Transformer – (1)

일단 코드 없이 Transformer의 콘셉트만 이해 해봅시다!

references

세미나를 시작하기 전에 참고자료들을 밝힙니다. 좋은 자료에 감사드립니다.

- 딥러닝을 이용한 자연어 처리 입문
- [NLP 논문 구현] pytorch로 구현하는 Transformer (Attention is All You Need)
- [Transformer]-1 Positional Encoding은 왜 그렇게 생겼을까? 이유

Tranformer의 핵심 콘셉트

- Attention
  - Self-Attention
  - Multi-head Attention
- Remove RNN and Parallel Processing Techniques
  - Why remove RNN?
  - Positional Encoding
  - Masking

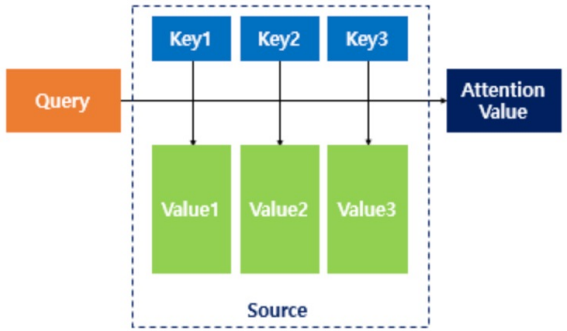
- Self-Attention
- Multi-head Attention

Attention

(review) Attention

Attention = 넓은 범위의 전체 데이터에서 **특정 부분에 집중**하겠다

Key-Value 자료형에 대한 이해를 가지고 어텐션 함수에 대해서 설명해보겠습니다.



1. Query가 입력됨
2. Query와 Key 사이의 **유사도**를 계산
3. 유사도를 기준으로 가장 비슷한 Value를 리턴
    1. 이때 attention distribution으로 weight sum
4. Query와 유사하다고 판단되는 Attention Value 얻음!

어텐션을 함수로 표현하면 주로 다음과 같이 표현됩니다.

Attention(Q, K, V) = Attention Value

어텐션 함수는 주어진 '쿼리(Query)'에 대해서 모든 '키(Key)'와의 유사도를 각각 구합니다. 그리고 구해낸 이 유사도를 키와 맵핑되어있는 각각의 '값(Value)'에 반영해줍니다. 그리고 유사도가 반영된 '값(Value)'을 모두 더해서 리턴합니다. 여기서는 이를 어텐션 값(Attention Value)이라고 하겠습니다.

지금부터 배우게 되는 seq2seq + 어텐션 모델에서 Q, K, V에 해당되는 각각의 Query, Keys, Values는 각각 다음과 같습니다.

```
Q = Query : t 시점의 디코더 셀에서의 은닉 상태
K = Keys : 모든 시점의 인코더 셀의 은닉 상태들
V = Values : 모든 시점의 인코더 셀의 은닉 상태들
```

간단한 어텐션 예제를 통해 어텐션을 이해해보겠습니다.

Self-Attention

Attention = 넓은 범위의 전체 데이터에서 **특정 부분에 집중**하겠다

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

위 문장에서 'it'은 무엇을 지칭하는 것일까? 사람이라면 직관적으로 'animal'과 연결지을 수 있지만, 컴퓨터는 'it'이 'animal'을 가리키는지, 'street'를 가리키는지 알지 못한다.

Self-Attention은 이러한 문제를 해결하기 위해 **같은 문장 내에서 두 token 사이의 연관성을 찾아내는 방법론**이다. Self가 붙는 이유는 문장 내에서 (같은 문장 내의 다른 token에 대한) Attention을 구하기 때문이다.

Self-Attention

Attention = 넓은 범위의 전체 데이터에서 **특정 부분에 집중**하겠다

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

위 문장에서 'it'은 무엇을 지칭하는 것일까? 사람이라면 직관적으로 'animal'과 연결지을 수 있지만, 컴퓨터는 'it'이 'animal'을 가리키는지, 'street'를 가리키는지 알지 못한다.

Self-Attention은 이러한 문제를 해결하기 위해 **같은 문장 내에서 두 token 사이의 연관성을 찾아내는 방법론**이다. Self가 붙는 이유는 문장 내에서 (같은 문장 내의 다른 token에 대한) Attention을 구하기 때문이다.

Query: embedding vector of 'it'

Key: embedding vector of other tokens in sentence

Value: embedding vector of other tokens in sentence

Self-Attention

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

Self-Attention은 이러한 문제를 해결하기 위해 **같은 문장 내에서 두 token 사이의 연관성을 찾아내는 방법론**이다. Self가 붙는 이유는 문장 내에서 (같은 문장 내의 다른 token에 대한) Attention을 구하기 때문이다.

Query: embedding vector of 'it'

Key: embedding vector of other tokens in sentence

Value: ~~embedding vector of other tokens in sentence~~

Query: embedding vector of 'it'

Key: embedding vector of other tokens in sentence

Value: token의 embedding vector를 fc layer로 가공한 vector

Key와 Value로 같은 값의 벡터를 쓰지 않을 수 있다!

Self-Attention

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

Self-Attention은 이러한 문제를 해결하기 위해 **같은 문장 내에서 두 token 사이의 연관성을 찾아내는 방법론**이다. Self가 붙는 이유는 문장 내에서 (같은 문장 내의 다른 token에 대한) Attention을 구하기 때문이다.

Query: embedding vector of 'it'

Key: embedding vector of other tokens in sentence

Value: ~~embedding vector of other tokens in sentence~~

Query: 'it' token의 embedding vector를 fc layer로 가공한 vector

Key: token의 embedding vector를 fc layer로 가공한 vector

Value: token의 embedding vector를 fc layer로 가공한 vector

단, Q, K, V를 가공하는 fc layer는 모두 다르다!

Q, K, V 각각을 위한 3개의 fc layer가 있는 것!

Self-Attention

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

같은 문장 내에서 두 token 사이의 연관성을 찾아내는 방법론

Query: 'it' token의 embedding vector를 fc layer로 가공한 vector

Key: e token의 embedding vector를 fc layer로 가공한 vector

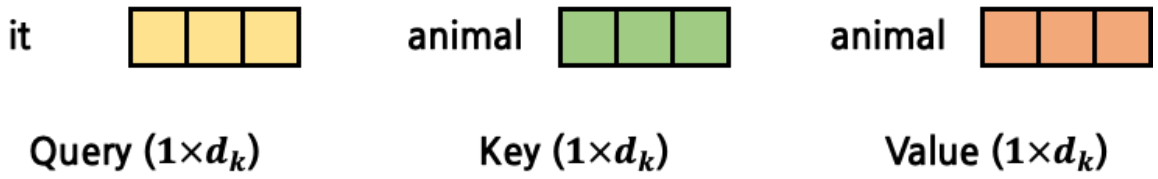
Value: token의 embedding vector를 fc layer로 가공한 vector

단, Q, K, V를 가공하는 fc layer는 모두 다르다!

Q, K, V 각각을 위한 3개의 fc layer가 있는 것!

token embedding vector의 크기: d_{embed}

fc layer로 가공한 후 vector의 크기: d_k



Self-Attention

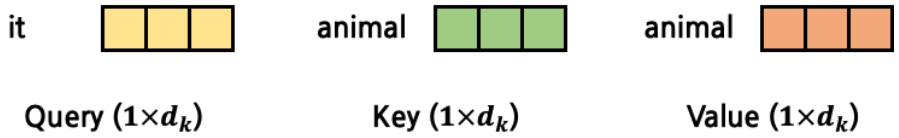
*The animal didn't cross the street, because **it** was too tired.*

Query: 'it' token의 embedding vector를 fc layer로 가공한 vector

Key: token의 embedding vector를 fc layer로 가공한 vector

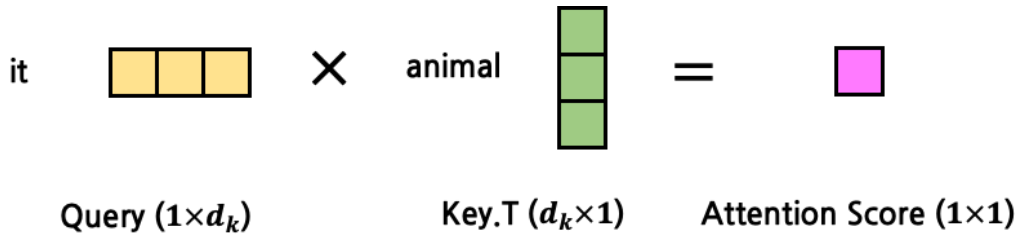
Value: token의 embedding vector를 fc layer로 가공한 vector

Q, K, V 각각을 위한 3개의 fc layer가 있는 것!



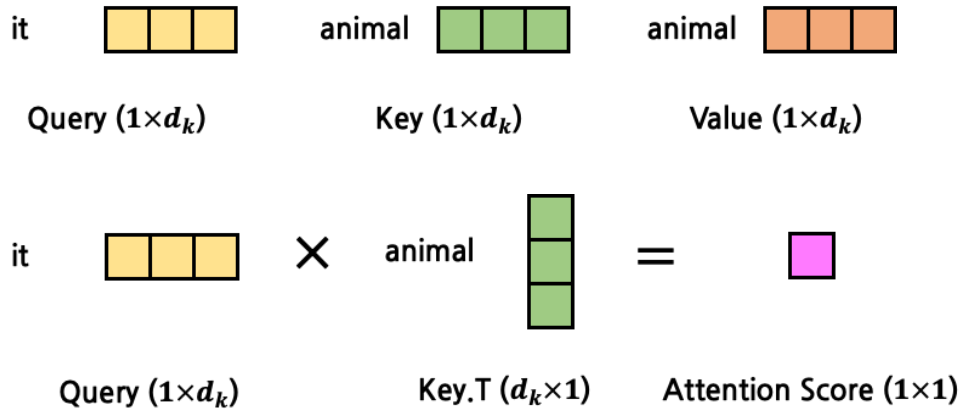
token embedding vector의 크기: d_{embed}

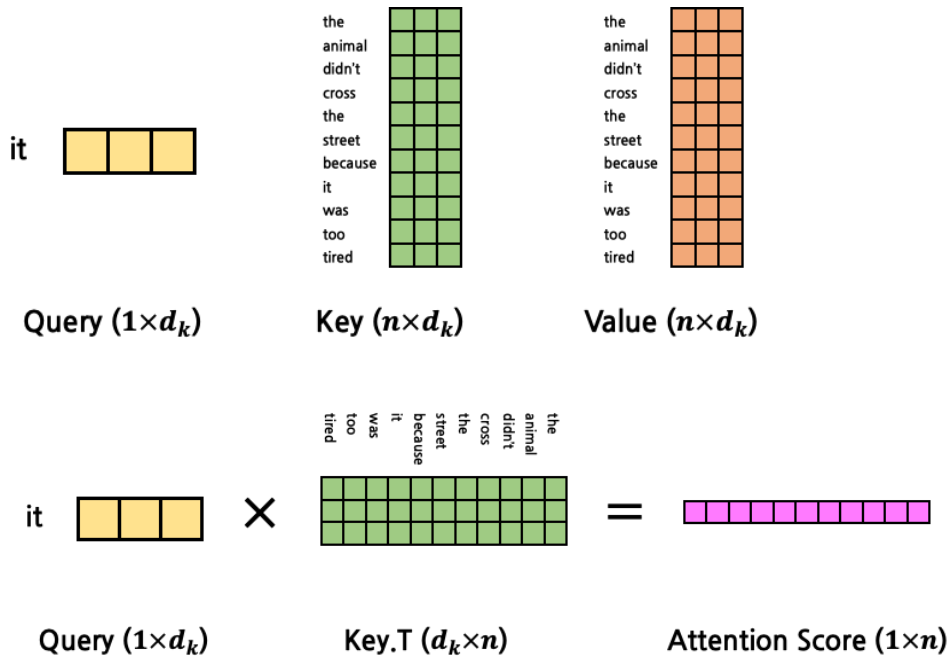
fc layer로 가공한 후 vector의 크기: d_k



attention score는 평소대로 dot-product으로 계산

Self-Attention





token embedding vector의 크기: d_{embed}

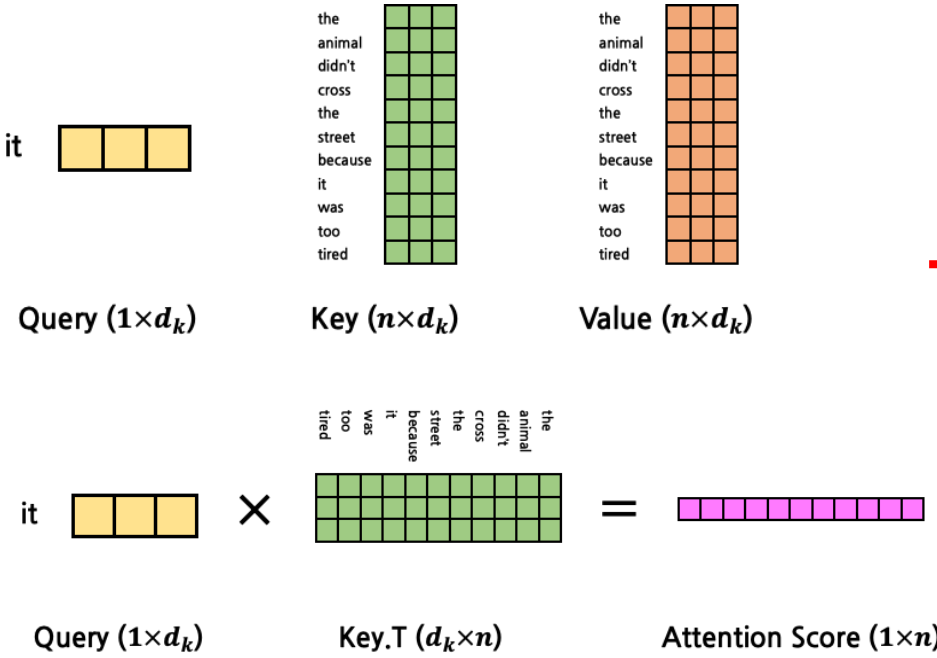
fc layer로 가공한 후 vector의 크기: d_k

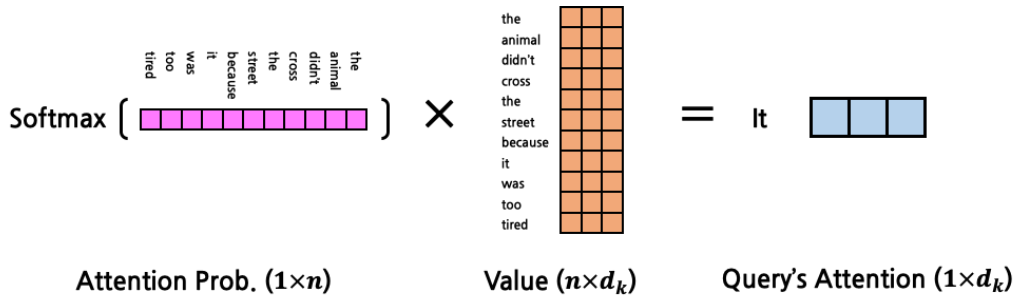
문장에 존재하는 n개 토큰에 대해

한번에 self-attention을 수행하고 싶다!

행렬곱으로 쉽게 가능!

