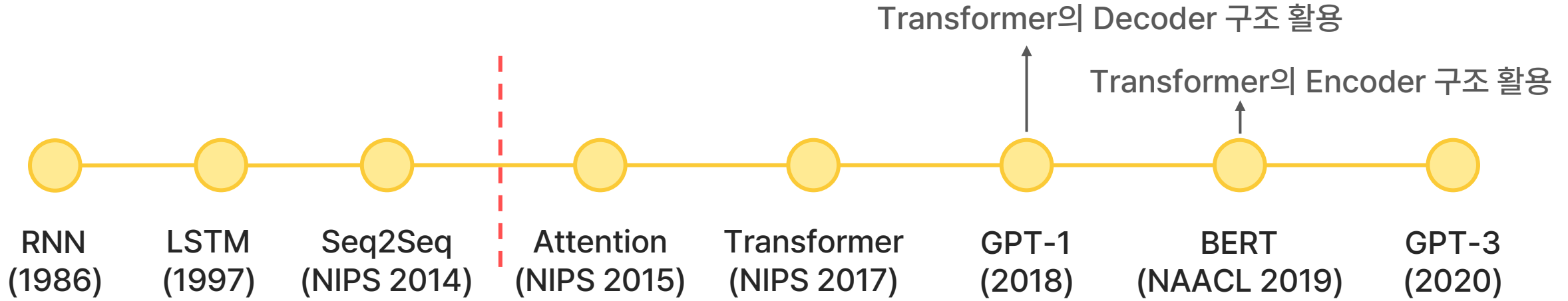


Transformer (Attention Is All You Need)

나동빈 님의 '꼼꼼한 딥러닝 논문 리뷰와 코드 실습' 을 정독 후 정리한 글입니다.

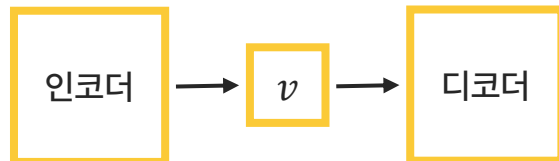
길종현 (github.com/hyeon-n-off)

딥러닝의 기반의 기계 번역 발전 과정



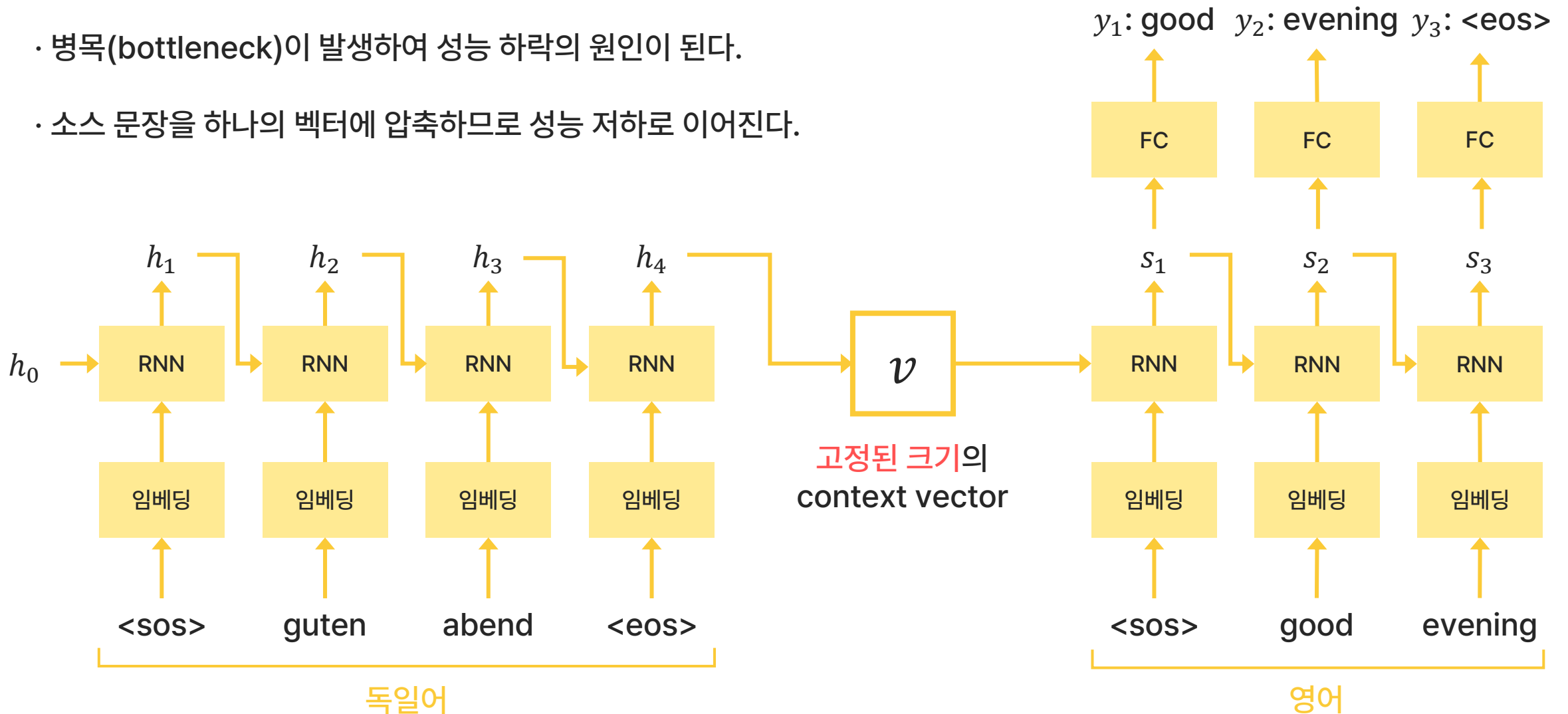
고정된 크기의
context vector 사용

→ 정보를 하나의 벡터로 압축해야하기에 **성능적 한계**가 존재
또한, **병렬계산이 불가능**



기존의 Seq2Seq 모델들의 한계점

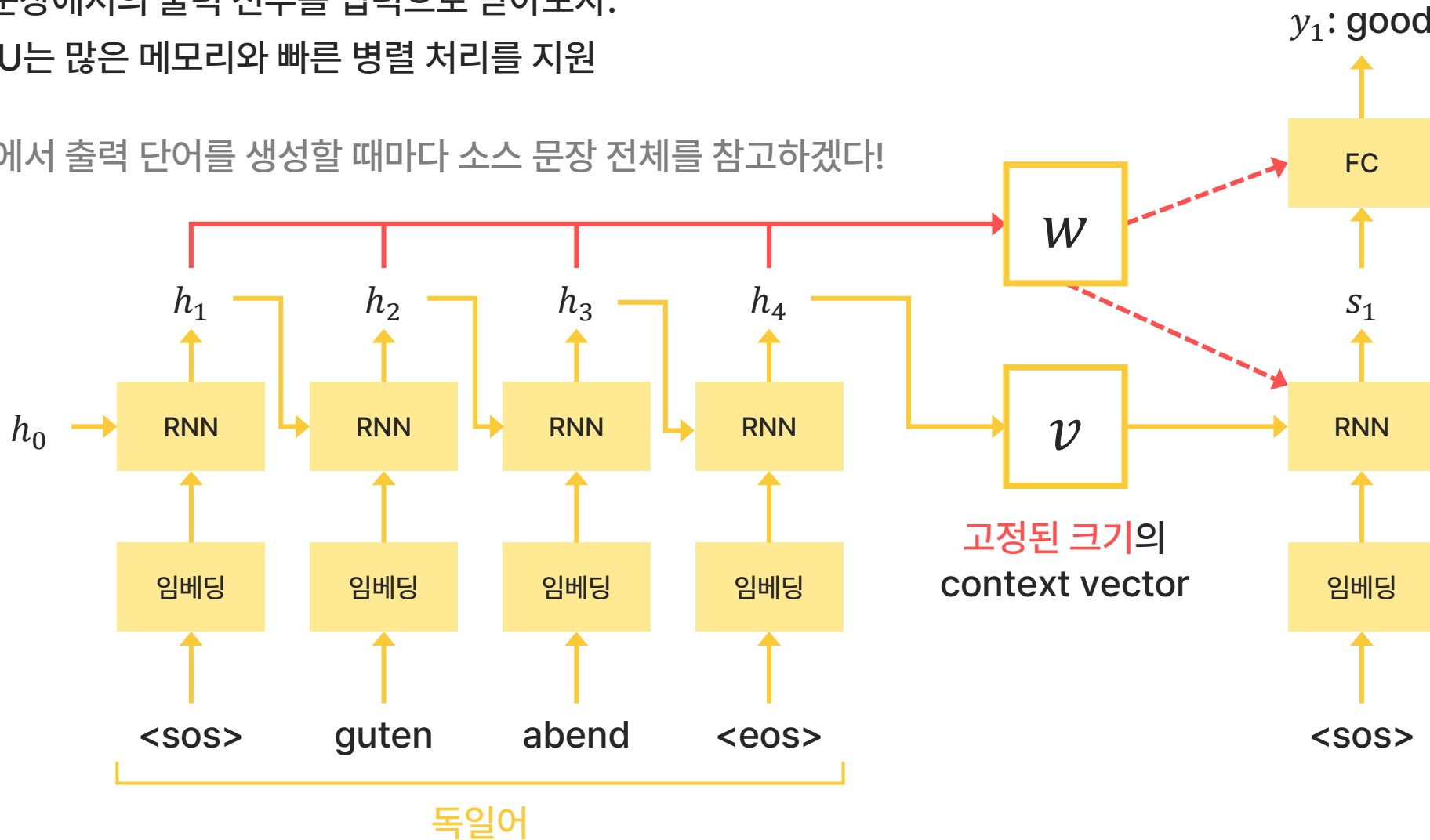
- 병목(bottleneck)이 발생하여 성능 하락의 원인이 된다.
- 소스 문장을 하나의 벡터에 압축하므로 성능 저하로 이어진다.



