Attention Is All You Need

Translation Example

JaeUk Lee

SeongJun Hong

HONGIK UNIVERSITY
HIGH-PERFORMANCE DATA PROCESSING & ANALYSIS LAB

Contents

1. Background : 데이터 셋 Tokenizer

Tokenization

- Character 토큰화 :

- Word Tokenization : 토큰의 기준을 단어로 하는 경우, 다만, 여기서 단어(word)는 단어 단위 외에도 단어구, 의미를 갖는 문자열로도 간주되기도 한다. 주로 띄어쓰기를 기준으로 토큰화를 한다.
- Time is an illusion. Lunchtime double so!
- "Time", "is", "an", "illustion", "Lunchtime", "double", "so"

1. Background : 데이터 셋 Tokenizer

BPE Tokenization

- SubWord Tokenization : 하나의 단어를 여러 서브워드로 분리해서 단어를 인코딩 및 임베딩하겠다는 의도를 가진 전처리 작업이다. 예시로 논문에서 언급된 BPE(Byte Pair Encoding)에 대해 간단히 소개한다.

- BPE(Byte-Pair Encoding)은 자연어 처리에서 sub word 분리 알고리즘으로 사용된다.
- character 단위에서 점차적으로 단어 집합을 만들어 내는 Bottom up 방식이다.
- 이는 학습되지 않은 단어들에 대응하지 못하는 OOV 문제를 해결 할 수 있는 방안이다.

1. Background : 데이터 셋 Tokenizer

BPE Tokenization

- BPE(Byte Pair Encoding) 알고리즘 과정

1. 기준이 되는 문서를 character 단위로 분리한다.
2. Character 단위로 분리했을 때 , 가장 많이 등장하는 bi-gram pairs를 찾아서 merge한다.
3. vocabulary에 새로 merge한 토큰을 업데이트한다.
3. Iteration을 정해놓고 주어진 횟수만큼 반복 수행한다.

초기 구성 dictionary & vocabulary

```
# dictionary
l o w : 5,  l o w e r : 2,  n e w e s t : 6,  w i d e s t : 3
```

```
# vocabulary
l, o, w, e, r, n, w, s, t, i, d
```

1회 Iteration

```
# dictionary update
l o w : 5,
l o w e r : 2,
n e w e s t : 6,
w i d e s t : 3

# vocabulary update
l, o, w, e, r, n, w, s, t, i, d, es
```

2회 Iteration

```
# dictionary update
l o w : 5,
l o w e r : 2,
n e w est : 6,
w i d est : 3

# vocabulary update
l, o, w, e, r, n, w, s, t, i, d, es, est
```

1. Background : 데이터 셋 Tokenizer

BPE Tokenization

- 같은 방법으로 10회 진행했을 때, 다음과 같다.
- Iteration을 정할 수 있고, vocab_size를 정할 수 있어서 제한된 크기의 vocabulary를 만들 때 유용하다.
- 또한 기존에는 OOV(Out of Vocabulary)에 해당 했던 단어가 BPE 알고리즘을 수행 후 더 이상 OOV가 아니게 되는 상황이 발생한다.
- 예를 들면 lowest라는 단어는 low, est로 인코딩 될 수 있으며 둘 다 vocabulary에 있는 토큰이다.

10회 Iteration

```
# dictionary update
low : 5,
low e r : 2,
newest : 6,
widest : 3

# vocabulary update
l, o, w, e, r, n, w, s, t, i, d, es, est, lo, low, ne, new, newest, wi, wid, widest
```

1. Background - 데이터 셋 Tokenizer

형태소 Tokenization

- 한국어는 어절이 독립적인 단어로 구성되는 것이 아니라 조사 등의 무언가가 붙어있는 경우가 많아서 띄어쓰기 단위로 토큰화를 하기보다는 이를 전부 분리해줘야 한다

- 이를 이해하기 위해 '형태소'의 개념을 이용한다.

- 자립 형태소 : 접사, 어미, 조사와 상관없이 자립하여 사용할 수 있는 형태소. 그 자체로 단어가 된다. 체언(명사, 대명사, 수사), 수식언(관형사, 부사), 감탄사 등이 있다.

- 의존 형태소 : 다른 형태소와 결합하여 사용되는 형태소. 접사, 어미, 조사, 어간를 말한다.

1. Background - 데이터 셋 Tokenizer

형태소 Tokenization

- 한국어를 토큰화하는 KONLPY 라이브러리에서는 형태소 단위로 단어를 분류하는 함수를 제공한다.
- 여러 함수 중 하나를 선택해서 토큰화에 사용할 수 있다.
- 이번 예제에서는 Okt를 이용해서 분류한 토큰화 방식을 사용하였다.

In [1]:
```
#Open Korea Text
from konlpy.tag import Okt

okt = Okt()
tokens = okt.morphs("나는 학생이다.")
print(tokens)
```

```
['나', '는', '학생', '이다', '.']
```

In [2]:
```
#Komoran
from konlpy.tag import Komoran
komoran=Komoran()
text = "나는 학생이다."
print(komoran.morphs(text))
```

```
['나', '는', '학생', '이', '다', '.']
```

2. Translation

HyperParameter 설정

- 논문에서 사용된 HyperParameter
- seq_len : 문장의 길이(문장에 포함된 토큰의 개수)
- d_model : 임베딩 벡터의 차원
- h : Attention을 병렬로 처리할 Head의 수
- N : 인코더 및 디코더의 Layer의 수
- d_ff : FFN에서 차원을 확장할 때 사용하는 변수

2. Translation

HyperParameter 설정

- I am a student -> ‘나는 학생이다’ 과정을 최대한 자세하게 다루면서 visualization할 수 있도록 HyperParameter를 줄여서 설정했다.

