

목차 3

1. 신경망 복습
2. 자연어와 단어의 분산 표현
3. word2vec
4. word2vec 속도 개선
5. 순환 신경망(RNN)
6. 게이트가 추가된 RNN
7. RNN을 사용한 문장 생성
8. 어텐션
9. 트랜스 포머 모델

1. 신경망 복습

1.1 수학과 파이썬 복습

1.2 신경망 추론

1.3 신경망의 학습

1.4 신경망으로 문제를 풀다.

벡터와 행렬의 예

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

행

열

벡터의 표현법

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

열벡터

행벡터

브로드캐스트의 예1 : 스칼라 값인 10이 2×2 행렬로 처리된다.

$$\begin{array}{c|c}
 1 & 2 \\
 \hline
 3 & 4
 \end{array} * 10 =
 \begin{array}{c|c}
 1 & 2 \\
 \hline
 3 & 4
 \end{array} *
 \begin{array}{c|c}
 10 & 10 \\
 \hline
 10 & 10
 \end{array} =
 \begin{array}{c|c}
 10 & 20 \\
 \hline
 30 & 40
 \end{array}$$

$$(2,2) \quad () \quad (2,2) \quad (2,2)$$

브로드캐스트의 예2 : 일차원 배열인 10,20이 2×2 행렬로 처리된다.

$$\begin{array}{c|c}
 1 & 2 \\
 \hline
 3 & 4
 \end{array} *
 \begin{array}{c|c}
 10 & 20
 \end{array} =
 \begin{array}{c|c}
 1 & 2 \\
 \hline
 3 & 4
 \end{array} *
 \begin{array}{c|c}
 10 & 20 \\
 \hline
 10 & 20
 \end{array} =
 \begin{array}{c|c}
 10 & 40 \\
 \hline
 30 & 80
 \end{array}$$

$$(2,2) \quad (2,) \quad (2,2) \quad (\textcolor{red}{2},2)$$

행렬의 곱

$$\begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline \end{array} \cdot \begin{array}{|c|c|c|} \hline 4 & 5 & 6 \\ \hline \end{array} = 1*4 + 2*5 + 3*6 = 32$$

$(3,)$ $(3,)$ $()$

$$\begin{array}{|c|c|} \hline 1 & 2 \\ \hline \end{array} \cdot \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 2 & 2 & 2 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 5 & 5 & 5 \\ \hline \end{array}$$

$(2,)$ $(2,3)$ $(3,)$

$$\begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 2 & 2 & 2 \\ \hline \end{array} \cdot \begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 6 & 12 \\ \hline \end{array}$$

$(2,3)$ $(3,)$ $(2,)$

벡터의 내적

$$x \cdot y = x_1y_1 + x_2y_2 + \cdots + x_ny_n$$

행렬의 곱

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1 * 5 + 2 * 7 \\ 3 * 5 + 4 * 7 \end{bmatrix} \begin{bmatrix} 1 * 6 + 2 * 8 \\ 3 * 6 + 4 * 8 \end{bmatrix}$$

A
B

$(2, 2)$
 $(2, 2)$
 $(2, 2)$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

형상 확인 : 행렬의 곱에서는 대응하는 차원의 원소 수를 일치시킨다.

$$\begin{array}{c} A \cdot B = C \\ \\ \begin{array}{c} 3 \times 2 \quad 2 \times 4 \quad 3 \times 4 \\ \hline \text{---} \quad \text{---} \quad \text{---} \\ | \quad | \quad | \\ \hline \end{array} \end{array}$$

일치시킨다.

1. 신경망 복습

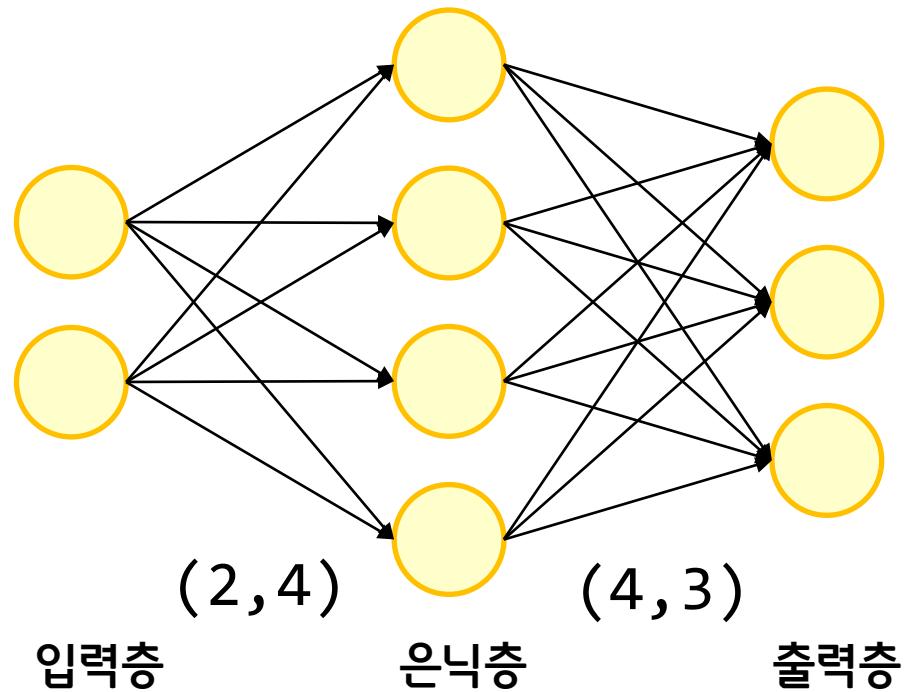
1.1 수학과 파이썬 복습

1.2 신경망 추론

1.3 신경망의 학습

1.4 신경망으로 문제를 풀다.

신경망의 예



신경망이 수행하는 계산 수식(단일층)

$$h_1 = x_1 w_{11} + x_2 w_{21} + b_1$$

신경망이 수행하는 계산 수식(여러층)

$$(1, 2)(2, 4) + (4,)$$

$$(h_1, h_2, h_3, h_4) = (x_1, x_2) \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{pmatrix} + (b_1, b_2, b_3, b_4)$$



$$h = xW + b$$

형상 확인 : 대응하는 차원의 원소 수가 일치함(편향은 생략)

$$A \cdot B = C$$

$$\frac{3 \times 2}{\text{---}} \quad \frac{2 \times 4}{\text{---}} \quad \frac{3 \times 4}{\text{---}}$$

일치시킨다.

형상 확인 : 미니배치 버전의 행렬 곱(편향은 생략)

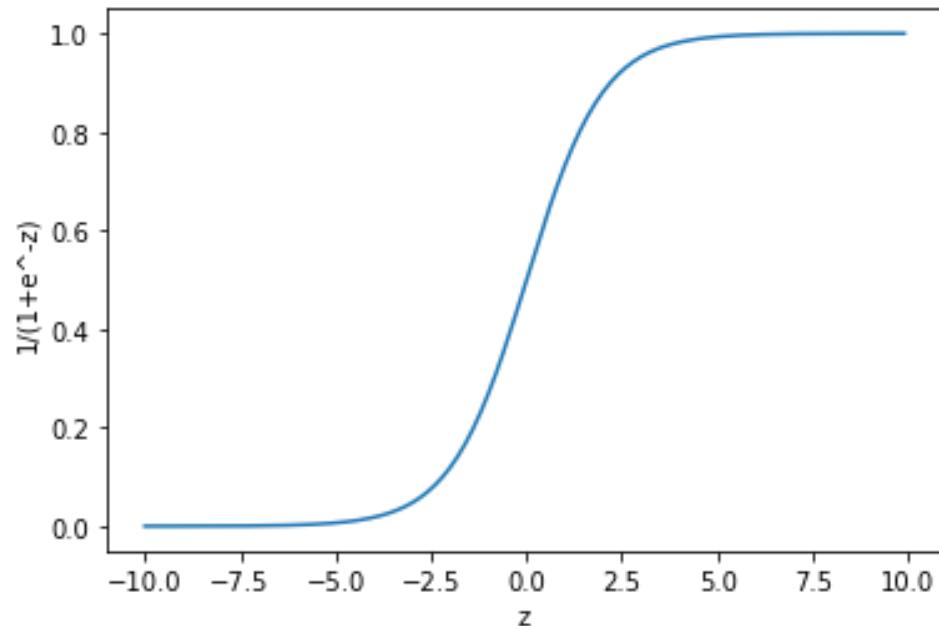
$$x \cdot w = h$$

$$\frac{N \times 2}{\text{---}} \quad \frac{2 \times 4}{\text{---}} \quad \frac{N \times 4}{\text{---}}$$

일치시킨다.

시그모이드 함수

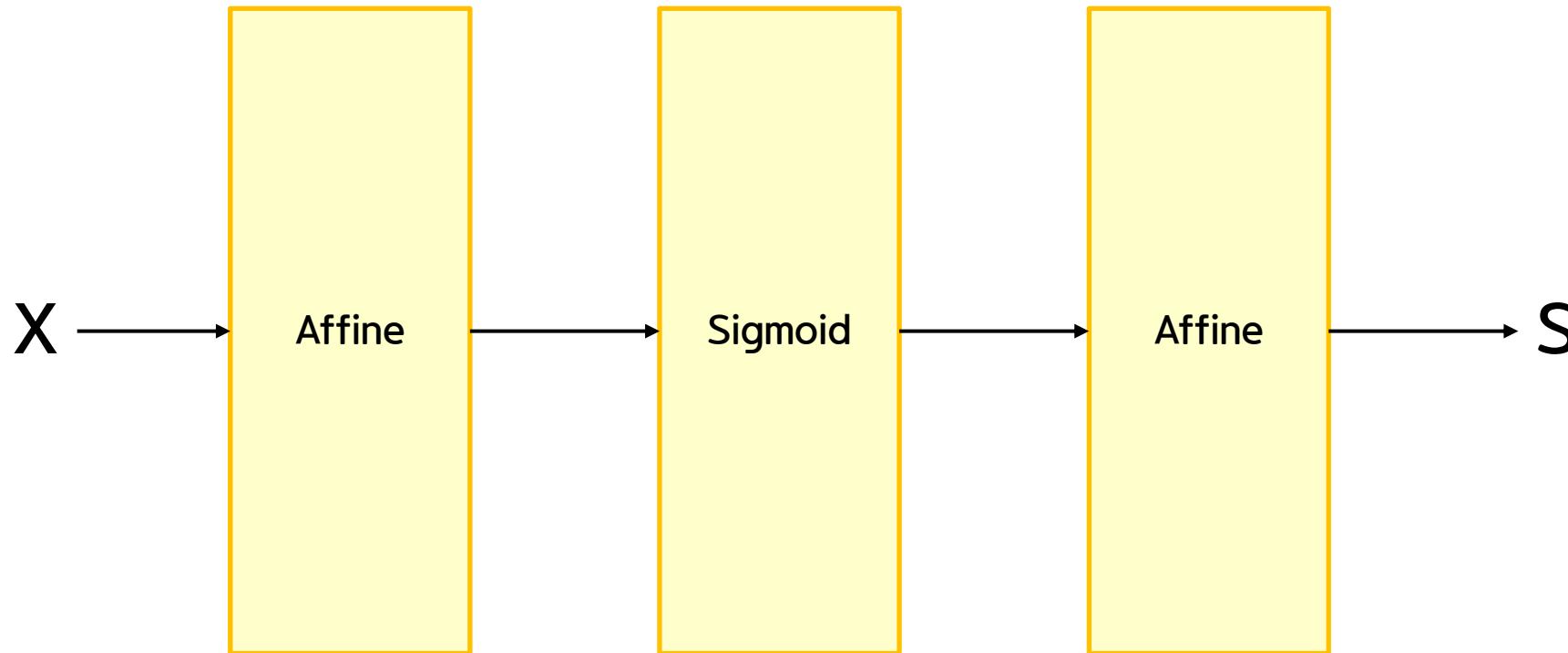
$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$



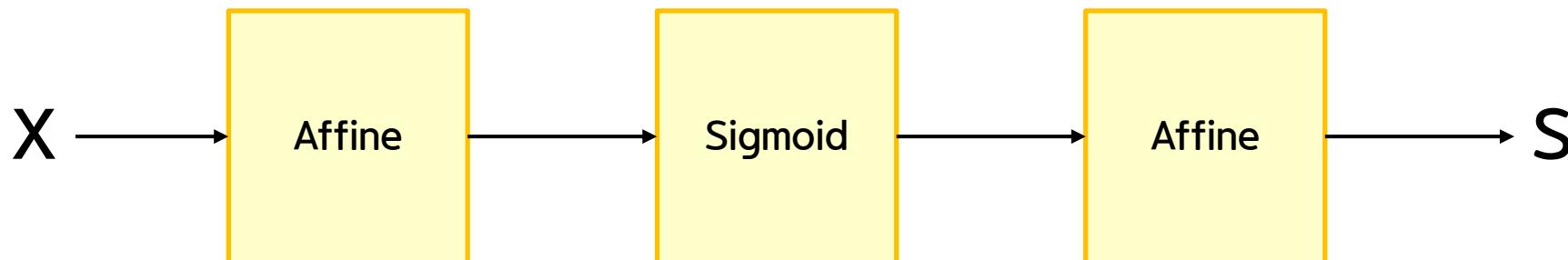
- 모든 계층은 forward()와 backward() 메서드를 가진다.
- 모든 계층은 인스턴스 변수인 param와 grads를 가진다.

※ 소스 참조

구현해볼 신경망의 계층 구성



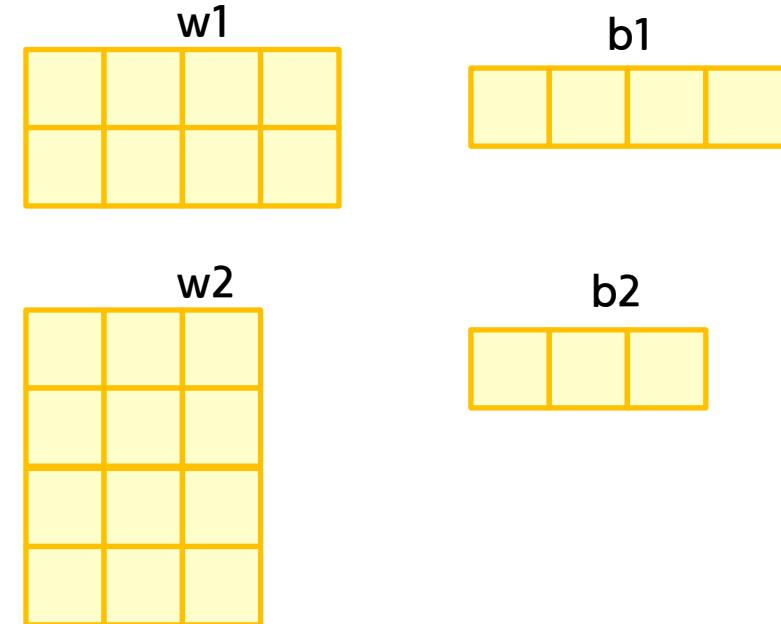
구현해볼 신경망의 계층 구성



```
class TwoLayerNet:  
    def __init__(self, input_size, hidden_size, output_size):  
        I, H, O = input_size, hidden_size, output_size  
  
        # 가중치와 편향 초기화  
        W1 = np.random.randn(I, H)  
        b1 = np.random.randn(H)  
        W2 = np.random.randn(H, O)  
        b2 = np.random.randn(O)  
  
    model = TwoLayerNet(2, 4, 3)
```

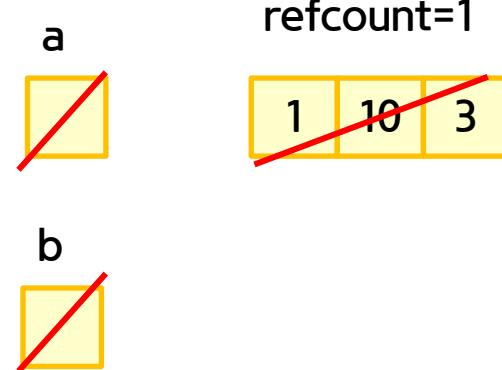
구현해볼 신경망의 계층 구성

```
class TwoLayerNet:  
    def __init__(self, input_size, hidden_size, output_size):  
        I, H, O = input_size, hidden_size, output_size  
  
        # 가중치와 편향 초기화  
        W1 = np.random.randn(I, H)  
        b1 = np.random.randn(H)  
        W2 = np.random.randn(H, O)  
        b2 = np.random.randn(O)  
  
    model = TwoLayerNet(2, 4, 3)
```



구현해볼 신경망의 계층 구성

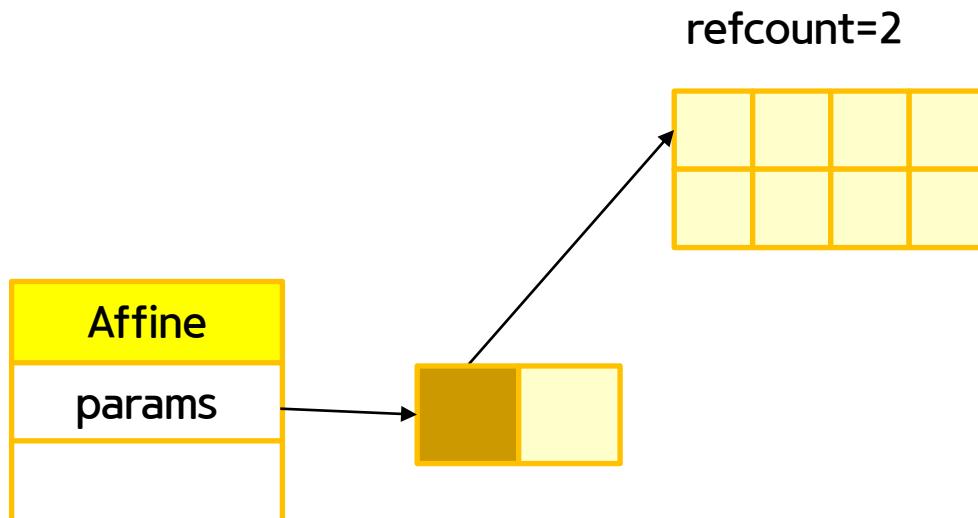
```
import sys  
a = np.array([1,2,3])  
print( sys.getrefcount(a) )  
b = a  
print( sys.getrefcount(a) )  
print(b)  
b[1] = 10  
print(b)  
print(a)
```



파이썬의 모든 객체는
heap 공간에 동적 할당 된다.

refcount가 다시 1이 되면
파이썬의 gc가 해당
메모리를 해지한다.

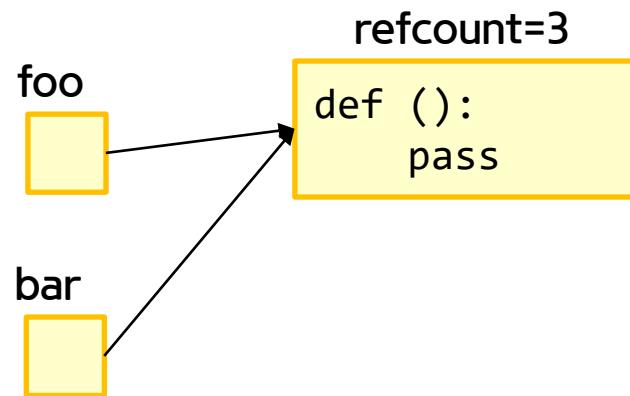
```
def __init__(self, input_size, hidden_size, output_size):
    I, H, O = input_size, hidden_size, output_size
    ...
    W1 = np.random.randn(I, H)
    ...
    Affine(W1, b1)
    ...
```



파이썬의 모든 객체는
heap 공간에 동적 할당 된다.

refcount가 다시 1이 되면
파이썬의 gc가 해당
메모리를 해지한다.

```
import sys  
  
def foo():  
    pass  
  
print( sys.getrefcount(foo) )
```



파이썬의 모든 객체는
heap 공간에 동적 할당 된다.

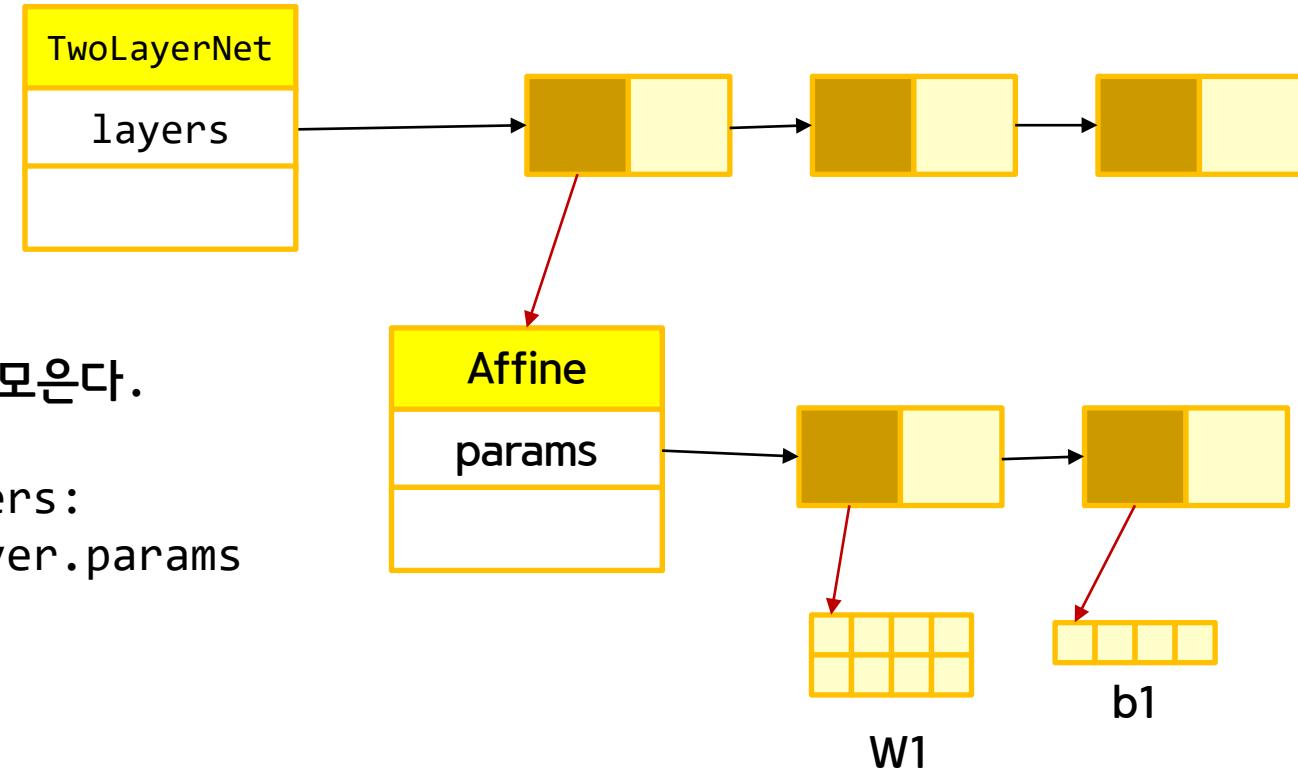
refcount가 다시 1이 되면
파이썬의 gc가 해당
메모리를 해지한다.

구현해볼 신경망의 계층 구성

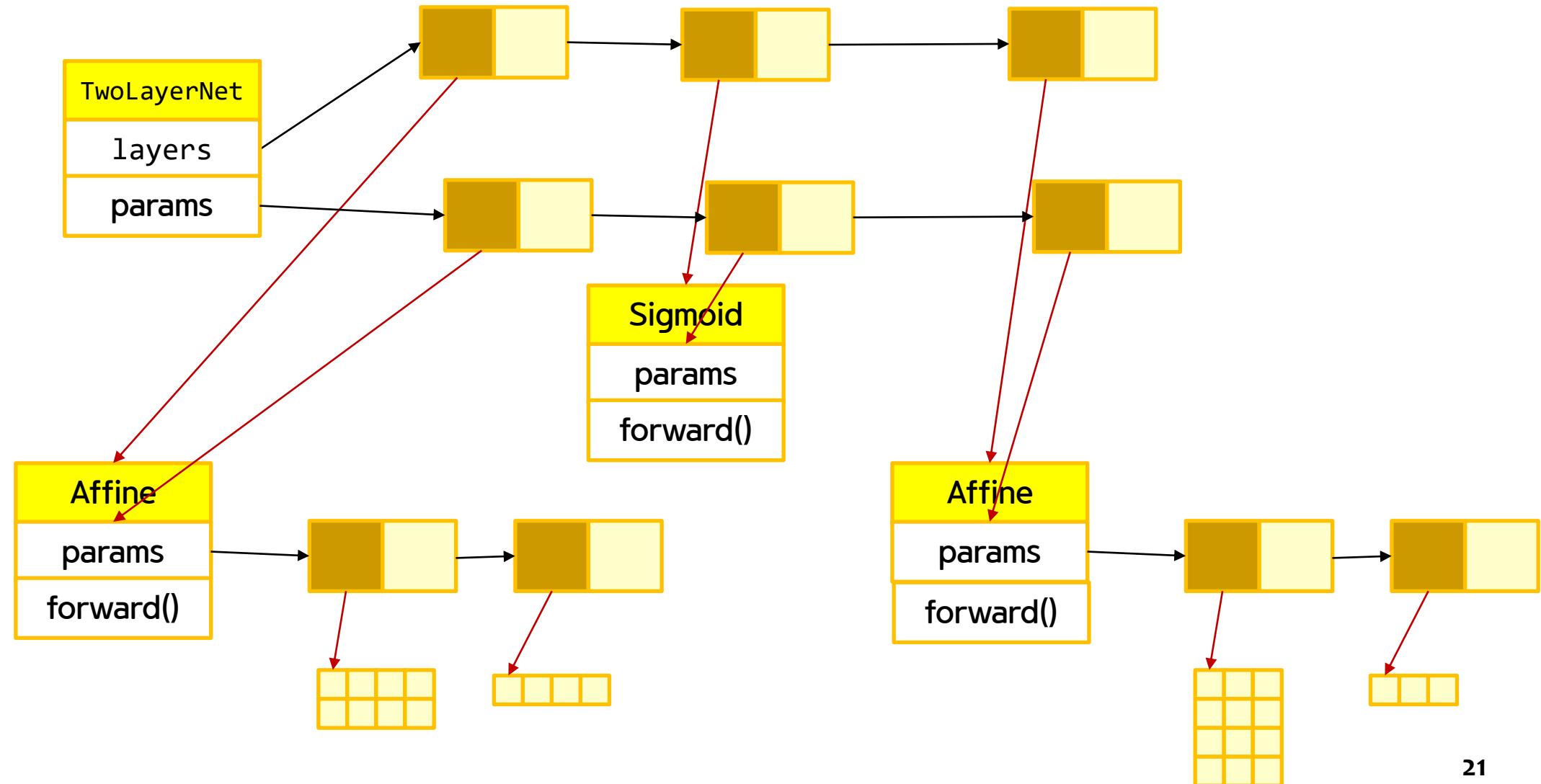
```

class TwoLayerNet:
    def __init__(self, input_size, hidden_size, output_size):
        ...
        # 계층 생성
        self.layers = [
            Affine(W1, b1),
            Sigmoid(),
            Affine(W2, b2)
        ]
        # 모든 가중치를 리스트에 모은다.
        self.params = []
        for layer in self.layers:
            self.params += layer.params
model = TwoLayerNet(2, 4, 3)
...

```



구현해볼 신경망의 계층 구성



1. 신경망 복습

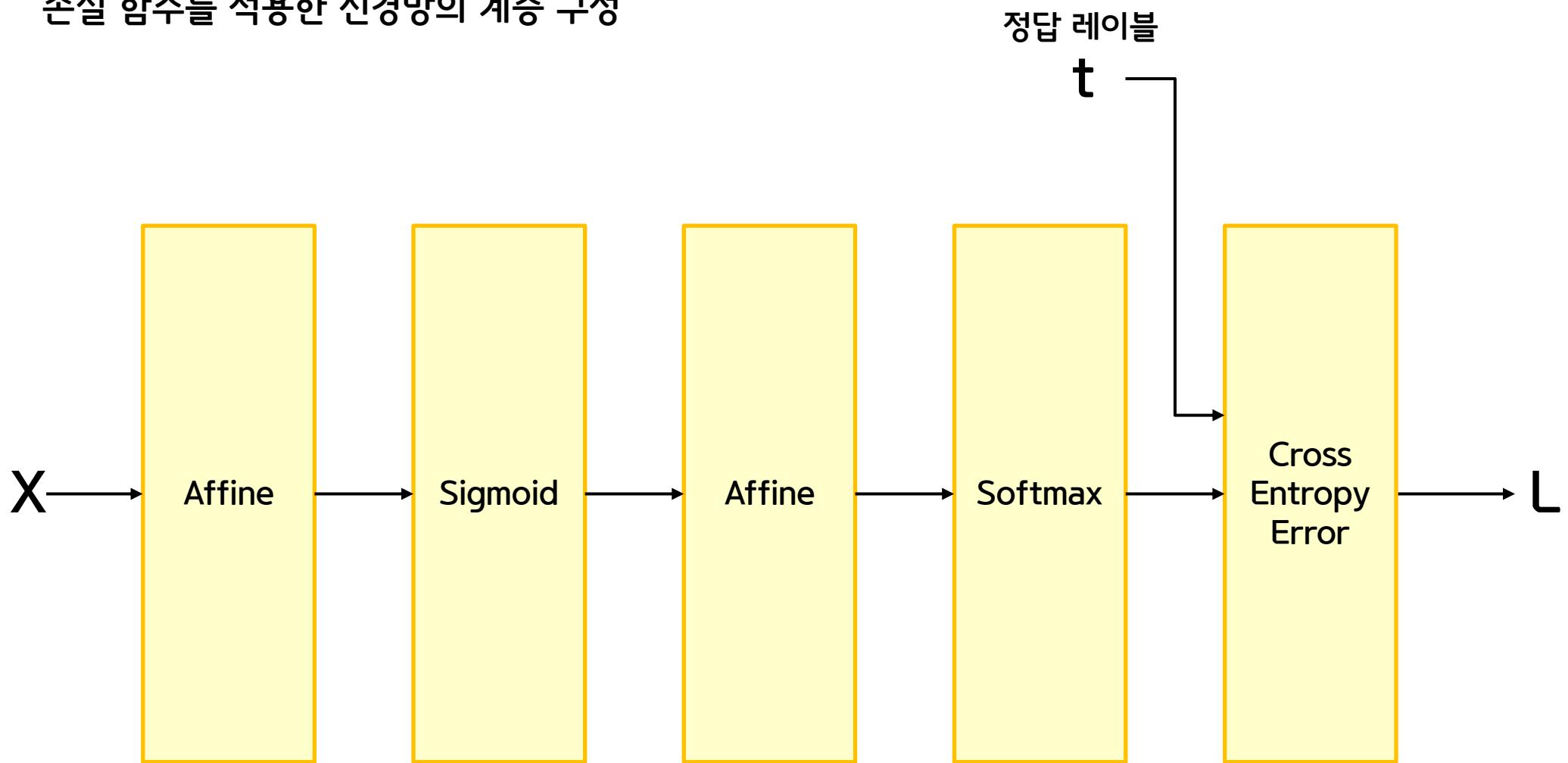
1.1 수학과 파이썬 복습

1.2 신경망 추론

1.3 신경망의 학습

1.4 신경망으로 문제를 풀다.

손실 함수를 적용한 신경망의 계층 구성



소프트맥스 함수 수식

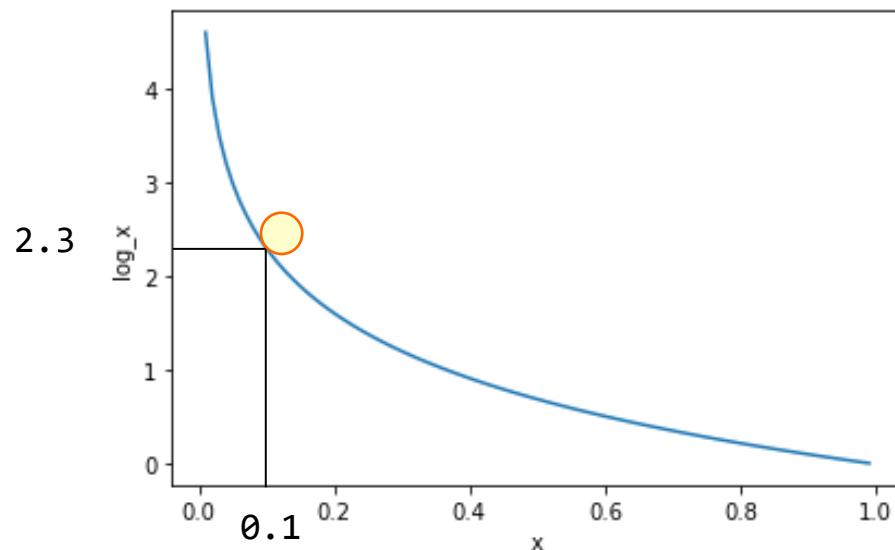
$$y_k = \frac{\exp(s_k)}{\sum_{i=1}^n \exp(s_i)}$$

크로스 엔트로피 오차의 수식

$$L = - \sum_k t_k \log y_k$$

y	t
0.1 0.2 0.7	1 0 0

$$-(1 * \log_e 0.1 + 0 * \log_e 0.2 + 0 * \log_e 0.7)$$

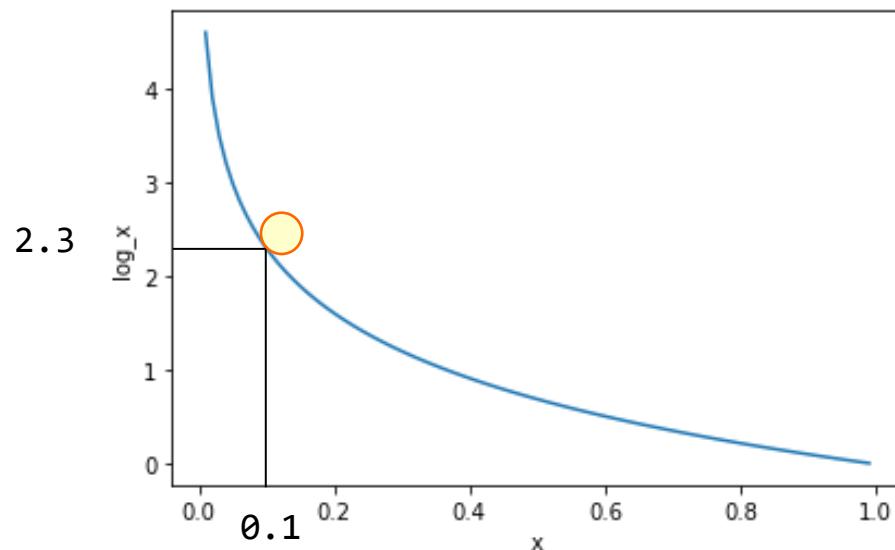


크로스 엔트로피 오차의 수식

$$L = - \sum_k t_k \log y_k$$

y	t
0.1 0.2 0.7	1 0 0

$$-(1 * \log_e 0.1 + 0 * \log_e 0.2 + 0 * \log_e 0.7)$$



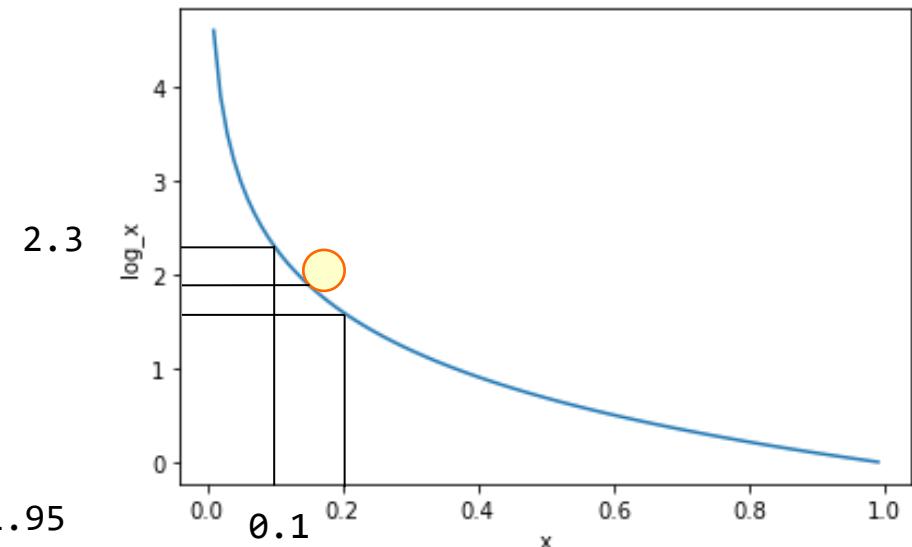
미니배치 처리를 고려한 크로스 엔트로피 오차의 수식

$$L = -\frac{1}{N} \sum_n \sum_k t_{nk} \log y_{nk}$$

y	0.1	0.2	0.7
0.3	0.2	0.5	

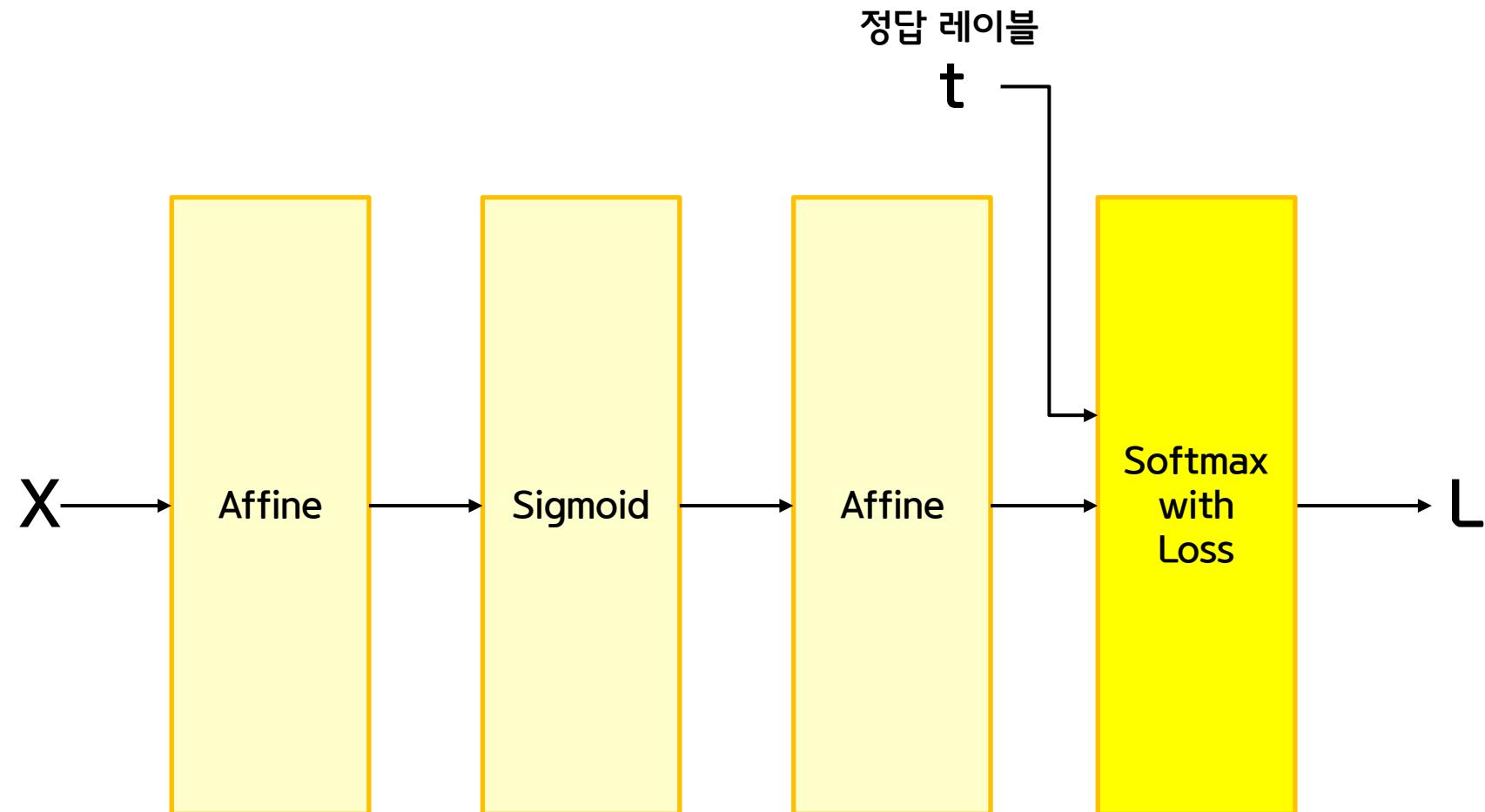
t	1	0	0
0	1	0	

2.3
1.95

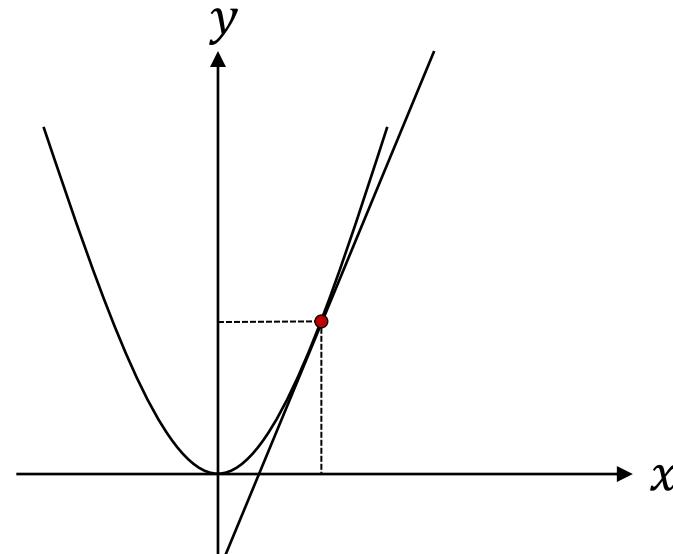


$$-0.5 * ((1 * \log_e 0.1 + 0 * \log_e 0.2 + 0 * \log_e 0.7) + \\ (0 * \log_e 0.3 + 1 * \log_e 0.2 + 0 * \log_e 0.5))$$

Softmax with Loss 계층을 이용하여 손실을 출력



$y = x^2$ 의 미분은 각 x에서의 기울기를 나타낸다.



