Machine Learning for Robotics: **Transformers**

Prof. Navid Dadkhah Tehrani

The original transformer paper [1] was published in 2017 and after the AlexNet [2]
paper which introduce CNNs, is the second ground-breaking architecture in deep learning.

The transformer paper targeting the text translation problem. And to understand the paper you need to know
the basic of text modeling such as tokenization and embedding that we covered in the RNN lecture.

Recall text translation with RNN requires encoder-decoder architecture:
The encoder compresses a long sentence into a latent representation which has the entire content of the sentence.
Then the decoder use that to translate.

The issue is the decoder must attend to different part of the original sentence while translating. Not just the word before!
Example:
The law will never be perfect but **its** application ...

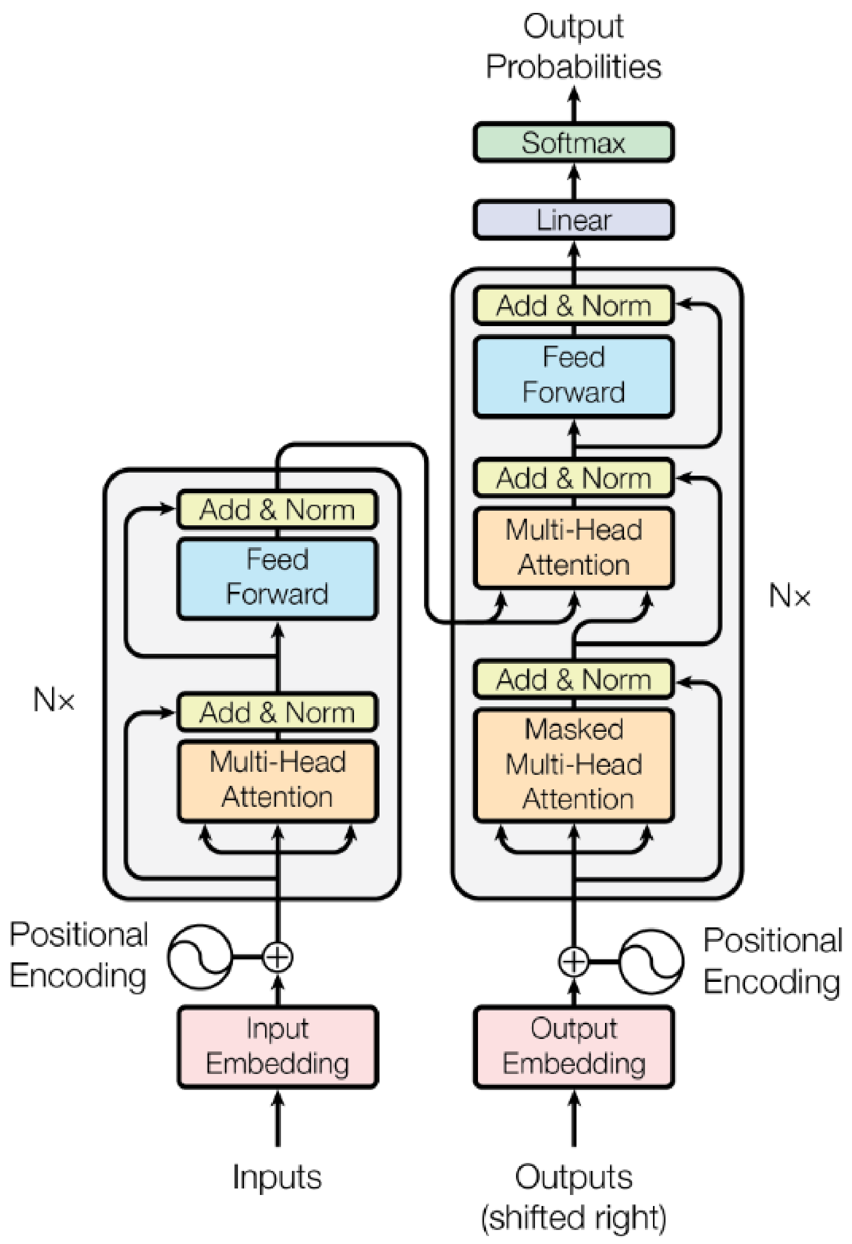
The vanilla encoder-decoder RNN does not have attention.
Researches spend a lot of time developing RNN architecture with attention.
The transformer paper, removed the RNN and kept the attention and it turned out that the RNN was not needed!.

[1] A. Vaswani et. al. “Attention Is All You Need”, 2017
[2] Alex Krizhevsky, et. al. “ImageNet Classification with Deep Convolutional Neural Networks”, 2012

The Transformer Architecture

It contains:

- Encoder-decoder
- Concept of Query-Key-Value
- Fully connected layers
- Multi-head attention
- Input embedding
- Layer normalization
- Positional encoding



Input embedding

The paper used embedding dimension of 512 (hyperparameter)
So each word is embedded into a 512 vector.

If maximum sentence length is, for example, 30, then the whole sentence is a 30*512 tensor.

Query, key , value

Imagine you’re searching google for vegetarian pizza recipe.
Each website that has pizza recipe is stored as a (key, value) pair. For example, the key being pizza name and the value being the content of the website.

When you search for “veggie pizza”, this is your query for the database.

Under the hood, the search engine takes your query which is an n-dimensional vector and see if matches with any key in the database.

To do that we calculate the dot product of query and key.

Then we multiply the values by this dot product. This is our attention; i.e how much we must attend to content of each website (to find what we searched for).

This is how to create learnable query, key and value (Here we assume max sentence length is 30)

$$\begin{tikzcd}[column sep=huge]
 & q_1 = \mathrm{x}W^q \rightarrow \in R^{1\times 512} \\
\text{Word \#1}, \ \mathrm{x} \in R^{1\times 512} \arrow[ur, "\times W^q \in R^{512\times 512}"] \arrow[r, "\times W^k \in R^{512\times 512}"] \arrow[dr, "\times W^v \in R^{512\times 512}"'] & k_1 = xW^k \\
 & v_1 = xW^v
\end{tikzcd}$$

$$\begin{tikzcd}[column sep=huge]
 & q_2 \\
\text{Word \#2} \in R^{1\times 512} \arrow[ur, "\times W^q \in R^{512\times 512}"] \arrow[r, "\times W^k \in R^{512\times 512}"] \arrow[dr, "\times W^v \in R^{512\times 512}"'] & k_2 \\
 & v_2
\end{tikzcd}$$

⋮

$$\begin{tikzcd}[column sep=huge]
 & q_{30} \\
\text{Word \#30} \in R^{1\times 512} \arrow[ur, "\times W^q \in R^{512\times 512}"] \arrow[r, "\times W^k \in R^{512\times 512}"] \arrow[dr, "\times W^v \in R^{512\times 512}"'] & k_{30} \\
 & v_{30}
\end{tikzcd}$$

To see how much word #1 must attend to itself, word 2 and 3, ...

$$A(q_1, K, V) = \sum_{i=1}^{30} \frac{\exp\left(q_1 . k_i^T\right)}{\sum q_1 . k_i^T} \times v_i \quad \rightarrow \quad \in R^{1\times 512}$$

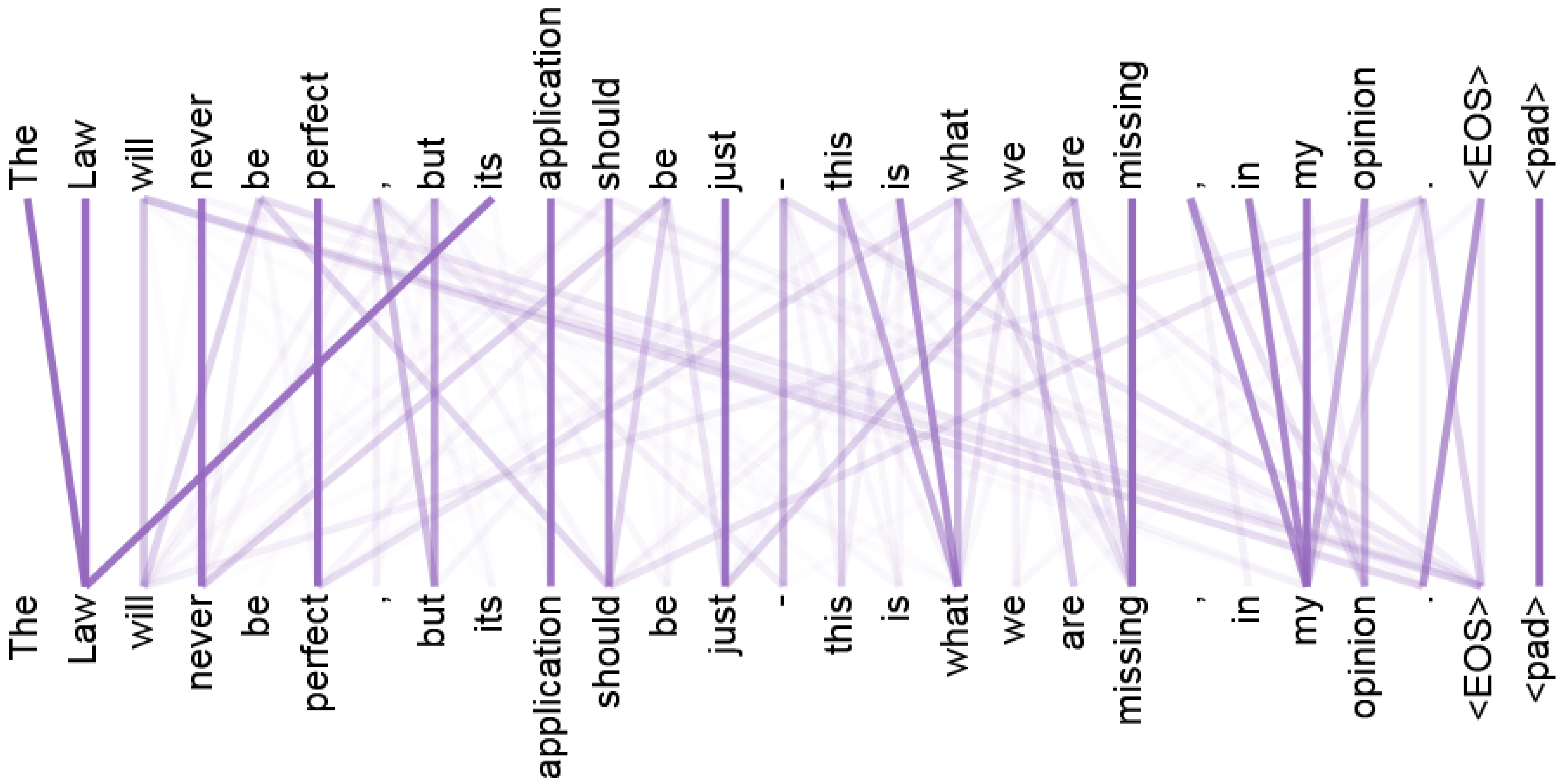
Similarly for word #2:

$$A(q_2, K, V) = \sum_{i=1}^{30} \frac{\exp\left(q_2 . k_i^T\right)}{\sum q_2 . k_i^T} \times v_i \rightarrow \quad \in R^{1\times 512}$$

⋮

and word #30:

$$A(q_{30}, K, V) = \sum_{i=1}^{30} \frac{\exp\left(q_{30} . k_i^T\right)}{\sum q_{30} . k_i^T} \times v_i \rightarrow \quad \in R^{1\times 512}$$

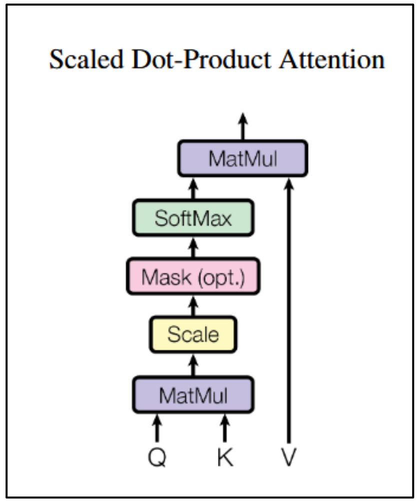

The
Law
will
never
be
perfect
,
but
its
application
should
be
just
-
this
is
what
we
are
missing
,
in
my
opinion
.
<EOS>
<pad>
The
Law
will
never
be
perfect
,
but
its
application
should
be
just
-
this
is
what
we
are
missing
,
in
my
opinion
.
<EOS>
<pad>

Now in vector form:

$a)$ $softmax(\frac{QK^T}{d_k})$: attention matrix, $\in R^{30\times 30}$ → this tells how much each work need to attend to the other word.

d_k=512 is for normalization to ensure QK^T doesn't get too big→ vanishing gradient

b) $softmax\left(\frac{QK^T}{d_k}\right)V$: attention embedding, $\in R^{30\times 512}$



Each element of (a) captures the relations between word (i,j)
Each row of (b) capture the relationship of word one w.r.t the entire context of the sentence.

$$\begin{bmatrix} a_1 & a_2 & \dots & \\ \dots & & & \\ & & & \\ & & & \end{bmatrix}_{30\times 30} \times \begin{bmatrix} 1^{st}\ value\ of\ 1^{st}\ word & \dots & \\ 1^{st}\ value\ of\ 2nd\ word & & \\ 1^{st}\ value\ of\ 3rd\ word & & \\ \dots & & \\ \dots & & \end{bmatrix}_{30\times 512} =$$

$$\begin{bmatrix} b_1 & \dots & \\ \dots & & \\ & & \\ & & \end{bmatrix}_{30\times 512}$$

a_1: attention of 1st word to itself
a_2: attention of 1st word to 2nd word
…
b_1: attention of 1st word to the 1st value of all words