Self-Attention





문장에 존재하는 n개 토큰에 대해

한번에 self-attention을 수행하고 싶다!

<u>행렬곱</u>으로 쉽게 가능!

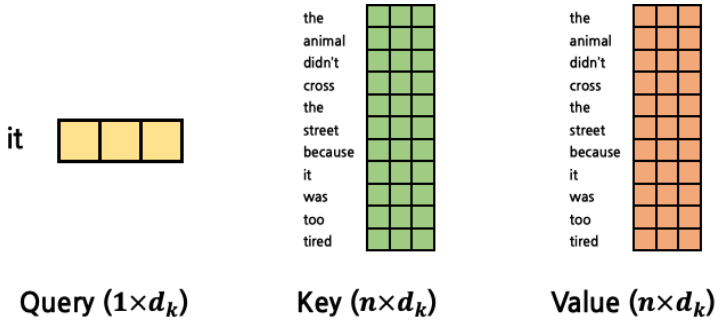
Attention Score -> (softmax) -> Attention Prob

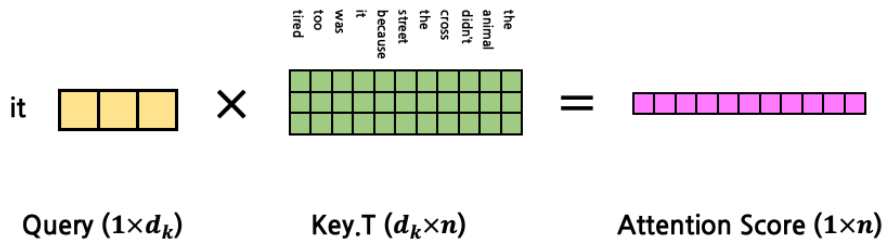
Value와 행렬곱(= weight sum)해서

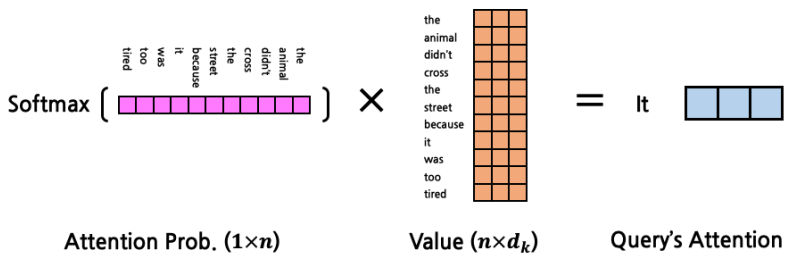
Query의 Attention Value 계산!

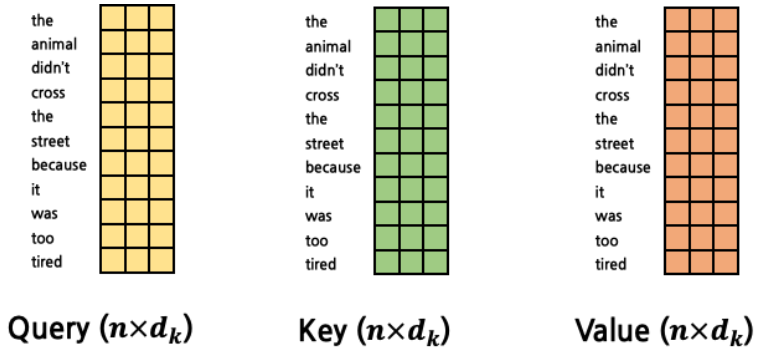
ps. 여기까진 직전 세미나의 Attention과 동일!

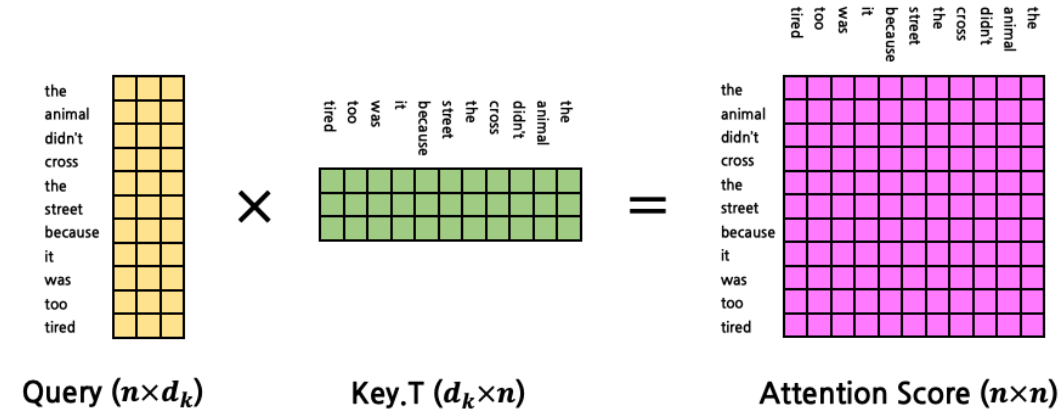
Self-Attention

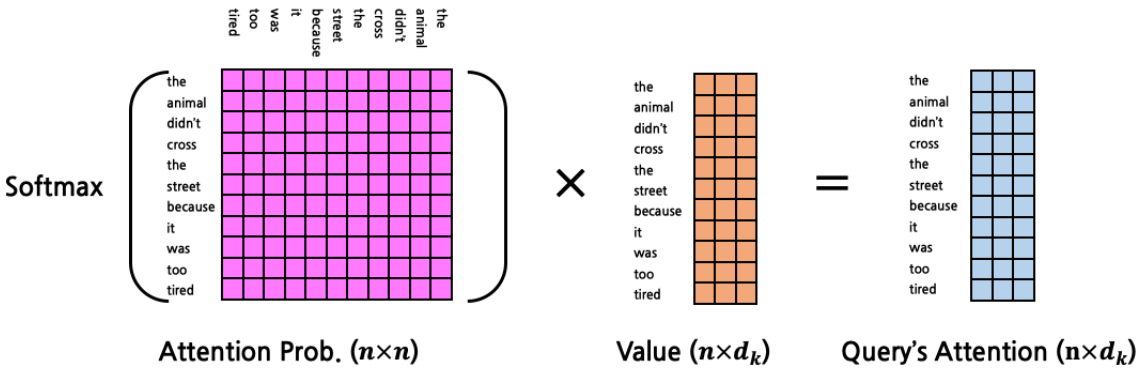






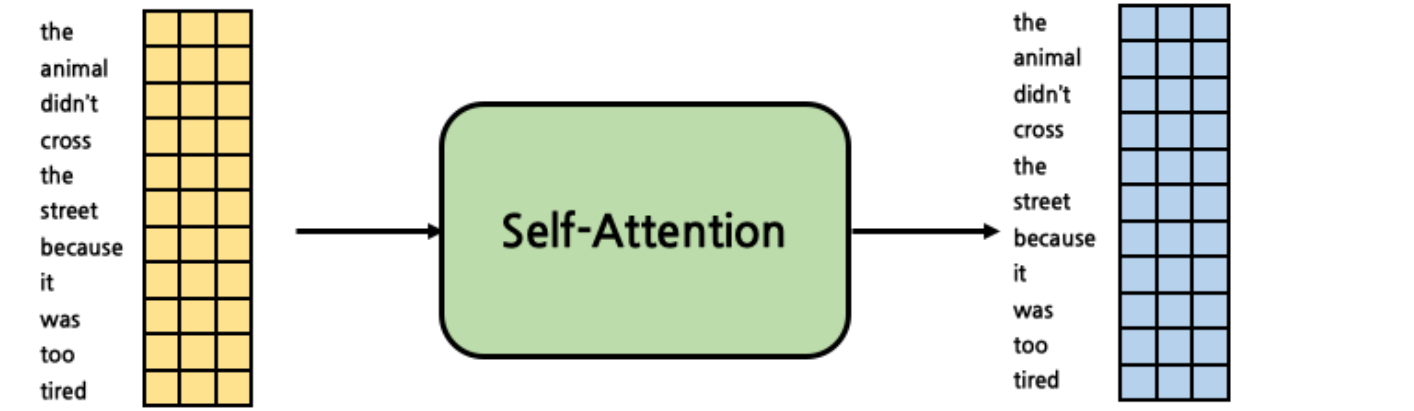






입력 문장의 n개 토큰에 대해 한번에 Attention 계산!

Self-Attention

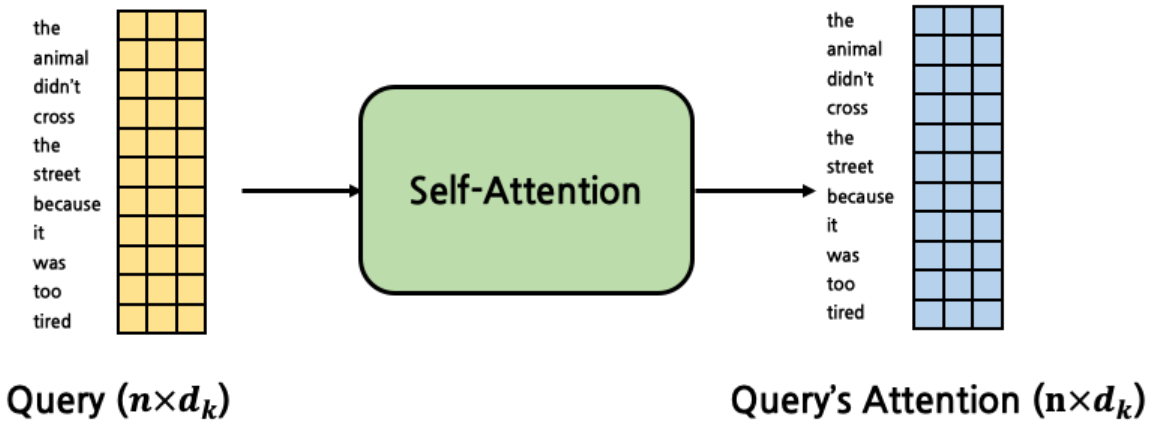


$$n \times d_k \rightarrow n \times d_k$$

self-attention 함수는

input Query의 shape를 보존한다.

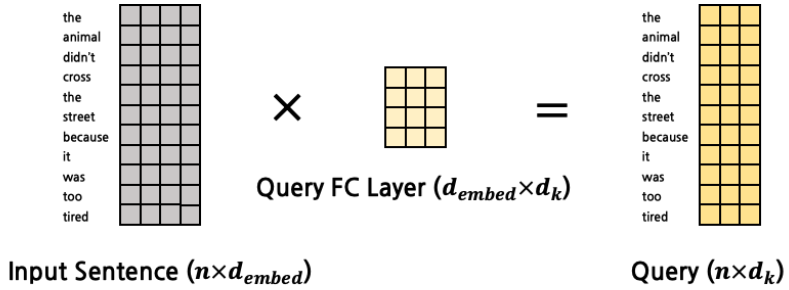
Self-Attention

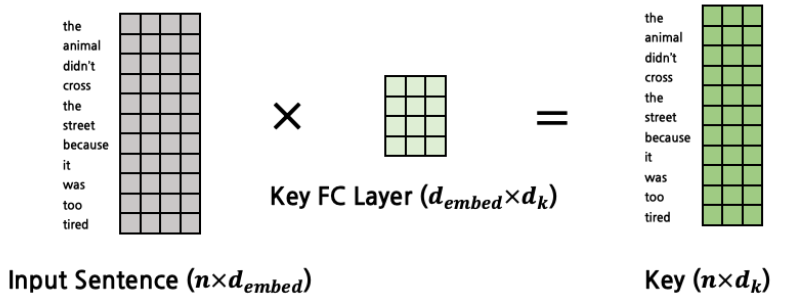


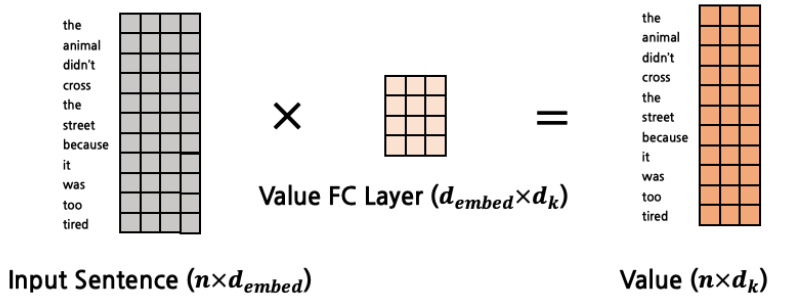
$n \times d_k \rightarrow n \times d_k$

self-attention 함수는

input Query의 shape를 보존한다.



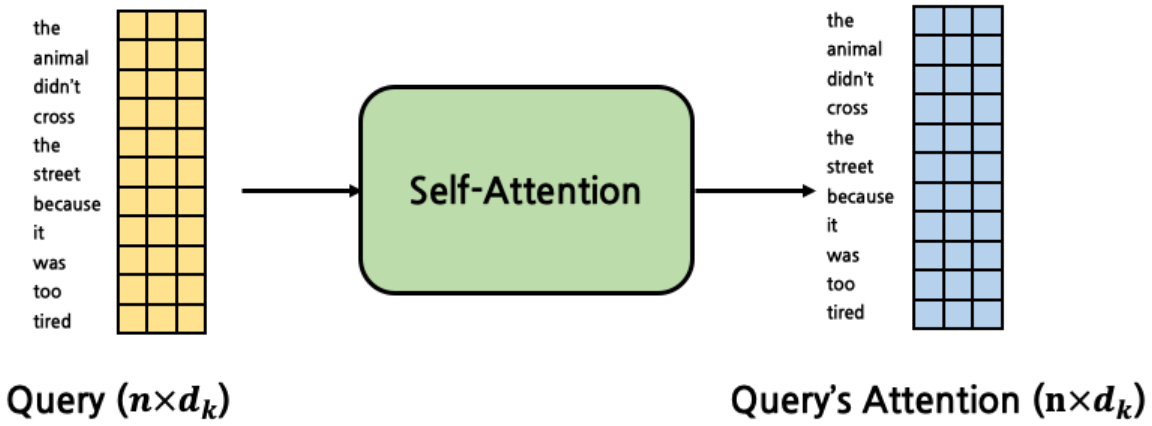




Self-Attention에 입력되는

Q, K, V는 각각의 fc layer를 거쳐서 가공된다.

Self-Attention



$n \times d_k \rightarrow n \times d_k$

self-attention 함수는

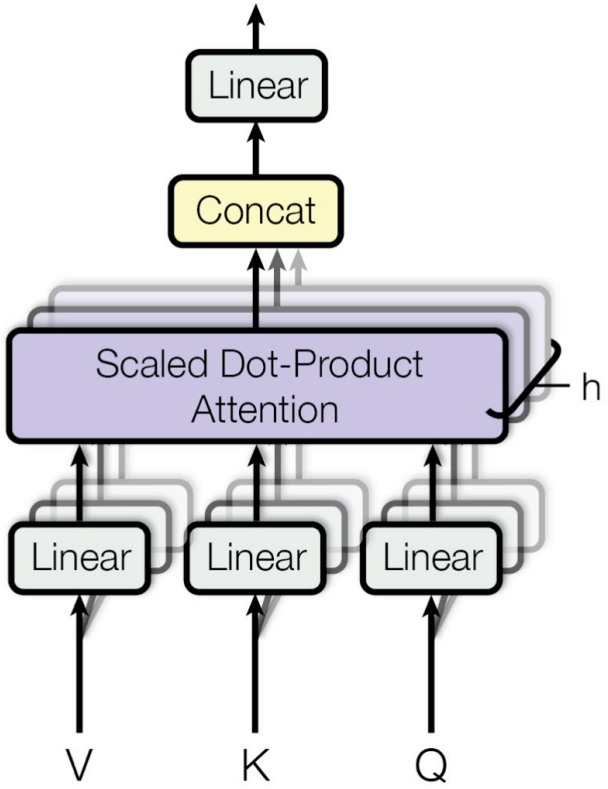
input Query의 shape를 보존한다.

