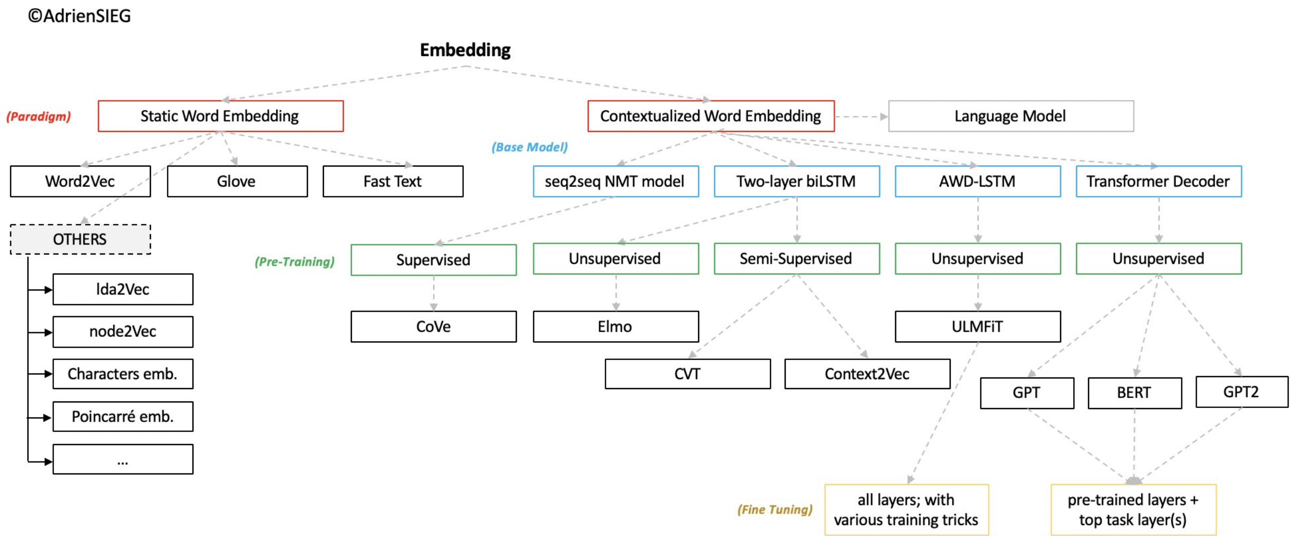
FROM Pre-trained Word Embeddings TO Pre-trained Language Models — Focus on BERT

FROM Static Word Embedding TO Dynamic (Contextualized) Word Embedding

Adrien Sieg Follow
Aug 29, 2019 · 23 min read ★

"It only seems to be a question of time until ***pretrained word embeddings*** *will be dethroned and replaced by* ***pretrained language models*** *in the toolbox of every NLP practitioner"* [*Sebastian Ruder*]



adsieg.github.io

Static Word Embedding

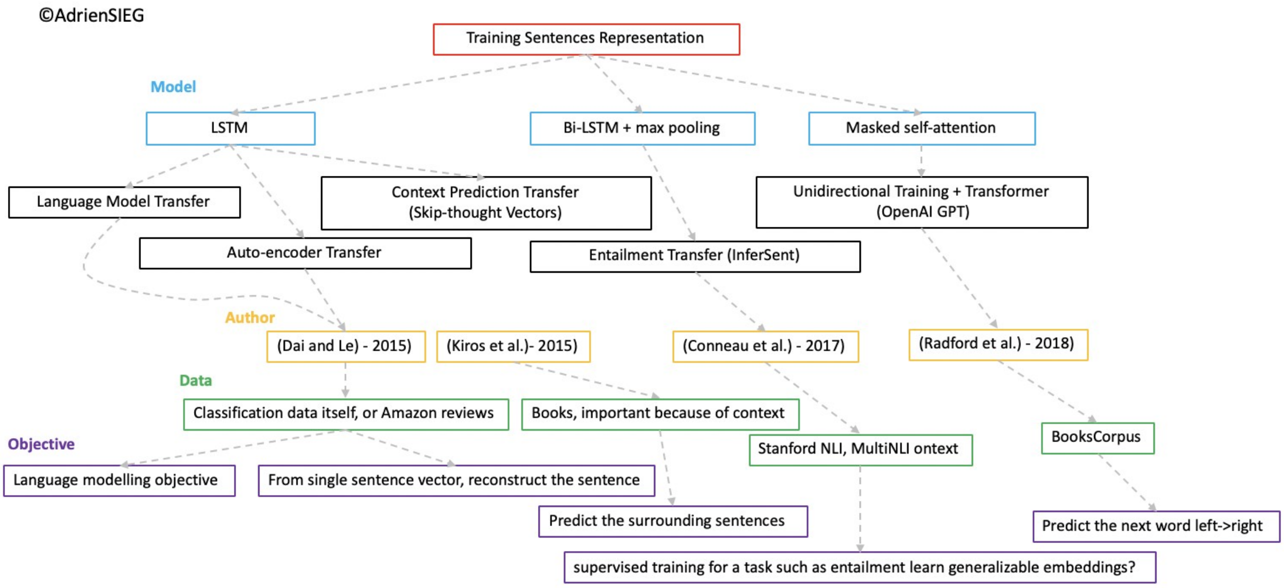
- Skip-Gram & CBOW (aka **Word2Vec**)

- **Glove**
- **fastText**
- ***Exotic***: Lda2Vec, Node2Vec, Characters Embeddings, CNN embeddings, …
- **Poincaré Embeddings** to learn hierarchical representation

Contextualized (Dynamic) Word Embedding (LM)

- **CoVe** (*Contextualized Word-Embeddings*)
- **CVT** (*Cross-View Training*)
- **ELMO** (*Embeddings from Language Models*)
- **ULMFiT** (*Universal Language Model Fine-tuning*)
- **BERT** (*Bidirectional Encoder Representations from Transformers*)
- **GPT** & **GPT-2** (*Generative Pre-Training*)
- **Transformer XL** (meaning extra long)
- **XLNet** (*Generalized Autoregressive Pre-training*)
- **ENRIE** *(Enhanced Representation through kNowledge IntEgration)*
- **(FlairEmbeddings** (*Contextual String Embeddings for Sequence Labelling*))

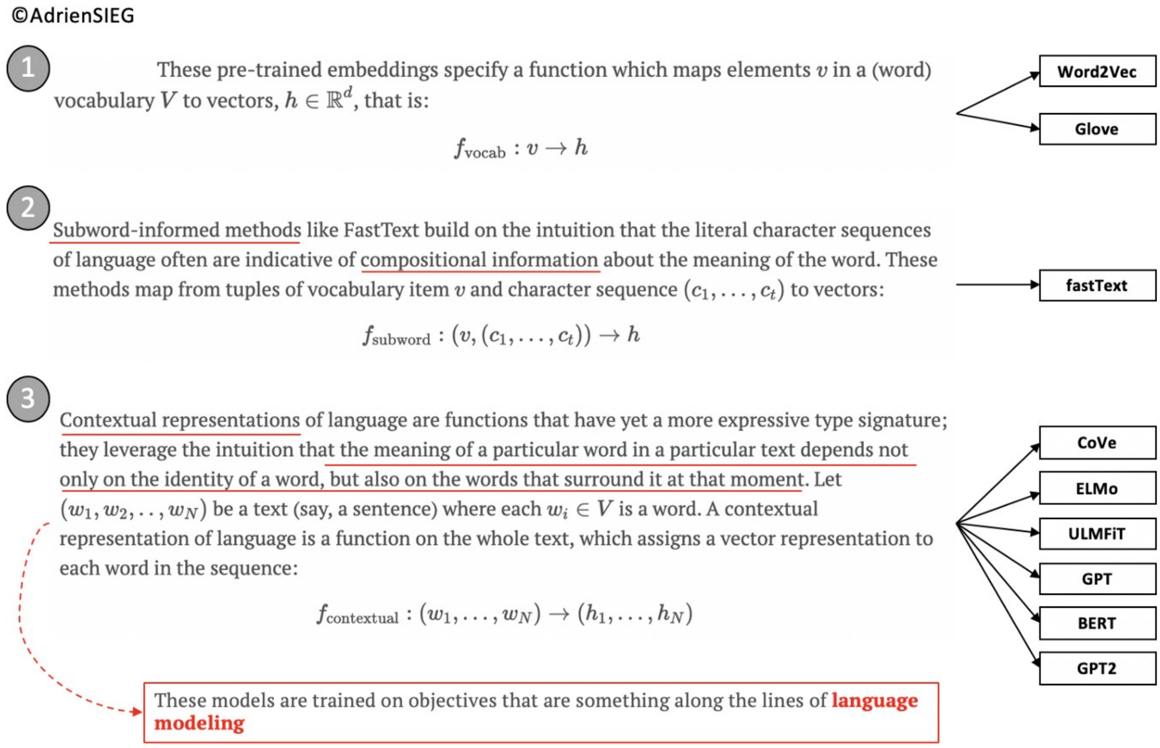
and many more else…



- **Semantic** approach *(relating to meaning in language -> stretch the words beyond, understanding ambiguities)*

- **Pragmatic** approach *(relating proximity between words and documents)*

What are the main families of models?



https://nlp.stanford.edu//~johnhew//structural-probe.html?utm_source=quora&utm_medium=referral#the-structural-probe

Language modeling is the task of assigning a probability distribution over sequences of words that matches the distribution of a language. Although it sounds formidable, **language modeling** *(i.e. ELMo, BERT, GPT)* is essentially just predicting words in a blank. More formally, given a context, a language model predicts the probability of a word occurring in that context.

