Transformer

Contents

1. 기존의 seq2seq 모델의 한계

기존의 seq2seq 모델은 Encoder – Decoder 구조로 구성되어 있었음.

Encoder: 입력 시퀀스를 1개의 벡터 표현(Context Vector)으로 압축함

Decoder: 이 벡터표현을 통해 출력 시퀀스를 만들어냄

Seq2seq의 단점

: Encoder가 입력 시퀀스를 1개의 벡터로 압축하는 과정에서 **입력 시퀀스의 정보가 일부 손실**된다.

⇒ 이를 보정하기 위해서 Attention이 사용됨

⇒ Attention을 RNN의 보정을 위한 용도 뿐 아니라, Attention만으로 Encoder와 Decoder를 만들어보자라는 생각에서 Transformer가 등장.

2. Transformer의 주요 하이퍼파라미터

시작 전, Transformer의 하이퍼파라미터를 정의한다.(Transformer에는 이러한 하이퍼파라미터가 존재한다는 정도로 이해해두자.)

아래에서 정의하는 수치는 Transformer를 제안한 논문에서 사용한 수치임

- $d_{model} = 512$:Transformer의 **Encoder와 Decoder에서 정해진 입력과 출력의 크기**를 의미함

-> 임베딩벡터의 차원 또한 d_{model}이며, Encoder와 Decoder가 다음층의 Encoder와 Decoder로 값을 보낼 때에도 이 차원을 유지한다.

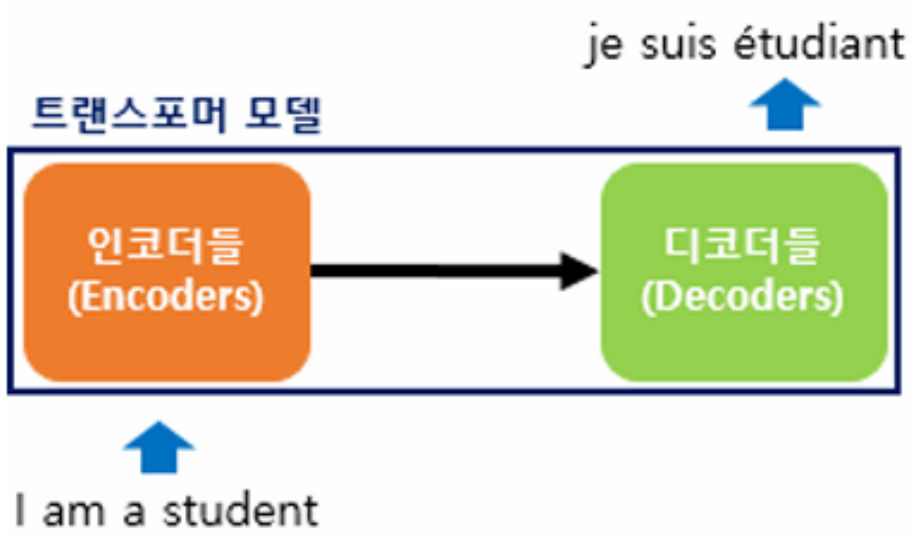
- num_layers = 6 : 1개의 Encoder와 Decoder를 층으로 생각하였을 때, Transformer 모델에서 **Encoder와 Decoder가 총 몇 층으로 구성되었는지**를 의미함
- num_heads = 8 : Transformer에서는 Attention을 사용할 때, 한번 하는 것보다 여러 개로 분할해서 병렬로 Attention을 수행하고 결과값을 다시 하나로 합치는 방식을 택함. 이때 **병렬의 개수**를 의미함.
- $d_{ff} = 2048$: Transformer의 내부에 존재하는 **Feed forward Neural Network의 은닉층의 크기**

+ FFNN의 입력층과 출력층의 크기는 d_{model}이다.

3. Transformer(트랜스포머)

Transformer는 RNN을 사용하지 않지만,

기존의 seq2seq처럼 Encoder에서 입력 시퀀스를 입력받고, Decoder에서 출력 시퀀스를 출력하는 **Encoder-Decoder 구조를 유지**하고 있음



< Transformer의 Encoder-Decoder구조 >

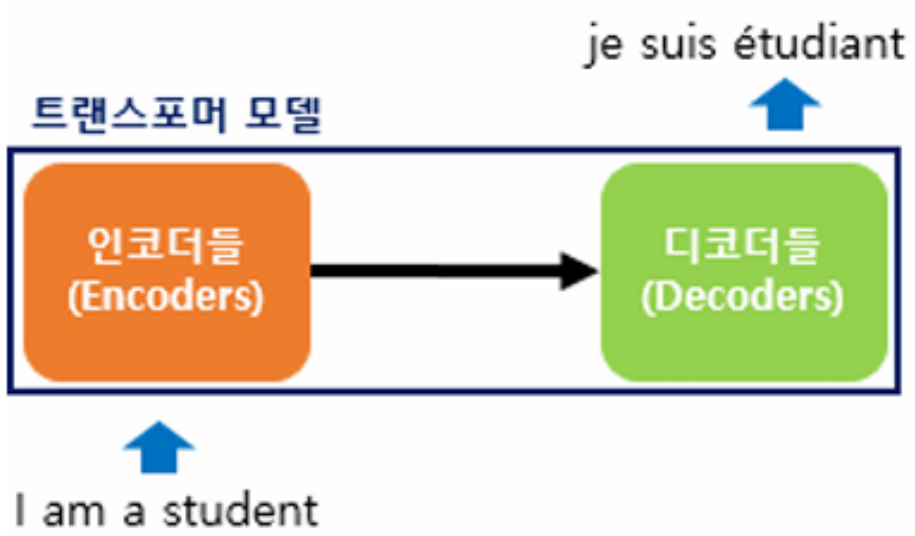
3. Transformer(트랜스포머)

다만, 이전 seq2seq에서는 Encoder와 Decoder에서 각각 1개의 RNN이 t개의 시점(time step)을 가지는 구조였다면

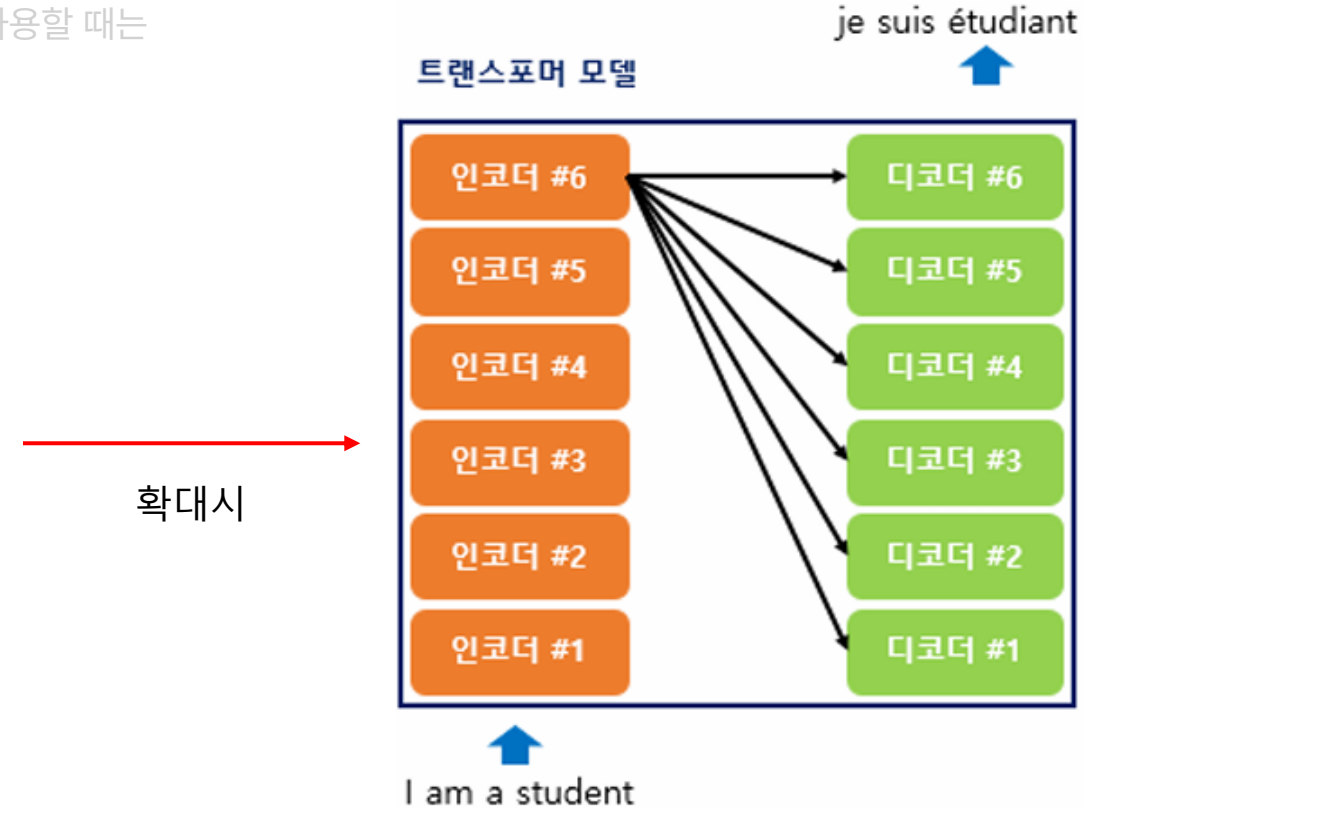
Transformer에서는 Encoder와 Decoder라는 **단위**가 N개로 구성되는 구조임.(논문에서는 Encoder와 Decoder의 개수를 각각 6개를 사용함)

편의를 위해 Encoder와 Decoder가 여러 개 쌓여있다는 의미를 사용할 때는

encoder**s**, decoder**s**라고 표현한다.



< Transformer의 Encoder-Decoder구조 >



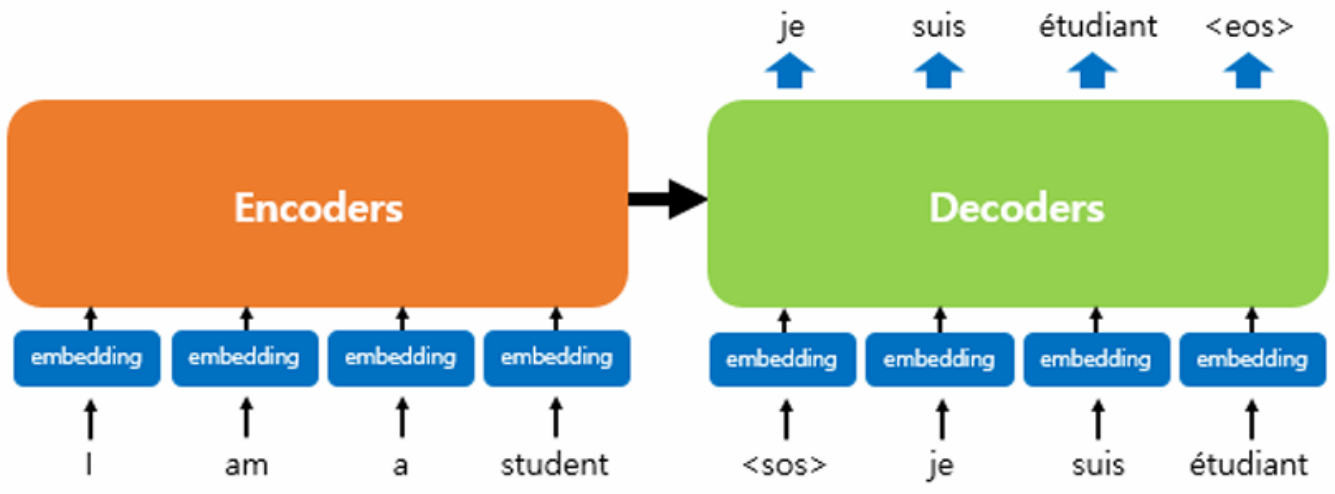
확대시

<Encoder와 Decoder가 6개씩 존재하는 Transformer의 구조>

3. Transformer(트랜스포머)

아래 그림은 Encoder로부터 정보를 전달받아 Decoder가 출력 결과를 만들어내는 Transformer 구조를 보여줌.

- Decoder는 기존의 seq2seq 구조처럼 시작심볼 <sos>를 입력으로 받아 종료 심볼 <eos>가 나올때까지 연산을 진행함
- 이는 RNN이 사용되지 않지만 **여전히 Encoder-Decoder구조가 유지되고 있음**을 보여줌.



- Transformer의 내부 구조를 조금씩 확대해가는 방식으로 설명을 진행할 것
- Transformer의 Encoder와 Decoder는 단순히 각 단어의 임베딩벡터들을 입력받는것이 아니라 **임베딩벡터에서 조정된 값을 입력**받음.

4. Positional Encoding

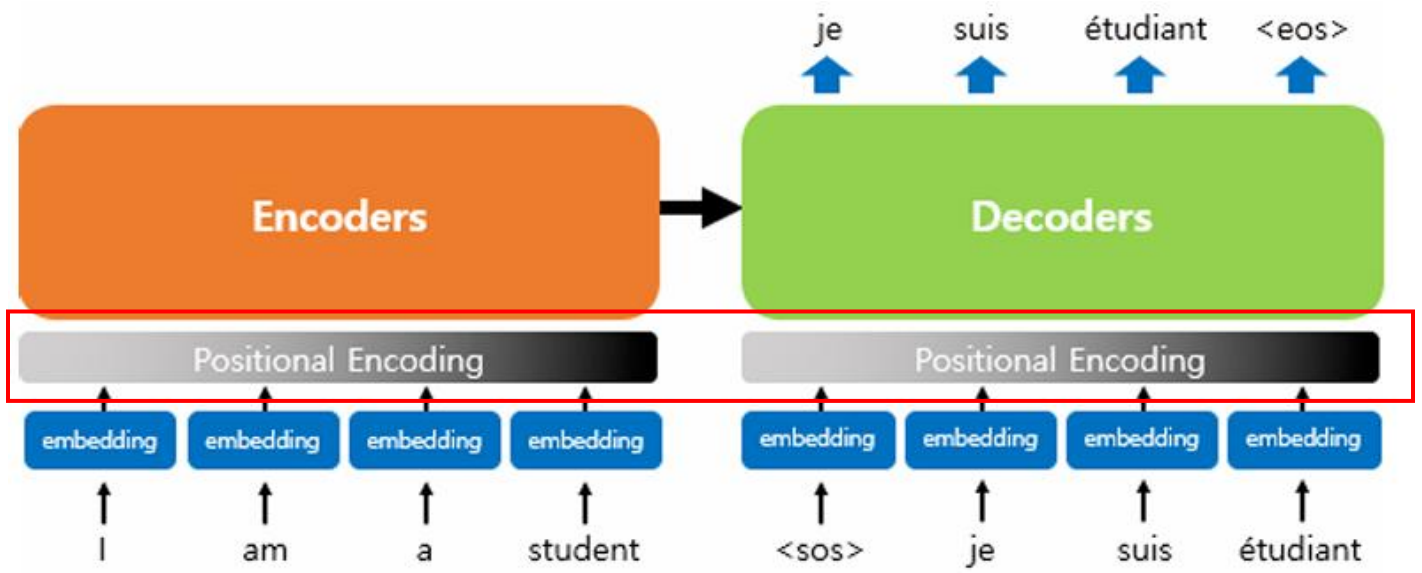
Transformer의 내부를 이해하기 전, Transformer의 입력에 대해 알아보자.

