Introduction to the Transformer Model

Hi everyone. Welcome to part 2 of the course. In this part, I will introduce and explain the concepts and features of the Transformer, an AI model that has revolutionized not only the field of Natural Language Processing (NLP) but the entire world of artificial intelligence.

--

The Transformer model was introduced in the paper titled "Attention Is All You Need," by Vaswani et al. in 2017.

Why is this publication considered a game-changer? The reason is that it addressed longstanding challenges in NLP up until 2017, paving the way for advanced AI models capable of understanding and generating human-like text.

---

In this part, we will explore several key topics in Natural Language Processing (NLP), focusing particularly on the impact and workings of Transformer model.

We'll start by examining NLP techniques before and after the advent of Transformers, highlighting the significant advancements they brought.

Next, we'll take a closer look into the inner workings of Transformer model, mastering its architecture block-by-block to understand how they function.

We will also cover the training process of Transformer, discussing the methods and strategies used to train this powerful model effectively.

Finally, we'll look at the inference process, understanding how Transformer generates predictions in a real-world task.

This part will equip you with a solid understanding of Transformer and its role in modern NLP.

---

Let’s start discussing the NLP landscape before and After Transformer’s arrival

---

Let's first examine which models were predominant in NLP before the advent of the Transformer.

Up until 2017, Recurrent Neural Networks (RNNs) and their variations, such as Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRU), were the primary architectures for NLP tasks.

--

RNNs offer several advantages, including, but not limited to:

* Efficient management of sequential data types, including text, speech, and time series.
* The capability to handle inputs of varying lengths, a feature absents in feedforward neural networks.
* Improved training efficiency through the sharing of weights across different time steps.

--

Now, let's explore some of the major limitations of RNNs.

Consider the task of language modeling as an example. Given the beginning of a sentence, such as "She stands up and opens the...", the model's objective is to predict the next word.

RNNs process words one at a time, generating a hidden state for each timestep that feeds into the computation of the next state.

This sequential processing limits opportunities for parallel computation, resulting in longer computation times for longer sequences.

--

Furthermore, due to the sequential nature of computing these hidden states, the influence of early states or information on the final state or prediction diminishes for contexts involving long sequences.

This leads to a loss of information for long-range dependencies.

--

Another significant limitation of RNNs is their susceptibility to vanishing or exploding gradients. RNNs update their weights using Backpropagation Through Time.

For instance, to compute the gradient of the loss function L with respect to the network's weights, the chain rule is applied across timesteps to calculate the product of gradients.

If the gradients are less than 1, their magnitude decreases with each multiplication. Over many timesteps, the gradient can diminish to zero, rendering the weight updates during optimization negligible. This hampers the learning process, particularly for long sequences or deep architectures.

On the other hand, the exploding gradient problem arises when gradients at each timestep exceed 1, causing the gradients to exponentially increase as they are propagated backward in time. This can lead to extremely large gradient values, causing instability during optimization and resulting in weight updates that either oscillate or diverge, thereby destabilizing the training process.

Both vanishing and exploding gradients significantly complicate the training of RNNs, impairing their ability to capture long-term dependencies in sequential data. These challenges render the training of RNNs unstable and exceedingly difficult.

--

To summarize, RNNs face several significant disadvantages:

* Sequential computation makes parallel processing with GPUs challenging.
* They often lose information pertaining to long-term dependencies.
* They are prone to vanishing or exploding gradient issues.

--

And then, the arrival of the Transformer model in 2017, as proposed in the paper "Attention Is All You Need," marked a significant advancement.

--

The Transformer architecture was initially introduced for the task of machine translation, such as translating from one language to another, for instance, from English to French. At a high-level overview, its architecture comprises an encoder and a decoder. In this section, we will dive into the two main components of the Transformer.

--

Let's see what aspects allowed the Transformer to outperform RNNs and become a groundbreaking innovation at the time.

Transformer employs self-attention mechanisms, enabling parallel computation, which allows for the use of GPUs to expedite the training process significantly.

