Attention

- Attention : seq2seq가 가지고 있는 근본적인 문제를 해결하는 '큰 개선'을 해주는 매커니즘
- 필요한 정보에만 주목하여 그 정보로부터 시계열 변환을 수행(인간이 하는 것 처럼)
- 두 시계열 데이터 사이의 대응관계를 데이터로부터 학습
- 모델이 수행하는 작업을 인간이 이해할 수도 있게 됨
    - 신경망 내부에서 어떠한 처리가 이뤄지고 있는지(어떠한 논리로 처리되고 있는지) 인간이 이해할 수 없는 것이 일반적임. 반면 어텐션의 matrix를 통해 단어와 단어의 관련성 등 '인간이 이해할 수 있는 구조나 의미'를 알 수 있음. 이를 보고 모델의 처리 논리가 인간의 논리를 따르는지 판단할 수 있음

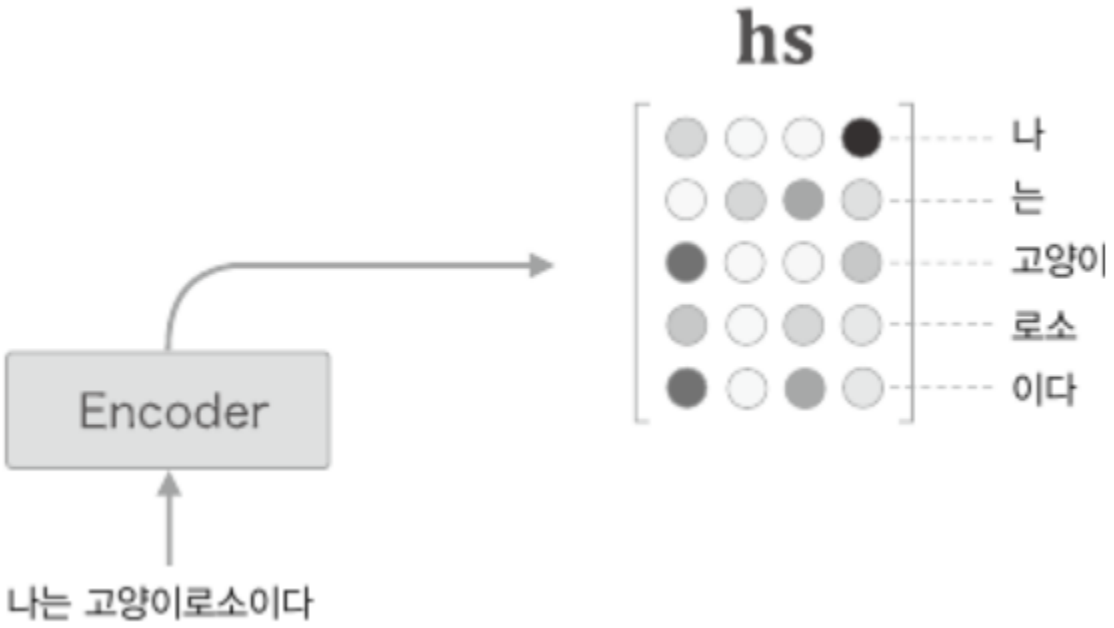
1. Attention의 구조

1.1 seq2seq의 문제점

1. Encoder의 출력이 '고정 길이의 벡터'
    - 같은 모델을 사용한다면 입력 문장 길이에 관계없이 항상 같은 길이의 벡터로 변환
    - ex) '나는 고양이이다'와 '아무튼 어두컴컴하고 축축한 데서 야옹야옹 울고 있었던 것만은 분명히 기억한다'의 hidden vector의 size가 같음

1.2 Encoder 개선

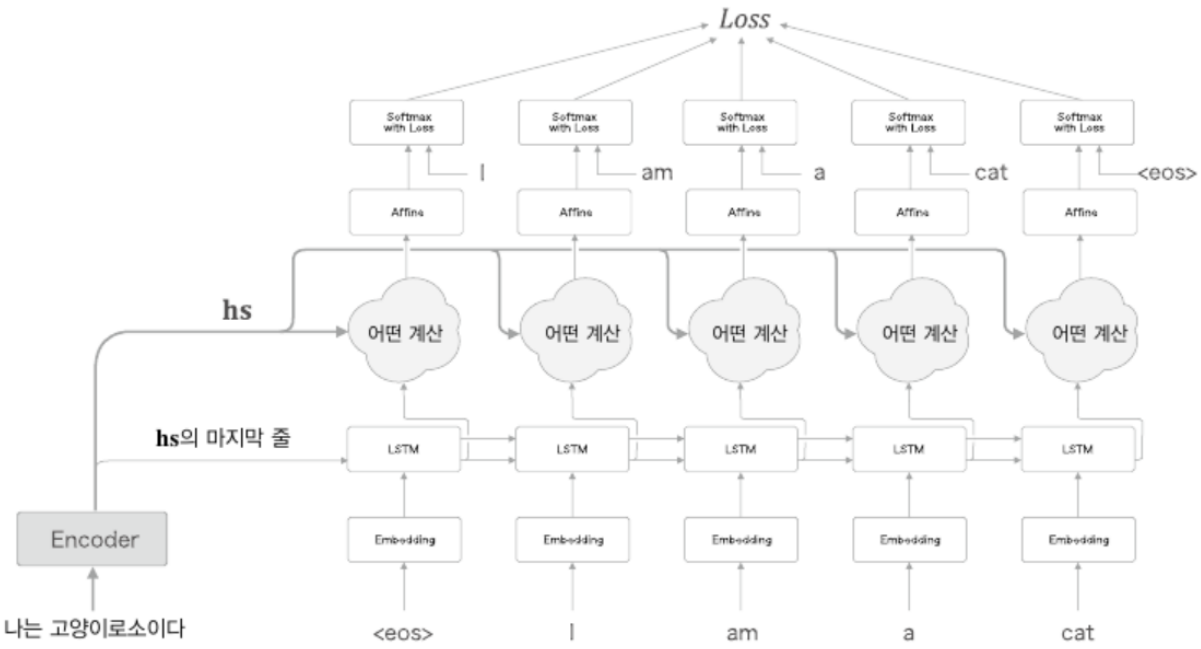
- 기존 : 마지막 시각의 LSTM의 출력만을 사용, 모든 input에 대해 hidden vector의 크기가 같음
- 개선 후 : 모든 시각의 LSTM의 출력을 사용, 각 input에 대해 hidden vector의 크기를 다르게 해줌
    - 모든 시각의 LSTM 출력을 사용하면 input 시각의 개수에 따라 output의 출력 크기가 달라짐. 만약 input layer의 시각이 5개면 output의 vector 개수도 5개.
    - 각 시각의 LSTM의 출력은 직전 input의 영향을 가장 많이 받음. 따라서 각 output vector는 각 단어에 해당하는 vector들의 집합
        - 단방향 RNN은 첫번째 input이 가장 많은 vector에 담겨 있음(양방향 *RNN(LSTM)*의 경우 모든 데이터를 균형있게 담을 수 있음)



1.3 Decoder 개선1

- "나" -> "I", "고양이" -> "cat"과 같은 **alignment**를 seq2seq를 통해 자동으로 도입
- 필요한 정보에만 주목하여 그 정보로부터 시계열 변환을 수행
- 즉, "나"에서 "I"로의 번역을 위해선 "나"라는 word의 정보가 가장 많이 필요. 이렇게 번역에 필요한 정보를 뽑아내어 번역에 이용하겠다는 뜻
  - 즉 hs 중 "나"에 대응하는 벡터를 선택. -> 이러한 선택을 '어떤 계산'으로 해내겠다.

그림 8-6 개선 후의 Decoder의 계층 구성



- encoder의 출력인 시각별 LSTM의 은닉상태에서 필요한 정보만 골라 위쪽의 Affine layer으로 출력

- 이 때, 단순히 vector를 선택하게 되면 역전파를 할 수 없음. 즉 선택되는 vector에 대해서만 gradient descent가 이루어짐. 따라서 모든 hs를 선택하지만 각 단어의 중요도(기여도)를 나타내는

'가중치'를 별도로 계산하도록 함

- '가중치'(a)의 범위 0.0~1.0 사이의 스칼라(단일 원소)이며, 모든 원소의 총합 : 1
- 가중치 a에 따라 vector와 weighted sum을 통해 원하는 vector(맥락벡터, c)를 얻음
  - 맥락벡터에는 가장 높은 가중치를 보이는 vector의 성분이 가장 많이 포함되어 있음. 즉, 가장 높은 가중치를 보이는 vector를 '선택'하는 작업이 이로 대체되었다고 할 수 있음(만약 가중치가 1이라면 '선택'한다고 해석 가능)

