**Tutorial on deep transformer**

Loc Nguyen

Loc Nguyen’s Academic Network, Vietnam

Email: ng\_phloc@yahoo.com

Homepage: www.locnguyen.net

# Abstract

Development of transformer is a far progressive step in the long journeys of both generative artificial intelligence (GenAI) and statistical translation machine (STM) with support of deep neural network (DNN), in which STM can be known as interesting result of GenAI because of encoder-decoder mechanism for sequence generation built in transformer. But why is transformer being preeminent in GenAI and STM? Firstly, transformer has a so-called self-attention mechanism that discovers contextual meaning of every token in sequence, which contributes to reduce ambiguousness. Secondly, transformer does not concern ordering of tokens in sequence, which allows to train transformer from many parts of sequences in parallel. Thirdly, the third reason which is result of the two previous reasons is that transformer can be trained from large corpus with high accuracy as well as highly computational performance. Moreover, transformer is implemented by DNN which is one of important and effective approaches in artificial intelligence (AI) in recent time. Although transformer is preeminent because of its good consistency, it is not easily understandable. Therefore, this technical report aims to describe transformer with explanations which are as easily understandable as possible.

**Keywords:** transformer, generative artificial intelligence, statistical translation machine, sequence generation, self-attention.

# 1. Introduction

Artificial intelligence (*AI*) is recent trend in technological world, especially in computer science, in which artificial neural network (ANN, NN) is one of important subjects of AI. Essentially, *ANN* models or implements a complicated function ***y*** = *f*(***x***) where ***x*** = (*x*1, *x*2,…, *xm*)*T* and ***y*** = (*y*1, *y*2,…, *yn*)*T* are vectors so that ***x*** and ***y*** are imitated by input layer and output layer of ANN, respectively with note that each layer is composed of units called neurons *xi*, *yi*. The complication degree of function ***y*** = *f*(***x***) is realized by hidden layers of ANN which are intermediated layers between input layer and output layer. We denote:

Where Θ denotes parameters of ANN which are often weights and biases. Because *f*(***x*** | Θ) is essentially vector-by-vector function whose input and output are vectors, it should have denoted as ***f***(***x*** | Θ) but it is still denoted as *f*(***x*** | Θ) for convenience and moreover, input ***x*** and output ***y*** will be matrices if their elements *xi* and *yi* are vectors. If there are many enough hidden layers, ANN becomes a so-called deep neural network (*DNN*) such that DNN is cornerstone of the main subject of this report which is transformer because transformer, as its name implies, is the highly abstract and complicated version of function ***y*** = *f*(***x***). In other words, a transformer will make the transformation between complex and different objects if it is implemented by DNN or set of DNNs according to viewpoint of DNN. Although transformer can be applied into many areas, especially machine translation and computer vision, this report focuses on statistical machine translation (*STM*) because complex and different objects ***x*** and ***y*** in STM transformer are two sentences in two different languages where ***x*** is source language sentence and ***y*** is target language sentence. If ordering of elements *xi* / *yi* in vector ***x***/ ***y*** specifying sentence is concerned as ordering of words *xi* / *yi* in a sentence, transformer will relate to *sequence generation*. Therefore, transformer as well as STM are inspired from sequence generation which, in turn, relates to recurrent neural network (*RNN*) as well as long short-term memory (*LSTM*) because sequence generation models are often implemented by RNN or LSTM. The most standard ANN/DNN called feedforward network (*FFN*) follows the one-way direction from input layer to hidden layers to output layer without reverse direction, which means that there is neither connections from output layer to hidden layers nor connections from hidden layers to input layers. In other words, there is no cycle in FFN, which cause the side-effect that it is difficult to model a sequence vector ***x*** = (*x*1, *x*2,…, *xm*)*T* like a sentence in natural language processing (NLP) because elements / words / terms / tokens *xi* in such sequence/sentence vector have the same structure and every connection *xi* → *xi*+1 of two successive words *xi* and *xi*+1 is, actually, a cycle. This is the reason that recurrent neural network (RNN) is better than FFN to generate sequence. Therefore, we research transformer after researching sequence generation which is concerned after RNN is concerned. Note, sequence and sentence are two exchangeable concepts in this research.

Suppose entire feedforward network (FNN) is reduced into a state in RNN and RNN is ordered list of neurons called sequence of neurons and moreover, output of previous neuron *xi*–1 contributes to input of current neuron *xi*. Namely, for formal definition, given *T* time points *t* = 1, 2,…, *T*, then RNN is ordered sequence of *T* states and each state is modeled by triple (*xt*, *ht*, *ot*) called state (*xt*, *ht*, *ot*) where *xt*, *ht*, and *ot* represent input layer, hidden layer, and output layer, respectively. Without loss of generality, let *xt*, *ht*, and *ot* represent input neuron, hidden neuron, and output neuron, respectively when a layer is represented by one of its neurons. Please pay attention that *xt*, *ht*, and *ot* are represented vectors of the *t*th word in sentence ***x*** = (*x*1, *x*2,…, *xm*)*T* modeled by RNN in context of NLP because a word is modeled by a numeric vector in NLP. Therefore, the aforementioned sentence ***x*** = (*x*1, *x*2,…, *xm*)*T* is a matrix indeed but ***x*** is mentioned as a vector. Exactly, ***x*** is vector of vectors, which leads to the convention that its elements are denoted by bold letter such as ***x****i* or ***x****t* because such elements are variable vectors representing words. Note, a word in NLP can be mentioned as term or token.

Note, the superscript “*T*” denotes vector/matrix transposition operator. Whether the sentence / sequence is denoted as vector notation ***x*** or matrix notation ***X*** belongs to contextual explanations. Recall that transformer as well as STM are inspired from sequence generation which, in turn, is related to recurrent neural network (RNN) as well as long short-term memory (LSTM) because sequence generation models are often implemented by RNN or LSTM. Function ***y*** = *f*(***x*** | Θ) implemented by DNNs such as RNN and LSTM is also called generator because it is sequence generation model indeed. Therefore, although transformer is different from RNN and LSTM, all of them are denoted by generator ***y*** = *f*(***x*** | Θ) because they are sequence generation models indeed.

The *t*th element/word in sequence/sentence ***x*** = (***x***1*T*, ***x***2*T*,…, ***x****mT*)*T* is represented by the *t*th state (***x****t*, ***h****t*, ***o****t*) of RNN where ***x****t* is the *t*th input word and ***o****t* is the *t*th output word. If RNN models ***x*** = (***x***1*T*, ***x***2*T*,…, ***x****mT*)*T*, then *T* = *m* and so, if RNN models ***y*** = (***y***1*T*, ***y***2*T*,…, ***y****nT*)*T*, then *T* = *n*. By a convention, word and sentence are mentioned as token and sequence, respectively. Moreover, ***x*** is called source sequence and ***y*** is called target sequence or generated sequence. Mathematical equation to update RNN is specified as follows (Wikipedia, Recurrent neural network, 2005):

|  |  |
| --- | --- |
|  | (1.1) |

Where ***Wh***, ***Uh***, and ***Wo*** are weight matrices of current hidden neuron ***h****t*, previous hidden neuron ***h****t*–1, and current output neuron ***o****t*, respectively whereas ***bh*** and ***bo*** are bias vectors of ***h****t* and ***o****t*, respectively. Moreover, *σ****h***(.) and *σ****o***(.) are activation functions of ***h****t* and ***o****t*, respectively, which are vector-by-vector functions.

RNN copes with the problem of vanishing gradient when learning a long RNN of many states and so, long short-term memory (LSTM) is proposed to restrict the problem of vanishing gradient. State in RNN becomes cell in LSTM and so, given *T* time points *t* = 1, 2,…, *T*, let the pair (***c****t*, ***h****t*) denote LSTM cell at current time point *t* where ***c****t* represents real information stored in memory and ***h****t* represents clear-cut information that propagates through next time points. A cell (***c****t*, ***h****t*) has four gates such as forget gate ***f****t*, input gate ***i****t*, output gate ***o****t*, and cell gate ***g****t*. At every time point *t* or every iteration *t*, cell (***c****t*, ***h****t*) updates its information based on these gates as follows:

|  |  |
| --- | --- |
|  | (1.2) |

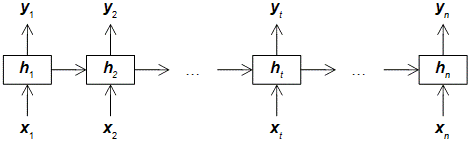
Note, ***W*(.)** and ***U*(.)** are weight matrices whereas ***b***(.) are bias vectors, which are parameters. The operator denotes wise-multiplication of two vectors where every pair of two corresponding elements of the two vectors are multiplied together, for instance, given two arbitrary vectors ***u*** = (*u*1, *u*2,…, *un*)*T* and ***v*** = (*v*1, *v*2,…, *vn*)*T*, we have ***uv*** = (*u*1*v*1, *u*2*v*2,…, *unvn*)*T*. Because core information of cell (***c****t*, ***h****t*) including ***c****t* and ***h****t* is calculated without any parameters, the problem of vanishing gradient can be alleviated when such gradient is calculated with regard to parameters such as weight matrices and bias vectors.

In general, when a sequence is modeled by a RNN or a LSTM, it is possible to generate a new sequence after RNN or LSTM is trained by backpropagation algorithm associated with stochastic gradient descent (*SGD*) algorithm. In other words, RNN and LSTM are important generation models although transformer is the main subject in this report because STM is, essentially, a sequence generation model that generates a new sentence in target language from a sentence in source language when sentence in NLP is represented by sequence. Because RNN and LSTM have the same methodological ideology, RNN is mentioned rather than LSTM because RNN is simpler one but they can be applied by exchangeable manner. For instance, given simplest case that source sequence ***X*** = (***x***1*T*, ***x***2*T*,…, ***x****mT*)*T* and target sequence also called generated sequence ***Y*** = (***y***1, ***y***2,…, ***y****n*)*T* have the same length *m* = *n*.

Generation model *f*(***x*** | Θ) is implemented by a RNN of *n* states (***x****t*, ***h****t*, ***o****t*) so that ***o****t* = ***y****t* for all *t* from 1 to *n*. After RNN was trained from sample by backpropagation algorithm associated with SGD, given source sequence ***X*** = (***x***1*T*, ***x***2*T*,…, ***x****nT*)*T*, target sequence ***Y*** = (***y***1*T*, ***y***2*T*,…, ***y****nT*)*T* is generated easily by evaluating *n* states of RNN.

|  |  |
| --- | --- |
|  | (1.3) |

Such generation process with *n*-state RNN is depicted by following figure:



**Figure 1.1.** RNN generation model

The next section will focus on sequence generation and attention which is a mechanism that improves generation process.

# 2. Sequence generation and attention

Recall that transformer as well as statistical translation machine (STM) are inspired from sequence generation which, in turn, is related to recurrent neural network (RNN) as well as long short-term memory (LSTM) because sequence generation models are often implemented by RNN or LSTM. Function ***y*** = *f*(***x*** | Θ) implemented by DNNs such as RNN and LSTM is also called generator because it is sequence generation model indeed. Because RNN and LSTM have the same methodological ideology, RNN is mentioned rather than LSTM.

Note, Θ denotes parameters of ANN which are often weights and biases whereas sequence is denoted as vector notation ***x*** or matrix notation ***X*** belonging to contextual explanations. This section focuses on sequence generation models such as RNN and LSTM before mentioning advanced concepts of transformer because, anyhow, transformer is next evolutional step of sequence generation models, especially in STM and natural language processing (NLP).

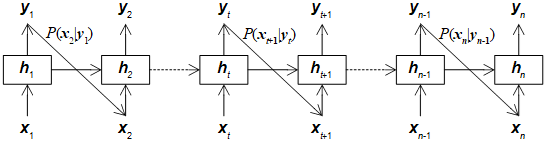
Given simplest case aforementioned that source sequence ***X*** = (***x***1*T*, ***x***2*T*,…, ***x****mT*)*T* and target sequence also called generated sequence ***Y*** = (***y***1*T*, ***y***2*T*,…, ***y****nT*)*T* have the same length *m* = *n*.

Generation model *f*(***X*** | Θ) is implemented by a RNN of *n* states (***x****t*, ***h****t*, ***o****t*) so that ***o****t* = ***y****t* for all *t* from 1 to *n*. After RNN was trained from sample by backpropagation algorithm associated with stochastic gradient descent (SGD) algorithm, given source sequence ***X*** = (***x***1*T*, ***x***2*T*,…, ***x****nT*)*T*, target sequence ***Y*** = (***y***1*T*, ***y***2*T*,…, ***y****nT*)*T* is generated easily by evaluating *n* states of RNN.

The simplest RNN generation needs to be extended if source sequence ***X*** is incomplete, for example, ***X*** has *k* token vectors ***x***1, ***x***2,…, ***x****k* where *k* < *n*. When ***X*** is incomplete, without loss of generality, given current output ***y****t*, it is necessary to predict the next output ***x****t*+1 (with suppose *t* > *k*). The prediction process, proposed by Graves (Graves, 2014), is based on estimating the predictive probability *P*(***x****t*+1 | ***y****t*) which is conditional probability of next input ***x****t*+1 given current output ***y****t*. As a result, RNN generation model is extended as follows (Graves, 2014, p. 4):

|  |  |
| --- | --- |
|  | (2.1) |

Following figure depicts the prediction model proposed by Graves (Graves, 2014, p. 3):



**Figure 2.1.** RNN prediction model

The problem here is how to specify predictive probability *P*(***x****t*+1 | ***y****t*). In the most general form, suppose joint probability *P*(***x****t*+1, ***y****t*) is parameterized by multivariate normal distribution with mean vector *μ* and covariance matrix Σ.

It is easy to estimate *μ* and Σ to determine *P*(***x****t*+1, ***y****t*) from sample by maximum likelihood estimation (MLE) method, for instance. Consequently, predictive probability *P*(***x****t*+1 | ***y****t*) is determined based on joint probability *P*(***x****t*+1, ***y****t*) as multivariate normal distribution with mean vector *μ*12 and covariance matric Σ12 specified as follows (Hardle & Simar, 2013, p. 157):

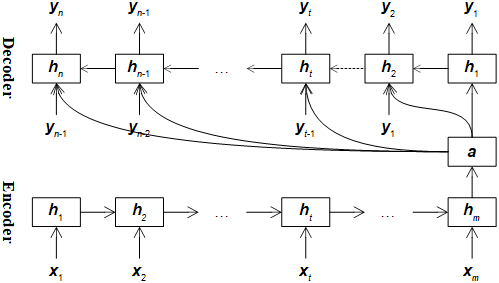
Because predictive probability *P*(***x****t*+1 | ***y****t*) gets highest at the mean *μ*12, it is possible to estimate ***x****t*+1 given ***y****t* by *μ*12.

|  |  |
| --- | --- |
|  | (2.2) |

The generation model above has only one RNN because source sequence ***X*** and target sequence ***Y*** have the same length. Some real applications, especially STM applications, require that lengths of ***X*** and ***Y*** are different, *m* ≠ *n*. This problem is called different-length problem.

