

Transformer

Xiao Wang

Department of Statistics
Purdue University

Table of contents

1 Introduction

2 Embedding

3 Self-Attention

4 Multi-Headed Attention

5 Positional Encoding

6 The Decoder

A High-Level Look

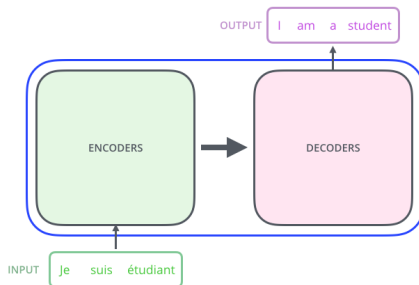
Let's begin by looking at the model as a single black box. In a machine translation application, it would take a sentence in one language, and output its translation in another.



¹<http://jalammar.github.io/illustrated-transformer/>

A High-Level Look

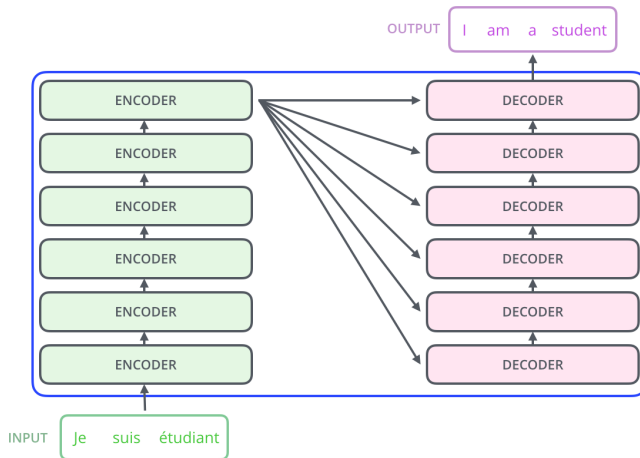
Add an encoding component, a decoding component, and connections between them.



¹<http://jalammar.github.io/illustrated-transformer/>

A High-Level Look

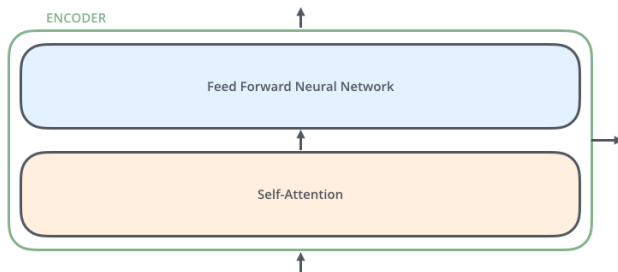
The encoding component is a stack of encoders. The decoding component is a stack of decoders of the same number.



¹<http://jalamar.github.io/illustrated-transformer/>

A High-Level Look

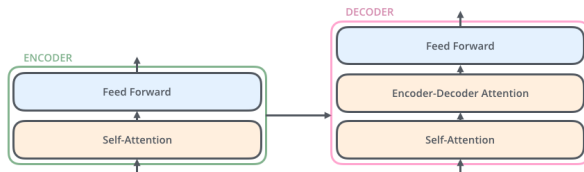
The encoders are all identical in structure (yet they do not share weights). Each one is broken down into two sub-layers.



¹<http://jalammar.github.io/illustrated-transformer/>

A High-Level Look

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.



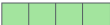
¹<http://jalammar.github.io/illustrated-transformer/>

Table of contents

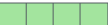
- 1 Introduction
- 2 **Embedding**
- 3 Self-Attention
- 4 Multi-Headed Attention
- 5 Positional Encoding
- 6 The Decoder

Embedding

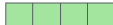
As is the case in NLP applications in general, we begin by turning each input word into a vector using an embedding algorithm.