Why is this method effective? Because this method forces the model to learn how to use information from the entire sentence in deducing what words are missing.

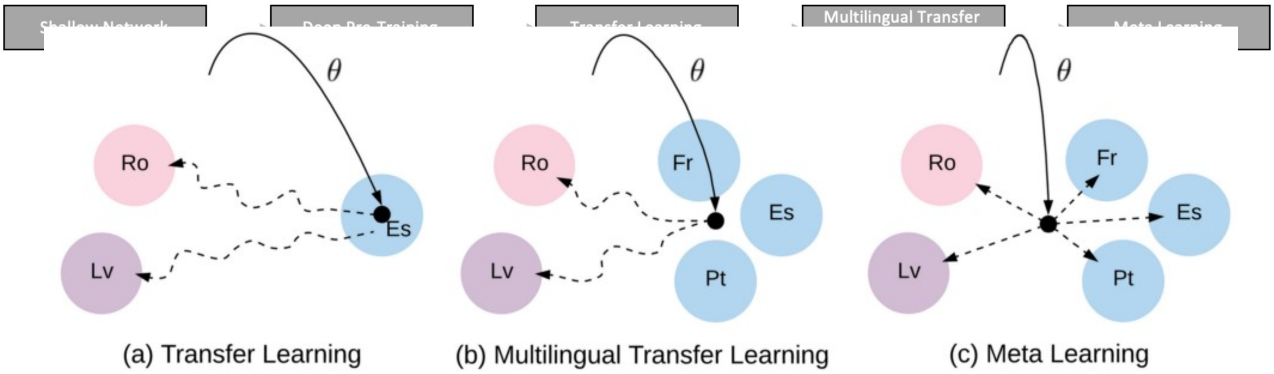
0. Static vs. Dynamic

- **Static Word Embeddings** fail to **capture polysemy**. They generate the **same embedding** for the **same word** in **different contexts**. ### Contextualized words embeddings aim at **capturing word semantics** in **different contexts** to address the **issue of polysemous** and the **context-dependent nature of words**.

- **Static Word Embeddings could only leverage off the vector outputs from unsupervised models for downstream tasks** — not the unsupervised models themselves.They were mostly **shallow models** to begin with and were often discarded after training (e.g. word2vec, Glove) ### The output of **Contextualized (Dynamic) Word Embedding** training is **the trained model and vectors** — not just vectors.

- Traditional word vectors are **shallow representations** (a single layer of weights, known as embeddings). They only **incorporate previous knowledge in the first layer of the model**. The rest of the network still needs to be trained from scratch for a new target task. They **fail to capture higher-level information** that might be even more useful. Word embeddings are useful in **only capturing semantic meanings of words** but we also need to understand higher level concepts like **anaphora**, **long-term dependencies**, agreement, negation, and many more.

Evolutions:



http://ruder.io/10-exciting-ideas-of-2018-in-nlp/

Transfer learning — a technique where instead of training a model from scratch, we use **models pre-trained on a large dataset** and then **fine-tune them for specific natural language tasks**.

Some particularities :

- **ULMFiT** → Transfer by **Fine Tuning**
- **ELMo** → Transfer by **Features Extraction**
- **BERT** → Transfer by **Attention Extraction**

Why using Transfer Learning?

In vision, it has been in practice for some time now, with people using models trained to learn features from the huge ImageNet dataset, and then training it further on smaller data for different tasks.

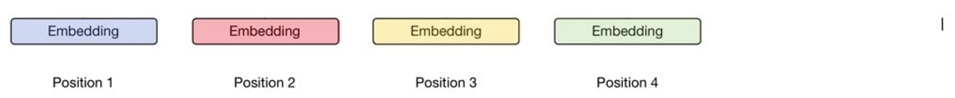
- Most **datasets for text classification** (or any other supervised NLP tasks) are rather **small**. This makes it very difficult to train deep neural networks, as they would **tend to overfit on these small training data** and **not generalize well in practice**.

In computer vision, for a couple of years now, the trend is to pre-train any model on the huge ImageNet corpus. This is much better than a random initialization *because the model learns general image features and that learning can then be used in any vision task (say captioning, or detection).*

In NLP, we trained on **a general language modeling (LM) task** and then **fine tuned** on text classification (or other task). This would, in principle, perform well because the model would be able to **use its knowledge of the semantics of language acquired from the generative pre-training**.

- it is able to **capture long-term dependencies** in language
- it effectively **incorporates hierarchical relations**
- it can help the model **learn sentiments**
- **large data corpus** is easily available for LM

BERT



https://mostafadehghani.com/2019/05/05/universal-transformers/

Improvements?

1. Differences between **GPT** vs. **ELMo** vs. **BERT ->** all pre-training model architectures. **BERT** uses a bidirectional Transformer *vs.* **GPT** uses a left-to-right Transformer *vs.* **ELMo** uses the concatenation of independently trained left-to-right and right-to-left LSTM to generate features for downstream task. **BERT** representations are jointly conditioned on both left and right context in all layers. In other words, it is deeply bidirectional, as opposed to **ELMo** (*shallow bidirectional*) and **OpenAI GPT** (*one direction, left to right*).

2. Transformers demonstrate that **recurrence** and **convolution** are not essential for building high-performance natural language models

3. They achieve state-of-the-art machine translation results using **a self-attention operation**

4. Attention is a **highly-efficient operation** due to its **parallelizability** and **runtime** characteristics

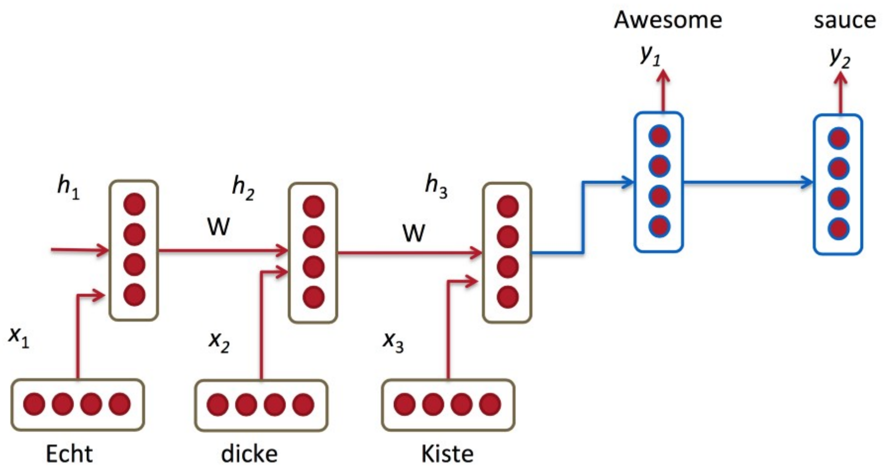
5. Traditional language models take the previous n tokens and predict the next one. In contrast, BERT trains a language model that takes **both the previous and next tokens** into account when predicting- really **bidirectional**.

6. If you simply ask a deep neural network to learn what typical English sentences look like by reading all of Wikipedia, what does it learn about the English language? **BERT encode human-like parse trees and find tree structures in these vector spaces** when computer have represented each word in the sentence as a real-valued vector, with no explicit representation of the parse tree. BERT has the ability to reconstruct parse trees from the Penn Treebank.

https://nlp.stanford.edu//~johnhew//structural-probe.html?utm_source=quora&utm_medium=referral#the-structural-probe

- Scoring the importance of **each layer for a specific NLP task** *(e.g. POS, NER etc.)*, **shows basic syntactic information is captured earlier** *(lower layers)* in the network followed by **semantic information in higher layers**. This reflected in the image on the right. *(This observation is similar to what was seen in ELMo model too)*.
- Also, information related to **syntactic tasks** seems to be **more localized in few layers**, where information for **semantic tasks** (SPR and Relations) is generally **spread across the entire network**
- Examining the output word vectors show not only are different senses of a word captured in distinct representations they are also spatially separated in a fine grained manner.

https://arxiv.org/pdf/1906.02715.pdf

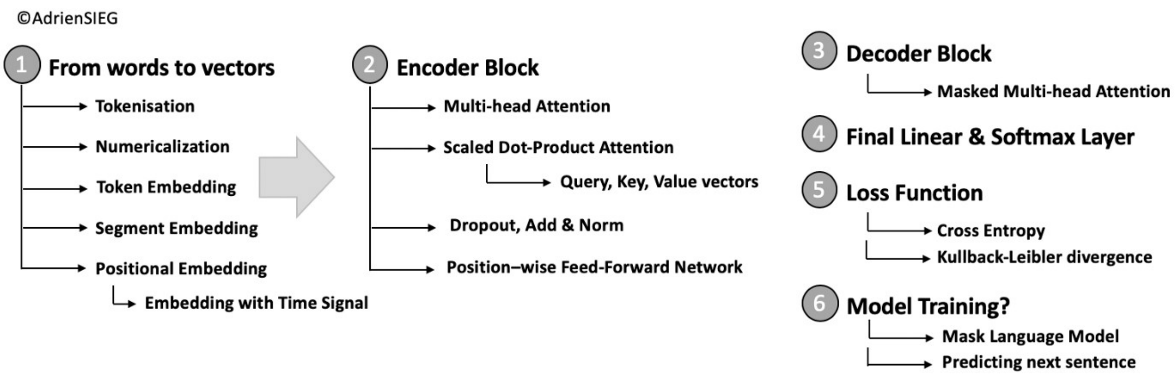


7. **Distance each word has to travel**: In a simple RNN, the word 'Echt' has to travel multiple steps. The last red layer has to store the encoded information. In large sentences which are over 50 words long, **the amount of distance each word has to travel increases linearly**. And since we keep writing over that encoded information, we are sure to loose important words that come early in the sentence. After encoding it also h as to travel to get to it's decoded destination.

