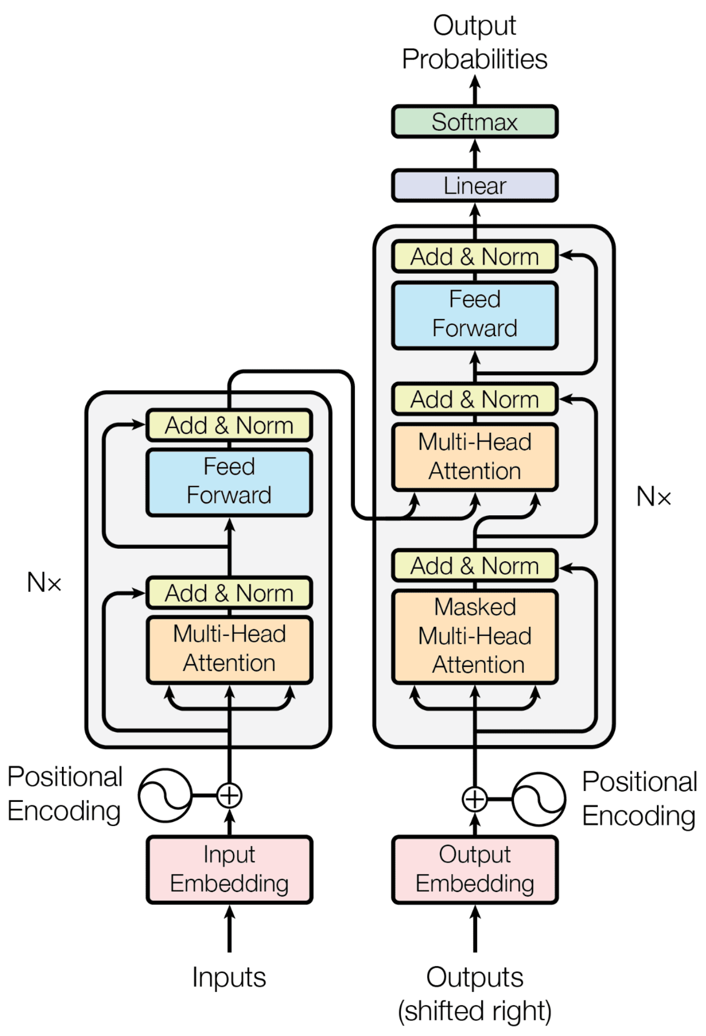


Transformers -Deep Learning-

For Natural Language Processing, and other AI applications including Computer Vision, Deep Reinforcement Learning, Generative AI and Others.

Transformers | Iheb BOUARICHE | 03-07-2023

I. What’s a language?

Language is a system of communication that encompasses various elements, such as grammar, vocabulary and rules that enable humans to express thoughts, ideas and feeling. it allows us to convey information, establish social connections and it’s a key aspect of culture and identity.

Training machines to comprehend language rules and contextual meaning is a complex and formidable task to tackle manually. Therefore, training machines for this task is necessary, and that's where Artificial intelligence’ Natural Language Processing (NLP) comes into play.

II. Natural Language Processing:

Natural Language Processing (NLP) is a field of artificial intelligence that focuses on enabling computers to understand, interpret, and generate human language. Various techniques are used in NLP, including machine learning algorithms and complex neural network architectures, such as the Transformer architecture based on the attention mechanism and recurrent neural networks.

NLP encompasses a wide range of tasks, including language translation, sentiment analysis, information extraction, question answering, text summarization and others.

III. Preprocessing of data:

1. Text Preprocessing:

The input text is preprocessed to remove any unnecessary characters or formatting issues. This may involve lowercasing the text, removing punctuation, and handling special characters.

2. Tokenization:

Tokenization is a crucial step in Natural Language Processing (NLP) where text, such as sentences or documents, is divided into smaller units called tokens. These tokens can be words, subwords, or characters, depending on the specific technique employed. The main objective of tokenization is to convert unstructured text data into a structured format that can be efficiently processed by NLP algorithms.

Tokenization can be performed in different ways depending on the requirement of the task and the language being processed. Here are some common tokenization techniques:

a. Character tokenization: involves splitting a text into individual characters.
b. Word Tokenization: it splits text into individual words or word-like units.
c. Subword tokenization: It breaks down words into smaller subword units, such as character n-grams or syllables. (n-gram is a sequence of characters of length “n”, and syllables are units of pronunciation that form the building blocks of words)
d. Sentence tokenization: involves splitting a text into individual sentences.