Moreover, it can effectively capture long-range dependencies. Importantly, its architecture makes the model less susceptible to the problems of vanishing or exploding gradients.

--

Following the success of the Transformer, from 2018 onwards, there has been a trend in NLP towards developing pre-trained language models based on the Transformer architecture. Examples of these include BERT, GPT, T5, Llama, Mistral, Phi, Falcon, Qwen, Gemma.

----

Now, Let's dive deep into transformer’s block by block.

---

In this session, we will answer the following questions

How does the model understand the text data.

And what are the input, output, and process of each block

-----

First of all, I am going to discuss the Input/Output blocks in the transformer architecture.

--

In the case of the Transformer, the model is used for machine translation task. Therefore, the Inputs will be in the source language, and the Outputs will be in the target language. For example, we have a pair of source-target texts, like in English: "I am fine." In French: "Je vais bien." Or in English: "Thank you very much," in French: "Merci beaucoup." This means that the Inputs/Outputs of the models are text data.

--

There are three points we need to consider:

Firstly, the source and target texts may have different lengths, but the model requires a fixed length for the vector that represents the sequence to perform matrix computations. Therefore, we need to predefine a fixed length for all the sequence vectors.

Secondly, the model needs to know when to start and end the prediction. We need to find ways to indicate to the model when to begin and end a sequence, such as using SOS (Start Of Sentence), EOS (End Of Sentence), and PAD (Padding) to represent instances where the sentence length is shorter than the chosen fixed length.

Thirdly, it is important to understand that the model only comprehends numerical values. Therefore, we need to transform text into a numerical representation. By utilizing a tokenizer, we can convert text into numerical values and also add special numerical values to indicate Start Of Sentence (SOS), End Of Sentence (EOS), and Padding (PAD). For example, 2 for SOS, 3 for EOS, and 1 for PAD.

---

Let's dive deep into Tokenizers.

---

What is a tokenizer?

A tokenizer possesses a large vocabulary and enables the transformation of text into numerical representations.

A token can represent a word or a character.

Additionally, it defines special tokens, for example: UNK for Unknown; PAD for padding; SOS for Start of Sentence; and EOS for End of Sentence.

----

Let's explore the two main processes in a tokenizer:

The first is encoding, which means transforming text tokens into numerical representations.

The second is decoding, which means converting numerical representations back into text tokens.

----

Now, let's put it all together to prepare Inputs for Encoder part

For example, we have a batch of 3 English sentences: "I am fine," "Thank you very much," and "I cook French cuisine."

* These 3 English sentences have a maximum length of 4, so for simplicity, let's choose a fixed length of 7.
* Then, we add the special tokens SOS (Start of Sentence), EOS (End of Sentence), and PAD (Padding) to achieve a fixed sequence length for all sentences.
* Using the encode method of the English tokenizer, we obtain the final inputs for the encoder part.

--

Similarly, we follow the same process for preparing the inputs for the decoder part, in the Transformer paper, it named “Outputs (shifted right)”. The difference is that we only use the SOS (Start of Sentence) and PAD (Padding) tokens, and do not use the EOS (End of Sentence) token. With the SOS token, it signals the model to start the prediction. Additionally, we use the target tokenizer for the French language to encode the tokens.

--

Finally, Let’s prepare the target for loss calculation.

In this case, we only use two special tokens: EOS (End of Sentence) and PAD (Padding). The idea is to optimize the model so that it knows when to stop the prediction.

In this case, we also use the target tokenizer for the French language to encode the tokens.

--

Next, let’s talk about the Encoder Input and Decoder Input. As they share the same structure and building blocks—Input Embedding and Positional Encoding—we only need to examine one of them.

--

Well, let’s take a closer look at Encoder Input, there are two main components: Input Embedding and Positional Encoding.

They utilize the numerical representation of text data as inputs to generate a tensor of dimension (batch, seq, d\_model).

Then, their results are combined to obtain the encoder input. Now, we will discuss more in detail the functions of each component.

--

Let’s talk about Input Embedding.

