

10.07 세미나 발표

Attention

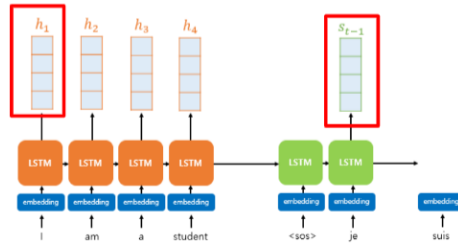
1. Attention Score를 구하는 과정에서 학습이 가능한 가중치 행렬을 사용합니다

$$score(s_{t-1}, h_i) = W_a^T \tanh(W_b s_{t-1} + W_c h_i)$$

$S(t-1)$ = 이전 시점의 LSTM hidden state

$H(i)$ = 인코더 부분의 LSTM hidden state

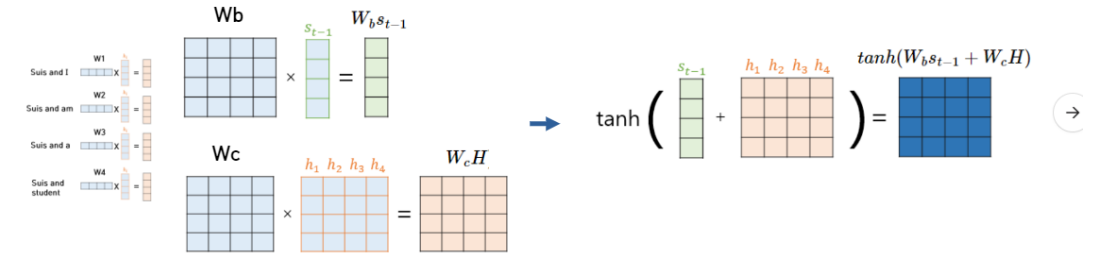
W_a, W_b, W_c : 학습하는 가중치 행렬



그림자료 : 딥러닝을 사용한 자연어 처리 입문

Attention

$$score(s_{t-1}, H) = W_a^T \tanh(W_b s_{t-1} + W_c H)$$



그림자료 : 딥러닝을 사용한 자연어 처리 입문

Attention

$$score(s_{t-1}, H) = W_a^T \tanh(W_b s_{t-1} + W_c H)$$

$$\tanh \left(\begin{matrix} s_{t-1} \\ h_1 \ h_2 \ h_3 \ h_4 \end{matrix} \right) = \begin{matrix} \tanh(W_b s_{t-1} + W_c H) \end{matrix} \rightarrow \begin{matrix} W_a^T \end{matrix} \times \begin{matrix} \tanh(W_b s_{t-1} + W_c H) \end{matrix} = \begin{matrix} \text{Attention Score} \\ h_1 \ h_2 \ h_3 \ h_4 \end{matrix}$$

$$\text{softmax} \left(\begin{matrix} \text{Attention Score} \\ h_1 \ h_2 \ h_3 \ h_4 \end{matrix} \right) = \begin{matrix} \text{Attention Distribution} \end{matrix} \rightarrow \begin{matrix} h_1 \ h_2 \ h_3 \ h_4 \end{matrix} \times \begin{matrix} \text{Attention Distribution} \end{matrix} = \begin{matrix} \text{Context Vector} \end{matrix}$$

그림자료 : 딥러닝을 사용한 자연어 처리 입문

Transformer

Attention Is All You Need

Ashish Vaswani*
Google Brain
avaswani@google.com

Noam Shazeer*
Google Brain
noam@google.com

Niki Parmar*
Google Research
nikip@google.com

Jakob Uszkoreit*
Google Research
usz@google.com

Llion Jones*
Google Research
llion@google.com

Aidan N. Gomez* †
University of Toronto
aidan@cs.toronto.edu

Lukasz Kaiser*
Google Brain
lukaszkaiser@google.com

Illia Polosukhin* ‡
illia.polosukhin@gmail.com

주요 관점 :

병렬성을 활용하여, 성능을 향상시키며, 시간을 단축하려는 목표
-> 기존의 LSTM Layer를 사용하지 않음

Transformer

Transformer :
기존 LSTM(RNN)의 구조를 Attention으로 대체한 모델 (encoder와 decoder는 사용)

1 Introduction

Recurrent models typically factor computation along the symbol positions of the input and output sequences. Aligning the positions to steps in computation time, they generate a sequence of hidden states h_t , as a function of the previous hidden state h_{t-1} and the input for position t . This inherently sequential nature precludes parallelization within training examples, which becomes critical at longer sequence lengths, as memory constraints limit batching across examples. Recent work has achieved significant improvements in computational efficiency through factorization tricks [21] and conditional computation [32], while also improving model performance in case of the latter. The fundamental constraint of sequential computation, however, remains.

RNN은 input과 output의 계산이 sequence하게 이루어 짐

- > 전체적인 계산의 복잡도 및 학습 시간을 늘리는 문제점 발생
- > 이를 해결하기 위한 노력들의 연구들은 진행되고 있음

하지만 근본적인 sequential computation의 제약조건은 해결되지 않음.

Transformer

1 Introduction

Attention mechanisms have become an integral part of compelling sequence modeling and transduction models in various tasks, allowing modeling of dependencies without regard to their distance in the input or output sequences [2, 19]. In all but a few cases [27], however, such attention mechanisms are used in conjunction with a recurrent network.

In this work we propose the Transformer, a model architecture eschewing recurrence and instead relying entirely on an attention mechanism to draw global dependencies between input and output. The Transformer allows for significantly more parallelization and can reach a new state of the art in translation quality after being trained for as little as twelve hours on eight P100 GPUs.

Attention 연산은 input과 output sequences에 영향을 받지 않는다.

이러한 Attention 모델은 주로 RNN(LSTM)과 결합되어 함께 사용됩니다. -> Seq2Seq + Attention(자연어 처리)

저자 : 트랜스포머라는 것을 제안하고,

이러한 트랜스포머는 더욱 병렬적인 계산이 가능하고, 기존보다 좋은 품질의 번역을 가능하게 합니다.

Transformer

2 Background

The goal of reducing sequential computation also forms the foundation of the Extended Neural GPU [16], ByteNet [18] and ConvS2S [9], all of which use convolutional neural networks as basic building block, computing hidden representations in parallel for all input and output positions. In these models, the number of operations required to relate signals from two arbitrary input or output positions grows in the distance between positions, linearly for ConvS2S and logarithmically for ByteNet. This makes it more difficult to learn dependencies between distant positions [12]. In the Transformer this is reduced to a constant number of operations, albeit at the cost of reduced effective resolution due to averaging attention-weighted positions, an effect we counteract with Multi-Head Attention as described in section 3.2.

목표는 sequential computation을 줄이는 것이며,
이는 ByteNet, ConvS2S의 기초 아이디어입니다.
Byte Net, ConvS2S, Extend Neural GPU는 대부분 CNN 모델을 사용하며,
긴 거리를 계산할 때의 계산 복잡도는 다음과 같습니다.

ConvS2S : Order by (n)

ByteNet : Order by (log n)

제시된 Transformer : Order by(1)

계산과정이 줄어드니까 성능이 다운
Multi-Head-Attention으로 이를 대응하여 해결

Transformer

2 Background

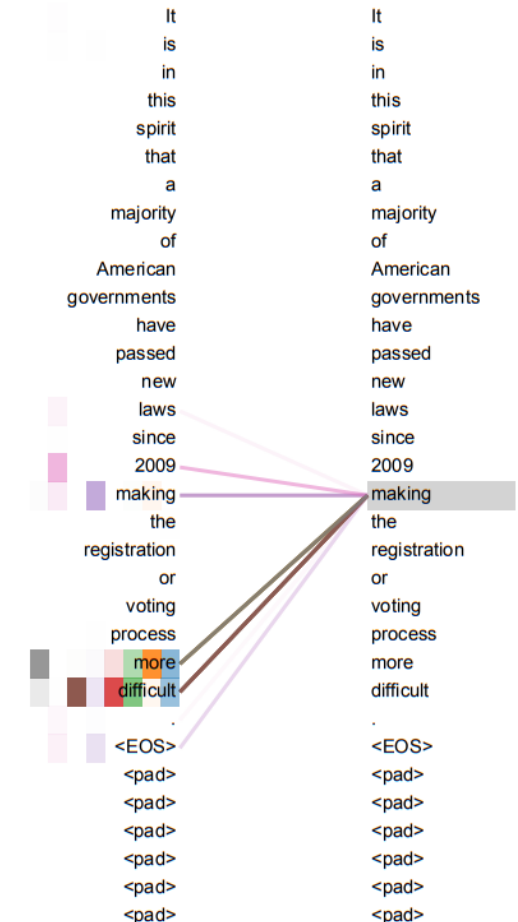
Self-attention, sometimes called intra-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence. Self-attention has been used successfully in a variety of tasks including reading comprehension, abstractive summarization, textual entailment and learning task-independent sentence representations [4, 27, 28, 22].

End-to-end memory networks are based on a recurrent attention mechanism instead of sequence-aligned recurrence and have been shown to perform well on simple-language question answering and language modeling tasks [34].

To the best of our knowledge, however, the Transformer is the first transduction model relying entirely on self-attention to compute representations of its input and output without using sequence-aligned RNNs or convolution. In the following sections, we will describe the Transformer, motivate self-attention and discuss its advantages over models such as [17, 18] and [9].

Self Attention은 하나의 문장에서 각 단어와 , 다른 위치에 있는 단어의 연관성을 계산해주는 Attention이며, intra-attention이라고도 불립니다.
Self - attention 는 독해, 문장 요약 등.. 에 사용됩니다.

제시한 Transformer 모델은 대부분을 self attention 구조로 사용한 최초의 번역모델 입니다.



Transformer

3 Model Architecture

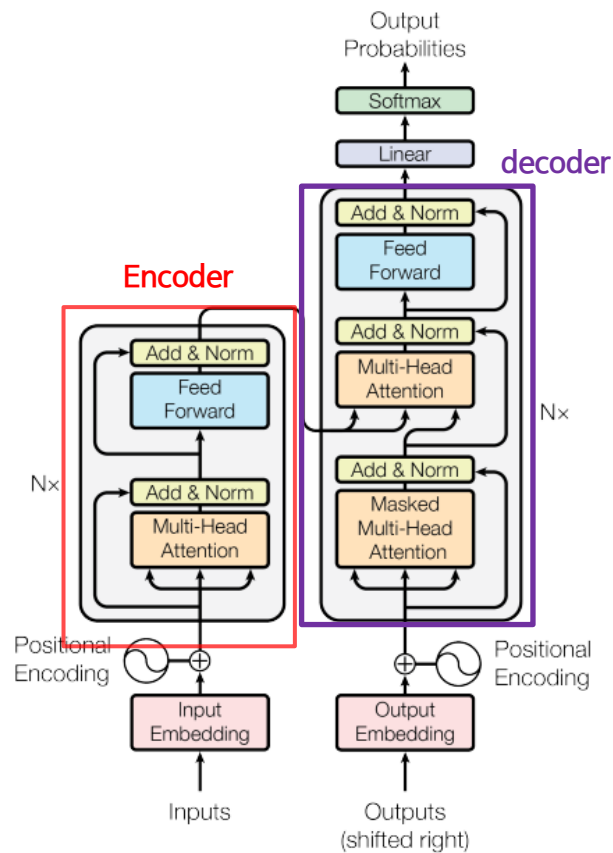


Figure 1: The Transformer - model architecture.

Seq2Seq + Attention와 같은 대부분의 모델 설명

Most competitive neural sequence transduction models have an **encoder-decoder structure** [5, 2, 35]. Here, the encoder maps an input sequence of symbol representations (x_1, \dots, x_n) to a sequence of continuous representations $\mathbf{z} = (z_1, \dots, z_n)$. Given \mathbf{z} , the decoder then generates an output sequence (y_1, \dots, y_m) of symbols one element at a time. At each step the model is auto-regressive [10], consuming the **previously generated symbols as additional input when generating the next.**

The Transformer follows this overall architecture using **stacked self-attention and point-wise, fully connected layers for both the encoder and decoder**, shown in the left and right halves of Figure 1, respectively.

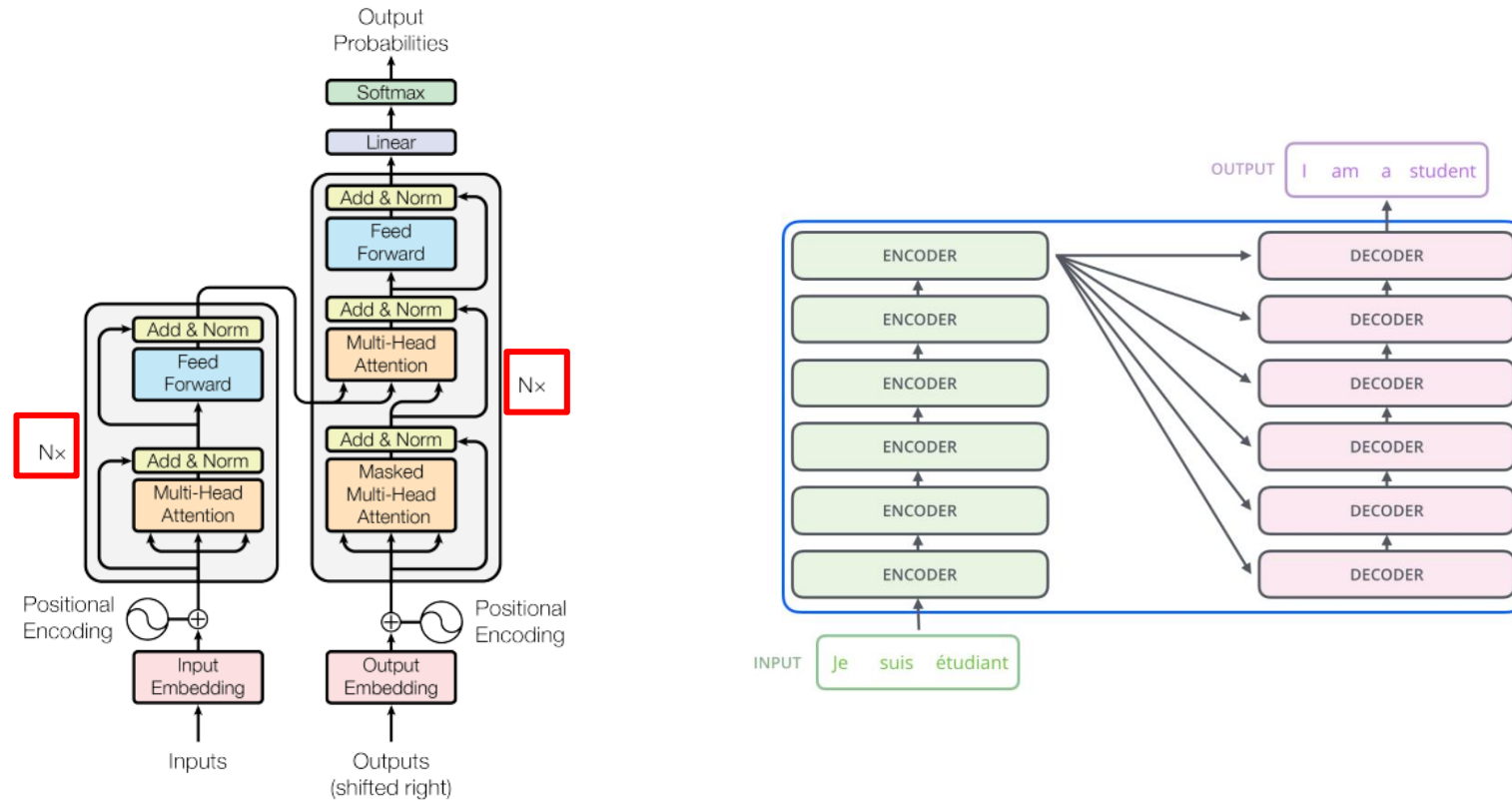
대부분의 경쟁력 있는 모델들은
Encoder - Decoder 구조를 가지고 있다.

