
Chapter8

Attention

Soojung Kim

Index

1. Structure of attention
 - seq2seq 문제점
 - Encoder 개선
 - Decoder 개선
2. Implement (jupyter notebook)
3. Evaluation
 - visualization
4. Etc
 - Bi-directional RNN
 - Use of attention layer
 - Skip Connection
5. Applications
 - Google Neural Machine Translation
 - Transformer
 - Neural Turning Machine

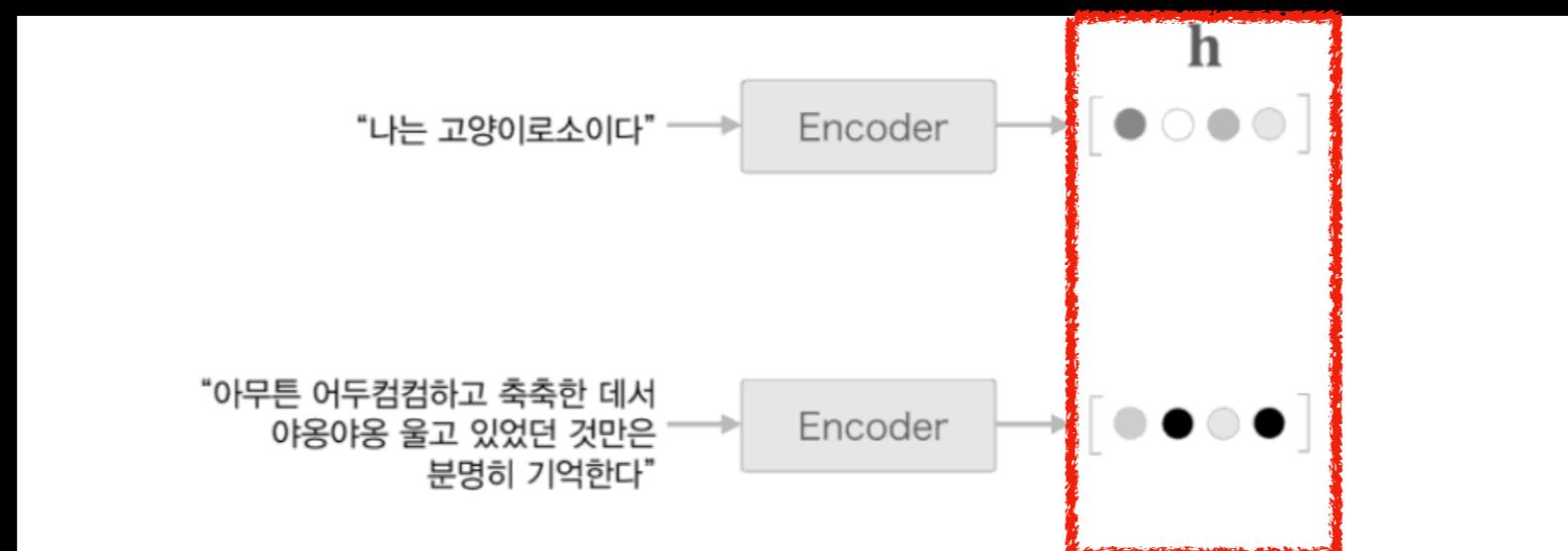
Thanks to Attention,

seq2seq의 문제점을 해결할 수 있다.
seq2seq은 필요한 정보에만 주목할 수 있다.

Structure of attention

seq2seq의 문제점 : 고정 길이의 벡터

문장이 길던, 짧던 간에 고정된 길이의 벡터로 표현
→ 필요한 정보가 벡터에 다 담기지 못하는 문제점



Encoder, Decoder 개선 필요

Encoder개선

Encoder의 시각별(단어별) LSTM 계층의 은닉 상태를 모두 이용

Decoder개선

Encoder 개선

기준

LSTM 계층의 마지막 은닉 상태만 Decoder에 전달
Keras, return_sequenced=False

개선

Encoder의 시각별(단어별) LSTM 계층의 은닉 상태를 모두 이용
Keras, return_sequenced=True



hs matrix는 각 단어에 해당하는 벡터들의 집합
- 단어 벡터 : 입력된 각 단어에 대한 정보가 많이 포함

단방향 LSTM일 때, '고양이'를 기준으로 앞의 단어들의 정보만 담겨 있음.

그러나,

정보를 균형있게 담아야할 때 즉, 고양이의 주변 단어와의 정보를 균형있게 담고 싶을 때는 양방향 RNN(bi-directional RNN)

Decoder 개선1

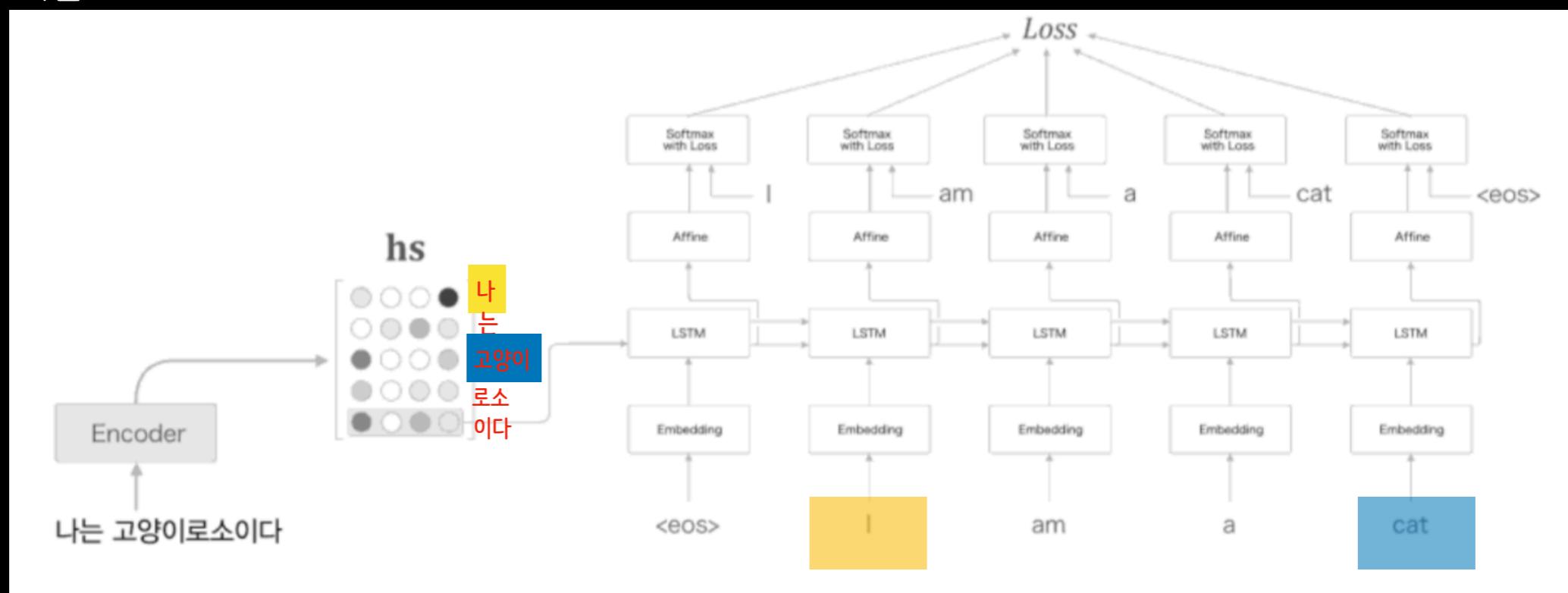
기존

LSTM 계층의 마지막 은닉 상태만 Decoder에 전달
hs에서 마지막 줄만 Decoder에 전달

개선

hs matrix 전부 활용
Alignment 추출

기존



hs matrix 전부 활용하기 앞서서

입력과 출력의 여러 단어 중 어떤 단어끼리 서로 관련되어 있는가에 대한 대응 관계를 나타내는 정보(alignment)를 학습 즉, 필요한 정보에만 주목하여 hs를 활용하는 것이 핵심