Input Embedding takes inputs of 2 dimensions: batch size, sequence length (batch, seq) and generates a tensor of 3 dimensions: batch size, sequence length and feature size of d\_model (batch, seq, d\_model), meaning it transforms each token into a feature vector of length d\_model for that token.

Input embedding is a lookup table or matrix that stores embeddings of a fixed dictionary and size, which can either be pre-trained or learned during model training.

The primary function of the input embedding is to map raw input tokens into dense vectors of fixed dimensions, where each token's value corresponds to its index in the Input Embedding matrix.

This process captures the semantic meaning of the tokens.

Here is an example, how tokens from a sample input sentence like "I am fine" are encoded by a tokenizer into a sequence of numerical indices.

These indices are then used to retrieve the corresponding vectors from the Input Embedding matrix, producing an output matrix of dimensions (batch, seq, d\_model) that encapsulates the semantic information of the input tokens.

--

Next, I'm going to describe the positional encoding component.

Positional Encoding transforms an input tensor with two dimensions: batch size and sequence length (denoted as batch, seq), into a three-dimensional tensor characterized by batch size, sequence length, and a feature size of d\_model (batch, seq, d\_model).

This component is used to inject information about the position of each token in the sequence into the input embeddings. The calculation of positional encodings involves sine and cosine functions of different frequencies, ensuring each position in the sequence has a unique encoding.

The formulas provided show that for a given position and dimension index I, the encoding values are computed using sine for even indices and cosine for odd indices.

It is also noted that in the original Transformer model, these positional encodings are not learnable parameters.

---

Here is an example, given dmodel=4, let’s see how the positional encoding vector is composed for each token corresponding to its position in the sequence.

--

To put it all together, for example, if we have a batch size of 2 sequences, each sequence with a fixed length of 7 tokens, and each token represented by a 512-dimensional feature vector:

By applying Input Embedding and Positional Encoding to the inputs of dimensions (batch, seq), we obtain two tensors with dimensions (batch, seq, d\_model).

These tensors are then added together to form the Encoder Input.

This combination helps the model to capture both the semantic meaning of the tokens and their respective positions within the sequence.

----

Now, I will explain the Encoder block.

The input to the Encoder block is the Encoder Input, which we have previously discussed. It represents the meaning and positional features of each token or word in the sequence.

The Encoder block consists of four sub-layers: multi-head attention (including the self-attention mechanism), layer normalization, a feedforward network, and a residual connection.

--

First, let's discuss Multi-Head Attention and its self-attention mechanisms.

The input to this block is the Encoder Input, a batch of tensors that represent the semantic meaning and the positional feature of each word or token. This input is duplicated into three identical tensors, named Q (Query), K (Key), and V (Value). Let's examine what happens inside the Multi-Head Attention block.

Before defining what multi-head attention is, let's consider a simpler case: a single-head attention layer. In a single-head attention layer, it calculates Attention, which is formulated by this formular:

The component with the Softmax function is called the Self-Attention mechanism.

Let's examine a concrete example to understand why it's called Self-Attention.

--

For instance, with a batch size of 1, a sequence length of 3, and d\_model equals d\_k equal to 512, applying this function will yield a matrix of dimension 3 by 3. The Softmax function normalizing the values of each row to range between 0 and 1, while ensuring the sum of all values in each row equals 1. This matrix, representing the correlation between words in a sequence, is why it's referred to as Self-Attention.

--

Afterward, the result of the Self-Attention is multiplied by V to obtain the output of the single-head Attention.

The Self-Attention mechanism highlights the correlation between words in a sentence, while V encapsulates both word meaning and positional features.

Consequently, the final attention tensor represents a synthesis of these elements.

Having understood the concept of self-attention for a single-head attention, let's extend this understanding to multi-head attention.

--

I am going to explain multi-head attention. Multi-head attention involves four main steps:

1. In the first step, we clone the encoder input into three identical tensors Q (queries), K (keys), and V (values)

Each of the tensors Q, K, V has dimensions: [batch size,sequence length,d\_model].

