

# Transformer1

23.02.28 / 8기 업소은

# CONTENTS

---

## 01. RNN Review

- RNN Recap
- Sequence to sequence

## 02. Seq2Seq Model

- Encoder-Decoder

## 03. Attention

- Why Attention?
- Attention Mechanism
- Key, Query, Value

## 04. Transformer

- Encoder
- Decoder

## 05. SUMMARY

## 06. APPENDIX

- Vision Transformer
- BERT

## RNN Recap

RNN, LSTM , GRU ...

여기에 Seq2seq, Encoder-Decoder, Attention, Transformer 까지?

# 01. RNN Review

우선 풀고자 하는 문제, 모델, 모델의 구조를 구분하자!

우리가 풀고자 하는 문제는? : Machine Translation, Question Answering, Automatic Summarization

-> 이것들이 sequence-to-sequence problem임 !

문제를 풀기 위해 사용하는 모델은?: RNN, LSTM, GRU, Encoder-Decoder Model, Transformer

-> 얘네들이 사실상 sequence to sequence model 임 !

모델의 구조를 어떻게 하면 좋을까? : Attention Mechanism

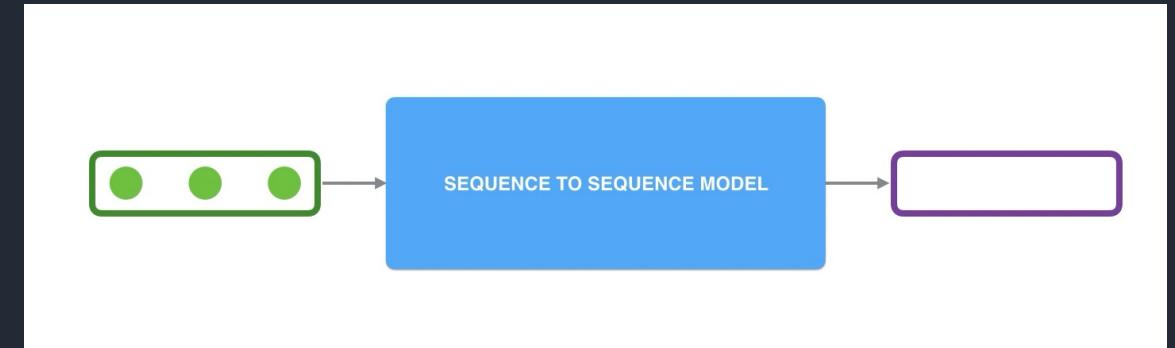
## Sequence to Sequence

Sequence?

-> 단어들의 나열 (=문장, 문단..)

Sequence to Sequence?

-> 하나의 sequence 를 다른 sequence 로 바꾸는 것 !



Ex. Machine Translation : 제 이름은 소은이고, 저는 학생입니다.-> Je m'appelle So-eun et je suis étudiante.

Summarization: 굉장히 긴 text -> 짧은 text

Dialog: 이전 대화 -> 이후에 나올 대화

Sequence to sequence 문제를 어떻게 해결하는데?

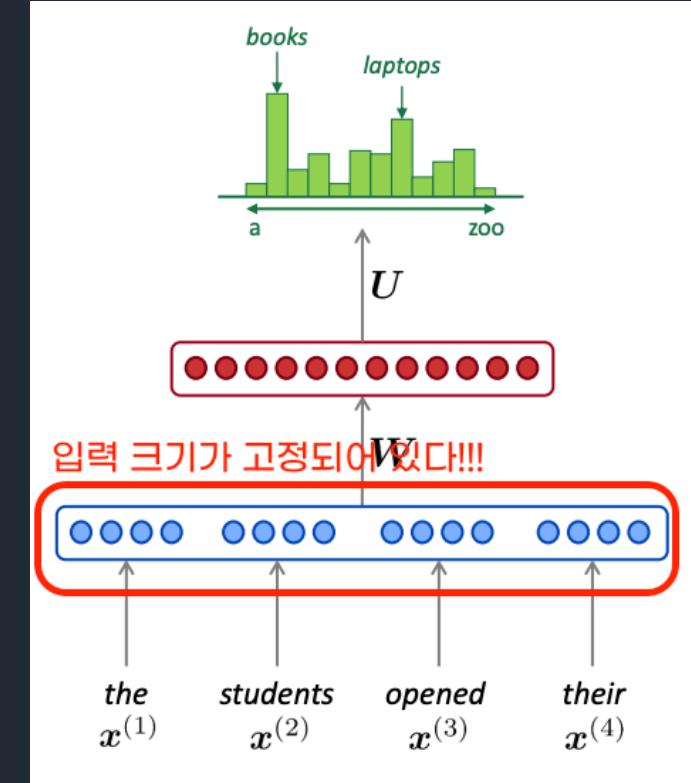
1. Fixed Language Model
2. Vanilla RNN, LSTM, GRU
3. Encoder-Decoder Model
4. Encoder-Decoder Model with Attention
5. Transformer

## RNN 이전의 Language Model

Fixed Window neural Language Model은 Input의 크기가 고정되어 있다!

-> 하지만 문장의 길이는 각양각색인걸?

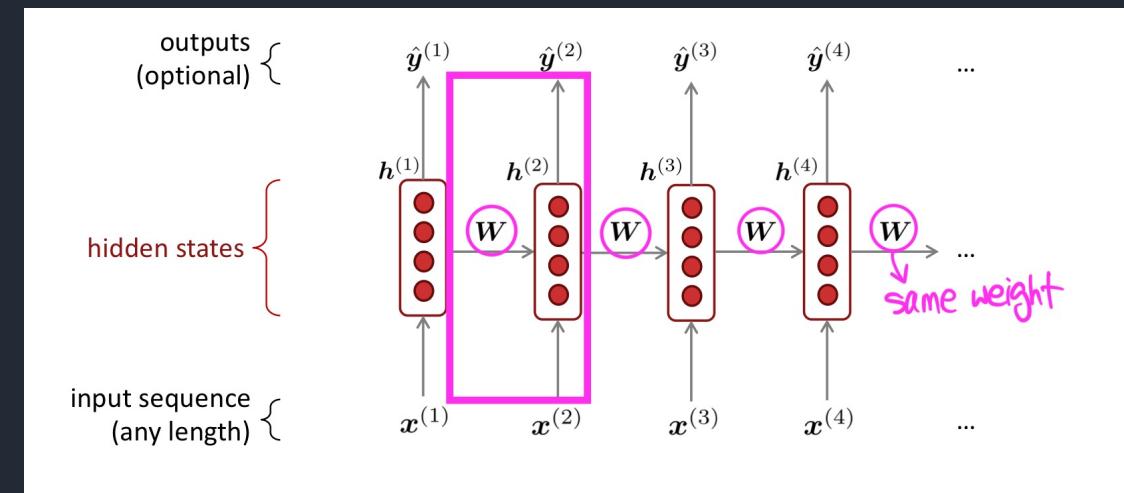
-> 어떠한 input 길이도 처리할 수 있는 neural architecture 가 필요해 !



# 01. RNN Review

## RNN의 등장!

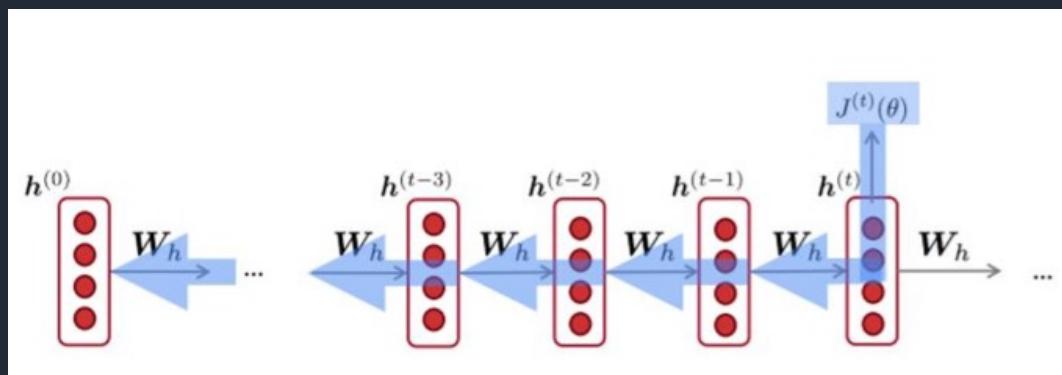
- RNN은 어떠한 input 길이도 처리할 수 있다 !
- RNN은 “메모리를 저장하는 네트워크이다.”
- RNN의 Hidden State 는 이전까지 들어왔던 input에 대한 모든 정보를 저장하고 있다(이론상으로!)
- 각 step마다 같은 weight을 적용해 매 input마다 같은 연산을 하면 된다.



# 01. RNN Review

## RNN의 한계

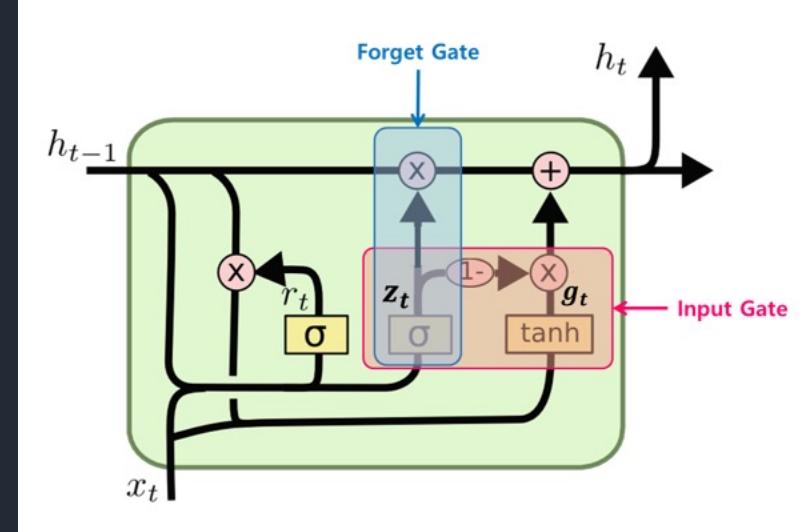
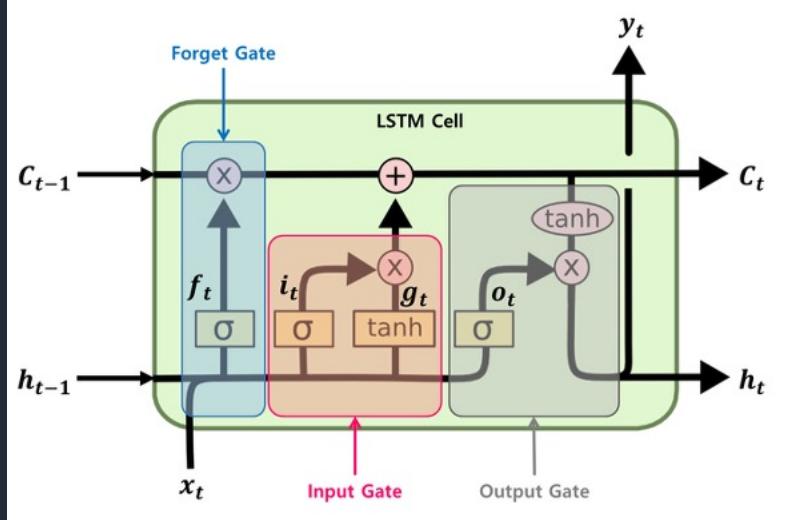
- RNN은 여러 cell들이 stack 되어 있는 구조이다. 즉, 1 dimension 으로 deep 하다.  
-> Loss를 이용하여 Back Propagation으로 가중치를 업데이트 할 때, 앞의 가중치가 업데이트되지 않는 문제가 발생한다!(Gradient Vanishing Problem)  
-> 멀리 떨어져 있는 단어와의 관계는 학습하지 못한다 ! (Long-term Dependency Problem)



Back Propagation through time(BPTT)

# 01. RNN Review

## RNN을 tuning 한 것이 LSTM, GRU

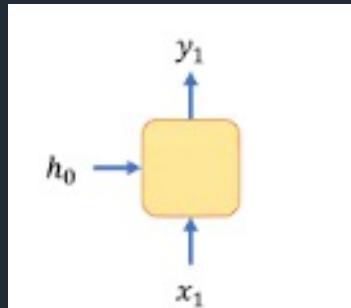


- LSTM: State를 두개 (hidden state, cell state)로 나누어, cell state는 장기 기억, hidden state는 단기 기억을 담당하게 하자 !
- GRU: LSTM에서 parameter 수를 줄여서(Gate 3개 -> 2개, cell state제거) 연산 속도를 빠르게 해보자!

여전히 문제인 것은 얘네도 앞 단어의 정보를 까먹는다는 것이다…!!  
-> 후에 설명할 Attention Mechanism 으로 해결 !

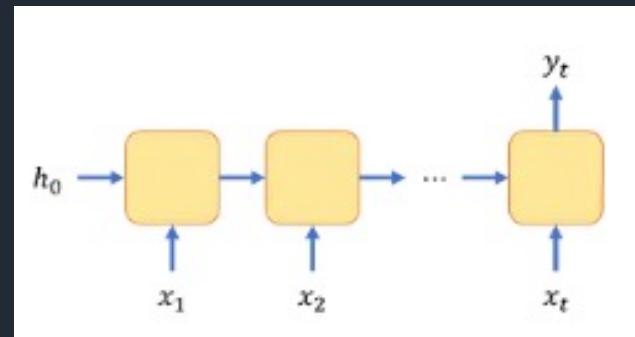
# 01. RNN Review

RNN은 어디에 쓰이는가?



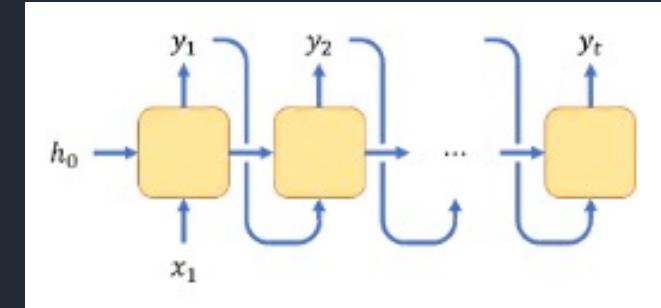
One-to-one

(전통적인 neural Network)



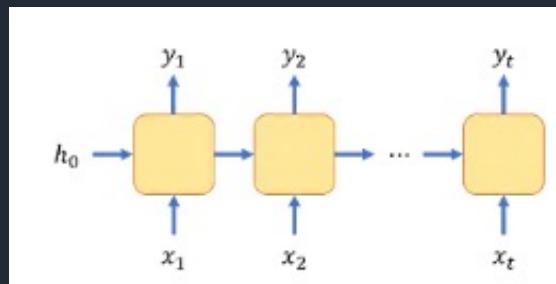
Many-to-one

(Sentiment Classification)

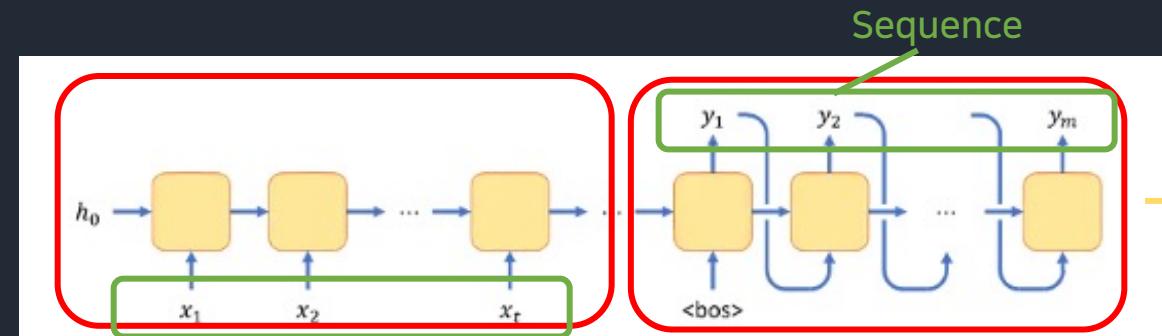


One-to-many

(Music Generation)



Many-to-Many  
(Pos tagging)



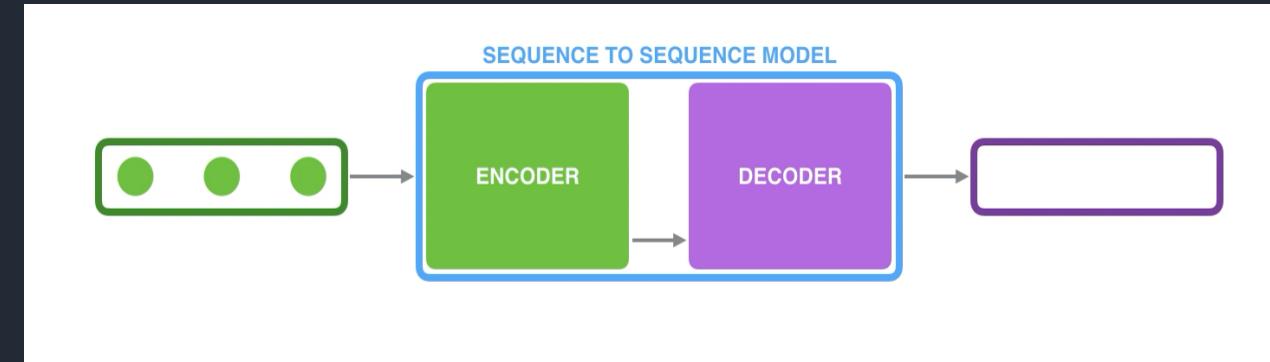
Sequence

Many-to-Many  
(Machine Translation-> Sequence to Sequence Problem)

사실 이게  
Encoder-Decoder  
구조 !!

## Seq2Seq Model = Encoder-Decoder Model

Seq2Seq Model의 뚜껑을 열어보면, 이 모델은  
2개의 RNN\* (Encoder RNN, Decoder RNN)  
으로 이루어져 있다 !

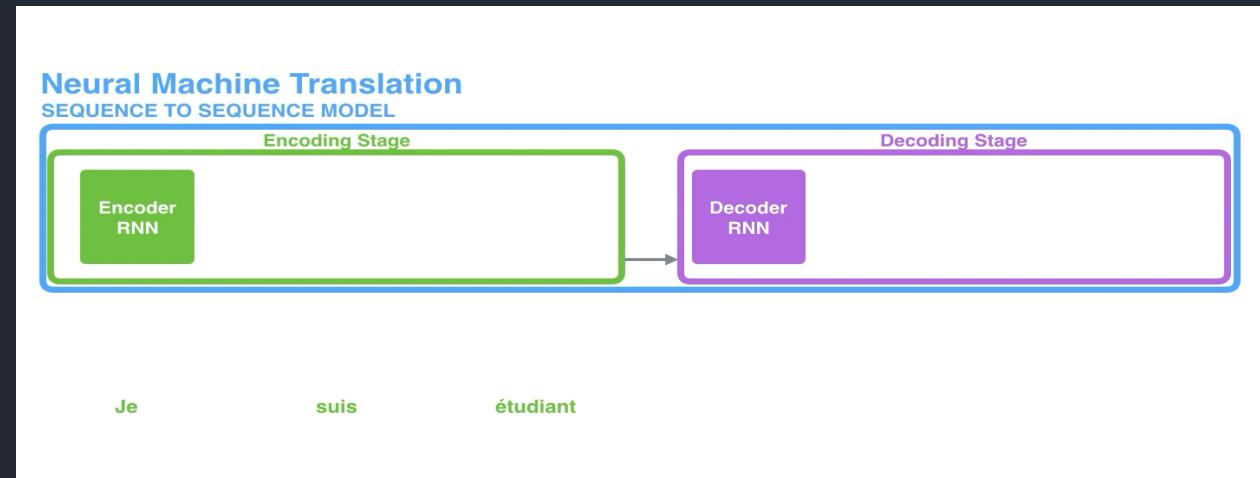


\*RNN으로 쓰긴 했지만, 요즘은 LSTM이 주로 많이 쓰인다.

# 02. Encoder-Decoder Model

## Encoder, Decoder ?

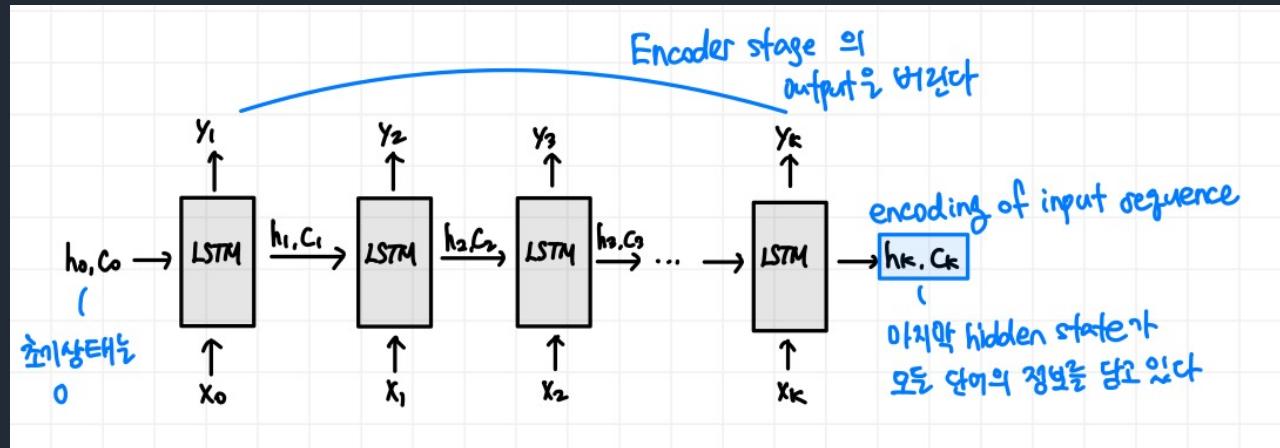
- **Encoder:** Input sequence 를 Hidden representation으로 바꿔준다
- Encoder Stage 내에 있는 마지막 RNN cell이 Decoder Stage에게 초기 Hidden state를 넘겨 준다
- **Decoder:** Hidden representation을 output sequence 로 바꿔준다



[프랑스어] -> [인코더, 디코더 세상의 상상의 언어] -> [영어]로 이해하면 쉽다 !

# 02. Encoder-Decoder Model

## Encoder, Decoder 의 학습 방법



$X_i$  : Input sequence at time step i

$h_i, c_i$  : hidden state, cell state

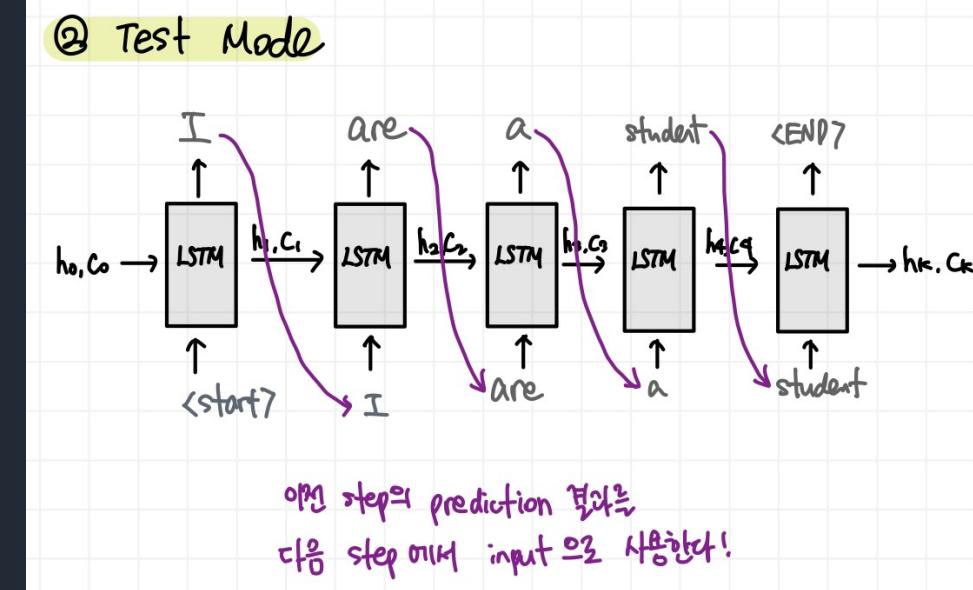
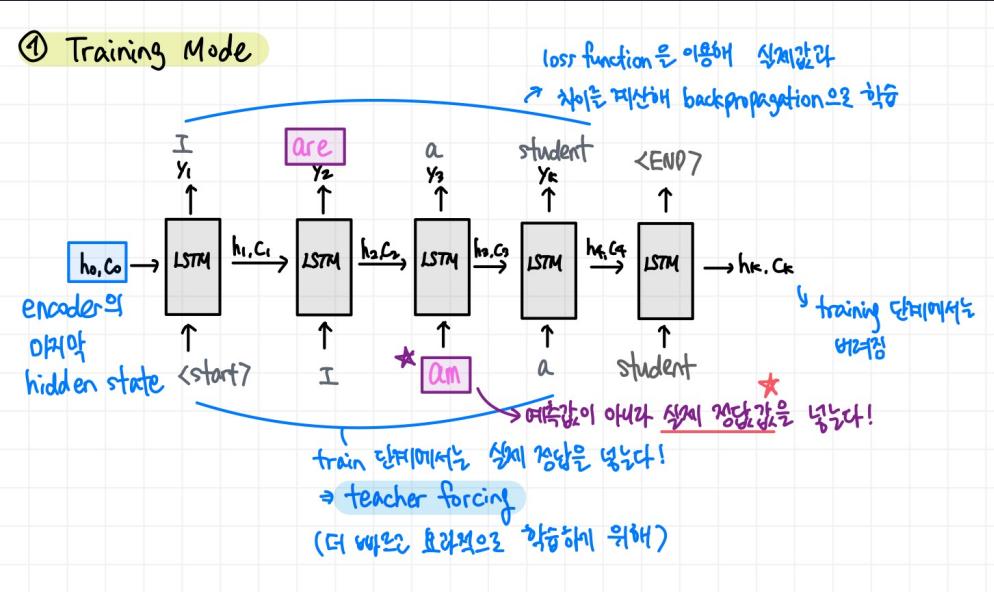
$y_i$  : output sequence at time step i

### <Encoder Stage>

- input 단어들을 전부 읽은 후에 output 을 만드므로, Encoder stage에서 만들어진 output  $y_i$ 는 버린다
  - 마지막  $h_k, c_k$ 가 모든 input sequence 에 대한 정보를 갖고 있다
- > Decoder의 input 값이 된다 !

# 02. Encoder-Decoder Model

## Encoder, Decoder 의 학습 방법



### <Decoder Stage>

- Decoder는 encoder에서 넘겨받은 hidden state로 자신을 초기화한다
- Decoder는 Train mode, Test mode 작동 방식이 다르다
- Train시에는 실제 정답을 넣는 teacher forcing을 사용(더 빠르게 학습하기 위해), Test시는 이전 step의 prediction 결과를 다음 step에서 input으로 사용

## 02. Encoder-Decoder Model

Encoder-Decoder 도 문장이 길어지면 문제가 생긴다…!

- Encoder-Decoder Model도 결국 Encoder가 전달해준 고정된 크기의 마지막 Hidden State에 의존해 Decoder가 output sequence를 만들어낸다
- 하나의 Hidden State에 모든 정보를 다 넣으면 Information Bottleneck 현상이 발생한다!  
(일차선 도로에 차 10대를 꾸겨 넣는 것을 상상해보자)
- 이 구조로는 문장이 길어지면 여전히 앞의 단어를 까먹는다 !
- Input Sequence에 대한 더 많은 정보를 전달해줄 수 없을까? = 모든 Hidden State를 전달한다면?

-> Attention Mechanism의 등장 !

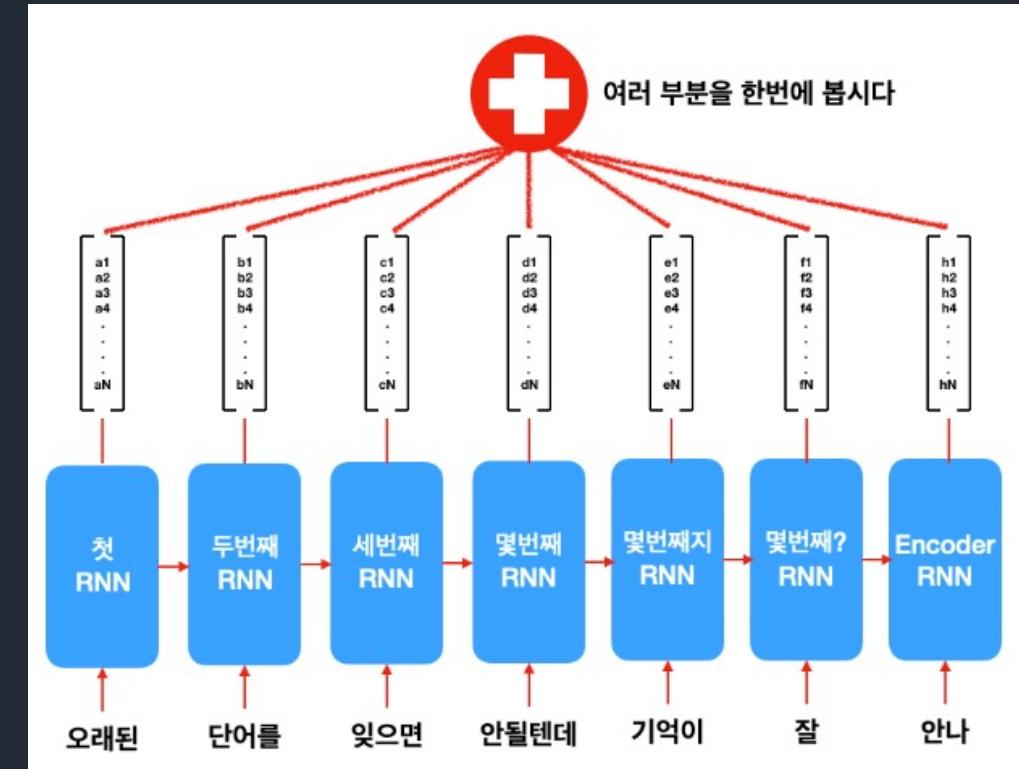
# 03. Attention

Input Sequence 의 모든 부분을 다 바라보자 !

Encoder State에서 모든 Hidden state의 정보를 다 주자 !

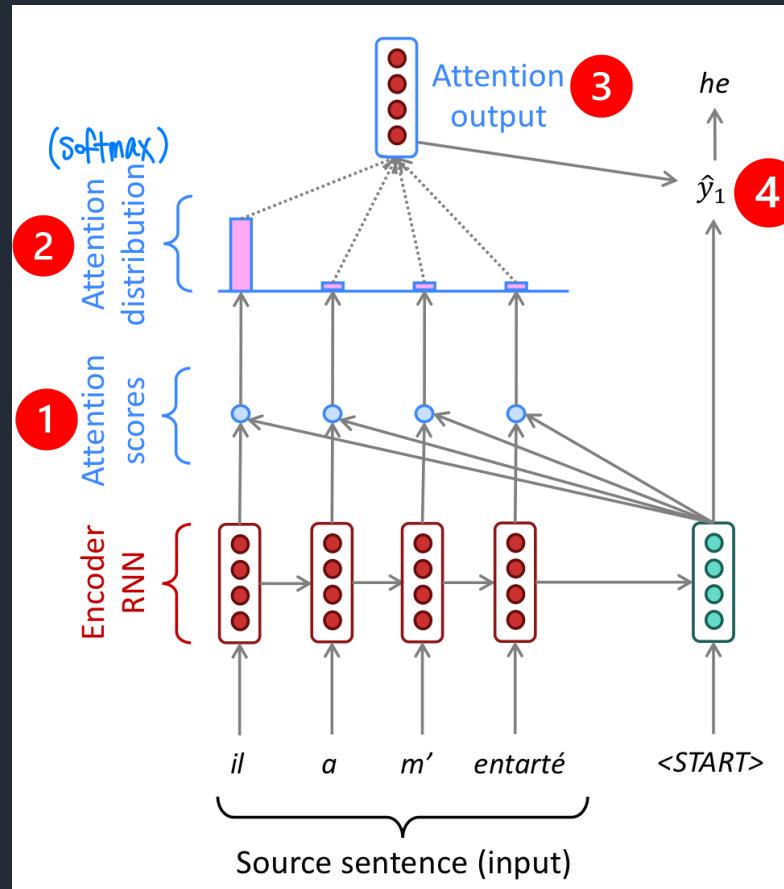
-> 첫번째 단어를 output으로 만들 때는 input의 첫번째  
단어에 더 관심을 가질 수 있도록 하자 !

-> Encoder-Decoder with Attention



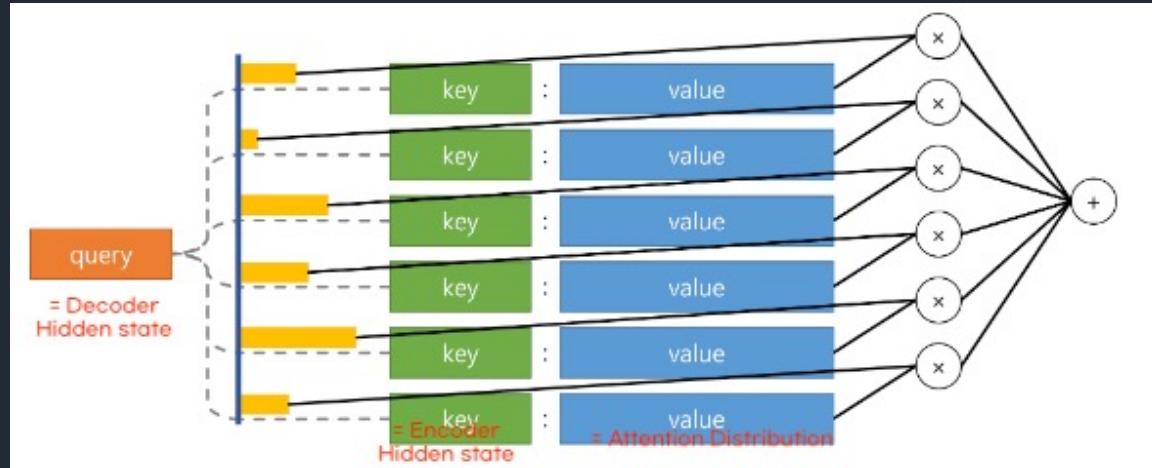
# 03. Attention

## Attention 은 어떻게 계산하나?



1. Decoder의 hidden state와 Encoder 의 hidden states들을 비교하여 Attention score를 계산한다.
2. SoftMax를 사용하여 합하면 1이 되는 Attention distribution을 구한다.
3. Attention output은 Attention distribution의 weighted sum이다.  
-> 가장 연관이 높은 encoder state에 대한 정보가 가장 많다
4. Attention output과 Decoder hidden state를 합쳐  $y_1$  을 만든다.

## Query, Key , Value?



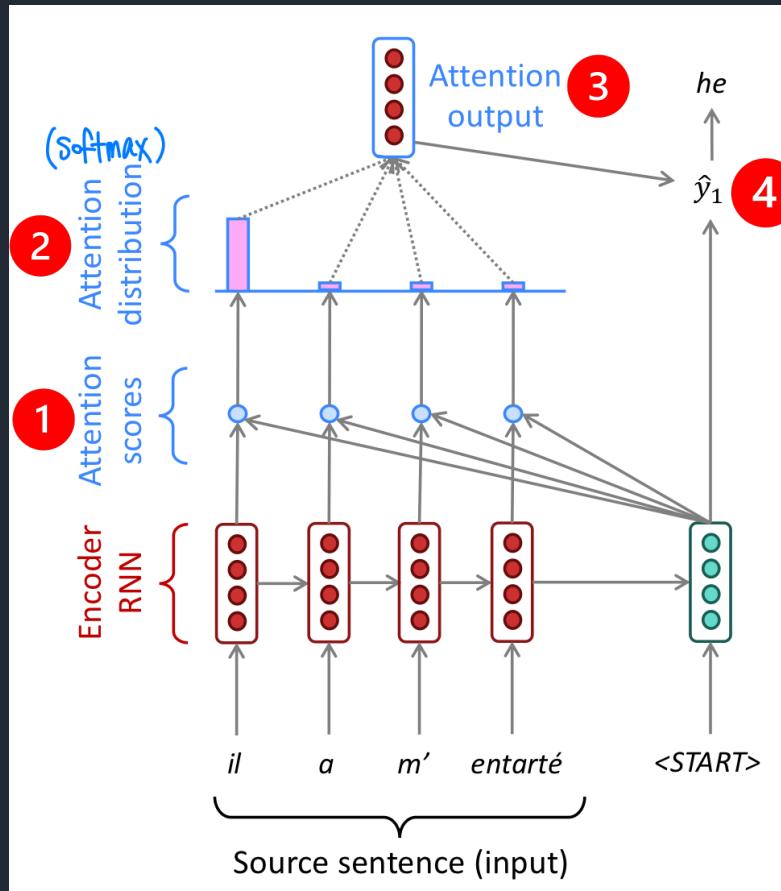
- Query: 사용자가 찾고 싶은 값  
-> Decoder Hidden state
- Key: Query와 유사도를 비교하고 싶은 대상  
-> Encoder Hidden state
- Value: Key에 대응되는 실제 값  
-> Attention Distribution

Attention 연산은 어떤 value에 '집중'할지 결정하는 것이다.

Attention 연산이 수행되면, 중요한(= query와 유사도가 높은 key를 가진) value에 더 '집중'하게 된다.

# 03. Attention

Query, Key , Value 를 이용해 다시 서술해보자 !



1. Decoder의 Query vector와 Encoder의 Key vector를 내적해 Attention score를 계산한다.
2. SoftMax 함수를 사용하여, query 와 key들 간의 Attention score를 Attention distribution(value에 해당)으로 변환한다.
3. Attention distribution을 이용해 각 value들의 weighted sum을 계산한다. -> Attention output
4. Attention output과 decoder hidden state를 합쳐(concat)  $y_1$ 을 만든다.

# 03. Attention

## RNN은 한쪽밖에 모르는 바보야..

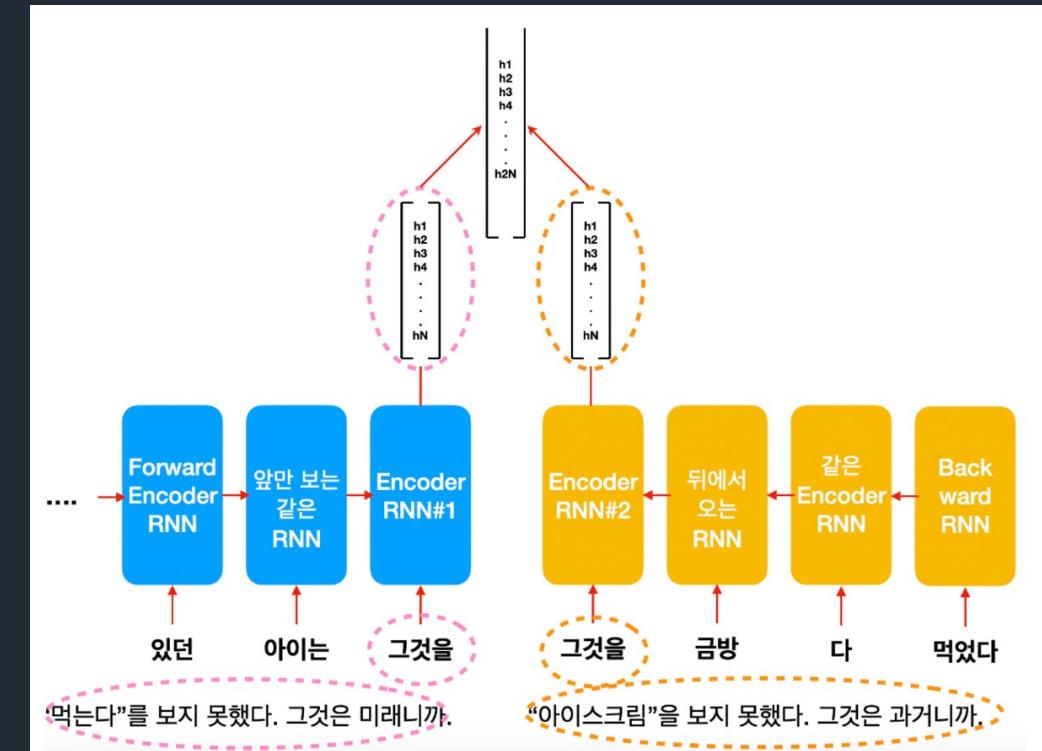
- Encoder-Decoder 구조에 Attention을 적용했지만, 여전히 Encoder-Decoder 안에서는 RNN을 쓴다.
  - RNN은 구조상 sequence를 순차적으로 입력 받는다.
- > '그것'이 '아이스크림'이라는 것을 알려면, 뒤에 나오는 단어인 '먹는다'까지 고려 해야 하는데.. RNN은 미래를 보지 못한다 !(바보)



# 03. Attention

## 앞에서부터 보는 RNN, 뒤에서부터 보는 RNN, 두개를 사용하면?

- 문장을 앞에서부터 보는 forward RNN, 뒤에서부터 보는 backward RNN 2개의 hidden state를 엮어서 보자!  
-> Bi-directional RNN
- 이것은 유명한 다른 모델인 ELMO의 방식
- 이 방식의 장점은 word embedding 때 나타난다  
-> 문맥에 따라 다른 의미를 가지는 단어들의 embedding 값을 다르게 학습시킬 수 있다  
(Contextualized Word Embedding)

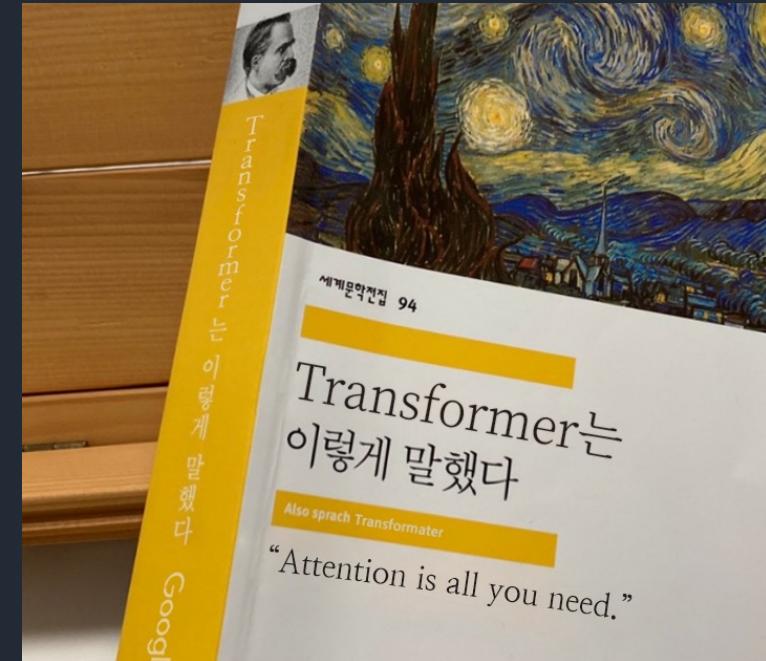


## Attention is All you Need

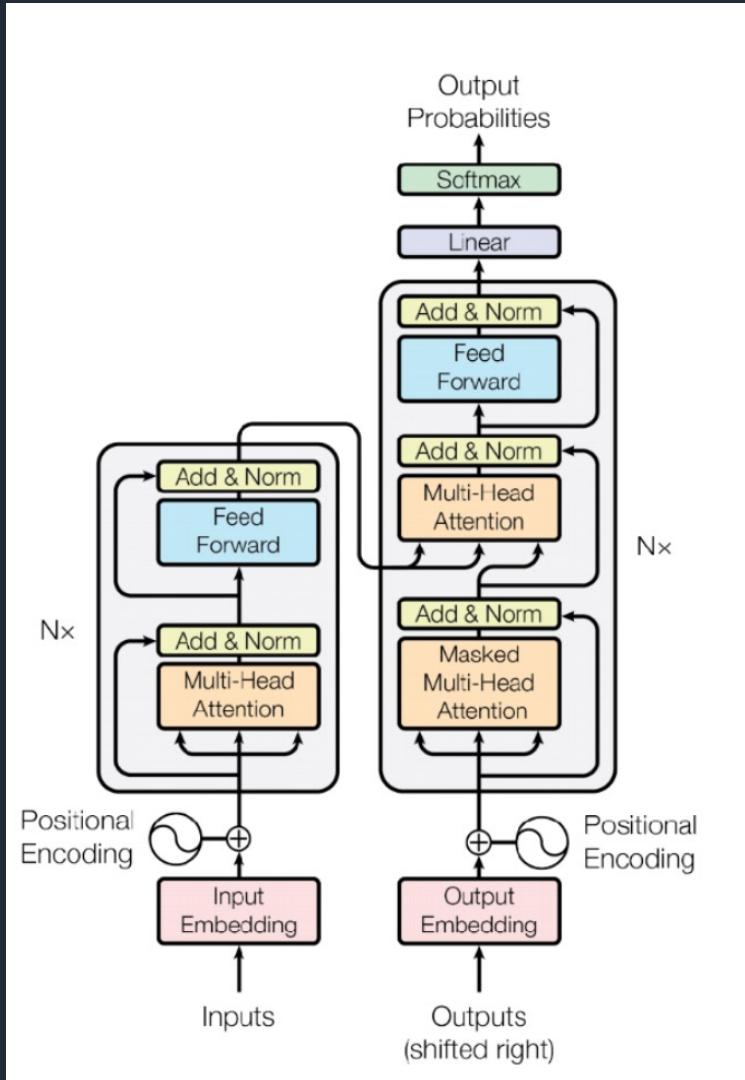
하지만.. 애초에 Attention으로 모든 단어에 관심을 가지는데,  
Attention 만으로도 문장을 모델링할 수 있지 않나?

- > 굳이 컨베이어 벨트 같은 RNN구조를 쓸 필요가 있을까?
- > Attention 만 있으면 되는거 아니야?
- > Attention is All you Need !

Attention 구조만 사용한 Transformer 모델 등장 !

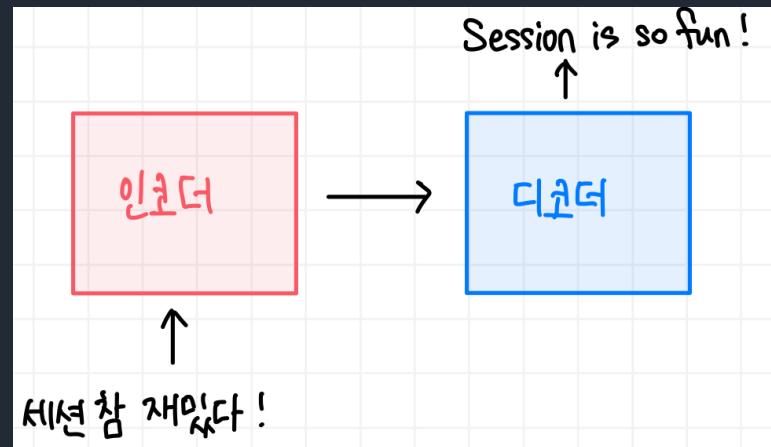


# 04. Transformer

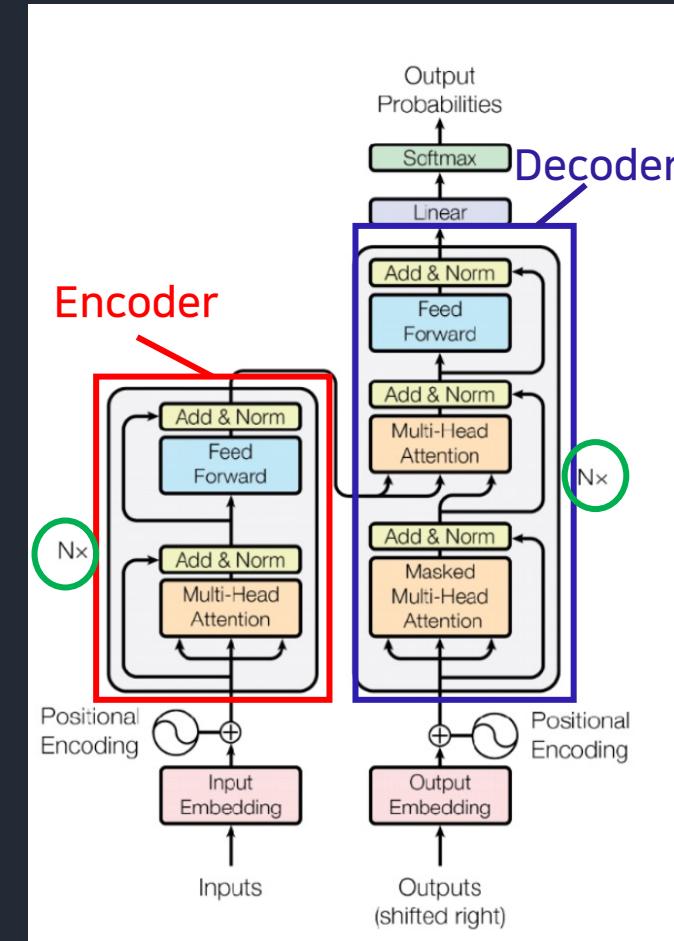


# 04. Transformer

# Transformer도 Encoder-Decoder 모델이다

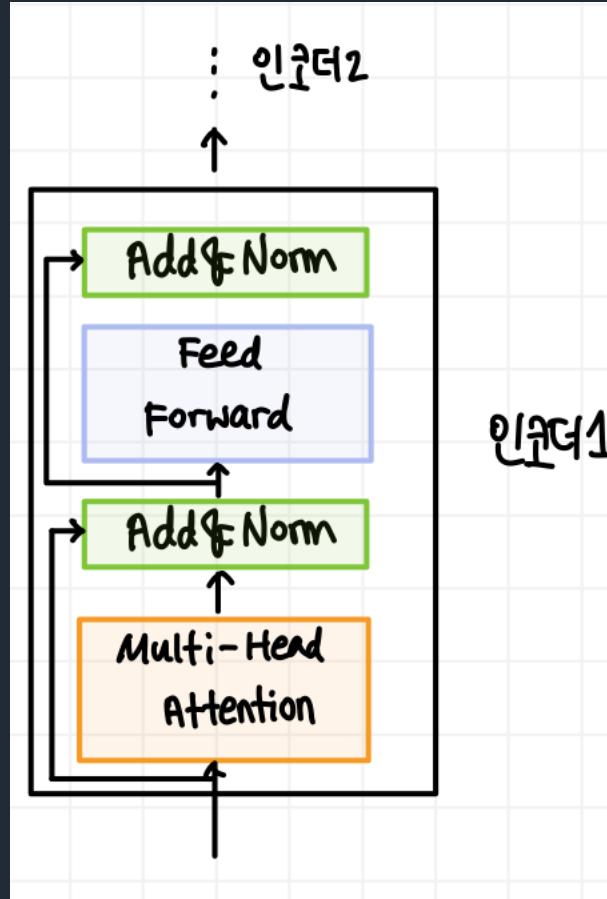
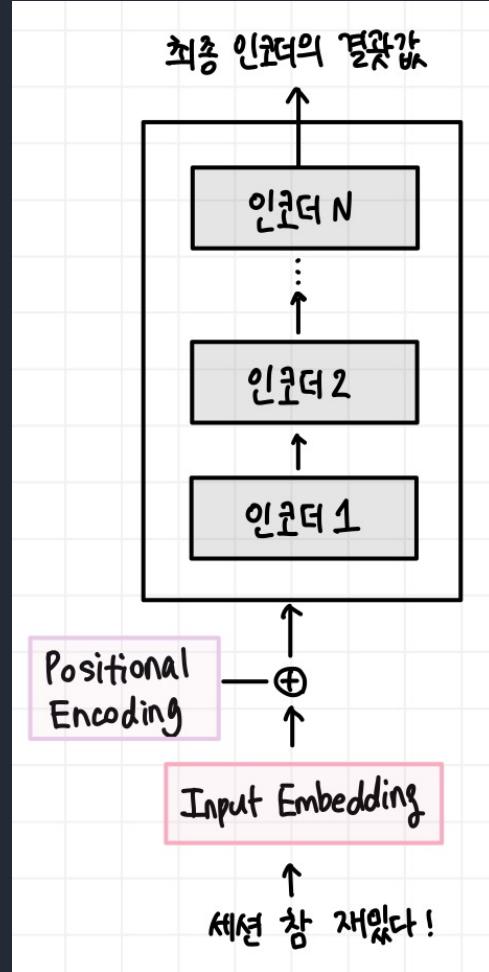


- Transformer 도 Encoder-Decoder 로 이루어진 모델이다.
  - Encoder N개, Decoder N개 를 누적해서 쌓아올린 형태이다.
  - RNN구조를 쓰지 않으니, 순차적으로 sequence를 입력하지 않고, Input Embedding을 하여 한번에 넣는다.



# 04. Transformer

## Transformer의 Encoder 부분부터 보자



- 모든 Encoder 구조는 동일한 구조로 되어 있다 (= Encoder의 Input, Output size가 동일하다)
- Encoder 내부는 Multi-Head Attention과 Feed Forward , Add & Norm 으로 구성되어 있다

# Positional Encoding

아까 Input Embedding을 순차적으로 넣지 않고, 한번에 넣는다고 했다.

그럼 단어의 순서는 어떻게 알 수 있는데? -> Positional Encoding

$$X = \text{am} \begin{bmatrix} 1.7 & 2.3 & 3.4 & 5.8 \\ 7.3 & 9.9 & 8.5 & 1.1 \\ 9.1 & 7.1 & 0.85 & 10.1 \end{bmatrix} z_1 + \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0.8 & 0.5 & 0.01 & 0.99 \\ 0.9 & -0.4 & 0.02 & 0.09 \end{bmatrix} z_2 + \begin{bmatrix} X \\ P \end{bmatrix}$$

embedding matrix [3x4]      positional encoding  
[문장길이 x dmodel]

Embedding matrix와 같은 차원의 positional encoding matrix를 더해준다

## Positional Encoding

Positional Encoding 계산 방법

-> 사인파 함수(sinusoidal function) 이용

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

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

$i$ : 차원의 index

$pos$ : 문장에서 단어의 위치

- sin, cos 함수는 -1~1을 반복하는 주기함수  
-> 긴 문장이 주어져도 위치 벡터 값의 차가 작아지지 않는다
- 주기함수면 다른 위치인데도 값이 같아지는 경우가 생기지 않나?  
-> 다양한 주기의 sin, cos 함수를 사용  
(하나의 positional encoding 벡터가 4개의 차원으로 표현된다면,  
각 차원은 서로 다른 4개의 주기를 갖게 된다!)

Multi-Head Attention을 이해하려면 Self Attention 부터 알아야 한다

A dog ate the food because it was hungry.

한 문장 내에서, it이 food가 아니라 dog라는 것을 어떻게 알 수 있을까?

-> Self Attention

The diagram shows the same sentence as above, but with blue curved arrows indicating attention weights between words. Arrows originate from the word 'it' and point to each word in the sentence: 'dog', 'ate', 'the', 'food', 'because', and 'hungry'. This visualizes how the word 'it' attends to all other words in the sentence to determine its meaning.

각 단어의 표현들을 문장 안에 있는 다른 모든 단어의 표현과 연결해  
문장 내에서 갖는 의미를 이해한다.

# 04. Transformer

## Self Attention의 수식적 계산

입력 문장 'I am good' →  
Input  
Embedding

I에 대한 embedding  $\mathbf{x}_1 = [1.76, 2.22, \dots, 6.66]$   
am에 대한 embedding  $\mathbf{x}_2 = [2.77, 0.36, \dots, 4.32]$   
good에 대한 embedding  $\mathbf{x}_3 = [10.45, 3.52, \dots, 0.76]$

논문에서는 512 차원으로  
embedding  $\mathbf{x}$ 의  $d_{model}$

$\begin{matrix} I & 1.76 & 2.22 & \cdots & 6.66 \\ am & 2.77 & 0.36 & \cdots & 4.32 \\ good & 10.45 & 3.52 & \cdots & 0.76 \end{matrix} \xrightarrow{\text{embedding 행렬}} \mathbf{x} \quad [3 \times 512] \quad [문장길이 \times d_{model}]$

→  $\mathbf{x}$ 에  $W^Q, W^K, W^V$  가중치 행렬을 곱해  
Query, Key, Value 행렬 생성  
→  $W^Q, W^K, W^V$ 는 처음에 임의값을 가지며,  
학습 과정에서 최적값으로 업데이트된다.

$\begin{matrix} I & 1.76 & 2.22 & \cdots & 6.66 \\ am & 2.77 & 0.36 & \cdots & 4.32 \\ good & 10.45 & 3.52 & \cdots & 0.76 \end{matrix} \xrightarrow{\text{embedding 행렬}} \mathbf{x} \quad [3 \times 512] \quad [문장길이 \times d_{model}]$

$\mathbf{x} \cdot W^Q = Q \quad [3 \times 64] \quad [문장길이 \times d_q]$

$\mathbf{x} \cdot W^K = K \quad [3 \times 64] \quad [문장길이 \times d_k]$

$\mathbf{x} \cdot W^V = V \quad [3 \times 64] \quad [문장길이 \times d_v]$

$\begin{matrix} I & 2.69 & 7.42 & \cdots & 1.42 \\ am & 2.68 & 3.52 & \cdots & 6.14 \\ good & 9.23 & 11.32 & \cdots & 8.16 \end{matrix} Q_1 \quad \begin{matrix} I & 2.32 & 5.14 & \cdots & 6.14 \\ am & 1.14 & 2.92 & \cdots & 10.11 \\ good & 9.52 & 7.76 & \cdots & 13.19 \end{matrix} K_1 \quad \begin{matrix} I & 1.14 & 11.14 & \cdots & 10.19 \\ am & 4.42 & 1.56 & \cdots & 15.39 \\ good & 3.41 & 2.19 & \cdots & 0.17 \end{matrix} V_1$

$\begin{matrix} I & 2.69 & 7.42 & \cdots & 1.42 \\ am & 2.68 & 3.52 & \cdots & 6.14 \\ good & 9.23 & 11.32 & \cdots & 8.16 \end{matrix} Q_2 \quad \begin{matrix} I & 2.32 & 5.14 & \cdots & 6.14 \\ am & 1.14 & 2.92 & \cdots & 10.11 \\ good & 9.52 & 7.76 & \cdots & 13.19 \end{matrix} K_2 \quad \begin{matrix} I & 1.14 & 11.14 & \cdots & 10.19 \\ am & 4.42 & 1.56 & \cdots & 15.39 \\ good & 3.41 & 2.19 & \cdots & 0.17 \end{matrix} V_2$

$\begin{matrix} I & 2.69 & 7.42 & \cdots & 1.42 \\ am & 2.68 & 3.52 & \cdots & 6.14 \\ good & 9.23 & 11.32 & \cdots & 8.16 \end{matrix} Q_3 \quad \begin{matrix} I & 2.32 & 5.14 & \cdots & 6.14 \\ am & 1.14 & 2.92 & \cdots & 10.11 \\ good & 9.52 & 7.76 & \cdots & 13.19 \end{matrix} K_3 \quad \begin{matrix} I & 1.14 & 11.14 & \cdots & 10.19 \\ am & 4.42 & 1.56 & \cdots & 15.39 \\ good & 3.41 & 2.19 & \cdots & 0.17 \end{matrix} V_3$

# 04. Transformer

## Self Attention의 수식적 계산

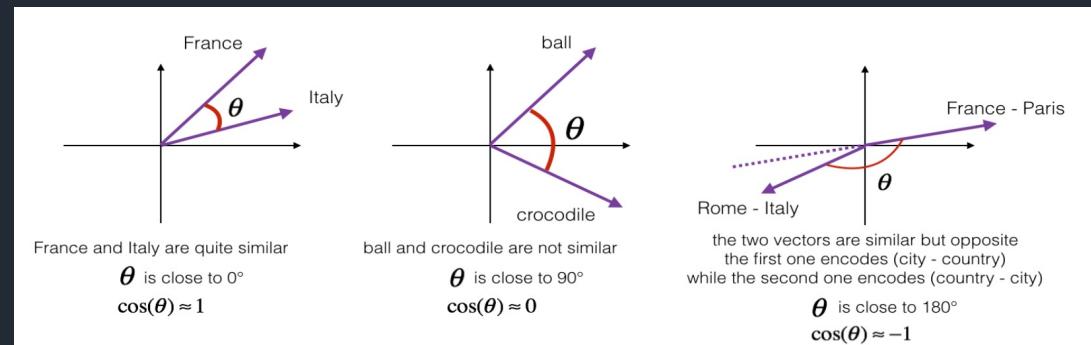
Query, Key, Value 를 이용해 특정 단어와 문장 내에 있는 모든 단어간의 관계를 보는 방법은?

Step1.

Query 행렬( $Q$ )와 Key행렬( $K^T$ )간의 내적 연산  $\longrightarrow$   
으로  $Q, K$ 간의 유사도를 구한다

$$\begin{aligned} & Q \cdot K^T \\ & \begin{bmatrix} I \\ am \\ good \end{bmatrix} \begin{bmatrix} 3.69 & 7.42 & \dots & 1.42 \\ 2.32 & 1.14 & & 3.52 \\ 5.14 & 2.32 & \dots & 7.78 \\ \vdots & \vdots & & \vdots \\ 6.14 & 10.11 & & 13.19 \end{bmatrix} \\ & \quad [3 \times 5(2)] \quad \cdot \quad [5 \times 2 \times 3] \\ & = \begin{bmatrix} I & am & good \\ am & & \\ good & & \end{bmatrix} \quad [3 \times 3] \end{aligned}$$

왜 내적 연산을 하면 벡터간의 유사도를 알 수 있을까?



두 벡터 사이의 각  $\theta$ 가 작으면 (=두 벡터의 방향이 비슷하면)  
두 벡터 사이의 내적 값이 커진다!

# 04. Transformer

## Self Attention의 수식적 계산

Step 2.

$Q \cdot K^T$  행렬을 key 차원 벡터의 제곱근 값으로 나누어준다.

$$\frac{Q \cdot K^T}{\sqrt{d_k}}$$

- > dimension이 크면,  $Q \cdot K^T$  의 값이 너무 커진다.
- > 후에 SoftMax함수에 넣을 때, 분포가 too sparse 해진다.
- > Gradient Vanishing 문제가 생긴다 !

따라서 이러한 문제가 생기는 것을 막기위해,  $\sqrt{d_k}$  로 나누어준다.

$$\frac{QK^T}{\sqrt{d_k}} = \frac{Q \cdot K^T}{\sqrt{d_k}} = \begin{matrix} \text{I} & \text{am} & \text{good} \\ \text{I} & [4.5 & 11.3 & 5.4] \\ \text{am} & [8.7 & 12.3 & 6.5] \\ \text{good} & [10.2 & 7.4 & 3.5] \end{matrix}$$

# 04. Transformer

## Self Attention의 수식적 계산

Step 3.

SoftMax함수를 써워 0~1 사이의 분포로 정규화 시킨다.

이와 같은 형태로 얻어진 행렬을 score matrix 라 한다.

옆의 예시에서는, I는 자기자신과 90%, am과 7%, good과 3% 관련되어 있다는 것을 알 수 있다.

$$\text{Softmax}\left(\frac{\mathbf{Q} \cdot \mathbf{K}^T}{\sqrt{d_k}}\right) = \begin{matrix} \begin{matrix} I & am & good \end{matrix} \\ \begin{matrix} I \\ am \\ good \end{matrix} \end{matrix} \begin{bmatrix} 0.9 & 0.07 & 0.03 \\ 0.025 & 0.95 & 0.025 \\ 0.21 & 0.03 & 0.76 \end{bmatrix}$$

# 04. Transformer



## Self Attention의 수식적 계산

Step 4.

앞서 구한 score matrix에 Value 행렬을 곱하여 Attention 행렬(Z)를 구한다.

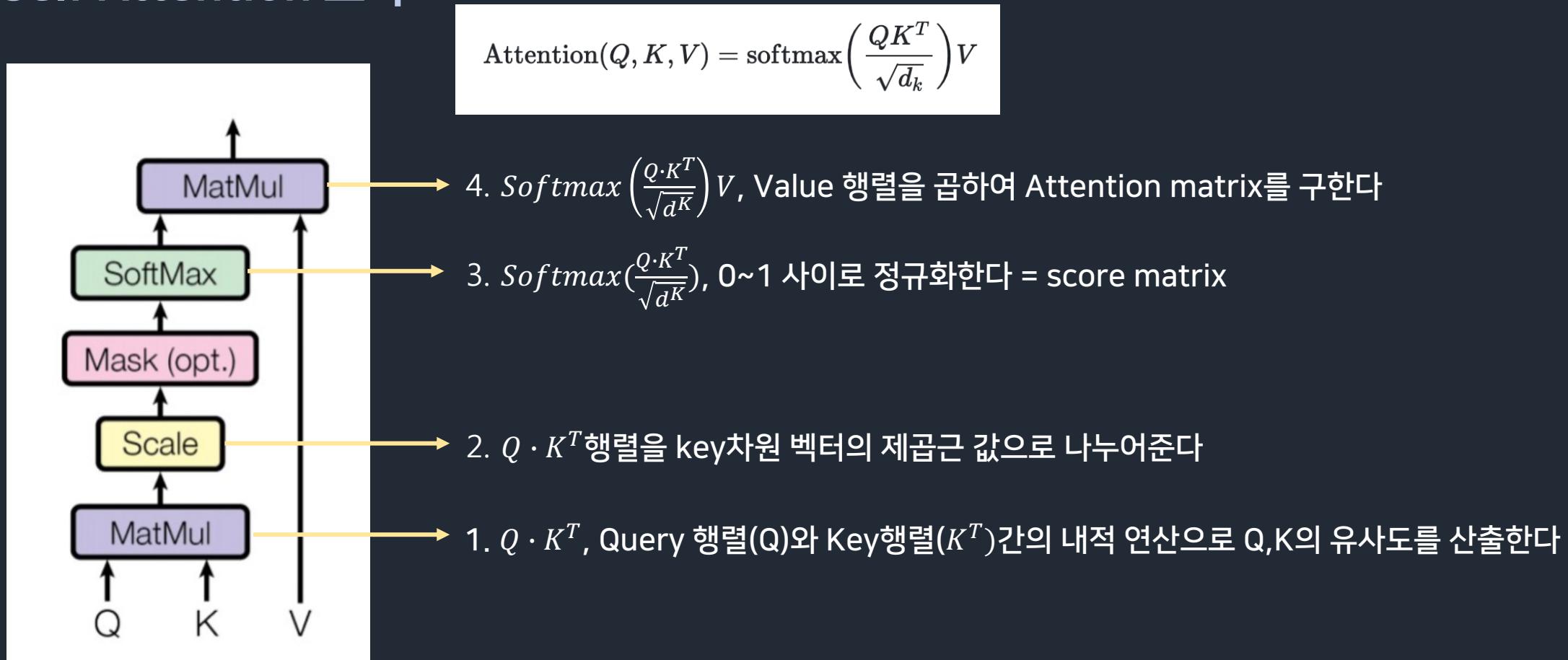
Z는 각 단어의 Value vector값의 가중치 합으로 계산된다.

Attention (Z) = 유사도를 나타내는 점수  $\cdot$  실제 단어들의 임베딩 값

$$\text{Attention (Z)} = \begin{bmatrix} I & am & good \\ I & 0.9 & 0.07 & 0.03 \\ am & 0.025 & 0.95 & 0.025 \\ good & 0.21 & 0.03 & 0.76 \end{bmatrix} \cdot \begin{bmatrix} I & 1.14 & 11.14 \cdots & 10.19 \\ am & 4.42 & 1.56 \cdots & 15.39 \\ good & 3.41 & 2.19 \cdots & 0.19 \end{bmatrix} V_1 \\ V_2 \\ V_3$$
$$\text{Softmax}\left(\frac{Q \cdot K^T}{\sqrt{d_k}}\right) \cdot V$$
$$[3 \times 64] = [3 \times 3] \cdot [3 \times 64]$$
$$Z = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} \quad \begin{aligned} z_1 &= 0.9 [1.14 \quad 11.14 \cdots \quad 10.19] + 0.07 [4.42 \quad 1.56 \cdots \quad 15.39] + 0.03 [3.41 \quad 2.19 \cdots \quad 0.19] \\ &\quad \text{---} \\ &\quad \text{---} \\ &\quad \text{---} \end{aligned}$$

$\Rightarrow$  I의 self-Attention

## Self Attention 요약



$\sqrt{d_K}$  로 나눠주는 작업 때문에 Scaled Dot-product Attention이라고도 한다!

## Multi-Head Attention

문장의 의미가 애매해서 score matrix가 이상하게 계산되면 어떡하지?

A dog ate the food because it was hungry.

$$z_{it} = 0.04 V_1(A) + 0.43 V_2(\text{dog}) + \dots + 0.49 V_5(\text{food}) + \dots + 0.01 V_9(\text{hungry})$$

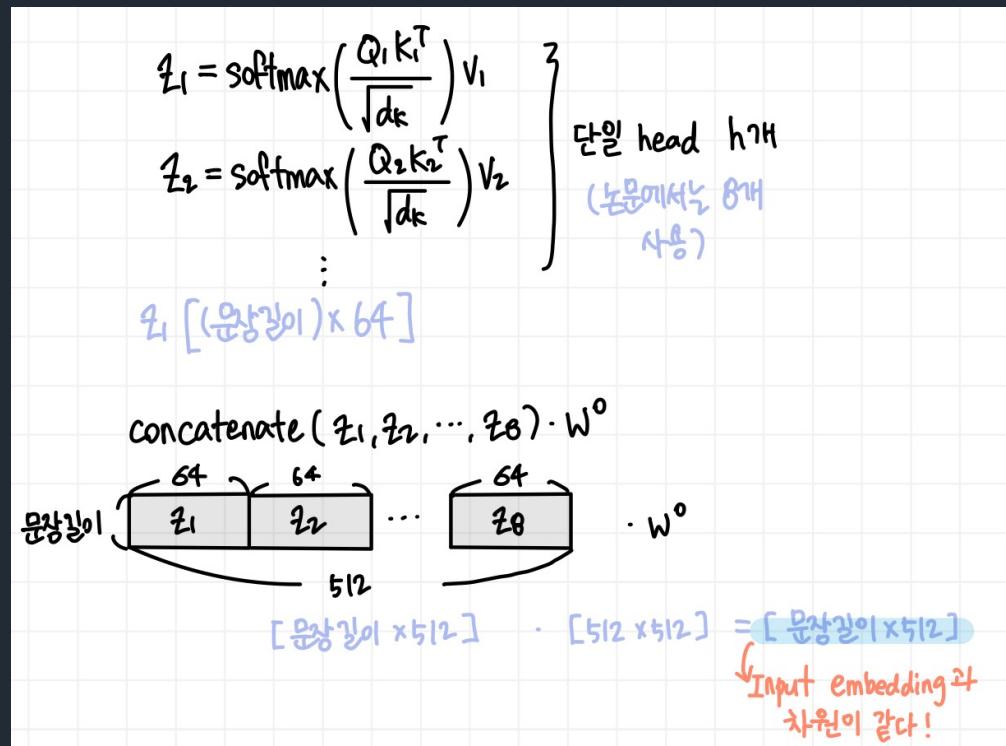
... 잘못 해석될 위험이 있다!

단일 Attention만을 사용한다면 잘못 해석해 food에 가중치를 높게 두면 의미가 잘못 연결될 수 있다!

-> 여러 번, 다른 각도(=다른 score matrix)로 해석하는 Multi-Head Attention을 쓰자!

# 04. Transformer

## Multi-Head Attention



*Multi - Head Attention = concatenate( $Z_1, Z_2, \dots, Z_h$ ) ·  $W_o$*

계속 차원을 표시했는데, 차원을 이해하는 것이 중요하다!

$$d_k = d_v = \frac{d_{model}}{h} \text{ 이다}$$

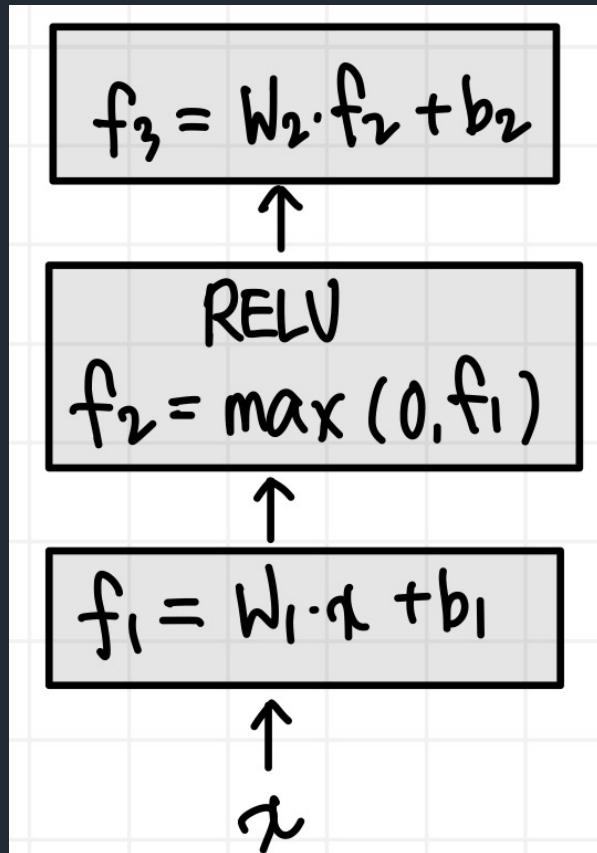
$d_k$  : key의 차원

$d_v$  : value의 차원

$d_{model}$  : 단어가 embedding 된 차원

$h$ : head 의 수

## Feed Forward Network



ReLU 함수로 non-linearity(비선형성) 을 더해준다!

-> 선형 함수를 계속해서 쌓으면, 결국 한개의 선형 함수로 나타내는 것과  
마찬가지이다. 그러면 layer를 쌓는 의미가 없다.

$$y_1 = ax_1$$

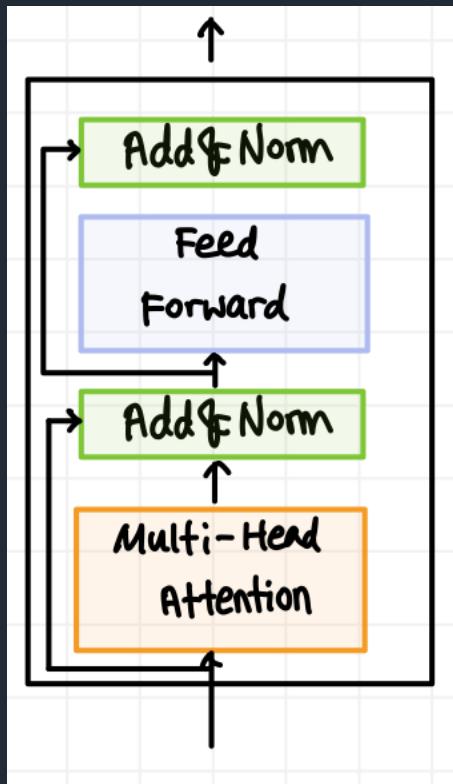
$$y_2 = a(y_1) = a^2x_1$$

$$y_3 = a(y_2) = a^3x_1$$

$$y_n = bx_1, b = a^n$$

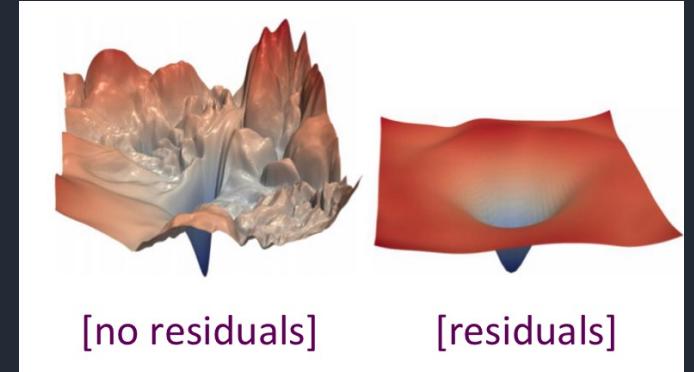
# 04. Transformer

## Add & Norm



### Add

- Multi-Head Attention 의 입력값과 출력값을 더한다.
  - Feed Forward의 입력값과 출력값을 더한다.
- > Residual Connection :  $X^i = X^{i-1} + \text{Layer}(X^{i-1})$

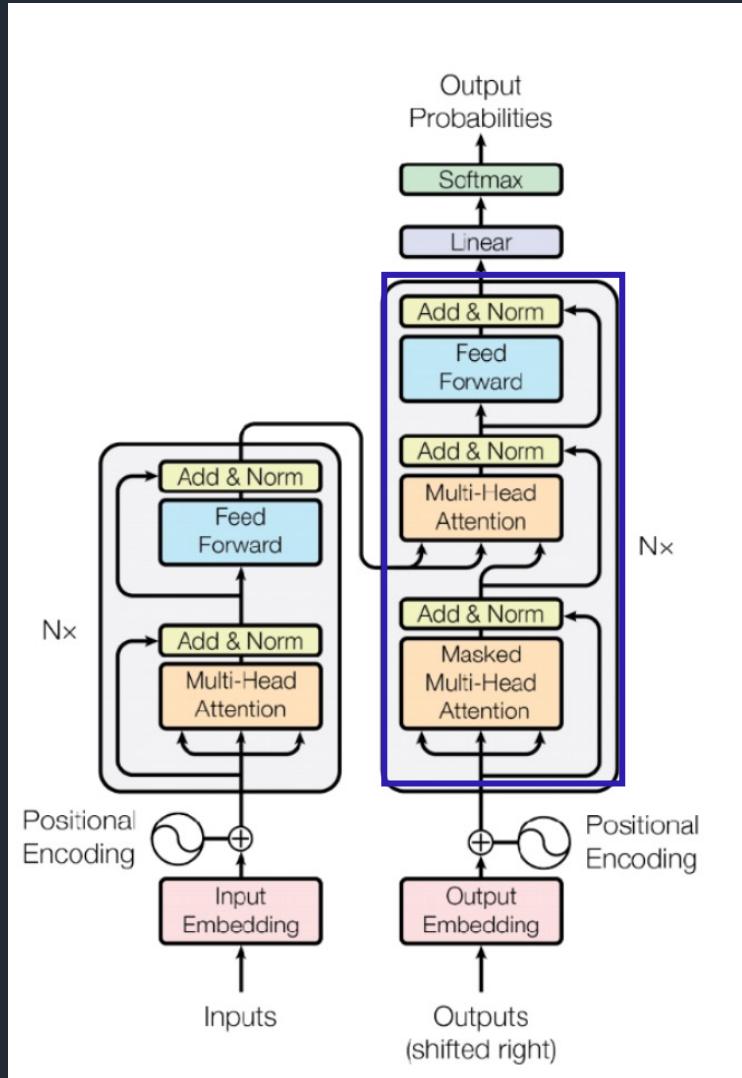


### Norm

- Layer Normalization은 각 layer값이 크게 변화하는 것을 방지해 모델이 더 빠르게 학습할 수 있게 한다.

$$\text{output} = \text{LayerNorm}(x + \text{sublayer}(x))$$

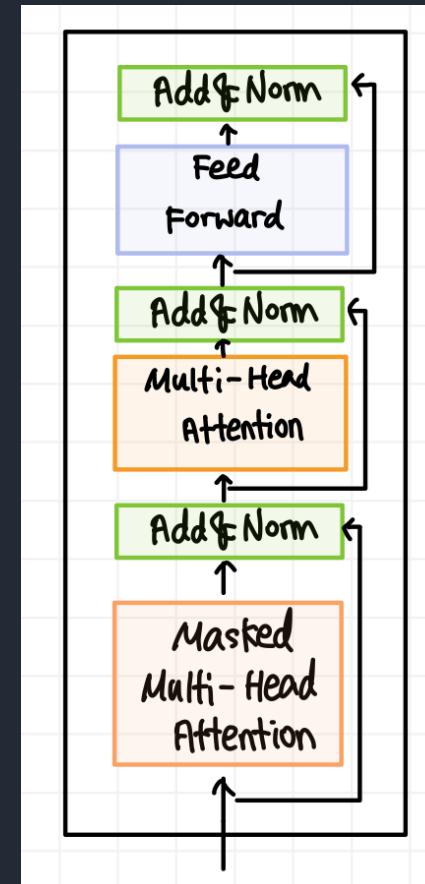
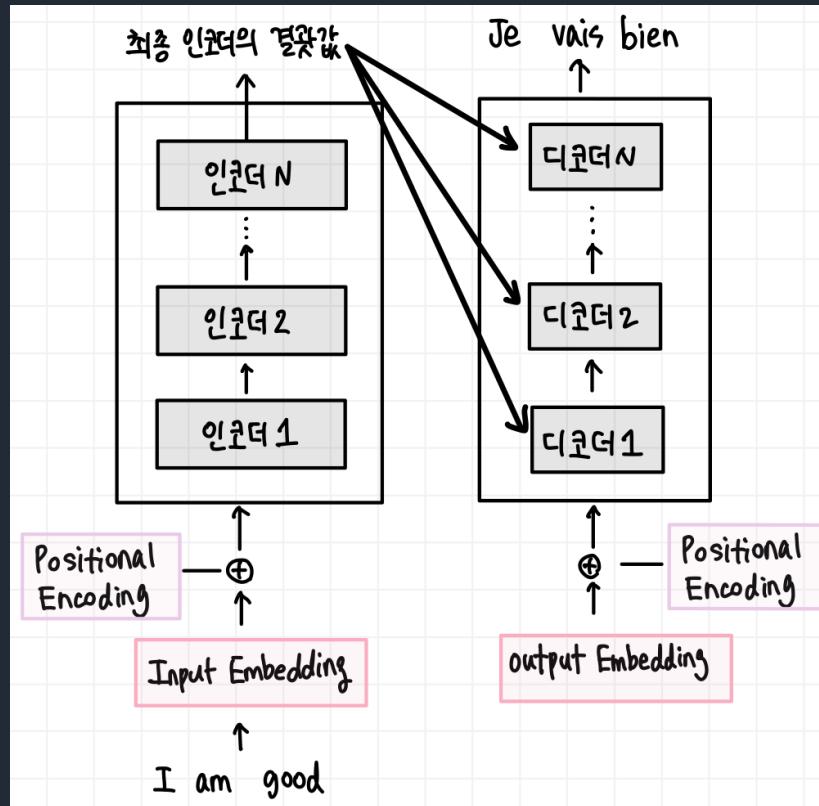
# 04. Transformer



이제 Decoder 부분을 봅시다 ..!

# 04. Transformer

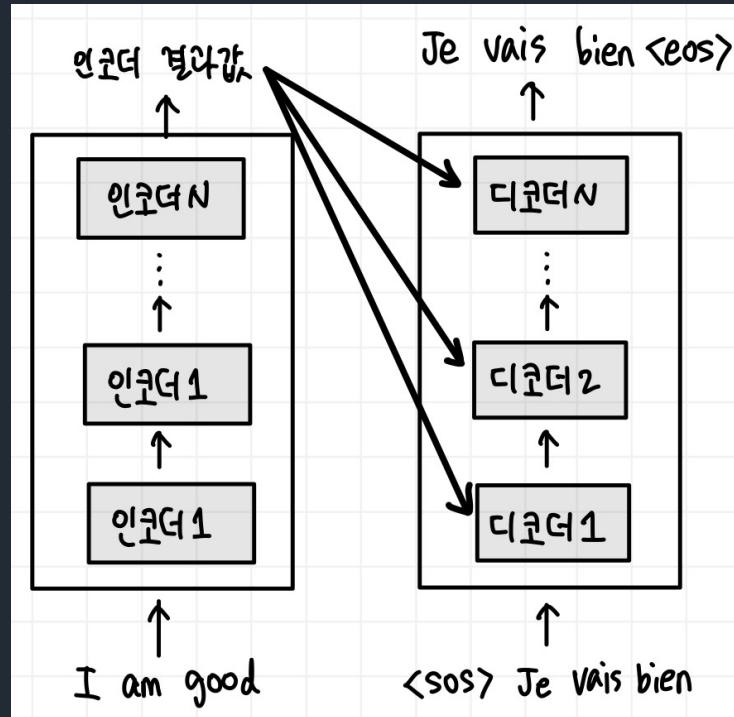
## Decoder



- 이전에 봤던 Encoder-Decoder 구조와 동일하다!
- 다만 encoder의 결과값이 **모든** decoder에 전송된다.  
-> Decoder의 입력값: 이전 Decoder의 출력값 & Encoder의 출력값
- Decoder는 **Masked Multi-Head Attention**, **Multi-Head Attention**, **Feed Forward**로 이루어져 있다.

# 04. Transformer

## Decoder의 입력과 출력



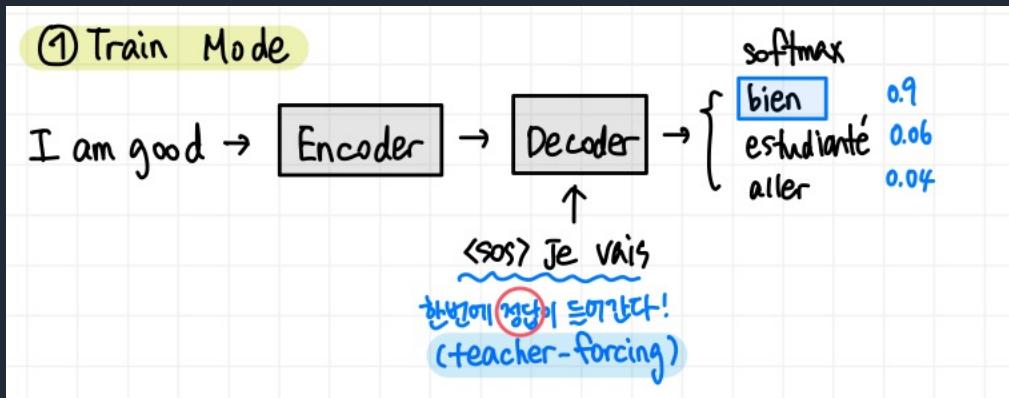
Decoder 학습시 타겟 문장 전체를 넣어준다. (앞에서 배웠던 teacher forcing)

Decoder 입력: <sos> Je vais bien

Decoder 출력: Je vais bien <eos>

# 04. Transformer

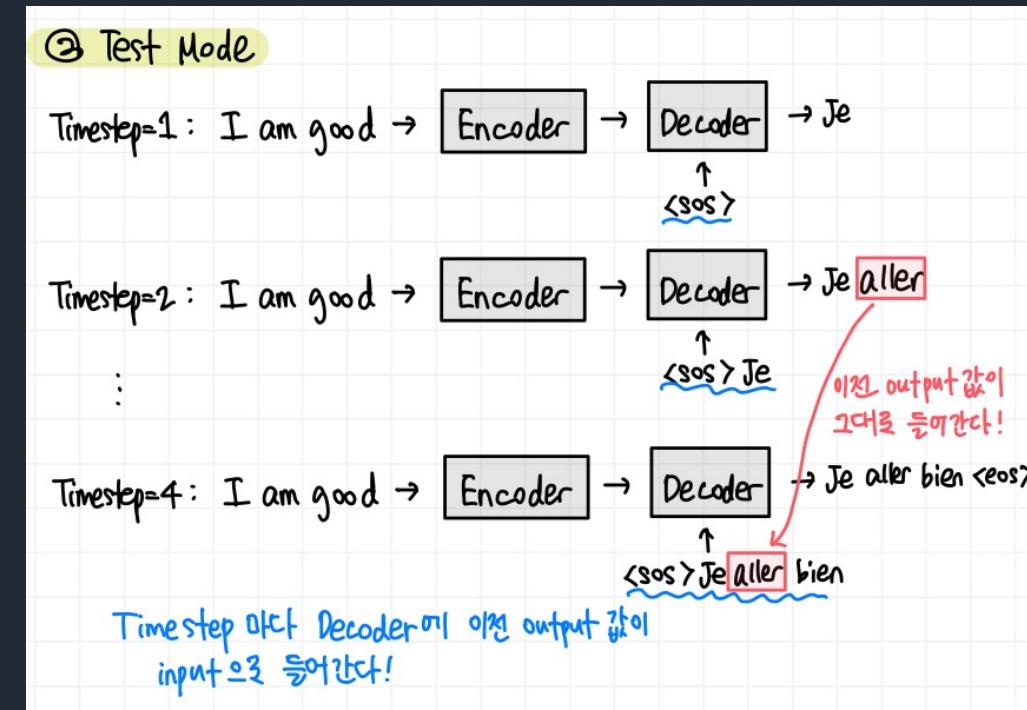
Decoder의 Test mode, Train mode는 다르다



<Train Mode>

한번에 정답값이 들어간다

(Teacher-Forcing)

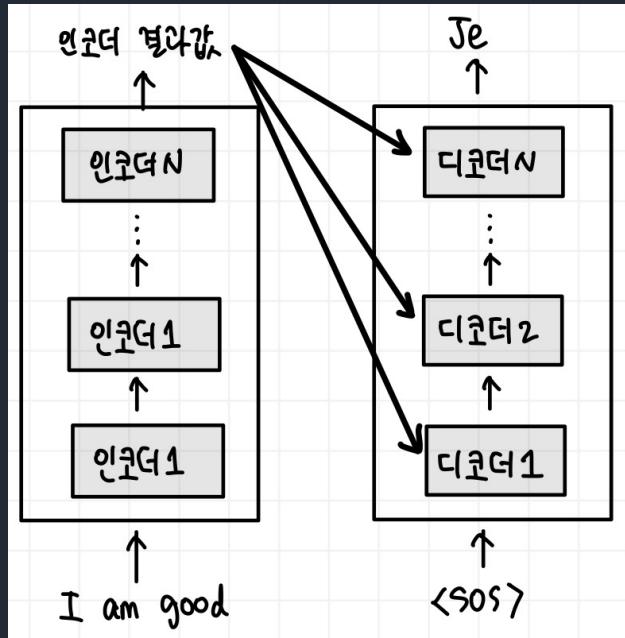


<Test Mode>

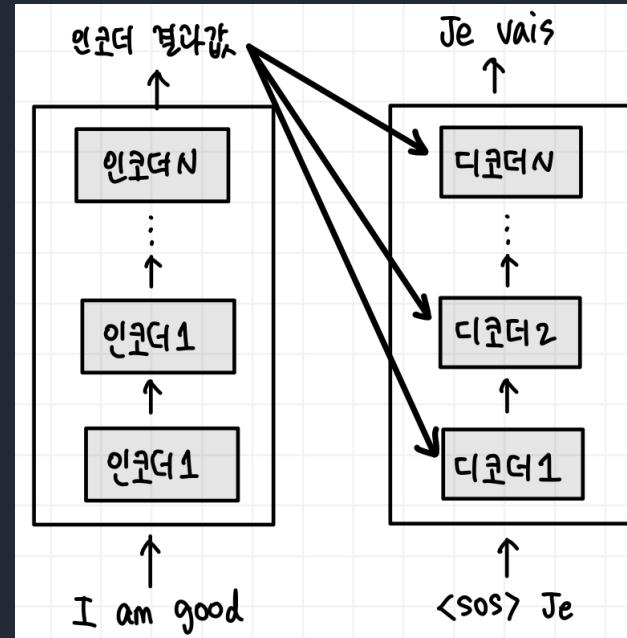
Timestep마다 Decoder에 이전 Timestep의 output 값이 그대로 input으로 들어간다

# 04. Transformer

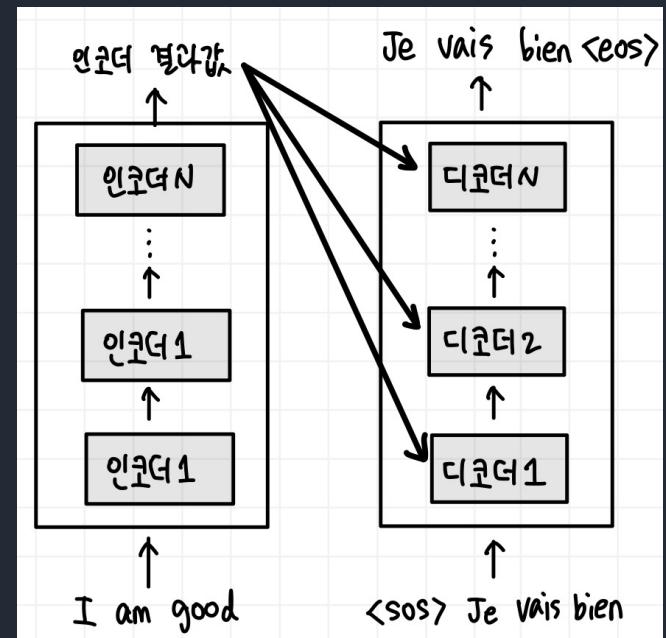
## Masked Multi-Head Attention



Timestep = 1



Timestep = 2



Timestep = 4

Decoder에서 실제로 진행(Test mode)할 때, 이전 단계에서 생성한 단어만 입력 문장으로 넣는다!

-> 이러한 특성을 살려서 모델을 학습(Train mode)시켜야 한다!

-> 미래의 정보는 참고하지 못하게 Masking 하자!

# 04. Transformer

## Masked Multi-Head Attention

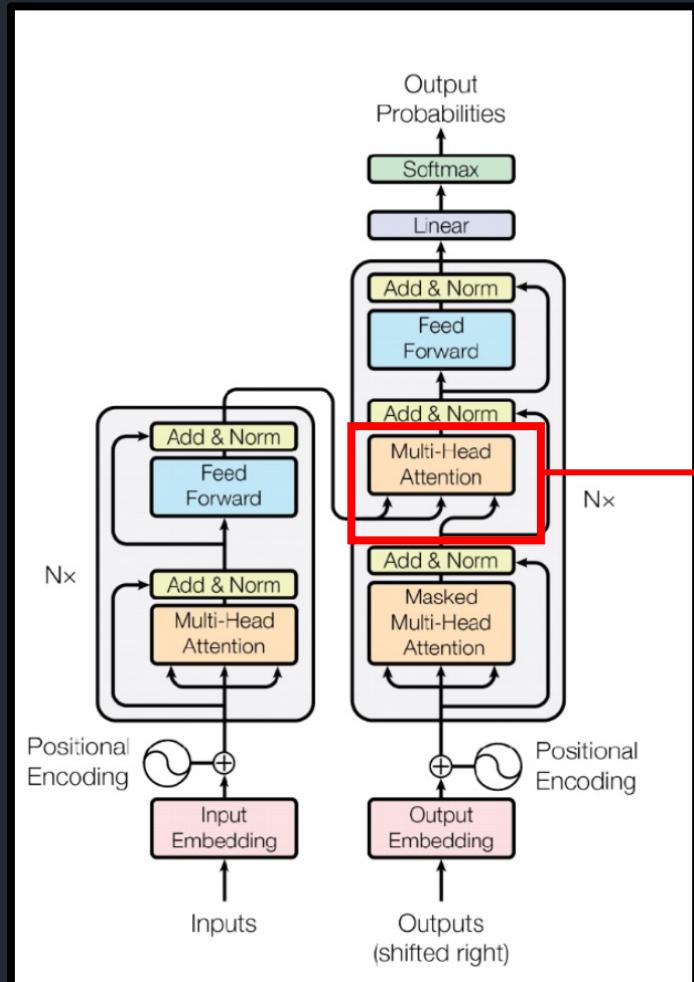
		< sos > Je suis bien			
< sos >		Mask	Mask	Mask	Mask
Je	1.2	Mask	Mask	Mask	Mask
vais	0.4	2.5	Mask	Mask	Mask
bien	0.8	1.7	3.5	Mask	Mask

이전 Self-Attention에서  $Q \cdot K^T$ , Query 행렬(Q)와 Key행렬( $K^T$ )간의 내적 연산으로 Q,K의 유사도를 산출하는 과정에서 뒤의 단어와의 관계는 고려하지 못하도록 Masking처리 한다!

		< sos > Je suis bien			
< sos >		-∞	-∞	-∞	-∞
Je	1.2	-∞	-∞	-∞	-∞
vais	0.4	2.5	-∞	-∞	-∞
bien	0.8	1.7	3.5	-∞	-∞

# 04. Transformer

## Decoder의 Multi-Head Attention 은 Self-Attention이 아니다!

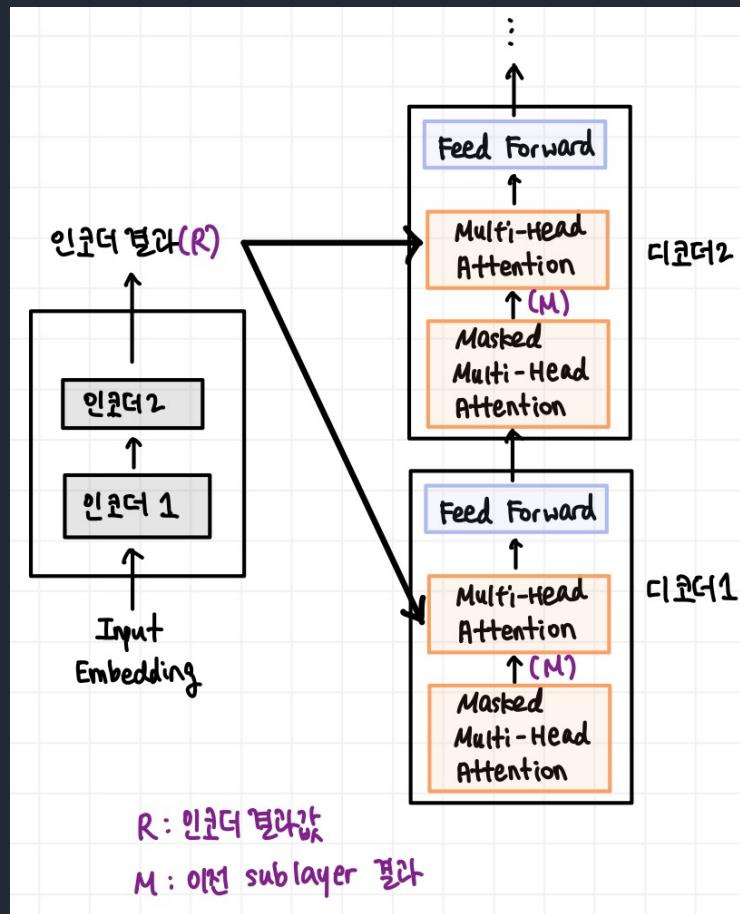


얘만 화살표 3개중 2개는 Encoder에서, 1개는 Decoder 내 sublayer의 출력값이다 !  
Key, Value는 Encoder에서, Query는 아래 Masked Multi-Head Attention에서 얻는다.  
-> 여기서 Encoder와 Decoder의 상호작용이 일어난다 !  
-> 출력 단어가 입력 단어 중 어떤 단어와 가장 연관성이 높은지 계산한다.

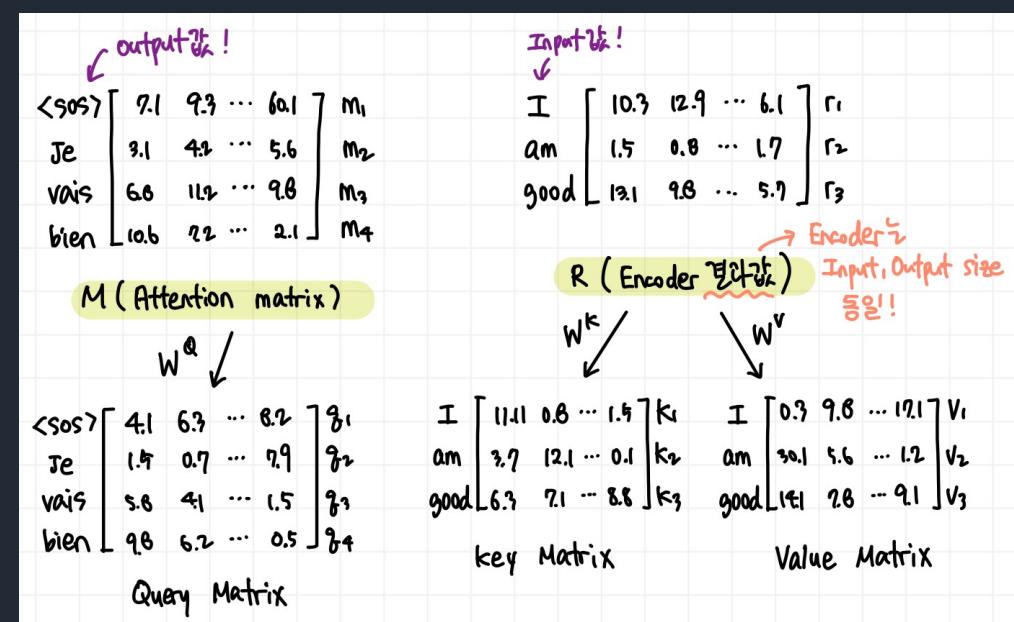
이것을 Self-Attention이라 하지 않고, 구분해서 Multi-Head Cross Attention 또는 Encoder-Decoder Attention이라고도 한다.

# 04. Transformer

## Decoder의 Multi-Head Attention 은 Encoder-Decoder Attention이다



- R(인코더 결과값)을 이용해 Key, Value 행렬 생성
  - M(이전 sublayer 결과값=Attention matrix)을 이용해 Query 행렬 생성
- > Query 는 타깃 문장의 표현을 포함하므로 M을 참고, Key, Value는 입력 문장의 표현을 가져서 R을 참고한다.



# 04. Transformer

## Decoder의 Multi-Head Attention 은 Encoder-Decoder Attention이다

아까 Self-Attention을 계산했던 것처럼 똑같이 해보자.

Step1.

Query 행렬( $Q$ )와 Key행렬( $K^T$ )간의 내적 연산으로  $Q, K$ 간의 유사도를 구한다

$$Q \cdot K^T = \begin{matrix} & \text{I} & \text{am} & \text{good} & \leftarrow \text{Input} \\ \langle \text{sos} \rangle & \begin{bmatrix} 91 & 60 & 77 \\ 96 & 63 & 12 \\ 41 & 11 & 48 \\ 36 & 45 & 65 \end{bmatrix} \\ \text{Je} \\ \text{vais} \\ \text{bien} \end{matrix}$$

*(The matrix shows the dot product of the query vector for '' and the transpose of the key matrix for the input words 'I', 'am', and 'good'. The input words are highlighted with a pink box and labeled with an arrow pointing to the right.)*

- Self-Attention은 같은 문장에서  $Q, K, V$  값을 뽑아서, 한 문장 내에 단어간의 유사도를 계산!
  - Encoder-Decoder Attention에서는  $Q$ 는 Decoder쪽에서,  $K$ 는 Encoder쪽을 참조해 Input 문장과 Output(Target) 문장 간의 유사도를 계산!
- (예시)  $\langle \text{sos} \rangle$ 가 각각 I, am, good 중 어떤 단어와 가장 유사한지 본다!

# 04. Transformer

## Decoder의 Multi-Head Attention 은 Encoder-Decoder Attention이다

### Step 2.

$Q \cdot K^T$  행렬을 key 차원 벡터의 제곱근 값으로 나누어준다.

### Step 3.

SoftMax 함수를 써워 0~1 사이의 분포로 정규화 시켜 score matrix 를 구한다.

### Step 4.

앞서 구한 score matrix에 Value 행렬을 곱하여 Attention 행렬 ( $Z$ )를 구한다.

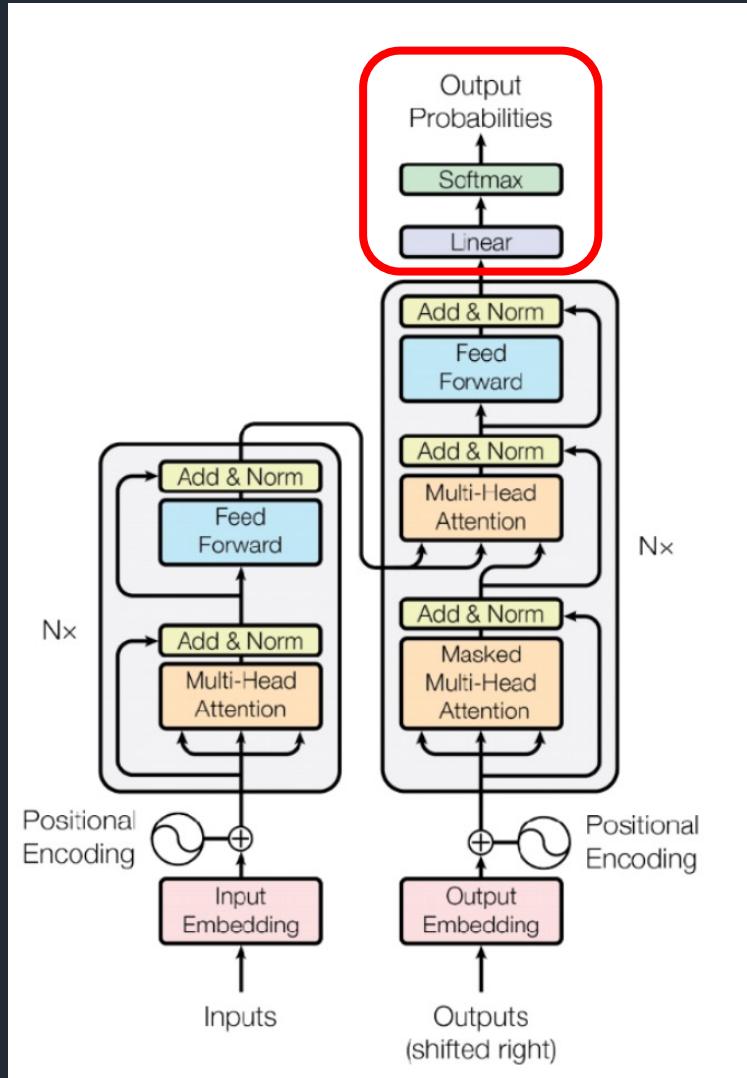
$Z$ 는 각 단어의 Value vector 값의 가중치 합으로 계산된다.

### Step 5.

$\text{Multi - Head Attention} = \text{concatenate}(Z_1, Z_2, \dots, Z_h) \cdot W_o$

$$\begin{aligned} Z &= \begin{bmatrix} \text{I} & \text{am} & \text{good} \\ \text{<sos>} & 0.8 & 0.01 \\ \text{Je} & 0.98 & 0.02 \\ \text{vais} & 0 & 1 \\ \text{bien} & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} \text{I} & 0.35 & 9.8 & \dots & 4.1 \\ \text{am} & 30.1 & 2.3 & \dots & 4.3 \\ \text{good} & 14.5 & 16.1 & \dots & 19.8 \end{bmatrix} V \\ &\quad \text{softmax}\left(\frac{QK^T}{\sqrt{dk}}\right) \cdot V \\ &= \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix} \begin{bmatrix} \text{<sos>} \\ \text{Je} \\ \text{vais} \\ \text{bien} \end{bmatrix} \rightarrow z_2 = 0.98[0.35, 9.8, \dots, 4.1] + 0.02[30.1, 2.3, \dots, 4.3] + 0[14.5, 16.1, \dots, 19.8] \\ &\quad \text{Je-I} \text{ 유사도 } V_1(\text{I}) \quad \text{Je-am} \text{ 유사도 } V_2(\text{am}) \quad \text{Je-good} \text{ 유사도 } V_3(\text{good}) \\ &\quad \rightarrow \text{score} \text{에 대한 가중치를 반영한 벡터값의 합} \end{aligned}$$

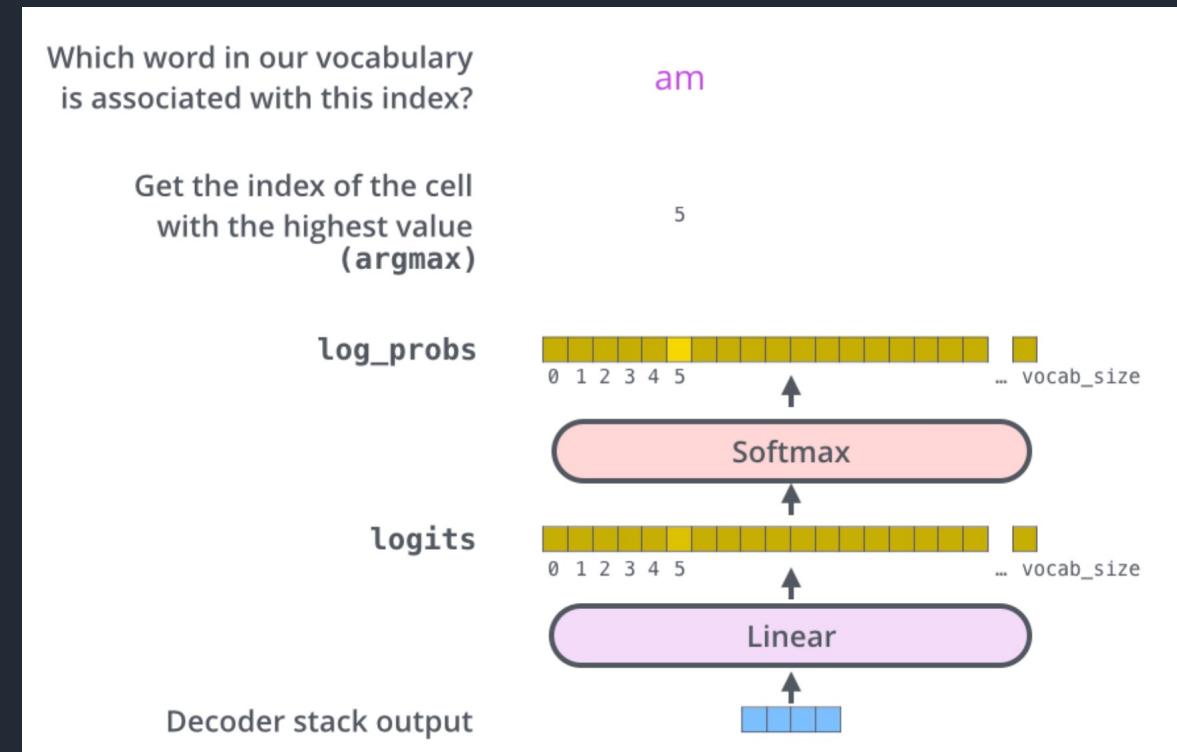
# 04. Transformer



마지막 부분만 마저 봅시다!

## Linear Layer & SoftMax Layer

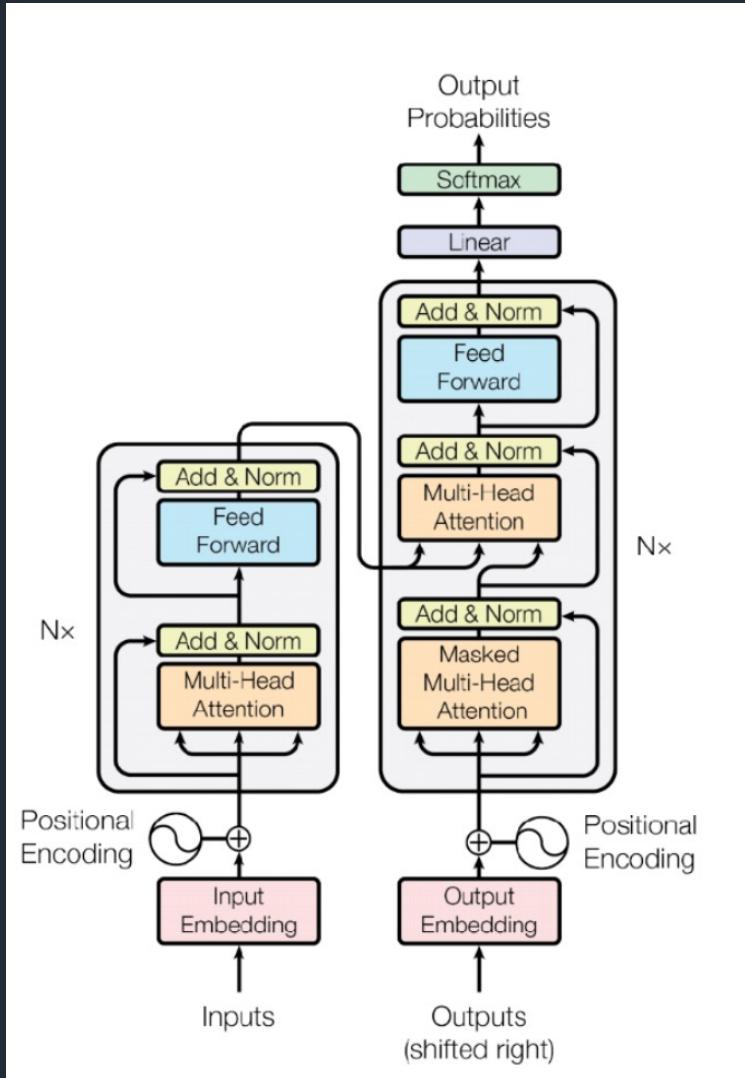
1. 디코더에서 나온 결과값은 Linear Layer로 들어간다.
2. Linear Layer는 크기가 vocab size 인 logit값을 반환한다.
3. Softmax 함수를 사용해 Logit값을 확률 값으로 변환 후, 가장 확률이 높은 index의 단어를 출력한다.



## Transformer 학습

- 올바른 문장을 생성하려면, 예측 분포와 실제 확률 분포 사이의 차이를 최소화해야 한다.
- 두 분포의 차이는 cross-entropy를 사용하여 알 수 있다.
- Loss function 을 cross-entropy loss 로 정의하고, 차이를 최소화하도록 모델을 학습한다.

# 05. Summary



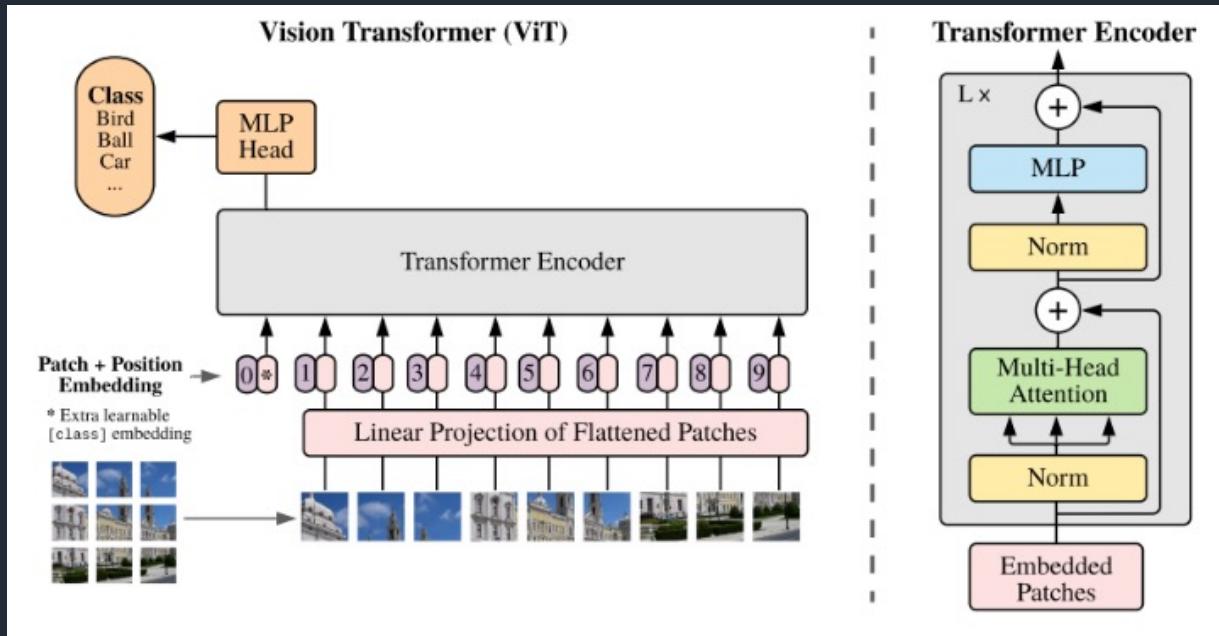
## ★ <기억해야 할 것>★

- Q,K,V의 개념
- Attention matrix 구하는 방법
- Encoder의 Multi-Head Attention, Decoder의 Masked Multi-Head Attention 은 Self-Attention 이고, Decoder의 Multi-Head Attention은 Encoder-Decoder Attention 이다.
- 각 Encoder, Decoder의 차원 계산
- Decoder의 Train mode, Test mode 차이

# 06. Appendix

Transformer 구조는 NLP에서만 쓰이는 것은 아니다 !

-> Vision 쪽에서도 Transformer 의 Encoder구조를 사용한 Vision Transformer 가 있다



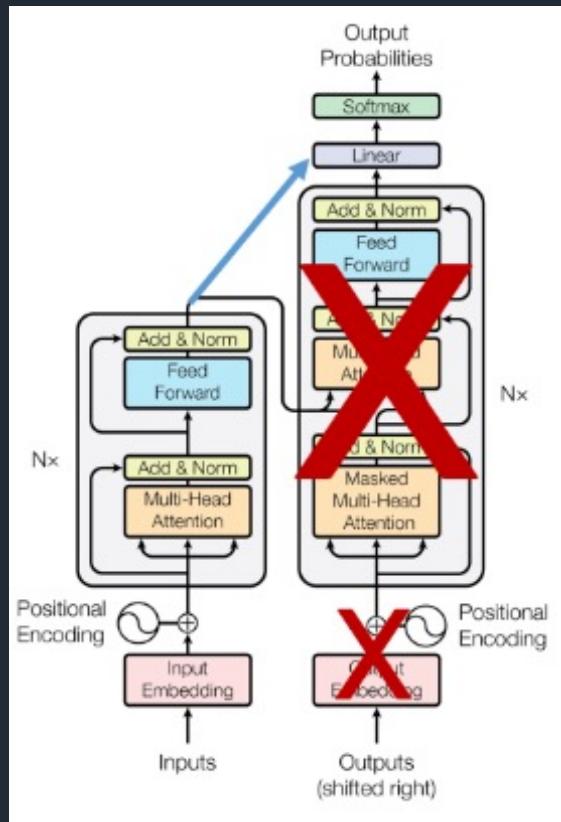
"An Image is worth 16\*16 words"

-> Image patch 의 sequence 를 입력 값으로 사용

# 06. Appendix

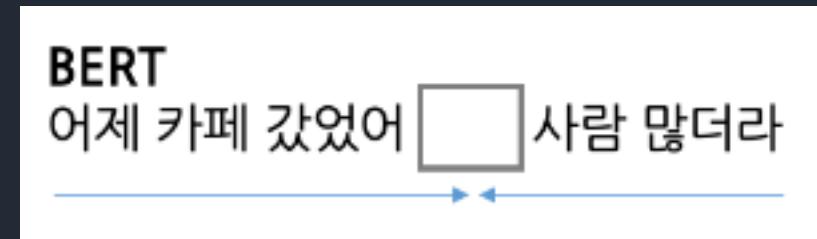
Transformer 의 Encoder 구조만 쓴 BERT (Bidirectional Encoder Representations from Transformers)

-> 우리는 현재 BERT의 시대에 살고 있다



Encoder의 Multi-Head Attention, 즉 Self-Attention 만 수행

-> 한 문장 내에 빈칸 뚫어 놓고, 양방향으로 문맥을 보며 빈칸 맞추기 !



- 한 문장 내에서 알아서 학습하므로, 레이블이 없는 아무 문장을 가지고 학습시킬 수 있다!
- 33억 단어에 대해 Pre-train 시켰다.
- 우리는 사전 학습된 BERT 모델을 이용하여, 우리가 하고자 하는 task에 맞춰서 위에 다른 모델만 쌓아주면 된다.

# 06. Reference

- Attention is All you need, Vaswani et al, NIPS, 2017, <https://arxiv.org/abs/1706.03762>
- 구글 BERT의 정석, 수다르산 라비찬디란 저
- Sequence to Sequence Learning with Neural Networks, Sutskever et al, 2014, <https://arxiv.org/abs/1409.3215>
- <https://jalammar.github.io/illustrated-transformer/>
- <https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/>
- Weekly NLP, <https://jiho-ml.com/weekly-nlp-25/>
- <https://github.com/christianversloot/machine-learning-articles/blob/main/from-vanilla-rnns-to-transformers-a-history-of-seq2seq-learning.md>
- <https://blog.promedius.ai/transformer/>
- <https://towardsdatascience.com/beautifully-illustrated-nlp-models-from-rnn-to-transformer-80d69faf2109>

# DATA SCIENCE LAB

---

발표자 엄소은 010-9887-7683  
E-mail: uhmturks@Yonsei.ac.kr