Intro to Transformer

Hi everyone, in this video I will explain to you all the concepts and features of Transformer, an AI model that has been revolutionizing not only the NLP space but also the whole AI world.

The Transformer model is proposed in the famous paper: ‘Attention Is All You Need’, published by Vaswani and co-authors in 2017 at Google Brain and Google Research.

But why this publication is a game changer? The answer is that it solved the problems remaining in NLP for a long time till 2017, and then open the way to the super AI model that can understand and generate human-like language.

So first, let’s take a look which models have dominated NLP before Transformer.

Until 2017, the common architectures like Recurrent Neural Networks (RNNs), and its variants Long Short-Term Memory (LSTM) networks, Gated Recurrent Units (GRU) were the mainstream in NLP tasks.

RNNs offer several benefits, including but not limited to:

* Efficient handling of sequential data types such as text, speech, and time series.
* Ability to process inputs of variable lengths, a feature lacking in feedforward neural networks.
* Enhanced training efficiency due to weight sharing across different time steps.

Now let’s see some major limitations of RNNs.

Take language modeling task as an example. Given a sentence starts with some words, like “She stands up and opens the …” ,the model needs to predict the next word. By using RNN, the model processes one word at a time step, to generate the hidden state for computation in the next time step, meaning that it is a sequential computation. As a consequence, there are less rooms for parallel computation. So longer sequence, longer computation time.

Next, due to its nature of sequential computation to product the hidden states, the contributions of initial states or information to the final state or the prediction are very small for long sequence context. This leads to the loss of information for long-range dependencies.

Another major limitation of RNNs is that they are prone to vanishing or exploding gradient problems. RNN use Backpropagation Through Time to updates the weights. For example, to calculate the gradient of loss function L with respect to the parameter weights of the network, it uses the chain rule, to calculate the product of gradients across time steps.

If the gradients are less than 1, each multiplication operation leads to a decrease in the magnitude of the gradient. If this process continues over many time steps, the gradient eventually diminishes to zero. As a result, the updates to the weights during optimization become insignificant, hindering the learning process, especially for long sequences or deep architectures.

Conversely, the exploding gradient problem occurs when the gradients at each time step are greater than 1, leading to exponential growth of gradients as they propagate backward through time, as a result the product of large gradients across time steps can result in extremely large gradient values. The large gradient values can cause instability during optimization, leading to weight updates that oscillate or diverge, making the training process highly unstable.

Both issues hinder the training of RNNs, affecting their ability to effectively capture long-term dependencies in sequential data. These problems make training RNN unstable and extremely hard.

To sum up, there are some major disadvantages of RNNs:

* + Sequential computation, hard to parallel computation with GPU
  + Loss of information for long-term dependencies
  + Vanishing or exploding gradient problems

And then, the arrival of Transformer in 2017 proposed by the paper “Attention Is All You Need”.

This paper proposed Transformer architecture for machine translation task, for example translate from one language to another, like from English to French. At the high level overview, its architecture includes encoder and decoder. In this video, we will deep dive into the two main blocks of Transformer.

What makes transformer so innovative at that time?

Transformer with Self-Attention mechanisms allow parallel computation, meaning that we can leverage GPU to accelerate the training process. Furthermore, It is able to capture long-range dependencies. Last but not least, its architecture allows the model less prone to vanishing or exploding gradient problems.

Followed by the success of Transformers, since 2018 onwards, there is a new trend in NLP to develop Pre-trained language models based on the Transformer architecture, to name a few: BERT, GPT, T5, Llama, Mistral, Phi, Falcon, OLMo.

----

Now let deep dive into transformer. Here is our plans of attack. I am going to explain to you

* Text data at input and output of the model. And how does the model understand the text data?
* Then, I will walk you through transformer block-by-block. I will talk about what are the input-output-process of each block. Furthermore, we will talk about its meaning.

-----

First of all, I am going to talk about Inputs / Outputs blocks in the transformer architecture. In the case of transformer

In the case of Transformer, the model deals with the machine translation task. Therefore, Inputs will be source language and the Outputs will be the target language.

For examples, 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”

It means that the Inputs / Outputs of the models are the text data

There are 3 points we need to consider.

First, It is important to know that the model only understands numerical value. So what we need to do is to transform text to numerical representation.

Second, the source and target texts may have different lengths, but the model needs a fixed length of sequence

Third, the model need to know when to start and to end the model’s prediction. So we need to find some ways to notify model when to start and to end a sequence.

----

To solve all of the concerns, we can

* Choose a fixed length
* Add special value to notify when to start and to end the sentence, also to know empty value when the length of the sentence is shorter the chosen sentence length.
* To deal with all of this we leverage tokenizer to transform from text to numerical representation

---

Let deep dive into Tokenizer

What is a tokenizer?

