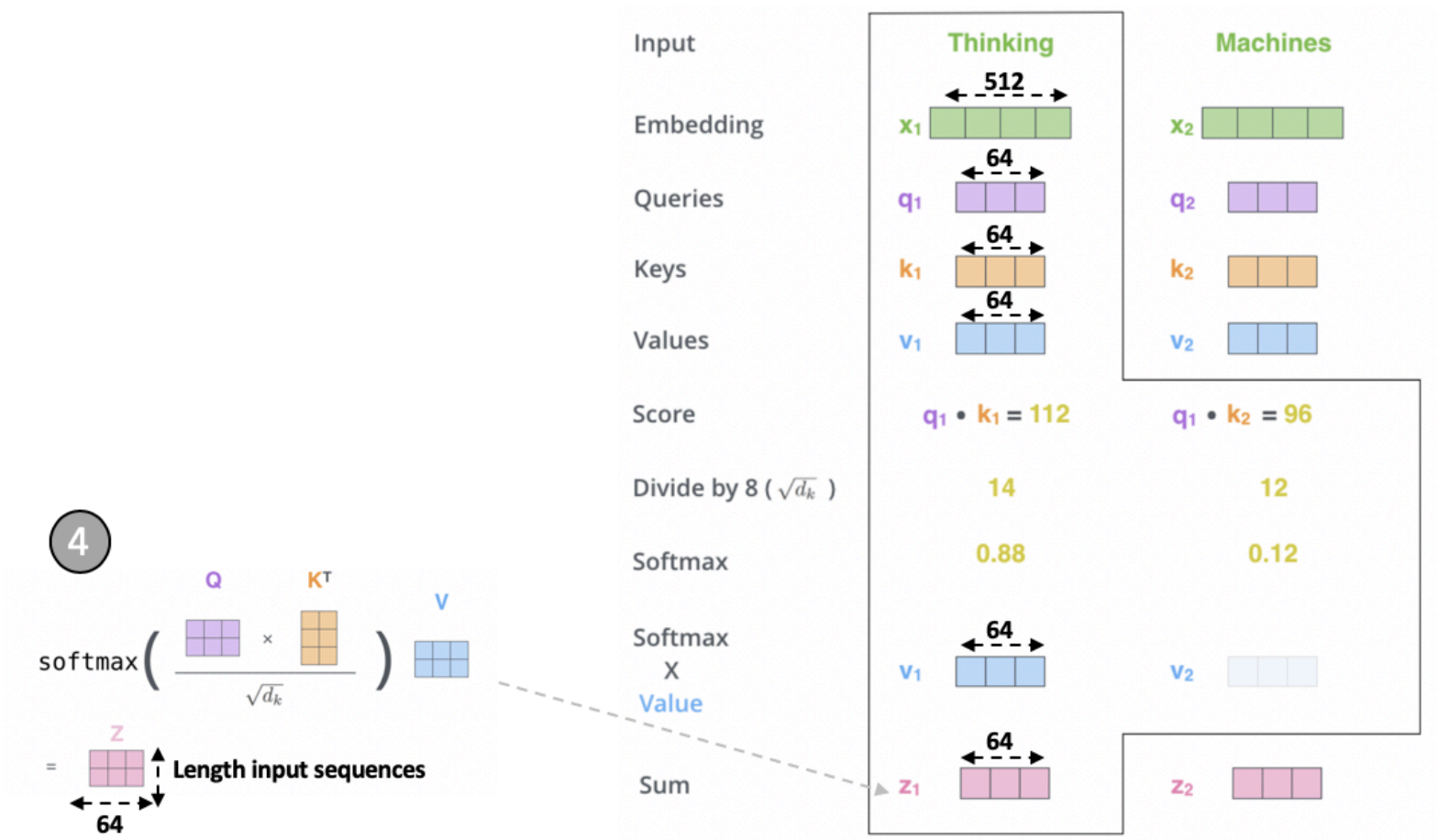
q·k: the dot product of the selected query vector and each of the keyvectors. This is the unnormalized attention score.

Softmax: the softmax of q·k / 8 across all target words. This normalizes the attention scores to be positive and sum to one. The constant factor 8 is the square root of the vector length (64). This softmax score determines how much each word will be expressed at this position. Clearly the word at this position will have the highest softmax score, but sometimes it's useful to attend to another word that is relevant to the current word. How to calculate these three vectors?





We calculate self-attention for every word in the input sequence.



Focus on "Score"

*[Q x Transpose of K] is a product scalar between **Query vector** and **Key Vector**. The closer the **Key Vector** is than the Query Vector, the higher the score — resulting from [Q x Transpose of K]- would be.*

Softmax will provide us a probability distribution which keeps increasing the value for key vectors which are similar to respective query vectors and consequently keep decreasing key vectors which are far away from query vector.

- **d_k** and **d_v** are set such that d_k = d_v = emb_dim/h.

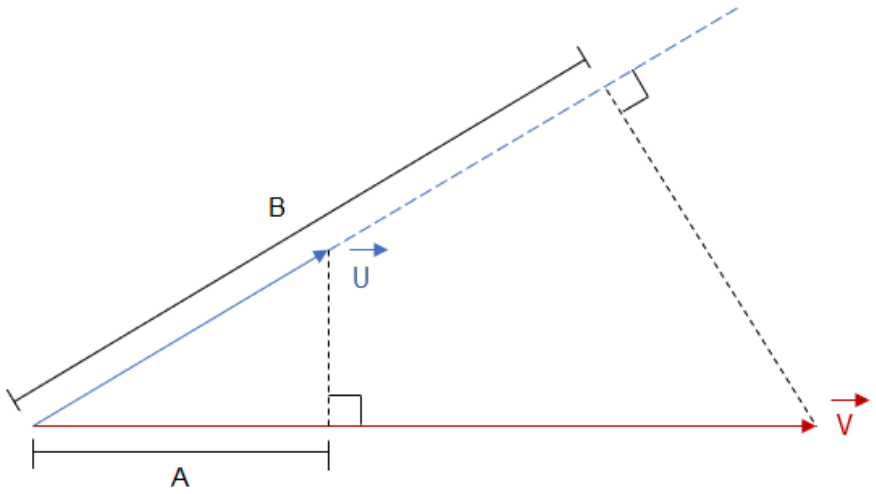
Remember that **Q** and **K** were different projections of the tokens into a **d_k (i.e. 64) dimensional space**. Therefore, we can think about **the dot product of those projections as a measure of similarity between tokens projections**. For every vector projected through Q the dot product with the projections through K measures the similarity between those vectors. If we call v_i and u_j the projections of the i-th token and the j-th token through Q and K respectively, their dot product can be seen as:

$$v_i u_j = \cos(v_i, u_j)||v_i||_2||u_j||_2$$

This is a measure of **how similar are the directions of u_i and v_j** and **how large are their lengths** (the closest the direction and the larger the length, the greater the dot product).

Another way of thinking about this matrix product is as the encoding of a **specific relationship between each of the tokens in the input sequence**(the relationship is defined by the matrices K, Q).

