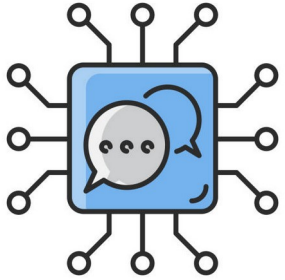
Transformers

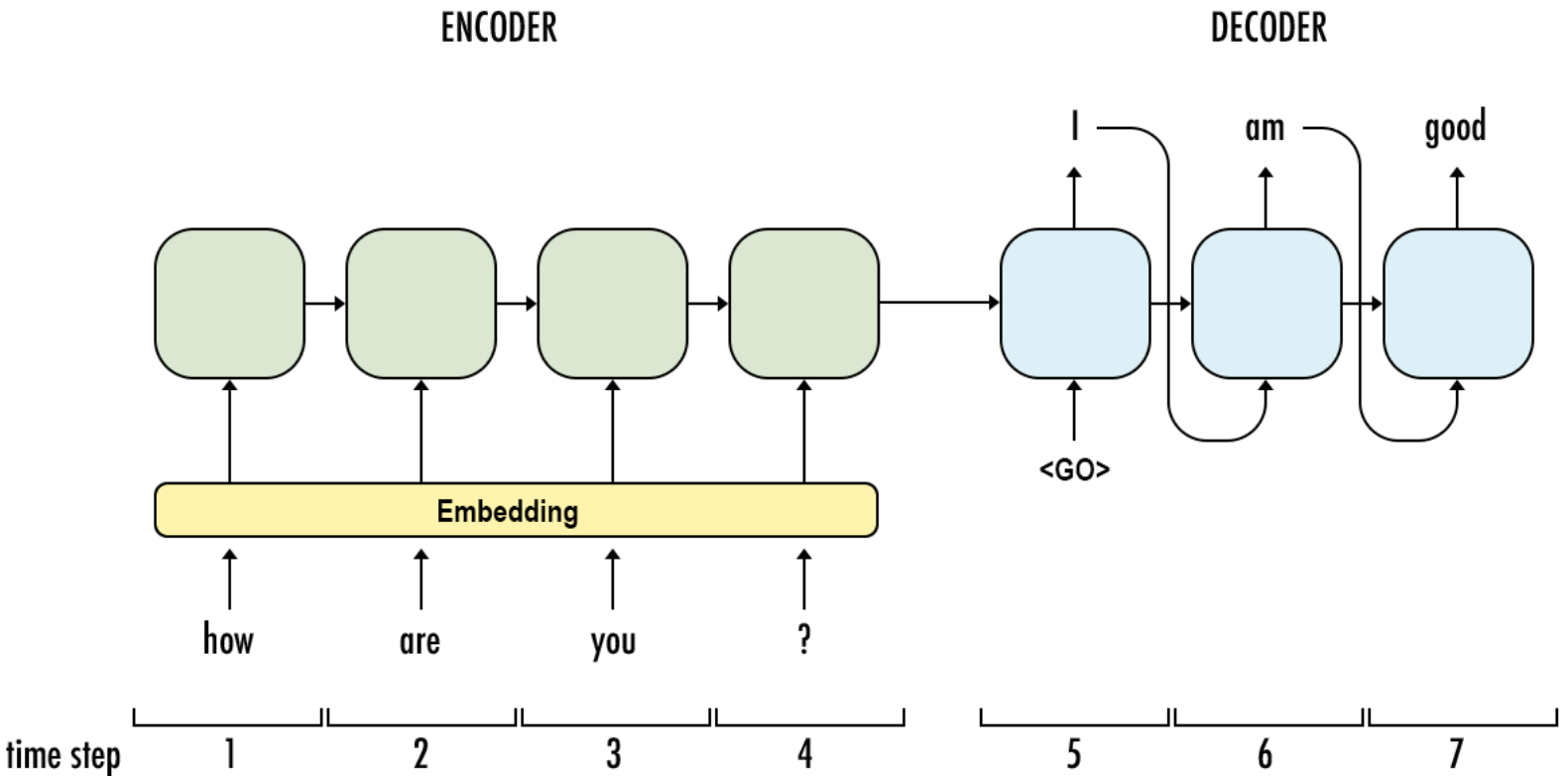
Tushar B. Kute,
http://tusharkute.com



Sequence to Sequence Models

- 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.

