# LLMs: a theoretical overview of the Transformer architecture and the novel concepts of LLaMA 3, Gemma and Mixtral

Rafael Guedes

July 11th, 2024

Medium Article

## Experience

Marley Spoon | Sr ML Engineer
2022-

ZAAI | Sr Data Scientist
2023-

Farfetch | Data Scientist
2019-2022

## Education

Masters in DS and Eng
2020-2022

Post-Grad. in BI and Analytics
2019-2020

Bachelor in Management
2015-2018

**Contacts:** https://www.linkedin.com/in/rafaelguedes97/ https://medium.com/@rjguedes **Github** (1) (2)

# LLMs: a theoretical overview of the Transformer architecture and the novel concepts of LLaMA 3, Gemma and Mixtral

Rafael Guedes

July 11th, 2024

Medium Article

## First things first: Tokenization…

## Tokenization

Tokenization is used to transform raw input text data into tokens.

SentencePiece was developed by Google and one of the most used tokenizers for LLMs.

Kudo, Taku, and John Richardson. “Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing.” *arXiv preprint arXiv:1808.06226* (2018).

## 1. Splits the words into individual characters

Imagine that the training data contains the following set of words and the frequency of each word has been determined:

```
("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)
```

Then, if we split into individual characters we will get:

```
("h" "u" "g", 10), ("p" "u" "g", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "u" "g" "s", 5)
```

## 2. Iteratively merges the most frequent character pairs into subwords until a predefined vocabulary size is reached

```
("h" "u" "g", 10), ("p" "u" "g", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "u" "g" "s", 5)
```

We see that `"u"` followed by `"g"` appears 20 times, the most frequent symbol pair, therefore we merge them:

```
("h" "ug", 10), ("p" "ug", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "ug" "s", 5)
```

Repeat until it reaches a vocabulary size of 9 tokens:

```
("hug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un", 4), ("hug" "s", 5)
```

## 3. When it comes to unseen data

```
("hug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un", 4), ("hug" "s", 5)
```

If we get the word `"bug"` in new data, it will be tokenized as `["b", "ug"]`

But, if the new word is `"mug"` since, we do not have `"m"` in the vocabulary, then it will be encoded as `["<unk>", "ug"]`

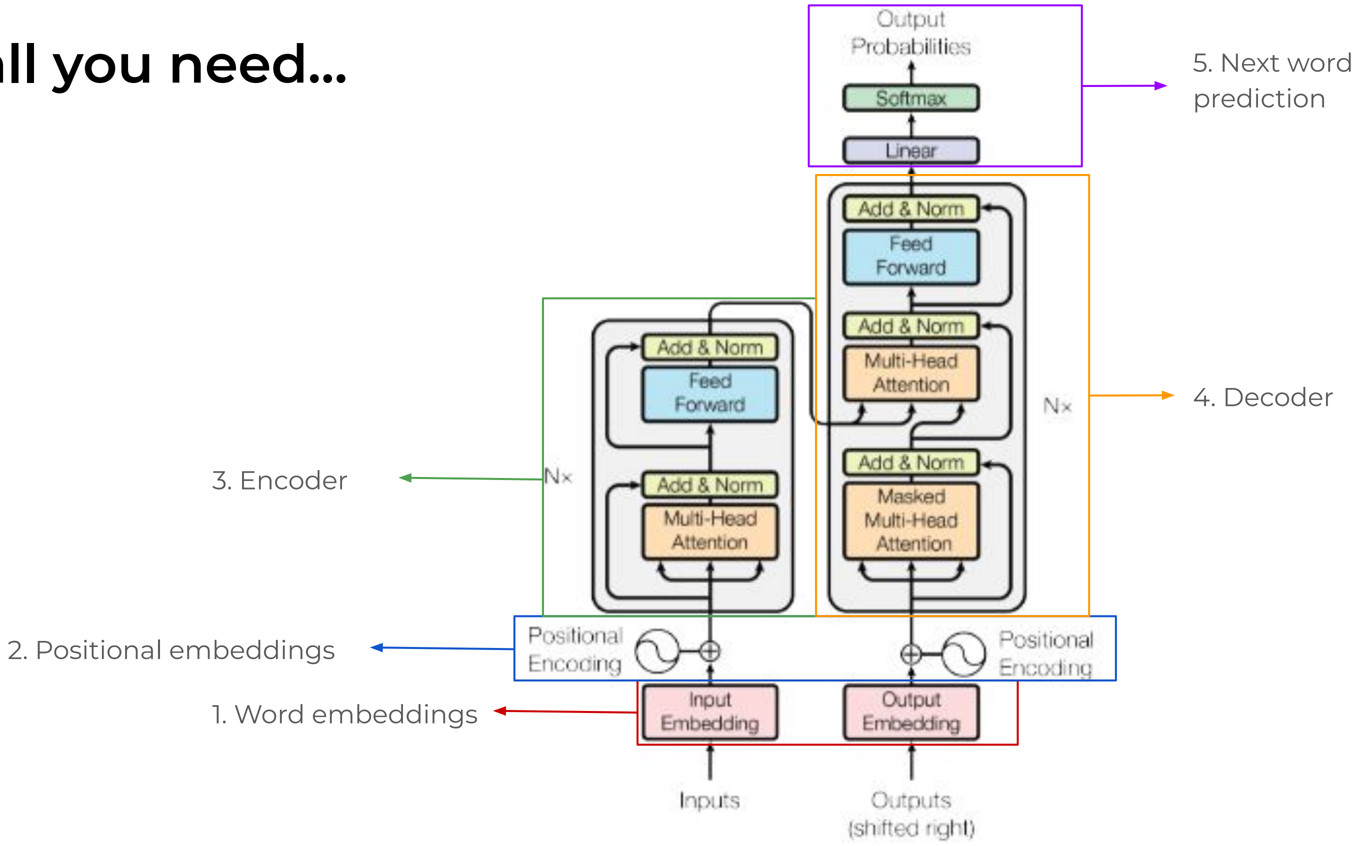
## 4. Encoding

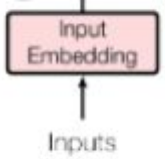
Finally, each subword will be encoded by assigning a token ID:

```
"hug" -> 1

"p" -> 2

"ug" -> 3

"un" -> 4

"b" -> 5

...
```