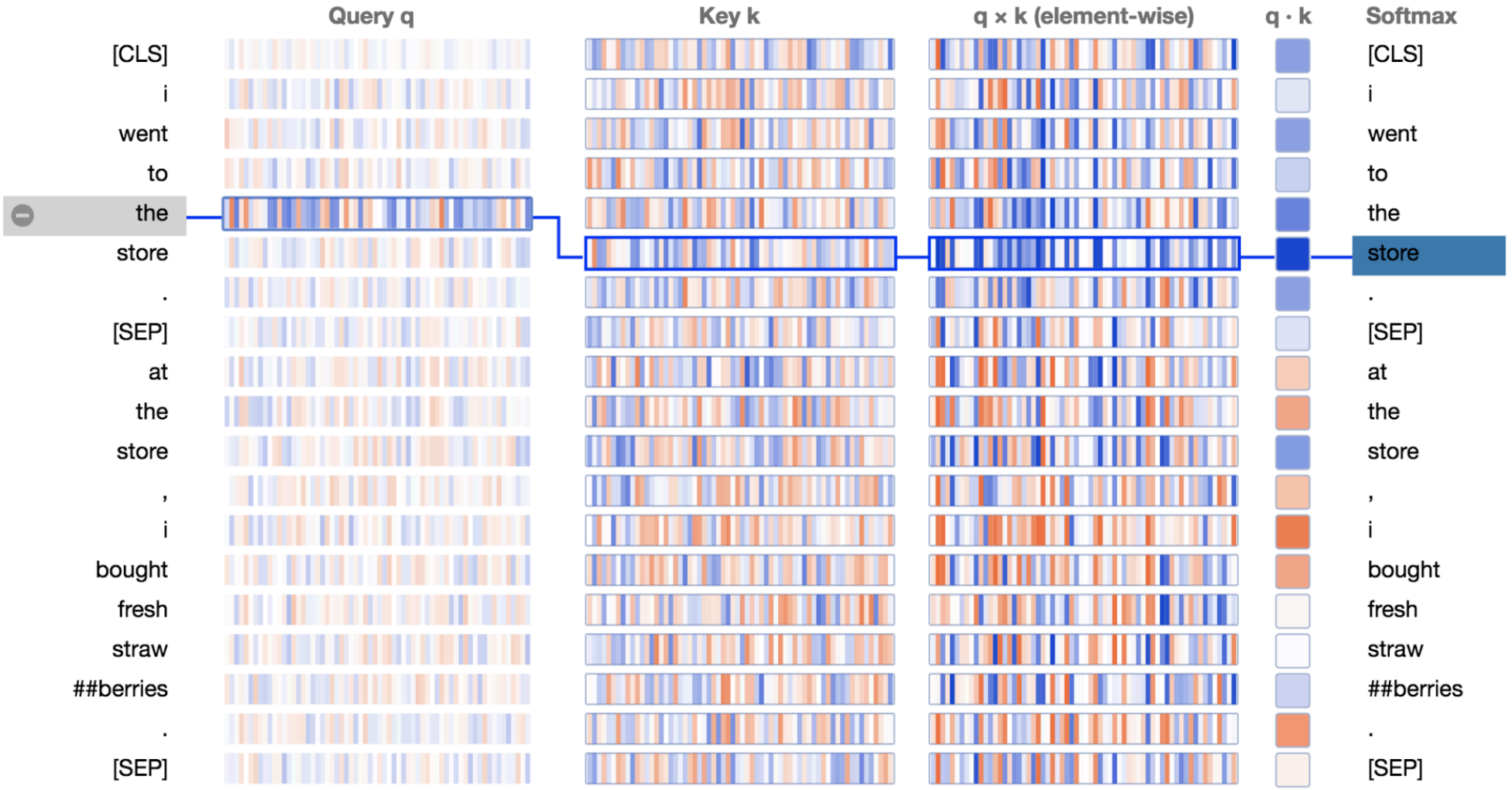
Produit scalaire



Le produit scalaire entre deux vecteurs correspond à la longueur du segment déterminé par la projection orthogonale d'un vecteur sur un autre.
Ici par exemple, la longueur A est le résultat du produit scalaire entre les vecteurs U et V, et la longueur B le résultat du produit scalaire entre les vecteurs V et U.

On s'aperçoit que deux vecteur orthogonaux on un produit scalaire nul.

Le produit scalaire sera d'autant plus grand que les vecteurs se « ressemblent », c'est-à-dire qu'ils ont une longueur proche et que leur direction est similaire.



(https://towardsdatascience.com/deconstructing-bert-part-2-visualizing-the-inner-workings-of-attention-60a16d86b5c1)

We see that the **product of the query vector** for “the” and **the key vector** for “store” (the next word) is strongly positive across most neurons. For tokens other than the next token, the **key-query product contains some combination of positive and negative values**. The result is a high attention score between “the” and “store”.

Lets think back to our original sequence of embeddings, $\mathbf{x}$. We can choose that Q, K, and V are all equal to $\mathbf{x}$. In the paper, this is called "Encoder self-attention" (one of three ways attention is used). Lets say our input tokens are "The man walked the dog". To compute the new vectors for each word, we take the projection of each words embedding onto each of the other embeddings (scaled by $\sqrt{d_k}$), normalize these projections so they sum to 1, and then sum all of the words embeddings multiplied with the corresponding weight. Thus we get a vector who's magnitude can't grow, and that is a linear combination of all of input embeddings. Interestingly, this means the "new" vector for a word, z_i, will give more weight in the sum to the embeddings, x_j, that were similar to its own. To make this a bit more explicit, let's imagine our embeddings $\mathbf{x}$ correspond to the sentence "the man walked the dog". We want to compute a new vector the first word, "the". To do so, we compute the similarity between "the"'s embedding vector, and the embedding vectors for "the", "man", "walked", "the", "dog". These similarities are normalised so they sum to 1 (and the two "the" tokens will contribute the maximum similarity score, $\frac{1}{\sqrt{d_k}}$). The new vector is therefore:

$$
\begin{aligned}
z_{\text{the}} = &\ \text{attention\_weight}(\text{“the”}, \text{“the”}) \times \text{embedding}(\text{“the”}) \\
&+ \text{attention\_weight}(\text{“the”}, \text{“man”}) \times \text{embedding}(\text{“man”}) \\
&+ \text{attention\_weight}(\text{“the”}, \text{“walked”}) \times \text{embedding}(\text{“walked”}) \\
&+ \dots
\end{aligned}
$$

Example: Consider this phrase — "Action gets results". To calculate the self-attention for the first word "Action", we will **calculate scores for all the words in the phrase with respect to "Action". This score determines the importance of other words when we are encoding a certain word in an input sequence.

1 The score for the first word is calculated by taking the dot product of the Query vector (q1) with the keys vectors (k1, k2, k3) of all the words:

Word	q vector	k vector	v vector	score
Action	q_1	k_1	v_1	$q_1 . k_1$
gets		k_2	v_2	$q_1 . k_2$
results		k_3	v_3	$q_1 . k_3$

2 Then, these scores are divided by 8 which is the square root of the dimension of the key vector:

Word	q vector	k vector	v vector	score	score / 8
Action	q_1	k_1	v_1	$q_1 . k_1$	$q_1 . k_1 / 8$
gets		k_2	v_2	$q_1 . k_2$	$q_1 . k_2 / 8$
results		k_3	v_3	$q_1 . k_3$	$q_1 . k_3 / 8$

3 Next, these scores are normalized using the softmax activation function:

Word	q vector	k vector	v vector	score	score / 8	Softmax
Action	q_1	k_1	v_1	$q_1 . k_1$	$q_1 . k_1 / 8$	x_{11}
gets		k_2	v_2	$q_1 . k_2$	$q_1 . k_2 / 8$	x_{12}
results		k_3	v_3	$q_1 . k_3$	$q_1 . k_3 / 8$	x_{13}

4 These normalized scores are then multiplied by the value vectors (v1, v2, v3) and sum up the resultant vectors to arrive at the final vector (z1). This is the output of the self-attention layer. It is then passed on to the feed-forward network as input:

Word	q vector	k vector	v vector	score	score / 8	Softmax	Softmax * v	Sum
Action	q_1	k_1	v_1	$q_1 . k_1$	$q_1 . k_1 / 8$	x_{11}	$x_{11} * v_1$	z_1
gets		k_2	v_2	$q_1 . k_2$	$q_1 . k_2 / 8$	x_{12}	$x_{12} * v_2$	
results		k_3	v_3	$q_1 . k_3$	$q_1 . k_3 / 8$	x_{13}	$x_{13} * v_3$	

So, z1 is the self-attention vector for the first word of the input sequence “Action gets results”. We can get the vectors for the rest of the words in the input sequence in the same fashion:

Word	q vector	k vector	v vector	score	score / 8	Softmax	Softmax * v	Sum#
Action		k_1	v_1	$q_2 . k_1$	$q_2 . k_1 / 8$	x_{21}	$x_{21} * v_1$	
gets	q_2	k_2	v_2	$q_2 . k_2$	$q_2 . k_2 / 8$	x_{22}	$x_{22} * v_2$	z_2
results		k_3	v_3	$q_2 . k_3$	$q_2 . k_3 / 8$	x_{23}	$x_{23} * v_3$	

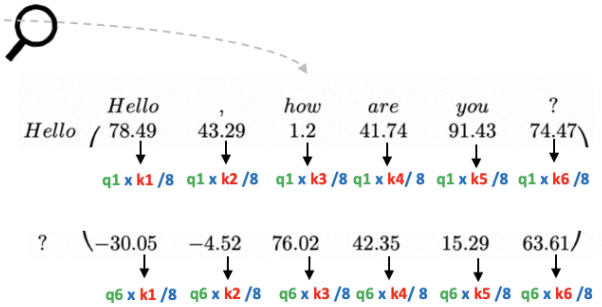
Word	q vector	k vector	v vector	score	score / 8	Softmax	Softmax * v	Sum#
Action		k_1	v_1	$q_3 . k_1$	$q_3 . k_1 / 8$	x_{31}	$x_{31} * v_1$	
gets		k_2	v_2	$q_3 . k_2$	$q_3 . k_2 / 8$	x_{32}	$x_{32} * v_2$	
results	q_3	k_3	v_3	$q_3 . k_3$	$q_3 . k_3 / 8$	x_{33}	$x_{33} * v_3$	z_3

Let's go back with our initial example:

7 **Score**

$$\frac{Q_i K_i^T}{\sqrt{d_k}}$$

	Hello	,	*how*	*are*	*you*	?
Hello	78.49	43.29	1.2	41.74	91.43	74.47
,	95.84	28.78	57.13	68.20	−60.94	26.85
how	−95.69	−52.16	17.00	45.71	48.49	64.35
are	−69.92	85.16	94.94	91.04	−92.83	77.49
you	65.85	55.85	62.54	−97.46	76.38	13.20
?	−30.05	−4.52	76.02	42.35	15.29	63.61



