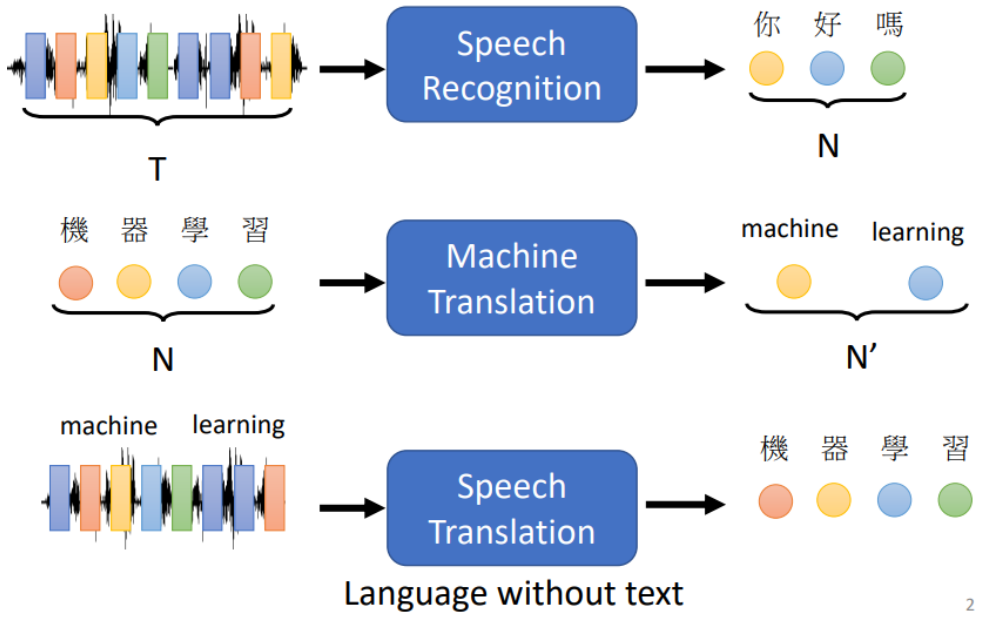
Lecture 5 Transformer Model

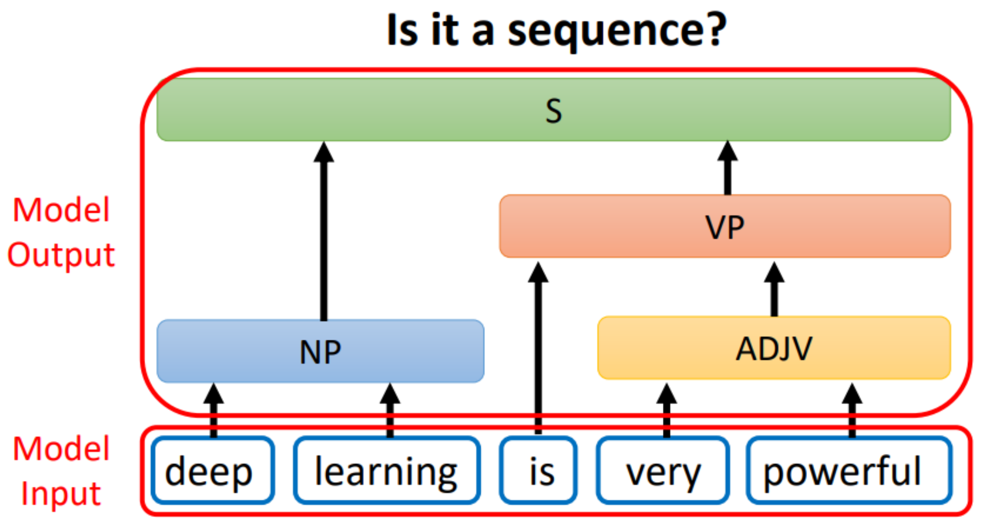
5.1 Introduction

Transformer is a sequence-to-sequence model (input a sequence, output a sequence) The output length is determined by the model.

Example 1 Input: Voice → Output: Text: Input is a sequence; output is a sequence. The output length is determined by the model.



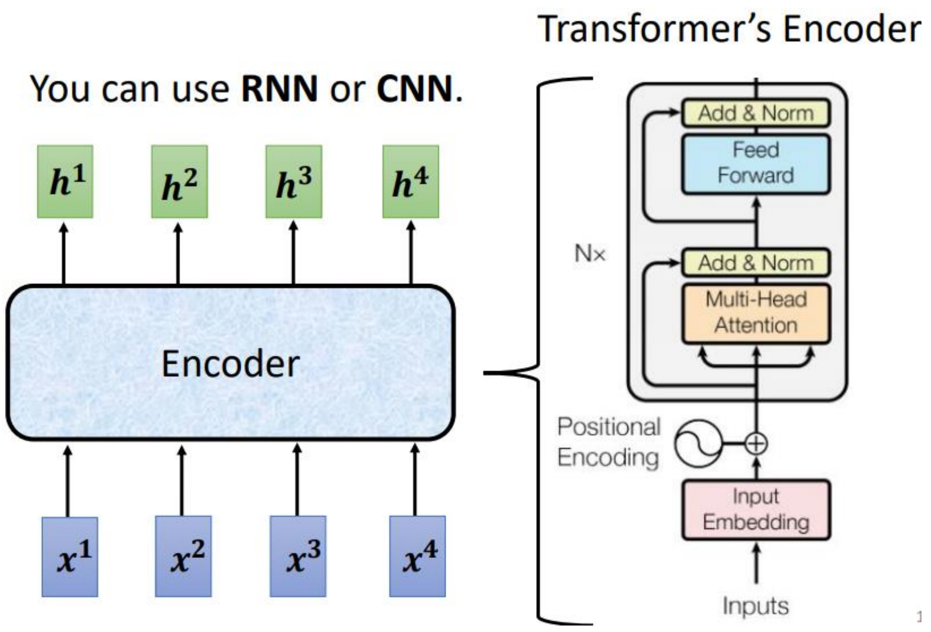
Example 2 Seq2seq for Syntactic Parsing: Model Input is a sequence. Model Output is a parsing tree. The parsing tree is essentially viewed as the sequence.



Source: Grammar as a Foreign Language.

In a Seq2seq model, the role of encoder is to take one sequence of vectors as input and generate a sequence of vectors as the output. Many models can achieve the goal. In Transformer model, the encoder uses self-attention.

5.2 Transformer's Encoder



Transformer's encoder has many blocks. Each block can take one sequence of vectors as input and generate a sequence of vectors as the output.

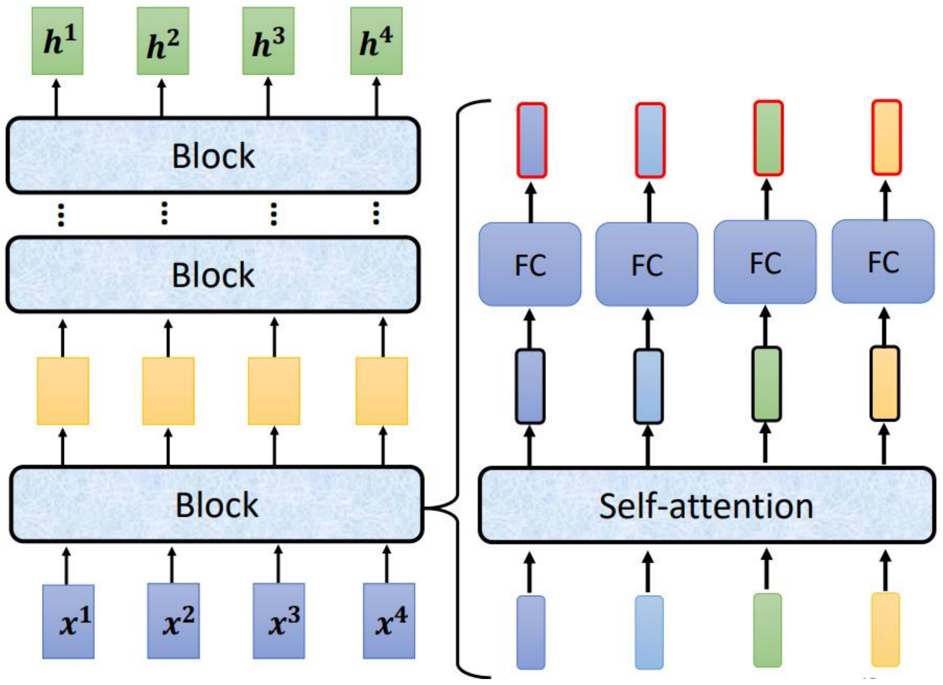
The first block takes a sequence of vectors as input and generates a sequence of vectors. Input embedding refers to the process of converting input tokens into dense vector representations that capture their semantic meaning and contextual information. This embedding process plays a crucial role in transforming raw input data, such as words or tokens, into a format that the Transformer model can effectively process.

Here's a breakdown of how input embedding works in a Transformer:

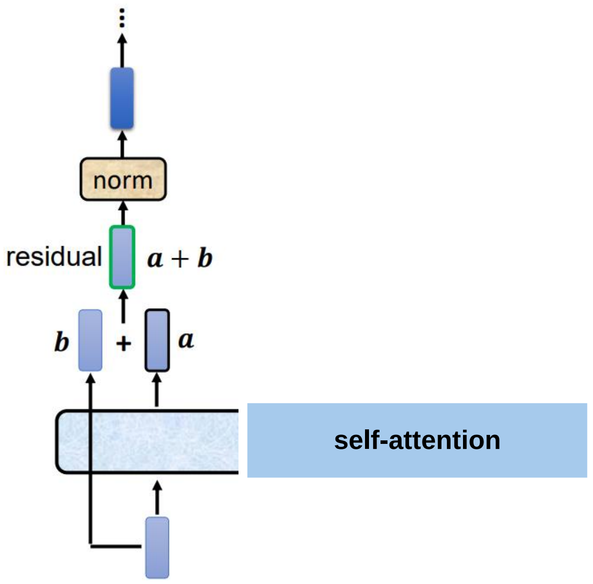
- Tokenization: The input text is tokenized into individual tokens, such as words or subwords, using techniques like word segmentation or Byte Pair Encoding (BPE).
- Word Embeddings: Each token is then converted into a dense vector representation known as a word embedding. These embeddings are typically pre-trained using methods like Word2Vec, GloVe, or FastText on large corpora to capture semantic relationships between words.
- Positional Encoding: In addition to word embeddings, positional encodings are added to capture the position of each token in the input sequence. This is crucial for the Transformer model to understand the sequential order of tokens.
- Summing or Concatenating: The word embeddings and positional encodings are combined, usually by summing or concatenating them together, to create the final input embeddings for each token in the input sequence.

These generated sequences of vectors are then passed to the next block one after another for further processing, until reaching the hidden layer. Although we sometimes also call each block a "layer." These blocks are not necessarily a single layer (e.g. feed forward neural network). Each block may use multiple layers of neural networks to process the input vectors.

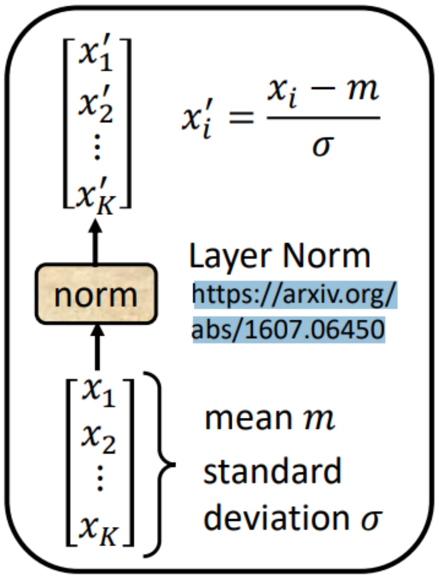
Inside each block, the input vectors first go through a layer of self-attention to capture the information carried by the input sequence, such as meaning, context, query, and key, etc. Then the self-attention passes the processed information into a fully connected neural network. Afterwards, the fully connected network will pass its output (as shown by the sequence with red frame below) into the next block.



The transformer model that is prevailing for the industry use is slightly more sophisticated. They use a framework called "**Residual Connection**."



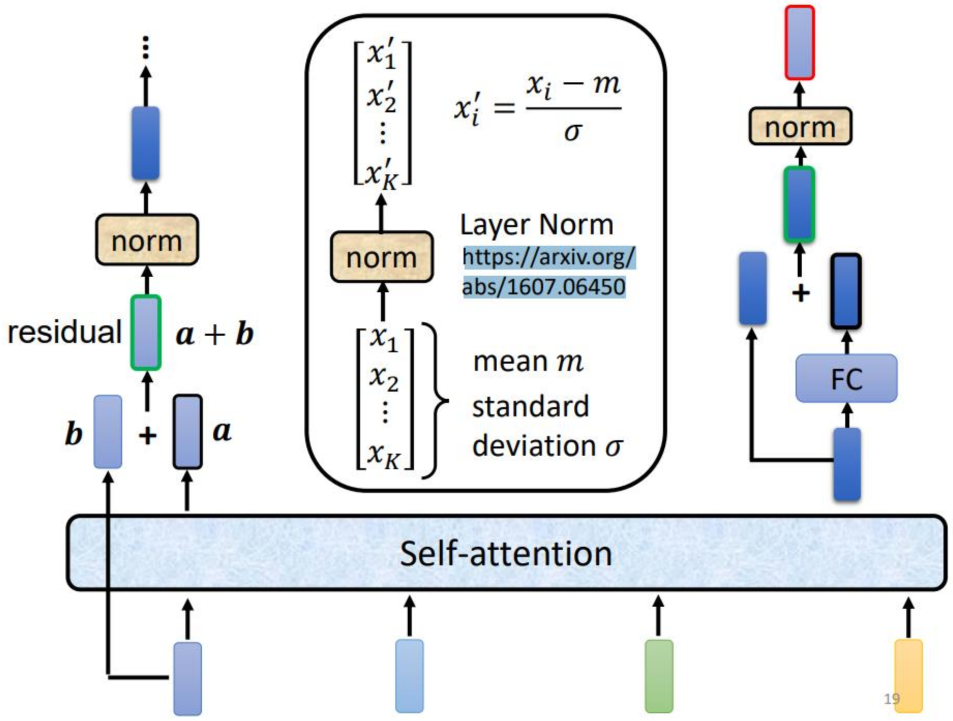
To apply the residual connection, the block first uses a self-attention to process the input b and obtains the sequence of vectors a. The block then added the input b on top of the out from the self-attention a to calculate the so-called sequence of vectors called "residual" (i.e., residual = $b + a$). Then we apply one special type of "normalization" to the residuals (different from "batch normalization"). The normalization method is called "layer normalization (Layer Norm: https://arxiv.org/abs/1607.06450)." The outcome from the layer normalization will later become the input of the fully connected network.


x'_1 x'_2 $\vdots$ x'_K
$x'_i = \frac{x_i - m}{\sigma}$
norm
Layer Norm
https://arxiv.org/abs/1607.06450
x_1 x_2 $\vdots$ x_K
mean m
standard deviation σ

Note:

Batch normalization: For features $X^1 \dots X^N$, batch normalization calculates the mean and the standard deviation across all N features for each timestamp, from $1 \dots K$.

Layer normalization: For a particular feature X, layer normalization calculates its mean as $m = \frac{1}{K}\sum_{i=1}^{K} x_i$ and the standard deviation of X as $\sigma^2 = \frac{1}{K}\sum_{i=1}^{K}(x_i - m)^2$.



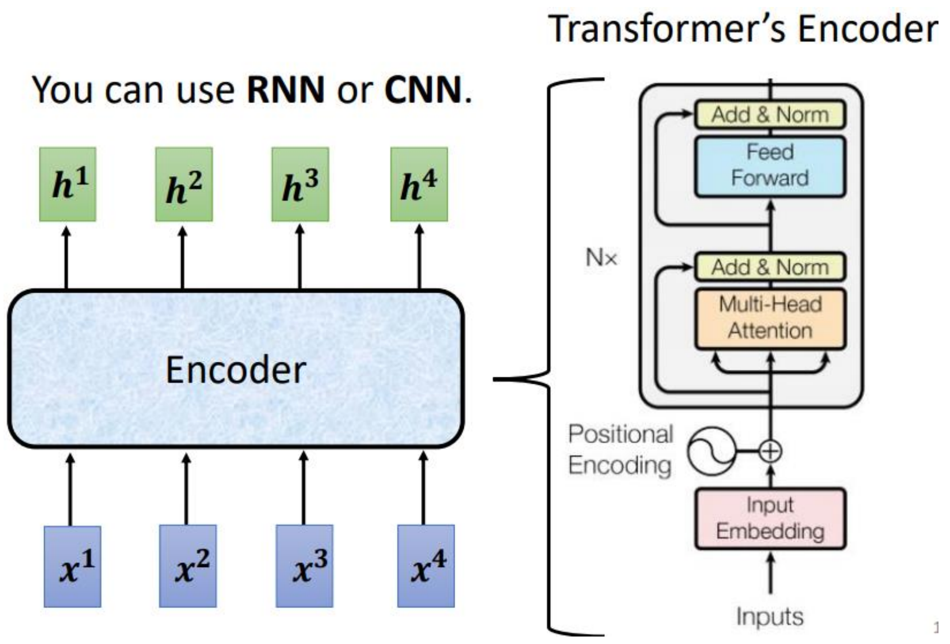
norm
residual
$a + b$
b
+
a
$\begin{bmatrix} x'_1 \\ x'_2 \\ \vdots \\ x'_K \end{bmatrix}$
$x'_i = \frac{x_i - m}{\sigma}$
norm
Layer Norm
https://arxiv.org/abs/1607.06450
$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_K \end{bmatrix}$
mean m
standard deviation σ
norm
+
FC
Self-attention
19

$$c = \; Layer\ Norm\ (\text{residual})$$

The fully connected network takes the outcome c from the layer normalization and applies the fully connected networks to it. The outcome d will add on top of c to obtain the output from the fully connected network (Residual Connection $e = d + c$). Finally, we apply the layer normalization again to e.

$$f = Layer\ Norm\ (e)$$

The output after the normalization will serve as the output from each “Block” in the Transformer.



To summarize what happens in the encoder of Transformer:

Step 1: Input goes through input embedding and positional encoding.

Step 2: The outcome from Step 1 goes into the "block." The block first applies Multi-Head Self-Attention to the block input, then the residual connection and finally the layer norm.

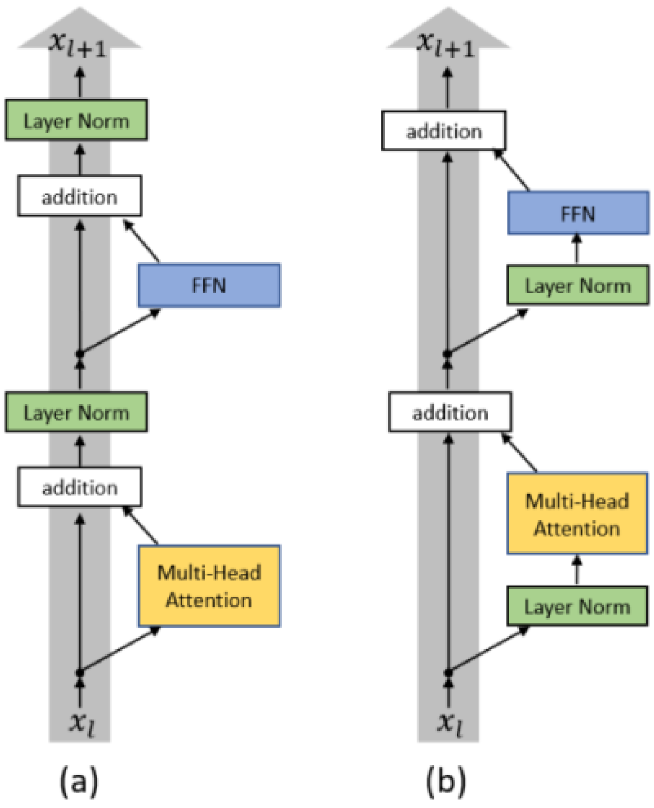
Step 3: The outcome from the layer norm will next be passed to the fully-connected feed forward neural networks. The outcome from the network will take the residual connection and the layer norm again to produce the outcome from the block.

Step 4: The block will repeat N times to compose of the Transformer's Encoder.

This is exactly what BERT does. BERT can be regarded as the encoder of Transformer. The above process describes how the original paper about Transformer configures the encoder, but it doesn't have to be the optimal configuration.

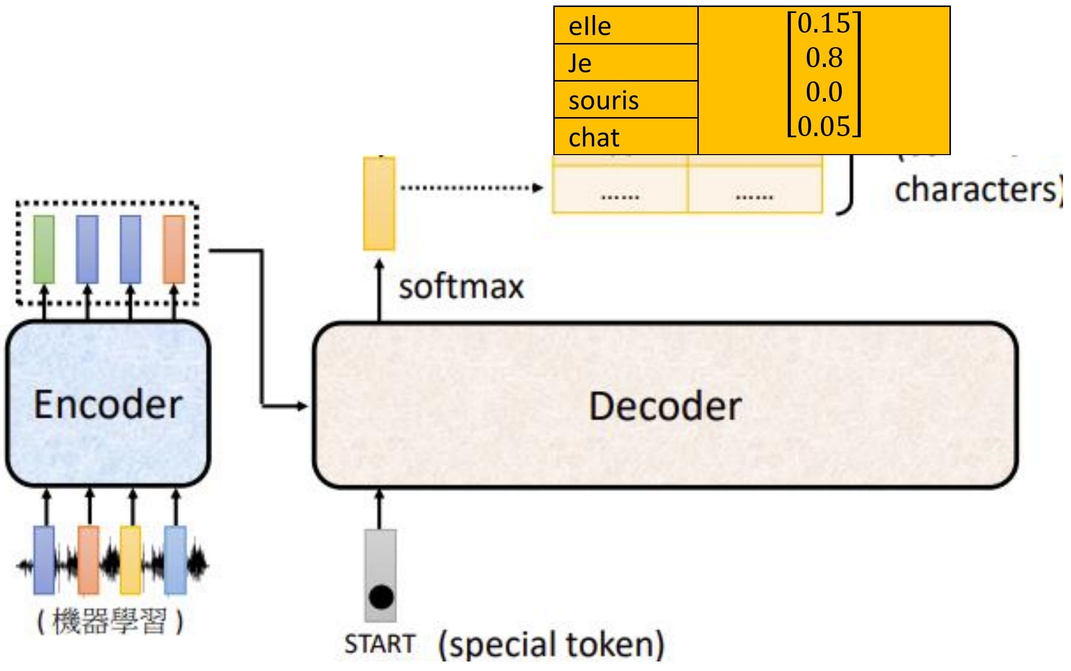
Alternatively, you could use the structure as specified in:

- On Layer Normalization in the Transformer Architecture (https://arxiv.org/abs/2002.04745)
- PowerNorm: Rethinking Batch Normalization in Transformers (https://arxiv.org/abs/2003.07845)



5.3 Transformer's Decoder

Decoder takes outcome from the encoder and produces output. Here is a description about how the decoder of Transformer works.



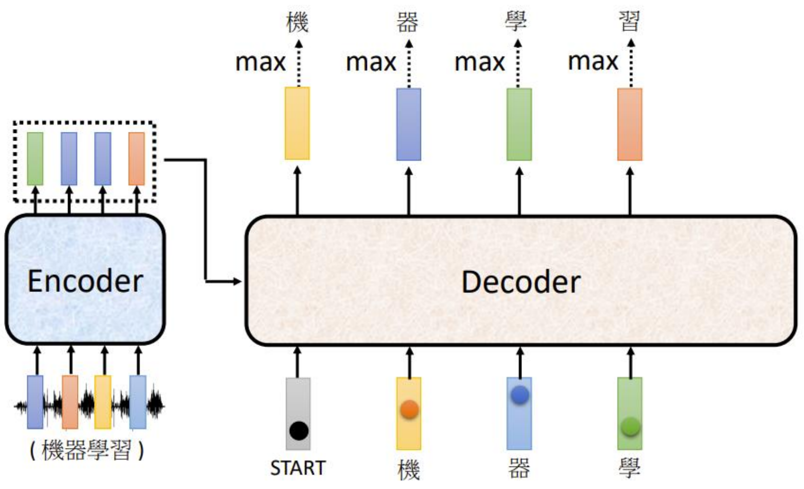
- Decoder first takes a special token as input to identify the “BEGIN”, the token is going to be a vector composed for 1 and 0, with one element alone to be 1 and the rest elements to be 0. For example,

$$BEGIN = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

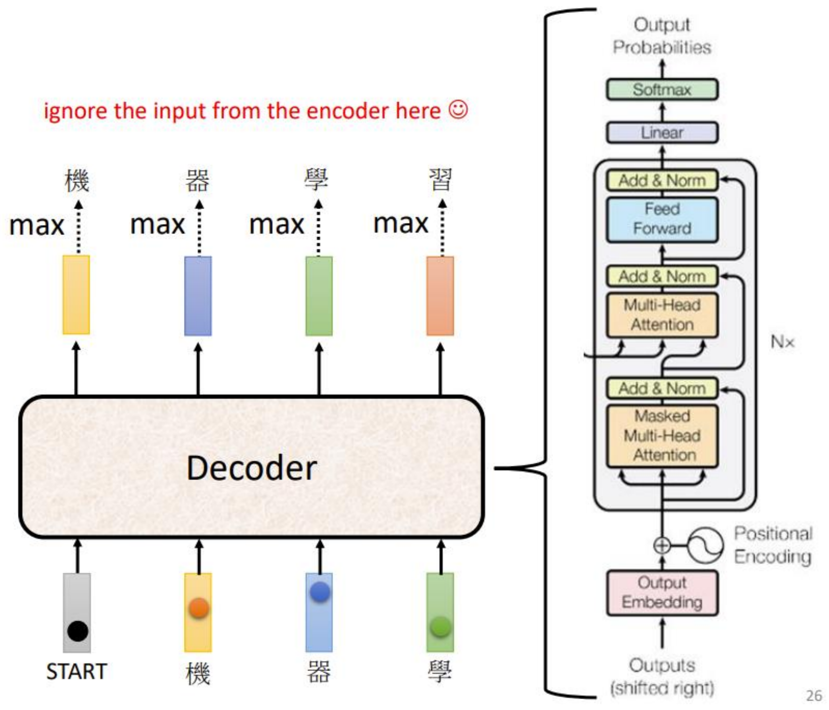
- The output from the decoder is a vector. The vector has the same size as your vocabulary. Vocabulary should contain all possibilities of your output. For example, if you specify the output to be letters in English, the vector should take the length of 26. If you specify the output to be characters in Chinese, the vector should take the same length of all commonly used characters, which is about 2500. If you specify the output to be commonly used words in English, the vector should take the length of 2500 to 3000.
- Before the output is produced, there exists a softmax process. Take English words for example, the output assigns a probability (from the softmax) to each word. The word with the highest probability is selected. The output vector is essentially a distribution, with the sum of all elements to be 1. The following table shows how the English to French translation takes place. Given the input to be a vector representing the word “I”, the output vector takes the following form:

elle	$\begin{bmatrix} 0.15 \\ 0.8 \\ 0.0 \\ 0.05 \end{bmatrix}$
Je	
souris	
chat	
… …	

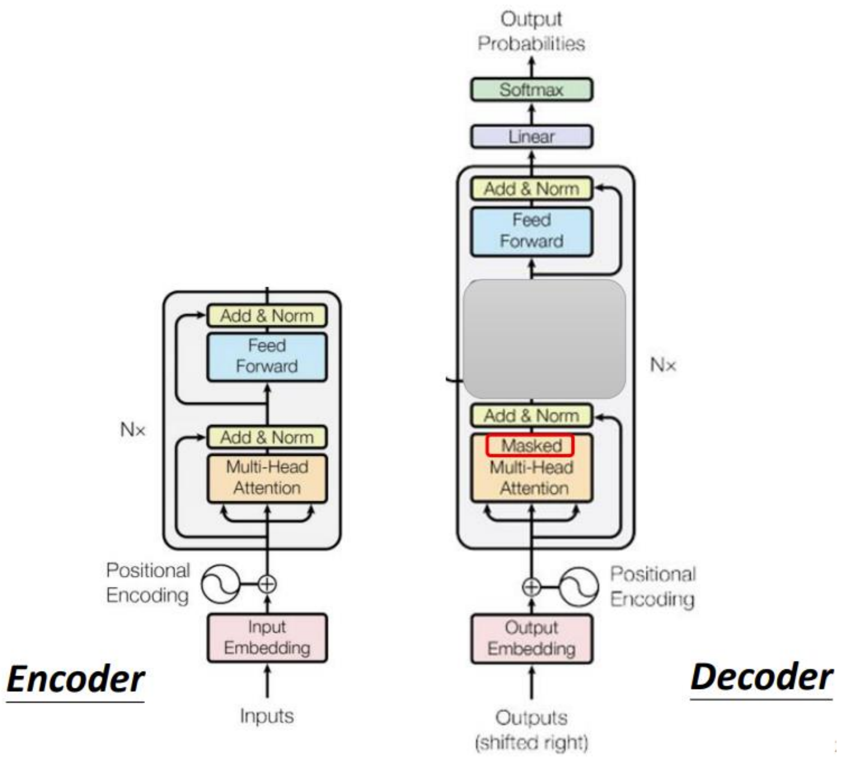
- Feed the output “Je” from the decoder into the decoder again. The decoder then uses softmax to output another word “suis.” The process repeats to determine the next output word. In the meantime, the decoder takes certain input from the encoder, which we are about to explain later. Here, the decoder takes its output from the previous step $o(t-1)$ as the input to produce the output $o(t)$. The risk here is that if output from any timestamp is wrong, the decoder is going to take the wrong input to derive the output for the next timestamp. The problem is called “error propagation.” For example, in Chinese to Chinese speech-to-text translation, the speech file should be translated into four characters: “机-器-学-习” , which means “machine learning” in Chinese when the transformer hears the pronunciation “ji-qi-xue-xi.” However, for the same pronunciation, “ji” may correspond to the word “鸡”, which is a noun and means “the chicken”; or the word “击”, which is a verb and means “hit” or “strike.”



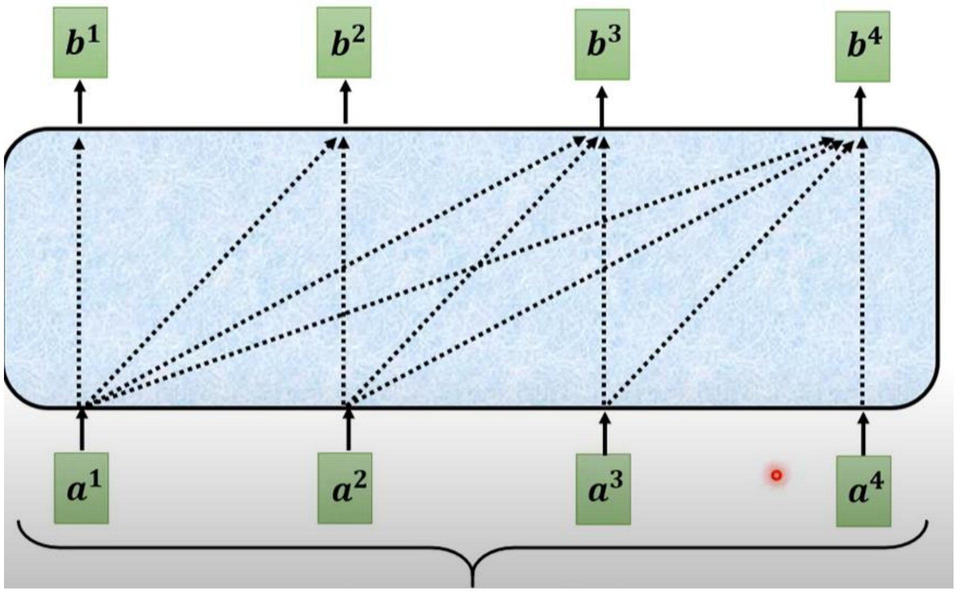
Let’s leave aside how the decoder interacts with the encoder for now and take an anatomy of how the decoder works. The structure of the decoder is illustrated below.



Let’s first compare Decoder to Encoder.



If we cover the Muti-Head Attention and Add & Norm part of the decoder, the decoder looks similar to the encoder. After the first block of the Muti-Head Attention, the decoder includes an additional step – "Masked."



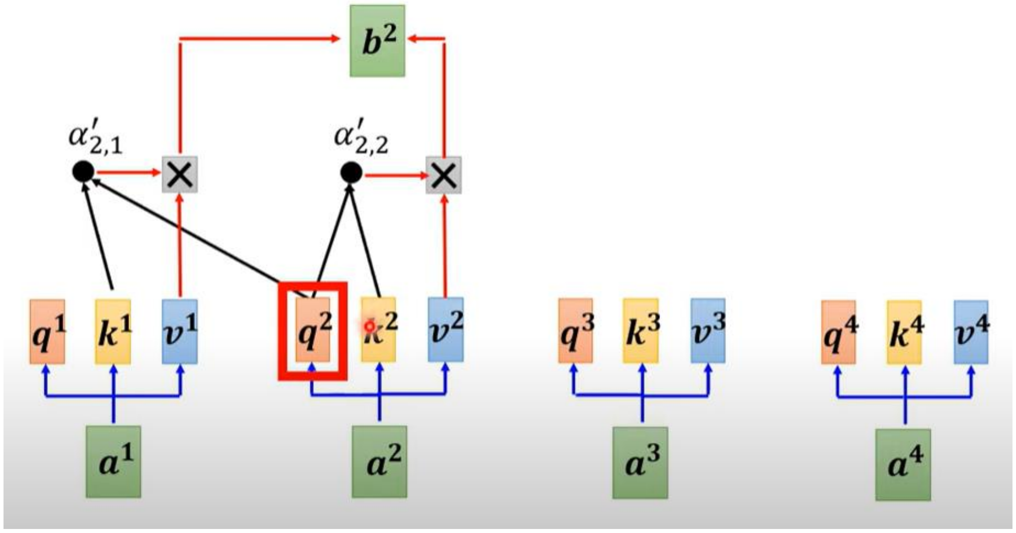
Can be either **input** or a **hidden layer.**

In an "unmasked" self-attention, each b^i contains information from the entire input sequence: a^1, a^2, a^3, a^4. In a "masked" self-attention, b^1 contains information from a^1 alone. b^2 contains information from a^1 and a^2. b^3 contains information from a^1, a^2 and a^3. Only b^4 contains information from the entire input sequence: a^1, a^2, a^3, a^4. Take the calculation of b^2 for instance,

$$b^2 = \alpha'_{2,1}v^1 + \alpha'_{2,2}v^2 \quad where$$

$$\alpha_{2,1} = q^2 \cdot k^1$$

$$\alpha_{2,2} = q^2 \cdot k^2$$



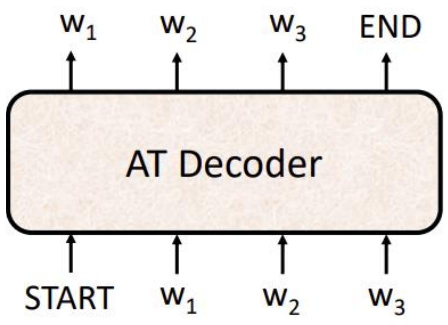
Why the decoder has to use masked self-attention and differ from the attention mechanism used in encoder? This is because when we process the input, we need to understand the meaning of the whole sentence. When we generate the output, we generate one word after another, which is sequential. The decoder thus only considers information carried from earlier timestamp.

Decoder needs to decide what is the length of the output. It is almost unlikely to infer the output length from the input sequence. The decoder needs to make inference about when and where to stop. In an Autoregressive decoder, we can resolve the problem by adding a "Stop Token." We name the vector: "END." It can take the same form as the "BEGIN."

$$END = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The role of the decoder is then to identify the meaning of the output at all previous timestamps and makes the probability assigned to the stop vector to be the "maximum." This is where the decoder ends the output sequence and decides the output length.

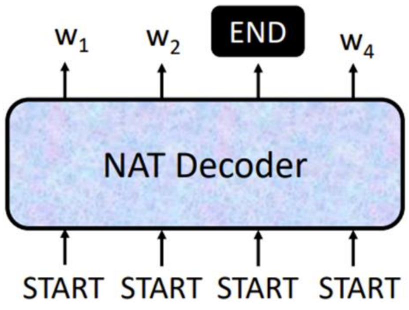
Note: Autoregressive Decoder vs Non-autoregressive Decoder



In an autoregressive decoder (AT), the model takes the special token "START" and gets the output w_1; then takes w_1 as the input to obtain the output w_2. The decoder next takes w_2 as input and produces the output w_3 until when the "Stop Token" receives the maximum probability from the softmax.

In a non-autoregressive decoder (NAT), the model takes a sequence of "BEGIN" token simultaneously and produces a sequence of sentences simultaneously. The advantage of NAT is to have parallel and controllable output length, but a lot of studies find that NAT has worse performance than AT (due to multi-modality). NAT decides the output length via the following methods:

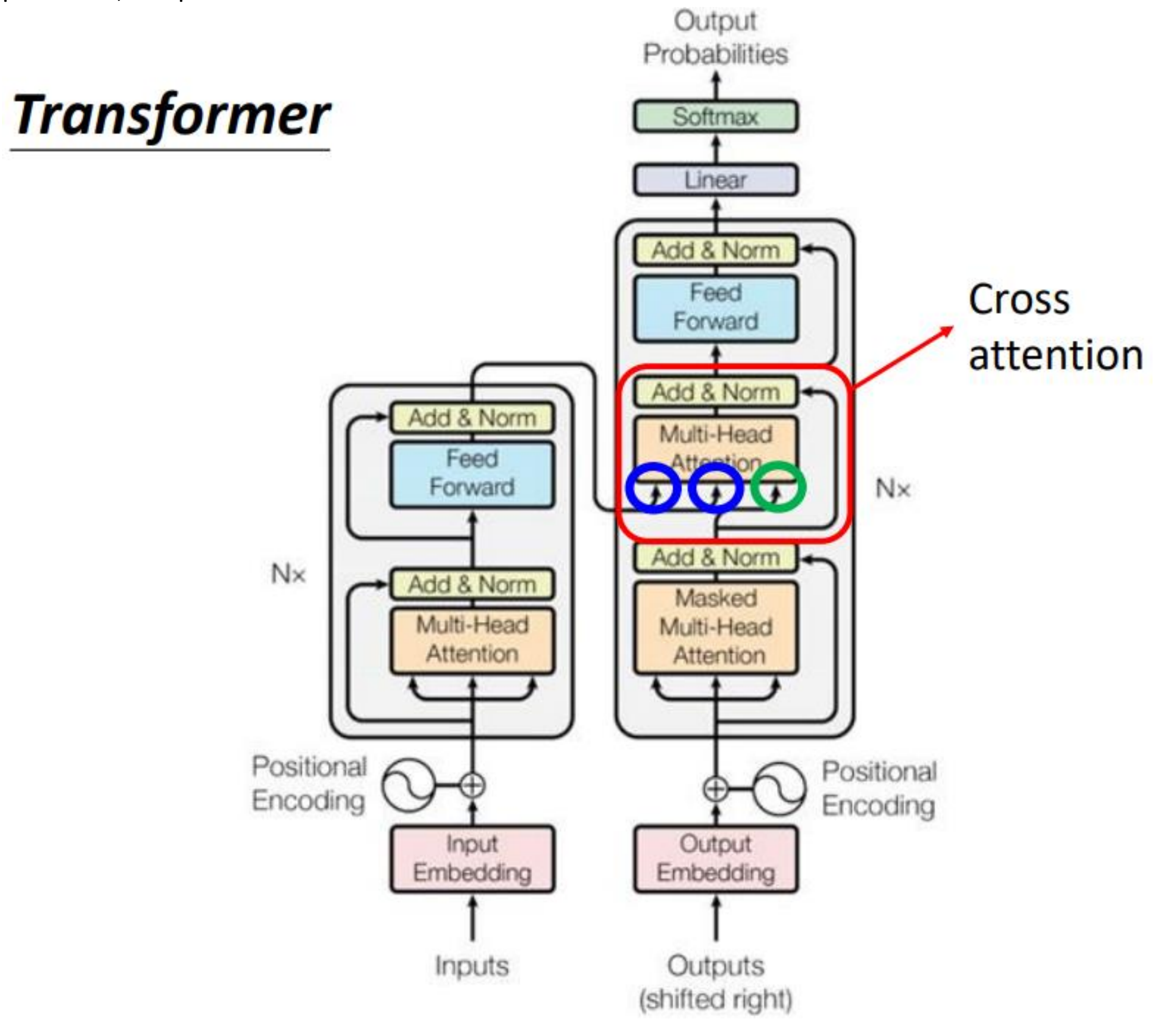
- Add another predictor for output length, e.g. add a classifier x = 4, decoder(x) = 6 then output length to be 6.
- Output a very long sequence, ignore tokens after END.



5.4 Cross Attention

Cross attention, also known as encoder-decoder attention, is a mechanism used in transformer-based models, particularly in sequence-to-sequence tasks like machine translation or text summarization. Cross Attention allows for communication between the encoder and the decoder. the decoder uses cross attention to incorporate information from the encoder's output, in addition to self-attention. For each position in the output sequence, the decoder calculates attention scores with respect to all positions in the encoder's output. These attention scores determine the relevance of each encoder position to the current decoder position.

In the highlighted block, there are two inputs from the encoder, and one input from the decoder. In particular, the process works as follows.



First, Encoder Representation. The encoder produces representations for each position in the input sequence. These representations capture the contextual information of each token based on its surrounding tokens. Assume the encoder takes a sequence of speech recordings as the input, and the outputs a^1, a^2, a^3. The encoder then produces keys k^1, k^2, k^3 from the encoder.

$$k^1 = W^k a^1$$

$$k^2 = W^k a^2$$

$$k^3 = W^k a^3$$

These are context vectors to be applied to each position in the decoder. These context vectors represent the relevant information from the encoder's output that should be considered while generating the corresponding token in the decoder.

In the meantime, Decoder Self-Attention. Like in the encoder, the decoder uses self-attention to capture dependencies within the output sequence itself. This helps the model understand the context of each token in the output sequence. The decoder then combines the context vector with its own representation and passes it through feedforward layers to generate the output token probabilities.

In particular, the decoder takes the BEGIN token into a masked self-attention. The output a^1 multiplies a weight matrix to obtain the decoder query.

$$q = W^q a^1$$

Cross-attention between the decoder and encoder is achieved by multiplying the query from decoder with the keys from the encoder.

$$\alpha_1' = softmax(q \cdot k^1)$$

$$\alpha_2' = softmax(q \cdot k^2)$$

$$\alpha_3' = softmax(q \cdot k^3)$$

In addition, inside the encoder, we prepare the meaning information from the input via calculating v^1, v^2 and v^3.

$$v^1 = W^v a^1$$

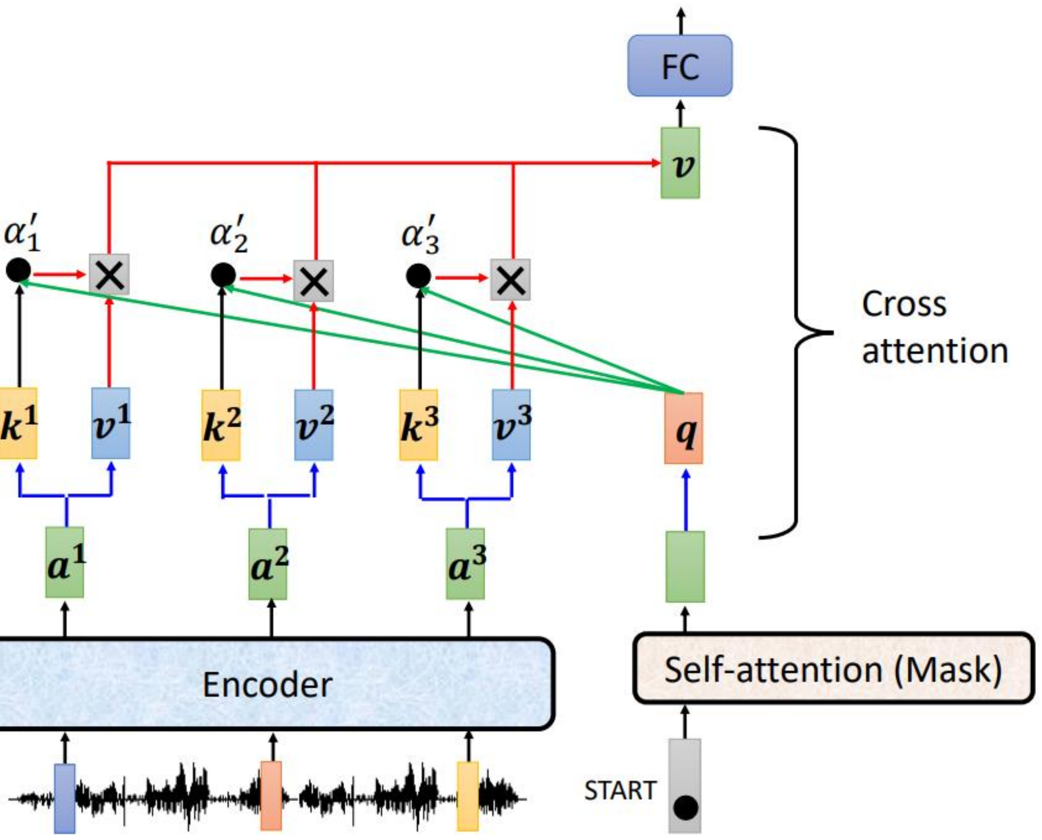
$$v^2 = W^v a^2$$

$$v^3 = W^v a^3$$

The meaning vector passed from encode to decoder is v

$$v = \alpha_1' \cdot v^1 + \alpha_2' \cdot v^2 + \alpha_3' \cdot v^3$$

The meaning vector v is to be passed into the fully connected network in the decoder. The entire procedure is called “Cross Attention.” Cross Attention takes the value information v and the key information k from encoder but query information q from the decoder.



In the next round, the decoder takes the first two vectors into the masked self-attention and produces query $q' = W^q a^2$. We multiply q' with the keys from the entire input sequence from the encoder to obtain attention scores:

$$\alpha'_1 = softmax(q' \cdot k^1)$$

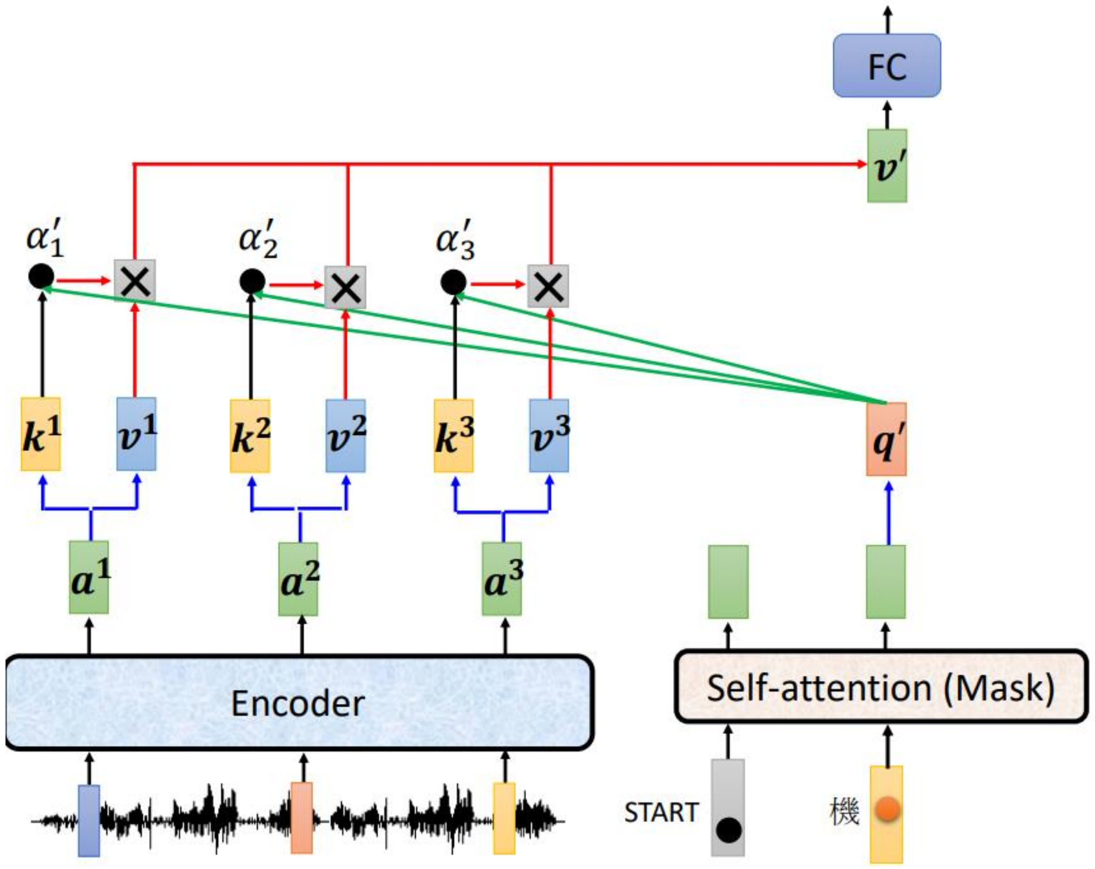
$$\alpha'_2 = softmax(q' \cdot k^2)$$

$$\alpha'_3 = softmax(q' \cdot k^3)$$

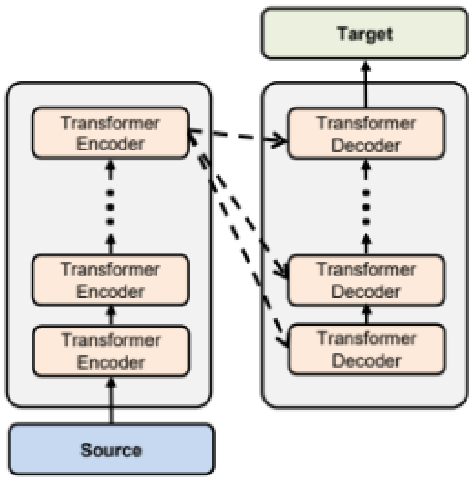
The meaning vector passed from encode to decoder is v', which is the weighted sum of input meaning weighted by attention scores.

$$v' = \alpha'_1 \cdot v^1 + \alpha'_2 \cdot v^2 + \alpha'_3 \cdot v^3$$

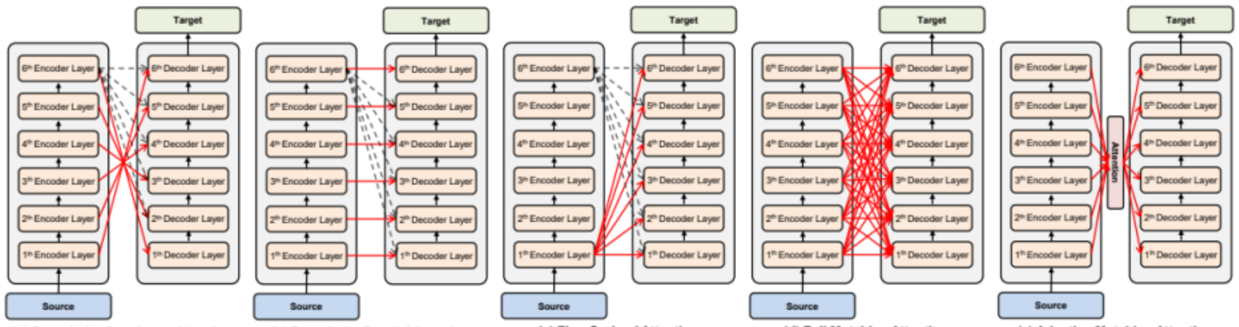
The process continues until we reach the “END” token.



In the conventional Transformer, each layer of the decoders should take input from the last layer of encoders. However, you could propose whichever structure that best fits your problem.



(a) Conventional Transformer



(a) Granularity Consistent Attention (b) Granularity Parallel Attention (c) Fine-Grained Attention (d) Full Matching Attention (e) Adaptive Matching Attention

5.5 Training

English: I am a student.

Fench: Je suis etudiant.

Ground truth: The first output from the decoder should stay close to the French word "Je" as possible. The character is represented by a one-hot vector, in which, only the element corresponds to the word "Je" is 1 but the rest elements are 0.

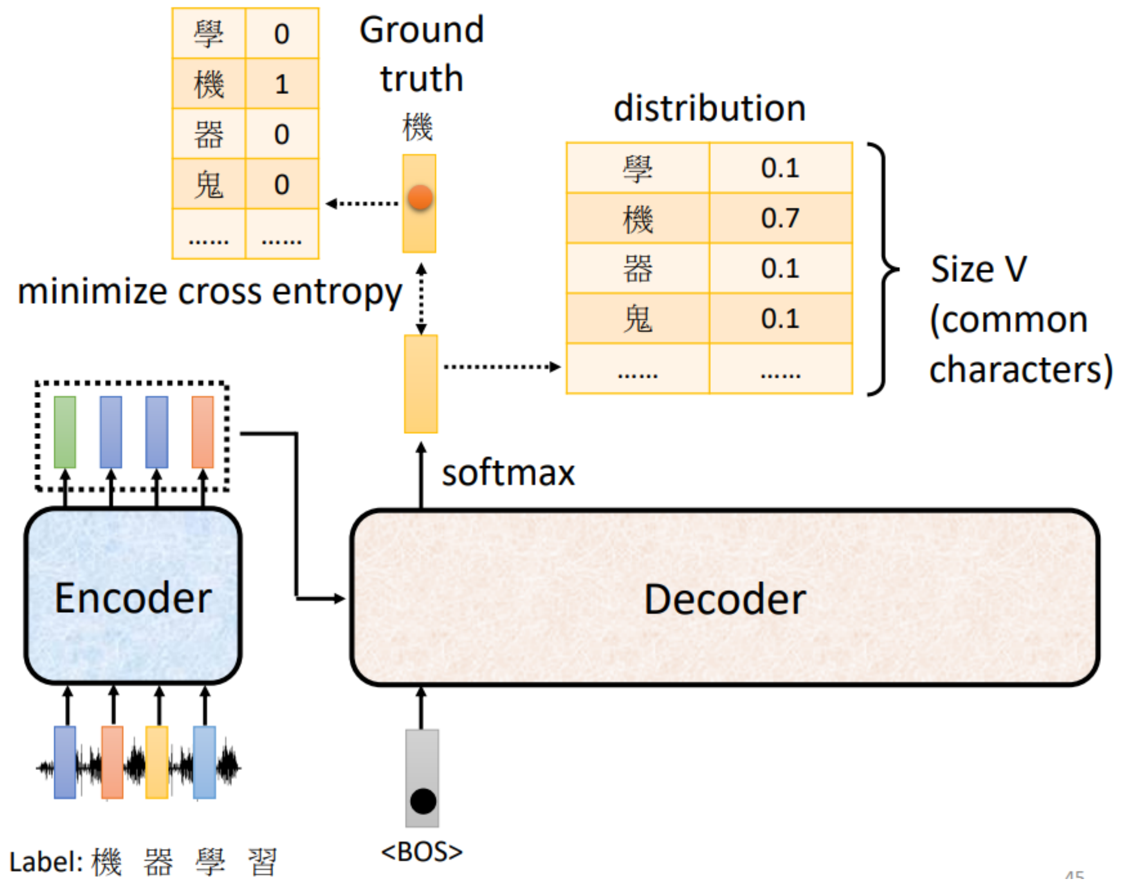
elle	$\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$
Je	
souris	
chat	

Model Output:

elle	$\begin{bmatrix}0.15\\0.8\\0.0\\0.05\end{bmatrix}$
Je	
souris	
chat	

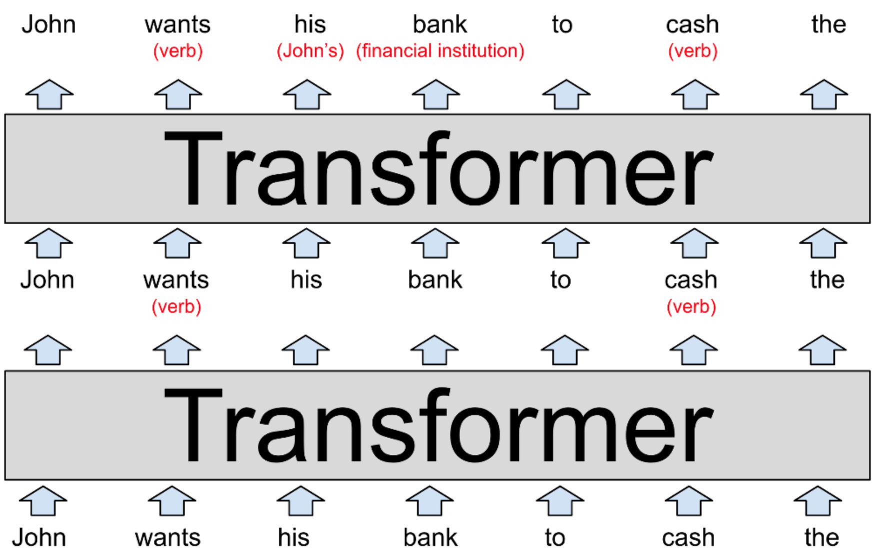
We want the distribution of the output vector from the decoder to stay as close to the "Ground Truth" as possible. We can calculate the cross-entropy between two vectors. Our target is to minimize cross-entropy. It works in the same way as in the example we have covered in RNN. For each character produced by the decoder, there is one classification resolved behind. The parameters are estimated in way to minimize the sum of cross entropy from all outputs.

Next, we feed the ground truth into the decoder. Then decoder is trained to produce "suis" when it takes "BEGIN" and "Je" as input; and to produce "etudiant" when it takes "BEGIN", "Je" and "suis" as input; and to produce "END" when it takes "BEGIN", "Je", "suis" and "etudiant" as input. The entire process is called "**Teacher Forcing**," in which the decoder sequentially takes vectors from the ground truth as input.



5.6 Application: Large Language Models

Today almost everyone has heard about LLMs, and tens of millions of people have tried them out. But, still, not very many people understand how they work. If you know anything about this subject, you've probably heard that LLMs are trained to "predict the next word," and that they require huge amounts of text to do this. But that tends to be where the explanation stops. The details of how they predict the next word is often treated as a deep mystery.



In fact, each layer of an LLM is a transformer, a neural network architecture that was first introduced by Google in the landmark 2017 paper "Attention is All You Need."

The model's input is the partial sentence "John wants his bank to cash the." These words, represented as word2vec-style vectors, are fed into the first transformer.

Research suggests that the first few layers focus on understanding the syntax of the sentence and resolving ambiguities. For instance, both "wants" and "cash" are verbs (both words can also be nouns). Later layers (which we're not showing to keep the diagram a manageable size) work to develop a high-level understanding of the passage as a whole.

The second transformer adds two other bits of context: it clarifies that "bank" refers to a financial institution rather than a riverbank, and that "his" is a pronoun that refers to John. The second transformer produces another set of hidden state vectors that reflect everything the model has learned up to that point.

Real LLMs tend to have a lot more than two layers. The most powerful version of GPT-3, for example, has 96 layers. For the input, the most powerful version of GPT-3 uses word vectors with 12,288 dimensions—that is, each word is represented by a list of 12,288 numbers. That's 20 times larger than Google's 2013 word2vec scheme. The goal is for the 96th and final layer of the network to output a hidden state for the final word that includes all of the information necessary to predict the next word.

For example, suppose we changed our diagram above to depict a 96-layer language model interpreting a 1,000-word story. The 60th layer might include a vector for **John** with a parenthetical comment like "(main character, male, married to Cheryl, cousin of Donald, from Minnesota, currently in Boise, trying to find his missing wallet)." Again, all of these facts (and probably a lot more) would somehow be encoded as a list of 12,288 numbers corresponding to the word **John**. Or perhaps some of this information might be encoded in the 12,288-dimensional vectors for **Cheryl**, **Donald**, **Boise**, **wallet**, or other words in the story.

5.7 A Short History of Transformers

First described in a 2017 paper from Google, transformers are among the newest and one of the most powerful classes of models invented to date. They're driving a wave of advances in machine learning some have dubbed transformer AI.

Stanford researchers called transformers "foundation models" in an August 2021 paper because they see them driving a paradigm shift in AI. The "sheer scale and scope of foundation models over the last few years have stretched our imagination of what is possible," they wrote.