x_1 

Je

x_2 

suis

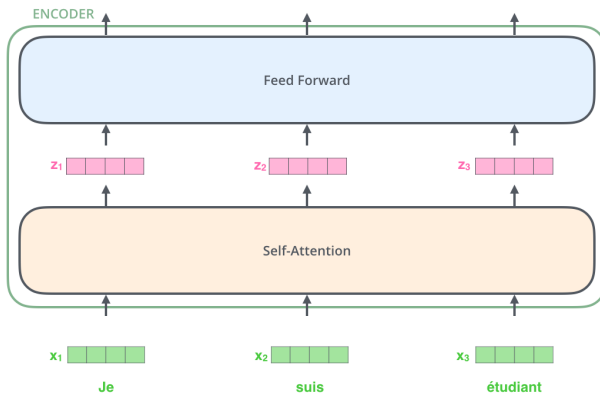
x_3 

étudiant

¹<http://jalammar.github.io/illustrated-transformer/>

Encoder

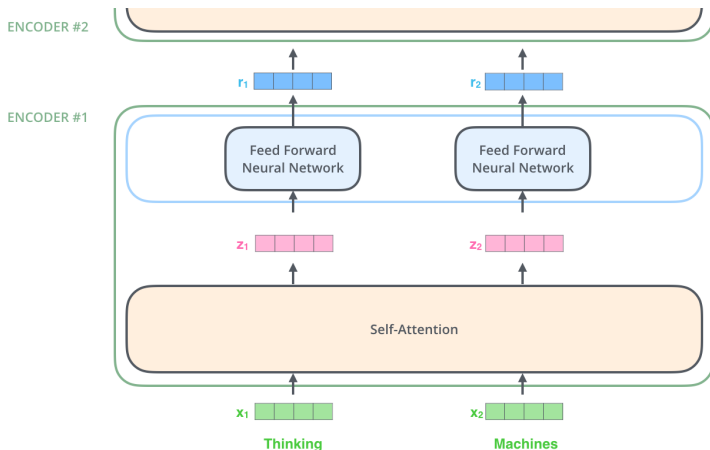
After embedding the words in our input sequence, each of them flows through each of the two layers of the encoder.



¹<http://jalammar.github.io/illustrated-transformer/>

Multiple Encoders

An encoder receives a list of vectors as input. It processes this list by passing these vectors into a 'self-attention' layer, then into a feed-forward neural network, then sends out the output upwards to the next encoder.



¹<http://jalammar.github.io/illustrated-transformer/>

Table of contents

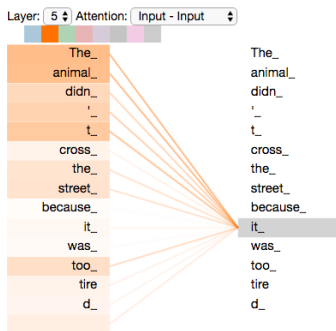
- 1 Introduction
- 2 Embedding
- 3 Self-Attention**
- 4 Multi-Headed Attention
- 5 Positional Encoding
- 6 The Decoder

Self-Attention at a High-Level

- Say the following sentence is an input sentence we want to translate:
"The animal didn't cross the street because it was too tired"
What does "it" in this sentence refer to? Is it referring to the street or to the animal? It's a simple question to a human, but not as simple to an algorithm.
- When the model is processing the word "it", self-attention allows it to associate "it" with "animal".
- As the model processes each word (each position in the input sequence), self attention allows it to look at other positions in the input sequence for clues that can help lead to a better encoding for this word.

Self-Attention at a High-Level

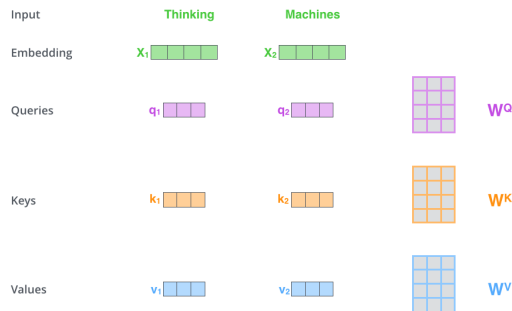
Self-attention is the method the Transformer uses to bake the “understanding” of other relevant words into the one we’re currently processing.