8 **Softmax**

$$Softmax\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right)$$

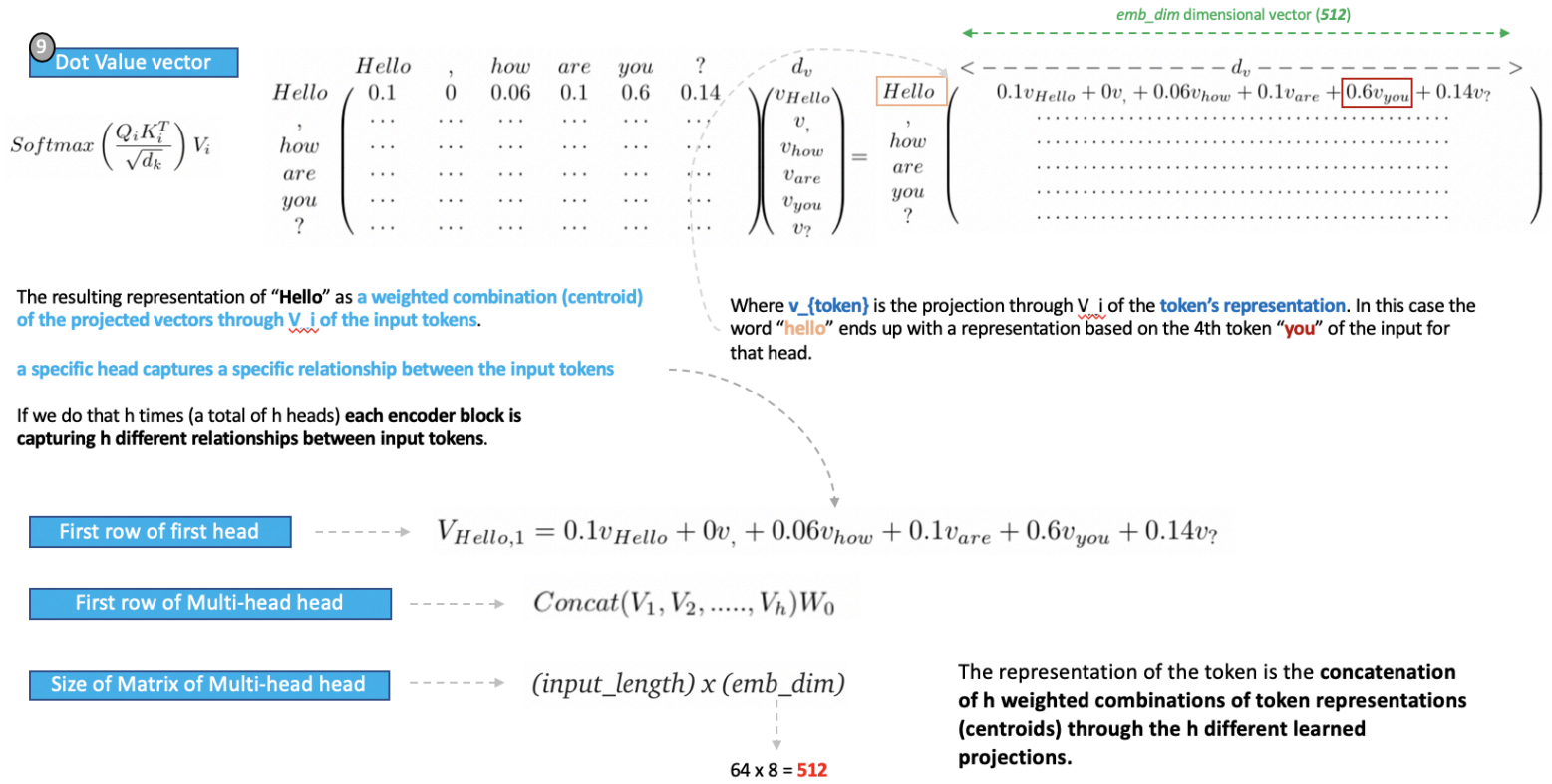
	Hello	,	*how*	*are*	*you*	?	
Hello	$72.40 * 10^{-06}$	$1.23 * 10^{-21}$	$6.51 * 10^{-40}$	$2.62 * 10^{-22}$	$9.99 * 10^{-01}$	$4.30 * 10^{-08}$	= 1
,	$1.00 * 10^{+00}$	$7.51 * 10^{-30}$	$1.54 * 10^{-17}$	$9.91 * 10^{-13}$	$8.15 * 10^{-69}$	$1.09 * 10^{-30}$	= 1
how	$3.12 * 10^{-70}$	$2.51 * 10^{-51}$	$2.72 * 10^{-21}$	$8.03 * 10^{-09}$	$1.29 * 10^{-07}$	$9.99 * 10^{-01}$	= 1
are	$2.47 * 10^{-72}$	$5.54 * 10^{-05}$	$9.80 * 10^{-01}$	$1.98 * 10^{-02}$	$2.77 * 10^{-82}$	$2.58 * 10^{-08}$	= 1
you	$2.67 * 10^{-05}$	$1.21 * 10^{-09}$	$9.75 * 10^{-07}$	$3.17 * 10^{-76}$	$9.99 * 10^{-01}$	$3.64 * 10^{-28}$	= 1
?	$8.59 * 10^{-47}$	$1.05 * 10^{-35}$	$9.99 * 10^{-01}$	$2.38 * 10^{-15}$	$4.21 * 10^{-27}$	$4.07 * 10^{-06}$	= 1

The network will learn over training time which relationships are more useful and will relate tokens to each other based on these relationships

UNDERSTANDING

	Hello	,	*how*	*are*	*you*	?	
Hello	0.1	0	0.06	0.1	0.6	0.14	= 1
,	…	…	…	…	…	…	= 1
how	…	…	…	…	…	…	= 1
are	…	…	…	…	…	…	= 1
you	…	…	…	…	…	…	= 1
?	…	…	…	…	…	…	= 1

For the sake of understanding let's propose **a dummy simplification** by simplifying the previous matrix



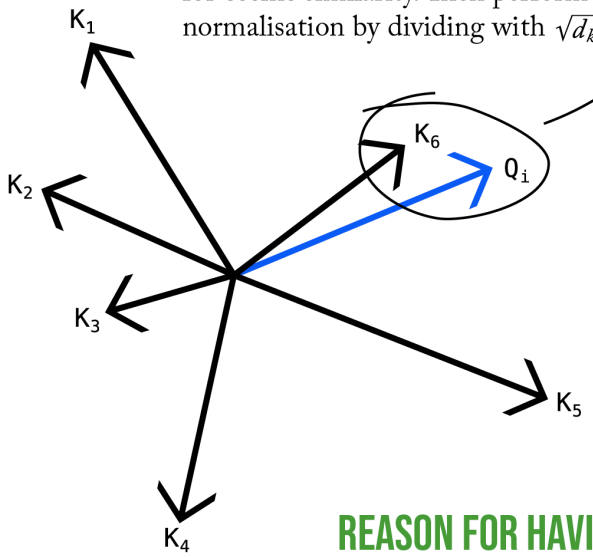
To sum up when it comes to One-Head Self Attention

The main idea behind attention is **lookup-table**, a table that has a large number of values for some other values and **you ask it a query** and **it returns one closest to it**. In the method used here, we feed it three values, key, value and query. There are large number of keys, basically 1-dimensional vectors in n-dimensional space, where each key has some corresponding value.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

1) Each *Query* vector performs similarity check with all the keys. QK^T is the formula for cosine similarity. Then perform normalisation by dividing with $\sqrt{d_k}$

Key	Value
—	—



2) Performing softmax over this value amplifies the larger values and supresses smaller values. Multiplying it with corresponding Values makes similar ones larger

REASON FOR HAVING THE ATTENTION FORMULA AS IT IS

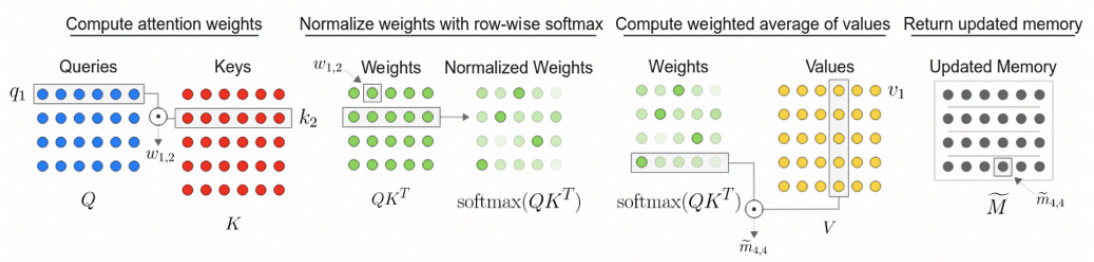
What is Attention?

- $Attention(query, Key, Value) = \underbrace{Softmax(query \cdot Key^T)}_{\text{Attention Weight}} \cdot Value$

- $Attention(query, Key, Value) = Softmax(query \cdot Key^T) \cdot Value$

Search keys which are similar to a query, and return the corresponding values.

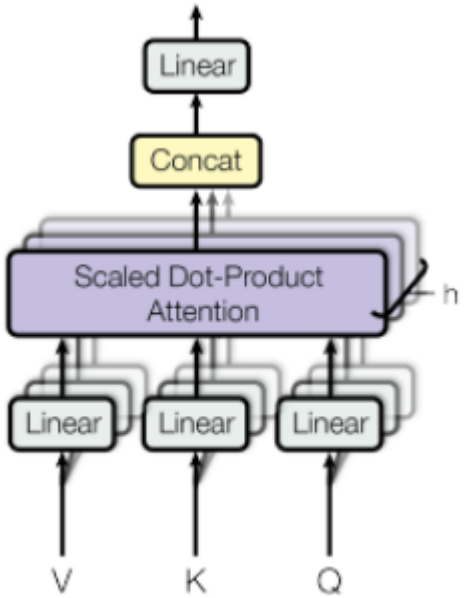
An attention function can be described as a **dictionary** object.