RNN이 자연어처리에 유용했던 이유는 단어 위치에 따라 단어를 순차적으로 입력받아서 처리하는 RNN의 특성으로 인해 **각 단어의 위치정보 (position information)를 가질 수 있다**는 점에 있다.

But! Transformer는 단어 입력을 순차적으로 받는 방식이 아니며, **입력된 문장을 병렬로 한번에 처리한다는 특징**을 가진다.

따라서, 단어의 위치 정보를 다른 방식으로 알려줄 필요가 있다. -> 논문에서는 이에 대해 Positional Encoding을 제안함

- **Positional Encoding**: 단어의 위치 정보를 얻기 위해 각 단어의 임베딩벡터에 위치정보들을 더하여 모델의 입력으로 사용하는 것

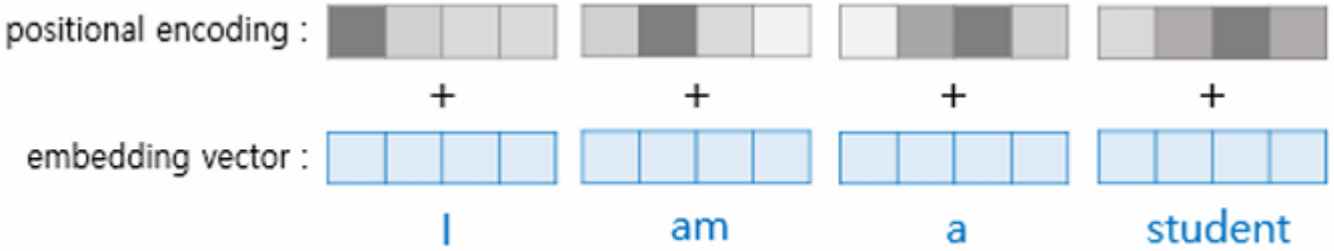


왼쪽 그림은 입력으로 사용되는 임베딩벡터들이 Transformer의 입력으로 사용되기 전에 Positional Encoding 값이 더해지는 것을 보여줌

Positional encoding: 방법론
Positional encoding 값: 어떤 위치를 나타내기 위해 단어 임베딩에 더해주는 값

4. Positional Encoding

임베딩벡터들이 Transformer의 입력으로 사용되기 전 positional encoding 값이 더해지는 과정을 시각화하면 아래와 같다.



Positional encoding 값들은 어떤 값이기에 위치정보를 반영해줄 수 있을 까?

- Transformer에서는 위치 정보를 가진 값을 만들기 위해서 아래의 2개 함수를 사용한다.
- 즉, sin함수와 cos함수의 값을 임베딩벡터에 더해줌으로서 단어의 순서 정보를 추가해주는 것이다.

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

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

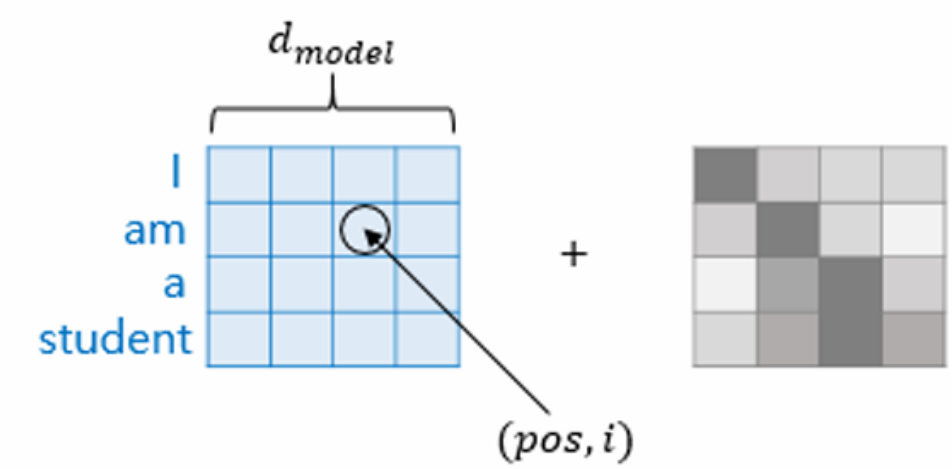
4. Positional Encoding

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

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

위의 식에는 pos, i, d_{model}등의 새로운 변수들이 존재하는데, 위의 함수를 이해하려면 아래의 정보를 이해해야한다.

- 임베딩벡터와 positional encoding의 덧셈은 **임베딩벡터가 모여 만들어진 문장 행렬**과 **positional encoding 행렬**의 **덧셈 연산**으로 이루어진다.



< 임베딩벡터와 positional encdoing의 덧셈 >

pos: 임베딩벡터의 위치(row)

i: 임베딩벡터 내의 차원의 index
- Index가 짝수인 경우$((pos, 2i))$ -> sin함수 값을 사용
- Index가 홀수인 경우$((pos, 2i+1))$ -> cos함수 값을 사용

d_{model}: Transformer의 모든 층의 출력 차원을 의미하는 하이퍼파라미터
- 임베딩벡터또한 d_{model}의 차원을 가짐

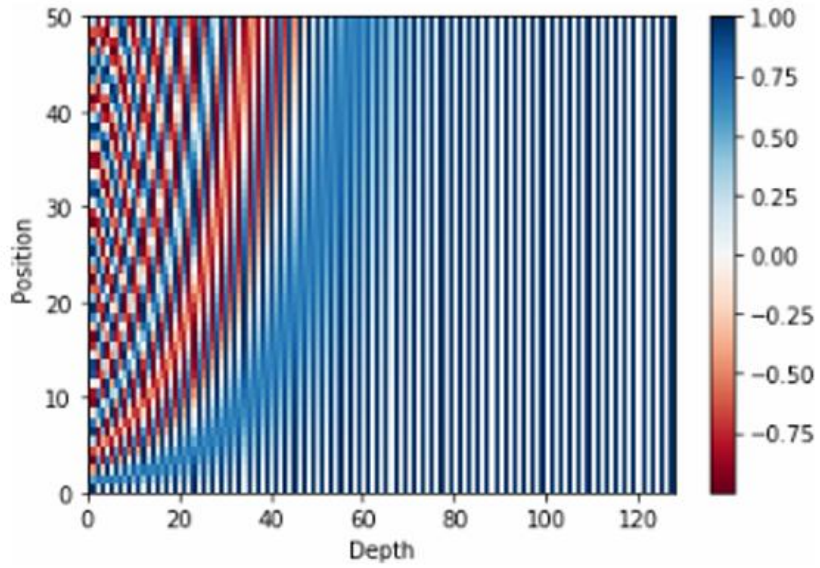
4. Positional Encoding

Positonal Encoding 방법을 사용하면 순서정보가 보존됨.

- Ex. 각 임베딩벡터에 positional encoding 값을 더하면 같은 단어라고 하더라도
  문장 내의 위치에 따라서 Transformer의 입력으로 들어가는 임베딩벡터의 값이 달라진다.
- 따라서 Transformer의 입력은 순서정보가 고려된 임베딩벡터가 됨.

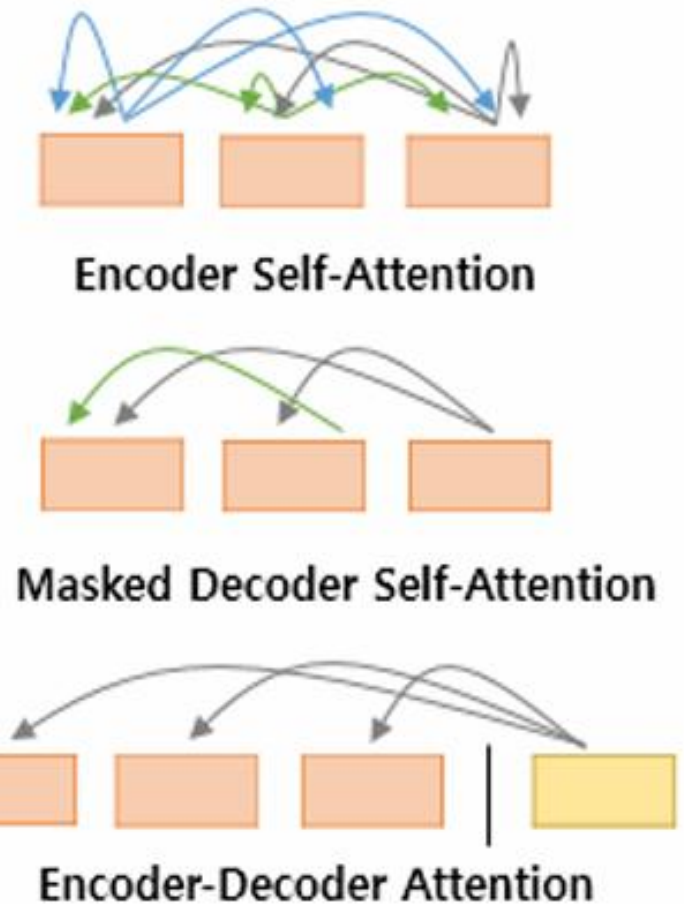
50 x 128 크기를 가지는 Positional Encoidng 행렬을 시각화해보자.

입력문장의 **단어가 50개**, 각 단어가 **128차원의 임베딩벡터**를 가질 때 사용할 수 있는 행렬이다.



5. Attention

Transformer에서 사용되는 3가지 Attention에 대해 간단히 정리해보자.(큰 그림에 대한 이해)



1번째 그림인 self-Attention은 **Encoder**에서 이루어지며

2번째 그림의 self-Attention과 3번째 그림의 Encoder-Decoder Attention은 **Decoder**에서 이루어진다.

Self-attention: 본질적으로 Query, Key, Value가 같은 경우의 Attention

- **Query, Key등이 같다**는 것은 벡터의 값이 같다는 것이 아니라
  벡터가 나온 출처가 같다(Q,K,V가 모두 Encoder의 입력시퀀스에서 나옴)는 의미이다.

3번째 그림의 Encoder-Decoder Attention에서는

- Query가 Decoder의 벡터인 반면에
- Key와 Value가 Encoder의 Vector이므로 self-attention이라고 부르지 않는다.

5. Attention

정리하면 아래와 같다.