Solution for different-length problem is to specify two RNNs: a RNN called *encoder* for ***X*** generation and the other one called *decoder* for ***Y*** generation. Intermediate vector ***a*** is proposed to connect encoder and decoder, which is called context vector in literature (Cho, et al., 2014, p. 2). The encoder-decoder mechanism is an important progressive step in STM as well as generative artificial intelligence (GenAI) because there is no requirement of mapping token-by-token between two sequences ***X*** and ***Y***, which is much more important than solving the different-length problem. On the other hand, sequence generation as well as its advanced development – transformer can also be classified into domain of GenAI.

According to Cho et al. (Cho, et al., 2014), context variable ***a***, which is last output of encoder, becomes input of decoder. Following figure depicts encoder-decoder model proposed by Cho et al. (Cho, et al., 2014, p. 2) with note that context vector ***a*** has fixed length.



**Figure 2.2.** Encoder-decoder model with fixed-length context

Note, both context and current token *t* are inputs of next token *t*+1. Moreover, there is an assignment ***y****t*+1 = ***o****t*. Therefore, each *t*th state of decoder is modified as follows:

Where ***Vh*** is weight matrix for context variable ***a***. Moreover, it may be not required to calculate output for each *t*th state of encoder. It may be only necessary to calculate hidden value of encoder.

In STM, given source sequence ***X*** and *t* target tokens ***y***1, ***y***2,…, ***y****t*, it is necessary to predict the next target token ***y****t*+1. In other words, predictive probability *P*(***y****t*+1 | Θ, ***X***, ***y***1, ***y***2,…, ***y****t*) needs to be maximized so as to obtain ***y****t*+1. Predictive probability *P*(***y****t*+1 | Θ, ***X***, ***y***1, ***y***2,…, ***y****t*) can be considered *likelihood* at the *t*th state of decoder. Consequently, parameter Θ of encoder-decoder model is maximizer of such likelihood.

|  |  |
| --- | --- |
|  | (2.3) |

Note, parameter Θ represents weight matrices and biases of RNN. By support of RNN and context vector ***a*** with implication of Markov property, likelihood *P*(***y****t*+1 | Θ, ***X***, ***y***1, ***y***2,…, ***y****t*) can become simpler:

Likelihood *P*(***y****t*+1 | Θ, ***X***, ***y***1, ***y***2,…, ***y****t*), which represents statistical language model, is object of maximum likelihood estimation (MLE) method for training encoder-decoder model (Cho, et al., 2014, p. 2). For example, the likelihood can be approximated by standard normal distribution, which is equivalent to square error function, as follows:

Where *f*(***X***, ***y***1, ***y***2,…, ***y****t* | Θ) denotes encoder-decoder chain.

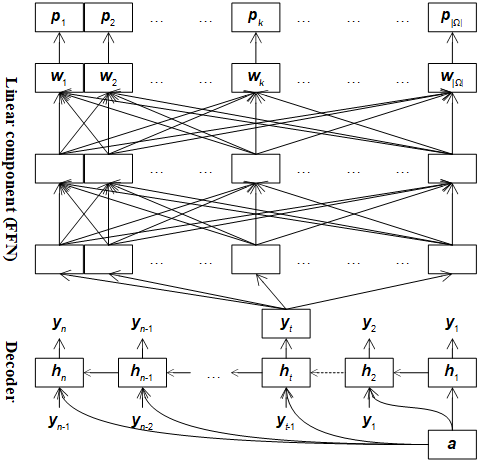
Therefore, training encoder-decoder model begins with MLE associated with backpropagation algorithm and SGD from decoder back to encoder.

Alternately, in STM with predefined word vocabulary, a simple but effective way to train encoder-decoder model is to replace likelihood *P*(***y****t*+1 | Θ, ***X***, ***y***1, ***y***2,…, ***y****t*) by a so-called linear component which is a feedforward network (FFN). Exactly, FNN maps the (*t*+1)th target token specified by token vector ***y****t*+1 to a weight vector ***w*** whose each element *wi* (0 ≤ *wi* ≤ 1) is weight of *i*th token (Alammar, 2018).

Length of weight vector ***w*** is the cardinality |Ω| where Ω is the vocabulary containing all tokens. After token weight vector ***w*** is determined, it is easily converted into output probability vector ***p*** = (*p*1, *p*2,…, *p*|Ω|)*T* where each element *pi* is probability of the *i*th token in vocabulary given the (*t*+1)th target token (Alammar, 2018).

|  |  |
| --- | --- |
|  | (2.4) |

Following figure depicts linear component.



**Figure 2.3.** Linear component of encoder-decoder model

It is interesting that likelihood *P*(***y****t*+1 | Θ, ***X***, ***y***1, ***y***2,…, ***y****t*) can be defined as output probability vector ***p*** = (*p*1, *p*2,…, *p*|Ω|)*T*. If the *i*th token is issued, its probability *pt* is 1 and other probabilities are 0.

Consequently, training encoder-decoder model begins with training linear component FFN(***y****t*+1) back to training decoder back to training encoder, which follows backpropagation algorithm associated stochastic gradient descent (SGD) method. Concretely, the following *cross-entropy* *L*(***p*** | Θ) is minimized so as to train FFN(***y****t*+1).

|  |  |
| --- | --- |
|  | (2.5) |

Where Θ is parameter of FFN(***y****t*+1) and the vector ***q*** = (*q*1, *q*2,…, *q*|Ω|)*T* is binary vector from sample whose each element *qi* has binary values {0, 1} indicating whether the *i*th token/word exists. For example, give sequence/sentence (“I”, “am”, “a”, “student”)*T*, if there is only one token/word “I” in sample sentence, the binary vector will be ***q*** = (1, 0, 0, 0)*T*. If three words “I”, “am”, and “student” are mutually existent, the binary vector will be ***q*** = (1, 1, 0, 1)*T*. When SGD is applied into minimizing the cross-entropy, partial gradient of *L*(***p*** | Θ) with regard to *wj* is:

|  |  |
| --- | --- |
|  | (2.6) |

Where,

Proof,

Due to:

We obtain:

So that gradient of *L*(***p*** | Θ) with regard to ***w*** is:

Therefore, parameter Θ is updated according to SGD associated with backpropagation algorithm:

Where *γ* (0 < *γ* ≤ 1) is *learning rate*. Please pay attention that ordering of source tokens is set from the end token back to the beginning token so that null tokens specified by zero vectors are always in the opening of sequence.

When encoder-decoder model is developed, context vector ***a*** becomes a so-called *attention*. The main difference between context vector and attention vector is that attention vector is calculated dynamically (customized) for each decoder state. Moreover, that context vector has fixed length restricts its prospect. Anyhow, attention mechanism fosters target sequence to pay attention to source sequence. In general, attention of a decoder state (token) is weighted sum of all encoder states (tokens) with regard to such decoder state. Suppose encoder RNN is denoted as follows:

For convenience, let ***s***1, ***s***2,…, ***s****m* denote *m* outputs of encoder such that:

Let score(***s****i*, ***h****t*) be score of encoder output ***s****i* and decoder hidden ***h****t* where score(***s****i*, ***h****t*) measures how much the *i*th token of source sequence modeled by encoder is close to the *t*th token of target sequence modeled by decoder. As usual, score(***s****i*, ***h****t*) is defined as dot product of ***s****i* and ***h****t* (Voita, 2023).

Where decoder hidden ***h****t* is:

Let weight(***s****i*, ***h****t*) be weight of encoder output ***s****i* and decoder hidden ***h****t* over *m* states of encoder, which is calculated based on soft-max function (Voita, 2023):

|  |  |
| --- | --- |
|  | (2.7) |

As a result, let ***a****t* be attention of source sequence ***X*** = (***x***1*T*, ***x***2*T*,…, ***x****nT*)*T* with regard to the *t*th token of target sequence ***Y*** = (***y***1*T*, ***y***2*T*,…, ***y****nT*)*T*, which is weighted sum of all encoder outputs with regard to such *t*th target token (Voita, 2023).

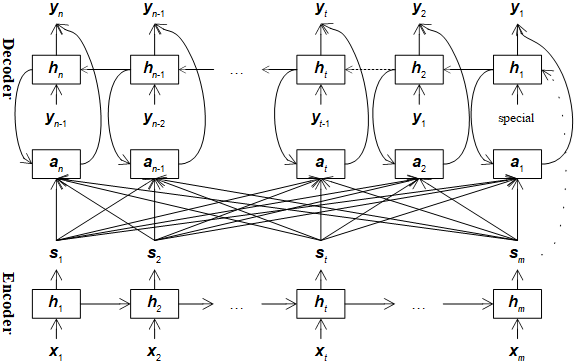
Obviously, ***a****t* becomes one of inputs of the *t*th token of target sequence ***Y*** = (***y***1*T*, ***y***2*T*,…, ***y****nT*)*T* such that:

Where ***Vo*** is weight matrix of attention ***a****t*. In general, decoder RNN associated with the attention mechanism called Luong attention (Voita, 2023) is specified as follows:

|  |  |
| --- | --- |
|  | (2.8) |

Where,

Following figure depicts encoder-decoder model with attention (Voita, 2023):



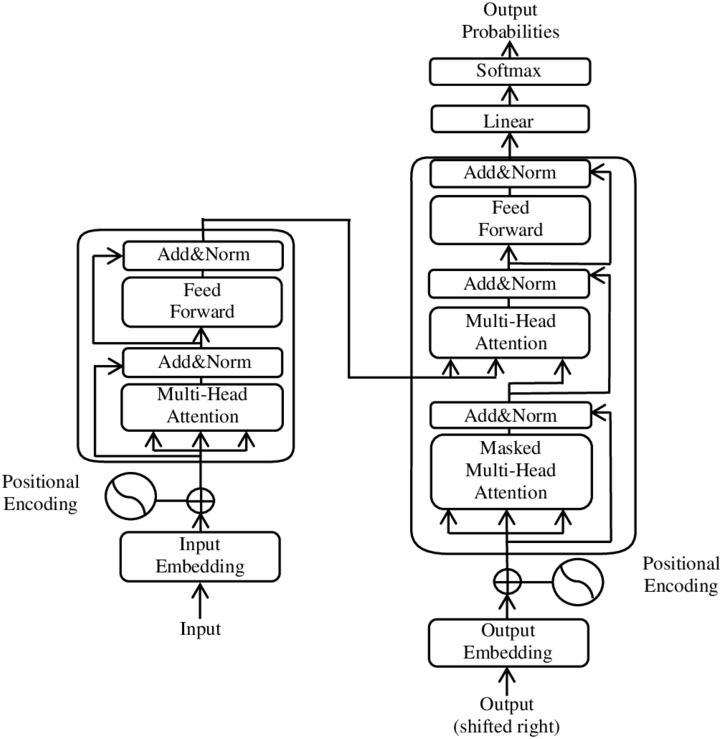
**Figure 2.4.** Encoder-decoder model with attention

Training encoder-decoder model with support attention is still based on likelihood maximization or linear component aforementioned. Attention mechanism mentioned here does not ever concern internal meaning of every token, which only fosters target sequence to pay attention at source sequence. The attention that concerns internal meanings of tokens is called self-attention which is an advancement of attention. In other words, self-attention fosters source sequence to pay attention to itself. Transformer mentioned in the next section will implement self-attention.

# 3. Transformer

*Transformer*, developed by Vaswani et al. (Vaswani, et al., 2017) in the famous paper “Attention Is All You Need”, has also attention mechanism and encoder-decoder mechanism like the aforementioned generation model that applies recurrent neural network (RNN) and short-term memory (LSTM) but transformer does not require to process successively tokens of sequence in token-by-token ordering, which improves translation speed. Moreover, another strong point of transformer is that it has *self-attention* which is the special attention that concerns internal meanings of its own tokens, which is the actually main strong point of transformer. Transformer supports both attention and self-attention, which fosters target sequence to pay attention to both source sequence and target sequence and also fosters source sequence to pay attention to itself. Besides, transformer does not apply RNN / LSTM. Note that word and sentence in natural language processing (NLP) are mentioned as token and sequence, respectively by a convention, so that source sequence ***X*** is fed to encoder and target sequence ***Y*** is fed to decoder where ***X*** and ***Y*** are concerned exactly as matrices.

Each encoder as well as each decoder in transformer are composed of some identical layers. The number of layer which is developed by Vaswani et al. (Vaswani, et al., 2017, p. 3) is 6. Each encoder layer has two sublayers which are *multi-head attention* sublayer and feedforward sublayer whereas each decoder layer has three sublayers which are masked multi-head attention sublayer, multi-head attention sublayer, and feedforward sublayer. Every sublayer is followed by association of *residual mechanism* and *layer normalization*, denoted as Add & Norm = LayerNorm(***X*** + Sublayer(***X***)). The residual mechanism means that sublayer Sublayer(***X***) is added with its input as the sum ***X*** + Sublayer(***X***). Note, Sublayer(***X***) can be attention sublayer or feedforward sublayer. The layer normalization is to normalize such sum. Following figure summarizes transformer developed by Vaswani et al. (Vaswani, et al., 2017, p. 3).



**Figure 3.1.** Architecture of transformer

Feedforward sublayer also called feedforward network (FNN) aims to fine-tune attention by increasing degree of complication.

Encoder and its attention are described firstly when multi-head attention is derived from basic concept of attention. Attention (self-attention) proposed by Vaswani et al. (Vaswani, et al., 2017) is based on three important matrices such as query matrix *Q*, key matrix *K*, and value matrix *V*. The number of rows of these matrices is *m* which is the number of tokens in sequence matrix ***X*** = (***x***1*T*, ***x***2*T*,…, ***x****mT*)*T* but the number of columns of query matrix *Q* and key matrix *K* is *dk* whereas the number of columns of value matrix *V* is *dv*. The number *m* of token is set according to concrete applications, which is often the number of words of the longest sentence. In literature (Vaswani, et al., 2017), *dk* and *dv* are called key dimension and value dimension, respectively. Dimensions of matrices *Q*, *K*, and *V* are *m* x *dk*, *m* x *dk*, and *m* x *dv*, respectively (Vaswani, et al., 2017), (Wikipedia, Transformer (deep learning architecture), 2019).

Where,

Suppose every token vector ***x****i* in sequence matrix ***X*** = (***x***1*T*, ***x***2*T*,…, ***x****mT*)*T* has *dm* elements such that *dm* is called model dimension which is often 512 in NLP.

Query matrix *Q*, key matrix *K*, and value matrix *V* are determined by products of sequence matrix ***X*** and query weight matrix *WQ*, key weight matrix *WK*, value weight matrix *WV*.

|  |  |
| --- | --- |
|  | (3.1) |

Of course, dimensions of weight matrices *WQ*, *WK*, and *WV* are *dm* x *dk*, *dm* x *dk*, and *dm* x *dv*, respectively. All of them have *dm* rows. Matrices *WQ* and *WK* have *dk* columns whereas matrix *WV* have *dv* columns.