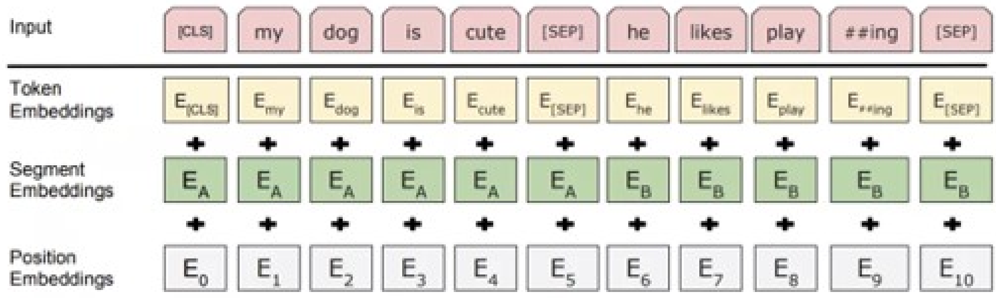
With an **attention mechanism** we no longer try encode the full source sentence **into a fixed-length vector**. Rather, we allow the decoder to "attend" to different parts of the source sentence at each step of the output generation.

The significant achievement of attention mechanism was to improve **spatial understanding of the model.**

Job?



1. →From words to vectors



- **Tokenization** is the task of chopping it up into pieces, called *tokens* , perhaps at the same time throwing away certain characters, such as punctuation.

- Using **wordpieces** (e.g. playing -> play + ##ing) instead of words. This is effective in reducing the size of the vocabulary and increases the amount of data that is available for each word.

- **Numericalization** aims at mapping each token to a unique integer in the corpus' vocabulary.

- **Token embedding** is the task of get the embedding (i.e. a vector of real numbers) for each word in the sequence. Each word of the sequence is mapped to a *emb_dim dimensional vector* that the model will learn during training. You can think about it as a vector look-up for each token. The elements of those vectors are treated as model parameters and are optimized with back-propagation just like any other weights.

- **Padding** was used to make the input sequences in a *batch* have the same length. That is, we increase the length of some of the sequences by adding '*<pad>*' tokens.

- **Positional encoding** :

Recall that the **positional encoding** is designed to help the model learn some notion of **sequences and relative positioning of tokens**. This is crucial for language-based tasks especially here because we are not making use of any traditional recurrent units such as *RNN*, *GRU* or *LSTM*

Intuitively, we aim to be able to **modify the represented meaning of a specific word depending on its position**. We don't want to change the full representation of the word but we want to **modify it a little to encode its position** by adding **numbers between *[-1,1]*** using predetermined (non-learned) sinusoidal functions to the token embeddings. For the rest of the **Encoder,** the word will be **represented slightly differently depending on the position the word is in** (even if it is the same word).

Encoder must be able to use the fact that **some words are in a given position** while, in the same sequence, other words are in other specific positions. That is, we want the network to able to **understand relative positions** and **not only absolute ones**.

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

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

Here the i denotes the vector index we are looking at, pos denotes the token, and d_{model} denotes a fixed constant representing the dimension of the input embeddings. Ok let's break it down further.

https://www.mihaileric.com/posts/transformers-attention-in-disguise/

The sinuosidal functions chosen by the authors allow positions to be represented as **linear combinations of each other** and thus allow the network to **learn relative relationships between the token positions**.

Let's consider an example:

Positional embeddings could be understood as the **distance between different words in the sequence**. The intuition **here is that adding** these values to the embeddings provides **meaningful distances between the embedding vectors once they're projected into Q/K/V vectors and during dot-product attention**.

https://mc.ai/seq2seq-pay-attention-to-self-attention-part-2/

2. →Encoder Blocks

https://jalammar.github.io/illustrated-transformer/

A total of N encoder blocks are chained together to generate the **Encoder's** output.

Note: *In **BERT's** experiments, the number of blocks N (or L, as they call it) was chosen to be 12 and 24.*

- The dimensions of the **input** and **output** of the encoder block are the same. Hence, it makes sense to **use the output of one encoder block as the input of the next encoder block**.

- A specific block is in charge of *finding relationships between the input representations and encode them* in its output.
- The blocks do not share weights with each other.
- This iterative process through the blocks will help the neural network capture more complex relationships between words in the input sequence.
- The **Transformer** uses ***Multi-Head Attention***, which means it computes attention ***h*** **different times** with **different weight matrices** and then concatenates the results together.

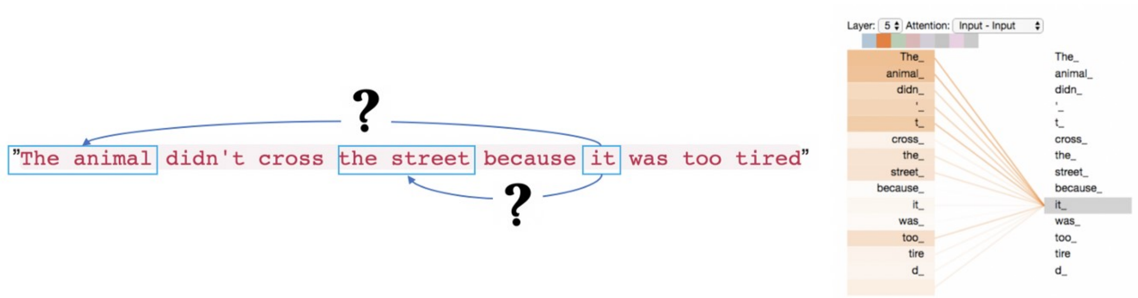
Attention Mechanism

Let's dive into attention mechanism. Note that Multi Head Self Attention is different between **ENCODER block** and **DECODER block.**

https://mc.ai/seq2seq-pay-attention-to-self-attention-part-2/

A — One Head Self-Attention

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



https://jalammar.github.io/illustrated-transformer/

What does "*it*" in this sentence refer to? Is it referring to *the street* or to *the animal*?

- Create **three vectors** *from* each of the **encoder's input vectors**
- 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.

Why ***dimensionality is 64?*** *As we must have :*

-> Output's dimension *is [length of input sequences] x [dimension of embeddings — 512]*

-> We use ***8 heads*** *during Multi-head Self-Attention process. The* ***output size of a given self attention vector*** *is [length of input sequences] x [64]. So* ***the concatenated vector resulting from all Multi-head Self-Attention*** *process would be [length of input sequences] x ([64] x [8]) = [length of input sequences] x ([512])*

How Self-Attention works?

Query q: the query vector *q* encodes the word/position on the left that is paying attention, i.e. the one that is "querying" the other words. In the example above, the query vector for "the" (the selected word) is highlighted.

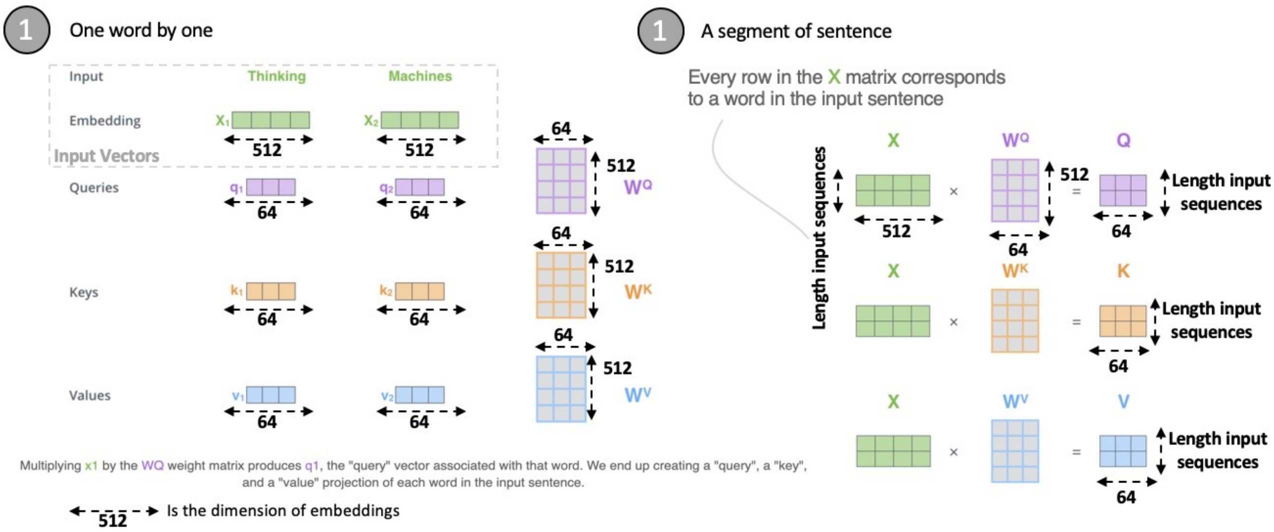
Key k: the key vector *k* encodes the word on the right to which attention is being paid. The key vector together with the query vector determine the attention score between the respective words, as described below.

q×k (element-wise): the element-wise product of the query vector and a key vector. This product is computed between the selected query vector and each of the key vectors. This is a precursor to the dot product (the sum of the element-wise product) and is included for visualization purposes because it shows how individual elements in the query and key vectors contribute to the dot product.

q·k: the dot product of the selected query vector and each of the keyvectors. This is the unnormalized attention score.

