자연어 처리: 트랜스포머

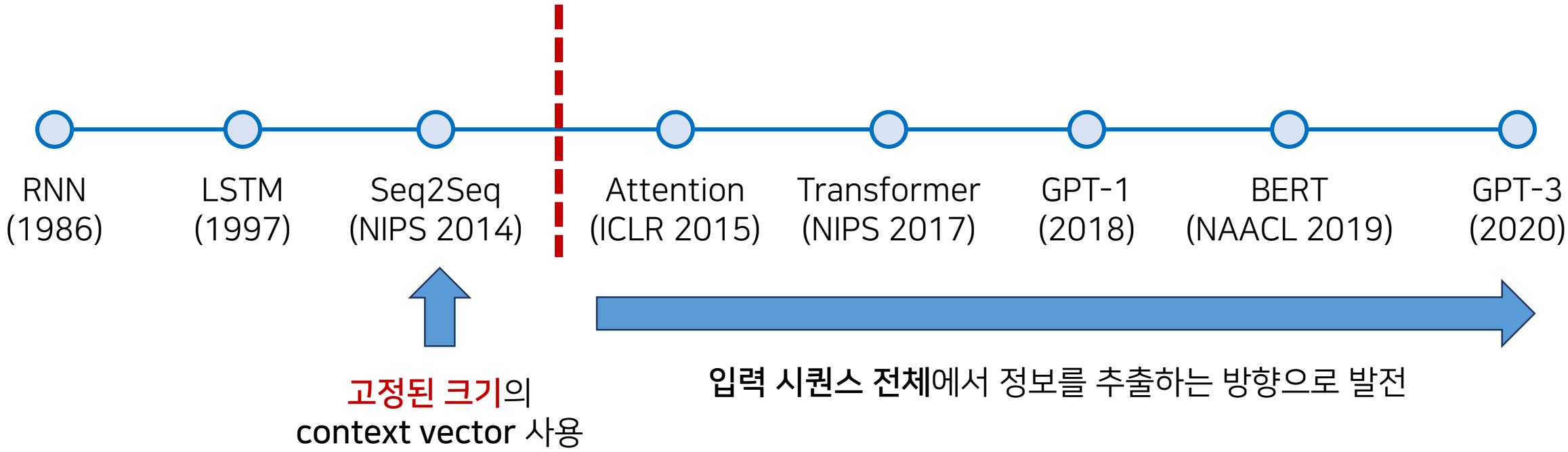
Transformer (Attention Is All You Need)

나동빈(dongbinna@postech.ac.kr)

Pohang University of Science and Technology

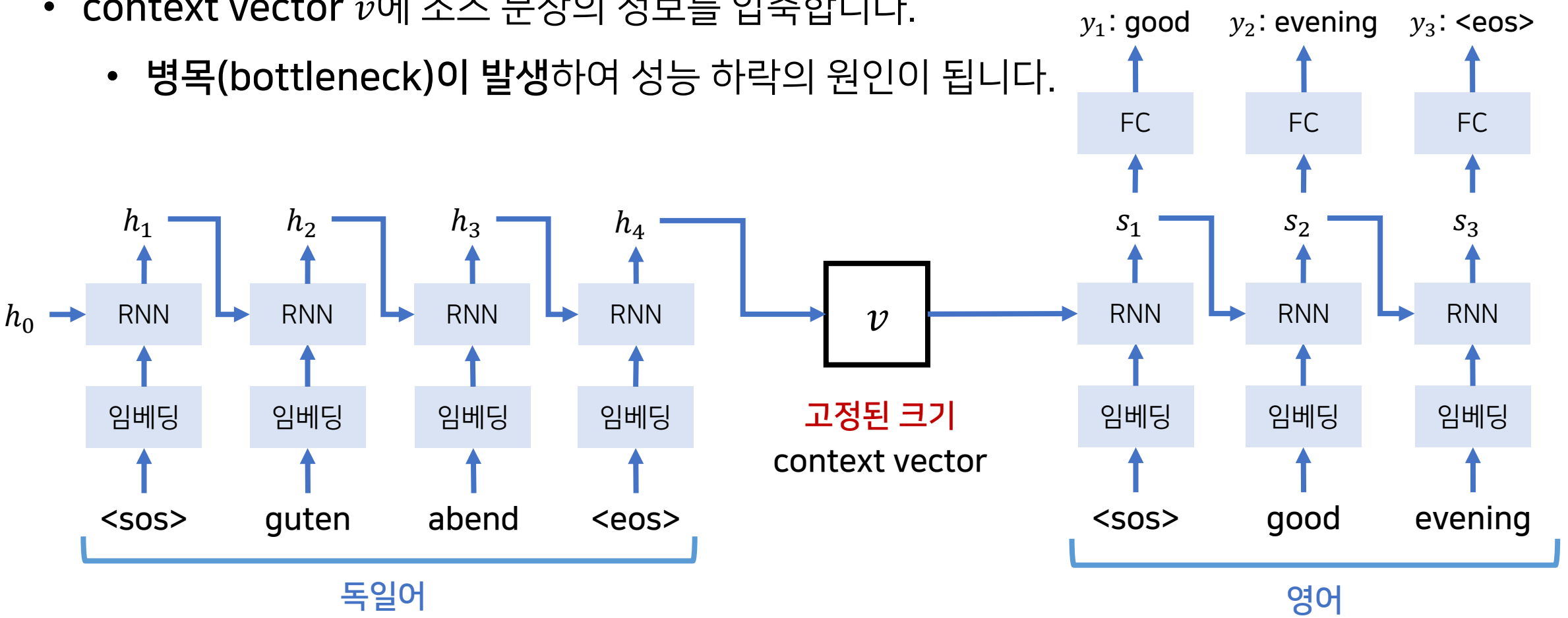
딥러닝 기반의 기계 번역 발전 과정

- 2021년 기준으로 최신 고성능 모델들은 Transformer 아키텍처를 기반으로 하고 있습니다.
  - **GPT**: Transformer의 **디코더(Decoder)** 아키텍처를 활용
  - **BERT**: Transformer의 **인코더(Encoder)** 아키텍처를 활용



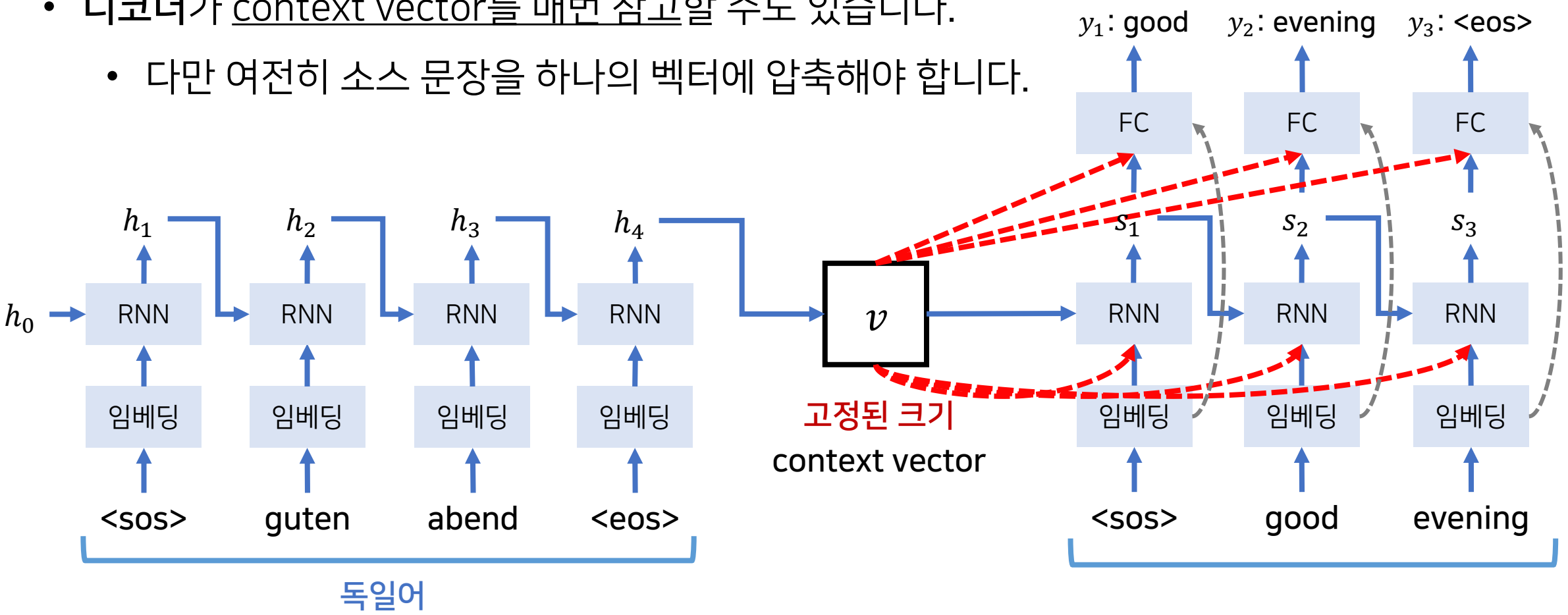
기존 Seq2Seq 모델들의 한계점

- context vector v에 소스 문장의 정보를 압축합니다.
  - 병목(bottleneck)이 발생하여 성능 하락의 원인이 됩니다.



기존 Seq2Seq 모델들의 한계점

- **디코더**가 context vector를 매번 참고할 수도 있습니다.
  - 다만 여전히 소스 문장을 하나의 벡터에 압축해야 합니다.