¹<http://jalammar.github.io/illustrated-transformer/>

Self-Attention in Detail

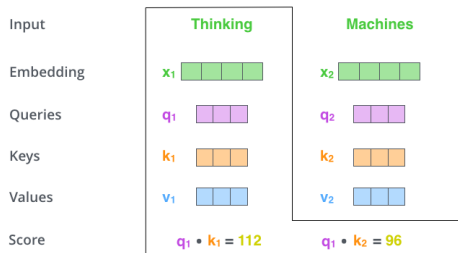
The first step in calculating self-attention is to create three vectors from each of the encoder's input vectors (in this case, the embedding of each word). So for each word, we create a Query vector, a Key vector, and a Value vector. These vectors are created by multiplying the embedding by three matrices that we trained during the training process.



¹<http://jalammar.github.io/illustrated-transformer/>

Self-Attention in Detail

The second step in calculating self-attention is to calculate a score. The score is calculated by taking the dot product of the query vector with the key vector of the respective word we're scoring.



¹<http://jalammar.github.io/illustrated-transformer/>

Self-Attention in Detail

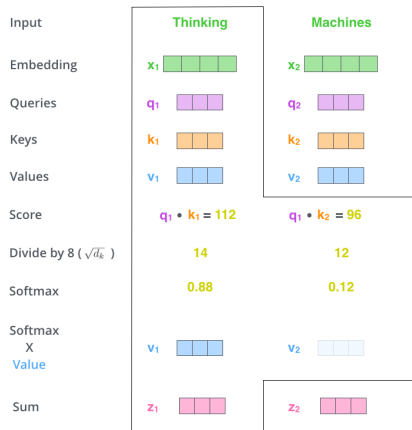
The third and fourth steps are to divide the scores by 8 (the square root of the dimension of the key vectors used in the paper – 64. This leads to having more stable gradients. There could be other possible values here, but this is the default), then pass the result through a softmax operation. Softmax normalizes the scores so they're all positive and add up to 1.

Input	Thinking	Machines
Embedding	x_1	x_2
Queries	q_1	q_2
Keys	k_1	k_2
Values	v_1	v_2
Score	$q_1 \cdot k_1 = 112$	$q_1 \cdot k_2 = 96$
Divide by $8 (\sqrt{d_k})$	14	12
Softmax	0.88	0.12

¹<http://jalammar.github.io/illustrated-transformer/>

Self-Attention in Detail

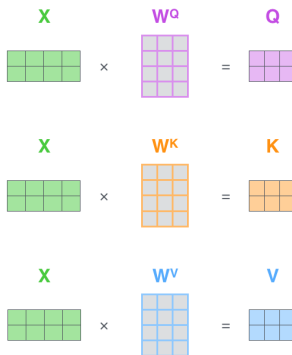
The fifth step is to multiply each value vector by the softmax score (in preparation to sum them up). The sixth step is to sum up the weighted value vectors. This produces the output of the self-attention layer at this position (for the first word).



¹<http://jalammar.github.io/illustrated-transformer/>

Matrix Calculation of Self-Attention

We do that by packing our embeddings into a matrix X , and multiplying it by the weight matrices we've trained (W^Q , W^K , W^V).



¹<http://jalammar.github.io/illustrated-transformer/>

Matrix Calculation of Self-Attention

Since we're dealing with matrices, we can condense steps two through six in one formula to calculate the outputs of the self-attention layer.

