Transformers

Língua Natural e Sistemas Conversacionais

Luiz Faria

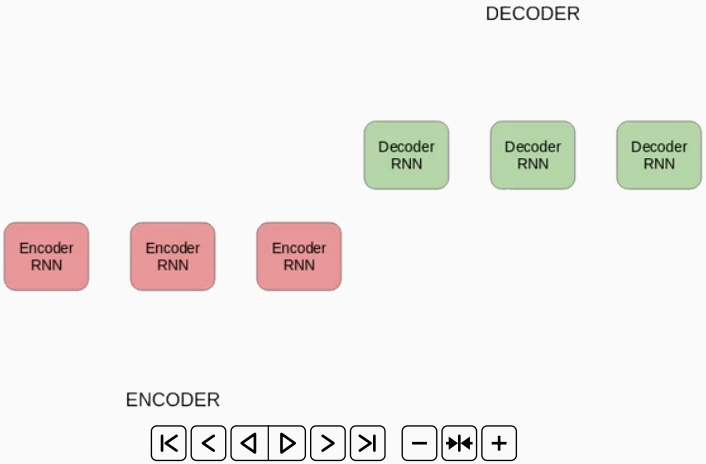
Limitations of Seq2Seq Models

Despite the success of Seq2Seq models, there are certain limitations of these models with attention:

- Dealing with long-range dependencies is still challenging, even when using LSTM with attention to implement the encoder and decoder
- The sequential nature of the model architecture prevents parallelization
  - With Recurrent Neural Networks (RNN's) we used to treat sequences sequentially to keep the order of the sentence in place. To satisfy that design, each RNN component (layer) needs the previous (hidden) output. As such, stacked LSTM computations were performed sequentially.

These challenges are addressed by Google Brain's **Transformer** concept

Limitations of Seq2Seq Models



A Seq2Seq model without attention used in a translation task: a sequential model that prevents parallelization

https://www.analyticsvidhya.com/blog/2019/06/understanding-transformers-nlp-state-of-the-art-models/

Introduction to the Transformer

- The Transformer in NLP is a novel architecture that aims to solve sequence-to-sequence tasks while handling long-range dependencies with ease
- It was proposed in the paper *Attention Is All You Need*
- Transformer NLP models are based on the Attention mechanism, taking its core idea even further: In addition to using Attention to compute representations (i.e., context vectors) out of the encoder's hidden state vectors, why not use Attention to compute the encoder's hidden state vectors themselves?
- The immediate advantage of this is getting rid of the inherent sequential structure of RNNs, which hinders the parallelization of models
- Additionally, Transformer NLP models have so far displayed better performance and speed than RNN models

Self-Attention

“The Transformer is the first transduction model relying entirely on self-attention to compute representations of its input and output without using sequence-aligned RNNs or convolution.”

“Transduction” means the conversion of input sequences into output sequences. The idea behind Transformer is to handle the dependencies between input and output with attention and recurrence completely

Self-Attention

Attention allow to focus on parts of the input sequence while predicting the output sequence. If the model predicted the word “rouge”, we are very likely to find a high weight-age for the word “red” in the input sequence. So attention, in a way, allowed us to map some connection/correlation between the input word “red” and the output word “rouge”

Self-attention, sometimes called intra-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence

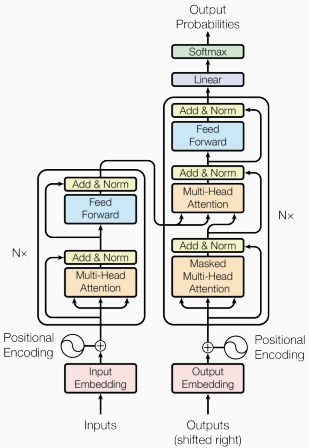
Self-Attention

Self-attention helps us create similar connections but within the same sentence

> “I poured water from the bottle into the **cup** until **it** was **full**.”
> it → cup
> “I poured water from the **bottle** into the cup until **it** was **empty**.”
> it → bottle

By changing one word “full” → “empty” the reference object for “it” changed. If we are translating such a sentence, we will want to know the word “it” refers to

