Natural Language Processing Transformer Architecture

Felipe Bravo-Marquez

June 5, 2023

What’s Wrong with RNNs?

- Given what we just learned on previous lecture, it would seem like attention solves all the problems with RNNs and encoder-decoder architectures[1].
- There are a few shortcomings of RNNs that another architecture called the **Transformer** tries to address.
- The Transformer discards the recursive component of the Encoder-Decoder architecture and purely relies on attention mechanisms [Vaswani et al., 2017].
- When we process a sequence using RNNs, each hidden state depends on the previous hidden state.
- This becomes a major pain point on GPUs: GPUs have a lot of computational capability and they hate having to wait for data to become available.
- Even with technologies like CuDNN, RNNs are painfully inefficient and slow on the GPU.

[1] The following material is based on: `http://mlexplained.com/2017/12/29/attention-is-all-you-need-explained/` and `http://jalammar.github.io/illustrated-transformer/`

Dependencies in neural machine translations

In essence, there are three kinds of dependencies in neural machine translations:

1. Dependencies between the input and output tokens.
2. Dependencies between the input tokens themselves.
3. Dependencies between the output tokens themselves.

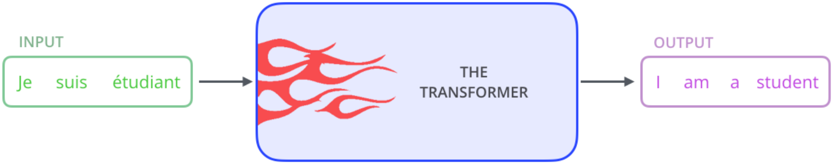
The traditional attention mechanism largely solved the first dependency by giving the decoder access to the entire input sequence. The second and third dependencies were addressed by the RNNs.

The Transformer

- The novel idea of the Transformer is to extend this mechanism to the processing input and output sentences as well.
- The RNN processes input sequences sequentially.
- The Transformer, on the other hand, allows the encoder and decoder to see the entire input sequence all at once.
- This is done using attention.

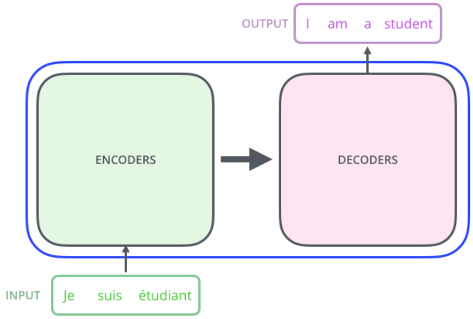
The Transformer

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



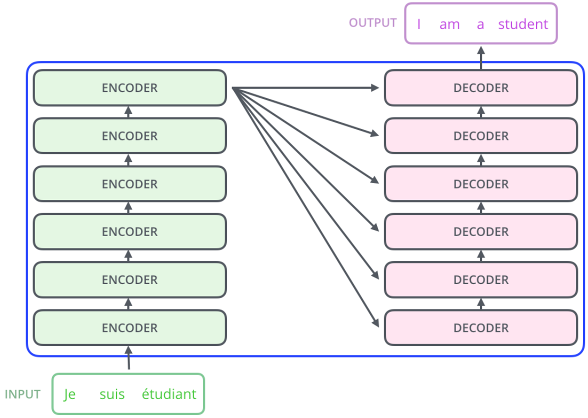
The Transformer

- This blackbox can be decomposed by an encoding component, a decoding component, and connections between them.

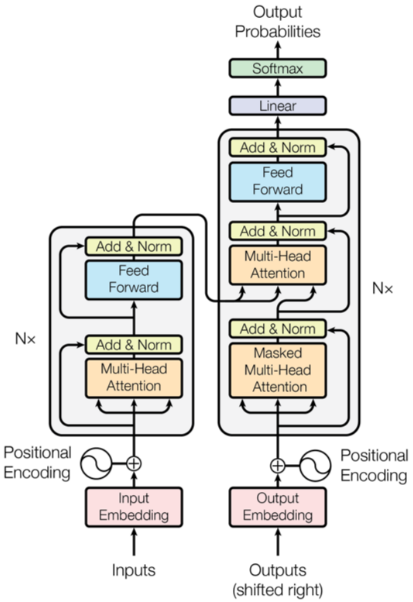


The Transformer

- The encoding component is a stack of encoders.
- The original Transformer stacks six of them on top of each other.
- But 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 Transformer