## Transformer Architecture

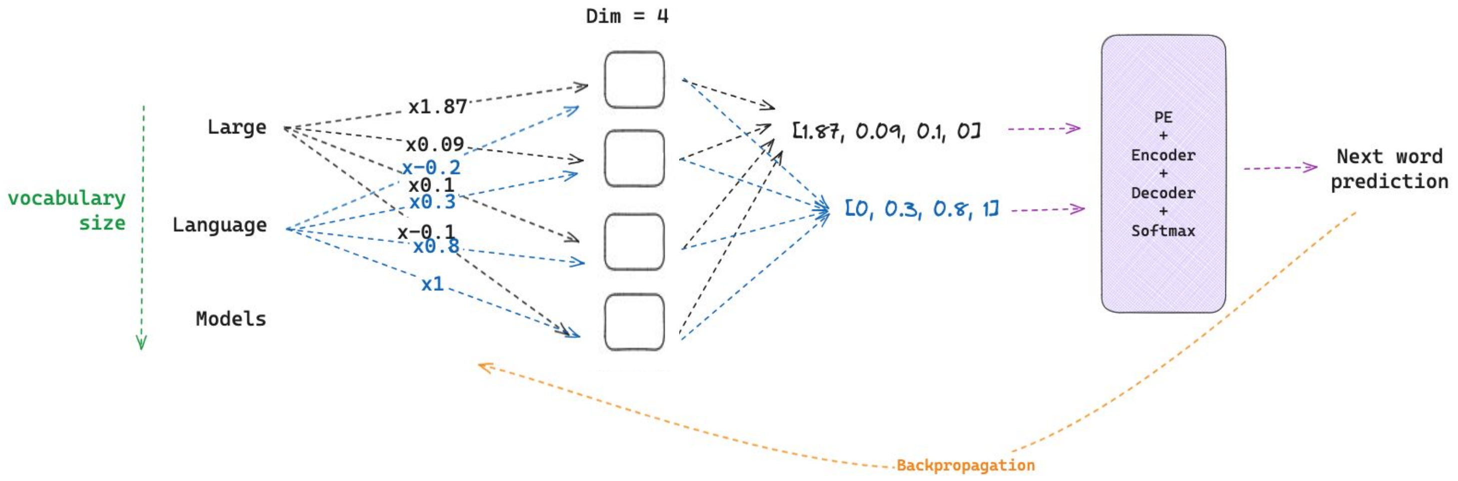
## Attention is all you need...

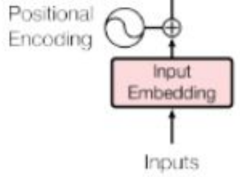




## 1. Word Embeddings

First layer in a Transformer is the word embedding that converts each token input into a d-dimensional vector. Let's consider each word as token:





## 2. Positional Embedding

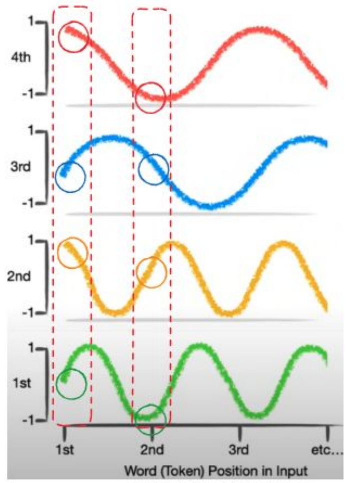
Second step in a Transformer is the positional embedding that helps Transformers keep track of order since they are order invariant.

If positional embeddings did not exist the sentences: ***‘LLaMA is better than Gemma’*** and ***‘Gemma is better than LLaMA’*** would mean the **same**.

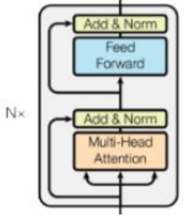
Large [1.87, 0.09, 0.1, 0] + [0, 1, 0, 1] = [1.87, 1.09, 0.1, 1]

Language [0, 0.3, 0.8, 1] + [-0.9, 0.4, 0.1, -0.9] = [-0.9, 0.7, 0.9, 0.1]

Sine and Cosine functions with different frequencies



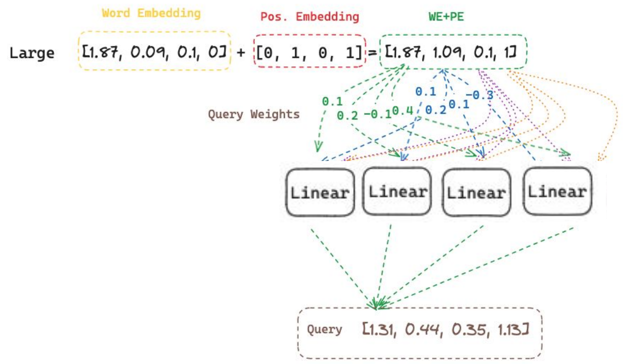
Vaswani et al., “Attention Is All You Need”, arXiv preprint arXiv:1706.03762 (2017).



