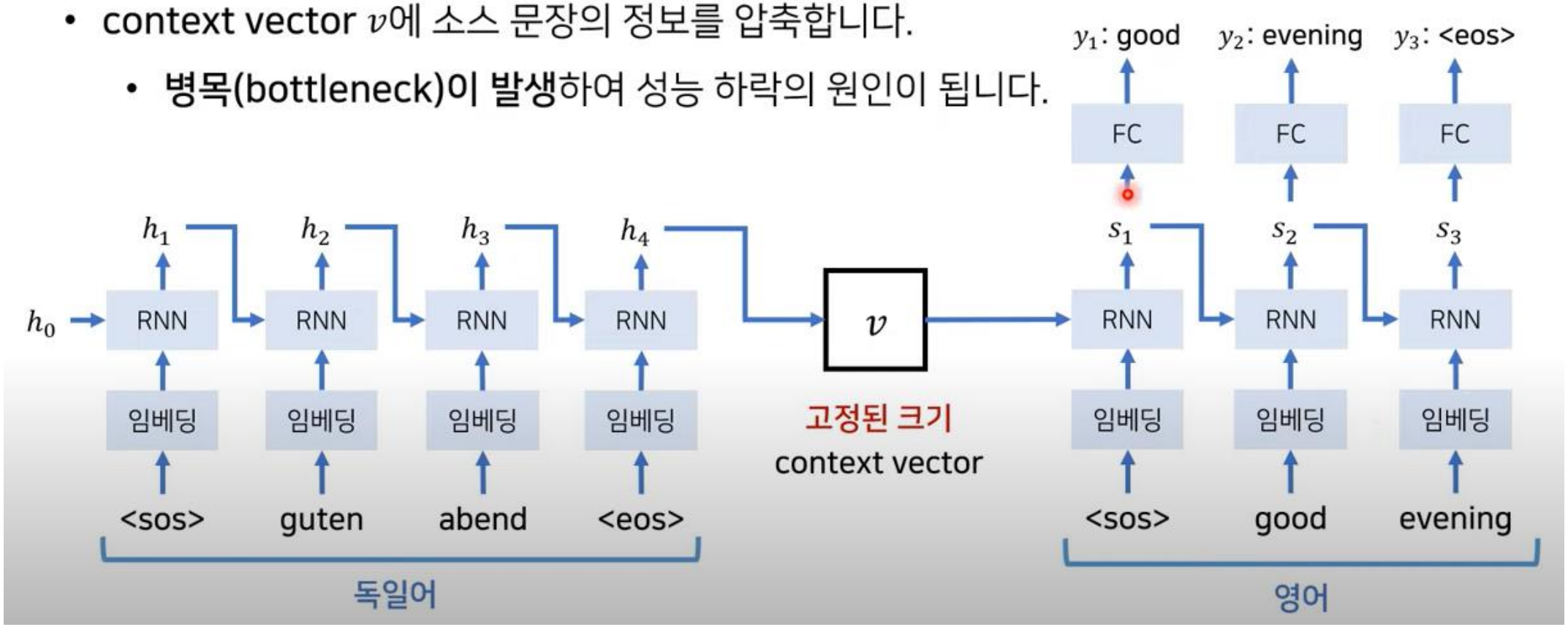
Transformer

1. 기존 seq2seq의 문제점

Encoder가 입력 시퀀스를 하나의 vector로 압축하는 과정에서 입력 시퀀스 정보가 일부 손실됨.



2. Transformer의 등장

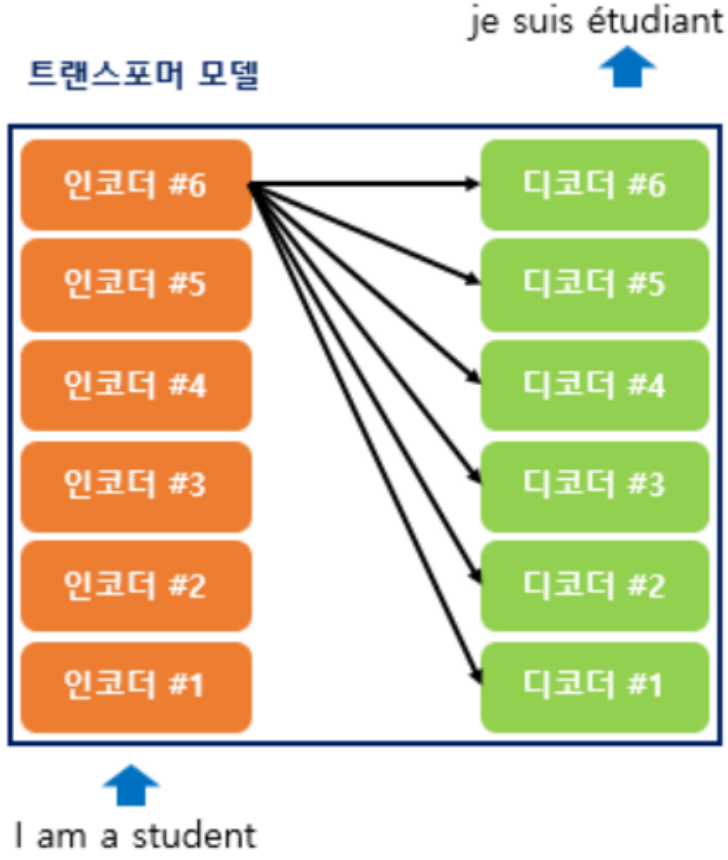
seq2seq :

인코더, 디코더 t개의 시점에 RNN(LSTM)이 존재함.

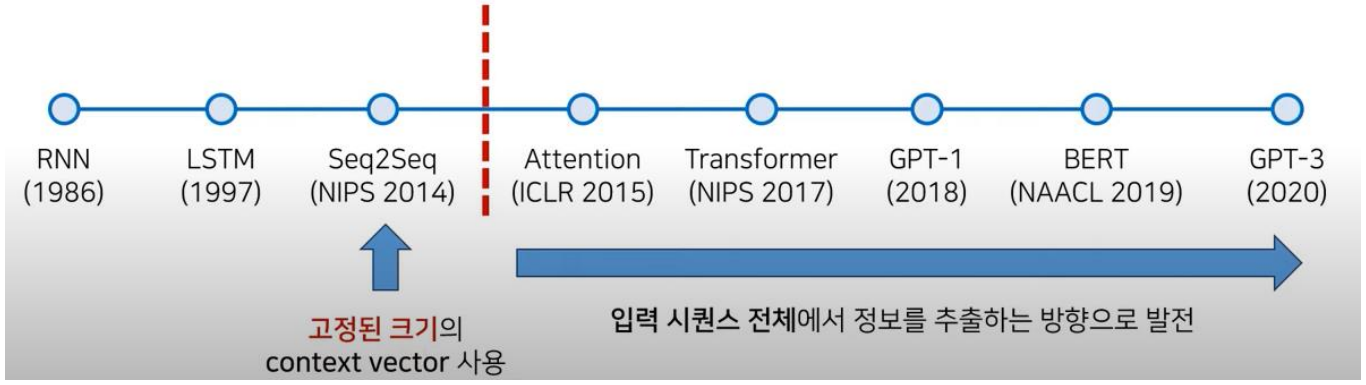
Transformer :

CNN, RNN 기법은 아예 사용하지 않음. 인코더와 디코더가 여러 개 존재하는 구조.

인코더들은 모두 똑같은 구조를 가지고 있지만, weight를 공유하지는 않는다.(디코더도 마찬가지)



- 2021년 기준으로 최신 고성능 모델들은 Transformer 아키텍처를 기반으로 하고 있습니다.
  - **GPT**: Transformer의 **디코더(Decoder)** 아키텍처를 활용
  - **BERT**: Transformer의 **인코더(Encoder)** 아키텍처를 활용



트랜스포머의 원리를 알아봅시다

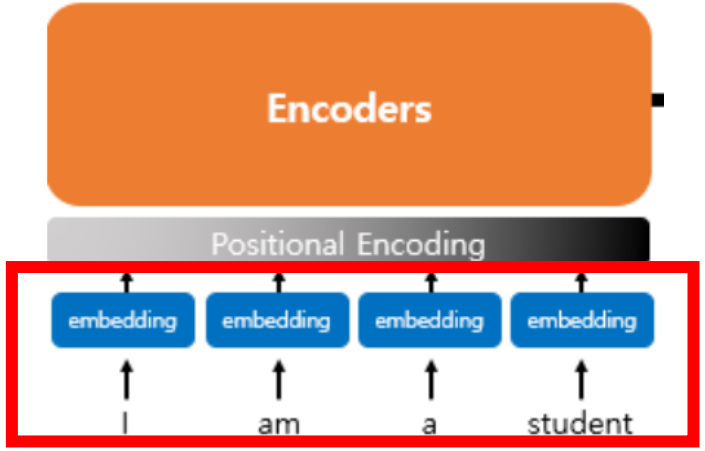
3. Transformer 입력단계

1. 단어들을 n차원 Embedding 벡터로 변환

I			…		
Am			….		
A			….		
Student			….		

n차원

I am a student



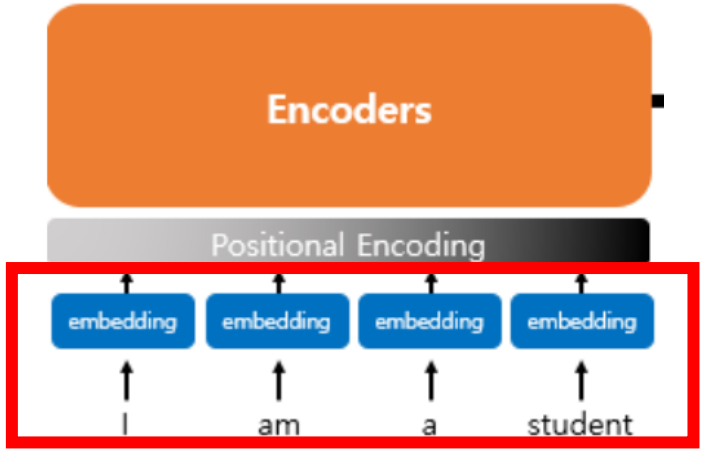
Embedding이란?

