How do Transformers Work in NLP? A Guide to the Latest State-of-the-Art Models

PRATEEK JOSHI, JUNE 19, 2019

Overview

- The Transformer model in NLP has truly changed the way we work with text data
- Transformer is behind the recent NLP developments, including Google's BERT
- Learn how the Transformer idea works, how it's related to language modeling, sequence-to-sequence modeling, and how it enables Google's BERT model

Introduction

I love being a data scientist working in Natural Language Processing (NLP) right now. The breakthroughs and developments are occurring at an unprecedented pace. From the super-efficient ULMFiT framework to Google's BERT, NLP is truly in the midst of a golden era.

And at the heart of this revolution is the concept of the Transformer. This has transformed the way we data scientists work with text data – and you'll soon see how in this article.

Want an example of how useful Transformer is? Take a look at the paragraph below:

> **"Griezmann's announcement comes as a bit of a shock. After enduring the drama surrounding his potential last summer, many thought he was committed to Atletico for more than a year, but the Frenchman seems to have changed his mind."**

The highlighted words refer to the same person – Griezmann, a popular football player. It's not that difficult for us to figure out the relationships among such words spread across the text. However, it is quite an uphill task for a machine.

Capturing such relationships and sequence of words in sentences is vital for a machine to understand a natural language. **This is where the Transformer concept plays a major role.**

Note: This article assumes a basic understanding of a few deep learning concepts:

- *Essentials of Deep Learning – Sequence to Sequence modeling with Attention*
- *Fundamentals of Deep Learning – Introduction to Recurrent Neural Networks*
- *Comprehensive Guide to Text Summarization using Deep Learning in Python*

Table of Contents

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.

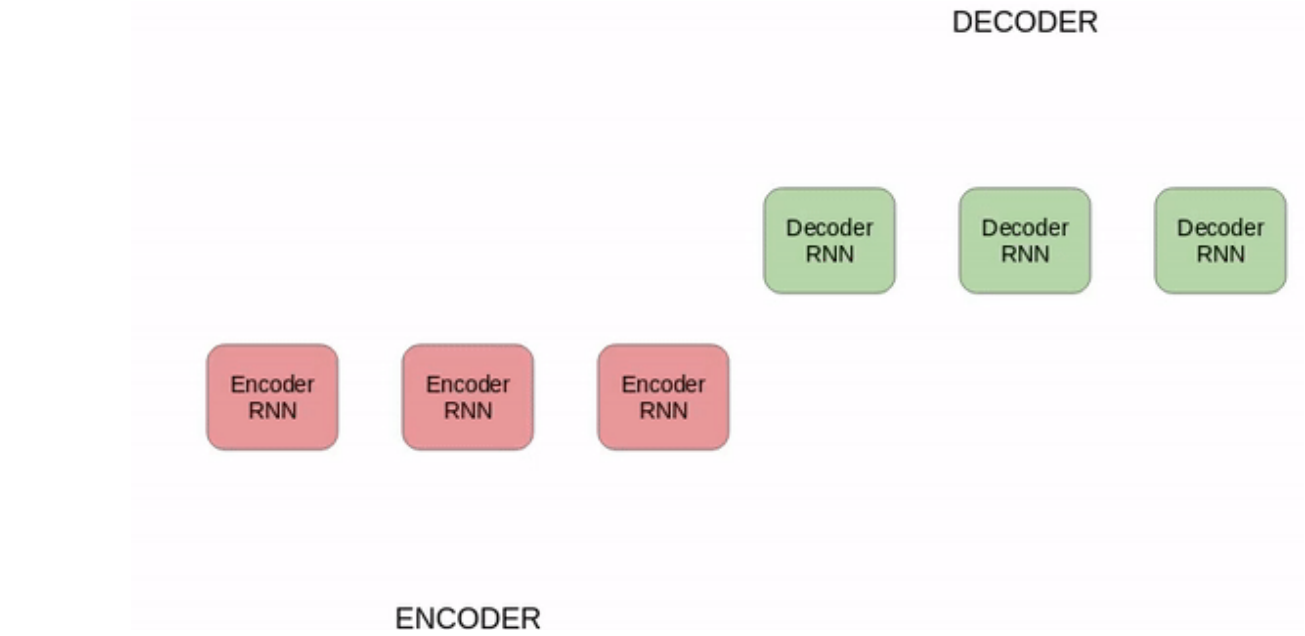
The performance of these seq2seq models was further enhanced with the addition of the **Attention Mechanism** in 2015. How quickly advancements in NLP have been happening in the last 5 years – incredible!