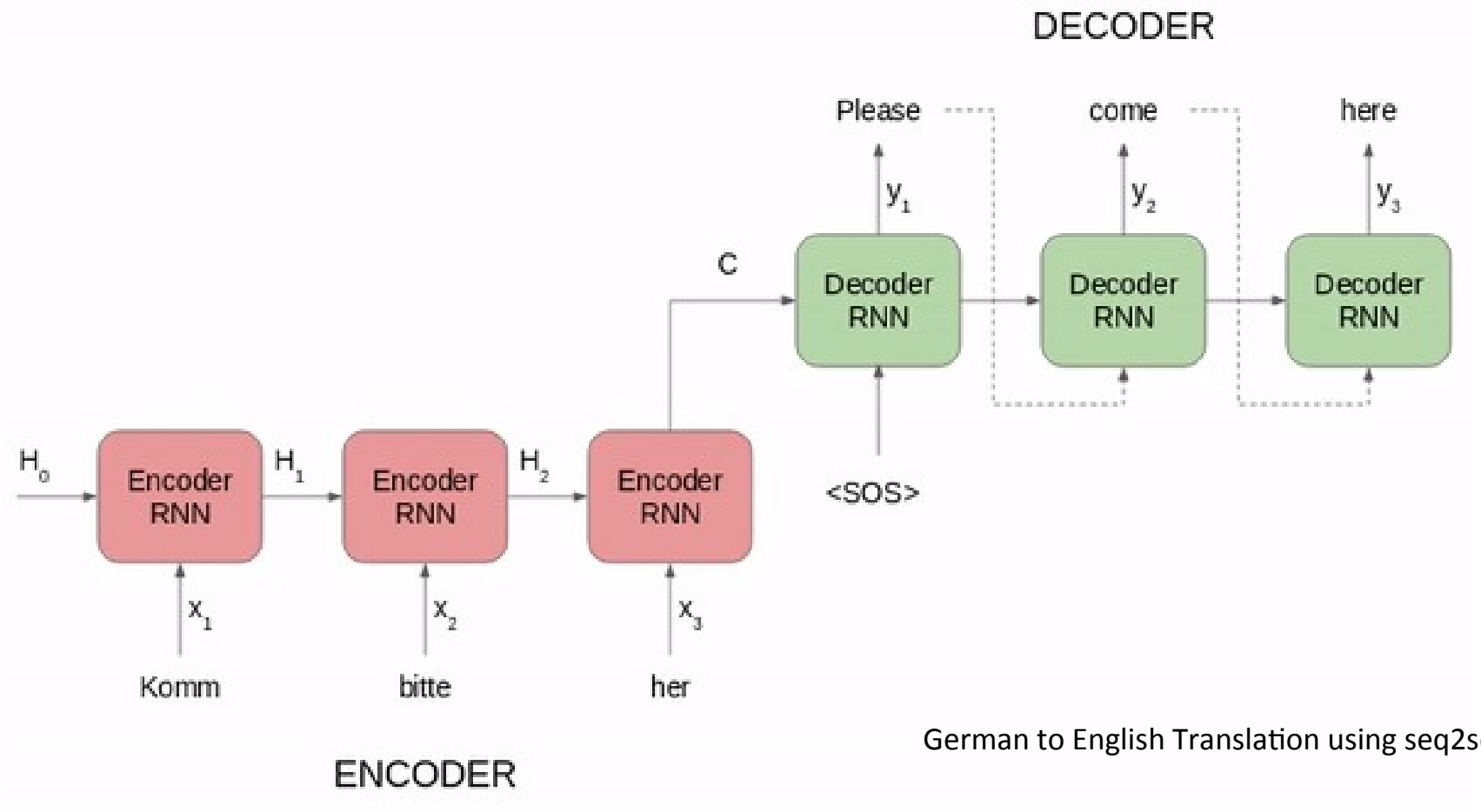
Sequence to Sequence Models



Sequence to Sequence Models

- The performance of these seq2seq models was further enhanced with the addition of the Attention Mechanism in 2015. How quickly advancements in NLP have been happening in the last 5 years – incredible!
- These sequence-to-sequence models are pretty versatile and they are used in a variety of NLP tasks, such as:
  - Machine Translation
  - Text Generation
  - Text Summarization
  - Speech Recognition
  - Question-Answering System, and so on

Sequence to Sequence Models



German to English Translation using seq2seq

Sequence to Sequence Models

- The above 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

Sequence to Sequence Models

- 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.

