**How do Transformers Work in NLP?**

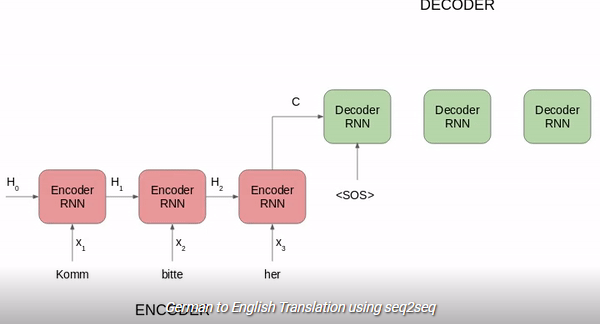
## Sequence-to-Sequence Models – A Backdrop

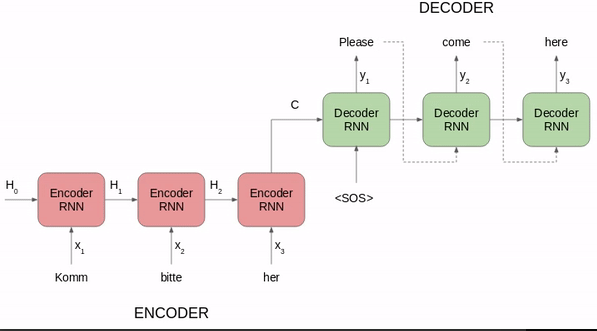
* **Sequence-to-sequence (seq2seq) models in NLP are used to convert sequences of Type A to sequences of Type B. For example, translation of English sentences to German sentences is a sequence-to-sequence task.**
* **Recurrent Neural Network (RNN) based sequence-to-sequence models have garnered a lot of traction ever since they were introduced in 2014. Most of the data in the current world are in the form of sequences – it can be a number sequence, text sequence, a video frame sequence or an audio sequence.**
* **The performance of these seq2seq models was further enhanced with the addition of the Attention Mechanism in 2015.**

**These sequence-to-sequence models are pretty versatile and they are used in a variety of NLP tasks, such as:**

* **Machine Translation**
* **Text Summarization**
* **Speech Recognition**
* **Question-Answering System, and so on**

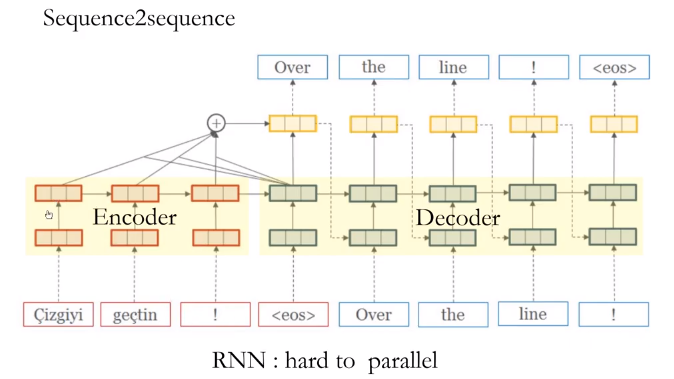
### RNN based Sequence-to-Sequence Model

****

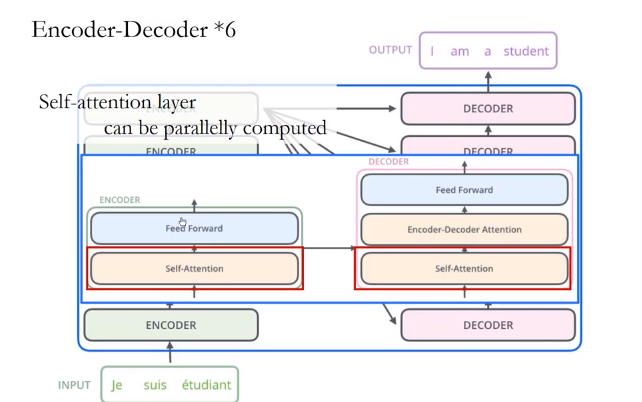
****

**The above seq2seq model is converting a German phrase to its English counterpart. Let’s break it down:**

* **Both Encoder and Decoder are RNNs**
* **At every time step in the Encoder, the RNN takes a word vector (xi) from the input sequence and a hidden state (Hi) from the previous time step**
* **The hidden state is updated at each time step**
* **The hidden state from the last unit is known as the context vector. This contains information about the input sequence**
* **This context vector is then passed to the decoder and it is then used to generate the target sequence (English phrase)**
* **If we use the Attention mechanism, then the weighted sum of the hidden states are passed as the context vector to the decoder**

****

**Sequence to sequence transformers the sentences are fed in to the model word by word and they have the information about the previous context not the forward context. The information encoded on encoder layer and finally passed to the decoder. If calculate the information for ‘x’ word then the information from the previous word mostly its lost if the sequence is large.**

****

**We can have multiple encoder and decoder parallel.6 cascaded layers are there in transformers.**

**Part 1: Sequence to Sequence Learning and Attention**

**Sequence-to-Sequence (or Seq2Seq) is a neural net that transforms a given sequence of elements, such as the sequence of words in a sentence, into another sequence.**

**Seq2Seq models are particularly good at translation, where the sequence of words from one language is transformed into a sequence of different words in another language. A popular choice for this type of model is Long-Short-Term-Memory (LSTM)-based models. With sequence-dependent data, the LSTM modules can give meaning to the sequence while remembering (or forgetting) the parts it finds important (or unimportant). Sentences, for example, are sequence-dependent since the order of the words is crucial for understanding the sentence. LSTM are a natural choice for this type of data.**

**Seq2Seq models consist of an Encoder and a Decoder. The Encoder takes the input sequence and maps it into a higher dimensional space (n-dimensional vector). That abstract vector is fed into the Decoder which turns it into an output sequence. The output sequence can be in another language, symbols, a copy of the input, etc.**

**Imagine the Encoder and Decoder as human translators who can speak only two languages. Their first language is their mother tongue, which differs between both of them (e.g. German and French) and their second language an imaginary one they have in common. To translate German into French, the Encoder converts the German sentence into the other language it knows, namely the imaginary language. Since the Decoder is able to read that imaginary language, it can now translates from that language into French. Together, the model (consisting of Encoder and Decoder) can translate German into French!**

**Suppose that, initially, neither the Encoder or the Decoder is very fluent in the imaginary language. To learn it, we train them (the model) on a lot of examples.**

**A very basic choice for the Encoder and the Decoder of the Seq2Seq model is a single LSTM for each of them.**

**You’re wondering when the Transformer will finally come into play, aren’t you?**

**We need one more technical detail to make Transformers easier to understand: *Attention*. The attention-mechanism looks at an input sequence and decides at each step which other parts of the sequence are important. It sounds abstract, but let me clarify with an easy example: When reading this text, you always focus on the word you read but at the same time your mind still holds the important keywords of the text in memory in order to provide context.**

**An attention-mechanism works similarly for a given sequence. For our example with the human Encoder and Decoder, imagine that instead of only writing down the translation of the sentence in the imaginary language, the Encoder also writes down keywords that are important to the semantics of the sentence, and gives them to the Decoder in addition to the regular translation. Those new keywords make the translation much easier for the Decoder because it knows what parts of the sentence are important and which key terms give the sentence context.**

**In other words, for each input that the LSTM (Encoder) reads, the attention-mechanism takes into account several other inputs at the same time and decides which ones are important by attributing different weights to those inputs. The Decoder will then take as input the encoded sentence and the weights provided by the attention-mechanism.**

**What is a Transformer?**

**The Transformer in NLP is a novel architecture that aims to solve sequence-to-sequence tasks while handling long-range dependencies with ease. It relies entirely on self-attention to compute representations of its input and output WITHOUT using sequence-aligned RNNs or convolution.**

## Introduction to the Transformer

**The Transformer in NLP is a novel architecture that aims to solve sequence-to-sequence tasks while handling long-range dependencies with ease.**

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

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

**Few things to know before diving into Transformers**

## Self-Attention:

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

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

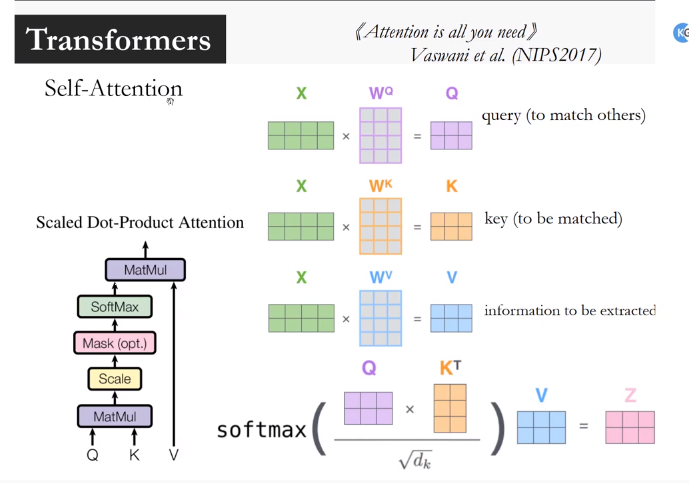
**In simpler terms, self attention helps us create similar connections but within the same sentence. Look at the following example:**

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

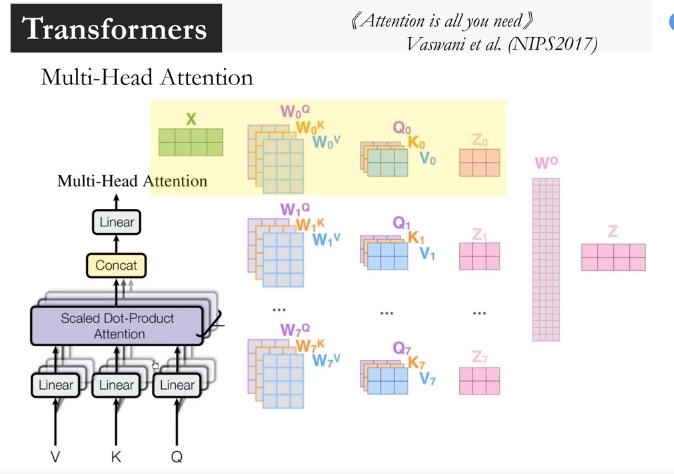
**By changing one word “*full*” — > “*empty*” the reference object for “*it*” changed. If we are translating such a sentence, we will want to know the word “*it*” refers to.**

## The three kinds of Attention possible in a model:

1. ***Encoder-Decoder Attention*: Attention between the input sequence and the output sequence.**
2. ***Self attention in the input sequence*: Attends to all the words in the input sequence.**
3. ***Self attention in the output sequence:* One thing we should be wary of here is that the scope of self attention is limited to the words that occur before a given word. This prevents any information leaks during the training of the model. This is done by masking the words that occur after it for each step. So for step 1, only the first word of the output sequence is NOT masked, for step 2, the first two words are NOT masked and so on.**

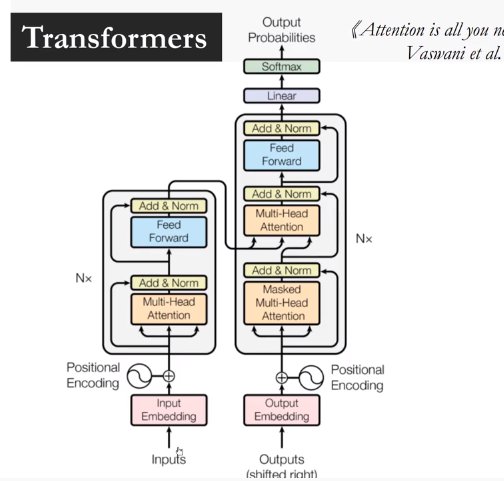
****

## Attention is like put attention on these particular words which are most important in a particular sentence. It takes 3 inputs. QKV. X are the encoded inputs. WQ, Wk, Wv are the feature vectors & they are multiplied together and produces the Q,k,v matrix . Finally the softmax function has the o/p actually. Z final matrix(classification/seq2seq matrix)

****

**In multihead attention many layers are cascaded together. The information from the sentence/word can be fed together. Here the all the words in the sentence fed together and the context are available together & then complex mathematical operations happens for each of these attention layer and finally the results been calculated Wo (z0,z1,z2 mutltipied)together to get the final results z.**

