

NLP 개론 (CH.8)

어텐션

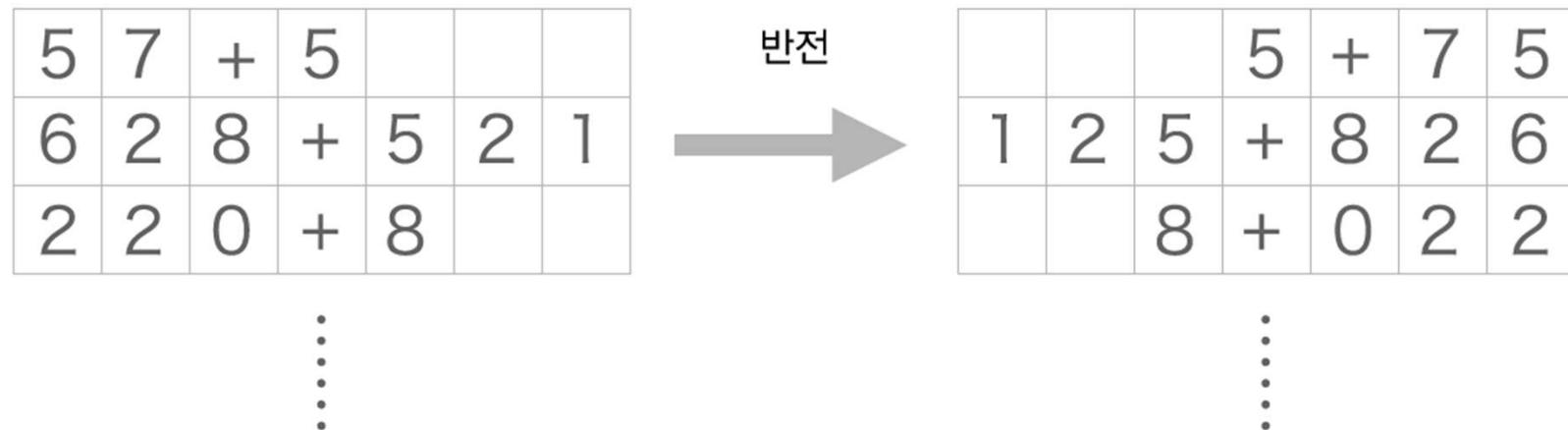
Seq2seq의 개선안

- Seq2seq의 개선안 복습
 - Reverse : 가중치 업데이트가 바로 될 수 있게, 가까이 붙일 수 있는 구조로 만들어준다.
 - Peeky : Decoder에서 받는 h 라는 가중치를 전체 LSTM에 INPUT으로.

Seq2seq의 개선안

- Seq2seq의 개선안 복습
 - Reverse : 가중치 업데이트가 바로 될 수 있게, 가까이 붙일 수 있는 구조로 만들어준다.
 - Peeky : Decoder에서 받는 h 라는 가중치를 전체 LSTM에 INPUT으로.

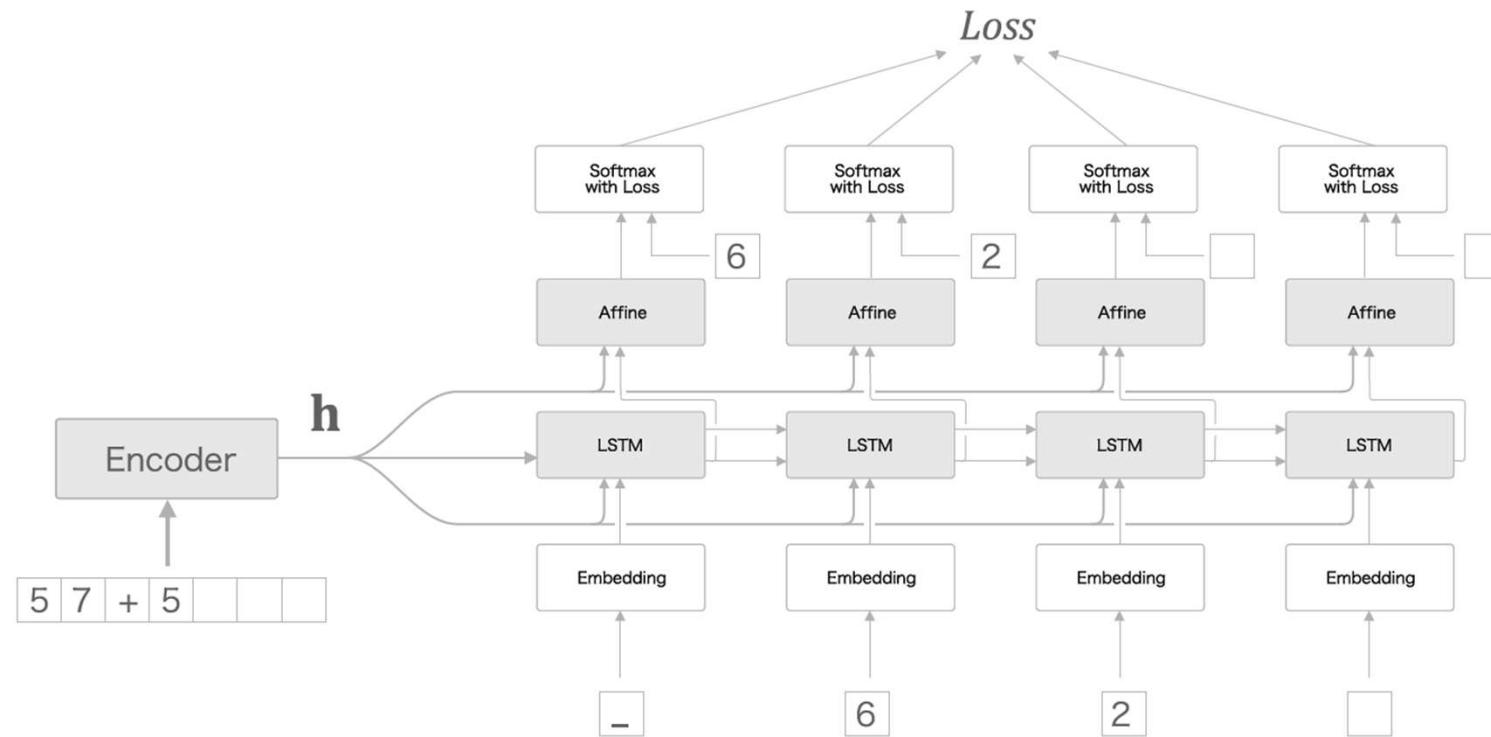
그림 7-23 입력 데이터를 반전시키는 예



Seq2seq의 개선안

- Seq2seq의 개선안 복습
 - Reverse : 가중치 업데이트가 바로 될 수 있게, 가까이 붙일 수 있는 구조로 만들어준다.
 - Peeky : Decoder에서 받는 가중치 h 를 전체 LSTM에 INPUT 으로.

그림 7-26 개선 후: Encoder의 출력 h 를 모든 시각의 LSTM 계층과 Affine 계층에 전해준다.

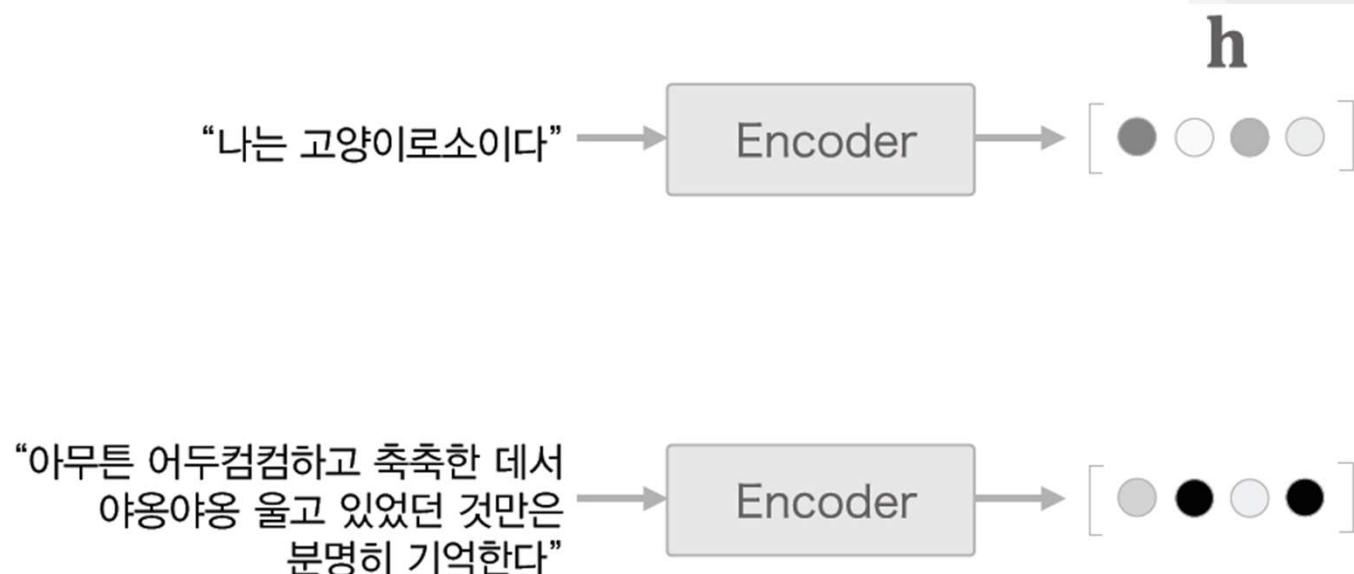


Seq2seq의 개선안

- Seq2seq의 작은 개선안 복습
 - Reverse : 가중치 업데이트가 바로 될 수 있게, 가까이 붙일 수 있는 구조로 만들어준다.
 - Peeky : Decoder에서 받는 h 라는 가중치를 전체 LSTM에 INPUT으로.
- Seq2seq의 새로운 개선안
 - 어텐션 : seq2seq를 한층 더 강력하게 하는 메커니즘
 1. Seq2seq의 근본적인 문제점
 2. 개선방안(Attention 소개)
 3. Attention 계층 구현
 4. 어텐션의 용용, 개선 소개 _ 11/26 이상수 석사과정생

Seq2seq의 근본적인 문제점

- Seq2seq의 Encoder
 - : Encoder가 시계열 데이터를 인코딩
-> 인코딩된 정보를 Decoder에 전달
 - * 이 때, Encoder의 출력은 고정 길이의 벡터
- 고정 길이 벡터의 문제점
 - 고정 길이 벡터 : 문장의 길이에 관계없이, 항상 같은 길이의 벡터로 변환한다.
EX) 번역에서는, 너무 긴 문장도 항상 같은 길이의 벡터에 삽입