Then, we apply *h* separate linear transformations to each of these tensors by multiplying them with *h* separate parameter matrices, which project them into smaller-dimensional spaces dk for each head.

This step allows for parallel processing of information, enhancing the model's ability to focus on different parts of the input sequence simultaneously.

1. In the second step, we calculate attention for each head, similar to how we calculate for a single-head attention.
2. Next, we concatenate the attention tensors from each head. This step combines the independently computed attention outputs into a single tensor, preserving the information captured by each head.
3. Finally, we apply another linear layer to the concatenated tensor to obtain the final multi-head attention tensor of dimension (batch, seq, d\_model), which is ready for subsequent processing.

Now that we have covered multi-head attention, let's continue to discuss another layer in encoder block.

--

Next, I am going to explain layer normalization as used in an encoder block.

Layer Normalization focuses on the normalization of the feature vector for each token, which has a dimensionality of *d\_*model​. The process involves two main steps:

1. The first step involves computing the mean and variance of the values across the *d\_*model​ dimensions.

The variance is calculated to measure how much the values of the feature vector vary from the mean value, indicating the dispersion of the feature vector values around the mean.

1. In the second step, we use the layer normalization formula to calculate the normalized feature vector.

It adjusts each feature vector by subtracting the mean and dividing by the square root of the variance, with a small constant *ε* added to the variance to prevent division by zero.

This normalization process ensures that the activations across the features for each token have a mean of 0 and a standard deviation of 1.

This step improves the stability and performance of the neural network by ensuring consistent scale across features.

By ensuring each feature vector has consistent statistics (mean and variance), layer normalization helps to stabilize the learning process and improve the model's convergence.

Putting it all together: Given a tensor with the dimensions of (batch, seq, d\_model), layer normalization normalizes the values within this tensor, resulting in an output tensor with the same dimensions but with normalized values.

--

Next, we'll discuss the feedforward layer.

Given an input tensor *x* with dimensions (seq, d\_model), the original Transformer architecture proposed the application of two linear layers to *x*:

* With the first linear layer, we multiply *x* by a weight matrix of dimensions (d\_model, d\_ff) and then add a bias, b1.
* We then apply the ReLU function to the result of the first linear layer, followed by the application of the second linear layer to the resulting tensor, with a weight matrix *W*2​ of dimensions (d\_ff, d\_model) and a bias *b*2​.
* The output tensor retains the dimension (seq, d\_model).

--

Next, I am going to talk about the residual connection:

Given an input tensor *x*, a sublayer and a residual connection,

A sublayer can be either a feedforward layer or multi-head attention

We can obtain the resulting tensor by adding *x* and the output of the sublayer applied to *x*.

--

We have now gone through all the building blocks of the encoder part, including multi-head attention with self-attention mechanism, layer normalization, feedforward layer, and residual connection.

In the Transformer architecture, it uses six identical encoder layers to formulate the encoder part

--

--

Next, I'm going to tell you about the Decoder part.

For the Decoder part, most blocks are the same as those in the Encoder part, except there are some differences in Multi-Head Attention and the inclusion of Masked Multi-Head Attention.

* In Multi-Head Attention within the Decoder, the Key and Value tensors come from the Encoder output, while the Query tensor comes from the previous block in the Decoder part.
* Another important block is the Masked Multi-Head Attention. It functions similarly to Multi-Head Attention, except that a mask is applied during the calculation of the Self-Attention score to prevent future tokens from influencing the prediction of the current token. We will discuss this in more detail soon.
* Finally, regarding the sequence length in the Decoder part: For training, the sequence length of the Decoder is equal to the sequence length of the Encoder, meaning the maximum sequence length. However, during inference, the sequence length of the Decoder can vary from 1 to the maximum sequence length.

--

Now, let's dive deeper into Masked Multi-Head Attention:

The major difference in the calculation of Self-Attention within this context is the addition of a mask (set to -infinity) to the values above the diagonal line of the Self-Attention matrix.

Applying the softmax function then ensures that positions with the mask are assigned a value of 0.