Note:

- It can be challenging to understand the context or generate text with only characters.
- Word tokenization requires a large vocabulary to handle all the words in a language.

- Subword tokenization is recommended when dealing with languages that have large vocabularies.
- Different pretrained models like GPT and BART use subword tokenization technique, but they may employ different strategies. While both models generate tokens as the output, the specific tokens produced by each strategy may not be the same.
- Tokenization is the first applied method to a text as a preprocessing method for solving NLP problems.
- After tokenization, the next steps in NLP can vary depending on the specific task or analysis being performed. Some common next steps are: Stop word removal, named entity recognition, tagging, Lemmatization or stemming, Named entity recognition, Numericalization and Feature extraction.

3. Numericalization:

Numericalization is the process of converting tokens or text data into numerical representations that can be used as input for NLP models. Numericalization typically involves assigning a unique integer or index to each token (IDs).

These IDs can then be represented as one-hot vectors, but this method is not efficient when feeding them to a neural network because most of the inputs will be zeros.

4. Word Embeddings:

Numericalized tokens can be mapped to dense vector representations known as word embeddings. The underlying concept behind word embeddings is to capture both the semantic and syntactic relationships between words, enabling machines to understand and process textual data more effectively. Word embeddings are typically learned from large corpora of text using unsupervised learning algorithms to train neural networks.

The resulting word vectors are continuous, dense, and low-dimensional representations. Similar words are located close to each other in the embedding space, and arithmetic operations can be applied to word vectors to capture analogical relationships.

The most popular algorithms for training word embeddings using neural networks are:

Word2Vec: The key idea behind Word2Vec is the distributional hypothesis, which suggests that words that appear in similar contexts tend to have similar meanings. Based on this hypothesis, Word2Vec aims to learn word embeddings by predicting the context of a target word within a given window of surrounding words.

GloVe (Global Vectors for Word Representation): The primary goal of GloVe is to capture both the local context of words and the global statistical properties of word co-occurrence. The algorithm begins by constructing a co-occurrence matrix that represents the frequency of words appearing together in a given corpus. The matrix is then factorized to obtain word vectors that capture the relationships between words.

Note: that GPT and BERT both have their own trained word embedding model.

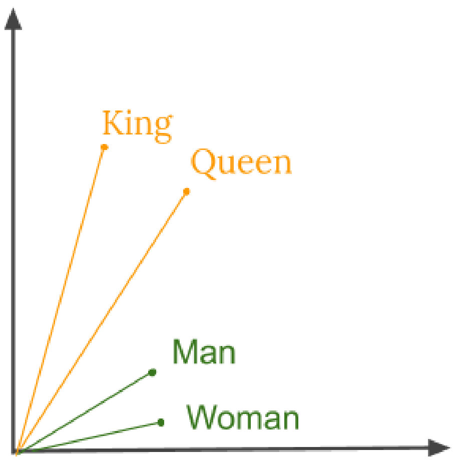


Figure 01: Word embeddings

Note 02: In PyTorch, when multiplying a one-hot vector with a matrix, it is equivalent to extracting a row from the matrix, and it simplifies the calculations.

IV. Traditional Techniques:

1. Machine learning:

a. <u>Naïve Bayes:</u>

Naive Bayes is a text classification algorithm commonly used in NLP. It assumes that the features (words or tokens) in a text are independent given the class label, making it computationally efficient and suitable for large feature spaces. Training the model can be applied by estimating prior probabilities and feature likelihoods, and finally, classification of new documents by calculating posterior probabilities using Bayes' theorem. Despite the "naive" assumption of feature independence, Naive Bayes classifiers often perform well in NLP tasks, requiring relatively small amounts of training data.