벡터인 경우 미분

$$\frac{\partial L}{\partial x} = \left(\frac{\partial L}{\partial x_1}, \frac{\partial L}{\partial x_2}, \dots, \frac{\partial L}{\partial x_n} \right)$$

행렬인 경우 미분

$$\frac{\partial L}{\partial W} = \begin{pmatrix} \frac{\partial L}{\partial W_{11}} & \cdots & \frac{\partial L}{\partial W_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial W_{m1}} & \cdots & \frac{\partial L}{\partial W_{mn}} \end{pmatrix}$$

연쇄 법칙

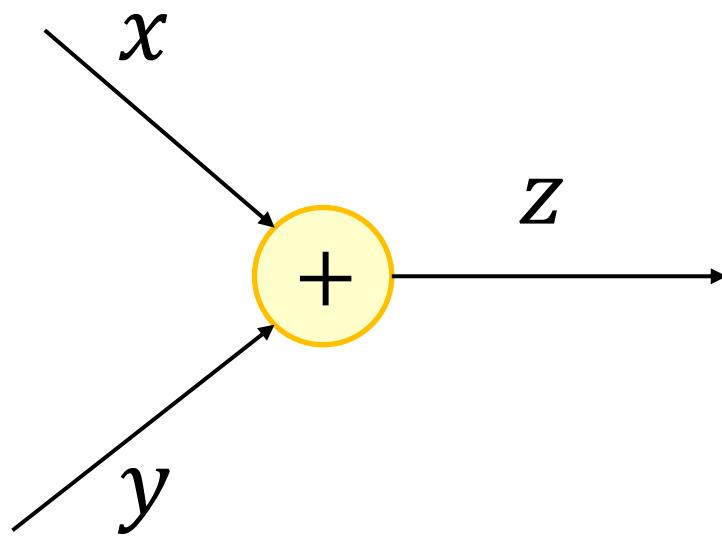
$$\begin{aligned}y &= f(x) \\z &= g(y)\end{aligned}$$

$$z = g(f(x))$$

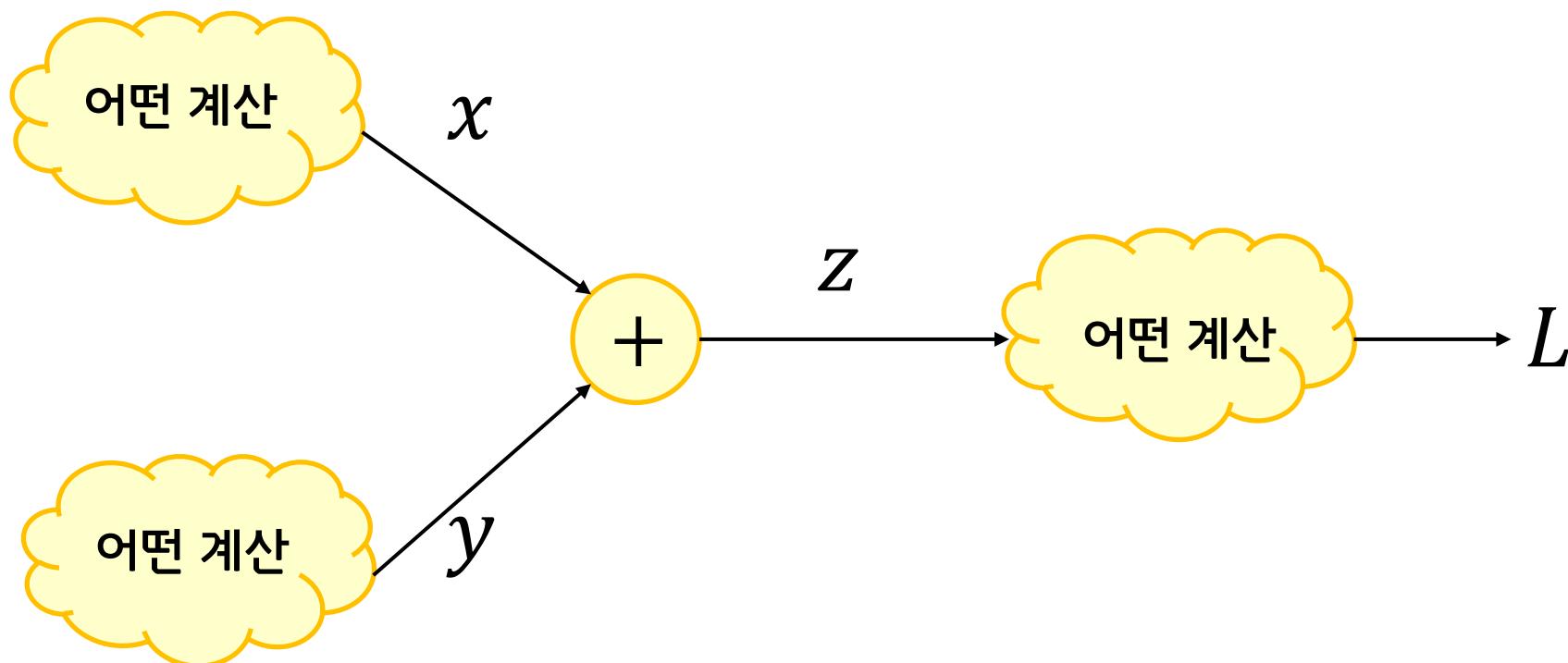
합성함수의 미분

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$$

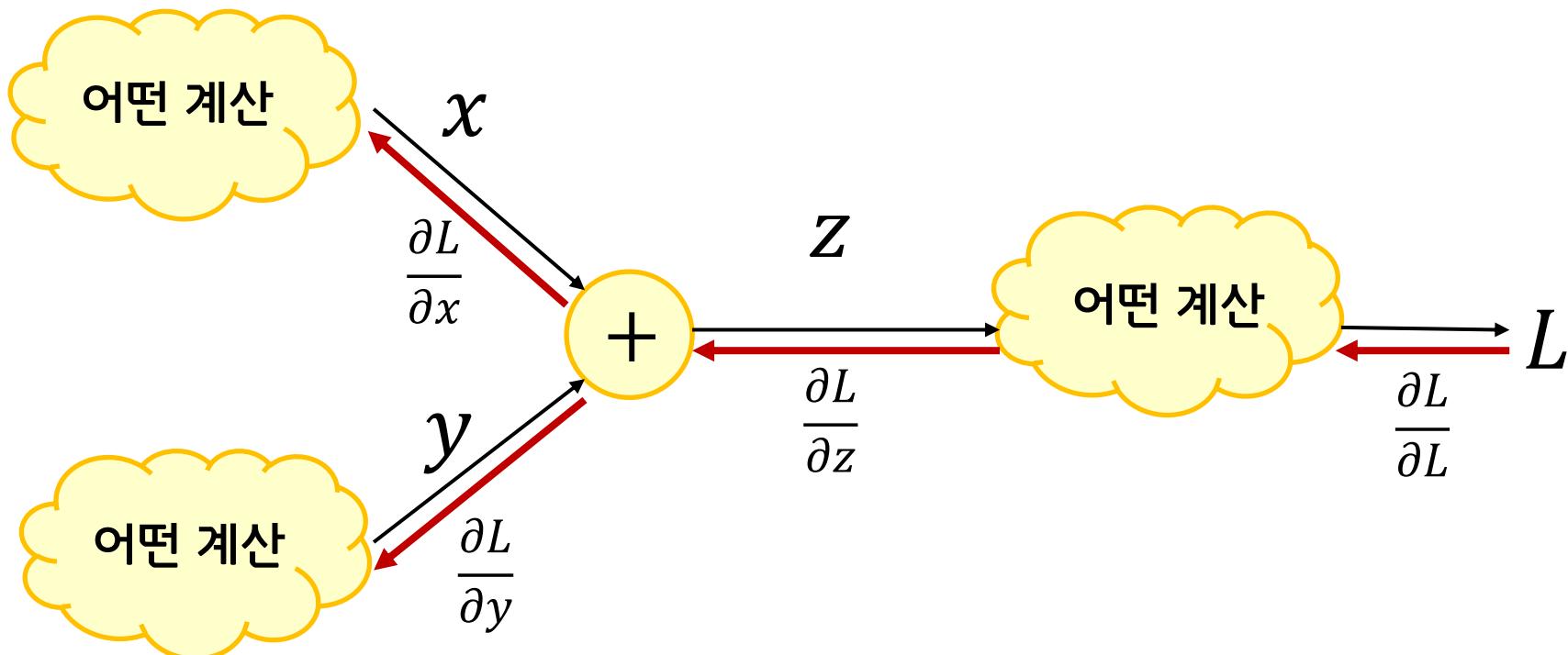
$z = x + y$ 를 나타내는 계산 그래프



앞뒤로 추가된 노드는 '복잡한 전체 계산'의 일부를 구성한다.

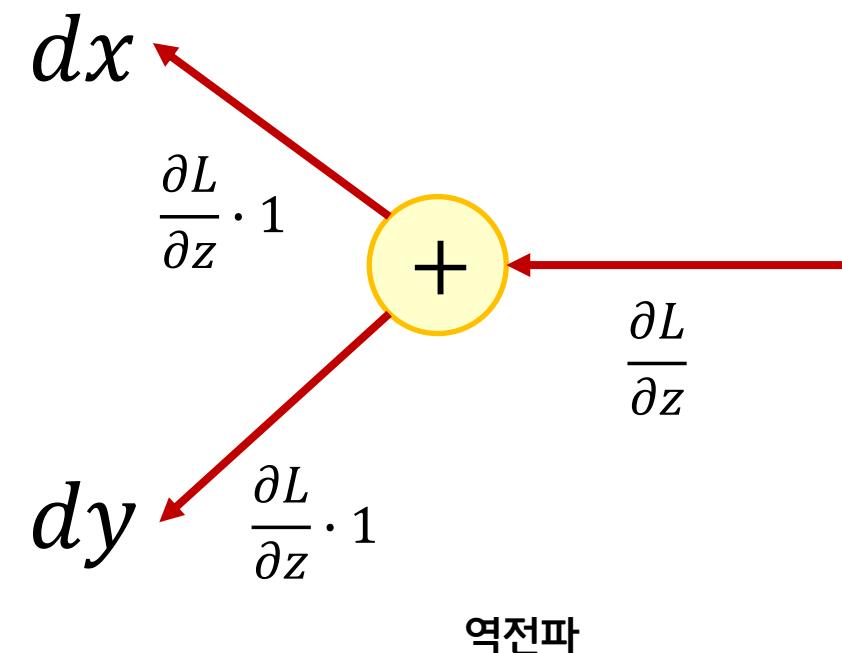
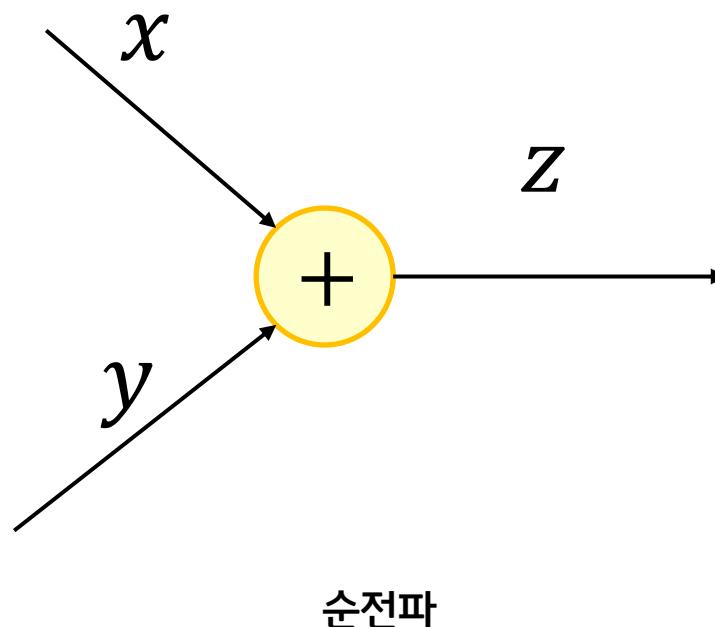


계산 그래프의 역전파



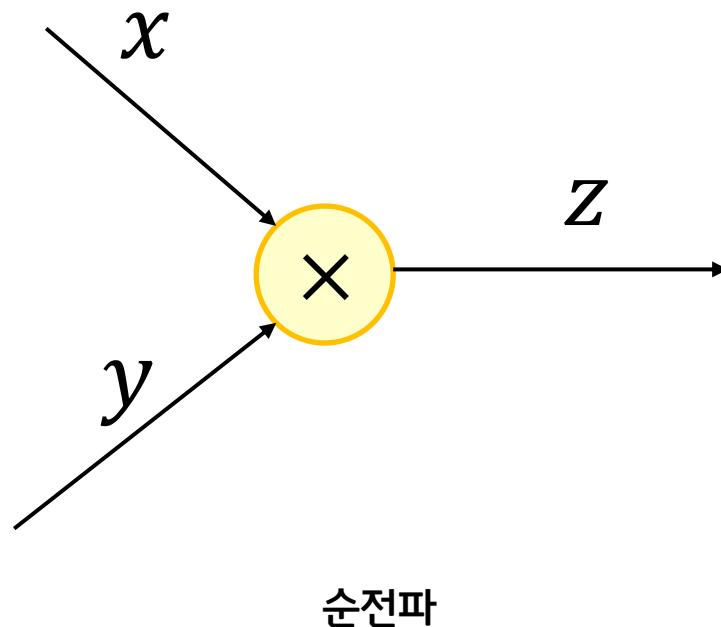
덧셈 노드의 순전파와 역전파

$$z = x + y$$

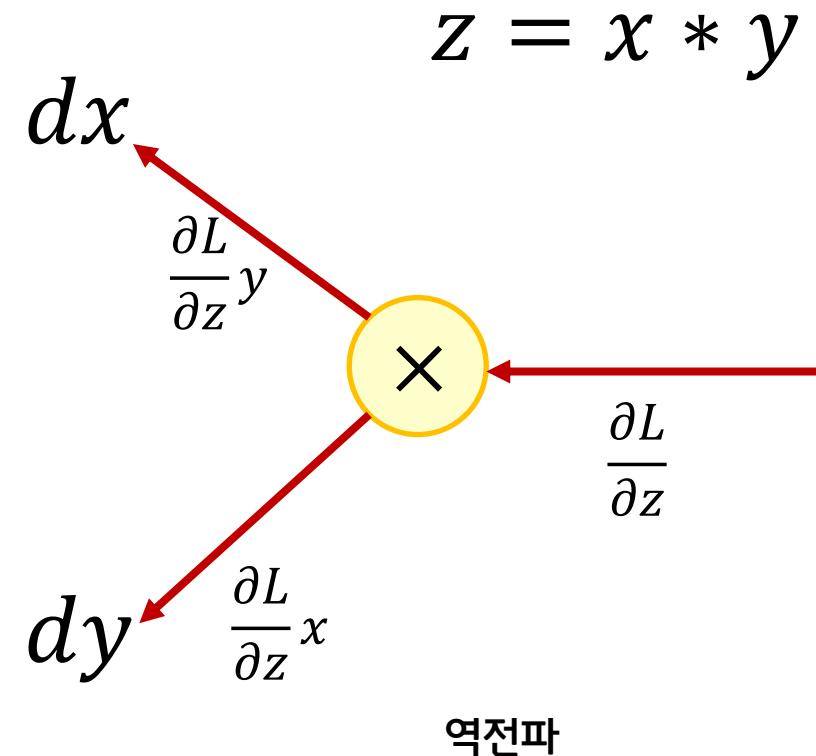


덧셈 노드의 미분은 리피트 연산이다.

곱셈 노드의 순전파와 역전파



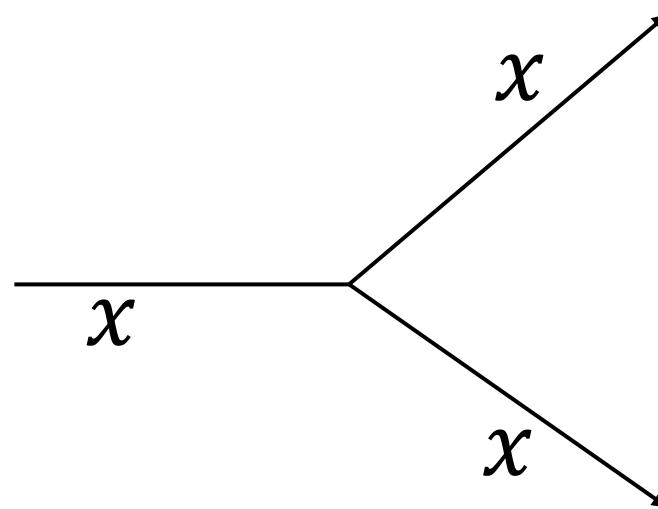
순전파



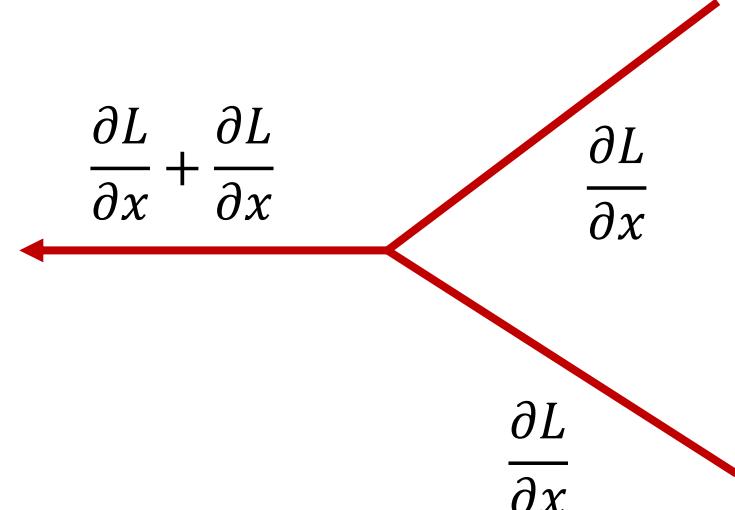
역전파

곱셈 모드의 미분은 뒤에서 온 미분값에
포워드시 전달된 반대편 값을 곱하여 전달 한다.

분기 노드의 순전파와 역전파



순전파

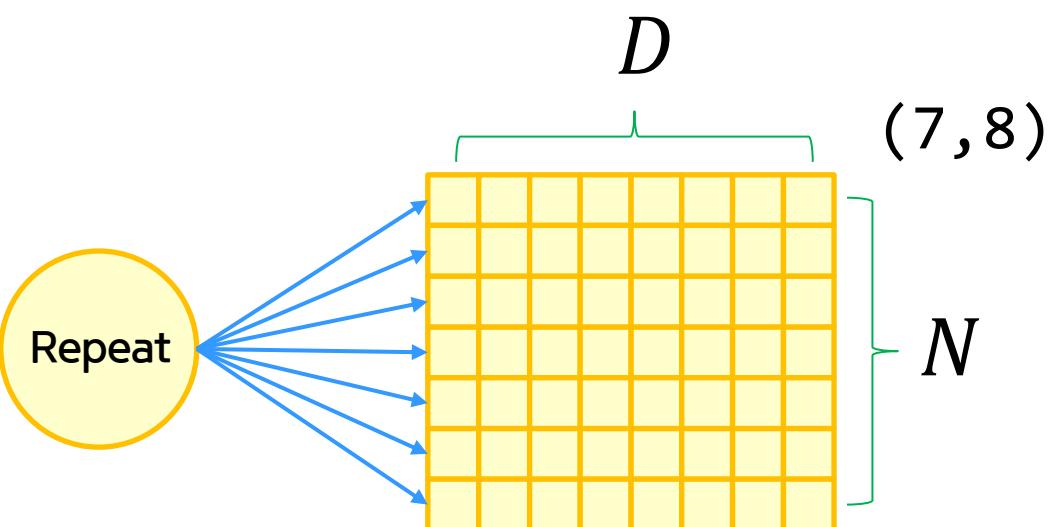
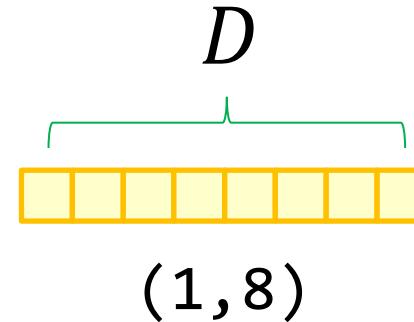


역전파

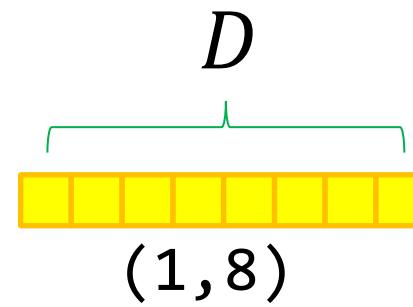
리피트 노드의 미분은 합 연산이다.

Repeat 노드의 순전파와 역전파

순전파



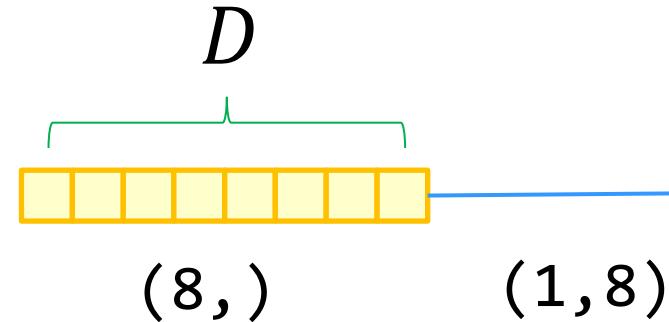
역전파



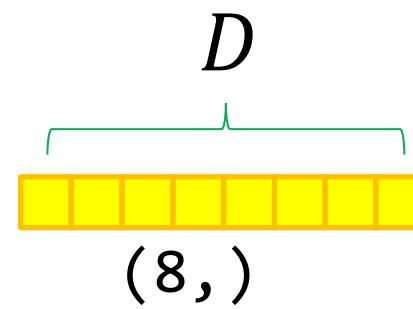
리피트 노드의 미분은 합 연산이다.

Repeat 노드의 순전파와 역전파

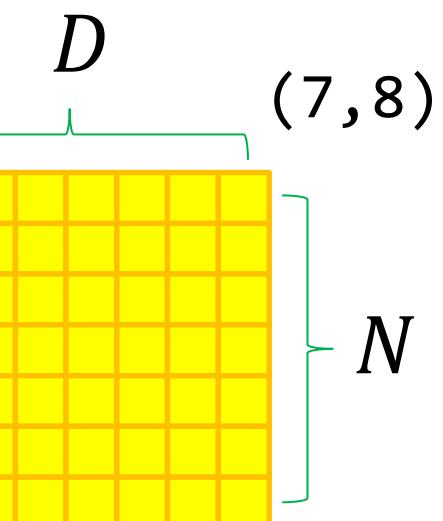
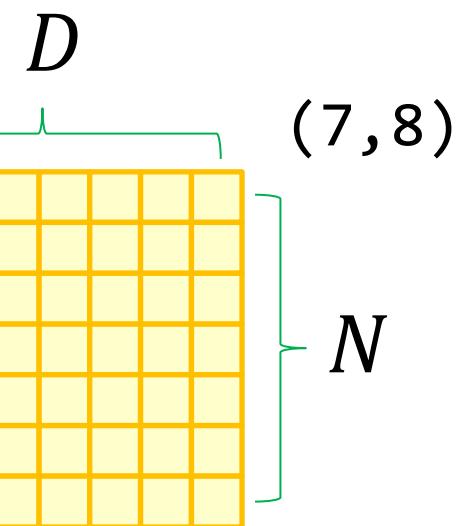
순전파



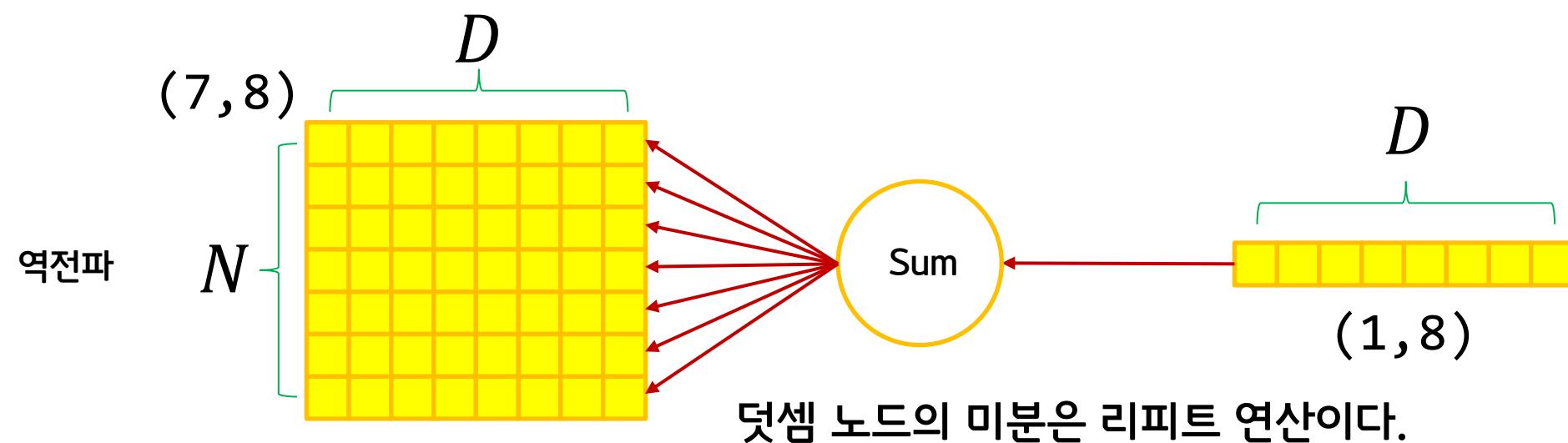
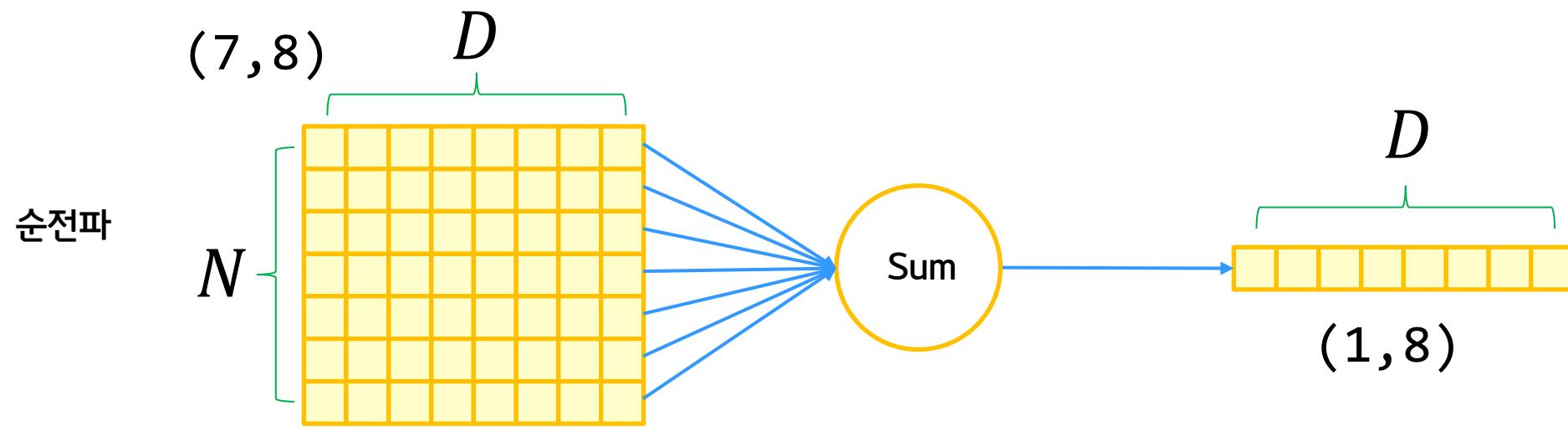
역전파



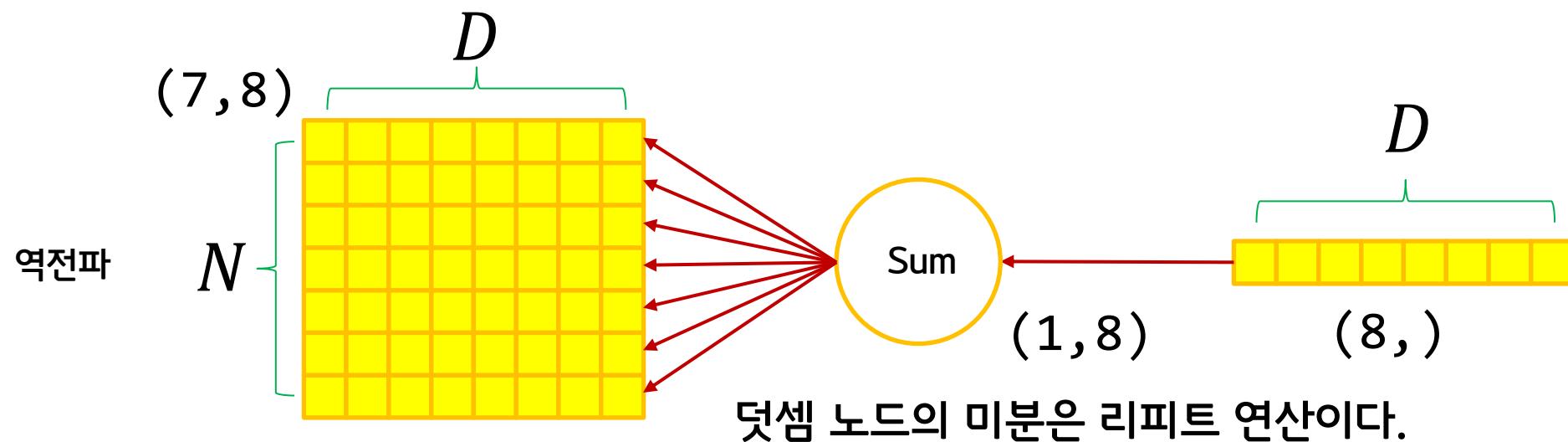
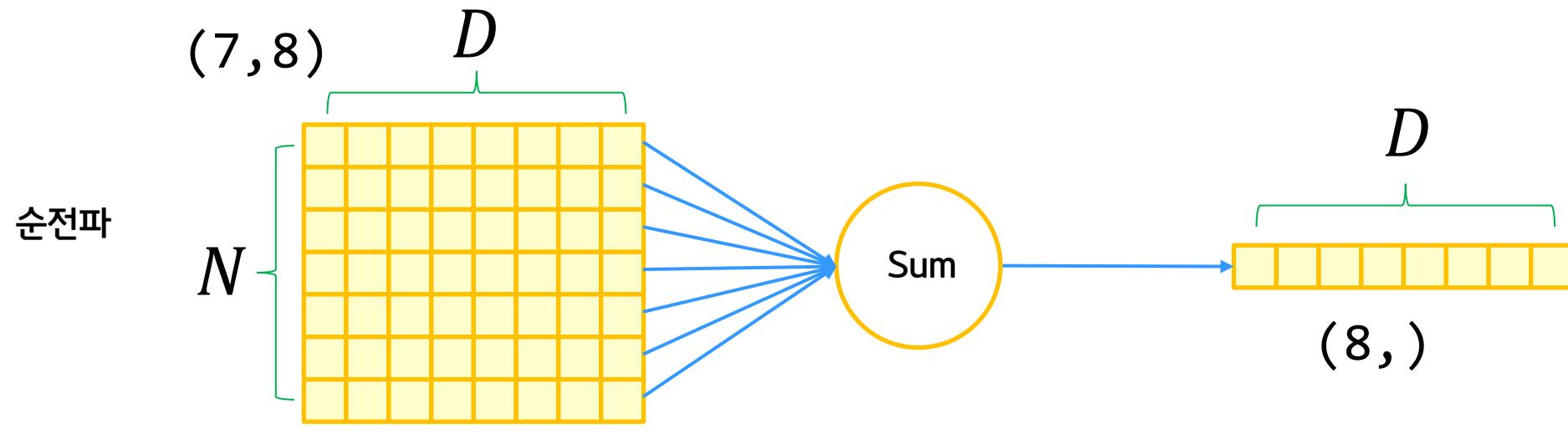
리피트 노드의 미분은 합 연산이다.



Sum 노드의 순전파와 역전파



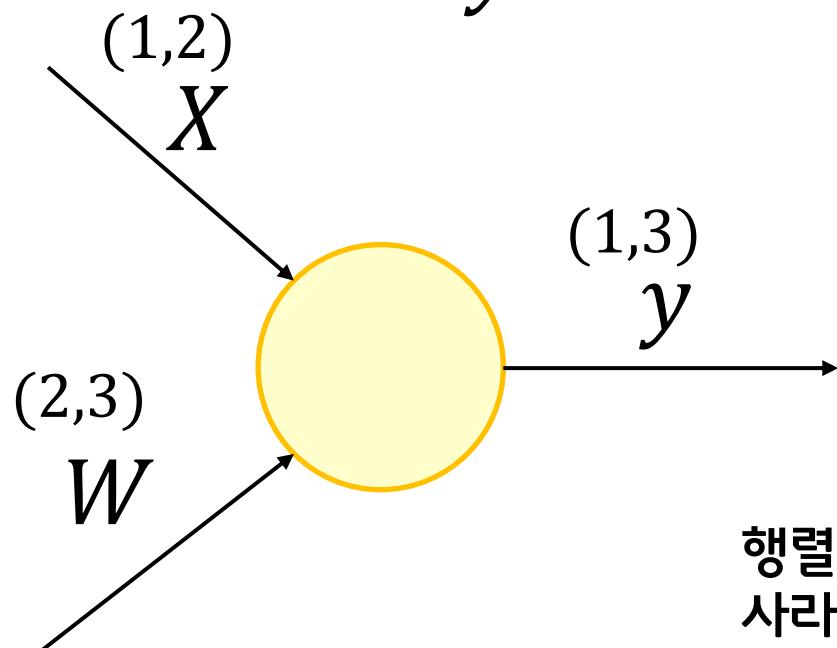
Sum 노드의 순전파와 역전파



MatMul 노드의 순전파

$$y = XW$$

(1,2)(2,3)



$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial y} W^T$$

(1,3)(3,2)

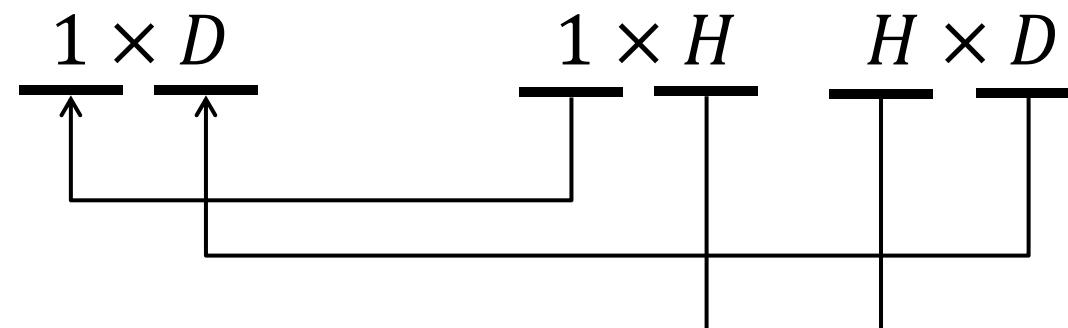
$$\frac{\partial L}{\partial W} = X^T \frac{\partial L}{\partial y}$$

(2,1)(1,3)

행렬곱의 미분은 뒤에서 온 미분을 편미분으로
사라진 변수 자리에 쓰고 남은 값은 전치하여
두값을 행렬 곱한다.