Seq2seq의 근본적인 문제점

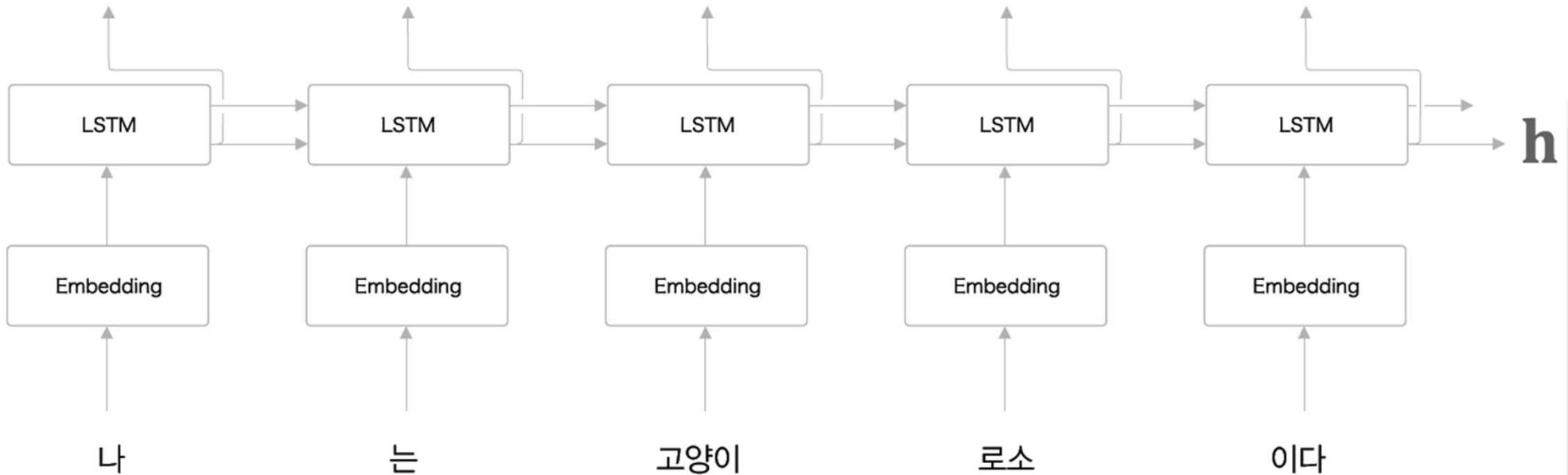
- Seq2seq의 근본적인 문제점 정리
: Encoder가 데이터를 인코딩하여 전달할 때,
고정 길이의 벡터로 전달하기에
필요한 정보가 벡터에 다 담기지 못하는 문제가 발생한다.

개선 순서

1. Encoder를 개선
2. 이후, Decoder을 개선

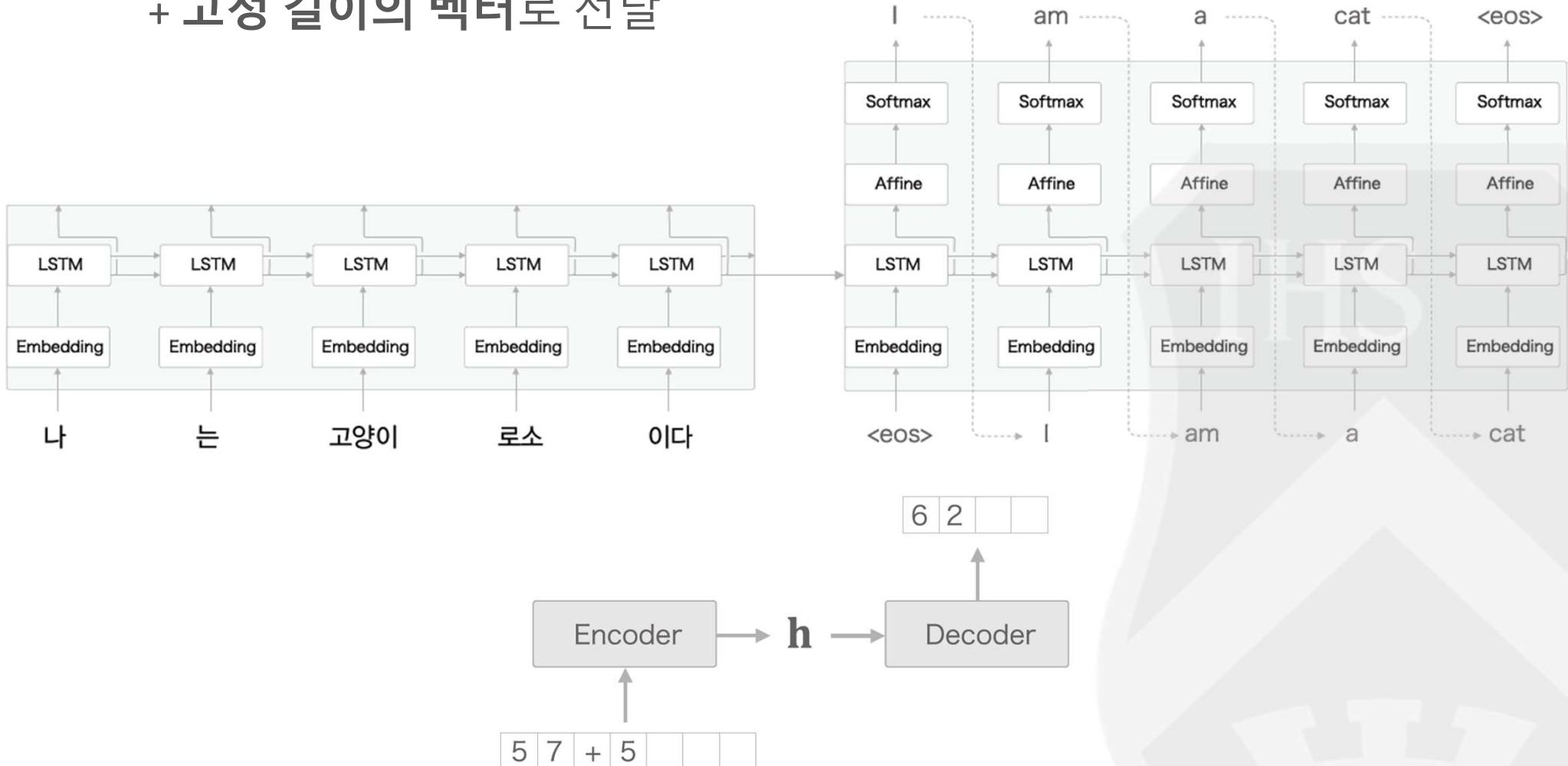
Encoder 개선

- 개선 전 Encoder:
: LSTM 계층의 마지막 은닉 상태만을 Decoder에 전달
+ 고정 길이의 벡터로 전달



Encoder 개선

- 개선 전 Encoder
 - : LSTM 계층의 마지막 은닉 상태만을 Decoder에 전달
 - + 고정 길이의 벡터로 전달



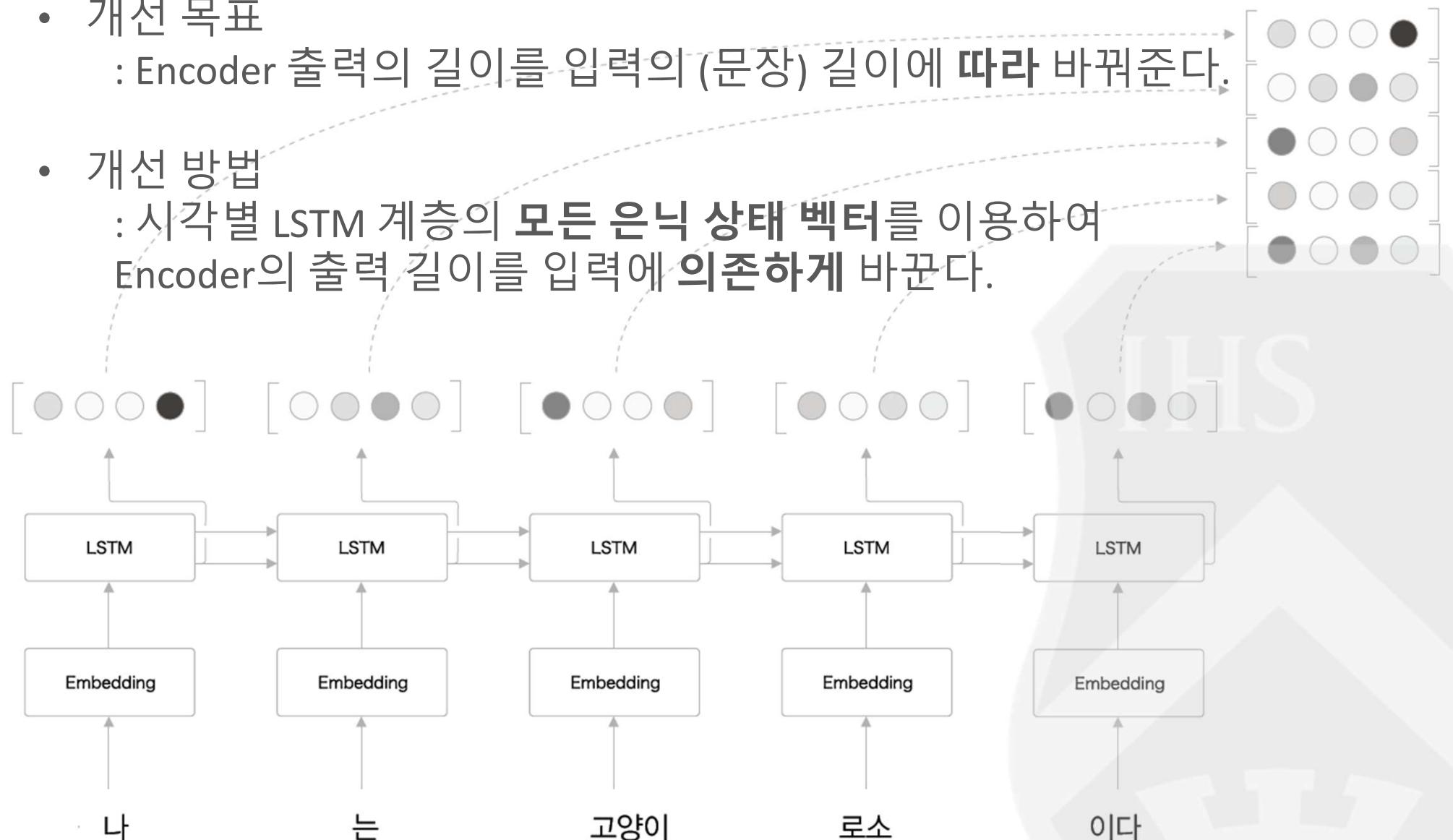
Encoder 개선

- 개선 목표
: Encoder 출력의 길이를 입력의 (문장) 길이에 따라 바꿔준다.
- 개선 방법
: 시각별 LSTM 계층의 모든 은닉 상태 벡터를 이용하여
Encoder의 출력 길이를 입력에 의존하게 바꾼다.

Encoder 개선

hs

- 개선 목표
: Encoder 출력의 길이를 입력의 (문장) 길이에 따라 바꿔준다.
- 개선 방법
: 시각별 LSTM 계층의 모든 은닉 상태 벡터를 이용하여 Encoder의 출력 길이를 입력에 의존하게 바꾼다.