Attention is calculated based on scaled product of query matrix *Q*, key matrix *K*, and value matrix *V* in order to make effects on value matrix *V* specifying real sequence by probabilities and moreover, these probabilities are calculated by matching query matrix *Q* specifying query sequence and key matrix *K* specifying key sequence, which is similar to searching mechanism. These probabilities are also based on soft-max function, which implies weights too. Moreover, attention focuses on all tokens of sequence, which improves meaningful context of sentence in NLP. Given matrices *Q*, *K*, and *V*, attention of *Q*, *K*, and *V* is specified as follows:

|  |  |
| --- | --- |
|  | (3.2) |

Note, the superscript “*T*” denotes vector/matrix transposition operator. It is easy to recognize this attention is self-attention of only one sequence ***X*** via *Q*, *K*, and *V* which are essentially calculated from ***X***and weight matrices *WQ*, *WK*, and *WV*. Note, self-attention concerns internal meanings of its own tokens. Transformer here fosters source sequence to pay attention to itself. The reason of dividing product *QKT* by the scaling factor is to improve convergence speed in training transformer. Before explaining how to calculate weight / probability matrix, it is necessary to skim the product *QKT* of query matrix *Q* and key matrix *K* which aims to match query sequence and key sequence.

The dot product ***q****iT****k****j* which indicates how much the query vector ***q****i* matches or attends mutually the key vector ***k****j* is specified as follows:

Probability matrix is specified as follows:

|  |  |
| --- | --- |
|  | (3.3) |

The *i*th row of probability matrix includes weights / probabilities that the *i*th token is associated with all tokens including itself with note that is *m* x *m* matrix, specified by weight/probability vector ***p****i*. It is necessary to explain the *i*th row of probability matrix which is the following vector:

|  |  |
| --- | --- |
|  | (3.4) |

Each probability *pij*, which is weight indeed, is calculated by soft-max function as follows:

|  |  |
| --- | --- |
|  | (3.5) |

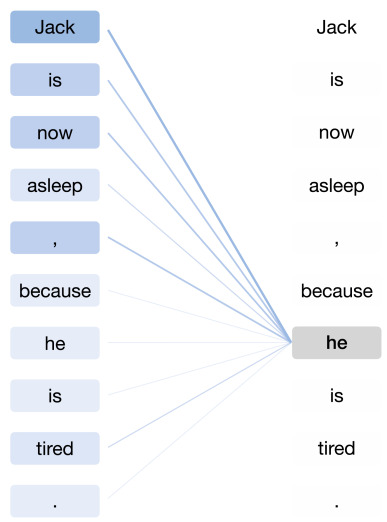
Where exp(.) is natural exponential function. Therefore, probability matrix is totally determined:

Where,

Self-attention of *Q*, *K*, and *V* is totally determined as follows:

Where,

Note, is the *j*th column vector of value matrix *V*. Of course, dimension of self-attention Attention(*Q*, *K*, *V*) is *m* x *dv* having *m* rows and *dv* columns. Attention Attention(*Q*, *K*, *V*) is also called scaled dot product attention because of dot product ***q****iT****k****j* and scaling factor . Each row ***a****i* = (*ai*1, *ai*2,…, )*T* of Attention(*Q*, *K*, *V*), which is a *dv*-length vector, is self-attention of the *i*th token which is contributed by all tokens via scaled dot products *QKT*. Therefore, the preeminence of self-attention is that self-attention concerns all tokens in detail instead of concerning only sequence and the self-attention ***a****i* = (*ai*1, *ai*2,…, )*T* of the *i*th token is attended by all tokens. For example, given sentence “Jack is now asleep, because he is tired.”, the word “he” is strongly implied to the word “Jack” by self-attention of the word “he” although the word “he” is ambiguous. Following figure (Han, et al., 2021, p. 231) illustrates the self-attention of the word “he” in which each strength of implication of another word (accept itself “he”) to the word “he” is indicated by strong degree of connection color.



**Figure 3.2.** Self-attention example

Vaswani et al. (Vaswani, et al., 2017) proposed an improvement of attention called multi-head attention which is concatenation of many attentions. The existence of many attentions aims to discover as much as possible different meanings under attentions and the concatenation mechanism aims to unify different attentions into one self-attention. Following equation specifies multi-head attention with note that the multi-head attention here is self-attention.

|  |  |
| --- | --- |
|  | (3.6) |

Where,

Of course, *WiQ*, *WiK*, and *WiV* are query weight matrix, key weight matrix, and value weight matrix for the *i*th head, respectively whereas *WO* is the entire weight matrix whose dimension is often set as *hdv* x *dm* so that multi-head attention MultiheadAttention(***X***) is *m* x *dm* matrix which is the same to dimension of input sequence matrix ***X*** = (***x***1*T*, ***x***2*T*,…, ***x****mT*)*T*. Note that the concatenation mechanism follows horizontal direction so that the concatenation concatenate(head1, head2,…, head*h*) is *m* x *hdv* matrix when each head head*i* = Attention(*Qi*, *Ki*, *Vi*) is *m* x *dv* matrix. There are *h* heads (attentions) in the equation above. In practice, *h* is set so that *hdv* = *dm* which is model dimension. Recall that *dm* is often 512 in NLP. For easy illustration, the concatenation of *h* attentions is represented as *m* x *hdv* as follows:

Obviously, weight matrix *WO* is *hdv* x *dm* matrix so that multi-head attention MultiheadAttention(***X***) is *m* x *dm* matrix, as follows:

After multi-head attention goes through residual mechanism and layer normalization of attention sublayer, it is fed to feedforward sublayer or feedforward network (FFN) to finish the processing of encoder. Let EncoderAttention(***X***) be output of encoder which is considered as attention:

If there is a stack of *N* encoders, the process above is repeated *N* times. In literature (Vaswani, et al., 2017), *N* is set to be 6. Without loss of generality, we can consider *N* = 1 as simplest case for easy explanations.

Now it is essential to survey how decoder applies encoder attention EncoderAttention(***X***) into its encoding task. Essentially, decoder has two multi-head attentions such as masked multi-head attention and multi-head attention whereas encoder has only one multi-head attention. Their attentions are similar to encoder’s attention but there is a slight difference. Firstly, decoder input sequence ***Y*** = (***y***1*T*, ***y***2*T*,…, ***y****nT*)*T* is fed to masked multi-head attention sublayer with note that ***Y*** is *n* x *dm* matrix with support that model dimension *dm*, which is often set to be 512 in natural language processing (NLP), may not be changed with regard to decoder. Because masked multi-head attention is composed by concatenation of masked head attentions by the same way of encoder, we should concern masked head attention. Sequence ***Y*** should have *n* = *m* tokens like sequence ***X*** in practice. This is necessary because the length *m* = *n* is the largest number of possible tokens in any sequence. For shorter sentences in NLP, redundant tokens are represented by zeros. Moreover, most of parameters (weight matrices) of encoder and decoder are independent from *m* and *n*, especially in the case *m* = *n*.

There is a principle that a token ***y****i* in sequence ***Y*** does not know its successive tokens ***y****i*+1, ***y****i*+2,…, ***y****n* with note that these tokens are called unknown tokens for token ***y****i*, which causes that soft-max function needs to be added a mask matrix *M* whose unknown positions are removed by setting them to be negative infinites because evaluation of negative infinite by exponential function is zero. *Masked attention* is self-attention too.

|  |  |
| --- | --- |
|  | (3.7) |

Where masked matrix *M* is triangle matrix with negative infinites on upper part and zeros on lower part as follows:

Note,

Where *WQ*, *WK*, and *WV* are weight matrices with note that they are different from the ones of encoder. Dimensions of weight matrices *WQ*, *WK*, and *WV* are *dm* x *dk*, *dm* x *dk*, and *dm* x *dv*, respectively. Dimensions of matrices *Q*, *K*, and *V* are now *n* x *dk*, *n* x *dk*, and *n* x *dv*, respectively whereas dimension of masked matrix *M* is *n* x *dm*. We have *QKT* is *n* x *n* matrix:

Recall that the purpose of masked matrix *M* is to remove the affections of current token from its after tokens such that:

Where,

Therefore, masked attention is determined as follows:

Where attention element *aij* is calculated by the aforementioned way:

Dimension of masked attention MaskedAttention(***Y***) is *n* x *dv* having *n* rows and *dv* columns. Following equation specifies masked multi-head attention which is concatenation of some masked attentions.

|  |  |
| --- | --- |
|  | (3.8) |

Where,

Please pay attention that weights matrices *WiQ*, *WiK*, *WiV*, and *WO* are different from the ones of encoder. Dimensions of *WiQ*, *WiK*, *WiV*, and *WO* are *dm* x *dk*, *dm* x *dk*, *dm* x *dv*, and *hdv* x *dm* so that dimension of masked multi-head attention MaskedMultiheadAttention(***Y***) is *n* x *dm*. Residual mechanism and layer normalization are applied into masked multi-head attention too:

Because mechanism of multi-head attention of decoder is relatively special, it is called complex multi-head attention for convention. Because complex multi-head attention is composed by concatenation of some complex attentions by the same way of encoder, we should concern complex attention.

|  |  |
| --- | --- |
|  | (3.9) |

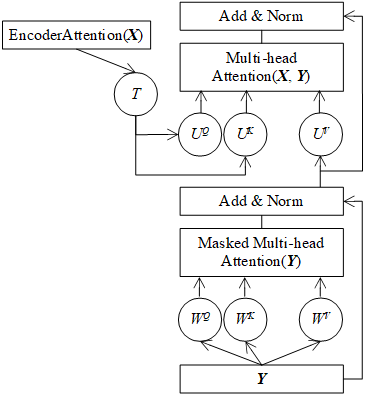
Query matrix *Q* and key matrix *K* of complex attention are products of encoder attention EncoderAttention(***X***) and query weight matrix *UQ* and key weight matrix *UK*, respectively.

|  |  |
| --- | --- |
|  | (3.10) |

Where *T* is *transposition matrix* whose dimension is *n* x *m*. If *n* = *m*, matrix *T* will be removed. Value matrix *V* of complex attention is product of masked multi-head attention and value weight matrix *UV*.

|  |  |
| --- | --- |
|  | (3.11) |

Dimensions of weight matrices *UQ*, *UK*, and *UV* are *dm* x *dk*, *dm* x *dk*, and *dm* x *dv*, respectively. Following figure depicts Attention(***X***, ***Y***) in general view.



**Figure 3.3.** Decoder attention Attention(***X***, ***Y***) in general view

Transformer here fosters target sequence to pay attention to itself and source sequence by masked self-attention and encoder attention. Of course, after complex attention is calculated, multi-head attention of decoder (complex multi-head attention) is totally determined.

|  |  |
| --- | --- |
|  | (3.12) |

Where,

Of course, *UiQ*, *UiK*, and *UiV* are query weight matrix, key weight matrix, and value weight matrix of the *i*th head, respectively whereas *UO* is entire weight matrix and *T* is transposition matrix. Because encoder attention EncoderAttention(***X***) is *m* x *dm* matrix, dimension of transposition matrix *T* is *n* x *m*. If *n* = *m*, matrix *T* will be removed. In practice, it is necessary to set *n* = *m*. Dimensions of *UiQ*, *UiK*, *UiV*, and *UO* are *dm* x *dk*, *dm* x *dk*, *dm* x *dv*, and *hdv* x *dm* so that dimension of multi-head attention MultiheadAttention(***X***, ***Y***) is *n* x *dm*. Residual mechanism and layer normalization are applied into decoder multi-head attention too:

Let ***Z*** be output of decoder which is decoder attention too, we obtain:

Where FFN denotes feedforward network or feedforward sublayer. If there is a stack of *N* decoders, the process above is repeated *N* times. In literature (Vaswani, et al., 2017), *N* is set to be 6. Without loss of generality, we can consider *N* = 1 as simplest case for easy explanations. Note, dimension of ***Z*** is *n* x *dm*. Model dimension *dm* is often set to be 512 in NLP.

In context of statistical translation machine (STM), it is necessary to calculate probabilities of words (tokens) in vocabulary Ω. Because these probabilities are calculated based on soft-max function, it is first to map decoder output matrix ***Z*** into weight vector ***w*** = (*w*1, *w*2,…, *w*|Ω |)*T* where every element *wi* of vector ***w*** is weight of the *i*th word in vocabulary Ω. The mapping is implemented by a feedforward network (FNN) called linear component in literature (Vaswani, et al., 2017, p. 3). In other words, input of linear component is sequence matrix ***Z*** whereas its output is weight vector ***w*** (Alammar, 2018). Please pay attention that the length of ***w*** is the number of words (tokens) in vocabulary Ω and so, ***w*** is also called token/word weight vector.

In practice, ***Z*** is flattened into long vector because ***w*** is vector too so that FNN can be implemented. After token weight vector ***w*** is determined, it is easily converted into output probability vector ***p*** = (*p*1, *p*2,…, *p*|Ω|)*T* where each element *pi* is probability of the *i*th word/token in vocabulary when sentence/sequence ***Z*** is raised (Alammar, 2018). If the *t*th word is issued, its probability *pt* is 1 and other probabilities are 0.

Consequently, the next token which is predicted in STM for example is the one whose probability is highest, which means that the largest element in ***p*** need to be found for STM translation after linear component ***w*** and output probability ***p*** are evaluated given ***Z*** which in turn determined based on source sequence ***X*** and target sequence ***Y*** via mechanism encoder/decoder and attention.

It is not difficult to learn linear component FFN(***Z***) by backpropagation algorithm associated stochastic gradient descent (SGD) method. Concretely, the following cross-entropy *L*(***p*** | Θ) is minimized so as to train FFN(***Z***).

Where Θ is parameter of FFN(***Z***) and the vector ***q*** = (*q*1, *q*2,…, *q*|Ω|)*T* is binary vector from sample whose each element *qi* has binary values {0, 1} indicating whether the *i*th token/word exists. For example, give sequence/sentence (“I”, “am”, “a”, “student”)*T*, if there is only one token/word “I” in sample sentence, the binary vector will be ***q*** = (1, 0, 0, 0)*T*. If three words “I”, “am”, and “student” are mutually existent, the binary vector will be ***q*** = (1, 1, 0, 1)*T*. When SGD is applied into minimizing the cross-entropy, partial gradient of *L*(***p*** | Θ) with regard to *wj* is:

Where,

Proof,

Due to:

We obtain:

So that gradient of *L*(***p*** | Θ) with regard to ***w*** is:

Therefore, parameter Θ is updated according to SGD associated with backpropagation algorithm:

Where *γ* (0 < *γ* ≤ 1) is learning rate.

For STM example, given French source sentence “Je suis étudiant” (Alammar, 2018) is translated into English target sentence “I am a student” (Alammar, 2018) by transformer which is trained with corpus before (transformer was determined), which goes through following rounds:

Round 1:

* French source sentence “Je suis étudiant” coded by sentence/sequence matrix ***X*** = (***x***1 = *c*(“<bos>”), ***x***2 = *c*(“je”), ***x***3 = *c*(“suis”), ***x***4 = *c*(“étudiant”), ***x***5 = *c*(“<eos>”))*T* where *c*(.) is embedding numeric vector of given word with note that words “<bos>” and “<eos>” are special predefined words indicating the beginning of sentence and the end of sentence, respectively. As a convention, *c*(.) is called word/token vector whose dimension can be *dm*=512. If predefined sentence length is longer, redundant word vectors are set to be zeros, for example, let ***x***6 = **0**, ***x***7 = **0**,…, ***x***100 = **0** given the maximum number words in sentence is 100. These zero vectors do not affect decoder evaluation and training parameters.
* Source sequence ***X*** is fed to encoder so as to produce encoder attention EncoderAttention(***X***).