*** scaled-dot Attention**

$$\text{Query's Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

기존 dot-product attention에서
중간에 scaling이 들어간 것 뿐!

Multi-head Attention



Attention을 한 번만 하는 것보단

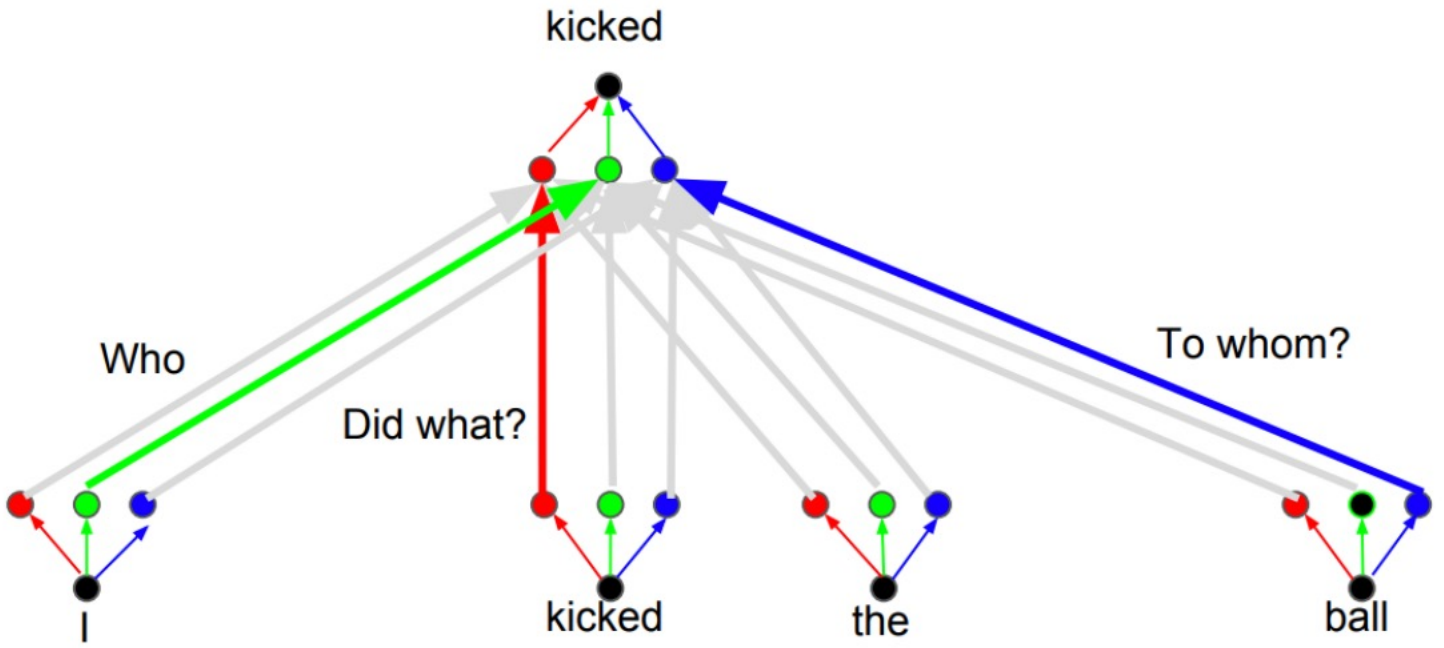
서로 다른 Attention을 h번 수행해서 그걸

종합하는게 좋지 않을까?

Why?

"여러 Attention을 잘 반영하기 위해서이다. 만약 하나의 Attention만 반영한다고 했을 때, 예시 문장에서 'it'의 Attention에는 'animal'의 것이 대부분을 차지하게 될 것이다. 하지만 여러 종류의 attention을 반영한다고 했을 때 'tired'에 집중한 Attention까지 반영된다면, 최종적인 'it'의 Attention에는 'animal'을 지칭한다는 정보, 'tired' 상태라는 정보까지 **모두** 담기게 될 것이다. 이 것이 Multi-Head Attention을 사용하는 이유이다."

Multi-head Attention

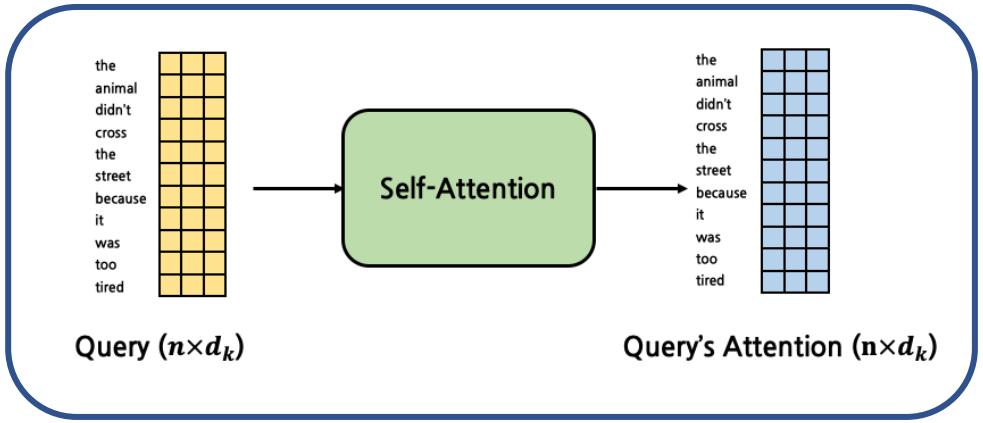


Multi-head Attention

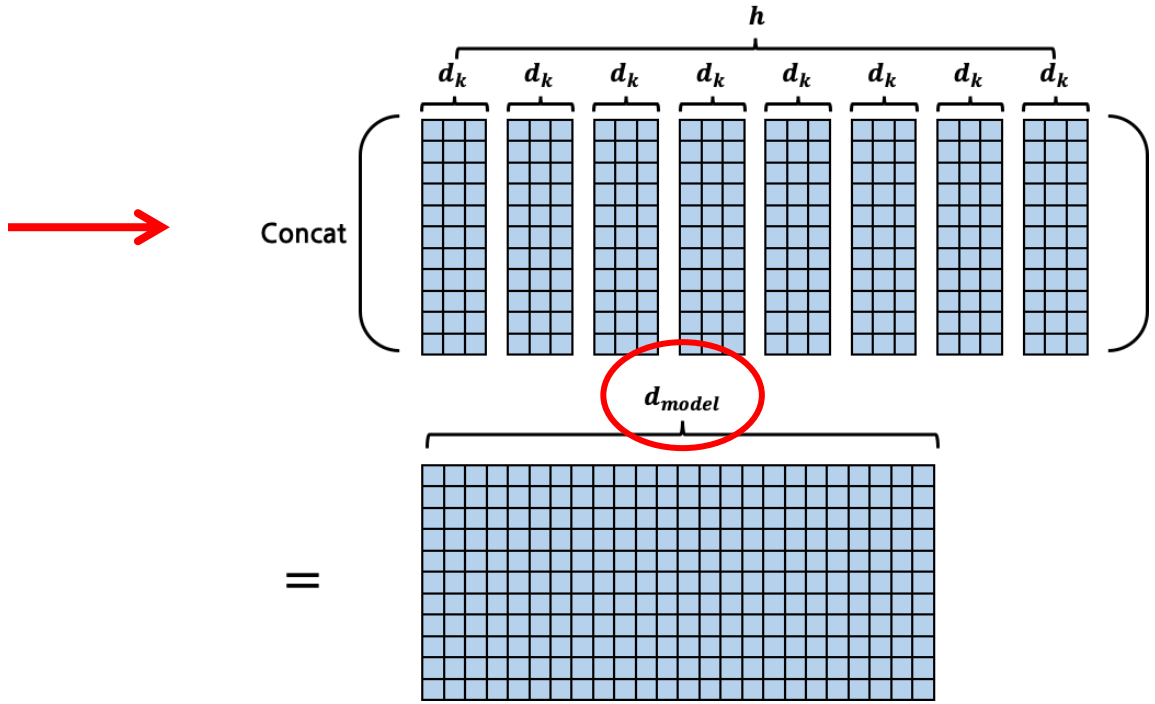
Attention을 한 번만 하는 것보단

서로 다른 Attention을 h번 수행해서 그걸

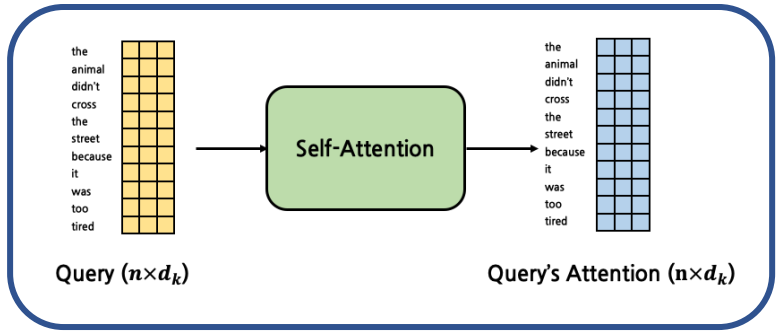
종합하는게 좋지 않을까?



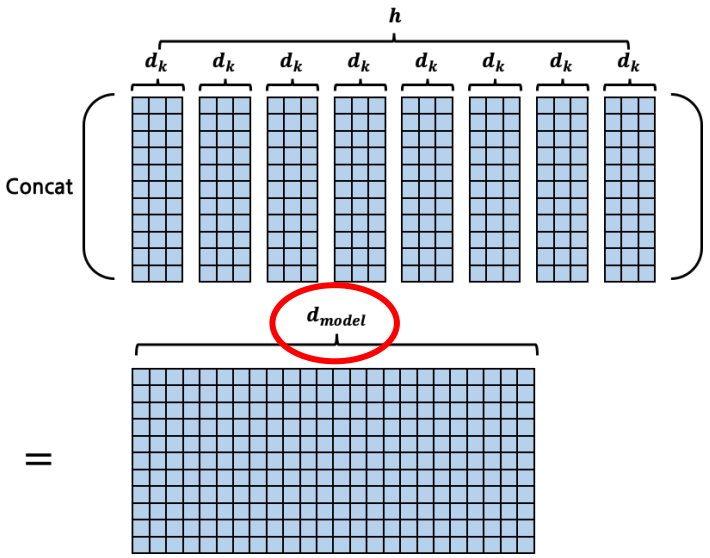
x h번 Attention을 수행!



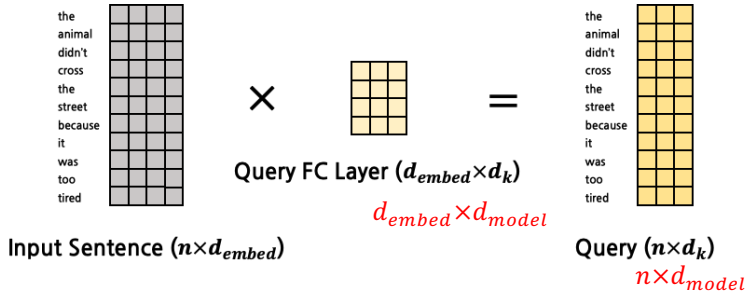
Multi-head Attention

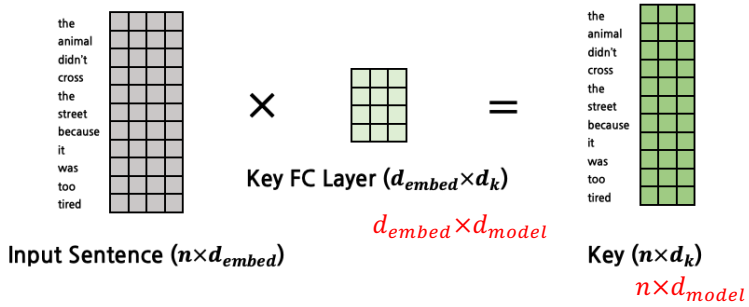


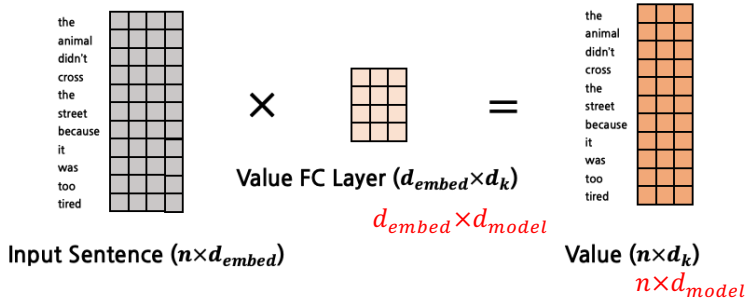
x h번 Attention을 수행!



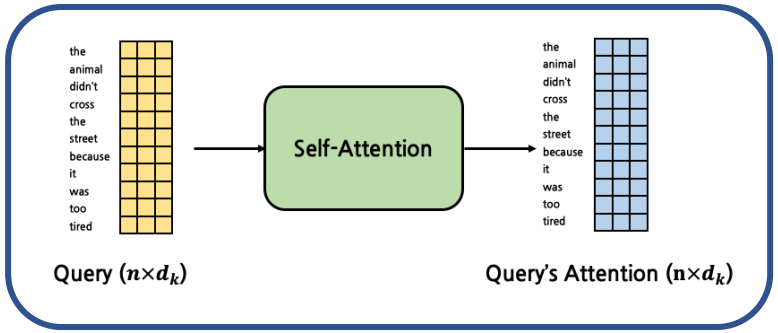
for문으로 h번 도는 것 대신 **<u>좀더 병렬적으로</u>** 만들어보자!



