

**Natural Language Processing**

# 언어 모델: Transformer 아키텍처

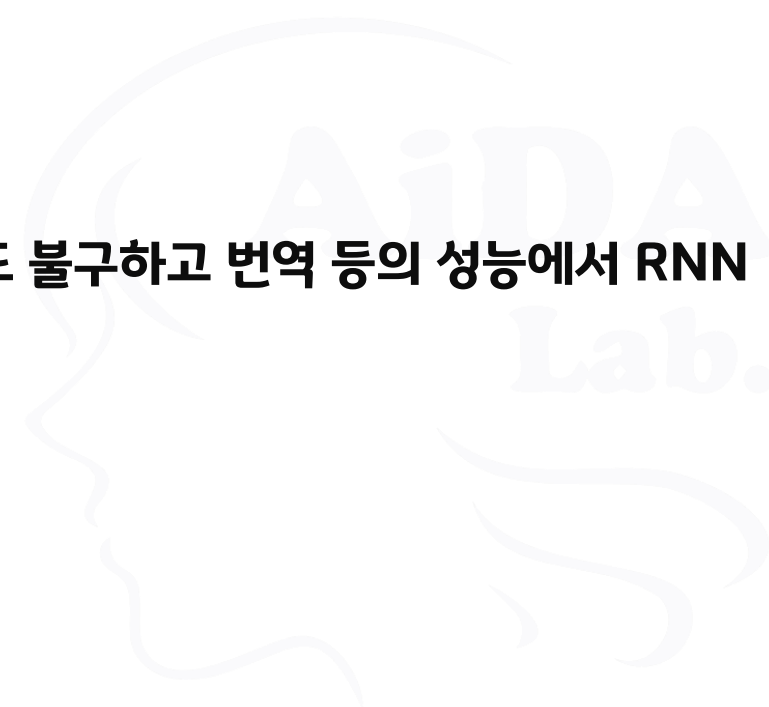
강사 양석환



# Transformer 아키텍처

- **트랜스포머(Transformer)란?**

- 2017년, 구글이 제안한 Sequence-to-Sequence 모델의 하나
- 구글이 발표한 “Attention is all you need”라는 논문에서 제안된 모델이며
- 기존의 seq2seq 구조인 Encoder-Decoder 방식을 따르면서도
- 논문 제목처럼 Attention 만으로 구현한 모델임
- RNN 모델을 사용하지 않고 Encoder-Decoder를 설계하였음에도 불구하고 번역 등의 성능에서 RNN 모델보다 우수한 성능을 보임



- 기존의 seq2seq 모델의 한계

- 기존의 seq2seq 모델은 인코더-디코더 구조로 구성
- 인코더는 입력 시퀀스를 하나의 벡터 표현으로 압축하고, 디코더는 이 벡터 표현을 통해서 출력 시퀀스를 만들어 냄

→ 이러한 구조는

- 인코더가 입력 시퀀스를 하나의 벡터로 압축하는 과정에서
- 입력 시퀀스의 정보가 일부 손실된다는 단점이 발생하였고
- 이를 보정하기 위해 어텐션(Attention) 모델이 사용됨



- 그런데...

- 어텐션을 RNN의 보정을 위한 용도로서 사용하는 것이 아니라
- 어텐션만으로 인코더와 디코더를 만들어보면 어떨까?

## → 트랜스포머의 등장

- RNN에서 사용한 순환방식을 사용하지 않고 순수하게 어텐션만 사용한 모델
- 기존에 사용되었던 RNN, LSTM, GRU 등은 점차 트랜스포머로 대체되기 시작
- GPT, BERT, T5 등과 같은 다양한 자연어 처리 모델에 트랜스포머 아키텍처가 적용됨

- 기계 번역에서의 seq2seq 작업 예시

소스 언어에서의 토큰 시퀀스

어제, 카페, 갔었어, 거기, 사람, 많더라



타겟 언어에서의 토큰 시퀀스

I, went, to, the, cafe, there, were, many, people, there

- 소스 시퀀스의 길이(토큰 6개)와 타겟 시퀀스의 길이(10개)가 다르다.
- 그러나 기존 seq2seq에서는 인코딩 길이는 고정 → Attention으로 극복
  - 제대로 된 seq2seq는 소스, 타겟의 길이가 달라도 과제 수행에 문제가 없어야 함

## • 트랜스포머의 인코더-디코더 구조

### • 인코더

- 소스 시퀀스의 정보를 압축해 디코더로 보냄

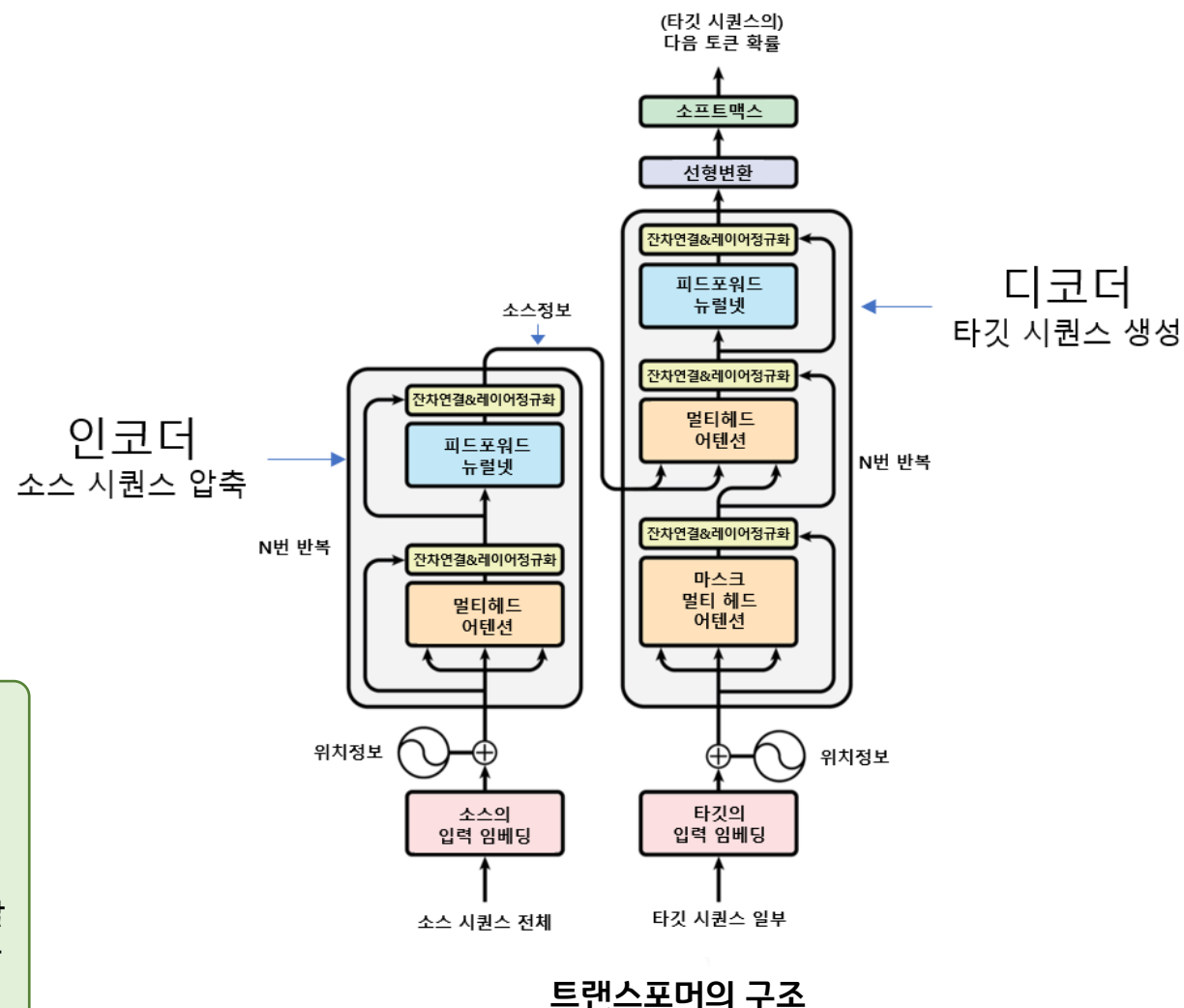
### • 디코더

- 인코더가 보내 준 소스 시퀀스의 정보를 받아서 타겟 시퀀스 생성

트랜스포머는 Sequence to Sequence 형태의 과제 수행에 특화된 모델.

임의의 시퀀스를 해당 시퀀스와 속성이 다른 시퀀스로 변환하는 작업이라면 꼭 기계 번역이 아니더라도 수행할 수 있음

예: 필리핀 앞바다의 한 달 치 기온 데이터 → 앞으로 1주일간 하루 단위로 태풍이 발생할지를 맞히는 과제(기온의 시퀀스 → 태풍발생 여부의 시퀀스)도 트랜스포머가 할 수 있는 일임



## • 트랜스포머의 학습 방법

### 1. 'I'를 예측하는 학습

- 인코더 입력: 어제, 카페, 갔었어, 거기, 사람, 많더라 → (소스 시퀀스 전체)

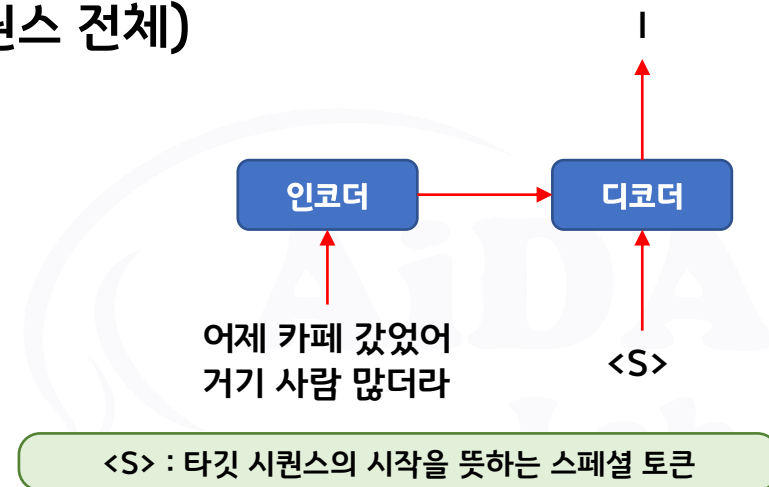
- 소스 시퀀스를 압축하여 디코더로 보내고

- 디코더 입력: <S>

- 인코더에서 보내온 정보와 현재 디코더 입력을 모두 고려하여 토큰(I)를 예측

- 디코더의 최종 출력:

- 타겟 언어의 어휘 수만큼의 차원으로 구성된 벡터 → 이 벡터는 요소의 값이 모두 확률 값  
(예: 타겟 언어의 어휘가 총 3만개라면 디코더 출력은 3만 차원의 벡터, 3만개 각각은 확률값)



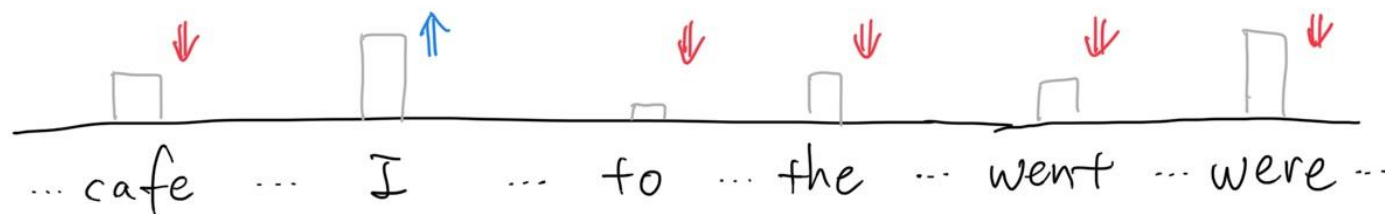


## 2. 트랜스포머의 학습 진행

- 인코더와 디코더의 입력이 주어졌을 때, 정답에 해당하는 단어의 확률을 높이는 방식으로 학습함

→ 그림에서...

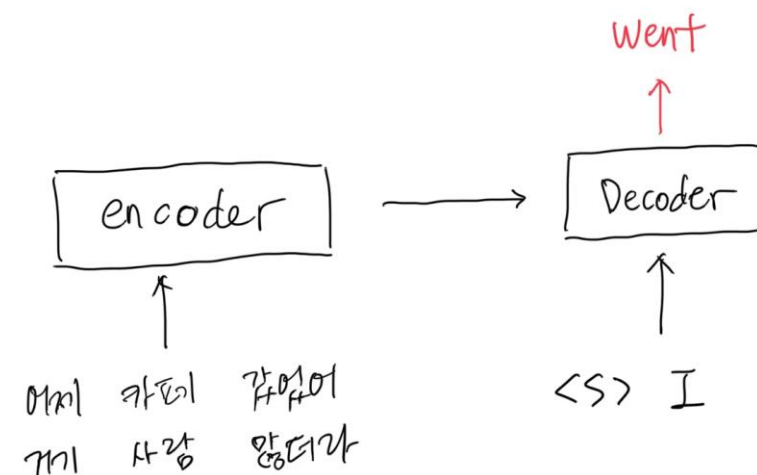
- 모델은 이번 시점의 정답인 I에 해당하는 확률은 높이고, 나머지 단어의 확률은 낮아지도록 모델 전체를 갱신



i 확률 높이기

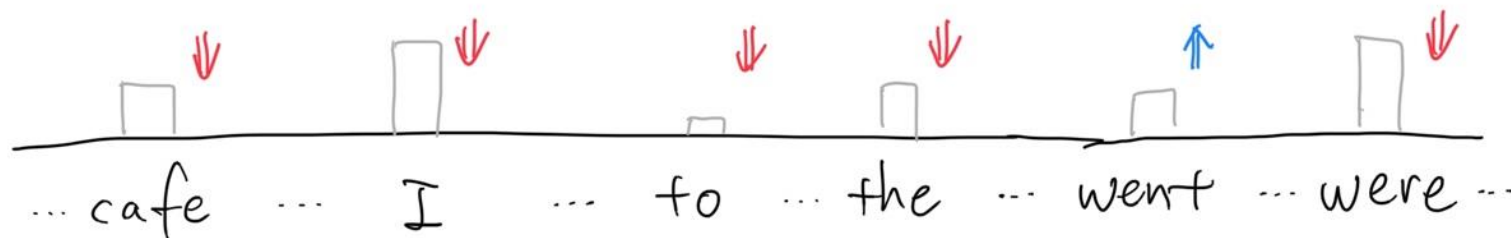
## 3. 'went'를 예측할 차례

- 인코더 입력: 어제, 카페, 갔었어, 거기, 사람, 많더라 → (소스 시퀀스 전체)
- 디코더 입력: <S> I
- 특이 사항
  - 학습 중의 디코더 입력과 학습을 마친 후 모델을 실제 기계번역에 사용할 때(인퍼런스)의 디코더 입력이 다름
    - 학습 중: 디코더 입력에 예측해야 할 단어(went) 이전의 정답 타깃 시퀀스(<s> I)를 넣어줌
    - 학습 후(인퍼런스 때): 현재 디코더 입력에 직전 디코딩 결과를 사용  
(예: 인퍼런스 때 직전 디코더 출력이 I 대신 you가 나오면 다음 디코더 입력은 <s> you)



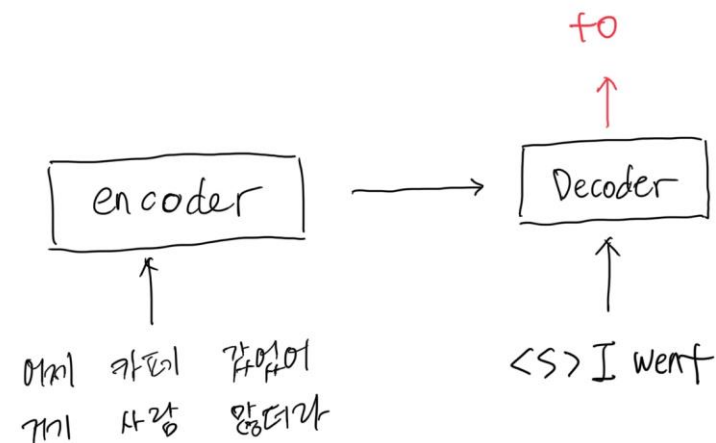
## 4. 'went' 확률 높이기

- 학습 과정 중 인코더, 디코더의 입력이 아래 그림과 같으면...
- 모델은 이번 시점의 정답인 went에 해당하는 확률은 높이고
- 나머지 단어의 확률은 낮아지도록 모델 전체를 갱신함