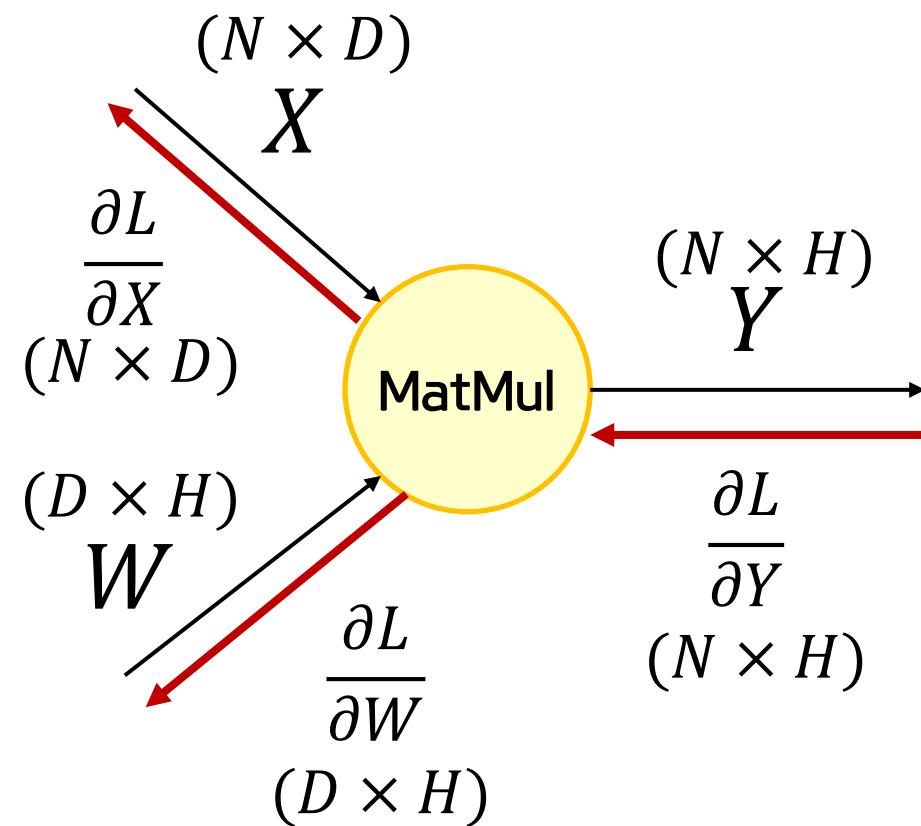
MatMul 미분

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} W^T$$



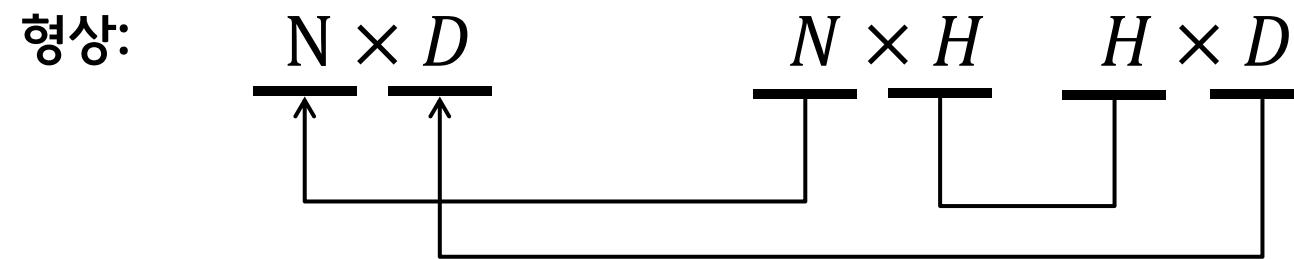
일치시킨다.

MatMul 노드의 역전파

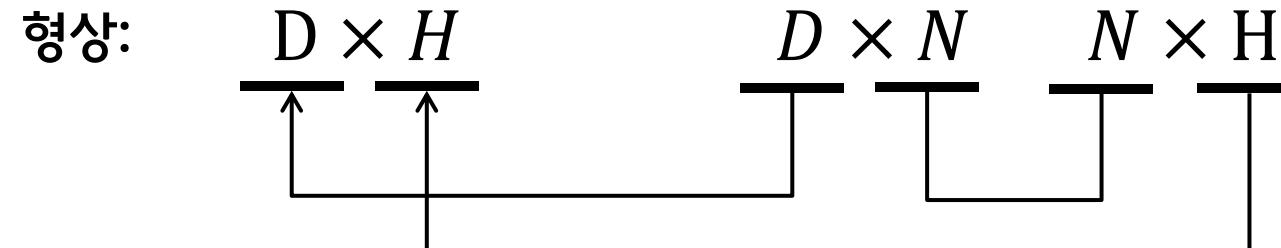


행렬의 형상을 확인하여 역전파 식을 유도 한다.

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} W^T$$

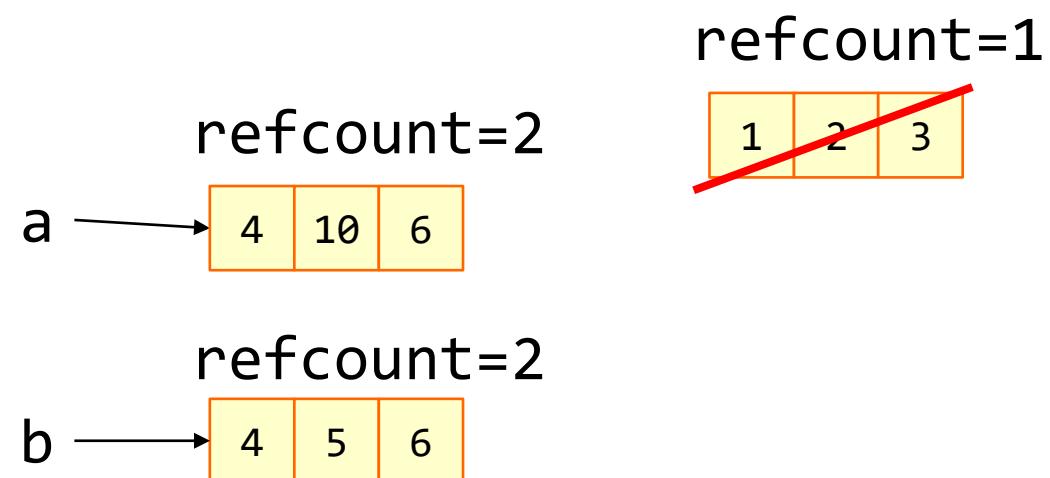
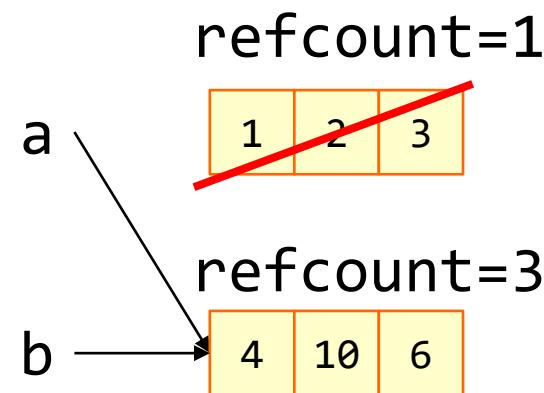


$$\frac{\partial L}{\partial W} = X^T \frac{\partial L}{\partial Y}$$



```
a = np.array([1, 2, 3])  
b = np.array([4, 5, 6])
```

```
a = b  
a[...] = b
```



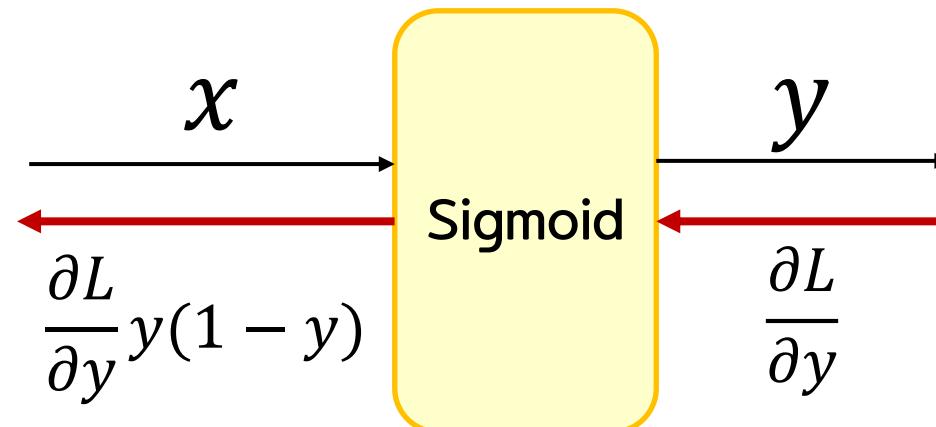
Sigmoid 계층

Sigmoid 식

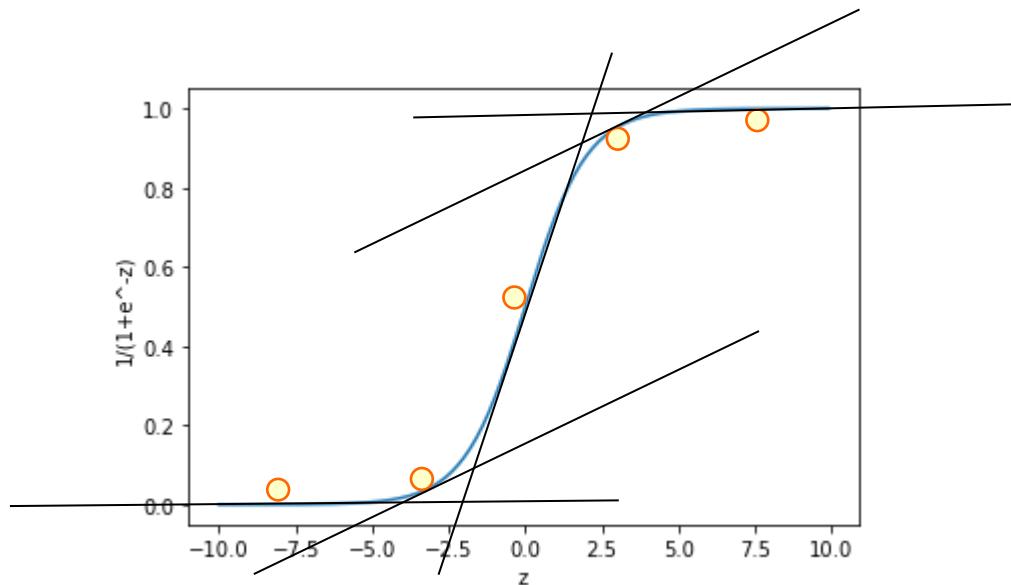
$$y = \frac{1}{1 + e^{-x}}$$

Sigmoid 미분식

$$\frac{\partial y}{\partial x} = y(1 - y)$$

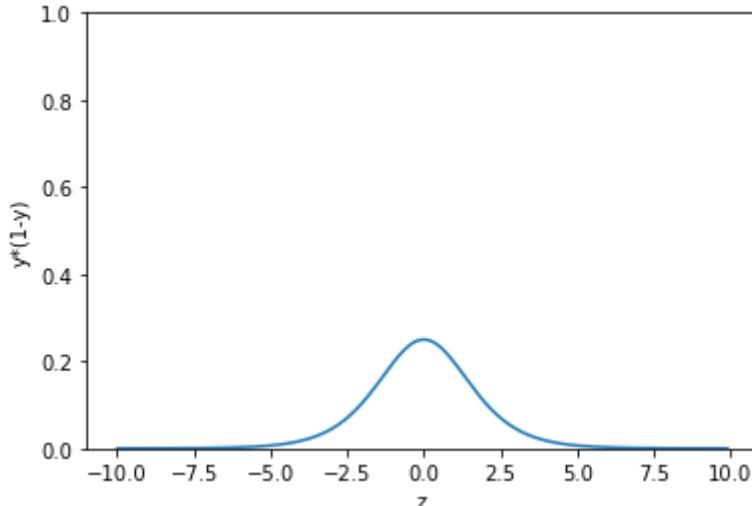


Sigmoid 계층

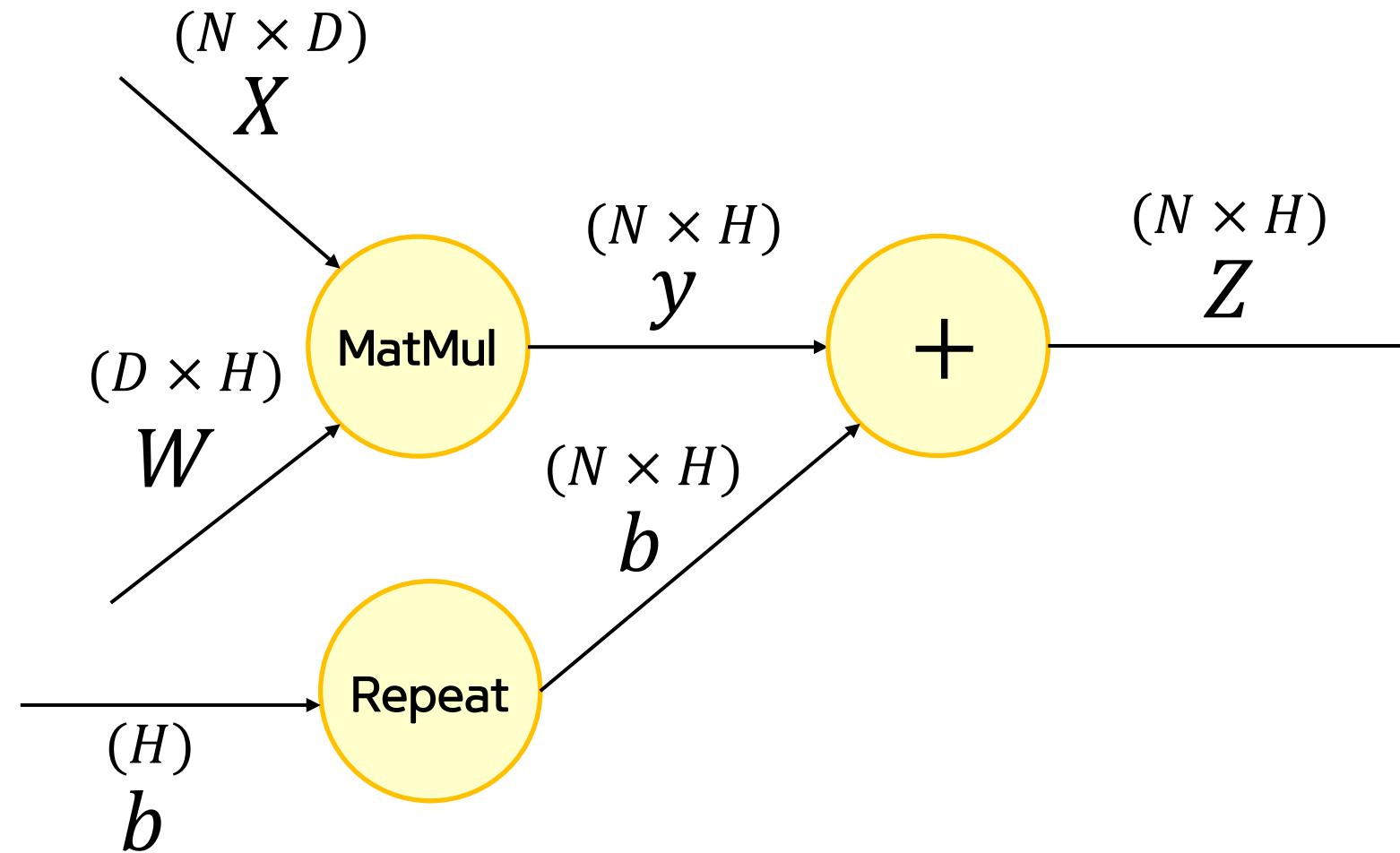


$$y = \frac{1}{1 + e^{-x}}$$

$$\frac{\partial y}{\partial x} = y(1 - y)$$

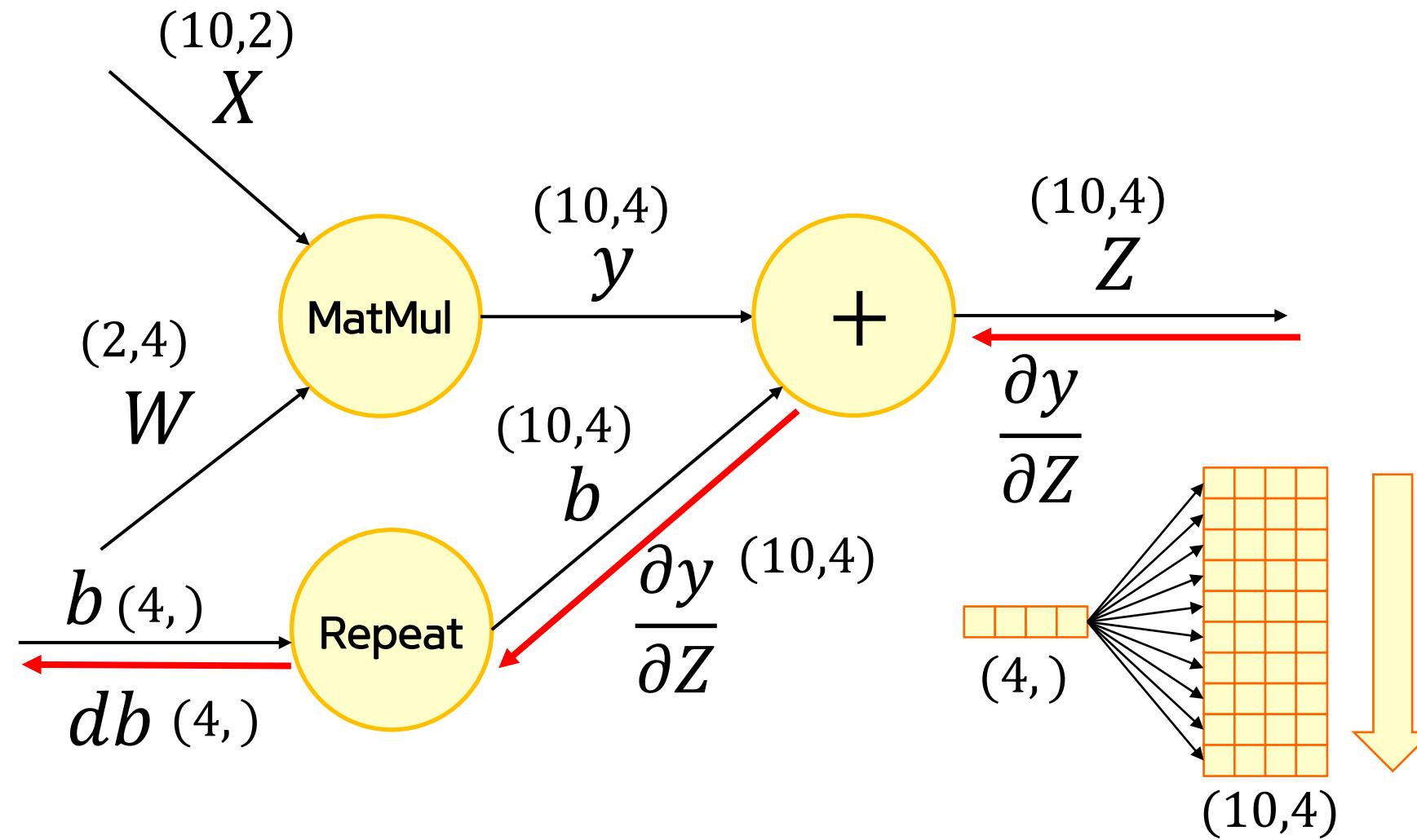


Affine 계층

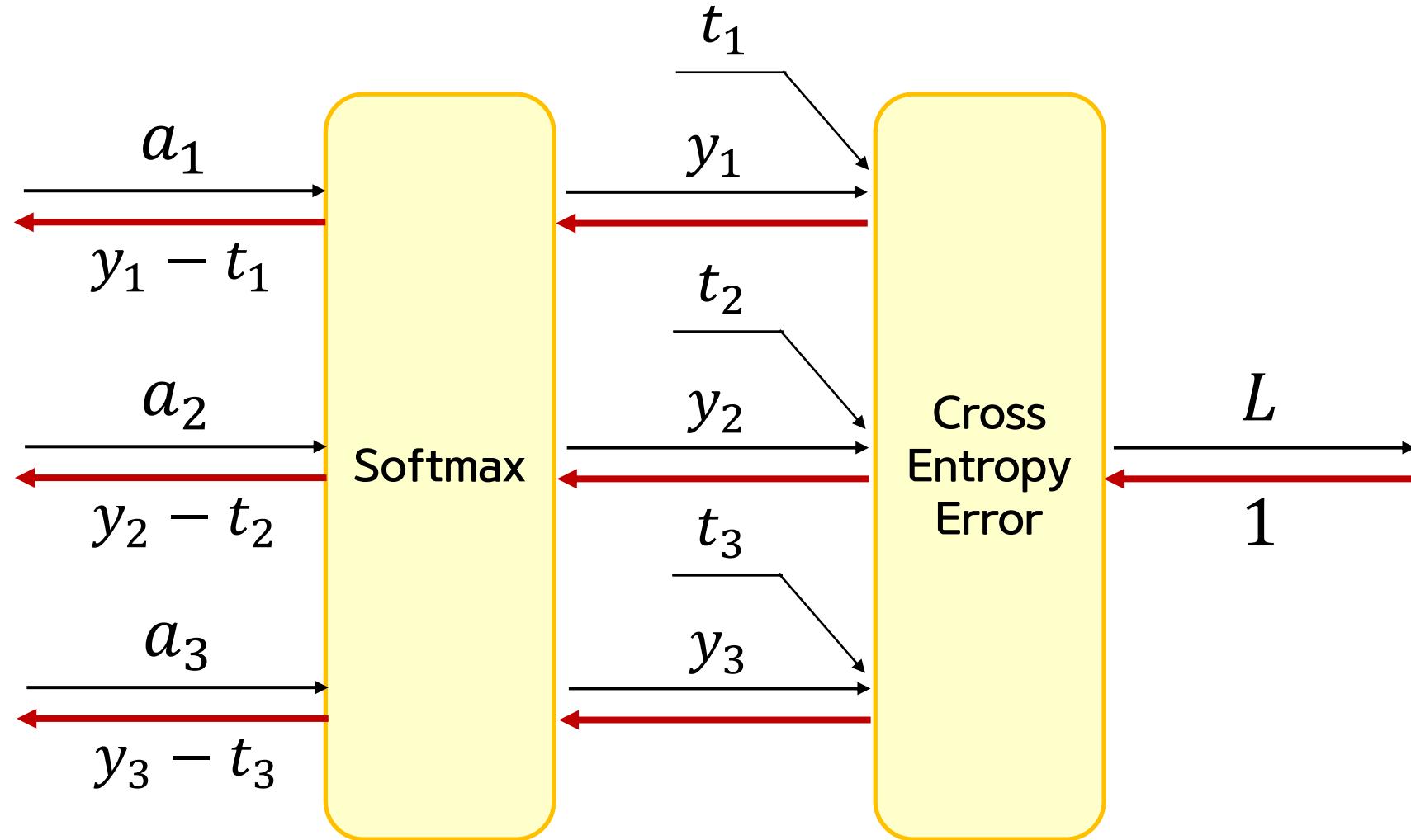


Affine 계층

$$Z = XW + b$$



Softmax with Loss 계층



- 1단계: 미니배치
훈련 데이터 중에서 무작위로 다수의 데이터를 골라낸다.
- 2단계: 기울기 계산
오차역전파법으로 각 가중치 매개변수에 대한 손실 함수의 기울기를 구한다.
- 3단계: 매개변수 갱신
기울기를 사용하여 가중치 매개변수를 갱신한다.
- 4단계: 반복
1~3 단계를 필요한 만큼 반복한다.

$$W \leftarrow W - \alpha \frac{\partial L}{\partial W}$$

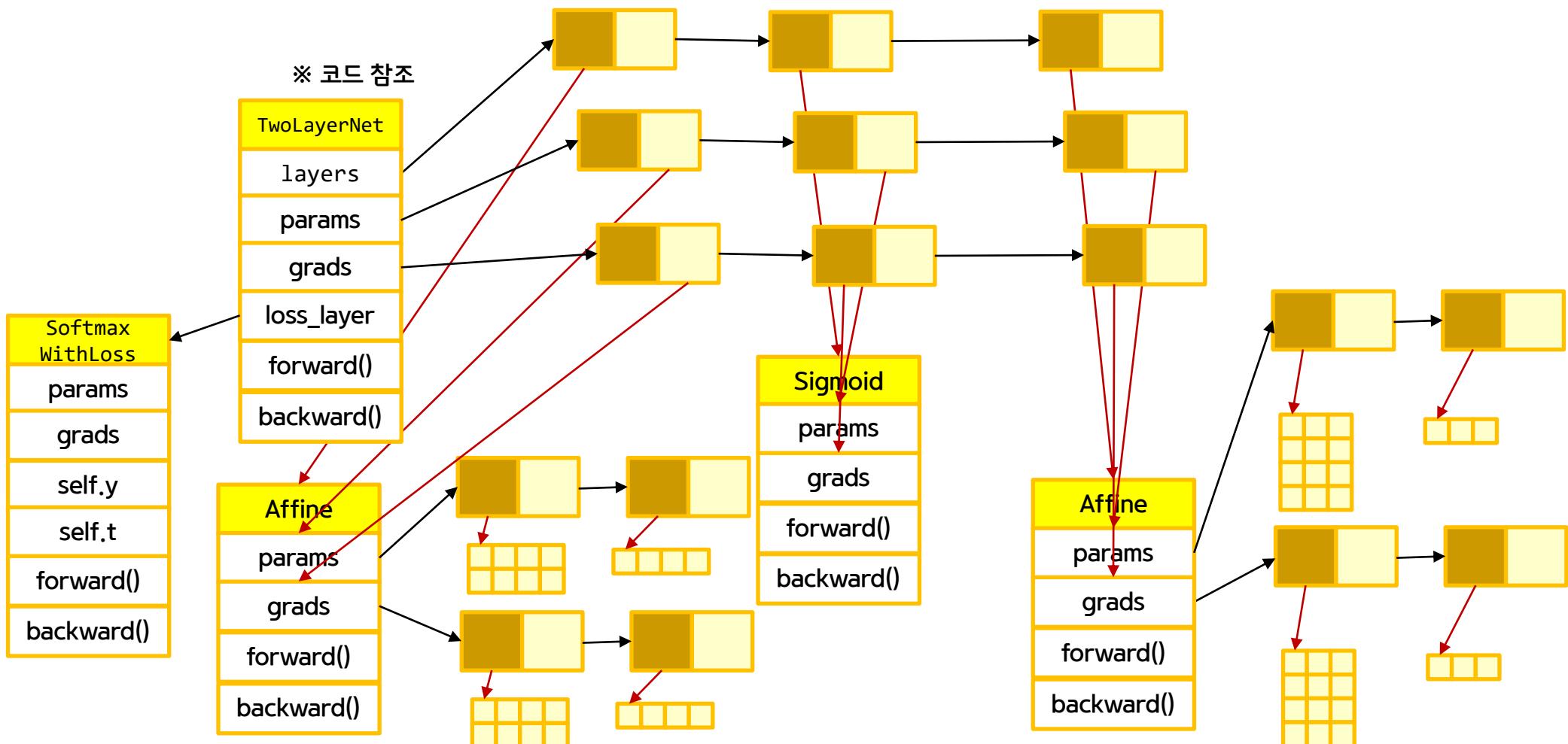
1. 신경망 복습

1.1 수학과 파이썬 복습

1.2 신경망 추론

1.3 신경망의 학습

1.4 신경망으로 문제를 풀다.



※ 코드 참조

x		
0	1	2
100	010	001

 $t = 2$

0.8	0
0.1	0
0.1	1

0.8 0.1 0.1

2.3025850929940456840179914546844

0.1 0.1 0.8

크로스 엔트로피 손실 함수

0.22314355131420975576629509030983

$$L = - \sum_{c=1}^C y_c \log(a_c) = -(y_1 \log(a_1) + y_2 \log(a_2) + \dots + y_c \log(a_c)) \\ -y_3 \log(a_3)$$

```
-np.sum(np.log(y[np.arange(batch_size), t] + 1e-7)) / batch_size
```

※ 코드 참조

인수	설명
x	입력 데이터
t	정답 레이블
max_epoch(=10)	학습을 수행하는 에폭 수
batch_size(=32)	미니배치 크기
eval_interval(=20)	결과(평균 손실 등) 출력하는 간격 예컨대 eval_interval=20으로 설정하면, 20번째 반복마다 손실의 평균을 구해 화면에 출력한다.
max_grad(=None)	기울기 최대의 노름 기울기 노름이 이 값을 넘어서면 기울기를 중인다(이를 기울기 클리핑이라 한다.)

2. 자연어와 단어의 분산 표현

2.1 자연어 처리란

2.2 시소러스

2.3 통계 기반 기법

2.4 통계 기반 기법 개선하기

한국어와 영어 등 우리가 평소에 쓰는 말을 자연어라고 한다.

자연어 처리(NLP)를 풀어서 말하면
'우리의 말을 컴퓨터에게 이해시키기 위한 기술(분야)'이다.

자연어 처리가 추구하는 목표는 사람의 말을 부드럽게
컴퓨터가 이해하도록 만들어서,
컴퓨터가 우리에게 도움이 되는 일을 수행하게 하는 것이다.

우리의 말은 '문자로'로 구성되며, 말의 의미는 '단어'로 구성된다.

단어는 의미의 최소 단위이기 때문에 자연어를 컴퓨터에게 이해시키는 데는 '단어의 의미'를 이해시키는 것이 중요하다.

세가지 기법

- 시소러스를 활용한 기법
- 통계 기반 기법
- 추론 기반 기법(word2vec)

2. 자연어와 단어의 분산 표현

2.1 자연어 처리란

2.2 시소러스

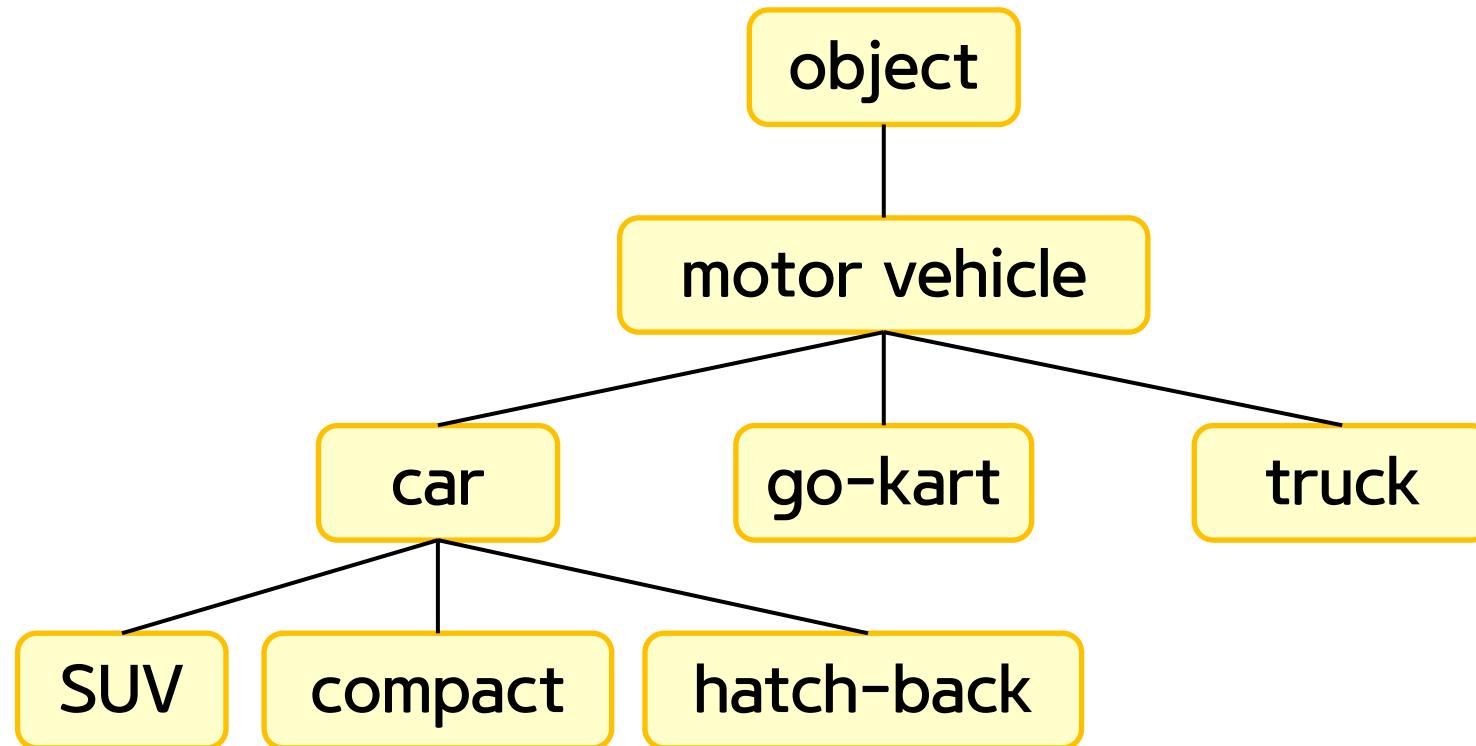
2.3 통계 기반 기법

2.4 통계 기반 기법 개선하기

동의어의 예: "car", "auto", "automobile" 등은 "자동차"를 뜻하는 동의어다.



단어들의 의미의 상/하위 관계에 기초해 그래프로 표현한다.



WordNet과 같은 시소러스에는 수많은 단어에 대한 동의어와 계층 구조 등의 관계가 정의되어 있다.

그리고 이 지식을 이용하면 '단어의 의미'를 컴퓨터에 전달할 수 있다.

하지만 사람이 수작업으로 레이블링하는 방식에는 문제들이 존재한다.

- **시대 변화에 대응하기 어렵다.**

신조어 혹은 의미 변화된 단어들을 바로 적용시키기 어렵다.

- **사람을 쓰는 비용이 듈다.**

현존하는 영어 단어의 수는 1,000만 개가 넘으며 WordNet에 등록된 단어는 20만 개 이상이다.

- **단어의 미묘한 차이를 표현할 수 없다.**

가령, 빈티지와 레트로의 의미는 같으나 용법의 차이가 존재한다.

위 문제점들을 피하기 위해 '통계 기반 기법'과 신경망을 사용한 '추론 기반 기법'을 알아볼 것이다.

2. 자연어와 단어의 분산 표현

2.1 자연어 처리란

2.2 시소러스

2.3 통계 기반 기법

2.4 통계 기반 기법 개선하기

자연어 처리에는 다양한 말뭉치가 사용되는데

예로는 구글 뉴스와 위키백과 등의 텍스트 데이터를 들 수 있다.

※코드로 확인

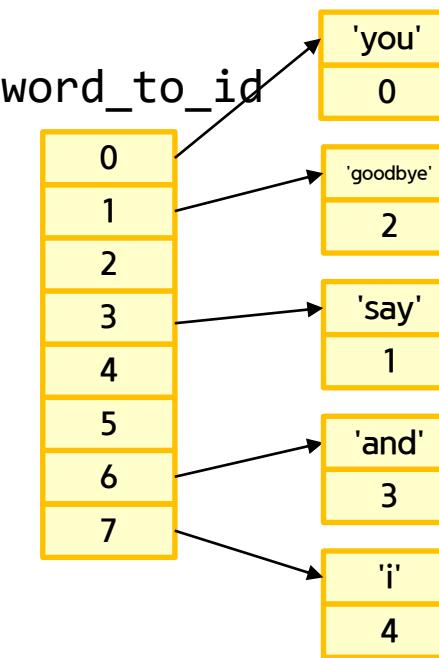
```
[ 'you', 'say', 'goodbye', 'and', 'i', 'say', 'hello', '.' ]
```



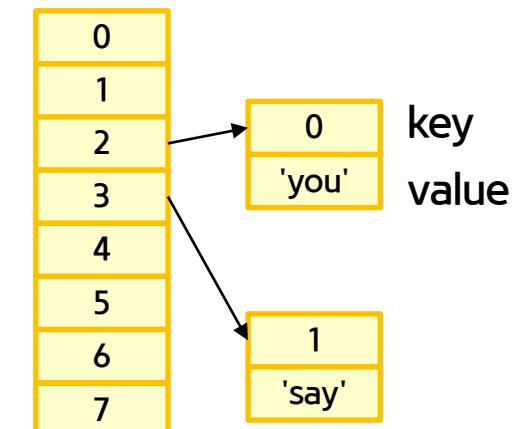
```
word_to_id = {}
id_to_word = {}

for word in words:
    if word not in word_to_id:
        new_id = len(word_to_id)
        word_to_id[word] = new_id
        id_to_word[new_id] = word
```

`hash('say') % 8`



`id_to_word`



자연어 처리에는 다양한 말뭉치가 사용되는데

예로는 구글 뉴스와 위키백과 등의 텍스트 데이터를 들 수 있다.

※코드로 확인

색에는 고유한 이름이 붙여진 다채로운 색들도 있고,
RGB(Red/Green/Blue)라는 세가지 성분이 어떤 비율로 섞여 있느냐로 표현하는 방법이 있다.
전자는 색의 가짓수만큼 의 이름을 부여하는 반면에 후자는 색을 3차원의 벡터로 표현한다.

여기서 주목할 점은 RGB같은 벡터 표현이 단 3개의 성분으로 간결하게 표현할 수 있고,
색을 더 정확하게 명시할 수 있다는 점이다.

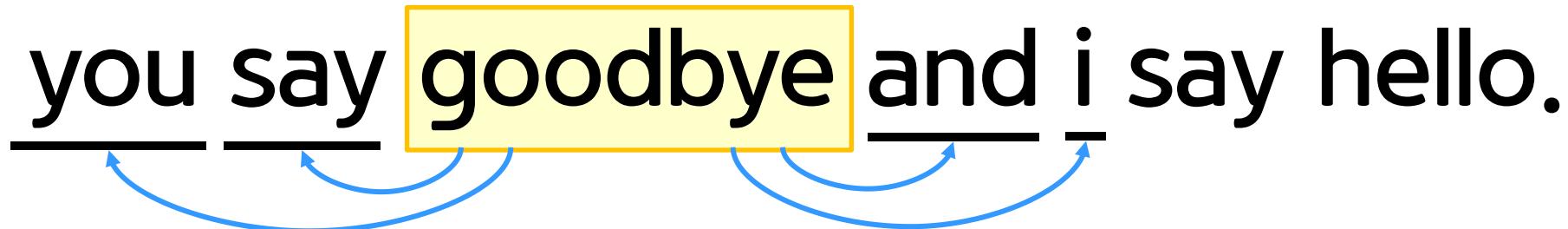
'색'을 벡터로 표현하듯 '단어'도 벡터로 표현할 수 있다. 이를 단어의 '분산 표현'이라고 한다.

- 'Red' = (255,0,0)
- 'Blue' = (0,0,255)
- 'Green' = (0,255,0)
- 'Yellow' = (255,255,0)

분포 가설이란 단어의 의미는 주변 단어에 의해 형성된다는 것이다.
분포 가설이 말하고자 하는 것은 단어 자체에는 의미가 없고,
그 단어가 사용된 '맥락'이 의미를 형성한다는 것이다.

예를 들어, I drink beer를 I Guzzle beer라고 해도 Guzzle을 drink로 이해할 수 있다는 것이다.

윈도우 크기가 2인 '맥락'의 예. 단어 "goodbye"에 주목한다면,
그 좌우의 두 단어(총 네 단어)를 맥락으로 이용한다.



위 그림에서 goodbye를 기준으로 좌우의 두 단어씩이 '맥락'에 해당한다.
맥락의 크기를 '윈도우 크기'라고 한다. 여기서는 '윈도우 크기'가 2이기 때문에
좌우로 두 단어씩이 맥락에 포함된다.

분포 가설에 기초해 단어를 벡터로 나타내는 방법을 생각해보면
주변 단어를 세어보는 방법이 떠오를 것이며 이를 '통계 기반'기법이라고 한다.