단어 간 유사도 및 중요도 파악을 위해 저 차원(512, 1024)의 실수 벡터로 mapping하여 의미적으로 비슷한 단어를 가깝게 배치(분산표현, Distributed Representation)

단어 의미 상의 관계를 반영하여 기하학적 공간에 반영함.

1) Word vector 임의로 생성, 학습하면서 적절한 값을 얻는 방식
2) Pretrained model을 이용해 Word Embedding하는 방식

3. Transformer 입력단계



1. 단어들을 n차원 Embedding 벡터로 변환

Seq2Seq :

단어를 순차적으로 입력 받기 때문에(RNN) 단어의 위치정보를 가질 수 있음.

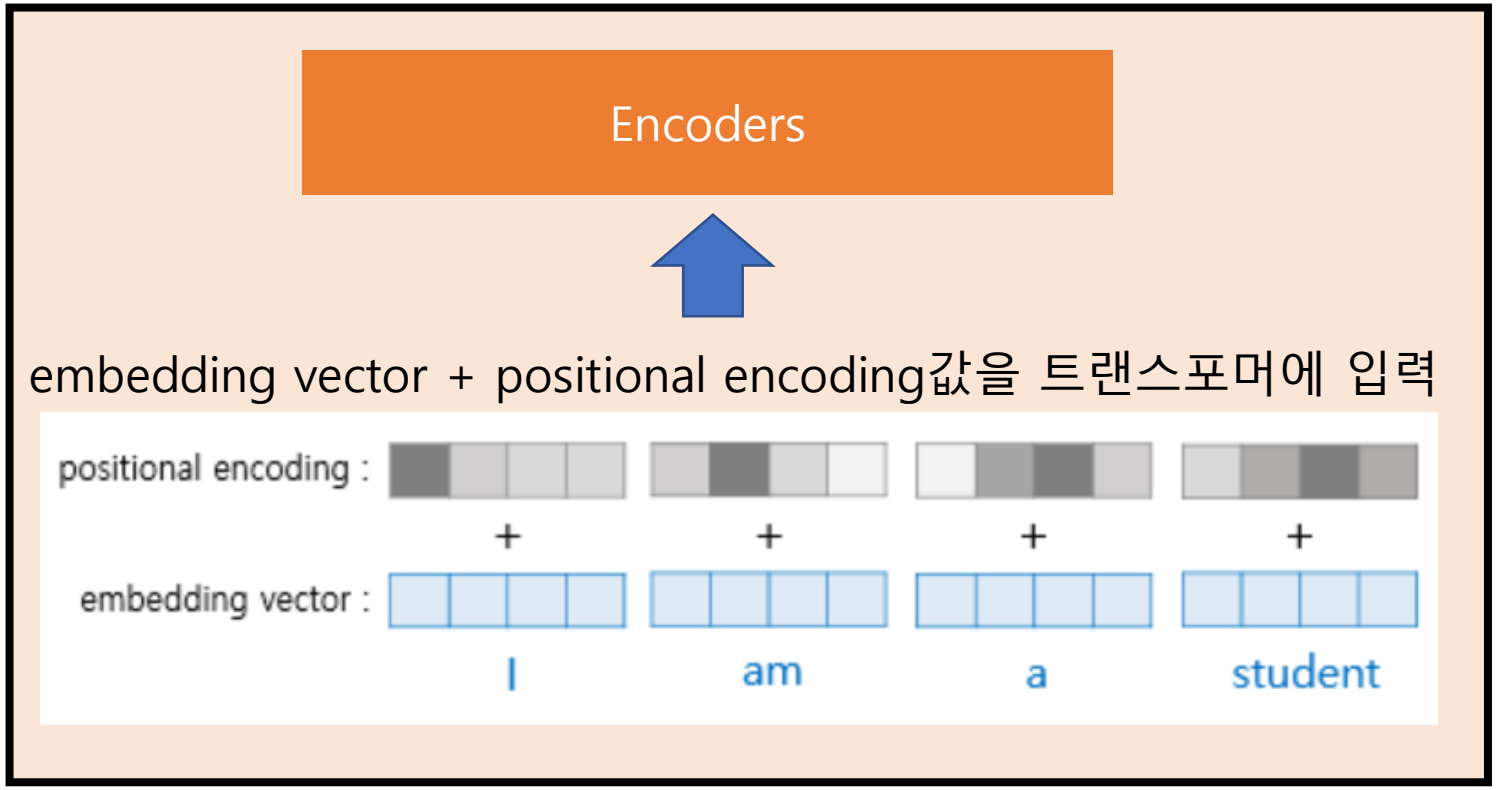
Transformer :

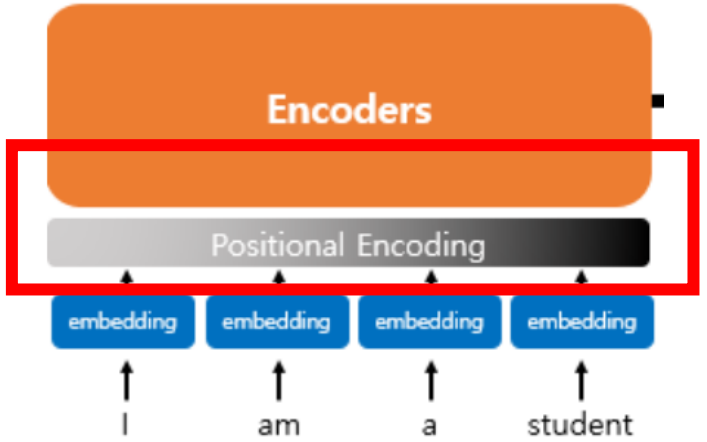
문장 전체를 한꺼번에 입력 받기 때문에 단어의 위치 정보를 가질 수 없다.

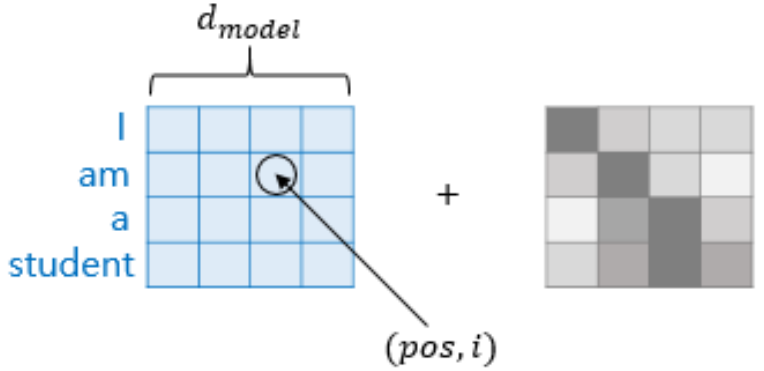
➔ Positional Encoding 기법을 통해 위치정보를 입력해줌

3. Transformer 입력단계

2. Positional Encoding & 결과 값 Encoder에 입력







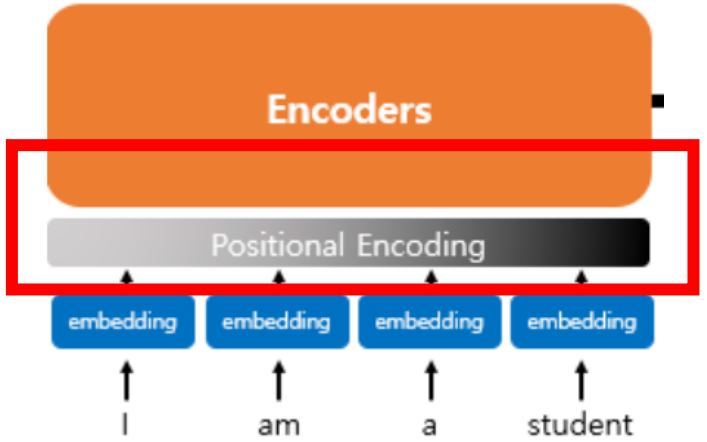
Positional Encoding함수는 sequential property를 나타내주기 위해 주기성을 가진다

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

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

d_model : 임베딩할 때 차원
pos : 입력 문장에서 임베딩 벡터 순서
i : 임베딩 벡터에서 차원의 인덱스

3. Transformer 입력단계



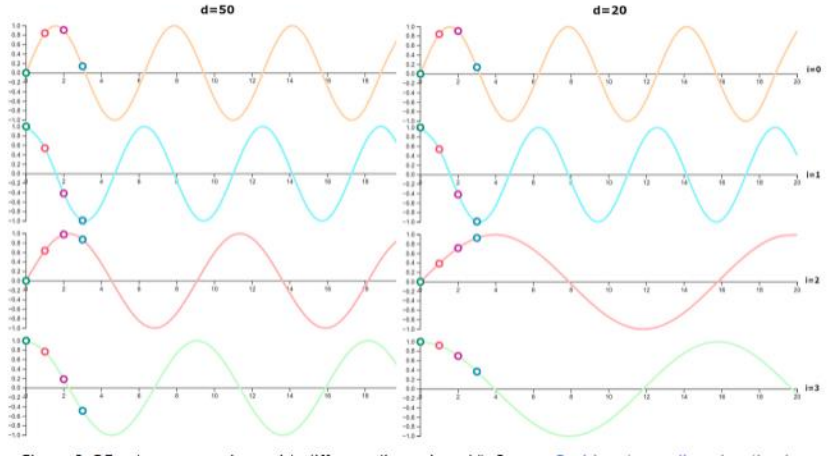
2. Positional Encoding & 결과 값 Encoder에 입력

적절한 positional encoding 함수에 대한 고민,, Why Sin & Cos?

