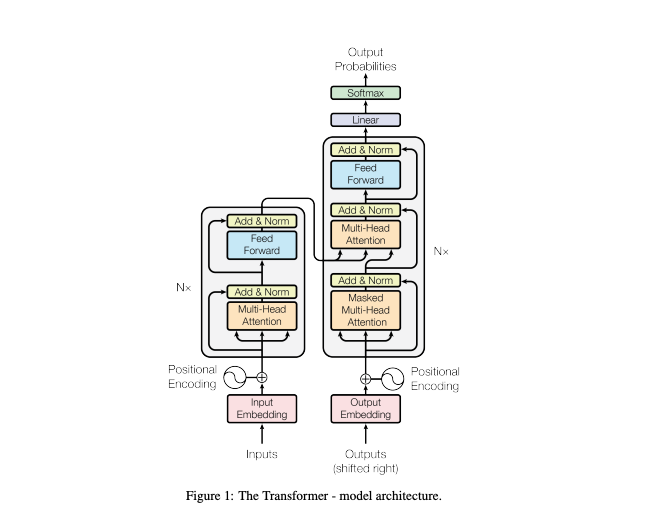
* What is transformer
* Attention
* Common archs
* Bert, GPT, VIT
* Fine tuning them
* RLHF
* Instruction-gpt

Why RNN’s failed

1. Long sentences requires long range dependencies
   1. Transformer with attention (query, key, value) helps
2. They suffer from vanishing gradient problem
   1. Parallel training, few FCLs help transformers to train
3. Larger training steps
4. Parallel computation
5. References
   1. <https://towardsdatascience.com/all-you-need-to-know-about-attention-and-transformers-in-depth-understanding-part-1-552f0b41d021>
   2. https://daleonai.com/transformers-explained

Transformers: different architectures for transformers model is below

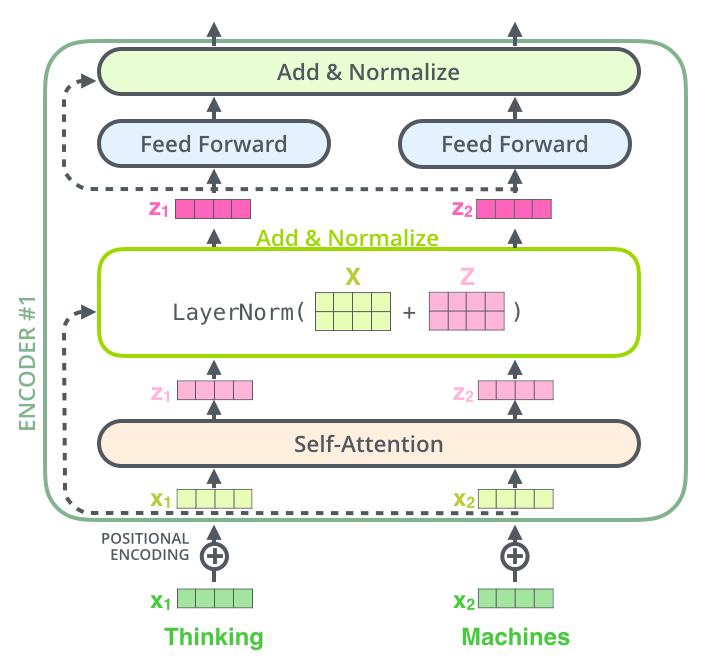


1. Encoder and decoder stack
2. Attention
3. Scaled dot product
4. Multi head attention
5. Point wise feedforward network
6. Embedding and softmax
7. Positional encoding

Autoregressive : Consuming the previously generated input symbols as additional input