B-Muti-Head Self-Attention

The paper notes that **“additive attention”** performs better than the self attention described above, though it is much slower. Additive attention uses a more complicated compatibility function — namely a feed forward neural network.

Self-attention is computed not once but multiple times in the Transformer’s architecture, in parallel and independently. It is therefore referred to as Multi-head Attention. The outputs are concatenated and linearly transformed as shown in the figure below:

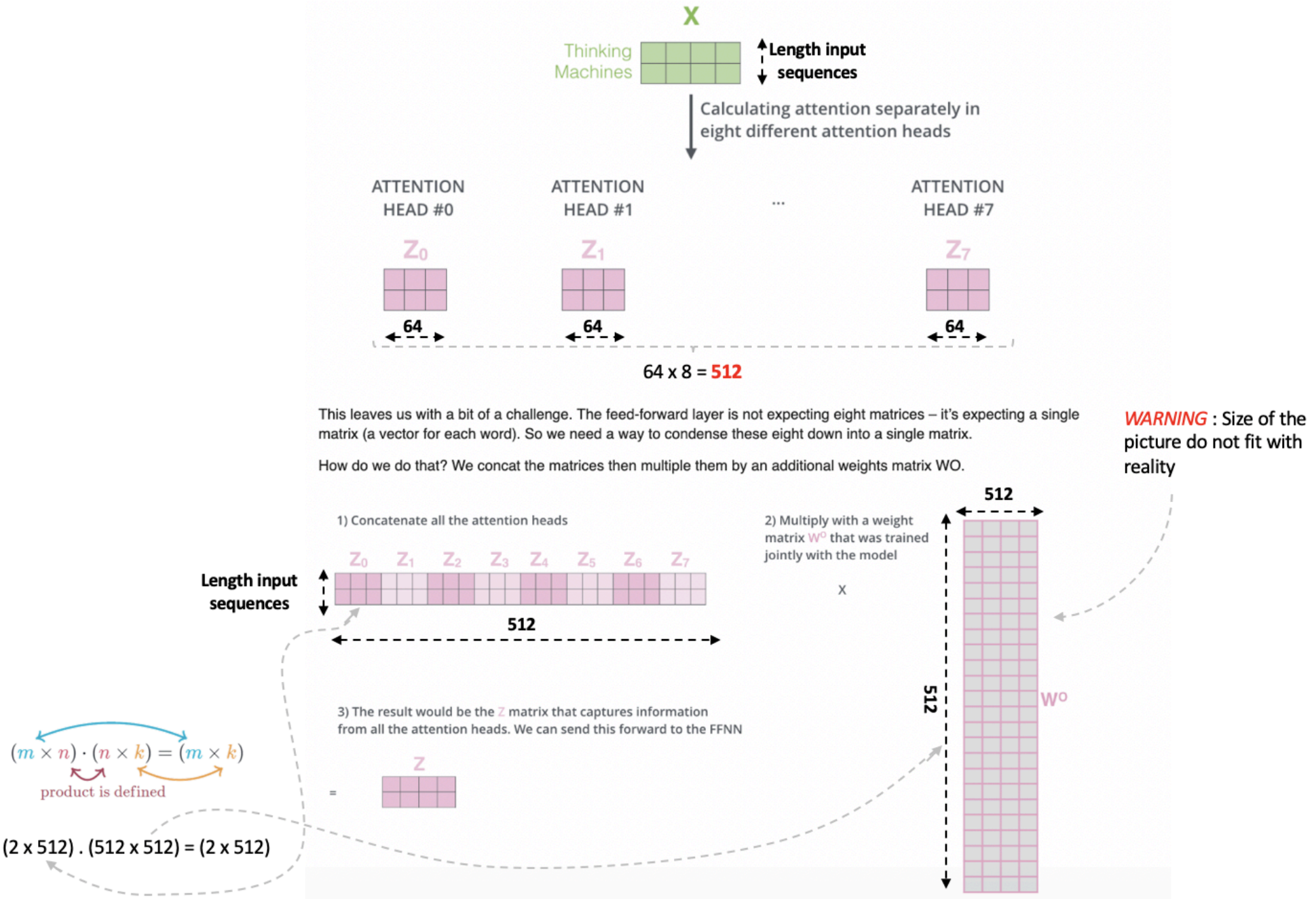


Multi-Head Attention

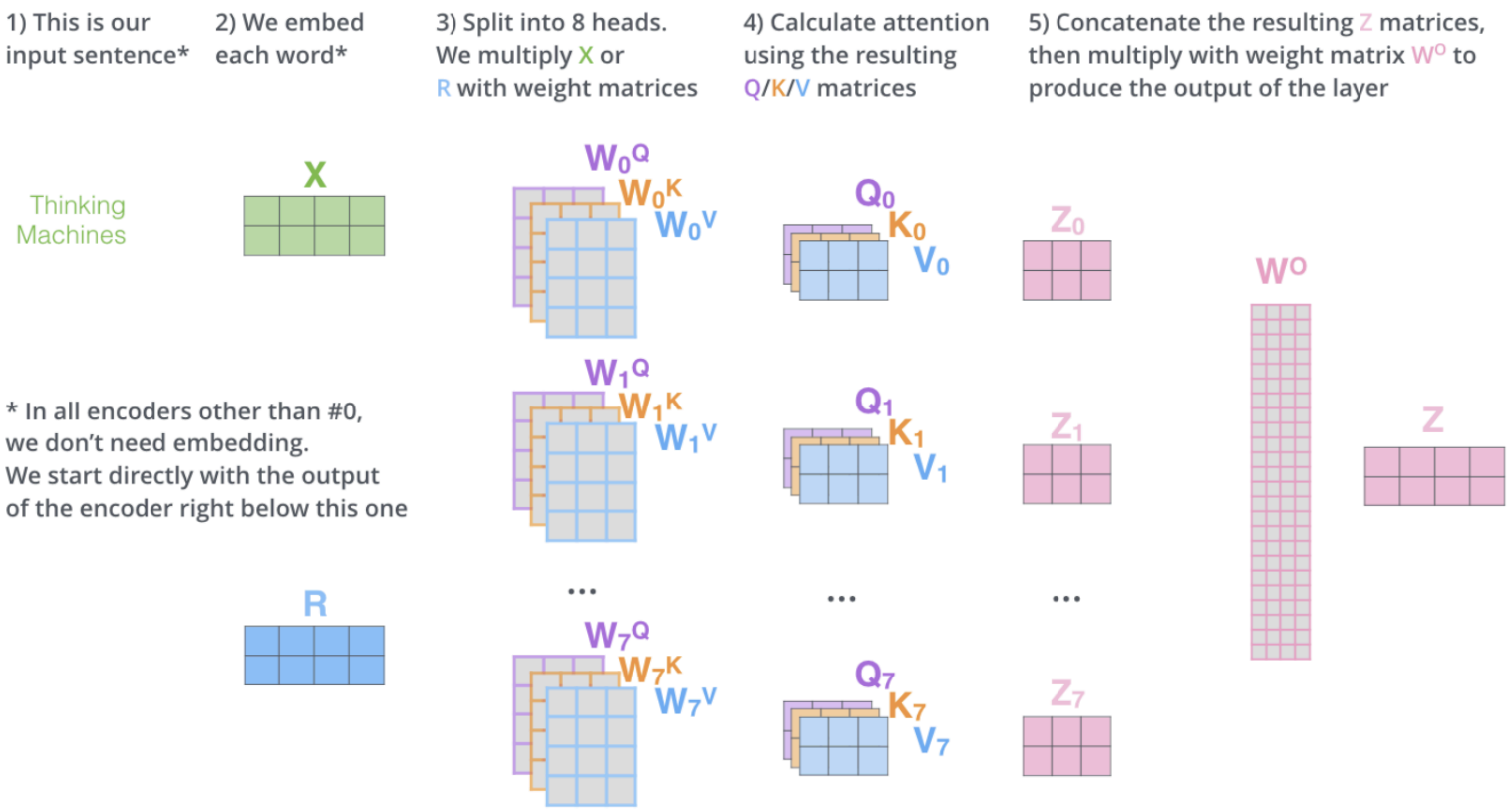
The Transformer uses **eight attention heads**, so we end up with **eight sets for each encoder/decoder**. Each set is used to **project the input embeddings into a different representation subspace**. If we do the same self-attention calculation we described just above, we end up with eight different Z matrices.