Transformer Overview



Transformer model architecture

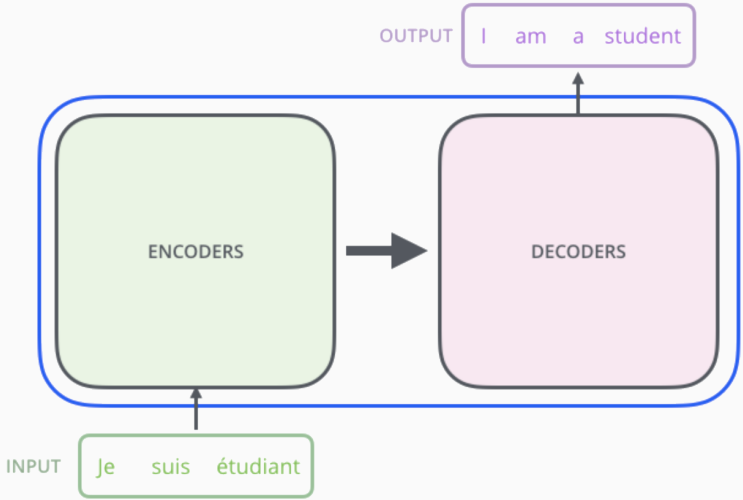
Attention Is All You Need

- The encoder maps an input sequence of symbol representations $(x_1, \ldots, x_n)$ to a sequence of continuous representations $z = (z_1, \ldots, z_n)$. Given z, the decoder then generates an output sequence $(y_1, \ldots, y_m)$ 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

Transformer as a Single Black Box

http://jalammar.github.io/illustrated-transformer/

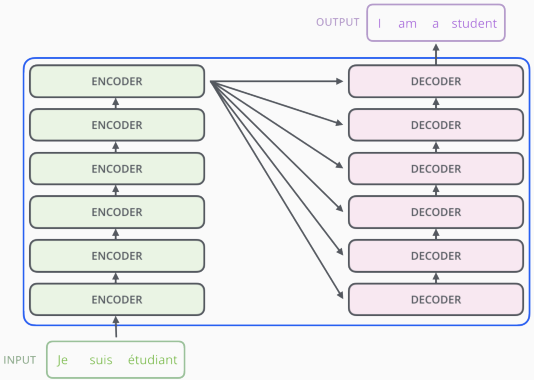
Encoder-Decoder



http://jalammar.github.io/illustrated-transformer/

Encoder and Decoder Components

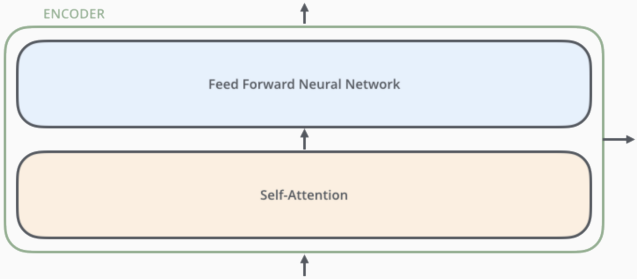
The encoding component is a stack of 6 encoders The decoding component is a stack of decoders of the same number



http://jalammar.github.io/illustrated-transformer/

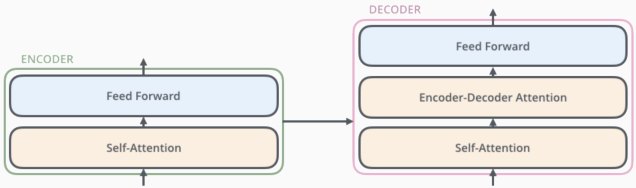
The Encoder

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



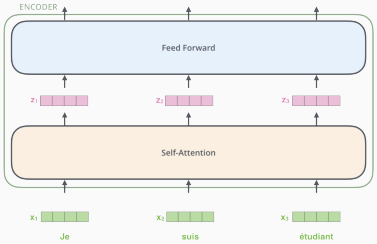
Encoder-Decoder Connection

- 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
- 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 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)



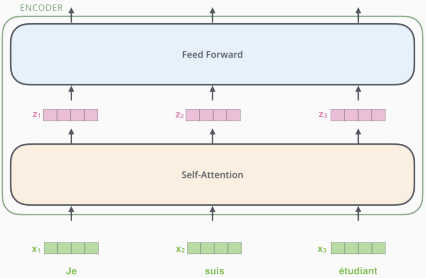
Input Sequence

- As usual, each word of the input sequence is converted into an embedding vector
- The embedding only happens in the bottom-most encoder
- Each encoder receives a list of vectors each of the size 512; the number of vectors is a hyperparameter – it would be the length of the longest sentence in the training dataset
- The bottom encoder receives the words embeddings; the remain decoders receive the output of the encoder that's directly below