### Understanding Transformer’s Model Architecture

****

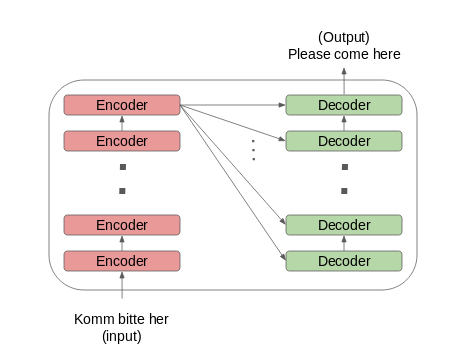
**The above image is a superb illustration of Transformer’s architecture. Let’s first focus on the Encoder and Decoder parts only.**

**Now focus on the below image. The Encoder block has 1 layer of a Multi-Head Attention followed by another layer of Feed Forward Neural Network. The decoder, on the other hand, has an extra Masked  Masked Multi-Head Attention.**

****

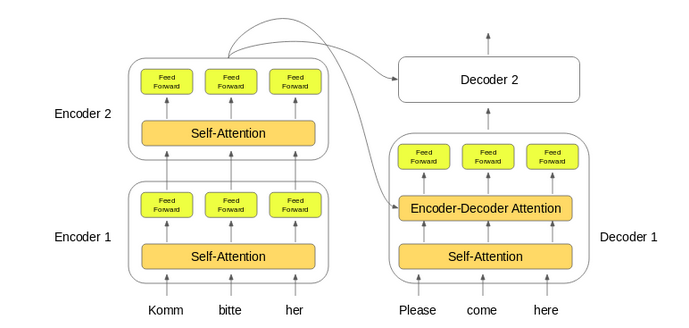
**The encoder and decoder blocks are actually multiple identical encoders and decoders stacked on top of each other. Both the encoder stack and the decoder stack have the same number of units.**

**The number of encoder and decoder units is a hyperparameter. In the paper, 6 encoders and decoders have been used.**

****

**Let’s see how this setup of the encoder and the decoder stack works:**

* **The word embeddings of the input sequence are passed to the first encoder**
* **These are then transformed and propagated to the next encoder**
* **The output from the last encoder in the encoder-stack is passed to all the decoders in the decoder-stack as shown in the figure below:**

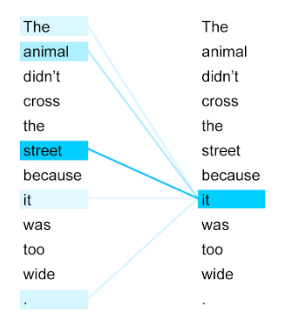
****

**An important thing to note here – in addition to the self-attention and feed-forward layers, the decoders also have one more layer of Encoder-Decoder Attention layer. This helps the decoder focus on the appropriate parts of the input sequence.**

### Getting a Hang of Self-Attention

**According to the paper:**

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

****

**Take a look at the above image. Can you figure out what the term “it” in this sentence refers to?**

**Is it referring to the street or to the animal? It’s a simple question for us but not for an algorithm. When the model is processing the word “it”, self-attention tries to associate “it” with “animal” in the same sentence.**

**Self-attention allows the model to look at the other words in the input sequence to get a better understanding of a certain word in the sequence. Now, let’s see how we can calculate self-attention.**

### Calculating Self-Attention

**I have divided this section into various steps for ease of understanding.**

**1. First, we need to create three vectors from each of the encoder’s input vectors:**

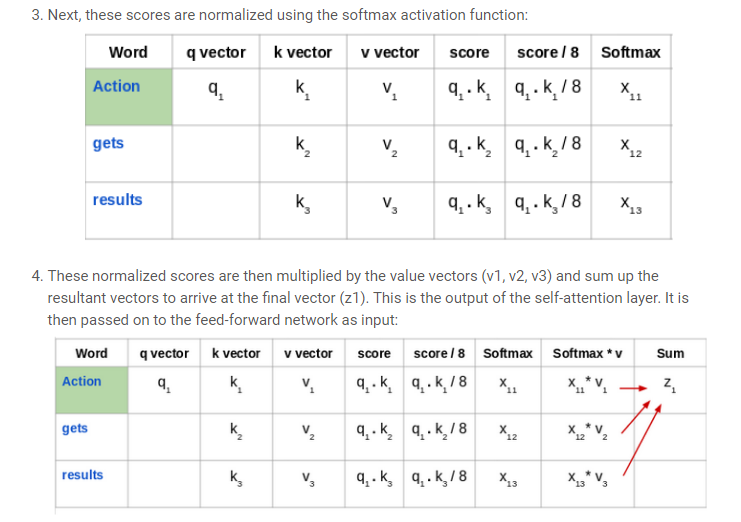
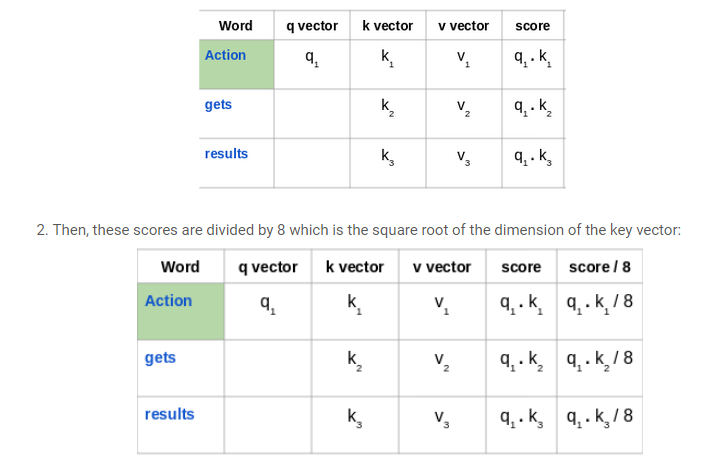
* 1. **Query Vector**
  2. **Key Vector**
  3. **Value Vector.**

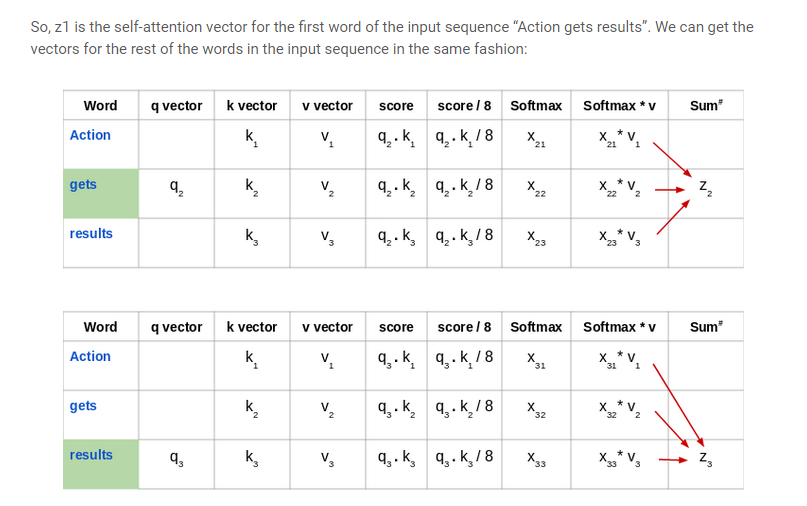
**These vectors are trained and updated during the training process. We’ll know more about their roles once we are done with this section**

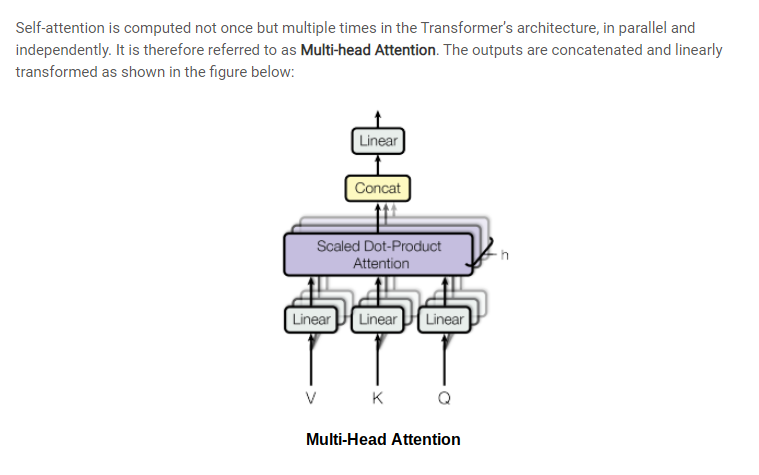
**2. Next, we will calculate self-attention for every word in the input sequence**

**3. Consider this phrase – “Action gets results”. To calculate the self-attention for the first word “Action”, we will calculate scores for all the words in the phrase with respect to “Action”. This score determines the importance of other words when we are encoding a certain word in an input sequence**

* 1. **The score for the first word is calculated by taking the dot product of the Query vector (q1) with the keys vectors (k1, k2, k3) of all the words:**

****

****

****

**According to the paper “Attention Is All You Need”:**

***“Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions.”***

### Keys, Values, and Queries:

**The three random words I just threw at you in this heading are vectors created as abstractions are useful for calculating self attention, more details on each below. These are calculated by multiplying your input vector(*X*) with weight matrices that are learnt while training.**

* ***Query Vector*: *q*= *X \* Wq.*Think of this as the current word.**
* ***Key Vector:* *k*= *X \* Wk.*Think of this as an indexing mechanism for Value vector. Similar to how we have key-value pairs in hash maps, where keys are used to uniquely index the values.**
* ***Value Vector:* *v*= *X \* Wv.*Think of this as the information in the input word.**

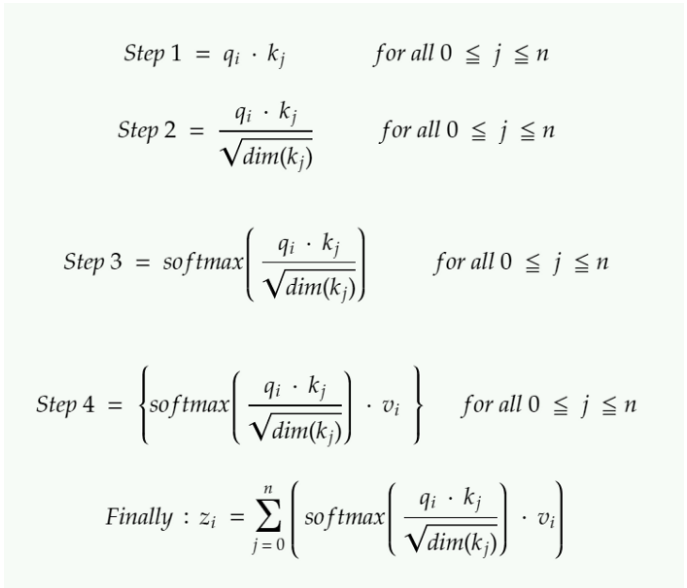
**What we want to do is take query *q*and find the most similar key *k*, by doing a dot product for *q*and *k*. The closest query-key product will have the highest value, followed by a softmax that will drive the *q.k*with smaller values close to 0 and *q.k*with larger values towards 1. This softmax distribution is multiplied with *v.*The value vectors multiplied with ~1 will get more attention while the ones ~0 will get less. The sizes of these *q, k*and*v*vectors are referred to as “*hidden size*” by various implementations.**