Decoder 개선1

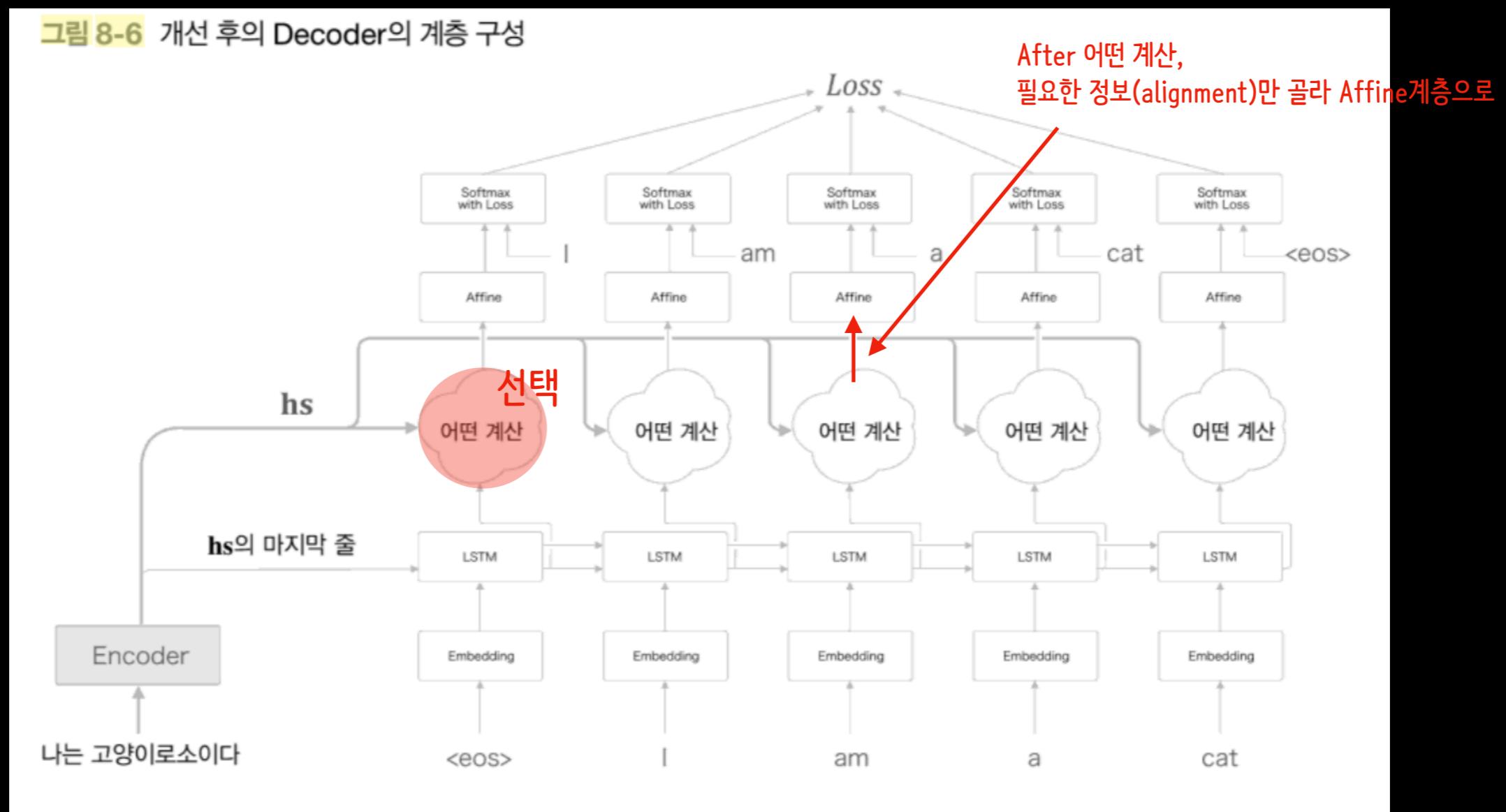
기준

LSTM 계층의 마지막 은닉 상태만 Decoder에 전달
hs에서 마지막 줄만 Decoder에 전달

개선

hs matrix 전부 활용 Alignment 추출

그림 8-6 개선 후의 Decoder의 계층 구성



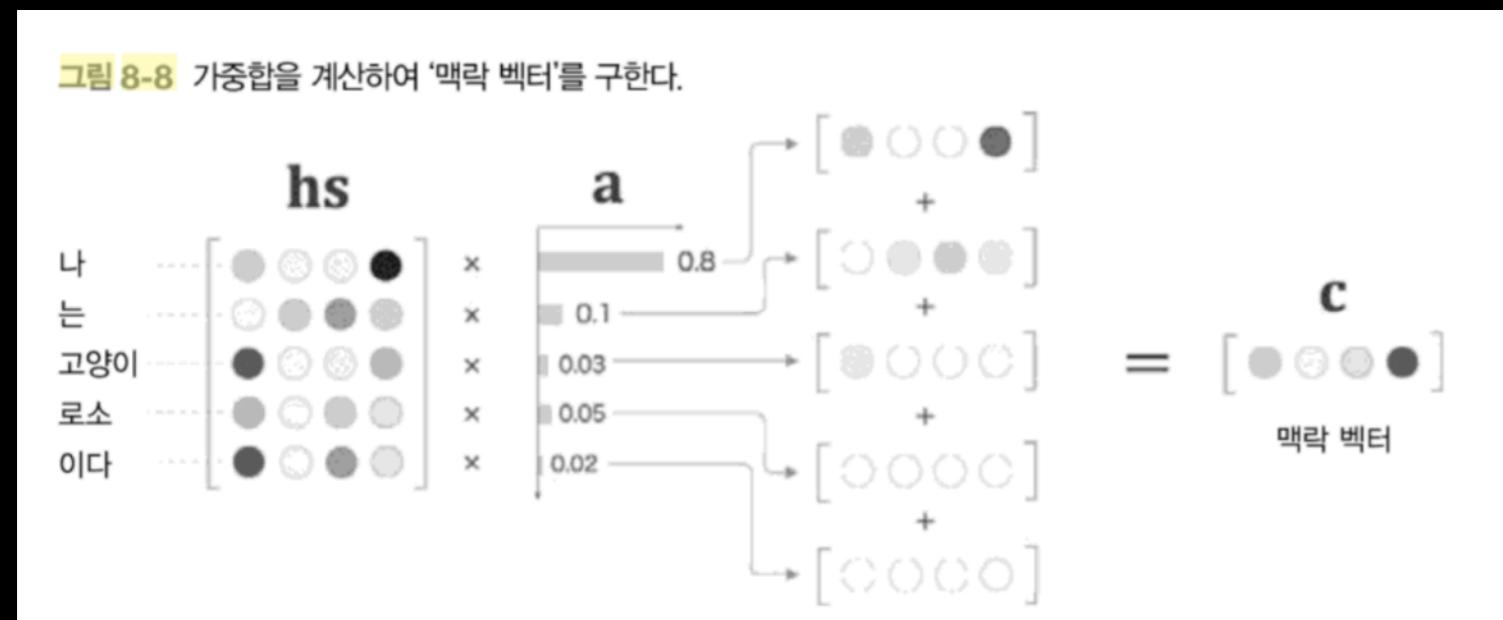
각 시각(t)에서 Decoder에 입력된 단어와 대응 관계인 단어의 벡터를 hs matrix에서 선택