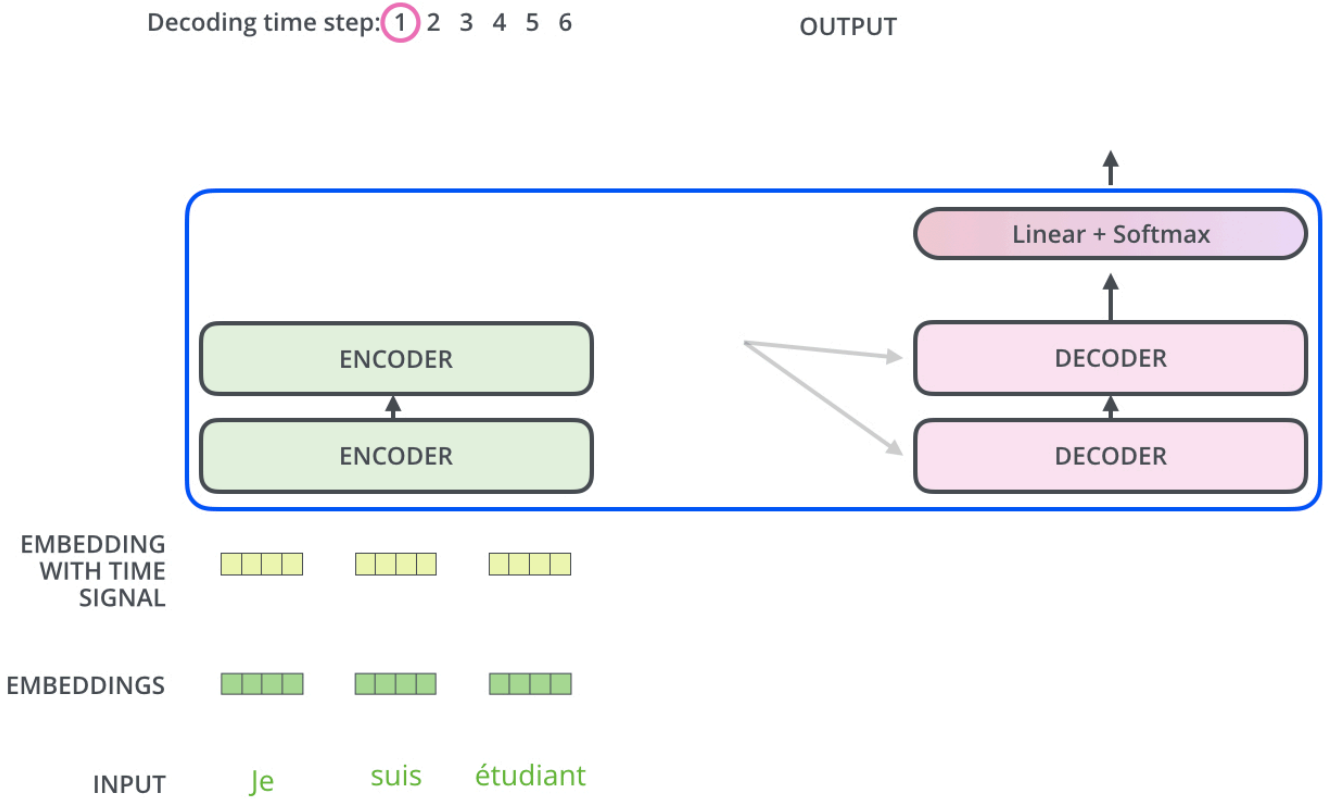
However, **the feed-forward layer is not expecting eight matrices**. We need to concatenate them and condense these eight down into a single matrix by multiply them with an additional weights matrix WO

How to return 8 matrices Z1...Z8 into a singe matrix Z in Multi-head attention?



To sum up...


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
X
R
* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one
W0Q
W0K
W0V
W1Q
W1K
W1V
W7Q
W7K
W7V
Q0
K0
V0
Q1
K1
V1
Q7
K7
V7
Z0
Z1
Z7
WO
Z



Here is the result :

Dropout, Add & Norm

https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1184/lectures/lecture12.pdf

Before this layer, there is always a layer for which inputs and outputs have the same dimensions (*Multi-Head Attention or Feed-Forward*). We will call that layer *Sublayer* and its input x.

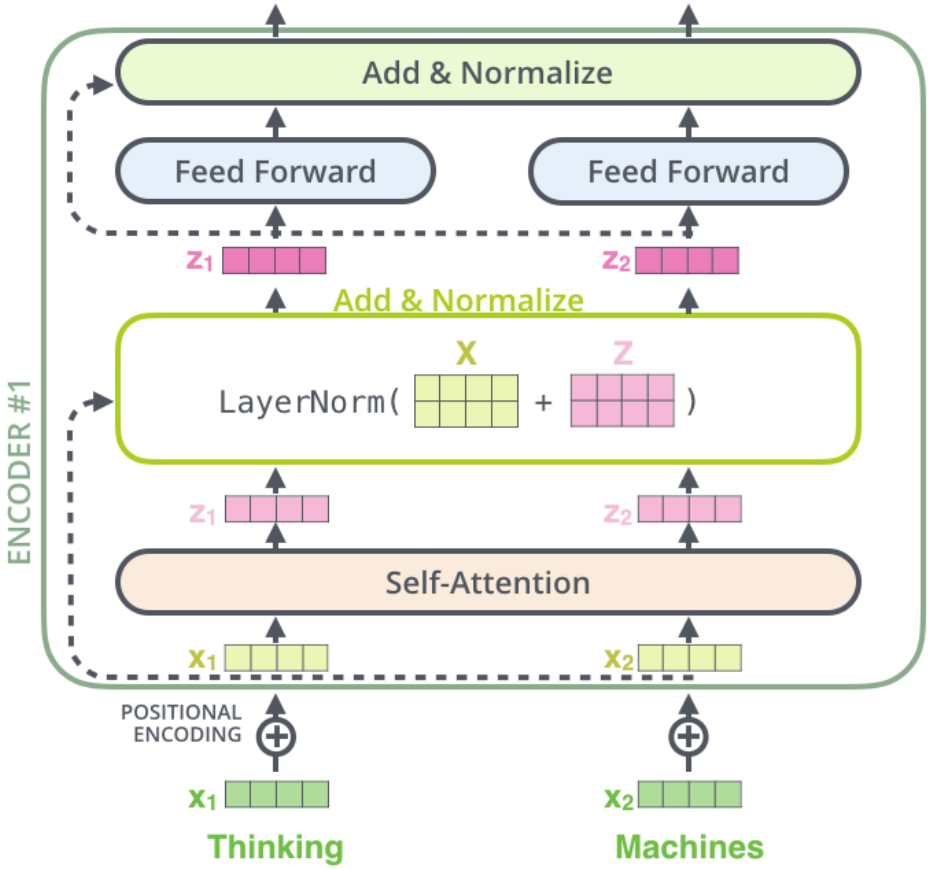
After each *Sublayer*, dropout is applied with 10% probability. Call this result *Dropout(Sublayer(x))*. This result is added to the Sublayer's input x, and we get x + *Dropout(Sublayer(x))*.

$$LayerNorm(x + Dropout(Sublayer(x)))$$

Observe that in the context of a *Multi-Head Attention* layer, this means **adding the original representation of a token x to the representation based on the relationship with other tokens**. It is like telling the token:

"Learn the relationship with the rest of the tokens, but don't forget what we already learned about yourself!"

Finally, a token-wise/row-wise normalization is computed with the mean and standard deviation of each row. This improves the stability of the network.



Layernorm changes input to have mean 0 and variance 1,
per layer and per training point (and adds two more parameters)

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^l - \mu^l)^2} \qquad h_i = f\left(\frac{g_i}{\sigma_i}(a_i - \mu_i) + b_i\right)$$

We compute **the mean and variance used for normalization** from all of the summed inputs to the neurons in a layer on a single training case.

Position-wise-Feed-Forward Network

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.

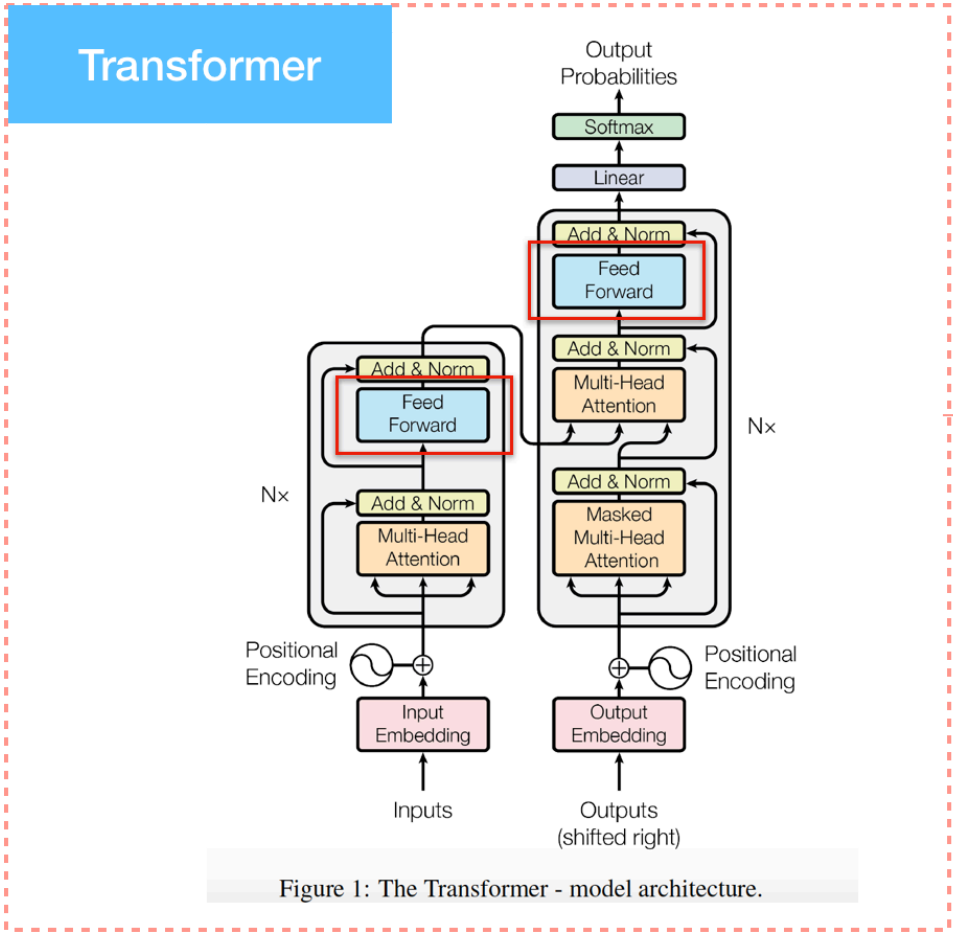


Figure 1: The Transformer - model architecture.