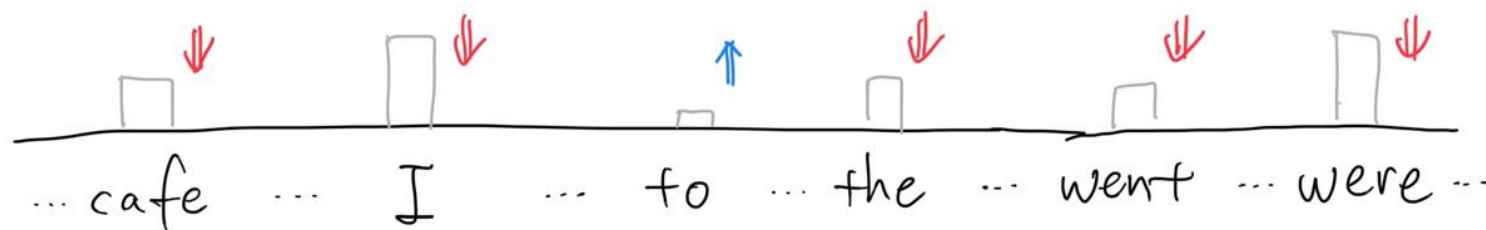


## 5. 'to' 예측하기

- 인코더 입력은 소스 시퀀스 전체
- 디코더 입력은 정답인 <s> I went
- 인퍼런스 할 때, 디코더 입력은 직전 디코딩 결과
- 이번 시점의 정답인 to에 해당하는 확률은 높이고
- 나머지 단어의 확률은 낮아지도록 모델 전체를 갱신함

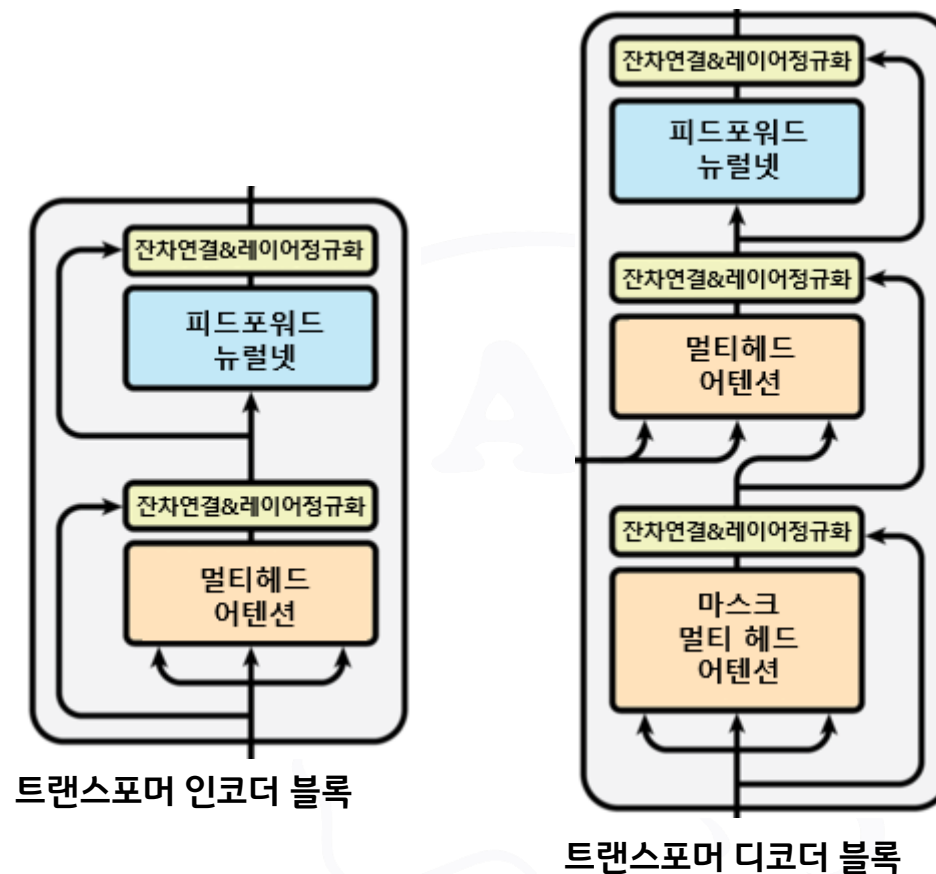


이상의 방식으로 말뭉치 전체를 반복해서 학습



## • 트랜스포머 블록

- 트랜스포머는 블록 형태로 구성된 인코더, 디코더 수십개가 반복적으로 쌓여 있는 형태
- 이런 구조를 블록 또는 레이어라고 부름
- 인코더 블록: 3가지 요소로 구성
  - 멀티 헤드 어텐션, 피드포워드 뉴럴 네트워크, 잔차 연결 & 레이어 정규화
- 디코더 블록: 인코더 블록이 변형된 형태
  - 마스크 멀티 헤드 어텐션 추가됨



## • 셀프 어텐션

트랜스포머 구조에서 “멀티 헤드 어텐션”을 “셀프 어텐션“ 이라고 부름

### • 어텐션

- 시퀀스 입력에 수행하는 기계학습 방법의 일종
- 시퀀스 요소 가운데 중요한 요소에 집중하고, 그렇지 않은 요소는 무시함으로써 태스크 수행 성능을 끌어올림
- 디코딩 시 소스언어의 단어 시퀀스 중에서 디코딩에 도움이 되는 단어 위주로 취사 선택(가장 중요한 것만 추려서), 번역 품질을 끌어올림

### • 셀프 어텐션은 자신에게 수행하는 어텐션 기법

### • 트랜스포머 경쟁력의 원천으로 평가

- 셀프 어텐션과 합성곱 신경망(CNN)을 비교하면...

- CNN: 합성곱 필터를 이용해 시퀀스의 지역적 특징을 잡아낼 수 있다

- 자연어는 기본적으로 시퀀스

- 특정 단어를 기준으로 한 주변 문맥이 의미 형성에 중요한 역할 수행


→ 자연어 처리에 CNN도 사용되기 시작

- 합성곱 필터 크기를 넘어서는 문맥은

읽어 내기 어렵다

- 필터 크기가 3이면 4칸 이상 떨어져 있는

단어 사이의 의미는 잡아내기 어려움



어제	카페	갔었어	거기	사람	많더라
어제	카페	갔었어	거기	사람	많더라
어제	카페	갔었어	거기	사람	많더라
어제	카페	갔었어	거기	사람	많더라

합성곱 신경망을 이용한 인코딩

- **셀프 어텐션과 순환 신경망(RNN)을 비교하면...**

- **RNN: 시퀀스 정보 압축에 강점이 있는 구조**

- 소스 시퀀스를 차례대로 처리
- 그러나 RNN은 시퀀스 길이가 길어질 수록 정보 압축에 문제 발생
  - 오래 전에 입력된 단어를 잊어버리거나
  - 특정 단어 정보를 과도하게 반영해 전체 정보를 왜곡시키는 문제

- “어제, 카페, 갔었어, 거기, 사람, 많더라”를 RNN으로 기계번역 한다면

→ 인코더가 디코더로 넘기는 정보는 소스 시퀀스의 마지막 단어인 “많더라”의 의미가 많이 반영됨(입력 정보를 차례로 처리하고, 오래된 단어는 잊어버리는 경향이 있기때문)



- 셀프 어텐션과 어텐션을 비교하면...



- 그림을 보면

- cafe에 대응하는 소스 언어의 단어: 카페 → 소스 시퀀스의 초반부에 등장
- cafe라는 단어를 디코딩 할 때, 카페를 반드시 참조
- 어텐션이 없는 단순 RNN이라면... 워낙 초반에 입력된 단어라 잊었을 가능성이 크고, 이 때문에 번역 품질이 낮아질 수 있다.

- **셀프 어텐션과 어텐션의 주요 차이**

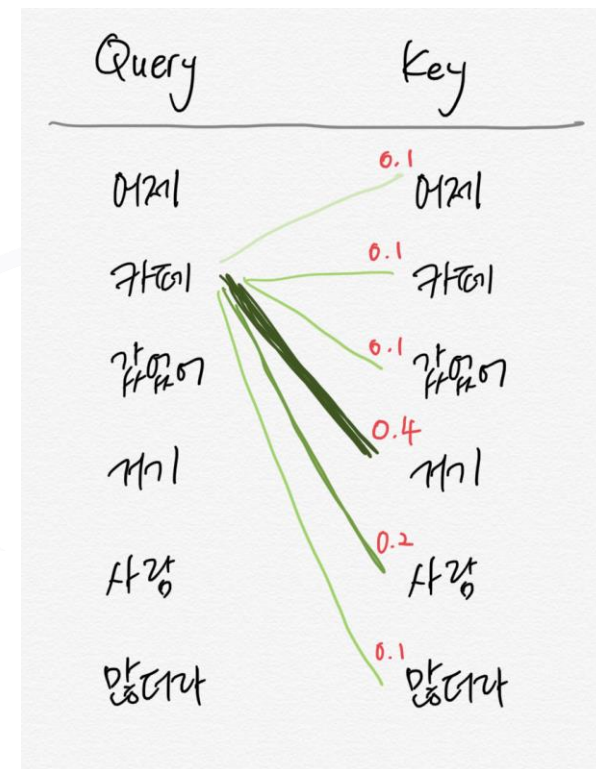
- 어텐션은 소스 시퀀스 전체 단어들과 타겟 시퀀스 단어 하나 사이를 연결하는데 사용.  
셀프 어텐션은 입력 시퀀스 전체 단어들 사이를 연결



- 어텐션은 RNN 구조 위에서 동작하지만 셀프 어텐션은 RNN 없이 동작함
- 타겟 언어의 단어를 1개 생성할 때 어텐션은 1회 수행하지만, 셀프 어텐션은 인코더, 디코더 블록의 개수만큼 반복 수행

## • 셀프 어텐션 계산 예시

- 쿼리(Q), 키(K), 값(V)이 서로 영향을 주고 받으면서 문장의 의미 계산
- 각 단어 벡터는 블록 내에서 계산과정을 통해 Q, K, V로 변환
- 쿼리 단어 각각을 대상으로 모든 키 단어와 얼마나 유의적인 관계를 맺는지를 총합이 1인 확률 값으로 표현
- “카페”와 가장 관련이 높은 단어는 “거기(0.4)”



## • 하이퍼 파라미터

- 머신러닝에서 모델링 시에 사용자가 직접 설정해 주는 값
- 학습률(Learning Rate) 등 다양한 종류가 있음
- 하이퍼 파라미터의 정의 및 특성(By Machine Learning Mastery)
  - 모델 하이퍼파라미터는 모델 외부에 있고 데이터에서 값을 추정할 수 없는 구성이다.
    - 모델 매개 변수를 추정하는 데 도움이 되는 프로세스에서 자주 사용된다.
    - 그들은 종종 관련 전문가에 의해 지정된다.
    - 휴리스틱을 사용하여 설정할 수 있다.
    - 그들은 종종 주어진 예측 모델링 문제에 맞게 조정된다.

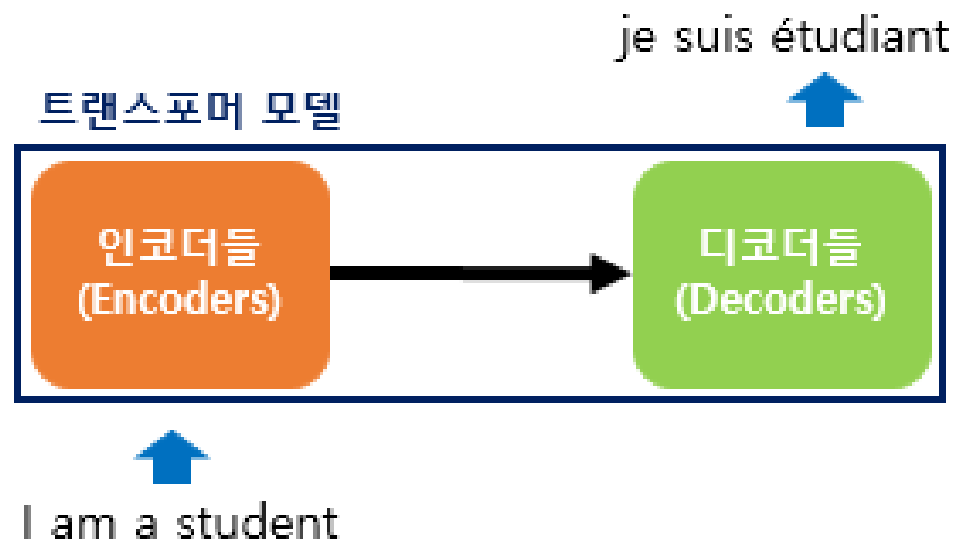
하이퍼 파라미터와 파라미터를 혼용하고 있는데 둘은 엄연히 다른 개념이다.  
파라미터는 모델 내부에서 결정되는 매개변수이며 데이터로부터 결정된다.  
하이퍼 파라미터는 모델 외부에서 사용자가 직접 지정하는 값이다.

## • 트랜스포머의 주요 하이퍼 파라미터

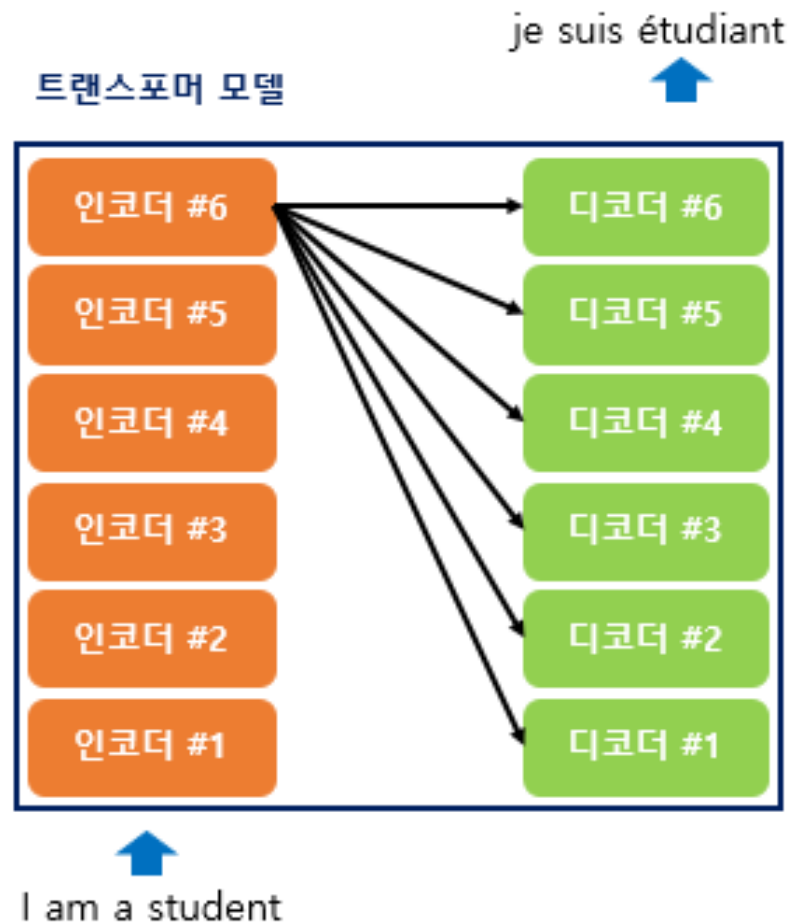
- $d_{model} = 512$ 
  - 트랜스포머의 인코더와 디코더에서의 정해진 입력과 출력의 크기를 의미함
- $num\_layers = 6$ 
  - 트랜스포머 모델에서 인코더와 디코더가 총 몇 층으로 구성되었는지를 의미함
- $num\_heads = 8$ 
  - 트랜스포머에서는 어텐션을 사용할 때, 한 번 하는 것 보다 여러 개로 분할해서 병렬로 어텐션을 수행하고 결과값을 다시 하나로 합치는 방식을 택했으며, 이때 이 병렬의 개수를 의미함
- $d_{ff} = 2048$ 
  - 트랜스포머 내부에 존재하는 피드 포워드 신경망의 은닉층의 크기를 의미함

## • 인코더-디코더 구조

- seq2seq: 인코더와 디코더에서 각각 하나의 RNN이  $t$ 개의 시점(time step)을 가지는 구조
- 트랜스포머: 인코더와 디코더라는 단위가  $N$ 개로 구성되는 구조

