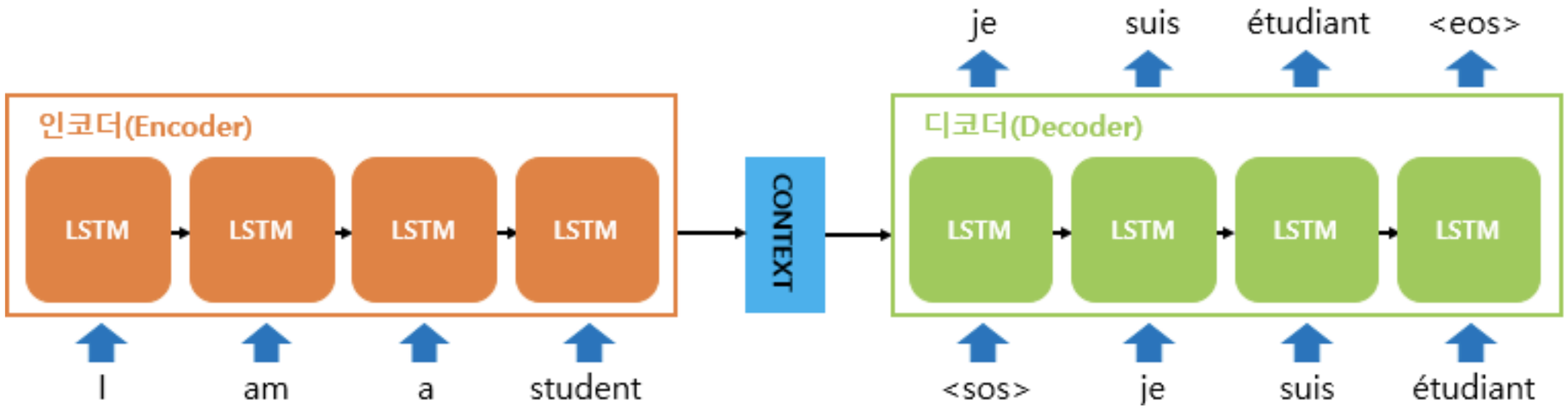
AI ROBOTICS KR : NLP STUDY

INHWAN LEE

TRANSFORMER

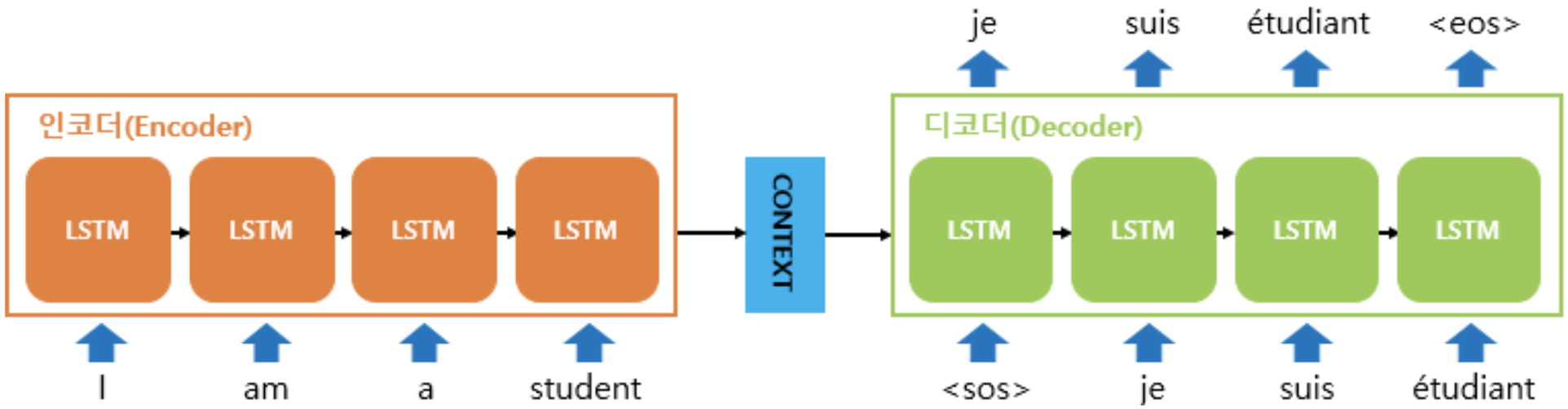
CONTENTS

SEQ2SEQ MODEL



(image source : https://wikidocs.net/24996)

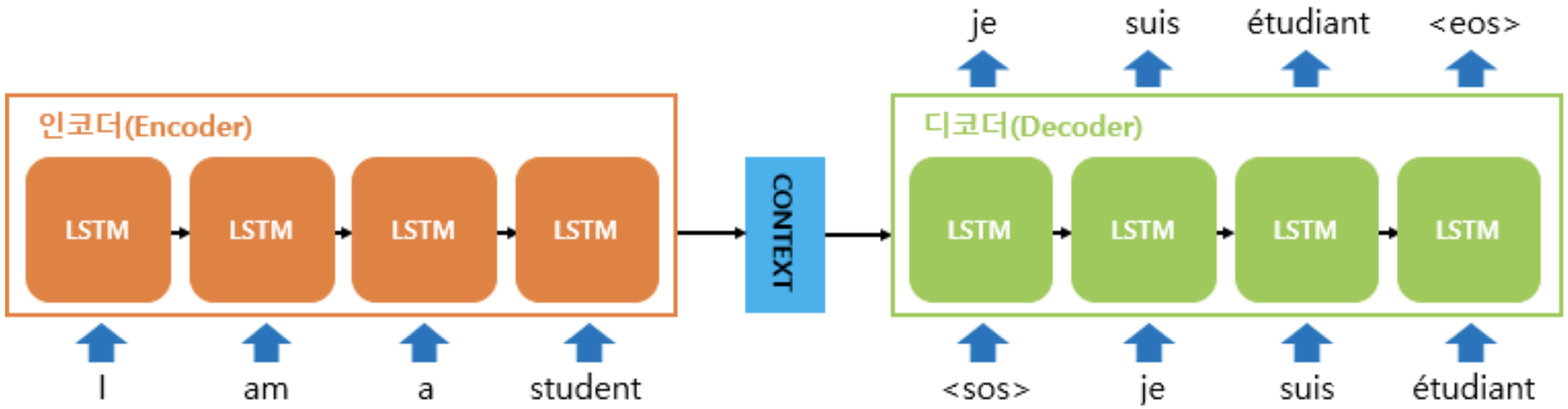
SEQ2SEQ MODEL



- 인코더와 디코더 모두 RNN 기반으로 구성
- 인코더는 입력값의 마지막 hidden state를 디코더에 전달

(image source : https://wikidocs.net/24996)

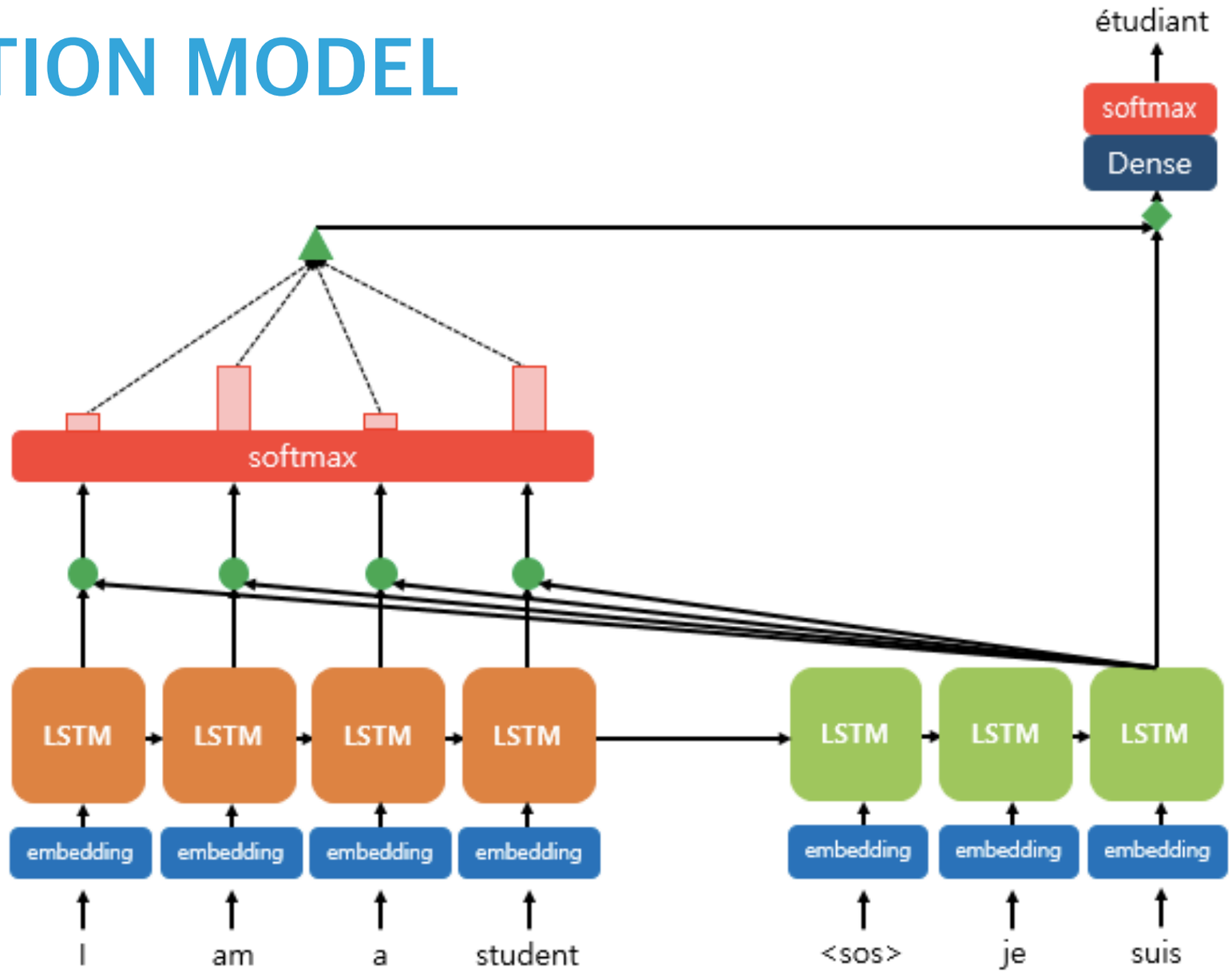
SEQ2SEQ MODEL



- LSTM의 특성 상 입력 문장이 긴 경우 정보의 유실이 발생
  (하나의 벡터에 모든 정보를 담는 것은 불가능)

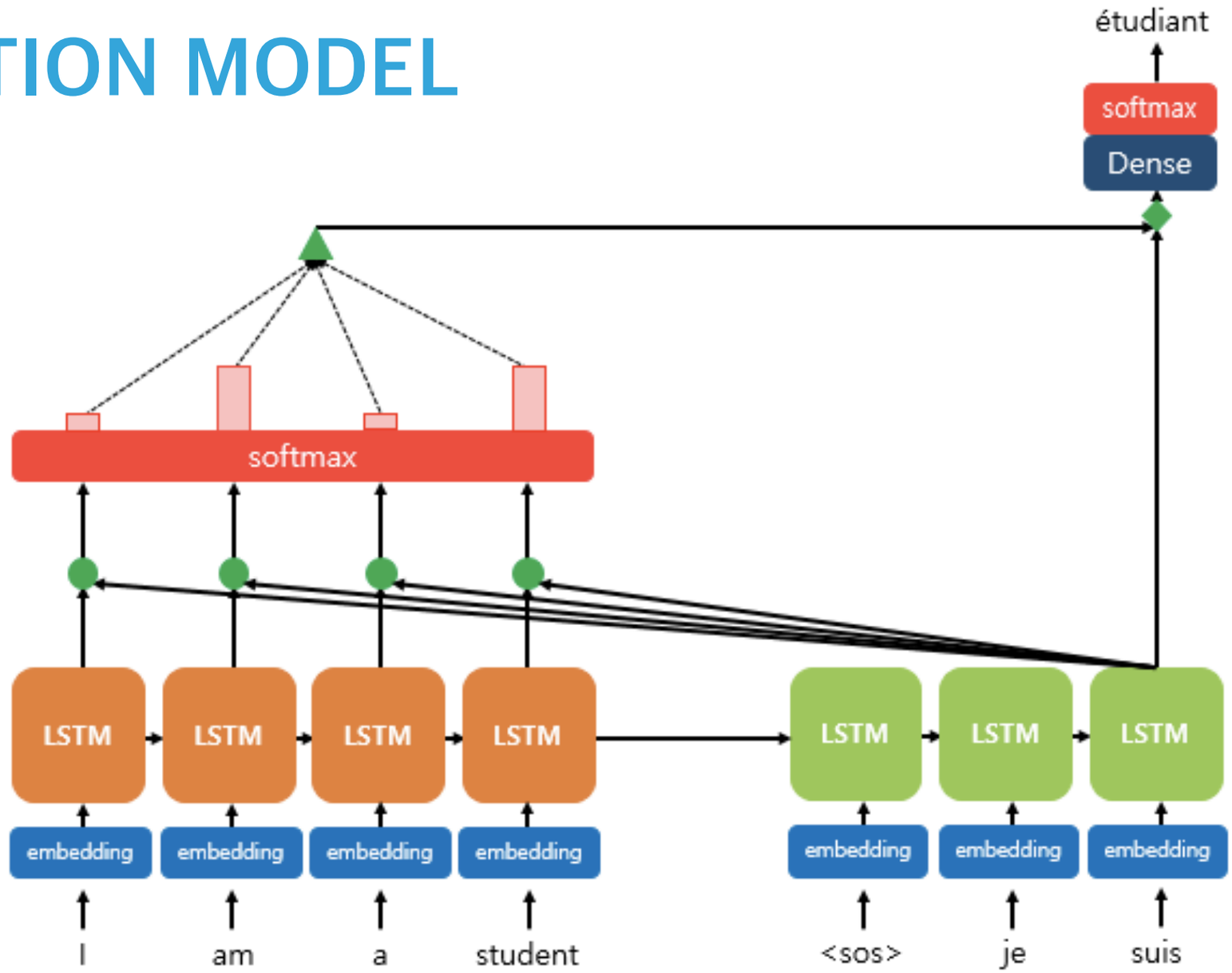
(image source : https://wikidocs.net/24996)

ATTENTION MODEL



(image source : https://wikidocs.net/22893)

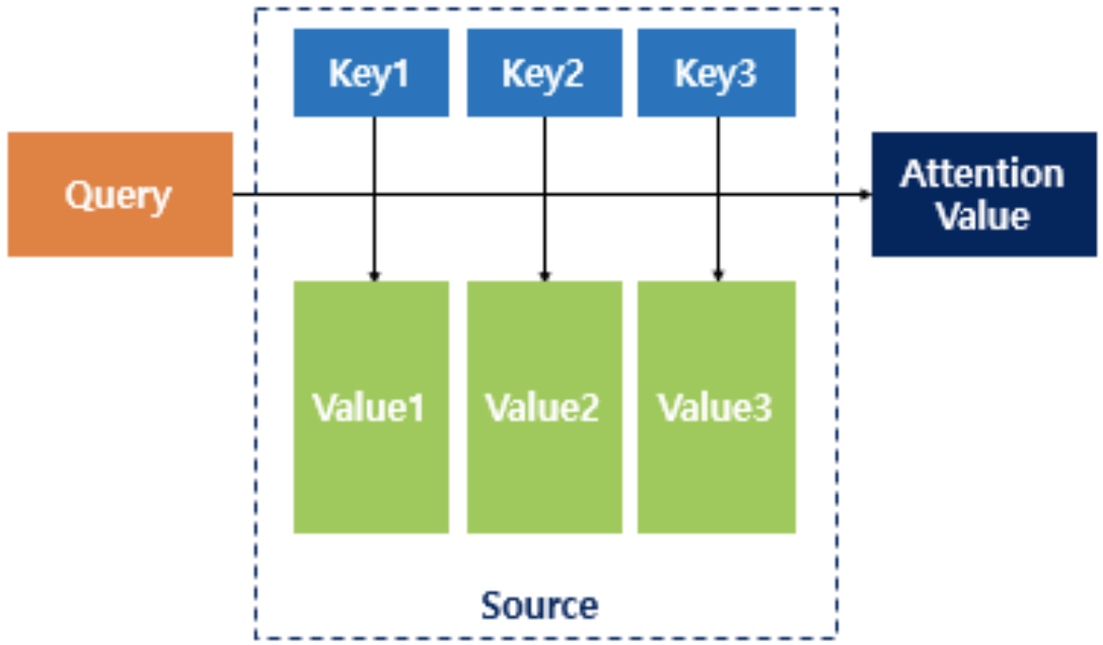
ATTENTION MODEL



- Seq2seq 모델에 Attention 계층을 추가하여 인코더의 정보를 추가로 전달

(image source : https://wikidocs.net/22893)

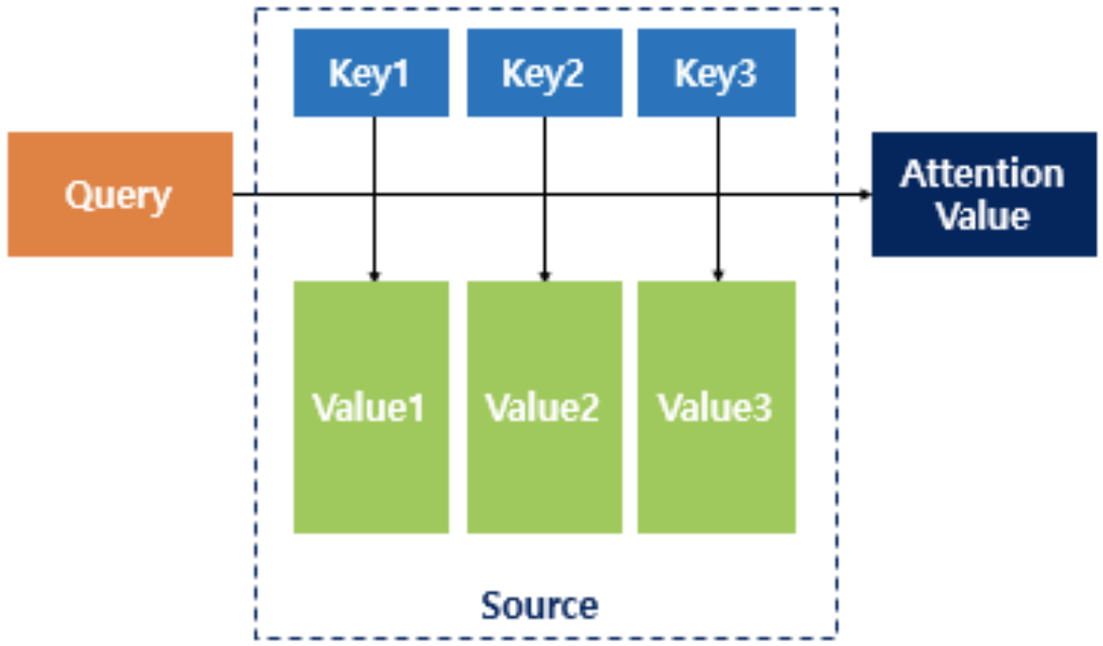
ATTENTION MODEL



- Query와 Key의 내적을 통해 Query를 처리하는 과정에서 Key(=Value)중 어떤 부분에 더 신경을 써야하는지 결정

(image source : https://wikidocs.net/22893)

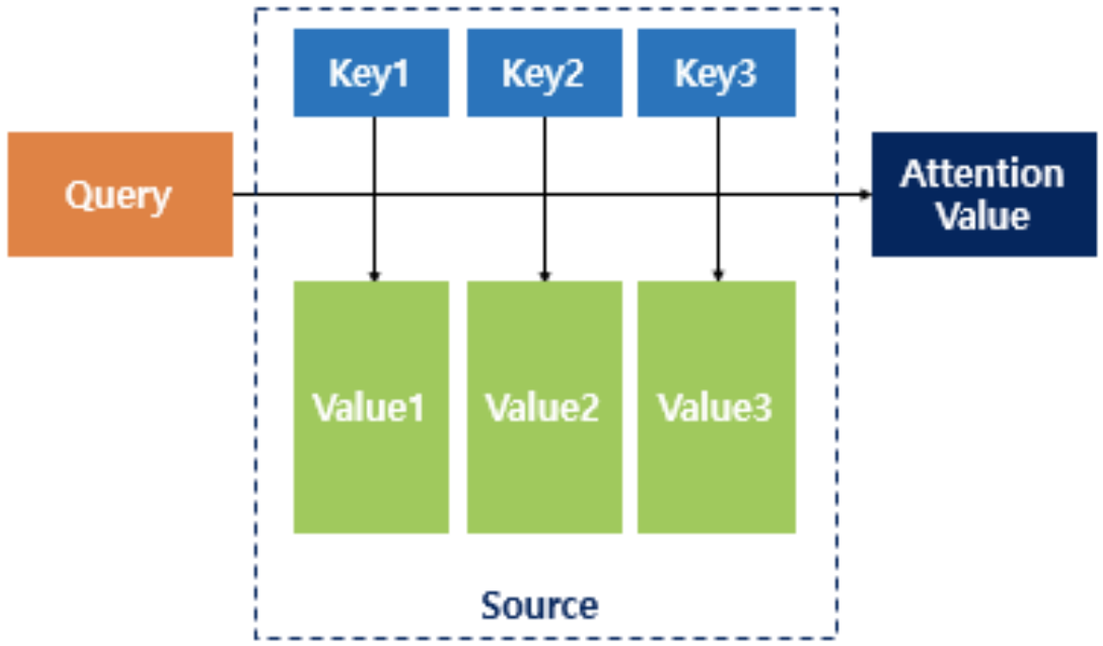
ATTENTION MODEL



- Query : 디코더의 t번째 hidden state
- Key & Value : 인코더의 hidden states

(image source : https://wikidocs.net/22893)

ATTENTION MODEL



- Bi-RNN 구조를 사용하고 있어 계산량이 많고 LSTM이 가지는 한계를 극복하지 못함

(image source : https://wikidocs.net/22893)

CORE IDEA

RNN 구조가 가지는 단점을 어떻게 극복할 수 있을까?

CORE IDEA

RNN 구조가 가지는 단점을 어떻게 극복할 수 있을까?

Attention is All You Need

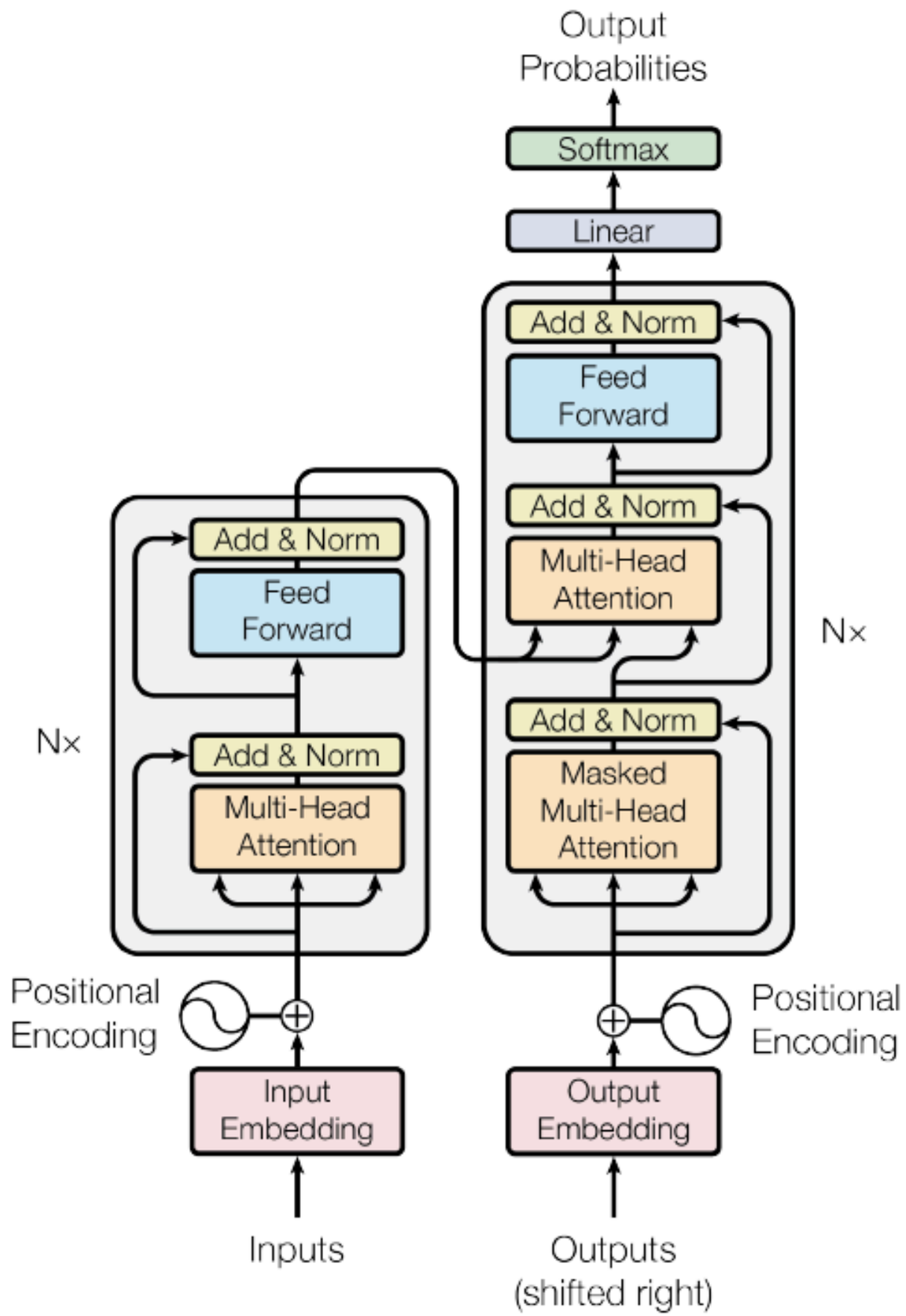
CORE IDEA

RNN 구조가 가지는 단점을 어떻게 극복할 수 있을까?

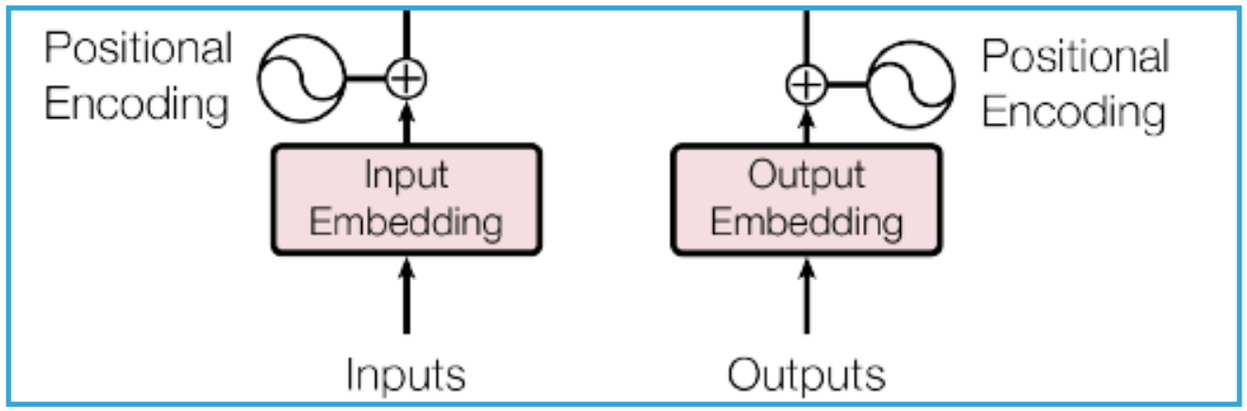
Attention is All You Need

- RNN 구조를 일절 사용하지 않고 모든 것을 Attention으로 해결

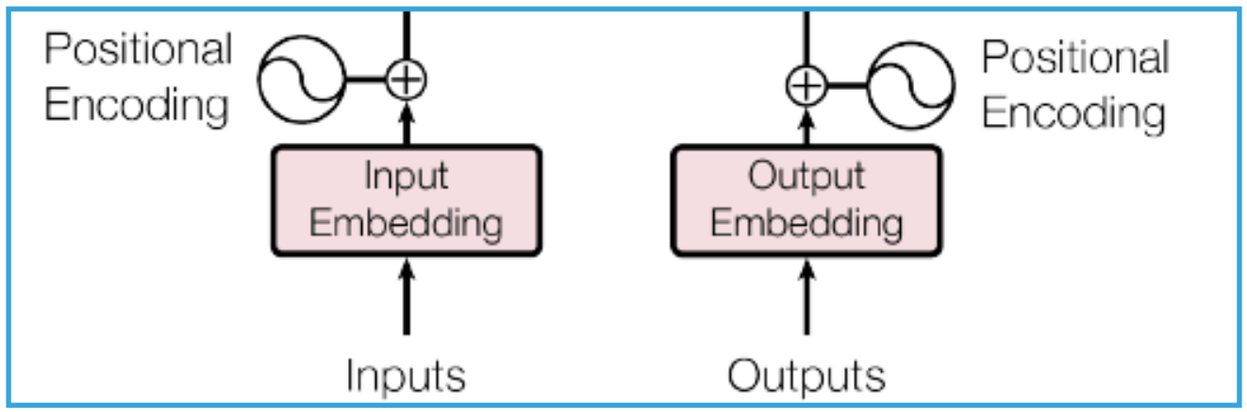
STRUCTURE


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

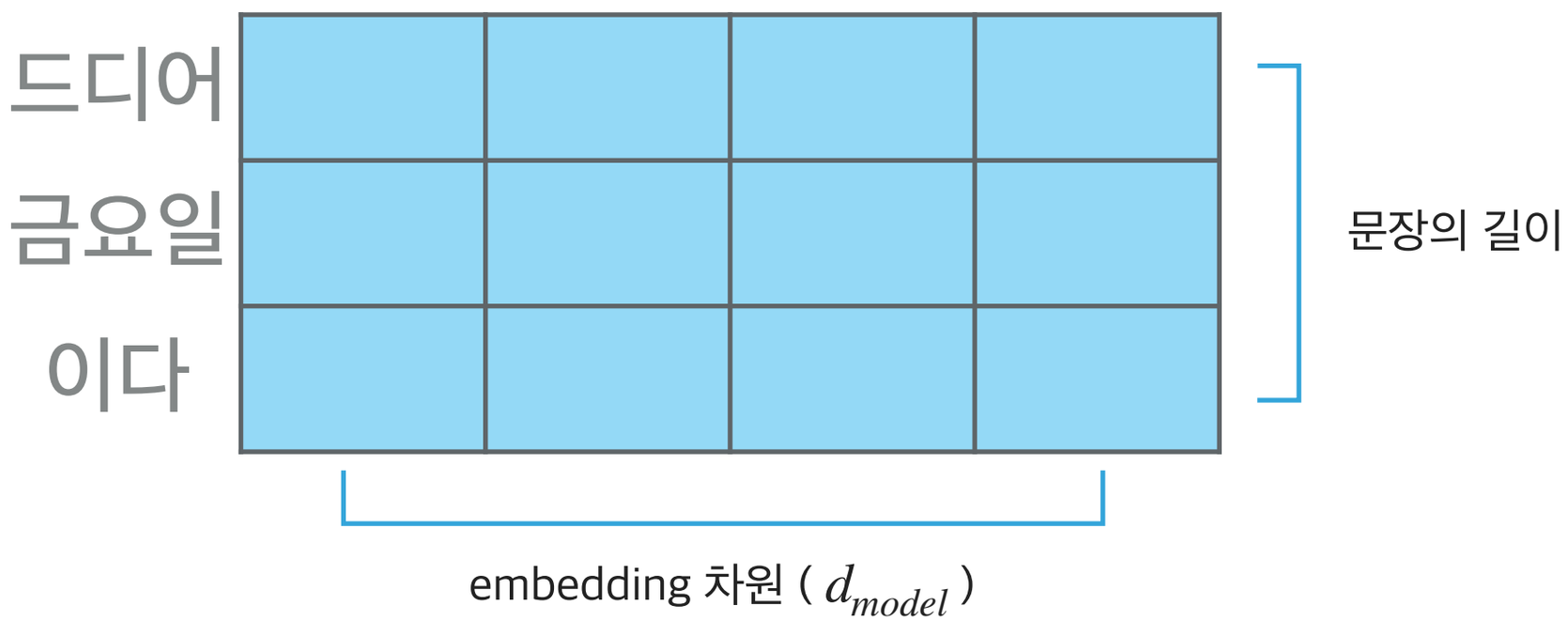
INPUTS & OUTPUTS



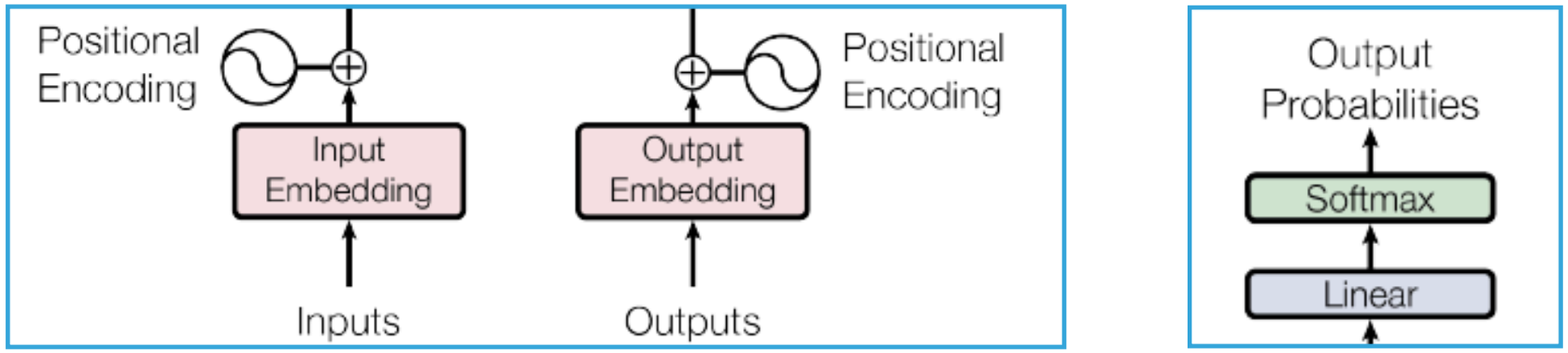
INPUTS & OUTPUTS



각 문장을 word embedding을 이용하여 다음 형태로 표현



INPUTS & OUTPUTS

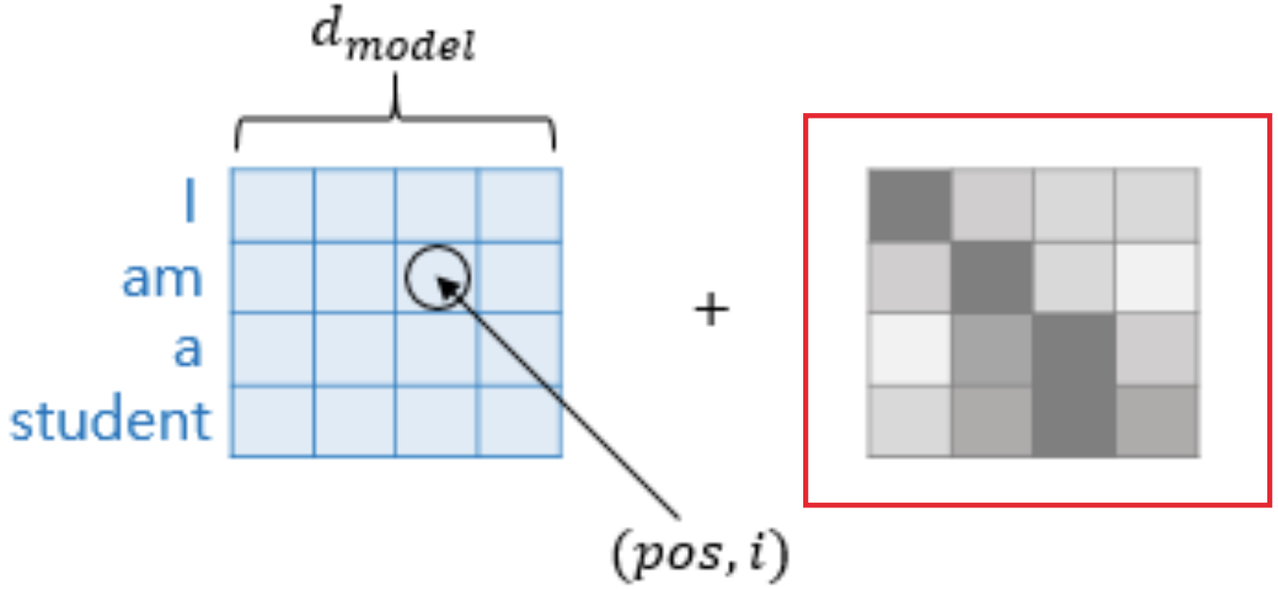


- word embedding은 pre-train한 모델을 사용
- Inputs, Outputs, 최종 출력 Linear 계층 모두 동일한 가중치를 사용하여 계산량을 줄임

POSITIONAL ENCODING

RNN 구조를 사용하지 않기 때문에 문장 내 순서가 반영되지 않음

- 이를 극복하기 위해 Positional Encoding을 도입



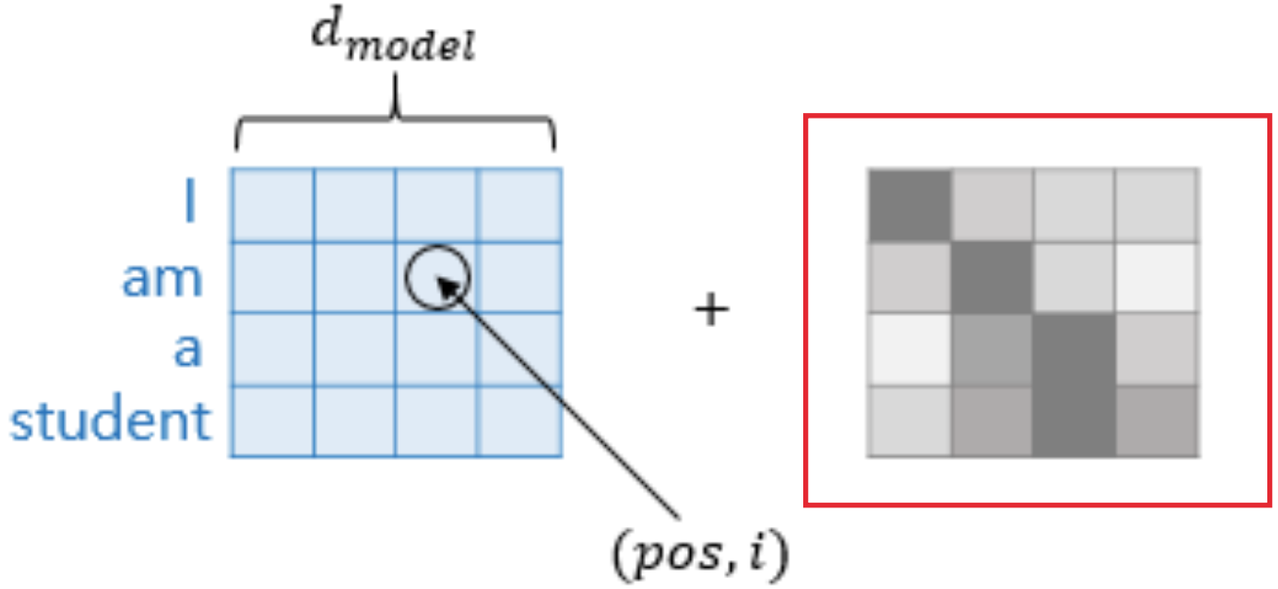
$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$

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

POSITIONAL ENCODING

RNN 구조를 사용하지 않기 때문에 문장 내 순서가 반영되지 않음

- 이를 극복하기 위해 Positional Encoding을 도입

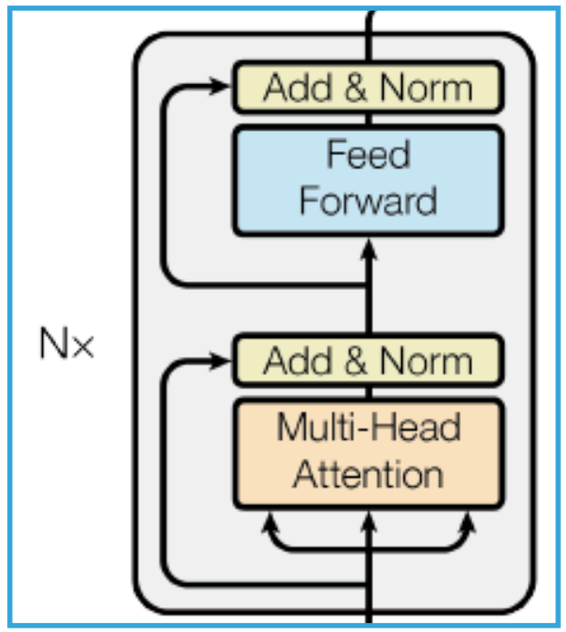


$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$

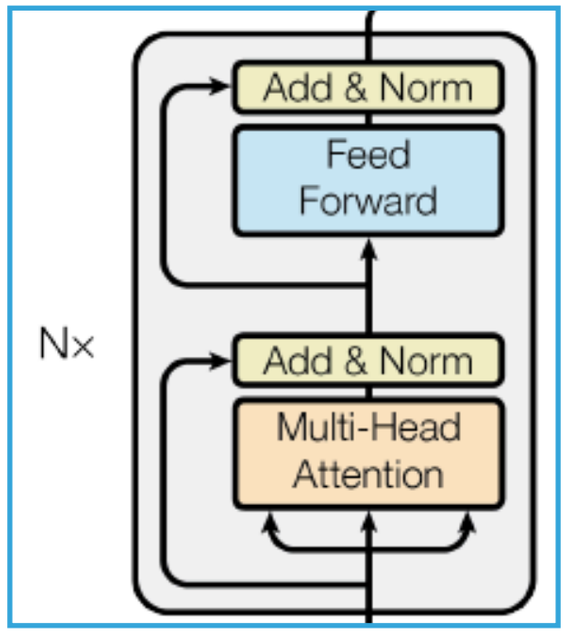
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

(*) $PE_{(pos+k,2i)} = \alpha \cdot PE_{(pos,2i)} + \beta \cdot PE_{(pos,2i+1)}$

ENCODER

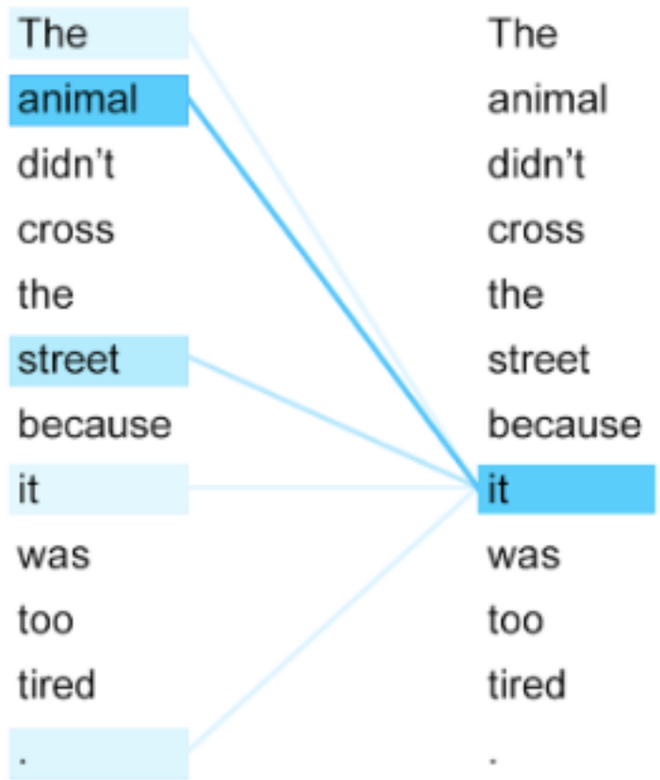


ENCODER

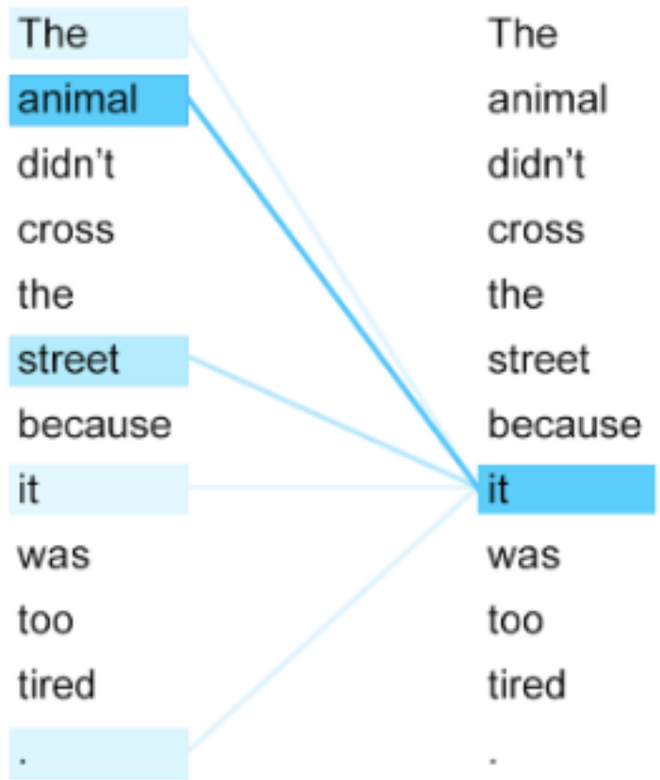


- 인코더는 위와 같은 Block을 여러층 쌓아올린 형태
- 논문에서는 6개의 layer로 구성하였으며, 차원은 모두 $d_{model} = 512$
- 각각의 Block은 Multi-head Self-Attention과 FFN 로 구성

SELF-ATTENTION



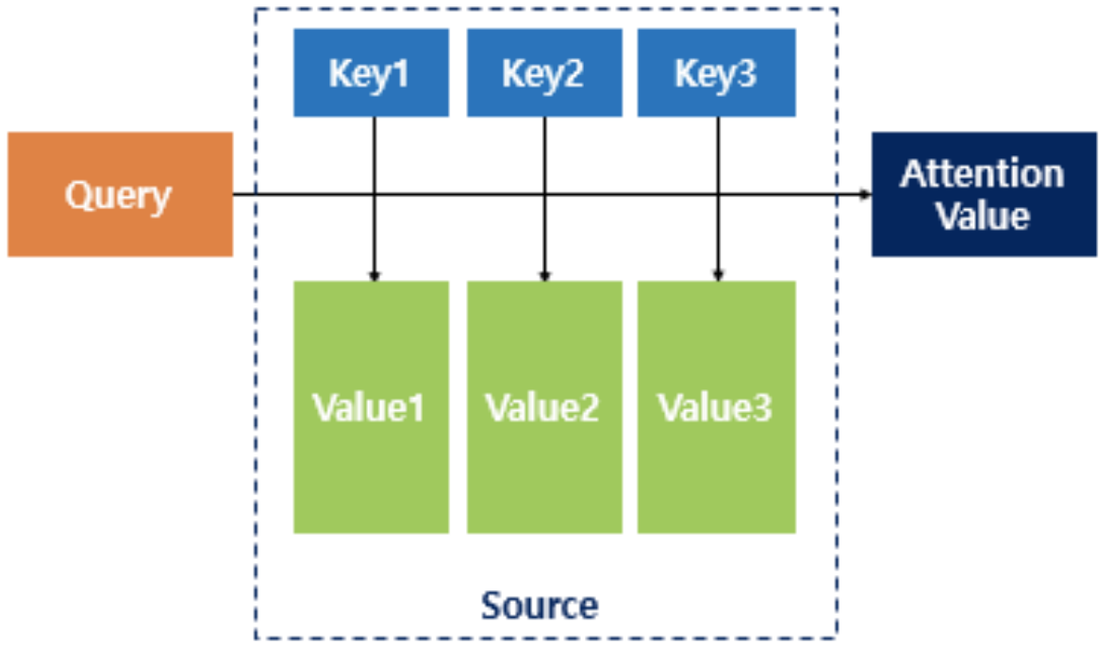
SELF-ATTENTION



Self-attention을 통해 모델은 입력 문장 내에서 단어간 관계를 학습

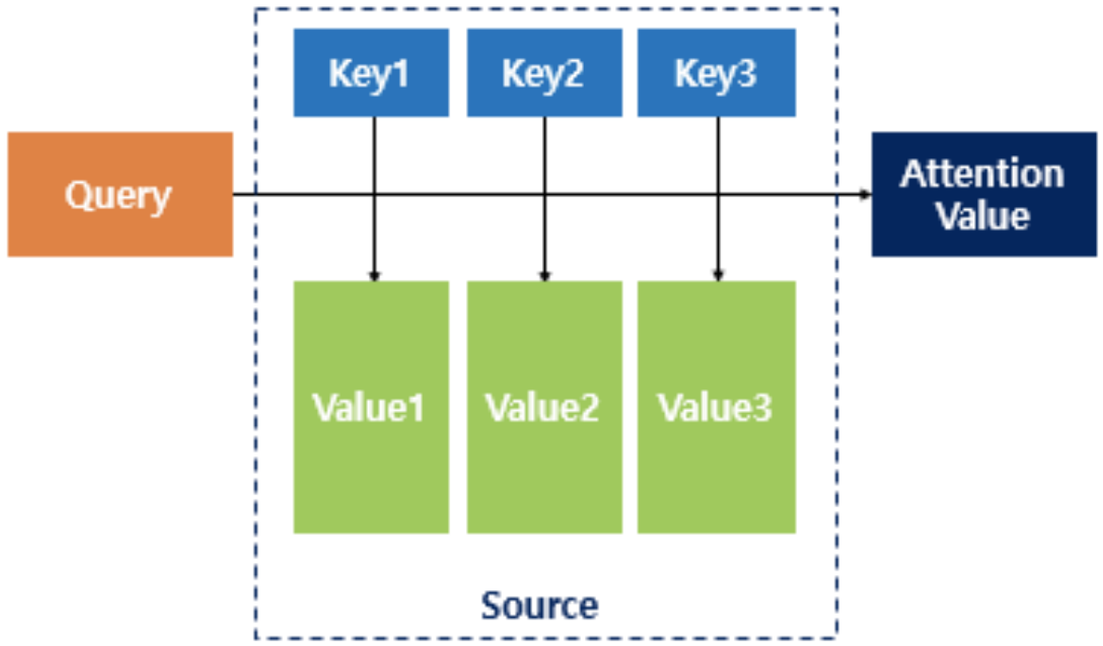
- 'it'은 'animal'과 'street' 중 전자에 가깝다!

SELF-ATTENTION

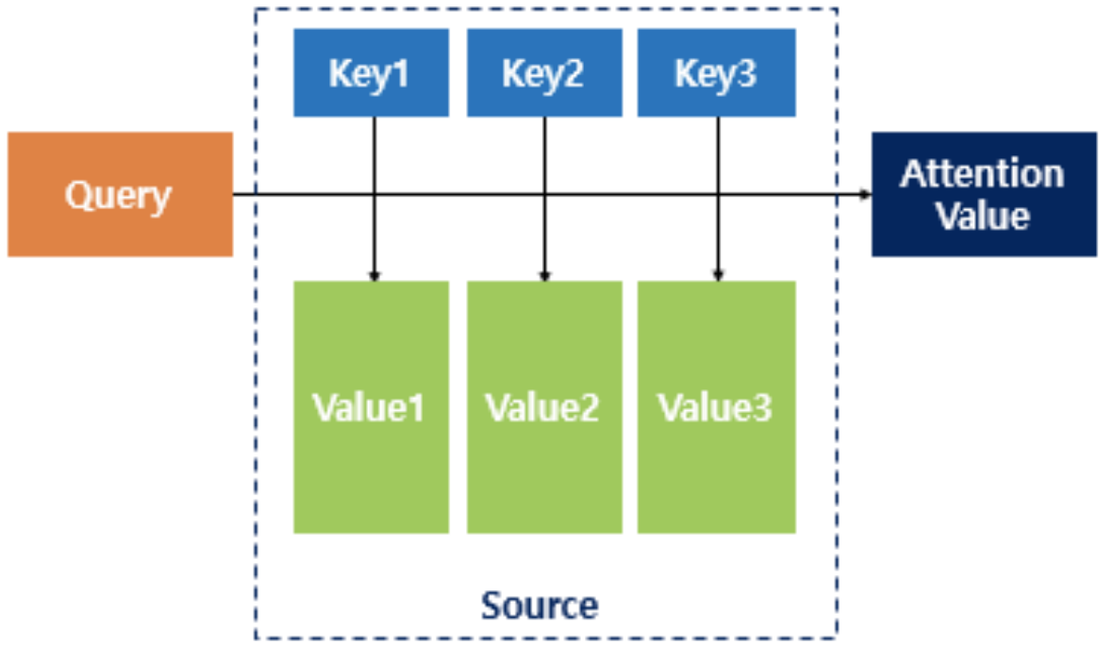


- Query와 Key의 내적을 통해 Query를 처리하는 과정에서 Key(=Value)중 어떤 부분에 더 신경을 써야하는지 결정

SELF-ATTENTION



SELF-ATTENTION



Query : 입력 문장의 단어 벡터들

Key : 입력 문장의 단어 벡터들

Value : 입력 문장의 단어 벡터들

SELF-ATTENTION EXAMPLE

- Q, K, V = ‘드디어 금요일 이다’.split()

SELF-ATTENTION EXAMPLE

- Q, K, V = ‘드디어 금요일 이다’.split()

$$Softmax\left(Q \cdot K^t\right) \cdot V =$$

	드디어	금요일	이다
드디어	0.2	0.7	0.1
금요일	…	…	…
이다	…	…	…

$\cdot$

$V_{드디어}$
$V_{금요일}$
$V_{이다}$

SELF-ATTENTION EXAMPLE

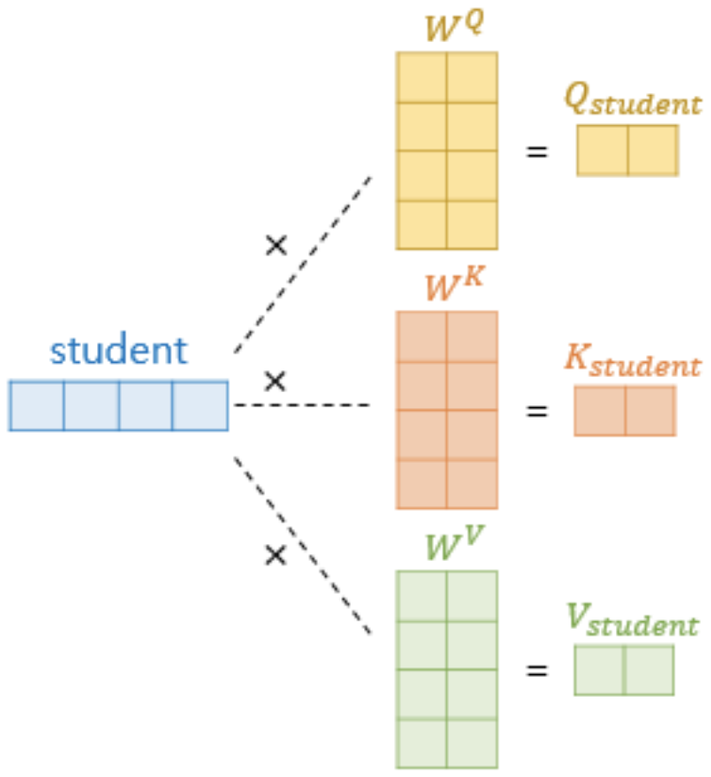
- Q, K, V = ‘드디어 금요일 이다’.split()

$$Softmax\left(Q \cdot K^t\right) \cdot V$$

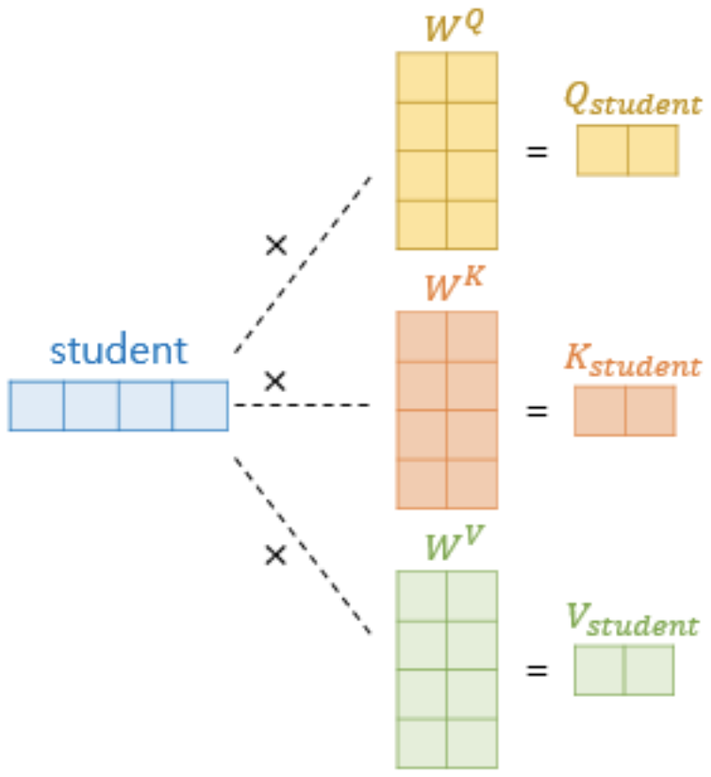
$$= \begin{array}{c|ccc} & \text{드디어} & \text{금요일} & \text{이다} \\ \hline \text{드디어} & 0.2 & 0.7 & 0.1 \\ \text{금요일} & \cdots & \cdots & \cdots \\ \text{이다} & \cdots & \cdots & \cdots \end{array} \cdot \begin{array}{c} \\ \hline V_{\text{드디어}} \\ V_{\text{금요일}} \\ V_{\text{이다}} \end{array}$$

$$= \begin{array}{c|c} \text{드디어} & 0.2*V_{\text{드디어}} + 0.7*V_{\text{금요일}} + 0.1*V_{\text{이다}} \\ \text{금요일} & \cdots \\ \text{이다} & \cdots \end{array}$$

SELF-ATTENTION

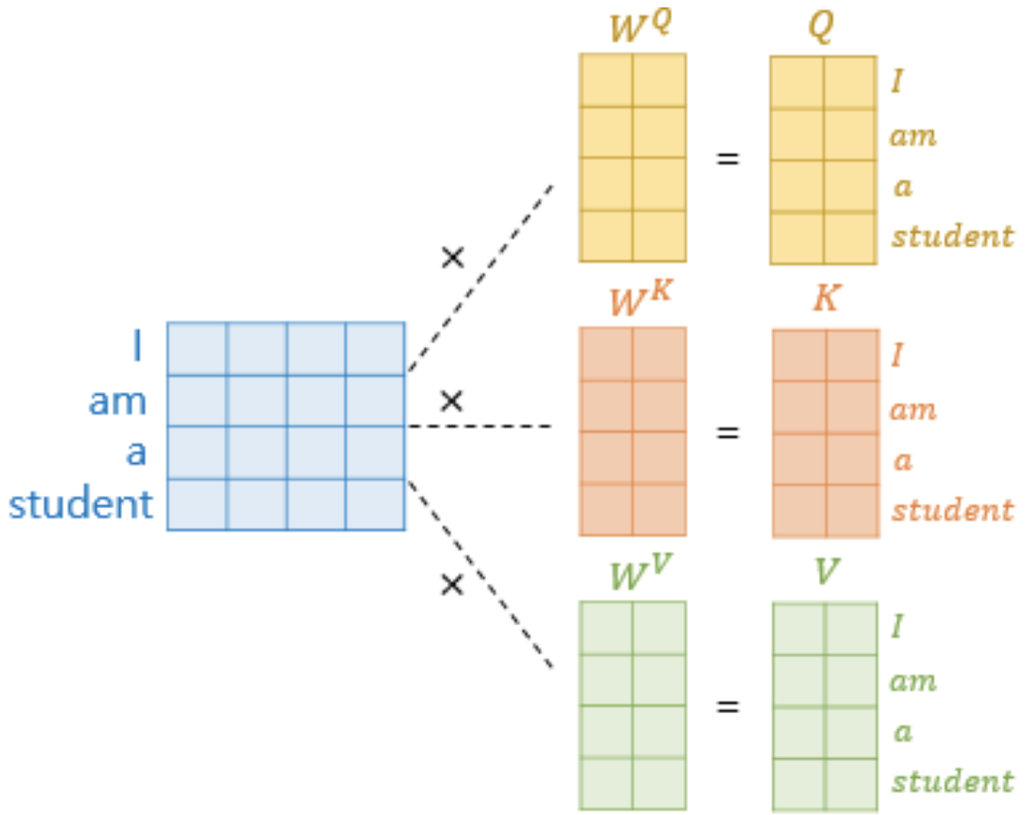


SELF-ATTENTION

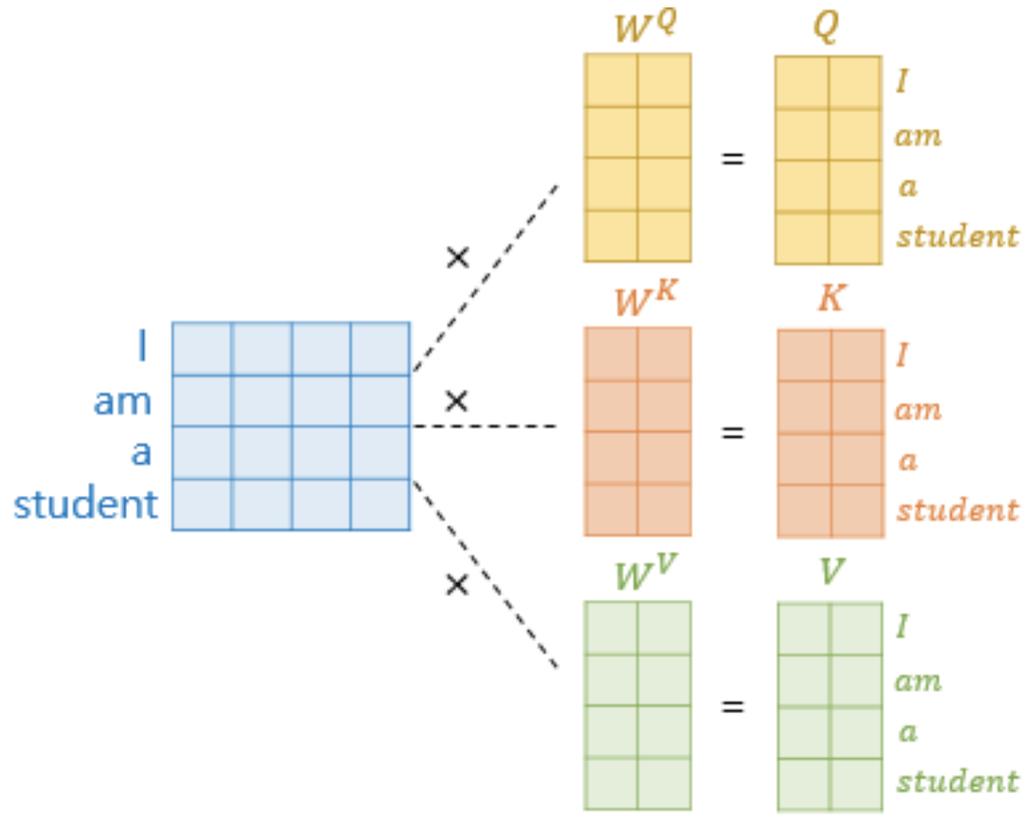


- 입력된 단어의 벡터 표현에 행렬(학습 대상)을 곱하여 Q, K, V를 얻음 (Q, K $\in \mathbb{R}^{d_k}$, V $\in \mathbb{R}^{d_v}$)

SCALED-DOT PRODUCT ATTENTION

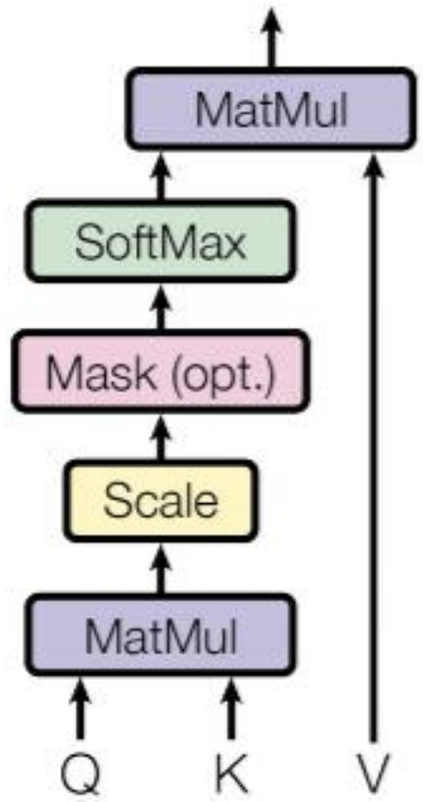


SCALED-DOT PRODUCT ATTENTION



- 단어 벡터별 내적 연산을 문장 단위, 행렬 곱셈으로 확대

SCALED-DOT PRODUCT ATTENTION



$$Attention(Q, K, V) = Softmax\left(\frac{Q \cdot K^t}{\sqrt{d_k}}\right) \cdot V$$

WHY SCALED?

$$Attention(Q, K, V) = Softmax\left(\frac{Q \cdot K^t}{\sqrt{d_k}}\right) \cdot V$$

WHY SCALED?

$$Attention(Q, K, V) = Softmax\left(\frac{Q \cdot K^t}{\sqrt{d_k}}\right) \cdot V$$

두 벡터 $x = (x_i), y = (y_i)$ 가 $y = Softmax(x)$ 라고 가정하자.

WHY SCALED?

$$Attention(Q, K, V) = Softmax\left(\frac{Q \cdot K^t}{\sqrt{d_k}}\right) \cdot V$$

두 벡터 $x = (x_i), y = (y_i)$ 가 $y = Softmax(x)$ 라고 가정하자.

($y_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$)

WHY SCALED?

$$Attention(Q, K, V) = Softmax\left(\frac{Q \cdot K^t}{\sqrt{d_k}}\right) \cdot V$$

두 벡터 $x = (x_i), y = (y_i)$ 가 $y = Softmax(x)$ 라고 가정하자.

($y_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$)

이 때, x 의 값이 커지면 y_i 중 값이 작은 것이 생긴다.

WHY SCALED?

$\frac{\partial y_i}{\partial x_i} = y_i(1 - y_i)$, $\frac{\partial y_i}{\partial x_j} = -y_i \cdot y_j$ 이므로, y_i 의 값이 작으면

gradient가 0에 가까워져 학습이 일어나지 않음!

WHY SCALED?

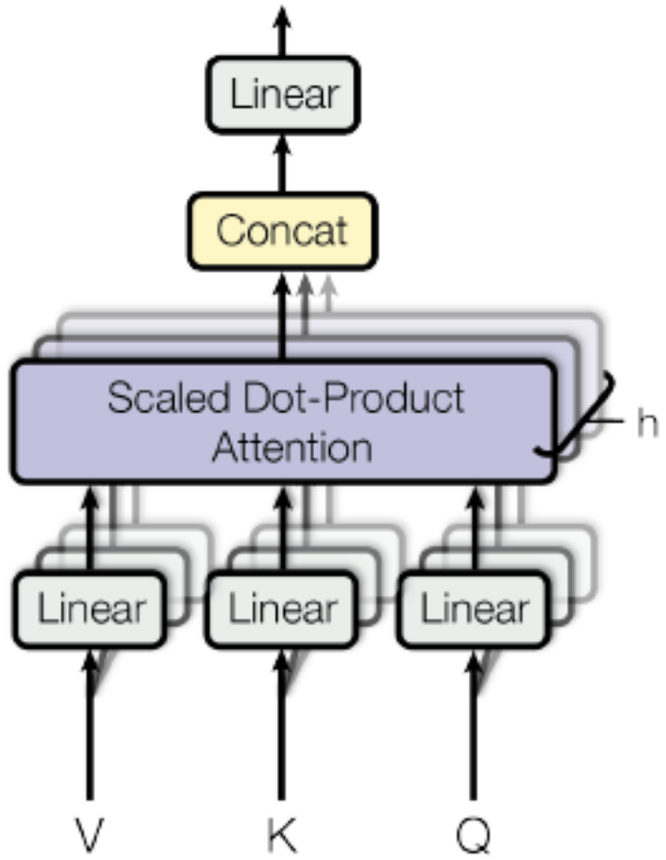
$\frac{\partial y_i}{\partial x_i} = y_i(1 - y_i), \quad \frac{\partial y_i}{\partial x_j} = -y_i \cdot y_j$ 이므로, y_i 의 값이 작으면

gradient가 0에 가까워져 학습이 일어나지 않음!

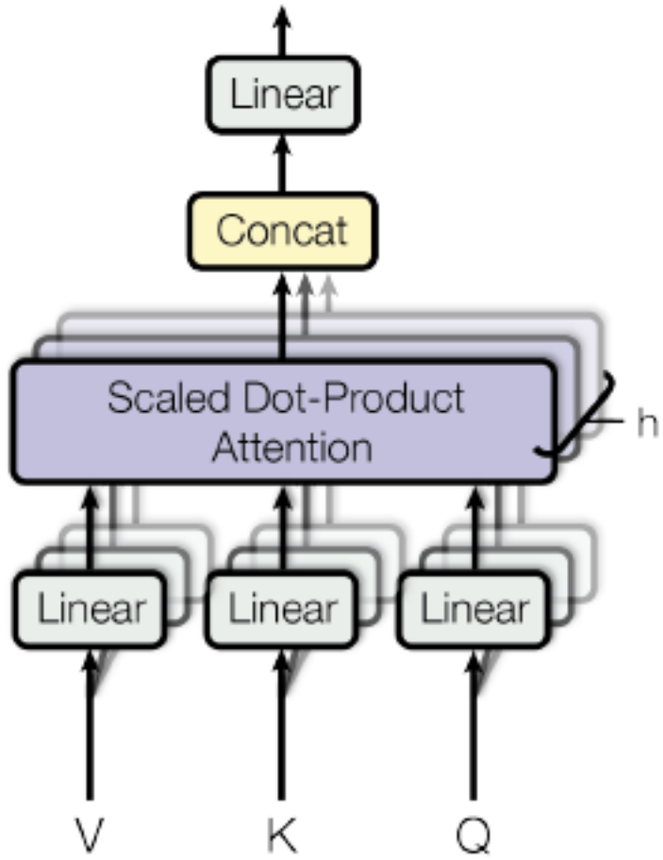
- 원활한 학습을 위해 scaling factor를 도입

  (차원이 증가하면 전체 내적값이 커짐 > 차원으로 나눠줌)

MULTI-HEAD ATTENTION

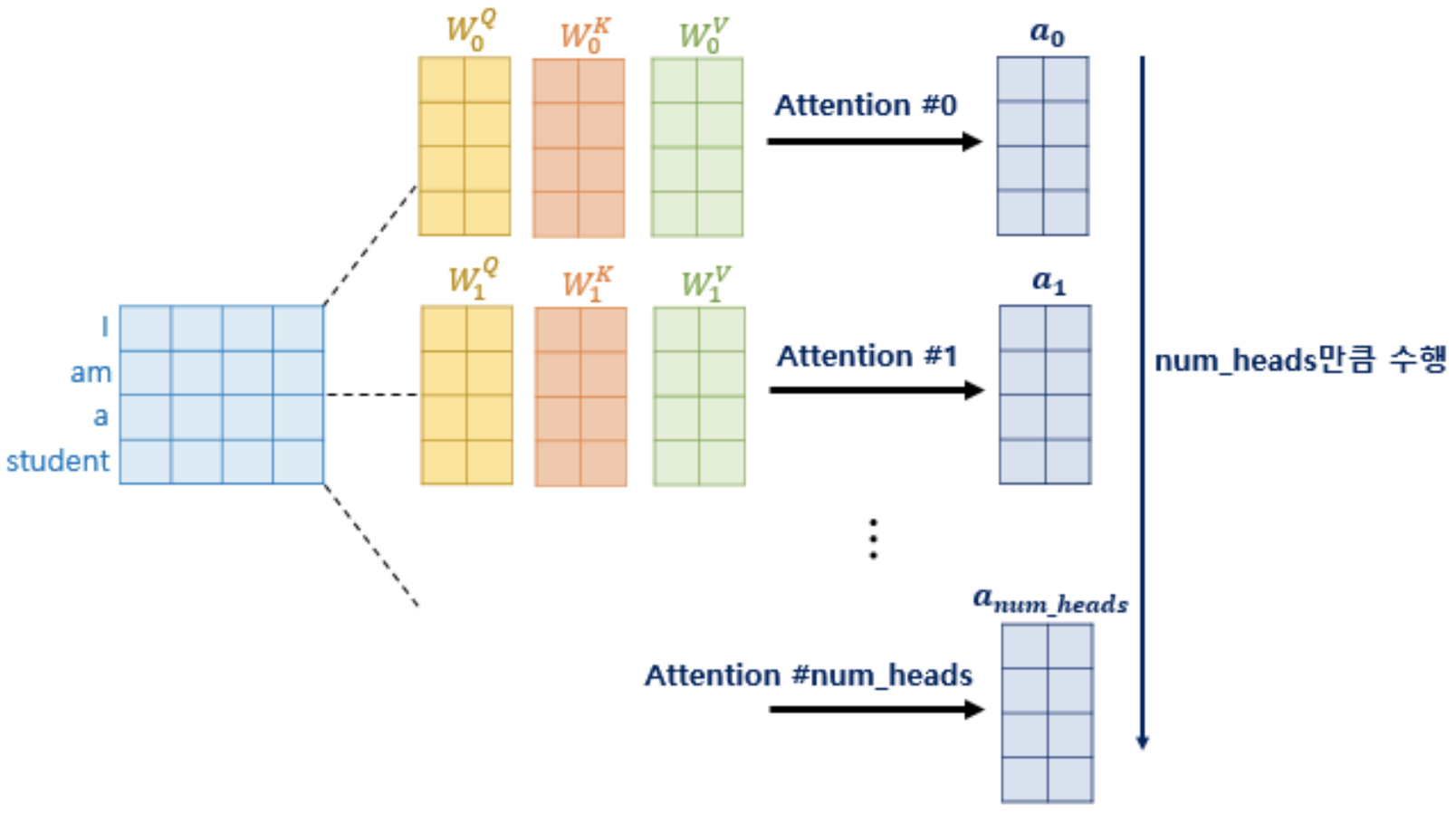


MULTI-HEAD ATTENTION

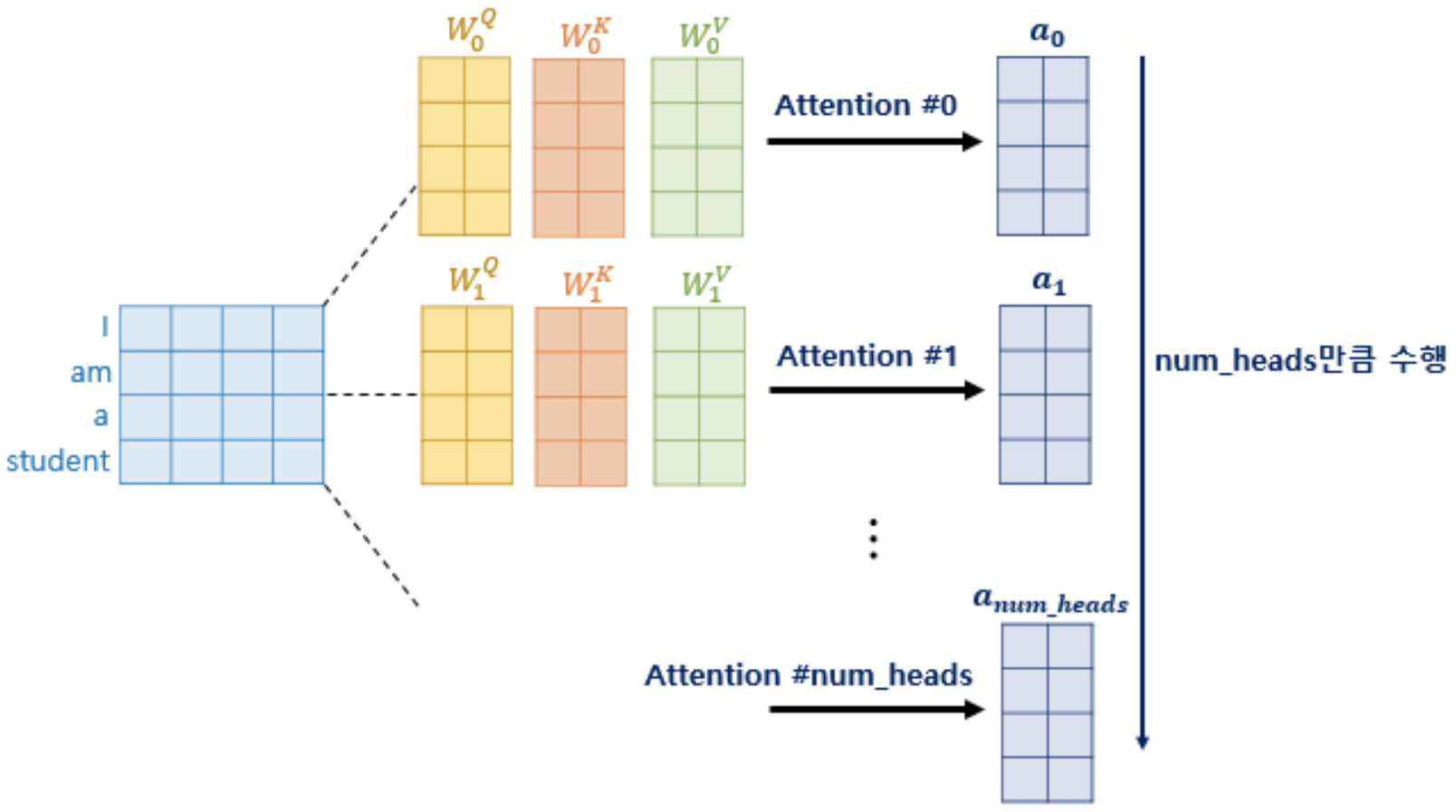


- 전체 차원에 대해서 한번 Attention을 적용하는 것이 아니라, 전체 차원을 h로 나눠서 Attention을 h번 적용!

MULTI-HEAD ATTENTION

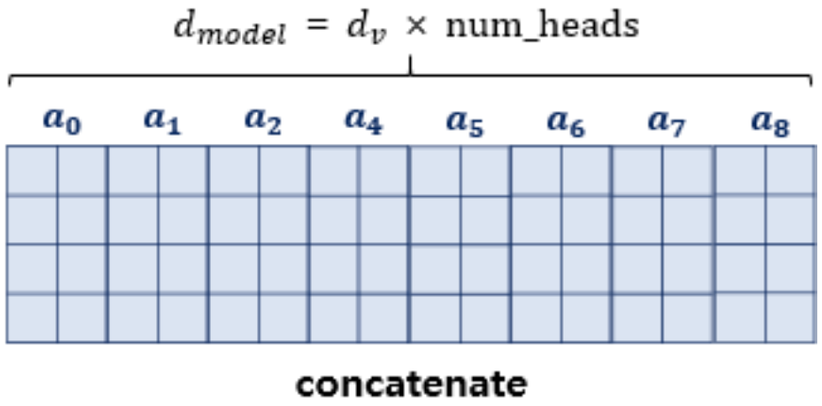


MULTI-HEAD ATTENTION



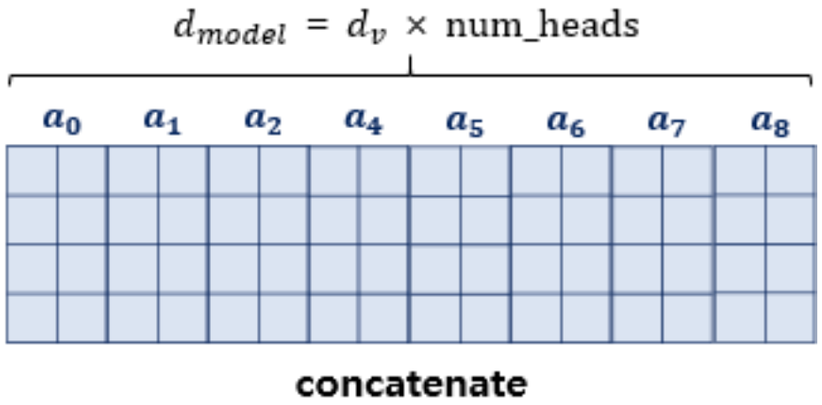
- Q, K, V 는 모두 같은 값을 복제하여 사용하나 가중치 행렬은 head별로 학습!
- 의사결정을 한 명이 하는 것이 아니라 다수결로 하는 것과 유사

MULTI-HEAD ATTENTION



- 각 head에서 산출한 결과 벡터를 병합하여 출력값으로 사용

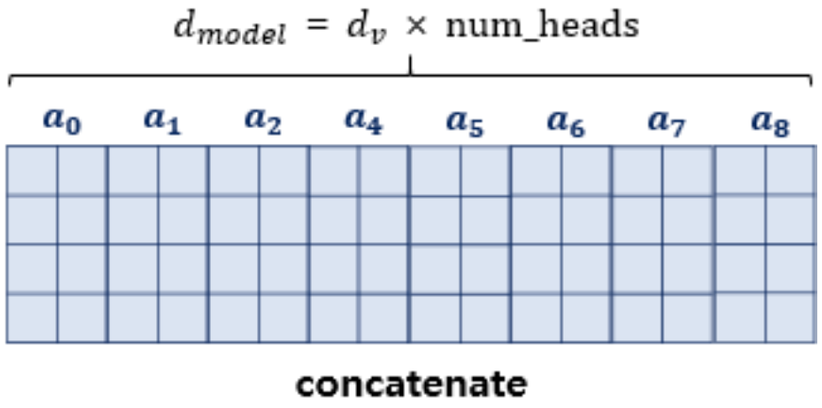
MULTI-HEAD ATTENTION



- 각 head에서 산출한 결과 벡터를 병합하여 출력값으로 사용

$$MultiHead(Q, K, V) = Concat\left(head_1, \ldots, head_h\right) \cdot W^O$$

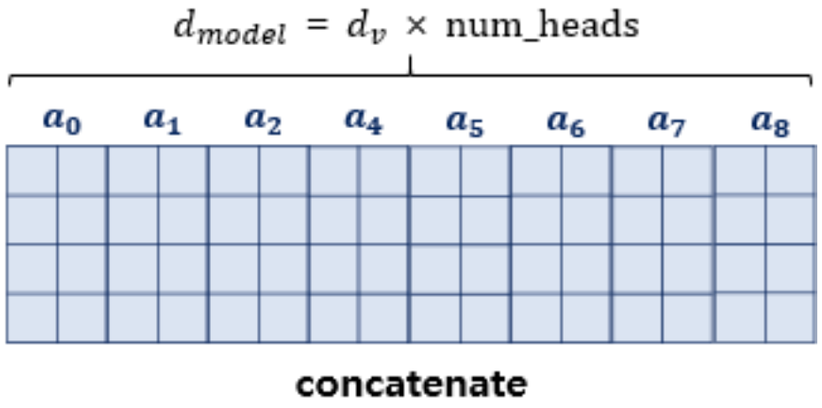
MULTI-HEAD ATTENTION



- 각 head에서 산출한 결과 벡터를 병합하여 출력값으로 사용

$$MultiHead(Q, K, V) = Concat\left(head_1, \ldots, head_h\right) \cdot W^O$$
$$where\ head_i = Attention(QW_i^Q, KW_i^K, VW_i^W)$$

MULTI-HEAD ATTENTION



- 각 head에서 산출한 결과 벡터를 병합하여 출력값으로 사용

$$MultiHead(Q, K, V) = Concat\left(head_1, \ldots, head_h\right) \cdot W^O$$

$$where\ head_i = Attention(QW_i^Q, KW_i^K, VW_i^W)$$

$$(W_i^Q, W_i^K \in \mathbb{R}^{d_{model} \times d_k},\ W_i^V \in \mathbb{R}^{d_{model} \times d_v},\ W^O \in \mathbf{R}^{hd_v \times d_{model}})$$

MULTI-HEAD ATTENTION

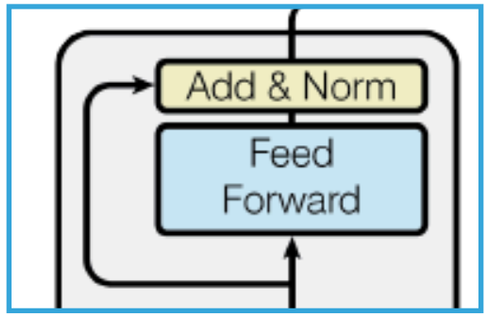
$$W_i^Q, W_i^K \in \mathbb{R}^{d_{model} \times d_k}, \; W_i^V \in \mathbb{R}^{d_{model} \times d_v}$$

MULTI-HEAD ATTENTION

$$W_i^Q, W_i^K \in \mathbb{R}^{d_{model} \times d_k}, \; W_i^V \in \mathbb{R}^{d_{model} \times d_v}$$

- 논문에서는 $h = 8, d_k = d_v = d_{model}/h = 64$ 를 사용하였으며, head의 갯수는 증가하였으나 각 head의 차원은 감소하여 전체적인 계산량은 비슷하다.
- 병렬 처리에 용이하다는 장점!

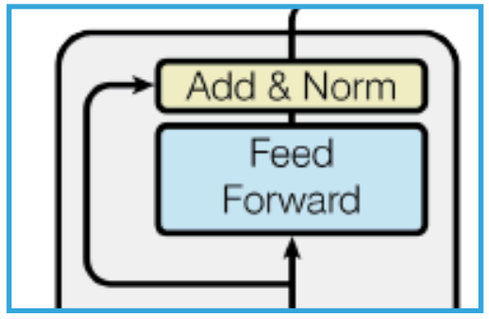
AND MORE…



- Attention 계층 이후 두 개의 Linear transformation과 ReLU 함수로 구성된 Feed-forward network를 통과시킨다.

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

AND MORE…

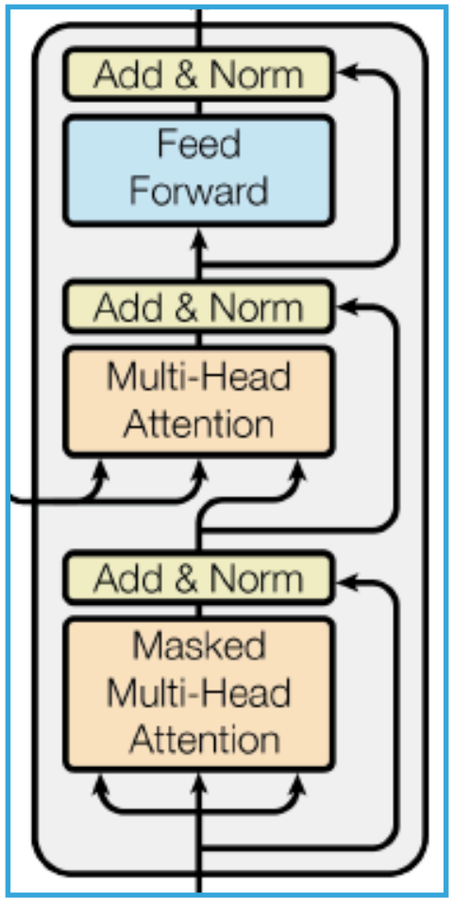


- Attention 계층 이후 두 개의 Linear transformation과 ReLU 함수로 구성된 Feed-forward network를 통과시킨다.

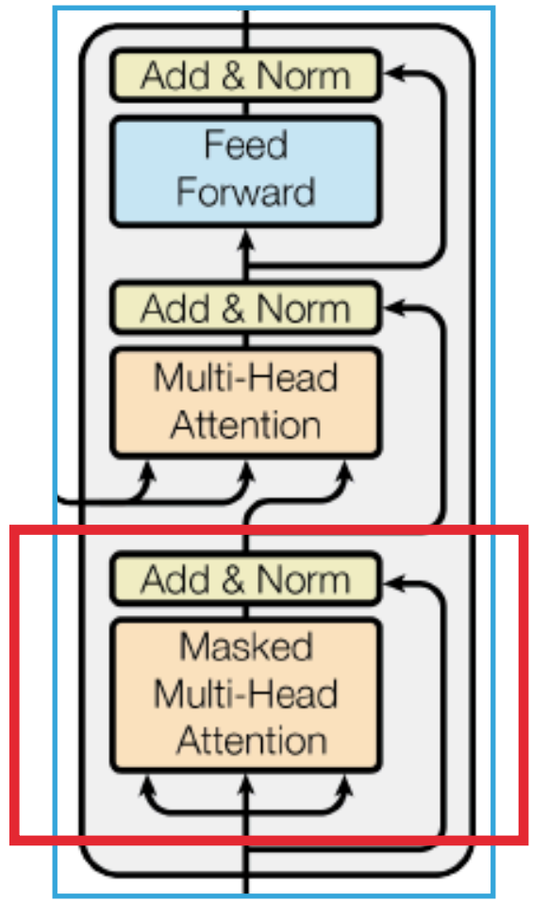
$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

- 또한 $x + sublayer(x)$ 형태의 residual connection 및 layer normalization 또한 적용

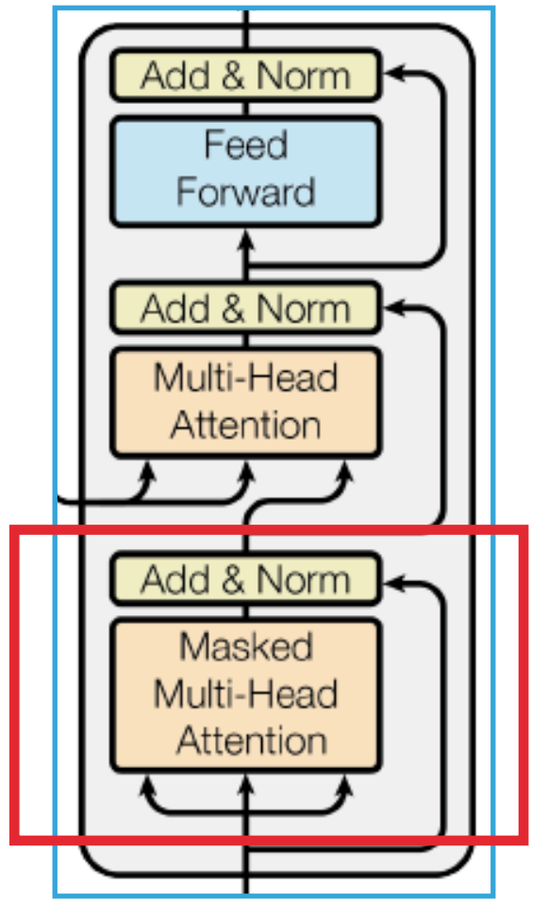
DECODER


Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention

DECODER

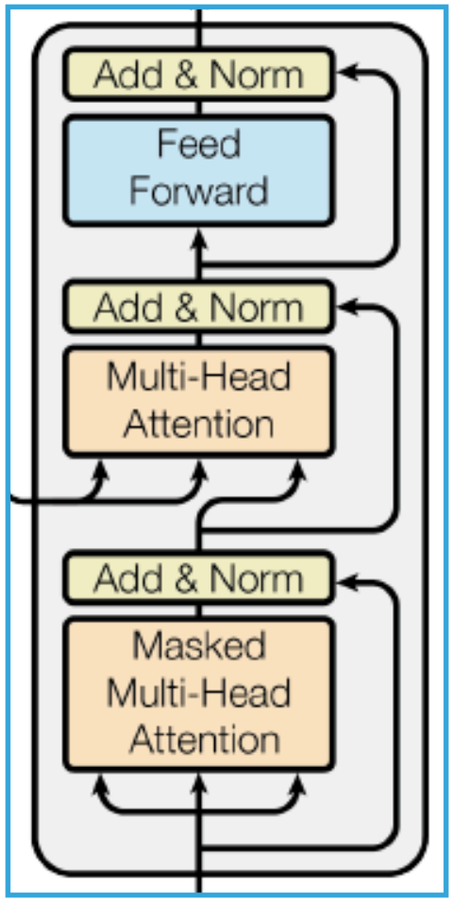

Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention

DECODER

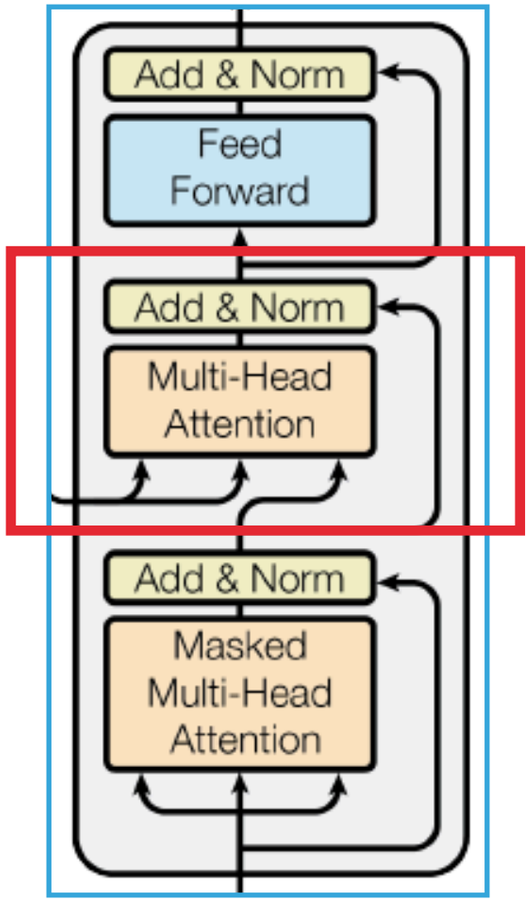


- 첫번째 Multi-Head Attention은 인코더와 동일하게 작동하나 i번째 단어를 처리할 때는 i 이전의 단어들만 참조한다(masked).
- Outputs 문장 내의 구조를 파악하는 역할 수행!

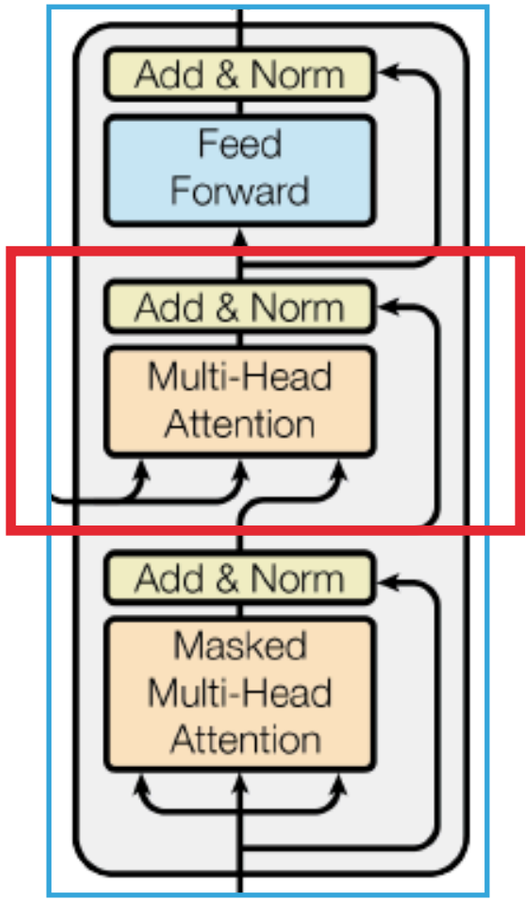
DECODER


Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention

DECODER

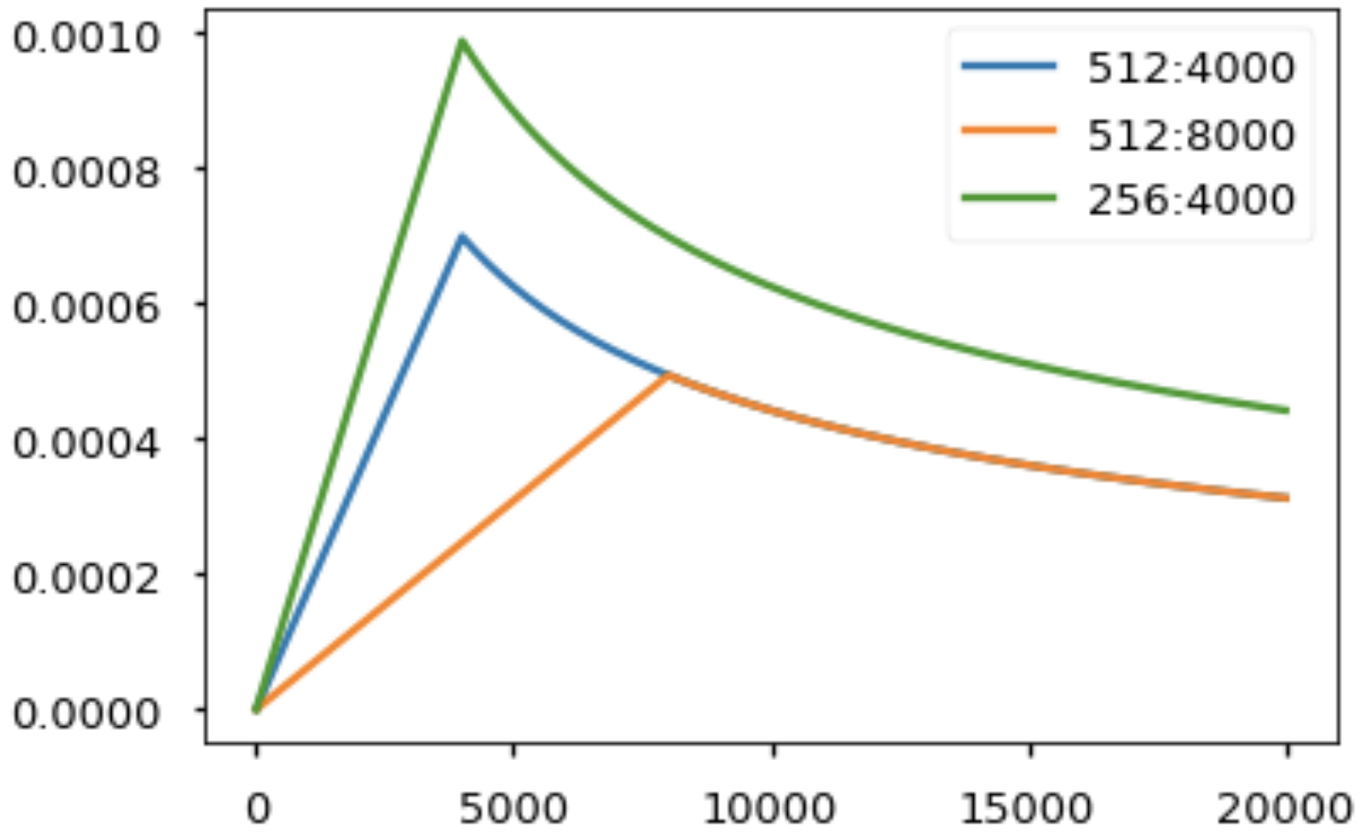

Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention

DECODER

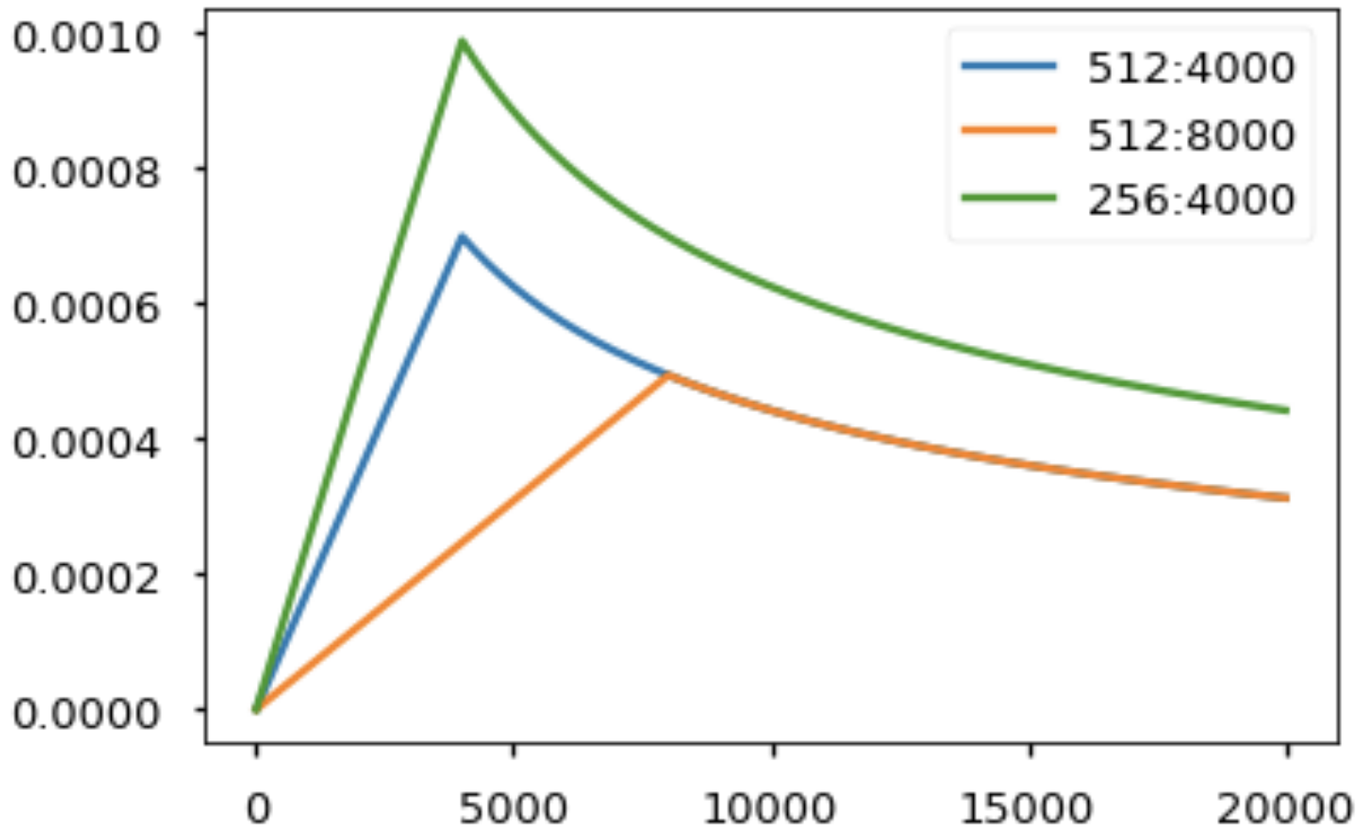


- 디코더의 두번째 Attention의 Query는 첫번째 Attention의 output, Key, Value는 인코더의 output을 사용한다.
- Outputs의 각 요소를 처리할 때 인코더 output중 어디를 참조할 지 결정!

LEARNING RATE

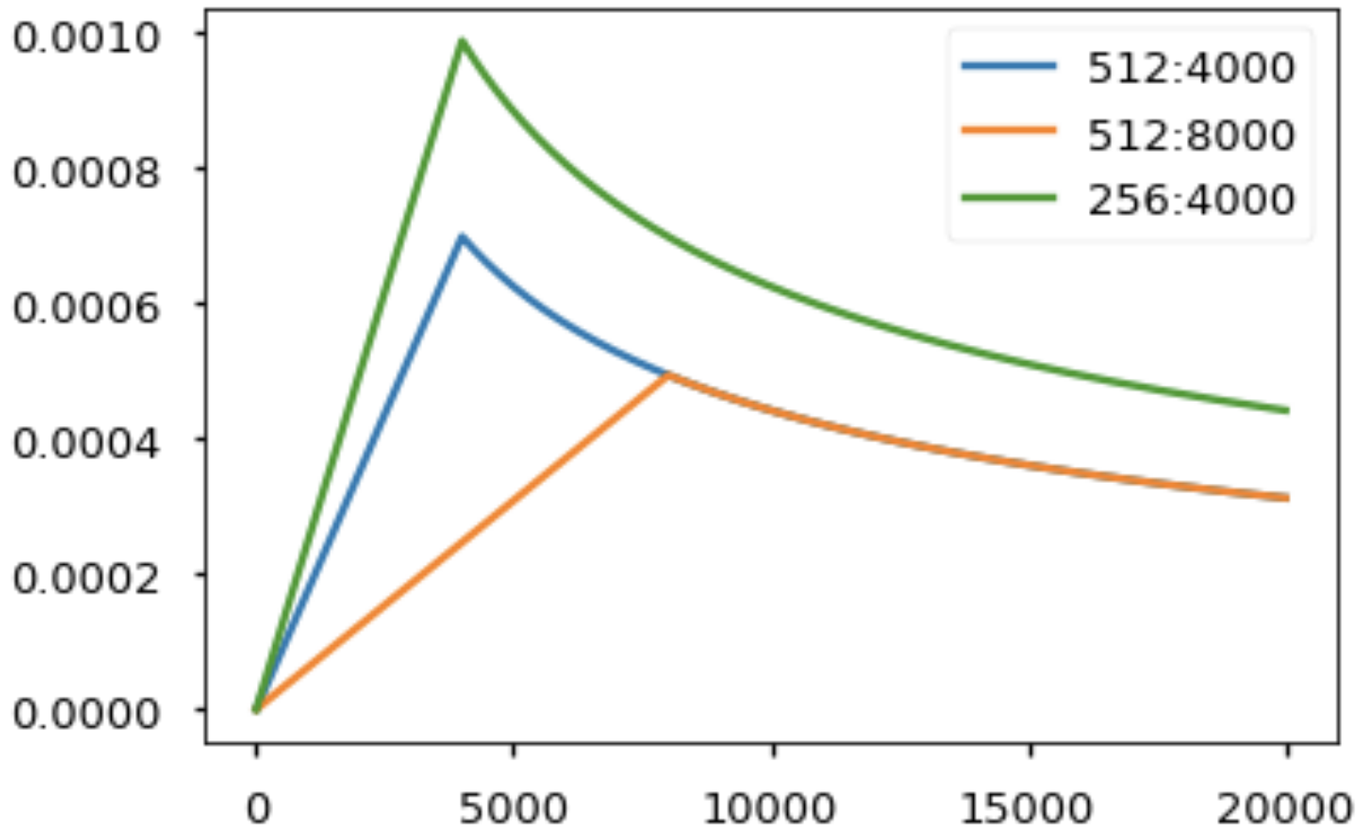


LEARNING RATE



$$lrate = d_{model}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warm\_steps^{-1.5})$$

LEARNING RATE



$$lrate = d_{model}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warm\_steps^{-1.5})$$

$(warm\_steps = 4000)$

SELF-ATTENTION 구조의 장점

SELF-ATTENTION 구조의 장점

1) 레이어당 전체 연산량이 줄어든다.

2) 병렬화가 가능한 연산이 늘어난다.

3) Long-term dependency를 잘 학습할 수 있다.

4) 모델의 결과를 해석하기 좋다.

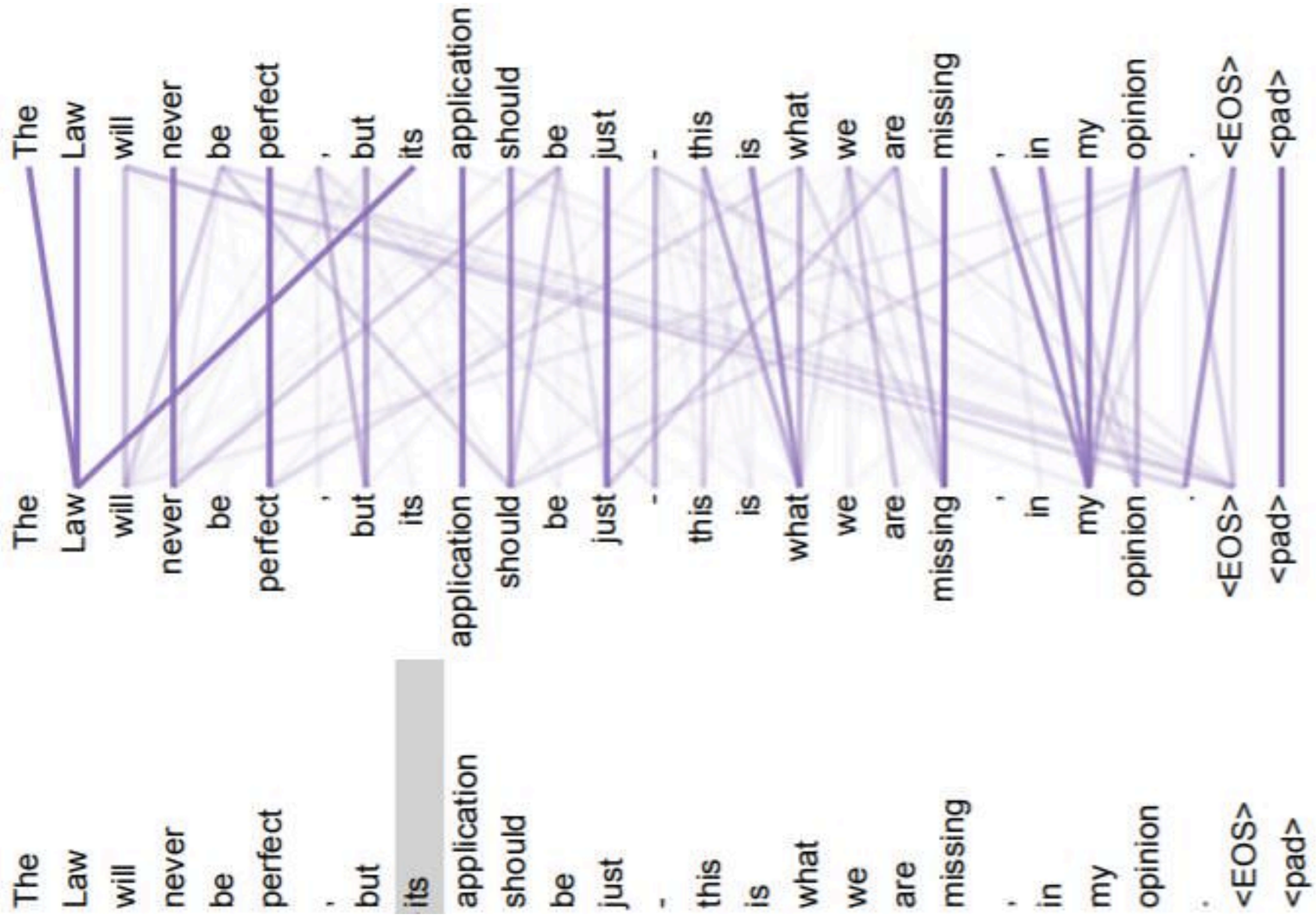
5) 무엇보다, 성능이 좋다!

COMPUTATIONAL COMPLEXITY

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

LONG-TERM DEPENDENCY & INTERPRETABILITY



RESULT

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [15]	23.75			
Deep-Att + PosUnk [32]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [31]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [8]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [26]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [32]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [31]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [8]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.0**	$2.3 \cdot 10^{19}$	

감사합니다!