A tokenizer contains a large amount of vocabulary

It allows to transform text to numerical representations

A token can be a word or a character

It defines special tokens, for example: UNK: Unknown; PAD: padding

SOS: Start of Sentence, EOS: End of Sentence.

----

Let’s see two main process in Tokenizer: one is encode, meaning that it transform text tokens into numerical representations. Second is decode, meaning that it transforms numerical representations into text token.

----

Now let put it all together.

First, let’s prepare Inputs for source language. For example, we have a batch of 3 english sentence: “I am fine”, “Thank you very much”, “I cook French cuisine”.

* The 3 Engish sentences have the maximum length is 4, so for simplicity, for example, I choose fix length is 7,
* Then, we add the special token SOS, EOS, PAD to obtain the fixed sequence length of all sentence.
* Then, using encode method of English tokenizer, we obtain the final inputs for encoder part

Similarly, we do the same process for preparing Outputs of the decoder part. The difference is that we only use SOS and PADING, and don’t use EOS. With SOS token, it notifies the model to start the prediction. In addition, we use target tokenizer for French language to encode the tokens.

Finally, the third use case of encoding processing is to prepare the target for loss calculation. In this case, We only use 2 special tokens: EOS and PAD. The idea is that optimize the model so that it knows when to stop the prediction. In this case, we also use target tokenizer for French language to encode the tokens.

Next, I walk you through all the building block of Transformer architecture, including Encoder Input, Decoder Input, Encoder, Decoder, Projection and Transformer Output. Let’s dive in.

Well, now I am going to talk about Encoder Input and Decoder Input. As they share the same structure and building blocks, Input Embedding and Positional Encoding, so we just need to discuss one of them.

Well, for example in the case of Encoder Input, there are two main functions: Input Embedding and Positional Encoding.

They use the numerical representation of text data as inputs, to generate a tensor of dimension (batch, seq, d\_model). Then, their results are added together to obtain the encoder input.

Now we will discuss more in detailed the function.

Input Embedding, takes Inputs of dimension (batch, seq) to generate a tensor of dimension (batch, seq, d\_model), meaning that it transform each token into a feature vector of length d\_model of that token.

---

Positional Encoding, it transform the input tensor of dimension (batch, seq) to a tensor of dimension (batch, seq, d\_model), its function only depend on the position of the token in the sequence, then the position index in the feature vector of size d\_model. It conveys the positional information of a sequence

----

Encoder Input, put it all together, for example we have a batch size of 2, a sequence length of 7 and a d\_model of 512, applying Input Embedding and Positional Encoding to the Inputs of (batch, seq), we obtain two tensors with dimension of (batch, seq, d\_model), then they are added together to obtain the Encoder Input

----

Now I am going to explain Encoder block. The input of Encoder block is Encoder Input that we have discussed previously.

It consists of 4 layers: multi-head attention, in which we will talk about Self-attention mechanism, layer normalization, feed forward, and residual connection.

---

First, we will talk about Multi-Head Attention and its self-attention mechanisms.

The input of this block is Encoder Inputs, which is a batch of tensor that represents the word meaning and the position feature of each word or token. Then, we copy it into 3 identical tensors, namely Q, stands for Query, K, stands for Key, and V, stands for Value.

Ok, let’s see what happens inside the Multi-Head Attention block.

---

Before we define what is multi-head attention, let’s take a look at a simpler case with 1-head attention layer. In 1-head attention layer, it calculate Attention which is formulated like this.

The component with Softmax function is called Self-Attention mechanism. Let’s see a concrete example why it is called Self-Attention.

For example, a batch size of 1, a sequence length of 3, and d\_model equals d\_k equal to 512, when we apply this function, we will obtain a matrix of dimension 3 by 3, that represents the correlation of word to each others, the softmax allows to normalize the value of each row in the range of 0 and 1, while the sum of all values in each row is equal to 1.

This matrix represents the correlation between words in a sequence so it is called Self-attention

--

Afterwards, the Self-attention result is multiplied by V to obtain 1-head Attention. As can be seen that the Self-attention represents the correlation between words in a sentence, while V represents word meaning and positional features. As a result, the final attention tensor represents all of the things.

Now, we have the understanding of self-attention for 1-head attention. Let’s scale it into multi-head attention.

--

For multi-head attention, there are 4 main step to realize this.

Firstly, we split each encoder inputs, Q, K, V, into h heads for each input, by applying the linear layers.

Secondly, We calculate attention for each head, like we calculate for 1-head attention.

Next, We concatenate attention tensor of each head

Finally, we apply a linear layer to obtain the final multi-head attention tensor of dimension (batch, seq, d\_model)

Well we have did it so far. Let ‘s continue.

--