Transformer Neural Network: Step-By-Step Breakdown of the Beast

Utkarsh Ankit · Follow
Published in Towards Data Science
13 min read · Apr 24, 2020

source: arseny togulev on unsplash.

The Transformer Neural Network 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"* 2017 [1]. It is the current state-of-the-art technique in the field of NLP.

Before directly jumping to Transformer, I will take some time to explain the reason why we use it and from where it comes into the picture. (If you want to skip this part

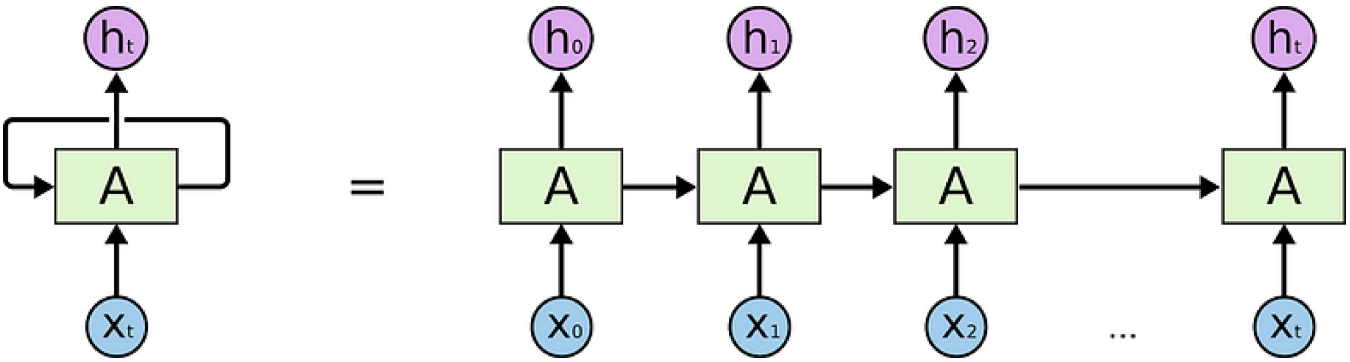
then directly go to the Transformer topic, but I suggest you read it sequentially for better understanding).

So, the story starts with RNN (Recurrent Neural Networks).

RNN

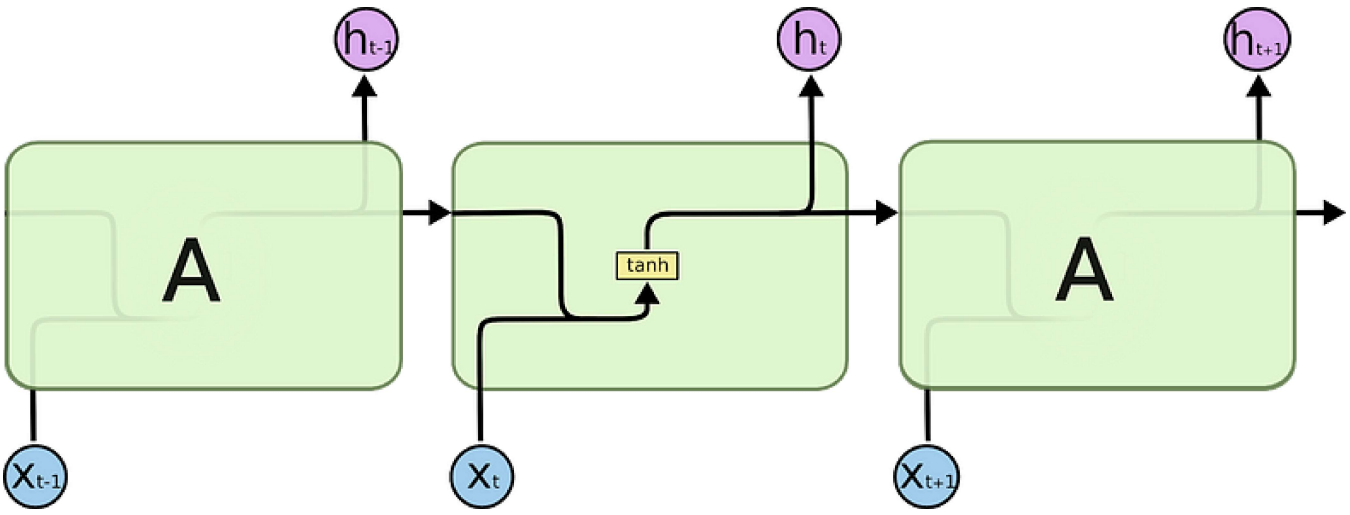
What is RNN? How is it different from simple ANN? What is the major difference?

RNNs are the Feed Forward Neural Networks that are rolled out over time.



source: colah's (CC0).

Unlike normal Neural Networks, RNNs are designed to take a **series of inputs** with *no predetermined limit on size*. "Series" as in any input of that sequence has some relationship with their neighbour's or have some influence on them.



Architecture of RNN. source: colah's (CC0).

Basic feedforward networks "remember" things too, but they remember the things they learnt during training. While RNNs learn similarly while training, in addition, they remember things learnt from prior input(s) while generating output(s).

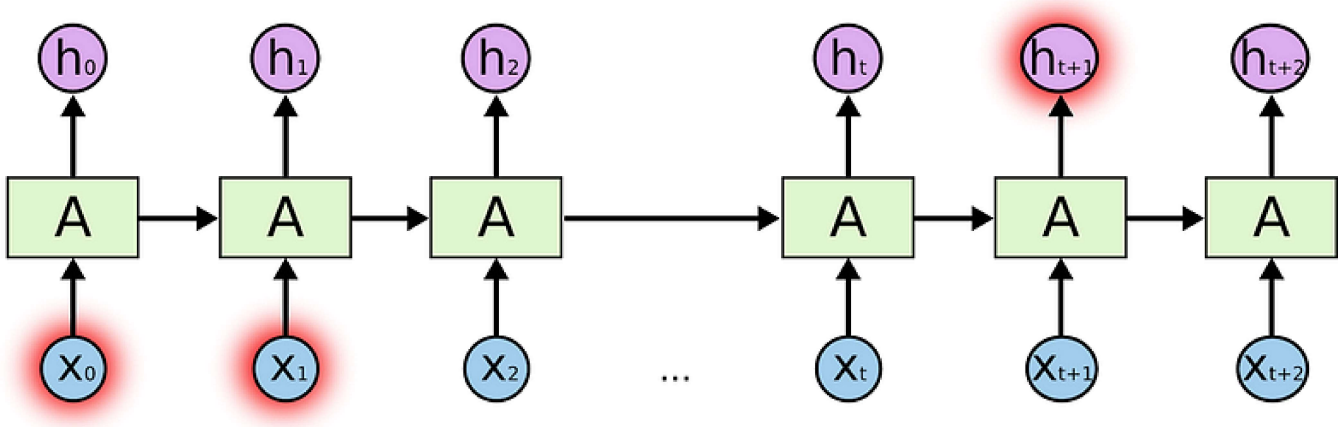


Image illustrating long-term dependencies. source: colah's (CC0).

It is used in different types of models-

1.) **Vector-Sequence Models**- They take fixed-sized vectors as input and output vectors of any length, for example, in image captioning, the image is given as an input and the output describes the image.

2.) **Sequence-Vector Model**- Take a vector of any size and output a vector of fixed size. Eg. Sentiment analysis of a movie rates the review of any movie as positive or negative as a fixed size vector.

3.) **Sequence-to-Sequence Model**- The most popular and most used variant, take input as a sequence and give output as another sequence with variant sizes. Eg. Language translation, for time series data for stock market prediction.

Its disadvantages-

1. Slow to train.
2. Long sequence leads to vanishing gradient or, say, the problem of long term dependencies. In simple terms, its memory is not that strong when it comes to remembering the old connection.

For **Eg.** "The clouds are in the ____."

It is obvious that the next word will be sky, as it is linked with the clouds. Here we see the distance between clouds and the predicted word is less so RNN can predict it easily.

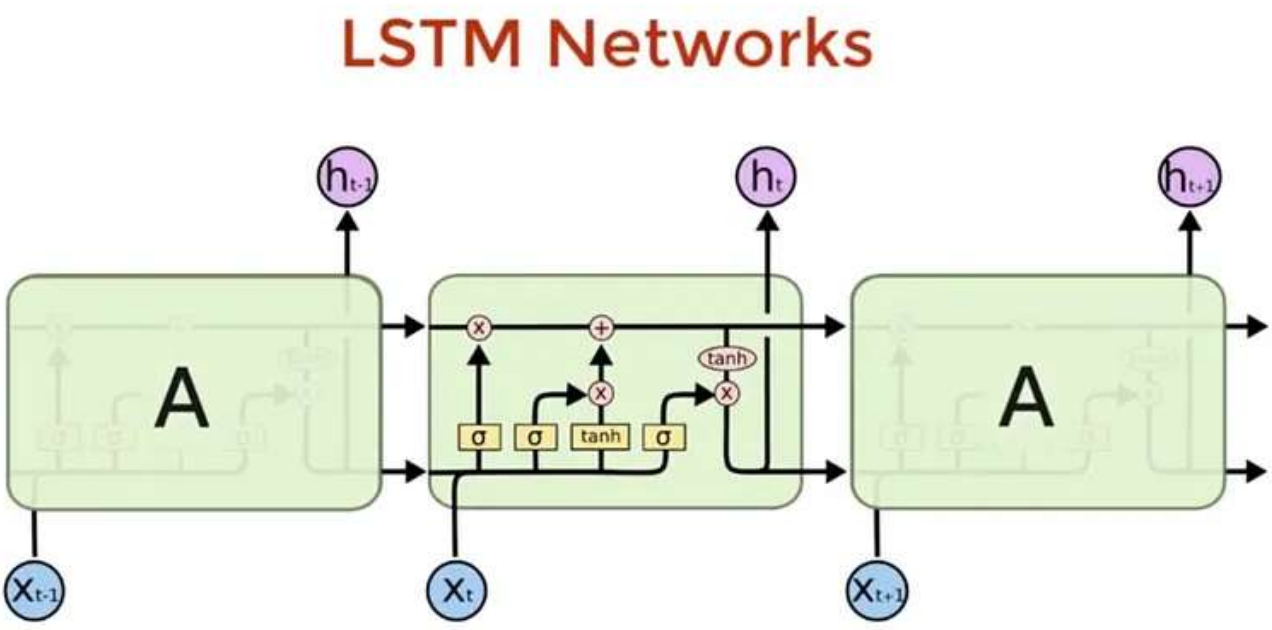
But, for another example,

"I grew up in Germany with my parents, I spent many years and have proper knowledge about their culture, that's why I speak fluent ____."