Seq2Seq with Attention

[문제 상황]

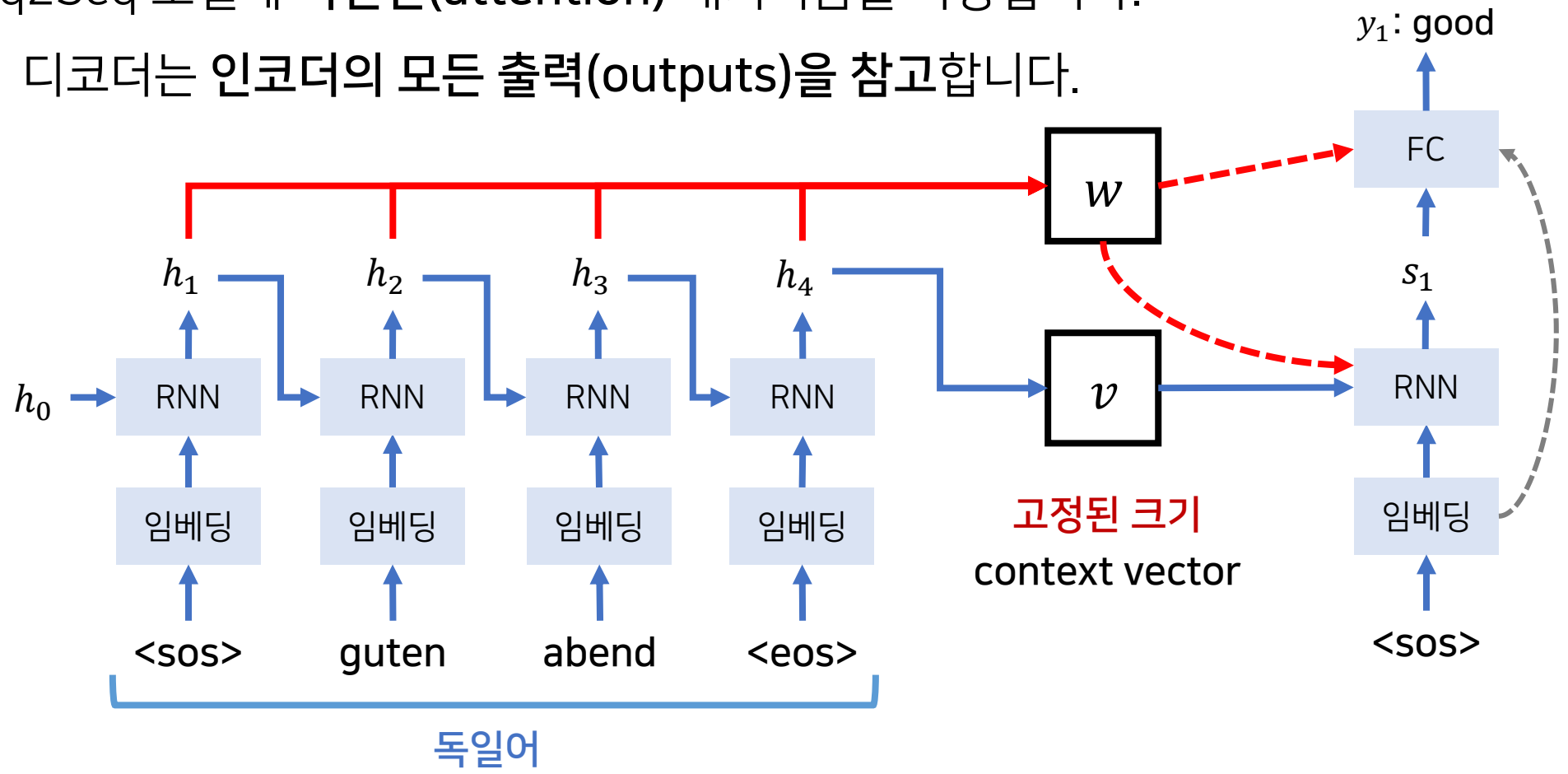
- 하나의 문맥 벡터가 소스 문장의 모든 정보를 가지고 있어야 하므로 성능이 저하됩니다.

[해결 방안]

- 그렇다면 **매번 소스 문장에서의 출력 전부를 입력으로** 받으면 어떨까요?
  - 최신 GPU는 많은 메모리와 빠른 병렬 처리를 지원합니다.

Seq2Seq with Attention

- Seq2Seq 모델에 **어텐션(attention)** 매커니즘을 사용합니다.
  - 디코더는 **인코더의 모든 출력(outputs)을 참고**합니다.



Seq2Seq with Attention: 디코더(Decoder)

- 디코더는 **매번 인코더의 모든 출력 중에서 어떤 정보가 중요한지를 계산**합니다.
  - i = 현재의 디코더가 처리 중인 인덱스
  - j = 각각의 인코더 출력 인덱스

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

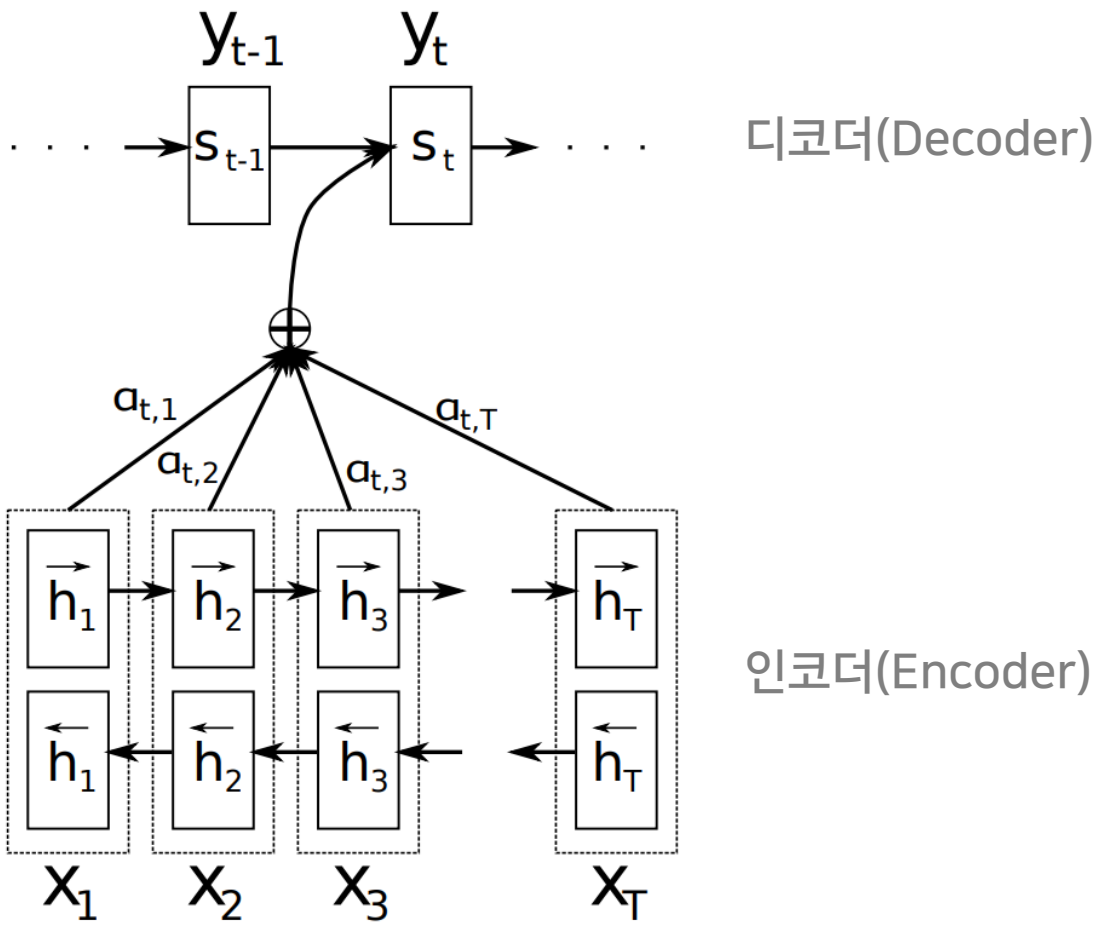
  - **가중치(Weight)** $\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$

Seq2Seq with Attention: 디코더(Decoder)

- 에너지(Energy) $e_{ij} = a(s_{i-1}, h_j)$
- 가중치(Weight) $\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$

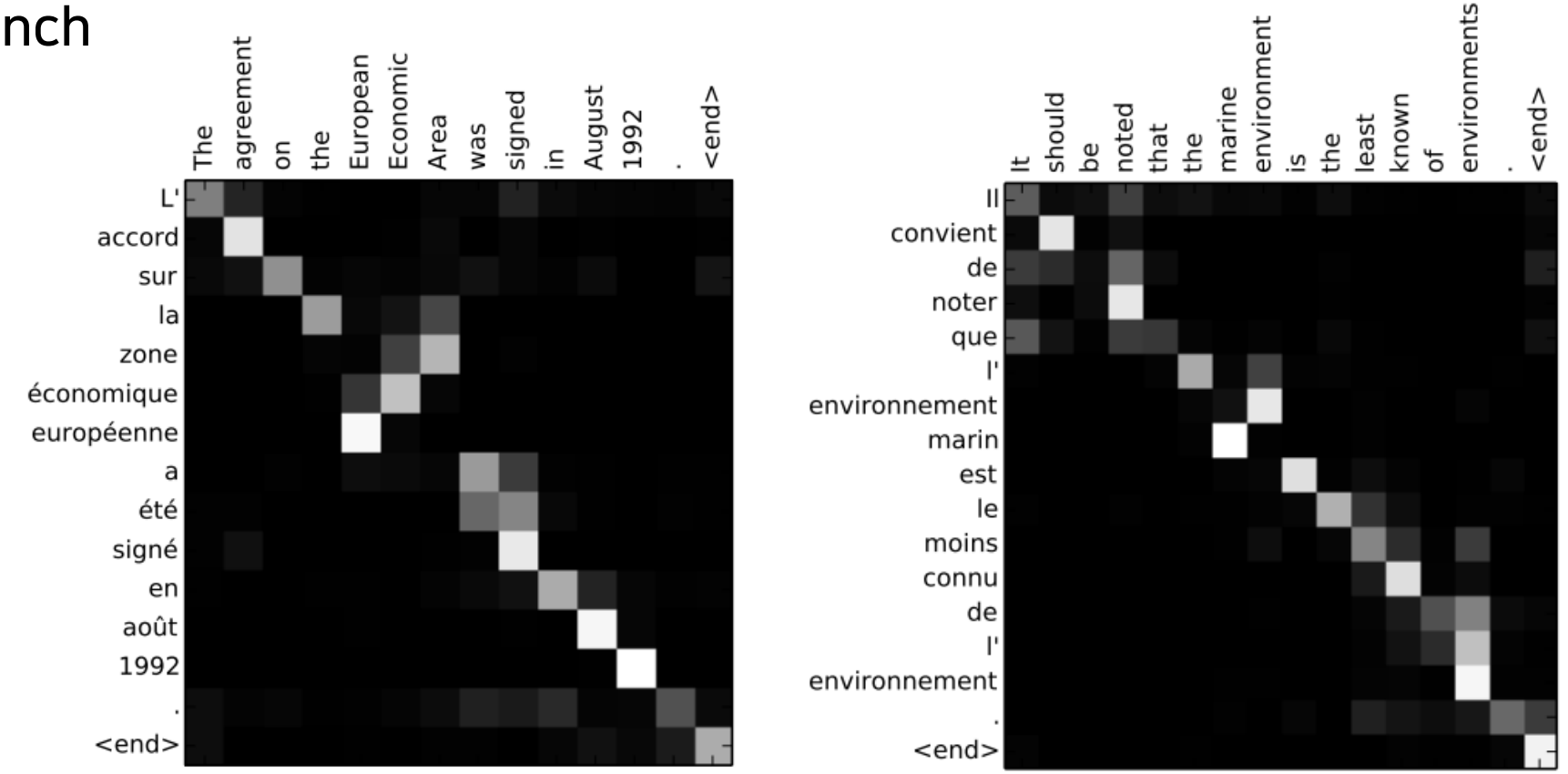
Weighted sum 이용

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



Seq2Seq with Attention: 어텐션 시각화

- 어텐션(attention) 가중치를 사용해 각 출력이 어떤 입력 정보를 참고했는지 알 수 있습니다.
  - English → French

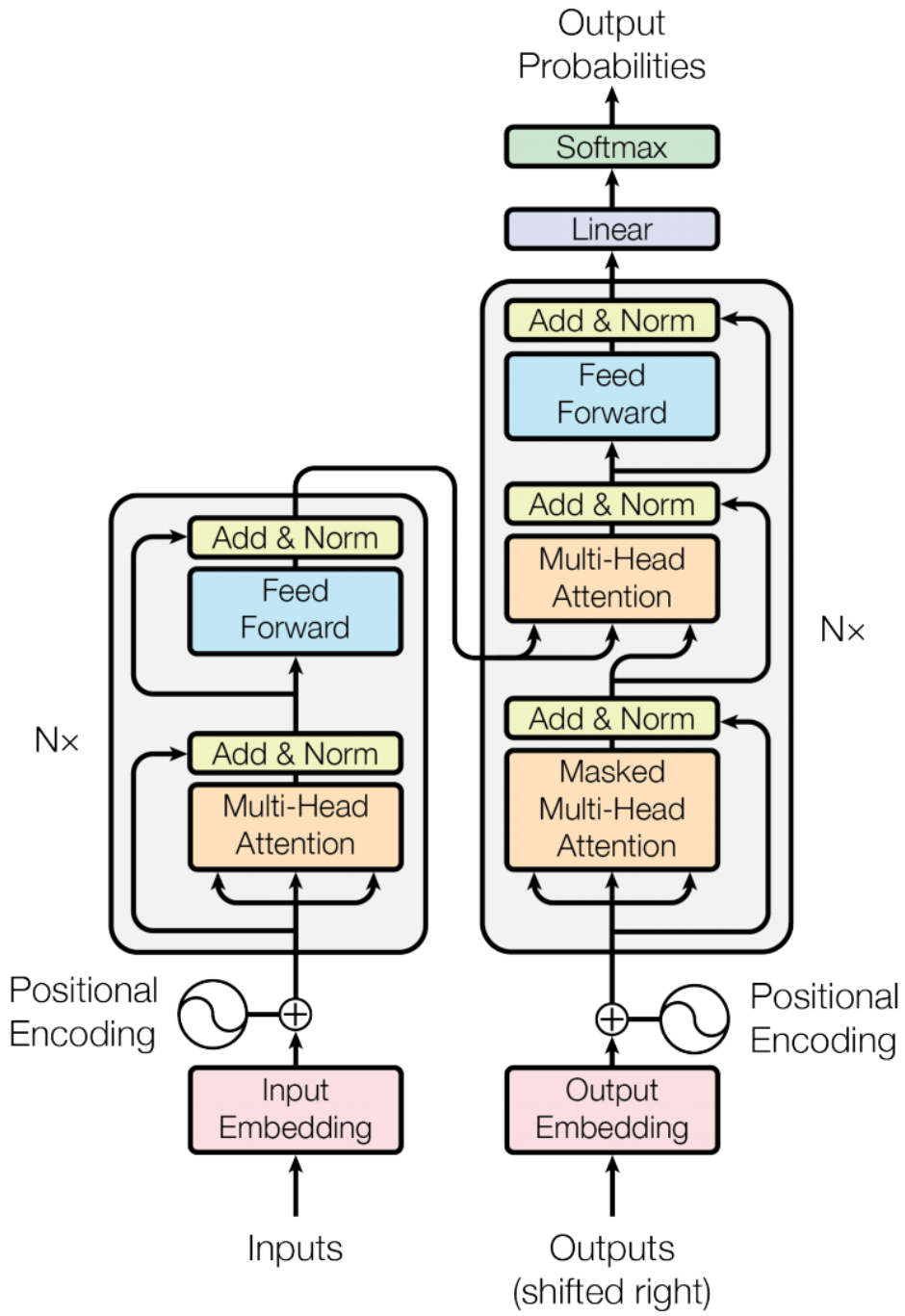


트랜스포머(Transformer)

- 2021년 기준으로 현대의 자연어 처리 네트워크에서 핵심이 되는 논문입니다.
  - 논문의 원제목은 Attention Is All You Need입니다.
- 트랜스포머는 RNN이나 CNN을 전혀 필요로 하지 않습니다.

트랜스포머(Transformer)

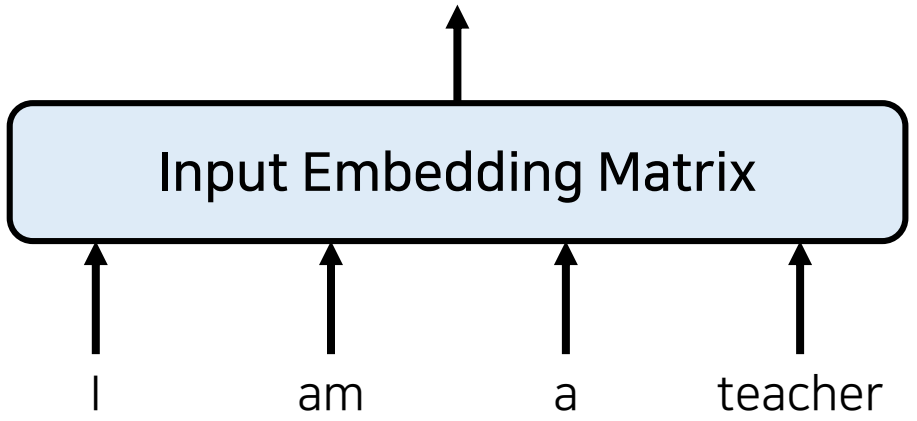
- 트랜스포머는 RNN이나 CNN을 전혀 사용하지 않습니다.
  - 대신 **Positional Encoding**을 사용합니다.
- BERT와 같은 향상된 네트워크에서도 채택되고 있습니다.
- 인코더와 디코더로 구성됩니다.
  - Attention 과정을 여러 레이어에서 반복합니다.



Attention is All You Need (NIPS 2017)

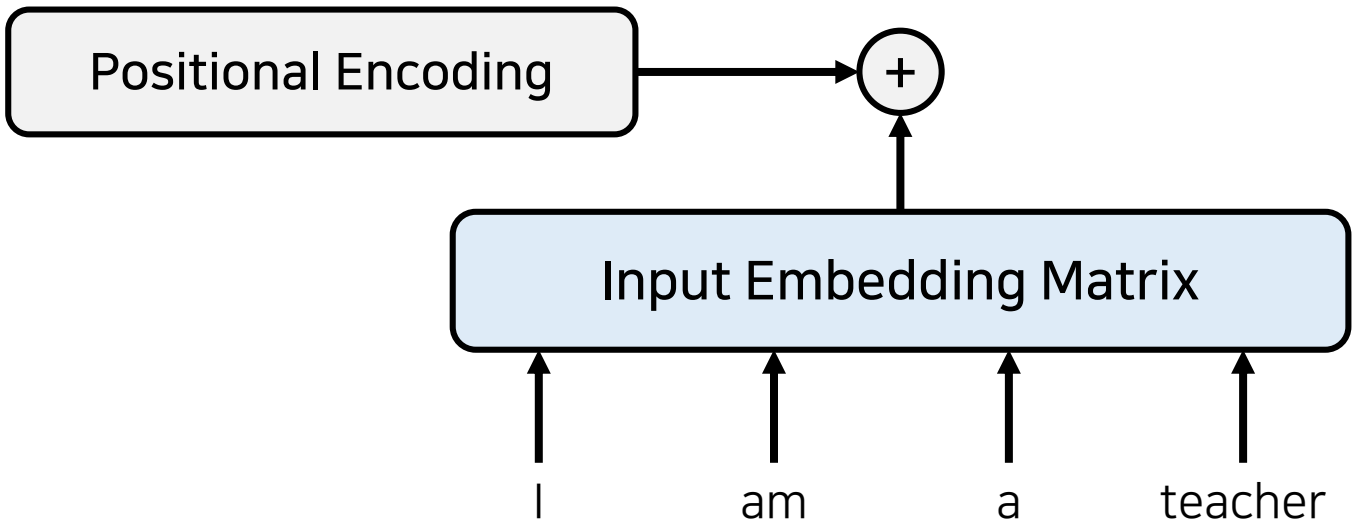
트랜스포머의 동작 원리: 입력 값 임베딩(Embedding)

- 트랜스포머 이전의 **전통적인 임베딩**은 다음과 같습니다.



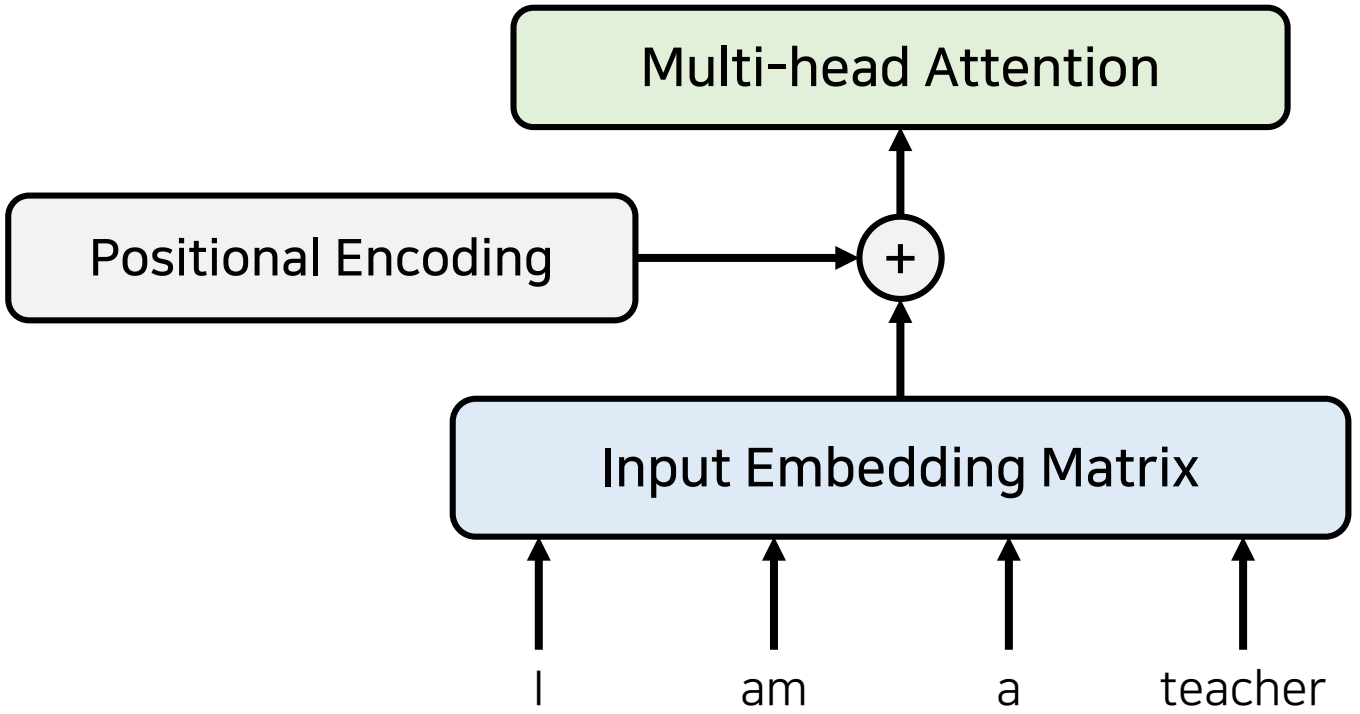
트랜스포머의 동작 원리: 입력 값 임베딩(Embedding)

- RNN을 사용하지 않으려면 위치 정보를 포함하고 있는 임베딩을 사용해야 합니다.
  - 이를 위해 트랜스포머에서는 Positional Encoding을 사용합니다.



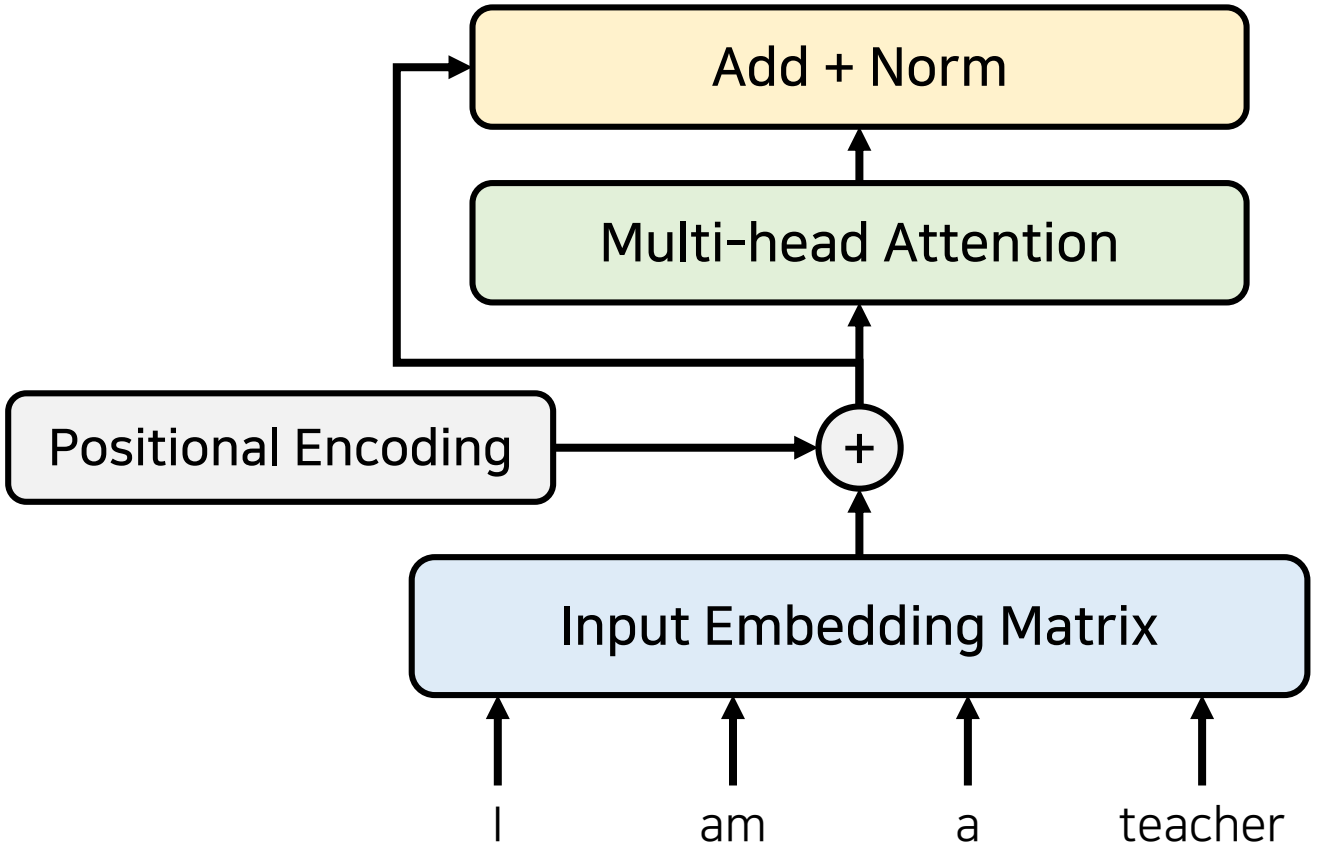
트랜스포머의 동작 원리: 인코더(Encoder)

- 임베딩이 끝난 이후에 **어텐션(Attention)을 진행**합니다.



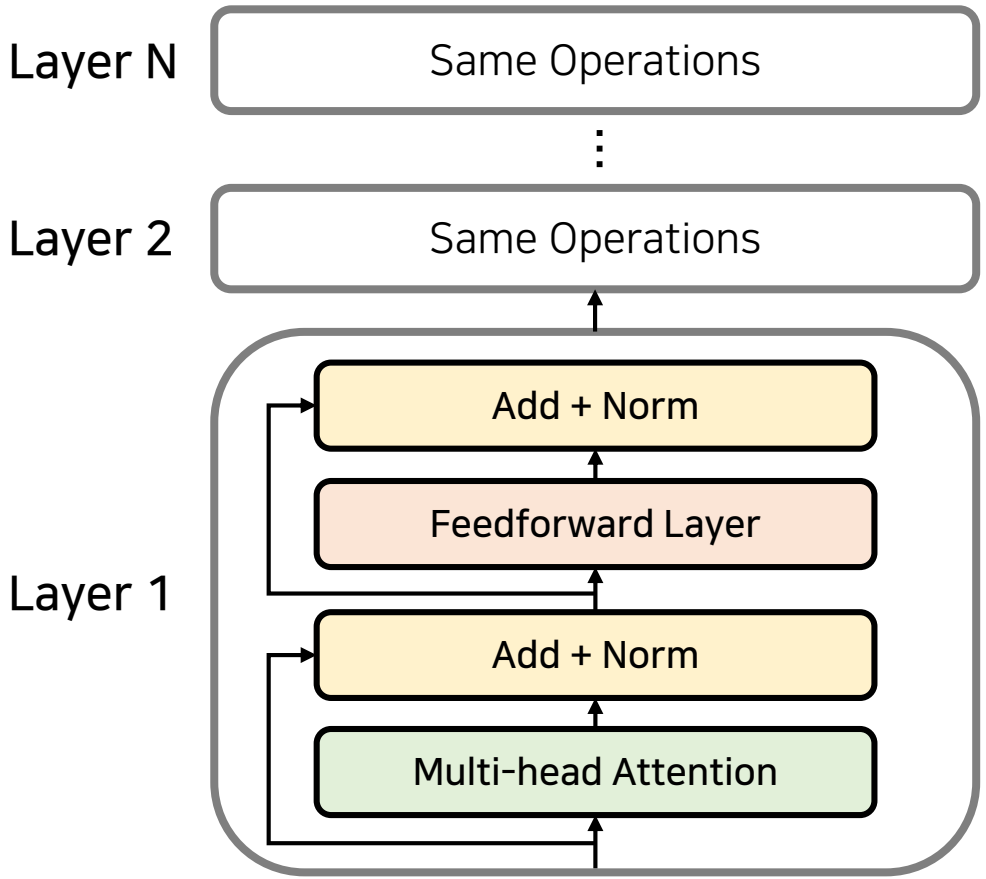
트랜스포머의 동작 원리: 인코더(Encoder)

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

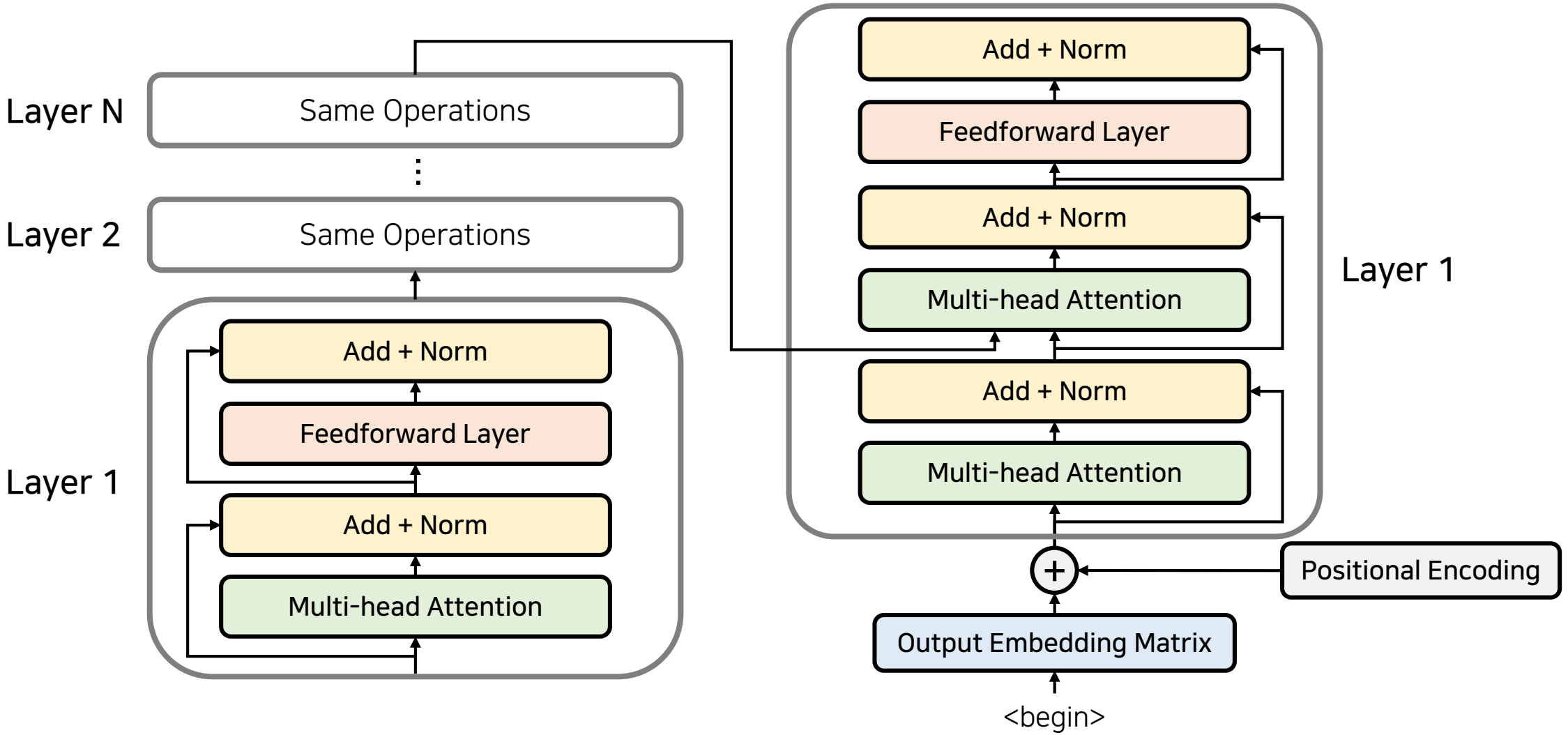


트랜스포머의 동작 원리: 인코더(Encoder)

- 어텐션(Attention)과 정규화(Normalization) 과정을 반복합니다.
  - **각 레이어는 서로 다른 파라미터**를 가집니다.