- Input Sequence : I am a student.
- Output Sequence : 나는 학생이다.
- d_model = 4
- d_ff = 16
- h = 2
- d_k = 4/2 = 2
- d_v = 4/2 = 2
- src_maxlen = 6 // PAD 마스킹을 설명하기 위해서 문장의 길이보다 길게 설정한다.
- trg_maxlen = 6

2. Translation 전처리 과정 - 데이터 셋 Tokenizer

Tokenization

- 예제로 사용할 src 과 trg 의 토큰화를 해준다면 다음과 같다.
- I, am, a, student, <PAD>, <PAD> // Word Tokenization 방식 사용
- <BOS>, '나', '는', '학생', '이다', <PAD>, <PAD>, <EOS> // Okt를 이용한 형태소 토큰화

2. Translation Encoder - Embedding

Embedding

- 문장을 구성하는 각각의 토큰은 그에 상응하는 정수 인코딩 값에 매칭이 되고,
  각각의 토큰 인덱스들은 Look-up 테이블로부터 d_model 차원의 임베딩 벡터를 생성할 수 있다.

- look-up table에 위치하는 임베딩 벡터는 학습과정에서 가중치가 업데이트 되는것과 같은 방식으로 훈련된다.

Token	Integer	LookUp Table			
<pad>	0	....	....	....	....
I	1	0.1	0.6	0.7	0.4
student	2	....	....	....	....
am	3	....	....	....	....
a	4	....	....	....	....
....	.....	....	....	....	....

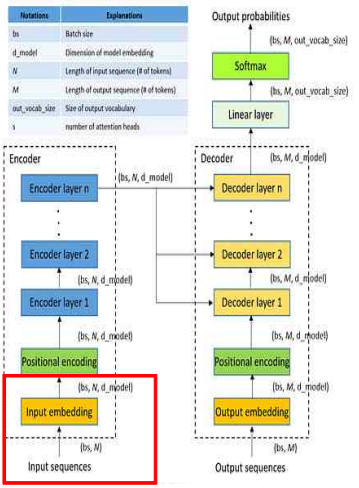
Embedding Vecor			
0.1	0.6	0.7	0.4

2. Translation Encoder – Positional Encoding

Input Embedding

- 입력 문장 I am a student 은 'I', 'am', 'a', 'student', <PAD>, <PAD> 로 토큰화되고, 문장의 길이는 토큰의 개수인 6, 임베딩 벡터의 차원은 d_model이다.
  따라서, Input Embedding 는 (6, 4) 벡터로 만들어진다.
- 위치정보를 표현하기 위해 Positional Encoding과 더해 사용한다.

I	0.1	0.6	0.7	0.4
am	. . . .	. . . .	. . . .	. . . .
a	. . . .	. . . .	. . . .	. . . .
student	. . . .	. . . .	. . . .	. . . .
<PAD>	. . . .	. . . .	. . . .	. . . .
<PAD>	. . . .	. . . .	. . . .	. . . .



2. Translation

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

Positional Encoding

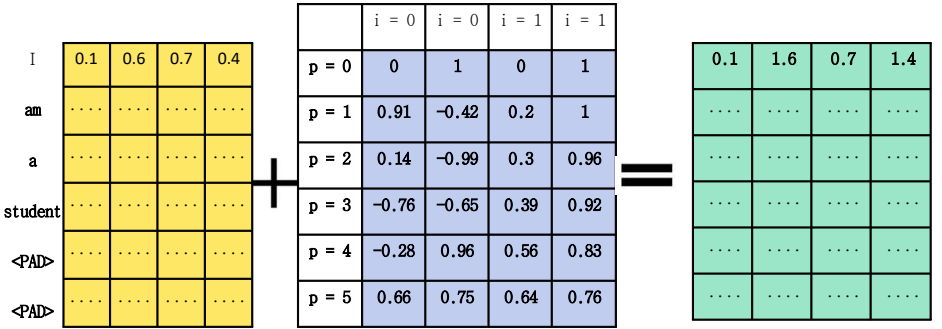
- 트랜스포머는 RNN 구조와 달리 단어 입력을 순차적으로 받는 방식이 아니므로 단어의 위치 정보를 다른 방식으로 알려줄 필요가 있다.
- 임베딩 벡터는 특정 단어가 문장 내에서 어느 위치에 있는지에 대한 정보를 전달하지 않는다.
- 임베딩 벡터에 Positional Encoding을 더해서 최종 Input Embedding으로 사용된다.

		i = 0	i = 0	i = 1	i = 1
I	p = 0	PE(0,0)	PE(0,1)	PE(0,2)	PE(0,3)
am	p = 1	PE(1,0)	PE(1,1)	PE(1,2)	PE(1,3)
a	p = 2	PE(2,0)	PE(2,1)	PE(2,2)	PE(2,3)
student	p = 3	PE(3,0)	PE(3,1)	PE(3,2)	PE(3,3)
<PAD>	p = 4	PE(4,0)	PE(4,1)	PE(4,2)	PE(4,3)
<PAD>	p = 5	PE(5,0)	PE(5,1)	PE(5,2)	PE(5,3)

2. Translation

최종 Input Embedding

- 임베딩 벡터에 Positional Encoding을 더해서 최종 Input Sequence으로 사용된다.

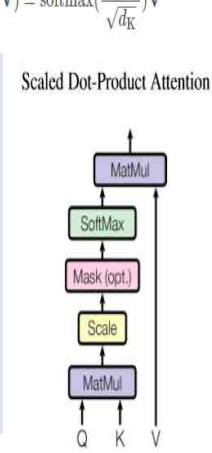


2. Translation

$$\text{Attention}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{softmax}(\frac{\mathbf{Q}\mathbf{K}^{\top}}{\sqrt{d_{\mathbf{K}}}})\mathbf{V}$$

Scaled Dot-Product Attention

- 주어진 입력 시퀀스로부터 Query(Q), Key(K), Value(V)를 생성한다. 이는 각 단어에 대해 선형 변환을 수행하여 얻는다.
- Q와 K 벡터를 내적해서 Attention score를 얻는다.
- 내적 값이 커지면 큰 값이 민감하게 반응하는 Softmax 함수 특성에 의해 문제가 생길 수 있다. 또한 역전파 중에 큰 내적 값이 매우 작은 기울기를 생성할 수 있는 gradient vanishing 문제를 해결하기 위해서 $\sqrt{d_k}$ 로 나눠준다.
- Softmax를 적용하기 전에, Pad 마스킹과, Look-Ahead 마스킹을 한다.
- Softmax를 적용시켜 Attention Weight를 얻는다.
- Attention Weight와 V벡터의 행렬곱을 계산한다.