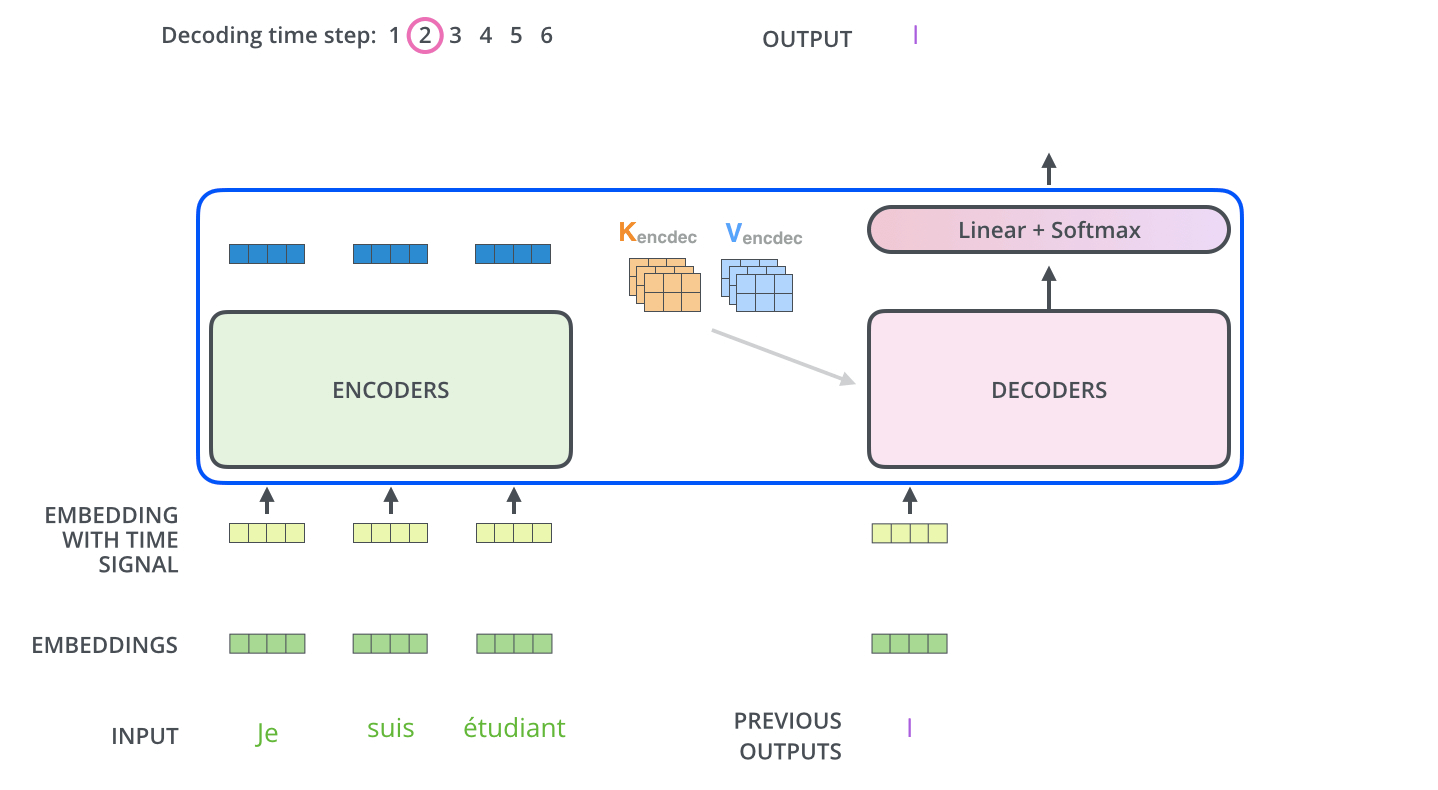
Encoder : Components of the encoder , there are 6

1. Multi head attention
2. Position wise feed forward network,
3. Residual connection → output of the previous later to next layers
4. Layer normalization



Decoder

1. Multi head attention
2. Position wise feed forward network,
3. Residual connection →
4. Layer normalization
5. The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence
6. Thismasking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.