The objective is to force each word in the target language to correlate only with the preceding words in that sentence, not the following words. This is crucial for ensuring the model generates text in a left-to-right manner, preserving the sequential nature of language.

--

Next, I am going to talk about the Multi-Head Attention in the Decoder.

The first point to note is that the Key and Value tensors are derived from the Encoder output, while the Query tensor comes from the preceding block of the Decoder.

Consequently, the shape of Q dictates the shape of the outputs of Multi-Head Attention in the Decoder.

Regarding sequence lengths (*seq*−*de*): For training, the sequence length of the Decoder is equal to the sequence length of the Encoder, meaning the maximum sequence length. However, during inference, the sequence length of the Decoder can vary from 1 up to the maximum sequence length.

--

Having covered all the building blocks and the important differences of the Decoder compared to the Encoder, we can proceed to construct the Decoder block. In the original Transformer paper, the authors proposed applying six identical Decoder blocks to form the final Decoder layer part.

--

Finally, we will discuss the prediction output part that includes the two last layers used to generate the Transformer output: Projection with a linear layer and Softmax.

To derive the Transformer's predictions, we start with a Decoder output *x of dimensions: batch size, sequence length, d\_model* (batch, *seqde*​, *dmodel*​). Here for the demonstration purpose I just consider one predicted token

* First, we apply a linear layer to project *x* to y. This transformation maps the sequence feature of length *d\_model*​ to the vocabulary size (*vocab*\_*size*).
* Then we apply the Softmax function to normalize the feature vector of length *vocab*\_*size* to a range between 0 and 1, with the sum of the probabilities equaling 1.
* By identifying the index corresponding to the maximum value, we can pinpoint the token with the highest probability of occurrence.
* Taking its index, we then use the decode method for the tokenizer of the target language to convert this index back into the corresponding word.
* As a result, we obtain the predicted word.

--

Congratulations! You have done it! Now take a bow. We have gone through all the building blocks of the Transformer. Such an achievement! Now, we can build the Transformer model. Next step, let's see how the model is trained and how to perform the inference for the machine translation task.

--

Let's examine what happens during the training of a Transformer, for the task of machine translation:

For instance, in this illustration, the batch size is set to 1.

Initially, we have a pair of sentences. The source language in English: 'It’s good.' And the target language, in French: ‘C’est bon’  
The model processes the entire text batch simultaneously in a parallel manner.

The predicted sequence generated by the decoder is compared to the target sequence to calculate the loss, using the Cross Entropy Loss function.

Afterward, it updates the model parameters in a way that reduces the model’s loss.

For the next time step, the entire training process is repeated.

Regarding the Cross Entropy Loss function: what is it, and how is it implemented? I will provide an explanation in the implementation part, where I present how to implement a transformer in practice. See you there.

At this point, you should have a high-level understanding of the model training process.

--

Now we talk about the inference process of transformer. For example, let's say we want to translate the English sentence "I am fine" into French.

1. At the first step, the English sequence is fed into the encoder, while only the special token SOS – Start of Sentence is processed by the decoder. This special token allows the decoder to predict the first word: "Je".
2. The model is auto-regressive, meaning it consumes the previously generated word as additional input when generating the next.

So, at the second step, the word "Je" is concatenated back to the existing input of the decoder, resulting in the decoder input as "SOS Je".

In the encoder, we don't need to recompute the processing; we just need to use the encoder output. As a result, the decoder predicts the next word "vais".

1. For the third step, the process repeats, and the word "vais" is added to the existing decoder input, resulting in "SOS Je vais". The encoder output is reused for the computation in the decoder. As a result, the decoder predicts the word "bien".
2. At the fourth step, the process repeats, and finally, we obtain the special token EOS, which indicates that the model's prediction has reached its end. At that point, we obtain the final prediction in French: "Je vais bien."

Well, congratulation, you should have now a high-level understanding of the model inference process.

--

Well, you have now finished Part 2, ‘Introduction to the Transformer’ . Let’s summarize what we have learned so far.

In this part, we have covered several key aspects of the evolution of Natural Language Processing (NLP) since the advent of the Transformer model in 2017.