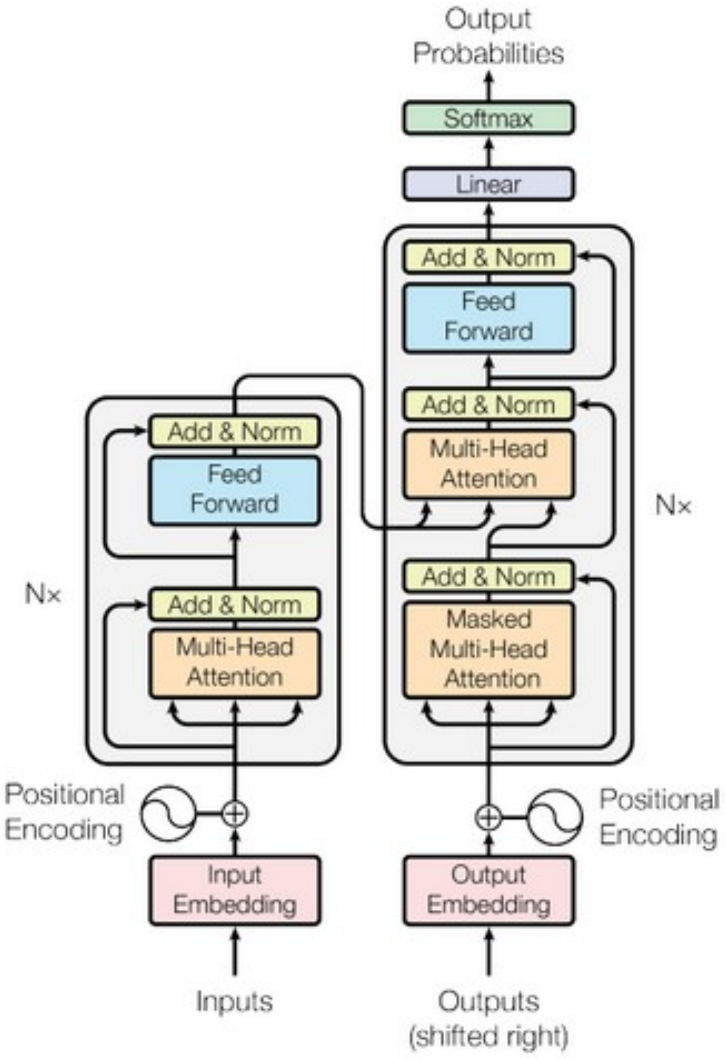
Challenges

- 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.

Transformer

- 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. It is recommended reading for anyone interested in NLP.
- Quoting from the 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.

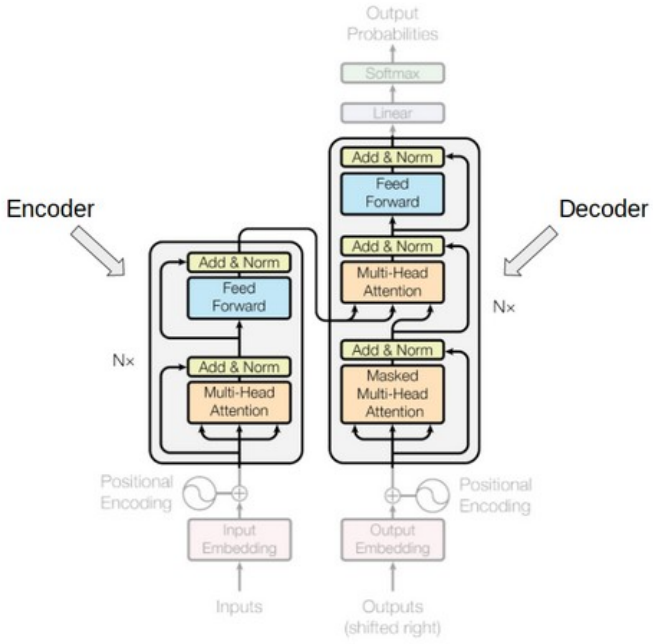
Transformer : Model



(Source: https://arxiv.org/abs/1706.03762)