b. <u>Support Vector Machine -SVM-:</u>

Support Vector Machines (SVMs) are commonly used in Natural Language Processing (NLP) tasks such as sentiment analysis and text classification. SVMs require converting textual data into numerical feature vectors through techniques like Bag-of-Words or word embeddings. The SVM model is then trained on a labeled dataset, aiming to find an optimal hyperplane that separates different classes. Hyperparameters of the SVM, such as the choice of kernel and regularization parameter, need to be tuned. The trained SVM model is evaluated using a testing set, and metrics like accuracy, precision, recall, and F1-score are used. Once trained, the SVM model can make predictions on new, unlabeled text data. While deep learning models have gained popularity in NLP, SVMs are still useful, especially for small datasets or when interpretability is important.

c. <u>Decision Tree:</u>

Decision tree algorithms can be applied to various NLP tasks like text classification, sentiment analysis, named entity recognition, and information extraction. Decision Tree is versatile as it can handle both categorical and numerical features. The process involves feature extraction from text, followed by labeling or classification, sentiment analysis, named entity recognition, or information extraction. Decision trees learn from labeled datasets and use extracted features to classify or predict sentiments, identify named entities, or extract specific information. However, for more complex problems, other models like random forests, gradient boosting, or deep learning architectures might be more suitable. The choice of model depends on the task requirements and available data.

Check: <u>My pdf of machine learning and Data science by Iheb BOUARICHE on my GitHub repository.</u>

2. Recurrent neural network:

a. Basic Recurrent neural network -RNN- architecture: are widely used in NLP due to their ability to process sequential data and capture temporal dependencies. In an RNN, each output serves as an input for the next step, allowing the network to retain information from previous data points. However, it suffers from the problem of vanishing gradients, where the network has difficulty retaining information from distant past inputs.

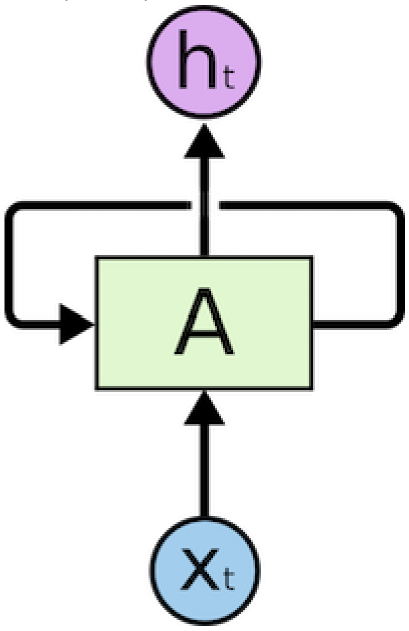


Figure 02: Basic recurrent neural network architecture

b. Long Short-Term Memory -LSTM- architecture: is commonly used in NLP tasks. It addresses the vanishing gradient problem in traditional RNNs by incorporating memory cells, enabling the model to capture and retain long-range dependencies in sequential data. The LSTM's ability to remember information over longer sequences makes it effective for tasks such as language modeling, machine translation, and sentiment analysis.