We began by exploring the NLP landscape before and after the Transformer's arrival, comparing the advantages and disadvantages of Recurrent Neural Networks (RNNs) and discussing how the Transformer has revolutionized AI and NLP.

We then explored the structure of the Transformer, examining it block-by-block to understand its components and functionality.

Finally, we reviewed the processes involved in training and inference within the Transformer.

Let’s continue our learning journey

**Title:**

Transformer Explained: A Comprehensive Guide to 'Attention Is All You Need'

**Description:**

In this video, I will introduce and explain the concepts and features of the Transformer, an AI model that has revolutionized not only the field of Natural Language Processing (NLP) but the entire world of artificial intelligence.

The Transformer model was introduced in the paper titled "Attention Is All You Need," authored by Vaswani and his colleagues in 2017, under the work at Google Brain and Google Research.

Below are the sections I will walk you through:

0:00 – Introduction to the Transformer

* 00: 21 Introduction to “Attention Is All You Need” by Vaswani et al., 2017
* 01:03 NLP until 2017 – RNN models - Advantages and Drawbacks of RNN
* 06:05 The arrival of Transformer – A game changer!

08:39 – Transformer Inputs – From text to numerical representation

* 10:58 Tokenizer
* 12:05 Transforming text to numerical representation

14:15 – Introduction to Transformer blocks

14:31 – Transformer – Encoder input & Decoder input

* 15:29 Input Embedding
* 15:53 Positional Encoding
* 16:57 Encoder / Decoder input – Putting it all together

17:35 – Transformer – Encoder

* 18:10 Multi-head attention, self-attention
* 22:59 Layer Normalization
* 25:31 Feed Forward layer
* 26:32 Residual connection
* 27:01 Encoder – Putting it all together

27:29 – Transformer – Decoder

* 29:05 Masked multi-head attention
* 30:04 Decoder multi-head attention
* 31:04 Decoder – Putting it all together

31:31 – Transformer output: Make predictions

33:09 – Transformer – Take a bow!

33:31 – Transformer training process

35:09 – Transformer inference process

37:45 – Thank you!

---

Credit:

***Attention Is All You Need, A. Vaswani et al, 2017, https://arxiv.org/abs/1706.03762***

The video is made using Camtasia with a purchased license. Music from the Camtasia library: "Tuesday."

The presentation and demonstration were created using Microsoft PowerPoint, Office 365, with a purchased license.

If there are any points you don't understand, please let me know in the comments, and I will try to answer them. Or if you find any mistakes, please let me know so I can correct them and improve for next time.

Finally, if you like my video, you can encourage me by subscribing to my channel, liking, and sharing my video with your friends or on your social networks. You can also follow me on Github and star my repository if you find it helpful. That helps others find it easily.

That will motivate me a lot for the upcoming videos. Let's learn and grow together. Thank you very much!

Medium

In this video, I will introduce and explain the concepts and features of the Transformer, an AI model that has revolutionized not only the field of Natural Language Processing (NLP) but the entire world of artificial intelligence.

The Transformer model was introduced in the paper titled "Attention Is All You Need," authored by Vaswani et al. in 2017, under the work at Google Brain and Google Research.

Below are the sections I will walk you through.

Introduction to the Transformer

* Introduction to “Attention Is All You Need” by Vaswani et al., 2017
* NLP until 2017 – RNN models - Advantages and Drawbacks of RNN
* The arrival of Transformer – A game changer!

Transformer Inputs – From text to numerical representation

* Tokenizer
* Transforming text to numerical representation

Introduction to Transformer blocks

Encoder input & Decoder input

* Input Embedding
* Positional Encoding
* Encoder / Decoder input – Putting it all together

Encoder

* Multi-head attention, self-attention
* Layer Normalization
* Feed Forward layer
* Residual connection
* Encoder – Putting it all together

Decoder

* Masked multi-head attention
* Decoder multi-head attention
* Decoder – Putting it all together

Transformer output: Make predictions

Take a bow!

Transformer training process

Transformer inference process

Thank you!