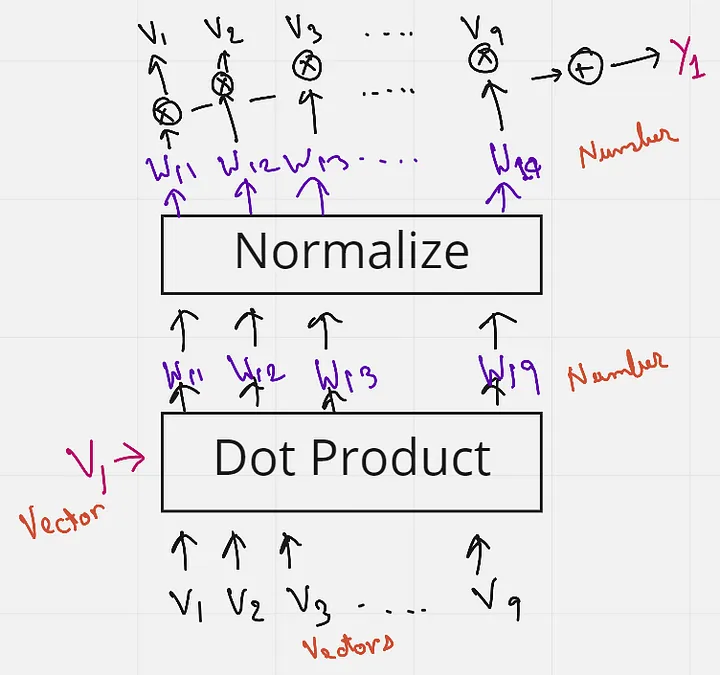
****

**The “Encoder-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.**

Attention :

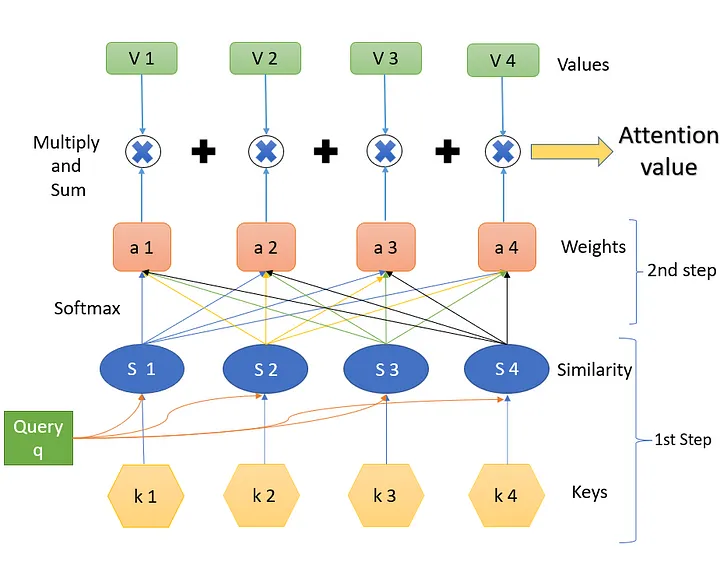
An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

Input is a token, then token is converted to embeddings, these embeddings themselves don't have context. One way to have context is to multiply each embedding with other embeddings, then they are summoned and normalized to give a value . This approach of adding some context to the words in a sentence is known as Self-Attention. (<https://towardsdatascience.com/all-you-need-to-know-about-attention-and-transformers-in-depth-understanding-part-1-552f0b41d021>)



Query, Key, and Values

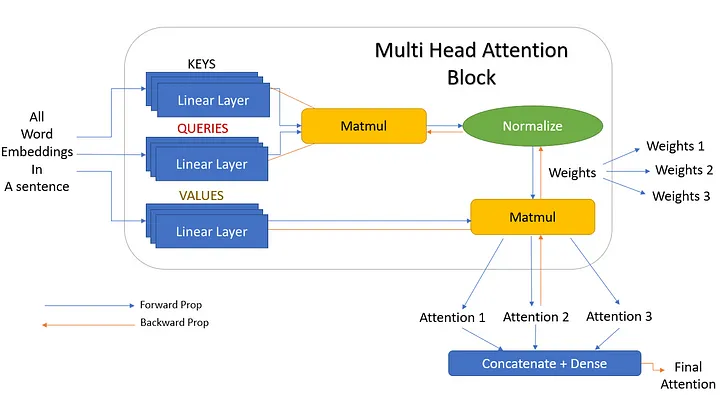
1. There is nothing trained in the self attention
2. They are used 3 times, as
   1. as a dot product between the first word embedding and all other words
   2. in the sentence to obtain the weight
   3. to the weights, to obtain the final embedding with context
3. Query - V1 as query
4. This query word will then do a dot product with all the words in the sentence (V1 to V9) — and these are the Keys
5. So the combination of the Query and the Keys give us the weights. These weights are then multiplied with all the words again (V1 to V9) which act as Values
6. Attention is also a retrieval process, The attention mechanism measures the similarity between the query q and each key-value ki. This similarity returns a weight for each key value. Finally, it produces an output that is the weighted combination of all the values in our database. The only difference between database retrieval and attention in a sense is that in database retrieval we only get one value as input, but here we get a weighted combination of values. In the attention mechanism, if a query is most similar to say, key 1 and key 4, then both these keys will get the most weights, and the output will be a combination of value 1 and value 4.