Seq2Seq with Attention

- 매번 소스 문장에서의 출력 전부를 입력으로 받아보자.
→ 최신 GPU는 많은 메모리와 빠른 병렬 처리를 지원

디코더 부분에서 출력 단어를 생성할 때마다 소스 문장 전체를 참고하겠다!

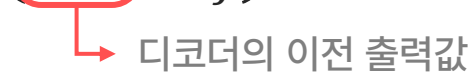


Seq2Seq with Attention : 디코더(Decoder)

- 디코더는 매번 인코더의 모든 출력 중에서 어떤 정보가 중요한지를 계산한다.

- i = 현재의 디코더가 처리 중인 인덱스

- j = 각 인코더의 출력 인덱스

- 에너지(Energy) $e_{ij} = a(s_{i-1}, h_j)$


디코더의 이전 값과 인코더의 각 출력값(소스 문장 전체)을(를) 비교한 연관성을 수치화 하겠다!

- 가중치(Weight) $\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})} \longrightarrow \text{소프트맥스 (상대적인 확률값으로 변환)}$

Seq2Seq with Attention : 디코더(Decoder)

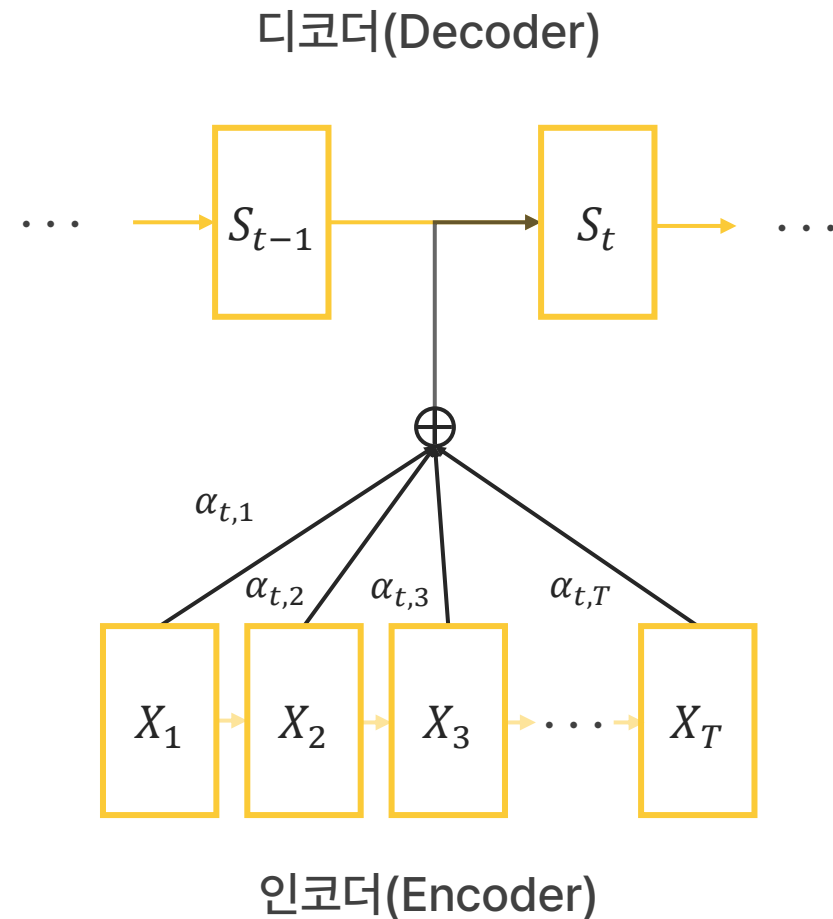
· 에너지(Energy) $e_{ij} = a(s_{i-1}, h_j)$

· 가중치(Weight) $\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$

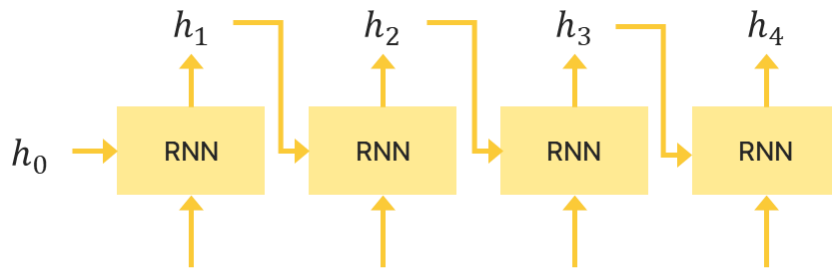
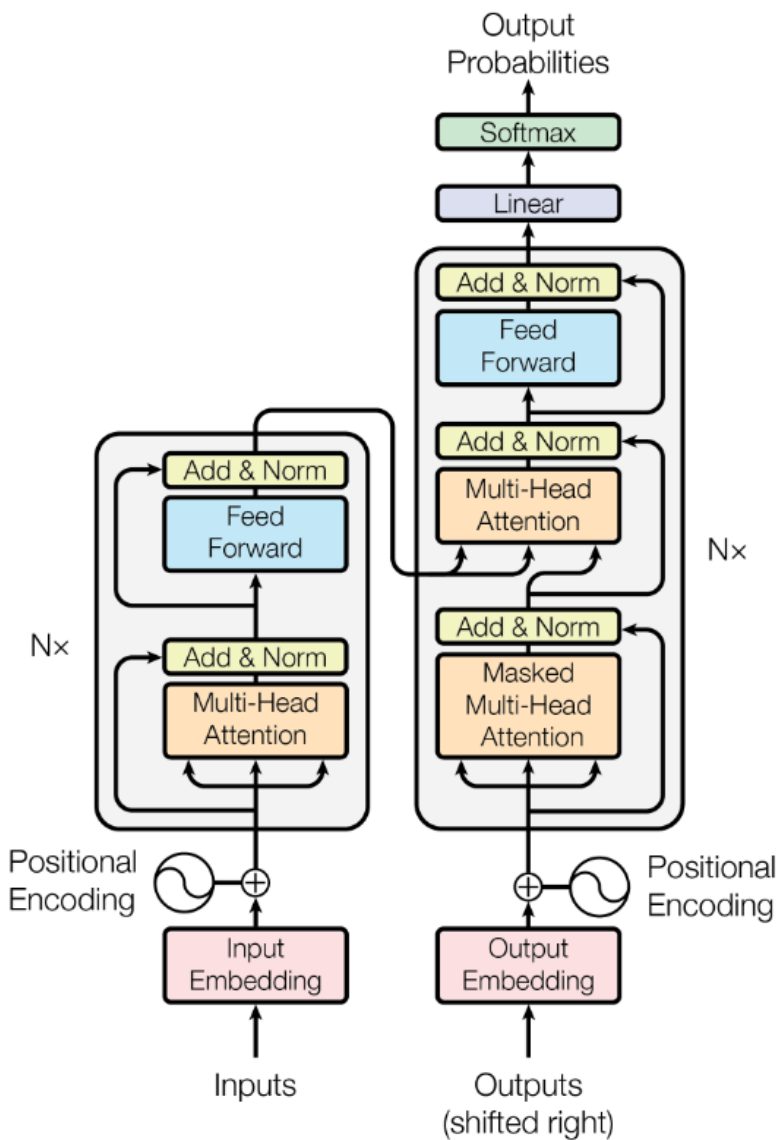
가중합(Weighted Sum)

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$

매번 디코더에서 새로운 단어를 출력할 때마다,
소스 문장을 전부 반영하여 사용할 수 있다.



