TRANSFORMER MODEL

The transformer model is a deep learning architecture designed to process sequential data more efficiently than traditional models like RNNs. It has become the foundation for many advanced AI models. The key feature of transformers is the self-attention mechanism, which allows the model to weigh the importance of different words in a sequence, regardless of their position. This enables it to handle long-range dependencies more effectively. Unlike RNNs, transformers process entire sequences in parallel, making them faster and more scalable. They are composed of encoders, which analyse input data, and decoders, which generate outputs. Multi-head attention enhances the model's ability to focus on different parts of the input simultaneously, while positional encoding helps it understand word order. Transformers are widely used in applications like natural language processing, machine translation, text generation, and even image processing.

The most important part is to understand the structure of the model, the way it works and it's architecture. As per the "Attention is all you need "paper the various parts which constitute the model of a transformer contains the following: -

1. Encoder and decoder stacks

2.Attention – It has two different parts--- *Self attention* and *Multi head -attention*

3. Position-wise Feed-Forward Networks

4. Embeddings and SoftMax

5. Positional Encoding

Working of a Transformer

To build an effective transformer, proper training is crucial. First, the input is broken into tokens (t**okenization**). Each token is then mapped to a unique vector in a higher-dimensional space (**embedding**). These embeddings, represented as numerical coordinates, may encode token meanings. These encoded vectors help the model understand and process language. We have already learnt about it while reading about tokenization and embeddings for words.

After this the first step is to do **positional encoding** , which basically gives the model an idea and significance of the order of the words they are being used and how it's important while performing certain operations later on the test data. Positional encoding helps transformers understand the order of tokens in a sequence by adding unique position-based values to embeddings. This ensures the model captures word order, which is crucial for meaning in language tasks.

Now, the most important thing is to use **Attention** blocks. An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key. To basically find out how the tokens relate to each other how effectively and how remaining words depend on the other ones, attention is all we need. To compute this , we basically use dot product method. We call this particular attention *Scaled Dot-Product Attention*. We compute the dot products of the query with all keys, divide each by $\sqrt{dk}$, and apply a softmax function to obtain the weights on the values. In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix Q. The keys and values are also packed together into matrices K and V . We compute the matrix of outputs as:

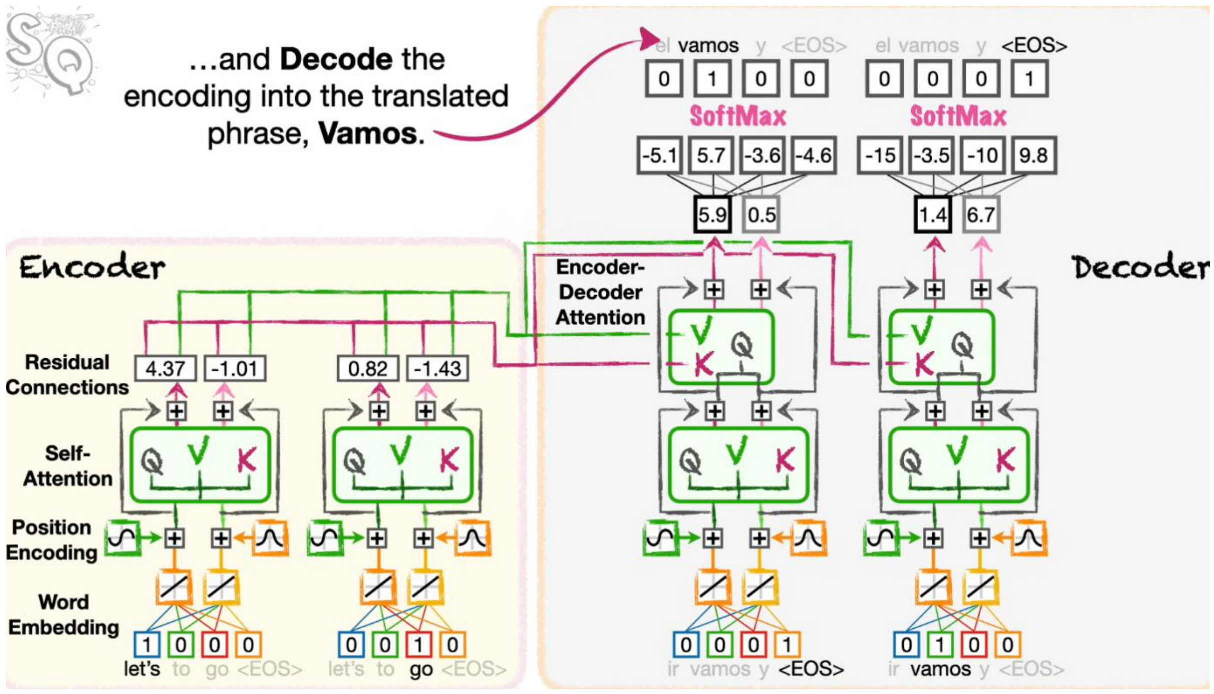
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