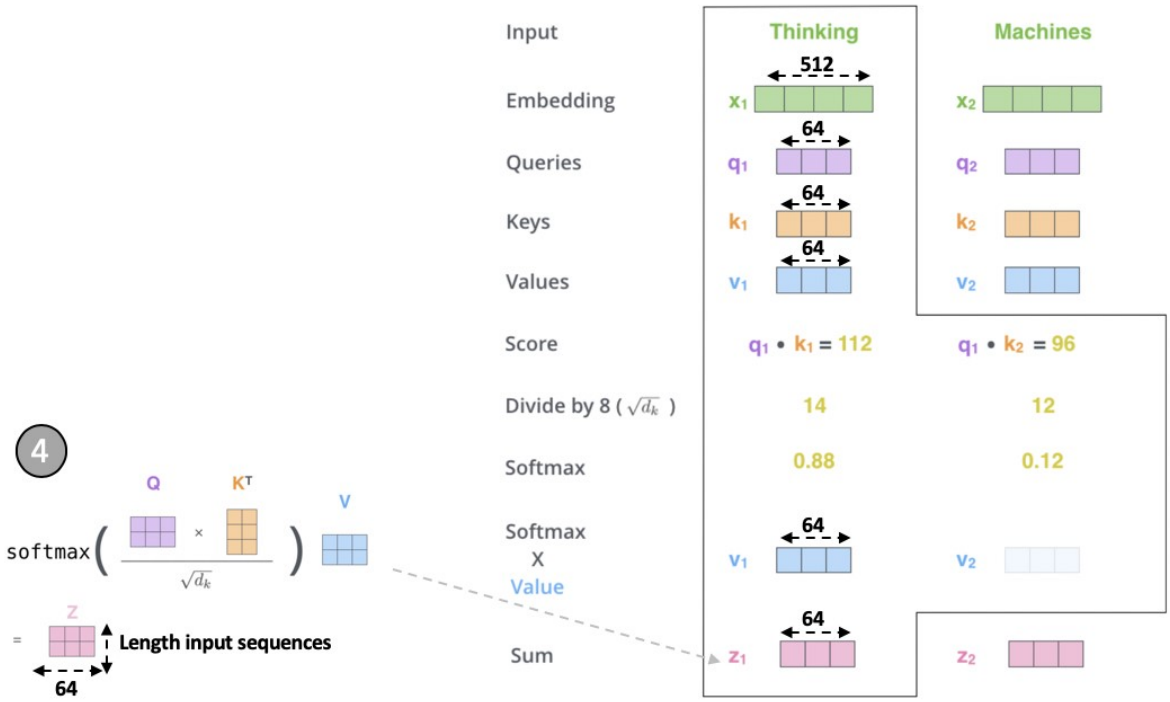
Softmax: the softmax of $q \cdot k$ / 8 across all target words. This normalizes the attention scores to be positive and sum to one. The constant factor 8 is the square root of the vector length (64). This softmax score determines how much each word will be expressed at this position. Clearly the word 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.

How to calculate these three vectors?



We calculate self-attention for every word in the input sequence.

Focus on "Score"



through K **measures the similarity between those vectors**. If we call ***v_i*** and ***u_j*** the projections of the ***i-th* token** and the ***j-th* token** through Q and K respectively, their dot product can be seen as:

This is a measure of **how similar are the directions of *u_i* and *v_j*** and **how large are their lengths** (the closest the direction and the larger the length, the greater the dot product).

Another way of thinking about this matrix product is as the encoding of a **specific relationship between each of the tokens in the input sequence** (the relationship is defined by the matrices K, Q).

https://lesdieuxducode.com/blog/2019/4/bert--le-transformer-model-qui-sentraine-et-qui-represente

https://towardsdatascience.com/deconstructing-bert-part-2-visualizing-the-inner-workings-of-attention-60a16d86b5c1

We see that the **product of the query vector** for "*the*" and **the key vector** for "*store*" (the next word) is strongly positive across most neurons. For tokens other than the next token, the **key-query product contains some combination of positive and negative values**. The result is a high attention score between "the" and "store".

Example: 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**

Produit scalaire

Le produit scalaire entre deux vecteurs correspond à la longueur du

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:

Word	q vector	k vector	v vector	score
Action	q_1	k_1	v_1	$q_1 \cdot k_1$
gets		k_2	v_2	$q_1 \cdot k_2$
results		k_3	v_3	$q_1 \cdot k_3$

2 Then, these scores are divided by 8 which is the square root of the dimension of the key vector:

Word	q vector	k vector	v vector	score	score / 8
Action	q_1	k_1	v_1	$q_1 \cdot k_1$	$q_1 \cdot k_1 / 8$
gets		k_2	v_2	$q_1 \cdot k_2$	$q_1 \cdot k_2 / 8$
results		k_3	v_3	$q_1 \cdot k_3$	$q_1 \cdot k_3 / 8$

3 Next, these scores are normalized using the softmax activation function:

Word	q vector	k vector	v vector	score	score / 8	Softmax
Action	q_1	k_1	v_1	$q_1 \cdot k_1$	$q_1 \cdot k_1 / 8$	x_{11}
gets		k_2	v_2	$q_1 \cdot k_2$	$q_1 \cdot k_2 / 8$	x_{12}
results		k_3	v_3	$q_1 \cdot k_3$	$q_1 \cdot k_3 / 8$	x_{13}

4 These normalized scores are then multiplied by the value vectors (v1, v2, v3) and sum up the resultant vectors to arrive at the final vector (z1). This is the output of the self-attention layer. It is then passed on to the feed-forward network as input:

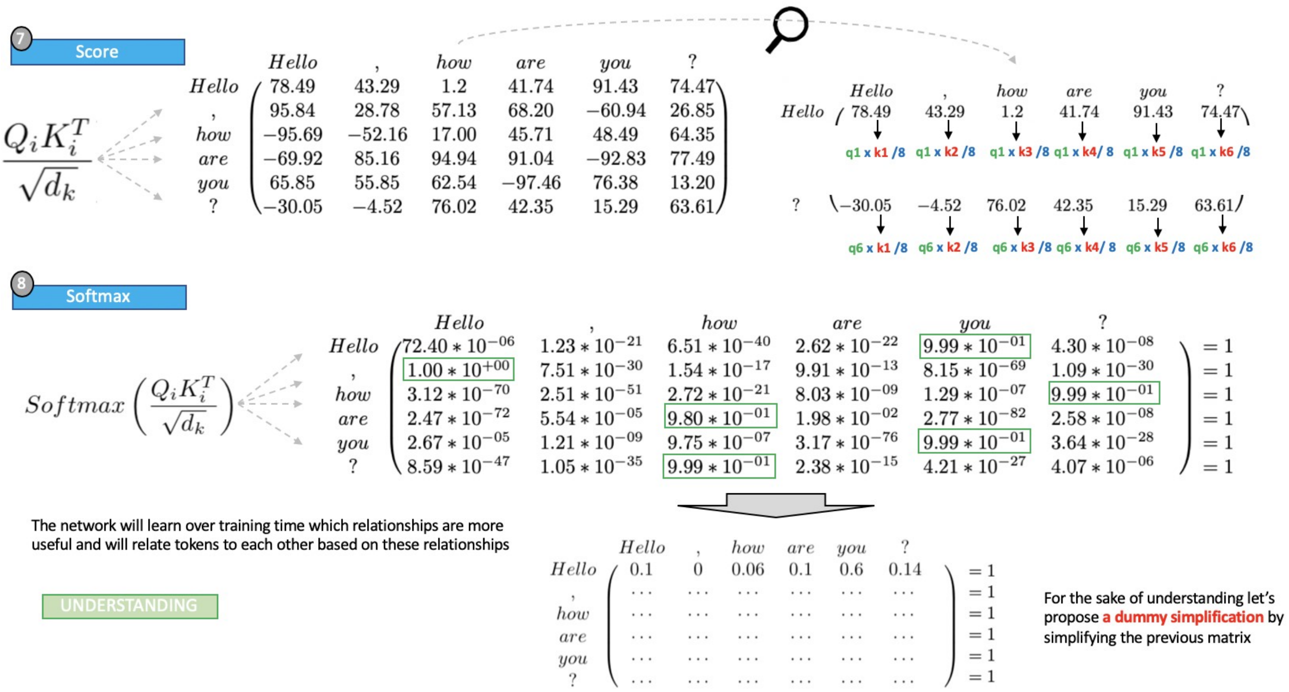
Word	q vector	k vector	v vector	score	score / 8	Softmax	Softmax * v	Sum
Action	q_1	k_1	v_1	$q_1 \cdot k_1$	$q_1 \cdot k_1 / 8$	x_{11}	$x_{11} * v_1$	z_1
gets		k_2	v_2	$q_1 \cdot k_2$	$q_1 \cdot k_2 / 8$	x_{12}	$x_{12} * v_2$	
results		k_3	v_3	$q_1 \cdot k_3$	$q_1 \cdot k_3 / 8$	x_{13}	$x_{13} * v_3$	

https://www.analyticsvidhya.com/blog/2019/06/understanding-transformers-nlp-state-of-the-art-models/

So, z1 is the self-attention vector for the first word of the input sequence “Action gets results”. We can get the vectors for the rest of the words in the input sequence in the same fashion:

Word	q vector	k vector	v vector	score	score / 8	Softmax	Softmax * v	Sum[#]
Action		k_1	v_1	$q_2 . k_1$	$q_2 . k_1 / 8$	x_{21}	$x_{21} * v_1$	
gets	q_2	k_2	v_2	$q_2 . k_2$	$q_2 . k_2 / 8$	x_{22}	$x_{22} * v_2$	z_2
results		k_3	v_3	$q_2 . k_3$	$q_2 . k_3 / 8$	x_{23}	$x_{23} * v_3$	