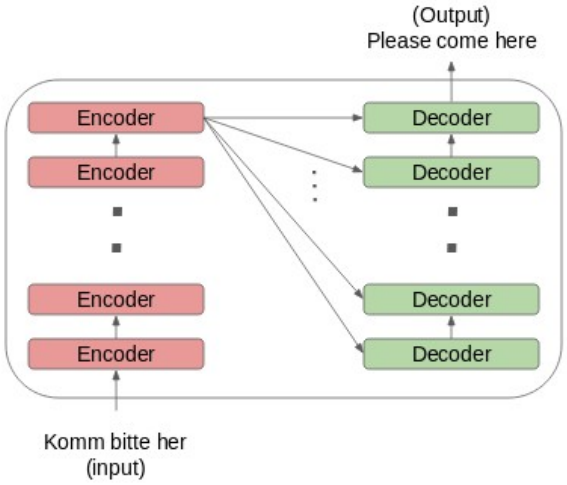
Transformer

- 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.



Transformer

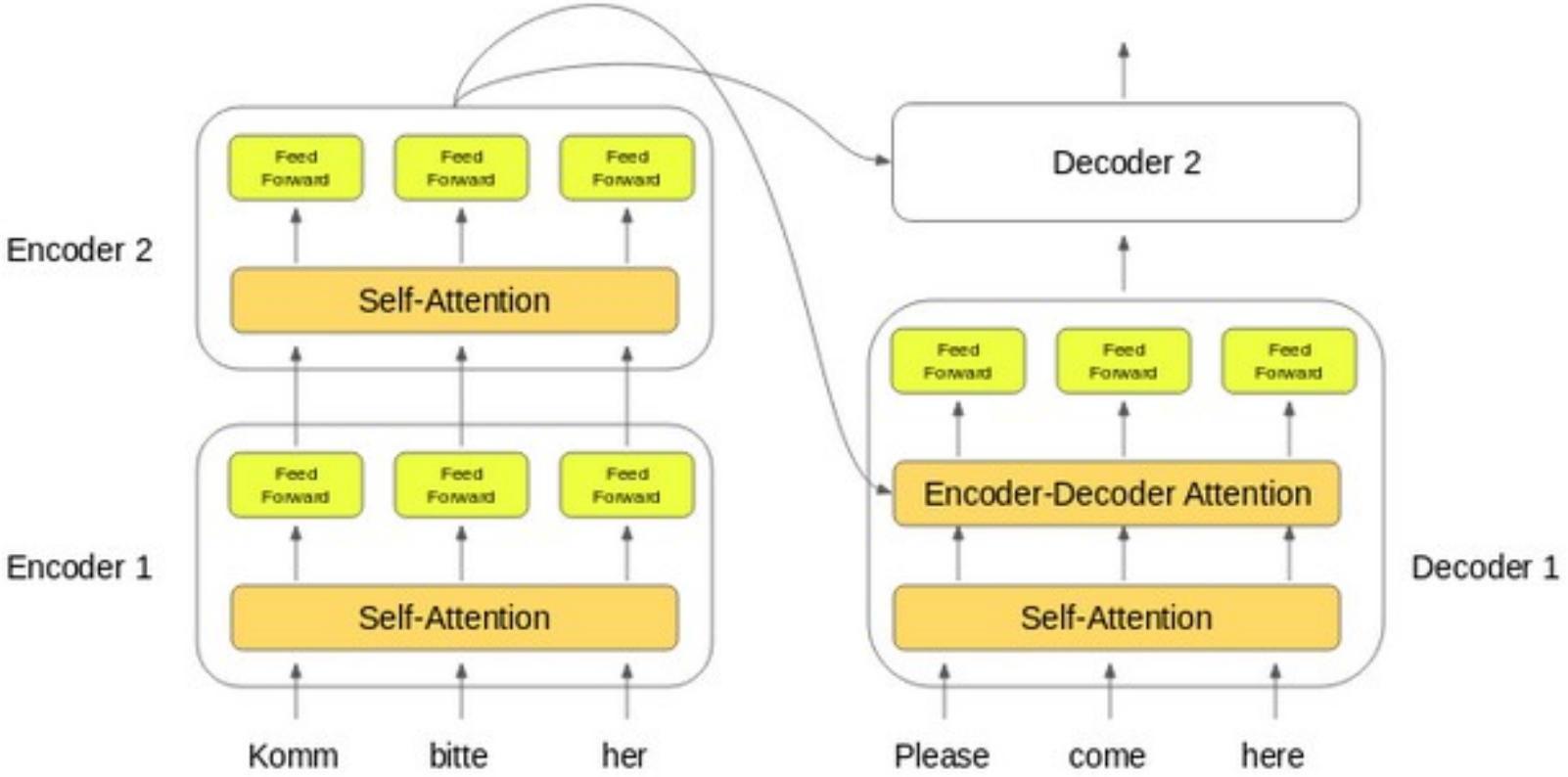
- The encoder and decoder blocks are actually multiple identical encoders and decoders stacked on top of each other. Both the encoder stack and the decoder stack have the same number of units.
- The number of encoder and decoder units is a hyperparameter. In the paper, 6 encoders and decoders have been used.