Transformer

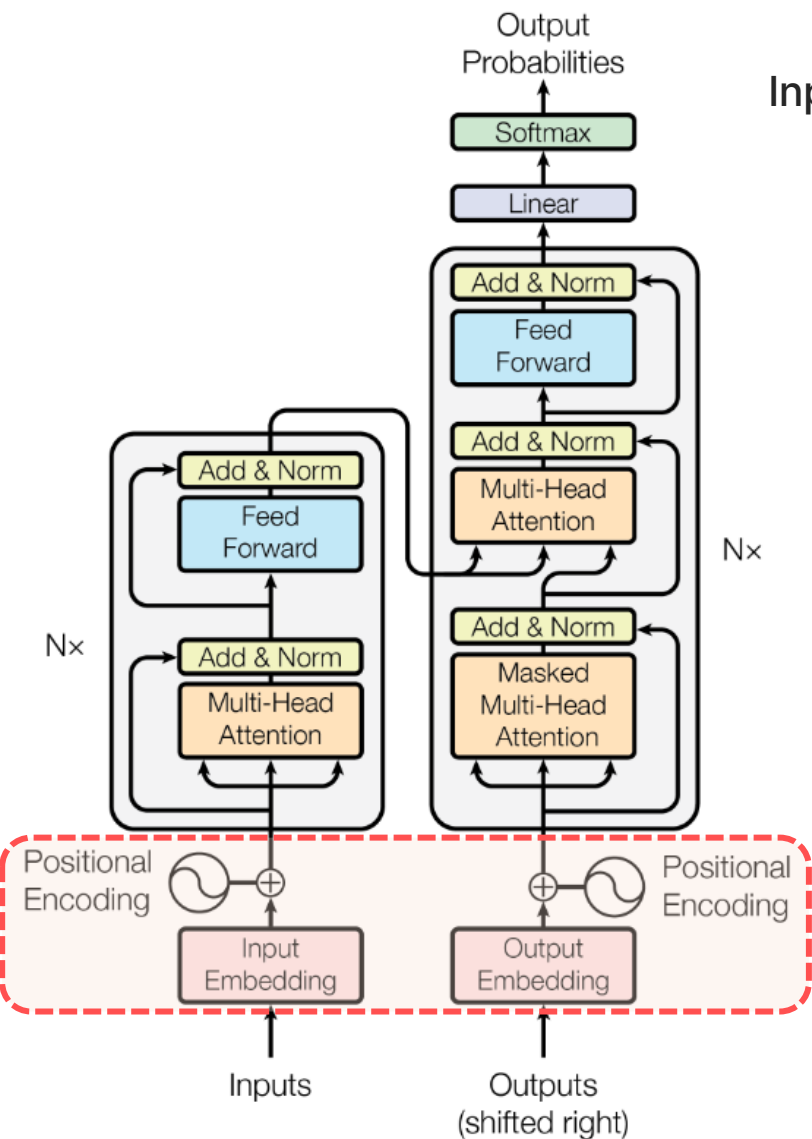


RNN 구조에서,
단어는 순서대로 모델에 입력되기 때문에, hidden state는 각 단어의 위치 정보를 포함한다.

하지만 Transformer 는 RNN 구조를 사용하지 않는다.
→ 단어의 위치 정보를 포함하는 임베딩을 사용해야 한다.

Transformer : Input Embedding

Input Embedding 은 입력으로 들어온 데이터를 컴퓨터가 이해할 수 있도록 행렬 값으로 바꾸어 준다.



Sentence

This is my car

①

Vocab (a aaron ... car ... is ... my ... this ... zombie)



Indices (0 1 ... 3412 ... 5281 ... 6899 ... 8678 ... 9999)

Vocab indices
= Input

X_0 X_1 X_2 X_3
8678 5381 6899 3412

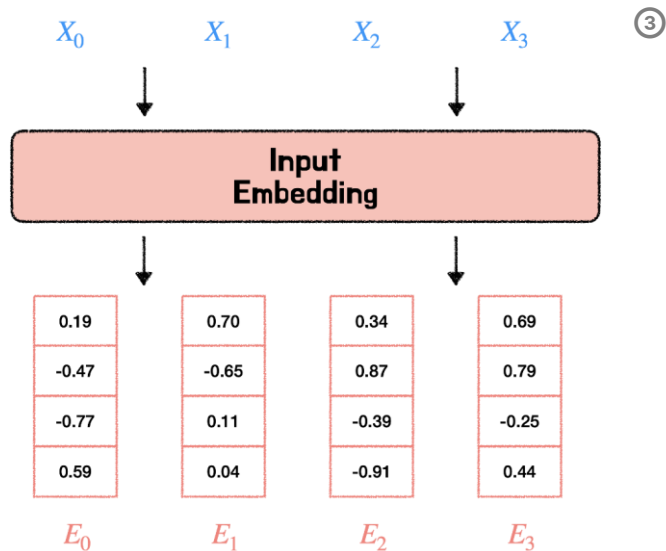
입력 문장이 주어졌을 때, 문장을 구성하는 각각의 단어는 그에 상응하는 인덱스 값에 매칭이 되고 인덱스 값들은 Input Embedding에 전달된다.

Transformer : Input Embedding



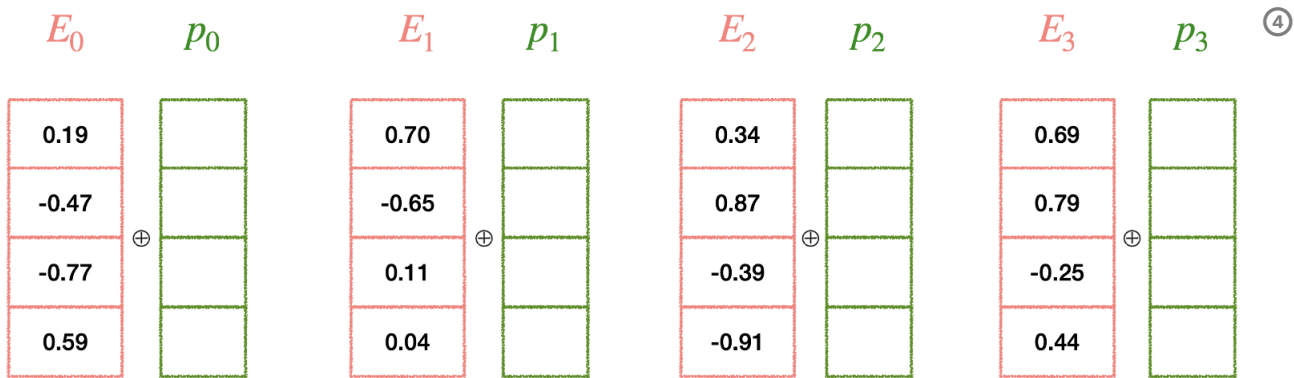
각각의 단어 인덱스들은 저마다 다른 벡터 값을 지니고 있다. (논문에서는 512차원의 벡터를 사용)

이 때 각각의 벡터 차원은 해당 단어의 Feature 값을 갖고 있고, 값이 유사할 수록 벡터 공간의 임베딩 벡터는 점점 가까워진다.



임베딩 레이어는 입력 인덱스 값들을 받아서 이를 각각의 단어 임베딩 벡터값으로 바꿔준다.

Transformer : Positional Encoding



위 그림처럼 각각의 단어 벡터에 Positional Encoding을 통해 얻은 위치 정보를 더해줘야 한다.

이 때 반드시 지켜야할 규칙 두 가지가 존재한다.

1. 모든 위치값은 시퀀스의 길이에 관계없이 동일한 식별자를 가져야한다.
2. 모든 위치값은 너무 크지 않아야 한다.

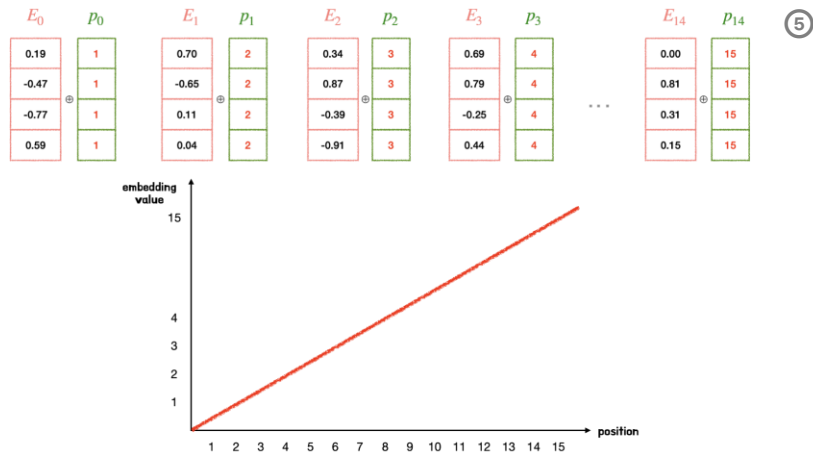
위치값이 너무 커져버리면 단어 간의 상관관계 및 의미를 유추할 수 있는 의미정보 값이 상대적으로 작아지게 되고, 학습 시 제대로 훈련되지 않을 수 있다.