Second understanding : Self attention is a way to uses to bake the “understanding” of other relevant words into the one we’re currently processing.

1. Create three vectors from each of the encoder’s input vectors (in this case, the embedding of each word). So for each word, we create a Query vector, a Key vector, and a Value vector. These vectors are created by multiplying the embedding by three matrices that we trained during the training process.
2. Score : We need to score each word of the input sentence against this word. The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.
3. Stable gradient : divide by the square root of the length of the key
4. Softmax : Softmax normalizes the scores so they’re all positive and add up to 1.
5. Focus word : multiply the value vector with softmax, is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words
6. sum up the weighted value vectors. This produces the output of the self-attention layer at this position (for the first word).
7. Matrix Multiplication
   1. is to calculate the Query, Key, and Value matrices. We do that by packing our embeddings into a matrix X, and multiplying it by the weight matrices we’ve trained (WQ, WK, WV).
   2. Finally, since we’re dealing with matrices, we can condense steps two through six in one formula to calculate the outputs of the self-attention layer.
   3. Z = softmax(Q\*KT/sqrt(dk)))V

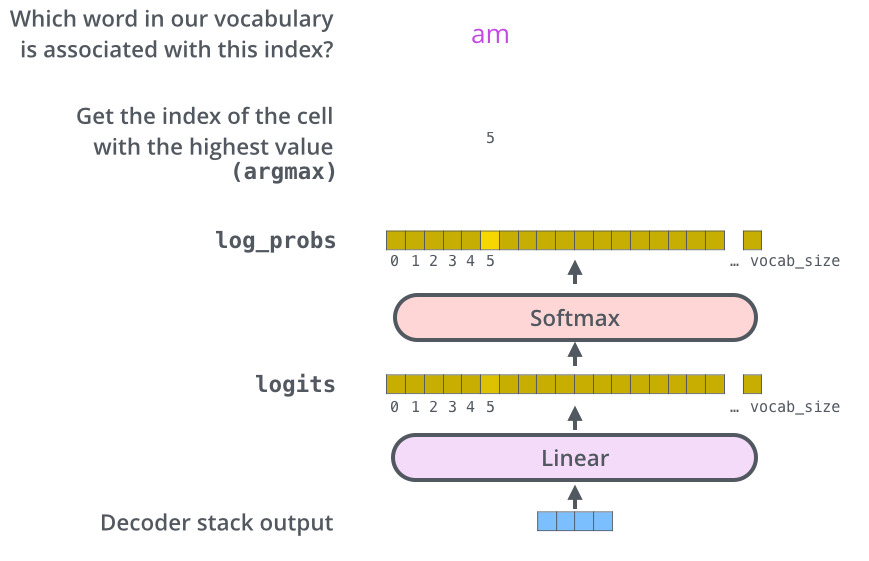
Multihead attention

1. It expands the model’s ability to focus on different positions.
2. Different Attention layer : he attention layer multiple “representation subspaces”.
3. Trained independently, These three attention blocks are finally concatenated to give one final attention output. 

## **Positional Encoding**

Input embedding has positional encoding, they use wavelets (sin/cosine) you can use inputs

## **The Final Linear and Softmax Layer**



1. 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.
2. 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.
3. 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.