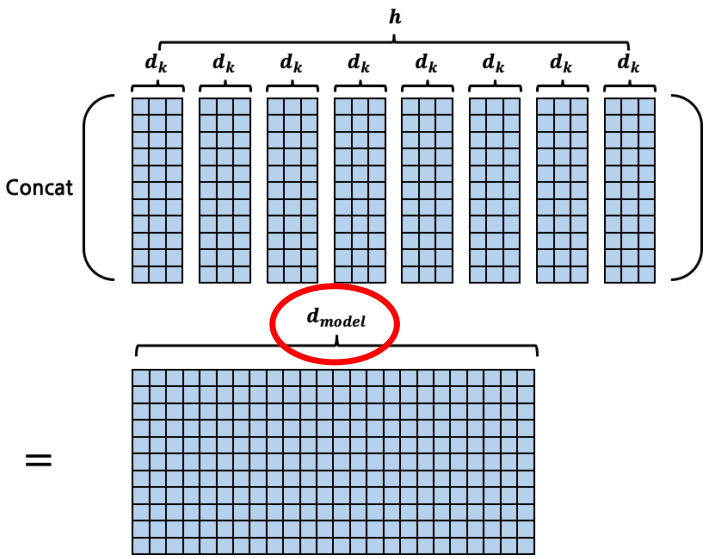




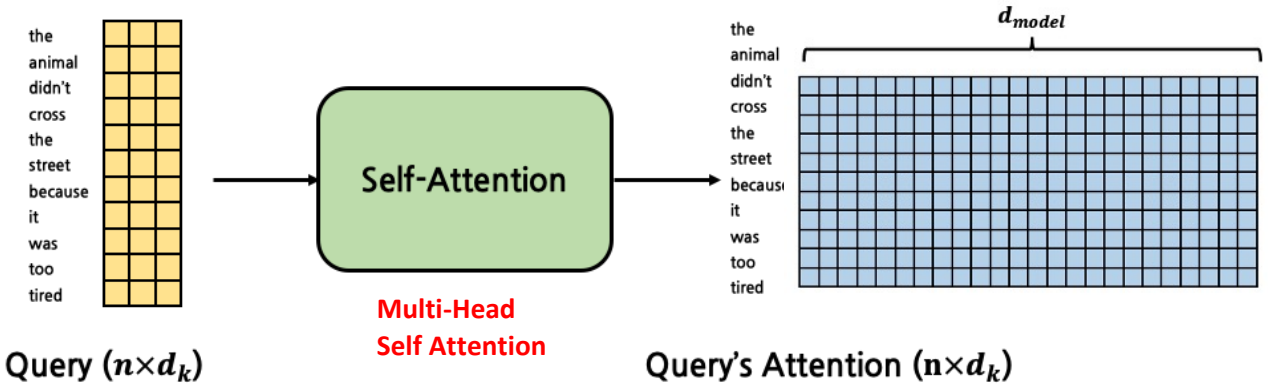
Multi-head Attention



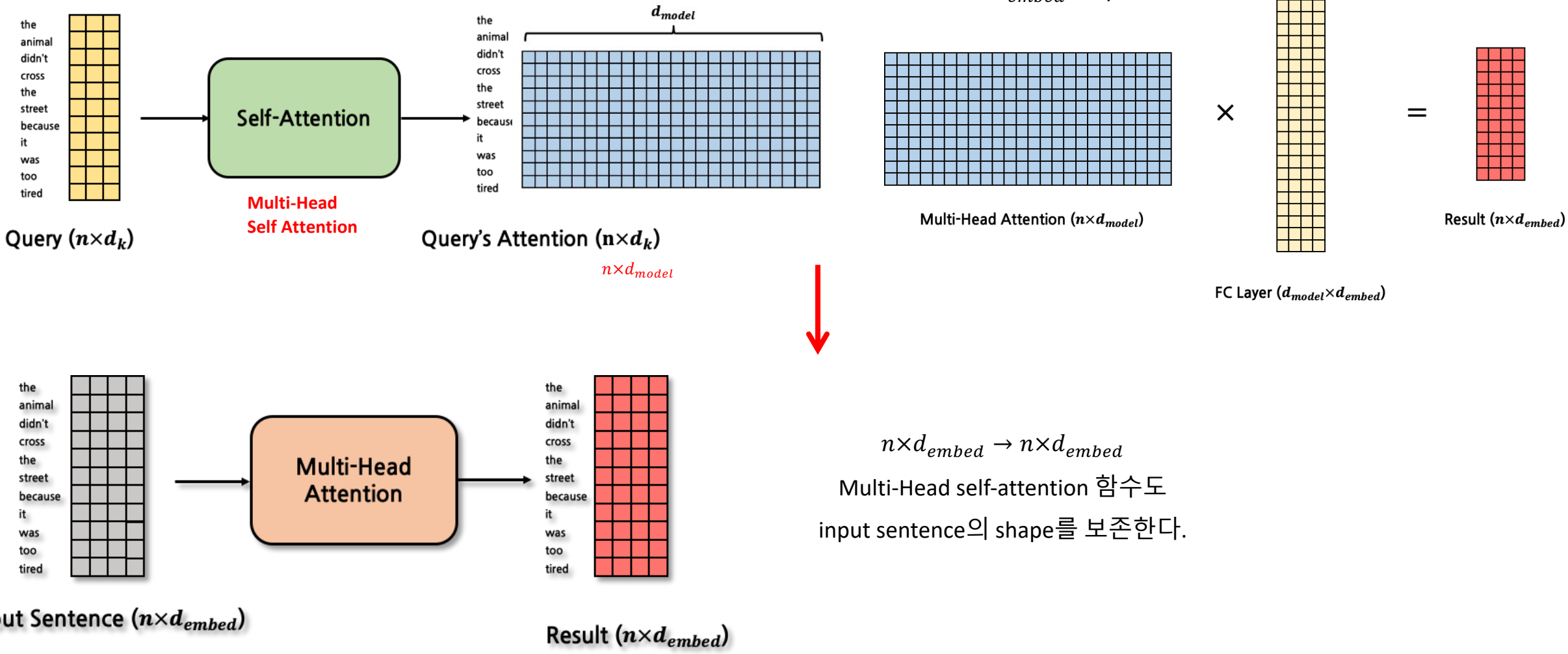
x h번 Attention을 수행!



for문으로 h번 도는 것 대신 **<u>좀더 병렬적으로</u>** 만들어보자!



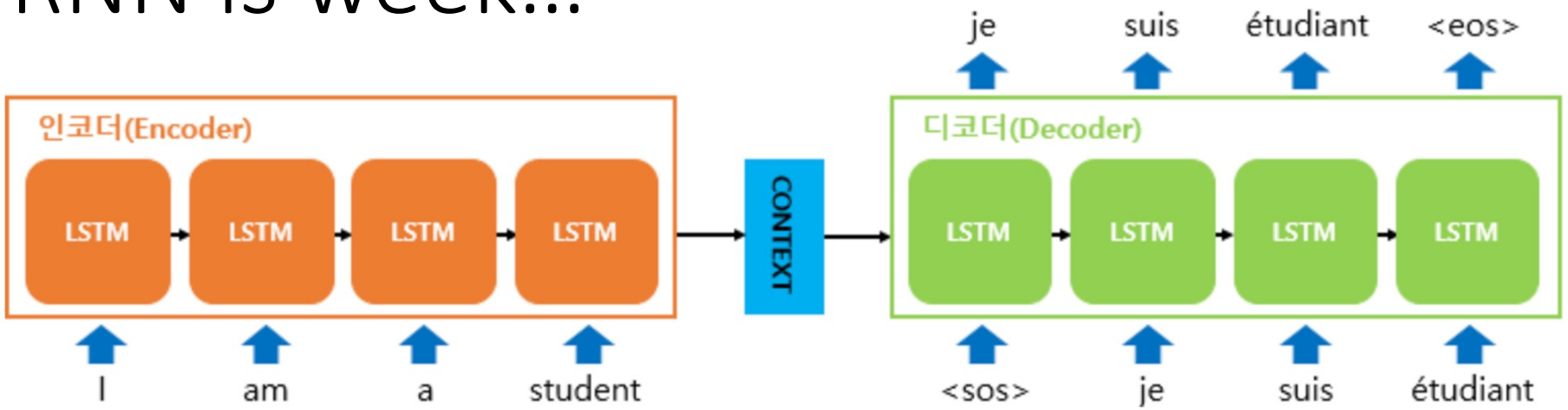
Multi-head Attention



- Why remove RNN?
- Positional Encoding
- Masking

Remove RNN and Parallel Processing Techniques

RNN is week…

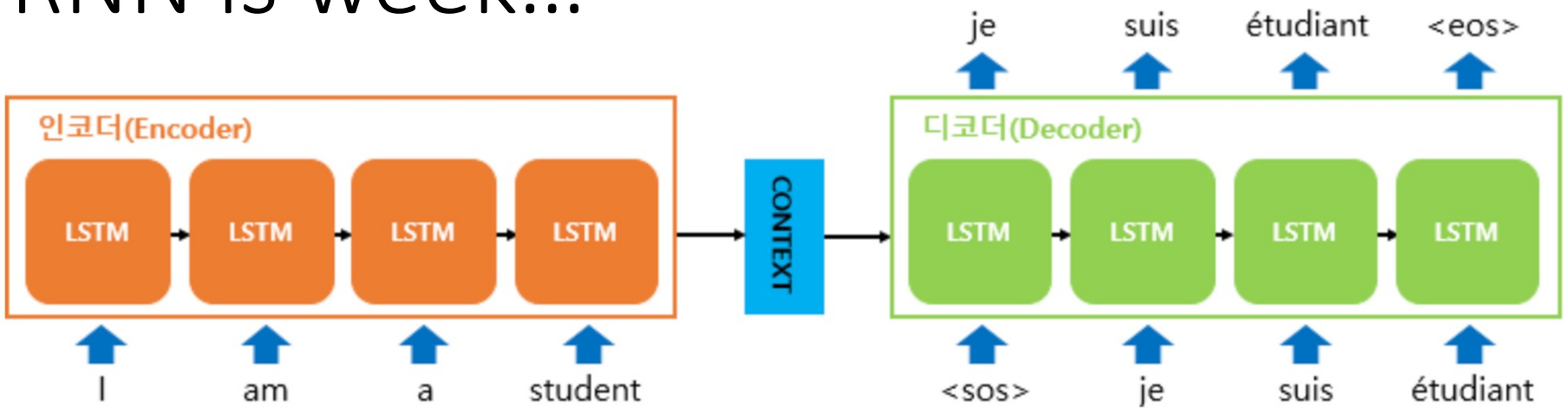


The animal *didn't cross the street, because* ***it*** *was too tired.*

"RNN은 이전 시점에 나온 토큰의 정보를 **hidden state**에 저장한다. 그래서 RNN의 경우 hidden state를 활용해 이번에 등장한 'it'이 이전의 'The Animal'을 가리킨다는 것을 알아낼 것이다."

그러나…

RNN is week…



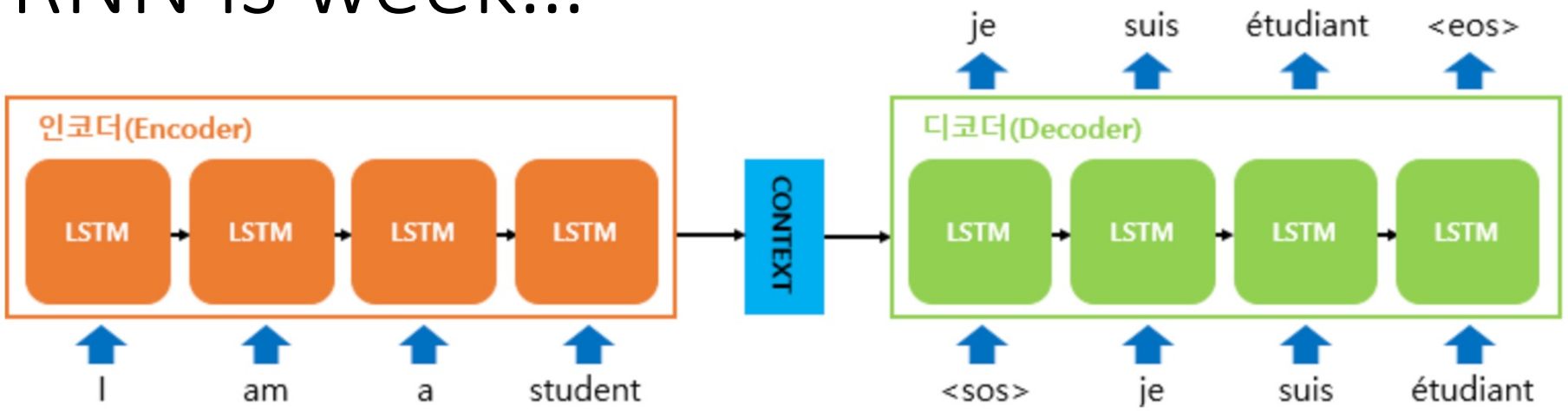
The animal *didn't cross the street, because* ***it*** *was too tired.*

"RNN은 이전 시점에 나온 토큰의 정보를 **hidden state**에 저장한다. 그래서 RNN의 경우 hidden state를 활용해 이번에 등장한 'it'이 이전의 'The Animal'을 가리킨다는 것을 알아낼 것이다."

그러나…

Recurrent Network는 i시점의 hidden state h_i를 구하기 위해서는 h_{i-1}가 필요했다. 결국, RNN은 앞에서부터 순차 계산을 해나가 $h_0, h_1, \ldots, h_n$을 구하는 방법밖에 없었기에 **병렬 처리가 불가능**했다.

RNN is week…



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

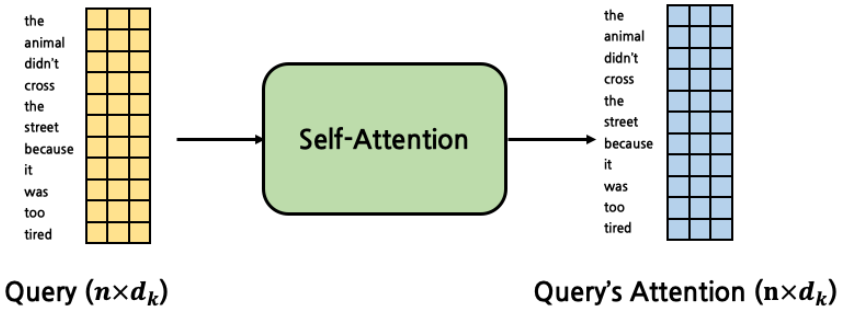
"RNN은 이전 시점에 나온 토큰의 정보를 **hidden state**에 저장한다. 그래서 RNN의 경우 hidden state를 활용해 이번에 등장한 'it'이 이전의 'The Animal'을 가리킨다는 것을 알아낼 것이다."

그러나…

RNN은 시간이 진행될수록 **오래된 시점의 token에 대한 정보가 점차 희미해져간다.** 위 문장의 예시에서 현재 'didn't'의 시점에서 hidden state를 구한다고 했을 때, 바로 직전의 token인 'animal'에 대한 정보는 뚜렷하게 남아있다. 하지만 점차 앞으로 나아갈수록, 'because'나 'it'의 시점에서는 'didn't' 시점보다는 'animal'에 대한 정보가 희미하게 남게 된다. 결국, **서로 거리가 먼 token 사이의 관계에 대한 정보가 제대로 반영되지 못하는 것이다.**

but Attention is strong!

Recurrent Network는 i시점의 hidden state h_i를 구하기 위해서는 h_{i-1}가 필요했다. 결국, RNN은 앞에서부터 순차 계산을 해나가 $h_0, h_1, \dots, h_n$을 구하는 방법밖에 없었기에 **병렬 처리가 불가능**했다.



Attention은 **행렬곱**으로

모든 토큰 쌍 사이의 attention value를 **한번에 구할 수 있다!**

RNN은 시간이 진행될수록 **오래된 시점의 token에 대한 정보가 점차 희미해져간다.** 위 문장의 예시에서 현재 'didn't'의 시점에서 hidden state를 구한다고 했을 때, 바로 직전의 token인 'animal'에 대한 정보는 뚜렷하게 남아있다. 하지만 점차 앞으로 나아갈수록, 'because'나 'it'의 시점에서는 'didn't' 시점보다는 'animal'에 대한 정보가 희미하게 남게 된다. 결국, **서로 거리가 먼 token 사이의 관계에 대한 정보가 제대로 반영되지 못하는 것이다.**

n개 토큰에 대해 $n \times n$ 연산으로 모든 토큰 쌍의 관계를 직접 구함. 중간의 다른 token들을 거치지 않고 바로 direct한 관계를 구하는 것이기 때문에 **RNN에 비해 더 명확하게 관계를 잡아낼 수 있다.**

그러나...

