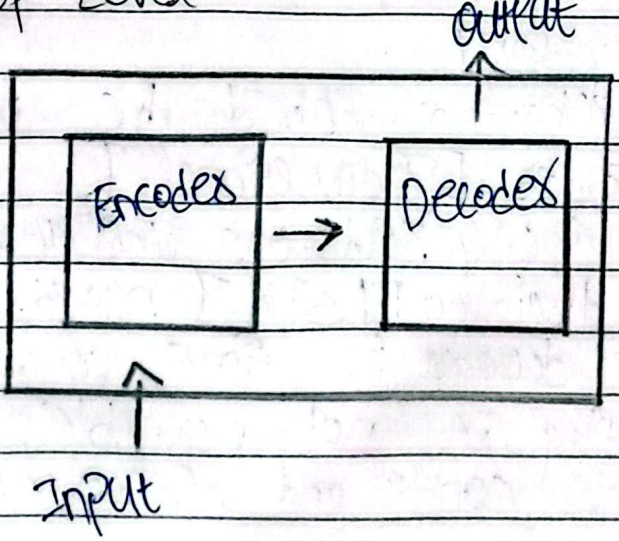
Date
Page

(*) Current improvements seeker for Transformers.

- Pouring
- Quantization
- Knowledge Distillation.
- Improve the multimodel capabilities
- Multi lingual capabilities.

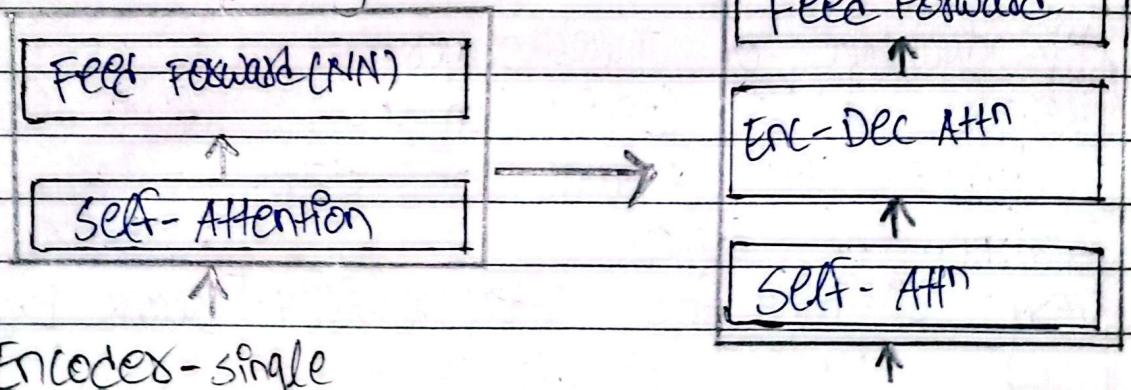
Transformer Architecture

(*) Top Level.



- Encoder
  → contains multiple Encoders.
- Decoder
  → contains multiple Decoders.

(*) Bit Detail (Fig)



- In the each [Encoder], takes the sequence, in parallel as input. Self Attn layer processes those sequence, this self Attn layer converts the [Raw vector] (embedding) to a different vector known as [Contextual vector]

- Contextual vector [Contextual Embedding] vector that keeps the context of a word or token relative to other token, or word.

- the generated [Contextual vector] will be passed as an input the the another Encoder in the stack.