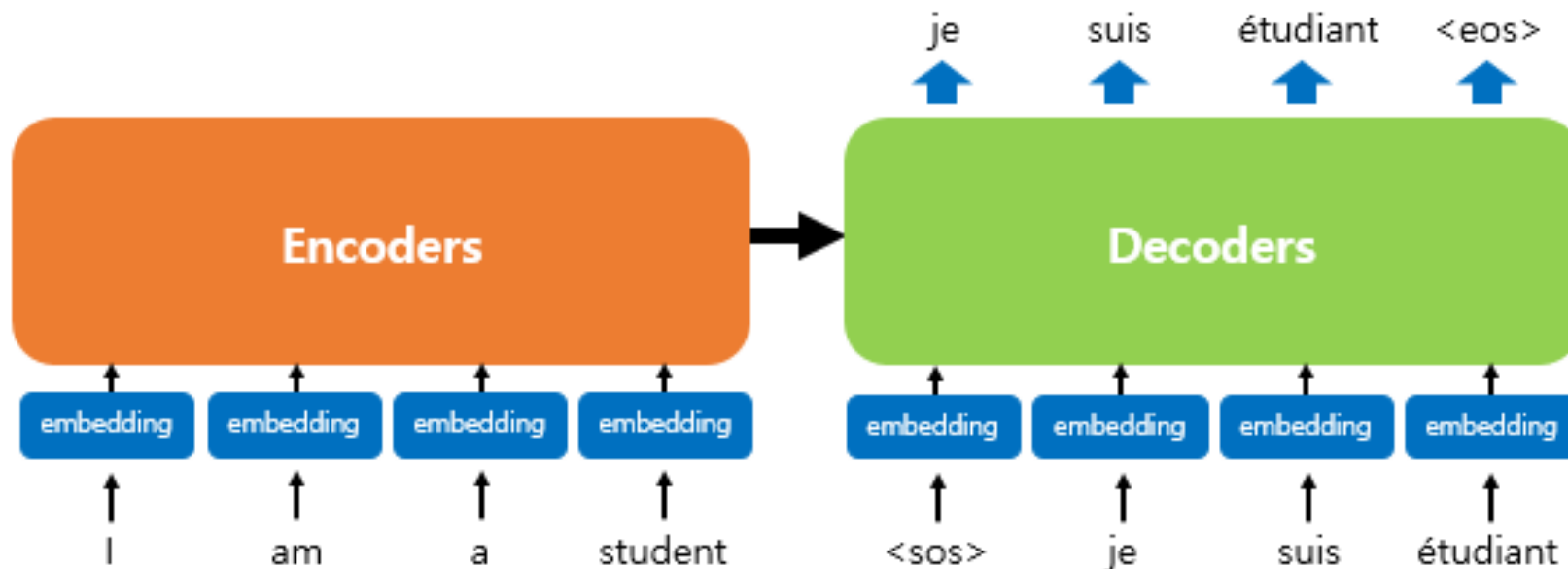


- 트랜스포머를 제안한 논문에서는 인코더와 디코더의 개수를 각각 6개 사용



- 트랜스포머의 동작 형태

- 인코더로부터 정보를 전달받아 디코더가 출력 결과를 만들어내는 구조
- 디코더는 seq2seq 구조처럼 시작 심볼 <sos>를 입력으로 받아 종료 심볼 <eos>가 나올 때까지 연산을 진행 → RNN은 사용되지 않지만 인코더-디코더의 구조는 유지되고 있





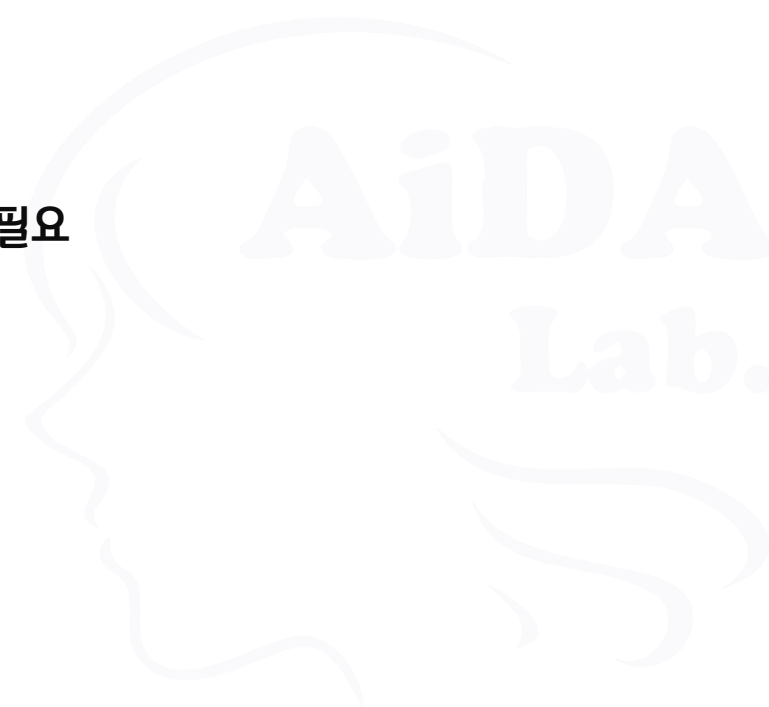
- **트랜스포머의 입력**

- **RNN이 자연어 처리에서 유용했던 이유**

- 단어의 위치에 따라 단어를 순차적으로 입력받아서 처리하는 RNN의 특성으로 인해 각 단어의 위치 정보 (position information)를 가질 수 있었기 때문

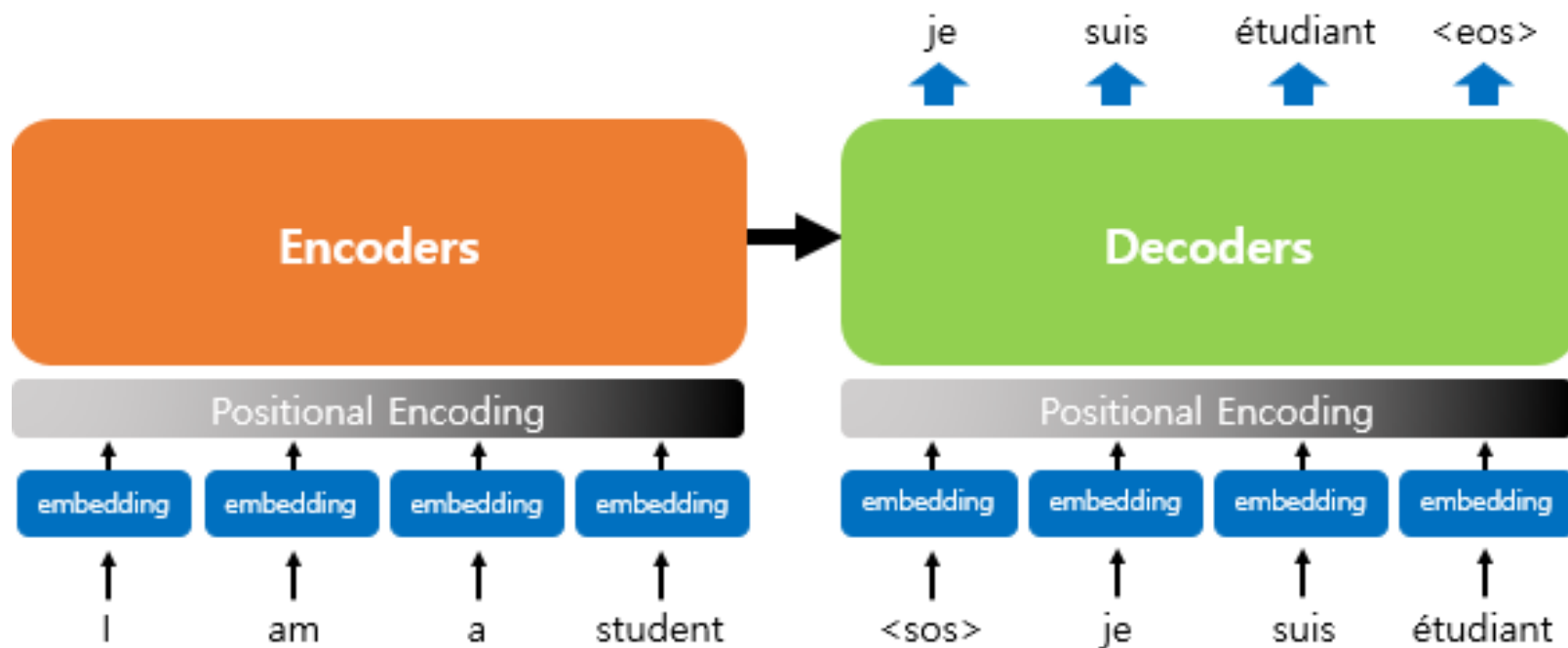
- **트랜스포머의 경우**

- 단어 입력을 순차적으로 받지 않음 → 단어의 위치 정보를 알려줄 방법 필요
    - 포지셔널 인코딩(Positional Encoding) 적용
      - 단어의 위치 정보를 얻기 위해서
      - 각 단어의 임베딩 벡터에 위치 정보들을 더하여 모델의 입력으로 사용

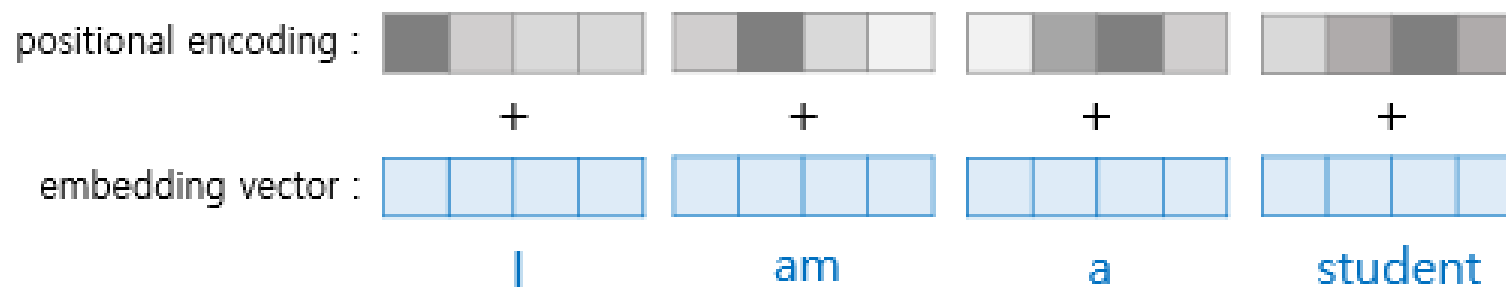


- 포지셔널 인코딩

- 입력되는 임베딩 벡터들은 트랜스포머의 입력으로 사용되기 전에 포지셔널 인코딩의 값이 추가됨



- 임베딩 벡터에 포지셔널 인코딩값이 더해지는 과정

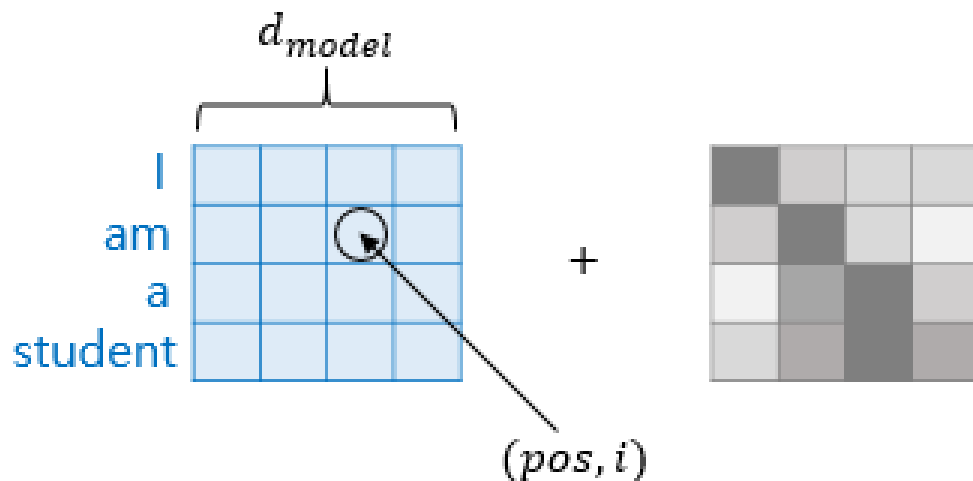


- 포지셔널 인코딩 값의 계산
  - 트랜스포머는 위치 정보를 가진 값을 만들기 위해서 아래의 두 개의 함수를 사용

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

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

사인, 코사인 함수의 그래프에서 값의 형태를 생각해볼 수 있음.  
트랜스포머는 사인 함수와 코사인 함수의 값을  
임베딩 벡터에 더해줌으로써 단어의 순서 정보를 추가함



$pos$  : 입력 문장에서의 임베딩 벡터의 위치

$i$  : 임베딩 벡터 내의 차원의 인덱스

$d_{model}$  : 트랜스포머의 모든 층의 출력 차원을 의미하는 트랜스포머의 하이퍼 파라미터

- 짝수( $pos, 2i$ ) 인 경우에는 사인 함수의 값을 사용
  - 홀수( $pos, 2i+1$ ) 인 경우에는 코사인 함수의 값을 사용
- 임베딩 벡터도  $d_{model}$ 의 차원을 가짐
  - 위의 그림은 4차원의 예시를 들고 있으나 논문에서는 512차원을 사용함

- 포지셔널 인코딩 방법을 사용하면 순서 정보가 보존됨
  - 각 임베딩 벡터에 포지셔널 인코딩의 값을 더하면 같은 단어라고 하더라도 문장 내의 위치에 따라서 트랜스포머의 입력으로 들어가는 임베딩 벡터의 값이 달라짐
  - 이에 따라 트랜스포머의 입력은 순서 정보가 고려된 임베딩 벡터가 됨



## • 트랜스포머에서 사용되는 어텐션

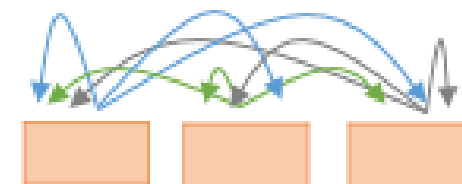
- Encoder Self-Attention
- Masked Decoder Self-Attention
- Encoder-Decoder Attention

- 셀프 어텐션은 본질적으로 Query, Key, Value가 동일한 경우를 말함
- 여기서 Query, Key 등이 같다는 것은 벡터의 값이 같다는 것이 아니라 벡터의 출처가 같다는 것을 의미함
- Encoder-Decoder Attention의 경우 Query가 디코더의 벡터인 반면에 Key와 Value가 인코더의 벡터이므로 셀프 어텐션이라고 부르지 않음

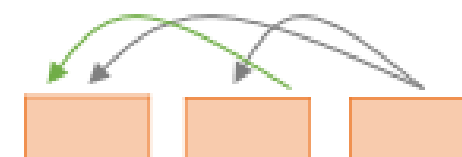
인코더의 셀프 어텐션 : Query = Key = Value

디코더의 마스크드 셀프 어텐션 : Query = Key = Value

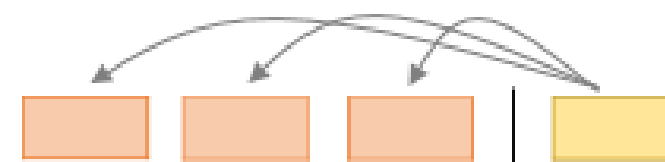
디코더의 인코더-디코더 어텐션 : Query : 디코더 벡터 / Key = Value : 인코더 벡터



