

Deep Learning Arcitecture

SEQUENCE TO SEQUENCE & ATTENTION & TRANSFORMER

5조

권민지 김재겸 이승우 임채현

Contents

01

SEQUENCE TO SEQUENCE

02

ATTENTION

03

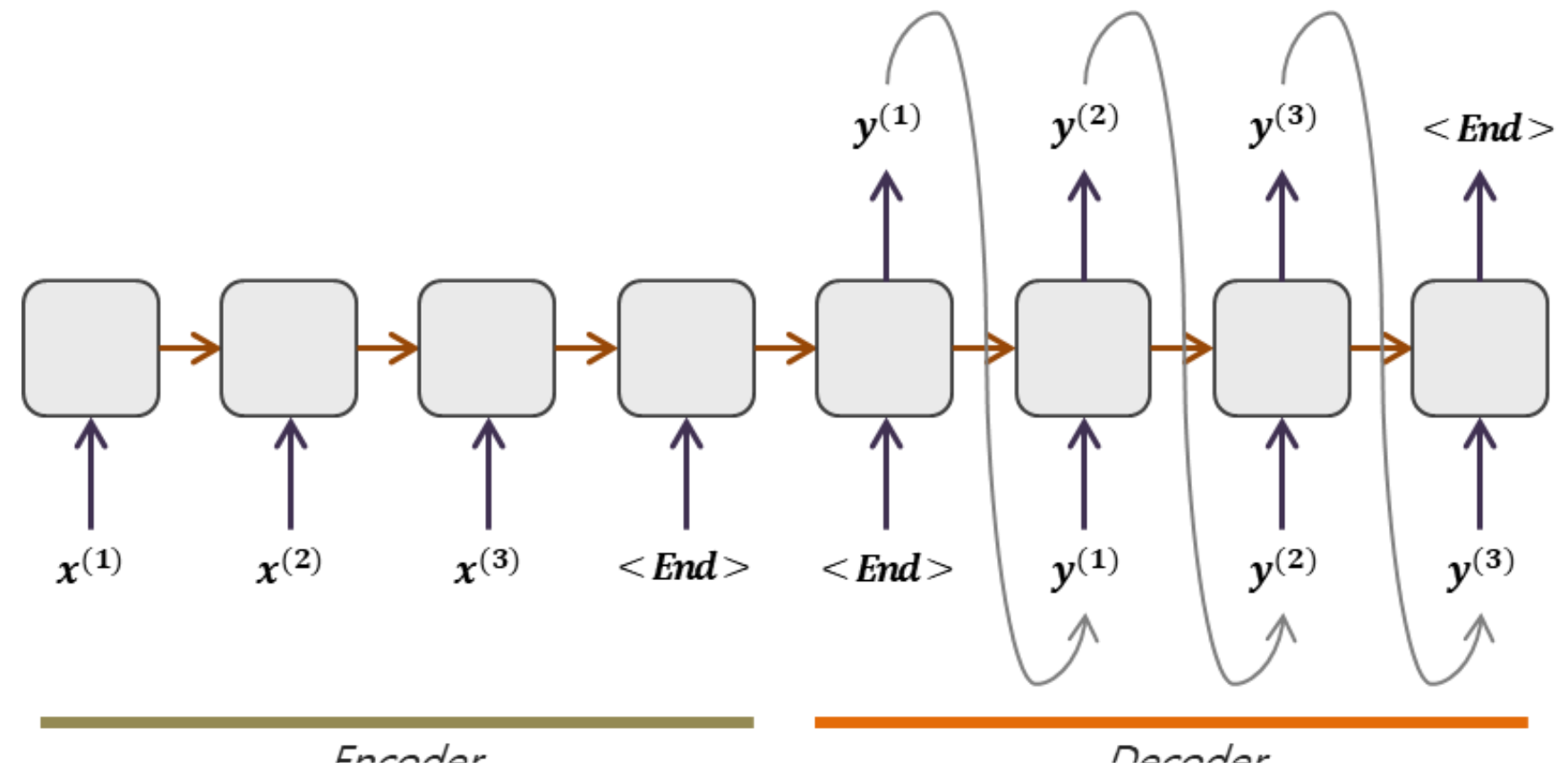
TRANSFORMER

Seq2Seq

Introduction



NIPS(신경정보처리시스템학회)

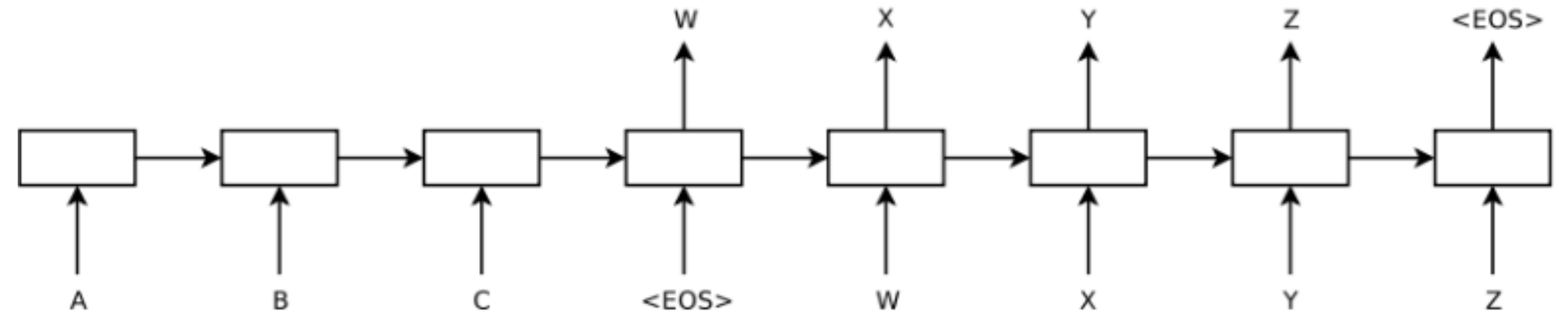


Seq2Seq

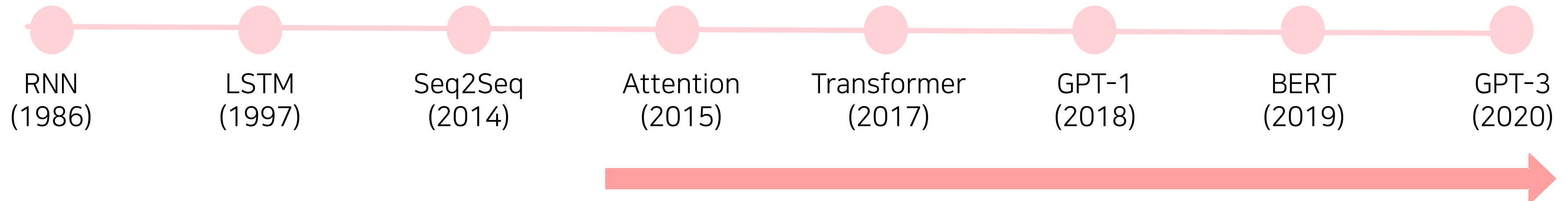
- Machine Translation
- Question Answering
- Text Generation

Seq2Seq

Introduction

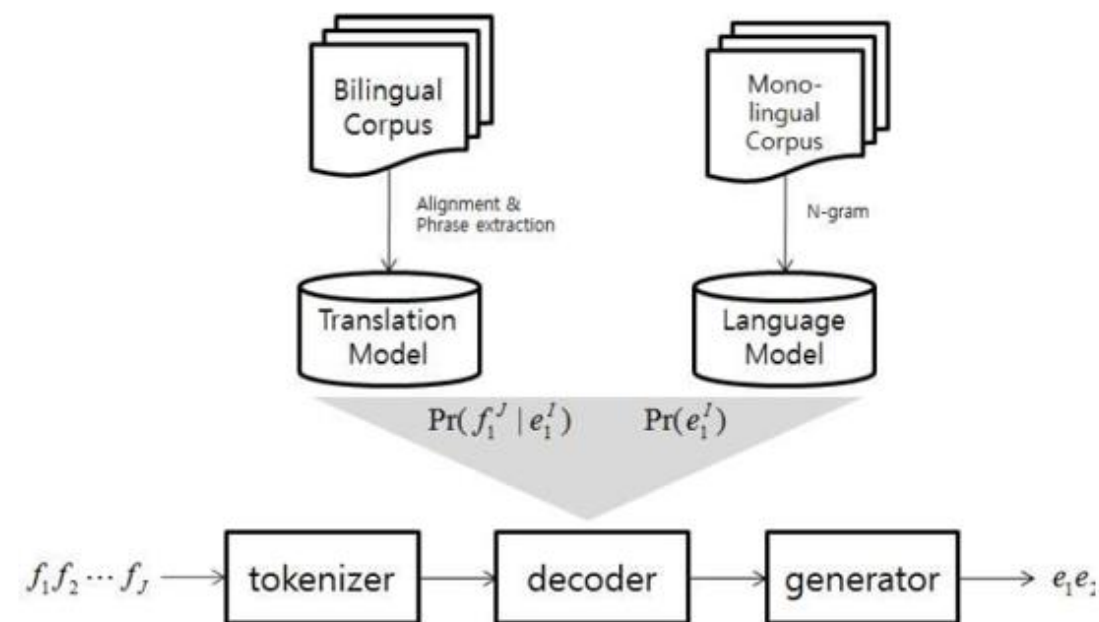


- DNN의 한계점
고정된 Input size로 인한 Sequential problem 해결에 대한 어려움
- Seq2Seq
LSTM 을 통해 Sequential problem을 해결 & 단어를 역순으로 배치하는 방식을 통해 성능 향상



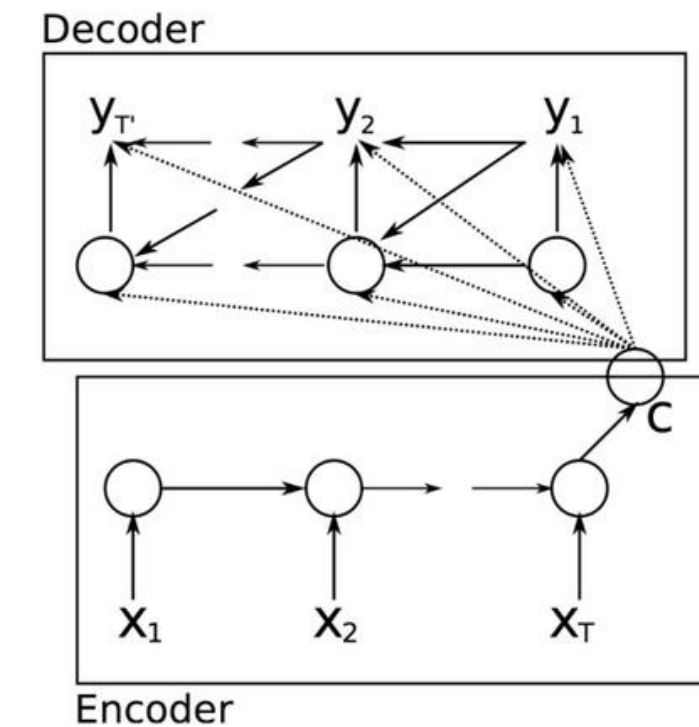
SMT & NMT

Statistical Machine Translation (SMT)



- 대용량 corpus로부터 학습된 통계정보 활용
- 번역모델과 언어모델로 각각 나누어서 번역 수행

Neural Machine Translation (NMT)



- 필요한 데이터: Only parallel corpus

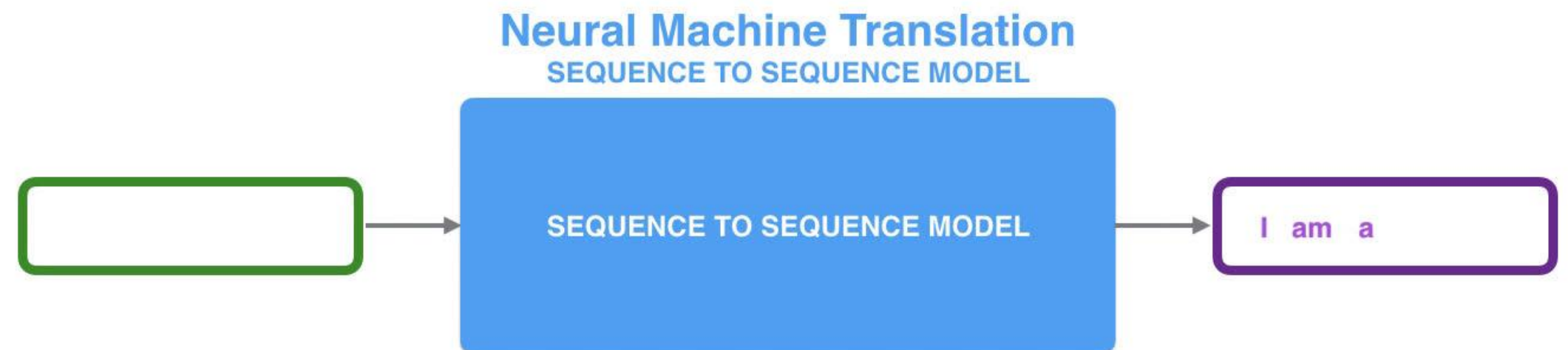
<주요 차이점>

SMT : phrase를 넘어서는 문장관계를 표현할 수 없음

NMT : 문장 전체 정보를 이용할 수 있음

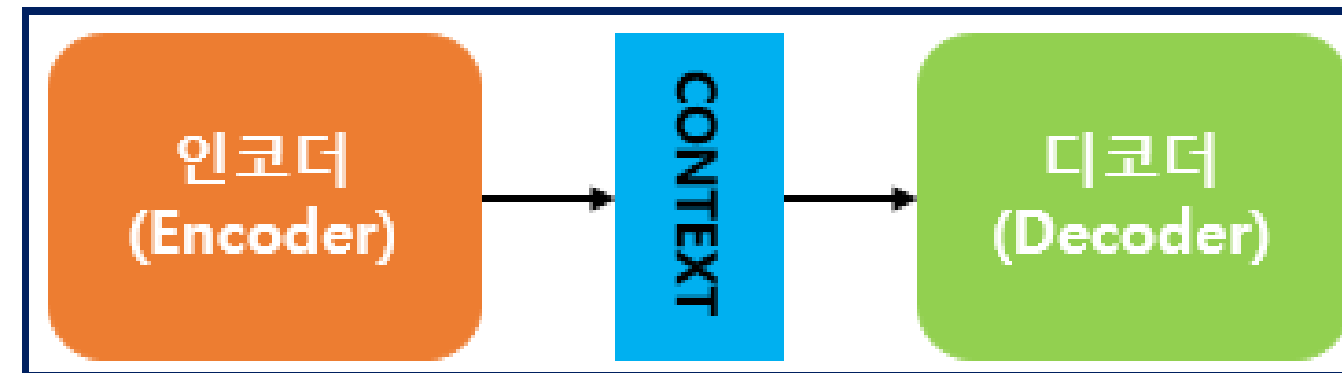
Seq2Seq

Seq2seq (Sequence to sequence)
하나의 시퀀스에서 다른 시퀀스로 출력 및 번역
Ex. 한국어 → 영어, 프랑스어 → 영어



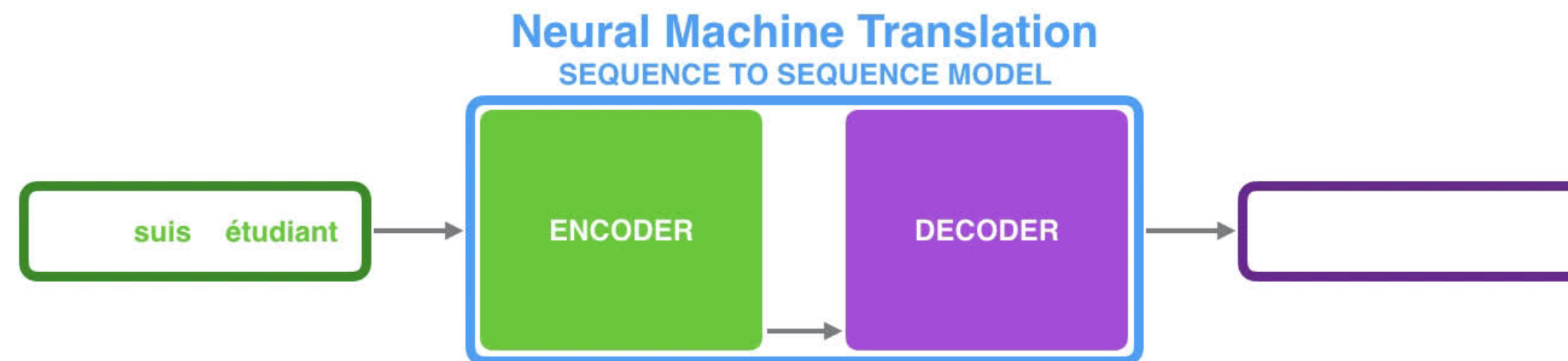
Seq2Seq

Seq2seq의 구성 - Encoder & Decoder



퀴즈 1번

Encoder: 입력 문장의 모든 단어들을 입력 받은 후 마지막에 모든 단어의 정보를 압축해서 하나의 벡터 생성
Decoder : context vector를 받아서 번역 대상 나라의 언어로 단어를 한 개씩 순차적으로 출력