- Self Attention layer
  - Basically it's just a function that takes the Raw Embedding [static] and generate Dynamic Contextual Embedding.
    - ↳ If we just use static word embedding we don't have the power powes to change the embeddings based on the content [Refer to apple example].
  - Self Attention provides the content of each word relative to other words in a sentence. This is what we call Attention, now we can know which word has the maximum impact for understanding the [meaning] of the sentence

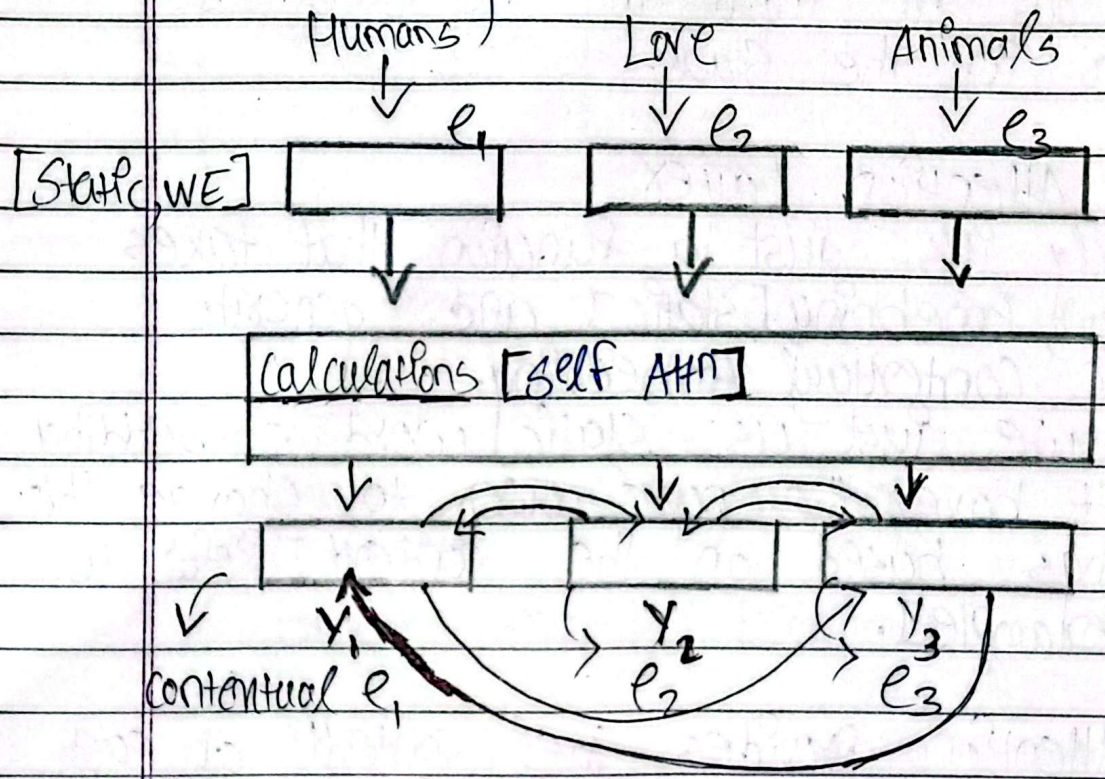
Bank Here, has the same [word Embedding] but, self Attn creates dynamic [contextual Embedding]

e.g [Rives Bank], [Money Bank]

Here, the word bank can be confusing so it is very important we need to build a model that captures or takes in consideration to it's surrounding context to make or understand the actual meaning of the sentence.

↳ Self Attention is what we're looking for.

(*) Understanding self Attention.



Let's take an example:

=> Two sentences

S_1 => [Money Bank Grows] S_2 => [Rives Bank Flows]

Here, for static word Embedding for Bank is same.

If we use static word embedding we'll not get our desired output because both the embeddings are same.

But what if we could represent each word relative to it's importance of other word.

So, for S_1

Money => 0.7 Money + 0.2 Bank + 0.1 grows
Bank => 0.25 Money + 0.7 Bank + 0.05 grows
Grows => 0.1 Money + 0.2 Bank + 0.7 Grows

and, S_2

River => 0.8 River + 0.15 Bank + 0.05 Flows
Bank => 0.2 River + 0.78 Bank + 0.02 Flows
Flows => 0.4 River + 0.01 Bank + 0.59 Flows

This way we're representing each word in the form (Relative) to the importance of other word.

And, if we compare two [Bank] word. We've different Embeddings in each sentence.

Now, in both the sentence the word [Bank] will have new embedding (that covers the content or takes in consideration of other words as well).

- The numbers used here, such as 0.8, 0.2 etc are represents the similarity between words.

We can write,

New Embedding For [Bank] as:

$$e_{bank}^{new} = [e_{bank} \cdot e^T_{money}]\, e_{money} + [e_{bank} \cdot e^T_{bank}]\, e_{bank} + [e_{bank} \cdot e^T_{grows}]\, e_{grows}$$

- Here the score or number we used before is Dot Product i.e Represents Similarity.

- The [Scalar] represents the similarity for two words.

↳ All the Dot Product are scaled i.e [Normalized] i.e [0 to 1] which also represents the probability.

- Softmax is used to calculate the probability for each [Dot Product.]

$$f(x) = \left[\frac{e^{x_i}}{\sum_{i=0}^{n} e^{x_i}} \right] \Rightarrow \text{Softmax}$$

↳ Once we get this probability we'll multiply with the Raw or original vector.

↳ Then sum the each multiplication to represent the vector (New Embedding)

Problem with simple self Attn

- As our self-Attention layer remains static i.e there's no any trainable parameter it will have no idea about how to adapt the Attention for each word or sequence.
  ↳ so it fails to adapt to the specific patterns or nuances present in the [Training] data required for a specific task i.e "sentiment", "Translation".

To fix this,
we introduce [Query, key, and value]

- For each word or embedding [3] new vectors i.e [Query vector,][key vector] and [value vector].

so, for embedding [Bank] we'll have

~~Q_{bank}, K_{bank} = [Money, Bank, Grows], and~~
~~V_{bank} = [Money, Bank, Grows].~~

$$Q_{bank} = E_{bank} \times W_Q$$

$$Q_{money} = E_{money} \times W_Q \quad \text{so, } Q = \begin{bmatrix} Q_{bank} \\ Q_{money} \\ Q_{grow} \end{bmatrix}$$

$$Q_{grow} = E_{grow} \times W_Q$$

• If ous, input seq of 3 we'll have 3x3 vectors i.e 3 Query, 3 key, 3 value vectors

This [Q] matrix is collection of all our query in the sequence,

Then similarly, for [key]

$K_{bank} = E_{bank} \times W_K$

$K_{money} = E_{money} \times W_K$ so, $K = \begin{bmatrix} K_{bank} \\ K_{money} \\ K_{flows} \end{bmatrix}$

$K_{Flows} = E_{money} \times W_K$

Also, for [value]

$V_{bank} = E_{bank} \times W_K$
$V_{money} = E_{money} \times W_K$ so, $V = \begin{bmatrix} V_{bank} \\ V_{money} \\ V_{flow} \end{bmatrix}$
$V_{flows} = E_{flow} \times W_K$

Now, once we've all the 3 vectors i.e query, key, value.

We'll need to follow these steps to generate contextual Embedding:

(i) Calculate Dot Product betw query and key matrix for similarity chek.

i.e $S = Q \cdot K^T$ (3x3, 3x3) => 3x3

(ii) Apply softmax to the Dot Product output matrix (Normalized, same unit) (scaled Dot-Product Attention)

$$A = \begin{bmatrix} \text{softmax}(a_1, a_2, a_3) \\ \text{softmax}(b_1, b_2, b_3) \\ \text{softmax}(b_1, c_2, c_3) \end{bmatrix}$$

(iii) Compute the weighted sum i.e Final Attn score

$C = A \cdot V$ (softmax output times the value matrix)

Now, this C matrix is the [Contextual Embedding matrix] for all the words in a sequence.

This C will be the output of a single self-Attention layer.

Here, W_q, W_k, and W_v are the weights for the query, key and value.

- The W_q, W_k, and W_v weights will be changing throughout the training with Backpropagation.
- These weights are randomly initialized before training.