진행방식 : Input $(x_1, x_2 \dots x_n) \rightarrow$ hidden state $(z_1, z_2 \dots z_n) \rightarrow$ output $(y_1, y_2 \dots y_n)$
(one element at a time)

Transformer는 stack(self-attention과 point-wise, fully connected layers) 구조

- 배워야 할 개념
(sub layer)
1. Input Embedding
 2. Positional Encoding
 3. Multi-Head Attention
 4. Add & Norm
 5. Feed Forward
 6. Masked Multi-Head Attention

Transformer



Transformer encoder, decoder 특징 :

입력과 출력의 차원이 항상 동일해야 해당 구조가 성립 가능

논문에서는 총 6개의 encoder와 decoder를 사용하였습니다.

Transformer

배워야 할 개념

1. **Input Embedding**
2. Positional Encoding
3. Multi-Head Attention
4. Add & Norm
5. Feed Forward
6. Masked Multi-Head Attention

Embedding이란:

자연어 처리 부분에서 많이 사용되며,
들어오는 입력에 대한
단어 사전을 만드는 개념입니다.

들어오는 입력 값이 I am a boy 라면

각각은 숫자에 대응되어서, 1, 5, 21, 30 (단어의 단위)
이러한 전처리를 거치게 될 텐데

단순 숫자로만 사용할 경우 유의어나 반의어 등의
단어사이의 연관성을 파악하기 힘드므로,
고정된 크기의 학습된 벡터를 사용합니다.

1 -> (0.1 0.2 0.4 0.5 0.7)
5 -> (0.3 0.1 0.2 0.4 0.6)
21 -> (0.2 0.8 0.3 0.1 0.9)
30 -> (0.4 0.5 0.9 0.5 0.2)

벡터 값을 만들어서 학습할 수 있는
전체 사전의 크기 또한 지정이 필요함

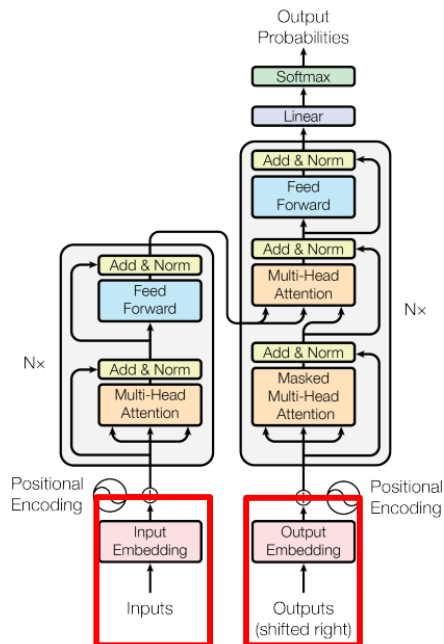


Figure 1: The Transformer - model architecture.

`tf.keras.layers.Embedding`(전체 사전의 크기, 출력 벡터의 크기, 입력의 크기)

Transformer

Embedding이란:

결국 단어가 입력되었을 때 고정된 크기의 벡터 값으로 변환해주는 Layer 입니다.

Ex) 고정된 벡터의 크기가 “8” 일 경우

“I” -> (0.1, 0.3, 0.2, 0.4, 0.62, 0.34, 0.54, 0.23)
“am” -> (0.4, 0.5, 0.32, 0.1, 0.52, 0.54, 0.42, 0.57)
“a” -> (0.15, 0.33, 0.64, 0.32, 0.4, 0.77, 0.85, 0.89)
“boy” -> (0.13, 0.30, 0.44, 0.52, 0.7, 0.5, 0.97, 0.13)

Ex) 고정된 벡터의 크기가 “5” 일 경우

“I” -> (0.61, 0.34, 0.22, 0.51, 0.68)
“am” -> (0.32, 0.46, 0.9, 0.1, 0.16)
“a” -> (0.46, 0.15, 0.5, 0.6, 0.46)
“boy” -> (0.97, 0.13, 0.6, 0.13, 0.47)

-> Transformer에서는 이러한 Embedding dimension을 512로 설정하였습니다.

Transformer

3.4 Embeddings and Softmax

Similarly to other sequence transduction models, we use learned embeddings to convert the input tokens and output tokens to vectors of dimension d_{model} . We also use the usual learned linear transformation and softmax function to convert the decoder output to predicted next-token probabilities. In our model, we share the same weight matrix between the two embedding layers and the pre-softmax linear transformation, similar to [30]. In the embedding layers, we multiply those weights by $\sqrt{d_{\text{model}}}$.

각 단어와의 유사성을 미리 학습해왔던 다른 학습된 임베딩 모델을 가지고 와서 벡터로 변환해줍니다.

-> 자연어 처리에서 매우 많이 사용되는 방식이며, 이미 수많은 데이터로 학습이 잘 된 Embedding layer를 활용해서, 벡터 값으로 변환해줍니다.

Transformer에서는 이러한 입력 값을 d_{model} 의 루트 값으로 곱해줍니다.

d_{model} : embedding size이며 해당 논문에서는 512로 설정

-> 곱해주는 이유는 추 후에 배율 포지셔널 인코딩을 더해주는 과정에서, 원본 정보의 손상을 덜하기 위해서 입니다.(다시 언급)

Transformer

배워야 할 개념

1. Input Embedding
2. Positional Encoding
3. Multi-Head Attention
4. Add & Norm
5. Feed Forward
6. Masked Multi-Head Attention

Positional Encoding 이란 :

결국 들어오는 입력에 대한 정보가 순차적으로 들어오지 않기 때문에,
각 입력 데이터들이 어느 데이터가 처음에 오고,
어느 데이터가 마지막에 오는지에 대한 위치 정보를 받을 수 없습니다.

이를 보완하기 위한 것이
Positional Encoding입니다.

각 위치 정보에 따라서, 값을 부여하고
해당 값을 더해줌으로써,
완벽하진 않더라도, 위치정보를 가지고 학습이 가능합니다.

-> 더해줘야 하기 때문에, embedding vector와
Positional Encoding의 사이즈는 동일해야 합니다.

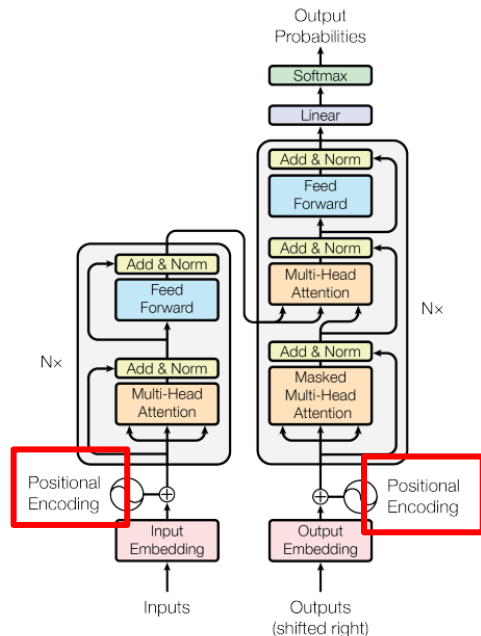


Figure 1: The Transformer - model architecture.

3.5 Positional Encoding

Since our model contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the sequence. To this end, we add "positional encodings" to the input embeddings at the bottoms of the encoder and decoder stacks. The positional encodings have the same dimension d_{model} as the embeddings, so that the two can be summed. There are many choices of positional encodings, learned and fixed [9].

Transformer

배워야 할 개념

1. Input Embedding
2. Positional Encoding
3. Multi-Head Attention
4. Add & Norm
5. Feed Forward
6. Masked Multi-Head Attention

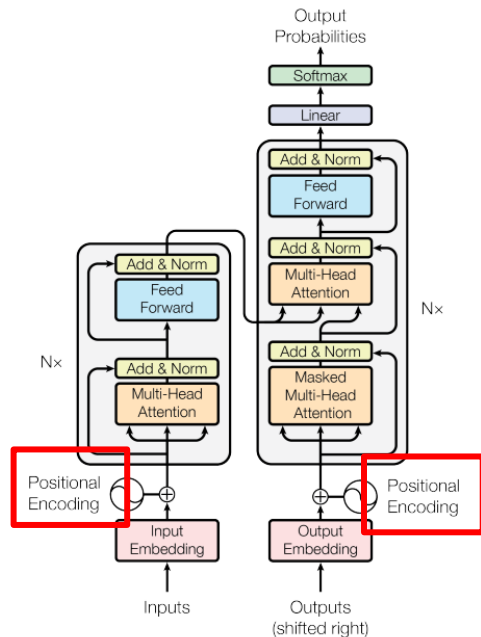
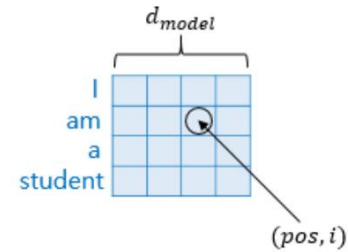


Figure 1: The Transformer - model architecture.

$$PE_{(pos, 2i)} = \sin(pos/10000^{2i/d_{model}})$$

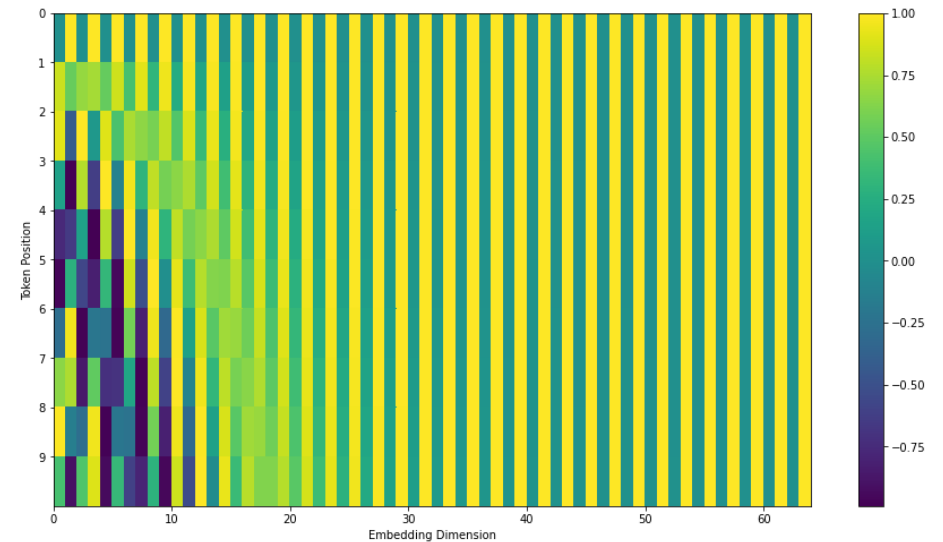
$$PE_{(pos, 2i+1)} = \cos(pos/10000^{2i/d_{model}})$$



직관적인 해석을 위한 예시

0 :	0	0	0	0	0
1 :	0	0	0	1	0
2 :	0	0	1	0	0
3 :	0	0	1	1	0
4 :	0	1	0	0	0
5 :	0	1	0	1	0
6 :	0	1	1	0	0
7 :	0	1	1	1	0
8 :	1	0	0	0	0
9 :	1	0	0	1	0
10 :	1	0	1	0	0
11 :	1	0	1	1	0
12 :	1	1	0	0	0
13 :	1	1	0	1	0
14 :	1	1	1	0	0
15 :	1	1	1	1	0

문장의 길이



임베딩 차원

Transformer

배워야 할 개념

1. Input Embedding
2. Positional Encoding
3. Multi-Head Attention
4. Add & Norm
5. Feed Forward
6. Masked Multi-Head Attention

3.5 Positional Encoding

We also experimented with using learned positional embeddings [9] instead, and found that the two versions produced nearly identical results (see Table 3 row (E)). We chose the sinusoidal version because it may allow the model to extrapolate to sequence lengths longer than the ones encountered during training.

논문의 저자들은

1. learned positional embedding
2. Positional encoding

두 가지의 방법을 사용해서, 실험을 하였고,
결과값은 거의 비슷했지만,

교육했을 때의 문장의 길이보다 더 긴 문장을 예측하는 것이
Positional encoding에서는 가능하기 때문에 이를 사용했다.

