
Transformer

Attention Is All You Need

20192241 이민옥



1. Transformer Overview

2. Encoder

1. Input imbedding
2. Positional Encoding
3. Self-Attention
4. Multi-Head Attention
5. Residual

3. Decoder

1. Masked Multi-Head Attention
2. Encoder-Decoder Attention
3. The Final Linear and Softmax Layer



1. Transformer Overview

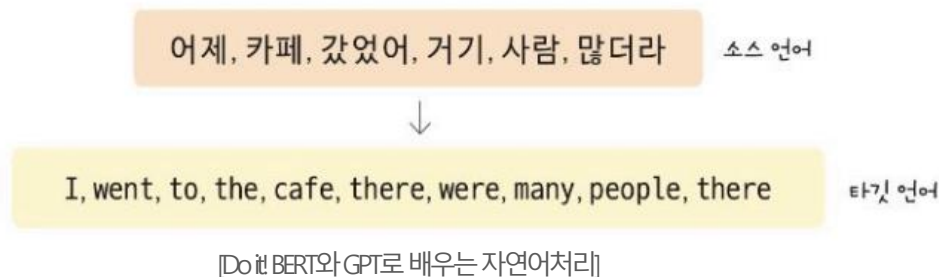
Transformer

- **트랜스포머**

- 2017년 구글이 제안한 **Sequence-to-Sequence** 모델
- Attention의 병렬적 사용을 통해 효율적인 학습이 가능한 구조의 언어 모델

- **Sequence-to-Sequence**

- 단어 같은 무언가의 나열 의미
- '특정 속성을 지닌 시퀀스를 다른 속성의 시퀀스로 변환하는 작업'



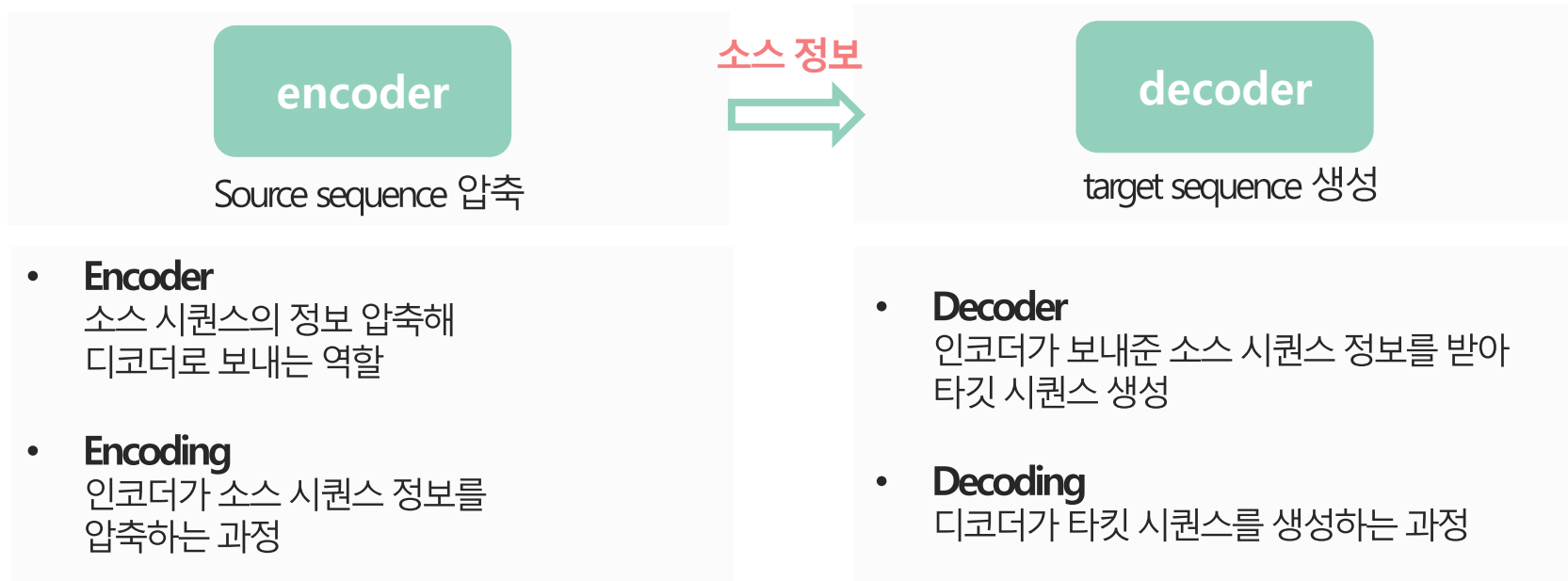
- **기계번역**

Source language의 토큰 시퀀스를 target language의 토큰 시퀀스로 변환하는 과제



Encoder and Decoder

- Transformer 내부에 Encoder와 Decoder 2개 파트로 구성



- 예를들어 기계 번역에서는 인코더가 한국어 문장을 압축해 디코더에 보내고, 디코더는 이를 받아 영어로 번역



Transformer based on encoder and decoder

Transformer 구조

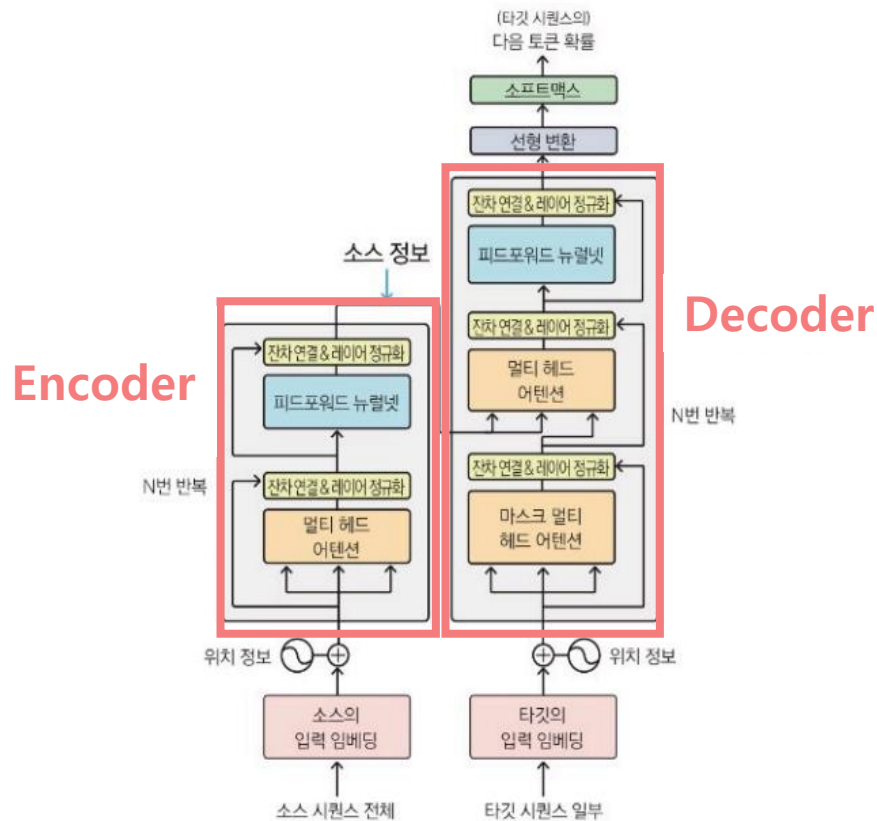
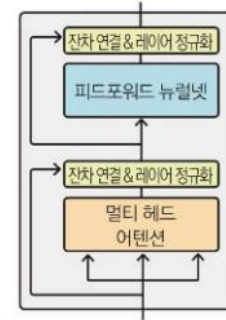


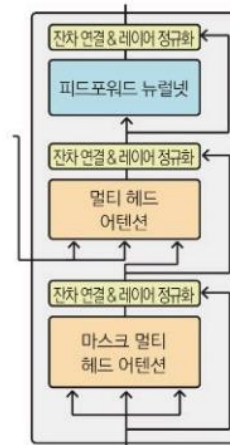
그림 3-8 트랜스포머의 구조

Transformer block 1, **Encoder Unmasked**



- 멀티 헤드 어텐션
- 피드포워드 뉴럴 네트워크
- 잔차연결 & 레이어 정규화

Transformer block 2, **Decoder Masked**



- **마스크 멀티 헤드 어텐션**
- 멀티 헤드 어텐션(인코더-디코더)
- 피드포워드 뉴럴 네트워크
- 잔차연결 & 레이어 정규화

2. Encoder

1. Input 임베딩

- 입력 값 임베딩
 - 네트워크에 넣기 위해 보통 임베딩 과정을 거침

입력 문장: I ate sandwich

$$\begin{array}{lcl}
 \text{I} & : x_1 = [12.13, 1.1, \dots, 5.38] & \\
 \text{ate} & : x_2 = [73.21, 9.12, \dots, 0.13] & \\
 \text{sandwich} & : x_3 = [23.48, 8.19, \dots, 48.23] & \\
 \end{array}
 \Rightarrow
 \begin{array}{lcl}
 \text{I} & \begin{pmatrix} 12.13 & 1.1 & \dots & 5.38 \end{pmatrix} & x_1 \\
 \text{ate} & \begin{pmatrix} 73.21 & 9.12 & \dots & 0.13 \end{pmatrix} & x_2 \\
 \text{sandwich} & \begin{pmatrix} 23.48 & 8.19 & \dots & 48.23 \end{pmatrix} & x_3
 \end{array}$$

입력 행렬(임베딩 행렬) X
3 x 512

x_1

Je

x_2

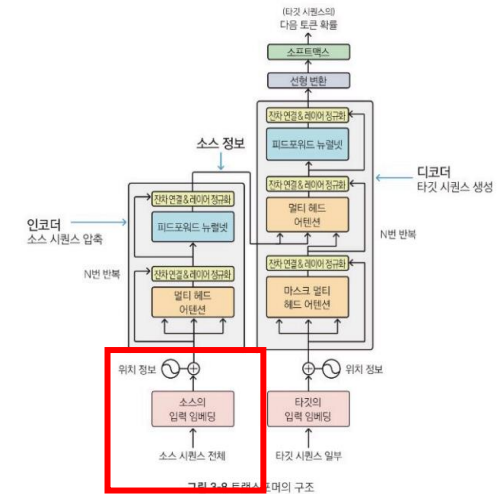
suis

x_3

étudiant

Each word is embedded into a vector of size 512. We'll represent those vectors with these simple boxes.

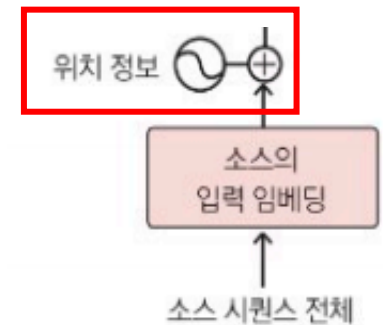
- 단어 임베딩은 가장 아래 위치한 인코더에서만 입력으로 1회 사용
- 나머지 인코더들을 하위 인코더에서 출력된 결과물을 입력으로 사용



논문에서의 임베딩 차원은 512



2. Positional Encoding



- Positional Encoding
 - Transformer는 단어를 순차적으로 받지 않고 한번에 받는 구조이므로 입력 시퀀스에서 단어들 간의 위치 관계를 표현해줄 필요가 있음
 - 이러한 역할을 수행하도록 설계된 벡터를 포지션 인코딩이라고 하며, 모든 단어 임베딩에 포지션 인코딩을 더해서 입력 벡터를 구성

- 좋은 Positional Encoding이 갖는 속성

1) 모든 위치값은 시퀀스의 길이나 Input 관계없이 동일 식별자를 가져야 한다.

→ 따라서 시퀀스가 변경되더라도 위치 임베딩은 동일하게 유지될 수 있다.

2) 모든 위치값은 너무 크면 안된다

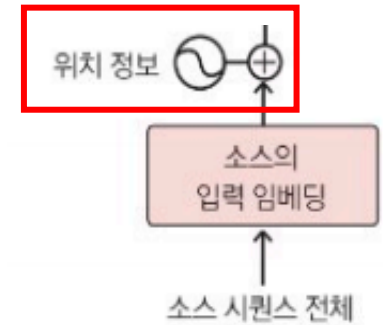
→ 위치값이 너무 커져버리면, 단어간의 상관관계 및 의미를 유추할 수 있는 의미정보 값이 상대적으로 작아지게 되고,

Attention layer에서 제대로 학습 및 훈련이 되지 않을 수 있다.

this	is	my	car
0.19	0.70	0.34	0.69
-0.47	-0.65	0.87	0.79
-0.77	0.11	-0.39	-0.25
0.59	0.04	-0.91	0.44
0.01	0.13	0.14	0.05
0.01	0.31	0.16	0.04
0.02	0.23	0.11	0.11
0.02	0.24	0.07	0.15

that	is	not	[PAD]
0.19	0.70	0.74	0.00
-0.41	-0.65	0.76	0.00
0.12	0.11	0.13	0.00
0.59	0.04	0.18	0.00
0.01	0.13	0.14	0.05
0.01	0.31	0.16	0.04
0.02	0.23	0.11	0.11
0.02	0.24	0.07	0.15

2. Positional Encoding



- Positional encoding 값을 구하는 수식
 - 각 원소간의 위치에 대한 정보를 알려주는 함수

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

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

- pos** 데이터의 위치
- i** position 표현 위한 차원의 인덱스
- d_model** 차원의 수

- A Simple Example (n = 10, dim = 10) 두 포지션 인코딩 벡터간의 거리

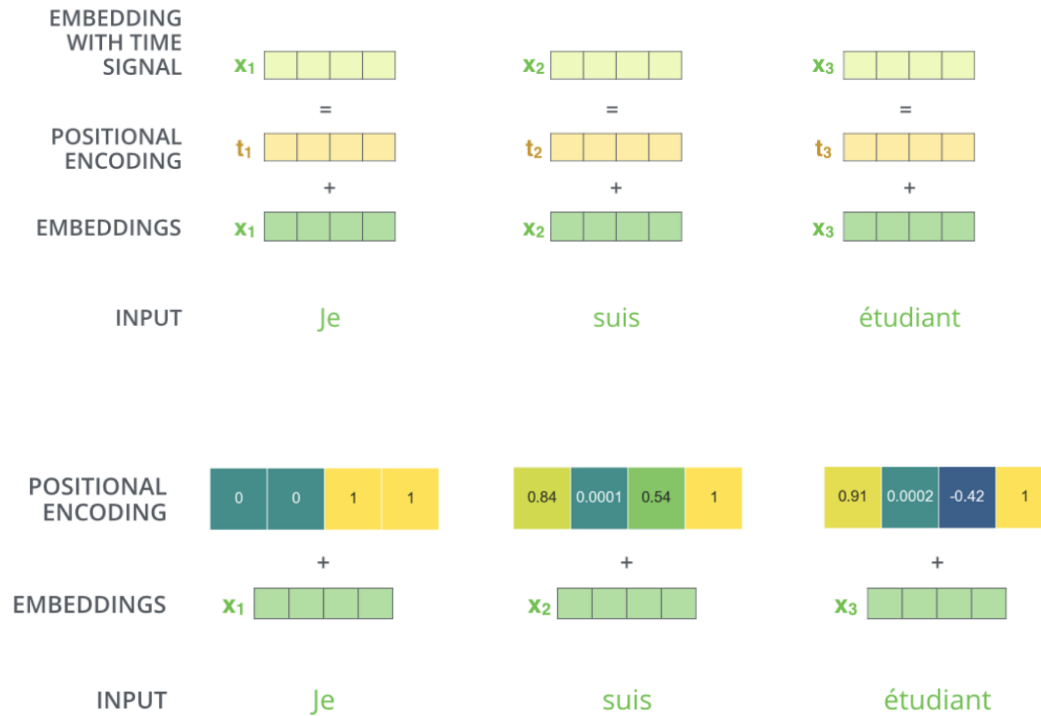
	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10
X1	0.000	1.275	2.167	2.823	3.361	3.508	3.392	3.440	3.417	3.266
X2	1.275	0.000	1.104	2.195	3.135	3.511	3.452	3.442	3.387	3.308
X3	2.167	1.104	0.000	1.296	2.468	3.067	3.256	3.464	3.498	3.371
X4	2.823	2.195	1.296	0.000	1.275	2.110	2.746	3.399	3.624	3.399
X5	3.361	3.135	2.468	1.275	0.000	1.057	2.176	3.242	3.659	3.434
X6	3.508	3.511	3.067	2.110	1.057	0.000	1.333	2.601	3.169	3.118
X7	3.392	3.452	3.256	2.746	2.176	1.333	0.000	1.338	2.063	2.429
X8	3.440	3.442	3.464	3.399	3.242	2.601	1.338	0.000	0.912	1.891
X9	3.417	3.387	3.498	3.624	3.659	3.169	2.063	0.912	0.000	1.277
X10	3.266	3.308	3.371	3.399	3.434	3.118	2.429	1.891	1.277	0.000

02

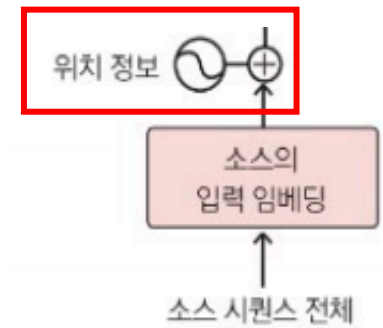
Encoding

2. Positional Encoding

- Positional Encoding Process



A real example of positional encoding with a toy embedding size of 4



3. Self-Attention

Encoding 과정

- Position encoding이 더해진 단어 임베딩은 첫 번째 인코더 블록에서 Self attention과 FFNN을 거치게 됨
- 입력된 벡터들은 self-attention과 FFNN을 거쳐 상위 인코더의 입력으로 투입

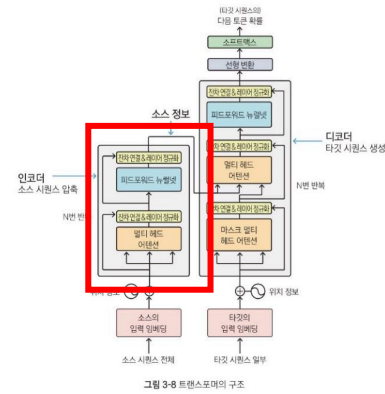
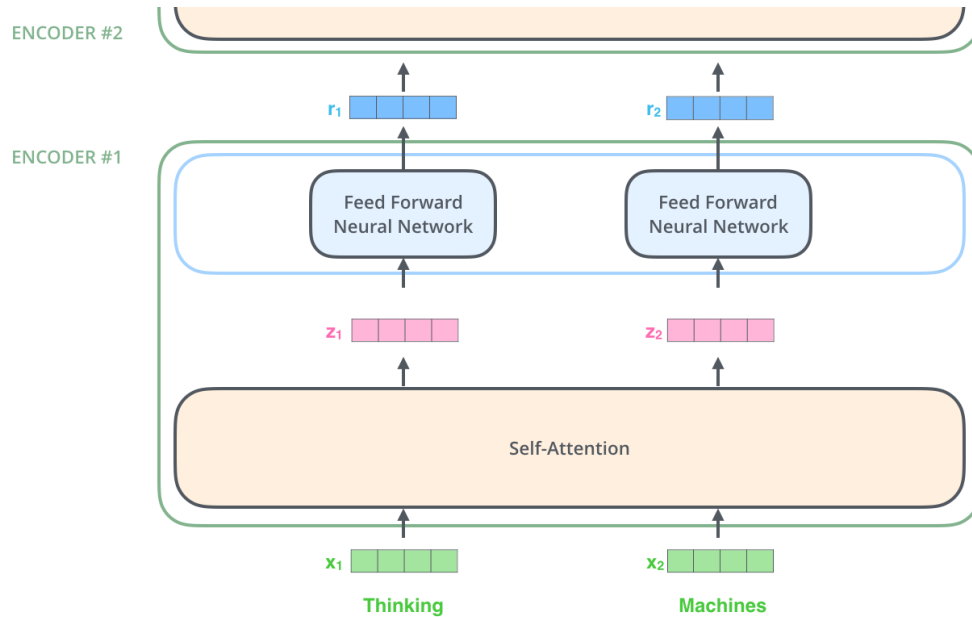
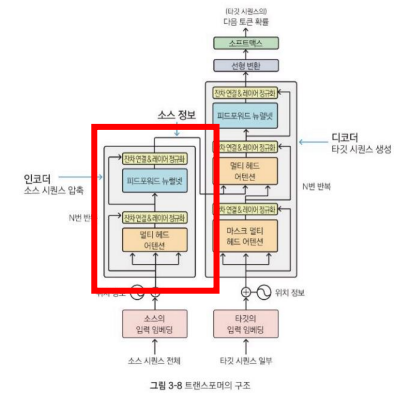
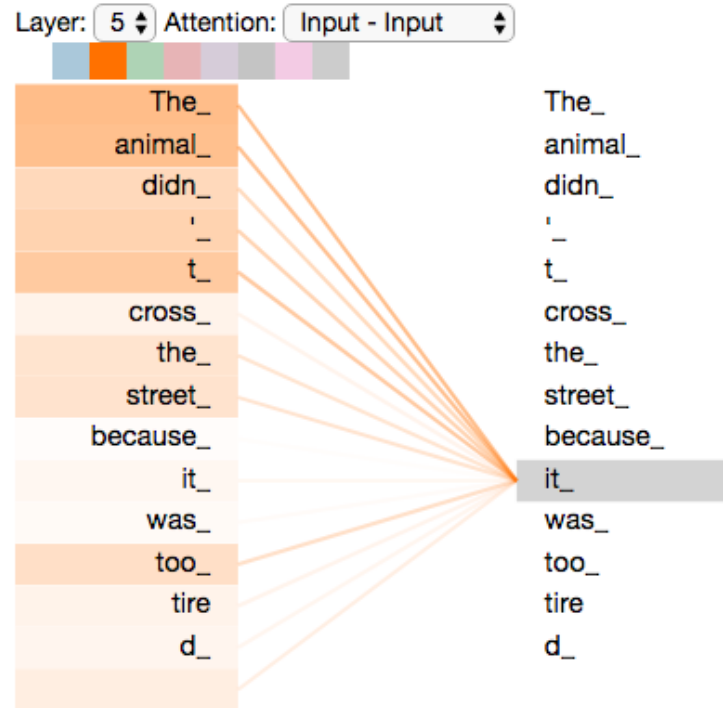


그림 3-8 트랜스포머의 구조

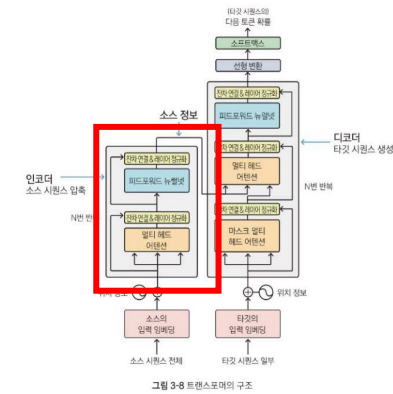
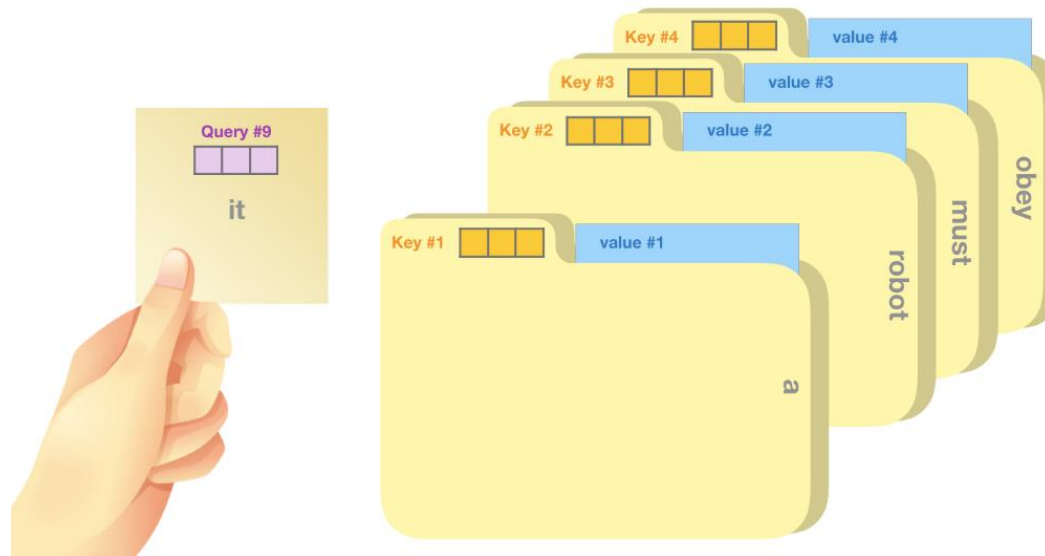
3. Self-Attention

- Self-Attention의 이해
 - 입력 시퀀스의 다른 위치에 있는 단어들을 둘러보며 특정 위치의 단어를 잘 설명할 수 있게 함



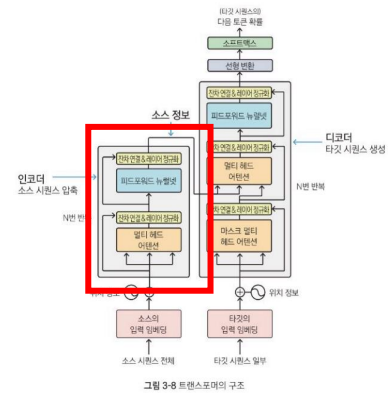
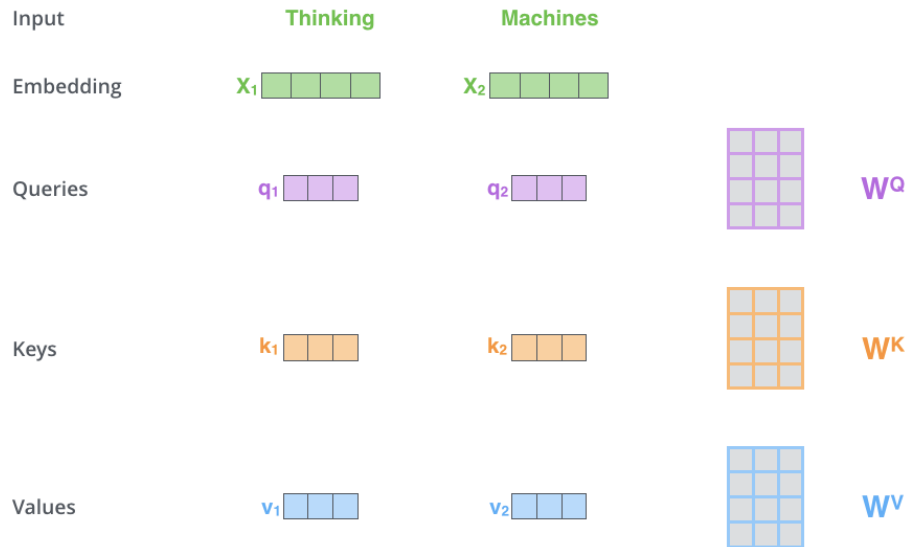
3. Self-Attention

- Self-Attention의 절차
- Step 1** Self-Attention을 위해 입력 벡터에 대한 세 가지의 벡터 생성
 - Query**: 다른 단어들을 고려하여 표현하고자 하는 대상이 되는 **현재 단어**에 대한 임베딩 벡터
 - Key**: Query가 들어왔을 때 다른 단어들과 매칭을 하기 위해 사용되는 **레이블**로 사용되는 임베딩 벡터
 - Value**: Key와 연결된 **실제 단어**를 나타내는 임베딩 벡터



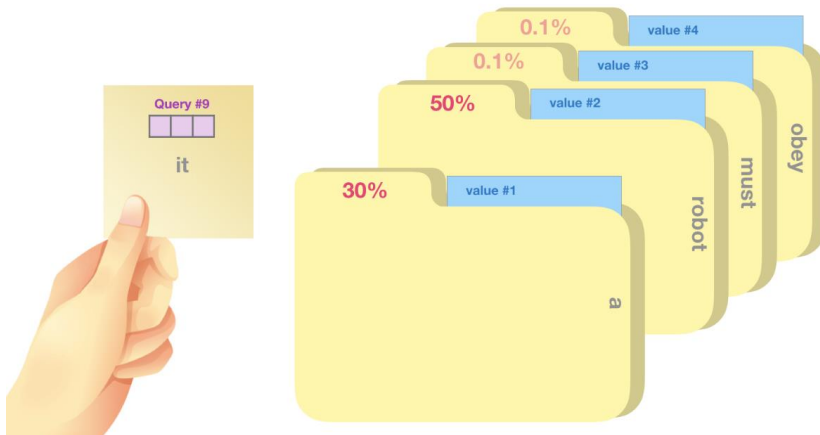
3. Self-Attention

- Self-Attention의 절차
- Step 1** Self-Attention을 위해 입력 벡터에 대한 세 가지의 벡터 생성
 - 실제로는 Query, Key, Value에 대응하는 행렬을 곱해서 생성



3. Self-Attention

- Self-Attention의 절차
- Step 2** $Q * K$ 에 소프트맥스 함수를 취해 score 계산
 - 지금 표현하고자 하는 단어(Q)에 대해 어떤 단어들을 고려해야 하는지(K)를 알려주는 스코어 산출



Input

Embedding

Queries

Keys

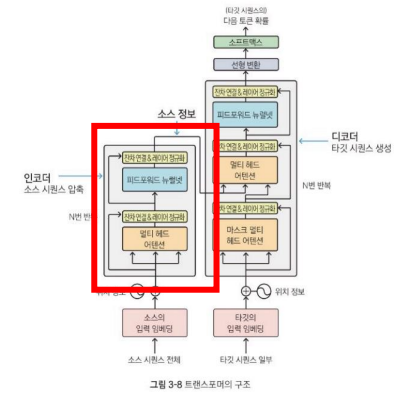
Values

Score

Thinking

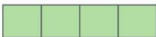
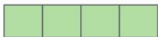
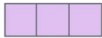
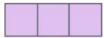
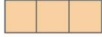
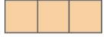
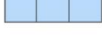
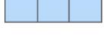
 x_1 [] [] [] [] q_1 [] [] [] k_1 [] [] [] v_1 [] [] [] $q_1 \cdot k_1 = 112$

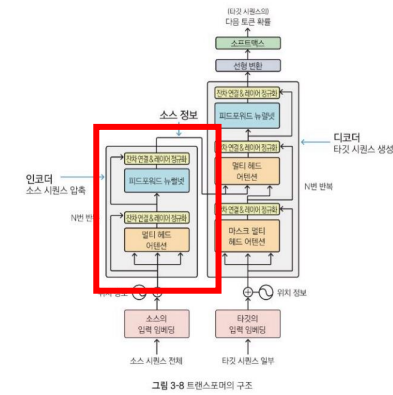
Machines

 x_2 [] [] [] [] q_2 [] [] [] k_2 [] [] [] v_2 [] [] [] $q_1 \cdot k_2 = 96$ 

3. Self-Attention

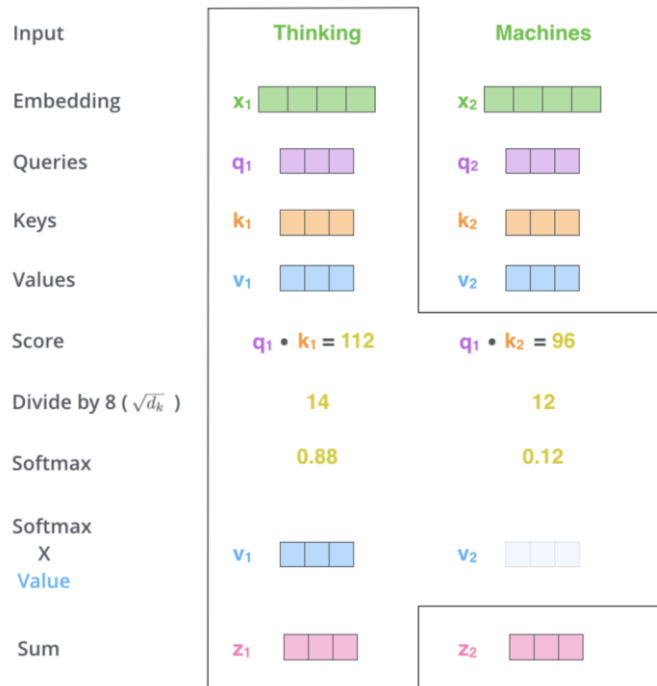
- Self-Attention의 절차
- Step 3** Step 2에서 계산한 score를 $\sqrt{d_k}$ (=8, 기존 논문 $d_k = 64$)로 나눠줌
- Step 4** Step 3의 결과물을 이용하여 소프트맥스 함수를 적용하여 해당 단어에 대한 집중도를 산출

Input	Thinking	Machines
Embedding	x_1 	x_2 
Queries	q_1 	q_2 
Keys	k_1 	k_2 
Values	v_1 	v_2 
Score	$q_1 \cdot k_1 = 112$	$q_1 \cdot k_2 = 96$
Divide by 8 ($\sqrt{d_k}$)	14	12
Softmax	0.88	0.12



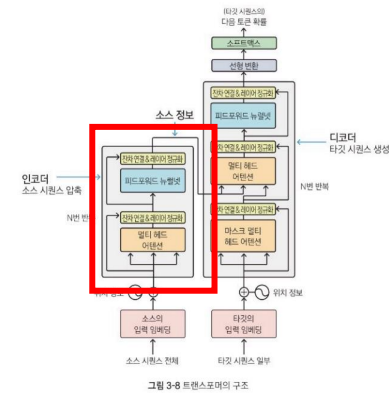
3. Self-Attention

- Self-Attention의 절차
- Step 5** Step 4에서 산출된 확률값과 해당 단어의 Value 값을 곱함
- Step 6** Step 5에서 산출된 모든 값들을 더해서 출력으로 반환



Word	Value vector	Score	Value X Score
<S>	[] [] []	0.001	[] [] []
a	[] [] []	0.3	[] [] []
robot	[] [] []	0.5	[] [] []
must	[] [] []	0.002	[] [] []
obey	[] [] []	0.001	[] [] []
the	[] [] []	0.0003	[] [] []
orders	[] [] []	0.005	[] [] []
given	[] [] []	0.002	[] [] []
it	[] [] []	0.19	[] [] []
		Sum:	[] [] []

[Self-attention 결과 예시]



3. Self-Attention

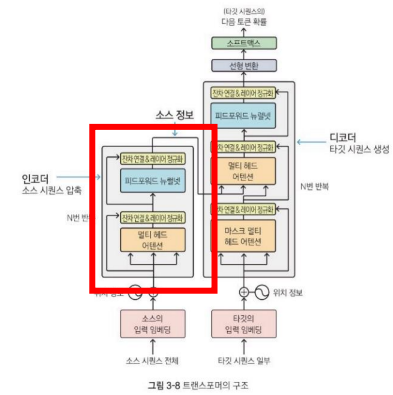
- Matrix calculation of Self-Attention

$$X \times W^Q = Q$$

$$X \times W^K = K$$

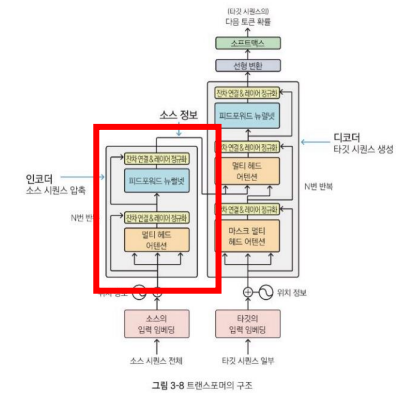
$$X \times W^V = V$$

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

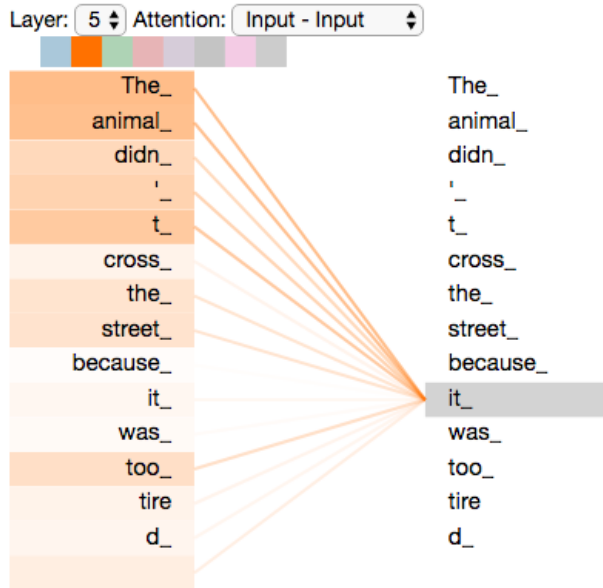


4. Multi-Head Attention

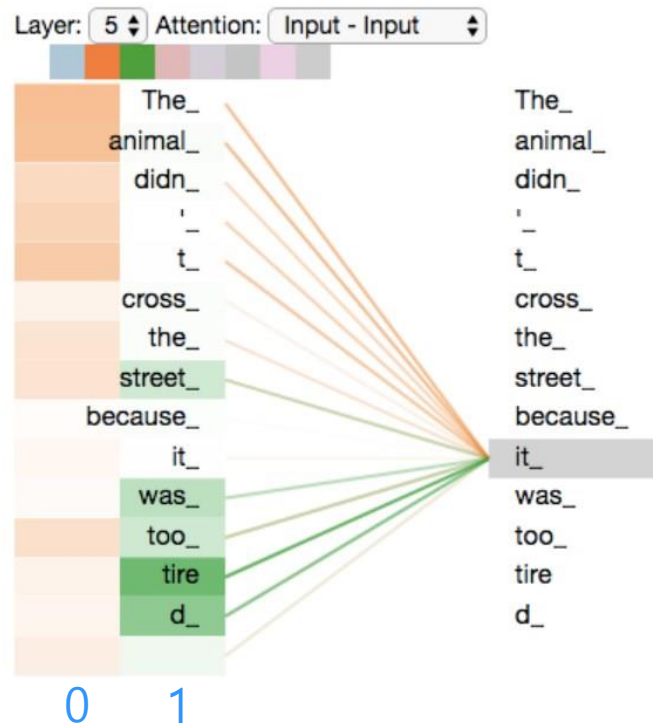
- Multi-Headed Attention의 과정



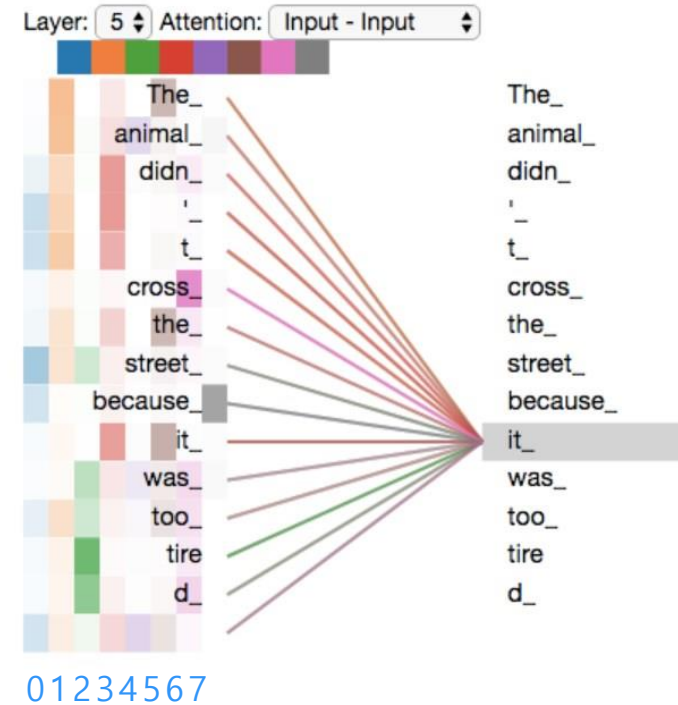
Attention with single heads



Attention with 2 heads



Attention with 8 heads



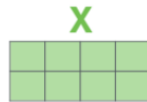
4. Multi-Head Attention

Multi-Headed Attention의 과정

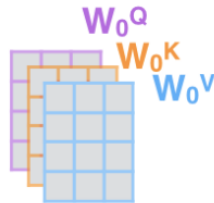
1) This is our input sentence*

Thinking
Machines

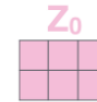
2) We embed each word*



3) Split into 8 heads.
We multiply X or R with weight matrices



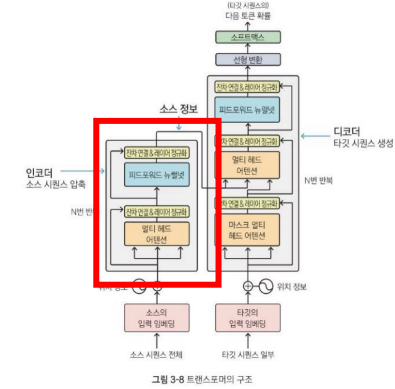
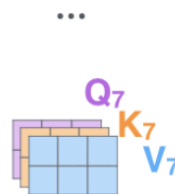
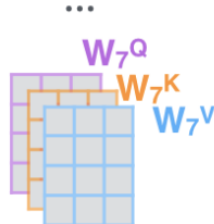
4) Calculate attention using the resulting $Q/K/V$ matrices



5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer



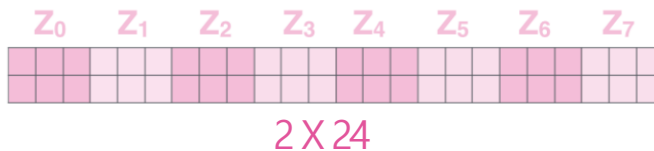
- 첫번째 encoder가 아닐경우, 이전 encoder output 그대로 사용한다.
- Original input embedding의 차원과 같기에 차원 보존하며 멀티 헤드 어텐션 수행 가능



4. Multi-Head Attention

- Multi-Headed Attention
 - 어텐션 결과물들은 병합(concatenation) 후 가중치 행렬과의 연산을 통해 원래 입력 차원과 동일한 차원의 출력 벡터를 생성하게 됨

1) Concatenate all the attention heads



2) Multiply with a weight matrix W^O that was trained jointly with the model

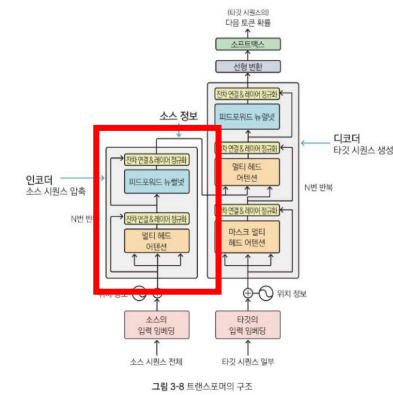
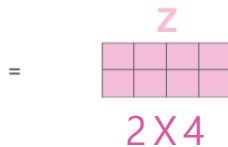
\times

24×4



W^O

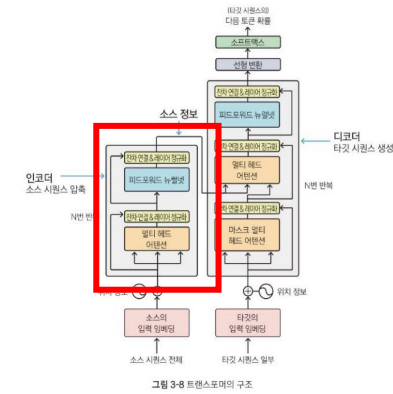
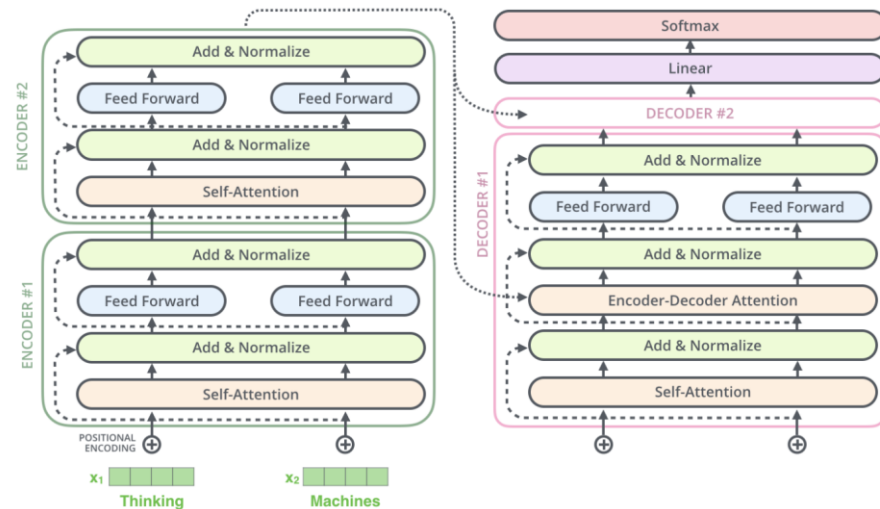
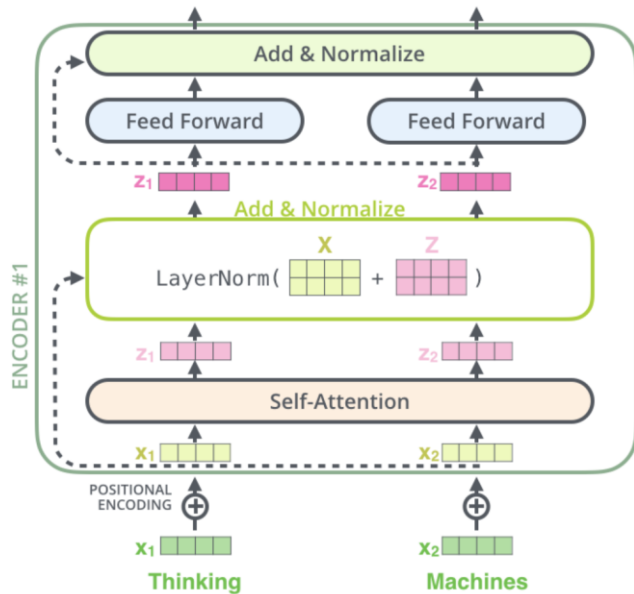
3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN



Encoding

5. Residual

- Residual Learning & Layer Normalize 과정
 - Residual learning : 각 레이어에서 입력값에 대한 잔차(residual)를 학습하는 방식
 - Layer Normalize 수행



Summarize Encoder

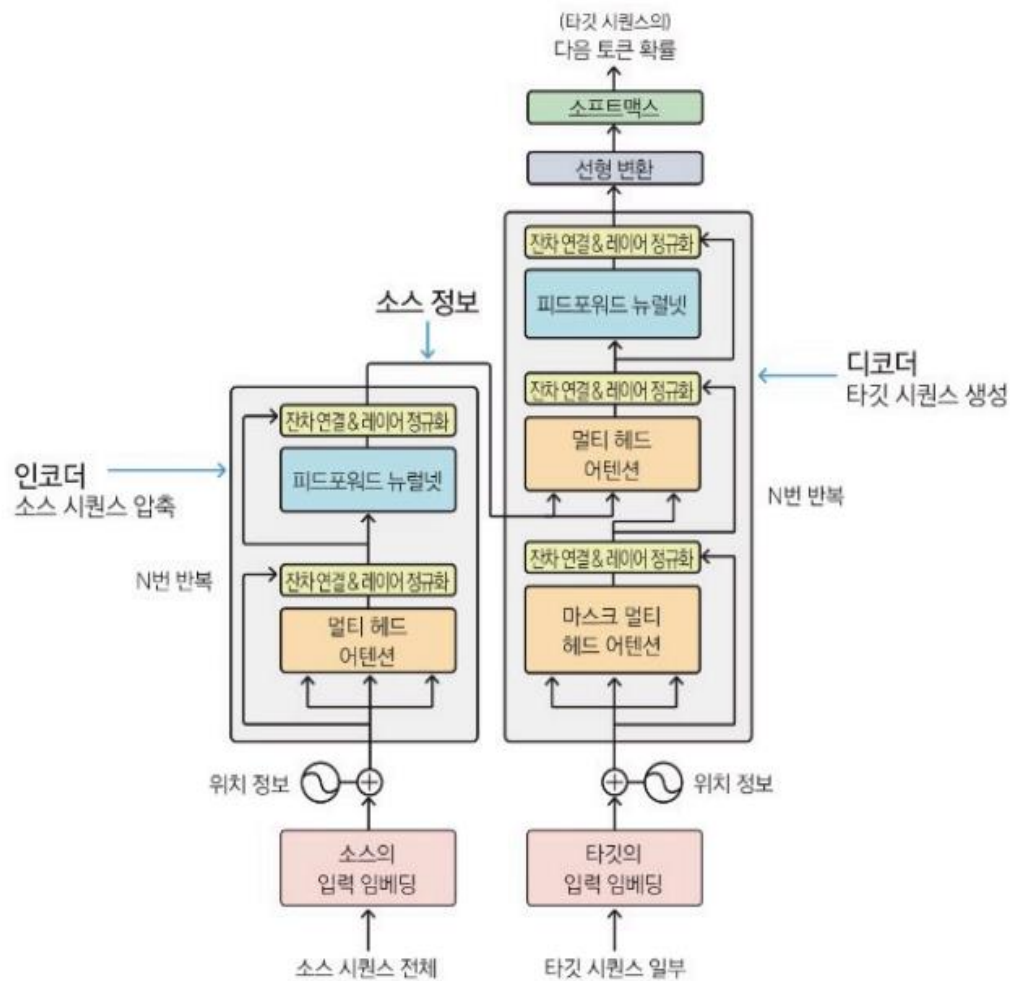


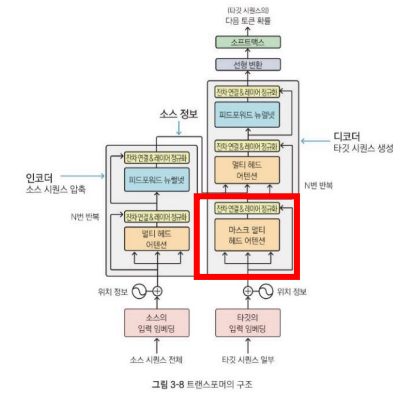
그림 3-8 트랜스포머의 구조

3. Decoder

1. Masked Multi-head Attention

- Masked Multi-head Attention

- 디코딩 단계에서 셀프 어텐션은 Query 토큰보다 뒤에 위치한 토큰들에 대한 정보는 가용하지 않다고 가정하고 해당 부분을 전부 마스킹Masking 처리



Encoder

Input	Thinking	Machines
Embedding	x_1 [] [] [] []	x_2 [] [] [] []
Queries	q_1 [] [] []	q_2 [] [] []
Keys	k_1 [] [] []	k_2 [] [] []
Values	v_1 [] [] []	v_2 [] [] []
Score	$q_1 \cdot k_1 = 112$	$q_1 \cdot k_2 = 96$
Divide by $8 (\sqrt{d_k})$	14	12
Softmax	0.88	0.12
Softmax X Value	v_1 [] [] []	v_2 [] [] []
Sum	z_1 [] [] []	z_2 [] [] []

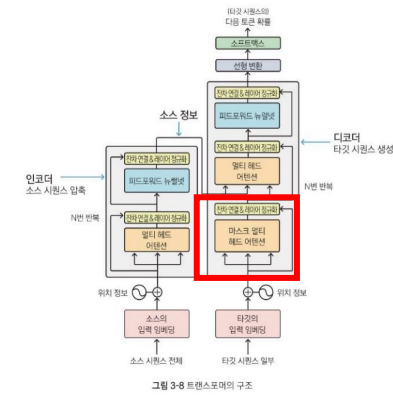
Decoder

Input	Thinking	Machines
Embedding	x_1 [] [] [] []	x_2 [] [] [] []
Queries	q_1 [] [] []	q_2 [] [] []
Keys	k_1 [] [] []	k_2 [] [] []
Values	v_1 [] [] []	v_2 [] [] []
Score	$q_1 \cdot k_1 = 112$	$q_1 \cdot k_2 = -\text{inf}$
Divide by $8 (\sqrt{d_k})$	14	$-\text{inf}$
Softmax	1	0
Softmax X Value	v_1 [] [] []	v_2 [] [] []
Sum	z_1 [] [] []	z_2 [] [] []

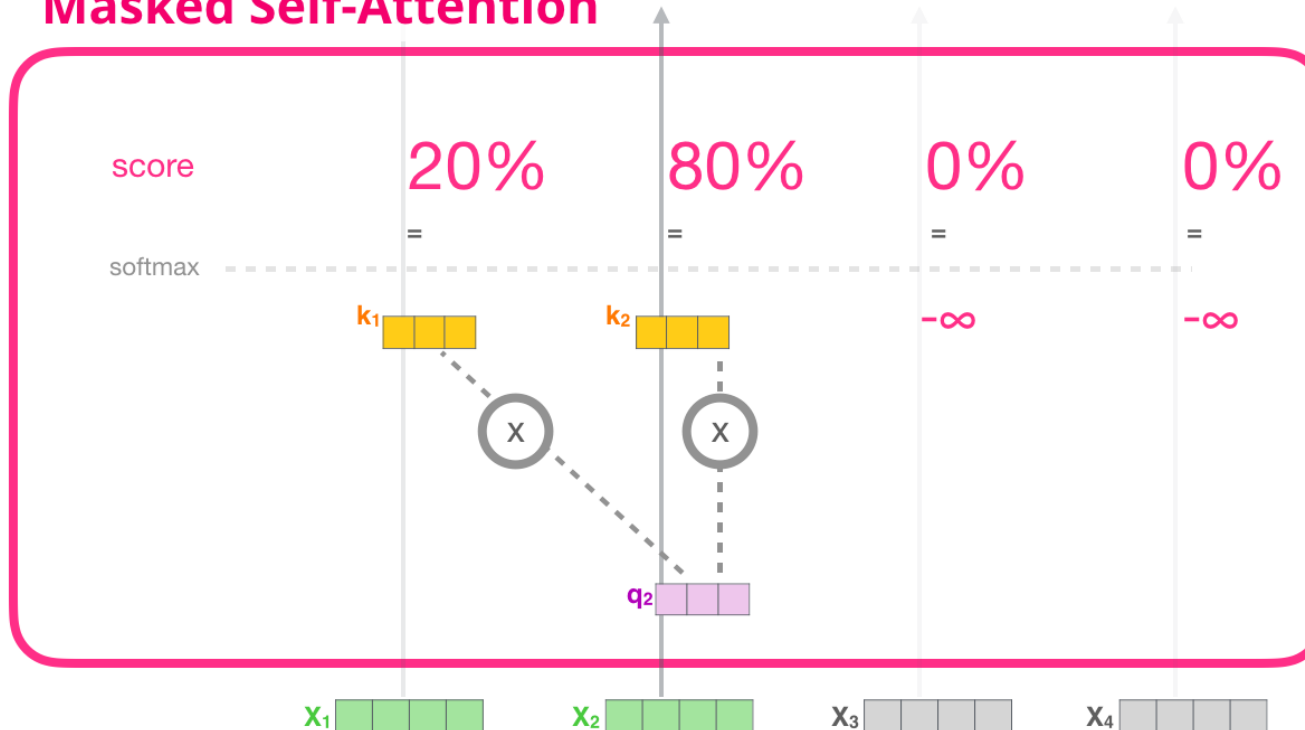


1. Masked Multi-head Attention

- Masked Multi-head Attention

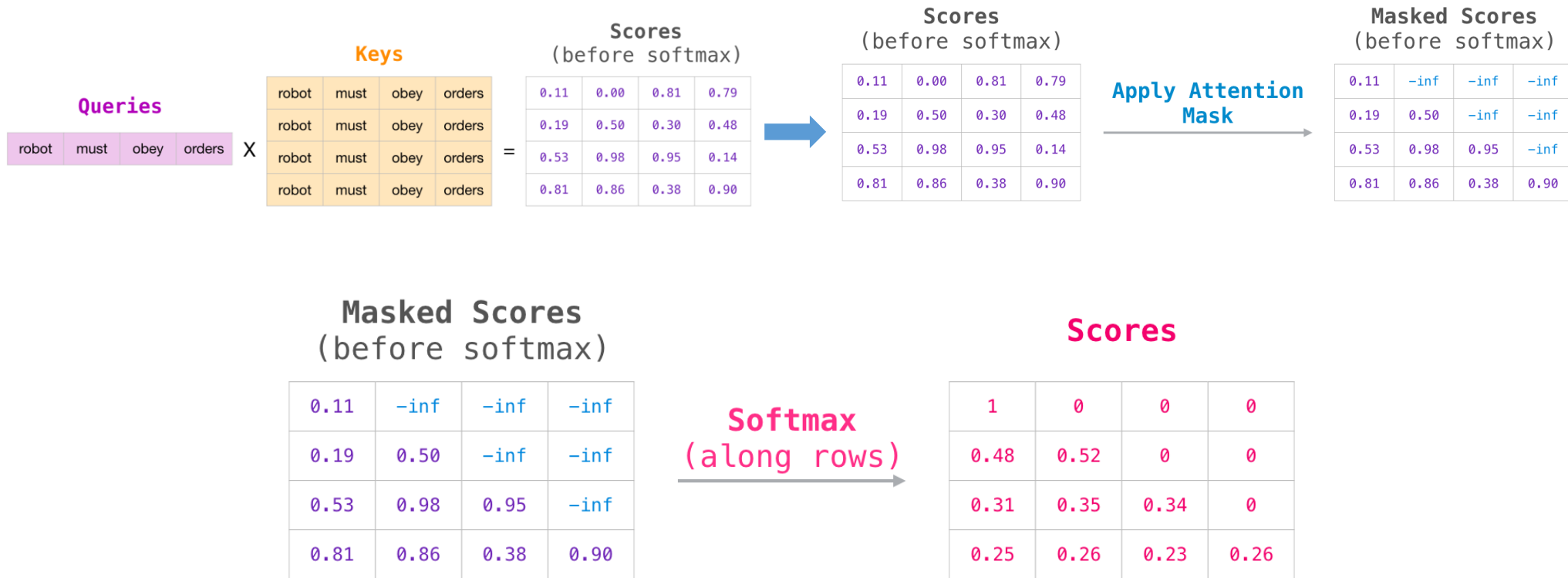
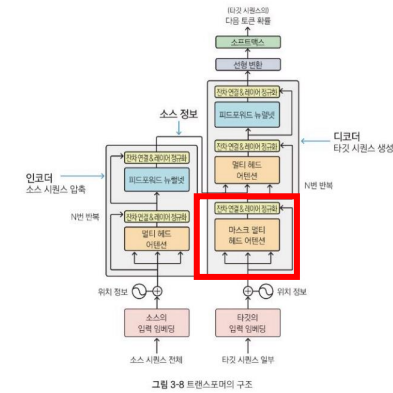


Masked Self-Attention



1. Masked Multi-head Attention

- Masked Multi-head Attention

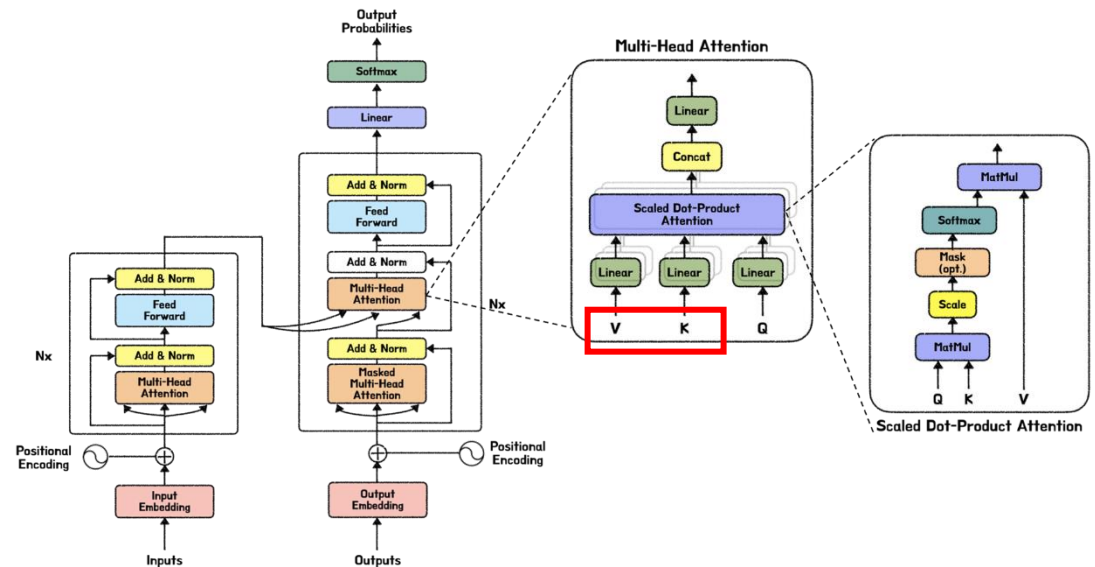
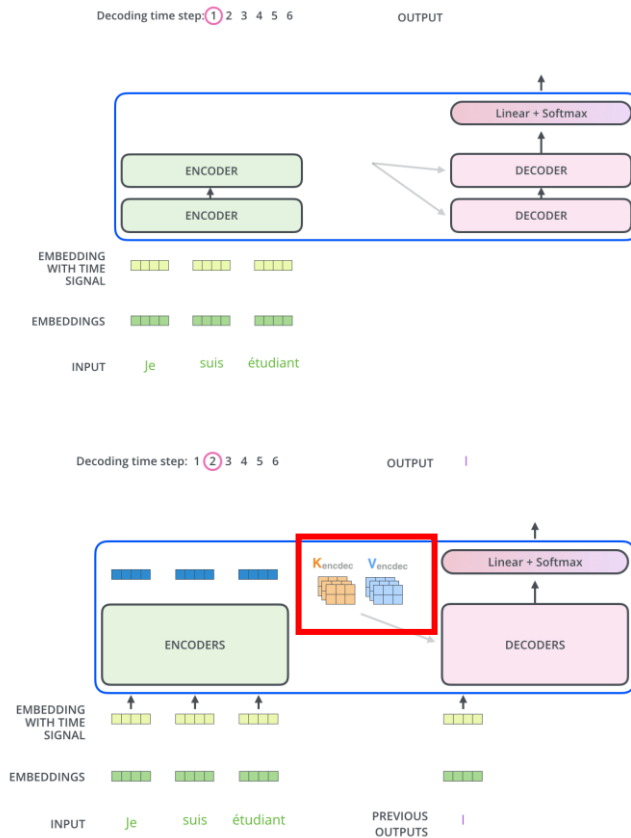


03

Decoder

2. Encoder-Decoder Attention

- Multi-head Attention with Encoder Outputs



3. The Final Linear and Softmax Layer

- The Final Linear and Softmax Layer

- Linear layer

단순 FFNN 형태로서 마지막 디코더의 출력 결과물을 이용하여
모든 단어들의 출력 확률을 산출하기 위해 차원을 늘리는 역할을 수행

- Softmax layer:

개별 단어들의 출력 확률 반환

Which word in our vocabulary
is associated with this index?

Get the index of the cell
with the highest value
(argmax)

am

5

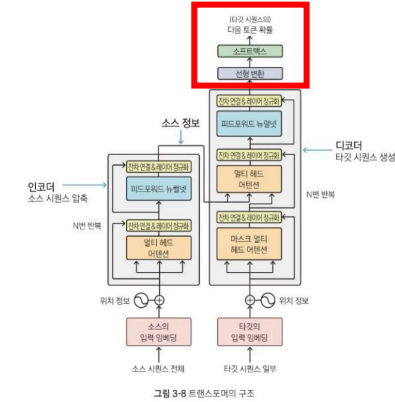
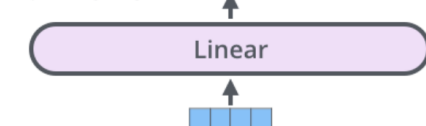
log_probs



logits



Decoder stack output



Summarize