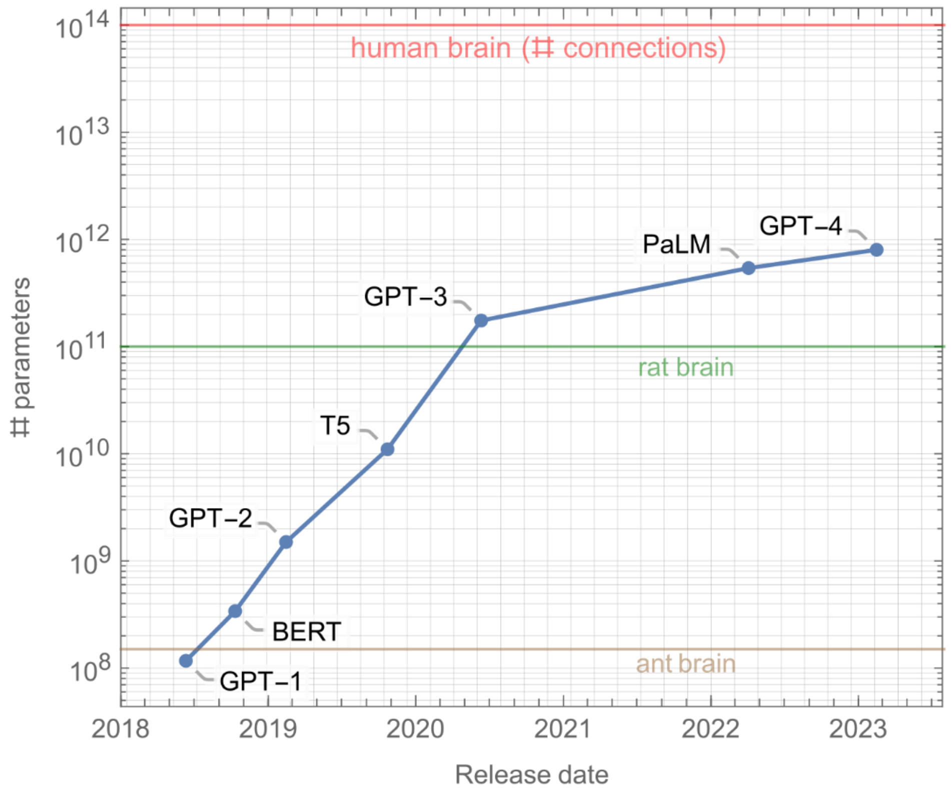
A year later, another Google team tried processing text sequences both forward and backward with a transformer. That helped capture more relationships among words, improving the model's ability to understand the meaning of a sentence.

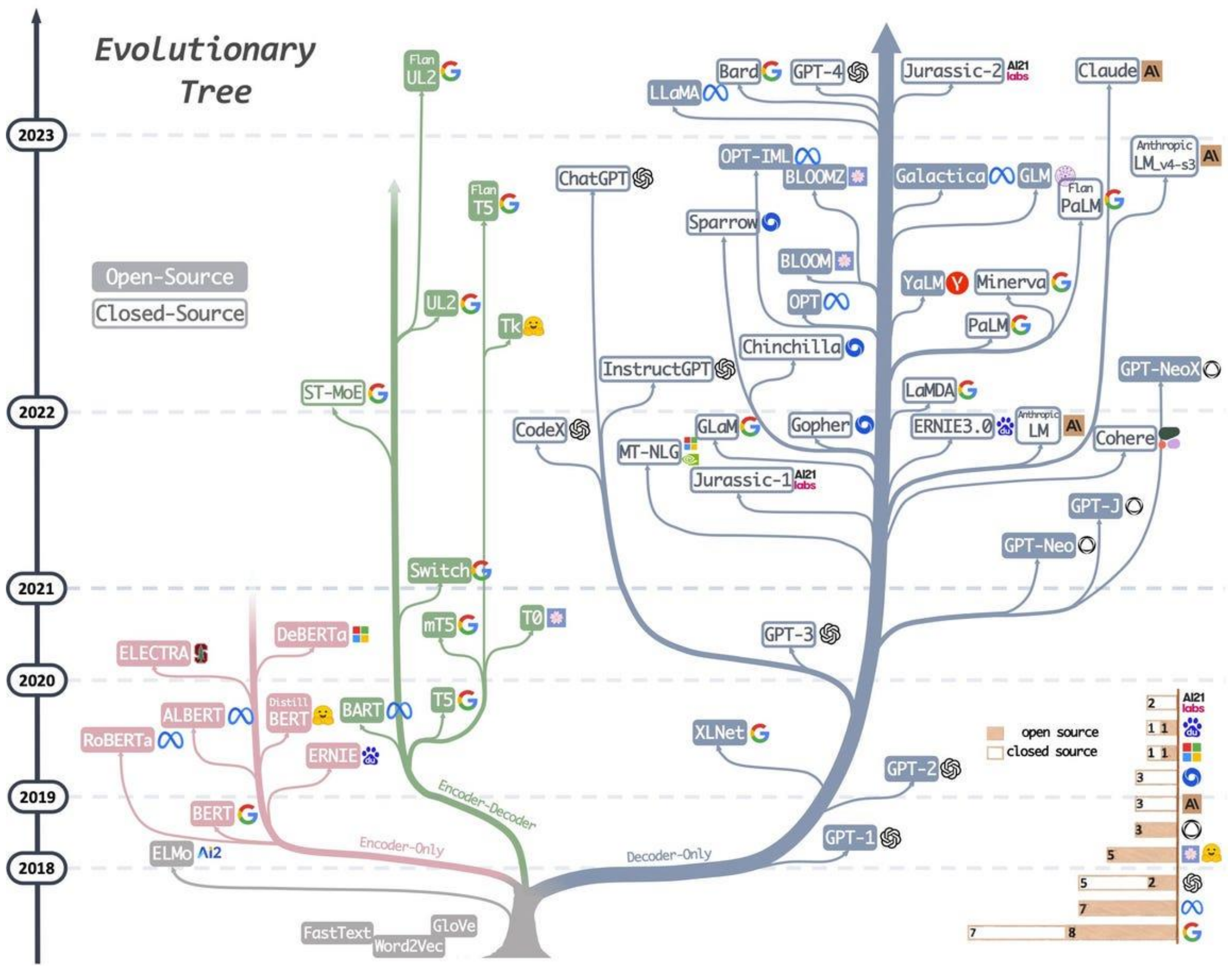
Their Bidirectional Encoder Representations from Transformers (BERT) model set 11 new records and became part of the algorithm behind Google search.

Within weeks, researchers around the world were adapting BERT for use cases across many languages and industries "because text is one of the most common data types companies have," said Anders Arpteg, a 20-year veteran of machine learning research.

Compared with traditional recurrent neural networks (RNNs) and convolutional neural networks (CNNs), transformers differ in their ability to capture long-range dependencies and contextual information.

The transformer "requires less training time than previous recurrent neural architectures, such as long short-term memory (LSTM), and its later variation has been prevalently adopted for training large language models on large (language) datasets," notes Wikipedia.





Appendix

To calculate the cross-entropy between two probability distributions represented as vectors, you can use the following formula:

$$H(p, q) = -\sum_{i=1}^{n} p_i \log(q_i)$$

Where:

- $H(p, q)$ is the cross-entropy between the distributions.
- p and q are probability distributions represented as vectors.
- p_i and q_i are the probabilities of the i-th event from the distributions p and q, respectively.

Here's how you can calculate it step by step:

1. Ensure that both vectors represent valid probability distributions. This means that each element in the vectors should be non-negative and sum up to 1.
2. Iterate over each element of the vectors.
3. Multiply the corresponding elements of the p and q vectors.
4. Take the natural logarithm (base e) of the q_i values.
5. Multiply p_i by $-\log(q_i)$
6. Sum up the results from step 5 over all elements.

Check the Python function to compute the cross-entropy between two vectors.

Reference

Grammar as a Foreign Language: https://arxiv.org/abs/1412.7449

Batch Norm: https://arxiv.org/abs/2003.07845

Layer Norm: https://arxiv.org/abs/1607.06450

Transformer: https://arxiv.org/pdf/1706.03762.pdf

https://www.nvidia.com/en-us/glossary/large-language-models/

https://www.elastic.co/what-is/large-language-models

https://towardsdatascience.com/transformers-141e32e69591

https://developers.google.com/machine-learning/resources/intro-llms

https://newsletter.theaiedge.io/p/what-are-large-language-models

https://pytorch.org/tutorials/beginner/transformer_tutorial.html

https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial6/Transformers_and_MHAttention.html

https://www.datacamp.com/tutorial/building-a-transformer-with-py-torch https://towardsdatascience.com/illustrated-guide-to-transformers-step-by-step-explanation-f74876522bc0

https://www.tensorflow.org/text/tutorials/transformer