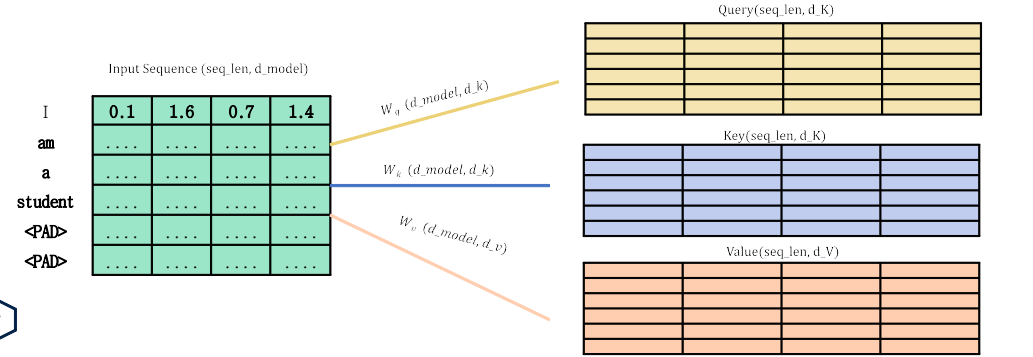


2. Translation

Scaled Dot-Product Attention

- Input Sequence로부터 Q, K, V 벡터를 얻는다.
- Head를 고려하지 않은 예시이므로 d_k = d_v = d_model로 설정한다.

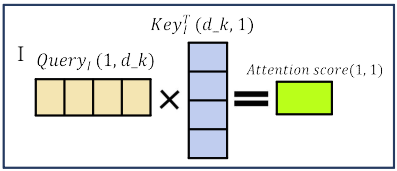
d_model = 4
d_k = d_v = 4
seq_len = 6

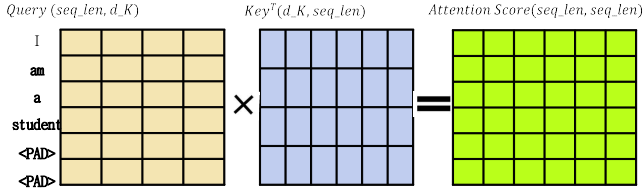


2. Translation

Self-Attention

- 첫번째 그림은 하나의 토큰에 대한 Q와, K를 통해 Attention을 수행한 결과이다.
- 결과적으로는 모든 토큰에 대한 Q와, K를 통해 Attention Score를 행렬연산을 통해 구할 수 있다.

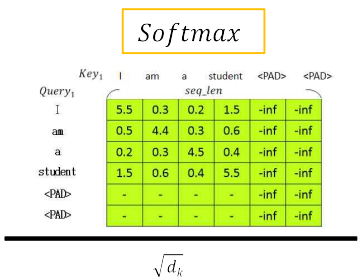




2. Translation

Self-Attention

- Attention Score 행렬에 d_K의 제곱근으로 scale 한 후 Softmax를 취해서 Attention Weight 행렬을 구한다.
- d_K의 제곱근으로 Scale하는 이유는 내적 연산으로 인해 값이 커지게 되는데, Softmax 함수가 작은 값에 비해 큰 값들을 높은 확률로 변환하기 때문에 범위를 조정해주기 위해 scale하였다.
- <PAD>에 -inf 값을 더해줌으로써 Softmax를 취했을 때 0이 되도록 한다.



=

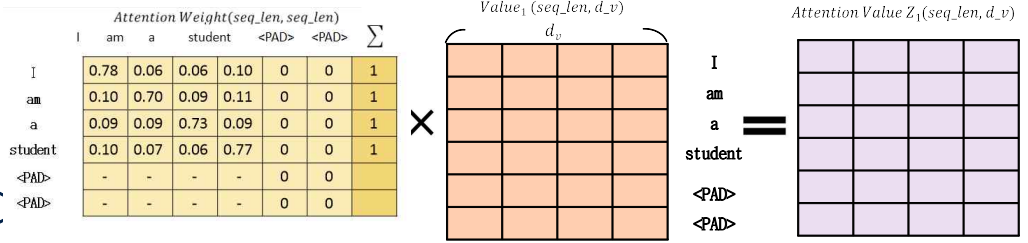
$Attention\ Weight(seq\_len, seq\_len)$

	I	am	a	student	<PAD>	<PAD>	$\sum$
I	0.78	0.06	0.06	0.10	0	0	1
am	0.10	0.70	0.09	0.11	0	0	1
a	0.09	0.09	0.73	0.09	0	0	1
student	0.10	0.07	0.06	0.77	0	0	1
<PAD>	-	-	-	-	0	0	
<PAD>	-	-	-	-	0	0	

2. Translation

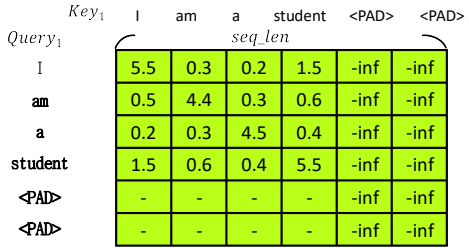
Self-Attention

- Softmax를 수행하면 Attention score가 0~1 사이의 확률값으로 변환된다.
- Attention Weight와 Value 벡터를 곱해서 Attention Value값을 얻는다.
- Score 값이 클수록