These sequence-to-sequence models are pretty versatile and they are used in a variety of NLP tasks, such as:

- Machine Translation
- Text Summarization
- Speech Recognition
- Question-Answering System, and so on

RNN based Sequence-to-Sequence Model

Let's take a simple example of a sequence-to-sequence model. Check out the below illustration:



German to English Translation using seq2seq

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
- 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

Introduction to the 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.

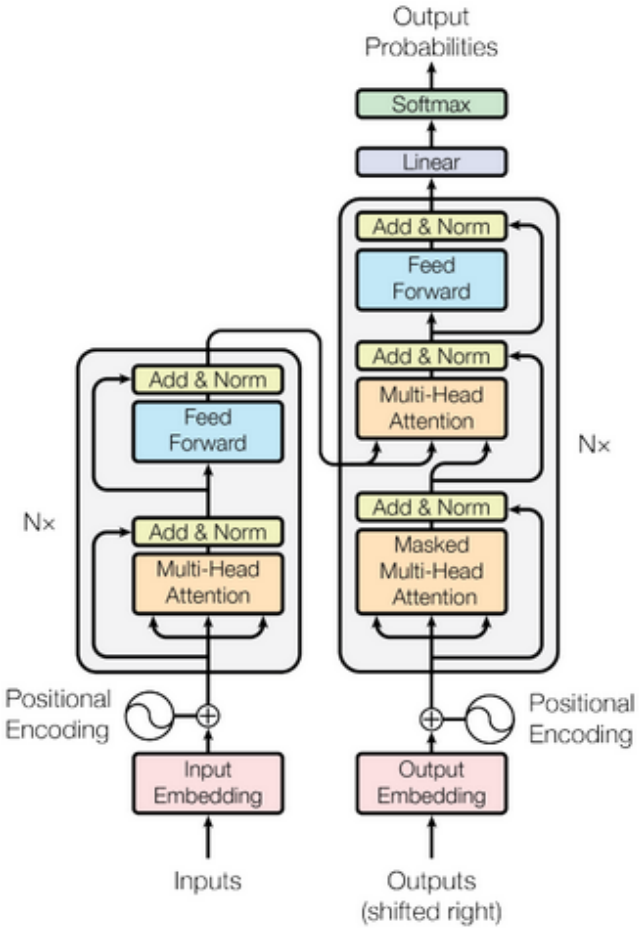
Quoting from the paper:

> *"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.

Let's take a look at the architecture of the Transformer below. It might look intimidating but don't worry, we will break it down and understand it block by block.