Formula

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Two linear Transformations + Relu

= Two convolution with kernel size 1 $(512 \rightarrow 2048 \rightarrow 512)$

Detail

1. **Doing FFN for each position in input sequence** $x_i \sim x_n$ ⟶ **position-wise**
2. **kernel size =1** ⟶ **So each position is calculated independently.**

   $(if\ kernel\ size \neq 1,\ then\ position\ dependency)$
3. **Why** $(512 \rightarrow 2048 \rightarrow 512)$ **?**

   reduce the number of feature maps **(Kaiming He. Resnet bottleneck)**

(https://mc.ai/seq2seq-pay-attention-to-self-attention-part-2/)

While the linear transformations are the same across different positions, they use different parameters from layer to layer. Another way of describing this is as two convolutions with kernel size 1. The dimensionality of input and output is dmodel=512, and the inner-layer has dimensionality dff=2048.

3. Decoder Block

Each **decoder layer** consists of sublayers:

1. Masked multi-head attention (with **look ahead mask and padding mask**)
2. Multi-head attention (with padding mask). **V (value) and K (key) receive the encoder output as inputs. Q (query) receives the output from *the masked multi-head attention sublayer.**
3. Point wise feed forward networks

Each of these sublayers has a residual connection around it followed by a layer normalization. The output of each sublayer is LayerNorm(x + Sublayer(x)).

There are N decoder layers in the transformer.

As **Q** receives **the output from decoder's first attention block**, and **K** receives the **encoder output**, **the attention weights represent the importance given to the decoder's input based on the encoder's output**. In other words, **the decoder predicts the next word by looking at the encoder output and self-attending to its own output**.

Multi Head Masked Self Attention

In *encoder*, self-attention layers process input **queries, keys and values** that comes from the **output of previous layer**. Each position in encoder can attend to all positions from previous layer of the encoder.

In *decoder*, self-attention layer enable **each position to attend to all previous positions in the decoder**, including the current position.

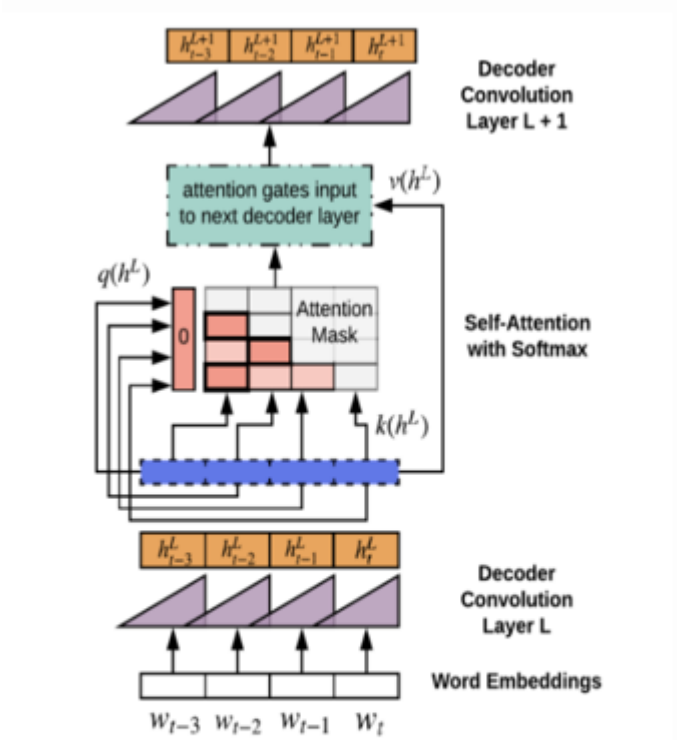
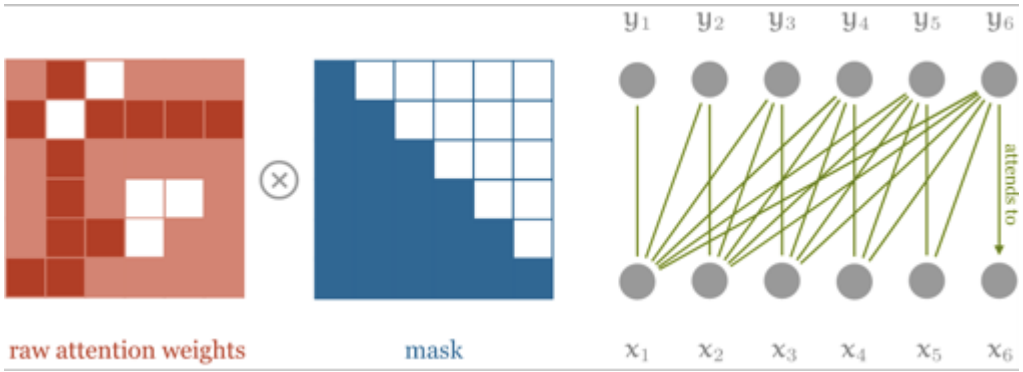
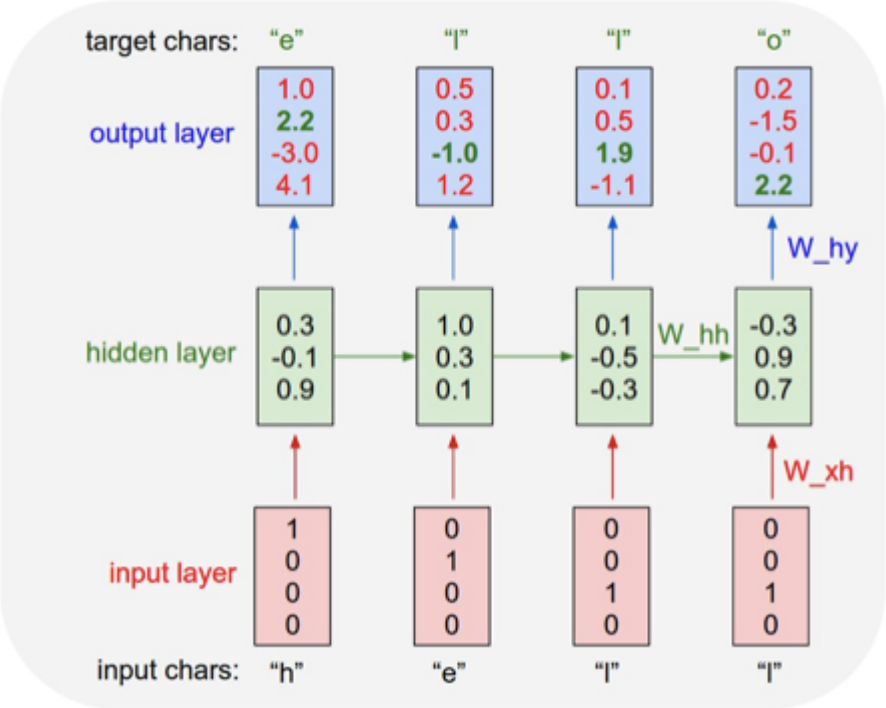


Figure 3: Multihead self-attention mechanism. The decoder layer depicted attends with itself to gate the input of the subsequent decoder layer.

(https://persagen.com/resources/biokdd-review-nlu.html)



In other words, the self-attention layer is only allowed to attend to **earlier positions in the output sequence**. Masking multi-head attention is done by masking future positions (setting them to -∞) before the softmax step in the self-attention calculation. **This step ensures that the predictions for position i can depend only on the known outputs at positions less than* i*. Since we want these elements to be zero after the softmax, we set them to −∞.



With RNNs — there is no issue like that, since **they cannot look forward into the input sequence: output i depends only on inputs 0 to i**. With a transformer, the output depends on the entire input sequence, so prediction of the next words/characters becomes vacuously easy, just retrieve it from the input.

To use self-attention as an autoregressive model, we'll need to ensure that it **cannot look forward into the sequence**. We do this by **applying a mask to the matrix of dot products**, before the softmax is applied. **This mask disables all elements above the diagonal of the matrix**.

After we've **handicapped the self-attention module like this**, the model **can no longer look forward in the sequence**.

The "**Decoder Attention**" layer works just like multiheaded self-attention, except it creates its **Queries matrix** from the **layer below it**, and **takes the Keys and Values matrix from the output of the encoder stack**.