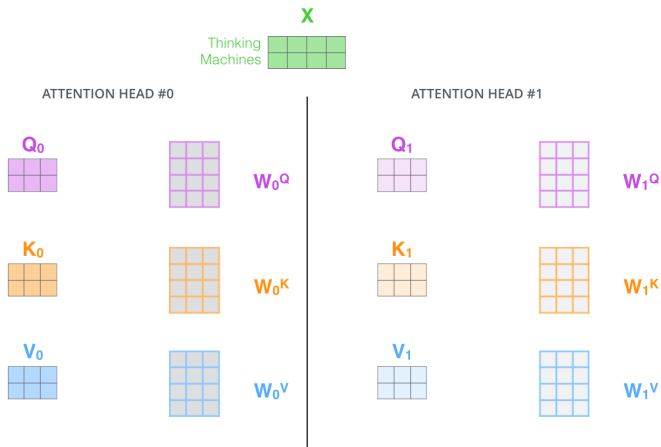
$$\text{softmax}\left(\frac{\begin{matrix} \text{Q} \\ \begin{matrix} \text{3x2 grid} \end{matrix} \end{matrix} \times \begin{matrix} \text{K}^T \\ \begin{matrix} \text{3x2 grid} \end{matrix} \end{matrix}}{\sqrt{d_k}}\right) \begin{matrix} \text{V} \\ \begin{matrix} \text{3x2 grid} \end{matrix} \end{matrix}$$
$$= \begin{matrix} \text{Z} \\ \begin{matrix} \text{3x2 grid} \end{matrix} \end{matrix}$$

¹<http://jalammar.github.io/illustrated-transformer/>

Table of contents

- 1 Introduction
- 2 Embedding
- 3 Self-Attention
- 4 Multi-Headed Attention**
- 5 Positional Encoding
- 6 The Decoder

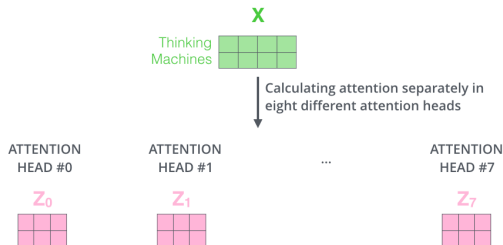
Multi-Headed Attention



¹<http://jalammar.github.io/illustrated-transformer/>

Multi-Headed Attention

If we do the same self-attention calculation we outlined above, just eight different times with different weight matrices, we end up with eight different Z matrices.



¹<http://jalammar.github.io/illustrated-transformer/>

Multi-Headed Attention

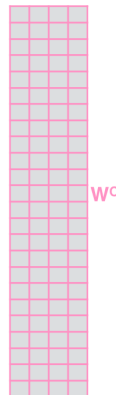
We concatenate the matrices then multiply them by an additional weights matrix W^O .

1) Concatenate all the attention heads



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

x



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



¹<http://jalammar.github.io/illustrated-transformer/>

Multi-Headed Attention

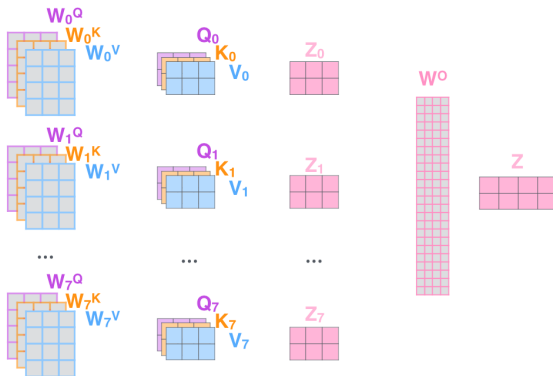
Let me try to put them all in one visual so we can look at them in one place.

- 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



¹<http://jalammr.github.io/illustrated-transformer/>

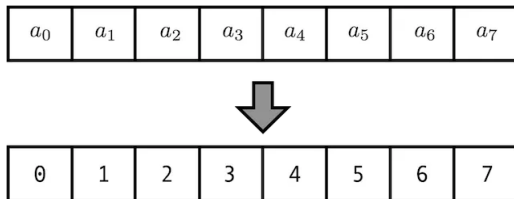
Table of contents

- 1 Introduction
- 2 Embedding
- 3 Self-Attention
- 4 Multi-Headed Attention
- 5 Positional Encoding**
- 6 The Decoder

Positional Encoding

A positional encoding is a finite dimensional representation of the location or “position” of items in a sequence. Given some sequence $A = [a_0, \dots, a_{n-1}]$, the positional encoding must be some type of tensor that we can feed to a model to tell it where some value a_i is in the sequence A .

Guess #1: just count



- The scale of these numbers is huge. If we have a sequence of 500 tokens, we will end up with a 500 in our vector.
- Not friendly to DNNs.