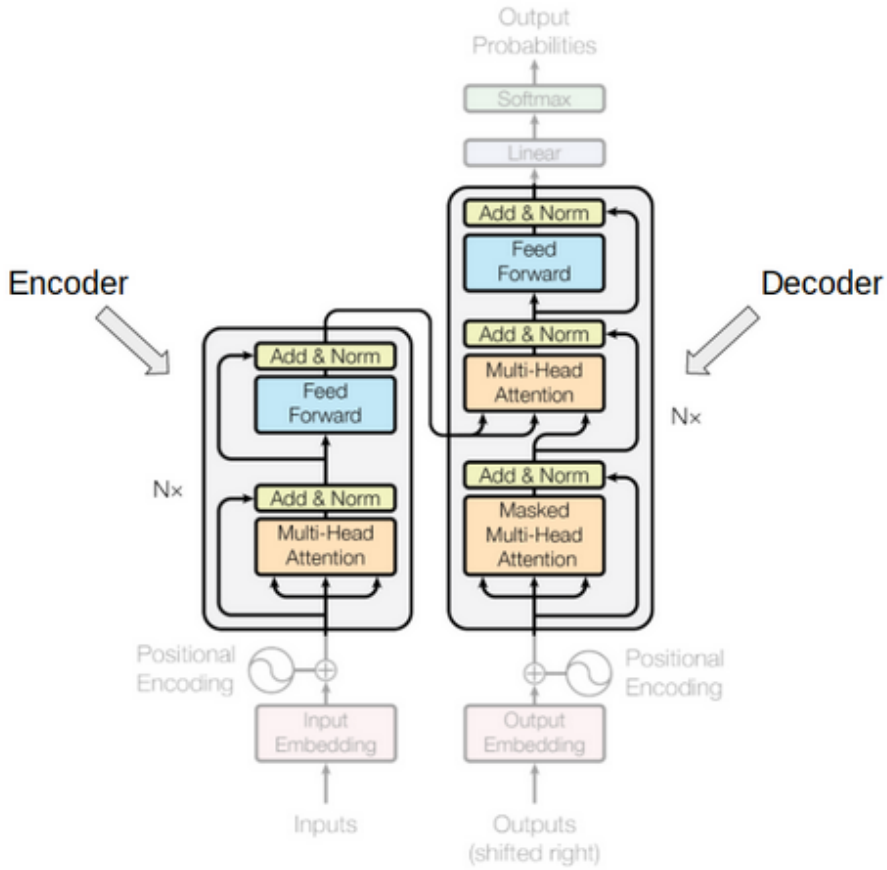
Understanding Transformer's Model Architecture



The Transformer – Model Architecture
(Source: https://arxiv.org/abs/1706.03762)

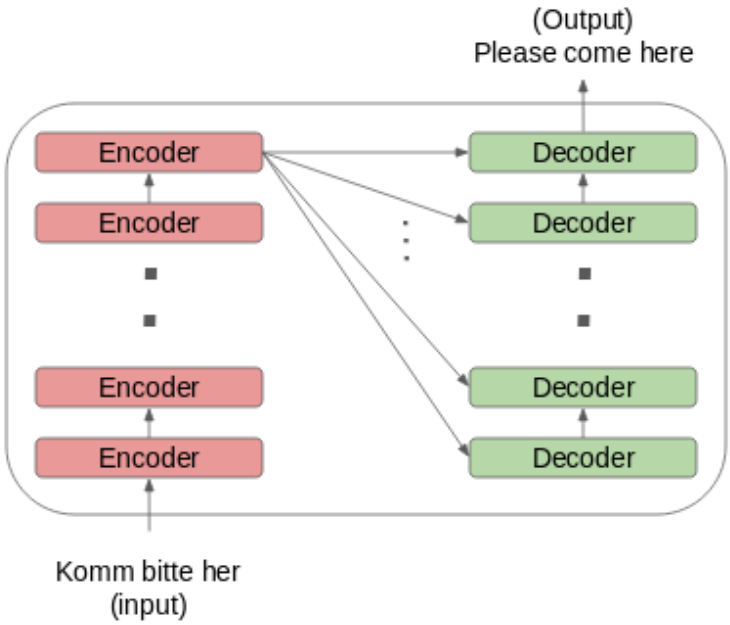
The above image is a superb illustration of Transformer's architecture. Let's first focus on the **Encoder**and **Decoder** parts only.

Now focus on the below image. 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 **MaskedMulti-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.

