Transformers Notes By Soham

2024-06-05

Before we start I want to state that I have learnt about two kinds of transformer architectures , one is encoder and decoder and other being decoder only.

Both have their own use cases. Like with encoder+decoder you could do tasks such as Language translation while other can perform well with text generation based on a prompt.The difference between both is that the E+D model can learn the entire sequence and understand the contextual meaning while the Decoder only can just see the current word and the words used before to generate the next coming word, this architecture is used by all the LLMs. Why though? We would discuss about it in very detail by the end of the notes.

First lets start with the architecture and the chronology of how things work in E+D models.

To explain the complete process of the Transformer model from embedding to generating logits, I'll walk through each component in detail. We'll use an example sentence and delve into the mathematics involved at each step.

Example Sentence

Let's use the sentence: **"The cat sat on the mat."** We'll assume the task is to predict the next word in the sequence, and we'll go through the encoding and decoding steps.

1. Input Embedding

1.1 Tokenization

First, the input sentence is tokenized. Tokenization breaks down the sentence into individual tokens (words or subwords). Assume the tokenization yields:

Tokens: `["The", "cat", "sat", "on", "the", "mat", "."]`

1.2 Token Embeddings

Each token is converted into a vector using an embedding matrix W_e. The embedding matrix W_e is of size $V \times d_{\text{model}}$, where V is the vocabulary size and d_{model} is the embedding dimension.

Mathematically, for a token x_i, the embedding is:

$$E(x_i) = W_e[x_i]$$

If $d_{\text{model}} = 512$ and V is large enough to cover all words in the vocabulary, each token is mapped to a 512-dimensional vector. For example:

- $E(\text{"The"}) = W_e[\text{"The"}]$
- $E(\text{"cat"}) = W_e[\text{"cat"}]$
- and so on…

1.3 Positional Encoding

Since Transformers don't have a built-in sense of order like RNNs, they add positional encodings to the embeddings to capture the position of each token in the sequence.

The positional encoding PE for a position pos and dimension i is given by:

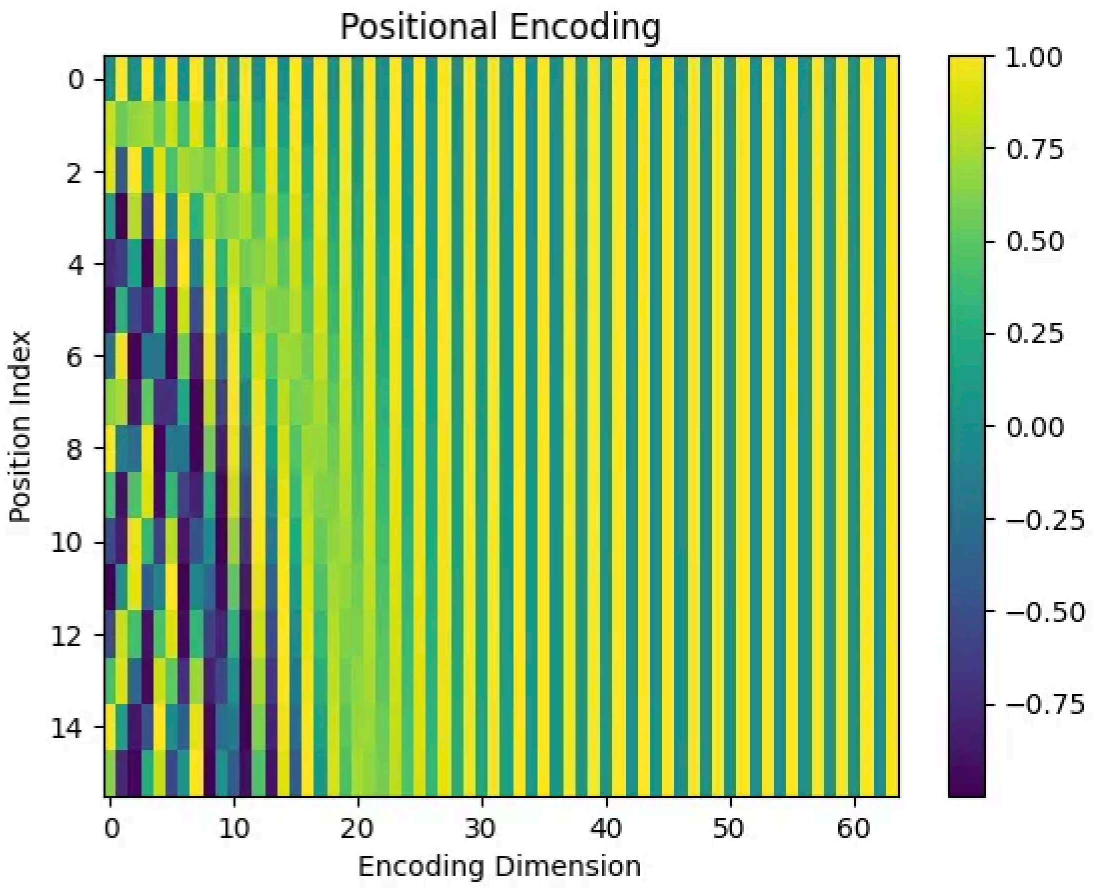
$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{\text{model}}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{\text{model}}})$$

