Encoder and decoder layers

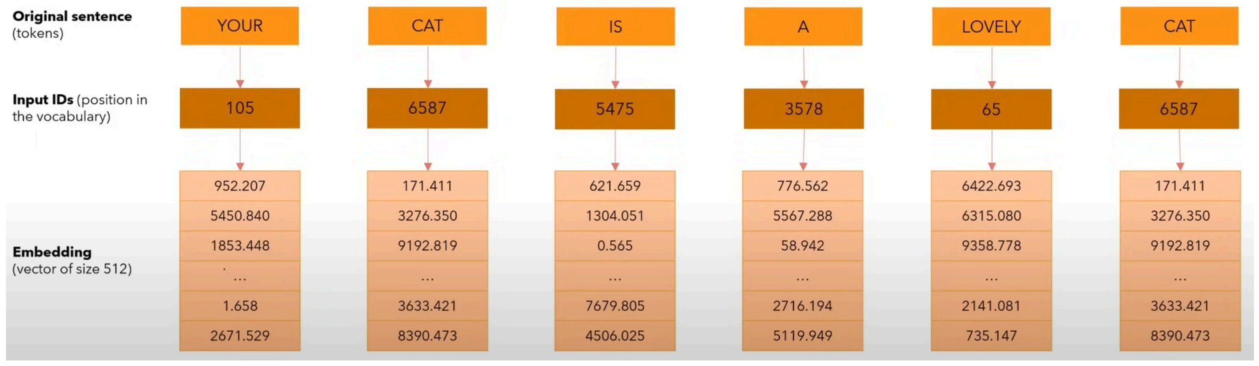
The encoder layer takes input sentences and then encodes them into contextual embeddings. Basically, it understands the input and passes it to the decoder block.

Then, the decoder layer takes the output of the encoder and tries to guess the next word, word by word (taking its own output as next input).

Both layers have similar structure, however with some differences. Encoder consists of a self-attention layer which is followed up by a forward feed network. The decoder consists of a masked self-attention layer (which doesn’t allow the model to cheat during training), cross-attention layer (that allows to connect encoder and decoder) and feed forward network. Between each layer there is normalization. Also before the data goes through positional and token embedding before it comes to encoder or decoder.

Input/output embedding

First, input is tokenized and then each token is converted to a vector embedding using a learned lookup table (like a big matrix of word vectors).
The output embedding converts the generated tokens into vectors as well.
After these embeddings, vector sequences are processed in positional embedding so that it also adds a vector to each token embedding to encode its position in the sentence.

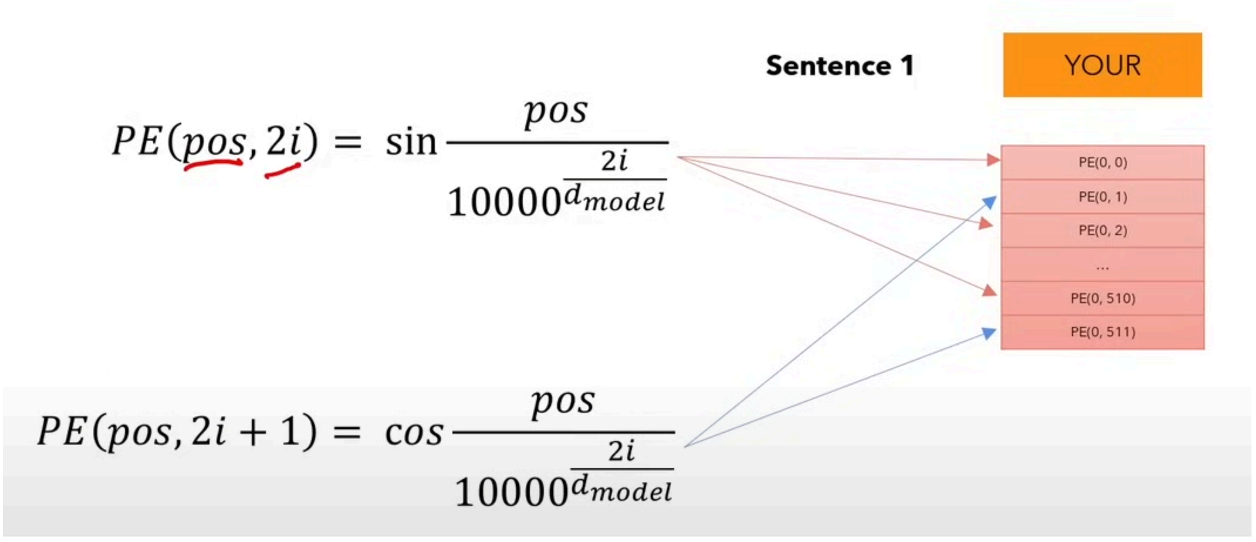


Original sentence (tokens)	YOUR	CAT	IS	A	LOVELY	CAT
Input IDs (position in the vocabulary)	105	6587	5475	3578	65	6587
Embedding (vector of size 512)	952.207	171.411	621.659	776.562	6422.693	171.411
	5450.840	3276.350	1304.051	5567.288	6315.080	3276.350
	1853.448	9192.819	0.565	58.942	9358.778	9192.819
	…	…	…	…	…	…
	1.658	3633.421	7679.805	2716.194	2141.081	3633.421
	2671.529	8390.473	4506.025	5119.949	735.147	8390.473

Positional embedding (positional encoding)

Transformers deal with inputs in parallel and don’t originally have a sense of order. That’s why every input token before proceeding further gets a unique vector that represents its position.

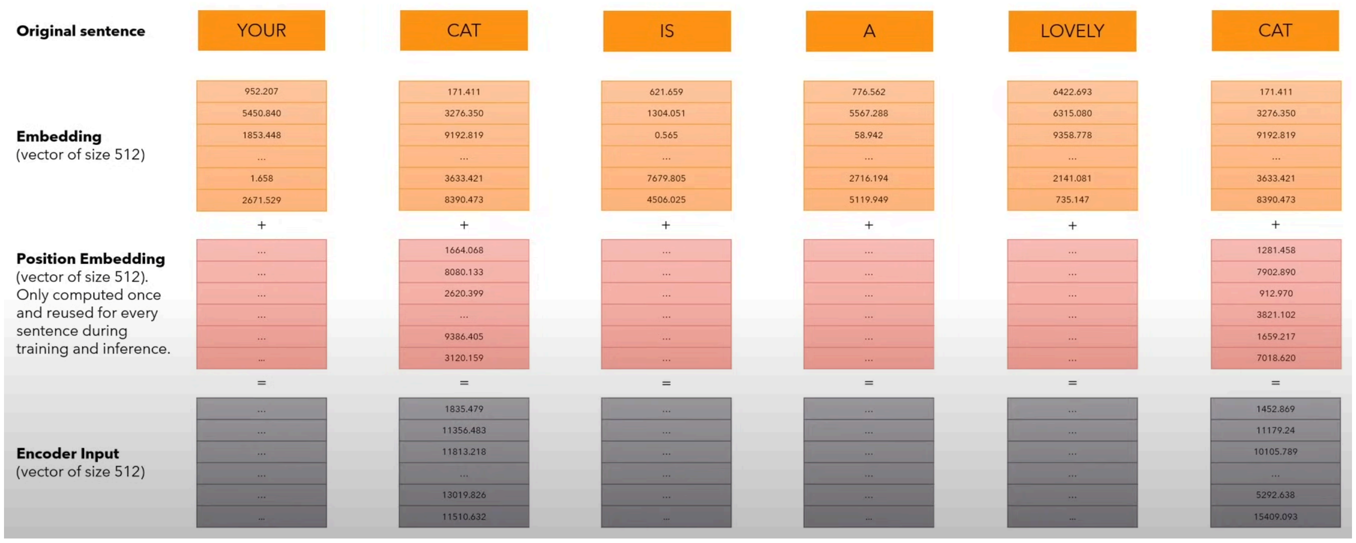
To achieve that, in the original paper they used sine and cosine functions to encode relative distance between positions. Sinus for even and cosinus for uneven "positions" in vector:



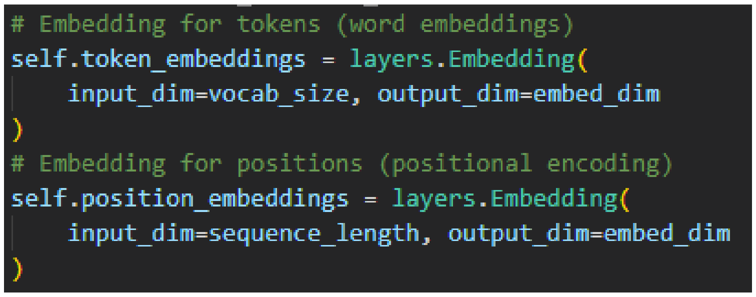
$$PE(pos, 2i) = \sin \frac{pos}{10000^{\frac{2i}{d_{model}}}}$$

$$PE(pos, 2i+1) = \cos \frac{pos}{10000^{\frac{2i}{d_{model}}}}$$

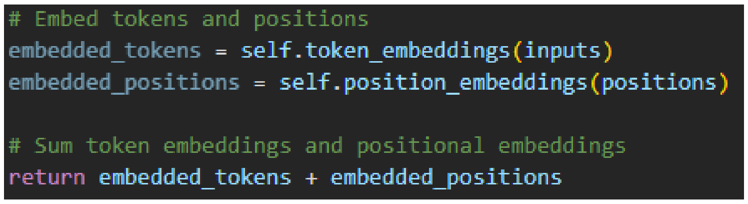
Also now sometimes another technique called Learned Positional Embedding is used, which has better task-specific adaptation but struggles to extrapolate to longer sequences.
Positional vectors are computed only once and then reused in every sentence.
In the end, the positional vector is combined with the token vector before going to the encoder and decoder stack.



Original sentence	YOUR	CAT	IS	A	LOVELY	CAT
Embedding (vector of size 512)	952.207	171.411	621.659	776.562	6422.693	171.411
	5450.840	3276.350	1304.051	5567.288	6315.080	3276.350
	1853.448	9192.819	0.565	58.942	9358.778	9192.819
	...	...	...	...	...	...
	1.658	3633.421	7679.805	2716.194	2141.081	3633.421
	2671.529	8390.473	4506.025	5119.949	735.147	8390.473
	+	+	+	+	+	+
Position Embedding (vector of size 512). Only computed once and reused for every sentence during training and inference.	...	1664.068	...	...	...	1281.458
	...	8080.133	...	...	...	7902.890
	...	2620.399	...	...	...	912.970
	...	...	...	...	...	3821.102
	...	9386.405	...	...	...	1659.217
	...	3120.159	...	...	...	7018.620
	=	=	=	=	=	=
Encoder Input (vector of size 512)	...	1835.479	...	...	...	1452.869
	...	11356.483	...	...	...	11179.24
	...	11813.218	...	...	...	10105.789
	...	...	...	...	...	...
	...	13019.826	...	...	...	5292.638
	...	11510.632	...	...	...	15409.093



```
# Embedding for tokens (word embeddings)
self.token_embeddings = layers.Embedding(
    input_dim=vocab_size, output_dim=embed_dim
)
# Embedding for positions (positional encoding)
self.position_embeddings = layers.Embedding(
    input_dim=sequence_length, output_dim=embed_dim
)
```



```
# Embed tokens and positions
embedded_tokens = self.token_embeddings(inputs)
embedded_positions = self.position_embeddings(positions)

# Sum token embeddings and positional embeddings
return embedded_tokens + embedded_positions
```

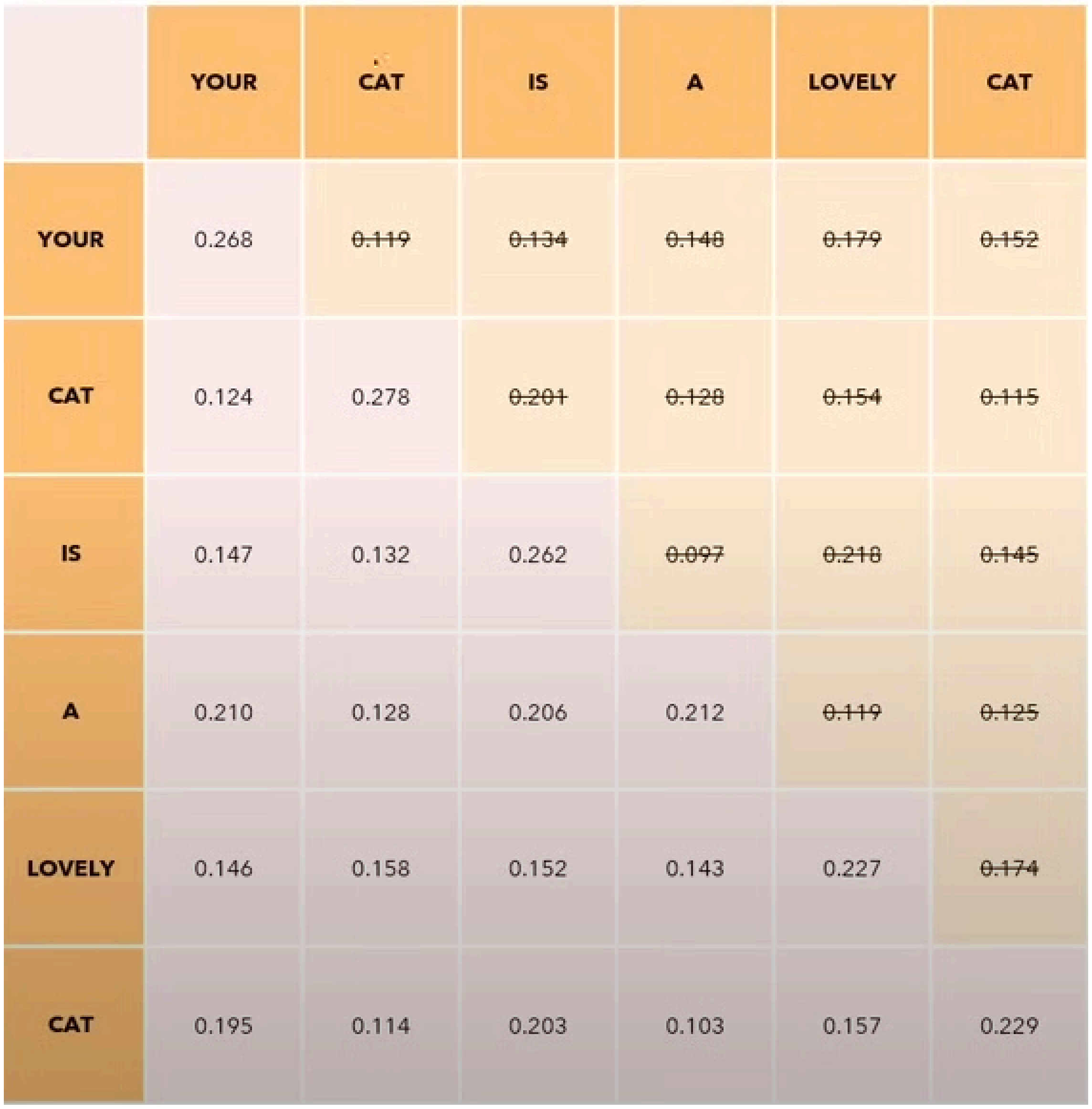
Attention layers (self-attention, encoder-decoder -attention, multi-head attention)

Link to video with great explanation

Self attention allows the model to relate the words to each other (inside the same sequence). Oversimplified, by multiplying Queries and Keys matrices we get a matrix with values that indicate how each word is related to another in the sentence (attention scores). Then, after multiplying by the Values matrix we get a new matrix, where each row captures meaning (token embedding), position (positional embedding) and interactions with other words.

Multihead attention is based on the Self attention formula, but instead of taking the whole matrix at once it splits it into smaller ones allowing it to capture more relationships between words.
For example, if we have a sentence with 6 words and each word will have a 512 parameter vector, our original matrix after input and positional embedding will have shape 6x512.
We then divide it by the amount of attention heads (let's say 4) and end up with 4 smaller matrices 6x128. So each head sees the whole sentence, but only part of the vector, which allows heads to look at different aspects of the word.
After, with each of smaller matrices, we perform similar to self attention math, concatenate them together and end up again with a 6x512 matrix.

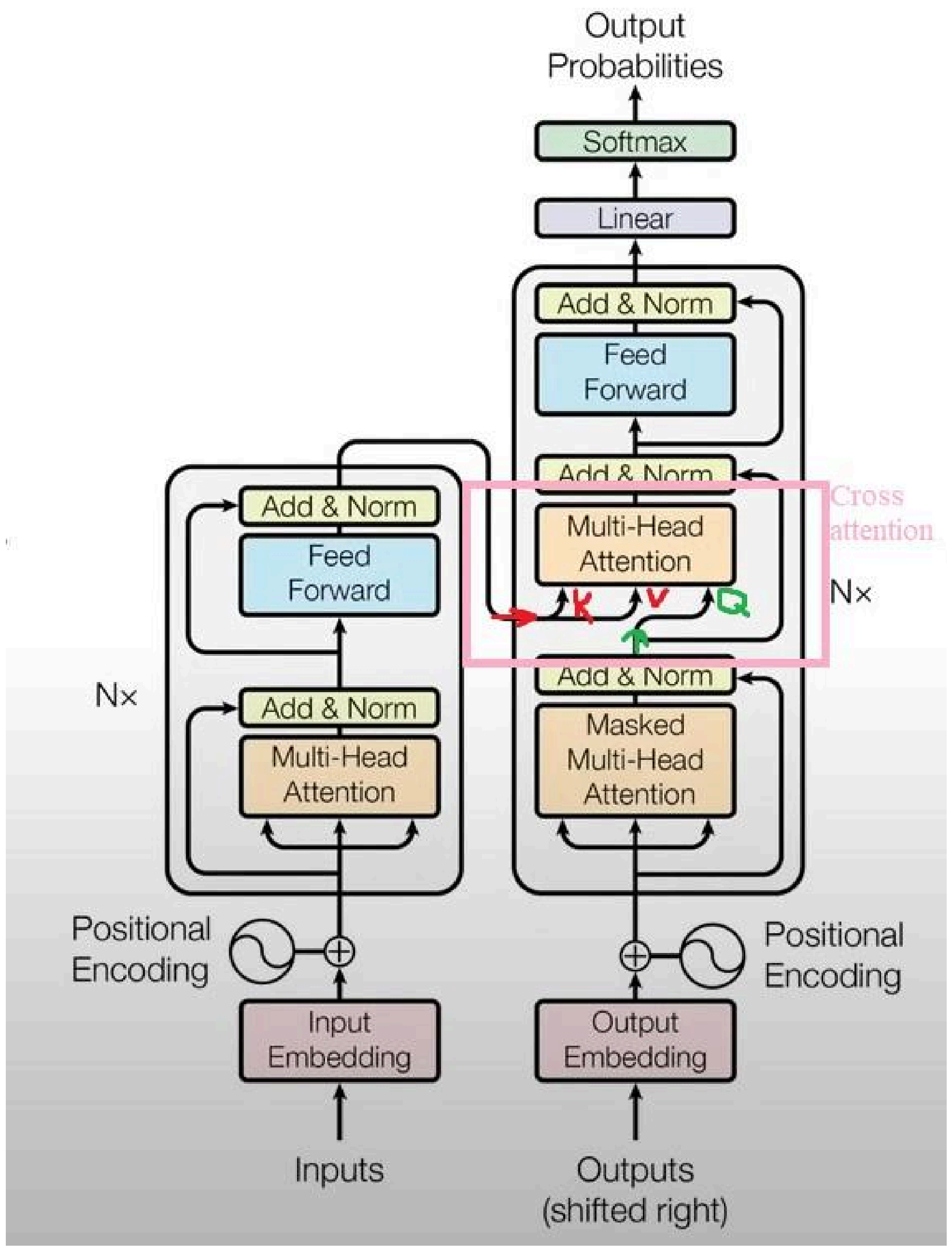
Masked Multi-Head attention in the decoder layer prevents the model from cheating. The model must not see future words and that is why after we count attention scores, we replace all values above the diagonal with zeros



	YOUR	CAT	IS	A	LOVELY	CAT
YOUR	0.268	~~0.119~~	~~0.134~~	~~0.148~~	~~0.179~~	~~0.152~~
CAT	0.124	0.278	~~0.201~~	~~0.128~~	~~0.154~~	~~0.115~~
IS	0.147	0.132	0.262	~~0.097~~	~~0.218~~	~~0.145~~
A	0.210	0.128	0.206	0.212	~~0.119~~	~~0.125~~
LOVELY	0.146	0.158	0.152	0.143	0.227	~~0.174~~
CAT	0.195	0.114	0.203	0.103	0.157	0.229

Here, for example, the first word "Your" must not be able to see and interact with the rest of the words.

Encoder-decoder attention, or cross attention, unlike self-attention, works with different sequences. This lets the decoder use information from the input sequence while generating the output sequence. Really useful in translation tasks. Uses the same math as other attention, but in this case Query matrix comes from decoder Masked Multi-Head attention while Keys and Values from encoder outputs.

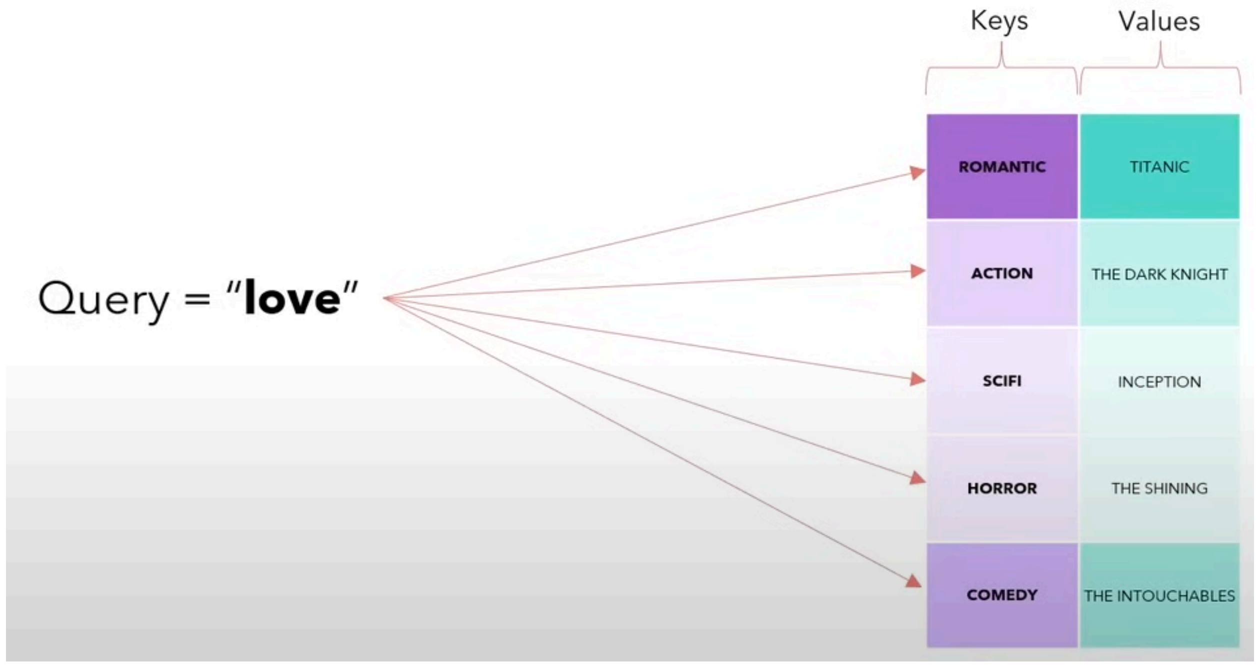


Example by ChatGPT: Decoder token `"chat"` (French for cat) can attend to `"cat"` in the encoder outputs, so the translation is accurate.

Query, Key, Value

For a set of tokens we create three matrices:

1. Query - the question that we are asking
2. Key - address label that describes each token
3. Value - actual desired information

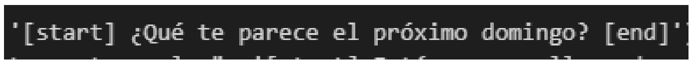


Transformer –specific hyperparameters

Vocabulary size (or buffer size) - literally how many words a model will understand. The more the size - the heavier and slower the model. But more expressive and accurate.
Batch size - How many sentences model is processing in one training step
Sequence length (or max tokens) - How long the sentence model can handle, defines the size of positional embeddings as how far attention can reach. In the examples with machine translations they are usually using up to 512 tokens, while real LLM dozens of thousands
Embedding dimension - Size of the vector representing word (token) after embedding. The larger it gets the more details about the word it can hold, but on the other hand computing becomes heavy
Latent dimension - Size of the hidden layer inside the feedforward network (FFN) of each encoder/decoder block.
Number of attention heads - Each new head takes part of the embedding, reducing the size of this part for other heads. Allows to catch more complex relationships between the tokens

Start of sequence and end of sequence model

In order to give a model when to start and when to end we should also include special tokens for starts and ends



```
'[start] ¿Qué te parece el próximo domingo? [end]'
```

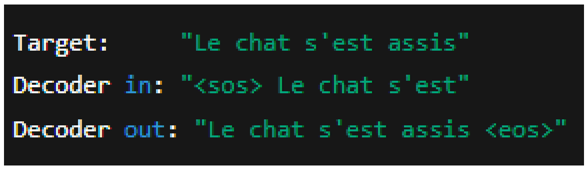
The total structure of the transformer network – "how does it work on the general level?"

In case of **already trained** english - spanish translation:

1. Embed original english sentence
2. Add positional embedding
3. Pass the vector to encoder layer
4. Using multihead attention, add “context” information to our vector
5. Pass the Key and Value to Decoder
6. Give [start] token to the decoder
7. Masked multihead attention produce Query
8. In cross attention block, count the resulting vector based on Query from decoder and Key and Value from encoder
9. Compare the vector to all vectors in vocabulary
10. Choose the best suitable

After we have our first world, we continue to work only with Decoder, since the original English sentence doesn’t change and there is no need to recompute encoder output. On the next step we add our previous output to decoder input and repeat it until end token:
[start] -> [start] Cuando -> … -> [start] Cuando ella era joven, era muy popular. [end]

In case of **training**, we use teacher forcing - we give the whole spanish sentence to decoder, but without [end] token:



```
Target:      "Le chat s'est assis"
Decoder in:  "<sos> Le chat s'est"
Decoder out: "Le chat s'est assis <eos>"
```

However, the decoder does not see the full target sentence directly. Instead, it sees the target shifted right (start token). The model learns to predict the next token at each step, and thanks to the masked multihead attention block, decoder has access only to the tokens it has already “predicted”.
Then we count the loss function and backprograde the loss to all the weights as in usual networks