Positional embedding : 학습 파라미터로 Position을 학습
-> 실제 예측일 경우 학습 이상의 Position이 불가능

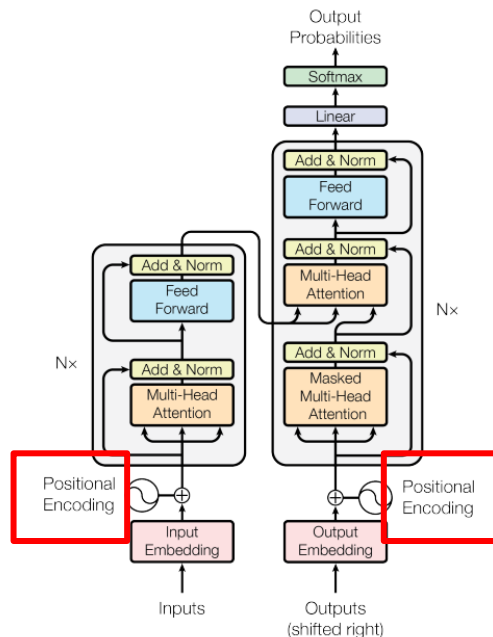
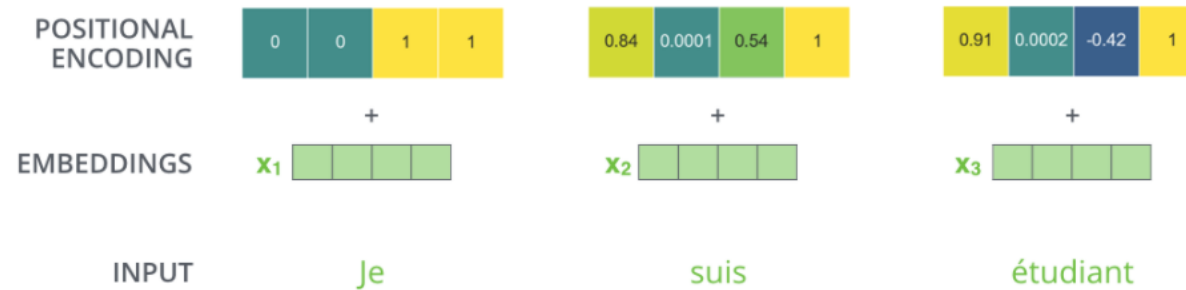


Figure 1: The Transformer - model architecture.

Transformer

이렇게 구한 Positional encoding을
Embedding vector에 더해지게 됩니다.



위치 정보를 이렇게 주게 되며,
결국 Positional encoding이
원본 embedding vector에 더해지면서 영향을 미치기 때문에

전에 말했던 임베딩 벡터 * 루트(임베딩 차원)을 통해,
기본 값을 더 크게 만들어서

Position encoding이 embedding vector의 원본 값을 덜 영향 주게끔 합니다.

-> 입력 데이터에 대한 전처리 과정 종료

Transformer

배워야 할 개념

1. Input Embedding
2. Positional Encoding
3. Multi-Head Attention
4. Add & Norm
5. Feed Forward
6. Masked Multi-Head Attention

Multi-Head Attention이란 ?

Attention : (대부분) Self Attention

Multi-Head : 여러 개의 Attention

-> 여러 개의 Attention을 사용한 Layer

1. Self Attention의 과정
2. 여러 개의 Attention 연산을 한번에 처리하는 방법

Self Attention : Scaled dot product attention

dot product attention = 루 옹 어텐션

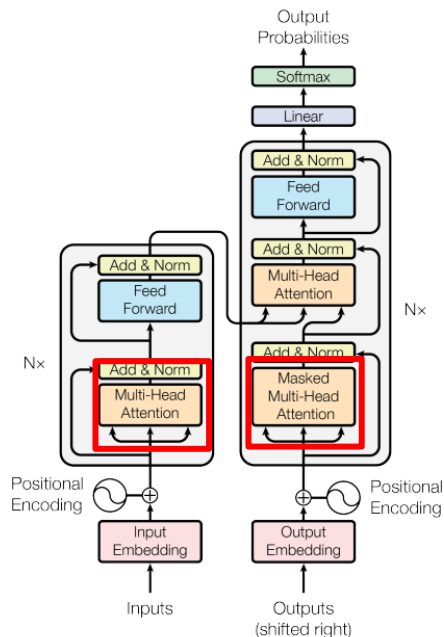


Figure 1: The Transformer - model architecture.

Transformer

출처: Alammr

배워야 할 개념

1. Input Embedding
2. Positional Encoding
3. Multi-Head Attention
4. Add & Norm
5. Feed Forward
6. Masked Multi-Head Attention

Attention 연산을 하기 위하여
Embedding Vector \rightarrow Q, K, V

바꾸는 과정 : W_q , W_k , W_v (학습가능한 가중치)를
Embedding Vector와 곱해져서 바꿔줍니다.

Q : 비교할 대상

K : 비교 당하는 대상

V : Attention Score를 구한 뒤 곱에 사용

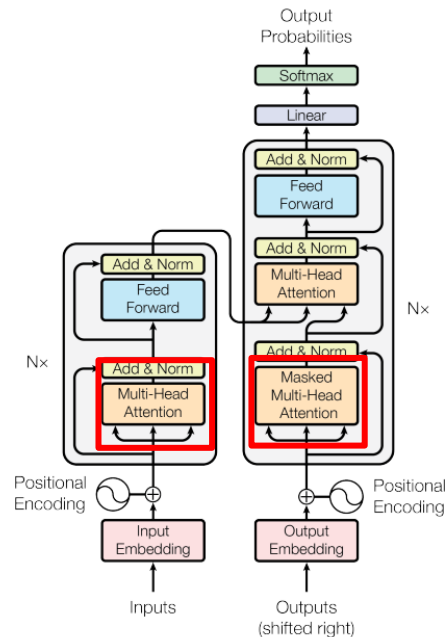
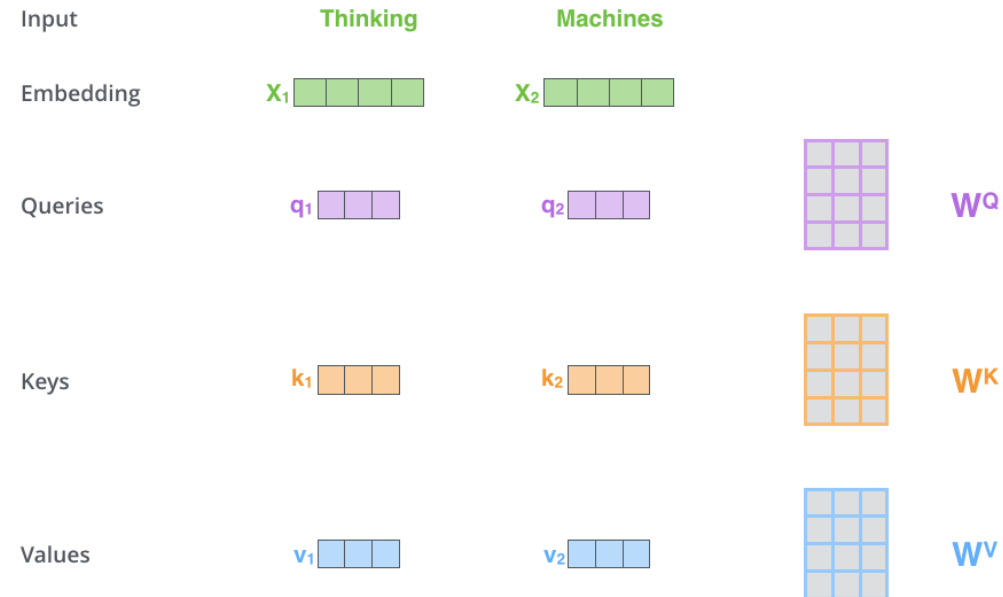


Figure 1: The Transformer - model architecture.



Transformer

Q : 비교할 대상
K : 비교 당하는 대상
V : Attention Score를 구한 뒤 곱에 사용

행렬로써 한번에 계산이 가능
모든 입력에 대한 임베딩 벡터를 하나의 행렬로 만들어 버림
→ W_q , W_k , W_v 는 하나의 Head에서 공유되기 때문

$$X \times W^Q = Q$$


$$X \times W^K = K$$

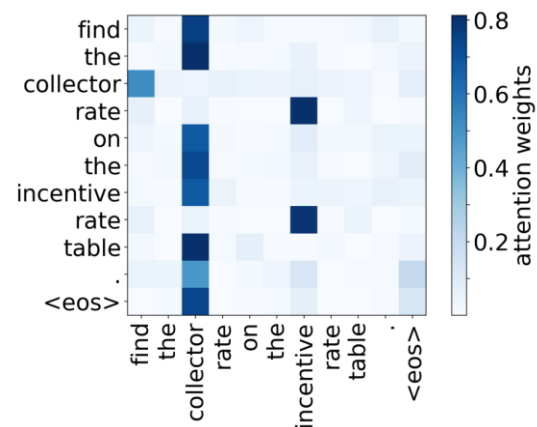

$$X \times W^V = V$$

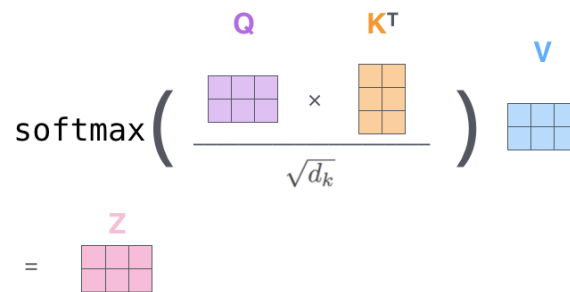

계산을 된 값을
root(임베딩 벡터의 차원)으로 나눔

Scaled Attention score를 만듦

Value Vector와 곱해서 output을 만듦

Ex) Self Attention output



$$\text{softmax} \left(\frac{Q \times K^T}{\sqrt{d_k}} \right) \times V = Z$$


출처: Alammr

Transformer

출처: Alammr

3.2.1 Scaled Dot-Product Attention

We call our particular attention "Scaled Dot-Product Attention" (Figure 2). The input consists of queries and keys of dimension d_k , and values of dimension d_v . We compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix Q . The keys and values are also packed together into matrices K and V . We compute the matrix of outputs as:

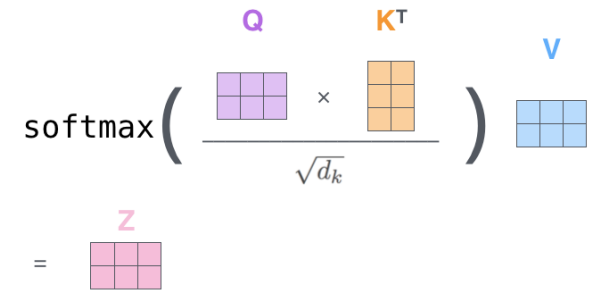
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (1)$$

The two most commonly used attention functions are additive attention [2], and dot-product (multiplicative) attention. Dot-product attention is identical to our algorithm, except for the scaling factor of $\frac{1}{\sqrt{d_k}}$. Additive attention computes the compatibility function using a feed-forward network with a single hidden layer. While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

➡ Dot product attention을 사용하는 이유
더 빠르고, 효율적이다.

While for small values of d_k the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of d_k [3]. We suspect that for large values of d_k , the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients⁴. To counteract this effect, we scale the dot products by $\frac{1}{\sqrt{d_k}}$.

➡ 루트(임베딩 차원)으로 나눠주는 이유
기울기 소실 현상을 막기 위해서



Transformer

배워야 할 개념

1. Input Embedding
2. Positional Encoding
3. Multi-Head Attention
4. Add & Norm
5. Feed Forward
6. Masked Multi-Head Attention

Multi-Head Attention이란 ?

Attention : (대부분) Self Attention

Multi-Head : 여러 개의 Attention

-> 여러 개의 Attention을 사용한 Layer

1. Self Attention의 과정
2. 여러 개의 Attention 연산을 한번에 처리하는 방법

Self Attention : Scaled dot product attention

dot product attention = 루 옹 어텐션

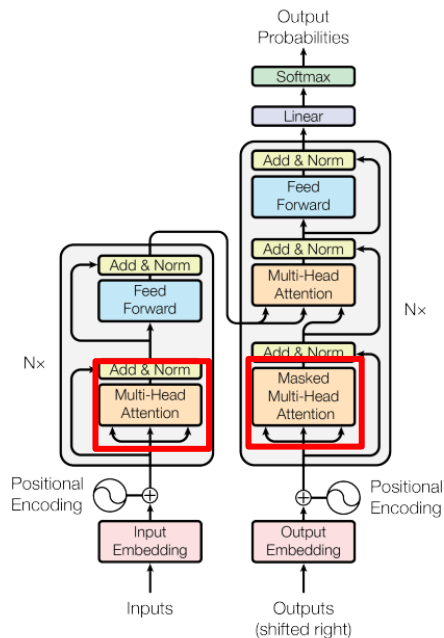


Figure 1: The Transformer - model architecture.

Transformer

여러 개의 Attention 연산을 한번에 처리하는 방법

Embedding이란:

결국 단어가 입력되었을 때 고정된 크기의 벡터 값으로 변환해주는 Layer 입니다.

Ex) 고정된 벡터의 크기가 “8” 일 경우

“I” -> (0.1, 0.3, 0.2, 0.4, 0.62, 0.34, 0.54, 0.23)
“am” -> (0.4, 0.5, 0.32, 0.1, 0.52, 0.54, 0.42, 0.57)
“a” -> (0.15, 0.33, 0.64, 0.32, 0.4, 0.77, 0.85, 0.89)
“boy” -> (0.13, 0.30, 0.44, 0.52, 0.7, 0.5, 0.97, 0.13)

Embedding Vector를 사용

-> 하나의 단어에 대한 설명이 Embedding Vector의 차원으로 변경되므로
Embedding Vector를 내가 원하는 Head의 수 만큼으로 나눠서, 해결한다.

위의 예시에서, Head의 개수가 2이라고 할 경우

$$8/2 = 4$$

하나의 Attention Vector당 4개의 Embedding 벡터를 가져간다.