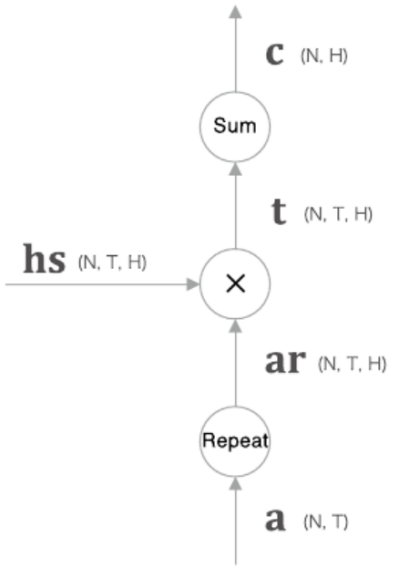
In [4]:
```
import numpy as np

# SGD(한 개의 데이터) 처리용 가중합
T, H = 5,4
hs = np.random.randn(T,H) # 각 시각별 output hidden vector
a = np.array([0.8, 0.1, 0.03, 0.05, 0.02]) # 각 hs vector의 weighted sum

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

t = hs*ar # weighted
print(t.shape)

c = np.sum(t, axis=0) # weighted 'sum'
print(c.shape)

"""
이 모든 과정은 np.matmul(a,hs) 한줄로 구현 가능.
but, mini-batch 처리로 확장하기 쉽지 않음. 이를 위해선 '텐서곱'계산이 필요
"""
```

```
(5, 4)
(5, 4)
(4,)
```

In [5]:
```
# 미니배치 처리용 가중합
N, T, H = 10, 5, 4
hs = np.random.randn(N,T,H)
a = np.random.randn(N,T)
ar = a.reshape(N,T,1).repeat(H, axis=2) # H축에 대해선 같은 가중치여야 함
# ar = a.reshape(N,T,1) -> broadcasting

t = hs*ar # weighted
print(t.shape)

c = np.sum(t, axis=1) # weighted sum
print(c.shape)
```

```
(10, 5, 4)
(10, 4)
```

그림 8-11 가중합의 계산 그래프



In [6]:
```
class WeightedSum:
    # weight와 encoder의 hs간 weighted sum을 구하는 class
    def __init__(self):
        # 학습하는 파라미터가 없기 때문에 파라미터와 gradient list가 []으로 설정됨
        self.params, self.grads = [], []
        self.cache = None

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

        # weighted sum
        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

        # T 방향으로 더해졌었기 때문에 T방향으로 repeat
        dt = dc.reshape(N,1,H).repeat(T, axis=1) # sum의 역전파
        dar = dt * hs # 가중치 matrix에 대한 미분값
        dhs = dt * ar # hs matrix에 대한 미분값

        # H 방향으로 repeat 되었었기 때문에 H방향으로 sum
        da = np.sum(dar, axis=2) # repeat의 역전파

        return dhs, da
```

1.4 Decoder 개선2

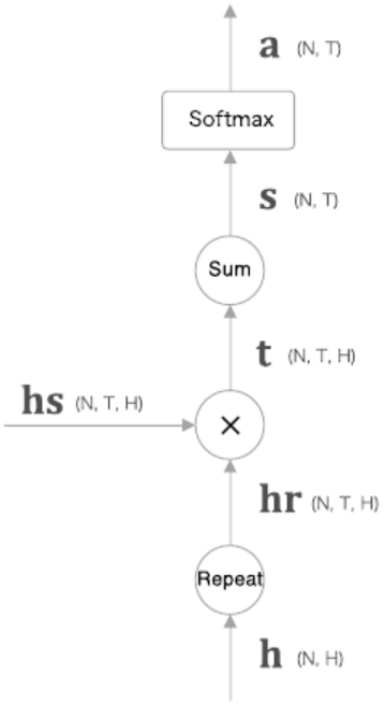
- 가중치 matrix 'a'를 구하기 위한 방법으로 벡터의 '내적'을 이용
  - 내적은 '두 vector가 얼마나 같은 방향을 향하고 있는가'의 직관적인 의미를 담 고있기 때문
  - 내적 외에도 다양한 방법이 사용될 수 있음(신경망 등)

- 내적 후 softmax function을 사용해서 정규화

In [15]:
```
import sys
import os
sys.path.append(os.path.join("..","master"))
from common.layers import Softmax
import numpy as np

N, T, H = 10,5,4
hs = np.random.randn(N,T,H) # encoder 출력 h matrix
h = np.random.randn(N,H)    # decoder 출력 h vector
hr = h.reshape(N,1,H).repeat(T, axis=1) # decoder 출력 h를 time을 축으로 repeat
# hr = h.reshape(N,1,H) # 브로드캐스트를 사용하는 경우

# 내적
t = hs*hr
print(t.shape)
s = np.sum(t, axis=2)
print(s.shape)

softmax = Softmax()
a = softmax.forward(s)
print(a.shape)
```

```
(10, 5, 4)
(10, 5)
(10, 5)
```

그림 8-15 각 단어의 가중치를 구하는 계산 그래프



In [ ]:
```
import sys
import os
import numpy as np
sys.path.append(os.path.join("..","master"))
from common.layers import Softmax

class AttentionWeight:
    # weight 를 구하는 class
    def __init__(self):
        # 파라미터가 없기 때문에 파라미터와 grads가 []로 설정됨
        self.params, 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)
        # 내적
        t = hs*hr
        s = np.sum(t, axis=2)
        # 정규화
        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) # sum의 역전파
        dhs = dt * hr
        dhr = dt * ds
        dh = np.sum(dhr, axis=1) # repeat의 역전파

        return dhs, dh
```

1.5 Decoder 개선3

- Attention Weight와 Weight Sum을 연결 -> 연결된 층을 Attention layer이라고 부름
- Attention layer -> Attention Weight, Weight Sum

그림 8-16 맥락 벡터를 계산하는 계산 그래프

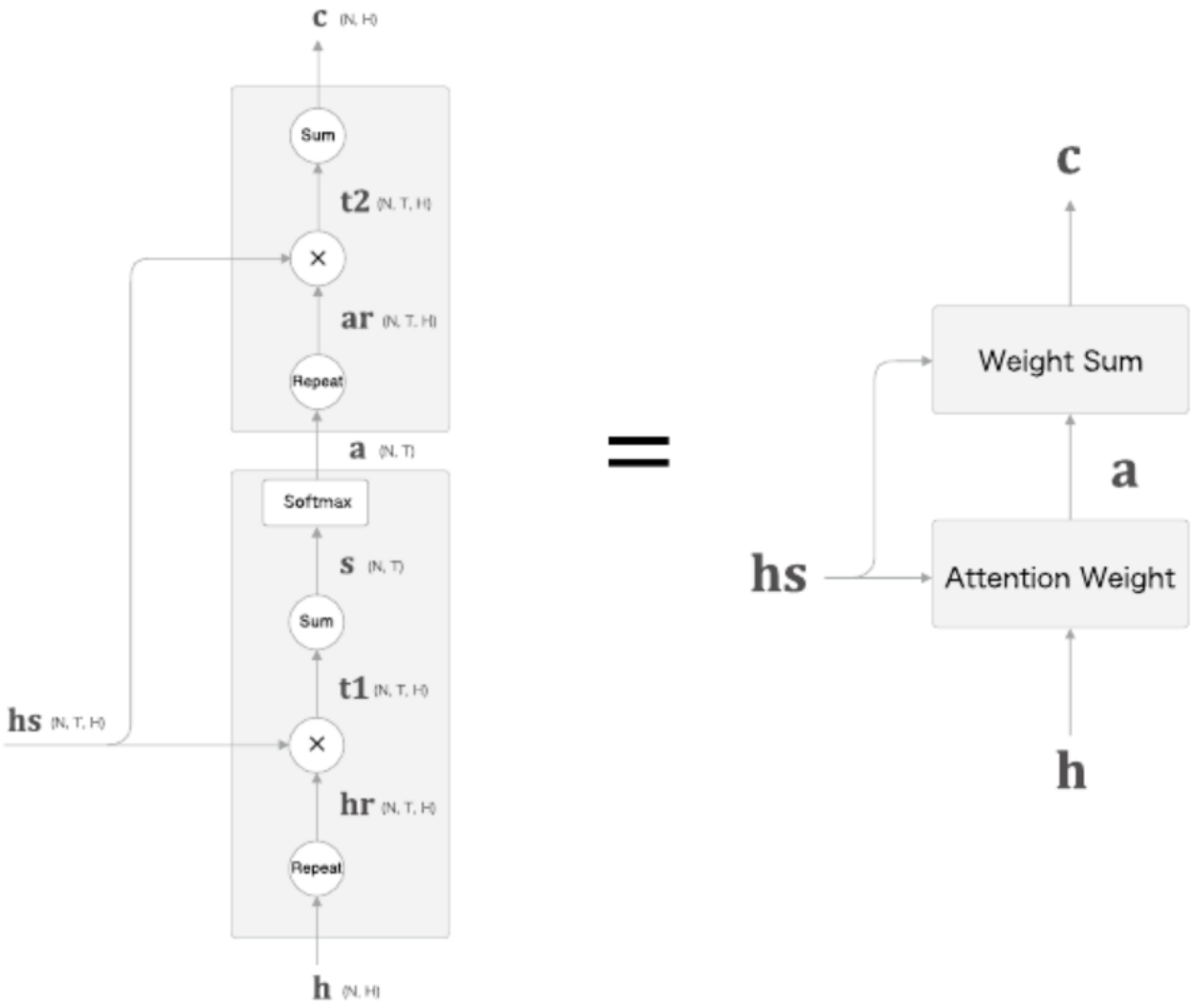
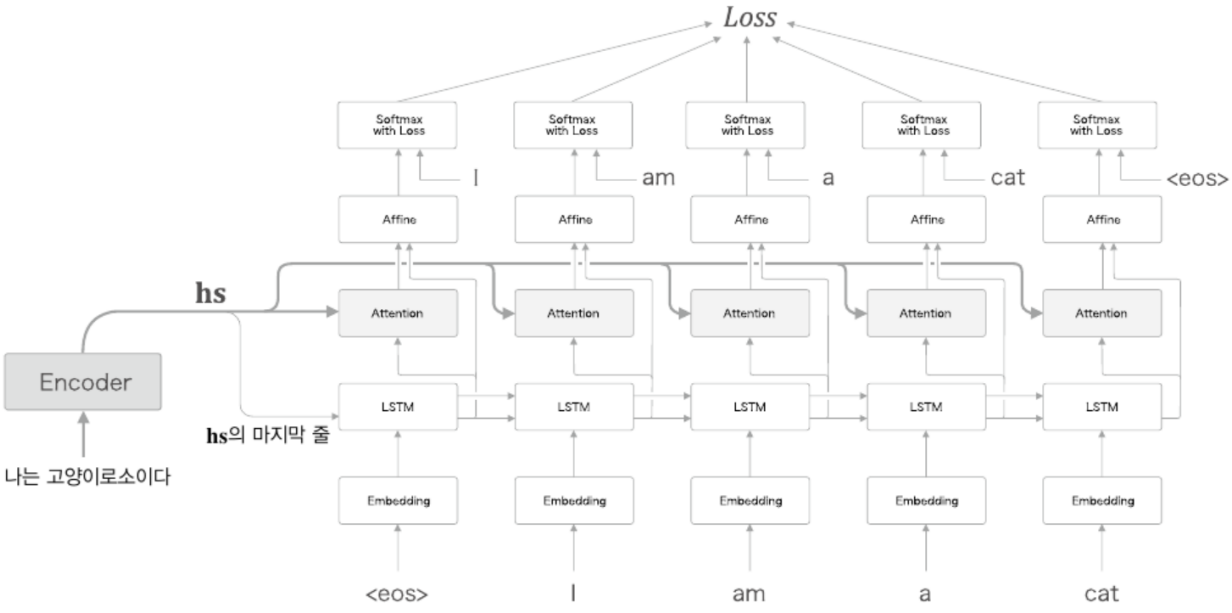