Transformer

- Let’s see how this setup of the 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 as shown in the figure below:

Transformer



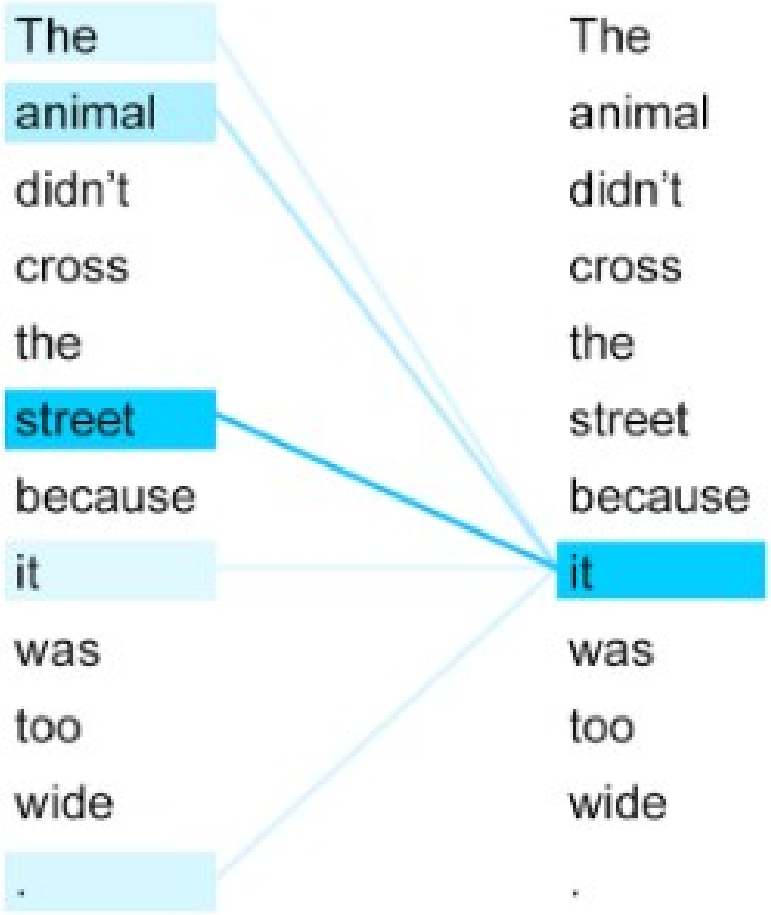
Transformer

- 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.
- You might be thinking – what exactly does this “Self-Attention” layer do in the Transformer?

Transformer: Self Attention

- According to the 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.”

Transformer: Self Attention



Transformer: Self Attention

- Take a look at the above image. Can you figure out what the term “it” in this sentence refers to?
- Is it referring to the street or to the animal? It’s a simple question for us but not for an algorithm. 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. Now, let’s see how we can calculate self-attention.

Transformer: Self Attention Calc.

- I have divided this section into various steps for ease of understanding.
- 1. 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

Transformer: Self Attention Calc.

- 2. Next, we will calculate self-attention for every word in the input sequence
- 3. 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

Transformer: Self Attention Calc.

- 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:

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	$q_1 . k_3$

Transformer: Self Attention Calc.