Positional Encoding

RNN이 자연어 처리에서 유용했던 이유는
단어의 위치에 따라 **단어를 순차적으로 입력받아서 처리**하는 RNN의 특성으로
인해 **각 단어의 위치 정보(position information)를 가질 수 있다**는 점에 있었습니다.

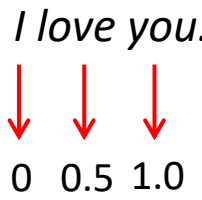
하지만 **Attention**은 단어 입력을 순차적으로 받는 방식이 아니므로
단어의 위치 정보를 다른 방식으로 알려줄 필요가 있습니다.

단어의 위치 정보를 수치화하자! -> Positional Encoding의 도입!

어떻게...?

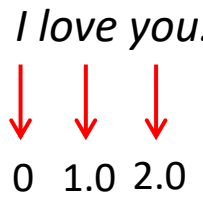
Positional Encoding: 어떻게…?

1. 데이터에 0 ~ 1 사이의 값을 붙인다



but… 토큰과 토큰 사이의 encoding 차이 값인 delta가 input sentence 크기에 따라 가변적

2. 데이터에 선형적으로 숫자를 할당



but… input sentence 길이가 길어지면 숫자가 매우매우 커질 수 있음. -> 모델 일반화가 어려워짐.

Positional Encoding: 어떻게...?

1. 데이터에 0 ~ 1 사이의 값을 붙인다

I love you.

0 0.5 1.0

but... 토큰과 토큰 사이의 encoding 차이 값인 delta가 input sentence 크기에 따라 가변적

2. 데이터에 선형적으로 숫자를 할당

I love you.

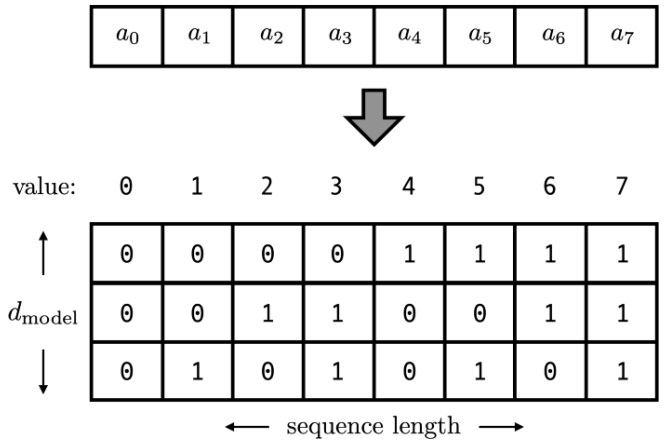
0 1.0 2.0

but... input sentence 길이가 길어지면 숫자가 매우매우 커질 수 있음. -> 모델 일반화가 어려워짐.

3. binary encoding으로 표현해보자

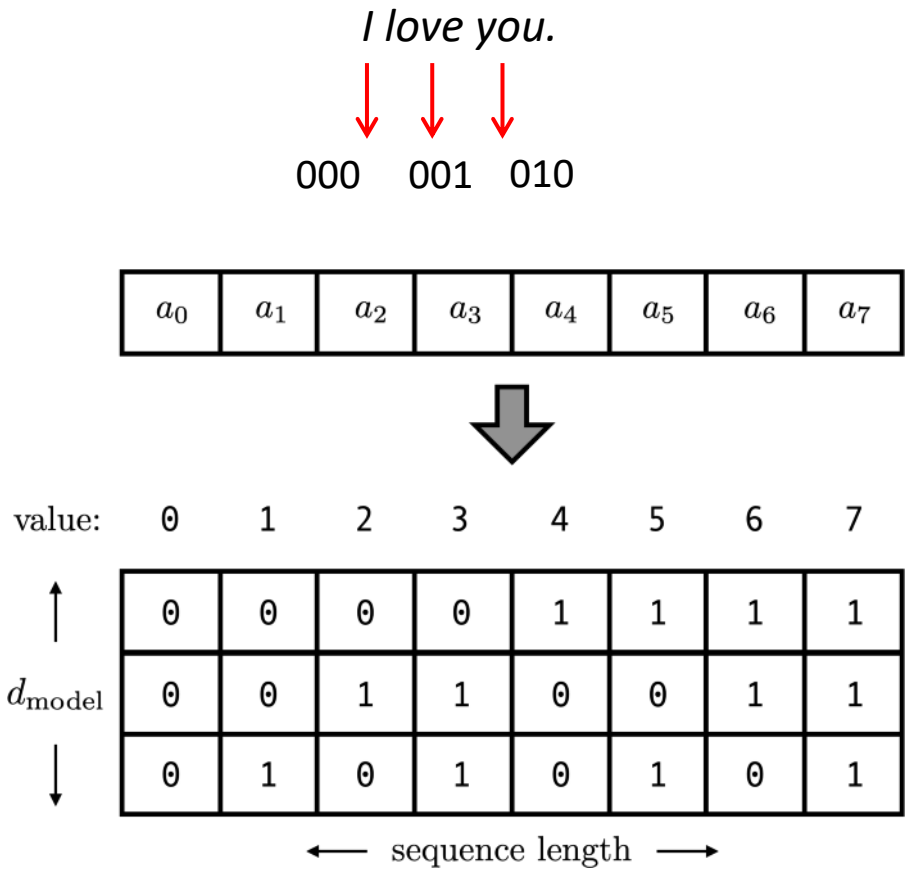
I love you.

000 001 010



Positional Encoding: 어떻게…?

3. binary encoding으로 표현해보자

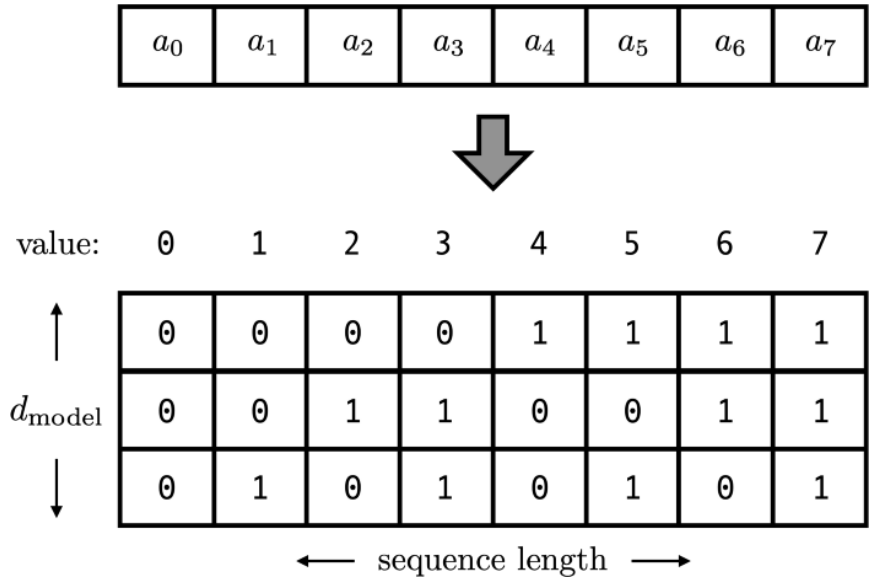


- 앞의 두 방식은 스칼라 인코딩 이었지만, binary encoding은 **벡터 인코딩**이다.
- 입력 문장의 길이가 바뀌어도 동일한 위치에서는 동일하게 인코딩!
  - 1의 단점을 커버
- 입력 문장의 길이가 2^d 만 넘지 않는다면 표현 가능!
  - (논문에선 $d = 256$, 웬만하면 안 넘는다!)
  - 2의 단점을 커버
- 값이 0 또는 1이기 때문에 exploding 걱정 X
  - normalize 되어 있다고도 함!
  - 2의 단점을 커버

그러나…

Positional Encoding: 어떻게…?

3. binary encoding으로 표현해보자



Our binary vectors come from a discrete function, and not a discretization of a continuous function.

대충 binary vector로는 continuous function인

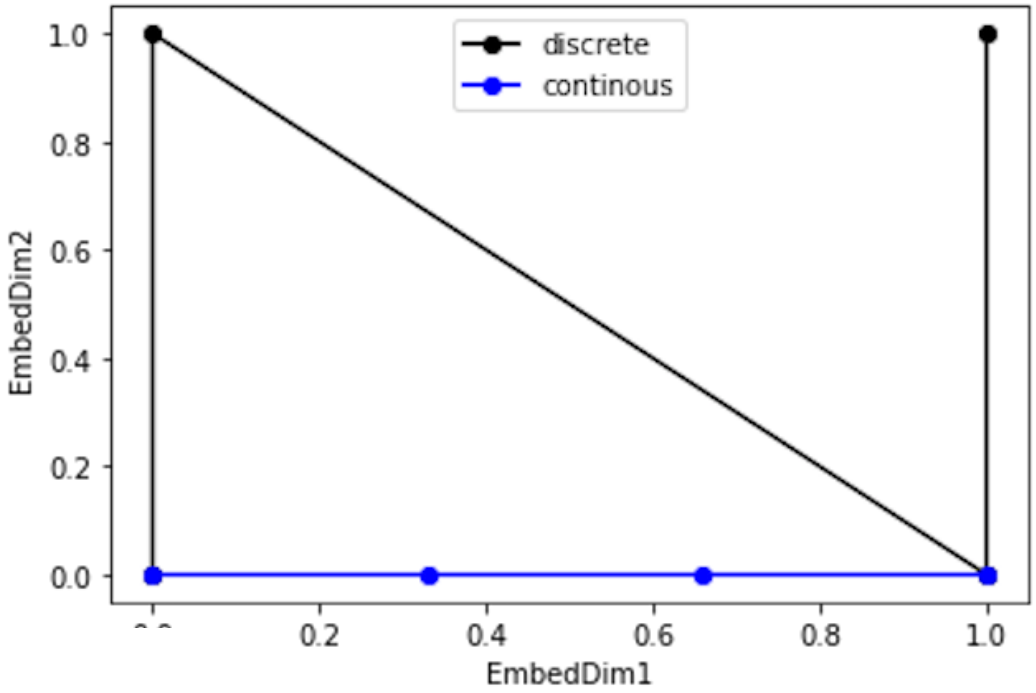
거리(distance)가 제대로 측정되지 않는다는 말이다!

Positional Encoding: 어떻게...?

Our binary vectors come from a discrete function, and not a discretization of a continuous function.

대충 binary vector로는 continuous function인

거리(distance)가 제대로 측정되지 않는다는 말이다!



연속함수인 y = x의 출력 $(0, 0.33, 0.66, 1.00)$ (방법2)-파랑

이산함수의 이진수 출력 $((0,0), (0,1), (1,0), (1,1))$ (방법3)-검정

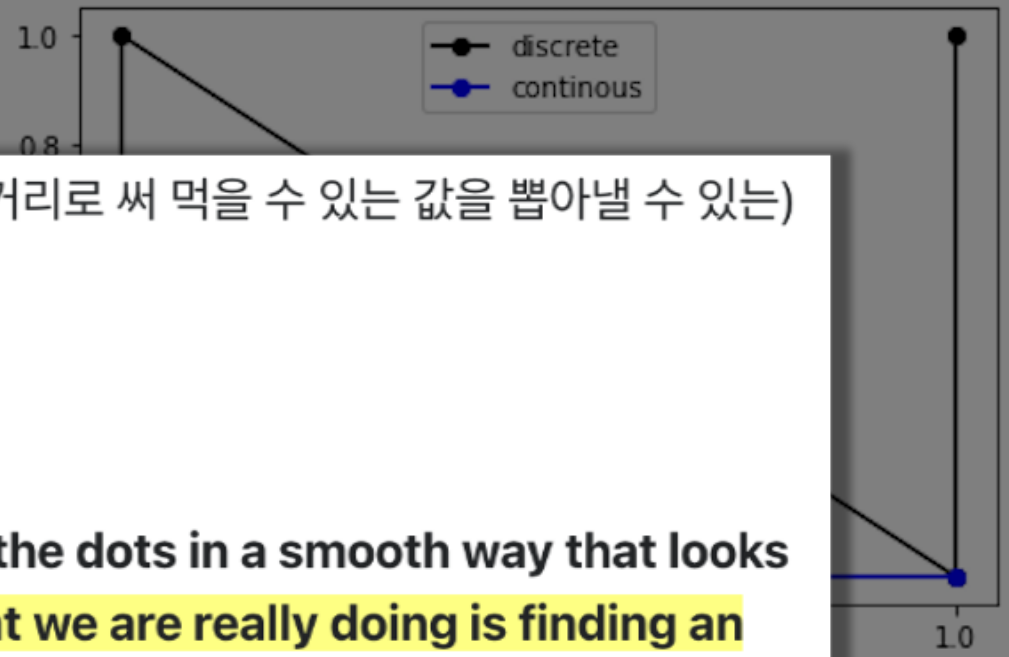
파랑 선은 점 사이 간격이 일정한 것을 볼 수 있지만, 검은 점들 간 거리는

$$(0,0) \rightarrow (0,1) : 1$$
$$(0,1) \rightarrow (1,0) : \sqrt{2}$$
$$(1,0) \rightarrow (1,1) : 1$$

가 됩니다. 같은 거리를 상정 $(D(0 \rightarrow 1) = D(1 \rightarrow 2)$ 이 되기를 원함) 했음에도 모델은 다른 거리로 판단하게 되는 것$(D(0 \rightarrow 1) \neq D(1 \rightarrow 2))$입니다. 이는 고차원으로 갈수록 더욱 문제가 되어 binary count를 사용하는 것은 바람직하지 않습니다.

Positional Encoding: 어떻게...?

Our binary vectors come from a discrete function, and not a discretization of a continuous function.



대충 따라서, 다음으로 해결해야 할 과제는 점들이 등간격으로(혹은 거리로 써 먹을 수 있는 값을 뽑아낼 수 있는)

거리 배열된 부드러운 곡선을 찾는것, 다시 말하자면

embedding manifold를 찾는 것입니다.

연속

이산

So with pictures, we want a function that connects the dots in a smooth way that looks natural. For anyone who has studied geometry, what we are really doing is finding an embedding manifold.

파랑 선

$$(0,0) \to (0,1) : 1$$
$$(0,1) \to (1,0) : \sqrt{2}$$
$$(1,0) \to (1,1) : 1$$

가 됩니다. 같은 거리를 상정 $(D(0 \to 1) = D(1 \to 2)$ 이 되기를 원함$)$ 했음에도 모델은 다른 거리로 판단하게 되는 것$(D(0 \to 1) \neq D(1 \to 2))$입니다. 이는 고차원으로 갈수록 더욱 문제가 되어 binary count를 사용하는 것은 바람직하지 않습니다.

Positional Encoding: embedding manifold?

what we are really doing is finding an embedding manifold.

discrete binary vector에서 영감을 얻어보자!