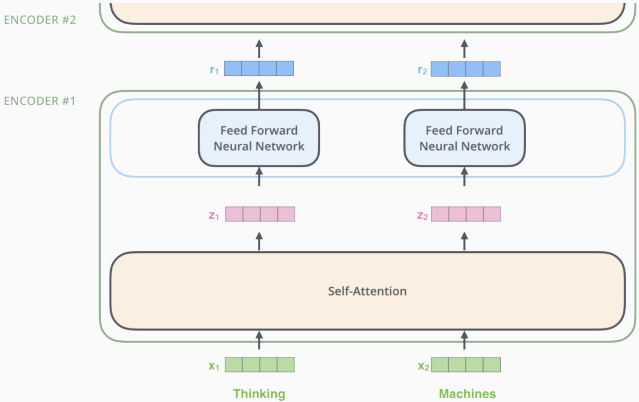


Input Sequence

- The embedding of each word flows through each of the two layers of the first encoder
- The word in each position flows through its own path in the encoder
- There are dependencies between these paths in the self-attention layer
- The feed-forward layer does not have those dependencies, however, and thus the various paths can be executed in parallel while flowing through the feed-forward layer



Encoding



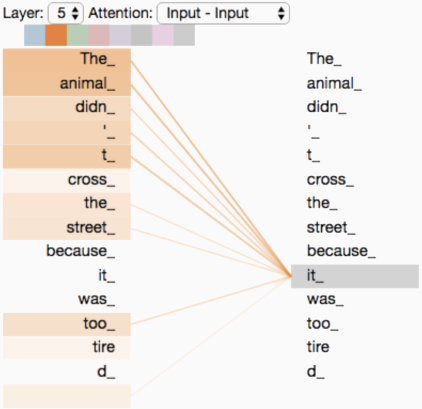
The word at each position passes through a self-attention process; then, they each pass through a feed-forward neural network – the exact same network with each vector flowing through it separately; the output is sent upwards to the next encoder

Self-Attention at a High Level

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

- "it" refers to the street or to the animal?
- When the model is processing the word "it", self-attention allows it to associate "it" with "animal"
- As the model processes each word in the input sequence, self-attention allows it to look at other positions in the sequence for clues that can help lead to a better encoding for this word
- In a RNN, maintaining a hidden state allows an RNN 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 incorporate the "understanding" of other relevant words into the one we're currently processing

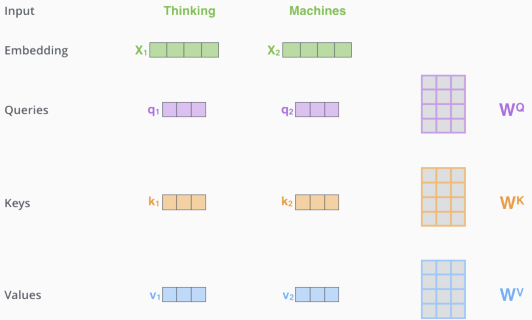
Self-Attention at a High Level



As we are encoding the word “it” in encoder #5 (the top encoder in the stack), part of the attention mechanism was focusing on “The Animal”, and incorporated a part of its representation into the encoding of “it”

Calculating Self-Attention
First step

- Determining vectors Q (Query), K (Key), and V (Value) by multiplying the embedding by three matrices that we trained during the training process
- Vectors Q, K, and V are smaller (size 64) than the embedding and encoder input/output vectors have dimensionality of 512



Multiplying x_1 by the W^Q weight matrix produces q_1, the "query" vector associated with that word; we end up creating a "query", a "key", and a "value" projection of each word in the input sentence

Calculating Self-Attention

Second step

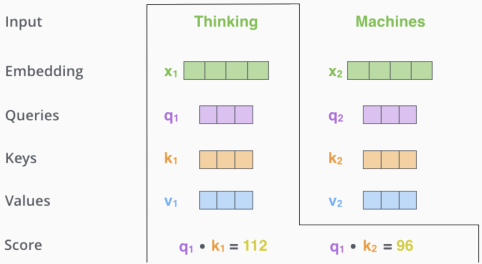
Calculating a score: the score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position

- For calculating the self-attention for the first word in this example, “Thinking”, we need to score each word of the input sentence against this word
- The score is calculated by taking the dot product of the query vector with the key vector of the respective word we’re scoring