Transformer : Positional Encoding

1. 모든 위치값은 시퀀스의 길이에 관계없이 동일한 식별자를 가져야한다.
2. 모든 위치값은 너무 크지 않아야 한다.

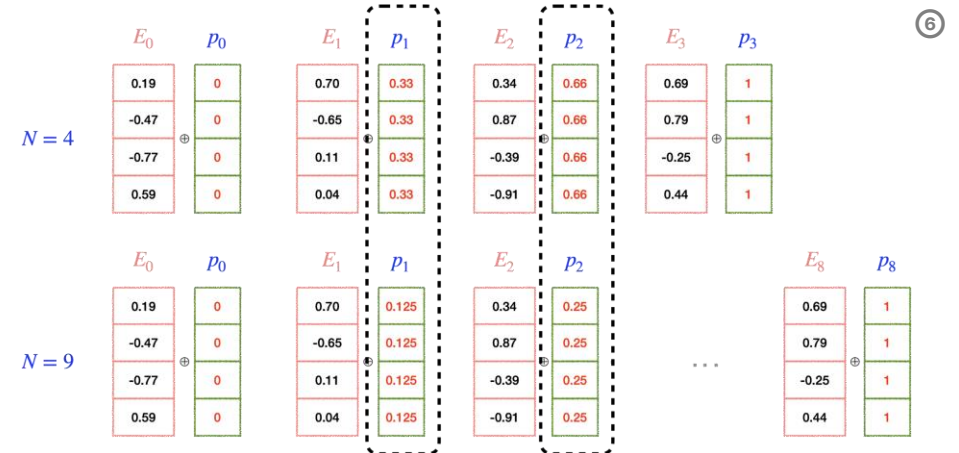
위치 벡터를 얻는 간단한 두 가지 방법과 문제점

1. 시퀀스 크기에 비례하여 일정하게 커지는 정수값을 부여하는 방식



위치 정보 값이 급격히 커져 단어 벡터와 더했을 때,
단어 자체의 정보보다 위치 정보가 지배적이라 단어의 의미가 훼손될 수 있다. (2번)

2. 첫 토큰에 0, 마지막 토큰에 1을 부여하는 정규화 방식



시퀀스 길이에 따라 같은 위치 정보에 해당하는 위치 벡터값이 달라질 수 있다.
바로 옆에 위치한 토큰들 간의 차이 역시 달라지는 문제점이 존재한다. (1번)

Transformer : Positional Encoding

1. 모든 위치값은 시퀀스의 길이에 관계없이 동일한 식별자를 가져야한다.
2. 모든 위치값은 너무 크지 않아야 한다.

Transformer의 Positional Encoding 함수

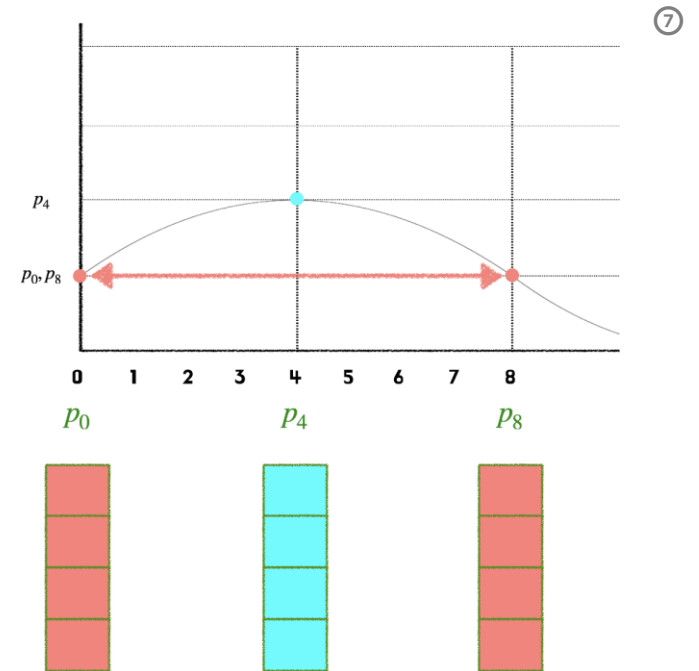
$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

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

- pos : 각 단어의 번호
- i : 단어의 임베딩 차원 인덱스
- d_{model} : 모델의 단어 임베딩 차원

sine 과 cosine 함수는 -1 ~ 1 사이를 반복하는 주기함수이다. 2번 조건을 만족한다.

하지만 주기함수이기 때문에 토큰들의 위치벡터값이 같은 경우가 생길 수 있지 않을까?



Transformer : Positional Encoding

주기함수이기 때문에 토큰들의 위치벡터값이 같은 경우가 생길 수 있지 않을까?

Positional Encoding은 스칼라값이 아닌 **벡터값**으로 **단어벡터와 같은 차원을 지닌 벡터값**이다.

따라서 위치벡터 값이 같아지는 문제를 해결하기 위해, 다양한 주기의 sine & cosine 함수를 동시에 사용한다.

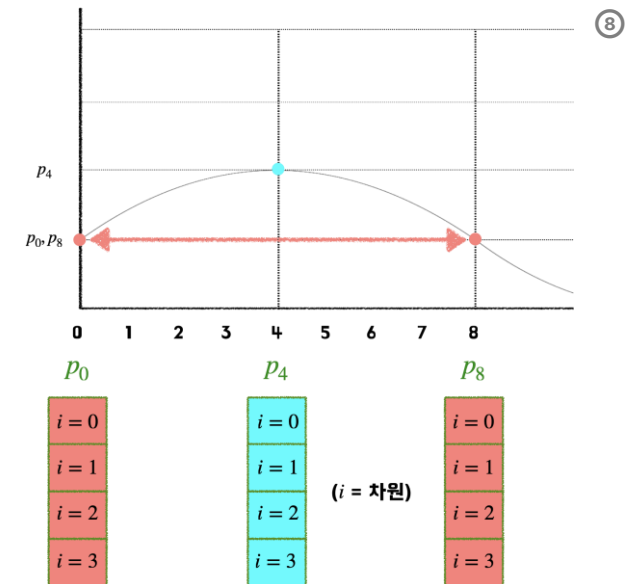
하나의 위치벡터가 4개의 차원으로 표현된다면, 각 요소는 서로 다른 4개의 주기를 갖게 때문에 서로 겹치지 않는다.

Transformer의 Positional Encoding 함수

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

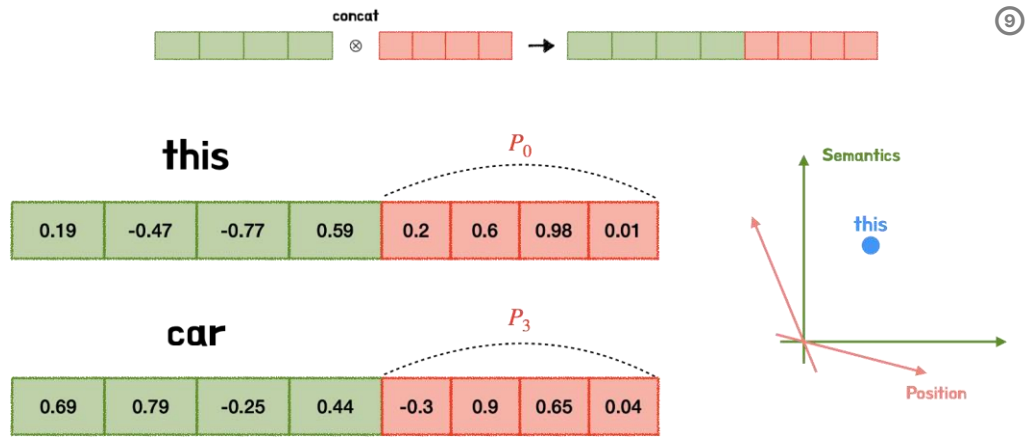
$$PE_{(pos, 2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

- pos : 각 단어의 번호
- i : 단어의 임베딩 차원 인덱스
- d_{model} : 모델의 단어 임베딩 차원



Transformer : Positional Encoding

왜 Concatenate 대신에 Summation 연산을 사용했을까?