I am a student
Ex1)
I : 1 / am : 2 / a : 3 / student : 4 (숫자가 너무 커진다)
Ex2)
I : 0.1 / am : 0.3 / a : 0.5...? (일반화 불가능)

따라서 **적절한 positional Encoding 함수**가 되기 위해선

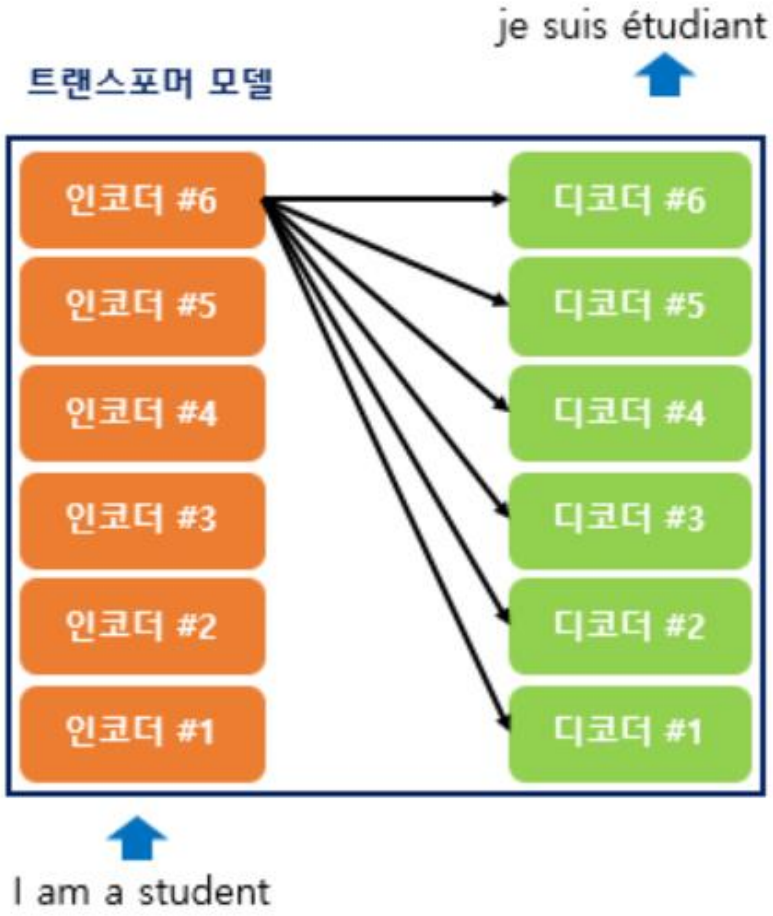
1) 숫자가 어느 특정 범위 내에 있어야 할 것
2) 값을 부여할 때 규칙성이 있어야 할 것(일반화)

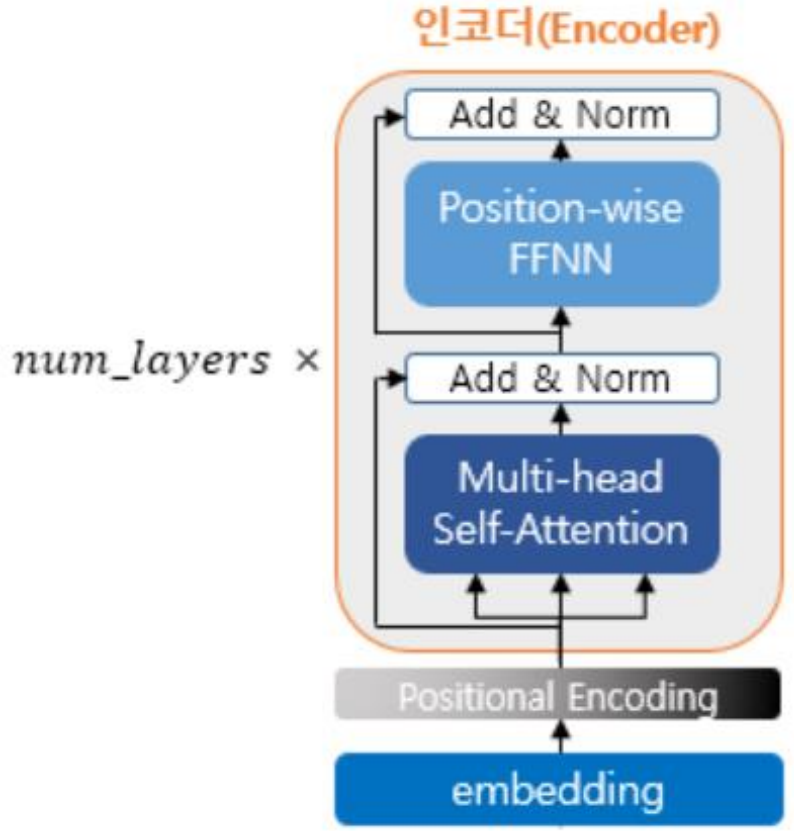


cos, sin함수

4. Transformer Encoding 단계

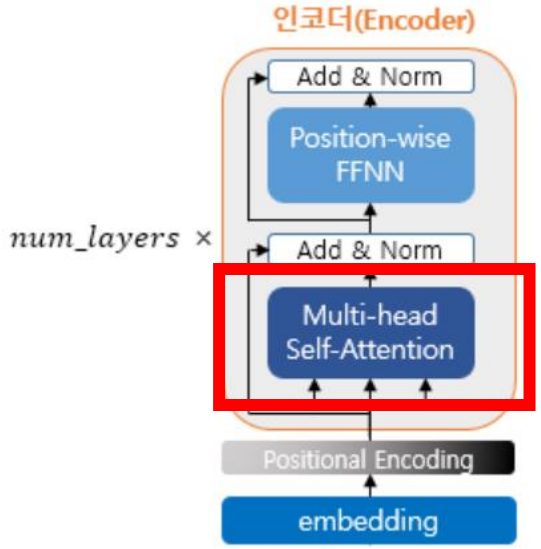
Transformer는 여러 개의 Encoder와 Decoder가 층으로 쌓여 있는 구조이다.





인코더 입력 → Multi-head Attention → Add+Norm → FFNN → Add+Norm

4-1) Encoder의 첫 layer, Multi-head Attention



Multi-head Attention 순서

1) query 행렬과 문장 내 모든 key 행렬을 곱해 attention score를 구함.

2) 스코어 값을 softmax함수에 넣어 attention 분포를 구함.

3) 각 분포 값을 value 벡터와 곱한 뒤 다 더해 attention value를 구함.

ex) The animal didn't cross the street, because **it** was too tired

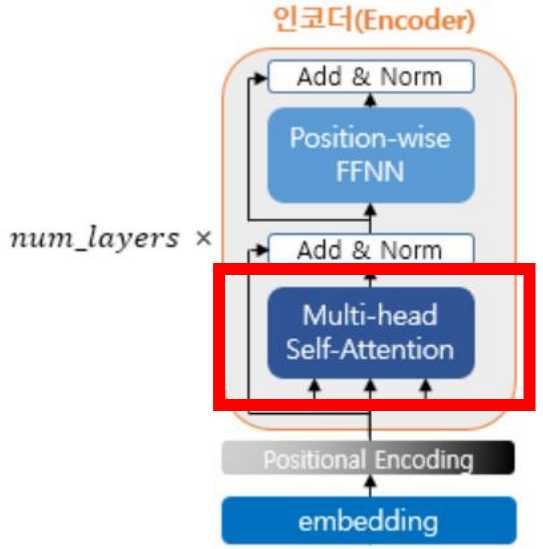
Multi-head Attention을 이용하여

it과 animal이 가장 높은 단어 연관성을 가진다는 것을 발견하여 it이

animal임을 알 수 있다!!

인코더 입력 → Multi-head Attention → Add+Norm → FFNN → Add+Norm

4-1) Encoder의 첫 layer, Multi-head Attention



Multi-head Attention 순서

1) query 행렬과 문장 내 모든 key 행렬을 곱해 attention score를 구함.

2) 스코어 값을 softmax함수에 넣어 attention 분포를 구함.

3) 각 분포 값을 value 벡터와 곱한 뒤 다 더해 attention value를 구함.

ex) The animal didn't cross the street, because **it** was too tired

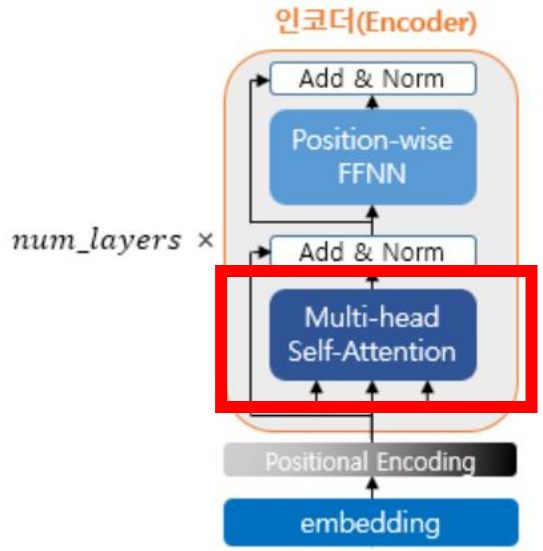
Multi-head Attention을 이용하여

it과 animal이 가장 높은 **단어 연관성**을 가진다는 것을 발견하여 it이

animal임을 알 수 있다!!