Guess #2: normalize the count

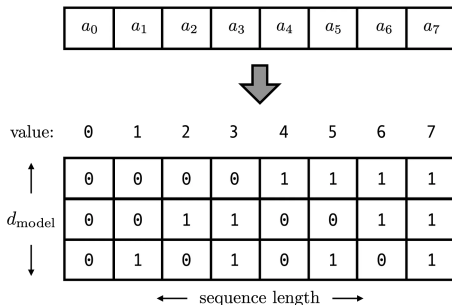
a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7
-------	-------	-------	-------	-------	-------	-------	-------



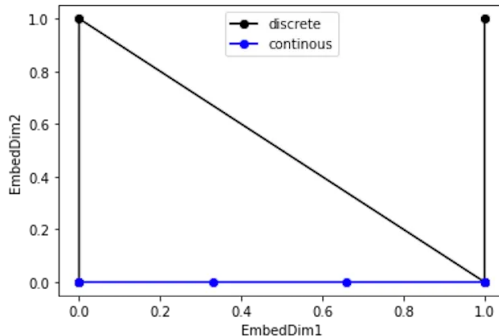
0	0.14	0.29	0.43	0.57	0.71	0.86	1
---	------	------	------	------	------	------	---

- We can no longer deal with an arbitrary sequence length. This is because each of these entries is divided by the sequence length.
- A positional encoding value of say 0.8 means a totally different thing to a sequence of length 5 than it does for one of length 20.

Guess #3: just count but use binary

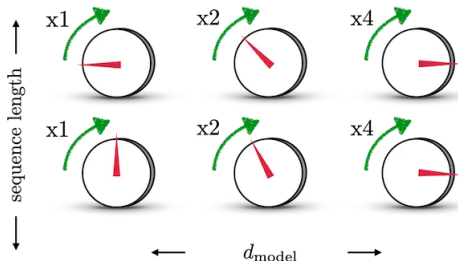


- We can choose the dimension of this new embedding space that holds the binary vector to be whatever we like.
- We still have not fully normalized. Remember that we want things to be positive and negative roughly equally. This is easily fixed: just rescale $[0, 1] \rightarrow [-1, 1]$ via $f(x) = 2x - 1$.
- Our binary vectors come from a discrete function, and not a discretization of a continuous function.



- We want to find a way to make the binary vector a discretization of something continuous.
- We want a function that connects the dots in a smooth way that looks natural. For anyone who has studied geometry, what we are really doing is finding an embedding manifold.

Guess #4: use a continuous binary vector

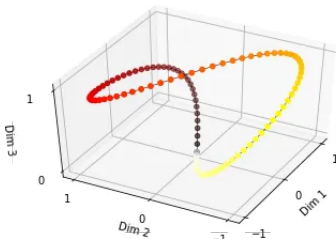


- To make our binary vector continuous, we need a function that interpolates a back and forth cycle $0 \rightarrow 1 \rightarrow 0 \rightarrow 1 \rightarrow 0 \dots$ etc
- Let's use a sine function.

- We now want our sine functions to undergo one rotation at precisely the right moments. For the first entry, it should shift $0 \rightarrow 1$ and go back every time we move one step in the sequence. This means that it needs a frequency of $\pi/2$ for the first dial, $\pi/4$ for the second, $\pi/8$ for the third, and so on.
- Write $M = (\text{sequence_length}, \text{d_model})$,

$$M_{ij} = \sin(2\pi i/2^i) = \sin(x_i \omega_i),$$

where x_i is an integer that gives the position of the sequence and ω_i is the frequency of the dial, which is monotonically decreasing with respect to the embedding dimension.



- The above plot is a smooth curve like we wanted. However, it is also a closed curve. By construction, the position $n+1$ is equivalent to position 1. This was necessary to make our curve continuous, but it is also not what we want!
- To fix this, we observe that there is no reason that we have to have our embedding vector line up with a binary vector. We can lower all frequencies and in this way keep ourselves far away from the boundary,

$$M_{ij} = \sin(x_i \omega_0^{j/d_{\text{model}}}),$$

where ω_0 is the minimal frequency.