Word	q vector	k vector	v vector	score	score / 8	Softmax	Softmax * v	Sum[#]
Action		k	v	q . k	q . k / 8	x	x * v	

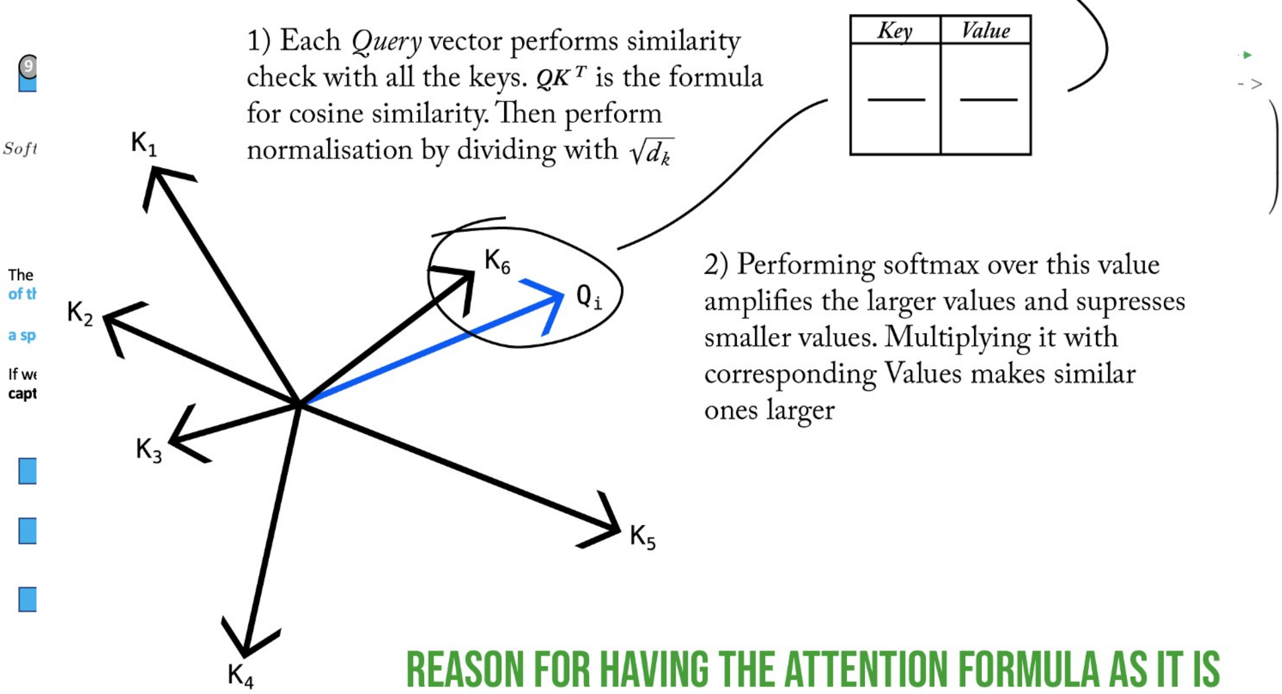


To sum up when it comes to One-Head Self Attention

The main idea behind attention is **lookup-table**, a table that has a large number of values for some other values and **you ask it a query** and **it returns one closest to it**. In the method used here, we feed it three values, **key**, **value**

Lets think back to our original sequence of embeddings, $\mathbf{x}$. We can choose that Q, K, and V are all equal to $\mathbf{x}$. In the paper, this is called “Encoder self-attention” (one of three ways attention is used). Lets say our input tokens are “The man walked the dog”. To compute the new vectors for each

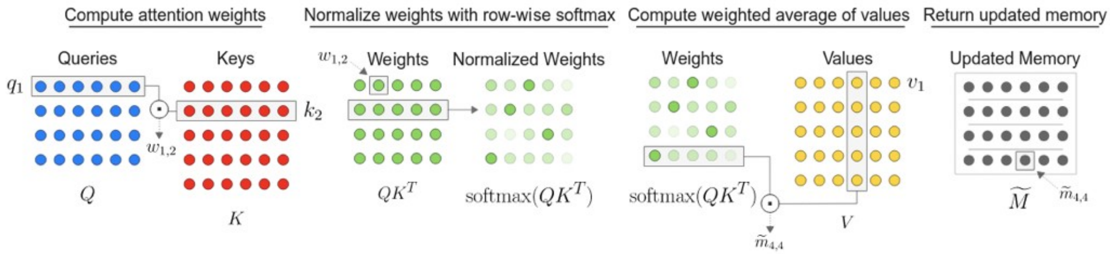
$$\boxed{Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V}$$



REASON FOR HAVING THE ATTENTION FORMULA AS IT IS

https://medium.com/datadriveninvestor/lets-build-attention-is-all-you-need-1-2-de377cebe22

An attention function can be described as a **dictionary object**.

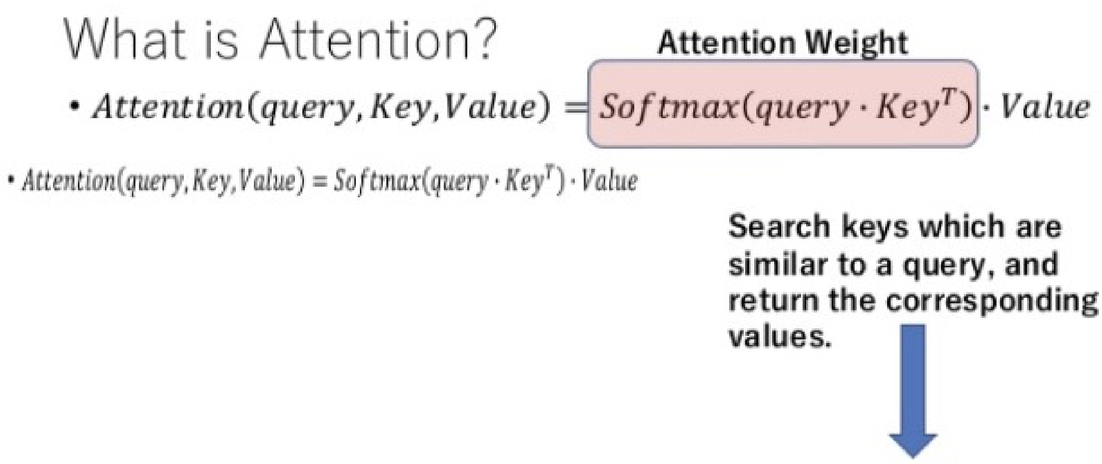


https://persagen.com/resources/biokdd-review-nlu.html

B — Multi-head Self-Attention

The paper notes that "**additive attention**" performs better than the self attention described above, though it is much slower. Additive attention uses a more complicated compatibility function — namely a feed forward neural network.

Self-attention is computed not once but multiple times in the Transformer's architecture, in parallel and independently. It is therefore referred to as **Multi-head Attention**. The outputs are concatenated and linearly transformed as shown in the figure below:



An attention function can be described as a **dictionary** object.

The Transformer uses **eight attention heads**, so we end up with **eight sets for each encoder/decoder**. Each set is used to **project the input embeddings into a different representation subspace.** If we do the same self-attention calculation we described just above, we end up with eight different Z matrices.

However, **the feed-forward layer is not expecting eight matrices**. We need to concatenate them and condense these eight down into a single matrix by multiply them with an additional weights matrix WO

How to return 8 matrices Z1…Z8 into a singe matrix Z in Multi-head attention?

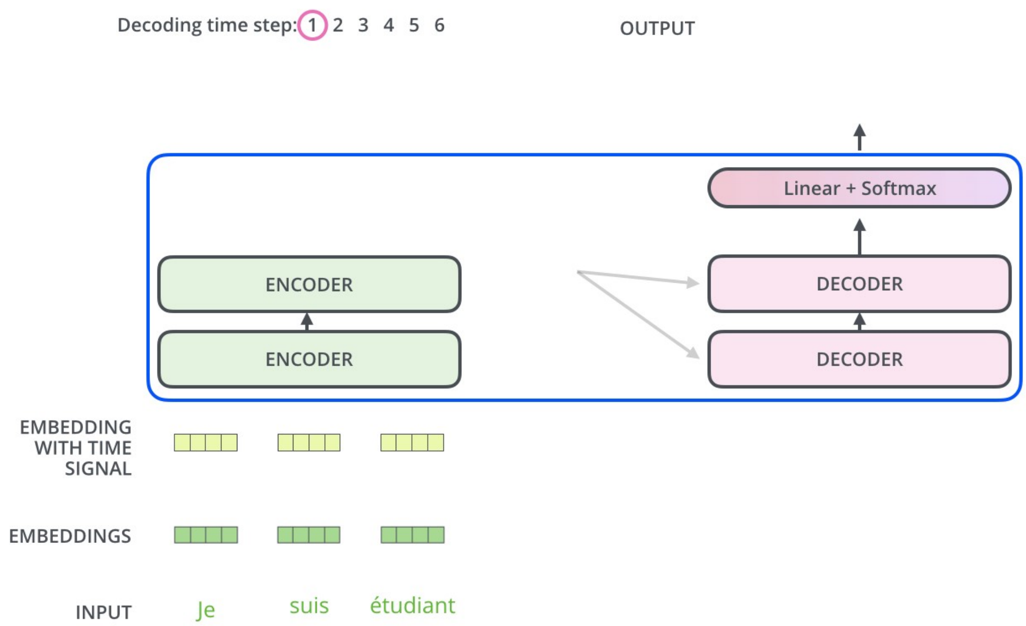
Enfin on concatène la sortie de chaque tête, et on multiplie par une matrice W0, de dimensions 512 x 512 ([(nombre de têtes) x (dimension requête ou clé ou valeur, i.e. 64)]x[dimension des embeddings]), qui apprend à projeter le résultat sur un espace de sortie aux dimensions attendues.