2. Translation

Self-Attention



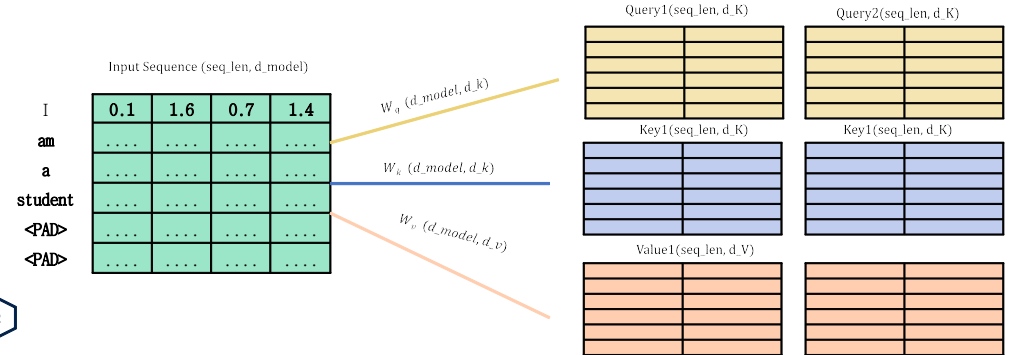
2. Translation

Multi-Head Attention

- Input Sequence로 부터

$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

d_model = 4
d_k = d_model / h = 2
d_v = d_model / h = 2
seq_len = 6



2. Translation

Multi-head Attention

$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

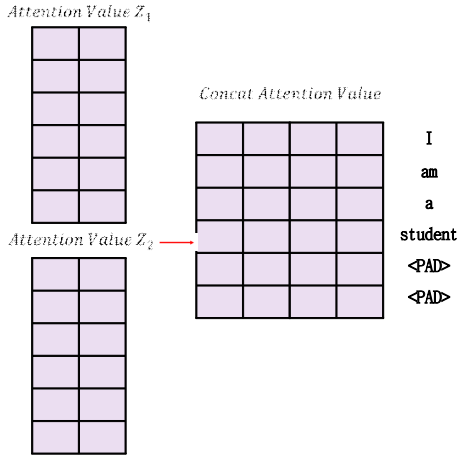
- d_model

I	0.1	1.6	0.7	1.4
am	....	....	....	....
a	....	....	....	....
student	....	....	....	....
<PAD>	....	....	....	....
<PAD>	....	....	....	....

2. Translation

Multi-head Attention

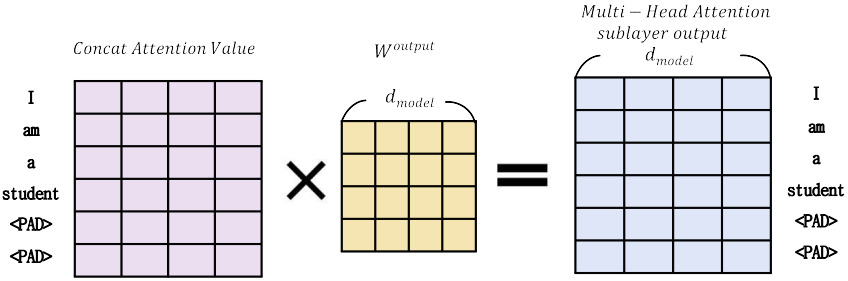
- 각 Head에 각각의 다른 Q, K, V 가중치 행렬을 가진다.
- 즉, 각 Head에서 Self-Attention 과정(Scaled-Dot Product Attention)을 다른 Q, K, V weight 행렬들에 대해 수행한다.
- Head의 수 만큼 서로 다른 Attention Value 행렬들을 가지게 되는데, 이를 Feed Forward에 하나의 입력으로 사용하기 위해 Concat 한다.
- Concat 후에 학습된 가중치인 Wout을 곱해서 최종 선형 변환을 수행한다.



2. Translation

Multi-Head Attention

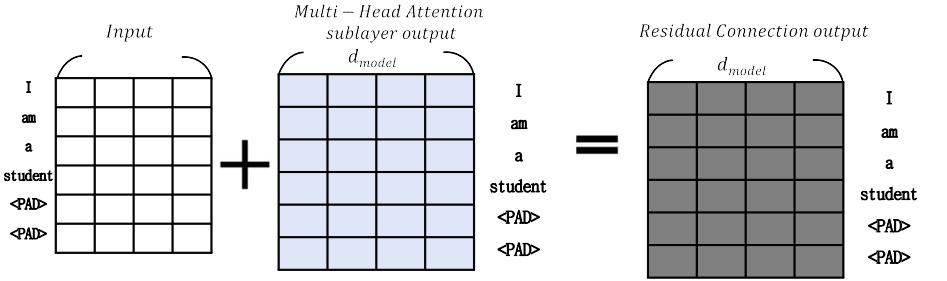
- Concat 후에 학습된 가중치인 W^{output} 를 곱해서 최종 선형변환 후 sublayer의 출력으로 사용된다.



2. Translation

Residual & Layer Normalization

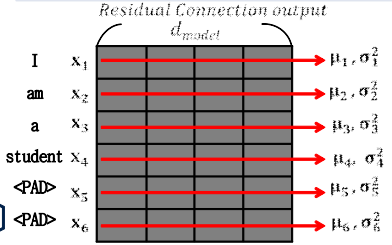
- Residual Connection : output = x +sublayer(x) 의 구성이며, 층이 깊어질 경우 발생할 수 있는 gradient 소실 문제를 해결한다.
- 잔차연결의 결과인 output은 layer-normalization의 입력으로 사용된다.