0: 0 0 0 0
1: 0 0 0 1
2: 0 0 1 0
3: 0 0 1 1
4: 0 1 0 0
5: 0 1 0 1
6: 0 1 1 0
7: 0 1 1 1
8: 1 0 0 0

[생성 규칙]

최하위 비트(LSB)는 1의 주기로 토글한다.

second LSB는 2의 주기로 토글한다.

third LSB는 4의 주기로 토글한다.

….

nth LSB는 2^{n-1}의 토글!

0: 0 0 0 0
1: 0 0 0 1
2: 0 0 1 0
3: 0 0 1 1
4: 0 1 0 0
5: 0 1 0 1
6: 0 1 1 0
7: 0 1 1 1
8: 1 0 0 0

[생성 규칙]

최하위 비트(LSB)는 1의 주기로 토글한다. → $\text{sign}(\sin(\pi \cdot pos))$

second LSB는 2의 주기로 토글한다. → $\text{sign}(\sin(\pi/2 \cdot pos))$

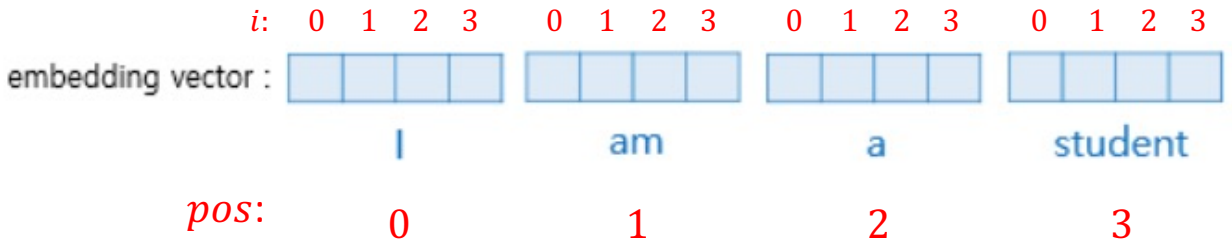
third LSB는 4의 주기로 토글한다. → $\text{sign}(\sin(\pi/2^2 \cdot pos))$

....

nth LSB는 2^{n-1}의 토글! → $\text{sign}(\sin(\pi/2^i \cdot pos))$

pos: 입력 문장에서 해당 토큰의 위치

i: 임베딩 벡터 내의 index



0: 0 0 0 0
1: 0 0 0 1
2: 0 0 1 0
3: 0 0 1 1
4: 0 1 0 0
5: 0 1 0 1
6: 0 1 1 0
7: 0 1 1 1
8: 1 0 0 0

[생성 규칙]

최하위 비트(LSB)는 1의 주기로 토글한다. → $\text{sign}(\sin(\pi \cdot pos))$

second LSB는 2의 주기로 토글한다. → $\text{sign}(\sin(\pi/2 \cdot pos))$

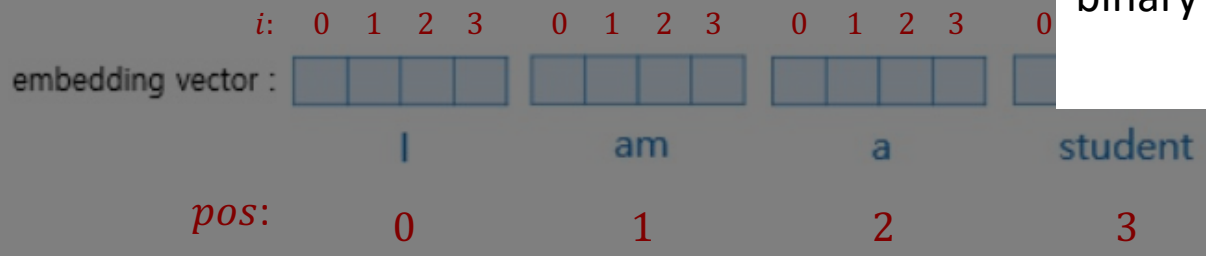
third LSB는 4의 주기로 토글한다. → $\text{sign}(\sin(\pi/2^2 \cdot pos))$

....

nth LSB는 2^{n-1}의 토글! → $\text{sign}(\sin(\pi/2^i \cdot pos))$

pos: 입력 문장에서 해당 토큰의 위치

i: 임베딩 벡터 내의 index

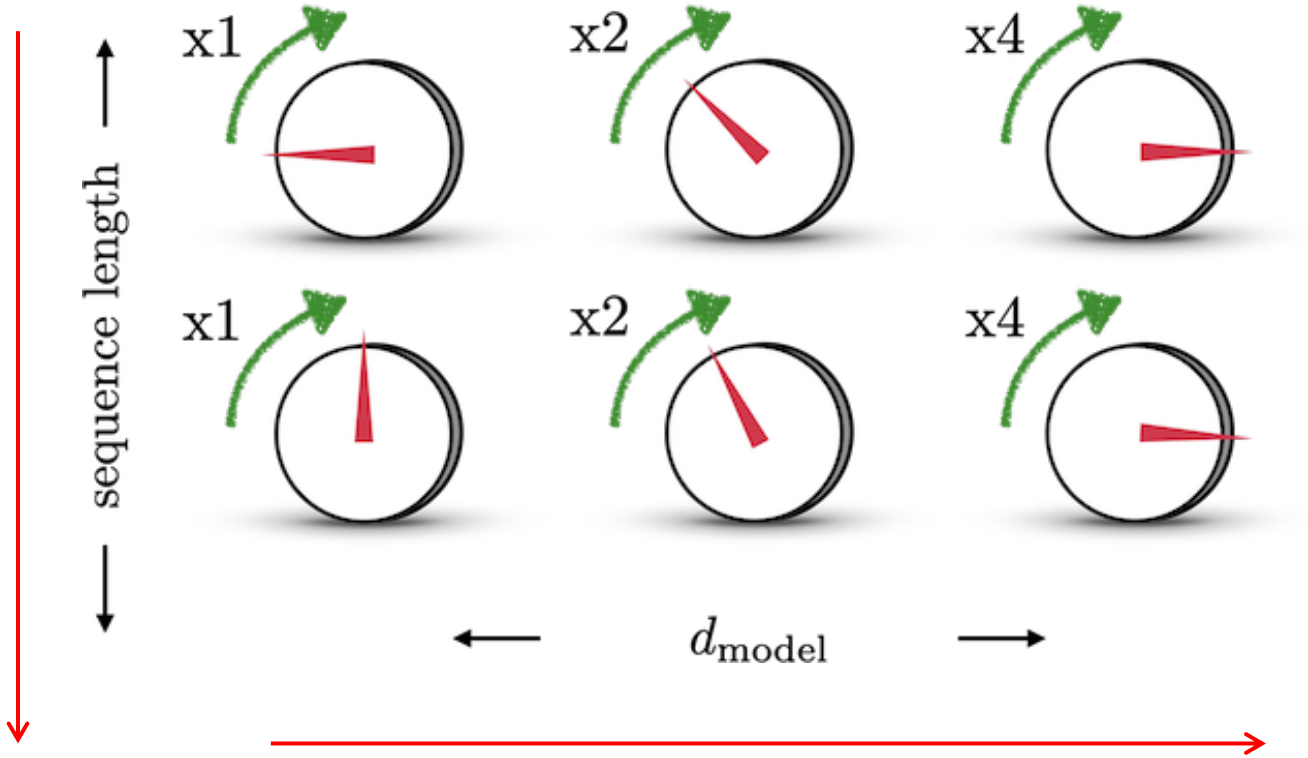


sign 때문에 discrete binary encoding이 되는 거라면, sign을 빼면 continuous binary encoding이 될까?

Positional Encoding: embedding manifold?

what we are really doing is finding an embedding manifold.

문장에서의 position에 따라 시계가 시작하는 값이 다름



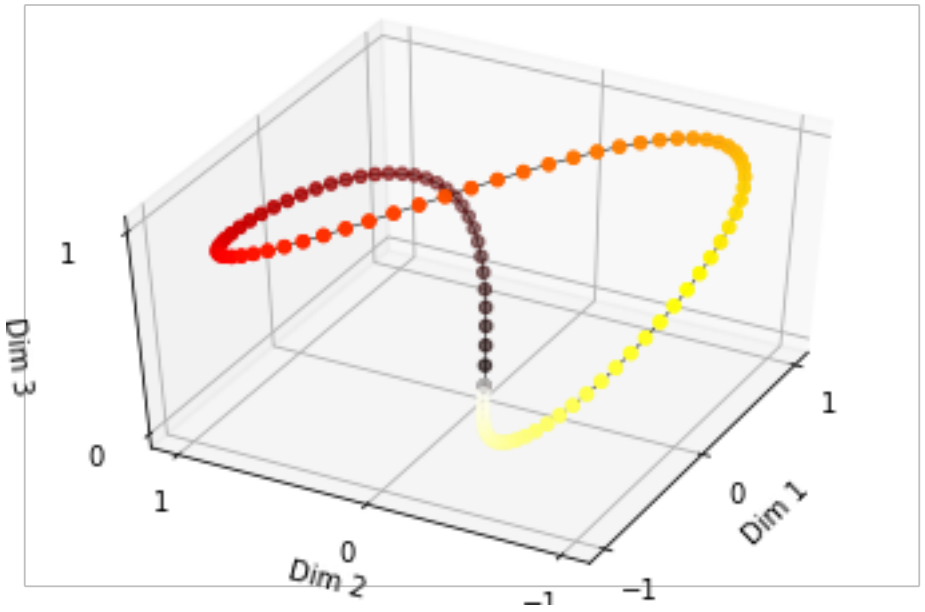
$$y_1 = sin(\frac{\pi}{2}x_1)$$

$$y_2 = sin(\frac{\pi}{4}x_2)$$

$$y_3 = sin(\frac{\pi}{8}x_3)$$

Positional Encoding: embedding manifold!

what we are really doing is finding an embedding manifold.



$$PE(pos, i) = \sin\left(\frac{\pi}{2^i} \cdot pos\right)$$

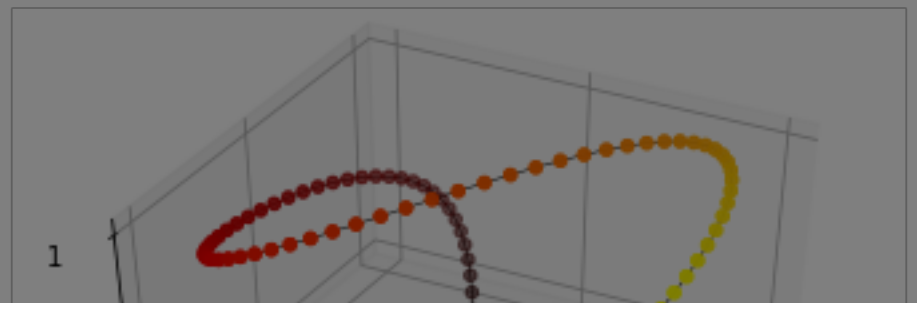
The colors indicate the absolute position that that coordinate represents. Frequencies are omega=pi/2, pi/4, pi/8 for dimensions 1, 2 and 3 respectively.

부드러운 함수는 찾았다!

그러나...

Positional Encoding: embedding manifold!

what we are really doing is finding an embedding manifold.



$$PE(pos, i) = \sin\left(\frac{\pi}{2^i} \cdot pos\right)$$

$PE(pos = 0)$과 $PE(pos = 9)$의 값이 동일하기 때문에 (둘다 $(0, 0, 0)$이다.)

$$|PE(pos = 0) - PE(pos = 1)| = |PE(pos = 8) - PE(pos = 1)|$$

를 만족하게 되고, 이에 따라 8번째 토큰이 1번째 토큰과 매우 가깝다는 positional encoding을 하게 된다.

sinusoidal positional encoding의 주기성 때문에 어쩔 수 없는 부분!

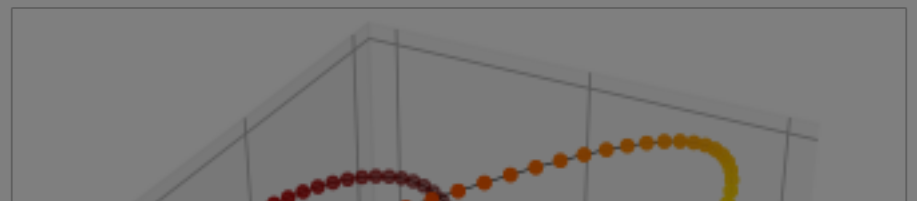
represents. Frequencies are omega=pi/2, pi/4, pi/8 for dimensions 1, 2 and 3 respectively.

부드러운 함수는 찾았다!

그러나...

Positional Encoding: embedding manifold!

what we are really doing is finding an embedding manifold.



$$PE(pos, i) = \sin\left(\frac{\pi}{2i} \cdot pos\right)$$

등간격을 인코딩인가? 즉, $D(0 \rightarrow 1) = D(1 \rightarrow 2)$인가?

위의 sinusoidal encoding은 **그렇지 않다**!

ex) $\left|(0,0) - \left(1, 1/\sqrt{2}\right)\right| \neq \left|\left(1, 1/\sqrt{2}\right) - (0,1)\right|$ 😭

좀더 어렵게 표현하면...

"$PE(pos + k) = T_k\left(PE(pos)\right)$를 만족하는 선형 변환 T_k가 존재하지 않는다!"

찾았다!

그러나...

Positional Encoding: embedding manifold!

등간격 인코딩을 찾자!

$$PE(pos) = \left[\sin\left(\frac{\pi}{2}\cdot pos\right), \sin\left(\frac{\pi}{2^2}\cdot pos\right), \dots, \sin\left(\frac{\pi}{2^2}\cdot pos\right)\right]$$

$$PE'(pos) = \left[\cos\left(\frac{\pi}{2}\cdot pos\right), \cos\left(\frac{\pi}{2^2}\cdot pos\right), \dots, \cos\left(\frac{\pi}{2^n}\cdot pos\right)\right]$$

똑같이 cos으로도 생성

$$PE(pos) = \left[\sin\left(\frac{\pi}{2}\cdot pos\right), \cos\left(\frac{\pi}{2}\cdot pos\right), \dots, \sin\left(\frac{\pi}{2^2}\cdot pos\right), \cos\left(\frac{\pi}{2^n}\cdot pos\right)\right]$$

sin, cos을 번갈아가며 사용!

Positional Encoding: embedding manifold!

등간격 인코딩을 찾자!

$$PE(pos) = \begin{cases} \sin\left(\frac{\pi}{2^k} \cdot pos\right), & if\ i = 2k \\ \cos\left(\frac{\pi}{2^k} \cdot pos\right), & if\ i = 2k+1 \end{cases}$$

좋은 인코딩인가?
등간격 인코딩인가?

Positional Encoding: embedding manifold!

등간격 인코딩을 찾자!

$$PE(pos) = \begin{cases} \sin\left(\frac{\pi}{2^k} \cdot pos\right), & if\ i = 2k \\ \cos\left(\frac{\pi}{2^k} \cdot pos\right), & if\ i = 2k+1 \end{cases}$$

좋은 인코딩인가?
등간격 인코딩인가?

$$\begin{pmatrix} \cos(\theta+\phi) \\ \sin(\theta+\phi) \end{pmatrix} = \begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix} \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix}$$

rotational transformation under 2-dim

선형 변환이다!
(matrix로 표현되는 모든 변환은 선형 변환임)

Positional Encoding: embedding manifold!

$$\mathbf{T}(\Delta x) = \begin{pmatrix} \begin{bmatrix} \cos(\omega_0 \Delta x) & -\sin(\omega_0 \Delta x) \\ \sin(\omega_0 \Delta x) & \cos(\omega_0 \Delta x) \end{bmatrix} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \begin{bmatrix} \cos(\omega_{n-1} \Delta x) & -\sin(\omega_{n-1} \Delta x) \\ \sin(\omega_{n-1} \Delta x) & \cos(\omega_{n-1} \Delta x) \end{bmatrix} \end{pmatrix}$$

sin/cos 교차하는 현재의 인코딩 방식으로 확장하면 요런 형태의
선형변환이 된다!

