Transformer – Attention Is All You Need

Table of Contents

1. Introduction

Recurrent neural networks, LSTM and GRU neural networks in particular, have been firmly established as state-of-the-art approaches in sequence modeling and transduction problems such as language modeling and machine translation.

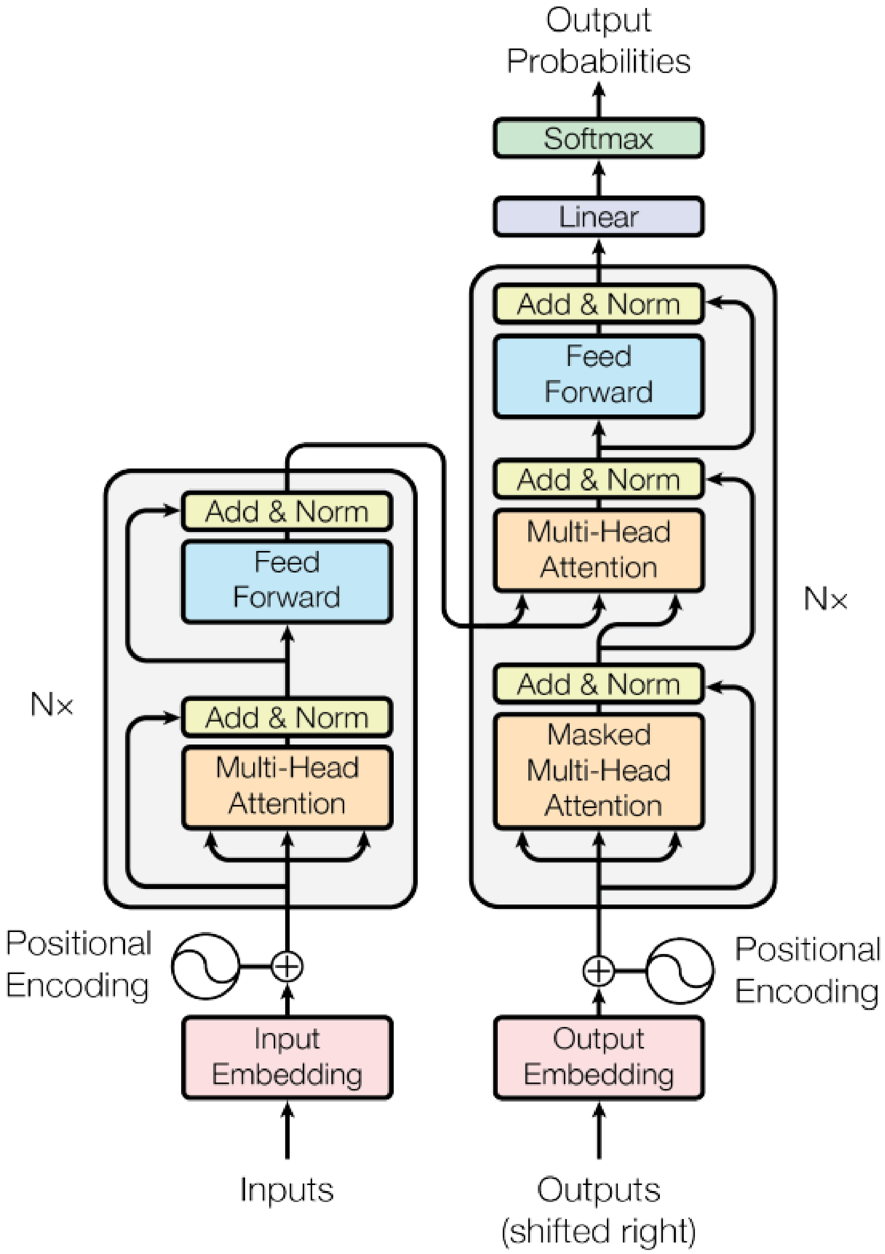
Numerous efforts have since continued to push the boundaries of recurrent language models and encoder-decoder architectures. This inherently sequential nature precludes parallelization within training examples, which becomes critical at longer sequence lengths, as memory constraints limit batching across examples.

Attention mechanisms have become an integral part of compelling sequence modeling and transduction models in various tasks, allowing modeling of dependencies without regard to their distance in the input or output sequences. [1]

Transformers are a type of artificial neural network architecture that is used to solve the problem of transduction or transformation of input sequences into output sequences in deep learning applications. some of the examples of sequence data can be something like time, series, speech, text, financial data, audio, video, weather, and many more.

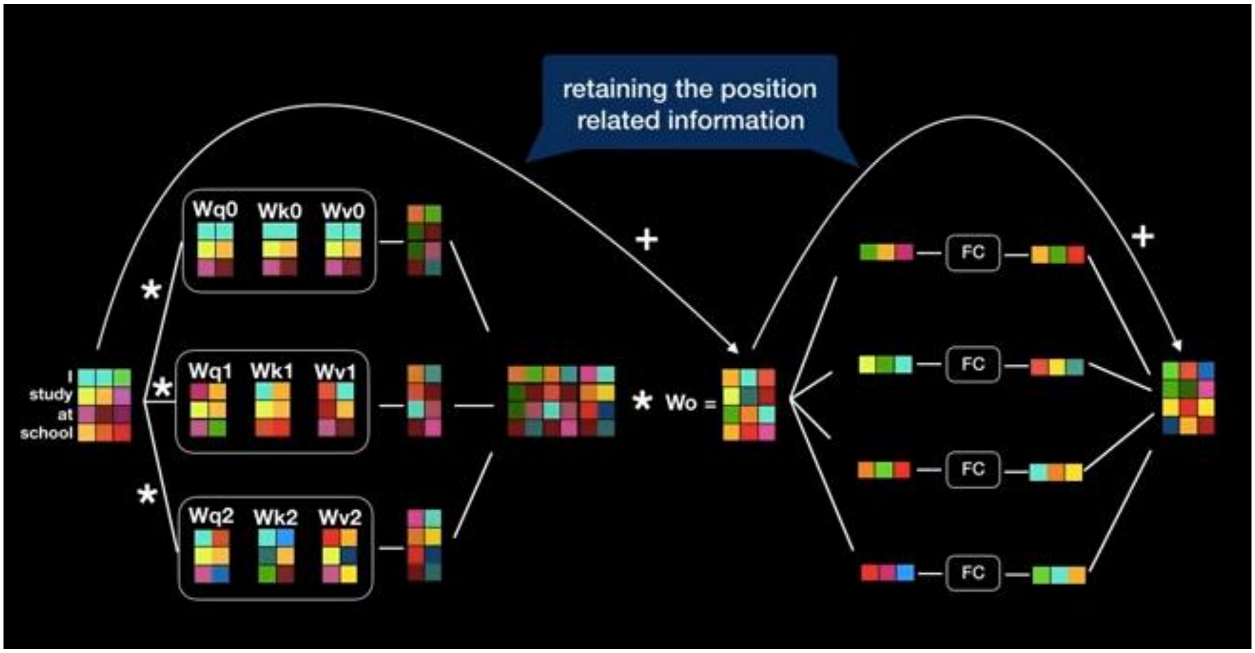
Transformer models apply an evolving set of mathematical techniques, called attention or self-attention, to detect subtle ways even distant data elements in a series influence and depend on each other.

2. Architecture

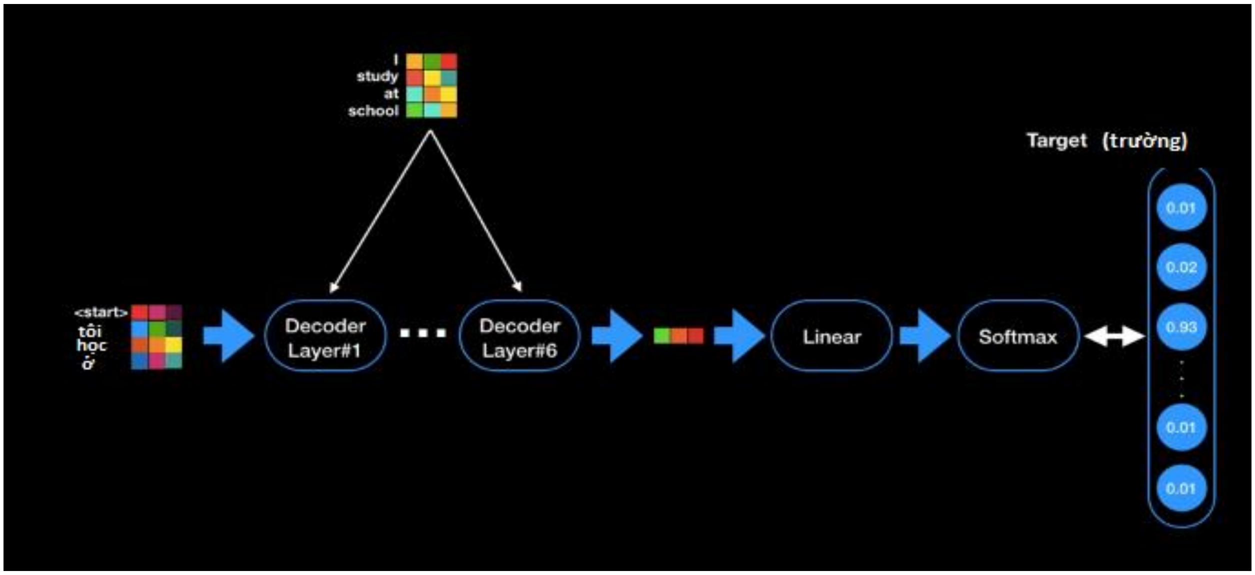


Encoder: The encoder is composed of a stack of N identical layers. Each layer has two sub-layers. [1]