인코더 입력 → Multi-head Attention → Add+Norm → FFNN → Add+Norm

4-1) Encoder의 첫 layer, Multi-head Attention



Multi-head Attention 순서

1) query 행렬과 문장 내 모든 key 행렬을 곱해 attention score를 구함.

2) 스코어 값을 softmax함수에 넣어 attention 분포를 구함.

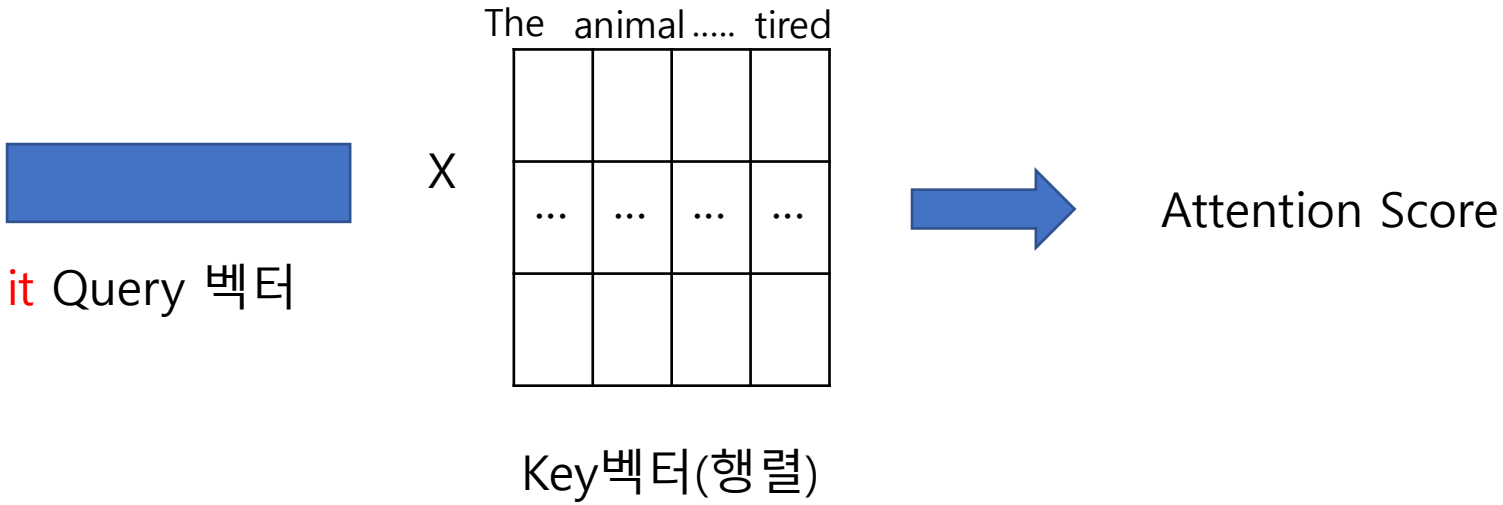
3) 각 분포 값을 value 벡터와 곱한 뒤 다 더해 attention value를 구함.

Attention을 위한 **세 가지** 입력 요소

- 쿼리(Query) : 물어보는 주체
- 키(Key) : 질문을 받는 대상
- 값(Value)

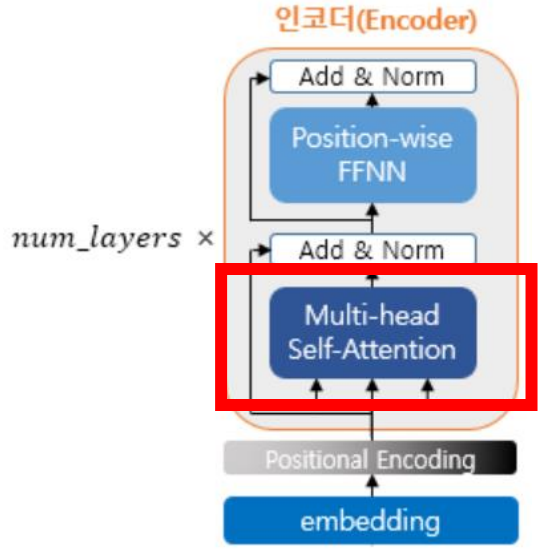
ex) The animal didn't cross the street, because **it** was too tired

이 문장에서 Query는 it, Key는 문자에 있는 11개의 단어들이다.



인코더 입력 → Multi-head Attention → Add+Norm → FFNN → Add+Norm

4-1) Encoder의 첫 layer, Multi-head Attention



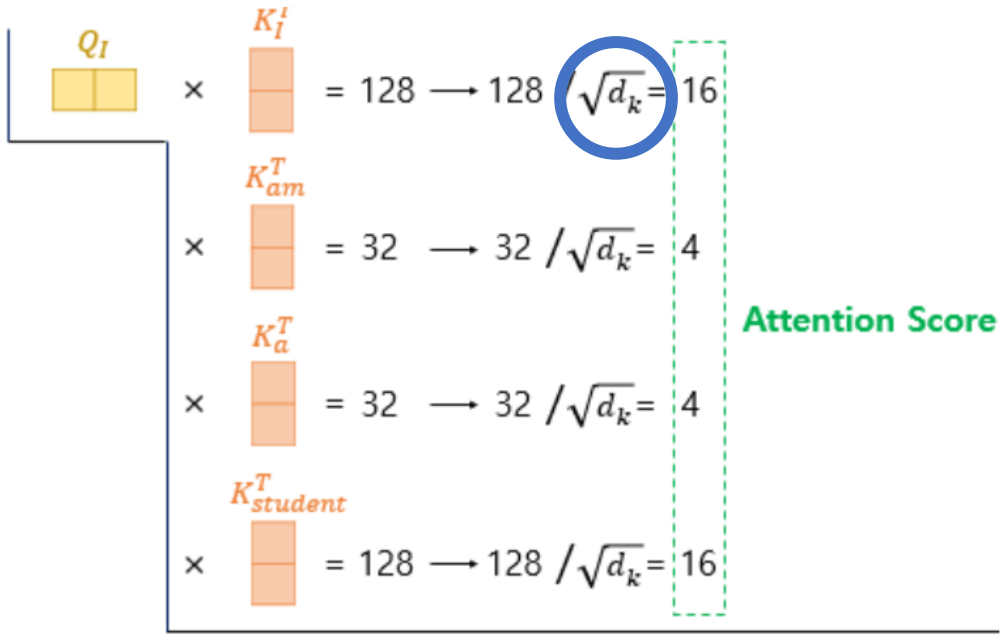
Multi-head Attention 순서

1) query 행렬과 문장 내 모든 key 행렬을 곱해 attention score를 구함.

2) 스코어 값을 softmax함수에 넣어 attention 분포를 구함.

3) 각 분포 값을 value 벡터와 곱한 뒤 다 더해 attention value를 구함.

ex) I am a student



Query 벡터와 Key 벡터들을 곱한 값을 d_k의 루트 값으로 나누어 준다.

Scaling Setting

: Attention Score를 바탕으로 Softmax 함수를 취하여 주는데, 이 때 값이 너무 커지는 것을 막기 위해 key의 차원의 루트 값으로 나누어 정규화를 진행함.

인코더 입력 → Multi-head Attention → Add+Norm → FFNN → Add+Norm

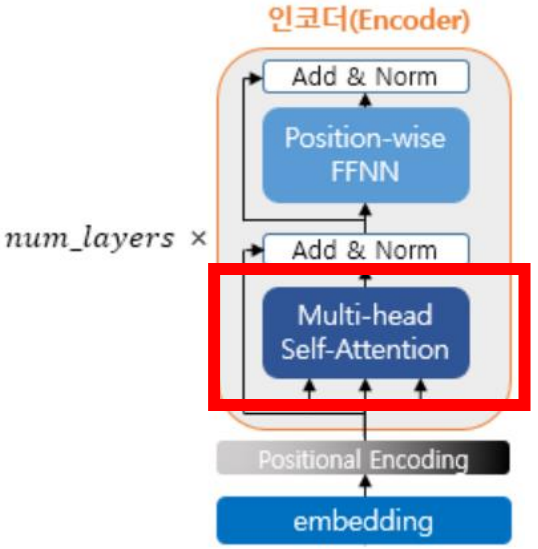
4-1) Encoder의 첫 layer, Multi-head Attention