Here the predicted word is German, which is directly connected with Germany, but the distance between Germany and the predicted word is more in this case, so it is difficult for RNN to predict.

So, Unfortunately, as that gap grows, RNNs become unable to connect, as their memory fades with distance.

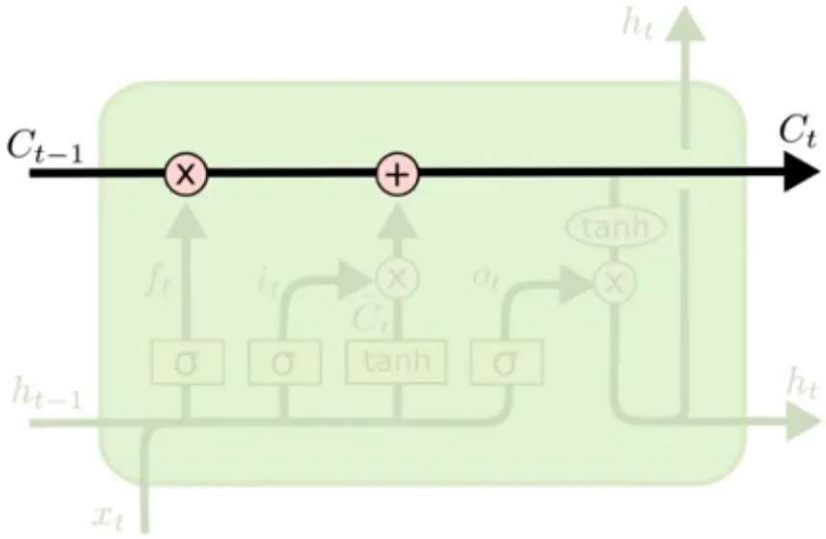
LSTM



The repeating module in an LSTM contains four interacting layers.

source: colah's (CC0).

Long Short Term Memory- Special kind of RNN, specially made for solving vanishing gradient problems. They are capable of learning Long-Term Dependencies. Remembering information for long periods of time is practically their default behaviour, not something they struggle to learn!



This branch allows to pass info and skip long processing of the cell. source: colah's (CC0).

The LSTM Neurons have unlike normal neurons have a branch that allows to pass information and to skip the long processing of the current cell, this allows the memory to be retained for a longer period of time. It does improve the situation of the vanishing gradient problem but not that amazingly, like it will do good till 100 words, but for like 1,000 words, it starts to lose its grip.

But like simple RNN it is also very slow to train, or even slower.

They take input sequentially one by one, which is not able to use up GPU's very well, which are designed for parallel computation.

How can we parallelize sequential data?? *(I will get back on this question.)*

For now, we are dealing with two issues-

- Vanishing gradient
- Slow training

Solving the vanishing gradient issue:

Attention

It answers the question of what part of the input we should focus on.

I am going to explain attention in a slightly different way. Let's take a situation-

Suppose someone gave us a book of machine learning and asked us to give information about categorical cross-entropy. There are two ways of doing it, first, read the whole book and come back with the answer. Second, go to the index, find the 'losses' chapter, go to the cross-entropy part and read the part of Categorical Cross Entropy.

What do you think is the faster method?

Like in the first method, it may take a whole week to read the entire book. While in second, it will hardly take 5 mins. Furthermore, our information from the first method will be more vague and diverse as it is based on too much information, while the information from the second one will be accurate to the requirement.

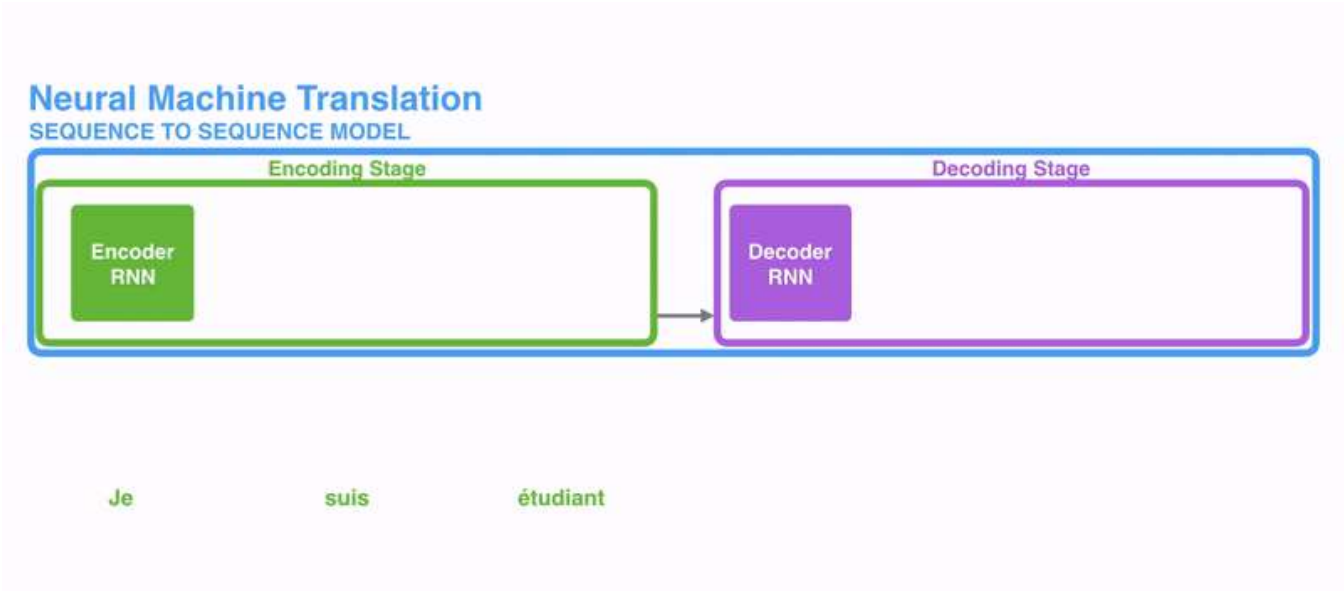
What did we do differently here?

In the former case we didn't focus on any part of the book specifically, whereas in the latter case, we focused our attention on the chapter of losses and then further focused our attention on the cross-entropy part where the concept of Categorical Cross Entropy is explained. Actually, this is the way most of us humans will do.

Attention in neural networks is somewhat similar to what we find in humans. They focus on the high resolution in certain parts of the inputs while the rest of the input is in low resolution [2].

Let's say we are making an NMT(Neural Machine Translator),

Check out this animation, this shows how a simple seq-to-seq model works.

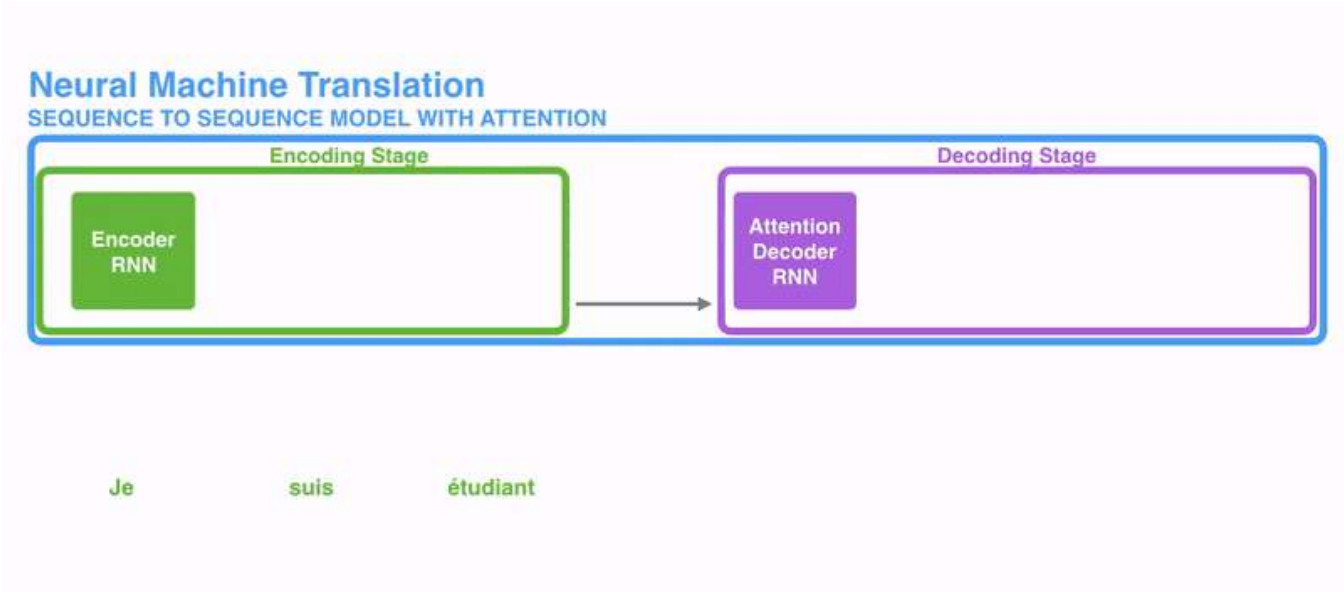


Working of a classic Seq-to-Seq model. source: jalammar's (CC BY-NC-SA 4.0).

We see that for each step of the encoder or decoder, RNN is processing its inputs and generating output for that time step. In each time step, RNN updates its hidden state based on inputs and previous outputs it has seen. In the animation, we see that the hidden state is actually the **context vector** we pass along to the decoder.

Time for "Attention".

The **context vector** turned out to be problematic for these types of models. Models have a problem while dealing with long sentences. Or say they were facing the vanishing gradient problem in long sentences. So, a solution came along in a paper [2], Attention was introduced. It highly improved the quality of machine translation as it allows the model to focus on the relevant part of the input sequence as needed.



Working of a Seq-to-Seq Model with Attention. source: jalammar's (CC BY-NC-SA 4.0).