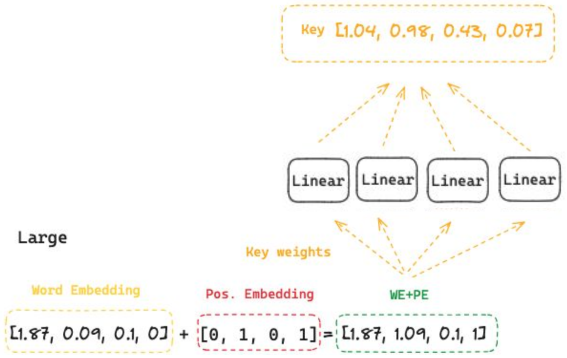
## 3. Encoder - Self-Attention

Self-attention calculates how similar each word is to all of the words in a sentence, including itself in order to determine how the Transformer will encode each word.

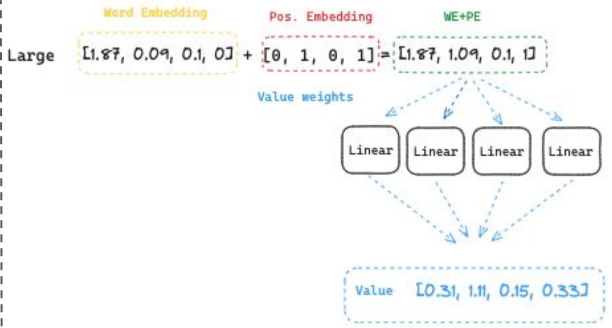
### 3.1. Query



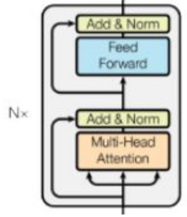
### 3.2. Key



### 3.3. Value



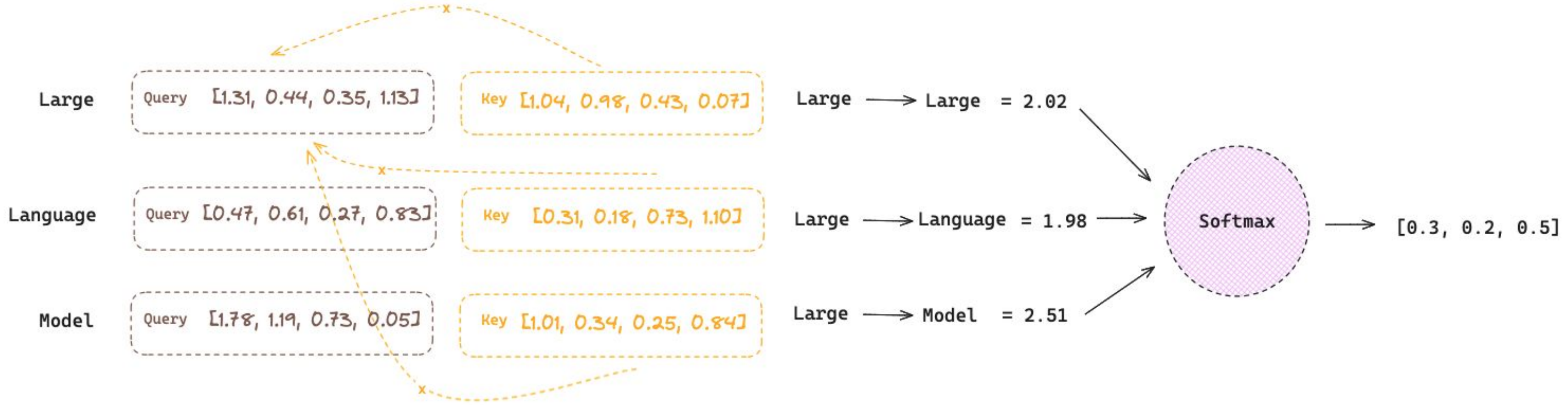
## 3. Encoder - Self-Attention

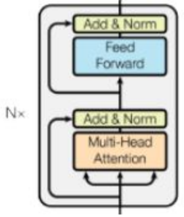


3.4. Calculate similarity between words based on query and keys.

For example, the similarity between the word *'Large'* and all words will be the dot product between the Query of the word *'Large'* and all the keys.

After that, the values go through a softmax activation function so that they sum up to 1 and we consider this the weight that each word has in the encoding of *'Large'*.