Concatenate를 사용했을 때,

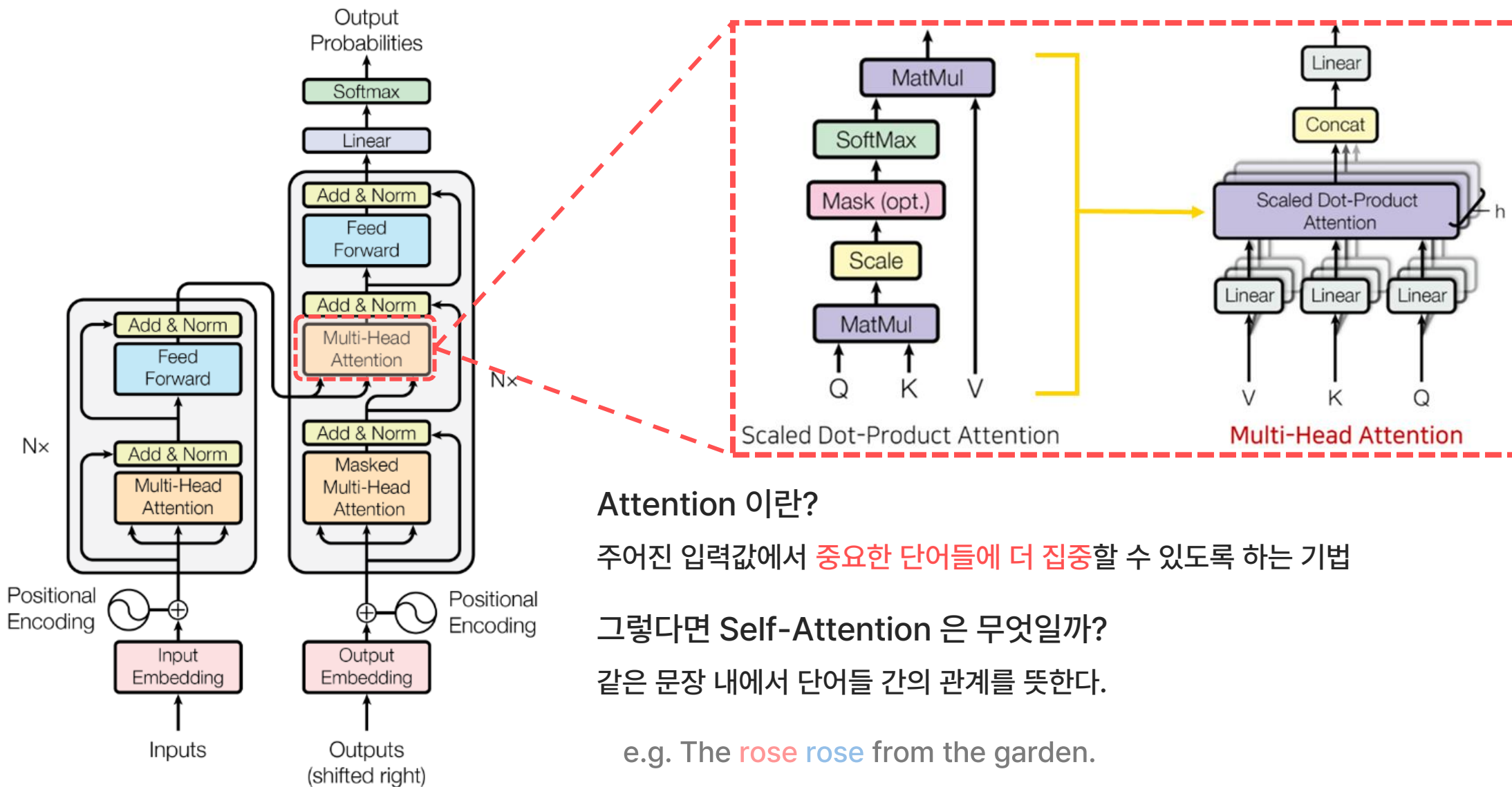
장점: 단어의 의미 정보와 위치 정보가 각 자체 차원 공간을 갖게되어 정보가 뒤섞이는 혼란을 피할 수 있게 해준다.

단점: 메모리, 파라미터, 런타임 등 비용 문제가 발생한다.

Summation을 사용한다면 단어 의미 정보와 위치 정보 간의 균형을 잘 맞출 수 있다.

하지만 GPU 등의 자원이 충분하고 비용 문제가 발생하지 않는다면 Concatenate 방식을 사용해도 무관하다.

Transformer : Self-Attention



Attention 이란?

주어진 입력값에서 **중요한 단어들에 더 집중**할 수 있도록 하는 기법

그렇다면 Self-Attention 은 무엇일까?

같은 문장 내에서 단어들 간의 관계를 뜻한다.

e.g. The **rose** **rose** from the garden.

Transformer : Self-Attention

Attention 은 입력 시퀀스에서 어떤 단어가 다른 단어들과 어떠한 연관성을 가지고 있는가를 알아보기 위함이다.

Query

- 입력 시퀀스에서 관련된 부분을 찾으려고 하는 정보 벡터 (소스)
- 관계성, 즉 연관된 정도를 표현하는 가중치를 계산하는데 사용

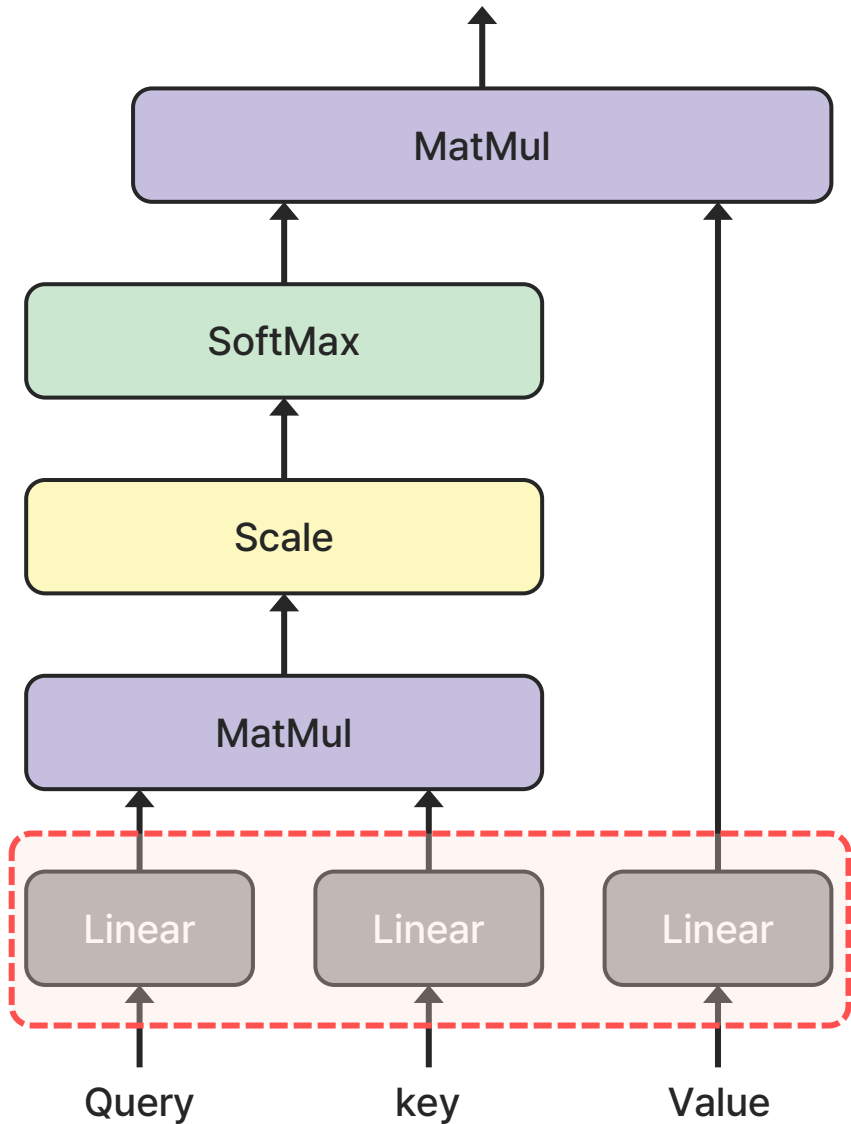
Key

- 관계의 연관도를 결정하기 위해 Query와 비교하는데 사용되는 벡터 (타겟)
- 관계성, 즉 연관된 정도를 표현하는 가중치를 계산하는데 사용

Value

- 특정 Key에 해당하는 입력 시퀀스의 정보로 가중치를 구하는데 사용되는 벡터 (밸류)
- 관계성을 표현하는 가중치 합이 최종 출력을 계산하는데 사용