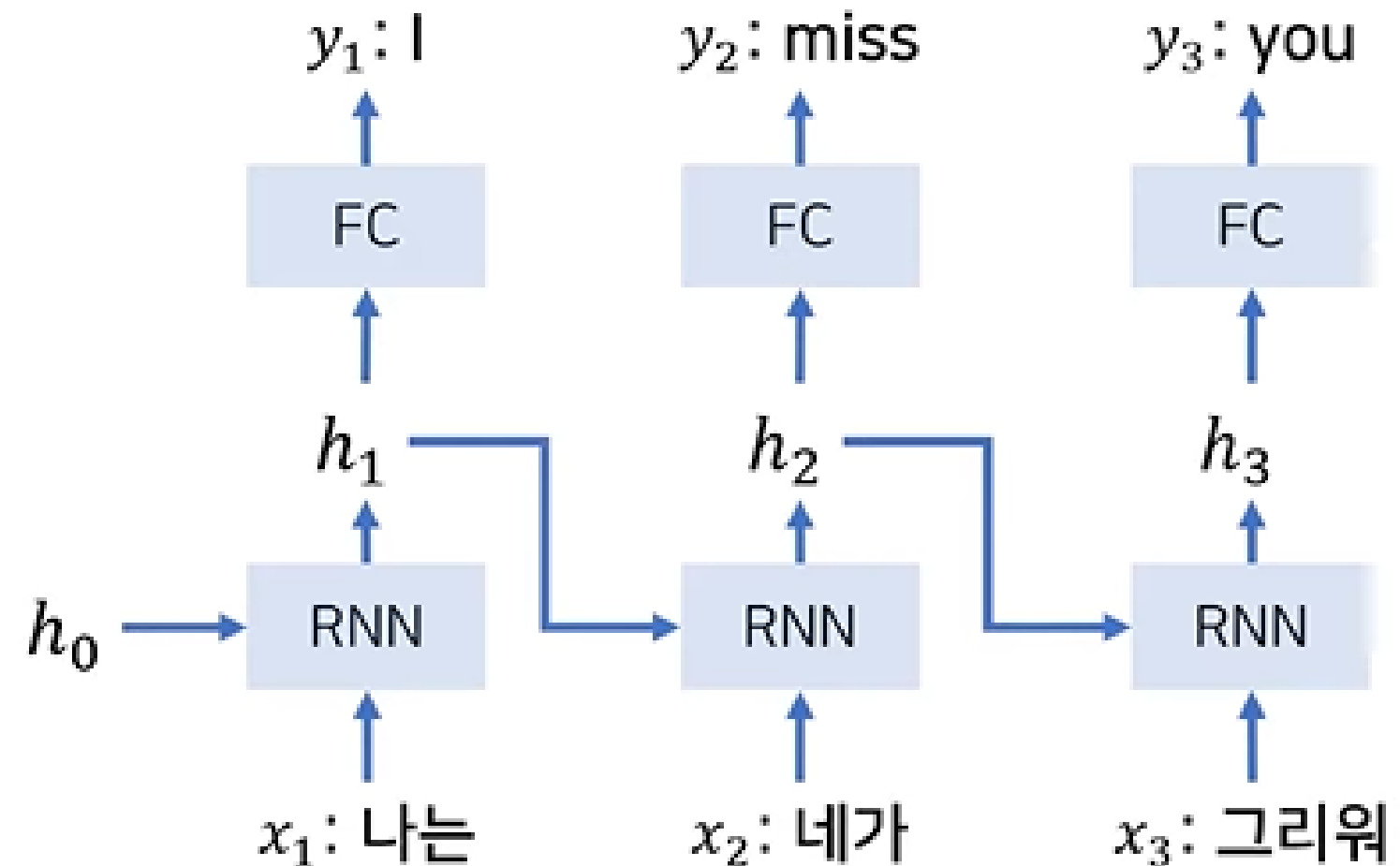
RNN

<전통적인 초창기 RNN 기반의 언어 모델에서 번역이 이루어진 과정>

입력: (x_1, \dots, x_T)
출력: (y_1, \dots, y_T)

단어의 개수

- $h_t = \text{sigmoid}(W^{hx}x_t + W^{hh}h_{t-1})$
- $y_t = W^{yh}h_t$



입력과 출력의 크기가 같다고 가정

hidden State : 이전까지 입력이 되었던 데이터에 대한 전반적인 정보

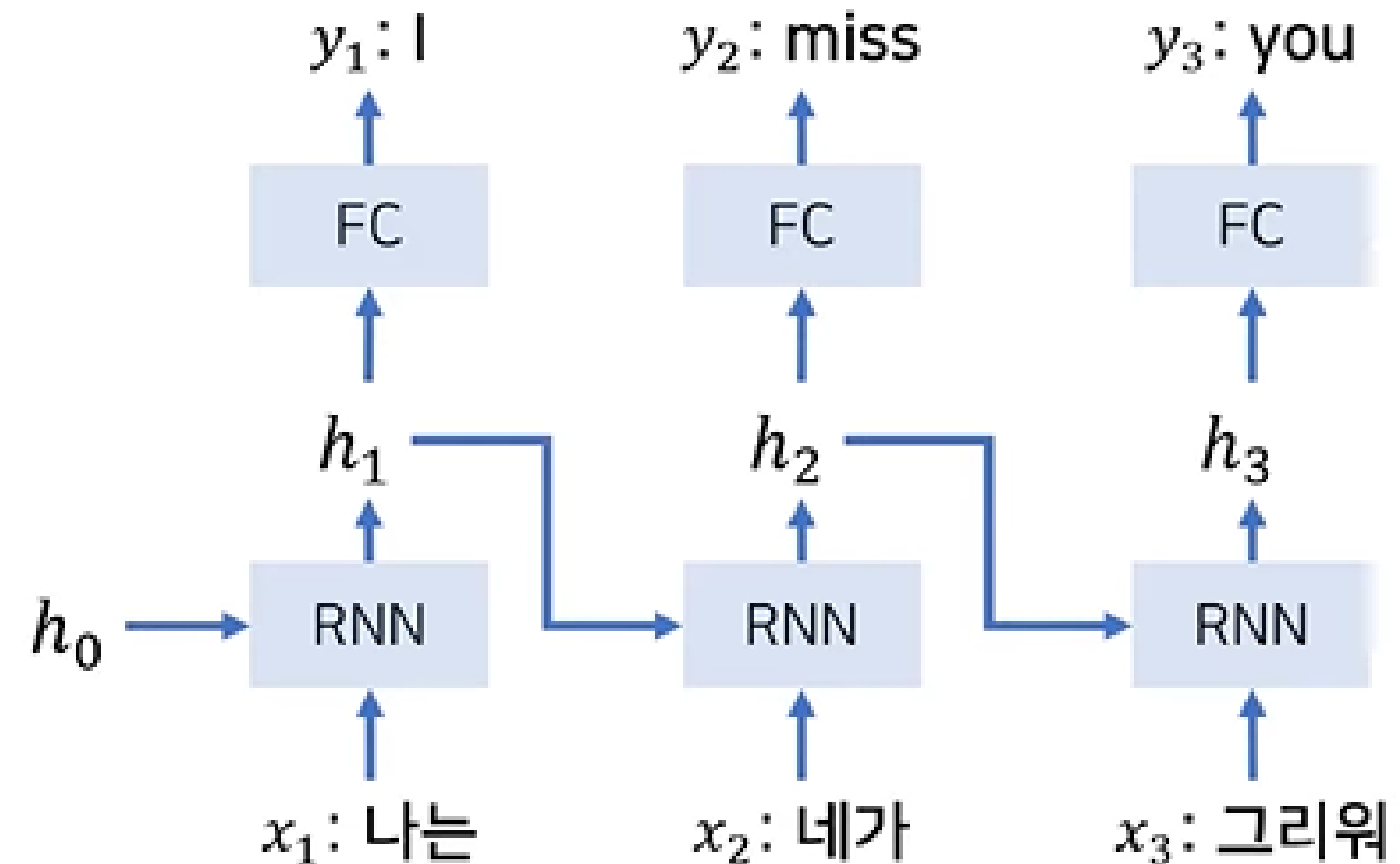
RNN

<전통적인 초창기 RNN 기반의 언어 모델에서 번역이 이루어진 과정>

입력: (x_1, \dots, x_T)
출력: (y_1, \dots, y_T)

단어의 개수

- $h_t = \text{sigmoid}(W^{hx}x_t + W^{hh}h_{t-1})$
- $y_t = W^{yh}h_t$



입력과 출력의 크기가 같다고 가정

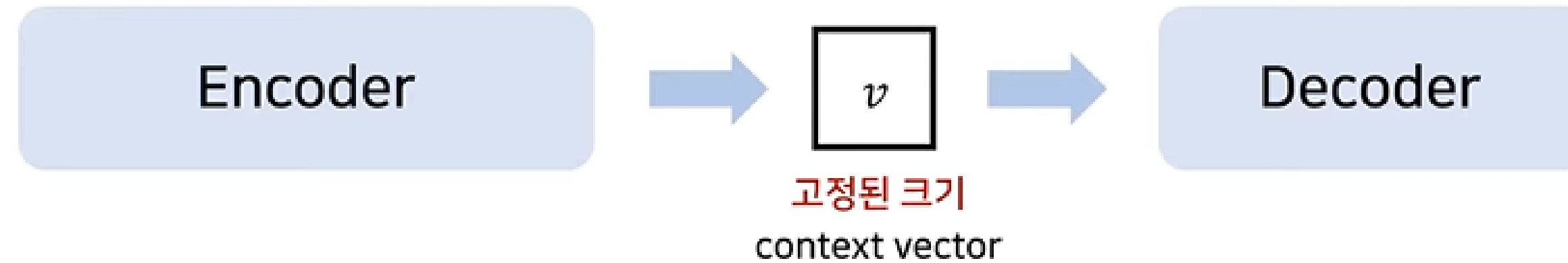
hidden State : 이전까지 입력이 되었던 데이터에 대한 전반적인 정보



정확한 결과 유추의 어려움

장기 의존성 문제

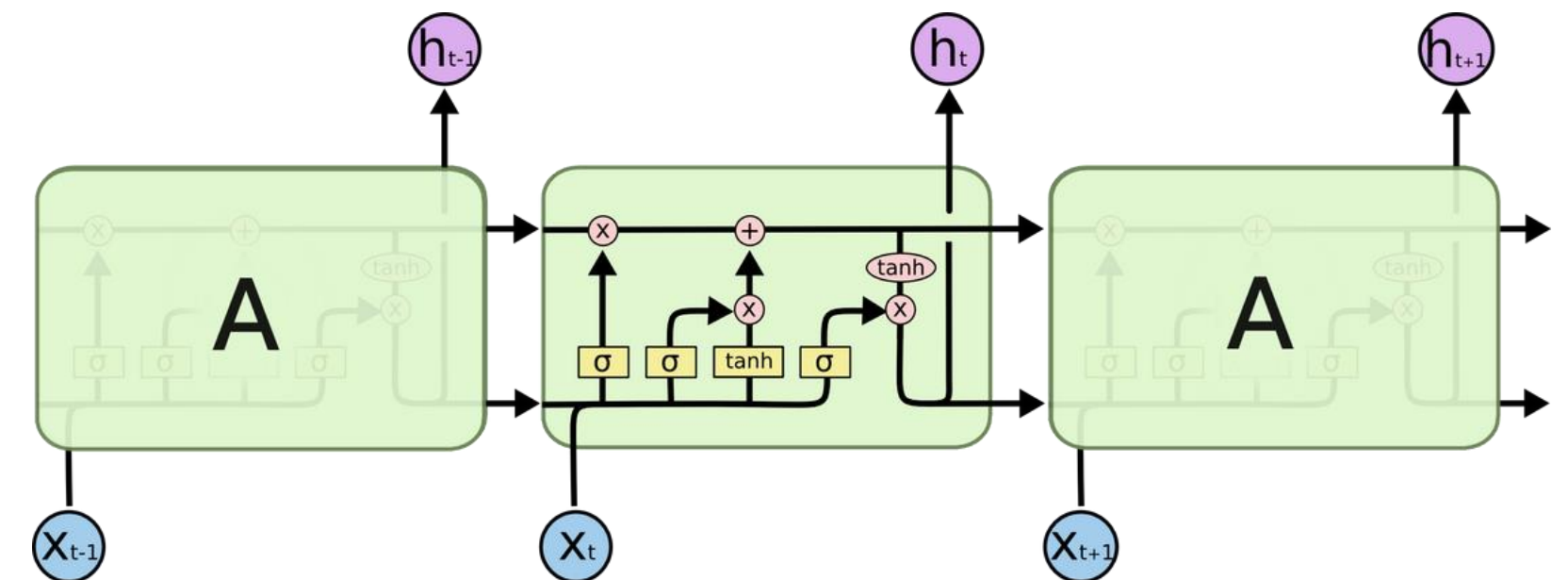
LSTM



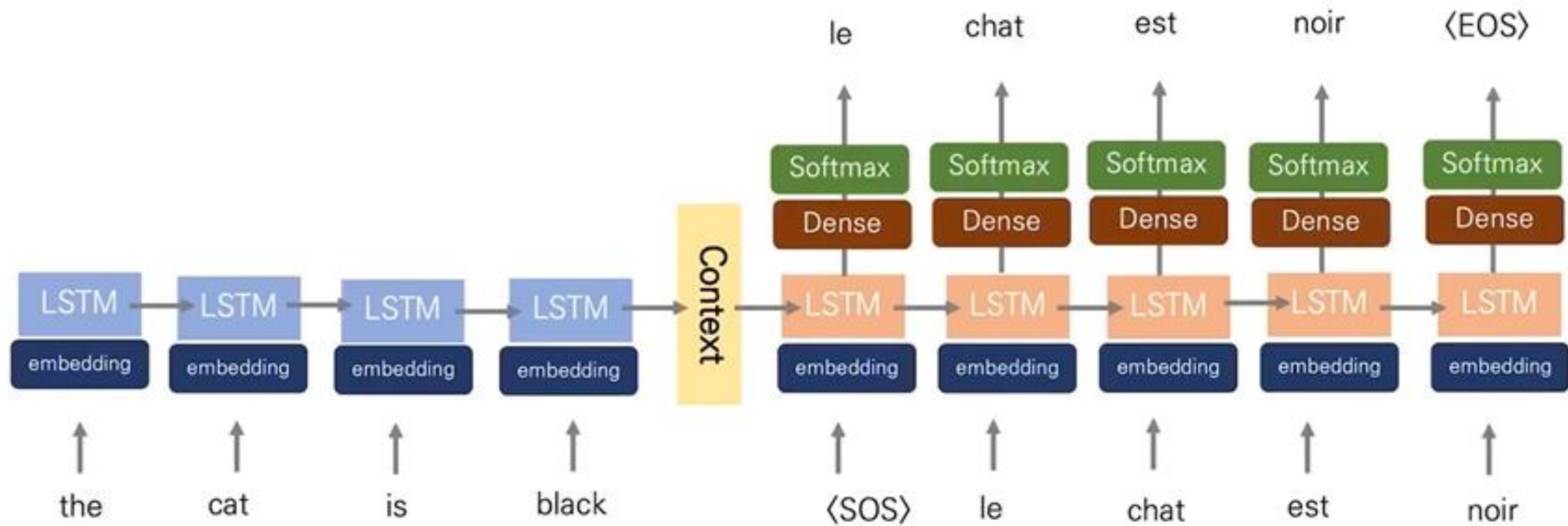
RNN의 한계점 해결

- Encoder - 고정된 크기에 context vector 추출
- context vector로부터 Decoder - 번역 결과 추론
- LSTM를 이용해 context vector를 추출하도록 하여 성능 향상
 - 인코더의 마지막 hidden state만을 context vector로 사용