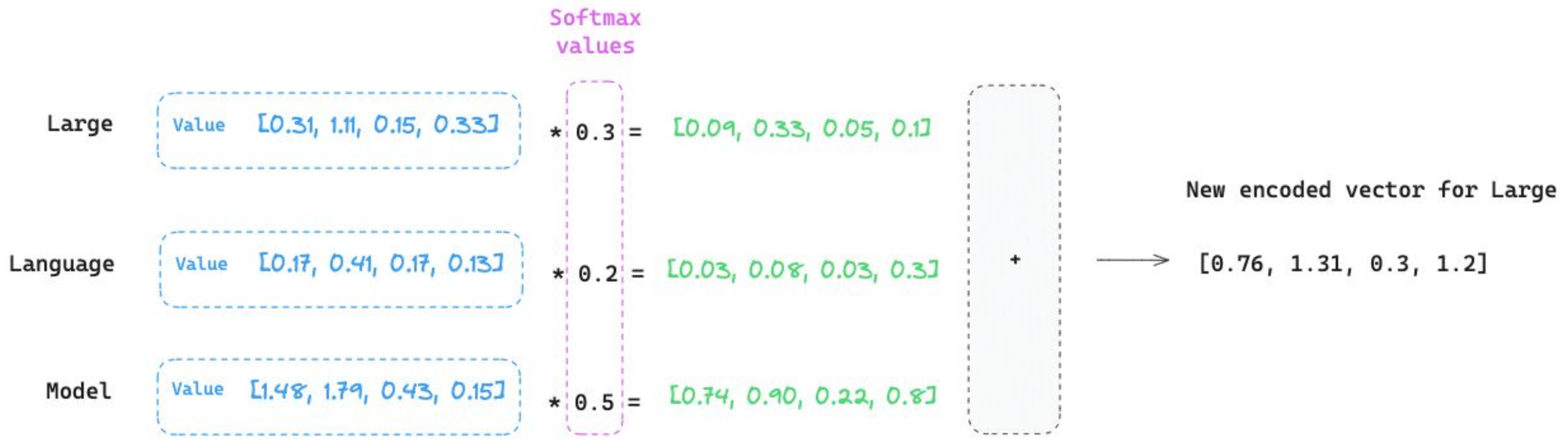

Large
Query [1.31, 0.44, 0.35, 1.13]
Key [1.04, 0.98, 0.43, 0.07]
Language
Query [0.47, 0.61, 0.27, 0.83]
Key [0.31, 0.18, 0.73, 1.10]
Model
Query [1.78, 1.19, 0.73, 0.05]
Key [1.01, 0.34, 0.25, 0.84]
x
x
x
Large → Large = 2.02
Large → Language = 1.98
Large → Model = 2.51
Softmax
[0.3, 0.2, 0.5]



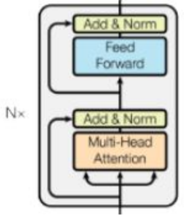
## 3. Encoder - Self-Attention

3.5. Create values to represent each word in the sentence (same process as queries and keys).

Calculate the new encoded vector of each word based on the Value of each word and the its weight. This way we embed context into the word vector representation.

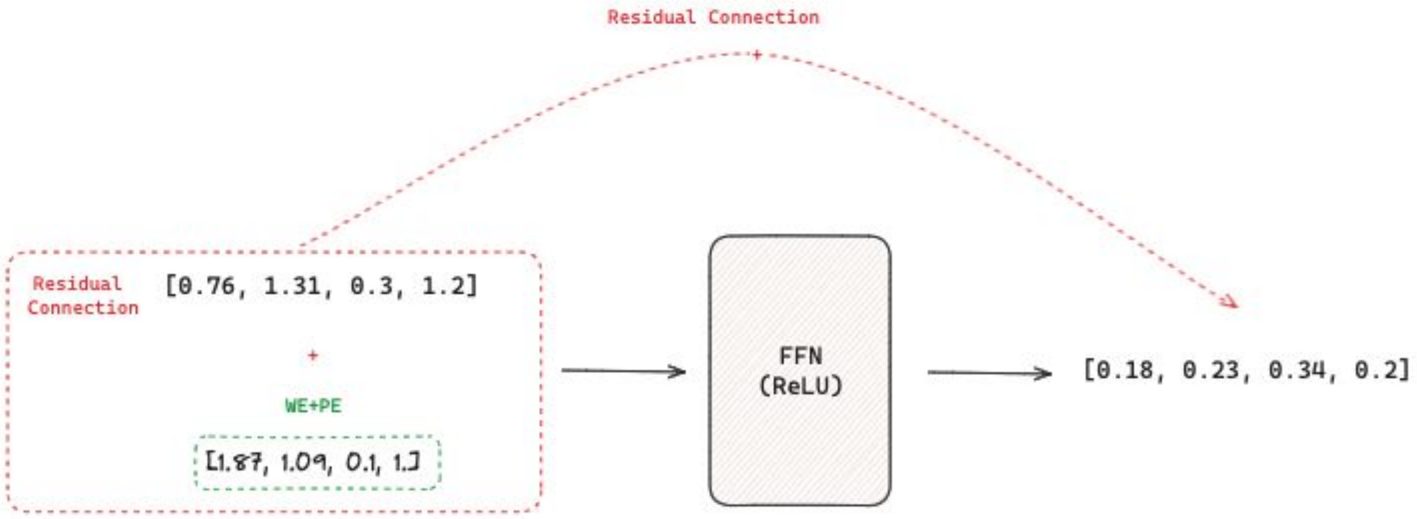


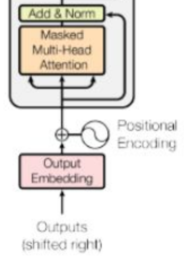
# 3. Encoder - Feed Forward Network



3.6. There is a residual connection in the end that adds the new word embedding to the initial word embedding with positional encoding.

This new vector is fed into a Feed Forward Neural Network with ReLU activation function.

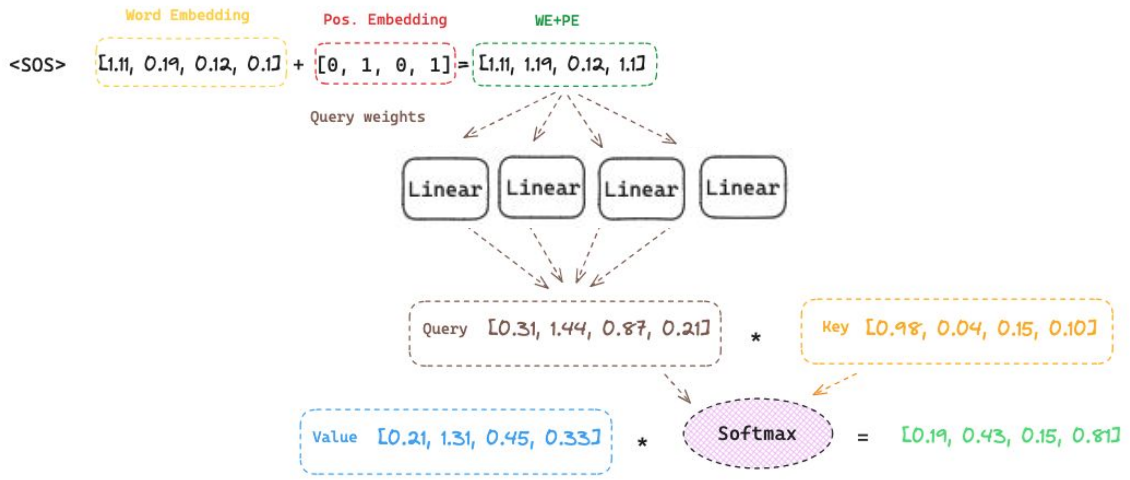


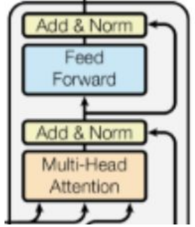


## 4. Decoder

The Decoder is responsible for generating the vector used in the next word prediction component based on the output of the encoder and all the tokens in the current sequence.