This attention model is different from the classic seq-to-seq model in two ways-

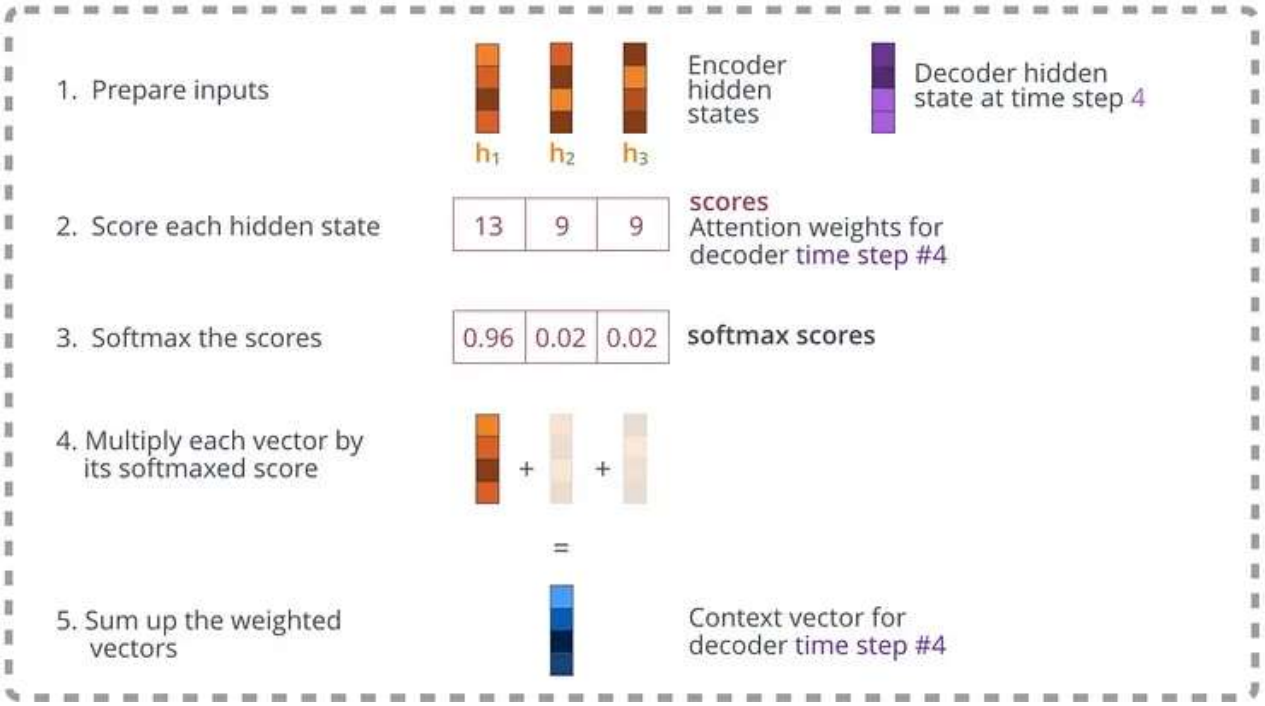
- As compared to a simple seq-to-seq model, here the encoder passes a lot more data to the decoder. Previously only the last, final hidden state of the encoding part is sent to the decoder, but now the encoder passes all the hidden states (even the intermediate ones) to the decoder.
- The decoder part does an extra step before producing its output. Explained below-

The last step of decoders proceed as follow-

1. It checks each hidden state that it received as every hidden state of the encoder is mostly associated with a particular word of the input sentence.

2. I give each hidden state a score.

3. Then each score is multiplied by their respective softmax score, thus amplifying hidden states with high scores, and drowning out hidden states with low scores. *(Refer to the image below for clear visualization.)*



source: jalammar's (CC BY-NC-SA 4.0).

This scoring exercise is done at each time step on the decoder side.

Now when we bring the whole thing together:

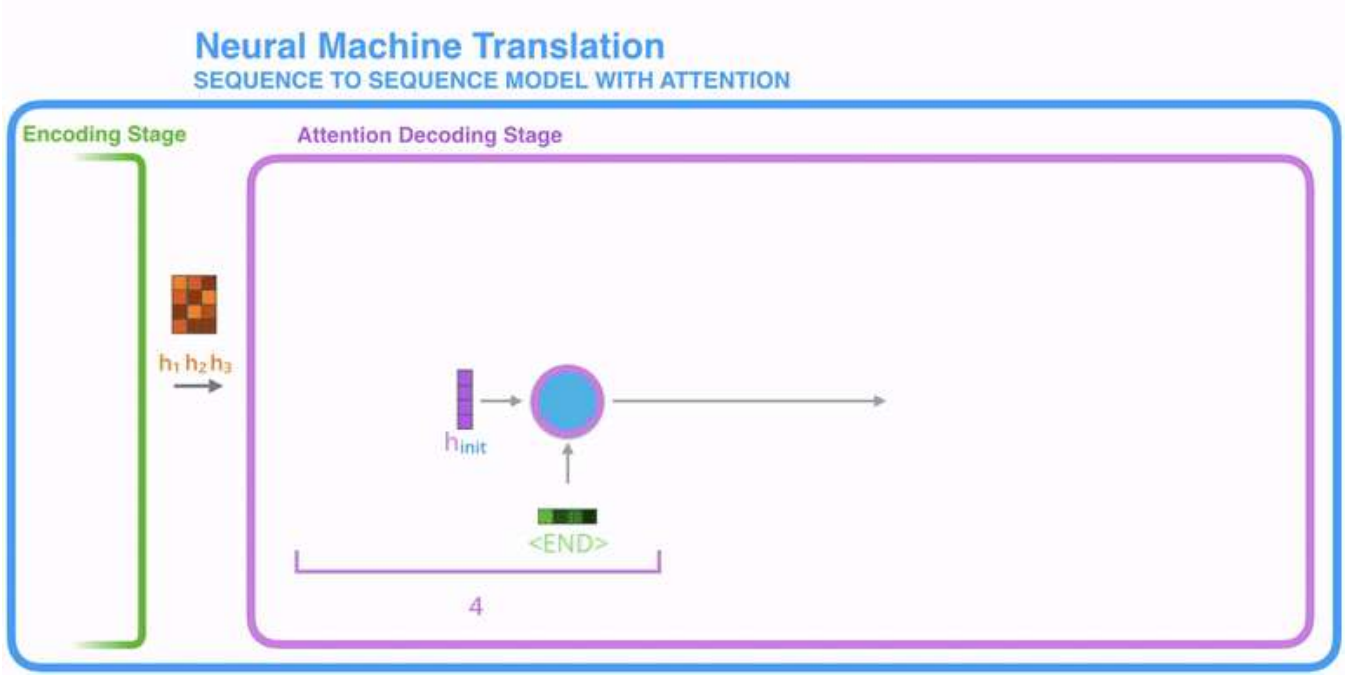
1. The attention decoder layer takes the embedding of the <END> token, and an initial decoder hidden state, the RNN processes its inputted and produces an output and a new hidden state vector(h4).

2. Now we use encoder hidden states and the h4 vector to calculate a context vector C4 for this time step. This is where the attention concept is applied, that's why it's called the Attention Step.

3. We concatenate (h4) and C4 in one vector.

4. Now, this vector is passed into a feed-forward neural network, the output of the feed-forward neural networks indicates the output word of this time step.

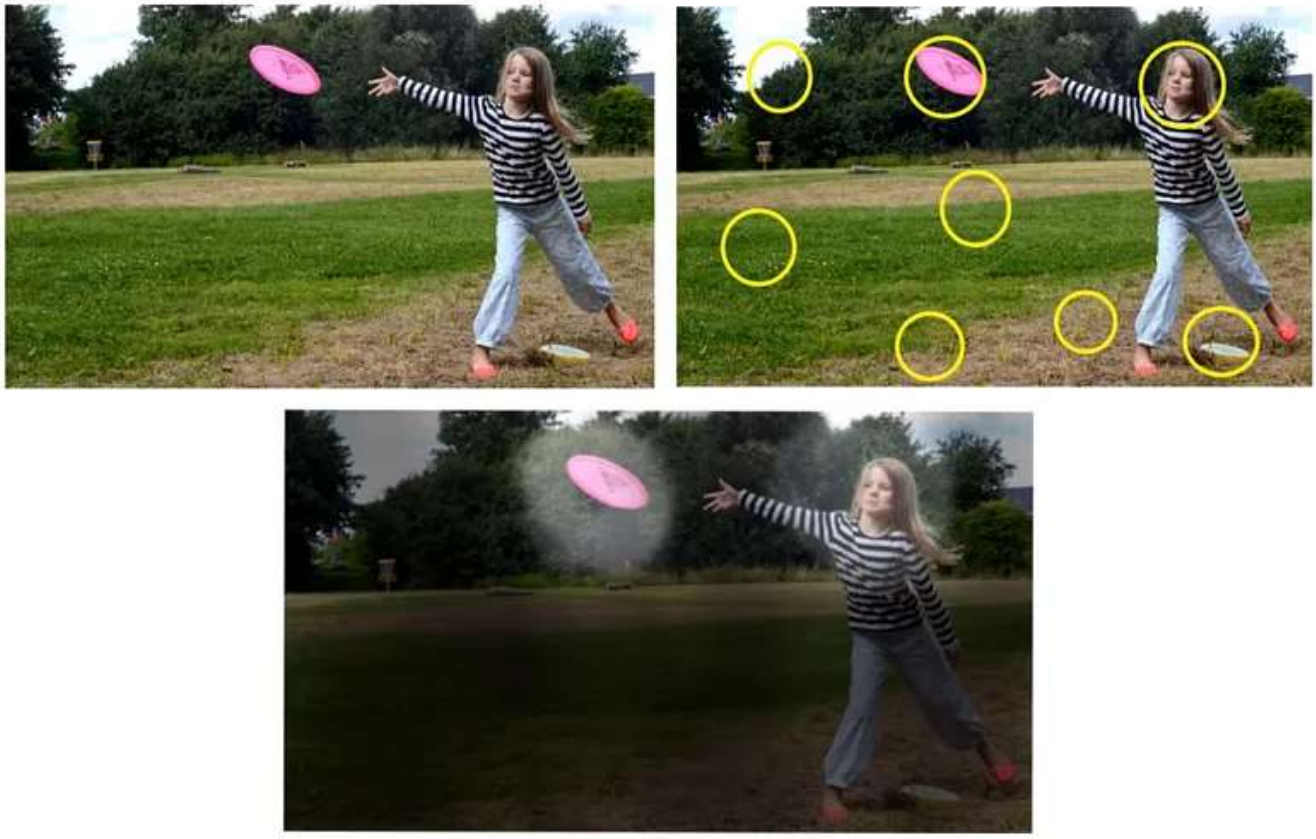
5. These steps get repeated for the next time steps. *(Refer to the slide below for clear visualization.)*



The Final Step. source: jalammar's (CC BY-NC-SA 4.0).

So, this is how **Attention** works.

Eg. Working of Attention In an Image Captioning Problem:-



Working of attention in an Image Captioning Problem. source: CodeEmporium (CC0).