2. Translation

Residual & Layer Normalization

- x_i 벡터를 평균과 분산을 통해 정규화를 한다.
- Layer Normalization 을 수행한 후에는 벡터 x_i 는 ln_i 라는 벡터로 정규화가 된다.
- 아래 수식의 γ, β 는 학습가능한 파라미터이며, 초기값은 각각 1과 0이다.
- Batch-Normalization과 달리 Batch-Size에 영향을 거의 받지 않고 테스트 할 때에도 훈련과 동일한 방식으로 정규화가 가능하다.



$$ln_i = LayerNorm(x_i)$$

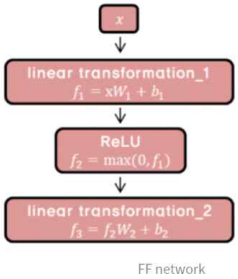
$$\hat{x}_{i,k} = \frac{x_{i,k} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

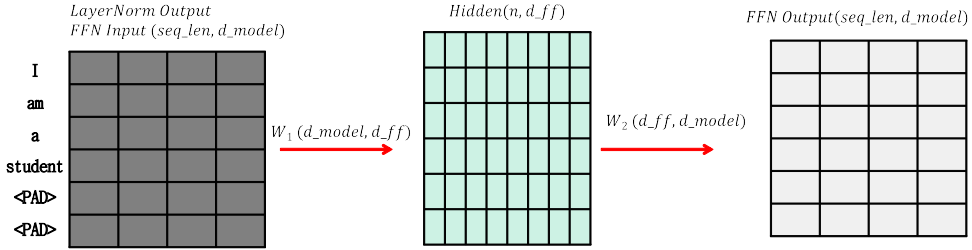
$$ln_i = \gamma \hat{x}_i + \beta = LayerNorm(x_i)$$

2. Translation

Feed Forward Layer

- 첫 번째 레이어는 입력을 확장하고, 활성화 함수를 통해 비선형 변환을 적용한 후, 두 번째 레이어를 통해 다시 원래 차원으로 줄어든다.
- 활성화 함수로는 ReLU가 사용되었으며 비선형성을 제공하여, 선형적으로 분리할 수 없는 데이터 패턴을 학습하는 데 도움을 준다.
- W1, W2 가중치 파라미터는 각 레이어마다 다르게 학습된다.

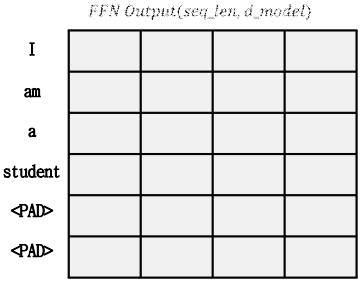




2. Translation

Feed Forward Layer

- 인코더의 FFN output은 Residual Connection과 Layer Normalization을 수행 후에 다음 층의 인코더의 입력으로 사용된다.



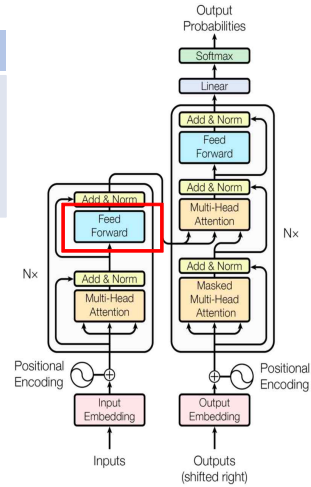
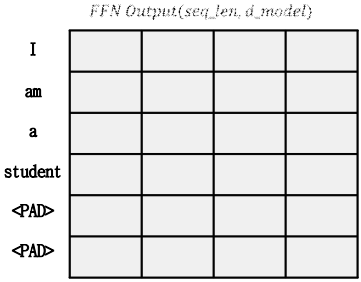


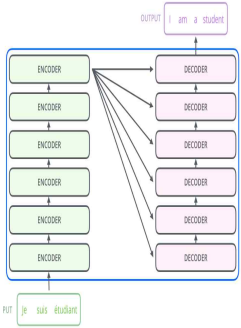
Figure 1: The Transformer - model architecture.

2. Translation

Encoder Output

- 인코더는 모든 층이 동일한 구조로 되어있으며, 각 층의 출력이 다음 층의 입력으로 사용된다.
- 인코더 마지막 층의 출력은 디코더의 2번째 sublayer의 입력으로 사용된다.





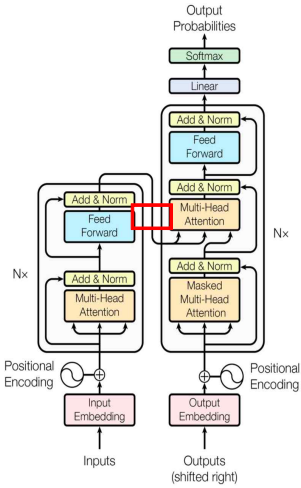


Figure 1: The Transformer - model architecture.

2. Translation

Tokenization

”나는 학생이다” 문장 토큰화

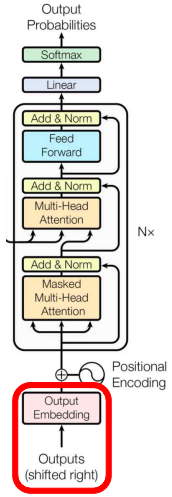
➔[<BOS>, ‘나’, ‘는’, ‘학생’, ’이다’, <EOS>, <PAD>, <PAD>]

각 문장 성분에 대해 단어 사전에 맞게 Encoding 하고,

Lookup table을 참조하여 Embedding Vector 생성.

➔학습할 때마다 Lookup table의 가중치 갱신.