Traduction

Français *"tu t'appelles lebowski lebowski"*

vers

Anglais *"your name's lebowski lebowski"*

→ Au moment de générer un nouveau mot, la matrice générée par $\text{softmax}(\frac{QK^T}{\sqrt{d_k}})$ dans la couche d'auto-attention du décodeur devrait, sans masquage, ressembler à ça :

	Your name	’s	lebowski	lebowski
Your	-26,845	-13,4225	34,749	-10,0113	-2,00227
name	0,093	-0,00693	0,006189	-0,0062	0,003095
’s	3,435	-495,767	3,74	-53,4913	-17282
lebowski	11,964	-0,02413	8,7227	-0,00061	3,5108
lebowski	-1,526	63,23468	2,950	55,15364	1,579

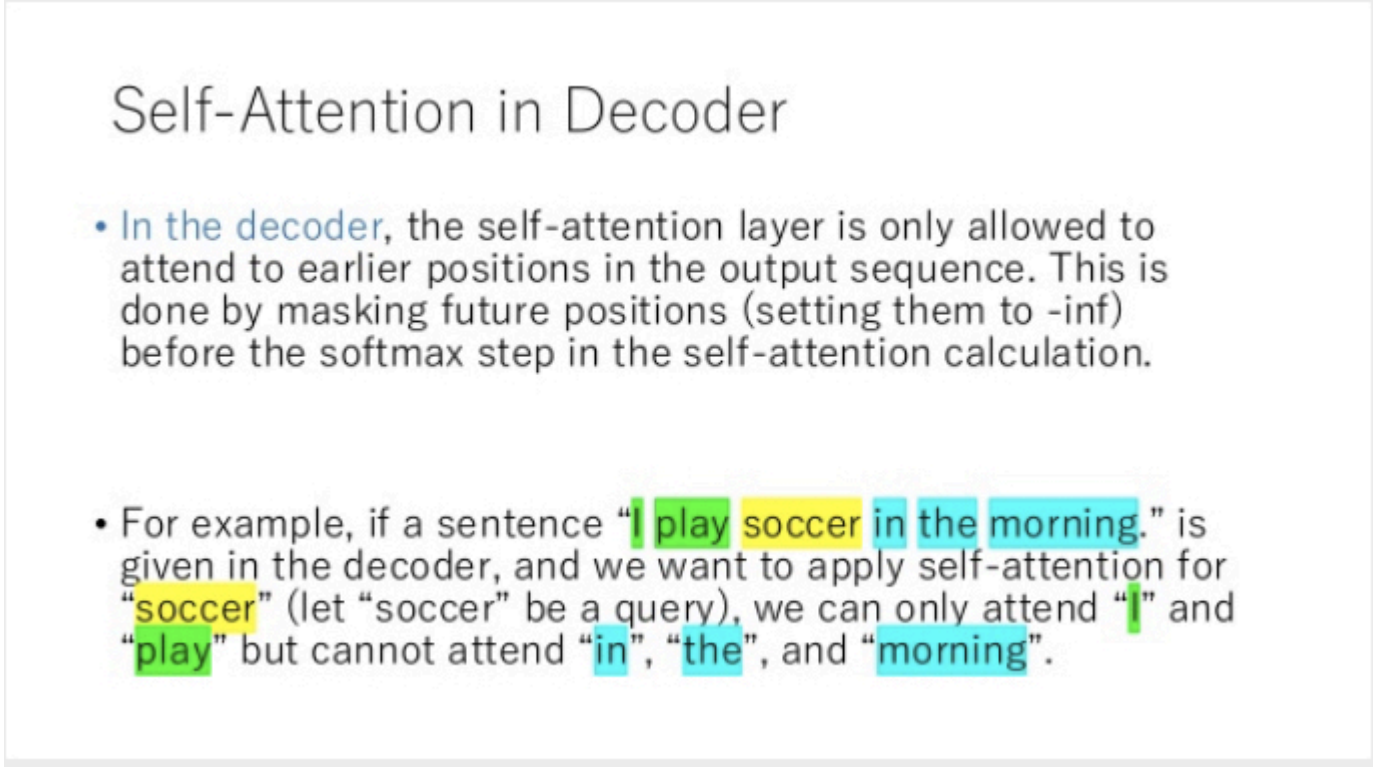
Or on veut obtenir de quoi masquer à chaque mot prévu les mots qu'il devrait prévoir, **donc on veut que les valeurs correspondantes aux mots à venir n'aient aucune attention**

Il faut donc que soit multipliée avec V une matrice de ce type : $\text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$

	Your name	’s	lebowski	lebowski
Your	-26,845	0	0	0	0
name	0,093	-0,00693	0	0	0
’s	3,435	-495,767	3,74	0	0
lebowski	11,964	-0,02413	8,7227	-0,00061	0
lebowski	-1,526	63,23468	2,950	55,15364	1,579

Et pour ce faire on forcera QK^T à avoir les valeurs au dessus de la diagonale de la matrice à moins l'infini (ou chiffre très négatif)

	Your name	’s	lebowski	lebowski
Your	-26,845	-∞	-∞	-∞	-∞
name	0,093	-0,00693	-∞	-∞	-∞
’s	3,435	-495,767	3,74	-∞	-∞
lebowski	11,964	-0,02413	8,7227	-0,00061	-∞
lebowski	-1,526	63,23468	2,950	55,15364	1,579



Self-attention layers in the decoder allow **each position in the decoder to attend to all positions in the decoder up to and including that position**. We need to prevent **leftward information flow** in the decoder to preserve the **auto-regressive property**. We implement this inside of scaled dot-product attention by masking out (setting to $-\infty$) **all values in the input of the softmax which correspond to illegal connections.

4. 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.

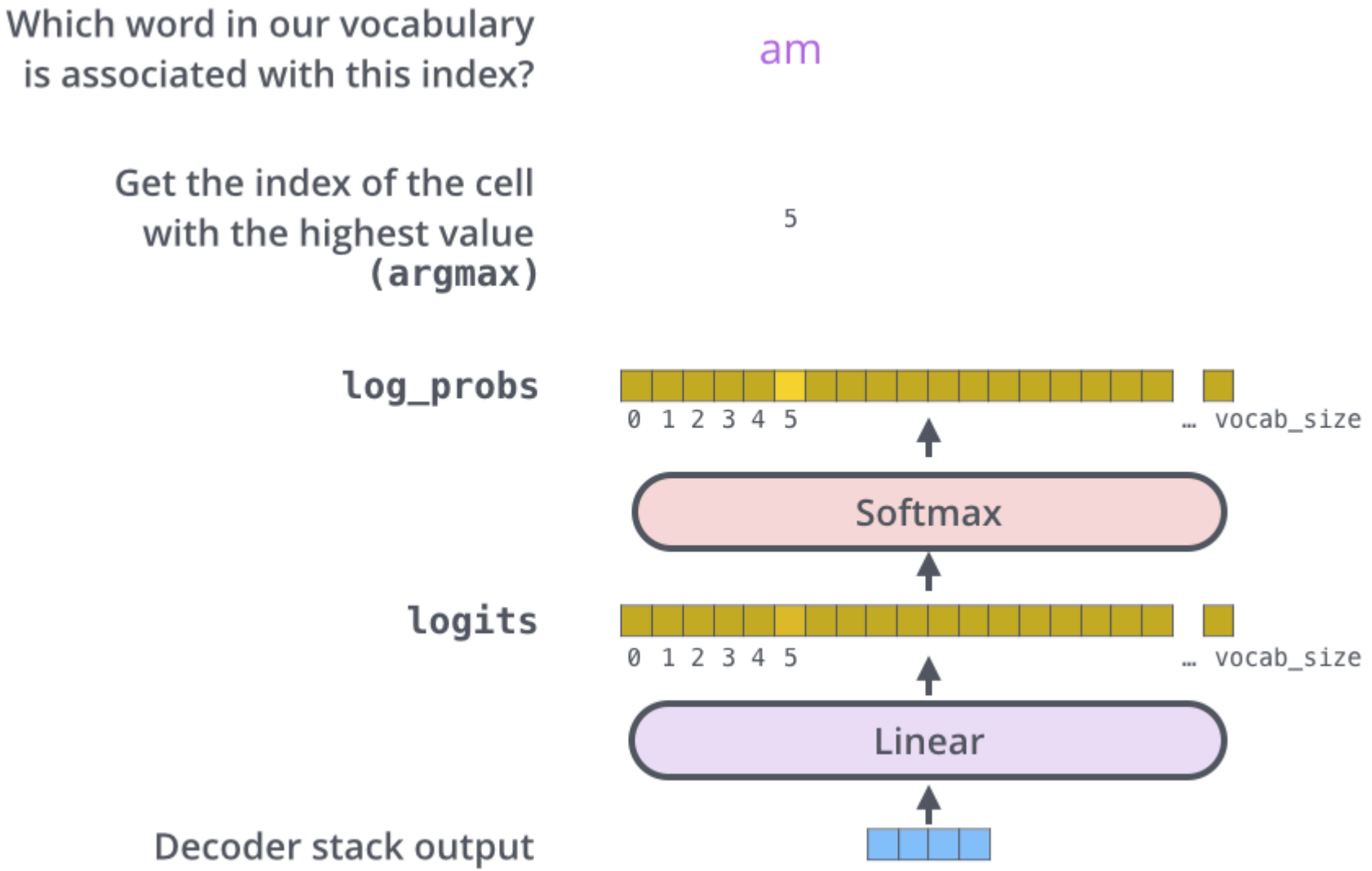
The Linear layer is a simple fully connected neural network that projects the vector produced by the stack of decoders, into a much, much larger vector called a logits vector. *This space is the size of vocabulary (all words). We just project the matrix of weights (provided by the decoder block) into a "vocabulary space".