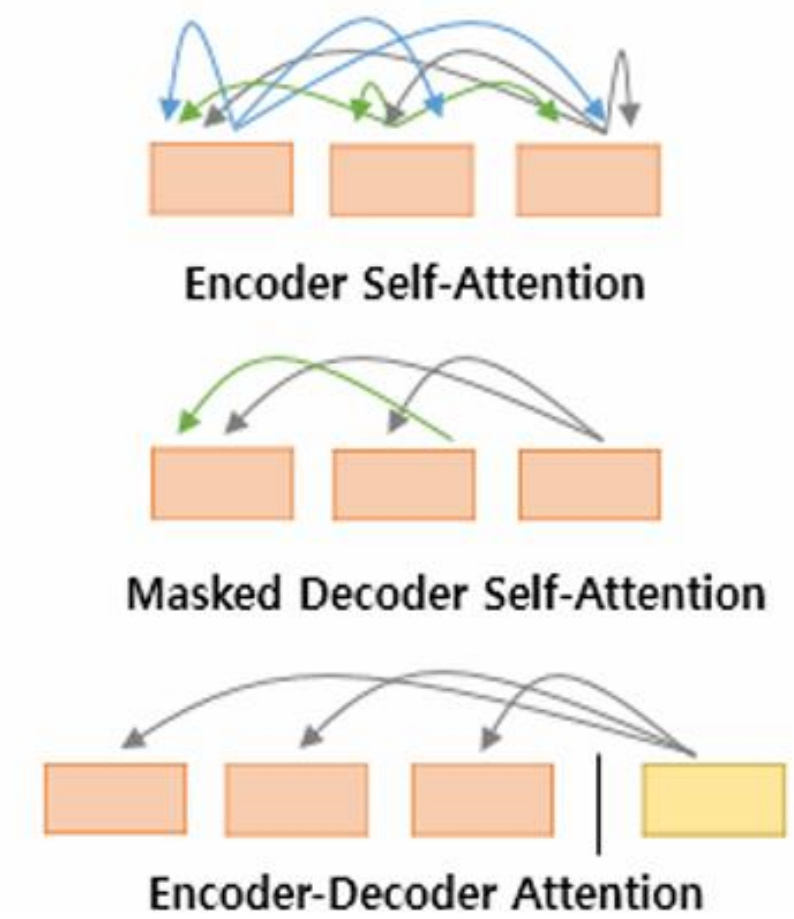
Encoder

1. Encoder의 Self-Attention: Query = Key = Value

Decoder

2. Decoder의 Masked Self-Attention: Query = Key = Value
3. Decoder의 Encoder-Decoder Attention:

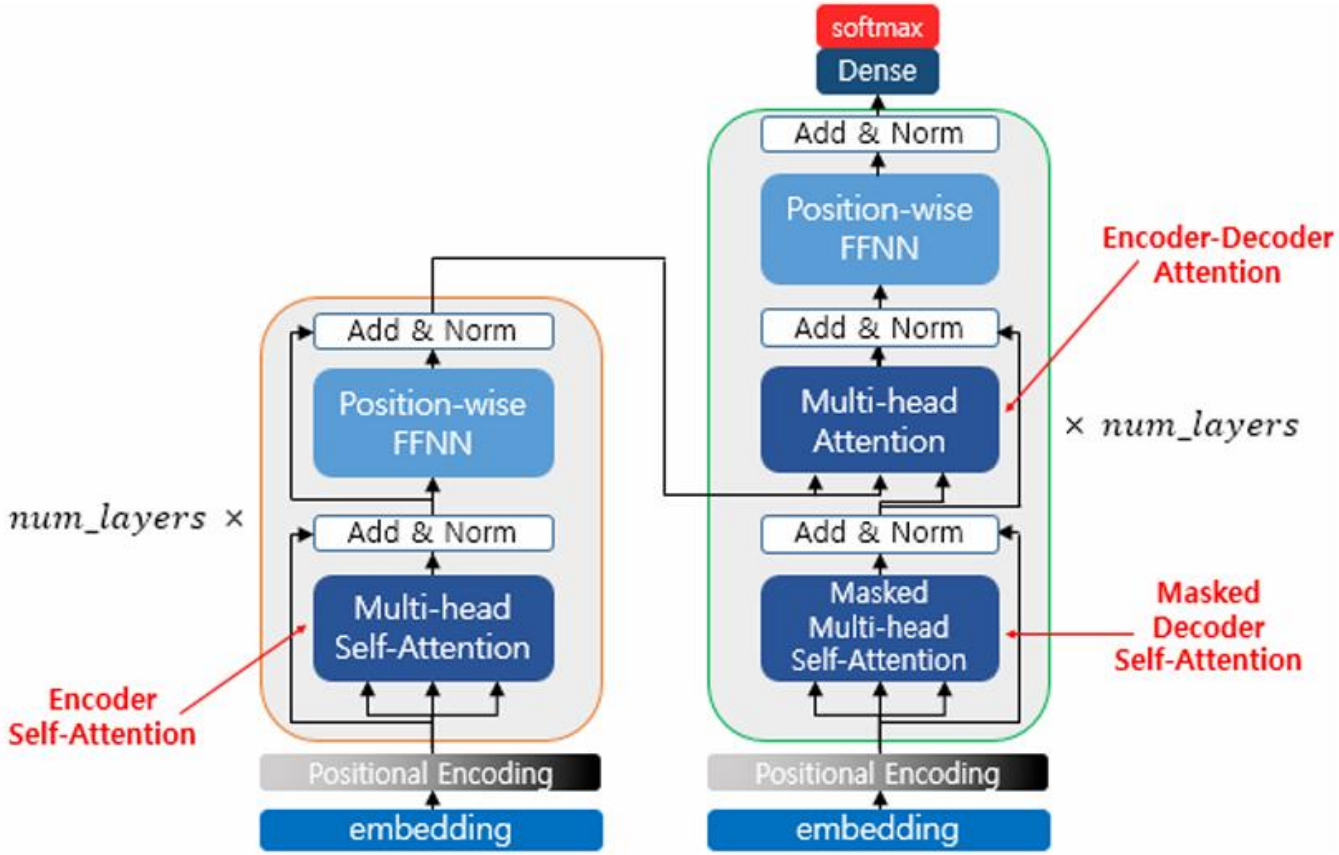
- Query: Decoder 벡터
- Key=Value: Encoder벡터



5. Attention

아래 그림은 Transformer 아키텍처에서 **3가지 Attention이 각각 어디에서 이루어지는지**를 보여줌

3개의 Attention에 ‘Multi-head’라는 이름이 추가로 붙어져 있는데, 이는 **Transformer가 Attention을 병렬적으로 수행하는 방법을 의미**함



6. Encoder

Encoder의 구조에 대해서 알아보자.

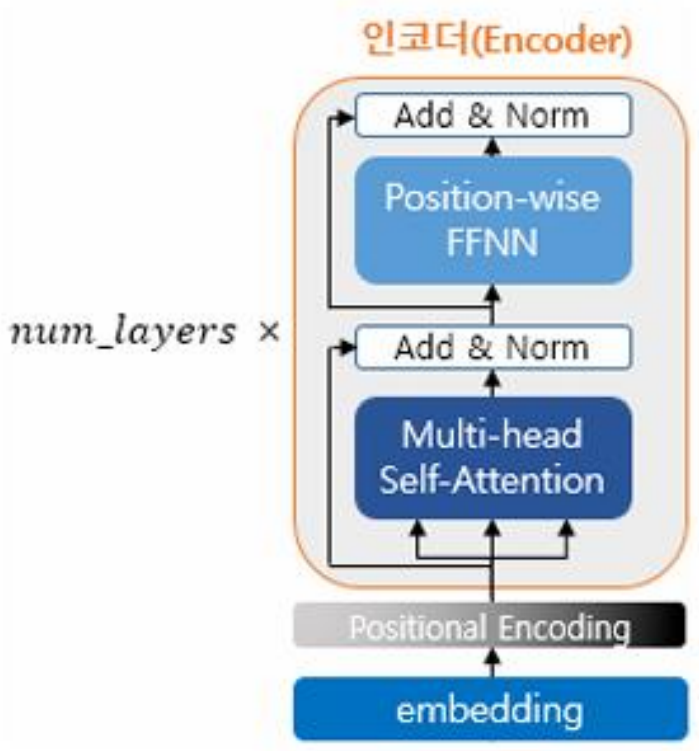
Transformer는 하이퍼파라미터인 **num_layers** 개수의 Encoder층을 쌓는다. (논문에서는 총 6개의 Encoder 층)

Encoder를 하나의 "층 " 이라는 개념으로 생각하면, **1개의 Encoder 층**은 총 **2개의 sublayer(서브층)으로 나뉘어진다.**

-> self-attention과 Feed Forward Neural Network(FFNN)

오른쪽 그림에서는 Multi-head Self-Attention과 Position-wise FFNN이라고 적혀있지만,
기존의 의미에서 추가적인 의미만을 가질 뿐이다.

- Multi-head Self-attention: self-attention을 병렬로 사용하였다는 의미
- Position-wise FFNN: 일반적인 Feed Forward 신경망

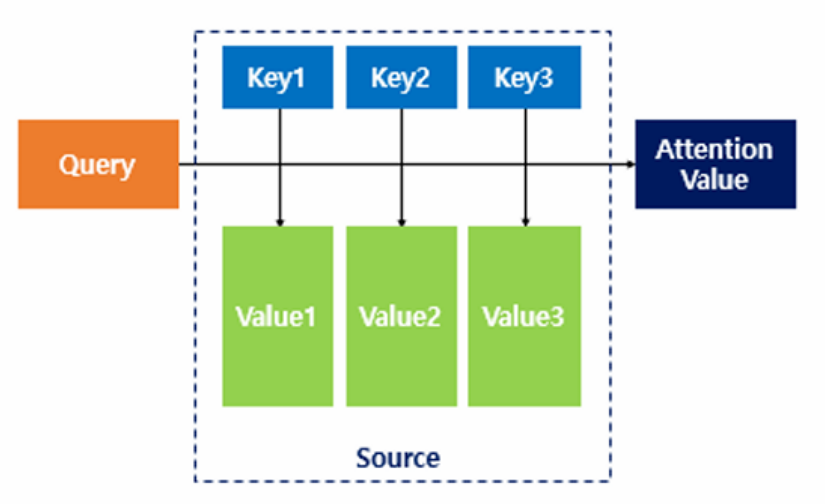


7. Encoder의 Self-attention :1) Self-attention의 의미와 이점

기존의 Attention함수는 주어진 Query에 대해서 모든 Key와의 유사도를 각각 구하고,

구해낸 이 유사도를 가중치로 해서 Key와 맵핑되어 있는 각각의 Value에 반영해준다.

-> 이후 유사도가 반영된 Value를 모두 가중합하여 리턴(=Attention Value를 리턴)한다.



Self-attention: Attention을 자기 자신에게 수행한다는 의미

7. Encoder의 Self-attention
:1) Self-attention의 의미와 이점

Seq2Seq에서 Attention을 사용할경우 Q, K, V의 정의는 아래와 같았다.

Q = Query: **t 시점**의 Decoder 셀에서의 hidden state

K = Keys: 모든 시점의 Encoder 셀에서의 hidden state들

V = Values: 모든 시점의 Encoder 셀에서의 hidden state들

➔ 이때, **t시점**이라는 것은 계속 변화하면서 반복적으로 Query를 수행하므로 결국 아래와 같이 **전체 시점에 대해서 일반화가 가능**하다.

Q = Query: 모든 시점의 Decoder 셀에서의 hidden state들

K = Keys: 모든 시점의 Encoder 셀에서의 hidden state들

V = Values: 모든 시점의 Encoder 셀에서의 hidden state들

이처럼 **기존**에는 Decoder 셀의 hidden state가 Q이고 / Encoder 셀의 hidden state가 K라는 점에서 **Q와 K가 서로 다른 값**을 가지고 있었다.

하지만 self-attention에서는 Q, K, V가 전부 동일하다. Transformer의 self-attention에서 Q,K,V는 아래와 같다.

Q: 입력 문장의 모든 단어벡터들

K: 입력 문장의 모든 단어벡터들

V: 입력 문장의 모든 단어벡터들

7. Encoder의 Self-attention
:1) Self-attention의 의미와 이점

Self-Attention을 통해 얻을 수 있는 대표적인 효과에 대해 알아보자

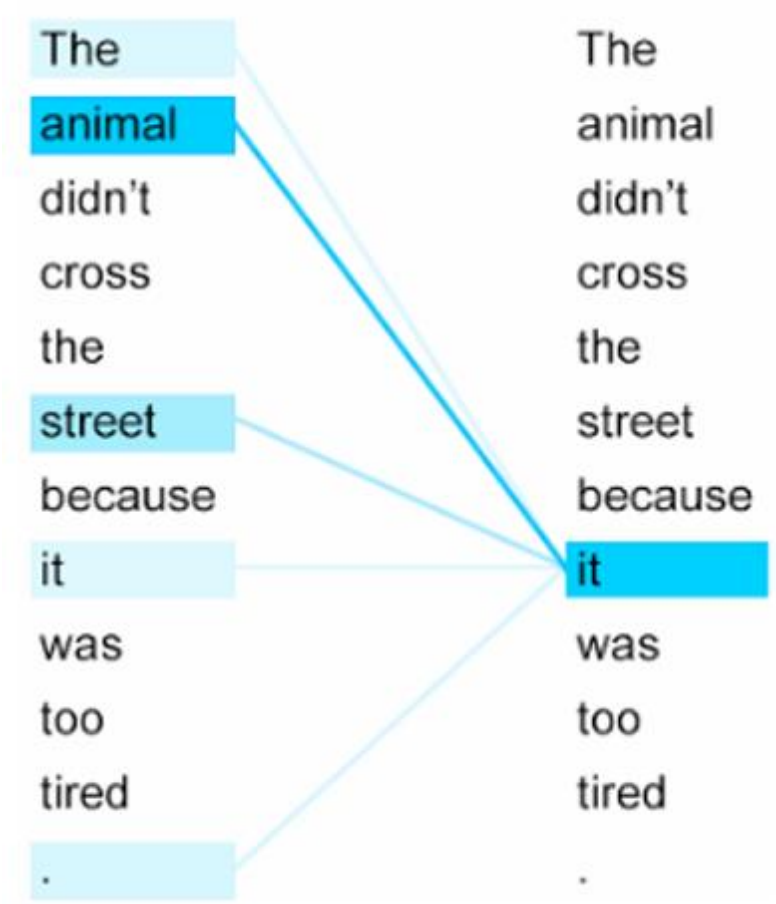
우측의 예시 문장을 번역하면 아래와 같은 의미가 된다.

➔ "그 동물은 길을 건너지 않았다. 왜냐하면 **그것**은 너무 피곤하였기 때문이다"

- 이때 그것(it)에 해당하는 주체가 animal이라는 것을 사람은 쉽게 알 수 있지만, 기계는 그럴 수 없다.

Self-Attention은 **입력 문장 내의 단어들끼리 유사도를 구함**으로서

it이 animal과 연관되었을 확률이 높다는 것을 찾아낸다.



7. Encoder의 Self-attention :2) Q, K, V 벡터 얻기

Transformer에서 Self-Attention의 동작 메커니즘을 알아보자.

앞에서 self-attention은 입력 문장의 단어 벡터들로 수행한다고 하였는데,

사실 Self-Attention은 초기 입력인 d_{model}의 차원을 가지는 단어벡터들을 사용하여 바로 self-attention을 수행하는 것이 아니라

각 단어들로부터 Q벡터, K벡터, V벡터를 얻는 작업을 먼저 거친다.

- 이때 만들어진 **Q벡터, K벡터, V벡터들**은 초기 입력인 d_{model}의 차원을 가지는 단어벡터들보다 **더 작은 차원을 가진다.**
- 논문에서는 $d_{model} = 512$였던 단어벡터들을 -> **64차원을 가지는 Q벡터, K벡터, V벡터로 변경**하였다.
    - 64라는 값은 Transformer의 하이퍼파라미터인 **num_heads**에 의해 결정되며,

      Transformer는 d_{model}을 num_heads로 나눈 값을 각 Q벡터, K벡터, V벡터의 차원으로 결정한다

      논문에서는 num_heads=8로 설정되었다.(512/8=64=Q벡터, K벡터, V벡터의 차원)

7. Encoder의 Self-attention :2) Q, K, V 벡터 얻기

그림을 통해 이해해보자.

student라는 단어의 임베딩벡터를 -> Q, K, V의 벡터로 변환하는 과정을 살펴보자.

기존의 벡터로부터 **더 작은 벡터를 만들려면 가중치 행렬을 곱하면 된다.**

각 가중치 행렬은 $d_{model} \times (d_{model}/num_heads)$의 크기를 가진다.