We use a scaling factor here . After this, In **multi-head attention**, the input embeddings first go through multiple self-attention mechanisms in parallel. Each head independently transforms the input using different sets of query (Q), key (K), and value (V) matrices, allowing the model to capture various aspects of word relationships—like syntax, meaning, and long-range dependencies.Each attention head performs the scaled dot-product attention, computing attention scores to determine which words are important for each token. These individual outputs are then merged and passed through a linear projection layer, which combines the information from all heads into a single unified representation.

Following the Attention Block, these vectors then pass through a Multilayer Perceptron, or **Feed-Forward Layer**. Here, the vectors don't talk to each other; they all go through the same operation in parallel. Besides the attention sub-layers, each layer in the encoder and decoder has a fully connected feed-forward network. This network is applied to each position independently in the same way. It consists of two linear transformations with a ReLU activation in between. the linear transformations are the same across different positions, they use different parameters from layer to layer. After that, the vectors pass through another attention block, then another multilayer perceptron block, then another attention block, and so on, getting altered by many variants of these two operations interlaced with one another. After many iterations, all the information necessary to predict the next word needs to be encoded into the last vector of the sequence, which will go through one final computation to produce a probability distribution over all the possible chunks of text that might come next.

Now, coming to **Encoder-Decoder stacks,** the encoder and decoder in a transformer both consist of six identical layers.

- Encoder: Each layer has two sub-layers—multi-head self-attention and a feed-forward network. It uses residual connections and layer normalization to stabilize training. All outputs have a fixed dimension of 512.
- Decoder: Similar to the encoder but adds a third sub-layer for multi-head attention over the encoder's output. It also uses masking to prevent positions from attending to future tokens, ensuring predictions rely only on past information.



The above image is an example of how encoder-decoder stacks look like. In the above one, the main purpose of the transformer is to convert English to Spanish. One other thing which plays a crucial part in the overall performance and prediction of words,etc is **Embedding and SoftMax**.

Like other sequence transduction models, learned embeddings are used to convert input and output tokens into vectors of dimension d-model. To generate predictions, a linear transformation is applied followed by a softmax function, which outputs probabilities for the next token. The model shares the same weight matrix between the embedding layers and the pre-softmax transformation, following a similar approach to previous work. Additionally, in the embedding layers, these weights are scaled by the square root of dmodel for better numerical stability. The softmax function is crucial as it converts raw model scores into probability distributions, ensuring the probabilities sum to one. This helps the model make confident and interpretable predictions for the next token.

Training

Training plays a very crucial role in the working of a transformer. Training is important for a transformer because it helps the model learn patterns and meanings from data. Without training, it would have random values and not work properly. During training, the model improves by adjusting its settings using large amounts of text, learning how to process words, focus on important parts, and make better guesses. This helps it understand language and work well with new sentences. Methods like adjusting the learning speed and preventing overfitting make training more effective. In short, training turns a basic transformer into a smart model that can handle complex language tasks. Thus . it's very significant . As per the *Attention is all you need* paper , the transformer they proposed to use was actively trained on enormous datasets . The model was trained on large datasets using byte-pair encoding for tokenization. Training was done in batches of about 25,000 tokens per language. The base model trained for 12 hours and a larger version for 3.5 days. The Adam optimizer was used with a dynamic learning rate that increased at first and then gradually decreased. To improve performance, regularization techniques like dropout and label smoothing were applied. Dropout helped prevent overfitting by randomly ignoring parts of the model during training, while label smoothing made predictions less confident but improved overall accuracy and BLEU scores.