



UE21CS343BB2

Topics in Deep Learning

Dr. Shylaja S S

Director of Cloud Computing & Big Data (CCBD), Centre
for Data Sciences & Applied Machine Learning (CDSAML)
Department of Computer Science and Engineering
shylaja.sharath@pes.edu

**Ack: Anashua Dastidar,
Teaching Assistant**

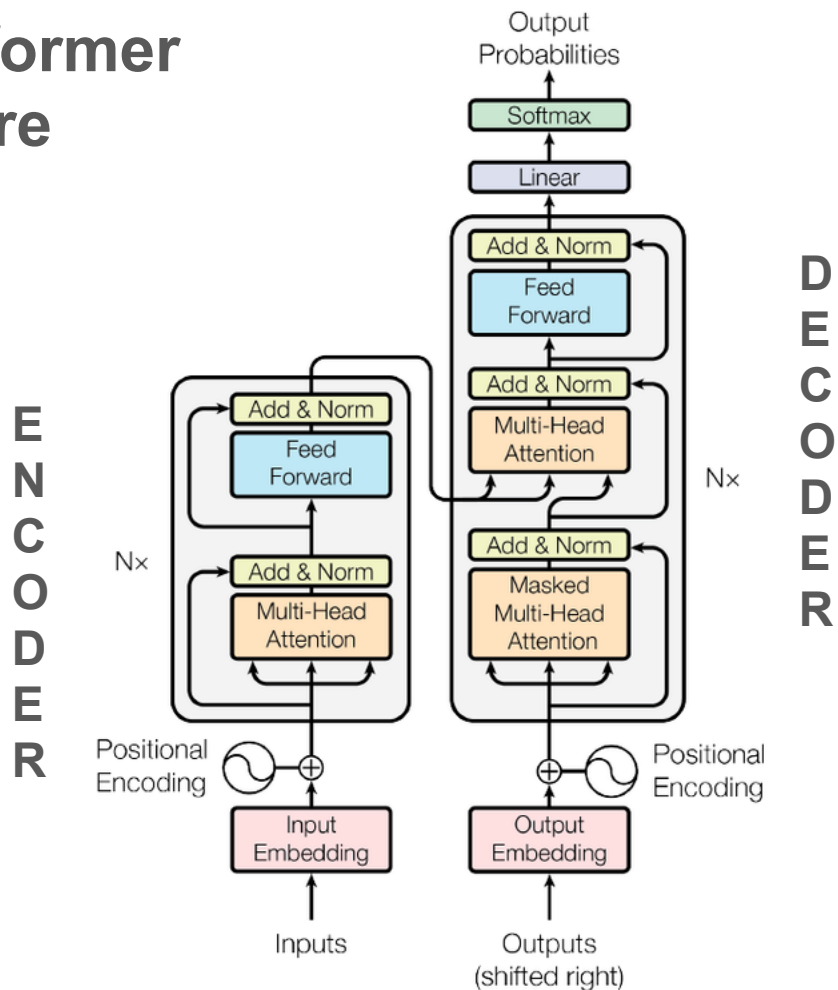
A small recap

Previously we have seen two approaches to solving the NMT or Neural Machine Translation task .

1. In the first approach we summarize the entire sentence into a single hidden state (i.e the context vector). We quickly see that the context vector becomes a bottleneck in this process.
1. In the second approach we use a weighted sum of all the hidden states along with the context vector to generate an output at every decoder timestep.

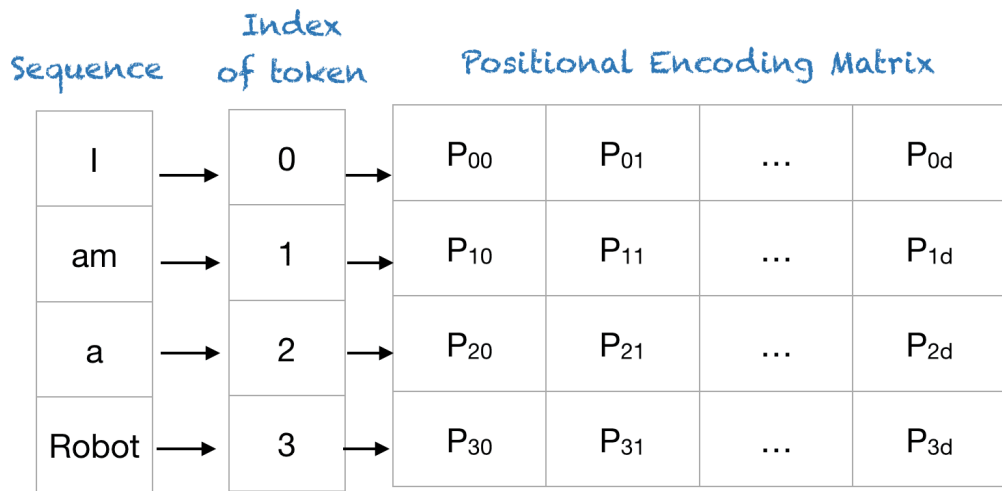
In the paper titled [Attention is All You Need](#) by Vaswani et al. the authors argue that the inherent sequential nature of RNNs precludes parallelization within training examples as parallelization is critical at longer sequence lengths . The propose to do way with the RNN architecture and propose a new architecture popularly known as the transformer.

The Transformer Architecture



Transformer architecture : Positional embeddings

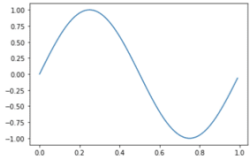
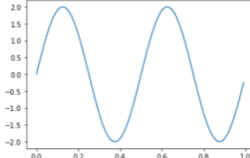
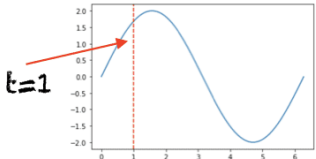
- The input to both the encoder and decoder to a transformer cell are a list word embeddings which represent a sentence. As we are not using a sequential algorithm like an RNN , we must find a method to encode positional information about the relative or absolute position of the tokens in a sequence.
- Transformers use a smart positional encoding scheme, where each position/index is mapped to a vector. Hence, the output of the positional encoding layer is a matrix, where each row of the matrix represents an encoded object of the sequence summed with its positional information.



Positional Encoding Matrix for the sequence 'I am a robot'

- These vectors follow a specific pattern that the model learns, which helps it determine the position of each word, or the distance between different words in the sequence.
- The intuition here is that adding these values to the embeddings provides meaningful distances between the embedding.

A quick recap of the trigonometric Sine function

Equation	Graph	Frequency	Wavelength
$\sin(2\pi t)$		1	1
$\sin(2 * 2\pi t)$		2	1/2
$\sin(t)$		$1/2\pi$	2π
$\sin(ct)$	Depends on c	$c/2\pi$	$2\pi/c$

Transformer architecture : Positional embeddings

Say we wish to calculate the positional embedding of a sequence of length L and we require the position for the k^{th} object in this sequence.

Here,

$d \rightarrow$:Dimension of the output embedding space

$n \rightarrow$ User-defined scalar, set to 10,000 by the authors

$i \rightarrow$ Used for mapping to column indices $0 < i < d/2$, with a single value of i maps to both sine and cosine functions

$$P(k, 2i) = \sin\left(\frac{k}{n^{2i/d}}\right)$$

$$P(k, 2i + 1) = \cos\left(\frac{k}{n^{2i/d}}\right)$$

Let us understand with an example ...

Positional Encoding example

To understand the previous expression, let's take an example of the phrase "I am a robot," with $n=100$ and $d=4$. The following table shows the positional encoding matrix for this phrase. In fact, the positional encoding matrix would be the same for any four-letter phrase with $n=100$ and $d=4$.

Sequence	Index of token, k	Positional Encoding Matrix with $d=4$, $n=100$			
		$i=0$	$i=0$	$i=1$	$i=1$
I	0	$P_{00}=\sin(0)$ = 0	$P_{01}=\cos(0)$ = 1	$P_{02}=\sin(0)$ = 0	$P_{03}=\cos(0)$ = 1
am	1	$P_{10}=\sin(1/1)$ = 0.84	$P_{11}=\cos(1/1)$ = 0.54	$P_{12}=\sin(1/10)$ = 0.10	$P_{13}=\cos(1/10)$ = 1.0
a	2	$P_{20}=\sin(2/1)$ = 0.91	$P_{21}=\cos(2/1)$ = -0.42	$P_{22}=\sin(2/10)$ = 0.20	$P_{23}=\cos(2/10)$ = 0.98
Robot	3	$P_{30}=\sin(3/1)$ = 0.14	$P_{31}=\cos(3/1)$ = -0.99	$P_{32}=\sin(3/10)$ = 0.30	$P_{33}=\cos(3/10)$ = 0.96

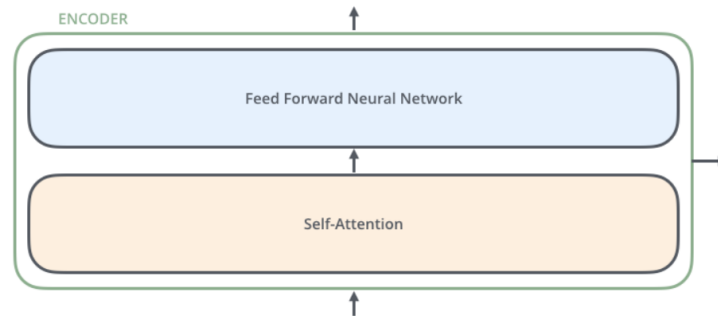
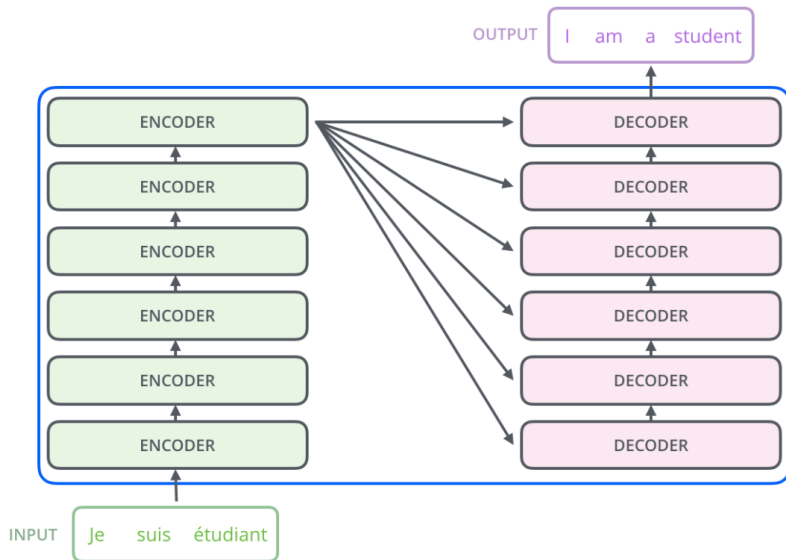
Positional Encoding Matrix for the sequence 'I am a robot'

As for each word we have a positional embedding that is of the same dimension as the word embedding (d) we can sum this with the word embedding.

This would add meaningful distances between the different word embeddings that can be learnt by the model

Transformer architecture : Encoder

The Transformer architecture used by Vaswani et al. contains 6 stacked encoders and decoders. The encoders are all identical in structure (yet they do not share weights). Each one is broken down into two sublayers:

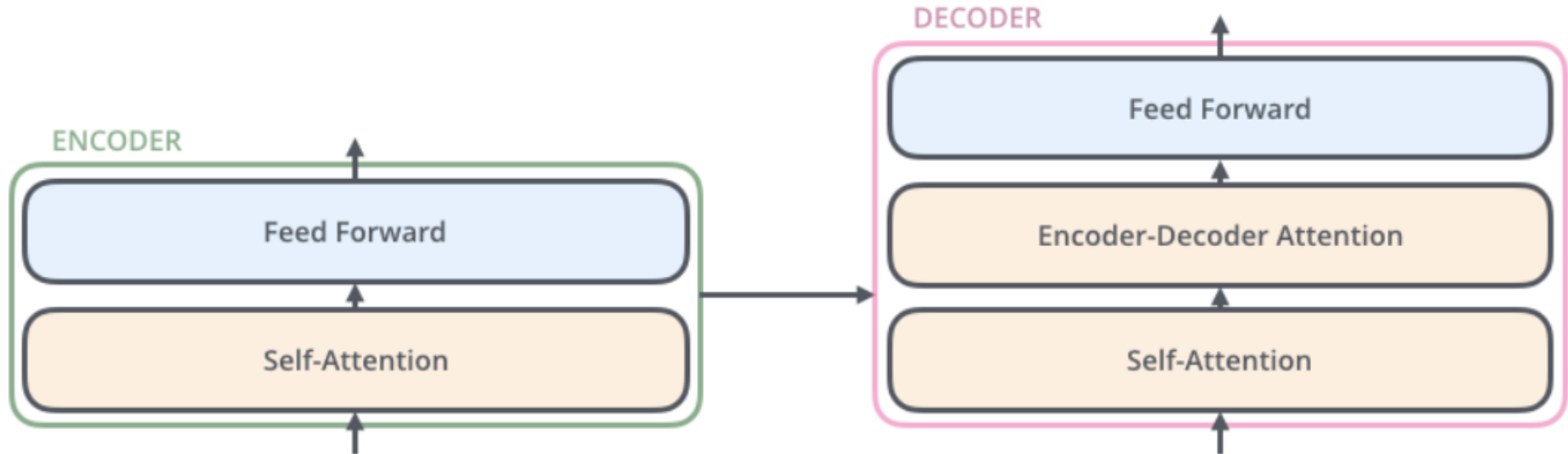


The **encoder's** inputs first flow through a self-attention layer – a layer that helps the encoder look at other words in the input sentence as it encodes a specific word.

The outputs of the self-attention layer are fed to a feed-forward neural network. The exact same feed-forward network is independently applied to each position.

Transformer architecture : Decoder

The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence (similar what attention does in seq2seq models in the previous lecture).

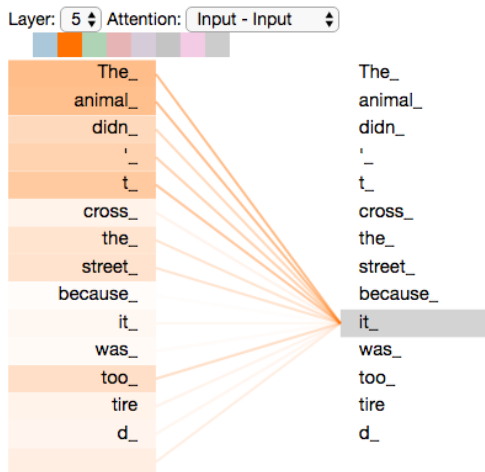


Transformer architecture: Self Attention at a high level

Say the following sentence is an input sentence we want to translate:

"The animal didn't cross the street because it was too tired"

What does "it" in this sentence refer to? Is it referring to the street or to the animal? It's a simple question to a human, but not as simple to an algorithm. When the model is processing the word "it", self-attention allows it to associate "it" with "animal".



As the model processes each word (each position in the input sequence), self attention allows it to look at other positions in the input sequence for clues that can help lead to a better encoding for this word.

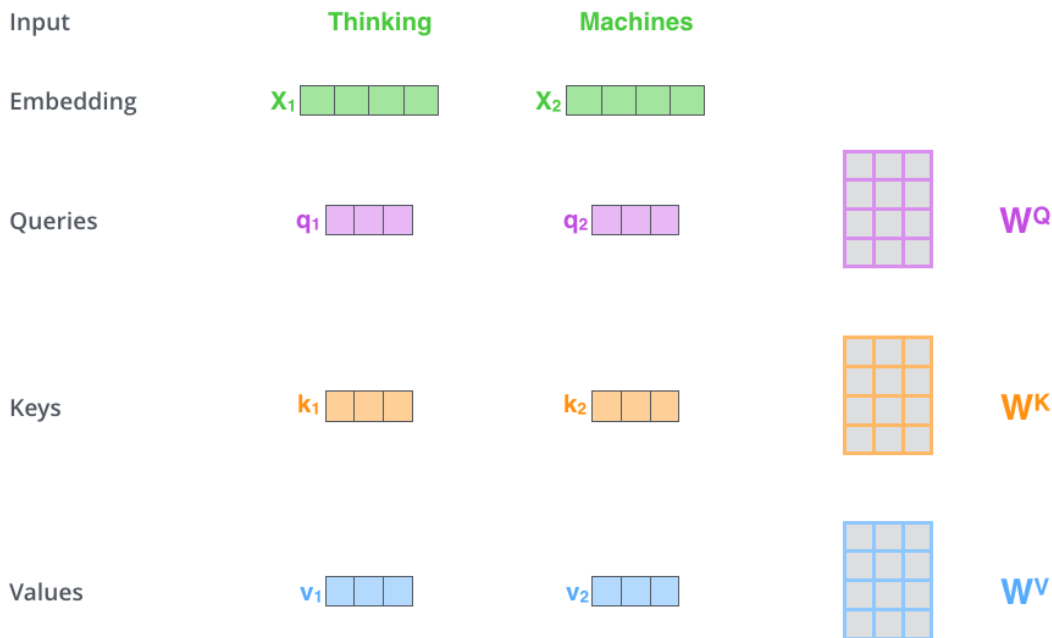
This is similar to the idea that maintaining a hidden state allows an RNN to incorporate its representation of previous words/vectors it has processed with the current one it's processing. Self-attention is the method the Transformer uses to bake the "understanding" of other relevant words into the one we're currently processing.

In the following slides we shall first understand how self attention is implemented using vectors and then see how it is actually implemented using matrices

Transformer architecture: Self Attention calculation step 1

The **first step** in calculating self-attention is to create three vectors from each of the encoder's input vectors .

So for each **word**, we create a **Query vector (qi)** , a **Key vector (ki)** , and a **Value vector(vi)** . These vectors are created by multiplying the embedding by three matrices that we trained during the training process.



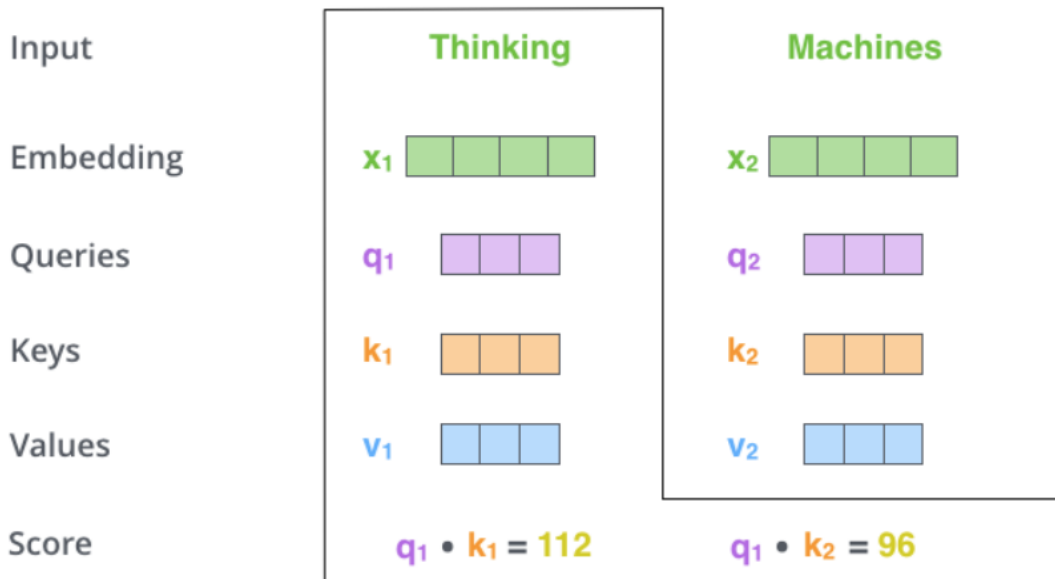
Notice that these new vectors are smaller in dimension than the embedding vector.

Their dimensionality is 64, while the embedding and encoder input/output vectors have dimensionality of 512.

They don't HAVE to be smaller, this is an architecture choice to make the computation of multi headed attention (mostly) constant.

Transformer architecture: Self Attention calculation step 2

The **second step** in calculating self-attention is to calculate a score. Say we're calculating the self-attention for the first word in this example, "Thinking". We need to score each word of the input sentence against this word. The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.

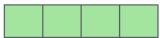
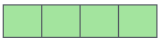
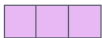
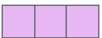
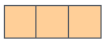
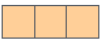
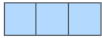
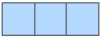


The score is calculated by taking the dot product of the **query vector** with the **key vector** of the respective word we're scoring.

So if we're processing the self-attention for the word in position **#1**, the first score would be the dot product of q_1 and k_1 . The second score would be the dot product of q_1 and k_2 .

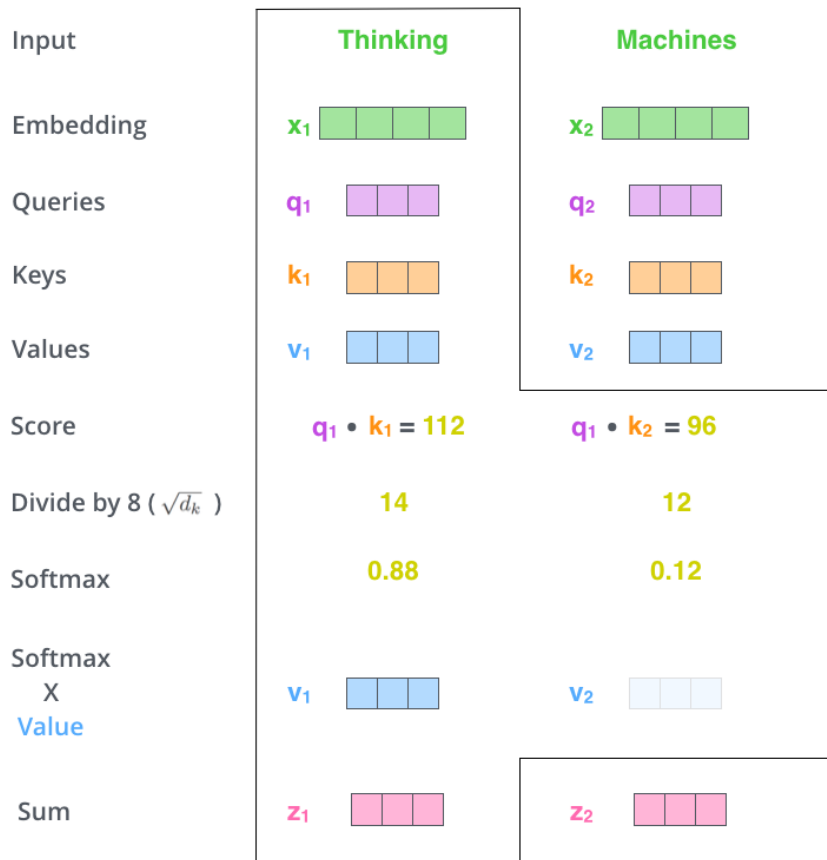
Transformer architecture: Self Attention calculation step 3 & 4

The **third and fourth steps** are to divide the scores by 8 (the square root of the dimension of the key vectors used in the paper – 64. This leads to having more stable gradients. There could be other possible values here, but this is the default), then pass the result through a softmax operation. Softmax normalizes the scores so they're all positive and add up to 1.

Input	Thinking	Machines
Embedding	x_1 	x_2 
Queries	q_1 	q_2 
Keys	k_1 	k_2 
Values	v_1 	v_2 
Score	$q_1 \cdot k_1 = 112$	$q_1 \cdot k_2 = 96$
Divide by $8 (\sqrt{d_k})$	14	12
Softmax	0.88	0.12

This softmax score determines how much each word will be expressed at this position. Clearly the word at this position will have the highest softmax score, but sometimes it's useful to attend to another word that is relevant to the current word.

Transformer architecture: Self Attention calculation step 5 & 6



The **fifth step** is to multiply each value vector by the softmax score (in preparation to sum them up). The intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words (by multiplying them by tiny numbers like 0.001, for example).

The **sixth step** is to sum up the weighted value vectors. This produces the output of the self-attention layer at this position (for the first word).

That concludes the self-attention calculation. The resulting vector is one we can send along to the feed-forward neural network. In the actual implementation, however, this calculation is done in matrix form for faster processing.

Transformer architecture: Self Attention calculation using matrices

The first step is to calculate the Query, Key, and Value matrices. We do that by packing our embeddings into a matrix X , and multiplying it by the weight matrices we've trained (W^Q , W^K , W^V).

$$\begin{array}{c}
 \text{X} \\
 \begin{array}{|c|c|c|c|} \hline \square & \square & \square & \square \\ \hline \square & \square & \square & \square \\ \hline \square & \square & \square & \square \\ \hline \end{array}
 \end{array}
 \times
 \begin{array}{c}
 W^Q \\
 \begin{array}{|c|c|c|c|} \hline \square & \square & \square & \square \\ \hline \square & \square & \square & \square \\ \hline \square & \square & \square & \square \\ \hline \square & \square & \square & \square \\ \hline \end{array}
 \end{array}
 =
 \begin{array}{c}
 Q \\
 \begin{array}{|c|c|c|} \hline \square & \square & \square \\ \hline \square & \square & \square \\ \hline \square & \square & \square \\ \hline \end{array}
 \end{array}$$

$$\begin{array}{c}
 \text{X} \\
 \begin{array}{|c|c|c|c|} \hline \square & \square & \square & \square \\ \hline \square & \square & \square & \square \\ \hline \square & \square & \square & \square \\ \hline \end{array}
 \end{array}
 \times
 \begin{array}{c}
 W^K \\
 \begin{array}{|c|c|c|c|} \hline \square & \square & \square & \square \\ \hline \square & \square & \square & \square \\ \hline \square & \square & \square & \square \\ \hline \square & \square & \square & \square \\ \hline \end{array}
 \end{array}
 =
 \begin{array}{c}
 K \\
 \begin{array}{|c|c|c|} \hline \square & \square & \square \\ \hline \square & \square & \square \\ \hline \square & \square & \square \\ \hline \end{array}
 \end{array}$$

$$\begin{array}{c}
 \text{X} \\
 \begin{array}{|c|c|c|c|} \hline \square & \square & \square & \square \\ \hline \square & \square & \square & \square \\ \hline \square & \square & \square & \square \\ \hline \end{array}
 \end{array}
 \times
 \begin{array}{c}
 W^V \\
 \begin{array}{|c|c|c|c|} \hline \square & \square & \square & \square \\ \hline \square & \square & \square & \square \\ \hline \square & \square & \square & \square \\ \hline \square & \square & \square & \square \\ \hline \end{array}
 \end{array}
 =
 \begin{array}{c}
 V \\
 \begin{array}{|c|c|c|} \hline \square & \square & \square \\ \hline \square & \square & \square \\ \hline \square & \square & \square \\ \hline \end{array}
 \end{array}$$

Finally, since we're dealing with matrices, we can condense steps two through six in one formula to calculate the outputs of the self-attention layer.

$$\begin{array}{c}
 \text{Q} \\
 \begin{array}{|c|c|c|} \hline \square & \square & \square \\ \hline \square & \square & \square \\ \hline \square & \square & \square \\ \hline \end{array}
 \end{array}
 \times
 \begin{array}{c}
 K^T \\
 \begin{array}{|c|c|} \hline \square & \square \\ \hline \square & \square \\ \hline \square & \square \\ \hline \end{array}
 \end{array}$$

$$\text{softmax} \left(\frac{\quad}{\sqrt{d_k}} \right)$$

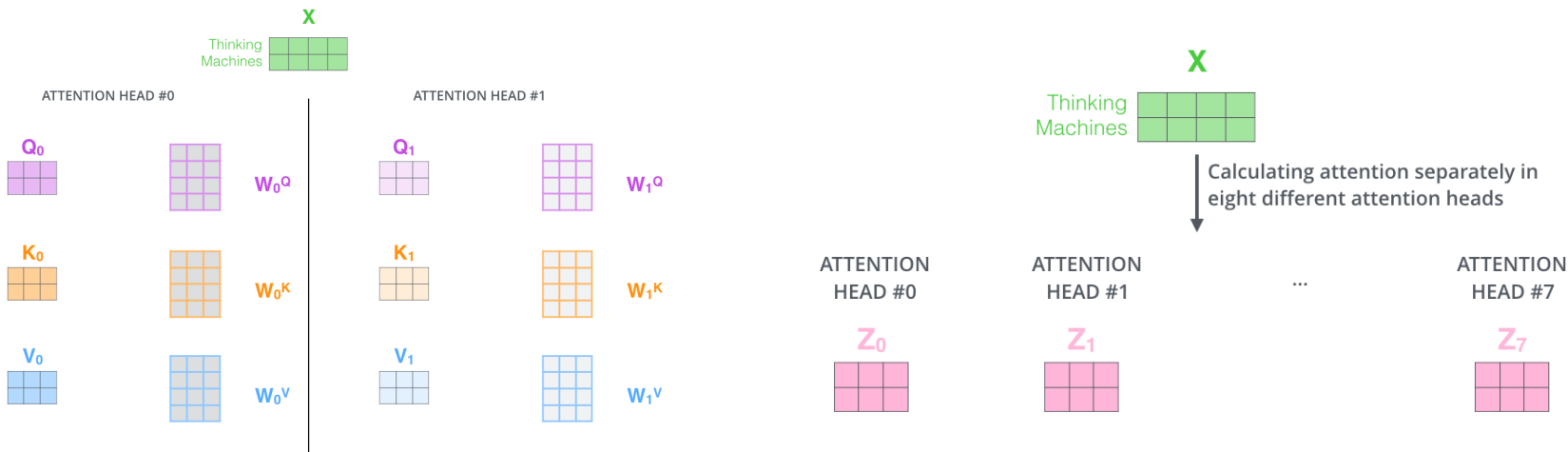
$$\begin{array}{c}
 V \\
 \begin{array}{|c|c|c|} \hline \square & \square & \square \\ \hline \square & \square & \square \\ \hline \square & \square & \square \\ \hline \end{array}
 \end{array}$$

$$=
 \begin{array}{c}
 Z \\
 \begin{array}{|c|c|c|} \hline \square & \square & \square \\ \hline \square & \square & \square \\ \hline \square & \square & \square \\ \hline \end{array}
 \end{array}$$

Transformer architecture: Multi head Attention

The paper further refined the self-attention layer by adding a mechanism called “multi-headed” attention. Instead of performing a single attention function with d dimensional keys, values and queries, they found it beneficial to linearly project the queries, keys and values h times with different, learned linear projections to d_k , d_k and d_v dimensions, respectively. On each of these projected versions of queries, keys and values they then performed the attention function in parallel, yielding d_v -dimensional output values.

I.e if we do the same self-attention calculation we outlined above, just eight different times with different weight matrices, we end up with eight different Z matrices



Transformer architecture: Multi head Attention

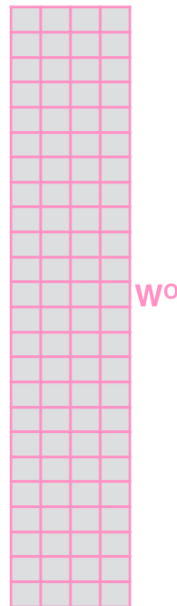
Now , The feed-forward layer is not expecting eight matrices – it's expecting a single matrix (a vector for each word). So we need a way to condense these eight down into a single matrix. So, we concat the matrices then multiply them by an additional weights matrix W^O

1) Concatenate all the attention heads

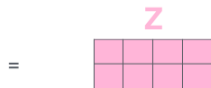


2) Multiply with a weight matrix W^O that was trained jointly with the model

x



3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN



Transformer architecture: Multi head Attention

This improves the performance of the attention layer in two ways:

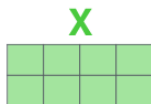
1. It expands the model's ability to focus on different positions. Yes, in the example above, z_1 contains a little bit of every other encoding, but it could be dominated by the actual word itself. If we're translating a sentence like "The animal didn't cross the street because it was too tired", it would be useful to know which word "it" refers to.
1. It gives the attention layer multiple "representation subspaces". With multi-headed attention we have not only one, but multiple sets of Query/Key/Value weight matrices (the Transformer uses eight attention heads, so we end up with eight sets for each encoder/decoder). Each of these sets is randomly initialized. Then, after training, each set is used to project the input embeddings (or vectors from lower encoders/decoders) into a different representation subspace.

A summary of multi headed attention..

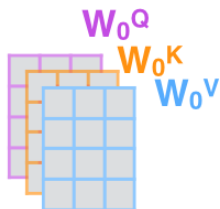
1) This is our
input sentence*

Thinking
Machines

2) We embed
each word*



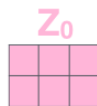
3) Split into 8 heads.
We multiply X or
 R with weight matrices



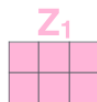
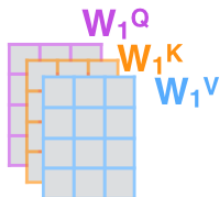
4) Calculate attention
using the resulting
 $Q/K/V$ matrices



5) Concatenate the resulting Z matrices,
then multiply with weight matrix W^O to
produce the output of the layer



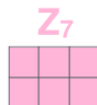
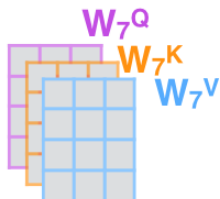
* In all encoders other than #0,
we don't need embedding.
We start directly with the output
of the encoder right below this one



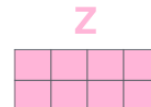
...

...

...

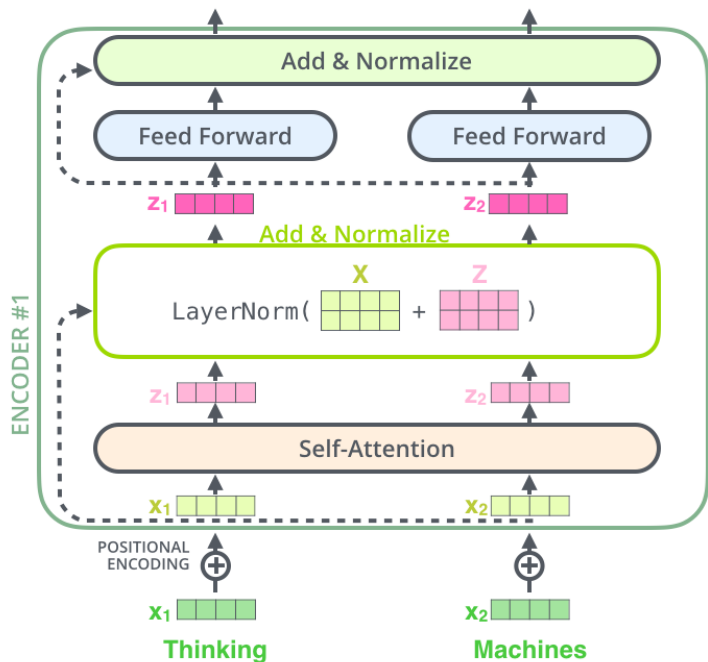


W^O

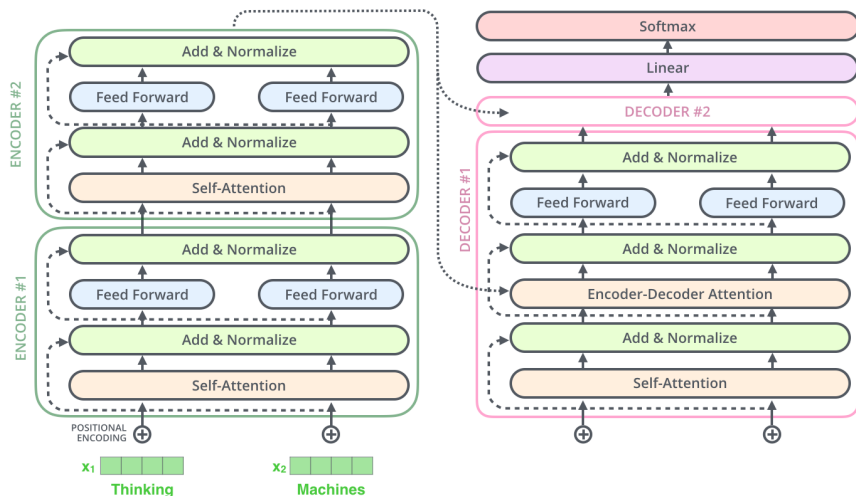


Transformer architecture : Encoder some finishing touches

One detail in the architecture of the encoder that we need to mention before moving on, is that each sub-layer (self-attention, ffn) in each encoder has a residual connection around it, and is followed by a layer-normalization step.



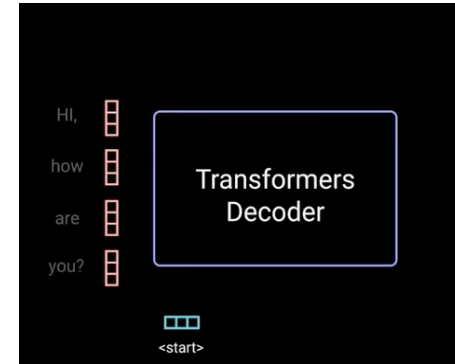
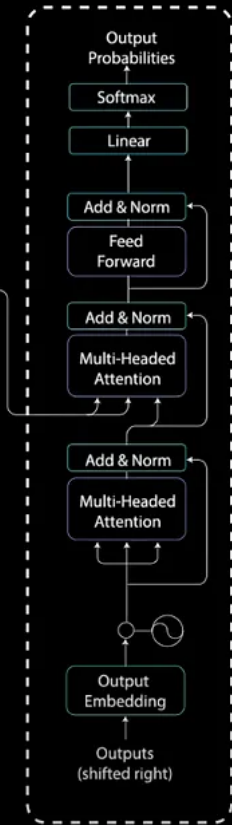
This goes for the sub-layers of the decoder as well. If we're to think of a Transformer of 2 stacked encoders and decoders, it would look something like this:



* Residual Connections have been seen before while discussing ResNet in chapter 1. This is the same idea !

Transformer architecture : The Decoder

- The decoder's job is to generate text sequences.
- The decoder has a similar sub-layer as the encoder. it has two multi-headed attention layers, a pointwise feed-forward layer, and residual connections, and layer normalization after each sub-layer.
- These sub-layers behave similarly to the layers in the encoder but each multi-headed attention layer has a different job.
- The decoder is capped off with a linear layer that acts as a classifier, and a softmax to get the word probabilities.
- The decoder is autoregressive, it begins with a start token, and it takes in a list of previous outputs as inputs, as well as the encoder outputs that contain the attention information from the input. The decoder stops decoding when it generates a token as an output.



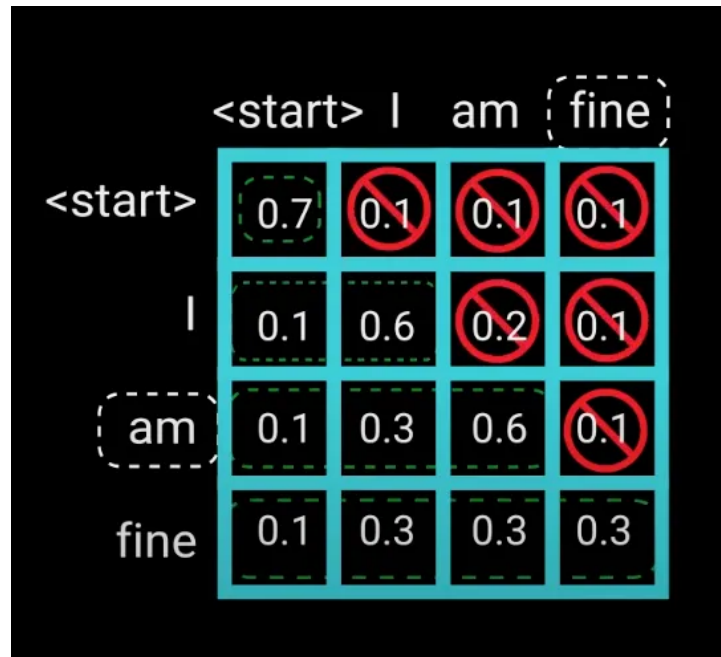
Transformer Architecture: Decoder Steps

Decoder Input Embeddings & Positional Encoding

The beginning of the decoder is pretty much the same as the encoder. The input goes through an embedding layer and positional encoding layer to get positional embeddings. The positional embeddings get fed into the first multi-head attention layer which computes the attention scores for the decoder's input.

Decoders First Multi-Headed Attention

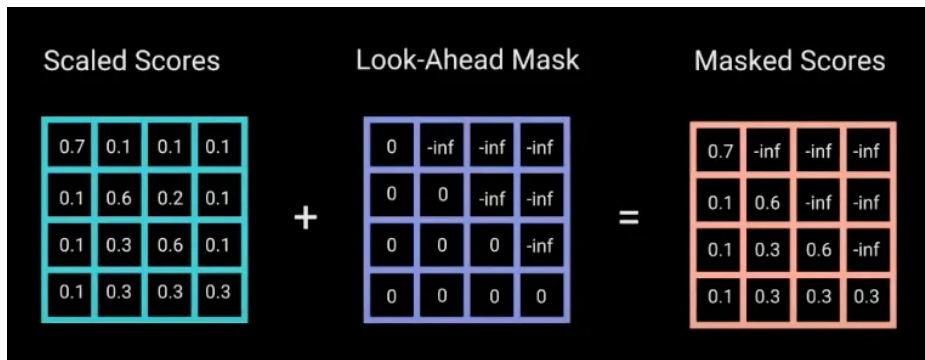
This multi-headed attention layer operates slightly differently. Since the decoder is autoregressive and generates the sequence word by word, you need to prevent it from conditioning to future tokens. For example, when computing attention scores on the word “am”, you should not have access to the word “fine”, because that word is a future word that was generated after. The word “am” should only have access to itself and the words before it. This is true for all other words, where they can only attend to previous words.



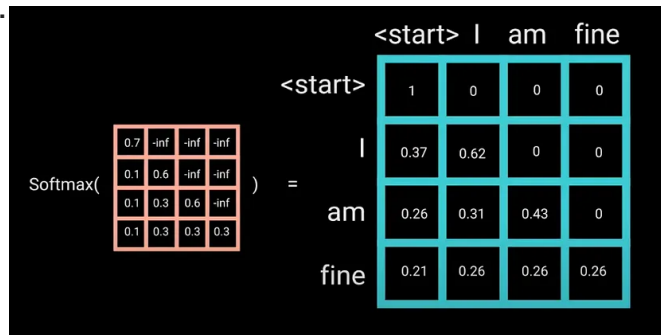
Transformer architecture : Masked Multi Head attention in the Decoder

We need a method to prevent computing attention scores for future words. This method is called masking. To prevent the decoder from looking at future tokens, you apply a look ahead mask. The mask is added before calculating the softmax, and after scaling the scores. Let's take a look at how this works.

Look-Ahead Mask: The mask is a matrix that's the same size as the attention scores filled with values of 0's and negative infinities. When you add the mask to the scaled attention scores, you get a matrix of the scores, with the top right triangle filled with negativity infinities.



The reason for the mask is because once you take the softmax of the masked scores, the negative infinities get zeroed out, leaving zero attention scores for future tokens.



*refer to [this discussion](#) on stack overflow to better understand the idea

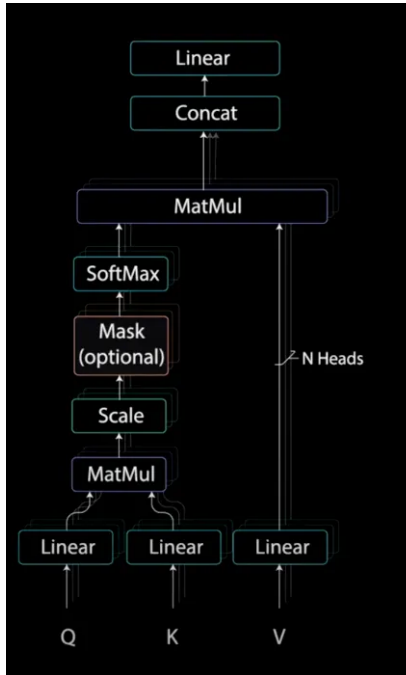
Transformer architecture : Masked Multi Head attention in the Decoder

This masking is the only difference in how the attention scores are calculated in the first multi-headed attention layer. This layer still has multiple heads, that the mask is being applied to, before getting concatenated and fed through a linear layer for further processing.

Decoder Second Multi-Headed Attention, and Pointwise Feed Forward Layer

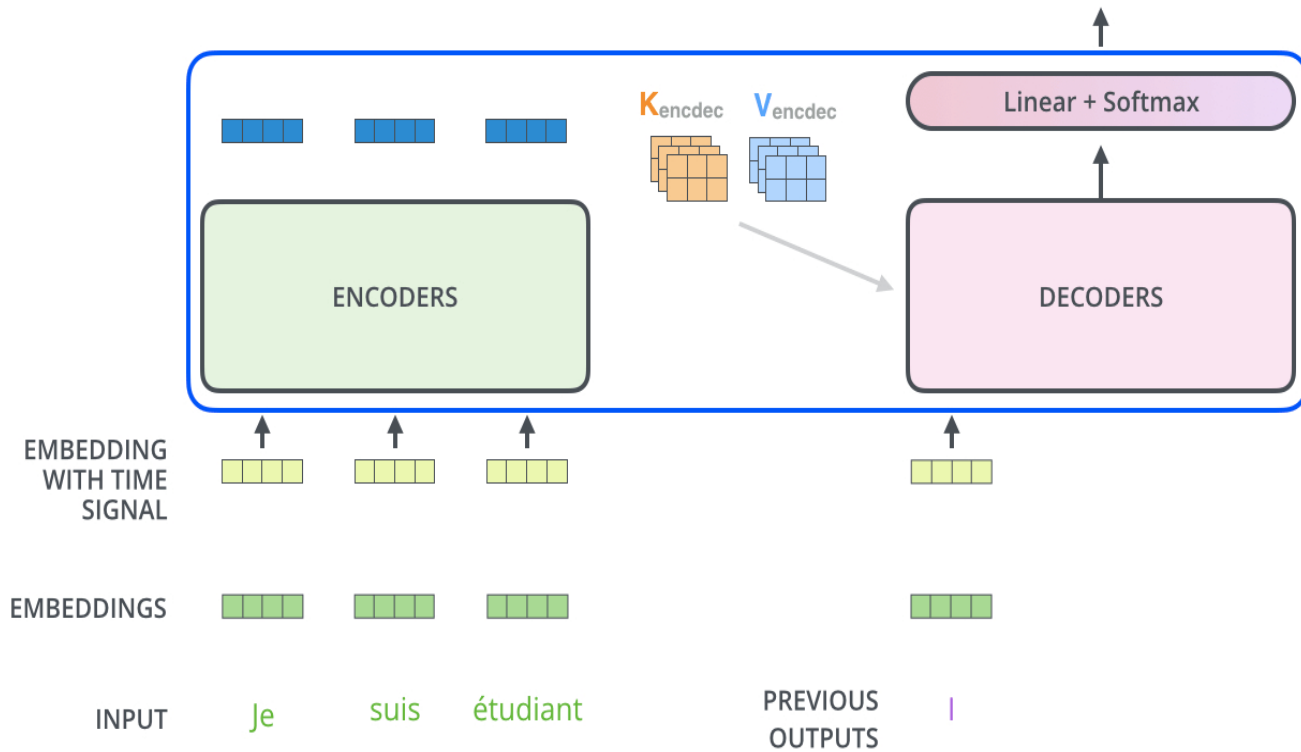
The second multi-headed attention layer. For this layer, the encoder's outputs are the queries and the keys, and the first multi-headed attention layer outputs are the values. This process matches the encoder's input to the decoder input, allowing the decoder to decide which encoder input is relevant to put a focus on. The output of the second multi-headed attention goes through a pointwise feedforward layer for further processing.

This process is illustrated in the next slide..



Decoding time step: 1 2 3 4 5 6

OUTPUT |

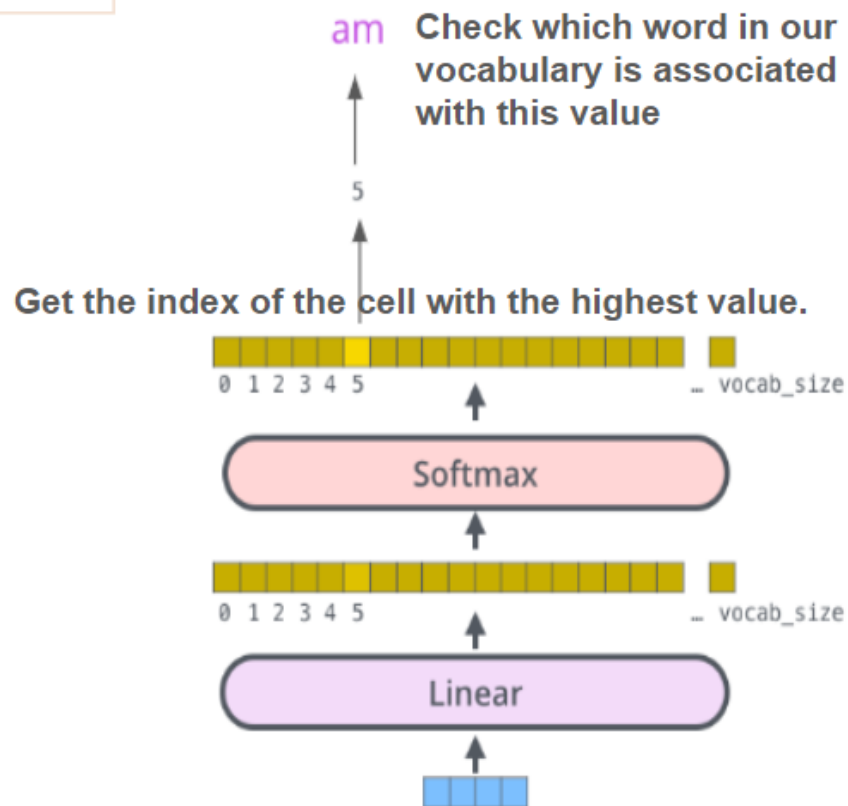


The Final Layer and the Softmax Layer

The Linear layer is a simple fully connected neural network that projects the vector produced by the stack of decoders, into a much, much larger vector called a logits vector.

Let's assume that our model knows 10,000 unique English words (our model's "output vocabulary") that it's learned from its training dataset. This would make the logits vector 10,000 cells wide – each cell corresponding to the score of a unique word. That is how we interpret the output of the model followed by the Linear layer.

The softmax layer then turns those scores into probabilities (all positive, all add up to 1.0). The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step.



Conclusion

We hope this preliminary understanding of Transformers was useful we would encourage students to read the paper[1] cited in the references and also refer to the blog posts and videos linked in the same.

References

- [1] A. Vaswani *et al.*, 'Attention Is All You Need', *arXiv [cs.CL]*. 2023.
- [2] <https://www.youtube.com/watch?v=iDulhoQ2pro&t=102s>
- [3] <https://jalammar.github.io/illustrated-transformer/>
- [4] <https://towardsdatascience.com/illustrated-guide-to-transformers-step-by-step-explanation-f7487652>
- [5] <https://www.youtube.com/watch?v=rBCqOTEfxvg>
- [6] <https://blog.research.google/2017/08/transformer-novel-neural-network.html>



UE21CS343BB2

Topics in Deep Learning

Dr. Shylaja S S

Director of Cloud Computing & Big Data (CCBD), Centre
for Data Sciences & Applied Machine Learning (CDSAML)
Department of Computer Science and Engineering
shylaja.sharath@pes.edu

**Ack: Anashua Dastidar,
Teaching Assistant**