논문과 같이 $d_{model} = 512, num\_heads$=8이라면

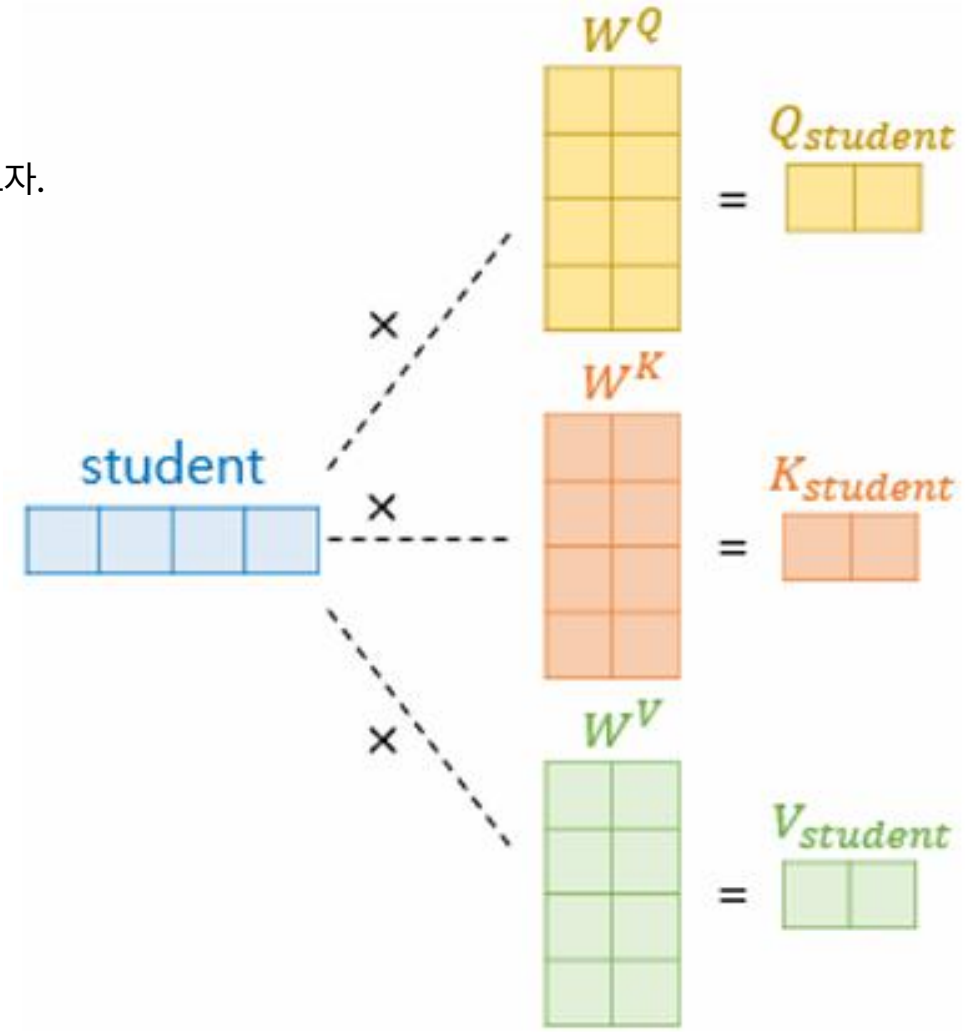
각 벡터에 **3개의 다른 가중치 행렬**(512 x 64)을 곱하고

64의 크기(= (d_{model}/num_heads)를 가지는 **Q, K, V벡터를 얻어낸다.**

(임베딩 벡터의 차원이 d_{model}이므로 (1 x 512) · (512 x 64) = (1 x 64))

모든 단어 벡터에 위와 같은 과정을 거치면

모든 단어에 대해서 각각의 Q, K, V 벡터를 얻게 된다.



7. Encoder의 Self-attention :3) Scaled dot-product Attention

Q,K,V벡터를 얻었다면 기존에 배운 Attention 메커니즘과 동일하다.

각 Q 벡터는 모든 K 벡터에 대해서 Attention Score를 구하고, (softmax function으로) Attention Distribution을 구한 뒤에 이를 사용하여 모든 V 벡터를 가중합하여 Attention Value를 구하게 된다. 이후, 모든 Q 벡터에 대해서 반복하면 된다.

다만, Transformer에서는 Attention score function으로 내적만을 사용하는 $score(q,k) = q \cdot k$를 사용하는 것이 아니라 **특정 값으로 나눠준 score function을 사용**한다.

-> $score(q,k) = q \cdot \frac{k}{\sqrt{n}}$

이러한 function을 사용하는 Attention을

dot-product Attention에서 값을 scaling하는 것을 추가하였다고 하여 **Scaled dot-product Attention**이라고 한다.

7. Encoder의 Self-attention :3) Scaled dot-product Attention

단어 'I'에 대한 Q벡터를 기준으로 설명해보자.(지금부터 설명하는 과정은 입력 문장의 모든 단어의 Q벡터에 대해 동일한 과정을 거친다고 가정한다)

아래 그림은 **단어 'I'에 대한 Q 벡터가 모든 K벡터에 대해서 Attention score를 구하는 모습**을 보여준다.(K벡터를 transpose하여 곱했다.)
(128과 32는 임의로 가정한 수치로, 신경쓰지 않아도 된다.)

각각의 Attention Score는 **단어 I가 단어 I, am, a, student와**

얼마나 연관되어있는지를 보여주는 수치이다.

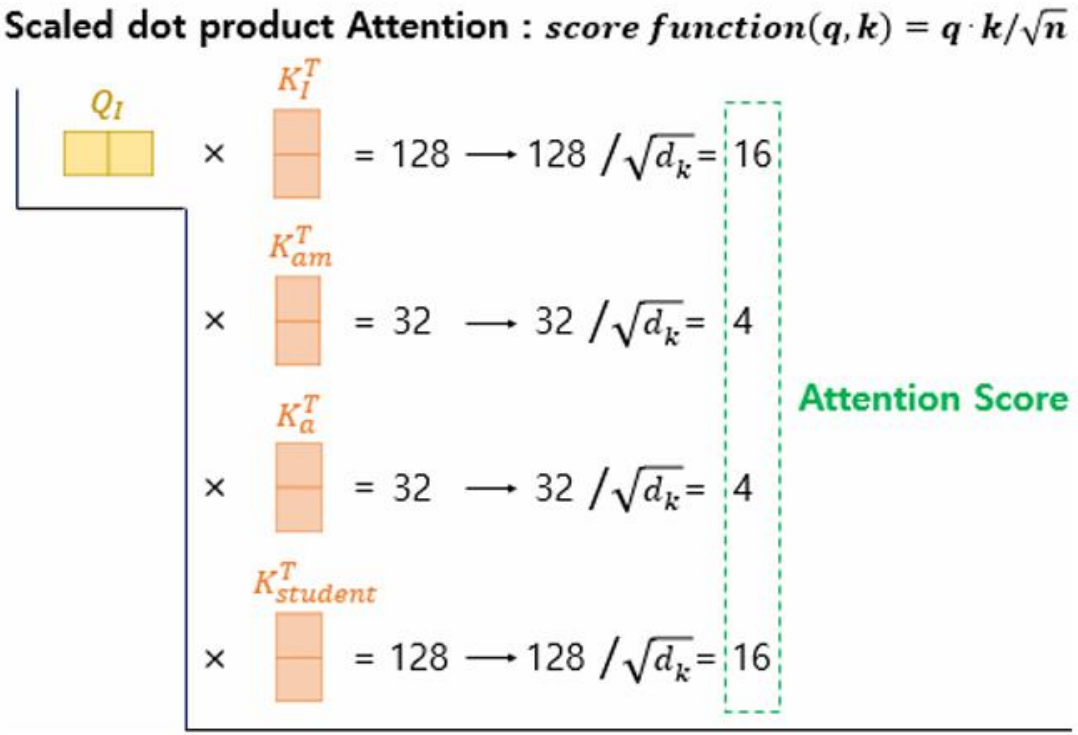
Transformer에서는 두 벡터의 내적값을 scaling하는 값으로

K벡터의 차원을 나타내는 d_k에 루트를 씌운 -> $\sqrt{d_k}$를 사용하였다.

앞에서 보았듯이 논문에 따르면

$d_k = (d_{model}/num\_heads)$ = 64이므로

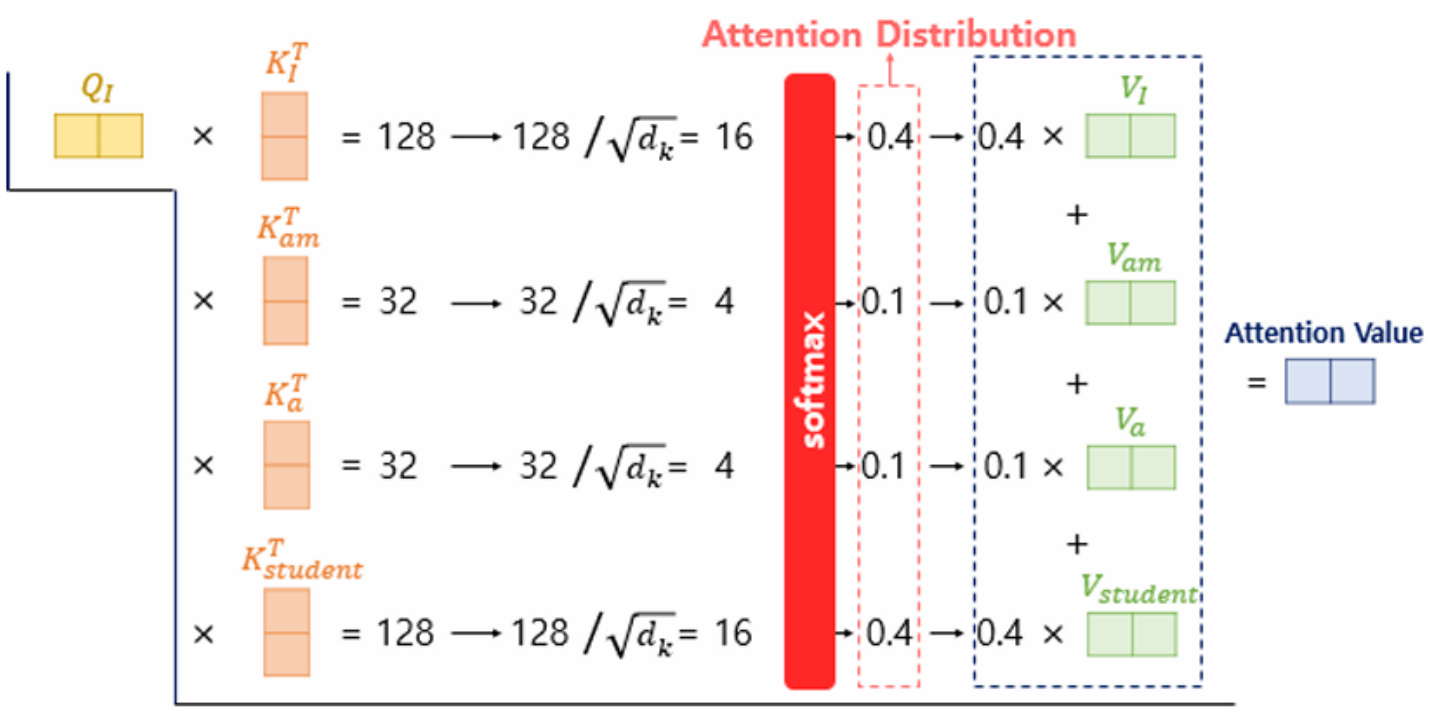
$\sqrt{d_k} = 8$이다.



7. Encoder의 Self-attention
:3) Scaled dot-product Attention

이렇게 구한 Attention Score에 softmax 함수를 사용하여 Attention Distribution을 구하고 V벡터와 가중합하여 Attention Value를 구한다.

- 이렇게 구한 Attention Value는 **단어 'I'에 대한 Attention Value 또는 단어 I에 대한 Context Vector**라고 할 수 있다.
- 이후, 'am', 'a', 'student'에 대한 Q벡터에 동일한 과정을 반복하여 각각의 Attention Vaue를 거치면 된다.

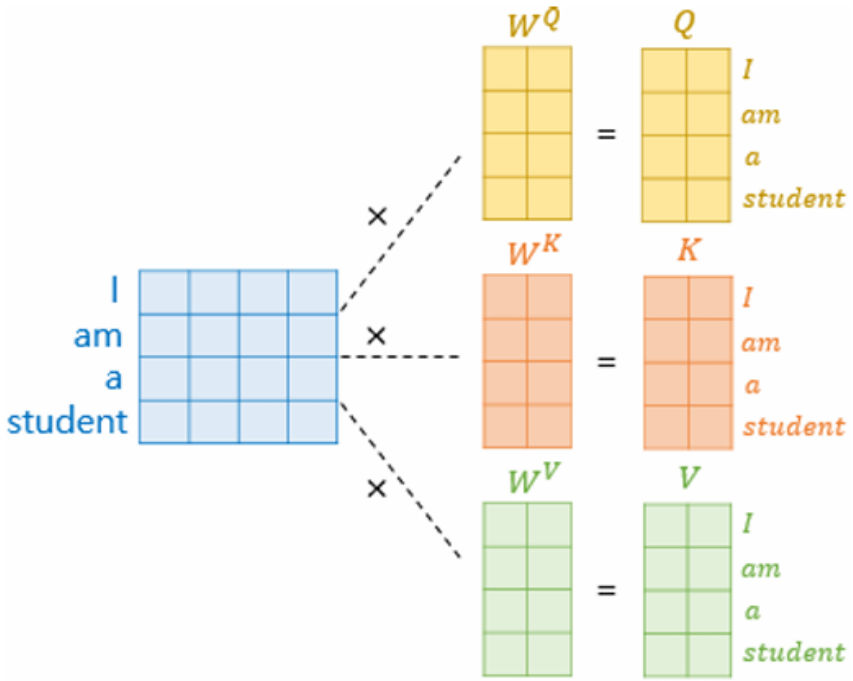


7. Encoder의 Self-attention :4) 행렬 연산으로 일괄 처리하기

다만, 굳이 Q벡터마다 일일이 따로 연산을 해야할 필요가 있을까?

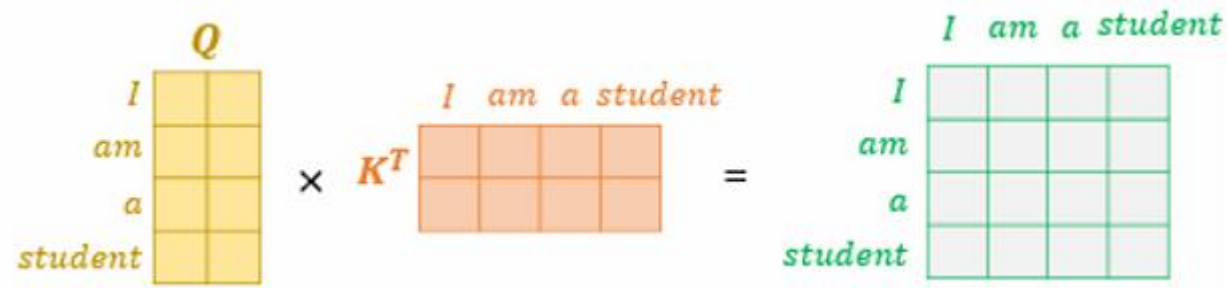
➔ 벡터 연산이 아니라 행렬 연산을 사용하면 일괄 계산이 가능함.

우선 각 단어벡터마다 일일이 가중치 행렬을 곱하는게 아니라 **문장 행렬**에 가중치 행렬을 곱해서 Q행렬, K행렬, V행렬을 구한다.