Calculating Self-Attention

Second step

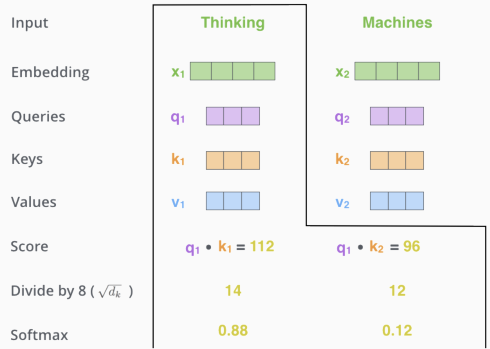
- If we're processing the self-attention for the word in position #1, the first score would be the dot product of q_1 and k_1, and the second score would be the dot product of q_1 and k_2



Calculating Self-Attention

Third and forth steps

- Divide the scores by 8 (the square root of the dimension of the key vectors (64) – this leads to having more stable gradients
- Then pass the result through a softmax operation – softmax normalizes the scores so they're all positive and add up to 1

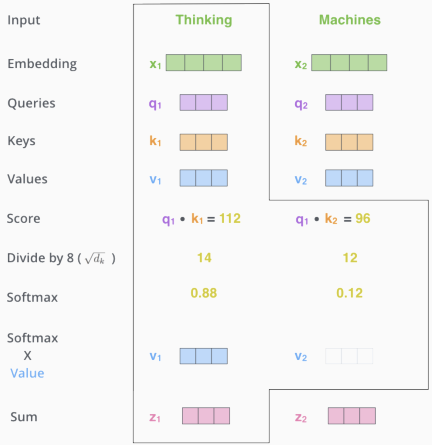


The softmax score determines how much each word will be expressed at this position; the word at this position will have the highest softmax score, but sometimes it's useful to attend to another word that is relevant to the current word

Calculating Self-Attention

Fifth and sixth steps

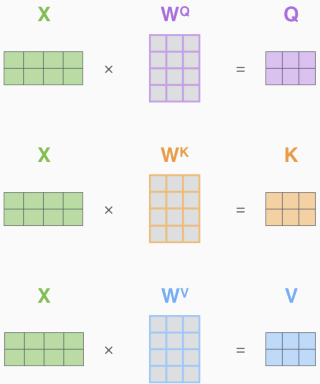
- Multiply each value vector by the softmax score – the intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words (by multiplying them by tiny numbers)
- Sum up the weighted value vectors – this produces the output of the self-attention layer at this position (for the first word)
- The resulting vector is sent along to the feed-forward neural network



Matrix Calculation of Self-Attention

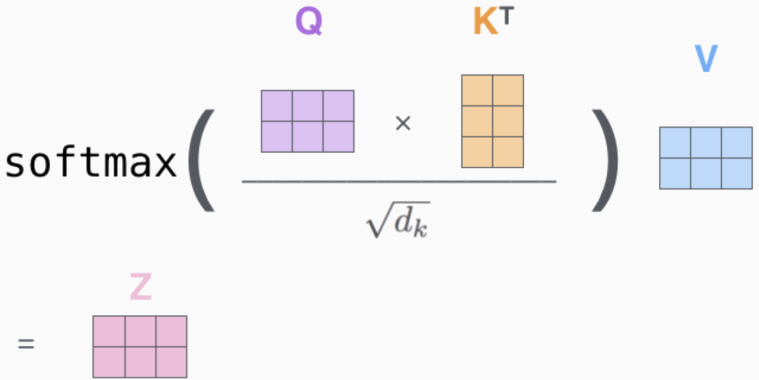
The self-attention calculation is done in matrix form for faster processing

In the first step, we calculate the Query, Key, and Value matrices packing the words embeddings into a matrix X, and multiplying it by the weight matrices we've trained (W^Q, W^K, W^V)



Matrix Calculation of Self-Attention

In the matrix form, steps two through six are condensed in one formula to calculate the outputs of the self-attention layer



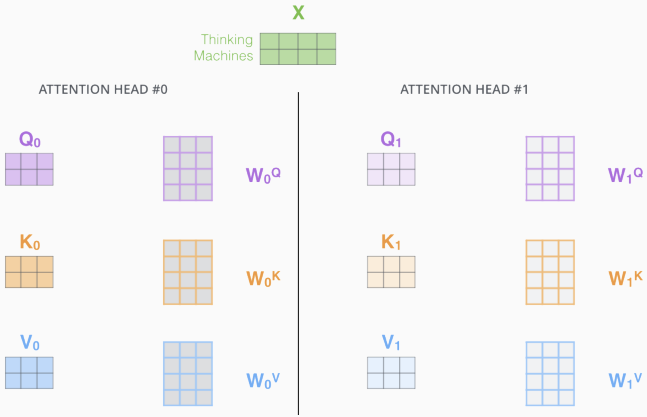
The self-attention calculation in matrix form