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



Decoder: The decoder is also composed of a stack of N identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs **multi-head attention** over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i. [1]



2.1 Encoder

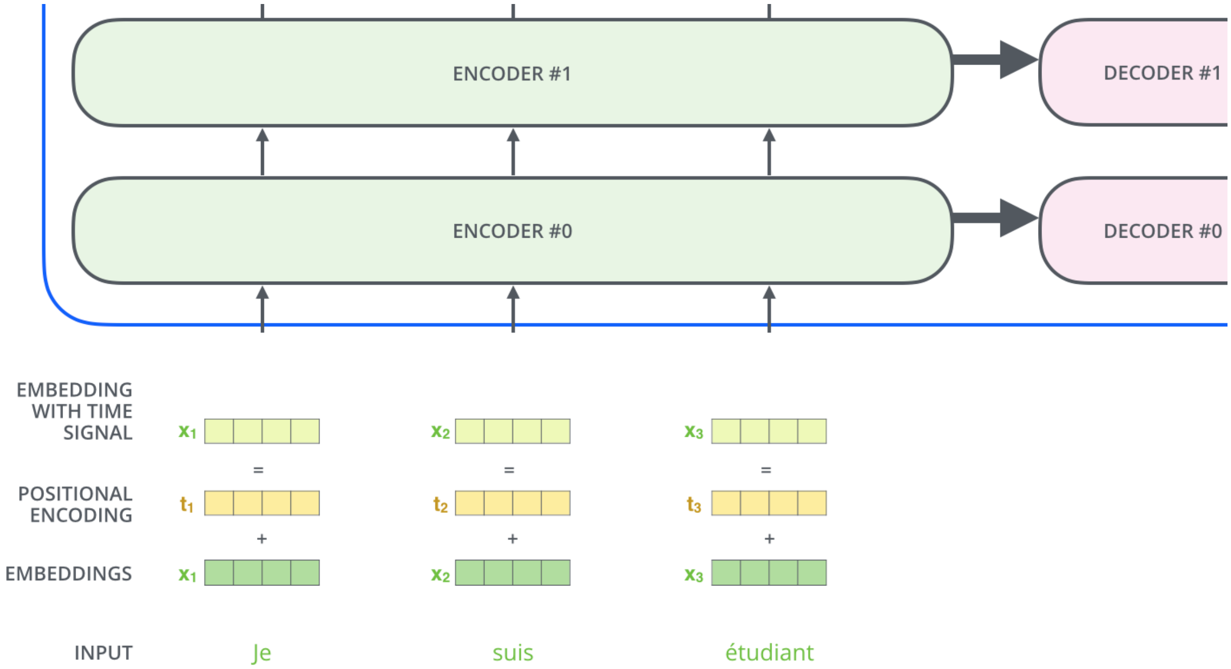
2.1.1 Positional Enconding

Since Transformer model contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the sequence. To this end, we add "positional encodings" to the input embeddings at the bottoms of the encoder and decoder stacks. The positional encodings have the same dimension dmodel as the embeddings, so that the two can be summed. There are many choices of positional encodings, learned and fixed [1, 4]

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

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

where pos is the position and i is the dimension.



2.1.2 Self - Attention

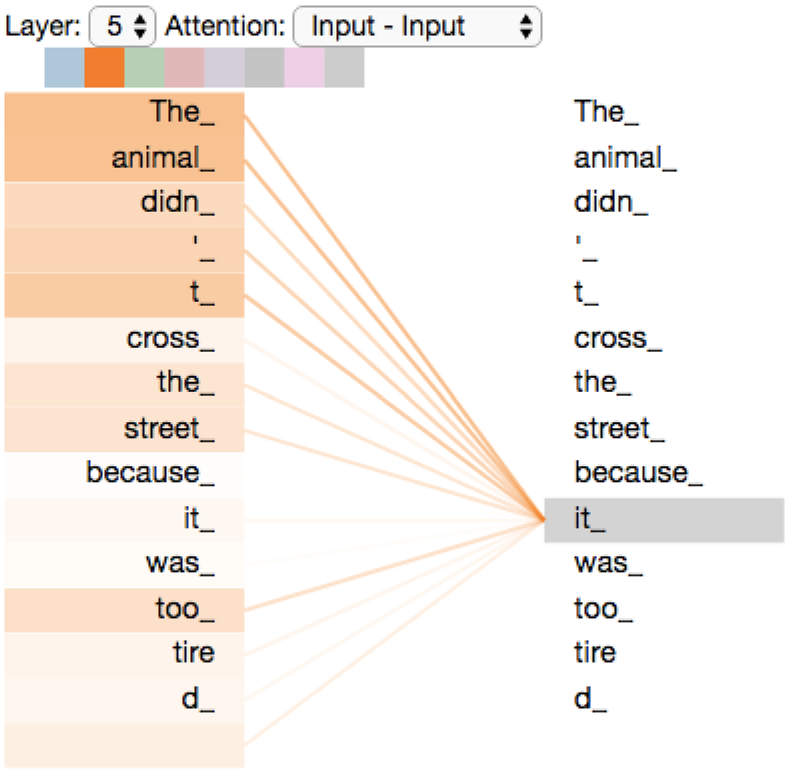
Say the following sentence is an input sentence we want to translate:

”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”.

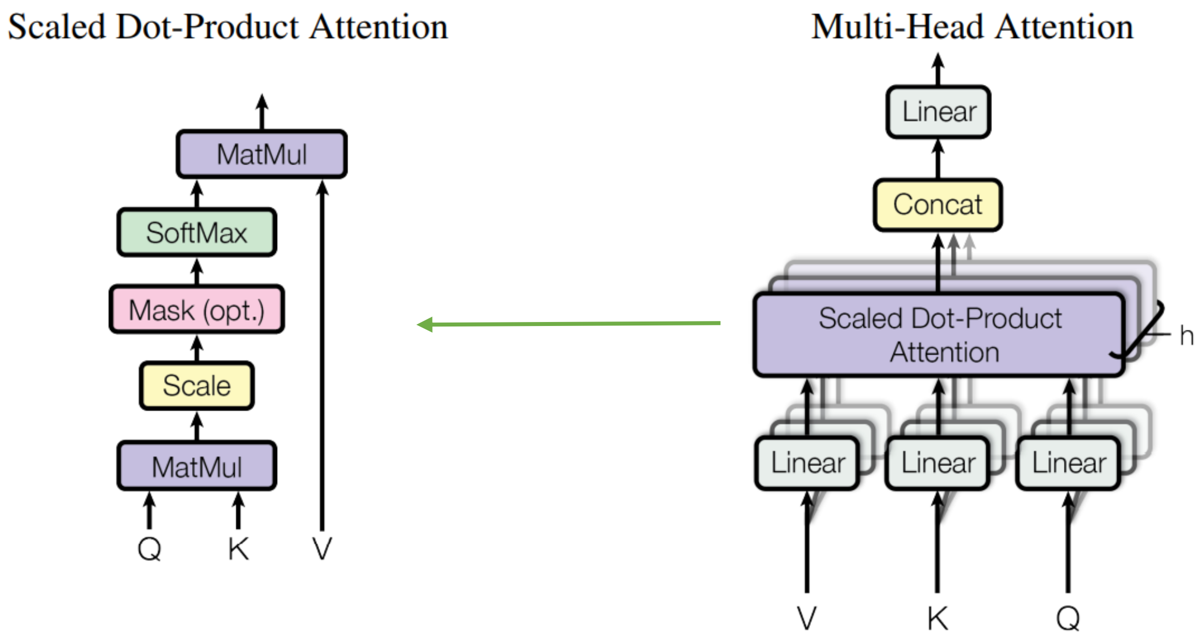
As the model processes each word (each position in the input sequence), self attention allows it to look at other positions in the input sequence for clues that can help lead to a better encoding for this word. Self-attention is the method the Transformer uses to bake the “understanding” of other relevant words into the one we’re currently processing. [3]



Layer: 5 Attention: Input - Input

An attention function can be described as each input mapping by multiplying the embedding by three matrices that we trained during the training process a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors.