Rotational transformation under 2-dim

선형 변환이다!

(matrix로 표현되는 모든 변환은 선형 변환임)

Positional Encoding: embedding manifold!

등간격 인코딩을 찾자!

$$PE(pos) = \begin{cases} \sin\left(\frac{\pi}{2^k} \cdot pos\right), & if\ i = 2k \\ \cos\left(\frac{\pi}{2^k} \cdot pos\right), & if\ i = 2k+1 \end{cases}$$

좋은 인코딩인가?
등간격 인코딩인가?

그렇다!!

사실 아직 주기성 문제를
해결하지 않았다!

Positional Encoding: embedding manifold!

사실 아직 주기성 문제를
해결하지 않았다!

$$PE(pos) = \begin{cases} \sin(\omega_k \cdot pos), & if\ i = 2k \\ \cos(\omega_k \cdot pos), & if\ i = 2k+1 \end{cases}$$

$$\omega_k = \pi/2^k$$

↓

$$\omega_k = 1/1000^k$$

사실 1000이라는 숫자는 hyper-parameter다

주기를 엄청나게 길게 만들어버린다!
(인간적으로 너무 길다!)

↓

$$\omega_k = 1/1000^{k/d}$$

마지막 index $i = d = 256$쯤 되면 주기가 엄청 길어지게 만들자~

Positional Encoding (final)

$$PE(pos) = \begin{cases} \sin(\omega_k \cdot pos), & if\ i = 2k \\ \cos(\omega_k \cdot pos), & if\ i = 2k+1 \end{cases}$$

$$\omega_k = 1/1000^{2k/d}$$

저자 왈: Positional Encoding은 아래의 조건을 만족해야 하고, 우리 껀 만족함 ㅎㅎ

1. 문장의 각 time step 마다 하나의 유일한 encoding 값을 출력 해야 한다.
2. 입력되는 문장의 길이에 상관없이 두 time-step 간의 거리는 동일해야 한다.
3. 모델에 대한 일반화가 가능해야 한다. 모델 학습 때보다 더 긴 길이의 문장이 들어와도 모델이 처리할 수 있어야 한다. (인코딩 값이 특정 범위 내에 있어 exploding 하지 않아야 한다)
4. 하나의 key 값처럼 결정되어야 한다. 같은 문장을 넣었을 때 매번 다른 값이 나오면 안 된다.

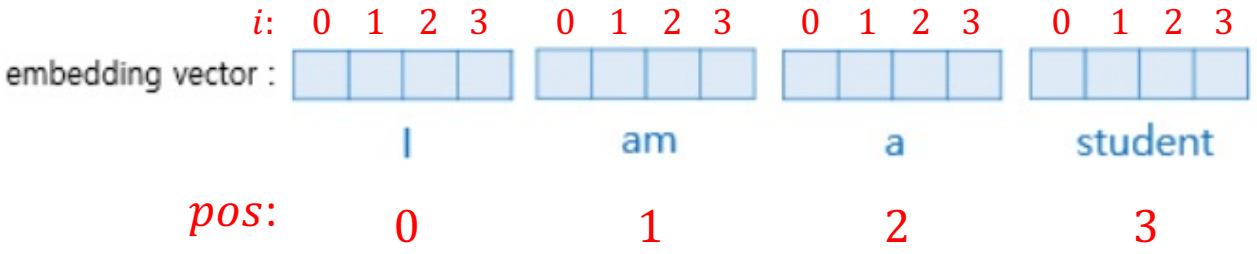
Positional Encoding

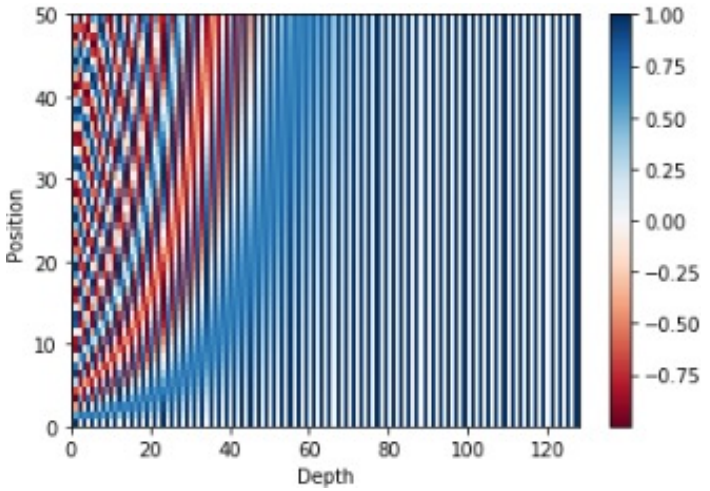
하지만 **Attention**은 단어 입력을 순차적으로 받는 방식이 아니므로

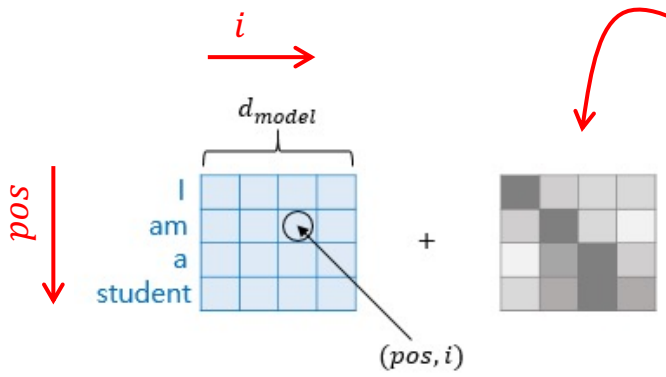
단어의 위치 정보를 다른 방식으로 알려줄 필요가 있습니다.

pos: 입력 문장에서 해당 토큰의 위치

i: 임베딩 벡터 내의 index







positional encoding

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

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

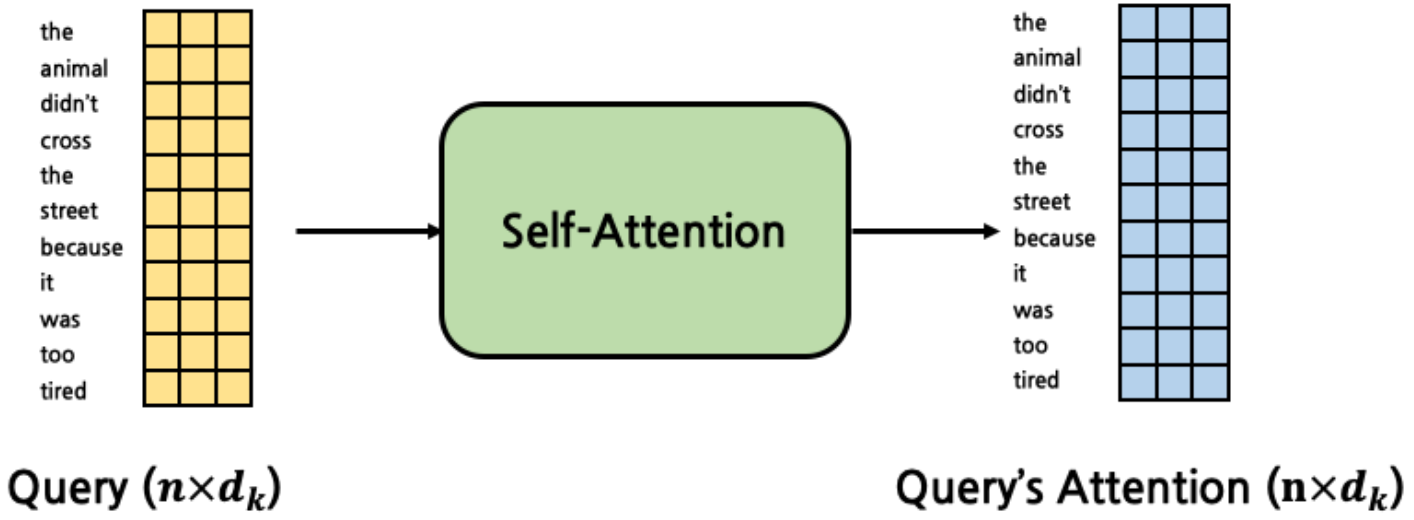
$(pos, 2i)$, 짝수 인덱스에서는 sin 함수

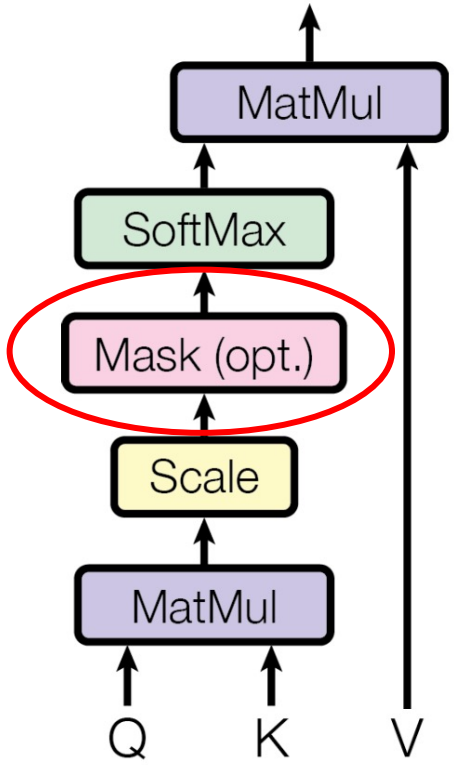
$(pos, 2i+1)$, 홀수 인덱스에서는 cos 함수

- Why remove RNN?
- Positional Encoding
- Masking

Remove RNN and Parallel Processing Techniques

self-attention: pad mask



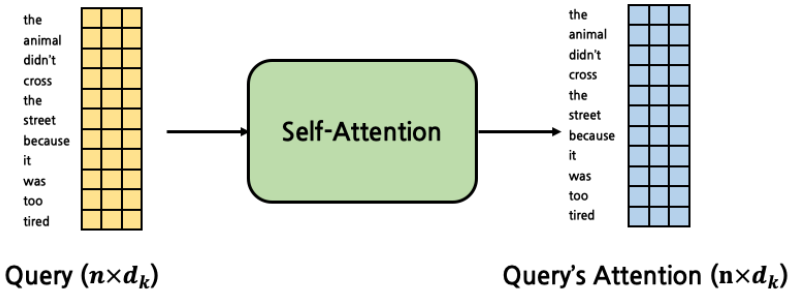


[1] The animal didn't cross the street, because it was too tired. 11개 토큰

[2] I am a student. 4개 토큰

여러 문장을 하나의 Mini-batch로 묶어서 동시에 학습시키려면 어떻게 해야할까?
-> pad mask!

self-attention: pad mask



[1] The animal didn't cross the street, because it was too tired. 11개 토큰

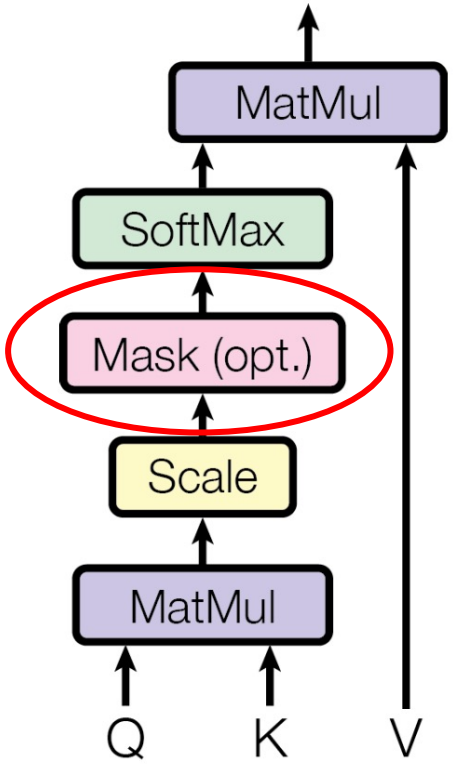
[2] I am a student. 4개 토큰

여러 문장을 하나의 Mini-batch로 묶어서 동시에 학습시키려면 어떻게 해야할까?

"I am a student." 문장의 말단에 <pad> 토큰을 붙여 mini-batch 내의 문장 길이를 동일하게 만들어준다!
보통은 `seq_len`라는 이름의 상수를 정해서 입력 문장에 padding을 한다. (`seq_len = 20`)

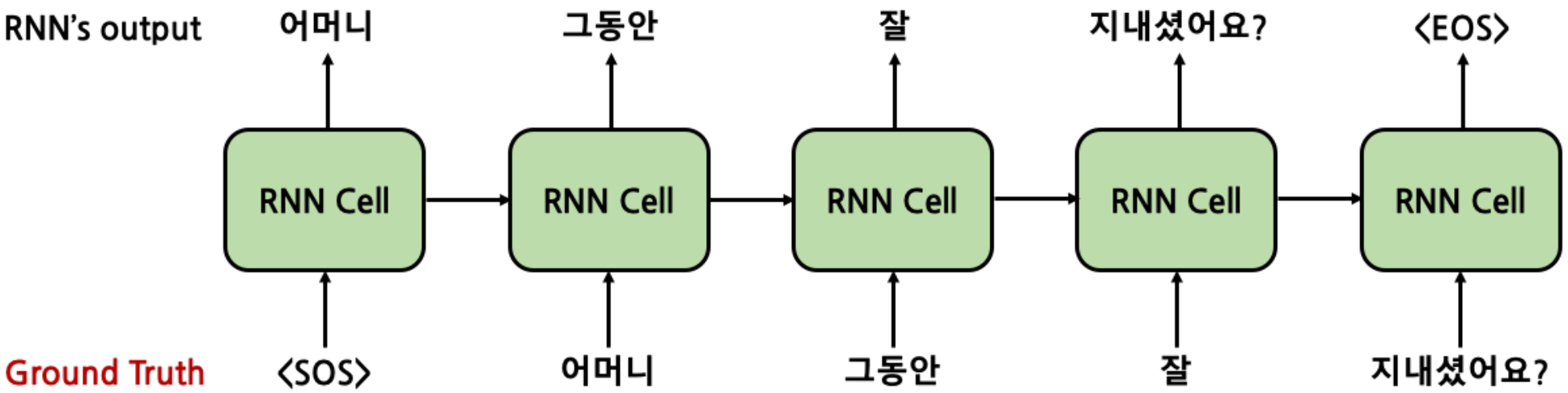
그러나 이 <pad> 토큰들에 대해선 다른 토큰들에 대한 attention이 부여되면 안 되기 때문에
`seq_len x seq_len` 크기의 mask matric를 결과에 곱해준다! 이때, <pad> 토큰에 해당하는 부분은 **– inf의 값을 가지게 한다!**

self-attention: pad mask



<pad> 토큰이 있는 부분의 attention score에 –inf가 곱해지면서 해당 부분은 softmax 이후에 attention probability가 0이 된다. 그래서 이후 attention value를 구할 때 **<pad> 토큰 부분은 전혀 기여하지 못 한다!**