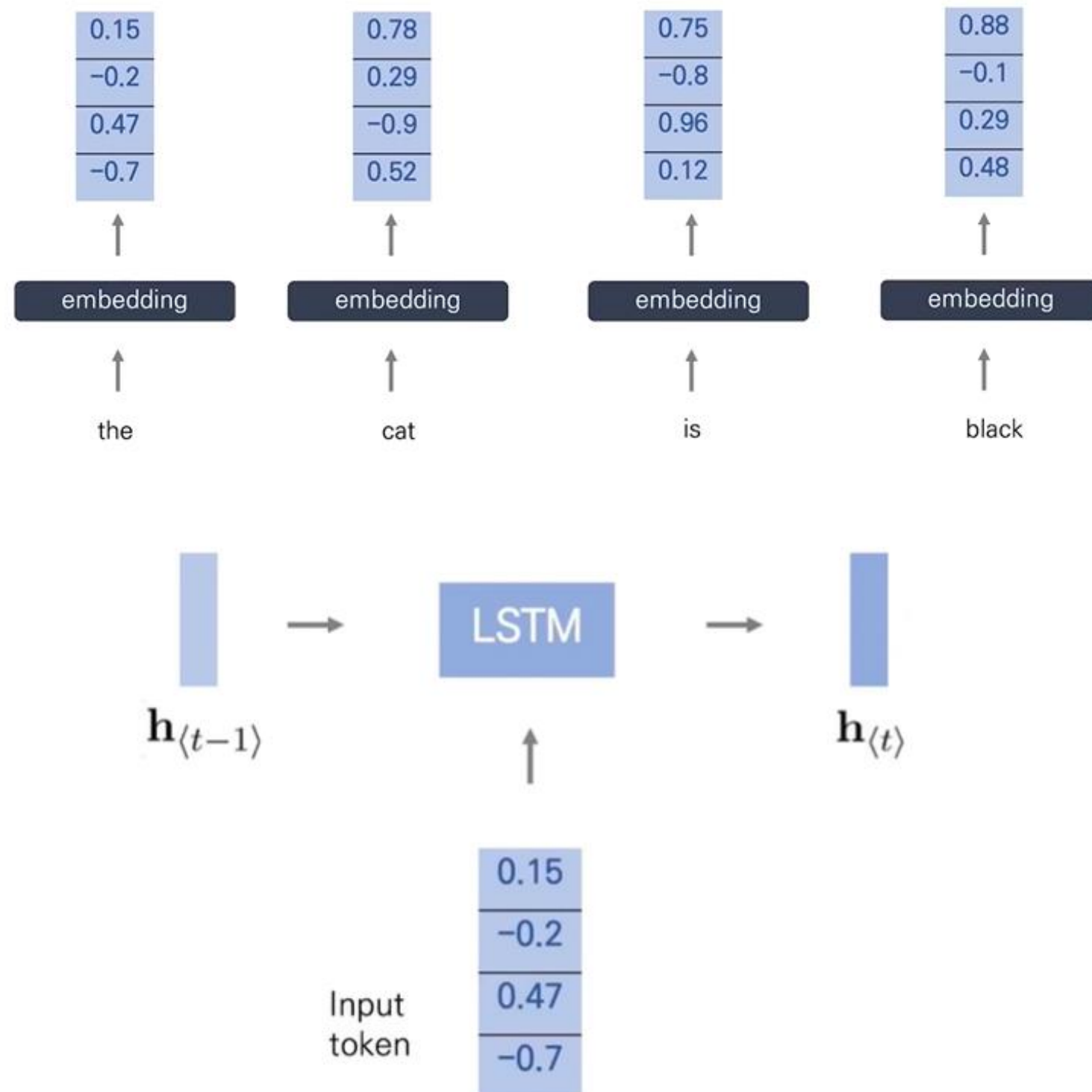
LSTM
긴 의존 기간을 필요로 하는 학습을 수행
할 능력을 가지고 있음



Seq2Seq Overview



Seq2Seq Overview



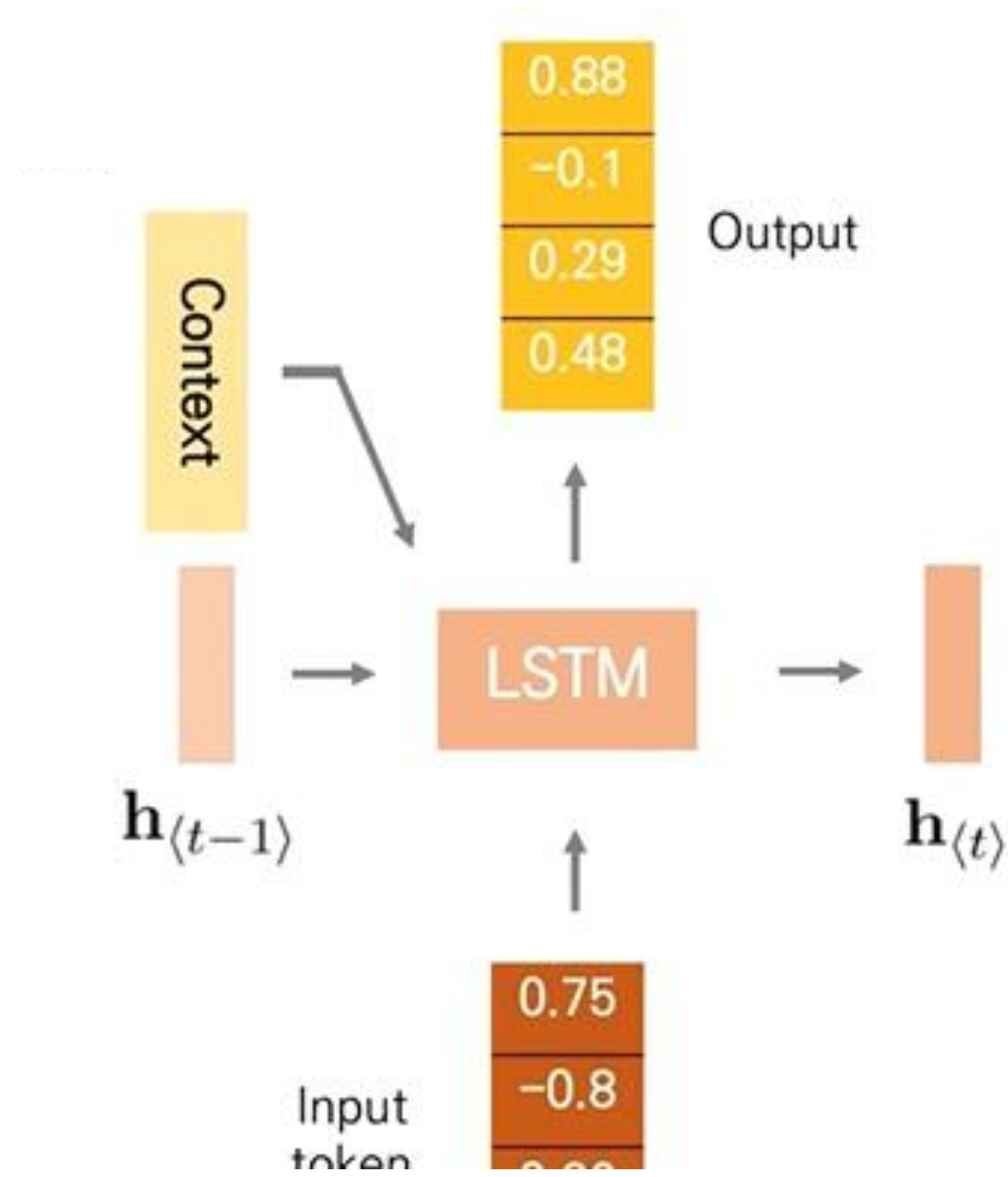
① Embedding

- Source sentence, Target sentence 각각을 임베딩
→ 인코더와 디코더의 input으로 활용

② Encoder

- 임베딩된 단어들 순차적으로 Encoder의 입력으로 활용
- 임베딩 단어와 hidden state를 LSTM cell로 연산
→ hidden state 업데이트
- 마지막 hidden state → context vector로 지정

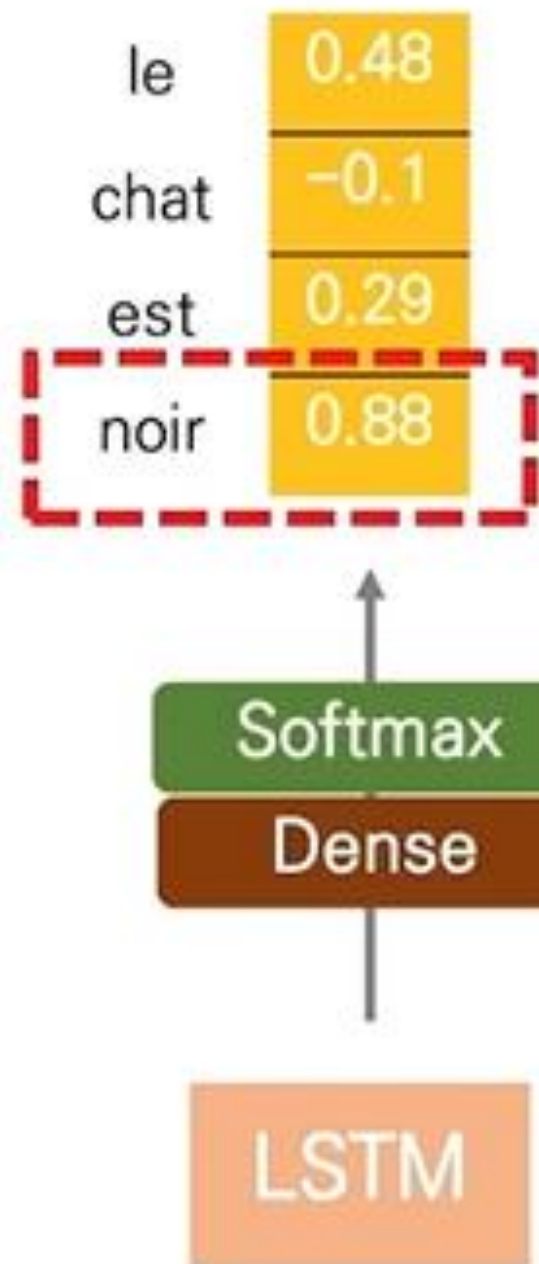
Seq2Seq Overview



③ Decoder

- 임베딩된 target 문장 단어 → 순차적으로 Decoder의 입력으로 활용
- 임베딩 단어와 hidden state를 LSTM cell로 연산
→ hidden state를 업데이트 → 다음에 나올 확률이 높은 단어 예측
- Target sentence의 임베딩은 학습 시에만 필요

Seq2Seq Overview

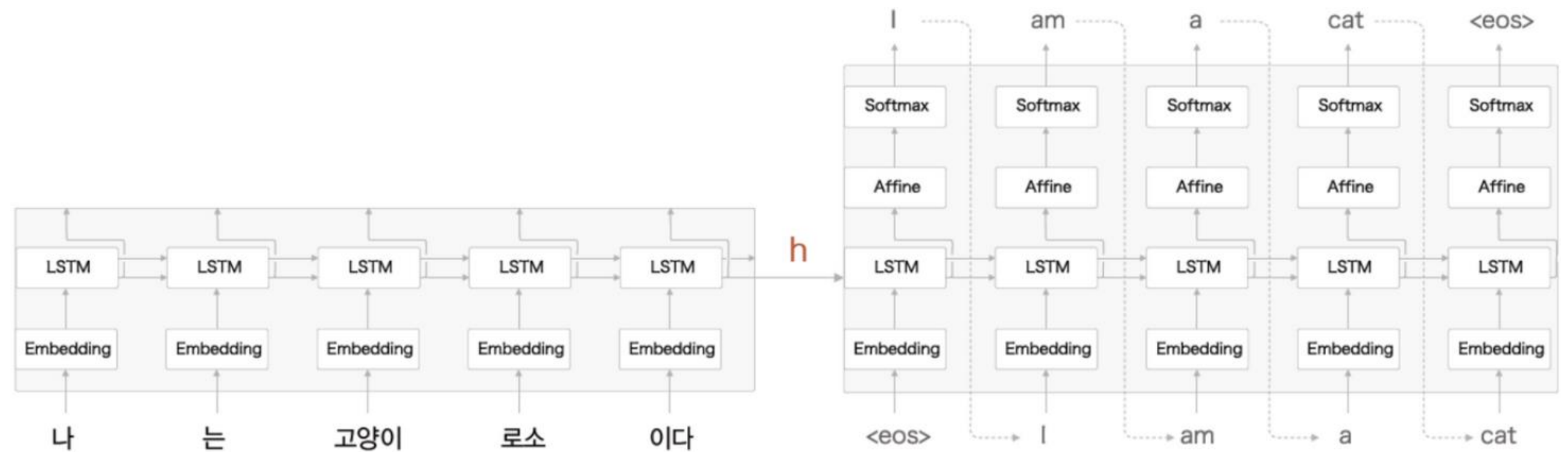
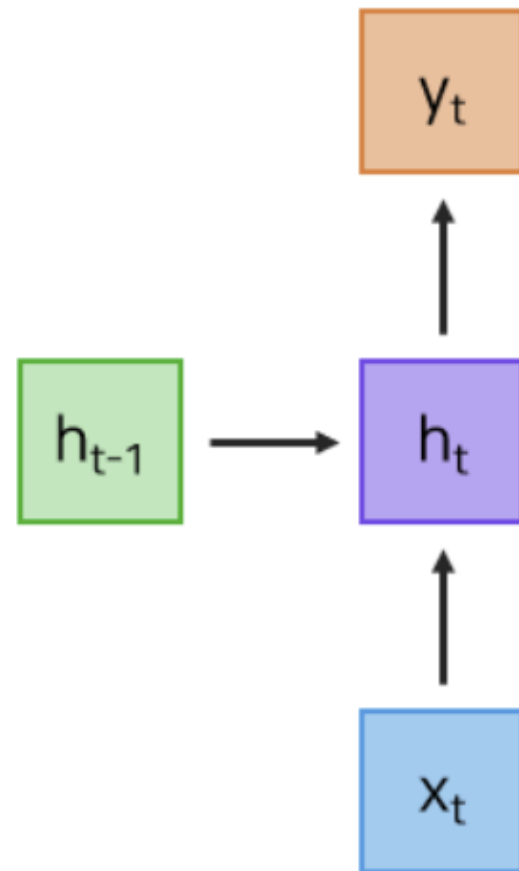


④ Softmax Layer

- LSTM의 output을 dense layer에 거침
- Softmax를 활용해 각 단어가 나올 확률 값을 계산
- Softmax layer를 거치며 각 토큰이 나올 확률 값이 output으로 나오게 됨
- 이 예시에는 noir가 나올 확률이 가장 높으므로, 다음 단어는 noir로 선택

Attention 등장 배경

Seq2Seq의 문제점



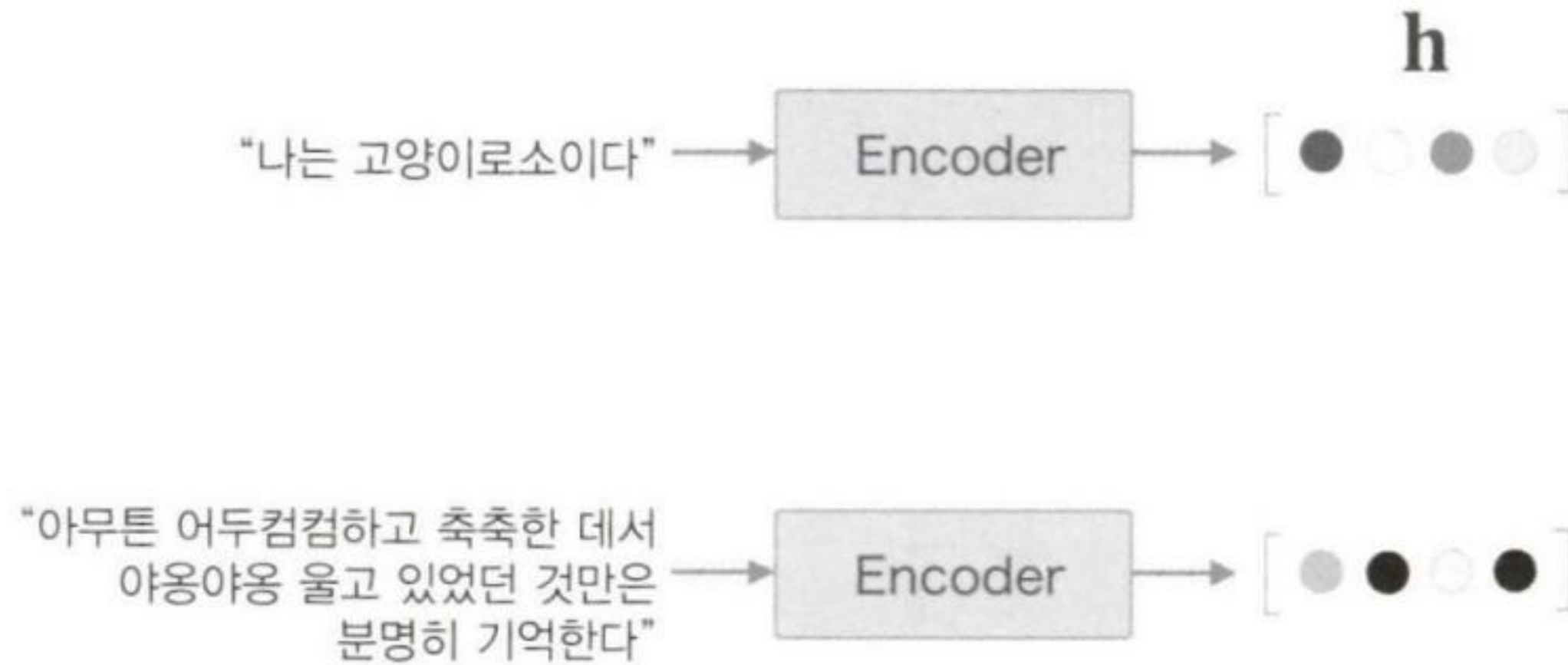
1. 병렬 처리의 어려움

→ 복잡한 계산 시 효율적으로 처리하지 못해 성능 저하 발생

2. 기울기 소실 문제 발생

3. 입력 문장의 길이에 상관없이 항상 고정 길이의 벡터를 출력

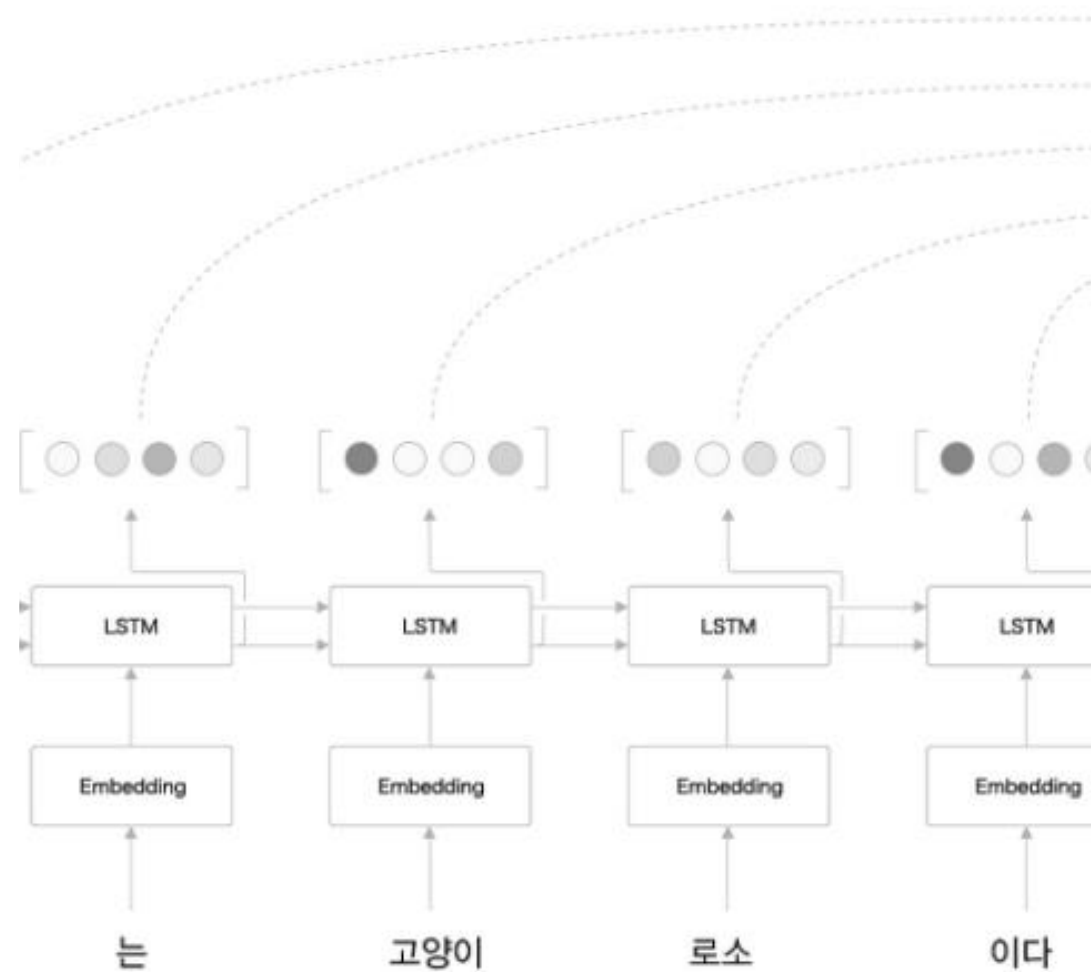
Attention 등장 배경



seq2seq의 Encoder : 마지막 hidden state만 디코더에 전달
→ 입력 문장의 길이에 상관없이 항상 고정된 길이의 벡터를 출력
→ 모든 입력 문장의 정보 활용 X

Attention : 입력 문장의 길이에 맞게 인코더의 출력 벡터 길이를 변화시키자!

Attention Encoder



Attention Encoder

LSTM에서 나오는 출력 결과값
→ 하나의 행렬(hs)로 묶음

각 시점에서 나오는 출력 벡터
→ 해당 시점 입력 단어의 정보 + 이전 시점에서의 정보

모든 벡터를 활용하는 방법
→ 각 시점의 출력을 하나의 행렬로 묶어서
Decoder의 모든 시점에 전달

Attention Decoder

Attention Decoder

본인 시점에 맞는 벡터만 선택해서 활용

→ 행렬 hs 안에 있는 각 벡터의 중요도를 계산

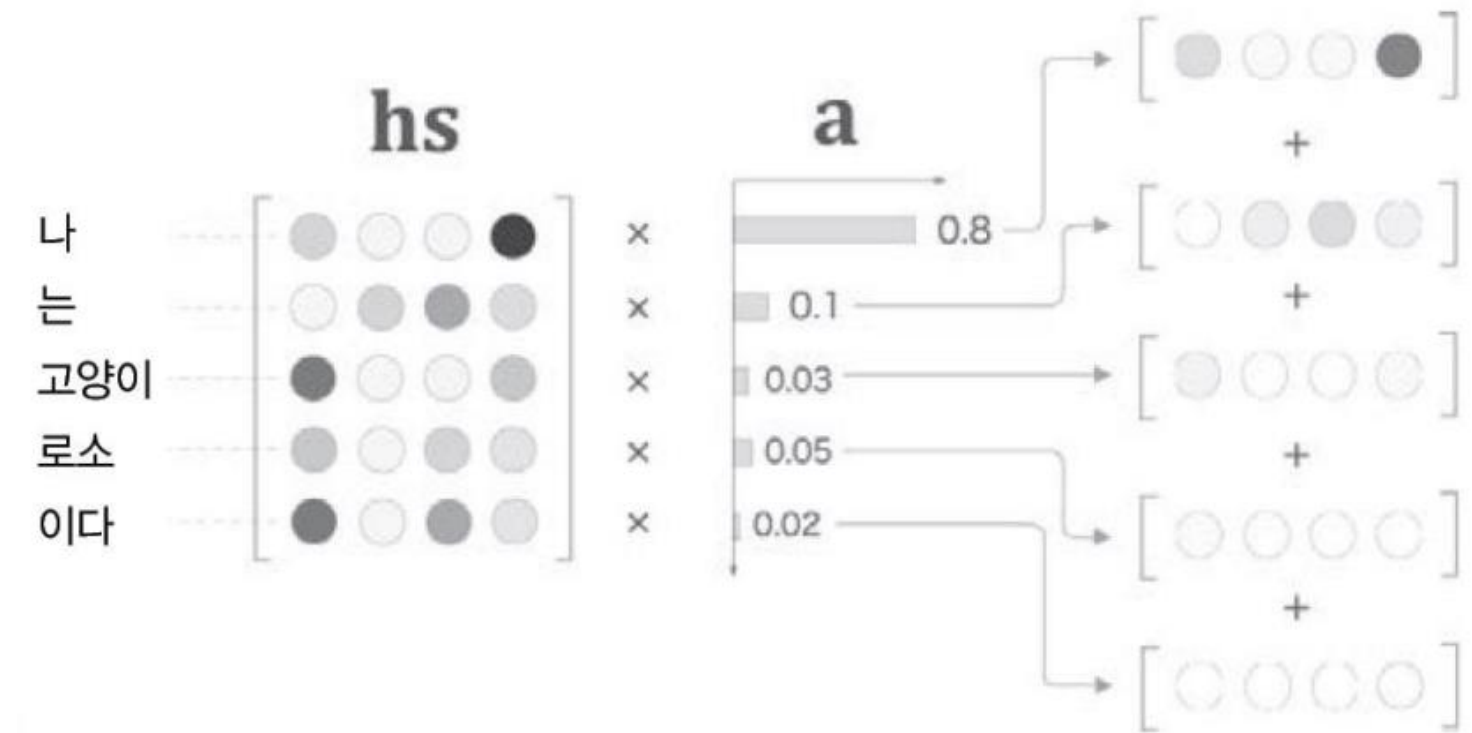
→ 가중치의 합 구하기

벡터의 중요도란?

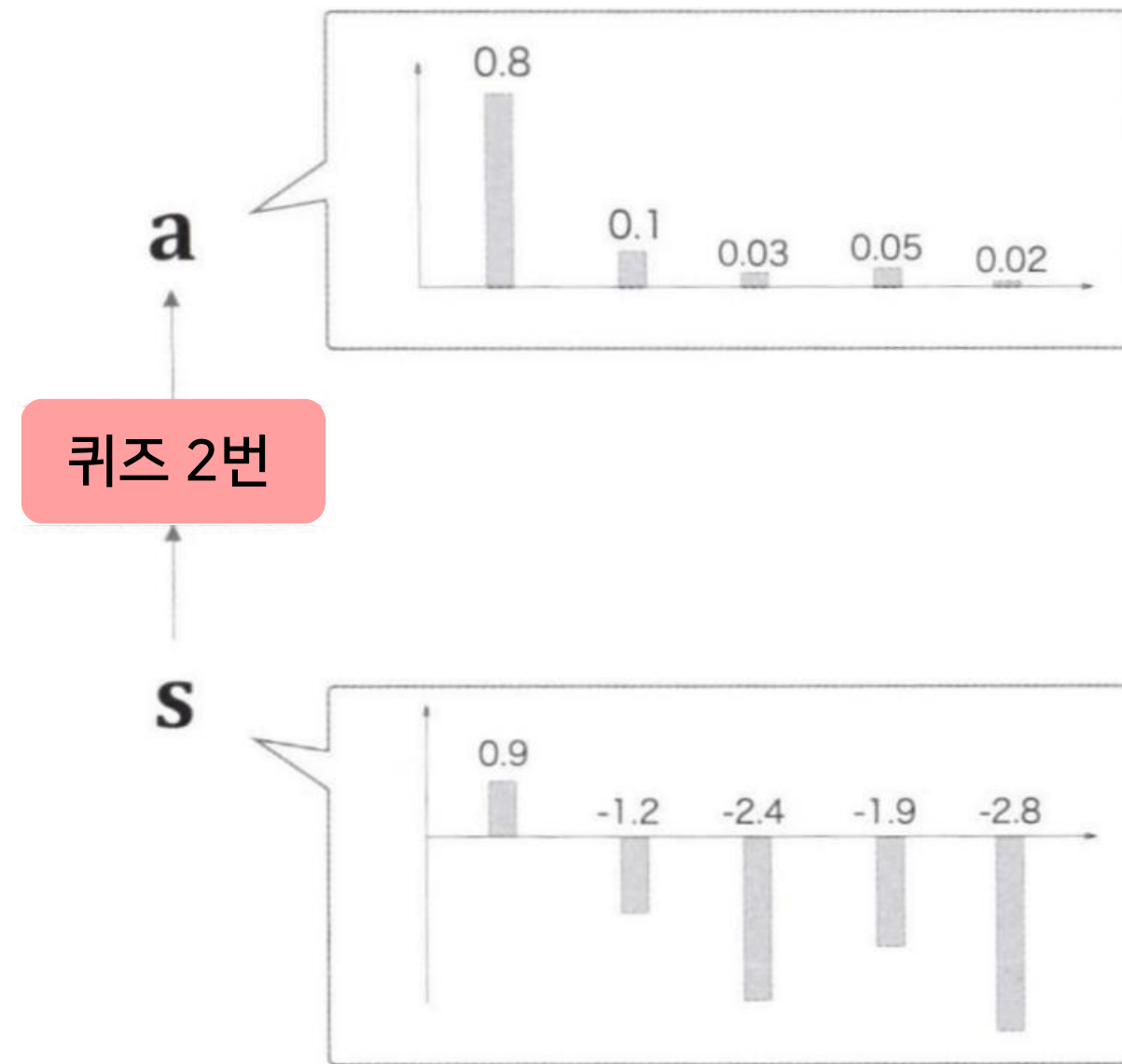
→ Decoder의 각 시점에서 hidden state 벡터와
행렬 hs 안에 있는 각 벡터가 얼마나 유사한지에 대한 것

→ hs 행렬과 h 벡터를 내적

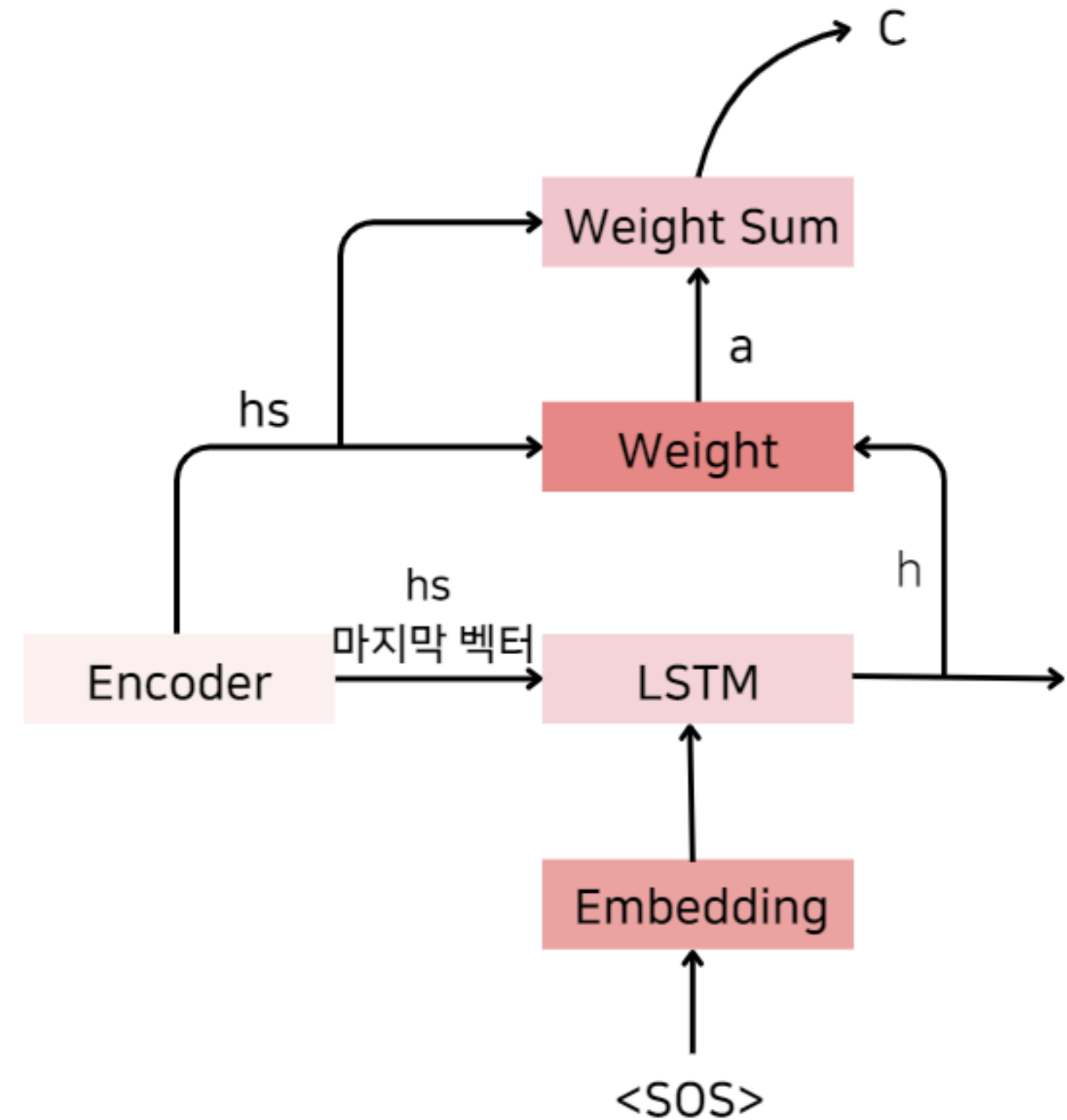
→ 내적인 값이 클 수록 두 벡터의 방향이 유사하다는 것



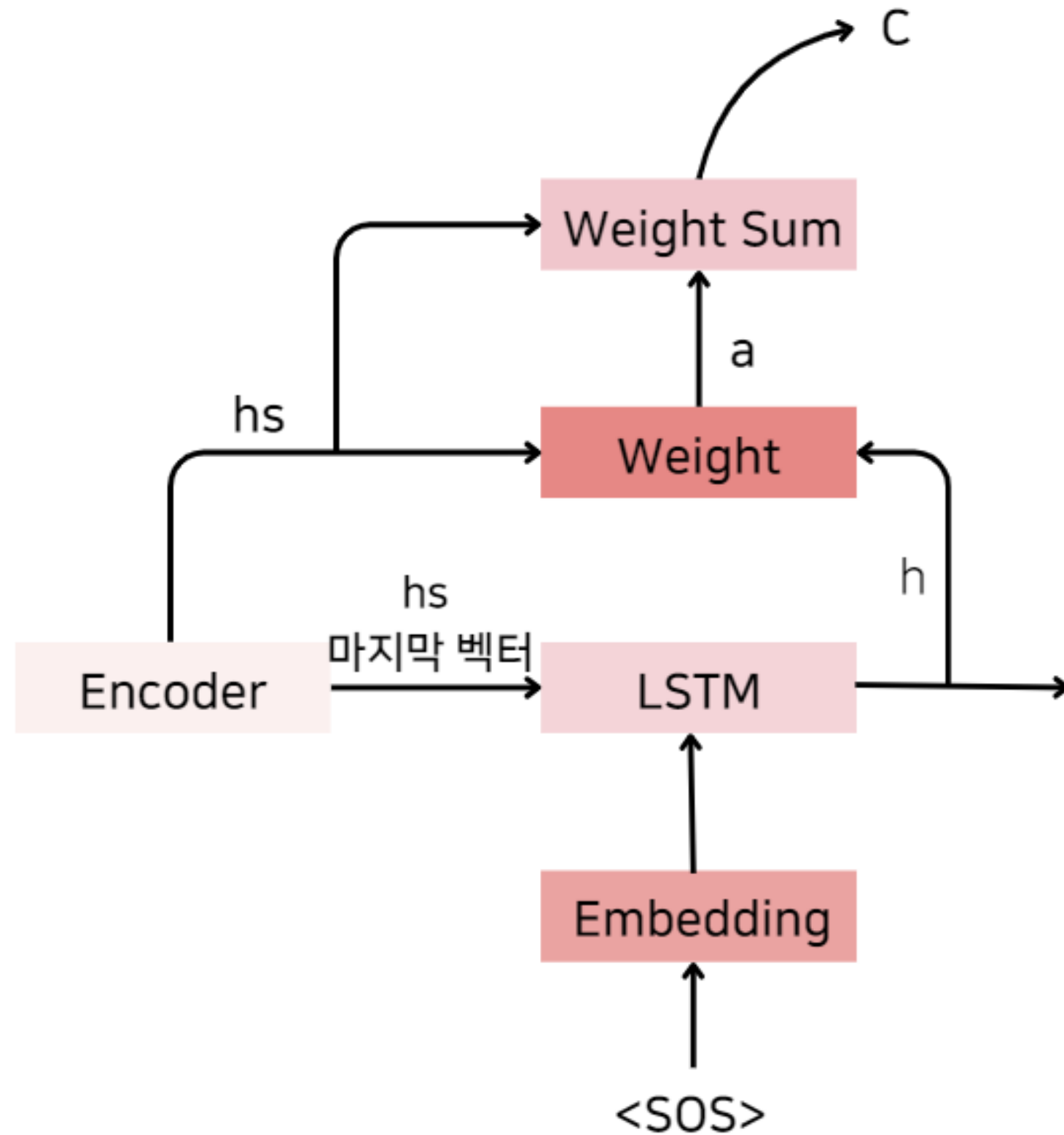
Attention Decoder



- 내적값 그대로 사용?
- 구하고자 하는 것은 가중치
 - 음수값인 내적값을 그대로 사용 X
 - 내적값에 함수를 적용하여 정규화



Attention Decoder



Context Vector

Attention - Encoder에서 전달받은 hs 행렬 중
현시점에 가장 필요한 정보를 담은 벡터

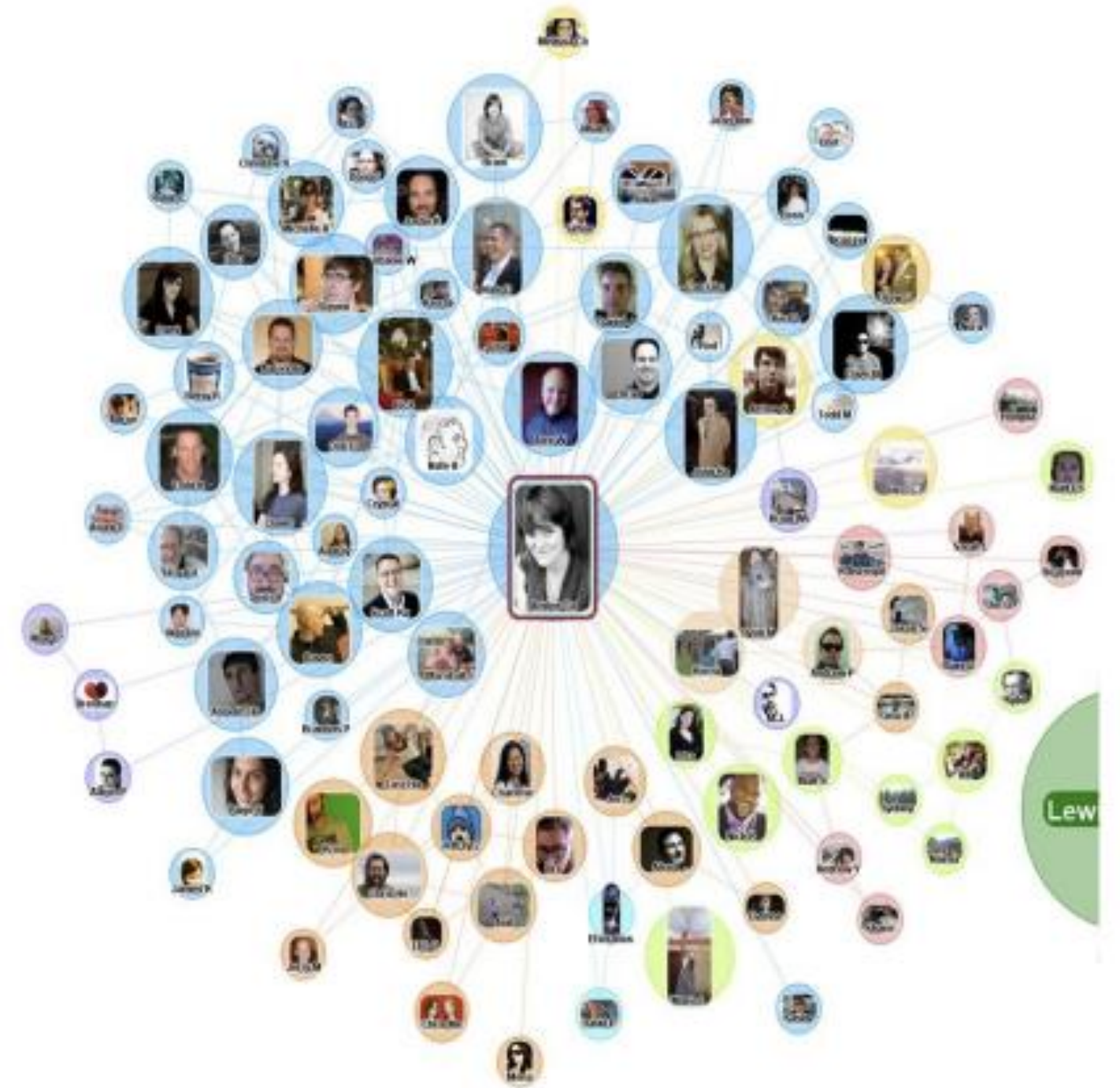
Context Vector 효과

단어 예측

- Decoder : 입력 문장의 특정 부분에 집중 O
- 입력 문장의 중요한 정보들을 효과적으로 활용 O

Transformer의 Main Idea

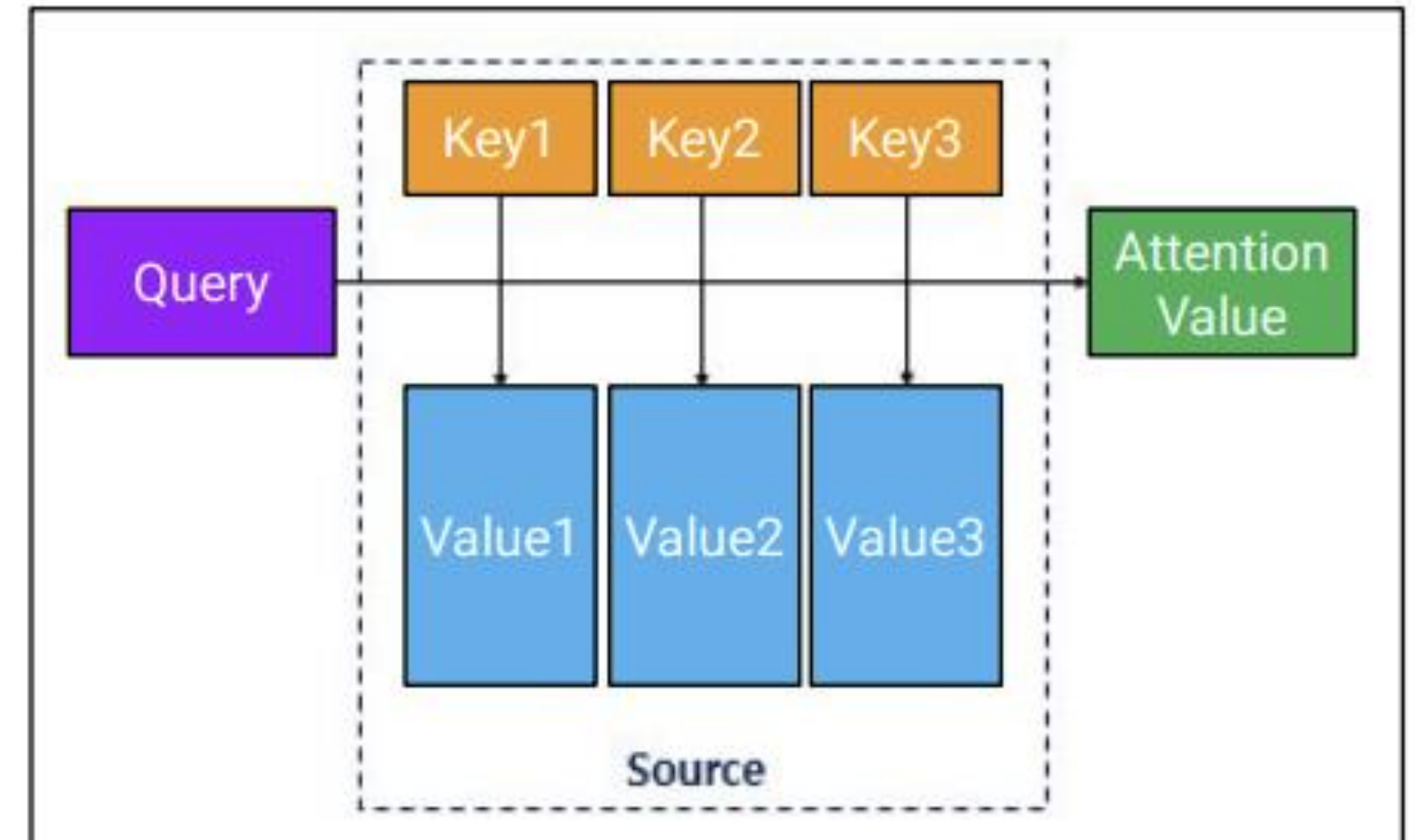
- 기본가정 : input x 는 서로 유기적으로 관련된 여러 요소로 분할될 수 있다.
 - 사회 속 사람들
 - 문장 속의 단어
 - 비디오의 프레임
 - ...
- Self-attention : 각 element는 해당 context(input의 다른 요소)에 참여하여 자신의 representation을 개선하는 방법으로 학습함



Attention Mechanism

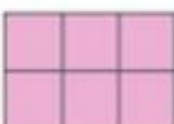
Attention 함수 : $\text{Attention}(Q, K, V)$

- Q(query) : 비교하는 기준
 - K(key) : 비교하는 대상
- V(value): 비교하는 실질적인 값
 - Q와 K는 동일한 차원
- V는 attention value와 동일한 차원
- Q와 K의 Dot-Product로 attention score를 구한 후 softmax를 취해 attention coefficient를 구함
- 이렇게 구한 attention coefficient에 V를 곱한 후 Weighted sum하여
퀴즈 3번 인 Z를 구함



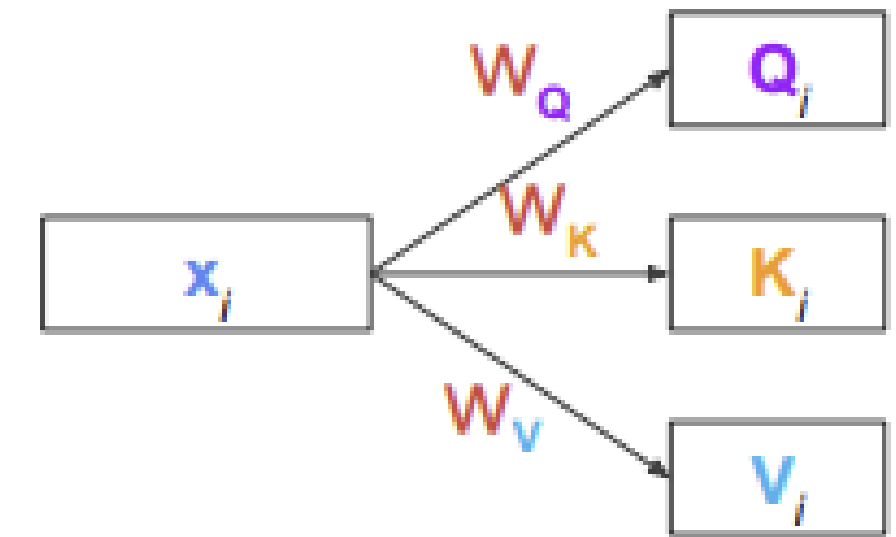
$$\text{softmax} \left(\frac{Q \times K^T}{\sqrt{d_k}} \right) V$$

Z

= 

transformer가 기존의 attention과 다른점은 Q,K,V를 만들 때 weight parameter를 사용한다는 점

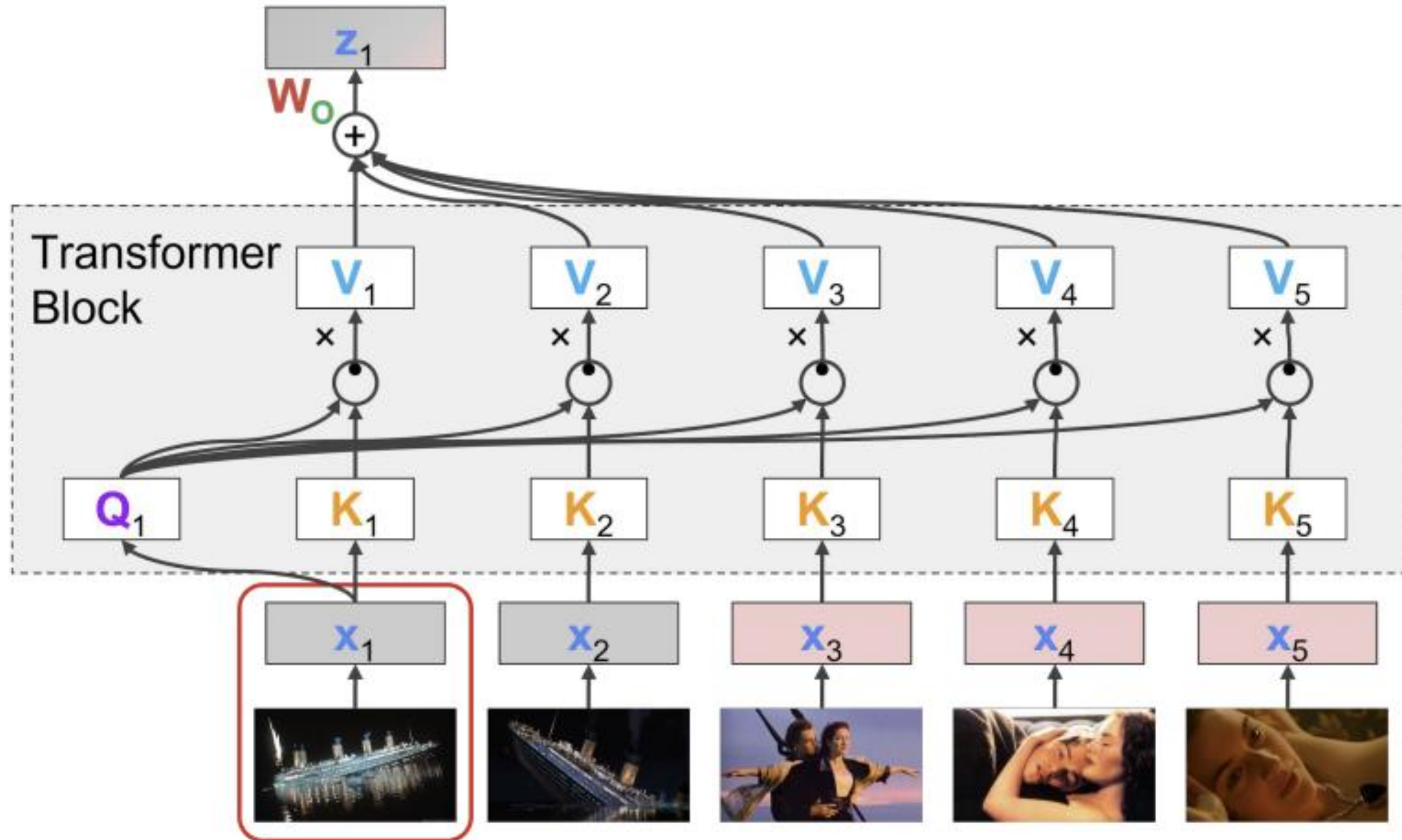
input token x 가 주어지면 (W_Q, W_K, W_V) 를 각각 곱해서 linear transformation에 의해 각각의 Q,K,V벡터로 매핑시킴



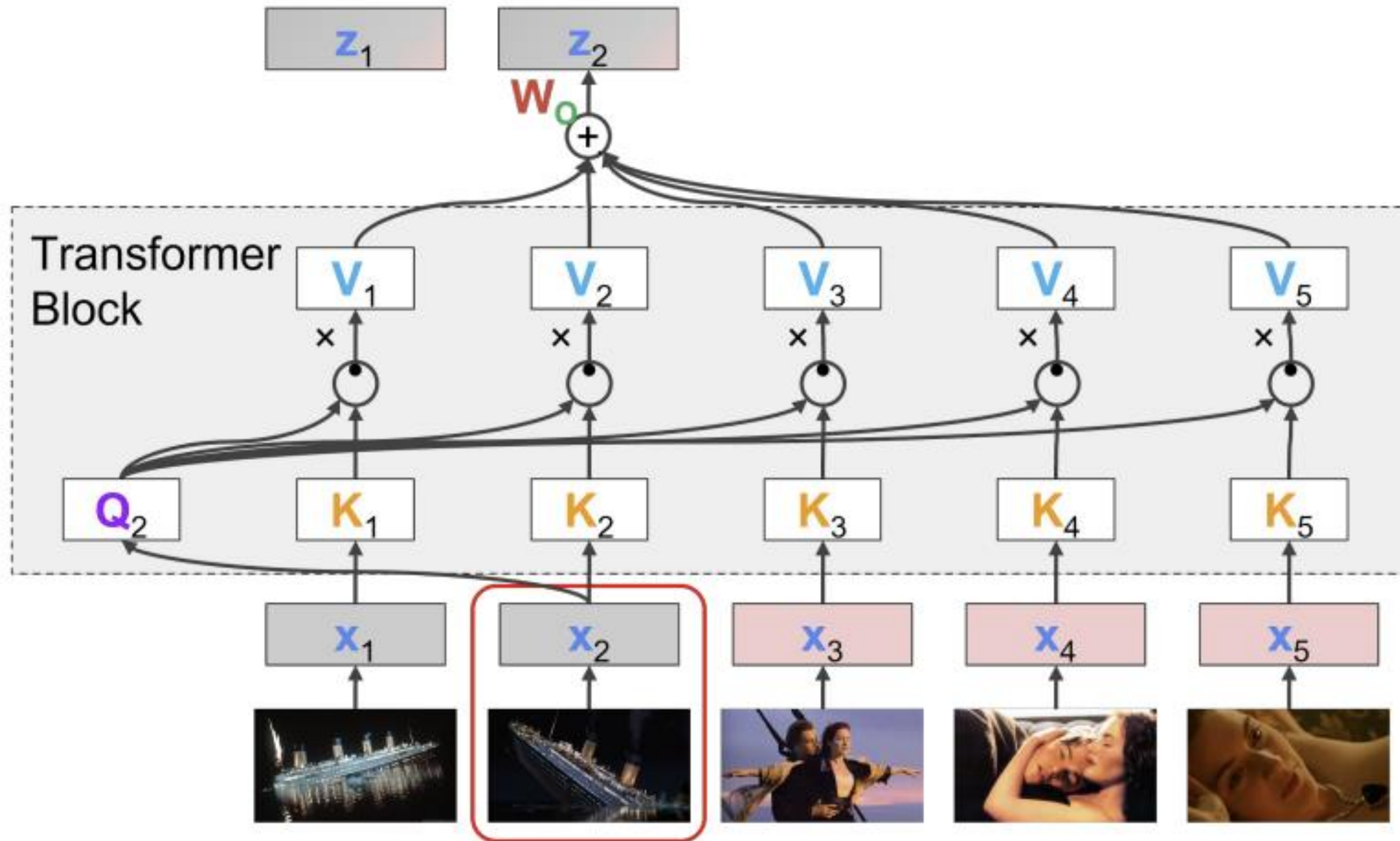
attention value를 원래 공간에 다시 매핑하는 또 다른 parameter인 W_O 도 필요함



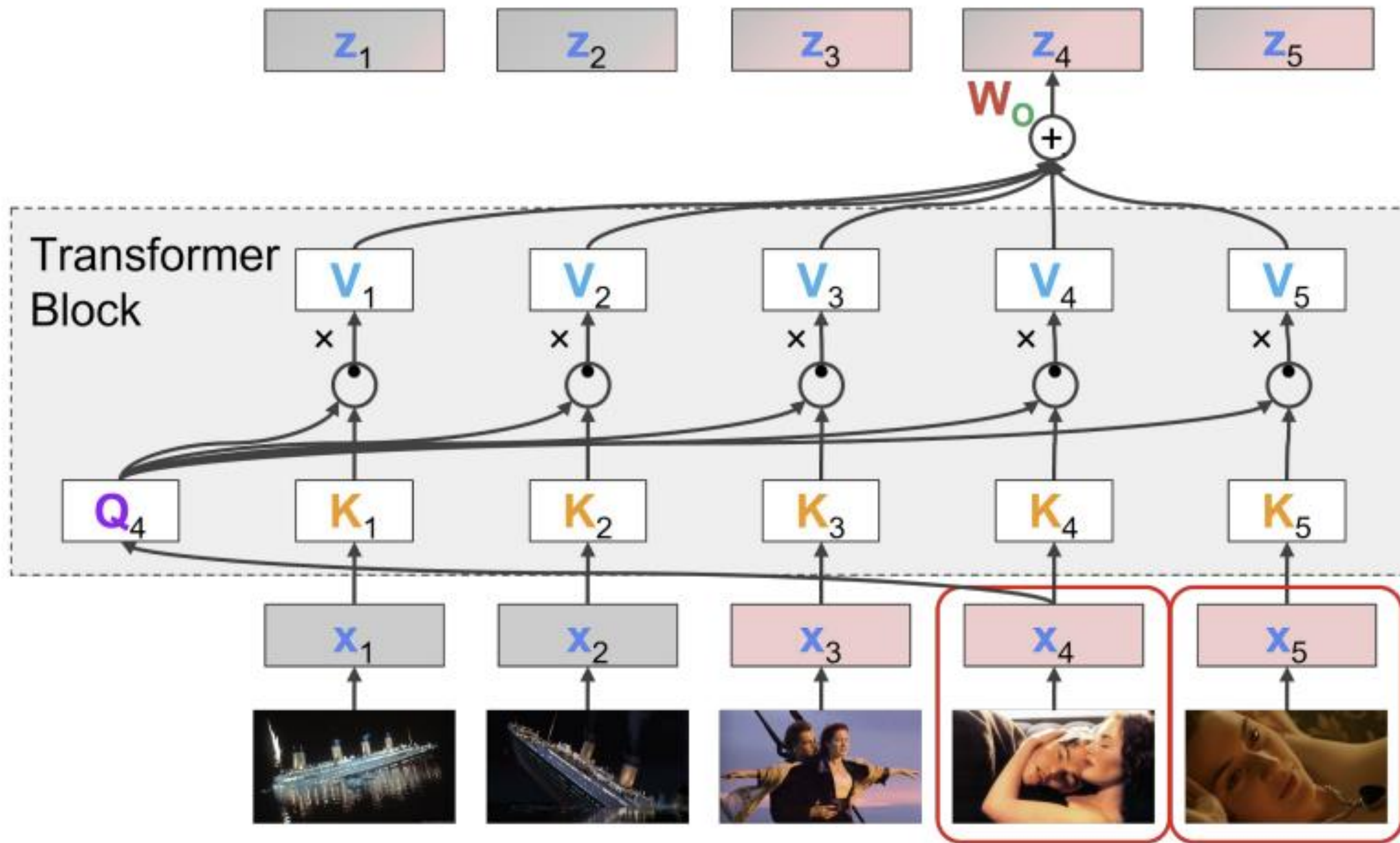
Attention Mechanism



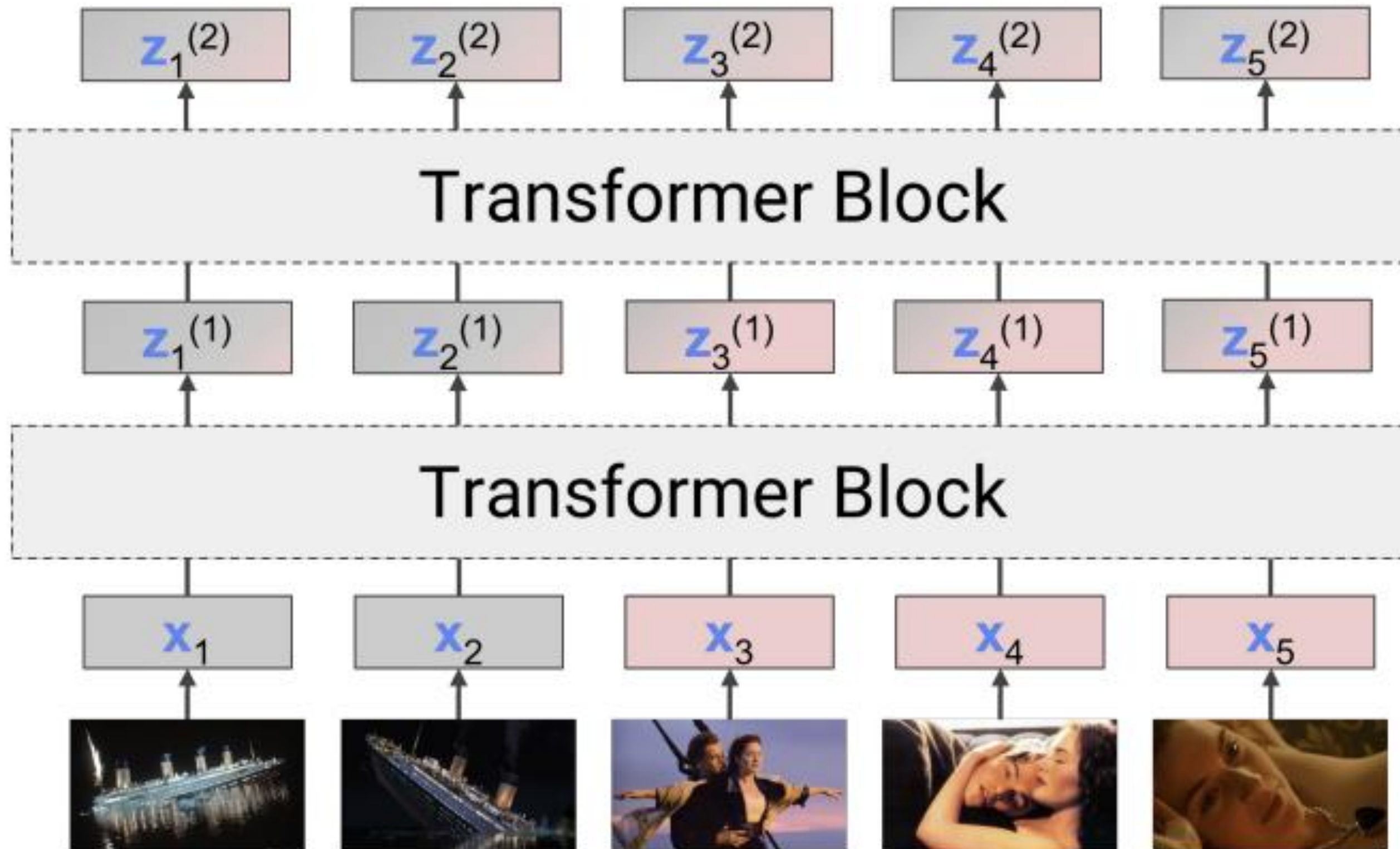
Attention Mechanism



Attention Mechanism



Attention Mechanism



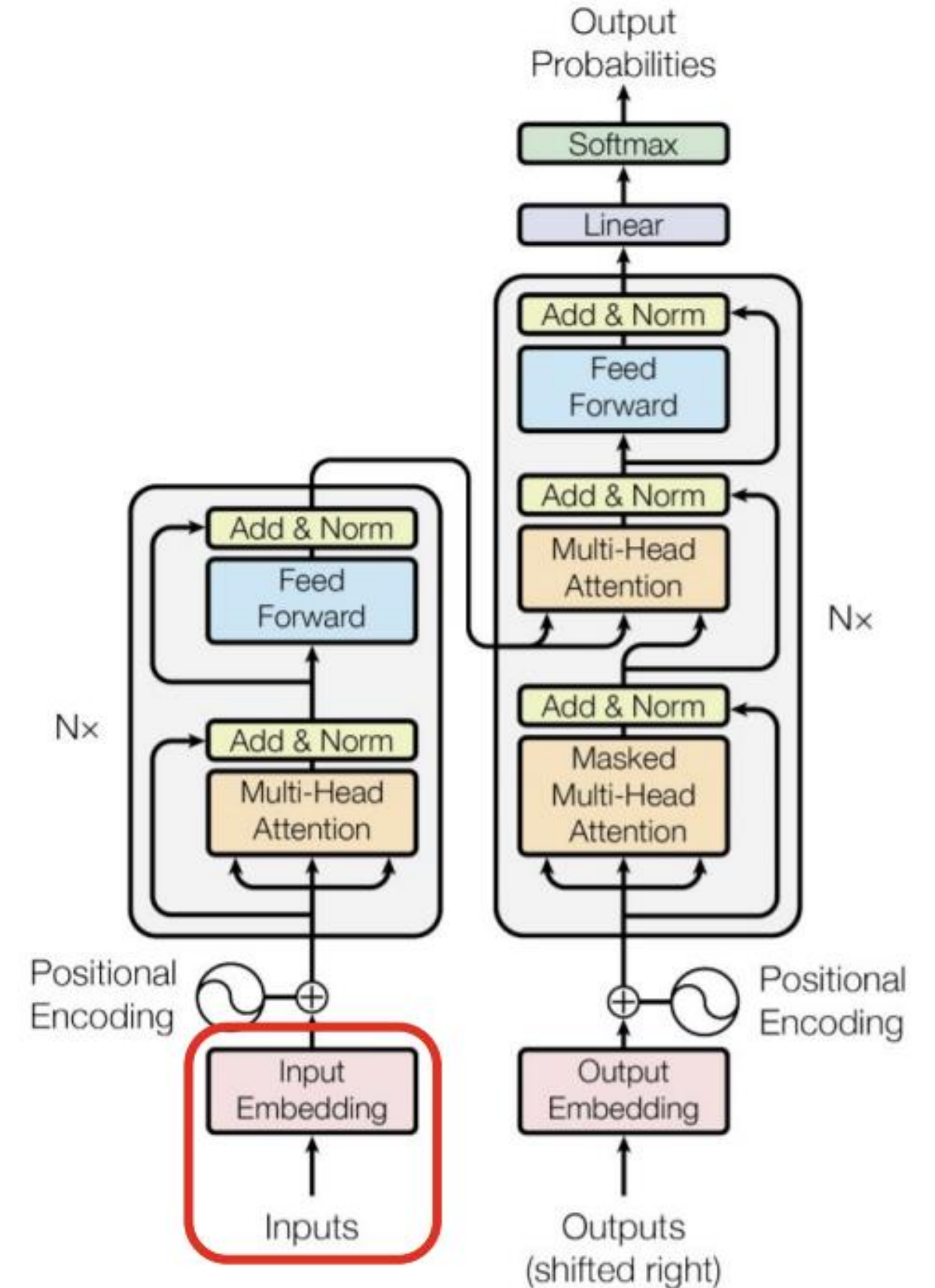
Transformer(Encoder)

Step 1 : input embedding

- input을 원하는 벡터로 임베딩
- 입력은 token들의 sequence로 주어짐

예)

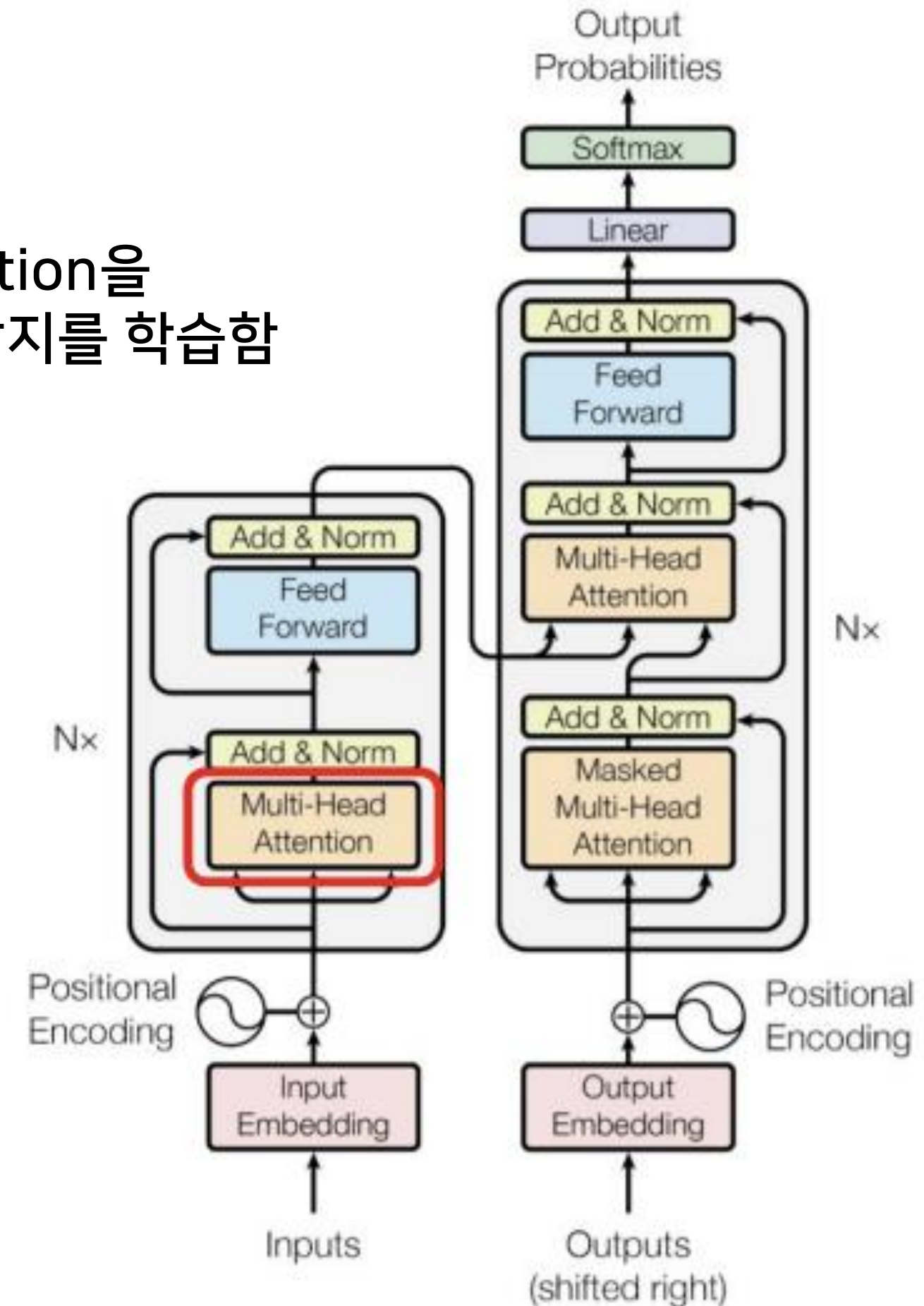
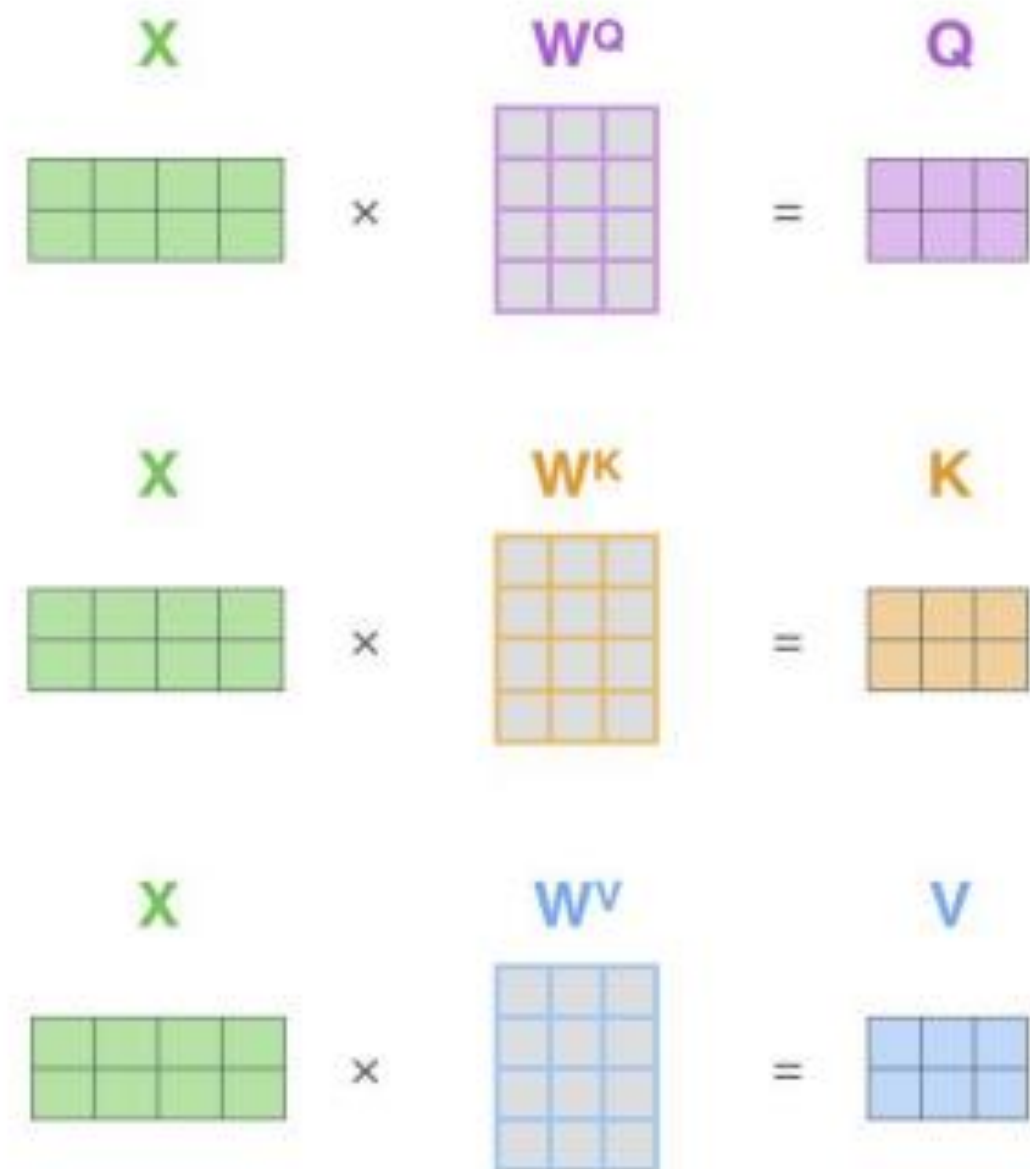
- Text : pre-trained 된 word embeddings (one-hot encoding)
- Image : 고정된 사이즈의 작은 이미지 패치
- Video : 프레임 임베딩



Transformer(Encoder)

Step 2 : Contextualizing the Embeddings

- Multi-head Attention을 통과함
- 각 단어에 대해 원래 embedding을 하는대신 linear transformation을 사용하여 query, key, value처럼 사용되는 Q,K,V를 어떻게 매핑할지를 학습함



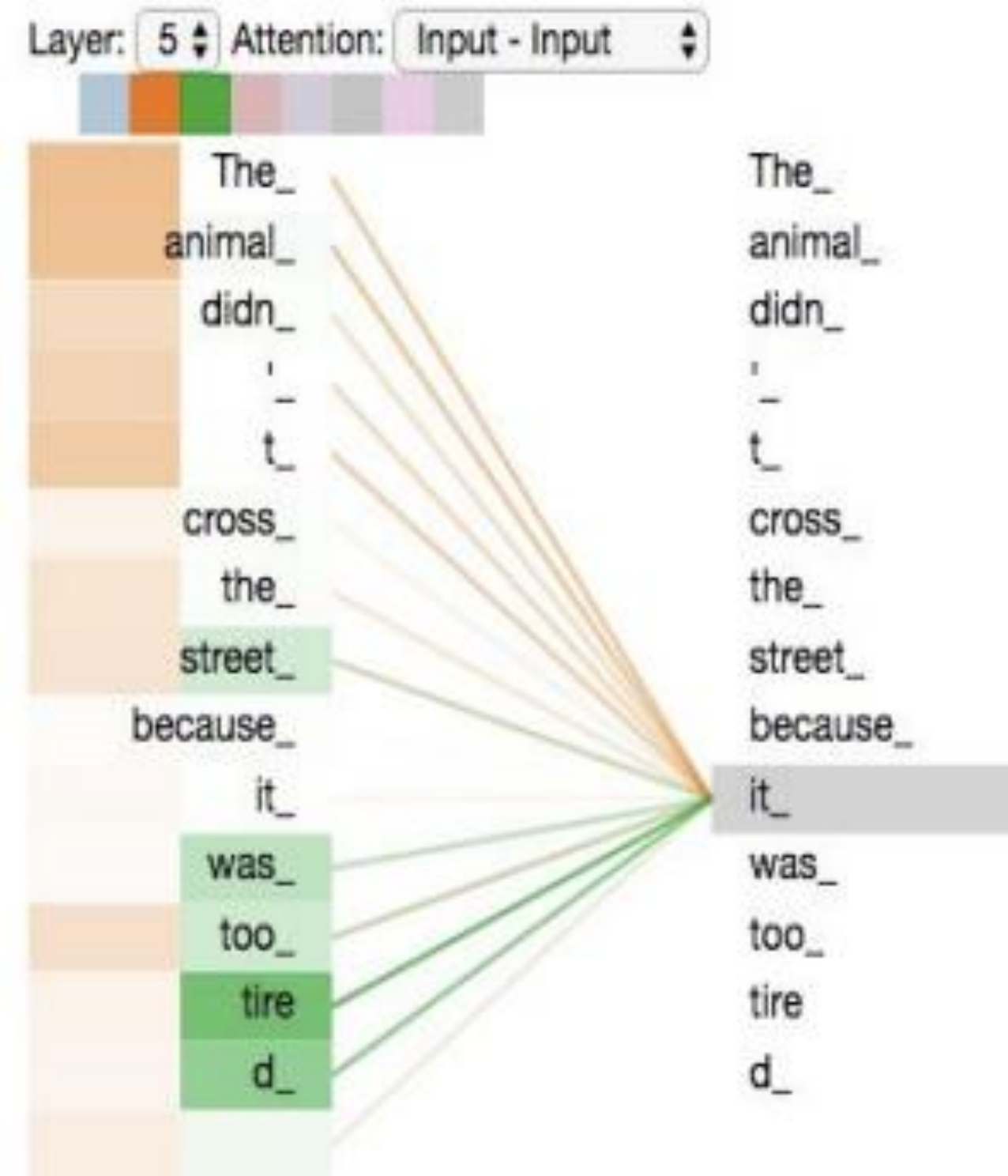
Transformer(Encoder)

Multi-head self-attention :

- Multi-head Self-attention은 Q,K,V로의 multiple projections을 가지도록 함

왜 Multi-head self-attention을 사용할까?

- 모델이 서로 다른 subspace에 있는 representation을 추출할 수 있어 성능 향상 측면에서 매우 유리함
- 예시에서 문장을 was까지만 본다면?
 - it의 예측은 animal과 street 둘 다 가능함
 - 이 두가지 경우를 모두 학습해야 한다!!



Transformer(Encoder)

1) Concatenate all the attention heads



2) Multiply with a weight matrix W^O that was trained jointly with the model

x



3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN



- Multiple self-attention은 multiple attention values(Z_0, Z_1, \dots, Z_{k-1})를 출력함
- 각각의 Z 들을 연결한 후 W^O 로 linear 변환을 하면 input sequence를 모두 고려한 하나의 embedding 벡터 Z 를 얻을 수 있음

Transformer(Encoder)

1) Concatenate all the attention heads



2) Multiply with a weight matrix W^O that was trained jointly with the model

x



3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN



- Multiple self-attention은 multiple attention values(Z_0, Z_1, \dots, Z_{k-1})를 출력함
- 각각의 Z 들을 연결한 후 W^O 로 linear 변환을 하면 input sequence를 모두 고려한 하나의 embedding 벡터 Z 를 얻을 수 있음

Transformer(Encoder)

Step 3 : Feed-forward Layer

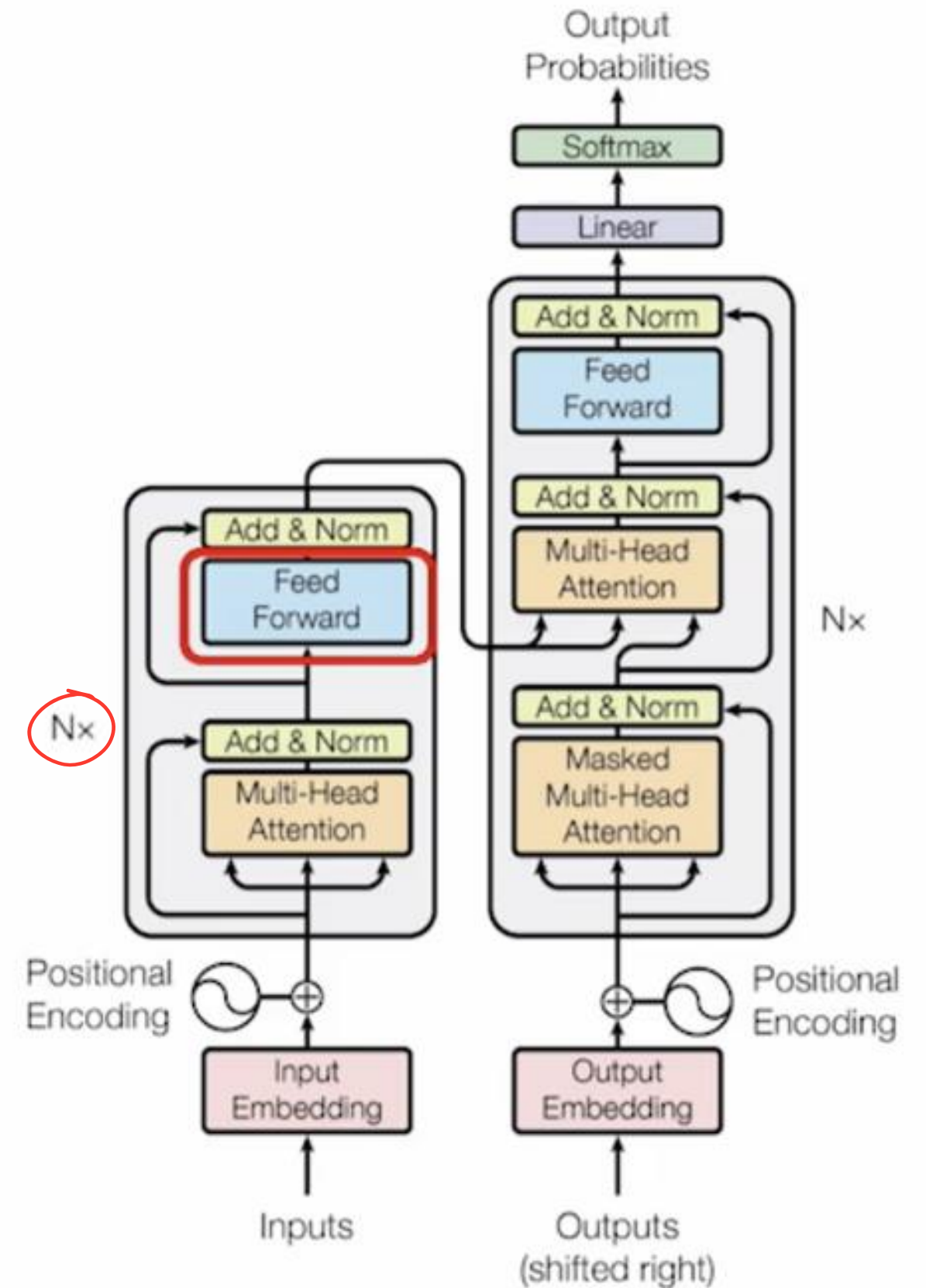
- 각 문맥화된 embedding은 추가적인 퀴즈 4번 를 통과함
- 추가적인 non linear operation을 적용해서 representation을 잘 학습하도록 함

Multi-head self-attention과 FC layer 끝에 Residual Connection과 Layer Normalization가 추가됨