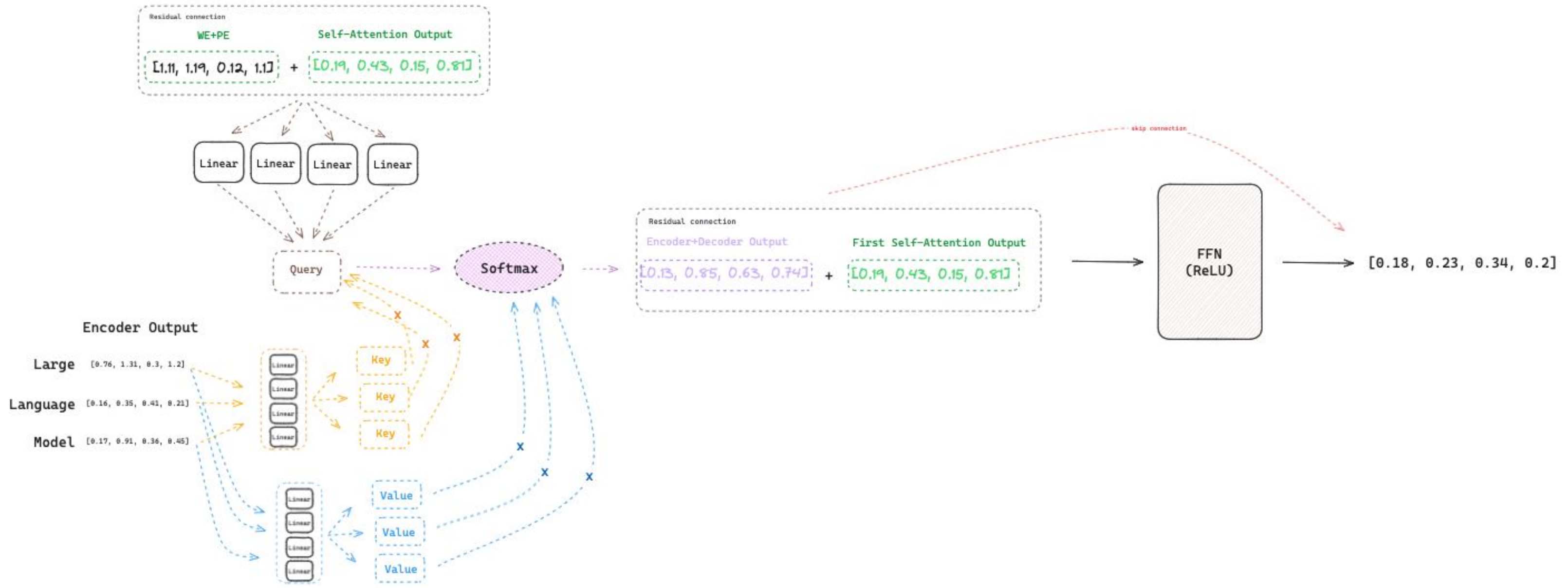
4.1. First token to be processed in <SOS> which means start of sentence. Just like before, this token will be encoded through word embedding + positional encoding + self-attention

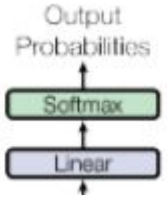




## 4. Decoder

4.2. There is a residual connection that will add to the new word representation the old word representation and the positional encoding. This final vector together with the encoder output go through a new self-attention layer, followed by a skip connection and a FNN.

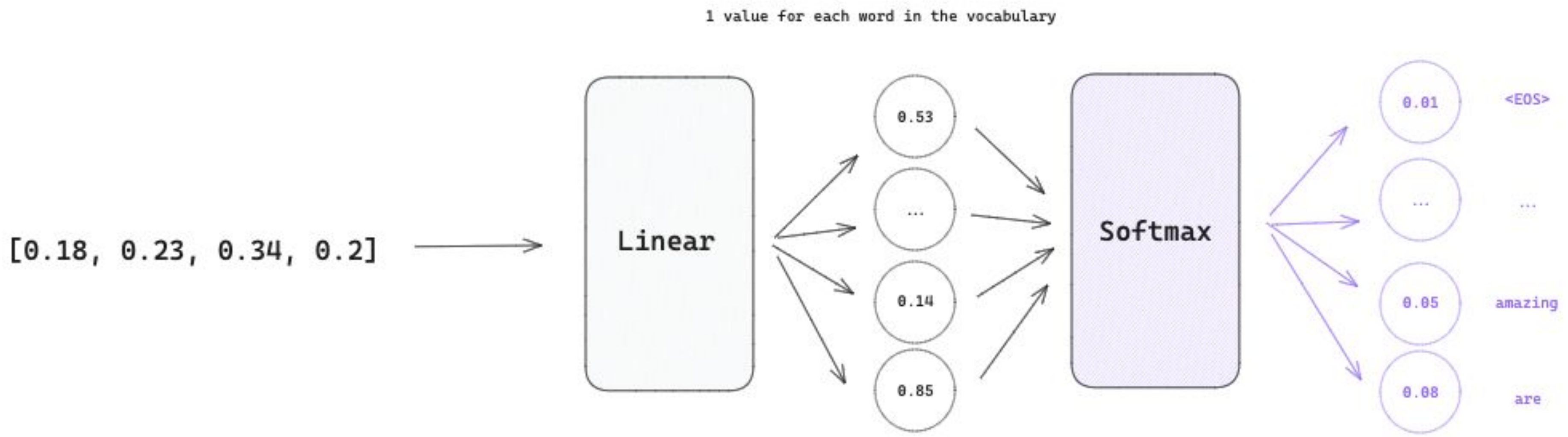




## 5. Next Word Prediction

The final output of the decoder goes through a simple neural network that will convert this vector into a new one with the same dimension as the vocabulary size.