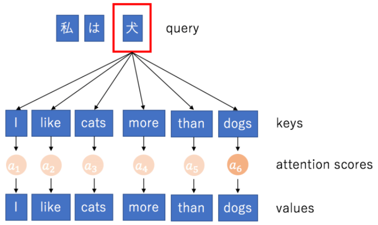
The Transformer

- The Transformer still uses the basic encoder-decoder design of RNN neural machine translation systems.
- The left-hand side is the encoder, and the right-hand side is the decoder.
- The initial inputs to the encoder are the embeddings of the input sequence.
- The initial inputs to the decoder are the embeddings of the outputs up to that point.
- The encoder and decoder are composed of N blocks (where $N = 6$ for both networks).
- These blocks are composed of smaller blocks as well.
- Before looking at each block in further detail, let't try to understand the Attention mechanism implemented by the Transformer.

Attention Mechanism in the Transformer

- The attention mechanism in the Transformer is interpreted as a way of computing the relevance of a set of **values** (information) based on some **keys** and **queries**.
- The attention mechanism is used as a way for the model to **focus on relevant information** based on what it is currently processing.
- In the RNN encoder-decoder architecture with attention:
    1. Attention weights were the relevance of the encoder hidden states (values) in processing the decoder state (query).
    2. These values were calculated based on the encoder hidden states (keys) and the decoder hidden state (query).

Attention Mechanism in the Transformer



- In this example, the query is the word being decoded (which means dog) and both the keys and values are the source sentence.
- The attention score represents the relevance, and in this case is large for the word “dog” and small for others.
- When we think of attention this way, we can see that the keys, values, and queries could be anything.
- They could even be the same.
- For instance, both values and queries could be input embeddings (self attention).

Queries

- Queries are representations of the target or output sequence that the Transformer model uses to determine how much attention should be given to each word in the input sequence.
- Each word or token in the output sequence is associated with a query.
- The queries help in retrieving relevant information from the input sequence.
- Example: If we are generating the translation for the sentence “Je adore les chats” (meaning ”I love cats” in French), each word in the output sequence would have its corresponding query representation.
- For example, “Je” would have a query vector, “adore” would have a query vector, and so on.

Keys

- Keys are representations of the input sequence that the Transformer model uses to compute the attention scores.
- Each word or token in the input sequence is associated with a key.
- These keys capture the information needed to understand the context and relationships between words in the sequence.
- Example: Consider the input sequence: “I love cats.”
- Each word in the sequence would have its corresponding key representation, such as “I” having a key vector, “love” having a key vector, and so on.

Values

- Values are the actual information or features associated with each word in the input sequence.
- These values are used to calculate the weighted sum during the attention computation, which helps determine the importance or relevance of each word.
- Example: Considering the same input sequence “I love cats,” each word would have its associated value representation.
- These values contain the contextual information for each word.
- For instance, “I” would have a value vector, “love” would have a value vector, and so forth.
- Keys and values are very difficult to distinguish at first sight because in RNN encoding and decoding with classical attention they are the same.
- We will see with the Transformer that although they come from the same sequence, they may correspond to different vectors.

Scaled Dot Product Attention

- Transformer uses a particular form of attention called the "Scaled Dot-Product Attention":
- For a given query vector $\vec{q}$, a sequence of key vectors $\vec{k}_{1:m}$, and a sequence of value vectors $\vec{v}_{1:m}$, the attention weights $\alpha_1, \dots, \alpha_m$ are computed as follows:

$$\alpha_1, \dots, \alpha_m = \text{softmax}\left(\frac{\vec{q} \cdot \vec{k}_1}{\sqrt{d}}, \dots, \frac{\vec{q} \cdot \vec{k}_1}{\sqrt{d}}\right)$$

- d represents the dimensionality of the queries and keys.
- The normalization over $\sqrt{d}$ is used to rescale the dot products between queries and keys (dot products tend to grow with the dimensionality).
- The attention weights are then multiplied by their corresponding values to compute a weighted sum, which is then passed to the subsequent layers of the network:

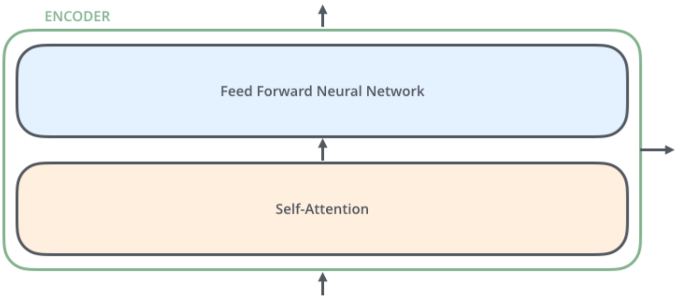
$$\alpha_1 * \vec{v}_1 + \dots + \alpha_m * \vec{v}_m$$

- Now we are ready to take a closer look at each part of the Transformer.