Embedding =



2. Translation

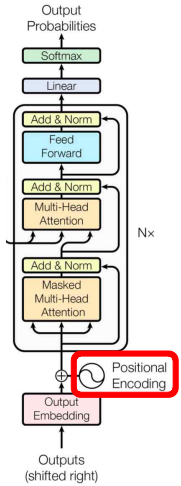
Positional Encoding

	d_model			
<BOS>				
'나'				
'는'				
'학생'				
'이다'				
<EOS>				
<PAD>				
<PAD>				

+

Positional Encoding 적용

d_model			
0.	1.	0.	1.
0.84	0.54	0.1	1.
0.91	-0.42	0.2	0.98
0.14	-0.99	0.3	0.96
-0.76	-0.65	0.39	0.92
-0.96	0.28	0.48	0.88
-0.28	0.96	0.56	0.83
0.66	0.75	0.64	0.76

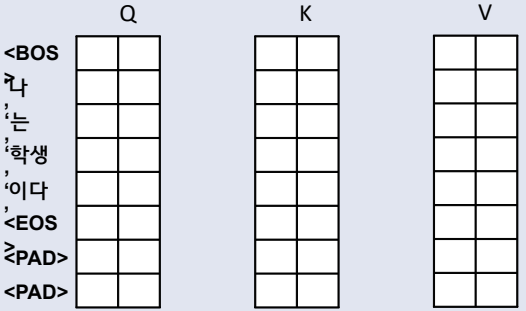


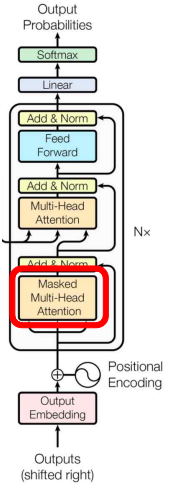
2. Translation

Masked Attention

Self-Attention Query, Key, Value 정의. $d_k = d_v = \frac{d_{model}}{h}$로 논문과 동일하게 가정.

Self-Attention이기 때문에 $d_q = d_k$ 로 모두 동일한 차원이다.

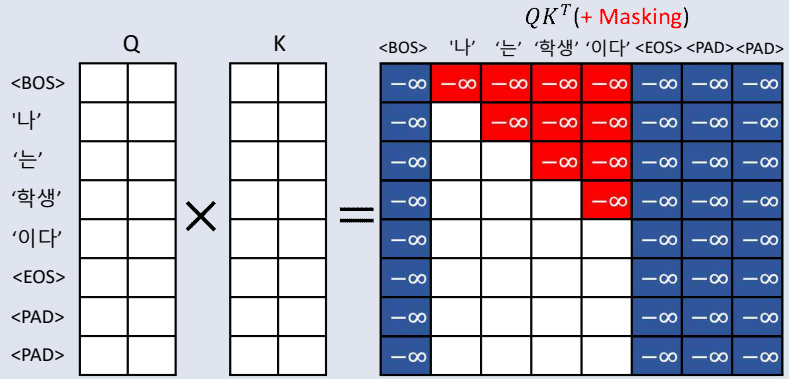


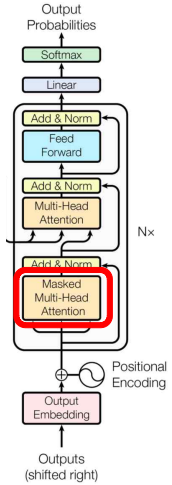


2. Translation

Masked Attention

Padding Masking과 Look-Ahead Masking 적용 후 QK 행렬곱





2. Translation

Masked Attention

$$softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

	<BOS>	'나'	'는'	'학생'	'이다'	<EOS>	<PAD>	<PAD>
<BOS>	0	0	0	0	0	0	0	0
'나'	0		0	0	0	0	0	0
'는'	0			0	0	0	0	0
'학생'	0				0	0	0	0
'이다'	0					0	0	0
<EOS>	0					0	0	0
<PAD>	0					0	0	0
<PAD>	0					0	0	0

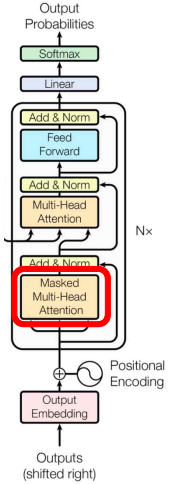
(max_len, max_len)

× V

(max_len, d_v)

= $softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$

(max_len, d_v)

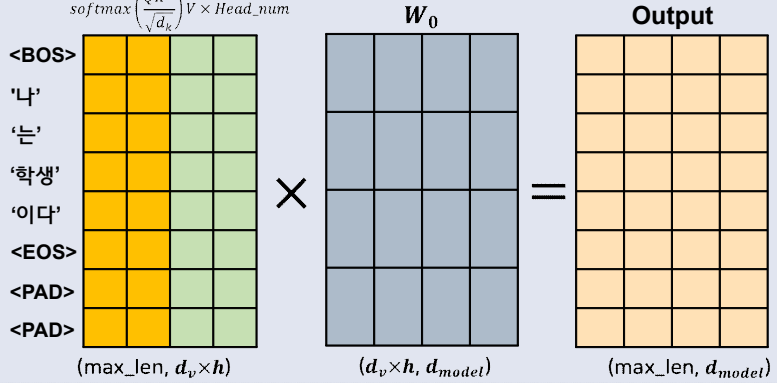


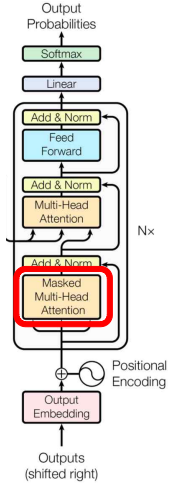
2. Translation

Multi-Head Attention

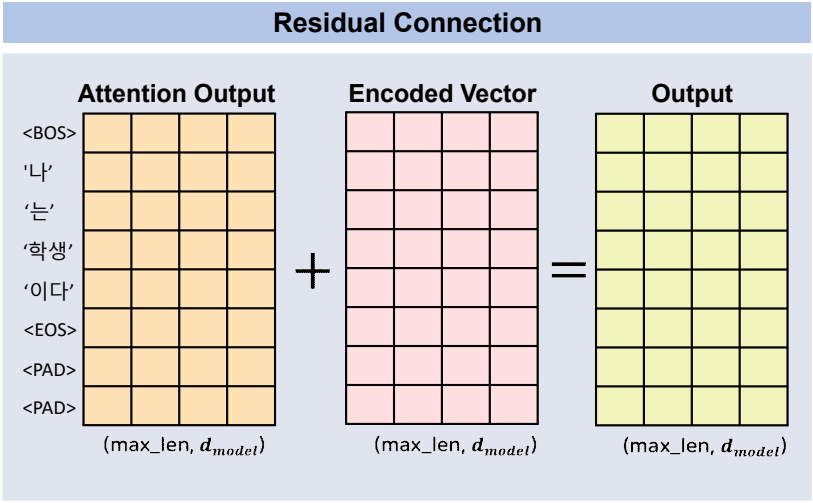
모두 합한 값에 원하는 차원인 d_{model}로 만들기 위해 W_0 가중치 행렬을 곱함.