Transformer

여러 개의 Attention 연산을 한번에 처리하는 방법

Basic) Head

Multi - Head

(Head 개수 : 2개)

(embedding size / head_num = multi head embedding dim)

“I” -> (0.1, 0.3, 0.2, 0.4 0.62, 0.34, 0.54,0.23)
“am” -> (0.4, 0.5, 0.32, 0.1, 0.52, 0.54, 0.42, 0.57)
“a” -> (0.15, 0.33, 0.64, 0.32, 0.4, 0.77, 0.85, 0.89)
“boy” -> (0.13, 0.30, 0.44, 0.52, 0.7, 0.5, 0.97,0.13)

head1
“I” -> (0.1, 0.3, 0.2, 0.4)
“am” -> (0.4, 0.5, 0.32, 0.1)
“a” -> (0.15, 0.33, 0.64, 0.32)
“boy” -> (0.13, 0.30, 0.44, 0.52)

head2
“I” -> (0.62, 0.34, 0.54,0.23)
“am” -> (0.52, 0.54, 0.42, 0.57)
“a” -> (0.4, 0.77, 0.85, 0.89)
“boy” -> (0.7, 0.5, 0.97,0.13)

입력 embedding dim을 Head의 개수 만큼 나누게 된다면,

동일한 데이터 내부에서 Head의 개수만큼의 Self-Attention이 가능해집니다,

Self Attention 1개(그림 왼쪽) vs Self-Attention 2개(그림 오른쪽)

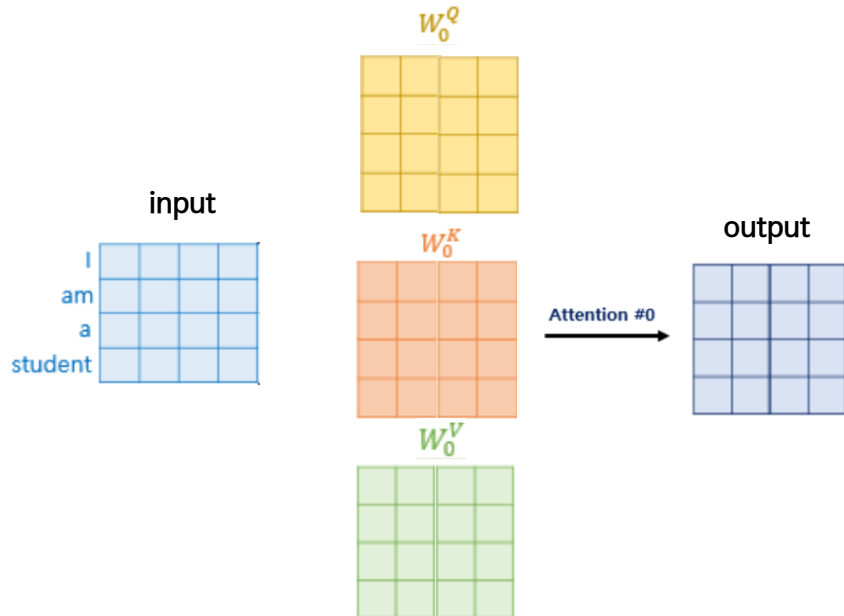
1. Self-Attention의 개수가 늘어나기 때문에, Size가 기존과 달라지는가?
2. Self-Attention의 개수가 늘어났을 때를 시각화

Transformer

Basic) Head

“I” -> (0.1, 0.3, 0.2, 0.4 0.62, 0.34, 0.54,0.23)
“am” -> (0.4, 0.5, 0.32, 0.1, 0.52, 0.54, 0.42, 0.57)
“a” -> (0.15, 0.33, 0.64, 0.32, 0.4, 0.77, 0.85, 0.89)
“boy” -> (0.13, 0.30, 0.44, 0.52, 0.7, 0.5, 0.97,0.13)

Self Attention 1개 (그림 왼쪽)



Multi - Head

head1

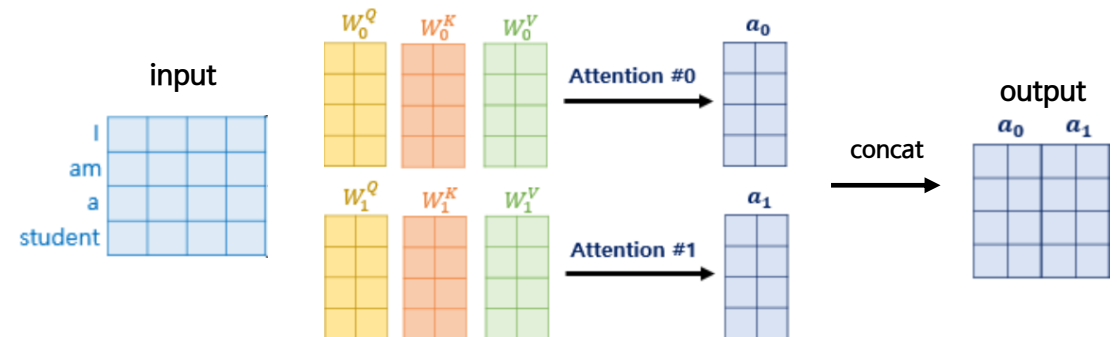
“I” -> (0.1, 0.3, 0.2, 0.4)
“am” -> (0.4, 0.5, 0.32, 0.1)
“a” -> (0.15, 0.33, 0.64, 0.32)
“boy” -> (0.13, 0.30, 0.44, 0.52)

head2

“I” -> (0.62, 0.34, 0.54,0.23)
“am” -> (0.52, 0.54, 0.42, 0.57)
“a” -> (0.4, 0.77, 0.85, 0.89)
“boy” -> (0.7, 0.5, 0.97,0.13)

vs

Self-Attention 2개 (그림 오른쪽)



Transformer

3.2.2 Multi-Head Attention

Instead of performing a single attention function with d_{model} -dimensional keys, values and queries, we found it beneficial to linearly project the queries, keys and values h times with different, learned linear projections to d_k , d_k and d_v dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding d_v -dimensional output values. These are concatenated and once again projected, resulting in the final values, as depicted in Figure 2.

Single Attention 대신 h 번의 Attention을 하며,
이는 병렬적으로 작용하고, 최종 결과물을 concatenate하고,
마지막으로 Fully connected layer에 넣어서, 최종 결과물을 낸다고 합니다.

Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$
$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

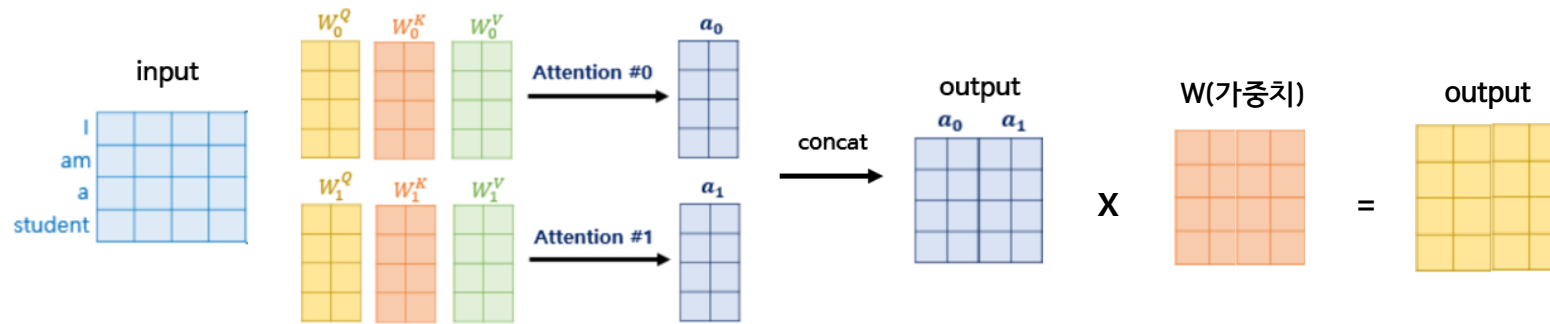
➡ 수식으로 표현

여러 개의 어텐션을 사용한다면, 하나의 정보를 공동으로 사용하여,
다른 위치의 하위 공간에서 얻을 수 있는 정보를 여러 가지로 표현할 수 있습니다.

Transformer

의문) concatenate를 한 뒤 Fully connected layer에 왜 넣는가. (논문에서 언급 X)

Concatenate를 한 뒤의 output size이 기존의 input size와 동일



Word_length = 4

Embedding size = 20

Head_num = 5

각 단어의 dimension = $20 / 5 = 4$

Output.shape = (4, 20) -> concatenate 했을 때 input과 output shape가 동일

Word_length = 4

Embedding size = 20

Head_num = 7

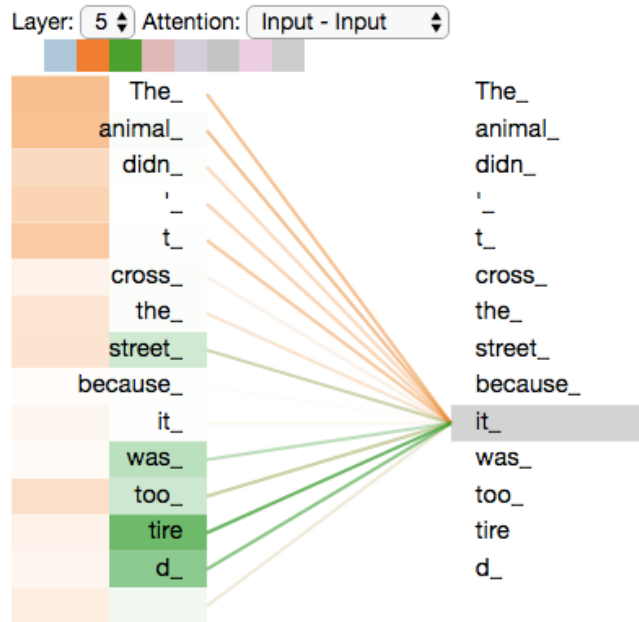
각 단어의 dimension = $20 / 7 = 2.xx$

최종 Output.shape = (4, 14) -> concatenate 했을 때 input과 output shape가 다름

Transformer

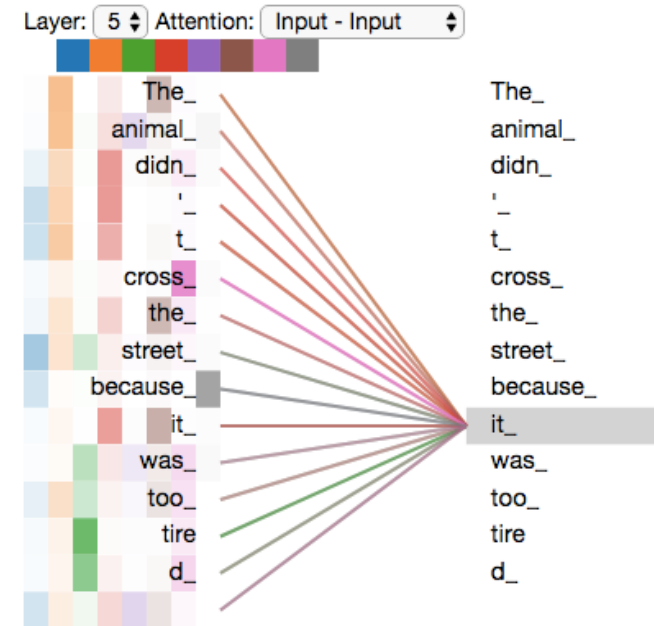
출처: Alammar

문제 : It은 어떤 단어를 지칭할까요?



Attention의 head가 2개 인 상황

It은 The animal, tire 두 개만을 의심



Attention의 head가 8개 인 상황

It은 The animal, tire 뿐만 아닌 다른 여러 가지의 상황을 생각

Transformer

배워야 할 개념

1. Input Embedding
2. Positional Encoding
3. Multi-Head Attention
4. Add & Norm
5. Feed Forward
6. Masked Multi-Head Attention

Add & Norm 이란 ?

Add : Residual connection

Norm : Layer Normalization

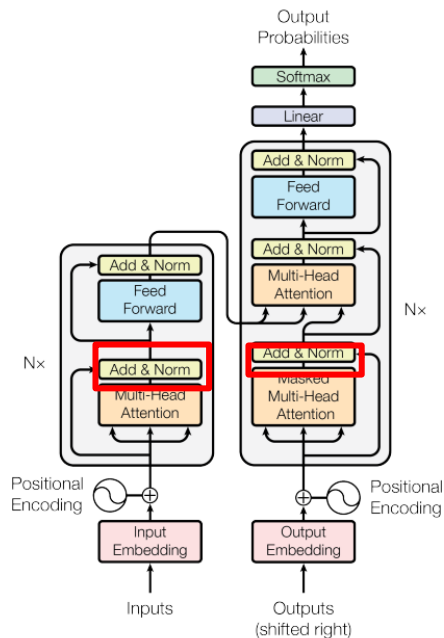
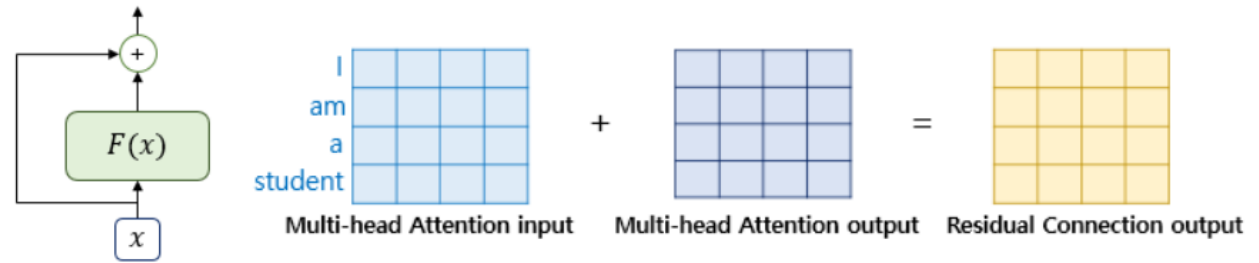


Figure 1: The Transformer - model architecture.

$$H(x) = x + F(x)$$



Residual connection : 입력을 출력에 더해주는 과정

Why : 어텐션 연산과 더불어, 병렬적인 과정의 연산이 이어지므로, 기존 데이터의 속성을 잃을 수 있는 문제가 생기는데, 이를 해결해주어서, 교육을 더 잘 될 수 있도록 도움을 줌

주의) 입력과 출력의 크기가 동일 해야함
(Resnet에 사용되는 개념)

Transformer

배워야 할 개념

1. Input Embedding
2. Positional Encoding
3. Multi-Head Attention
4. Add & Norm
5. Feed Forward
6. Masked Multi-Head Attention

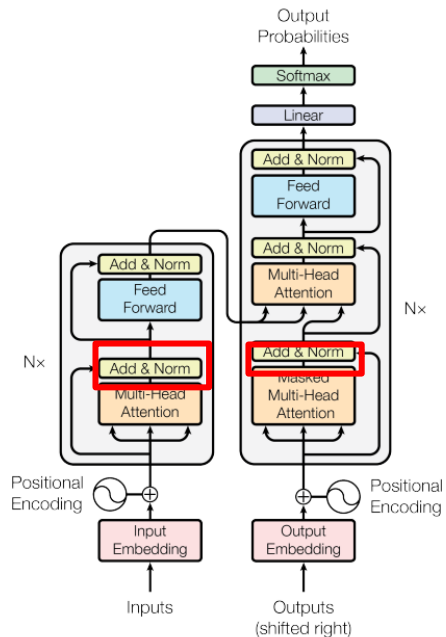
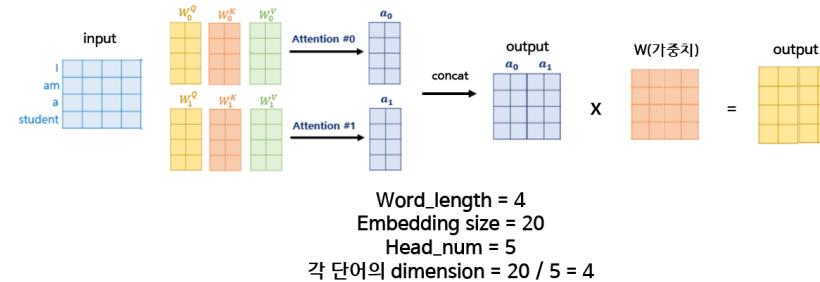


Figure 1: The Transformer - model architecture.