Round 2:

* English target sentence is coded by sequence/matrix ***Y*** = (***y***1 = *c*(“<bos>”))*T*. If predefined sentence length is longer, redundant word vectors are set to be zeros.
* Target sequence ***Y*** = (***y***1 = *c*(“<bos>”))*T* and encoder attention EncoderAttention(***X***) are fed to decoder so as to produce decode output ***Z***.
* Output ***Z*** goes through linear component ***w*** = linear(***Z***) and soft-max function component ***p*** = softmax(***w***) so as to find out the maximum probability *pi* so that the *i*th associated word in vocabulary is “i”. As a result, the embedding numeric vector of the word “i” is added to target sequence so that we obtain ***Y*** = (***y***1 = *c*(“<bos>”), ***y***2 = *c*(“i”))*T*.

Round 3:

* Both target sequence ***Y*** = (***y***1 = *c*(“<bos>”), ***y***2 = *c*(“i”))*T* and encoder attention EncoderAttention(***X***) are fed to decoder so as to produce decode output ***Z***.
* Output ***Z*** goes through linear component ***w*** = linear(***Z***) and soft-max function component ***p*** = softmax(***w***) so as to find out the maximum probability *pi* so that the *i*th associated word in vocabulary is “am”. As a result, the embedding numeric vector of the word “am” is added to target sequence so that we obtain ***Y*** = (***y***1 = *c*(“<bos>”), ***y***2 = *c*(“i”) , ***y***3 = *c*(“am”))*T*.

Similarly, rounds 4, 5, and 6 are processed by the same way so as to obtain final target sequence ***Y*** = (***y***1 = *c*(“<bos>”), ***y***2 = *c*(“i”), ***y***3 = *c*(“am”) , ***y***4 = *c*(“a”), ***y***5 = *c*(“student”) , ***y***6 = *c*(“<eos>”))*T* which is the English sentence “I am a student” translated from the French sentence “Je suis étudiant”. Note, the translation process is stopped when the end-of-sentence word “<eos>” is met.

Main ideas of transformer were described but there are two improvements such as positional encoding and normalization. Firstly, *positional encoding* is that sequences ***X*** and ***Y*** were added by their corresponding position vectors:

|  |  |
| --- | --- |
|  | (3.13) |

Without loss of generality, let POS(***X***) = (pos(***x***1)*T*, pos(***x***2)*T*,…, pos(***x****m*)*T*)*T* be position matrix whose each element is position vector pos(***x****i*) of token ***x****i*. It is necessary to survey pos(***x****i*).

|  |  |
| --- | --- |
|  | (3.14) |

This implies how to calculate position vector POS(***X***) is how to calculate position pos(*xij*) where *i* is position of the *i*th token and *j* is position of the *j*th numeric value of such token vector. We have:

Suppose two successive numeric values such as *j*th numeric value and (*j*+1)th numeric value such that *j* = 2*k* and *j*+1 = 2*k*+1, we need to calculate two kinds of positions as follows:

Fortunately, these positions are easily calculated by sine function and cosine function as follows (Vaswani, et al., 2017, p. 6):

|  |  |
| --- | --- |
|  | (3.15) |

Recall that *dm* is model dimension which is the length of token vector ***x****i*. It is often set to be 512 in natural language processing (NLP). As a result, we have:

Please pay attention that target sequence ***Y*** is added by position vector POS(***Y***) by the same way too. There may be a question that why sequences ***X*** and ***Y*** are added by their position matrices before they are fed into encoder/decoder when tokens in a sequence have their own orders because a sequence is an ordered list of tokens indeed. The answer depends on computational effectiveness as well as flexibility. For example, when sequences are added by their position vectors, transformer can be trained by incomplete French source sequence “<bos> Je suis” and incomplete English target sequence “a student <eos>” because there is no requirement of token ordering. Moreover, sequences can be split into many parts and these parts are trained parallel. This improvement is necessary in case of training huge corpus.

The second improvement is layer (network) normalization:

LayerNorm(***X*** + Sublayer(***X***))

LayerNorm(***Y*** + Sublayer(***Y***))

Because residual mechanism is implemented by the sum ***X*** + Sublayer(***X***) or ***Y*** + Sublayer(***Y***), it is necessary to survey the following normalization without loss of generality:

LayerNorm(***x***)

Where ***x*** = (*x*1, *x*2,…, *xn*)*T* is layer of *n* neuron *xi* with note that each neuron *xi* is represented by a number. Suppose ***x*** as a sample conforms normal distribution, its sample mean and variance are calculated as follows:

As a result, layer normalization is distribution normalization:

|  |  |
| --- | --- |
|  | (3.16) |

In literature, layer normalization aims to improve convergence speed in training.

It is not difficult to train transformer from corpus which can be a huge set of pairs of source/target sequences. Backpropagation algorithm associated with stochastic gradient descent (SGD) is a simple and effective choice. Feedforward sublayer represented by feedforward network (FFN) is easily trained by backpropagation algorithm associated SGD, besides attention sublayers can be trained by backpropagation algorithm associated SGD too. For instance, attention parameters for encoder such as weight matrices *WiQ*, *WiK*, *WiV*, and *WO* can be learned by backpropagation algorithm associated with SGD. Attention parameters for decoder such as weight matrices *WiQ*, *WiK*, *WiV*, *WO*, *T*, *UiQ*, *UiK*, *UiV*, and *UO* can be learned by backpropagation algorithm associated SGD too. Note, starting point for backpropagation algorithm to train transformer is to make comparison of target sequence (for example, the English target sentence “I am a student” given the French source sentence “Je suis étudiant”) and evaluated sequence (for example, the English evaluated sentence “We are scholars” given the same French source sentence “Je suis étudiant”) at decoder, which goes backward encoder. Moreover, please pay attention that zero vectors representing redundant tokens do not affect updating these weight matrices when training transformer.

Recall that training multi-head attention is to learn weight matrices *WiQ*, *WiK*, *WiV*, *WO*, *T*, *UiQ*, *UiK*, *UiV*, and *UO* and thus, without loss of generality, it is necessary to illustrate how to train weight matrices *WQ*, *WK*, and *WV* from data ***X*** = (***x***1*T*, ***x***2*T*,…, ***x****mT*)*T* by stochastic gradient descent (SGD) algorithm associated backpropagation algorithm. Let *A* be computed attention from weight matrices and input data ***X*** and let *A*’ be real output. Of course, *A*’ is also real attention without computation. Note, dimensions of *WQ*, *WK*, and *WV* are *dm* x *dk*, *dm* x *dk*, and *dm* x *dv*, respectively whereas dimension of ***X*** and *A* (*A*’) is *m* x *dm* and *m* x *dv*, respectively.

Where,

Given:

Let,

Such that:

Note, the superscript “*T*” denotes transposition operator of matrix and vector.

*Likelihood* function *L*(*A*) is defined to be maximized to estimate parameters *WiQ*, *WiK*, *WiV*, *WO*, *T*, *UiQ*, *UiK*, *UiV*, and *UO*. Firstly, without loss of generality, weight matrices *WQ*, *WK*, and *WV* are estimated as maximizer of *L*(*A*).

According to stochastic gradient descent (SGD) algorithm, parameters *WQ*, *WK*, and *WV* are computed iteratively as follows:

Note, *γ* (0 < *γ* ≤ 1) is learning rate whereas , , and are first-order derivatives of *L*(*A*) with respect to *WQ*, *WK*, and *WV*, respectively. Note,

Where,

As usual, likelihood *L*(*A*) is defined as the negative of loss function which in turn is squared *Frobenius norm* of the difference between two matrices *A* and *A*’. Therefore, maximizing likelihood function is the same to minimize error function.

|  |  |
| --- | --- |
|  | (3.17) |

Because every soft-max probability vector ***p****i* is mutually independent, the entire likelihood *L*(*A*) is partitioned into *m* *partial likelihood* *li*(***a****i*) which in turn is the negative of squared Euclidean distance between real attention vector ***a****i*’ and computed attention vector ***a****i*.

|  |  |
| --- | --- |
|  | (3.18) |

Such that:

Differential of *li*(***a****i*) with respect to ***q****i* is:

Where ***p****i*’ is derivative of the soft-max function ***p****i*, which is mentioned later.

Due to:

We obtain:

Due to:

Differential of *li*(***a****i*) with respect to *WQ* is:

Therefore, gradient of *li*(***a****i*) with respect to *WQ* is:

This means gradient of *L*(*A*) with respect to *WQ* is:

|  |  |
| --- | --- |
|  | (3.19) |

Due to:

Similarly, gradient of *L*(*A*) with respect to is:

|  |  |
| --- | --- |
|  | (3.20) |

Where ***p****i*’ is derivative of the soft-max function ***p****i*, which is mentioned later.

Recall that estimating *WQ* and *WK* is, in turn, to estimate partial parametric vectors and , which is totally determined as follows:

Consequently, it is necessary to calculate gradient of *L*(*A*) with respect to *WV*. Indeed, differential of *L*(*A*) with respect to *WV* is:

Therefore, gradient of *L*(*A*) with respect to *WV* is:

|  |  |
| --- | --- |
|  | (3.21) |

Such that *WV* is totally estimated as follows:

Where *γ* (0 < *γ* ≤ 1) is learning rate.

The most important thing to estimate parametric matrices *WQ* and *WK* is to calculate the derivative ***p****i*’ of the probability ***p****i* of soft-max function.

Indeed, we have:

|  |  |
| --- | --- |
|  | (3.22) |

Where,

The equation above which is kept intact in case of masked attention is the core of how to calculate derivative of soft-max function. As a result, the three weight matrices *WQ*, *WK*, and *WV* are iteratively estimated by SGD as follows:

Where,

In general, calculating derivative of soft-max function of query matrix and key matrix is the main point of attention estimation.

By extending attention to multi-head attention, the estimation process is modified a little bit. Recall that multi-head attention is concatenation of *h* attentions, as follows:

Where,

The estimation process is now to estimate *h* heads of triples *WiQ*, *WiK*, and *WiV* where every triple of *WiQ*, *WiK*, and *WiV* is query weight matrix, key weight matrix, and value weight matrix for the *i*th head. Moreover, the process is also to estimate the entire weight matrix *WO* whose dimension is often set as *hdv* x *dm*. Let ***A*** be computed multi-head attention from weight matrices and input data ***X*** and let ***A***’ be real output.

Let *Ai* denote the head *headi*, which is an attention, of course.

Such that:

Where,

Of course, each *Ai* is easily extracted from at the *i*th position but vertical cutting at the *i*th position.

Note, dimensions of *WiQ*, *WiK*, *WiV*, and *WO* are *dm* x *dk*, *dm* x *dk*, *dm* x *dv*, and *hdv* x *dm*, respectively whereas dimension of ***X***, , and ***A*** (***A***’) is *m* x *dm*, *m* x *hdv*, and *m* x *dm*, respectively with suppose that *n* = *m*.

Likelihood function *L*(***A***) between computed multi-head attention ***A*** and real multi-headed attention ***A***’ is not essentially changed:

Note,

So that gradient of likelihood function with regard to *WO* is:

Due to:

The entire weight matrix *WO* are iteratively computed by SGD as follows:

|  |  |
| --- | --- |
|  | (3.23) |

Where *γ* (0 < *γ* ≤ 1) is learning rate and,

Let be the *backward error* which is calculated as follows:

|  |  |
| --- | --- |
|  | (3.24) |

Due to:

Note, the superscript “*T*” denotes transposition operator of matrix and vector and dimension of is *m* x *hdv*. The deviation (*Ai*’ – *Ai*) between real attention *Ai*’ and computed attention *Ai* is easily extracted from by the same way to extract *Ai* from , which denoted ***ε****i*.

|  |  |
| --- | --- |
|  | (3.25) |

As a result, every triple of three weight matrices *WiQ*, *WiK*, and *WiV* are iteratively estimated by SGD as usual:

Please pay attention that index [*l*] indicates the *l*th row of a matrix.

Recall that multi-head attention of decoder is called complex attention:

Query matrix *Q* and key matrix *K* of complex attention are products of encoder attention EncoderAttention(***X***) and query weight matrix *UQ* and key weight matrix *UK*, respectively.

Let

We have:

Where *T* is *transposition matrix* whose dimension is *n* x *m*. If *n* = *m*, matrix *T* will be removed. In the most general case, it is necessary to calculate two transposition matrices *T*1 and *T*2 by maximizing likelihood which is the same to minimizing squared error.

Note, dimensions of *T*1 and *T*2 are *n* x *m* and *d* x *dm* (or *dm* x *dm*), respectively whereas dimensions of *UQ*, *Q*, *UK*, *K*, and *E* are *dm* x *dk*, *n* x *dk*, *dm* x *dk*, *n* x *dk*, and *m* x *d* (or *m* x *dm*), respectively. As usual, entire likelihood function *L*(*A*) and *n* partial likelihood functions *li*(***a****i*) are specified according to squared Frobenius norm of computed attention *A* and real attention *A*’, as follows:

Such that:

Matrices *T*1 and *T*2 are iteratively computed with gradient by SGD by maximizing *L*(*A*) as follows:

Where and are gradients of *L*(*A*) with respect to *T*1 and *T*2, respectively. As usual, the SGD is partitioned with *n* partial gradients of *li*(a) as and . Let,

Differential of *li*(***a****i*) with respect to ***q****i* is:

Where ***p****i*’ is derivative of the soft-max function ***p****i*, which is mentioned later.

Due to:

We obtain:

Let *T*1[*i*] specified the vector at the *i*th row of the transposition matrix *T*1 with note that *T*1[*i*] is column vector such that:

Differential of *li*(***a****i*) with respect to *T*1[*i*] is:

Therefore, gradient of *li*(***a****i*) with respect to *T*1[*i*] within context of query matrix is:

Similarly, gradient of *li*(***a****i*) with respect to *T*1[*i*] within context of key matrix is:

Please pay attention that:

As an estimation, gradient of *li*(***a****i*) with respect to *T*1[*i*] is mean of the ones above:

|  |  |
| --- | --- |
|  | (3.26) |

Differential of *li*(***a****i*) with respect to *T*2 is:

Therefore, gradient of *li*(***a****i*) with respect to *T*2 within context of query matrix is:

Similarly, gradient of *li*(***a****i*) with respect to *T*2 within context of key matrix is:

As an estimation, gradient of *li*(***a****i*) with respect to *T*2 is mean of the ones above:

This means gradient of *L*(*A*) with respect to *T*2 is:

|  |  |
| --- | --- |
|  | (3.27) |

Due to:

As a result, the two transposition matrices *T*1 and *T*2 are iteratively estimated by SGD as follows:

Where,

Note, the index [*i*] indicates the *i*th row of attention *A* which is considered as a column vector.