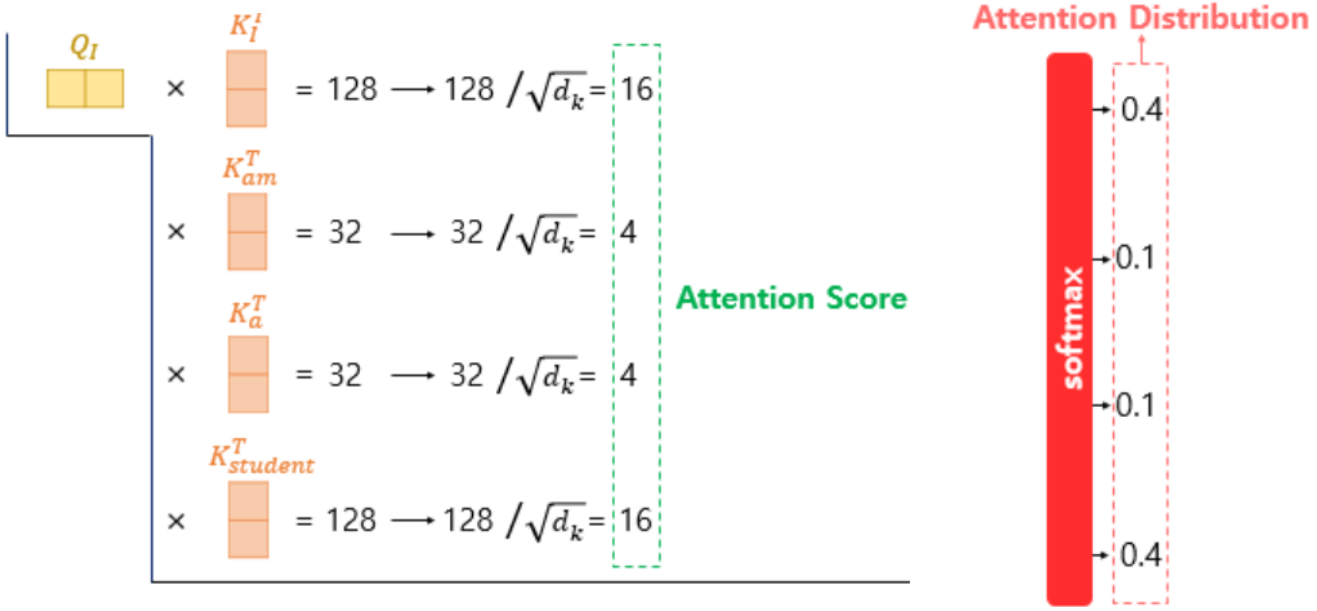
Multi-head Attention 순서

1) query 행렬과 문장 내 모든 key 행렬을 곱해 attention score를 구함.

2) 스코어 값을 softmax함수에 넣어 attention 분포를 구함.

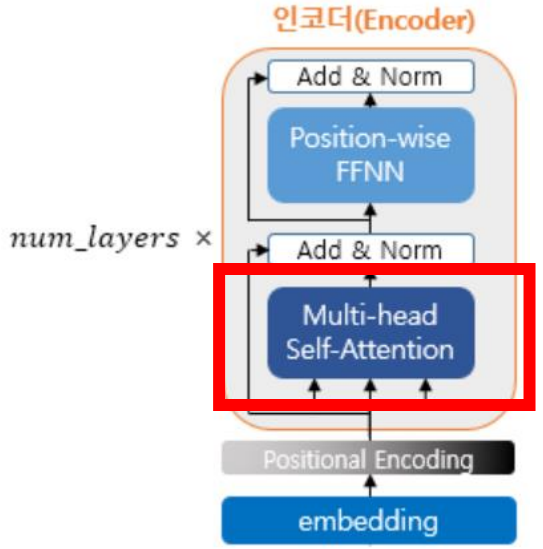
3) 각 분포 값을 value 벡터와 곱한 뒤 다 더해 attention value를 구함.





인코더 입력 → Multi-head Attention → Add+Norm → FFNN → Add+Norm

4-1) Encoder의 첫 layer, Multi-head Attention

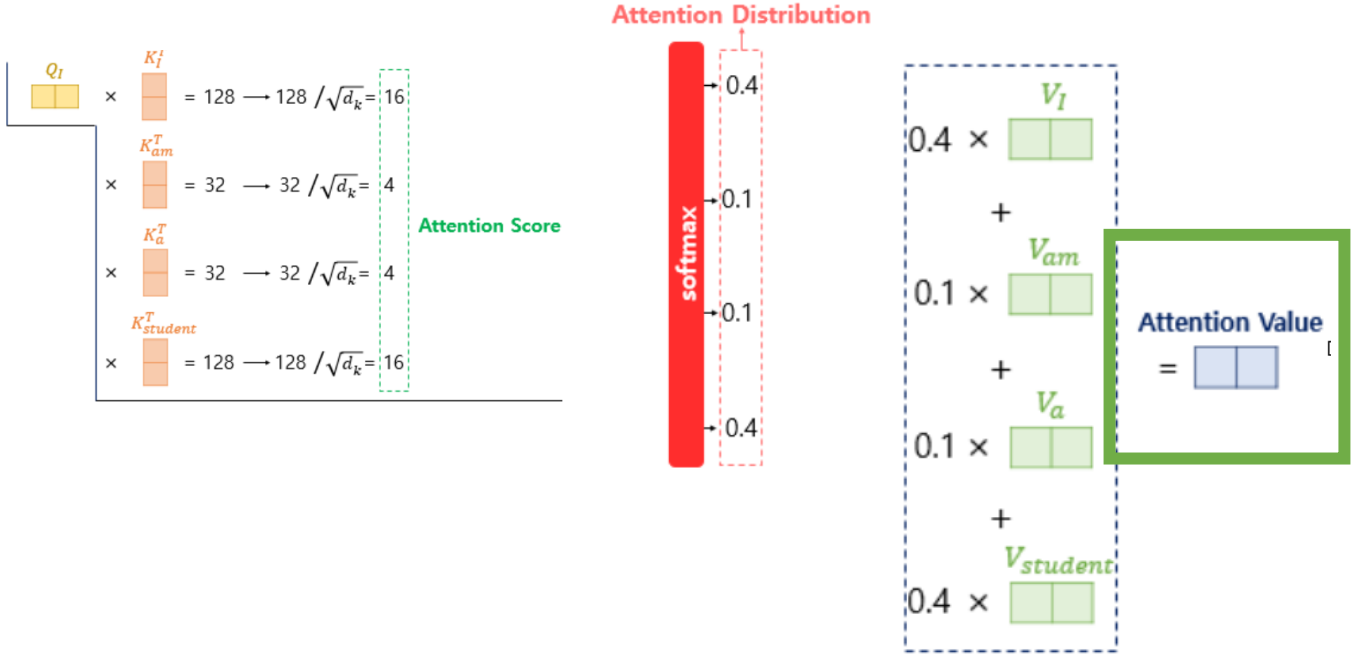


Multi-head Attention 순서

1) query 행렬과 문장 내 모든 key 행렬을 곱해 attention score를 구함.

2) 스코어 값을 softmax함수에 넣어 attention 분포를 구함.

3) 각 분포 값을 value 벡터와 곱한 뒤 다 더해 attention value를 구함.



왜 어텐션 분포를 그대로 사용하지 않는가?

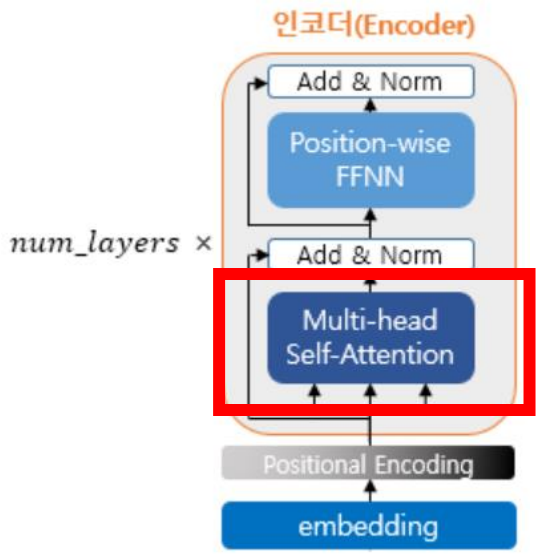
→ 집중하고 싶은 단어들만 남겨두기 위해 분포 값이 0.001처럼 작은 숫자들은 곱함으로써 없애 버릴 수 있기 때문이다.

이러한 attention value는 인코더의 문맥을 포함하고 있다고 하여 컨텍스트 벡터(Context vector)라고 부르기도 함.

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

인코더 입력 → Multi-head Attention → Add+Norm → FFNN → Add+Norm

4-1) Encoder의 첫 layer, Multi-head Attention



Multi-head Attention 순서

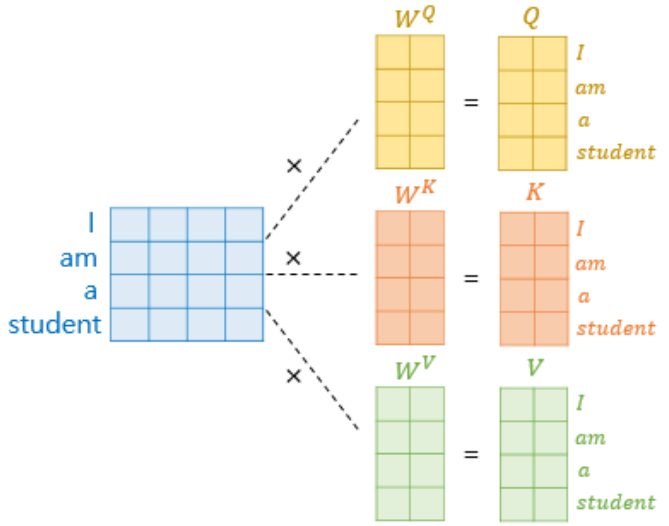
1) query 행렬과 문장 내 모든 key 행렬을 곱해 attention score를 구함.

2) 스코어 값을 softmax함수에 넣어 attention 분포를 구함.