The Encoder

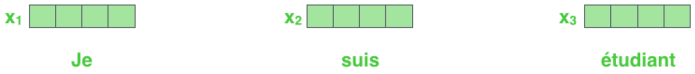
- The encoder contains self-attention layers.
- In a self-attention layer all of the keys, values and queries come from the same place, in this case, the output of the previous layer in the encoder.
- Each position in the encoder can attend to all positions in the previous layer of the encoder.
- The encoder is composed of two blocks (which we will call sub-layers to distinguish from the N blocks composing the encoder and decoder).
- One is the Multi-Head Attention sub-layer over the inputs, mentioned above.
- The other is a simple feed-forward network.

The Encoder



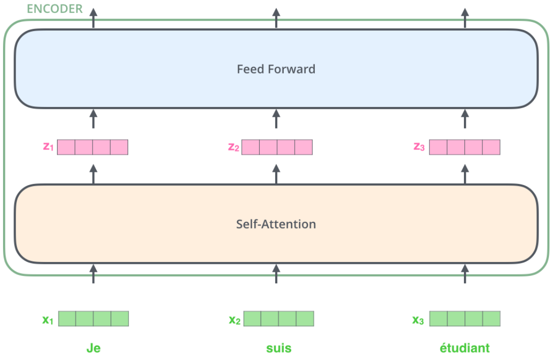
Bringing The Tensors Into The Picture

- We begin by turning each input word into a vector using an embedding layer of the size 512 in the bottom-most encoder.



- In the bottom encoder that would be the word embeddings, but in other encoders, it would be the output of the encoder that’s directly below.
- The size of this list is hyperparameter we can set – basically it would be the length of the longest sentence in our training dataset.

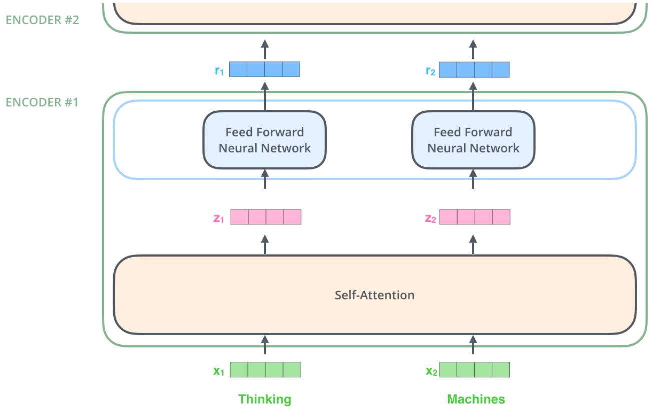
Bringing The Tensors Into The Picture



- 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.

Now We’re Encoding!

- As we’ve mentioned already, 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.

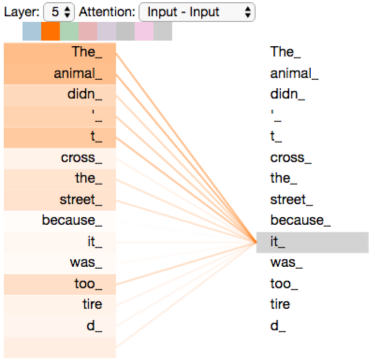


Self-Attention at a High Level

- 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

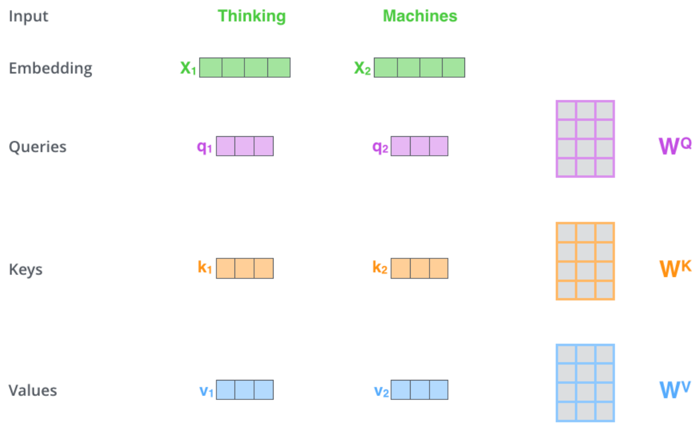
- Think of how maintaining a hidden state allows an RNN to incorporate its representation of previous words/vectors it has processed with the current one it's processing.
- Self-attention is the method the Transformer uses to bake the "understanding" of other relevant words into the one we're currently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



Self-Attention in Detail: step 1

- The **first step** in calculating scaled dot product 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.
- 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.
- They don't HAVE to be smaller, this is an architecture choice to make the computation of multiheaded attention (mostly) constant.