어떤 계산(a.k.a선택)

여기서의 '선택'은 Pick me, Pick me가 아니라, 모든 것을 선택하는 의미.

다만, 모든 것을 선택함에 있어서 중요도(가중치)로 다르게 주어 차별화 함 → 미분 가능하게 하기 위함

* Remind : neural train은 backprop이 가능해야함 → 미분이 가능해야함



a (가중치) : 0.0~1.0 사이의 스칼라(단일 원소), 모든 원소의 합은 1

c (맥락 벡터) : weighted sum(단어의 벡터 hs와 a의 원소별 곱) 의 결과

```
import numpy as np

T, H = 5, 4
hs = np.random.randn(T, H)
a = np.array([0.8, 0.1, 0.03, 0.05, 0.02])

ar = a.reshape(5, 1).repeat(4, axis=1)
print(ar.shape)
# (5, 4)

t = hs * ar
print(t.shape)
# (5, 4)

c = np.sum(t, axis=0)
print(c.shape)
# (4, )
```

Weighted Sum

Weighted sum을 하는 다양한 방법들

- repeat() + sum()
- np.broadcast
- np.matmul()

- 구현 효율을 생각하면 repeat() 보다 브로드 캐스트
- matmaul(a, hs)로 간단하게 할 수 있지만 미니배치로 확장하기에 구현이 쉽지 않다.

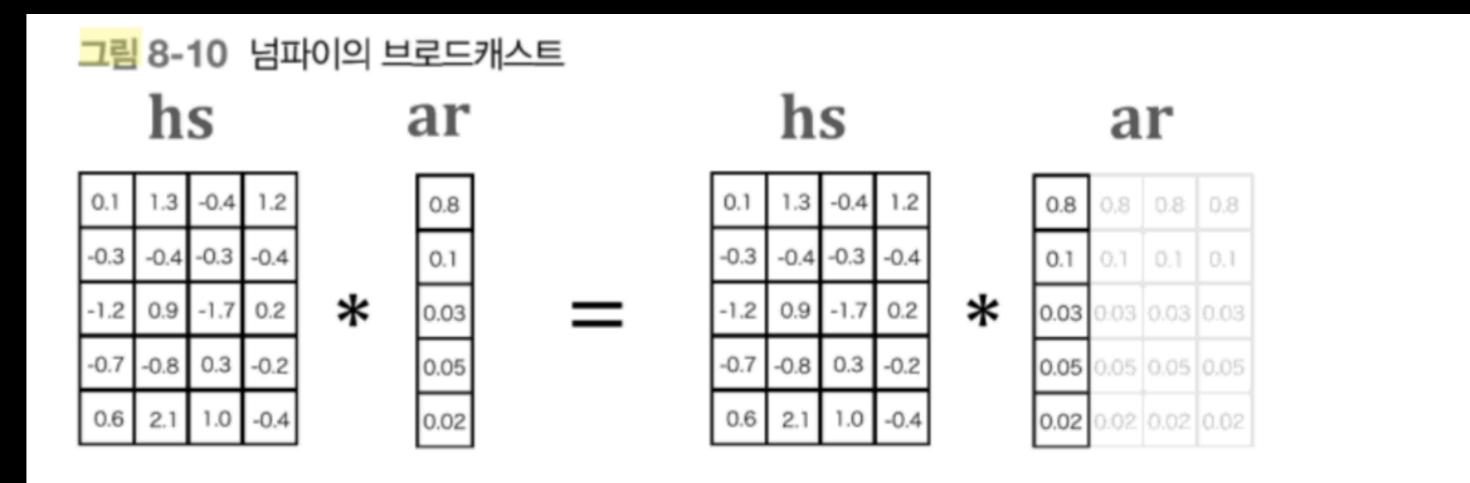
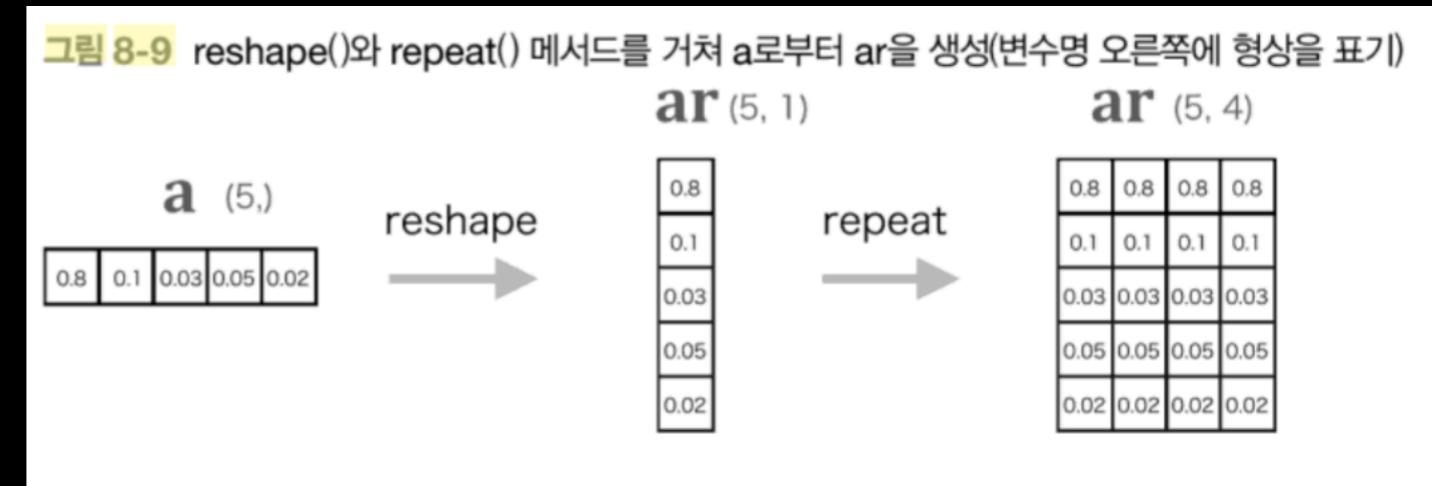
```
import numpy as np

T, H = 5, 4
hs = np.random.randn(T, H)
a = np.array([0.8, 0.1, 0.03, 0.05, 0.02])

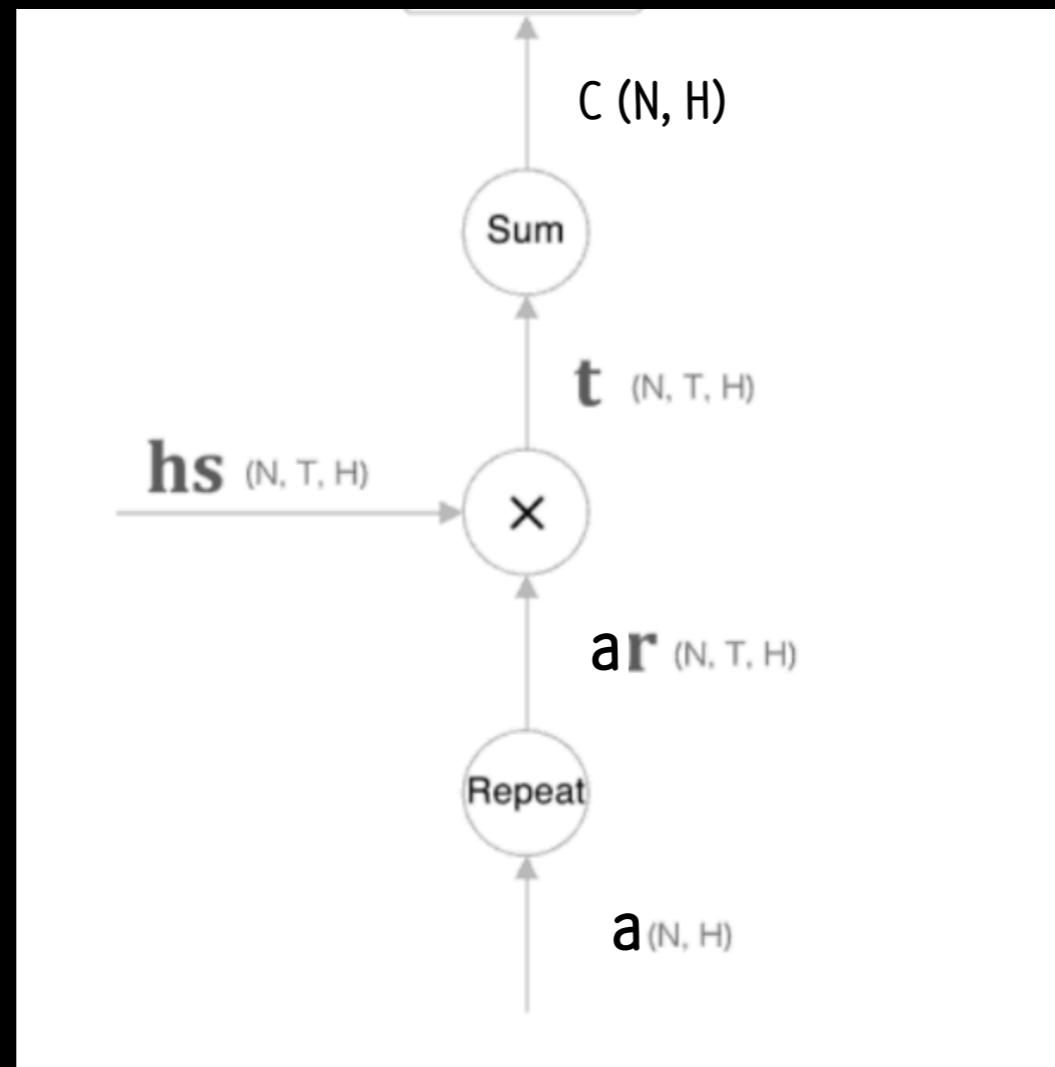
ar = a.reshape(5, 1).repeat(4, axis=1)
print(ar.shape)
# (5, 4)

t = hs * ar
print(t.shape)
# (5, 4)

c = np.sum(t, axis=0)
print(c.shape)
# (4,)
```



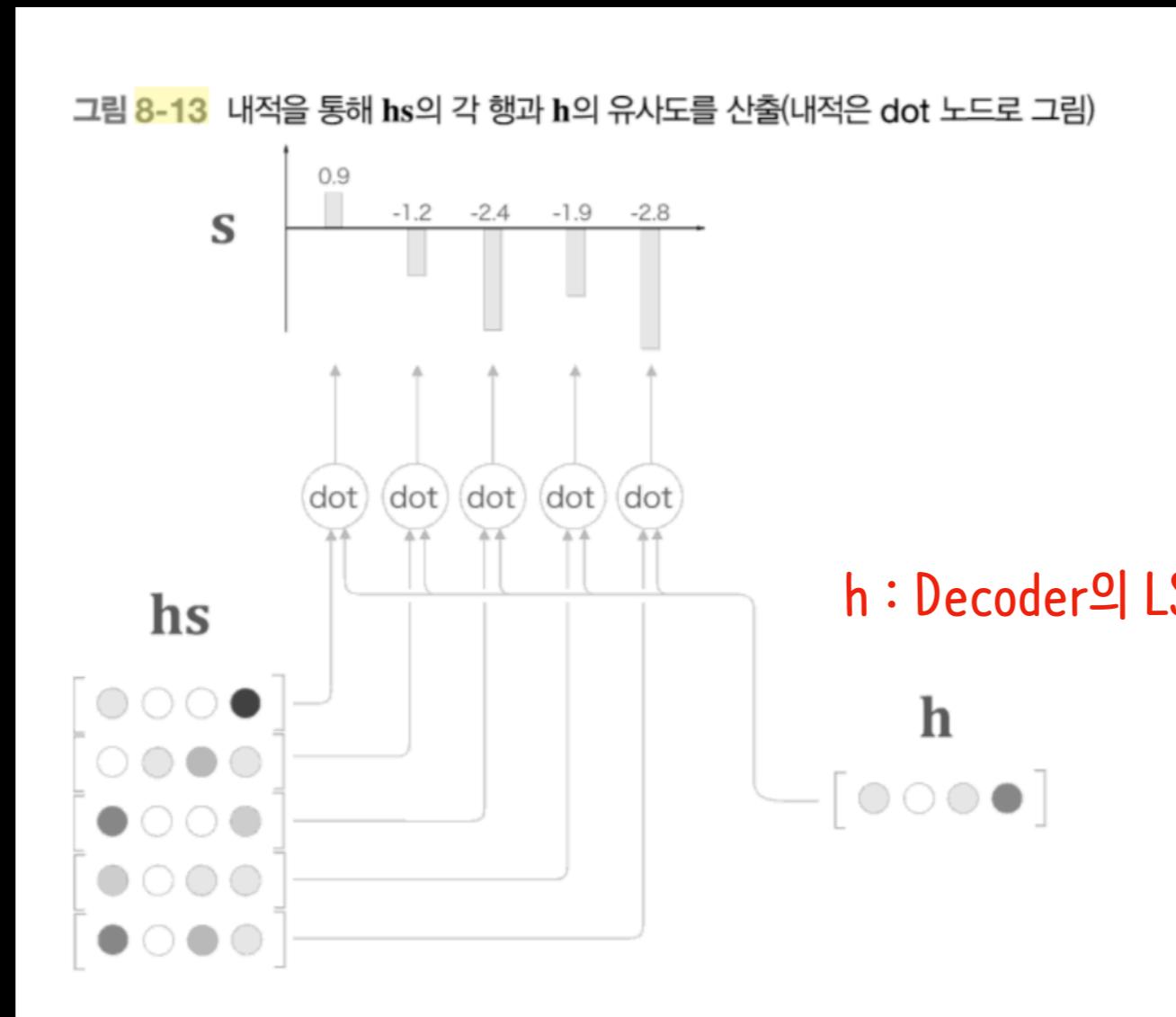
Decoder 개선1 정리



Weight Sum을 구함

Decoder 개선2

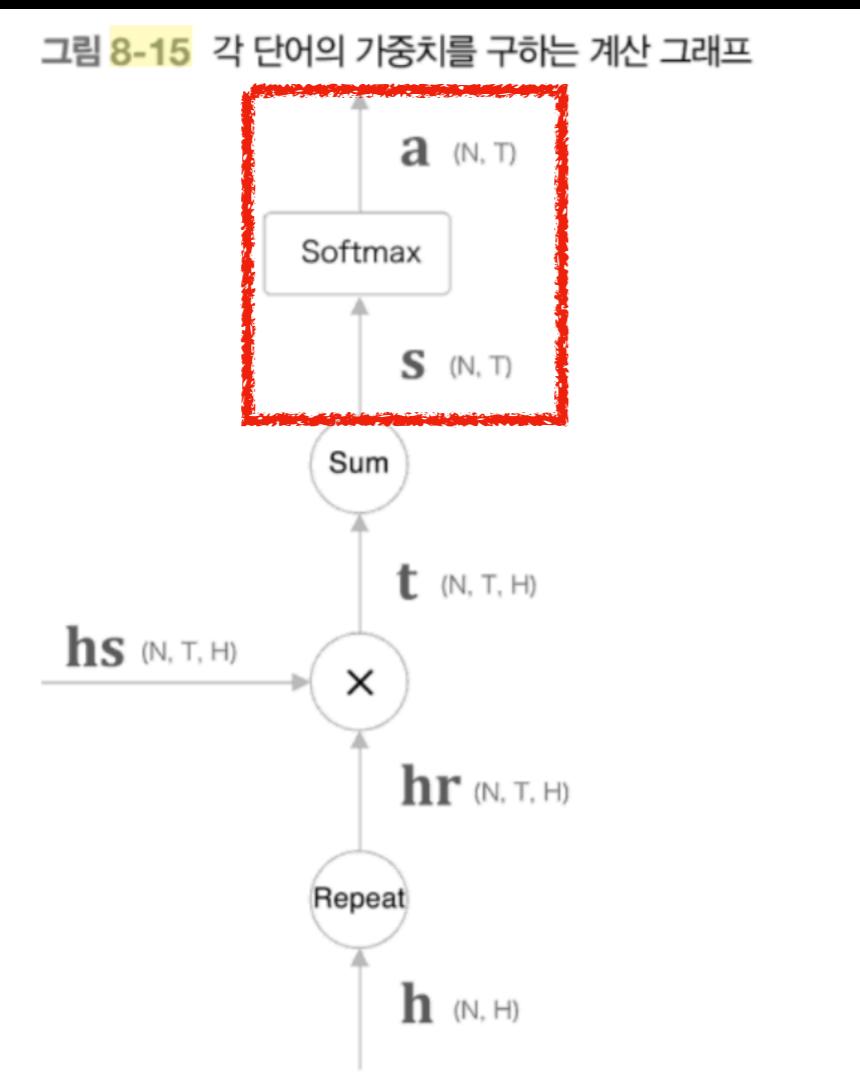
각 단어의 가중치 a 를 구하는 방법



따라서 h 가 hs 의 각 단어 벡터와 얼마나 비슷한가(내적)를 수치로 나타내는 것

Decoder 개선2

각 단어의 가중치 a 를 구하는 방법



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

N, T, H = 10, 5, 4
hs = np.random.randn(N, T, H)
h = np.random.randn(N, H)
hr = h.reshape(N, 1, H).repeat(T, axis=1)
# hr = h.reshape(N, 1, H) # 브로드캐스트를 사용하는 경우

t = hs * hr
print(t.shape)
# (10, 5, 4)

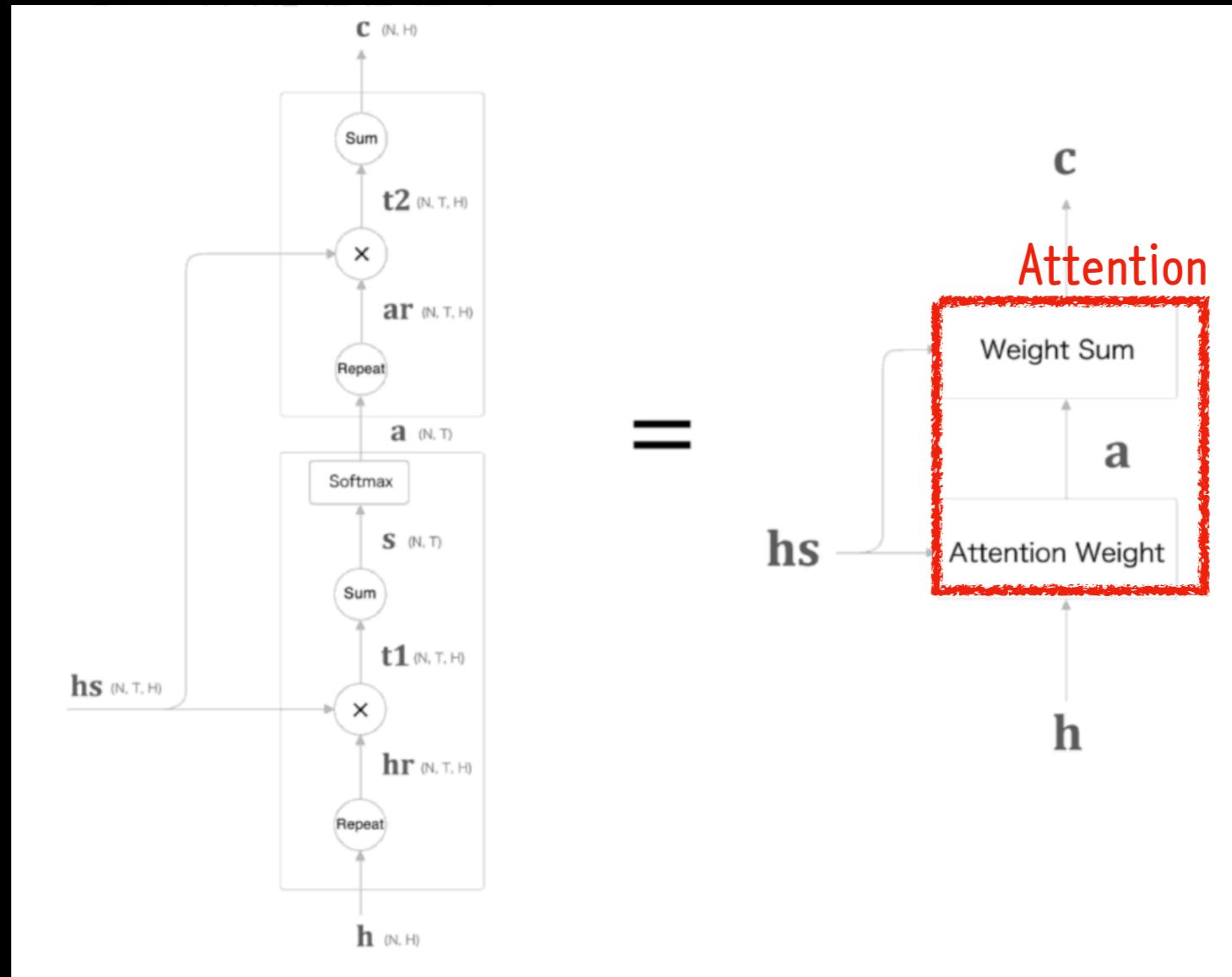
s = np.sum(t, axis=2)
print(s.shape)
# (10, 5)

softmax = Softmax()
a = softmax.forward(s)
print(a.shape)
# (10, 5)
```

softmax를 이용하면 출력값인 a 의 각 원소는 $0.0 \sim 1.0$ 사이의 값이 되고, 모든 원소의 총합은 1. 이렇게 해서 Attention weight를 구함.

Decoder 개선3

Attention = Weighted sum + Attention weight



Attention weight : Encoder가 출력하는 각 단어의 벡터 hs 에 주목하여 weight a 구함
Weighted sum : a 와 hs 의 가중합을 구하고, 맥락 벡터 c 로 출력

어텐션을 갖춘 seq2seq 구현 - Encoder

Diff. 모든 은닉 상태를 반환

Encoder

```
class Encoder:
    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')

        self.embed = TimeEmbedding(embed_W)
        self.lstm = TimeLSTM(lstm_Wx, lstm_Wh, lstm_b, stateful=False)

        self.params = self.embed.params + self.lstm.params
        self.grads = self.embed.grads + self.lstm.grads
        self.hs = None

    def forward(self, xs):
        xs = self.embed.forward(xs)
        hs = self.lstm.forward(xs)
        self.hs = hs
        return hs[:, -1, :]

    def backward(self, dh):
        dhs = np.zeros_like(self.hs)
        dhs[:, -1, :] = dh

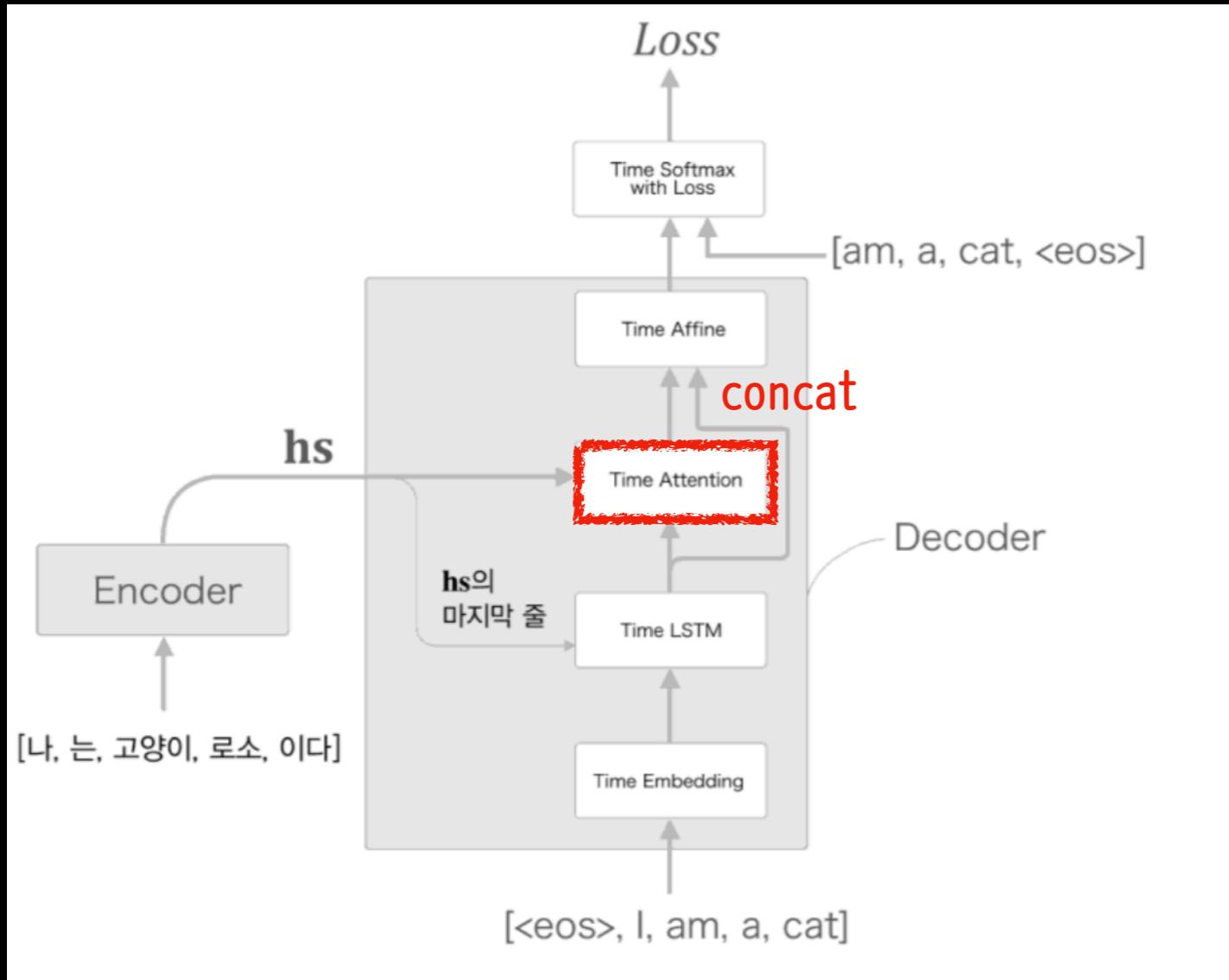
        dout = self.lstm.backward(dhs)
        dout = self.embed.backward(dout)
        return dout
```

AttentionEncoder(Encoder)

```
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
```

어텐션을 갖춘 seq2seq 구현 - Decoder



```
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]
```

```
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
```

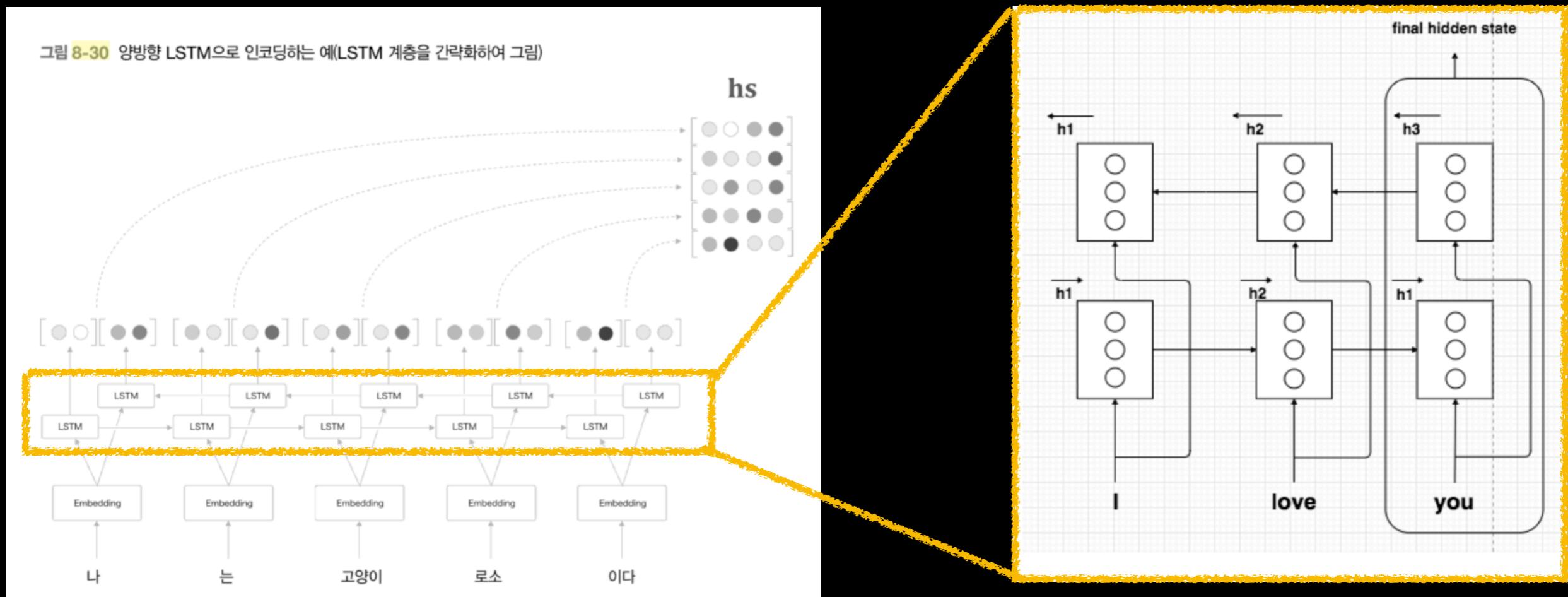
Evaluation

jupyter notebook

Bi-directional RNN

bi-directional LSTM = concat(정방향 LSTM , 역방향 LSTM)

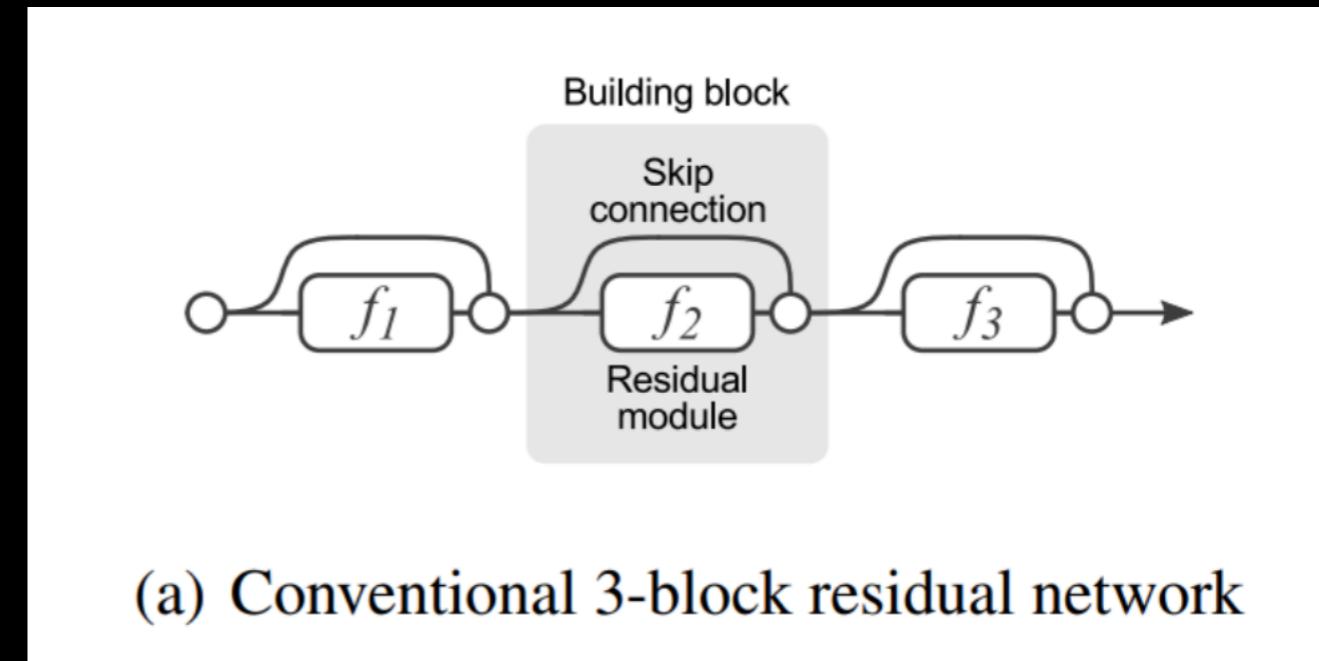
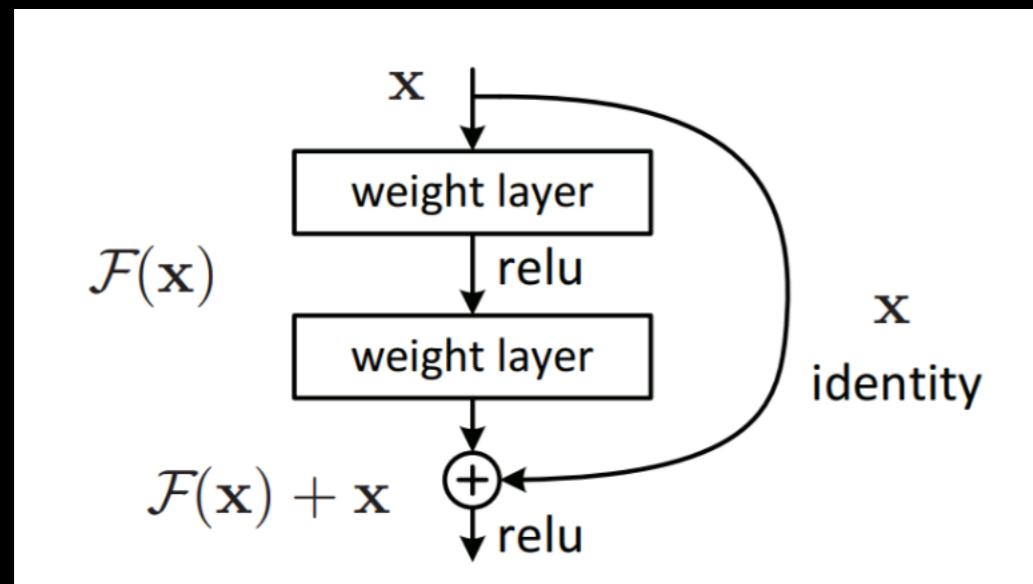
-> 왼쪽, 오른쪽 방향의 정보를 concat하여 균형 잡힌 정보를 인코딩하는 것이 목적



Skip Connection

layer를 깊게 할 때 사용되는 중요한 기법

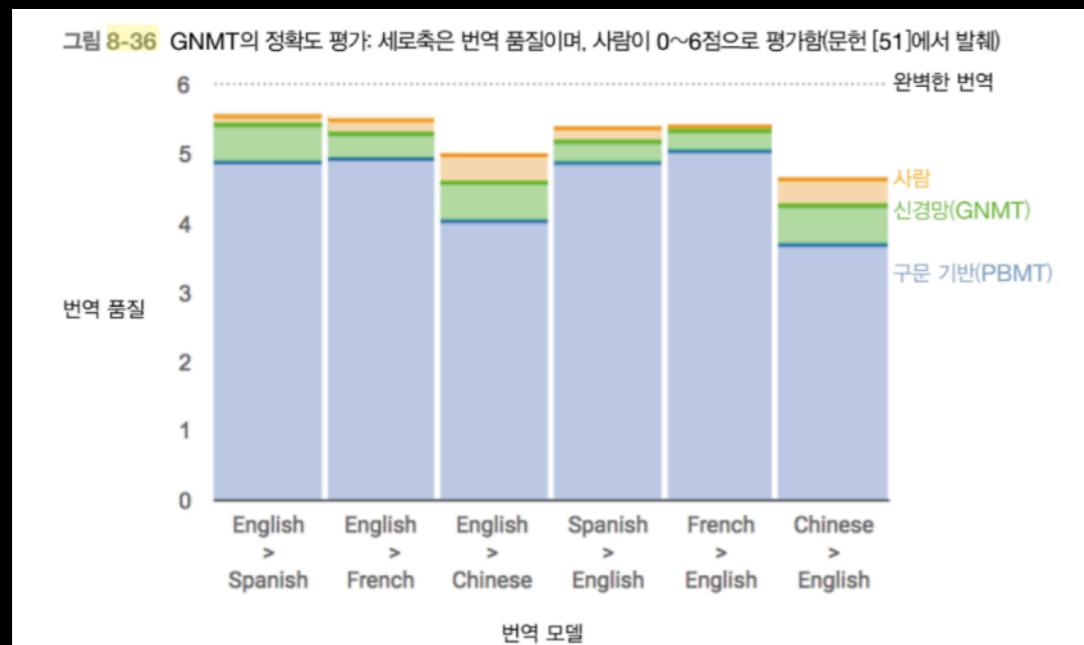
Skip connection을 사용함으로써 input data와 gradient가 오갈 수 있는 통로가 늘어남



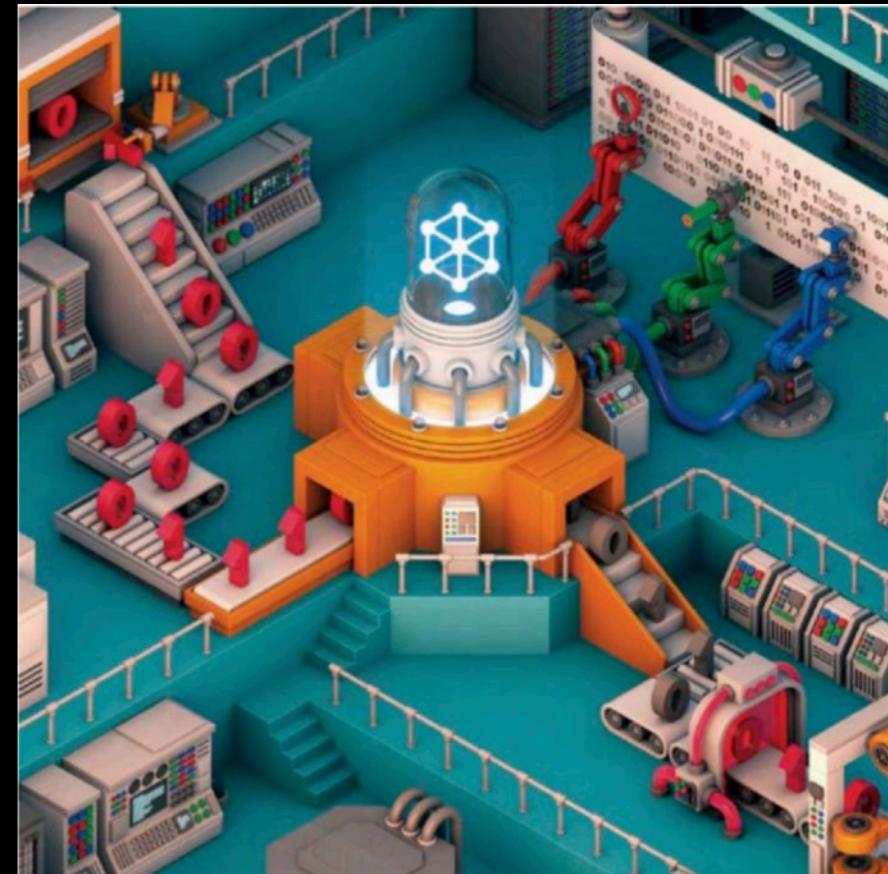
n 개의 skip-connection $\rightarrow 2^n$ 의 통로 \rightarrow 성능향상으로 이어짐

Applications

GNMT (google neural machine translation)



NTM (neural turning machine)



Transformer

- attention is all you need
- self attention
- 하나의 시계열 데이터 내에서 각 원소가 다른 원소들과 어떻게 관련되는지

memory를 활용한 연구
Encoder가 메모리에 정보를 쓰고, Decoder가 읽음