Now, remember the question which I stated earlier-

How can we parallelize sequential data??

So, here comes our ammunition-

Transformers

A paper called *"Attention Is All You Need"* published in 2017 comes into the picture, it introduces an encoder-decoder architecture based on attention layers, termed as the transformer.

One main difference is that the input sequence can be passed parallelly so that GPU can be utilized effectively, and the speed of training can also be increased. And it is based on the multi-headed attention layer, vanishing gradient issue is also overcome by a large margin. The paper is based on the application of transformer on NMT(Neural Machine Translator).

So, here both of our problems which we highlighted before are solved to some level here.

Like for example in a translator made up of simple RNN we input our sequence or the sentence in a continuous manner, one word at a time to generate word embeddings. As every word depends on the previous word, its hidden state acts accordingly, so it is necessary to give one step at a time. While in transformer, it is not like that, we can pass all the words of a sentence simultaneously and determine the word embedding simultaneously. So, how it is actually working, let's see ahead-

The architecture:-

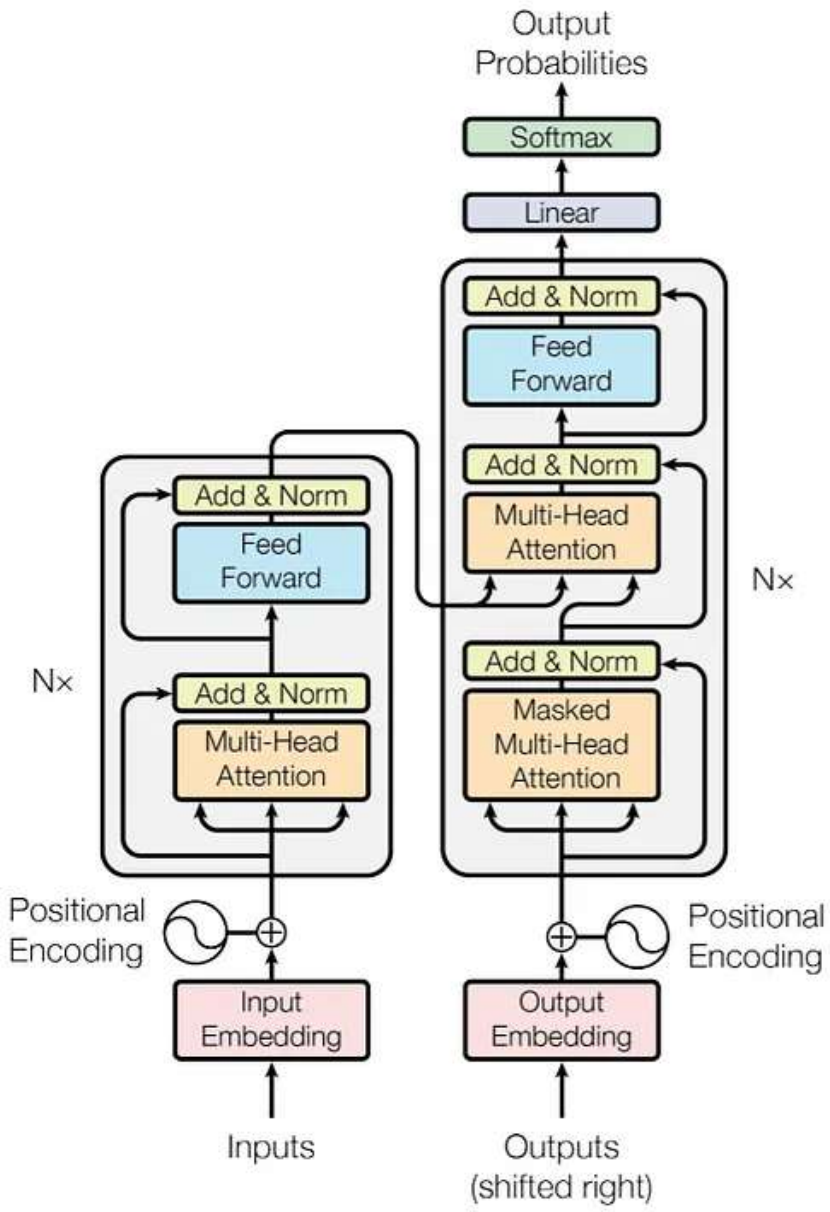
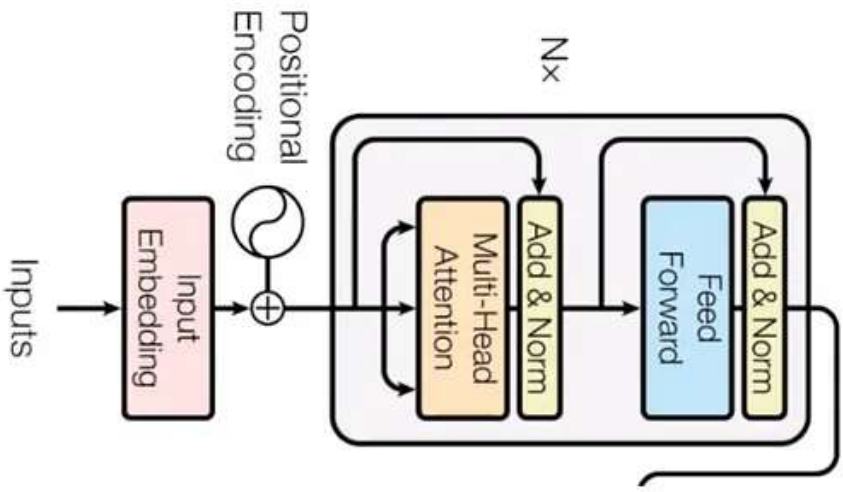


Figure 1: The Transformer - model architecture.

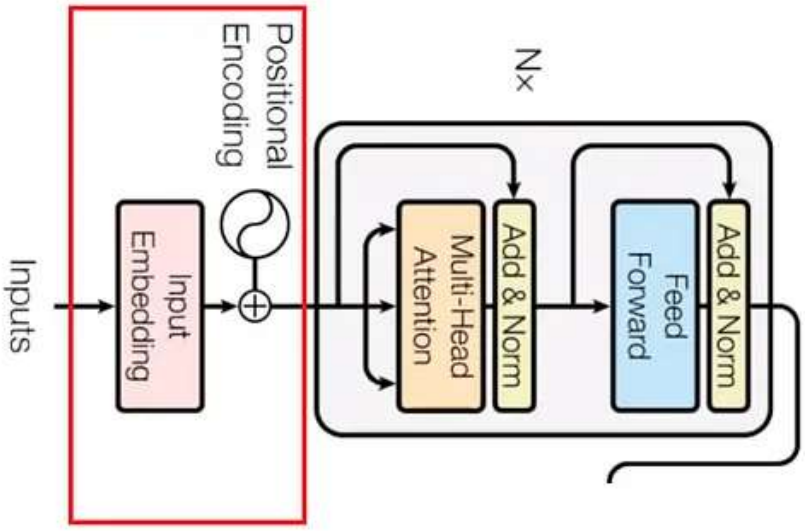
source: arXiv:1706.03762 **[cs.CL].**

> *1. Encoder Block -*



source: arXiv:1706.03762 **[cs.CL].**

It's a fact that computers don't understand words, it works on numbers, vectors or matrices. So, we do need to convert our words to a vector. But how this can be possible. So, here the concept of **Embedding Space** comes. It's like an open space or dictionary where words of similar meanings are grouped together or are present close to each other in that space. This space is termed as embedding space, and here every word, according to its meaning, is mapped and assigned with a particular value. So, here we convert our words to vectors.



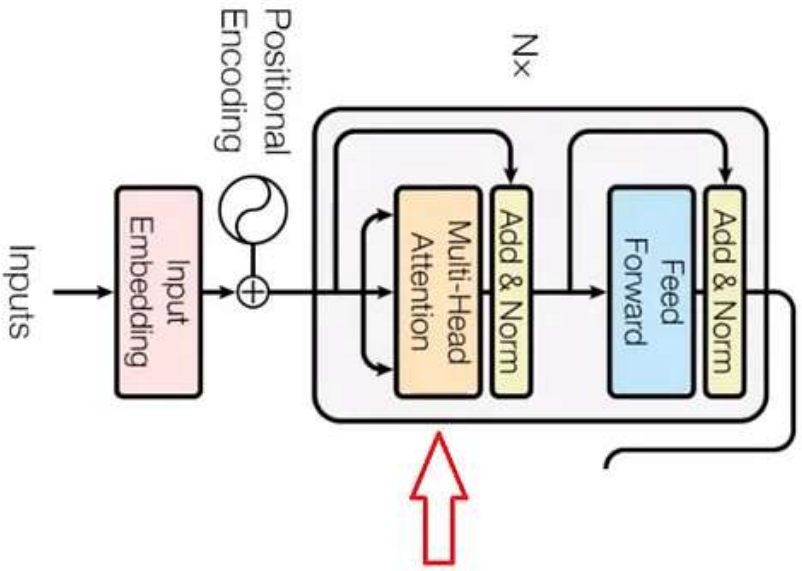
source: arXiv:1706.03762 **[cs.CL].**

But one other issue we will face is that every word in different sentences has different meanings. So, to solve this issue, we take the help of **Positional Encoders**. It is a vector that gives context according to the position of the word in a sentence.

Word → Embedding → Positional Embedding → Final Vector, termed as Context.

So, our input is ready, now it goes to the encoder block.