# 4. Pre-trained model

AI models cope with two problems of model learning: 1) it is impossible to preprocess or annotate (label) huge data so as to make the huge data better for training, and 2) huge data is often come with data stream rather than data scratch. Note, the first problem is most important. *Transfer learning* (Han, et al., 2021, pp. 226-227) can solve the two problems by separating the training process by two stages: 1) *pre-training stage* aims to draw valuable knowledge from data stream / data scratch, and 2) *fine-tuning stage* later will take advantages of knowledge from pre-training stage so as to apply the knowledge into solving task-specific problem just by fewer samples or smaller data. As its name hints, transfer learning draws knowledge from pre-training stage and then transfers such knowledge to fine-tuning stage for doing some specific task. Capturing knowledge in pre-training stage is known as source task and doing some specific task is known as target task (Han, et al., 2021, p. 227). Source task and target task may be essentially similar like GPT model and BERT model for token generation mentioned later but these tasks can be different or slightly different. The fine-tuning stage is dependent on concrete application and so, pre-training stage is focused in this section. The purpose of pre-training stage is to build a *large-scale pre-trained model* called *PTM* which must have ability to process huge data or large-scale data. If large-scale data is come from data stream called downstream data, PTM will need to reach the strong point that is parallel computation. If large-scale is too huge, PTM will need to reach the strong point that is efficient computation. When efficient computation can be reached by good implementation, parallel computation requires an improvement of methodology. In order to catch knowledge inside data without human interference with restriction that such knowledge represented by label, annotation, context, meaning, etc. is better than cluster and group, self-supervised learning is often accepted as a good methodology for PTM (Han, et al., 2021, pp. 227-229). Essentially, *self-supervised learning* tries to draw pseudo-supervised information from unannotated/unlabeled data so that such pseudo-supervised information plays the role of supervised information like annotation and label that fine-tuning stage applies into supervised learning tasks for solving specific problem with limited data. The pseudo-supervised information is often relationships and contexts inside data structure. Anyhow, self-supervised learning is often associated with transfer learning because, simply, annotating entirely huge data is impossible. Self-supervised learning associated with pre-training stage is called self-supervised pre-training. Although self-supervised pre-training is preeminent, pre-training stage can apply other learning approaches such as supervised learning and unsupervised learning.

That the essentially strong point of transformer is self-attention makes transformer appropriate to be a good PTM when self-attention follows essentially ideology of self-supervised learning because self-attention mechanism tries to catch contextual meaning of every token inside its sequence. Moreover, transformer supports parallel computation based on its other aspect that transformer does not concern token ordering in sequence. Anyhow, transformer is suitable to PTM for transfer learning and so this section tries to explain large-scaled pre-trained model (PTM) via transformer as an example of PTM. Note, fine-tuning stage of transfer learning will take advantages of PTM for solving task-specific problem; in other words, fine-tuning stage will fine-tune or retrain PTM with downstream data, smaller data, or a smaller group of indications. When fine-tuning stage is not focused in description, PTM is known as transfer learning model which includes two stages such as pre-training stage and fine-tuning stage. In this case, source task and target task of transfer learning have the same model architecture (model backbone) which is the same PTM architecture. Large-scale PTM implies its huge number of parameters as well as huge data from which it is trained.

Generative Pre-trained Transformer (*GPT*), developed in 2018 with GPT-1 by OpenAI (www.openai.com) whose product is ChatGPT launched in 2022, is a PTM that applies only decoder of transformer into sequence generation. In pre-training stage, GPT trains its decoder from huge data over internet and available sources so as to predict next word ***y****t*+1 from previous words ***y***1, ***y***2,…, ***y****t* by maximizing likelihood *P*(***y****t*+1 | Θ, ***y***1, ***y***2,…, ***y****t*) and taking advantages of self-attention mechanism aforementioned (Han, et al., 2021, p. 231). Maximization of likelihood *P*(***y****t*+1 | Θ, ***y***1, ***y***2,…, ***y****t*) belongs to *autoregressive language model*.

|  |  |
| --- | --- |
|  | (4.1) |

Where,

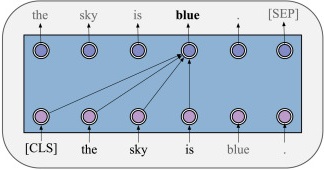
And,

Because GPT has only one decoder, sequence ***X*** is null in GPT.

Likelihood *P*(***y****t*+1 | Θ, ***y***1, ***y***2,…, ***y****t*) is simplified for easy explanation. Exactly, given sequence ***Y*** = (***y***1*T*, ***y***2*T*,…, ***y****n*+1*T*)*T*, GPT aims to maximize log-likelihood *L*(Θ | ***Y***) as follows (Han, et al., 2021, p. 231):

|  |  |
| --- | --- |
|  | (4.2) |

Later on, GPT improves its pre-trained decoder in fine-tuning stage by re-training the decoder with annotated data, high-quality data, and domain-specific data so as to improve pre-trained parameters. Moreover, GPT adds extra presentation layers in fine-tuning stage (Han, et al., 2021, p. 231). Following figure (Han, et al., 2021, p. 232) depicts prediction process of GPT.



**Figure 4.1.** Prediction process of GPT

Bidirectional Encoder Representations from Transformers (*BERT*), developed in 2018 by Google, is a PTM that applies only encoder of transformer into sequence generation. In pre-training stage, BERT trains its encoder from huge data over internet and available sources. Given (*t*+1)-length sequence (***x***1, ***x***2,…, ***x****t*+1)*T*, BERT applies masked language model to randomize an unknown token at random position denoted *masked* where the random index *masked* is randomized in *t*+1 indices {1, 2,…, *t*+1} with note that the randomization process can be repeated many times. Such unknown token, which is called masked token denoted ***y****masked*, will be predicted given *t*-length sequence (***x***1, ***x***2,…, ***x****t*) without loss of generality. In order words, masked words ***x****masked* is predicted from other words ***x***1, ***x***2,…, ***x****t* by maximizing likelihood *P*(***x****masked* | Θ, ***x***1, ***x***2,…, ***x****t*) and taking advantages of self-attention mechanism aforementioned (Han, et al., 2021, p. 232).

|  |  |
| --- | --- |
|  | (4.3) |

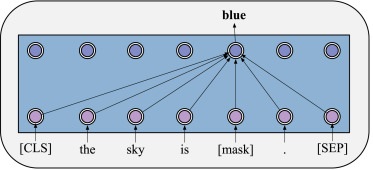
Where,

And,

Likelihood *P*(***y****masked* | Θ, ***x***1, ***x***2,…, ***x****m*) is simplified for easy explanation, thereby it is necessary to explain more how BERT defines and maximizes likelihood with support of masked language model. Given sequence ***X*** = (***x***1, ***x***2,…, ***x****m*), let *R* = {*r*1, *r*2,…, *rk*} be the set of indices whose respective tokens are initially masked, for instance, token will be initially masked if *rj* belongs to mask set *R*. Let be the set of *rj*–1 tokens which are unmasked later, for instance, the tokens , ,…, which were initially masked before are now unmasked (known) at current iteration. Note, the set *R* is called mask set or mask pattern and does not include token . BERT randomizes *k* masked indices where *k* is large enough so as to establish mask set *R*. Let *S* be the set of indices whose tokens are always known, which is the complement of mask set *R* with regard to all indices so that union of *R* and *S* is {1, 2,…, *m*}. Thereby, let ***S*** be the set of tokens whose indices are in *S*. In other words, ***S*** contains tokens which are always known. BERT aims to maximize log-likelihood *L*(Θ | ***X***) as follows (Han, et al., 2021, p. 232):

|  |  |
| --- | --- |
|  | (4.4) |

Later on, BERT improves its pre-trained encoder in fine-tuning stage by re-training the encoder with annotated data, high-quality data, and domain-specific data so as to improve pre-trained parameters. By support of *masked language model* (autoencoding language model) for masking tokens, BERT can predict a token at any position in two directions given a list of other tokens while GPT only predicts a token at next position given previous tokens. The name “BERT”, which is abbreviation of “Bidirectional Encoder Representations from Transformers”, hints that BERT can generate tokens/words in bidirectional way at any positions. Therefore, GPT is appropriate to language generation and BERT is appropriate to language understanding (Han, et al., 2021, p. 231). BERT also adds extra presentation layers in fine-tuning stage (Han, et al., 2021, p. 232). Following figure depicts prediction process of BERT.



**Figure 4.2.** Prediction process of BERT

Recall that given a transfer model, capturing knowledge in pre-training stage is known as source task and doing some specific task is known as target task (Han, et al., 2021, p. 227), thereby there is a question that how source task transfers knowledge to target task or how PTM makes connection between source task and knowledge task. The answer is that there are two transferring approaches such as feature transferring and parameter transferring (Han, et al., 2021, p. 227). *Feature transferring* converts coarse data like unlabeled data into fine data like labeled data so that fine data considered as feature is fed to fine-tuning stage. *Parameter transferring* transfers parameters learned at pre-training stage to fine-tuning stage. If pre-training stage and fine-tuning stage share the same model architecture which is the same PTM architecture, parameter transferring will always occur in PTM. Both GPT and BERT apply parameter transferring because they will initialize or set up their models such as GPT decoder and BERT encoder by billions of parameters that were learned in pre-training stage with the same model architecture (model backbone) such as GPT decoder and BERT encoder before they perform fining-tuning task in fine-tuning stage. Self-supervised learning which trains unlabeled data is appropriate to pre-training stage because unlabeled data is much more popular than labeled data, thereby parameter transferring is often associated with self-supervised learning. Because transformer is suitable to self-supervised learning due to its self-attention mechanism, parameter transferring is suitable to PTMs like GPT and BERT. Moreover, if they apply transformer into annotating or creating task-specific data / fine data for improving their decoder and encoder in fine-tuning stage, they will apply feature transferring too. In general, within parameter transferring and same architecture, PTM itself is backbone for both pre-training stage and fine-tuning stage.

Suppose pre-training stage and fine-tuning stage share the same PTM architecture like GPT and BERT, we focus on parameter transferring in fine-tuning stage now. There are two approaches for parameter transferring: 1) improving parameters which are network weights by specific tasks is called *task-specific improvement* and 2) improving parameters by learning feedbacks from environment is called *feedback improvement*. According to Gemini 2025, some popular techniques of task-specific improvement are text classification, name entity recognition (NER), question answering, machine translation, and text summarization. These techniques, which often belong to supervised learning, often add extra layers to transformer with attention that the extra layers are added to encoder, decoder, or both of them, which depends on PTM architecture like GPT or BERT. Task-specific improvement with the supervised learning techniques is called *supervised learning fine-tuning* and now it is necessary to browse these techniques. In *text classification* technique, an extra layer which is feedforward network (FFN) whose activation function is soft-max function is added to output of transformer. Given a text sequence which is classified into a class is fed to transformer as an input, and then the output of such text sequence via transformer goes through the classification layer so that the FFN predicts class of such text sequence. Of course, the class prediction or classification task is implemented by probability distribution represented by soft-max function, hereby which class has highest probability is predictive class. Following stochastic gradient descent (SGD) algorithm, cross-entropy within context of soft-max function (entropy loss) is minimized by comparing the predictive class and actual class so as to improve parametric weights propagative in backward direction. Training data with text classification is classified or labeled data. Note, the text sequence which is classified is begun with token “<cls>” as usual with note that the term “cls” implies abbreviation of “classification”. In general, the input of text classification technique is text sequence and the output of this technique is predictive class, whose method is entropy loss minimization associated with SGD with support of soft-max function instead of sigmoid function as usual. *Name entity recognition* (NER) is similar to text classification except that the output of NER is label in text form. According to Gemini about NER, sometimes another extra layer is added to the top of FFN so as to discover dependencies between adjacent labels for improving performance. *Question answering* technique tries to discover the answer of a given question, thus, the input of this technique includes a given question and a given contextual paragraph. Consequently, two extra FFN are added to the top (output layer) of transformer: 1) one is responsible for predicting the probability of a token which is the start of the answer in the given contextual paragraph and 2) one is responsible for predicting the probability of a token which is the end of the answer in the given contextual paragraph. Therefore, output of question answering technique is the start and the end of the answer in the given contextual paragraph. Activation functions of the two extra FFN should be soft-max functions because of probabilistic context, whose method is entropy loss minimization associated with SGD. *Machine translation* technique in fine-tuning stage is indeed normal task of traditional transformer having full of encoder and decoder as aforementioned, where the input which is text sequence in source language is fed into encoder and the output which is text sequence in target language is output of decoder indeed. When machine translation technique in fine-tuning stage is applied into GPT (BERT) having only decoder (encoder), an additional encoder (decoder) must be added so that multilingual corpus can be trained. *Text summarization* technique tries to summarize a given text paragraph as a shorter text sequence, thus, the input and the output of this technique are a text paragraph and a text sequence, respectively. It is not necessary to add an extra FFN to the top (output layer) of transformer because the output layer of transformer will produce the summarized text sequence. However, for incomplete transformers like GPT or BERT which has only decoder or encoder, text summarization technique can implement complement encoder / decoder so that the input of text paragraph is fed to encoder and the output of summarized text sequence is output of decoder. In general, supervised learning fine-tuning requires training data to be more specific or more improved so that training data will not be coarse, for example, text classification technique and NER technique require annotated / labeled data. Moreover, backpropagation algorithm is often associated with SGD so as to improve parametric weights in backward direction with note that SGD is a popular optimization algorithm for maximizing likelihood function or minimizing loss function.

Because *feedback improvement* is to learn feedbacks from environment, reinforcement learning is a popular technique of feedback improvement in fine-tuning stage. In practice, *reinforcement learning* (RL) goes after supervised learning fine-tuning when fine-tuning stage is trained two times, for example, recent versions of GPT apply proximal policy optimization (PPO) algorithms into strengthening parameters after parametric weights are improved by supervised learning fine-tuning (classification) with annotated data. Now we should browse reinforcement learning in fine-tuning stage. Different from task-specific improvement which focuses on improving parameters directly, RL aims to improve criteria that cannot be specified by annotated data. In other words, RL aims to align with preferences related to language, human, and human values. General in literature, RL includes two main partners such as environment and agent, in which whenever environment issues a feedback called *state*, agent will response a respective *action*, and then a so-called *reward function* will qualify agent’s action by benefit or penalty so that later actions are better, which means that later actions will response more exactly to environment’ states. The cycle “state → action → reward → state” is continuously iterative process that aims to improve agent’s parameters where agent represents a model with full of parameters and action represents output of such model. In context of fine-tuning stage with RL, agent is the core of the concerned PTM which is its transformer called *original transformer*, of course whereas reward function is coded as an associated transformer called *reward transformer* or *reward model*. Moreover, environment in RL fine-tuning stage can be human, another AI model, and specific task, which issues three kinds of feedback such as human feedback, AI model feedback, and task-specific optimization, which in turn issues three respective RL methods in fine-tuning stages according to Gemini 2025, such as reinforcement learning from human feedback (RLHF), reinforcement learning from AI feedback (RLAIF), and reinforcement learning for task-specific optimization (RLTSO). Because RLHF is the most basic one, it is necessary to study it. According to Gemini 2025, RLHF aims to align PTM’s generated text to user’s preferences such as helpfulness, harmlessness, and truthfulness. Especially, a numeric scalar value called reward score or reward signal which indicates how well the generated text is aligned to users’ preferences is produced by a separated reward model which is often an extra transformer called reward model that is trained on human preference data. As a result, RL fine-tuning process includes three steps as follows: 1) human preference data is collected in which each record of such data includes a response text (output) generated by original transformer (target PTM) associated with a rating or annotation given by user on the response text with note that the input of original transformer is a prompt from dialogue interaction, 2) given a pair of prompt as input and response as output of original transformer, a separated reward transformer is trained on human preference data so that given a so-called *preference pair* of prompt as input and response as output of original transformer, such preference pair is fed to reward transformer as input of reward transformer, and then output of reward transformer is the reward score which indicates that how much the prompt and the response inside the preference pair are matched together, and 3) some RL algorithm like proximal policy optimization (PPO) is applied into fine-tuning stage by improving parameters of original transformer by association of original transformer and reward transformer via iterative reward process.