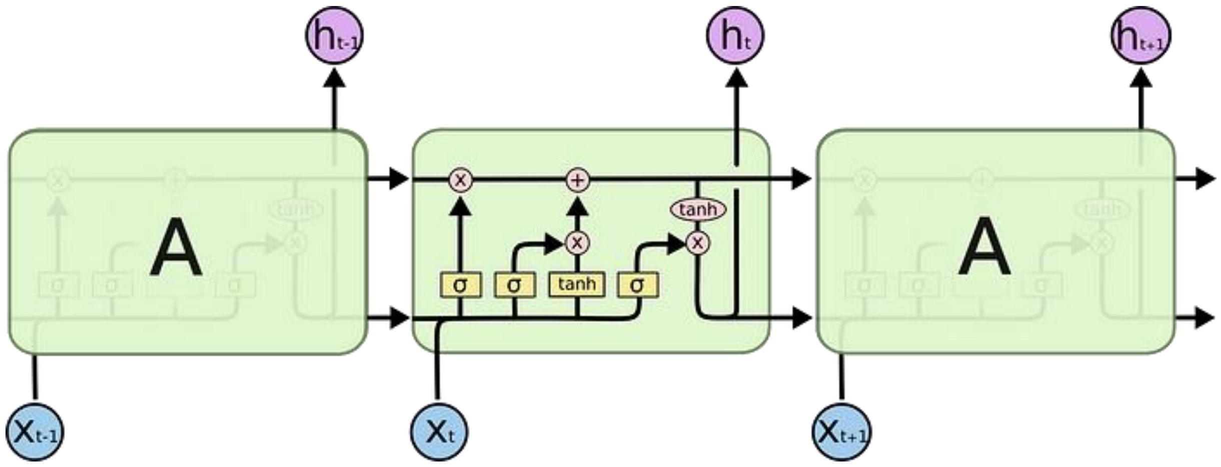
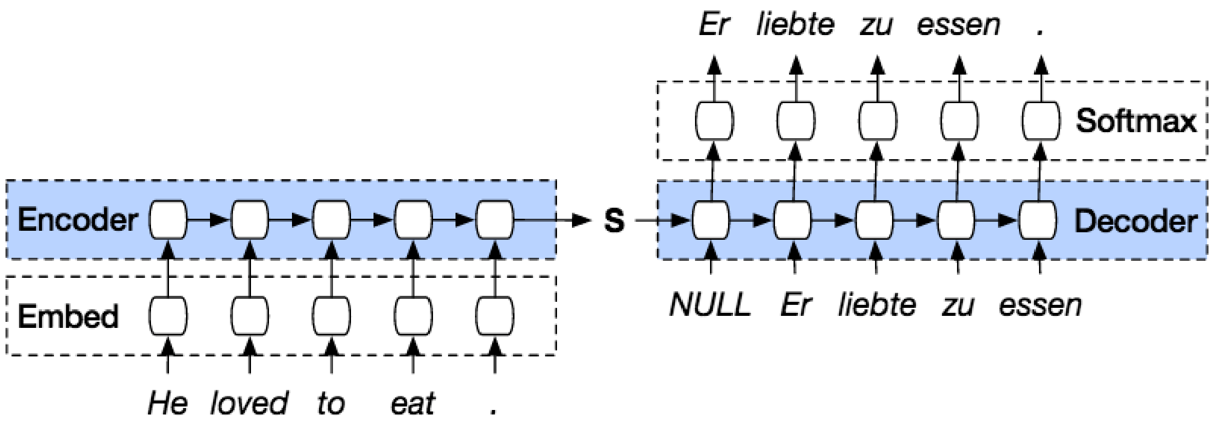


Figure 03: Long short-term memory architecture.

However, when it comes to more complex tasks like sequence-to-sequence problems (e.g., machine translation) or situations where the input and output sequences differ in length, the

LSTM architecture can face a bottleneck problem. This issue arises because the last output of the encoder "**S**" is responsible for encapsulating all the information from the preceding data, which might not be sufficient.



Another limitation of LSTM architecture is its inability to parallelize computations effectively. The sequential nature of LSTM computations makes it difficult to take full advantage of parallel processing capabilities, which can slow down the training and inference processes, especially for large-scale models and datasets.

One popular approach to address the bottleneck problem is the incorporation of <u>attention mechanisms</u>. However, despite this improvement, the issue of parallelization still persists. To overcome the parallelization issue, alternative architectures have been developed, such as the <u>Transformer</u> model, which is built upon the foundation of attention mechanisms.

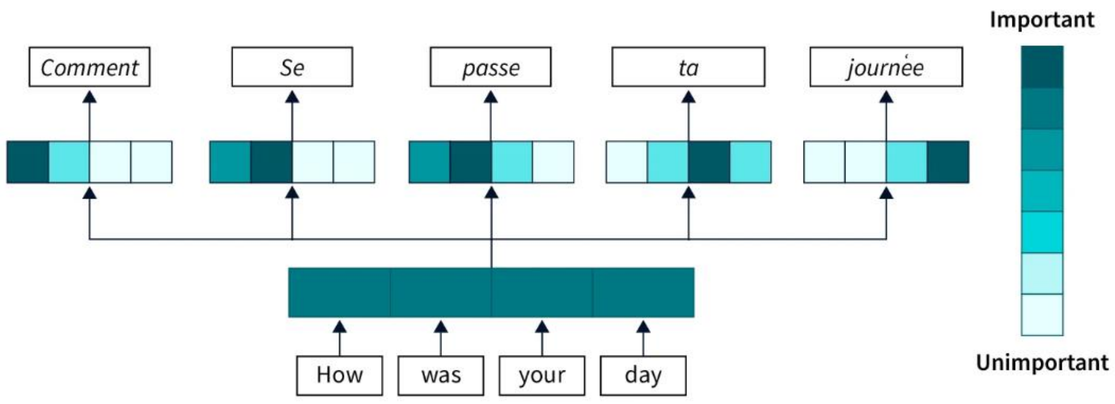
We will cover these two concepts of Transformers and attention mechanism in the following section.