7. Encoder의 Self-attention
:4) 행렬 연산으로 일괄 처리하기

이후, Attention score를 구할 때 Q 행렬을 K행렬을 Transpose한 행렬과 곱해주면 **Q벡터와 K 벡터의 내적을 원소로 가지는** 행렬이 결과로 나옴



결과 행렬의 값에 전체적으로 $\sqrt{d_k}$를 나누어주면 각 행과 열이 Attention Score를 가지는 행렬이 됨

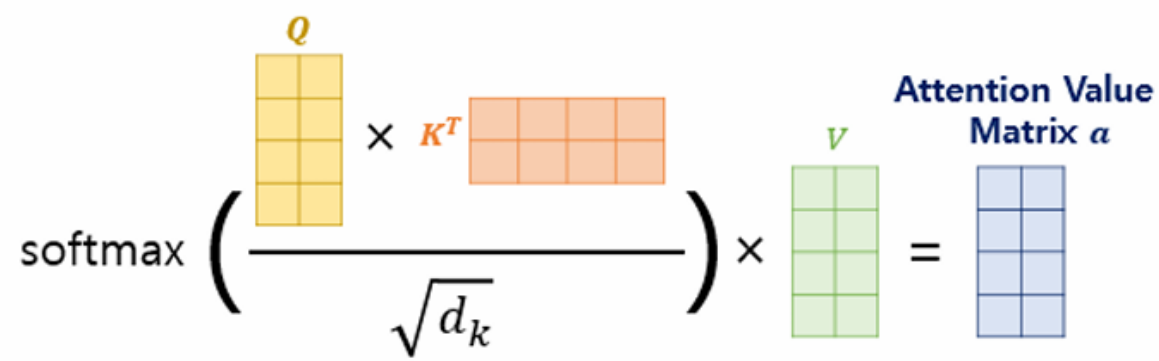
➔ Ex. 'I'행과 'student'열의 값은 'I'의 Q벡터와 'student'의 K벡터의 Attention Score값

즉, 결과 행렬에 $\sqrt{d_k}$를 나누어준 행렬을 **Attention Score 행렬**이라고 할 수 있음

7. Encoder의 Self-attention :4) 행렬 연산으로 일괄 처리하기

Attention Score 행렬을 구했다면 -> Attention Distribution을 구하고, 모든 단어에 대한 Attention Value를 구하면 됨

즉, **Attention Score행렬에 Softmax 함수**를 사용하고, **V행렬을 곱하면 각 단어의 Attention Value를 모두 가지는 Attention Value 행렬**이 결과로 나옴



위의 그림은 행렬 연산을 통해 모든 값이 일괄 계산되는 과정을 식으로 보여주며,

이 식은 실제 Transformer 논문에 기재된 아래의 수식과 정확하게 일치함

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

7. Encoder의 Self-attention :4) 행렬 연산으로 일괄 처리하기

행렬 연산에 사용된 행렬의 크기를 정리해보자.

입력문장의 길이(단어의 개수)를 seq_len이라고 하면 **문장 행렬의 크기**는 (seq_len, d_{model})

- 이 문장 행렬에 3개의 가중치 행렬을 곱해서 Q, K, V행렬을 만들어야함

Q, K, V 행렬

Q, K, V행렬의 각 행에 해당되는 **Q벡터와 K벡터의 차원을 d_k**라고 하고, **V벡터의 차원을 d_v**라고 해보자

- 논문에서 $d_k = d_v = d_{model}/num\_heads$

Q행렬의 크기 = K행렬의 크기: (seq_len, d_k)

V행렬의 크기: (seq_len, d_v)

가중치 행렬

Q, K, V행렬의 크기로부터 **가중치 행렬의 크기 추정이 가능해진다.**

-> W^Q, W^K의 크기: (d_{model}, d_k)

-> W^V의 크기 : (d_{model}, d_v)

$softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$ 식을 적용한 **Attention Value 행렬 a의 크기**: **(seq_len, d_v)**

7. Encoder의 Self-attention :5) Multi-head Attention

앞서 배운 Attention에서는 d_{model}의 차원을 가진 단어 벡터를 num_heads로 나눈 차원을 가지는 Q, K, V 벡터로 바꾸고 Attention을 수행하였다.

num_heads의 의미와

왜 d_{model}의 차원을 가진 단어벡터를 가지고 Attention을 수행하지 않고 **차원을 축소시킨 벡터로 Attention을 수행하였을까?**

Transformer 연구진은 한번의 Attention을 하는것보다 **여러 번의 Attention을 병렬로 사용하는 것이 더 효과적이라고 판단**하였다.

따라서 d_{model}의 차원을 num_heads개로 나누어 d_{model}/num_heads의 차원을 가지는 Q,K,V에 대해서

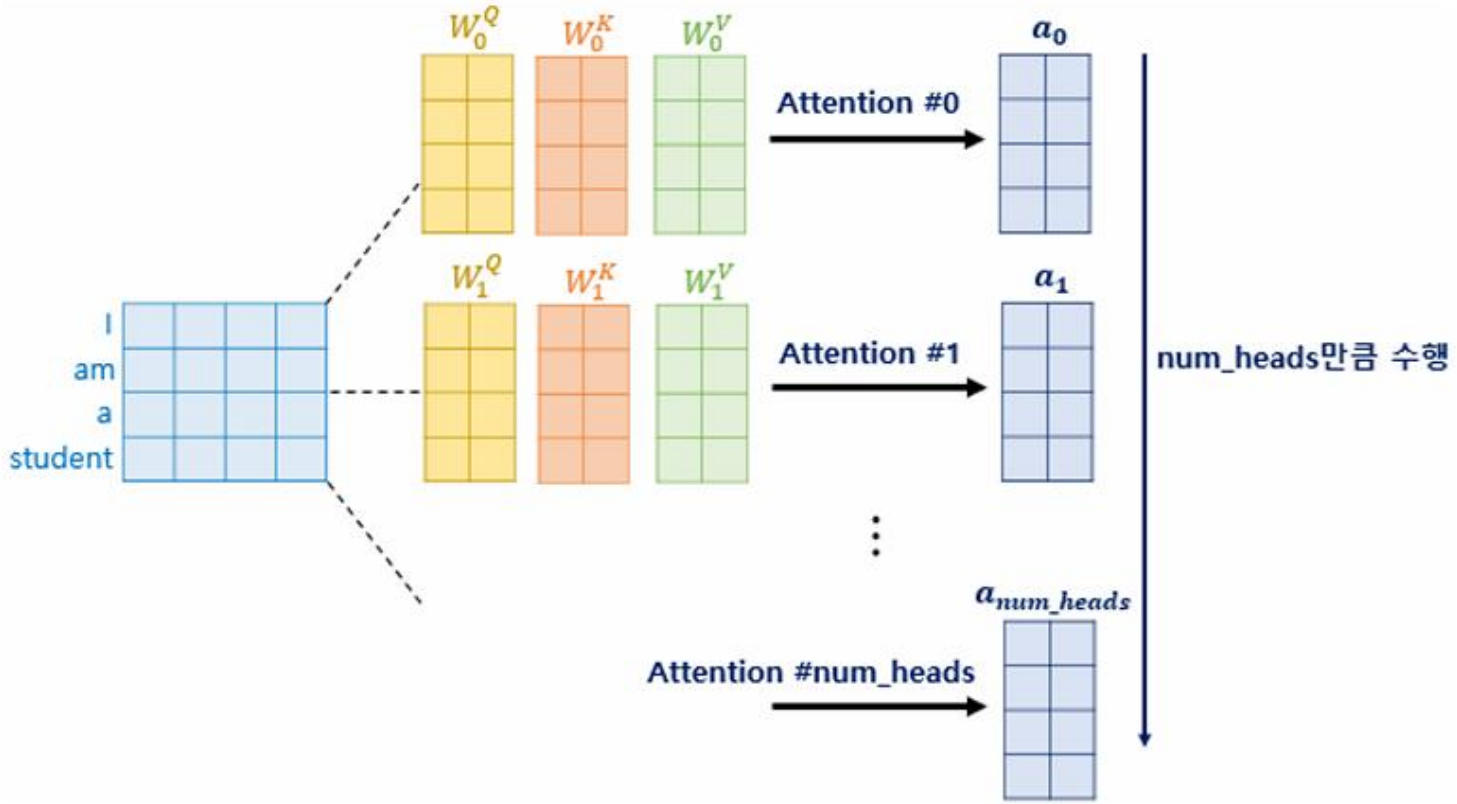
num_heads개의 병렬 Attention을 수행한다.

- 논문에서는 num_heads의 값을 8로 지정하였기 때문에 8개의 병렬 Attention을 수행한다.

7. Encoder의 Self-attention :5) Multi-head Attention

즉, 앞서 설명한 Attention이 8개로 병렬로 이루어지게 되는데, 이때 **각각의 Attention Value 행렬을 Attention head**라고 부른다.

- 가중치 행렬 W^Q, W^K, W^V 의 값은 8개의 Attention head마다 전부 다르다.



7. Encoder의 Self-attention :5) Multi-head Attention

Multi-head Attention을 수행하면 여러 시각으로 정보를 수집할 수 있다.

Example

'그 동물은 길을 건너지 않았다. 왜냐하면 그것은 너무 피곤하였기 때문이다.' 라는 예문을 들어 이해해보자.

단어 '그것(it)'이 Query였다고 해보자. 'it'에 대한 Q벡터로부터 다른 단어와의 연관도를 구하였을 때 아래와 같은 상황이 발생할 수 있다.

1번째 Attention-head: 'it'과 '동물(animal)'의 연관도를 높게본다

2번째 Attention-head: 'it'과 '피곤하였기 때문이다(tired)'의 연관도를 높게본다.

-> 이는 각 Attention-head가 전부 다른 시각에서 볼 수 있기 때문이다.

7. Encoder의 Self-attention :5) Multi-head Attention

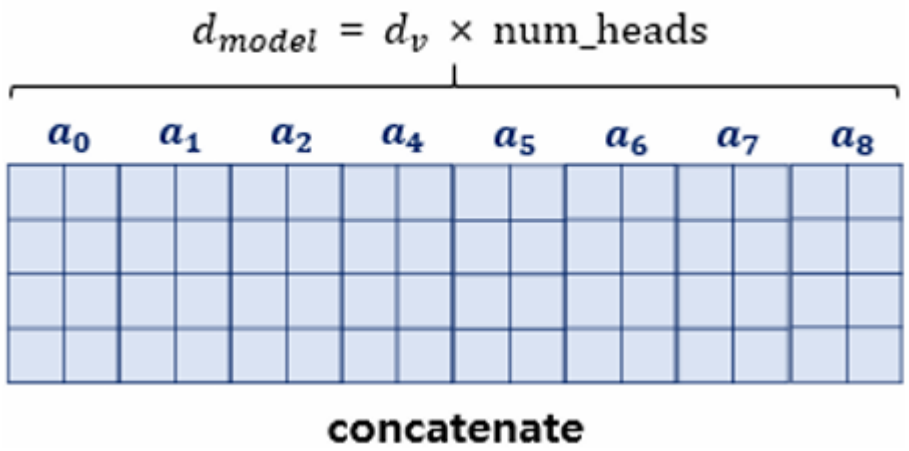
1개의 Attention head의 크기: $softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$ 식을 적용한 **Attention Value 행렬 a의 크기**: **(seq_len, d_v)**

병렬 Attention을 모두(num_heads만큼) 수행하였다면 모든 Attention head를 concatenate한다.

➔ 모두 연결된 Attention head 행렬의 크기는 **(seq_len, d_{model})**이 된다.

지면상의 한계로 4차원을 $d_{model} = 512$로 표현하고, 2차원을 $d_v = 64$로 표현해왔지만,

아래 그림에서만 8개의 Attention head의 concatenate 과정의 이해를 위해 d_{model}의 크기를 d_v의 8배인 16차원으로 하여 표현하였다.

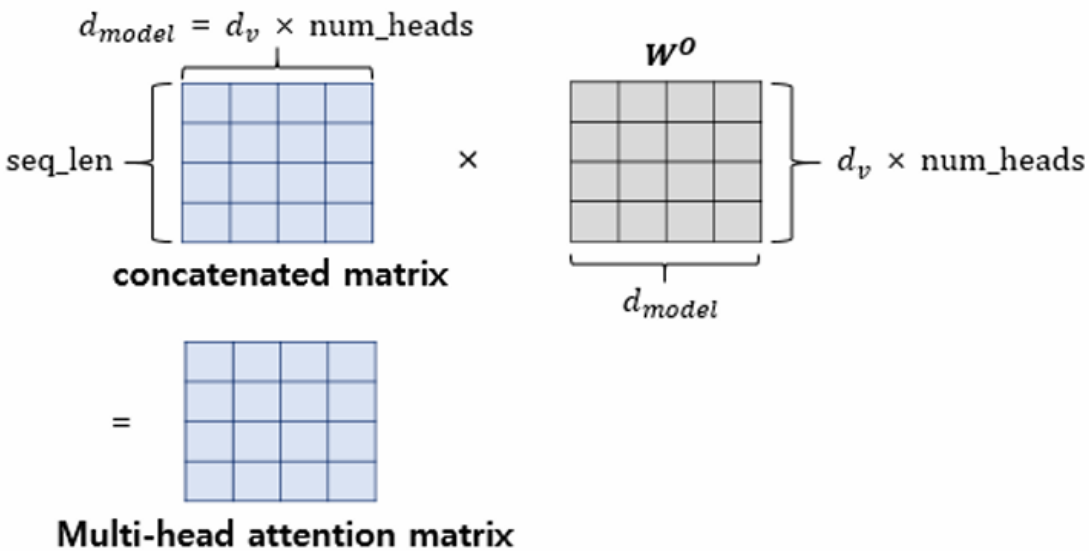


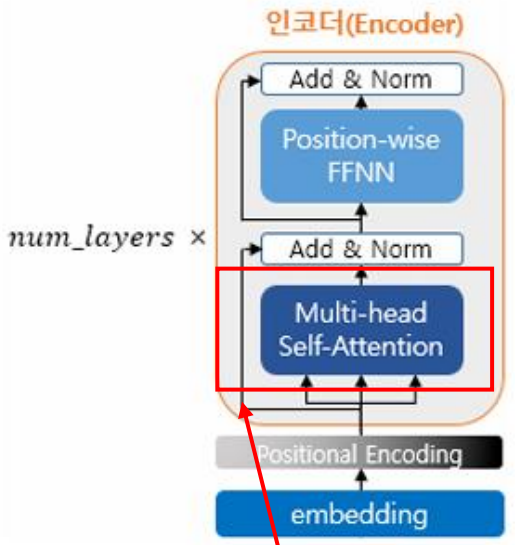
7. Encoder의 Self-attention :5) Multi-head Attention

다시 d_{model}을 4차원으로 표현한다.

아래 그림과 같이 Attention head를 모두 연결한 행렬은 또다른 가중치 행렬 W^o를 곱하게 되며,

이렇게 나온 결과 행렬이 Multi-head Attention의 최종 결과값이다.





이 때 결과물인 **Multi-head Attention 행렬의 크기**는 Encoder의 입력이었던 문장 행렬의 크기인 **(seq_len, d_{model})**과 동일하다.