Self-Attention in Detail: step 1

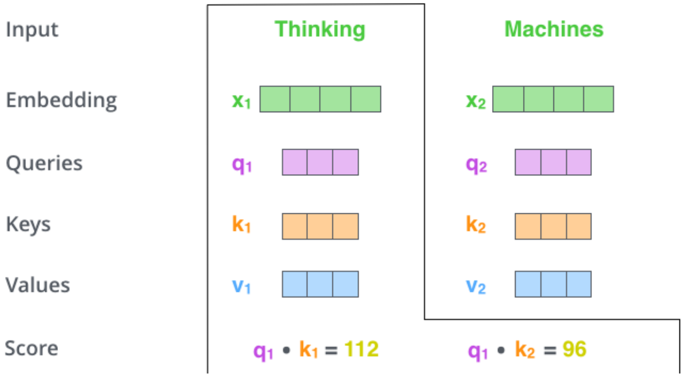


- Example sentence: “Thinking Machines”.
- Multiplying x_1 by the WQ weight matrix produces q_1, the “query” vector associated with that word.
- We end up creating a “query”, a “key”, and a “value” projection of each word in the input sentence.

Self-Attention in Detail: step 2

- The **second step** in calculating self-attention is to calculate a score.
- Say we're 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 determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.
- The score is calculated by taking the dot product of the query vector with the key vector of the respective word we're scoring.
- So if we're processing the self-attention for the word in position #1, the first score would be the dot product of q1 and k1.
- The second score would be the dot product of q1 and k2.

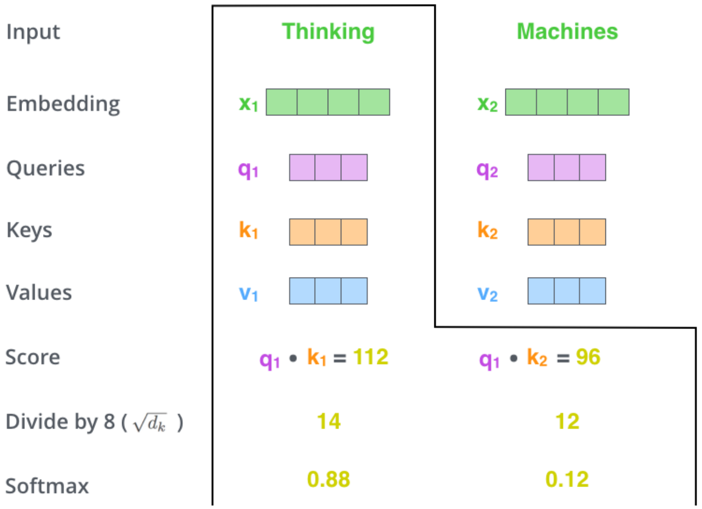
Self-Attention in Detail: step 2



Self-Attention in Detail: steps 3 and 4

- 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.
- This softmax score determines how much each word will be expressed at the current position.
- Clearly the word at the current position will have the highest softmax score, but sometimes it’s useful to attend to another word that is relevant to the current word.