Multi-headed Attention

A mechanism called “multi-headed” attention improves performance of the self-attention layer in two ways:

- It expands the model’s ability to focus on different positions; in a previous figure, z_1 contains a little bit of every other encoding, but it could be dominated by the actual word itself
- It gives the attention layer multiple “representation subspaces” – with multi-headed attention we have not only one, but multiple sets of Query/Key/Value weight matrices (the Transformer uses eight attention heads, so we end up with eight sets for each encoder/decoder); each of these sets is randomly initialized; after training, each set is used to project the input embeddings (or vectors from lower encoders/decoders) into a different representation subspace

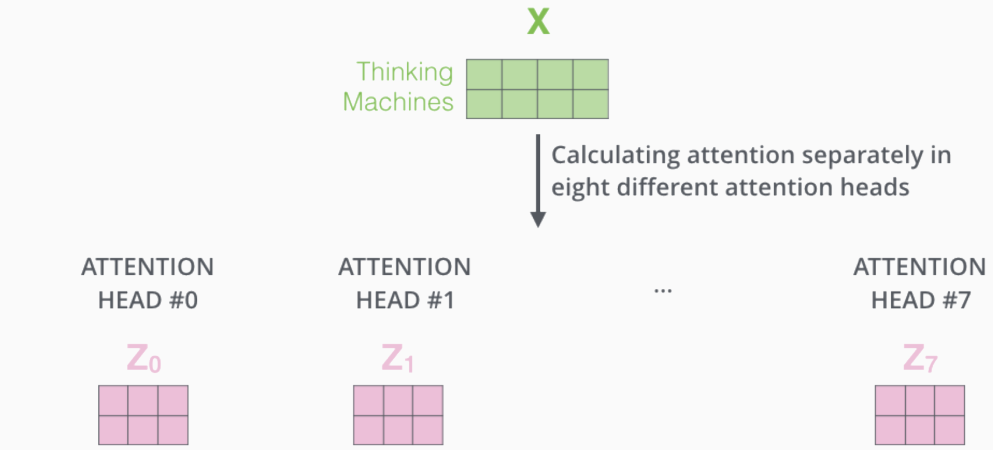
Multi-headed Attention



With multi-headed attention, we maintain separate $Q/K/V$ weight matrices for each head resulting in different $Q/K/V$ matrices; matrix X is multiplied by the $W^Q/W^K/W^V$ matrices to produce $Q/K/V$ matrices

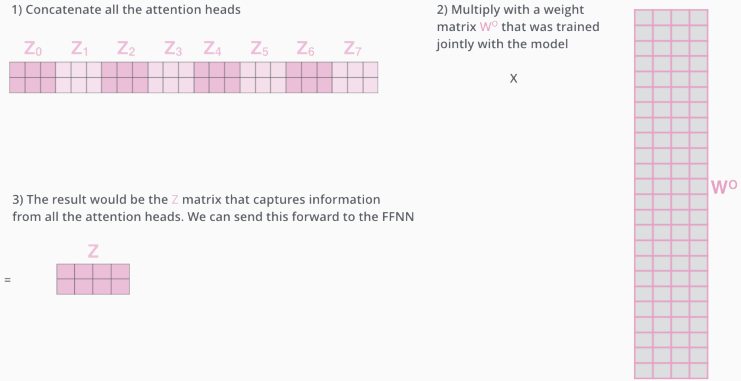
Multi-headed Attention

Now, we do the same self-attention calculation, just eight different times with different weight matrices, and we end up with eight different *Z* matrices



Multi-headed Attention

However, the feed-forward layer is not expecting eight matrices – it's expecting a single matrix (a vector for each word) – So we need a way to condense these eight down into a single matrix, by concatenating the matrices and then multiple them by an additional weights matrix W^O



Multi-headed Attention – Joining all the Steps

1) This is our input sentence*

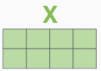
2) We embed each word*

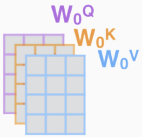
3) Split into 8 heads. We multiply X or R with weight matrices

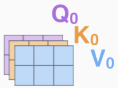
4) Calculate attention using the resulting Q/K/V matrices

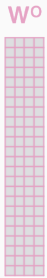
5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer

Thinking Machines



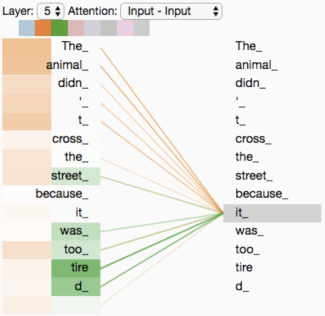


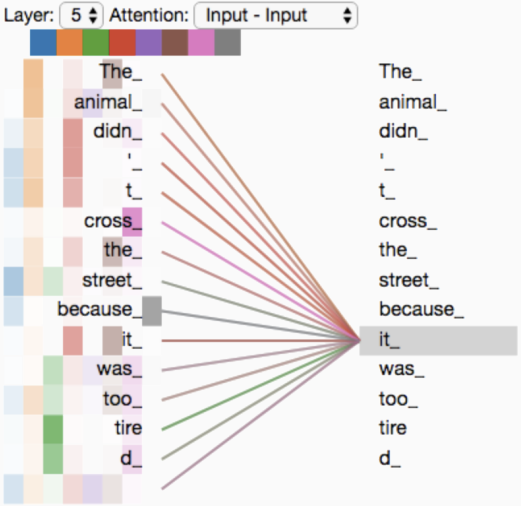




* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one

Multi-headed Attention Result



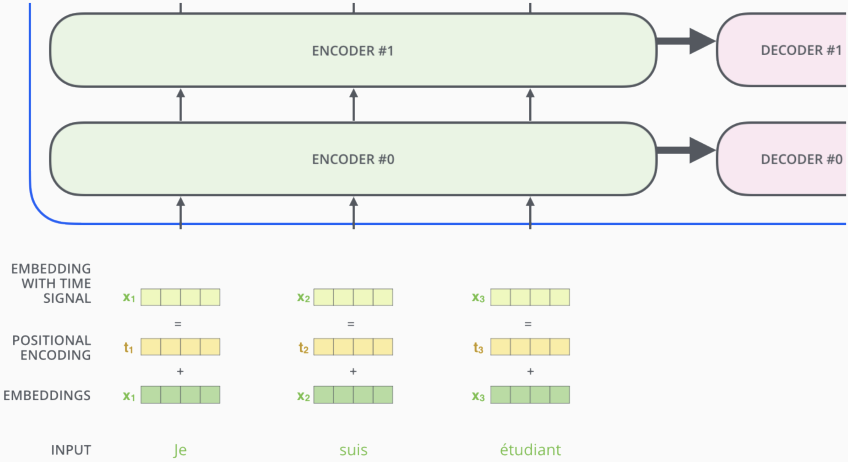


As we encode the word “it”, one attention head is focusing most on “the animal”, while another is focusing on “tired” – in a sense, the model’s representation of the word “it” incorporates some of the representation of both “animal” and “tired”

Representing The Order of The Sequence Using Positional Encoding

- The model described is away to account for the order of the words in the input sequence
- To address this, the transformer adds a vector to each input embedding
- These vectors follow a specific pattern that the model learns, which helps it determine the position of each word, or 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

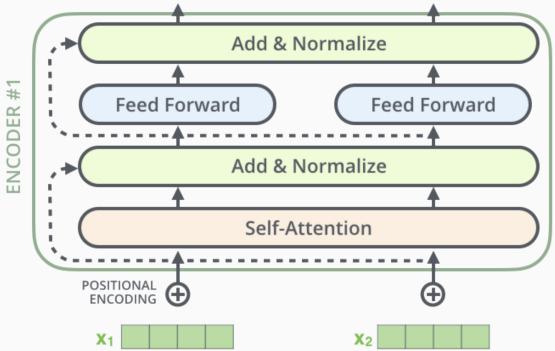
Representing The Order of The Sequence Using Positional Encoding



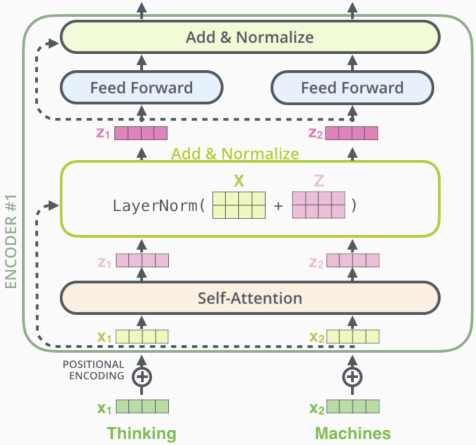
To give the model a sense of the order of the words, we add positional encoding vectors