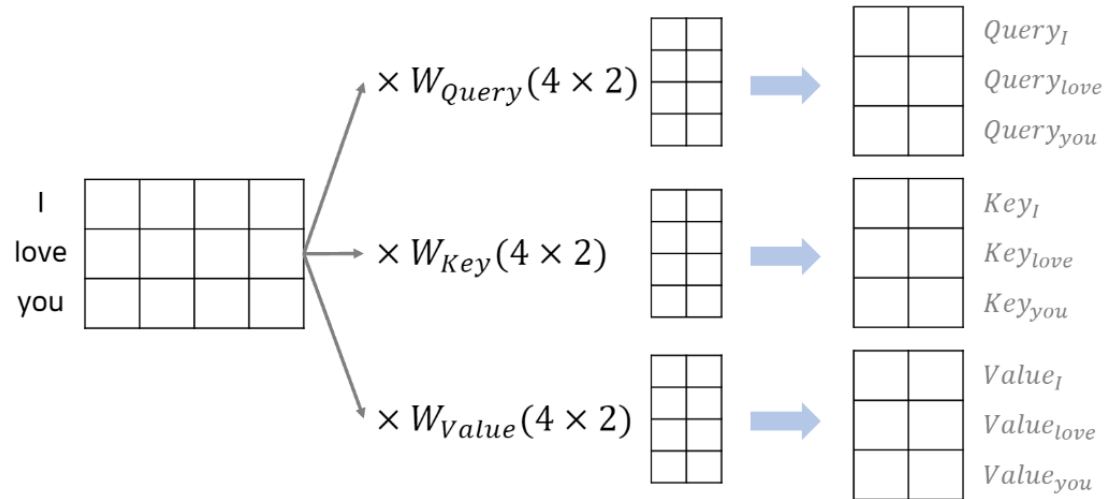
Transformer : Self-Attention



Linear 연산을 거치는 이유

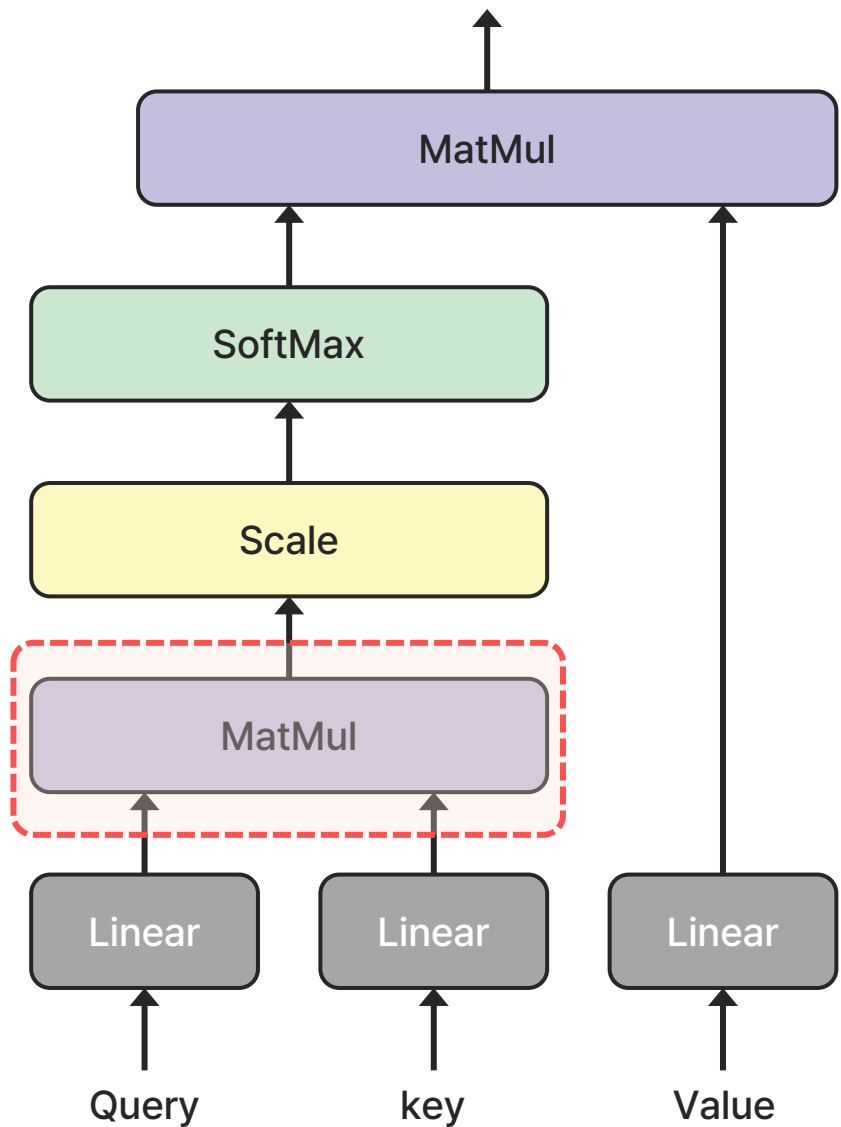
Linear Layer는 행렬이나 벡터의 차원을 바꿔주는 역할을 한다.
즉 Query, Key, Value 각각의 차원을 줄여
병렬 연산에 적합한 구조를 만드려는 목적이 있기 때문이다.

이 때 사용되는 행렬이 W^Q, W^K, W^V 이다.



위 그림처럼 실제로는 행렬 연산을 통해 한꺼번에 계산이 가능하다.

Transformer : Self-Attention



Query와 Key의 전치행렬을 내적하는 이유

두 행렬 간 내적을 할 때 $(A \times B) * (C \times D)$

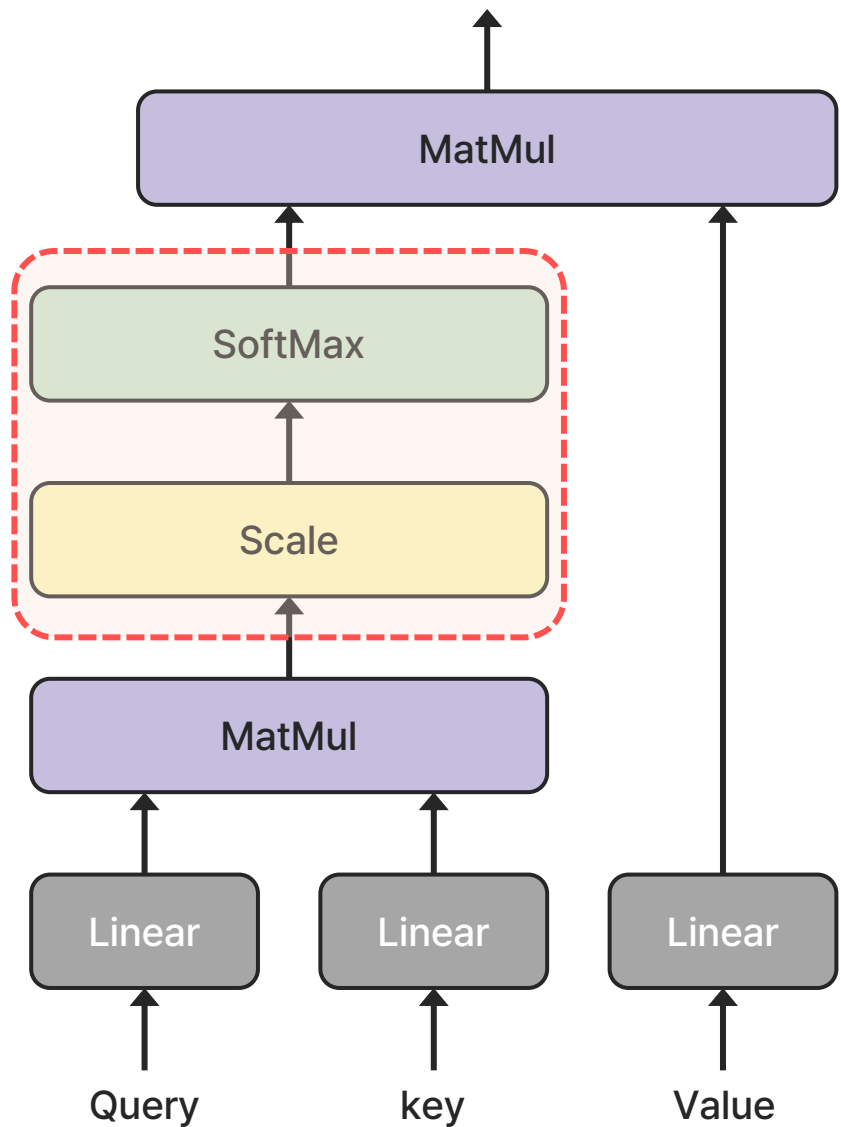
B와 C의 차원이 동일해야 내적이 가능하며, $(A \times D)$ 행렬이 결과로 나온다.

따라서, 같은 차원($n_{word} \times d_{model}$)의 Query와 Key를 내적하기 위해서는 Query와 Key^T 의 형태로 내적을 해야한다.

결과로 $(n_{word} \times n_{word})$ 행렬, 즉 **Attention Score** 가 나온다.

