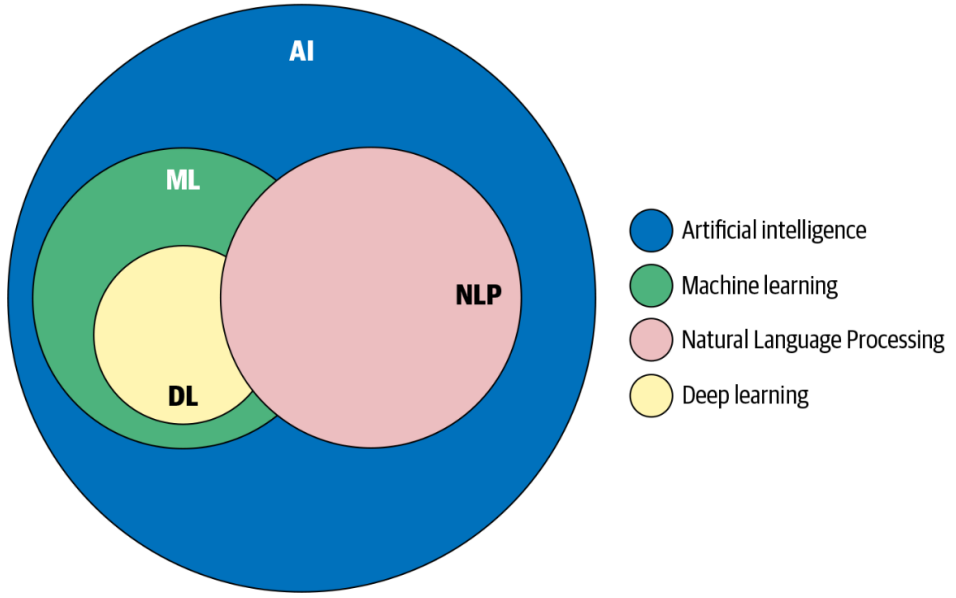
NLP – Transformers

Dr. Sarwan Singh

Scientist – D, NIELIT Chandigarh

Agenda

- Sequence-to-sequence (seq2seq) Models
- Recurrent Neural Network (RNN) Models
- Attention Mechanism
- Transformer

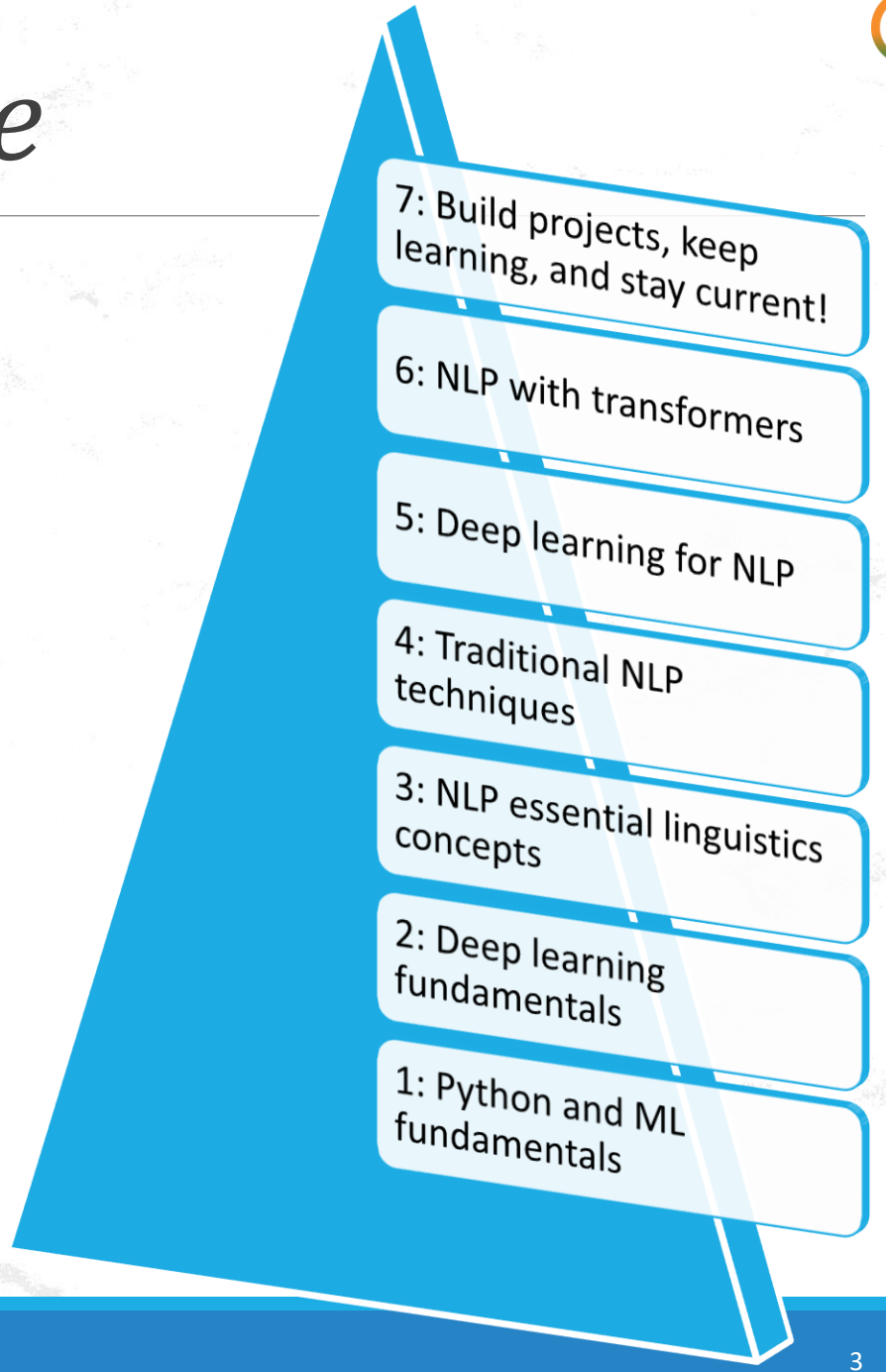


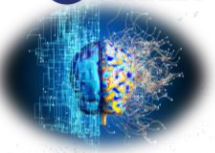
References

- towardsdatascience.com, huggingface.co,

NLP Ecosystem - *glance*

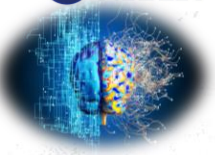
- Step 1: Python and ML fundamentals
- Step 2: Deep learning fundamentals
- Step 3: NLP essential linguistics concepts
- Step 4: Traditional NLP techniques
- Step 5: Deep learning for NLP
- Step 6: NLP with transformers
- Step 7: Build projects, keep learning, and stay current!





Sequence-to-Sequence Models – A Backdrop

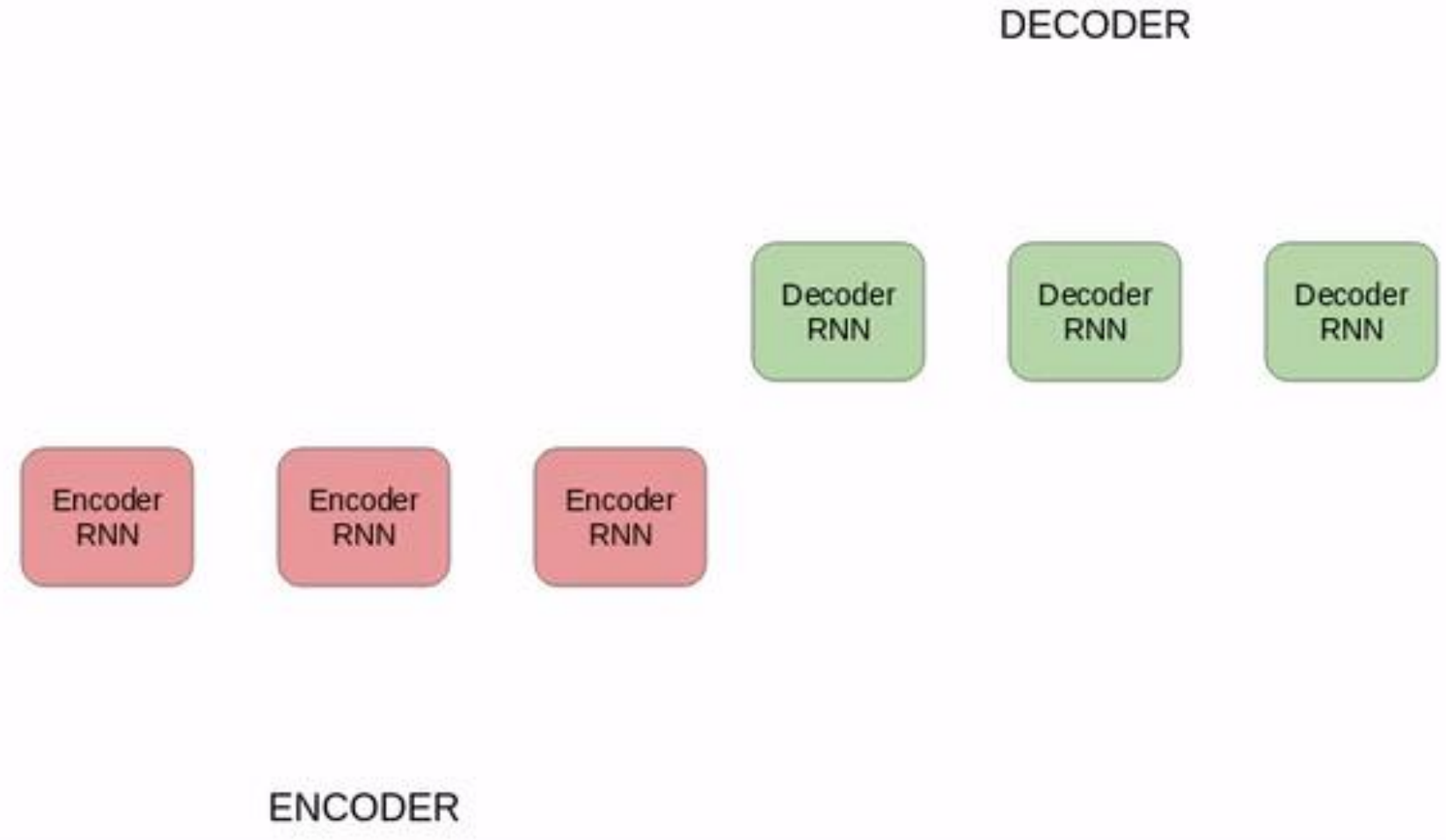
- **Sequence-to-sequence (seq2seq)** models in NLP are used to convert sequences of Type A to sequences of Type B. For example, translation of English sentences to German sentences is a sequence-to-sequence task.
- **Recurrent Neural Network (RNN) based sequence-to-sequence models** have garnered a lot of traction ever since they were introduced in 2014. Most of the data in the current world are in the form of sequences – it can be a number sequence, text sequence, a video frame sequence or an audio sequence.
- The performance of these seq2seq models was further enhanced with the addition of the **Attention Mechanism** in 2015.



- These sequence-to-sequence models are pretty versatile and they are used in a variety of NLP tasks, such as:
  - Machine Translation
  - Text Summarization
  - Speech Recognition
  - Question-Answering System, and so on

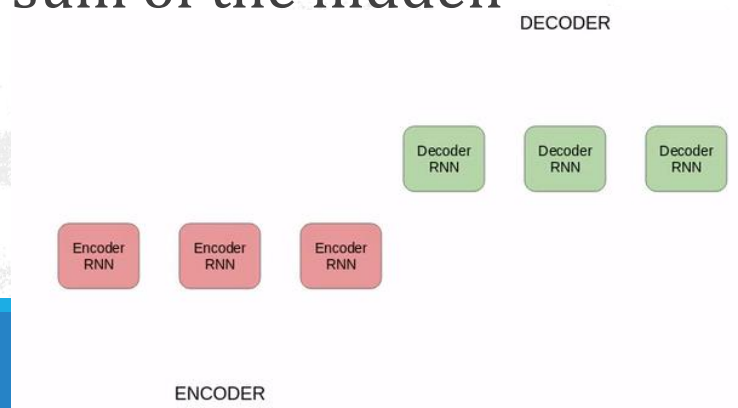
RNN based Sequence-to-Sequence Model

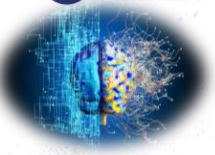
- A simple example of a sequence-to-sequence model
- *German to English Translation using seq2seq*



German to English Translation using seq2seq

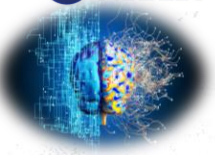
- The previous eg. seq2seq model is converting a German phrase to its English counterpart. Let's break it down:
  - Both **Encoder** and **Decoder** are RNNs
  - At every time step in the Encoder, the RNN takes a word vector (xi) from the input sequence and a hidden state (Hi) from the previous time step
  - The hidden state is updated at each time step
  - The hidden state from the last unit is known as the **context vector.** This contains information about the input sequence
  - This context vector is then passed to the decoder and it is then used to generate the target sequence (English phrase)
  - If we use the **Attention mechanism**, then the weighted sum of the hidden states are passed as the context vector to the decoder





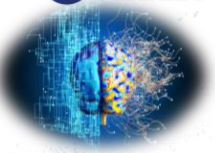
Challenges

- Despite being so good at what it does, there are certain limitations of seq-2-seq models with attention:
- Dealing with long-range dependencies is still challenging
- The sequential nature of the model architecture prevents parallelization. These challenges are addressed by Google Brain's Transformer concept



Before transformers - attention

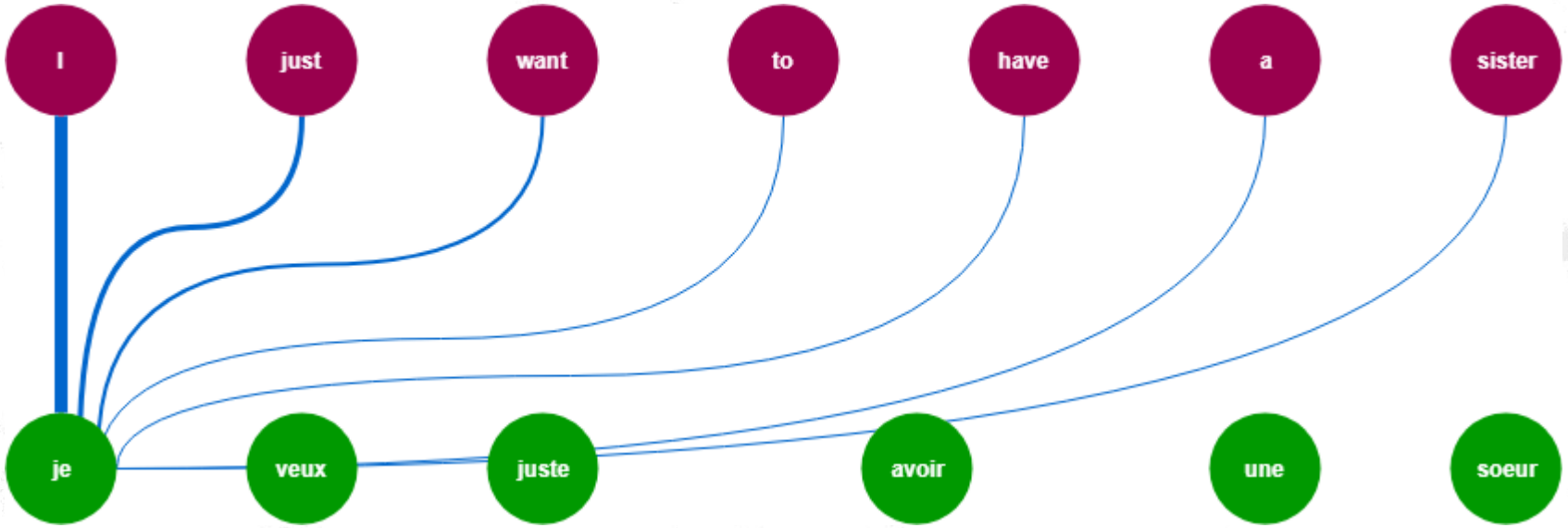
- ***Self attention**, sometimes called intra-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence.*
- In simpler terms, ***self attention helps us create similar connections but within the same sentence.*** Look at the following example:
  - "I poured water from the bottle into the cup until it was full."
  - it => cup
  - "I poured water from the bottle into the cup until it was empty."
  - it=> bottle
  - By changing one word "*full*" --> "*empty*" the reference object for "*it*" changed. While translating such a sentence, we want to know the word "*it*" refers to.



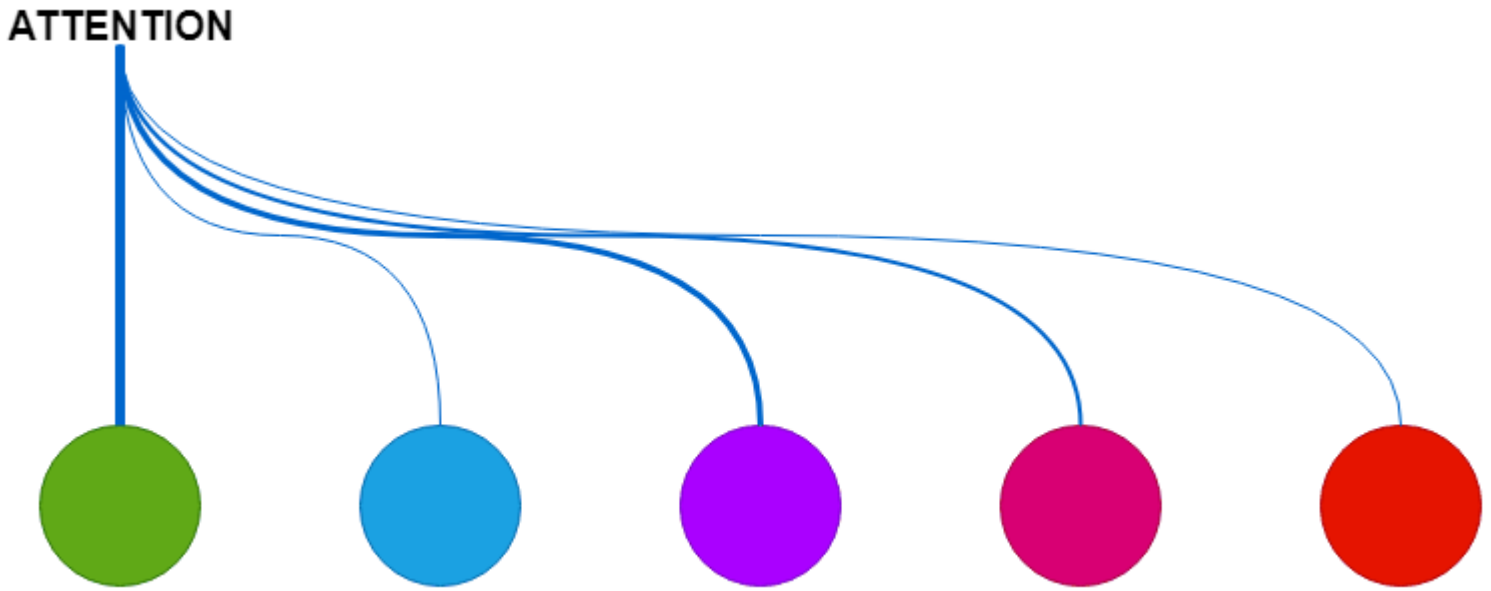
Three kinds of Attention possible in a model

- ***Encoder-Decoder Attention*:** Attention between the input sequence and the output sequence.
- ***Self attention in the input sequence*:** Attends to all the words in the input sequence.
- ***Self attention in the output sequence:*** One thing we should be wary of here is that the scope of self attention is limited to the words that occur before a given word. This prevents any information leaks during the training of the model. This is done by masking the words that occur after it for each step. So for step 1, only the first word of the output sequence is NOT masked, for step 2, the first two words are NOT masked and so on.

- Attention based encode - decoder



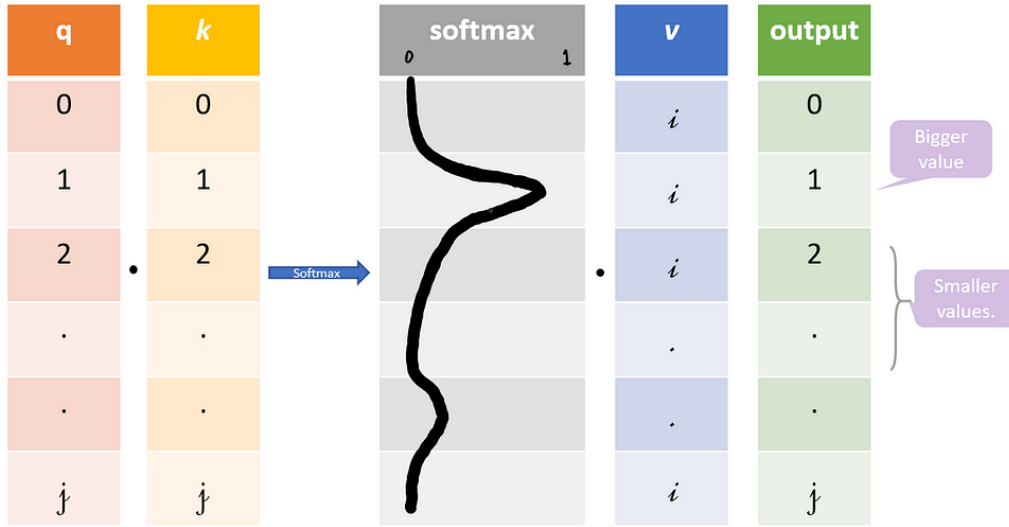
- Query, Key and Value in Attention mechanism



Keys, Values, and Queries:

- vectors created as abstractions are useful for calculating self attention. These are calculated by multiplying your input vector(*X*) with weight matrices that are learnt while training.
- ***Query Vector***: *q= X * Wq.* Think of this as the current word.
- ***Key Vector****: k= X * Wk.* Think of this as an indexing mechanism for Value vector. Similar to how we have key-value pairs in hash maps, where keys are used to uniquely index the values.
- ***Value Vector:*** *v= X * Wv.* Think of this as the information in the input word.

- query *q* and find the most similar key *k*, by doing a dot product for *q* and *k*.
- The closest query-key product will have the highest value, followed by a softmax that will drive the *q.k* with smaller values close to 0 and *q.k* with larger values towards 1.
- This softmax distribution is multiplied with *v*. The value vectors multiplied with ~1 will get more attention while the ones ~0 will get less.
- The sizes of these *q*, *k* and *v* vectors are referred to as "***hidden size***" by various implementations.



Calculating Self attention from q, k and v

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

- ***Step 1:*** Multiply q_i by the k_j key vector of wo
- ***Step 2:*** Then divide this product by the square root of the dimension of key vector.
  This step is done **for better gradient flow** which is specially important in cases when the value of the dot product in previous step is too big. As using them directly might push the softmax into regions with very little gradient flow.
- ***Step 3:*** Once we have scores for all *j*s, we pass these through a softmax. We get normalized value for each *j*.

- ***Step 4:*** Multiply softmax scores for each *j* with v_i vector.
The idea/purpose here is, very similar attention, to keep preserve only the values *v* of the input word(s) we want to focus on by multiplying them with high probability scores from softmax ~1, and remove the rest by driving them towards 0, i.e. making them very small by multiplying them with the low probability scores ~0 from softmax.

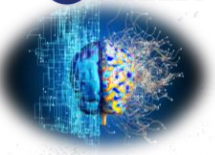
$$Step\ 1 = q_i \cdot k_j \qquad for\ all\ 0 \leqq j \leqq n$$

$$Step\ 2 = \frac{q_i \cdot k_j}{\sqrt{dim(k_j)}} \qquad for\ all\ 0 \leqq j \leqq n$$

$$Step\ 3 = softmax\left(\frac{q_i \cdot k_j}{\sqrt{dim(k_j)}}\right) \qquad for\ all\ 0 \leqq j \leqq n$$

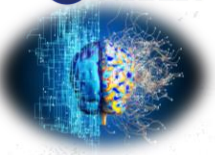
$$Step\ 4 = \left\{softmax\left(\frac{q_i \cdot k_j}{\sqrt{dim(k_j)}}\right) \cdot v_i\right\} \qquad for\ all\ 0 \leqq j \leqq n$$

$$Finally : z_i = \sum_{j=0}^{n}\left(softmax\left(\frac{q_i \cdot k_j}{\sqrt{dim(k_j)}}\right) \cdot v_i\right)$$



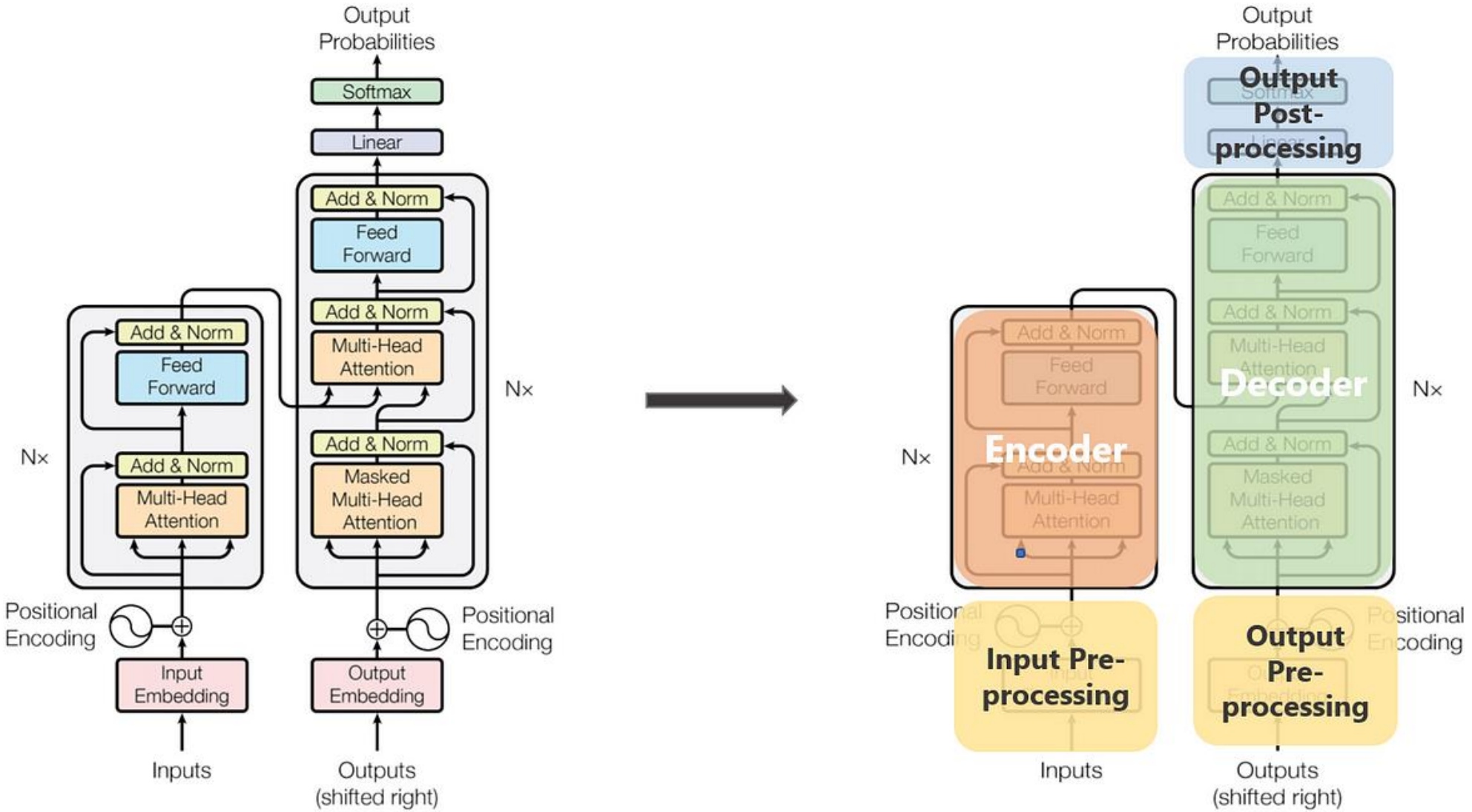
What are Transformers

- Many models fell short when it came to handling long-range dependencies.
- The concept of attention somewhat allowed us to overcome that problem and now in Transformers we build on top of attention and unleash its full potential.
- The Transformer in NLP is a novel architecture that aims to solve sequence-to-sequence tasks while handling long-range dependencies with ease.
- It relies entirely on self-attention to compute representations of its input and output WITHOUT using sequence-aligned RNNs or convolution.

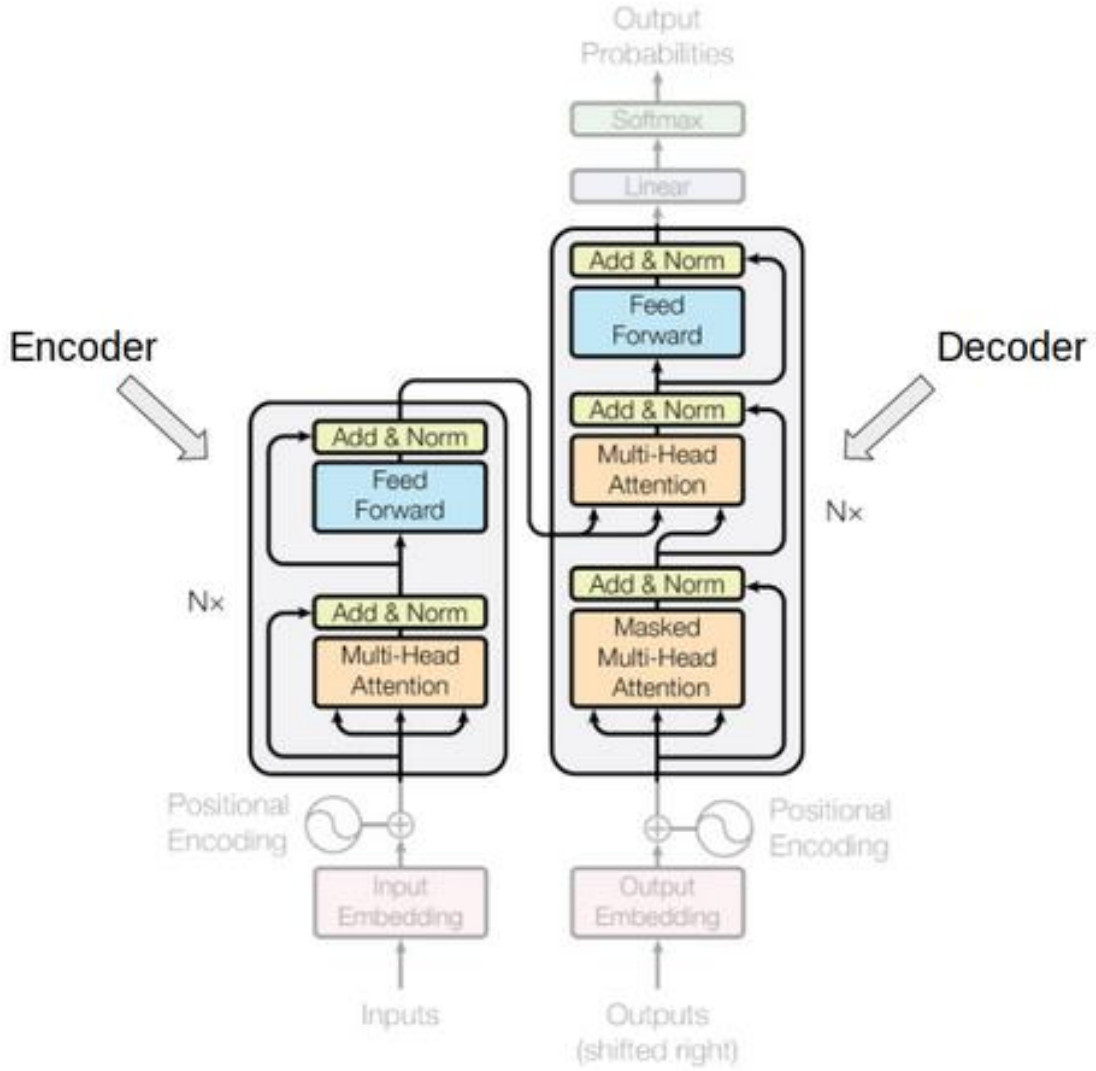


Transformers

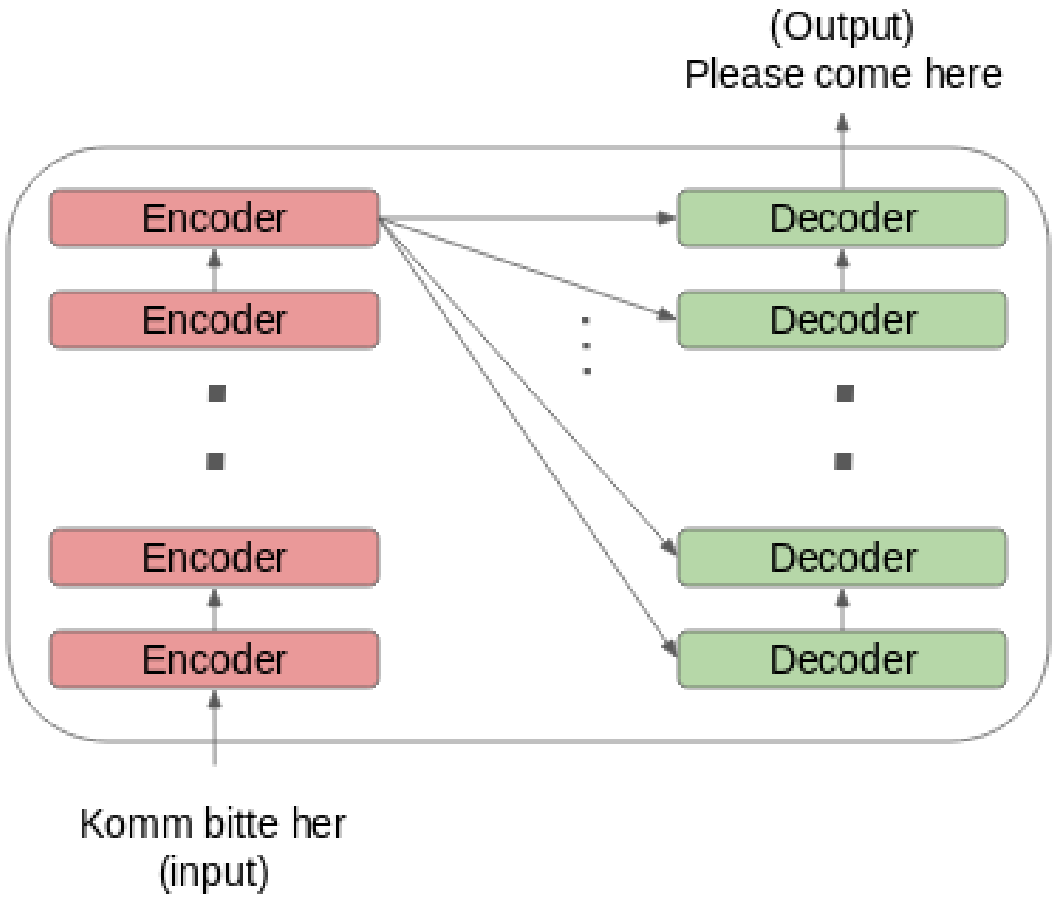
- The Transformer in NLP is a novel architecture that aims to solve sequence-to-sequence tasks while handling long-range dependencies with ease.
- The Transformer was proposed in the paper Attention Is All You Need.
  - ***Quote from paper*** "The Transformer is the first transduction model relying entirely on self-attention to compute representations of its input and output without using sequence-aligned RNNs or convolution."
  - Here, "transduction" means the conversion of input sequences into output sequences. The idea behind Transformer is to handle the dependencies between input and output with **attention** and recurrence completely.


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)
Output Post-processing
Encoder
Decoder
Input Pre-processing
Output Pre-processing

- The Encoder block has 1 layer of a **Multi-Head Attention** followed by another layer of **Feed Forward Neural Network**. The decoder, on the other hand, has an extra **Masked Multi-Head Attention.**
- **The encoder and decoder blocks are actually multiple identical encoders and decoders stacked on top of each other.** Both the encoder stack and the decoder stack have the same number of units.

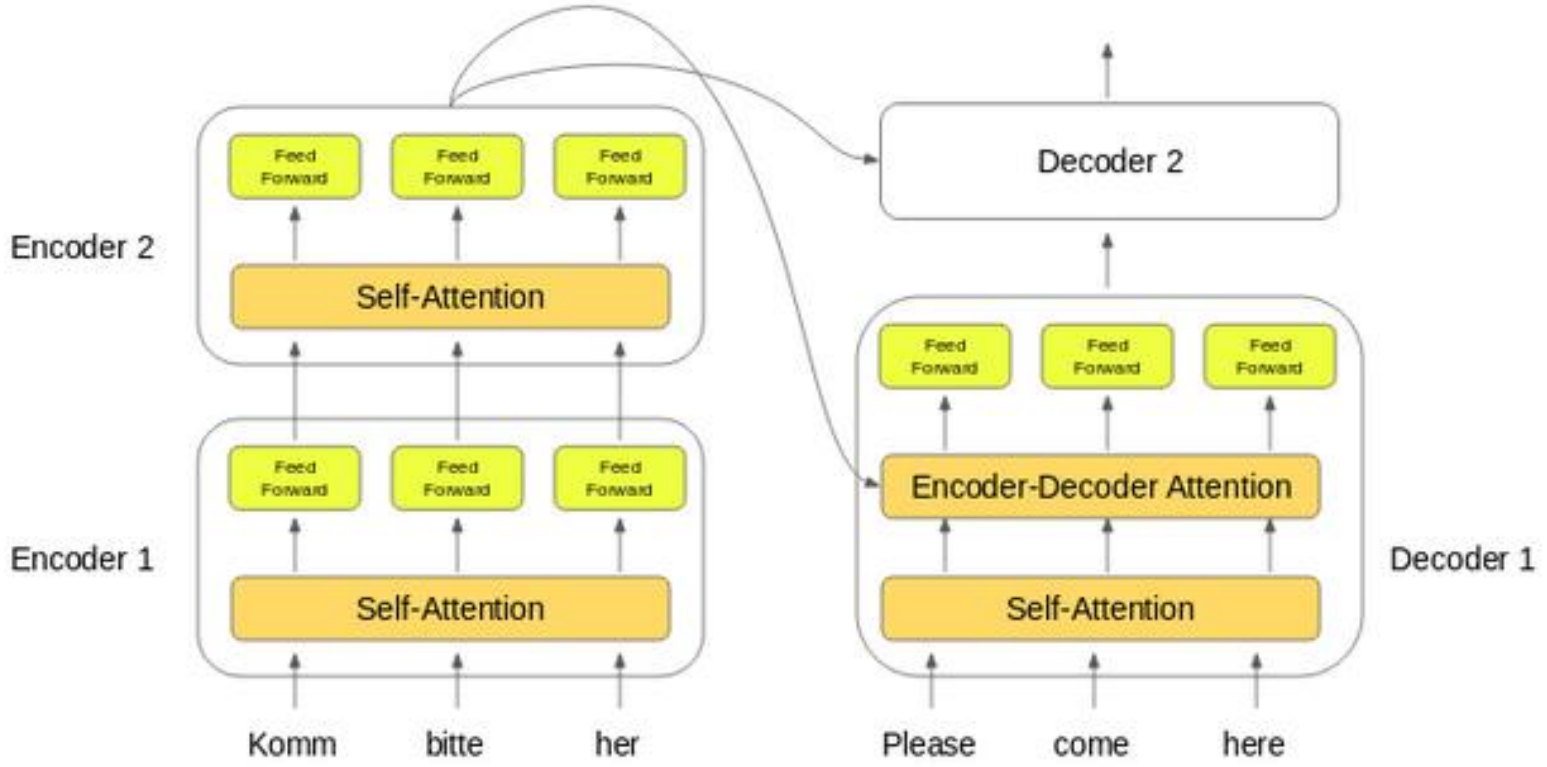


- The number of encoder and decoder units is a hyperparameter.
- In the paper, 6 encoders and decoders have been used.



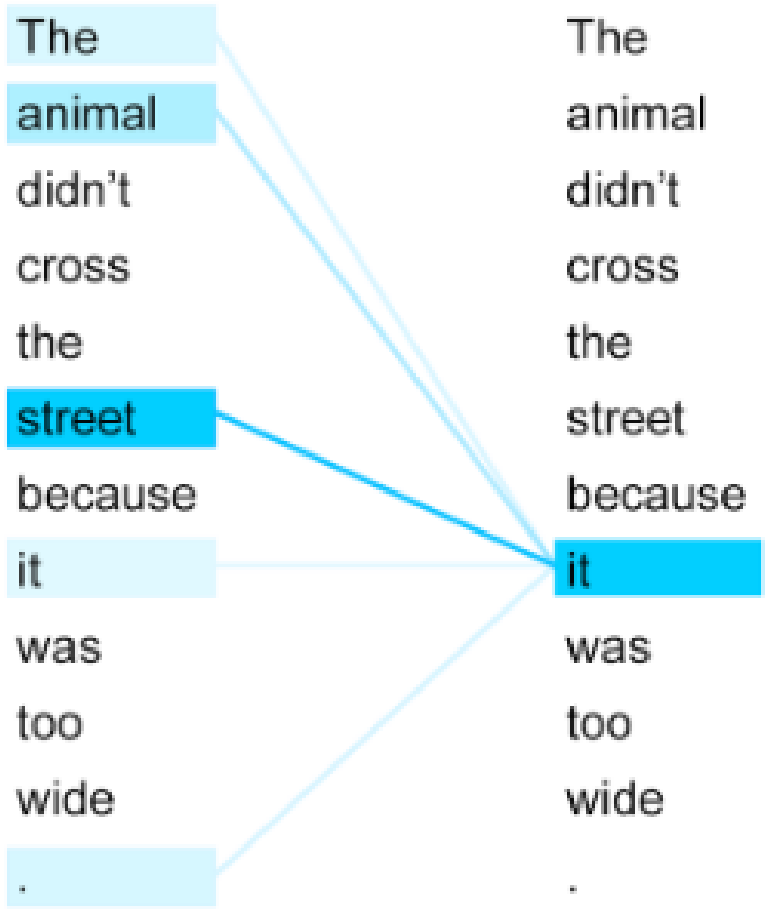
how encoder and the decoder stack works:

- The word embeddings of the input sequence are passed to the first encoder
- These are then transformed and propagated to the next encoder
- The output from the last encoder in the encoder-stack is passed to all the decoders in the decoder-stack



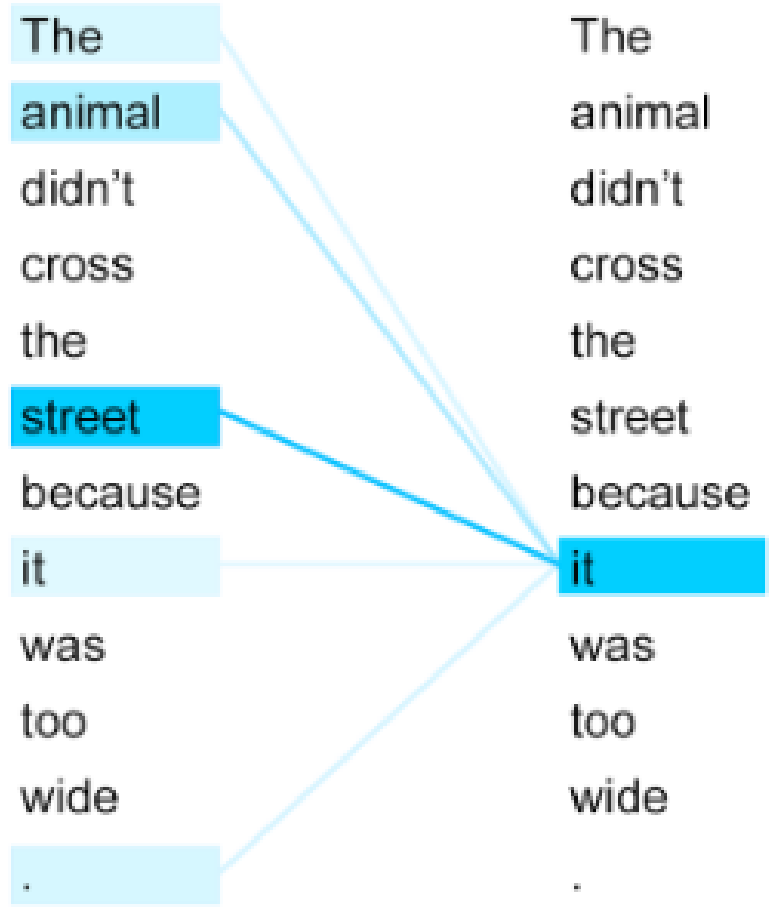
More on self-attention

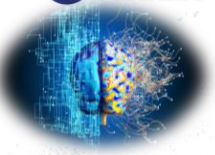
- An important thing to note here – in addition to the **self-attention** and feed-forward layers, the decoders also have one more layer of Encoder-Decoder Attention layer. This helps the decoder focus on the appropriate parts of the input sequence.
- According to paper
  - *"Self-attention, sometimes called intra-attention, is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence."*



the term **"it"** in this sentence refers to?

- When the model is processing the word “it”, self-attention tries to associate “it” with **“animal”** in the same sentence.
- Self-attention allows the model to look at the other words in the input sequence to get a better understanding of a certain word in the sequence





Calculating Self-Attention

- First, we need to create three vectors from each of the encoder's input vectors:
  - Query Vector
  - Key Vector
  - Value Vector.
- These vectors are trained and updated during the training process. We'll know more about their roles once we are done with this section

- calculate self-attention for every word in the input sequence
- Consider this phrase – “Action gets results”. To calculate the self-attention for the first word “Action”, we will calculate scores for all the words in the phrase with respect to “Action”. This score determines the importance of other words when we are encoding a certain word in an input sequence

Word	q vector	k vector	v vector	score
Action	q_1	k_1	v_1	$q_1 . k_1$
gets		k_2	v_2	$q_1 . k_2$
results		k_3	v_3	$q_1 . k_3$

1. The score for the first word is calculated by taking the dot product of the Query vector (q1) with the keys vectors (k1, k2, k3) of all the words:

2. Then, these scores are divided by 8 which is the square root of the dimension of the key vector:

Word	q vector	k vector	v vector	score	score / 8
Action	q_1	k_1	v_1	$q_1 . k_1$	$q_1 . k_1 / 8$
gets		k_2	v_2	$q_1 . k_2$	$q_1 . k_2 / 8$
results		k_3	v_3	$q_1 . k_3$	$q_1 . k_3 / 8$

3. Next, these scores are normalized using the softmax activation function:

Word	q vector	k vector	v vector	score	score / 8	Softmax
Action	q_1	k_1	v_1	$q_1 . k_1$	$q_1 . k_1 / 8$	x_{11}
gets		k_2	v_2	$q_1 . k_2$	$q_1 . k_2 / 8$	x_{12}
results		k_3	v_3	$q_1 . k_3$	$q_1 . k_3 / 8$	x_{13}

- 4. These normalized scores are then multiplied by the value vectors (v1, v2, v3) and sum up the resultant vectors to arrive at the final vector (z1). This is the output of the self-attention layer. It is then passed on to the feed-forward network as input.

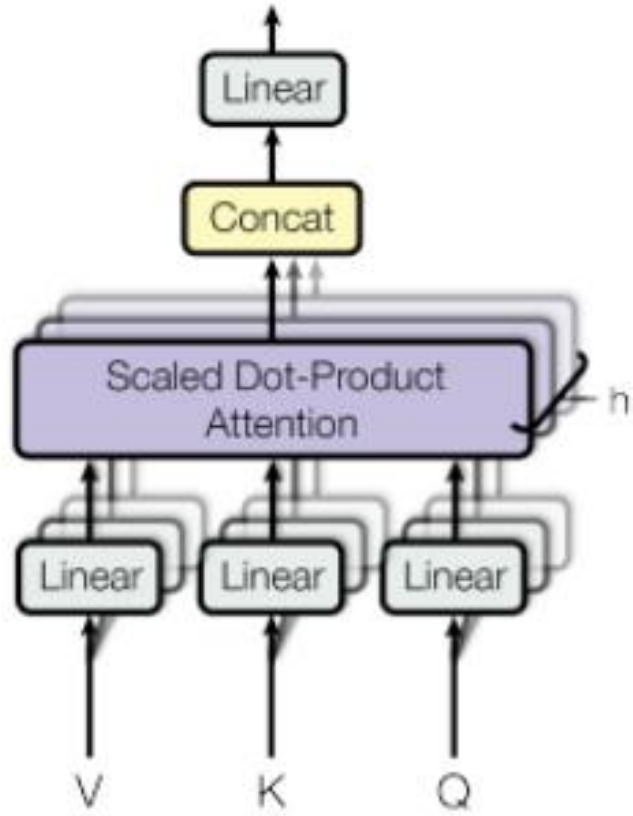
Word	q vector	k vector	v vector	score	score / 8	Softmax	Softmax * v	Sum
Action	q_1	k_1	v_1	$q_1 . k_1$	$q_1 . k_1 / 8$	x_{11}	$x_{11} * v_1$	z_1
gets		k_2	v_2	$q_1 . k_2$	$q_1 . k_2 / 8$	x_{12}	$x_{12} * v_2$	
results		k_3	v_3	$q_1 . k_3$	$q_1 . k_3 / 8$	x_{13}	$x_{13} * v_3$	

- So, z1 is the self-attention vector for the first word of the input sequence "Action gets results". We can get the vectors for the rest of the words in the input sequence in the same fashion:

Word	q vector	k vector	v vector	score	score / 8	Softmax	Softmax * v	Sum#
Action		k_1	v_1	$q_2 . k_1$	$q_2 . k_1 / 8$	x_{21}	$x_{21} * v_1$	
gets	q_2	k_2	v_2	$q_2 . k_2$	$q_2 . k_2 / 8$	x_{22}	$x_{22} * v_2$	z_2
results		k_3	v_3	$q_2 . k_3$	$q_2 . k_3 / 8$	x_{23}	$x_{23} * v_3$	

Word	q vector	k vector	v vector	score	score / 8	Softmax	Softmax * v	Sum#
Action		k_1	v_1	$q_3 . k_1$	$q_3 . k_1 / 8$	x_{31}	$x_{31} * v_1$	
gets		k_2	v_2	$q_3 . k_2$	$q_3 . k_2 / 8$	x_{32}	$x_{32} * v_2$	
results	q_3	k_3	v_3	$q_3 . k_3$	$q_3 . k_3 / 8$	x_{33}	$x_{33} * v_3$	z_3

- Self-attention is computed not once but multiple times in the Transformer's architecture, in parallel and independently. It is therefore referred to as **Multi-head Attention**. The outputs are concatenated and linearly transformed as shown in the figure.
- According to the paper "Attention Is All You Need" :
  - *"Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions."*



Multi-Head Attention

Limitations of the Transformer

- Transformer is undoubtedly a huge improvement over the RNN based seq2seq models. But it comes with its own share of limitations:
- Attention can only deal with fixed-length text strings. The text has to be split into a certain number of segments or chunks before being fed into the system as input
- This chunking of text causes **context fragmentation**. For example, if a sentence is split from the middle, then a significant amount of context is lost. In other words, the text is split without respecting the sentence or any other semantic boundary
- So how do we deal with these pretty major issues? That's the question folks who worked with Transformer asked. And out of this came Transformer-XL.