Note, reward signal or reward score produced by reward transformer is also based on some purposive criteria while state and action now are represented by prompt and response, respectively whereas environment and agent are human and original transformer, respectively. Moreover, reward signal indicates that reward score acts as the signal that guides RL algorithm updates original transformer’s parameters to be better so that original transformer will produces responses that achieve higher reward score in future. The third step called RL fine-tuning step is the most important step of RL fine-tuning process includes four sub-steps as follows: 1) original transformer generates a response from a prompt, 2) reward transformer produces a reward score for a pair of such prompt and such response, 3) some RL algorithm like PPO maximizes expected reward from many reward scores in order to update original transformer’s parameters to be better and of course, the maximizing and updating process is followed by a iterative process of continuous adjusting a so-called policy so as to produce good reward score as high as possible with note that policy is the mapping from prompt (state) to response (action), and 4) reward signals or reward scores guide original transformer to generate better response text which is aligned much to human preferences with note that human preference or human feedback is represented by users’ rating and reward transformer is learned from human preference data.

RLAIF is similar to RLHF except that human feedback is replaced by feedback from an AI model, which means that a *feedback AI model* plays the role of evaluator for original transformer. Human preference data is replaced by preference data in RLAIF, in which each preference pair inside preference data is consisted of response from original transformer and feedback as rating issued from feedback AI model. Moreover, reward transformer is replaced by feedback AI model in RLAIF. In other words, RLAIF reward model is represented by a feedback AI model and thus, reward score or reward score is produced by feedback AI model with preference pair as input. RL fine-tuning step in RLAIF is kept intact but please pay attention that feedback AI model plays now two roles: 1) it is evaluator for original transformer as human so that it issues feedbacks as preferences and 2) it is reward model that produces reward score for preference pair. Note, feedback AI model issues preferences (feedbacks) but it is also trained on preference data and it produces reward scores based on preference pair too. RLTSO is similar to RLAIF except that feedback is replaced by outcome of a specific task, which means that feedback AI model is replaced by specific task. Preference pair inside preference data is now consisted of response from original transformer and outcome from specific task. RL fine-tuning step in RLTSO is kept intact but please pay attention that the specific task plays now two roles: 1) it is evaluator for original transformer as human so that it issues outcomes as preferences and 2) the environment where the specific task performs plays the role of reward model that produces reward score for preference pair. Some popular specific tasks were mentioned in the section about task-specific improvement in fine-tuning stage, which can be text classification, name entity recognition (NER), question answering, machine translation, and text summarization. Besides, there are some other specific tasks such as teaching tool, generating programming code / mathematical code, solving problem tool, and improving step-by-step reasoning tool, according to Gemini 2025. Reward score of RLTSO is often metric which is automatically evaluated on outcome of a specific task, which often indicates correctness, relevance, and accuracy of outcome.

Recall that the third step called RL fine-tuning step is the most important step of RL fine-tuning process and moreover, RL algorithm like PPO is the core of such third step and so, PPO developed by Schulman et al. at OpenAI in 2017 is described in detailed now according to guidance of Gemini (gemini.google.com). Exactly, the fourth sub-step of RL fine-tuning step is described in detailed. Recall that RL has two main approaches such as model-based approach and model-free approach where model-free approach has two main methods such as value-based method and policy-based method, according to Gemini 2025. Policy gradient method is the most popular policy-based method to which PPO belongs. Given *original transformer* known as target PTM is specified by *policy* *π*Φ(*a* | *s*) which is probability of next token in generated text *a* given prompt *s* with note that *s* and *a* are input and output of original transformer, respectively when contexts of *s* and *a* are state of environment and action of agent in RL model. Indeed, policy *π*Θ(*a* | *s*) is the probability of action *a* corresponding to given state *s* so that action *a* is optimized and so Θ is parametric weights of original transformer and *π*Θ(*a* | *s*) itself, which is a feedforward network (FFN) associated with soft-max function, is added to the top of original transformer. As a convention Θ is called *policy parameter* or *original parameter*. Reward function *R*Ψ(*s*, *a*) is represented by *reward transformer* where the preference pair (*s*, *a*) is input of reward transformer and *R*Ψ(*s*, *a*) itself which is a real number indicating reward score or *immediate reward* is output of reward transformer. Note, Ψ is parametric weights of reward transformer which is trained on preference data and so Ψ is called *preference parameter* or reward parameter. Because PPO is iterative process in which states and actions go through many iterations, they are indexed by time point *t* as *st* and *at*. Therefore, we specified policy (original transformer) and reward function (reward transformer) with regard to time point *t* as follows:

|  |  |
| --- | --- |
|  | (4.5) |

The reward score or immediate reward over all actions is expected value of all *R*Ψ(*st*, *at*) over all actions:

|  |  |
| --- | --- |
|  | (4.6) |

The expected value now known as immediate reward as usual is calculated in more practical way as follows:

Where *K* is the number of all possible actions at current state *st*. In context of transformer where training data is split into many epoch, suppose each epoch has *T* batches corresponding to *T* time points so that such there are *T* states *st* (s) in the epoch and suppose the *t*th batch has *K* paired records of state *sk* and action *ak*, then following equation is the most practical method to calculate the immediate reward *R*Ψ(*st*) among *T* immediate rewards in the epoch:

|  |  |
| --- | --- |
|  | (4.7) |

When reward transformer is trained separately, it is possible to denote *rt*+1 as immediate reward from state *st* to next state *st*+1:

|  |  |
| --- | --- |
|  | (4.8) |

Let *V*Φ(*st*) be the *value function* which is expectation (average value) of all future reward scores from current state *st* to infinity over all actions. Value function *V*Φ(*st*) is defined based on reward function but its parameter Φ is different from reward weights Ψ and moreover *V*Φ(*st*) has no mathematical formulation and so, *V*Φ(*st*) can be modeled as a FFN so that it is evaluated as output of such FFN and its input is current state *st*. Therefore, it is essential to describe how to train *V*Φ(*st*), which is how to estimate parametric weights Φ. Indeed, Φ is learned by minimizing mean square error between value *V*Φ(*st*) and its target value where *target value* is the true value which is combination of immediate reward *rt*+1 at current state and the estimated value of next state. Target value *Rt*(1) at state *st* over one next state is specified as follows:

Where *α* is discount factor in interval (0, 1] implies how much the next reward contributes is valued (important) when compared to immediate reward *R*Ψ(*st*). Please pay attention that Φ*old*, which is known, is the old version of parameter Φ and so target value is always determined. Target value *Rt*(*n*) at state *st* over *n* next states, which is called the *n-step return*, is specified as follows:

|  |  |
| --- | --- |
|  | (4.9) |

Mean square error between the *n*-step return and value function *V*Φ(*st*), which is function of Φ, is specified as follows:

|  |  |
| --- | --- |
|  | (4.10) |

So that the estimate Φ\* is minimizer of MSE(Φ).

This mean square error MSE(Φ) is a popular representation of value function loss. By applying stochastic gradient descent (SGD) algorithm, it is easy to estimate Φ over some steps as follows:

|  |  |
| --- | --- |
|  | (4.11) |

Where *γ* is learning rate in interval (0, 1] and ∇MSE(Φ) is gradient of MSE(Φ) with regard to Φ. Please pay attention that when immediate reward *R*Ψ(*st*) is expected reward at current state, value function *V*Φ(*st*) is future reward / *expected reward* at current state and so, *V*Φ(*st*) is also called future reward model / *expected reward model* and Φ is called future reward parameter / *expected reward parameter*.

After value function *V*Φ(*st*) is determined, let *δt* be *temporal difference* (TD) error between value function *V*Φ(*st*) at current time point and value function *V*Φ(*st*+1) at next time point:

|  |  |
| --- | --- |
|  | (4.12) |

*Advantage function* *At* qualifies how much better each action which is taken at current state *st* will produce good future reward aligned with the expected reward *V*Φ(*st*). Note, *At* can be considered as expected reward like *V*Φ(*st*). Parameter of advantage function *At* is Φ, which means that *At* is defined based on future reward model *V*Φ(*st*). Equation of *At* is defined based on TD error as follows:

|  |  |
| --- | --- |
|  | (4.13) |

Where *α* is discount factor and *λ* is the parameter that controls bias-variance trade-off of the advantage estimate. The length of the equation of *At* is arbitrary so that, the longer the length is, the more accurate the advantage function is. Please pay attention that advantage function *At* is the cornerstone of PPO because it will be used later to train original transformer (policy). In practice, *At* is approximated by its estimate over *n* times points as follows:

|  |  |
| --- | --- |
|  | (4.14) |

Let *r*Θ(*st*, *at*) be the ratio of the probability of taking action under current policy *π*Θ(*at* | *st*) to the probability of taking action under old policy . Of course, *r*Θ(*st*, *at*) indicates how much the policy is changed from old one to current one, which is called *probability ratio* specified as follows:

|  |  |
| --- | --- |
|  | (4.15) |

Where Φ*old*, which is known, is the old version of parameter Φ of original transformer with note that original transformer itself is policy *π*Θ(*at* | *st*). In practice, probability ratio *r*Θ(*st*, *at*) is taken as an expected value *r*Θ(*st*) over all actions *ak* as follows:

Where *K* is the number of all possible actions at current state *st*. For easy computation, the expected probability ratio *r*Θ(*st*) is known as probability ratio as usual. In context of transformer where training data is split into many epoch, suppose each epoch has *T* batches corresponding to *T* time points so that such there are *T* states *st* (s) in the epoch and suppose the *t*th batch has *K* paired records of state *sk* and action *ak*, then following equation is the most practical method to calculate the probability ratio *r*Θ(*st*) among *T* probability ratios in the epoch:

|  |  |
| --- | --- |
|  | (4.16) |

Because probability ratio *r*Θ(*st*) is important to limit the size of policy updates according to Gemini, a so-called *clipping parameter* *ε* is proposed to clipthe probability ratio *r*Θ(*st*). Exactly, clipping parameter *ε* is a small positive integer between 0.1 and 0.3, which is used to define the range [1 – *ε*, 1 + *ε*] in which *r*Θ(*st*) is clipped. Let be *clipped probability ratio*, as follows:

|  |  |
| --- | --- |
|  | (4.17) |

Recall that advantage function *At* expresses how much better each action which is taken at current state *st* will produce good future reward aligned with the expected future reward *V*Φ(*st*). Therefore, the product of probability ratio and advantage function is ideal to be used to estimate policy parameter Θ of original transformer (policy). Exactly, such product is formally formulated in expectation called *surrogate objective function* *LCLIP*(Θ) as function of Θ, as follows:

|  |  |
| --- | --- |
|  | (4.18) |

Suppose there are *T* time points in PPO, surrogate objective function *LCLIP*(Θ) is simplified as follows:

|  |  |
| --- | --- |
|  | (4.19) |

The superscript “CLIP” indicates that surrogate objective function *LCLIP*(Θ) is clipped because of clipped probability ratio . As a result, surrogate objective function *LCLIP*(Θ) is maximized to estimate parametric weights Θ, which is the ultimate purpose of PPO. Let Θ\* be the estimate of Θ which is maximizer of *LCLIP*(Θ).

By applying stochastic gradient descent (SGD) algorithm, it is easy to estimate Θ over some steps as follows:

|  |  |
| --- | --- |
|  | (4.20) |

Where *γ* is learning rate in interval (0, 1] and ∇*LCLIP*(Θ) is gradient of *LCLIP*(Θ) with regard to Θ. Let *d* denote index of epoch where epoch is a part of training data, pseudo-code of PPO which trains reward transformer *R*Ψ(*s*, *a*), future reward model (value function) *V*Φ(*s*), original transformer (policy) *π*Θ(*a* | *s*) and is described as follows:

The old policy parameter Θ*old* is initialized by current one as Θ*old* = Θ.

Repeat whenever an epoch is fed:

The epoch is a part of training data including prompt and actual response so that original transformer (policy) *π*Θ(*a* | *s*) generates new responses. Later on, users give feedbacks to the new responses so as to collect partial preference data on which reward transformer *R*Ψ(*s*, *a*) is trained so as to estimate preference parameter Ψ with note that prompt, response, and user rating are coded as state, action, and preference, respectively.

Suppose the epoch has *T* batches so as to calculate *T* important quantities such as immediate reward function *R*Ψ(*st*), future reward (value function) *V*Φ(*st*), *n*-step return *Rt*(*n*), advantage function *At*, and probability ratio *r*Θ(*st*) as expectations.

Future reward parameter Φ is estimated by minimizing mean square error of future reward (value function) *V*Φ(*st*) according to SGD as follows:

Surrogate object function *LCLIP*(Θ) is specified and policy parameter Θ is estimated as maximizer of *LCLIP*(Θ) by SGD with this current epoch *d* or after *D* epochs (delaying estimation) as follows:

The old policy parameter Θ*old* is re-assigned by current one as follows:

Until some terminated conditions are met.

In general, PPO estimates preference parameter Ψ, expected reward parameter Φ, and policy parameter Θ. The terminated conditions are often the number of iterations or insignificant change in these estimates. An important task in PPO is to calculate the gradient ∇*LCLIP*(Θ) of surrogate objective function *LCLIP*(Θ) with regard to Θ. Such gradient is called surrogate objective gradient. The function *LCLIP*(Θ) is re-written as follows:

Where *f*(Θ) is called partial surrogate objective function:

So that

|  |  |
| --- | --- |
|  | (4.21) |

Where ∇*f*(Θ) is called partial surrogate objective gradient which is necessary to be determined.

|  |  |
| --- | --- |
|  | (4.22) |

Where *r*’Θ(*st*) is the first-order derivative of probability ratio *r*Θ(*st*) with regard to Θ:

It is easy to calculate the first-order derivative *π*’Θ(*at* | *st*) of the policy *π*Θ(*at* | *st*) with regard to Θ where the action *at* is output of policy transformer (original transformer) *π*Θ(*at* | *st*).