Add : Residual connection

의문) concatenate를 한 뒤 Fully connected layer에 왜 넣는가.

Concatenate를 한 뒤의 output size이 기존의 input size와 동일



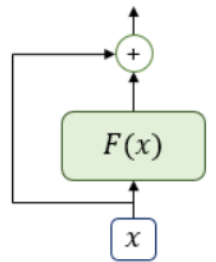
Output.shape = (4, 20) -> concatenate 했을 때 input과 output shape가 동일

Word_length = 4
Embedding size = 20
Head_num = 7
각 단어의 dimension = 20 / 7 = 2.xx

최종 Output.shape = (4, 14) -> concatenate 했을 때 input과 output shape가 다름 -> 뒤에서 다시 언급

Output을 낼 때 기존의 dimension에 대한 정보를 잃게 되는데 이를 해결하는 목적 또한 보입니다.

$$H(x) = x + F(x)$$



Transformer

배워야 할 개념

1. Input Embedding
2. Positional Encoding
3. Multi-Head Attention
4. Add & Norm
5. Feed Forward
6. Masked Multi-Head Attention

Add & Norm 이란 ?

Add : Residual connection

Norm : Layer Normalization

$$\hat{x}_{i,k} = \frac{x_{i,k} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

$$ln_i = \gamma \hat{x}_i + \beta = LayerNorm(x_i)$$

Layer Normalization : 정규화 + 학습가능한 파라미터 (감마, 베타)
각 단어의 차원을 각각 정규화를 해주어서,
학습을 더욱 잘 될 수 있도록 도와줍니다.

(감마의 초기 값 : 1, 베타의 초기 값 : 0)

$$LN = LayerNorm(x + Sublayer(x))$$

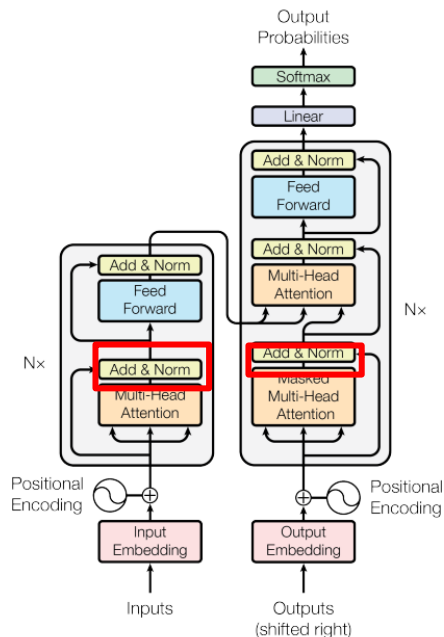


Figure 1: The Transformer - model architecture.

Transformer

배워야 할 개념

1. Input Embedding
2. Positional Encoding
3. Multi-Head Attention
4. Add & Norm
5. **Feed Forward**
6. Masked Multi-Head Attention

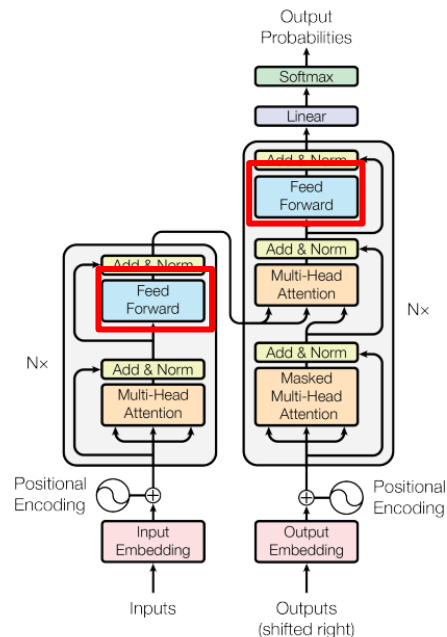


Figure 1: The Transformer - model architecture.

Feed Forward

3.3 Position-wise Feed-Forward Networks

In addition to attention sub-layers, each of the layers in our encoder and decoder contains a **fully connected feed-forward network**, which is applied to each position separately and identically. This consists of two linear transformations with a **ReLU** activation in between.

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \quad (2)$$

While the linear transformations are the same across different positions, they use different parameters from layer to layer. **Another way** of describing this is as **two convolutions with kernel size 1**. The dimensionality of input and output is $d_{\text{model}} = 512$, and the inner-layer has dimensionality $d_{\text{ff}} = 2048$.

Feed-Forward Network :

Fully connected layer를 두 개 사용한 네트워크 (activation : ReLU)

방법 : 2가지

1. Fully connected layer 2가지 사용
2. convolutions layer 2가지 사용

Transformer

배워야 할 개념

1. Input Embedding
2. Positional Encoding
3. Multi-Head Attention
4. Add & Norm
5. Feed Forward
6. **Masked Multi-Head Attention**

Masked Multi-Head Attention이란 ?

Decoder의 입력에서 사용되며,
기존의 Multi-Head Attention과 동일

하지만, Attention Score를 구한 뒤
미래의 정보를 참고하면 안되므로,
각 단어의 위치 이전 정보들만 제외하고, Masking을 걸어주는 Attention 입니다.

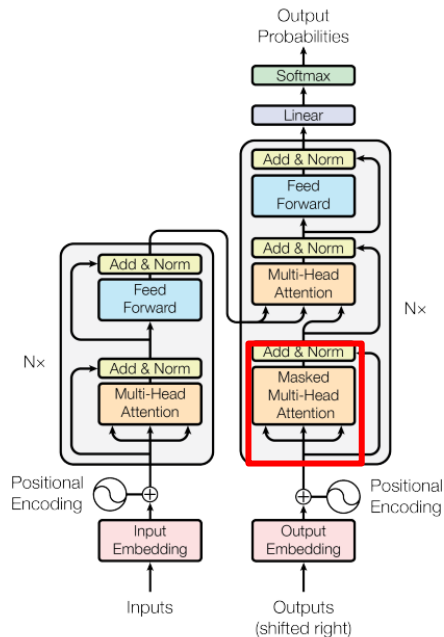
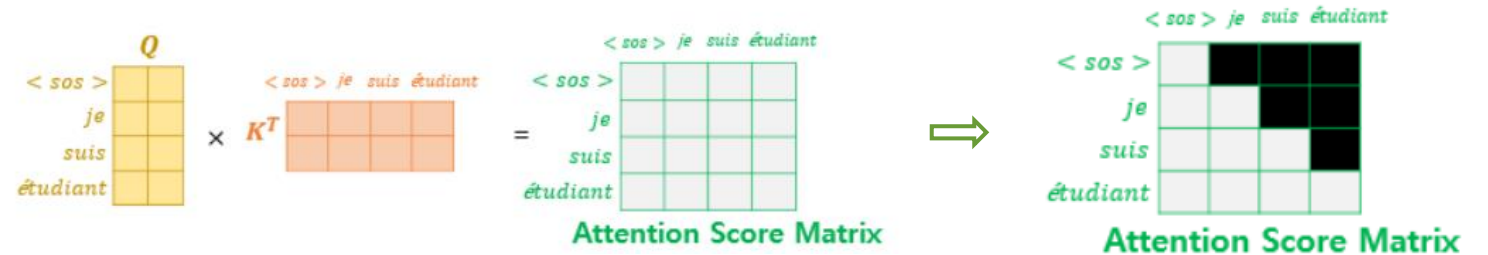


Figure 1: The Transformer - model architecture.



Transformer

배워야 할 개념

1. Input Embedding
2. Positional Encoding
3. Multi-Head Attention
4. Add & Norm
5. Feed Forward
6. Masked Multi-Head Attention

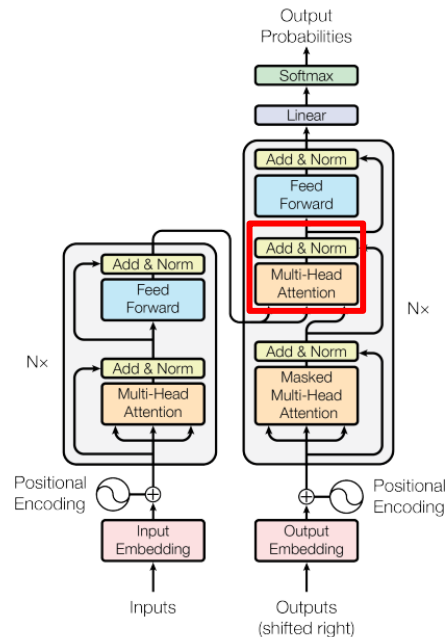


Figure 1: The Transformer - model architecture.

Encoder Decoder Attention :

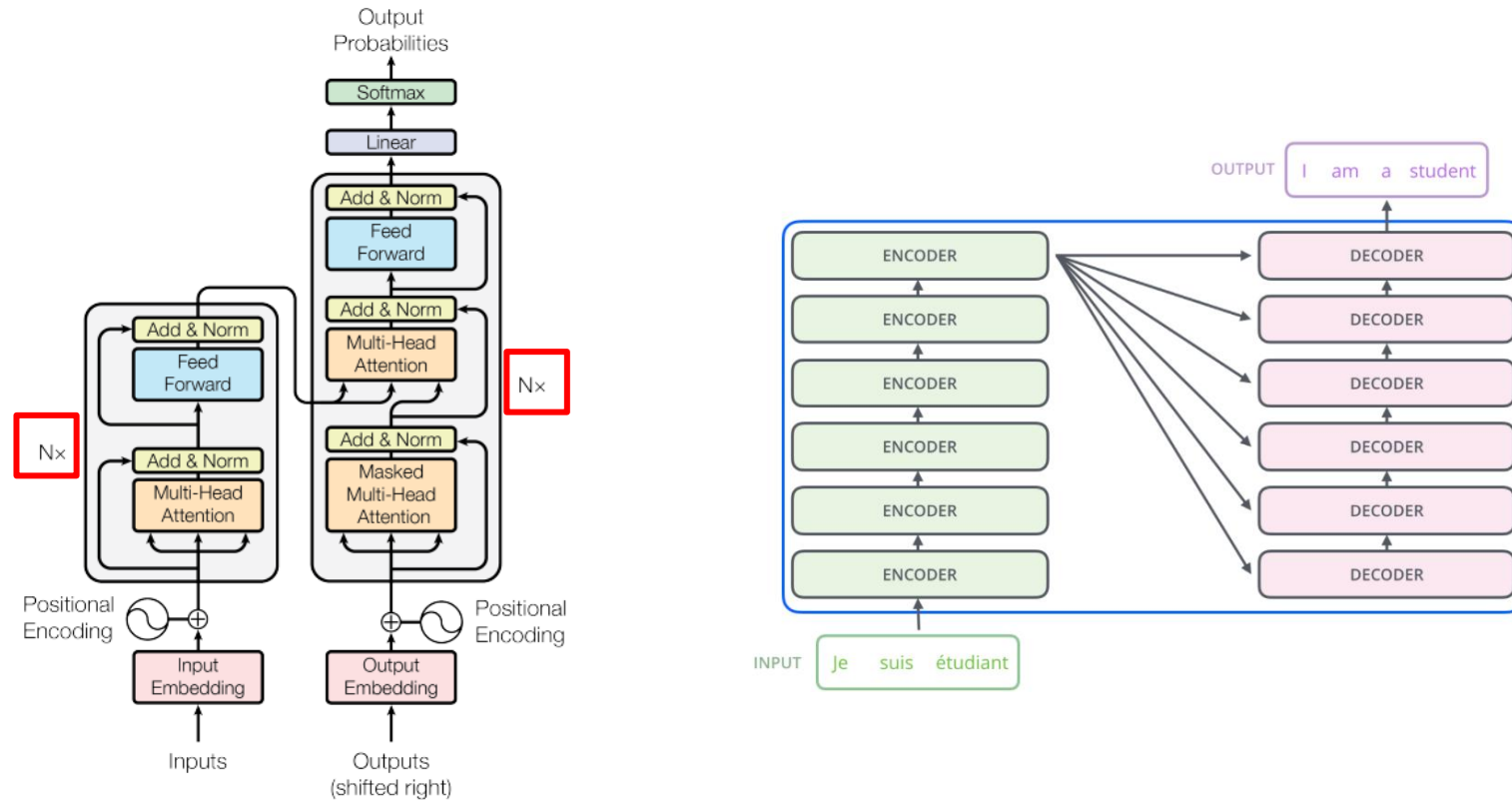
Encoder의 output 값과,
Masked Multi-Head Attention + Add&Norm 과정을 거친 output을

Attention 하는 과정입니다.