그림 8-18 Attention 계층을 갖춘 Decoder의 계층 구성



In [ ]:
```
class Attention:
    def __init__(self):
        self.params, self.grads = [], []
        self.attention_weight_layer = AttentionWeight()
        self.weight_sum_layer = WeightSum()
        self.attention_weight = None

    def forward(self, hs, h):
        a = self.attention_weight_layer.forward(hs, h)
        out = self.weight_sum_layer.forward(hs, a)
        self.attention_weight = a
        return out

    def backward(self, dout):
        dhs0, da = self.weight_sum_layer.backward(dout)
        dhs1, dh = self.attention_weight_layer(da)
        dhs = dhs0+dhs1
        return dhs, dh
```

In [16]:
```
class TimeAttention:
    def __init__(self):
        self.params, self.grads = [], []
        self.layers = None
        self.attention_weights = None

    def forward(self, hs_enc, hs_dec):
        """
        hs_enc : encoder에서 출력된 hs matrix
        hs_dec : decoder에서 출력된 hs matrix
        """
        N, T, H = hs_dec.shape
        out = np.empty_like(hs_dec) # 최종적으로 나와야하는 matrix 크기
        self.layers = []
        self.attention_weights = []

        for t in range(T):
            layer = Attention()
            out[:,t,:] = layer.forward(hs_enc, hs_dic[:,t,:])
            self.layers.append(layer)
            self.attention_weights.append(layer.attention_weight)

        return out

    def backward(self, dout):
        N, T, H = dout.shape
        dhs_enc = 0
        dhs_dec = np.empty_like(dout)

        for t in range(T):
            layer = self.layers[t]
            dhs, dh = layer.backward(dout[:,t,:])
            dhs_enc += dhs # 모두 같은 dhs가 사용되기 때문에 더해줌
            dhs_dec[:,t,:] = dh # 사용되는 dh가 다르기 때문에 각각 배정해줌

        return dhs_enc, dhs_dec
```

2. Attention을 갖춘 seq2seq 구현

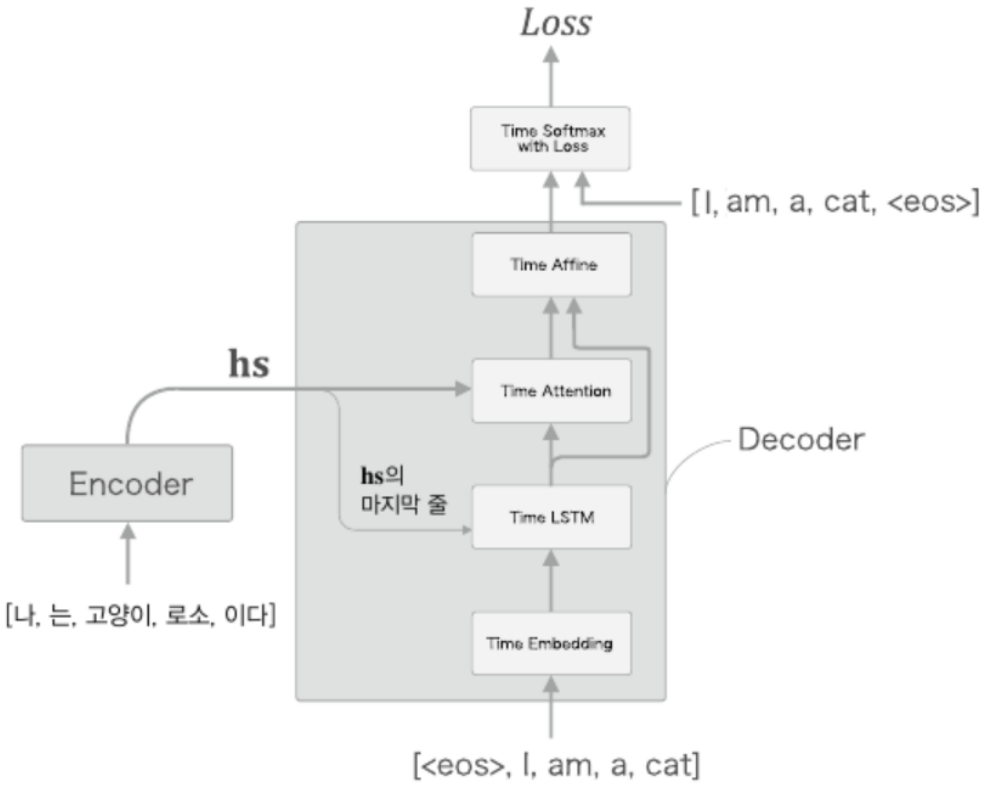
2.1 Encoder

- 기존의 Encoder에 비해 모든 은닉 상태를 반환

In [ ]:
```
import sys
import os
sys.path.append(os.path.join("..","master"))
sys.path.append(".")
import numpy as np
from common.time_layers import *
from seq2seq import Encoder, Seq2seq
from attention_layer import TimeAttention

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

    def backward(self, dhs):
        # 모든 hs에 대한 gradient가 존재
        dout = self.lstm.backward(dhs)
        dout = self.embed.backward(dout)
        return dout
```

2.2 Decoder

그림 8-21 Decoder의 계층 구성



In [1]:
```
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(D)).astype("f")
        lstm_b = np.zeros(4*H).astype("f")
        # weighted sum된 h를 concat해서 input으로 넣어주기 때문
        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() # attention layer가 추가됨
        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)
        # attention layer 추가 및 attention layer output concat
        c = self.attention.forward(enc_hs, dec_hs)
        out = np.concatenate((c, dec_hs), axis=2)
        score = self.affine.forward(out)

        return score

    def backward(self, dscore):
        dout = self.affine.backward(dscore)
        N,T,H2 = dout.shape
        H = H2//2

        dc, ddec_hs0 = dout[:,:,:H], dout[:,:,H:]
        dnec_hs, ddec_hs1 = self.attention.backward(dc)
        ddec_hs = ddec_hs0 + ddec_hs1
        dout = self.lstm.backward(ddec_hs)
        dh = self.lstm.dh
        dnec_hs[:,-1] += dh
        self.embed.backward(dout)

        return dnec_hs

    def generate(self, enc_hs, start_id, sample_size):
        sampled = []
        sample_id = start_id
        h = enc_hs[:,-1]
        self.lstm.set_state(h)

        for _ in range(sample_size):
```

```
            x = np.array([sample_id]).reshape((1,1)) # batch 처럼 넣어줌

            out = self.embed.forward(x)
            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)

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

        return sampled
```

2.3 Seq2seq

In [ ]:
```
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
```

3. Attention 평가

- '날짜 형식'을 변경하는 문제를 통해 평가
  - 다양한 변형이 존재하기 때문에 변환 규칙이 복잡하여 수작업으로 모두 써내려가기 어려움
  - 알기 쉬운 대응관계에 있음
  - 입력과 출력의 구분문자로 '_' 사용
  - 출력의 문자 수가 일정하기 때무에 출력의 끝을 알리는 구분문자는 사용하지 않음.