단어 "you"의 맥락을 세어본다.

you say goodbye and i say hello.



단어가 총 7개이며 윈도우 크기는 1로 하고 단어 ID가 0인 'you'부터
단어의 맥락에 해당하는 단어의 빈도를 세어보겠다.
'you'의 맥락은 'say'라는 단어 하나뿐이다.

	you	say	goodbye	and	i	hello	.
you	0	1	0	0	0	0	0

단어 "say"의 맥락을 세어본다.

you **say** goodbye and i **say** hello.

'say'라는 단어는 벡터 [1, 0, 1, 0, 1, 1, 0]으로 표현할 수 있다.

	you	say	goodbye	and	i	hello	.
say	1	0	1	0	1	1	0

모든 단어 각각의 맥락에 해당하는 단어의 빈도를 세어 표로 정리 한다.

	you	say	goodbye	and	i	hello	.
you	0	1	0	0	0	0	0
say	1	0	1	0	1	1	0
goodbye	0	1	0	1	0	0	0
and	0	0	1	0	1	0	0
i	0	1	0	1	0	0	0
hello	0	1	0	0	0	0	1
.	0	0	0	0	0	1	0

위의 표는 모든 단어에 대해 동시발생하는 단어를 표에 정리한 것이다.

위 표의 각 행은 벡터이며 행렬의 형태를 띠어 동시발생 행렬이라 한다.

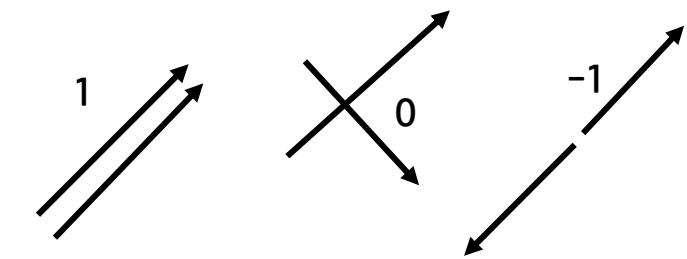
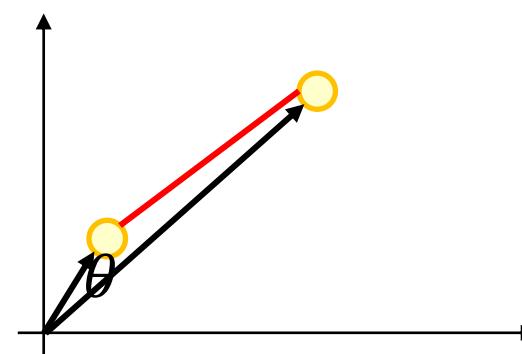
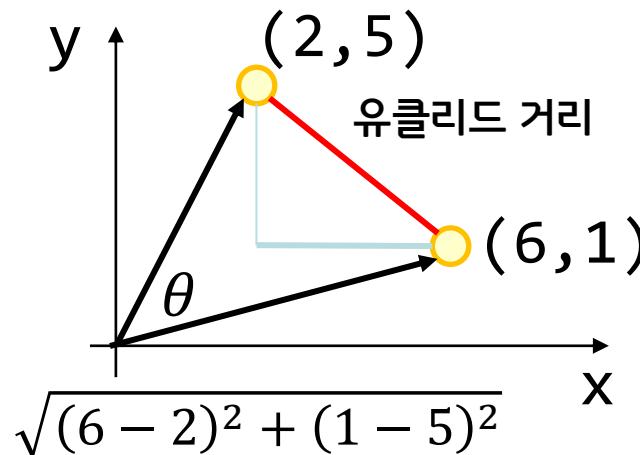
단어 벡터의 유사도를 나타낼 때는 코사인 유사도를 자주 이용한다.

$$\text{similarity}(x, y) = \frac{x \cdot y}{\|x\| \|y\|} = \frac{x_1 y_1 + \cdots + x_n y_n}{\sqrt{x_1^2 + \cdots + x_n^2} \sqrt{y_1^2 + \cdots + y_n^2}}$$

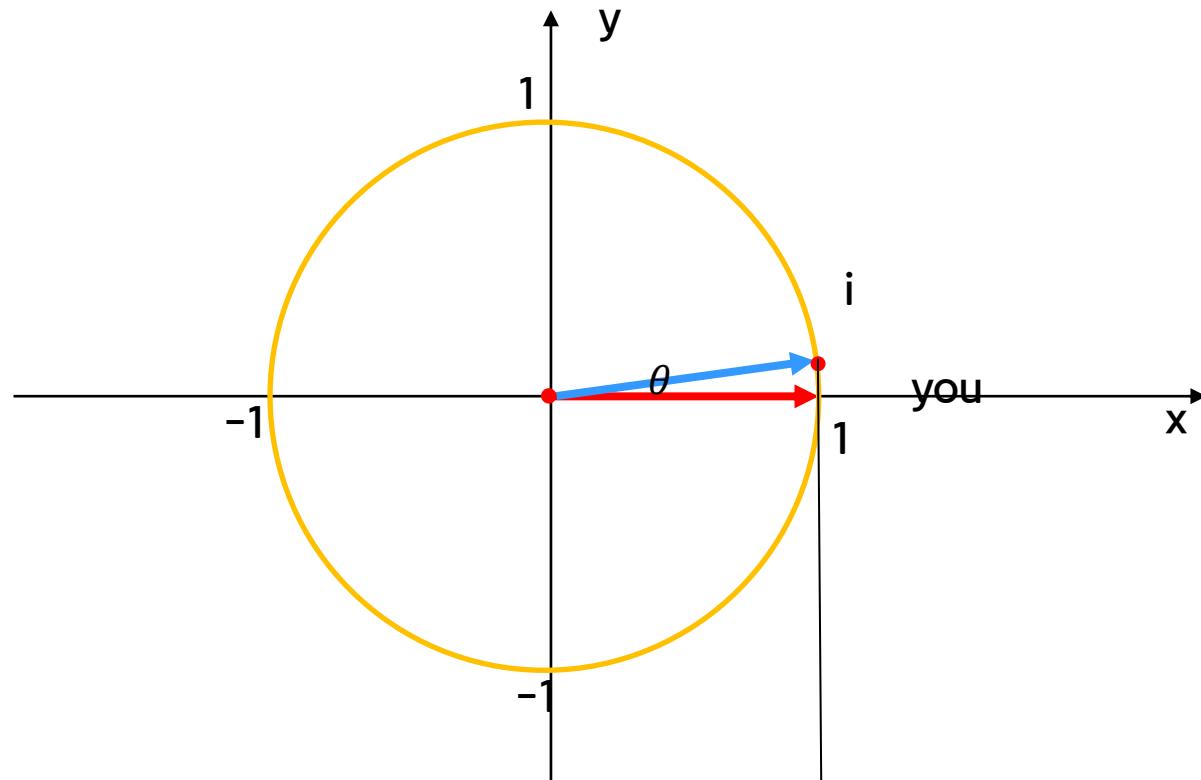
$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

$$\frac{1}{\sqrt{1} * \sqrt{2}} = 0.7071$$

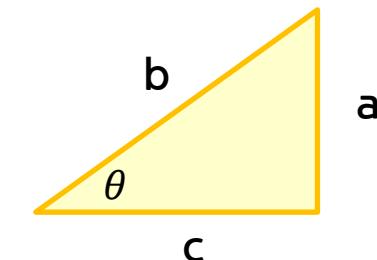
위의 식처럼 정의되며 분자에는 벡터의 내적이 분모에는 벡터의 노름(크기)이 등장한다.
위 식의 핵심은 벡터를 정규화하고 내적을 구하는 것이다.



코사인 유사도



Win : 7 * 3



$$\cos(\theta) = \frac{c}{b}$$

$$\text{similarity}(x, y) = \frac{x \cdot y}{\|x\| \|y\|} = \frac{x_1 y_1 + \cdots + x_n y_n}{\sqrt{x_1^2 + \cdots + x_n^2} \sqrt{y_1^2 + \cdots + y_n^2}}$$

코사인 유사도를 이용하여 어떤 단어가 주어지면,
그 검색어와 비슷한 단어를 유사도 순으로 출력하는 함수를 만들어 본다.

이를 구현하기 위한 코드에는 밑에 같은 함수의 인수들이 쓰인다.

인수명	설명
query	검색어(단어)
word_to_id	단어에서 단어 ID로의 딕셔너리
id_to_word	단어 ID에서 단어로의 딕셔너리
word_matrix	단어 벡터들을 한데 모은 행렬, 각 행에는 대응하는 단어의 벡터가 저장되어 있다고 가정한다.
top	상위 몇 개까지 출력할지 설정

※ 코드 참조

```

def create_co_matrix(corpus, vocab_size, window_size=1):
    corpus_size = len(corpus)
    co_matrix = np.zeros((vocab_size, vocab_size), dtype=np.int32)

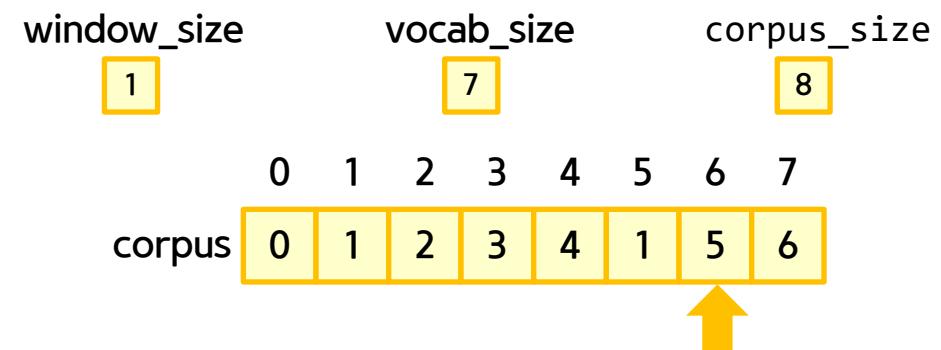
    for idx, word_id in enumerate(corpus):
        for i in range(1, window_size + 1):
            left_idx = idx - i
            right_idx = idx + i

            if left_idx >= 0:
                left_word_id = corpus[left_idx]
                co_matrix[word_id, left_word_id] += 1

            if right_idx < corpus_size:
                right_word_id = corpus[right_idx]
                co_matrix[word_id, right_word_id] += 1

    return co_matrix

```

$$\begin{bmatrix}
 [0 1 0 0 0 0 0] \\
 [1 0 1 0 1 1 0] \\
 [0 1 0 1 0 0 0] \\
 [0 0 1 0 1 0 0] \\
 [0 1 0 1 0 0 0] \\
 [0 1 0 0 0 0 1] \\
 [0 0 0 0 0 1 0]
 \end{bmatrix}$$


0	1	0	0	0	0	0	0
1	0	1	0	1	1	1	0
	1		1				
		1			1		
			1				
				1			
					1		
						1	

2. 자연어와 단어의 분산 표현

2.1 자연어 처리란

2.2 시소러스

2.3 통계 기반 기법

2.4 통계 기반 기법 개선하기

동시발생 행렬의 원소는 두 단어가 동시에 발생한 횟수를 나타내지만
'발생' 횟수라는 것은 사실 좋은 특징이 아니다.

예를 들어 'the' 와 'car'의 동시발생을 생각해보자.

'...the car...'라는 문구가 자주 보일 것이며 'car'와 'drive'는 관련이 깊다.

하지만 'the'가 고빈도 단어이기 때문에 'car'와 더 관련이 있어 보이게 결과가 나올 수 있다.

이를 해결하기 위해 점별 상호정보량이라는 척도를 사용한다.

PMI(Pointwise Mutual Information)는 확률 변수 x 와 y 에 대해 다음 식으로 정의 된다.

$$PMI(x, y) = \log_2 \frac{P(x, y)}{P(x)P(y)}$$

$P(x)$ 는 x 가 일어날 확률, $P(y)$ 는 y 가 일어날 확률, $P(x,y)$ 는 x,y 가 동시에 일어날 확률이다.
PMI값이 높을수록 관련성이 높다는 의미이다.

$$PMI(x, y) = \log_2 \frac{P(x, y)}{P(x)P(y)} = \log_2 \frac{\frac{C(x, y)}{N}}{\frac{C(x)}{N} \frac{C(y)}{N}} = \log_2 \frac{C(x, y) \cdot N}{C(x)C(y)}$$

여기서 C는 동시발생 행렬, C(x,y)는 단어 x와 y가 동시발생하는 횟수,
 C(x)와 C(y)는 각각 단어 x와 y의 등장 횟수이며 N은 말뭉치에 포함된 단어 수이다.

이 식을 토대로 1,000번 등장한 'the', 20번 등장한 'car'와 10번 등장한 'drive'를 계산해보자.

$$N=10000, \quad C(x, y)=10, \quad C(x)=1000, \quad C(y)=20$$

$$PMI("the", "car") = \log_2 \frac{10 \cdot 10000}{1000 \cdot 20} \approx 2.32$$

$$N=10000, \quad C(x, y)=5, \quad C(x)=20, \quad C(y)=10$$

$$PMI("car", "drive") = \log_2 \frac{5 \cdot 10000}{20 \cdot 10} \approx 7.97$$

두 PMI의 결과를 살펴보면 'car'와 'drive'의 관계성이 강하다는 것을 볼 수 있다.
이러한 결과가 나온 이유는 단어가 단독으로 출현하는 횟수가 고려되었기 때문이다.
이 예에서는 'the'가 자주 출현하였기 때문에 PMI값이 낮아진 것이다.

하지만 PMI에도 문제가 하나 있다.

이는 두 단어의 동시발생 횟수가 0이면 $\log(0, 2) = -\text{infinite}$ 가 된다.
이 문제를 피하기 위해 실제 구현할 때는 양의 상호정보량(PPMI)을 사용한다.

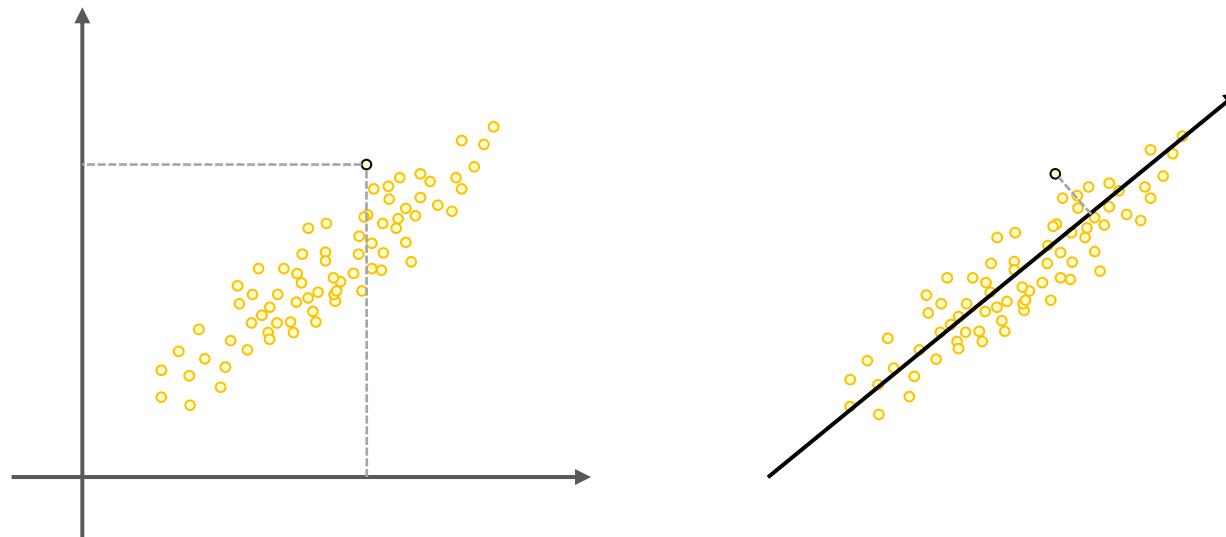
$$\text{PPMI}(x, y) = \max(0, \text{PMI}(x, y))$$

이 식에 따라 PMI가 음수인 때는 0으로 취급하여 단어 사이의 관련성을
0 이상의 실수로 나타낼 수 있다.

하지만 PPMI 행렬에도 문제가 있는데 말뭉치의 어휘 수가 증가함에 따라
각 단어 벡터의 차원 수도 증가한다는 문제이다.

이 문제를 대처하고자 자주 수행하는 기법이 '벡터의 차원 감소'이다.

그림으로 이해하는 차원 감소: 2차원 데이터를 1차원으로 표현하기 위해 중요한 축을 찾는다.



위의 그림 예시처럼 데이터의 분포를 고려해 중요한 '축'을 찾는 일을 수행한다.
왼쪽 그림은 데이터점들을 2차원 좌표에 표시한 모습이고
오른쪽 그림은 새로운 축을 도입하여 똑같은 데이터를 좌표축 하나만으로 표시했다.

여기서 중요한 것은 가장 적합한 축을 찾아내는 일로,
1차원 값만으로 데이터의 본질적인 차이를 구별할 수 있어야 한다.
그리고 다차원 데이터에 대해서도 수행 가능하다.

차원을 감소시키는 방법 중 하나인 특잇값분해(SVD)는 임의의 행렬을 세 행렬의 곱으로 분해하며, 수식으로는 다음과 같다.

$$(n,m)(m,n)$$

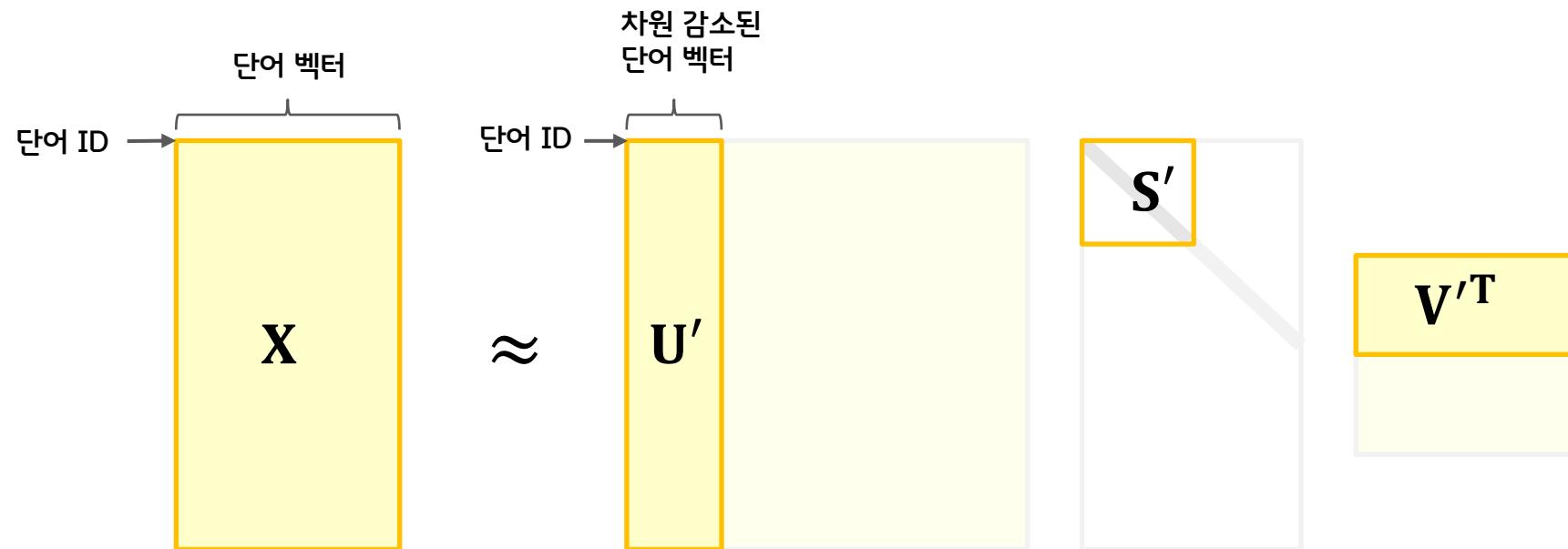
$$X = USV^T$$

SVD에 대한 행렬의 변환(행렬의 '흰 부분'은 원소가 0임을 뜻함)

$$\begin{array}{c}
 (m,n) \\
 \boxed{X} \\
 \\
 = \\
 \boxed{U} \\
 \\
 (m,n)(n,m) \quad XX^T
 \end{array}
 \qquad
 \begin{array}{c}
 (m,m) \\
 \boxed{S} \\
 \\
 (m,n)
 \end{array}
 \qquad
 \begin{array}{c}
 (n,n) \\
 \boxed{V^T} \\
 \\
 X^T X \\
 (n,m)(m,n)
 \end{array}$$

행렬 S 에서 특 값이 작다면 중요도가 낮다는 뜻이므로
행렬 U 에서 여분의 열벡터를 깎아내려 원래의 행렬을 근사할 수 있다.

SVD에 의한 차원의 감소



이를 '단어의 PPMI 행렬'에 적용하면 행렬 X 의 각 행에는 해당 단어 ID의 단어 벡터가 저장되어 있으며, 그 단어 벡터가 행렬 U' 라는 차원 감소된 벡터로 표현된다.

우리가 사용할 PTB(펜 트리뱅크) 말뭉치는 word2vec의 발명자인 토마스 미콜로프의 웹 페이지에서 받을 수 있다.

※ 코드 참조

결과적으로 말뭉치를 사용해 맥락에 속한 단어의 등장 횟수를 센 후 PPMI 행렬로 변환하고 다시 SVD를 이용해 차원을 감소시킴으로서 더 좋은 단어 벡터를 얻었다.
이것이 단어의 분산 표현이고, 각 단어는 고정 길이의 밀집벡터로 표현되었다.

3. word2vec

3.1 추론 기반 기법과 신경망

3.2 단순한 word2vec

3.3 학습 데이터 준비

3.4 CBOW 모델 구현

3.5 word2vec 보충

단어를 벡터로 표현하는 방법은 크게 두 부분이 있다.

1. 통계 기반 기법
2. 추론 기반 기법

단어의 의미를 얻는 방식은 서로 크게 다르지만,
그 배경에는 모두 분포 가설이 있다.

이번 절에서는 통계 기반 기법의 문제를 지적하고,
그 대안인 추론 기반 기법의 이점을 거시적 관점에서 설명한다.

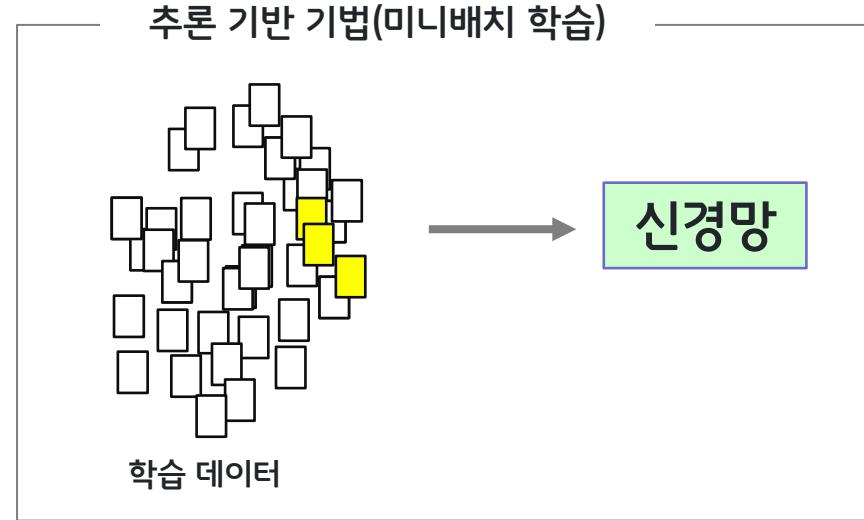
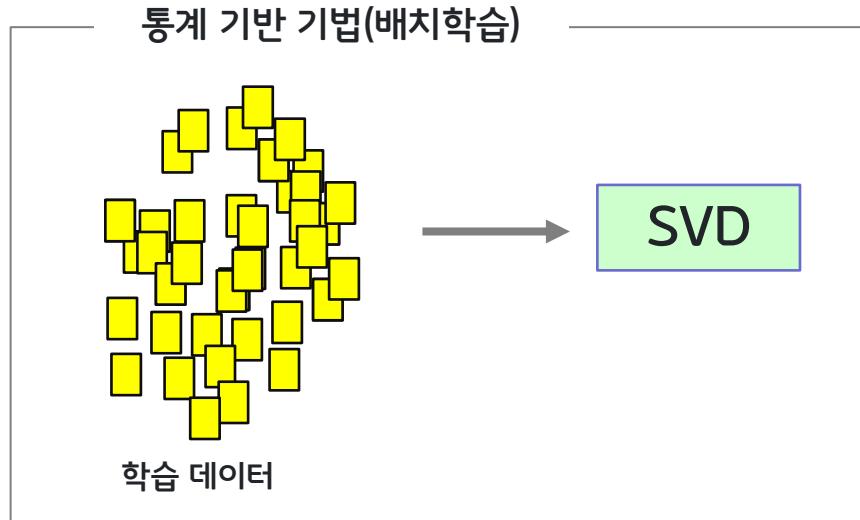
지금까지 본 것처럼 통계 기반 기법에서는 주변 단어의 빈도를 기초로 단어를 표현했다.
구체적으로는 단어의 동시발생 행렬을 만들고 그 행렬에 SVD를 적용하여
밀집벡터:단어의 분산 표현을 얻었다.

그러나 이 방식은 대규모 말뭉치를 다룰 때 문제가 발생한다.

현업에서 다루는 말뭉치의 어휘 수는 어마어마하다.
이런 거대 행렬에 SVD를 적용하는 일은 현실적이지 않다.

SVD를 $n*n$ 행렬에 적용하는 비용은 $O(n^3)$ 이다.

통계 기반 기법과 추론 기반 기법 비교



통계 기반 기법은 학습 데이터를 한꺼번에 처리한다.

배치학습 추론 기반 기법은 학습 데이터의 일부를 사용하여 순차적으로 학습한다.

미니배치 학습 :

말뭉치의 어휘 수가 많아 SVD 등 계산량이 큰 작업을 처리하기
어려운 경우에도 신경망을 학습시킬 수 있다는 의미이다.
데이터를 작게 나눠 학습하기 때문이다.

추론 기반 기법에서는 추론이 주된 작업이다.
추론이란, 주변 단어: 맥락이 주어졌을 때,
?에 무슨 단어가 들어가는지를 추측하는 작업이다.

주변 단어들의 맥락으로 사용해 "?"에 들어갈 단어를 추측한다.

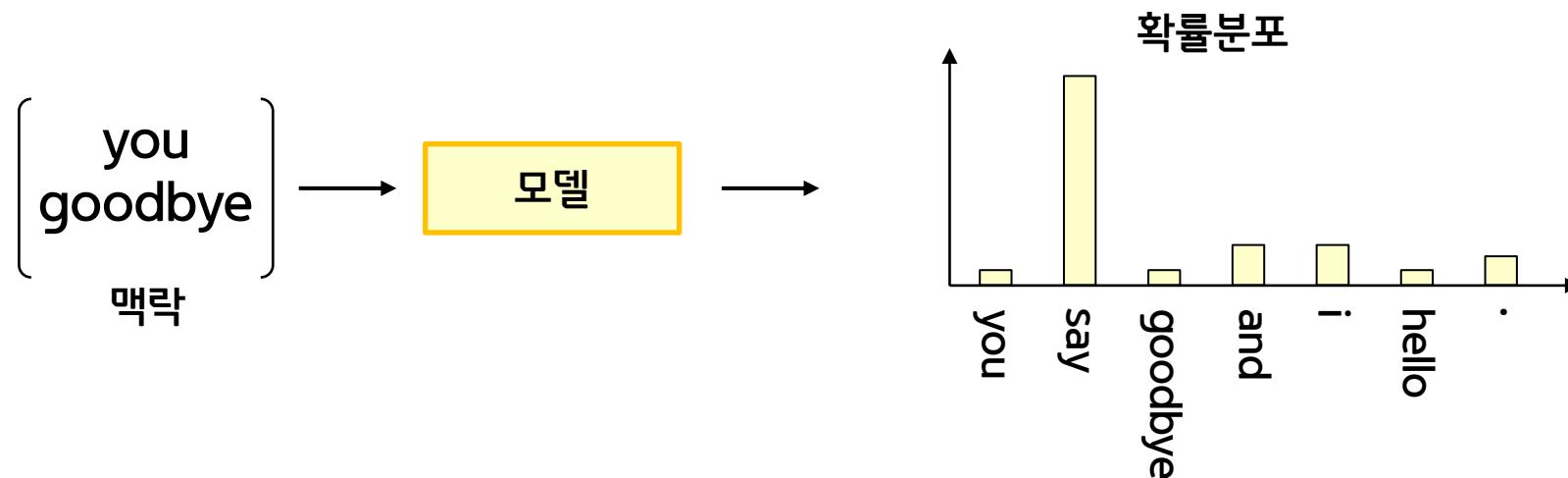
you **say** goodbye and i say hello.



추론 문제를 풀고 학습하는 것이 추론 기반 기법이 다루는 문제이다.
이러한 추론 문제를 반복해서 풀면서 단어의 출현 패턴을 학습하는 것이다.

모델 관점에서 보면, 추론 문제는 다음과 같다.

추론 기반 기법: 맥락을 입력하면 모델은 각 단어의 출현 확률을 출력한다.



추론 기반 기법에는 어떠한 모델이 등장한다.

우리는 이 모델로 신경망을 사용한다.

모델은 맥락 정보를 입력 받아 출현할 수 있는 각 단어의 출현 확률을 출력한다.

이러한 틀 안에서 말뭉치를 사용해 모델이 올바른 추측을 내놓도록 학습시킨다.

그리고 그 학습의 결과로 단어의 분산 표현을 얻는 것이 추론 기반 기법의 전체 그림이다.

지금부터 신경망을 이용해 단어를 처리해보자.

단어를 있는 그대로 처리할 수 없으니 고정 길이의 벡터로 변환해야 한다.

이때 사용하는 대표적인 방법이 단어를 원 핫 표현으로 변환하는 것이다.

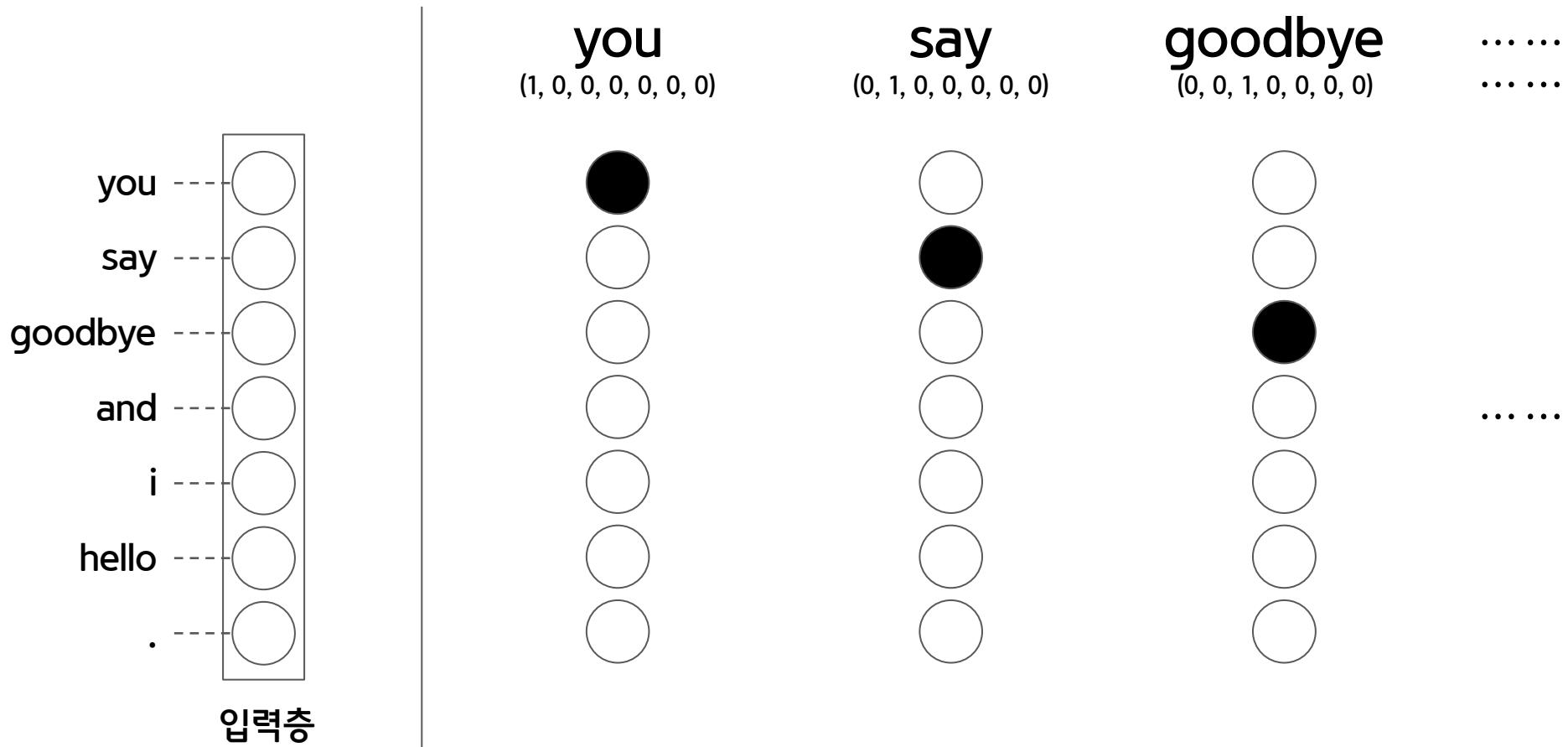
원 핫 표현이란, 벡터의 원소 중 하나만 1이고, 나머지는 모두 0인 벡터를 말한다.

단어(텍스트)	단어 ID	원 핫 표현
$\begin{bmatrix} \text{you} \\ \text{goodbye} \end{bmatrix}$	$\begin{bmatrix} 0 \\ 2 \end{bmatrix}$	$\begin{bmatrix} (1, 0, 0, 0, 0, 0, 0) \\ (0, 0, 1, 0, 0, 0, 0) \end{bmatrix}$

단어를 원 핫 표현으로 변환하는 방법

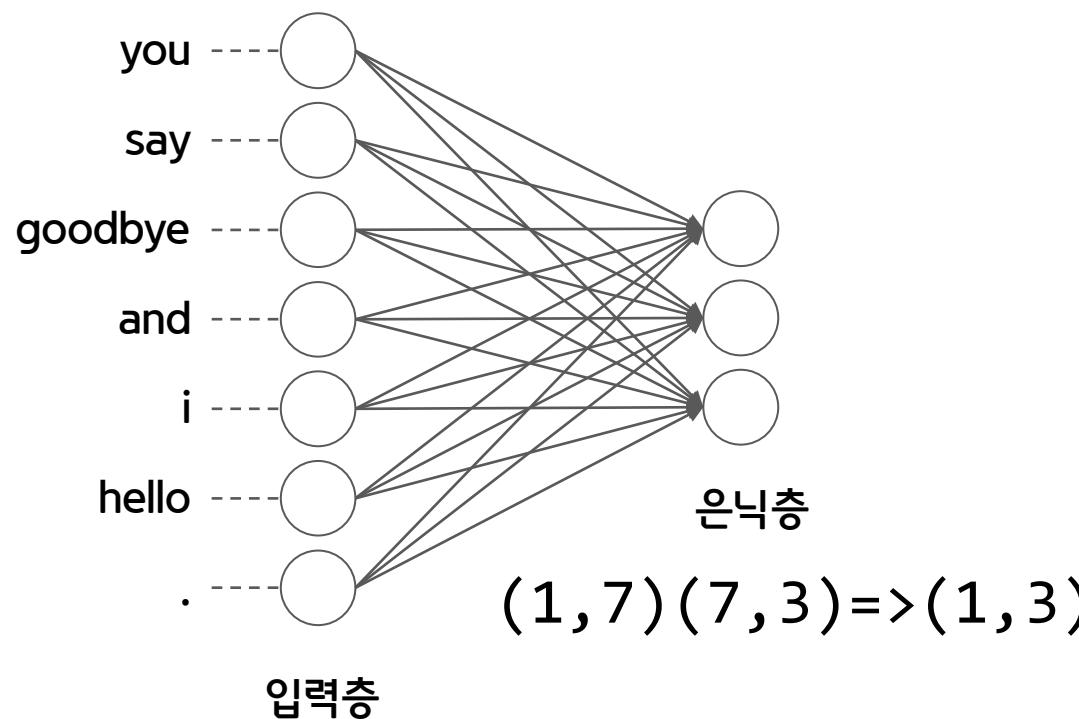
- 먼저 총 어휘 수만큼의 원소를 갖는 벡터를 준비하고,
- 인덱스가 단어 ID 와 같은 원소를 1로, 나머지는 모두 0으로 설정한다.
이처럼 단어를 고정 길이 벡터로 변환하면, 신경망의 입력층은 뉴런의 수를 고정할 수 있다.

입력층의 뉴런: 각 뉴런이 각 단어에 대응한다.(해당 뉴런이 1이면 검은색, 0이면 흰색)



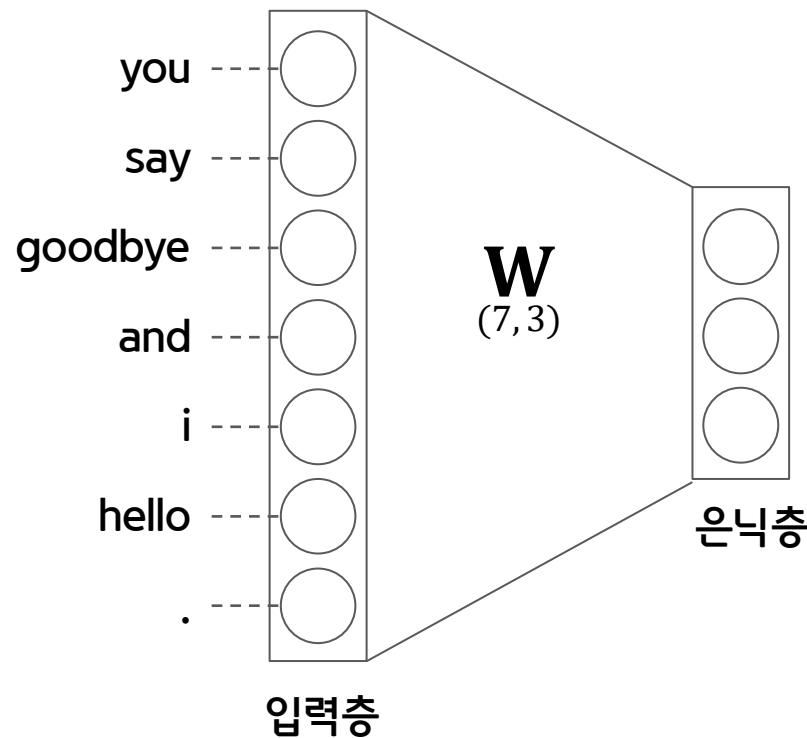
단어를 벡터로 나타낼 수 있고, 신경망을 구성하는 계층들은 벡터를 처리할 수 있다.
다시 말해, 단어를 신경망으로 처리할 수 있다는 뜻이다.