Encoder Self-Attention

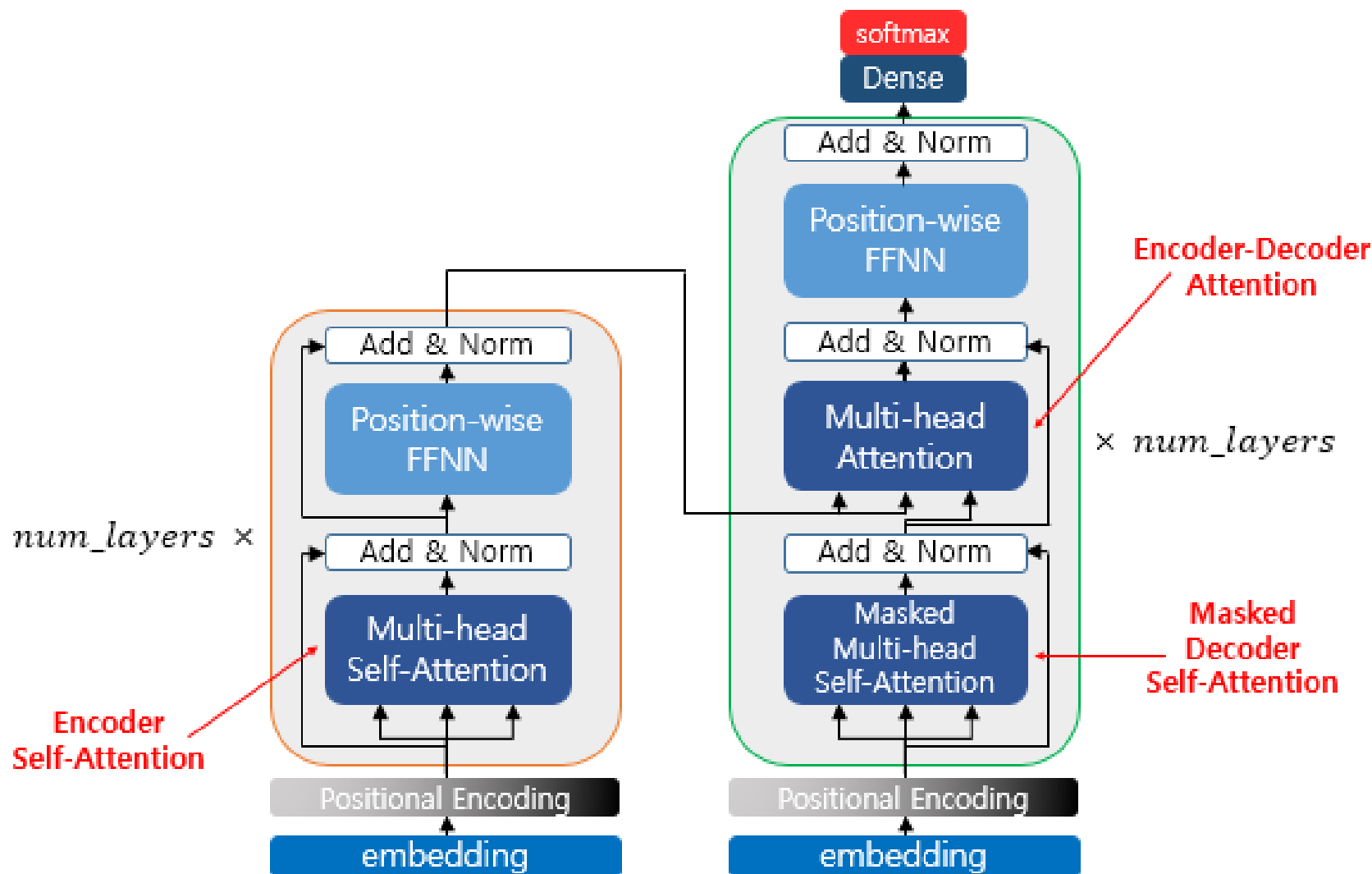


Masked Decoder Self-Attention



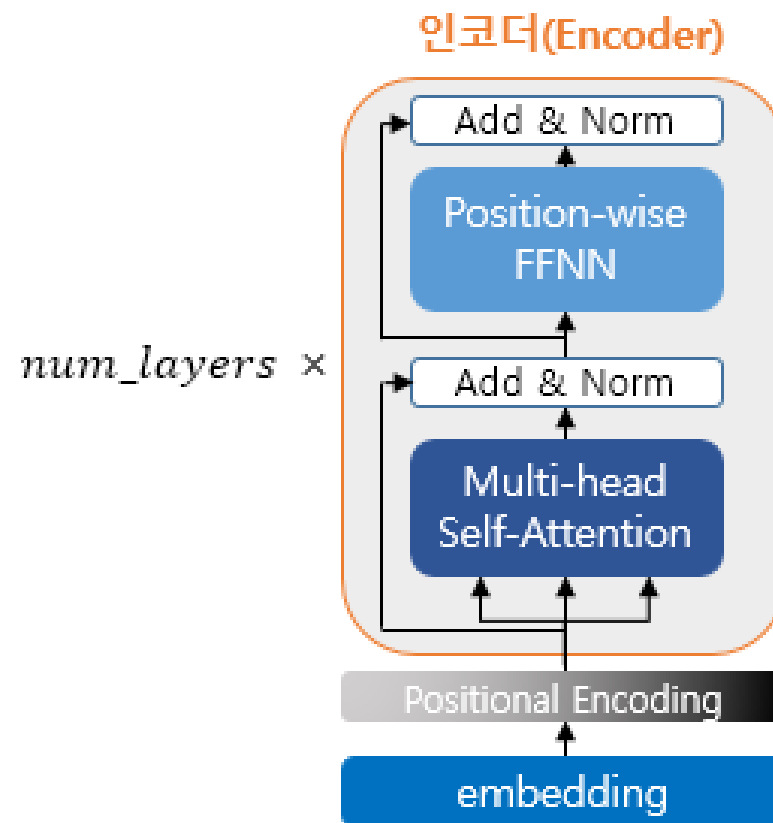
Encoder-Decoder Attention

Encoder Self-Attention은 인코더에서,  
나머지는 디코더에서 실행됨



## • 인코더의 구조

- 트랜스포머는 하이퍼 파라미터인 `num_layers` 개의 인코더 층이 쌓여서 구성됨
- 인코더를 하나의 층으로 생각한다면
  - 하나의 인코더 층은 크게 총 2개의 서브층으로 구분
    - 셀프 어텐션 (= 멀티 헤드 셀프 어텐션)
    - 피드 포워드 신경망 (= 포지션 와이즈 피드 포워드 신경망)

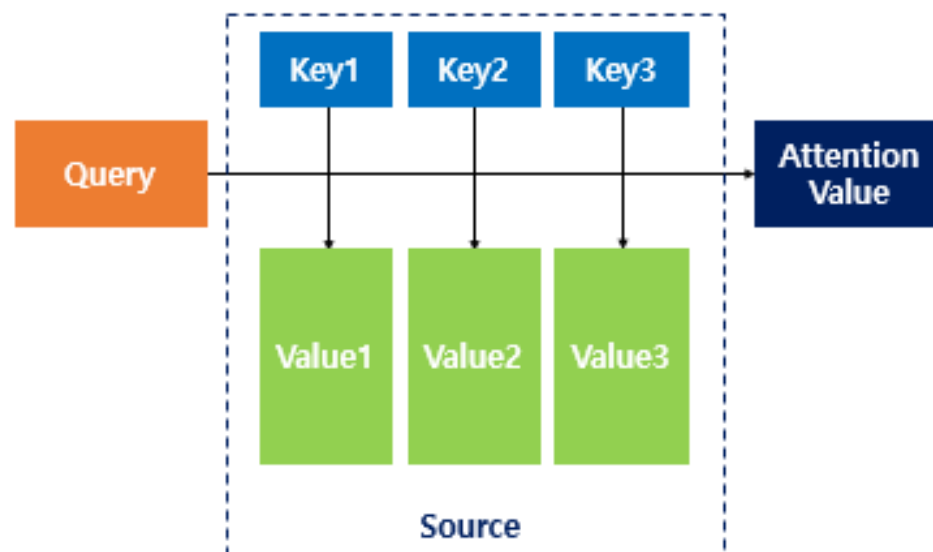




- 셀프 어텐션

- 기존 어텐션의 개념

- 어텐션 함수는 주어진 '쿼리(Query)'에 대해서 모든 '키(Key)'와의 유사도를 각각 계산
    - 계산된 유사도를 가중치로 하여 키와 매핑되어있는 각각의 '값(Value)'에 반영
    - 유사도가 반영된 '값(Value)'을 모두 가중합하여 리턴



- 셀프 어텐션(self-attention)
  - 어텐션을 자기 자신에게 수행한다는 의미
  - 어텐션을 사용할 경우의 Q, K, V의 정의

Q = Query : t 시점의 디코더 셀에서의 은닉 상태

K = Keys : 모든 시점의 인코더 셀의 은닉 상태들

- V = Values : 모든 시점의 인코더 셀의 은닉 상태들

→ 전체 시점에 대해서 일반화한다면

Q = Querys : 모든 시점의 디코더 셀에서의 은닉 상태들

K = Keys : 모든 시점의 인코더 셀의 은닉 상태들

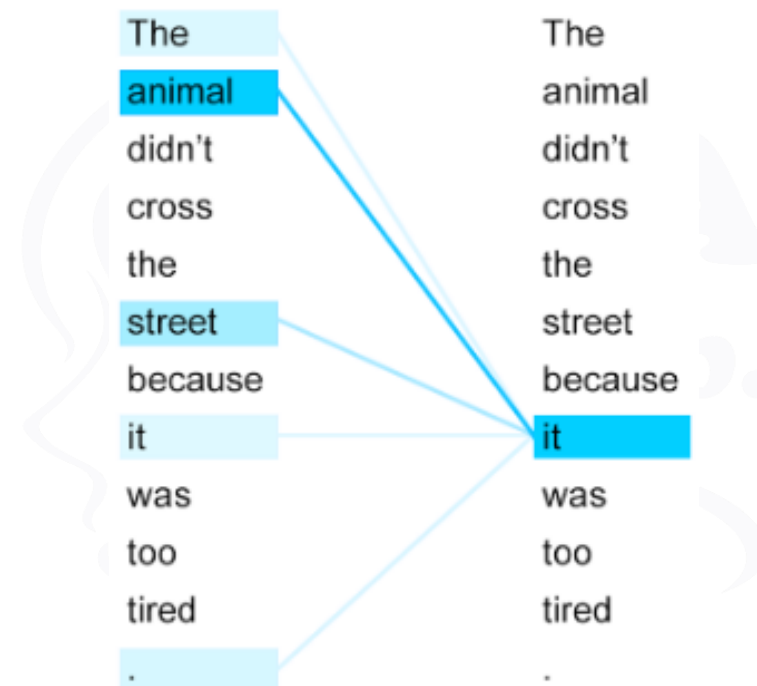
V = Values : 모든 시점의 인코더 셀의 은닉 상태들



- 기존 어텐션: 디코더 셀의 은닉 상태가 Q, 인코더 셀의 은닉 상태가 K → Q와 K가 서로 다른 값
- 셀프 어텐션: Q, K, V가 전부 동일

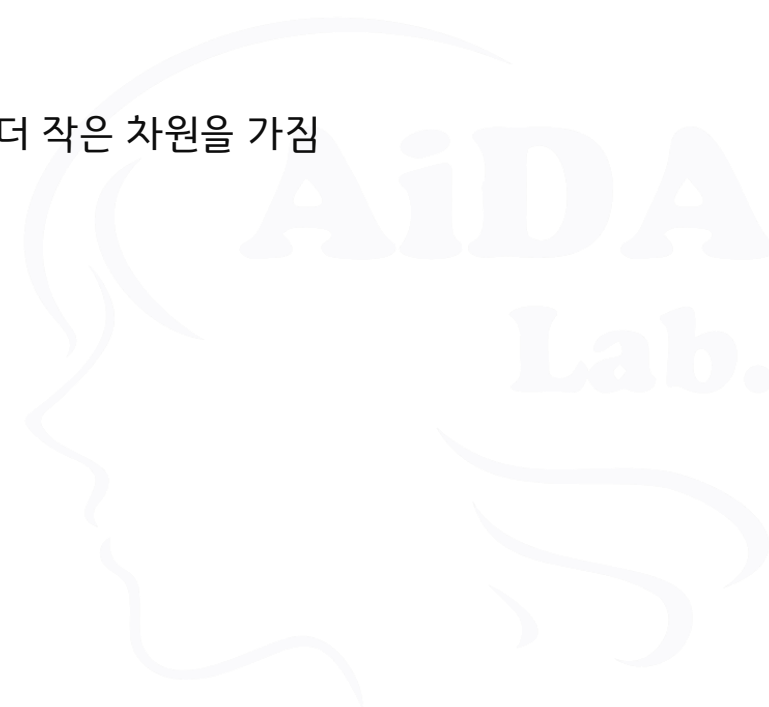
Q : 입력 문장의 모든 단어 벡터들  
 K : 입력 문장의 모든 단어 벡터들  
 V : 입력 문장의 모든 단어 벡터들

- 셀프 어텐션을 통해 얻을 수 있는 대표적인 효과
  - '그 동물은 길을 건너지 않았다. 왜냐하면 그것은 너무 피곤하였기 때문이다.'
  - 그것(it)에 해당하는 것은 과연 길(street)일까? 동물(animal)일까? → 사람과 달리 기계에게는 어려움
  - 셀프 어텐션은 입력 문장 내의 단어들끼리 유사도를 구함으로써 그것(it)이 동물(animal)과 연관되었을 확률이 높다는 것을 찾아냄

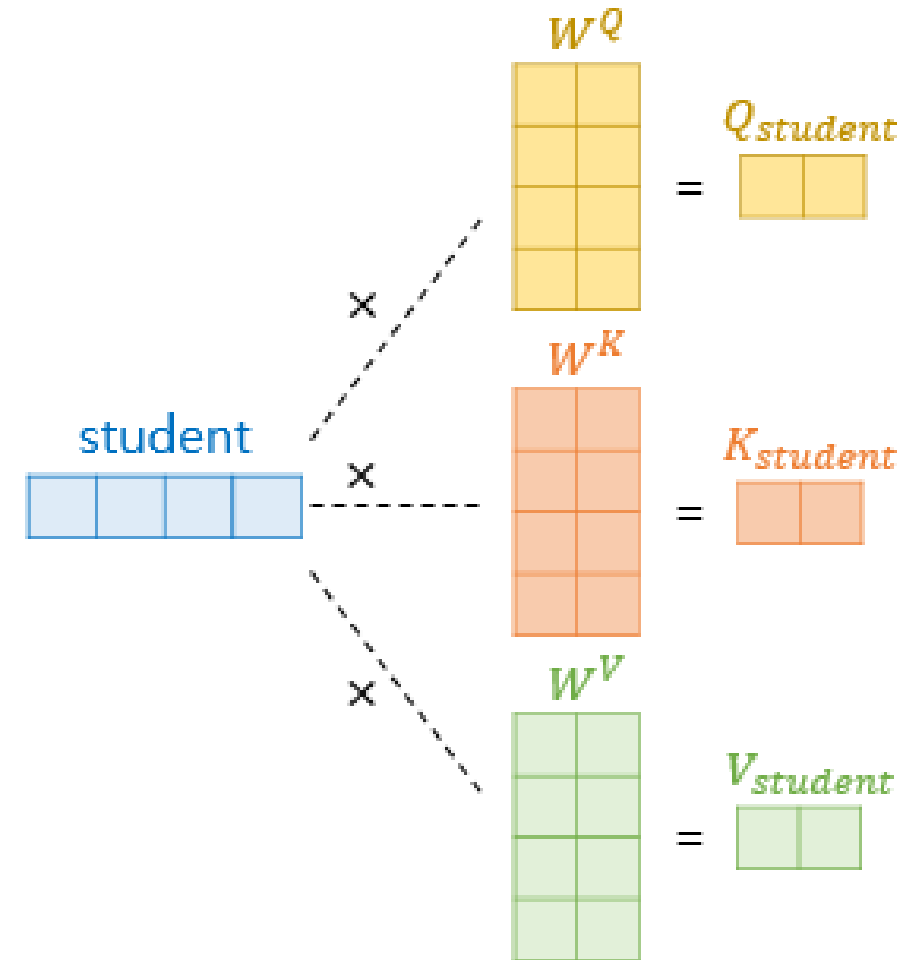


트랜스포머에 대한 구글 AI 블로그 포스트에서

- Q, K, V 벡터 얻기
  - 셀프 어텐션은 입력 문장의 단어 벡터들을 가지고 수행한다
    - 정확히는... 인코더의 초기 입력인  $d_{model}$ 의 차원을 가지는 단어 벡터들을 사용하여 셀프 어텐션을 수행하는 것이 아니라...
  - 각 단어 벡터들로부터 Q벡터, K벡터, V벡터를 얻는 작업 수행
    - Q, K, V들은 초기 입력인  $d_{model}$ 의 차원을 가지는 단어 벡터들보다 더 작은 차원을 가짐
    - 작은 차원의 크기는 하이퍼파라미터인 num\_heads로 인해 결정
      - Q, K, V 차원 =  $\frac{d_{model}}{num\_heads}$



- 예문 중 student라는 단어 벡터를 Q, K, V의 벡터로 변환하는 과정
  - 기존의 벡터로부터 더 작은 벡터는  
가중치 행렬을 곱해서 만들
  - 각 가중치 행렬의 크기는  $d_{model} \times \frac{d_{model}}{num\_heads}$
  - 가중치는 훈련 과정에서 학습됨
  - 논문에서의 경우  $d_{model} = 512, num\_heads = 8$   
이므로 각 벡터에 3개의 서로 다른 가중치  
행렬을 곱하고 64의 크기를 가지는 Q, K, V  
벡터 획득



- Scaled dot-product Attention

- Q, K, V 벡터를 얻었다면 나머지는 기존에 배운 어텐션 메커니즘과 동일

- 반복

- 각 Q벡터는 모든 K벡터에 대해서 어텐션 스코어를 구하고,
    - 어텐션 분포를 구한 뒤에
    - 이를 사용하여 모든 V벡터를 가중합하여 어텐션 값 또는 컨텍스트 벡터를 확보
    - 이를 모든 Q벡터에 대해서 반복

- 단, 어텐션 스코어를 구하는 어텐션 함수의 종류가 다름

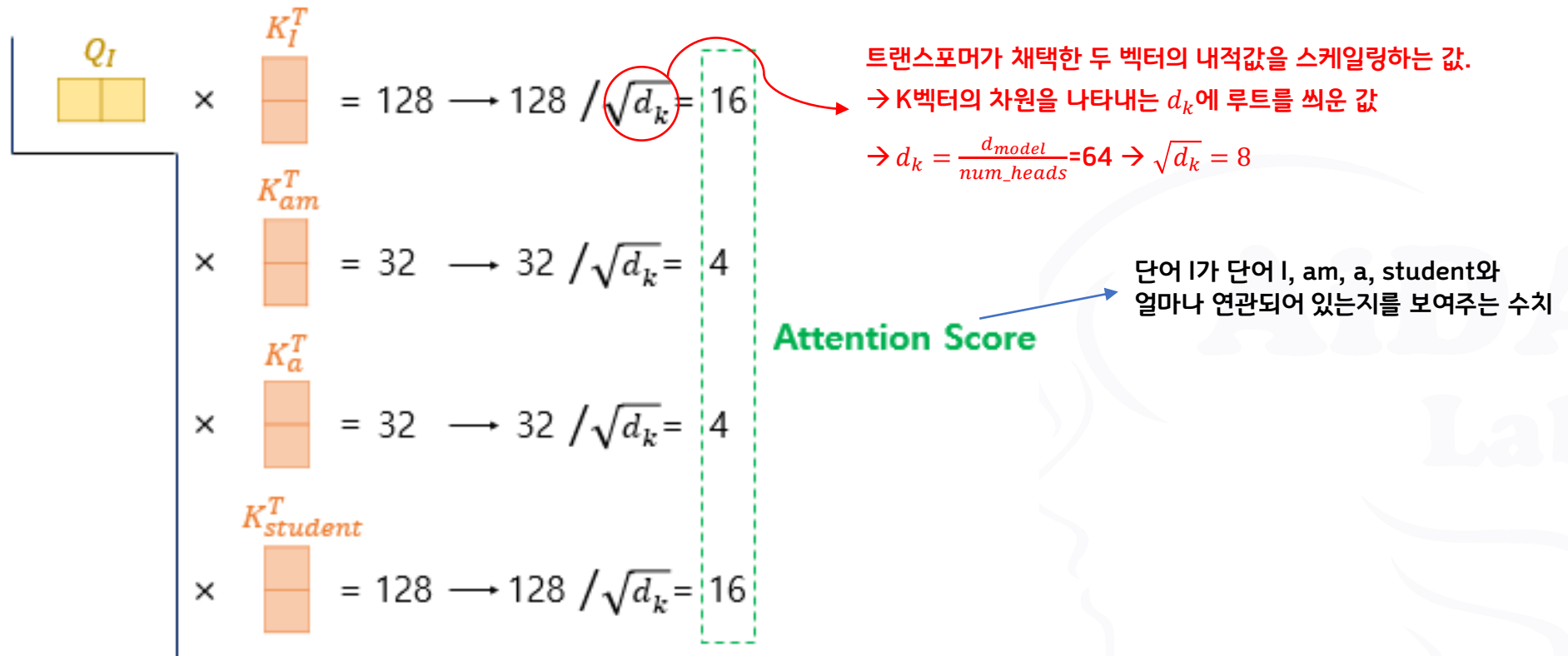
→ 트랜스포머에서는 기존의 Dot-Product Attention의 어텐션 함수  $score(q, k) = q \cdot k$  를 특정 값으로 나눈