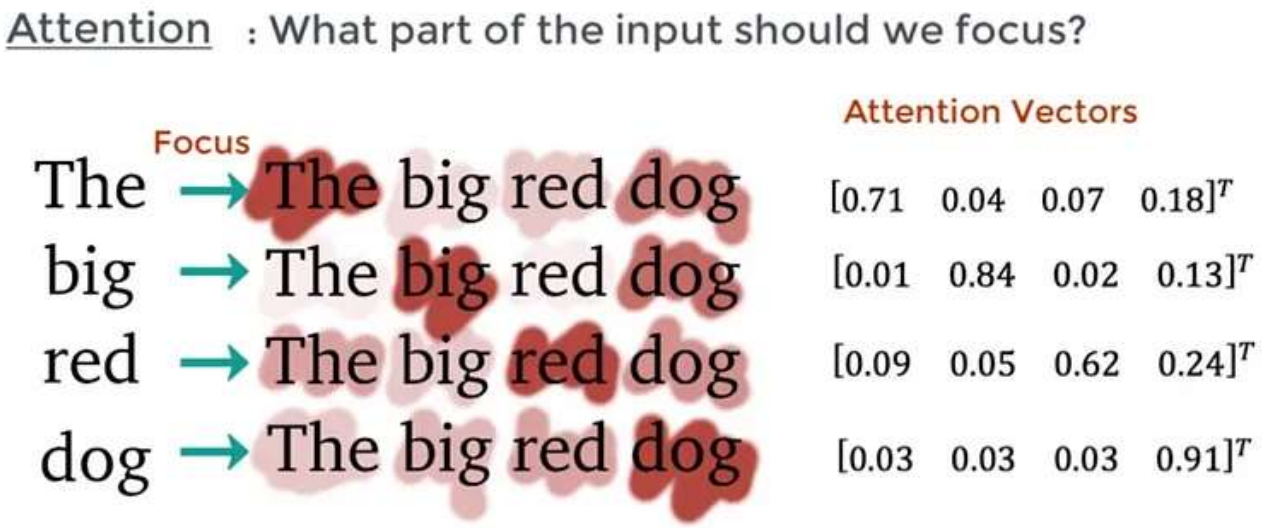
> *Multi-Head Attention Part -*



source: arXiv:1706.03762 **[cs.CL].**

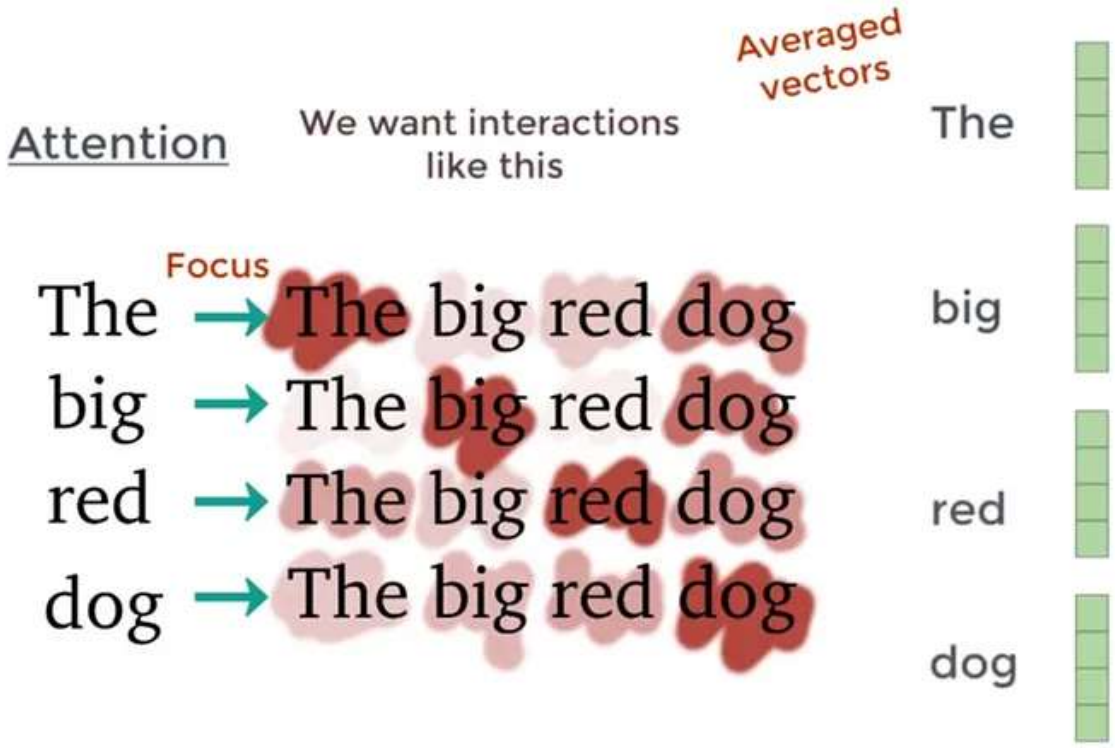
Now the main essence of the transformer comes in, “Self Attention”.

It focuses on the part that how relevant a particular word is w.r.t to other words in that sentence. It is represented as an attention vector. For every word, we can have an attention vector generated, which captures the contextual relationship between words in that sentence.



source: CodeEmporium (CC0).

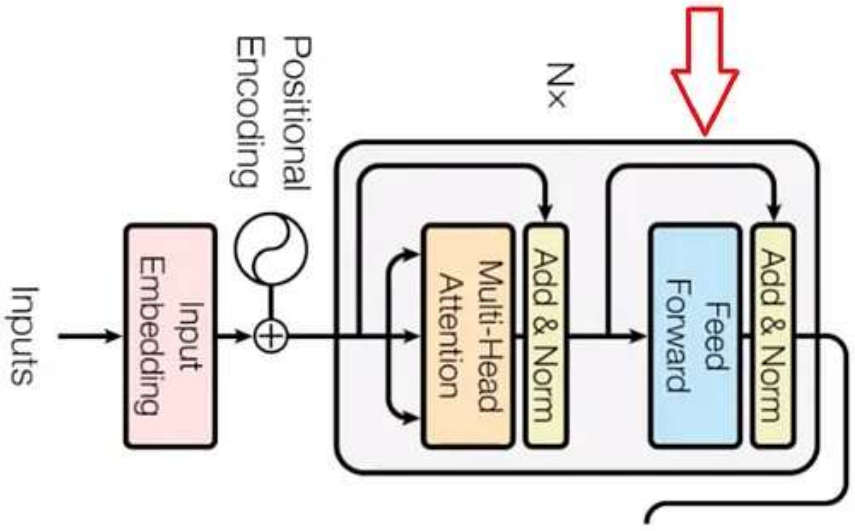
The only problem it faces is that for every word it weighs its value much higher on itself in the sentence, be we are inclined towards its interaction with other words of that sentence. So, we determine multiple attention vectors per word and take a weighted average, to compute the final attention vector of every word.



source: CodeEmporium (CC0).

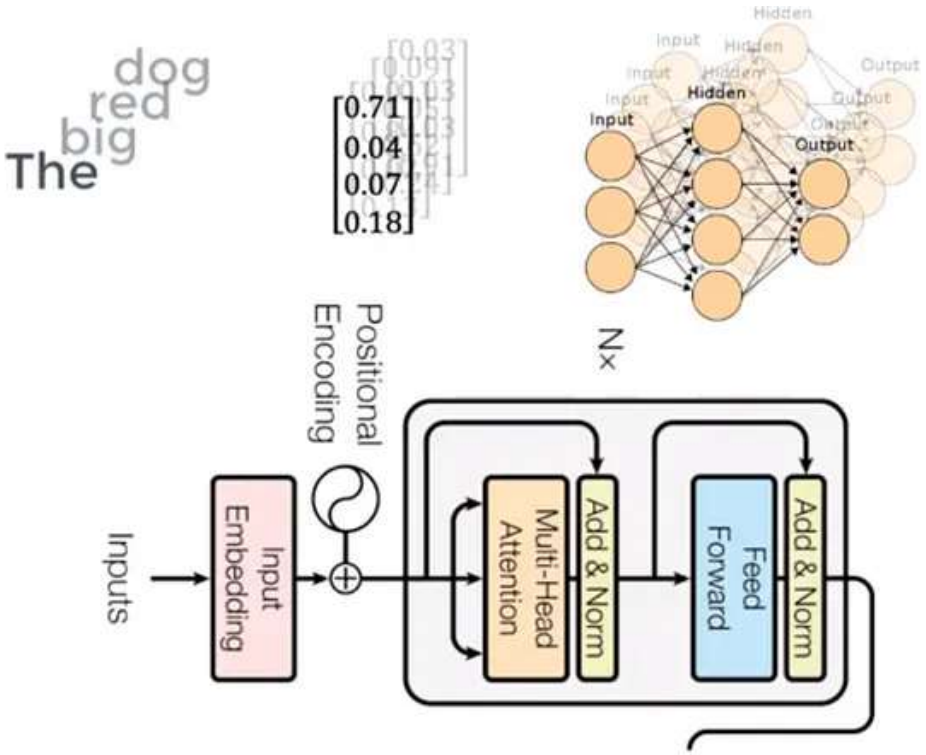
As we are using multiple attention vectors, it is called the **Multi-Head Attention Block.**

> *Feed Forward Network -*



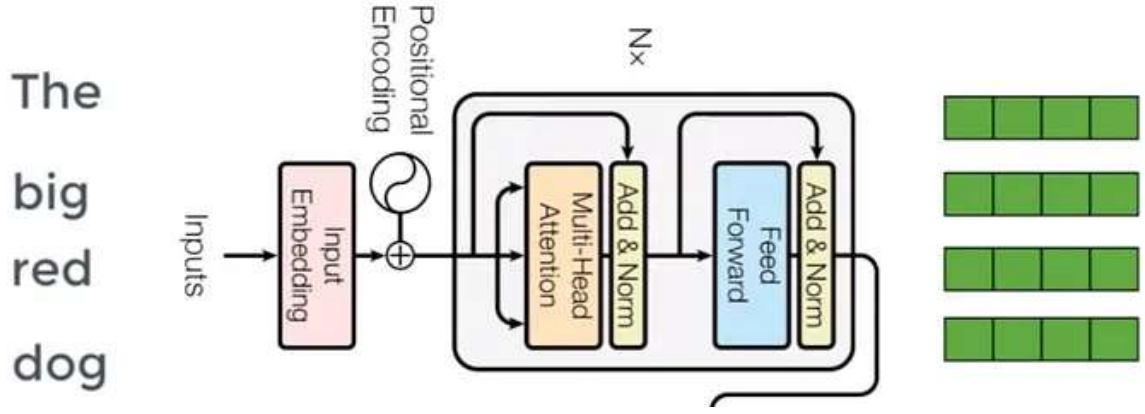
source: arXiv:1706.03762 **[cs.CL].**

Now, the second step is the Feed Forward Neural Network. This is the simple feed-forward Neural Network that is applied to every attention vector, it's the main purpose is to transform the attention vectors into a form that is acceptable by the next encoder or decoder layer.



source: arXiv:1706.03762 **[cs.CL].**

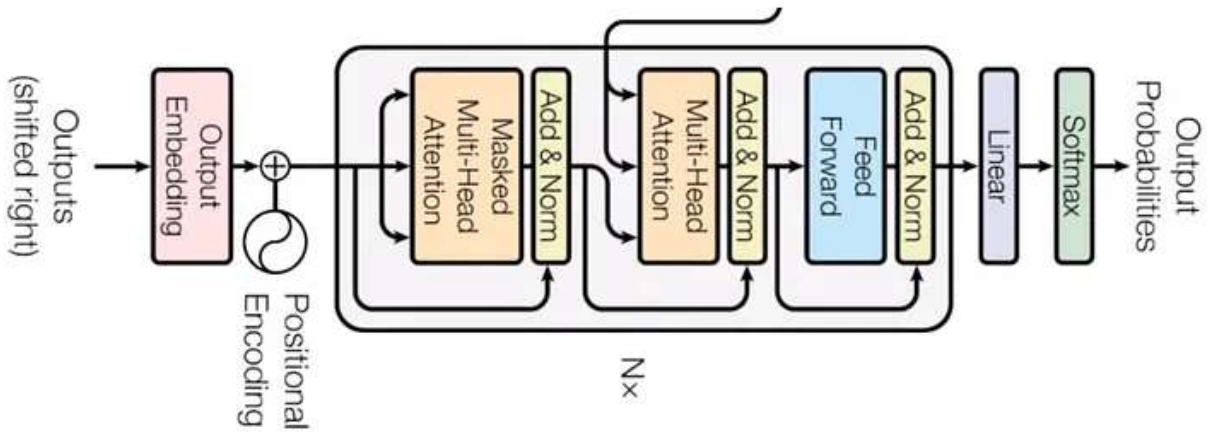
Feed Forward Network accepts attention vectors "one at a time". And the best thing here is unlike the case of RNN, here each of these attention vectors is *independent* of each other. So, **parallelization** can be applied here, *and that makes all the difference*.