In [1]:

```
import sys
sys.path.append("../function")
sys.path.append("../master")
import numpy as np
from dataset import sequence
from common.optimizer import Adam
from common.trainer import Trainer
from common.util import eval_seq2seq
from attention_seq2seq import AttentionSeq2seq
from seq2seq import Seq2seq
from peeky_seq2seq import PeekySeq2seq

# 데이터 읽기
(x_train, t_train), (x_test, t_test) = sequence.load_data("date.txt")
char_to_id, id_to_char = sequence.get_vocab()

# 입력 문장 반전
x_train, x_test = x_train[:,::-1], x_test[:,::-1]

# 하이퍼파라미터 설정
vocab_size = len(char_to_id)
wordvec_size = 16
hidden_size = 256
batch_size = 128
max_epoch = 10
max_grad = 5.0

model = AttentionSeq2seq(vocab_size, wordvec_size, hidden_size)
optimizer = Adam()
trainer = Trainer(model, optimizer)

acc_list = []
for epoch in range(max_epoch):
    trainer.fit(x_train, t_train, max_epoch=1,
                batch_size=batch_size, max_grad=max_grad)
    correct_num = 0
    for i in range(len(x_test)):
        question,correct = x_test[[i]], t_test[[i]]
        verbose = i<10 # 10개만 출력
        correct_num += eval_seq2seq(model, question, correct,
                                    id_to_char, verbose, is_reverse=True)

    acc = float(correct_num)/len(x_test)
    acc_list.append(acc)
    print('val acc %.3f%%'%(acc*100))

model.save_params()
```

```
---
val acc 8.940%
| 에폭 3 |  반복 1 / 351 | 시간 1[s] | 손실 0.54
| 에폭 3 |  반복 21 / 351 | 시간 25[s] | 손실 0.51
| 에폭 3 |  반복 41 / 351 | 시간 47[s] | 손실 0.42
| 에폭 3 |  반복 61 / 351 | 시간 71[s] | 손실 0.30
| 에폭 3 |  반복 81 / 351 | 시간 94[s] | 손실 0.21
| 에폭 3 |  반복 101 / 351 | 시간 117[s] | 손실 0.14
| 에폭 3 |  반복 121 / 351 | 시간 142[s] | 손실 0.10
| 에폭 3 |  반복 141 / 351 | 시간 165[s] | 손실 0.07
| 에폭 3 |  반복 161 / 351 | 시간 189[s] | 손실 0.05
```

```
| 에폭 3 |  반복 181 / 351 | 시간 211[s] | 손실 0.04

| 에폭 3 |  반복 201 / 351 | 시간 234[s] | 손실 0.03
| 에폭 3 |  반복 221 / 351 | 시간 257[s] | 손실 0.03
| 에폭 3 |  반복 241 / 351 | 시간 279[s] | 손실 0.02
| 에폭 3 |  반복 261 / 351 | 시간 299[s] | 손실 0.02
| 에폭 3 |  반복 281 / 351 | 시간 317[s] | 손실 0.01
| 에폭 3 |  반복 301 / 351 | 시간 333[s] | 손실 0.01
| 에폭 3 |  반복 321 / 351 | 시간 350[s] | 손실 0.01
```

In [6]:

```
from matplotlib import font_manager, rc
import matplotlib as mpl
import matplotlib.pyplot as plt
font_path = "C:WWUsersWW이혜림WWDesktopWWBita5/malgun.ttf"
font_name = font_manager.FontProperties(fname=font_path).get_name()
rc("font",family=font_name)
mpl.rcParams["axes.unicode_minus"]=False

# 그래프 그리기
x = np.arange(len(acc_list))
plt.plot(x, acc_list, marker = "o")
plt.xlabel("에폭")
plt.ylabel("정확도")
plt.ylim(0,1.3)
plt.show()
```

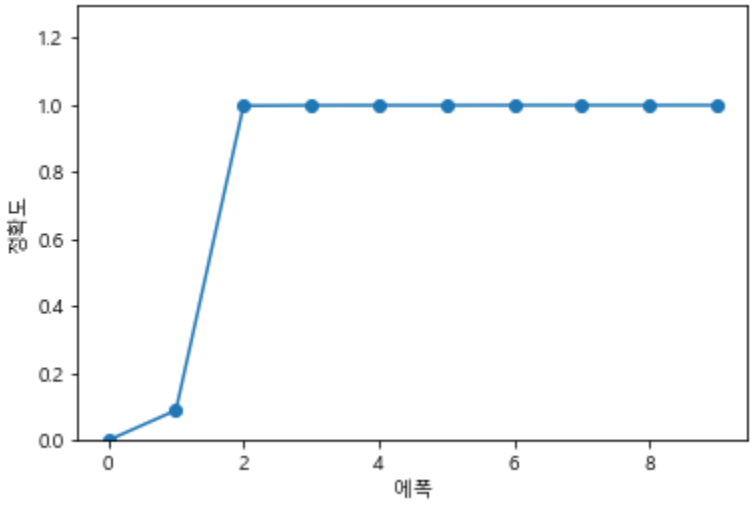


그림 8-27 학습된 모델을 사용하여 시계열 변환을 수행했을 때의 어텐션 가중치 시각화: 가로축은 입력 문장, 세로축은 출력 문장. 맵의 각 원소는 밝을수록 값이 크다(1.0에 가깝다).

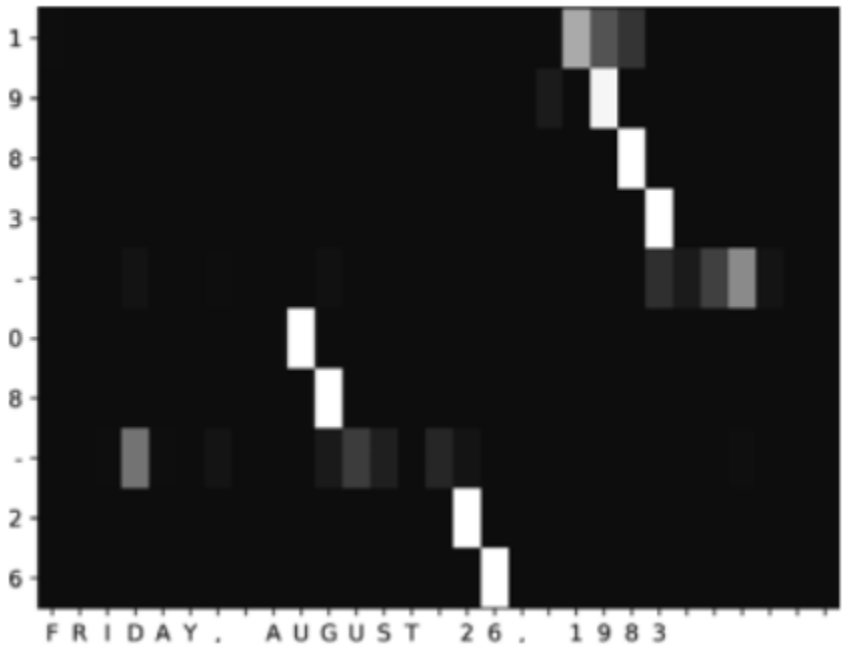
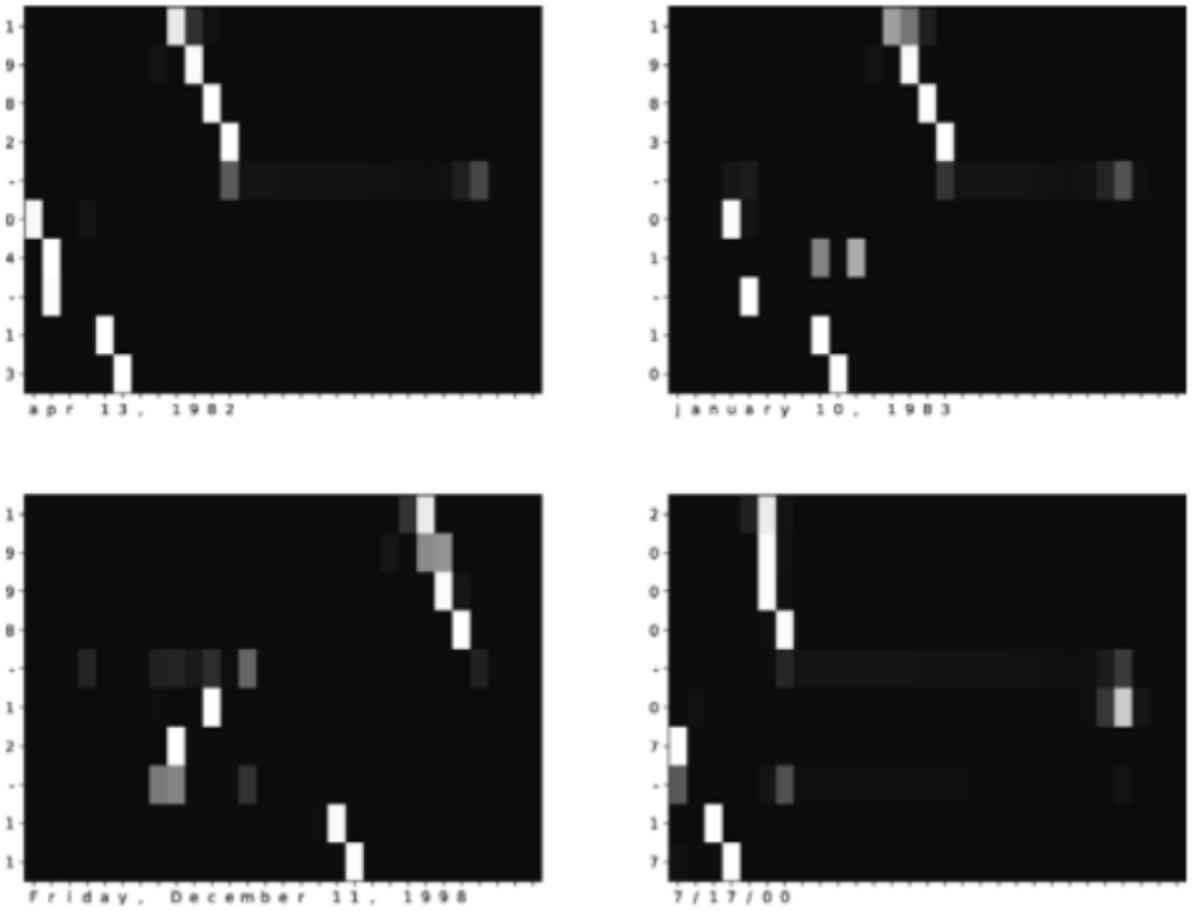