### All these matrices *Wq, Wk* and *Wv* are learnt while being jointly trained during the model training.

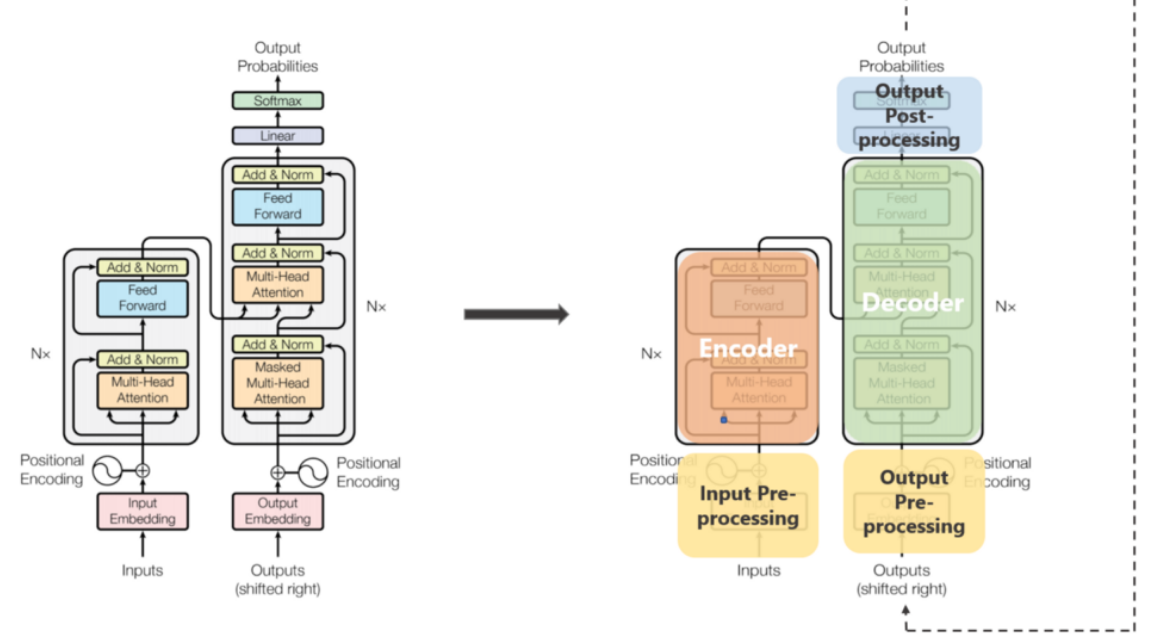
### Calculating Self attention from q, k and v:

### C:\Users\dell\Desktop\w.jpg

### C:\Users\dell\Desktop\a.jpg

****

**The Transformer**

****

## Beast #1: Encoder-Decoder stacks

***Encoder*: The encoder maps an input sequence of symbol representations *(x*₁*, …, x*ₙ*)* to a sequence of representations *z = (z*₁*, …, z*ₙ*)*. Think of them as the outputs from self attention with some post-processing.**

**Each encoder has two sub-layers.**

1. **A multi-head self attention mechanism on the input vectors (Think parallelized and efficient sibling of self attention).**
2. **A simple, position-wise fully connected feed-forward network (Think post-processing).**

**Decoder: Given z, the decoder then generates an output sequence (y₁, …, yₘ) of symbols one element at a time.**

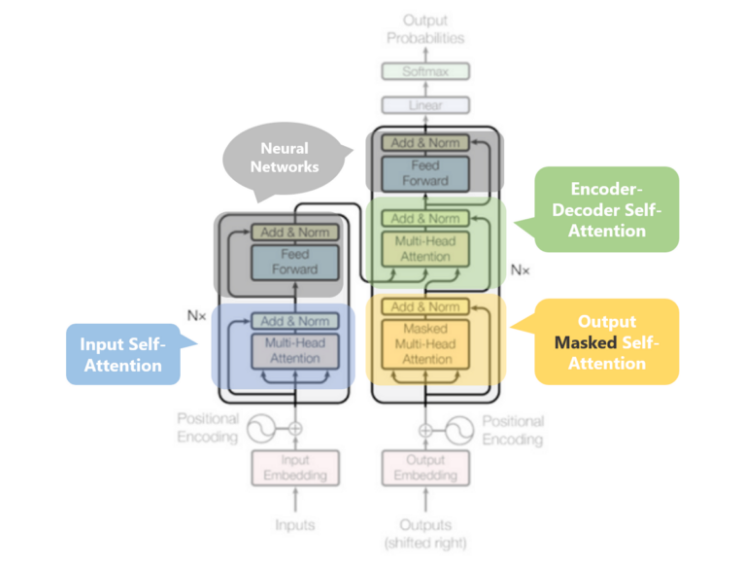
**Each decoder has three sub-layers.**

1. **A masked multi-head self attention mechanism on the output vectors of the previous iteration.**
2. **A multi-head attention mechanism on the output from encoder and masked multi-headed attention in decoder.**
3. **A simple, position-wise fully connected feed-forward network (think post-processing).**

**A few additional points:**

* **In the original paper, 6 layers were present in the encoder stack (2 sub-layer version) and 6 in the decoder stack (3 sub-layer version).**
* **All sub-layers in the model, as well as the embedding layers, produce outputs of the same dimension. This is done to facilitate the residual connections.**

## Beast #2 Inside Encoder-Decoder stacks — Multi-Head Attention:

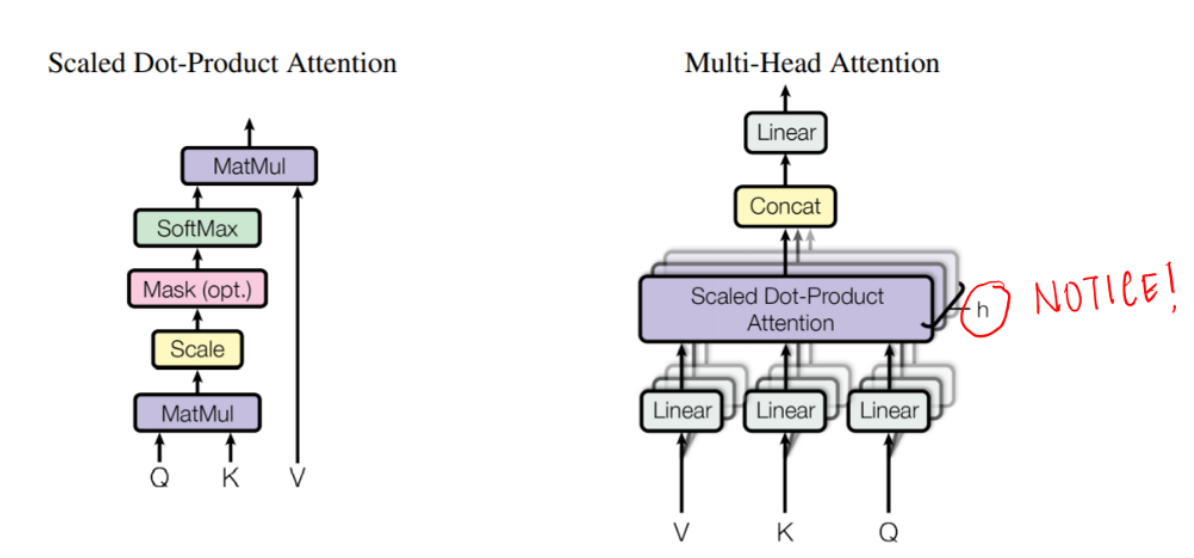
****

## We just noted that the output of each sub-layer needs to be of the same dimension which is 512 in our paper. => zᵢ needs to be of 512 dimensions. => vᵢ needs to be of 512 dimensions as zᵢ are just sort of weighted sums of vᵢs.

## Additionally, we want to allow the model to focus on different positions is by calculating self attention multiple times with different sets of q, k and v vectors, then take an average of all those outputs to get our final z.

## So instead of dealing with these humongous vectors and averaging multiple outputs, we reduce the size of our k,q and v vectors to some smaller dimension — reduces size of Wq, Wk, and Wv matrices as well. We keep the multiple sets (h) of k,q and v and refer to each set as an “attention head”, hence the name multi-headed attention. And lastly, instead of averaging to get final z, we concatenate them.

## The size of the concatenated vector will be too large to be fed to the next sub-layer, so we scale it down by multiplying it with another learnt matrix Wo.

****

## Multiple attention heads allowed the model to jointly attend to information from different representation sub-spaces at different positions which was inhibited by averaging in a single attention head.

## Beast #3— Input and Output Pre-processing:

## The input words are represented using some form of embedding. This is done for both encoder and decoder.

## Word embedding on their own lack any positional information which is achieved in RNNs by virtue of their sequential nature. Meanwhile in self-attention, due to softmax, any such positional information is lost.

## To preserve the positional information, the transformer injects a vector to individual input embeddings (could be using word embeddings for corresponding to the input words). These vectors follow a specific periodic function (Example: combination of various sines/cosines having different frequency, in short not in sync with each other) that the model learns and is able to determine the position of individual word wrt each other based on the values .

## This injected vector is called “*positional encoding*” and are added to the input embeddings at the bottoms of both encoder and decoder stacks.

## Beast #4 — Decoder stack: Revisited

## The output of the decoder stack at each step is fed back to the decoder in the next time step — pretty similar to how outputs from previous steps in RNNs were used as next hidden states. And just as we did with the encoder inputs, we embed and add positional encoding to those decoder inputs to preserve the position of each word. This positional encoding + word embedding combo is then fed into a masked multi-headed self attention.

## This self-attention sub-layer in the decoder stack is modified to prevent positions from attending to subsequent positions — you can’t look at future words. This masking ensures that the predictions for position *i* can depend only on the known outputs at positions less than *i*.

## The outputs from the encoder stack are then used as multiple sets of key vectors *k*and value vectors *v*, for the “encoder decoder attention” — shown in green in the diagram — layer. It helps the decoder focus on the contextually relevant parts in the input sequence for that step. (The part similar to global attention vectors.) The *q*vector comes from the “output self attention” layer.

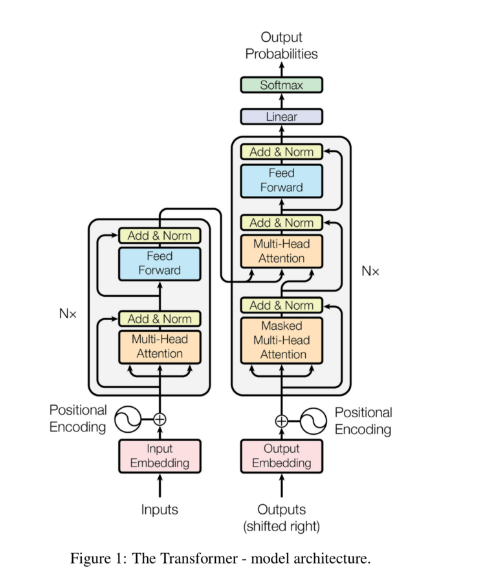
## Once we get the output from the decoder, we do a softmax again to select the final probabilities of words.

# ****Part 2: The Transformer****

**The paper ‘Attention Is All You Need’ introduces a novel architecture called Transformer. As the title indicates, it uses the attention-mechanism we saw earlier. Like LSTM, Transformer is an architecture for transforming one sequence into another one with the help of two parts (Encoder and Decoder), but it differs from the previously described/existing sequence-to-sequence models because it does not imply any Recurrent Networks (GRU, LSTM, etc.).**

**Recurrent Networks were, until now, one of the best ways to capture the timely dependencies in sequences. However, the team presenting the paper proved that an architecture with only attention-mechanisms without any RNN (Recurrent Neural Networks) can improve on the results in translation task and other tasks! One improvement on Natural Language Tasks is presented by a team introducing BERT:**[**BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding**](https://arxiv.org/abs/1810.04805)**.**

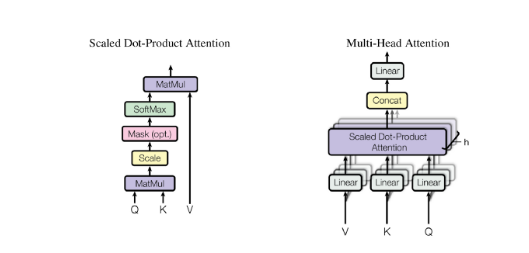
**So, what exactly is a Transformer?**

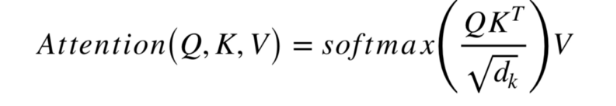
****

**The Encoder is on the left and the Decoder is on the right. Both Encoder and Decoder are composed of modules that can be stacked on top of each other multiple times, which is described by Nx in the figure. We see that the modules consist mainly of Multi-Head Attention and Feed Forward layers. The inputs and outputs (target sentences) are first embedded into an n-dimensional space since we cannot use strings directly.**