After that, a Softmax function is applied to determine which word should come next. The process stops when a <EOS> token (end of sentence) is predicted.



## Final remarks about Self-Attention:

- Query, Keys and Values are calculated in parallel, i.e., Q, K and V are calculated at the same time for each word.
- We can have several Self-Attention Cells to train different weights and capture different relationships between words (the original had 8 cells that are combined through concatenation followed by a linear layer).
- There are multiple stacked layers (N=6) of self-attention cells + FFN which means the second layer input is the output of the first layer, the third layer input is the second layer output and so on and so forth.
- After each self-attention cell and FFN block, a normalization step is applied.

Vaswani et al., “Attention Is All You Need”, arXiv preprint arXiv:1706.03762 (2017).

## LLaMA 3, Mixtral and Gemma: what’s new?

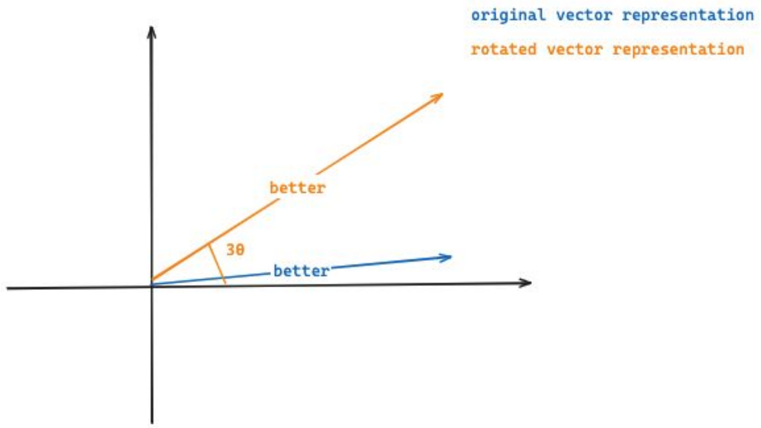
## Overview

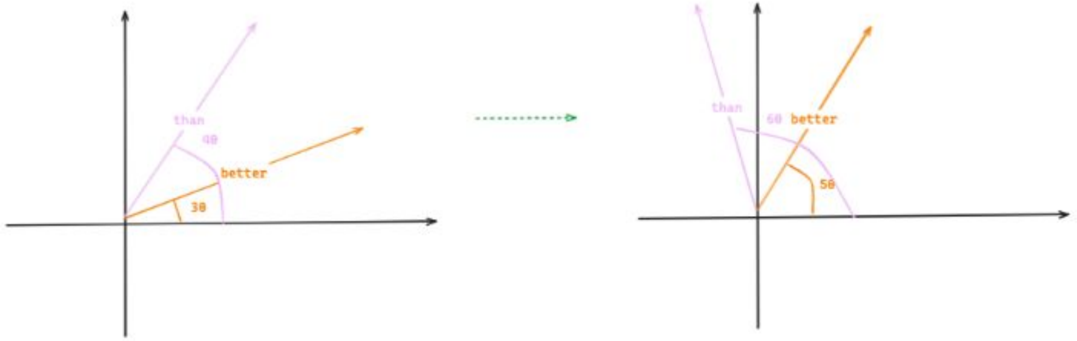
| | Original | Gemma | Mixtral | LLaMA 3 |
|---|---|---|---|---|
| Vocab. Size | 32k | 256k | 32k | 128k |
| Context Length | 512 | 8192 | 4096 | 8192 |

## Positional Embeddings

LLaMA 3 and Gemma use Rotary Positional Embedding (RoPE) instead of the original version. This approach brings benefits such as modelling the relative position between tokens which means that the tokens in position 1 and 2 will be more similar than the tokens in position 1 and 500.

Let’s consider the sentence *‘Gemma is better than LLaMA’* and a 2D word embedding.
The positional embedding of the word better will be given by a rotation of the vector based on position 3 and a constant θ. If two new words are added to the sentence, then the angle between *‘better’* and *‘than’* will keep the same.

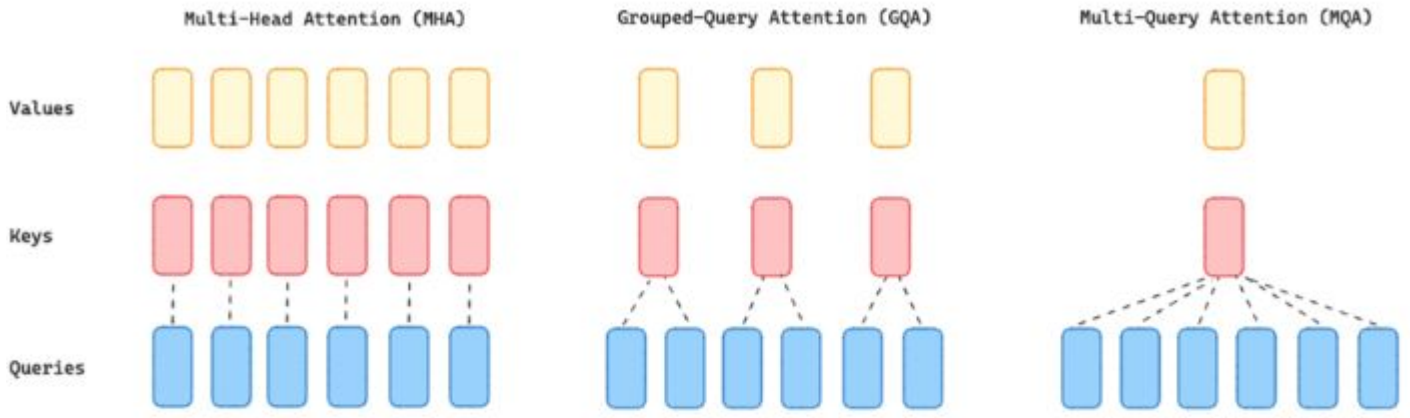




## Grouped Query Attention instead of Multi Head Attention

LLaMA 3, Gemma and Mixtral replaced the traditional Multi Head Attention with Grouped Query Attention for faster decoding, hence, faster inference.