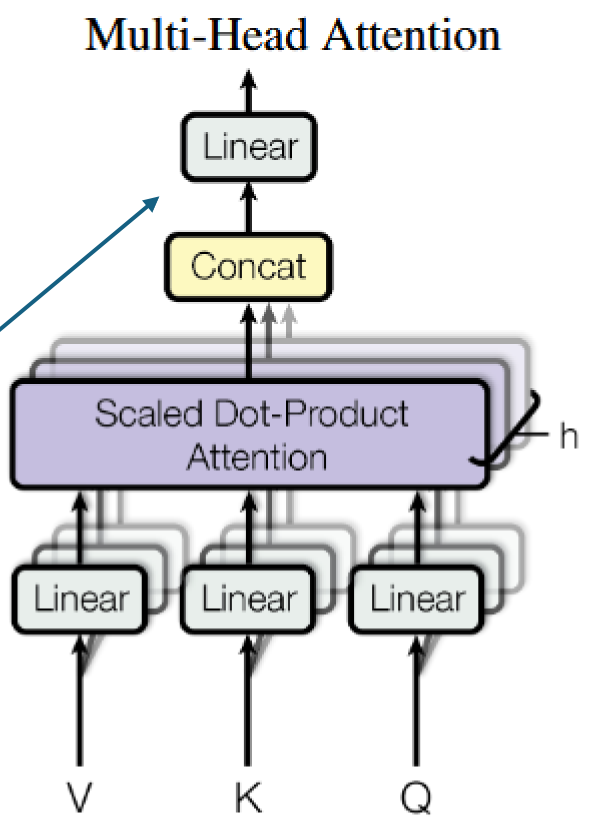
Multi-Head Attention

In the previous slides the dimension of $W^q, W^k, W^v \in R^{512\times512}$

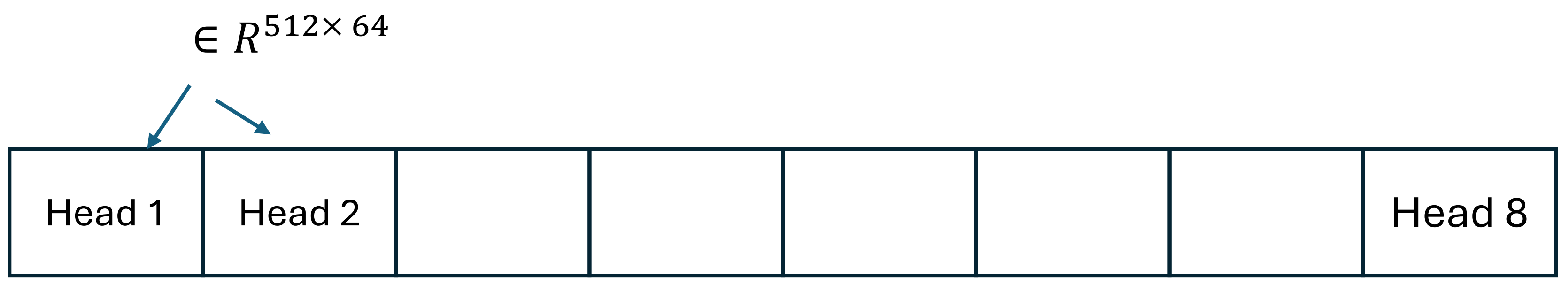
What if we make their dimension $R^{512\times64}$ but use 8 of these Ws?
(note that 8 is hyperparameter and what's used in the original paper)

That's the concept behind multi-head attention.
It Allows attending to different parts in the sequence differently.
At the end we concatenate all the "$softmax\left(\frac{QK^T}{d_k}\right)V$" together.

$$\mathrm{MultiHead}(Q,K,V) = \mathrm{Concat}(\mathrm{head_1}, ..., \mathrm{head_h})W^O$$
$$\text{where } \mathrm{head_i} = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$



The paper also used another linear matrix multiplication $W^O \in R^{512\times512}$ after the concatenation.



Probability of the translated word

2 MLP layers: 512 to 2048 → Relu → 2048 to 512)

Encoder and decoder are repeated 6 times with different weights, much like a Resnet

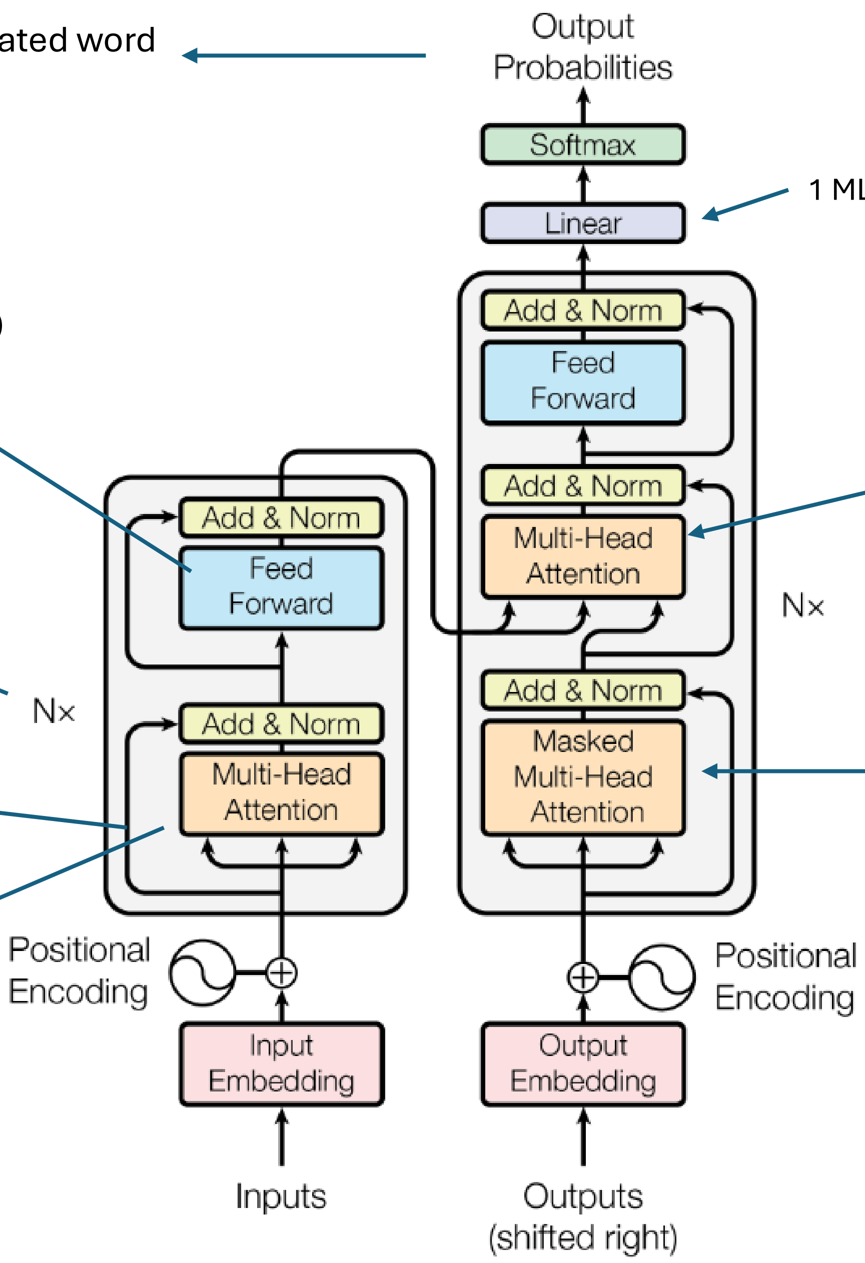
Skip connection like in Resnet

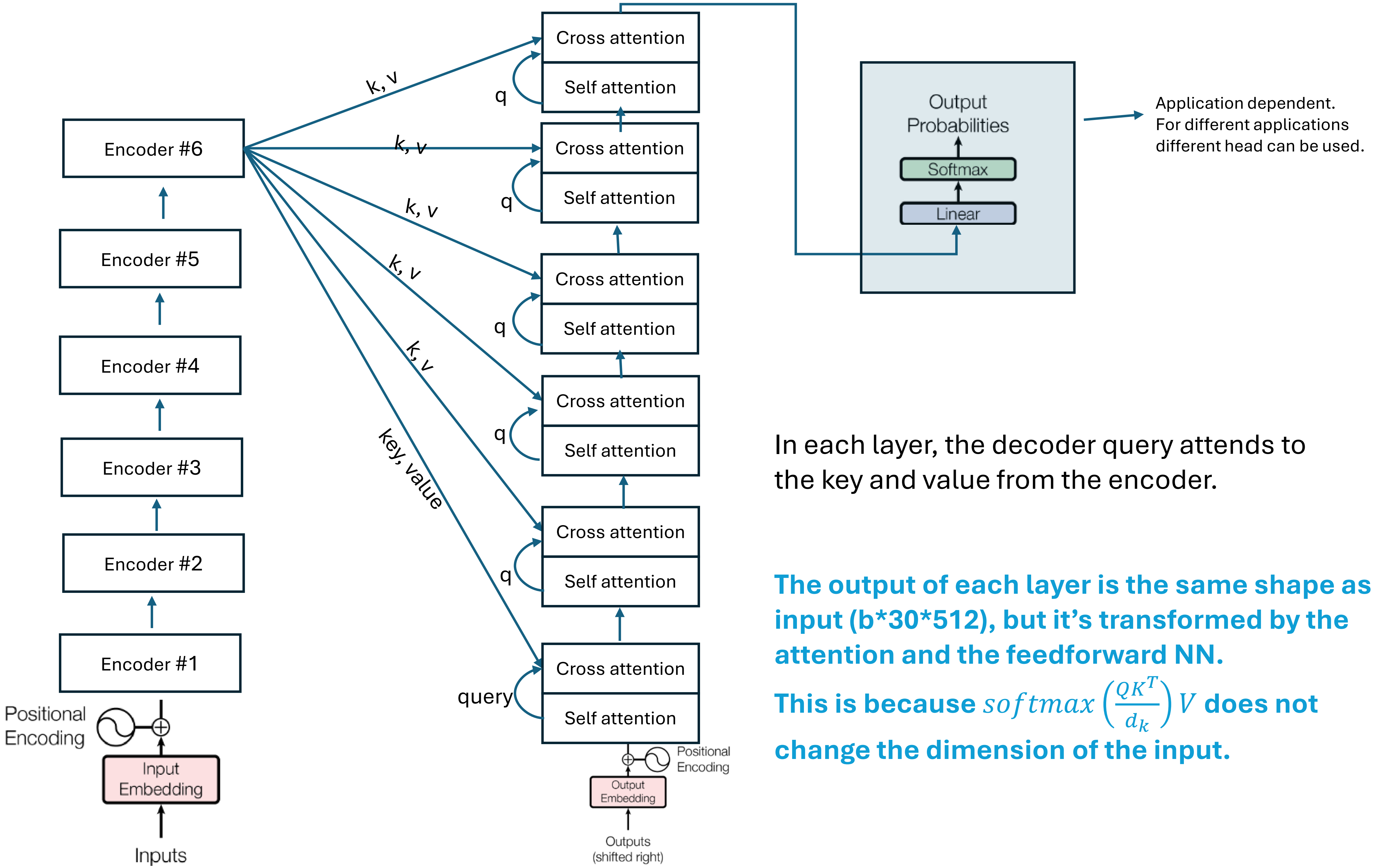
Encoder's self attention.

1 MLP layers: 512 to N where N is the size of target vocabulary

Cross attention: the decoder query attends to the key and value from the encoder

Decoder self attention to attend the already generated tokens.





In each layer, the decoder query attends to the key and value from the encoder.

The output of each layer is the same shape as input (b*30*512), but it's transformed by the attention and the feedforward NN.

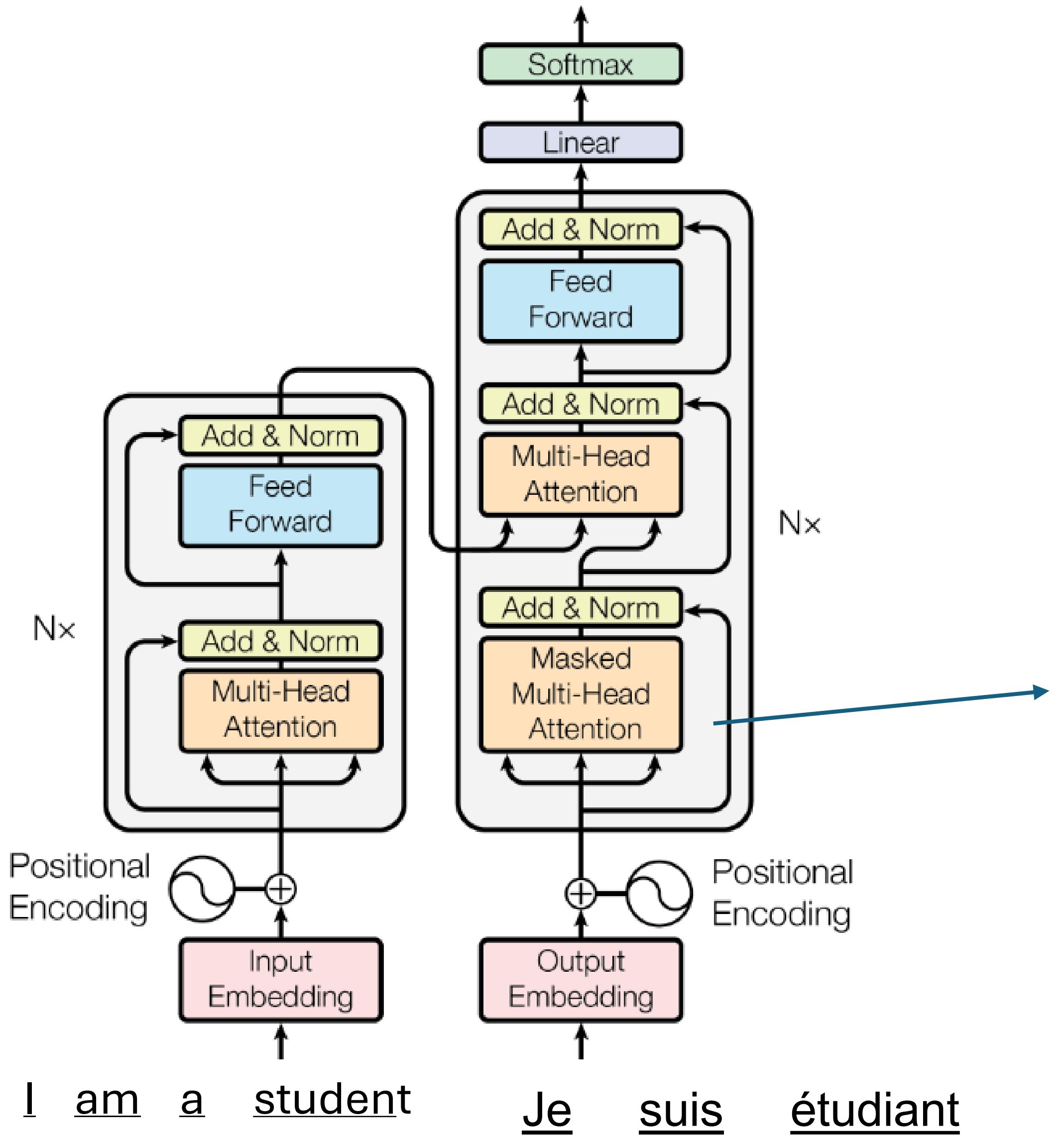
This is because $softmax\left(\frac{QK^T}{d_k}\right)V$ does not change the dimension of the input.

Predicts next word→ “suis”

Masked Multi-head attention

Mask subsequent sequence elements. I.e., only allow to attend to positions up to and including the current position.

In software this is done by setting softmax values for those to –inf.



Positional Encoding

- Provides information about the order of tokens in a sequence, allowing the model to understand the position of each token.
- Sinusoidal positional encoding is a vector of small constants values added to the embeddings.
- This means that if the embedding dimension is 512 and max sentence length is 30, then the positional embedding for each word has 512 dimension.
- Therefore, the entire positional encoding tensor is of dimension 30*512.
- The positional encoding is unique for each position in the sequence. This means that the positional encoding for the first word (position 1) is different from the positional encoding for the second word (position 2), and so on.

$$PE_{(pos,2i)} = \sin\frac{pos}{1000^{\frac{2i}{512}}}, \qquad i = 1, \dots 512$$

$$PE_{(pos,2i+1)} = \cos\frac{pos}{1000^{\frac{2i+1}{512}}}, \quad i = 1, \dots, 512$$

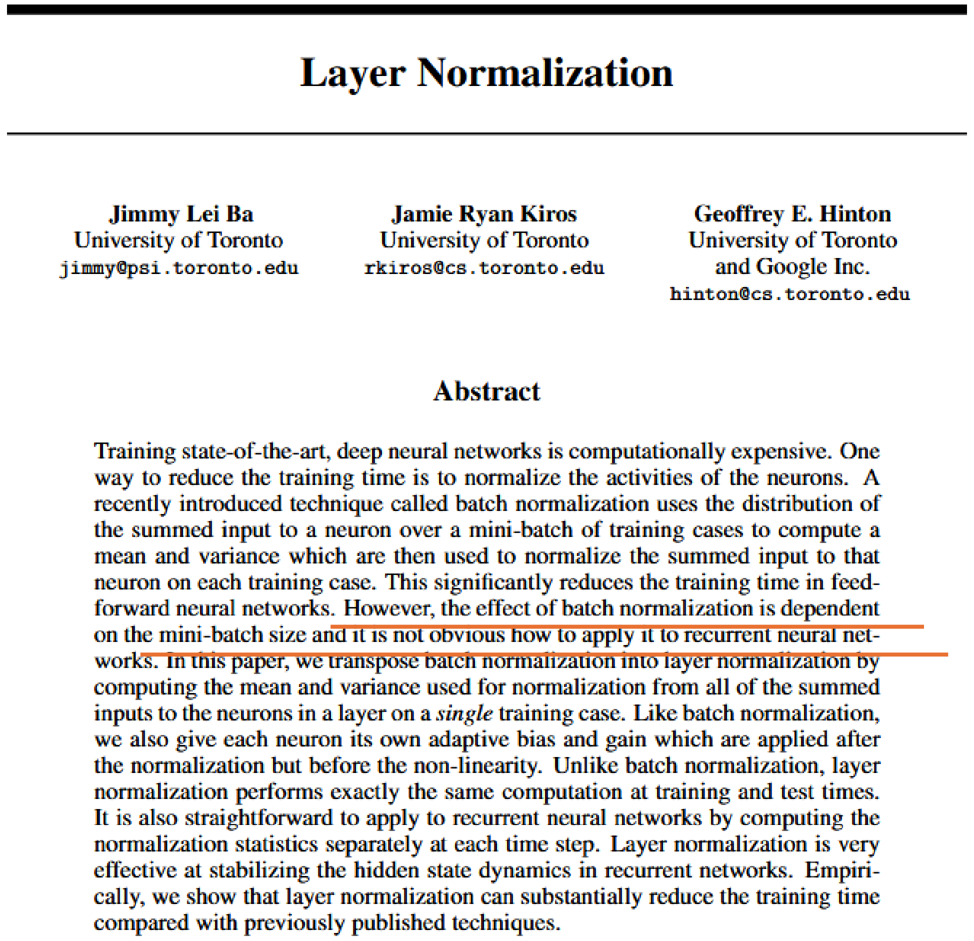
i: location within the embedding

pos: is the position of the token in the sequence

“i” essentially iterates over the dimensions of the embedding vector, and different frequencies of sine and cosine functions are applied to create a unique positional encoding for each dimension.

Layer Normalization

batch normalization does not work effectively with self-attention mechanisms.
batch normalization struggles with sequential data, which is a fundamental aspect of transformer models.
Introduced in this 2016 paper:



Layer Normalization

Jimmy Lei Ba
University of Toronto
jimmy@psi.toronto.edu

Jamie Ryan Kiros
University of Toronto
rkiros@cs.toronto.edu

Geoffrey E. Hinton
University of Toronto
and Google Inc.
hinton@cs.toronto.edu

Abstract

Training state-of-the-art, deep neural networks is computationally expensive. One way to reduce the training time is to normalize the activities of the neurons. A recently introduced technique called batch normalization uses the distribution of the summed input to a neuron over a mini-batch of training cases to compute a mean and variance which are then used to normalize the summed input to that neuron on each training case. This significantly reduces the training time in feed-forward neural networks. However, the effect of batch normalization is dependent on the mini-batch size and it is not obvious how to apply it to recurrent neural networks. In this paper, we transpose batch normalization into layer normalization by computing the mean and variance used for normalization from all of the summed inputs to the neurons in a layer on a *single* training case. Like batch normalization, we also give each neuron its own adaptive bias and gain which are applied after the normalization but before the non-linearity. Unlike batch normalization, layer normalization performs exactly the same computation at training and test times. It is also straightforward to apply to recurrent neural networks by computing the normalization statistics separately at each time step. Layer normalization is very effective at stabilizing the hidden state dynamics in recurrent networks. Empirically, we show that layer normalization can substantially reduce the training time compared with previously published techniques.

Layer Normalization

Shape of input to the layer norm: batch*30*512
shape of output of the layer norm: batch*30*512

the mean and standard deviation (μ, σ) are calculated across the embedding dimension (512)
for each token in the sequence (30 of them)

Then each element of the embedding has its own learnable γ and β that scales and shift its distribution after the normalization (like what we had in batch norm)

$$LN(X) = \frac{X - \mu}{\sigma + 0.001} * \gamma + \beta$$

Perhaps one of the reasons that batch norm does not work well here is that each sentence in a batch has different sizes that are padded to be the same size.

During evaluation

Unlike training, where the target sentence is masked to prevent the model from "seeing" future tokens,
in evaluation, there is no need for this mask because we only use previously generated tokens up to the current step.

Autoregressive Generation: the target sentence is generated autoregressively,
meaning that it’s produced one token at a time.
At each decoding step, the Transformer uses all tokens generated so far as input to predict the next token.

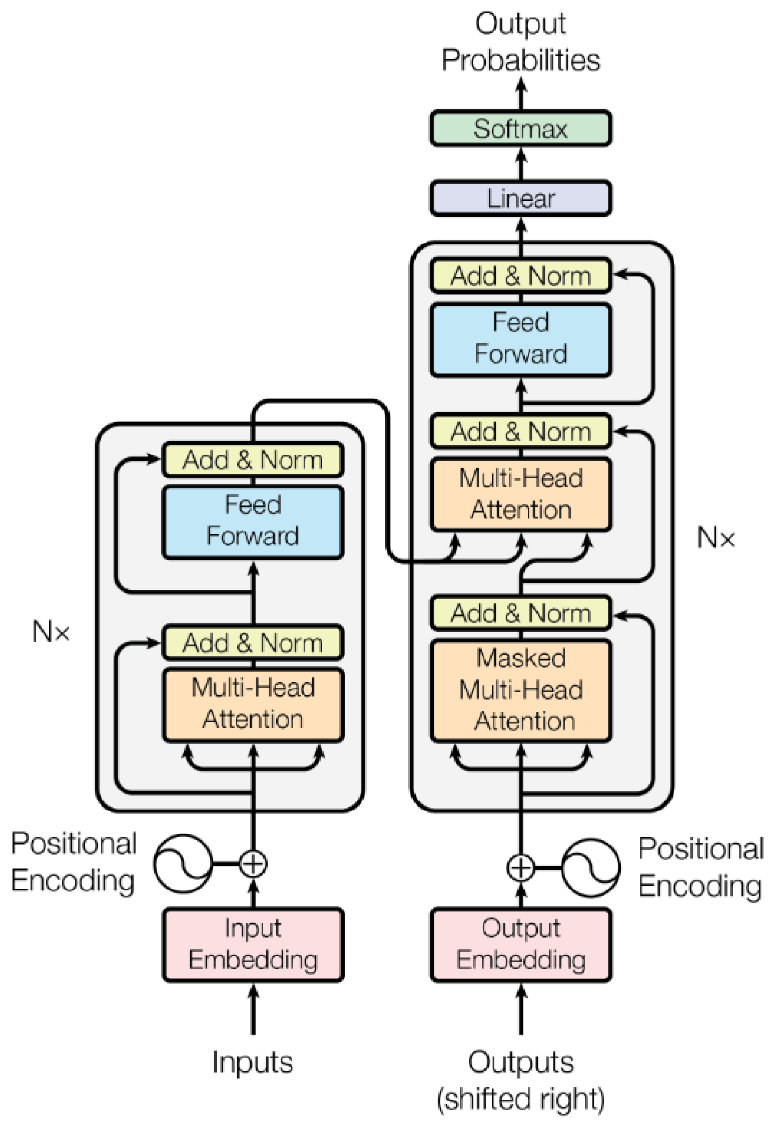
Key observations

The output of each layer (either encoder or decoder) is the same shape as input (in our example: b*30*512), but it's transformed (i.e. transformer ☺).

As opposed to MLP and CNN that change the dimension of the input, the transformer does not change the dimension of the input. It just transforms it.

In some applications, we only use the transformer encoder and we process its output with other neural networks such as MLP or CNN.

And in some application like the text translation here we use both transformer encoder and decoder.


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)