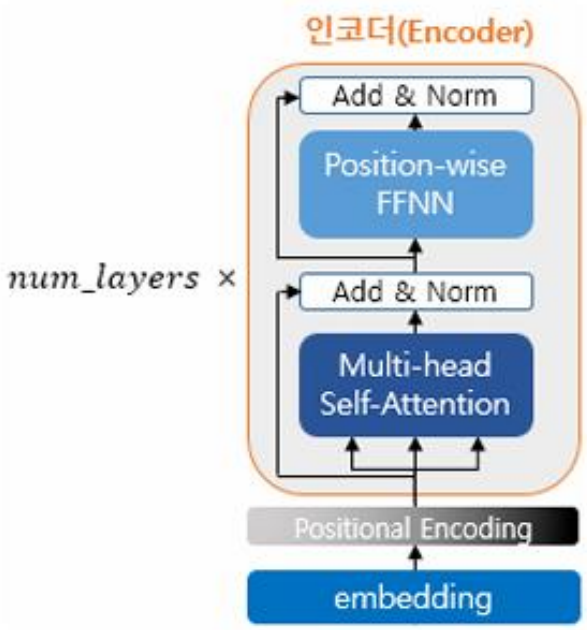
-> 즉, Encoder의 첫번째 서브층인 **Multi-head Attention 단계를 끝마쳤을 때**, Encoder의 입력으로 들어왔던 행렬의 크기가 유지된다.

7. Encoder의 Self-attention :5) Multi-head Attention

1번째 서브층인 Multi-head Attention과 2번째 서브층인 Position-wise Feed Forward Neural Network를 지나면서

Encoder의 입력으로 들어올 때의 행렬의 크기는 계속 유지되어야한다.

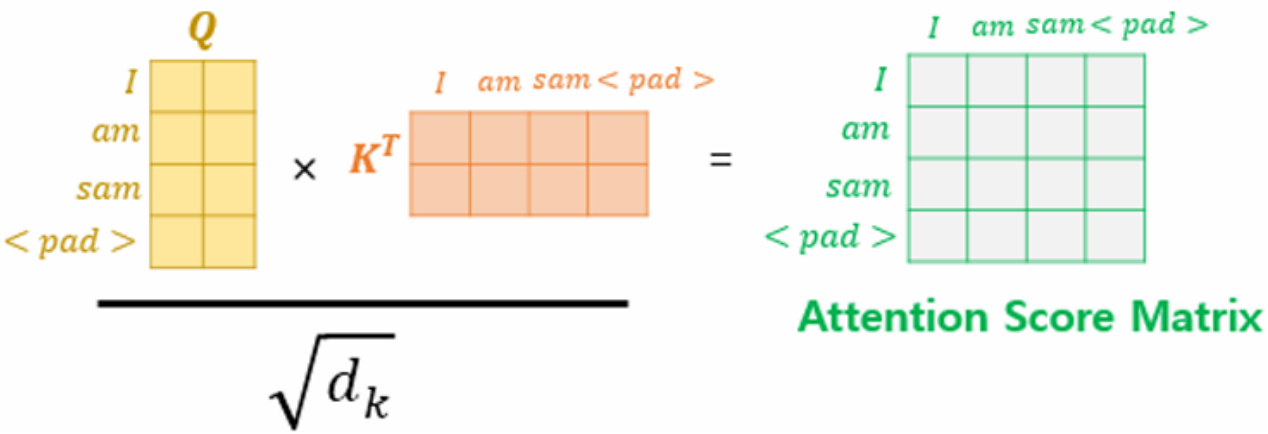
- Transformer는 동일한 구조의 Encoder를 쌓은 구조이기 때문에, Encoder의 입력의 크기가 출력에서도 동일 크기로 유지되어야만 다음 Encoder에서도 다시 입력이 될 수 있기 때문이다.
- 논문 기준으로 Encoder는 총 6개이다.



7. Encoder의 Self-attention :6) 패딩 마스크(Padding Mask)

Padding mask: 입력문장에 <PAD> 토큰이 있을 경우 Attention에서 제외하기 위한 연산이다.

Transformer에서 **<PAD>가 포함된 입력 문장**에 self-attention을 수행하고 Attention Score 행렬을 얻는 과정은 아래와 같다.



단어 <PAD>의 경우에는 실질적인 의미를 가진 단어가 아니므로,

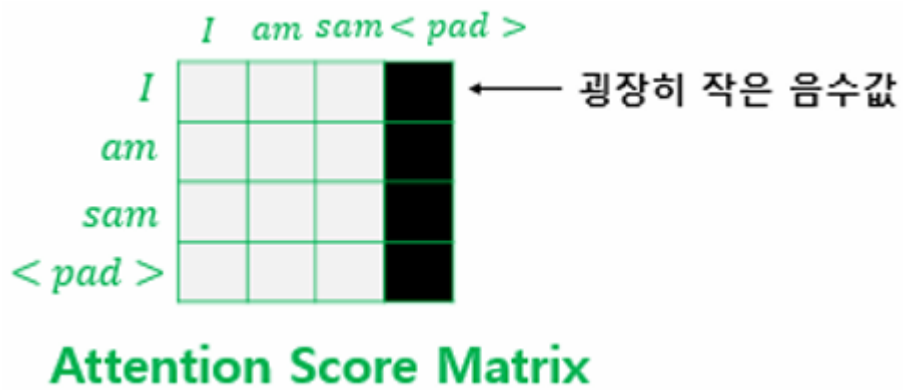
Transformer에서는 **Key에 <PAD> 토큰이 존재한다면 이에 대해서는 유사도를 구하지 않도록 Masking**을 해주기로 하였다.

- Masking: Attention에서 제외하기 위해 값을 가린다는 의미

7. Encoder의 Self-attention :6) 패딩 마스크(Padding Mask)

Attention Score 행렬에서 **행에 해당하는 문장은 Query**이고, **열에 해당하는 문장은 Key**이다.

즉, **Key에 <PAD>가 있는 경우는 -> 해당 열 전체를 masking**해주어야한다.



Masking을 하는 방법은 Attention Score 행렬의 Masking 위치에 매우 작은 음수값(-무한대에 가까운 수)를 넣어주는 것이다.

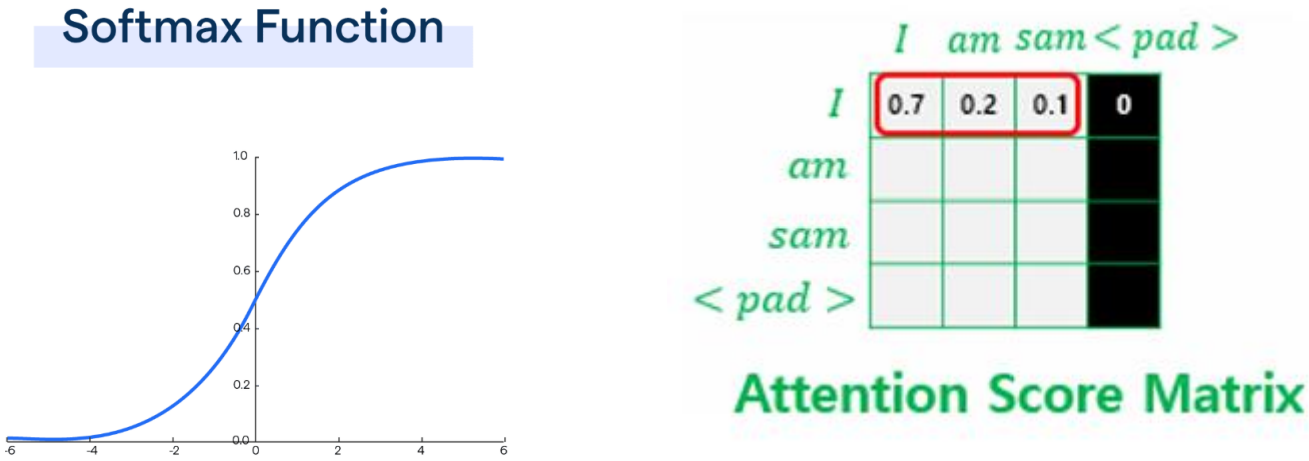
- 매우 작은 음수값을 넣음으로서 Softmax함수의 성질을 이용하여 masking할 수 있다.
- 현재 Attention Score 행렬은 Softmax함수를 지나지 않은 상태이다.

7. Encoder의 Self-attention :6) 패딩 마스크(Padding Mask)

Attention의 연산순서: Attention Score 행렬이 -> softmax함수를 지나고 -> Value행렬과 곱해져야한다.

현재 Masking 위치에 **매우 작은 음수값**이 들어가 있으므로,

Attention Score행렬이 softmax함수를 지난 후에는 **해당 위치의 값이 0이되어**, 단어간 유사도를 구하는 일에 <PAD> 토큰이 반영되지 않게 된다.



<Softmax함수를 지난 Attention Score Matrix>

Softmax함수를 지나면 각 행의 Attention weight의 총합은 1이 되는데, 단어 <PAD>의 경우에는 0이 되어 어떤 유의미한 값을 가지지 않게 된다.

8. Position-wise FFNN

Position-wise FFNN(Feed Forward Neural Network)은 **Encoder와 Decoder에서 공통적으로 가지고 있는 서브층**이다.

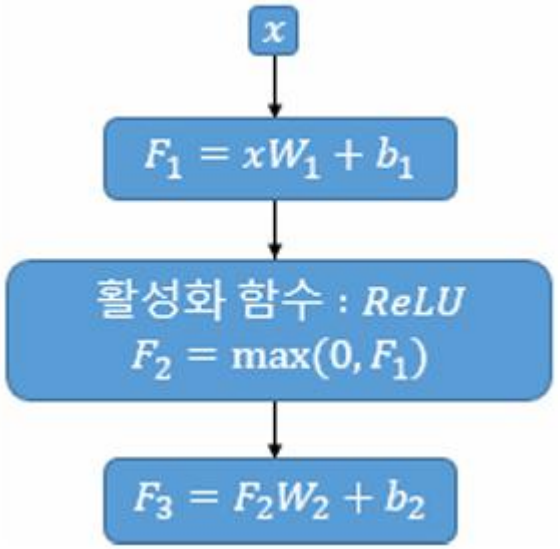
- 쉽게 말해서 Position-wise FFNN은 -> Fully-connected FFNN으로 해석할 수 있다.

아래는 Position-Wise FFNN의 수식이다.

-> $FFNN(x) = MAX(0, xW_1 + b_1)W_2 + b_2$

식을 그림으로 표현하면 오른쪽과 같다.

- x: Multi-head Attention의 결과로 나온 (seq_len, d_{model}) 크기의 행렬
- W_1: (d_{model}, d_{ff})의 크기
  - d_{ff}: 은닉층의 크기로, 2048의 크기를 가지는 하이퍼파라미터
- W_2: (d_{ff}, d_{model})의 크기



매개변수 W_1, b_1, W_2, b_2는 **하나의 Encoder 층 내에서는 다른 문장, 다른 단어들이더라도 동일하게 사용**된다.

- 하지만 Encoder 층마다는 다른 값을 가진다.

8. Positon-wise FFNN

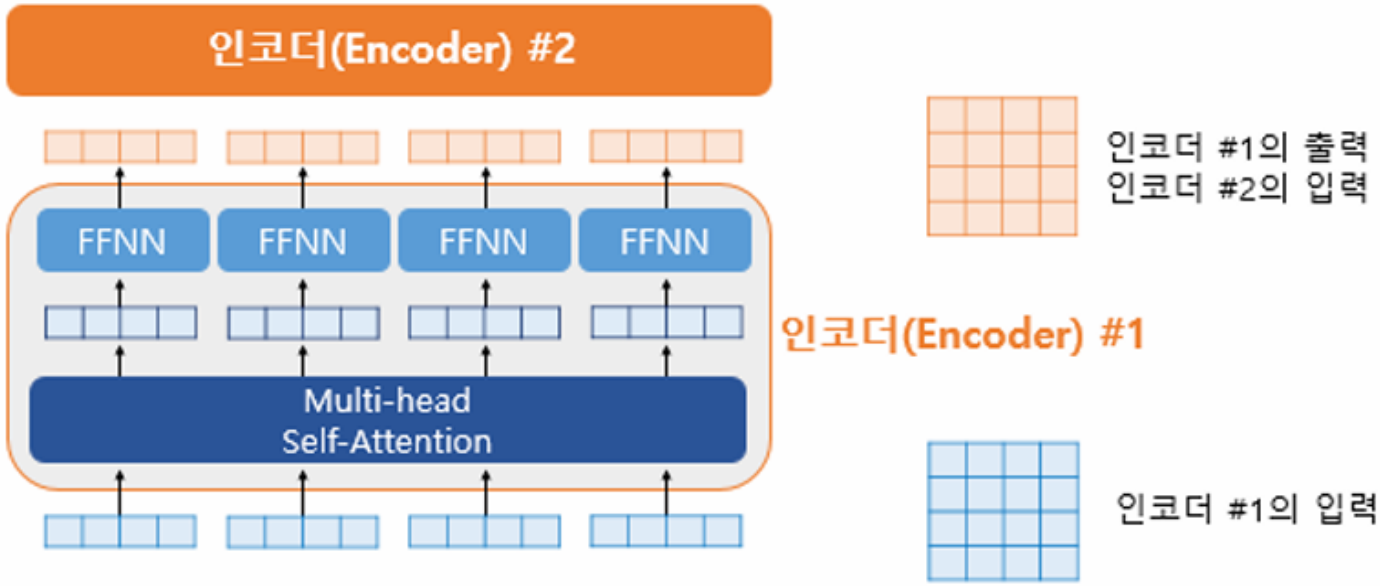
아래의 그림에서 좌측은 Encoder의 입력을 벡터 단위로 보았을 때,

각 벡터들이 **Multi-head Attention 층이라는 Encoder의 1번째 서브층을 지나 2번째 서브층인 Position-wise FFNN을 통과하는 것**을 보여준다.

실제로는 그림의 우측과 같이 행렬로 연산되는데,

2번째 서브층을 지난 Encoder의 최종 출력은 Encoder의 입력의 크기였던 (seq_len, d_{model})의 크기로 보존되고 있다.

- 하나의 Encoder 층을 지난 이 행렬은 다음 Encoder 층으로 전달되고, 다음 층에서도 동일한 Encoder 연산이 반복된다.