Final guess #5: use both sine and cosine to make translation easy

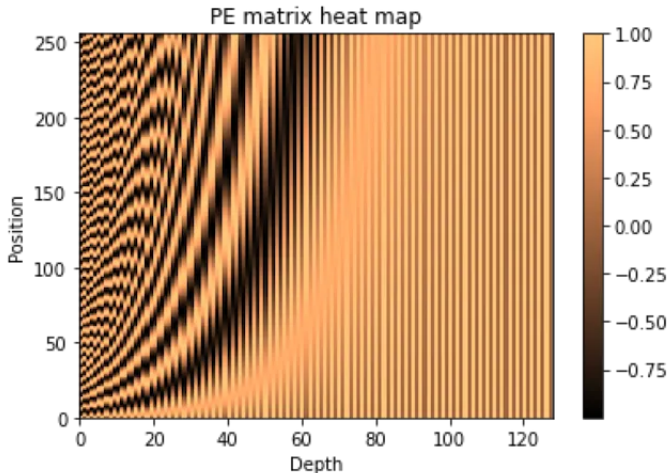
- Our current positional encoding matrix is

$$PE = (v^{(0)}, \dots, v^{(d_{\text{len}}-1)})^T,$$

where $v^{(i)} = [\sin(\omega_0 x_i), \dots, \sin(\omega_{n-1} x_i)]$.

- To build an encoding that supports translations via linear operators, we do the following.
 - ▶ Create a duplicate encoding matrix using cosine instead of sine.
 - ▶ Build a new positional encoding matrix by alternating between the cosine and sine matrices.
 - ▶ This is equivalent to replacing every $\sin(\cdot)$ with a $[\cos(\cdot) \sin(\cdot)]$ pair. In our new PE matrix, the row-vectors look like the following:

$$v^{(i)} = [\cos(\omega_0 x_i), \sin(\omega_0 x_i), \dots, \cos(\omega_{n-1} x_i), \sin(\omega_{n-1} x_i)].$$



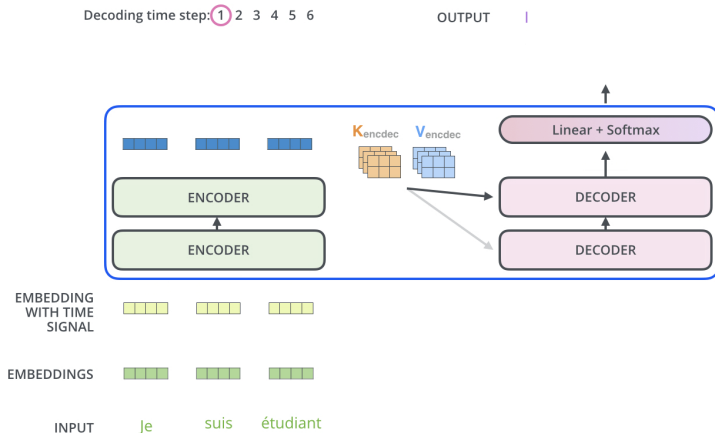
- Most of the PE matrix is not needed for encoding. By following the black curve-ish thing, activating a dial one step deeper along depth becomes exponentially more difficult.
- Vertical lines at greater depth vary less than loss at lower depth.

Table of contents

- 1 Introduction
- 2 Embedding
- 3 Self-Attention
- 4 Multi-Headed Attention
- 5 Positional Encoding
- 6 The Decoder

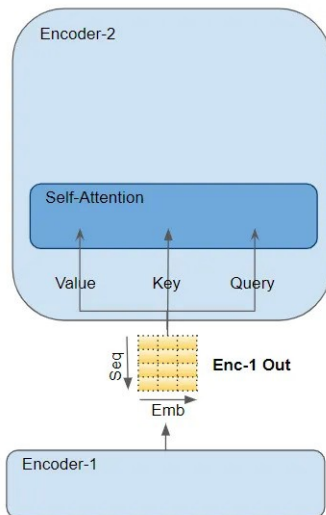
The Decoder

The encoder starts by processing the input sequence. The output of the top encoder is then transformed into a set of attention vectors K and V . These are to be used by each decoder in its “encoder-decoder attention” layer which helps the decoder focus on appropriate places in the input sequence.



