



I. Attention

22.09.15 / 7기 전혜령

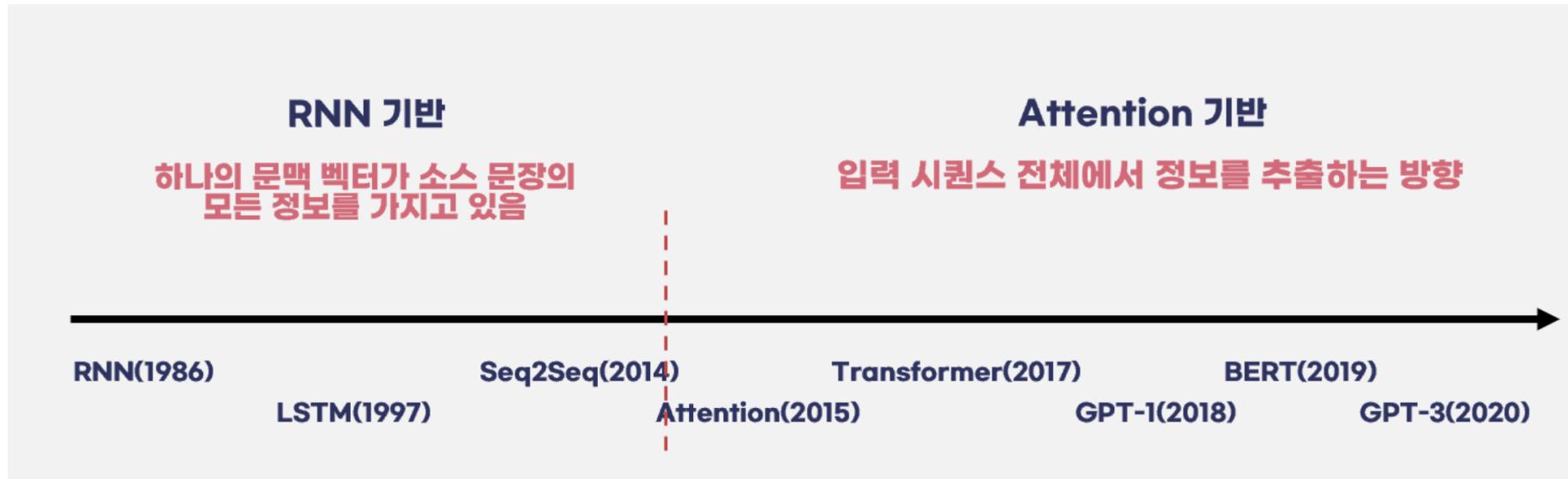
1. RNN with Attention

- Seq-to-Seq with RNN (개념 recap)
- Seq-to-Seq with attention
- Example: Image Captioning

2. Attention layers

- Attention layer
- Self-attention layer
- Variants of the Self-attention layer

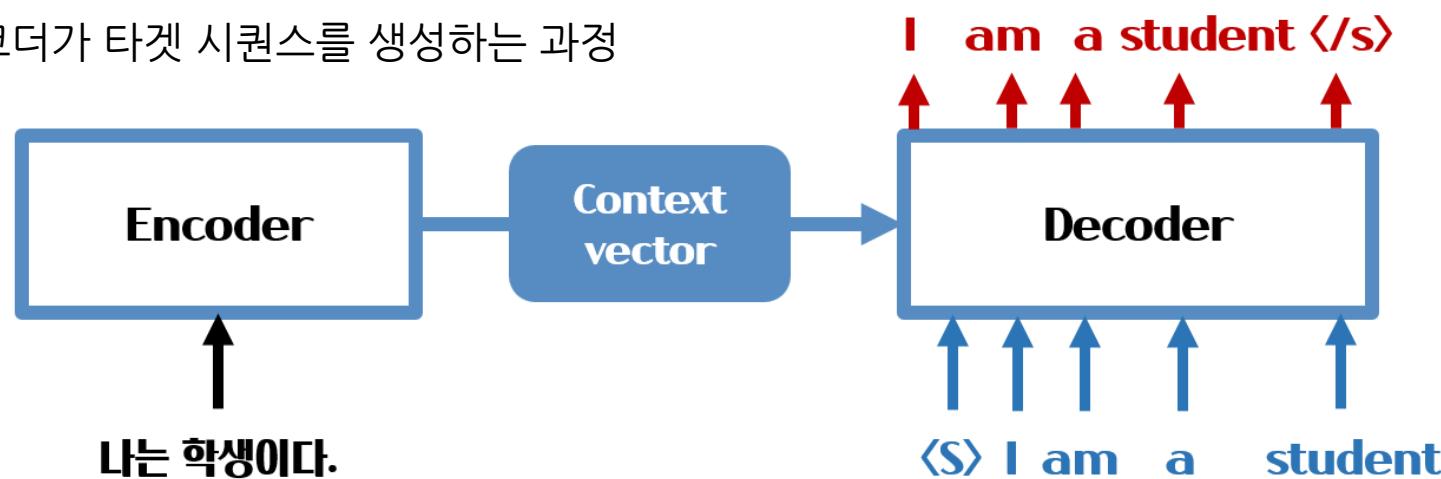
(0) Timeline



- RNN: NN에서 순환형 구조를 추가한 것
- LSTM: vanishing gradient 문제 해결 위해 hidden state에 cell state를 추가하여 발전시킨 것
- Seq2Seq: RNN의 구조를 사용해서 인코더, 디코더 구조로 변환해서 수행
- Attention: 모든 입력 시퀀스 참고
- Transformer: 오직 Attention만 사용

(I) Seq to Seq with RNN

- Seq 2 Seq: 문장을 입력으로 받아 문장을 출력하는 모델 / 기계번역에 주로 사용
 - 시퀀스: 단어 같은 무언가의 나열
- 인코더는 소스 시퀀스의 정보를 압축해 디코더로 보내주는 역할을 담당
 - 인코딩 = 인코더가 소스 시퀀스의 정보를 압축하는 과정
- 디코더는 인코더가 보내준 소스 시퀀스를 받아서 타겟 시퀀스를 생성
 - 디코딩 = 디코더가 타겟 시퀀스를 생성하는 과정

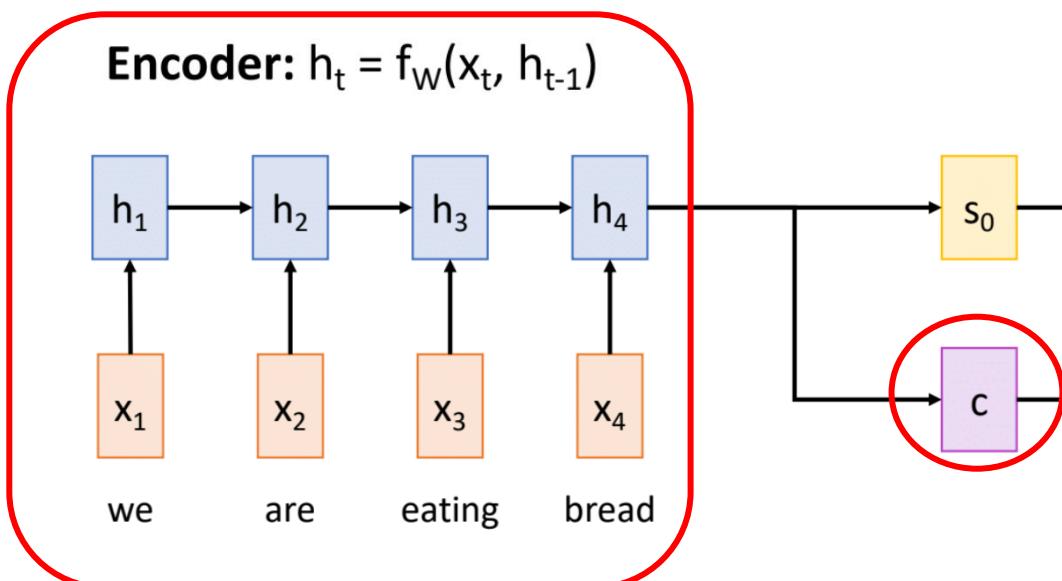


I. RNN with Attention

YONSEI Data Science Lab | DSL

(I) Seq to Seq with RNN

- 인코더는 입력 문장의 모든 단어들을 순차적으로 입력받은 뒤에 마지막에 이 모든 단어 정보들을 압축해서 하나의 벡터로 만들어 냄
- 이를 Context vector라고 함



- S_0 : 디코더 RNN의 initial value
- h_4 : 인코더 RNN의 최종 hidden state,
번역해야 할 문장에 대한 모든 정보 들어있음
- Context vector: h_4 Copy, 디코더에게 전달

Input: Sequence x_1, \dots, x_T

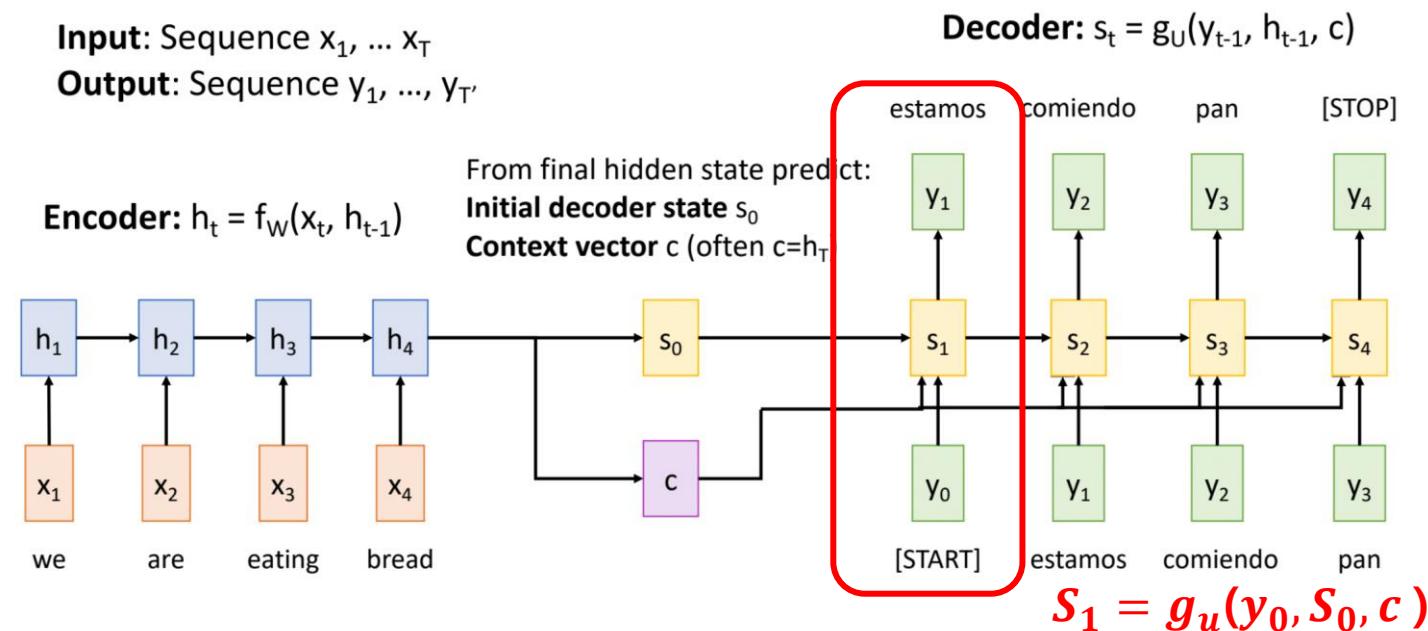
Output: Sequence $y_1, \dots, y_{T'}$

I. RNN with Attention

YONSEI Data Science Lab | DSL

(I) Seq to Seq with RNN

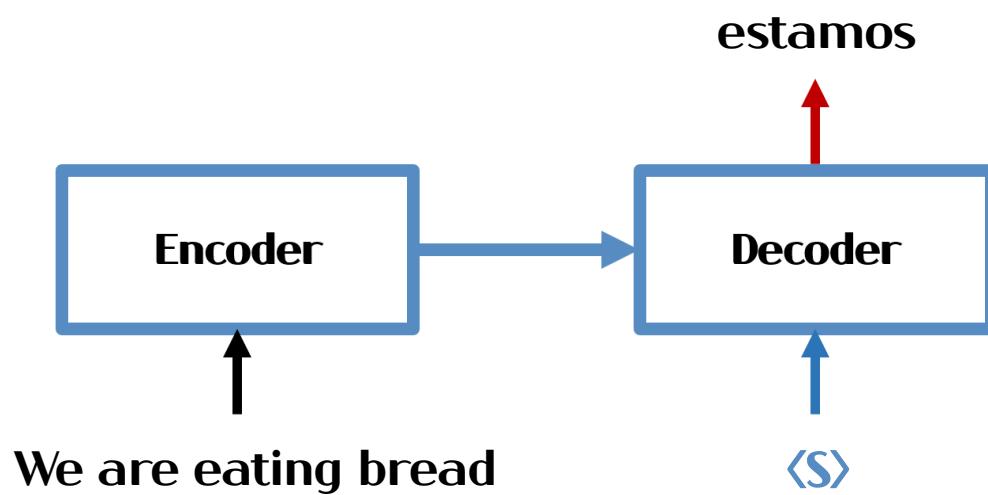
- 디코더는 인코더 정보를 이용해 타깃 문장을 생성해 냄
 - Train: 타깃 문장의 이번 시점 단어가 다음 시점의 단어를 예측하기 위한 입력으로 사용
 - Inference: 디코더의 이번 시점 예측 결과가 다음 시점의 입력으로 사용



I. The Transformer

YONSEI Data Science Lab | DSL

(I) Seq to Seq with RNN

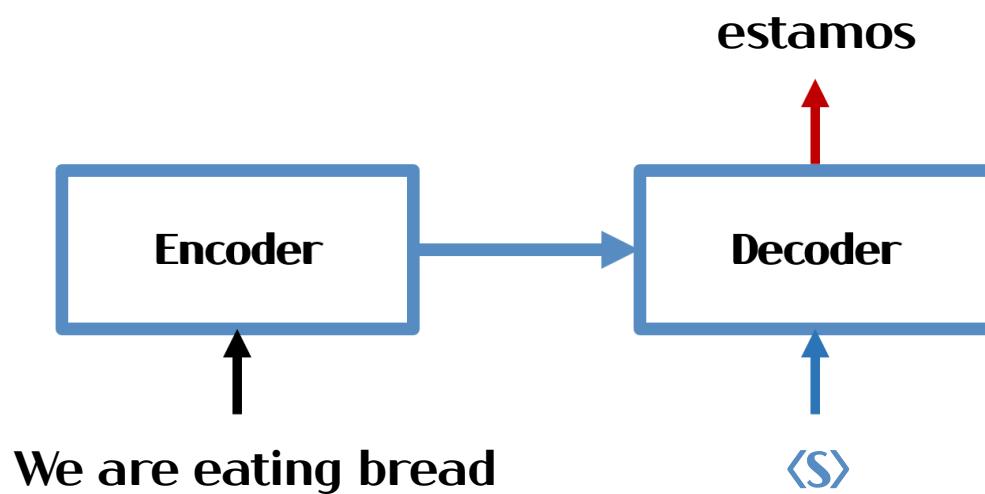


- 이번 학습은 그림처럼 *estamos*를 맞춰야 하는 차례
 - 이 때 인코더 입력은 소스 시퀀스 전체
 - 디코더 입력은 <S>로, 이는 타깃 시퀀스의 시작을 뜻하는 스페셜 토큰
 - 인코더는 소스 시퀀스를 압축해 디코더로 보내고, 디코더는 인코더에서 보내온 정보와 디코더 입력을 모두 감안해 다음 토큰 *estamos*를 맞춤
 - 최종 출력, 즉 디코더 출력은 타깃 언어의 어휘 수 만큼의 차원으로 구성된 벡터

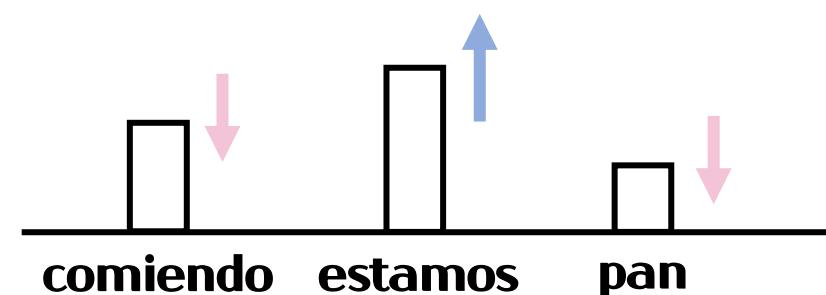
I. The Transformer

YONSEI Data Science Lab | DSL

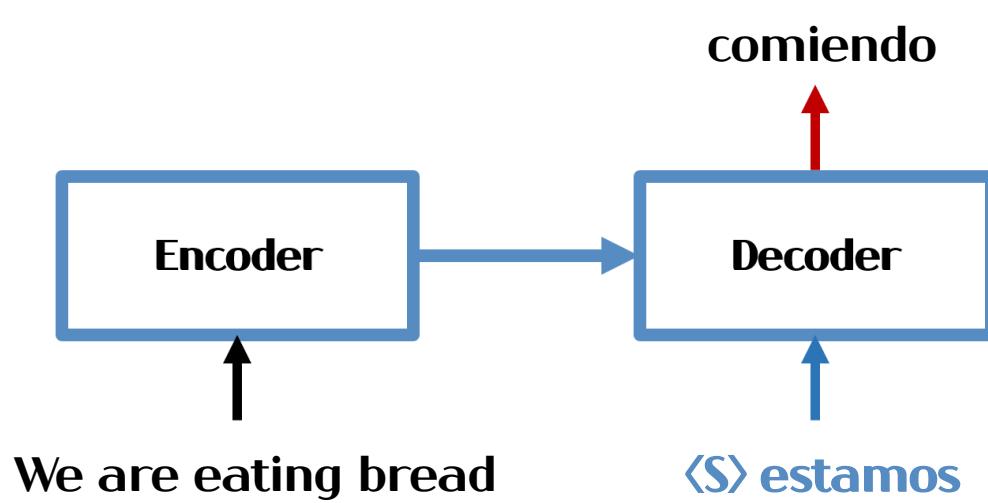
(I) Seq to Seq with RNN



- 이번 학습은 그림처럼 *estamos*를 맞춰야 하는 차례
 - 트랜스포머의 학습은 인코더와 디코더 입력이 주어졌을 때, 정답에 해당하는 단어의 확률값을 높이는 방식으로 수행됨
 - *estamos*에 해당하는 확률 높이고 나머지 단어의 확률은 낮아져야 함



(3) Model training and Inference



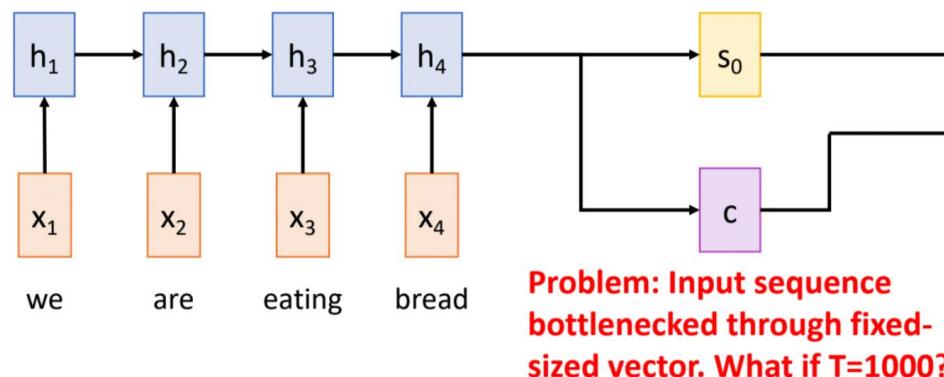
- 이제 comiendo를 맞출 차례

- 그러나 인퍼런스 때는 현재 디코더 입력에 직전 디코딩 결과를 사용
 - 학습이 잘못되어 comiendo 대신 dormir라는 단어가 예측되었다면 입력은 <s> dormir가 됨
- 학습 과정 중 모델은 이번 시점의 정답인 comiendo에 해당하는 확률을 높이고 나머지 단어의 확률은 낮아지도록 모델 갱신

(I) Seq to Seq with RNN

- 여기서 문제점!

- 입력이 정말 긴 문장이라면?
→ 고정된 크기의 context vector가 앞 부분을 잘 반영하지 못하지 않을까?
- 입력이 짧은 문장이라면? → 잘 context를 추출했을지 의심스러워



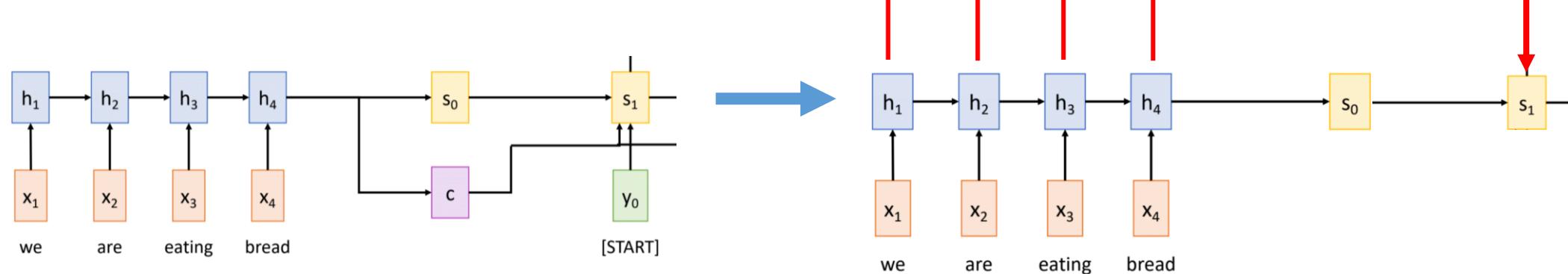
- h_1 이 굳이 다른 단어들을 다 거쳐서 전달되는 것이 불만, 각자 state에 direct하게 연결될 수는 없을까?
→ 새 context vector을 디코더 매 스텝마다 사용해보자! = Attention

I. RNN with Attention

YONSEI Data Science Lab | DSL

(2) Seq to Seq with Attention

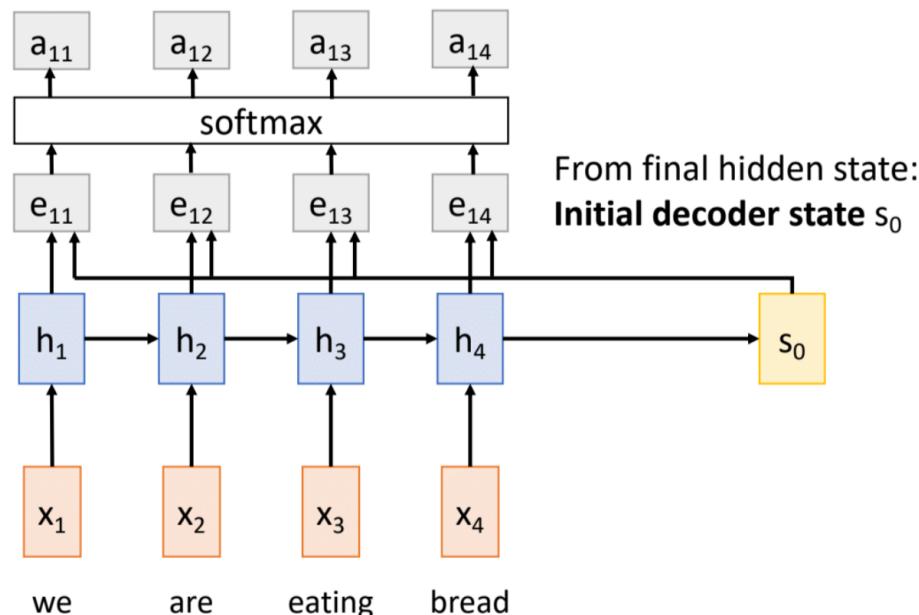
- 새로운 state를 만들 때 정보를 바로 꽂아주기



- Seq2Seq with RNN
- Seq2Seq with RNN and Attentions

(2) Seq to Seq with Attention

- 순서대로 살펴보자
- 먼저 인코더의 시점 별 인코딩과 디코더의 hidden state의 유사도를 계산하여 attention score와 attention weight(probability)를 계산한다.



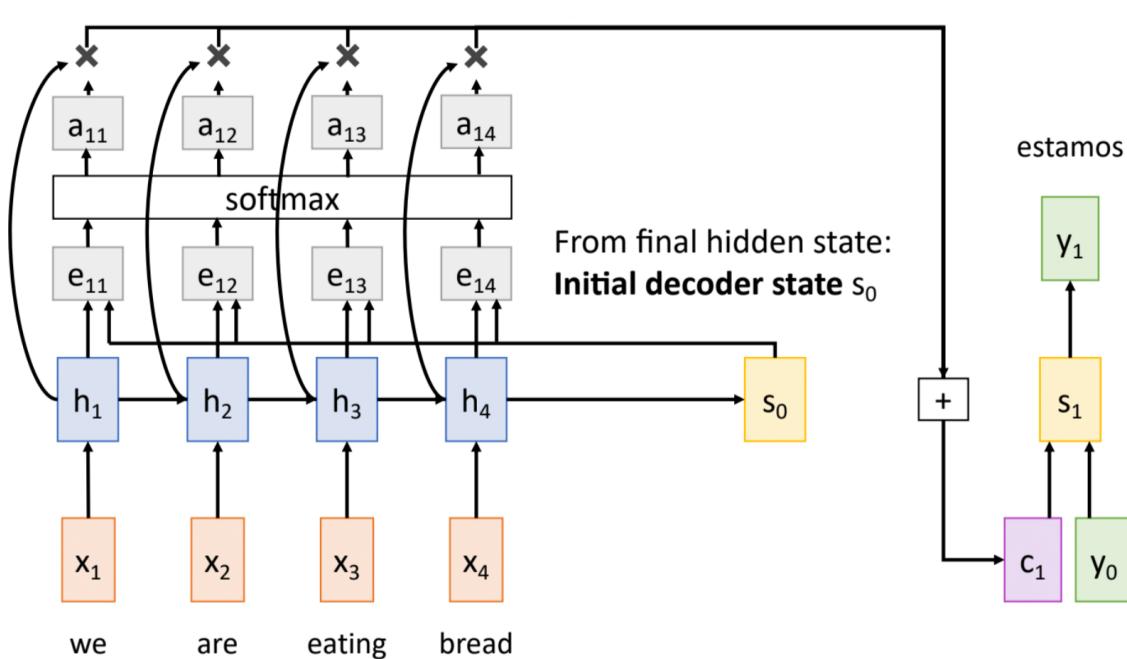
- **Attention score:** $= [s_t^T h_1, \dots, s_t^T h_N] \in \mathbb{R}^N$
- **Attention weight:** $= \text{softmax}(e^t) \in \mathbb{R}^N$

I. RNN with Attention

YONSEI Data Science Lab | DSL

(2) Seq to Seq with Attention

- Attention distribution을 사용해 인코더 hidden state를 가중합하여 attention output을 계산한다.
- 그 뒤 Attention output과 디코더 hidden state를 concat하여 해당 시점의 단어를 생성한다.



- **Attention output:** $y_t = \sum_{i=1}^N \alpha_i^t h_i \in \mathbb{R}^h$
 - **Concat with attention:** $[a_t; s_t] \in \mathbb{R}^{2h}$
- $$c_1 = a_{11}h_1 + a_{12}h_2 + a_{13}h_3 + a_{14}h_4$$
- 만약
 $a_{11} = a_{12} = 0.45$
 $a_{13} = a_{14} = 0.05$ 라면
→ 그럴듯한 linear combination

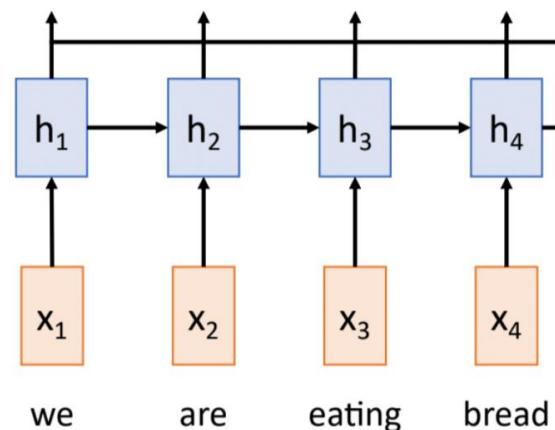
I. RNN with Attention

YONSEI Data Science Lab | DSL

(2) Seq to Seq with Attention

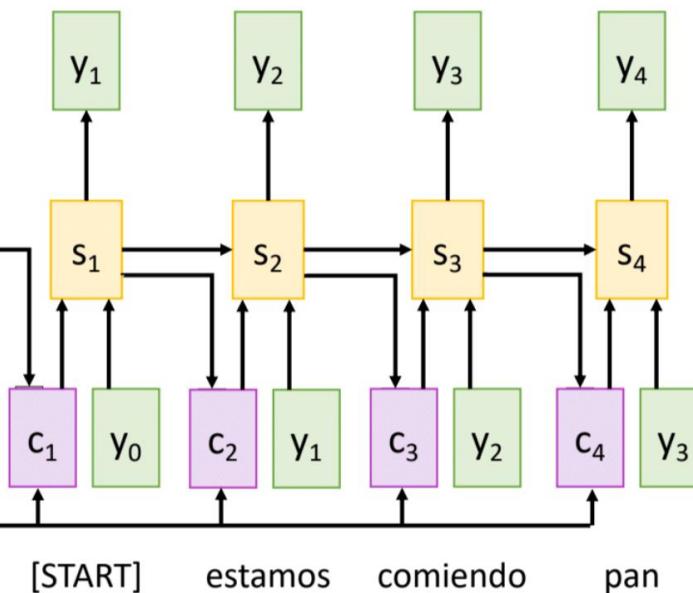
- 디코더에서 각각의 c 가 입력의 서로 다른 파트를 하이라이트 해준다고 생각 가능

- $c_1 = s_1$ 만을 위한 context vector
- $c_2 = s_2$ 만을 위한 context vector
- $c_3 = s_3$ 만을 위한 context vector
- $c_4 = s_4$ 만을 위한 context vector



$$S_2 = g_u(y_1, s_1, c) \rightarrow S_2 = g_u(y_1, s_1, c_2)$$

estamos comiendo pan [STOP]



I. RNN with Attention

YONSEI Data Science Lab | DSL

(2) Seq to Seq with Attention

- attention matrix를 시각화한 heatmap
- 열을 소스, 행을 타겟이라고 보면 됨
- 색깔이 밝을수록 attention score가 높은 것

Input: “The agreement on the European Economic Area **was signed** in August 1992.”

Output: “L'accord sur la zone économique européenne **a été signé** en août 1992.”

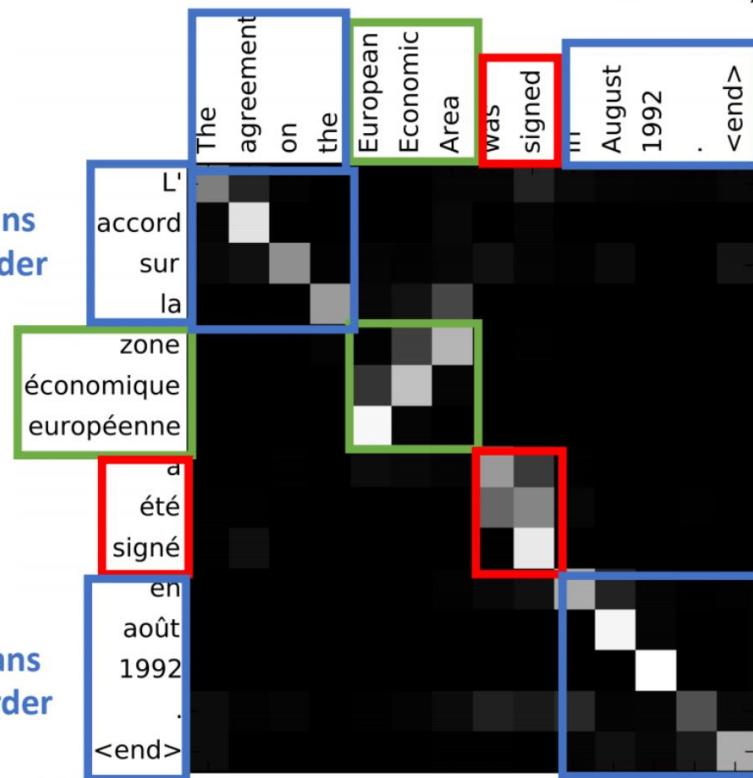
Diagonal attention means words correspond in order

Attention figures out different word orders

Verb conjugation

Diagonal attention means words correspond in order

Visualize attention weights $a_{t,i}$

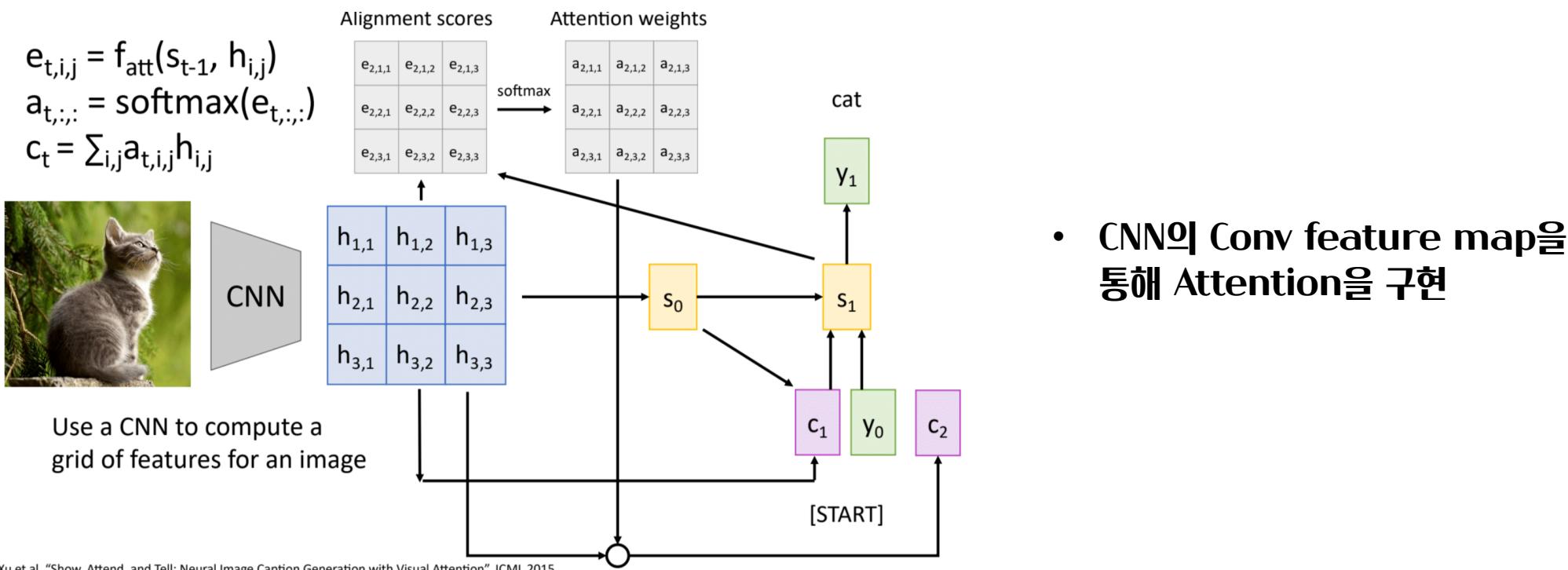


I. RNN with Attention

YONSEI Data Science Lab | DSL

(3) Example: Image captioning

- 각각의 context vector을 만들 때 hidden state의 순서 구조에 의존하지 않았음
- 굳이 시퀀스 데이터가 아니어도 적용 가능할 듯? → 이미지에 적용해보자



I. RNN with Attention

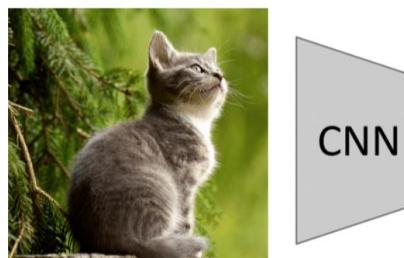
YONSEI Data Science Lab | DSL

(3) Example: Image captioning

- 디코더가 매번 다른 context 벡터를 사용함으로써 매번 이미지의 다른 부분에 집중하도록 함

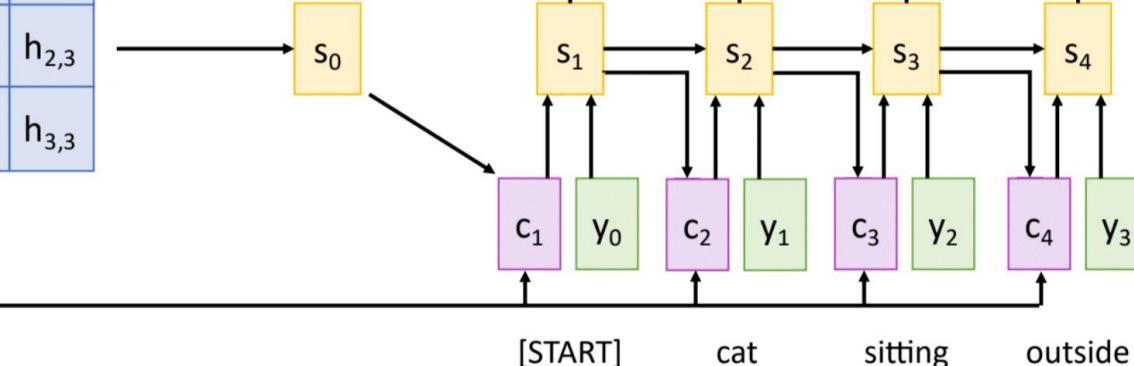
$$\begin{aligned} e_{t,i,j} &= f_{\text{att}}(s_{t-1}, h_{i,j}) \\ a_{t,:,:} &= \text{softmax}(e_{t,:,:}) \\ c_t &= \sum_{i,j} a_{t,i,j} h_{i,j} \end{aligned}$$

Each timestep of decoder
uses a different context
vector that looks at different
parts of the input image



Use a CNN to compute a
grid of features for an image

$h_{1,1}$	$h_{1,2}$	$h_{1,3}$
$h_{2,1}$	$h_{2,2}$	$h_{2,3}$
$h_{3,1}$	$h_{3,2}$	$h_{3,3}$

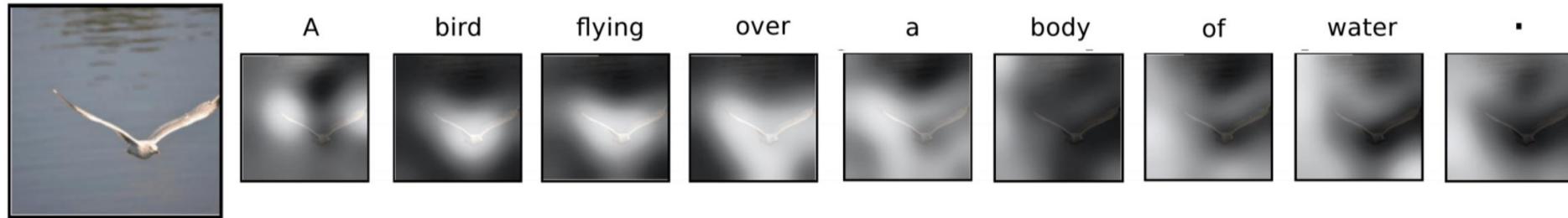


1. RNN with Attention

YONSEI Data Science Lab | DSL

(3) Example: Image captioning

- 디코더가 매번 다른 context 벡터를 사용함으로써 매번 이미지의 다른 부분에 집중하도록 함
- 인간의 시지각도 실제로 이런 식으로 작동



A dog is standing on a hardwood floor.



A stop sign is on a road with a mountain in the background.



A group of people sitting on a boat in the water.



A giraffe standing in a forest with trees in the background.

(I) Attention layer

- NN의 새로운 layer로 attention 일반화
 - Attention을 써먹으려는 시도 되게 많았음
 - 이후 발전해 여기저기 다양한 architecture와 task에서 사용

“Show, attend, and tell” (Xu et al, ICML 2015)

Look at image, attend to image regions, produce question

“Ask, attend, and answer” (Xu and Saenko, ECCV 2016)

“Show, ask, attend, and answer” (Kazemi and Elqursh, 2017)

Read text of question, attend to image regions, produce answer

“Listen, attend, and spell” (Chan et al, ICASSP 2016)

Process raw audio, attend to audio regions while producing text

“Listen, attend, and walk” (Mei et al, AAAI 2016)

Process text, attend to text regions, output navigation commands

“Show, attend, and interact” (Qureshi et al, ICRA 2017)

Process image, attend to image regions, output robot control commands

“Show, attend, and read” (Li et al, AAAI 2019)

Process image, attend to image regions, output text

2. Attention layers

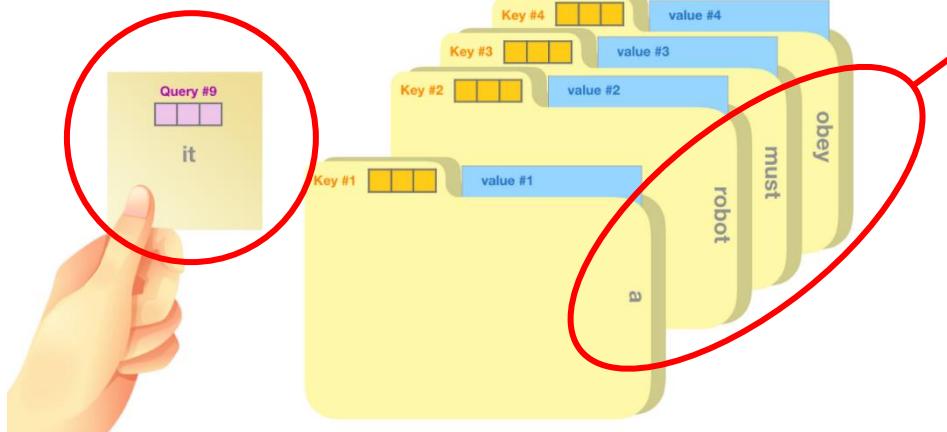
YONSEI Data Science Lab | DSL

(I) Attention layer

- 새로 등장하는 개념
- 쿼리, 키, 밸류 (Query, Key, Value)

- 쿼리: 디코더의 이전 레이어 hidden state , 영향을 받는 디코더의 토큰
- 키: 인코더의 output state , 영향을 주는 인코더의 토큰들
- 밸류: 인코더의 output state , 그 영향에 대한 가중치가 곱해질 인코더 토큰들

디코더 특정 시점 hidden state 가 쿼리로 주어짐



Query	Key	Value	Output	주제
손승진	손승진	추천시스템	$1*int(V_1)$	추천시스템
최명현	최명현	컴퓨터비전	$0*int(V_2)$	
김한빈	김한빈	자연어처리	$0*int(V_3)$	

a, robot, must, obey 는 각각 키

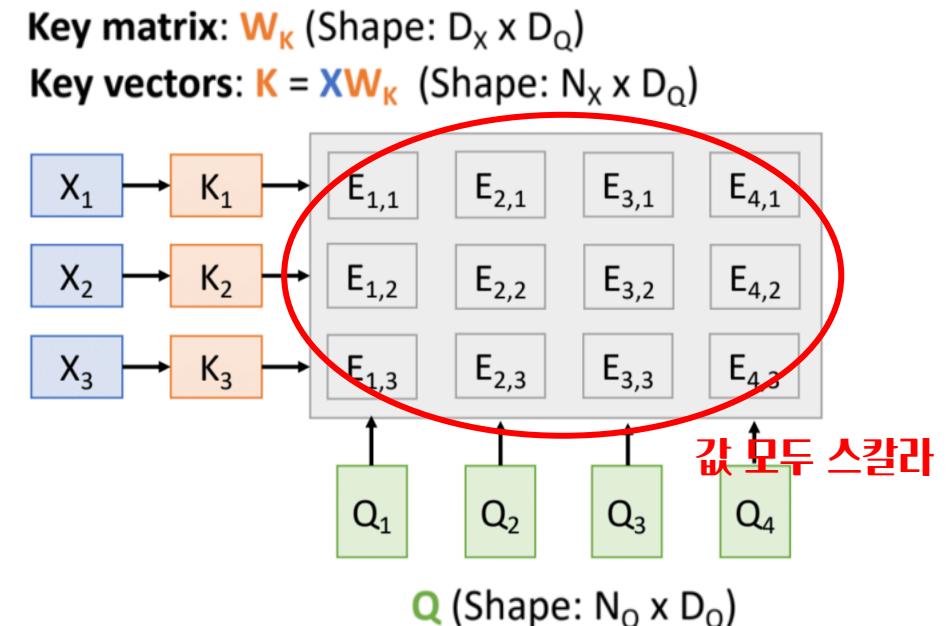
- 위 그림에서 Query #9 벡터는 단어“it”에 연관된 벡터
- 컴퓨터는 모든 단어의 키를 살펴보며 쿼리와 가장 잘 맞는 것을 찾음
(= 쿼리와 키의 유사도를 구함 → 어떤 키가 가장 유사한가?)
- 그리고 해당하는 키에 걸려있는 실제 밸류를 가져가게 됨
(밸류는 키와 연결된 실제 토큰, 쿼리와 키의 유사도가 곱해질 값)

2. Attention layers

YONSEI Data Science Lab | DSL

(I) Attention layer

- 키, 쿼리, 밸류가 어떻게 만들어지는지 순서대로 살펴보자
- 이전 기계번역 예시와 비교했을 때, X는 인코더의 hidden state, Q는 디코더의 hidden state라고 생각하면 됨
 - 쿼리(디코더의 hidden state 가져오기) : Q
 - 키: $K = XW_k$
 - 밸류: $V = XW_v$
 - Attention score (E): $E_{i,j} = Q_i \cdot K_j / \sqrt{D_k}$
 - Attention weight (A): $A = \text{softmax}(E, \text{dim} = 1)$
 - Output vector: $Y = AV$



2. Attention layers

YONSEI Data Science Lab | DSL

(I) Attention layer

- 키, 쿼리, 밸류가 어떻게 만들어지는지 순서대로 살펴보자
- 이전 기계번역 예시와 비교했을 때, X는 인코더의 hidden state, Q는 디코더의 hidden state라고 생각하면 됨
 - 쿼리(디코더의 hidden state 가져오기) : Q
 - 키: $K = XW_k$
 - 밸류: $V = XW_v$
 - Attention score (E): $E_{i,j} = Q_i \cdot K_j / \sqrt{D_k}$
 - Attention weight (A): $A = \text{softmax}(E, \dim = 1)$
 - Output vector: $Y = AV$

$\sqrt{D_k}$ 로 나눠주는 이유?
→ Scaling 해주기 위해서!

2. Attention layers

YONSEI Data Science Lab | DSL

(I) Attention layer

▪ (참고) Why scaling?

- D_k 가 커지면 \mathbf{QK}^T 값이 커진다.

$$QK^T = \sum_{i=1}^{d_k} q_i k_i \xrightarrow{\text{q, k가 서로 독립이라고 가정}} \quad \quad \quad$$

$$\begin{aligned} E(q_i k_i) &= E(q_i) \cdot E(k_i) \\ \text{Var}(q_i k_i) &= (\sigma_{q_i}^2 + \mu_{q_i}^2) (\sigma_{k_i}^2 + \mu_{k_i}^2) - \mu_{q_i}^2 \mu_{k_i}^2 \end{aligned}$$

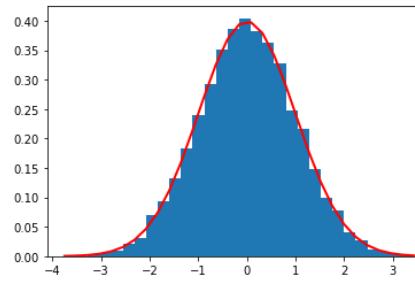
q, k가 가우시안 분포를 따른다고
가정했을 때 기댓값과 분산

$$E(q_i k_i) = 0 \quad \text{Var}(q_i k_i) = 1$$

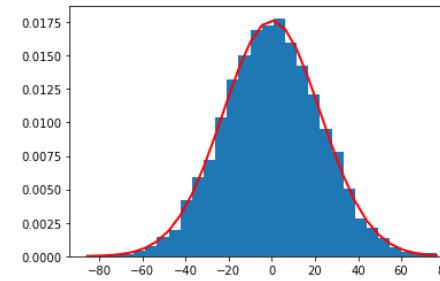
$$E(QK^T) = E\left(\sum_{i=1}^{d_k} q_i k_i\right) = \sum_{i=1}^{d_k} E(q_i k_i) = 0$$

$$\text{Var}(QK^T) = \text{Var}\left(\sum_{i=1}^{d_k} q_i k_i\right) = \sum_{i=1}^{d_k} \text{Var}(q_i k_i) = d_k$$

수식을 통해 d_k 가 커짐에 따라 분산이 커진다는 것을 확인,
분산이 크다는 것은 곧 \mathbf{QK}^T 가 큰 값을 가질 확률이 커졌다는 것
(실제로 분산이 각각 1일 때, 512 일 때 분포는 옆 그림과 같다)



Mu = 0, sigma = 1



Mu = 0, sigma = sqrt(512)

(I) Attention layer

▪ (참고) Why scaling?

- D_k 가 커지면 \mathbf{QK}^T 값이 커진다.
- \mathbf{QK}^T 값이 커지면 소프트맥스 함수의 gradient vanishing이 발생한다.

소프트맥스 함수는 scale variant한 특성을 갖고 있음

[1, 2, 3, 4, 1, 2, 3]을 입력하면 소프트맥스값은 [0.024, 0.064, 0.175, 0.475, 0.024, 0.064, 0.175]

→ '4'가 대부분의 가중치를 갖게 됨, 나머지 값은 0에 가까워짐

[0.1, 0.2, 0.3, 0.4, 0.1, 0.2, 0.3](합계 1.6)을 입력하면 소프트맥스값은 [0.125, 0.138, 0.153, 0.169, 0.125, 0.138, 0.153]

→ 0과 1 softmax 사이의 값에 대해 실제로 최대값을 덜 강조한다는 것을 보여줌

(0.169는 0.475보다 작을 뿐만 아니라 초기 비율인 $0.4/1.6=0.25$ 보다 작은)

- D_k 가 커짐에 따라 \mathbf{QK}^T 의 분산이 커진다.
- 분산이 커짐에 따라 \mathbf{QK}^T 의 성분 사이 편차도 커진다.
- 성분 사이 큰 편차는 scale variant한 소프트맥스 함수와 만나 gradient vanishing을 일으키는 형태가 된다!

(I) Attention layer

- (참고) Why scaling?

- D_k 가 커지면 \mathbf{QK}^T 값이 커진다.
- \mathbf{QK}^T 값이 커지면 소프트맥스 함수의 gradient vanishing이 발생한다.
- 결국 scaling을 통해서 gradient vanishing을 줄일 수 있고 scaling은 $\sqrt{D_k}$ 값으로 한다.

$$\text{Var}(QK^T) = \text{Var}\left(\sum_{i=1}^{d_i} q_i k_i\right) = \sum_{i=1}^{d_i} \text{Var}(q_i k_i) = d_k \quad \longleftarrow \quad \sqrt{D_k} \text{로 나눠주면 분산을 다시 } 1 \text{로 줄일 수 있게 되며,}
이것이 스케일링 값으로 } \sqrt{D_k} \text{ 을 사용한 근거가 됨$$

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

결론: $\sqrt{D_k}$ 로 나눠주는 이유?
→ Scaling 통해 gradient vanishing
을 줄이기 위해서

2. Attention layers

YONSEI Data Science Lab | DSL

(I) Attention layer

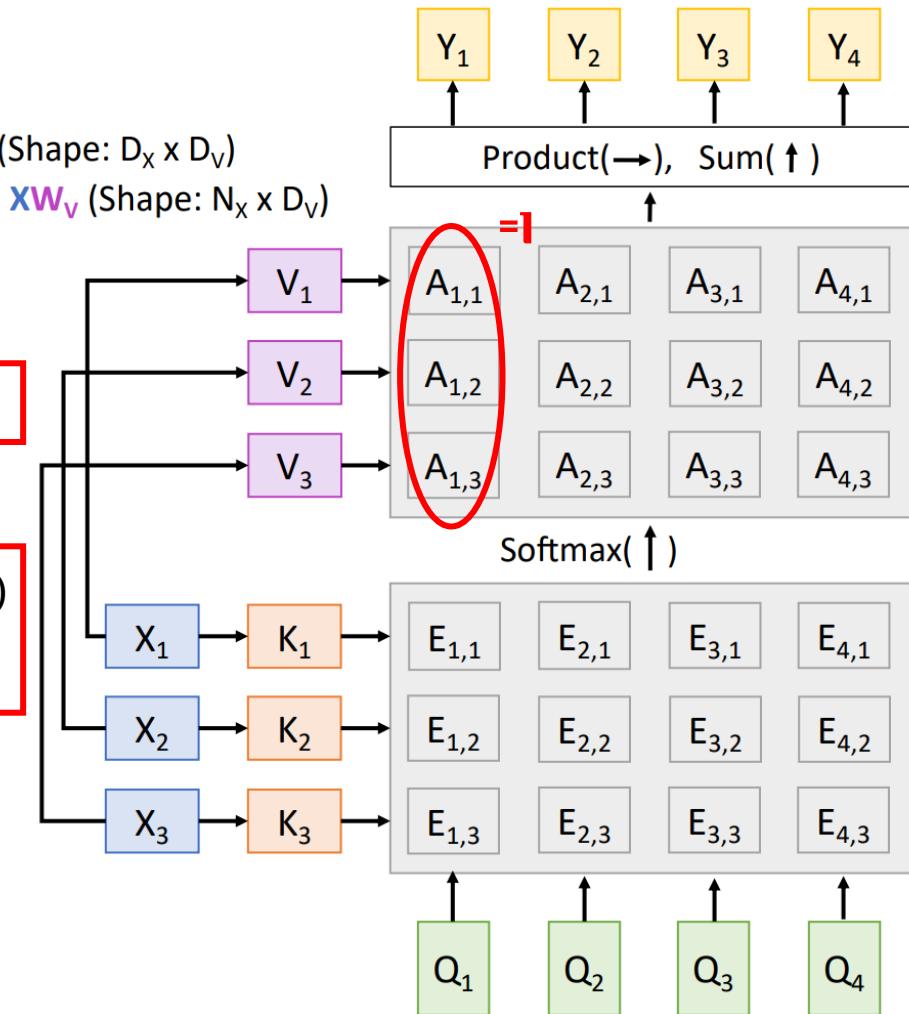
- 다음으로,
 - 쿼리: Q
 - 키: $K = XW_k$ 키랑 밸류는 **learnable weight**
 - 밸류: $V = XW_v$
 - Attention score (E): $E_{i,j} = Q_i \cdot K_j / \sqrt{D_Q}$
 - Attention weight (A): $A = \text{softmax}(E, \text{dim} = 1)$
 - Output vector: $Y = AV$

$$Y_1 = A_{11}V_1 + A_{12}V_2 + A_{13}V_3$$

최종 출력 = 밸류 벡터들의 가중합

Value matrix: W_v (Shape: $D_x \times D_v$)

Value Vectors: $V = XW_v$ (Shape: $N_x \times D_v$)



(2) Self - Attention layer

그 동물은 길을 건너지 않았다. 왜냐하면 그것은 너무 피곤했기 때문이다.

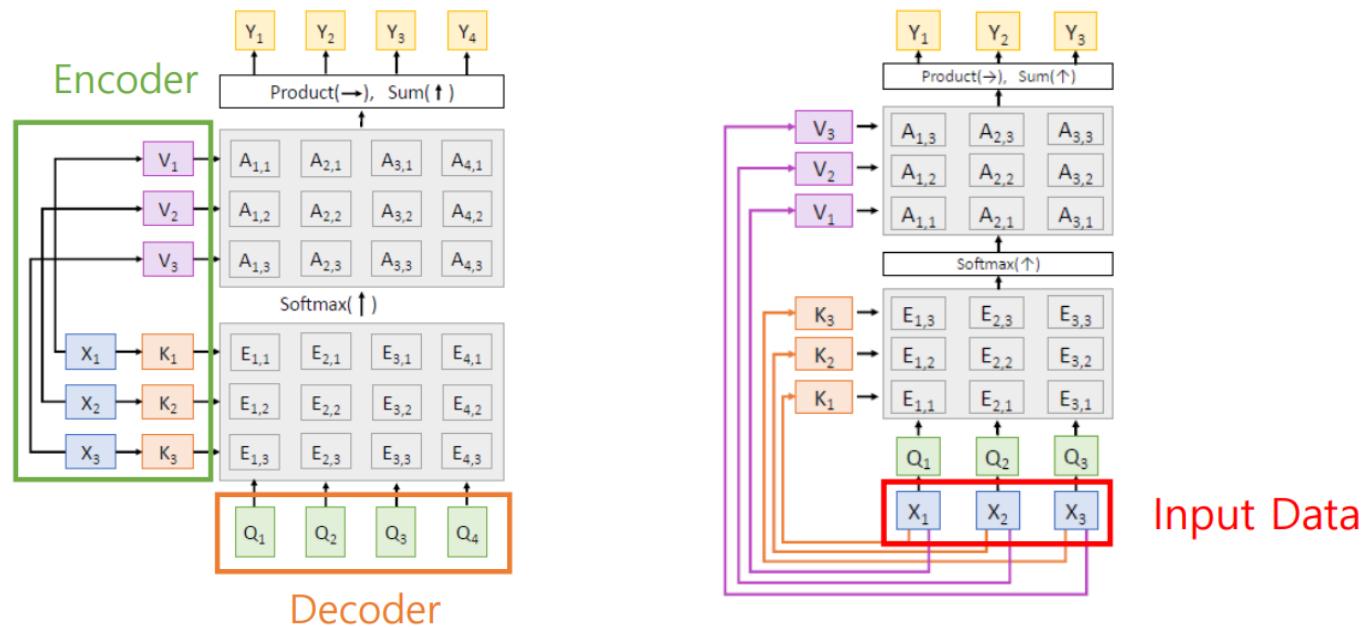
- "그것"이 "그 동물"을 가리키는 것인지 "길"을 가리키는 것인지 알아내기가 컴퓨터에겐 쉽지 않다
- 그것과 동물을 연결하는 일이 바로 Self-Attention의 임무!
- **Self-Attention?**
 - 입력 문장 내의 다른 위치에 있는 단어들을 보고 힌트를 얻어 현재 단어를 더 잘 인코딩하는 것
- **How?**
 - "그것"이란 단어를 인코딩할 때, 입력의 여러 단어들 중에서 "그 동물"이란 단어에 집중하고 이 단어의 의미 중 일부를 "그것"이라는 단어를 인코딩할 때 사용

2. Attention layers

YONSEI Data Science Lab | DSL

(2) Self - Attention layer

- Attention vs Self-Attention
- Attention (Decoder → Query / Encoder → Key, Value) / encoder, decoder 사이의 상관관계를 바탕으로 특징 추출
- Self attention (입력 데이터 → Query, Key, Value) / 데이터 내의 상관관계를 바탕으로 특징 추출



2. Attention layers

YONSEI Data Science Lab | DSL

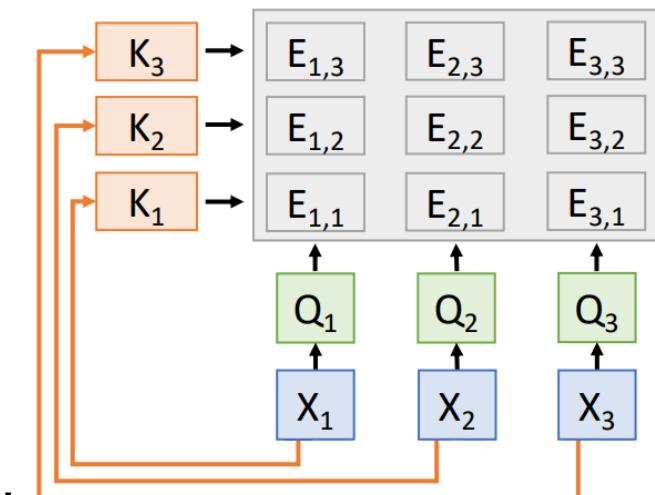
(2) Self - Attention layer

- 셀프 어텐션에서는 키, 쿼리, 밸류가 어떻게 만들어지는지 순서대로 살펴보자
- 이번에는 쿼리도 만들어 주어야 함 → 자기 자신(X)을 이용해서! (이전에는 디코더 state 이용)
 - 다음 수식처럼 입력 벡터 시퀀스(X)에 쿼리, 키를 만들어주는 행렬(W)을 각각 곱하기
 - $Q = X \times W_Q$, $K = X \times W_K$, ($V = X \times W_V$)

Input **Thinking**

Embedding X_1 \times $W_Q = q_1$

X_1 \times $W_K = k_1$



- Attention score (E): $E_{i,j} = Q_i \cdot K_j / \sqrt{D_k}$ → Q, K 구해서 E 계산

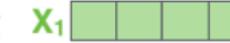
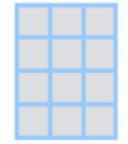
2. Attention layers

YONSEI Data Science Lab | DSL

(2) Self - Attention layer

- 다음으로,
 - 다음 수식처럼 밸류를 만들어주는 행렬(W)을 곱하기
 - $V = X \times Wv$

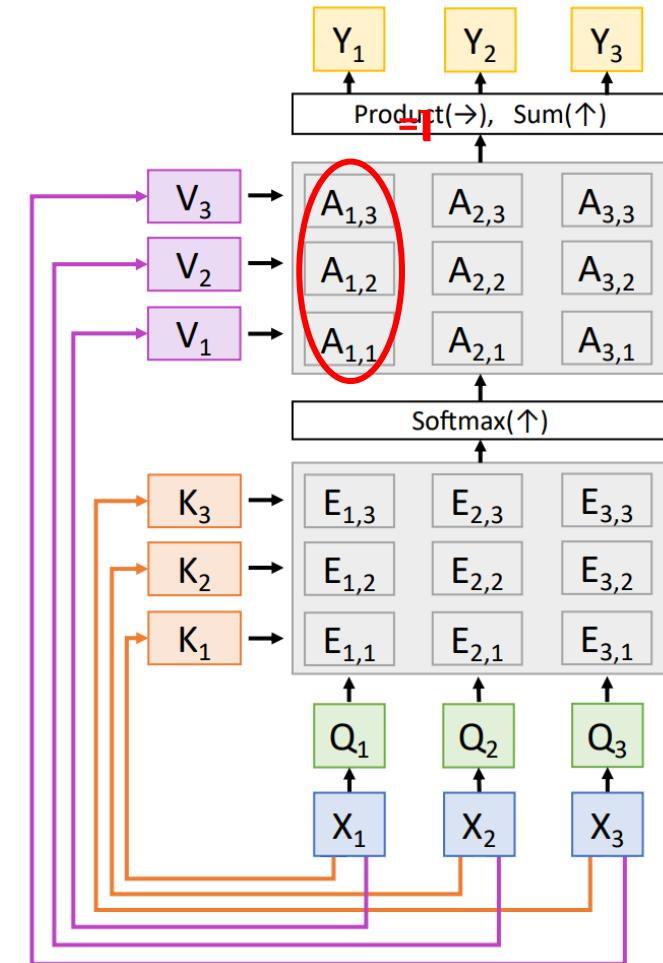
Input **Thinking**

Embedding X_1  \times  $=$ v_1 

- Attention weight (A): $A = softmax(E, dim = 1)$
- Output vector: $Y = AV$

$$Y_1 = A_{11}V_1 + A_{12}V_2 + A_{13}V_3$$

최종 출력 = 밸류 벡터들의 가중합

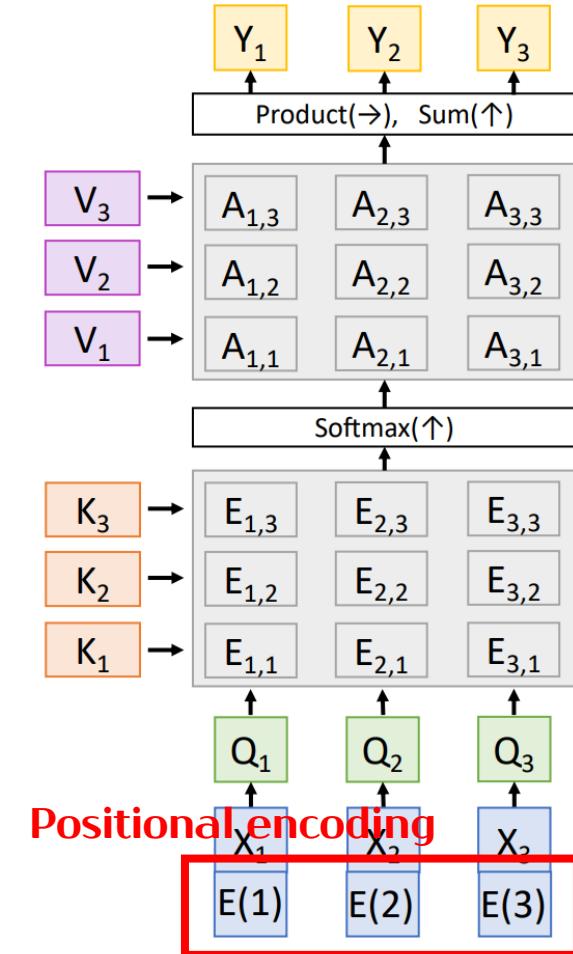
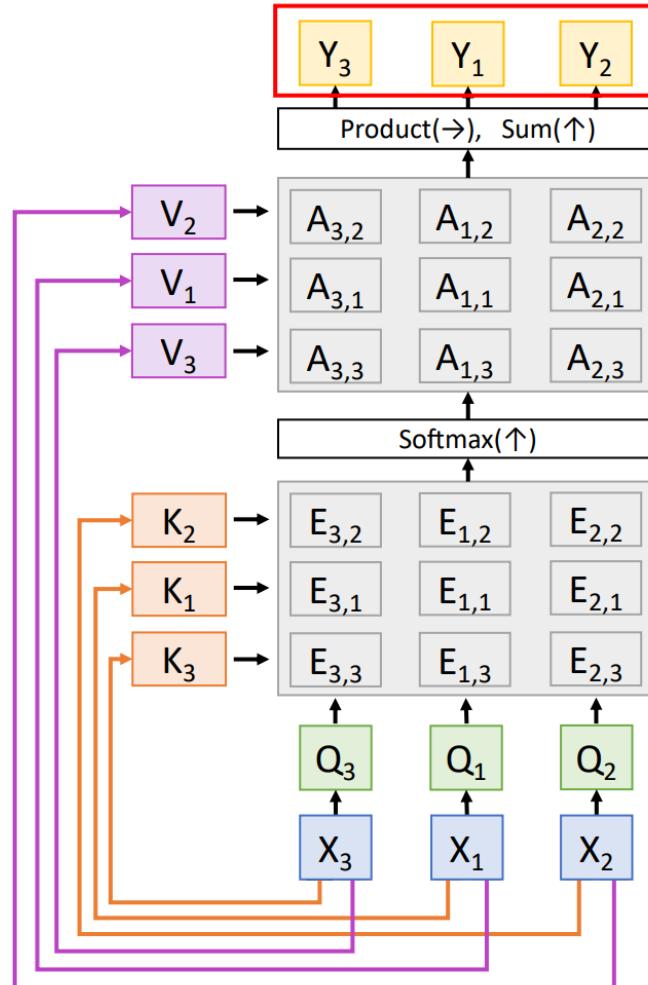


2. Attention layers

YONSEI Data Science Lab | DSL

(2) Self - Attention layer

- 순서가 바뀐다면?
 - 벡터 seq 순서 모름
(순서를 기억하는 애 없음)
 - 기억해보자!
 - 위치 정보 추가
: Positional encoding

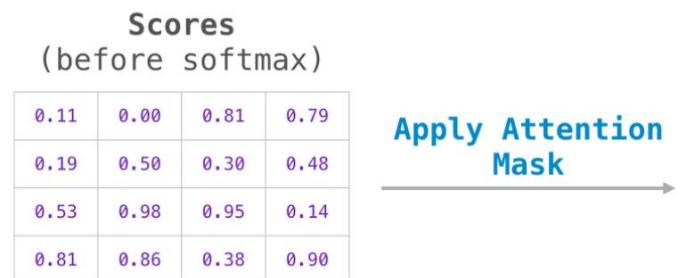
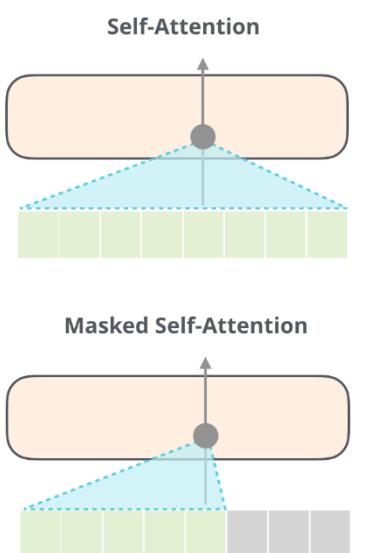


2. Attention layers

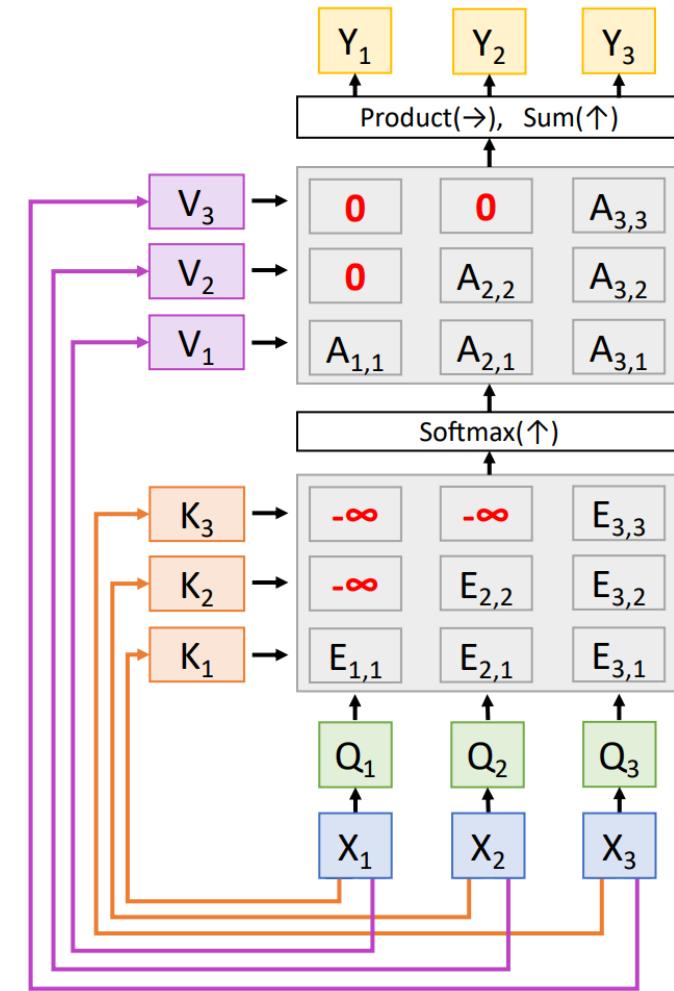
YONSEI Data Science Lab | DSL

(3) Variants of the Self-attention layer

- Masked Self-Attention Layer 뒤에서 더 보자
- Don't let vectors "look ahead" in the sequence
- 디코더 과정에서 타깃 단어 이후의 단어를 보지 않고 예측하기 위해 사용
- 가려주고자 하는 토큰에 -무한대의 값에 해당하는 수를 더해줌



	0.11	-inf	-inf	-inf
0.11	0.11	-inf	-inf	-inf
0.19	0.19	0.50	-inf	-inf
0.53	0.53	0.98	0.95	-inf
0.81	0.81	0.86	0.38	0.90



2. Attention layers

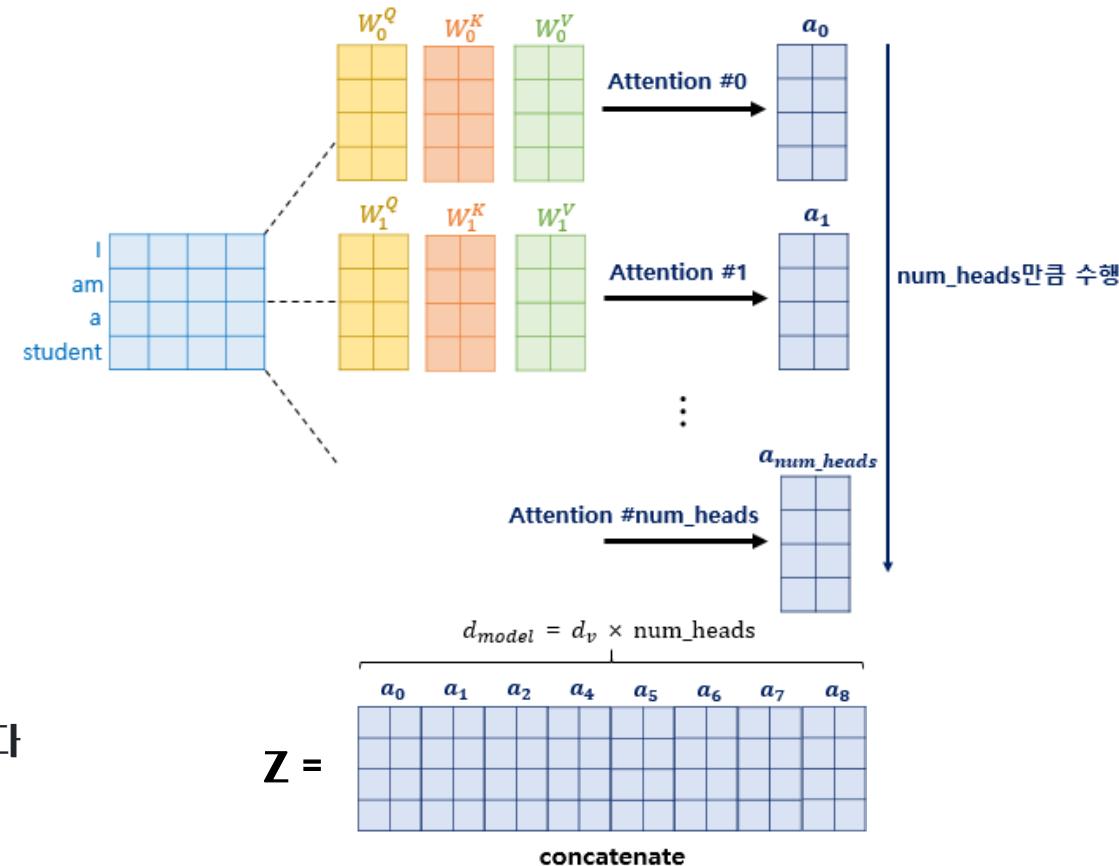
YONSEI Data Science Lab | DSL

(3) Variants of the Self-attention layer

- Multihead Self-Attention Layer [뒤에서 더 보자](#)
- 한 번의 어텐션을 하는 것보다 어텐션을 병렬로 여러번 사용하는 것이 더 효과적
- d_{model} 의 차원을 num_heads 개로 나누어서 $d_{\text{model}}/\text{num_heads}$ 의 차원을 갖는 Q, K, V에 대해 num_heads 개의 병렬 어텐션을 수행
- 가중치 행렬의 값은 8개의 어텐션 헤드마다 전부 다름
= 헤드마다 다른 시각으로 정보를 수집

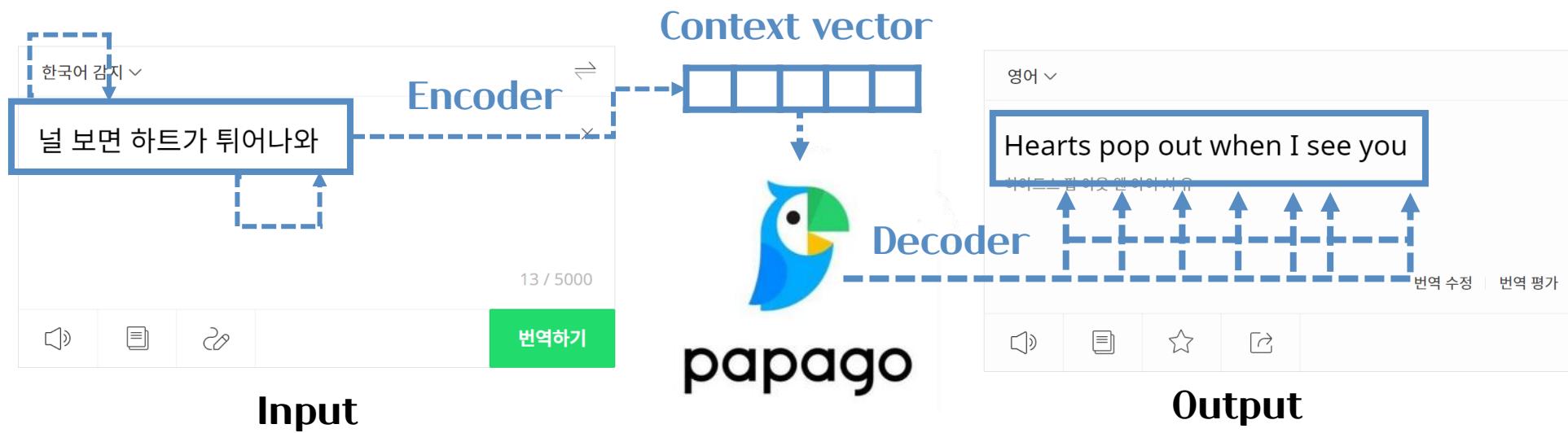
예) 그 동물은 길을 건너지 않았다. 왜냐하면 그것은 너무 피곤하였기 때문이다

- 1st head: '그것'과 '동물'의 연관도를 높게
- 2nd head: '그것'과 '피곤하였기 때문이다'의 연관도를 더 높게

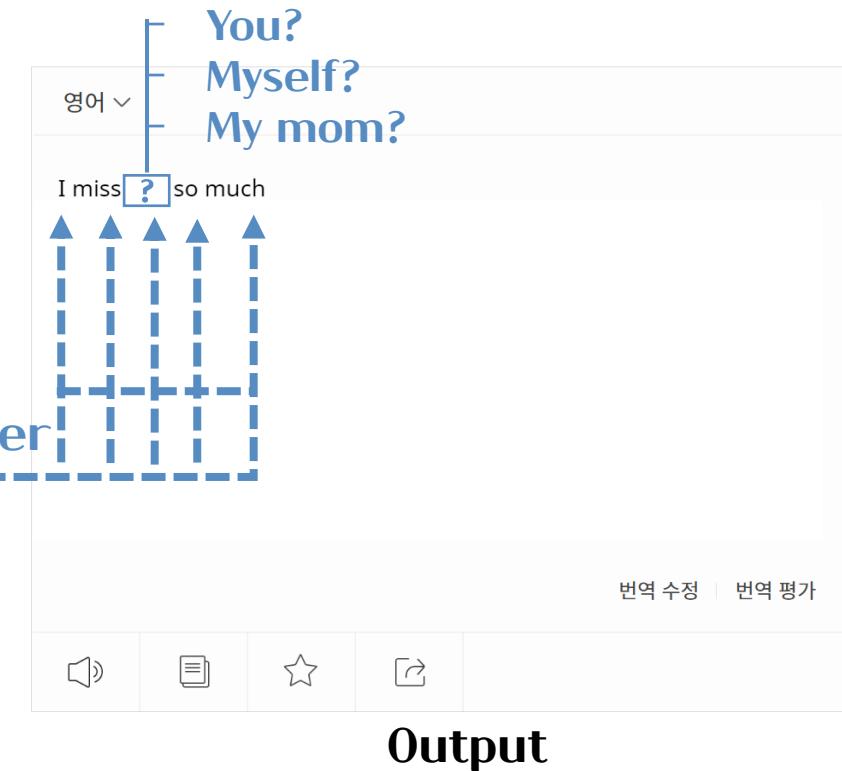
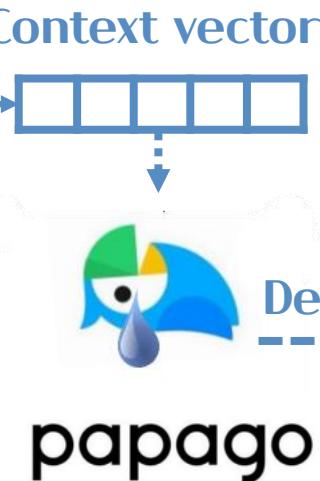
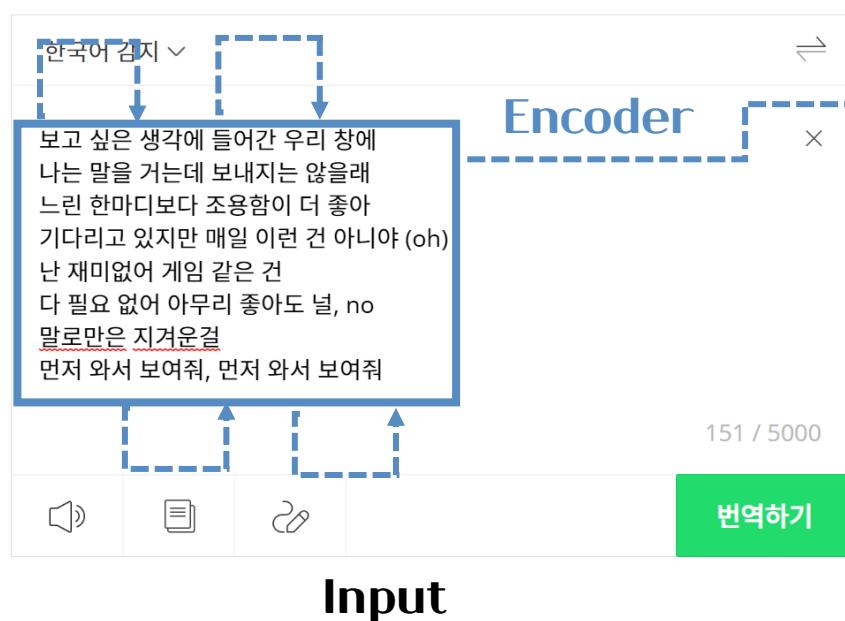


$$Z' = Z \times W^0 : \text{최종 output}$$

Seq2Seq with RNN vs Seq2Seq with Attention

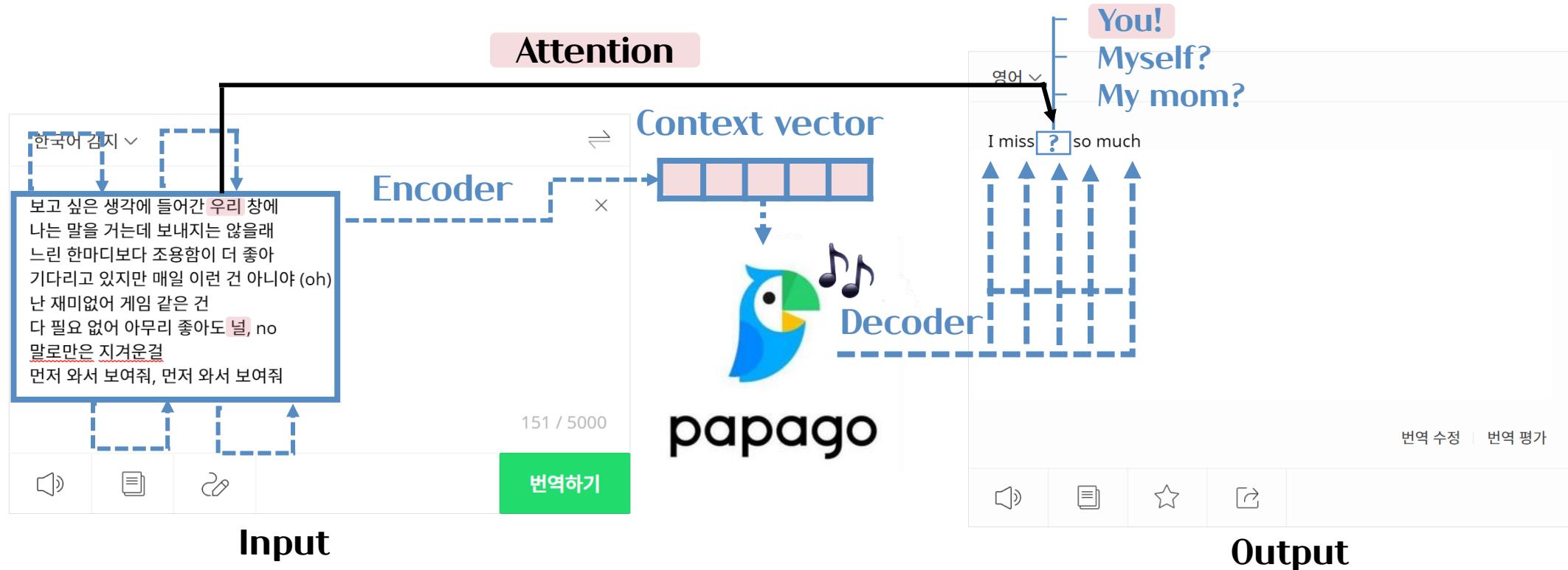


Seq2Seq with RNN vs Seq2Seq with Attention



- 장기 의존성 문제(과거 정보가 마지막까지 전달되지 못하는 것)
- 고정된 Context vector 크기

Seq2Seq with RNN vs Seq2Seq with Attention



- 디코더가 특정 시점의 단어를 출력할 때 인코더의 정보 중 연관성이 있는 부분에 주목하여 **adaptive context vector**를 만들어 냄

Attention vs Self-Attention



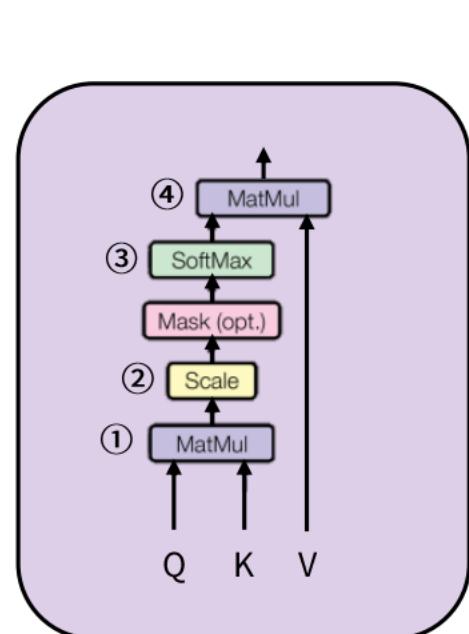
▪ Q, K, V 생성

- Attention : Q는 디코더에서 도출되고, K와 V는 인코더에서 도출됨
- Self-Attention : Q, K, V는 모두 동일한 Vector(Embedding Vector)에서 도출됨

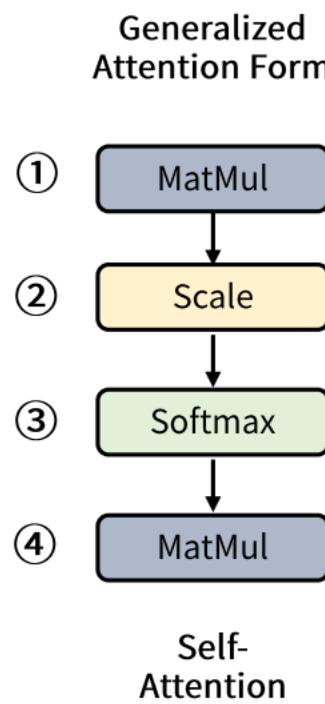
▪ Time Step

- Attention : Time-Step을 활용함
- Self-Attention : Time-Step을 활용하지 않음
 - ✓ Attention은 이전 Model에서 반환된 Hidden State Vector를 활용함. 즉, Time Step이 필요한 연산
 - ✓ Self-Attention은 이전 단어에 대한 Attention 결과를 활용하지 않음. 단지, 단어 한 개에 대해 모든 단어에 대하여 Attention Score를 구하므로 이전 연산이 현재 단어에 대한 연산에 영향을 끼치지 못함. 즉 Time-Step이 의미가 없음

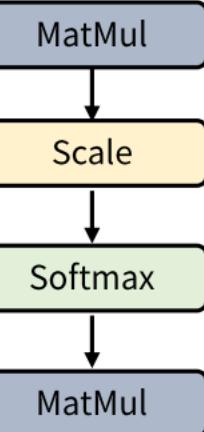
Self-Attention vs Multi head Attention



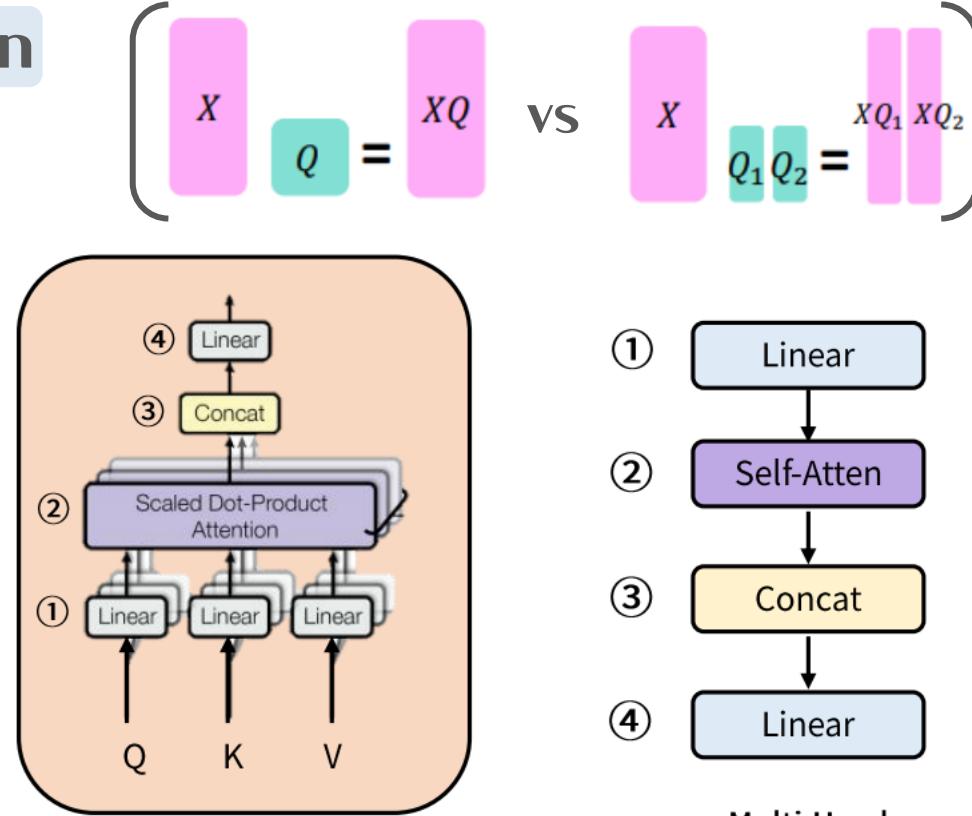
Scaled Dot-Product
Attention



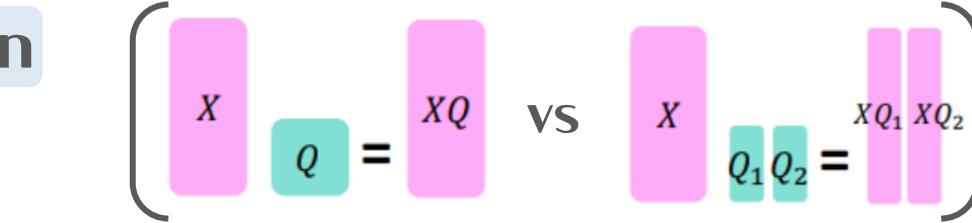
Generalized
Attention Form



Self-
Attention



Multi-Head
Attention



- 인코더 디코더 사이가 아닌 한 문장 안에서의 attention 구성
- 한번에 여러 부분에 집중하기 위해 여러 개의 attention head 구성

- 지금까지 살펴본 Attention Mechanism
- 타겟 문장을 생성할 때 소스 문장의 여러 부분을 한꺼번에 참고하기 위해 만들어진 구조
 - "그것"을 번역할 때 "그 동물", "길", 2단어에 함께 attention을 줄 수 있으면 "그것"이 "그 동물"이란 걸 컴퓨터가 알 수 있지 않을까?
- 근데 Attention Mechanism만 있으면 순차적으로 읽는 RNN 필요하지 않음
- 그렇게 인코더와 디코더만 남게 되었다 = 앞으로 살펴볼 Transformer



II. Transformer

22.09.15 / 7기 전혜령

1. The Transformer

- Sequence to Sequence
- Encoder & Decoder
- Model Training & Inference
- Transformer Block
- Self-attention

2. How self-attention works

- Input & Output
- Self-attention 내부 동작
- Multi-head attention
- Encoder에서의 Self-attention
- Decoder에서의 Self-attention

3. Technics

- Transformer Block (Feedforward / Add / Norm)
- 모델 학습 기법

4. Limitation

- Limitation of CNN
- Limitation of RNN
- CNN vs RNN vs Transformer
- Limitation of Transformer

I. The Transformer

YONSEI Data Science Lab | DSL

(0) Transformers

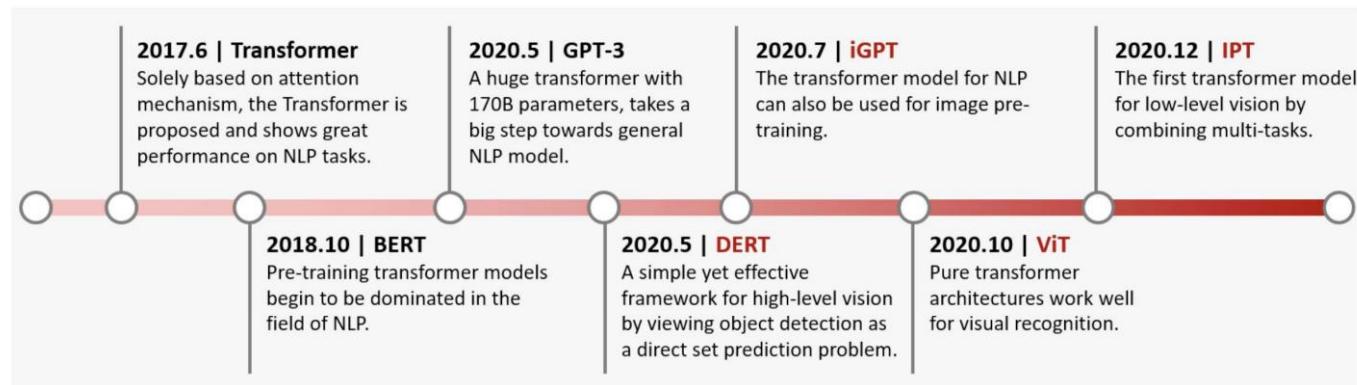
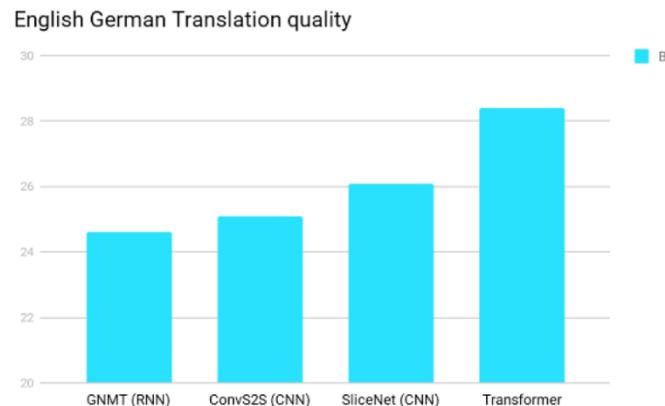


Figure 1: Key milestones in the development of transformer. The visual transformer models are marked in red.

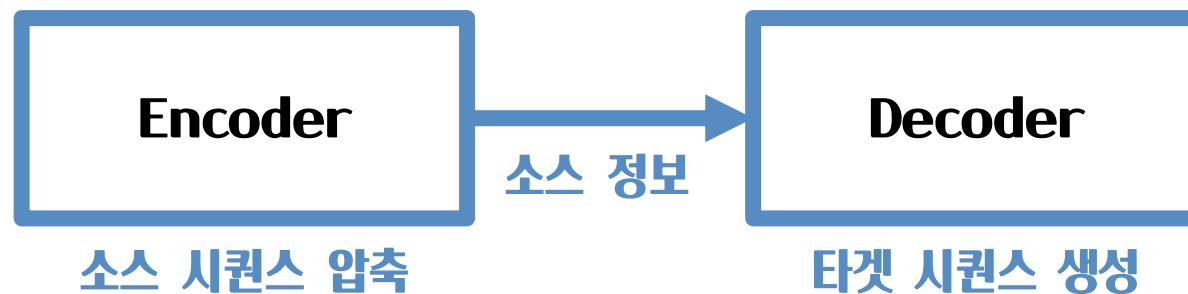
- Self -attention 기반의 뉴럴 네트워크 (Attention is all you need = RNN과 CNN 모두 필요 없음)
- 2017년에 기계 번역 태스크를 위해 처음 등장
- Transformer = Encoder + Decoder + Attention Modules
- 현재 NLP 뿐만 아니라 Vision 태스크에서도 널리 사용됨

(I) Sequence to Sequence

- Transformer는 주로 seq to seq 과제를 수행하기 위한 모델
- 시퀀스 투 시퀀스란? (= seq to seq)
 - 시퀀스: 단어 같은 무언가의 나열
 - 특정 속성을 지닌 시퀀스를 다른 속성의 시퀀스로 변환하는 과정
- 기계 번역에서의 seq to seq
 - 기계 번역이란 어떤 언어의 단어 시퀀스를 다른 언어의 단어 시퀀스로 변환하는 과제
 - 어제, 카페, 갔었어, 거기, 사람, 많더라
 - I, went, to, the, café, There, were, many, people, there
 - 이처럼 seq to seq task는 소스와 타깃의 길이가 달라도 해당 과제를 수행하는 데 문제가 없어야 함

(2) Encoder & Decoder

- seq to seq 과제를 수행하는 모델은 대개 인코더(encoder)와 디코더(decoder) 두 개 파트로 구성됨



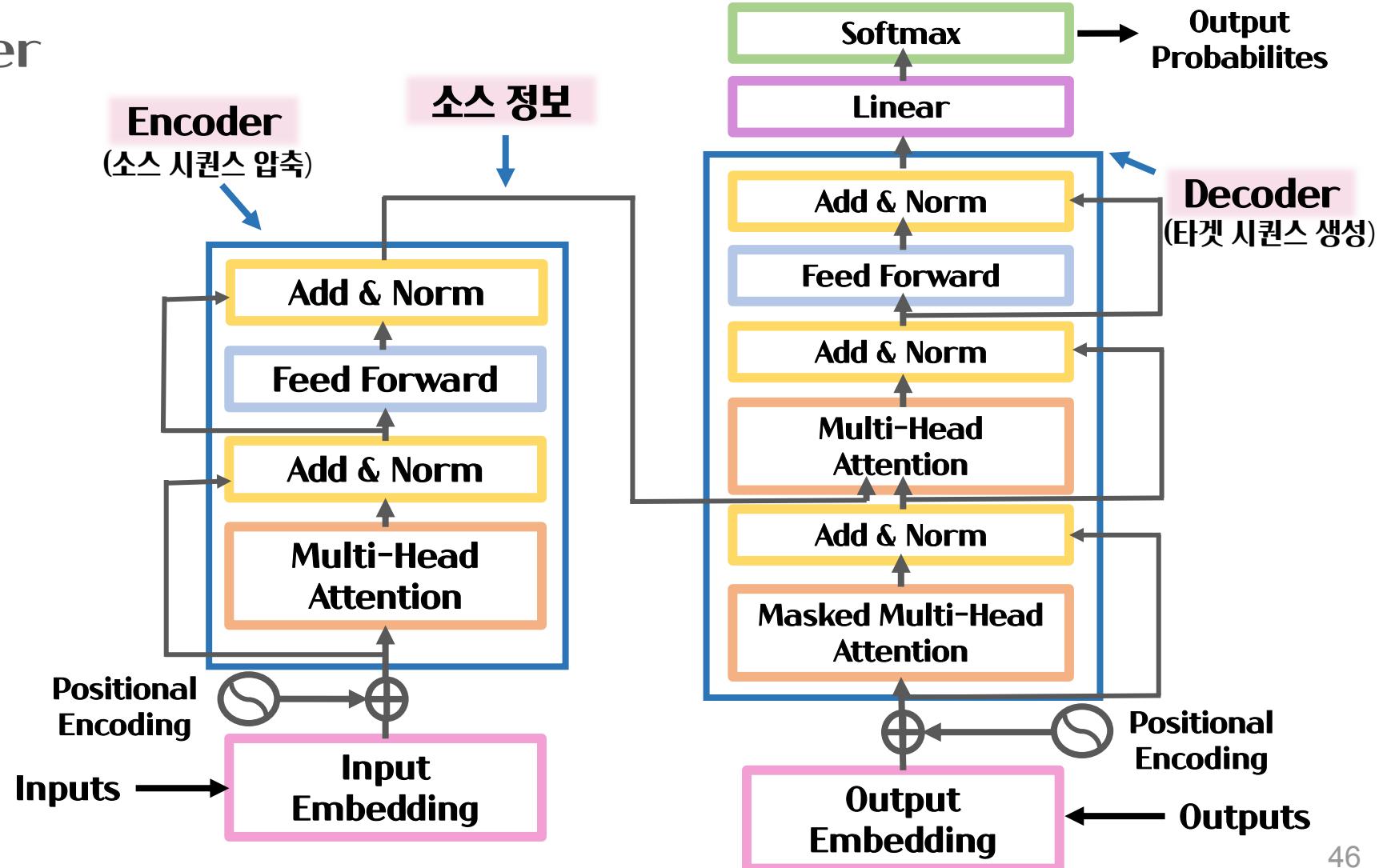
- 인코더는 소스 시퀀스의 정보를 압축해 디코더로 보내주는 역할을 담당
 - 인코딩 = 인코더가 소스 시퀀스의 정보를 압축하는 과정
- 디코더는 인코더가 보내준 소스 시퀀스를 받아서 타겟 시퀀스를 생성
 - 디코딩 = 디코더가 타겟 시퀀스를 생성하는 과정

I. The Transformer

YONSEI Data Science Lab | DSL

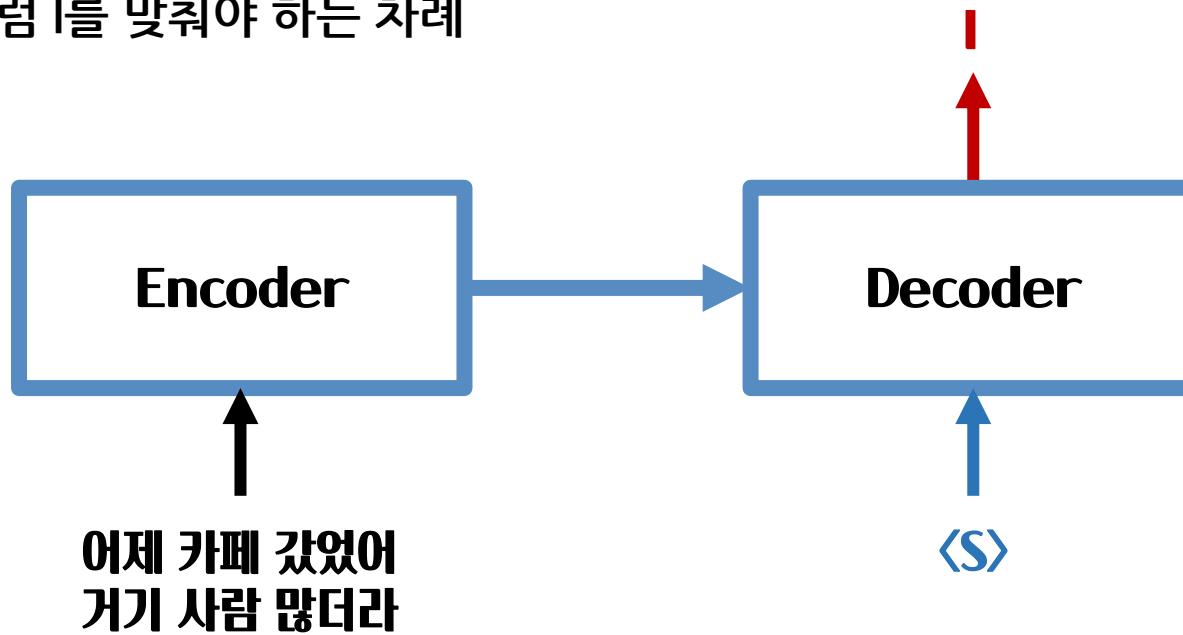
(2) Encoder & Decoder

- 트랜스포머 역시 인코더와 디코더의 구조를 따르며 다음 그림과 같음

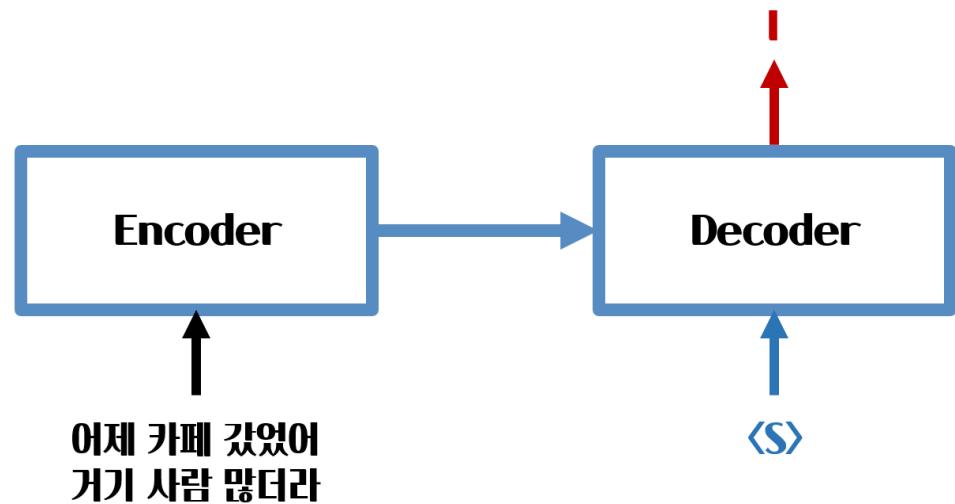


(3) Model training and Inference

- 트랜스포머가 어떻게 학습되는지 알아보자
- 이번 학습은 그림처럼 I를 맞춰야 하는 차례

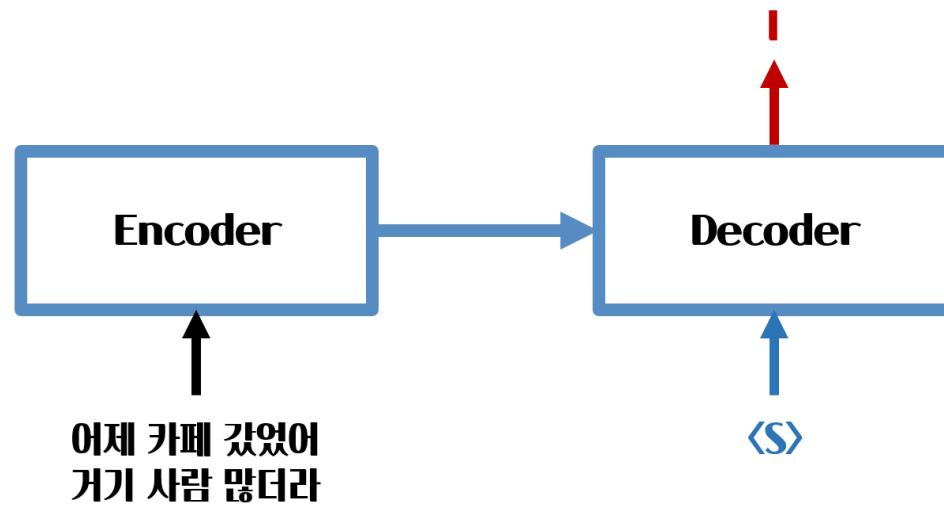


(3) Model training and Inference

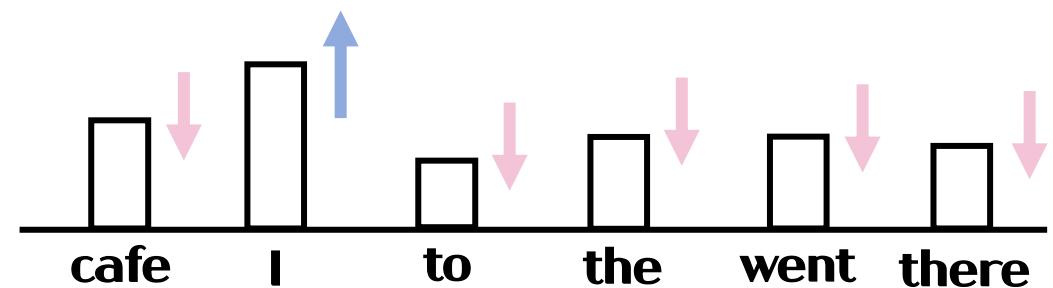


- 이번 학습은 그림처럼 I를 맞춰야 하는 차례
 - 이 때 인코더 입력은 소스 시퀀스 전체
 - 디코더 입력은 <S>로, 이는 타깃 시퀀스의 시작을 뜻하는 스페셜 토큰
 - 인코더는 소스 시퀀스를 압축해 디코더로 보내고, 디코더는 인코더에서 보내온 정보와 디코더 입력을 모두 감안해 다음 토큰 I를 맞춤
 - 최종 출력, 즉 디코더 출력은 타깃 언어의 어휘 수 만큼의 차원으로 구성된 벡터

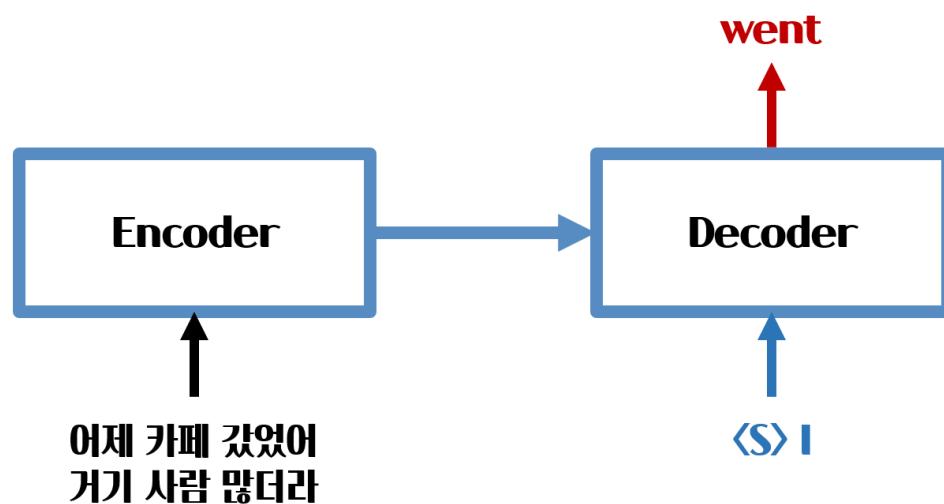
(3) Model training and Inference



- 이번 학습은 그림처럼 I를 맞춰야 하는 차례
 - 트랜스포머의 학습은 인코더와 디코더 입력이 주어졌을 때, 정답에 해당하는 단어의 확률값을 높이는 방식으로 수행됨
 - I에 해당하는 확률 높이고 나머지 단어의 확률은 낮아져야 함



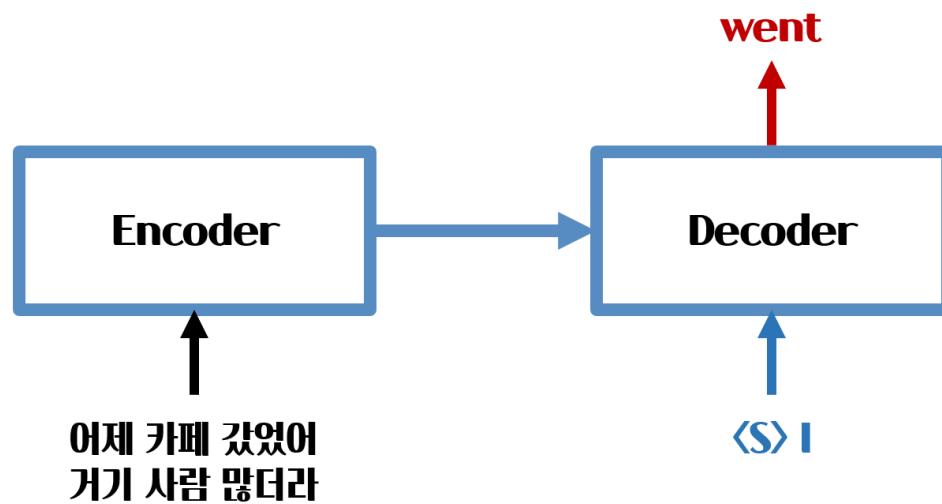
(3) Model training and Inference



- 이제 **went**를 맞출 차례

- 인코더 입력은 소스 시퀀스 전체, 디코더 입력은 $\langle s \rangle$ |
- 여기서 특이한 점은 학습 중의 디코더 입력과 학습 후 모델을 실제 기계 번역에 사용할 때 (Inference) 디코더 입력이 다름
- 학습 과정에서는 디코더 입력에 맞춰야 할 단어 (**went**) 이전의 정답 시퀀스 ($\langle s \rangle$ |)를 넣어줌

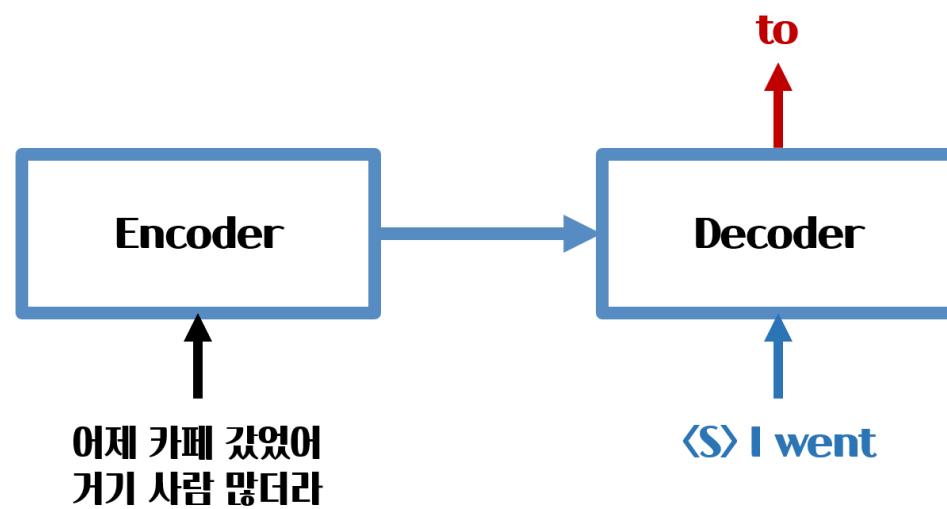
(3) Model training and Inference



- 이제 **went**를 맞출 차례

- 그러나 인퍼런스 때는 현재 디코더 입력에 직전 디코딩 결과를 사용
 - 학습이 잘못되어 | 대신 you라는 단어가 예측되었다면 입력은 <s> you가 됨
- 학습 과정 중 모델은 이번 시점의 정답인 **went**에 해당하는 확률을 높이고 나머지 단어의 확률은 낮아지도록 모델 갱신

(3) Model training and Inference

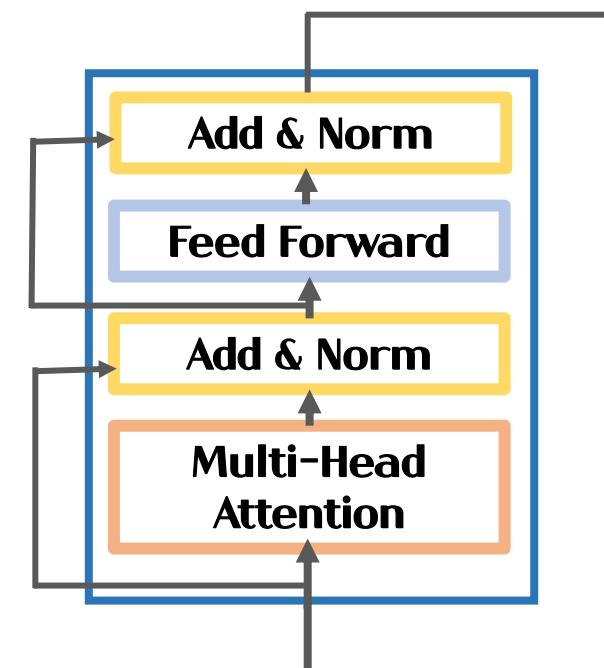


▪ 이제 to를 맞출 차례

- 인코더 입력은 소스 시퀀스 전체, 디코더 입력은 학습 과정 중에는 <S> I went, 인퍼런스 때는 직전 디코딩 결과
- 학습 과정 중 모델은 이번 시점의 정답인 to에 해당하는 확률은 높이고 나머지 단어의 확률은 낮아지도록 모델 갱신
- 이런 식으로 말뭉치 전체 반복 학습

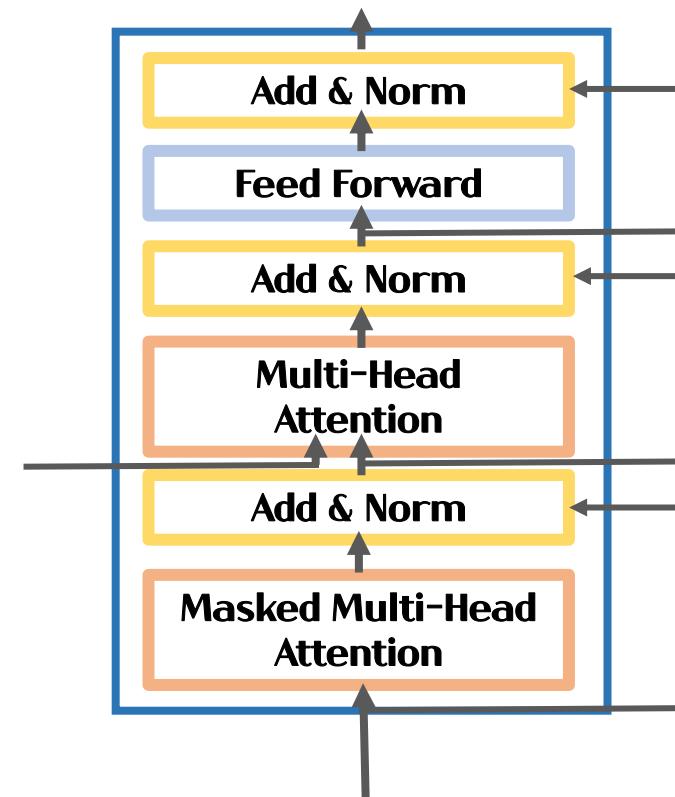
(4) Transformer Block

- 인코더 블록
- 옆의 그림은 트랜스포머의 인코더 가운데 반복되는 요소를 떼어내 다시 나타낸 것
- 트랜스포머의 인코더는 이같은 블록을 수십개 쌓아서 구성
- 트랜스포머의 인코더 블록은 다음과 같은 세가지 요소로 구성
 - 멀티 헤드 어텐션 (Multi-head attention)
 - 피드포워드 뉴럴네트워크 (Feed Forward)
 - 잔차 연결 및 레이어 정규화 (Add & Norm)



(4) Transformer Block

- 디코더 블록
- 디코더 쪽 블록의 구조도 인코더 블록과 본질적으로 동일
- 다만 마스크를 적용한 멀티 헤드 어텐션이 인코더 쪽과 다르고,
인코더가 보내온 정보와 디코더 입력을 함께 이용해 멀티 헤드 어텐션
을 수행하는 모듈이 추가
- 트랜스포머의 디코더 블록은 다음과 같은 네 가지 요소로 구성
 - 마스크를 적용한 멀티 헤드 어텐션
(Masked Multi-head attention)
 - 멀티 헤드 어텐션 (Multi-head attention)
 - 피드포워드 뉴럴네트워크 (Feed Forward)
 - 잔차 연결 및 레이어 정규화 (Add & Norm)



(5) Self-attention

- 어텐션은 시퀀스 입력에 수행하는 기계 학습 방법의 일종
 - 시퀀스 요소들 가운데 태스크 수행에 중요한 요소에 집중하고 그렇지 않은 요소는 무시해 태스크 수행 성능을 끌어올림
 - 기계 번역에 어텐션을 적용한다면 타깃 언어를 디코딩할 때 소스 언어의 단어 시퀀스 가운데 디코딩에 도움되는 단어들 위주로 취사 선택해서 번역 품질을 끌어올림
- 셀프 어텐션이란, 말 그대로 자기 자신에 수행하는 어텐션 기법
 - 입력 시퀀스 가운데 태스크 수행에 의미 있는 요소들 위주로 정보를 추출

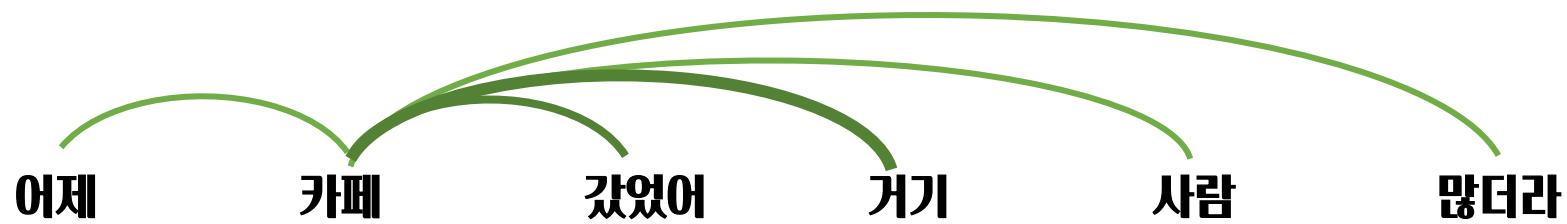
(5) Self-attention



- 잘 학습된 셀프 어텐션 모델이라면 ‘거기’에 대응하는 장소는 ‘카페’임을 알 수 있음
- 뿐만 아니라 ‘거기’는 ‘갔었어’와도 연관이 있음을 알 수 있음
- 트랜스포머의 인코더 블록 내부에서는 ‘거기’라는 단어를 인코딩 할 때 ‘카페’, ‘갔었어’라는 단어의 의미를 강조해서 반영

(5) Self-attention

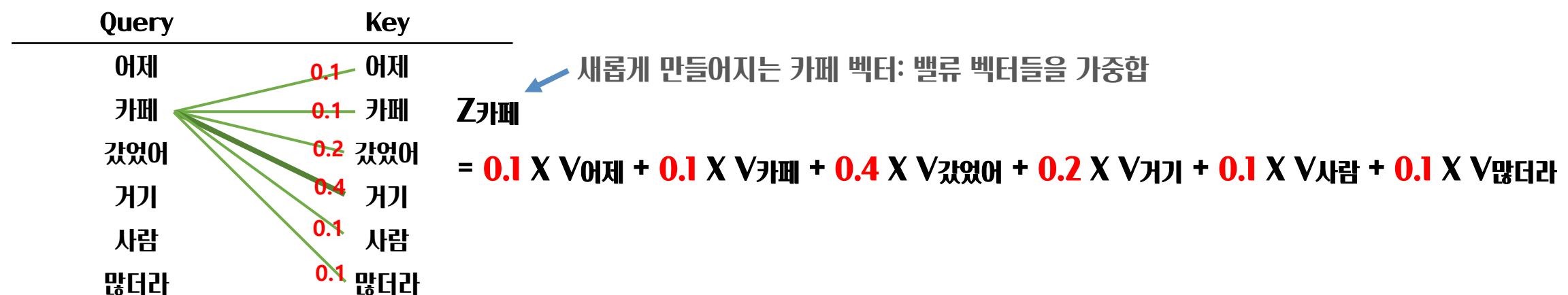
- ‘카페’라는 단어의 의미를 계산하는 경우 ‘거기’, ‘갔었어’라는 단어의 의미를 강조해서 반영



- 셀프 어텐션 수행 대상은 입력 시퀀스 전체
 - ‘카페’, ‘거기’ 뿐만 아니라 ‘어제’, ‘갔었어’, … 모두 어텐션 계산을 수행

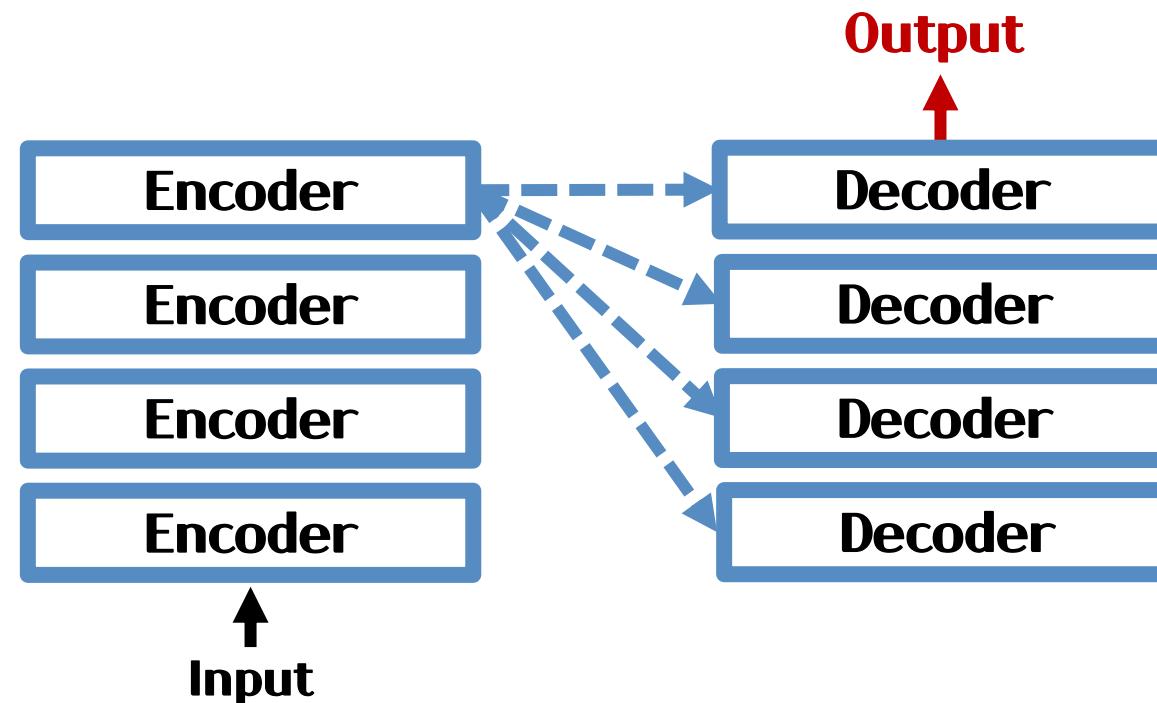
(5) Self-attention

- 계산 예시
- 셀프 어텐션은 쿼리(query), 키(key), 밸류(value) 세 가지 요소가 서로 영향을 주고 받는 구조
- 트랜스포머 블록에는 문장 내 각 단어가 벡터 형태로 입력
- 각 단어 벡터는 블록 내에서 계산 과정을 거쳐 키, 쿼리, 밸류로 변환됨 (뒤에서 예시로 보자!)
 - 문장이 여섯 단어라면 쿼리 벡터 6개, 키 벡터 6개, 밸류 벡터 6개 모두 18개가 됨



(5) Self-attention

- 모든 시퀀스 대상으로 셀프 어텐션 계산이 끝나면 그 결과를 다음 블록으로 넘김
- 이처럼 트랜스포머 모델은 셀프 어텐션을 레이어 수만큼 반복

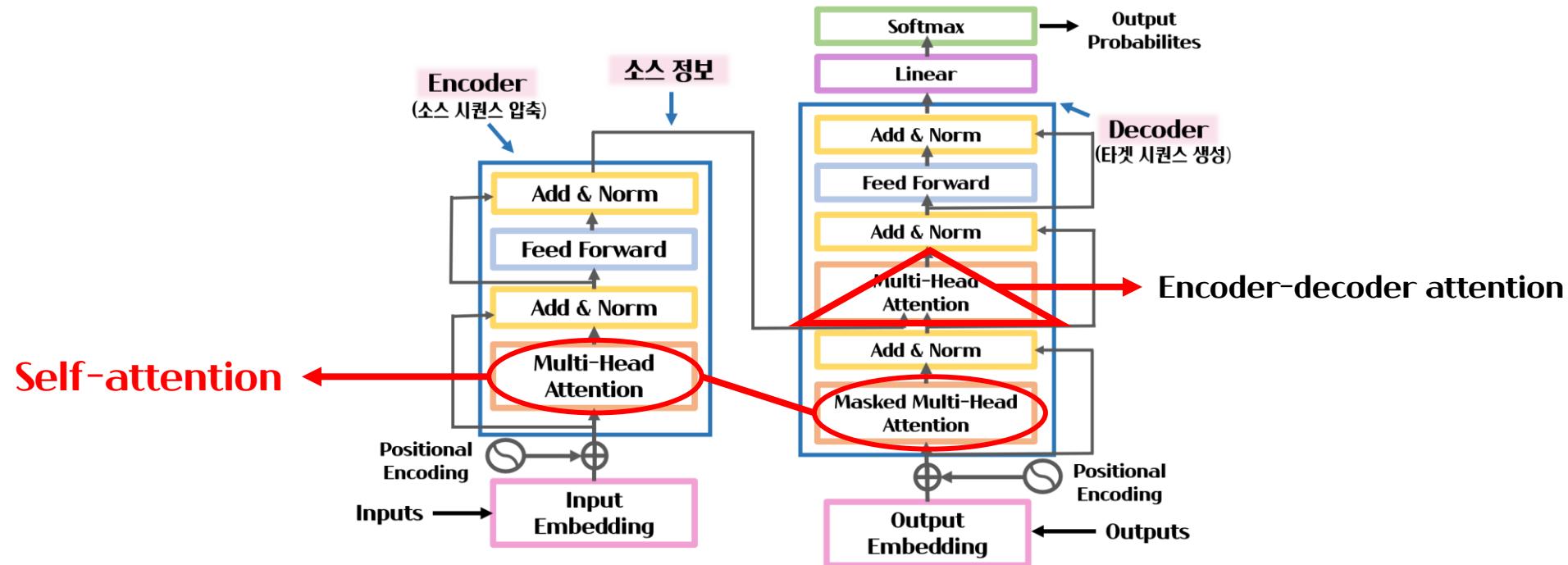


I. The Transformer

YONSEI Data Science Lab | DSL

(5) Self-attention

- 모든 시퀀스 대상으로 셀프 어텐션 계산이 끝나면 그 결과를 다음 블록으로 넘김
- 이처럼 트랜스포머 모델은 셀프 어텐션을 레이어 수만큼 반복

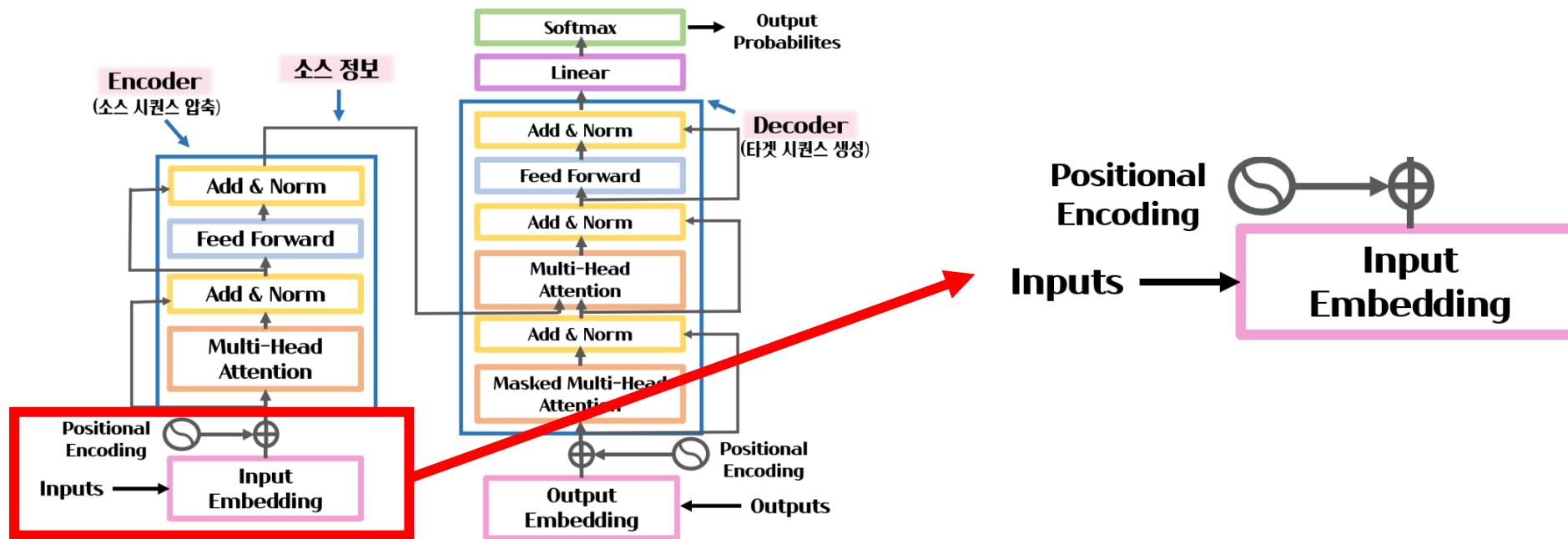


2. How Self Attention works

YONSEI Data Science Lab | DSL

(I) Input & Output

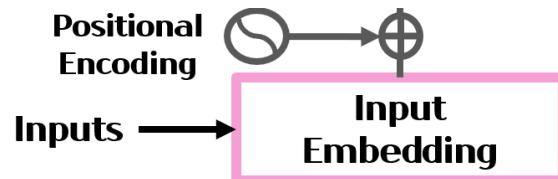
- 입력
- 왼쪽 그림은 트랜스포머 모델의 전체 구조를, 오른쪽 그림은 인코더 입력만을 떼어서 나타낸 그림
- 오른쪽 그림과 같이 모델의 입력을 만드는 계층(layer)를 입력층(input layer)이라고 함



2. How Self Attention works

YONSEI Data Science Lab | DSL

(I) Input & Output



- 그림처럼 인코더의 입력은 소스 시퀀스의 입력 임베딩(input embedding)에 위치 정보(positional encoding)를 더해서 생성
- 입력 임베딩 (input embedding)
 - 인코더에서 단어들을 처리하기 전에 먼저 벡터들로 변환해주어야 함
 - Word embedding 알고리즘 이용 / pre-trained embeddings 이용 / 데이터셋을 이용해 직접 학습

어제

0.901	-0.651	-0.194	-0.882
-------	--------	--------	--------



카페

-0.351	0.123	0.435	-0.200
--------	-------	-------	--------



갔었어

0.081	0.458	-0.400	0.480
-------	-------	--------	-------



2. How Self Attention works

YONSEI Data Science Lab | DSL

(1) Input & Output

- 이 벡터 리스트의 사이즈는 hyperparameter으로 우리가 마음대로 정할 수 있음
 - 가장 간단하게 생각한다면 우리의 학습 데이터 셋에서 가장 긴 문장의 길이로 둘 수 있음
 - 발표에서는 예시로서 크기 4의 벡터 사용 (논문에서는 크기 512의 벡터 사용)



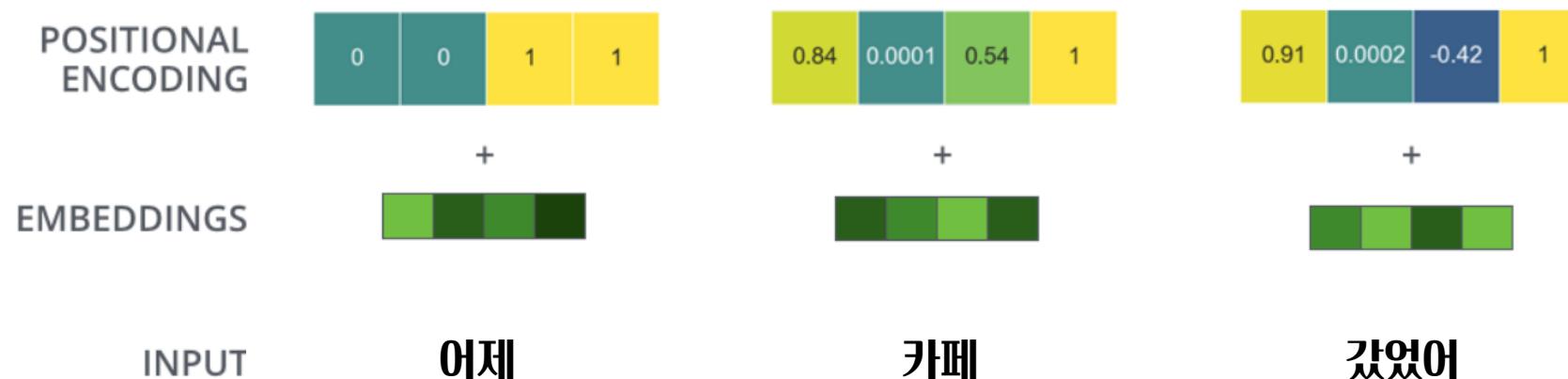
- 위치 정보(positional encoding)
 - 입력 임베딩에 더하는 위치 정보는 해당 토큰이 문장 내에서 몇 번째 위치인가에 대한 정보를 나타냄
 - 그림에서는 ‘어제’가 첫 번째, ‘카페’가 두 번째, ‘갔었어’가 세 번째

2. How Self Attention works

YONSEI Data Science Lab | DSL

(1) Input & Output

- 위치 정보(positional encoding)
 - 시각화하면 아래와 같음



- 어떻게 만드는 걸까?

2. How Self Attention works

YONSEI Data Science Lab | DSL

(I) Input & Output

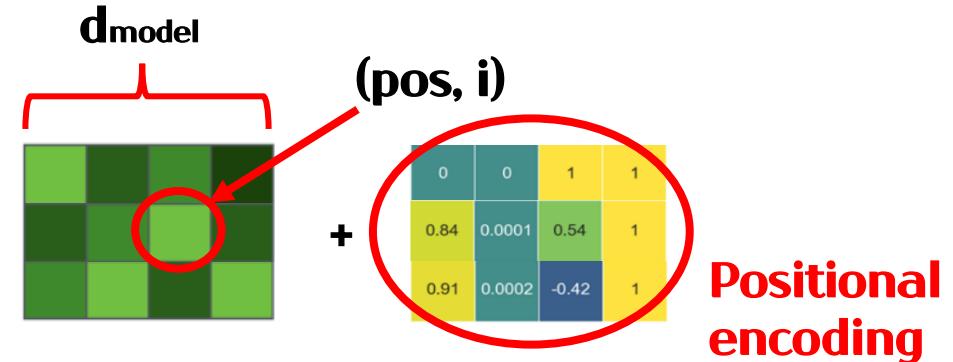
▪ 위치 정보(positional encoding)

- 아래의 두 개의 함수를 사용

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

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

- 임베딩 벡터 내의 각 차원의 인덱스가 짝수인 경우에는 사인 함수의 값을 사용하고 홀수인 경우에는 코사인 함수의 값을 사용
- 각 임베딩 벡터에 포지셔널 인코딩의 값을 더하면 같은 단어라고 하더라도 문장 내의 위치에 따라서 트랜스포머의 입력으로 들어가는 임베딩 벡터의 값이 달라짐
- 이에 따라 트랜스포머의 입력은 순서 정보가 고려된 임베딩 벡터가 됨



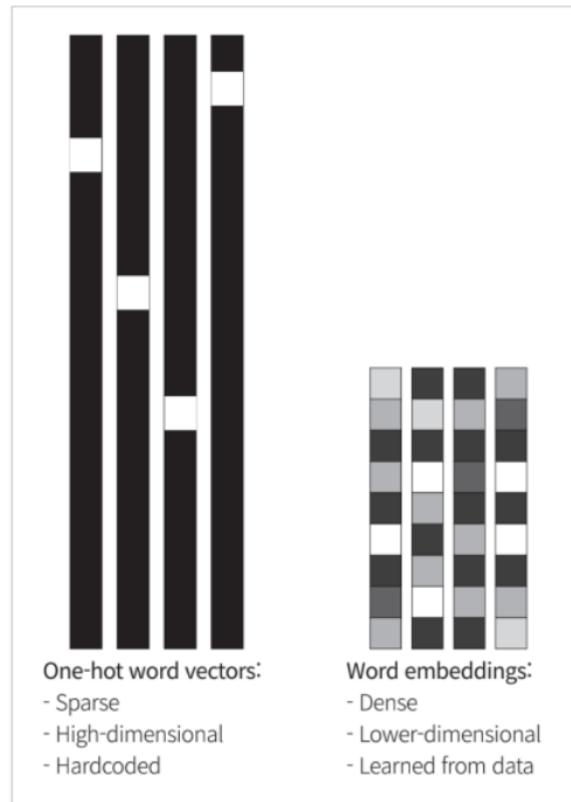
2. How Self Attention works

YONSEI Data Science Lab | DSL

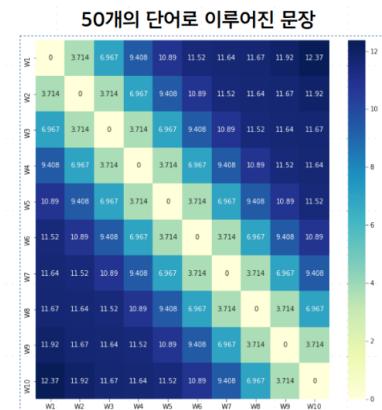
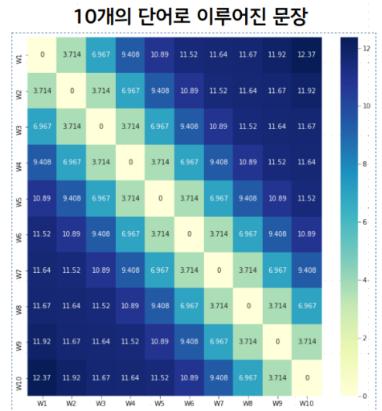
(I) Input & Output

▪ 위치 정보(positional encoding)

- one hot encoding에 비해 word2vec 알고리즘을 활용한다면 정해진 차원 내로 단어를 표현하여 차원이 기하급수적으로 늘어나는 것을 방지할 수 있는 장점 존재
- 문장의 길이가 달라지더라도 단어 사이의 거리는 보존 (문장의 길이가 길어져도 포지셔널 인코딩 값이 한없이 커지지 않음)



<그림3> ‘One-hot Vector’와 워드 임베딩(Word2Vector)



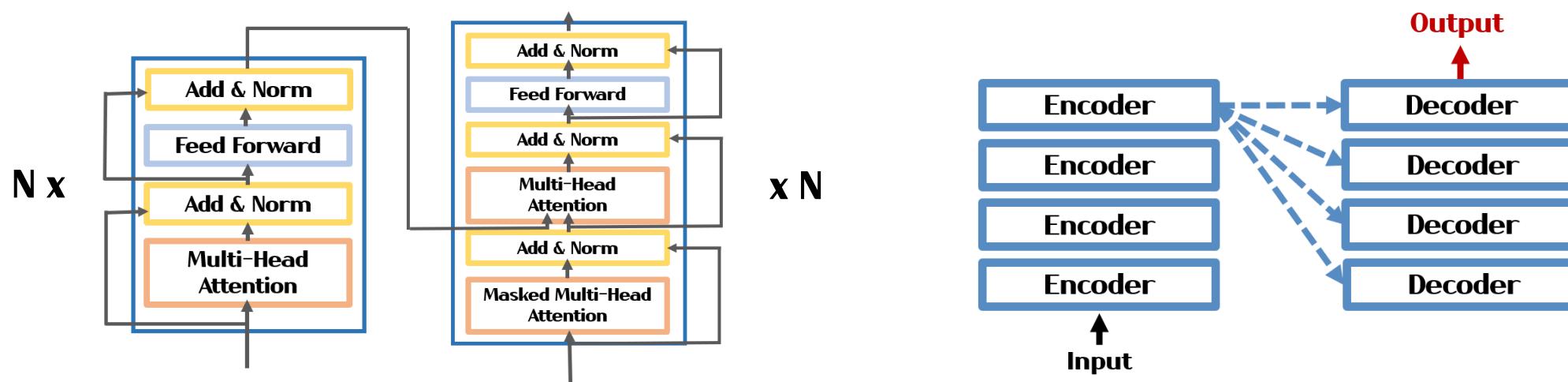
(앞 10단어 시각화)

2. How Self Attention works

YONSEI Data Science Lab | DSL

(I) Input & Output

- 트랜스포머 모델은 이와 같은 방식으로 소스 언어의 토큰 시퀀스를 이에 대응하는 벡터 시퀀스로 변환해 인코더 입력을 만들어 냄!
- 입력층에서 만들어진 벡터 시퀀스가 최초 인코더 블록이 되며, 그 출력 벡터 시퀀스는 두 번째 인코더 블록의 입력이 됨
 - 다음 인코더 블록의 입력은 이전 블록의 출력, 이를 N번 반복

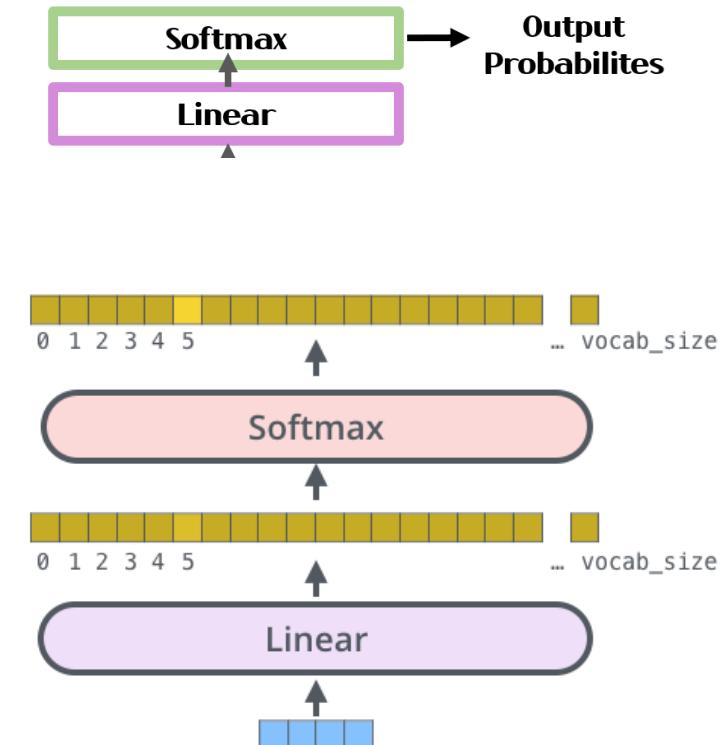


2. How Self Attention works

YONSEI Data Science Lab | DSL

(I) Input & Output

- 출력
- 그림은 트랜스포머의 전체 구조에서 출력층(Output layer)만을 떼어낸 것
- 이 출력층의 입력은 디코더 마지막 블록의 출력 벡터 시퀀스
- 출력층의 출력은 타깃 언어의 어휘 수 만큼의 차원을 갖는 확률 벡터
 - 벡터의 요소 값 모두 더하면 1
- 트랜스포머의 학습은 인코더와 디코더 입력이 주어졌을 때 모델 최종 출력에서 정답에 해당하는 단어의 확률값을 높이는 방식으로 수행



2. How Self Attention works

YONSEI Data Science Lab | DSL

(I) Input & Output

▪ 출력

Target Model Outputs

Output Vocabulary: a am I thanks student <eos>



a am I thanks student <eos>

Trained Model Outputs

Output Vocabulary: a am I thanks student <eos>



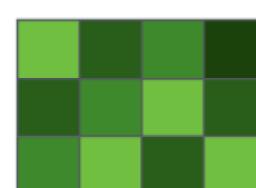
a am I thanks student <eos>

2. How Self Attention works

YONSEI Data Science Lab | DSL

(2) Self-attention 내부 동작

- 트랜스포머 모델 핵심인 셀프 어텐션 기법이 내부에서 어떻게 동작하는지를 살펴보자
- 키, 쿼리, 밸류 만들기
 - 인코더에서 수행되는 셀프 어텐션의 입력은 이전 인코더 블록의 출력 벡터 시퀀스
 - 그림처럼 단어 임베딩 차원 수(d_{model})가 4이고, 인코더에 입력된 단어 개수가 3일 경우 셀프 어텐션 입력은 아래와 같은 형태가 됨



+

0	0	1	1
0.84	0.0001	0.54	1
0.91	0.0002	-0.42	1

=

X
(입력 벡터 시퀀스)

$$= \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 2 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

2. How Self Attention works

YONSEI Data Science Lab | DSL

(2) Self-attention 내부 동작

- 키, 쿼리, 밸류 만들기

The diagram shows the calculation of query, key, and value vectors from an input vector sequence. On the left, a 4x4 grid of colored squares represents the input vector sequence. To its right is a plus sign (+). Next is a 4x4 matrix with numerical values. To the right of the matrix is an equals sign (=). Then there is a bold italicized X symbol followed by the text "(입력 벡터 시퀀스)". To the right of this is another equals sign (=). Finally, there is a large bracketed matrix containing four rows of numbers: 1 0 1 0, 0 2 0 2, 1 1 1 1.

$$\begin{matrix} \text{Input Vector Sequence} & + & \begin{matrix} 0 & 0 & 1 & 1 \\ 0.84 & 0.0001 & 0.54 & 1 \\ 0.91 & 0.0002 & -0.42 & 1 \end{matrix} & = & \mathbf{X} & = & \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 2 \\ 1 & 1 & 1 & 1 \end{pmatrix} \end{matrix}$$

- 4차원짜리 단어 임베딩이 3개 모였음을 확인 가능
- 셀프 어텐션은 쿼리, 키, 밸류 3개 요소의 문맥적 관계성을 추출하는 과정
- 다음 수식처럼 입력 벡터 시퀀스(X)에 쿼리, 키, 밸류를 만들어주는 행렬(W)을 각각 곱하기
- $Q=X \times W_Q$, $K=X \times W_K$, $V=X \times W_V$ (X : 행렬 곱셈)

2. How Self Attention works

YONSEI Data Science Lab | DSL

(2) Self-attention 내부 동작

- 키, 쿼리, 밸류 만들기
 - 쿼리 벡터 구축하기 ($Q = X \times W_Q$)

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 2 \\ 1 & 1 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 2 \\ 2 & 2 & 2 \\ 2 & 1 & 3 \end{pmatrix}$$

X
(입력 벡터 시퀀스)

W_Q

2. How Self Attention works

YONSEI Data Science Lab | DSL

(2) Self-attention 내부 동작

- 키, 쿼리, 밸류 만들기

- 키, 밸류 벡터 구축하기 ($K=X \times W_K$, $V=X \times W_V$)

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 2 \\ 1 & 1 & 1 & 1 \end{pmatrix}_X \times \begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix}_{W_K} = \begin{pmatrix} 0 & 1 & 1 \\ 4 & 4 & 0 \\ 2 & 3 & 1 \end{pmatrix}_K$$

|

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 2 \\ 1 & 1 & 1 & 1 \end{pmatrix}_X \times \begin{pmatrix} 0 & 2 & 0 \\ 0 & 3 & 0 \\ 1 & 0 & 3 \\ 1 & 1 & 0 \end{pmatrix}_{W_V} = \begin{pmatrix} 0 & 1 & 1 \\ 4 & 4 & 0 \\ 2 & 3 & 1 \end{pmatrix}_V$$

- W_Q W_K W_V 세 행렬은 태스크를 가장 잘 수행하는 방향으로 학습 과정에서 업데이트 됨

2. How Self Attention works

YONSEI Data Science Lab | DSL

(2) Self-attention 내부 동작

- 쿼리의 셀프 어텐션 출력값 계산하기

- 이제 셀프 어텐션을 계산할 준비 끝! 아래 수식은 셀프 어텐션의 정의

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_K}}\right)\mathbf{V} \quad (\sqrt{d_k} : \text{키 벡터의 차원 수})$$

- 쿼리와 키를 행렬곱한 뒤, 해당 행렬의 모든 요소값을 키 차원수의 제곱근 값으로 나눠주고, 이 행렬을 행 단위로 소프트맥스를 취해 스코어 행렬을 만들어 주기
 - 이 스코어 행렬에 밸류를 행렬곱 해줘서 계산 마무리
 - 계산해보자!

2. How Self Attention works

YONSEI Data Science Lab | DSL

(2) Self-attention 내부 동작

- 쿼리의 셀프 어텐션 출력값 계산하기

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_K}}\right)\mathbf{V}$$

$$\begin{pmatrix} 1 & 0 & 2 \\ 2 & 2 & 2 \\ 2 & 1 & 3 \end{pmatrix} \times \begin{pmatrix} 0 & 4 & 2 \\ 1 & 4 & 3 \\ 1 & 0 & 1 \end{pmatrix} = \begin{matrix} \text{어제} & \text{카페} & \text{갔었어} \\ \text{어제} & \text{카페} & \text{갔었어} \\ \text{카페} & \text{카페} & \text{카페} \\ \text{갔었어} & \text{카페} & \text{갔었어} \end{matrix} \begin{pmatrix} 2 & 4 & 4 \\ 4 & 16 & 12 \\ 4 & 12 & 10 \end{pmatrix}$$

\mathbf{Q} \mathbf{K}^\top

→ 행과 열은 단어의 개수와 동일

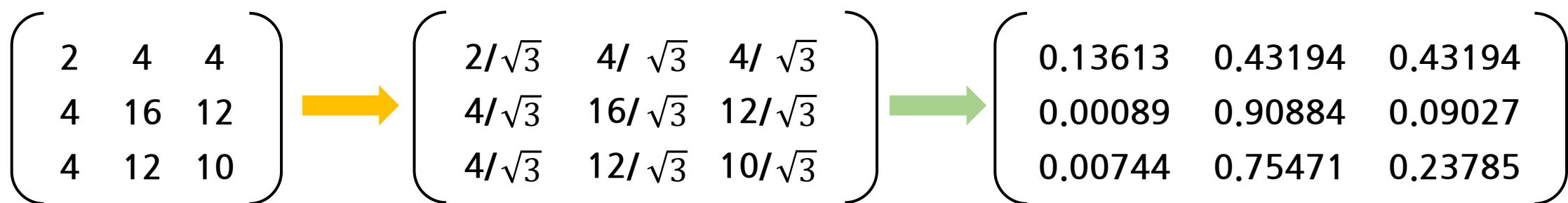
2. How Self Attention works

YONSEI Data Science Lab | DSL

(2) Self-attention 내부 동작

- 쿼리의 셀프 어텐션 출력값 계산하기

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax} \left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_K}} \right) \mathbf{V}$$



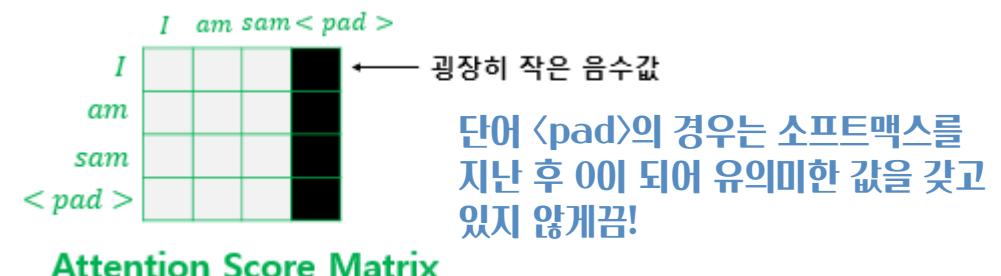
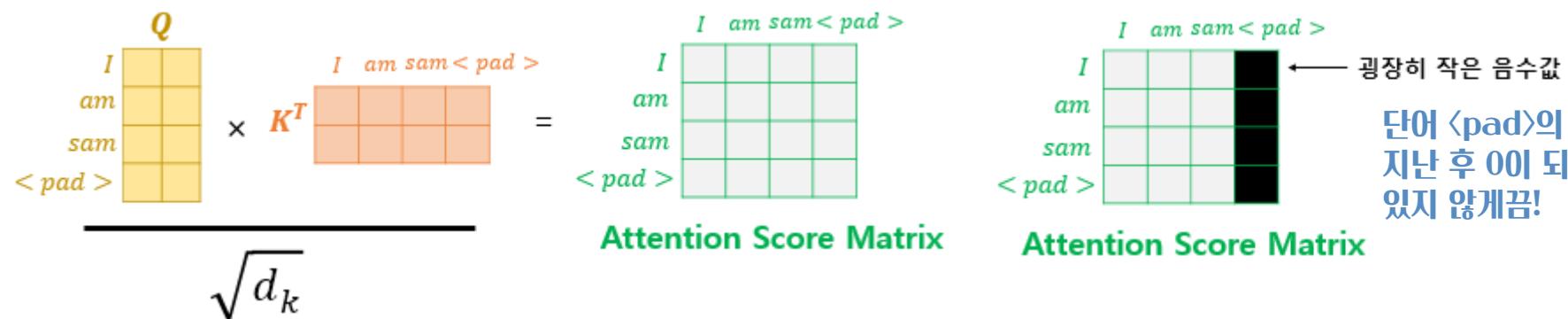
2. How Self Attention works

YONSEI Data Science Lab | DSL

(2) Self-attention 내부 동작

- (참고) 마스킹(Masking) 및 패딩 (Padding)

- 마스킹: 특정 값들을 가려서 실제 연산에 방해가 되지 않도록 하는 기법
- 패딩: 입력되는 문장의 길이를 동일하게 해주기 위해 0으로 채우는 기법
 - 0은 의미있는 값이 아니기 때문에 실제 어텐션 연산에서 제외해야 함 → 패딩 마스킹



2. How Self Attention works

YONSEI Data Science Lab | DSL

(2) Self-attention 내부 동작

- 쿼리의 셀프 어텐션 출력값 계산하기

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_K}}\right)\mathbf{V}$$

$$\begin{pmatrix} 0.13613 & 0.43194 & 0.43194 \\ 0.00089 & 0.90884 & 0.09027 \\ 0.00744 & 0.75471 & 0.23785 \end{pmatrix} \times \begin{pmatrix} 1 & 2 & 3 \\ 2 & 8 & 0 \\ 2 & 6 & 3 \end{pmatrix} = \begin{pmatrix} 1.8639 & 6.3194 & 1.7042 \\ 1.9991 & 7.8141 & 0.2735 \\ 1.9926 & 7.4796 & 0.7359 \end{pmatrix}$$

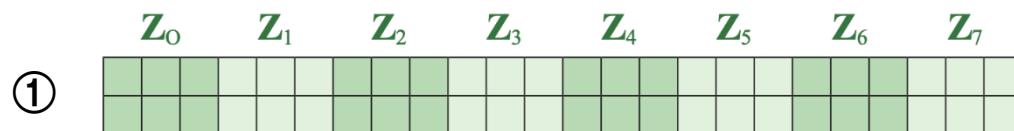
$\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_K}}\right)$

2. How Self Attention works

YONSEI Data Science Lab | DSL

(3) Multi-head attention

- 멀티 헤드 어텐션 = 셀프 어텐션을 여러 번 수행한 것
- 여러 헤드가 독자적으로 셀프 어텐션을 계산
 - 비유하자면 같은 문서(입력)를 두고 독자(헤드) 여러 명이 함께 읽는 구조라 할 수 있음
- 그림은 입력 단어 수 2개, 밸류의 차원수는 3, 헤드는 8개인 멀티 헤드 어텐션을 나타낸 것



② $\mathbf{Z} \times \mathbf{W}^0 = \mathbf{Z}'$

\mathbf{Z}

→ 동일한 입력에 대해 각각의 헤드가 분석한 결과의 총합

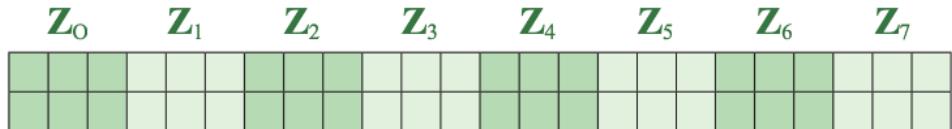
- \mathbf{W}^0 는 학습 과정에서 업데이트 됨

2. How Self Attention works

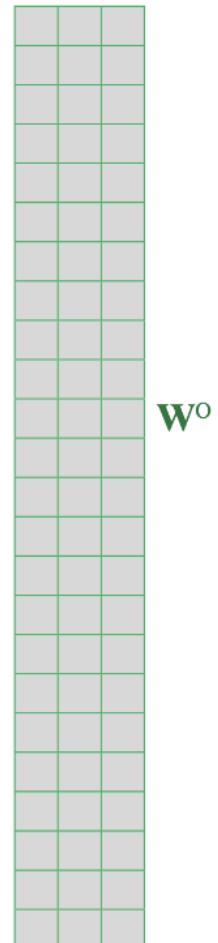
YONSEI Data Science Lab | DSL

(3) Multi-head attention

- ① 모든 헤드의 셀프 어텐션 출력 결과를 이어 붙인다.



- ② ①의 결과로 도출된 행렬에 \mathbf{W}^o 를 곱한다. 이 행렬은 개별 헤드의 셀프 어텐션 관련 다른 행렬($\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V$)과 마찬가지로 태스크(기계 번역)를 가장 잘 수행하는 방향으로 업데이트된다.



- ③ 새롭게 도출된 \mathbf{Z} 행렬은 동일한 입력(문서)에 대해 각각의 헤드가 분석한 결과의 총합이다.

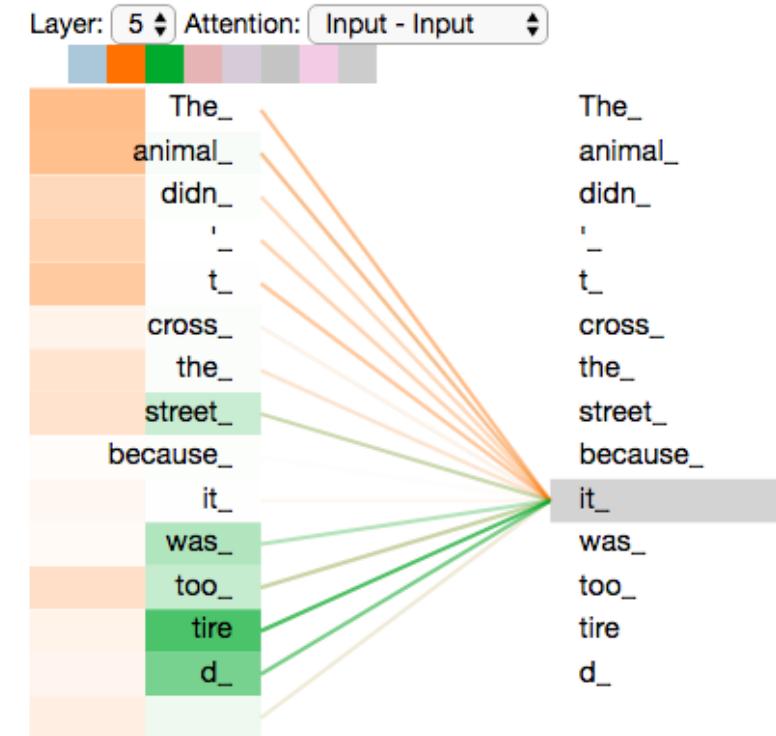
$$\mathbf{Z} = \begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}$$

2. How Self Attention works

YONSEI Data Science Lab | DSL

(3) Multi-head attention

- 그림을 보면 주황색 Attention Head는 Animal과 가장 큰 연관성을, 초록색 Attention Head는 Tired와 가장 큰 연관성을 가짐
 - 주황색 Attention Head는 "그것이 무엇인가"에 집중하여 관찰한 것이고, 초록색 Attention Head는 "그것이 어떤 상태인가?"에 집중하여 관찰한 것
- 여기서 Representation 공간이 넓어진다는 의미 파악 가능
 - 예를 들어 Attention Head가 하나였다면 Tired나 Animal 둘 중 하나에만 더 집중해서 봤을 것. 하지만, 이 2개 단어 모두 it에게는 중요한 단어이며, 중요성의 방향이 다를 뿐임
 - 즉, 서로 다른 방향성을 가지고 단어 간의 관계성을 파악하고, 이를 통해 중요한 2개 단어를 모두 활용할 수 있게 되는 것

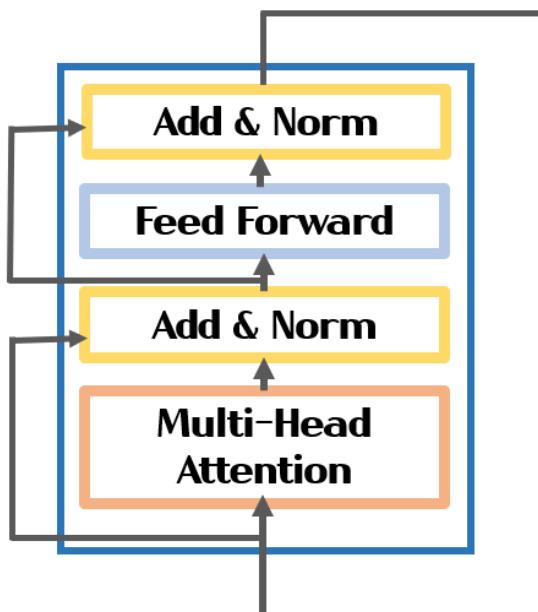


2. How Self Attention works

YONSEI Data Science Lab | DSL

(4) Encoder에서의 Self-Attention

- 트랜스포머 인코더에서 수행하는 계산 과정을 셀프 어텐션을 중심으로 살펴보자



- 인코더 블록의 입력은 이전 블록의 단어 벡터 시퀀스
- 출력은 이번 블록 수행 결과로 도출된 단어 벡터 시퀀스

2. How Self Attention works

YONSEI Data Science Lab | DSL

(4) Encoder에서의 Self-Attention

- 인코더에서 수행되는 셀프 어텐션은 쿼리, 키, 밸류가 모두 소스 시퀀스와 관련된 정보

Query	Key
어제	어제 0.1
카페	카페 0.1
갔었어	갔었어 0.3
거기	거기 0.1
사람	사람 0.1
많더라	많더라 0.3

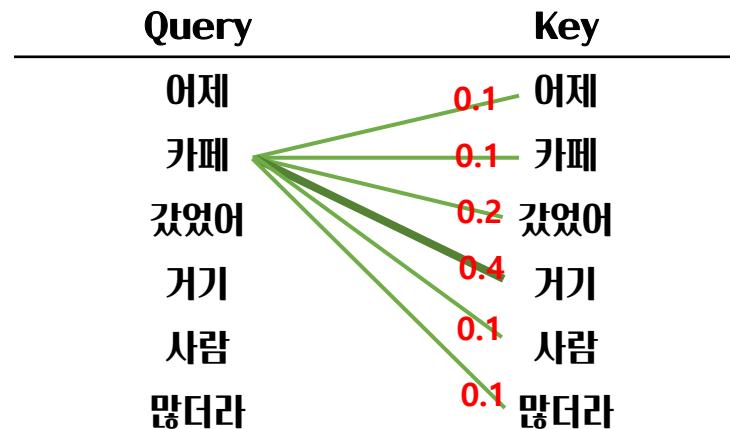
- 잘 학습되었다면, 쿼리가 ‘어제’인 경우 쿼리, 키로부터 계산한 소프트맥스 확률 가운데 과거 시제에 해당하는 갔었어, 많더라 등의 단어가 높은 값을 지닐 것
- 이 확률값들과 밸류 벡터를 가중합해서 셀프 어텐션 계산을 마침

2. How Self Attention works

YONSEI Data Science Lab | DSL

(4) Encoder에서의 Self-Attention

- 쿼리가 ‘카페’인 경우 장소를 지칭하는 대명사 ‘거기’가 높은 값을 지니게 될 것
- 이 같은 계산을 남은 단어들에 대해서도 수행
- 결국 인코더에서 수행하는 셀프 어텐션은 소스 시퀀스 내의 모든 단어 쌍(pair) 사이의 관계를 고려하게 됨

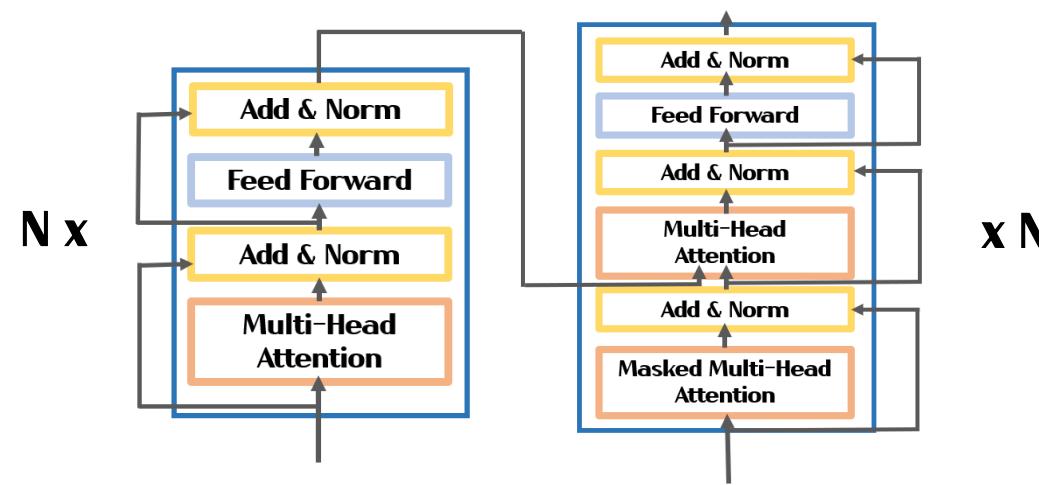


2. How Self Attention works

YONSEI Data Science Lab | DSL

(5) Decoder에서의 Self-Attention

- 아래 그림에서도 확인 가능하듯 디코더 입력은
 - 인코더 마지막 블록에서 나온 소스 단어 벡터 시퀀스
 - 이전 디코더 블록의 수행 결과로 도출된 타깃 단어 벡터 시퀀스
- 디코더에서 수행되는 셀프 어텐션을 순서대로 살펴보자

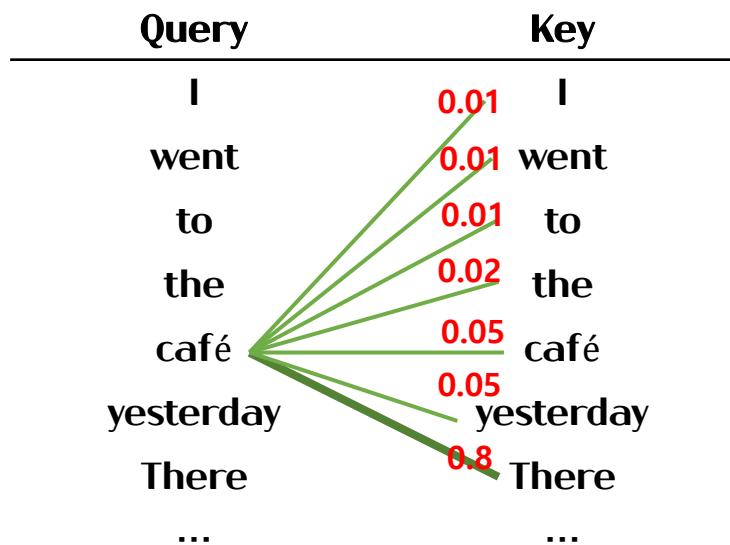


2. How Self Attention works

YONSEI Data Science Lab | DSL

(5) Decoder에서의 Self-Attention

- 우선 마스크 멀티 헤드 어텐션 (Masked Multi-head Attention)
- 이 모듈에서는 타깃 언어의 단어 벡터 시퀀스를 계산 대상으로 함
- 입력 시퀀스가 타깃 언어(영어)로 바뀌었을 뿐 인코더 쪽 셀프 어텐션과 크게 다를 바 없음



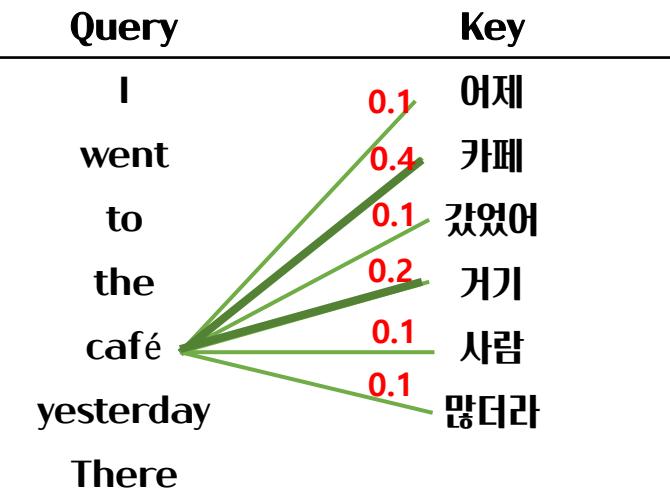
- 확률값들과 밸류 벡터를 가중합해서 셀프 어텐션 계산을 마침

2. How Self Attention works

YONSEI Data Science Lab | DSL

(5) Decoder에서의 Self-Attention

- 그 다음은 멀티 헤드 어텐션 (Masked Multi-head Attention) = Encoder-Decoder Attention
- 인코더와 디코더 쪽 정보를 모두 활용
 - 인코더에서 넘어온 정보는 소스 언어의 문장의 단어 벡터 시퀀스
 - 디코더에서 넘어온 정보는 타깃 언어의 문장의 단어 벡터 시퀀스
 - 전자를 키, 후자를 쿼리로 삼아 어텐션 계산을 수행
- 쿼리 단어가 ‘café’인 경우, 쿼리에 대응하는 해당 장소를 지칭하는 단어 ‘카페’가 높은 값을 지닐 것



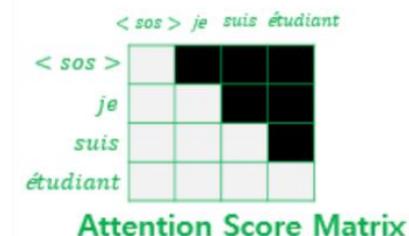
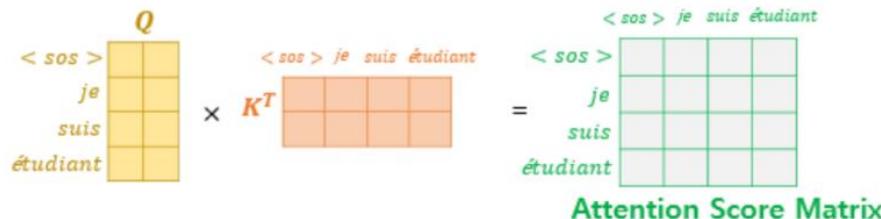
2. How Self Attention works

YONSEI Data Science Lab | DSL

(5) Decoder에서의 Self-Attention

- 그런데!! 학습 과정에서는 약간의 트릭을 씀
 - 트랜스포머 모델의 최종 출력은 타겟 시퀀스에 대한 확률 분포
 - 모델이 한국어를 영어로 번역하고 있다면 영어 문장의 다음 단어가 어떤 것이 적절할지에 관한 확률이 됨
 - 학습 과정에서 모델에 이번에 맞춰야 할 정답인 'I'를 알려주게 되면 학습하는 의미가 사라짐
- 따라서 정답을 포함한 미래 정보를 셀프 어텐션에서 제외하게 됨
- 이 때문에 디코더 블록의 첫 번째 어텐션을 마스크 멀티 헤드 어텐션이라고 부름

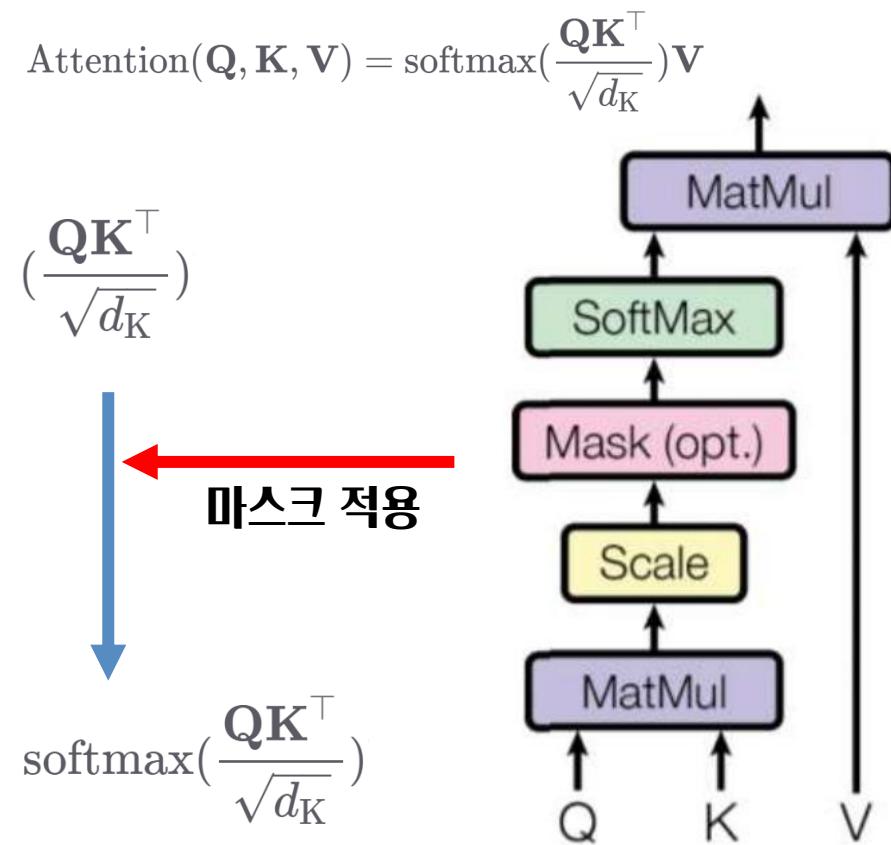
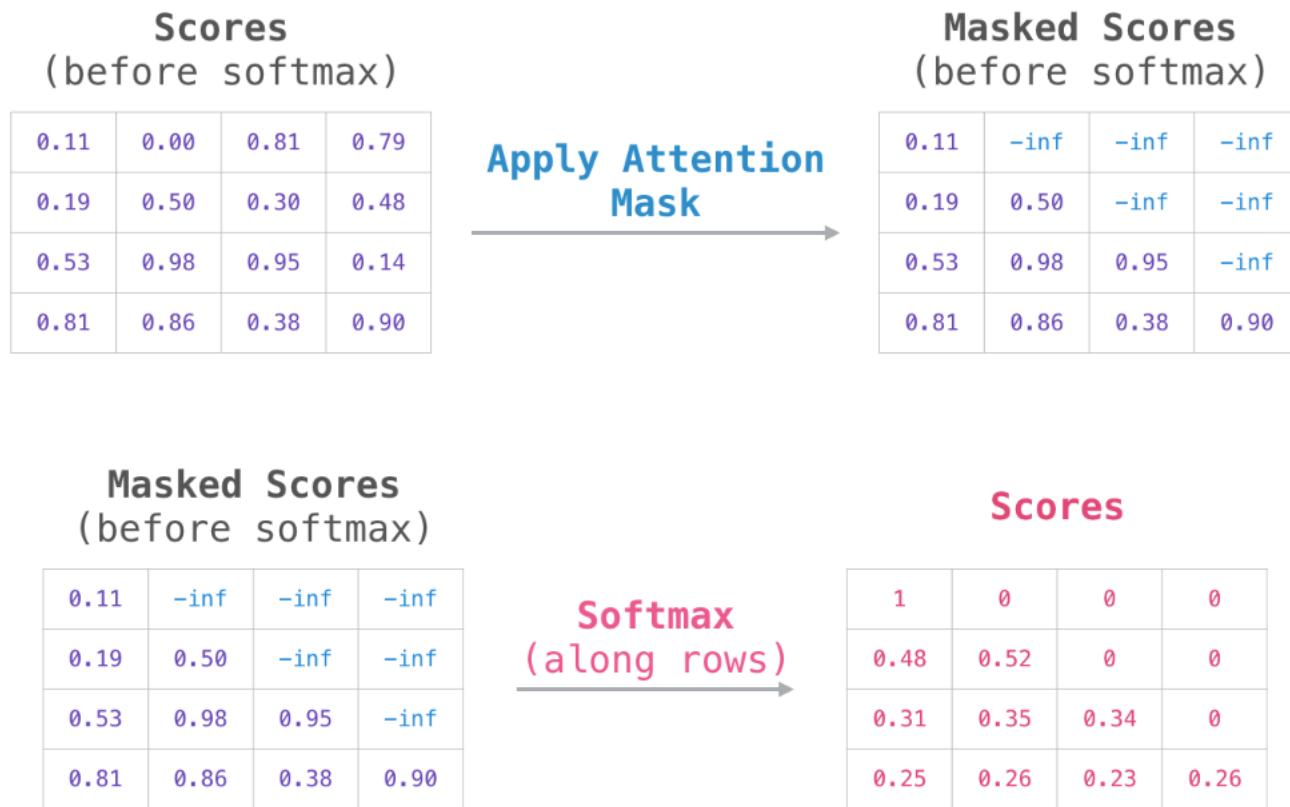
- 정확하게 표현하면 룩어헤드 마스킹
(앞의 패딩 마스킹과 차이)



2. How Self Attention works

YONSEI Data Science Lab | DSL

(5) Decoder에서의 Self-Attention



2. How Self Attention works

YONSEI Data Science Lab | DSL

(5) Decoder에서의 Self-Attention

Query	Key
$\langle s \rangle$	어제
I	카페
went	갔었어
to	거기
the	사람
café	많더라
...	

- 마스킹은 확률이 0이 되도록 하여, 밸류와의 가중합에서 해당 단어 정보들이 무시되게끔 하는 방식으로 수행
- 이처럼 셀프 어텐션을 수행하면 디코더 마지막 출력 벡터 가운데 $\langle s \rangle$ 에 해당하는 벡터에는 소스 문장 전체의 문맥적 관계성이 함축되어 있음
- 트랜스포머 모델은 이 $\langle s \rangle$ 벡터를 가지고 I를 맞추도록 학습

2. How Self Attention works

YONSEI Data Science Lab | DSL

(5) Decoder에서의 Self-Attention

Query	Key
⟨s⟩	어제
I	카페
went	갔었어
to	거기
the	사람
café	많더라
...	

masking



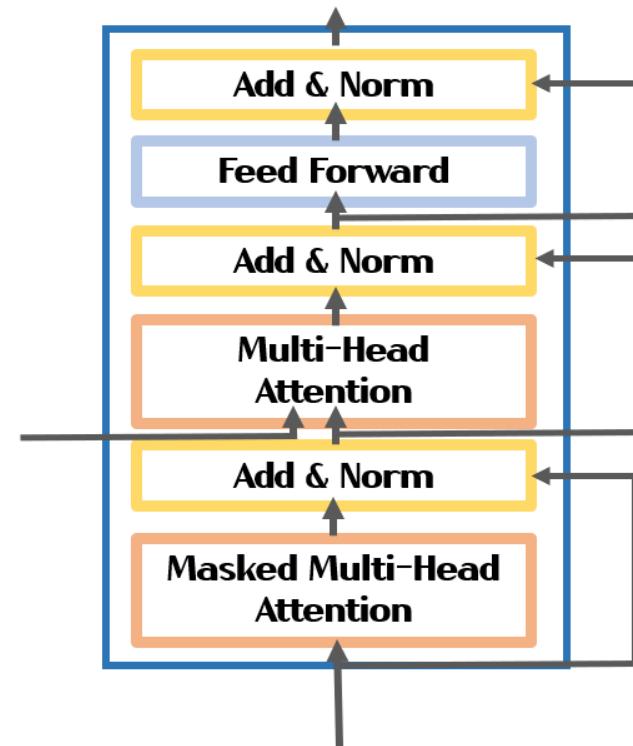
- 디코더에 <s> |가 입력된 경우에는 그림과 같이 마스킹
- 이 경우 디코더 마지막 블록의 | 벡터에는 소스 문장과 <s> | 사이의 문맥적 관계성이 녹아 있음
- 트랜스포머 모델은 이 | 벡터를 가지고 went를 맞추도록 학습
- 트랜스포머 모델은 이런 방식으로 말뭉치(corpus) 전체를 훑어가면서 반복 학습

2. How Self Attention works

YONSEI Data Science Lab | DSL

(5) Decoder에서의 Self-Attention

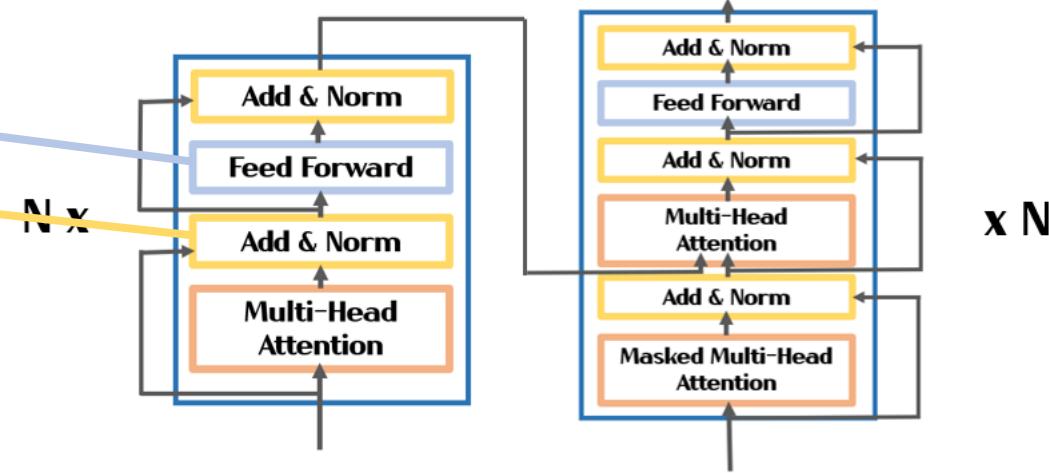
- 학습을 마친 모델은 다음처럼 기계 번역을 수행
 - 1) 소스 언어 문장을 인코더에 입력해 인코더 마지막 블록의 단어 벡터 시퀀스를 추출
 - 2) 인코더에서 넘어온 소스 언어 문장 정보와 디코더에 타깃 문장 시작을 알리는 스페셜 토큰 < s >를 넣어서, 타깃 언어의 첫 번째 토큰을 생성
 - 3) 인코더 쪽에서 넘어온 소스 언어 문장 정보와 이전에 생성된 타깃 언어 소스 시퀀스를 디코더에 넣어서 만든 정보로 타깃 언어의 다음 토큰을 생성
 - 4) 문장 끝을 알리는 스페셜 토큰 </s>가 나올 때까지 3)을 반복



(1) 트랜스포머 블록

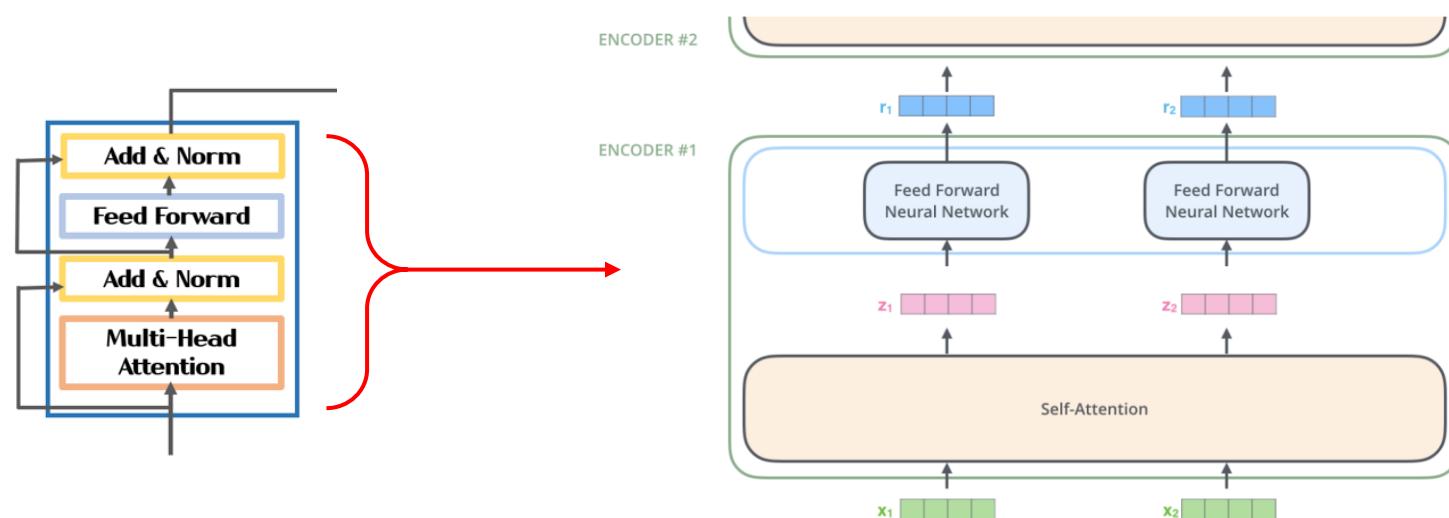
- 인코더와 디코더 블록의 구조는 디테일에서 차이가 있을 뿐 본질적으로 크게 다르지 않음
- 즉, 멀티 헤드 어텐션, 피드 포워드 뉴럴 네트워크, 잔차 연결 및 레이어 정규화 등 세가지 구성 요소를 기본으로 함
- 지금까지 멀티 헤드 어텐션 보았으니 이제 나머지 보자!

- 피드 포워드 뉴럴 네트워크
- 잔차 연결 및 레이어 정규화



(I) 트랜스포머 블록

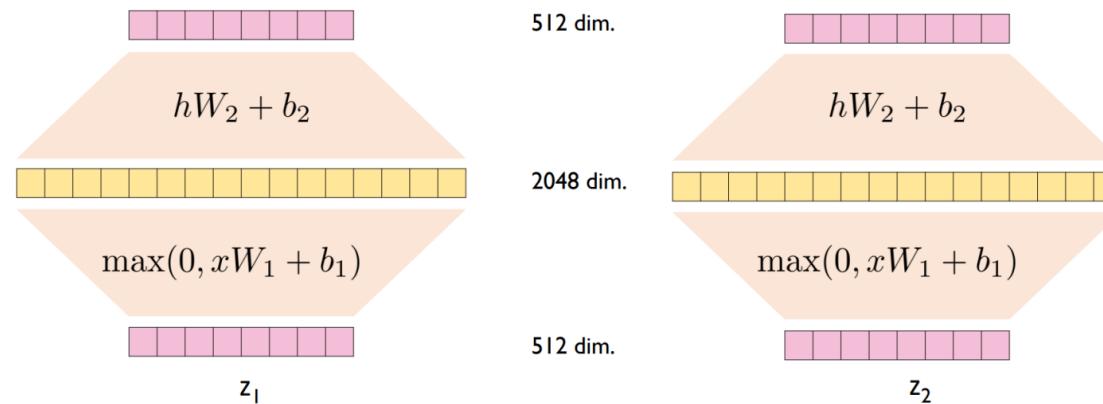
- Position-wise Feed Forward : position마다, 즉 개별 단어마다 적용되기 때문에 position-wise
$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \rightarrow \text{두 번의 선형 결합과 중간에 relu함수를 활성화 함수로 적용}$$
- MLP 모델로 작동하기 위해 비선형 함수 필요! 각각의 output에 feed forward network 추가함으로써 해결
- 피드 포워드 뉴럴 네트워크의 입력은 현재 블록의 멀티 헤드 어텐션의 개별 출력 벡터가 됨



(I) 트랜스포머 블록

- Position-wise Feed Forward: 피드 포워드 뉴럴 네트워크
- 피드 포워드 뉴럴 네트워크의 학습 대상은 가중치와 바이어스
- 트랜스포머에서는 은닉층의 뉴런 개수를 입력층의 4배로 설정
 - NN의 입력벡터가 512차원이라면 은닉층은 2048차원까지 늘렸다가 출력층에서 이를 다시 512차원으로 줄임

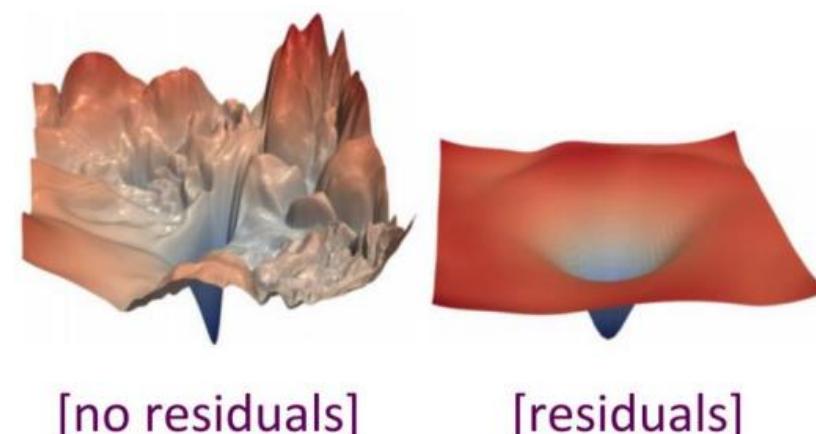
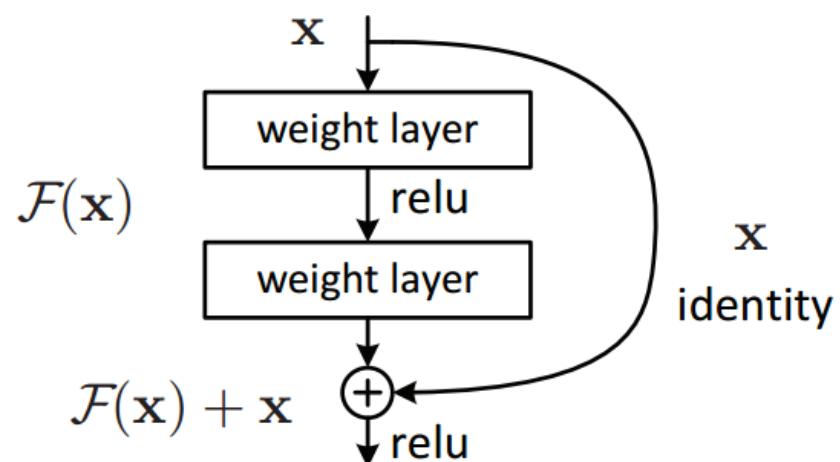
같은 레이어의 feed forward network는 같은 가중치를 공유!



- 인코더는 입력으로 벡터들의 리스트를 받고, 이 리스트를 먼저 self-attention layer에, 그 다음으로 feed-forward 에 통과시키고 그 결과물을 그 다음 인코더에게 전달

(I) 트랜스포머 블록

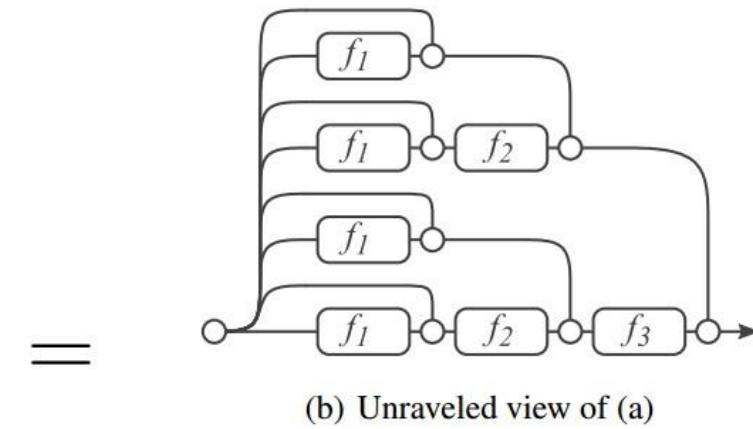
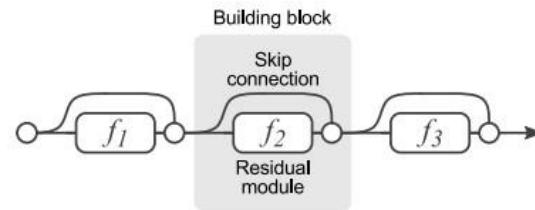
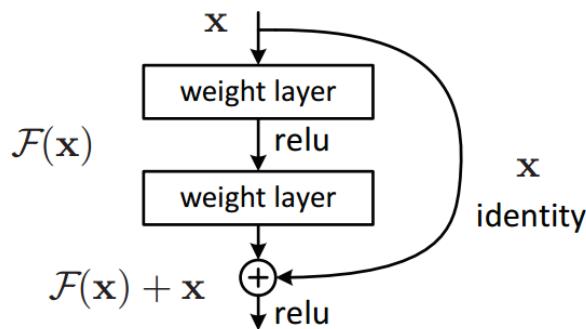
- Add: 잔차 연결
- 트랜스포머 블록의 Add는 잔차 연결을 가리킴
- 잔차 연결이란 아래 그림처럼 블록 계산을 건너뛰는 경로를 하나 두는 것
- 입력이 x , 이번 계산 대상 블록을 F 라고 할 때 잔차 연결은 $F(x) + x$ 로 간단히 표현



[Loss landscape visualization, Li et al., 2018, on a Resnet]

(I) 트랜스포머 블록

- Add: 잔차 연결



- (a)처럼 블록과 블록 사이에 잔차 연결을 모두 적용한다면, 사실 (b)처럼 계산하는 형태가 됨
- 잔차 연결을 블록마다 설정해둠으로써 모두 8가지의 새로운 경로가 생김
 - 즉, 모델이 다양한 관점에서 블록 계산을 수행하게 됨
 - 레이어가 깊어질수록 vanishing gradient가 발생하게 되지만, 잔차 연결은 모델 중간에 블록을 건너뛰는 경로를 설정함으로써 학습을 용이하게 하는 효과

(I) 트랜스포머 블록

- Norm: 레이어 정규화
- 레이어 정규화란 미니배치의 인스턴스(x) 별로 평균을 빼고 표준편차로 나누어 정규화를 수행하는 기법
- 학습 안정 & 속도 빨라짐

$$y = \frac{\mathbf{x} - \mathbb{E}[\mathbf{x}]}{\sqrt{\mathbb{V}[\mathbf{x}] + \epsilon}} * \gamma + \beta$$

- β, γ 는 학습을 통해 업데이트 됨
- ϵ 는 분모가 0이 되는 것을 방지 (보통 e^{-5} 로 설정)

- 배치 크기가 3인 경우 레이어 정규화의 수행 과정은 아래와 같음

배치			
	평균		표준편차
	1	2	3
	2	0.8164	이 값들을 바탕으로 위 수식 계산
	1	0	

(2) 모델 학습 기법

- Drop out: 드롭 아웃
 - 과적합 현상을 방지하고자 뉴런의 일부를 확률적으로 0으로 대치하여 계산에서 제외하는 기법
 - 트랜스포머 모델에서 드롭 아웃은 입력 임베딩과 최초 블록 사이, 마지막 블록과 출력층 사이에 적용
 - 학습 과정에만 적용!
-
- Adam optimizer: 아담 옵티마이저
 - 트랜스포머 모델의 최적화 도구
 - 그 외에도 널리 쓰임

(1) Limitation of CNN

- 합성곱 신경망과 비교
- CNN은 합성곱 필터(convolution filter)라는 특수한 장치를 이용해 시퀀스의 지역적인 특징을 잡아내는 모델
- 자연어는 기본적으로 시퀀스고 특정 단어 기준 주변 문맥이 의미 형성에 중요한 역할을 하므로 CNN이 자연어 처리에 사용되기도 함
- 그림은 CNN이 문장을 어떻게 인코딩하는지 나타낸 것
- CNN은 합성곱 필터 크기를 넘어서는 문맥은 읽어내기 어렵다는 단점이 존재

어제 카페 갔었어 거기 사람 많더라
어제 카페 갔었어 거기 사람 많더라
어제 카페 갔었어 거기 사람 많더라
어제 카페 갔었어 거기 사람 많더라

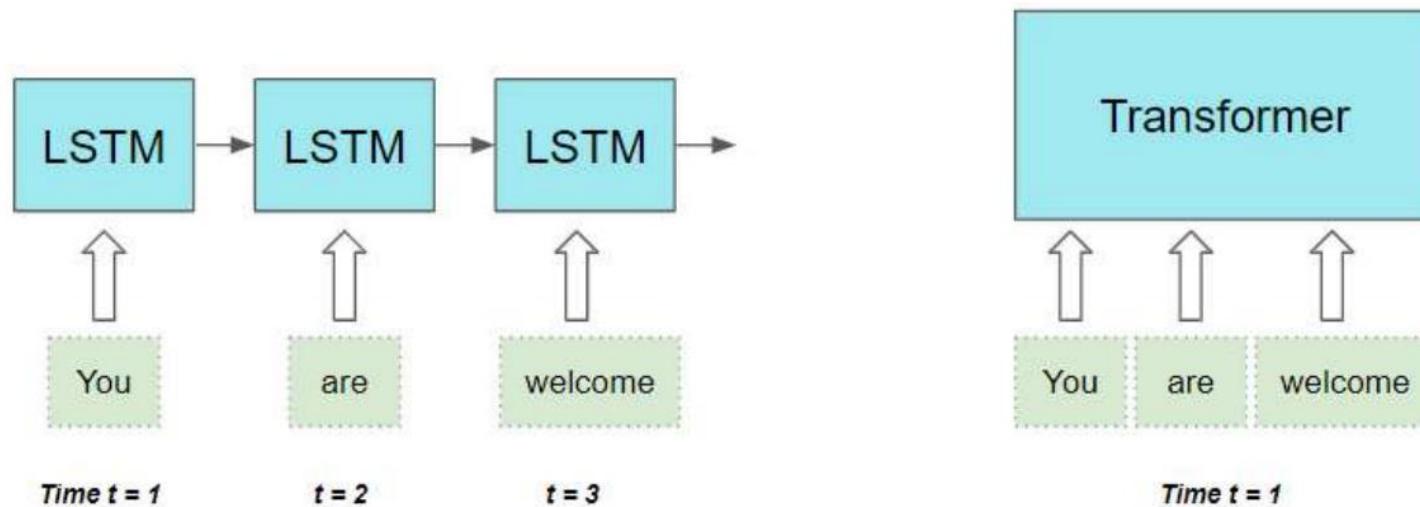
(2) Limitation of RNN

- 순환 신경망과 비교
- RNN은 소스 시퀀스를 차례대로 처리
- RNN은 시퀀스 길이가 길어질수록 정보 압축에 문제 발생
 - 오래 전에 입력된 단어는 잊어버리거나, 특정 단어 정보를 과도하게 반영해 전체 정보를 왜곡하는 경우 자주 발생
- 기계 번역 시 RNN을 사용한다면 인코더가 디코더로 넘기는 정보는 소스 시퀀스의 마지막인 ‘많더라’가 많이 반영될 수 밖에 없음

어제 → 카페 → 갔었어 → 거기 → 사람 → 많더라

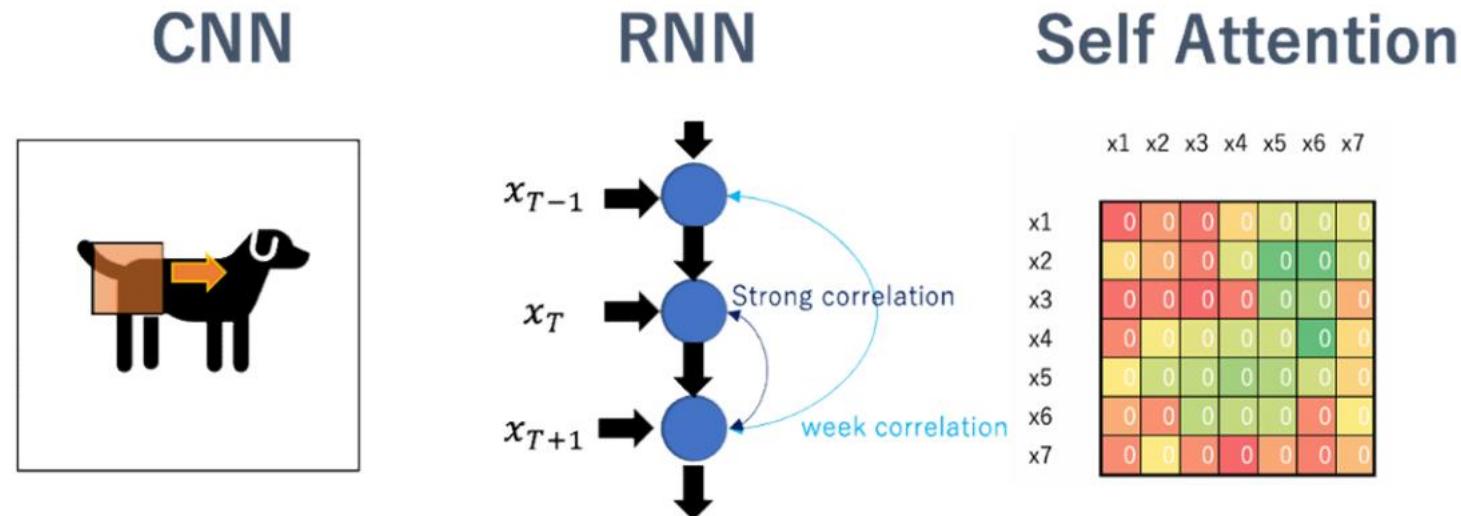
(3) CNN vs RNN vs Transformer

- 트랜스포머는 개별 단어와 전체 시퀀스 대상으로 어텐션 계산을 수행해 문맥 전체를 고려하기 때문에 지역적인 문맥만 고려하는 CNN 대비 강점 존재
- 아울러 모든 경우의 수를 고려(단어들이 서로 1대 1로 바라보게 함)하기 때문에 시퀀스 길이가 길어지더라도 정보를 잊거나 왜곡할 염려 없음 (+) 병렬 연산 가능. 이는 RNN의 단점을 극복한 지점



(4) Limitation of Transformer

- 트랜스포머의 계산 복잡도는 $O(n^2)$, n은 시퀀스 길이
- CNN, RNN에 비해 지역적인 특징을 잡아내는 것에 약함



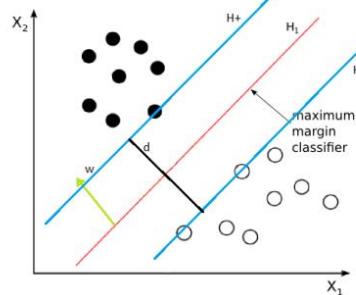
(Left) CNN, which has a strong inductive bias that the information is locally aggregated.

(Center) RNN, which has a strong inductive bias in that it is strongly correlated with the previous time.

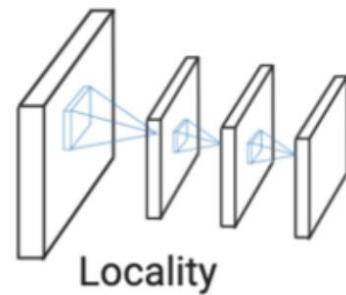
(Right) Self-Attention, which has a relatively weak inductive bias because it only correlates all features.

(4) Limitation of Transformer

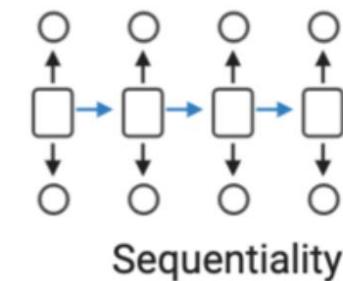
- (참고) Inductive bias
- Inductive bias : 새로운 데이터에 대해 좋은 성능을 내기 위해 모델에 사전적으로 주어지는 가정
- SVM: Margin 최대화 / CNN: 지역적인 정보 / RNN: 순차적인 정보
- Transformer: inductive bias↓, 모델의 자유도↑



SVM

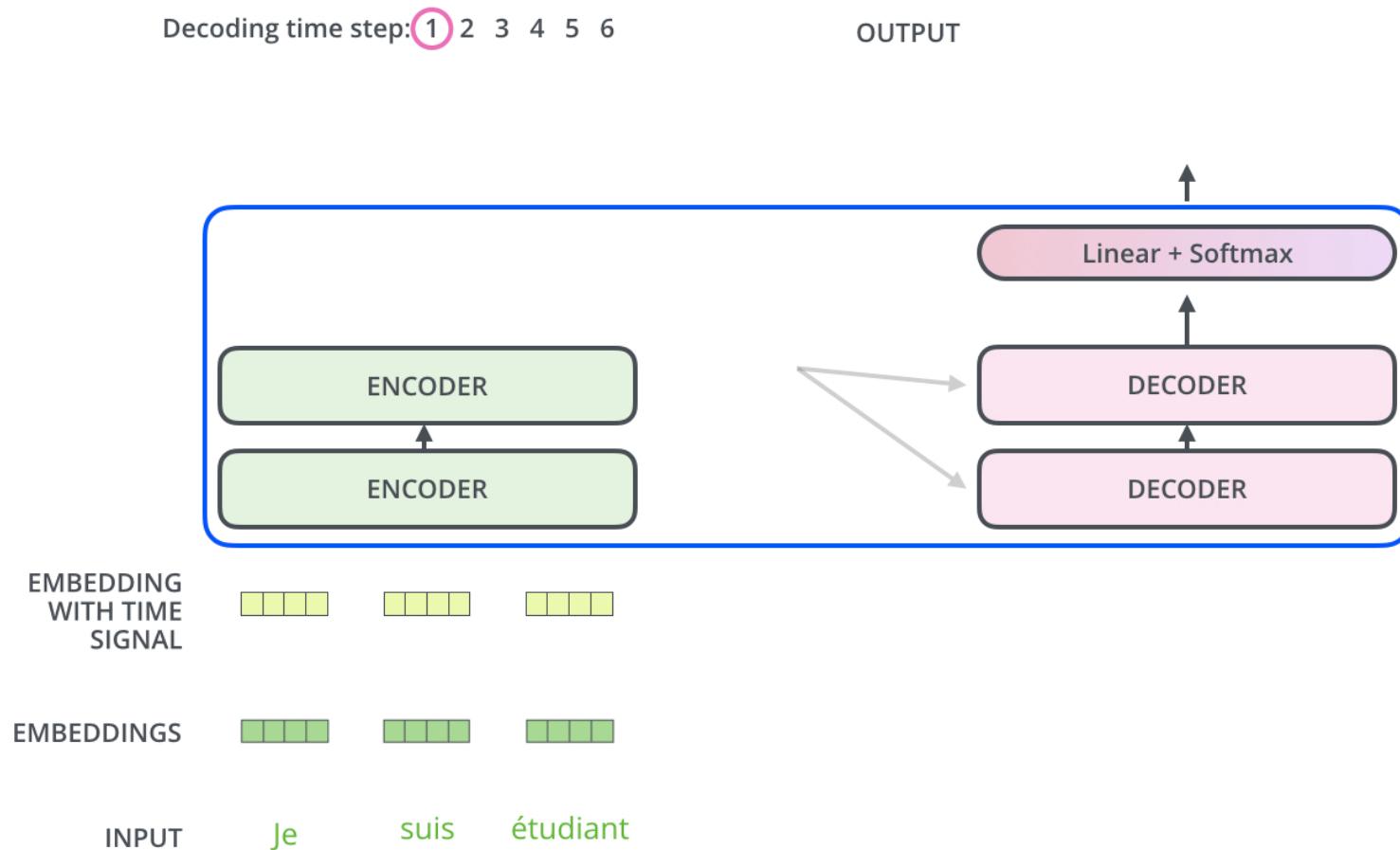


CNN

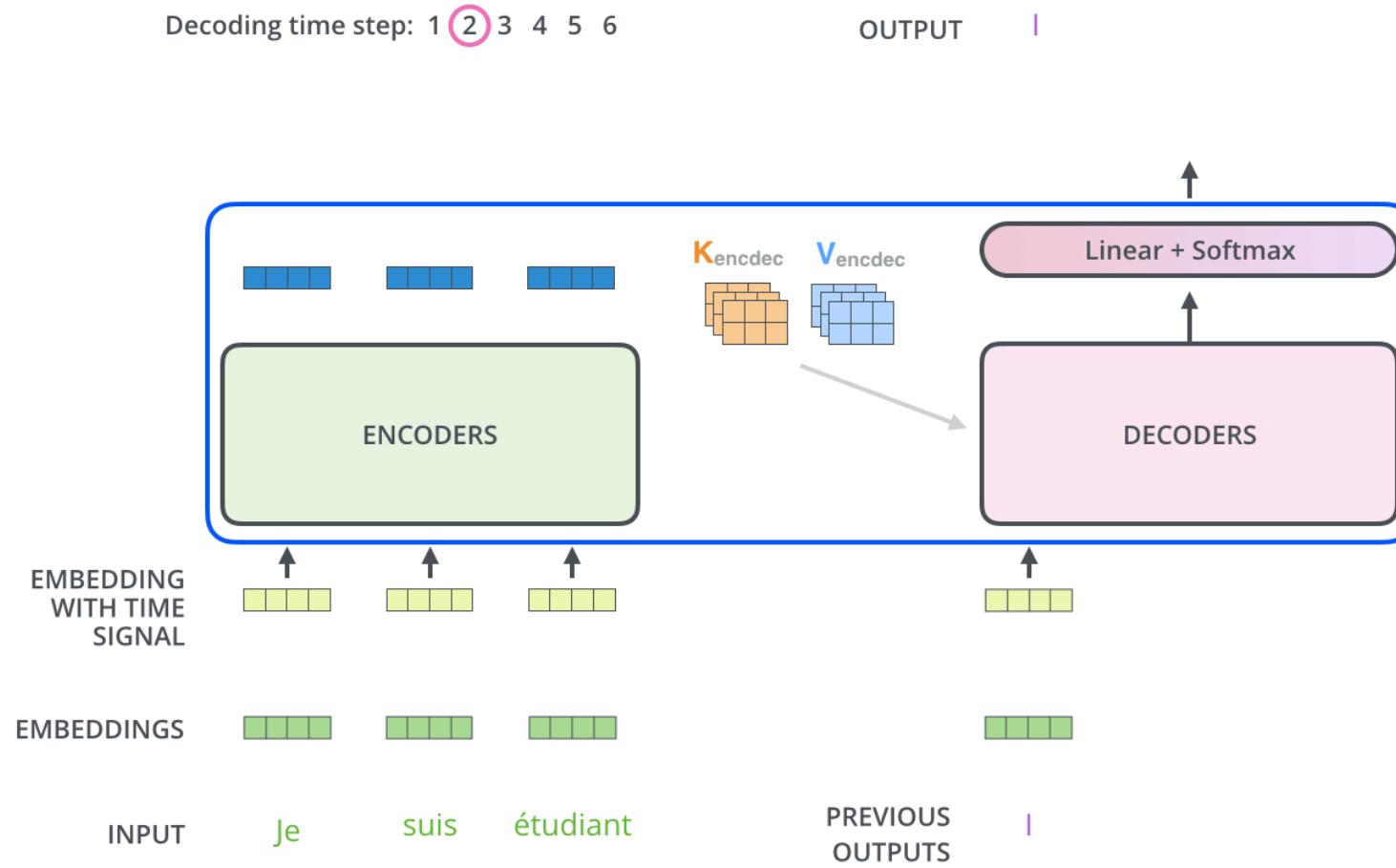


RNN

Transformer



Transformer



연세대학교 전기전자공학부 Andrew Beng Jin Teoh 교수님 “Neural Network” 강의안

고려대학교 산업경영공학부 DSBA 연구실 노건호님 “Self- Attention and Transformers” 강의안

고려대학교 산업경영공학부 DSBA 연구실 강현규님 “Visual Attention” 강의안

<https://web.eecs.umich.edu/~justincj/teaching/eecs498/WI2022/>

<http://web.stanford.edu/class/cs224n/>

<https://nlpinkorean.github.io/illustrated-transformer/>

<https://nlpinkorean.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/>

https://yngie-c.github.io/nlp/2020/07/01/nlp_transformer/