Encoder's Output. source: arXiv:1706.03762 **[cs.CL].**

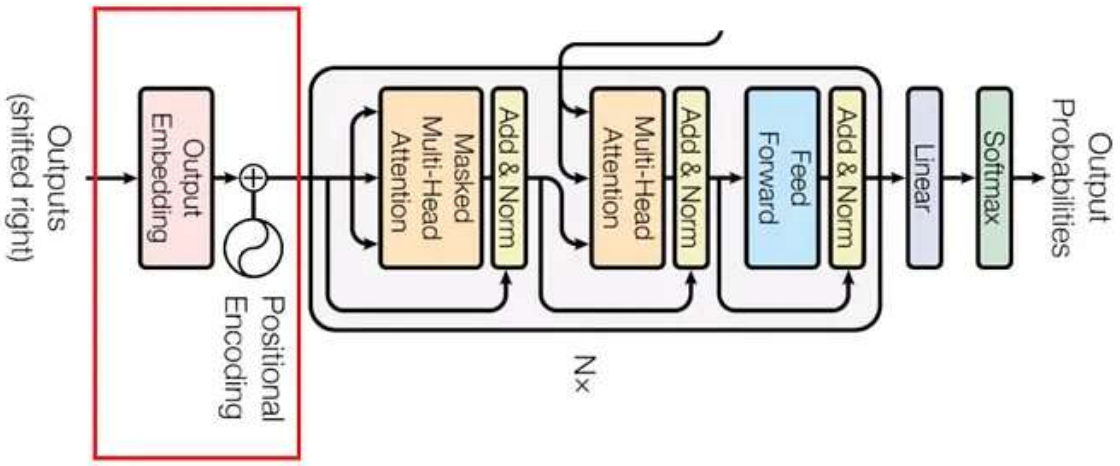
Now we can pass all the words at the same time into the encoder block, and get the set of Encoded Vectors for every word simultaneously.

2. Decoder Block -



source: arXiv:1706.03762 **[cs.CL].**

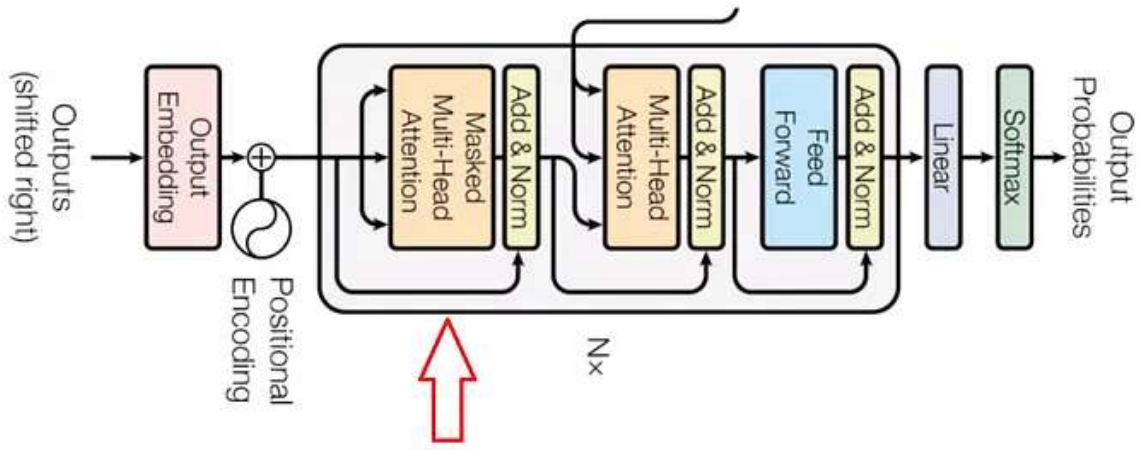
Now, like if we are training a translator for English to the French language, so for training we need to give an English sentence along with its translated French sentence for the model to learn. So, our English sentences pass through Encoder Block, and French sentences pass through the Decoder Block.



source: arXiv:1706.03762 **[cs.CL].**

At first, we have the Embedding layer and Positional encoder part which changes the words into respective vectors, It is similar to what we have seen in the Encoder part.

Masked Multi-Head Attention Part -



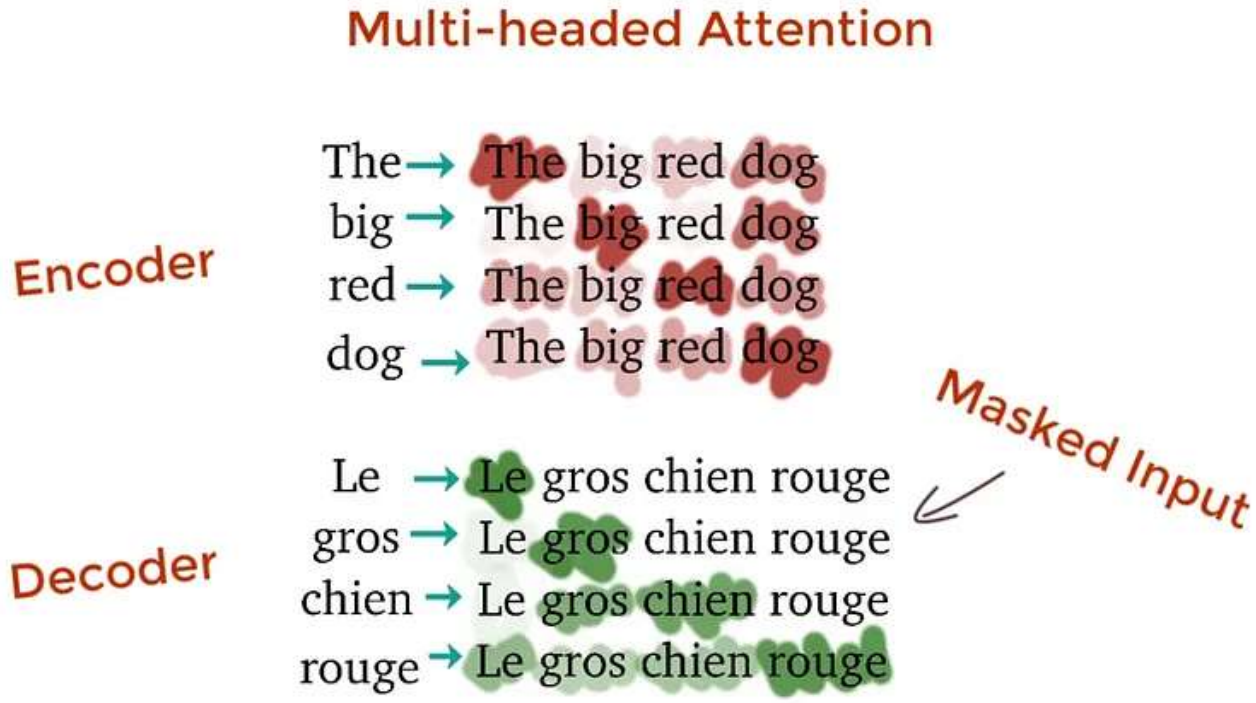
source: arXiv:1706.03762 **[cs.CL].**

Now it will pass through the self-attention block, where attention vectors are generated for every word in French sentences to represent how much each word is related to every word in the same sentence. (Just like we saw in Encoder Part).

But this block is called the **Masked Multi-Head Attention Block**, and I am going to explain in simple terms-

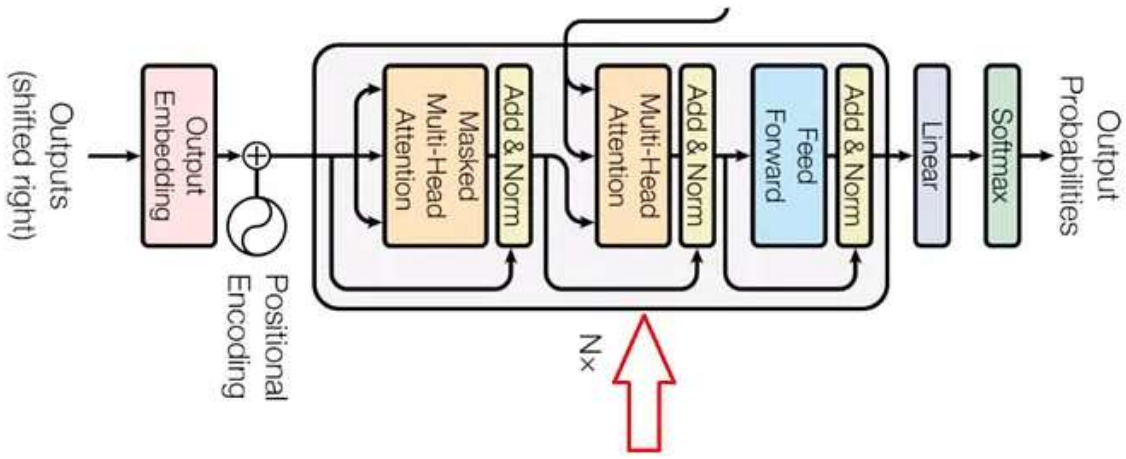
For that, we need to know how the learning mechanism works. First, we give an English word, it will translate in its French version itself using previous results, then it will match and compare with the actual French translation (which we fed in the decoder block). After comparing both, it will update its matrix value. This is how it will learn after several iterations.

What we observe is that we need to hide the next French word, so that at first it will predict the next word itself using previous results, without knowing the real translated word. For learning to take place, it will make no sense if it already knows the next French word. Therefore, we need to hide (mask) it.



This is an example of English-French Translation. source: CodeEmporium (CC0).

We can take any word from the English sentence, but we can only take the previous word of the French sentence to learn. So, while performing the parallelization with matrix operation, we make sure that the matrix should **mask** the words appearing later by transforming them into 0's so that the attention network can't use them.



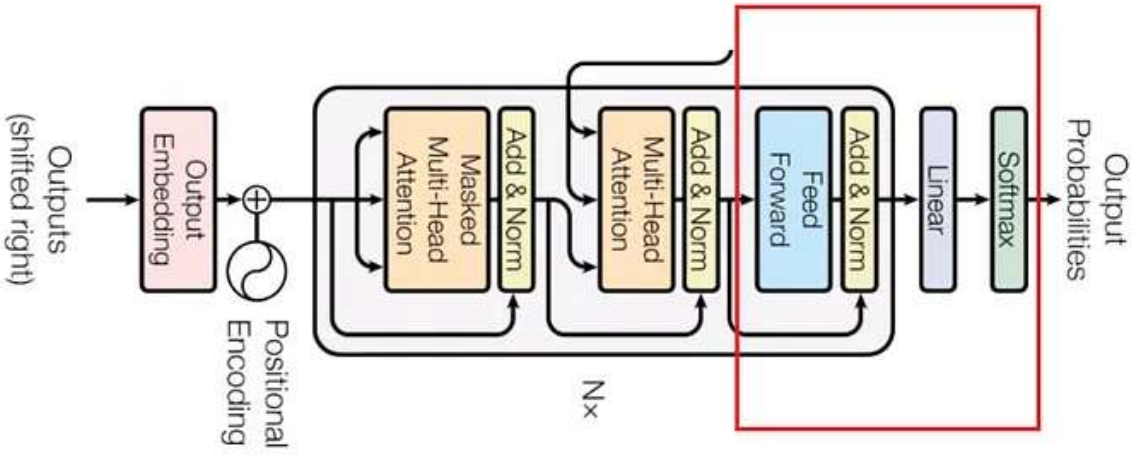
source: arXiv:1706.03762 **[cs.CL].**

Now, the resulting attention vectors from the previous layer and the vectors from the Encoder Block are passed into another **Multi-Head Attention Block.** (*this part is*

where the results from the encoder block also come into the picture. In the diagram also it is clearly visible that the results from the Encoder blocks are coming here.). That's why it is called **Encoder-Decoder Attention Block.**

As we have one vector of every word for each English and French sentence. This block actually does the mapping of English and French words and finds out the relation between them. So, this is the part where the main English to French word mapping happens.

The output of this block is attention vectors for every word in English and French sentences. *Each vector represents the relationship with other words in both languages.*



source: arXiv:1706.03762 **[cs.CL].**

Now we pass each attention vector into a feed-forward unit, it will make the output vectors form into something which is easily acceptable by another decoder block or a linear layer.

A linear layer is another feed-forward layer. It is used to expand the dimensions into numbers of words in the French language after translation.

Now it is passed through a Softmax Layer, which transforms the input into a probability distribution, which is human interpretable.

And the resulting word is produced with the highest probability after translation.

Below is the example that was illustrated in *Google's AI Blog [6]*, I put it here for your reference.

Overview - Working of Transformer Network. source: Google AI (CC0).

The Transformer starts by generating initial representations, or embeddings, for each word. These are represented by the unfilled circles. Then, using self-attention, it aggregates information from all of the other words, generating a new representation per word informed by the entire context, represented by the filled balls. This step is then repeated multiple times in parallel for all words, successively generating new representations.

The decoder operates similarly, but generates one word at a time, from left to right. It attends not only to the other previously generated words but also to the final representations generated by the encoder.

So, this is how the transformer works, and it is now the state-of-the-art technique in NLP. It is giving wonderful results, using a ***self-attention mechanism*** and also solves the parallelization issue. Even Google uses **BERT** that uses a transformer to pre-train models for common NLP applications.

References -

1. *Attention is all you need*; Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin.
2. *Effective Approaches to Attention-based Neural Machine Translation*; Minh-Thang Luong, Hieu Pham, Christopher D. Manning.
3. Colah's Blog.
4. Jay Alammar's Blog.
5. CodeEmporium : Youtube.

NLP | Attention | Nmt | Naturallanguageprocessing | Deep Learning

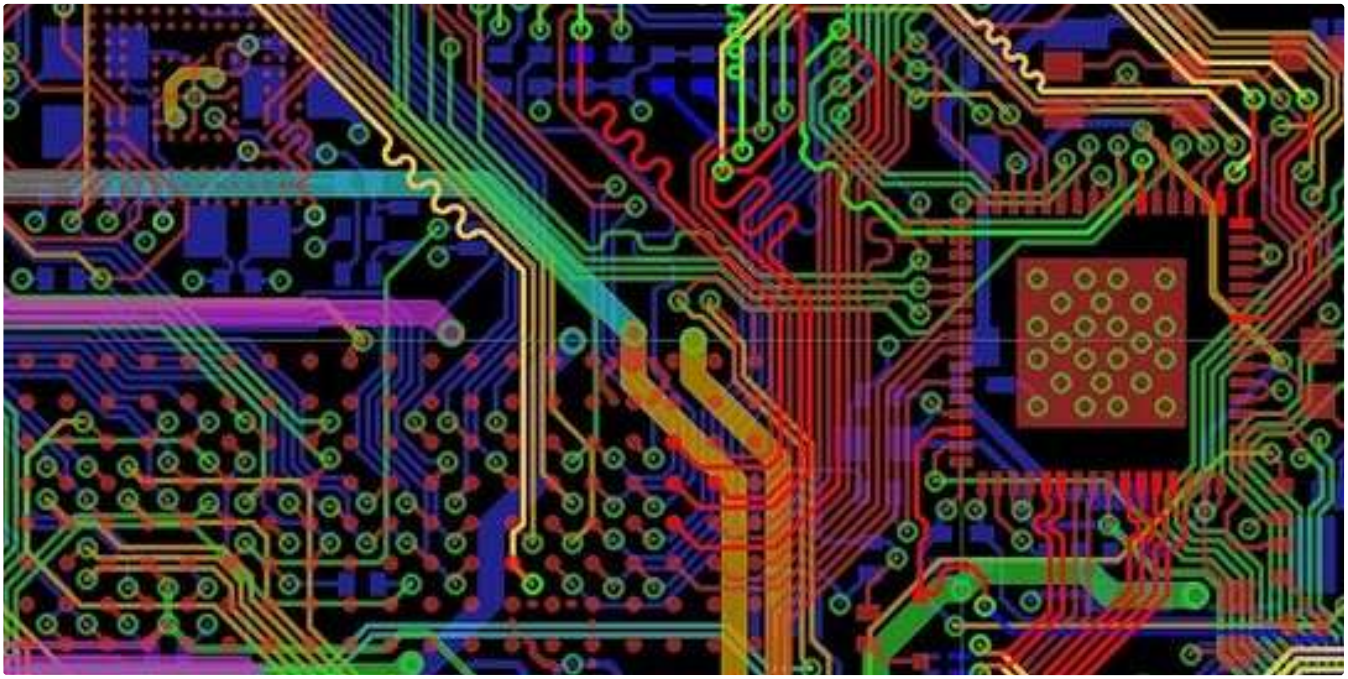
Follow

Written by Utkarsh Ankit

176 Followers · Writer for Towards Data Science

Data Scientist, Former ISRO, IIT Research Fellow (AI) [LinkedIn-https://www.linkedin.com/in/utkarsh-ankit] [GitHub-https://github.com/utkarsh-ankit]

More from Utkarsh Ankit and Towards Data Science



Utkarsh Ankit in Towards Data Science

Image Classification of PCBs and its Web Application (Flask)

This blog is about creating an Image Classification Model for PCBs and creating a Web Application for it using Flask.

10 min read · Aug 2, 2021

663

Bex T. in Towards Data Science

130 ML Tricks And Resources Curated Carefully From 3 Years (Plus Free eBook)

Each one is worth your time

48 min read · Aug 1

3.1K 10

Maxime Labonne in Towards Data Science

Fine-Tune Your Own Llama 2 Model in a Colab Notebook

A practical introduction to LLM fine-tuning

✨ · 12 min read · Jul 25

1.8K 33

Utkarsh Ankit in Towards Data Science

Semantic Segmentation of Aerial images Using Deep Learning

Pixel-wise image segmentation is a challenging and demanding task in computer vision and image processing. This blog is about segmentation...