3) 각 분포 값을 value 벡터와 곱한 뒤 다 더해 attention value를 구함.

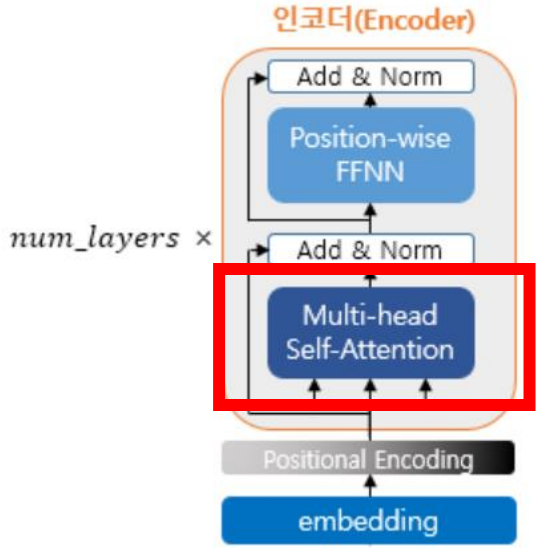
모든 단어 사이의 관련도를 알기 위해선 문장 내 모든 query 벡터를 구해서 연산을 해줘야 하지만!

일일이 하는 것 대신 행렬로 만들어주어 행렬 연산으로 한번에 구할 수 있다.



인코더 입력 → Multi-head Attention → Add+Norm → FFNN → Add+Norm

4-1) Encoder의 첫 layer, Multi-head Attention



위의 과정(연관성 찾는 과정)들이 **여러 개의 head(num_heads개)**에서 진행된다.
(Multi-head Attention인 이유)

Head를 여러 개 사용하는 이유?

: 여러 개의 head에서 병렬로 어텐션을 수행하여 다른 관점으로 정보(연관성)를 수집하기 위해

ex) The animal didn't cross the street, because **it** was too tired

이 문장에서 Query는 it, Key는 문자에 있는 11개의 단어들이다.

↓ Attention ?

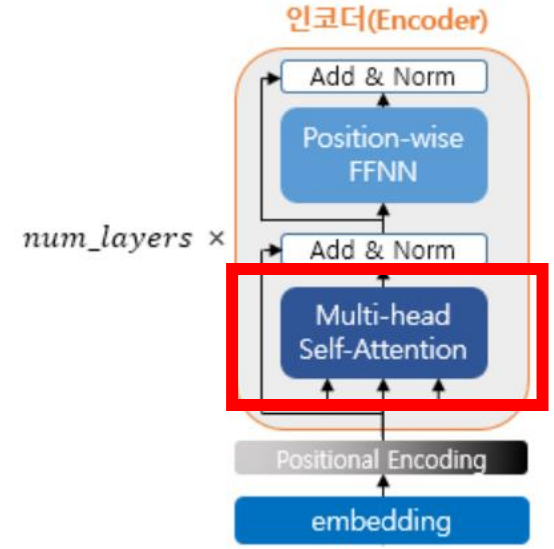
Of course! it 과의 연관성이 가장 크다고 나올 것

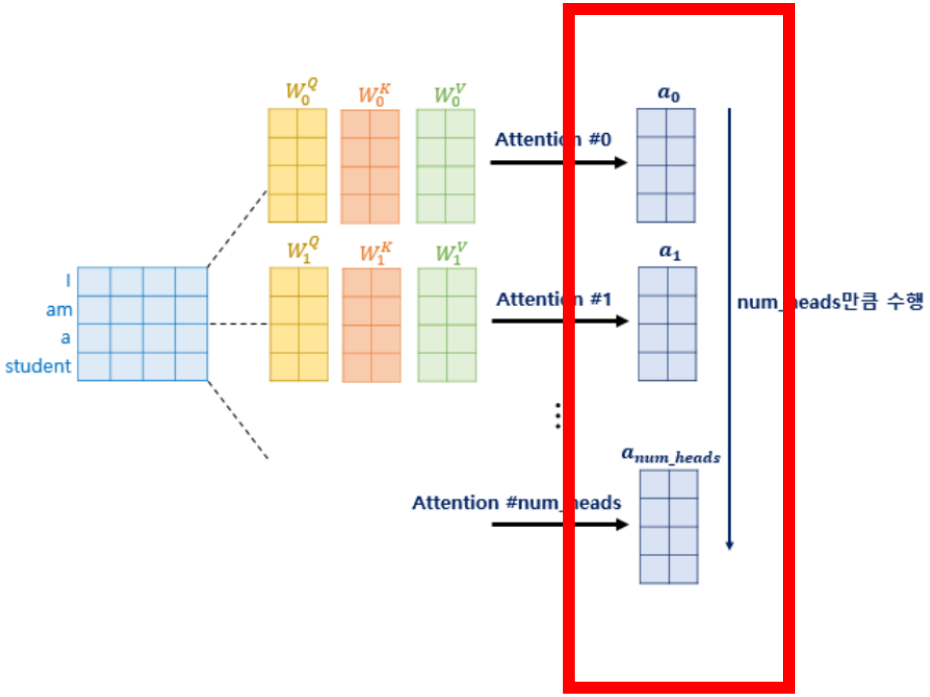
여러 개의 head를 이용하게 되면 다른 단어들(animal)과의 연관성도 파악할 수 있다.

인코더 입력 → Multi-head Attention → Add+Norm → FFNN → Add+Norm

4-1) Encoder의 첫 layer, Multi-head Attention

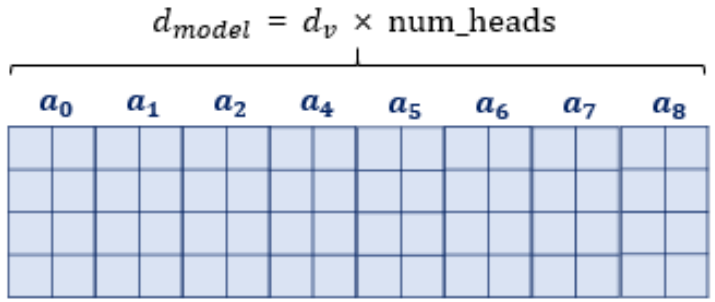
위의 과정(연관성 찾는 과정)들이 **여러 개의 head(num_heads개)**에서 진행된다.
(Multi-head Attention인 이유)





각각의 head를 거쳐 나온 행렬들

concatenate



인코더 입력 → Multi-head Attention → Add+Norm → FFNN → Add+Norm

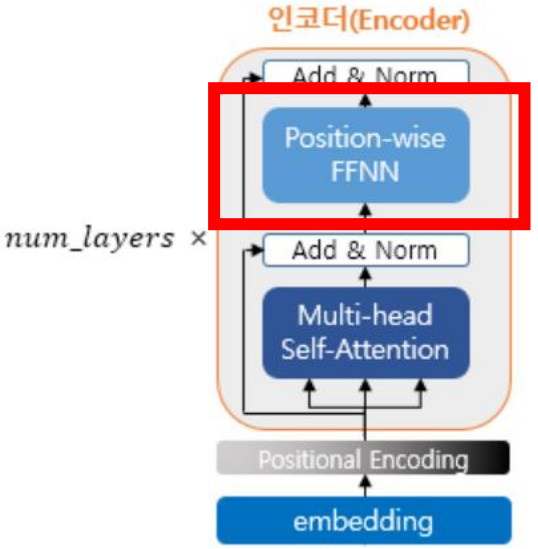
4-2) Encoder의 두 번째 layer, FFNN

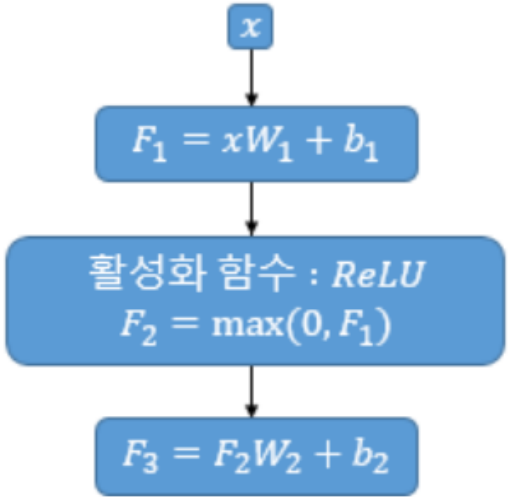
FFNN(fully connected Feed Forward Neural Network)

: 입력층에서 출력층 방향으로 연산이 전개되는 신경망

: FFNN은 인코더와 디코더에서 공통적으로 사용하고 있는 layer

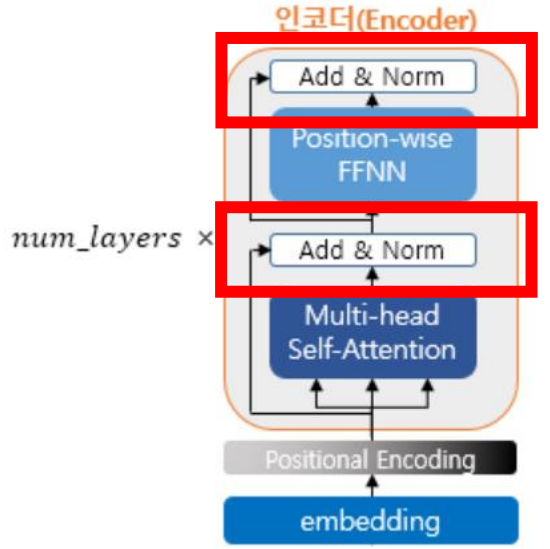
FFNN은 각 위치의 단어마다 독립적으로 적용되어 출력값을 만듦.