$score(q, k) = \frac{q \cdot k}{\sqrt{n}}$  를 사용 → Scaled dot-product Attention

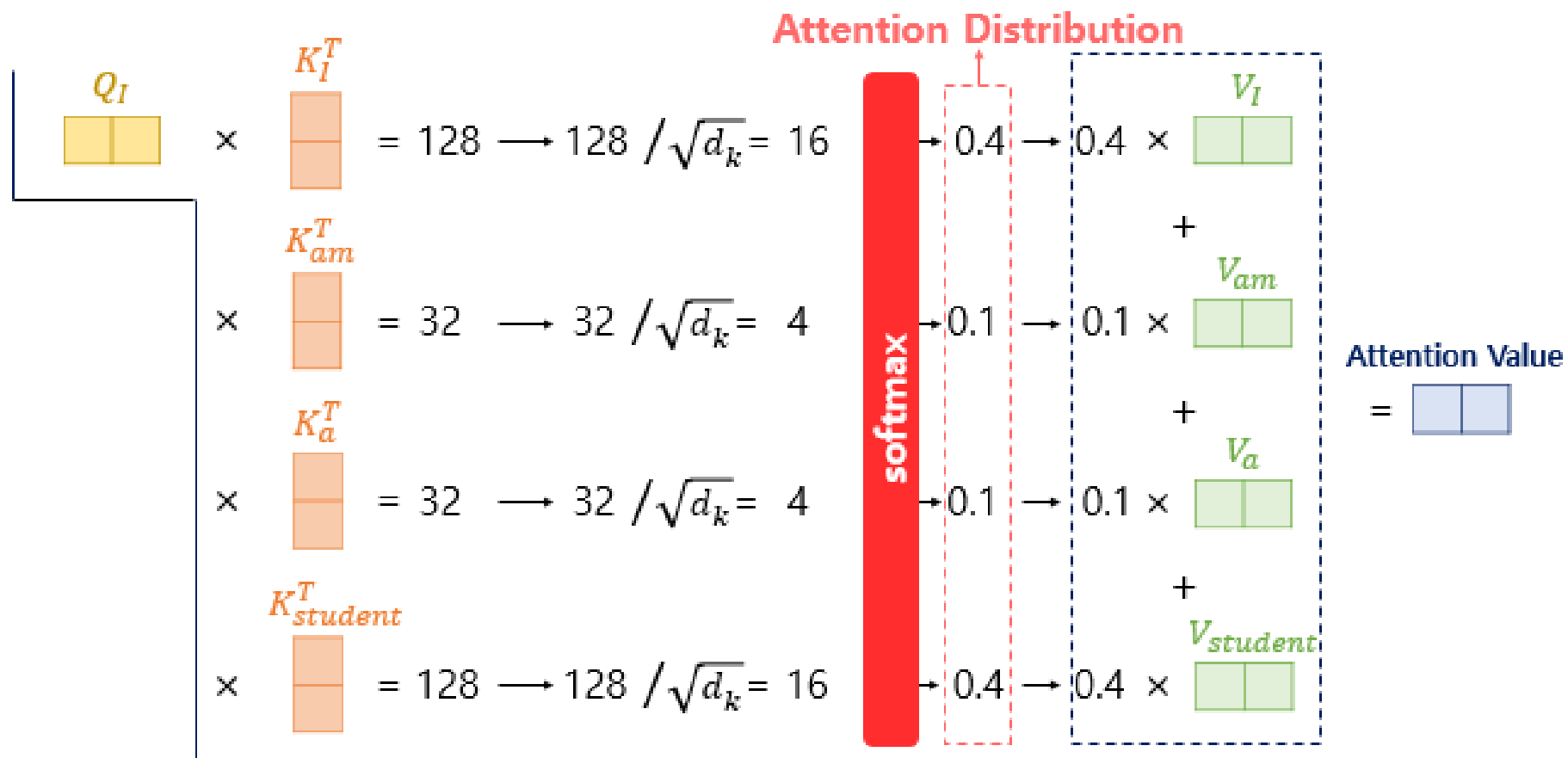
dot-product attention에서 값을 스케일링하는 것을 추가하였다고 하여 Scaled dot-product Attention

- 단어 “I” 에 대한 Q 벡터를 기준으로...

Scaled dot product Attention :  $score\ function(q, k) = q \cdot k / \sqrt{n}$

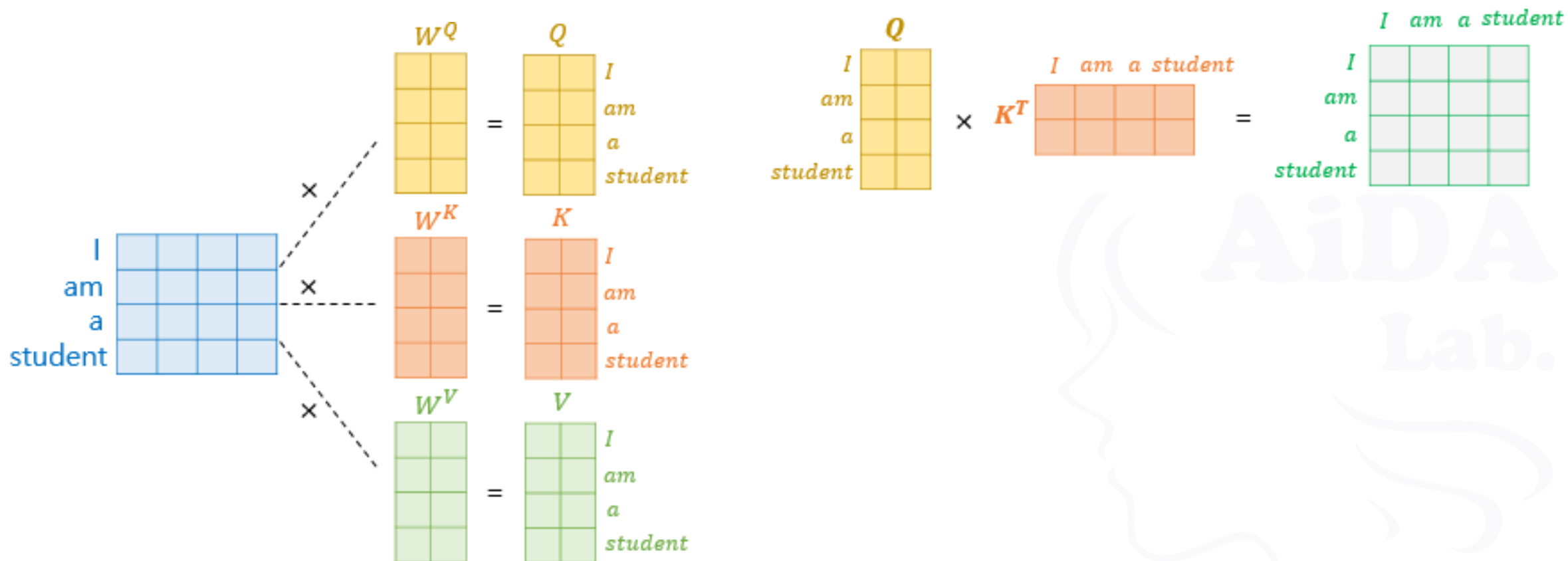


단어 I에 대한 Q벡터가 모든 K벡터에 대해서 어텐션 스코어를 구하는 과정





- 굳이 각 Q벡터마다 따로 연산할 필요가 있을까? → 행렬 연산으로 일괄 처리하기



결과 행렬의 값에 전체적으로  $\sqrt{d_k}$ 를 나누어주면  $\rightarrow$  각 행과 열이 어텐션 스코어 값을 가지는 행렬이 됨

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

행렬 연산을 통해 모든 값이 일괄 계산되는 과정

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

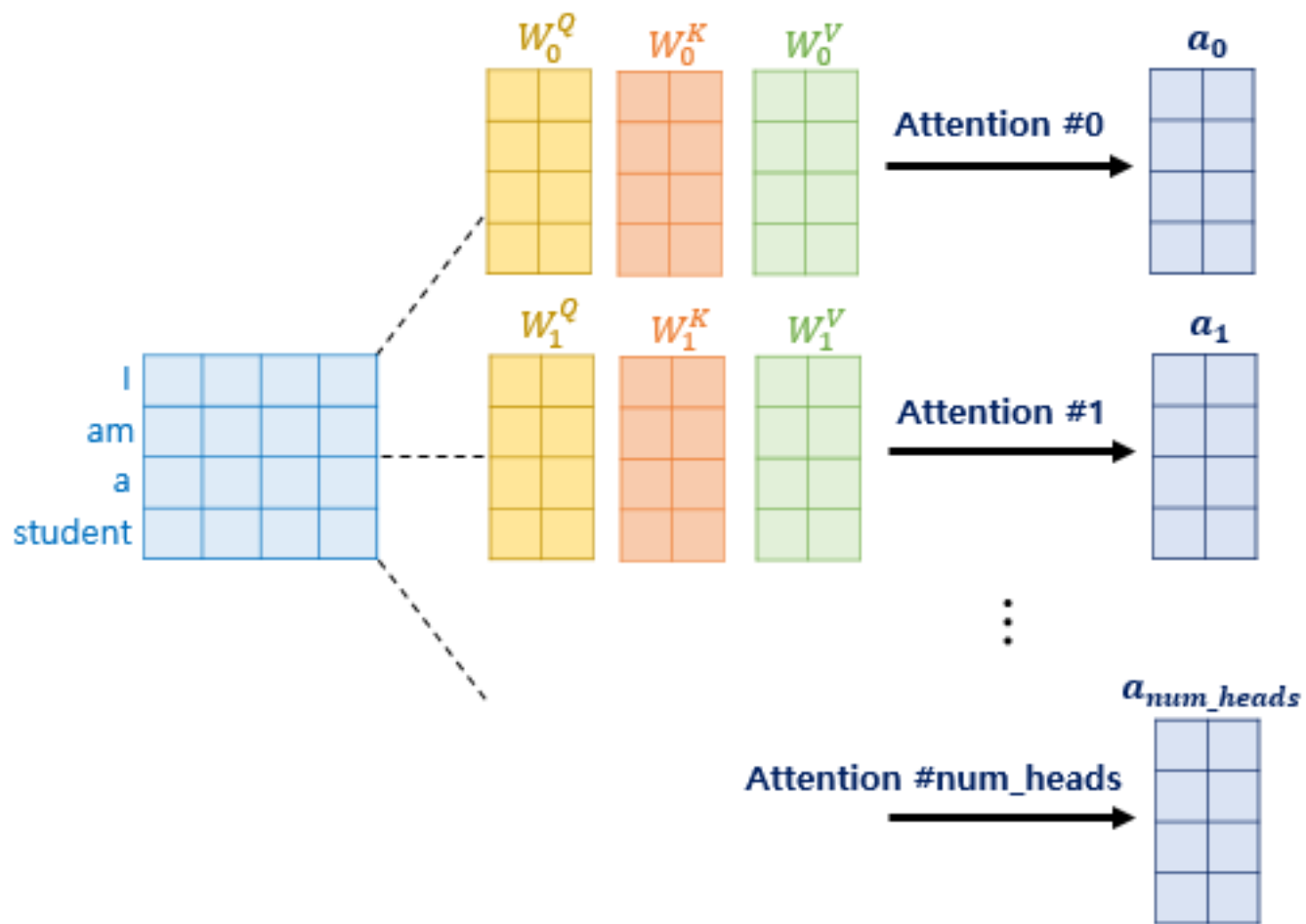
트랜스포머 논문에 기재된 아래의 수식과 정확하게 일치

- 행렬연산에 사용된 행렬의 크기를 정리하면...
  - 입력 문장의 길이:  $seq\_len \rightarrow$  문장 행렬의 크기는  $(seq\_len, d_{model})$
  - 문장 행렬의 크기에 3개의 가중치 행렬을 곱해서  $\rightarrow$  Q, K, V 행렬 만들기
  - Q벡터와 K벡터의 차원:  $d_k$ , V벡터의 차원:  $d_v$  이라고 하면
    - $\rightarrow$  Q행렬과 K행렬의 크기는  $(seq\_len, d_k)$ , V행렬의 크기는  $(seq\_len, d_v)$
    - $\rightarrow$  문장 행렬과 Q, K, V 행렬의 크기로부터 가중치 행렬의 크기 추정 가능
  - $W^Q, W^K$ 의 크기:  $(d_{model}, d_k)$ ,  $W^V$ 의 크기:  $(d_{model}, d_v)$   
(단,  $d_k$ 와  $d_v$ 의 차원의 크기는  $\frac{d_{model}}{num\_heads}$ )
  - 최종적으로  $softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$ 의 결과로 나오는 어텐션 값 행렬의 크기는  $(seq\_len, d_v)$

- 멀티 헤드 어텐션

- 앞서 배운 어텐션에서는  $d_{model}$ 의 차원을 가진 단어 벡터를  $\rightarrow$  num\_heads로 나눈 차원을 가지는 Q, K, V 벡터로 바꾸고 어텐션을 수행
- 왜  $d_{model}$ 의 차원을 가진 단어 벡터를 가지고 어텐션을 하지 않고 차원을 축소시킨 벡터로 어텐션을 수행하였는가?





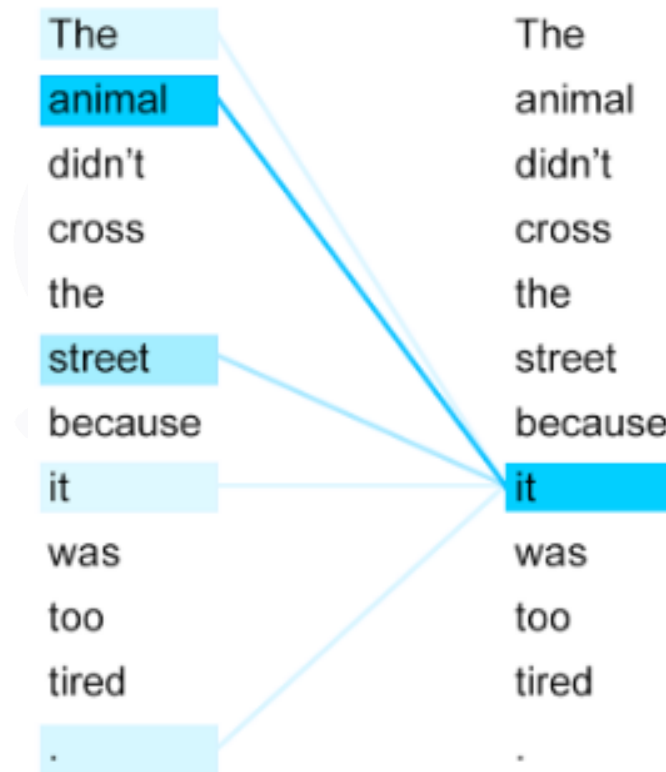
한 번의 어텐션을 하는 것보다 여러번의 어텐션을 병렬로 사용하는 것이 더 효과적이라고 판단

num\_heads만큼 수행

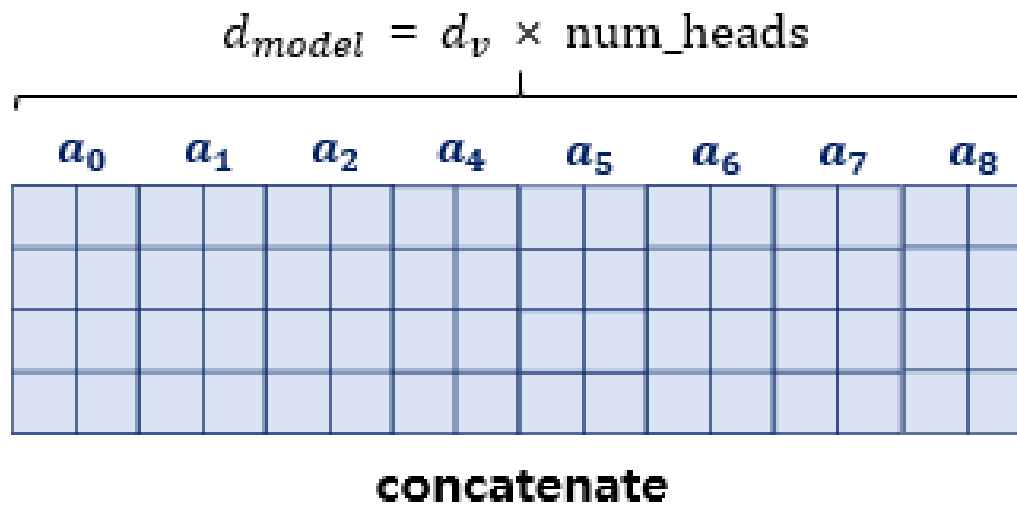
$d_{\text{model}}$ 의 차원을 num\_heads개로 나누어  
 $\frac{d_{\text{model}}}{\text{num\_heads}}$ 의 차원을 가지는 Q, K, V에 대하여  
 num\_heads개의 병렬 어텐션 수행

각 어텐션 값 행렬  $\rightarrow$  어텐션 헤드

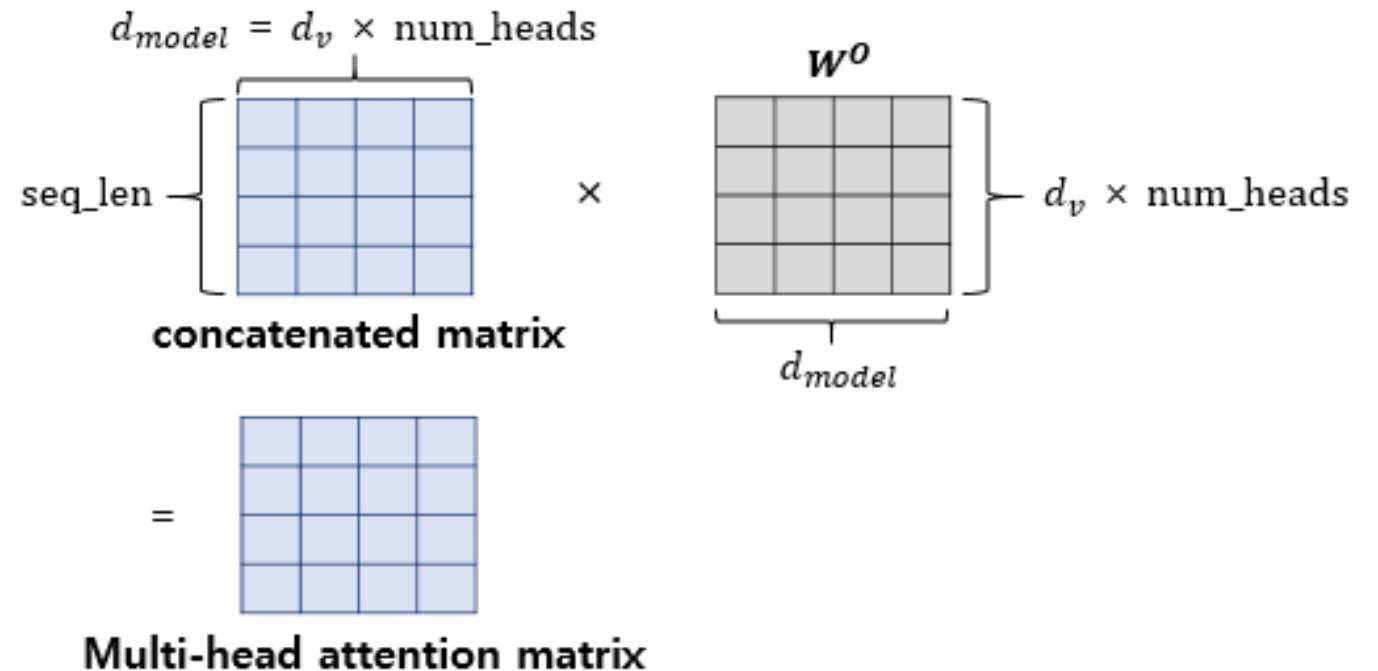
- 병렬 어텐션으로 얻을 수 있는 효과는?
  - 어텐션을 병렬로 수행하여 다른 시각으로 정보들을 수집하겠다는 것
  - 예시
    - '그 동물은 길을 건너지 않았다. 왜냐하면 그것은 너무 피곤하였기 때문이다.'
    - 단어 그것(it)이 쿼리였다고 하면
      - 즉, it에 대한 Q벡터로부터 다른 단어와의 연관도를 구하였을 때 첫번째 어텐션 헤드는 '그것(it)'과 '동물(animal)'의 연관도를 높게 본다면
      - 두번째 어텐션 헤드는 '그것(it)'과 '피곤하였기
      - 때문이다(tired)'의 연관도를 높게 볼 수 있다



- 병렬 어텐션 수행 후, 모든 어텐션 헤드를 연결(concatenate)  
→ 모두 연결된 어텐션 헤드 행렬의 크기는  $(seq\_len, d_{model})$



- 어텐션 헤드를 모두 연결한 행렬에 또 다른 가중치 행렬  $W^O$  을 곱하기  
→ 이 결과 행렬이 멀티-헤드 어텐션의 최종 결과물임
- 결과물인 멀티-헤드 어텐션 행렬은 인코더의 입력이었던 문장 행렬의 크기 ( $seq\_len, d_{model}$ ) 와 동일함  
→ 인코더의 입력으로  
들어왔던 행렬의 크기가  
아직 유지되고 있음





- 패딩 마스크 (Padding Mask)

- <PAD>가 포함된 입력 문장의 셀프 어텐션

$$\begin{array}{c}
 \begin{array}{c} Q \\ I \\ am \\ sam \\ <pad> \end{array}
 \begin{array}{|c|c|} \hline \hline \hline \hline \hline \end{array}
 \times K^T
 \begin{array}{c} I \quad am \quad sam \quad <pad> \\ \begin{array}{|c|c|c|c|} \hline \hline \hline \hline \hline \end{array} \\
 \end{array}
 =
 \begin{array}{c}
 \begin{array}{c} I \quad am \quad sam \quad <pad> \\ \begin{array}{|c|c|c|c|} \hline \hline \hline \hline \hline \end{array} \\
 \end{array}
 \end{array}$$

$\sqrt{d_k}$

Attention Score Matrix

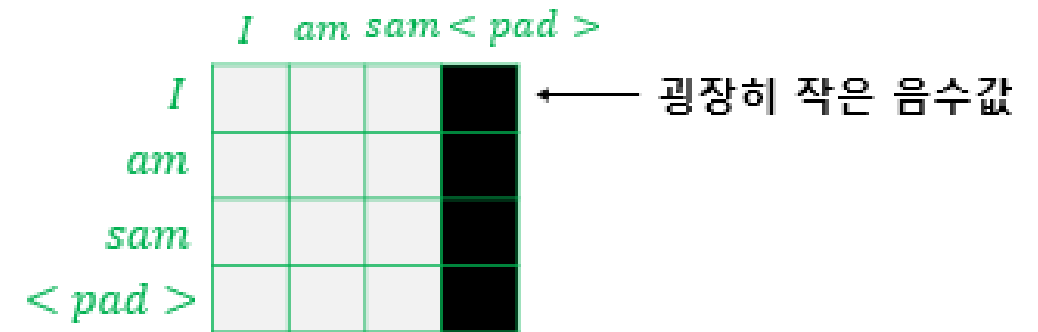
어텐션을 수행하고 어텐션 스코어 행렬을 얻는 과정

- <PAD>의 경우에는 실질적인 의미를 가진 단어가 아니므로  
→ 트랜스포머에서는 Key의 경우에 <PAD> 토큰이 존재한다면 이에 대해서는 유사도를 구하지 않도록 마스킹 (Masking)을 적용함
  - 마스킹: 어텐션에서 제외하기 위해 값을 가린다는 의미
- 어텐션 스코어 행렬에서
  - 행에 해당하는 문장은 Query
  - 열에 해당하는 문장은 Key
  - Key에 <PAD>가 있는 경우에는 해당 열 전체를 마스킹



- 마스크를 하는 방법

- 어텐션 스코어 행렬의 마스크 위치에 매우 작은 음수 값을 넣어주는 것
- 매우 작은 음수 값이라는 것은 -1,000,000,000과 같은 -무한대에 가까운 수라는 의미



- 현재 어텐션 스코어 함수는 소프트맥스 함수를 지나지 않은 상태

- 어텐션 스코어 함수는 소프트맥스 함수를 지나고, 그 후 Val
- 현재 마스크 위치에 매우 작은 음수 값이 들어가 있으므로
- 어텐션 스코어 행렬이 소프트맥스 함수를 지난 후에는 해당 위치의 값은 0이 됨  
→ 단어 간 유사도를 구하는 일에 <PAD> 토큰이 반영되지 않게 됨

**Attention Score Matrix**

- 어텐션 스코어 함수가 소프트맥스 함수를 지난 후라면...
  - 각 행의 어텐션 가중치의 총 합은 1
  - 그러나 단어 <PAD>의 경우에는 0이 되어 어떤 유의미한 값을 가지고 있지 않음

	<i>I</i>	<i>am</i>	<i>sam</i>	<i>&lt; pad &gt;</i>
<i>I</i>	0.7	0.2	0.1	0
<i>am</i>				
<i>sam</i>				
<i>&lt; pad &gt;</i>				

Attention Score Matrix

## • 포지션-와이즈 피드 포워드 신경망(Position-wise FFNN)

- 인코더와 디코더에서 공통적으로 가지고 있는 서브층이며 완전 FFNN(Fully-connected FFNN)

- 포지션 와이즈 FFNN의 수식:

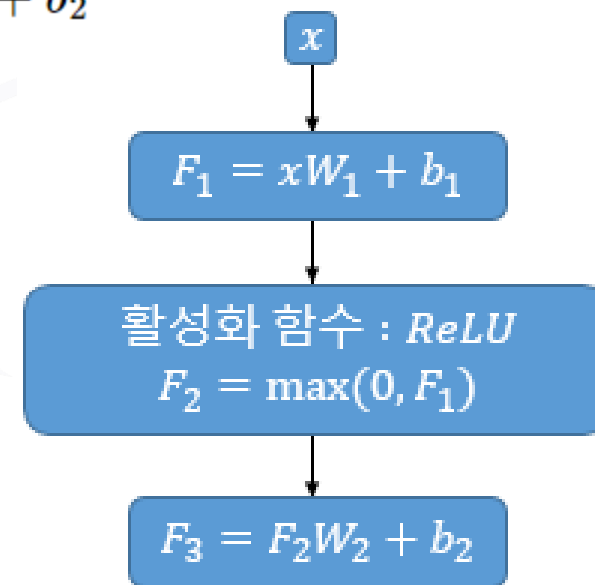
- $x$ : 멀티 헤드 어텐션의 결과로 나온  $(seq\_len, d_{model})$ 의 크기를 가지는 행렬

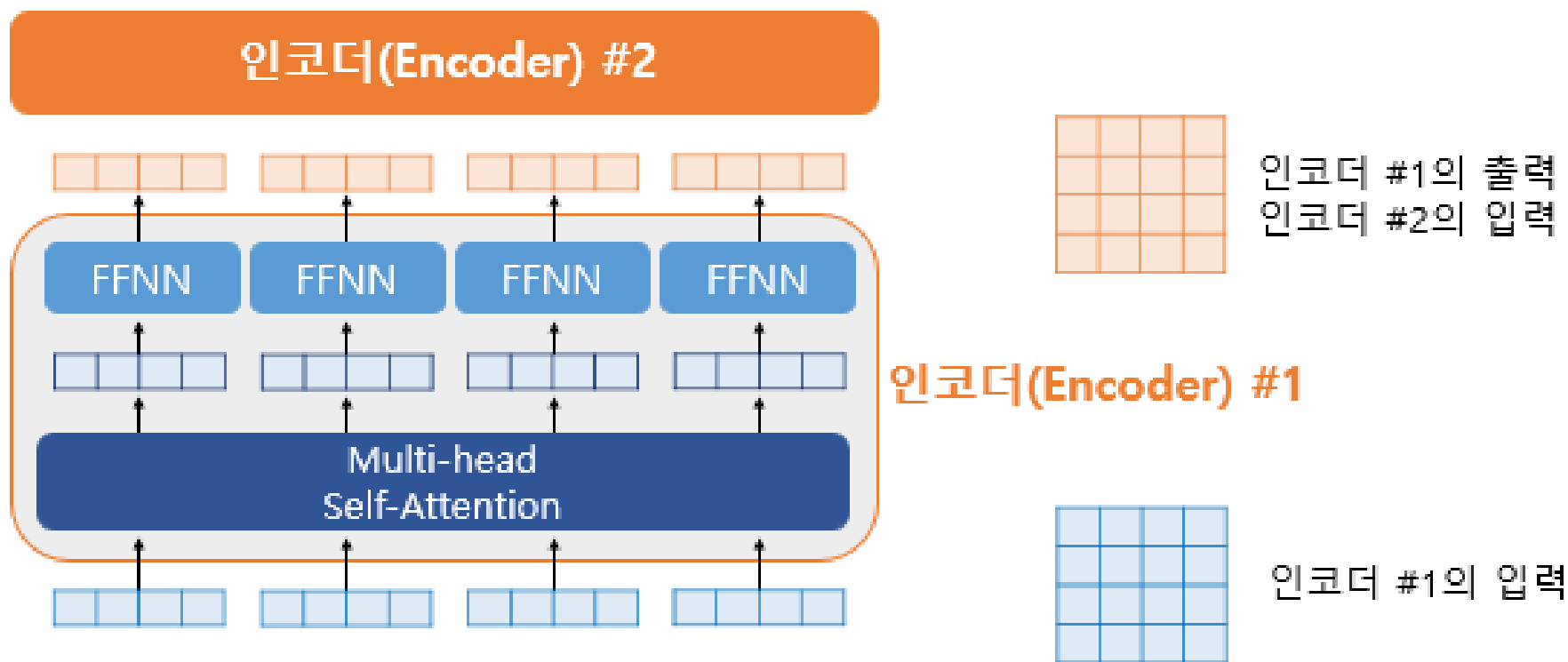
- 가중치 행렬  $W_1$ 의 크기:  $(d_{model}, d_{ff})$

- 가중치 행렬  $W_2$ 의 크기:  $(d_{ff}, d_{model})$

- $d_{ff}$ : 은닉층의 크기, 논문에서는  $d_{ff} = 2048$

- 매개변수  $W_1, b_1, W_2, b_2$ : 하나의 인코더 층 내에서는 다른 문장, 다른 단어들마다 정확하게 동일하게 사용. 하지만 인코더 층마다 다른 값을 사용





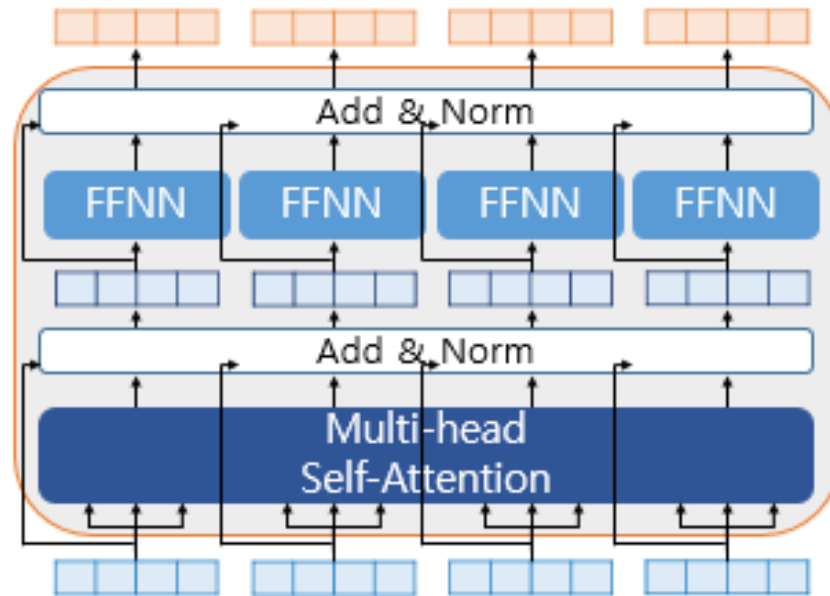
- 인코더의 입력을 벡터 단위로 봤을 때, 각 벡터들이 멀티 헤드 어텐션 층이라는 인코더 내 첫번째 서브 층을 지나 FFNN을 통과하는 모습  
→ 두번째 서브층인 Position-wise FFNN을 의미

- 실제로는 그림의 우측과 같이 행렬로 연산
- 두번째 서브층을 지난 인코더의 최종 출력은 여전히 인코더의 입력의 크기였던 ( $seq\_len, d_{model}$ )의 크기가 보존됨

인코더 층을 지난 이 행렬은 다음 인코더 층으로 전달되고, 다음 층에서도 동일한 인코더 연산이 반복

- 잔차 연결(Residual connection)과 층 정규화(Layer Normalization)

- 트랜스포머에서는 인코더의 두 개의 서브층에 추가적으로 Add & Norm을 적용
  - Add : 잔차 연결(residual connection)
  - Norm: 층 정규화(layer normalization)



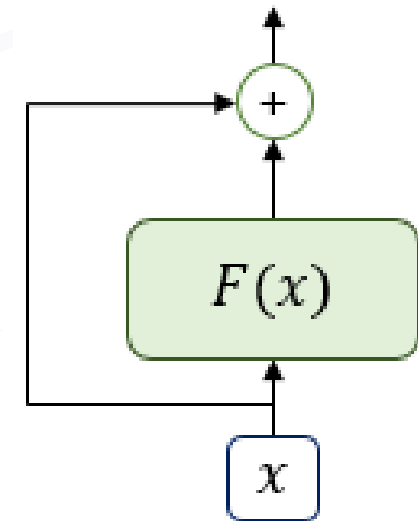
인코더(Encoder) #1

인코더 블록에  
화살표와 Add & Norm(잔차 연결과 층 정규화 과정)을  
추가한 그림

- 잔차 연결(Residual connection)
  - 서브층의 입력과 출력을 더하는 것
  - 트랜스포머에서 서브층의 입력과 출력은 동일한 차원을 갖고 있으므로, 서브층의 입력과 서브층의 출력은 덧셈 연산이 가능
    - 앞 페이지의 인코더 그림에서 각 화살표가 서브층의 입력에서 출력으로 향하도록 그려졌던 이유
  - 잔차 연결은 컴퓨터 비전 분야에서 주로 사용되는, 모델의 학습을 돕는 기법
  - 수식으로 표현하면

$$x + \text{Sublayer}(x)$$

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

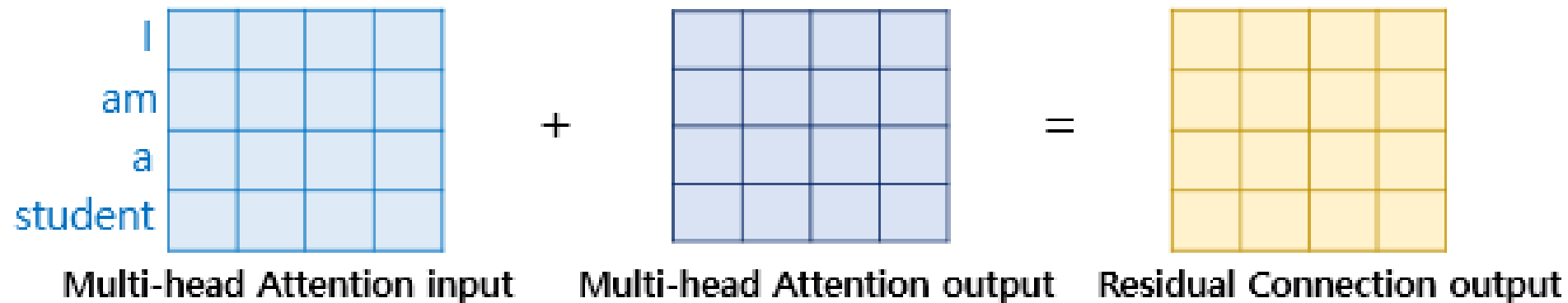


입력  $x$ 와  $x$ 에 대한 어떤 함수  $F(x)$ 의 값을 더한 함수  $H(x)$ 의 구조  
어떤 함수  $F(x)$ 가 트랜스포머에서는 서브층에 해당함



- 서브층이 멀티 헤드 어텐션인 경우, 잔차 연결 연산 수식

$$H(x) = x + \text{Multi-head Attention}(x)$$



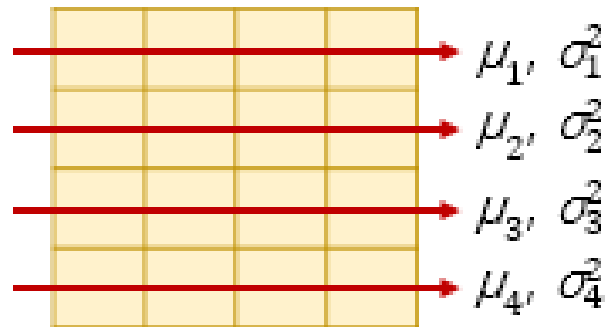
멀티 헤드 어텐션의 입력과 멀티 헤드 어텐션의 결과가 더해지는 과정

- 층 정규화(Layer Normalization)
  - 잔차 연결을 거친 결과는 이어서 층 정규화 과정을 거치게 됨
  - 잔차 연결의 입력을  $x$ , 잔차 연결과 층 정규화 두 가지 연산을 모두 수행한 후의 결과 행렬을  $LN$ 이라고 하였을 때, 잔차 연결 후 층 정규화 연산을 수식으로 표현하면

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



- 층 정규화: 텐서의 마지막 차원에 대해서 평균과 분산을 구하고, 이를 가지고 어떤 수식을 통해 값을 정규화하여 학습을 돕는 작업
  - 텐서의 마지막 차원: 트랜스포머에서는  $d_{model}$  차원



Residual Connection output  
(seq\_len,  $d_{model}$ )

$d_{model}$  차원의 방향을 화살표로 표현



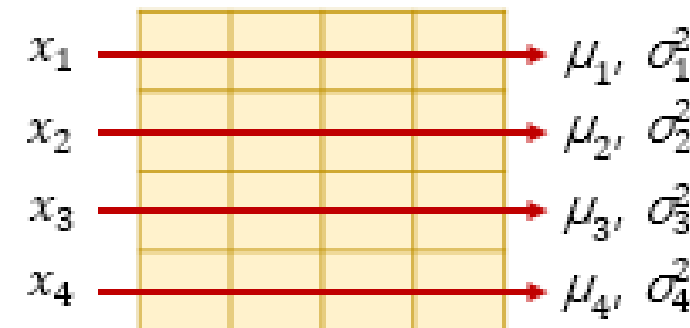
- 층 정규화를 위해서 화살표 방향으로 각각 평균  $\mu$ 과 분산  $\sigma^2$  계산
- 각 화살표 방향의 벡터는  $x_i$
- 정규화 수행 후  $x_i$ 는  $ln_i$ 라는 벡터로 정규화 됨

$$ln_i = LayerNorm(x_i)$$

- 평균과 분산을 통한 정규화
  - 평균과 분산을 통해 벡터  $x_i$ 를 정규화
  - $x_i$ 는 벡터, 평균  $\mu_i$ 과 분산  $\sigma_i^2$ 은 스칼라이므로
  - 벡터  $x_i$ 의 각 차원을  $k$ 라고 하였을 때,  $x_{i,k}$ 는 다음의 수식과 같이 정규화 됨

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

- $\epsilon$ (입실론)은 분모가 0이 되는 것을 방지하는 값



Residual Connection output

- 감마와 베타 도입

- $\gamma$ (감마)와  $\beta$ (베타)라는 벡터 준비 (초기값은 각각 1과 0)

 $\gamma$ 

1	1	1	1
---	---	---	---

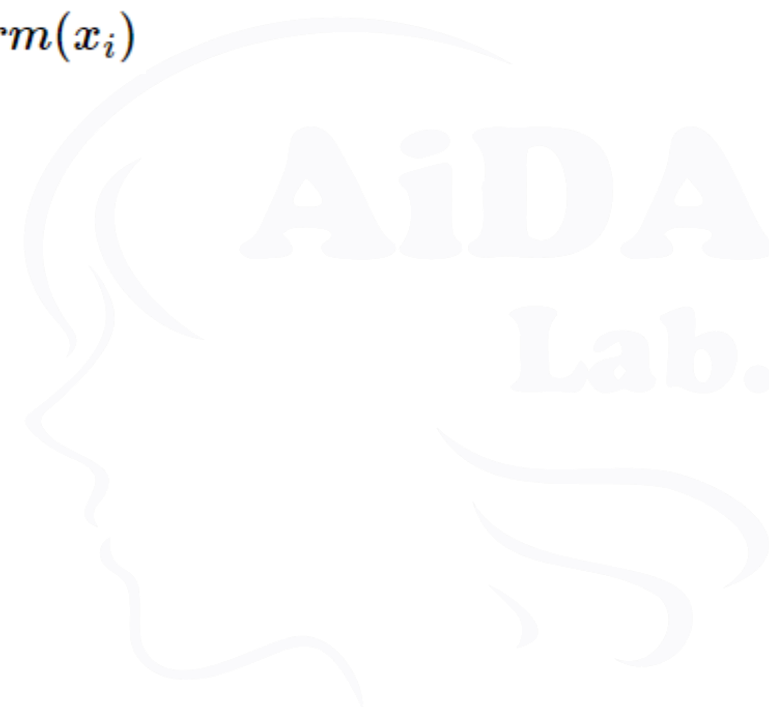
 $\beta$ 

0	0	0	0
---	---	---	---

- $\gamma$ 와  $\beta$  를 도입한 층 정규화의 최종 수식

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

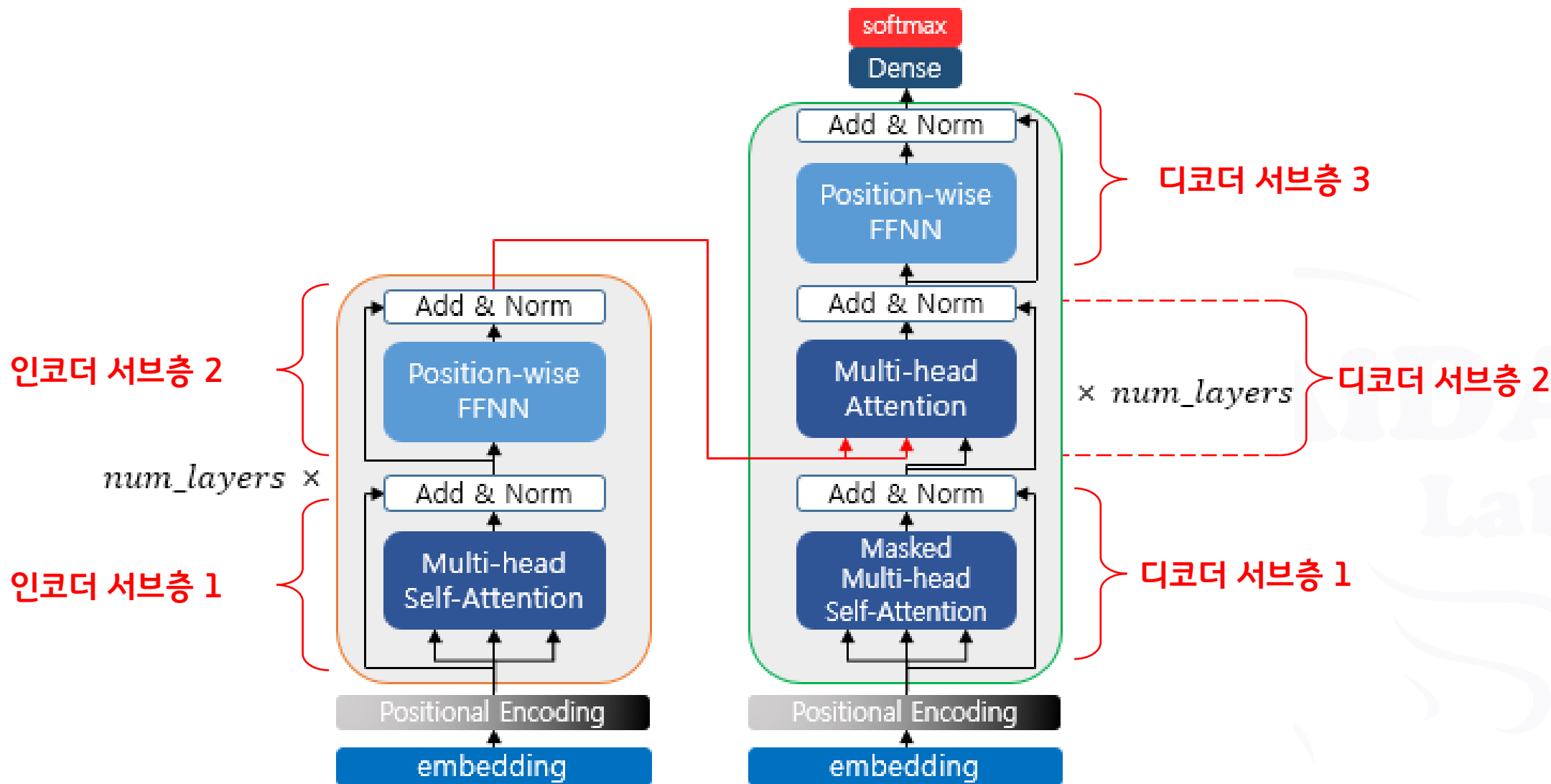
- $\gamma$ 와  $\beta$ 는 학습 가능한 파라미터



- 인코더 → 디코더

- 인코더는 총 num\_layers 만큼의 층 연산을 순차적으로 한 후
- 마지막 층의 인코더의 출력을 디코더에게 전달
- 디코더도 num\_layers 만큼의 연산을 수행
- 매 연산마다 인코더가 보낸 출력을 각 디코더 층 연산에 사용

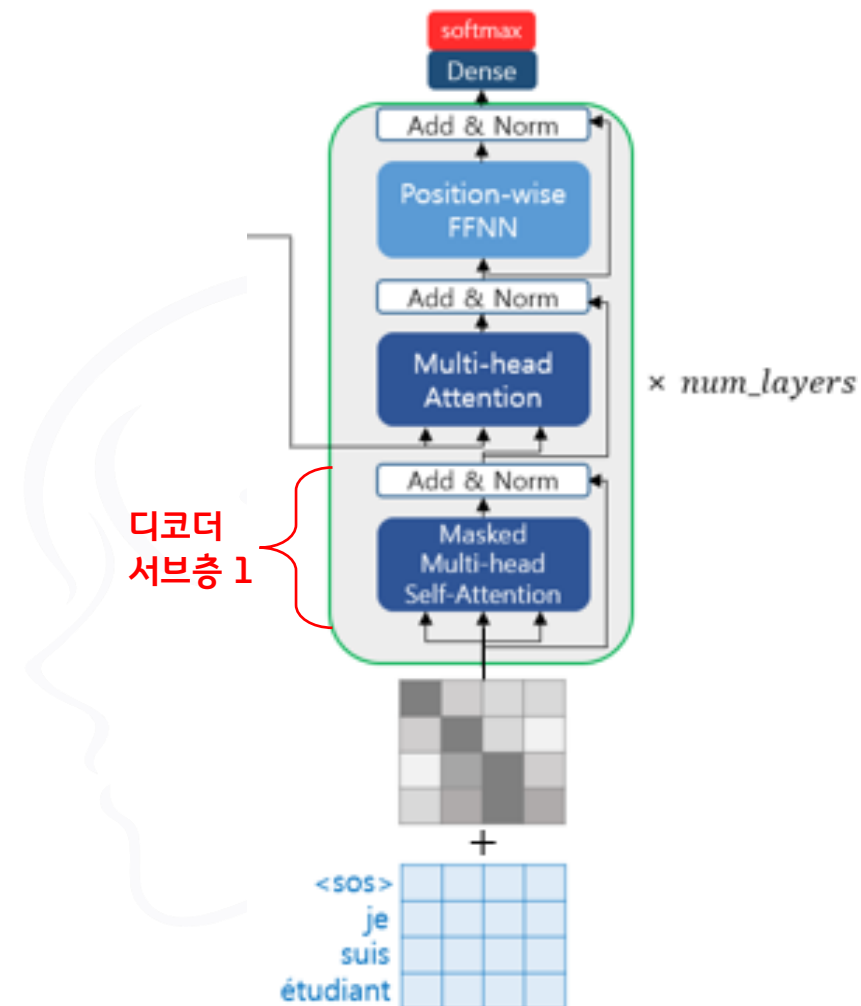




## • 디코더의 서브층 (1): 마스크드 멀티헤드 셀프 어텐션

- 셀프 어텐션 + 록-어헤드 마스크
- 디코더도 인코더와 동일하게 임베딩 층과 포지셔널 인코딩을 거친 후의 문장 행렬이 입력
- 디코더는 학습 과정에서 번역할 문장에 해당되는 **<sos> je suis étudiant**의 문장 행렬을 한 번에 입력받음  
→ 입력된 문장 행렬로부터 각 시점의 단어를 예측 하도록 훈련됨