Query와 Key가 같지 않아서, Self Attention은 아니지만,
연산과정은 Self Attention과 동일하게 진행합니다.

$$\begin{matrix} & Q \\ \begin{matrix} < sos > \\ je \\ suis \\ \text{étudiant} \end{matrix} & \times K^T & \begin{matrix} I & am & a & student \end{matrix} \\ & & \end{matrix} = \begin{matrix} & I & am & a & student \\ \begin{matrix} < sos > \\ je \\ suis \\ \text{étudiant} \end{matrix} & \begin{matrix} & & & & \end{matrix} \\ & \text{Attention Score Matrix} \end{matrix}$$

Transformer



Transformer encoder, decoder 특징 :

입력과 출력의 차원이 항상 동일하므로,
인코더 및 디코더를 여러 개 연결하여 결과 값을 낼 수 있습니다.

논문에서는 총 6개의 encoder와 decoder를 사용하였습니다.

Transformer

Transformer의 발전에 관한 review 논문 : A Survey of Transformers

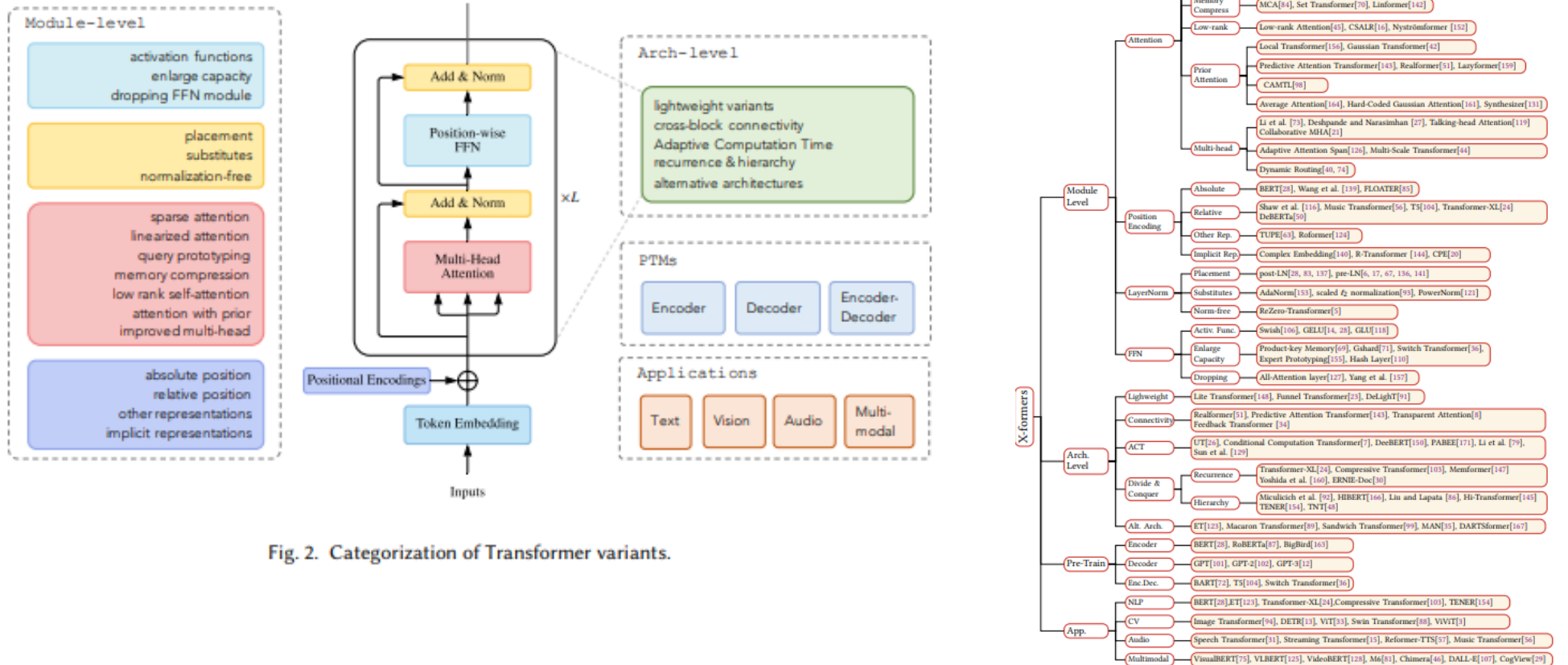


Fig. 2. Categorization of Transformer variants.

Stock Prediction

Research Track Paper

KDD '21, August 14–18, 2021, Virtual Event, Singapore

Accurate Multivariate Stock Movement Prediction via Data-Axis Transformer with Multi-Level Contexts

Jaemin Yoo
Seoul National University
Seoul, South Korea
jaeminyoo@snu.ac.kr

Yejun Soun
Seoul National University
DeepTrade Inc.
Seoul, South Korea
sony7819@snu.ac.kr

Yong-chan Park
Seoul National University
Seoul, South Korea
wjdakf3948@snu.ac.kr

U Kang
Seoul National University
DeepTrade Inc.
Seoul, South Korea
ukang@snu.ac.kr

Stock Prediction

1 INTRODUCTION

How can we efficiently correlate multiple stocks for accurate stock movement prediction? Stock movement prediction is one of the core applications of financial data mining, which has received growing interest in data mining and machine learning communities due to its substantial impact on financial markets [9, 25, 31]. The problem is to predict the movement (rise or fall) of stock prices at a future moment. The potential of the problem is unquestionable, as accurate predictions can lead to the enormous profit of investment.

It is challenging to achieve high accuracy of stock movement prediction, since stock prices are inherently random; no clear patterns exist unlike in other time series such as temperature or traffic. On the other hand, most stocks can be clustered as sectors by the industries that they belong to [3]. Stocks in the same sector share a similar trend even though their prices are perturbed randomly in a short-term manner, and such correlations can be a reliable evidence for investors. For instance, one can buy or sell a stock considering the prices of other stocks in the same sector, based on the belief that their movements will coincide in the future.

주가의 움직임을 높은 확률로 예측하는 것이 목표이며,

그 근거로 사용되는 조건 중 1개는

대부분의 주가들은 산업에 따라 Sector가 구분되며,
움직임을 기간에 따라 구분을 두면 Short-term은 random하게 움직이지만,
Long term은 결국 Sector의 트렌드에 따라가는 것을 볼 수 있습니다.

Stock Prediction

Most previous works that utilize the correlations between stocks rely on pre-defined lists of sectors [15, 18]. However, using a fixed list of sectors makes the following limitations. First, one loses the dynamic property of stock correlations that naturally change over time, especially when training data span over a long period. Second, a prediction model cannot be applied to stocks that have no information of sectors or whose sectors are ambiguous. Third, the performance of predictions relies heavily on the quality of sector information rather than the ability of a prediction model.

그동안의 상관관계를 이용하여 주가를 예측하는 모델들의 단점을 기술

1. 정해진 섹터만 가능하며, 장기간 학습할 경우 동적인 속성을 잃게 됩니다.
(장기간 학습 시 예측 데이터는 학습했던 데이터와 특징이 달라진 데이터로 변경)
2. 업종이 매매하면 섹터의 고정이 불가능합니다.
(일정 섹터의 주식들만 예측이 가능)
3. 예측의 성능은 모델의 성능보다는 섹터의 정보에 크게 의존합니다.
(해당 데이터로 다른 모델에 넣어도 비슷한 성능을 냄)

Stock Prediction

In this work, we design an end-to-end framework that learns the correlations between stocks for accurate stock movement prediction; the quality of correlations is measured by how much it contributes to improving the prediction accuracy. Specifically, we aim to address the following challenges that arise from the properties of stock prices. First, multiple features at each time step should be used together to capture accurate correlations. For instance, the variability of stock prices in a day can be given to a model by including both the highest and lowest prices as features. Second, the global movement of a market should also be considered, since the relationship between stocks is determined not only by their local movements, but also by the global trend. Third, the learned correlations should be asymmetric, reflecting the different degrees of information diffusion in a market.

데이터들의 상관관계를 이용하여, 주가의 움직임을 예측하는 모델을 제안합니다.

상관관계의 기준 : 예측 정확도의 향상에 얼마만큼 기여하는지

1. 주가의 단일 특성(종가)만을 사용하는 것이 아닌, 고가, 저가 등의 다른 특성을 같이 입력합니다.
2. 종목 (local trend) 만을 학습하는 것이 아닌, 종목들의 전체 트렌드(global trend)를 같이 학습
Ex) 삼성전자, 하이닉스 등의 데이터 + 지수(코스피 100) 데이터도 입력
3. Global trend가 끼치는 영향이 각 종목마다 다르기 때문에, 반영 정도를 다르게 학습 시켜야 합니다.

Stock Prediction

We propose DTML (Data-axis Transformer with Multi-Level contexts), an end-to-end framework that automatically correlates multiple stocks for accurate stock movement prediction. The main ideas of DTML are as follows. First, DTML generates each stock's

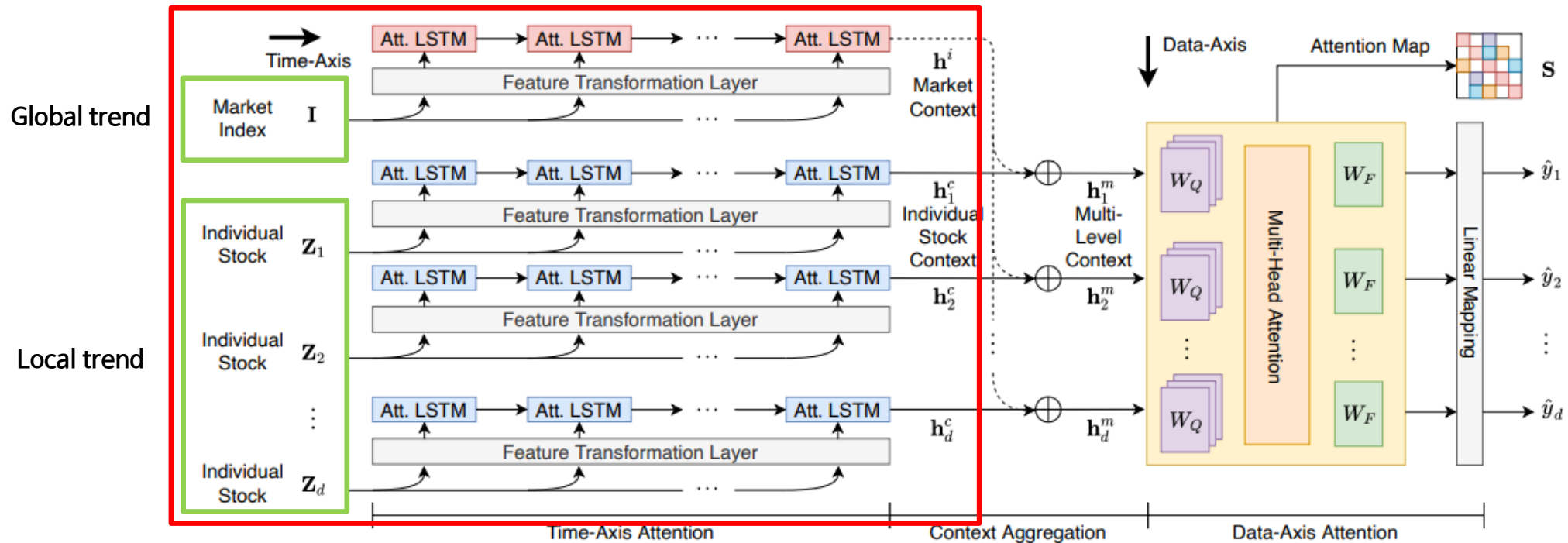
comprehensive context vector that summarizes multivariate historical prices by temporal attention. Second, DTML extends the generated contexts into multi-level by combining it with the global movement of the market. Third, DTML learns asymmetric and dynamic attention scores from the multi-level contexts using the transformer encoder, which calculates different query and key vectors for multi-head attention between stocks.

여기서 제안하는 모델 DTML의 경우
여러가지의 데이터를 가지고 와서, 각 데이터 간의 상관관계를 체크 한 뒤 예측을 실시합니다.

메인 아이디어 :

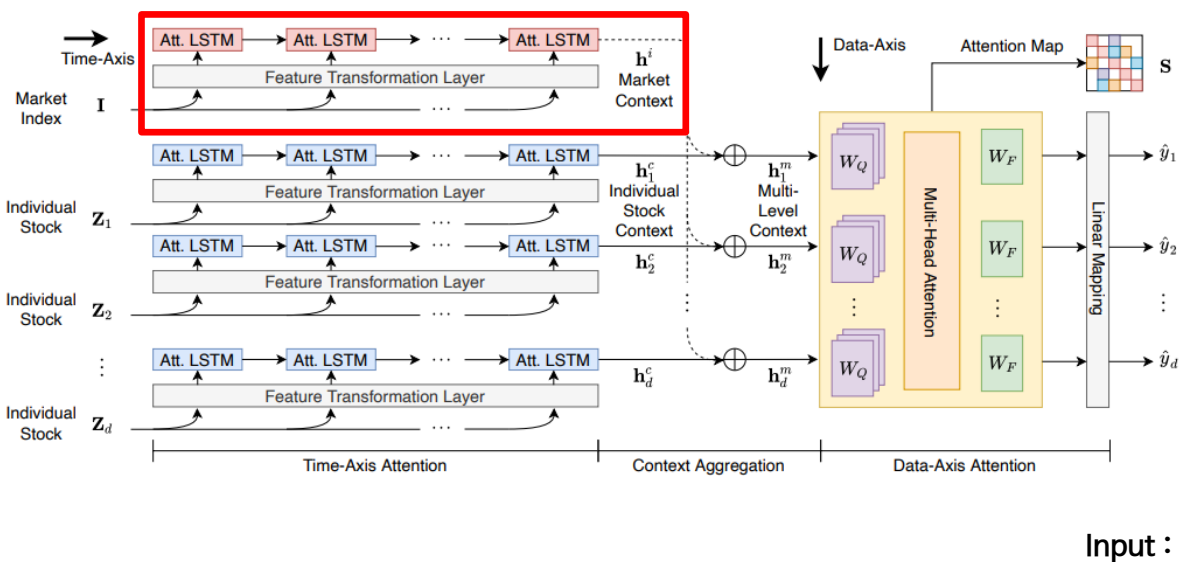
1. 주가 종목의 과거 데이터를 이용하여, 이를 요약한 context vector 만들기 (LSTM -Attention)
2. Multi - level context vector 만들기
(Multi - level 이란 : local context vector + global context vector)
3. 위에서 얻은 Multi-level context vector가 입력으로 사용되어,
Transformer encoder구조를 사용한 동적인 Attention score를 만들어서 입력을 예측하고,
마지막에 Fully connected layer(activation = sigmoid)사용하여, 상승 및 하락 예측

Stock Prediction



메인 아이디어 : 1. 주가 종목의 과거 데이터를 이용하여, 이를 요약한 context vector 만들기(LSTM -Attention)

Stock Prediction



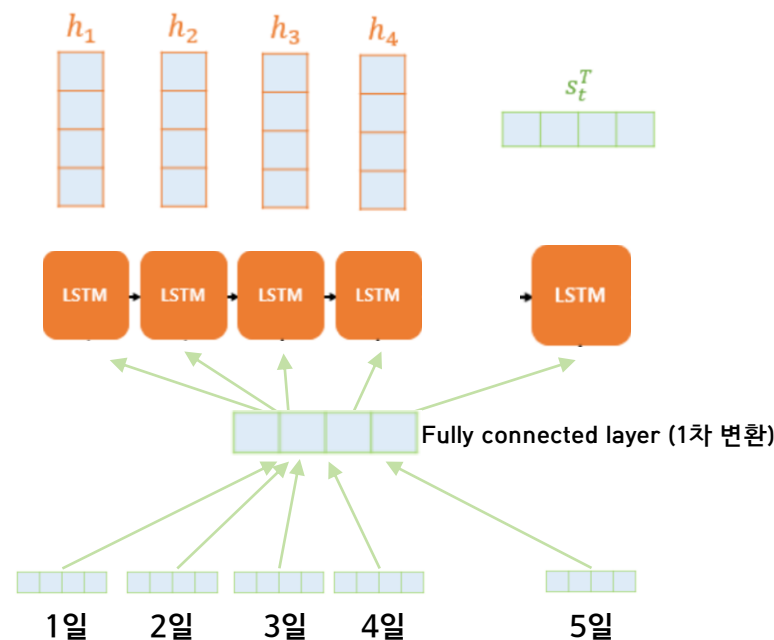
Fully connected layer

Input 데이터를 그대로 사용하지 않고,
Fully connected layer에 한번 넣어서 사용한다.