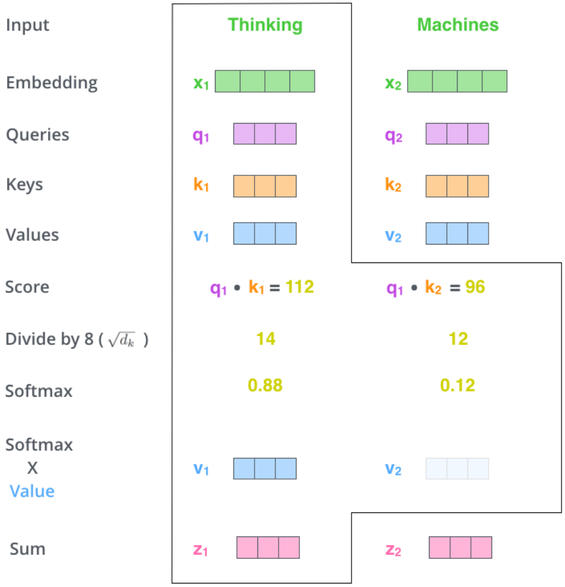
Self-Attention in Detail: steps 3 and 4



Self-Attention in Detail: steps 5 and 6

- The **fifth step** is to multiply each value vector by the softmax score (in preparation to sum them up).
- 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 like 0.001, for example).
- The **sixth step** is to sum up the weighted value vectors.
- This produces the output of the self-attention layer at the current position (for the first word in this example).
- That concludes the scaled dot product self-attention calculation.
- The resulting vector is one we can send along to the feed-forward neural network.

Self-Attention in Detail: steps 5 and 6

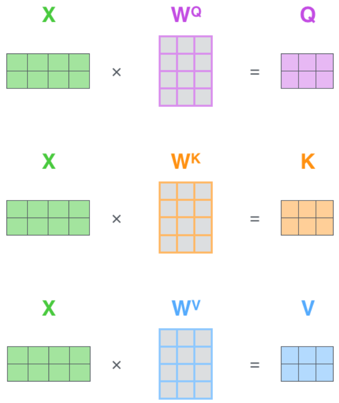


Matrix Calculation of Self-Attention

- In the actual implementation, scaled dot product self-attention is computed in matrix form for faster processing.
- So let's look at that now that we've seen the intuition of the calculation on the word level.
- The first step is to calculate the Query, Key, and Value matrices: Q, K, V.
- We do that by packing our embeddings into a matrix X, and multiplying it by the weight matrices we've trained (WQ, WK, WV).

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

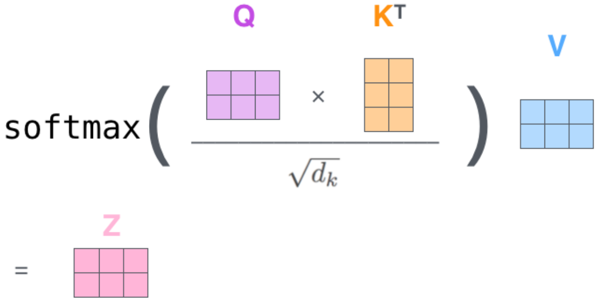
Matrix Calculation of Self-Attention



- Every row in the X matrix corresponds to a word in the input sentence.
- We again see the difference in size of the embedding vector (512, or 4 boxes in the figure), and the q/k/v vectors (64, or 3 boxes in the figure).

Matrix Calculation of Self-Attention

- Finally, 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.

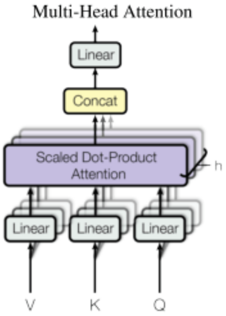


Multi-Head Attention

- If we only compute a single attention weighted sum of the values, it would be difficult to capture various different aspects of the input.
- In the example above, $z1$ contains a little bit of every other encoding, but it could be dominated by the actual word itself.
- If we're translating a sentence like “The animal didn't cross the street because it was too tired”, it would be useful to know which word “it” refers to.
- To solve this problem, the Transformer uses the Multi-Head Attention block.
- This component expands the model's ability to focus on different positions.
- Multi-head attention computes multiple attention weighted sums instead of a single attention pass over the values.
- Hence the name “Multi-Head” Attention.
- To learn diverse representations, the Multi-Head Attention applies different linear transformations to the values, keys, and queries for each “head” of attention.

Multi-Head Attention

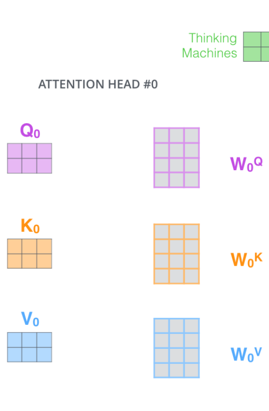
- The Multi-Head Attention block applies multiple blocks in parallel, concatenates their outputs, then applies one single linear transformation.