The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key. [1]



2.1.3 Scaled Dot-Product Attention

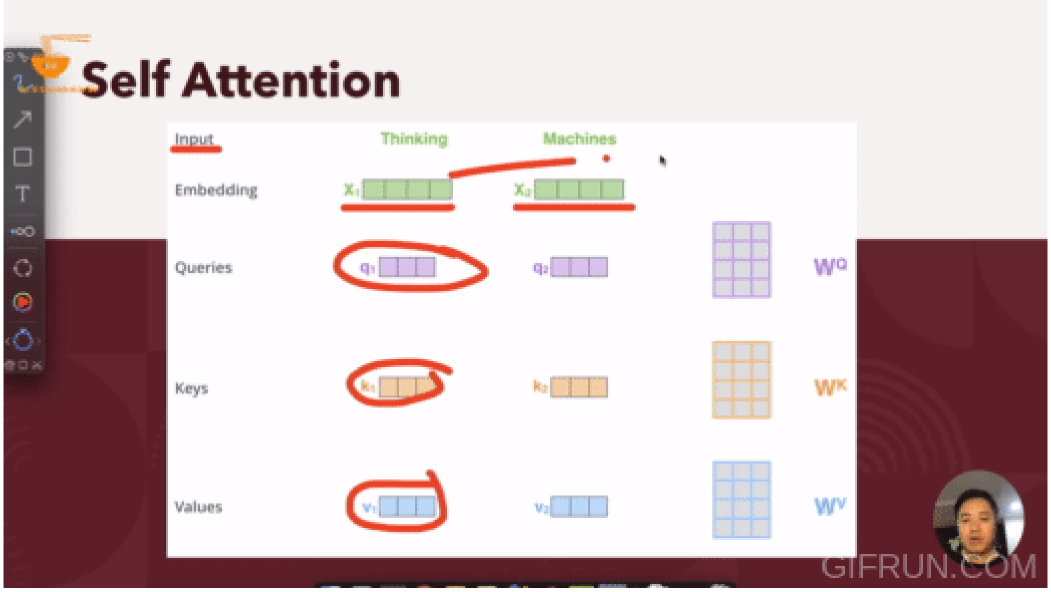
The input consists of queries and keys of dimension d$_k$, and values of dimension d$_v$We compute the dot products of the query with all keys, divide each by $\sqrt{dk}$, and apply a softmax function to obtain the weights on the values [1]

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

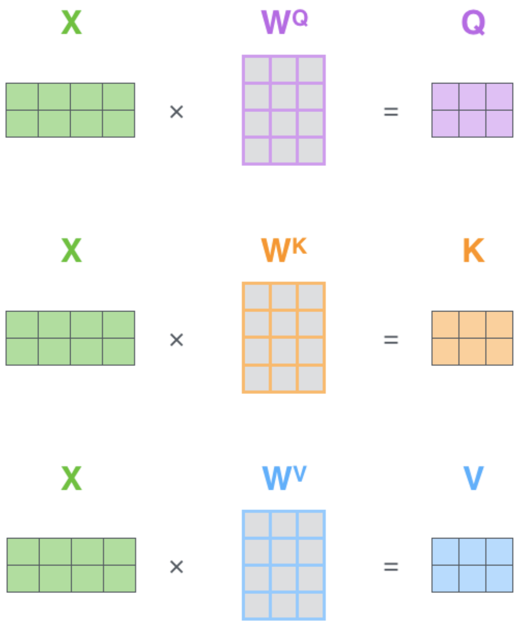
We suspect that for large values of d$_k$, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients 4.

To counteract this effect, we scale the dot products by $\sqrt{dk}$. The scaling factor, therefore, serves to pull the results generated by the dot product multiplication down, preventing this problem. [1]

For example, assumed that we have input "Thinking Machines", create three vectors Q, K, V (each of these vectors is randomly initialized and fine-tuning while training) from each of the encoder's input vectors (in this case, the embedding of each word) [3]