완전연결층에 의한 변환 : 입력층의 각 뉴런은 7개의 단어 각각에 대응(은닉층의 뉴런은 3개를 준비함)



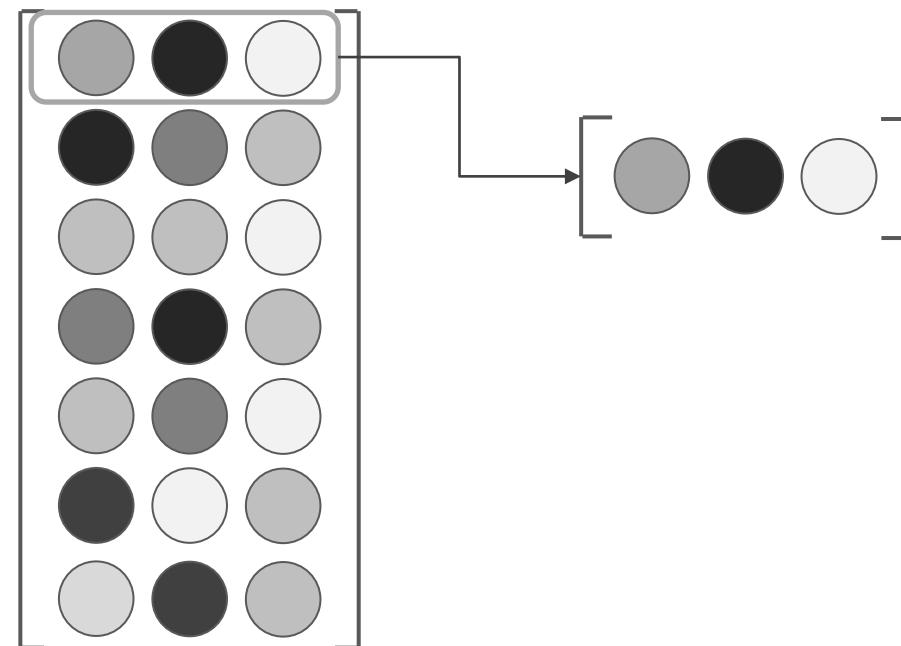
화살표에는 가중치: 매개변수가 존재하여, 입력층 뉴런과의 가중합이 은닉층 뉴런이 된다.
 간단한 설명을 위해 완전연결계층에서는 편향을 생략했다.
 편향을 이용하지 않은 완전연결계층은 행렬 곱 계산에 해당한다.

완전연결층에 의한 변환을 단순화한 그림(완전연결계층의 가중치를 7×3 크기의 W라는 행렬로 표현)



맥락 c 와 가중치 W 의 곱으로 해당 위치의 행벡터가 추출된다.
(각 요소의 가중치 크기는 흑백의 진하기로 표현)

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

 c W $=$ h

$$(1, 7)(7, 3) \Rightarrow (1, 3)$$

3. word2vec

3.1 추론 기반 기법과 신경망

3.2 단순한 word2vec

3.3 학습 데이터 준비

3.4 CBOW 모델 구현

3.5 word2vec 보충

앞 절에서 추론 기반 기법을 배우고, 신경망으로 단어를 처리하는 방법을 코드로 살펴보았다.
이제 word2vec 을 구현할 차례이다.

지금부터 할 일은 모델을 신경망으로 구축하는 것이다.
이번 절에서 사용할 신경망은

word2vec 에서 제안하는 CBOW, continuous bag-of-words 모델이다.

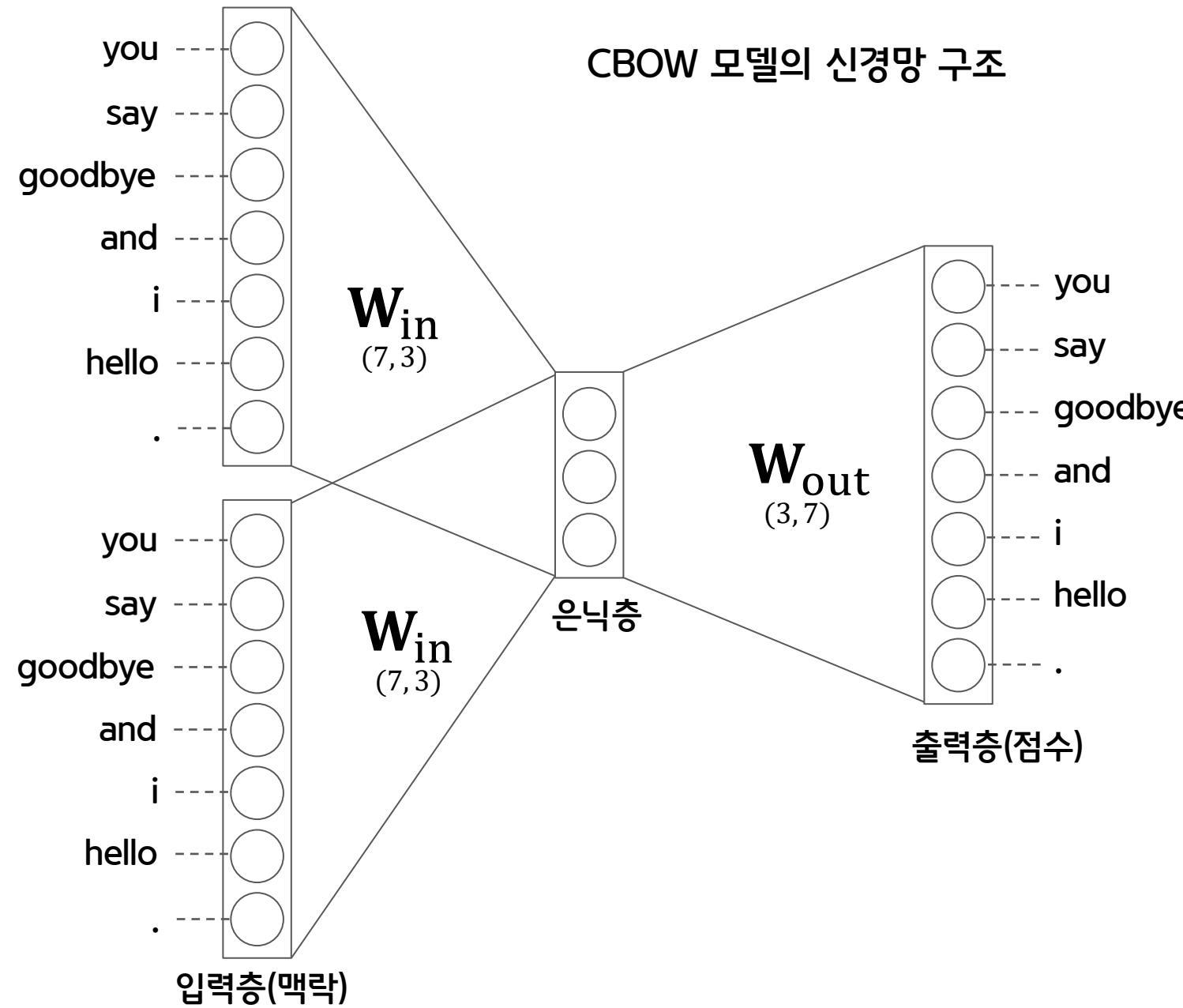
CBOW 모델은 맥락으로부터 타깃을 추측하는 용도의 신경망이다.

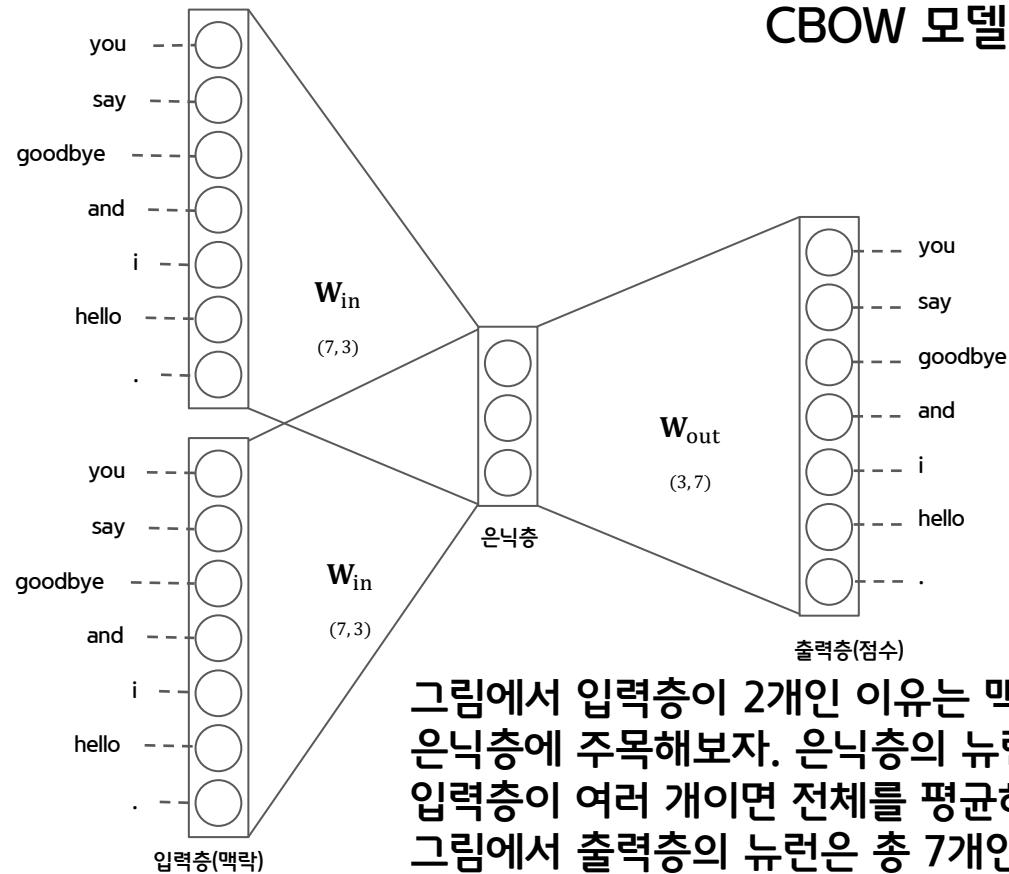
타깃은 중앙 단어이고, 그 주변 단어들이 맥락이다.

우리는 이 CBOW 모델이 가능한 한 정확하게 추론하도록 훈련시켜서 단어의 분산 표현을 얻어낼 것이다.

CBOW 모델의 입력은 맥락이다.

가장 먼저, 이 맥락을 원핫 표현으로 변환하여 CBOW 모델이 처리할 수 있도록 준비한다.

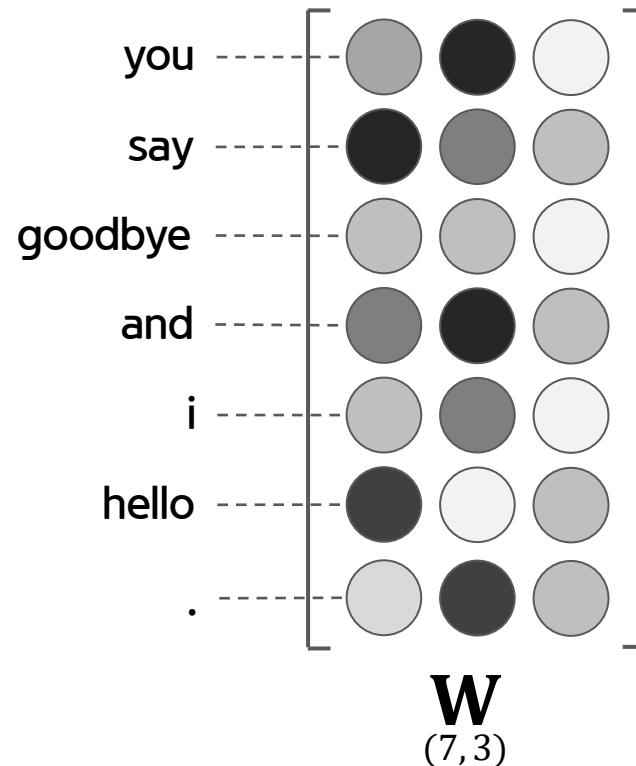




CBOW 모델의 신경망 구조

그림에서 입력층이 2개인 이유는 맥락으로 고려할 단어를 2개로 정했기 때문이다.
은닉층에 주목해보자. 은닉층의 뉴런은 입력층의 완전연결계층에 의해 변환된 값이 되는데,
입력층이 여러 개이면 전체를 평균하면 된다.
그림에서 출력층의 뉴런은 총 7개인데, 중요한 것은 이 뉴런 하나하나가
각각의 단어에 대응한다는 점이다.
그리고 출력층 뉴런은 각 단어의 점수를 뜻하며, 값이 높을수록 대응 단어의 출현 확률도 높아진다.
여기서 점수란, 확률로 해석되기 전의 값이고, 이 점수에 소프트맥스
함수를 적용해서 확률을 얻을 수 있다.
점수를 Softmax 계층에 통과시킨 후의 뉴런을 출력층이라고도 한다.

가중치의 각 행의 해당 단어의 분산 표현이다.



은닉층의 뉴런 수를 입력 층의 뉴런 수보다 적게 하는 것이 중요한 핵심이다.

이렇게 해야 은닉층에는 단어 예측에 필요한 정보를 간결하게 담게 되며,

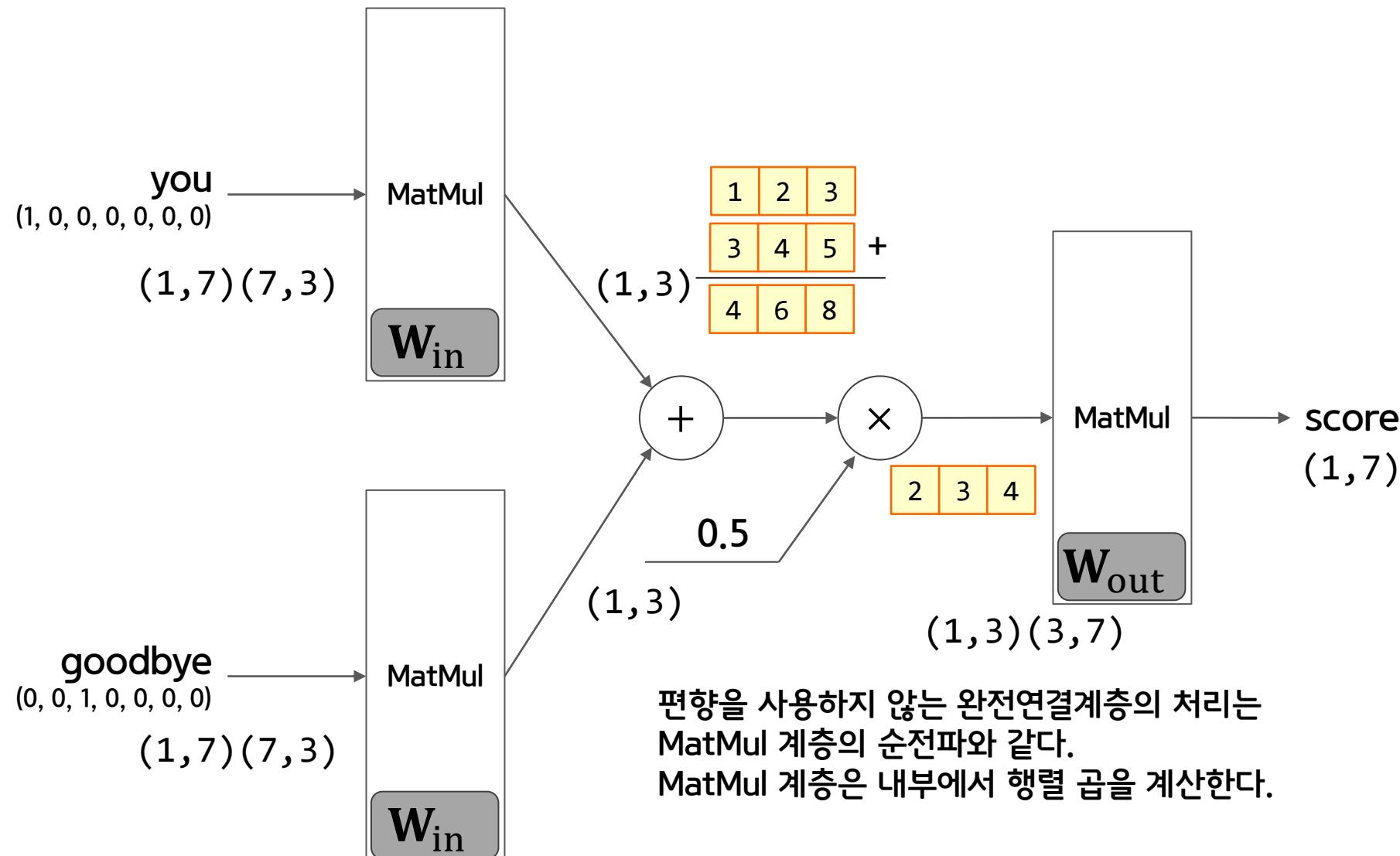
결과적으로 밀집벡터 표현을 얻을 수 있다.

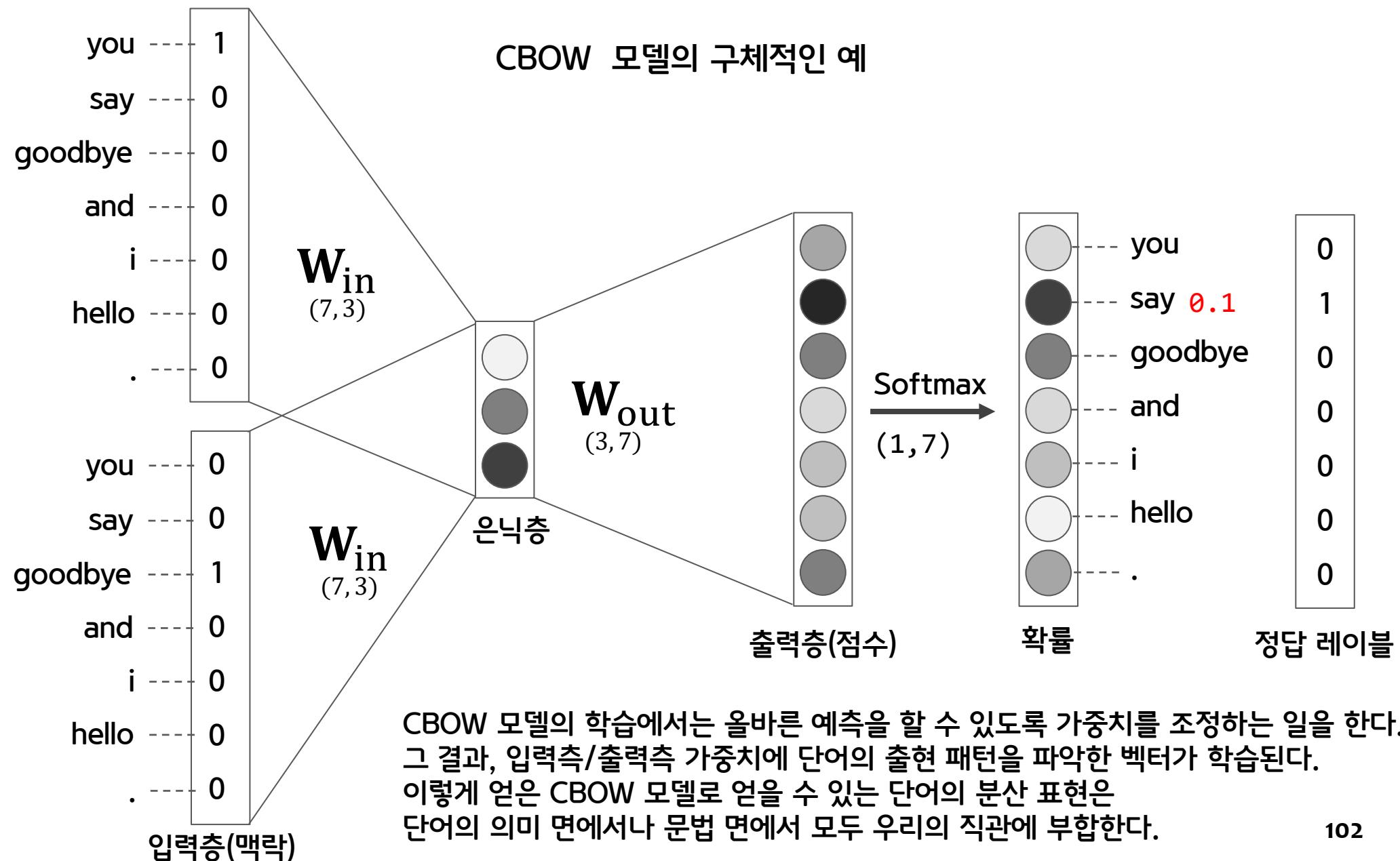
이때 은닉층 정보는 인간이 이해할 수 없는 코드로 쓰여 있다. (인코딩)

한편, 은닉층의 정보로부터 원하는 결과를 얻는 작업은 디코딩이라고 한다.

즉, 디코딩이란 인코딩된 정보를 인간이 이해할 수 있는 표현으로 복원하는 작업이다.

계층 관점에서 본 CBOW 모델의 신경망 구성





신경망의 학습에 대해 생각해보자.

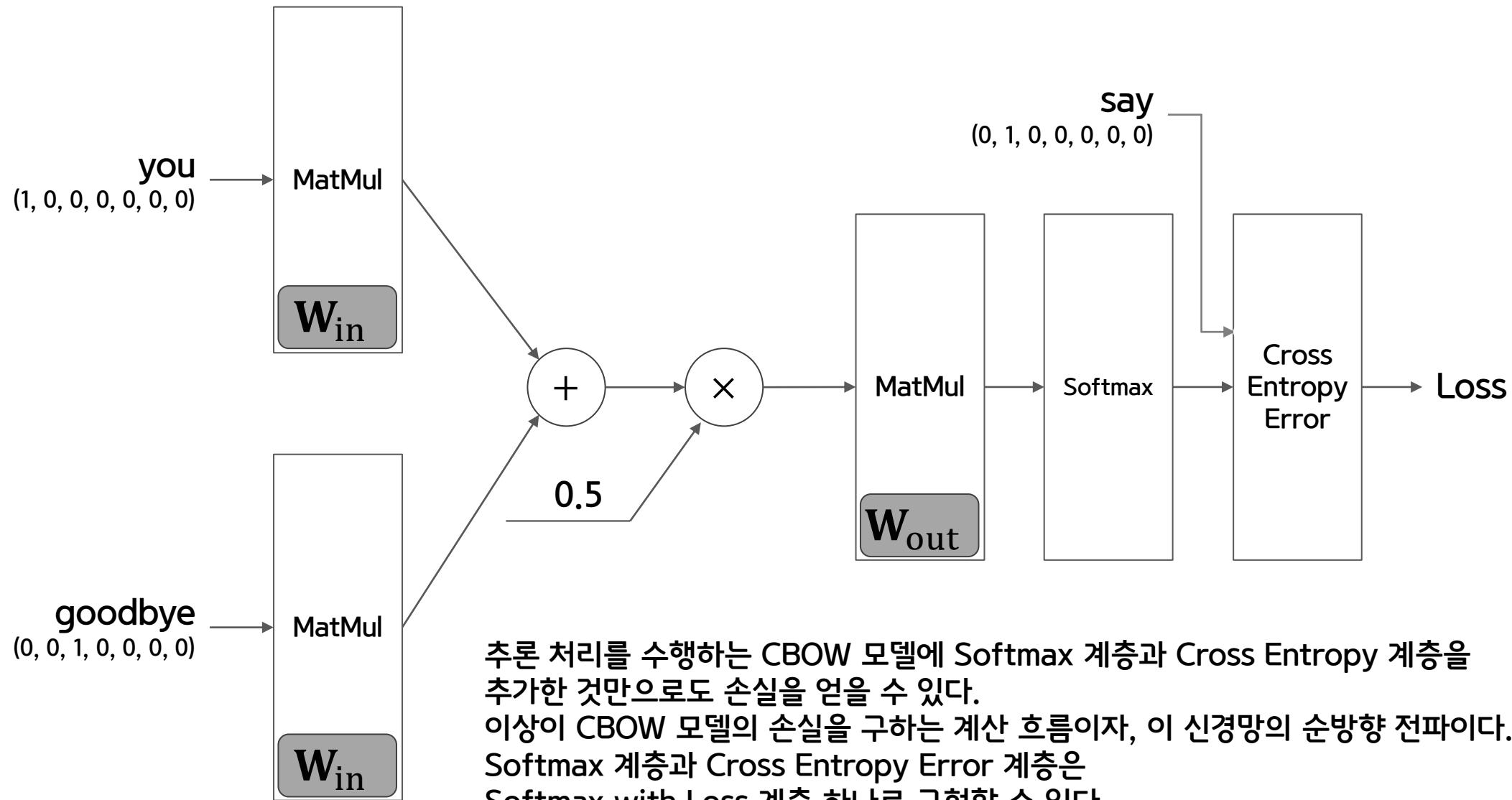
우리가 다루는 모델은 다중 클래스 분류를 수행하는 신경망이다.

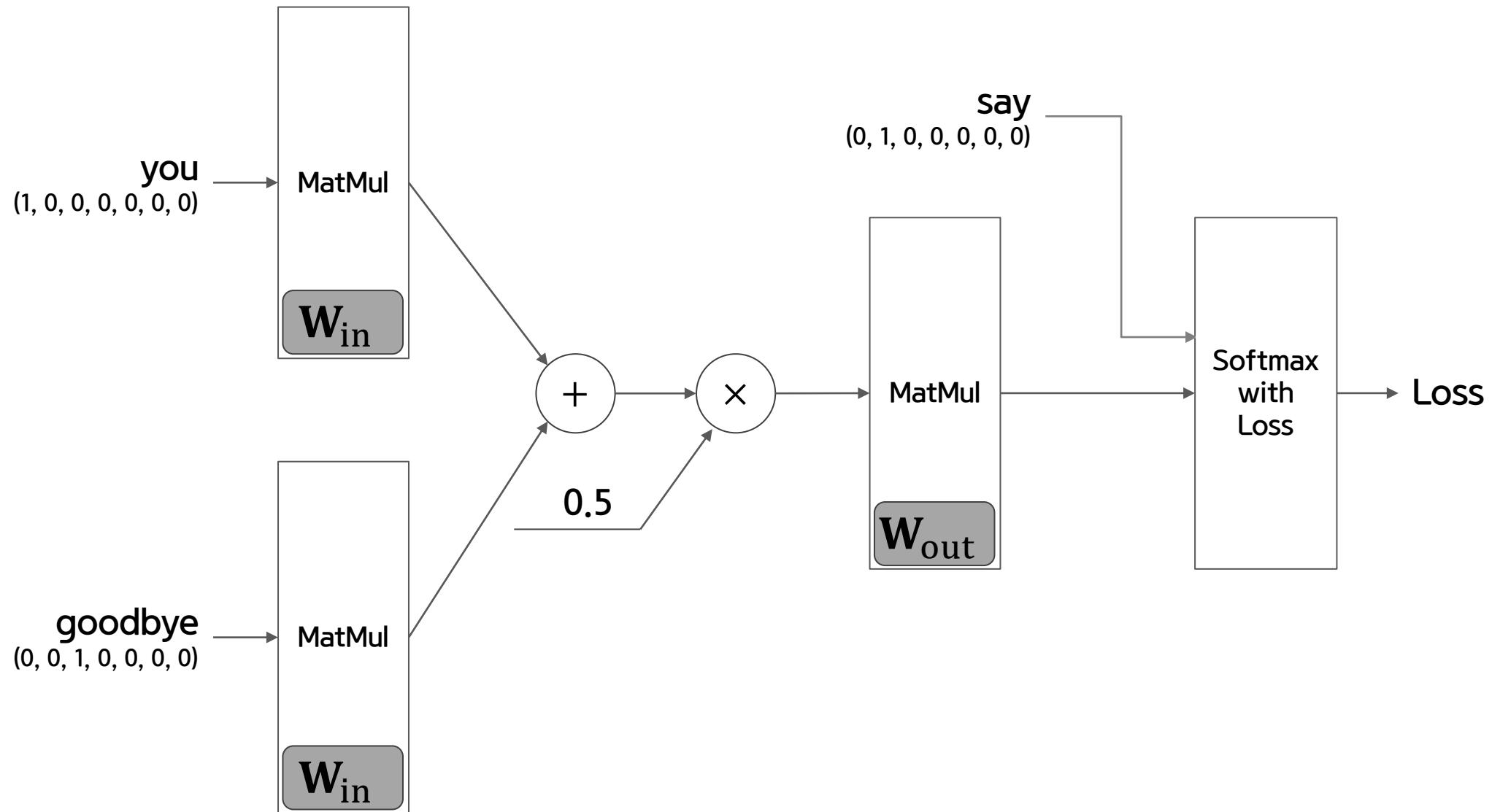
따라서 이 신경망을 학습하려면, 소프트맥스 함수와 교차 엔트로피 오차만 이용하면 된다.

소프트맥스를 이용해 점수를 확률로 변환하고,

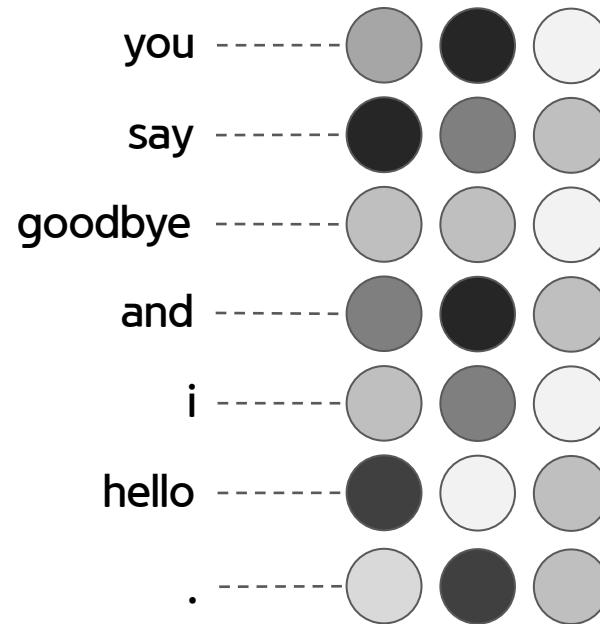
그 확률과 정답 레이블로부터 교차 엔트로피 오차를 구한 후,

그 값을 손실로 사용해 학습을 진행한다.



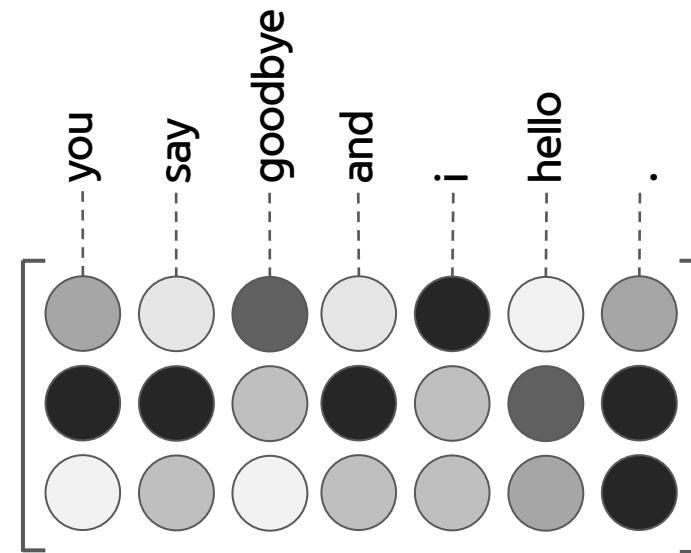


각 단어의 분산 표현은 입력 측과 출력 측 모두의 가중치에서 확인할 수 있다.



$$W_{in} \quad (7, 3)$$

(V, H)



$$W_{out} \quad (3, 7)$$

(H, V)

그러면 최종적으로 이용하는 단어의 분산 표현으로는 어느 쪽 가중치를 사용하면 좋을까?
선택지는 3가지.

1. 입력 측의 가중치만 이용
2. 출력 측의 가중치만 이용
3. 양쪽 가중치를 모두 이용

word2vec, 특히 skip-gram 모델에서는 입력 측 가중치만 이용하는 것이 가장 대중적이다.

3. word2vec

3.1 추론 기반 기법과 신경망

3.2 단순한 word2vec

3.3 학습 데이터 준비

3.4 CBOW 모델 구현

3.5 word2vec 보충

word2vec에서 이용하는 신경망의 입력은 맥락이다.

그리고 정답 레이블은 맥락에 둘러싸인 중앙의 단어, 즉 타깃이다.

우리가 해야 할 일은 신경망에 맥락을 입력했을 때 타깃이 출현할 확률을 높이는 것이다.

말뭉치에서 맥락과 타깃을 만드는 예

말뭉치	맥락(contexts)	타깃
you say goodbye and i say hello .	you, goodbye	say
you say goodbye and i say hello .	say, and	goodbye
you say goodbye and i say hello .	goodbye, i	and
you say goodbye and i say hello .	and, say	i
you say goodbye and i say hello .	i, hello	say
you say goodbye and i say hello .	say, .	hello

맥락과 타깃을 원핫 표현으로 변환하는 예

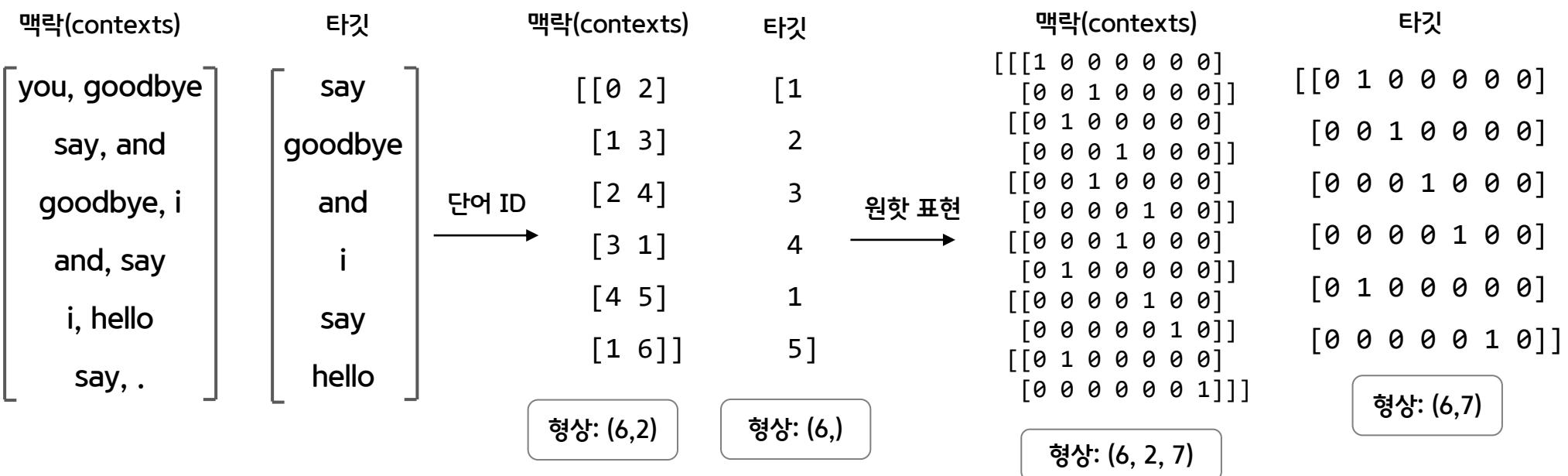
말뭉치	맥락(contexts)	타깃
[0 1 2 3 4 1 5 6]	[[0 2]	[1
	[1 3]	2
	[2 4]	3
	[3 1]	4
	[4 5]	1
	[1 6]]	5]

형상: (8,)

형상: (6,2)

형상: (6,)

맥락과 타깃을 원핫 표현으로 변환하는 예



```

def create_contexts_target(corpus, window_size=1):
    target = corpus[window_size:-window_size]
    contexts = []

    for idx in range(window_size, len(corpus)-window_size):
        cs = []
        for t in range(-window_size, window_size + 1):
            if t == 0:
                continue
            cs.append(corpus[idx + t])
        contexts.append(cs)

    return np.array(contexts), np.array(target)

contexts, target = create_contexts_target(corpus, window_size=1)

```

cs = [1,3]

corpus = [0 1 2 3 4 1 5 6]

target = [1 2 3 4 1 5]

contexts = [[0,2],[1,3],[2,4],[3,1],[4,5],[1,6]]

[1 2 3 4 1 5]

target

1 2 3 4 1 5

contexts

0 2	[0 2]
1 3	[1 3]
2 4	[2 4]
3 1	[3 1]
4 5	[4 5]
1 6	[1 6]

```

def create_contexts_target(corpus, window_size=2):
    target = corpus[window_size:-window_size]
    contexts = []

    for idx in range(window_size, len(corpus)-window_size):
        cs = []
        for t in range(-window_size, window_size + 1):
            if t == 0:
                continue
            cs.append(corpus[idx + t])
        contexts.append(cs)

    return np.array(contexts), np.array(target)

contexts, target = create_contexts_target(corpus, window_size=1)

```

[2 3 4 1]

target

2	3	4	1
---	---	---	---

contexts

0	1	3	4
1	2	4	1
2	3	1	5
3	4	5	6

[[0 1 3 4]

[1 2 4 1]

[2 3 1 5]

[3 4 5 6]]

corpus = [0 1 2 3 4 ↓ 1 5 6]

cs = [0,1,3,4]

target = [2 3 4 1]

contexts = [[0,1,3,4],[1,2,4,1],[2,3,1,5],[3,4,5,6]]

```

def convert_one_hot(corpus, vocab_size):
    N = corpus.shape[0]

    if corpus.ndim == 1:
        one_hot = np.zeros((N, vocab_size), dtype=np.int32)
        for idx, word_id in enumerate(corpus):
            one_hot[idx, word_id] = 1

    elif corpus.ndim == 2:
        C = corpus.shape[1]
        one_hot = np.zeros((N, C, vocab_size), dtype=np.int32)
        for idx_0, word_ids in enumerate(corpus):
            for idx_1, word_id in enumerate(word_ids):
                one_hot[idx_0, idx_1, word_id] = 1

    return one_hot

target = convert_one_hot(target, vocab_size)

```

N=6

vocab_size = 7

corpus

0	1	2	3	4	5
1	2	3	4	1	5

one_hot (6,7)

0	1	2	3	4	5	6
1		1				
2			1			
3				1		
4	1					
5					1	

```

[[0 1 0 0 0 0 0]
 [0 0 1 0 0 0 0]
 [0 0 0 1 0 0 0]
 [0 0 0 0 1 0 0]
 [0 1 0 0 0 0 0]
 [0 0 0 0 0 1 0]]

```

```

def convert_one_hot(corpus, vocab_size):
    N = corpus.shape[0]

    if corpus.ndim == 1:
        one_hot = np.zeros((N, vocab_size), dtype=np.int32)
        for idx, word_id in enumerate(corpus):
            one_hot[idx, word_id] = 1

    elif corpus.ndim == 2:
        C = corpus.shape[1]
        one_hot = np.zeros((N, C, vocab_size), dtype=np.int32)
        for idx_0, word_ids in enumerate(corpus):
            for idx_1, word_id in enumerate(word_ids):
                one_hot[idx_0, idx_1, word_id] = 1

    return one_hot

contexts = convert_one_hot(contexts, vocab_size)