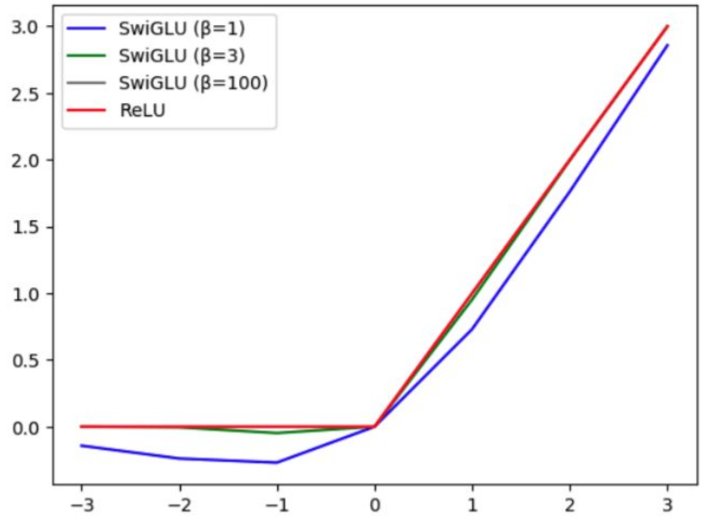
GQA-G divides query values into G groups that share a single key and value head (GQA-1 = MQA  while a GQA-H = MHA). This approach reduces the number of keys and values heads into a single key and value per query group, accelerating the inference speed and reducing the memory requirements during decoding with a quality closer to MHA.

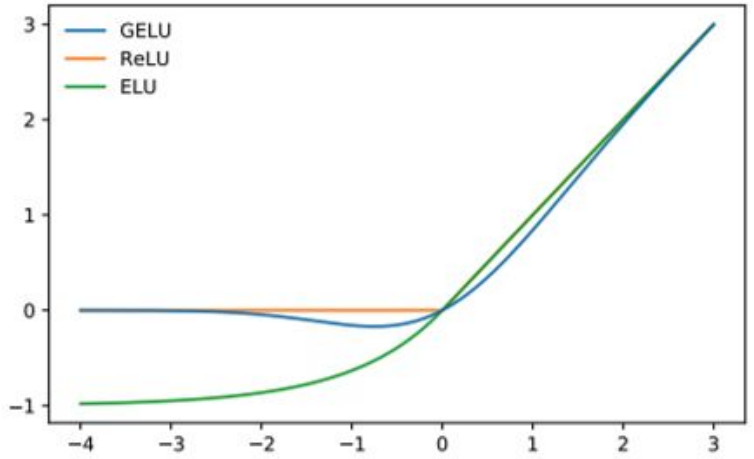

Multi-Head Attention (MHA)
Grouped-Query Attention (GQA)
Multi-Query Attention (MQA)
Values
Keys
Queries

## Activation function

LLaMA 3 and Gemma replaced the traditional ReLU activation function in the FFN block with SwiGLU and GeGLU, respectively.

These functions, unlike ReLU that converts all negative values to 0, have a parameter that smooths this conversion where the probability of setting negative values to 0 increases as these values are closer to 0.





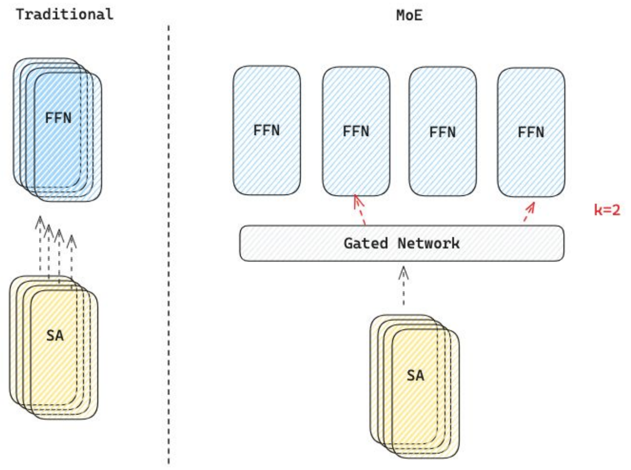
## SMoE: Sparse Mixture of Experts

Mixtral differs from the other architectures by using Mixture of Experts rather than stacking a FFN on top of the different attention cells.

Each Expert is responsible for processing a type of token, for example, one can be a punctuation expert, a visual description expert, or a number expert.

The Expert(s) that is going to process the token is chosen by a Gated Network trained to perform this allocation.

**Advantages:**
- More efficiency by activating less model parameters;
- More accurate prediction because each expert is focused on a specific task



# LLaMA 3 8B vs Gemma 7B vs Mistral 7B

**Use Case:** Simple RAG system with 100 questions, contexts and ground truth answers.

**Metrics:**

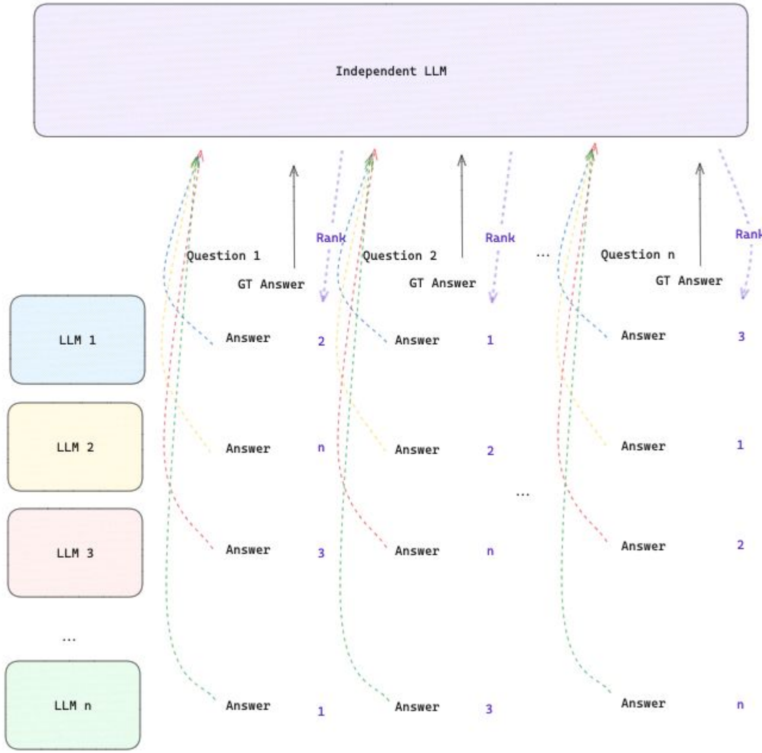
- Words produced per second

- Answer Length

- RAQ (Relative Answer Quality) based on a rank provided by GPT 3.5.

Prompt:

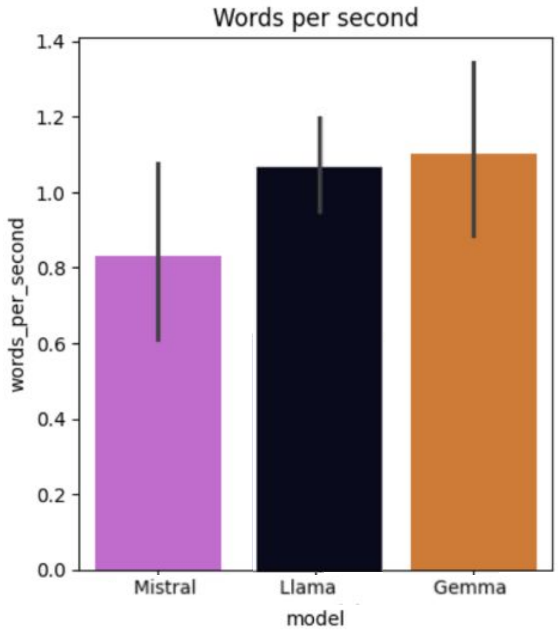
```
Based on the correct answer: {Ground Truth Answer},
rank the IDs of the following answers from the most
to the least correct one:
ID: 1 Answer: {LLM 1 Answer}
ID: 2 Answer: {LLM 2 Answer}
ID: 3 Answer: {LLM 3 Answer}
...
```

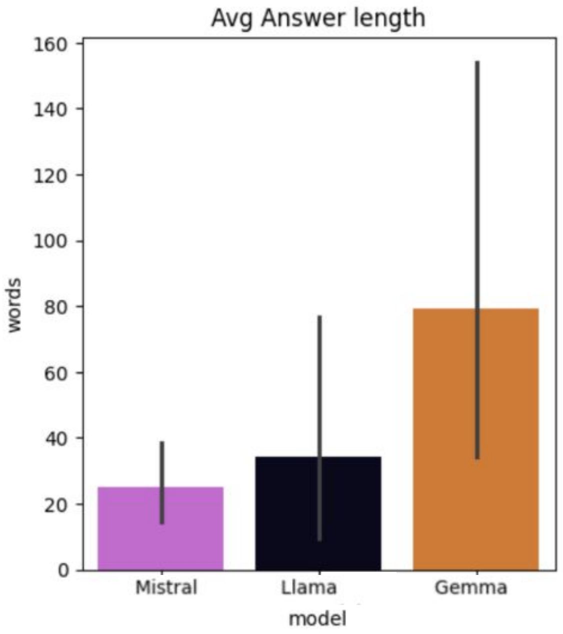


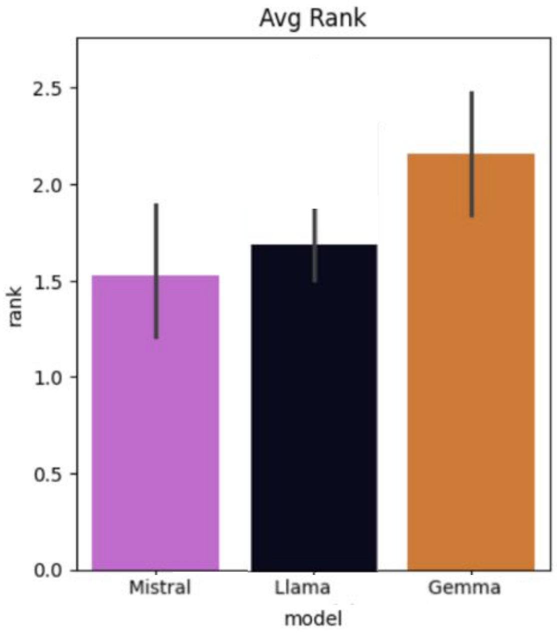
RAQ Article

# LLaMA 3 8B vs Gemma 7B vs Mistral 7B

**Gemma** is the fastest and with the one producing more words per answer. But, **Mistral** produces better answers according to GPT 3.5.







## Hands-on with DSPy