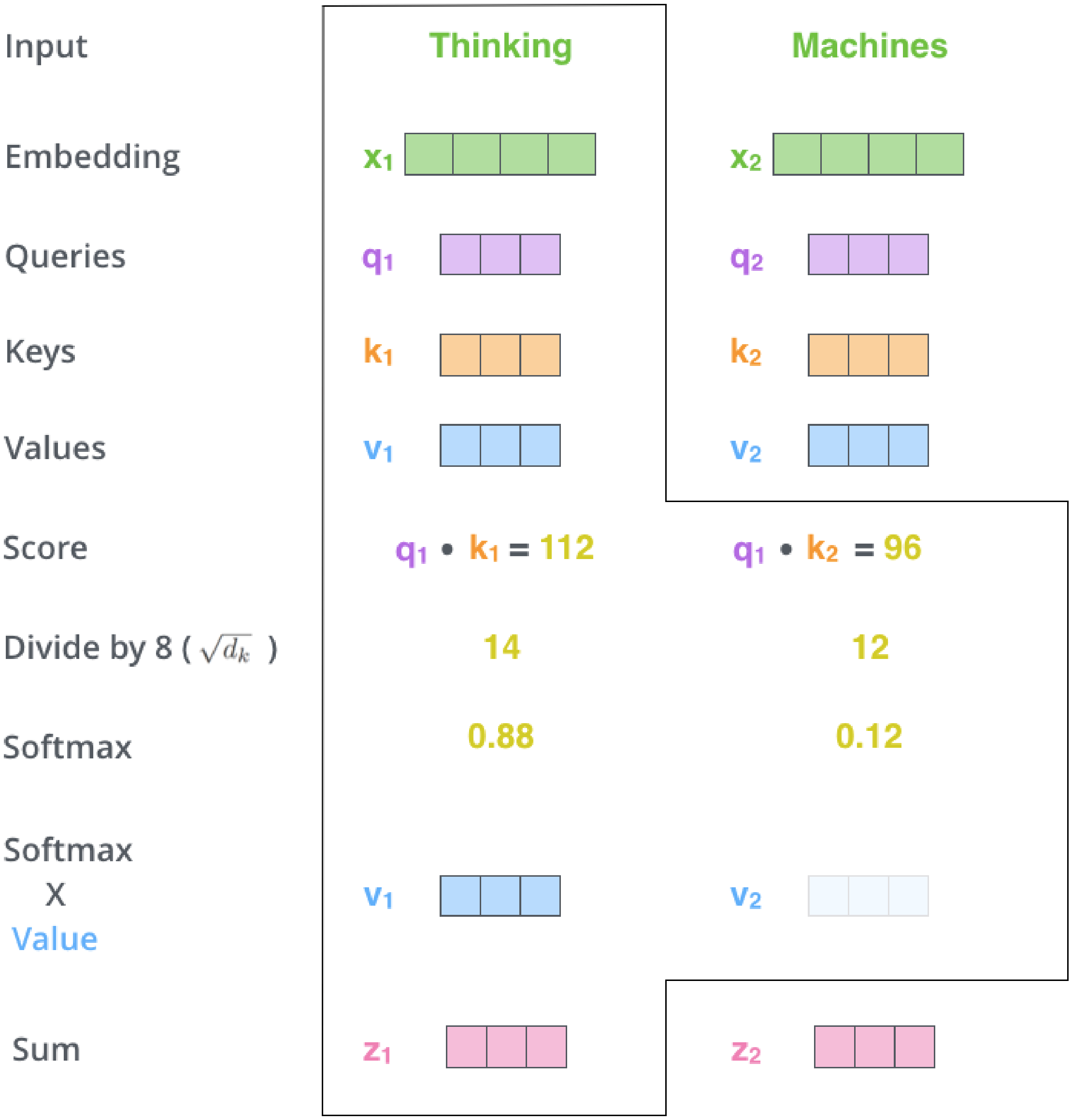


Self Attention

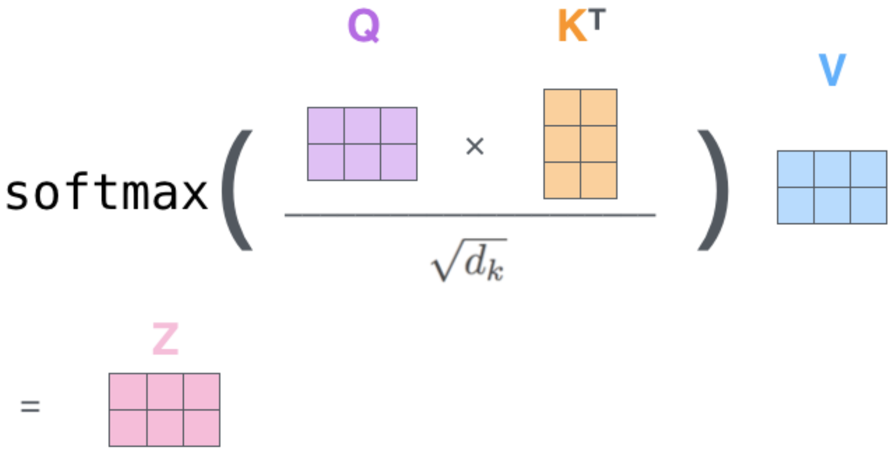


Multiplying x1 by the WQ weight matrix produces q1, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

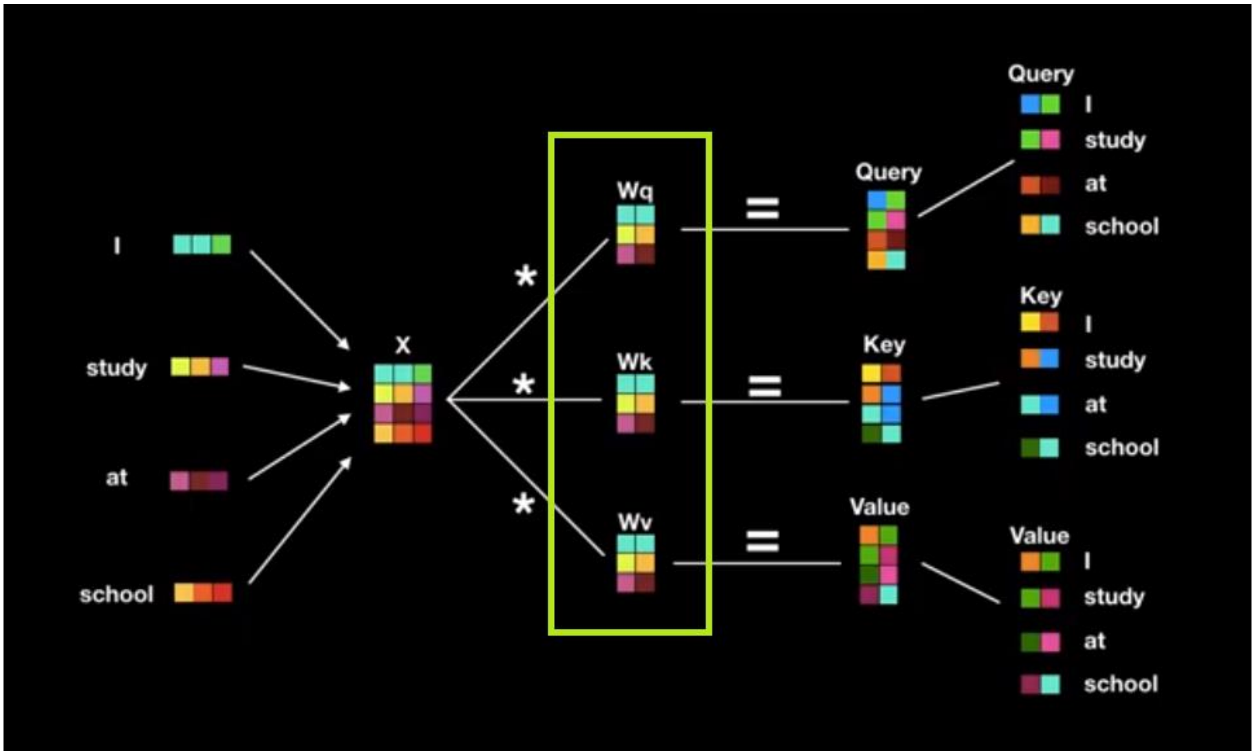
Calculating self-attention is to calculate a score following Attention formula above. Say we’re calculating the self-attention for the first word in this example, “Thinking”. We need to score each word of the input sentence against this word. The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position. [3]

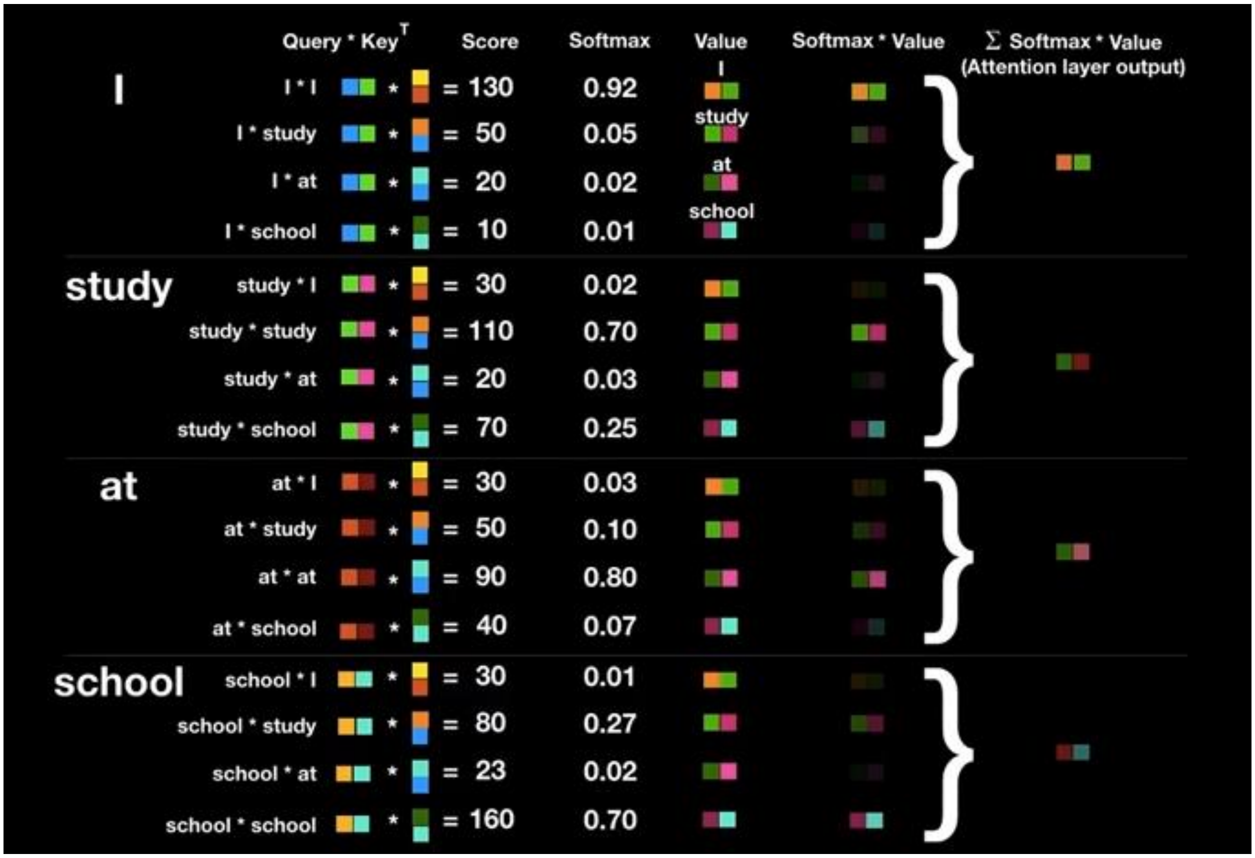


Softmax normalizes the scores so they're in [0; 1]. This softmax score determines how much each word will be expressed at this position. Clearly the word at this position will have the highest softmax score, but sometimes it's useful to attend to another word that is relevant to the current word. Then, 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). [3]



Finally, sum up the weighted value vectors. This produces the output of the self-attention layer at this position (for the first word).