```

corpus	vocab_size = 7 N=6 C=2																																																	
<table border="1"> <tr><td>0</td><td>2</td></tr> <tr><td>1</td><td>3</td></tr> <tr><td>2</td><td>4</td></tr> <tr><td>3</td><td>1</td></tr> <tr><td>4</td><td>5</td></tr> <tr><td>1</td><td>6</td></tr> </table>	0	2	1	3	2	4	3	1	4	5	1	6	one_hot (6,2,7)																																					
0	2																																																	
1	3																																																	
2	4																																																	
3	1																																																	
4	5																																																	
1	6																																																	
<table border="1"> <tr><td>1</td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td>1</td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td>1</td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td>1</td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td>1</td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td>1</td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td>1</td></tr> </table>	1								1								1								1								1								1								1	[[[1 0 0 0 0 0 0] [0 0 1 0 0 0 0]]]
1																																																		
	1																																																	
		1																																																
			1																																															
				1																																														
					1																																													
						1																																												
<table border="1"> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> </table>																																																		[[[0 1 0 0 0 0 0] [0 0 0 1 0 0 0]]]
<table border="1"> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> </table>																																																		[[[0 0 1 0 0 0 0] [0 0 0 0 1 0 0]]]
<table border="1"> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> </table>																																																		[[[0 0 0 1 0 0 0] [0 0 0 0 1 0 0]]]
<table border="1"> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> </table>																																																		[[[0 0 0 0 1 0 0] [0 0 0 0 0 1 0]]]
<table border="1"> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr> </table>																																																		[[[0 1 0 0 0 0 0] [0 0 0 0 0 0 1]]]]

3. word2vec

3.1 추론 기반 기법과 신경망

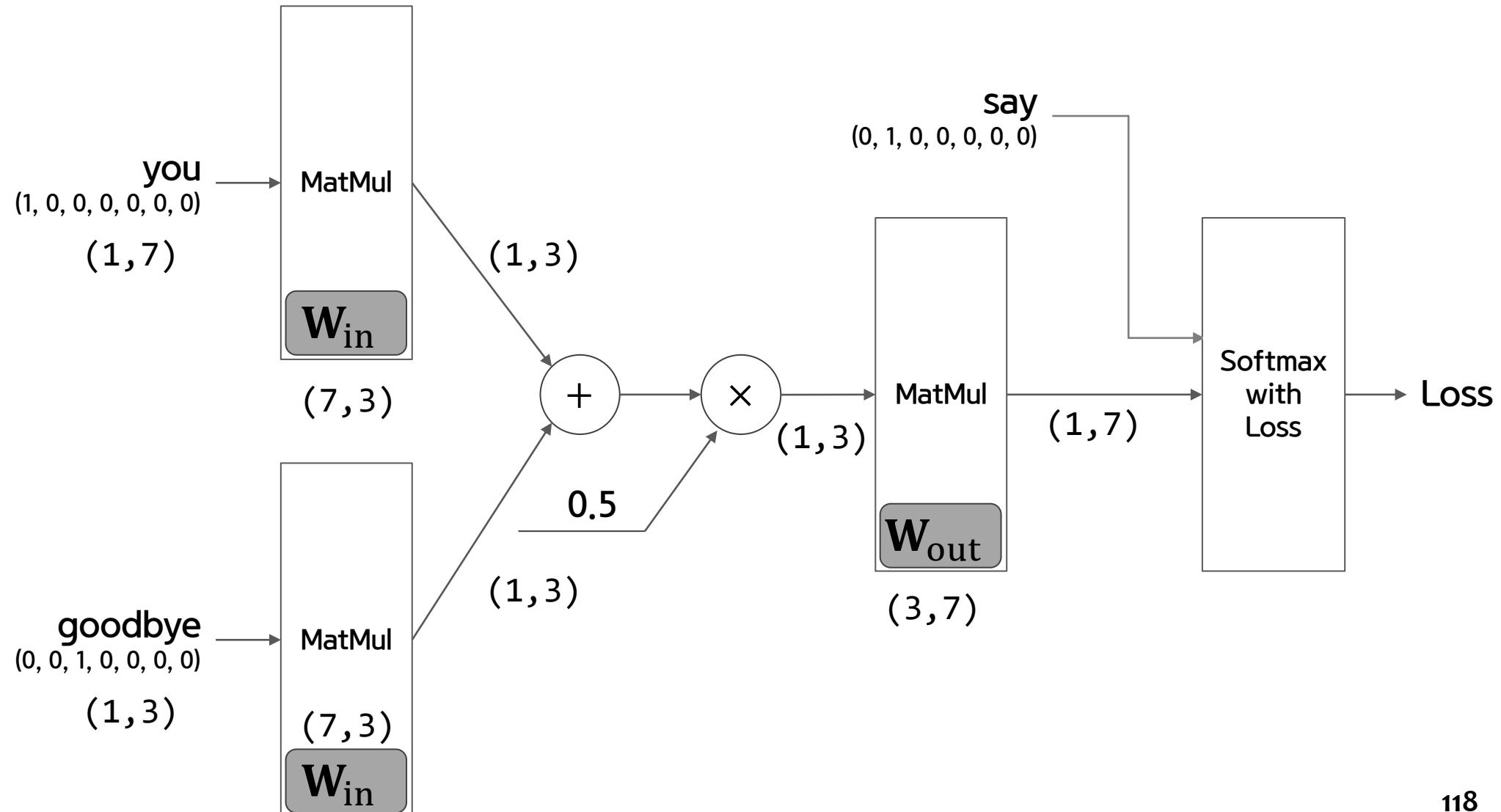
3.2 단순한 word2vec

3.3 학습 데이터 준비

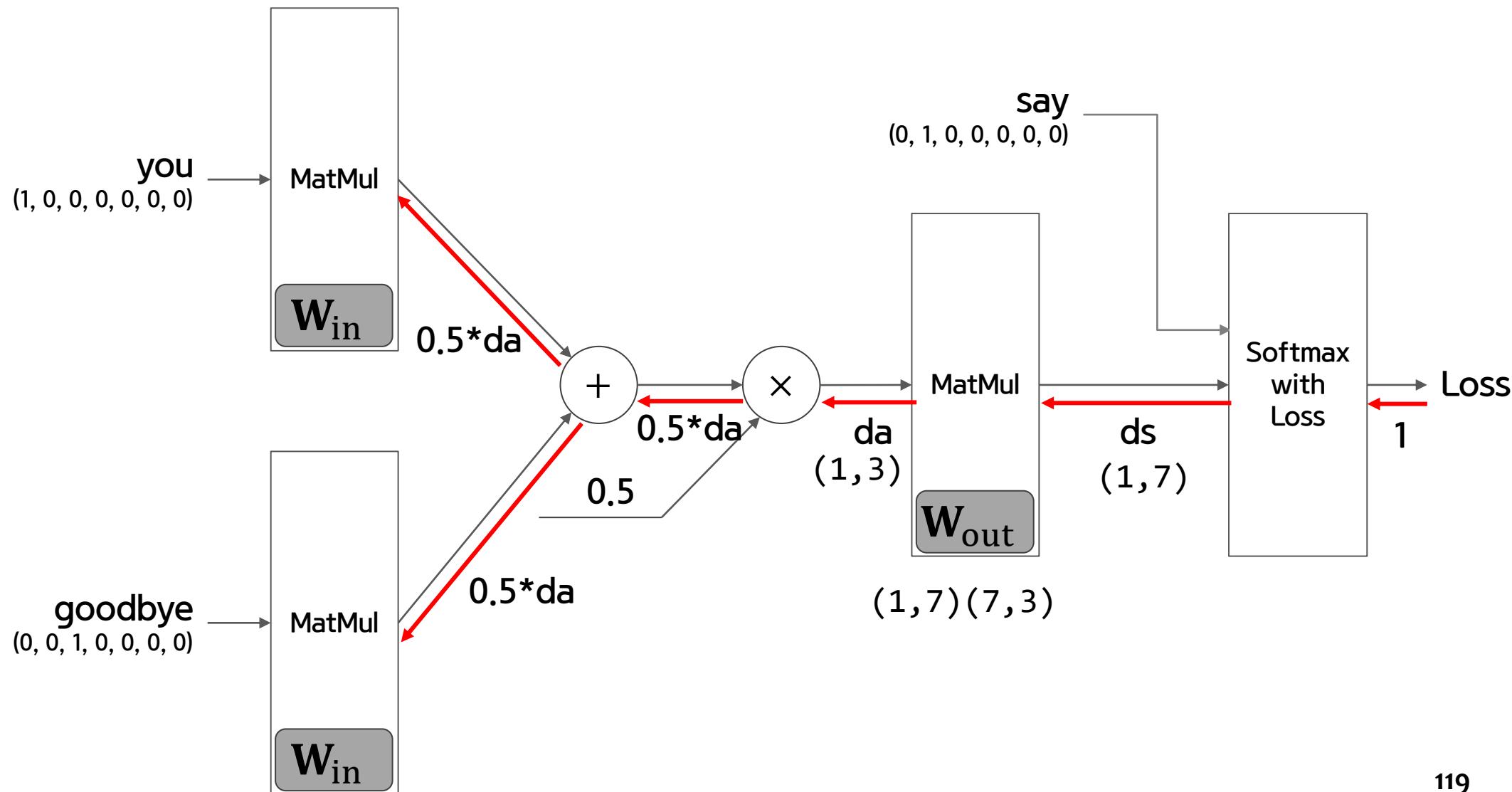
3.4 CBOW 모델 구현

3.5 word2vec 보충

CBOW 모델의 신경망 구성



CBOW 모델의 역전파



3. word2vec

3.1 추론 기반 기법과 신경망

3.2 단순한 word2vec

3.3 학습 데이터 준비

3.4 CBOW 모델 구현

3.5 word2vec 보충

확률의 표기법을 간단하게 살펴보자.

확률 $P()$

동시 확률 $P(A, B)$, A와 B가 동시에 일어날 확률.

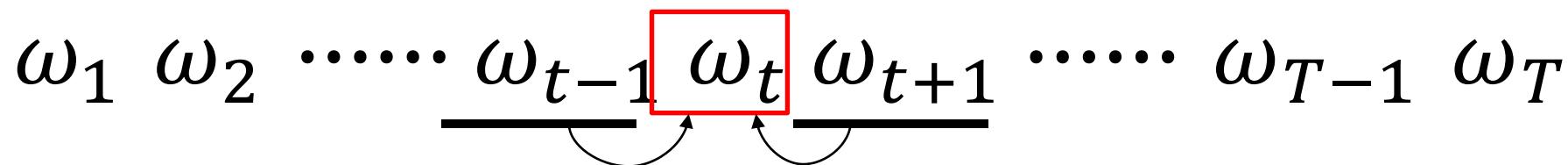
사후 확률 $P(A|B)$, 사건이 일어난 후의 확률.

B라는 정보가 주어졌을 때, A가 일어날 확률.

그럼 CBOW 모델을 확률 표기법으로 기술해보자.

CBOW 모델이 하는 일은, 맥락을 주면 타깃 단어가 출현할 확률을 출력하는 것이다.

word2vec 의 CBOW 모델 - 맥락의 단어로부터 타깃 단어를 추측



CBOW 모델은 다음 식을 모델링하고 있다.

$$P(w_t | w_{t-1}, w_{t+1})$$

위 식을 이용하면 CBOW 모델의 손실 함수도 간결하게 표현할 수 있다.

교차 엔트로피 오차를 적용해보자.

다음 식을 유도할 수 있다.

$$L = -\log P(w_t | w_{t-1}, w_{t+1})$$

이 식을 보듯, CBOW 모델의 손실 함수는 단순히 식의 확률에 \log 를 취한 다음 마이너스를 붙이면 된다. 덧붙여 식은 샘플 데이터 하나에 대한 손실 함수이며, 이를 말뭉치 전체로 확장하면 다음 식이 된다.

$$L = -\frac{1}{T} \sum_{t=1}^T \log P(w_t | w_{t-1}, w_{t+1})$$

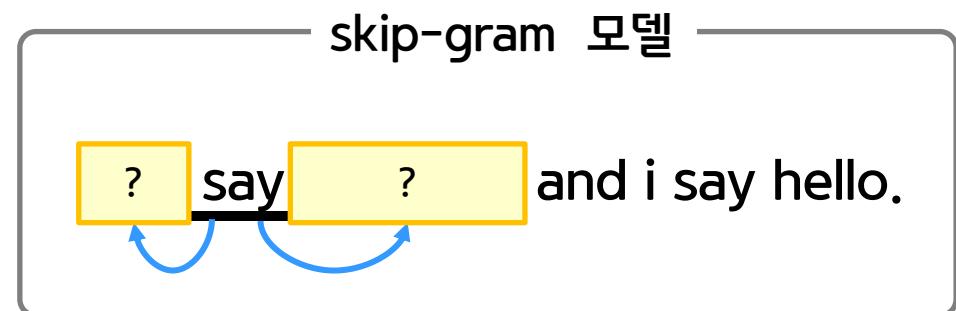
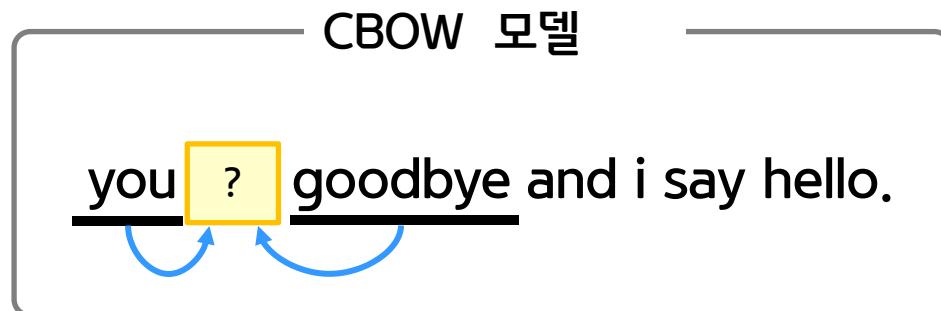
CBOW 모델의 학습이 수행하는 일은, 손실 함수 식을 가능한 작게 만드는 것이다.
그리고 이때의 가중치 매개변수가 우리가 얻고자 하는 단어의 분산 표현이다.

word2vec은 2개의 모델을 제안하고 있다.

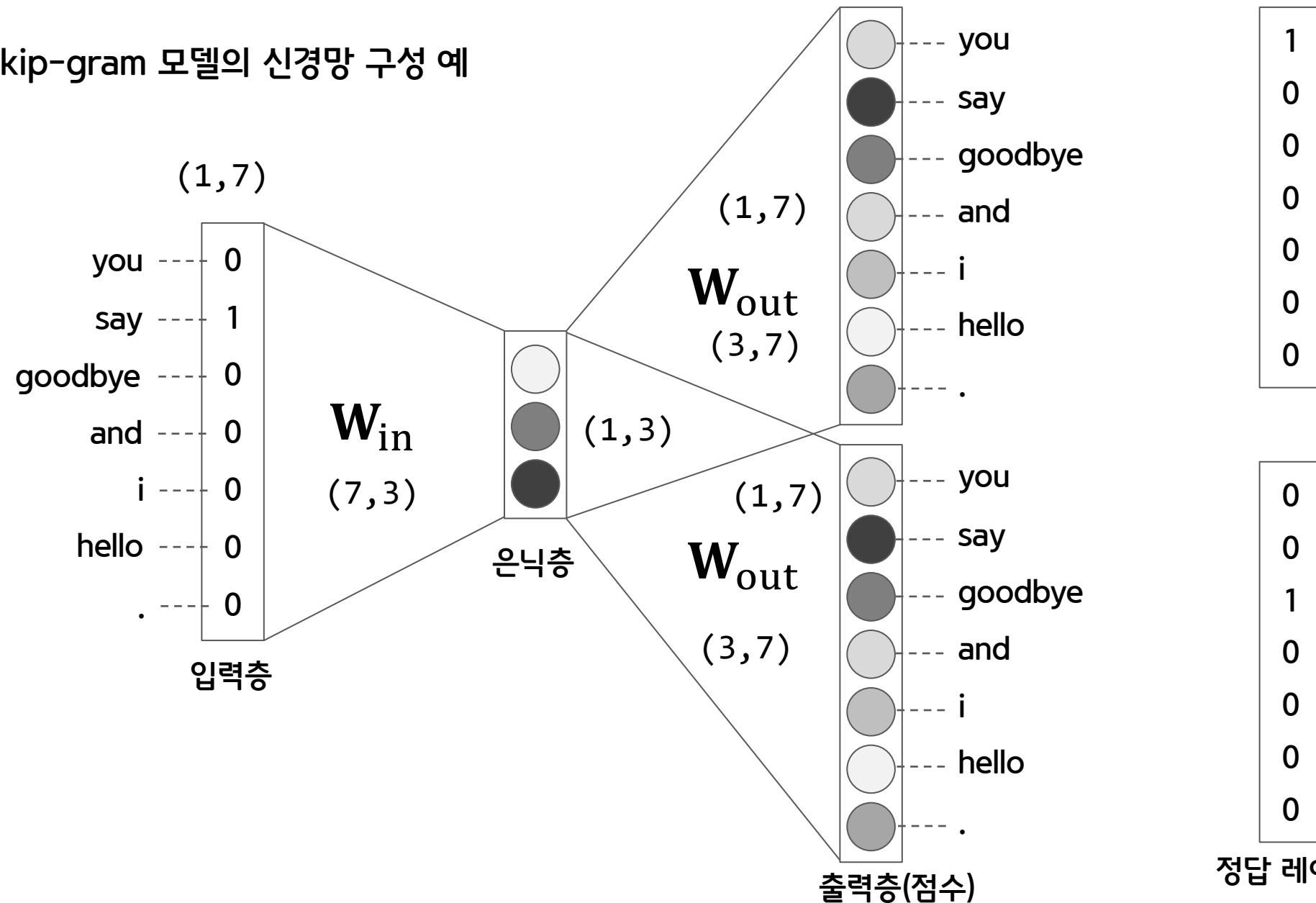
1. CBOW 모델
2. skip-gram 모델

skip-gram 은 CBOW 에서 다루는 맥락과 타깃을 역전시킨 모델.

CBOW 모델과 skip-gram 모델이 다루는 문제



skip-gram 모델의 신경망 구성 예



skip-gram 모델을 확률 표기로 나타내보자.
skip-gram 은 다음 식을 모델링한다.

$$P(w_{t-1}, w_{t+1} | w_t)$$

skip-gram 모델에서는 맥락의 단어들 사이에 관련성이 없다고 가정하고, 다음과 같이 분해한다.

$$P(w_{t-1}, w_{t+1} | w_t) = P(w_{t-1} | w_t) P(w_{t+1} | w_t)$$

위 식을 교차 엔트로피 오차에 적용하여 skip-gram 모델의 손실 함수를 유도할 수 있다.

$$\begin{aligned} L &= -\log P(w_{t-1}, w_{t+1} | w_t) \\ &= -\log P(w_{t-1} | w_t) P(w_{t+1} | w_t) \\ &= -(\log P(w_{t-1} | w_t) + \log P(w_{t+1} | w_t)) \end{aligned}$$

위 식에서 알 수 있듯, skip-gram 모델의 손실 함수는 맥락별 손실을 구한 다음 모두 더한다.
위 식은 샘플 데이터 하나짜리 skip-gram 의 손실 함수이다.

이를 말뭉치 전체로 확장하면 skip-gram 모델의 손실 함수는 다음과 같다.

$$L = -\frac{1}{T} \sum_{t=1}^T (\log P(w_{t-1}|w_t) + \log P(w_{t+1}|w_t))$$

위 식을 CBOW 모델의 식과 비교해보자.

$$L = -\frac{1}{T} \sum_{t=1}^T \log P(w_t|w_{t-1}, w_{t+1})$$

skip-gram 모델은 맥락의 수만큼 추측하기 때문에,
그 손실 함수는 각 맥락에서 구한 손실의 총합이어야 한다.
반면, CBOW 모델은 타깃 하나의 손실을 구한다.

그렇다면, CBOW 모델과 skip-gram 모델 중 어느 것을 사용해야 할까?

답은 skip-gram.

단어 분산 표현의 정밀도 면에서 skip-gram 모델의 결과가 더 좋은 경우가 많기 때문이다.

특히 말뭉치가 커질수록 저빈도 단어나 유추 문제의 성능 면에서 skip-gram 모델이 더 뛰어난 경향이 있다.

반면, 학습 속도 면에서는 CBOW 모델이 더 빠르다.

skip-gram 모델은 손실을 맥락의 수만큼 구해야 해서 계산 비용이 그만큼 커지기 때문이다.

다행히 CBOW 모델의 구현을 이해할 수 있다면, skip-gram 모델의 구현도 특별히 어려울 게 없다.

통계 기반 기법에서는 주로 단어의 유사성이 인코딩된다.

한편 word2vec, 특히 skip-gram 모델에서는 단어의 유사성은 물론, 한층 복잡한 단어 사이의 패턴까지도 파악되어 인코딩된다.

추론 기반 기법이 통계 기반 기법보다 정확하다고 흔히 오해하곤 한다.

하지만 단어의 유사성을 정량 평가해본 결과, 추론 기반과 통계 기반 기법의 우열을 가릴 수 없었다고 한다.

추론 기반 기법과 통계 기반 기법은 서로 관련되어 있다.

word2vec 이후 추론 기반 기법과 통계 기반 기법을 융합한 GloVe 기법이 등장했다.

GloVe 의 기본 아이디어는, 말뭉치 전체의 통계 정보를 손실 함수에 도입해 미니배치 학습을 하는 것이다.

4. word2vec 속도 개선

4.1 word2vec 개선 I

4.2 word2vec 개선 II

4.3 개선판 word2vec 학습

word2vec의 속도 개선하는 법을 알아보겠다.

앞서 3장에서 보았던 CBOW(Continuous Bag of Words) 모델은 처리 효율이 떨어져 말뭉치에 포함된 어휘 수가 많아지면 계산량도 커진다.

따라서, 단순한 word2vec에 두가지 개선을 추가한다.

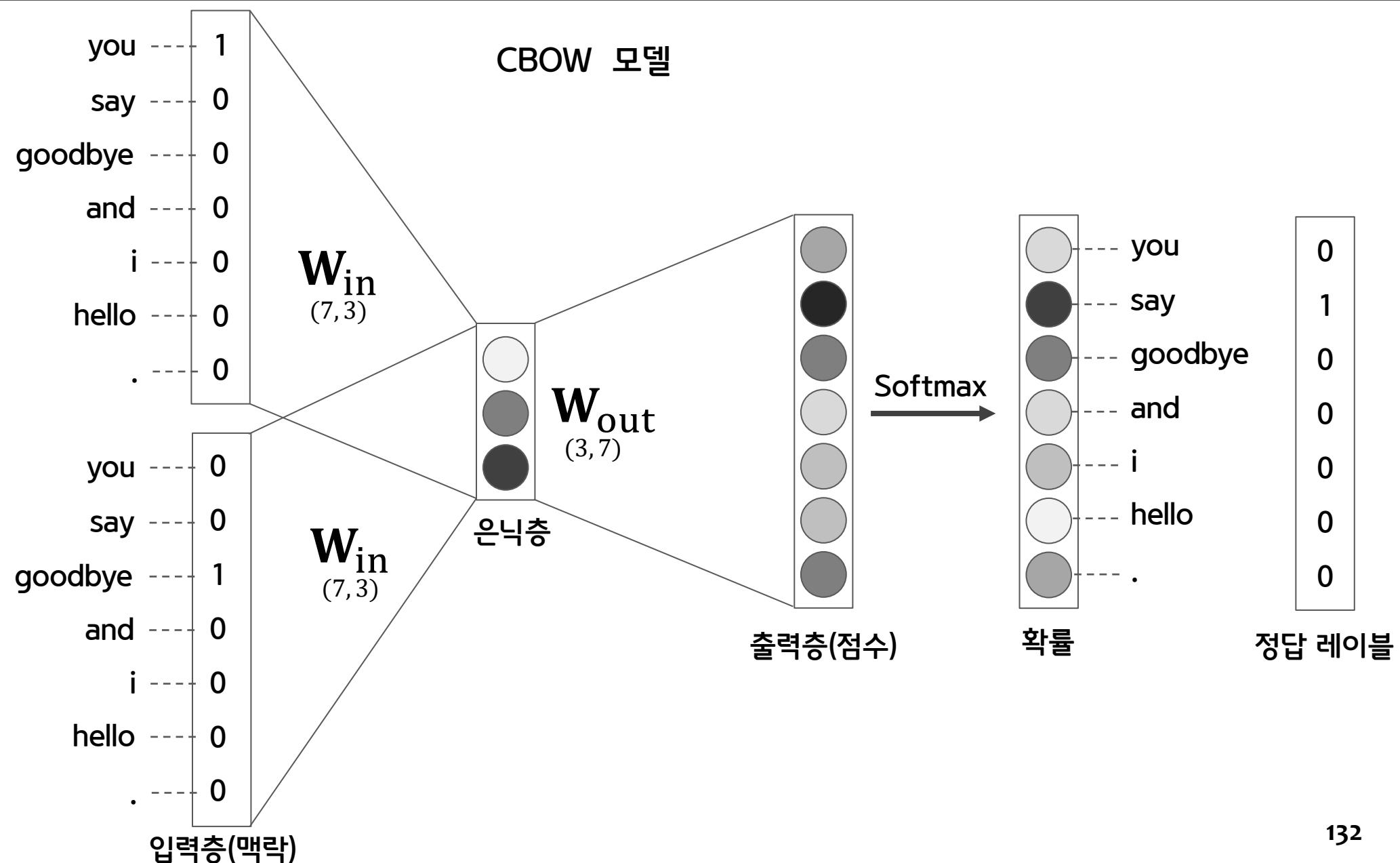
1. Embedding 이라는 새로운 계층을 만든다.
2. 네거티브 샘플링 이라는 새로운 손실함수를 도입한다.

이로써 '진짜'word2vec을 완성할 수 있다.

완성된 word2vec 모델을 가지고 PTB 데이터셋(=실용적인 크기의 말뭉치)를 가지고 학습을 수행하고, 결과를 평가해 보도록 하자.

CBOW 모델은, 복수 단어 문맥에 대한 문제 즉, 여러개의 단어를 나열한 뒤 이와 관련된 단어를 추정하는 문제이다.

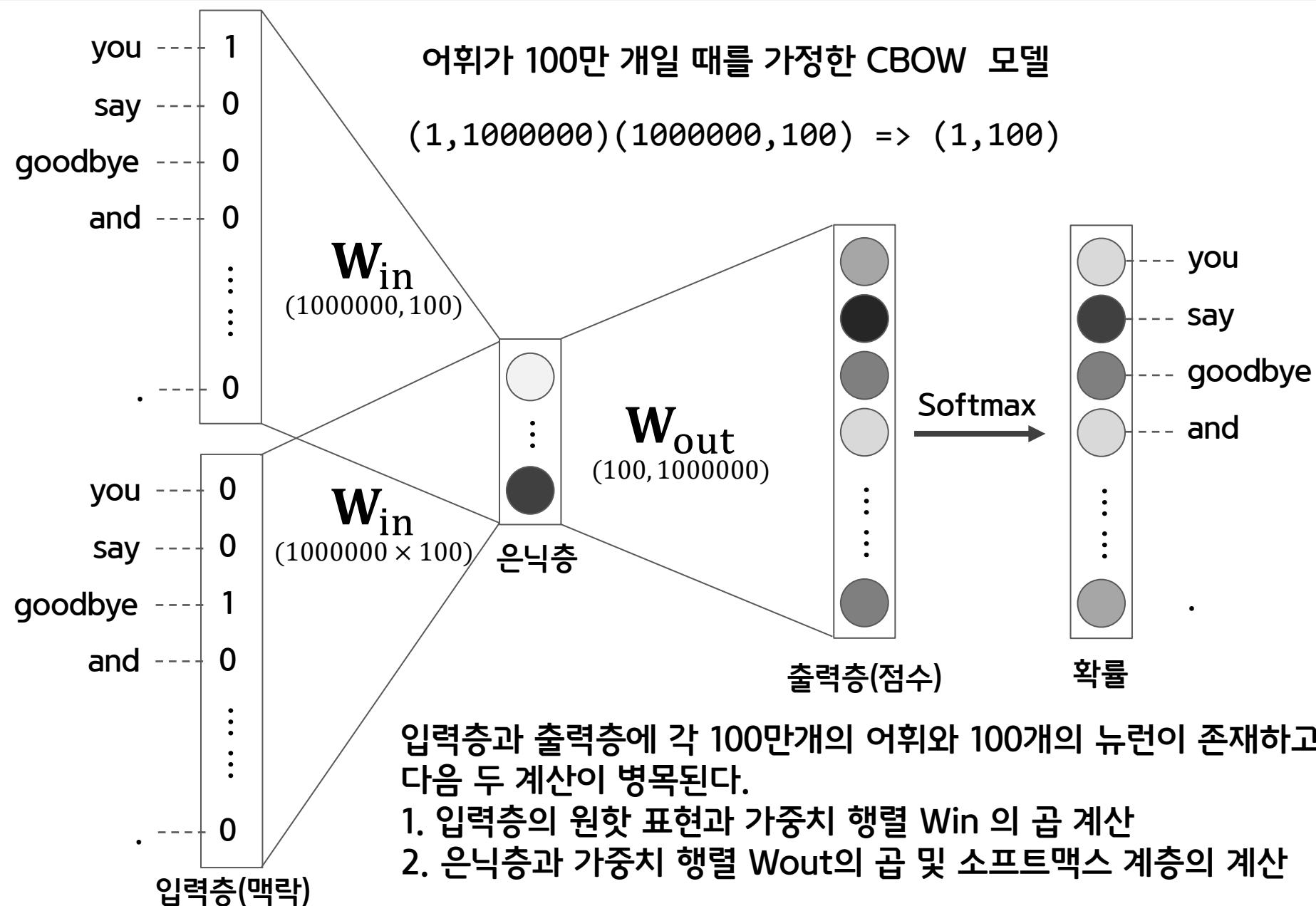
즉, 문자에서 나오는 n개의 단어 열로부터 다음 단어를 예측하는 모델이다.



입력 층 가중치와 행렬 곱으로 은닉층이 계산되고,
다시 출력층 가중치와의 행렬 곱으로 각 단어 점수를 계산,
소프트맥스 함수를 적용해 각 단어의 출현 확률을 얻어 정답 레이블과 비교하여 손실을 구한다.

위의 그림은 다루는 어휘가 7개일 때이다.

만약 어휘가 100만개, 은닉층의 뉴런이 100개인 CBOW 모델을 생각해 보면,



입력층과 출력층에 각 100만개의 어휘와 100개의 뉴런이 존재하고, 다음 두 계산이 병목된다.

1. 입력층의 원핫 표현과 가중치 행렬 W_{in} 의 곱 계산
2. 은닉층과 가중치 행렬 W_{out} 의 곱 및 소프트맥스 계층의 계산

Embedding 이란, 텍스트를 구성하는 하나의 단어를 수치화하는 방법의 일종이다.

텍스트 분석에서 흔히 사용하는 방식은 단어 하나에 인덱스 정수를 할당하는 Bag of Words 방법이다.
이 방법을 사용하면 문서는 단어장에 있는 단어의 갯수와 같은 크기의 벡터가 되고 단어장의
각 단어가 그 문서에 나온 횟수만큼 벡터의 인덱스 위치의 숫자를 증가시킨다.

즉 단어장이 "I", "am", "a", "boy", "girl" 다섯개의 단어로 이루어진 경우 각 단어에 다음과 같이 숫자를 할당한다.

"I": 0

"am": 1

"a": 2

"boy": 3

"girl": 4

이 때 "I am a girl" 이라는 문서는 다음과 같이 벡터로 만들 수 있다.

[1, 1, 1, 0, 1]

단어 임베딩은 하나의 단어를 하나의 인덱스 정수가 아니라 실수 벡터로 나타낸다.

예를 들어 2차원 임베딩을 하는 경우 다음과 같은 숫자 벡터가 될 수 있다.

"I": (0.3, 0.2)

"am": (0.1, 0.8)

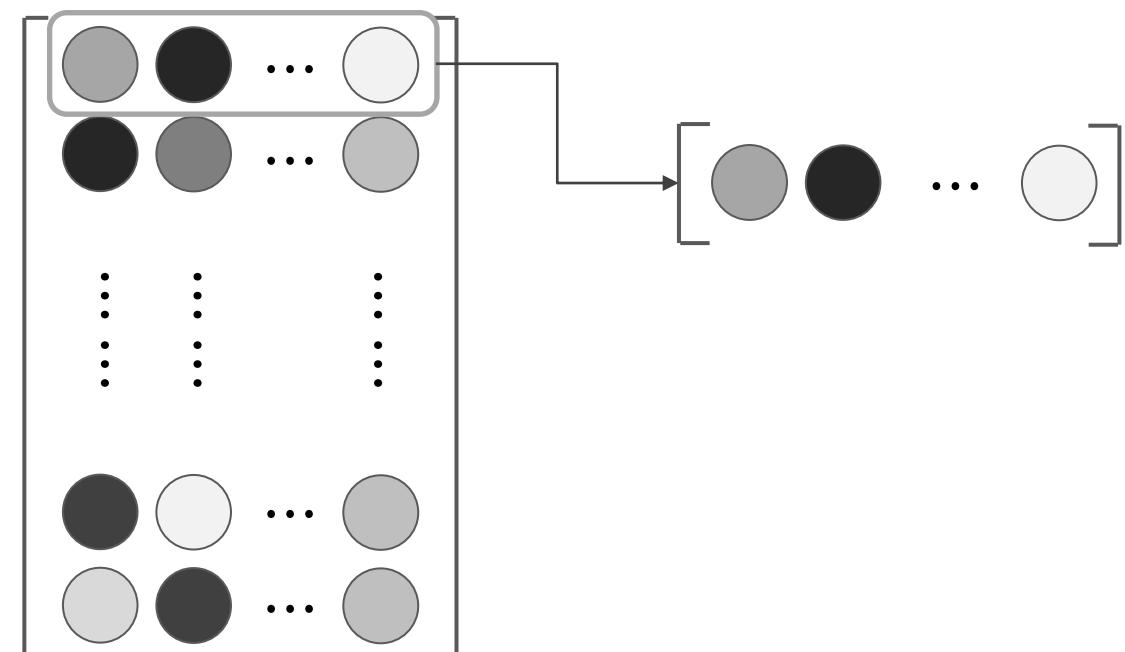
"a": (0.5, 0.6)

"boy": (0.2, 0.9)

"girl": (0.4, 0.7)

맥락(원핫 표현)과 MatMul 계층의 가중치를 곱한다.

$$\begin{bmatrix} 1 & 0 & 0 & \dots & \dots & 0 & 0 \end{bmatrix}$$

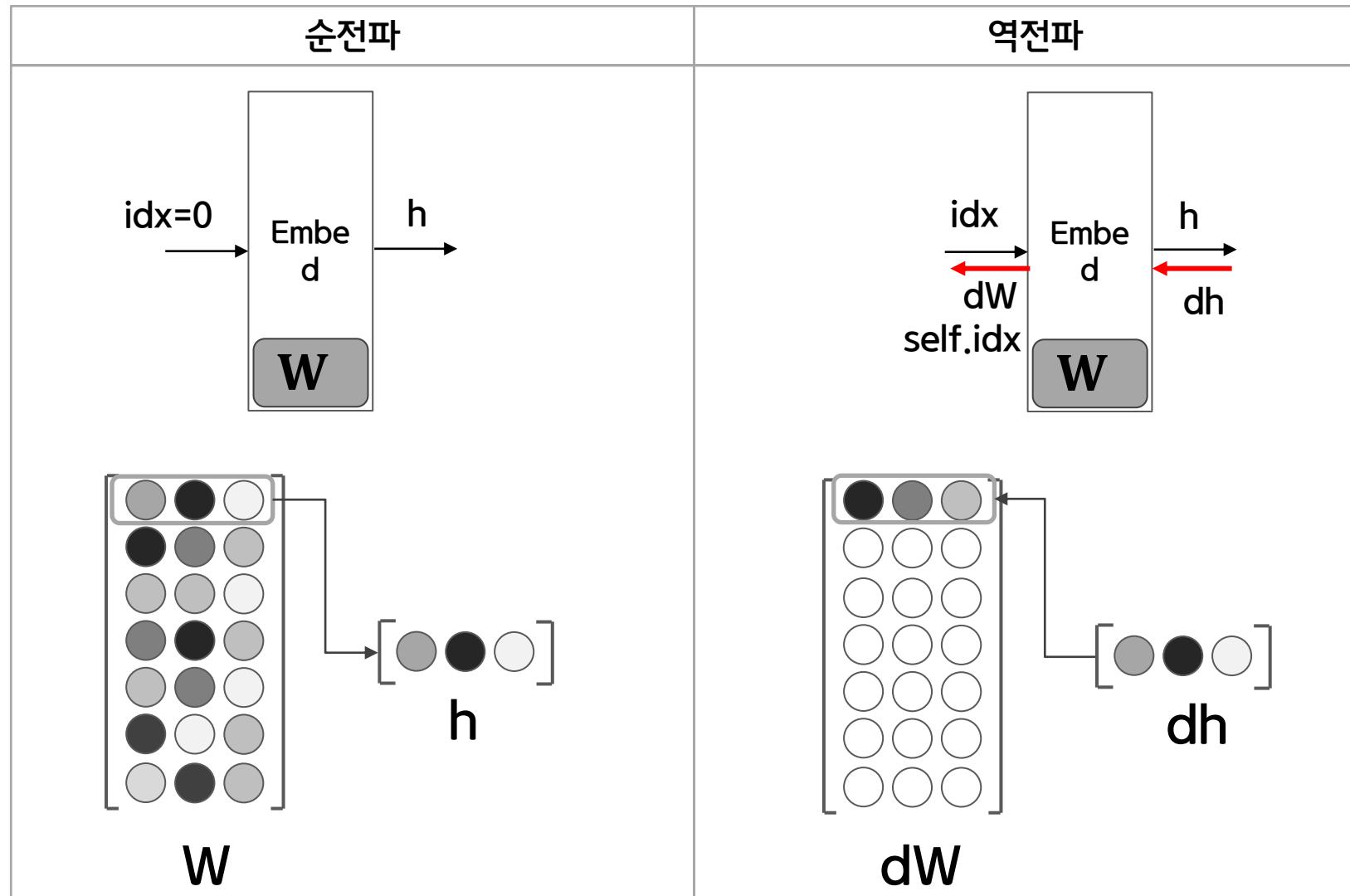


$$c \quad (1, 1000000)$$

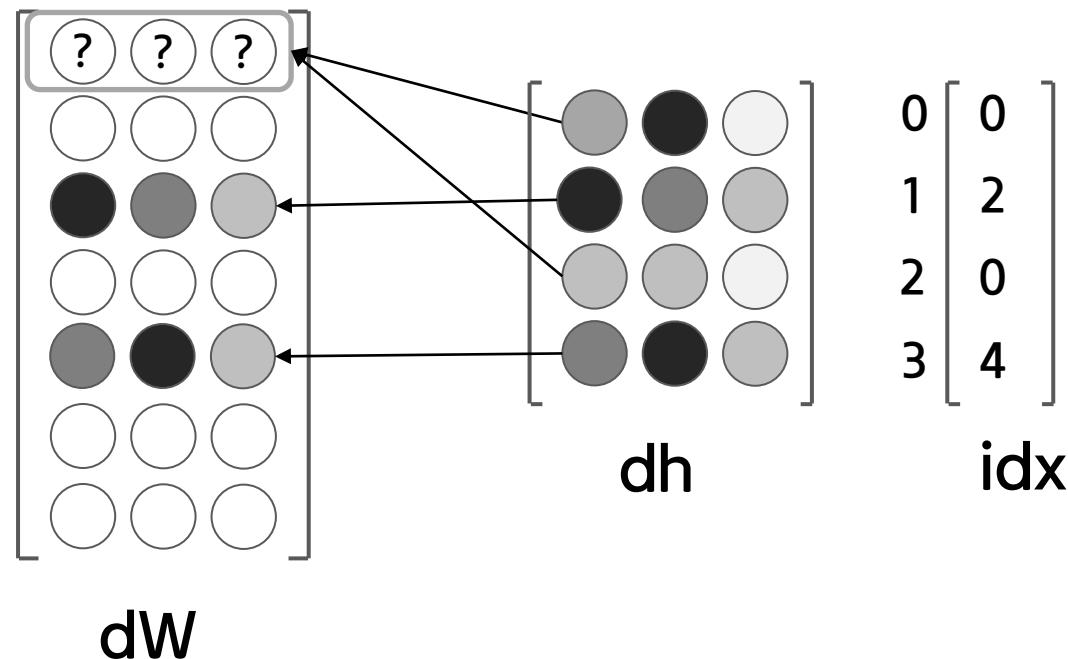
$$W_{in} \quad (1000000, 100)$$

$$= \quad h \quad (1 \times 100)$$

Embedding 계층의 forward와 backward 처리



idx 배열의 원소 중 값(행 번호)이 같은 원소가 있다면, dh 를 해당 행에 할당할 때 문제가 생긴다.