그림 8-28 어텐션 가중치를 시각화한 예



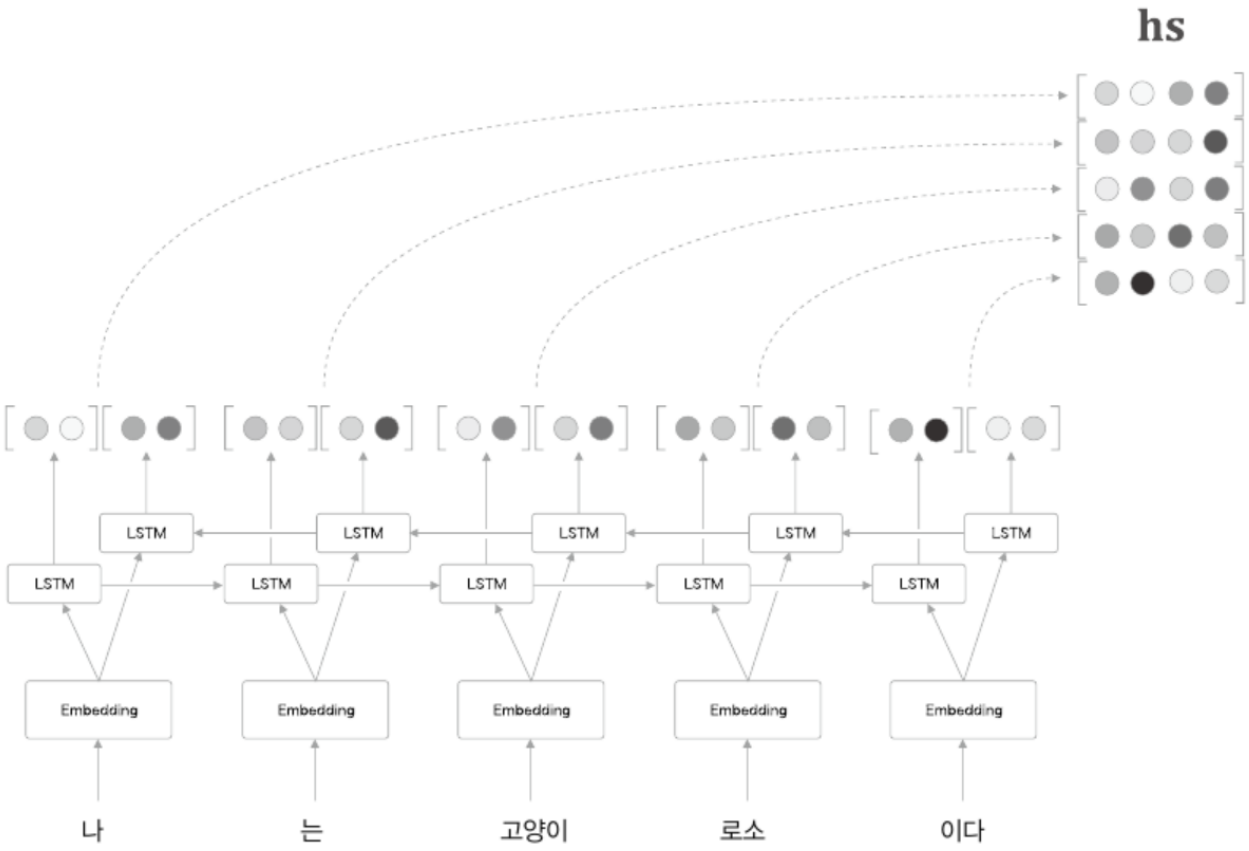
- Attention을 통해 모델이 수행하는 작업을 인간이 이해할 수 있게 되었다고도 생각할 수 있음

4. Attention에 관한 남은 이야기

4.1 양방향 RNN

- 이제까지는 왼쪽에서 오른쪽으로 정보를 전해줬다면, 오른쪽에서 왼쪽으로도 정보를 전해줌으로써 '주변'정보를 균형있게 encoding함
- ex) 'A B C D', 'D C B A'(reversed) 두 방향의 정보를 넣어줌

그림 8-30 양방향 LSTM으로 인코딩하는 예(LSTM 계층을 간략화하여 그림)

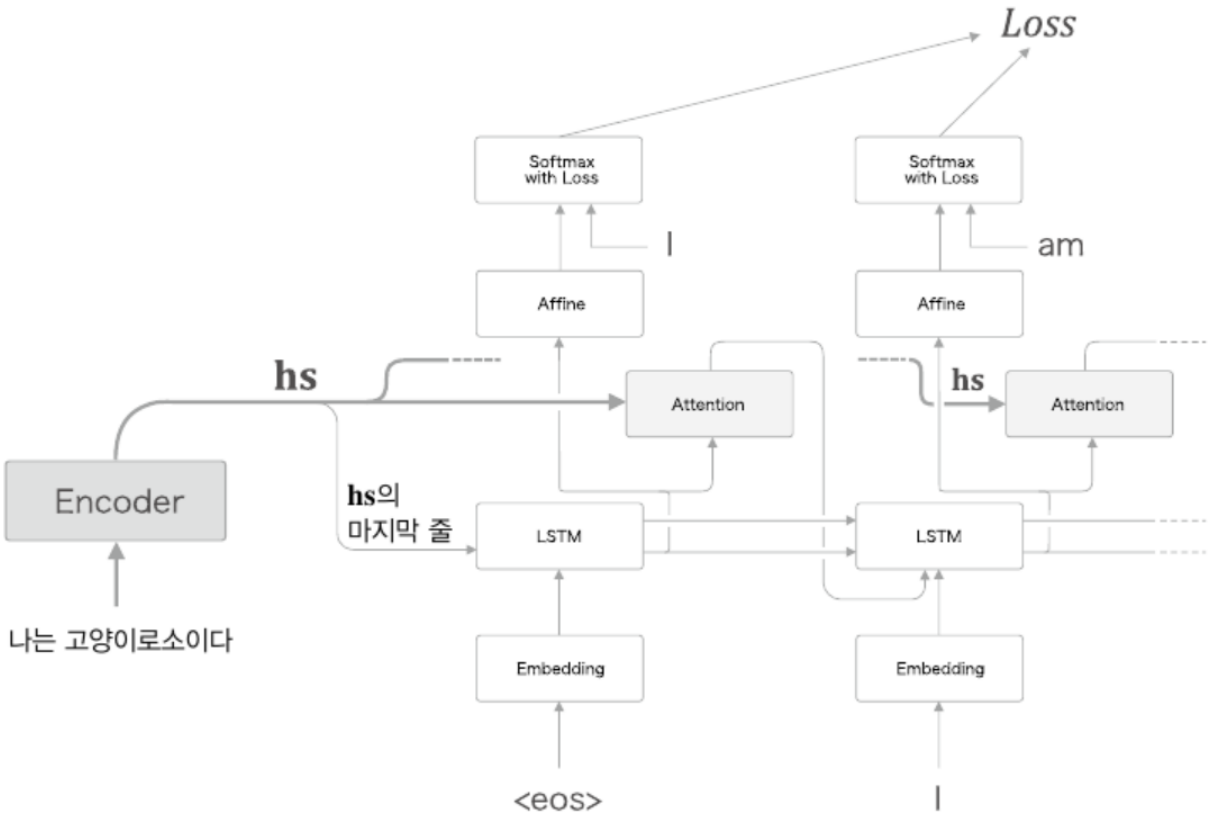


- 두 방향의 LSTM의 output을 'concat(연결)'하는 방법 외에도, '합', '평균'등의 방법이 있음

4.2 Attention 계층 사용 방법

- 기존 LSTM과 Affine 계층 사이에 Attention layer을 넣는 방법 외에도 다양한 방법이 있음
- Attention layer의 출력이 다음 시각의 LSTM layer에 입력되도록 만들 수도 있음
  - 즉 LSTM 계층의 맥락 벡터의 정보를 이용할 수 있음
  - 성능은 실제 데이터에 적용해보아야 알 수 있음
  - 구현의 관점에서는 아래에서 위로 이어지는 구조인 LSTM과 Affine 계층 사이에 삽입하는 것이 편함