	Query * Key^T	Score	Softmax	Value	Softmax * Value	Σ Softmax * Value (Attention layer output)
I	I * I 🟦🟩 * 🟨🟧	= 130	0.92	I 🟧🟩	🟧🟩	🟧🟩
	I * study 🟦🟩 * 🟧🟦	= 50	0.05	study 🟩🟪	⬛⬛	
	I * at 🟦🟩 * 🟦🟦	= 20	0.02	at 🟩🟪	⬛⬛	
	I * school 🟦🟩 * 🟩🟦	= 10	0.01	school 🟪🟦	⬛⬛	
study	study * I 🟩🟪 * 🟨🟧	= 30	0.02	🟧🟩	⬛⬛	🟩🟥
	study * study 🟩🟪 * 🟧🟦	= 110	0.70	🟩🟪	🟩🟪	
	study * at 🟩🟪 * 🟦🟦	= 20	0.03	🟩🟪	⬛⬛	
	study * school 🟩🟪 * 🟩🟦	= 70	0.25	🟪🟦	🟪🟦	
at	at * I 🟧🟥 * 🟨🟧	= 30	0.03	🟧🟩	⬛⬛	🟩🟪
	at * study 🟧🟥 * 🟧🟦	= 50	0.10	🟩🟪	⬛⬛	
	at * at 🟧🟥 * 🟦🟦	= 90	0.80	🟩🟪	🟩🟪	
	at * school 🟧🟥 * 🟩🟦	= 40	0.07	🟪🟦	⬛⬛	
school	school * I 🟨🟦 * 🟨🟧	= 30	0.01	🟧🟩	⬛⬛	🟥🟦
	school * study 🟨🟦 * 🟧🟦	= 80	0.27	🟩🟪	🟩🟪	
	school * at 🟨🟦 * 🟦🟦	= 23	0.02	🟩🟪	⬛⬛	
	school * school 🟨🟦 * 🟩🟦	= 160	0.70	🟪🟦	🟪🟦	

2.1.4 Multi-Head Attention

Instead of performing a single attention function with d_{model}-dimensional **keys, values and queries**, we found it beneficial to linearly project the queries, keys and values h times with different, learned linear projections to d_Q, d_k and d_v dimensions, respectively. On each of these projected versions of queries, keys, and values we then perform the attention function in parallel, yielding d_v-dimensional output values. These are concatenated and once again projected, resulting in the final values. [1]

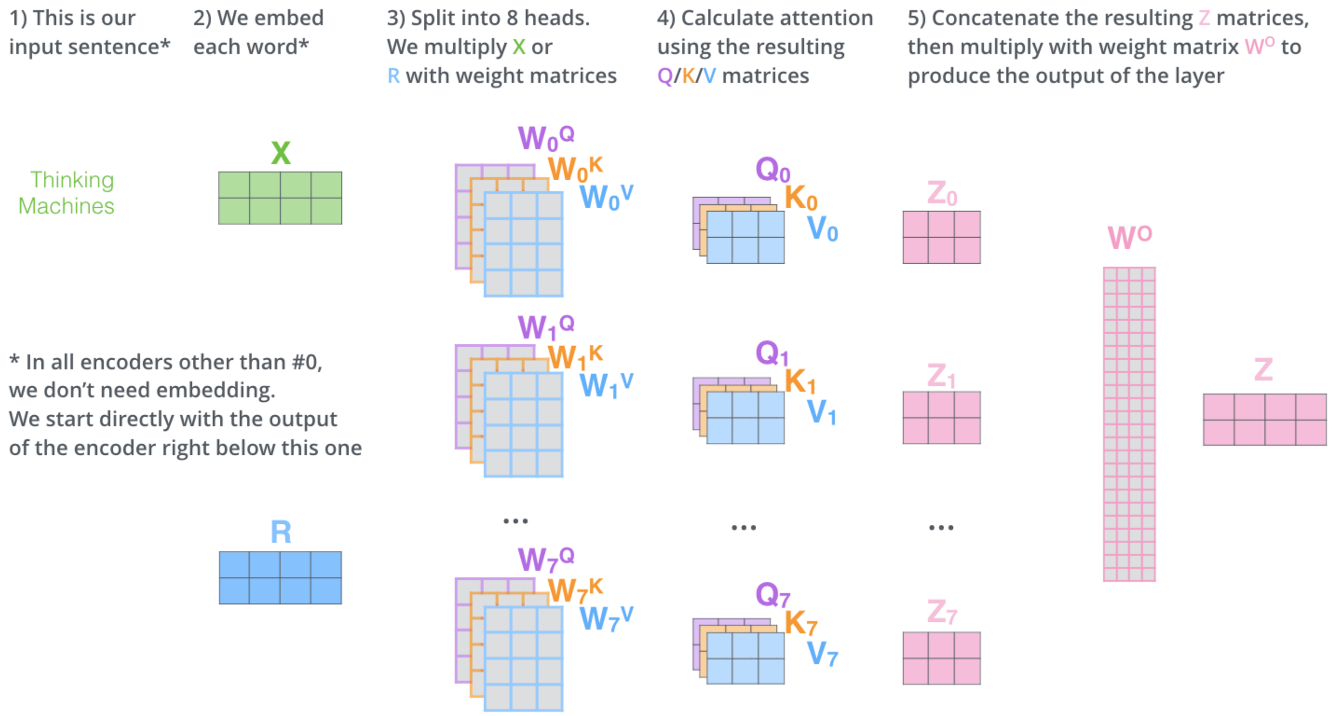
Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this. [1]

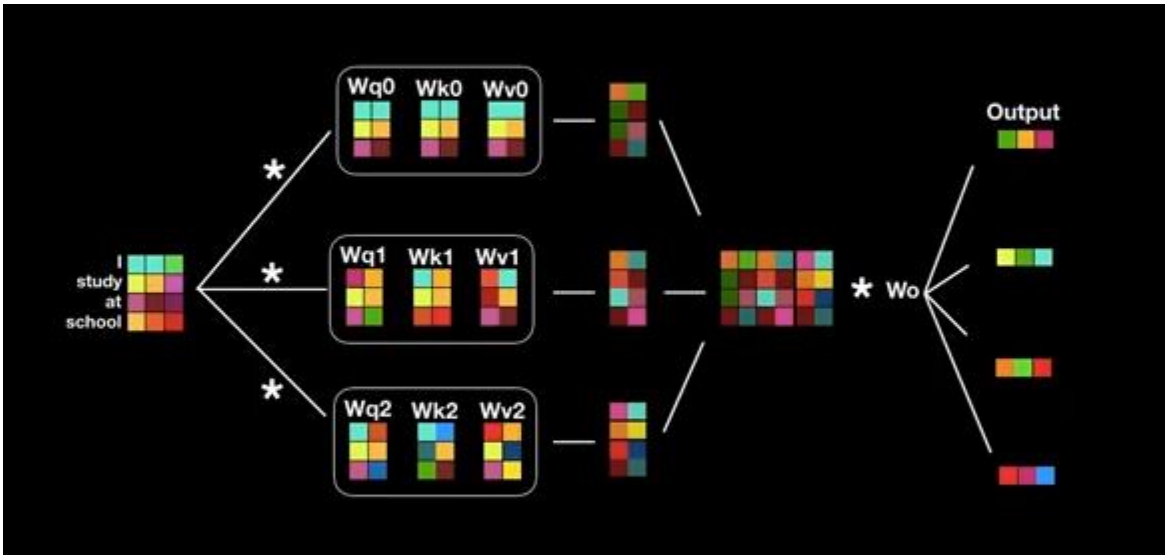
$$\mathrm{MultiHead}(Q,K,V) = \mathrm{Concat}(\mathrm{head}_1, ..., \mathrm{head}_\mathrm{h})W^O$$
$$\text{where } \mathrm{head}_\mathrm{i} = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_\text{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_\text{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_\text{model} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_\text{model}}$.