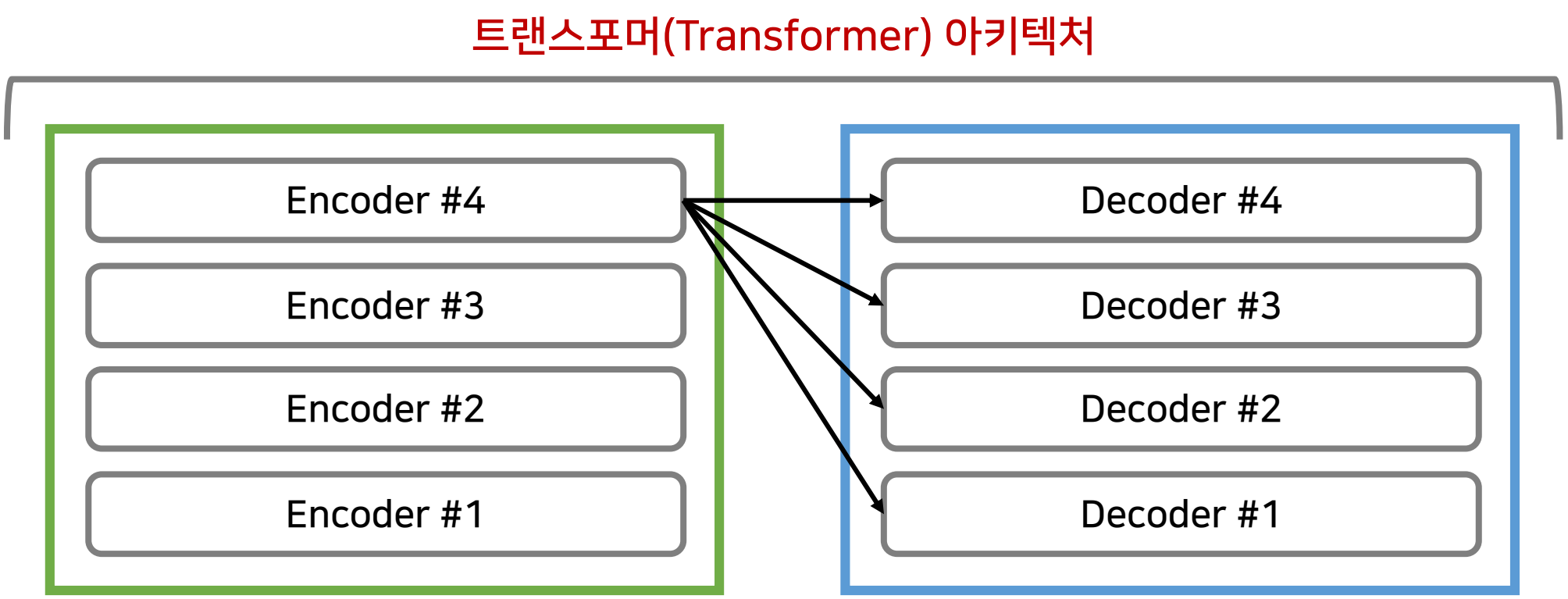


트랜스포머의 동작 원리: 인코더(Encoder)와 디코더(Decoder)



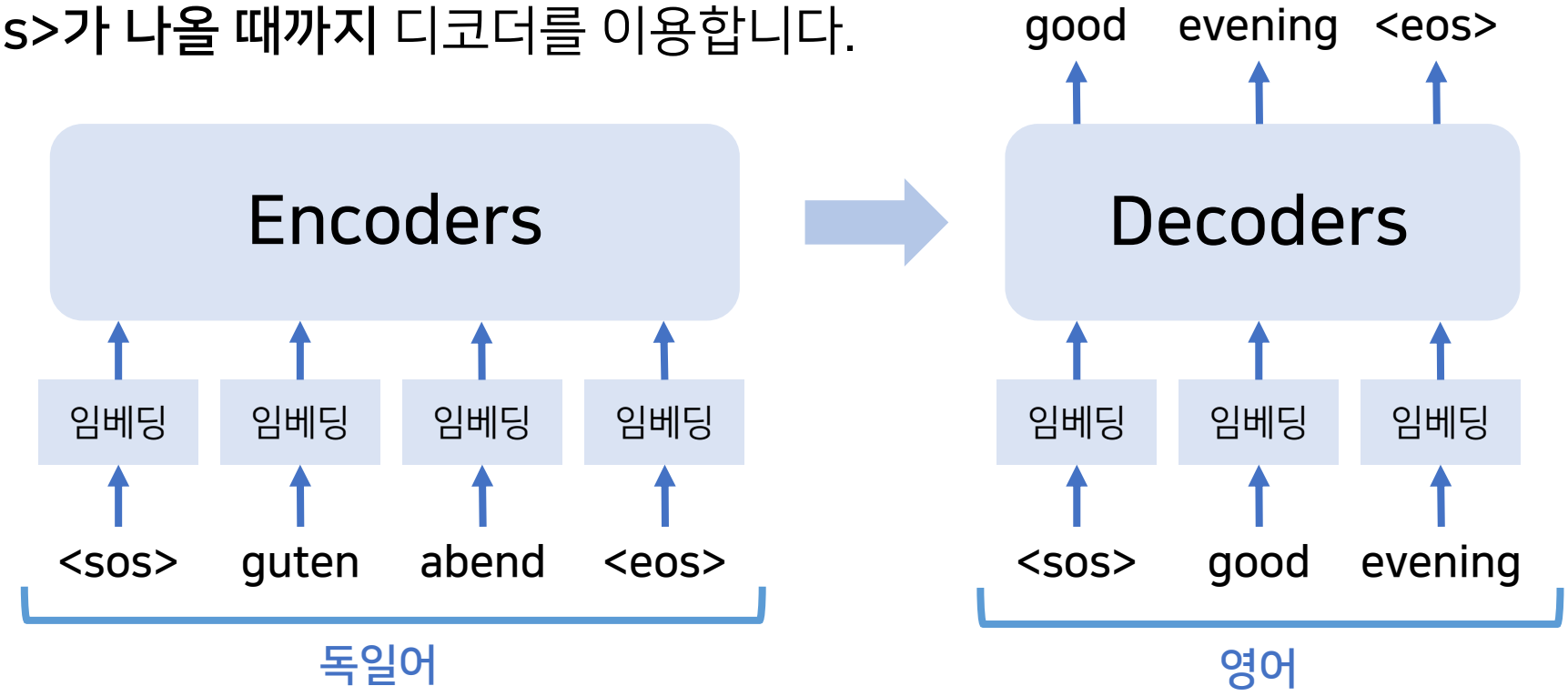
트랜스포머의 동작 원리: 인코더(Encoder)와 디코더(Decoder)

- 트랜스포머에서는 **마지막 인코더 레이어의 출력**이 모든 디코더 레이어에 입력됩니다.
  - n_layers = 4일 때의 예시는 다음과 같습니다.



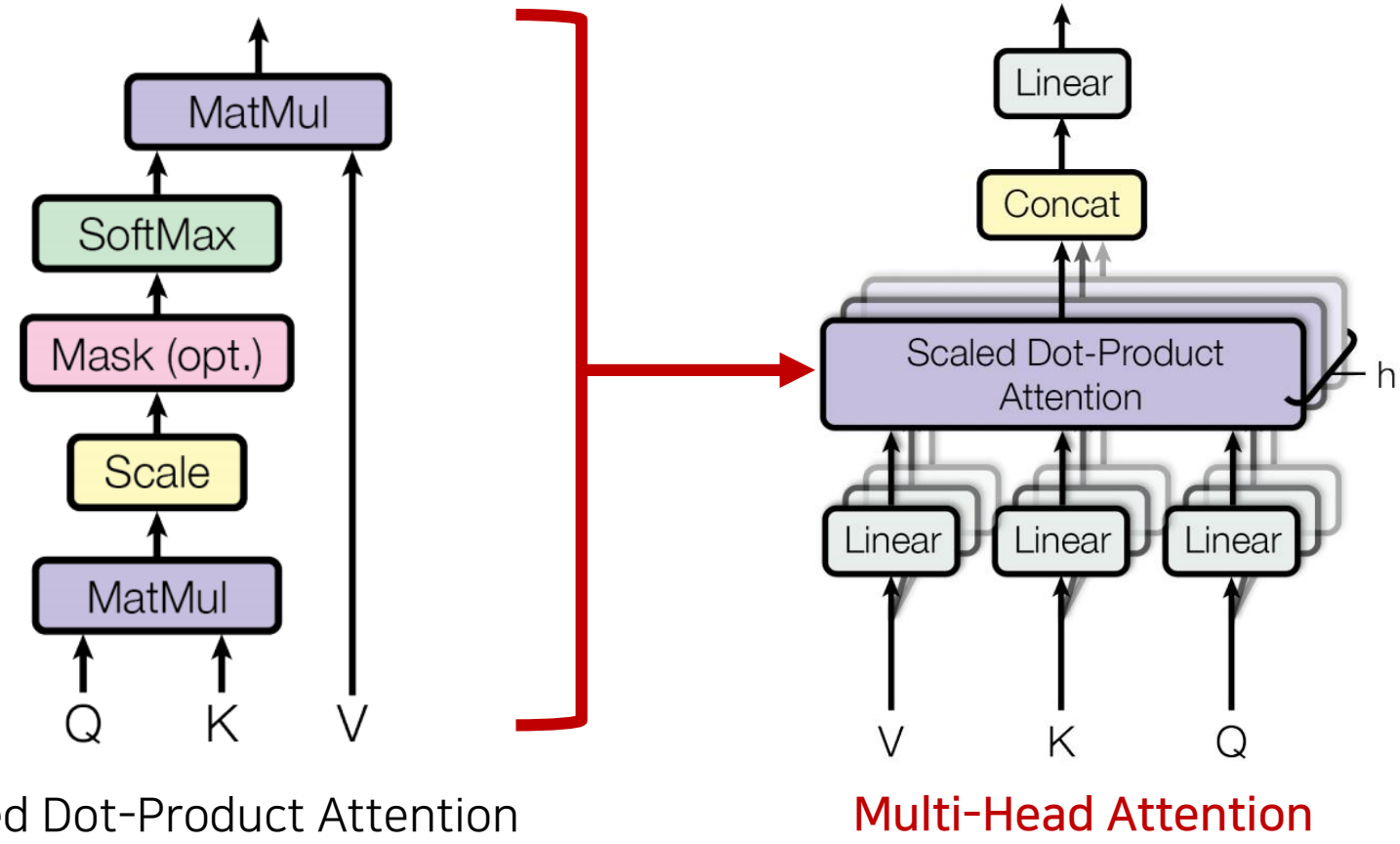
트랜스포머의 동작 원리: 인코더(Encoder)와 디코더(Decoder)

- 트랜스포머에서도 인코더(Encoder)와 디코더(Decoder)의 구조를 따릅니다.
  - 이때 **RNN을 사용하지 않으며 인코더와 디코더를 다수 사용**한다는 점이 특징입니다.
  - **<eos>가 나올 때까지** 디코더를 이용합니다.