**One slight but important part of the model is the positional encoding of the different words. Since we have no recurrent networks that can remember how sequences are fed into a model, we need to somehow give every word/part in our sequence a relative position since a sequence depends on the order of its elements. These positions are added to the embedded representation (n-dimensional vector) of each word.**

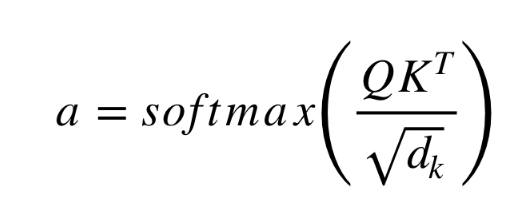
**Let’s have a closer look at these Multi-Head Attention bricks in the model:**

****

**Let’s start with the left description of the attention-mechanism. It’s not very complicated and can be described by the following equation:**

**Q is a matrix that contains the query (vector representation of one word in the sequence), K are all the keys (vector representations of all the words in the sequence) and V are the values, which are again the vector representations of all the words in the sequence. For the encoder and decoder, multi-head attention modules, V consists of the same word sequence than Q. However, for the attention module that is taking into account the encoder and the decoder sequences, V is different from the sequence represented by Q.**

**To simplify this a little bit, we could say that the values in V are multiplied and summed with some attention-weights a, where our weights are defined by:**

****

**This means that the weights a are defined by how each word of the sequence (represented by Q) is influenced by all the other words in the sequence (represented by K). Additionally, the SoftMax function is applied to the weights a to have a distribution between 0 and 1. Those weights are then applied to all the words in the sequence that are introduced in V (same vectors than Q for encoder and decoder but different for the module that has encoder and decoder inputs).**

**The righthand picture describes how this attention-mechanism can be parallelized into multiple mechanisms that can be used side by side. The attention mechanism is repeated multiple times with linear projections of Q, K and V. This allows the system to learn from different representations of Q, K and V, which is beneficial to the model. These linear representations are done by multiplying Q, K and V by weight matrices W that are learned during the training.**

**Those matrices Q, K and V are different for each position of the attention modules in the structure depending on whether they are in the encoder, decoder or in-between encoder and decoder. The reason is that we want to attend on either the whole encoder input sequence or a part of the decoder input sequence. The multi-head attention module that connects the encoder and decoder will make sure that the encoder input-sequence is taken into account together with the decoder input-sequence up to a given position.**

**After the multi-attention heads in both the encoder and decoder, we have a pointwise feed-forward layer. This little feed-forward network has identical parameters for each position, which can be described as a separate, identical linear transformation of each element from the given sequence.**

## ****Training****

**How to train such a ‘beast’? Training and inferring on Seq2Seq models is a bit different from the usual classification problem. The same is true for Transformers.**

**We know that to train a model for translation tasks we need two sentences in different languages that are translations of each other. Once we have a lot of sentence pairs, we can start training our model. Let’s say we want to translate French to German. Our encoded input will be a French sentence and the input for the decoder will be a German sentence. However, the decoder input will be shifted to the right by one position. ..Wait, why?**

**One reason is that we do not want our model to learn how to copy our decoder input during training, but we want to learn that given the encoder sequence and a particular decoder sequence, which has been already seen by the model, we predict the next word/character.**

**If we don’t shift the decoder sequence, the model learns to simply ‘copy’ the decoder input, since the target word/character for position i would be the word/character i in the decoder input. Thus, by shifting the decoder input by one position, our model needs to predict the target word/character for position i having only seen the word/characters 1, …, i-1 in the decoder sequence. This prevents our model from learning the copy/paste task. We fill the first position of the decoder input with a start-of-sentence token, since that place would otherwise be empty because of the right-shift. Similarly, we append an end-of-sentence token to the decoder input sequence to mark the end of that sequence and it is also appended to the target output sentence. In a moment, we’ll see how that is useful for inferring the results.**

**This is true for Seq2Seq models and for the Transformer. In addition to the right-shifting, the Transformer applies a mask to the input in the first multi-head attention module to avoid seeing potential ‘future’ sequence elements. This is specific to the Transformer architecture because we do not have RNNs where we can input our sequence sequentially. Here, we input everything together and if there were no mask, the multi-head attention would consider the whole decoder input sequence at each position.**

**The process of feeding the correct shifted input into the decoder is also called Teacher-Forcing, as described in**[**this blog**](https://machinelearningmastery.com/teacher-forcing-for-recurrent-neural-networks/)**.**

**The target sequence we want for our loss calculations is simply the decoder input (German sentence) without shifting it and with an end-of-sequence token at the end.**

## ****Inference****

**Inferring with those models is different from the training, which makes sense because in the end we want to translate a French sentence without having the German sentence. The trick here is to re-feed our model for each position of the output sequence until we come across an end-of-sentence token.**

**A more step by step method would be:**

* **Input the full encoder sequence (French sentence) and as decoder input, we take an empty sequence with only a start-of-sentence token on the first position. This will output a sequence where we will only take the first element.**
* **That element will be filled into second position of our decoder input sequence, which now has a start-of-sentence token and a first word/character in it.**
* **Input both the encoder sequence and the new decoder sequence into the model. Take the second element of the output and put it into the decoder input sequence.**
* **Repeat this until you predict an end-of-sentence token, which marks the end of the translation.**

**We see that we need multiple runs through our model to translate our sentence.**

**Transformers**

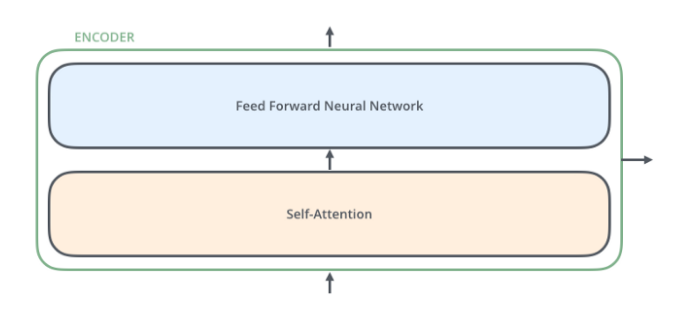
**To solve the problem of parallelization, Transformers try to solve the problem by using Convolutional Neural Networks together with attention models. Attention boosts the speed of how fast the model can translate from one sequence to another.**

**Let’s take a look at how Transformer works. Transformer is a model that uses attention to boost the speed. More specifically, it uses self-attention.**

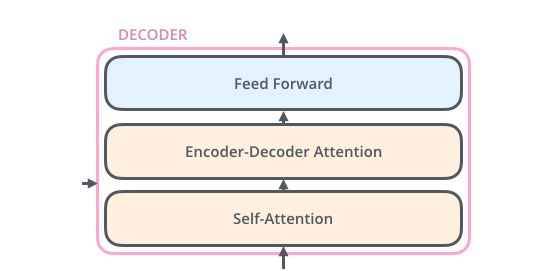
### C:\Users\dell\Desktop\ll.jpg

### Internally, the Transformer has a similar kind of architecture as the previous models above. But the Transformer consists of six encoders and six decoders.C:\Users\dell\Desktop\eeeee.jpg

**Each encoder is very similar to each other. All encoders have the same architecture. Decoders share the same property, i.e. they are also very similar to each other. Each encoder consists of two layers: Self-attention and a feed Forward Neural Network.**

****

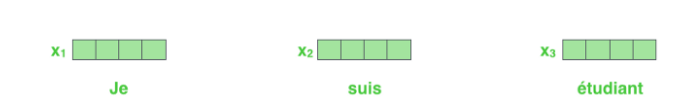
**The encoder’s inputs first flow through a self-attention layer. It helps the encoder look at other words in the input sentence as it encodes a specific word. The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence.**

****

# Self-Attention

**Note: This section comes from Jay Allamar**[**blog post**](http://jalammar.github.io/illustrated-transformer/)

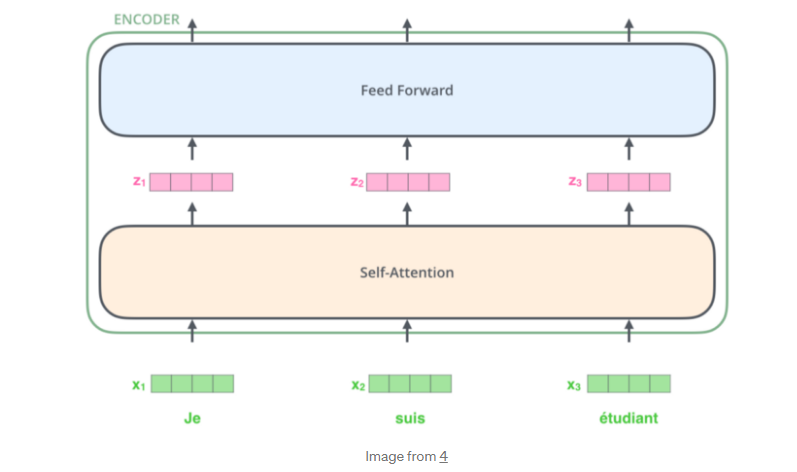
**Let’s start to look at the various vectors/tensors and how they flow between these components to turn the input of a trained model into an output. As is the case in NLP applications in general, we begin by turning each input word into a vector using an**[**embedding algorithm**](https://medium.com/deeper-learning/glossary-of-deep-learning-word-embedding-f90c3cec34ca)**.**

****

**Each word is embedded into a vector of size 512. We’ll represent those vectors with these simple boxes.**

**The embedding only happens in the bottom-most encoder. The abstraction that is common to all the encoders is that they receive a list of vectors each of the size 512.**

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

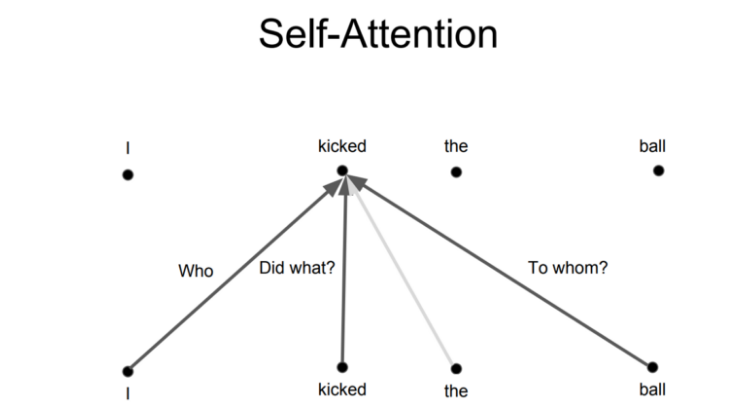
****

**Here we begin to see one key property of the Transformer, which is that 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.**

**Next, we’ll switch up the example to a shorter sentence and we’ll look at what happens in each sub-layer of the encoder.**

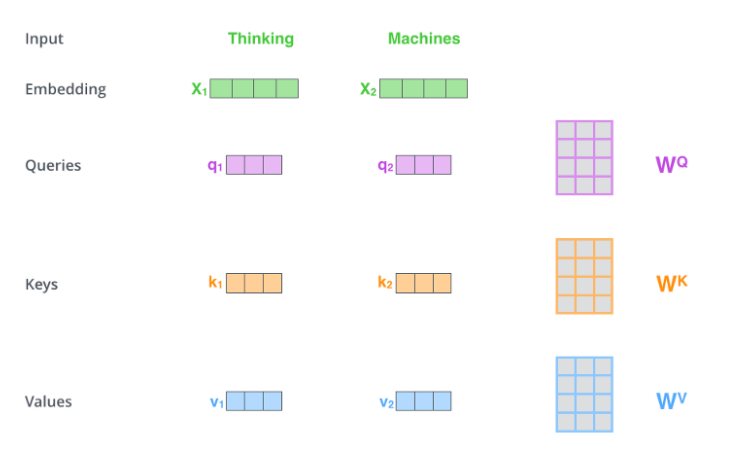
## Self-Attention

**Let’s first look at how to calculate self-attention using vectors, then proceed to look at how it’s actually implemented — using matrices.**