- 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	$q_1 . k_3$	$q_1 . k_3 / 8$

Transformer: Self Attention Calc.

- 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}

Transformer: Self Attention Calc.

- 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}	$x_{13} * v_3$	

Transformer: Self Attention Calc.

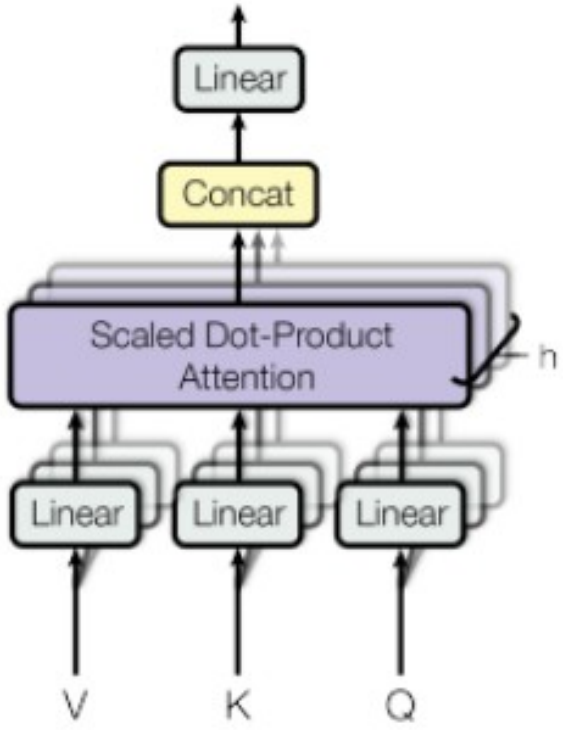
- So, z1 is the self-attention vector for the first word of the input sequence “Action gets results”. We can get the vectors for the rest of the words in the input sequence in the same fashion:

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

Transformer: Self Attention Calc.

- 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:



Multi-Head Attention

Transformer: Self Attention Calc.

- 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.”

Transformer: Limitations

- 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

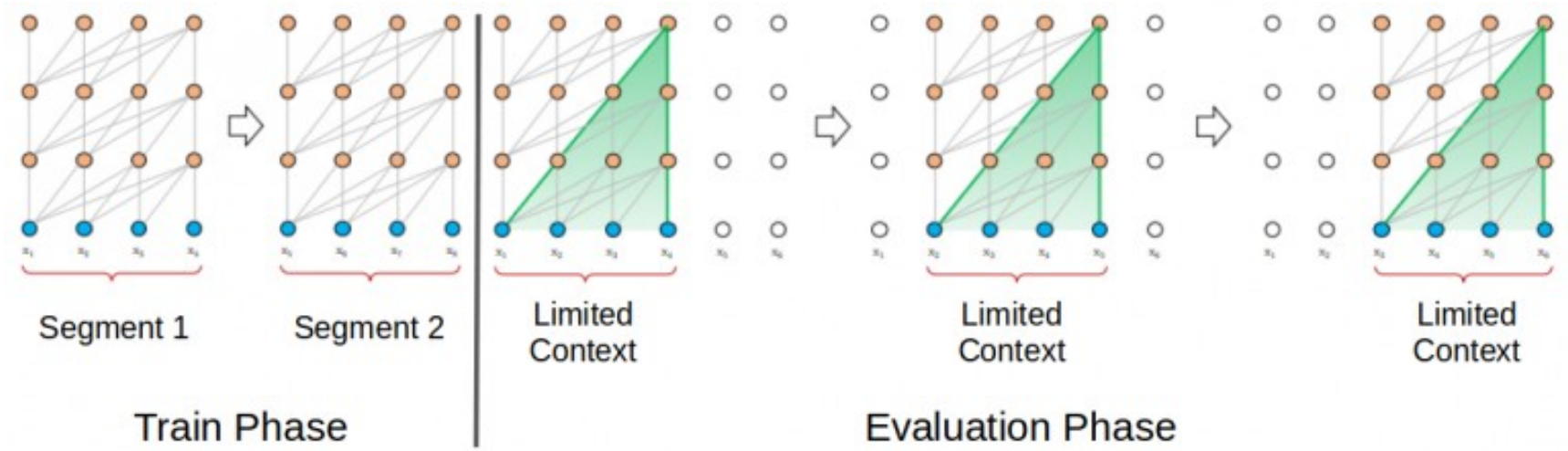
TransformerXL

- Transformer architectures can learn longer-term dependency. However, they can’t stretch beyond a certain level due to the use of fixed-length context (input text segments).
- A new architecture was proposed to overcome this shortcoming in the paper – Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context.
- In this architecture, the hidden states obtained in previous segments are reused as a source of information for the current segment.
- It enables modeling longer-term dependency as the information can flow from one segment to the next.