인코더 입력 → Multi-head Attention → Add+Norm → FFNN → Add+Norm

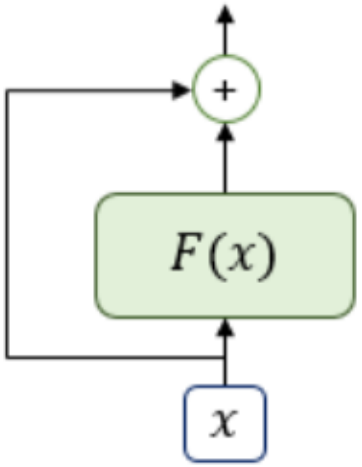
4-2) Encoder의 추가 기법, Add & Norm



잔차 연결(Residual Connection)

: 특정 layer를 건너 뛰어 복사된 값을 그대로 넣어주는 기법

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



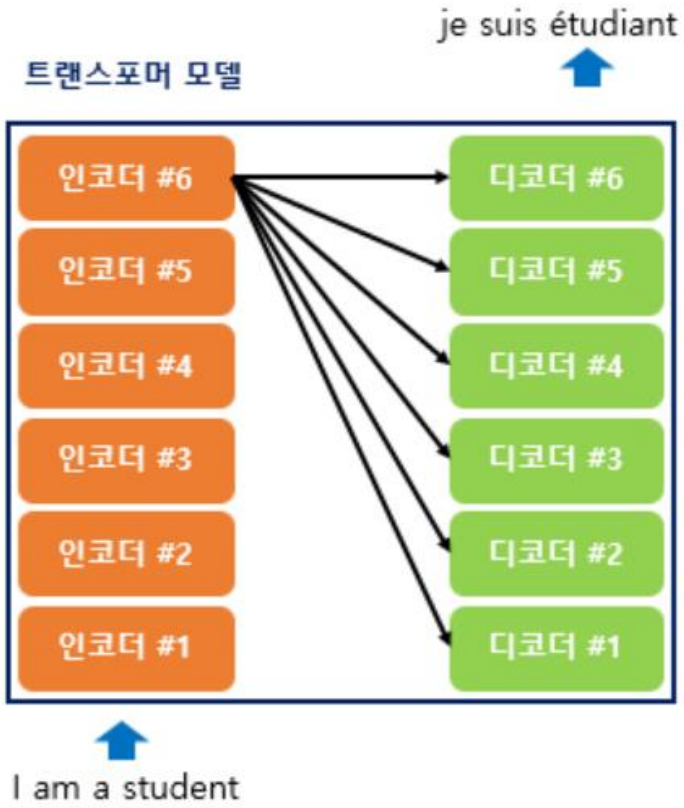
입력 받으면서 잔여된 부분만 학습하기 때문에 전반적인 학습 난이도가 낮아 초기 모델 수렴 속도 높아짐

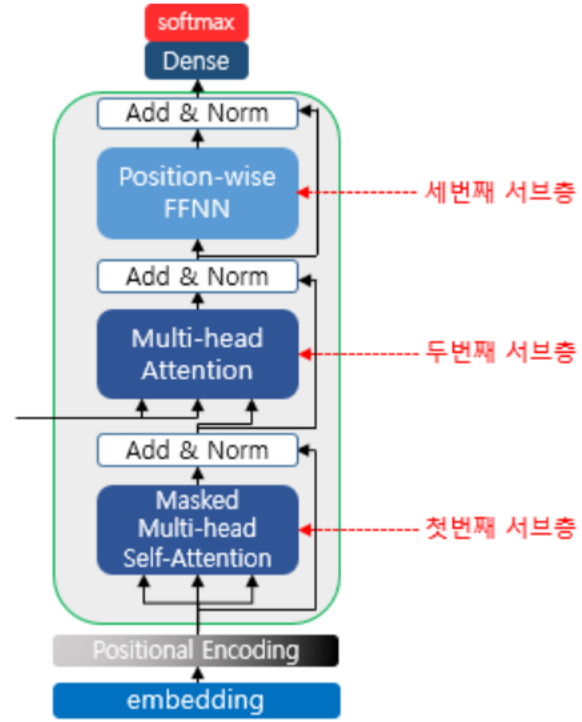
Normalization (층 정규화)

: 잔차 연결을 거쳐 층 정규화 과정을 거친다.

인코더 입력 → Multi-head Attention → Add+Norm → FFNN → Add+Norm

5. Transformer Decoding 단계





Encoder의 연산이 다 끝나고, 최종 Encoder의 출력을 각 디코더 층 연산에 사용한다.

디코더 입력→Masked 멀티헤드어텐션→Add+Norm→ 멀티헤드어텐션 →Add+Norm→FFNN→Add+Norm →Linear+Softmax

5-1) Decoder의 첫 layer, Masked Multi-head self attention

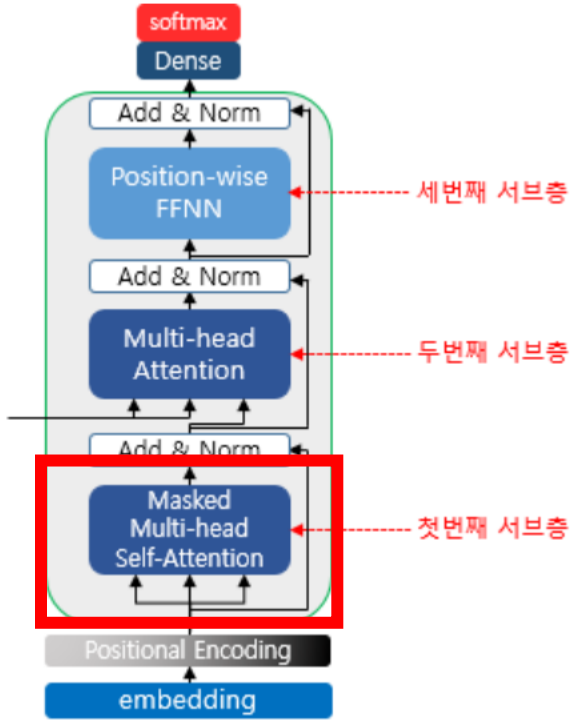
디코더에서도 embedding과 PE과정을 거쳐 문장이 입력됨.

Seq2Seq :

입력 단어를 매 시점마다 순차적으로 입력 받아 예측 시기 바로 이전 단어만 참고함

Transformer :

애초에 문장 전체를 입력 받기 때문에 미래 시점의 단어까지 참고하게 됨

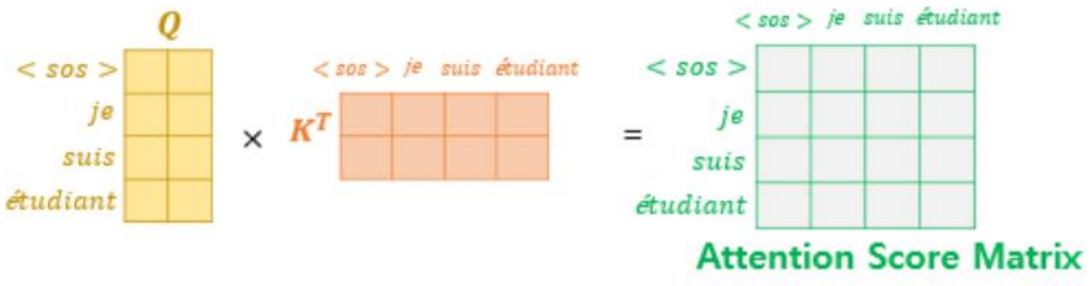


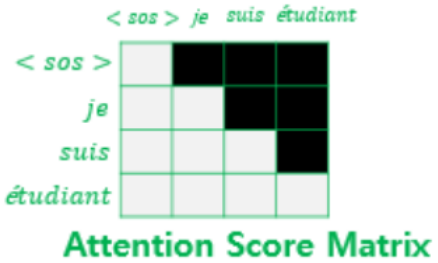
➔ **Look-Ahead Mask** 기법 도입

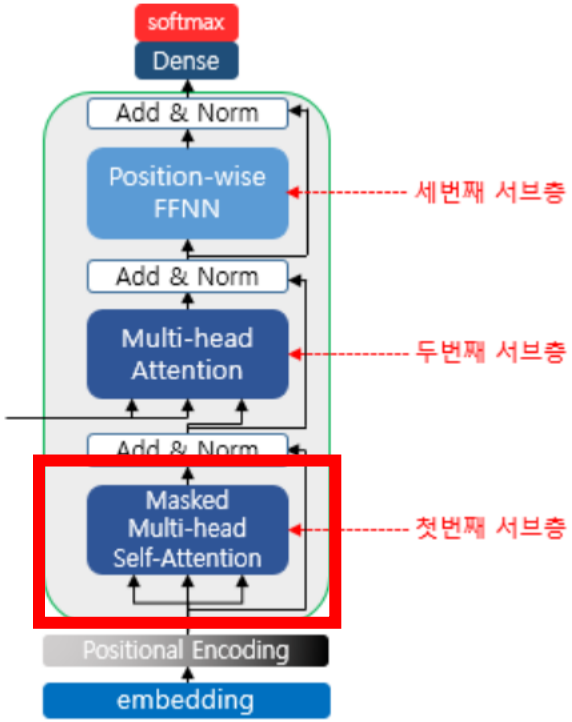
현재 예측 시점보다 미래의 단어를 참고하지 못한다.

디코더 입력→Masked 멀티헤드어텐션→Add+Norm→ 멀티헤드어텐션 →Add+Norm→FFNN→Add+Norm →Linear+Softmax

5-1) Decoder의 첫 layer, Masked Multi-head self attention







Multi-head Attention 순서

1) query 행렬과 문장 내 모든 key 행렬을 곱해 attention score를 구함.

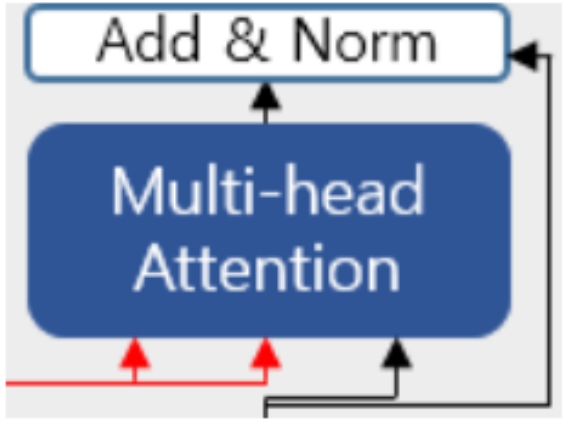
2) 스코어 값을 softmax함수에 넣어 attention 분포를 구함.

3) 각 분포 값을 value 벡터와 곱하여 attention value를 구함.

t 시점 기준 미래의 단어들의 attention score에는 매우 작은 값을 곱해주어 softmax함수를 취할 때 0이 되도록 만들어준다.

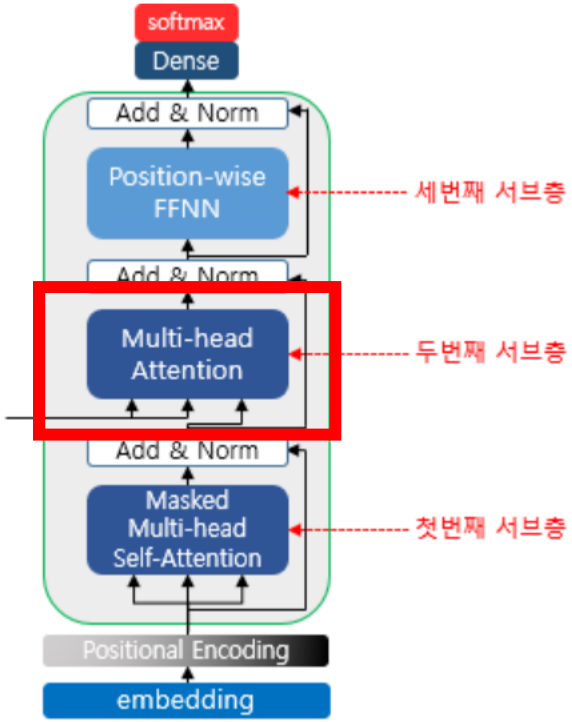
디코더 입력→Masked 멀티헤드어텐션→Add+Norm→ 멀티헤드어텐션 →Add+Norm→FFNN→Add+Norm →Linear+Softmax

5-2) Decoder의 두 번째 layer, 인코더-디코더 어텐션

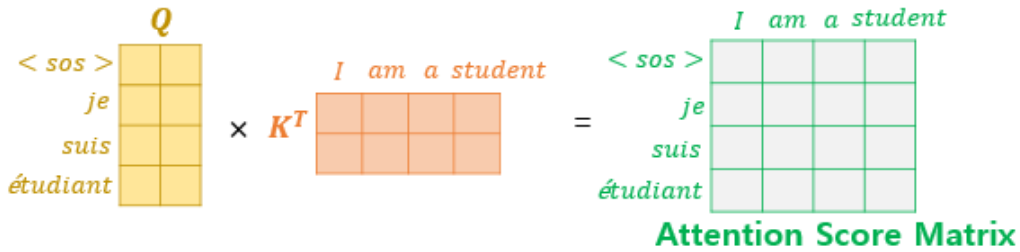


Attention을 위한 **세 가지** 입력 요소

- 인코더 마지막 층의 Key행렬
- 인코더 마지막 층의 Value행렬
- Masked Multi-head self attention의 출력값 (Query로 사용됨)



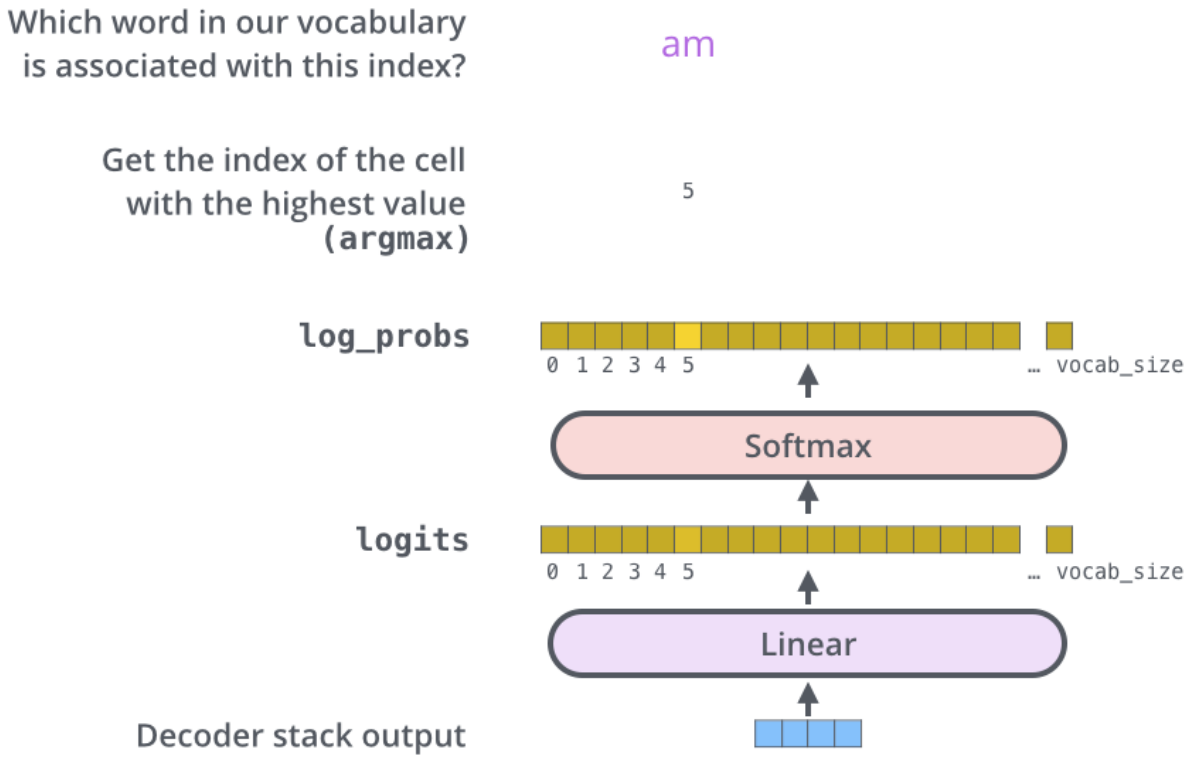
두 개의 화살표는 각각 Key와 Value를 의미하며, 이는 인코더의 마지막 층에서 온 행렬로부터 얻습니다. 반면, Query는 디코더의 첫번째 서브층의 결과 행렬로부터 얻는다는 점이 다릅니다. Query가 디코더 행렬, Key가 인코더 행렬일 때, 어텐션 스코어 행렬을 구하는 과정은 다음과 같습니다.

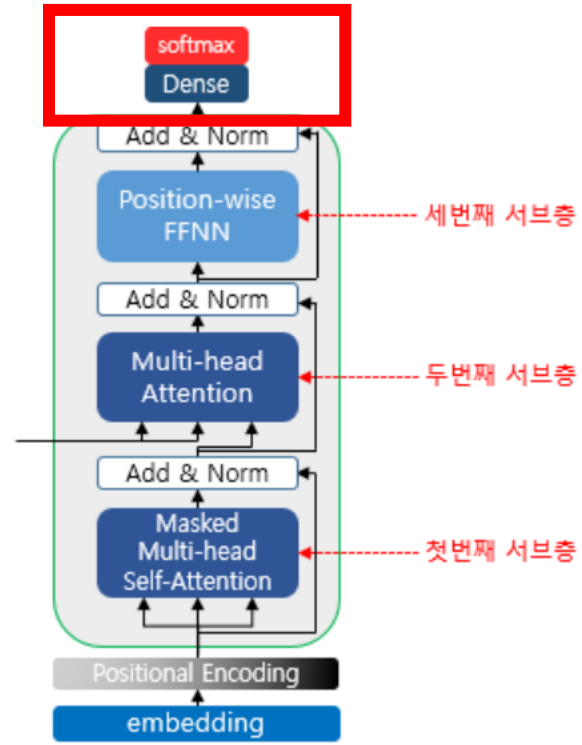


그 외에 멀티 헤드 어텐션을 수행하는 과정은 다른 어텐션들과 같습니다.

디코더 입력→Masked 멀티헤드어텐션→Add+Norm→ 멀티헤드어텐션 →Add+Norm→FFNN→Add+Norm →Linear+Softmax

5-2) Decoder의 마지막 layer, Linear + Softmax Layer





디코더 입력→Masked 멀티헤드어텐션→Add+Norm→ 멀티헤드어텐션 →Add+Norm→FFNN→Add+Norm →Linear+Softmax