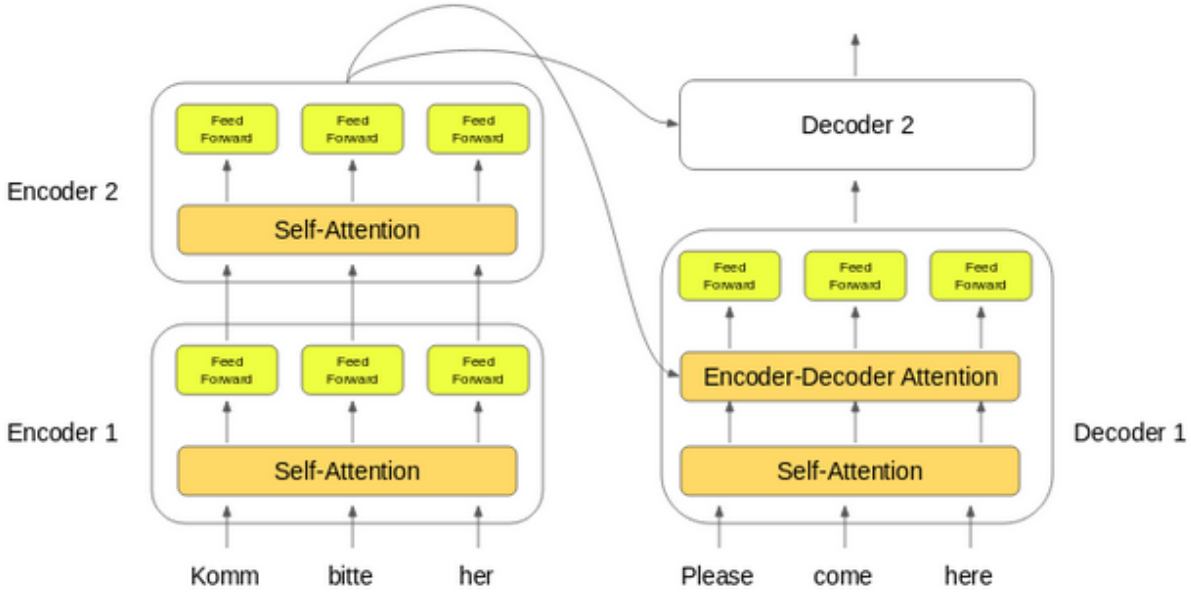
The number of encoder and decoder units is a hyperparameter. In the paper, 6 encoders and decoders have been used.



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:



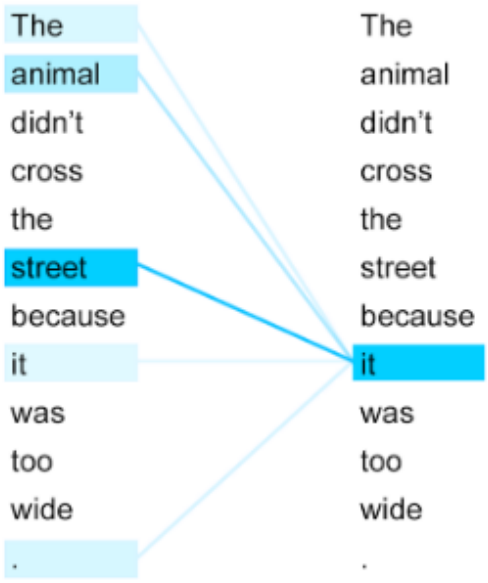
An important thing to note here – in addition to the **self-attention** and feed-forward layers, the decoders also have one more layer of Encoder-Decoder Attention layer. This helps the decoder focus on the appropriate parts of the input sequence.

You might be thinking – what exactly does this "Self-Attention" layer do in the Transformer? Excellent question! This is arguably the most crucial component in the entire setup so let's understand this concept.

Getting a Hang of 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."*



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.