-> LSTM Layer에 입력으로 들어가기 전에

복잡성을 늘리지 않으면서,

학습을 더욱 잘되게 할 수 있기 때문에 사용됨



Attention (Dot-product Attention(루 웡 어텐션))

Qurey : s_t (제일 마지막 입력된 5일째의 데이터)

Key : h_1, h_2, h_3, h_4

제일 마지막으로 입력되는 데이터가 Qurey로 설정됩니다.
가장 많은 과거 데이터를 함축하고 있기 때문

각 주식 종목마다 context vector를 만들게 됩니다.

$$\begin{aligned}
 & \text{5일, 1일} \quad s_t^T \times h_1 = \text{Score1} \\
 & \text{5일, 2일} \quad s_t^T \times h_2 = \text{Score2} \\
 & \text{5일, 3일} \quad s_t^T \times h_3 = \text{Score3} \\
 & \text{5일, 4일} \quad s_t^T \times h_4 = \text{Score4}
 \end{aligned}$$

Stock Prediction

개별 주식(주가) : Local context

시장의 흐름(지수) : global context

-> Layer Normalization

각각의 context vector 마다

Layer Normalization을 시도한다.

-> 각각의 주식들은 실제 가격이
정규화 되어있지 않기 때문에,
정규화를 해주는 과정이 필요

Ex) 삼성전자 range : 40000 ~ 90000

하이닉스 range : 80000 ~ 150000

...

Context Normalization. The context vectors generated by the attention LSTM have values of diverse ranges, because each stock has its own range of features and pattern of historical prices. Such diversity makes subsequent modules unstable, such as the multi-level context aggregation (Section 3.3) or data-axis self-attention (Section 3.4). We thus introduce a context normalization, which is a variant of the layer normalization [1]:

$$h_{ui}^c = \gamma_{ui} \frac{\tilde{h}_{ui}^c - \text{mean}(\tilde{h}_{ui}^c)}{\text{std}(\tilde{h}_{ui}^c)} + \beta_{ui}, \quad (3)$$

where i is the index of an element in a context vector, $\text{mean}(\cdot)$ and $\text{std}(\cdot)$ are computed for all stocks and elements, and γ_{ui} and β_{ui} are learnable parameters for each pair (u, i) .

Norm : Layer Normalization

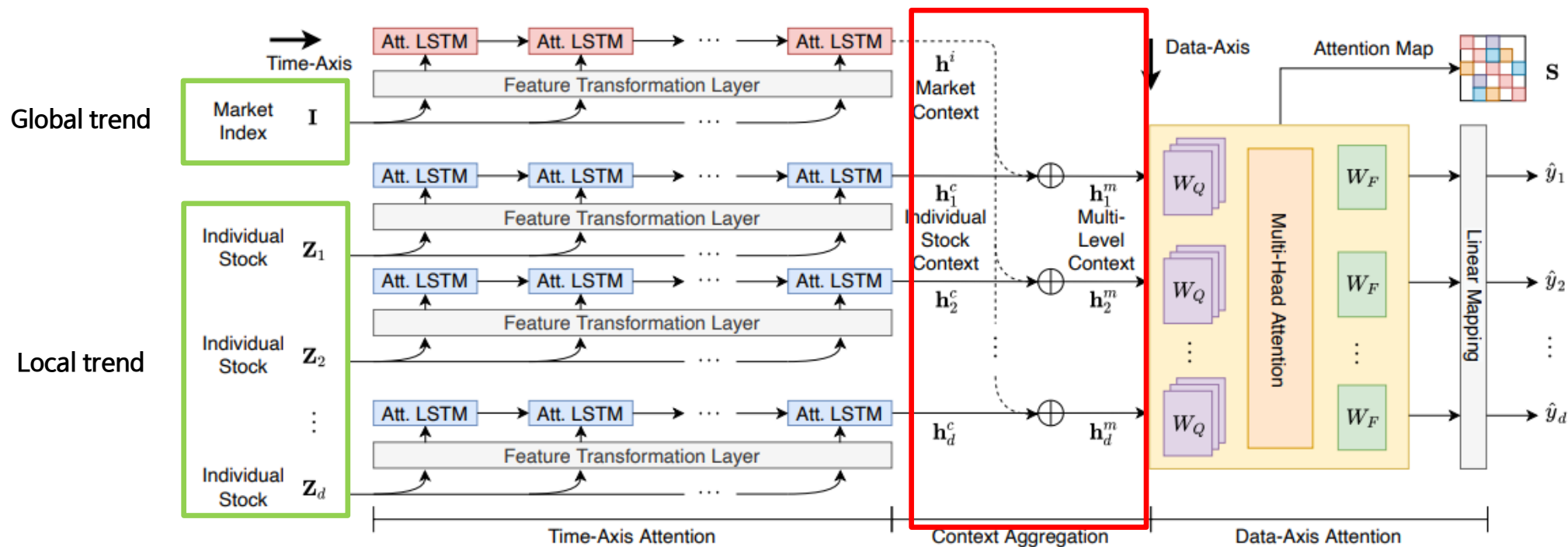
$$\hat{x}_{i,k} = \frac{x_{i,k} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

$$ln_i = \gamma \hat{x}_i + \beta = \text{LayerNorm}(x_i)$$

Layer Normalization : 정규화 + 학습가능한 파라미터 (감마, 베타)
각 단어의 차원을 각각 정규화를 해주어서,
학습을 더욱 잘 될 수 있도록 도와줍니다.

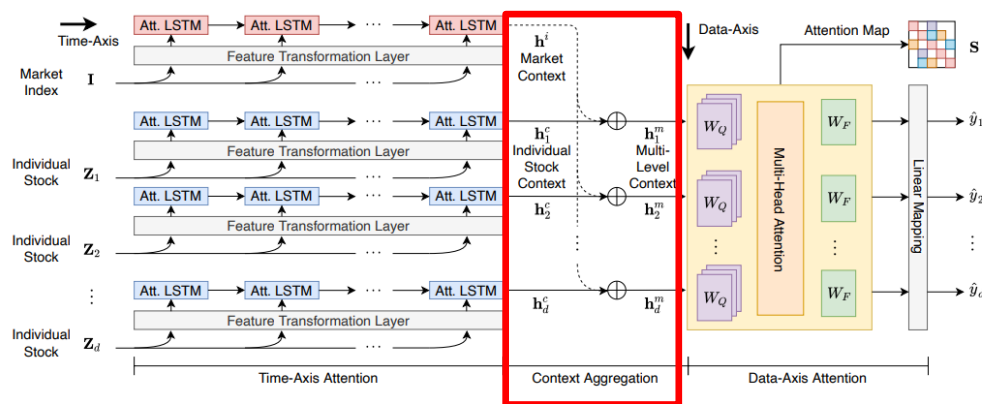
(감마의 초기 값 : 1, 베타의 초기 값 : 0)

Stock Prediction



메인 아이디어 : 2. Multi - level context vector 만들기
(Multi - level 이란 : local context vector + global context vector)

Stock Prediction



Multi-Level Contexts. Then, we generate a multi-level context h_u^m for each stock u by using the global market context h^i as base knowledge of all correlations:

$$h_u^m = h_u^c + \beta h^i, \quad (4)$$

where β is a hyperparameter that determines the weight of h^i . As a result, the correlation between two stocks is determined not only by their local movements, but also by the relationship to the global market context h^i .

메인 아이디어 : 2. Multi – level context vector 만들기
(Multi – level 이란 : local context vector + global context vector)

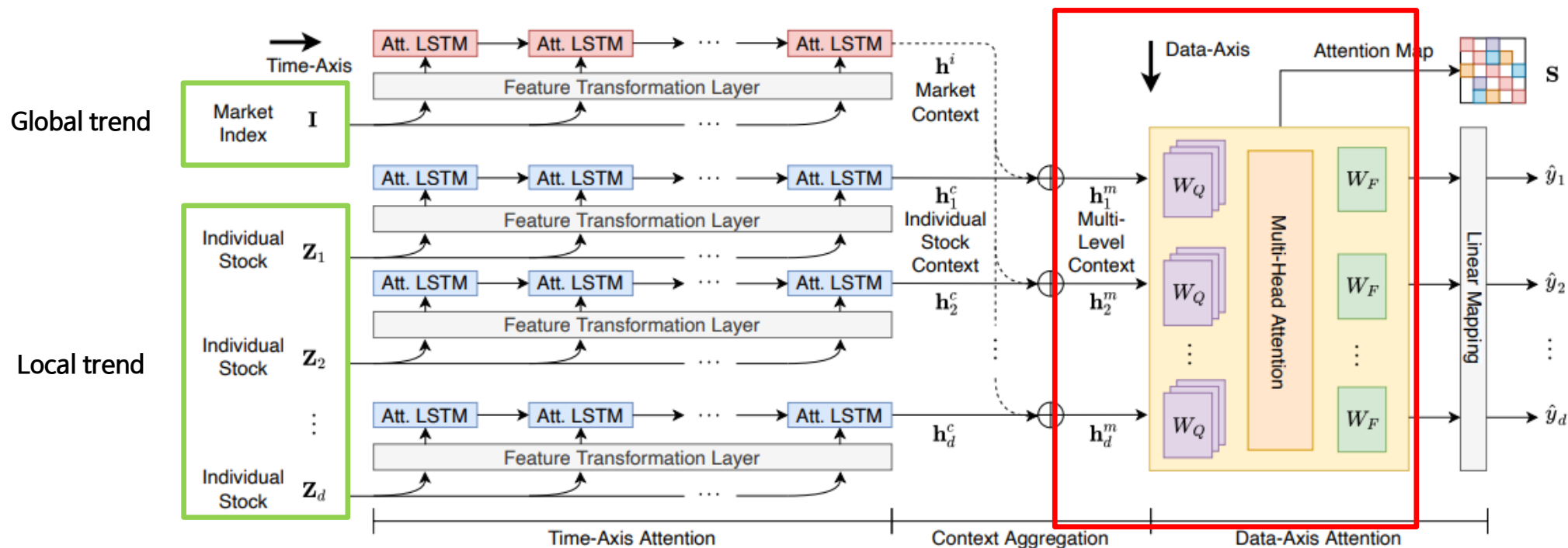
$$h_u^m = h_u^c + \beta h^i,$$

Multi-context vector = local context vector + beta * global context vector

Beta : hyperparameter로 결정하며,

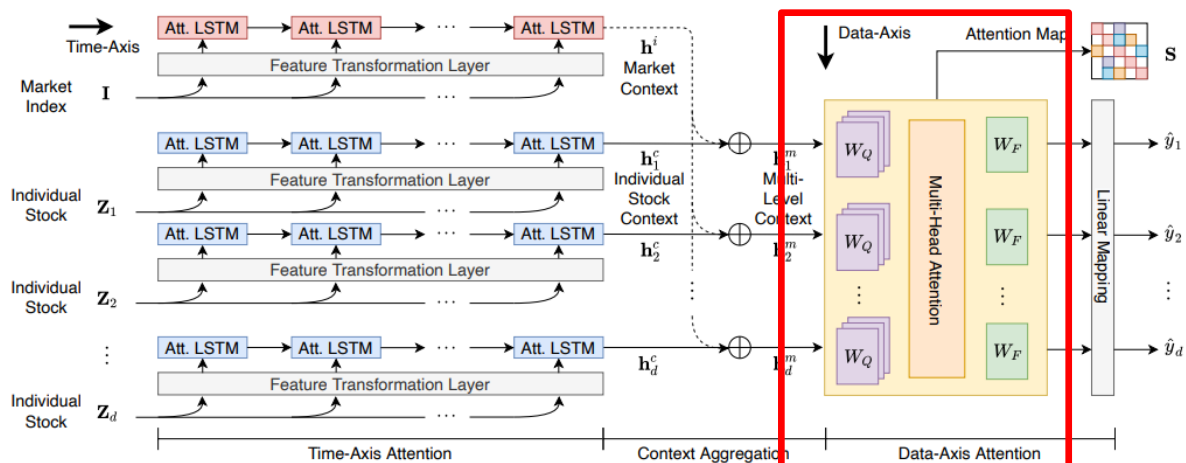
Beta의 비율에 따라서, 얼마나 시장의 흐름(global context vector)을 반영할 것인지 결정한다고 합니다.