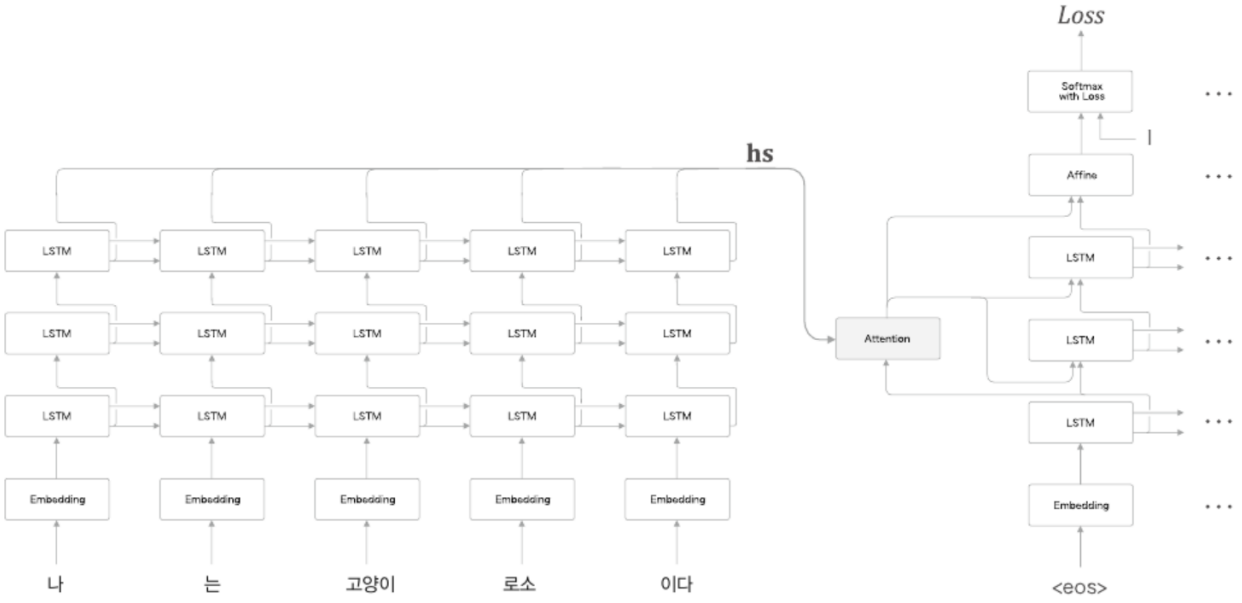
그림 8-32 Attention 계층의 다른 사용 예(문헌 [48]을 참고하여 단순화한 신경망 구성)



4.3 seq2seq 심층화와 skip 연결

- seq2seq 모델에 더 높은 표현력을 제공하기 위해 RNN(LSTM 계층)을 깊게 쌓음
  - 다양한 방법이 있을 수 있음. 그림의 예에서는 Attention layer의 맥락벡터를 LSTM layer와 Affine layer로 전파
  - overfitting의 문제가 있을 수 있기 때문에, Dropout 혹은 가중치 공유 등의 기술이 효과적
  - 기울기 소실/폭발의 문제가 있을 수 있음

그림 8-33 3층 LSTM 계층을 사용한 어텐션을 갖춘 seq2seq

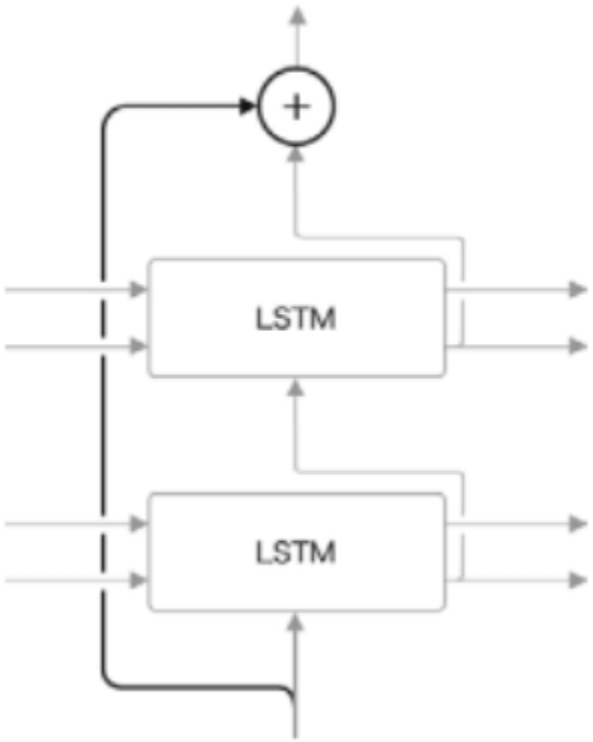


Skip 연결(skip connection)

- 'residual connection(잔차연결)' or 'short-cut(숏컷)' 이라고도 불림
- '계층을 건너뛰는 연결'
- 출력 2개를 합함으로써 층이 깊어져도 기울기가 소실(혹은 폭발)되지 않음
  - 덧셈연산은 기울기를 '그대로 흘려'보내기 때문. 보존된 기울기가 여러 곳에 전파됨으로써 좋은 학습을 기대할 수 있음

- 시간방향의 기울기 소실*(폭발)*
  - 게이트가 달린 RNN(LSTN, GRU)
- 깊이방향의 기울기 소실*(폭발)*
  - Skip연결

그림 8-34 LSTM 계층의 skip 연결 예



5. Attention 응용

- Attention은 seq2seq외에도 다양한 범용적인 문제에 적용 가능

5.1 구글 신경망 기계 번역(GNMT)

- **Neural Machine Translation(NMT)** 신경망 기계번역이 주목받고 있음
  - '규칙 기반 번역'에서 '용례 기반 번역'으로, 다시 '통계 기반 버역'으로 옮겨왔음
- **구글 신경망 기계 번역(Google Neural Machine Translation, GNMT)**
  - LSTM 계층의 다층화, 양방향 LSTM(Encoder의 첫번째 계층만), skip 연결 등의 개선이 이루어진 seq2seq model
  - 다수의 GRU로 분산학습함으로써 학습 시간 단축

- 번역 품질을 크게 끌어올리는 데 성공

5.2 트랜스포머(Transformer)

*RNN*의 한계점

- RNN의 병렬 처리 불가
  - RNN은 이전 시각에 계산한 결과를 이용하여 순서대로 계산. 따라서 RNN의 계산을 시간 방향으로 병렬 계산하기란 불가능. 보통 GRU를 사용한 병렬 계산 환경에서 학습이 이루어지는 딥러닝 학습에서 이는 큰 단점
- 따라서, RNN을 없애는 연구 / 병렬 계산 가능 RNN 연구가 활발히 진행
  - transformer
  - CNN을 이용한 seq2seq