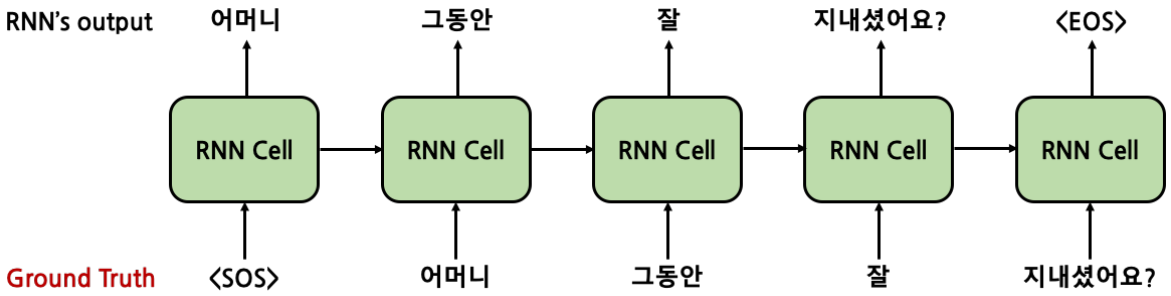
Decoder self-attention과 teacher forcing (교사 강요)

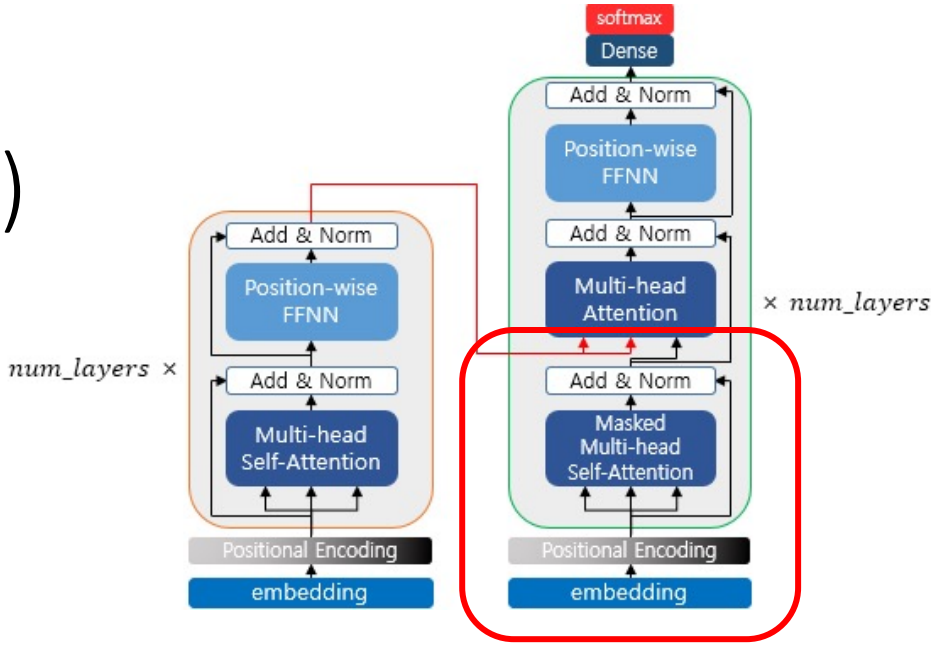


교사강요: 디코더 **학습** 때는 초기 학습 불안정 때문에 gt 값을 입력으로 넣어준다.

-> Transformer의 Self-attention Decoder에서도 동일!

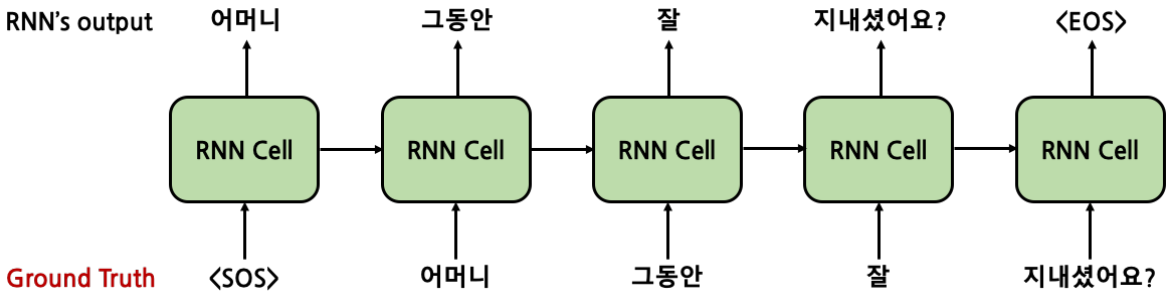
Decoder self-attention (train phase)

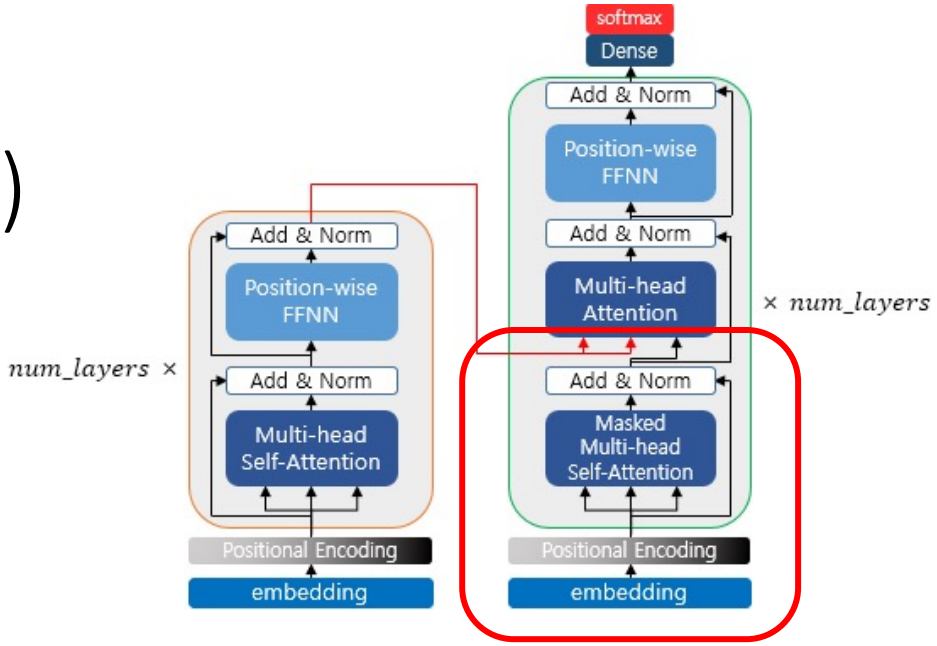




- Encorder의 self-attention
  - i번째 토큰과 $i-N$번째 토큰, $i+N$번째 토큰과의 attention을 모두 계산한다!
- Decorder의 self-attention
  - i번째 토큰은 $\boldsymbol{i-N}$**번째 토큰들에 대해서만 attention을 계산해야 한다!**
  - 그러나 교사 강요를 하게 되면, 전체 문장에 대한 gt를 받기 때문에 $i+N$번째 토큰들을 쓰지 않도록 **마스킹** 해줘야 한다!

Decoder self-attention (train phase)





- Encorder의 self-attention
  - i번째 토큰과 $i-N$번째 토큰, $i+N$번째 토큰과의 attention을 모두 계산한다!
- Decorder의 self-attention
  - i번째 토큰은 **$i-N$번째 토큰들에 대해서만 attention을 계산해야 한다!**
  - 그러나 교사 강요를 하게 되면, 전체 문장에 대한 gt를 받기 때문에 $i+N$번째 토큰들을 쓰지 않도록 **마스킹** 해줘야 한다!

subsequent masking

또는

look-ahead masking

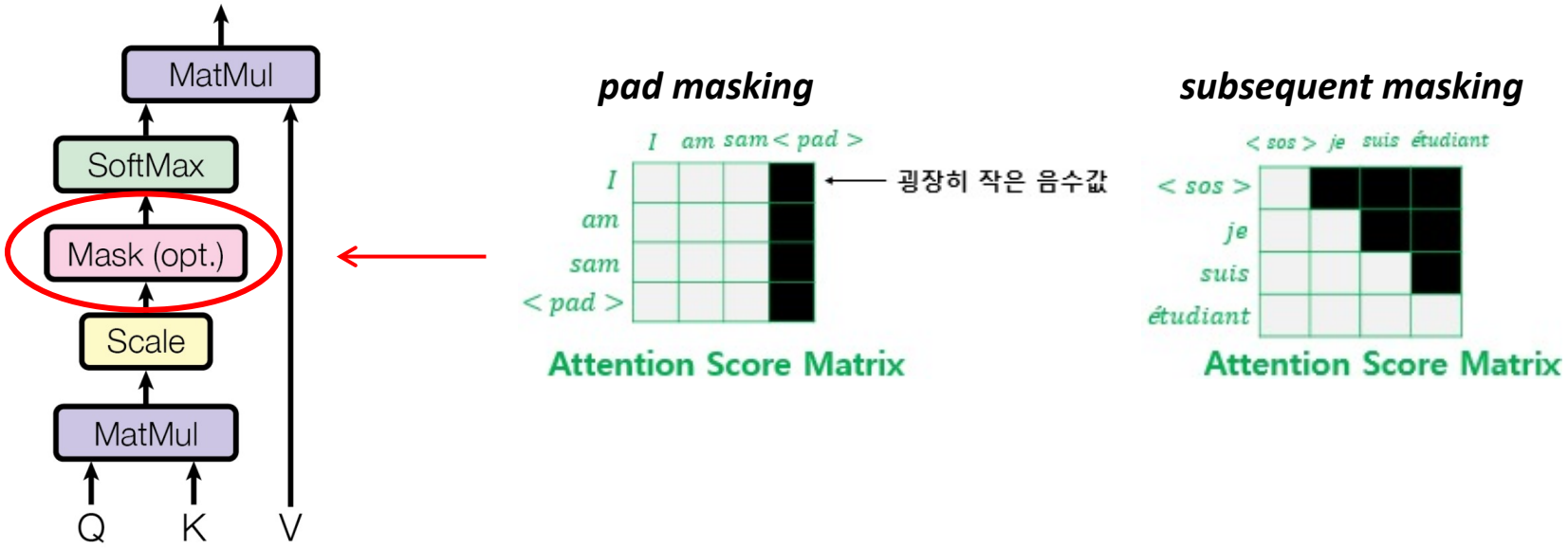
Decoder self-attention (train phase)

"현재 시점의 예측에서 현재 시점보다 미래에 있는 단어들을 참고하지 못하도록"

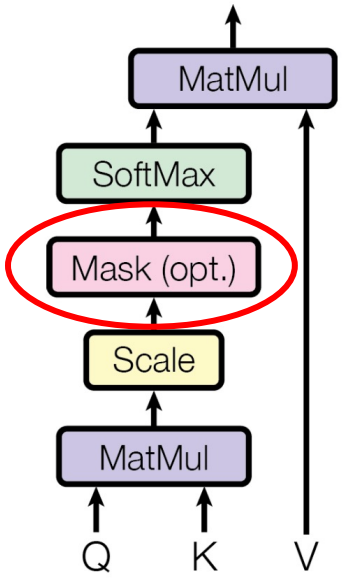
subsequent masking

또는

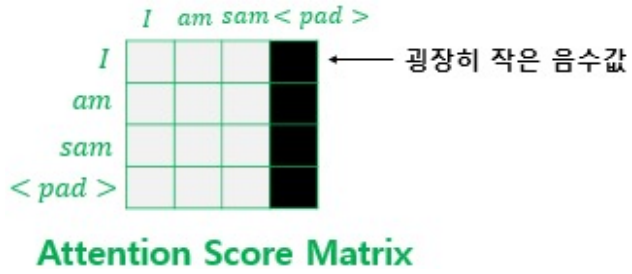
look-ahead masking



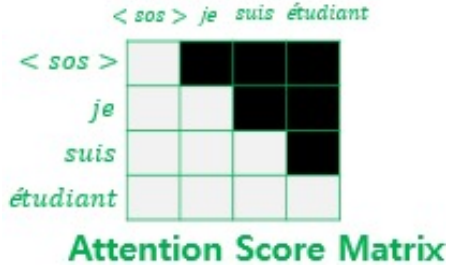
Decoder self-attention (eval phase)



pad masking

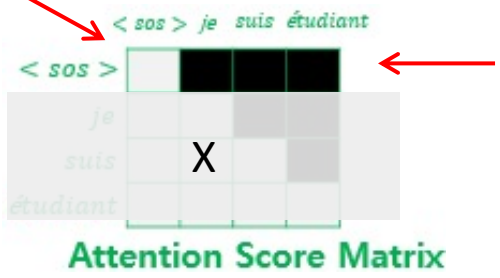


subsequent masking



Train 때는 디코더도 전체 출력 문장을 받는다.
그러나 Eval 때는 <sos> 토큰 하나만 받는다.

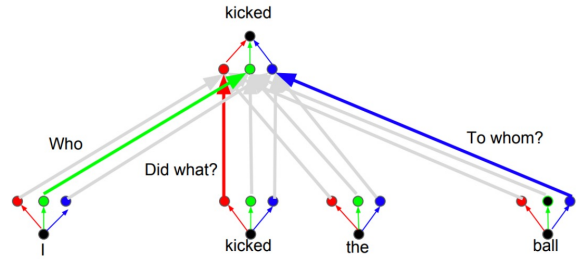
첫 부분 <sos> 부분만 남기는
subsequent masking



seq_len 만큼 ***pad masking***

맺음말

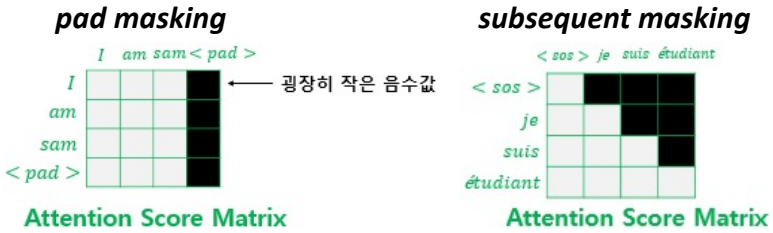
맺음말



- Attention
  - Self-Attention: Q, K, V가 모두 동일 문장에서 유래
  - Multi-head Attention: Attention을 여러 번 해서 feature extraction
- Remove RNN and Parallel Processing Techniques
  - Why remove RNN?: (1) 병렬 처리! (2) Attention이 더 direct relation catch
  - Positional Encoding: RNN을 안 쓰니까 토큰 위치 정보를 직접 인코딩 해야 함. 등간격 어쩌구 선형 변환 어쩌구…
  - Masking
    - pad masking: 병렬 처리 위해 짧은 문장들 padding
    - subsequent masking: 디코더 교사 강요에서 미래의 토큰들 참조하지 못하게

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

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



Thank you!

references

- 딥러닝을 이용한 자연어 처리 입문
- [NLP 논문 구현] pytorch로 구현하는 Transformer (Attention is All You Need)
- [Transformer]-1 Positional Encoding은 왜 그렇게 생겼을까? 이유