****

**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.**

**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.**

****

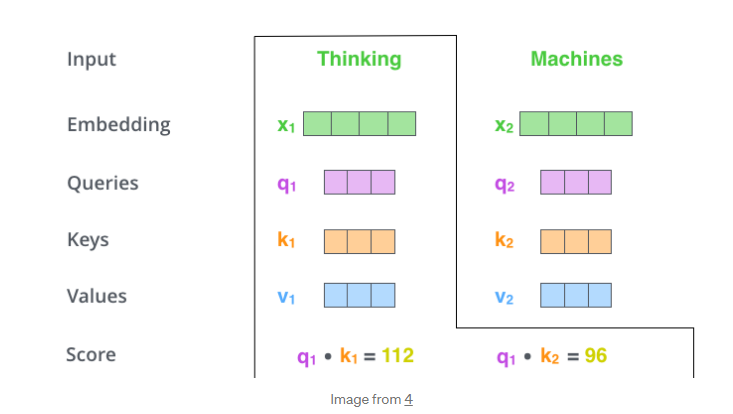
**Multiplying x1 by the WQ weight matrix produces q1, the “query” vector associated with that word. We end up creating a “query”, a “key”, and a “value” projection of each word in the input sentence.**

**What are the “query”, “key”, and “value” vectors?**

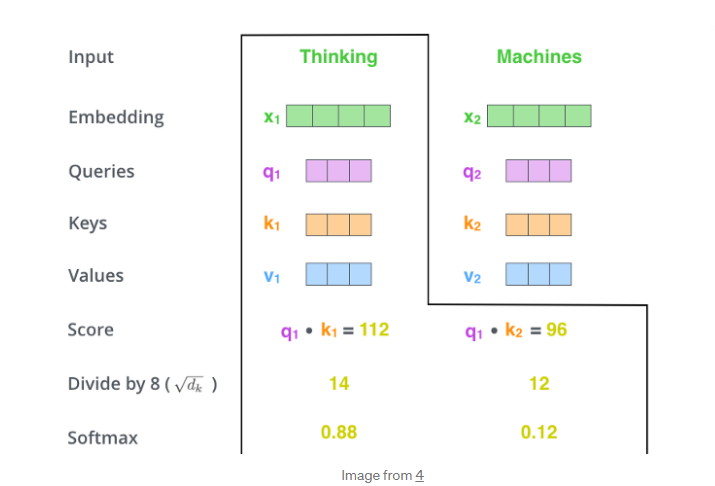
**They’re abstractions that are useful for calculating and thinking about attention. Once you proceed with reading how attention is calculated below, you’ll know pretty much all you need to know about the role each of these vectors plays.**

**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.**

****

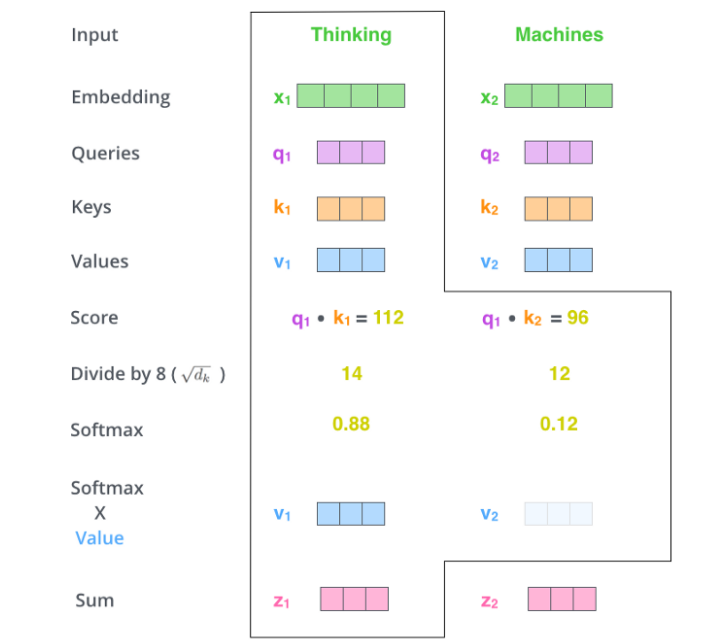
**The third and forth steps are to divide the scores by 8 (the square root of the dimension of the key vectors used in the paper — 64. This leads to having more stable gradients. There could be other possible values here, but this is the default), then pass the result through a softmax operation. Softmax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

****

**This softmax score determines how much 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.**

**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 this position (for the first word).**

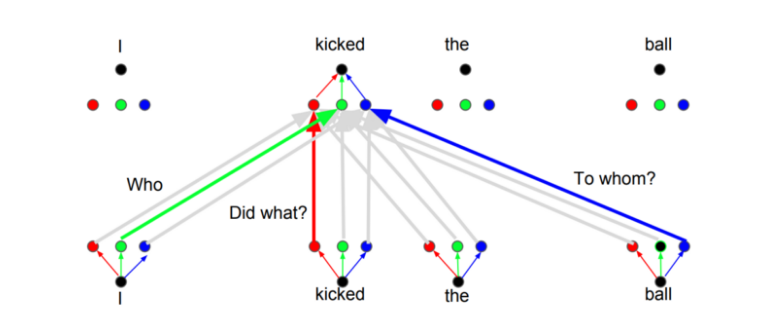
****

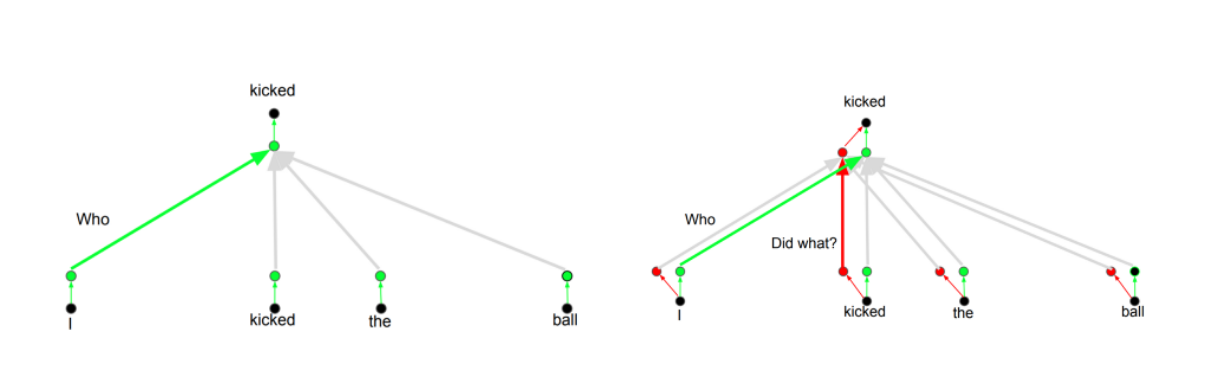
**That concludes the self-attention calculation. The resulting vector is one we can send along to the feed-forward neural network. In the actual implementation, however, this calculation is done in matrix form for faster processing. So let’s look at that now that we’ve seen the intuition of the calculation on the word level.**

## Multihead attention

**Transformers basically work like that. There are a few other details that make them work better. For example, instead of only paying attention to each other in one dimension, Transformers use the concept of Multihead attention.**

**The idea behind it is that whenever you are translating a word, you may pay different attention to each word based on the type of question that you are asking. The images below show what that means. For example, whenever you are translating “kicked” in the sentence “I kicked the ball”, you may ask “Who kicked”. Depending on the answer, the translation of the word to another language can change. Or ask other questions, like “Did what?”, etc…**

****

****

## Positional Encoding

**Another important step on the Transformer is to add positional encoding when encoding each word. Encoding the position of each word is relevant, since the position of each word is relevant to the translation.**

## Fundamental concepts of the Transformer

**This section provides some necessary background. Feel free to skip it and jump in self-attention straight on if you already feel comfortable with the concepts.**

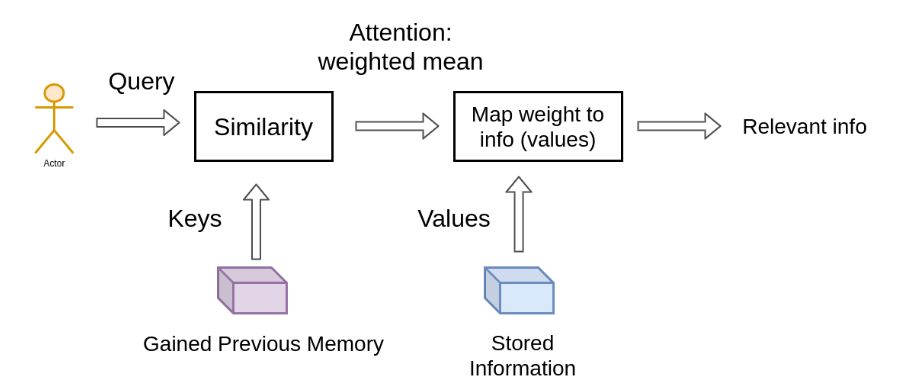
### Feature-based attention: The Key, Value, and Query

**Key-value-query concepts come from**[**information retrieval**](https://click.linksynergy.com/deeplink?id=r24KwW5qbBo&mid=40328&murl=https%3A%2F%2Fwww.coursera.org%2Fspecializations%2Fdata-mining)**systems. I found it extremely helpful to clarify these concepts first.**

**Let’s start with an example of searching for a video on youtube.**

**When you search (query) for a particular video, the search engine will map your query against a set of keys (video title, description, etc.) associated with possible stored videos. Then the algorithm will present you the best-matched videos (values). This is the foundation of content/feature-based lookup.**

**Bringing this idea closer to the transformer’s attention we have something like this:**

****

**In the single video retrieval, the attention is the choice of the video with a maximum relevance score.**

**But we can relax this idea. To this end, the main difference between attention and retrieval systems is that we introduce a more abstract and smooth notion of ‘retrieving’ an object. By defining a degree of similarity (weight) between our representations (videos for youtube) we can weight our query.**

**Instead of choosing where to look according to the position within a sequence, we now attend to the content that we wanna look at!**

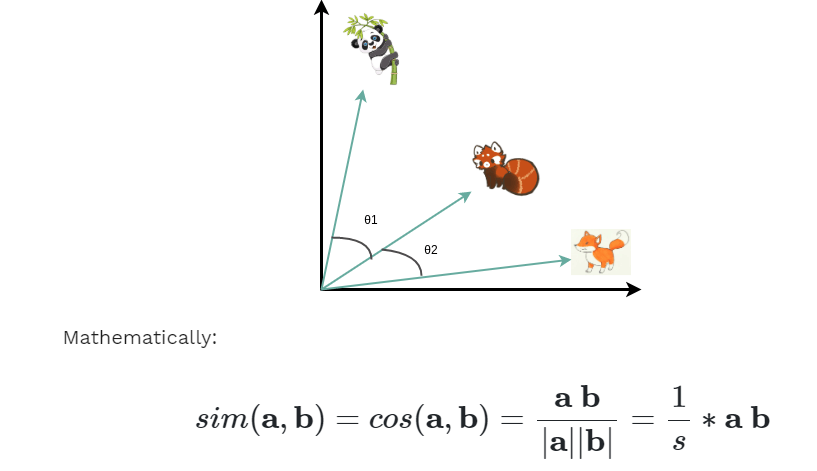
**So, by moving one step forward, we further split the data into key-value pairs.**

**We use the keys to define the attention weights to look at the data and the values as the information that we will actually get.**

**For the so-called mapping, we need to quantify similarity, that we will be seeing next.**

### Vector similarity in high dimensional spaces

**In geometry, the inner vector product is interpreted as a vector projection. One way to define vector similarity is by computing the normalized inner product. In low dimensional space, like the 2D example below, this would correspond to the cosine value.**

****

**We can associate the similarity between vectors that represent anything (i.e. animals) by calculating the scaled dot product, namely the cosine of the angle.**

**In transformers, this is the most basic operation and is handled by the self-attention layer as we’ll see.**

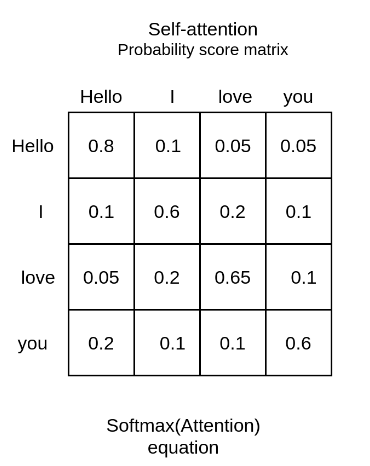
## Self-Attention: The Transformer encoder