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 to concatenate all these attention heads into a single matrix. [1, 3]

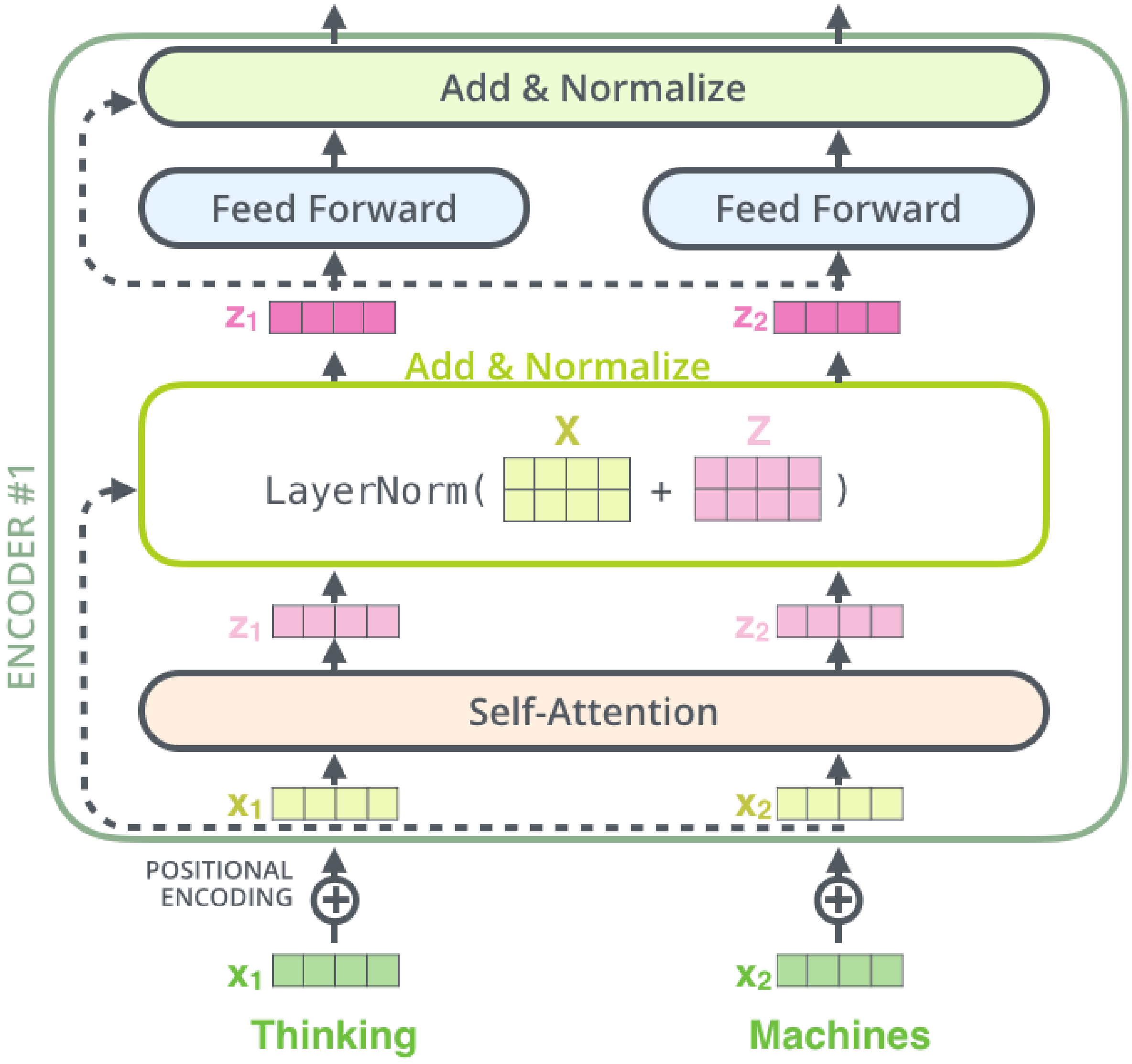

1) This is our input sentence*
2) We embed each word*
3) Split into 8 heads. We multiply X or R with weight matrices
4) Calculate attention using the resulting Q/K/V matrices
5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer
Thinking Machines
X
* In all encoders other than #0, we don’t need embedding. We start directly with the output of the encoder right below this one
R
W0Q
W0K
W0V
W1Q
W1K
W1V
...
W7Q
W7K
W7V
Q0
K0
V0
Q1
K1
V1
...
Q7
K7
V7
Z0
Z1
...
Z7
WO
Z



2.1.5 Resudial and Normalization

One detail in the architecture of the encoder that we need to mention before moving on, is that 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 across each feature instead of each sample, it’s better for stabilization [1, 3]

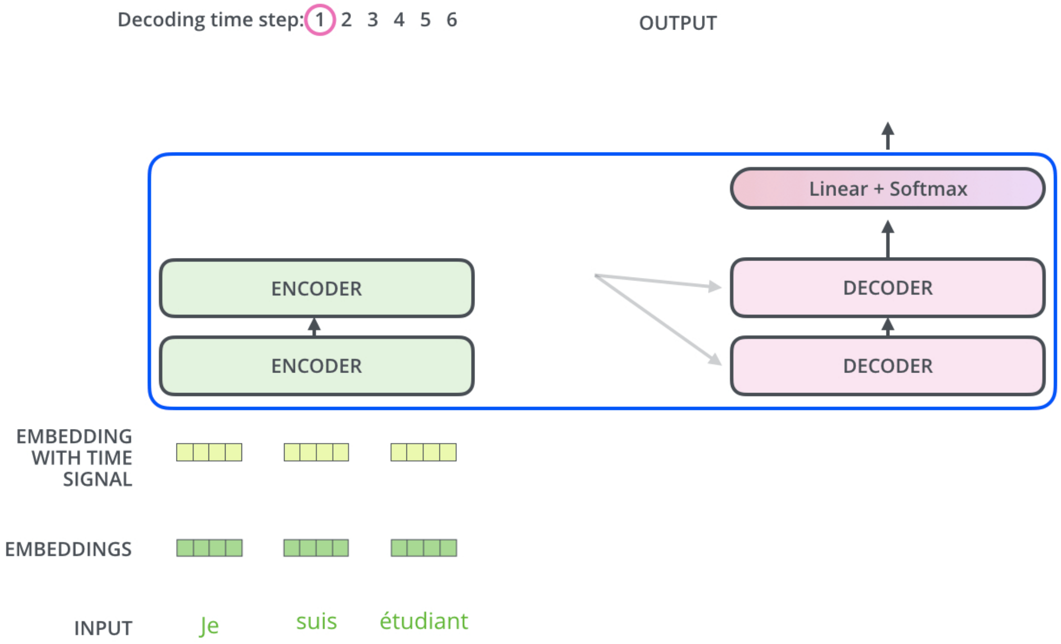


2.1.6 Feed Forward Network

Applied every one of the attention vector, these FFN are used in practice to transform the attention into a form that is digestible by the next encoder block or decoder block [1, 3]

2.2 Decoder

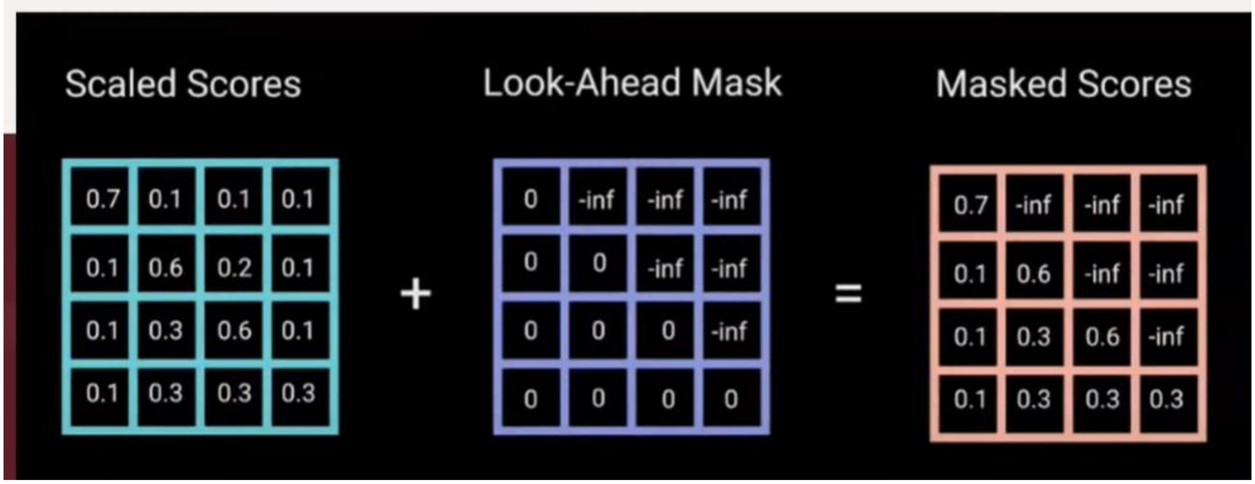
The encoder start by processing the input sequence. The output of the top encoder is then transformed into a set of attention vectors K and V. These are to be used by each decoder in its “encoder-decoder attention” layer which helps the decoder focus on appropriate places in the input sequence:



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

In the decoder, the self-attention layer is only allowed to attend to earlier positions in the output sequence. **Masked Multi-Head Attention** masking future positions (setting them to -inf) before the softmax step in the self-attention calculation. It have the input is the label, but it shifted to the right by insert a token (e.g <start>) and the sign to stop Decoder is another token (e.g <end>)