- feed-forward가 필요하지 않은 경우 건너뛸 수 있게 해주고 정규화를 해줌

Stacked Self-attention

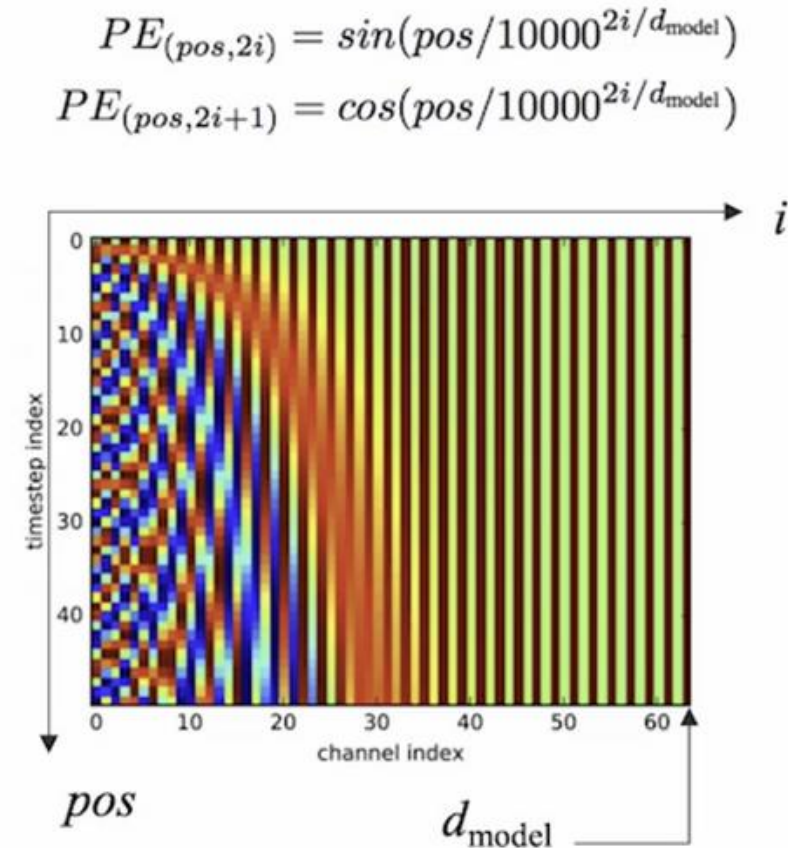
- 여러번(N 번) 쌓아서 반복하면 더 복잡한 패턴학습 가능



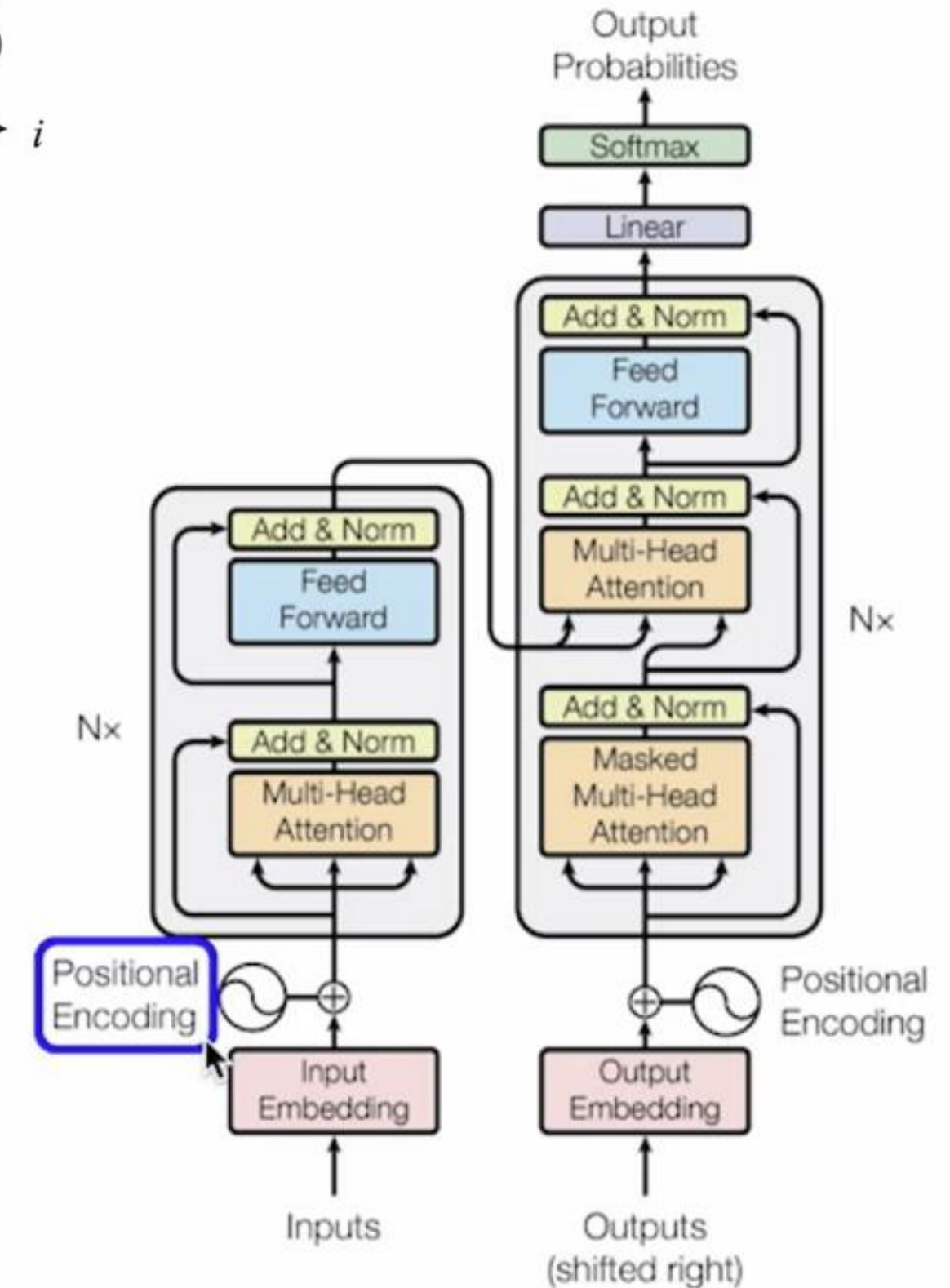
Transformer(Encoder)

Positional Encoding

- RNN과 달리 각각의 토큰에 순서 개념이 없다
- 단어가 섞여도 같은 임베딩 벡터가 나옴
⇒ 각각의 단어가 독립적으로 연산되기 때문
- 이때 순서개념을 넣어주는 것



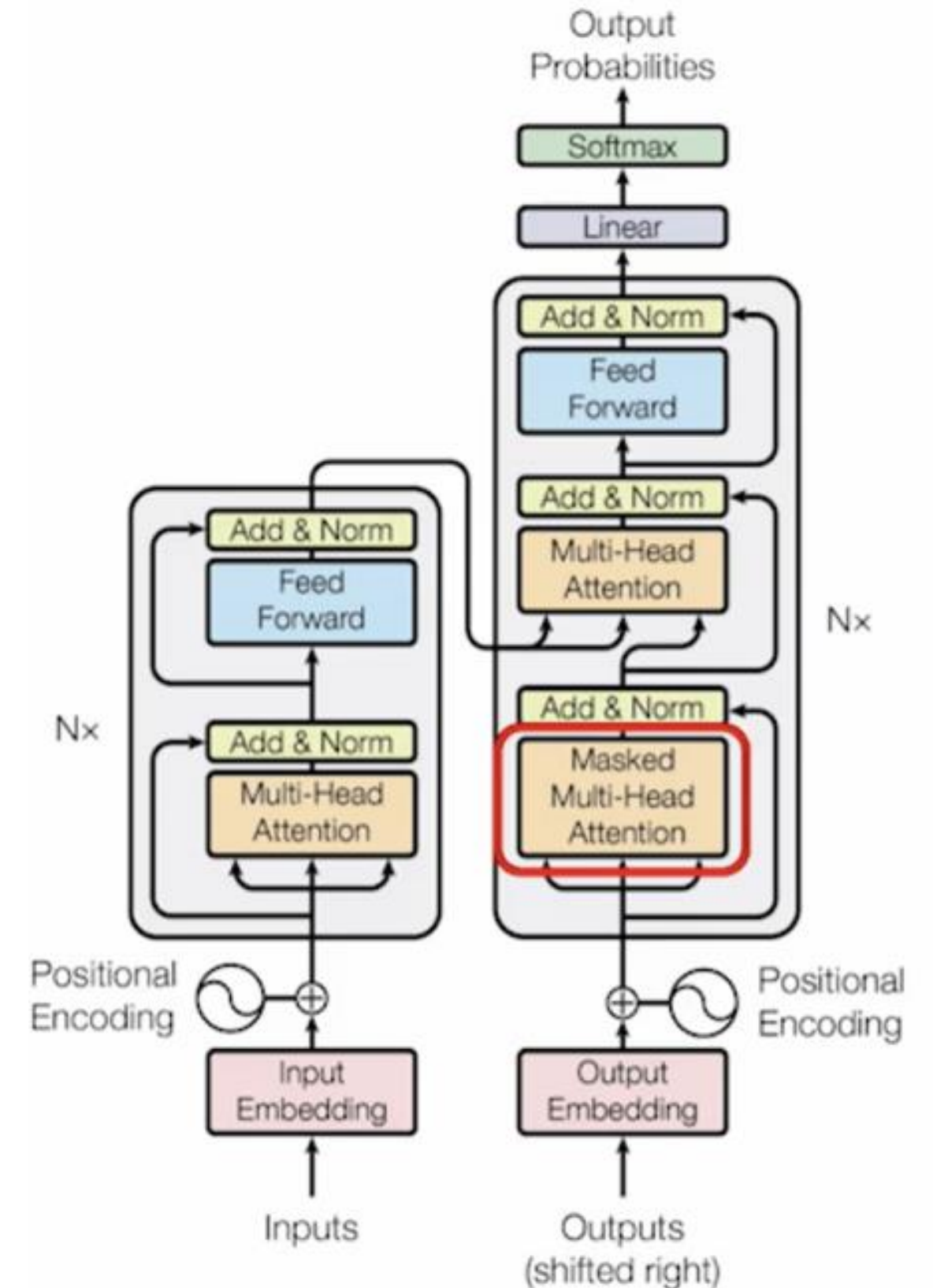
- sinusoidal encoding을 사용해서 각각 순서를 나타내는 특유한 벡터로 만듦
- 각각의 i마다 다른 벡터를 input embedding 벡터에 더해주면서 각각의 벡터들이 순차적인 시퀀스를 반영할 수 있도록 해줌
 - 그림을 보면 각각의 i가 세로축의 선에 대해 모두 다른 패턴을 가짐
 - 하지만 가까우면 유사한 포지셔널 인코딩을 가짐



Transformer(Decoder)

Step5 : Masked Multi-Head Self-attention

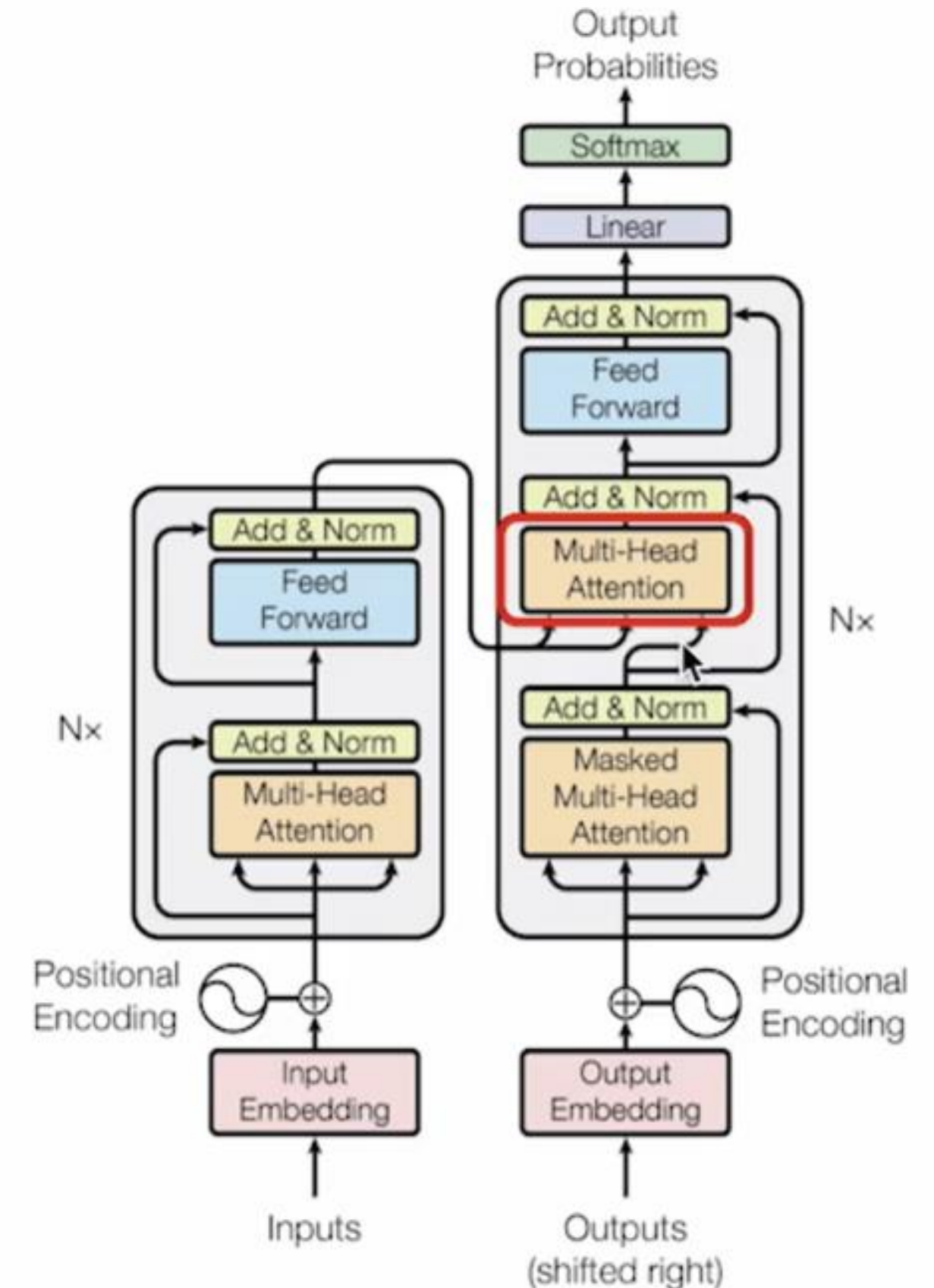
- input sequence를 인코딩 할때는 모든 문장이 들어감 그러나 디코더에선 전체 시퀀스를 볼 수 없기 때문에 마스킹이 필요함
- 디코더에서는 아는 정보만 unmasked처리를 해주고, 앞으로 예측할 정보는 mask처리 함 나머지는 인코딩의 multi-head-attention과 동일함



Transformer(Decoder)

Step 6 : Encoder-Decoder Attention

- Masked Multi-Head attention의 결과를 Q로 사용
- K,V는 퀴즈 5번 에서 전달받음
 - input의 embedding 벡터와의 상관관계를 통해 학습을 해야기 때문
 - 인코더의 단어들과 디코더의 단어들의 유사도를 통해서 어떤 단어에 더 집중할지를 정해줌
- encoder의 단어들과 decoder의 단어들의 유사도를 통해서 어떤 단어에 더 집중할지를 정해줌



Transformer(Decoder)

Step7 : Feed-forward Layer

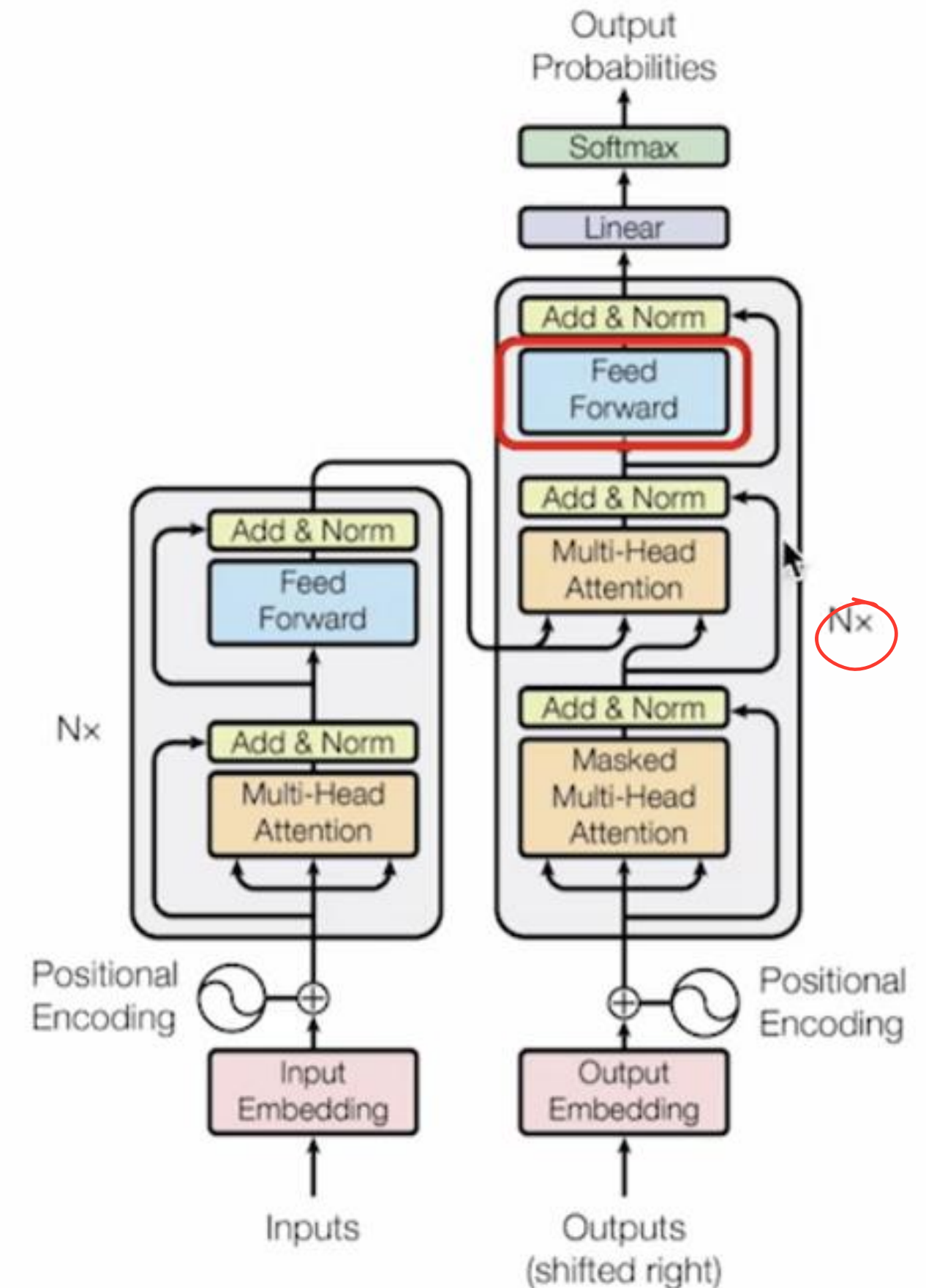
- Encoder와 동일하게 Residual, normalization을 적용
- Encoder 와 동일하게 N번 쌓아서 반복

Step 8 : Linear Layer

- output embedding을 linear 변환해서 class score로 매핑함

Step 9 : Softmax Layer

- class scores에 softmax를 취해서 0과 1사이의 확률로 변환함
- softmax의 성질을 이용해서 output의 총 합은 1이 되어야함
- 이를 이용해서 ground truth와 비교해서 loss를 가지고 backprop함
- 이 Decoding 단계는 다음 단어가 [EOS] (End of Sentence)토큰으로 예측될 때 까지 반복됨



Reference

<https://coshin.tistory.com/47>

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

<https://rahites.tistory.com/74> <https://littlefoxdiary.tistory.com/4> <https://techy8855.tistory.com/10>

<https://www.youtube.com/watch?v=0lgWzluKq1k> <https://www.youtube.com/watch?v=PipiRRL50p8>

<https://www.youtube.com/watch?v=4DzKM0vgG1Y> <https://wikidocs.net/21697>

<https://wikidocs.net/31695> <https://miinkang.tistory.com/23>

<https://velog.io/@nochesita/%EB%94%A5%EB%9F%AC%EB%8B%9D-%EC%96%B4%ED%85%90%EC%85%98-Attention-1>

<https://www.boostcourse.org/boostcampaitch7/lecture/1543556?isDesc=false>

감사합니다

BITAMIN 13&14