# 5. Large language model

Artificial intelligence (AI) raises two main trends related to biological evolution such as reproductive capacity represented by generative AI (GenAI) and self-thinking capacity started with artificial general intelligence (AGI), where GenAI is being developed strongly. GenAI generates any kind of data but if it focuses on linguistics, GenAI will produce human-understanding text which is result of the collaboration of GenAI and statistical translation machine (STM) when STM is more preeminent than grammar-based translation machine. The collaboration of GenAI and STM which is called conventionally large language model (LLM) aims to generate human-understanding text, which is firstly applied into human-machine conservational applications like chat-box although GenAI potential is extended much more than its previous applications. Note, the term “language model” in LLM indicates the incorporation of GenAI and STM for generating human-understanding text like linguistic translation applications do and the term “large” in LLM indicates that LLM is trained on large data (large corpus). LLM applies recurrent neural network (RNN) and long short-term memory (LSTM) into generating text at the first developing stage but later on, because transformer which is invented with self-attention is more preeminent than RNN and LSTM, transformer becomes fundamental of LLM which means that LLM is built upon transformer but the development of LLM is not stopped yet with just only transformer although transformer is still the essential fundamental of LLM now. Transfer learning is concerned recently (2016) again in order to solve the problem of huge data and coarse data in machine learning, which includes two stages such as pre-training stage and fine-tuning stage. As an excellent result from transfer learning, a model of transfer learning produces which is trained with arbitrary, huge, and coarse data will produces the beginning result known as the template which in turn is improved into the best result in fine-tuning stage by many ways such as supervised learning with standard / finer data and optimization with reinforcement learning. Because pre-training stage in transfer learning is very important, it requires self-supervised learning method due to coarse training data. Fortunately, transformer implements very well its essential self-attention mechanism which is actually a self-supervised learning method, and thus, transformer becomes ideal model for pre-training stage of transfer learning with note that parameters of transformer which are also neural network weights are easy to be improved consequently in fine-tuning stage. Because transformer is essential for both LLM and transfer learning, LLM is developed naturally as a model of transfer learning, which includes two stages such as pre-training stage and fine-tuning stage, which implies that the incorporation of LLM and transfer learning based on transformer where transformer is still the core of LLM is the second developing step of LLM. Note, if pre-training stage is concerned, LLM can be considered as a pre-trained model. Fine-tuning stage of LLM is often applied supervised learning methods with standard / finer data but LLMs are developed recently (2024) is being applied reinforcement learning methods into fine-tuning stage so as to make the final result (generated text) better. The third developing step of LLM is to extend the generating text capacity to the generating multimodal data capacity where multimodal data includes text, image, sound and video. The third generation of LLM, which can generate multimodal data such as text, image, sound and video, is called multimodal LLM. Recent multimodal LLMs like Gemini 2025 can generate extensively mathematical formulas and programming codes, even though they can have reasoning capacity which produces step-by-step reasoning explanations. However, the reasoning capacity is achieved from training the very large data including a lot of knowledge about reasoning, inferences, and mathematics. Therefore, I think that the current reasoning capacity (2024) is not self-thinking mechanism (self-consciousness mechanism) and so, maybe it is necessary to research the integration of fuzzy logic and reinforcement learning to train LLM to understand logic reasoning like LLM was trained to currently “speak” (generate) linguistic text with note that human firstly studies language and then studies logic later.

Although modern *LLM until 2025 which is based on transformer is a pre-trained model* indeed, LLM has its own developing history starting from domain of natural language processing (NLP). LLM is the language model that processes large linguistic dataset known as large corpus and LM starts from NLP. By formal definition, language model which is a foundational component of NLP is designed to understand and predict human language patterns (Gemini 2025). Actually, there are two approaches for language model such as grammar analysis approach and statistical approach but statistical approach becomes preeminent day after day so that it is possible to think that language mode is developed recently from statistical approach. Shortly, language model is developed from *statistical language model* (SLM) in pre-2010s to *neural language model* (NLM) in 2010s to transformer-based LLM until 2025. It is helpful to skim some original language models with note that recurrent neural network and long short-term memory aforementioned are applied as typical NLMs although other applications of recurrent neural network and long short-term memory are more diversified. Please pay attention that some SLMs an NLMs can be considered as LLMs if they process large corpus. Given sequence ***X*** = (***x***1*T*, ***x***2*T*,…, ***x****mT*)*T* where every ***x****i* called token vector represents the *i*th token in sequence ***X*** with note that sequence is sentence and token represents word in context of natural language processing, SLM specified the joint probability of sequence ***X*** as follows:

Where *P*(***x****t* | ***x***1, ***x***2,…, ***x****t*–1) is conditional probability of current token given all its previous tokens. Note,

Let

Where the notation ***x***<*t* denotes the sub-sequence of all its previous tokens ***x***1, ***x***2,…, ***x****t*–1.

Such that:

The joint probability *P*(***X***) can be considered as SLM likelihood but the logarithm of *P*(***X***) is known as *SLM likelihood* as usual:

Suppose there is a language corpus including *N* sentences ***X****i*, then SLM likelihood is extended easily as follows:

As a convention, the sequence ***X*** represents the training corpus .

Most of SLMs aims to maximize its likelihood *l*(***X***) for estimating itself and so, please pay attention to the likelihood *l*(***X***). If SLM has parameter Θ, estimation of SLM is to maximize the likelihood *l*(Θ | ***X***) where *l*(Θ | ***X***) is function Θ given data ***X*** so as to compute the parameter Θ with note that *P*(***x****t* | ***x***<*t*) can be parameter itself or can be dependent partially on Θ.

If *P*(***x****t* | ***x***<*t*) includes entirely or partially the parameter Θ as *P*(***x****t* | Θ, ***x***<*t*), we can denote:

|  |  |
| --- | --- |
|  | (5.1) |

The ultimate purpose of SLM is to predict the next token ***x****t* given sub-sequence ***x***1, ***x***2,…, ***x****t*–1 after the parameter Θ of SLM is estimated as Θ\* by maximizing the likelihood *l*(***X***).

Therefore, the conditional *P*(***x****t* | ***x***<*t*) of the current token ***x****t* given all its previous tokens ***x***1, ***x***2,…, ***x****t*–1 is called *predictive probability*. In some case, the predictive probability *P*(***x****t* | ***x***<*t*) is considered as SLM likelihood so as to be maximized for estimating SLM if *P*(***x****t* | ***x***<*t*) includes entirely or partially the parameter Θ as *P*(***x****t* | Θ, ***x***<*t*), especially in the case of illustration or training small corpus.

Or,

Because the predictive probability *P*(***x****t* | ***x***<*t*) depends on previous tokens of given token, it is a regressive function, indeed and so the likelihood *l*(***X***) or the predictive probability *P*(***x****t* | ***x***<*t*) represents the so-called *autoregressive language model* (ALM). Therefore, the common SLM is ALM and so, ALM is a representation of SLM. Note, the predictive probability *P*(***x****t* | ***x***<*t*) can be called *autoregressive probability* or autoregressive function whereas the likelihood *l*(Θ | ***X***) can be called *autoregressive likelihood*.

An popular SLM is *n-gram model* in which every token is called *gram* (word) and the parameter of *n*-gram model is the set of predictive probabilities *P*(***x****t* | ***x***<*t*).

|  |  |
| --- | --- |
|  | (5.2) |

The *n*-gram model does not maximize its likelihood *l*(Θ | ***X***) because its parameter *θt* = *P*(***x****t* | ***x***<*t*) are easily calculated based on Markov assumption. Given the factor *n* in *n*-gram model, the *n*-degree Markov assumption states that the predictive probability of current token ***x****t* is only dependent on *n*–1 its previous tokens ***x****t*–1, ***x****t*–2,…, ***x****t*–(*n*–1).

|  |  |
| --- | --- |
|  | (5.3) |

Please pay attention that the notation *c*(*…*) denotes the number of occurrences of the sub-sequence “…” in the training corpus ***X*** (). Therefore, the 1-gram model specifies their parameters as follows:

Where *c*(***x****t*) is the number of occurrences of the word ***x****t* in training corpus. For example, given sentence “*transformer is the preeminent large language model until year 2025*”, then *θ*1 = *P*(“*transformer*”) = *c*(“*transformer*”) / 10 = 1/10.

The 2-gram model specifies their parameters as follows:

Where *c*(***x****t*–1, ***x****t*) is the number of occurrences of the sequence (***x****t*–1, ***x****t*) in training corpus. For example, given sentence “*transformer is the preeminent large language model until year 2025*”, then *θ*7 = *P*(“*model*” | “*language*”) = *c*(“*language model*”) / *c*(“*language*”) = 1/1.

The 3-gram model specifies their parameters as follows:

Where *c*(***x****t*–2, ***x****t*–1, ***x****t*) is the number of occurrences of the sequence (***x****t*–2, ***x****t*–1, ***x****t*) in training corpus. For example, given sentence “*transformer is the preeminent large language model until year 2025*”, then *θ*7 = *P*(“*model*” | “*language*”, “*large*”) = *c*(“*large* *language model*”) / *c*(“*language model*”) = 1/1. In practice, 2-gram model and 3-gram model are most popular.

Neural language model (NLM) is later version of statistical language model (SLM), which is also a kind of SLM, indeed. An implementation of NLM, which is more effective than *n*-gram model, is recurrent neural network (RNN) and long short-term memory (LSTM) which are aforementioned. The methodology of LSTM is the same to LSTM, which follows autoregressive language model (ALM) with note that ALM is a common SLM. In other words, RNN or LSTM is the union of ALM (SLM) and NLM where ALM is implemented by artificial neural network. It is necessary to recall the role of RNN as an NLM when we knew that RNN and LSTM are sequence generation models, which implies that sequence generation model does not go beyond language model. When applied into large corpus, RNN and LSTM are large language models, indeed. Given source sequence ***X*** = (***x***1*T*, ***x***2*T*,…, ***x****mT*)*T* and target sequence ***Y*** = (***y***1*T*, ***y***2*T*,…, ***y****nT*)*T*, autoregressive language model with RNN has two RNNs: RNN called *encoder* for ***X*** generation and the other one called *decoder* for ***Y*** generation. The predictive probability also called autoregressive probability is extended as follows:

Note, training large corpus applies autoregressive likelihood where autoregressive likelihood is sum of logarithms of autoregressive probabilities.

|  |  |
| --- | --- |
|  | (5.4) |

As usual, the autoregressive probability can be approximated by standard normal distribution, which is equivalent to square error function, as follows:

Where *f*(***X***, ***y***1, ***y***2,…, ***y****t*–1 | Θ) denotes encoder-decoder chain.

Please remember parameters of RNN where ***Wh***, ***Uh***, and ***Wo*** are weight matrices of current hidden neuron ***h****t*, previous hidden neuron ***h****t*–1, and current output neuron ***o****t*, respectively whereas ***bh*** and ***bo*** are bias vectors of ***h****t* and ***o****t*, respectively. Moreover, *σ****h***(.) and *σ****o***(.) are activation functions of ***h****t* and ***o****t*, respectively.

Therefore, training encoder-decoder model begins with maximum likelihood estimation (MLE) associated with backpropagation algorithm and SGD from decoder back to encoder.

Which implies:

By advancement, in STM with predefined word vocabulary, a simple but effective way to train encoder-decoder model is to replace *P*(***y****t* | Θ, ***X***, ***y***<*t*) by a so-called linear component which is a feedforward network (FFN) with support of soft-max function and cross-entropy loss minimization aforementioned. Recall that FNN maps the *t*th target token specified by token vector ***y****t* to a weight vector ***w*** whose each element *wi* (0 ≤ *wi* ≤ 1) is weight of *i*th token (Alammar, 2018).

Length of weight vector ***w*** is the cardinality |Ω| where Ω is the vocabulary containing all tokens. After token weight vector ***w*** is determined, it is easily converted into output probability vector ***p*** = (*p*1, *p*2,…, *p*|Ω|)*T* where each element *pi* is probability of the *i*th token in vocabulary given the *t*th target token (Alammar, 2018).

|  |  |
| --- | --- |
|  | (5.5) |

Consequently, training encoder-decoder model begins with training linear component FFN(***y****t*) back to training decoder back to training encoder, which follows backpropagation algorithm associated stochastic gradient descent (SGD) method. Concretely, the following cross-entropy loss loss(***p*** | Θ) is minimized so as to train FFN(***y****t*).

Such that:

Where the vector ***q*** = (*q*1, *q*2,…, *q*|Ω|)*T* is binary vector from sample whose each element *qi* has binary values {0, 1} indicating whether the *i*th token/word exists. The gradient of loss(***p*** | Θ) with regard to ***w*** is:

Where,

Therefore, parameter Θ is updated according to SGD associated with backpropagation algorithm:

Where *γ* (0 < *γ* ≤ 1) is learning rate. Anyhow, the essence of FFN and cross-entropy minimization still follows the methodology of maximizing autoregressive likelihood.

Transformer is an advanced version of generation model where the ordering of tokens in the same sequence is not important because of the mechanism of positional encoding so that transformer is the best LLM until 2025. The autoregressive probability known as SLM likelihood is conserved in transformer about the SLM methodology. Moreover, the training transformer is improved by association of autoregressive likelihood and the linear feedforward network (FFN) which is aforementioned and so it is not necessary to repeat how to train FFN by minimizing cross-entropy loss. Recall that the essence of FFN and cross-entropy minimization still follows the methodology of maximizing the autoregressive likelihood *l*(Θ | ***Y***).

Where Θ includes important parameters of transformer such as query weight matrix, key weight matrix, and value weight matrix. Transformer is the most successful union of ALM (SLM) and NLM until 2025 where ALM is implemented by transformer. GPT which is the successful transformer-based LLM takes advantages of autoregressive language model (ALM).

Alternately, transformer can be trained by a so-called masked language model (MLM) in advance. Given sequence ***Y*** = (***y***1, ***y***2,…, ***y****n*), let *R* = {*r*1, *r*2,…, *rk*} be the set of indices whose respective tokens are initially masked, for instance, token will be initially masked if *rj* belongs to mask set *R*. Recall that ***Y*** is output of transformer decoder. Let be the set of *rj*–1 tokens which are unmasked later, for instance, the tokens , ,…, which were initially masked before are now unmasked (known) at current iteration. Note, the set *R* is called mask set or mask pattern and does not include token . MLM randomizes *k* masked indices where *k* is large enough so as to establish mask set *R*. Let *S* be the set of indices whose tokens ***y****i* are always known, which is the complement of mask set *R* with regard to all indices so that union of *R* and *S* is {1, 2,…, *n*}.

Thereby, let ***S*** be the set of tokens ***y****i* whose indices are in *S*. In other words, ***S*** contains tokens which are always known.

MLM aims to maximize log-likelihood *l*(Θ | ***Y***) called *masked likelihood* as follows (Han, et al., 2021, p. 232):

|  |  |
| --- | --- |
|  | (5.6) |

The set ***S*** can be empty so that the parametric estimation is independent between different iterations, such that:

Note, in this situation,

Where the notation “\” denotes complement operator in set theory.