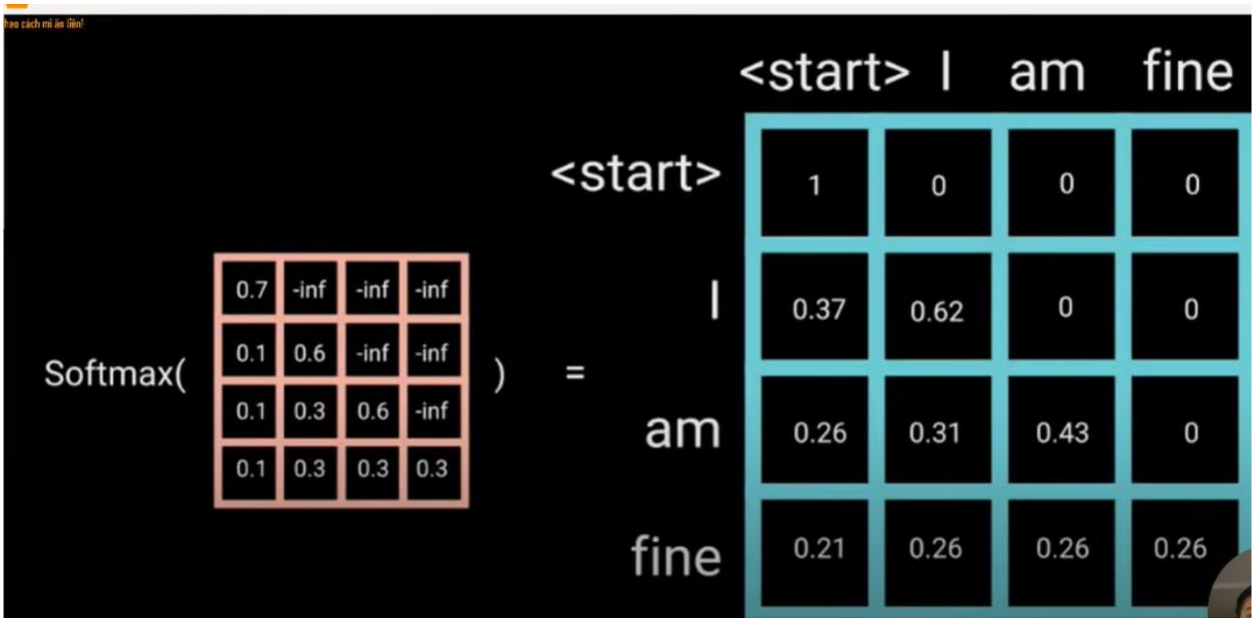
For instances, we have a matrix (4 vector of 4 token) after Position Encoding and we add a Mask matrix



Scaled Scores Look-Ahead Mask Masked Scores

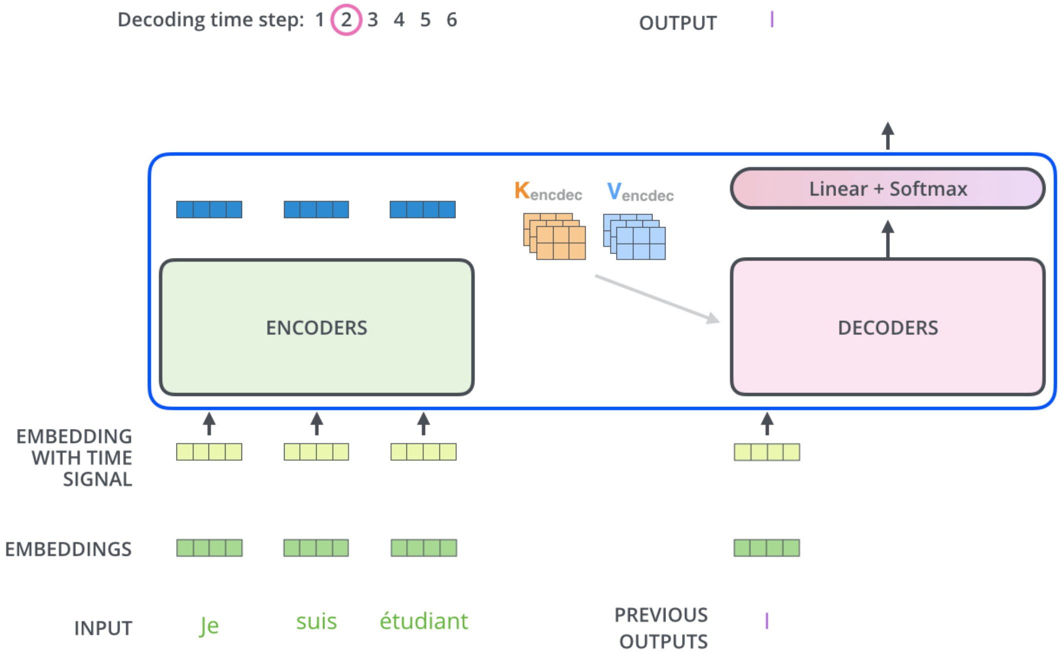
$$\begin{bmatrix} 0.7 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.6 & 0.2 & 0.1 \\ 0.1 & 0.3 & 0.6 & 0.1 \\ 0.1 & 0.3 & 0.3 & 0.3 \end{bmatrix} + \begin{bmatrix} 0 & -inf & -inf & -inf \\ 0 & 0 & -inf & -inf \\ 0 & 0 & 0 & -inf \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0.7 & -inf & -inf & -inf \\ 0.1 & 0.6 & -inf & -inf \\ 0.1 & 0.3 & 0.6 & -inf \\ 0.1 & 0.3 & 0.3 & 0.3 \end{bmatrix}$$

Assumed that is a result we have, the output become an input vector Q of the Multi-Head Attention. Then we calculate as the same Encoder and the attention Q dot K^T has -inf value. Therefor Softmax scale -inf into 0 and the others value becoming Score of token.



$$\text{Softmax}\left(\begin{bmatrix} 0.7 & -inf & -inf & -inf \\ 0.1 & 0.6 & -inf & -inf \\ 0.1 & 0.3 & 0.6 & -inf \\ 0.1 & 0.3 & 0.3 & 0.3 \end{bmatrix} \right) = \begin{array}{c|cccc} & \text{<start>} & \text{I} & \text{am} & \text{fine} \\ \hline \text{<start>} & 1 & 0 & 0 & 0 \\ \text{I} & 0.37 & 0.62 & 0 & 0 \\ \text{am} & 0.26 & 0.31 & 0.43 & 0 \\ \text{fine} & 0.21 & 0.26 & 0.26 & 0.26 \end{array}$$

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. And just like we did with the encoder inputs, we embed and add positional encoding to those decoder inputs to indicate the position of each word.



3. Application

Any application using sequential text, image or video data is a candidate for transformer models: [2]