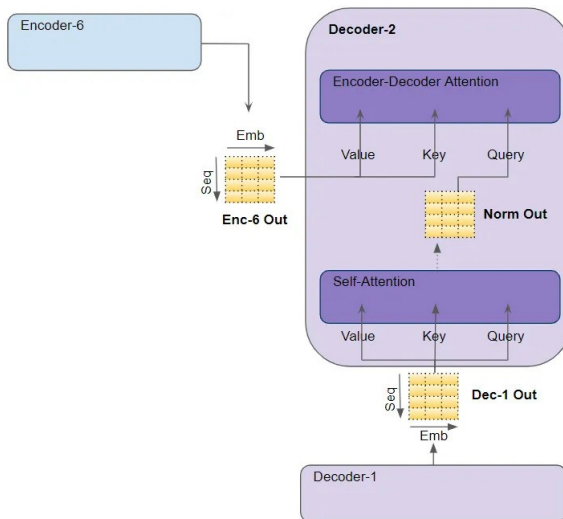
¹<http://jalammar.github.io/illustrated-transformer/>

The Decoder



¹<https://towardsdatascience.com/transformers-explained-visually-part-2-how-it-works-step-by-step-b49fa4a64f34>

The Decoder

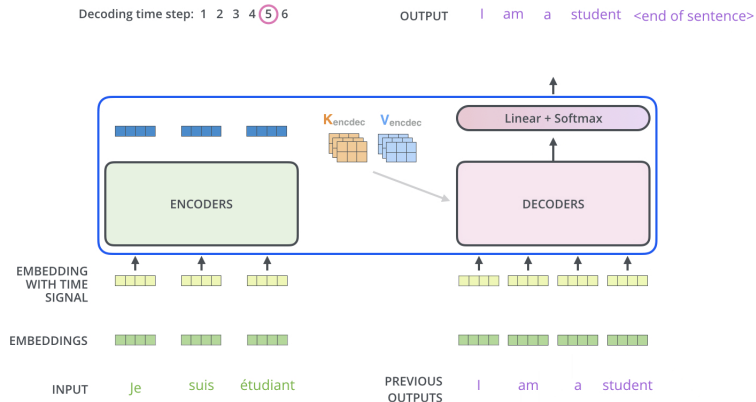


¹<https://towardsdatascience.com/transformers-explained-visually-part-2-how-it-works-step-by-step-b49fa4a64f34>

The Decoder

- The following steps repeat the process 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.
- The self attention layers in the decoder operate in a slightly different way than the one in the encoder:
 - ▶ In the decoder, the self-attention layer is only allowed to attend to earlier positions in the output sequence. This is done by masking future positions (setting them to $-\infty$) before the softmax step in the self-attention calculation.
 - ▶ The “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.

The Decoder



¹<http://jalammar.github.io/illustrated-transformer/>

The Final Linear and Softmax Layer

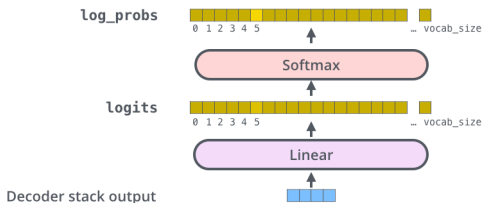
The softmax layer then turns those scores into probabilities (all positive, all add up to 1.0). The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step.

Which word in our vocabulary
is associated with this index?

am

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

5



¹<http://jalammar.github.io/illustrated-transformer/>

References

- 1 <http://jalammar.github.io/illustrated-transformer/>
- 2 <https://towardsdatascience.com/master-positional-encoding-part-i-63c05d90a0c3>
- 3 <https://towardsdatascience.com/transformers-explained-visually-part-2-how-it-works-step-by-step-b49fa>