For example, for the word "The" at position 1, the positional encoding would add certain sine and cosine values to the embedding vector.

Think of positional encoding as to lean the distance between two words as to understand their relation for example in the previous example if I say “A sat cat on a mat”, it would understand the sat as maybe the name of the cat cause of the closeness of two words so by applying this layer the machine learns to know their distance and hence helps in learning their contextual relation.

They values would be between -1 ans 1 as they are sine and cosine functions.



The output of input as well as output after positional encoding would be (number of batches, number of words in each batch, number of dimensions in each word) for the notes I would be taking (4,10,512)

1.4 Combined Embedding

The final input embedding for a token is the sum of the token embedding and the positional encoding:

$$z_i = E(x_i) + PE(pos_i)$$

2. Encoder

The encoder processes the input sequence embeddings through a series of identical layers, each consisting of two sub-layers: Multi-Head Self-Attention and Feed-Forward Neural Network.

2.1 Multi-Head Self-Attention

2.1.1 Calculation of Query, Key, and Value

“*Lets first understand the literal meaning of keys,*

In essence, keys are used to determine ‘how to pay attention’, i.e., they compute the compatibility score with the query, and values are used to determine ‘what to pay attention to’, i.e., they contribute to the final output based on the attention scores.

In a sentence, for each word, a query, a key, and a value are calculated. The query of a particular word interacts with all the keys (including its own) to compute attention scores, which are then used to weight the values.

For example, if we have the sentence “The cat sat on the mat”, when processing the word “sat”, the model would use the query associated with “sat” and calculate its dot product with the keys associated with “The”, “cat”, “sat”, “on”, “the”, and “mat”. This would result in attention scores that represent how much the model should focus on each of these words when *trying to understand the meaning of “sat”. These scores are then used to weight the value vectors associated with each word and sum them to produce the output.*

It’s important to note that the queries, keys, and values are not explicitly linked to specific words or meanings. They are learned during training, and the model determines how to best use them to complete the task it is trained on.”

For each input embedding z_i, three vectors are computed: Query (Q), Key (K), and Value (V). These are obtained through learned linear transformations:

$$Q_i = W_Q z_i$$

$$K_i = W_K z_i$$

$$V_i = W_V z_i$$

Where W_Q, W_K, W_V are weight matrices of size $d_{\text{model}} \times d_k$, $d_{\text{model}} \times d_k$, and $d_{\text{model}} \times d_v$, respectively.

Here the weights are randomised in the beginning and updated during the later part. the dimension of the vectors after the positional encoding would be (4,10,512) that are later divided into multiple heads I would take 8 so the dimensions that are 512 are divided between 8 heads so the single head would be having dimension of (4,10,512/8)=(4,10,64) we then process every batch parallely We can do matrix multiplication on each of the **4** X’s sub-matrix against Wq, Wk, Wv to get matrices Q,K and V matrices all of which would have dimensions of (10, dimension of weight matrices), in our notes lets take these weight matrices dimension to be (10,64) so the dimensions of Q K and . would also be the same/

2.1.2 Scaled Dot-Product Attention

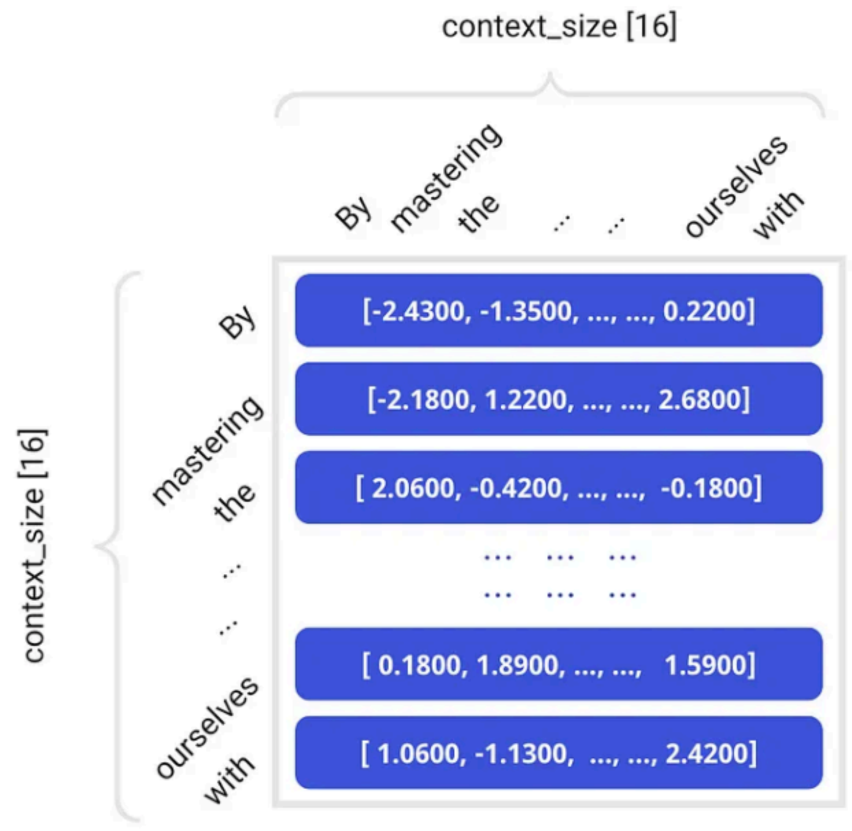
For each pair of tokens, the attention score is computed by taking the dot product of the Query vector from one token with the Key vector from another token, scaling by $\sqrt{d_k}$, and applying a softmax function:

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

This calculation results in a weighted sum of the Value vectors, where the weights are determined by the similarity between the Query and Key vectors.

Now as I said think Q as the value that signifies what the word wants from other words and the K signifies how ma current word deliver to other words so we want to combine q value of every word to k value of every word so to understand the strongest relationships, the highest value between 2 words shows the strongest bond and so we do the QKtranspose doint it would yield a matrix of dimension 10x10 where every word would be placed in horizontal(q) and vertical(k) having the relationship score as the values.

Example:



Remember our context length is 10,We would then divide the outputs with root(dimension of Q,K,V that is 64) just to reduce the large numbers this is the hyperparameter by the way so its not fixed you can change it accordingly. Now we would apply softmax to convert these numbers into probabilities and to normalise them between 0 to 1. Now the output is in the shape of 10x10, but I would need the output in shape of the input vectors that is (10,64) so lets multiply it with a vector V that carries values of bonds, So the probability of the softmax functions are then multiplied to V so that the highest bonds contributes the most and vice-versa . The output after multiplication also called ‘z’ would be in the shape of the input vector that is (10,64)

2.1.3 Multi-Head Mechanism

The self-attention process is repeated h times (with different sets of weight matrices for Q, K, V), allowing the model to focus on different parts of the sequence from different perspectives. The results are concatenated and linearly transformed:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W_O$$

Where W_O is a weight matrix of size $hd_v \times d_{\text{model}}$.

All the ‘z’ from different heads (8 in our case are concatenated) and multiplied by some weight Wo to produce the final output that would be feeded to NN. The multiple heads are used to determine different relationships within words for example in example “the cat was running on a mat” first head might associate the cat wit mat and identify the position while another might join cat and running to find the activity performed by a cat.

2.2 Feed-Forward Neural Network

The output from the multi-head attention is passed through a position-wise feed-forward network, which consists of two linear transformations with a ReLU activation in between:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

This helps in learning complex mappings and transformations.

The neural networks generally increases dimensions of the input just to study advance patterns and then again shrinking the output back in the same dimensions

2.3 Layer Normalization and Residual Connections

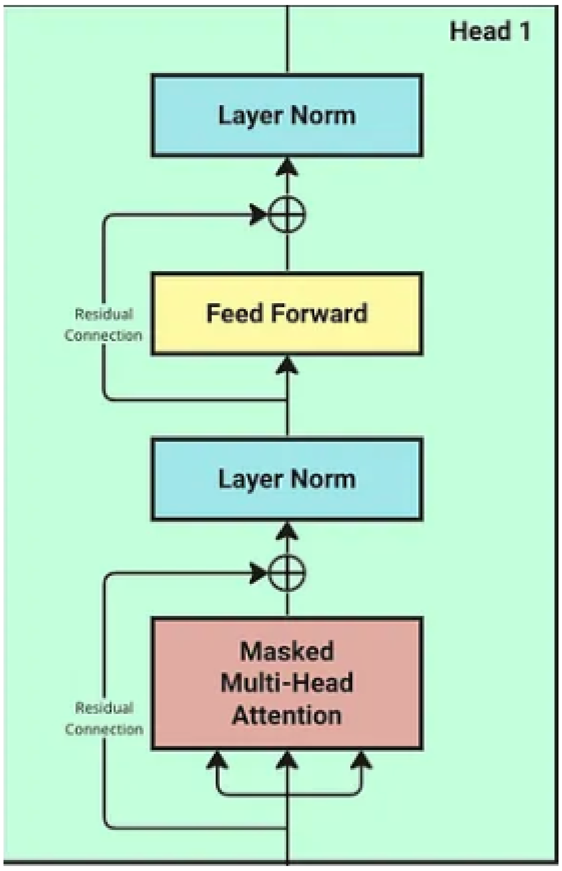
Each sub-layer (self-attention and feed-forward) in the encoder has a residual connection around it, followed by layer normalization:

$$\text{LayerNorm}(x + \text{Sublayer}(x))$$

A residual connection, sometimes referred as a skip connection, is a connection that let original input **X** bypasses one or more layers.

This is simply a summation of the original input **X** and the **output** of the multi-head attention layer. Since they are in same shape, adding them up is straightforward.

```
output = output + X
```



Ignore masked word cause have taken this photo from a decoder architecture rest all is the same.

After the residual connection, the process goes to layer normalization. LayerNorm is a technique that normalizes the outputs of each layer in the network. This is done by subtracting the mean and dividing by the standard deviation of the layer's outputs. This technique is used to prevent the outputs of a layer from becoming too large or too small, which can cause the network to become unstable.

3. Decoder

The decoder also consists of several identical layers, each with three sub-layers: Masked Multi-Head Self-Attention, Multi-Head Attention over the encoder's output, and a Feed-Forward Neural Network.

3.1 Masked Multi-Head Self-Attention

This sub-layer is similar to the encoder's self-attention, but with a mask applied to prevent attending to future positions. This ensures the model generates the output sequence in a step-by-step manner, attending only to past tokens.

3.2 Multi-Head Attention Over Encoder's Output

In addition to attending to previous decoder outputs, the decoder also attends to the encoder's outputs. This helps the decoder incorporate information from the entire input sequence when generating each token.

The attention mechanism here uses the encoder's output as the Keys and Values, and the decoder's masked self-attention output as the Queries:

$$\text{Attention}(\text{Q}_{\text{decoder}}, K_{\text{encoder}}, V_{\text{encoder}})$$

3.3 Feed-Forward Neural Network and Layer Normalization

Similar to the encoder, the decoder has a feed-forward network and uses residual connections and layer normalization.

4. Output Layer and Logits

The final output of the decoder goes through a linear transformation (fully connected layer) followed by a softmax function to produce the logits, which represent the probability distribution over the vocabulary for each position in the output sequence.

Think of it as the output is projected on a matrix of (Numbers of words in vocabulary , scores of those words) then these scores are converted to probabilities using a softmax function o predict the next word that is the word with the maximum probability

4.1 Linear Transformation

The decoder's output is projected onto the vocabulary size dimension:

$$\text{logits} = \text{DecoderOutput}W + b$$

Where W is a weight matrix and b is a bias vector, both learned during training. The shape of logits is $[\text{seq\\_len}, V]$, where V is the vocabulary size.

4.2 Softmax

To convert the logits into probabilities, a softmax function is applied:

$$P(\text{word} \mid \text{context}) = \frac{\exp(\text{logit})}{\sum_i \exp(\text{logit}_i)}$$

This gives the probability of each word in the vocabulary being the next word in the sequence.

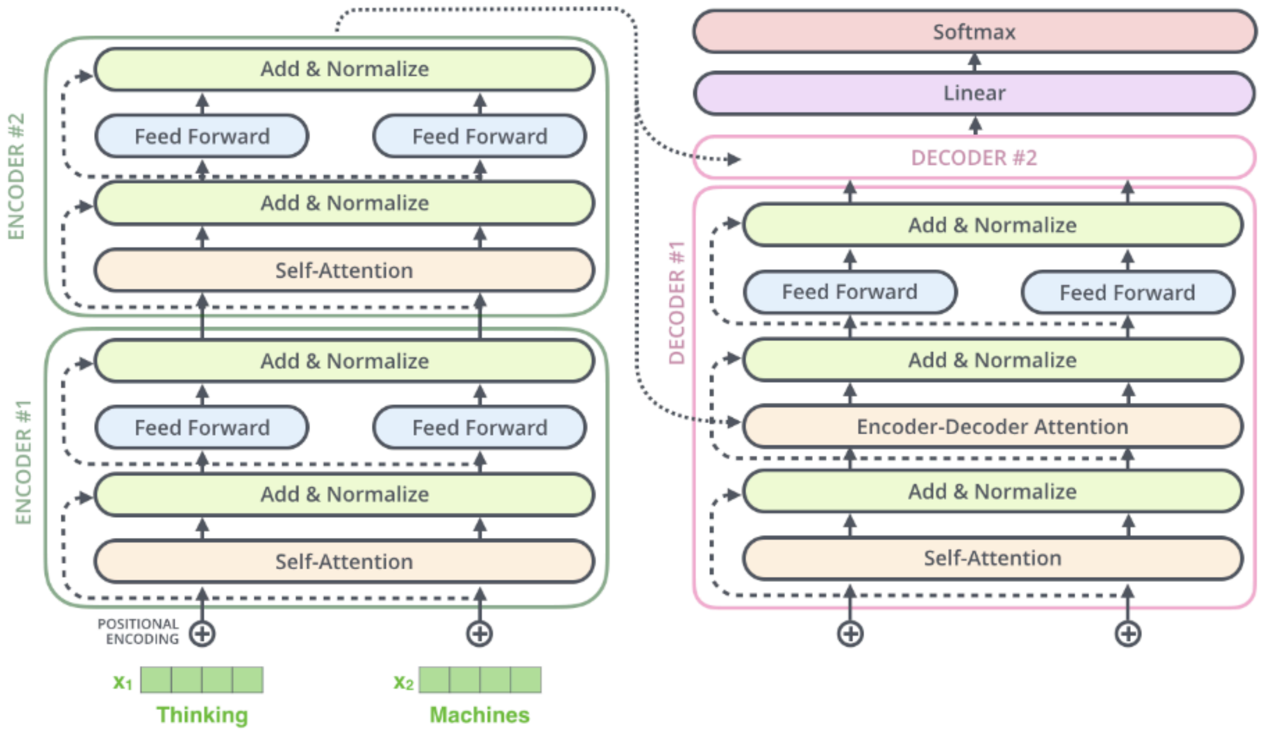
Conclusion and Example Continuation

For our example sentence "The cat sat on the mat," the Transformer would:

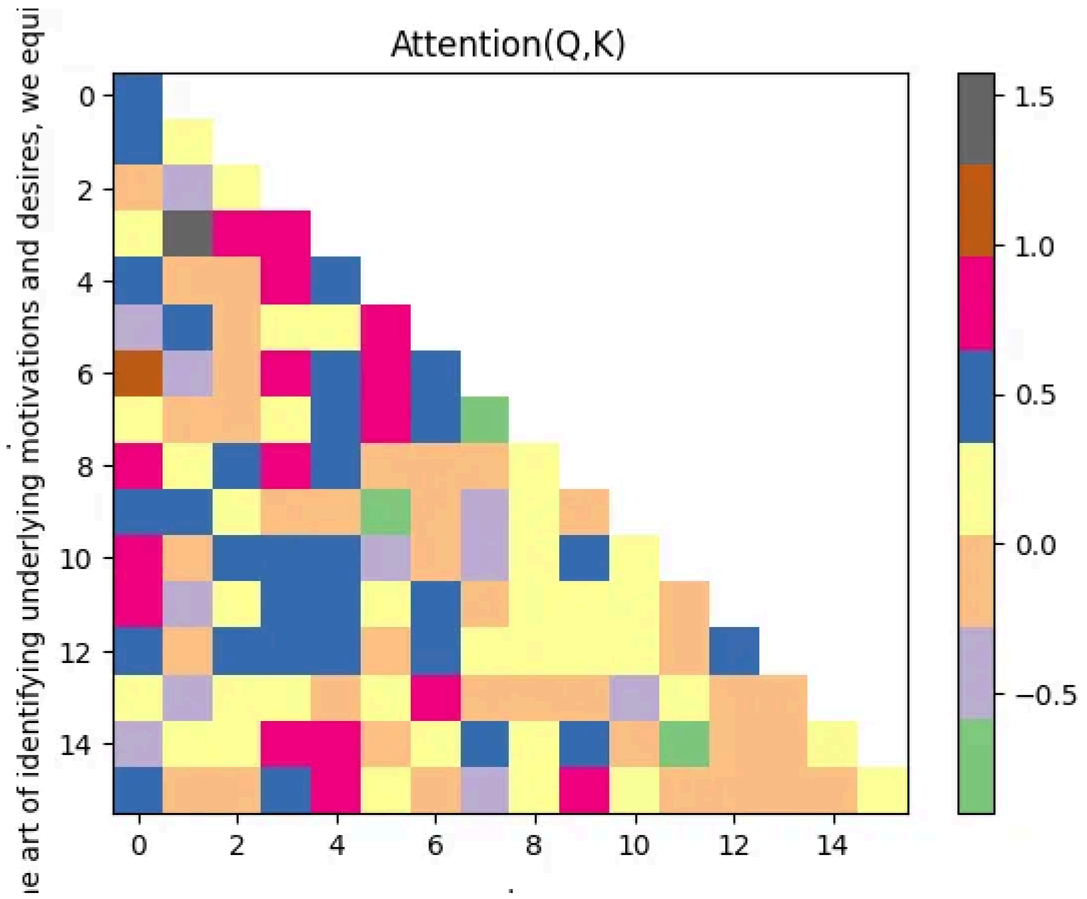
1. **Tokenize and embed** the input.
2. **Process through the encoder**, generating a comprehensive representation of the sentence.
3. **Pass through the decoder**, attending to the encoder's output and previously generated words to predict the next word.

For instance, after processing "The cat sat," the decoder would predict "on" as the most probable next word based on the encoded input and the previously generated words.

This detailed step-by-step process ensures that the model understands the input sequence's context and generates coherent and contextually appropriate outputs.



This is the complete diagram the encoder processes the entire prompt all at once and carry on all the steps. All the encooders are connected to each other such as the output of first is the input of second. It learns the entire sequence and then the output of the encoder is passed to every decoder that are sequentially connected too . The difference is that decoder only looks at the current and previous words so even the attention scores are masked.



The encoder is specialized in building a comprehensive understanding of the input sequence by considering the entire context (bidirectional attention). This capability is crucial for tasks where understanding the full structure and meaning of the input is essential. On the other hand, the decoder is designed to generate sequences based on this understanding and previous outputs, focusing on coherent and contextually relevant generation.

The decoder's self-attention is masked to prevent it from attending to future tokens during training and inference. This is because, during generation, the model should not "see" future words. For instance, if generating a sentence word by word, the model shouldn't use information about words that have not been generated yet.