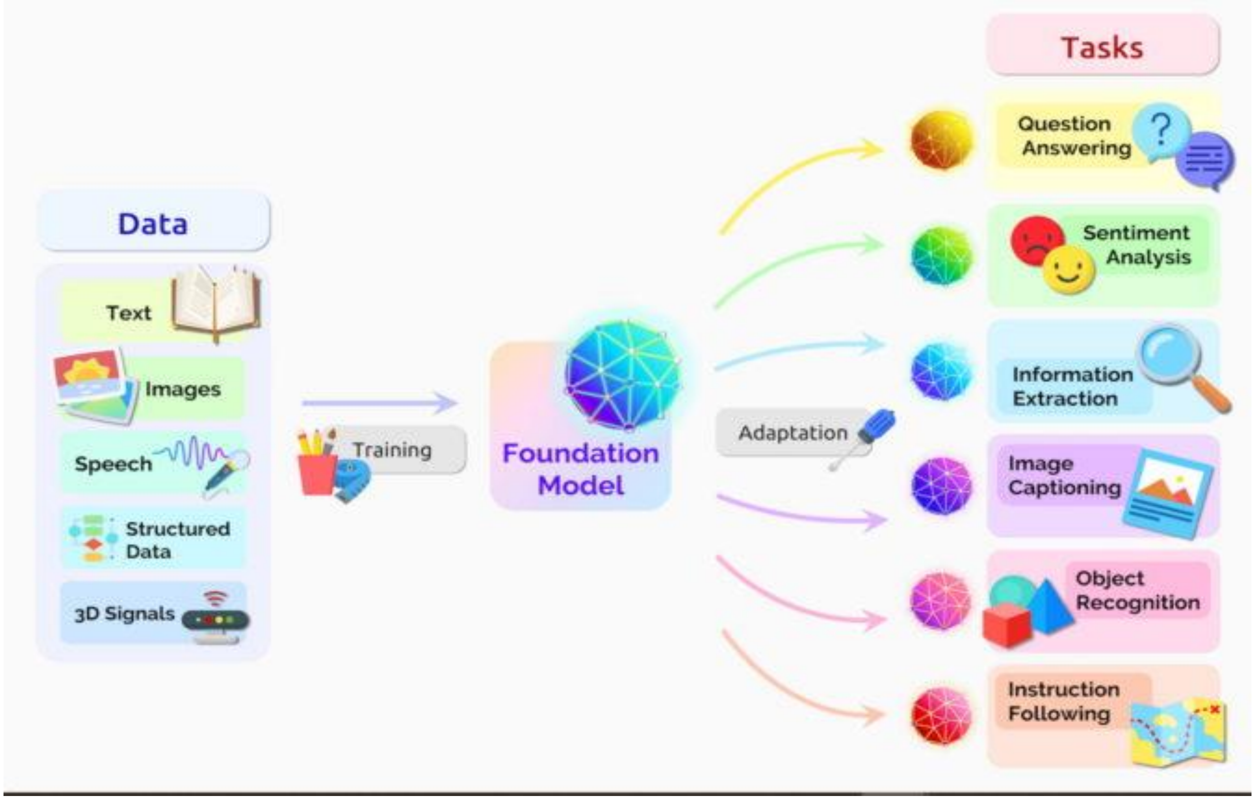
Transformers are translating text and speech in near real-time, opening meetings and classrooms to diverse and hearing-impaired attendees.

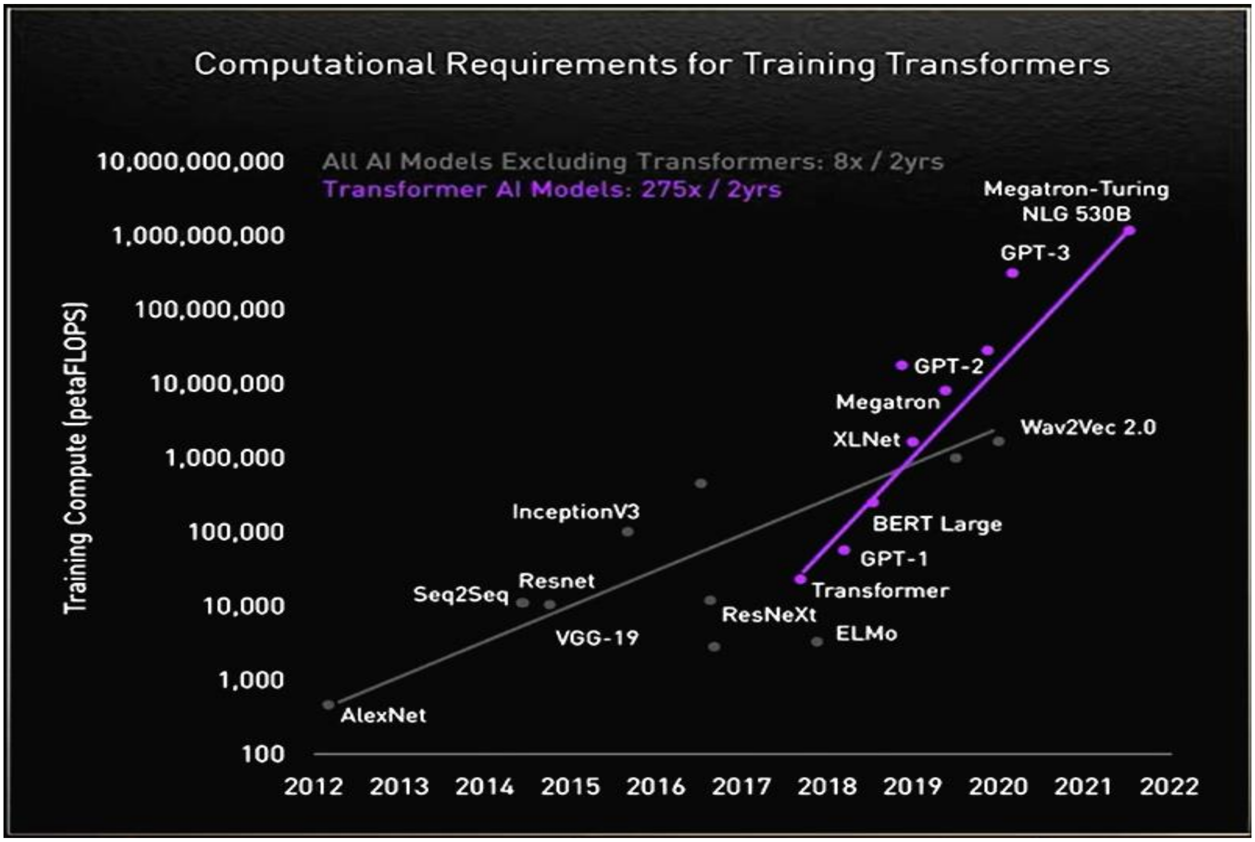
They’re helping researchers understand the chains of genes in DNA and amino acids in proteins in ways that can speed drug design.

Transformers can detect trends and anomalies to prevent fraud, streamline manufacturing, make online recommendations or improve healthcare.

People use transformers every time they search on Google or Microsoft Bing.

For more related model [7]


Data
Text
Images
Speech
Structured Data
3D Signals
Training
Foundation Model
Adaptation
Tasks
Question Answering
Sentiment Analysis
Information Extraction
Image Captioning
Object Recognition
Instruction Following


Computational Requirements for Training Transformers
All AI Models Excluding Transformers: 8x / 2yrs
Transformer AI Models: 275x / 2yrs
Training Compute (petaFLOPS)
10,000,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
1,000
100
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
Megatron-Turing NLG 530B
GPT-3
GPT-2
Megatron
XLNet
Wav2Vec 2.0
InceptionV3
BERT Large
GPT-1
Transformer
Seq2Seq
Resnet
ResNeXt
ELMo
VGG-19
AlexNet

4. Comparison to Recurrent and Convolutional Layers [8]

1. Self-attention layers were found to be faster than recurrent layers for shorter sequence lengths and can be restricted to consider only a neighborhood in the input sequence for very long sequence lengths.
2. The number of sequential operations required by a recurrent layer is based on the sequence length, whereas this number remains constant for a self-attention layer.
3. In convolutional neural networks, the kernel width directly affects the long-term dependencies that can be established between pairs of input and output positions. Tracking long-term dependencies would require using large kernels or stacks of convolutional layers that could increase the computational cost.

No Labels, More Performance [2]

Before transformers arrived, users had to train neural networks with large, labeled datasets that were costly and time-consuming to produce. By finding patterns between elements mathematically, transformers eliminate that need, making available the trillions of images and petabytes of text data on the web and in corporate databases.

In addition, the math that transformers use lends itself to parallel processing, so these models can run fast.

Transformers now dominate popular performance leaderboards like SuperGLUE, a benchmark developed in 2019 for language-processing systems.

5. Disadvantages

The main challenge in training transformers to learn from data is that they are costly in terms of time and memory, especially for long input sequences. To address these shortcomings, researchers have proposed other variants of transformers. Some of these variants are compressive transformers that use compressed representations to map information, reformers to improve transformer efficiency, and extended transformer construction based on sparse attention mechanisms. [5, 6]

Reference

[1] [1706.03762] Attention Is All You Need (arxiv.org)

[2] What Is a Transformer Model? | NVIDIA Blogs

[3] The Illustrated Transformer – Jay Alammar – Visualizing machine learning one concept at a time. (jalammar.github.io)

[4] A Gentle Introduction to Positional Encoding in Transformer Models, Part 1 - MachineLearningMastery.com

[5] [D] Weaknesses of the transformers : MachineLearning (reddit.com)

[6] Transformers: What They Are and Why They Matter - Blog | AI Exchange (scale.com)

[7] Transformer models: an introduction and catalog—2023 Edition - AI, software, tech, and people, not in that order… by X (amatriain.net)

[8] The Transformer Model - MachineLearningMastery.com

Bonus

[visualize attention] Visualizing A Neural Machine Translation Model (Mechanics of Seq2seq Models With Attention) – Jay Alammar – Visualizing machine learning one concept at a time. (jalammar.github.io)

[] Transformer Applications - Text Summarization | Coursera

[] An Overview of Transformers | Papers With Code