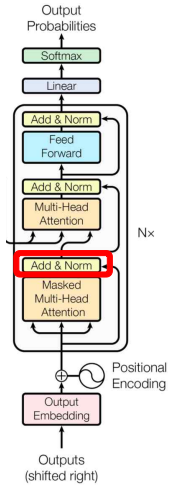
$$softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \times Head\_num$$





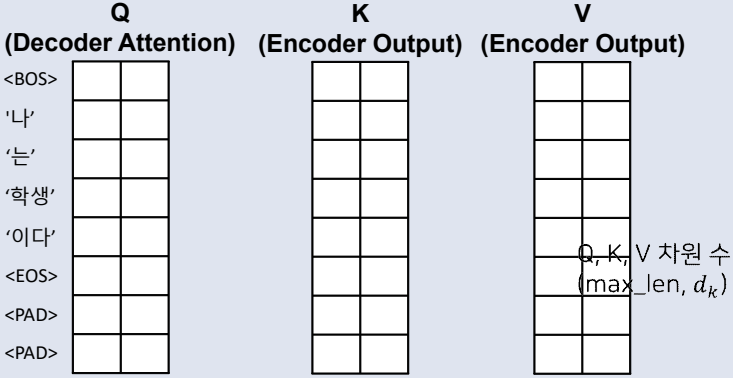
2. Translation

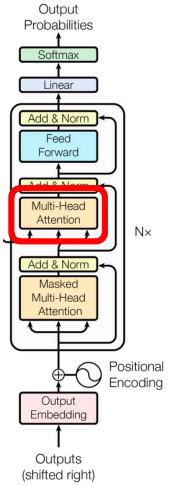




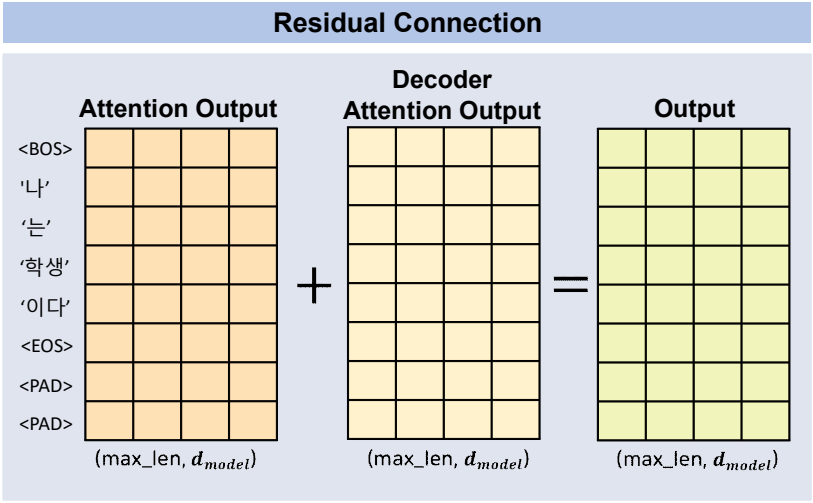
2. Translation

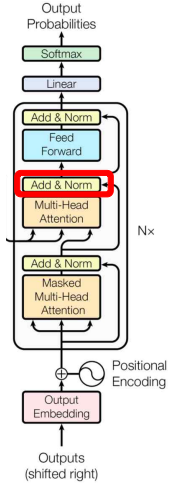
Multi-Head Attention





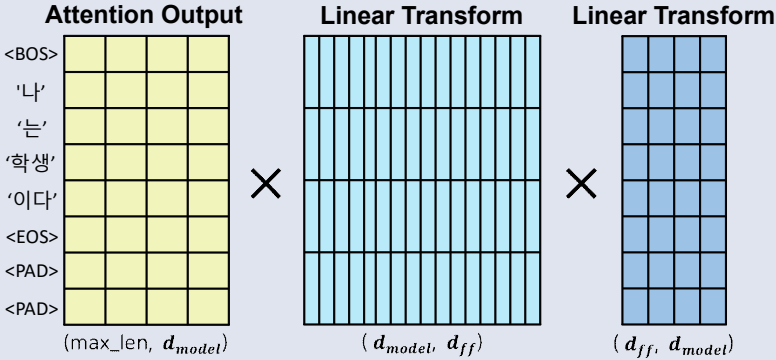
2. Translation



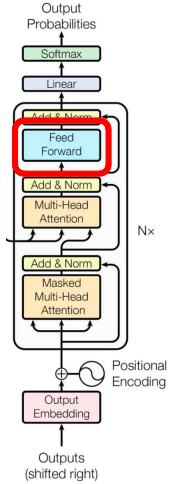


2. Translation

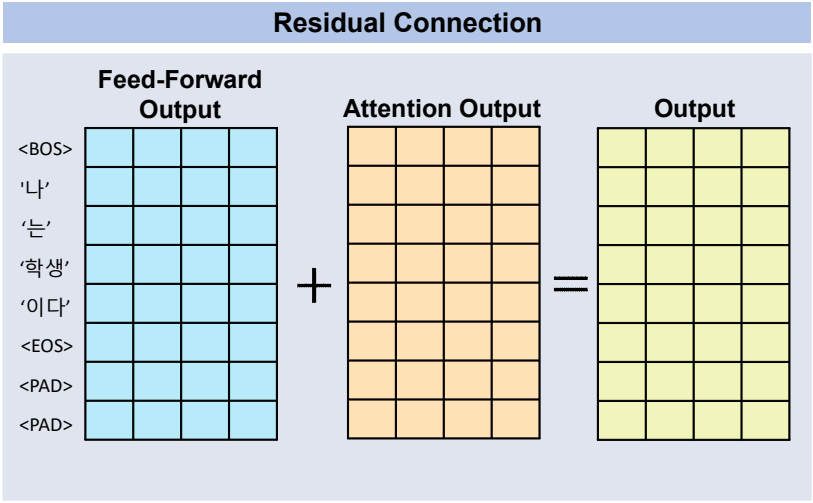
Feed-Forward Network



➔최종 Output (max_len, d_{model})