The training transformer within context of MLM is improved by association of the masked likelihood and the linear feedforward network (FFN) which is aforementioned and so it is not necessary to repeat how to train FFN by minimizing cross-entropy loss, in detailed. Shortly, we have following equations to minimize cross-entropy loss associated with MLM.

|  |  |
| --- | --- |
|  | (5.7) |

Such that:

BERT which is the successful transformer-based LLM takes advantages of masked language model (MLM).

# 6. Conclusions

As the paper title “Attention is all you need” (Vaswani, et al., 2017) hints, attention-awarded transformer is the important framework for generative artificial intelligence and statistical translation machine whose applications are not only large but also highly potential. For instance, it is possible for transformer to generate media content like sound, image, video from texts, which is very potential for cartoon industry and movie making applications (film industry). The problem of difference in source data and target data, which can be that, for example, source sequence is text sentence and target sequence is raster data like sound and image, can be solved effectively and smoothly because of two aforementioned strong points of transformer such as self-attention and not concerning token ordering. Moreover, transformer’s methodology is succinct with support of encoder-decoder mechanism and deep neural network. Therefore, it is possible to infer that applications of transformer can go beyond some recent pre-trained models and/or pre-trained models based on transformer can be improved more.

# Appendices

**A1. Positional encoding**

Although the main strong point of transformer is self-attention mechanism that discovers internal meanings of input data, there is another strong point of transformer, which is *positional encoding* (PE) which allows transformer to process input tokens in parallel so that the requirement of processing sequentially tokens in recurrent neural network (RNN) is removed. Implementing PE is not so difficult but it is necessary to mention PE in the viewpoint of mathematical fundamental. Given sequence ***X*** as *m*x*dm* matrix includes *m* tokens and each token is modeled as a *dm*-element vector ***x****i*, PE technique adds position information into every token vector ***x****i* so that the sequential order of ***x****i* is not necessary to be fixed because tokens themselves contain their positional information. For instance, let ***X*** and POS(***X***) be token sequence (token matrix) and its position matrix, respectively, the ***X*** is added by its POS(***X***).

Note, pos(***x****i*) is position vector of token vector ***x****i* where token matrix ***X*** is the set of token vectors.

Consequently, how to determine the position matrix becomes how to calculate the position pos(*xij*) where *i* is position of the *i*th token and *j* is position of the *j*th numeric value of such token vector such that:

Suppose two successive numeric values such as *j*th numeric value and (*j*+1)th numeric value such that *j* = 2*k* and *j*+1 = 2*k*+1, we need to calculate two kinds of positions as follows:

Fortunately, these positions are easily calculated by sine function and cosine function as follows (Vaswani, et al., 2017, p. 6):

|  |  |
| --- | --- |
|  | (A1.1) |

Therefore,

The PE equation here:

Which is called sinusoidal positional encoding (SPE). Recall that *dm* is model dimension which is the length of token vector ***x****i*, which is often set to be 512 in natural language processing (NLP). Given *i* is the position of token vector ***x****i*, then pos(*i*, *j*) is the position of the *j*th dimension of the *dm*-dimension vector ***x****i*, thus, let *pos*=*i* denote the position of token vector ***x****i*, the PE equation is re-written as follows:

Or,

When the *token position* *pos* is anchored, it is necessary to assert that the dimension position PE(*pos*, *j*) is aligned with the token position *pos*. In other words, the aim of PE in transformer is to make the dimension position PE(*pos*, *j*) to be relative offset from some hinge so that the position matrix POS(***X***) is not affected significantly when tokens are fed and learned in parallel. Let *λj* be the wavelength at the *j*th dimension (Gemini 2025):

Such that:

Suppose the token position is shifted by distance *d* so that the *j*th dimension position PE(*pos*+*d*, *j*) within context of the new token position *pos*+*d* is:

It is necessary to prove that PE(*pos*+*d*, *j*) is aligned with the token position *pos*, which means that PE(*pos*+*d*, *j*) is relative from *pos*. Indeed, we have:

As well as:

As a summary, we obtain (Gemini 2025):

Which implies that the dimension position PE(*pos*+*d*, *j*) at the new shifted token position *pos*+*d* is aligned relatively from the original token position *pos*, which is also the linear transformation of PE(*pos*, *j*) with relatively shifted factors sin(*d*/*λj*) and cos(*d*/*λj*) when sin(*d*/*λj*) and cos(*d*/*λj*) are relative quantities from *pos* because they are only dependent on the shift *d*. Shortly, the dimension position PE(*pos*+*d*, *j*) at token position *pos*+*d* is relative *d*-distance offset from the one PE(*pos*, *d*) at previous token position *pos*.

The PE equation is re-written:

Which implies:

Where notation denotes wise-multiplication and the expression “condition ? 1 : 0” returns 1 if the condition “condition” is true, otherwise, the expression returns 0. Note, given two vectors (matrices) whose have the same dimension, the wise-multiplication operator between them produces a new vector (matrix) whose every element is result of multiplication of one element from a vector (matrix) and one element from another vector (matrix) with constraint that the two elements have the same index in their own vectors (matrices). Although the indexing index(*j*) is not important, the matrix which is the linear mapping from token position *pos* to token position *pos*+*d* is much more important, and so, let:

Such that:

|  |  |
| --- | --- |
|  | (A1.2) |

If the dimension position *j* is ignored so as to cover all dimension positions, the linear mapping can be denoted conventionally as follows:

Which means that the position PE(*pos*+*d*) is linear function of the position PE(*pos*) (Vaswani, et al., 2017). Note, the mapping map(*d*) is always a matrix. As a result, in general case, PE of given sequence (matrix) which is started from particular position *pos* of some token is specified as follows:

When the starting position of ***X*** is moved by distance *d*, PE of ***X*** is specified as follows:

Recall that query matrix *Q* and key matrix *K* is calculated by sequence matrix ***X*** and weigh matrices as follows:

The important aspect of transformer is the probability matrix which in turn depends on the product *QKT* of query matrix *Q* and key matrix *K*:

When ***X*** is added itself to some starting token position *pos*:

Such that:

The product *QKT* is expended as follows:

Let,

We obtain:

In the general case that *Q* and *K* are derived from different movement of starting token positions *posQ* and *posK* such that:

The product *QKT* becomes:

Without loss of generality, suppose the key starting token position *posK* is the one that is shifted *d*-distance from the query starting token position *posQ* such that:

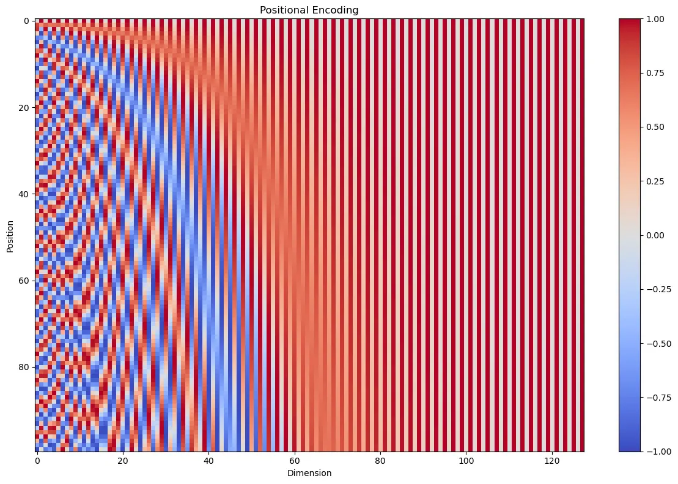
The product *QKT* is not changed in contextual meaning because of the linear mapping map(*d*) that maps PE(***X***, *posQ*) into PE(***X***, *posK*) by the relative *d*-shift in context of relative position.

Please pay attention that the linear mapping map(*d*) is a matrix.

Ordering of tokens is so important in natural language processing (NLP) and large language model (LLM) that recurrent neural network (RNN) is an appropriate NLP model and so, transformer tries its best to reserve the toke ordering by adding positional encoding (PE) information which is different from the mechanism of RNN. It is not easy to assert the preeminence of PE over RNN but another strong point of PE is to support transformer as well as self-attention mechanism to process parallel tokens and segments of tokens. The aforementioned PE which is based on sinusoidal function is called sinusoidal PE, which is explained basically. Now it is necessary to survey carefully the interesting concept of PE (Paleti, 2024).

Given binary presentation of 1-byte integer, for instance, 165 = 27 + 25 + 22 + 1 = 10100101, the least significant (LS) bit is the left most bit and the most significant (MS) bit is the right most bit (Paleti, 2024). Similarly, in the sequence of tokens, for instance, “I am a student”, each token, for instance, “student” is represented by a token vector ***x****i* whose PE vector plays the roles of binary presentation aforementioned in context of significances.

The is the dimension PE of the *j*th dimension of the token ***x****i* among *dm* dimensions is similar to the *j*th bit of the binary presentation of an integer. Therefore, the first dimension is the least significant (LS) one and the last dimension is the most significant (MS) one. As a result, given token PE at the token position , the more significant the dimension PE is, the more stable the dimension PE is. This means that highly significant dimension PE does not change its value much. The following color table (Paleti, 2024) called *heat map* show dimension PE (s) of tokens whose each row represents a token (position) and each column represents a dimension so that each cell is of a dimension given a token.



**Table A1.1.** Color table of dimension PE (s) of tokens

Note, the red color indicates value 1 and the green value indicates value –1 because the image domain of sin function and cosine function is the range [–1, 1]. Please pay attention that it is not totally matched with binary representation of integer, the left most column which is the left most dimension is LS one and the right most column which is the right most dimension is MS one so that the color of the right most dimension is most stable. This phenomenon is explained by effectiveness of sequence of tokens. In the sequence of tokens, for instance, “I am a student”, the right most token “student” is the most significant (MS) token because it stands for the meaning of the whole sentence, or, in other words, the right most token “student” accumulates the meanings of its previous tokens. Of course, the left most token “I” is the least significant (LS) token because it only starts the representation of the whole sentence meaning. As a result, dimension PE is affected by token position where right (left) most token is the most (least) significant one.

**A2. Layer normalization**

There are two improvements such as positional encoding and normalization. The second improvement is layer (network) normalization:

LayerNorm(***X*** + Sublayer(***X***))

LayerNorm(***Y*** + Sublayer(***Y***))

Because residual mechanism is implemented by the sum ***X*** + Sublayer(***X***) or ***Y*** + Sublayer(***Y***), it is necessary to survey the following normalization without loss of generality:

LayerNorm(***x***)

Where ***x*** = (*x*1, *x*2,…, *xn*)*T* is layer of *n* neuron *xi* with note that each neuron *xi* is represented by a number. Suppose ***x*** as a sample conforms normal distribution, its sample mean and variance are calculated as follows:

As a result, layer normalization is distribution normalization such that:

Which is not difficult to be implemented but it is useful to survey layer normalization in literature.

Training deep neural network which implements linear sub-layer of transformer copes with the vanishing gradient problem where target gradient in stochastic gradient descent (SGD) algorithm approaches nearly zero because of three reasons: 1) deep neural network has many layers which decreases the target gradient in propagative learning, 2) learning rate needs to be continuously decreased so as to ensure the convergence of SGD, which forces the target gradient to be continuously decreased, and 3) learning error is always continuously decreased due to effectiveness of SGD, which forces the target gradient to be continuously decreased too. That the SGD target gradient approaches nearly zero makes SGD itself to be stuck in some point, which in turn makes SGD ineffective. *Residual connection* (RC) is a popular technique to solve the vanishing gradient problem, for instance, given deep neural network denoted by function *F*(***x*** | Θ) where ***x*** is input layer and Θ denotes parametric weights, RC technique adds the input vector (input layer) ***x*** into the *F*(***x*** | Θ) such that:

Therefore, the Add & Norm component of transformer as Add & Norm = LayerNorm(***X*** + Sublayer(***X***)) is actually the incorporation of RC and normalization. Note, residual connection (RC) is also called *residual mechanism* or *skip connection*. Given the real output ***y***’ from environment, let *l*(***y***) be the likelihood for training the deep neural network *F*(***x***):

The association of stochastic gradient descent (SGD) algorithm and backpropagation algorithm is based on calculating the gradient of *l*(***y***) with respect to ***y***.

Where *F*’(***x*** | Θ) is the gradient / first-order derivative of *F*(***x*** | Θ) with respect to ***x***.

Of course, the gradient of *l*(***y***) with respect to parameter Θ will be expended from the gradient ∇*l*(***y***) later but please pay attention to the most important thing that the expression (**1** + *F*’(***x*** | Θ)) alleviates the vanishing gradient problem that *F*’(***x*** | Θ) may approach nearly zero by the occurrence of **1** with note that the notation **1** often indicates identity matrix so that the gradient ∇*l*(***y***) is never zero given ***y***’ ≠ ***y*** due to the expression (**1** + *F*’(***x*** | Θ)) when *F*’(***x*** | Θ) is non-negative. Note, *F*’(***x*** | Θ) is often non-negative because the derivative of activation function like sigmoid function is often non-negative.

# References

Alammar, J. (2018, June 27). *The Illustrated Transformer*. (GitHub) Retrieved June 2024, from Jay Alammar website: https://jalammar.github.io/illustrated-transformer

Cho, K., Merrienboer, B. v., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014, September 3). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *arXiv preprint*, 1-15. doi:10.48550/arXiv.1406.1078

Graves, A. (2014, June 5). Generating Sequences With Recurrent Neural Networks. *arXiv preprint*, 1-43. doi:10.48550/arXiv.1308.0850

Han, X., Zhang, Z., Ding, N., Gu, Y., Liu, X., Huo, Y., . . . Zhu, J. (2021, August 26). Pre-trained models: Past, present and future. *AI Open, 2*(2021), 225-250. doi:10.1016/j.aiopen.2021.08.002

Hardle, W., & Simar, L. (2013). *Applied Multivariate Statistical Analysis.* Berlin, Germany: Research Data Center, School of Business and Economics, Humboldt University.

Paleti, C. N. (2024, July 5). *Positional Encoding Explained: A Deep Dive into Transformer PE.* Retrieved August 2, 2025, from The Deep Hub: https://medium.com/thedeephub/positional-encoding-explained-a-deep-dive-into-transformer-pe-65cfe8cfe10b

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., . . . Polosukhin, I. (2017). Attention Is All You Need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, & S. Vishwanathan (Ed.), *Advances in Neural Information Processing Systems (NIPS 2017).* *30.* Long Beach: NeurIPS. Retrieved from https://arxiv.org/abs/1706.03762

Voita, L. (2023, November 17). *Sequence to Sequence (seq2seq) and Attention*. (GitHub) Retrieved June 2024, from Elena (Lena) Voita website: https://lena-voita.github.io/nlp\_course/seq2seq\_and\_attention.html

Wikipedia. (2005, April 7). *Recurrent neural network*. (Wikimedia Foundation) Retrieved from Wikipedia website: https://en.wikipedia.org/wiki/Recurrent\_neural\_network

Wikipedia. (2019, August 25). *Transformer (deep learning architecture)*. (Wikimedia Foundation) Retrieved from Wikipedia website: https://en.wikipedia.org/wiki/Transformer\_(deep\_learning\_architecture)