10 min read · Feb 4, 2019

1.8K 7

See all from Utkarsh Ankit

See all from Towards Data Science

Recommended from Medium

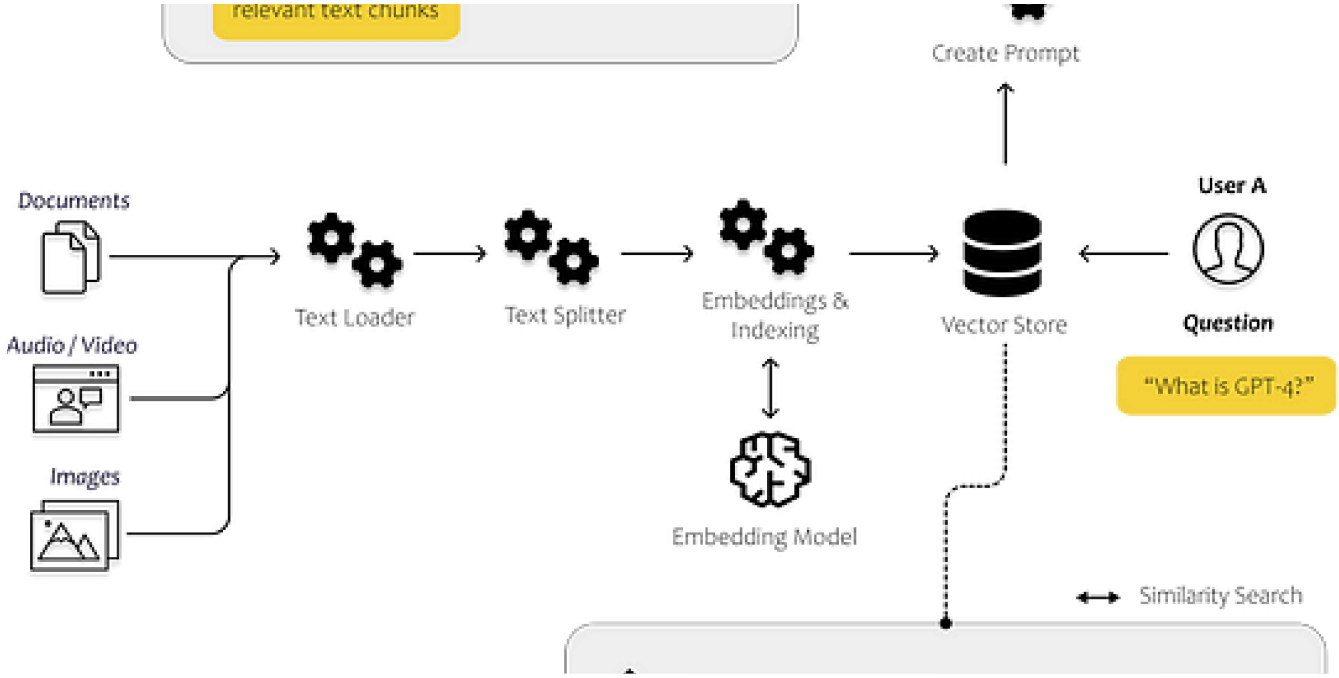
TeeTracker

Fine-tuning LLMs

Tasks to finetune

6 min read · Jul 22

111



Dominik Polzer in Towards Data Science

All You Need to Know to Build Your First LLM App

A step-by-step tutorial to document loaders, embeddings, vector stores and prompt templates

✦ · 26 min read · Jun 22

4.6K 42

Lists

Natural Language Processing
548 stories · 169 saves

The New Chatbots: ChatGPT, Bard, and Beyond
13 stories · 94 saves

Now in AI: Handpicked by Better Programming
266 stories · 106 saves

Practical Guides to Machine Learning
10 stories · 339 saves

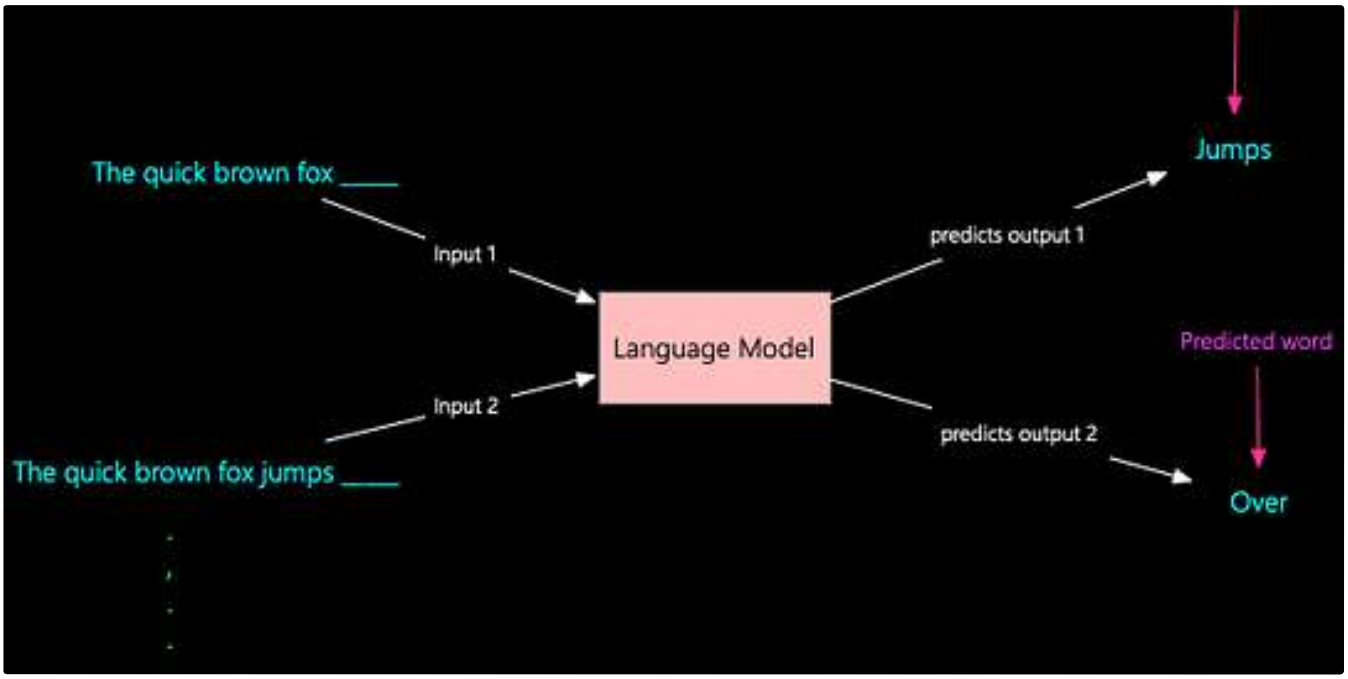
Tomas Vykruta

Understanding Causal LLM's, Masked LLM's, and Seq2Seq: A Guide to Language Model Training...

In the world of natural language processing (NLP), choosing the right training approach is crucial for the success of your language model...

7 min read · Apr 30

8



Pradeep Menon

Introduction to Large Language Models and the Transformer Architecture

ChatGPT is making waves worldwide, attracting over 1 million users in record time. As a CTO for startups, I discuss this revolutionary...

7 min read · Mar 9

110

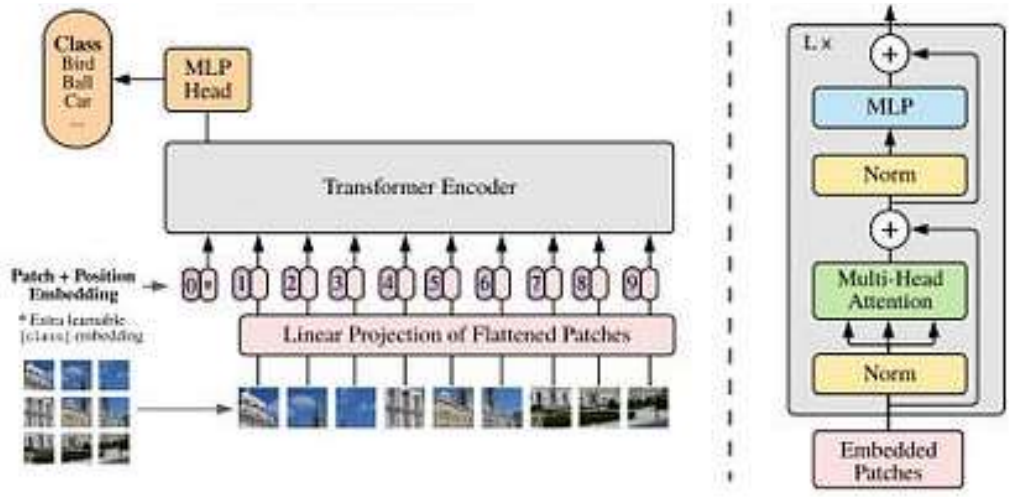


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence. The illustration of the Transformer encoder was inspired by

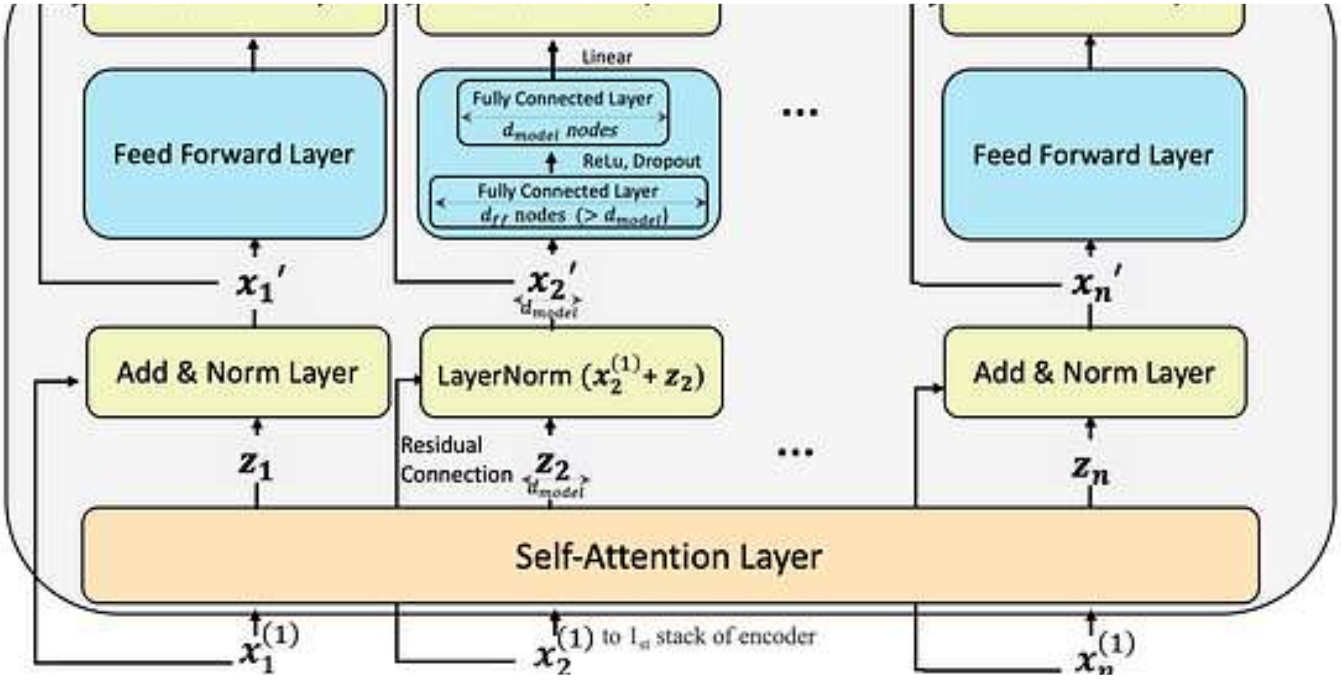
Fahim Rustamy, PhD

Vision Transformers vs. Convolutional Neural Networks

This blog post is inspired by the paper titled AN IMAGE IS WORTH 16×16 WORDS: TRANSFORMERS FOR IMAGE RECOGNITION AT SCALE from google's...

7 min read · Jun 4

386 6



Yule Wang, PhD

Step-by-Step Illustrated Explanations of Transformer

with detailed explained Attention Mechanism

8 min read · Feb 27