4. word2vec 속도 개선

4.1 word2vec 개선 I

4.2 word2vec 개선 II

4.3 개선판 word2vec 학습

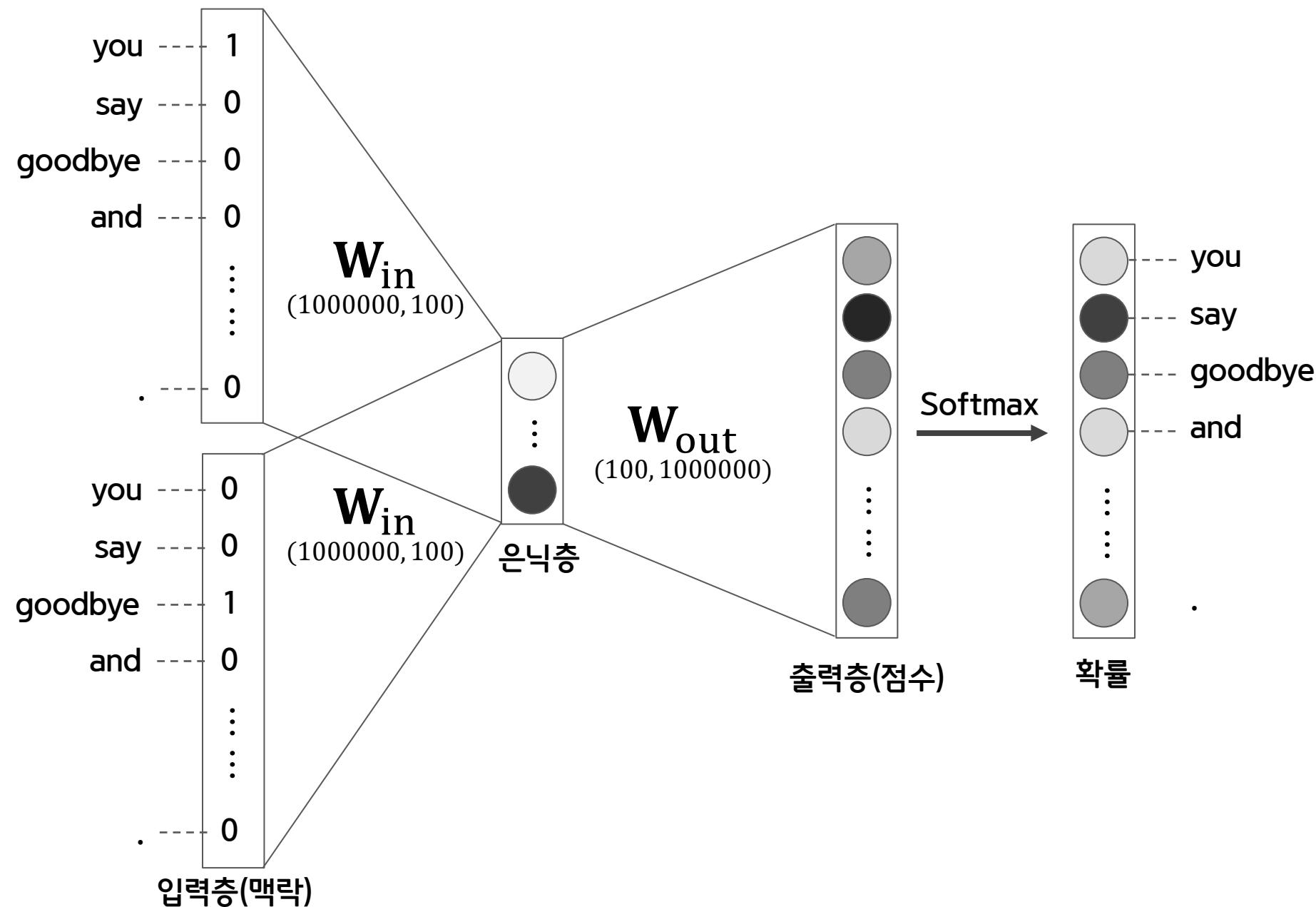
남은 병목은 은닉층 이후의 처리(행렬 곱과 softmax 계층의 계산) 이다.

어휘가 100만개일 때를 가정한 word2vec 모델이 있었는데,
은닉층의 뉴런과 가중치 행렬의 곱을 할때,
크기가 100인 은닉층 뉴런과 100×100 만인 가중치 행렬을 곱해야 하고,
역전파 때도 같은 계산을 수행한다.

또한, 소프트맥스의 계산량도 \exp 계산만 100만번 수행해야 한다. 따라서,

1. 은닉층의 뉴런과 가중치 행렬의 곱
2. 소프트맥스 계층의 계산

을 가볍게 해야한다.



어휘가 많아 지면 Softmax의 계산량이 증가한다.

$$y_k = \frac{\exp(s_k)}{\sum_{i=1}^{1000000} \exp(s_i)}$$

네거티브 샘플링

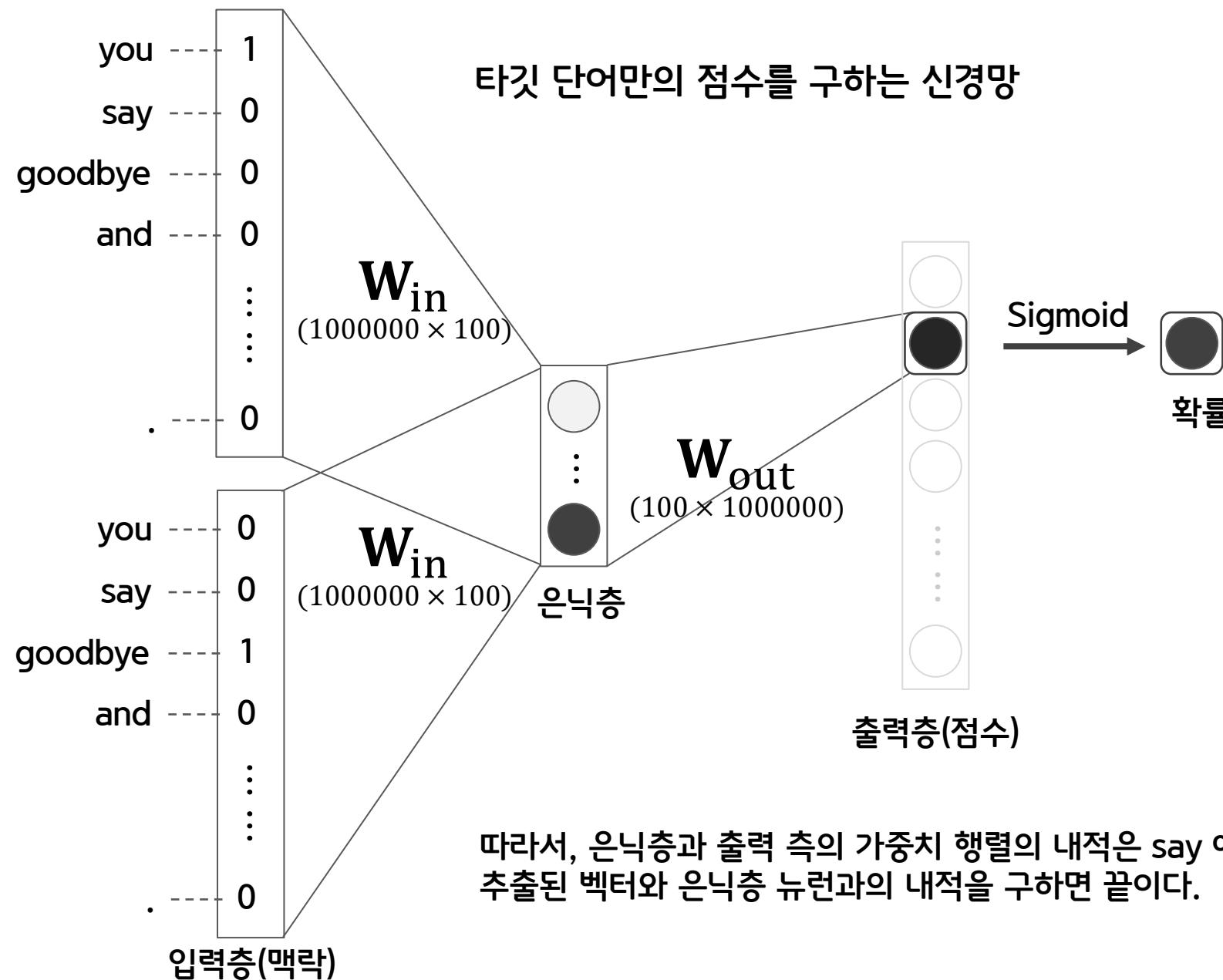
네거티브 샘플링의 핵심은 다중분류를 이진분류로 근사하는 것이다.

예를 들면,

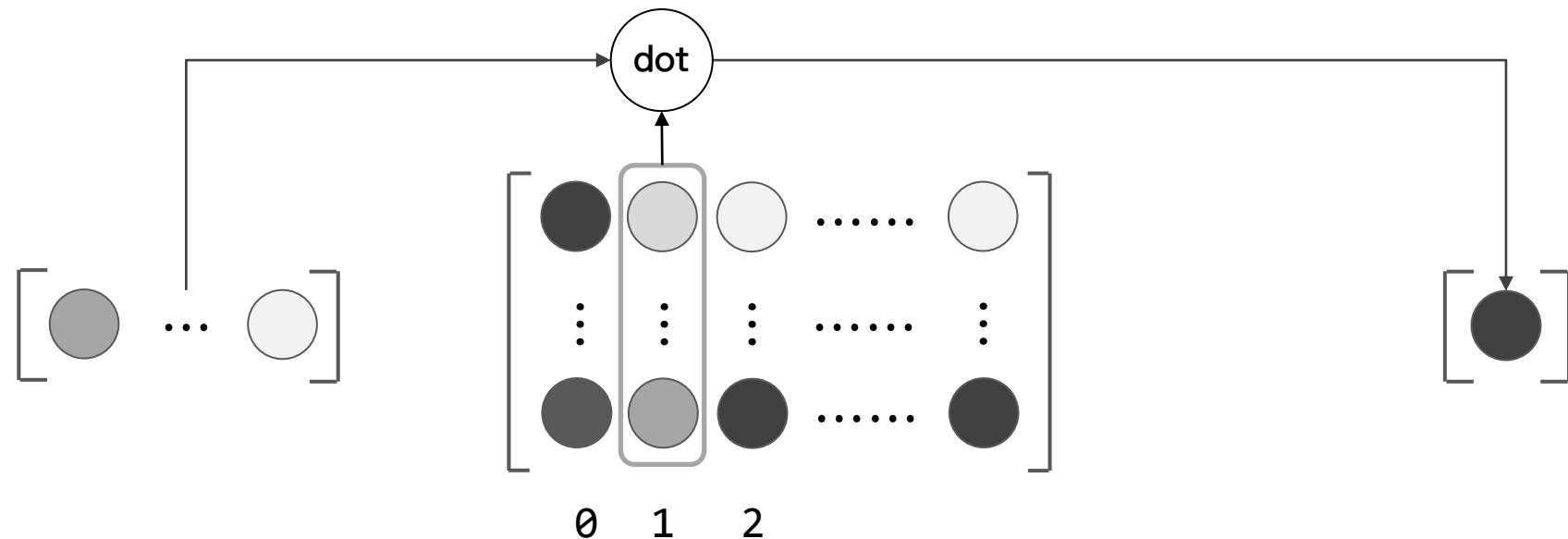
다중 분류는 맥락이 you 와 goodbye 일때, 타깃 단어는 무엇입니까?
에 대답하는 것이고,

이진 분류는 맥락이 you 와 goodbye 일때, 타깃 단어는 say 입니까?
에 대답하는 것이다.

이런식으로 하면 출력층에 뉴런을 하나만 준비하면 된다.
출력층의 이 뉴런이 say 의 점수를 출력하는 것이다.



"say"에 해당하는 열벡터와 은닉층 뉴런의 내적을 계산한다.



형상 :

h
(1, 100)

W_{out}
(100, 1000000)

S
(1, 1)

$$(1, 100) \times (100, 1000000) \Rightarrow (1, 1000000)$$

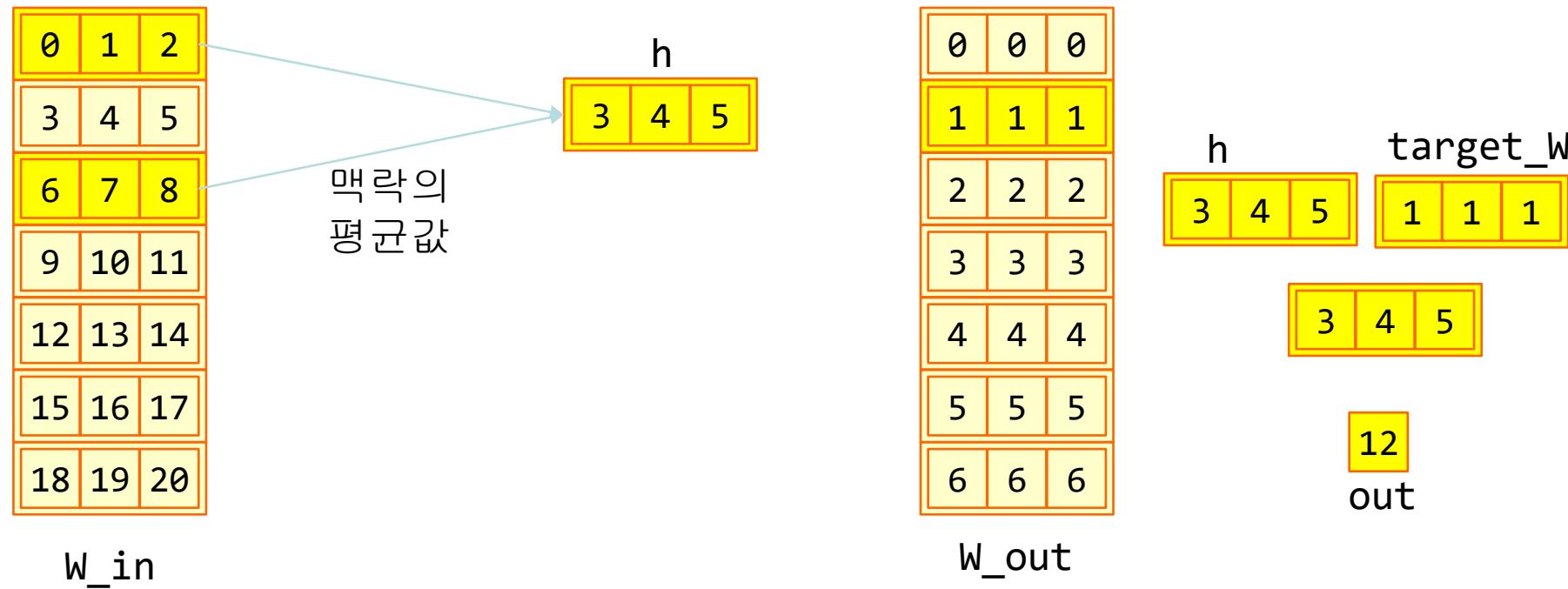
$$(1, 100) \times (100, 1) \Rightarrow (1, 1)$$

```

def forward(self, h, idx):
    target_w = self.embed.forward(idx)
    print(target_w)
    out = np.sum(target_w * h, axis=1)

    self.cache = (h, target_w)
    return out

```

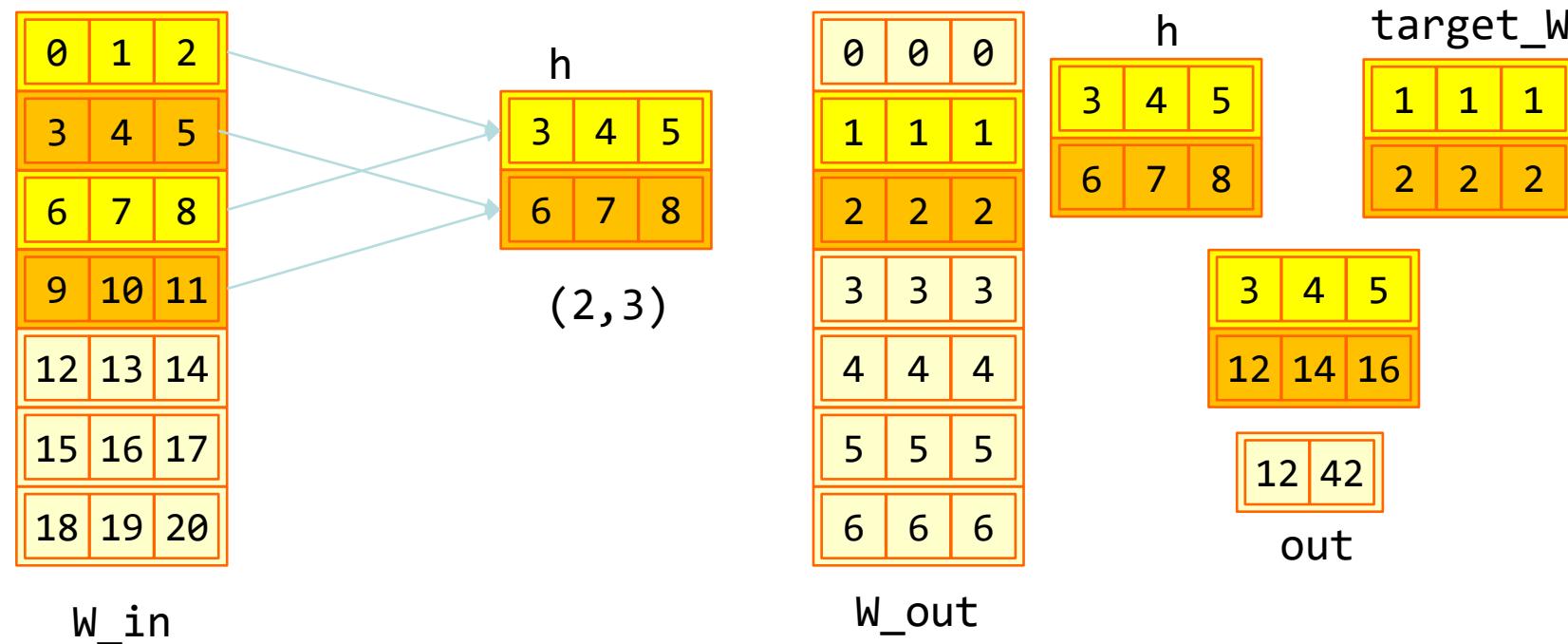


```

def forward(self, h, idx):
    target_w = self.embed.forward(idx)
    print(target_w)
    out = np.sum(target_w * h, axis=1)

    self.cache = (h, target_w)
    return out

```



```

def backward(self, dout):
    h, target_w = self.cache
    dout = dout.reshape(dout.shape[0], 1)

    dtarget_w = dout * h
    self.embed.backward(dtarget_w)
    dh = dout * target_w
    return dh

```

0	1	2
3	4	5
6	7	8
9	10	11
12	13	14
15	16	17
18	19	20

w_in

맥락의 평균값

3	4	5
3	3	3

h

dh

0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
0	0	0
0	0	0
0	0	0

self.idx=1

w_out

0	0	0
9	12	15
0	0	0
0	0	0
0	0	0

dw_out

9	12	15
---	----	----

$$h \quad target_w \\ 3 \quad 4 \quad 5 \quad * \quad 1 \quad 1 \quad 1$$

3	3	3
---	---	---

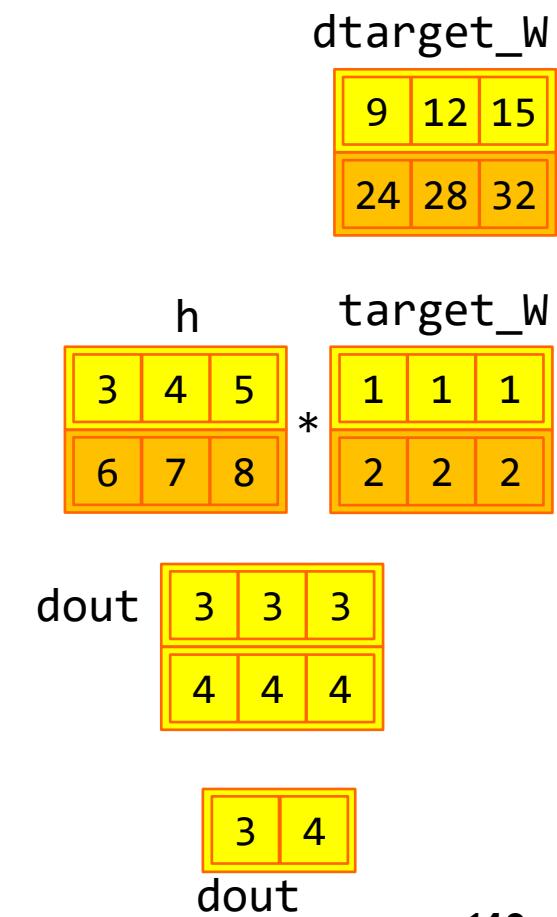
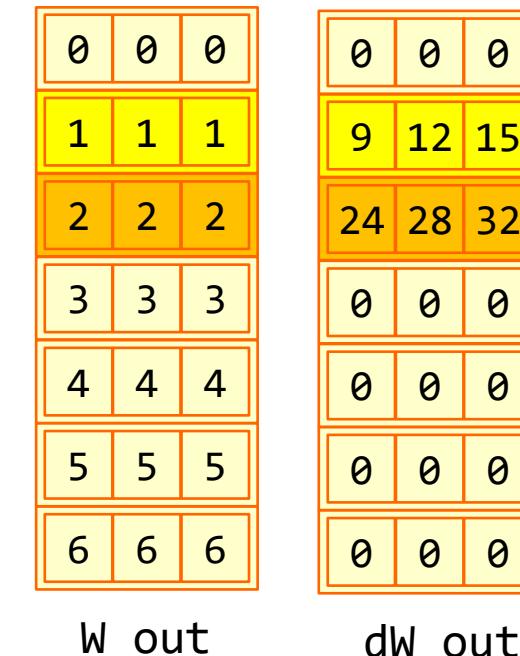
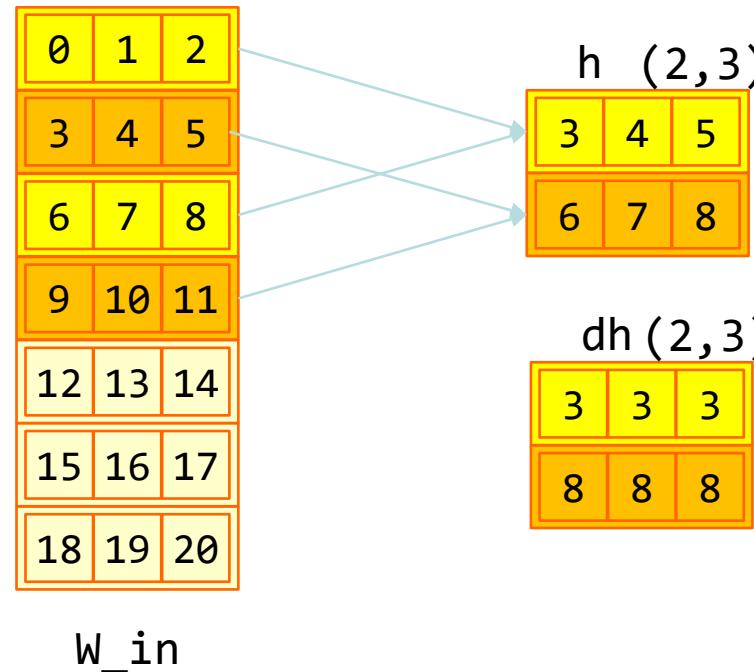
dout

3

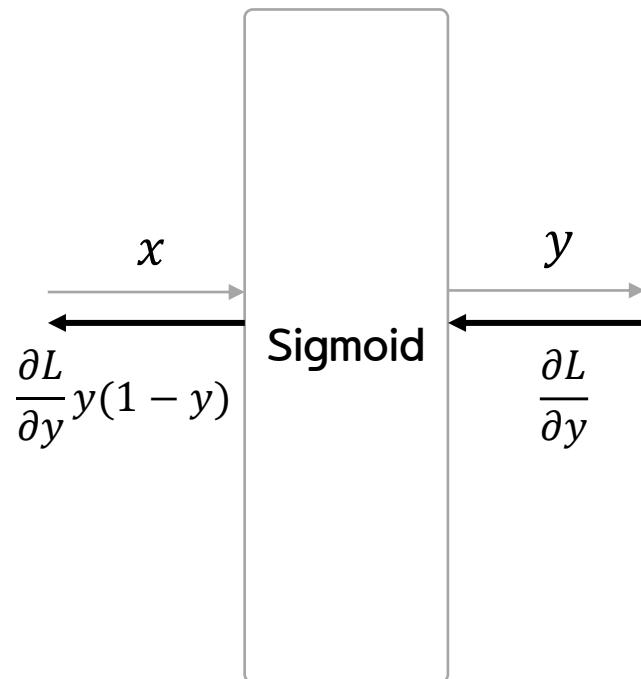
dout

```
def backward(self, dout):
    h, target_w = self.cache
    dout = dout.reshape(dout.shape[0], 1)
```

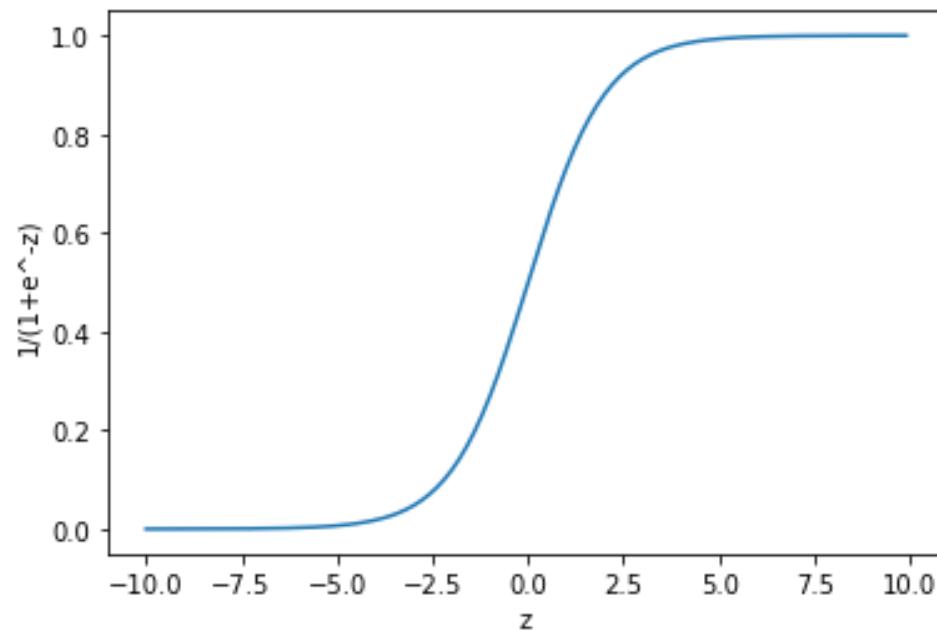
```
dtarget_w = dout * h
self.embed.backward(dttarget_w)
dh = dout * target_w
return dh
```



Sigmoid 계층과 시그모이드 함수의 그래프

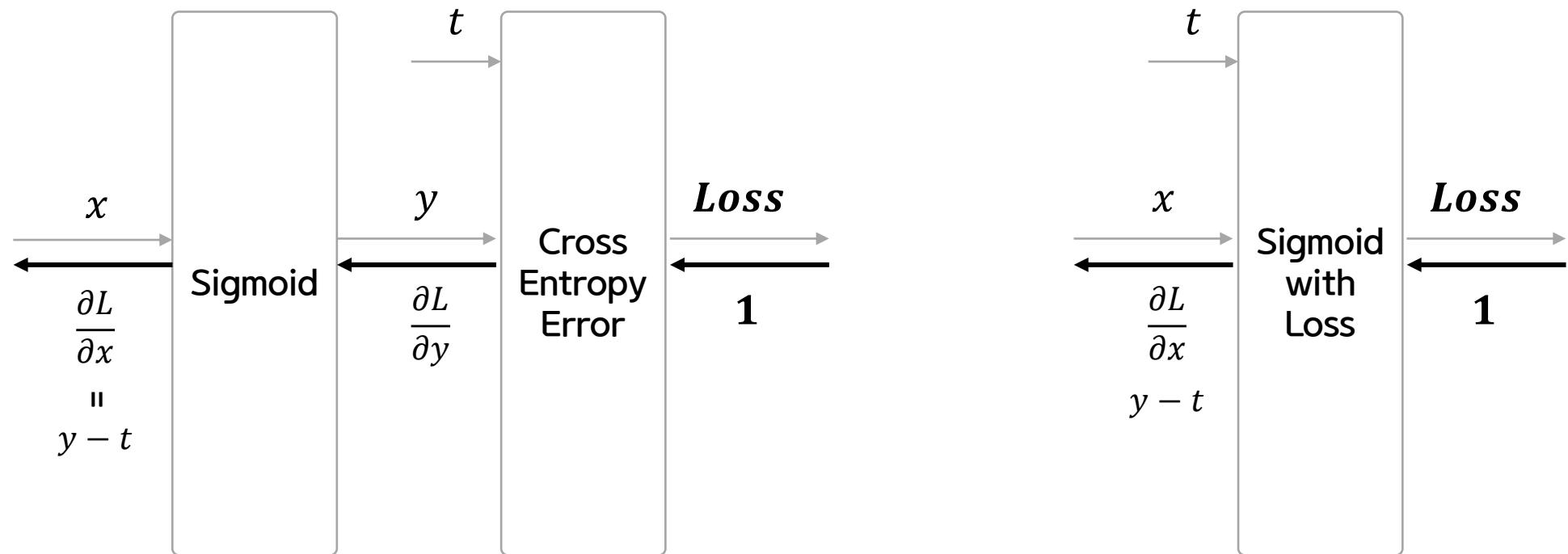


$$y = \frac{1}{1 + e^{-x}}$$

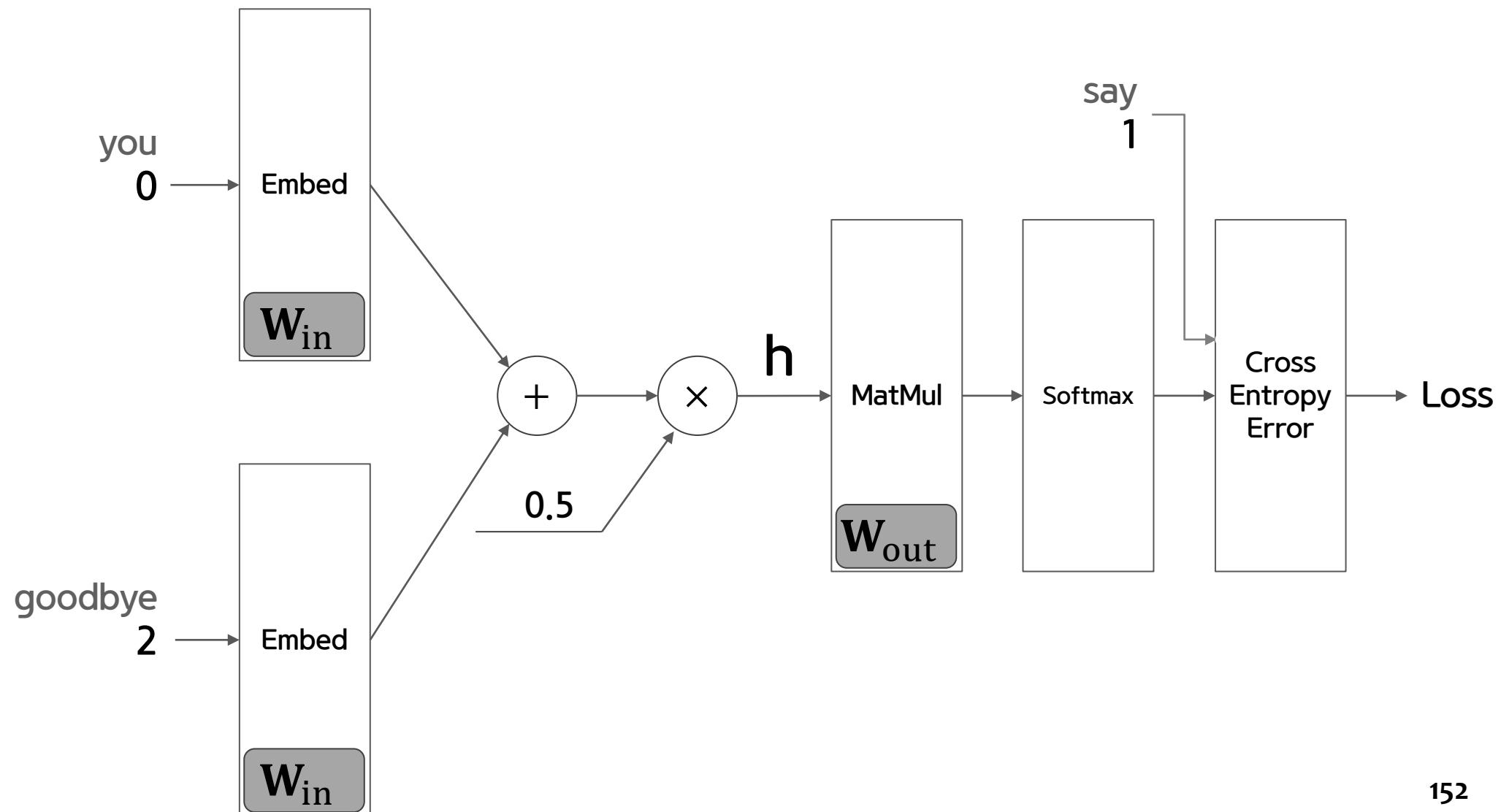


Sigmoid 계층과 Cross Entropy Error 계층의 계산 그래프

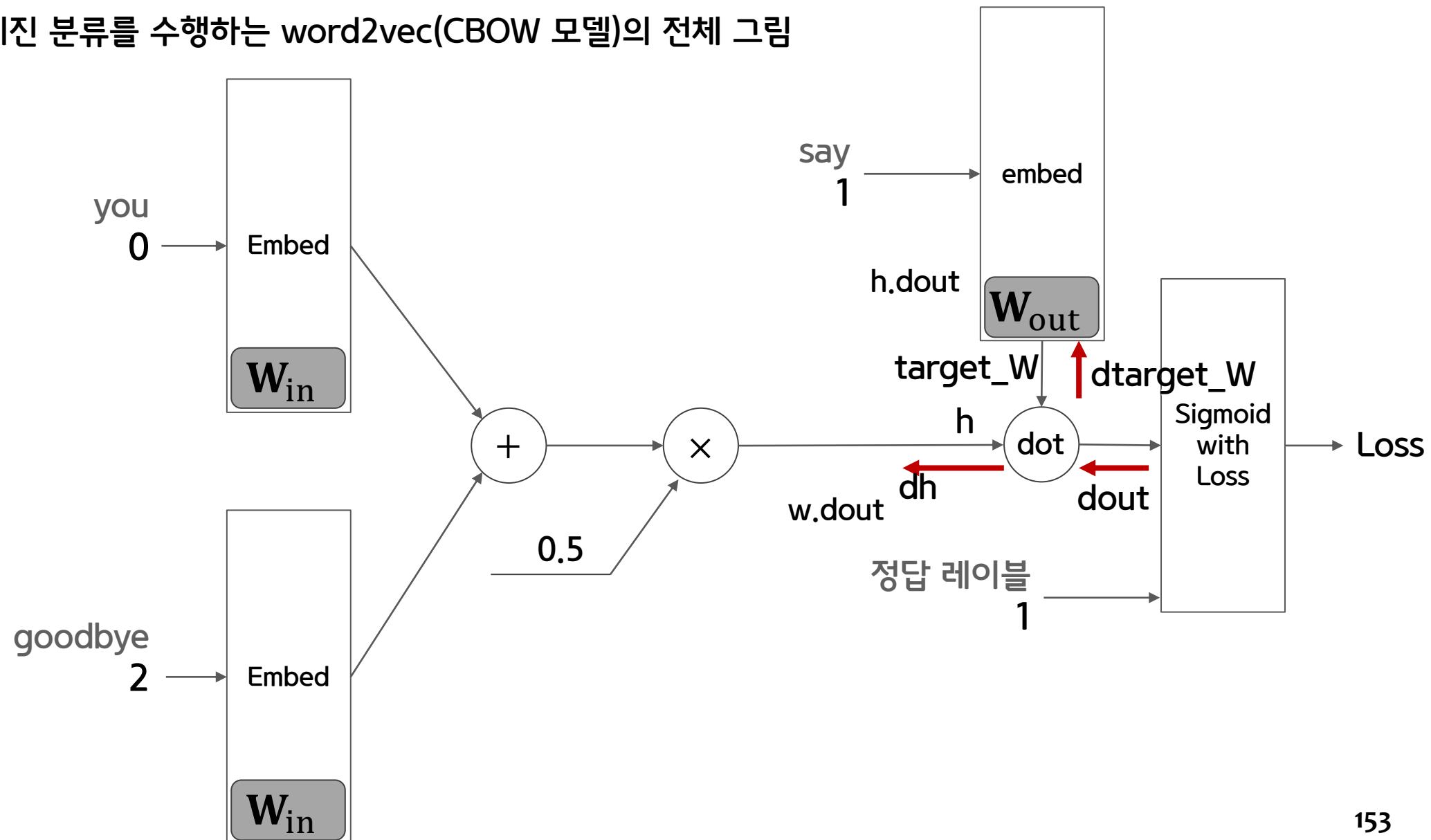
$$L = -(t \log y + (1 - t) \log(1 - y))$$