V. Transformers:

a. <u>Attention mechanism:</u>

An attention mechanism is a crucial component in many machine learning models, particularly in natural language processing. It enables models to focus on specific parts of the input data by assigning varying weights or importance. It is commonly employed in sequence-to-sequence models, such as recurrent neural networks (RNNs).

The attention mechanism calculates attention weights for each element in the input sequence, indicating their relative importance. These attention weights are then used as inputs instead of the output of the last hidden layer in the RNN. By doing this, the vector can capture all the sequence



information, representing the focused parts. This context vector, combined with other information, helps generate the output or make predictions.

In the context of the attention mechanism, RNNs are used to process the input sequence and generate a sequence of hidden states. These hidden states capture the contextual information at each step of the input sequence. The attention mechanism then calculates attention weights based on these hidden states, indicating the relative importance of each input element. These attention weights are used to create a weighted context vector that represents the focused parts of the input sequence. This context vector, along with other information, is then used to generate the output or make predictions. By utilizing RNNs within the attention mechanism, the model can effectively capture the sequential dependencies and contextual information present in the input sequence.

Attention mechanisms improve model performance by capturing all the relevant information in the input sequence, thereby leading to better results in tasks such as machine translation, text summarization, and sentiment analysis. One limitation of this mechanism is that it cannot be parallelized efficiently due to the recurrent neural network.

"This approach continues to have an important limitation; each sequence must be treated one element at a time. Both the encoder and the decoder have to wait till the completion of t-1 steps to process thet-th step. So, when dealing with huge corpus it is very time consuming and computationally inefficient."
-Eduardo Muñoz- Towards Data Science

b. What's Transformer?

The transformer is currently regarded as the most powerful deep learning model architecture and has gained widespread usage in the field of artificial intelligence, particularly in natural language processing (NLP) tasks. It was introduced in a groundbreaking paper titled "Attention Is All You Need," published by Vaswani et al. in 2017.

The transformer architecture revolutionized NLP by addressing the limitations of previous recurrent neural network (RNN) and convolutional neural network (CNN) models. It achieves this by relying on a mechanism called self-attention, which allows the model to weigh the importance of different words or tokens in a sequence when processing each individual word.

The transformer model consists of an encoder and a decoder. The encoder processes the input sequence, such as a sentence, and extracts its contextual representation, while the decoder generates an output sequence based on the encoder's representation and an attention mechanism. Each encoder and decoder layer in the transformer contains multiple self-attention heads and feed-forward neural networks, enabling the model to capture complex dependencies and long-range dependencies in the input sequence.

One of the key advantages of the transformer architecture is its ability to parallelize computations, making it more efficient to train and allowing for faster inference times. This parallelization is possible because the self-attention mechanism allows the model to process all input tokens in parallel, unlike the sequential nature of RNNs.

Transformers have been successfully applied to a wide range of NLP tasks, such as machine translation, text summarization, sentiment analysis, question answering, and language generation. Moreover, the transformer architecture has also been adapted to other domains, including computer vision (e.g., vision transformers), audio processing, reinforcement learning and others.

The Transformer model has significantly advanced the state-of-the-art in NLP and has become a fundamental building block in many AI systems due to its ability to capture complex dependencies and improve performance on various tasks.