트랜스포머는 교사 강요(Teacher Forcing)을 사용하여 훈련을 수행





- **교사 강요(Teacher Forcing)**

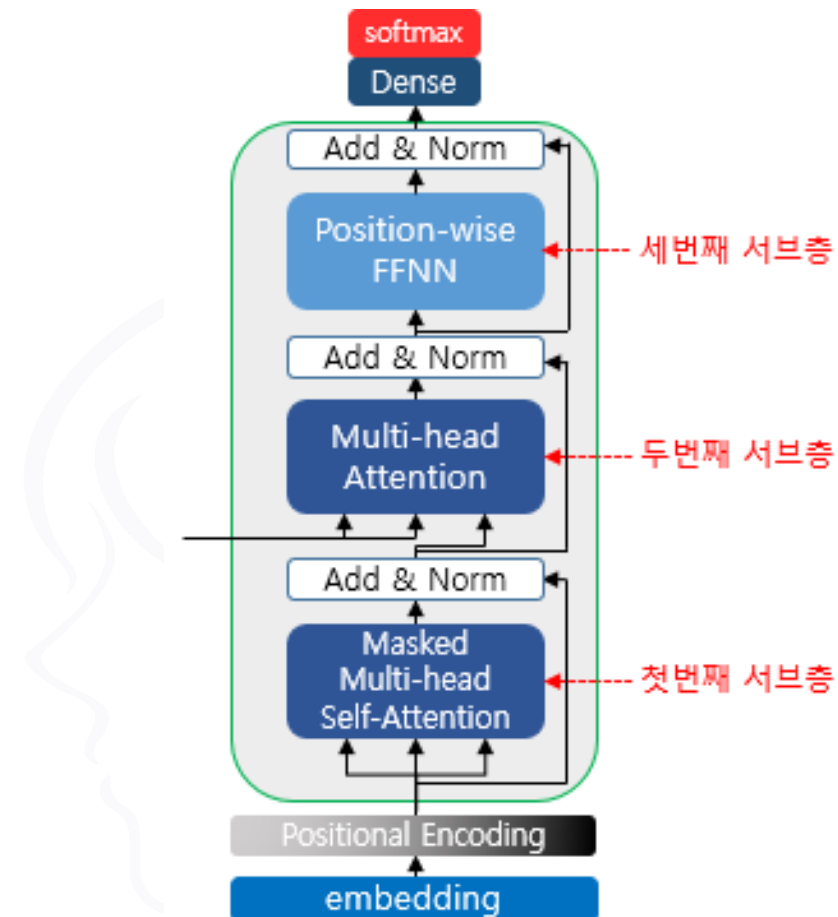
- 트랜스포머 아키텍처는 seq2seq를 기반으로 하고 있으며, seq2seq는 RNN을 기반으로 함
- 현재 시점의 디코더 셀의 입력은 오직 이전 디코더 셀의 출력을 입력으로 받는다고 설명하였는데 decoder\_input이 왜 필요할까?
- 이전 시점의 디코더 셀의 예측이 틀렸는데 이를 현재 시점의 디코더 셀의 입력으로 사용하면
  - 현재 시점의 디코더 셀의 예측도 잘못될 가능성이 높고
  - 이는 연쇄 작용으로 디코더 전체의 예측을 어렵게 만들며
  - 이런 상황이 반복되면 훈련 시간이 길어짐
- 이전 시점의 디코더 셀의 예측값 대신 실제 값을 현재 시점의 디코더 셀의 입력으로 사용함으로써 이러한 상황을 회피하는 방법 → 교사 강요(Teacher Forcing)

## • 문제점

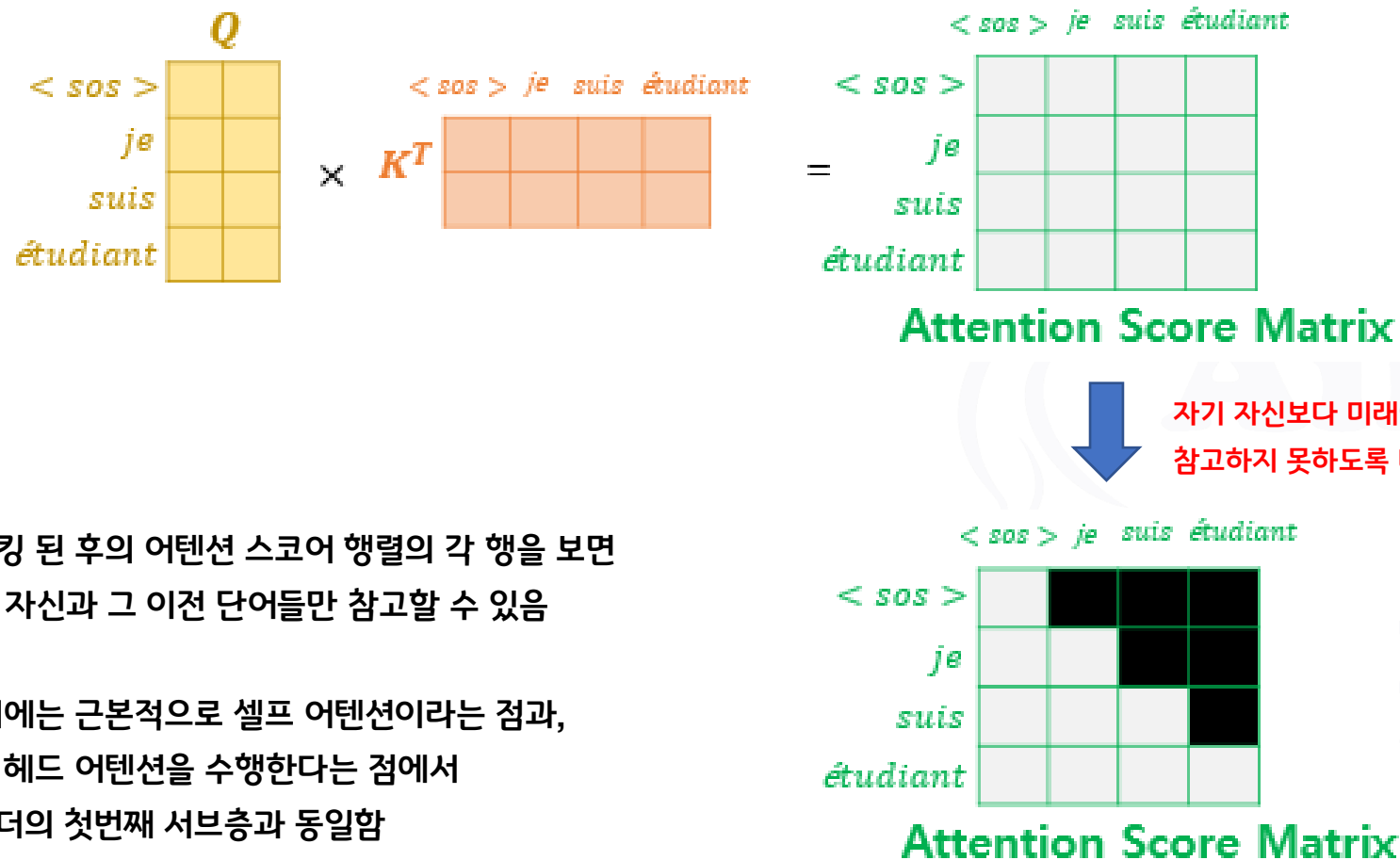
- seq2seq의 디코더에 사용되는 RNN 계열의 신경망은
    - 입력 단어를 매 시점마다 순차적으로 입력 받으므로
    - 다음 단어 예측에 현재 시점을 포함한 이전 시점에 입력된 단어들만 참고할 수 있음
  - 트랜스포머는
    - 문장 행렬로 입력을 한 번에 받으므로
    - 현재 시점의 단어를 예측하고자 할 때, 입력 문장 행렬로부터 미래 시점의 단어까지도 참고할 수 있는 현상이 발생
- ➔ 트랜스포머의 디코더에서는 현재 시점의 예측에서 현재 시점보다 미래에 있는 단어들을 참고하지 못하도록 룩-어헤드 마스크(look-ahead mask, 미리보기에 대한 마스크)를 도입

- 록-어헤드 마스크(look-ahead mask)

- 디코더의 첫번째 서브층(멀티 헤드 셀프 어텐션)에서 이루어짐
- 멀티 헤드 셀프 어텐션 층
  - 인코더의 첫번째 서브층인 멀티 헤드 셀프 어텐션 층과 동일한 연산을 수행
  - 다른 점: 어텐션 스코어 행렬에서 마스킹을 적용한다는 점만 다름



- 록-어헤드 마스크를 적용한 어텐션 스코어 행렬 계산



- 마스킹 된 후의 어텐션 스코어 행렬의 각 행을 보면 자기 자신과 그 이전 단어들만 참고할 수 있음
- 그 외에는 근본적으로 셀프 어텐션이라는 점과, 멀티 헤드 어텐션을 수행한다는 점에서 인코더의 첫번째 서브층과 동일함

## • 트랜스포머 아키텍처에 존재하는 어텐션

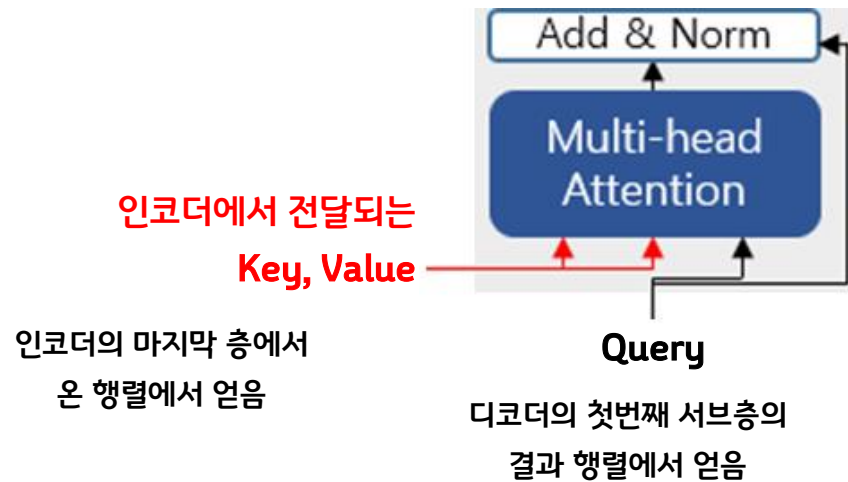
- 인코더의 셀프 어텐션
  - 패딩 마스크를 전달
- 디코더의 마스크드 셀프 어텐션(첫번째 서브층)
  - 록-어헤드 마스크를 전달  
(록-어헤드 마스크를 한다고 해서 패딩 마스크가 불필요한 것은 아님)
- 디코더의 인코더-디코더 어텐션(두번째 서브층)
  - 패딩 마스크를 전달

- 같은 점
  - 모두 멀티 헤드 어텐션을 수행하고,
  - 멀티 헤드 어텐션 함수 내부에서 스케일드 닷 프로덕트 어텐션 함수를 호출
- 다른 점
  - 각 어텐션 시, 함수에 전달하는 마스크는 다름

## • 디코더의 서브층 (2): 인코더-디코더 어텐션

### • 셀프 어텐션이 아님

- Query는 디코더 행렬, Key와 Value는 인코더 행렬이기 때문



Query가 디코더 행렬, Key가 인코더 행렬일 때,  
어텐션 스코어 행렬을 구하는 과정

$$\begin{matrix} & Q \\ \begin{matrix} < sos > \\ je \\ suis \\ \acute{e}tudiant \end{matrix} & \begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}
 \end{matrix}
 \times
 \begin{matrix} K^T \\ \begin{matrix} I & am & a & student \end{matrix} \\ \begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}
 \end{matrix}
 =
 \begin{matrix} & I & am & a & student \\ \begin{matrix} < sos > \\ je \\ suis \\ \acute{e}tudiant \end{matrix} & \begin{bmatrix} & & & & \\ & & & & \\ & & & & \\ & & & & \end{bmatrix}
 \end{matrix}$$

Attention Score Matrix

- 그 외에 멀티 헤드 어텐션을 수행하는 과정은 다른 어텐션들과 동일함

## • 잔차연결(Residual Connectio) 은 왜 필요한가?

### • 잔차 연결이란?

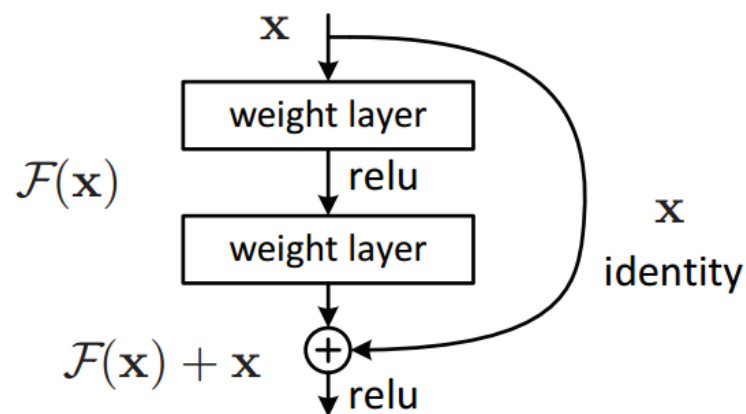
- 블록이나 레이어 계산을 건너뛰는 경로를 하나 두는 것

### • 구현

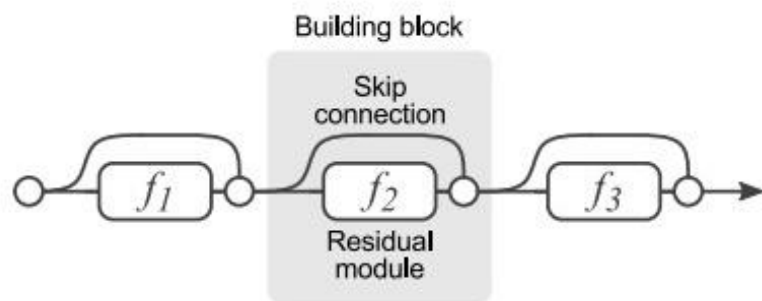
- 입력을  $x$ , 계산 대상 블록을  $F$ 라고 할 때  
잔차 연결은  $F(x) + x$ 로 간단히 실현

### • 효과

- 동일한 블록 계산이 계속 반복될 때,  
모델이 다양한 관점에서 블록 계산을 수행할 수 있도록 한다

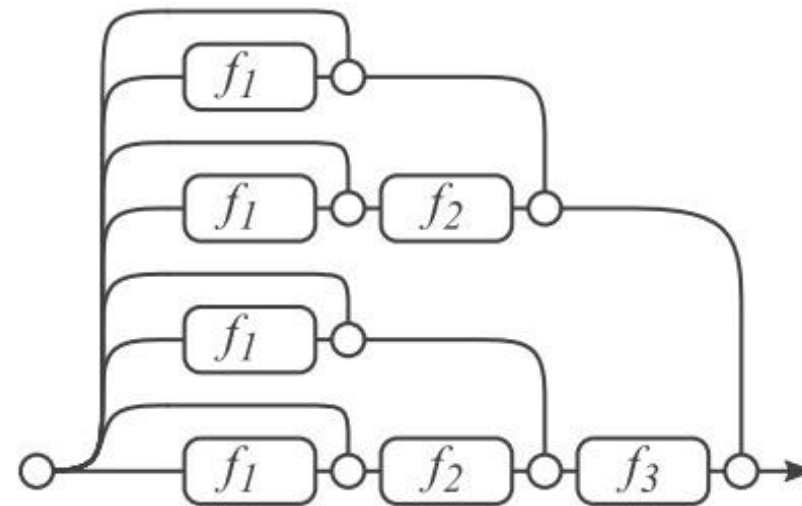


## • 잔차연결을 두지 않았을 때와 비교



(a) Conventional 3-block residual network

=



(b) Unraveled view of (a)

- 잔차연결을 두지 않았을 때는  $f_1$ ,  $f_2$ ,  $f_3$  을 연속으로 수행하는 경로 한 가지만 존재
- 잔차 연결을 블록마다 설정해 둌으로써 모두 8가지의 새로운 경로가 생김  
→ 모델이 다양한 관점에서 블록 계산을 수행하게 됨



- 잔차연결의 필요성 및 효과

- 딥러닝 모델은

- 레이어가 많아지면
    - 모델을 업데이트하기 위한 신호(그라디언트)가 전달되는 경로가 길어지기 때문에
    - 학습이 어려워지는 경향이 있음

- 잔차연결의 도입은

- 모델 중간에 블록을 건너뛰는 경로를 설정함으로써
    - 학습을 용이하게 하는 효과까지 얻을 수 있음



## • 레이어 정규화(Layer Normalization)

- 미니 배치의 인스턴스( $x$ )별로 평균을 빼 주고, 표준편차로 나누어서 정규화(normalization)을 수행하는 기법

$$y = \frac{x - \mathbb{E}[x]}{\sqrt{\mathbb{V}[x] + \epsilon}} * \gamma + \beta$$

- $x$ : 미니 배치의 인스턴스
- $\mathbb{E}[x]$ : 미니 배치의 인스턴스  $x$ 의 평균
- $\mathbb{V}[x]$ : 미니 배치의 인스턴스  $x$ 의 표준편차
- $\gamma$ : 학습과정에서 업데이트되는 가중치, 모델의 weight에 대응. 1로 초기화 됨
- $\beta$ : 학습과정에서 업데이트되는 가중치, 모델의 bias에 대응. 0으로 초기화 됨

- 효과: 레이어 정규화를 수행하면 학습이 안정되고 속도가 빨라짐
- $\gamma$ 와  $\beta$ 는 왜 1과 0으로 초기화 하는가?
  - 1을 곱하고 마지막으로 0을 더해준다는 이야기 → 학습 외에는 인위적인 변경을 주지 않음

THANK  
YOU