https://lesdieuxducode.com/blog/2019/4/bert--le-transformer-model-qui-sentraine-et-qui-represente

To sum up…

https://jalammar.github.io/illustrated-transformer/

Here is the result :



https://jalammar.github.io/illustrated-transformer/

Dropout, Add & Norm

https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1184/lectures/lecture12.pdf

Before this layer, there is always a layer for which inputs and outputs have the same dimensions (*Multi-Head Attention* or *Feed-Forward*). We will call that layer *Sublayer* and its input *x.*

After each *Sublayer*, dropout is applied with 10% probability. Call this result *Dropout(Sublayer(x))*. This result is added to the *Sublayer*'s input *x,* and we get *x + Dropout(Sublayer(x))*.

$$LayerNorm(x + Dropout(Sublayer(x)))$$

Observe that in the context of a *Multi-Head Attention* layer, this means **adding the original representation of a token *x* to the representation based on the relationship with other tokens**. It is like telling the token:

> "Learn the relationship with the rest of the tokens, but don't forget what we already learned about yourself!"

Finally, a token-wise/row-wise normalization is computed with the mean and standard deviation of each row. This improves the stability of the network.

https://jalammar.github.io/illustrated-transformer/

Layernorm changes input to have mean 0 and variance 1,
per layer and per training point (and adds two more parameters)

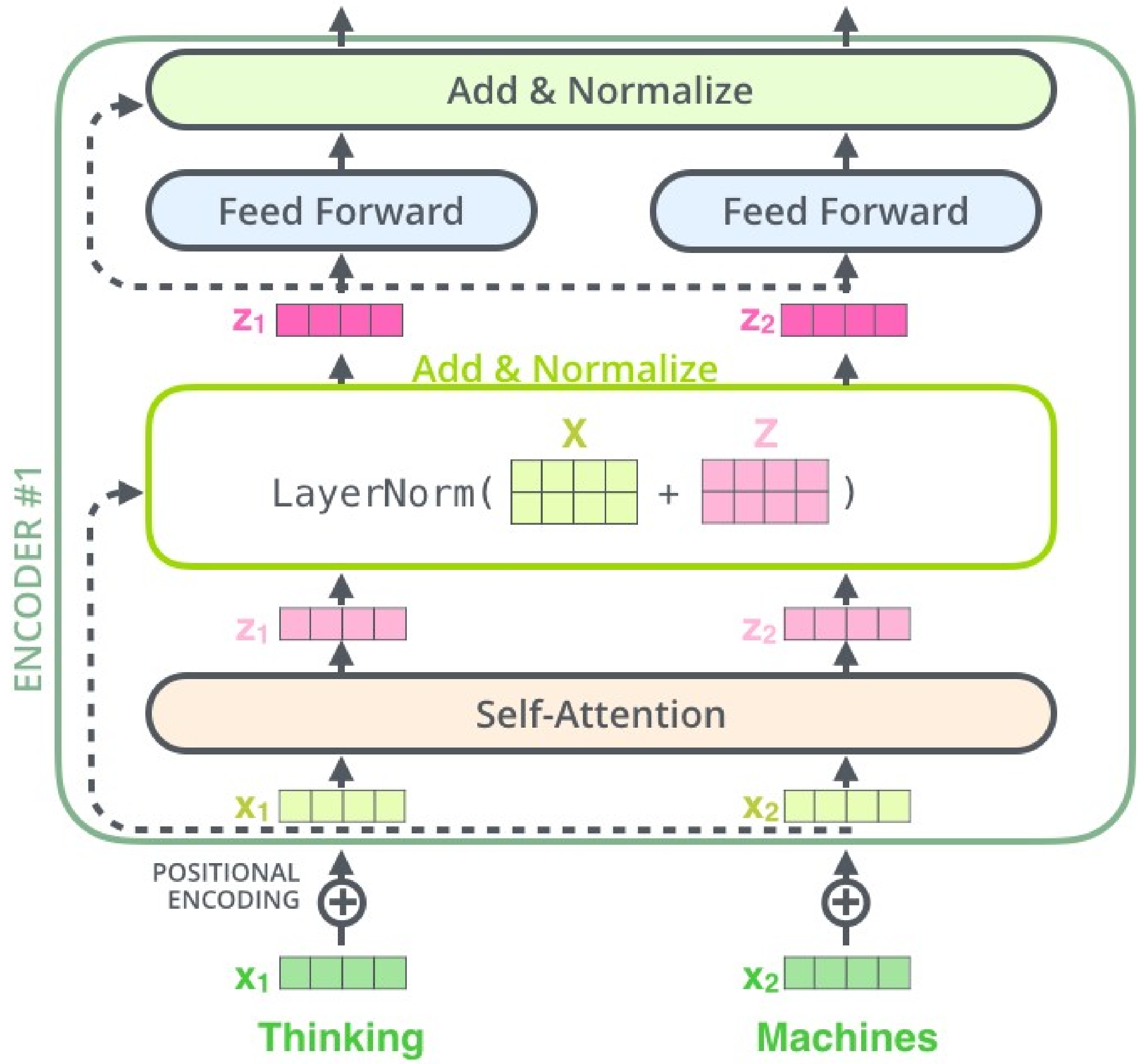
$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2} \qquad h_i = f\left(\frac{g_i}{\sigma_i}\left(a_i - \mu_i\right) + b_i\right)$$

https://arxiv.org/pdf/1607.06450.pdf

We compute **the mean and variance used for normalization** from all of the summed inputs to the neurons in a layer on a single training case.

Position-wise Feed-Forward Network

In addition to attention sub-layers, each of the layers in our encoder and decoder contains **a fully connected feed-forward network**, which is applied to **each position separately and identically**. This consists of two linear transformations with a ReLU activation in between.



des embeddings]. Source ici.

3.→Decoder Block

Each **decoder layer** consists of sublayers:

1. Masked multi-head attention (with **look ahead mask** and **padding mask**)

2. Multi-head attention (with padding mask). V (value) and K (key) receive the *encoder output* as inputs. Q (query) receives the *output from the masked multi-head attention sublayer.*

3. Point wise feed forward networks

Each of these sublayers has a residual connection around it followed by a layer normalization. The output of each sublayer is `LayerNorm(x + Sublayer(x))`.

There are N decoder layers in the transformer.

As **Q** receives the **output from decoder's first attention block**, and **K** receives the **encoder output**, **the attention weights represent the importance given to the decoder's input based on the encoder's output**. In other words, **the decoder predicts the next word by looking at the encoder output and self-attending to its own output**.

The `Decoder` consists of:

1. Output Embedding

2. Positional Encoding

3. N decoder layers

The target is put through an embedding which is summed with the positional encoding. The output of this summation is the input to the decoder layers. The output of the decoder is the input to the final linear layer.

What are the inputs of Transformer?

We feed it both the input and output sentences at the same time. The outputs initially can be filled with anything, the model ignores whatever you fill into that. It uses the entire input sentence and output sentence to predict the next word in a single go. Once we predict the word, we replace that in output sequence, and model only considers output till that point and ignores what is ahead of it. We continue to do that till we have a complete sentence.

Multi Head Masked Self Attention

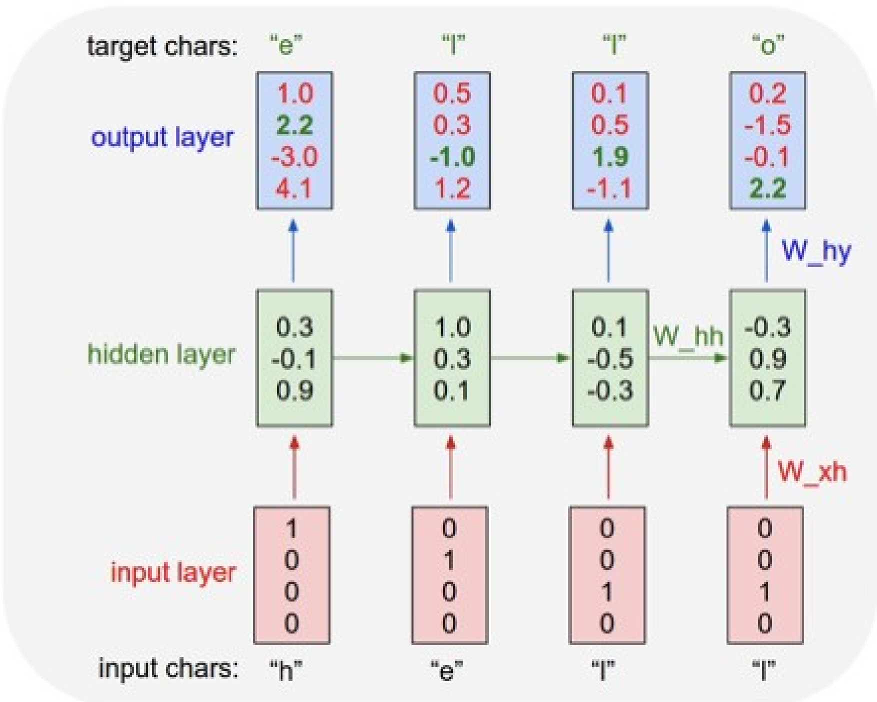
In *encoder*, self-attention layers process input **queries**, **keys** and **values** that comes from the **output of previous layer**. **Each position in encoder can attend to all positions from previous layer of the encoder**.