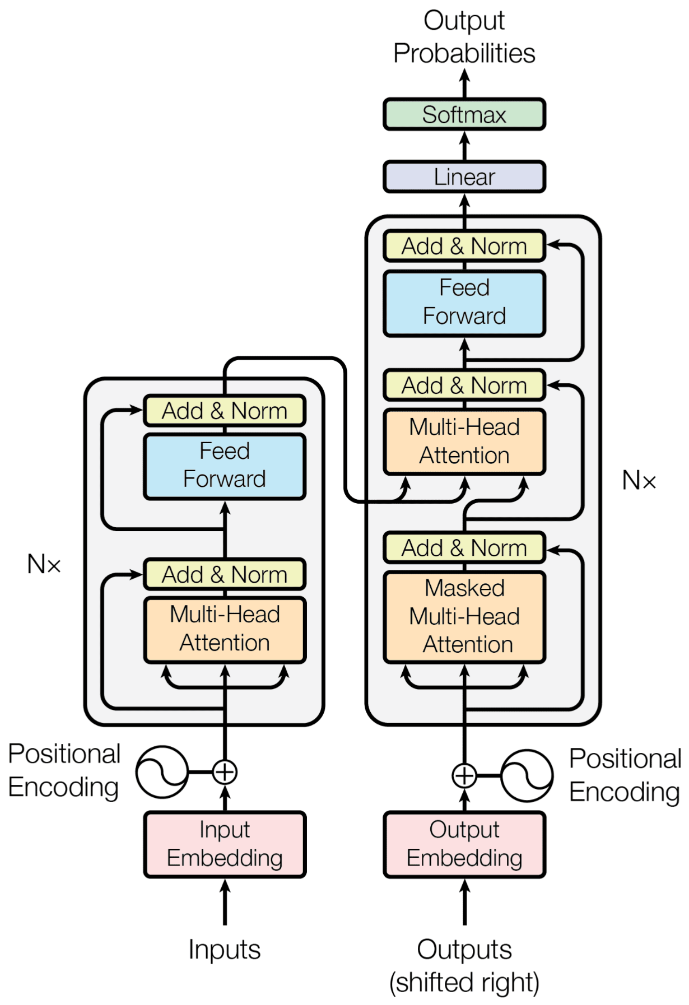


Figure 04: The Transformer architecture

We will describe the components of this model and analyze their operations in the following papers.

c. <u>Self-Attention mechanism:</u>

The core element in transformers is the self-attention mechanism, which empowers the model to concentrate on relevant parts of the input by calculating weights for each position in the sequence and considering the relationships between all positions.

In natural language processing (NLP), the input consists of vectors that represent each word in a sequence. Self-attention operates on these vectors and computes a weighted sum of the input vectors. This computation is achieved by utilizing three trainable vectors known as Keys, Queries, and Values.

d. <u>Key and Query and Value:</u>

"Keys," "queries," and "values" describe the three linear transformations that occur during the calculation of attention weights. These transformations are typically applied to input sequences or feature maps to capture relationships and compute the importance of different elements within the sequence.

Here's a breakdown of each component:

1- Keys: are used to encode information about the elements in the input sequence. They are derived from the input sequence by applying a linear transformation to it. The purpose of keys is to represent the elements in a way that captures their characteristics and allows the model to learn their relationships with other elements in the sequence.
2- Queries: Queries are derived from the input sequence as well, but they are used to obtain the attention weights by comparing them with the keys. Queries, similar to keys, undergo a linear transformation to represent the elements in a suitable format for comparison.
3- Values: Values are also derived from the input sequence, and they contain the actual information or features associated with each element. They are transformed linearly to obtain the value representations.

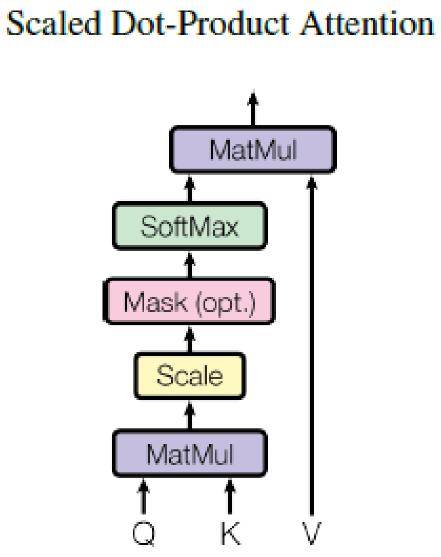
Once the keys, queries, and values are obtained, the attention mechanism calculates the attention weights by measuring the similarity between the queries and keys. This similarity score is used to determine the importance or relevance of each element in the sequence to the others. Finally, the attention weights are used to weigh the corresponding values, generating the context vector or weighted sum of values, which represents the output of the self-attention mechanism.

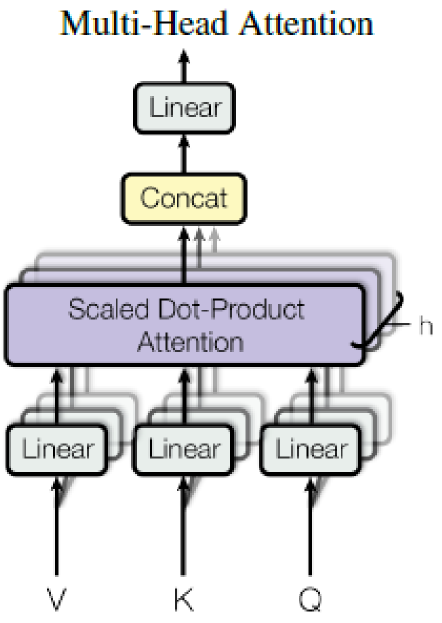
Mathematically, one head of self-attention mechanism computes its outputs by first transforming the input sequence into three matrices: the key matrix (K), the query matrix (Q), and the value matrix (V). These transformations are obtained by multiplying the input sequence by learnable weight matrices. The attention weights are then calculated by taking the dot product between each query vector and the corresponding key vectors, scaled by a factor of the square root of the dimensionality of the key vectors (in order to have a stable gradient). This similarity score is further passed through a SoftMax function to obtain normalized attention weights. The attention weights are then multiplied element-wise with the value vectors to obtain the weighted value vectors. Finally, these weighted value vectors are summed together to form the output of the self-attention mechanism, which represents the context vector capturing the important relationships between the elements in the input sequence.

e. <u>Multi-head self-attention mechanism:</u>

Multi-head attention is an extension of the self-attention mechanism that introduces multiple parallel self-attention layers, each called a "head." Each head has its own set of learnable weight matrices for keys, queries, and values, enabling it to attend to different information and capture diverse patterns in the input sequence.

After obtaining the outputs from the multi-head attention mechanism, concatenating them and feeding the concatenated outputs into a feed-forward neural network helps capture complex relationships and dependencies among elements in the input sequence. This process enhances the representation and expressive power of the self-attention mechanism.





f. Encoder:

The Transformer encoder is responsible for processing the input sequence. It consists of multiple layers, each containing two sub-components: multi-head self-attention mechanism and position-wise feed-forward neural networks (FFNNs). The input sequence is transformed into a sequence of context-aware representations through these layers. Each layer in the encoder attends to the previous layer's representations, allowing the model to capture dependencies and relationships within the input sequence effectively.

g. Decoder:

The Transformer decoder, on the other hand, generates the output sequence based on the encoded representations. It also consists of multiple layers, each containing three sub-components: masked multi-head self-attention mechanism, encoder-decoder attention mechanism, and position-wise FFNNs. The decoder attends to both the encoded representations from the encoder and the previously generated tokens in an autoregressive manner to generate the next token in the output sequence.

The encoder-decoder attention mechanism allows the decoder to focus on relevant parts of the input sequence while generating the output. The masked self-attention mechanism ensures that the decoder attends only to the previously generated tokens and prevents it from attending to future tokens during the generation process.

There are other variations of the Transformer architecture that utilize either only an encoder or a decoder, depending on the specific task.

h. Methods for solving NLP problems:

Methods used for solving NLP problems involve transfer learning from a pre-trained model, followed by fine-tuning it for a specific task. The most commonly utilized pre-trained models for specific tasks are as follows:

- GPT: A large language model developed by Open AI, trained on a vast corpus of text. It is a decoder-only model designed for predicting the next word given a context from the corpus.

- BERT: Another significant language model developed by Google, which is an encoder-only model.

- T5: A transformer model commonly used for various NLP tasks.

There are many other open-sourced models available on different platforms, such as Hugging Face. Training these Language Models (LLMs) from scratch or fine-tuning them requires significant computational resources, large datasets, and time for training. Each model is designed for specific tasks and can be further fine-tuned for similar tasks

The second method is designing a Transformer from scratch for a specific task and this method is very recommended in case of the availability of data and computational resources.

Bonus:

The shape calculations involved in self-attention are as follows:

- Input matrix X: (L x d)
- Query matrix Q: (L x d)
- Key matrix K: (L x d)
- Value matrix V: (L x d)
- Dot Product: (L x L)
- Scaled Dot Product: (L x L)
- Attention Weights: (L x L)
- Attention Output: (L x d)

Where: "L" is the length of the sentence, and "d" is the embedding vector size.