Residual Connection around Encoders' Sub-layers

- Each sub-layer (self-attention, ffnn) in each encoder has a residual connection around it, and is followed by a layer-normalization step
  - Layer-normalization – improves both the training time and the generalization performance of the model by normalizing each feature of the activations to zero mean and unit variance



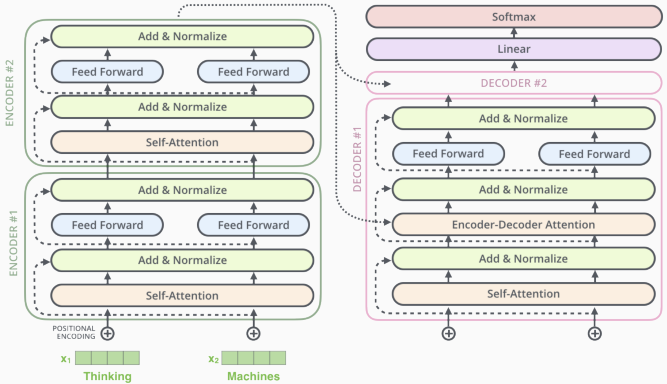
Residual Connection around Encoders' Sub-layers



Vectors and the layer-norm operation associated with self attention

Residual Connection around Encoders' Sub-layers

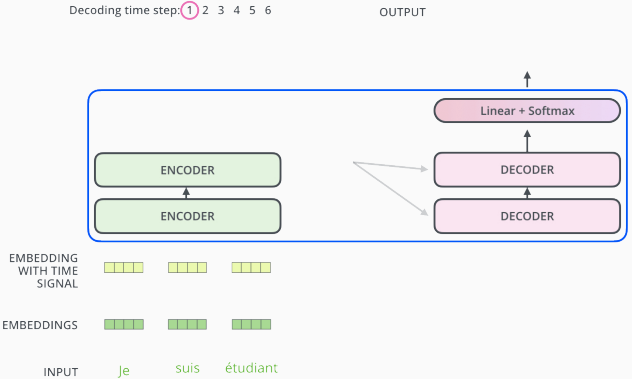
The residual connection, and the layer-normalization step goes for the sub-layers of the **decoder** as well



Example of a Transformer with 2 stacked encoders and decoders

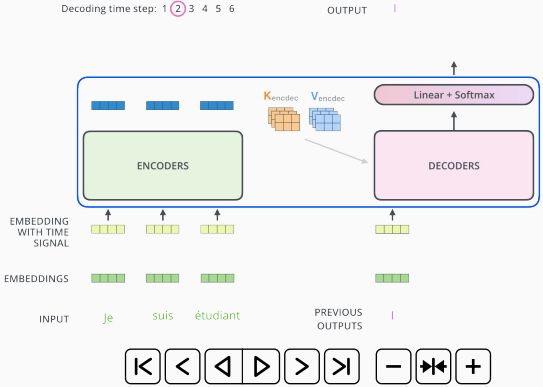
Decoder

- The decoder uses the same components present on the encoder
- The output of the top encoder is transformed into a set of attention vectors *K* and *V*
- These vectors are used by each decoder in its “encoder-decoder attention” layer which helps the decoder focus on appropriate places in the input sequence:



Decoder

- The following steps are repeated until a special symbol is reached indicating the transformer decoder has completed its output
- The output of each step is fed to the bottom decoder in the next time step, and the decoders bubble up their decoding results just like the encoders did
- Similar to what we did with the encoder inputs, we embed and add positional encoding to those decoder inputs to indicate the position of each word



Self-Attention Layer in the Decoder

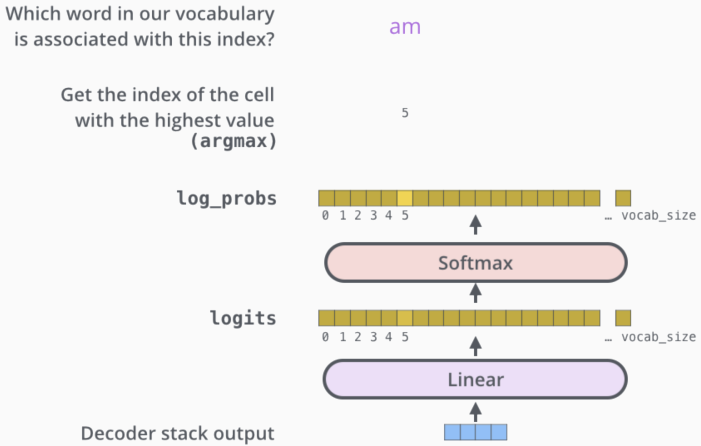
The self attention layer in the decoder operate in a slightly different way than the one in the encoder:

- 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 – inf) before the softmax step in the self-attention calculation
- The “Encoder-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

Final Linear and Softmax Layer of the Decoder

- The **decoder stack** outputs a vector of floats – this vector must be transformed in a word
- This transformation is performed by the final Linear layer which is followed by a Softmax layer
- The **Linear layer** is a simple fcnn that projects the vector produced by the stack of decoders, into a much larger vector called a **logits** vector (or not-yet normalised predictions (or outputs) of the model)
- The size of the logits vector is the size of the vocabulary and each cell corresponds to the score of a unique word
- The softmax layer 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

Final Linear and Softmax Layer of the Decoder



Training

- During training, an untrained model would go through the exact same forward pass
- As we are using a labeled training dataset, we can compare its output with the actual correct output
- Example of a output vocabulary only contains six words created in the preprocessing phase, before training start:

Output Vocabulary

WORD	a	am	I	thanks	student	<eos>
INDEX	0	1	2	3	4	5

Training

- Each word will be represented by an one-hot encoding vector with the size of the vocabulary:

Output Vocabulary

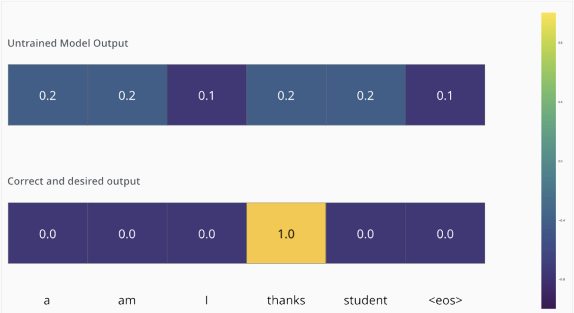
WORD	a	am	I	thanks	student	<eos>
INDEX	0	1	2	3	4	5

One-hot encoding of the word “am“

0.0	1.0	0.0	0.0	0.0	0.0

Training

- The loss function will be the metric used to optimize the model during the training phase
- Lets's consider the example of translating "merci" to "thanks"
- The output will be a probability distribution indicating the word "thanks"
- During training, we need to compare the output probability distribution with the target distribution

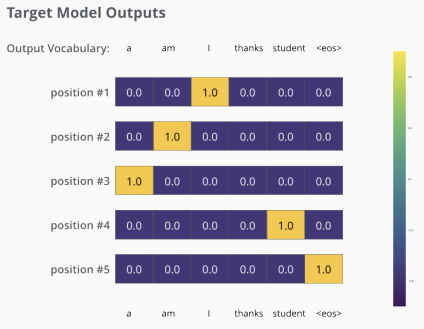


Since the model's parameters (weights) are all initialized randomly, the (untrained) model produces a probability distribution with arbitrary values for each cell/word

Training

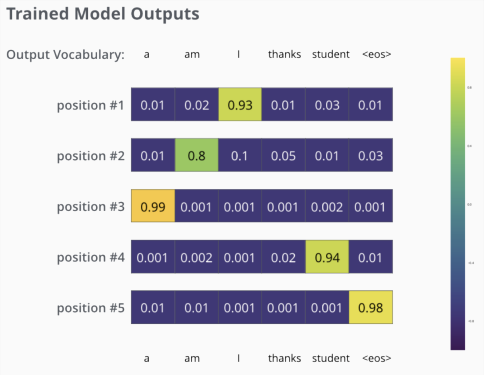
Example: “*je suis étudiant*” → “*i am a student*”

- Each probability distribution is represented by a vector of width *vocab_size* (6 in our toy example, but more realistically a number like 30,000 or 50,000)
- The first probability distribution (first time step) has the highest probability at the cell associated with the word “i”
- The second probability distribution has the highest probability at the cell associated with the word “am”
- And so on, until the fifth output distribution indicates < *eos* > symbol, which also has a cell associated with it from the 10,000 element vocabulary



Training

After training the model on a large enough dataset, the produced probability distributions would look like this:



At each time step, the model will select the word with the highest probability from that probability distribution (greedy decoding); alternatively, using beam search