The diagram shows the matrix multiplication for Self-Attention. On the left, a 3x2 matrix is labeled with rows $Query_I$, $Query_{love}$, and $Query_{you}$. This is multiplied by a 2x3 matrix with columns K_I , K_{love} , and K_{you} . The result is a 3x3 matrix labeled with rows I , $love$, and you , and columns I , $love$, and you . This resulting matrix is enclosed in a red dashed border and labeled 'Attention Energies'.

Transformer : Self-Attention



Scaling을 하는 이유

내적 계산은 특성 상 문장의 길이가 길어질 수록 더 큰 숫자를 가지게 된다.

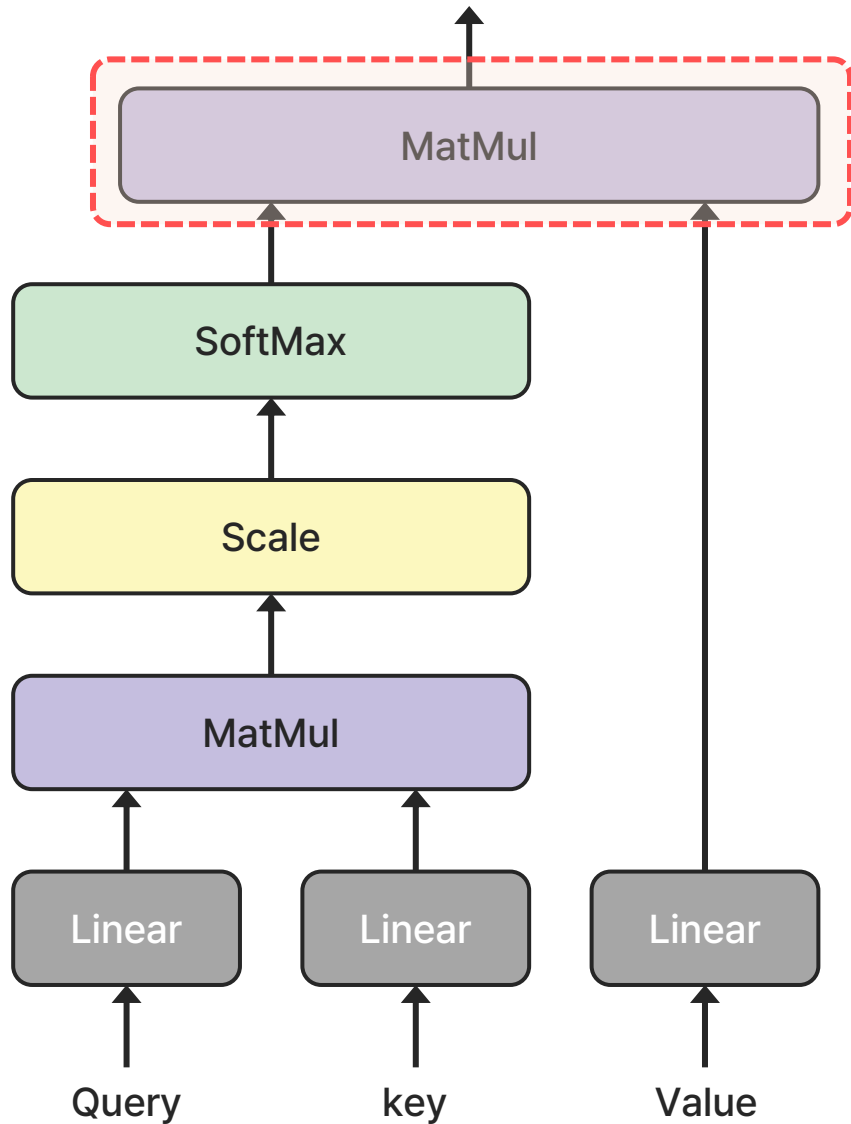
큰 숫자들에 Softmax를 취하게 되면 특정값만 과도하게 살아남고 나머지 값들은 완전히 죽어버리는 과한 정제가 이루어져 버린다.

이를 완화하기 위하여 원래 값을 Scaled-down 해주면 Softmax 이후에도 살아남는 gradient가 충분히 많아질 수 있다.

Scaling Factor는 Key 벡터 차원의 제곱근이다.

$$\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

Transformer : Self-Attention

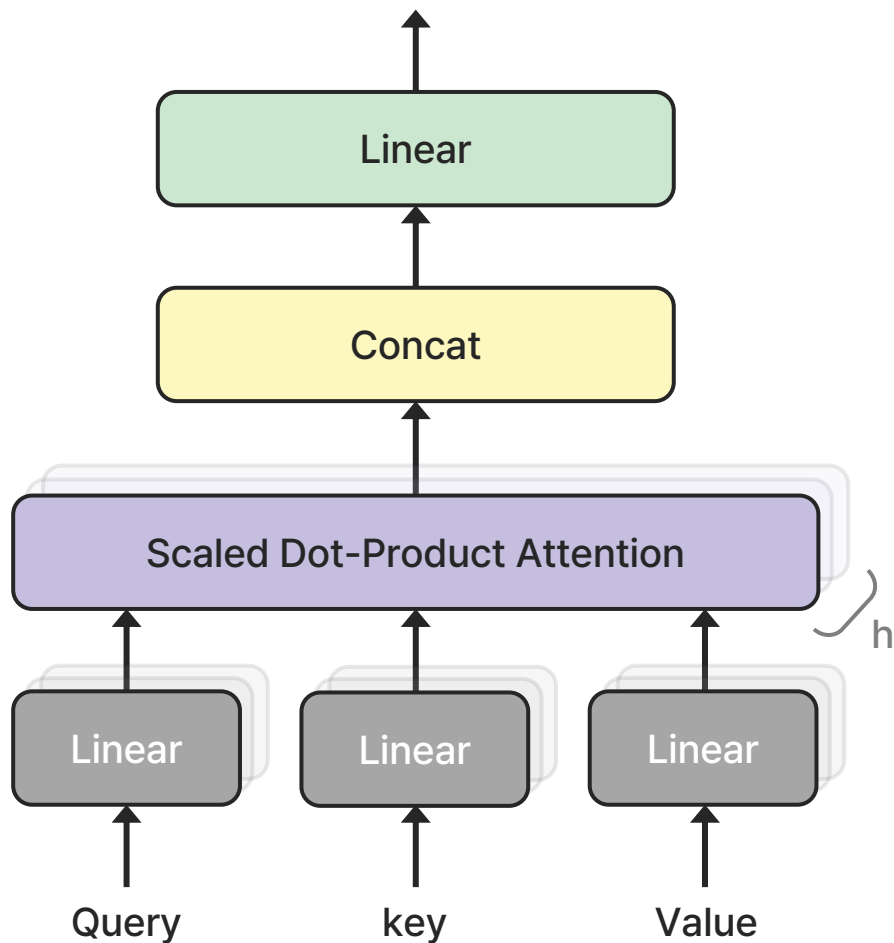


마지막으로 Attention Score 와 앞서 구했던 Value를 내적하면 Self-Attention Value를 구할 수 있다.

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

Transformer : Multi-Head Attention

Transformer는 Self-Attention을 **병렬로 h번 학습**시키는 Multi-Head Attention 구조로 이루어져 있다.



Which **do** you like better, coffee or tea?

- 문장 타입에 집중하는 어텐션 ^⑩

Which do **you** like better, **coffee** or **tea**?

- 명사에 집중하는 어텐션

Which do you **like** better, **coffee** or tea?