5.2.1 트랜스포머(Transformer)의 장점

- 성능을 개선
- 훈련 속도가 빠르고, 병렬화하기 쉬움

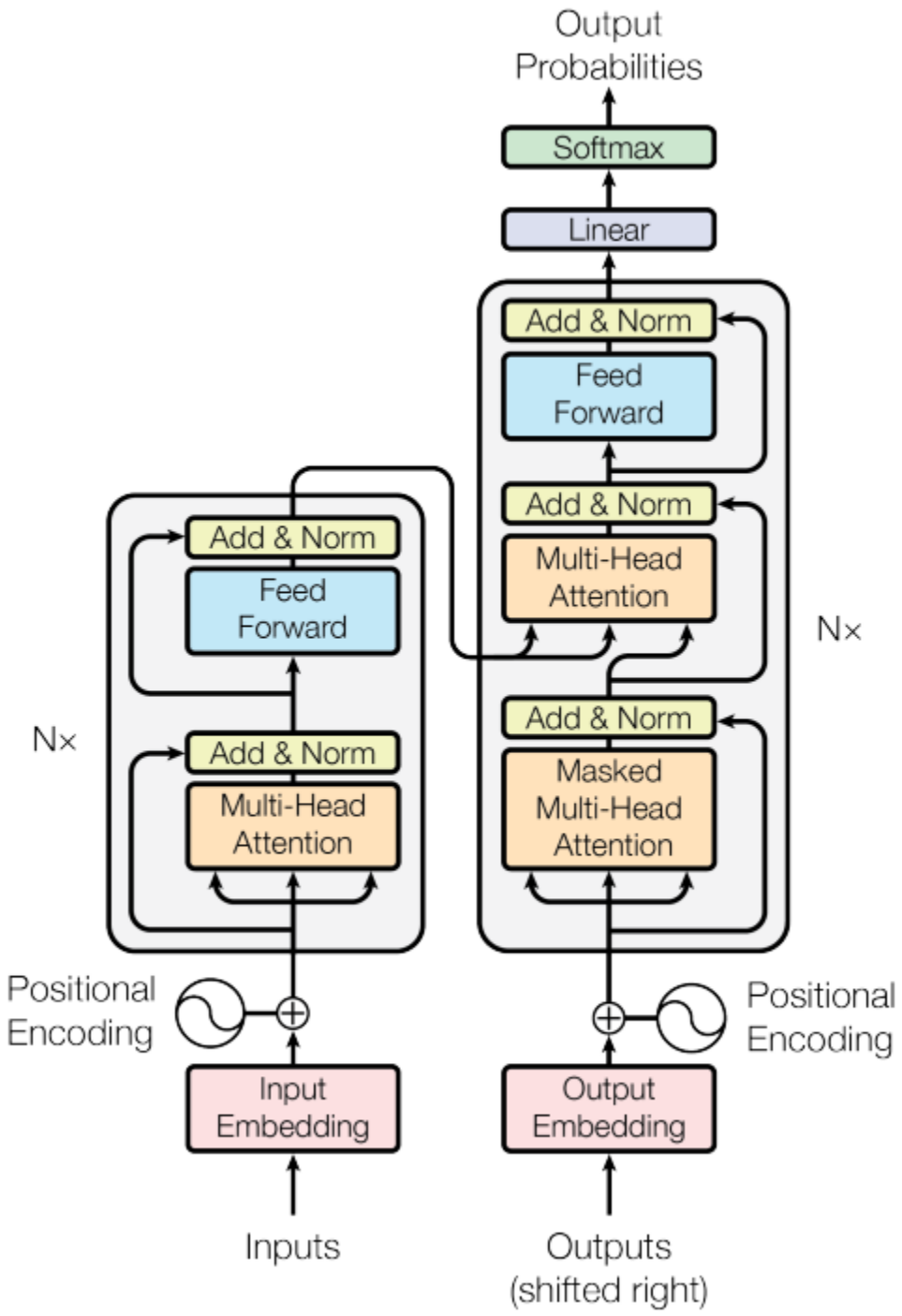
5.2.2 트랜스포머의 기본 구조

- 큰 틀 : Encoder, Decoder
- **셀프어텐션(self-Attention)** 기술을 이용
  - 하나의 시계열 데이터를 대상으로 한 어텐션. '하나의 시계열 데이터 내에서' 각 원소가 다른 원소들과 어떻게 관련되는지를 살핌
  - 문장 자기 자신에 주의를 기울임
  - 멀티 헤드 어텐션을 통해 수행
- **위치 인코딩을 제외한 모든 층은 타임 스텝에 독립적**
  - 오직 위치 인코딩을 통해서만 상대적/절대적 위치가 전해짐
- 전체적인 구조
  - **Encoder**

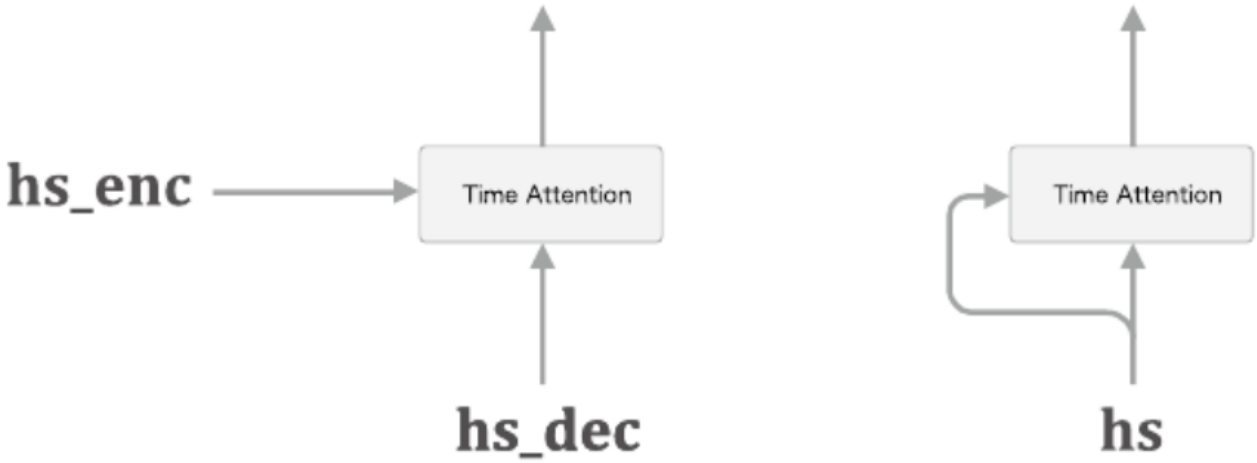
    1. Input Embedding
    2. Positional Encoding결과를 더함
    3. Multi-Head Attention
    4. 2의 output과의 residual connection & Normalize
    5. Feed Forward
    6. 4의 output과의 residual connection & Normalize
    7. 3~6의 과정을 N번(여기선 6번) 반복
  - **Decoder**

    1. Output Embedding
    2. Positional Encoding결과를 더함
    3. Masked Multi-Head Attention
    4. 2의 output과의 redidual connection & Normalize
    5. Multi-Head Attention(Encoding output을 사용)
    6. 4의 output과의 residual connection & Normalize

7. Feed Forward
8. 6의 output과의 residual connection & Normalize
9. 3~7의 과정을 N번(여기선 6번)반복
10. Linear
11. Softmax



셀프 어텐션

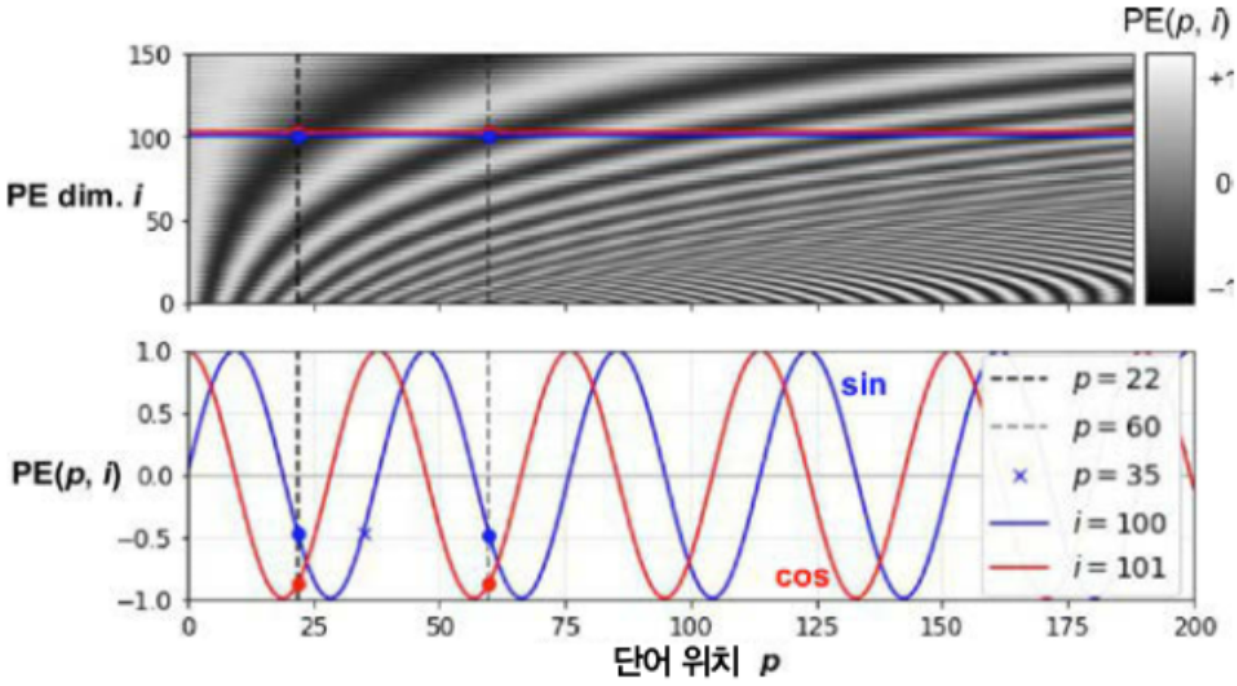


위치 인코딩(positional encoding)

- 문장 안에 있는 단어의 위치를 인코딩한 밀집벡터
- n번째 위치 인코딩이 각 문장에 있는 n번째 단어의 단어 임베딩에 더해짐
- i번째의 위치 인코딩이 i번째 단어의 단어 임베딩에 더해짐
- 모델을 통해 학습할 수도, sin, cos함수로 정의한 고정된 위치 인코딩을 할 수도 있음

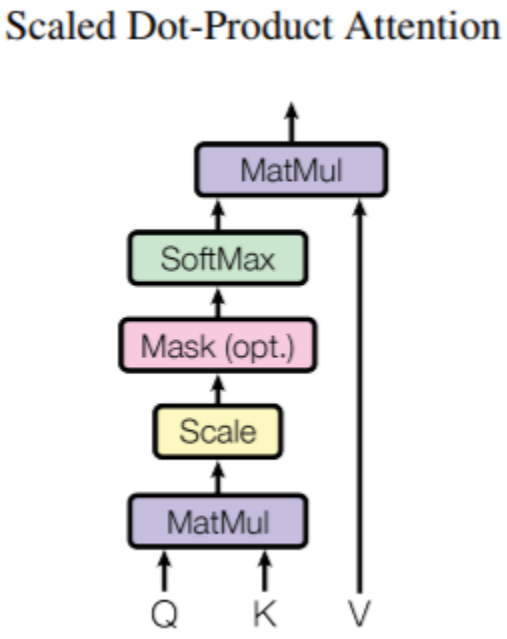
고정 위치 인코딩

- 학습된 위치 인코딩과 동일한 성능을 내면서 임의의 긴 문장으로 확장 가능
- **절대적 위치를 알 수 있음**
  - i에 따라 sin, cos의 주기가 달라지기 때문에(i가 커질수록 파장이 커짐) 각 위치마다 고유한 vector가 만들어짐
  - 따라서 위치 인코딩을 단어 임베딩에 더하면 모델이 문장에 있는 단어의 절대 위치를 알 수 있음.
- **상대적 위치를 알 수 있음**
  - **같은 주기의 sin, cos함수를 사용함으로써** : 모델이 문장에 있는 단어의 상대 위치 또한 알 수 있음. 예를 들어 그래프에서 볼 수 있듯이 38개 단어만큼 떨어진 두 단어(ex p=22, p=60)는 위치 인코딩 차원 i=100, i=101에서 항상 같은 위치 인코딩 값을 가짐. 즉, 아, i=100, i=101에서 값이 같다면, '아! 이 단어들은 38개만큼 떨어진 단어이구나'를 알 수 있음. 따라서 같은 주기의 사인과 코사인 함수를 사용해야 함.(만약 주기가 다르다면 동일한 주기 마다 i=100, i=101의 원소에서 값이 같지 않을 것임.)
  - **sin, cos 함수를 모두 이용함으로써** : 둘 중 하나의 함수만 사용하면 모델이 p=22과 p=35의 위치를 구별할 수 없음.
- 파라미터
  - p : 해당 word의 순서
  - d : encoding vector의 크기(column)
  - $P_{p,2i}$: 각 위치 p에서의 2xi번째 원소 값
- 위치 인코딩 하는 방법(같은 주기의 sin, cos 함수를 사용해야 함)
  - $P_{p,2i} = sin(p/10000^{2i/d})$
  - $P_{p,2i+1} = cos(p/10000^{2i/d})$



스케일드 점-곱 어텐션*(scaled dot-product attention)*

- Attention에 스케일링 인자를 추가한 것
  - d_{keys}를 나눔으로써 softmax function이 포화되지 않도록 함
- 파라미터
  - Q : 영향을 받는 인자(query)
    - $[n_{queries}, d_{keys}]$
  - K : 영향을 주는 인자(key)
    - $[n_{keys}, d_{keys}]$
  - V : 영향을 주는 인자의 값
    - $[n_{keys}, d_{values}]$

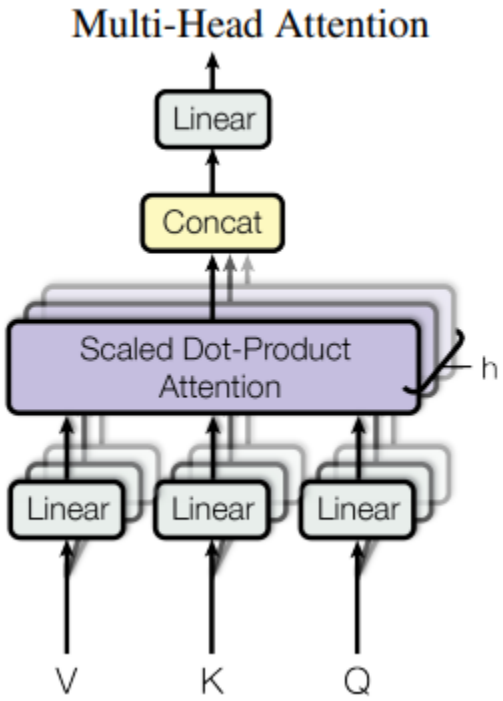


- Attention 계산
  - $Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_{keys}}})V$

    - QK^T : 영향을 받는 word와 영향을 주는 word 간의 가중치
        - $[n_{queries}, n_{keys}]$
    - 최종 출력 : $[n_{queries}, d_{values}]$
- 마스크 계층
    - decoder에서만 사용
    - 정보가 왼쪽으로 가는 흐름을 막기 위해서 사용. i번째 output을 i+1번째 input으로 사용하는 auto-regressive한 특성을 유지하기 위함
    - 즉, **각 단어는 이전에 등장한 단어에만 주의를 기울일 수 있음**
    - softmax이전에 흐름에 $-\infty$ 으로 마스킹

멀티-헤드 어텐션(multi-head attention)

- 관련이 많은 단어에 더 많은 주의를 기울이면서 각 단어와 동일한 문장에 있는 다른 단어의 관계를 인코딩.(decoder의 Multi-Head Attention의 경우 encoder의 output인 입력문장과의 관계를 인코딩)
    - ex) 'They welcomed the Queen of the United Kingdom'과 같은 문장이 있다면 단어 Queen에 대해 이 층의 출력은 모든 단어에 의존하겠지만, 특히 United와 Kingdom에 더 주의를 기울일 것.
- scaled dot-product attention을 여러 층(6층)을 병렬적으로 쌓은 것
    - 이를 통해 모델이 단어 표현을 여러 부분 공간(subspace)로 다양하게 투영할 수 있음. 각 부분 공간은 단어의 일부 특징에 주목. 정보를 다양하게 표현할 수 있게 되면서 성능이 향상
    - 따라서 각 6층의 weight가 다르게 학습됨
- multi-head attention 계산
    - $MultiHead(Q, K, V) = Concat(head_1, \dots, head_h)W^o)$
        - where $head_i = Attention(QW_i^Q, KW_i^K, VW_i^v)$
        - $W_i^Q \in R^{d_{model} \times d_{keys}}, W_i^K \in R^{d_{model} \times d_{keys}}, W_i^V \in R^{d_{model} \times d_v}, W_i^O \in R^{hd_v \times d_{model}}$
        - d_{model}은 embedding vector와 feature 개수가 같음(여기선 512)
        - $d_v = d_h = d_{model}/6 = 64$



Position-wise Feed-Forward Networks

- 중간에 ReLU 활성화 함수가 있는 두 개의 선형변환으로 구성
- $FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$
- 모든 position에 같은 weight를 사용하지만, layer마다 다른 weight를 사용
  - W_1, b_1, W_2, b_2는 position에 무관하게 동일하게 적용되지만, W_1, b_1이 각각 W_2, b_2와 같지는 않음
  - 즉, kernel size가 1인 convolutions layer을 2번 사용하는 것과 같음
    - d_{ff} : 2048, (W_1 의 차원)
    - W_2의 차원 : 512
    - ff layer에 입력되는 data의 차원 : 512

가중치 공유 기법을 사용

- Embedding layer Weight와 output의 linear transformation layer Weight 간에 같은 가중치 matrix를 공유(학습된 embedding matrix를 사용)
- Embedding layer에는 $\sqrt{d_{model}}$을 곱해줌
- 효과
  - 파라미터의 개수가 줄어들면서 학습할 매개변수의 수를 줄일 수 있음
    - 학습하기가 더 쉬워짐
    - overfitting 방지 효과

최적화

- Adam을 사용
  - $\beta_1 = 0.9, \beta_2 = 0.98, \epsilon = 10^{-9}$
- learning rate를 학습동안 변화시킴
- $lrate = d_{model}^{-0.5} \cdot min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$
  - warmup_step까지는 linear하게 ($warmup\_steps^{-1.5}$ 씩만큼 learning rate를 증가시키다가 warmup_step 이후에는 step_num의 inverse square root에 비례하도록 감소시킴(사용 warmup_step = 4000)

Regularization

1. Residual Dropout

- 각 sub-layer의 output에 dropout을 적용하고, sub-layer input에 추가하고 normalized한다.
- 각 stack의 embedding 및 positional encoding의 합에 dropout을 적용. (dropout 비율은 0.1로 설정)

2. Label Smooting

- 훈련하는 동안, 라벨 스무딩을 적용
- 이것을 통해 모델이 불확실함을 학습함.
- 정확성과 BLEU 점수를 향상시킴
- $\epsilon_{ls} = 0.1$

결론

- recurrence를 이용하지 않고 encoder와 decoder에서 multi-headed self-attention을 이용하여 sequential data를 처리할 수 있는 model
- recurrent or convolutional layers를 이용한 구조보다 훨씬 **빠르고**, **정확**

5.3 뉴럴 튜닝 머신(NTM)

- 인간이 복잡한 문제를 풀 때 '종이'와 '펜'을 사용하는 것처럼 **'외부 메모리를 통한 확장'** 을 통해 성능을 향상
- 외부 메모리에 필요한 정보를 두고 적절하게 기록하고 필요할 때 꺼내서 사용
  - RNN/LSTM은 내부 상태를 입력할 때 hidden state 길이가 고정되어 있기 때문에 채워 넣을 수 있는 정보량이 제한적.
  - Attention을 이용해서 외부 메모리로부터 필요한 정보를 읽거나 씀
  - 각 node가 필요한 데이터를 쓰고 읽을 수 있음
  - 데이터의 입출력으로부터 '알고리즘을 학습' 할 수 있음
- LSTM 계층 : '컨트롤러', Write head : LSTM의 hidden state를 Write Head layer가 받아서 필요한 정보를 메모리에 씀, Read Head layer : 메모리로부터 중요한 정보를 읽어 들여 다음 시각의 LSTM layer로 전달
- Write Head & Read Head layer : Attention을 사용
  - Content-based Attention : 일반적인 Attention과 같음. 입력으로 주어진 벡터와 비슷한 벡터를 메모리로부터 찾아내는 것
  - Position-based Attention : 이전 시각에서 주목한 메모리의 위치(=메모리의 각 위치에 대한 가중치)를 기준으로 그 전후 이동(시프트)하는 용도로 사용
    - 1차원의 합성곱 연산으로 구현
    - 메모리를 위치를 하나씩 옮겨가며 읽어 나가는 컴퓨터 특유의 움직임을 재현
  - 그 이외에도 Attention의 가중치를 날카롭게 다듬는 처리 / 이전 시각 Attention의 가중치를 더해주는 처리 등이 이루어짐

그림 8-41 NTM의 계층 구성: 메모리 쓰기와 읽기를 수행하는 Write Head 계층과 Read Head 계층이 새로 등장