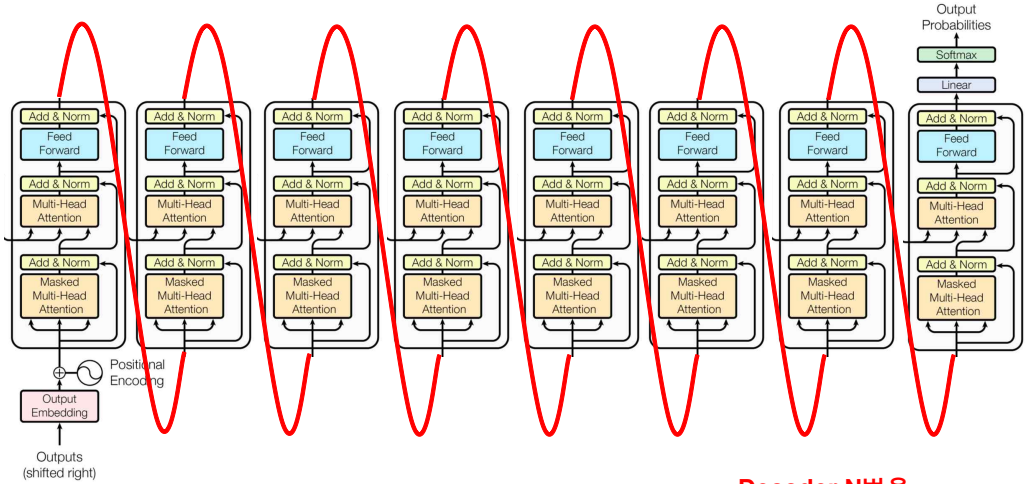


2. Translation



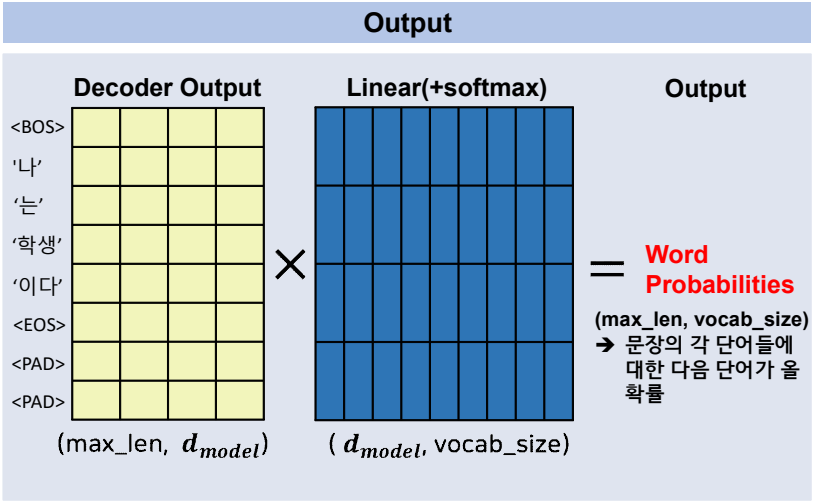
2. Translation

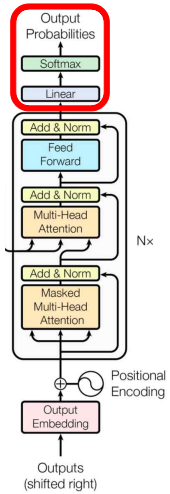
Decoder × N



Decoder N번을 반복

2. Translation





3. 번역 모델 전처리 진행 상황

번역 모델 전처리 1차 시도

토큰화와 단어 임베딩 과정은

1. **무슨 데이터로 딕셔너리를 만들어서**
2. **어떻게 단어 성분을 나눠서 토큰화를 할지**

정하는 과정부터 진행할 수 있으나, 모든 것이 정의된 pretrained 모델을 사용하는 것으로 이 부분은 간단하게 해결하였음.

SKTBrain에서 설계한 KoBERT Pretrained 모델에서 Tokenizer만 가져와서 토큰화를 시도하였지만 실패하였음.

- Pretrained Tokenizer의 최대 문장 길이가 정의가 되어있지 않아 문장에 Padding 적용이 번거롭고,
- 한글 위키 기반의 8002개 단어로만 이루어져 있어 영어 Encoding에는 취약하다는 문제점이 존재.

➔**한영 Tokenizer가 모두 존재하는 모델을 찾아서 적용 시도 중**

3. 번역 모델 전처리 진행 상황

진행중인 번역 모델 전처리 과정

학습 데이터셋 : AiHub 한국어-영어 번역 말뭉치(기술과학)

HuggingFace의 transformers MarianMTModel 라이브러리 기반으로 제작된 Tokenizer를 사용.

Tokenizer 모델 이름 : Helsinki-NLP/opus-mt-ko-en

학습에 사용된 총 단어 개수 : 65000

학습에 사용된 문장 최대 길이 : 512

Thank You!

HONGIK UNIVERSITY
HIGH-PERFORMANCE DATA PROCESSING & ANALYSIS LAB