Multi-Head Attention

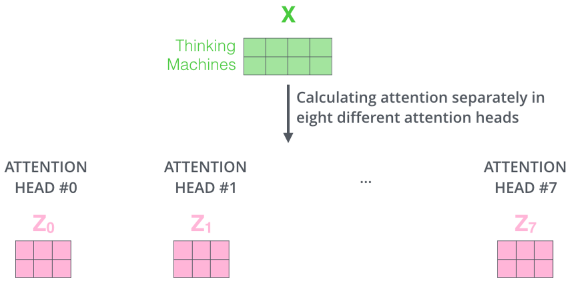
- The multi-headed component gives the attention layer multiple “representation subspaces”.
- With multi-headed attention, we have not only one, but multiple sets of Query/Key/Value weight matrices.
- The Transformer uses eight attention heads, so we end up with eight sets for each encoder/decoder.
- Each of these sets is randomly initialized.
- Then, after training, each set is used to project the input embeddings (or vectors from lower encoders/decoders) into a different representation subspace.

Multi-Head Attention



- With multi-headed attention, we maintain separate Q/K/V weight matrices for each head resulting in different Q/K/V matrices.
- As we did before, we multiply X by the WQ/WK/WV matrices to produce Q/K/V matrices.

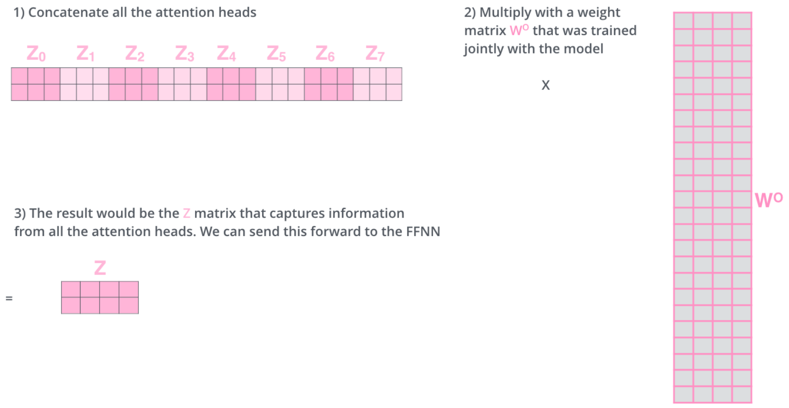
Multi-Head Attention



- 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.
- This leaves us with a bit of a challenge.
- 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.

Multi-Head Attention

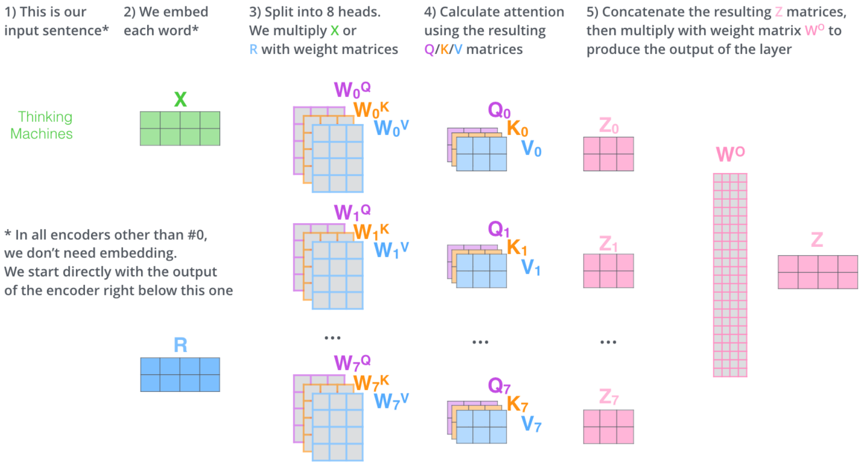
- How do we do that? We concat the matrices then multiply them by an additional weights matrix WO.



1) Concatenate all the attention heads
Z_0 Z_1 Z_2 Z_3 Z_4 Z_5 Z_6 Z_7
2) Multiply with a weight matrix W^O that was trained jointly with the model
X
W^O
3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN
=
Z

Multi-Head Attention

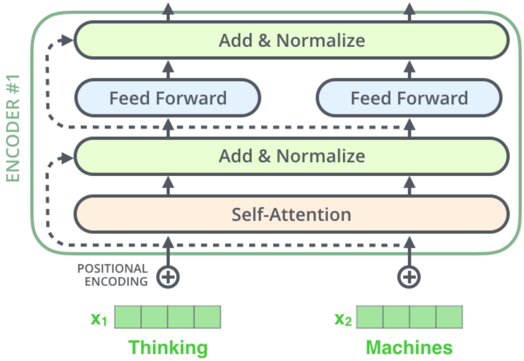
- Let’s put all these matrices in one visual so we can look at them in one place.



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
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

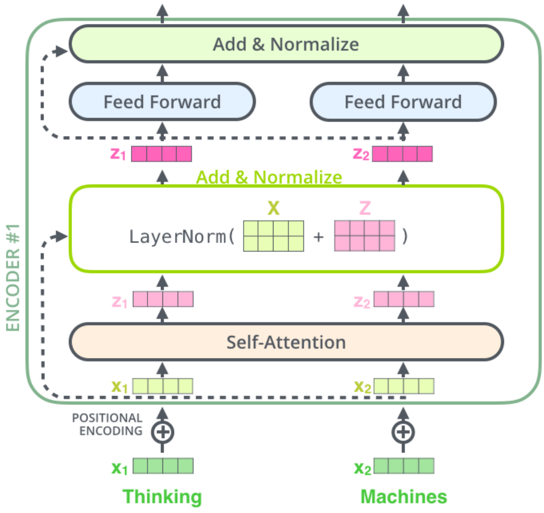
Residual Connections

- Between each sub-layer, there is a residual connection followed by a layer normalization.
- A residual connection is basically just taking the input and adding it to the output of the sub-network, and is a way of making training deep networks easier.
- Layer normalization is a normalization method in deep learning that is similar to batch normalization.

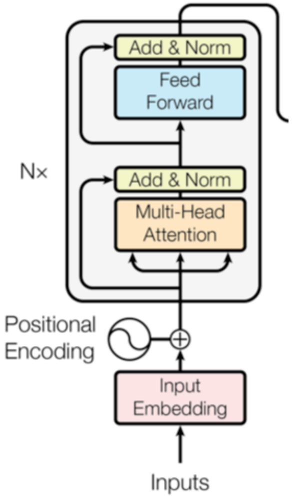


Residual Connections

- If we're to visualize the vectors and the layer-norm operation associated with self attention, it would look like this:



The Encoder: summary



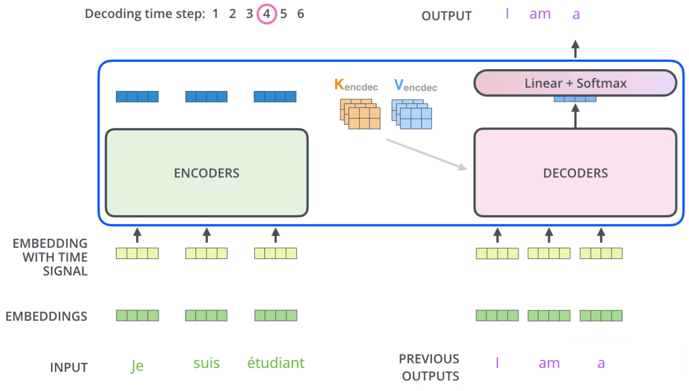
The Encoder: summary

- What each encoder block is doing is actually just a bunch of matrix multiplications followed by a couple of element-wise transformations.
- This is why the Transformer is so fast: everything is just parallelizable matrix multiplications.
- The point is that by stacking these transformations on top of each other, we can create a very powerful network.
- The core of this is the attention mechanism which modifies and attends over a wide range of information.

The Decoder

- The Transformer decoder utilizes the output of the top encoder to generate attention vectors K and V.
- These attention vectors are then employed by each decoder in its “encoder-decoder attention” layer to enable the decoder to focus on relevant sections in the input sequence.
- By doing so, every position in the decoder is able to attend to all positions in the input sequence, replicating the typical encoder-decoder attention mechanisms found in sequence-to-sequence models.
- Additionally, self-attention layers within the decoder allow each position to attend to all positions within the decoder, including itself, similar to the role played by the decoder hidden state in RNN machine translation architectures.
- In summary, 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

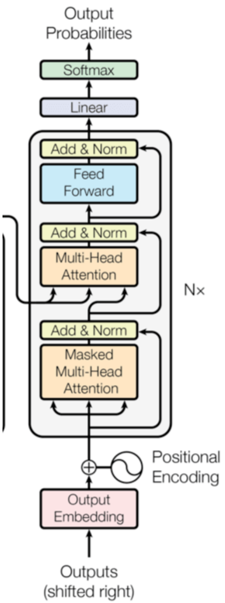


- Each step in the decoding phase outputs an element from the output sequence (the English translation sentence in this case).
- This process is repeated until a special symbol is reached indicating the transformer decoder has completed its output.