Using Transformer for Language Modeling

- Think of language modeling as a process of estimating the probability of the next word given the previous words.
- Al-Rfou et al. (2018) proposed the idea of applying the Transformer model for language modeling. As per the paper, the entire corpus can be split into fixed-length segments of manageable sizes.
- Then, we train the Transformer model on the segments independently, ignoring all contextual information from previous segments:

Using Transformer for Language Modeling



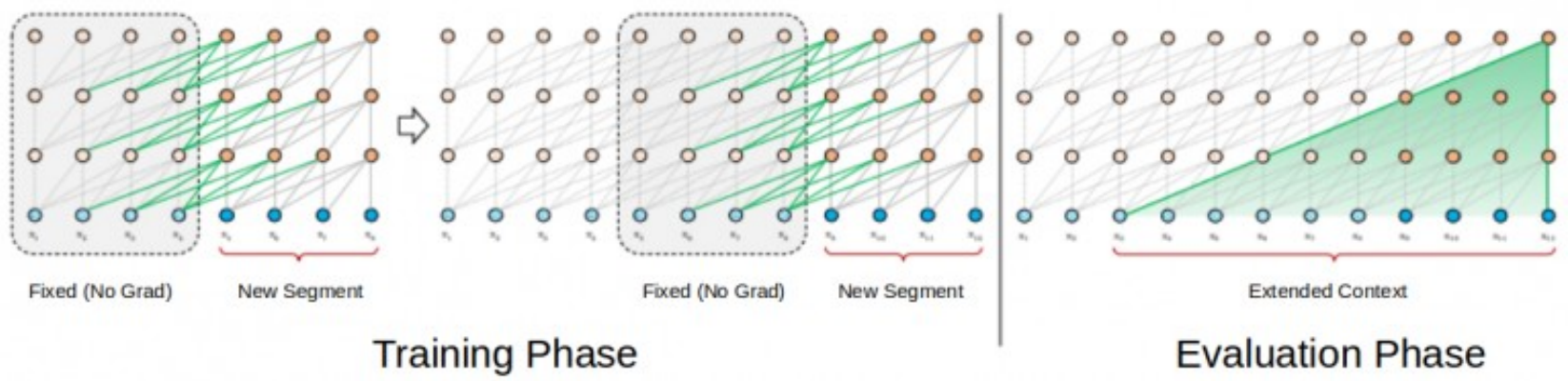
Transformer Model with a segment length of 4 (Source: https://arxiv.org/abs/1901.02860)

Using Transformer for Language Modeling

- This architecture doesn’t suffer from the problem of vanishing gradients. But the context fragmentation limits its longer-term dependency learning.
- During the evaluation phase, the segment is shifted to the right by only one position. The new segment has to be processed entirely from scratch.
- This evaluation method is unfortunately quite compute-intensive.

Using Transformer for Language Modeling

- During the training phase in Transformer-XL, the hidden state computed for the previous state is used as an additional context for the current segment.
- This recurrence mechanism of Transformer-XL takes care of the limitations of using a fixed-length context.



Transformer XL Model with a segment length of 4

Using Transformer for Language Modeling

- During the evaluation phase, the representations from the previous segments can be reused instead of being computed from scratch (as is the case of the Transformer model).
- This, of course, increases the computation speed manifold.

Thank you

This presentation is created using LibreOffice Impress 7.4.1.2, can be used freely as per GNU General Public License

@mitu_skillologies

@mITuSkillologies

@mitu_group

@mitu-skillologies

@MITUSkillologies

kaggle

@mituskillologies

Web Resources
https://mitu.co.in
http://tusharkute.com

@mituskillologies

contact@mitu.co.in

tushar@tusharkute.com