- 관계에 집중하는 어텐션

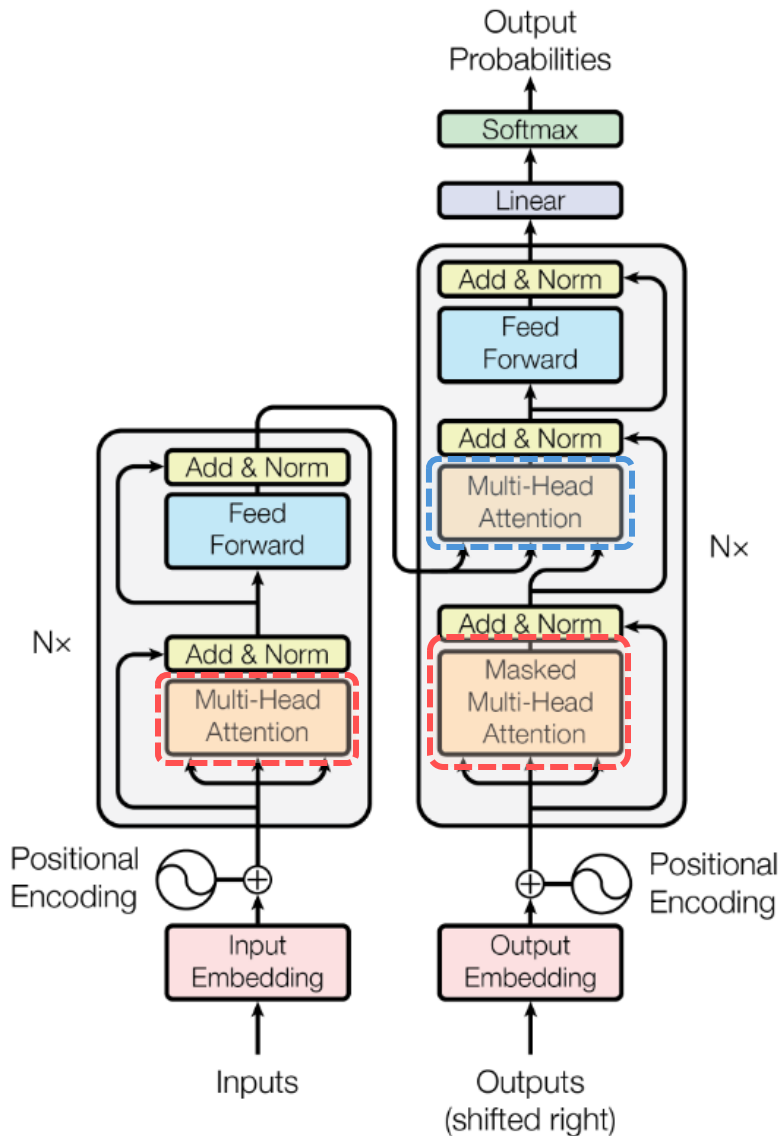
Which do you **like better**, coffee or tea?

- 강조에 집중하는 어텐션

각 Head는 입력 시퀀스의 서로 다른 부분에 Attention을 하기 때문에 모델이 **입력 토큰 간의 더 복잡한 관계**를 다룰 수 있다.

또한 **다양한 유형의 종속성을 포착**할 수 있어 **표현력이 향상**될 수 있고 토큰 간의 미묘한 관계 역시 더 잘 포착할 수 있다.

Transformer : Encoder & Decoder



성능 향상을 위해 **잔여 학습**(Residual Learning)을 사용한다.

인코더와 디코더는 여러 개의 레이어로 구성되며
각 레이어는 서로 다른 파라미터를 가진다.



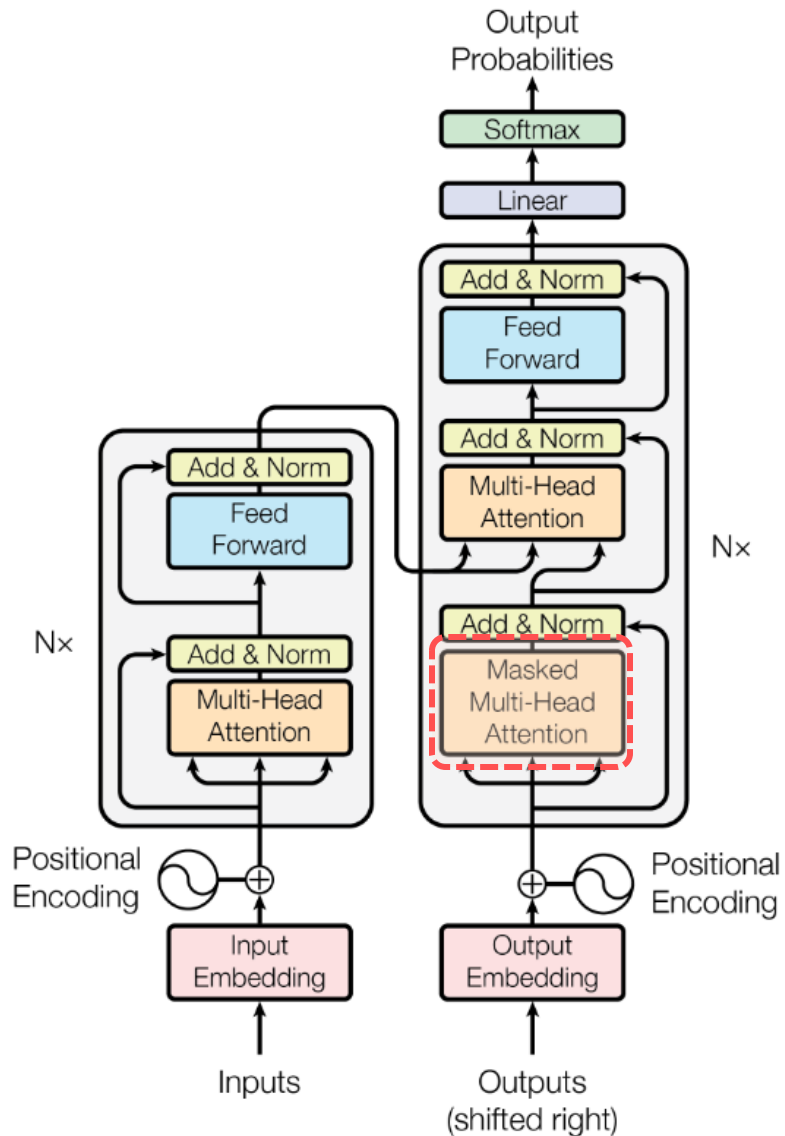
마지막 인코더 레이어의 출력은 모든 디코더의 입력으로 들어간다.

 : Self-Attention이라고도 불린다.

 : Encoder-Decoder Attention이라고도 불린다.

디코더의 출력이 **Query**, 인코더의 출력이 **Key**, **Value**로 동작한다.

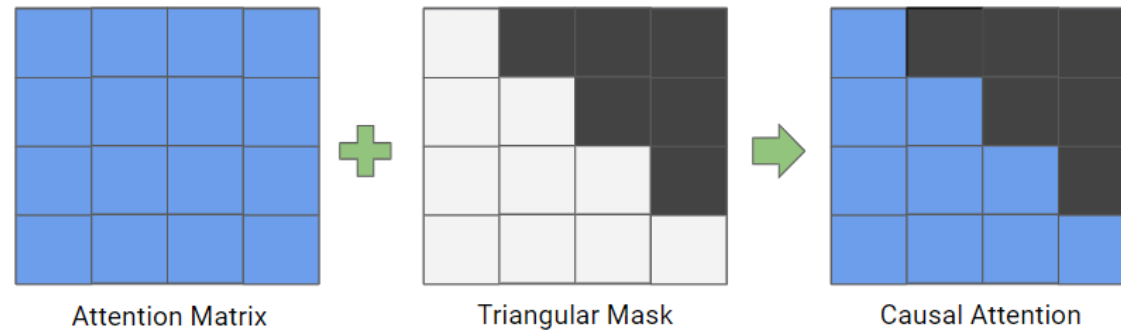
Transformer : Decoder Masked Matrix



인코더는 입력 시퀀스를 전체를 보고 시퀀스에 대한 이해를 한다.
하지만 디코더는 **이전 단어를 통해 다음에 올 단어를 예측**한다.

기본적인 Self-Attention 시에,
이전 단어들 정보만으로 현재 단어를 예측해야 하는 상황에서
미래의 단어를 미리보는 즉, **컨닝**을 하게되는 일이 발생한다.

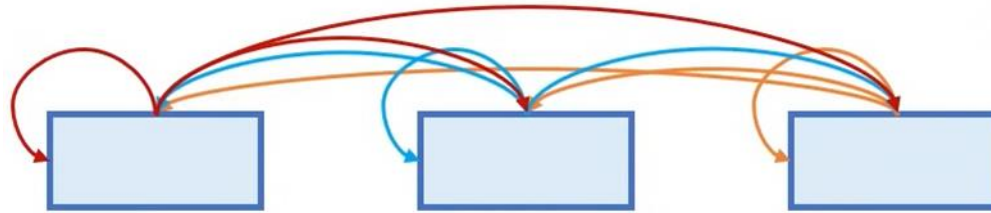
마스크 행렬(Mask Matrix)를 이용해 특정 단어는 무시할 수 있다.



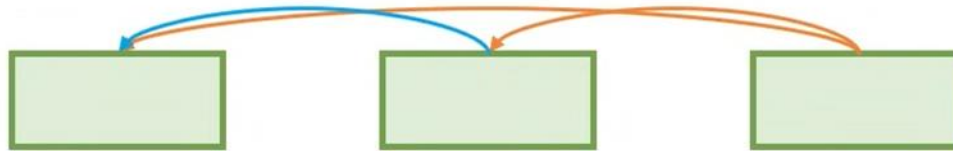
마스크 값으로 $-\infty$ 를 넣어 softmax 함수의 출력이 0에 가까워지도록 한다.

Transformer : Attention의 종류

Encoder Self-Attention:



Masked Decoder Self-Attention:



Encoder-Decoder Attention:

