밑바닥부터 시작하는 딥러닝2

20213093 정현우

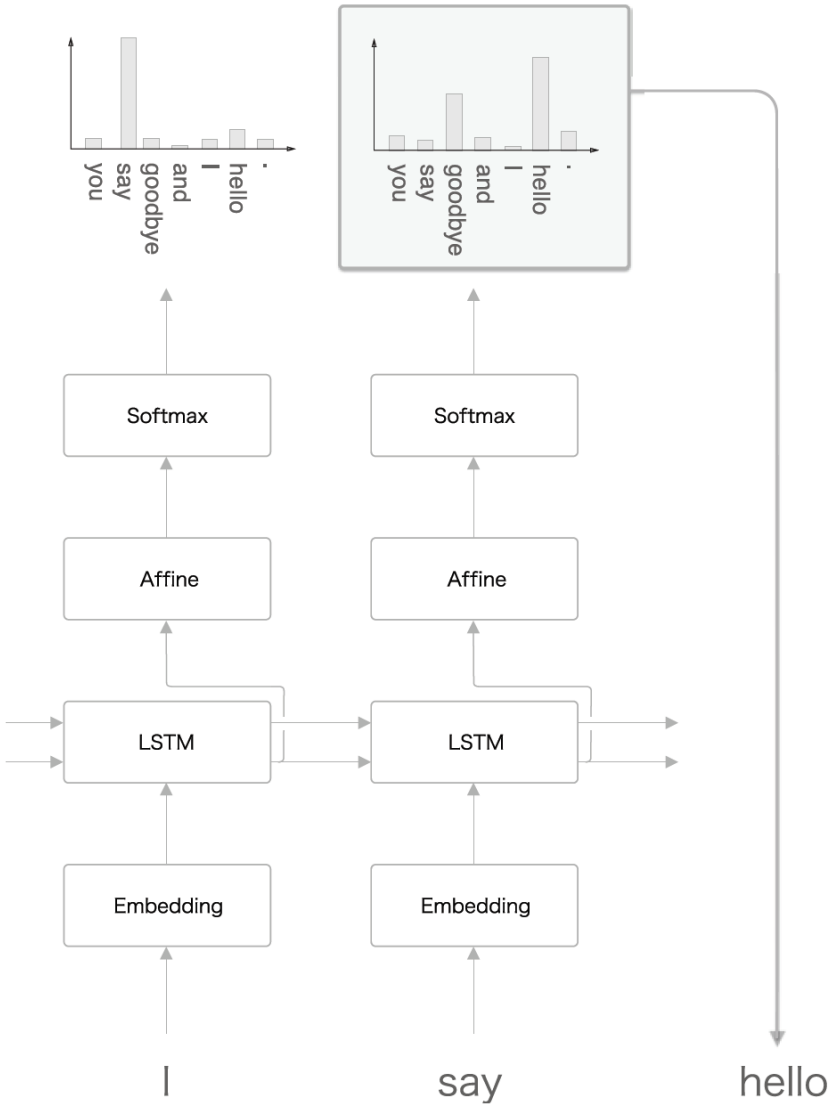
CONTENTS

언어모델 사용한
문장 생성

어텐션

1. 문장 생성

그림 7-4 확률분포 출력과 샘플링을 반복한다.



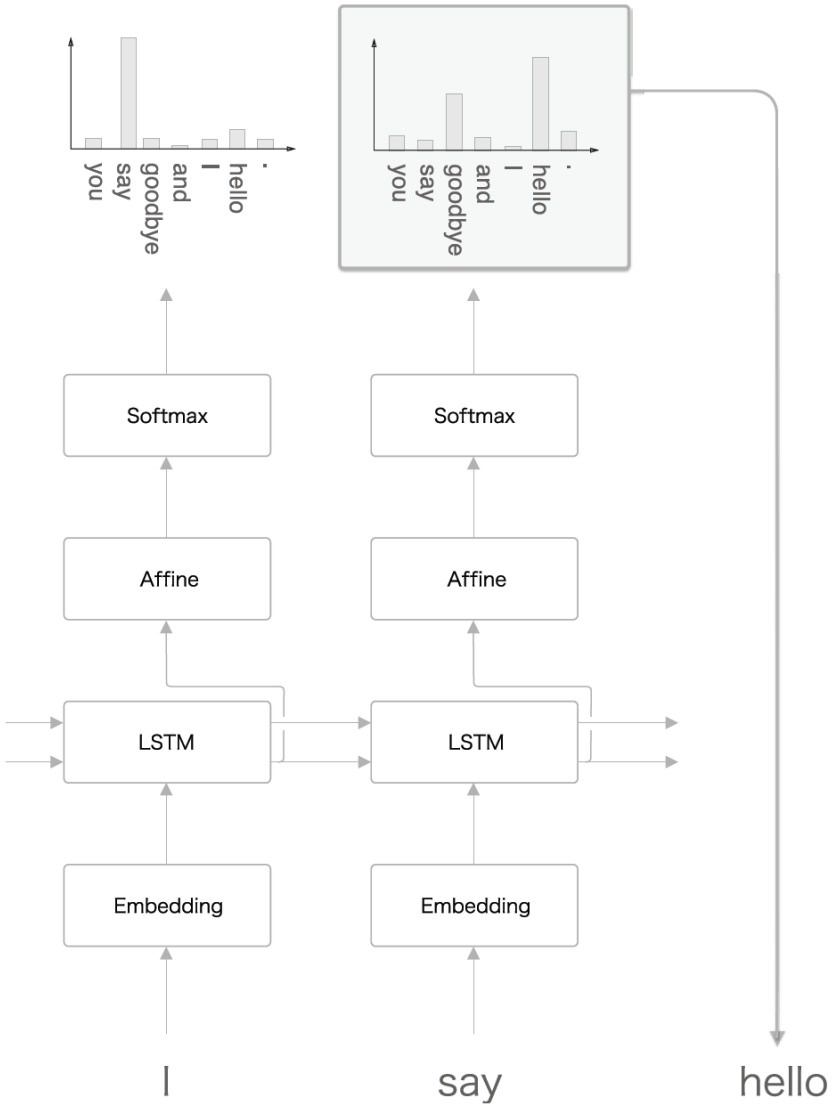
문장 생성 원리

확률 분포를 이용함.

학습을 통해 다음에 나올 법한 단어를 출력함.

1. 문장 생성

그림 7-4 확률분포 출력과 샘플링을 반복한다.

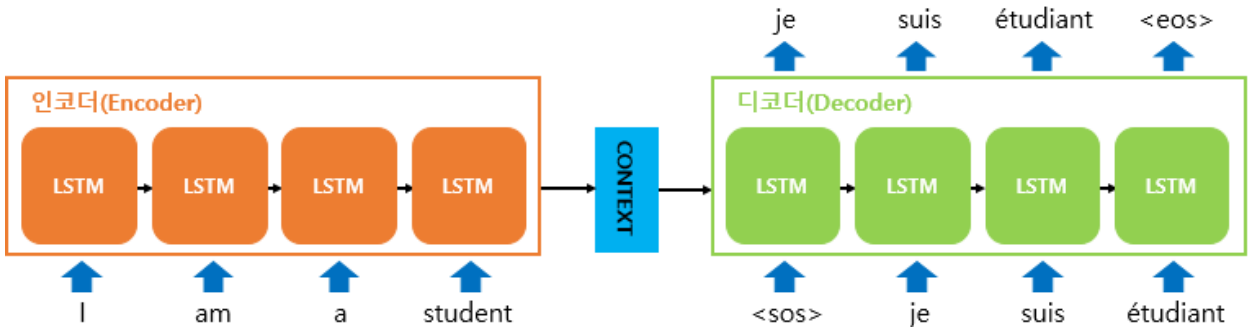


문장 생성 원리

확률 분포를 이용함.

학습을 통해 다음에 나올 법한 단어를 출력함.

1. 문장 생성



seq2seq

시계열을 또 다른 시계열로 변환.

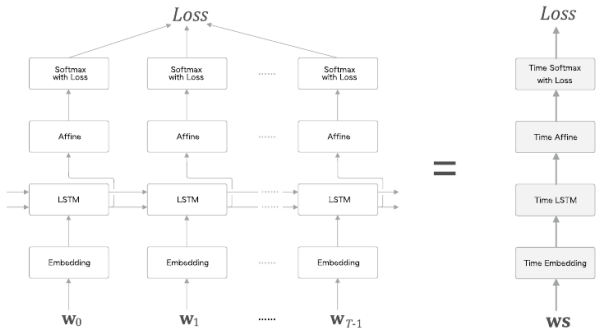
인코더(암호화)와 디코더(복호화) 구조 사용.

Context에 암호화된 정보가 응축되어 있음.

인코딩하는 것은 문장을 벡터로 변환하는 것임.

1. 문장 생성

Time LSTM



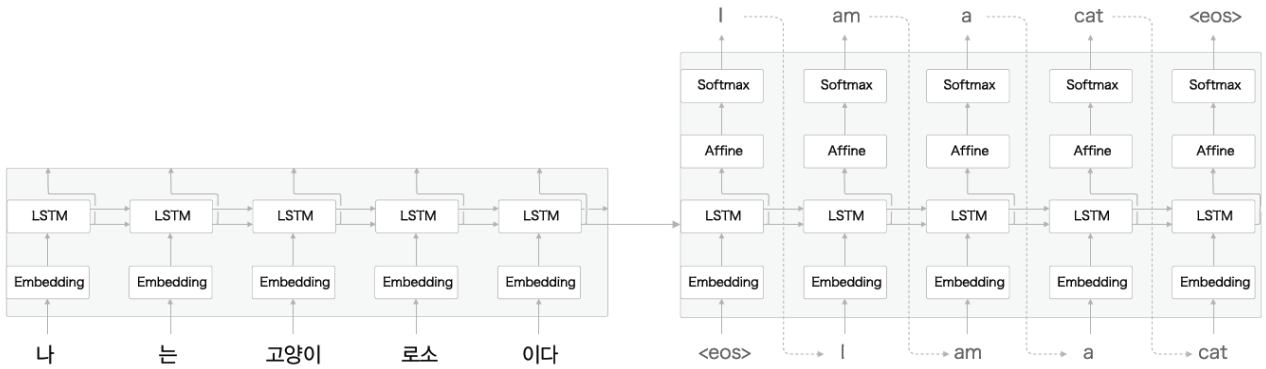
seq2seq

다음과 같은 구조로 되어있음.

앞에서 했던 Time LSTM 구조임.

인코더의 출력 -> 디코더의 입력 구조임

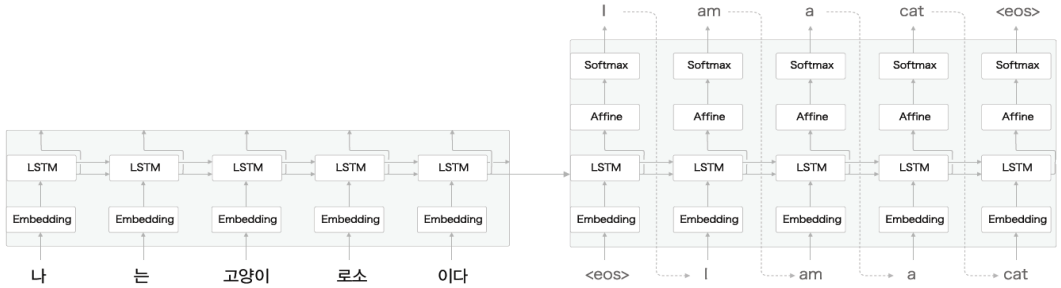
그림 7-9 seq2seq의 전체 계층 구성



1. 문장 생성

```python
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
```

그림 7-9 seq2seq의 전체 계층 구성



Seq2seq : Eecoder

Encoder 클래스 코드임.

앞에 나온 내용대로면 Embedding과 LSTM이 있어야 함.

TimeEmbedding & TimeLSTM 사용.

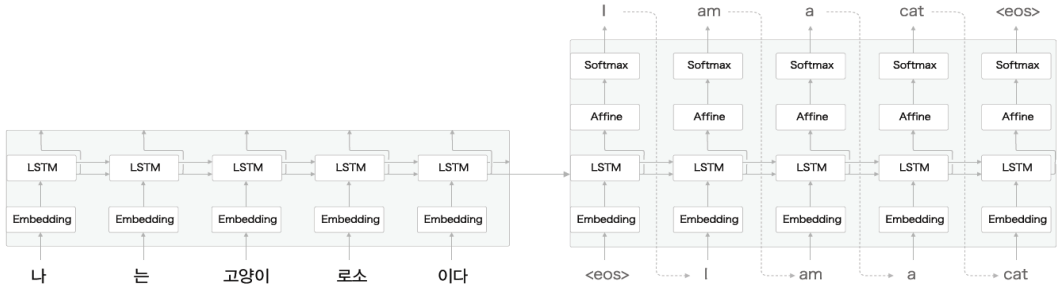
순전파는 LSTM의 마지막만 반환함.

역전파는 LSTM의 마지막 은닉 상태에 대한 기울기를 전해줌.

1. 문장 생성

```python
class Decoder:
    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(H, V) / np.sqrt(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.affine = TimeAffine(affine_W, affine_b)

        self.params, self.grads = [], []
        for layer in (self.embed, self.lstm, self.affine):
            self.params += layer.params
            self.grads += layer.grads

    def forward(self, xs, h):
        self.lstm.set_state(h)

        out = self.embed.forward(xs)
        out = self.lstm.forward(out)
        score = self.affine.forward(out)
        return score

    def backward(self, dscore):
        dout = self.affine.backward(dscore)
        dout = self.lstm.backward(dout)
        dout = self.embed.backward(dout)
        dh = self.lstm.dh
        return dh
```

그림 7-9 seq2seq의 전체 계층 구성



Seq2seq : Decoder

디코더 클래스 코드임.

Affine 층이 추가됨.

Embed, LSTM, affine 세트임.

위의 슬라이드 구조처럼 층을 하나씩 지나는 구조이다.

역전파의 경우도 하나씩 지나고 지난다.

전해주는 것은 LSTM의 기울기이다.

1. 문장 생성

```
def generate(self, h, start_id, sample_size):
    sampled = []
    sample_id = start_id
    self.lstm.set_state(h)

    for _ in range(sample_size):
        x = np.array(sample_id).reshape((1, 1))
        out = self.embed.forward(x)
        out = self.lstm.forward(out)
        score = self.affine.forward(out)

        sample_id = np.argmax(score.flatten())
        sampled.append(int(sample_id))

    return sampled
```

Seq2seq : Generate

디코더의 생성 함수이다.

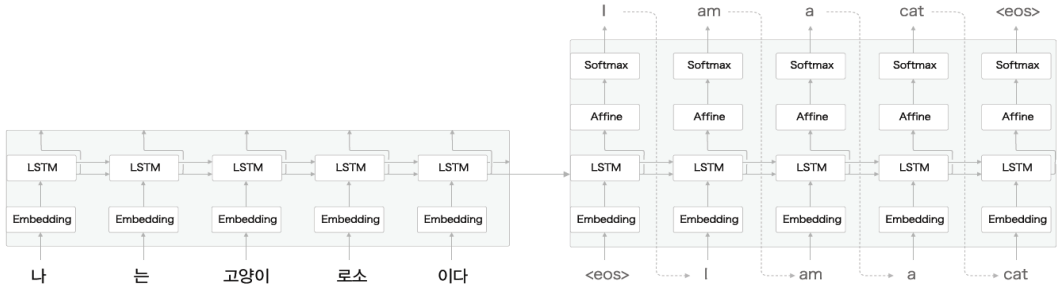
가장 확률이 높은 단어를 출력해서 추가한다.

스코어가 제일 높은 id를 뽑아서 저장함.

1. 문장 생성

```python
class Seq2seq(BaseModel):
    def __init__(self, vocab_size, wordvec_size, hidden_size):
        V, D, H = vocab_size, wordvec_size, hidden_size
        self.encoder = Encoder(V, D, H)
        self.decoder = Decoder(V, D, H)
        self.softmax = TimeSoftmaxWithLoss()

        self.params = self.encoder.params + self.decoder.params
        self.grads = self.encoder.grads + self.decoder.grads

    def forward(self, xs, ts):
        decoder_xs, decoder_ts = ts[:, :-1], ts[:, 1:]

        h = self.encoder.forward(xs)
        score = self.decoder.forward(decoder_xs, h)
        loss = self.softmax.forward(score, decoder_ts)
        return loss

    def backward(self, dout=1):
        dout = self.softmax.backward(dout)
        dh = self.decoder.backward(dout)
        dout = self.encoder.backward(dh)
        return dout

    def generate(self, xs, start_id, sample_size):
        h = self.encoder.forward(xs)
        sampled = self.decoder.generate(h, start_id, sample_size)
        return sampled
```

그림 7-9 seq2seq의 전체 계층 구성



seq2seq

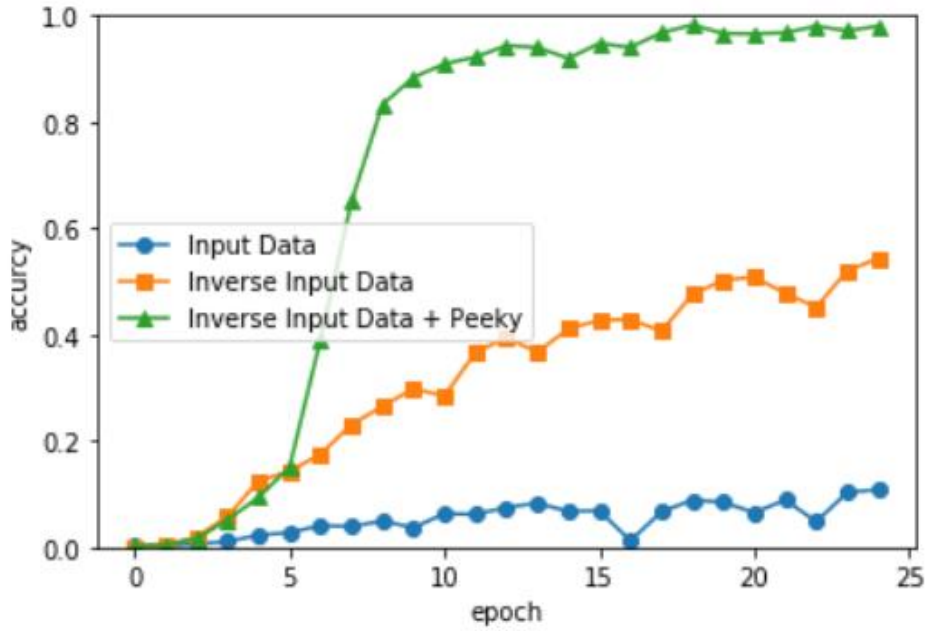
앞에서 한 인코더와 디코더 그리고 손실함수로 구성된다.

순전파는 Encoder -> Decoder -> 손실함수

역전파는 손실함수 -> Decoder -> Encoder

h 에 인코더의 출력을 저장하고 이것을 디코더에 넣어주면서 출력한다.

1. 문장 생성



Seq2seq : 개선

개선 방법으로는

입력 데이터 반전 (Reverse)

엿보기(Peeky)

가 있음.

입력 데이터 반전을 사용하면 데이터를 빠르게 학습 할 수 있고

엿보기를 사용하면 정확도가 크게 증가한다.

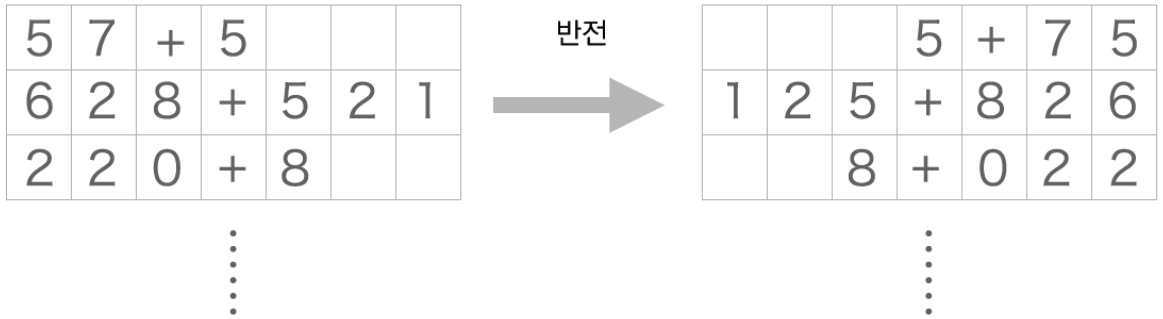
1. 문장 생성

Seq2seq : 개선

데이터 반전은 데이터를 뒤집어서 입력시킴.

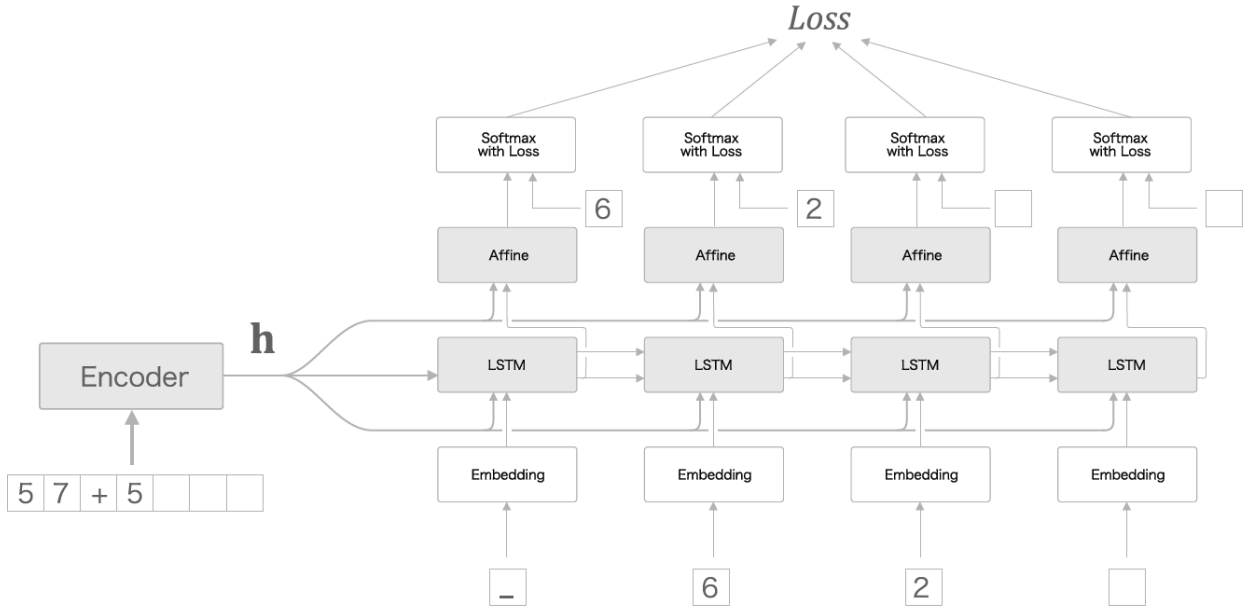
왜인지는 설명이 없지만 대부분의 경우에 성능이 좋아진다고 한다.

그림 7-23 입력 데이터를 반전시키는 예



1. 문장 생성

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



Seq2seq : 개선

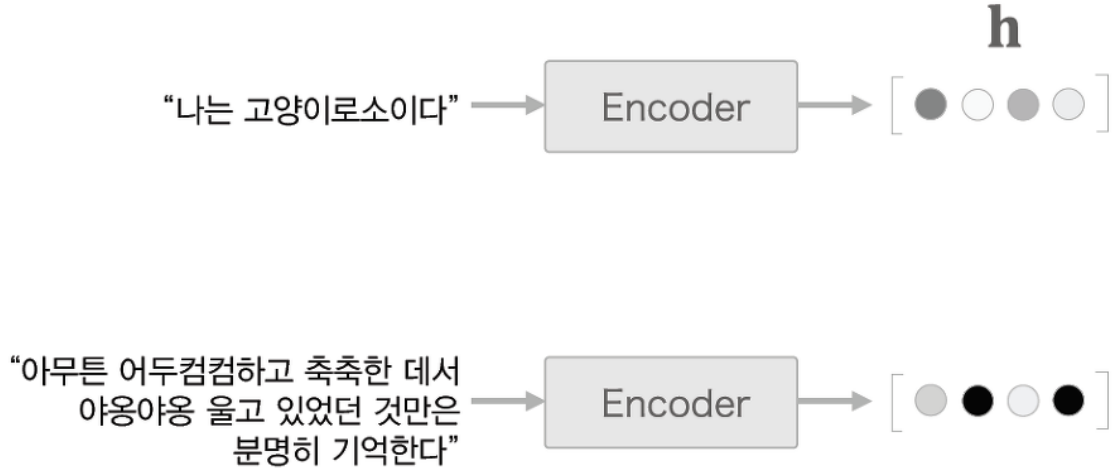
Peeky는 인코더의 결과 h를 디코더의 첫번째 계층 뿐만이 아니라

더 많은 계층에도 넣어주는 것이다.

2. 어텐션

Seq2seq : 문제점

그림 8-1 입력 문장의 길이에 관계없이, Encoder는 정보를 고정 길이의 벡터로 밀어 넣는다.



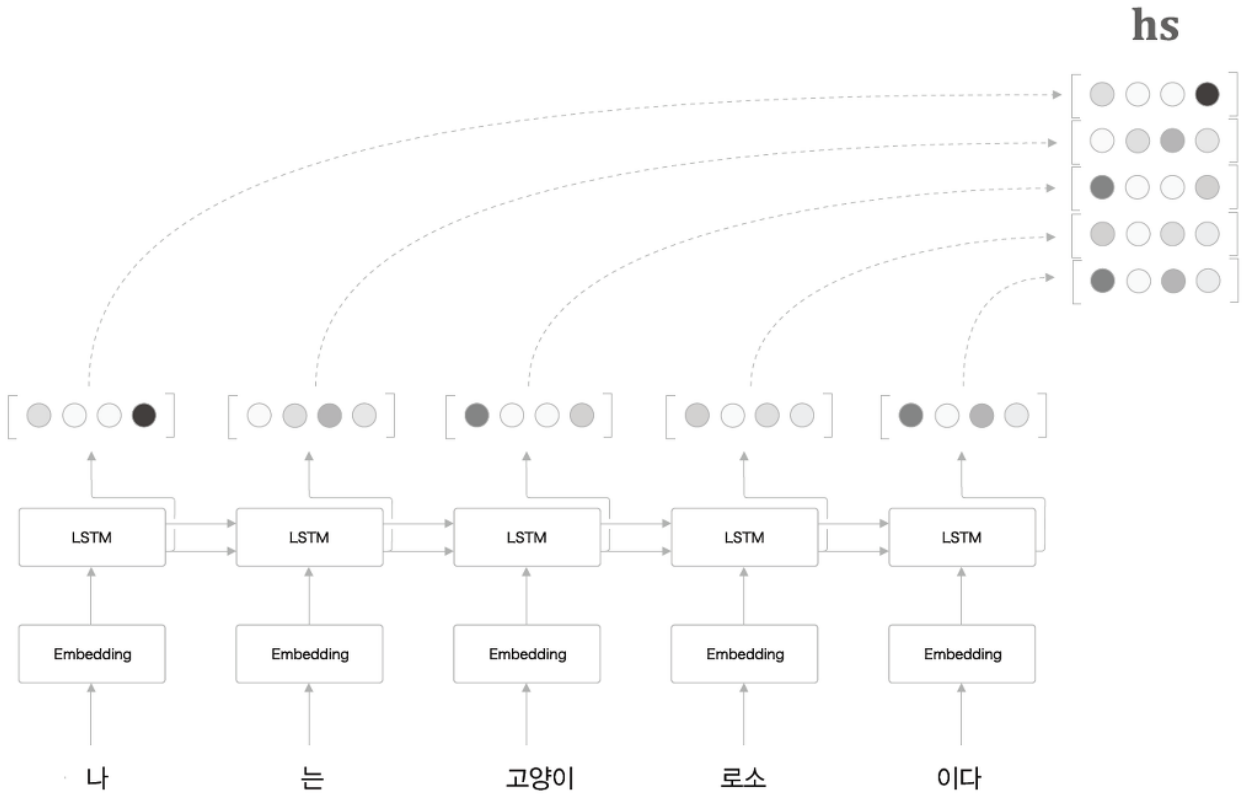
Encoder 의 출력이 항상 고정된 길이의 벡터임.

아무리 긴 문장도 항상 똑같은 길이의 벡터가 됨.

그렇게 되면 h 벡터는 정보를 온전히 담기 어려움.

2. 어텐션

그림 8-2 Encoder의 시각별(단어별) LSTM 계층의 은닉 상태를 모두 이용(**hs**로 표기)



Encoder : 개선

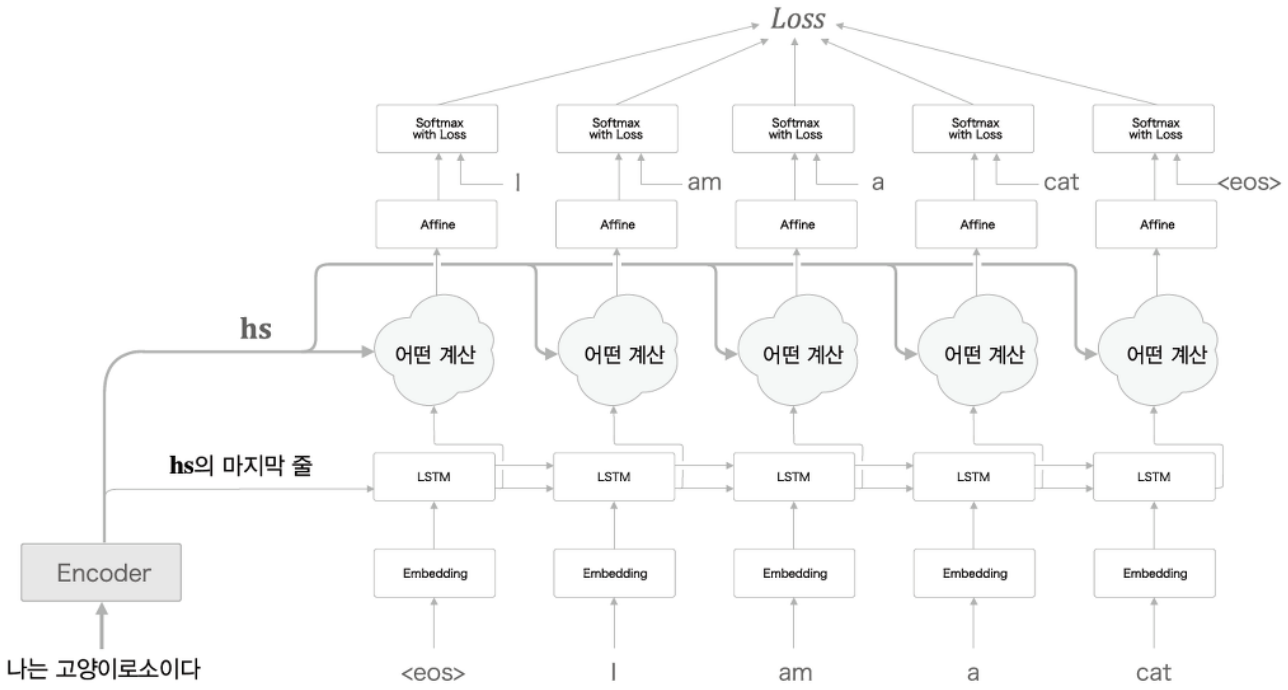
이전에서 Encoder 의 출력 h는 고정된 길이의 벡터였지만 이렇게 하면 "고정된" 이라는 제약이 풀린다.

원래라면 이 hs에서에서 마지막 정보만 넘겨주었던 것이다.

우리는 이 hs를 적절하게 디코더에게 넘겨 주어야 한다.

2. 어텐션

그림 8-6 개선 후의 Decoder의 계층 구성



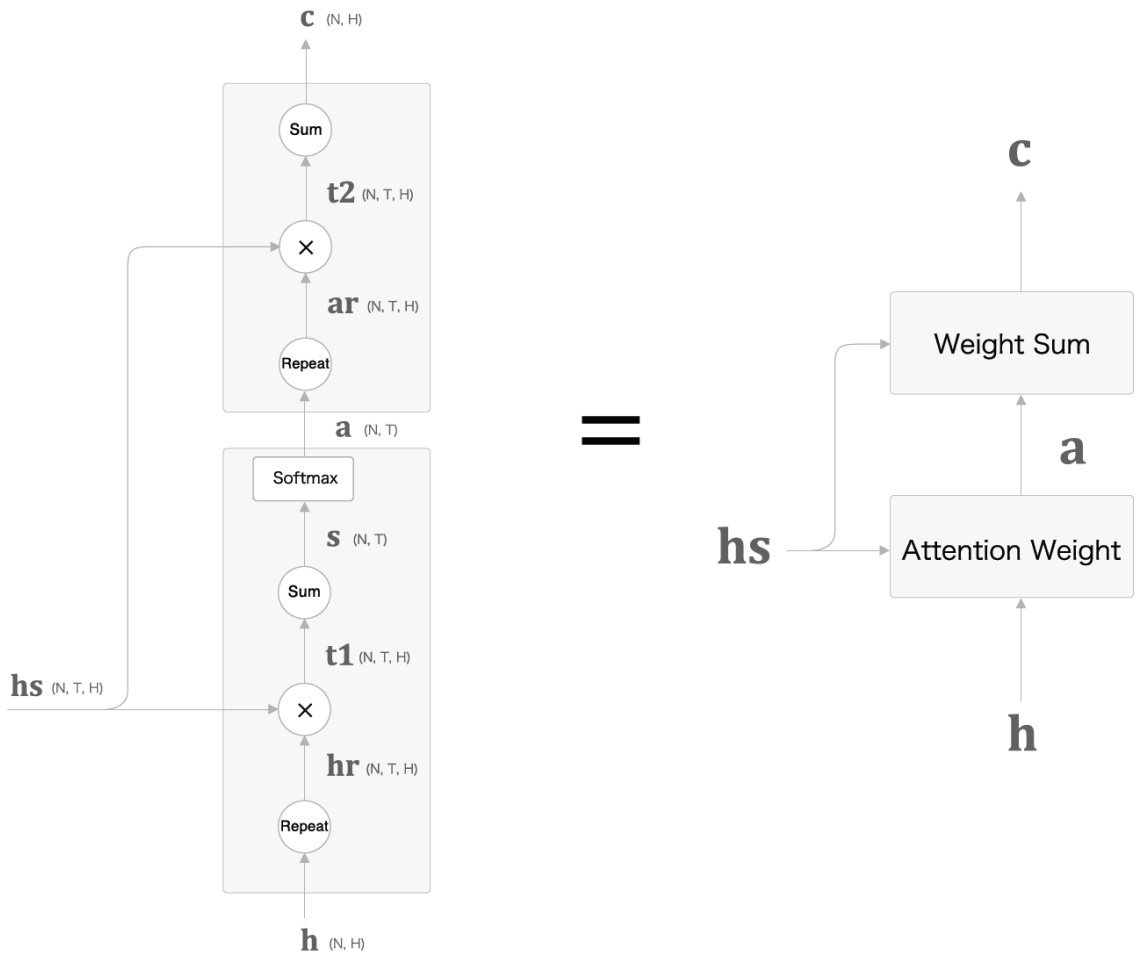
Decoder : 개선

Encoder 의 개선에 맞게 이제 Decoder를 적절하게 개선해주어야 한다.

여기서 각각의 단어에 해당하는 벡터를 적절하게 골라내는 것이 어떤 계산이다.

2. 어텐션

그림 8-16 맥락 벡터를 계산하는 계산 그래프



Decoder : Attention

다음과 같이 Attention Weight와 Weight Sum이 있다.

이 두개를 합친 것을 Attention이라고 부른다.

Hs는 Encoder의 출력이다.

H는 LSTM 계층의 은닉 상태 벡터이다.

A는 단어의 가중치이다.

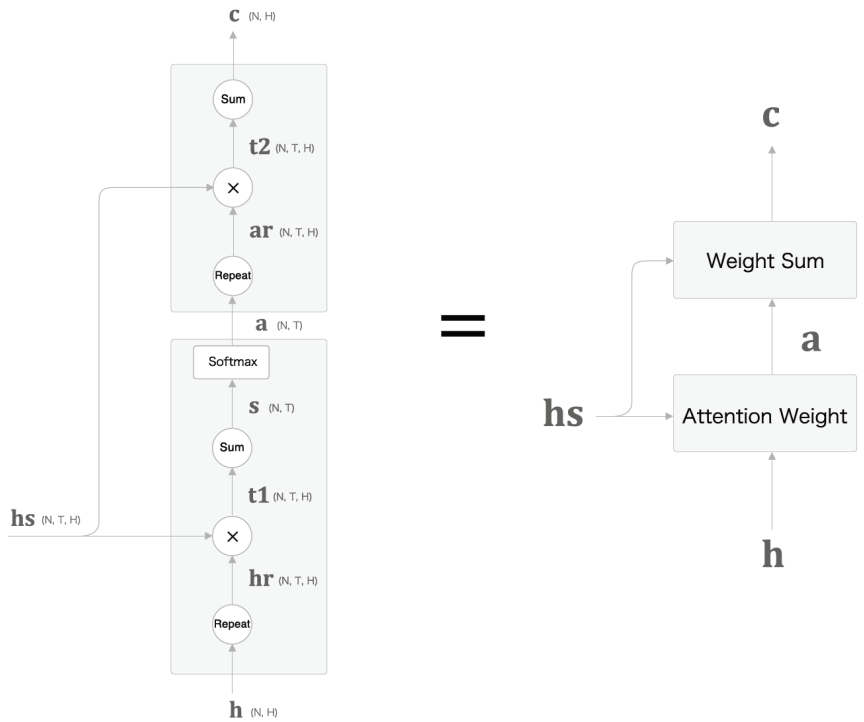
이를 통해 h가각 단어 벡터와 얼마나 비슷한가를 수치로 나타내는 것이다.

2. 어텐션

```python
class AttentionWeight:
    def __init__(self):
        self.parmas, self.grads = [], []
        self.softmax = softmax()
        self.cache = None

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

        hr = h.reshape(N, 1, H).repeat(T, axis=1)
        t = hs * hr
        s = np.sum(t, axis=2)

        a = self.softmax.forward(s)
        self.cache = (hs, hr)
        return a

    def backward(self, da):
        hs, hr = self.cache
        N, T, H = hs.shape

        ds = self.softmax.backward(da)
        dt = ds.reshape(N, T, 1).repeat(H, axis=2)
        dhs = dt * hr
        dhr = dt * hs
        dh = np.sum(dhr, axis=1)

        return dhs, dh
```

Decoder : Attention

그림 8-16 맥락 벡터를 계산하는 계산 그래프

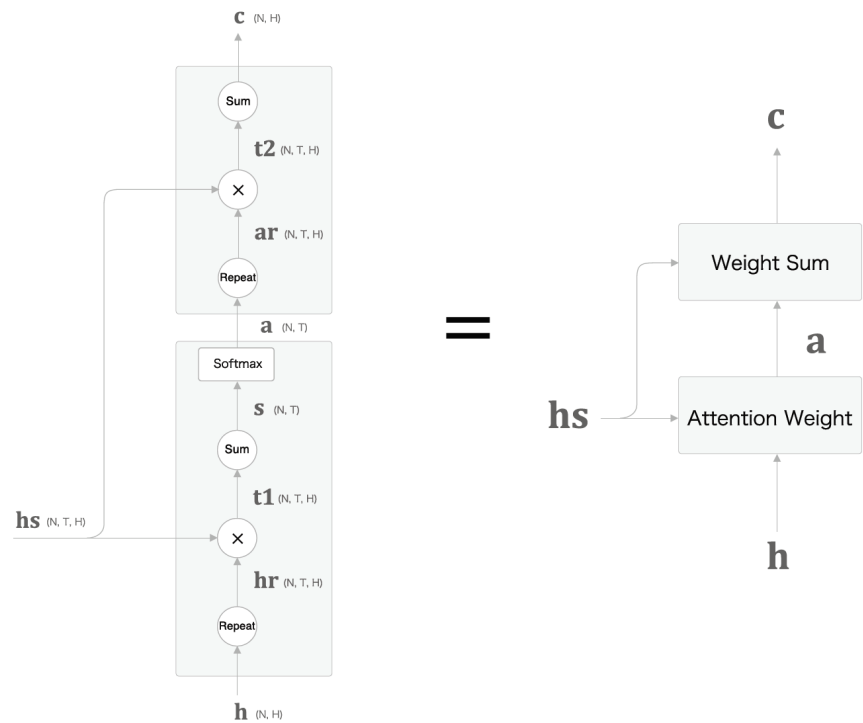


2. 어텐션

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

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

        ar = a.reshape(N, T, 1).repeat(H, axis=2)
        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
```

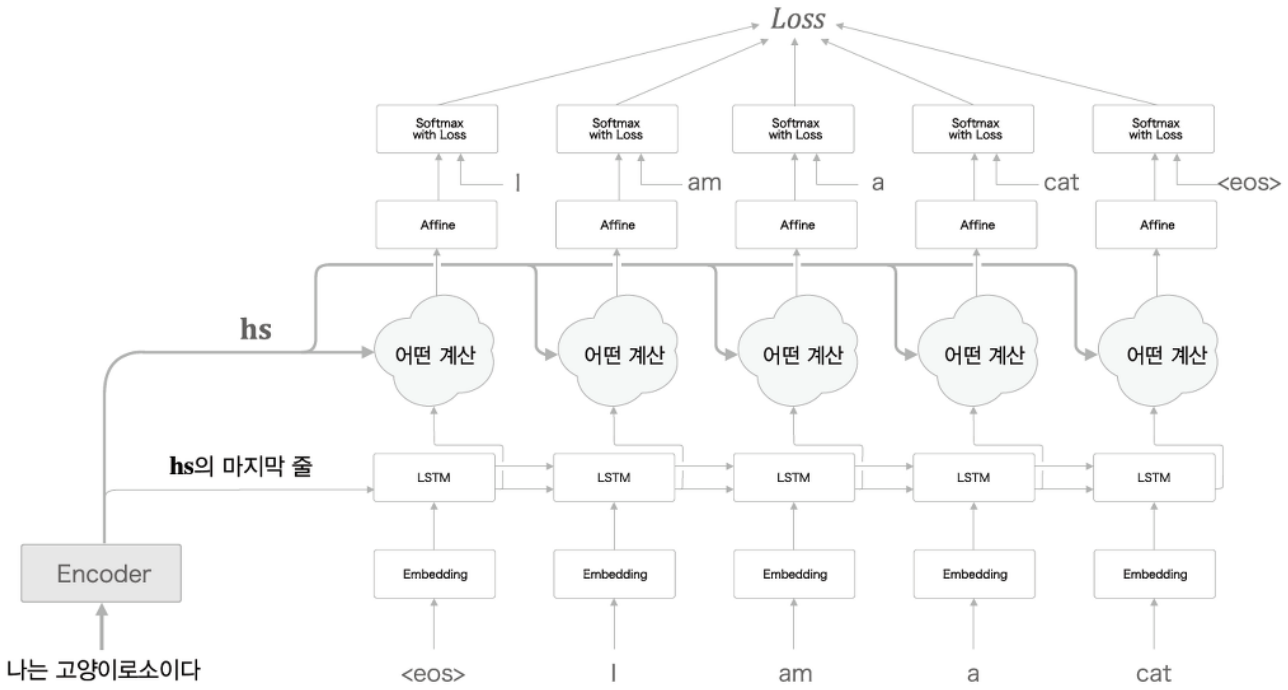
Decoder : Attention

그림 8-16 맥락 벡터를 계산하는 계산 그래프



2. 어텐션

그림 8-6 개선 후의 Decoder의 계층 구성



Decoder : 개선

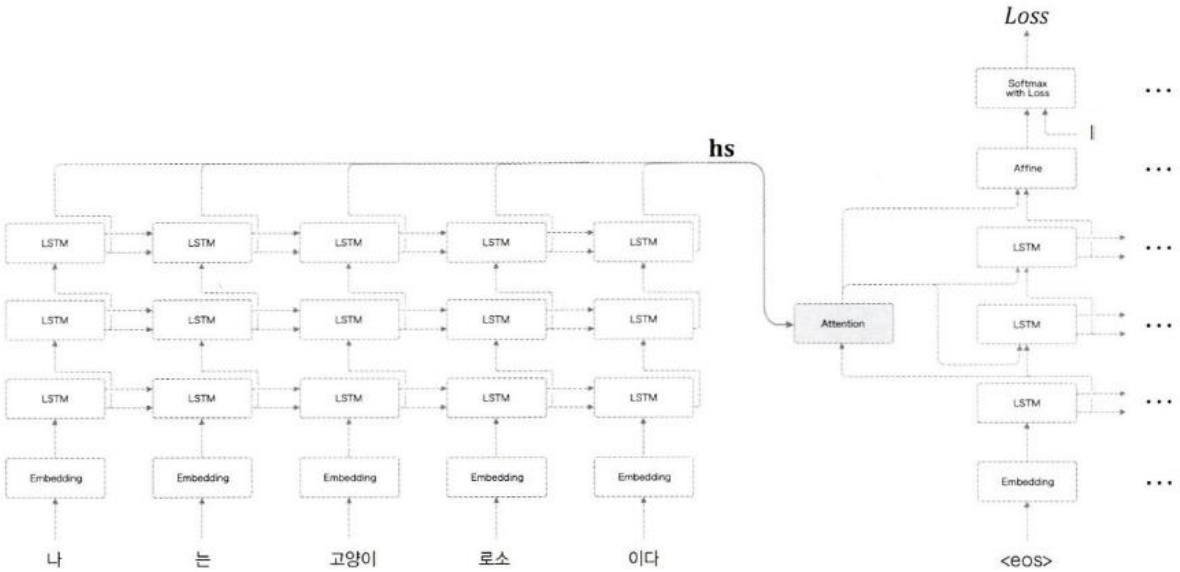
전체적으로 보면 다음과 같다.

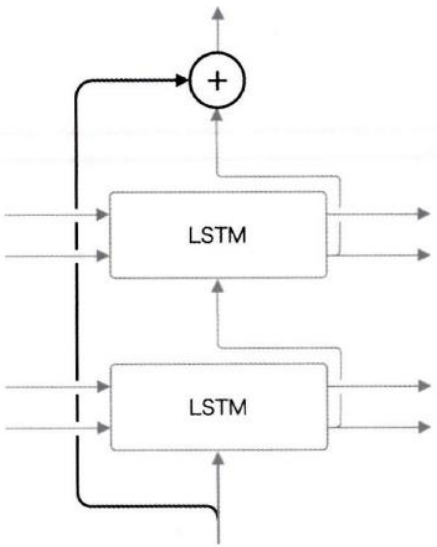
어떤 계산은 어텐션이다.

Hs에 encoder의 단어별 벡터가 있고

이를 어텐션 계층에 넘겨준다.

2. 어텐션





Seq2seq 심층화 : skip 연결

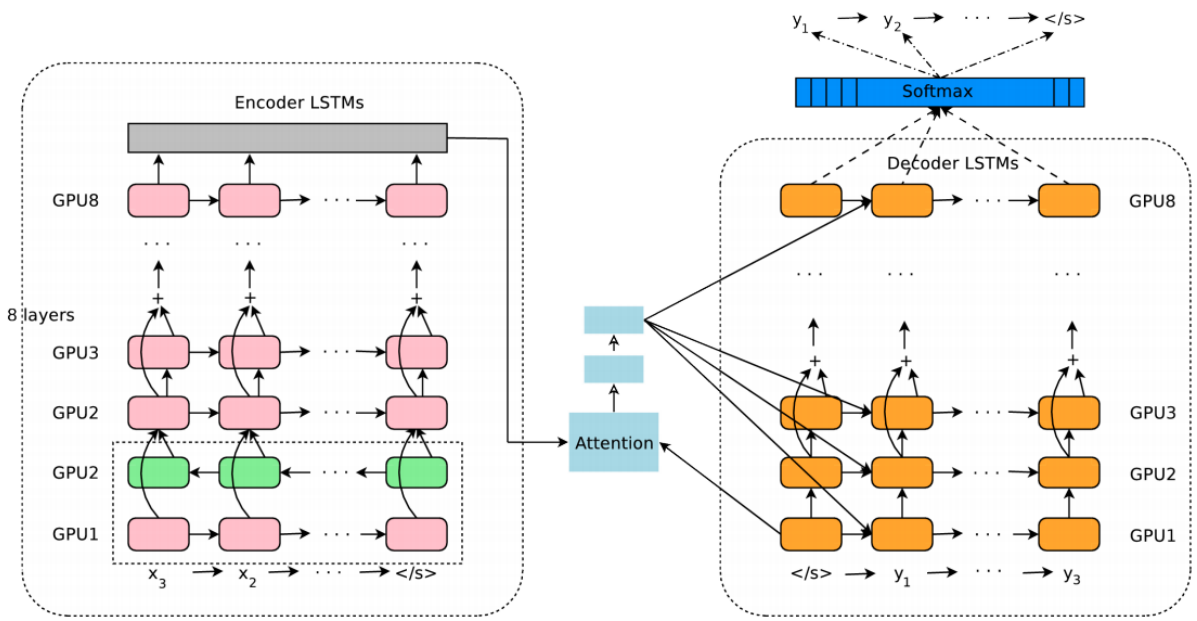
현실에서의 문제를 풀 때는 다음과 같이 LSTM 계층을 깊게 쌓기도 한다.

이때 기울기 소실이나 폭발과 같은 문제가 일어나는 것을 방지 하기 위해서

아래의 그림과 같이 몇 개의 층을 뛰어넘어 더해주는 방식으로

이를 방지하고 좋은 결과를 기대한다.

2. 어텐션



어텐션 응용 : GNMT

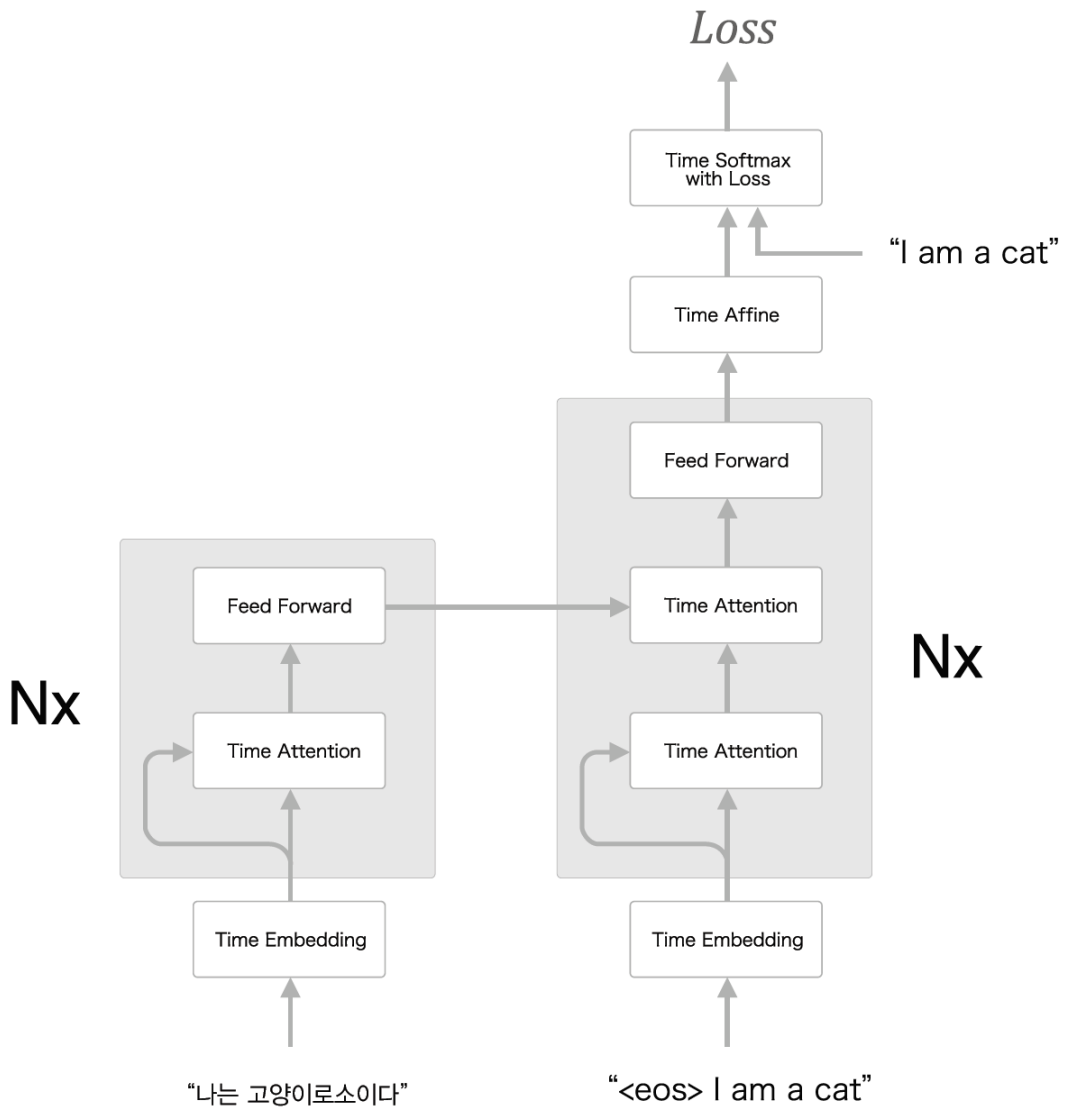
여러 개선이 이루어진 모습이다.

LSTM 계층의 다양화, 양방향 LSTM, skip 연결 등을 볼 수 있다.

GPU 분산 학습을 수행하여 학습 시간을 단축시키고 있는 모습도 보이고 있다.

2. 어텐션

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

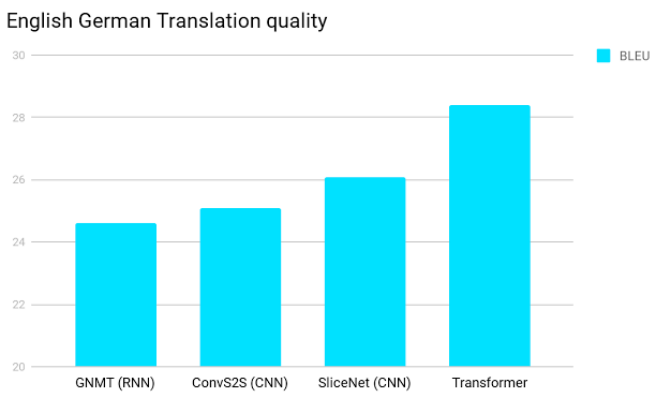


Self Attention : 트랜스포머

RNN 을 없애려는 시도.

Self Attention 을 만들어냄.

RNN, CNN 기반보다 좋은 성능을 냄.



Thank you