The Decoder

- When we train the Transformer, we want to process all the sentences at the same time.
- However, if we give the decoder access to the entire target sentence, the model can just repeat the target sentence (in other words, it doesn’t need to learn anything).
- The self-attention layer should only be allowed to attend to earlier positions in the output sequence.
- This is done by masking the “future” tokens when decoding a certain word.
- The masking is done by setting to $-\infty$ all values in the input of the softmax which correspond to illegal connections.
- This is why “multi-head attention blocks” in the decoder are referred to as “masked”: the inputs to the decoder from future time-steps are masked.

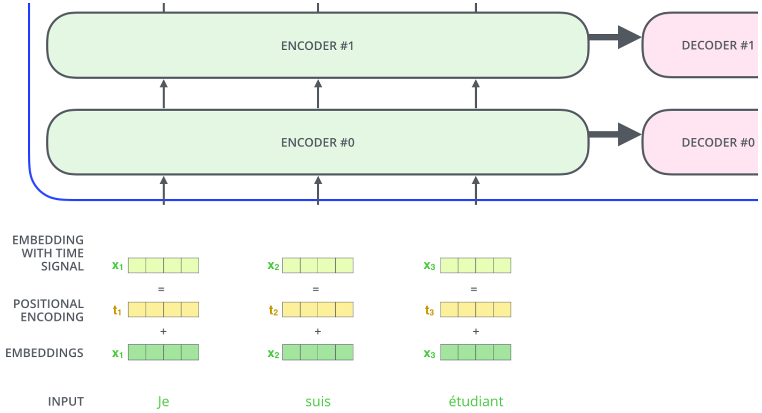
The Decoder


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Add & Norm
Masked Multi-Head Attention
Positional Encoding
Output Embedding
Outputs (shifted right)

Positional Encodings

- Unlike recurrent networks, the multi-head attention network cannot naturally make use of the position of the words in the input sequence.
- Without positional encodings, the output of the multi-head attention network would be the same for the sentences “I like cats more than dogs” and “I like dogs more than cats”.
- Positional encodings explicitly encode the relative/absolute positions of the inputs as vectors and are then added to the input embeddings.

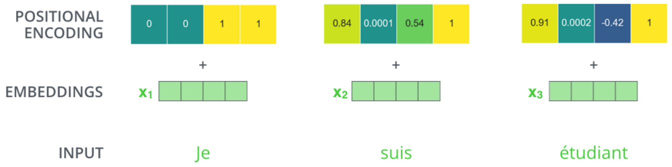
Positional Encodings



- To give the model a sense of the order of the words, we add positional encoding vectors – the values of which follow a specific pattern.

Positional Encodings

- The paper uses the following equation to compute the positional encodings:
  $PE(pos, 2i) = \sin(pos/10000^{2i/d_{model}})$
  $PE(pos, 2i+1) = \cos(pos/10000^{2i/d_{model}})$
- Where *pos* represents the position, and *i* is the dimension.
- Basically, each dimension of the positional encoding is a wave with a different frequency.



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

Conclusions

- The Transformer achieves better BLUE scores than previous state-of-the-art models for English-to-German translation and English-to-French translation at a fraction of the training cost.

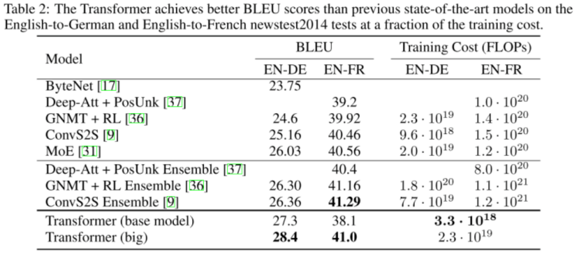


Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [17]	23.75			
Deep-Att + PosUnk [37]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [36]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [31]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [37]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [36]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$\mathbf{3.3 \cdot 10^{18}}$	
Transformer (big)	**28.4**	**41.0**	$2.3 \cdot 10^{19}$	

- The Transformer is a powerful and efficient alternative to recurrent neural networks to model dependencies using only attention mechanisms.
- A very illustrative blog post about the Transformer: `http://jalammar.github.io/illustrated-transformer/`.

Questions?

Thanks for your Attention!

References I

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017).
Attention is all you need.
Advances in neural information processing systems, 30.