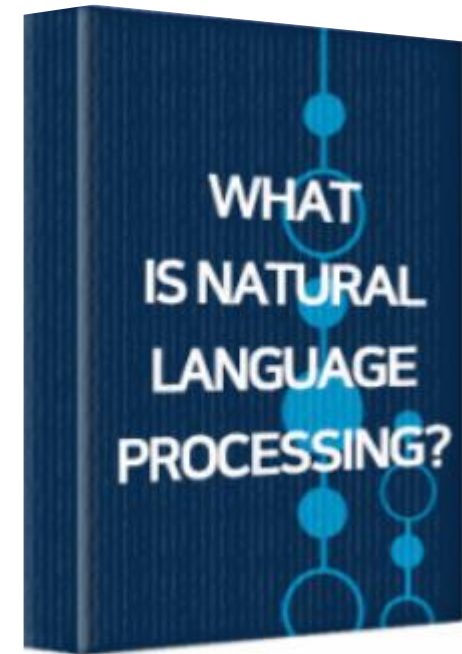


# Tensorflow를 활용한 딥러닝 자연어 처리 입문. 8강

# 앞으로 배우게 될 내용

- Text preprocessing for NLP & Language Model
- Basic Tensorflow & Vectorization
- Word Embedding (Word2Vec, FastText, GloVe)
- Text Classification (using RNN & CNN)
- Chatbot with Deep Learning
- Sequence to Sequence
- Attention Mechanism
- **Transformer & BERT**

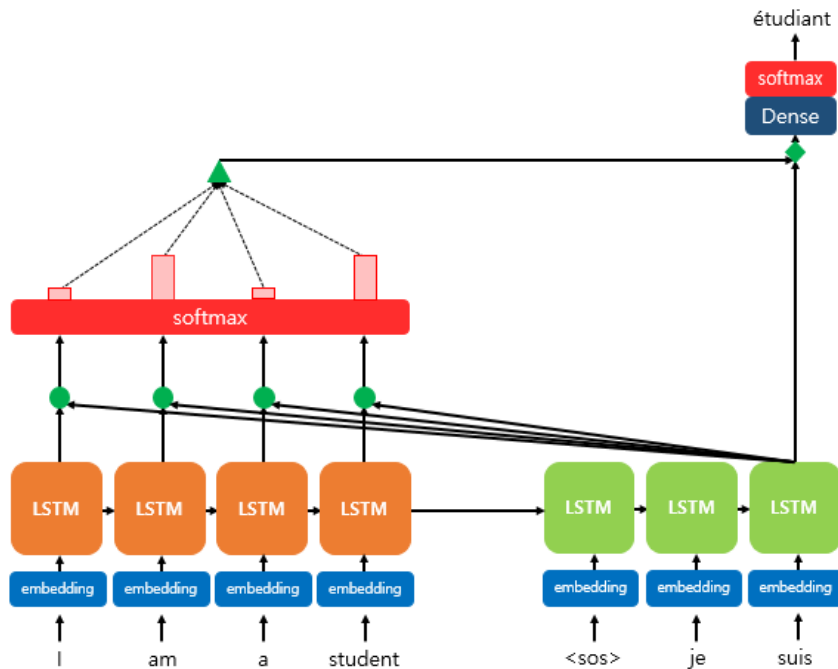


참고 자료 : <https://wikidocs.net/book/2155>

# Transformer

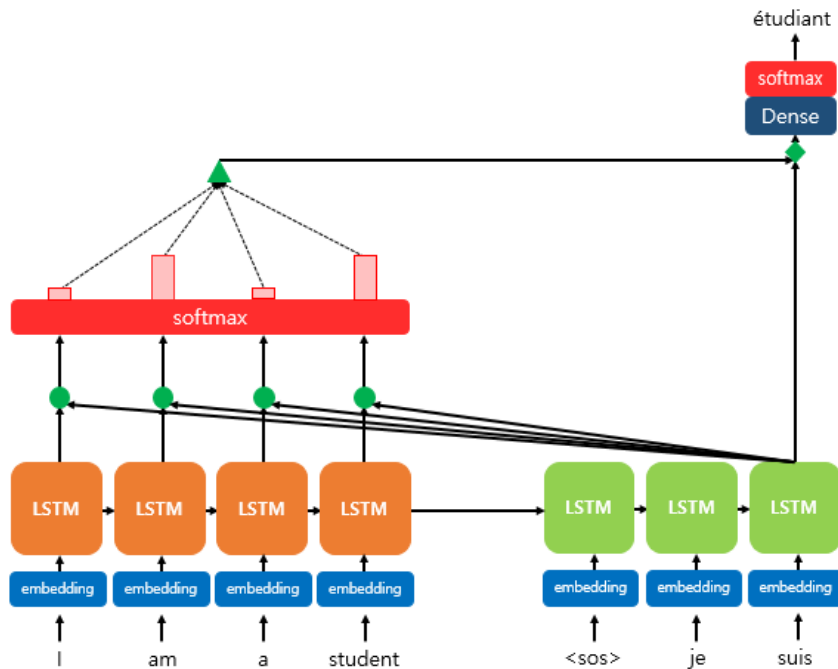
# Attention Mechanism

- **RNN** 계열의 신경망의 순차적 연산은 병렬 연산을 할 수 없도록 한다.
- **LSTM, GRU**을 사용한다고 하더라도, 긴 문장에 대해서는 성능이 저하되는 현상 발생.
- 어텐션 메커니즘은 **RNN 계열 seq2seq** 구조에 도입되어 기계 번역의 성능을 상당 부분 개선.



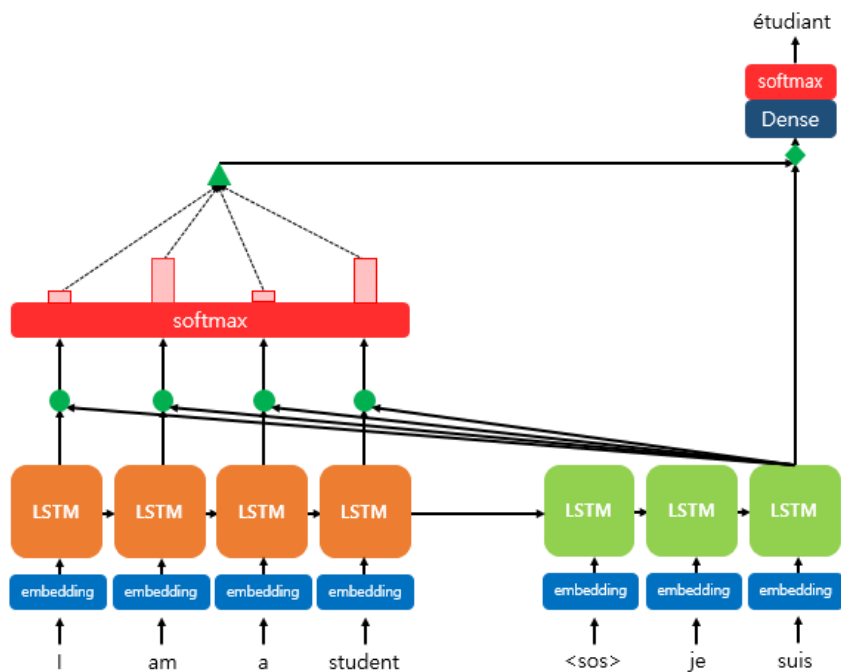
# Attention Mechanism

- **RNN** 계열의 신경망의 순차적 연산은 병렬 연산을 할 수 없도록 한다.
- **LSTM, GRU**을 사용한다고 하더라도, 긴 문장에 대해서는 성능이 저하되는 현상 발생.
- 어텐션 메커니즘은 **RNN 계열 seq2seq** 구조에 도입되어 기계 번역의 성능을 상당 부분 개선.
- 그런데 어텐션으로 모든 state에 접근할 수 있다면 굳이 RNN이 필요할까?



# Attention Mechanism

- **RNN** 계열의 신경망의 순차적 연산은 병렬 연산을 할 수 없도록 한다.
- **LSTM, GRU**을 사용한다고 하더라도, 긴 문장에 대해서는 성능이 저하되는 현상 발생.
- 어텐션 메커니즘은 **RNN 계열 seq2seq** 구조에 도입되어 기계 번역의 성능을 상당 부분 개선.
- 그런데 어텐션으로 모든 state에 접근할 수 있다면 굳이 RNN이 필요할까?



우리에게 필요한 것은  
오직 어텐션뿐!

# Attention is All you need

- 기계 번역을 위해 탄생
- 인코더-디코더 구조를 여전히 유지.

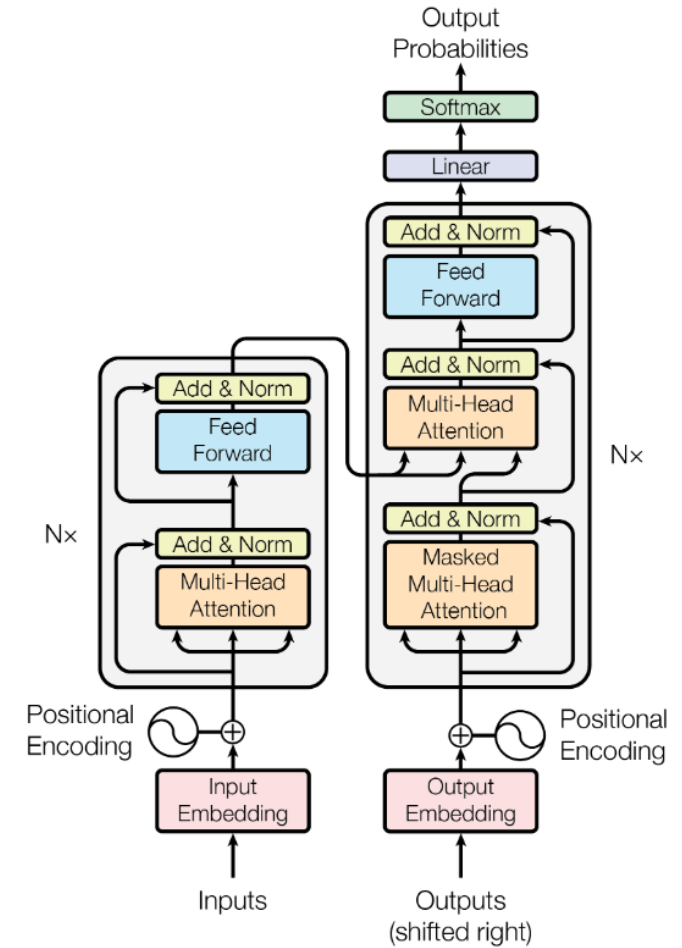


Figure 1: The Transformer - model architecture.

# Attention is All you need

- 기계 번역을 위해 탄생
- 인코더-디코더 구조를 여전히 유지.
- RNN의 경우, 각 step 별로 연산하므로 병렬 연산 불가.
- 많은 어텐션 메커니즘이 RNN과 함께 사용됨.

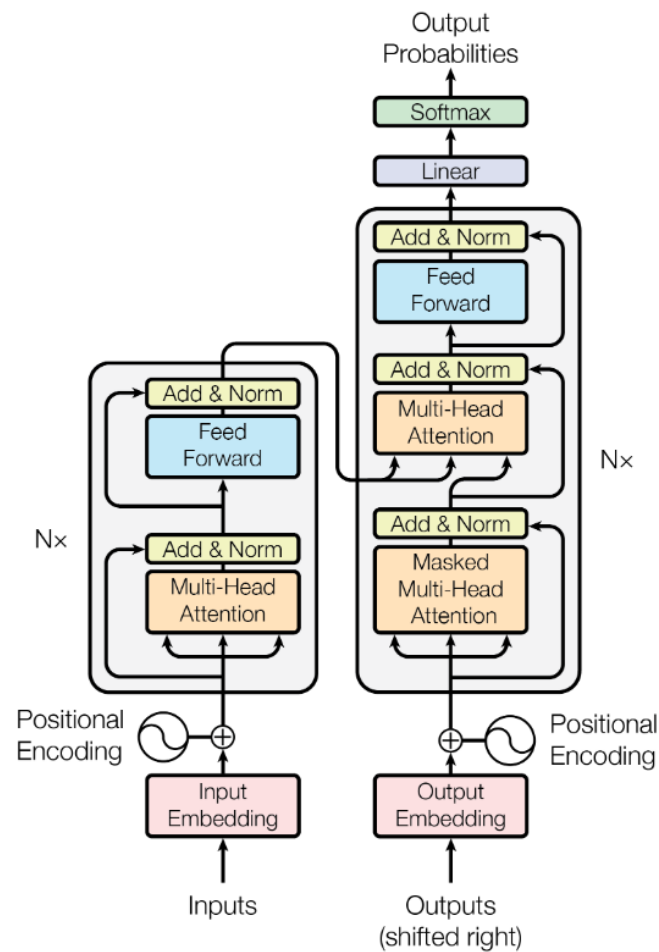


Figure 1: The Transformer - model architecture.



# Attention is All you need

- 기계 번역을 위해 탄생
- 인코더-디코더 구조를 여전히 유지.
- RNN의 경우, 각 step 별로 연산하므로 병렬 연산 불가.
- 많은 어텐션 메커니즘이 RNN과 함께 사용됨.
- 이 모델은 병렬 연산을 추구하며 RNN을 사용 X

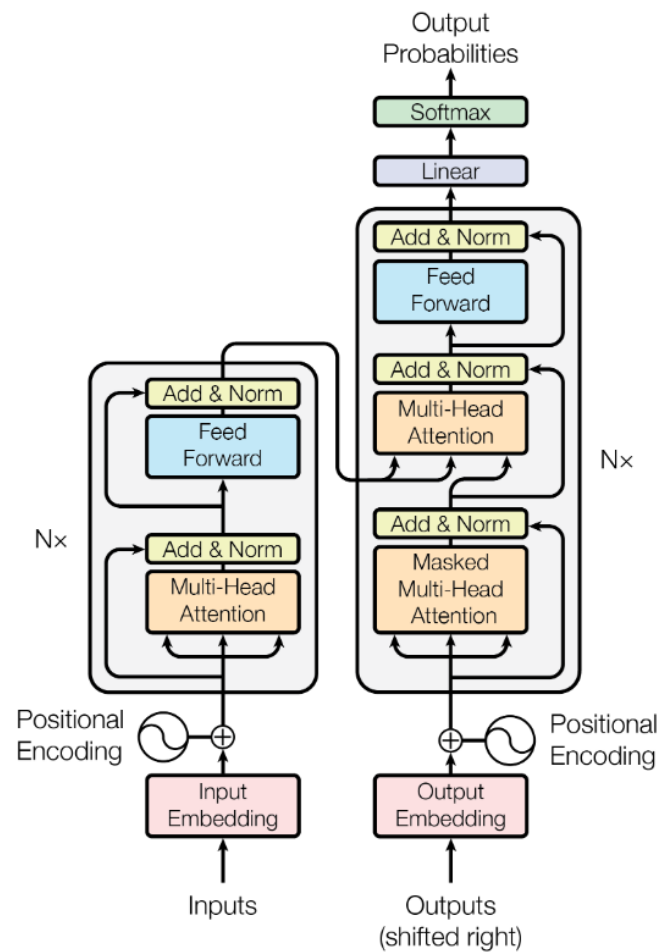


Figure 1: The Transformer - model architecture.

# Attention is All you need

- 기계 번역을 위해 탄생
- 인코더-디코더 구조를 여전히 유지.
- RNN의 경우, 각 step 별로 연산하므로 병렬 연산 불가.
- 많은 어텐션 메커니즘이 RNN과 함께 사용됨.
- 이 모델은 병렬 연산을 추구하며 RNN을 사용 X

모델 이름은 'Transformer'

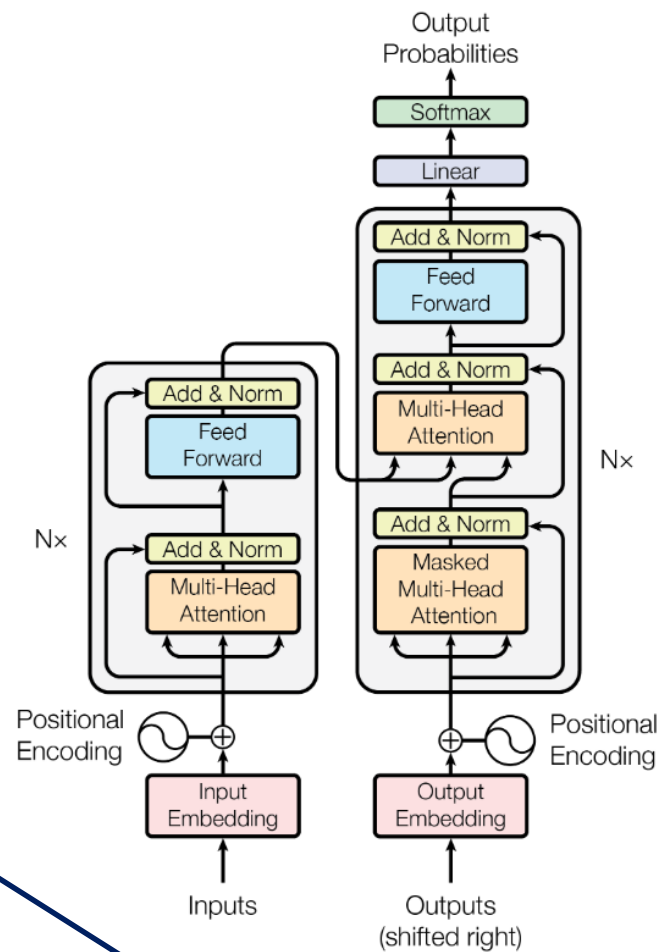


Figure 1: The Transformer model architecture.

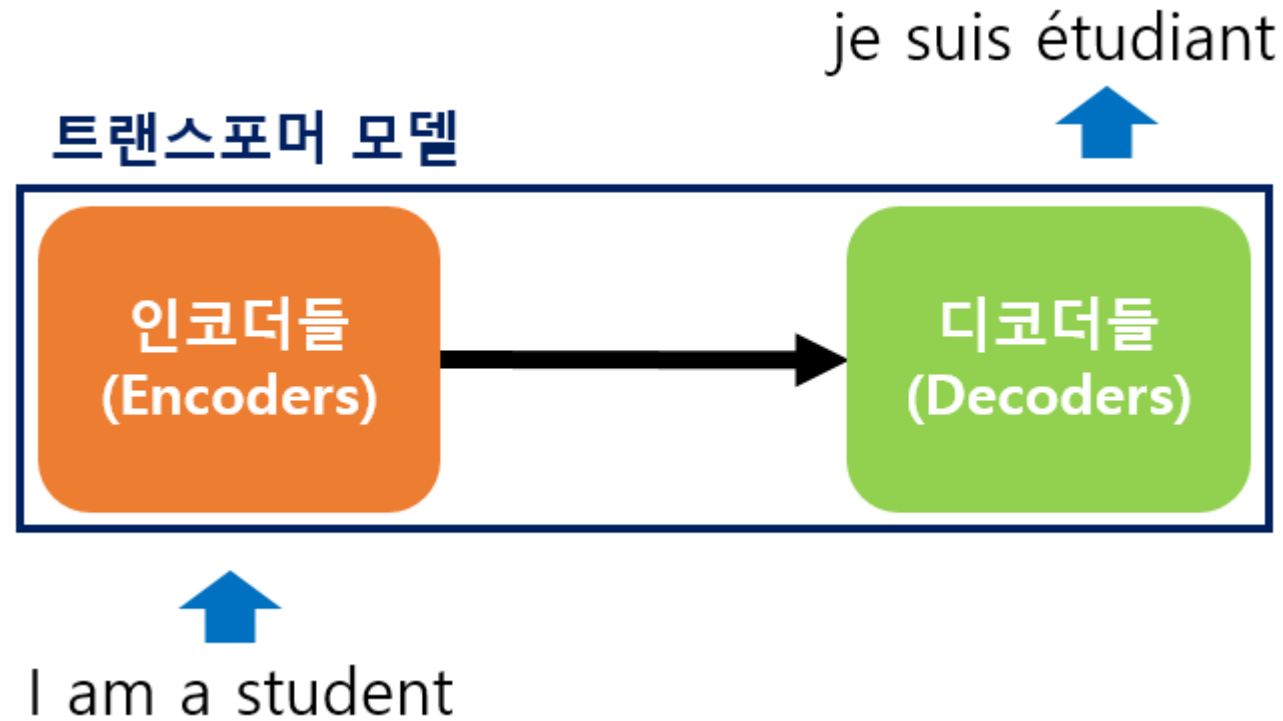
# Transformer

- 트랜스포머는 기본적으로 기계 번역을 위해 제안된 모델.
- 번역하고자 하는 문장을 입력하면, 번역 문장이 출력.



# Transformer

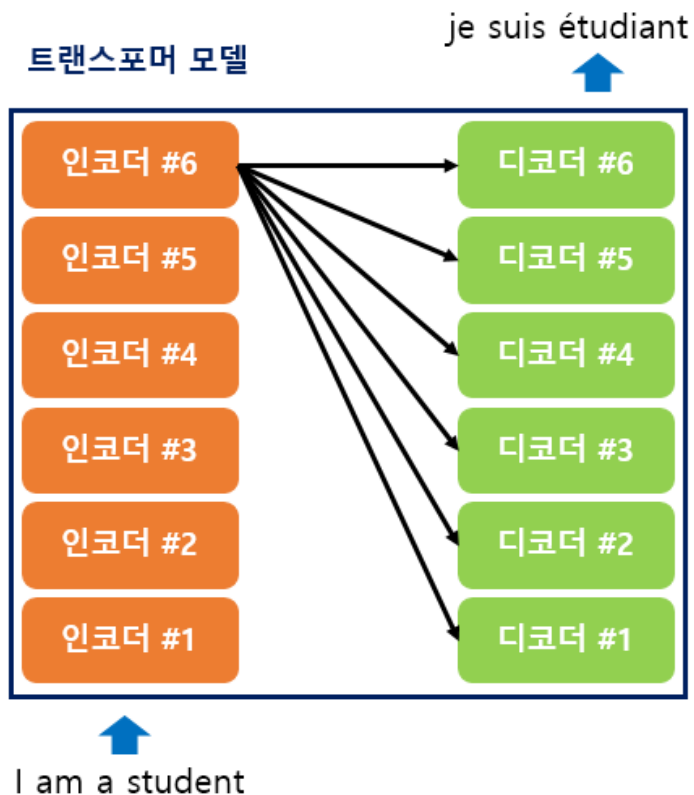
- 트랜스포머는 기본적으로 기계 번역을 위해 제안된 모델.
- 번역하고자 하는 문장을 입력하면, 번역 문장이 출력.
- 인코더-디코더 구조.



# Transformer

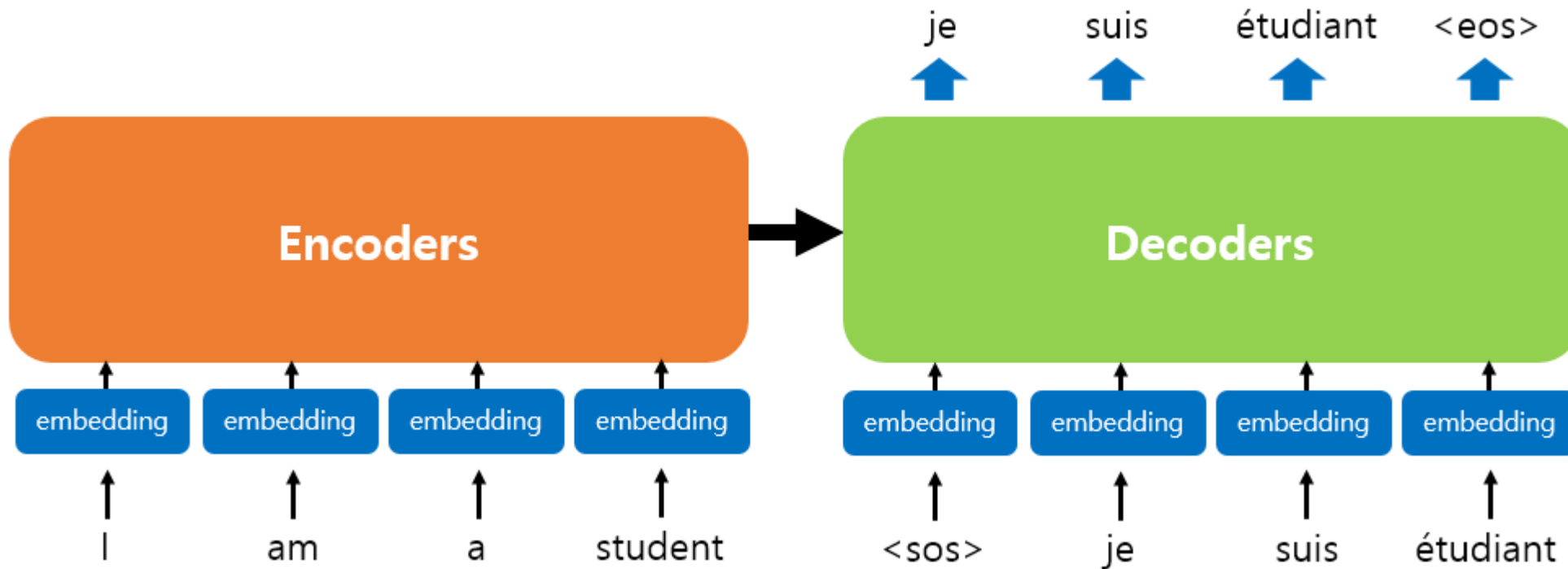
- 트랜스포머는 기본적으로 기계 번역을 위해 제안된 모델.
- 번역하고자 하는 문장을 입력하면, 번역 문장이 출력.
- 인코더-디코더 구조.

- 인코더와 디코더 블록(layer)이 N개 존재.
- 논문의 경우 각 6개.



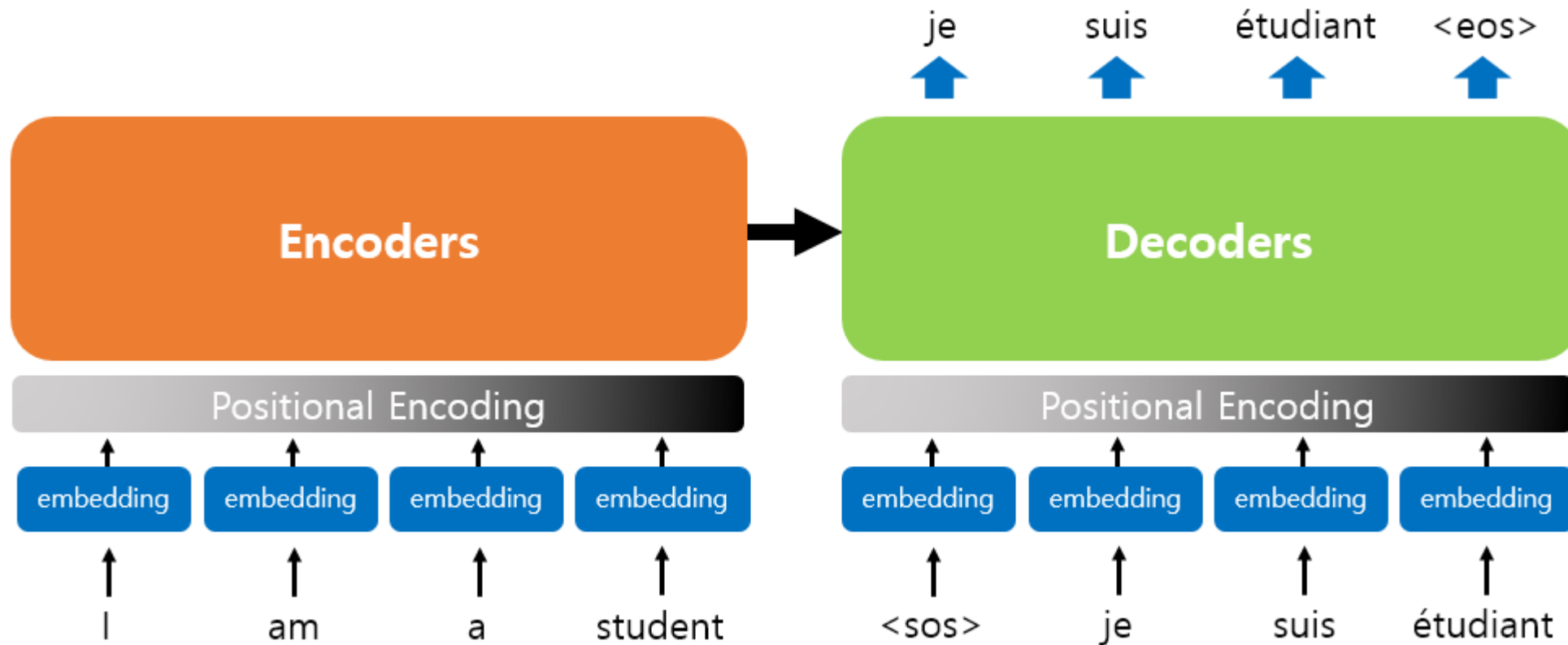
# Transformer

- 트랜스포머도 다른 딥 러닝 모델과 마찬가지로 Embedding layer를 사용
- Embedding layer를 통해서 얻은 임베딩 벡터를 인코더와 디코더의 입력으로 한다.



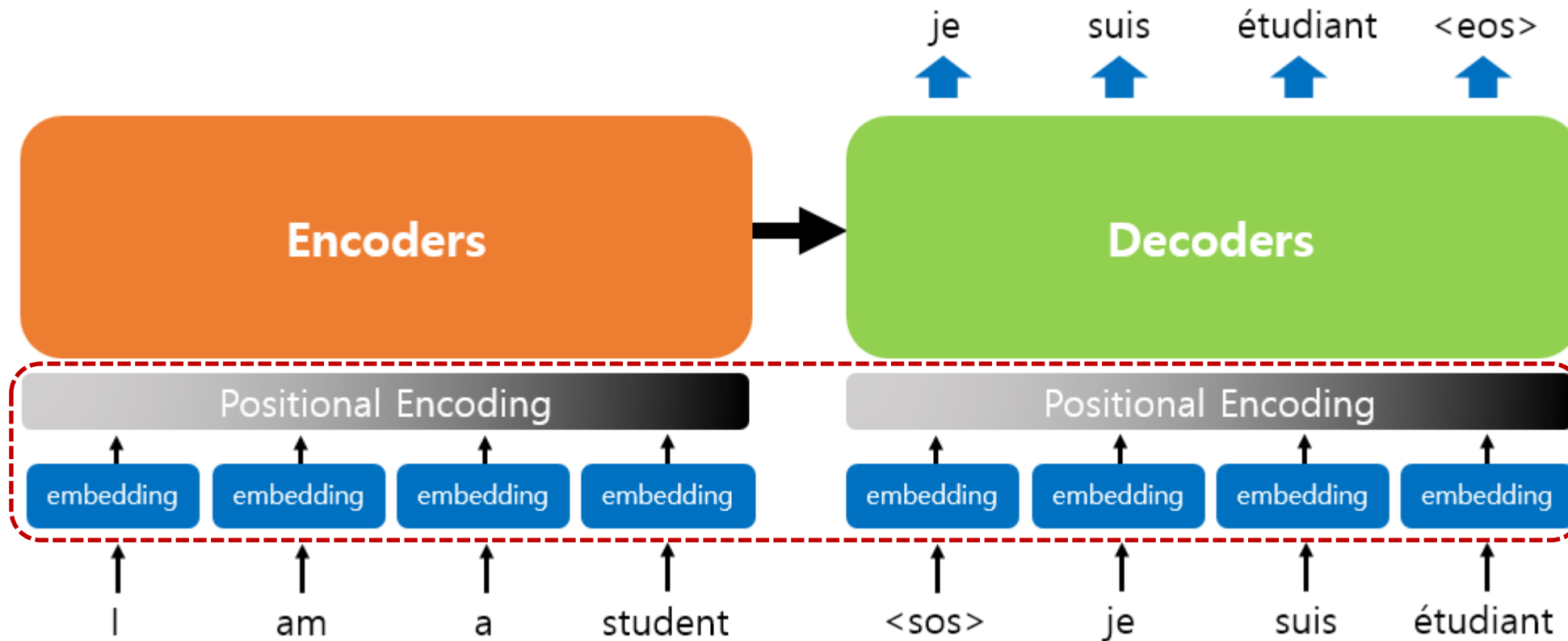
# Transformer : Positional Encoding

- 트랜스포머도 다른 딥 러닝 모델과 마찬가지로 Embedding layer를 사용
- Embedding layer를 통해서 얻은 임베딩 벡터를 인코더와 디코더의 입력으로 한다.
- 임베딩 벡터에 **Postional Encoding**이라는 과정을 거친 후에 입력으로 한다.



# Transformer : Positional Encoding

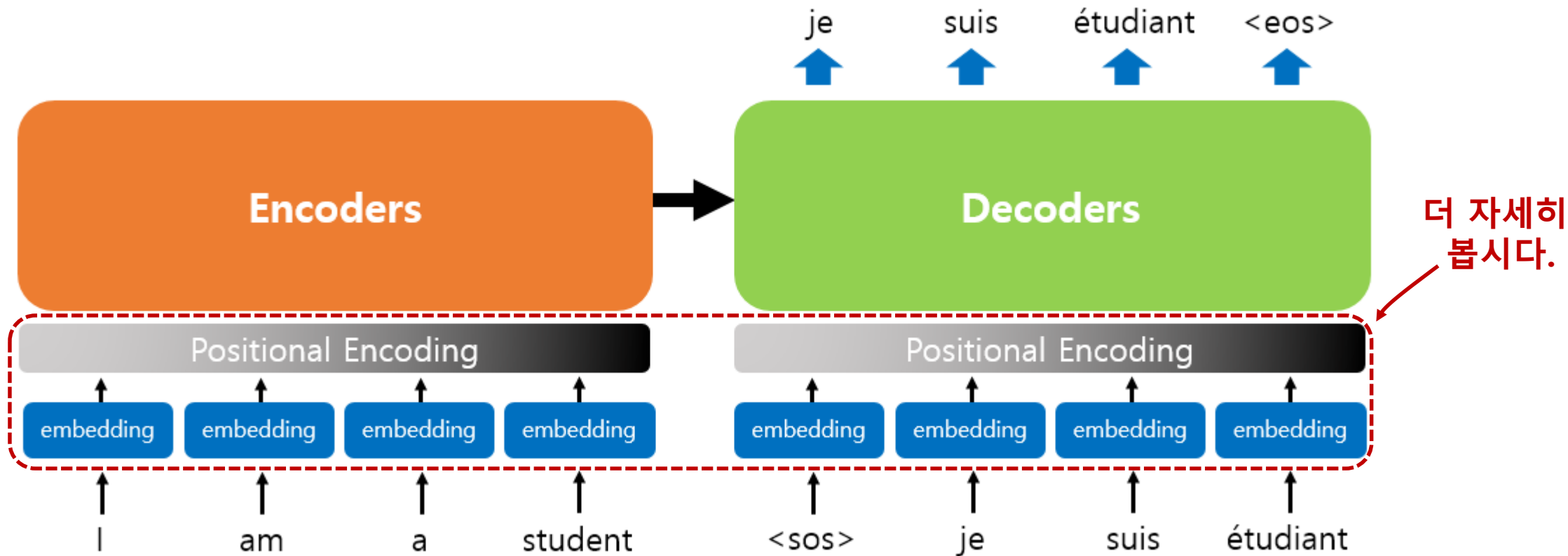
- 트랜스포머도 다른 딥 러닝 모델과 마찬가지로 Embedding layer를 사용
- Embedding layer를 통해서 얻은 임베딩 벡터를 인코더와 디코더의 입력으로 한다.
- 임베딩 벡터에 **Postional Encoding**이라는 과정을 거친 후에 입력으로 한다.





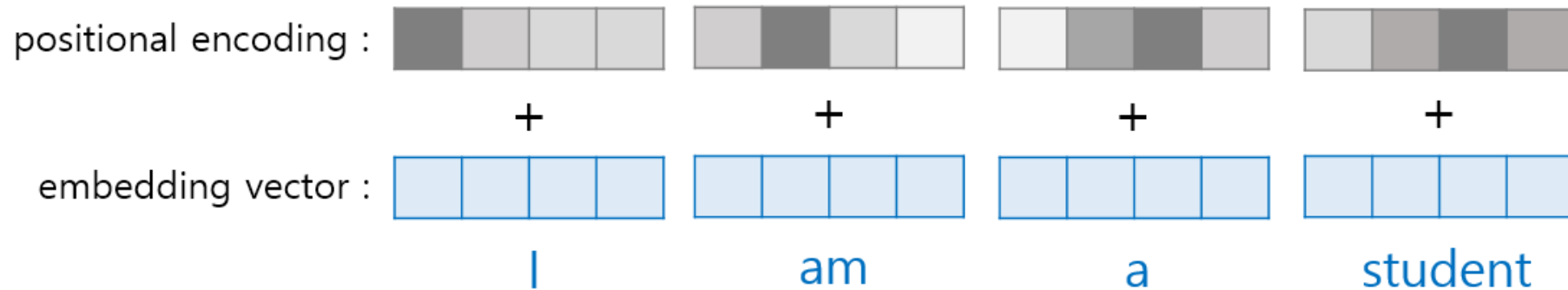
# Transformer : Positional Encoding

- 트랜스포머도 다른 딥 러닝 모델과 마찬가지로 Embedding layer를 사용
- Embedding layer를 통해서 얻은 임베딩 벡터를 인코더와 디코더의 입력으로 한다.
- 임베딩 벡터에 **Postional Encoding**이라는 과정을 거친 후에 입력으로 한다.



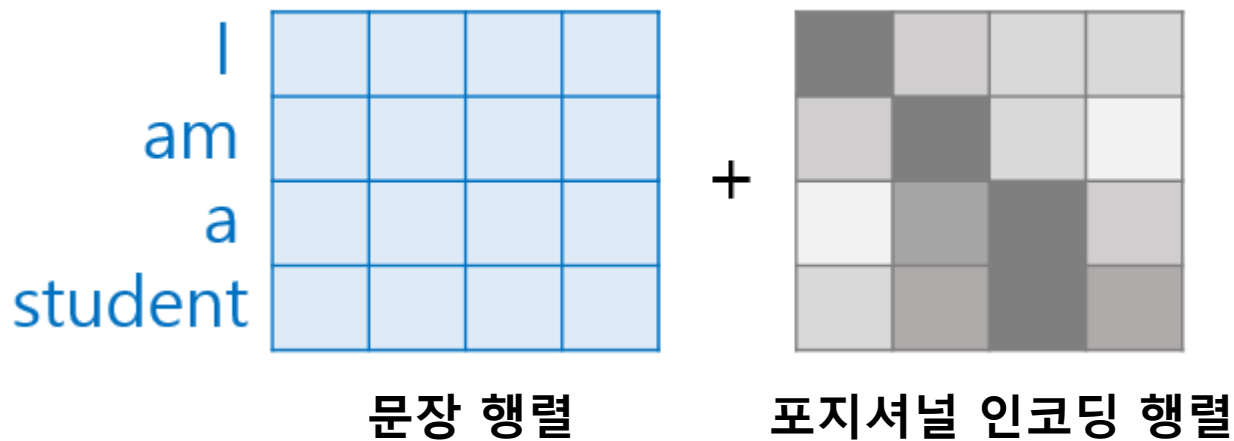
# Transformer : Positional Encoding

- 트랜스포머도 다른 딥 러닝 모델과 마찬가지로 Embedding layer를 사용
- Embedding layer를 통해서 얻은 임베딩 벡터를 인코더와 디코더의 입력으로 한다.
- 임베딩 벡터에 **Postional Encoding**이라는 과정을 거친 후에 입력으로 한다.



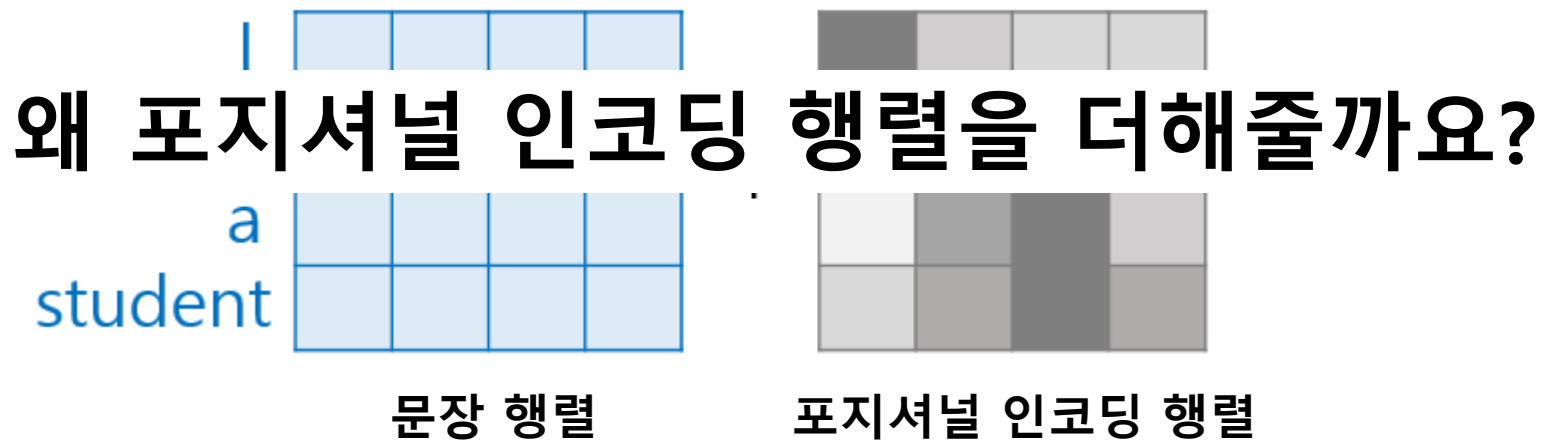
# Transformer : Positional Encoding

- 트랜스포머도 다른 딥 러닝 모델과 마찬가지로 Embedding layer를 사용
- Embedding layer를 통해서 얻은 임베딩 벡터를 인코더와 디코더의 입력으로 한다.
- 임베딩 벡터에 **Postional Encoding**이라는 과정을 거친 후에 입력으로 한다.
- 이를 행렬 연산으로 이해해본다면 다음과 같이 이해할 수 있다.



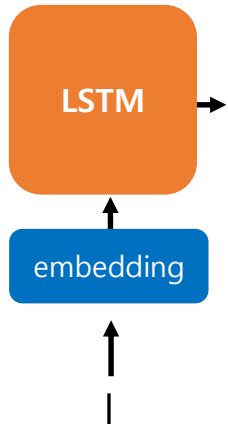
# Transformer : Positional Encoding

- 트랜스포머도 다른 딥 러닝 모델과 마찬가지로 Embedding layer를 사용
- Embedding layer를 통해서 얻은 임베딩 벡터를 인코더와 디코더의 입력으로 한다.
- 임베딩 벡터에 **Positional Encoding**이라는 과정을 거친 후에 입력으로 한다.
- 이를 행렬 연산으로 이해해본다면 다음과 같이 이해할 수 있다.



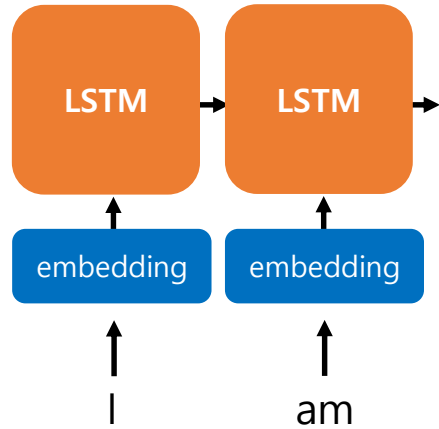
# Transformer : Positional Encoding

- RNN이 자연어 처리에서 유용했었던 이유는 단어 입력을 순차적으로 받으므로 각 단어의 위치 정보(position information)을 가질 수 있었기 때문.



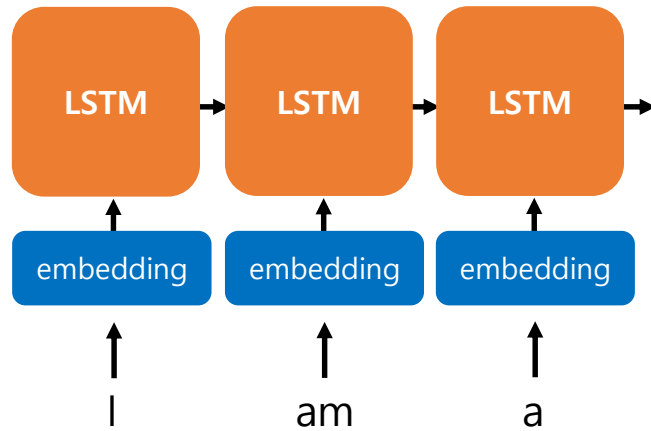
# Transformer : Positional Encoding

- RNN이 자연어 처리에서 유용했었던 이유는 단어 입력을 순차적으로 받으므로 각 단어의 위치 정보(position information)을 가질 수 있었기 때문.



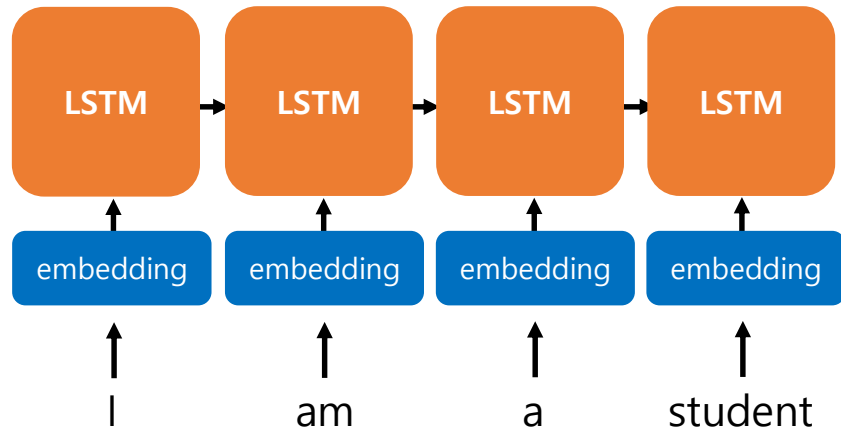
# Transformer : Positional Encoding

- RNN이 자연어 처리에서 유용했었던 이유는 단어 입력을 순차적으로 받으므로 각 단어의 위치 정보(position information)을 가질 수 있었기 때문.



# Transformer : Positional Encoding

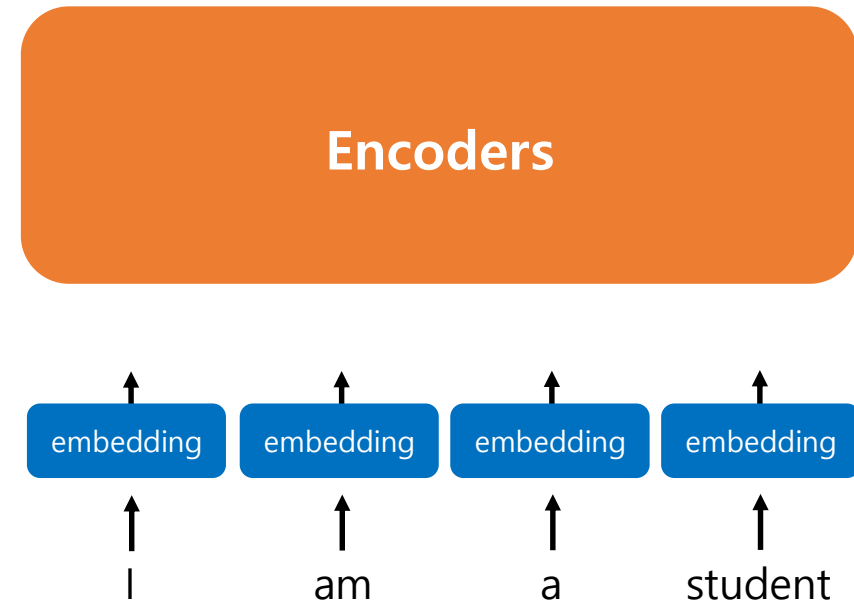
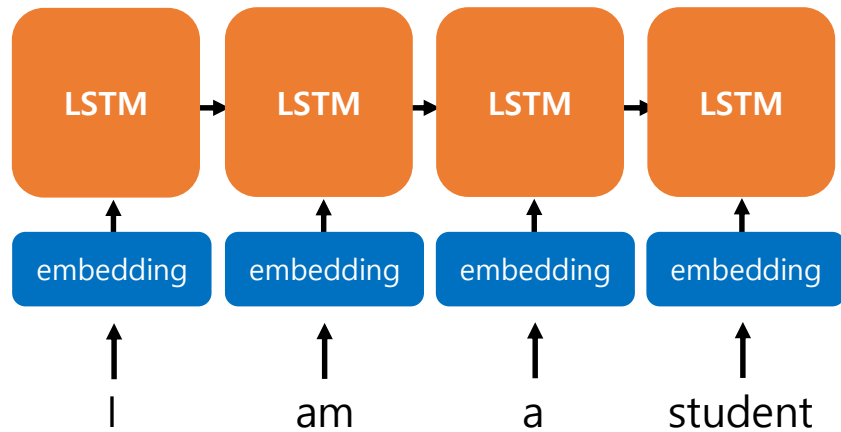
- RNN이 자연어 처리에서 유용했었던 이유는 단어 입력을 순차적으로 받으므로 각 단어의 위치 정보(position information)을 가질 수 있었기 때문.





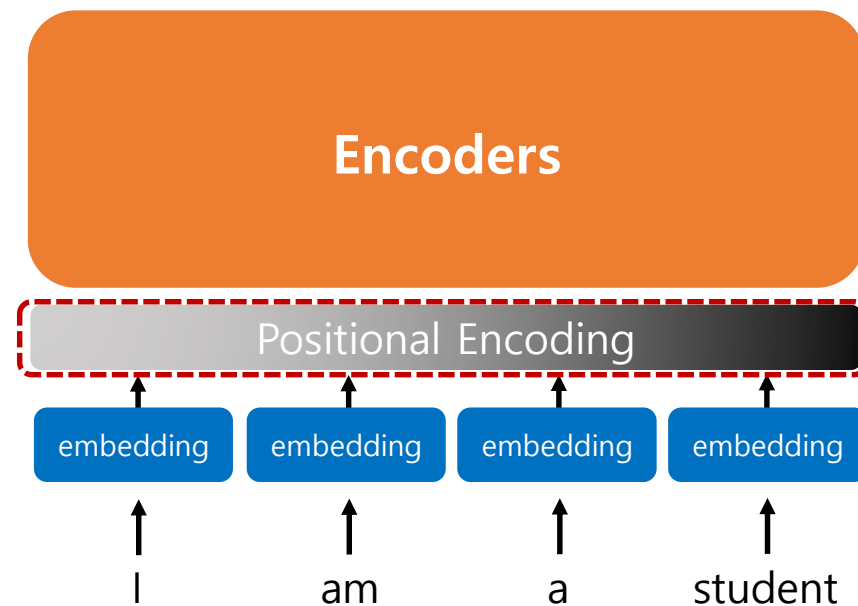
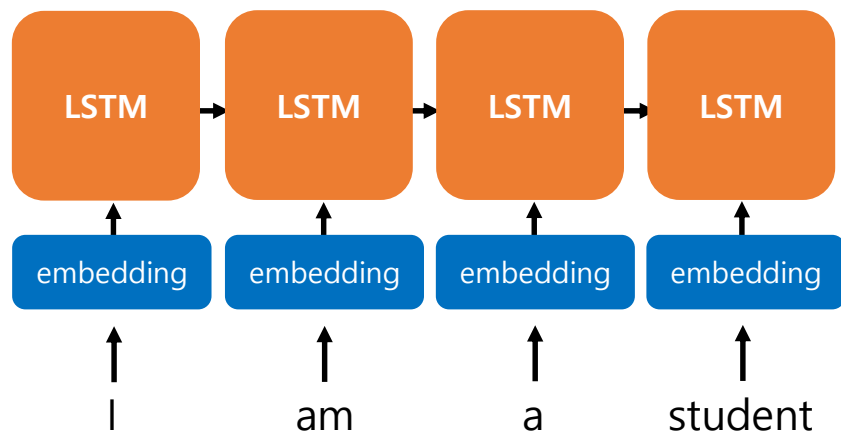
# Transformer : Positional Encoding

- RNN이 자연어 처리에서 유용했었던 이유는 단어 입력을 순차적으로 받으므로 각 단어의 위치 정보(position information)을 가질 수 있었기 때문.
- 하지만 트랜스포머의 경우, 입력을 병렬로 받으므로



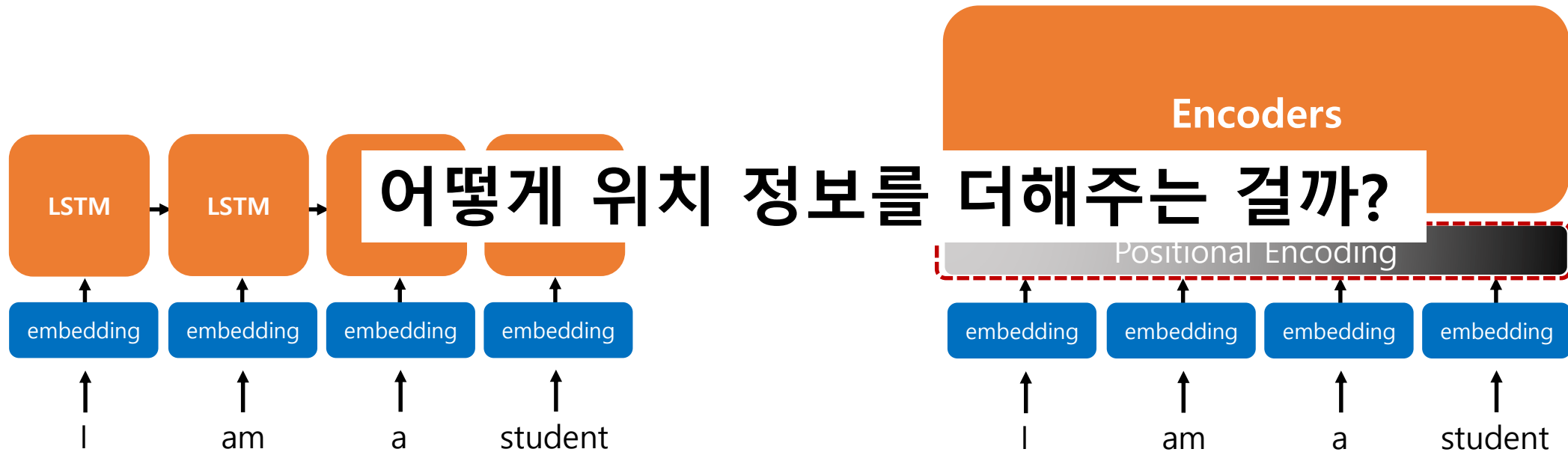
# Transformer : Positional Encoding

- RNN이 자연어 처리에서 유용했었던 이유는 단어 입력을 순차적으로 받으므로 각 단어의 위치 정보(position information)을 가질 수 있었기 때문.
- 하지만 트랜스포머의 경우, 입력을 병렬로 받으므로 **위치 정보를 더해줄 필요가 있음.**



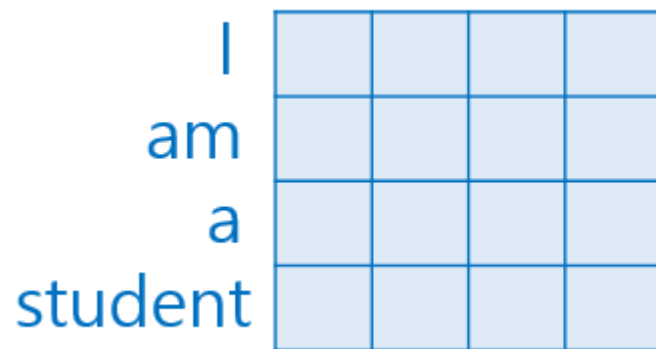
# Transformer : Positional Encoding

- RNN이 자연어 처리에서 유용했었던 이유는 단어 입력을 순차적으로 받으므로 각 단어의 위치 정보(position information)을 가질 수 있었기 때문.
- 하지만 트랜스포머의 경우, 입력을 병렬로 받으므로 **위치 정보를 더해줄 필요가 있음.**



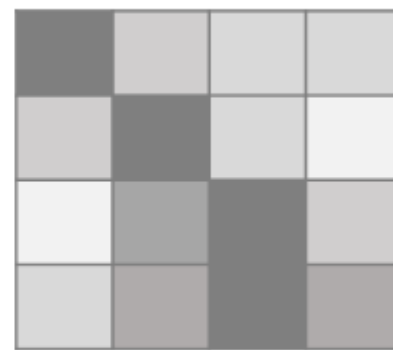
# Transformer : Positional Encoding

- 트랜스포머도 다른 딥 러닝 모델과 마찬가지로 Embedding layer가 존재.
- Embedding layer를 통해서 얻은 임베딩 벡터를 인코더와 디코더의 입력으로 한다.
- 임베딩 벡터에 **Postional Encoding**이라는 과정을 거친 후에 입력으로 한다.
- 이를 행렬 연산으로 이해해본다면 다음과 같이 이해할 수 있다.



문장 행렬

+



포지셔널 인코딩 행렬

이건 어떻게  
만드는 걸까요?

# Transformer : Positional Encoding

- **Postional Encoding** 행렬을 만들기 위해서 트랜스포머는 아래와 같은 수식을 사용.
- 사인 함수와 코사인 함수의 그래프를 통해서 위치에 따라 다른 정보를 더한다.

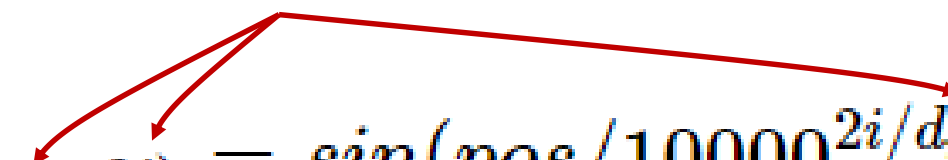
$$PE_{(pos, 2i)} = \sin(pos/10000^{2i/d_{model}})$$

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

# Transformer : Positional Encoding

- **Postional Encoding** 행렬을 만들기 위해서 트랜스포머는 아래와 같은 수식을 사용.
- 사인 함수와 코사인 함수의 그래프를 통해서 위치에 따라 다른 정보를 더한다.

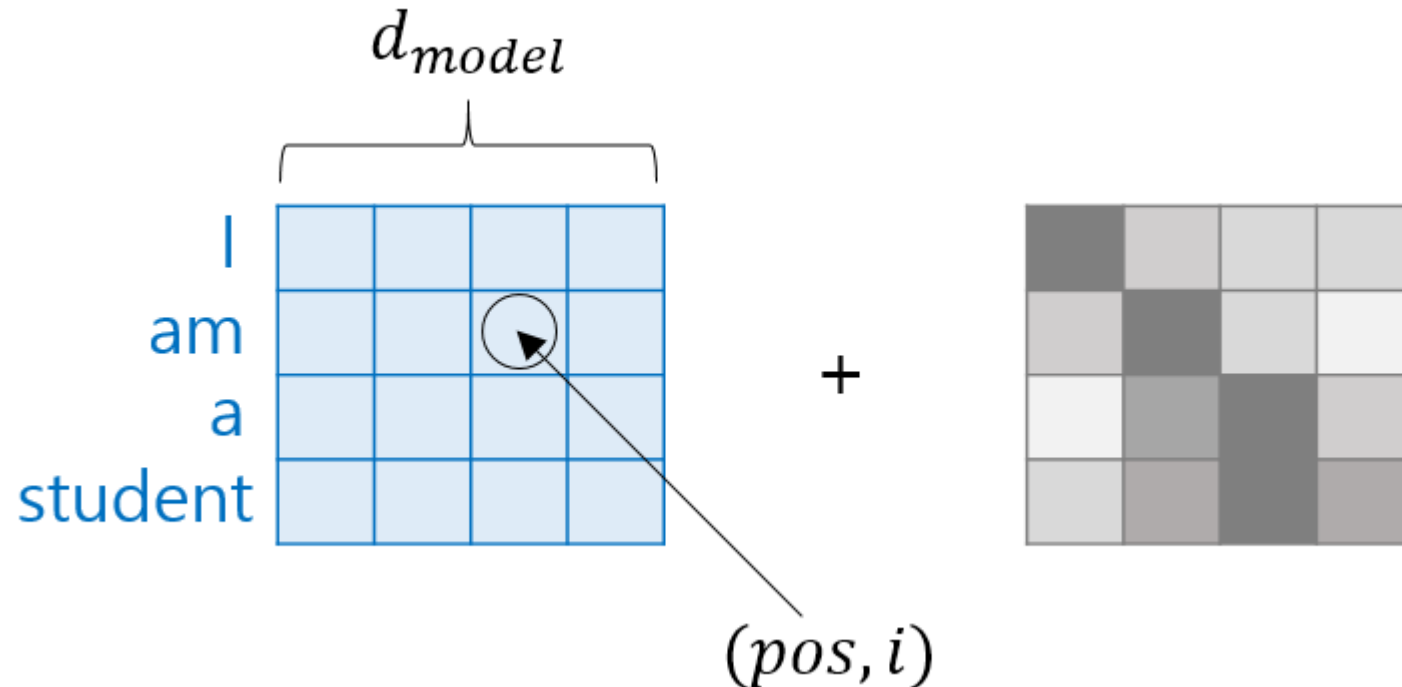
각 변수들의 의미?


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

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

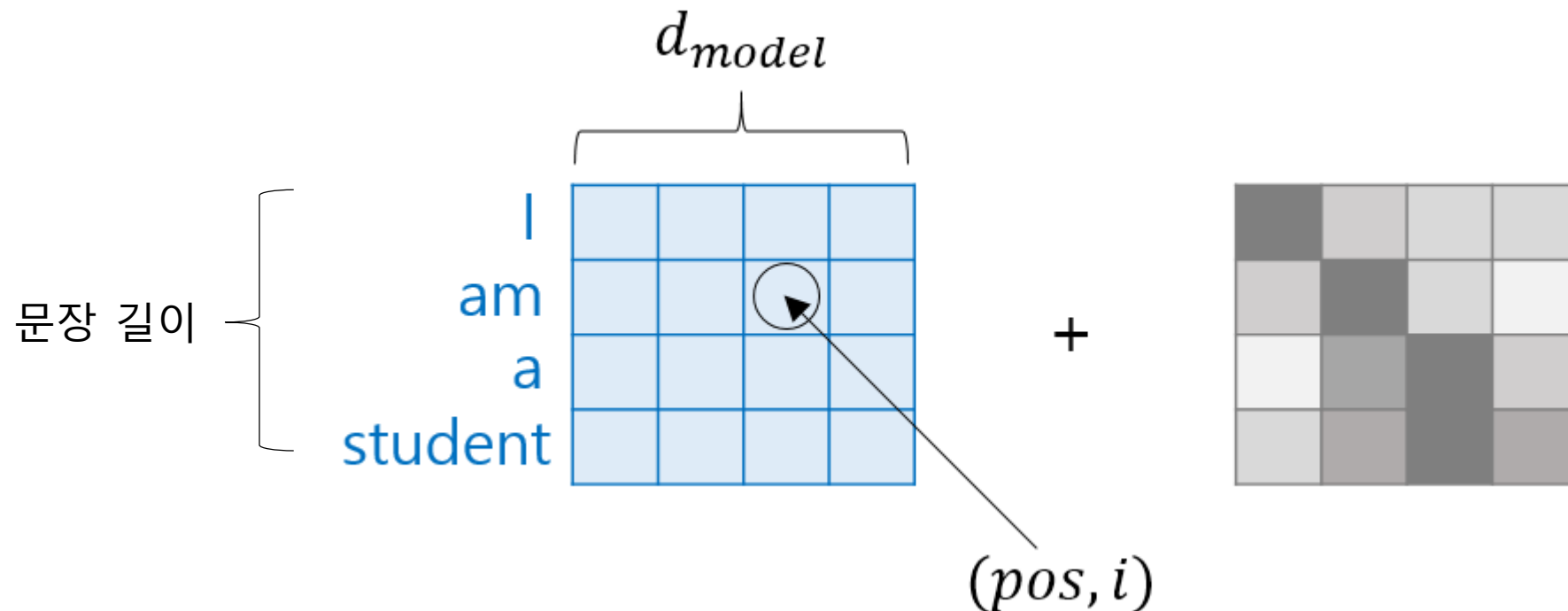
# Transformer : Positional Encoding

- $pos$ 는 입력 문장에서의 임베딩 벡터의 위치
- $i$ 는 임베딩 벡터의 차원
- $d_{model}$ 은 트랜스포머의 입, 출력 차원. 또한 임베딩 벡터의 차원이기도 하다.



# Transformer : Positional Encoding

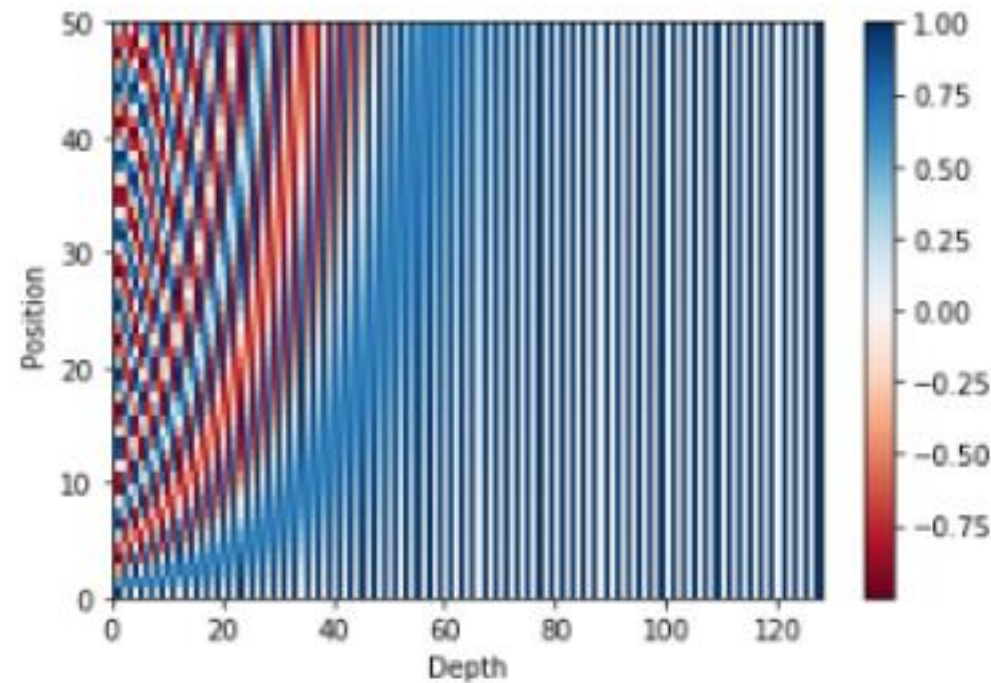
- $pos$ 는 입력 문장에서의 임베딩 벡터의 위치
- $i$ 는 임베딩 벡터의 차원
- $d_{model}$ 은 트랜스포머의 입, 출력 차원. 또한 임베딩 벡터의 차원이기도 하다.





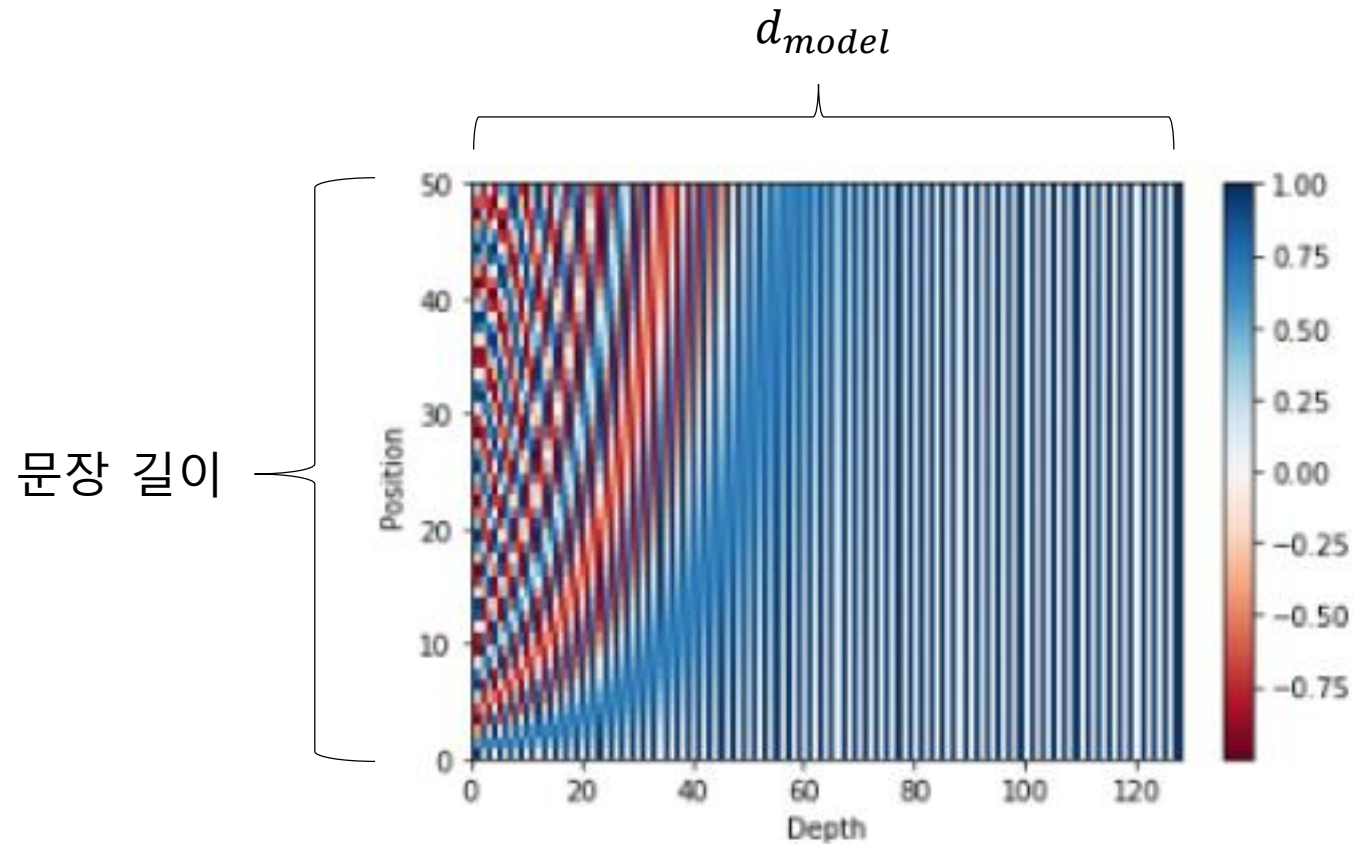
# Transformer : Positional Encoding

- 다음의 그림은 문장 길이 50, 임베딩 벡터의 차원이 128일 경우의 Positional Encoding 행렬

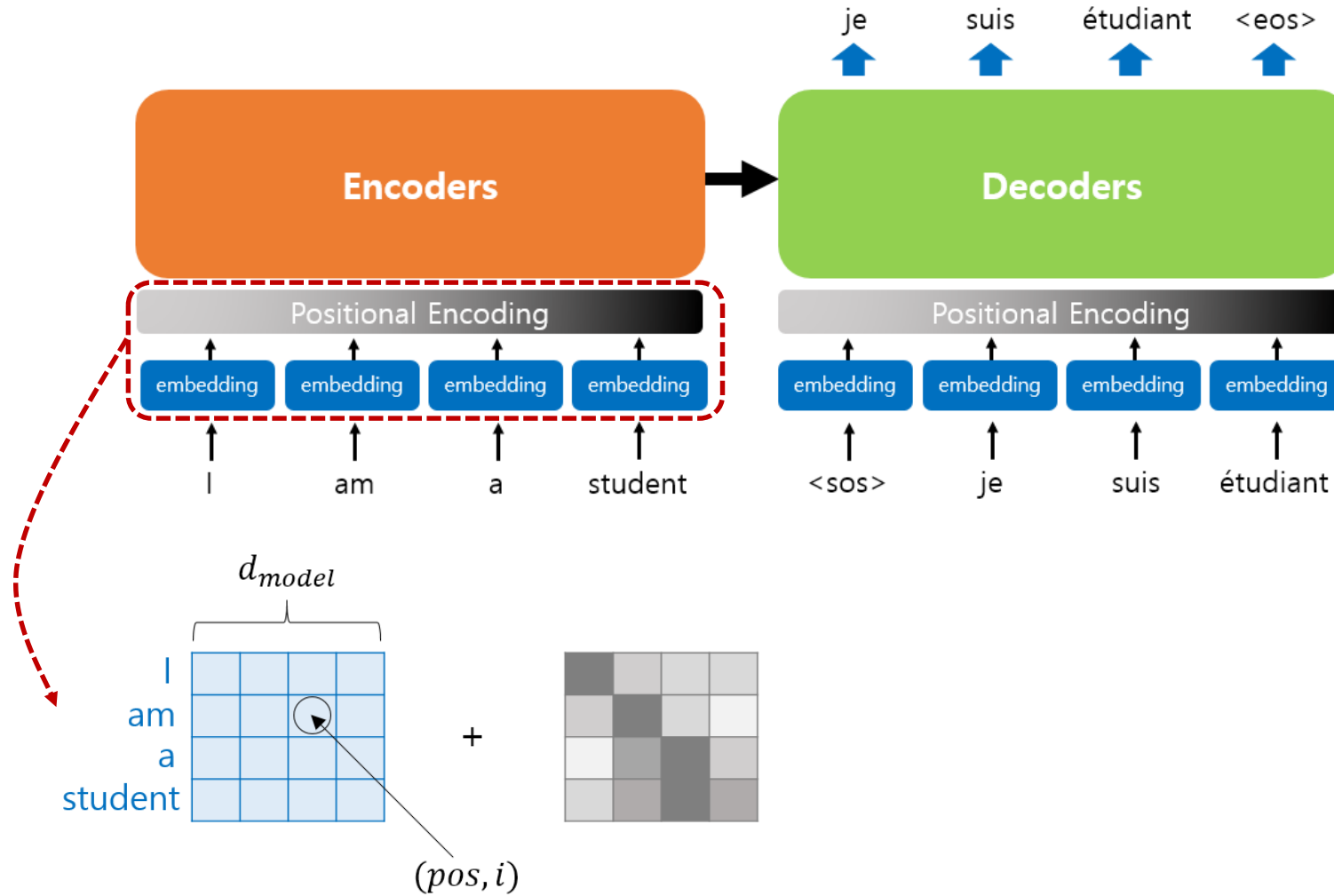


# Transformer : Positional Encoding

- 다음의 그림은 문장 길이 50, 임베딩 벡터의 차원이 128일 경우의 Positional Encoding 행렬



# Transformer : Positional Encoding

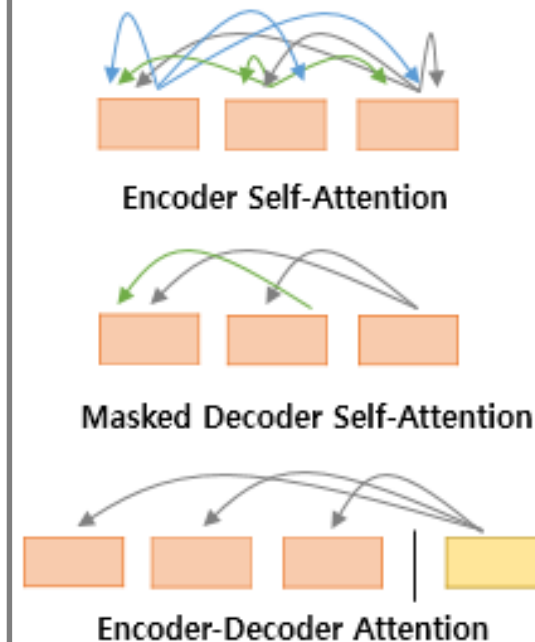
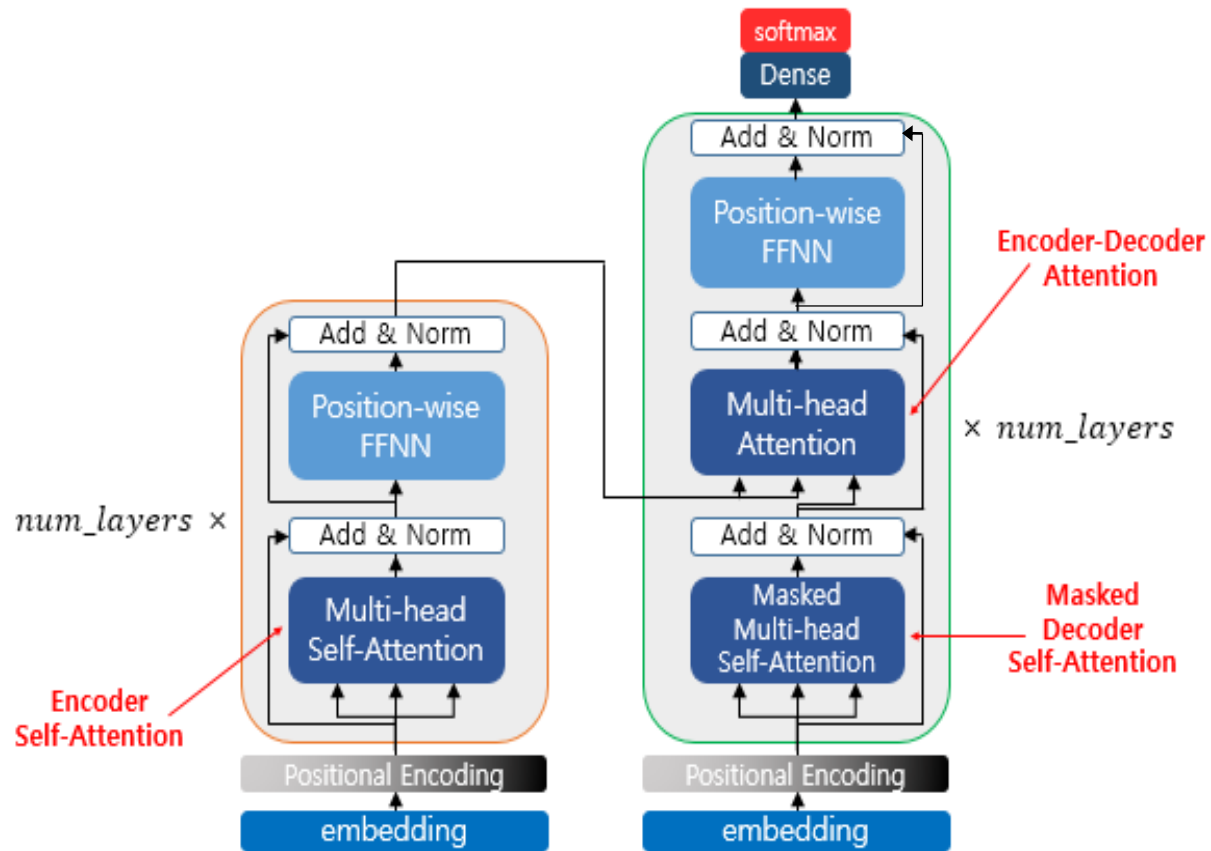


# Transformer : Positional Encoding

- 포지셔널 인코딩에 대한 수학적 고찰 : <https://www.aitimes.kr/news/articleView.html?idxno=17442>

# Transformer : Attention Mechanism

- 트랜스포머는 총 세 종류의 어텐션이 존재.



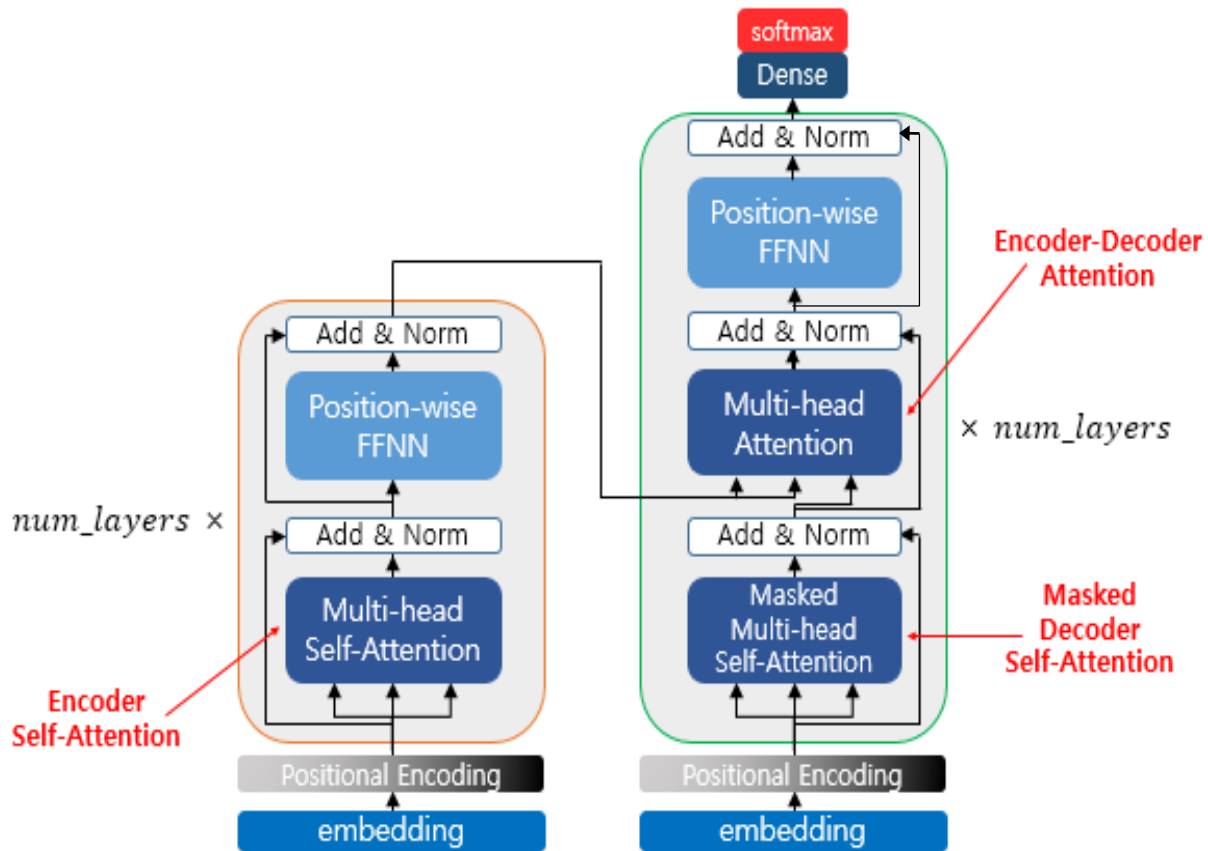
Query = Key = Value

Query = Key = Value

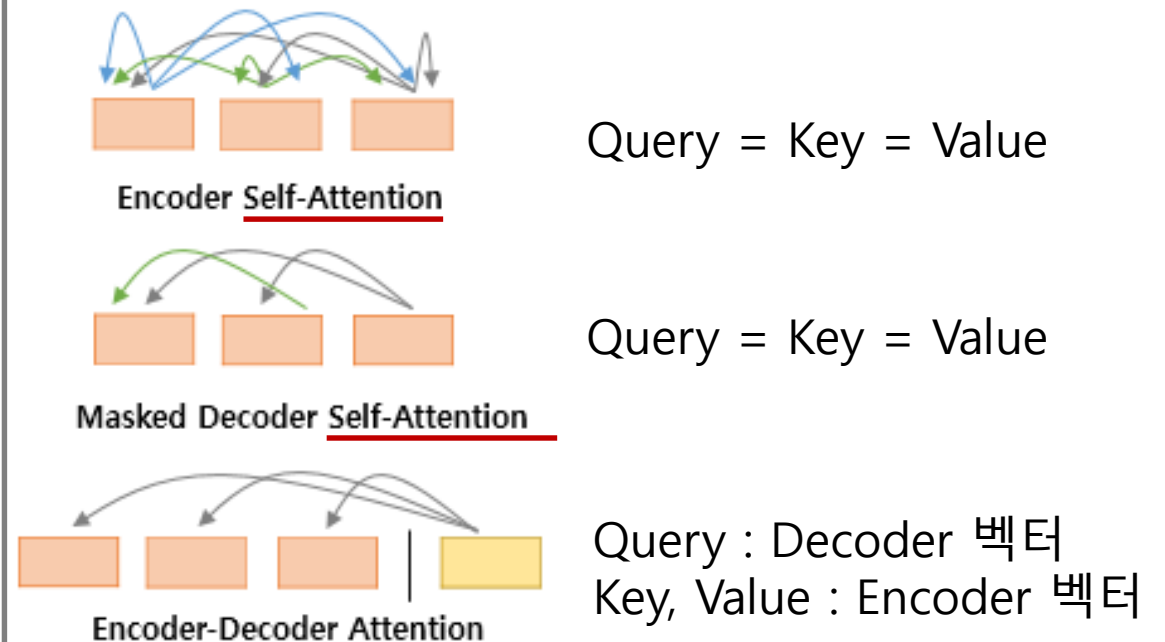
Query : Decoder 벡터  
Key, Value : Encoder 벡터

# Transformer : Attention Mechanism

- 트랜스포머는 총 세 종류의 어텐션이 존재.

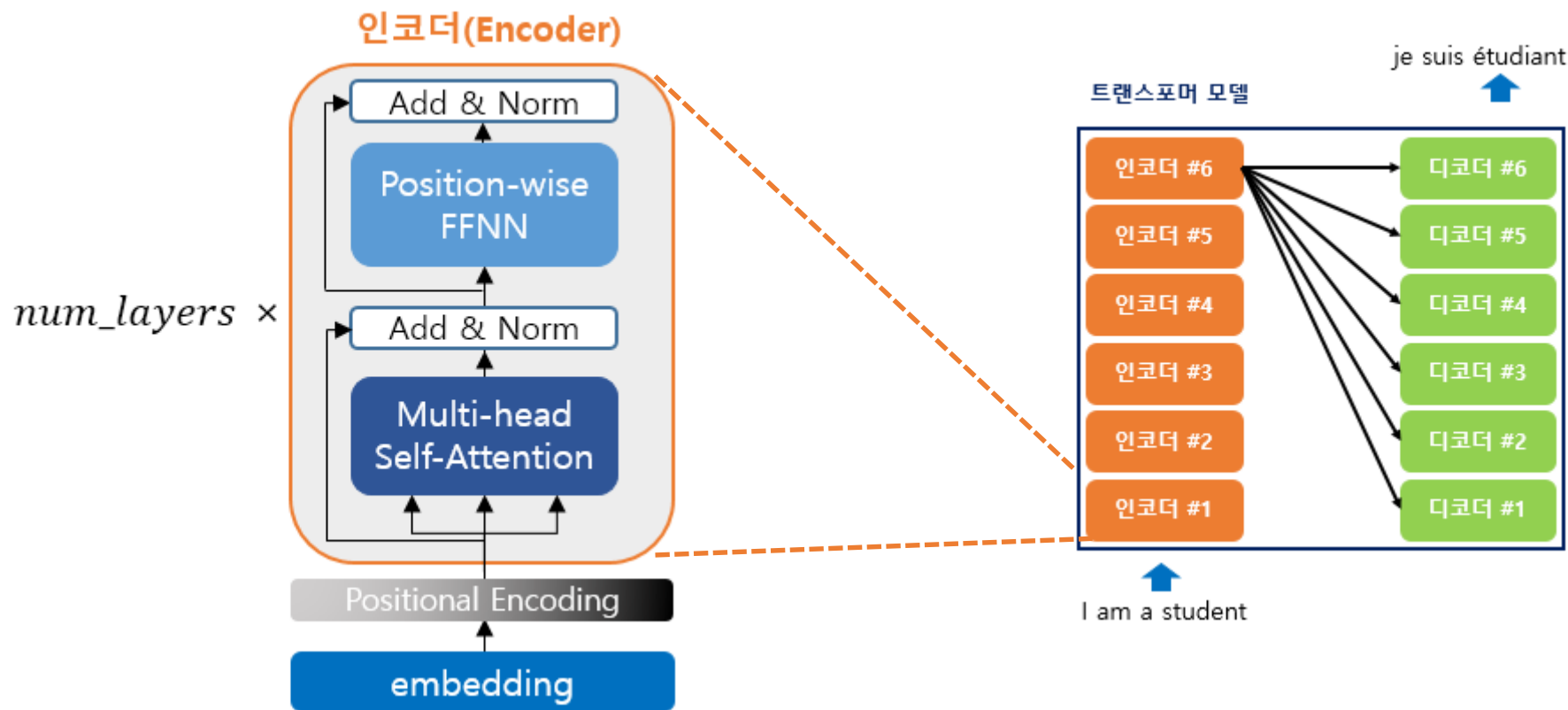


$Q = K = V$ 인 경우에는  
Self-Attention이라고 부른다.



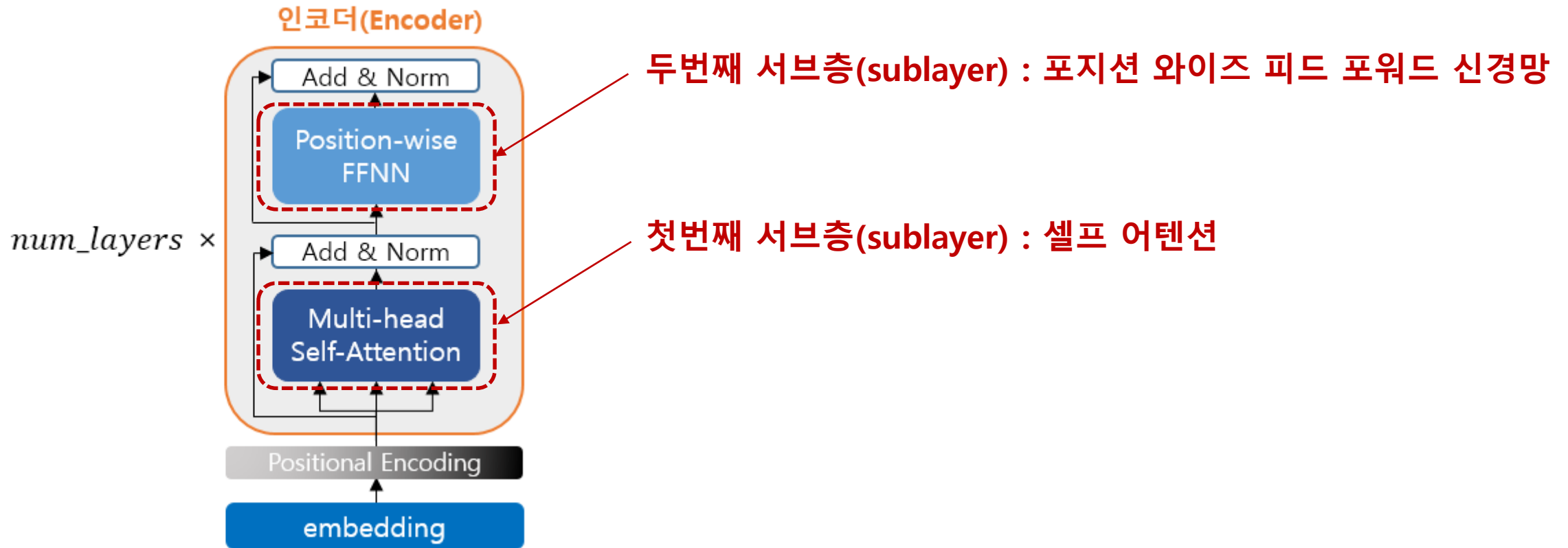
# Transformer Encoder

- 인코더를 1개의 층(layer)으로 생각하였을 때, 논문에서는 총 6개의 layer가 존재.
- $\text{num\_layers} = 6$
- 인코더 내부에는 총 2개의 서브층이 존재.



# Transformer Encoder

- 인코더를 1개의 층(layer)으로 생각하였을 때, 논문에서는 총 6개의 layer가 존재.
- `num_layers = 6`
- 인코더 내부에는 총 2개의 서브층이 존재.





# Transformer Encoder : Self-Attention

- seq2seq + Attention의 경우, Query, Key, Value는 다음과 같다.

Q = Query : t 시점의 디코더 셀에서의 은닉 상태  
K = Keys : 모든 시점의 인코더 셀의 은닉 상태들  
V = Values : 모든 시점의 인코더 셀의 은닉 상태들

# Transformer Encoder : Self-Attention

- seq2seq + Attention의 경우, Query, Key, Value는 다음과 같다.
- 디코더의 t시점이라는 것은 계속 변화하면서 반복되므로 다음과 같이 일반화가 가능.

Q = Query : **t 시점의 디코더 셀에서의 은닉 상태**

K = Keys : 모든 시점의 인코더 셀의 은닉 상태들

V = Values : 모든 시점의 인코더 셀의 은닉 상태들

# Transformer Encoder : Self-Attention

- seq2seq + Attention의 경우, Query, Key, Value는 다음과 같다.
- 디코더의 t시점이라는 것은 계속 변화하면서 반복되므로 다음과 같이 일반화가 가능.

Q = Querys : **모든 시점의 디코더 셀에서의 은닉 상태들**

K = Keys : 모든 시점의 인코더 셀의 은닉 상태들

V = Values : 모든 시점의 인코더 셀의 은닉 상태들

Q는 디코더 벡터지만, K와 V는 인코더 벡터.  
K와 V는 같지만 Q는 같지 않다.

# Transformer Encoder : Self-Attention

- seq2seq + Attention의 경우, Query, Key, Value는 다음과 같다.
- 디코더의 t시점이라는 것은 계속 변화하면서 반복되므로 다음과 같이 일반화가 가능.

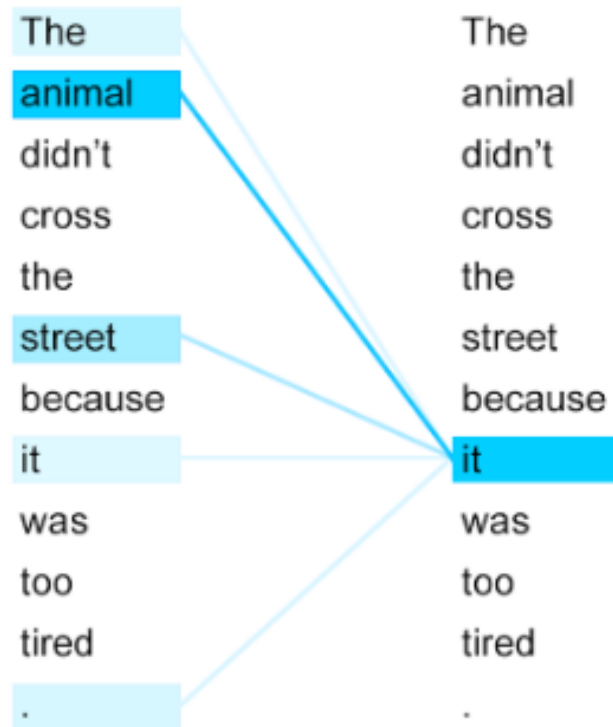
Q = Querys : 모든 시점의 디코더 셀에서의 은닉 상태들  
K = Keys : 모든 시점의 인코더 셀의 은닉 상태들  
V = Values : 모든 시점의 인코더 셀의 은닉 상태들

- Self-Attention : Querys = Keys = Values가 모두 동일한 경우를 의미함.

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

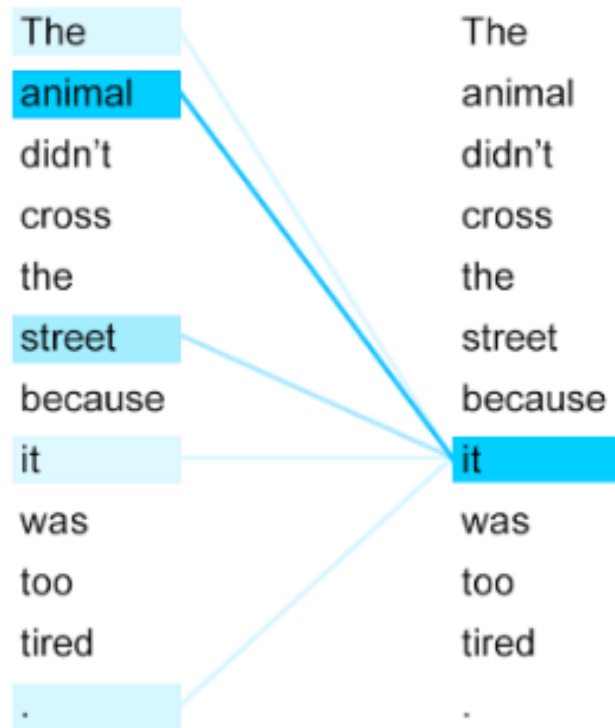
# Transformer Encoder : Self-Attention

- '그 동물은 길을 건너지 않았다. 왜냐하면 그것은 너무 피곤하였기 때문이다.' 라는 의미
- 여기서 그것(it)에 해당하는 것은 과연 길(street)일까, 동물(animal)일까?



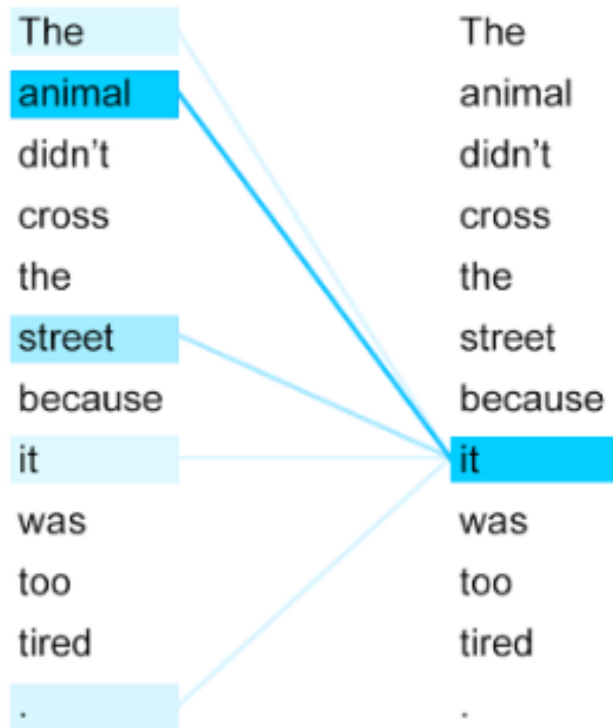
# Transformer Encoder : Self-Attention

- '그 동물은 길을 건너지 않았다. 왜냐하면 그것은 너무 피곤하였기 때문이다.' 라는 의미
- 여기서 그것(it)에 해당하는 것은 과연 길(street)일까, 동물(animal)일까?
- 인간에게는 굉장히 쉬운 문제지만 기계에게는 그렇지 않음.



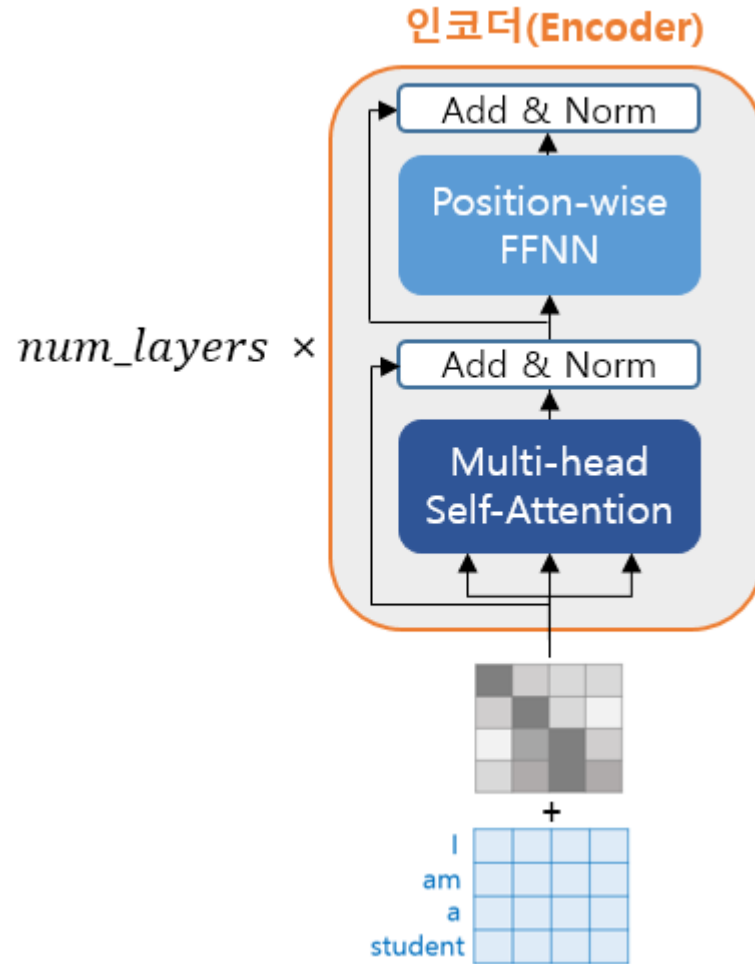
# Transformer Encoder : Self-Attention

- '그 동물은 길을 건너지 않았다. 왜냐하면 그것은 너무 피곤하였기 때문이다.' 라는 의미
- 여기서 그것(it)에 해당하는 것은 과연 길(street)일까, 동물(animal)일까?
- 인간에게는 굉장히 쉬운 문제지만 기계에게는 그렇지 않음.



**셀프 어텐션은 입력 문장 내의 단어들끼리 유사도를 구하므로서 그것(it)이 동물(animal)과 연관되었을 확률이 높다는 것을 찾아낸다.**

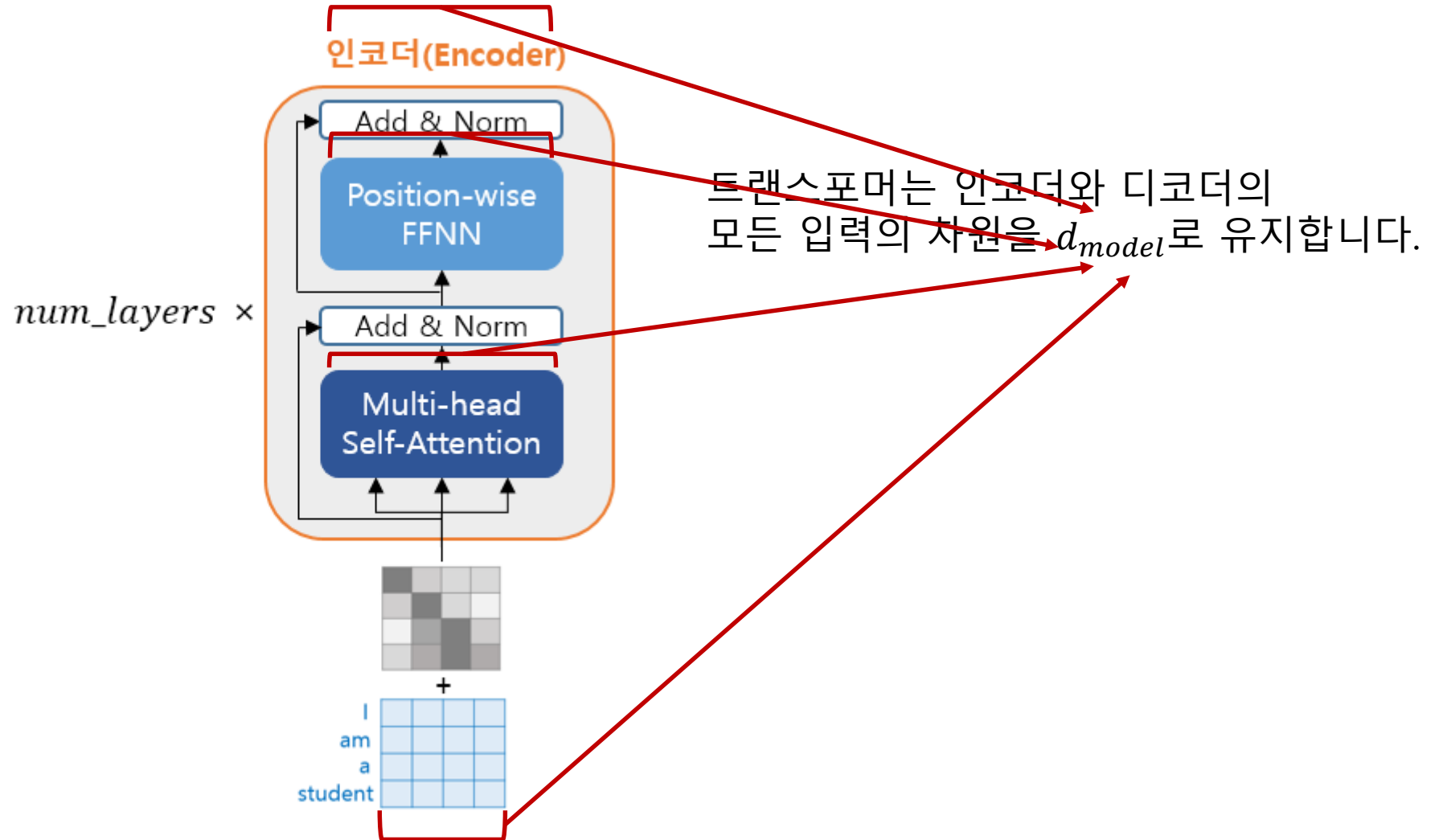
# Transformer Encoder



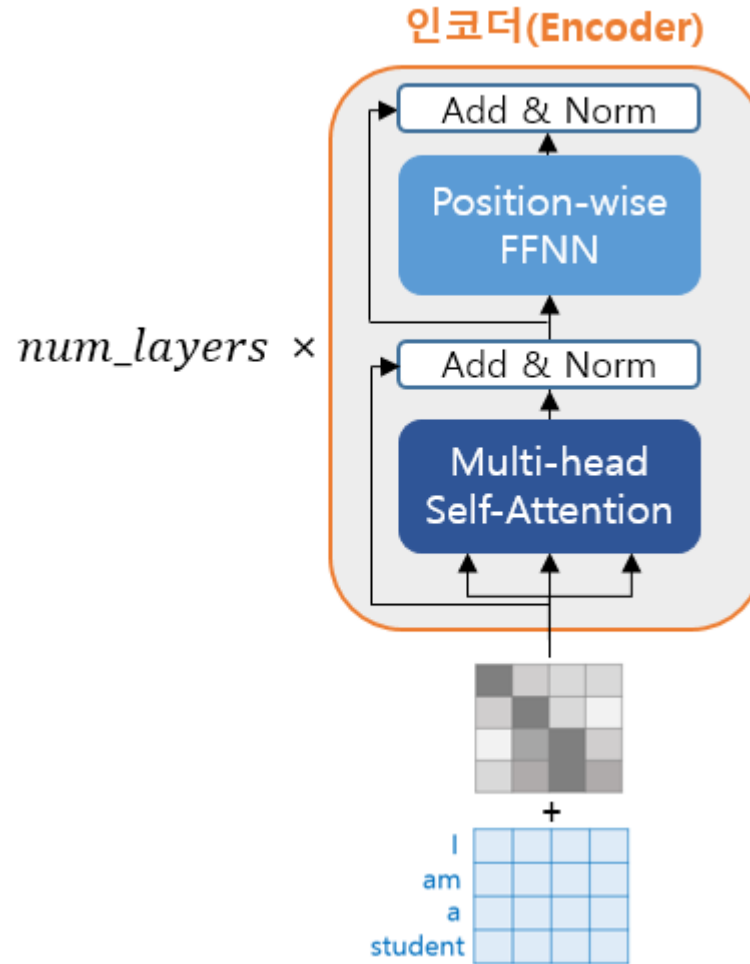
트랜스포머는 인코더와 디코더의 모든 입력의 차원을  $d_{model}$ 로 유지합니다.



# Transformer Encoder



# Transformer Encoder

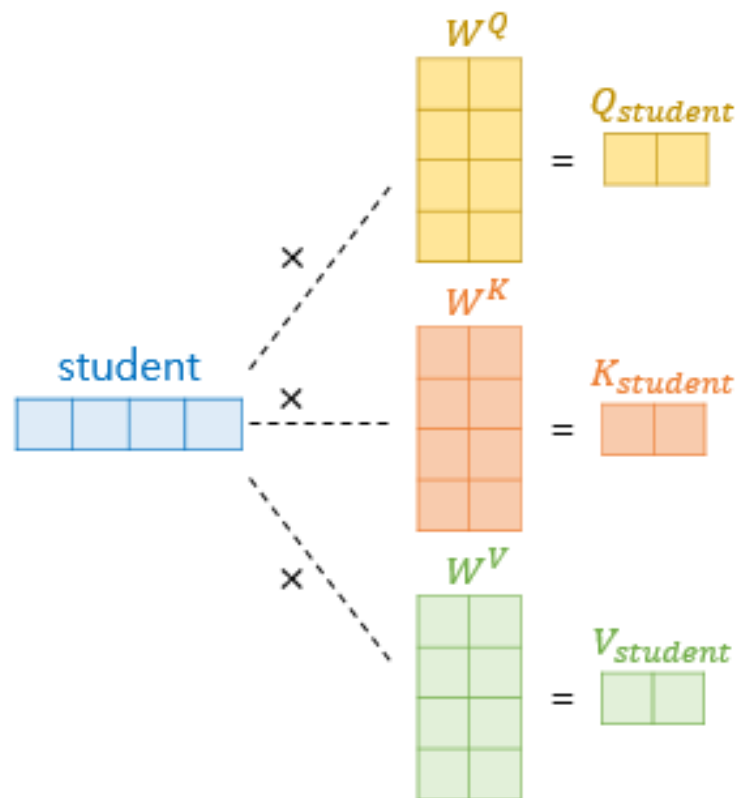


트랜스포머는 인코더와 디코더의 모든 입력의 차원을  $d_{model}$ 로 유지합니다.

논문에서  $d_{model} = 512$

# Transformer Encoder : Self-Attention

- 셀프 어텐션은 입력 벡터에서 가중치 행렬 곱으로 Q, K, V 벡터를 얻고 수행.
- 스케일드 닷 프로덕트 어텐션을 통해 각각의 Q벡터가 각각의 K벡터에 대해서 스코어 연산.



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

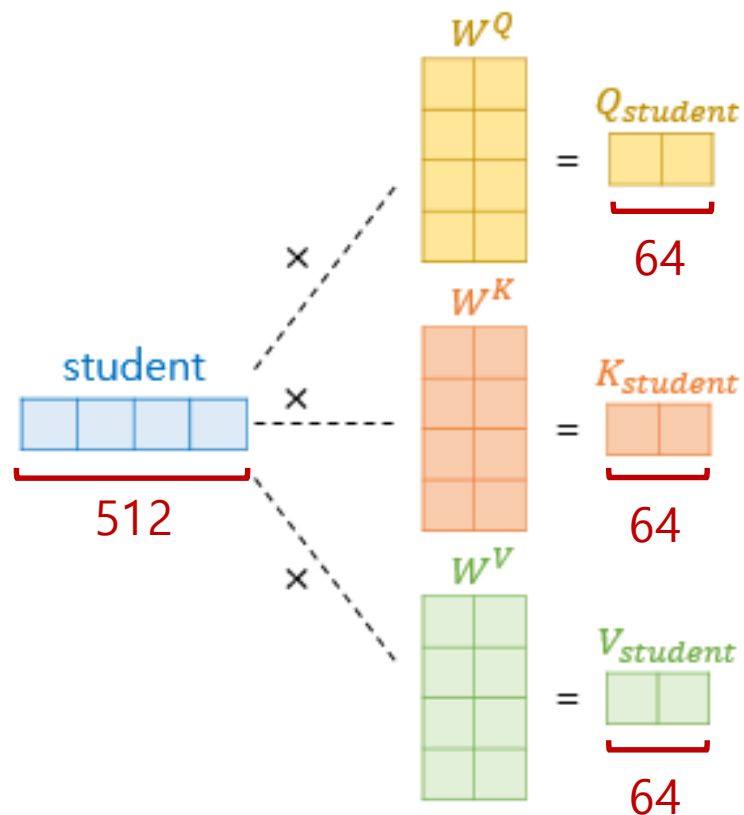
The diagram shows the calculation of attention scores for a specific Q vector ( $Q_I$ ) against multiple K vectors ( $K_I^T$ ,  $K_{am}^T$ ,  $K_a^T$ , and  $K_{student}^T$ ). The scores are calculated as follows:

Q Vector	K Vector	Dot Product	Scaled Dot Product
$Q_I$	$K_I^T$	$128$	$128 / \sqrt{d_k} = 16$
$Q_I$	$K_{am}^T$	$32$	$32 / \sqrt{d_k} = 4$
$Q_I$	$K_a^T$	$32$	$32 / \sqrt{d_k} = 4$
$Q_I$	$K_{student}^T$	$128$	$128 / \sqrt{d_k} = 16$

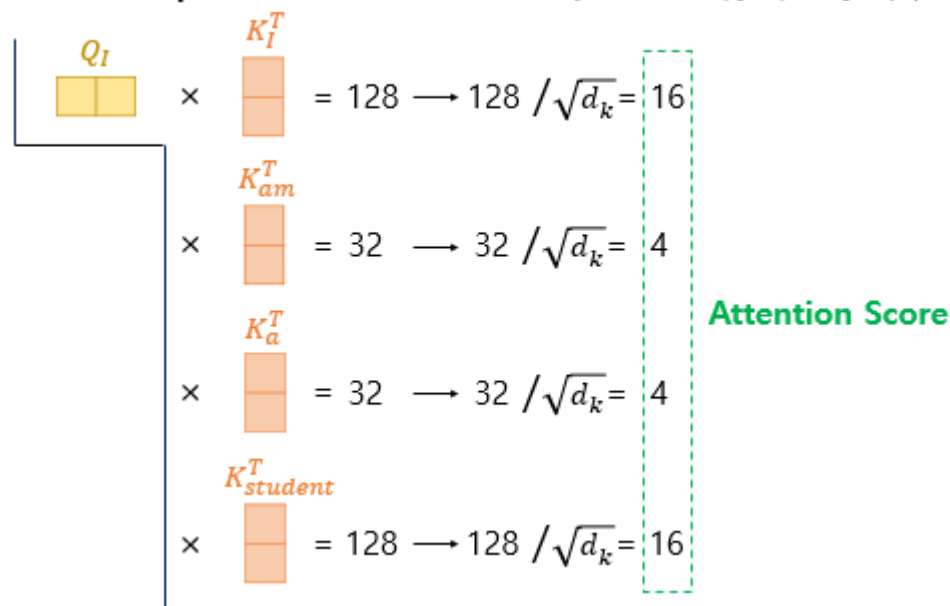
The scores 16, 4, 4, and 16 are grouped together in a green dashed box labeled "Attention Score".

# Transformer Encoder : Self-Attention

- 셀프 어텐션은 입력 벡터에서 가중치 행렬 곱으로 Q, K, V 벡터를 얻고 수행.
- 스케일드 닷 프로덕트 어텐션을 통해 각각의 Q벡터가 각각의 K벡터에 대해서 스코어 연산.



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



# 어텐션 메커니즘의 종류 (지난 강의)

어텐션 메커니즘은 다음 순서로 제안되었다. 이들의 큰 차이는 어텐션 스코어 함수가 다르다는 점이다.

## 1. 바다나우 어텐션 (Neural Machine Translation by Jointly Learning to Align and Translate)

- 콘캣 어텐션(Concat Attention) :  $score(s_t, h_i) = W_a^T \tanh(W_b[s_t; h_i])$

## 2. 루옹 어텐션 (Effective Approaches to Attention-based Neural Machine Translation)

- 닷 프로덕트 어텐션(Dot-Product Attention) :  $score(s_t, h_i) = s_t^T h_i$
- 제네럴 어텐션(General Attention) :  $score(s_t, h_i) = s_t^T W_a h_i$

## 3. 스케일드 닷 프로덕트 어텐션(Attention is All you need)

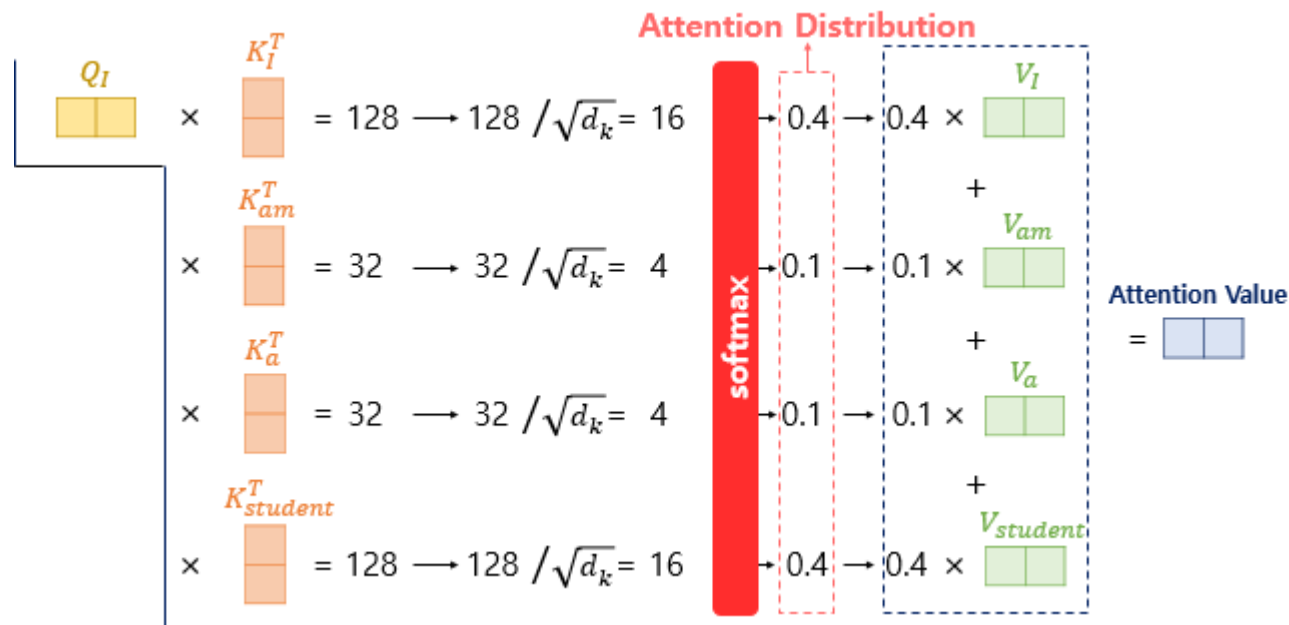
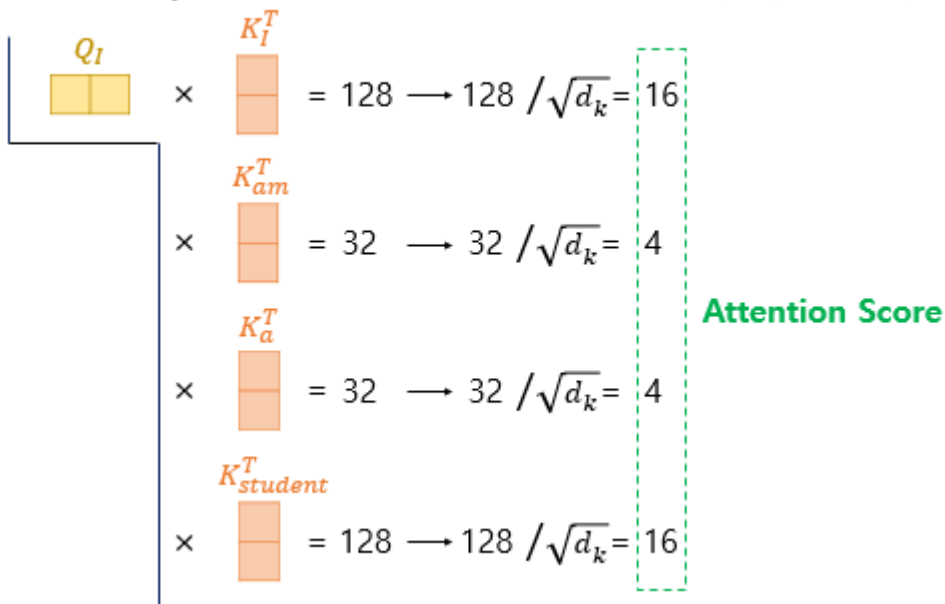
- **스케일드 닷 프로덕트 어텐션(Scaled Dot-Product Attention)** :  $score(s_t, h_i) = \frac{s_t^T h_i}{\sqrt{n}}$

트랜스포머 논문에서 등장

# Self-Attention

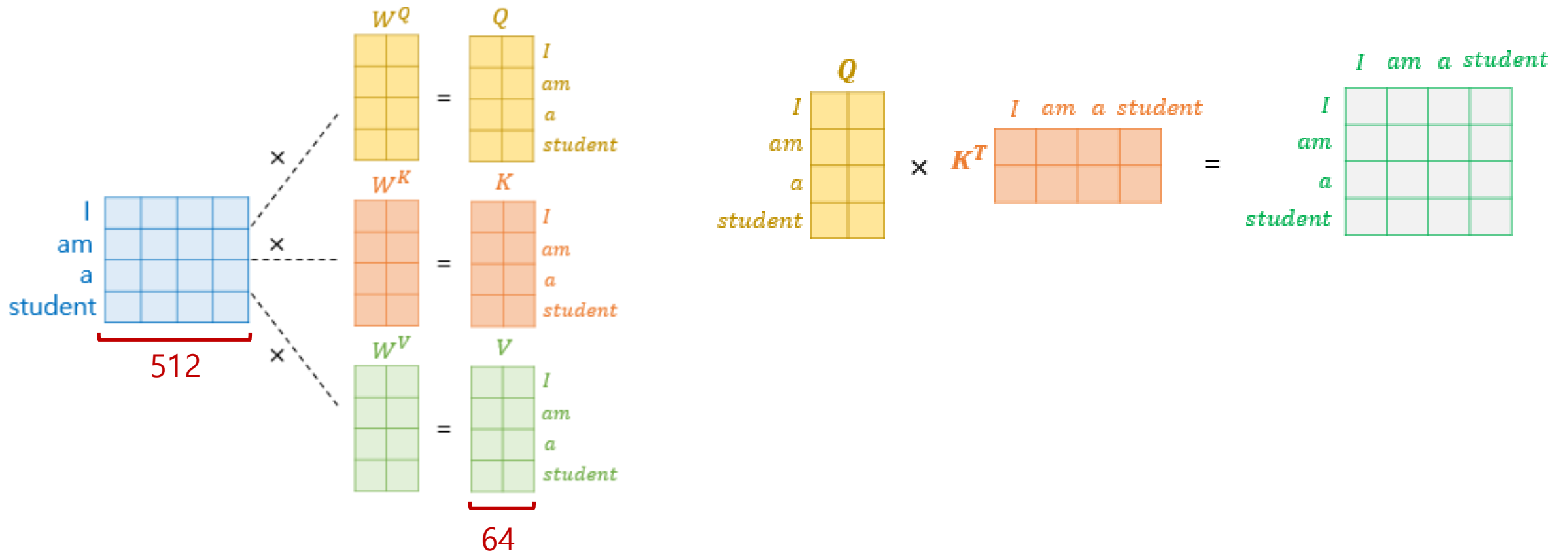
- 셀프 어텐션은 입력 벡터에서 가중치 행렬 곱으로 Q, K, V 벡터를 얻고 수행.
- 스케일드 닷 프로덕트 어텐션을 통해 각각의 Q벡터가 각각의 K벡터에 대해서 스코어 연산.

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



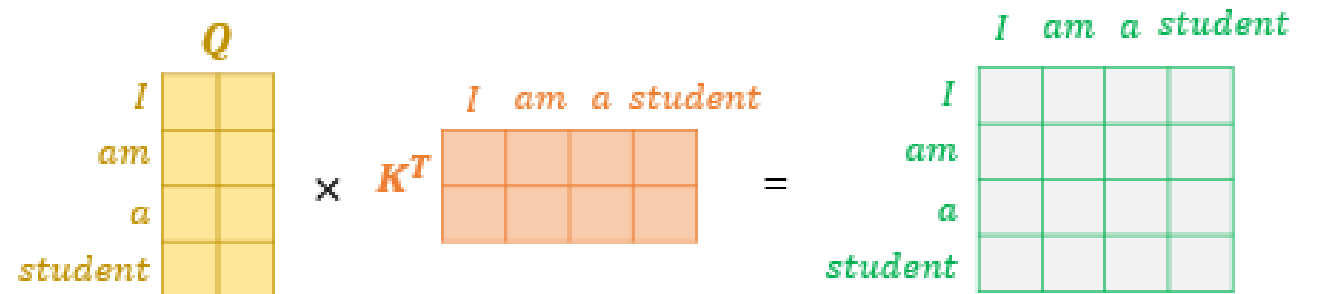
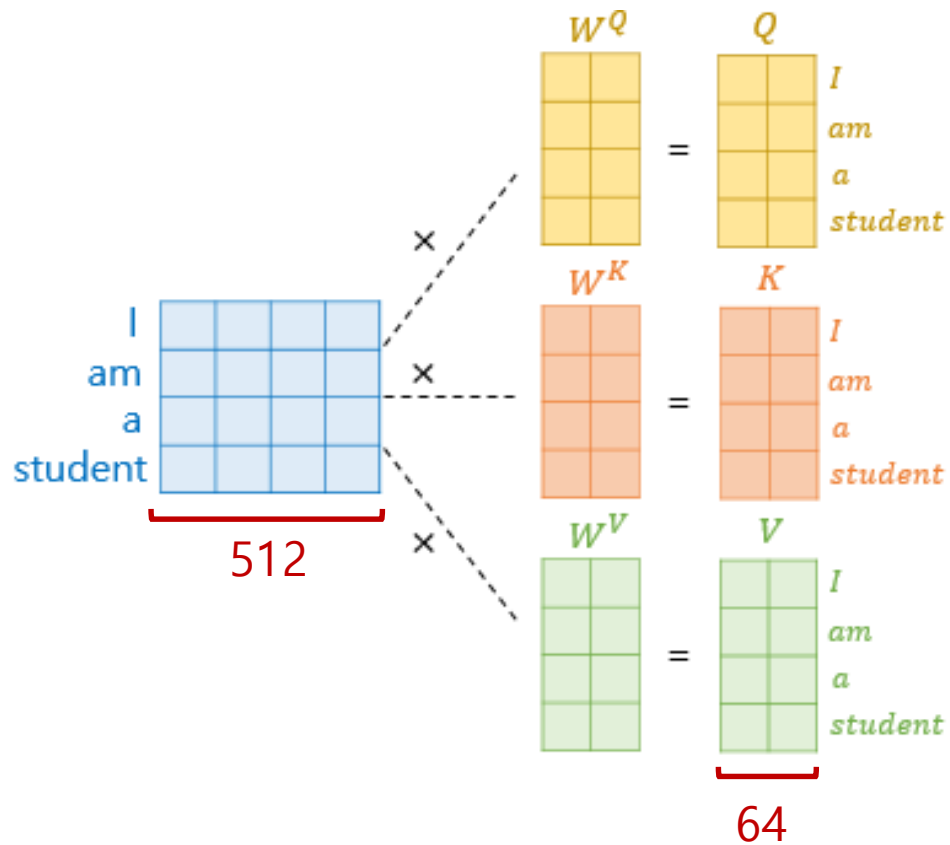
# Self-Attention 행렬 연산으로 이해하기

- 실제로 이 연산은 벡터 간 연산이 아니라 행렬 연산으로 이루어진다.



# Self-Attention 행렬 연산으로 이해하기

- 실제로 이 연산은 벡터 간 연산이 아니라 행렬 연산으로 이루어진다.

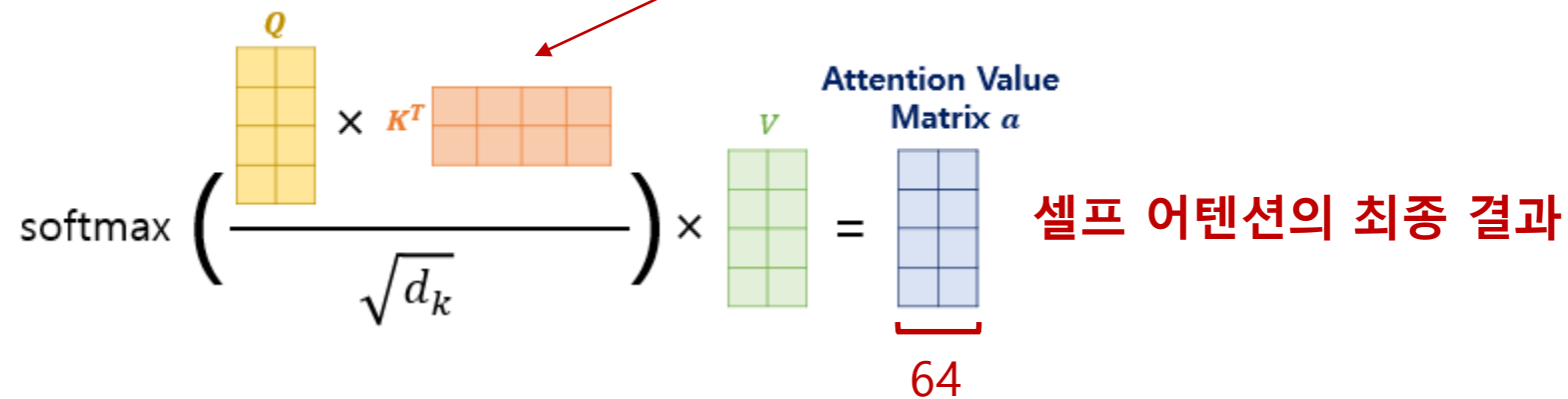
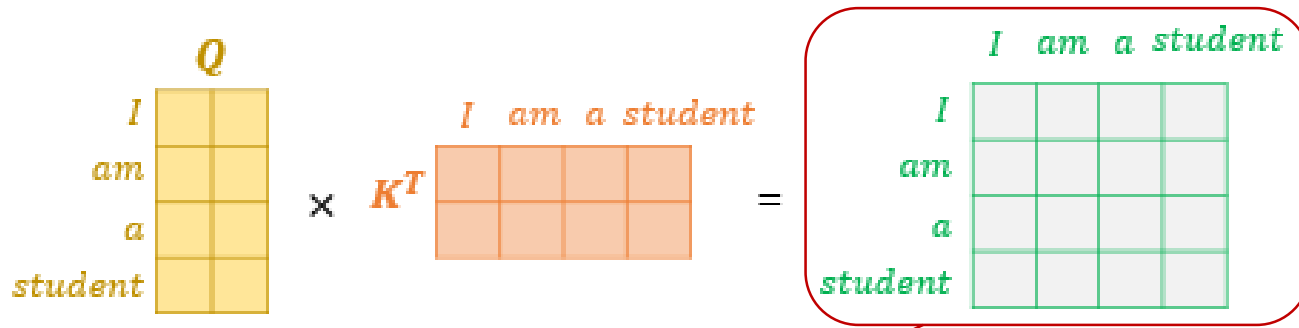


위 행렬에 특정 값을 나누어주면 어텐션 스코어 행렬.



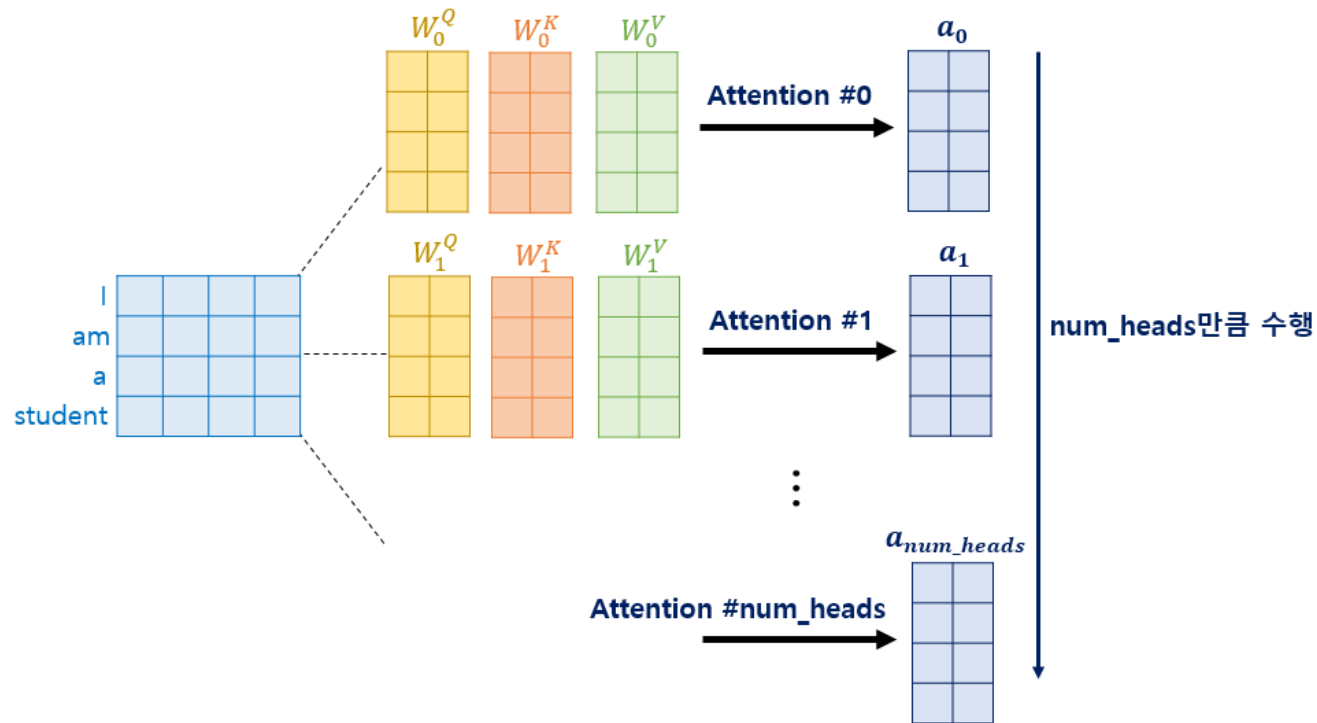
# Self-Attention 행렬 연산으로 이해하기

- 실제로 이 연산은 벡터 간 연산이 아니라 행렬 연산으로 이루어진다.



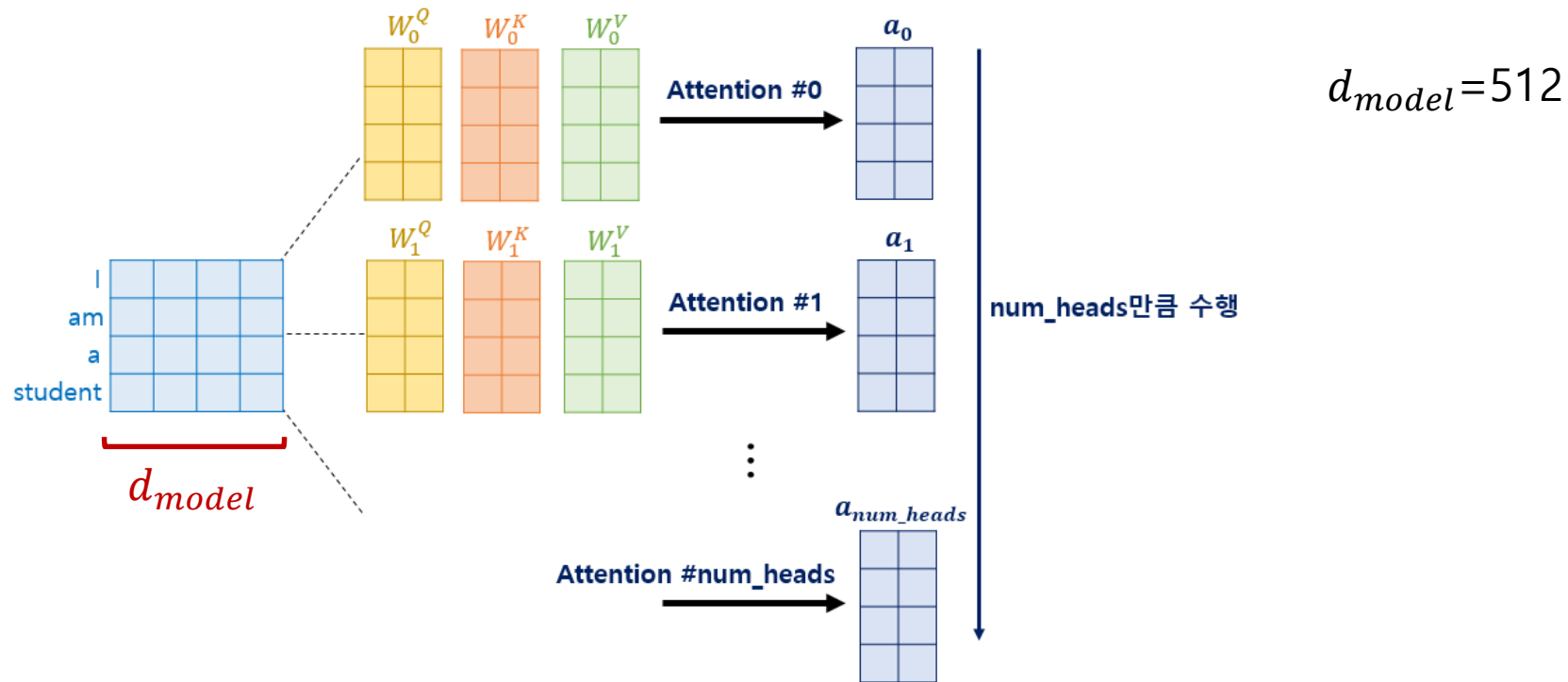
# Multi-Head Attention

- 트랜스포머는 어텐션에 대해서 병렬적으로 수행한다는 특징을 가지고 있다.
- 이를 집단 지성, 다수의 머리를 이용한다고 하여 Multi-Head Attention이라고 부른다.



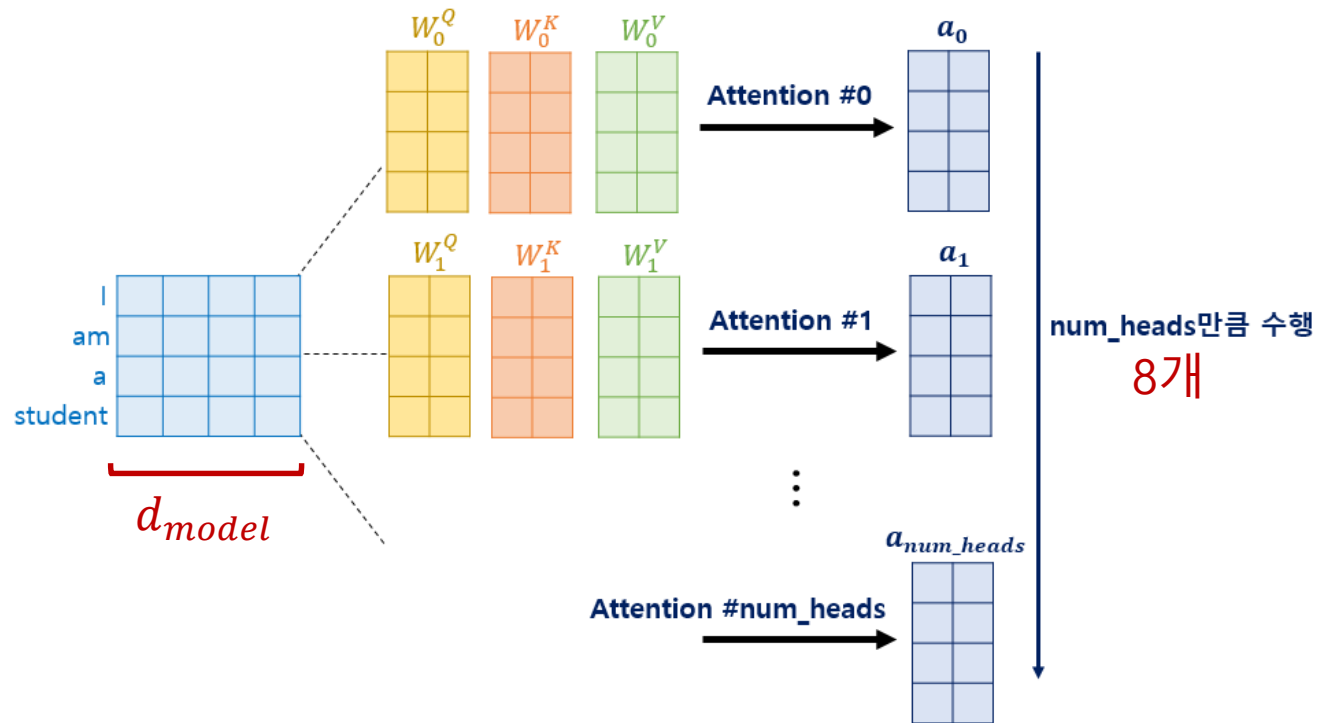
# Multi-Head Attention

- 트랜스포머는 어텐션에 대해서 병렬적으로 수행한다는 특징을 가지고 있다.
- 이를 집단 지성, 다수의 머리를 이용한다고 하여 Multi-Head Attention이라고 부른다.



# Multi-Head Attention

- 트랜스포머는 어텐션에 대해서 병렬적으로 수행한다는 특징을 가지고 있다.
- 이를 집단 지성, 다수의 머리를 이용한다고 하여 Multi-Head Attention이라고 부른다.

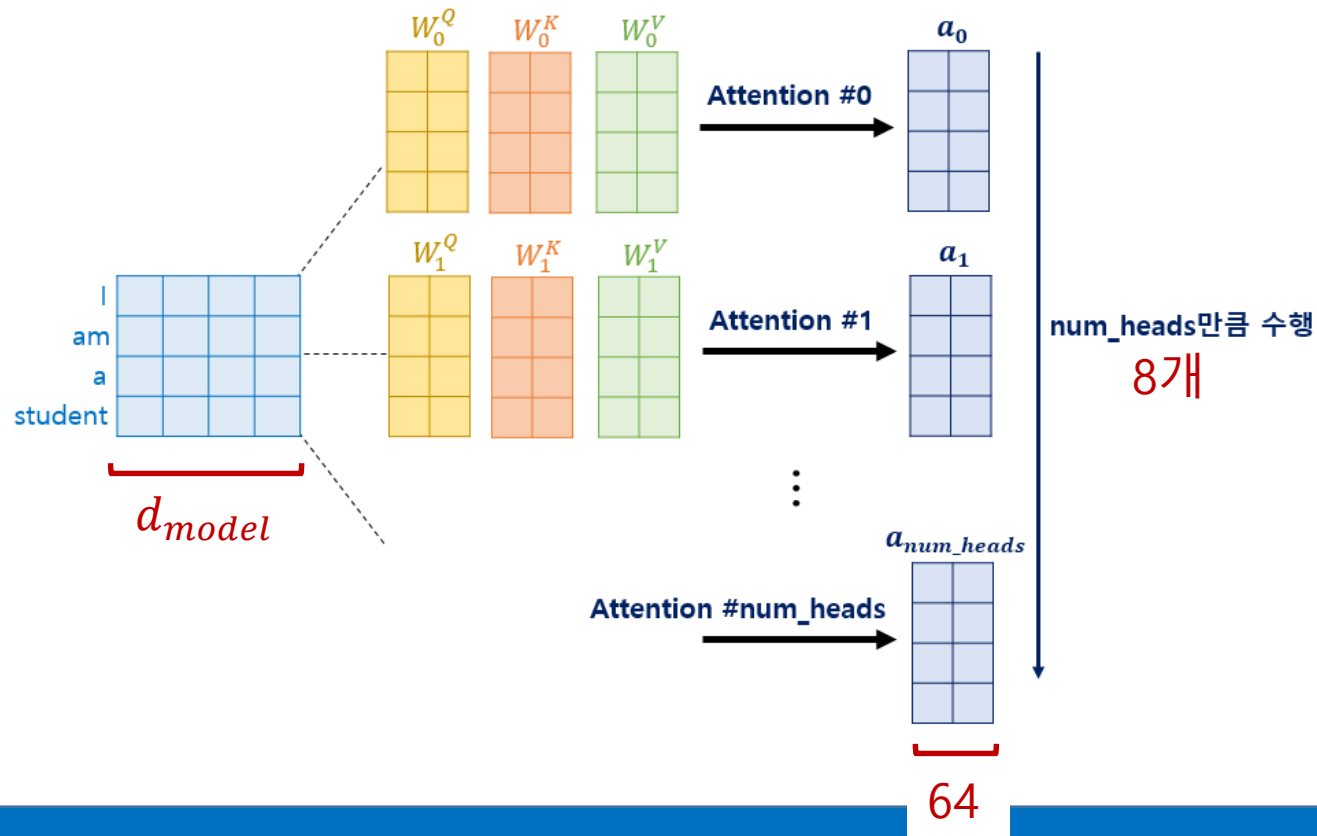


$$d_{model}=512$$

$$num\_heads = 8$$

# Multi-Head Attention

- 트랜스포머는 어텐션에 대해서 병렬적으로 수행한다는 특징을 가지고 있다.
- 이를 집단 지성, 다수의 머리를 이용한다고 하여 Multi-Head Attention이라고 부른다.



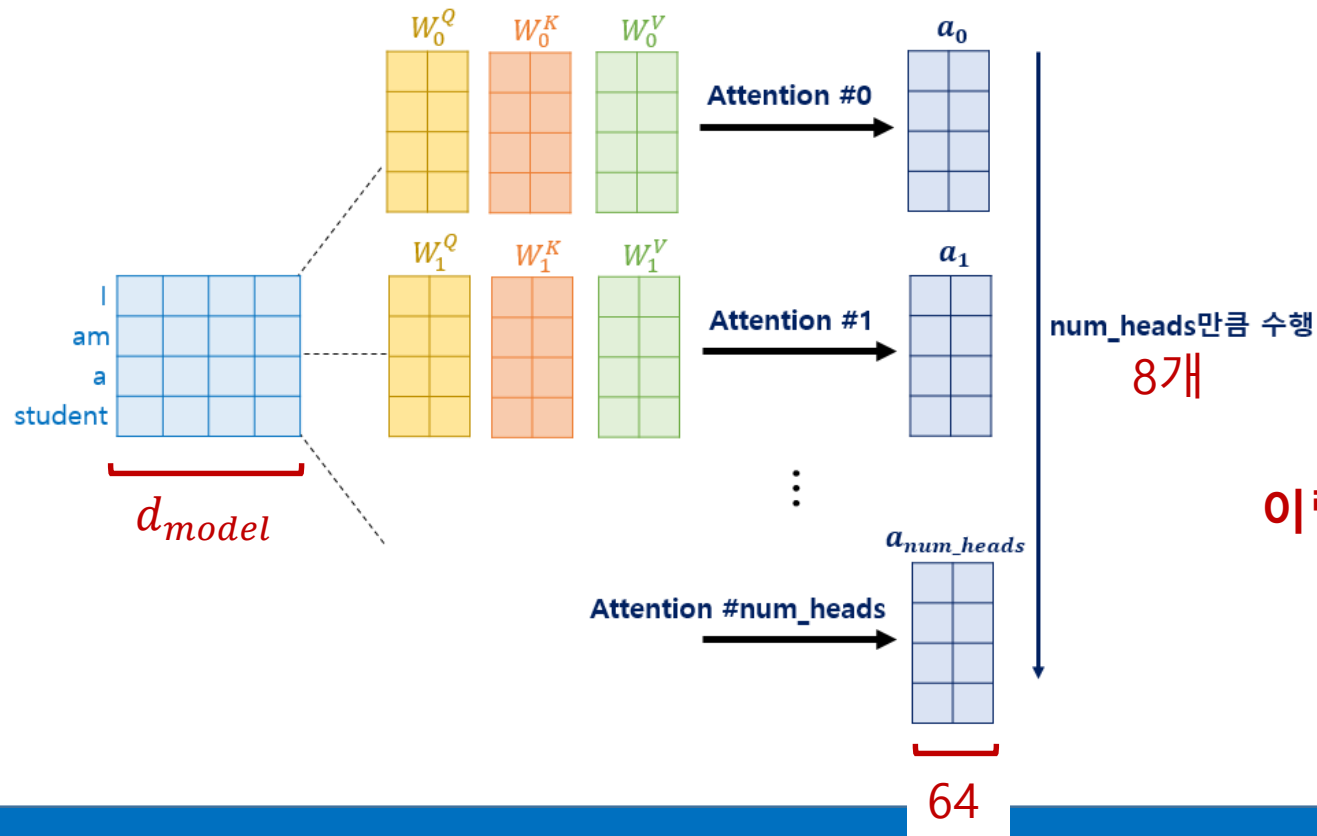
$$d_{model} = 512$$

$$\text{num\_heads} = 8$$

$$d_{model} / \text{num\_heads} = 64$$

# Multi-Head Attention

- 트랜스포머는 어텐션에 대해서 병렬적으로 수행한다는 특징을 가지고 있다.
- 이를 집단 지성, 다수의 머리를 이용한다고 하여 Multi-Head Attention이라고 부른다.



$$d_{model} = 512$$

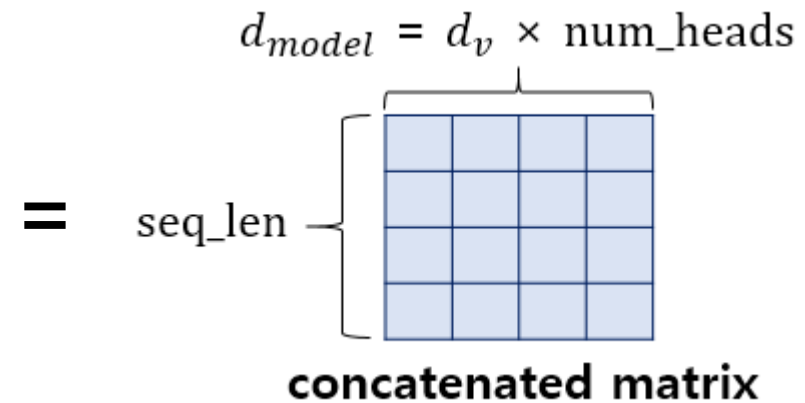
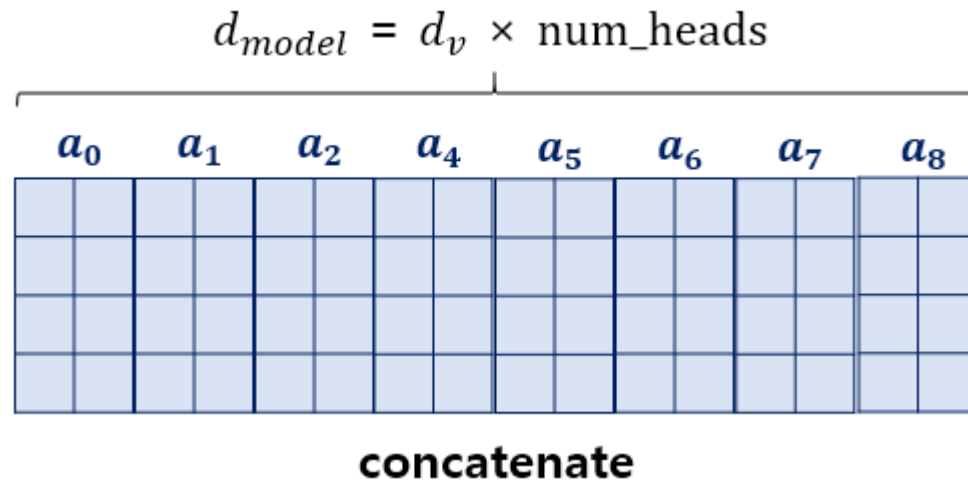
$$num\_heads = 8$$

$$d_{model} / num\_heads = 64$$

이렇게 얻은 8개의 결과를 다시 전부 연결한다.

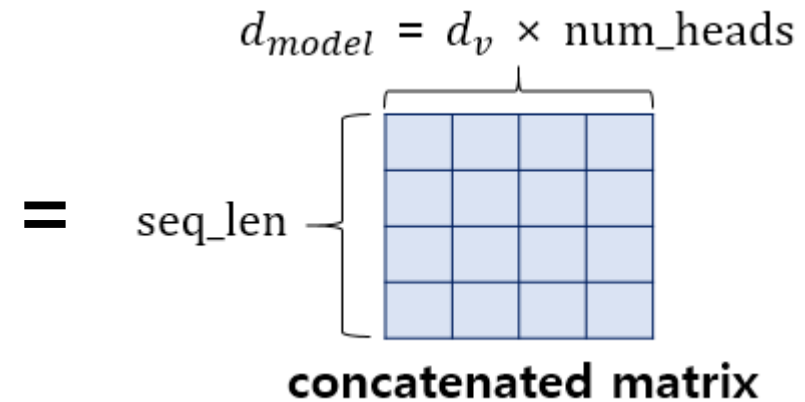
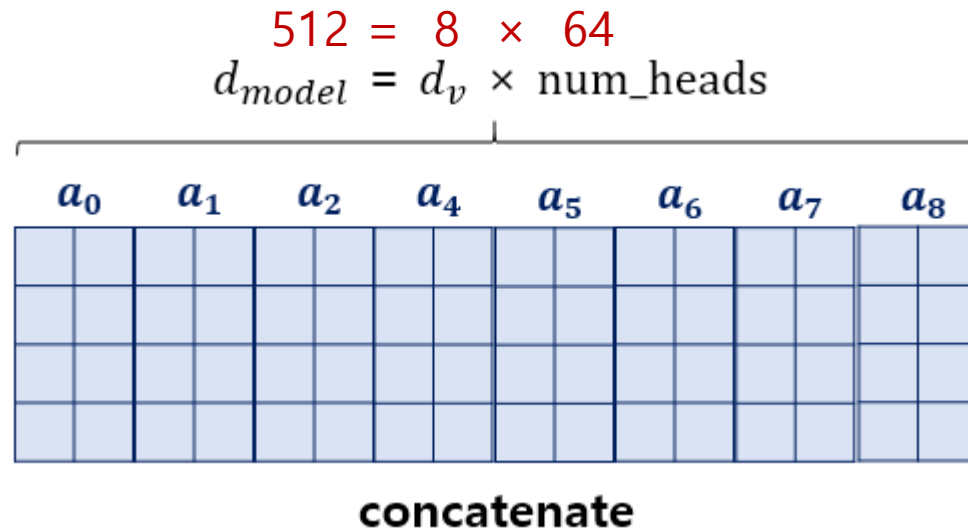
# Multi-Head Attention

- 트랜스포머는 어텐션에 대해서 병렬적으로 수행한다는 특징을 가지고 있다.
- 이를 집단 지성, 다수의 머리를 이용한다고 하여 Multi-Head Attention이라고 부른다.
- 병렬적으로 수행한 어텐션은 마지막에 전부 연결한다.



# Multi-Head Attention

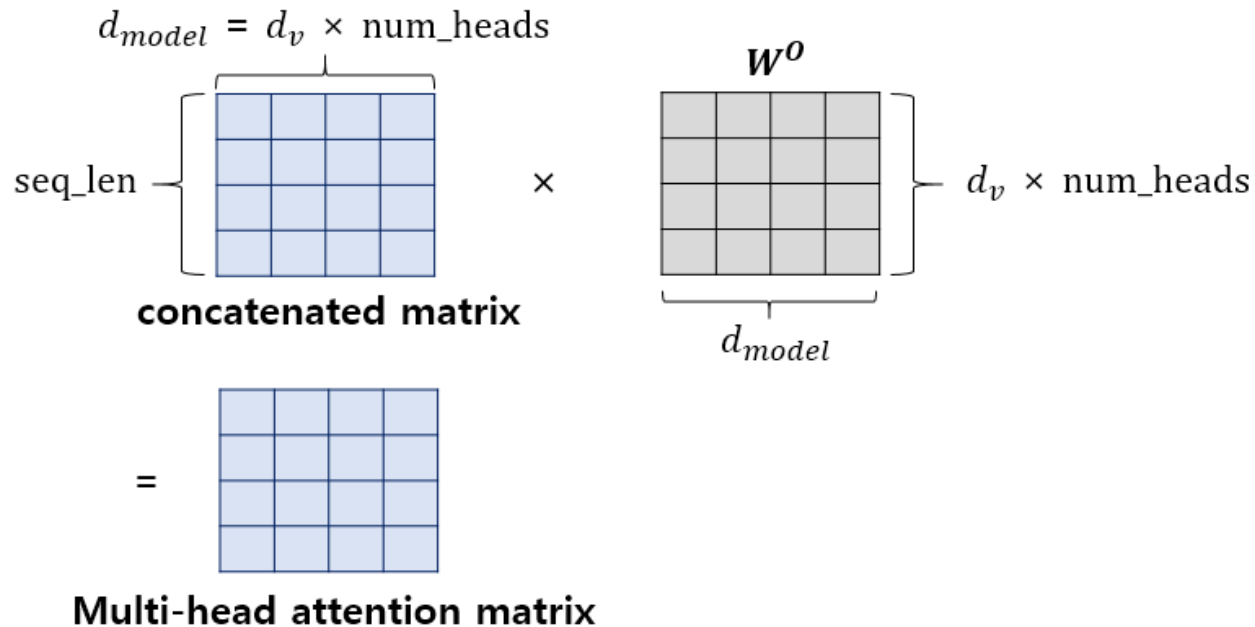
- 트랜스포머는 어텐션에 대해서 병렬적으로 수행한다는 특징을 가지고 있다.
- 이를 집단 지성, 다수의 머리를 이용한다고 하여 Multi-Head Attention이라고 부른다.
- 병렬적으로 수행한 어텐션은 마지막에 전부 연결한다.





# Multi-Head Attention

- 트랜스포머는 어텐션에 대해서 병렬적으로 수행한다는 특징을 가지고 있다.
- 이를 집단 지성, 다수의 머리를 이용한다고 하여 Multi-Head Attention이라고 부른다.
- 병렬적으로 수행한 어텐션은 마지막에 전부 연결한다.
- 연결한 행렬에 마지막으로 가중치 행렬을 한 번 더 곱해준다.



# Multi-Head Attention

- 논문에서는 지금까지의 과정을 아래의 수식으로 정리한다.
- $\text{Attention}(Q, K, V)$ 는 Attention Value를 얻는 어텐션 함수.
- $h$ 는 head 개수 /  $i$ 는  $i$ 번째 head를 의미한다.

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

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

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

# Pad Masking

- 어텐션을 수행할 때, Key에 해당하는 문장에 <pad>가 있는 경우
- 단어 <pad>는 사실 실제 단어가 아니라 문장의 길이를 맞춰주는 용도.

Diagram illustrating the calculation of an Attention Score Matrix:

Query Matrix  $Q$  (rows: *I*, *am*, *sam*, *<pad>*) is multiplied by the Transpose of the Key Matrix  $K^T$  (columns: *I*, *am*, *sam*, *<pad>*).

The result is the Attention Score Matrix (rows: *I*, *am*, *sam*, *<pad>*).

**Attention Score Matrix**

# Pad Masking

- 어텐션을 수행할 때, Key에 해당하는 문장에 <pad>가 있는 경우
- 단어 <pad>는 사실 실제 단어가 아니라 문장의 길이를 맞춰주는 용도.
- 단어 <pad>는 어텐션 스코어를 계산하는 일에는 불필요하므로 이를 마스킹(Masking)한다.

$$\begin{matrix} & Q \\ \begin{matrix} I \\ am \\ sam \\ <pad> \end{matrix} & \begin{bmatrix} & \\ & \\ & \\ & \end{bmatrix} \end{matrix} \times \begin{matrix} I \quad am \quad sam \quad <pad> \\ K^T \\ \begin{bmatrix} & & & \\ & & & \\ & & & \end{bmatrix} \end{matrix} = \begin{matrix} I \quad am \quad sam \quad <pad> \\ \begin{matrix} I \\ am \\ sam \\ <pad> \end{matrix} & \begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix} \end{matrix}$$

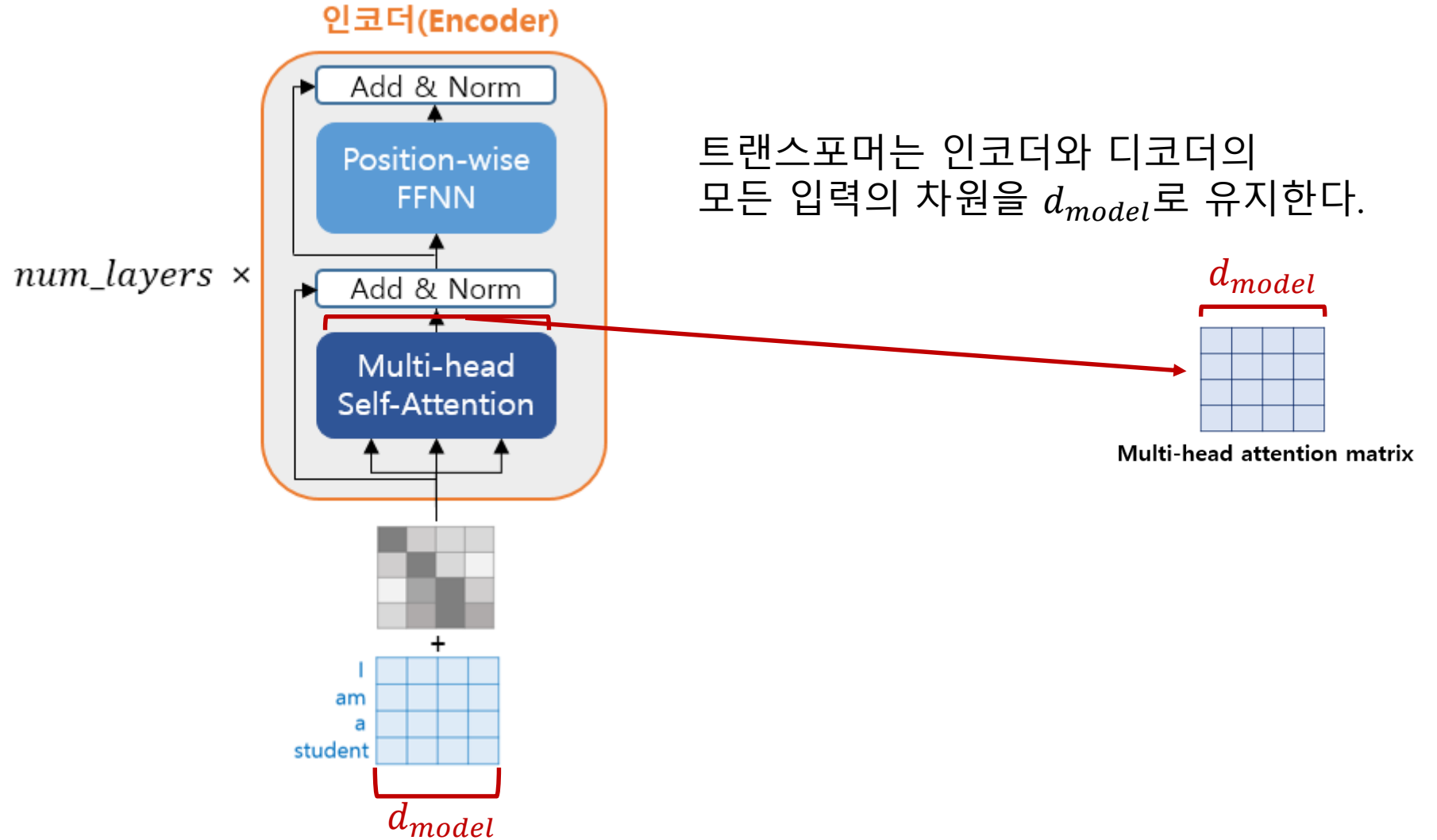
**Attention Score Matrix**

$$\begin{matrix} I \quad am \quad sam \quad <pad> \\ \begin{matrix} I \\ am \\ sam \\ <pad> \end{matrix} & \begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix} \end{matrix}$$

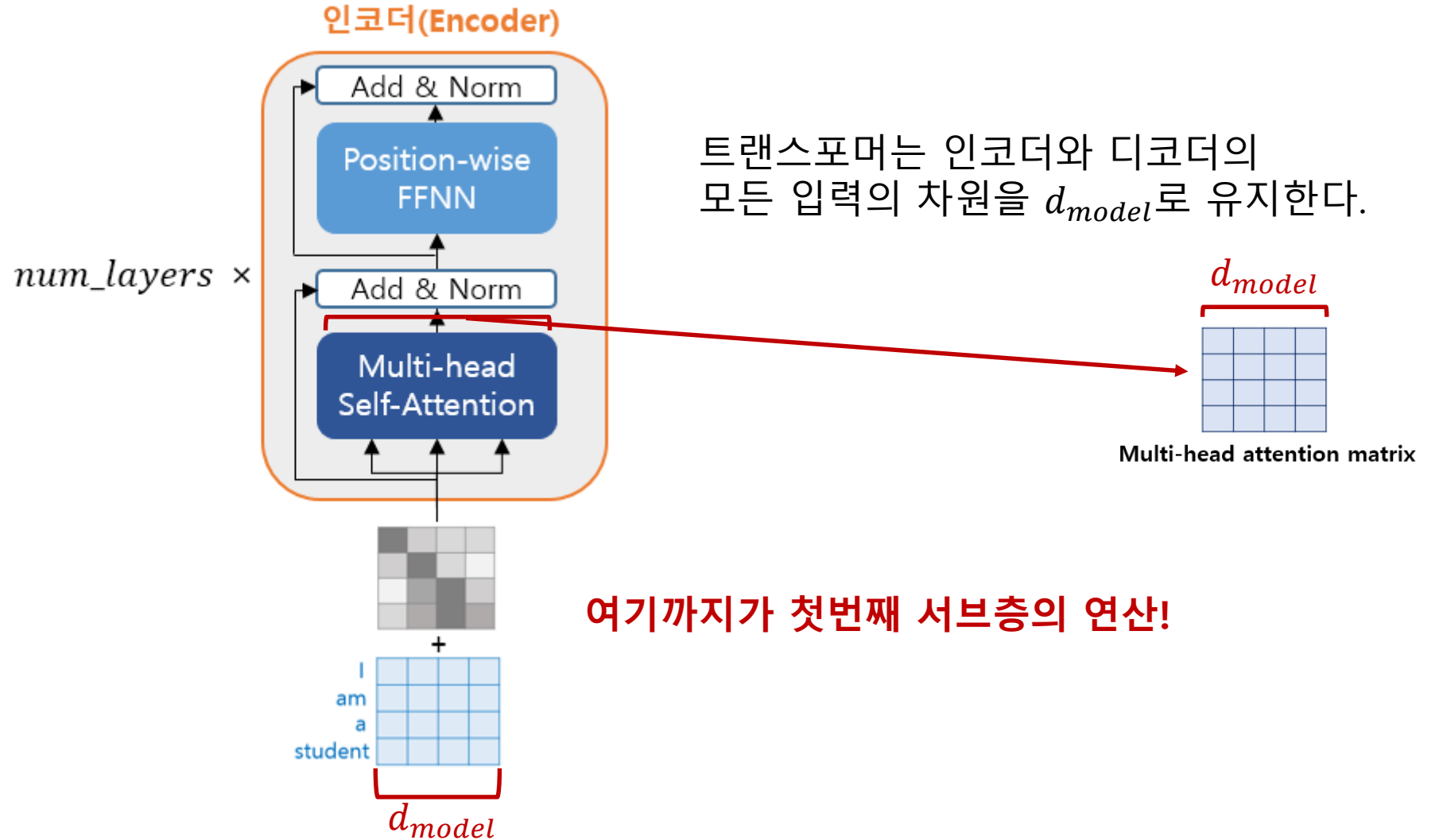
**Attention Score Matrix**

← 굉장히 작은 음수값

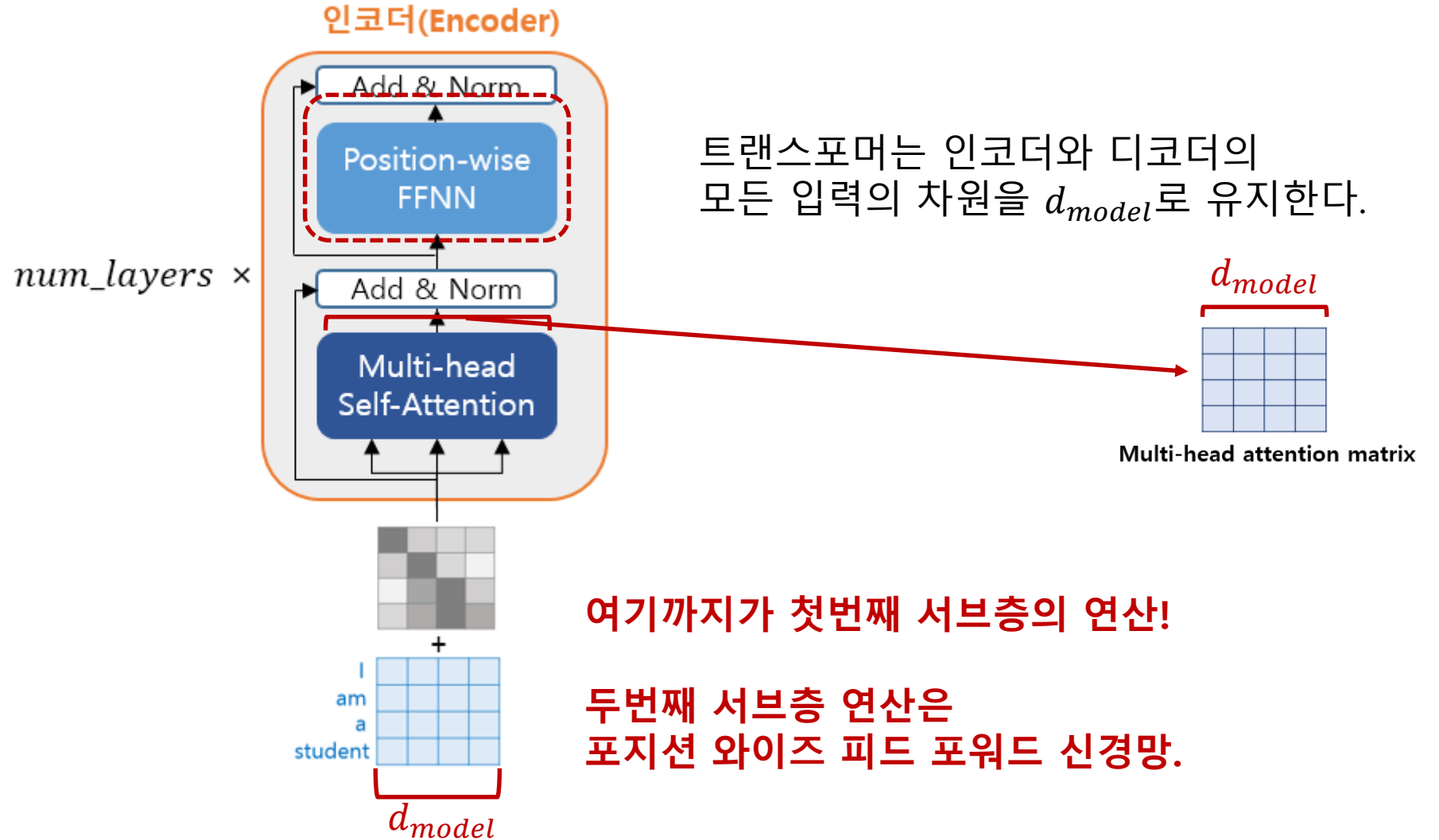
# Transformer Encoder



# Transformer Encoder

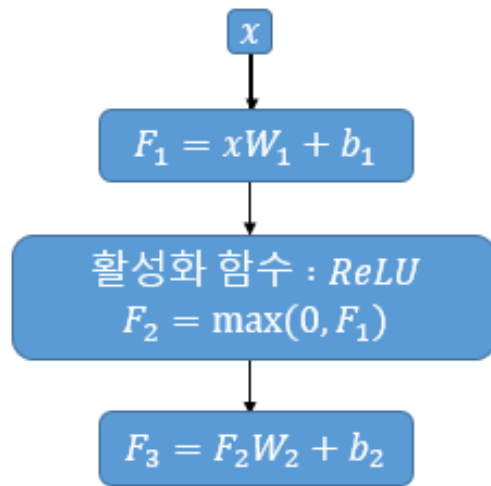


# Transformer Encoder



# Position-Wise Feed Forward Neural Network

- 포지션 와이즈 피드 포워드 신경망은 단순 피드 포워드 신경망이다.
- 은닉층에서는 활성화 함수로 ReLU 함수를 사용한다.
- $d_{ff}$ 는 피드 포워드 신경망의 은닉층의 크기로 논문에서는 2,048이 사용되었다.

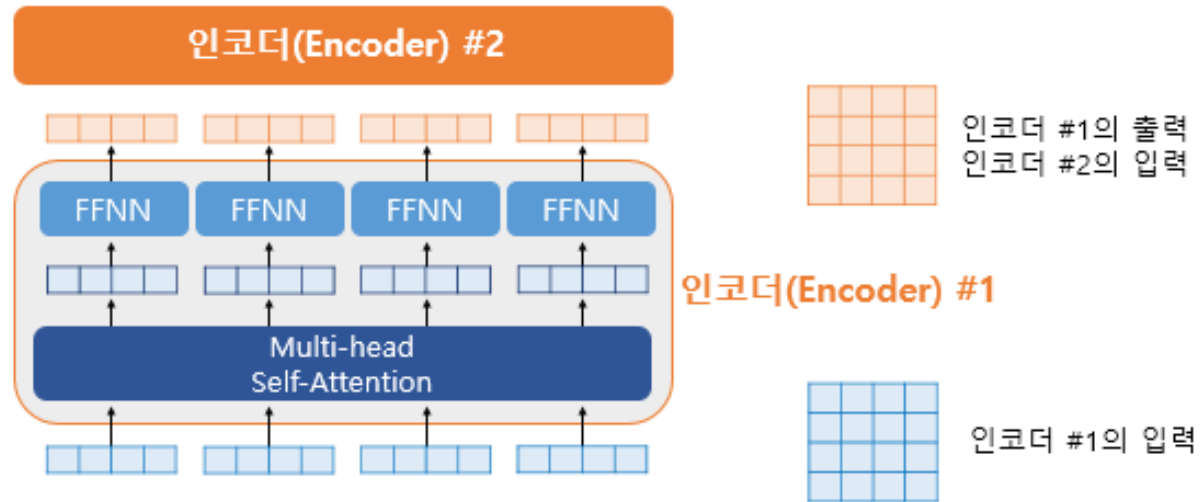


$$\begin{aligned}x &= (\text{seq\_len}, d_{\text{model}}) \\ W_1 &= (d_{\text{model}}, d_{ff}) \\ W_2 &= (d_{ff}, d_{\text{model}})\end{aligned}$$



# Position-Wise Feed Forward Neural Network

- 포지션 와이즈 피드 포워드 신경망은 단순 피드 포워드 신경망이다.
- 은닉층에서는 활성화 함수로 ReLU 함수를 사용한다.
- $d_{ff}$ 는 피드 포워드 신경망의 은닉층의 크기로 논문에서는 2,048이 사용되었다.



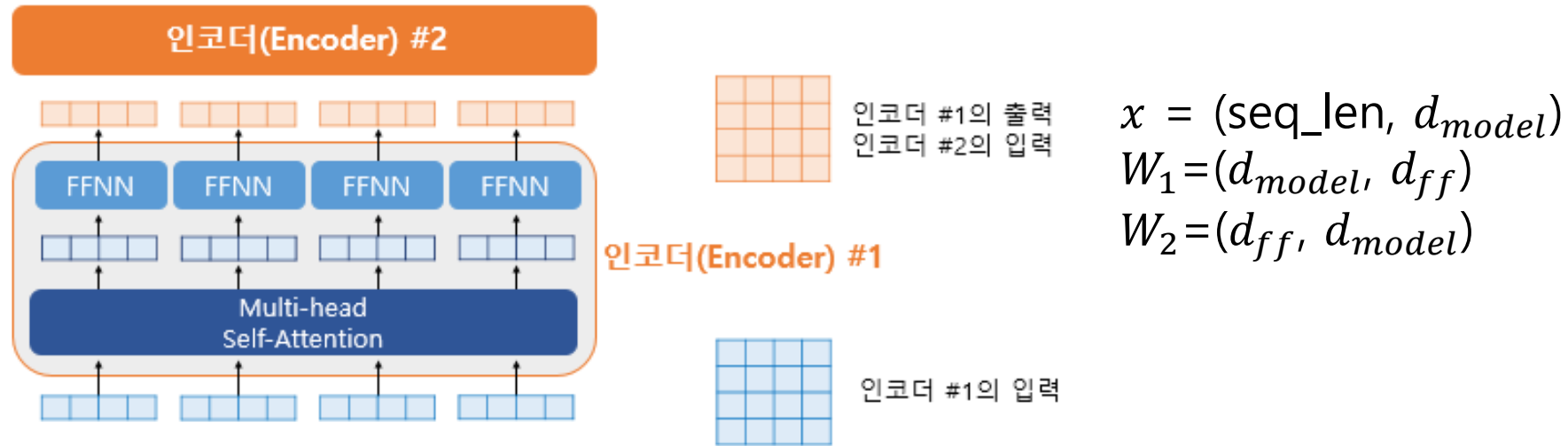
$$x = (\text{seq\_len}, d_{\text{model}})$$

$$W_1 = (d_{\text{model}}, d_{\text{ff}})$$

$$W_2 = (d_{\text{ff}}, d_{\text{model}})$$

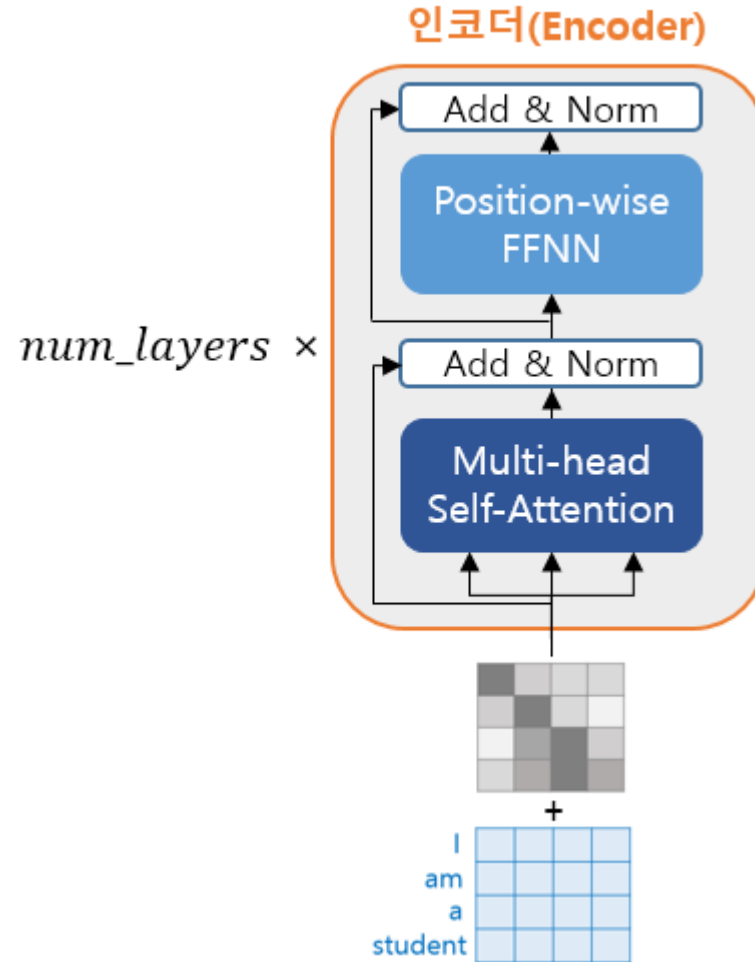
# Position-Wise Feed Forward Neural Network

- 포지션 와이즈 피드 포워드 신경망은 단순 피드 포워드 신경망이다.
- 은닉층에서는 활성화 함수로 ReLU 함수를 사용한다.
- $d_{ff}$ 는 피드 포워드 신경망의 은닉층의 크기로 논문에서는 2,048이 사용되었다.



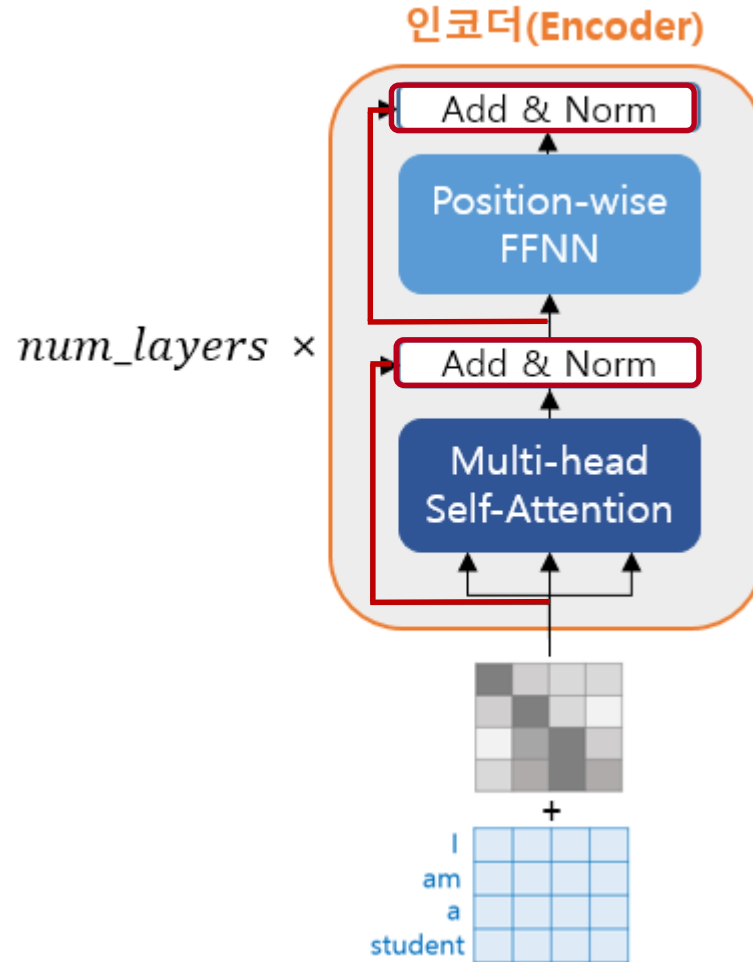
입력의 크기는 두번째 서브층을 통과한 후에도 여전히 보존된다.

# Transformer Encoder



아직 설명하지 않은 부분

# Transformer Encoder

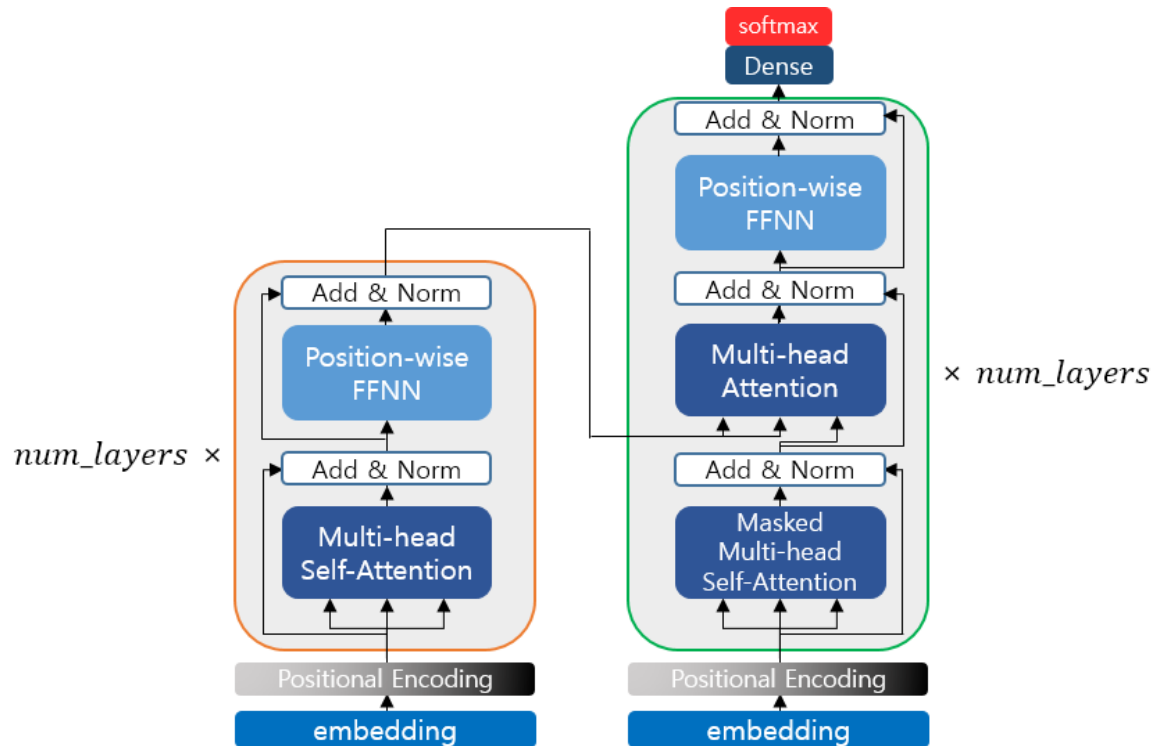


아직 설명하지 않은 부분

- Add = Residual Connection
- Norm = Layer Normalization

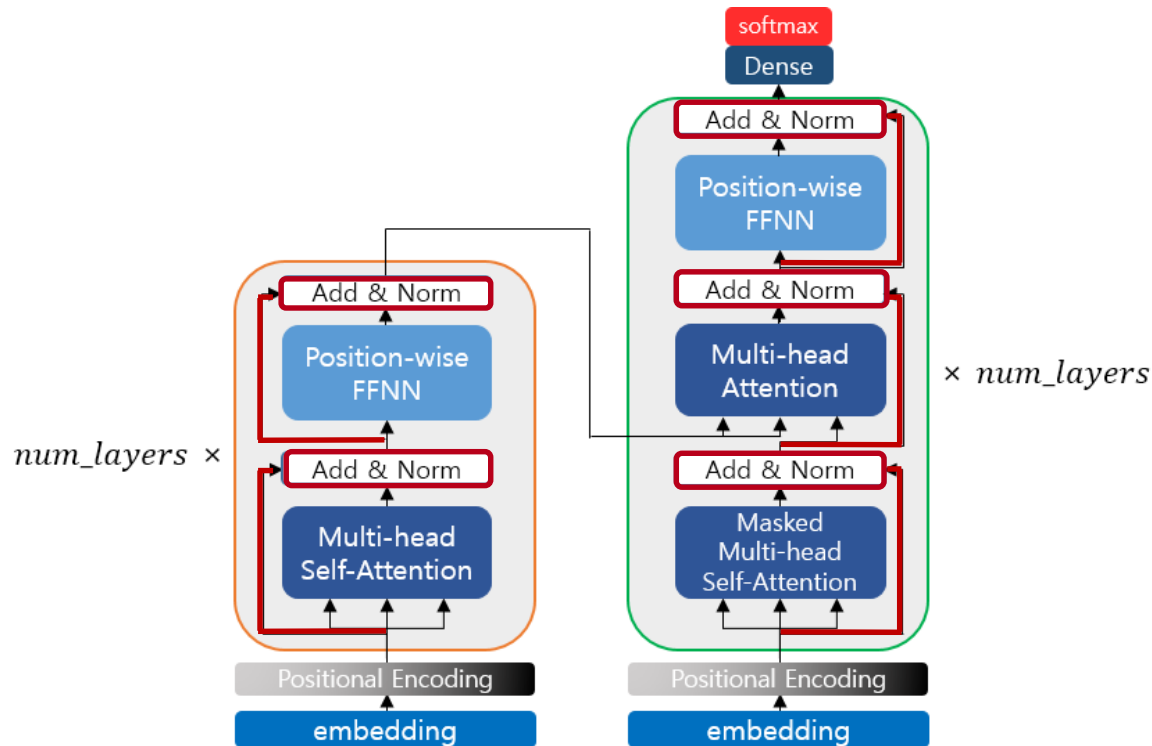
# Residual Connection & Layer Normalization

- 트랜스포머에서는 Residual Connection과 Layer Normalization이라는 두 가지 추가 테크닉을 사용.
- 아래 그림에서 각각 Add & Norm이라고 표현된 부분.
- 인코더, 디코더의 모든 서브층이 끝날 때마다 계속해서 적용.



# Residual Connection & Layer Normalization

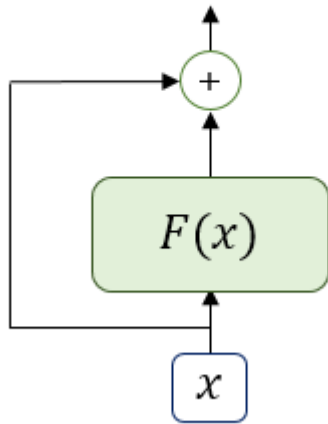
- 트랜스포머에서는 Residual Connection과 Layer Normalization이라는 두 가지 추가 테크닉을 사용.
- 아래 그림에서 각각 Add & Norm이라고 표현된 부분.
- 인코더, 디코더의 모든 서브층이 끝날 때마다 계속해서 적용.



# Add(Residual Connection)

- Residual Connection은 어떤 연산을 한 결과를 연산의 입력과 다시 더해주는 것을 말한다.
- 트랜스포머의 경우 서브층의 연산 결과를 입력과 다시 더해준다.

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

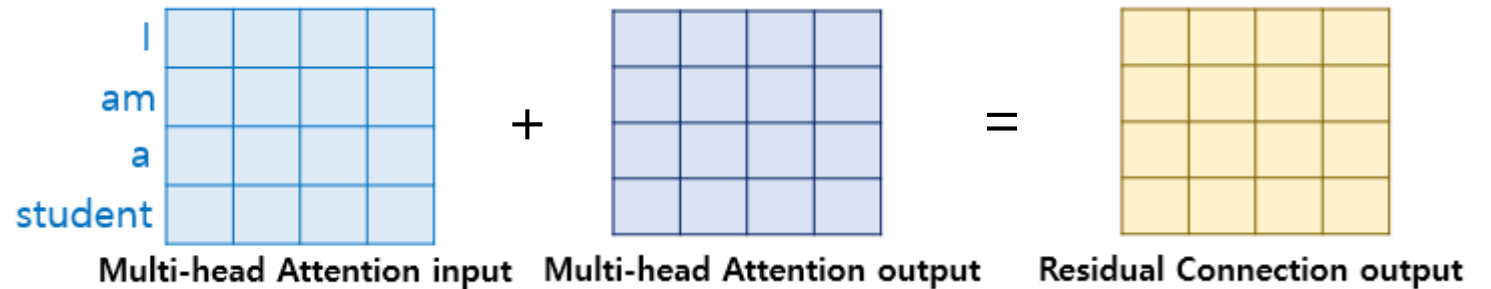
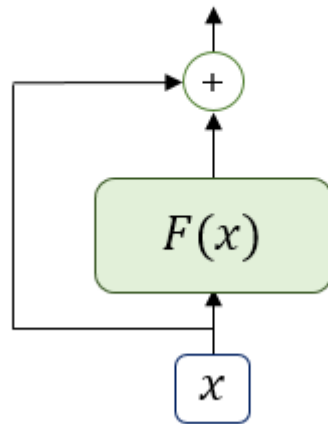


# Add(Residual Connection)

- Residual Connection은 어떤 연산을 한 결과를 연산의 입력과 다시 더해주는 것을 말한다.
- 트랜스포머의 경우 서브층의 연산 결과를 입력과 다시 더해준다.
- 만약 서브층이 멀티 헤드 어텐션이었다면?

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

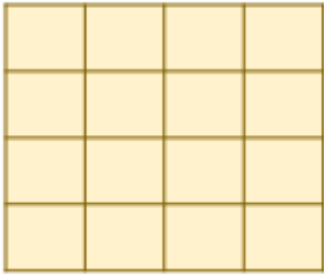
~~$$H(x) = x + F(x)$$~~





# Norm(Layer Normalization)

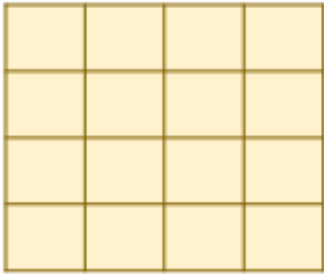
- Layer Normalization은 텐서의 마지막 차원에 대해서 평균과 분산을 구한다.



Residual Connection output

# Norm(Layer Normalization)

- Layer Normalization은 텐서의 마지막 차원에 대해서 평균과 분산을 구한다.

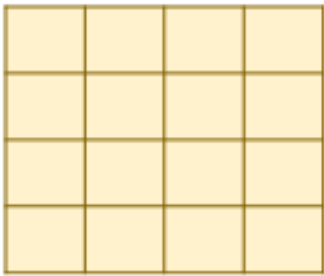


**Residual Connection output**

(batch\_size, seq\_len, d\_model)

# Norm(Layer Normalization)

- Layer Normalization은 텐서의 **마지막 차원**에 대해서 평균과 분산을 구한다.

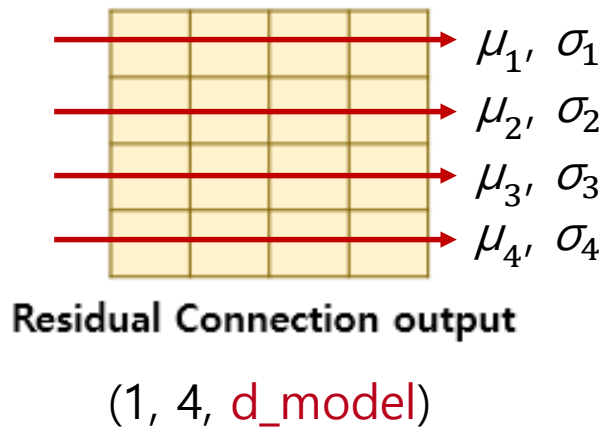


Residual Connection output

(1, 4, **d\_model**)

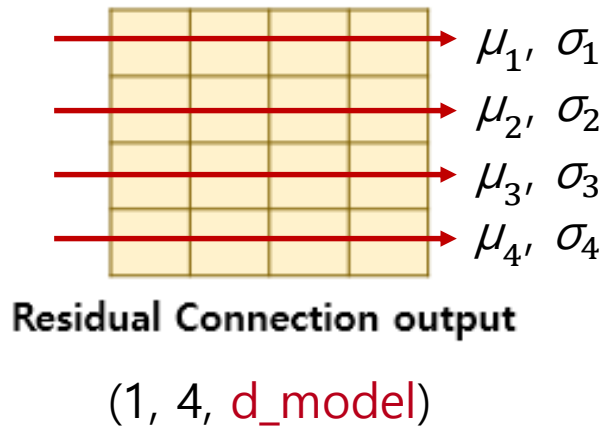
# Norm(Layer Normalization)

- Layer Normalization은 텐서의 마지막 차원에 대해서 평균과 분산을 구한다.
- $\mu$ 와  $\sigma$ 는 각각 평균과 표준편차를 의미하는 기호이다.



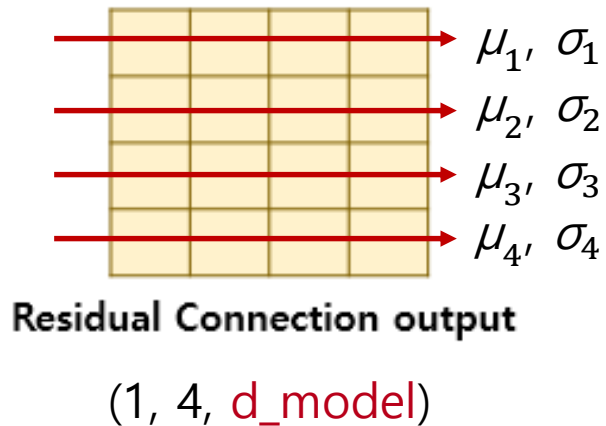
# Norm(Layer Normalization)

- Layer Normalization은 텐서의 마지막 차원에 대해서 평균과 분산을 구한다.
- $\mu$ 와  $\sigma$ 는 각각 평균과 표준편차를 의미하는 기호이다.
- 여기서는 이 평균과 분산을 이용해서 화살표 방향의 각 벡터의 값을 정규화하는 것이 목표이다.



# Norm(Layer Normalization)

- Layer Normalization은 텐서의 마지막 차원에 대해서 평균과 분산을 구한다.
- $\mu$ 와  $\sigma$ 는 각각 평균과 표준편차를 의미하는 기호이다.
- 여기서는 이 평균과 분산을 이용해서 화살표 방향의 각 벡터의 값을 정규화하는 것이 목표이다.
- $\gamma$ (감마)와  $\beta$ (베타)라는 변수를 준비한다. 단, 이들의 초기값은 각각 1과 0이다.

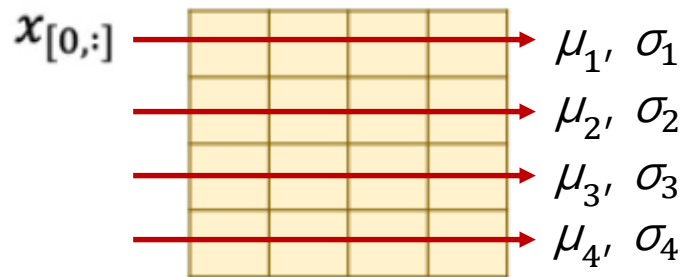


$$\gamma \quad \begin{array}{|c|c|c|c|} \hline 1 & 1 & 1 & 1 \\ \hline \end{array}$$

$$\beta \quad \begin{array}{|c|c|c|c|} \hline 0 & 0 & 0 & 0 \\ \hline \end{array}$$

# Norm(Layer Normalization)

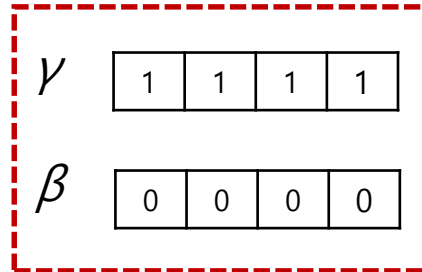
- Layer Normalization은 텐서의 마지막 차원에 대해서 평균과 분산을 구한다.
- $\mu$ 와  $\sigma$ 는 각각 평균과 표준편차를 의미하는 기호이다.
- 여기서는 이 평균과 분산을 이용해서 화살표 방향의 각 벡터의 값을 정규화하는 것이 목표이다.
- $\gamma$ (감마)와  $\beta$ (베타)라는 변수를 준비한다. 단, 이들의 초기값은 각각 1과 0이다.



Residual Connection output

(1, 4, d\_model)

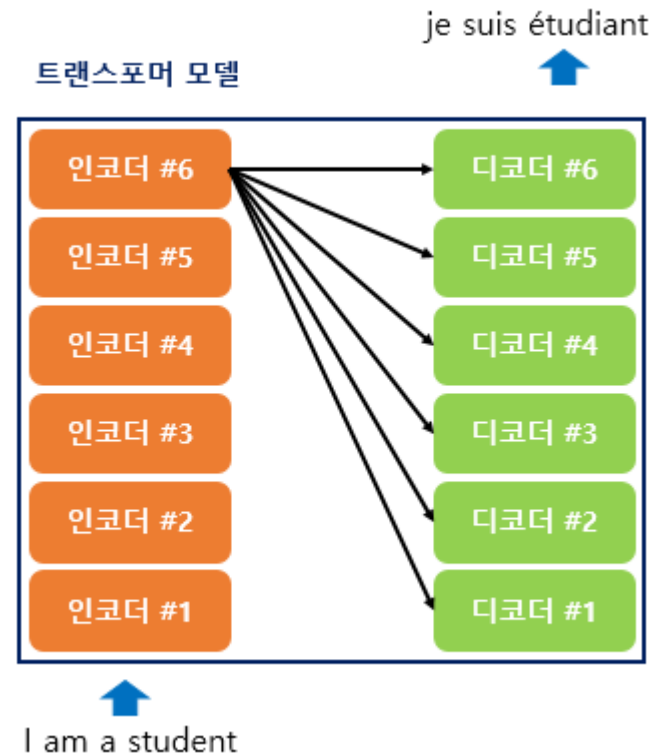
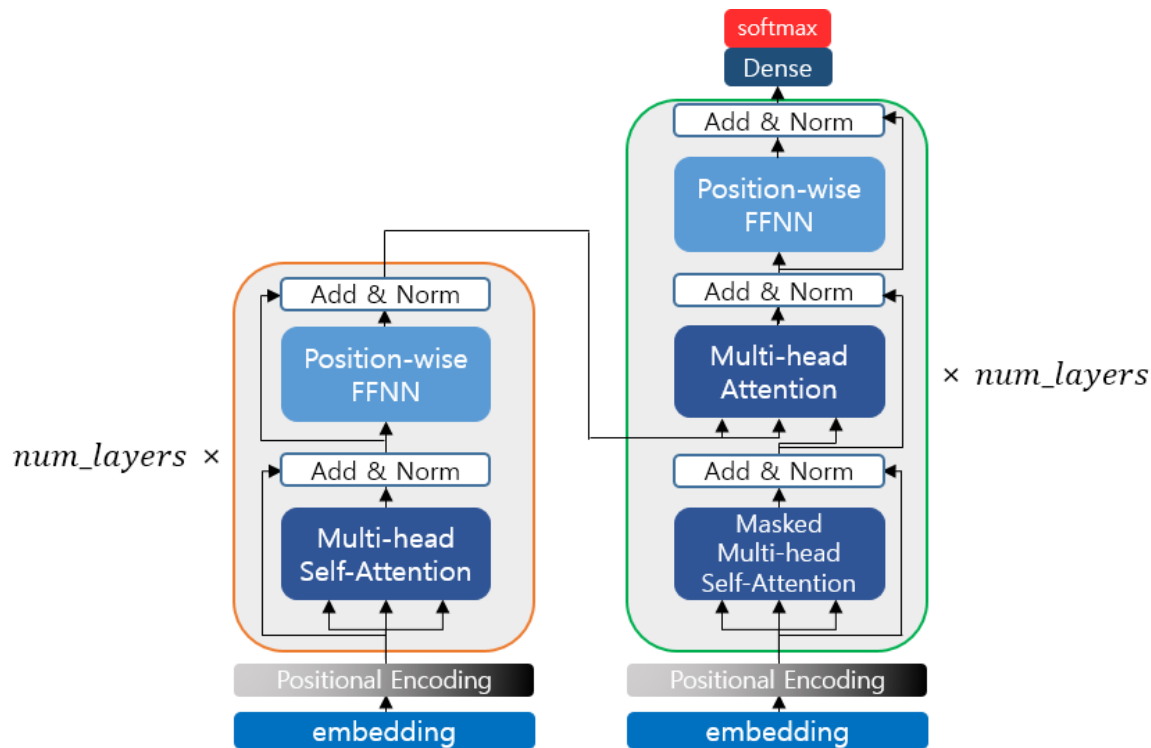
학습 대상



$$ln_{[0,:]} = \gamma \frac{x_{[0,:]} - \mu}{\sigma} + \beta$$

# From Encoder to Decoder

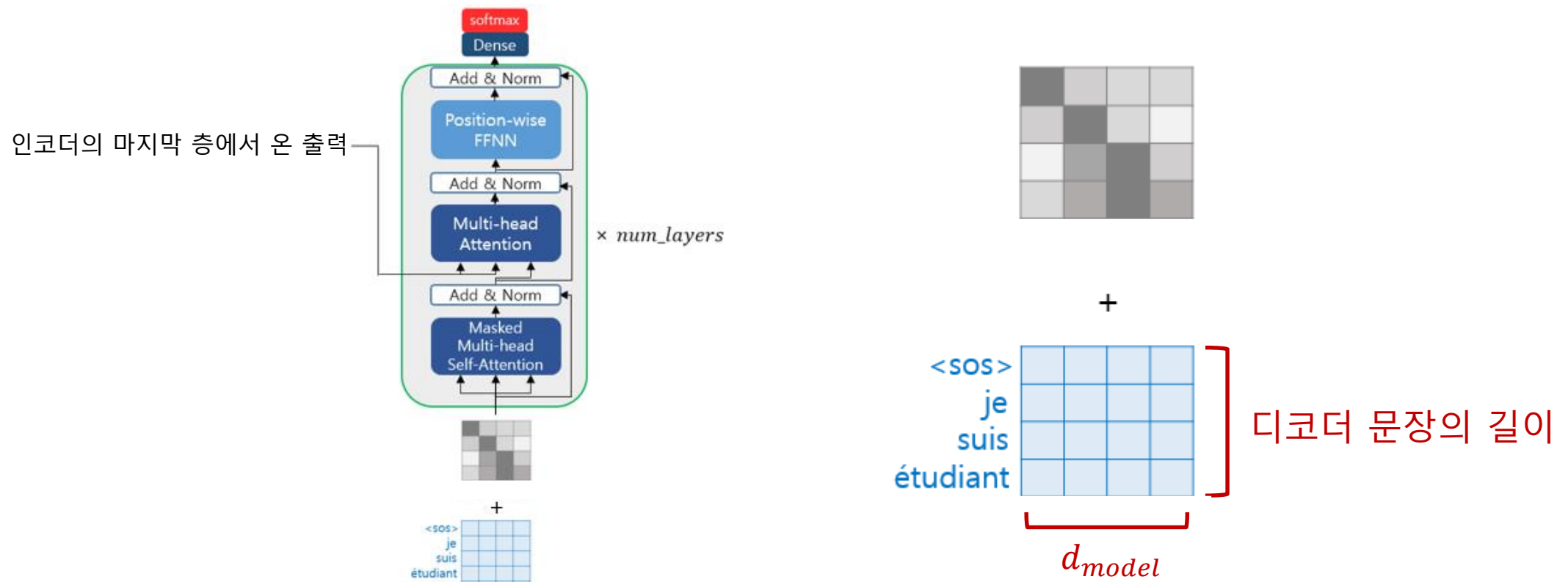
- 총  $\text{num\_layers}$ (논문에서는 6)회의 인코더 연산을 한 후, 마지막 인코더의 출력이 디코더로 전달된다.
- 디코더는 인코더의 출력을 전달받아서 다시  $\text{num\_layers}$ (논문에서는 6)회의 연산을 한다.





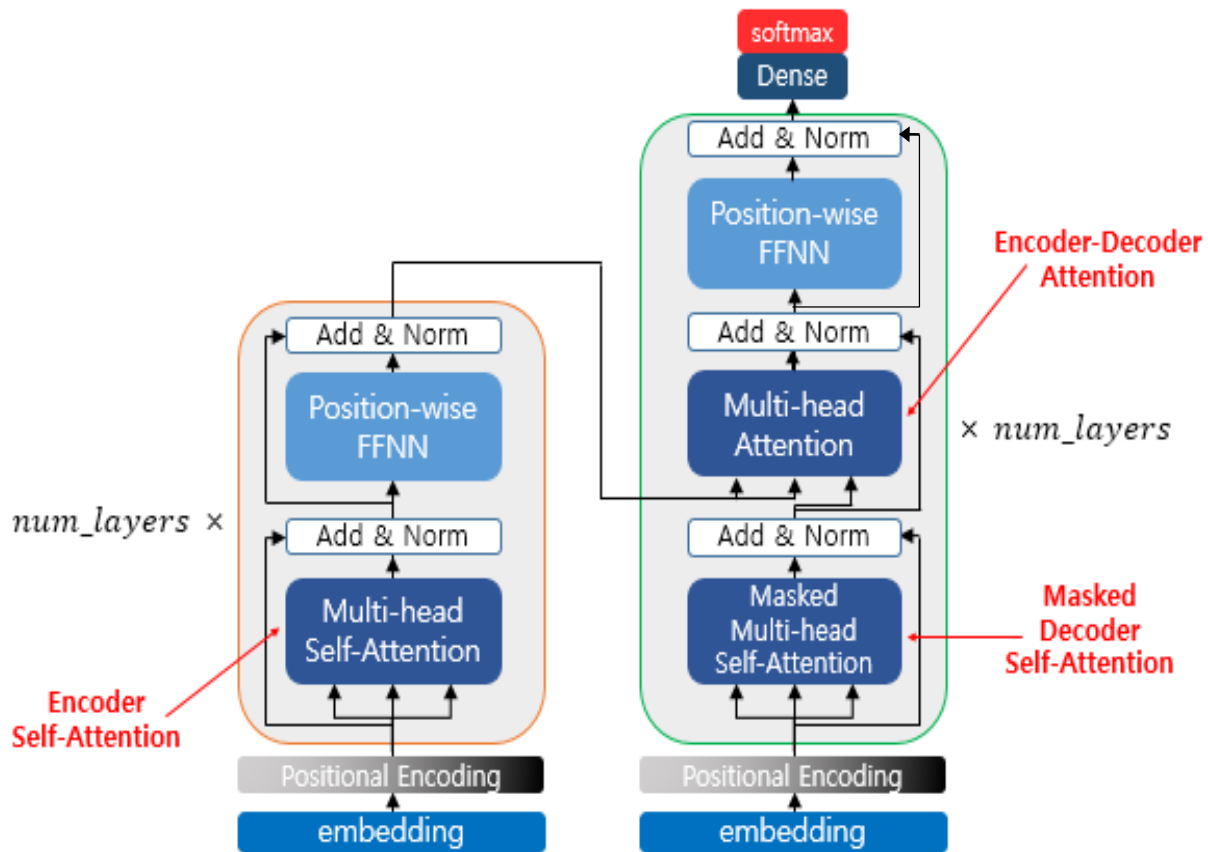
# Transformer Decoder : Positional Encoding

- 트랜스포머의 디코더 또한 포지셔널 인코딩을 사용한다.
- 디코더의 입력인 문장 행렬에서는 (seq2seq와 마찬가지로) 시작 토큰과 종료 토큰이 있다.

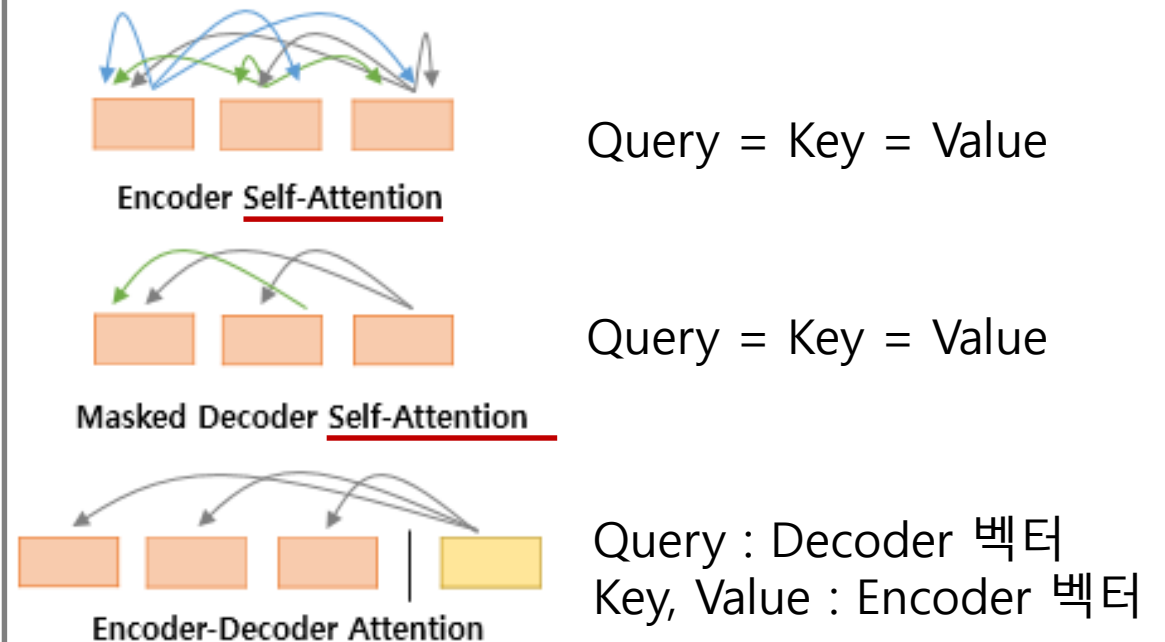


# Transformer : Attention Mechanism

- 트랜스포머는 총 세 종류의 어텐션이 존재.

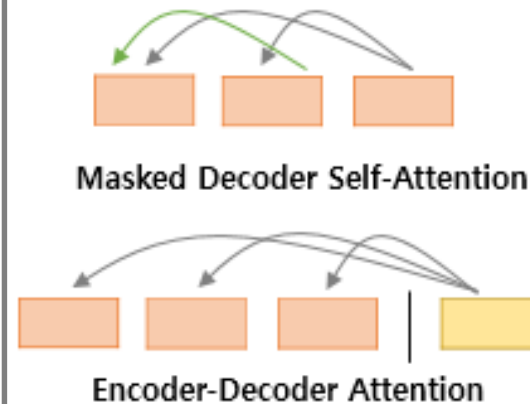
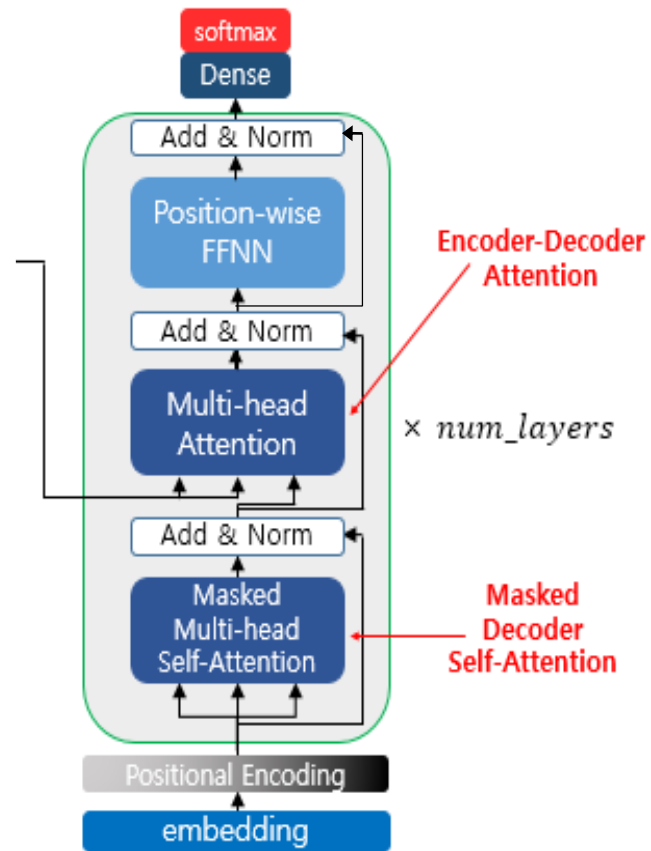


Q = K = V인 경우에는  
Self-Attention이라고 부른다.



# Transformer : Attention Mechanism

- 트랜스포머 디코더에는 총 두 종류의 어텐션이 존재.

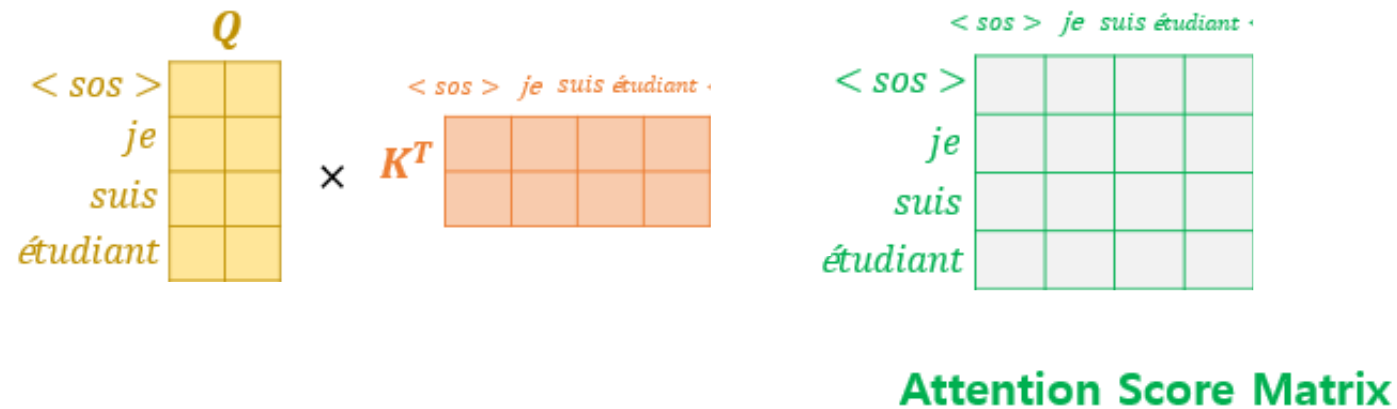


Query = Key = Value

Query : Decoder 벡터  
Key, Value : Encoder 벡터

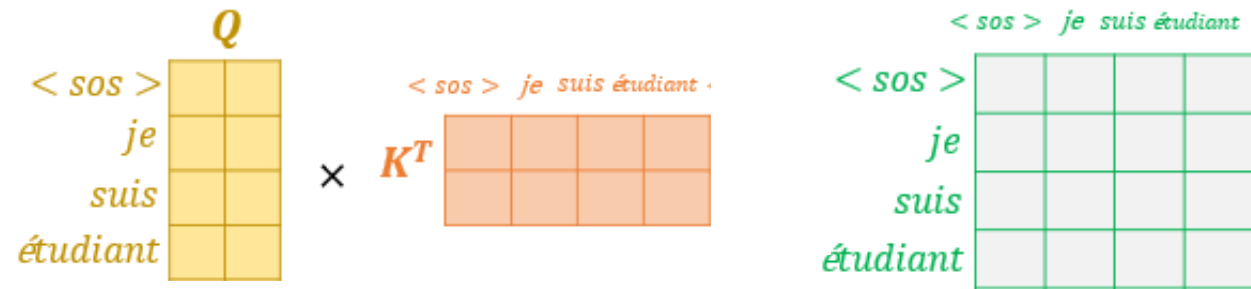
# Transformer Decoder: Self-Attention

- 트랜스포머 디코더의 Masked Self-Attention은 근본적으로 Self-Attention과 동일하다.

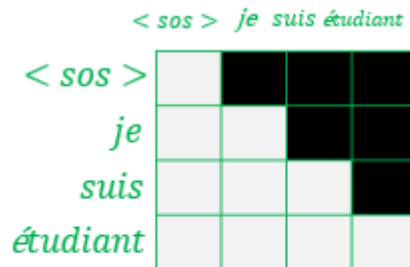


# Transformer Decoder: Self-Attention

- 트랜스포머 디코더의 Masked Self-Attention은 근본적으로 Self-Attention과 동일하다.
- 하지만 어텐션 스코어 매트릭스에 직각 삼각형의 마스킹을 해준다.



Attention Score Matrix

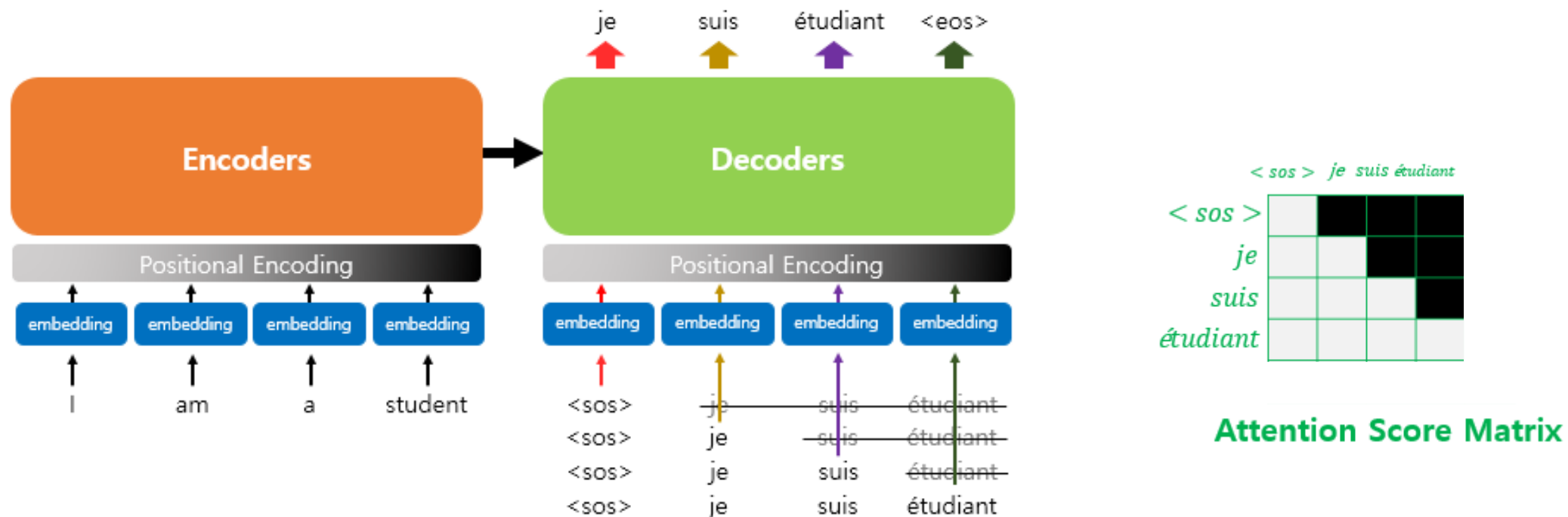


Attention Score Matrix

왜 해주는걸까?

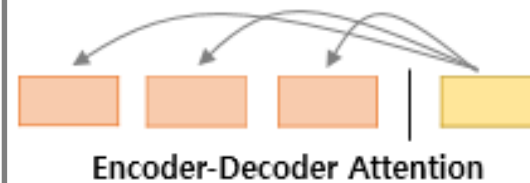
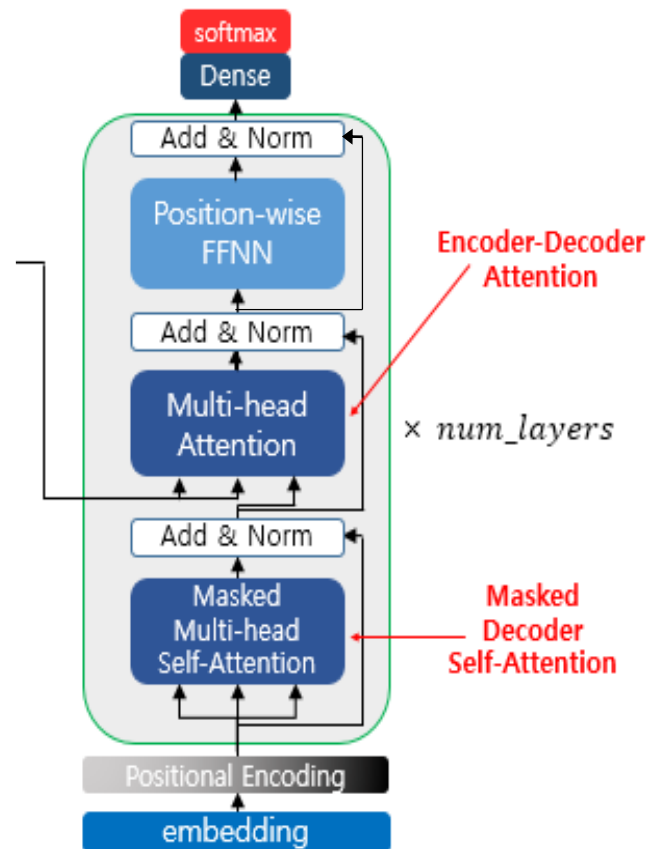
# Transformer Decoder: Self-Attention

- 트랜스포머 디코더는 훈련 과정에서 실제 예측할 문장 행렬을 입력으로 넣어준다.
- 훈련 과정에서 정답을 알려주는 이 과정을 교사 강요(Teacher Forcing)라 부른다.
- 그런데 셀프-어텐션을 할 때, 다음 단어를 이미 알고있다는 가정 하에 어텐션을 해서는 안된다.



# Transformer Decoder: Encoder-Decoder Attention

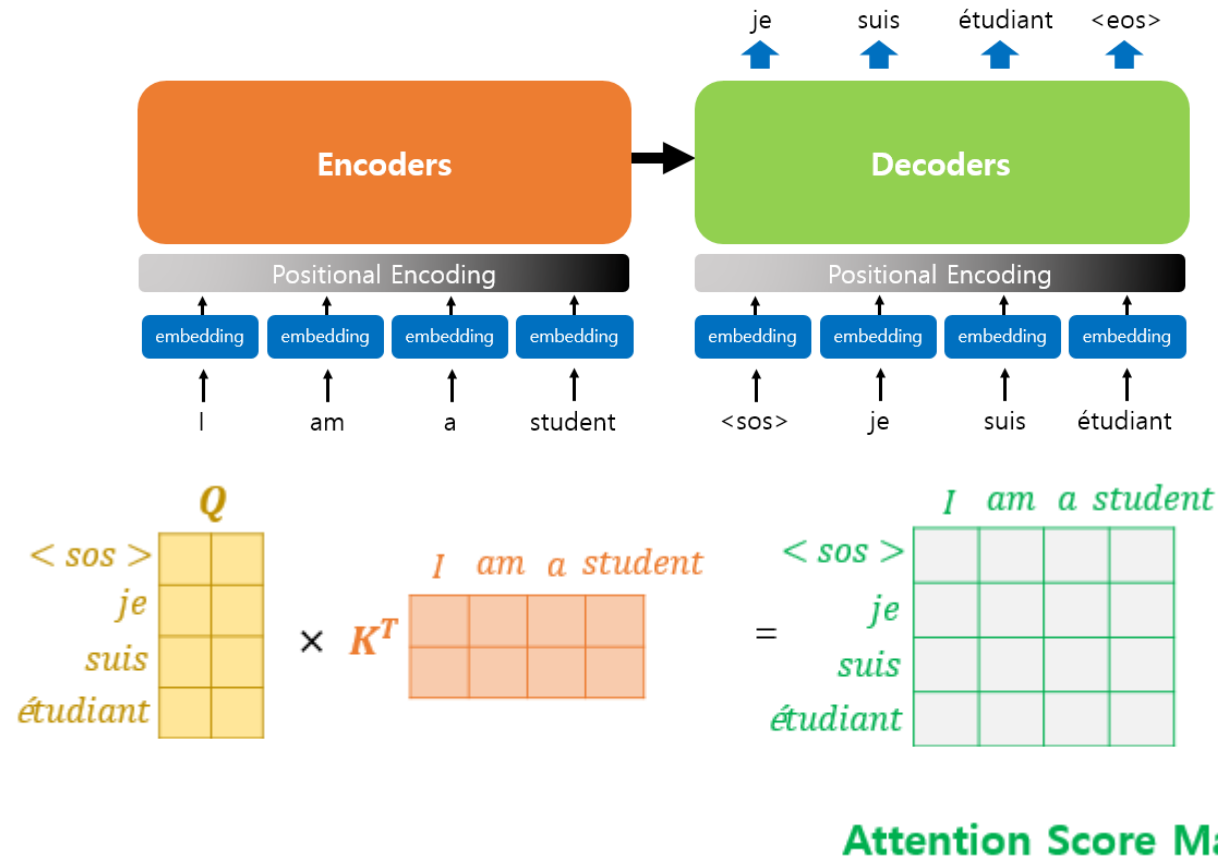
- 트랜스포머 디코더에서 이루어지는 인코더-디코더 어텐션은 Q와 K, V는 다르다.
- Q는 디코더의 벡터인 반면, K, V는 인코더의 벡터이다.



Query : Decoder 벡터  
Key, Value : Encoder 벡터

# Transformer Decoder: Encoder-Decoder Attention

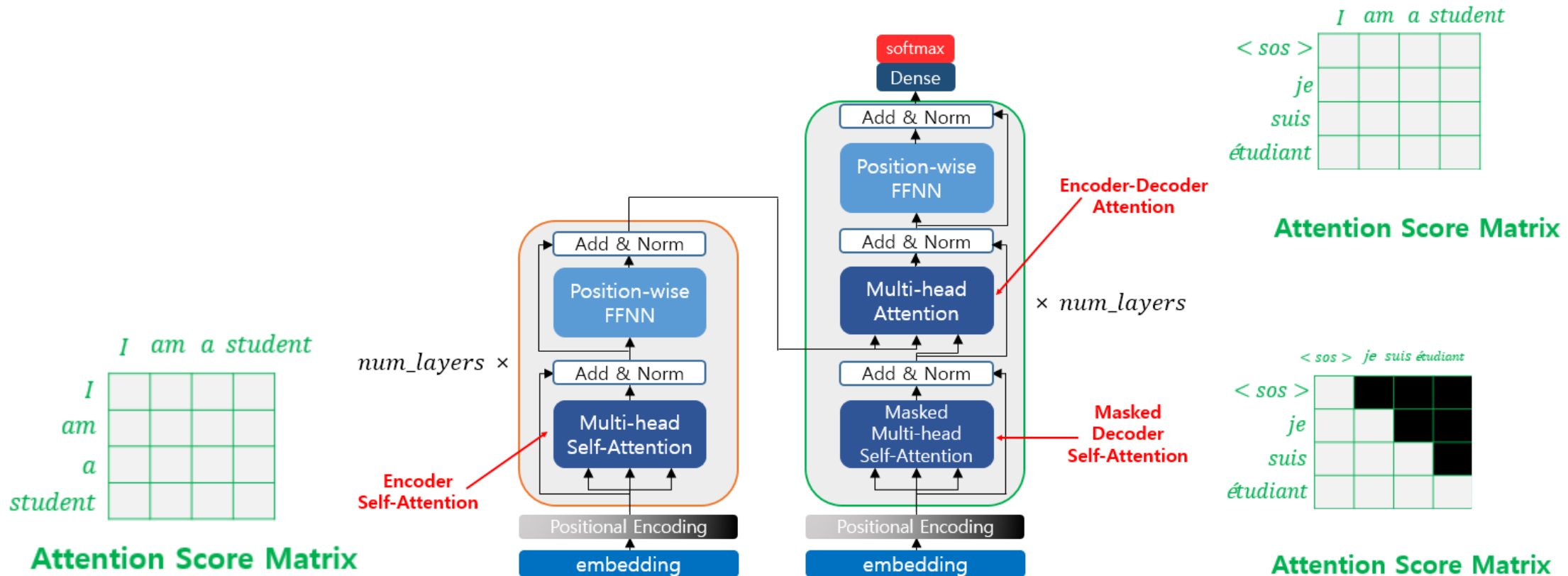
- 트랜스포머 디코더에서 이루어지는 인코더-디코더 어텐션은 Q와 K, V는 다르다.
- Q는 디코더의 벡터인 반면, K, V는 인코더의 벡터이다.





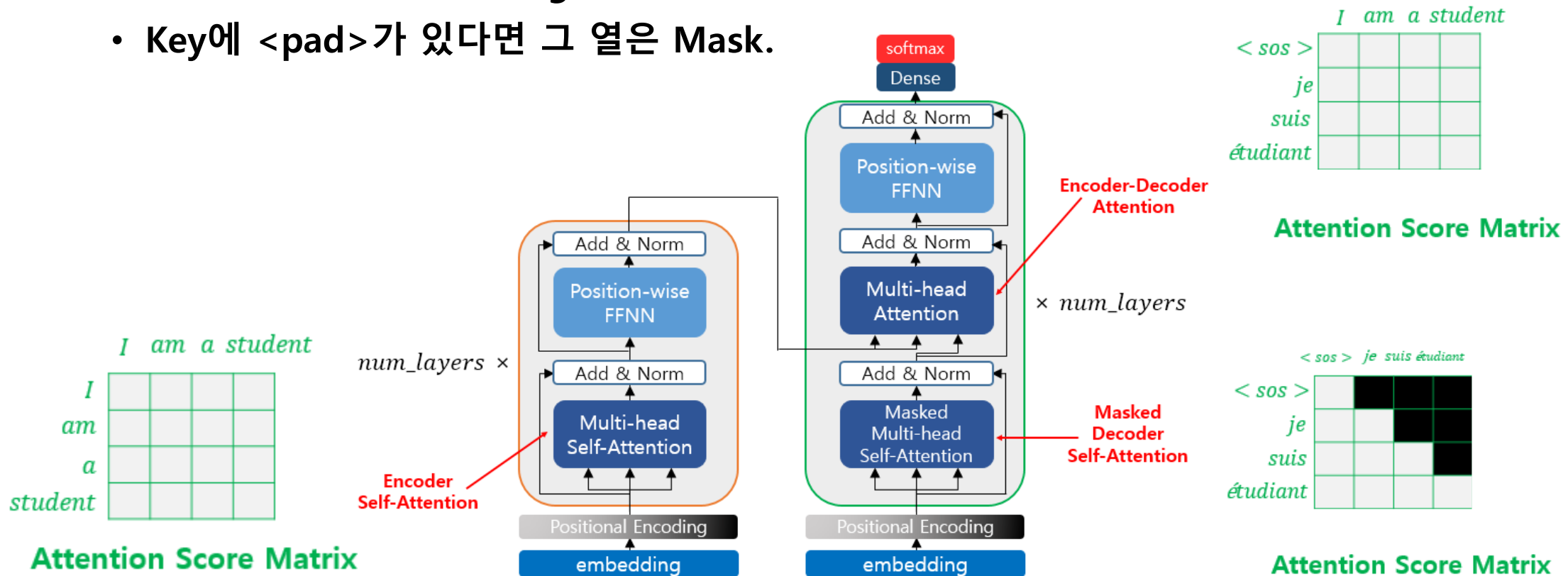
# Transformer : Attention Mechanism

- 트랜스포머는 총 세 종류의 어텐션이 존재.



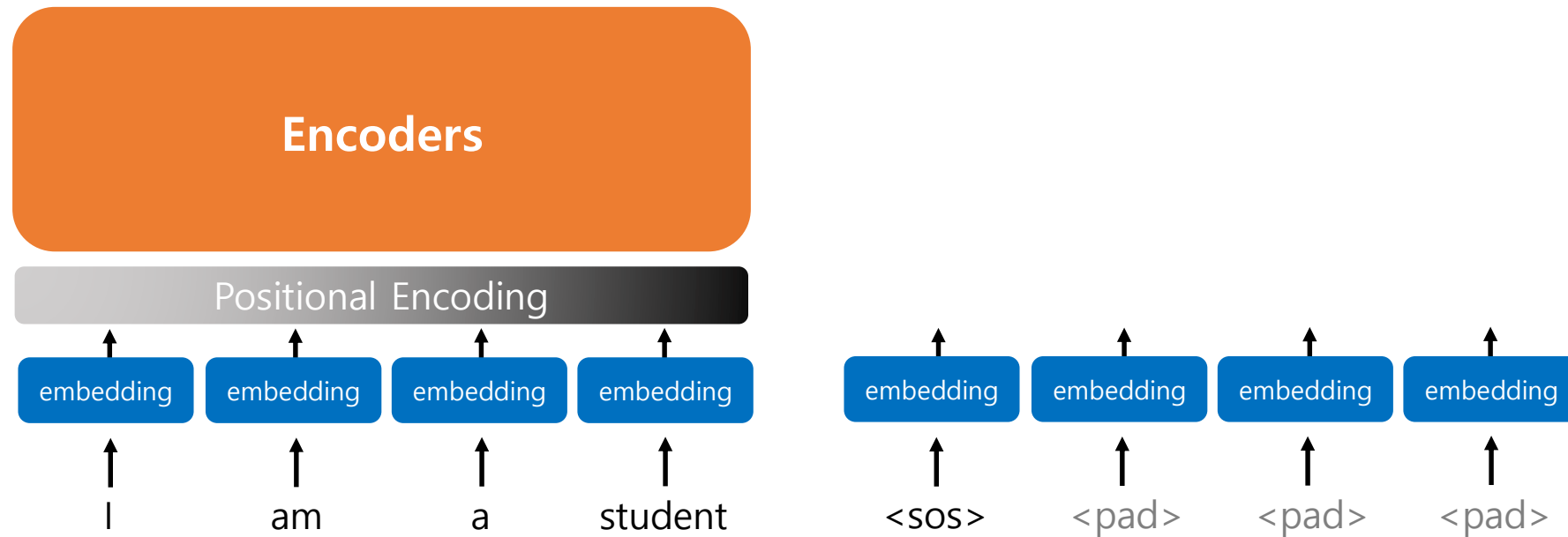
# Transformer : Attention Mechanism

- 트랜스포머는 총 세 종류의 어텐션이 존재.
- 주의할 점은 Pad Masking은 모든 어텐션에서 적용된다.
- Key에 <pad>가 있다면 그 열은 Mask.



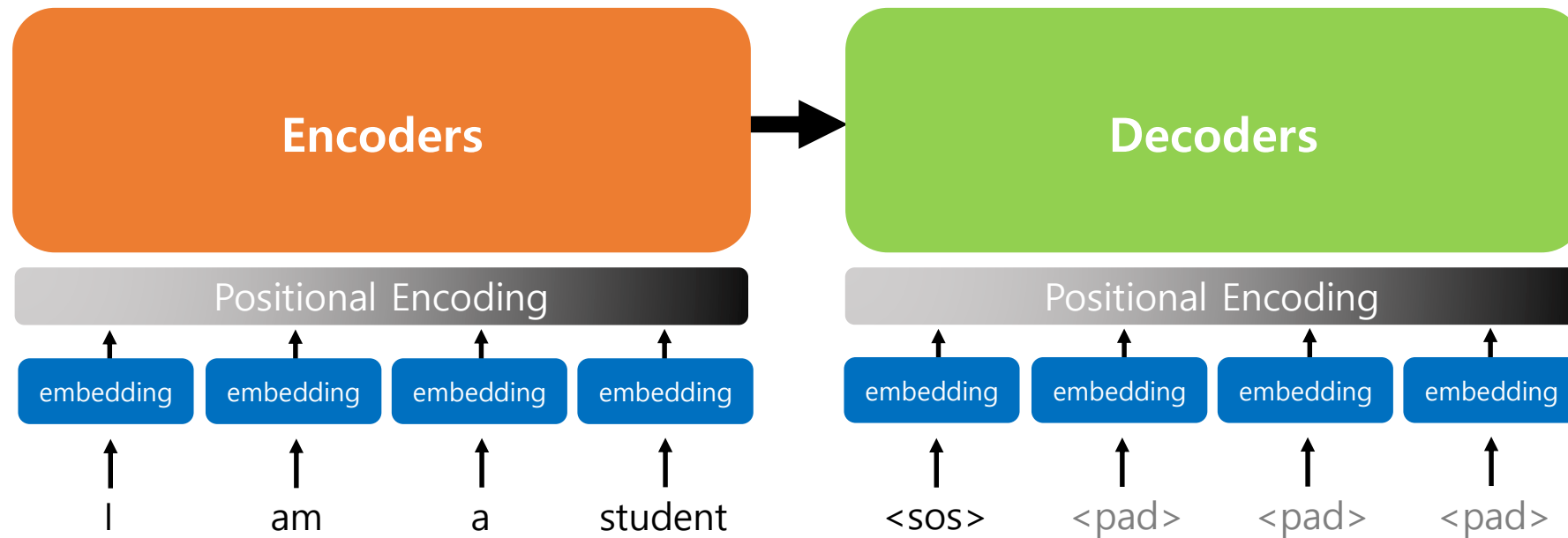
# Transformer : Inference

- 테스트 단계(실제 번역기나 챗봇이 구동될 때)에는 디코더가 단어를 1개씩 생성해내야 한다.



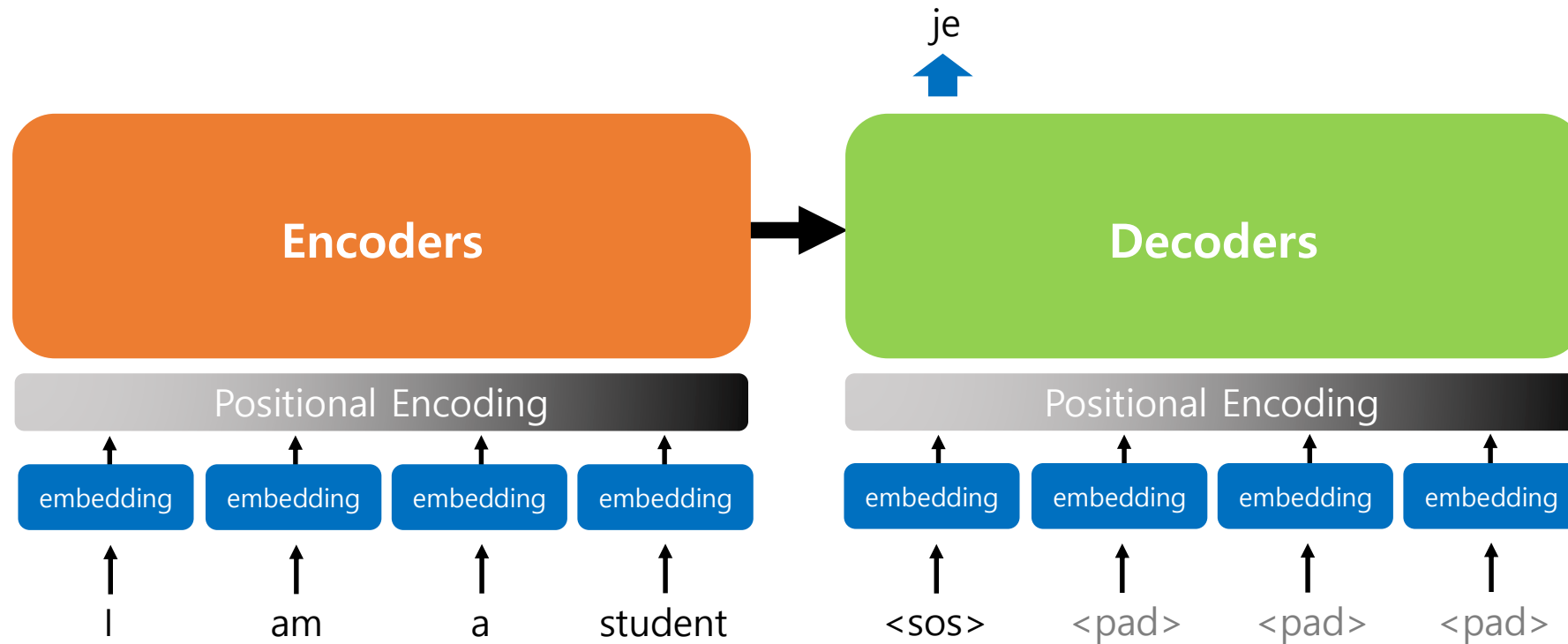
# Transformer : Inference

- 테스트 단계(실제 번역기나 챗봇이 구동될 때)에는 디코더가 단어를 1개씩 생성해내야 한다.



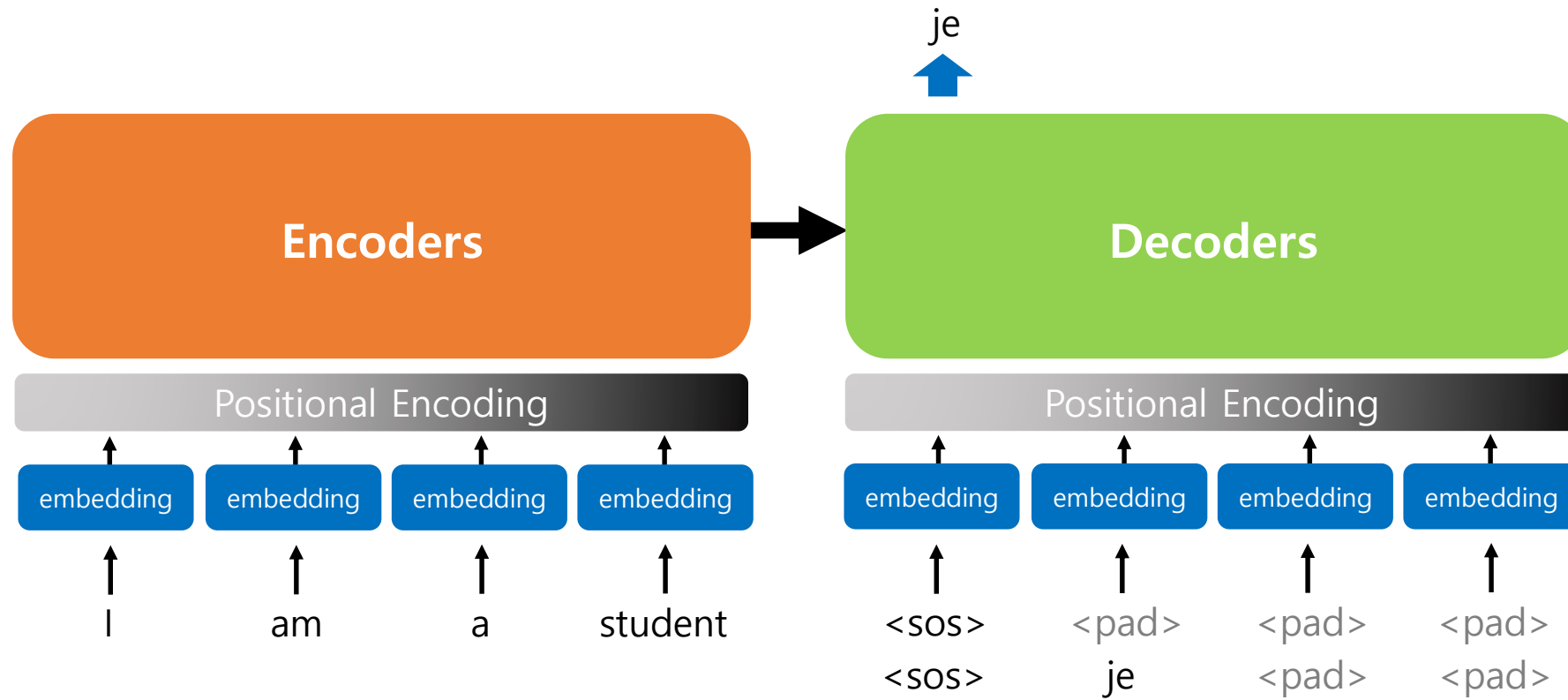
# Transformer : Inference

- 테스트 단계(실제 번역기나 챗봇이 구동될 때)에는 디코더가 단어를 1개씩 생성해내야 한다.



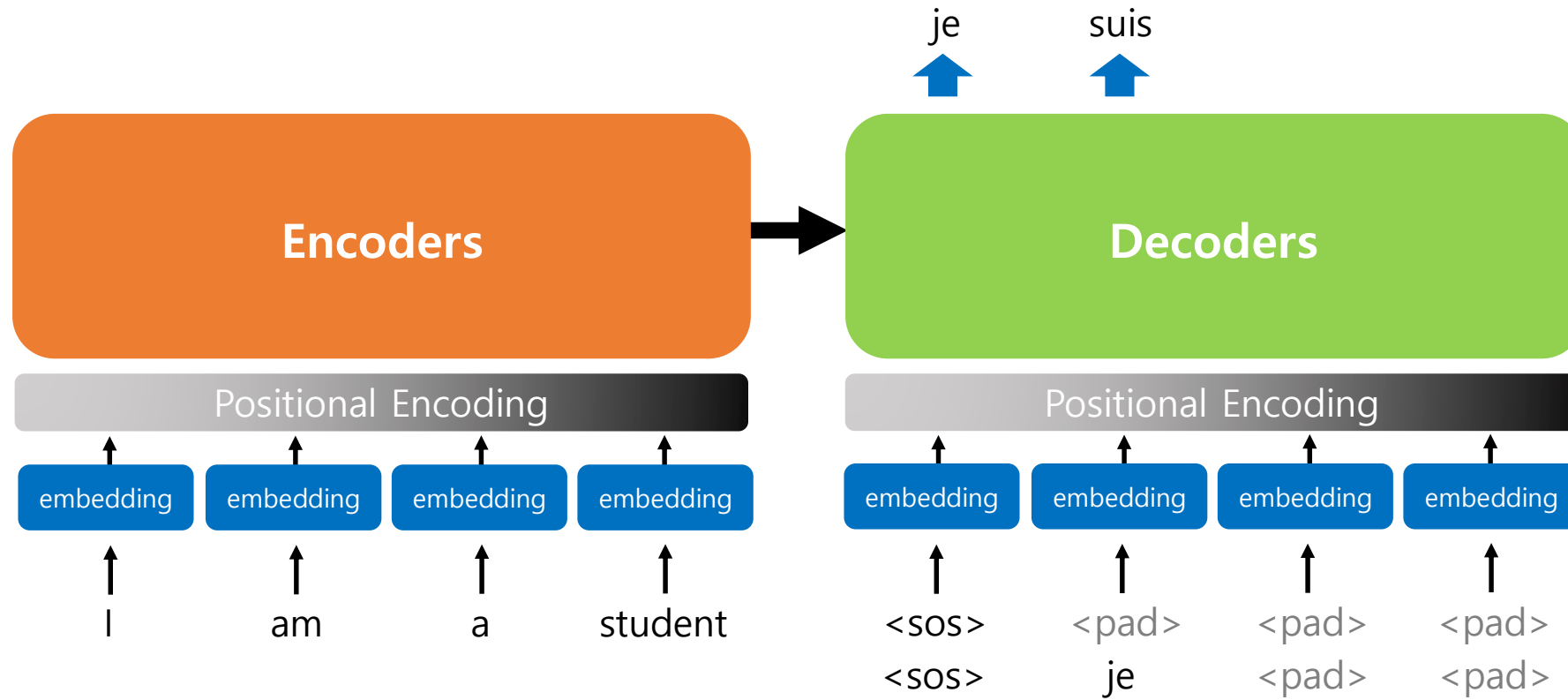
# Transformer : Inference

- 테스트 단계(실제 번역기나 챗봇이 구동될 때)에는 디코더가 단어를 1개씩 생성해내야 한다.



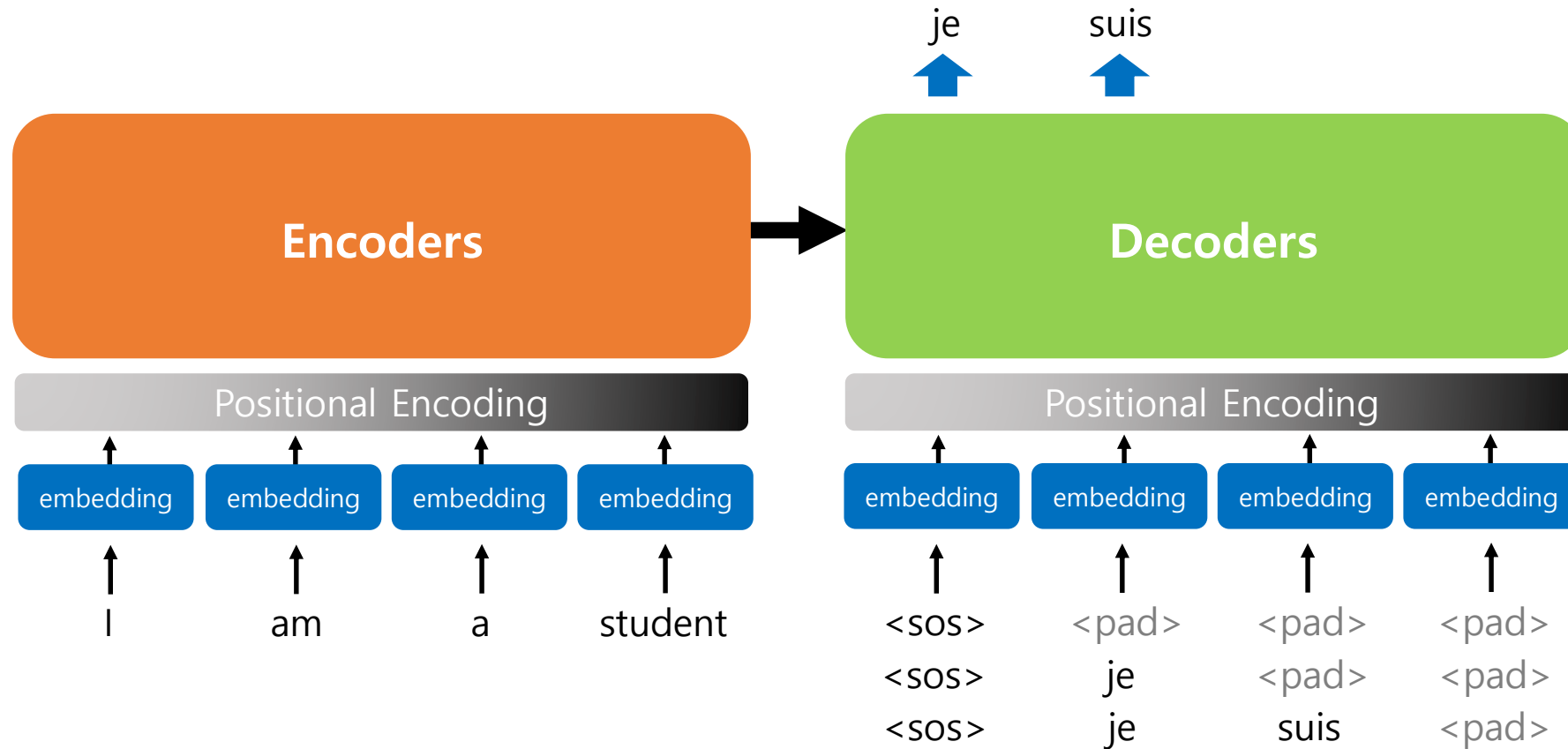
# Transformer : Inference

- 테스트 단계(실제 번역기나 챗봇이 구동될 때)에는 디코더가 단어를 1개씩 생성해내야 한다.



# Transformer : Inference

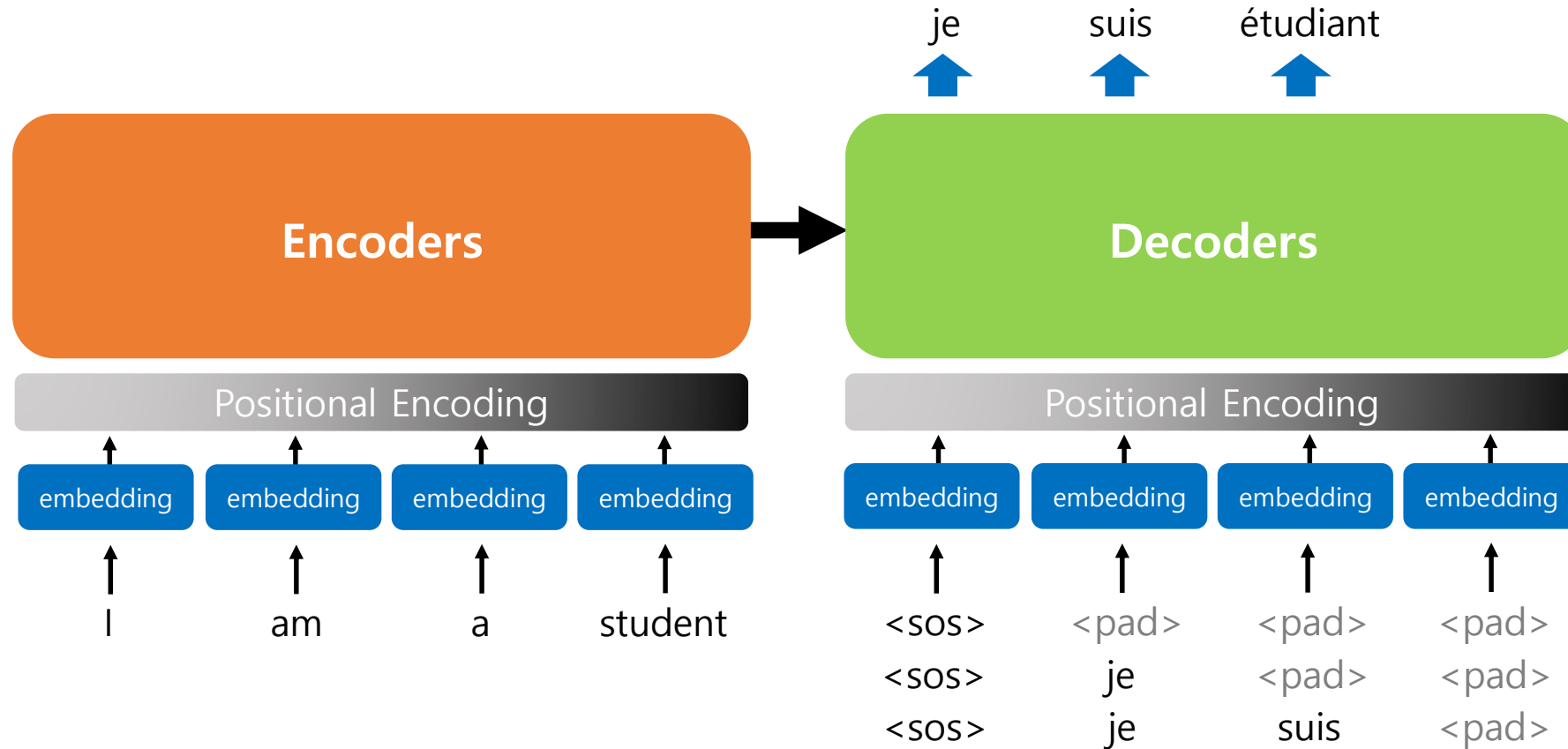
- 테스트 단계(실제 번역기나 챗봇이 구동될 때)에는 디코더가 단어를 1개씩 생성해내야 한다.





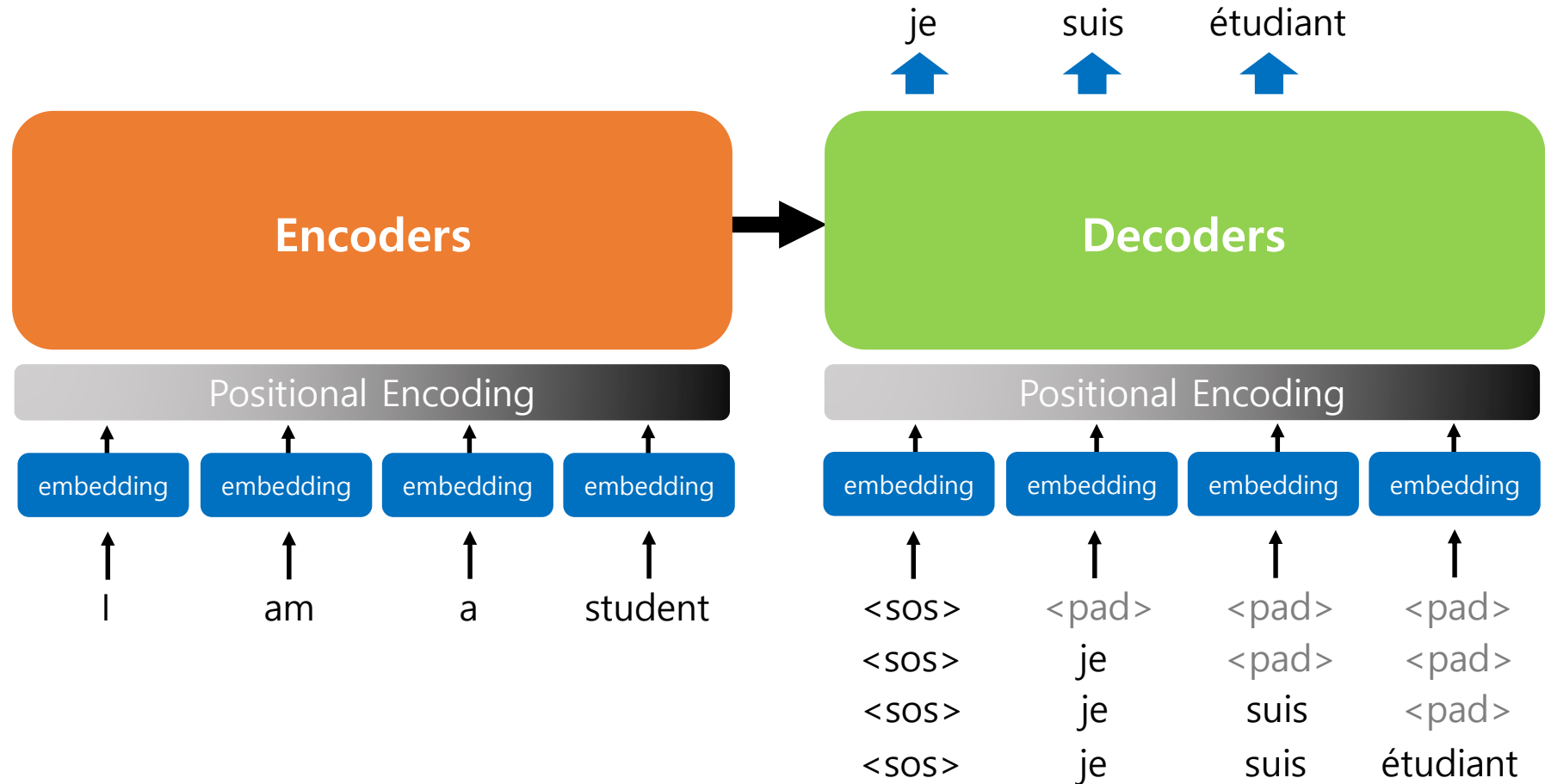
# Transformer : Inference

- 테스트 단계(실제 번역기나 챗봇이 구동될 때)에는 디코더가 단어를 1개씩 생성해내야 한다.



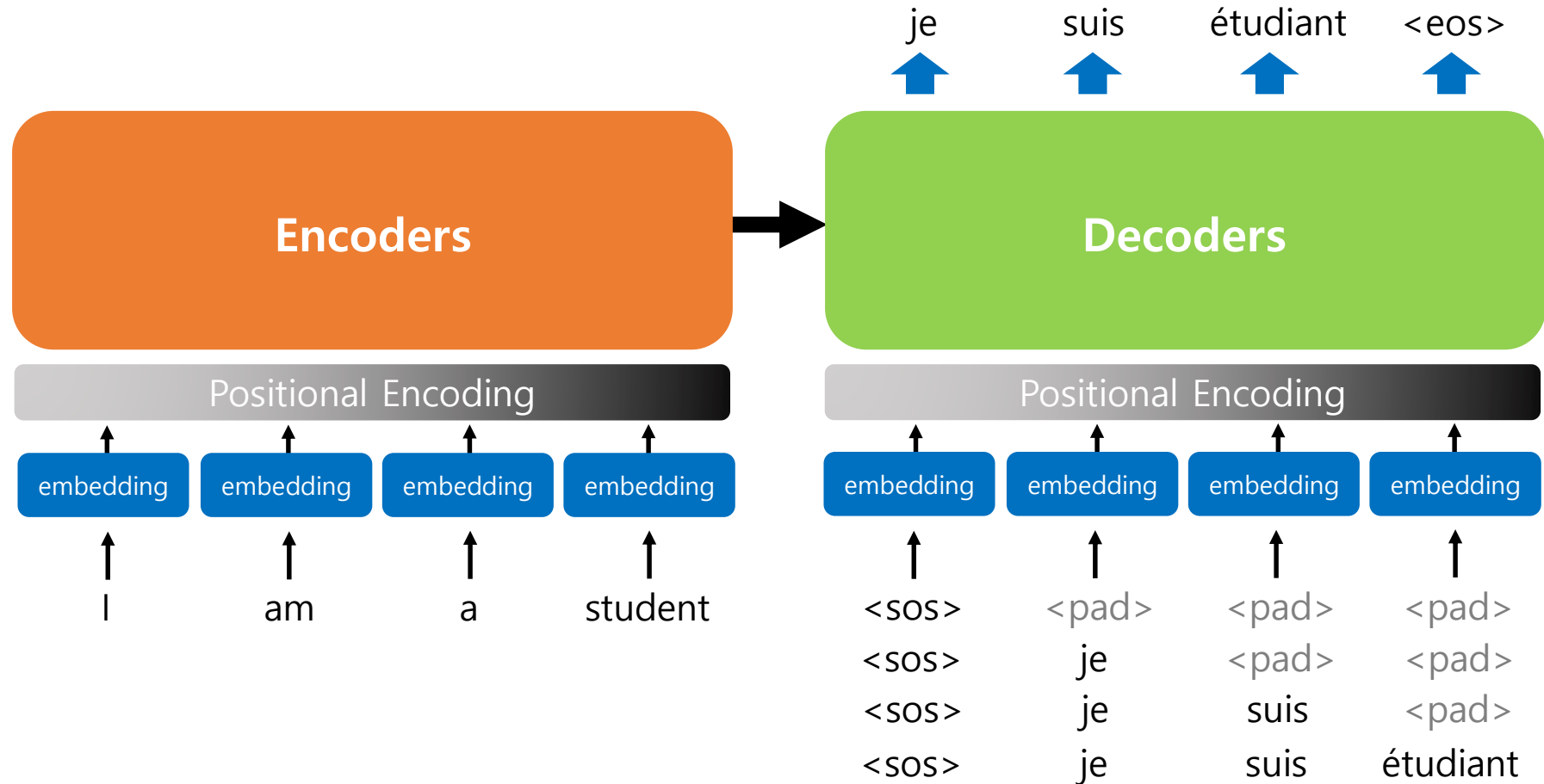
# Transformer : Inference

- 테스트 단계(실제 번역기나 챗봇이 구동될 때)에는 디코더가 단어를 1개씩 생성해내야 한다.



# Transformer : Inference

- 테스트 단계(실제 번역기나 챗봇이 구동될 때)에는 디코더가 단어를 1개씩 생성해내야 한다.



**ELMo**

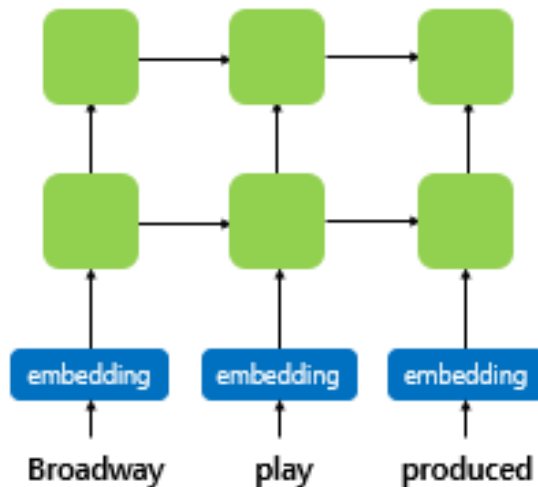
# 엘모(ELMo) – Embeddings from Language Model

- 엘모는 직역하면 '언어 모델로부터 얻는 임베딩'
- 엘모는 BiLM(양방향 언어 모델)을 사용한다.
- 엘모는 순방향과 역방향으로 RNN 언어 모델을 따로 학습시킨다.

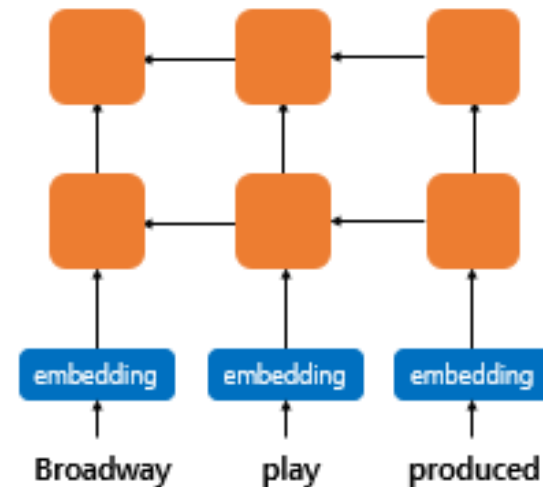


이 캐릭터 이름이 엘모.

순방향 언어 모델  
(Forward Language Model)

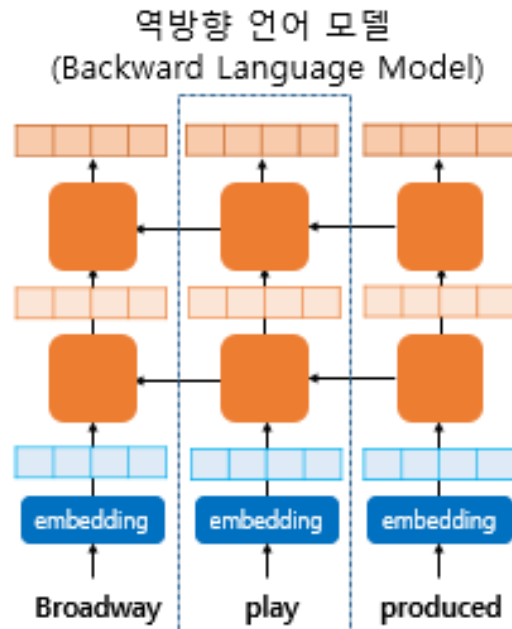
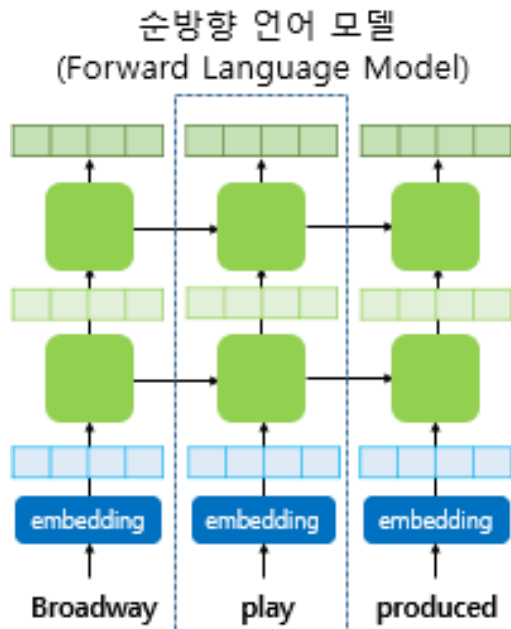


역방향 언어 모델  
(Backward Language Model)



# 엘모(ELMo) – Embeddings from Language Model

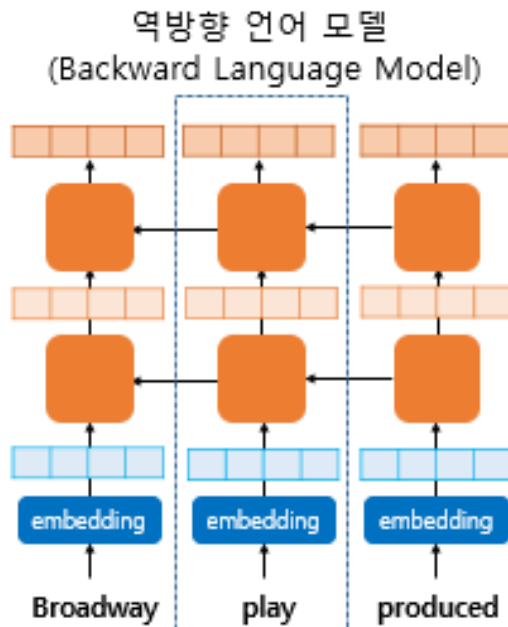
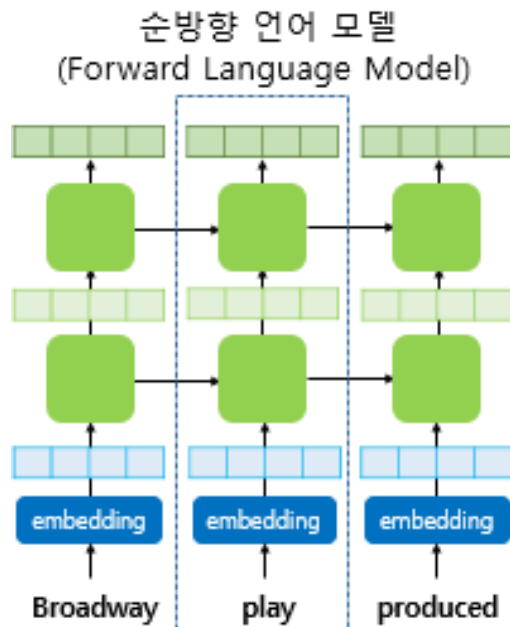
- 엘모는 BiLM(양방향 언어 모델)을 사용한다.
- 엘모는 순방향과 역방향으로 언어 모델을 따로 학습시킨다. 이것이 사전 학습(pre-training) 단계이다.
- 예를 들어 단어 'play'에 대한 임베딩을 얻는다고 가정해보자.



ELMo는 좌측 점선의 사각형 내부의  
각 층의 결과값을 재료로 사용합니다.

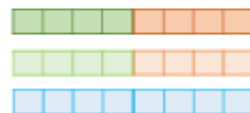
# 엘모(ELMo) – Embeddings from Language Model

- 예를 들어 단어 'play'에 대한 임베딩을 얻는다고 가정해보자.

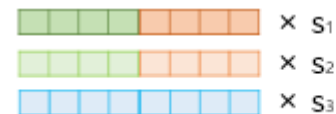


ELMo는 좌측 점선의 사각형 내부의  
각 층의 결과값을 재료로 사용합니다.

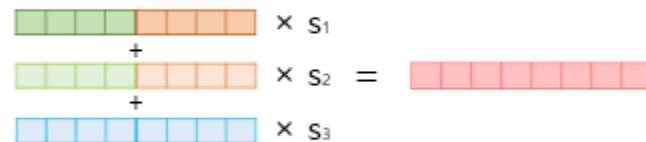
- 1) 각 층의 출력값을 연결(concatenate)한다.



- 2) 각 층의 출력값 별로 가중치를 준다.

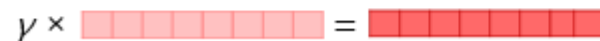


- 3) 각 층의 출력값을 모두 더한다.



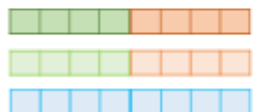
2)번과 3)번의 단계를 요약하여 가중합(Weighted Sum)을 한다고 할 수 있습니다.

- 4) 벡터의 크기를 결정하는 스칼라 매개변수를 곱한다.

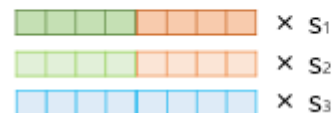


# 엘모(ELMo) – Embeddings from Language Model

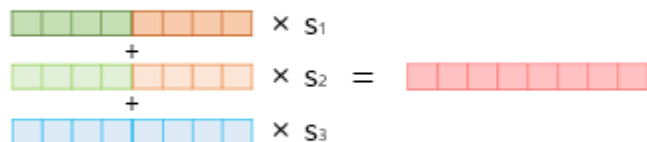
1) 각 층의 출력값을 연결(concatenate)한다.



2) 각 층의 출력값 별로 가중치를 준다.



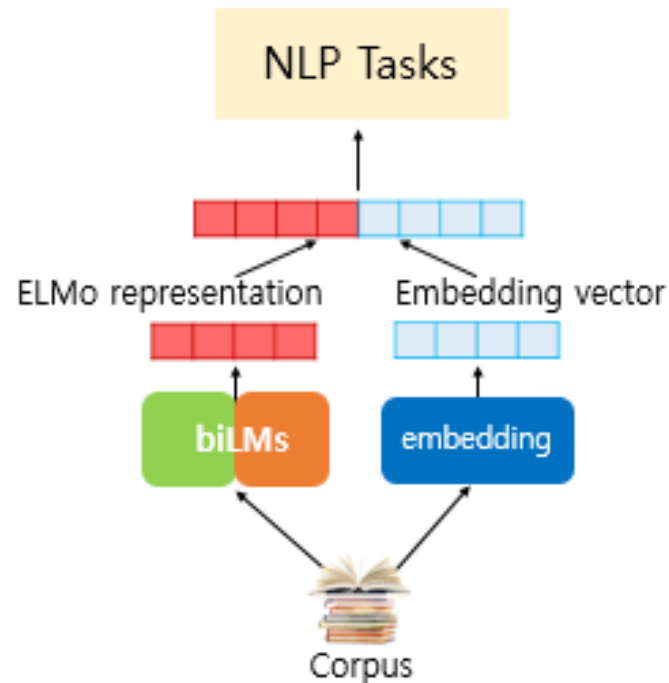
3) 각 층의 출력값을 모두 더한다.



2)번과 3)번의 단계를 요약하여 가중합(Weighted Sum)을 한다고 할 수 있습니다.

4) 벡터의 크기를 결정하는 스칼라 매개변수를 곱한다.

$$\gamma \times \text{[red vector]} = \text{[red vector]}$$





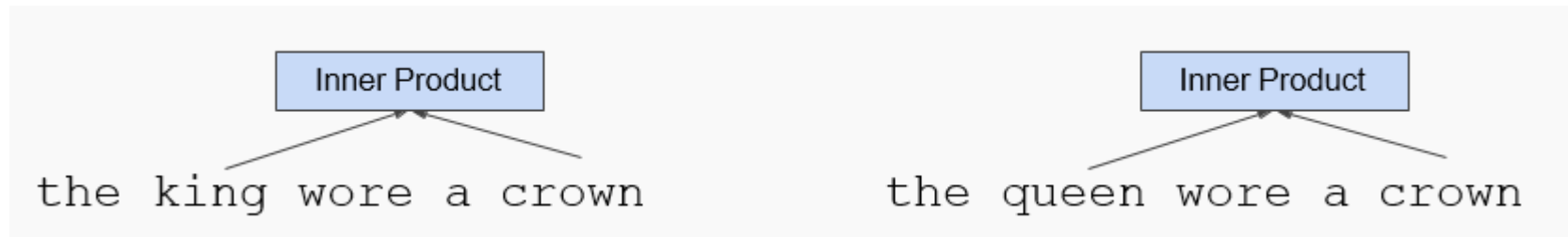
# Pretraining in NLP

# Pre-training in NLP

- 워드 임베딩은 딥 러닝 자연어 처리의 기본.

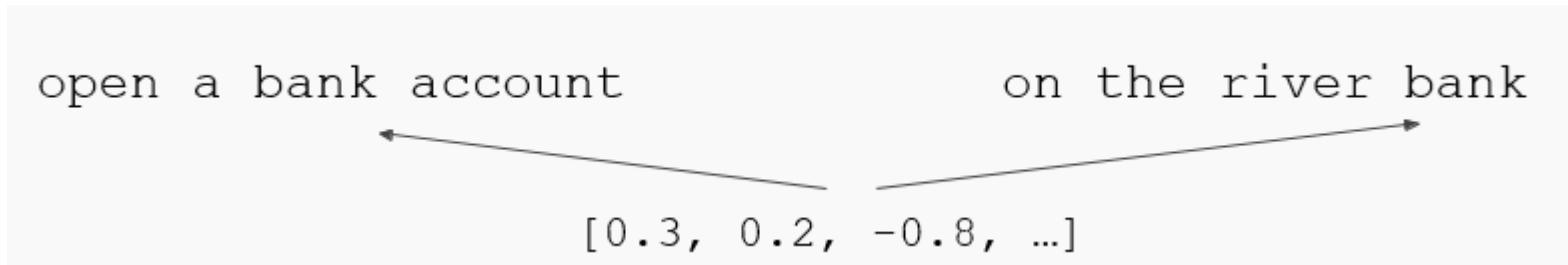


- 사전 훈련된 임베딩(Word2Vec, GloVe)은 대용량 텍스트의 단어들의 동시 등장 통계로부터 훈련시키는 방법.

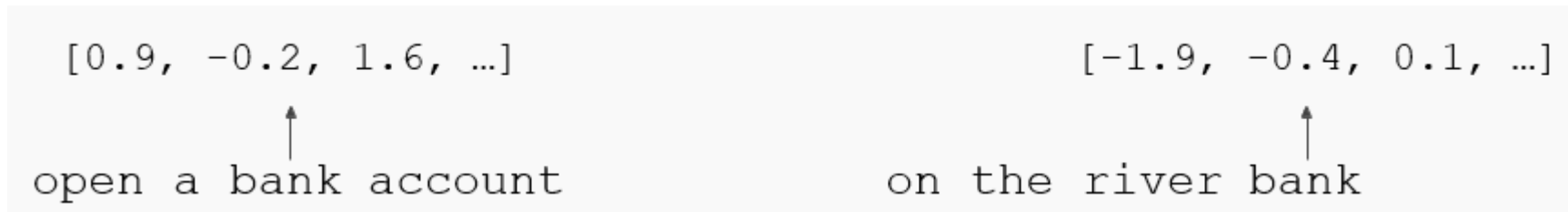


# Contextual Representation

- **문제점** : 단어는 문맥에 따라서 뜻이 달라질 수 있음.  
Ex) 동음이의어. 배, 사과 등.



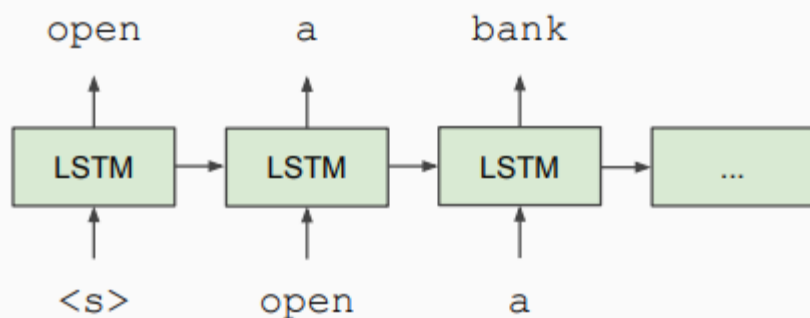
- **해결 방법** : 언어 모델을 사전 훈련 시켜서 문맥에 따른 표현(Contextual Representation)을 얻는다.



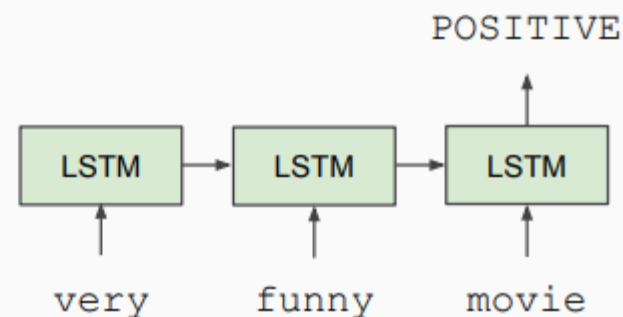
# History of Contextual Representations

- *Semi-Supervised Sequence Learning*, Google, 2015

## Train LSTM Language Model



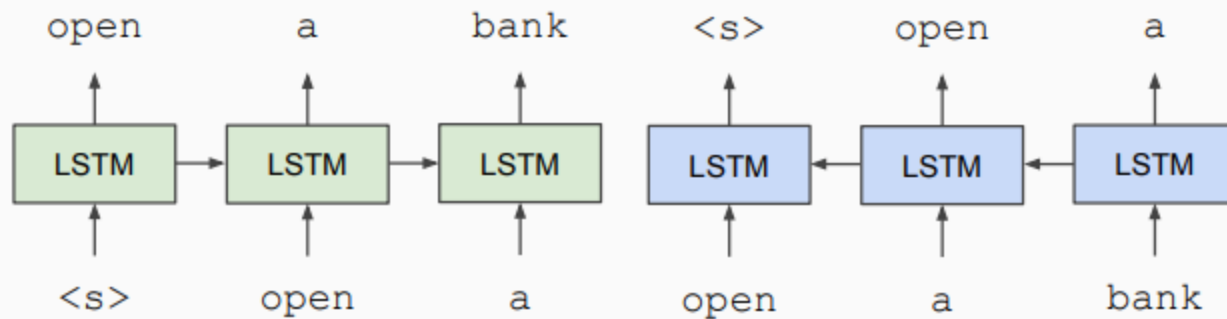
## Fine-tune on Classification Task



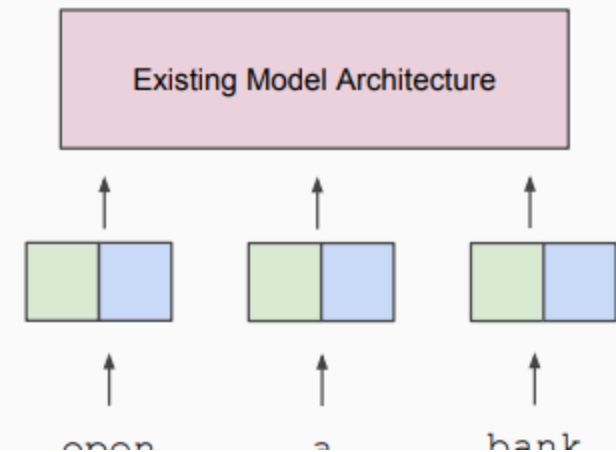
# History of Contextual Representations

- *ELMo: Deep Contextual Word Embeddings*, AI2 & University of Washington, 2017

**Train Separate Left-to-Right and Right-to-Left LMs**



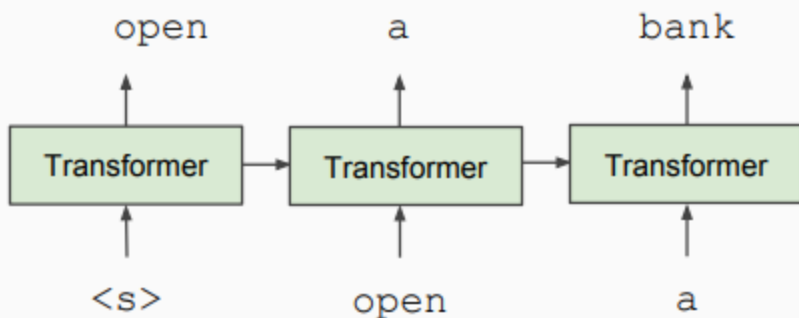
**Apply as “Pre-trained Embeddings”**



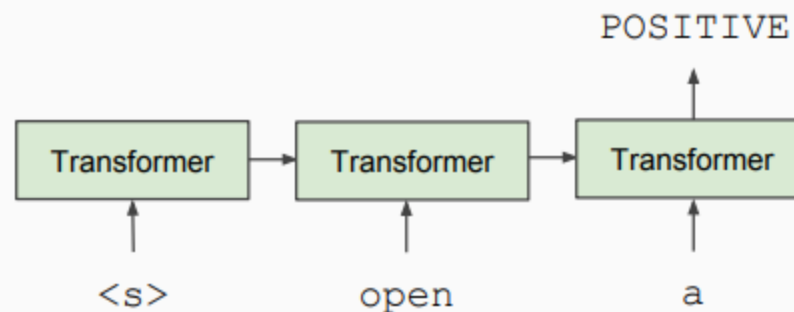
# History of Contextual Representations

- *Improving Language Understanding by Generative Pre-Training*, OpenAI, 2018

**Train Deep (12-layer)  
Transformer LM**



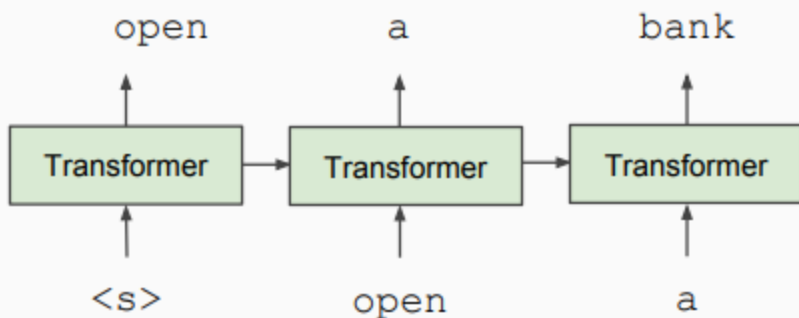
**Fine-tune on  
Classification Task**



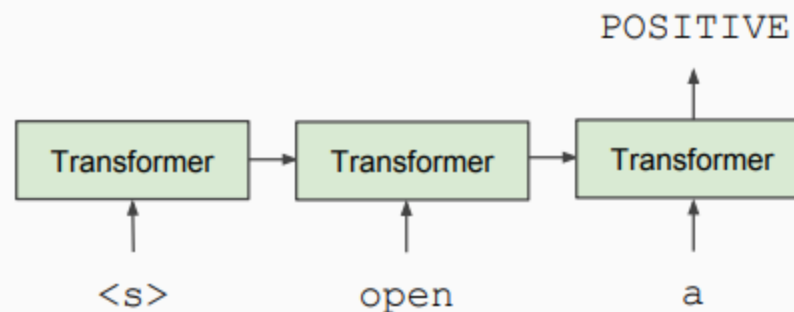
# History of Contextual Representations

- *Improving Language Understanding by Generative Pre-Training*, OpenAI, 2018 이 모델을 GPT라고 한다.

**Train Deep (12-layer)  
Transformer LM**



**Fine-tune on  
Classification Task**



# Problem with Previous Methods

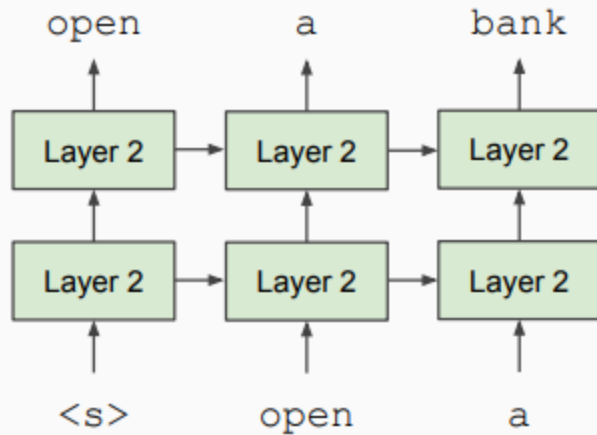
- **문제점** : 언어 모델은 정방향 또는 역방향으로만 진행된다.  
하지만 사실 언어의 문맥이라는 것은 양방향을 통해서 결정이 된다.
- 그렇다면 왜 언어 모델은 **단방향(Unidirectional)**이어야만 할까?
- **이유** : 양방향 언어모델을 만들 경우,  
미리 예측할 단어에 대한 정보를 보게되기 때문.



# Problem with Previous Methods

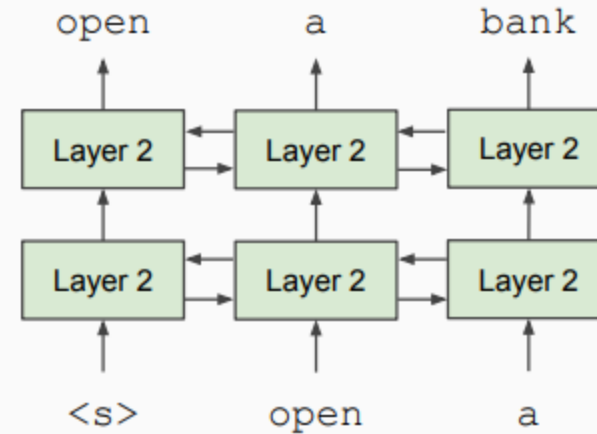
## Unidirectional context

Build representation incrementally



## Bidirectional context

Words can “see themselves”



# Masked Language Model

- **해결 방법** : 훈련 데이터의 단어 중  $k\%$ 의 단어를 [MASK]로 변경하여 이를 예측하도록 한다.

○ We always use  $k = 15\%$

store                      gallon  
↑                            ↑  
the man went to the [MASK] to buy a [MASK] of milk

Masked Language Model을 제안한 논문에서는  $k = 15$ 로 사용.

# Masked Language Model

15%의 [MASK] token을 만들어 낼 때 몇 가지 추가적인 처리를 해준다.

- 80%의 경우 : token을 [MASK]로 바꾼다.

eg., my dog is hairy -> my dog is [MASK]

- 10%의 경우 : token을 random word로 바꾼다.

eg., my dog is hairy -> my dog is apple

- 10%의 경우 : token을 원래의 단어로 그대로 놔둔다.

이는 실제 관측된 단어에 대한 정보 또한 남겨주기 위해서이다.

**BERT**

# 기존의 모델들

BERT 이전의 Pre-trained 모델은 크게 두 가지로 구성된다.

- Feature-based (사전 훈련된 모델의 파라미터를 고정)
  - **ELMo**
  - **Shallowly** 양방향 언어 모델(biLM)
  - **ELMo의 경우 양방향이지만 진정한 양방향이 아니다.**
  - 언어 모델의 파라미터는 Freezing하고,  
각 layer에 대한 비율 조정 파라미터( $\gamma$ )를 계산
    - $[\gamma_{input}, \gamma_{layer1}, \gamma_{layer2}, \gamma_{layer3}]$
- Fine-tuning (사전 훈련된 모델의 파라미터를 함께 학습)
  - **GPT**
  - **단방향(Unidirectional)** 언어 모델(language models)
  - **GPT의 경우 단방향 모델이다.**

# 기존의 모델들

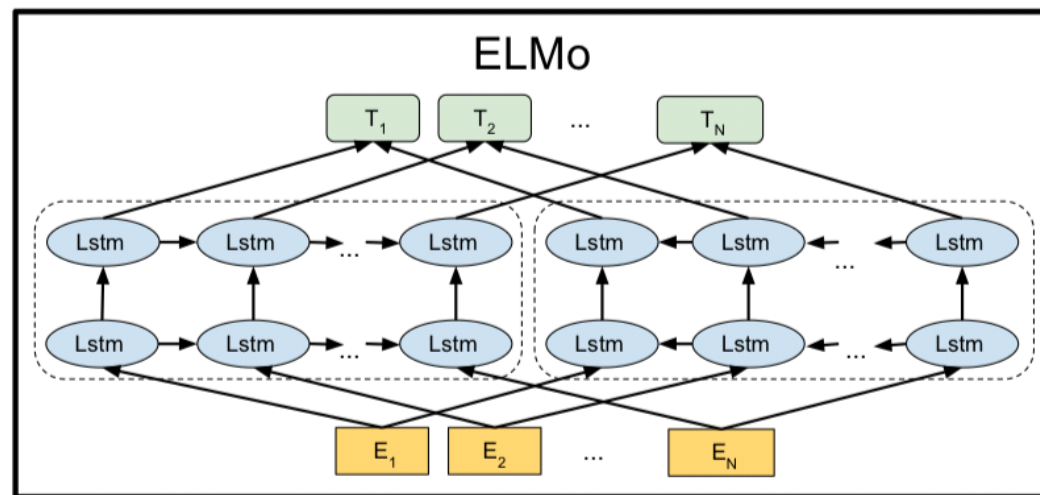
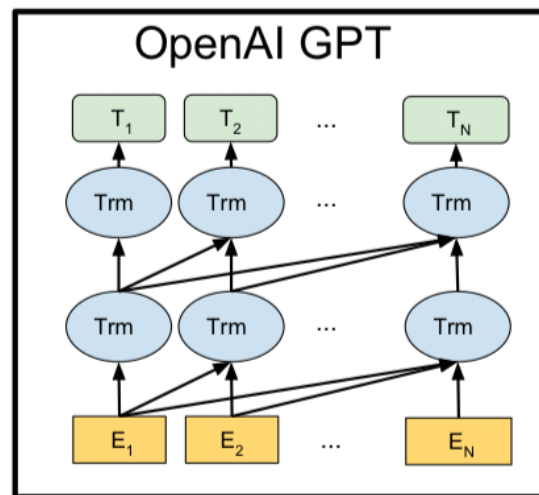
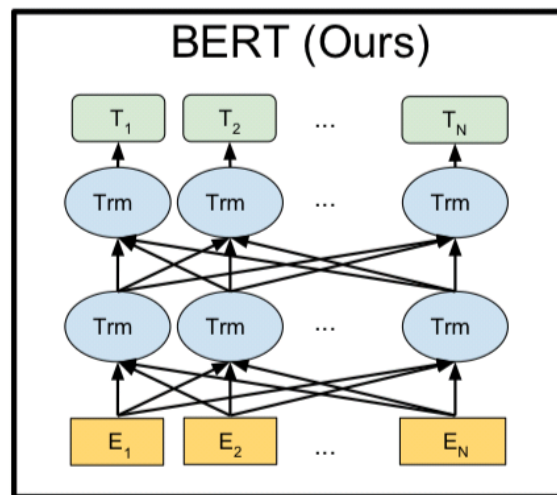
BERT 이전의 Pre-trained 모델은 크게 두 가지로 구성된다.

- Feature-based (사전 훈련된 모델의 파라미터를 고정)
  - **ELMo**
  - **Shallowly** 양방향 언어 모델(biLM)
  - **ELMo의 경우 양방향이지만 진정한 양방향은 아니다.**
  - 언어 모델의 파라미터는 Freezing하고,  
각 layer에 대한 비율 조정 파라미터( $\gamma$ )를 계산
    - $[\gamma_{input}, \gamma_{layer1}, \gamma_{layer2}, \gamma_{layer3}]$

**BERT**  
Fine-tuning 방식을 취하되,  
GPT와는 달리 양방향을 실현한다.

- Fine-tuning (사전 훈련된 모델의 파라미터를 함께 학습)
  - **GPT**
  - **단방향(Unidirectional)** 언어 모델(language models)
  - **GPT의 경우 단방향 모델이다.**

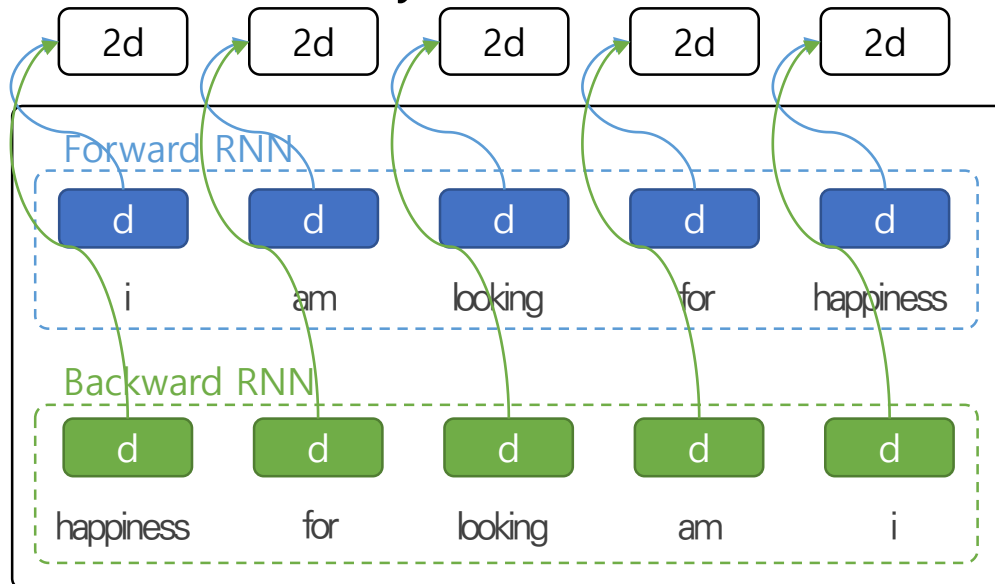
# 기존의 모델들



# Un-directional Vs. Bi-directional

- RNN Language Model, Transformer: 이전 단어들을 보고 다음 단어를 예측(**Uni-directional**)
- BiLM : Shallowly Bi-directional
- BERT
  - Masked Language Model : 문장에 Masking 15% 처리한 후 Mask된 단어를 예측
    - 1. 주변 단어들을 보고 Masked 단어를 예측 (**Bi-directional**)

Bidirectional RNN(shallowly 양방향)



Bidirectional BERT(deeply 양방향)

i am looking for happiness . I am Donghwa

Input: i am looking for [MASK1] . I am [MASK2]

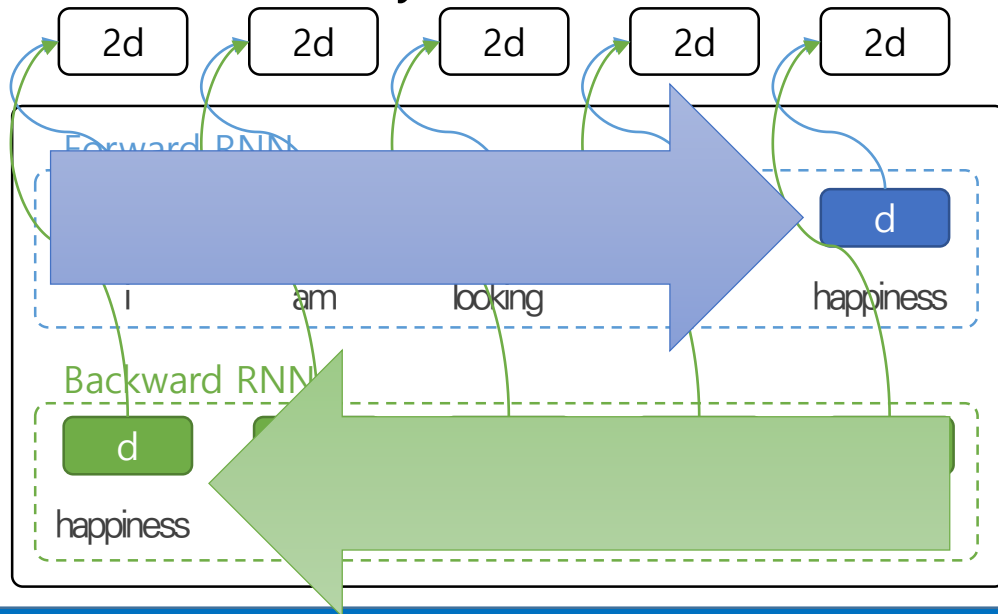
Labels: [MASK1] = happiness; [MASK2] = Donghwa



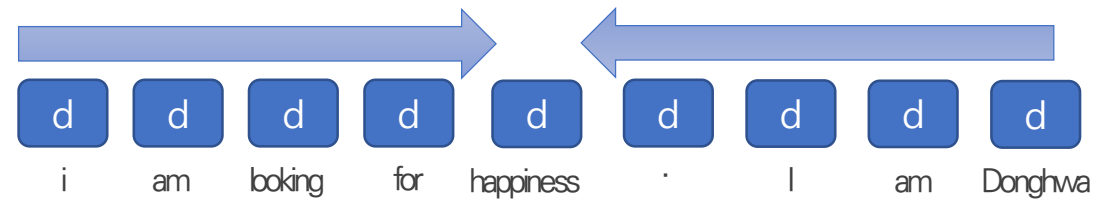
# Un-directional Vs. Bi-directional

- RNN Language Model, Transformer: 이전 단어들을 보고 다음 단어를 예측(**Uni-directional**)
- BiLM : Shallowly Bi-directional
- BERT
  - Masked Language Model : 문장에 Masking 15% 처리한 후 Mask된 단어를 예측
    - 1. 주변 단어들을 보고 Masked 단어를 예측 (**Bi-directional**)

Bidirectional RNN(shallowly)



Bidirectional BERT(deeply )



**Input:** i am looking for [MASK1] . I am [MASK2]

**Labels:** [MASK1] = happiness; [MASK2] = Donghwa

# Next Sentence Prediction

- BERT
  - 2. 문장들 사이의 관계를 학습하기 위해 “다음 문장 예측 태스크를 도입

BERT에서는 훈련 데이터에서 두 문장을 이어붙여 원래의 훈련 데이터에서 붙여져 있던 문장인지를 맞추는 다음 문장 예측 문제를 수행합니다.

50% : Sentence A, B가 실제 다음 문장

50% : Sentence A, B가 랜덤으로 뽑힌 관계가 없는 두 문장

Sentence A: i am looking for happiness

Sentence B: i am Wonjoon

Labels: **Is**NextSentence

Sentence A: i am looking for happiness

Sentence B: i am Sana

Labels: **Not**NextSentence

# BERT의 Pre-training

결론적으로 BERT는 사전 학습 단계에서 두 가지 방법으로 학습된다.

- 1. Masked Language Model : 문장에 Masking 15% 처리한 후 Mask된 단어를 예측
- 2. 문장들 사이의 관계를 학습하기 위해 “다음 문장 예측 태스크를 도입”

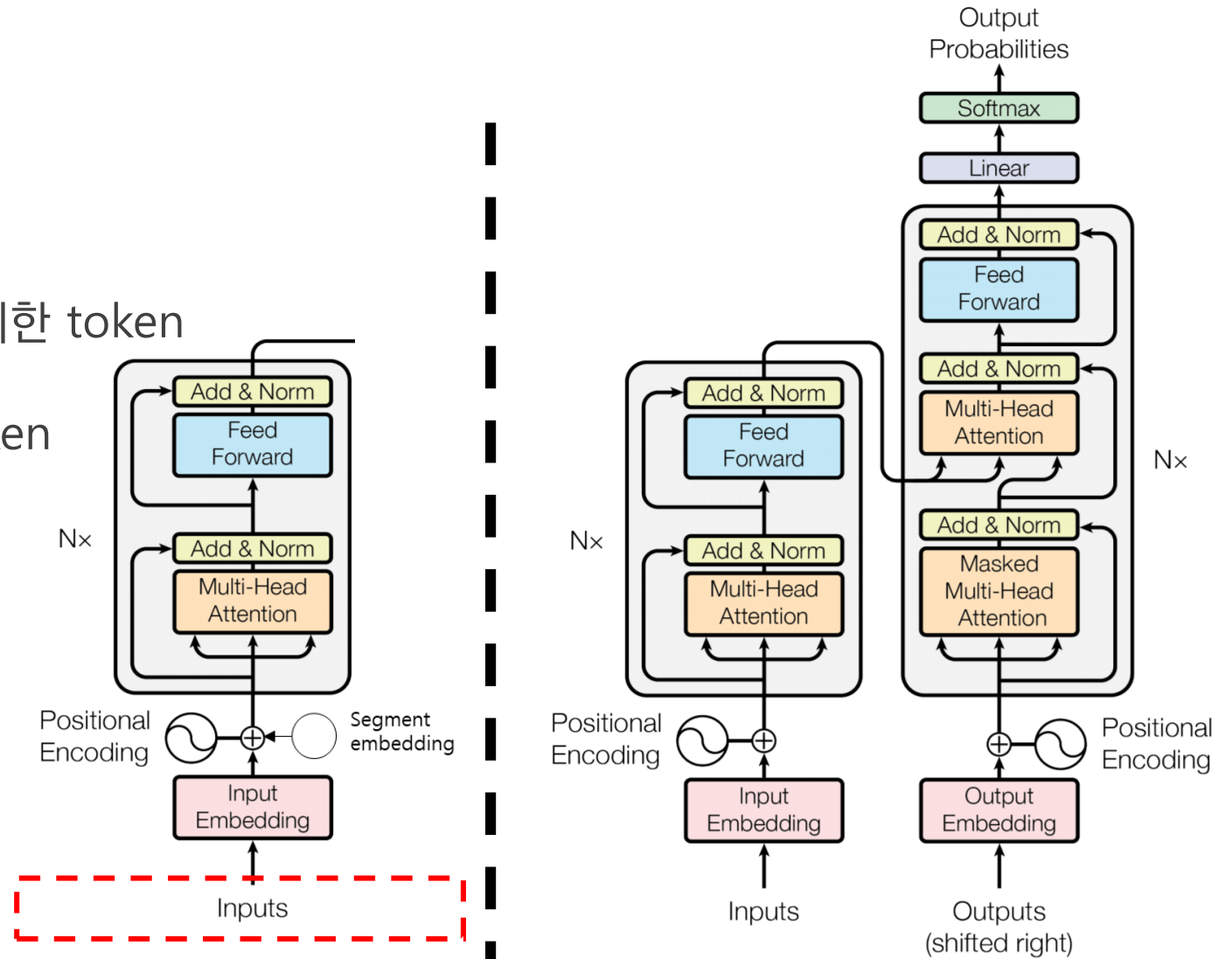
사실 BERT의 성능의 대부분은 1. Masked Language Model에서 나오는 편이고,  
2.의 경우에는 BERT가 향후 파인 튜닝 시에 두 개의 문장을 입력으로 받는 태스크를 풀 경우를 위해 추가한 것.

Ex) 두 개의 문장을 입력받고 이 문장이 모순 관계인지, 독립 관계인지를 맞추는 태스크 등.

실제로 BERT 이후의 연구들에서는 2.를 그다지 중요하게 생각하지 않는 경우가 많음.

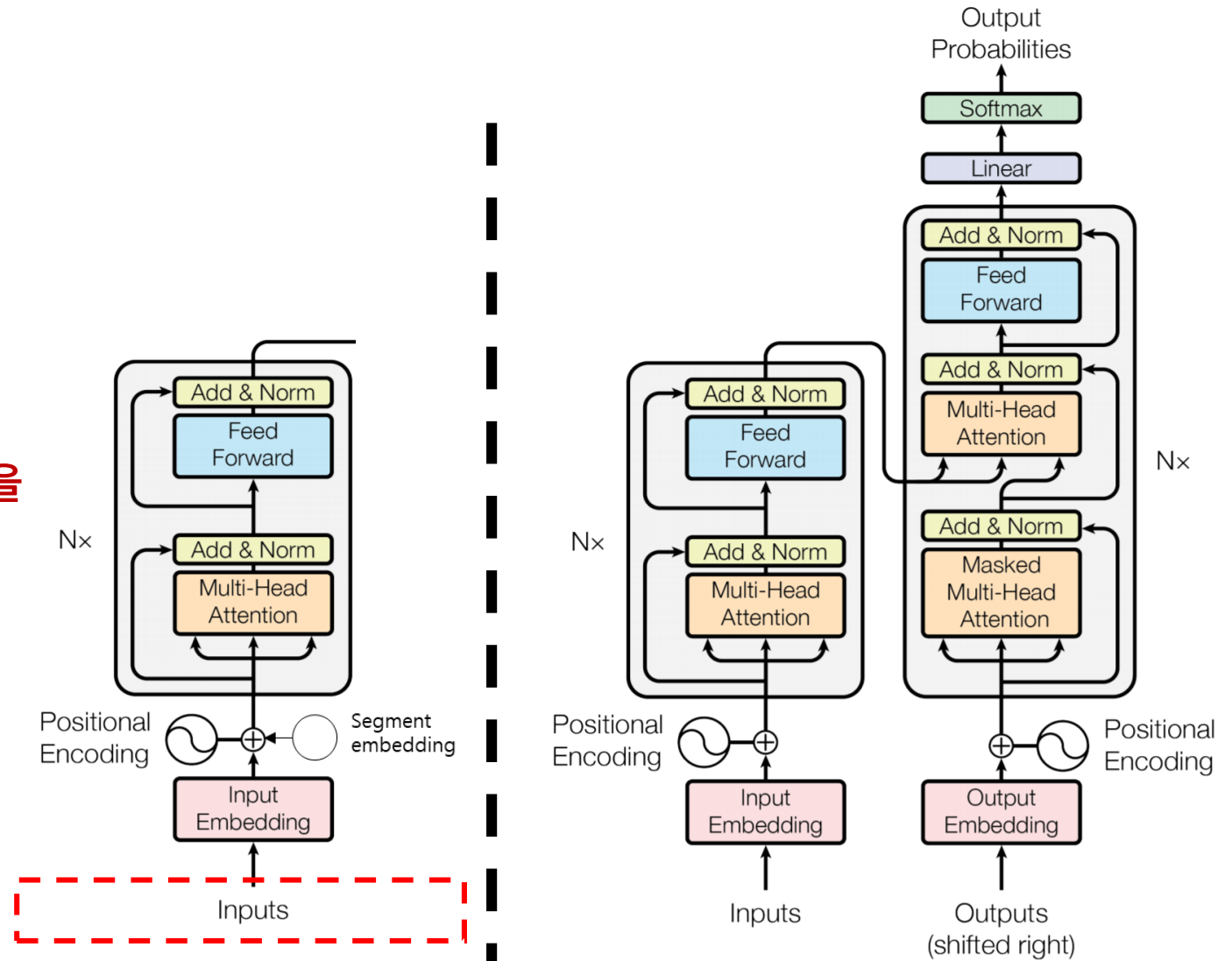
# Transformer Vs. BERT

- Transformer
  - [SOS]: 문장의 시작을 알리는 token
  - [EOS]: 문장의 끝을 알리는 token
- BERT
  - [CLS]: Task-specific한 정보를 주기 위한 token
  - [SEP]: Token A, B를 구분하는 token
  - [SEP]: Token A, B의 끝을 알리는 token
  - [MASK]: 예측하는 Target



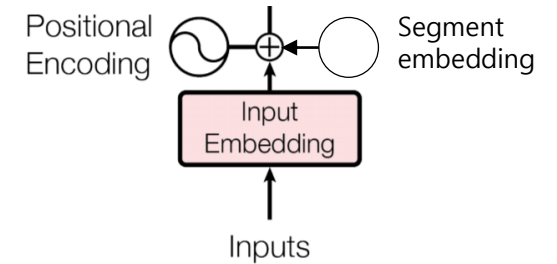
# Transformer Vs. BERT

- Transformer
  - Input embedding
  - Positional embedding
- BERT
  - Input embedding
  - Positional embedding
  - **Segment embedding 추가**
    - **Token A group, Token B group을 구분하는 정보**



# BERT의 세 가지 Embedding

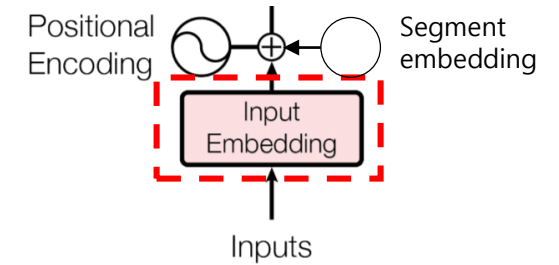
- BERT에는 총 세 가지 Embedding이 사용된다.
- Token Embedding은 우리가 알고있는 Input Word Embedding
- Segment Embedding은 두 개의 문장을 구분하기 위한 Embedding
- Position Embedding은 트랜스포머의 Positional Encoding을 대체할 수 있는 방법



Input	[CLS]	my	dog	is	cute	[SEP]	he	likes	play	##ing	[SEP]
Token Embeddings	$E_{[CLS]}$	$E_{my}$	$E_{dog}$	$E_{is}$	$E_{cute}$	$E_{[SEP]}$	$E_{he}$	$E_{likes}$	$E_{play}$	$E_{##ing}$	$E_{[SEP]}$
	+	+	+	+	+	+	+	+	+	+	+
Segment Embeddings	$E_A$	$E_A$	$E_A$	$E_A$	$E_A$	$E_A$	$E_B$	$E_B$	$E_B$	$E_B$	$E_B$
	+	+	+	+	+	+	+	+	+	+	+
Position Embeddings	$E_0$	$E_1$	$E_2$	$E_3$	$E_4$	$E_5$	$E_6$	$E_7$	$E_8$	$E_9$	$E_{10}$

# Token Embedding

- Encoder
  - Sent1: [CLS] i am looking for happiness [SEP], B type sentence [SEP]



looking  $x_j$

Vocab size(Encoder)

<PAD>	<s>	</s>	<UNK>	am	for	...	i	looking	...	w	...	z
0	0	0	0	0	0	0	0	1	0	0	0	0

Embed size

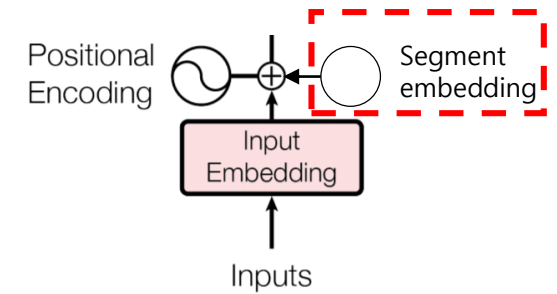
<PAD>	0.0	0.0	0.0	0.0
⋮	0.0	0.3	0.7	0.0
looking	0.2	0.1	0.7	0.2
⋮	0.9	0.1	0.7	0.9
⋮	0.5	0.7	0.1	0.5
⋮	0.2	0.9	0.4	0.2
z	0.2	0.0	0.7	0.2

0.2	0.1	0.7	0.2
-----	-----	-----	-----

$w_j$  looking

# Segment Embedding

- Encoder
  - Sent1: [CLS] i am looking for happiness [SEP], B type sentence [SEP]



looking  $x_j$

Vocab size(Encoder)

A	B
1	0

Embed size

A				
B				

0.1	0.1	0.1	0.3
-----	-----	-----	-----

$w_j$  looking



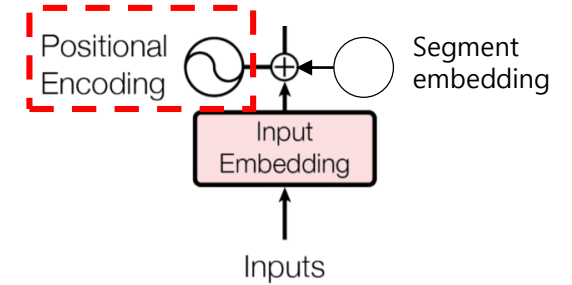
# Position Embedding

- Encoder
  - Sent1: [CLS] i am looking for happiness [SEP], B type sentence [SEP]

3th position

Position vocab					
0	...	3	...	9	10
0	...	1	0	0	0

Embed size				
0	0.1	0.4	0.7	0.1
⋮	0.0	0.3	0.7	0.0
3	0.1	0.1	0.9	0.1
⋮	...	...	...	...
10	0.2	0.0	0.7	0.2



3th position embedding

0.1 0.1 0.9 0.1

# Position Embedding

- Encoder

- Sent1: [CLS] i am looking for happiness [SEP], B type sentence [SEP]

0 1 2 3 4 5 6 7 8 9 10

샘플의 길이가 11이라고 가정해보자.

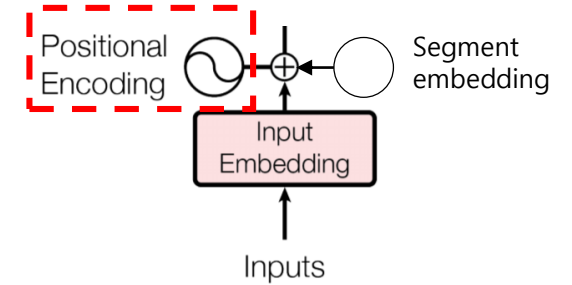
3th position

Position vocab

0	...	3	...	9	10
0	...	1	0	0	0

Embed size

0	0.1	0.4	0.7	0.1
⋮	0.0	0.3	0.7	0.0
3	0.1	0.1	0.9	0.1
⋮	...	...	...	...
10	0.2	0.0	0.7	0.2

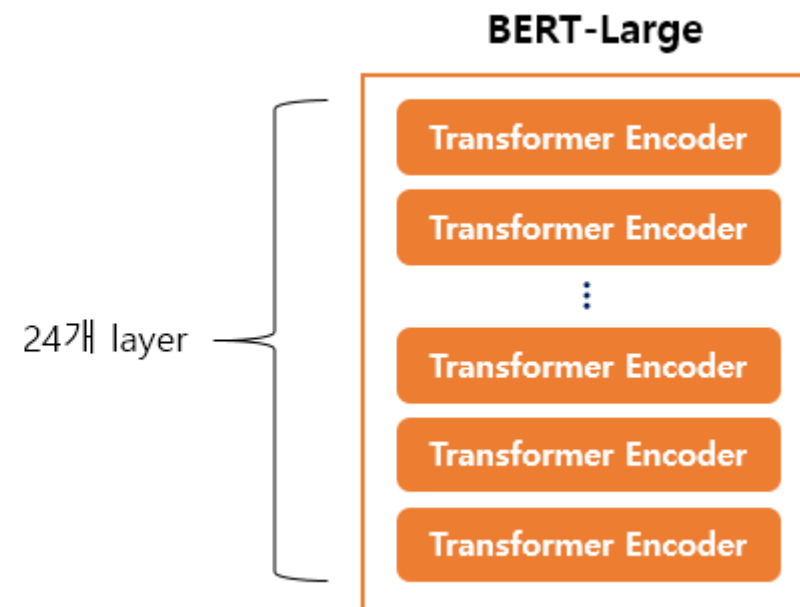
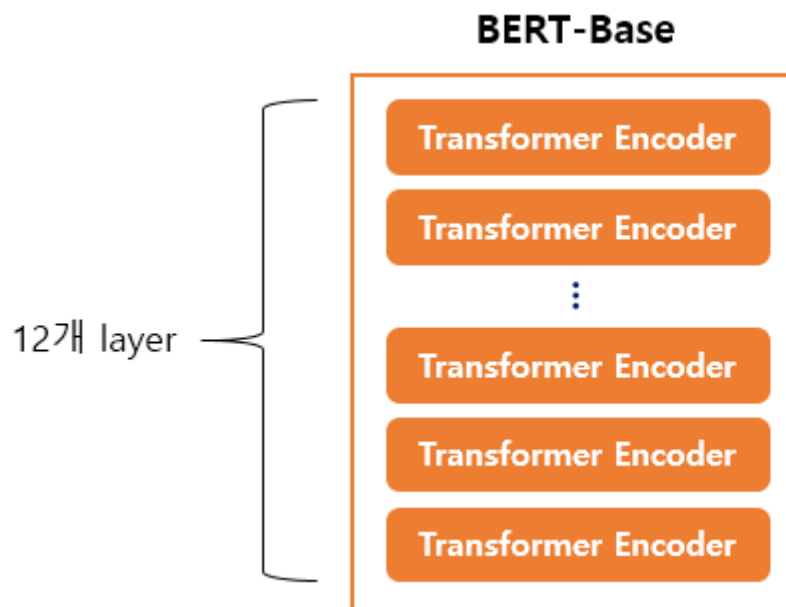
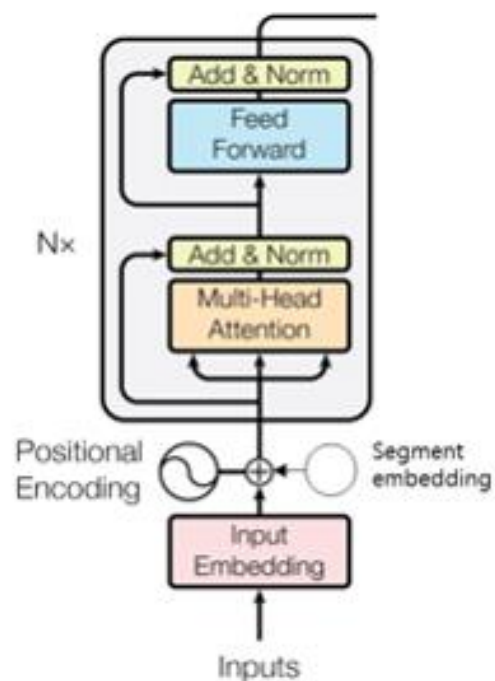


3th position embedding

0.1 0.1 0.9 0.1

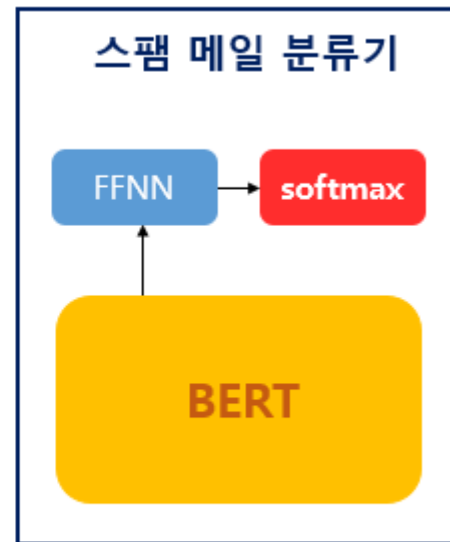
# BERT-Base Vs. BERT-Large

- BERT-BASE는 트랜스포머의 인코더를 12층을 적재.
- BERT-LARGE는 트랜스포머의 인코더를 24층을 적재.



# BERT의 적용

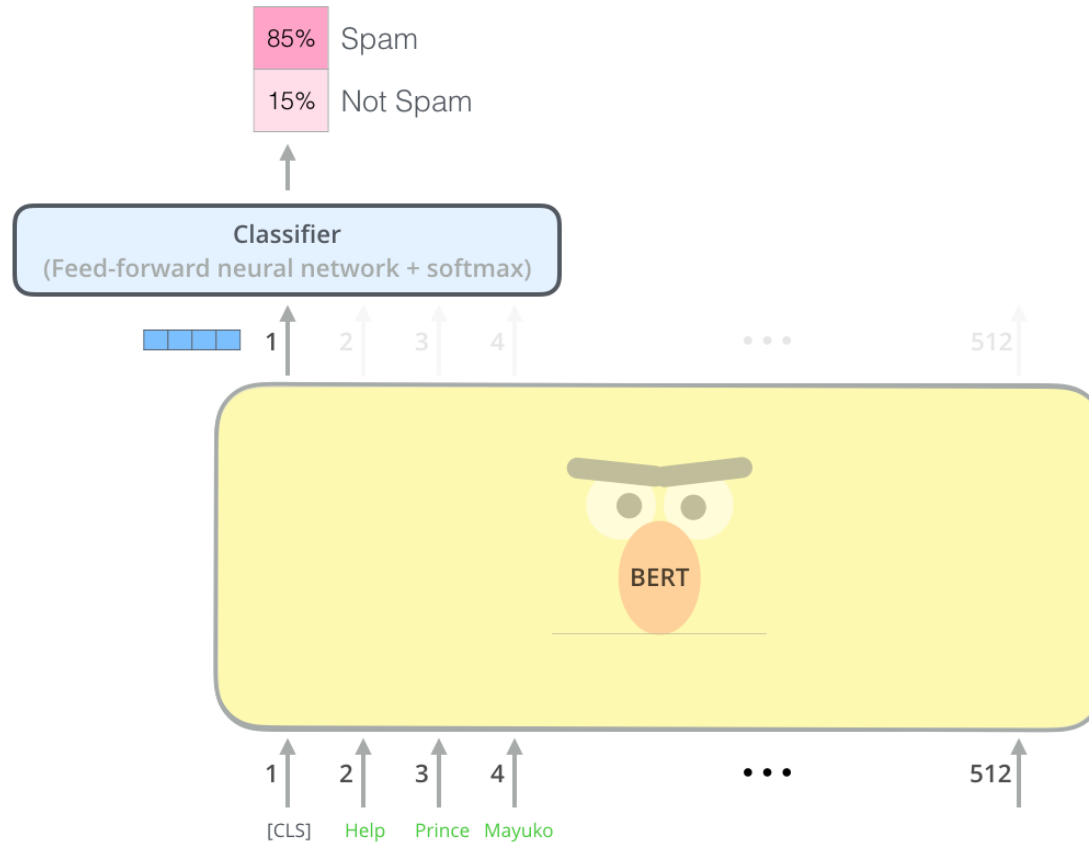
- 방대한 데이터로 사전 훈련된 언어 모델 BERT.
- BERT에 풀고자 하는 태크스에 맞는 추가적인 신경망을 추가.
- 풀고자 하는 태스크의 오차에 맞추어서 BERT를 학습(fine-tuning)



33억 단어에 대해서 4일간 학습시킨 언어 모델

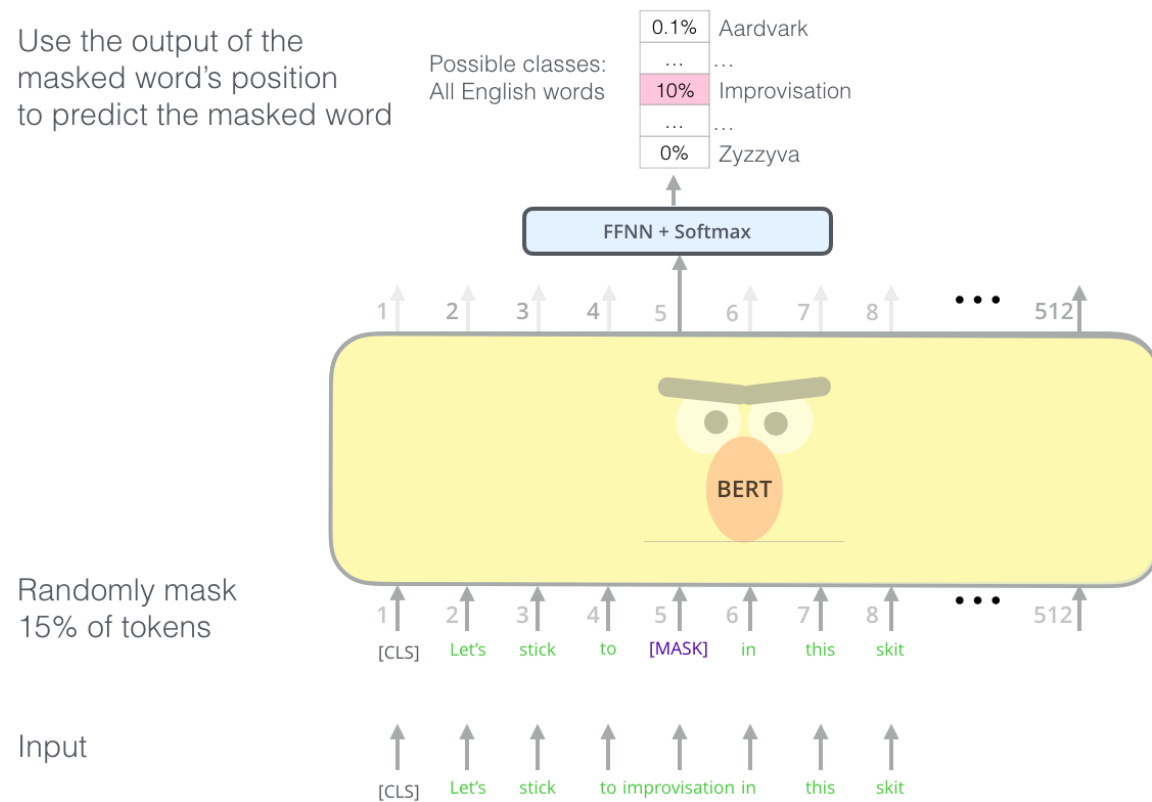
# BERT의 적용

- BERT의 윗단의 신경망을 통해서 원하는 Task를 수행.
- 텍스트 분류의 경우 <CLS>의 토큰을 최종 Classification을 위한 입력으로 사용.



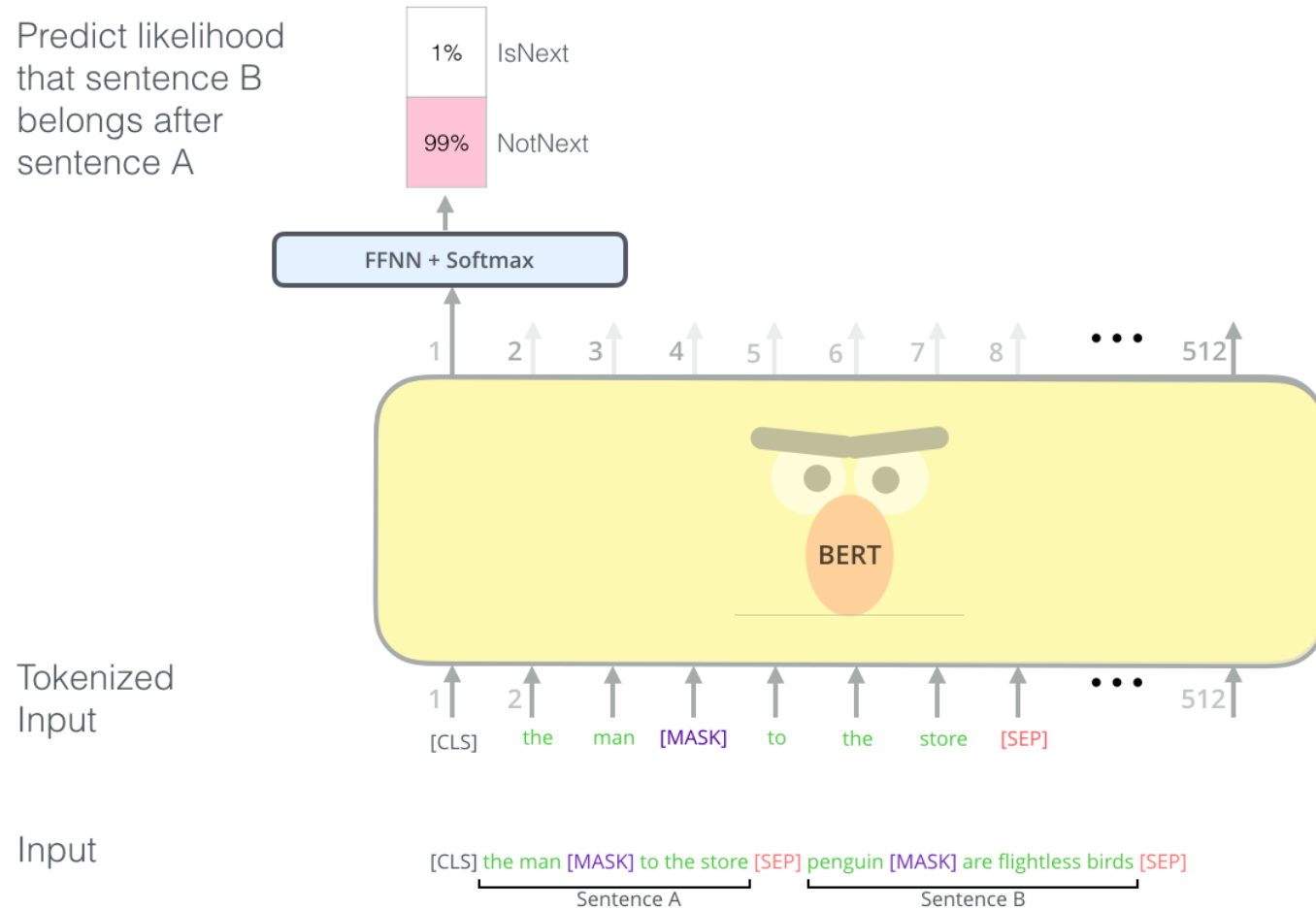
# BERT의 사전 훈련

- BERT는 사전 훈련 시 첫번째 Task로 Masked Language Model을 학습



# BERT의 사전 훈련

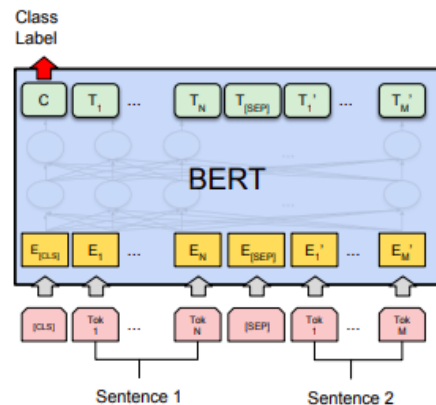
- BERT는 사전 훈련 시 두번째 Task로 다음 문장 예측



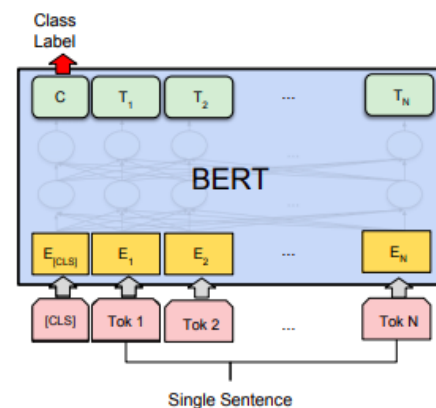
# BERT의 다양한 적용

- BERT를 다양한 태스크에 적용 가능.

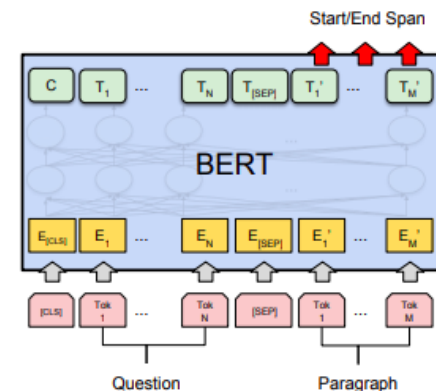
- Text Classification
- Named Entity Recognition
- QA
- NLI



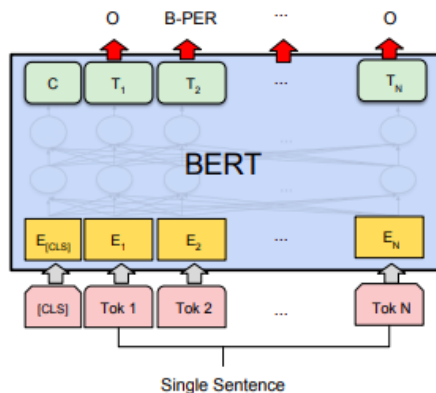
(a) Sentence Pair Classification Tasks:  
MNLI, QQP, QNLI, STS-B, MRPC,  
RTE, SWAG



(b) Single Sentence Classification Tasks:  
SST-2, CoLA



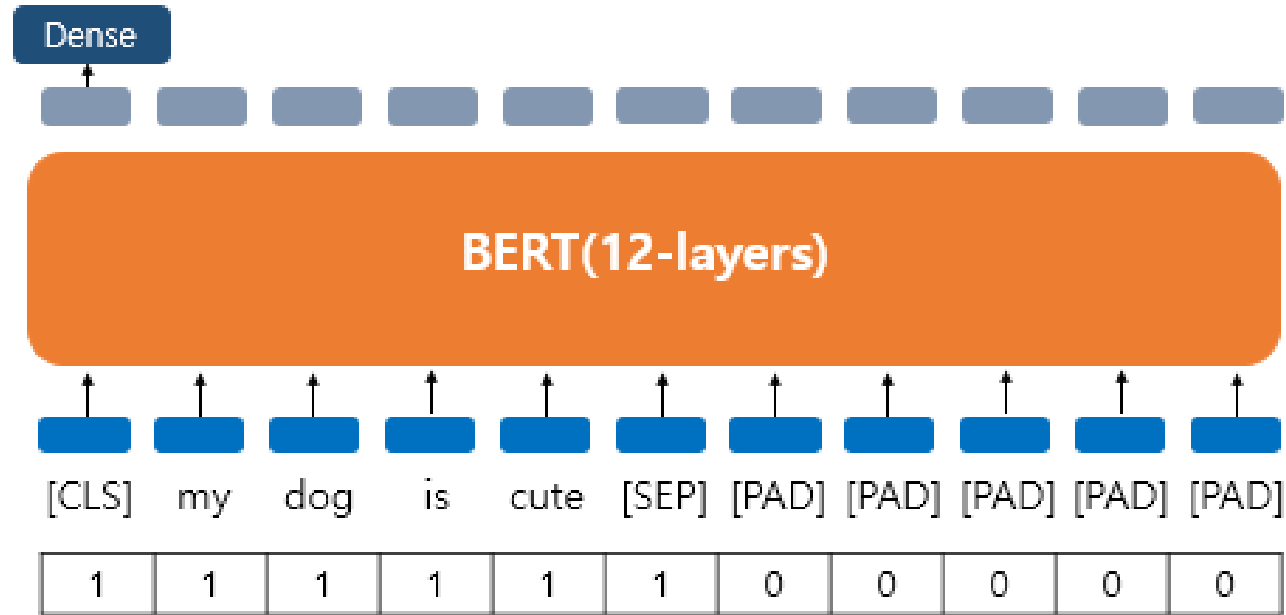
(c) Question Answering Tasks:  
SQuAD v1.1



(d) Single Sentence Tagging Tasks:  
CoNLL-2003 NER



# 어텐션 마스크(Attention Mask)



- 앞서 논문 설명에서는 언급되지 않았지만, 실제 BERT를 사용할 경우에 필요한 입력.
- 숫자 1은 해당 토큰은 실제 단어이므로 마스킹을 하지 않는다는 의미이고, 숫자 0은 해당 토큰은 패딩 토큰이므로 마스킹을 한다는 의미. 위의 그림과 같이 실제 단어의 위치에는 1, 패딩 토큰의 위치에는 0의 값을 가지는 시퀀스를 만들어 BERT의 또 다른 입력으로 사용하면 됩니다.