Calculating Self-Attention

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:

    1. Query Vector
    2. Key Vector
    3. 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

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

    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:

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	$q_1 . k_3$

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:

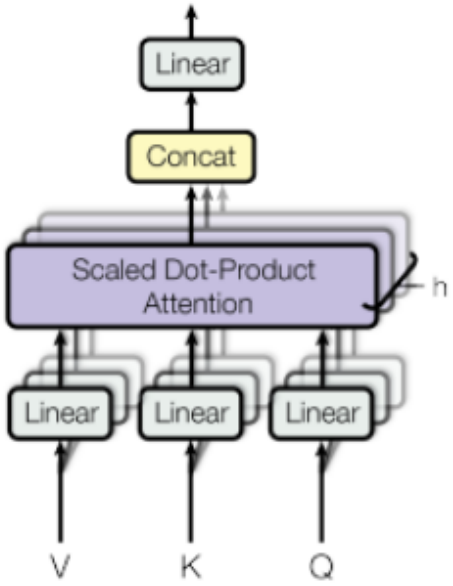
Word	q vector	k vector	v vector	score	score / 8	Softmax	Softmax * v	Sum
Action	q_1	k_1	v_1	$q_1 . k_1$	$q_1 . k_1 / 8$	x_{11}	$x_{11} * v_1$	z_1
gets		k_2	v_2	$q_1 . k_2$	$q_1 . k_2 / 8$	x_{12}	$x_{12} * v_2$	
results		k_3	v_3	$q_1 . k_3$	$q_1 . k_3 / 8$	x_{13}	$x_{13} * v_3$	

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 below:



Multi-Head Attention

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."*

You can access the code to implement Transformer here.

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.

Understanding Transformer-XL

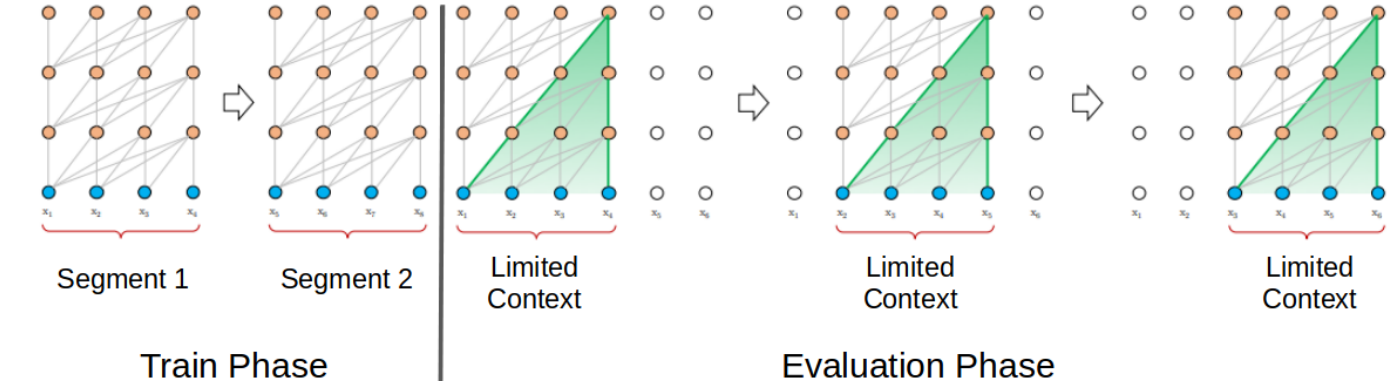
Transformer architectures can learn longer-term dependency. However, they can't stretch beyond a certain level due to the use of fixed-length context (input text segments). A new architecture was proposed to overcome this shortcoming in the paper – Transformer-XL: Attentive Language Models Beyond a Fixed-Length 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:

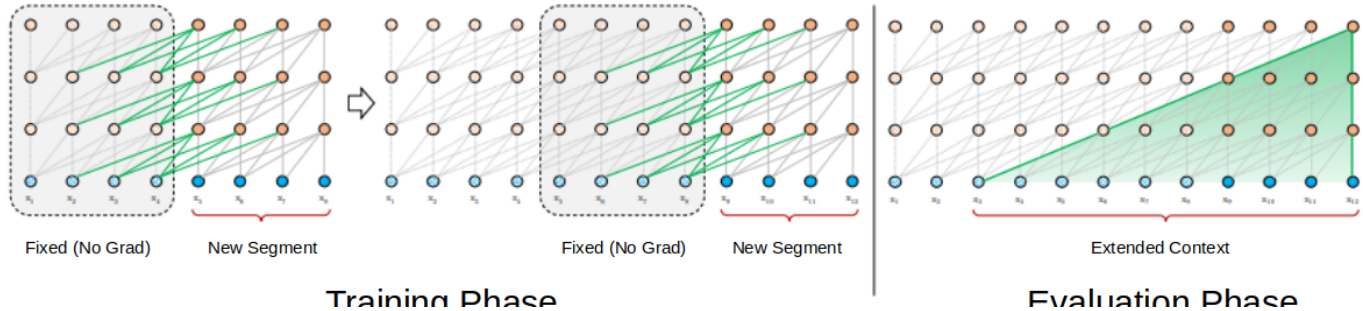


Transformer Model with a segment length of 4 (Source: https://arxiv.org/abs/1901.02860)

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-XL 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

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.

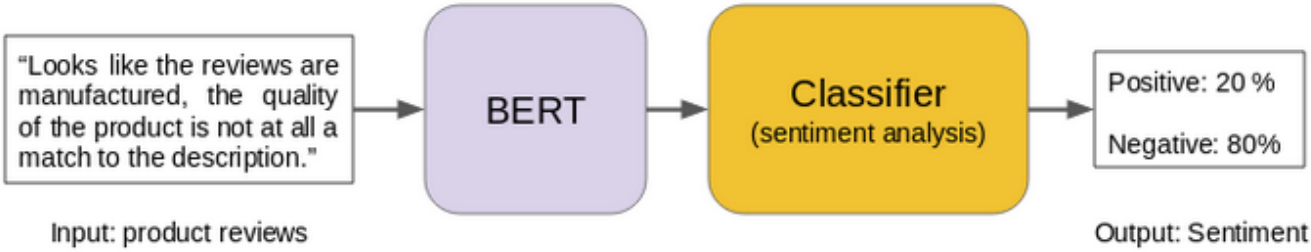
You can access the code to implement Transformer-XL here.

The New Sensation in NLP: Google's BERT (Bidirectional Encoder Representations from Transformers)

We all know how significant **transfer learning** has been in the field of computer vision. For instance, a pre-trained deep learning model could be fine-tuned for a new task on the ImageNet dataset and still give decent results on a relatively small labeled dataset.

Language model pre-training similarly has been quite effective for improving many natural language processing tasks: (https://paperswithcode.com/paper/transformer-xl-attentive-language-models and https://paperswithcode.com/paper/transformer-xl-attentive-language-models).

The BERT framework, a new language representation model from Google AI, uses pre-training and fine-tuning to create state-of-the-art models for a wide range of tasks. These tasks include question answering systems, sentiment analysis, and language inference.

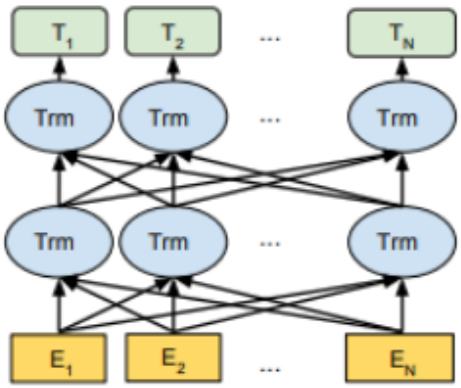


Fine-Tuning BERT on a Sentiment Analysis Task

BERT's Model Architecture

BERT uses a multi-layer bidirectional Transformer encoder. Its self-attention layer performs self-attention in both directions. Google has released two variants of the model:

1. **BERT Base**: Number of Transformers layers = 12, Total Parameters = 110M
2. **BERT Large**: Number of Transformers layers = 24, Total Parameters = 340M



BERT Architecture

BERT uses bidirectionality by pre-training on a couple of tasks — **Masked Language Model** and **Next Sentence Prediction**. Let's discuss these two tasks in detail.

BERT Pre-Training Tasks

BERT is pre-trained using the following two unsupervised prediction tasks.

1. Masked Language Modeling (MLM)