Mathematically speaking, what does it mean?

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**. 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.** Softmax provides us **the most likely word to predict** (we take the word of the column which give us the highest probability).



This figure starts from the bottom with the vector produced as the output of the decoder stack. It is then turned into an output word.

5. Residual Connection

A residual connection is basically just taking the input and adding it to the output of the sub-network, making training deep networks easier in the field of computer vision. Layer normalization is a normalization method in deep learning that is similar to batch normalization. In layer normalization, the statistics are computed across each feature and are **independent of other examples**. The independence between inputs means that each input has a different normalization operation.

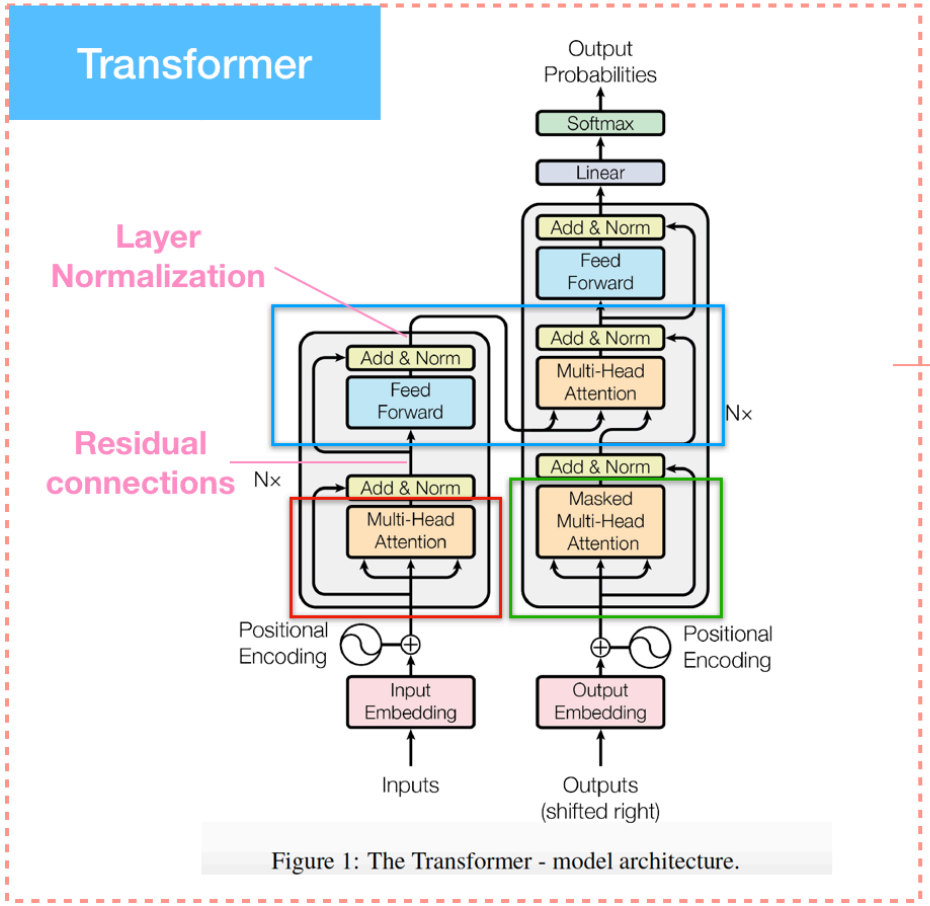
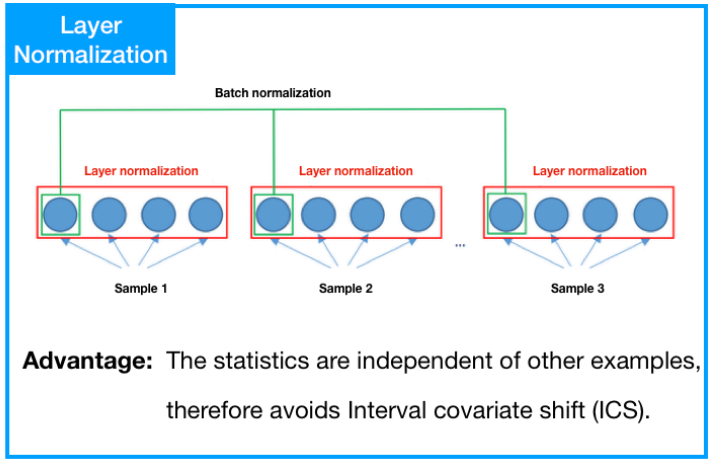
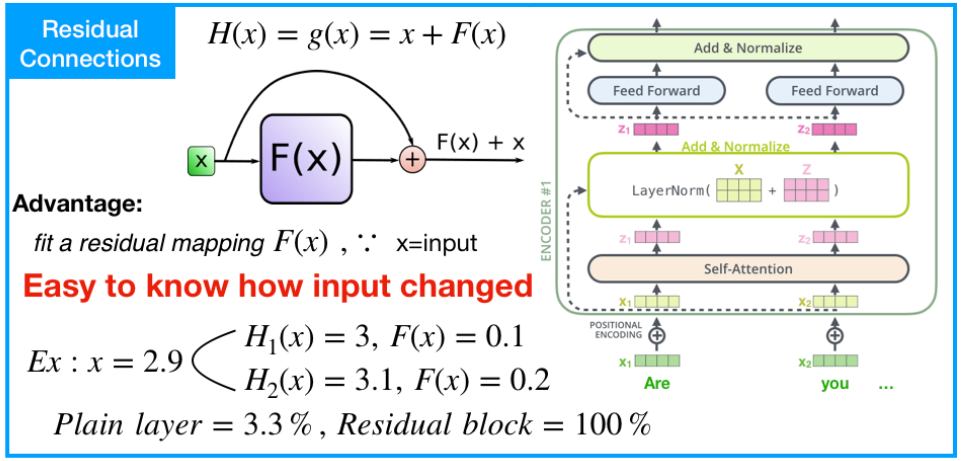
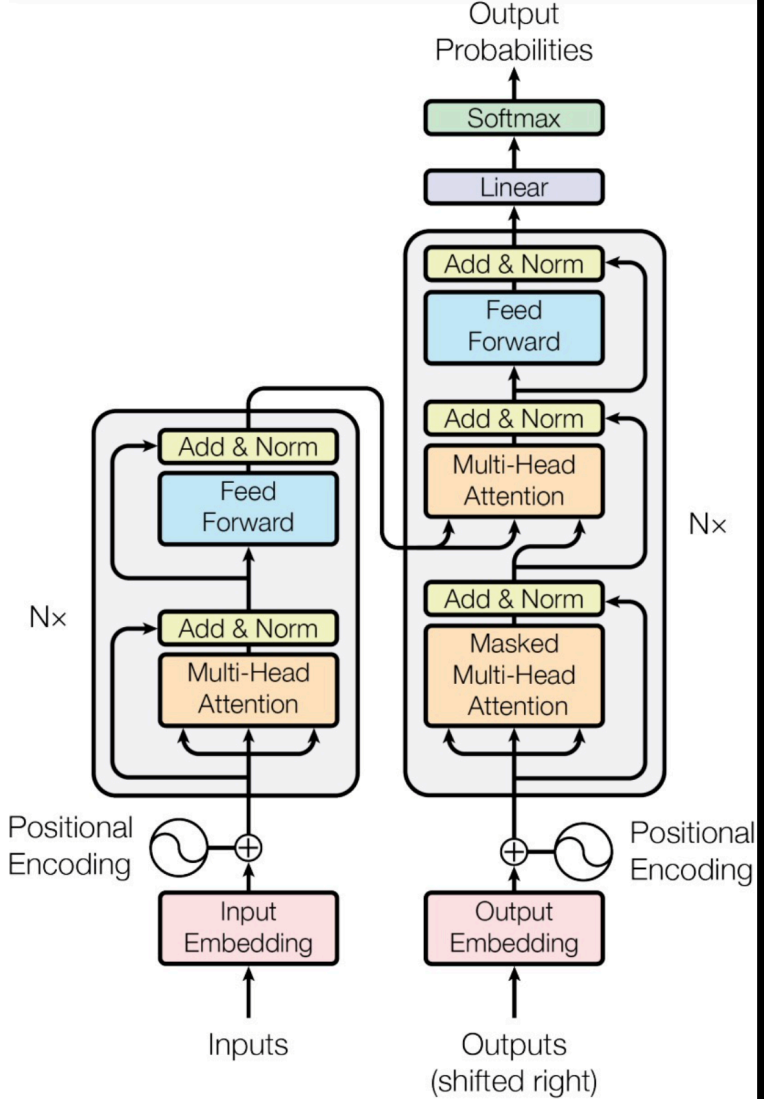


Figure 1: The Transformer - model architecture.





Round Up : Transformer



Revisit to Transformer

- Parallel Processing
- Tokenization
- Positional Embedding
- Three types of Attentions: Multi-head Self-Attention, Masked Multi-head Self-Attention, Cross Attention
- Layer Normalization
- Residual Connection

In [ ]: