

NLP팀

2팀

박상훈
곽동길
박윤아
김수진
신민서

INDEX

1. Attention
2. Non-recurrent encoder
3. Transformer

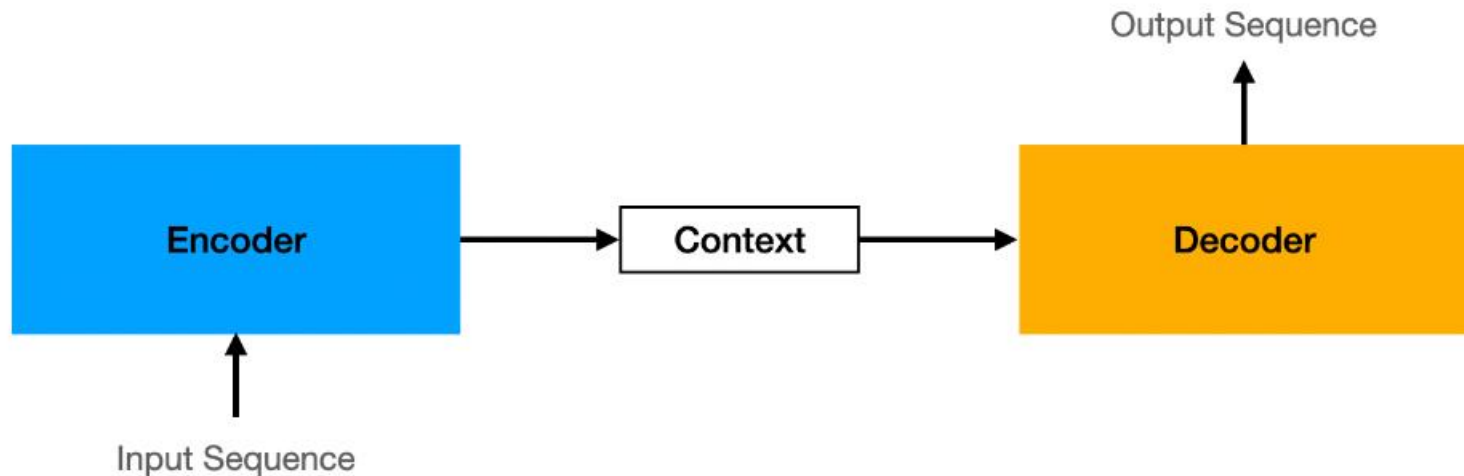
1

Attention

Seq2Seq

Seq2Seq

Sequence를 입력 받고 Sequence를 출력하는 모델로,
Encoder-Decoder 구조로 되어 있음



Seq2Seq의 한계점



정보의 손실

입력 문장 내 모든 시점에서의 정보를
하나의 context vector로 압축하는 과정에서 발생함

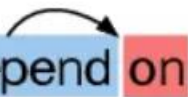


기울기 소멸 문제

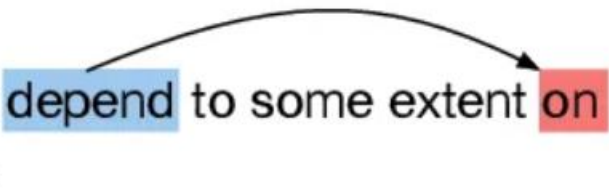
멀리 떨어진 두 시점의 토큰이 상호작용하기 위해
수많은 비선형 변환을 거치는 과정에서 발생함

Seq2Seq의 한계점

You may depend on the accuracy of the report.



The exact amounts spent depend to some extent on appropriations legislation.



The man who wore a Stetson on his head went inside.



세 번째 문장 같이 장기 의존성을 갖고 있는 경우 정확한 학습이 어려움

Bahdanau Attention

NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

Dzmitry Bahdanau

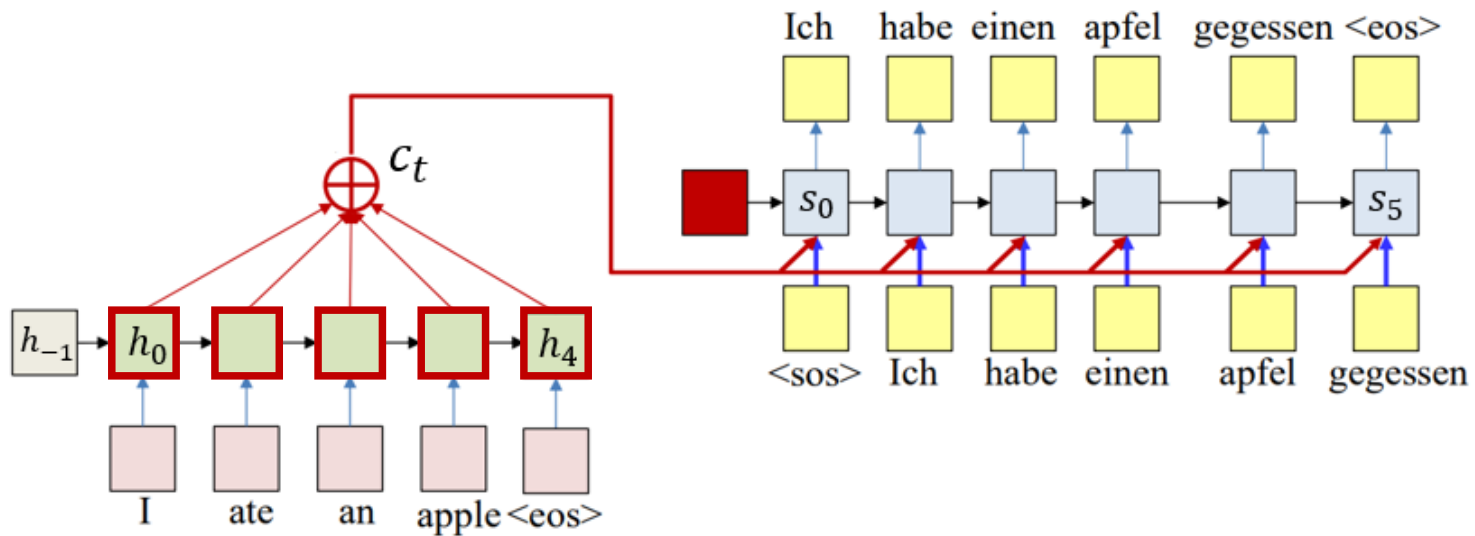
Jacobs University Bremen, Germany

KyungHyun Cho Yoshua Bengio*

Université de Montréal

정보의 손실을 줄이고자 Seq2seq에 추가된 새로운 요소가
바로 (Bahdanau) Attention mechanism

Bahdanau Attention



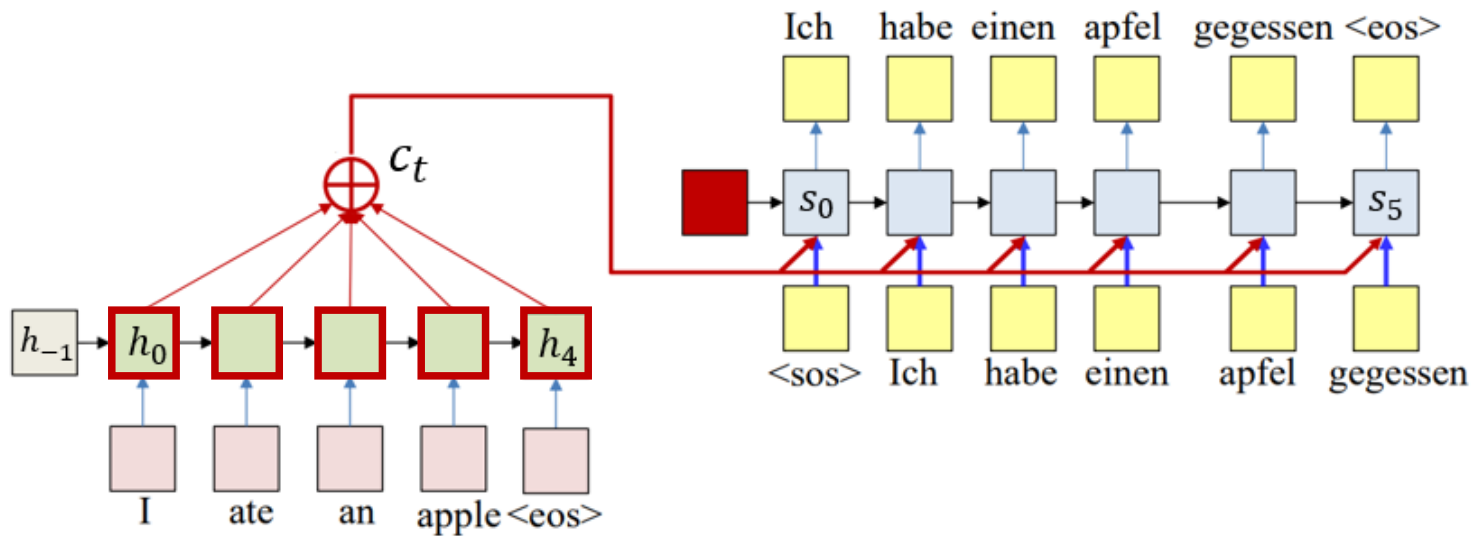
모든 시점의 정보가 압축된 하나의 벡터가 아닌

인코더의 hidden state h_t 전부를 직접 참조해 Context를 생성함

1

Attention

Bahdanau Attention

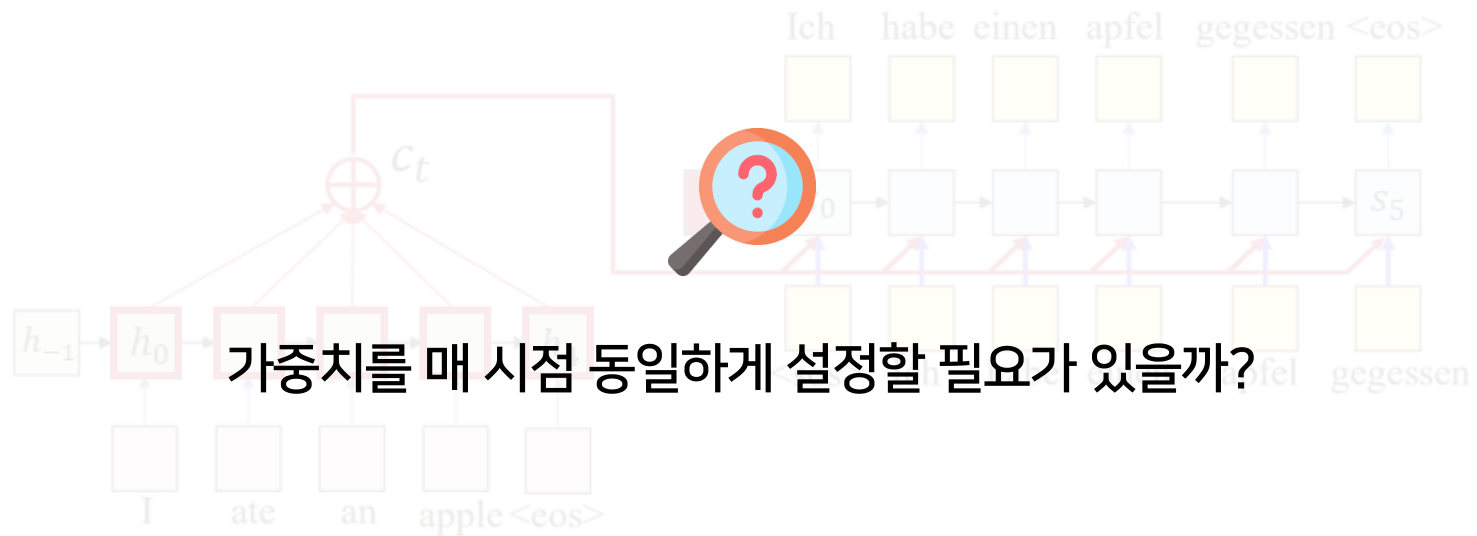


디코딩의 매 시점마다 $h_i (i = 1, \dots, N)$ 들의 가중합을 계산해 c_t 로 사용

1

Attention

Bahdanau Attention



디코딩의 매 시점마다 $h_i (i = 1, \dots, N)$ 들의 가중합을 계산해 c_t 로 사용

Dynamic context vector

디코딩의 각 시점마다 h_i 에 주는 가중치가 달라질 수 있어야 함



Context vector를 $c_t = \sum_{i=1}^N w_i(t)h_i$ 로 모델링

디코딩 시점의 인덱스 : t

인코딩 시점의 인덱스 : i

입력 문장의 길이 : N



가중치가 더 높게 계산된 토큰에 더 집중(attention)할 수 있게 함

Dynamic context vector

디코딩의 각 시점마다 h_i 에 주는 가중치가 달라질 수 있어야 함



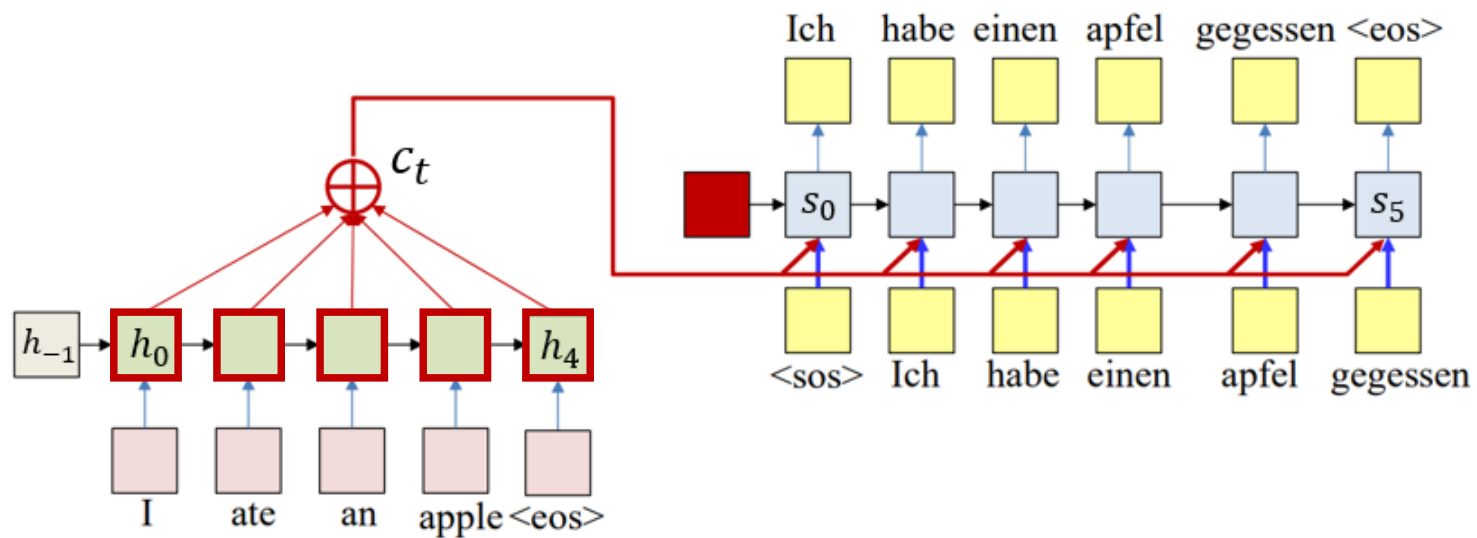
이때 가중치들은 규칙 기반 시스템이 아닌
데이터를 통한 학습으로 결정될 수 있도록 함

가중치가 더 높게 계산된 토큰에 더 집중(attention)할 수 있게 함

1

Attention

Dynamic context vector

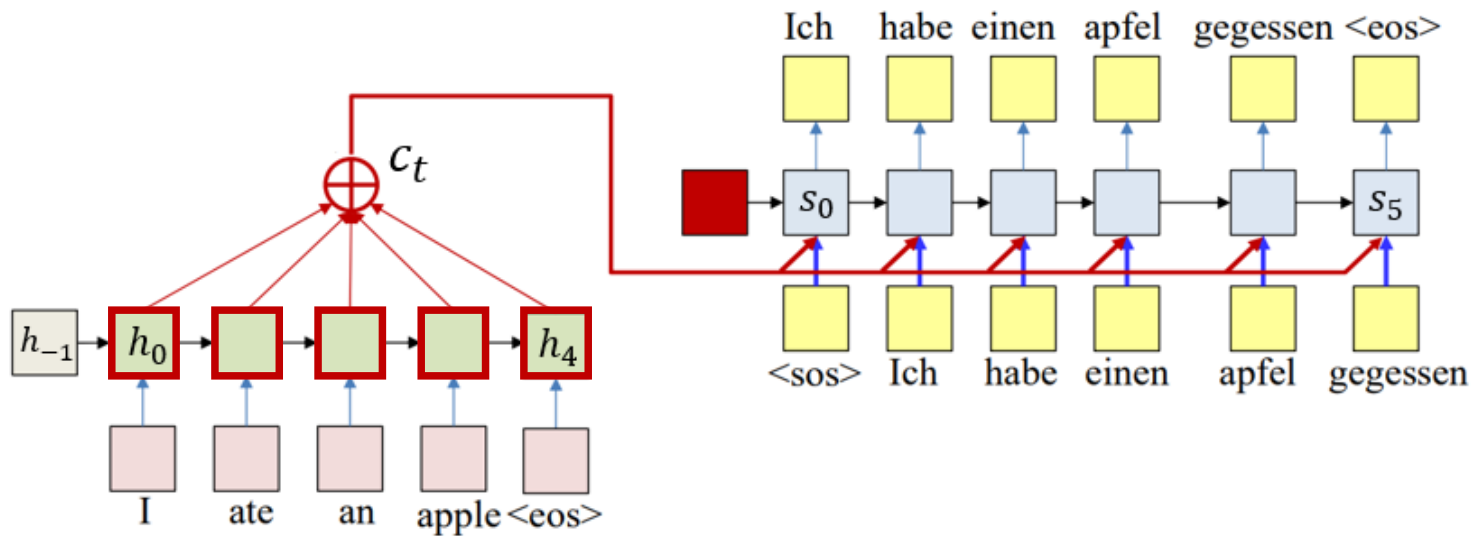


c_t 는 이제 **동적인 context vector**가 되고
 디코더의 hidden state s_t 를 계산하는 데 사용됨

1

Attention

Dynamic context vector



토큰 추출을 위한 확률 벡터 y_t 의 전체적인 계산 구조는

$y_t = g(s_t)$, $s_t = f(s_{t-1}, x_t, c_t)$ 로 기존 Seq2seq와 유사함

Query, Key, Value

Context c_t 를 계산하기 위해 Weight $w_i(t)$ 의 보편적인 계산 방식을 정의할 필요가 있고 아래 세 가지의 벡터를 우선 정의

Query

디코딩 시점 t 의
query 벡터

$$q_t = W_q s_{t-1}$$

Key

인코딩 시점 i 의
Key 벡터

$$k_i = W_k h_i$$

Value

인코딩 시점 i 의
value 벡터

$$v_i = W_v h_i$$

Query, Key, Value

Context c_t 를 계산하기 위해 Weight $w_i(t)$ 의 **보편적인 계산 방식**을 정의할 필요가 있고 아래 세 가지의 벡터를 우선 정의

Query

Key

Value

디코딩 시점 t 각 Hidden state h_i 에 적절한 $w_i(t)$ 를 부여하는 데 디코딩 시점 i 의

query 벡터 왜 Query, Key, Value 용어가 등장하는지 value 벡터

$q_t = W_q s_{t-1}$ 데이터베이스 시스템을 비유로 들어 이해해보자! $v_i = W_v h_i$

Query, Key, Value

{Query: "Order details of order_104"}

OR

{Query: "Order details of order_106"}

```
{ "order_100": { "items": "a1", "delivery_date": "a2", ... } },  
{ "order_101": { "items": "b1", "delivery_date": "b2", ... } },  
{ "order_102": { "items": "c1", "delivery_date": "c2", ... } },  
{ "order_103": { "items": "d1", "delivery_date": "d2", ... } },  
{ "order_104": { "items": "e1", "delivery_date": "e2", ... } },  
{ "order_105": { "items": "f1", "delivery_date": "f2", ... } },  
{ "order_106": { "items": "g1", "delivery_date": "g2", ... } },  
{ "order_107": { "items": "h1", "delivery_date": "h2", ... } },  
{ "order_108": { "items": "i1", "delivery_date": "i2", ... } },  
{ "order_109": { "items": "j1", "delivery_date": "j2", ... } },  
{ "order_110": { "items": "k1", "delivery_date": "k2", ... } }
```

시스템은 사용자로부터 Query를 받아 데이터베이스 내부 key와 매칭하며
적절한 value를 반환함

Query, Key, Value

{Query: "Order details of order_104"}

OR

{Query: "Order details of order_106"}

```
{
  "order_100": {
    "items": "a1",
    "delivery_date": "a2",
    ...
  },
  "order_101": {
    "items": "b1",
    "delivery_date": "b2",
    ...
  },
  "order_102": {
    "items": "c1",
    "delivery_date": "c2",
    ...
  },
  "order_103": {
    "items": "d1",
    "delivery_date": "d2",
    ...
  },
  "order_104": {
    "items": "e1",
    "delivery_date": "e2",
    ...
  },
  "order_105": {
    "items": "f1",
    "delivery_date": "f2",
    ...
  },
  "order_106": {
    "items": "g1",
    "delivery_date": "g2",
    ...
  },
  "order_107": {
    "items": "h1",
    "delivery_date": "h2",
    ...
  },
  "order_108": {
    "items": "i1",
    "delivery_date": "i2",
    ...
  },
  "order_109": {
    "items": "j1",
    "delivery_date": "j2",
    ...
  },
  "order_110": {
    "items": "k1",
    "delivery_date": "k2",
    ...
  }
}
```

각 Hidden state h_i 에 대한 적절한 가중치를 찾는 과정을

어떤 h_i 에 집중해야 하는지를 묻는 질의 q_t 를 시스템에게 보내는 것처럼

생각할 수 있음

Query, Key, Value

{Query: "Order details of order_104"}

OR

{Query: "Order details of order_106"}

```
{ "order_100": { "items": "a1", "delivery_date": "a2", ... },  
  { "order_101": { "items": "b1", "delivery_date": "b2", ... },  
  { "order_102": { "items": "c1", "delivery_date": "c2", ... },  
  { "order_103": { "items": "d1", "delivery_date": "d2", ... },  
  { "order_104": { "items": "e1", "delivery_date": "e2", ... },  
  { "order_105": { "items": "f1", "delivery_date": "f2", ... },  
  { "order_106": { "items": "g1", "delivery_date": "g2", ... },  
  { "order_107": { "items": "h1", "delivery_date": "h2", ... },  
  { "order_108": { "items": "i1", "delivery_date": "i2", ... },  
  { "order_109": { "items": "j1", "delivery_date": "j2", ... },  
  { "order_110": { "items": "k1", "delivery_date": "k2", ... }
```

시스템은 q_t 를 가지고, 데이터베이스 내부의 모든 키 k_i 에 대해 연산 수행



사용자에게 반환할 결과값 v_i 를 결정함

Query, Key, Value

Value v_i 가 반환된다는 것은 v_i 에는 1, 나머지에는 0의 가중치를 배정한 후
모두 더해서 반환하는 것으로 생각할 수 있음



이러한 가중치의 이산적인 배분을 **연속적인 과정**으로 확장 가능
실수 값을 갖는 유사도 $\alpha(q_t, k_i)$ 를 계산하여 최종 결과 c_t 를 반환

$$c_t = \sum_{i=1}^N w_i(t) v_i, \quad w_i(t) = \frac{\exp(\alpha(q_t, k_i))}{\sum_{j=1}^N \exp(\alpha(q_t, k_j))}$$

$w_i(t)$ 를 위와 같이 정의할 경우 비음의 값을 갖고 합이 1인 유용한 성질을 갖게 됨

Query, Key, Value

Value v_i 가 반환된다는 것은 v_i 에는 1, 나머지에는 0의 가중치를 배정한 후

모두 더해서 반환하는 것으로 생각할 수 있음

트랜스포머 모델에서는 대표적으로

scaled dot product 기반 유사도 $\alpha(q_t, k_i)$ 계산을 진행

이러한 가중치의 이산적인 배분을 **연속적인 과정**으로 확장 가능

실수 값을 갖는 유사도 $\alpha(q_t, k_i)$ 를 계산하여 최종 결과를 반환

$$\alpha(q_t, k_i) = \frac{q_t^T k_i}{\sqrt{d_k}}$$

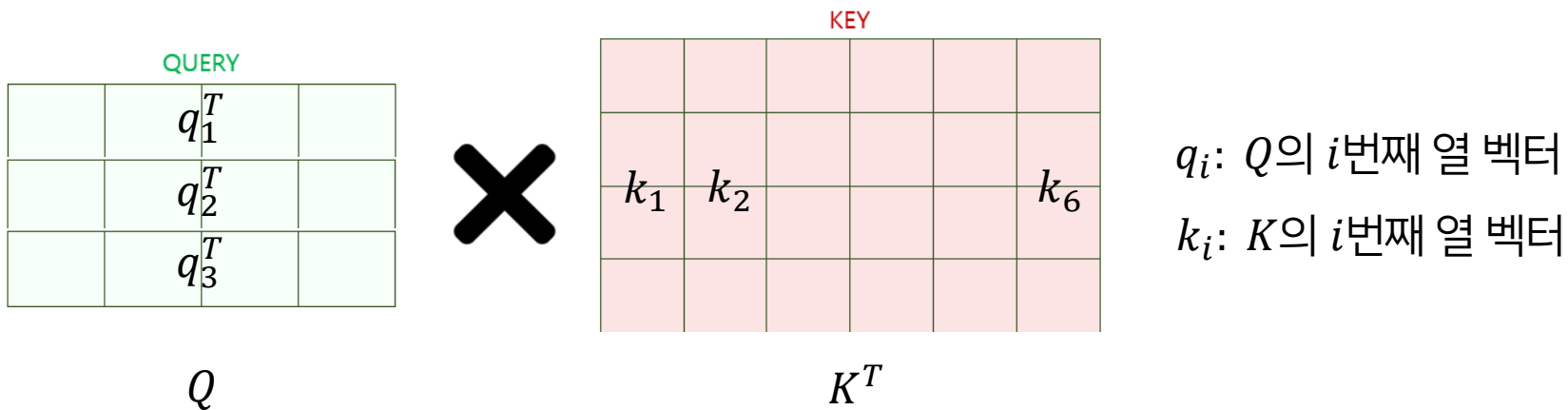
$$c_t = \sum_{i=1}^N w_i(t) v_i, \quad w_i(t) = \frac{\exp(\alpha(q_t, k_i))}{\sum_{j=1}^N \exp(\alpha(q_t, k_j))}$$

$w_i(t)$ 를 위와 같이 정의할 경우 비음의 값을 갖고 합이 1인 유용한 성질을 갖게 됨

Attention 연산

QK^T - 유사도 계산

Query matrix Q 와 Key matrix인 K 에 대해서 행렬 곱 수행



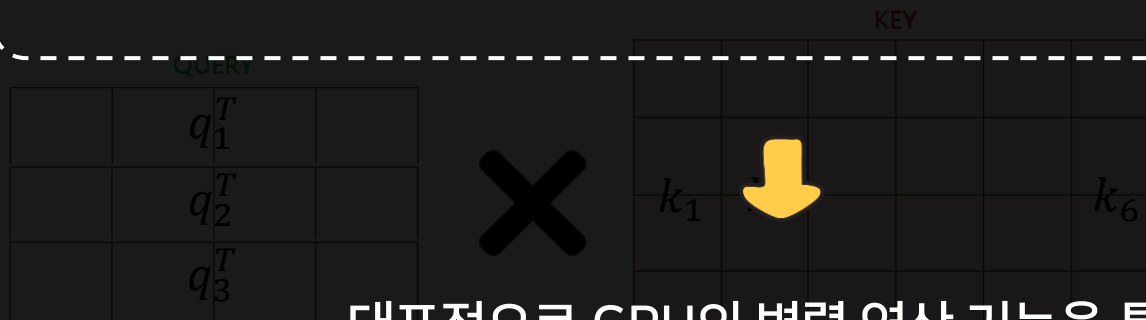
Attention 연산



QK^T - 유사도 행렬 계산 왜 벡터 연산이 아닌 행렬 곱을 사용할까?

Query matrix Q 와 Key matrix인 K 에 대해서 행렬 곱 수행

행렬 곱 또한 벡터 연산의 연장선이며 내적 연산을
쿼리 하나가 아닌 **모든 쿼리 벡터**에 동시 적용 가능하기 때문



q_i^T : Q 의 i 번째 행 벡터

k_i^T : K 의 i 번째 행 벡터

대표적으로 GPU의 병렬 연산 기능을 통해
추가적으로 계산의 효율성을 높일 수 있음

Attention 연산

QK^T - 유사도 계산

쿼리 하나에 대해서만 계산하는 예제를
영한 번역 예시와 함께 알아보자!



Attention 연산

*He **left** his right shoe at home / 그는 그의 오른쪽 신발을 집에 **두고** 왔다*

*Turn **left** at the next intersection / 다음 교차로에서 **왼쪽으로** 돌아라*

첫 번째 문장의 *left*는 '놓고 오다'의 'leave'의 과거형

두 번째 문장의 *left*는 '왼쪽'이라는 '방향'을 나타냄

Attention 연산

*He **left** his right shoe at home / 그는 그의 오른쪽 신발을 집에 **두고** 왔다*

*Turn **left** at the next intersection / 다음 교차로에서 **왼쪽으로** 돌아라*

첫 번째 문장의 query로 “**두고**”, 두 번째 문장의 query로 “**왼쪽으로**”가 쓰였다고 가정

Attention 연산

*He **left** his right shoe at home / 그는 그의 오른쪽 신발을 집에 **두고** 왔다*

*Turn **left** at the next intersection / 다음 교차로에서 **왼쪽으로** 돌아라*

<Ex>

첫 번째 문장의 query로 “**두고**”

두 번째 문장의 query를 “**왼쪽으로**”가 쓰였다고 가정

“**두고**”라는 토큰을 생성해야 하는 시점에

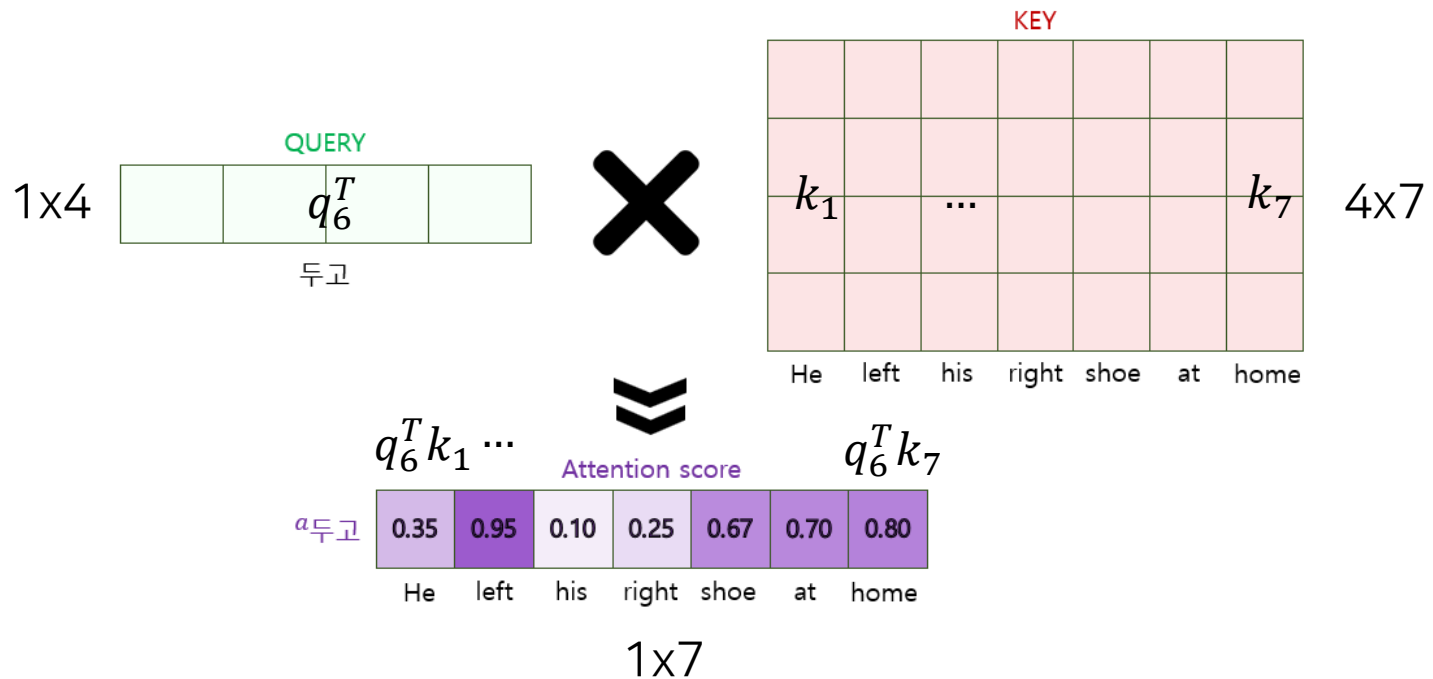
인코더 쪽 입력 문장 속 토큰들에게 어느 정도로 집중해야 하는지를 질의

1

Attention

Attention 연산

S1 : 그는 그의 오른쪽 신발을 집에 두고 왔다

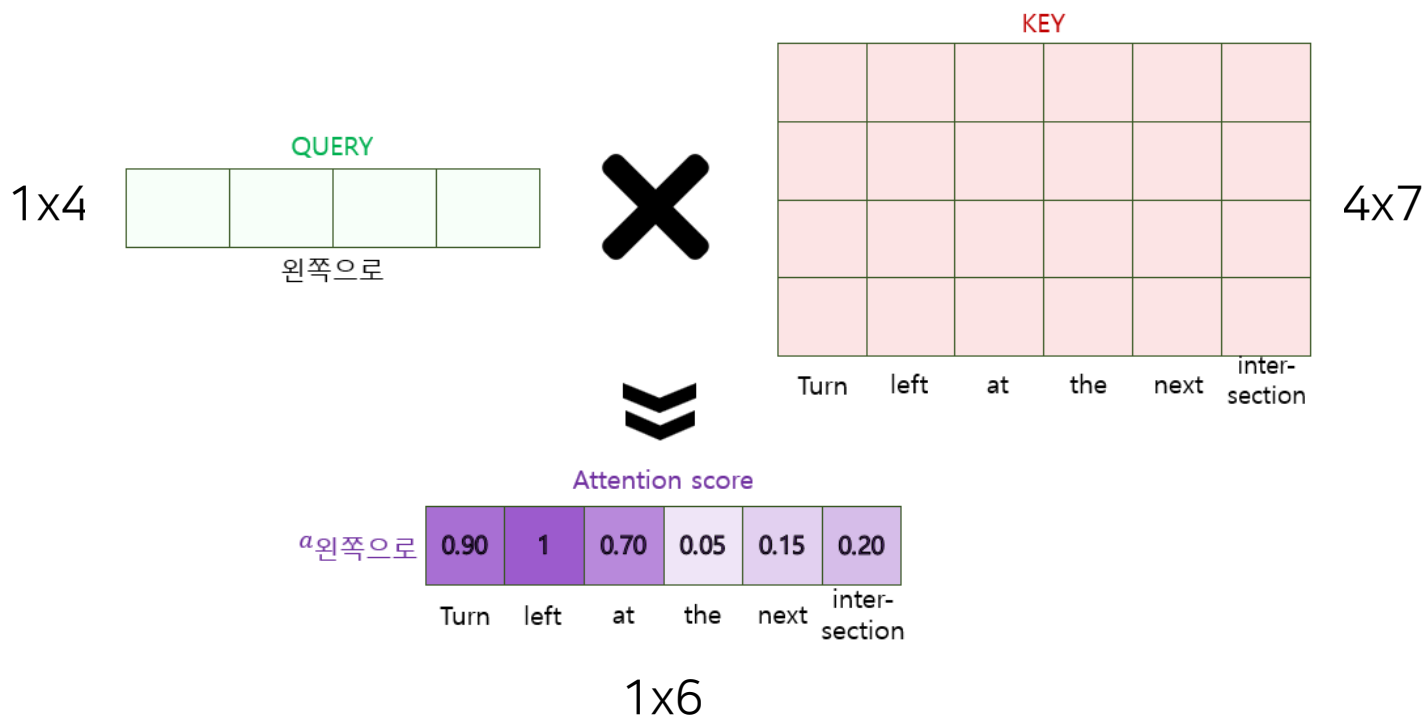


1

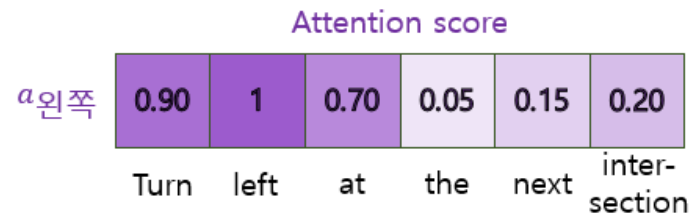
Attention

Attention 연산

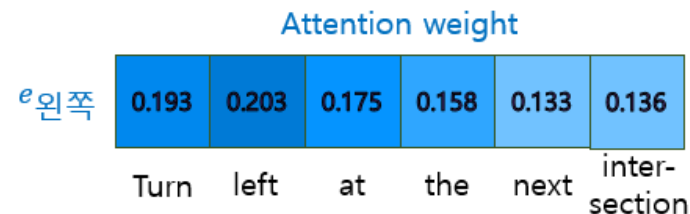
S2 : 다음 교차로에서 **왼쪽으로** 돌아라



Attention 연산



key 벡터의 차원 수 d_k 의 제곱근의
역수로 스케일링



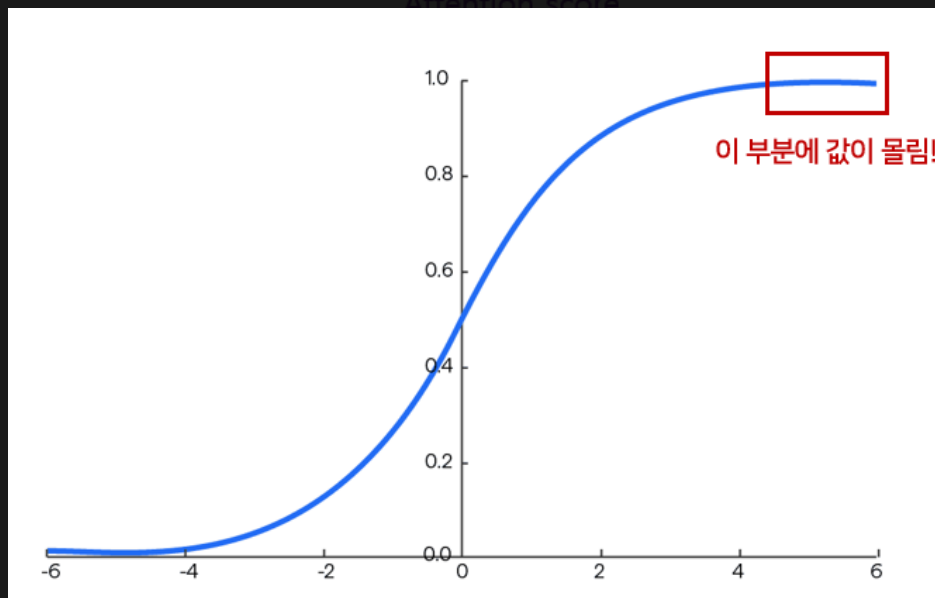
$\sqrt{d_k}$ 로 나눠준 후 softmax 함수를 적용해
(0,1) 사이의 값으로 바뀌어서 Attention weight를 구함

1

Attention



Attention 연산 - $\text{softmax}()$ 와 $\sqrt{d_k}$ 표준화
 왜 $\sqrt{d_k}$ 로 나누는 스케일링이 필요할까?



Key 벡터의 차원이 크다면 QK^T 의 각 성분이 큰 분산을 갖게 되고

이로 인해 Attention score의 분포가 한쪽으로 쏠리는 것을 방지

Attention 연산

유사도 계산과 표준화를 통해 Attention weight,
각 토큰에게 집중하는 정도를 구할 수 있었음



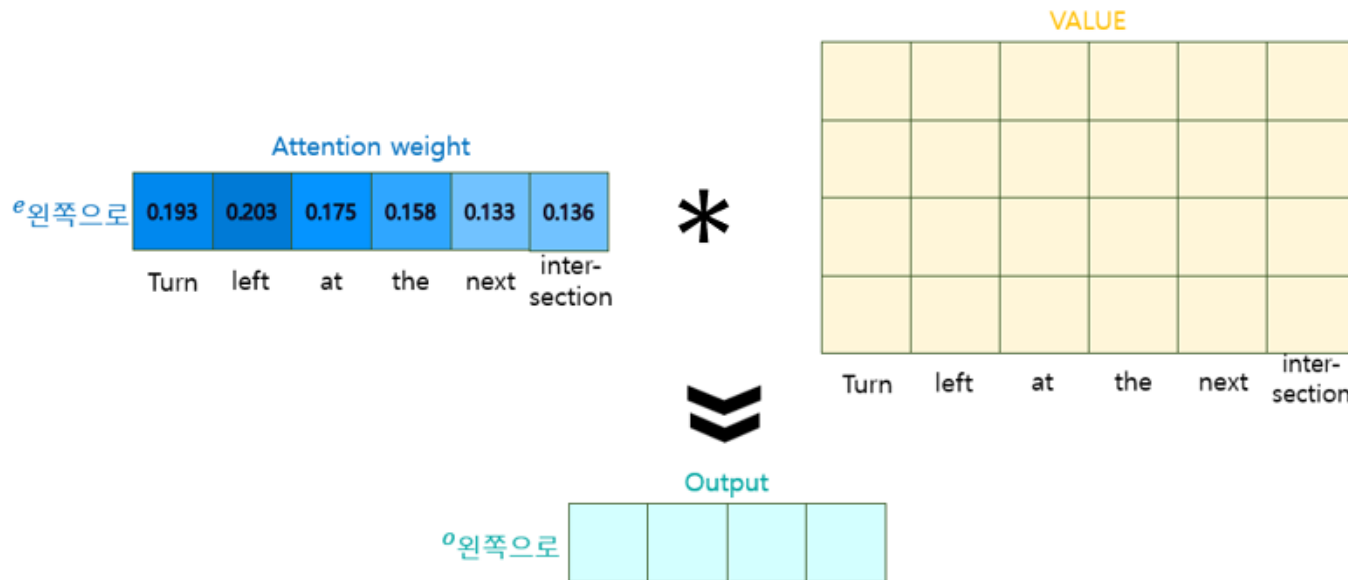
행렬 V 와의 곱 연산을 통해 context vector 생성



1

Attention

Attention 연산



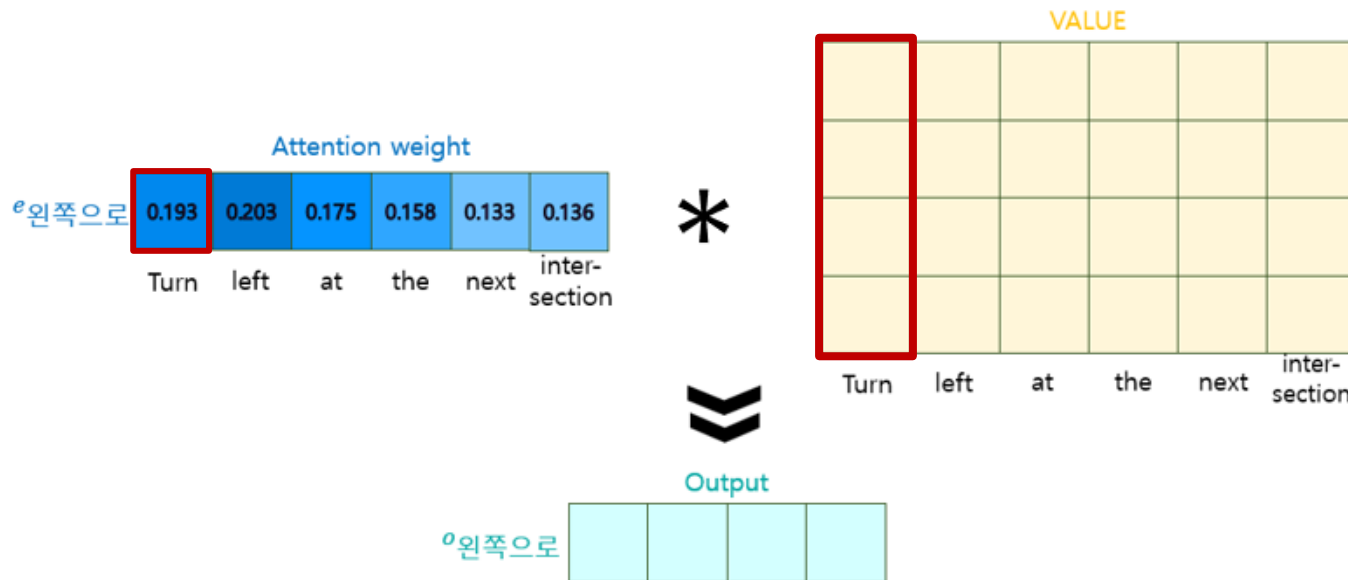
디코딩 시점 2에 사용될 Context Vector

$$0.193 * v_{\text{Turn}} + 0.203 * v_{\text{left}} + 0.175 * v_{\text{at}} + 0.158 * v_{\text{the}} + 0.133 * v_{\text{next}} + 0.136 * v_{\text{inter-section}}$$

1

Attention

Attention 연산



디코딩 시점 2에 사용될 Context Vector

$$0.193 * v_{\text{Turn}} + 0.203 * v_{\text{left}} + 0.175 * v_{\text{at}} + 0.158 * v_{\text{the}} + 0.133 * v_{\text{next}} + 0.136 * v_{\text{inter-section}}$$

2

Non-recurrent Encoder

Non-recurrent Encoder

그동안 인코더로는 단방향 혹은 양방향 RNN 모델을 주로 사용해 왔음



인코더가 만들어낸 Hidden state h_i 들을 디코더가 매 시점 참조할 수 있도록 제공



여기서 h_i 는 토큰들의 잠재 표현(hidden representation)

순차성을 고려한 RNN이 문맥을 포착해 만들어낸, 특정한 문맥이 반영된 임베딩

Context-specific encoder embedding이라고 볼 수 있음

Non-recurrent Encoder

*"The animal didn't cross the street because **it** was too tired"*

단어 '*it*'이 "*The animal*"을 참조한다는 걸 반영한 임베딩은 더 유용할 것

모델 학습 이전에 TF-IDF나 Word2Vec으로 만들어진
it'의 임베딩은 데이터셋의 모든 문장에서 동일



밀집 표현 혹은 희소 표현으로 사전에 만들어진 x_i 보다
 h_i 를 참조하는 게 task에 훨씬 유용함

Non-recurrent Encoder

하지만 트랜스포머 모델에서는 오직 Attention만을 이용하여
특정 문맥이 반영된 표현을 만들어내는 방법을 택함



RNN의 필요성이 사라져 비재귀적 인코더가 됨

Non-recurrent Encoder

하지만 트랜스포머 모델에서는 오직 Attention만을 이용하여



이때 쓰는 연산을 **Self-attention**이라고 함

주어진 문장 자기 자신과의 Attention 연산을 수행!

RNN의 필요성이 사라져 비재귀적 인코더가 됨

Self-Attention

Self-Attention

동일한 두 시퀀스를 대상으로 하는 Attention 연산



동일한 시퀀스의 서로 다른 위치에 있는 단어들을 연관시켜
해당 문장 내 단어 간 의존성을 고려해 문맥을 파악할 수 있음

Self-Attention

Self-Attention

동일한 두 시퀀스를 대상으로 하는 Attention 연산



동일한 시퀀스의 서로 다른 위치에 있는 단어들을 연관시켜
해당 문장 내 단어 간 의존성을 고려해 문맥을 파악할 수 있음

즉, RNN의 도움 없이 유용한 잠재 표현을 만들어낼 수 있음

Self-Attention

Self-Attention

동일한 두 시퀀스를 대상으로 하는 Attention 연산

$q_i = x_i W_q$, $k_i = x_i W_k$, $v_i = x_i W_v$ 로 Query, Key, Value를 정의



자기 자신과의 attention이므로 Query, Key, Value의

Source embedding이 x_i 로 동일함

x_i : 시퀀스 내 i 번째 토큰의 임베딩 벡터 (행 벡터)

Self-Attention

행렬 표현 $Q = XW_Q, K = XW_K, V = XW_V$ 로

시퀀스 내 모든 token에 대해 동시에 Attention 계산 가능



<행렬 표현이 가능한지 간단하게 확인하기>

$$X = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad Q = \begin{pmatrix} q_1 \\ q_2 \end{pmatrix} \quad W_Q = (w_1 \quad w_2)$$

w_i 는 열 벡터

q_i 는 행 벡터

$$Q = XW_Q = \begin{pmatrix} x_1 w_1 & x_1 w_2 \\ x_2 w_1 & x_2 w_2 \end{pmatrix} = \begin{pmatrix} x_1 W_Q \\ x_2 W_Q \end{pmatrix} = \begin{pmatrix} q_1 \\ q_2 \end{pmatrix}$$

Self-Attention

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

$$Q = \begin{pmatrix} q_1 \\ q_2 \end{pmatrix} \quad K = \begin{pmatrix} k_1 \\ k_2 \end{pmatrix} \quad V = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$$

k_i 는 행 벡터

v_i 는 행 벡터

$$QK^T = \begin{pmatrix} q_1 \\ q_2 \end{pmatrix} (k_1^T \quad k_2^T) = \begin{pmatrix} q_1 k_1^T & q_1 k_2^T \\ q_2 k_1^T & q_2 k_2^T \end{pmatrix}$$

$$softmax\left(\begin{pmatrix} q_1 k_1^T / \sqrt{d_k} & q_1 k_2^T / \sqrt{d_k} \\ q_2 k_1^T / \sqrt{d_k} & q_2 k_2^T / \sqrt{d_k} \end{pmatrix}\right) = \begin{pmatrix} s_{11}/s_1 & s_{12}/s_1 \\ s_{21}/s_2 & s_{22}/s_2 \end{pmatrix}$$

Self-Attention

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

$$softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) = \begin{pmatrix} s_{11}/s_1 & s_{12}/s_1 \\ s_{21}/s_2 & s_{22}/s_2 \end{pmatrix} = \begin{pmatrix} w_1(1) & w_2(1) \\ w_1(2) & w_2(2) \end{pmatrix}$$

$softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)$ 는 각 쿼리에 대해,

모든 key들과 유사도를 계산해 얻은 attention weight 행렬

Self-Attention

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

$$softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V = \begin{pmatrix} w_1(1) & w_2(1) \\ w_1(2) & w_2(2) \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$$

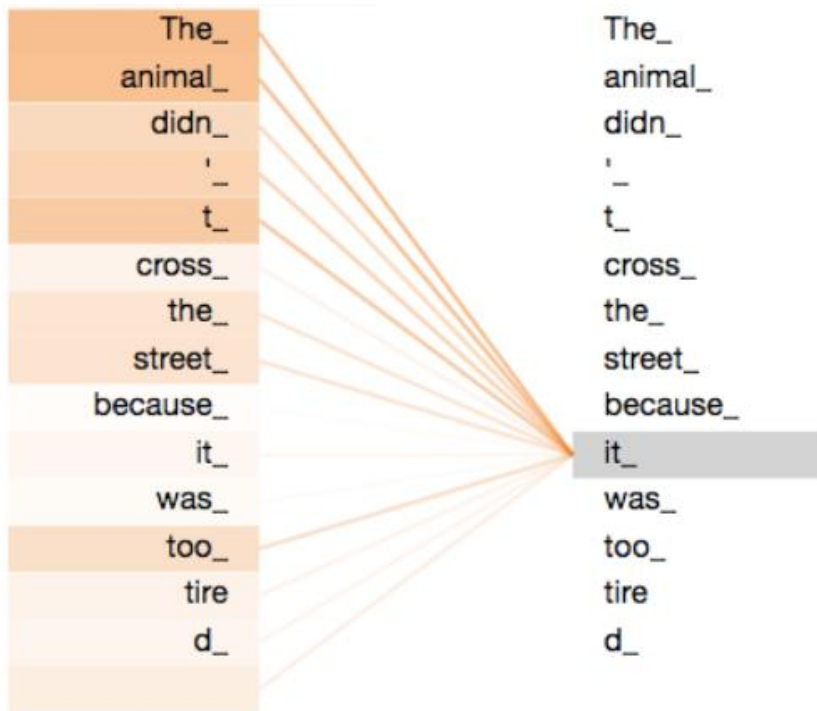
$$= \begin{pmatrix} w_1(1)v_1 + w_2(1)v_2 \\ w_1(2)v_1 + w_2(2)v_2 \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}$$

c_t 는 행 벡터

$$c_t = \sum_{i=1}^N w_i(t)v_i$$

$Attention(Q, K, V)$ 는 결국 토큰의 Value 벡터를 가중 평균 낸 결과

Self-Attention

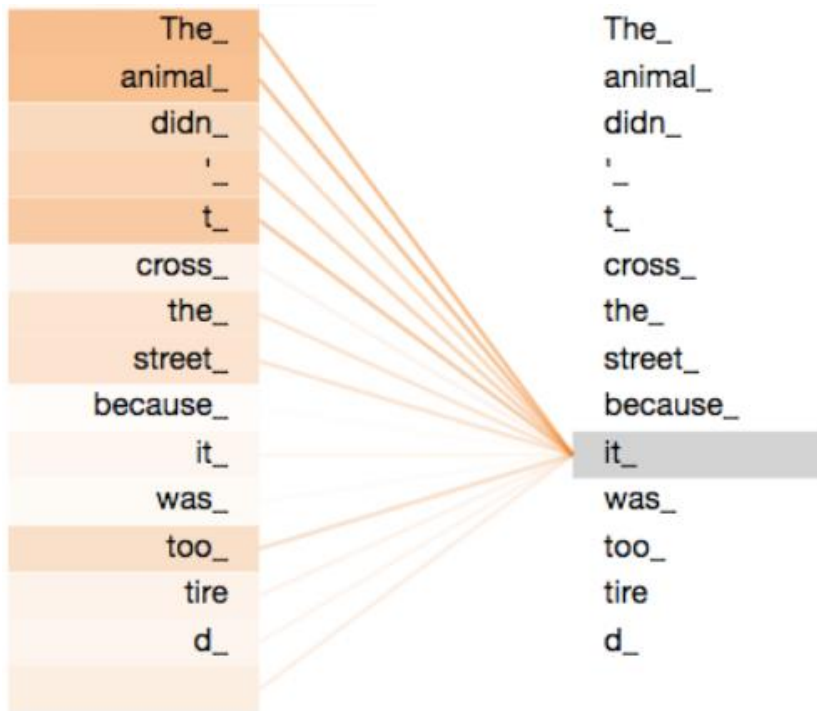


오른쪽이 쿼리, 왼쪽이 키

모델의 표현력이 충분하고
최적화가 제대로 된 상황이라면
 c_t 는 x_t 와 연관된 Value 벡터들에게
더 큰 가중치를 부여하게 됨

c_t 는 문맥이 반영된 x_t 의 잠재 표현

Self-Attention



오른쪽이 쿼리, 왼쪽이 키

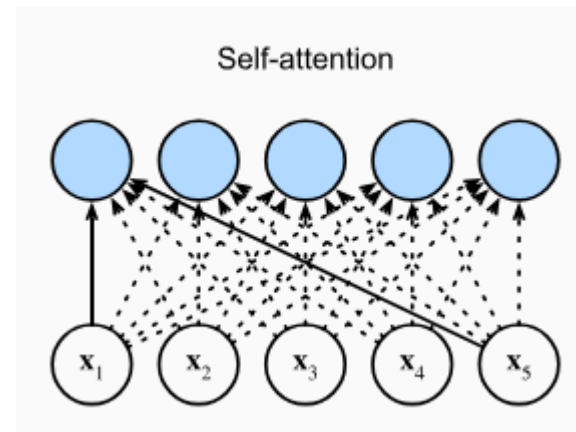
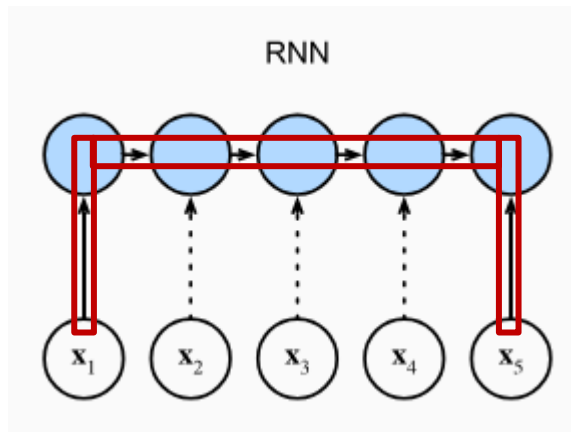
실제로 Attention 연산
한 번으로는 충분하지 않음

다음 레이어에서 반복적으로
연산을 수행하도록 인코더를
구성하는 것이 일반적

RNN vs Self-Attention



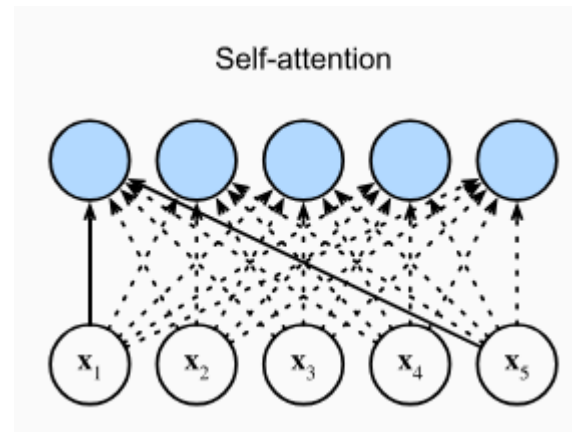
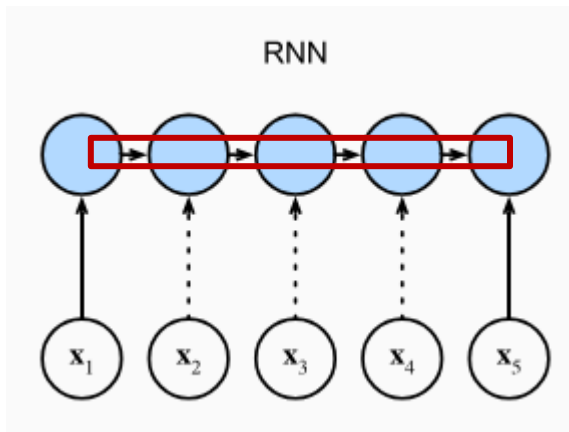
Self-Attention 구조를 사용하면 시점 상 멀리 떨어진 두 토큰
사이의 상호작용을 **최소한의 정보 손실로 고려 가능**



RNN vs Self-Attention



또한 순차적 연산을 진행하는 RNN과 달리 병렬화가 가능해
훈련 시간을 줄일 수 있음



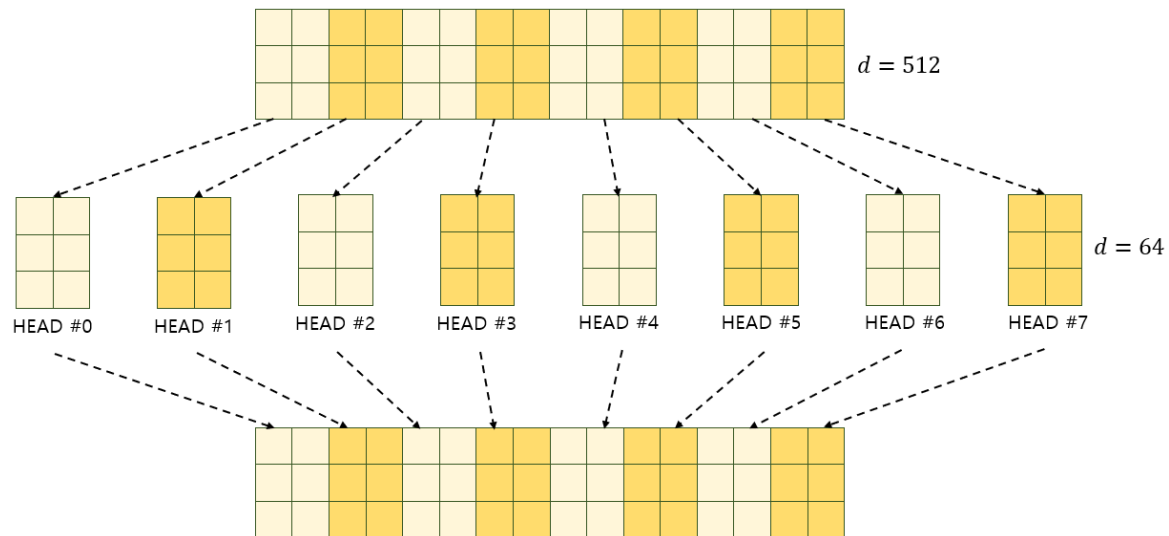
2

Non-recurrent Encoder

Multi-Head Attention

Multi-Head Attention

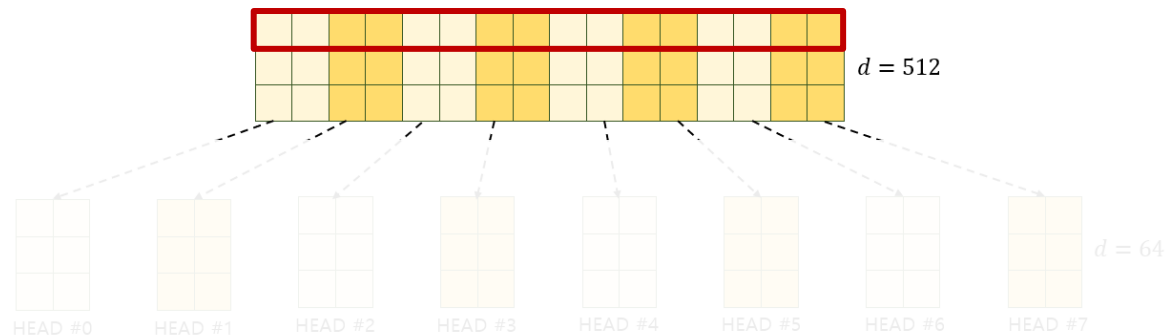
한 시퀀스에 여러 개의 Attention을 병렬적으로 적용하는 방법



Multi-Head Attention

Multi-Head Attention

한 시퀀스에 여러 개의 Attention을 병렬적으로 적용하는 방법



하나의 행 벡터가 토큰 임베딩 하나를 의미하며
행의 수가 곧 시퀀스의 길이와 같음

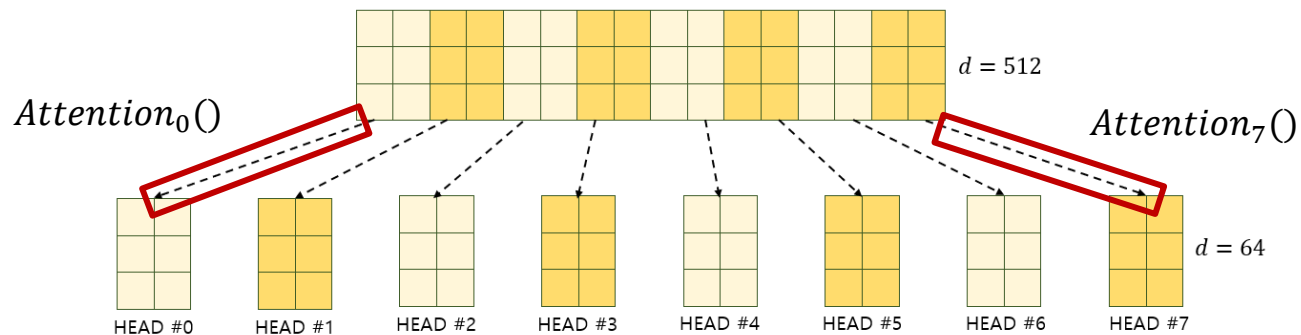
2

Non-recurrent Encoder

Multi-Head Attention

Multi-Head Attention

한 시퀀스에 여러 개의 Attention을 병렬적으로 적용하는 방법



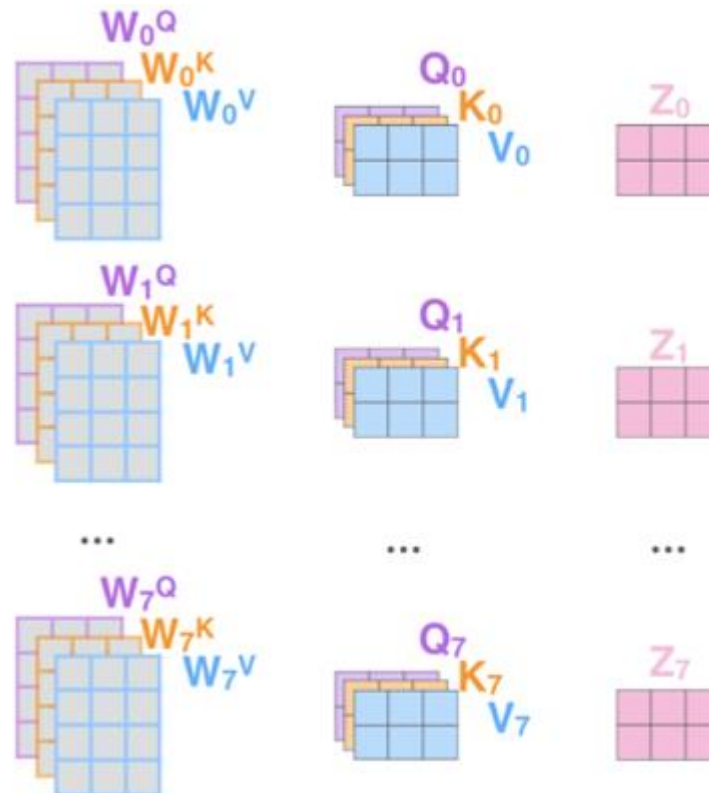
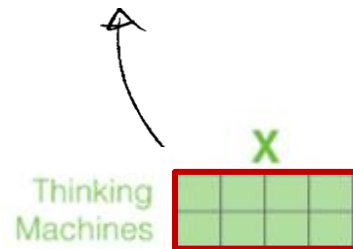
병렬로 연결된 각 Attention layer를 Head라고 부르고
그 layer가 여러 개라서 Multi-head attention!

2

Non-recurrent Encoder

Multi-Head Attention

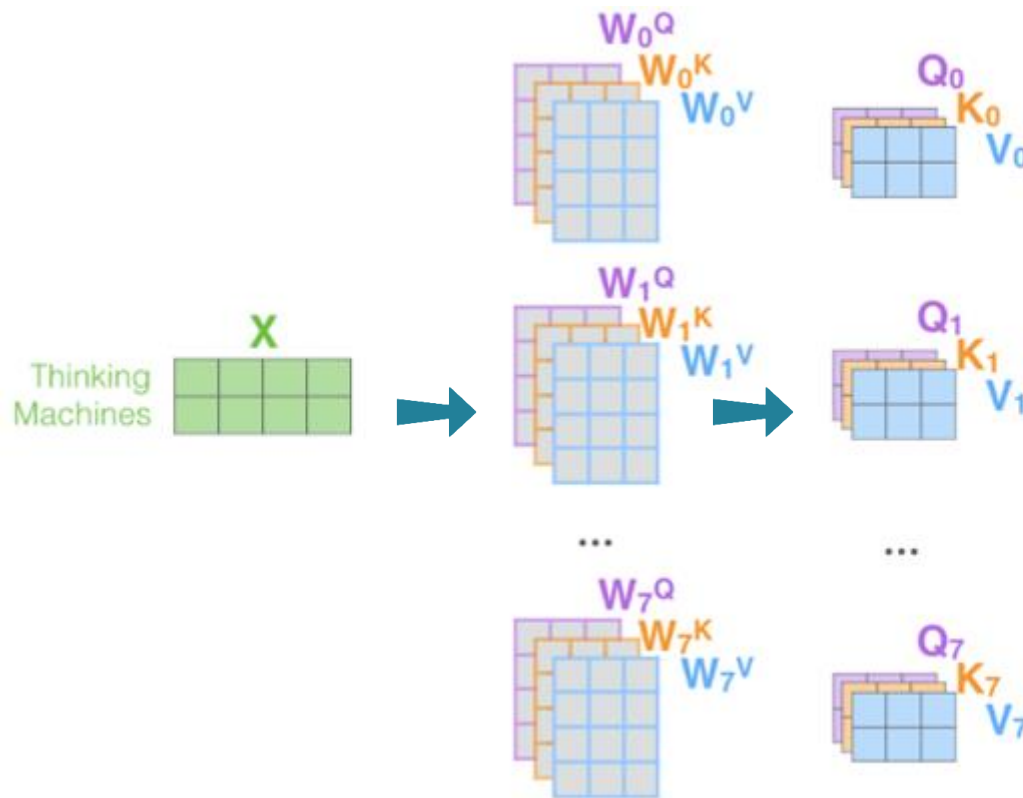
토큰들의 임베딩 행렬 X



2

Non-recurrent Encoder

Multi-Head Attention



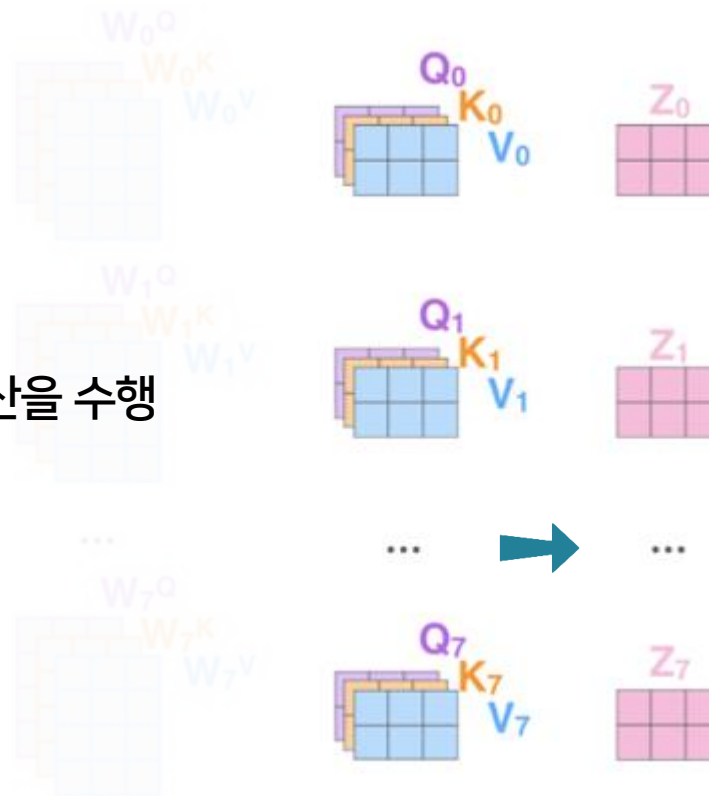
① 임베딩 행렬 X 에
 W_i^Q, W_i^K, W_i^V 를 곱해
저차원의 Q, K, V 를
 각 Head에 대해 만들

2

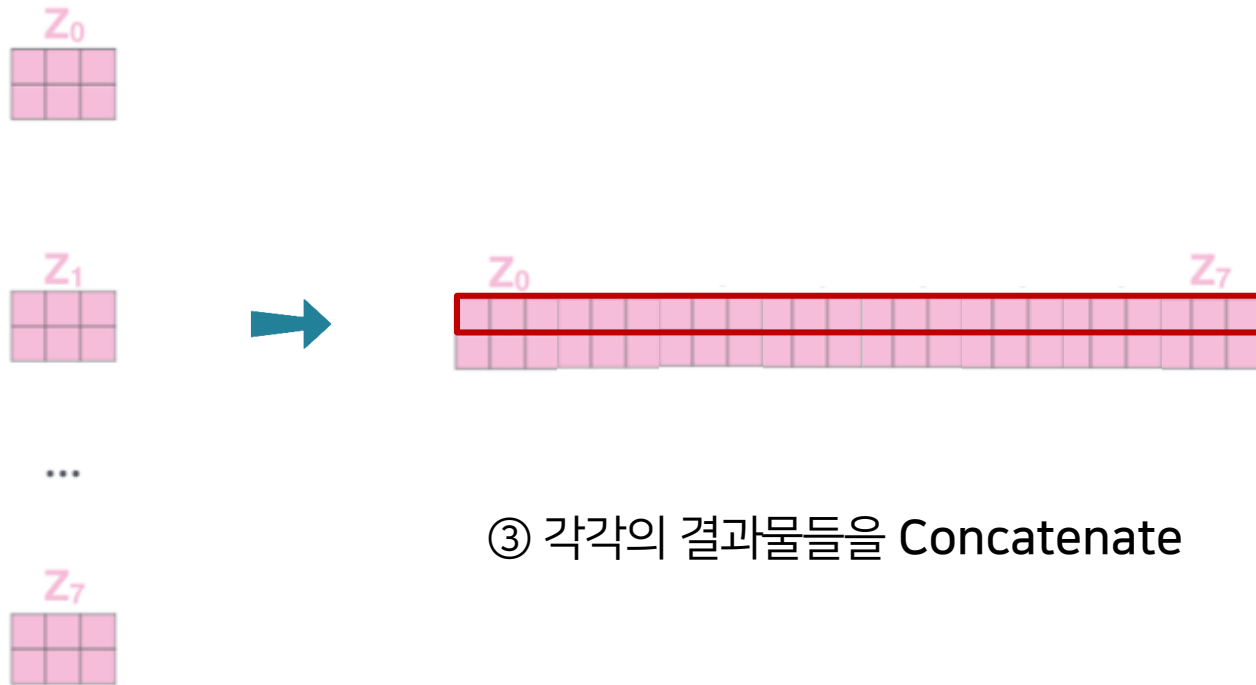
Non-recurrent Encoder

Multi-Head Attention

② 독립적으로 어텐션 연산을 수행

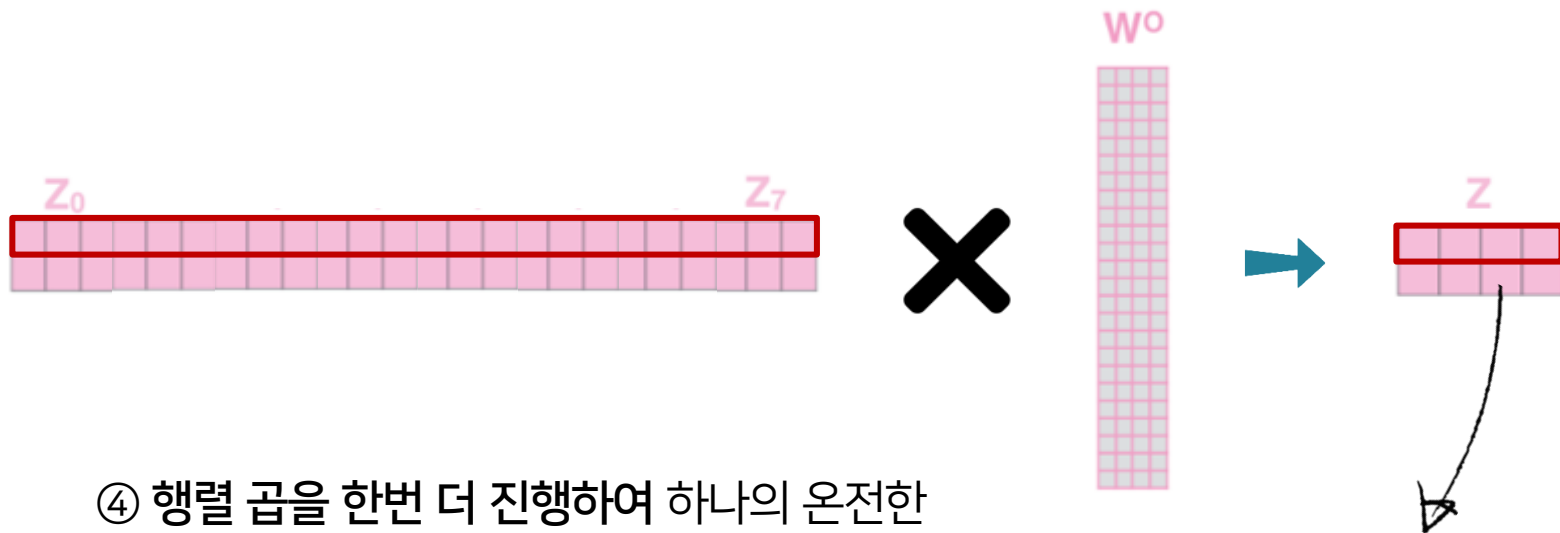


Multi-Head Attention



③ 각각의 결과물들을 Concatenate

Multi-Head Attention



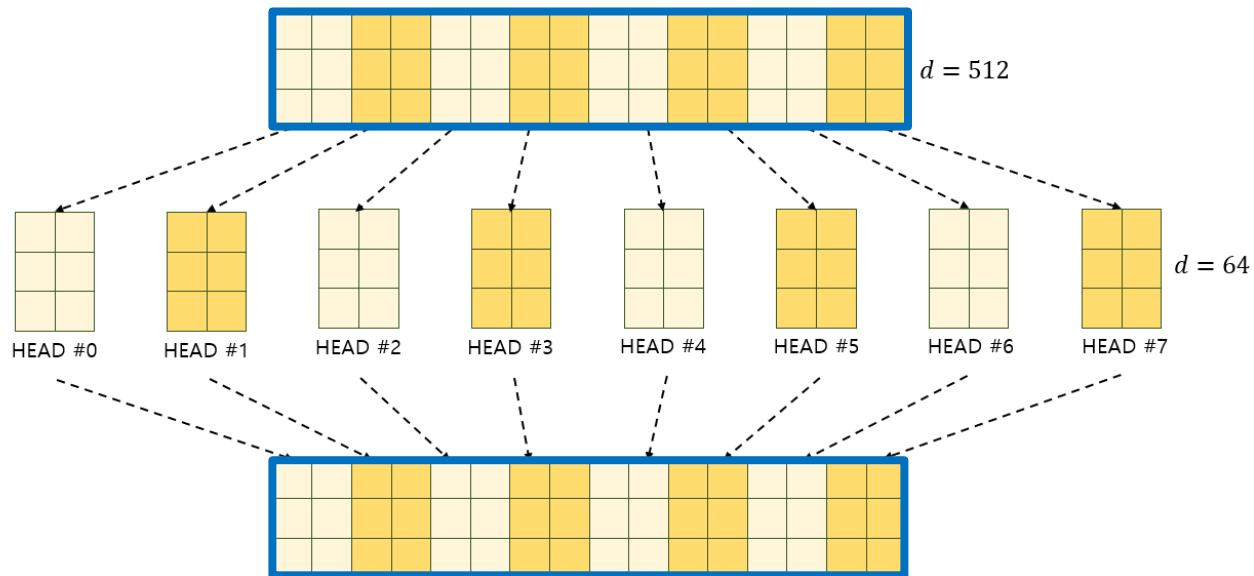
- ④ 행렬 곱을 한번 더 진행하여 하나의 온전한 context-specific embedding을 만듦

'Thinking'의 context-specific embedding

2

Non-recurrent Encoder

Multi-Head Attention

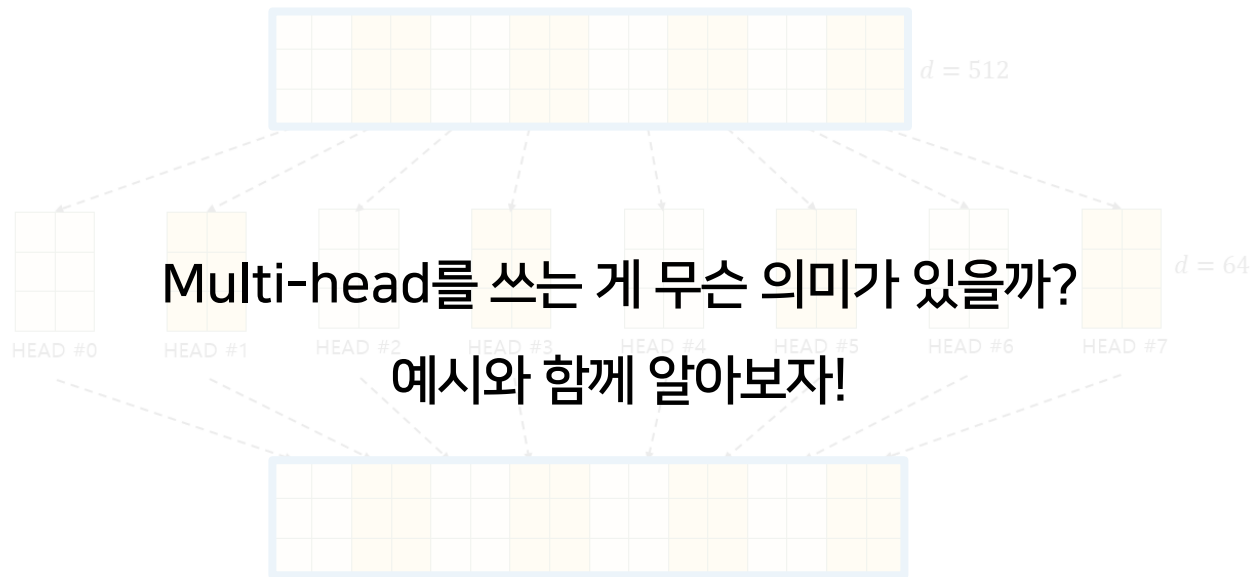


보통의 경우 Multi-head Attention 연산 전과
연산 후의 차원을 동일하게 맞춰 줌

2

Non-recurrent Encoder

Multi-Head Attention



보통의 경우 Multi-head Attention 연산 전과
연산 후의 차원을 동일하게 맞춰 줌

Multi-Head Attention

I run a small **business**

I run a **mile** in 10 minuets

The **robber** made **a run** for it

The **stocking** had **a run**

위 예시에서 run을 한국어로 제대로 번역하기 위해선
각 문장의 문맥을 정확히 파악해야 함

Multi-Head Attention

I run a small **business**

I run a **mile** in 10 minuets

The **robber** made a run for it

The **stocking** had a run

run의 품사를 알아내기 위해선 가까이 위치한 'I'와 'a'를,
run의 의미를 알아내기 위해선 상대적으로 멀리 떨어진
'business', 'mile', 'robber', 'stocking'을 참조해야 함

Multi-Head Attention

I run a small **business**

I run a **mile** in 10 minuets

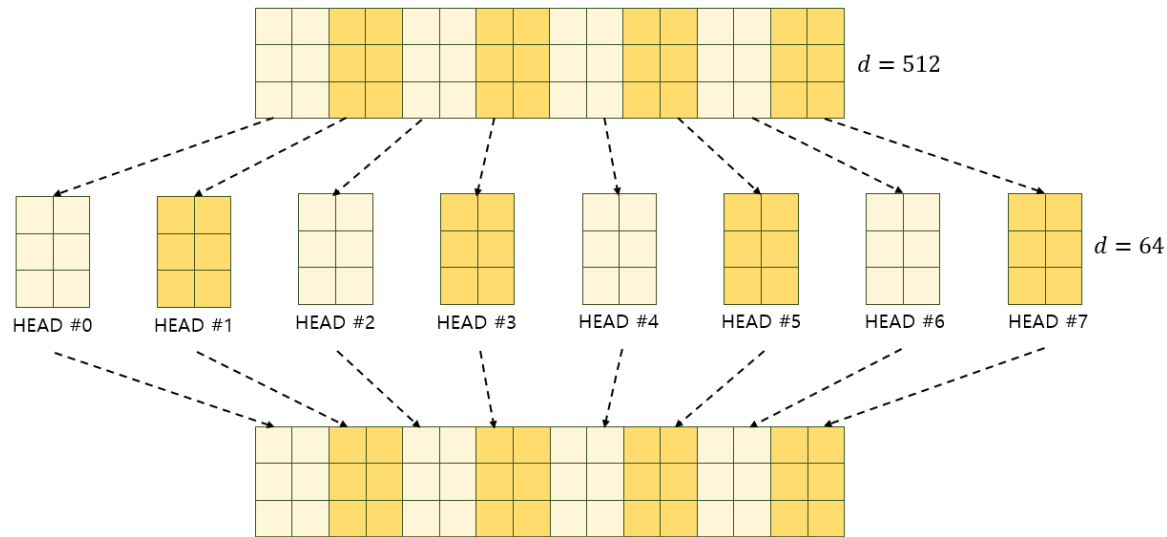
The **robber** made **a** run for it

The **stocking** had **a** run



즉 서로 다른 종류의 문맥적 정보를 정확히 포착하기 위해
별개의 Attention module을 여러 개 두는 것이 도움이 됨

Multi-Head Attention



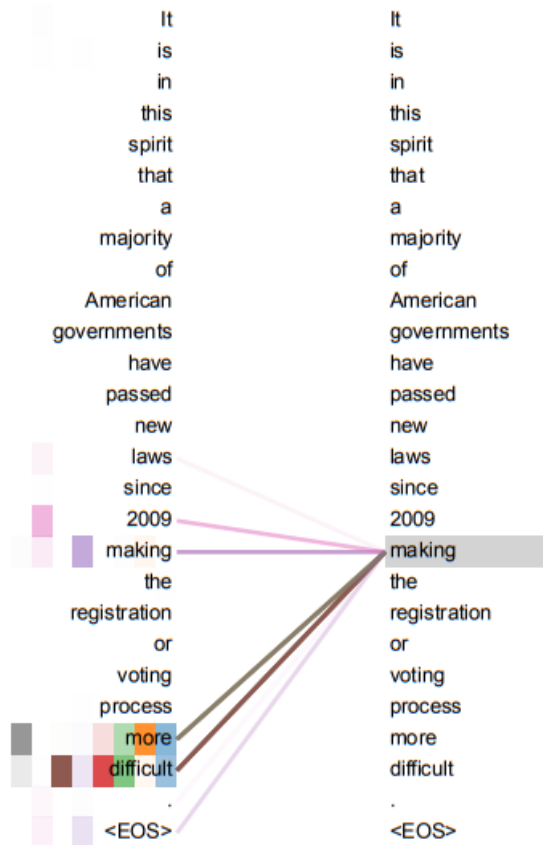
이 Head들은 같은 문장 내 서로 다른 특징들을 학습할 수 있기 때문에

서로 다른 문맥적 구조를 학습할 수 있음

Ex: 품사 파악, 문맥에 따라 변하는 의미, co-reference resolution ...

Multi-Head Attention

<Multi-Head Attention weight 시각화>

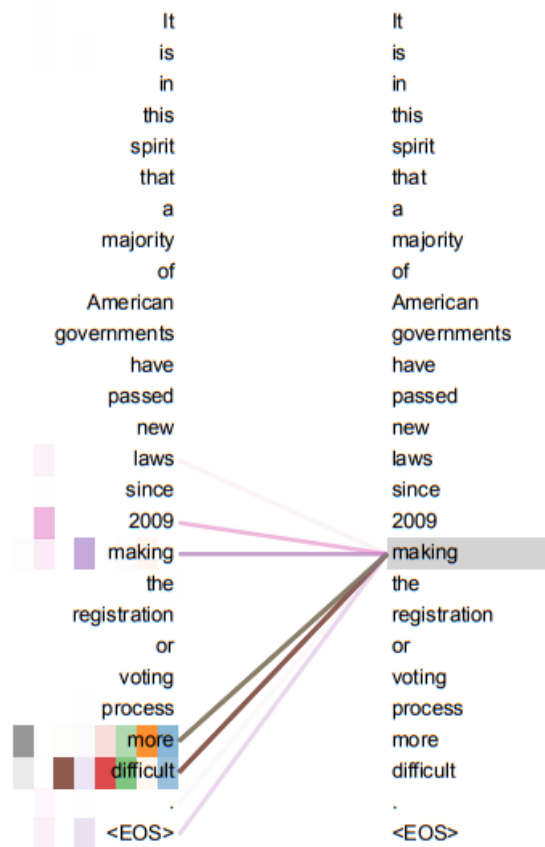


'making'에 대한 Query가 어느 key에
더 집중했는지는 Head마다 다름

각 Head마다 가중치의 분포를
서로 다른 색으로 표시하여 시각화 할 수 있음

Multi-Head Attention

<Multi-Head Attention weight 시각화>



일부 Head들은 다른 Head들과
다른 문맥적 구조를 학습했음을 짐작 가능

모든 Attention layer의 시각화 결과가
직관과 부합하지는 않으니 해석에 주의

3

Transformer

트랜스포머 (Transformer)

Transformer

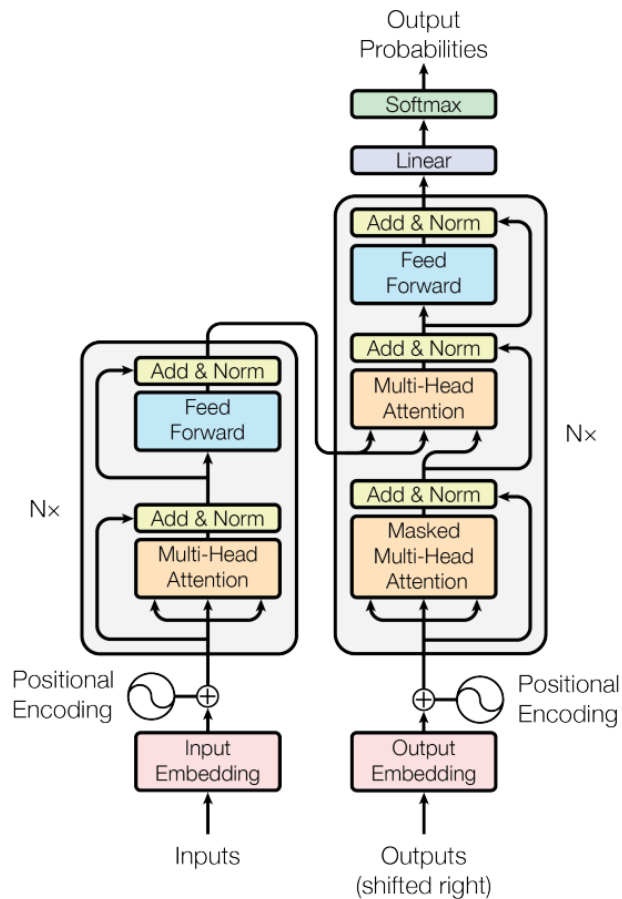
기존의 Encoder-Decoder 구조를 따르지만
오로지 Attention 기법만을 이용해 문맥을 학습하는 모델

2017년 구글의 논문 "Attention is all you need"에서 소개됨
기존의 RNN 기반 Encoder-Decoder에 비해 훨씬 더 좋은 성능을 보였고,
현재 많은 모델들이 이 모델에서 파생되어 사용되고 있음
비전, 오디오 등등

3

Transformer

트랜스포머 (Transformer)



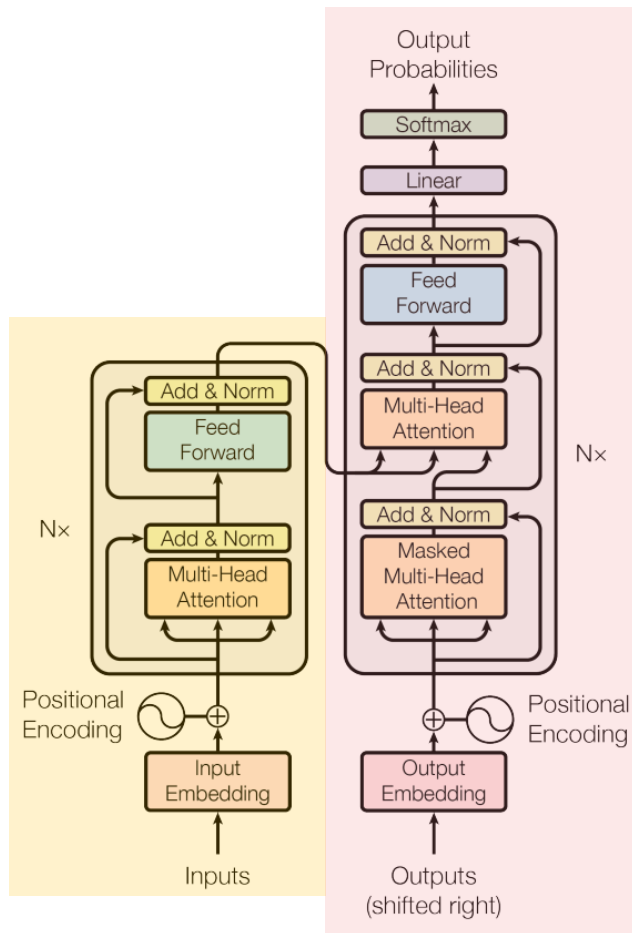
Transformer 모델의 구조

일반적인
Encoder-Decoder 구조를 따름

인코더 또는 디코더는
각각의 블록을 N번 쌓고 있음

같은 구조의 연산이
반복적으로 수행됨을 의미

트랜스포머 (Transformer)



Transformer 모델의 구조

일반적인

Encoder-Decoder 구조를 따름

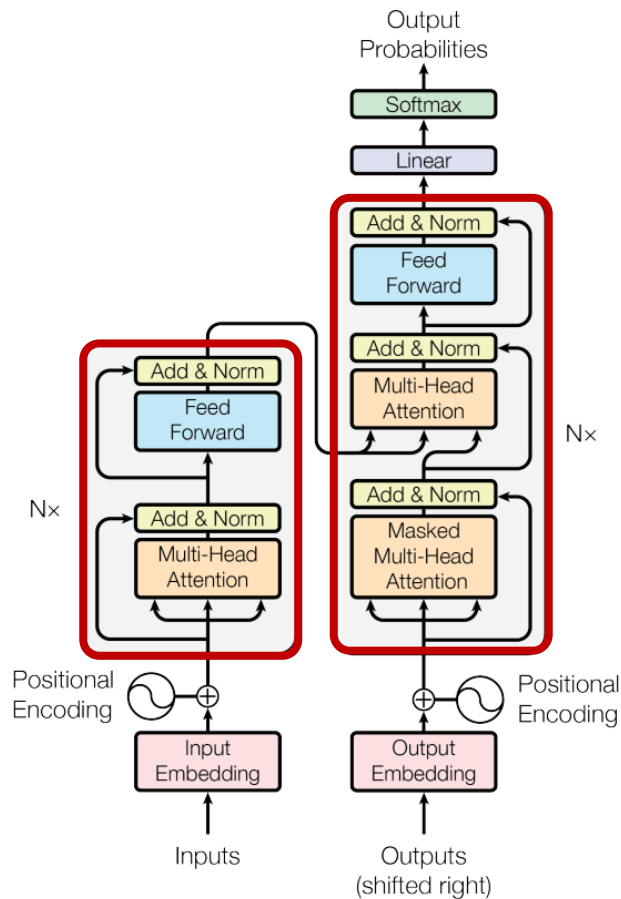
인코더 또는 디코더는

각각의 블록을 N 번 쌓고 있음

같은 구조의 연산이

반복적으로 수행됨을 의미

트랜스포머 (Transformer)



Transformer 모델의 구조

일반적인

Encoder-Decoder 구조를 따름

인코더 또는 디코더는

각각의 블록을 N 번 쌓고 있음

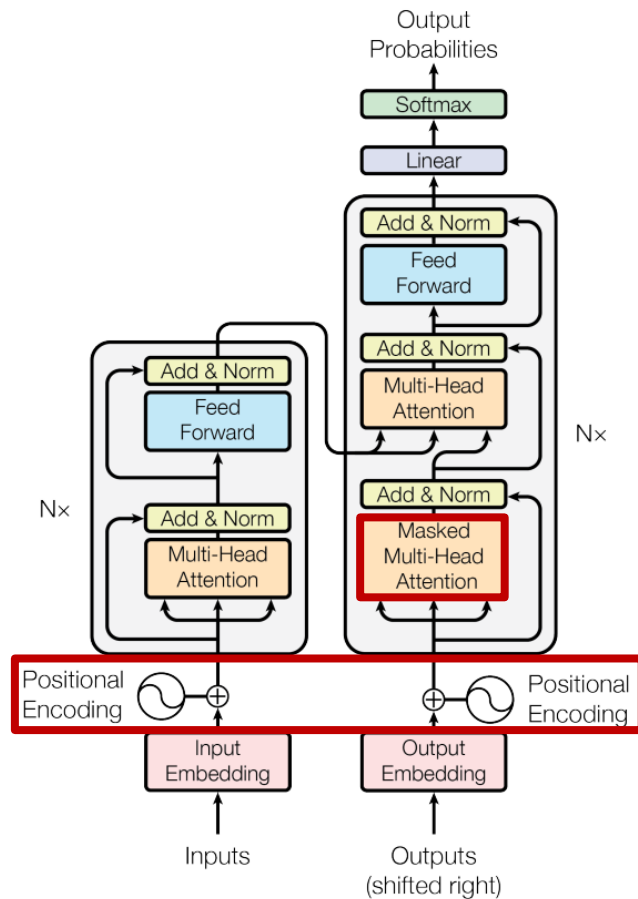
같은 종류의 연산이

반복적으로 수행됨을 의미

3

Transformer

트랜스포머 (Transformer)



Transformer 모델의 구조

일반적인

Encoder-Decoder 구조를 따름

대부분의 연산은 기존의 Attention과
이쿠더 또는 디쿠더는
같으나 몇 가지 새로운 요소들이 존재
각각의 블록을 N 번 쌓고 있음

같은 구조의 연산이

반복적으로 수행됨을 의미

트랜스포머 구조 보완하기 (Self-Attention 문제 해결)

Transformer는 RNN 기반 Encoder-Decoder가 아닌
Attention 기반 Encoder-Decoder 구조를 가짐

➡ Attention만으로 모델을 만들기 위해서
몇 가지 보완해줘야 할 부분들이 있음

어떤 문제들이 있었고,
새로운 요소들을 추가하면서 어떻게 문제를 해결할 수 있었는지 알아보자!



트랜스포머 구조 보완하기 | Positional Encoding

Attention은 행렬 곱을 통해 모든 토큰들에 대해
한꺼번에 내적 연산을 함 즉 순차적인 연산이 아님

→ 자연어의 특성인 **순차성**을 무시하게 됨



순서가 무시되면 "dogs chase cats"라는 문장과
"cats chase dogs"를 구별할 수 없음

트랜스포머 구조 보완하기 | Positional Encoding

Attention은 행렬 곱을 통해 한꺼번에 내적 연산을 함
즉 순차적인 연산이 아님

→ 역으로 **위치 정보**를 넣어줄 필요가 있음!

이를 위해 등장한 것이 **Positional Encoding**

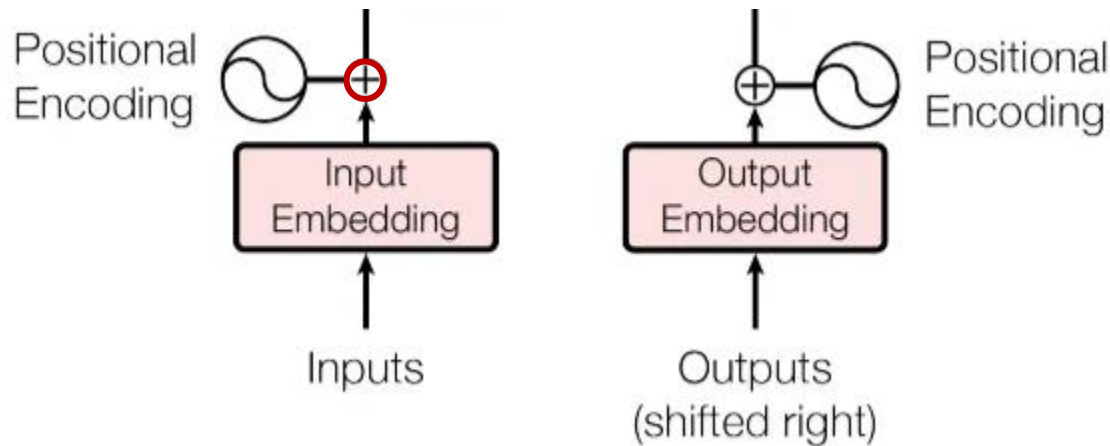


순서가 무시되면 "dogs chase cats"라는 문장과
"cats chase dogs"를 구별할 수 없음

트랜스포머 구조 보완하기 | Positional Encoding

Positional Encoding

각 토큰마다 **위치에 대한 정보**를 인코딩하는 것



임베딩이 모델에 들어가기 전 위치 정보를 넣어주는 모습

3

Transformer

트랜스포머 구조 보완하기 | Positional Encoding

단어마다 임베딩 벡터에

포지션 벡터를 더해주어

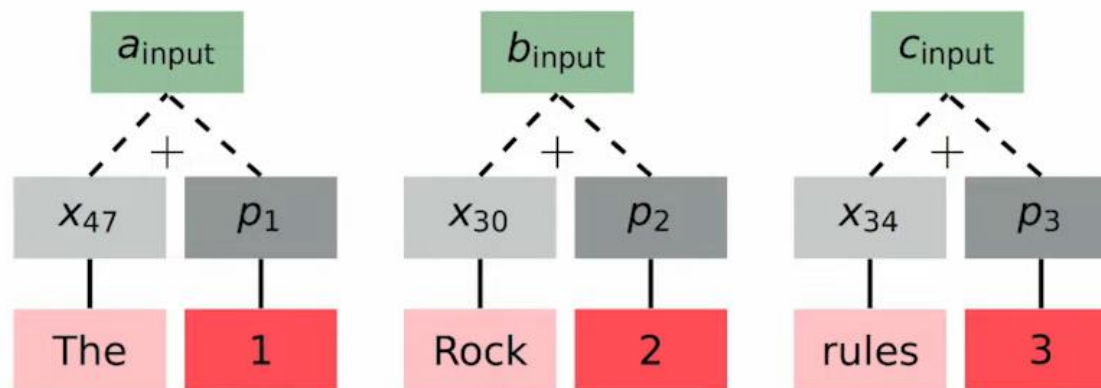
위치에 대한 정보를 반영

즉 학습 이전에 입력 값에 위치를 반영

문장의 순서를 바꿔 넣었을 때 전혀

다른 값이 모델에 입력되므로

모델이 구별할 수 있게 됨



왼쪽 x : 단어 임베딩, 오른쪽: 위치 임베딩

트랜스포머 구조 보완하기 | Positional Encoding



Positional Encoding이 갖춰야 할 조건

- ① 각 위치의 Positional 값은 유일해야 함
- ② 서로 다른 길이의 시퀀스에 대해서도 time-step마다 동일한 간격을 가져야 함
- ③ 더 긴 시퀀스를 만나도 모델이 일반화하는데 문제가 없어야 함
- ④ 각 위치의 Positional Encoding 값은 결정론적이어야 함

Ex) $[0,1]$ 를 N등분하는 값을 각 토큰에 할당하는 것 **X**

단순히 a번째 토큰에 숫자 a를 할당하는 것 **X**

트랜스포머 구조 보완하기 | Positional Encoding



Positional Encoding이 갖춰야 할 조건

- ① 각 위치의 Positional 값은 유일해야 함
- ② 트랜스포머를 다룬 논문에서는 Positional Encoding이 동일한 간격을 가져야 하는지 어떤 방법으로 제안됐을까?
Ex) $[0,1]$ 를 N 등분하는 값을 각 토큰에 할당하는 것 **X**
- ③ 더 긴 시퀀스를 만나도 모델이 일반화하는데 문제가 없어야 함
Ex) 단순히 a 번째 토큰에 숫자 a 를 할당하는 것 **X**
- ④ 각 위치의 Positional Encoding 값은 결정론적이어야 함

트랜스포머 구조 보완하기 | Positional Encoding



Sinusodial Encoding

Sine & Cosine 함수를 사용한 방법

$$PE_{(pos, 2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_m}}}\right)$$

$$PE_{(pos, 2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_m}}}\right)$$



이 방법은 4가지 조건을 만족하며 각 (pos, i) 쌍에 적절한 값을 제공

트랜스포머 구조 보완하기 | Positional Encoding

He left his right shoe at home

pos
(value = 1)

Positional
Embedding

He				
left				
his				
right				
shoe				
at				
home				

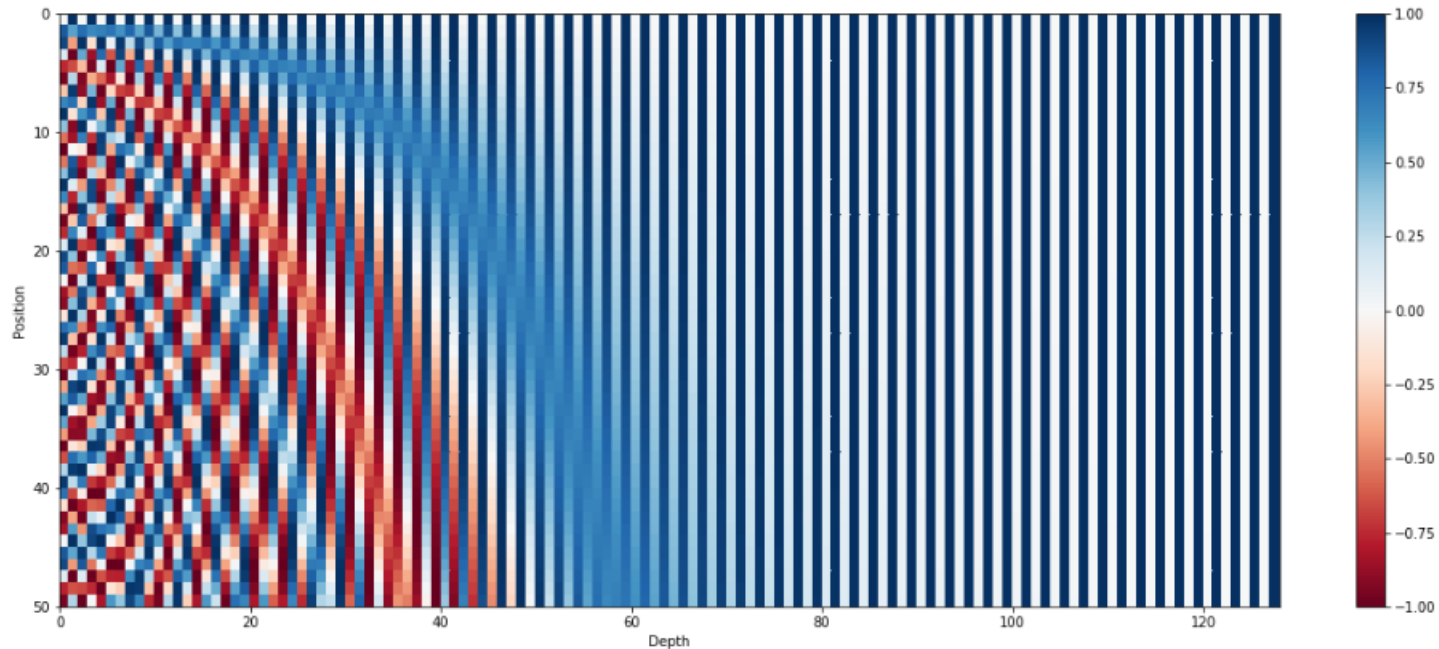
i
(value = 1)



pos 는 시퀀스에서의 위치 인덱스, i 는 임베딩 벡터의 인덱스를 의미

트랜스포머 구조 보완하기 | Positional Encoding

(예시) Sinusodial encoding을 이용해 위치 임베딩을 했을 때
얻어지는 약 50개의 위치 임베딩 벡터들의 값 분포를 시각화



x축: 임베딩 벡터 차원 인덱스, y축: 시퀀스 내 위치 인덱스

트랜스포머 구조 보완하기 | Feed-Forward NN

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

$Attention(Q, K, V)$ 은 단순 행렬 곱이므로
Linear Transformation의 일종



비선형적 관계를 학습하는데 어려움이 있음

트랜스포머 구조 보완하기 | Feed-Forward NN

완전 연결층(Fully-Connected Feed Forward Neural Network)를

추가해 이를 보완해줌

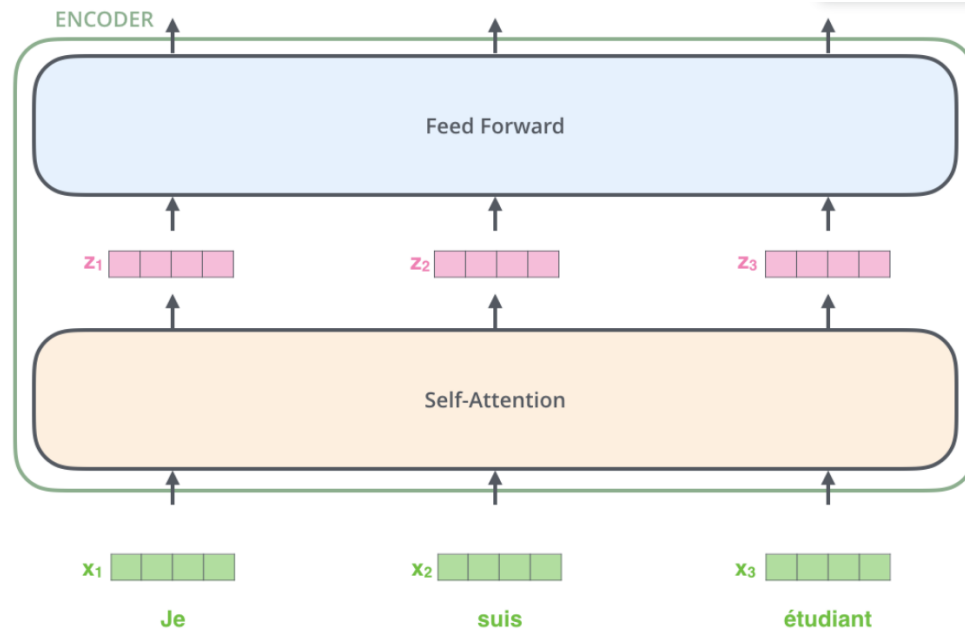
$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

2개의 Linear Transformation을 수행하되

중간에 비선형 함수 ReLU를 통과시켜줌을 의미

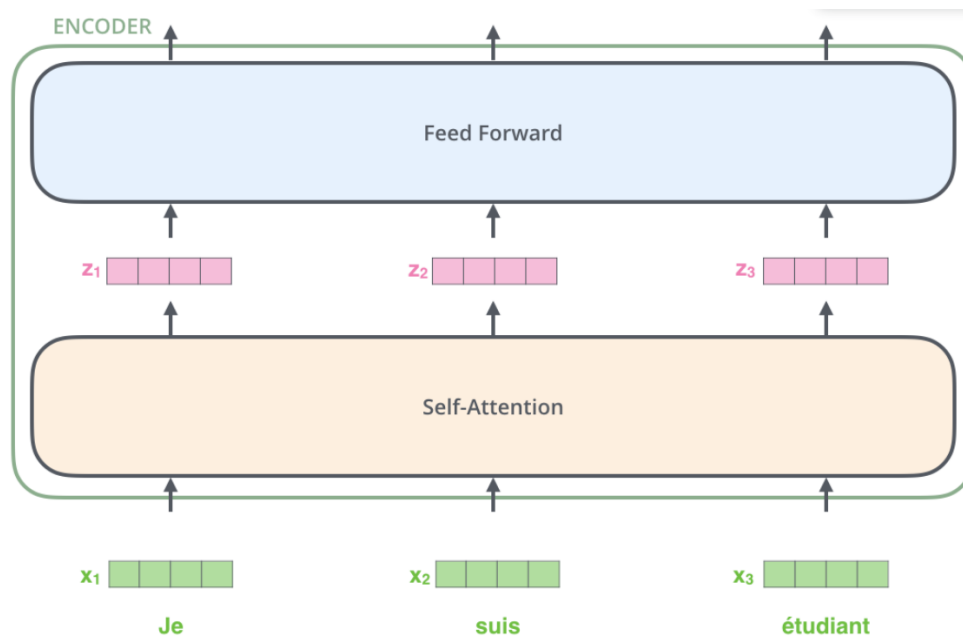
트랜스포머 구조 보완하기 | FFNN

Attention 연산 후 얻어진 새로운 임베딩 벡터들을 $x_i (i = 1, \dots, N)$ 라고 할 때,
이 서로 다른 위치에 있는 x_i 들을 함수 $FFN(x)$ 에 각각 넣어줌



트랜스포머 구조 보완하기 | FFNN

이때 같은 완전 연결층 내에서는 파라미터의 값이 공유됨

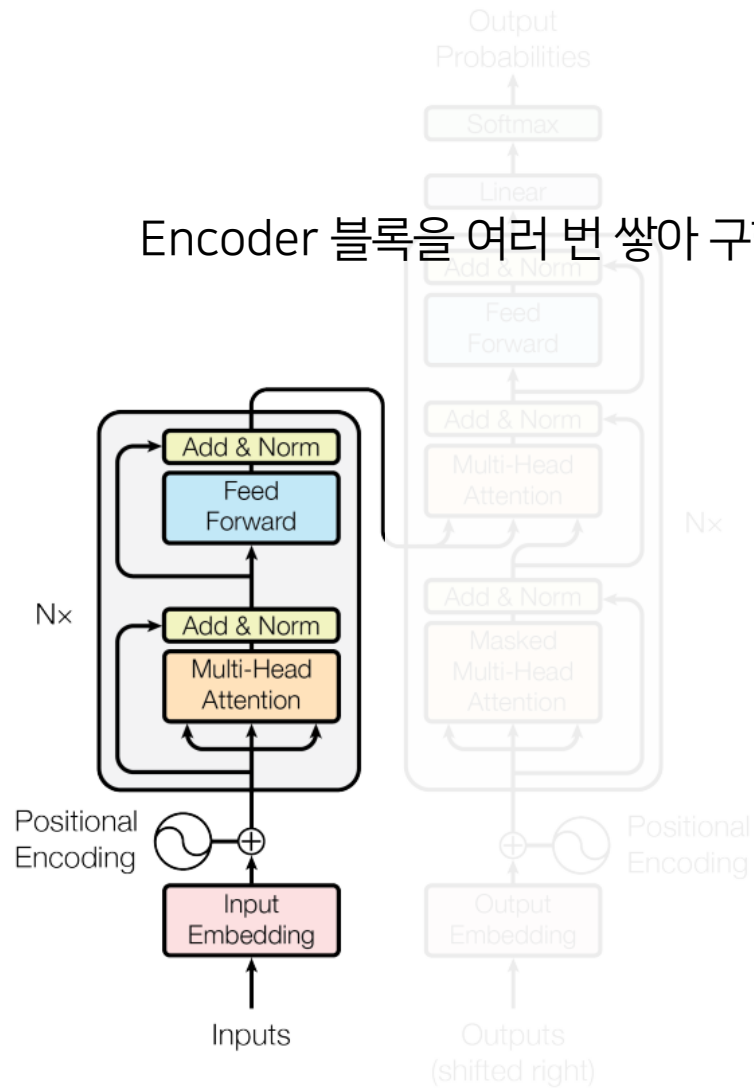


3

Transformer

Encoder

Encoder 블록을 여러 번 쌓아 구현

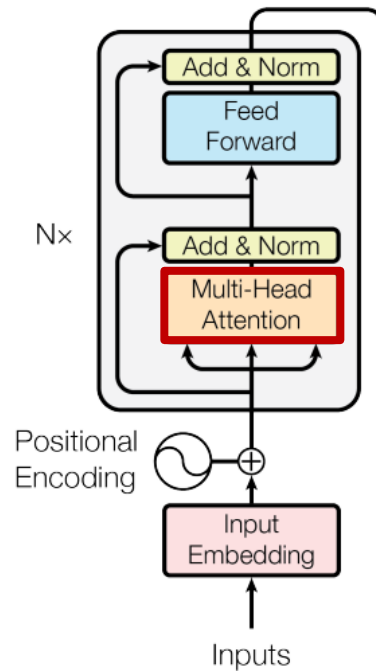


3

Transformer

Encoder

Encoder 블록을 여러 번 쌓아 구현



블록에 들어온 입력 값을 대상으로

Multi-Head Attention 연산 수행

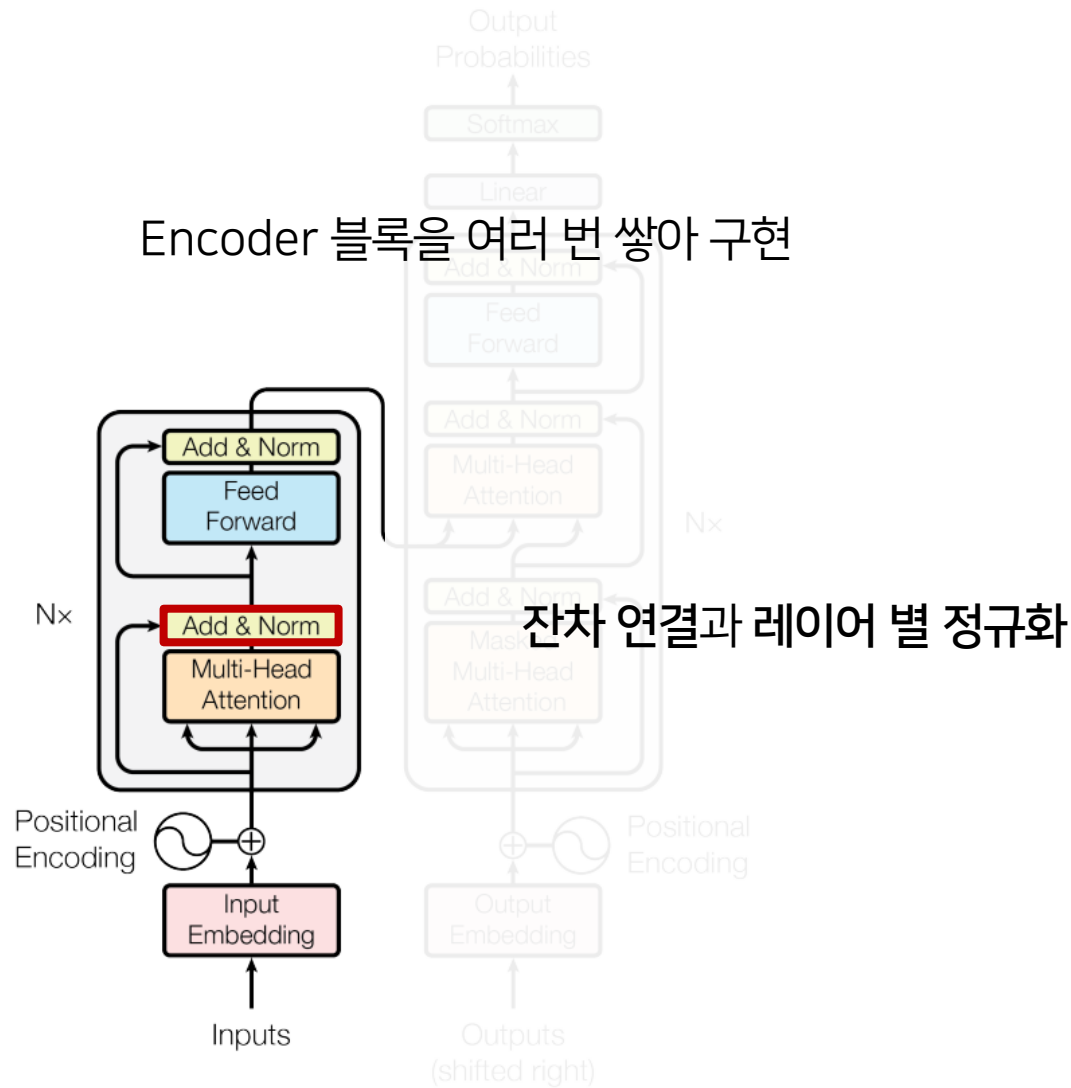


3

Transformer

Encoder

Encoder 블록을 여러 번 쌓아 구현

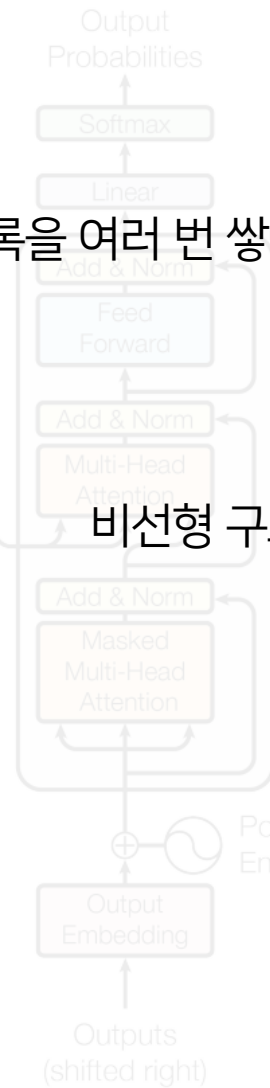
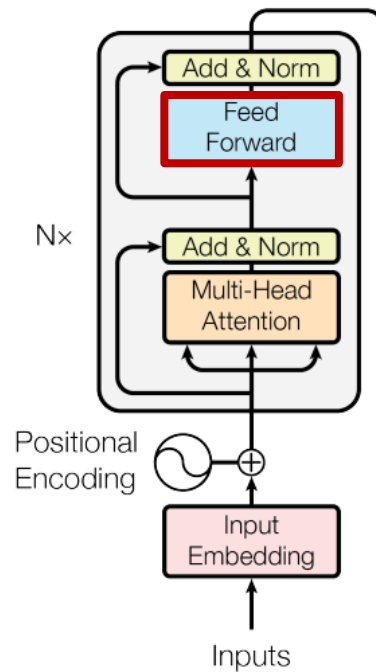


3

Transformer

Encoder

Encoder 블록을 여러 번 쌓아 구현



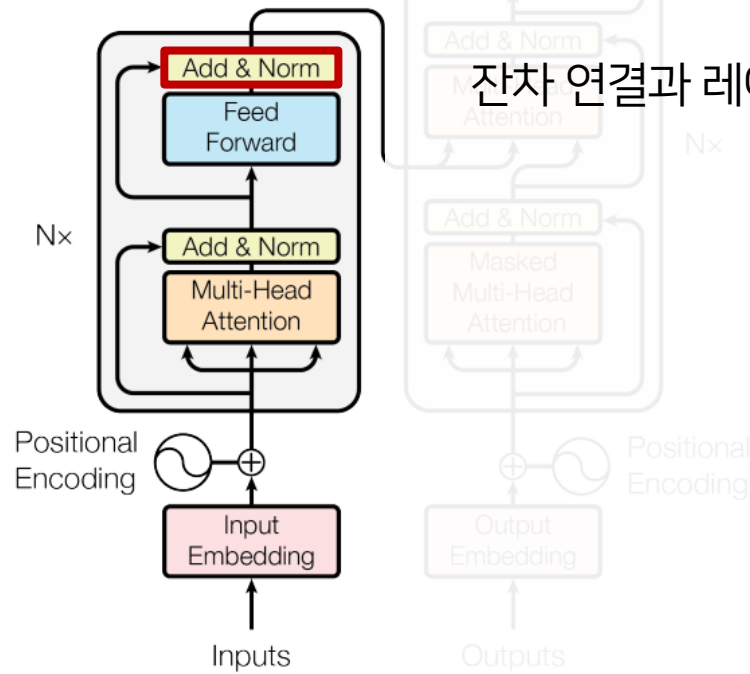
비선형 구조를 학습하기 위해 FFNN을 추가로 통과

3

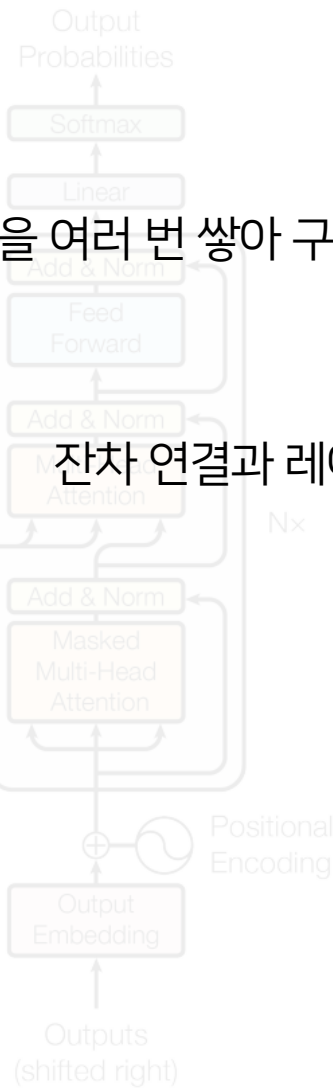
Transformer

Encoder

Encoder 블록을 여러 번 쌓아 구현



잔차 연결과 레이어 별 정규화를 재수행

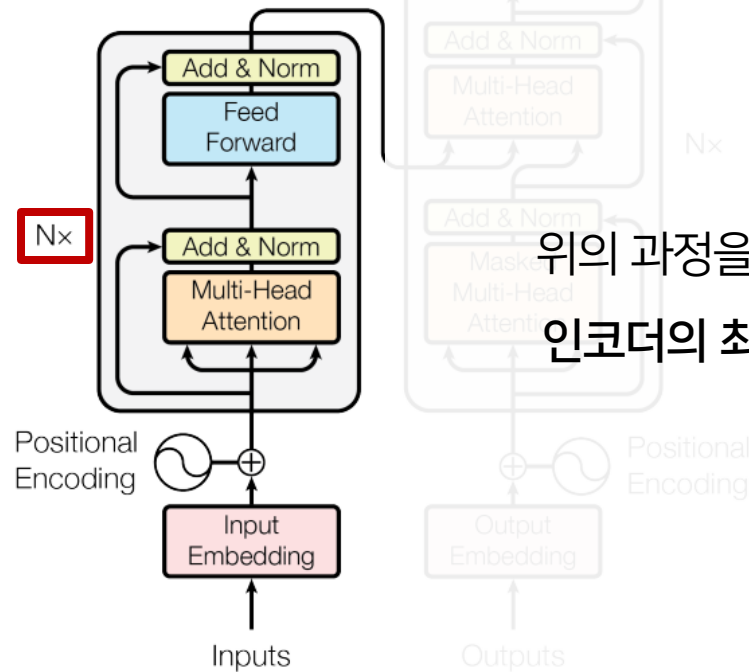


3

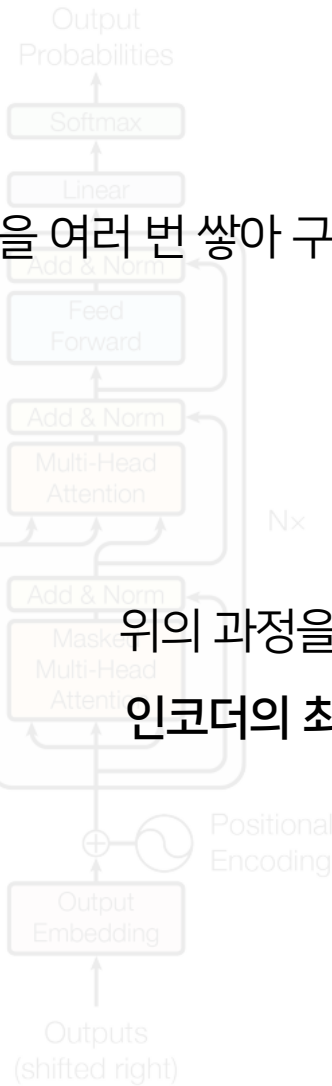
Transformer

Encoder

Encoder 블록을 여러 번 쌓아 구현



위의 과정을 총 N번 반복하여
인코더의 최종 Output 출력



성능 향상 기법

잔차 연결

모델의 입력 값을 복제하여 중간층의 출력 값과 더하는
Identity mapping을 수행하는 기법

레이어 별 정규화

데이터 샘플 단위로 정규화를 실시하는 것

Transformer 또한 심층 신경망이며 학습이 쉽지 않음
안정적인 최적화를 돕기 위해 위의 2가지 기법들이 적용됨

3

Transformer

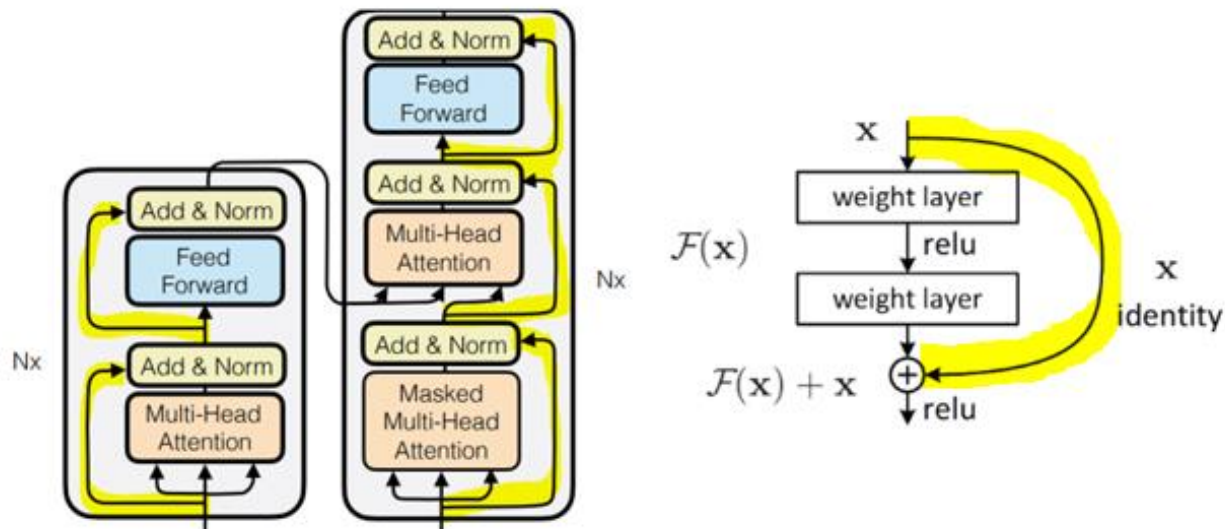
성능 향상 기법

잔차 연결

모델의 입력 값을 복제하여 중간층의 출력 값과 더하는

Identity mapping을 수행하는 기법

등장 배경은 CV팀 2주차 클린업 참고!



3

Transformer

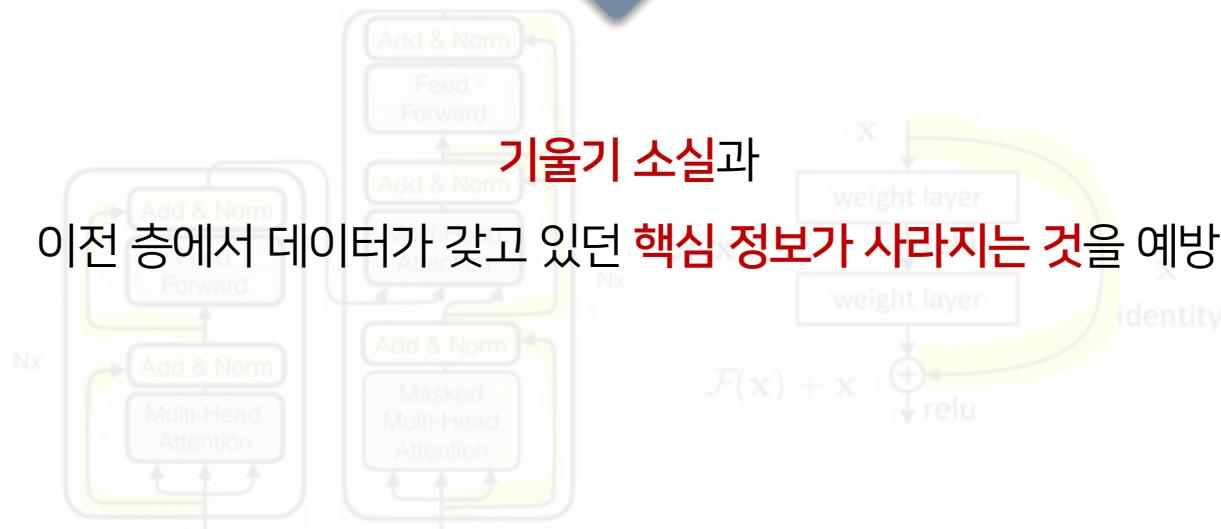
성능 향상 기법

잔차 연결

모델의 입력 값을 복제하여 중간층의 출력 값과 더하는

Identity mapping을 수행하는 기법

등장 배경은 CV팀 2주차 클린업 참고!



성능 향상 기법

레이어 별 정규화

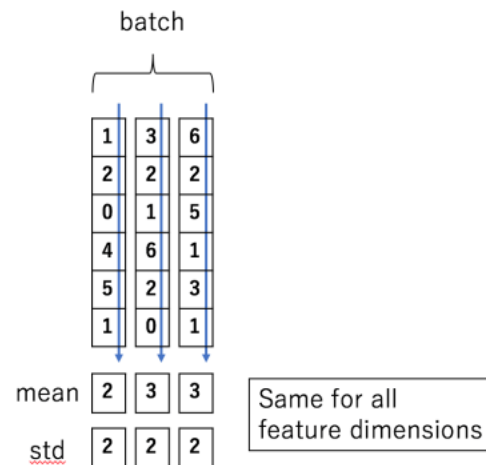
데이터 샘플 단위로 정규화를 실시하는 것

정규화 개념은 딥러닝팀 1주차 클린업 참고!

Batch Normalization



Layer Normalization



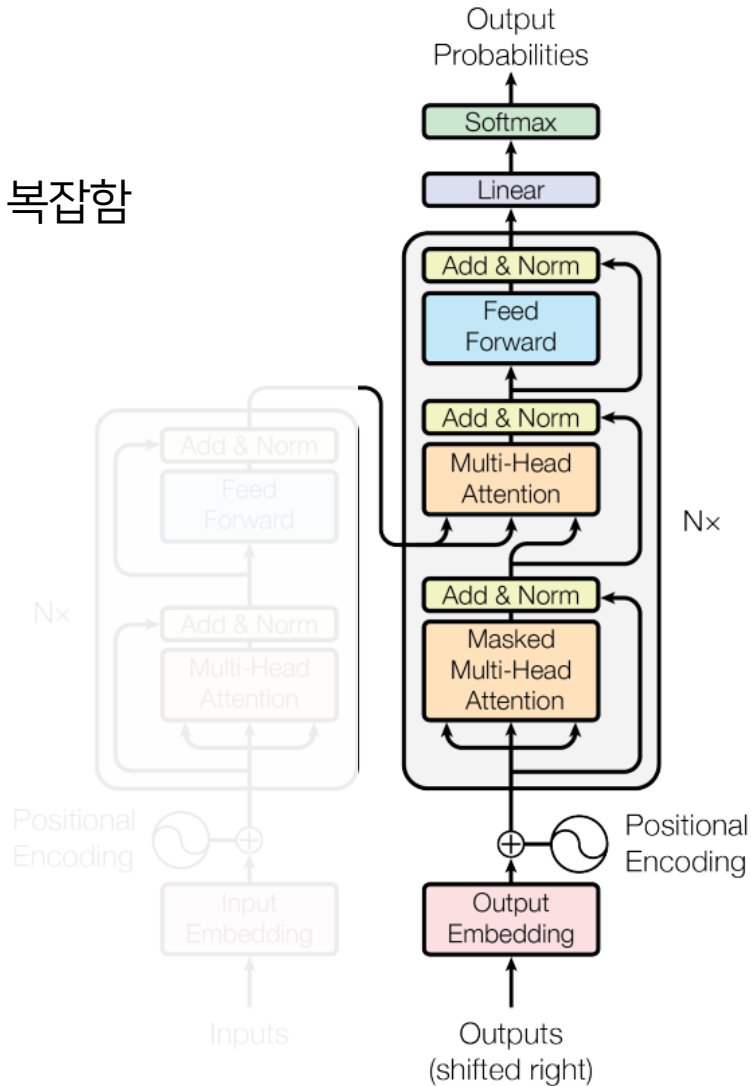
배치 정규화와 마찬가지로 내부 공변량 문제를 해결하는 데 기여

3

Transformer

Decoder

인코더에 비해 그 구조가 복잡함



3

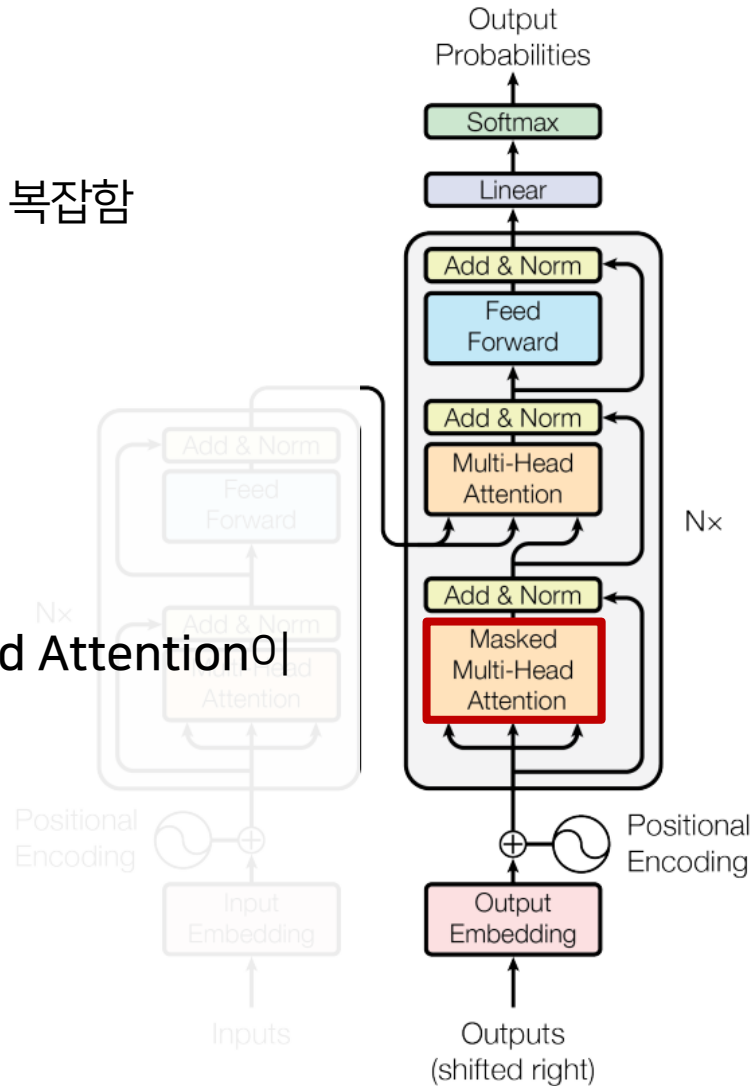
Transformer

Decoder

인코더에 비해 그 구조가 복잡함

우선 Masked Multi-Head Attention이

수행됨



Masked Attention

Masked Attention

기존의 Attention에 **마스킹을 추가**한 형태의 연산

Scaled Scores					Look-Ahead Mask					Masked Scores				
	<sos>	je	suis	étudiant		<sos>	je	suis	étudiant		<sos>	je	suis	étudiant
<sos>	0.7	0.1	0.1	0.1	+	<sos>	0	$-\infty$	$-\infty$	=	<sos>	0.7	$-\infty$	$-\infty$
je	0.1	0.6	0.2	0.1		je	0	0	$-\infty$		je	0.1	0.6	$-\infty$
suis	0.1	0.3	0.6	0.1		suis	0	0	0		suis	0.1	0.3	0.6
étudiant	0.1	0.3	0.3	0.3		étudiant	0	0	0		étudiant	0.1	0.3	0.3

마스킹이란 matrix의 성분 값 일부를 강제로 계산에서 제외시키는 것

Masked Attention

Masked Attention

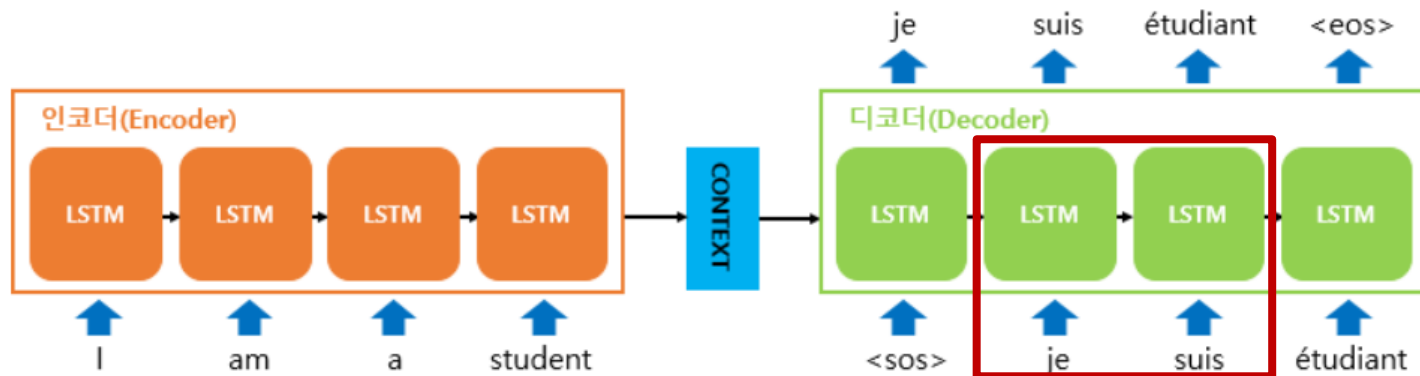
기존의 Attention에 **마스킹을 추가**한 형태의 연산

	Scaled Scores					Look-Ahead Mask					Masked Scores			
	<sos>	je	suis	étudiant		<sos>	je	suis	étudiant		<sos>	je	suis	étudiant
<sos>	0.7	0.1	0.1	0.1	+	<sos>	0	$-\infty$	$-\infty$	=	<sos>	0.7	$-\infty$	$-\infty$
je	0.1	0.6	0.2	0.1		je	0	0	$-\infty$		je	0.1	0.6	$-\infty$
suis	0.1	0.3	0.6	0.1		suis	0	0	0		suis	0.1	0.3	0.6
étudiant	0.1	0.3	0.3	0.3		étudiant	0	0	0		étudiant	0.1	0.3	0.3

왜 마스킹 연산이 필요할까?

Masked Attention

Decoder가 추론을 할 때는 **과거 시점에 주어진 정보만 활용 가능하며**
 훈련 도중 미래 토큰이 과거 토큰 예측에 활용되지 않아야 함

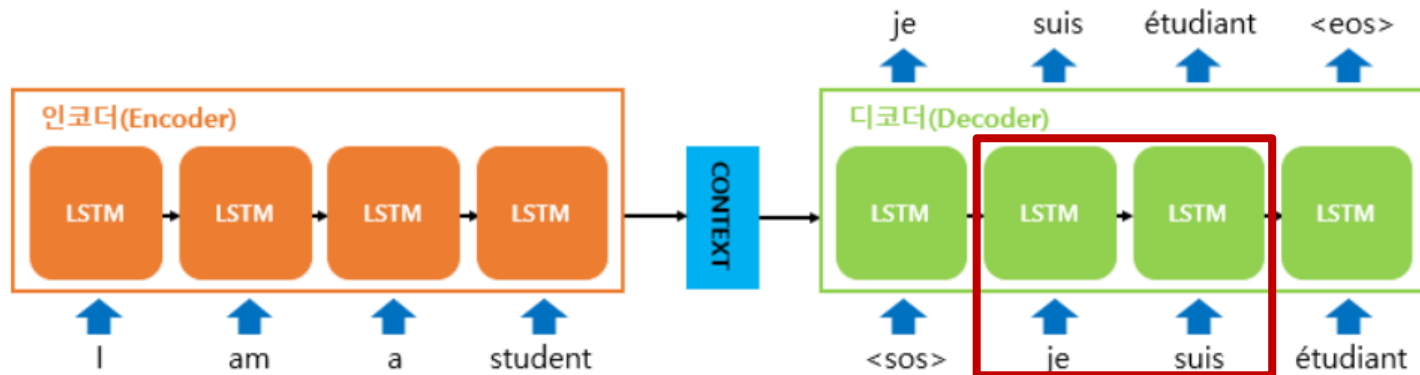


시점 2의 디코딩 스텝에서는 'je', 'suis'만 고려 가능

이때 RNN 기반 Seq2seq은 이미 Auto-regressive 구조를 가짐

Masked Attention

Decoder가 추론을 할 때는 **과거 시점에 주어진 정보만 활용 가능하며**
 훈련 도중 미래 토큰이 과거 토큰 예측에 활용되지 않아야 함



시점 2의 디코딩 스텝에서는 'je', 'suis'만 고려 가능

즉 **교사 강습을 진행해 Target sentence가 한번에 주어져도**
 미래 토큰이 과거 토큰 예측에 쓰일 가능성이 없음

Masked Attention

Scaled Scores

	<sos>	je	suis	étudiant
<sos>	0.7	0.1	0.1	0.1
je	0.1	0.6	0.2	0.1
suis	0.1	0.3	0.6	0.1
étudiant	0.1	0.3	0.3	0.3

+

Look-Ahead Mask

	<sos>	je	suis	étudiant
<sos>	0	0	0	0
je	0	0	0	0
suis	0	0	0	0
étudiant	0	0	0	0

Masked Scores

	<sos>	je	suis	étudiant
<sos>	-∞	-∞	-∞	-∞
je	-∞	-∞	-∞	-∞
suis	-∞	-∞	-∞	-∞
étudiant	0.1	0.3	0.3	0.3

Transformer 디코더의 경우 학습 과정에서
Target sentence가 한번에 주어지면
과거 시점의 query가 미래 시점의 토큰에
가중치를 부여하는 문제가 발생

Masked Attention

Scaled Scores					Look-Ahead Mask					Masked Scores						
	<sos>	je	suis	étudiant		<sos>	je	suis	étudiant		<sos>	je	suis	étudiant		
<sos>	0.7	0.1	0.1	0.1	+	<sos>	0	-∞	-∞	-∞	=	<sos>	0.7	-∞	-∞	-∞
je	0.1	0.6	0.2	0.1		je	0	0	-∞	-∞		je	0.1	0.6	-∞	-∞
suis	0.1	0.3	0.6	0.1		suis	0	0	0	-∞		suis	0.1	0.3	0.6	-∞
étudiant	0.1	0.3	0.3	0.3		étudiant	0	0	0	0		étudiant	0.1	0.3	0.3	0.3

가중치에 $-\infty$ 의 값을 지정하면 softmax 연산으로
미래 시점의 토큰에는 가중치 0이 부여 돼 이를 막을 수 있음!

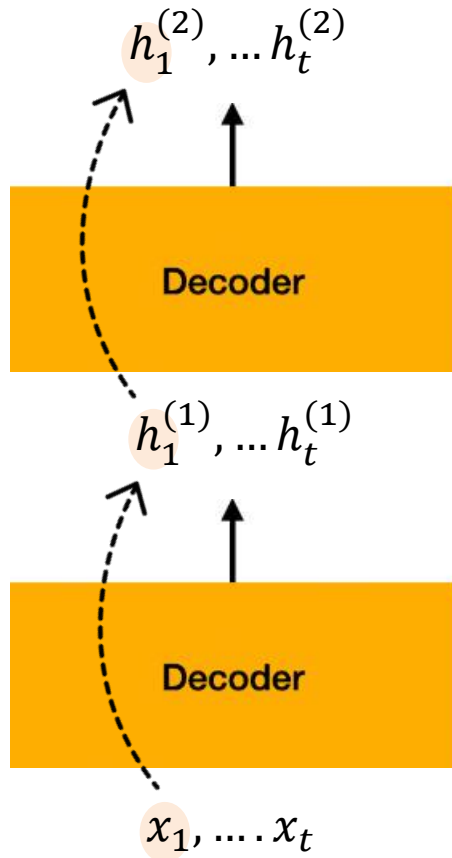
Masked Attention (추론 과정)

Scaled Scores					Look-Ahead Mask					Masked Scores				
	<sos>	je	suis	étudiant		<sos>	je	suis	étudiant		<sos>	je	suis	étudiant
<sos>	0.7	0.1	0.1	0.1	<sos>	0	$-\infty$	$-\infty$	$-\infty$	<sos>	0.7	$-\infty$	$-\infty$	$-\infty$
je	0.1	0.6	0.2	0.1	je	0	0	$-\infty$	$-\infty$	je	0.1	0.6	$-\infty$	$-\infty$
suis	0.1	0.3	0.6	0.1	suis	0	0	0	$-\infty$	suis	0.1	0.3	0.6	$-\infty$
étudiant	0.1	0.3	0.3	0.3	étudiant	0	0	0	0	étudiant	0.1	0.3	0.3	0.3

Ground truth가 한번에 제공되지 않는 추론 과정에서도
이러한 **Masking** 과정이 필요할까?

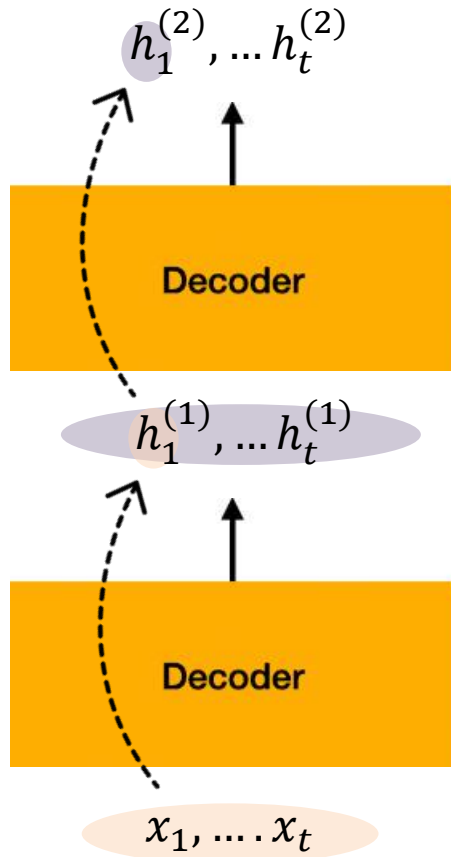
가중치에 $-\infty$ 의 값을 지정하면 softmax 연산으로
미래 시점의 토큰에는 가중치 0이 부여 돼 이를 막을 수 있음!

Masked Attention (추론 과정)



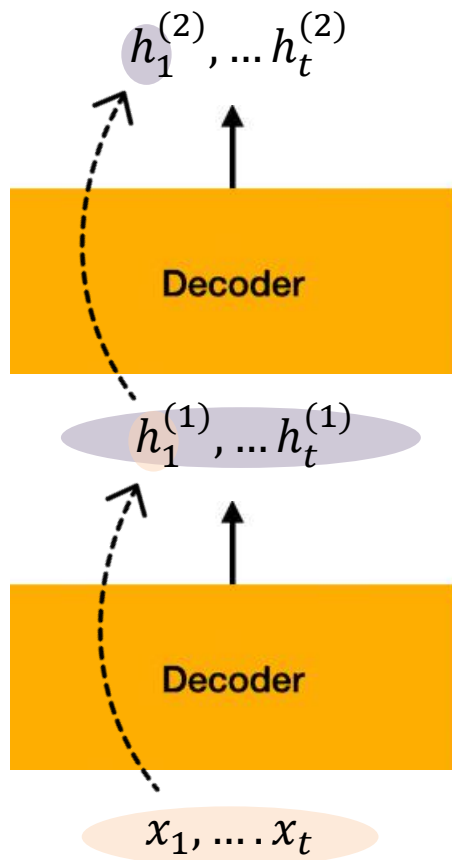
마스킹을 사용하는 훈련 과정에서
각 디코더 블록의 출력 값들은 미래 시점의
값들을 참조할 수 없었음

Masked Attention (추론 과정)



추론 과정에서 **마스킹을 해제할 경우**
각 디코더 블록의 출력 값들은 미래 시점의
값들을 참조할 수 있게 됨

Masked Attention (추론 과정)



학습과 추론 과정 간 **Data의 분포 차이**가
발생할 가능성이 존재

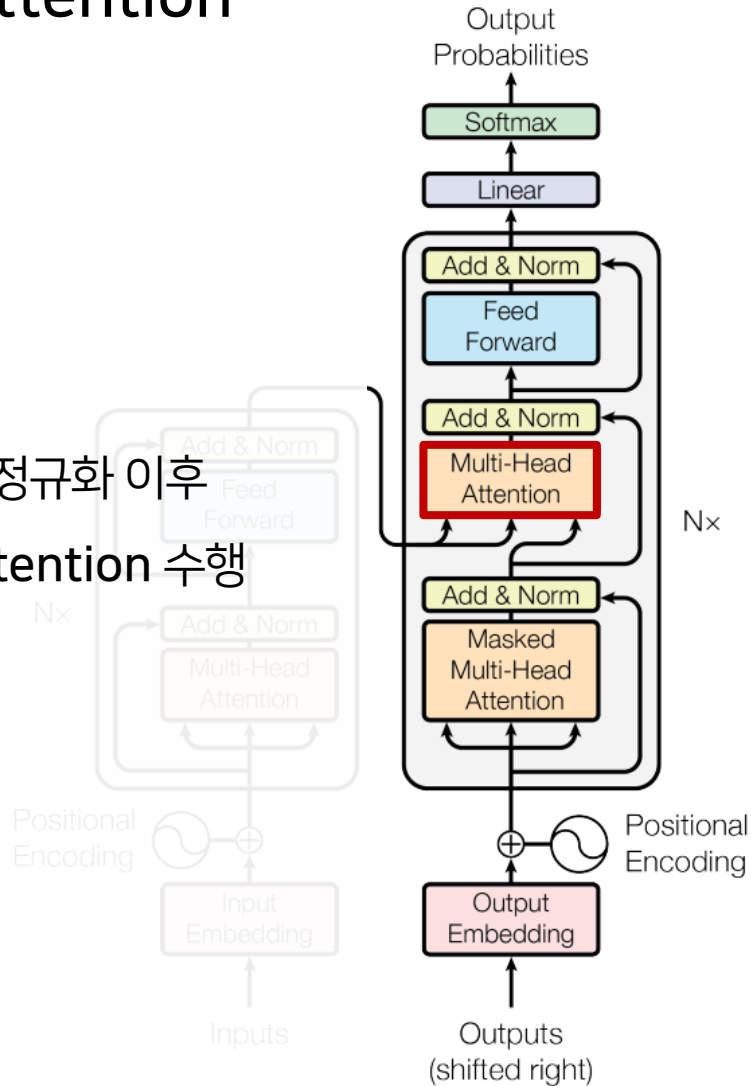
따라서 추론 과정에서도 마스킹을 적용
(디코더 마지막 출력층은 제외 가능)

3

Transformer

Encoder-Decoder Attention

추가적인 잔차 연결 & 정규화 이후
Encoder-Decoder attention 수행



Encoder-Decoder Attention

Query source

Decoder 블록의 첫 두 연산을 수행한 결과 값

Masked Multi-head attention과 Add & Norm

Key, Value source

Encoder 마지막 층의 출력 값

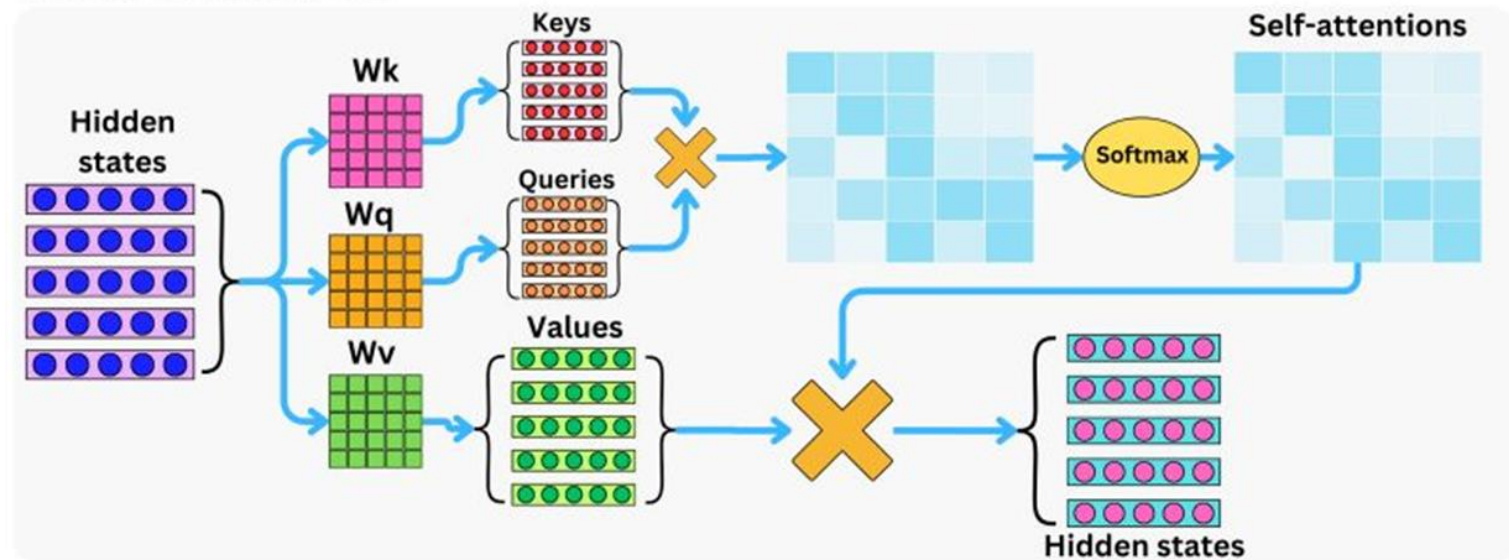
Self-attention이 아닌 **Cross Attention**을 수행

3

Transformer

Encoder-Decoder Attention

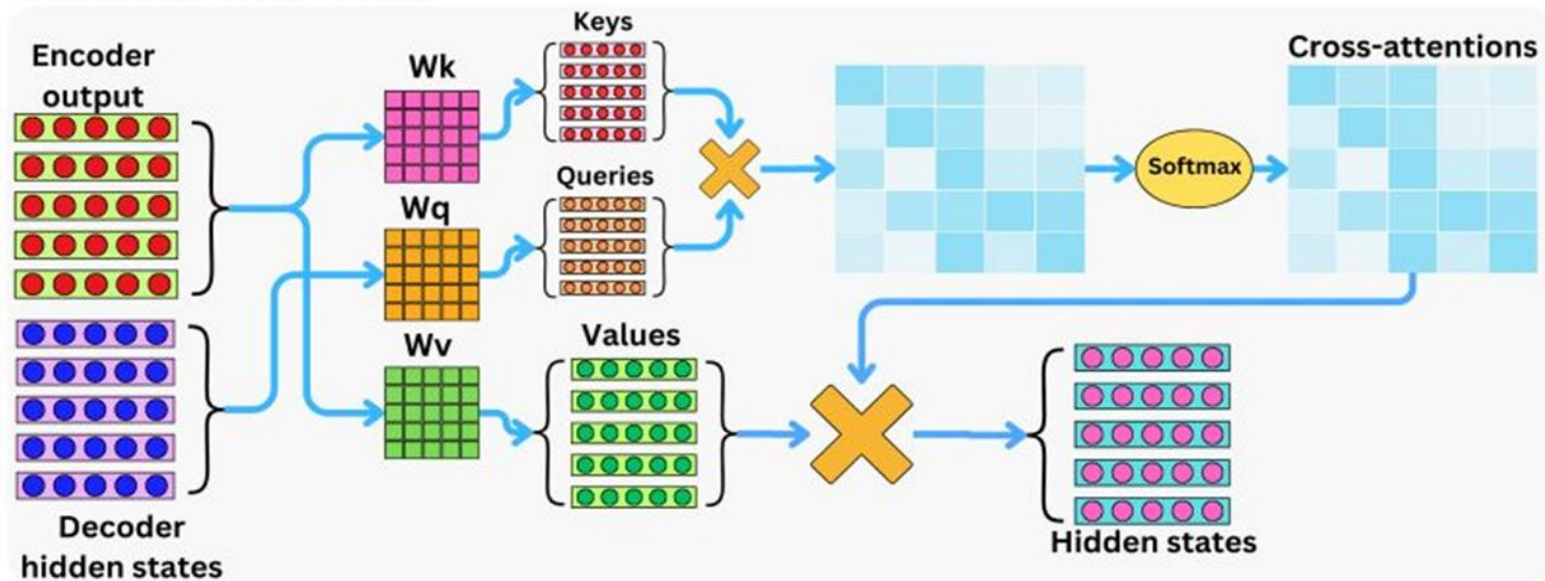
The Self-Attentions



Self Attention : Query source = Key source = Value source

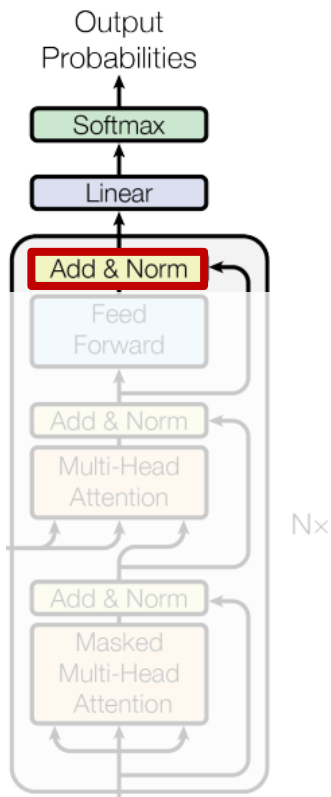
Encoder-Decoder Attention

The Cross-Attentions



Cross-Attention: Query source \neq Key source = Value Source

Linear & Softmax



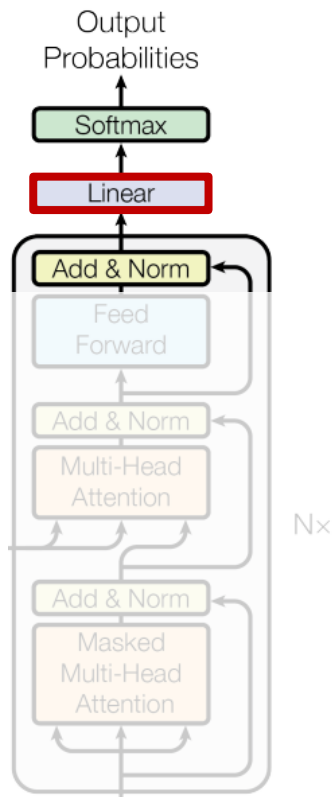
훈련 과정에서는 **모든 시점에서의 Decoder output vector**가 병렬 연산을 통해 동시에 계산됨

그 벡터들을 행 벡터로 삼는 행렬을 D 라고 할 때 shape은

$$N_{target} \times d_{model}$$

Target sentence 속 토큰의 개수

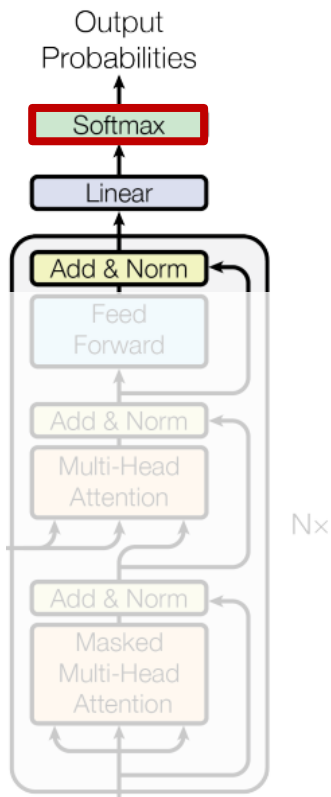
Linear & Softmax



Linear Layer에서는 DW_{emb}^T 를 계산함
 이때 W_{emb}^T 는 Input embedding 시 사용한 행렬의 전치

그후 $N_{target} \times |V|$ 의 차원을 갖는 행렬이 도출됨
 (단어집합의 크기)

Linear & Softmax



행렬 전체가 Softmax를 통과 후 $N_{target} \times |V|$ 의 shape을 갖는

각 디코딩 시점에 대한 확률 벡터들의 집합이 계산됨

Seq2seq에서 BPTT로 역전파를 한 것과 동일하게 진행
(Ground truth와의 Cross entropy loss 계산...)

Linear & Softmax



추론 과정에서의 Decoder 연산

훈련 과정에서는 교사 강습을 사용하므로
 행렬 전체가 Softmax를 통과 후 $N_{target} \times |V|$ 의 shape을 갖는
 디코더의 Query 전체가 **matrix** 단위로 동시에 처리될 수 있음
 각 디코딩 시점에 대한 확률 벡터들의 집합이 계산됨



Seq2seq에서 BPTT로 역전파를 한 것과 동일하게 진행

(Ground truth와의 Cross entropy loss 계산...)

추론 과정에서는 Sep2seq 모델이 하던 것처럼

토큰을 하나씩 추출 후 다음 시점의 입력으로 넣는 과정이 반복됨

디코더의 연산이 병렬화 되지 않고 순전파가 여러 번 진행됨



THANK Y..



4

늘 화목한 늘피

팀원 전원이 T이지만 따스하기 그지없는 늘피 ㅎㅎ



갑자기 이상한
생각이 든 팀장님..

박상훈
나 갑자기 그런 생각이 들었어
질문 들어오면
팀장이 아닌 기존이 나가서 답해주면
더 맛있지 않을까? 오후 6:56

박상훈
@곽동길 오후 6:56

박윤아
ㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋ 오후 6:56

박상훈
ㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋㅋ 오후 6:56

갑자기
그런 끔찍한 생각을 하다니
오후 6:56

많이 피곤한가보구나 상훈아
오후 6:56

김수진

늦은 시간에 죄송합니다당 패키지 1번에 5에서 행정동을 따로 지정해서 union하는 게 아니라 4, 5월 데이터에 각각 월 추가하고 그냥 데이터 전체를 query로 union하면 되는 건가여

오전 1:11

신민서

전 이게했어여

오후 1:12

김수진

r에서저렴 열 굴라서 합치는 거 아니고 강 저렇게 돌리면 되는 거 맞져

오후 1:12

신민서

김수진에게 답장
r에서저렴 열 굴라서 합치는 거 아니고 강 저렇게 돌리면 되는 거 맞져
열을 고르는게 무슨 말인지 약간 모르겠는데 열은 고르지 않았어요 ㅎㅎ

오후 1:13

김수진

r에서 데이터프레임 열 굴라서 합치는 게 잊었던 거 같은데 자체히는 기억이 안나서

신민서에게 답장
열을 고르는게 무슨 말인지 약간 모르겠는데 열은 고르지 않았어요 ㅎㅎ
앗사 그럼 맞았다 생각하고 자러가야지

오후 1:13

가사합니다

새벽까지 꺼지지
않는 늘피의 열정..