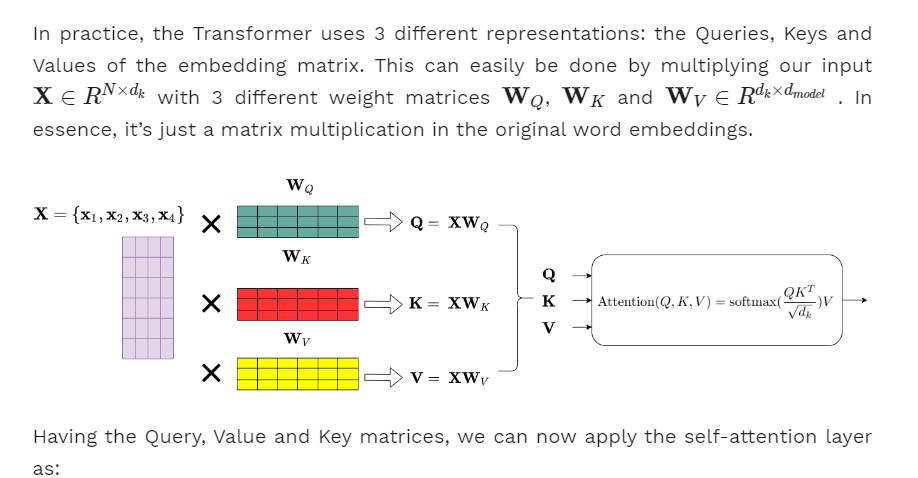
**What is self-attention?**

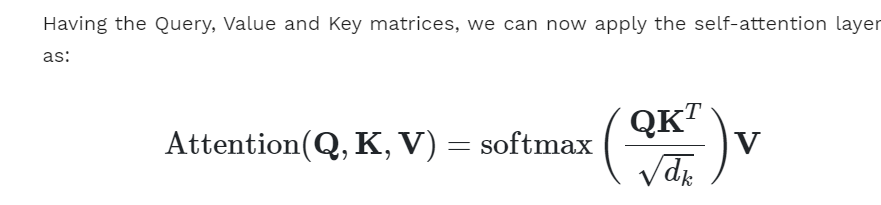
**“Self-attention, sometimes called intra-attention, is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence.” ~ Ashish Vaswani et al. [2] from Google Brain.**

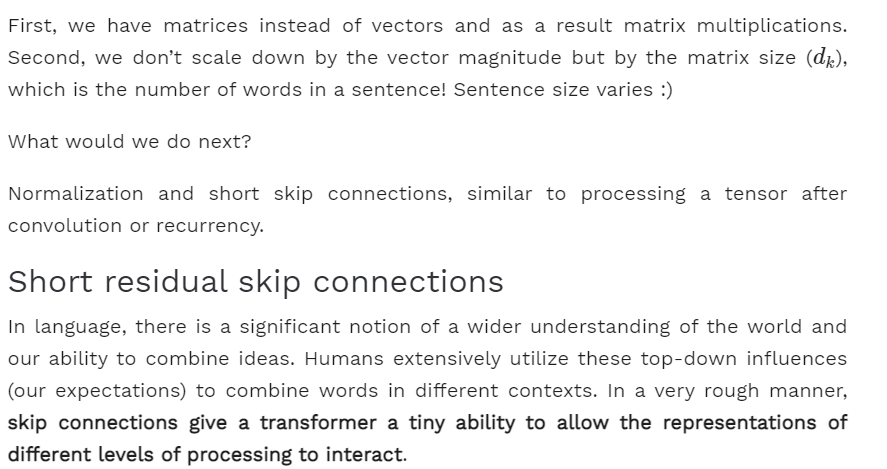
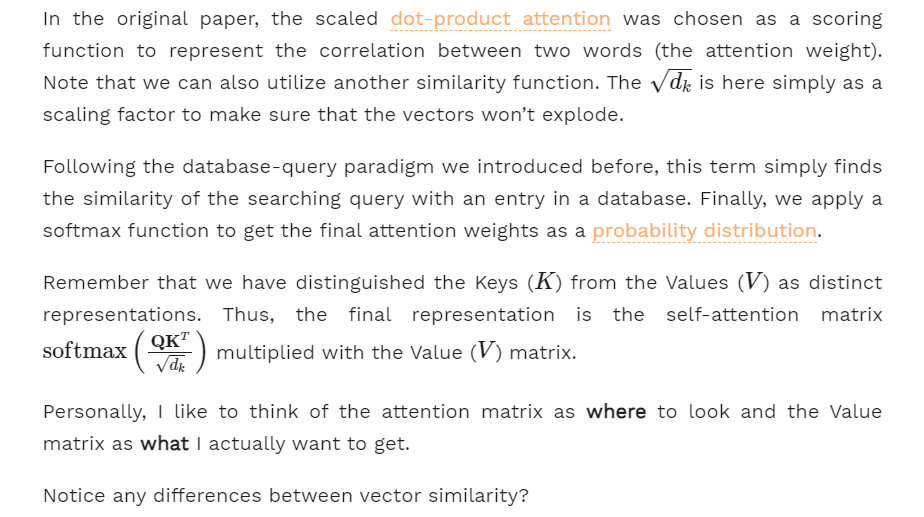
**Self-attention enables us to find correlations between different words of the input indicating the syntactic and contextual structure of the sentence.**

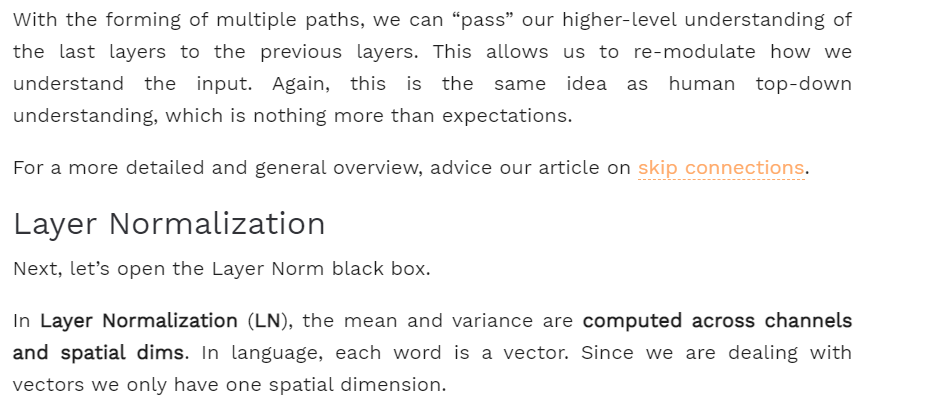
**Let’s take the input sequence “Hello I love you” for example. A trained self-attention layer will associate the word “love” with the words ‘I” and “you” with a higher weight than the word “Hello”. From linguistics, we know that these words share a subject-verb-object relationship and that’s an intuitive way to understand what self-attention will capture.**