In *decoder*, self-attention layer enable **each position to attend to all previous positions in the decoder**, including the current position.

https://persagen.com/resources/biokdd-review-nlu.html

To prevent positions from attending to subsequent position (http://www.peterbloem.nl/blog/transformers)

In other words, the self-attention layer is only allowed to attend to **earlier positions in the output sequence**. Masking multi-head attention is done by masking future positions (setting them to -∞) before the softmax step in the self-attention calculation. **This step ensures that the predictions for position *i* can depend only on the known outputs at positions less than *i*.** Since we want these elements to be zero after the softmax, we set them to $-\infty$.



With RNNs — there is no issue like that, since **they cannot look forward into the input sequence: output i depends only on inputs 0 to i**. With a transformer, the output depends on the entire input sequence, so prediction of the next words/characters becomes vacuously easy, just retrieve it from the input.

To use self-attention as an autoregressive model, we'll need to ensure that it **cannot look forward into the sequenc**e. We do this by **applying a mask to the matrix of dot products**, before the softmax is applied. **This mask disables all elements above the diagonal of the matrix**.

After we've **handicapped the self-attention module** like this, the model **can no longer look forward in the sequence**.

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

Traduction

Français *"tu t'appelles lebowski lebowski"*

vers

Anglais *"your name's lebowski lebowski"*

Au moment de générer un nouveau mot, la matrice générée par $\mathrm{softmax}(\frac{QK^T}{\sqrt{d_k}})$ dans la couche d'auto-attention du décodeur devrait, sans masquage, ressembler à ça :

$$
\begin{array}{r c}
 & \begin{array}{ccccc} \text{Your} & \text{name} & \text{'s} & \text{lebowski} & \text{lebowski} \end{array} \\
\begin{array}{r} \text{Your} \\ \text{name} \\ \text{'s} \\ \text{lebowski} \\ \text{lebowski} \end{array} &
\begin{pmatrix}
-26{,}845 & -13{,}4225 & 34{,}749 & -10{,}0113 & -2{,}00227 \\
0{,}093 & -0{,}00693 & 0{,}006189 & -0{,}0062 & 0{,}003095 \\
3{,}435 & -495{,}767 & 3{,}74 & -53{,}4913 & -17282 \\
11{,}964 & -0{,}02413 & 8{,}7227 & -0{,}00061 & 3{,}5108 \\
-1{,}526 & 63{,}23468 & 2{,}950 & 55{,}15364 & 1{,}579
\end{pmatrix}
\end{array}
$$

Or on veut obtenir de quoi masquer à chaque mot prévu les mots qu'il devrait prévoir, **donc on veut que les valeurs correspondantes aux mots à venir n'aient aucune attention**

Il faut donc que soit multipliée avec V une matrice de ce type : $\mathrm{softmax}(\frac{QK^T}{\sqrt{d_k}})V$

$$
\begin{array}{r c}
 & \begin{array}{ccccc} \text{Your} & \text{name} & \text{'s} & \text{lebowski} & \text{lebowski} \end{array} \\
\begin{array}{r} \text{Your} \\ \text{name} \\ \text{'s} \\ \text{lebowski} \\ \text{lebowski} \end{array} &
\begin{pmatrix}
-26{,}845 & 0 & 0 & 0 & 0 \\
0{,}093 & -0{,}00693 & 0 & 0 & 0 \\
3{,}435 & -495{,}767 & 3{,}74 & 0 & 0 \\
11{,}964 & -0{,}02413 & 8{,}7227 & -0{,}00061 & 0 \\
-1{,}526 & 63{,}23468 & 2{,}950 & 55{,}15364 & 1{,}579
\end{pmatrix}
\end{array}
$$

Et pour ce faire on forcera QK^T à avoir les valeurs au dessus de la diagonale de la matrice à moins l'infini (ou chiffre très négatif)

$$
\begin{array}{r c}
 & \begin{array}{ccccc} \text{Your} & \text{name} & \text{'s} & \text{lebowski} & \text{lebowski} \end{array} \\
\begin{array}{r} \text{Your} \\ \text{name} \\ \text{'s} \\ \text{lebowski} \\ \text{lebowski} \end{array} &
\begin{pmatrix}
-26{,}845 & -\infty & -\infty & -\infty & -\infty \\
0{,}093 & -0{,}00693 & -\infty & -\infty & -\infty \\
3{,}435 & -495{,}767 & 3{,}74 & -\infty & -\infty \\
11{,}964 & -0{,}02413 & 8{,}7227 & -0{,}00061 & -\infty \\
-1{,}526 & 63{,}23468 & 2{,}950 & 55{,}15364 & 1{,}579
\end{pmatrix}
\end{array}
$$

Self-attention layers in the decoder allow **each position in the decoder to attend to all positions in the decoder up to and including that position**. We need to prevent **leftward information flow** in the decoder to preserve the **auto-regressive property**. We implement this inside of scaled dot- product attention by masking out (setting to $-\infty$) **all values in the input of the softmax which correspond to illegal connections**.

4.→The Final Linear and Softmax Layer

The decoder stack outputs a **vector of floats**. **How do we turn that into a word?** That's the job of the final Linear layer which is followed by a Softmax Layer.

> ***The Linear layer is a simple fully connected neural network that projects the vector produced by the stack of decoders, into a much, much larger vector called a logits vector***. *This space is the size of vocabulary (all words). We just project the matrix of weights (provided by the decoder block) into a "vocabulary space".*

Mathematically speaking, what does it mean?

> *Appelons S la matrice en sortie du décodeur. On la multiplie par une matrice de poids (qui peuvent apprendre) W1*. C'est une couche totalement connectée qui

projette simplement la sortie précédente dans un espace de la taille de notre vocable.

W1 est la matrice qui va permettre d'extraire un mot dans notre dictionnaire de vocabulaire. Elle aura donc pour dimensions [dimension des embeddings, i.e. dmodel*] x [nombre de mots dans notre vocable].*

Let's assume that our model knows 10,000 unique English words (our model's "output vocabulary") that it's learned from its training dataset. This would **make the logits vector 10,000 cells wide — each cell corresponding to the score of a unique word**. That is how we interpret the output of the model followed by the Linear layer.

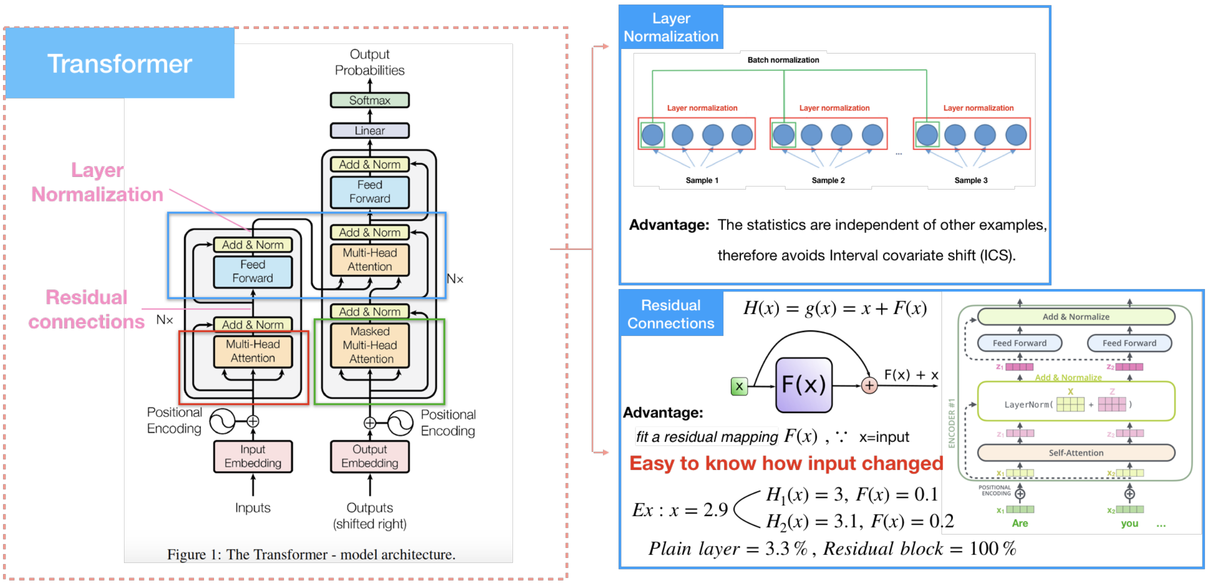
The **softmax layer** then **turns those scores into probabilities** (all positive, all add up to 1.0). **The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step**. Softmax provides us **the most likely word to predict** (we take the word of the column which give us the highest probability).

https://jalammar.github.io/illustrated-transformer/

This figure starts from the bottom with the vector produced as the output of the decoder stack. It is then turned into an output word.

5.→ Residual connection

A residual connection is basically just taking the input and adding it to the output of the sub-network, making training deep networks easier in the field of computer vision. Layer normalization is a normalization method in deep learning that is similar to batch normalization. In layer normalization, the statistics are computed across each feature and are **independent of other examples**. The independence between inputs means that each input has a different normalization operation.



7. → Model Training — How BERT is trained?

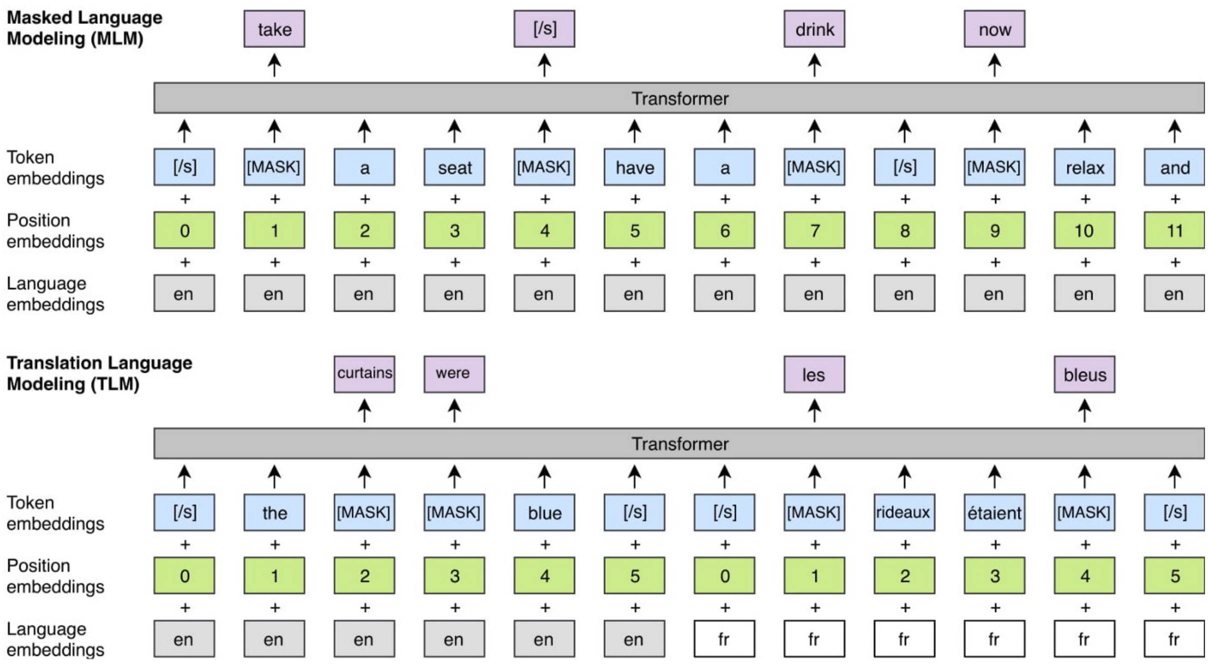
A — Masked Language Modeling (MLM)

"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



XLNet

B- 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

Applications?

A- Real Applications : Pre-trained vs. Fine-tuned

A review of BERT based models (https://towardsdatascience.com/a-review-of-bert-based-models-4ffdc0f15d58)

Models pretrained on domain/application specific corpus are **Pre-trained models.** Training on domain specific corpus has shown to yield better performance when fine-tuning them on downstream NLP tasks like NER etc. for those domains, in comparison to fine tuning BERT *(which was trained on BooksCorpus and Wikipedia)*.

- BioBERT *(biomedical text)*
- SciBERT *(scientific publications*)
- ClinicalBERT *(clinical notes)*
- G-BERT (*medical/diagnostic code representation and recommendation*)
- M-BERT from 104 languages for zero-shot cross lingual model transfer *(task specific annotations in one language is used to fine tune model for evaluation in another language)*
- ERNIE *(knowledge graph)* + ERNIE (2) incorporates knowledge into pre-training but by masking entities and phrases using KG.
- TransBERT — unsupervised, followed by two supervised steps, for a story ending prediction task
- videoBERT (*model that jointly learns video and language representation learning*) by representing video frames as special descriptor tokens along with text for pretraining. This is used for video captioning.

Fine tuned models. Models fine tuned for a specific task using a pretrained model :

- DocBERT (Document classification)
- PatentBERT(Patent classification)

B- Case study

- Better Sentiment Analysis with BERT : Fine-tune by applying a single new layer and softmax on top of the pre-trained model + Serving with Docker and Tensorflow + API

- Building a Multi-label Text Classifier using BERT and TensorFlow : In multi-label classification instead of **softmax()**, use **sigmoid()** to get the probabilities. Sigmoid allows to deal with non-exclusive labels (a.k.a. multi-labels), while softmax deals with exclusive classes + compute a *logit* (also called a score)

- Logistic regression & BERT : run logistic regression with BERT embeddings

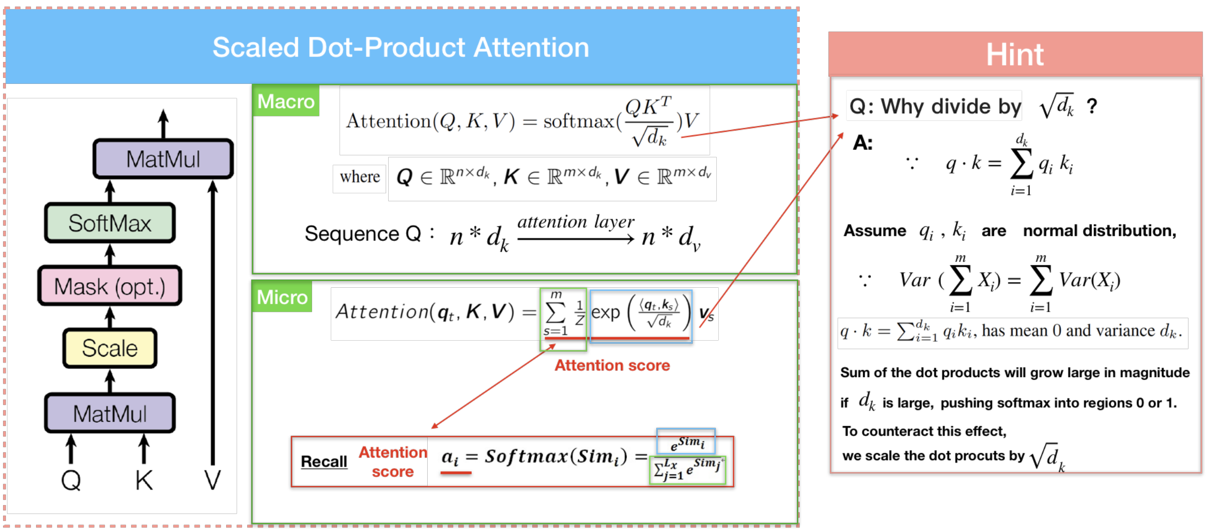
- BERT Fine-Tuning Tutorial with PyTorch :

- Taming the BERT — a baseline : Fine-tune the BERT model, instead of using the pre-trained weights + use a mix of the BERT layers, instead of just the output of the last layer + tune some of the hyperparameters of the MLP model

- BERT : Faire comprendre le langage naturel à une machine, en pré-entraînant des Transformers bi-directionnels profonds : French classification

BONUS :

1. Why divide by square(dk) ?



2. Fine tuning

From https://yashuseth.blog/2019/06/12/bert-explained-faqs-understand-bert-working/

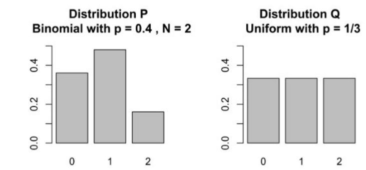
- What is the fine-tuning procedure for sequence classification tasks?
- What is the fine-tuning procedure for sentence pair classification tasks?
- What is the fine-tuning procedure for Question Answering tasks?
- What is the fine-tuning procedure for single sentence tagging tasks?

3. BERT as a service

The final hidden states (the transformer outputs) of the input tokens can be concatenated and / or pooled together to get the encoded representation of a sentence. **bert-as-a-service** is an open source project that provides BERT sentence embeddings optimized for production. **Serving Google BERT in Production using Tensorflow and ZeroMQ**.

4. Kullback Leibler divergence

Kullback[2] gives the following example (Table 2.1, Example 2.1). Let P and Q be the distributions shown in the table and figure. P is the distribution on the left side of the figure, a binomial distribution with $N = 2$ and $p = 0.4$. Q is the distribution on the right side of the figure, a discrete uniform distribution with the three possible outcomes $x = 0, 1$, or 2 (i.e. $\mathcal{X} = \{0, 1, 2\}$), each with probability $p = 1/3$.



x	0	1	2
Distribution *P*(x)	0.36	0.48	0.16
Distribution *Q*(x)	0.333	0.333	0.333

The KL divergences $D_{\text{KL}}(P \parallel Q)$ and $D_{\text{KL}}(Q \parallel P)$ are calculated as follows. This example uses the natural log with base *e*, designated $\ln$ to get results in nats (see units of information).

$$\begin{aligned} D_{\text{KL}}(P \parallel Q) &= \sum_{x \in \mathcal{X}} P(x) \ln\left(\frac{P(x)}{Q(x)}\right) \\ &= 0.36 \ln\left(\frac{0.36}{0.333}\right) + 0.48 \ln\left(\frac{0.48}{0.333}\right) + 0.16 \ln\left(\frac{0.16}{0.333}\right) \\ &= 0.0852996 \end{aligned}$$

$$\begin{aligned} D_{\text{KL}}(Q \parallel P) &= \sum_{x \in \mathcal{X}} Q(x) \ln\left(\frac{Q(x)}{P(x)}\right) \\ &= 0.333 \ln\left(\frac{0.333}{0.36}\right) + 0.333 \ln\left(\frac{0.333}{0.48}\right) + 0.333 \ln\left(\frac{0.333}{0.16}\right) \\ &= 0.097455 \end{aligned}$$

https://en.wikipedia.org/wiki/Kullback–Leibler_divergence