트랜스포머의 동작 원리: 어텐션(Attention)

- 인코더와 디코더는 **Multi-Head Attention 레이어**를 사용합니다.
- 어텐션을 위한 **세 가지** 입력 요소
  - 쿼리(Query)
  - 키(Key)
  - 값(Value)



Scaled Dot-Product Attention

Multi-Head Attention

트랜스포머의 동작 원리: 어텐션(Attention)

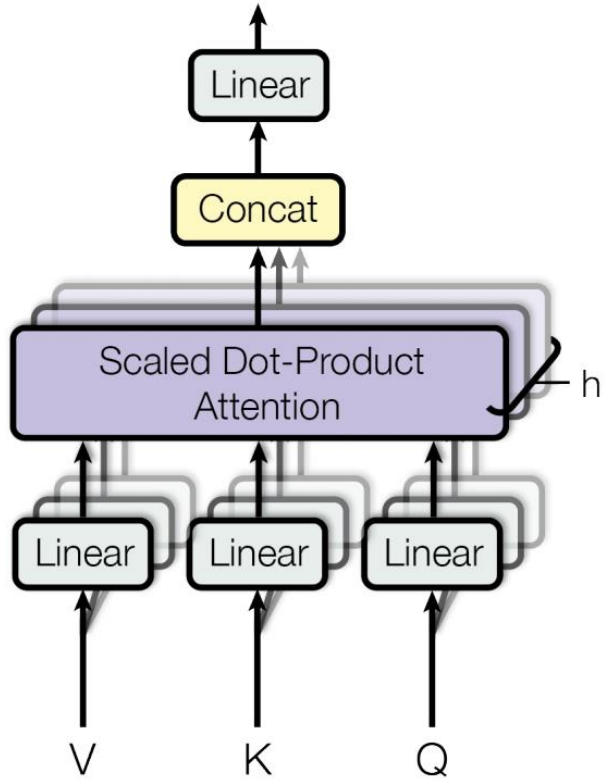
- 인코더와 디코더는 Multi-Head Attention 레이어를 사용합니다.

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

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

$$MultiHead(Q, K, V) = Concat(head_1, \dots, head_h)W^O$$

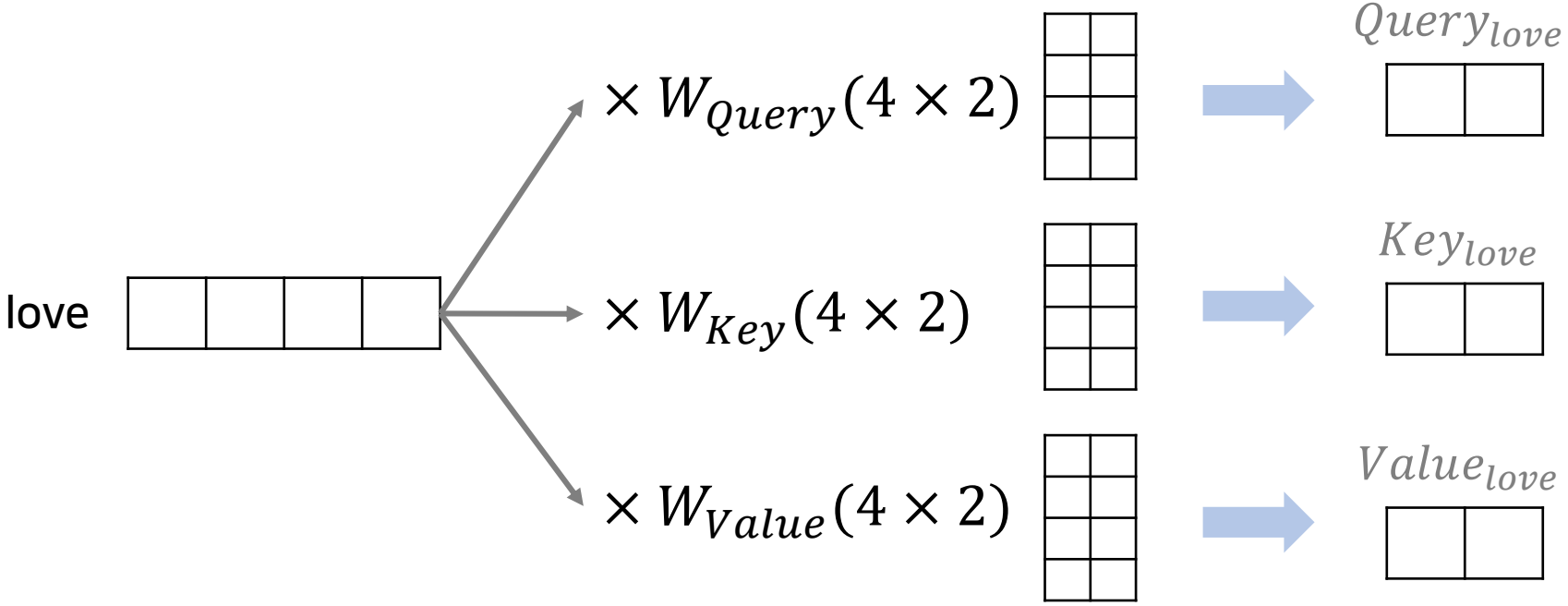
h: 헤드(head)의 개수



Multi-Head Attention

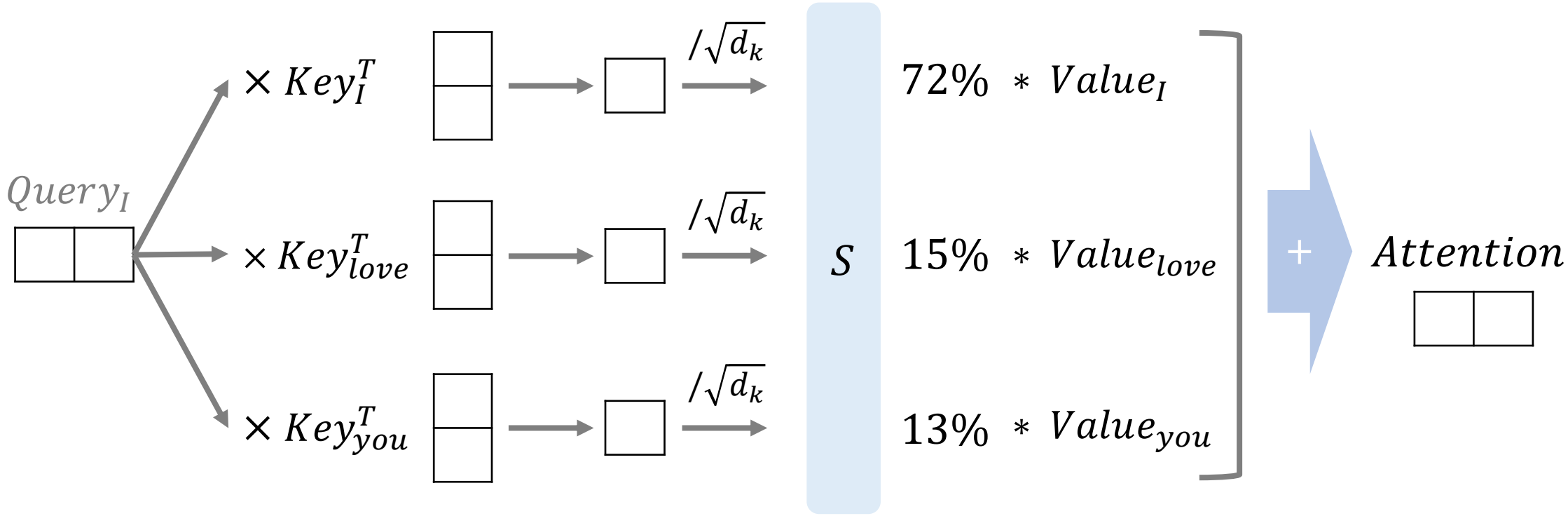
트랜스포머의 동작 원리(하나의 단어): 쿼리(Query), 키(Key), 값(Value)

- 어텐션을 위해 쿼리(Query), 키(Key), 값(Value)이 필요합니다.
- 각 단어의 임베딩(Embedding)을 이용해 생성할 수 있습니다.
  - 임베딩 차원(d_{model}) → $Query$, Key, $Value$ 차원(d_{model} / h)



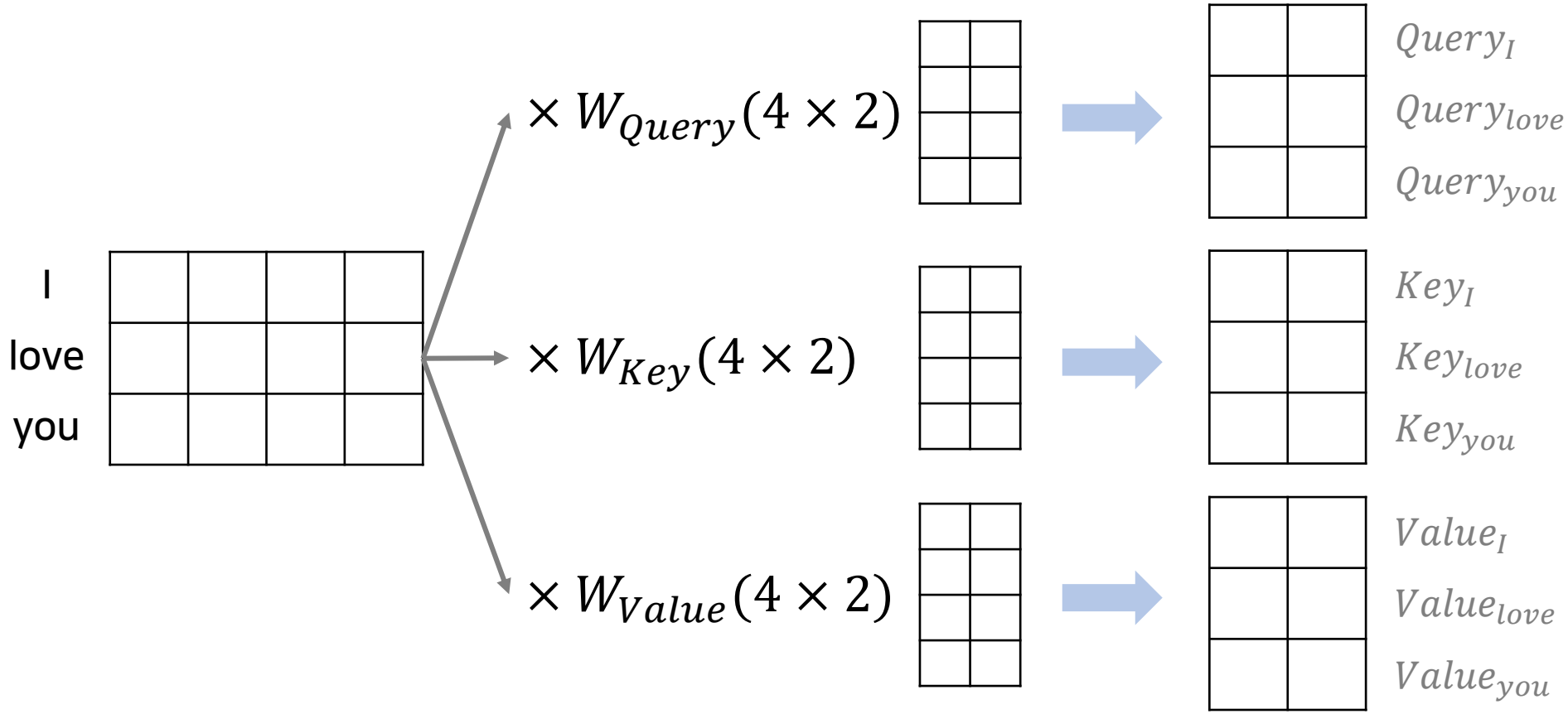
트랜스포머의 동작 원리(하나의 단어): Scaled Dot-Product Attention

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



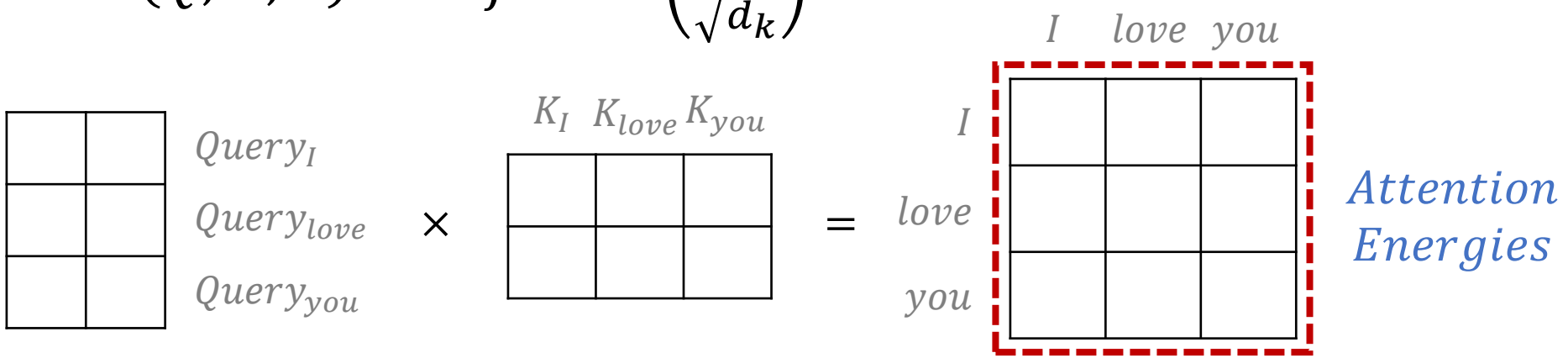
트랜스포머의 동작 원리(행렬): 쿼리(Query), 키(Key), 값(Value)

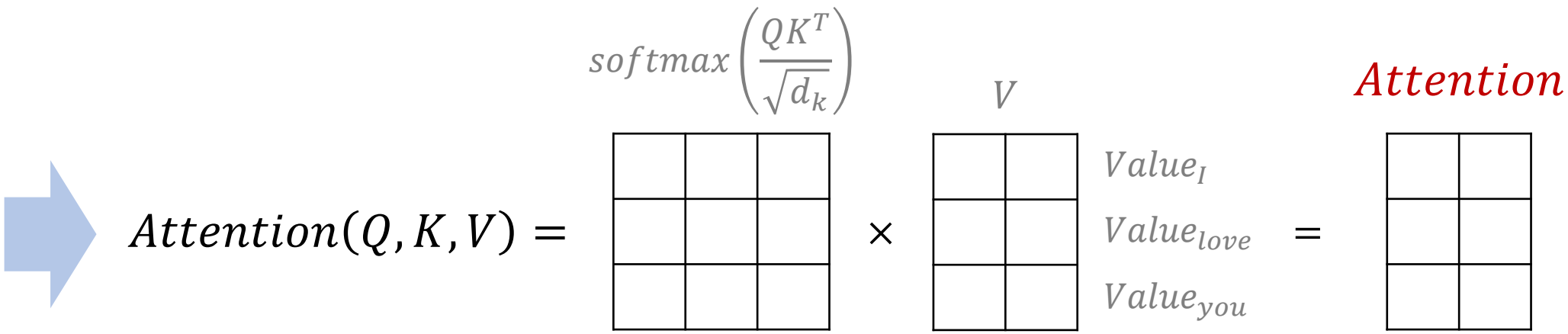
- 실제로는 **행렬(matrix) 곱셈 연산**을 이용해 한꺼번에 연산이 가능합니다.



트랜스포머의 동작 원리(행렬): Scaled Dot-Product Attention

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





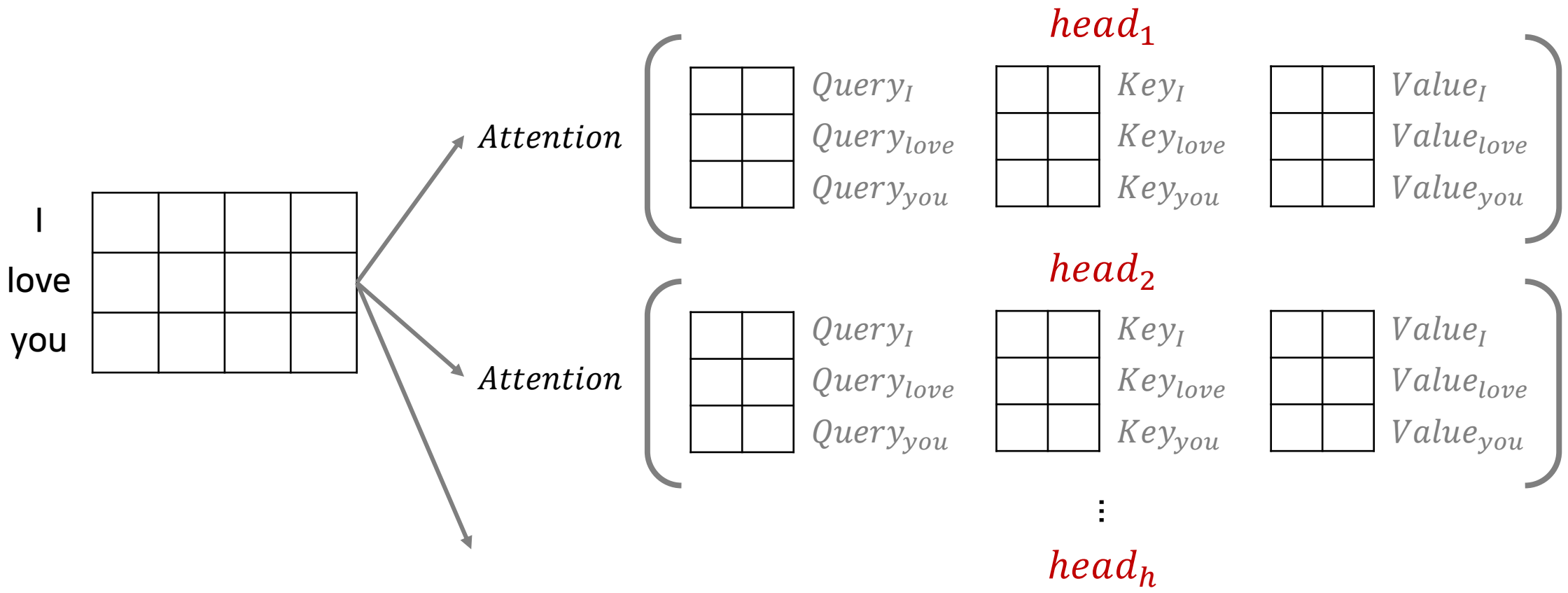
트랜스포머의 동작 원리(행렬): Scaled Dot-Product Attention

- **마스크 행렬(mask matrix)**를 이용해 특정 단어는 무시할 수 있도록 합니다.

- 마스크 값으로 **음수 무한**의 값을 넣어 $softmax$ 함수의 출력이 0%에 가까워지도록 합니다.

트랜스포머의 동작 원리: Multi-Head Attention

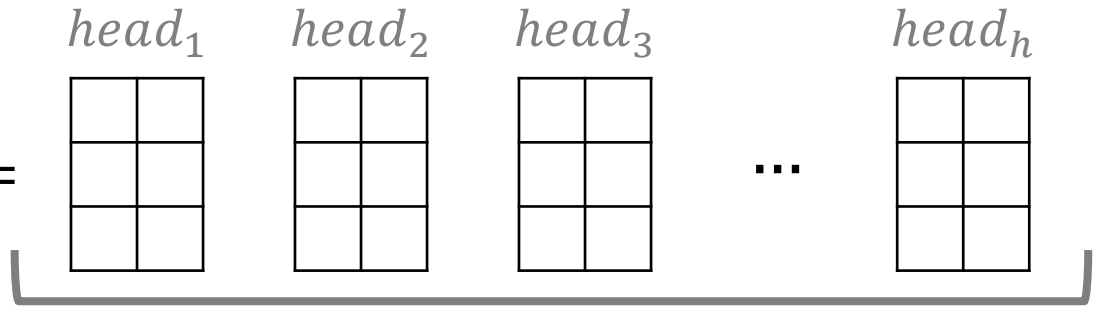
- $MultiHead(Q, K, V) = Concat(head_1, \dots, head_h)W^O$



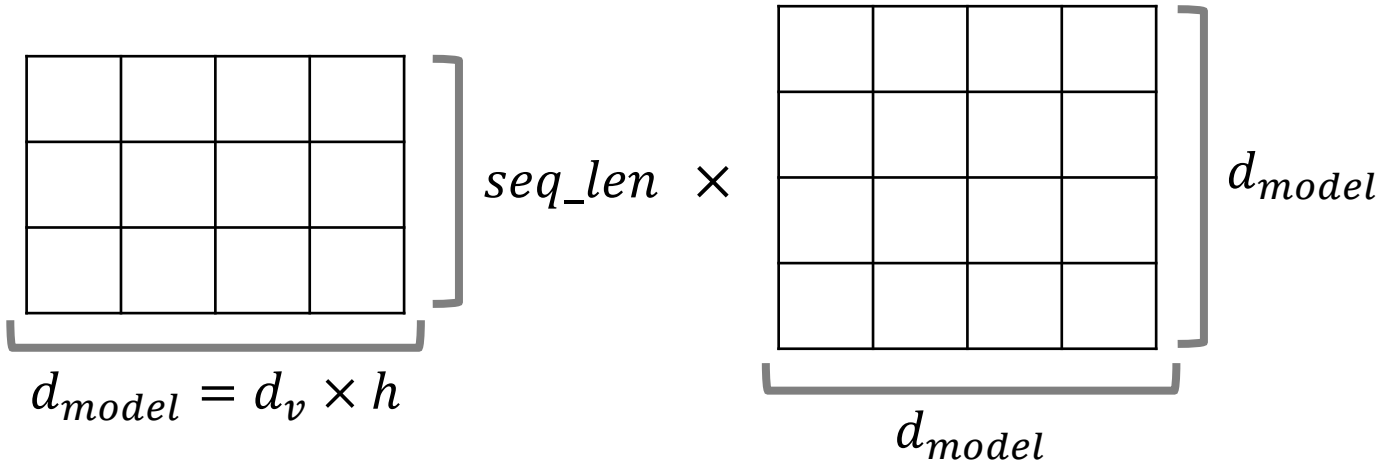
트랜스포머의 동작 원리: Multi-Head Attention

- $MultiHead(Q, K, V)$를 수행한 뒤에도 **차원(dimension)이 동일하게 유지**됩니다.

$$Concat(head_1, \dots, head_h) =$$



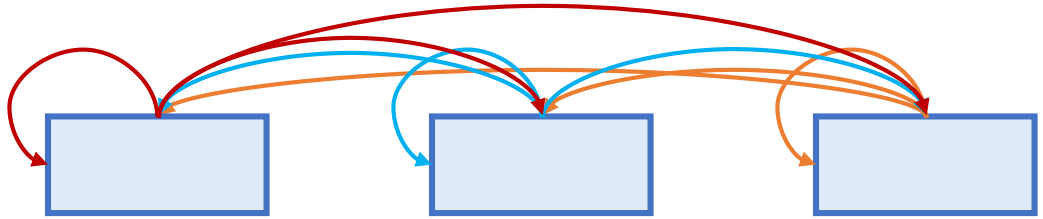
$$MultiHead(Q, K, V) =$$



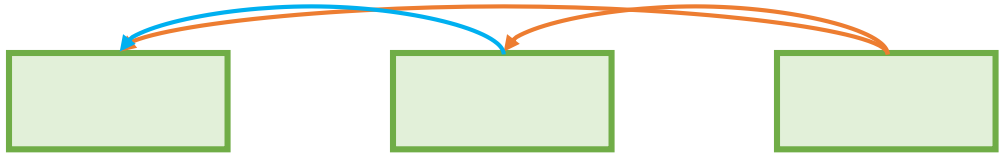
트랜스포머의 동작 원리: 어텐션(Attention)의 종류

- 트랜스포머에서는 세 가지 종류의 **어텐션(attention) 레이어**가 사용됩니다.

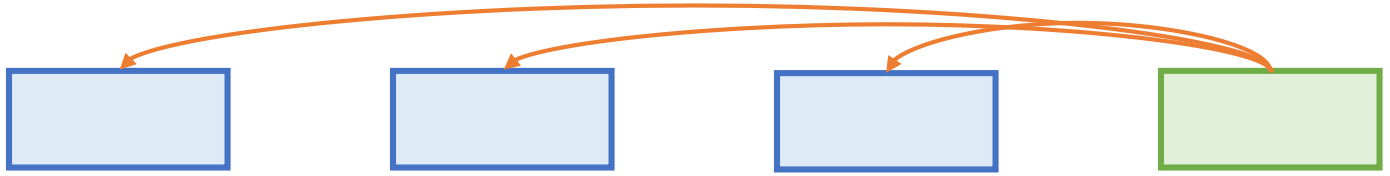
Encoder Self-Attention:



Masked Decoder Self-Attention:

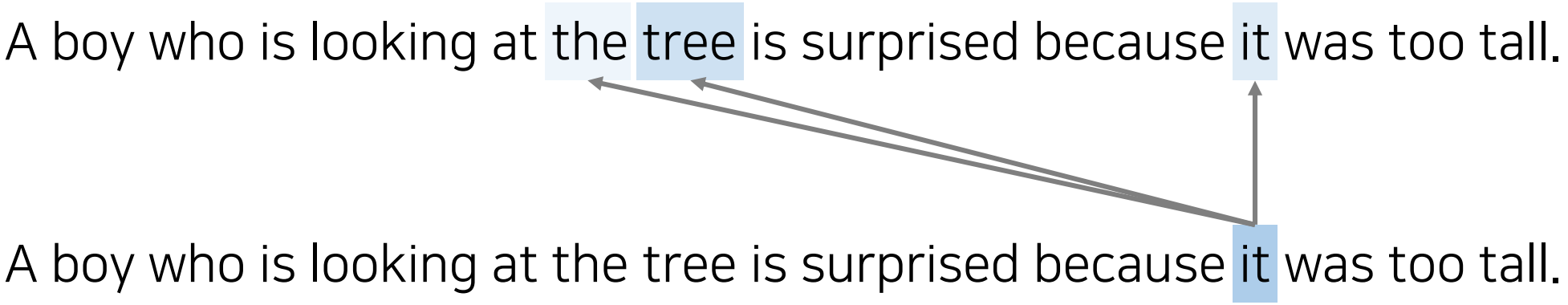


Encoder-Decoder Attention:



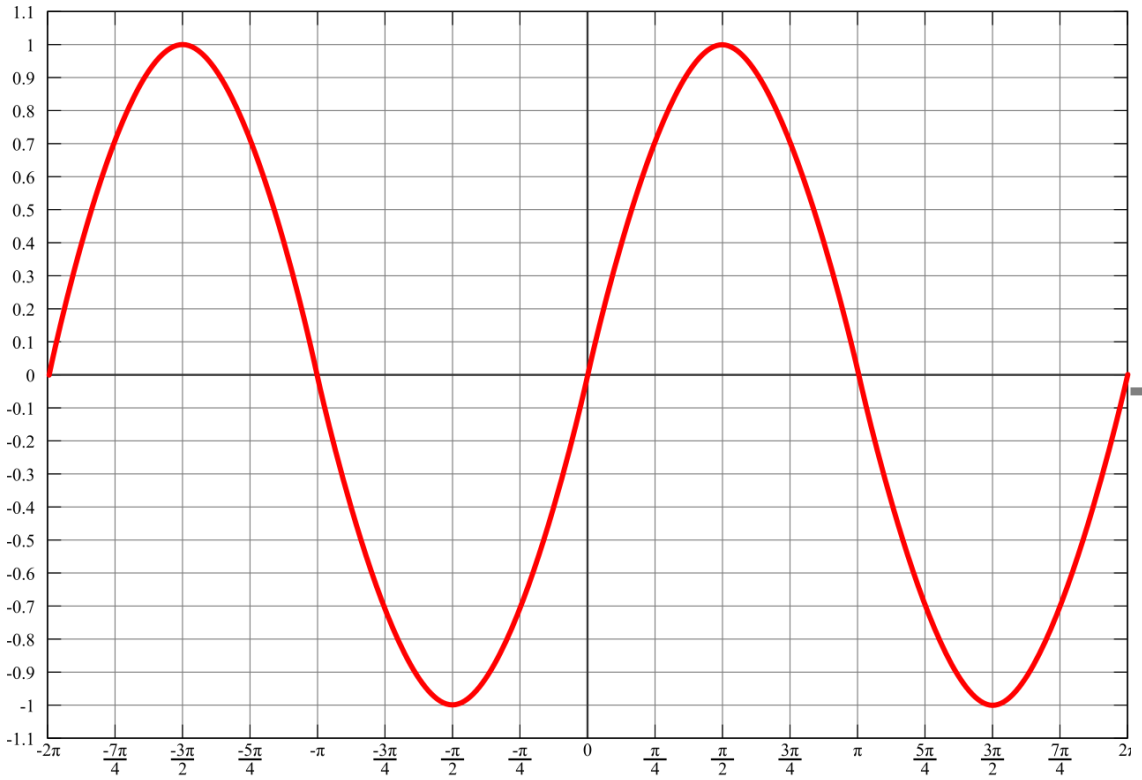
트랜스포머의 동작 원리: Self-Attention

- Self-Attention은 인코더와 디코더 모두에서 사용됩니다.
  - 매번 **입력 문장에서 각 단어가 다른 어떤 단어와 연관성이 높은 지** 계산할 수 있습니다.



트랜스포머의 동작 원리: Positional Encoding

- Positional Encoding은 다음과 같이 주기 함수를 활용한 공식을 사용합니다.
  - 각 단어의 **상대적인 위치 정보를 네트워크에게 입력**합니다.



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

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

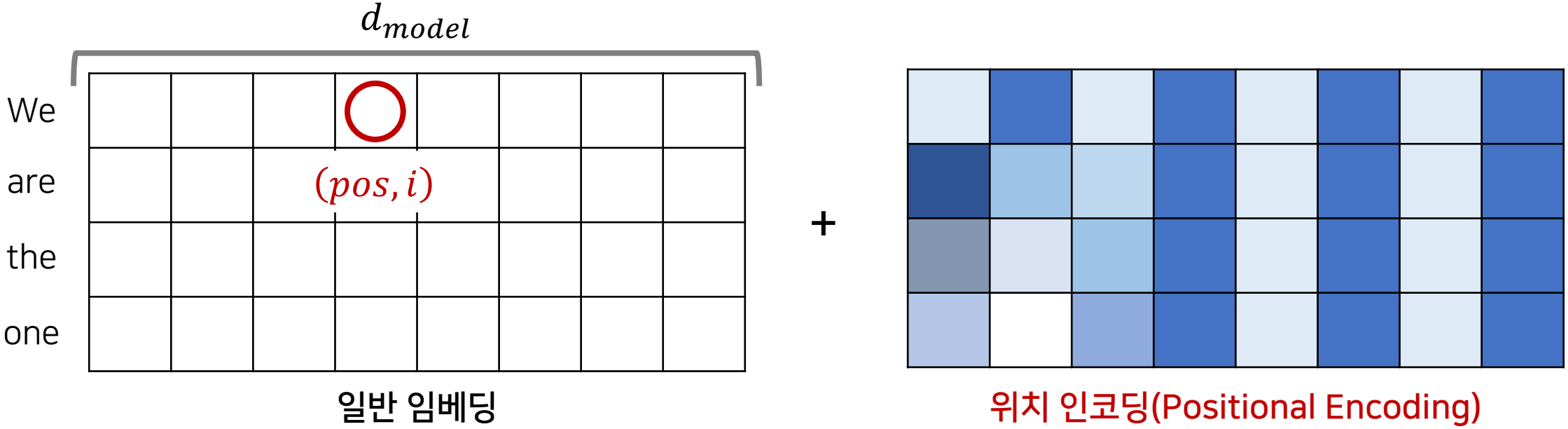
: 사인(sine) 주기 함수 예시

트랜스포머의 동작 원리: Positional Encoding

- Positional Encoding은 다음과 같이 주기 함수를 활용한 공식을 사용합니다.

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

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



트랜스포머의 동작 원리: Positional Encoding

```python
import math
import matplotlib.pyplot as plt

n = 4 # 단어(word)의 개수
dim = 8 # 임베딩(embedding) 차원

def get_angles(pos, i, dim):
    angles = 1 / math.pow(10000, (2 * (i // 2)) / dim)
    return pos * angles

def get_positional_encoding(pos, i, dim):
    if i % 2 == 0: # 짝수인 경우 사인 함수
        return math.sin(get_angles(pos, i, dim))
    # 홀수인 경우 코사인 함수
    return math.cos(get_angles(pos, i, dim))

result = [[0] * dim for _ in range(n)]

for i in range(n):
    for j in range(dim):
        result[i][j] = get_positional_encoding(i, j, dim)
```

출력 결과: `plt.pcolormesh(result, cmap='Blues')`