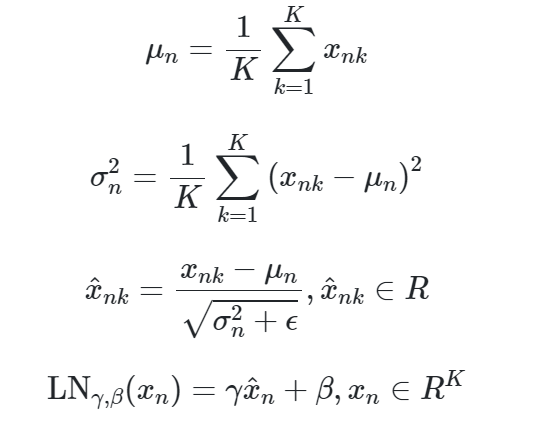
****

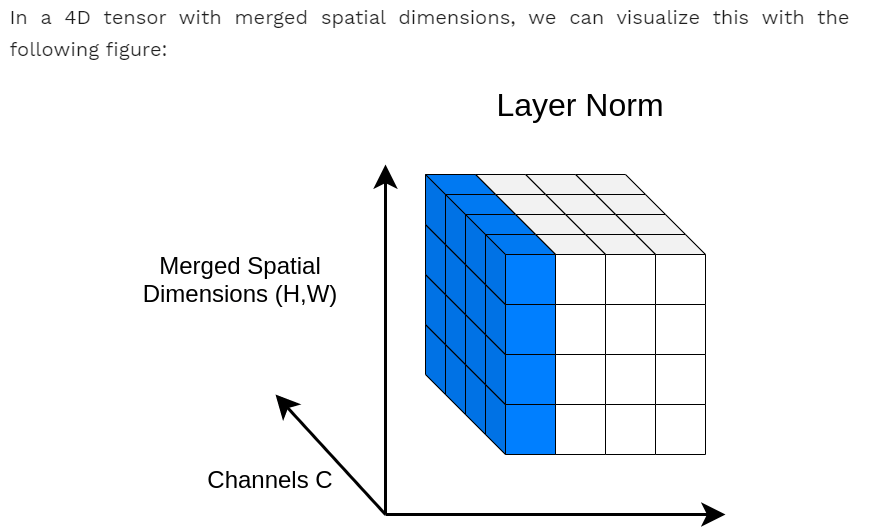
****

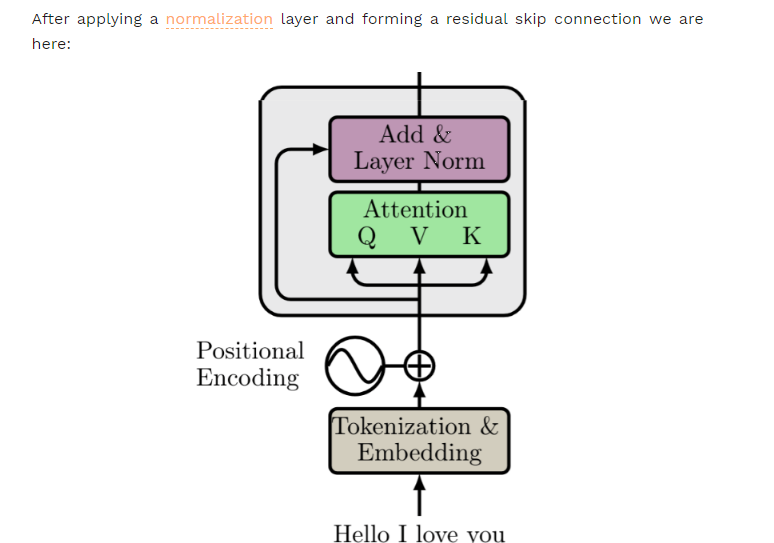
****

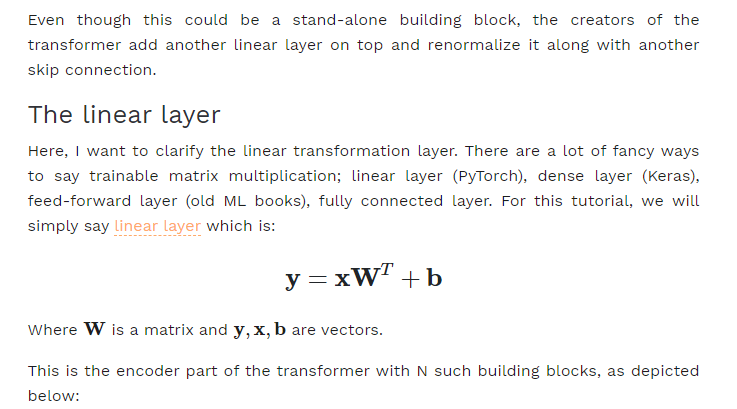
****

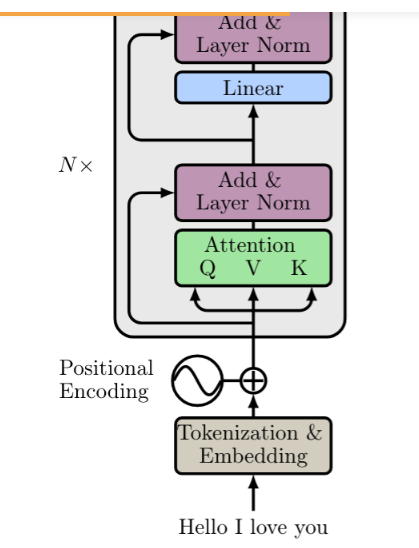
****

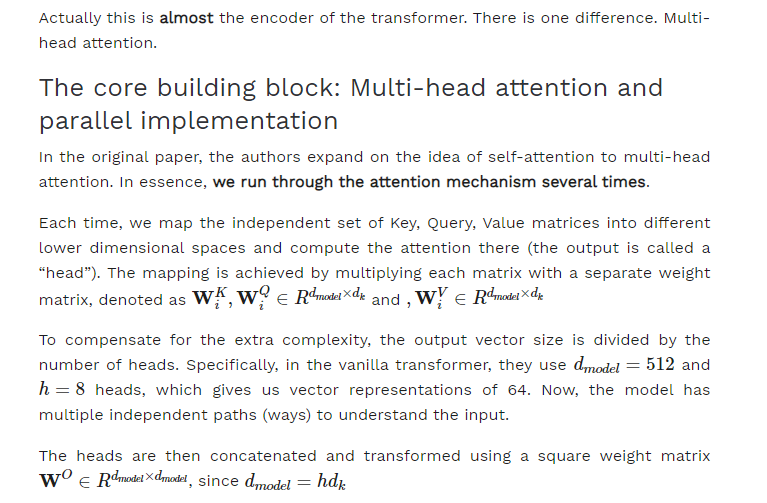
****

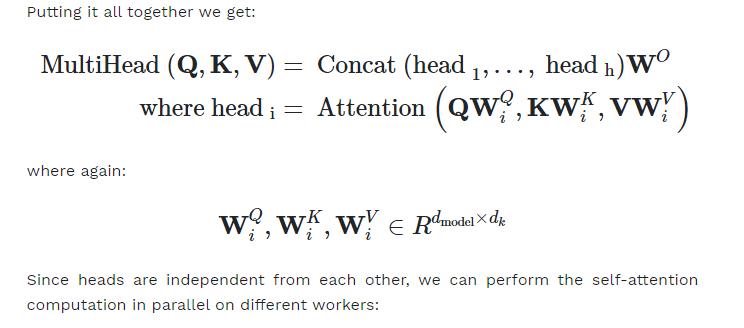
****

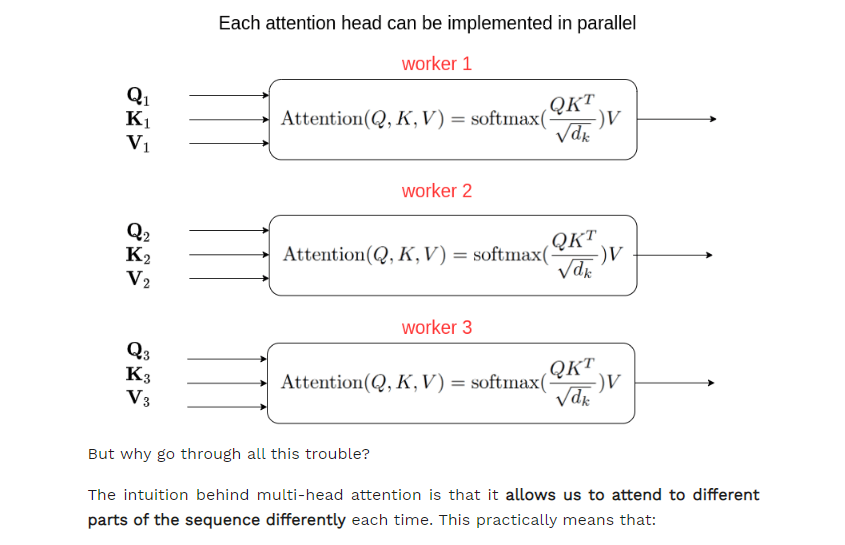
****

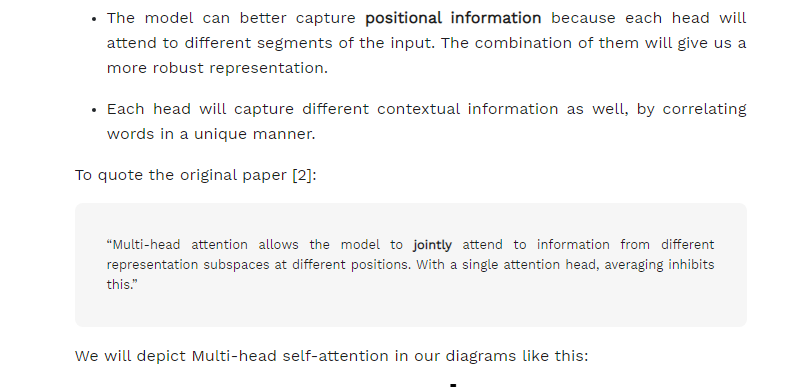
****

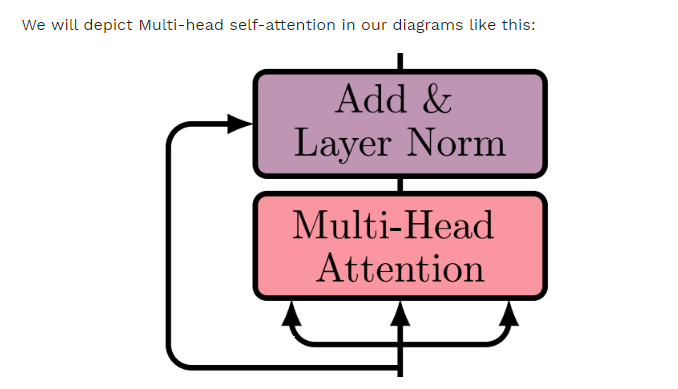
****

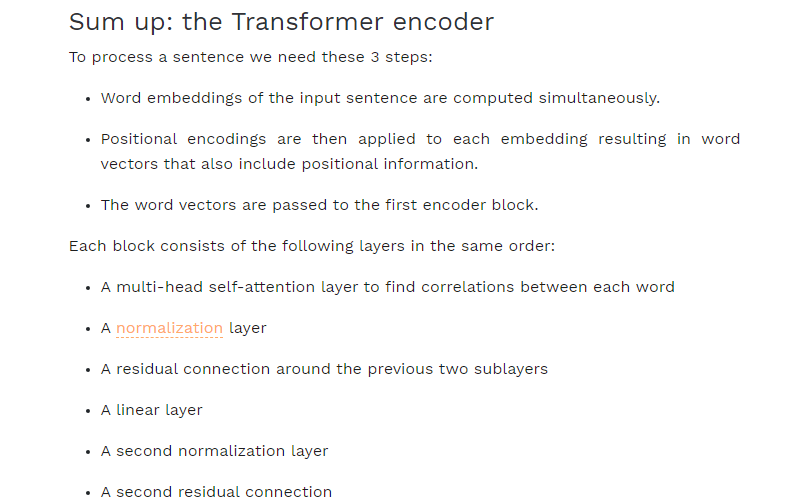
****

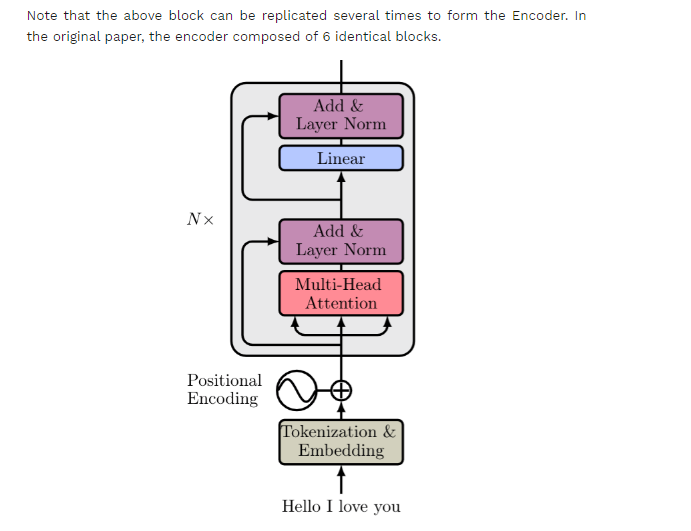
****

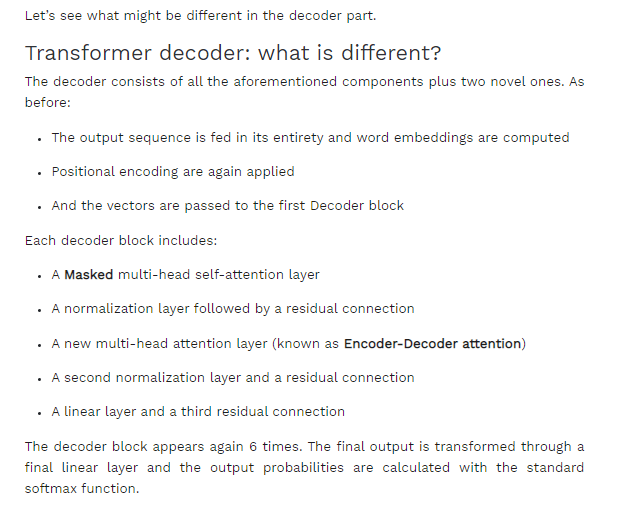
****

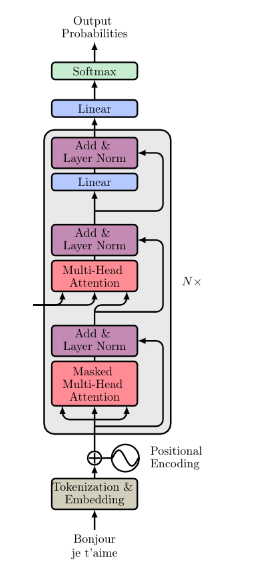
****

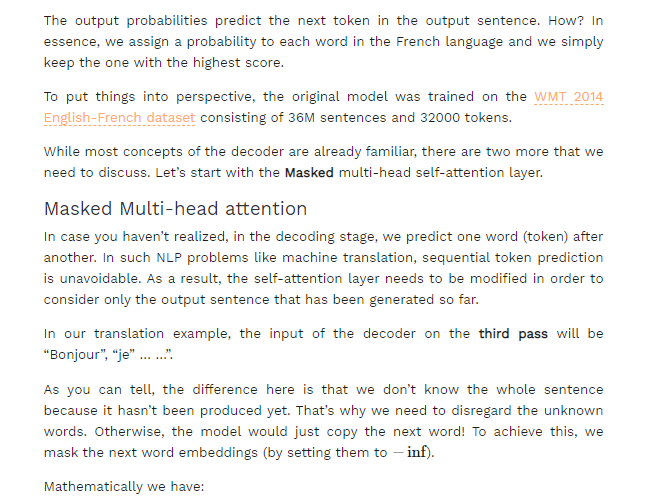
****

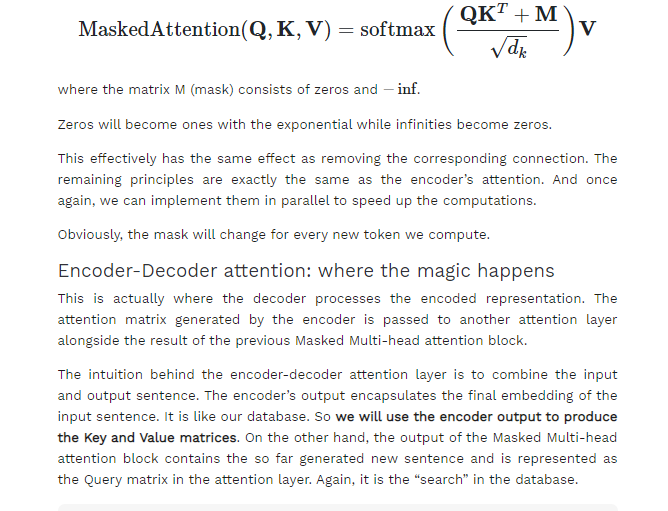
****

****

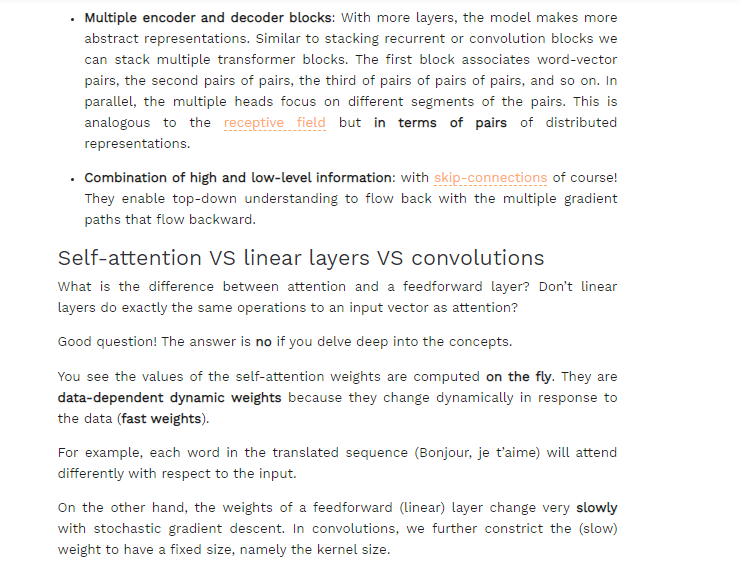
****

****

****

****

****

****

### Limitations of the Transformer

**Transformer is undoubtedly a huge improvement over the RNN based seq2seq models. But it comes with its own share of limitations:**

* **Attention can only deal with fixed-length text strings. The text has to be split into a certain number of segments or chunks before being fed into the system as input**
* **This chunking of text causes context fragmentation. For example, if a sentence is split from the middle, then a significant amount of context is lost. In other words, the text is split without respecting the sentence or any other semantic boundary**

**So how do we deal with these pretty major issues? That’s the question folks who worked with Transformer asked. And out of this came Transformer-XL.**

### Understanding Transformer-XL

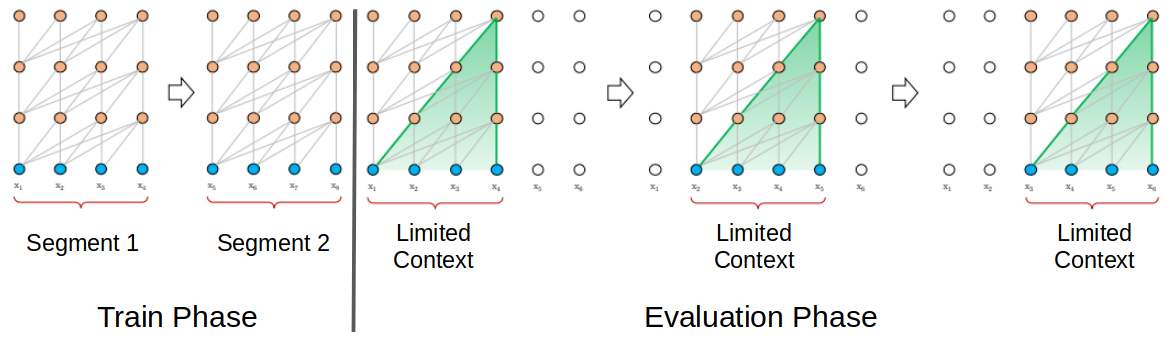
**Transformer architectures can learn longer-term dependency. However, they can’t stretch beyond a certain level due to the use of fixed-length context (input text segments). A new architecture was proposed to overcome this shortcoming in the paper –**[**Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context**](https://arxiv.org/pdf/1901.02860.pdf)**.**

**In this architecture, the hidden states obtained in previous segments are reused as a source of information for the current segment. It enables modeling longer-term dependency as the information can flow from one segment to the next.**

### Using Transformer for Language Modeling

**Think of language modeling as a process of estimating the probability of the next word given the previous words.**

[**Al-Rfou et al. (2018)**](https://arxiv.org/abs/1808.04444)**proposed the idea of applying the Transformer model for language modeling. As per the paper, the entire corpus can be split into fixed-length segments of manageable sizes. Then, we train the Transformer model on the segments independently, ignoring all contextual information from previous segments:**

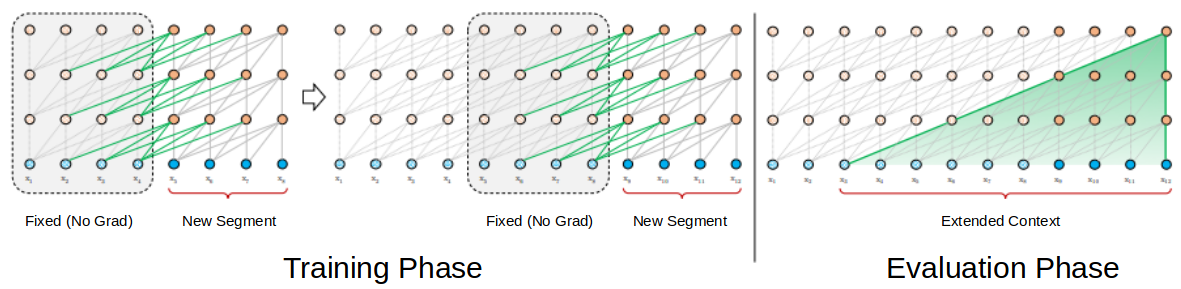
**[](https://cdn.analyticsvidhya.com/wp-content/uploads/2019/06/Screenshot-from-2019-06-18-08-20-09.png)**

**Transformer Model with a segment length of 4 (Source: https://arxiv.org/abs/1901.02860)**

**This architecture doesn’t suffer from the problem of vanishing gradients. But the context fragmentation limits its longer-term dependency learning. During the evaluation phase, the segment is shifted to the right by only one position. The new segment has to be processed entirely from scratch. This evaluation method is unfortunately quite compute-intensive.**

### Using Transformer-XL for Language Modeling

**During the training phase in Transformer-XL, the hidden state computed for the previous state is used as an additional context for the current segment. This recurrence mechanism of Transformer-XL takes care of the limitations of using a fixed-length context.**

**[](https://cdn.analyticsvidhya.com/wp-content/uploads/2019/06/Screenshot-from-2019-06-18-08-32-41.png)**

**Transformer XL Model with a segment length of 4**

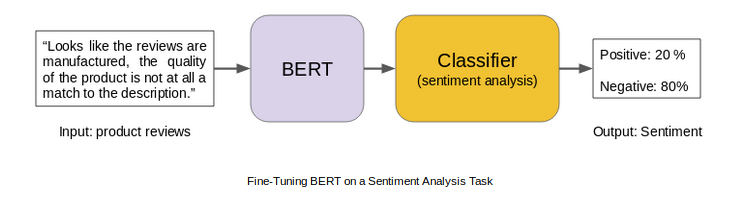
**During the evaluation phase, the representations from the previous segments can be reused instead of being computed from scratch (as is the case of the Transformer model). This, of course, increases the computation speed manifold.**

## The New Sensation in NLP: Google’s BERT (Bidirectional Encoder Representations from Transformers)

**We all know how significant transfer learning has been in the field of computer vision. For instance, a pre-trained deep learning model could be fine-tuned for a new task on the ImageNet dataset and still give decent results on a relatively small labeled dataset.**

**Language model pre-training similarly has been quite effective for improving many natural language processing tasks: (**[**https://paperswithcode.com/paper/transformer-xl-attentive-language-models**](https://openai.com/blog/language-unsupervised/)**and**[**https://paperswithcode.com/paper/transformer-xl-attentive-language-models**](https://arxiv.org/abs/1801.06146)**).**

**The BERT framework, a new language representation model from Google AI, uses pre-training and fine-tuning to create state-of-the-art models for a wide range of tasks. These tasks include question answering systems, sentiment analysis, and language inference.**

****

### BERT’s Model Architecture

**BERT uses a multi-layer bidirectional Transformer encoder. Its self-attention layer performs self-attention in both directions. Google has released two variants of the model:**

1. **BERT Base: Number of Transformers layers = 12, Total Parameters = 110M**
2. **BERT Large: Number of Transformers layers = 24, Total Parameters = 340M**

****

**BERT uses bidirectionality by pre-training on a couple of tasks — Masked Language Model and Next Sentence Prediction. Let’s discuss these two tasks in detail.**

### BERT Pre-Training Tasks

**BERT is pre-trained using the following two unsupervised prediction tasks.**

#### 1. Masked Language Modeling (MLM)

**According to the**[**paper**](https://arxiv.org/pdf/1810.04805.pdf)**:**

**“The masked language model randomly masks some of the tokens from the input, and the objective is to predict the original vocabulary id of the masked word based only on its context. Unlike left-to-right language model pre-training, the MLM objective allows the representation to fuse the left and the right context, which allows us to pre-train a deep bidirectional Transformer.”**

**The Google AI researchers masked 15% of the words in each sequence at random. The task? To predict these masked words. A caveat here – the masked words were not always replaced by the masked tokens [MASK] because the [MASK] token would never appear during fine-tuning.**

**So, the researchers used the below technique:**

* **80% of the time the words were replaced with the masked token [MASK]**
* **10% of the time the words were replaced with random words**
* **10% of the time the words were left unchanged**

#### 2. Next Sentence Prediction

#### Generally, language models do not capture the relationship between consecutive sentences. BERT was pre-trained on this task as well.

#### For language model pre-training, BERT uses pairs of sentences as its training data. The selection of sentences for each pair is quite interesting. Let’s try to understand it with the help of an example.

#### Imagine we have a text dataset of 100,000 sentences and we want to pre-train a BERT language model using this dataset. So, there will be 50,000 training examples or pairs of sentences as the training data.

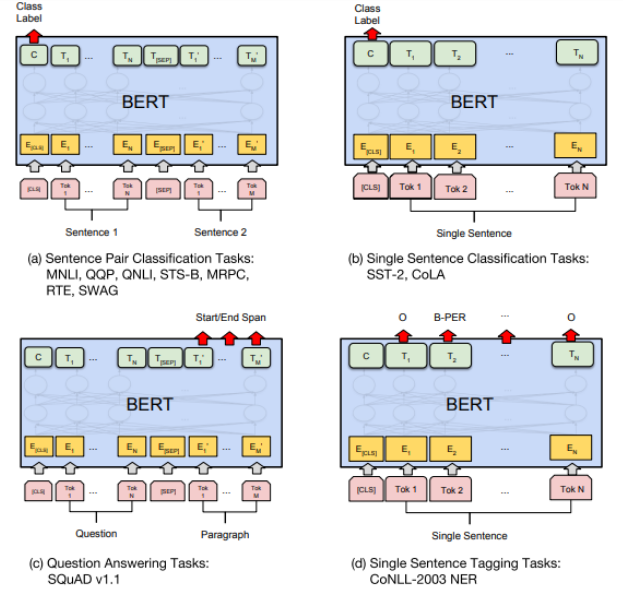
#### For 50% of the pairs, the second sentence would actually be the next sentence to the first sentence

#### For the remaining 50% of the pairs, the second sentence would be a random sentence from the corpus

#### The labels for the first case would be ‘IsNext’ and ‘NotNext’ for the second case

#### Architectures like BERT demonstrate that unsupervised learning (pre-training and fine-tuning) is going to be a key element in many language understanding systems. Low resource tasks especially can reap huge benefits from these deep bidirectional architectures.

#### Below is a snapshot of a few NLP tasks where BERT plays an important role:

**[](https://cdn.analyticsvidhya.com/wp-content/uploads/2019/06/Screenshot-from-2019-06-18-08-50-57.png)**

**Source: https://arxiv.org/abs/1810.04805**

[**https://huggingface.co/transformers/**](https://huggingface.co/transformers/)

[**https://towardsdatascience.com/transformers-141e32e69591**](https://towardsdatascience.com/transformers-141e32e69591)

[**https://theaisummer.com/transformer/**](https://theaisummer.com/transformer/)

[**https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04**](https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04)

[**https://towardsdatascience.com/transformers-89034557de14**](https://towardsdatascience.com/transformers-89034557de14)