Stock Prediction



메인 아이디어 : 3. 위에서 얻은 Multi-level context vector가 입력으로 사용되어, Transformer encoder구조를 사용하며 내부에 있는 Multi head Attention을 통해 개별 주식들의 상관관계를 파악하고, Feed-Forward Network를 통해서 output을 만듭니다.

Stock Prediction

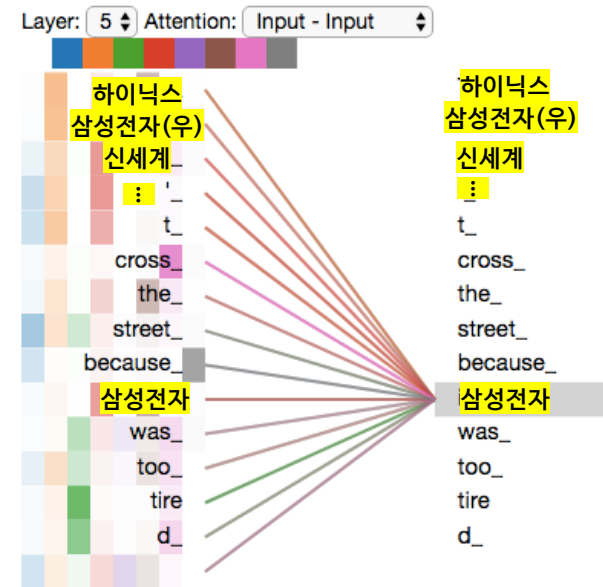


Self-Attention. To apply the self-attention with respect to the data-axis, we first build a multi-level context matrix $\mathbf{H} \in \mathbb{R}^{d \times h}$ by stacking $\{\mathbf{h}_u^m\}_u$ for $u \in [1, d]$, where d is the number of stocks and h is the length of context vectors. We then generate query, key, and value matrices by learnable weights of size $h \times h$:

$$\mathbf{Q} = \mathbf{H}\mathbf{W}_q \quad \mathbf{K} = \mathbf{H}\mathbf{W}_k \quad \mathbf{V} = \mathbf{H}\mathbf{W}_v. \quad (6)$$

Then, we compute the attention scores from the query and key vectors, and aggregate the value vectors as follows:

$$\tilde{\mathbf{H}} = \mathbf{S}\mathbf{V} \quad \text{where} \quad \mathbf{S} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{h}}\right). \quad (7)$$

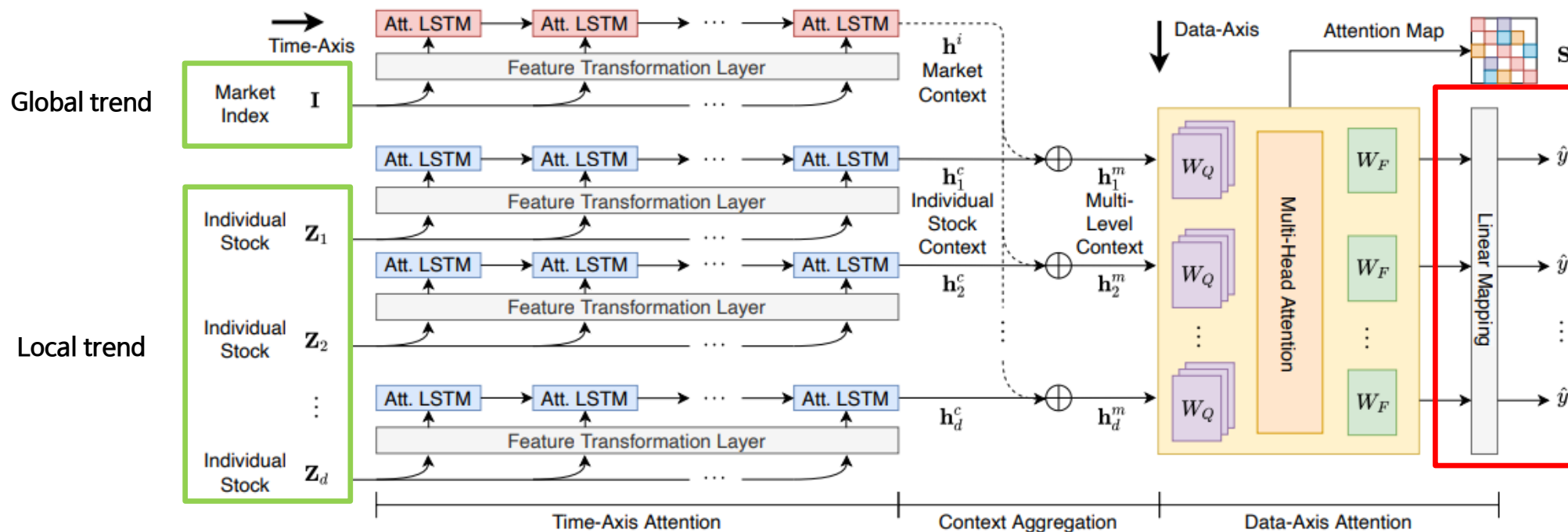


Nonlinear Transformation. We update the aggregated contexts with residual connections as follows:

$$\mathbf{H}_p = \tanh(\mathbf{H} + \tilde{\mathbf{H}} + \text{MLP}(\mathbf{H} + \tilde{\mathbf{H}})), \quad (8)$$

Transformer의 encoder 구조와 완벽히 동일한 것을 알 수 있음

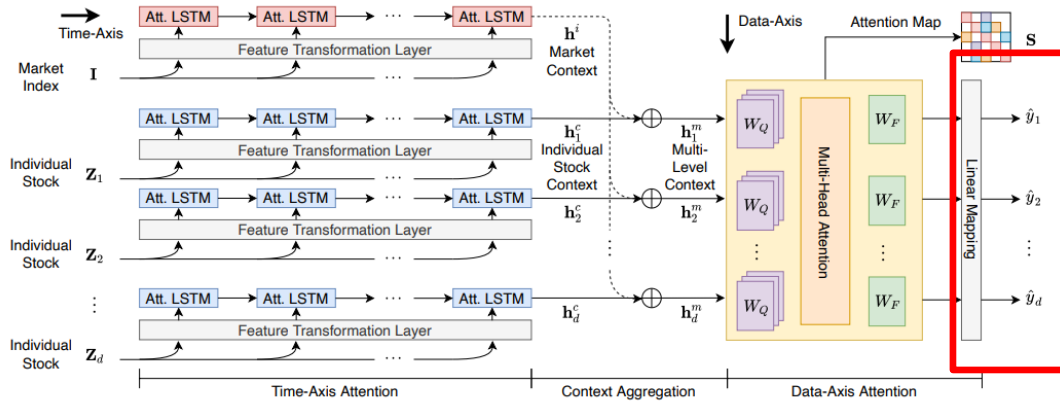
Stock Prediction



최종 Output :

Transformer encode의 output은 결국 입력의 크기와 동일하게 바뀌며, 해당 output을 최종적으로 **이진 분류**를 통해, 상승 및 하락을 예측합니다.

Stock Prediction



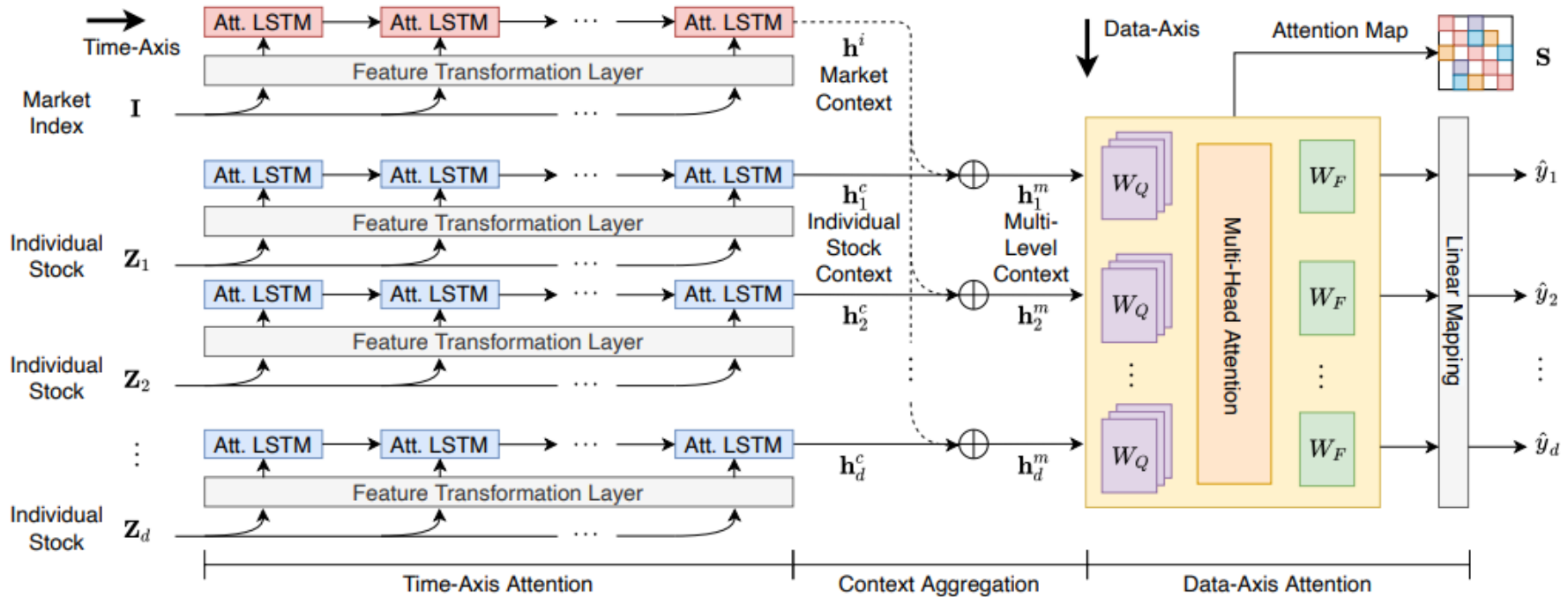
Final Prediction. We lastly apply a single linear layer to the transformed contexts to produce the final predictions as

$$\hat{y} = \sigma(H_p W_p + b_p). \quad (9)$$

We apply the logistic sigmoid function σ to interpret each element \hat{y}_u for stock u as a probability and use it directly as the output of DTML for stock movement prediction.

Sigmoid : 이진 분류 activation function으로 사용되며, 입력 값을 0 ~ 1사이로 결정
0.5이상의 결과가 나온다면 1, 0.5미만의 결과가 나온다면 0으로 해석합니다.

Stock Prediction



최종 정리 : 데이터가 예측되는 범위는 코스피 100이라고 가정.

코스피 100에서의 기업 중 선별된 기업 전부를
상승 혹은 하락 예측 (선별의 기준점은 나와있지 않음, 개수만 나옴)

Stock Prediction

Hyperparameters. We search the hyperparameters of DTML as follows: the window size w in $\{10, 15\}$, the market context weight β in $\{0.01, 0.1, 1\}$, the hidden layer size h in $\{64, 128\}$, the number of epochs in $\{100, 200\}$, and the learning rate in $\{0.001, 0.0001\}$. We set the strength λ of selective regularization to 1 and the dropout rate to 0.15. We use the Adam optimizer [14] for the training with the early stopping by the validation accuracy. For competitors, we use the default settings in their public implementations.

하루를 예측할 때 window size = 10 or 15 / global context의 반영 비율 (0.01, 0.1, 1)

Model	ACL18 (US)		KDD17 (US)		NDX100 (US)	
	ACC	MCC	ACC	MCC	ACC	MCC
LSTM [24]	0.4987 ± 0.0127	0.0337 ± 0.0398	0.5118 ± 0.0066	0.0187 ± 0.0110	0.5263 ± 0.0003	0.0037 ± 0.0049
ALSTM [31]	0.4919 ± 0.0142	0.0142 ± 0.0275	0.5166 ± 0.0041	0.0316 ± 0.0119	0.5260 ± 0.0007	0.0028 ± 0.0084
StockNet [31]	0.5285 ± 0.0020	0.0187 ± 0.0011	0.5193 ± 0.0001	0.0335 ± 0.0050	0.5392 ± 0.0016	0.0253 ± 0.0102
Adv-ALSTM [9]	0.5380 ± 0.0177	0.0830 ± 0.0353	0.5169 ± 0.0058	0.0333 ± 0.0137	0.5404 ± 0.0003	0.0046 ± 0.0090
DTML (proposed)	0.5744 ± 0.0194	0.1910 ± 0.0315	0.5353 ± 0.0075	0.0733 ± 0.0195	0.5406 ± 0.0037	0.0310 ± 0.0193

Model	CSI300 (China)		NI225 (Japan)		FTSE100 (UK)	
	ACC	MCC	ACC	MCC	ACC	MCC
LSTM [24]	0.5367 ± 0.0038	0.0722 ± 0.0050	0.5079 ± 0.0079	0.0148 ± 0.0162	0.5096 ± 0.0065	0.0187 ± 0.0129
ALSTM [31]	0.5315 ± 0.0036	0.0625 ± 0.0076	0.5060 ± 0.0066	0.0125 ± 0.0139	0.5106 ± 0.0038	0.0231 ± 0.0077
StockNet [31]	0.5254 ± 0.0029	0.0445 ± 0.0117	0.5015 ± 0.0054	0.0050 ± 0.0118	0.5036 ± 0.0095	0.0134 ± 0.0135
Adv-ALSTM [9]	0.5337 ± 0.0050	0.0668 ± 0.0084	0.5160 ± 0.0103	0.0340 ± 0.0201	0.5066 ± 0.0067	0.0155 ± 0.0140
DTML (proposed)	0.5442 ± 0.0035	0.0826 ± 0.0074	0.5276 ± 0.0103	0.0626 ± 0.0230	0.5208 ± 0.0121	0.0502 ± 0.0214

최종 결과

출처

A Survey of Transformers : <https://arxiv.org/pdf/2106.04554.pdf>

Attention Is All You Need : <https://arxiv.org/pdf/1706.03762.pdf>

Accurate Multivariate Stock Movement Prediction via Data-Axis Transformer with Multi-Level Contexts:
<https://dl.acm.org/doi/pdf/10.1145/3447548.3467297>

[그림출처 :](#)

[Jalammar : https://jalammar.github.io/illustrated-transformer/](https://jalammar.github.io/illustrated-transformer/)

딥러닝으로 배우는 자연어 처리 입문 : <https://wikidocs.net/31379>