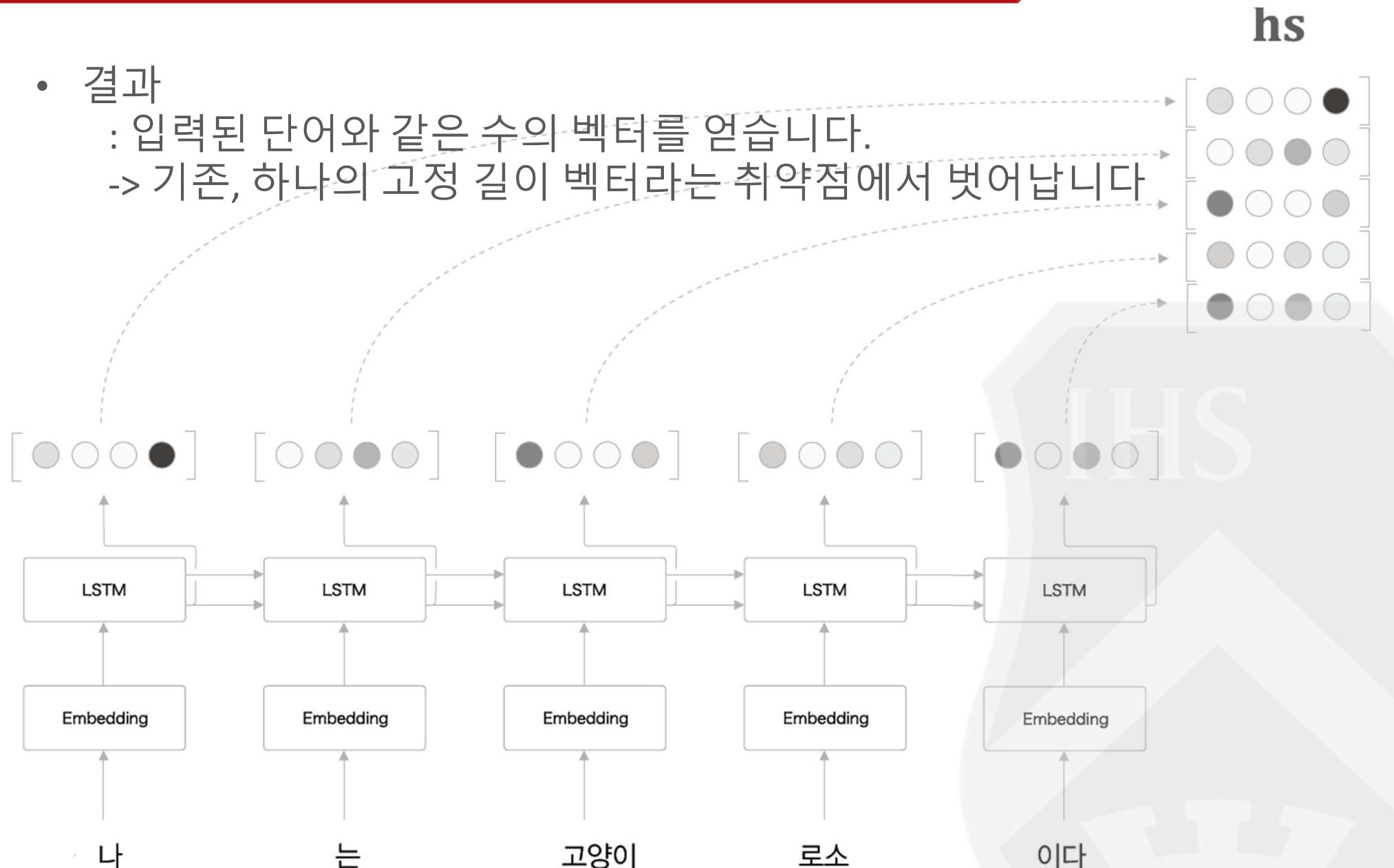


Encoder 개선

- 결과

: 입력된 단어와 같은 수의 벡터를 얻습니다.

-> 기존, 하나의 고정 길이 벡터라는 취약점에서 벗어납니다.

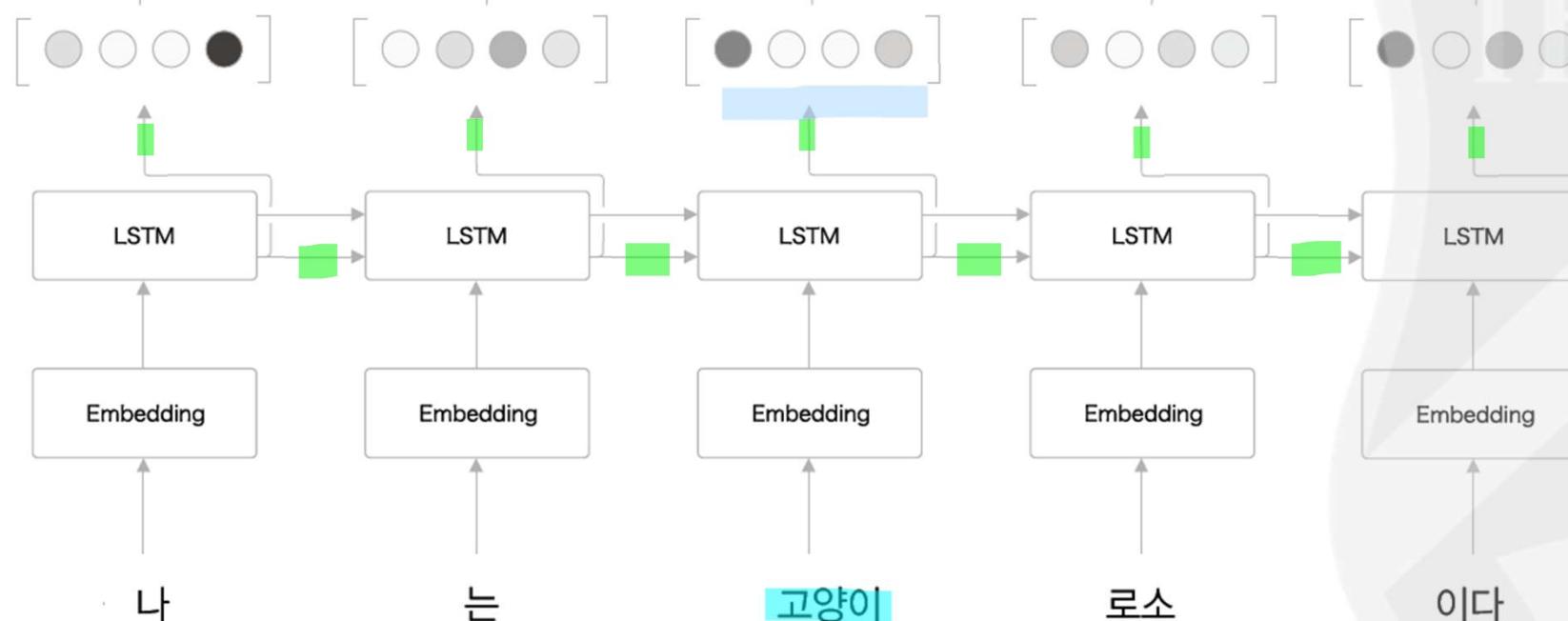


Encoder 개선

hs

- Observation in Hidden state of LSTM
: LSTM의 각 시각의 은닉 상태는
직전에 입력된 정보가 많이 포함되어 있는 '내용'이다.

Ex) 고양이 단어를 입력했을 때의 LSTM 은닉 상태는
직전에 입력한 고양이 라는 단어의 영향을 가장 크게 받는다
-> 고양이의 성분이 많이 들어간 벡터

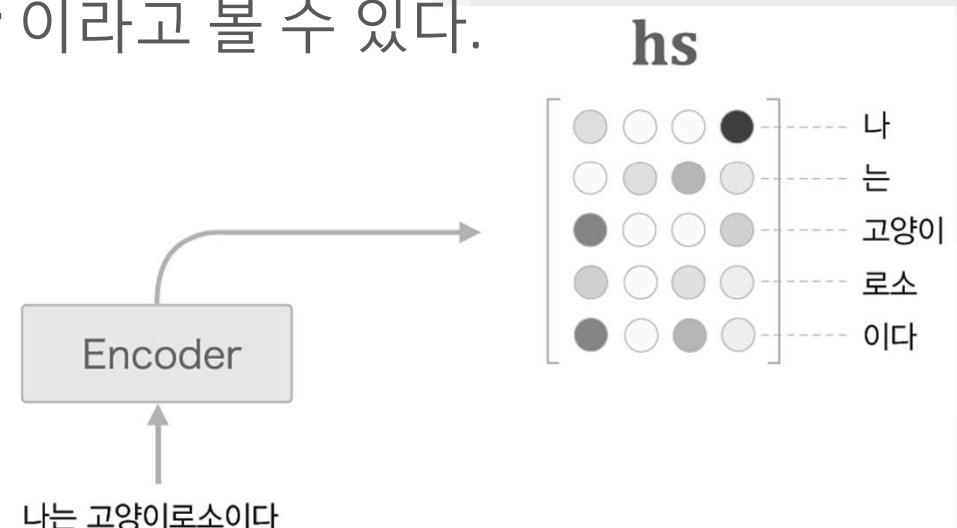


Encoder 개선

- Observation in Hidden state of LSTM
: LSTM의 각 시각의 은닉 상태는
직전에 입력된 정보가 많이 포함되어 있는 '내용'이다.

Ex) 고양이 단어를 입력했을 때의 LSTM 은닉 상태는
직전에 입력한 고양이 라는 단어의 영향을 가장 크게 받는다
→ 고양이의 성분이 많이 들어간 벡터

따라서, Encoder가 출력하는 hs 행렬은
각 단어에 해당하는 벡터들의 집합이라고 볼 수 있다.



Encoder 개선

- 개선 전 Encoder
 - : LSTM 계층의 마지막 은닉 상태만을 Decoder에 전달
+ 고정 길이의 벡터로 전달
- 개선 목표
 - : Encoder 출력의 길이를 입력의 (문장) 길이에 따라 바꿔준다.
- 개선 방법
 - : 시각별 LSTM 계층의 모든 은닉 상태 벡터를 이용하여 Encoder의 출력 길이를 입력에 의존하게 바꾼다.
- 결과
 - : 입력된 단어와 같은 수의 벡터를 얻습니다.
-> 기존, 하나의 고정 길이 벡터라는 취약점에서 벗어남

Encoder 개선

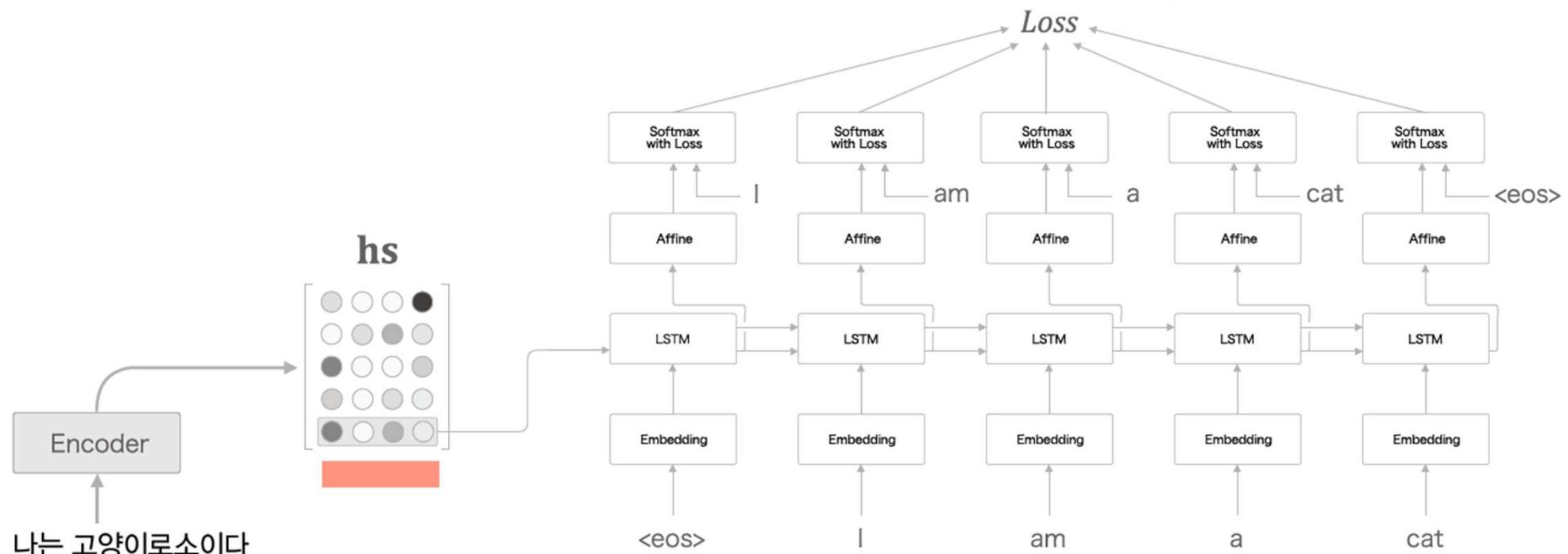
- Deocoder에서는 어떻게 활용해야 할지?

1. Decoder 개선 1_ 선택 작업 계층
2. Decoder 개선 2_ 가중치 계산 계층
3. Decoder 개선 3_ 1,2 결합 계층

Decoder 개선 목표

- Encoder가 개선된 후 상황
: Encoder는 각 단어에 대응하는 LSTM 계층의 은닉 상태 벡터를 hs로 모아 출력

개선 전) LSTM 계층의 마지막 은닉 상태를 출력
= hs에서의 마지막 행만 출력



Decoder 개선 목표

- Encoder가 개선 된 후 상황
: Encoder는 각 단어에 대응하는 LSTM 계층의 은닉 상태 벡터를 hs로 모아 출력

Decoder 목표

: hs를 전부 활용할 수 있도록 개선

아이디어

: 번역 할 때의 **직독직해** 상황

즉, 어떤 단어에 주목하여 그 단어의 변환을 수시로 진행

-> 같은 과정을 Seq2seq로 재현

즉, **입력과 출력**의 여러 단어중 어떤 단어끼리 서로 관련되어 있는가
라는 **대응 관계**를 Seq2seq에게 학습

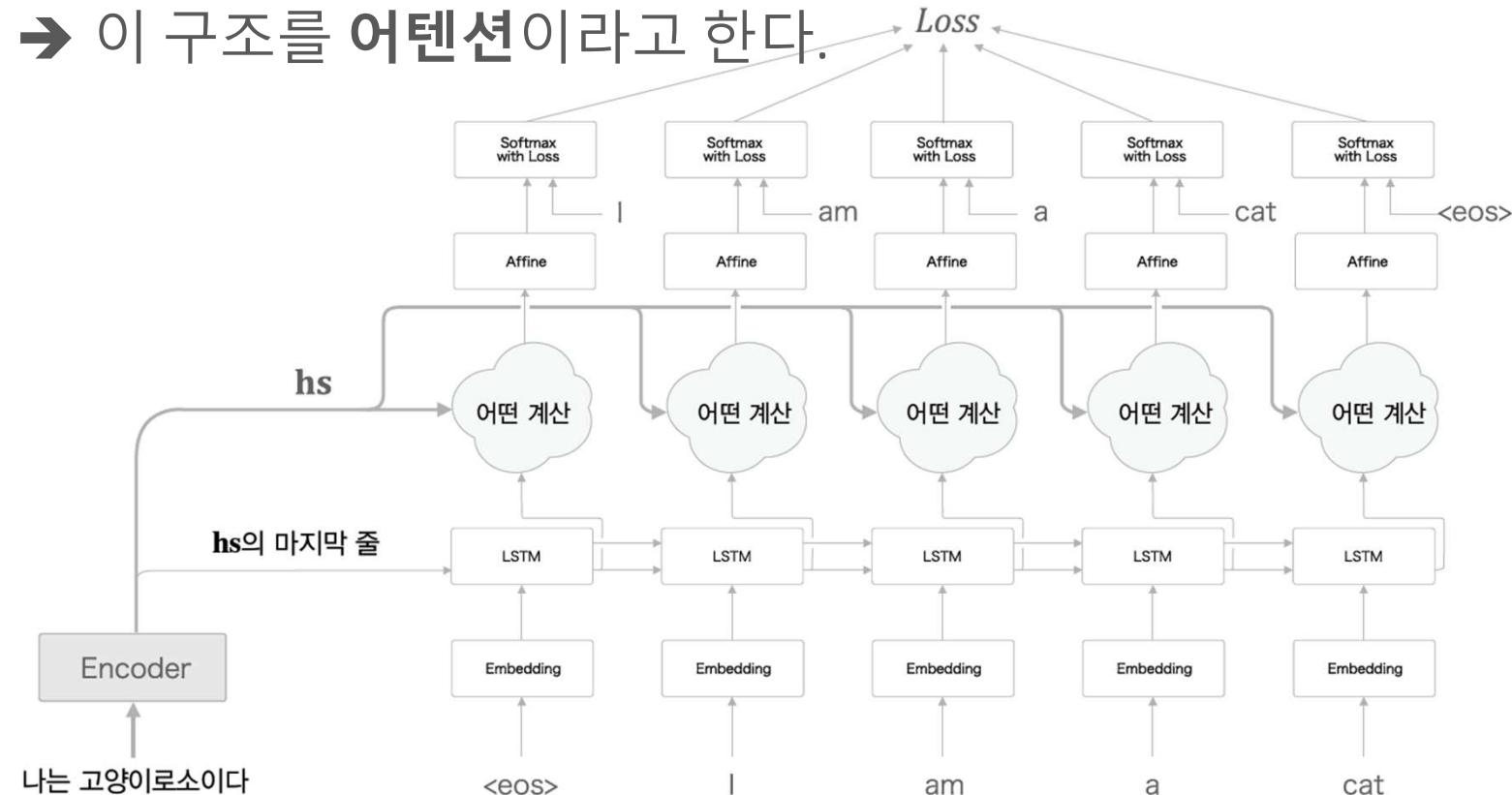
*단어간의 대응 관계를 나타내는 정보 : Alignment

Decoder 개선 목표

- 목표

- 도착어 단어와 대응 관계에 있는 출발어 단어의 정보를 골라내는 것
- 이 정보를 이용하여 번역을 수행

즉, 필요한 정보에만 주목하여 그 정보로부터 시계열 변환을 수행
 → 이 구조를 어텐션이라고 한다.

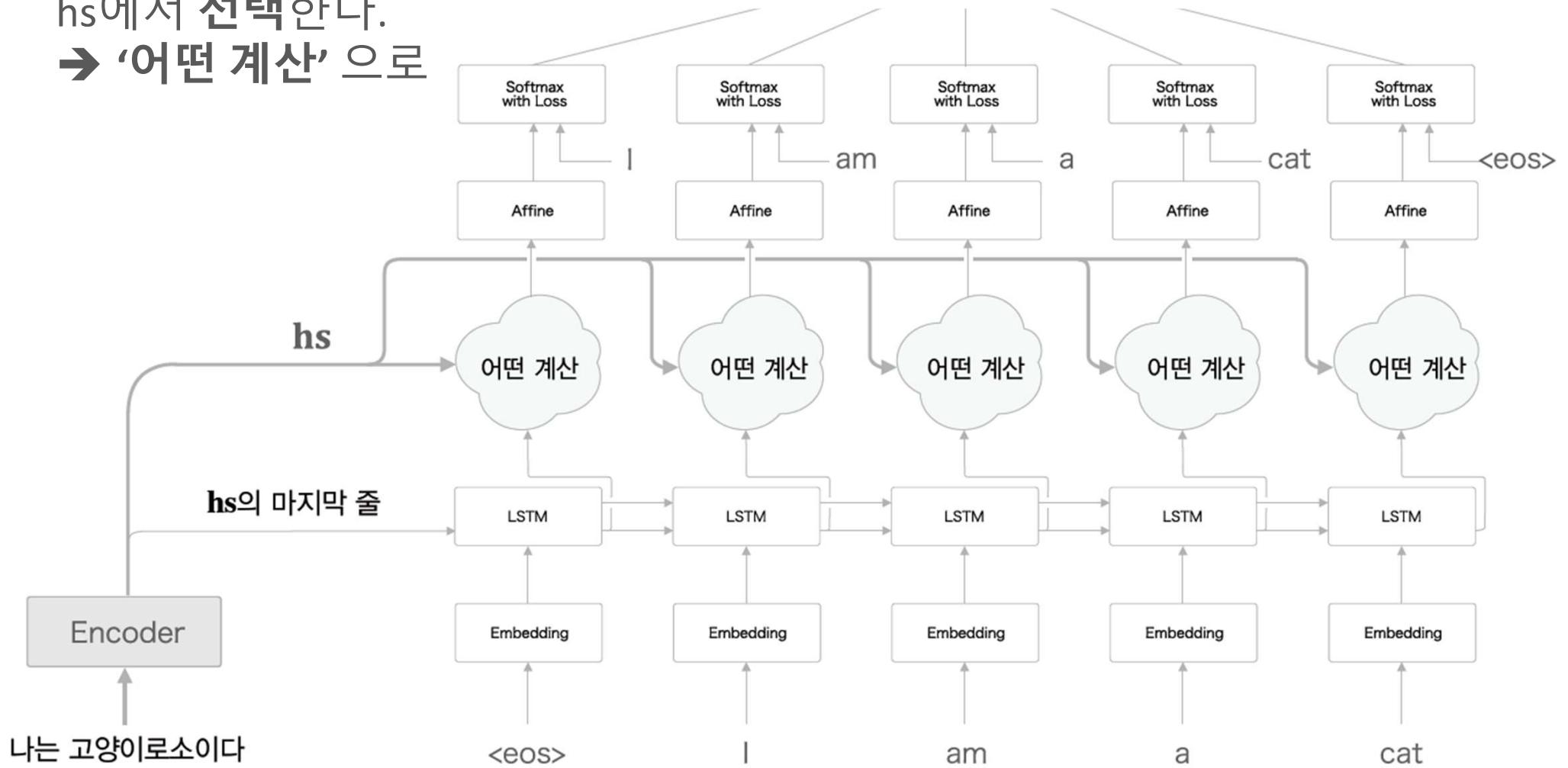


Decoder 개선 목표

- 신경망 목표 : 단어들의 얼라인먼트 추출

즉, 각 시간에서 Decoder에 입력된 단어와 대응관계인 단어의 벡터를 hs에서 선택한다.

→ ‘어떤 계산’으로



Decoder 개선 1

- 신경망 목표 : 단어들의 얼라인먼트 추출
즉, 각 시간에서 Decoder에 입력된 단어와 대응관계인 단어의 벡터를
hs에서 선택한다.
→ ‘어떤 계산’으로

Q) ‘선택’이라는 연산이 미분 가능한가?
Why? 오차 역전파로 학습할 때, 미분 가능한 연산자여야 학습 가능

Decoder 개선 1

- 신경망 목표 : 단어들의 얼라인먼트 추출
즉, 각 시간에서 Decoder에 입력된 단어와 대응관계인 단어의 벡터를 hs 에서 선택한다.
→ ‘어떤 계산’으로

Q) ‘선택’이라는 연산이 미분 가능한가?

Why? 오차 역전파로 학습할 때, 미분 가능한 연산자여야 학습 가능

아이디어)

hs 에서 하나만 선택하는 것이 아니라, 모든 것을 선택한다.

How? 각 단어의 중요도를 나타내는 가중치를 이용하여 모두를 선택

Decoder 개선 1

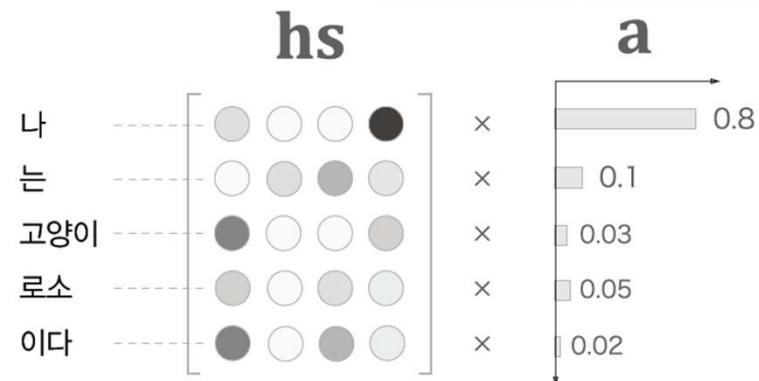
- 신경망 목표 : 단어들의 얼라인먼트 추출
 즉, 각 시간에서 Decoder에 입력된 단어와 대응관계인 단어의 벡터를 hs에서 선택한다.
 → ‘어떤 계산’으로

Q) ‘선택’이라는 연산이 미분 가능한가?
 Why? 오차 역전파로 학습할 때, 미분 가능한 연산자여야 학습 가능

아이디어)

hs에서 하나만 선택하는 것이 아니라, 모든 것을 선택한다.

How? 각 단어의 중요도를 나타내는 가중치를 이용하여 모두를 선택

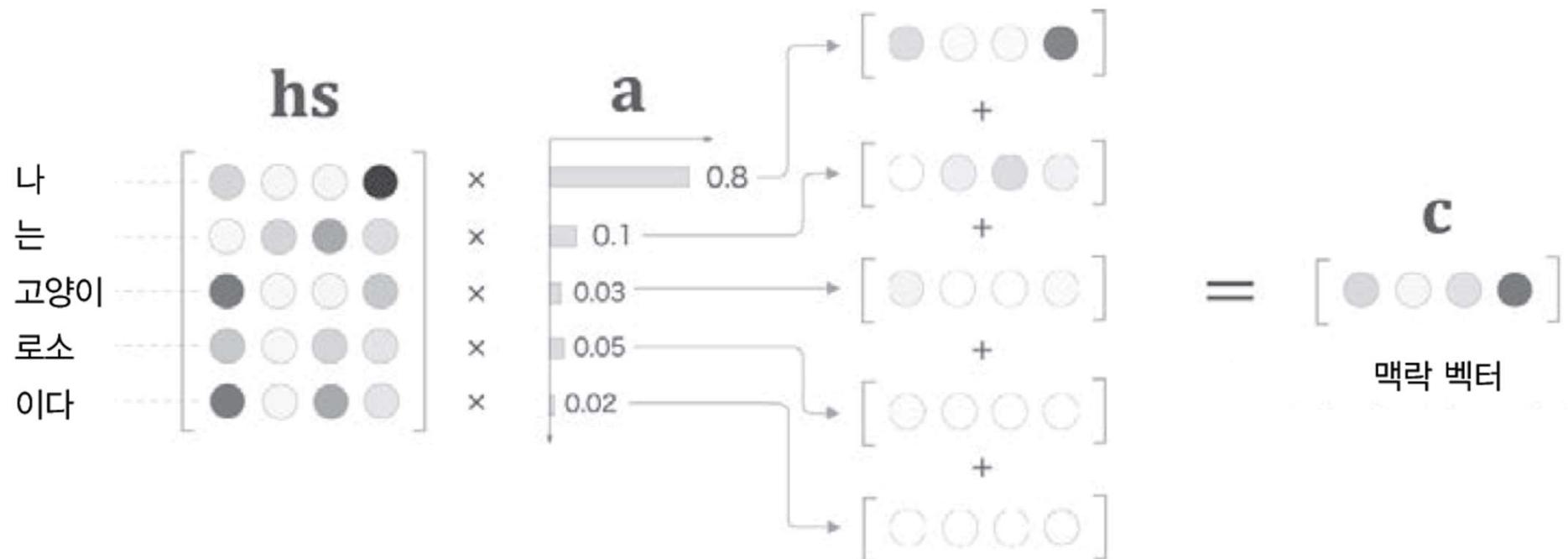


Decoder 개선 1_Weight Sum

- Weight Sum

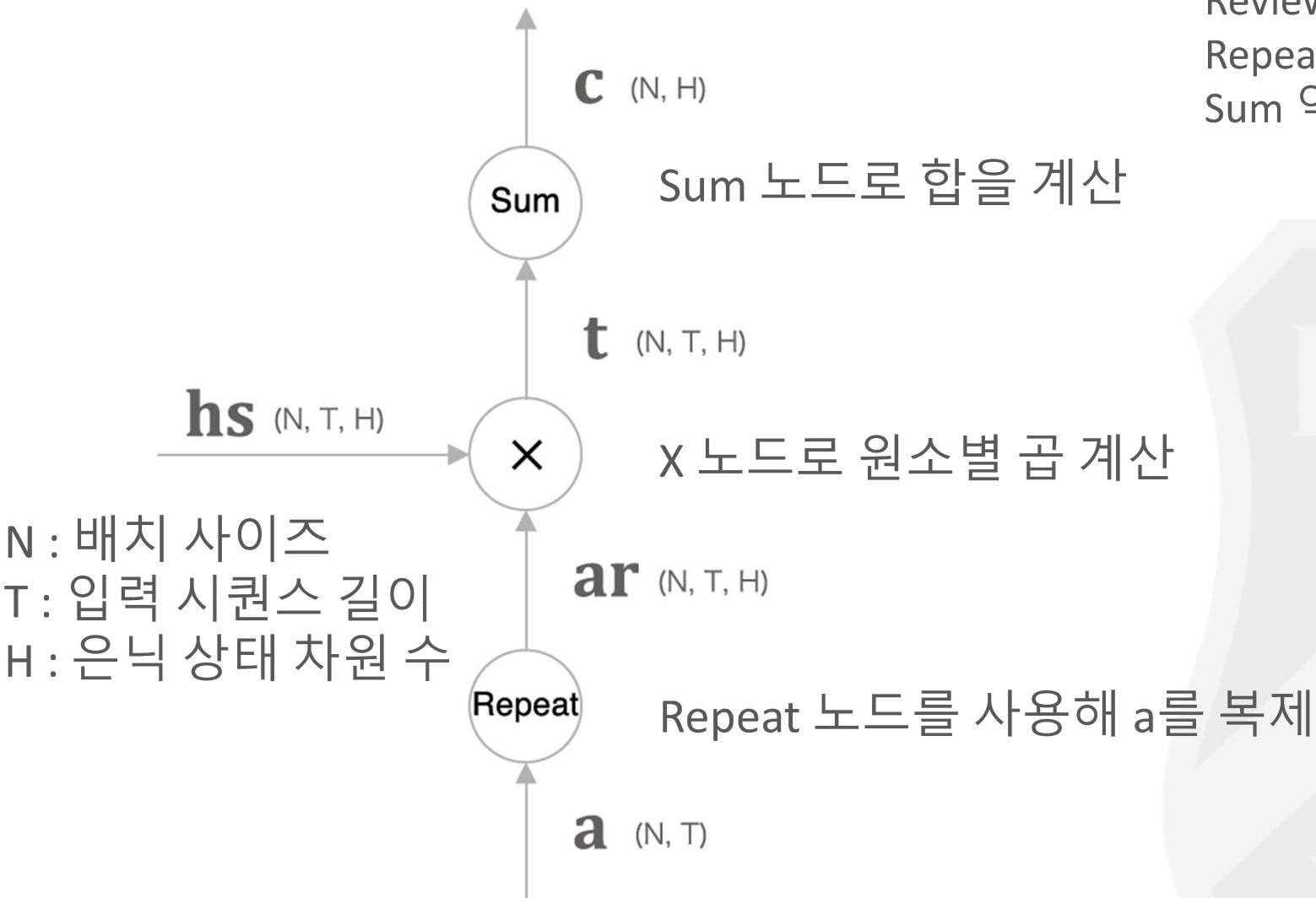
: 각 단어의 중요도를 나타나는 가중치(a)를 이용하여 각 단어의 벡터 hs로 부터 가중합을 계산하는 방식으로 맥락벡터(c)를 구하는 계층

a : 각 단어에 대해 그것이 현재 얼마나 중요한지를 나타내는 가중치
 이는 확률 분포처럼, 원소는 0.0 ~ 1.0 사이의 스칼라이며 총합은 1



Decoder 개선 1_Weight Sum

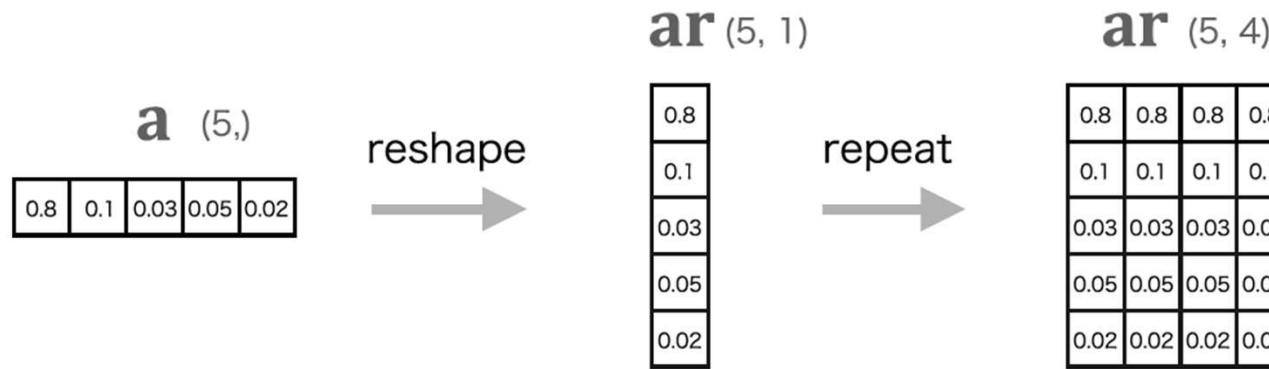
- 선택 작업 계층의 계산 그래프



Review)
 Repeat 역전파 – Sum
 Sum 역전파 – Repeat

Decoder 개선 1_Weight Sum

- 선택 작업 계층의 계산 그래프 (Detail)



```

import sys
sys.path.append('..')
from common.np import * # import numpy as np
from common.layers import Softmax

class WeightSum:
    def __init__(self):
        self.params, self.grads = [], []
        self.cache = None

    def forward(self, hs, a):
        N, T, H = hs.shape

        ar = a.reshape(N, T, 1).repeat(T, axis=1)
        t = hs * ar
        c = np.sum(t, axis=1)

        self.cache = (hs, ar)
        return c

    def backward(self, dc):
        hs, ar = self.cache
        N, T, H = hs.shape
        dt = dc.reshape(N, 1, H).repeat(T, axis=1)
        dar = dt * hs
        dhs = dt * ar
        da = np.sum(dar, axis=2)

        return dhs, da
  
```

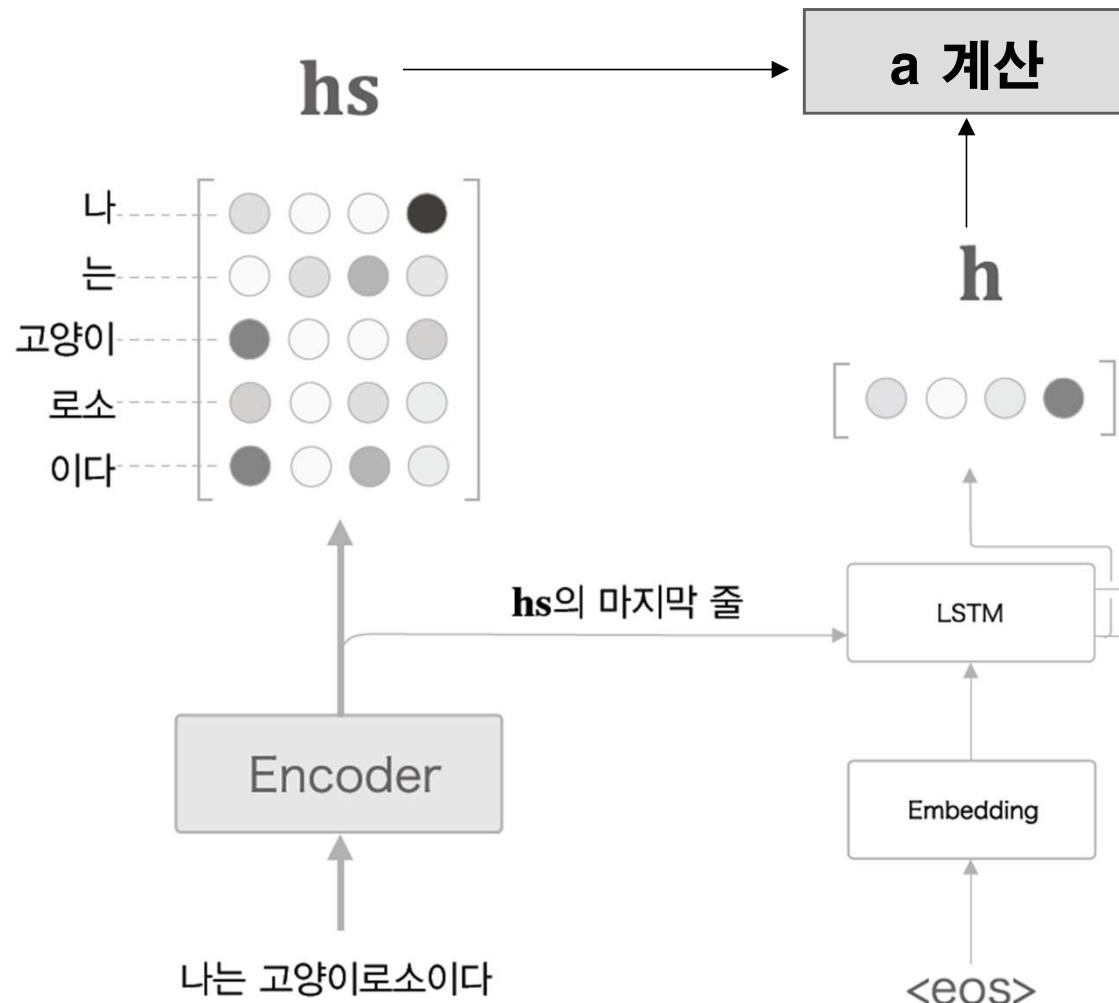
Decoder 개선 2

- Weight Sum
 - : 각 단어의 중요도를 나타나는 **가중치(a)**를 이용하여 각 단어의 벡터 h_s 로 부터 가중합을 계산하는 방식으로 맥락벡터(c)를 구하는 계층
- a : 각 단어에 대해 그것이 현재 얼마나 중요한지를 나타내는 가중치
이는 확률 분포처럼, 원소는 0.0 ~ 1.0 사이의 스칼라이며 총합은 1

Decoder 개선2에서 다뤄야 하는 내용 :
가중치 a 를 (자동으로) 구하는 방법

Decoder 개선 2_가중치 a 계산 계층

- 목표 : 가중치 a를 구하기



입력 : hs, h
출력 : 가중치 a

목표 : h가 hs에서의 각 은닉 상태와 얼마나 **비슷한지를** 수치로 표현하는 가중치 a를 만들어야 한다.

Decoder 개선 2_가중치 a 계산 계층

- 목표 : 가중치 a를 구하기

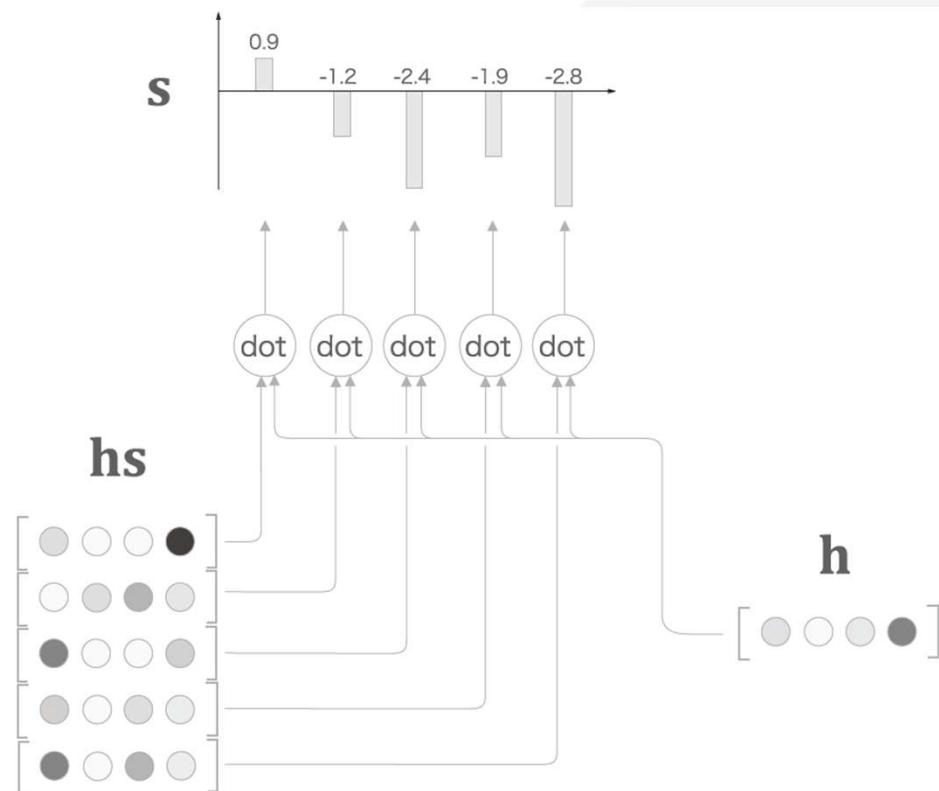
입력 : hs, h

출력 : 가중치 a

목표 : h 가 hs 에서의 각 은닉 상태와 얼마나 **비슷한지를** 수치로 표현하는 가중치 a를 만들어야 한다.

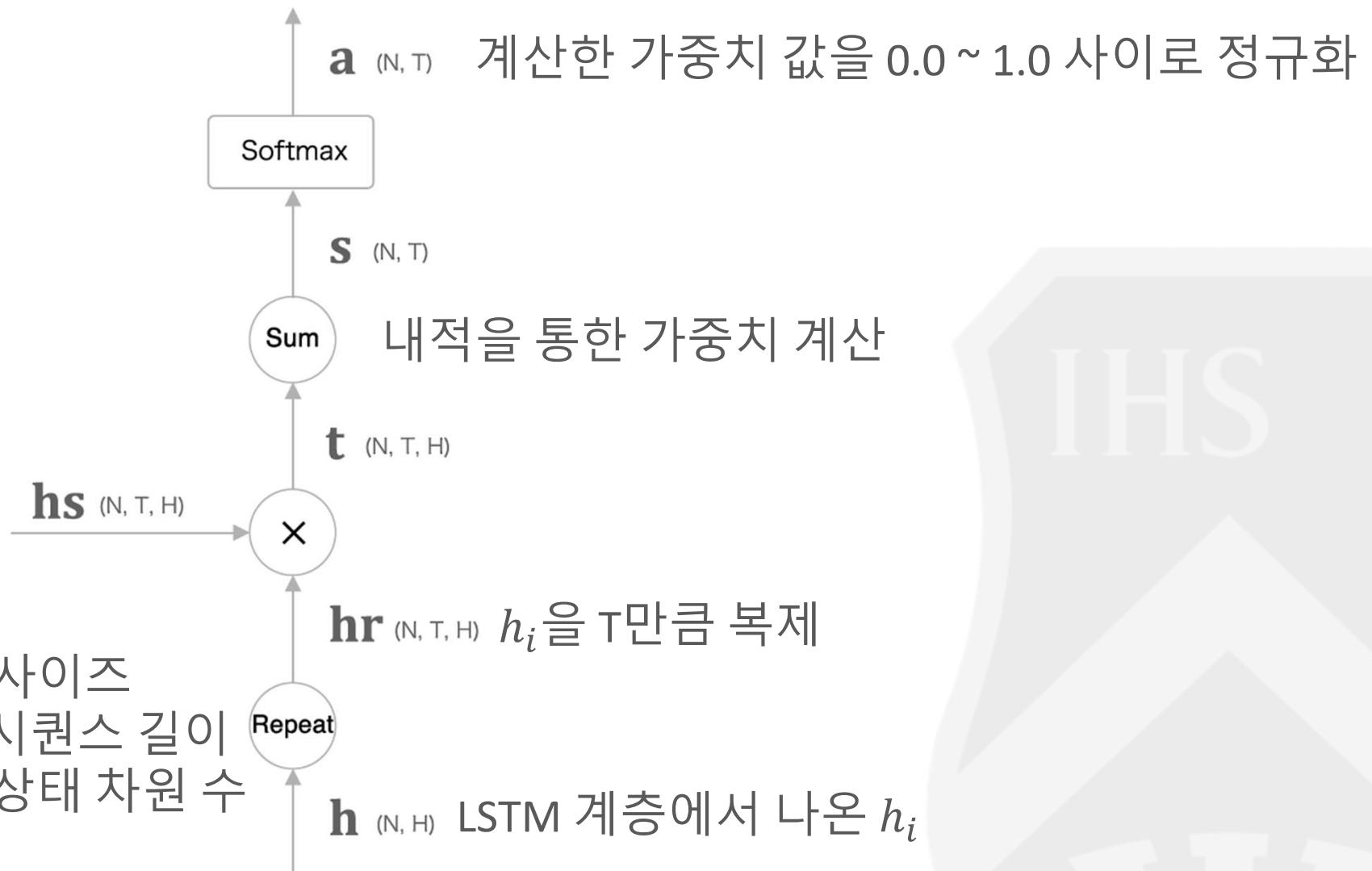
얼마나 비슷한가?
 → Similarity Measure

여기서는, 벡터의 내적을 이용



Decoder 개선 2_가중치 a 계산 계층

- 가중치 a 계산 계층의 계산 그래프



Decoder 개선 3

- Decoder 개선 기법의 결합
: 앞의 두 Decoder 개선 기법을 결합하고 기존 Decoder를 계승하여 마무리

맥락 벡터를 구하는 계산 그래프 :

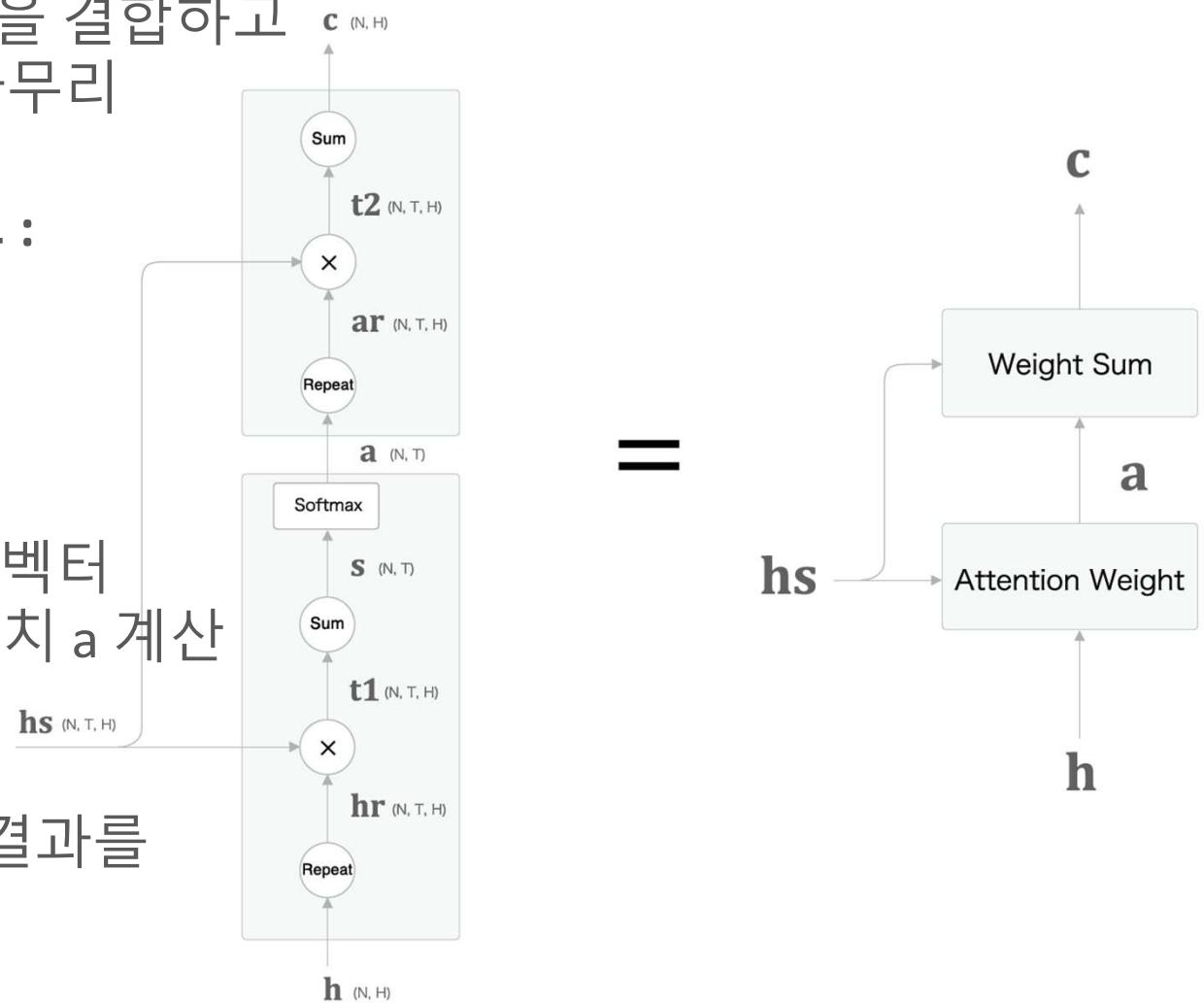
계산의 두 계층

1. Attention Weight 계층 (개선2)

- Encoder가 출력하는 각 단어의 벡터 hs 에 주목하여 해당 단어의 가중치 a 계산

2. Weight Sum 계층 (개선1)

- A 와 hs 의 가중합을 구하고, 그 결과를 맥락벡터 c 로 출력



Decoder 개선 3

- Decoder 개선 기법의 결합
 - : 앞의 두 Decoder 개선 기법을 결합하고 기존 Decoder를 계승하여 마무리

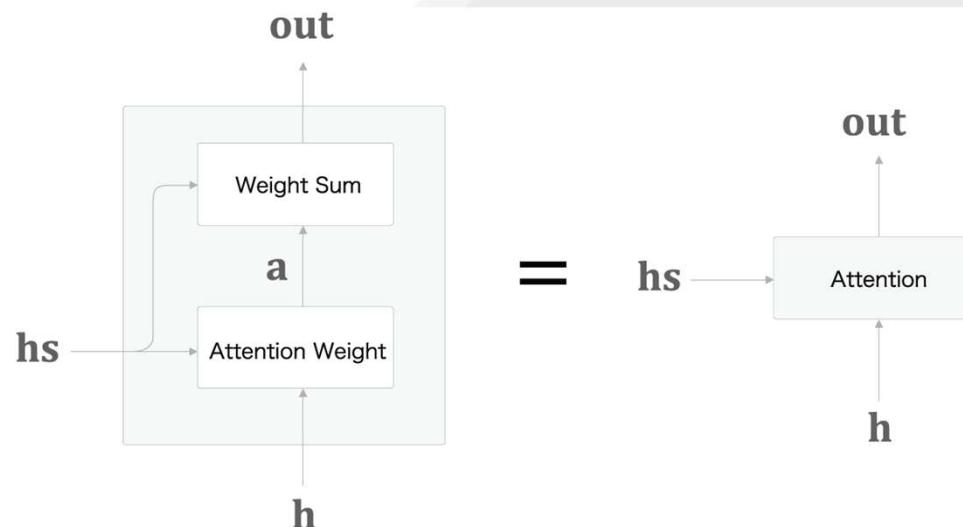
계산의 두 계층 -> Attention 계층으로 결합

1. Attention Weight 계층

- Encoder가 출력하는 각 단어의 벡터 hs 에 주목하여 해당 단어의 가중치 a 계산

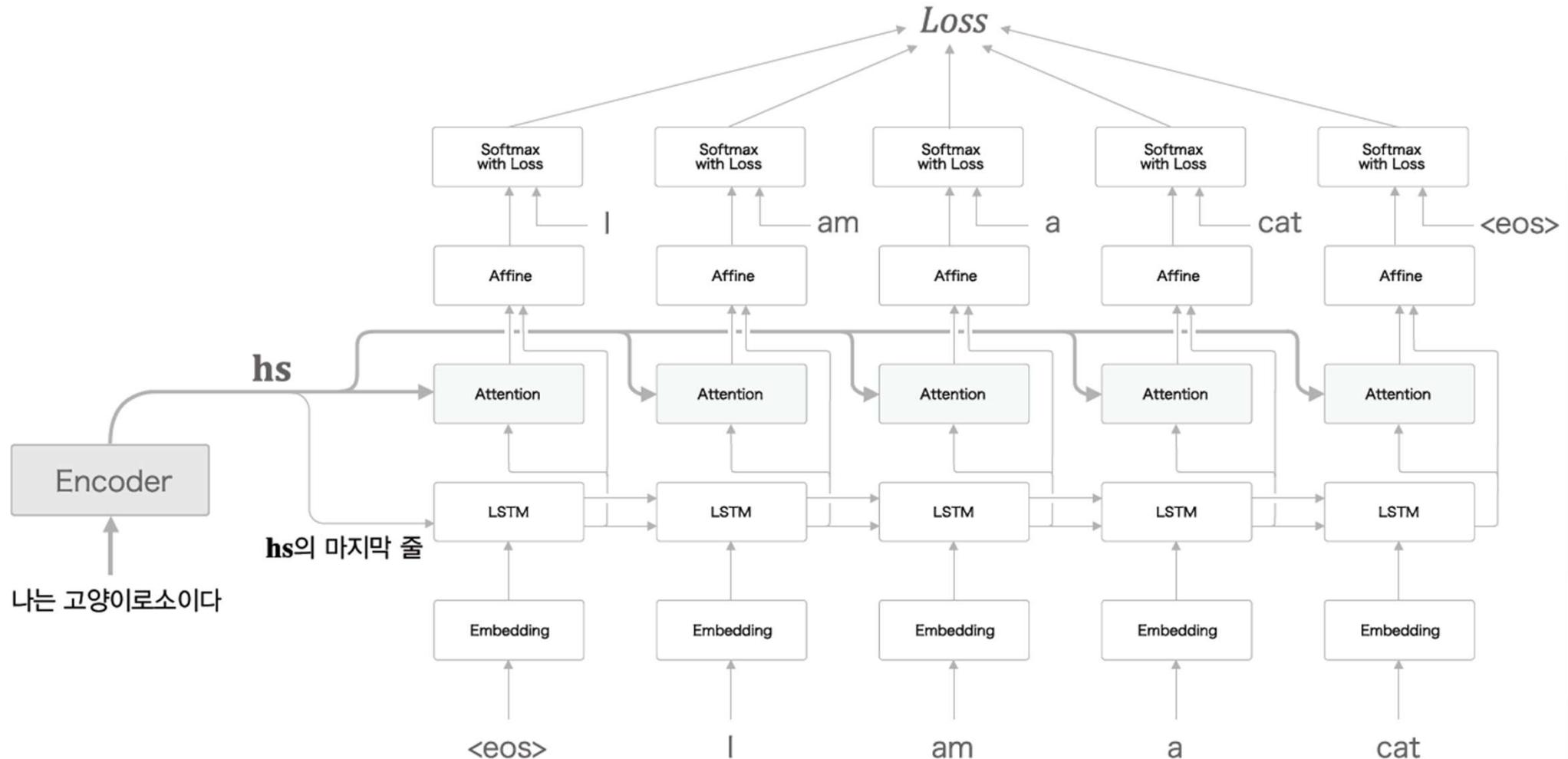
2. Weight Sum 계층

- A 와 hs 의 가중합을 구하고, 그 결과를 맥락벡터 c 로 출력



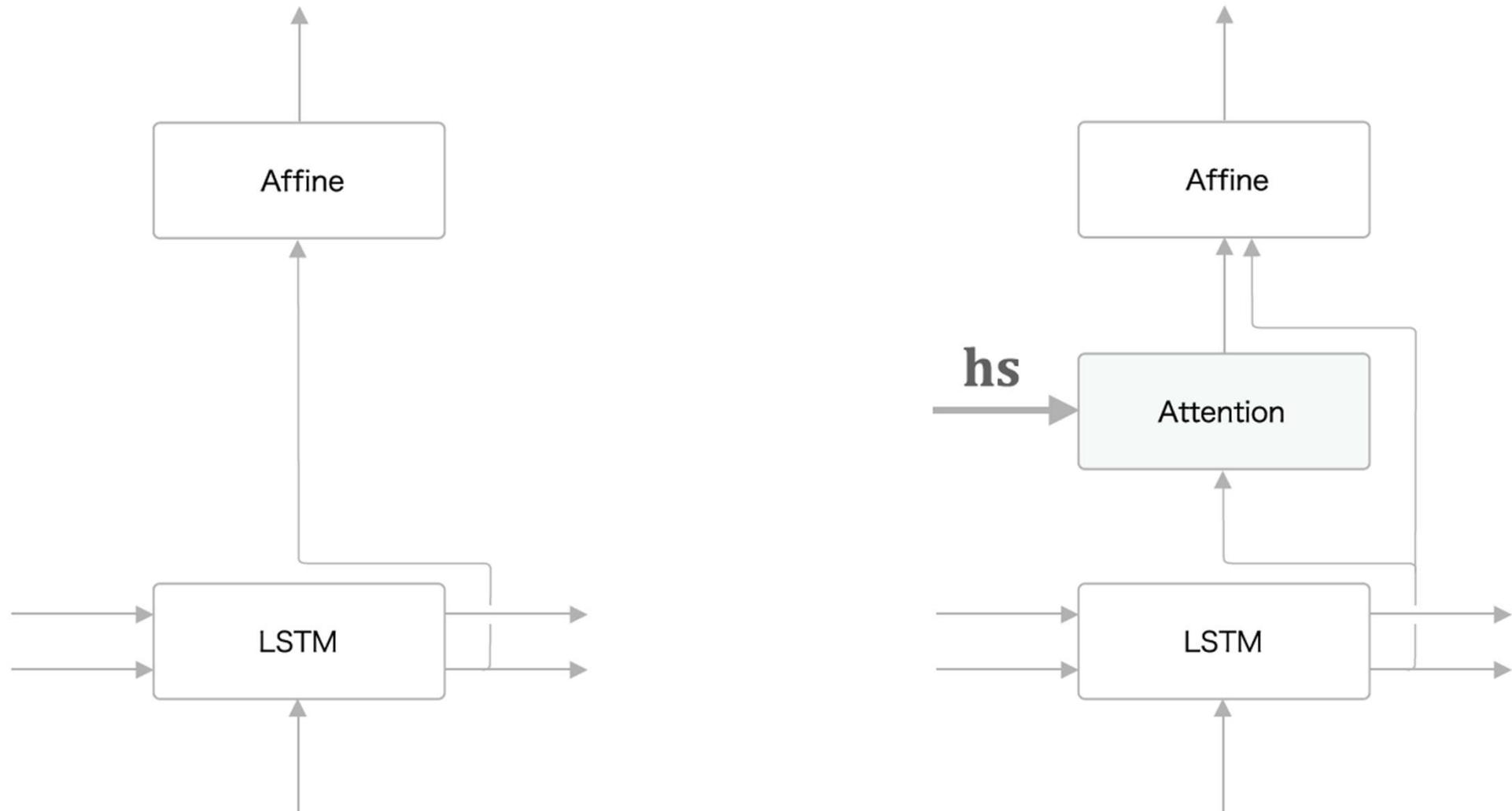
Decoder 개선 구현

- Attention 계층을 갖춘 Decoder 계층 구성
- Encoder가 건네 주는 정보 hs 에서 중요한 원소에 주목 이를 바탕으로 맥락 벡터를 구해 위쪽 계층으로 전파



Decoder 개선 구현

- 기존 Decoder(왼쪽) 와 Attention이 추가된 Decoder(오른쪽) 비교



어텐션을 갖춘 seq2seq 구현

- 구현 방법 (seq2seq 계승)
 - 7장에서 구현한 3개의 클래스 (Encoder, Decoder, seq2seq) 대신, AttentionEncoder, AttentionDecoder, Attentionseq2seq 클래스를 구현하고 해당 클래스로 치환만 하면 된다.



AttentionEncoder 구현

- Encoder와 AttentionEncoder 차이:
: Encoder의 forward()클래스는 LSTM 마지막 계층만 반환
반면, 이번에는 모든 은닉 상태를 반환

```
import sys
sys.path.append('..')
from common.time_layers import *
from ch07.seq2seq import Encoder, Seq2seq
from ch08.attention_layer import TimeAttention

class AttentionEncoder(Encoder):
    def forward(self, xs):
        xs = self.embed.forward(xs)
        hs = self.lstm.forward(xs)
        return hs

    def backward(self, dhs):
        dout = self.lstm.backward(dhs)
        dout = self.embed.backward(dout)
        return dout
```

AttentionDecoder 구현

- Decoder와 AttentionDecoder 차이
 - : Time Attention 계층이 새롭게 Decoder 클래스에 사용
이는, Forward()에서 Time Attention 계층의 출력과
LSTM 계층의 출력을 연결하고, 이 때 np.concatenate() 메서드를 사용

```
class AttentionDecoder:
    def __init__(self, vocab_size, wordvec_size, hidden_size):
        V, D, H = vocab_size, wordvec_size, hidden_size
        rn = np.random.randn

        embed_W = (rn(V, D) / 100).astype('f')
        lstm_Wx = (rn(D, 4 * H) / np.sqrt(D)).astype('f')
        lstm_Wh = (rn(H, 4 * H) / np.sqrt(H)).astype('f')
        lstm_b = np.zeros(4 * H).astype('f')
        affine_W = (rn(2*H, V) / np.sqrt(2*H)).astype('f')
        affine_b = np.zeros(V).astype('f')

        self.embed = TimeEmbedding(embed_W)
        self.lstm = TimeLSTM(lstm_Wx, lstm_Wh, lstm_b, stateful=True)
        self.attention = TimeAttention()
        self.affine = TimeAffine(affine_W, affine_b)
        layers = [self.embed, self.lstm, self.attention, self.affine]

        self.params, self.grads = [], []
        for layer in layers:
            self.params += layer.params
            self.grads += layer.grads
```

```
def forward(self, xs, enc_hs):
    h = enc_hs[:, -1]
    self.lstm.set_state(h)

    out = self.embed.forward(xs)
    dec_hs = self.lstm.forward(out)
    c = self.attention.forward(enc_hs, dec_hs)
    out = np.concatenate((c, dec_hs), axis=2)
    score = self.affine.forward(out)

    return score
```

AttentionSeq2seq 구현

- AttentionSeq2seq 클래스 구현

: 앞 장의 seq2seq 클래스를 상속하고, 초기화 메서드를 수정하는 것만으로 AttentionSeq2seq를 구현 가능

```
class AttentionSeq2seq(Seq2seq):  
    def __init__(self, vocab_size, wordvec_size, hidden_size):  
        args = vocab_size, wordvec_size, hidden_size  
        self.encoder = AttentionEncoder(*args)  
        self.decoder = AttentionDecoder(*args)  
        self.softmax = TimeSoftmaxWithLoss()  
  
        self.params = self.encoder.params + self.decoder.params  
        self.grads = self.encoder.grads + self.decoder.grads
```

Attention 응용

- 셀프 어텐션
- 트랜스포머

... 11/19일 : 트랜스포머 아키텍쳐

그림 8-38 트랜스포머의 계층 구조(문헌 [52]를 참고로 단순화한 모델)

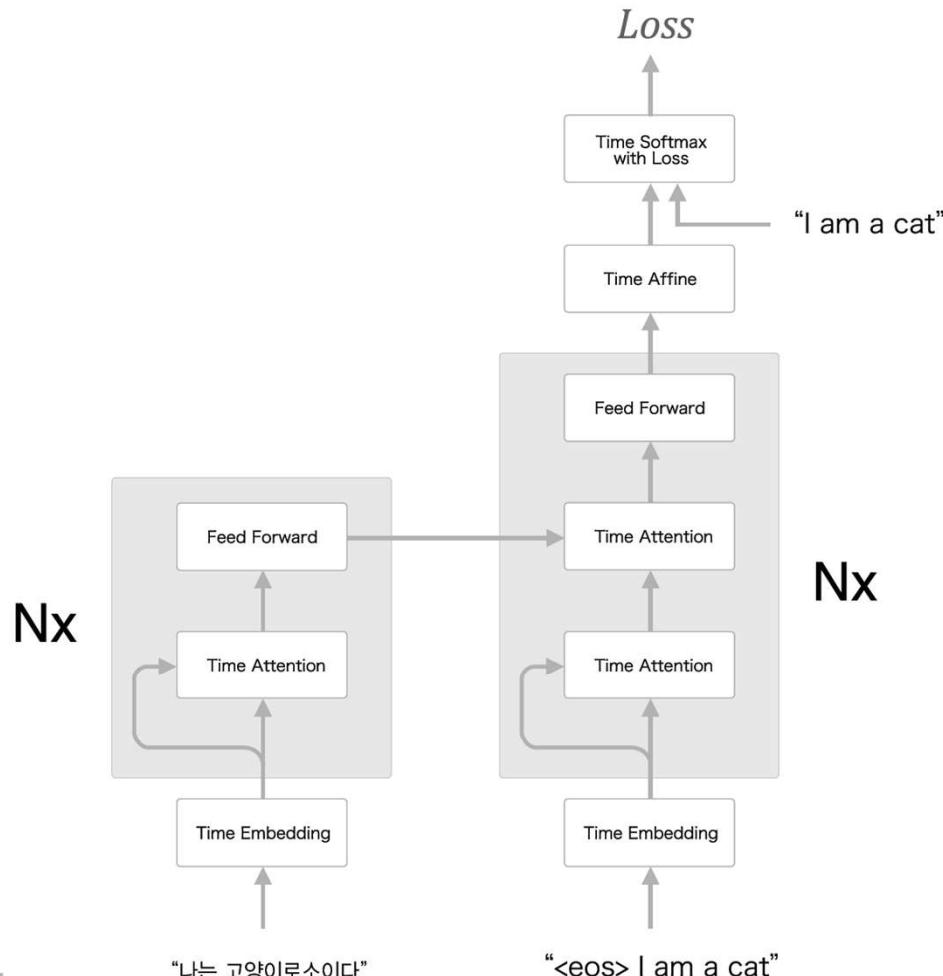


그림 8-37 왼쪽이 일반적인 어텐션, 오른쪽이 셀프어텐션