9. Residual connection과 Layer Normalization

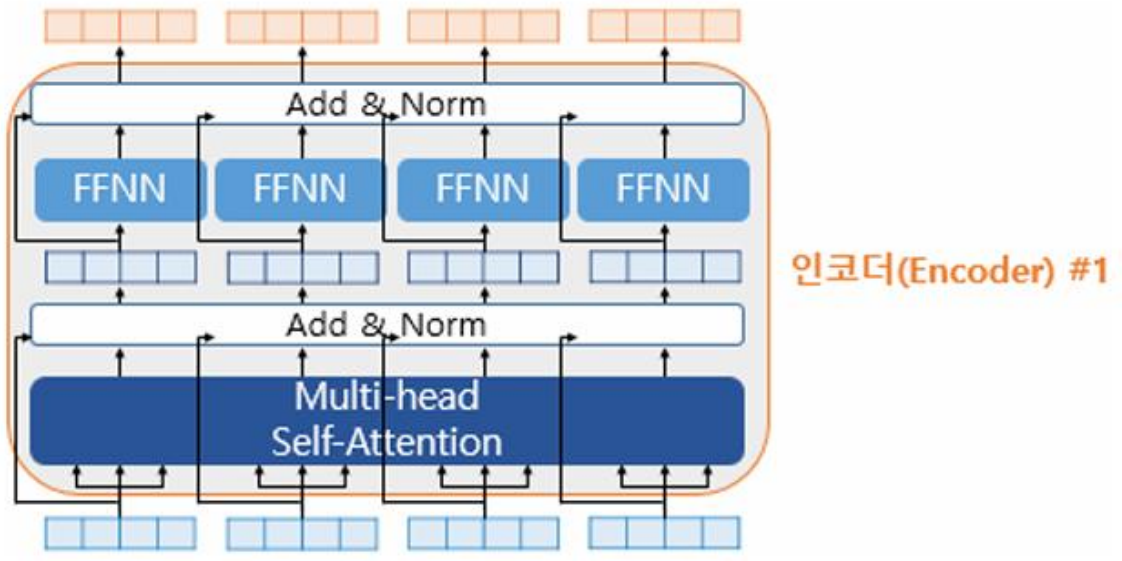
Transformer에서는 이러한 2개의 서브층을 가진 Encoder에 추가적으로 **Add & Norm 기법**,

즉, **residual connection**과 **layer normalization**을 사용한다.

아래 그림은 Position-wise FFNN을 설명할 때 사용한 그림에서

화살표와 Add & Norm(residual connection과 layer normalization)을 추가한 그림이다.

- **추가된 화살표들이 서브층 이전의 입력에서 시작되어 서브층의 출력 부분으로 향하고 있음**에 주목하자.



9. Residual connection과 Layer Normalization: 1) Residual Connection

우측 하단의 그림은 입력 x와 x에 대한 어떤 함수 $F(x)$의 값을 더한 $H(x)$의 구조를 보여준다.

- 어떤 함수 $F(x)$는 **Transformer에서의 서브층**을 의미한다.

Residual Connection: 서브층의 입력과 출력을 더하는 것

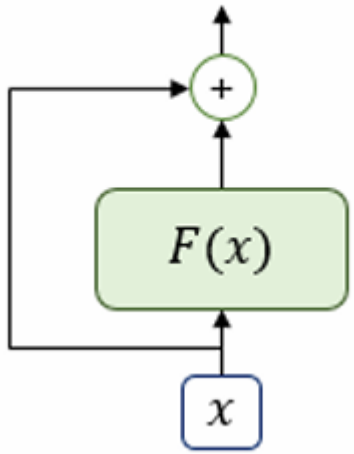
Transformer에서 서브층의 입력과 출력은 동일한 차원을 가지고 있으므로, 서브층의 입력과 서브층의 출력은 **덧셈 연산이 가능**하다.

- Encoder 그림에서 각 화살표가 서브층의 입력에서 출력으로 향하도록 그려졌던 이유이다.
- Residual Connection은 컴퓨터 비전분야에서 주로 사용되는 모델의 학습을 돕는 기법이다.

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

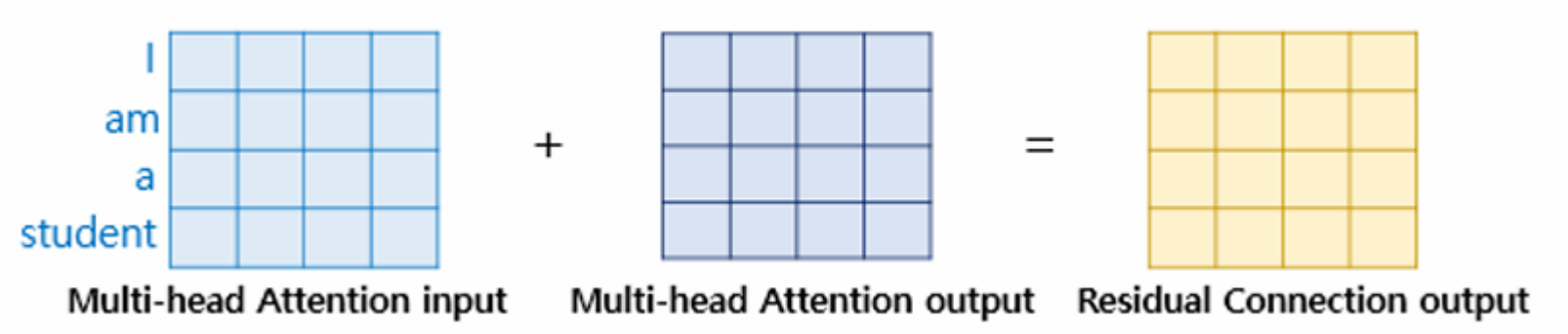
이를 식으로 표현하면 => $x + Sublayer(x)$이다.

- 서브층이 Multi-head Attention이었다면 Residual Connection은 다음과 같다.
- $H(x) = x + Multi - head\ Attention(x)$



9. Residual connection과 Layer Normalization: 1) Residual Connection

아래 그림은 Multi-head Attention의 입력과 Multi-head Attention의 결과가 더해지는 과정을 보여준다.



9. Residual connection과 Layer Normalization: 2) Layer Normalization

Residual Connection을 마친 결과는 이어서 **Layer Normalization(층 정규화)**과정을 거친다.

Residual Connection의 입력을 : x,

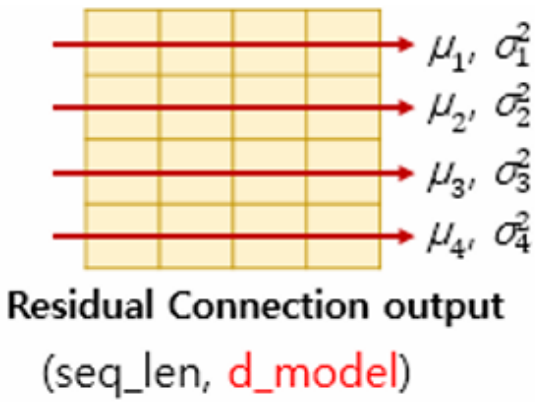
Residual Connection과 Layer Normalization **2가지 연산을 모두 수행한 후의 결과 행렬**을 : LN이라고 하였을 때,

Residual Connection 후의 Normalization 연산을 수식으로 표현하면 아래와 같다.

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

Layer Normalization: **Tensor의 마지막 차원에 대해서** 평균과 분산을 구하고, 이를 가지고 어떤 수식을 통해 값을 정규화하는 것

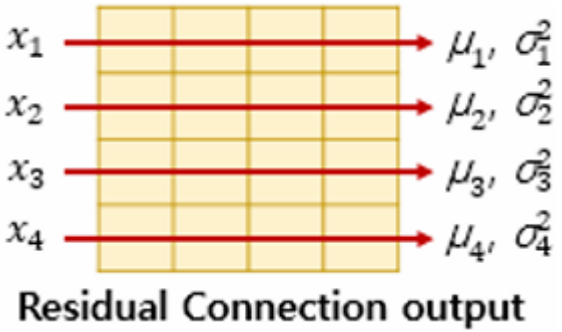
- Layer Normalization을 수행함으로서 학습을 도울 수 있다.
- Tensor의 마지막 차원이란 Transformer에서는 d_{model} 차원을 의미한다.
- 우측의 그림에서는 d_{model}의 차원의 방향을 화살표로 표현하였다.



9. Residual connection과 Layer Normalization: 2) Layer Normalization

Layer Normalization을 위해 우선 **화살표 방향으로 각각 평균 μ와 σ^2을 구한다.**

각 화살표 방향의 벡터를 x_i라고 해보자.



Layer Normalization을 수행한 이후에는 벡터 x_i가 → $\mathbf{ln}_i$ **라는 벡터로 정규화된다**.

$$\mathrm{ln}_i = LayerNorm(x_i)$$

9. Residual connection과 Layer Normalization: 2) Layer Normalization

Layer Normalization 수식($LayerNorm$)을 알아보자. $\ln_i = LayerNorm(x_i)$

Layer Normalization을 2가지 과정으로 나누어 설명할 것이다.

1. 평균과 분산을 통한 x_i 정규화
2. 감마와 베타를 도입

1. 평균과 분산을 통한 x_i 정규화

우선, 평균과 분산을 통해 x_i를 정규화해준다.

- **x_i는 벡터**인 반면, **평균 μ_i와 분산 σ_i^2은 스칼라**이다.

x_i의 각 차원을 k라고 하였을 때, $x_{i,k}$는 아래 수식과 같이 정규화할 수 있다.(벡터 x_i의 각 k차원의 값이 아래와 같이 정규화되는 것이다.)

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

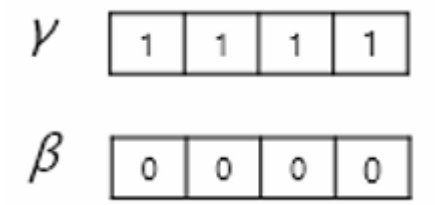
- ϵ: 분모가 0이되는 것을 방지하는 값

9. Residual connection과 Layer Normalization: 2) Layer Normalization

2. 감마와 베타를 도입

이후, γ(감마)와 β(베타)라는 벡터를 준비한다.

- γ(감마)의 초깃값은 1, β(베타)의 초깃값은 0이다.



γ(감마)와 β(베타)를 도입한 Layer Normalization의 최종 수식은 아래와 같으며, γ(감마)와 β(베타)는 학습 가능한 파라미터이다.

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

코드에서는 Keras에서 Layer Normalization을 위한 **LayerNormalization()**를 제공하므로, 이를 가져와 사용한다.

10. From Encoder to Decoder

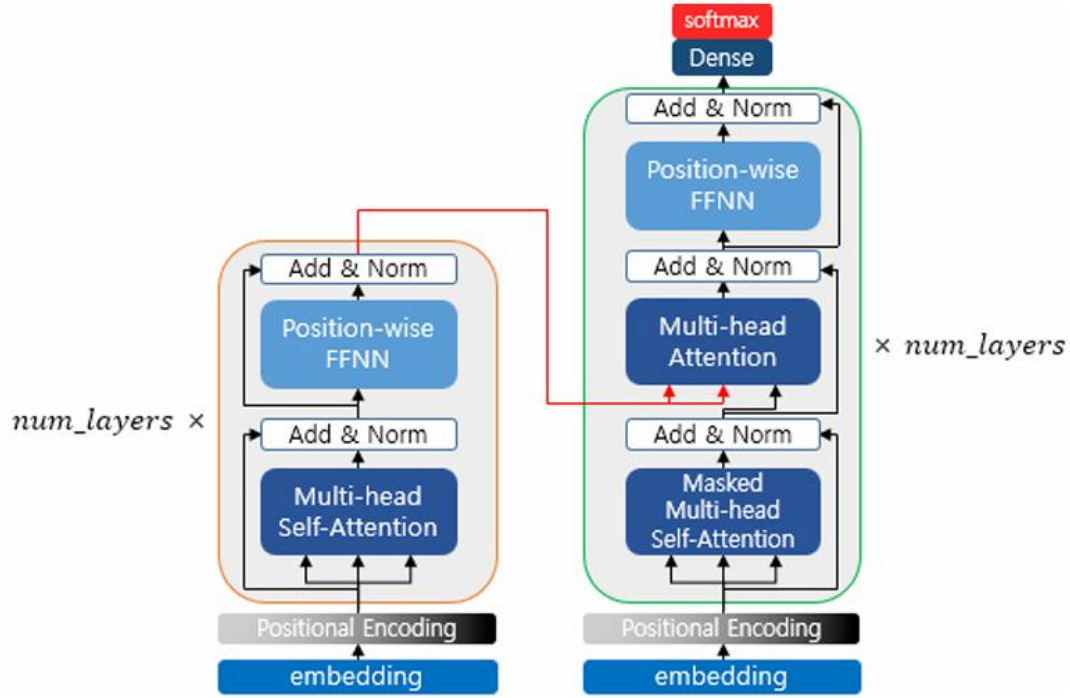
Encoder에 대해서 정리해보았다.

- 이렇게 구현된 Encoder는 총 num_layers 만큼의 층 연산을 순차적으로 한후, **마지막 층의 Encoder 출력을 Decoder에게 전달**한다.

Encoder 연산이 끝났다면 Decoder 연산이 시작되어 **Decoder 또한 num_layers 만큼의 연산**을 하는데,

이때마다 **Encoder가 보낸 출력을 각 층의 Decoder 연산에 사용**한다.

Decoder에 대해서 이해해보자.



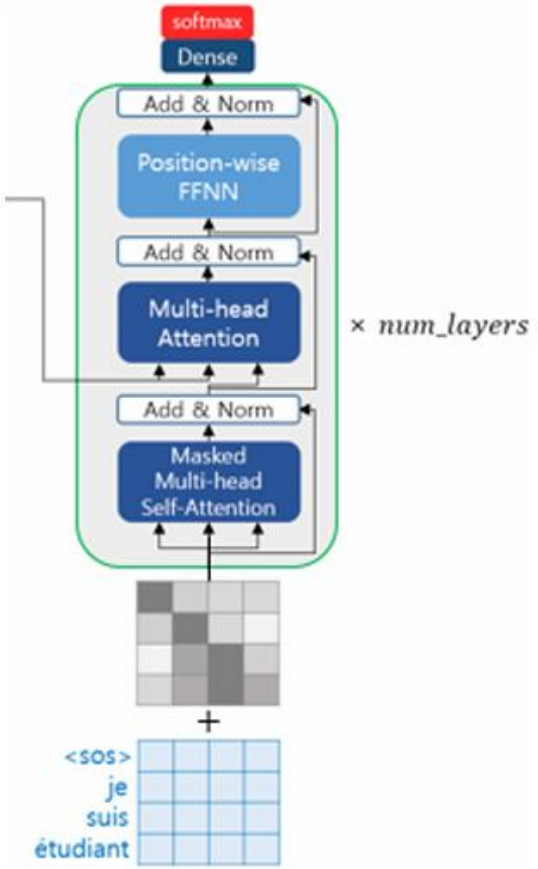
11. Decoder의 첫번째 서브층 : Self-Attention과 look-ahead mask

Decoder도 Encoder와 동일하게 입력으로 **Embedding 층과 Positional Encoding을 거친 후의 문장 행렬**이 사용된다.

Transformer도 Seq2Seq와 마찬가지로 Teacher Forcing(교사강요)를 사용하여 훈련되므로,

Decoder는 번역할 문장에 해당되는 <sos> je suis étudiant의 **문장 행렬을 한번에 입력받는다.**

- 이후, Decoder는 이 문장행렬로부터 각 시점의 단어를 예측하도록 훈련된다.



11. Decoder의 첫번째 서브층 : Self-Attention과 look-ahead mask

여기서 **문제**가 발생한다.

Seq2seq의 Decoder에 사용되는 RNN 계열의 신경망은 입력단어를 매 시점마다 순차적으로 입력받으므로,

다음 단어 예측에 현재 시점을 포함한 이전 시점에 입력된 단어들만 참고할 수 있다.

반면, **Transformer는 문장 행렬로 입력을 한번에 받으므로** 현재 시점의 단어를 예측하고자 할 때,

입력 문장 행렬로부터 미래 시점의 단어까지도 참고할 수 있게 된다.

Ex. suis를 예측해야하는 시점이라고 하면,

- RNN 계열의 seq2seq: 현재까지 Decoder에 <sos>와 je만 입력받음
- Transformer: je suis étudiant를 입력받음

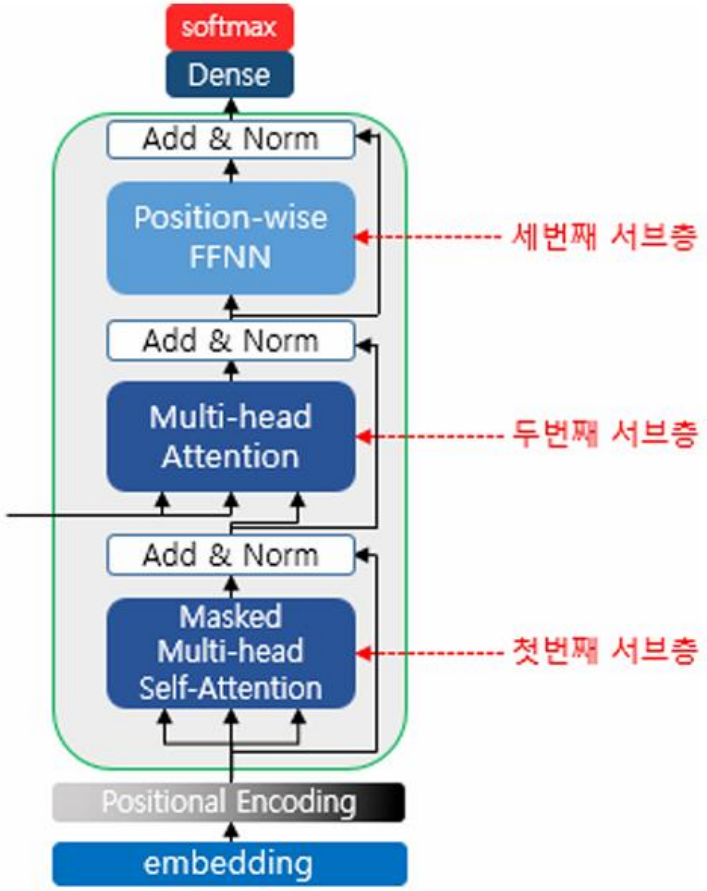
이 때문에 Transformer의 **Decoder에서는** 현재 시점의 예측에서 현재시점보다 미래에 있는 단어들을 참고하지 못하도록

-> look-ahead mask를 도입하였다.(직역하면 '미리보기에 대한 마스크')

11. Decoder의 첫번째 서브층 : Self-Attention과 look-ahead mask

Look-ahead mask는 오른쪽 그림에서 보이듯 Decoder의 1번째 서브층에서 이루어짐

- Decoder의 첫번째 서브층인 **Multi-head Self-Attention** 층은
  Encoder의 첫번째 서브층인 Multi-head self-attention 층과 동일한 연산을 수행하며,
  Attention Score 행렬에서 마스킹을 적용한다는 점만 다르다.

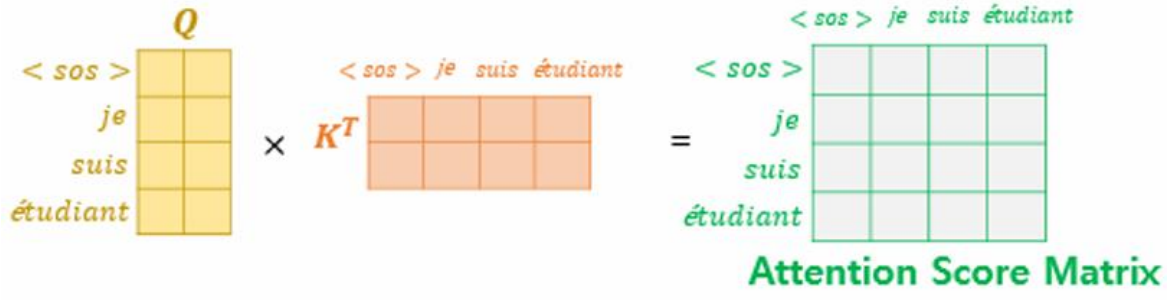


Look-ahead mask가 없다면?:

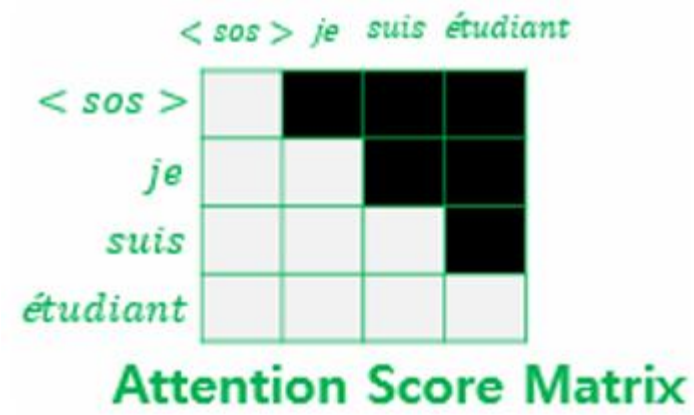
- **학습(Training) 단계:** 모델을 학습시킬 때, 디코더는 정답 문장 전체를 입력으로 받습니다. 예를 들어 "나는 학교에 간다"라는 문장을 학습한다면, "나는" 다음에는 "학교에"가 와야 하고, "나는 학교에" 다음에는 "간다"가 와야 한다는 것을 배워야 합니다.
- **문제점:** 만약 look-ahead mask가 없다면, 모델이 "나는" 다음에 올 단어를 예측해야 하는 상황에서 이미 뒤에 있는 "학교에"나 "간다"라는 정답을 볼 수 있게 됩니다. 이렇게 되면 모델은 단순히 다음 위치의 단어를 그대로 베끼기만 할 뿐, 문맥을 이해하고 다음 단어를 **예측하는 능력**을 제대로 학습할 수 없습니다. 🧐

11. Decoder의 첫번째 서브층 : Self-Attention과 look-ahead mask

우선, 아래와 같이 self-attention을 통해서 Attention-score 행렬을 얻는다.



이후, 자기자신보다 미래에 있는 단어들은 참고하지 못하도록 아래와 같이 masking한다.



Masking된 후의 Attention Score 행렬의 각 행을 보면 자기 자신과 그 이전단어들만 참고할 수 있는 것을 확인할 수 있다.

11. Decoder의 첫번째 서브층 : Self-Attention과 look-ahead mask

이 외에는 self-attention이라는 점과, multi-head attention을 수행한다는 점에서 Encoder의 첫번째 서브층과 같다.

Transformer에는 총 3가지 Attention이 존재한다.

- 모두 multi-head attention을 수행하고,
- multi-head attention function 내부에서 score 함수로 scaled dot product attention 함수를 호출하는데,

  각 attention시 함수에 전달하는 masking은 아래와 같다.

1. Encoder의 self-attention: **패딩 마스크**를 전달
2. Decoder의 1번째 서브층인 Masked-self attention : **look-ahead mask**를 전달
   - 이때, look-ahead mask를 한다고 해서 패딩 마스크가 불필요한 것이 아니므로, **look-ahead mask는 패딩 마스크를 포함하도록 구현한다.**
3. Decoder의 2번째 서브층인 Encoder-Decoder attention: **패딩 마스크**를 전달

12. Decoder의 두번째 서브층 : Encoder-Decoder Attention

Decoder의 2번째 서브층은 Multi-head Attention을 수행한다는 점에서는 이전의 Attention들(Encoder와 Decoder의 첫번째 서브층)과는 공통점이 있지만, 이번에는 **self-attention이 아니다.**

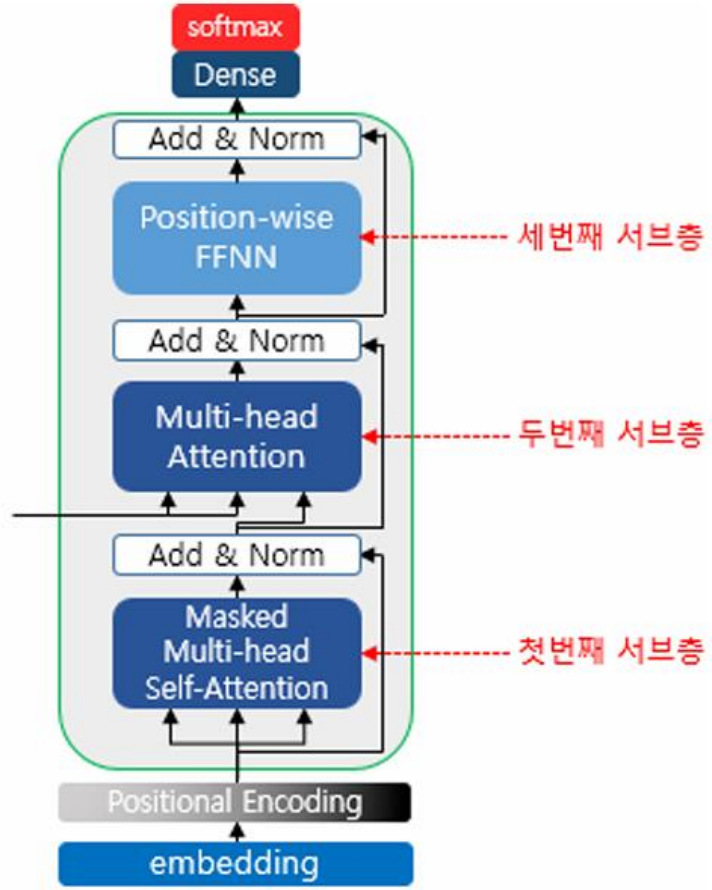
- Encoder-Decoder attention은 **Query가 Decoder 행렬**인 반면, **Key와 Value는 Encoder 행렬**이기 때문이다.

각 서브층에서의 Q, K, V 관계를 정리해보면 아래와 같다.

Encoder의 첫번째 서브층: Query = Key = Value

Decoder의 첫번째 서브층: Query = Key = Value

Decoder의 두번째 서브층: Query: Decoder 행렬 / Key = Value: Encoder 행렬



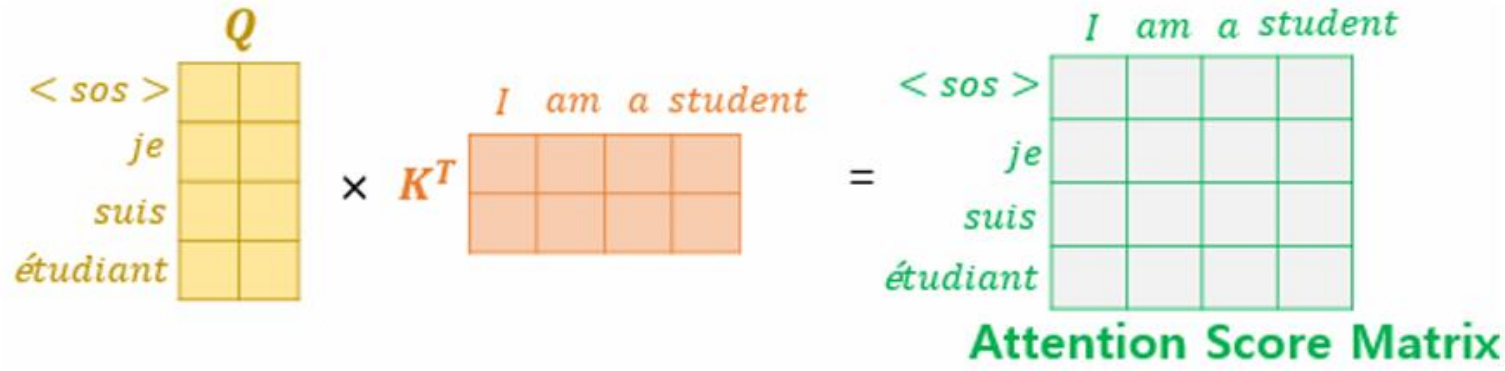
12. Decoder의 두번째 서브층 : Encoder-Decoder Attention

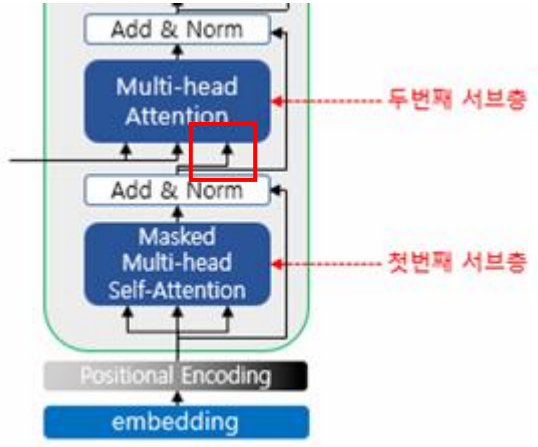
Decoder의 2번째 서브층을 확대해보면 우측과 같이 Encoder로부터 2개의 화살표가 그려져 있다.

- 2개의 화살표는 각각 Key와 Value를 의미하며, 이는 **Encoder의 마지막층에서 온 행렬**로부터 얻는다.

반면 **Query**는 우측 하단의 그림과 같이 **Decoder의 첫번째 서브층의 결과 행렬로부터 얻는다.**

Query가 Decoder행렬, Key가 Encoder행렬일 때, Attention Score 행렬을 구하는 과정은 아래와 같다.





이 외에 Multi-head attention을 수행하는 과정은 다른 Attention들과 같다.

Ref

모든 Summary 자료는 직접 작성하였으며,
사용된 설명 및 사진의 원 출처는 딥러닝 파이토치 교과서의 공개 내용임을 밝힙니다.