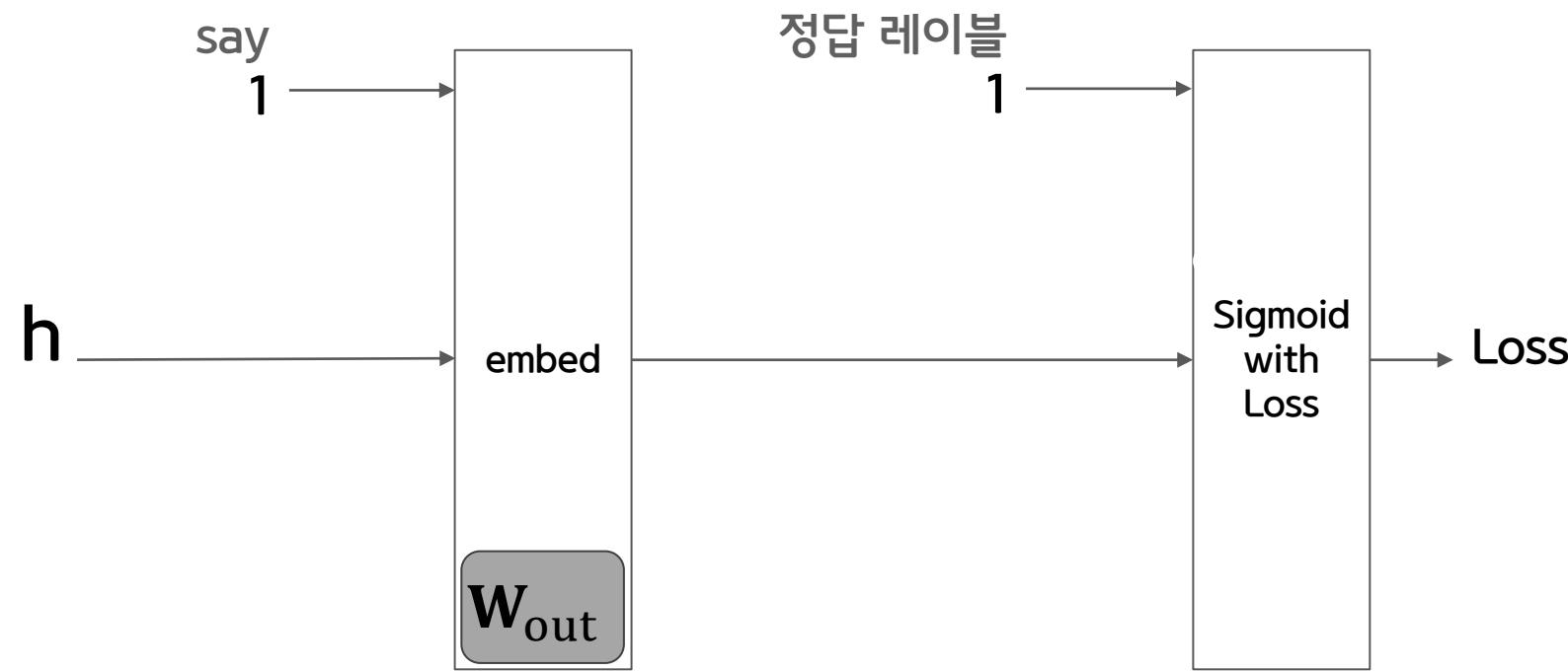
다중 분류를 수행하는 CBOW 모델의 전체 그림



이진 분류를 수행하는 word2vec(CBOW 모델)의 전체 그림



은닉층 이후 처리(Embedding Dot 계층을 사용하여 Embedding 계층과 내적 계산을 한 번에 수행)



Embedding Dot 계층의 각 변수의 구체적인 값

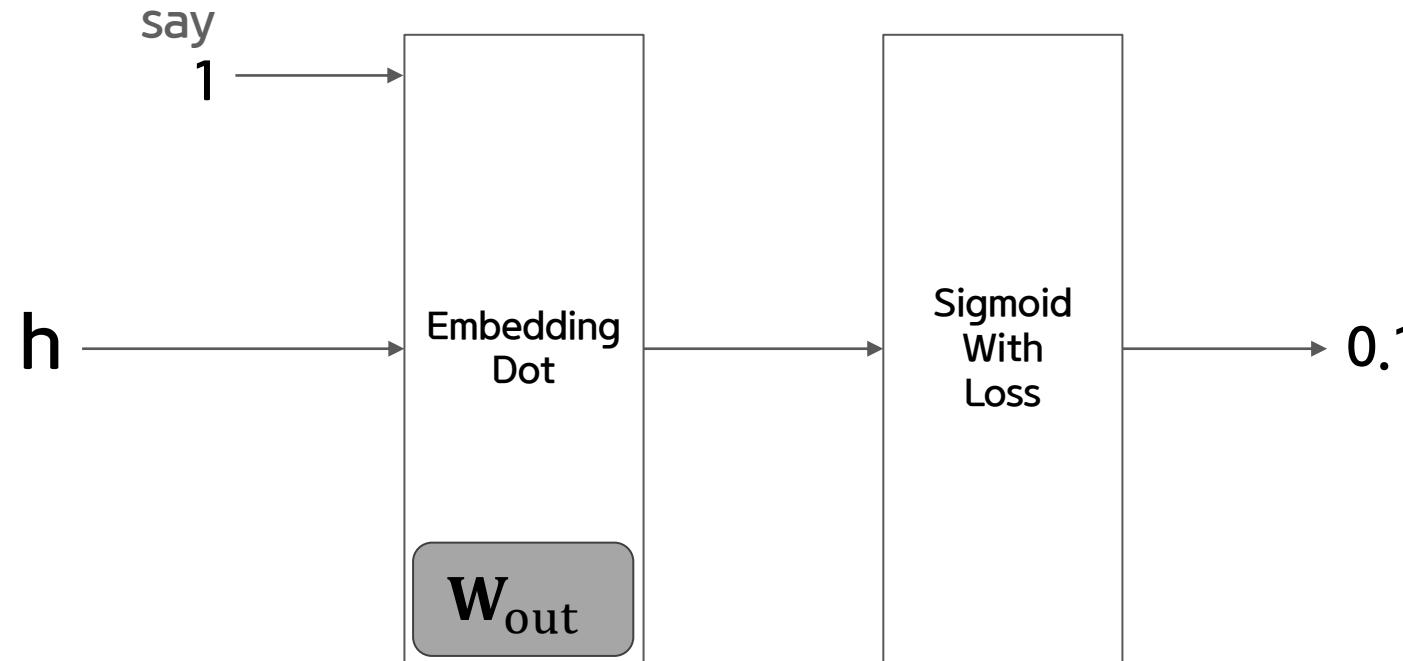
```

embed = Embedding(W)
target_W = embed.forward(idx)
out = np.sum(target_W*h, axis=1)

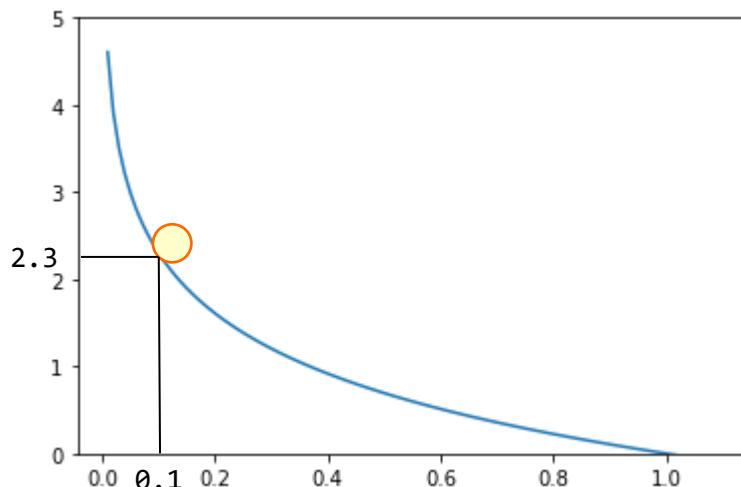
```

W	idx	target_W	h	target_W * h	out
[[0 1 2]	[[0 3 1]]	[[0 1 2]	[[0 1 2]	[[0 1 4]	[[5 122 86]]
[3 4 5]		[9 10 11]	[3 4 5]	[27 40 55]	
[6 7 8]		[3 4 5]]	[6 7 8]]	[18 28 40]]	
[9 10 11]					
[12 13 14]					
[15 16 17]					
[18 19 20]]					

CBOW 모델의 은닉층 이후의 처리 예: 맥락은 "you"와 "goodbye"이고, 타깃이 "say"일 확률은 0.1(10%)이다.

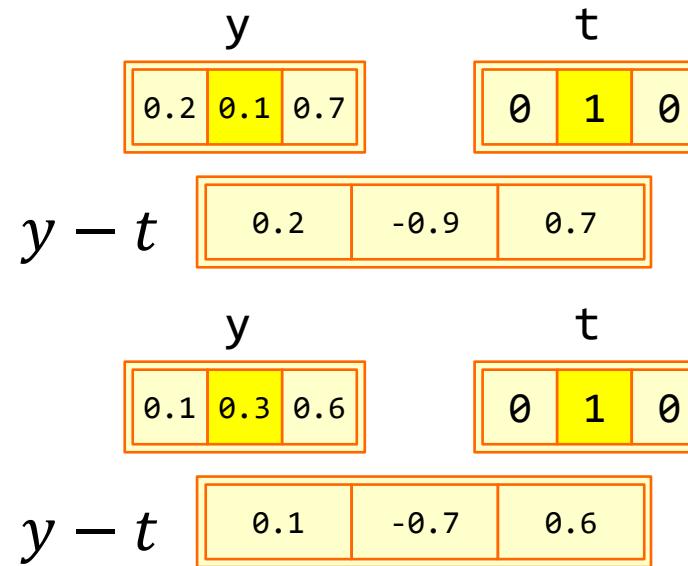
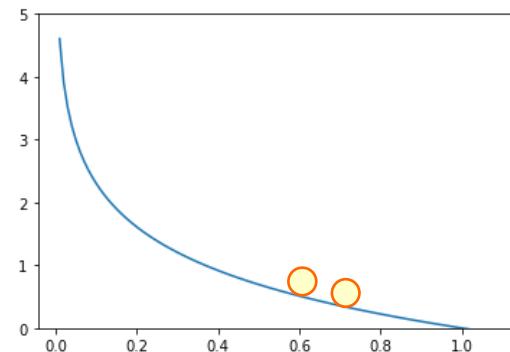
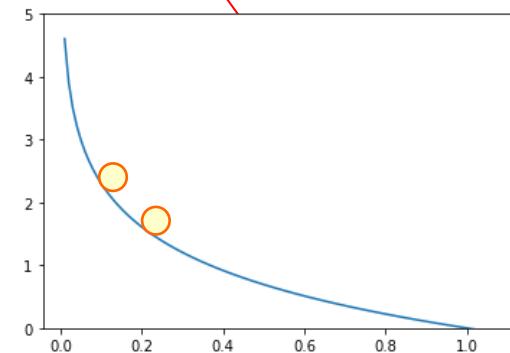
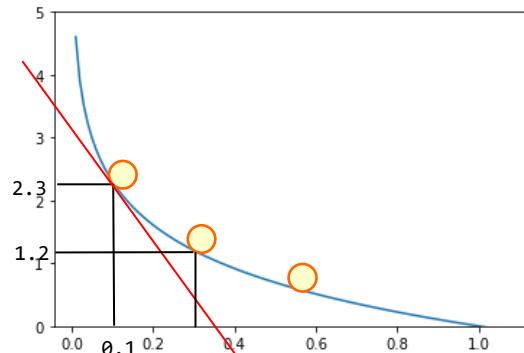


CBOW 모델의 은닉층 이후의 처리 예: 맥락은 "you"와 "goodbye"이고, 타깃이 "say"일 확률은 0.1(10%)이다.

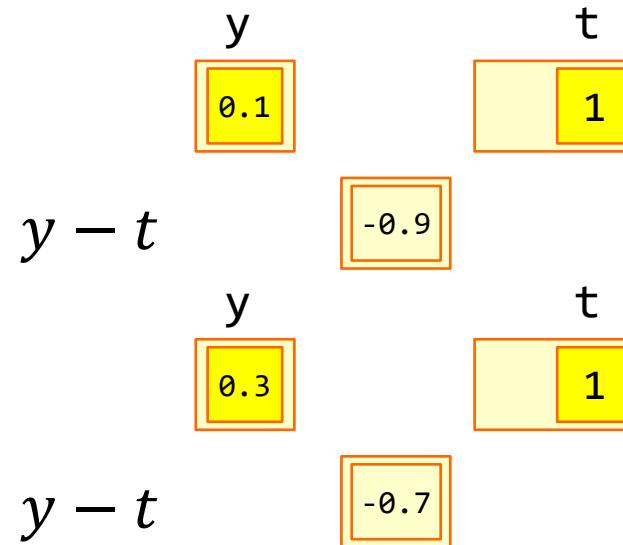
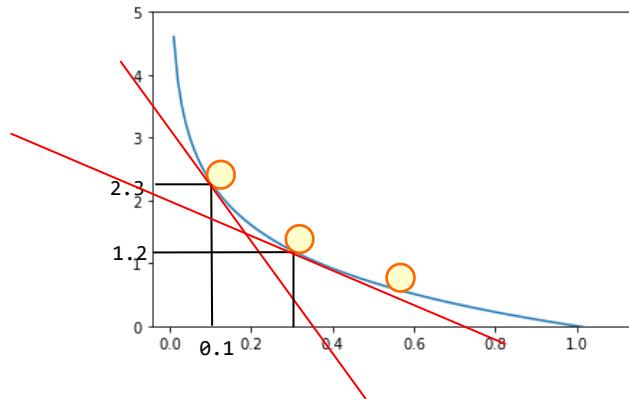


$$\begin{aligned}
 & - \sum_{c=1}^3 t_c \log(y_c) \\
 & - (t_1 \log(y_1) + t_2 \log(y_2) + t_3 \log(y_3)) \\
 & - (0 * \log(y_1) + 1 * \log(0.1) + 0 * \log(y_3))
 \end{aligned}$$

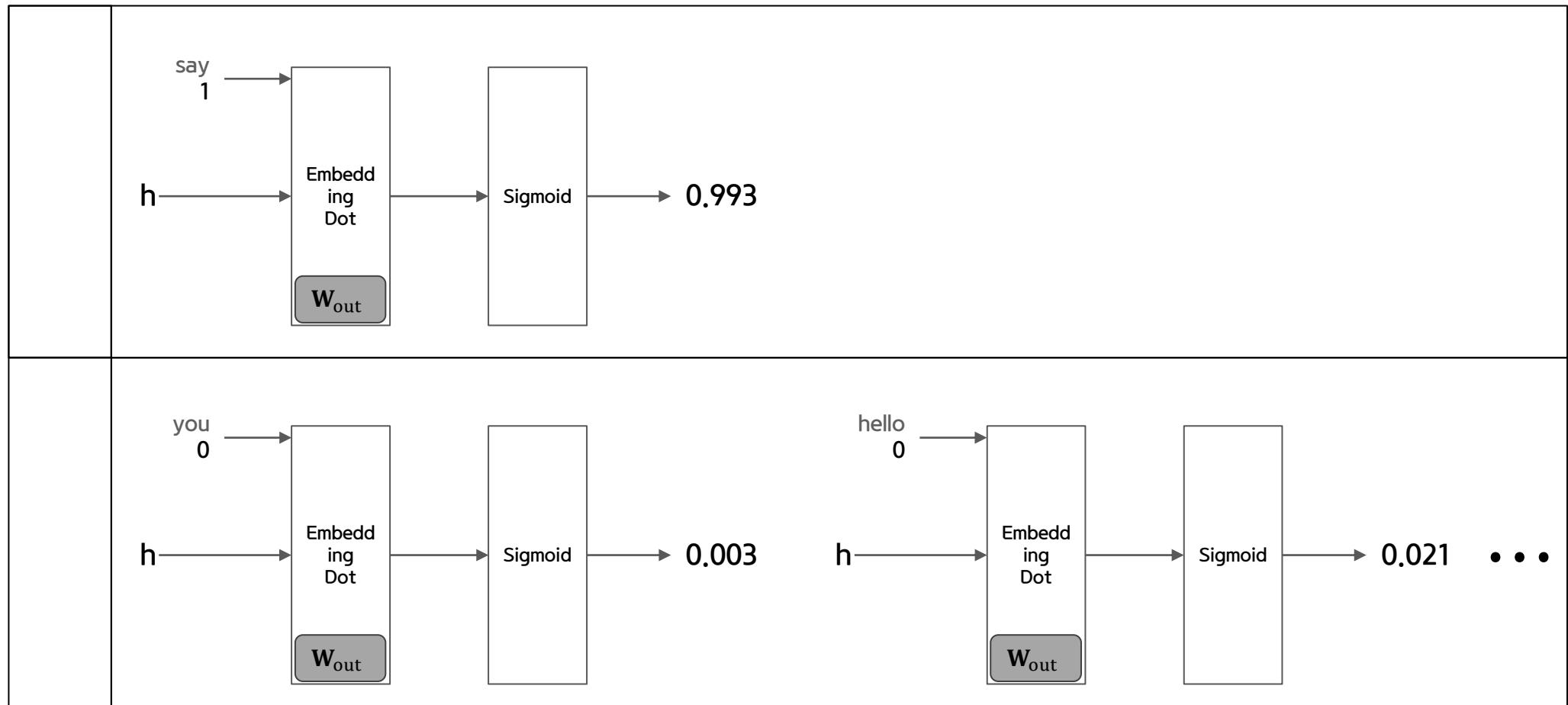
CBOW 모델의 은닉층 이후의 처리 예: 맥락은 "you"와 "goodbye"이고, 타깃이 "say"일 확률은 0.1(10%)이다.



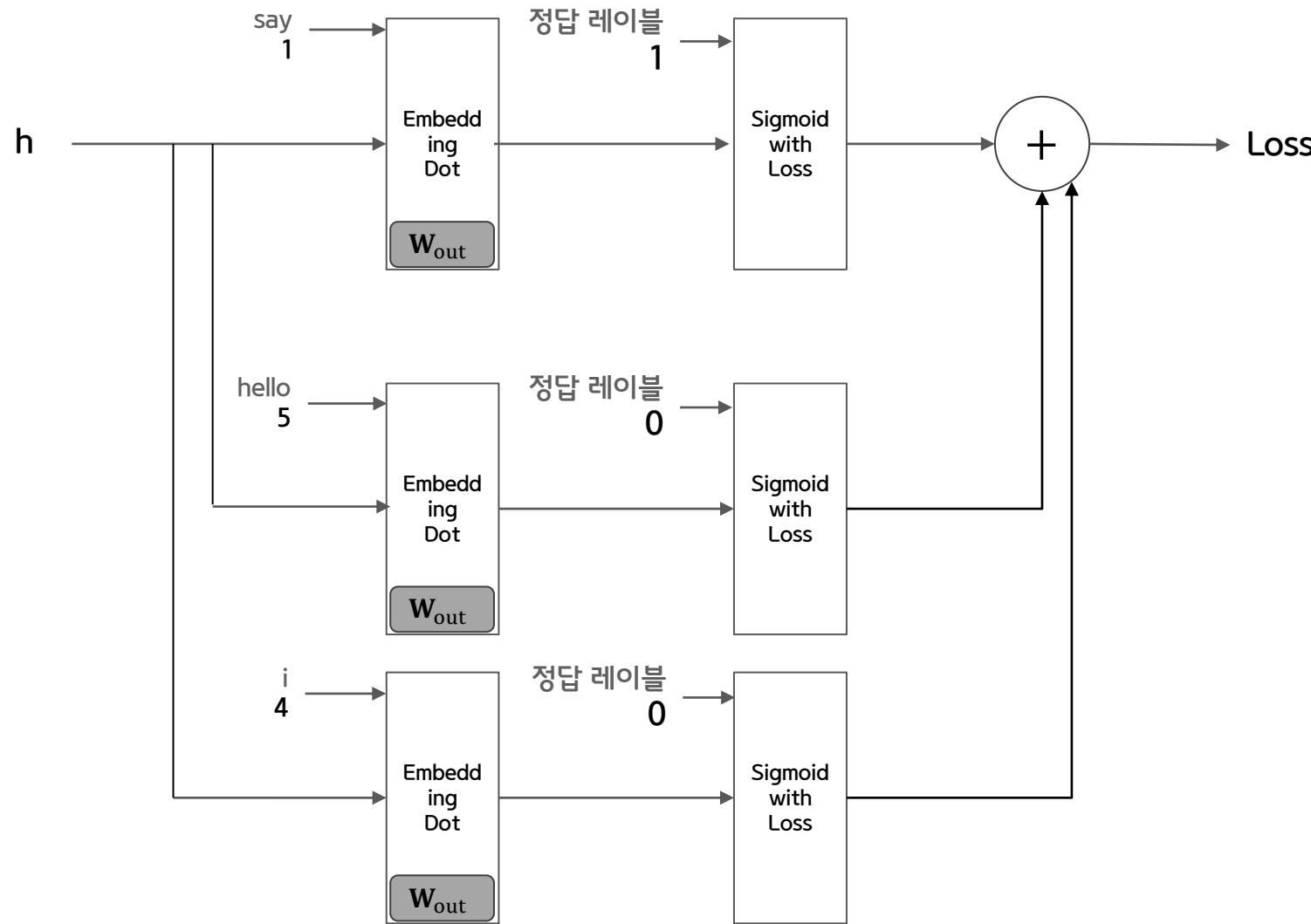
CBOW 모델의 은닉층 이후의 처리 예: 맥락은 "you"와 "goodbye"이고, 타깃이 "say"일 확률은 0.1(10%)이다.



긍정적 예(정답)를 "say"라고 가정하면, "say"를 입력했을 때의 Sigmoid 계층 출력은 1에 가깝고, "say" 이외의 단어를 입력했을 때의 출력은 0에 가까워야 한다. 이런 결과를 내어주는 가중치가 필요하다.



네거티브 샘플링의 예



```

correct_label = np.ones(batch_size, dtype=np.int32)
loss = self.loss_layers[0].forward(score, correct_label)

self.y = 1 / (1 + np.exp(-x))
self.loss = cross_entropy_error(np.c_[1 - self.y, self.y], self.t)

def cross_entropy_error(y, t):
    if y.ndim == 1:
        t = t.reshape(1, t.size)
        y = y.reshape(1, y.size)

    # 정답 데이터가 원핫 벡터일 경우 정답 레이블 인덱스로 변환
    if t.size == y.size:
        t = t.argmax(axis=1)

    batch_size = y.shape[0]

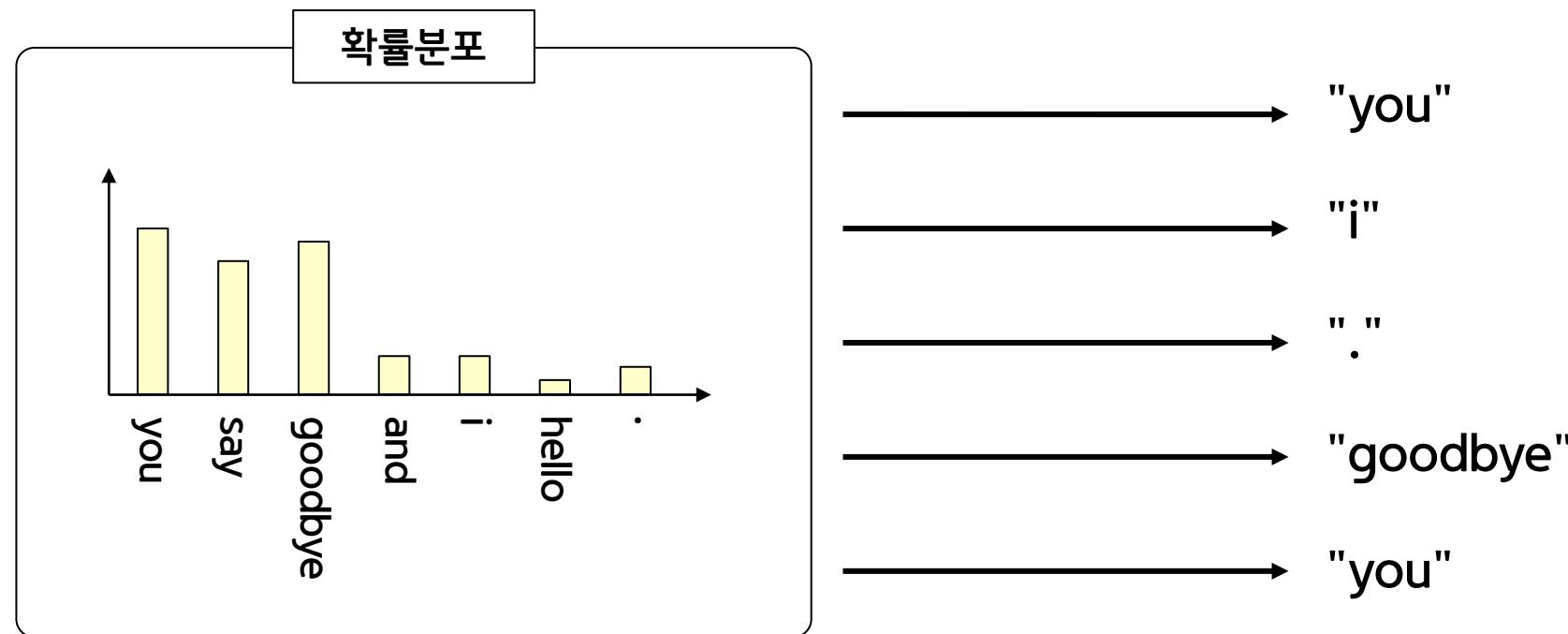
    return -np.sum(np.log(y[np.arange(batch_size), t] + 1e-7)) / batch_size

```

1-y	y	correct_label
0.1	0.9	1

$$-(t \log(y) + (1 - t) \log(1 - y))$$

확률분포에 따라 샘플링을 여러 번 수행한다.



$$P'(w_i) = \frac{P(w_i)^{0.75}}{\sum_j^n P(w_j)^{0.75}}$$

```
p = [0.7, 0.29, 0.01]
new_p = np.power(p, 0.75)
new_p /= np.sum(new_p)
print(new_p)
[0.64196878 0.33150408 0.02652714]
```

0.7	0.29	0.01
-----	------	------

0.76	0.39	0.03
------	------	------

0.64	0.33	0.02
------	------	------

※ 코드 참조

4. word2vec 속도 개선

4.1 word2vec 개선 I

4.2 word2vec 개선 II

4.3 개선판 word2vec 학습

임베딩 계층과 네거티브 샘플링 기법을 사용하여 개선된 신경망 모델에 PTB 데이터셋을 사용해 학습시키고, 실용적인 단어의 분산 표현을 얻어보겠다.

다음은 개선된 CBOW 모델 코드다.

앞의 단순한 CBOW 모델에서 임베딩 계층과 네거티브 샘플링 손실함수 계층을 적용했다.

※ 코드 참조

맥락과 타깃을 단어 ID로 나타낸 예(맥락의 원도우 크기는 1)

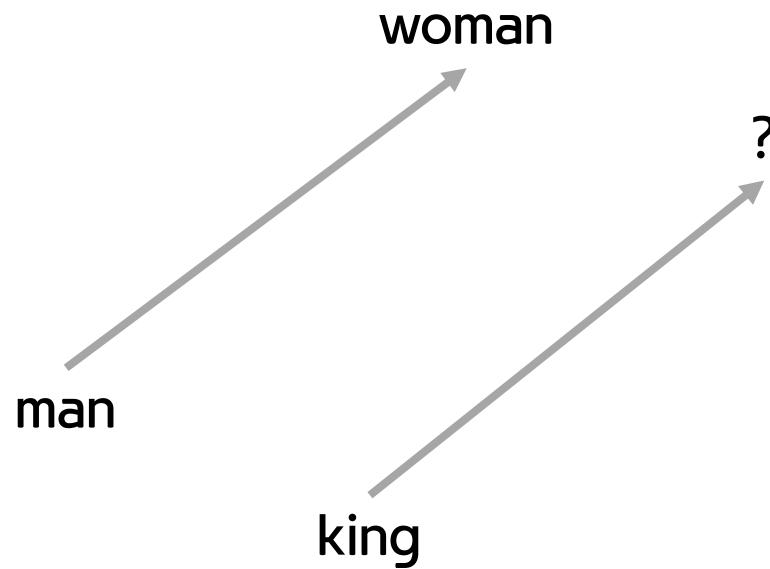
맥락(contexts)	타깃	맥락(contexts)	타깃
you, goodbye	say	[[0 2]]	[[1
say, and	goodbye	[1 3]	2
goodbye, i	and	[2 4]	3
and, say	i	[3 1]	4
i, hello	say	[4 5]	1
say, .	hello	[1 6]]	5]

단어 ID →

※ 코드 참조

※ 코드 참조

"man : woman = king : ?" 유추 문제 풀기(단어 벡터 공간에서 각 단어의 관계성)



5. 순환 신경망(RNN)

5.1 확률과 언어 모델

5.2 RNN 이란

5.3 RNN 구현

5.4 시계열 데이터 처리 계층 구현

5.5 RNNLM 학습과 평가

피드포워드(feed forward) 신경망

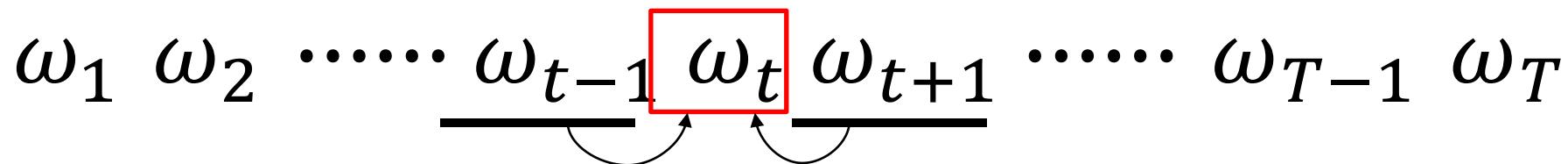
- 흐름이 단방향
- 시계열 데이터의 성질(패턴)을 충분히 학습할 수 없음

순환 신경망(Recurrent Neural Network, RNN)의 등장

CBOW(Continuous bag-of-words)모델

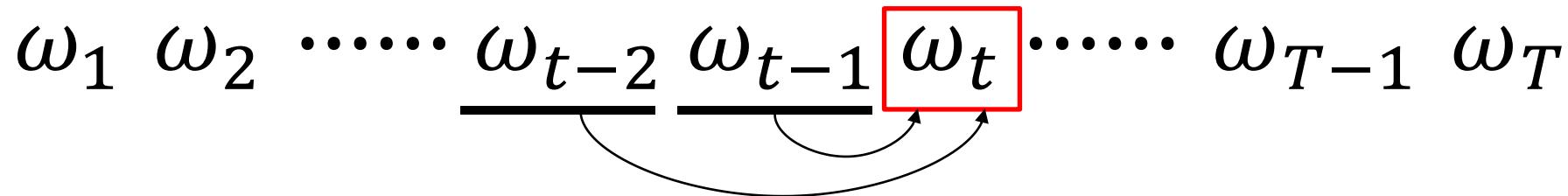
- CBOW 모델의 학습 -> 손실함수(말뭉치 전체의 손실함수의 총합)를 최소화하는 가중치 매개변수를 찾는다. -> 맥락으로부터 타깃을 더 정확하게 추측 가능
- 맥락 안의 단어 순서가 무시된다는 한계가 있음
말뭉치 : $w_1, w_2, w_3, \dots, w_t$

word2vec의 CBOW 모델 : 맥락의 단어로부터 타깃 단어를 추측한다.



$$P(w_t | w_{t-1}, w_{t+1})$$

왼쪽 원도우만 맥락으로 고려한다.



$w(t-2)$ 과 $w(t-1)$ 이 주어졌을 때 타깃이 w_t 가 될 확률(CBOW 모델이 출력할 확률)을 수식으로 표현하면

$$P(w_t | w_{t-2}, w_{t-1})$$

CBOW모델이 다루는 손실함수

$$L = -\log P(w_t | w_{t-2}, w_{t-1})$$

단어 나열에 확률을 부여

특정한 단어의 시퀀스에 대해서 그 시퀀스가 일어날 가능성이 어느 정도인지(얼마나 자연스러운 단어 순서인지)를 확률로 평가한다.

- 언어 모델의 사용
기계 번역과 음성 인식
새로운 문장을 생성

w_1, \dots, w_m 이라는 m 개 단어로 된 문장이 있을 때

w_1, \dots, w_m 순서로 출현할 확률 $P(w_1, \dots, w_m)$
(여러 사건이 동시에 일어날 확률이므로 동시확률이라고 한다.)

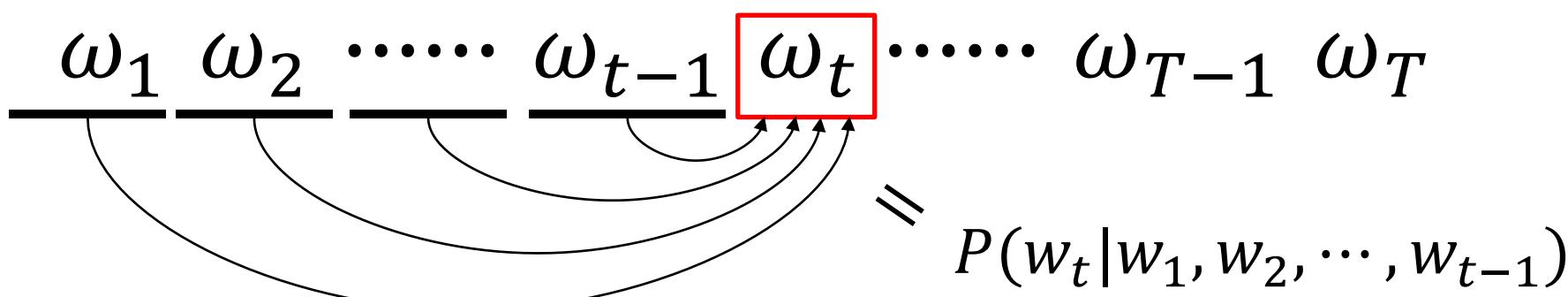
$$\begin{aligned} P(w_1, \dots, w_m) &= P(w_m | w_1, \dots, w_{m-1})P(w_{m-1} | w_1, \dots, w_{m-2}) \\ &\quad \cdots P(w_3 | w_1, w_2)P(w_2 | w_1)P(w_1) \\ &= \prod_{t=1}^m P(w_t | w_1, \dots, w_{t-1}) \end{aligned}$$

$$P(A, B) = P(A|B)P(B)$$

A, B가 모두 일어날 확률 $P(A, B)$ 은 B가 일어날 확률 $P(B)$ 와 B가 일어난 후 A가 일어날 확률 $P(A|B)$ 를 곱한 값과 같다.

$$\underbrace{P(w_1, \dots, w_{m-1}, w_m)}_A = P(A, w_m) = P(w_m|A)P(A)$$

$$P(A) = P(\underbrace{w_1, \dots, w_{m-2}}_{A'}, w_{m-1}) = P(A', w_{m-1}) = P(w_{m-1}|A'')P(A'')$$



- word2vec의 CBOW모델을 언어 모델에 적용하려면 맥락의 크기를 특정 값으로 한정하여 근사적으로 나타낼 수 있다.
- 맥락의 크기는 임의 길이로 설정할 수 있지만 결국 특정 길이로 '고정'된다.
 - 예를 들어 왼쪽 10개의 단어를 맥락으로 CBOW 모델을 만든다고 하면 그 맥락보다 더 왼쪽에 있는 단어의 정보는 무시된다.
- CBOW모델의 맥락 크기를 키울 수는 있으나 맥락 안의 단어 순서가 무시된다는 한계가 있다.
- 맥락의 단어 순서를 고려하기 위해 맥락의 단어 벡터를 은닉층에서 연결(concatenate)하는 방식을 생각할 수 있으나 맥락의 크기에 비례해 가중치 매개변수가 늘어난다는 문제가 발생한다.

그래서 순환 신경망 즉 RNN이 등장하게 되었는데 RNN은 맥락이 아무리 길더라도

맥락의 정보를 기억하는 메커니즘을 갖추고 있기에 아무리 긴 시계열 데이터에도 대응할 수 있다.

5. 순환 신경망(RNN)

5.1 확률과 언어 모델

5.2 RNN 이란

5.3 RNN 구현

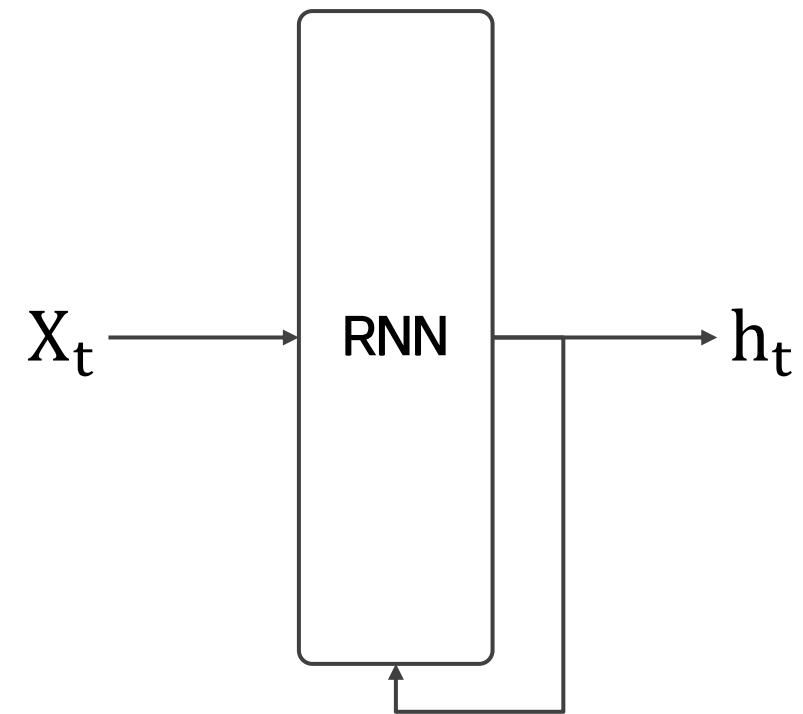
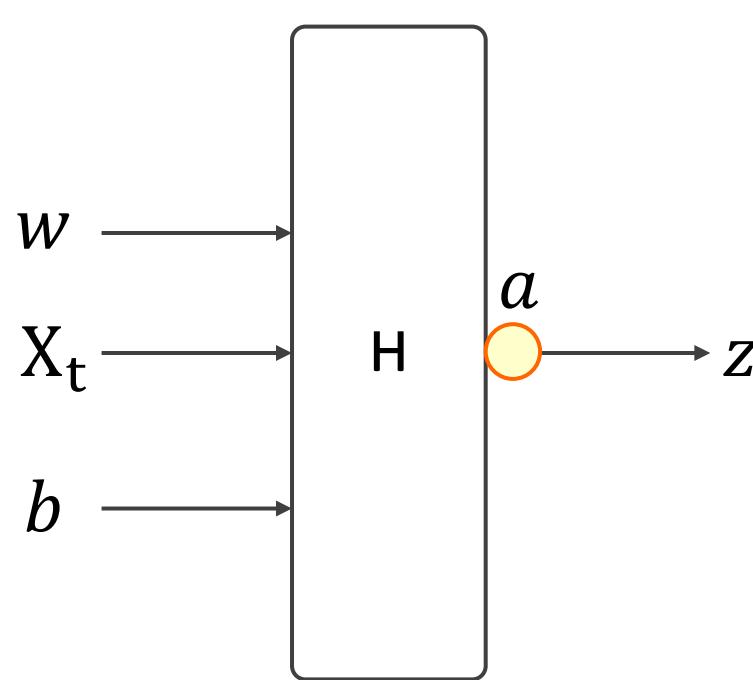
5.4 시계열 데이터 처리 계층 구현

5.5 RNNLM 학습과 평가

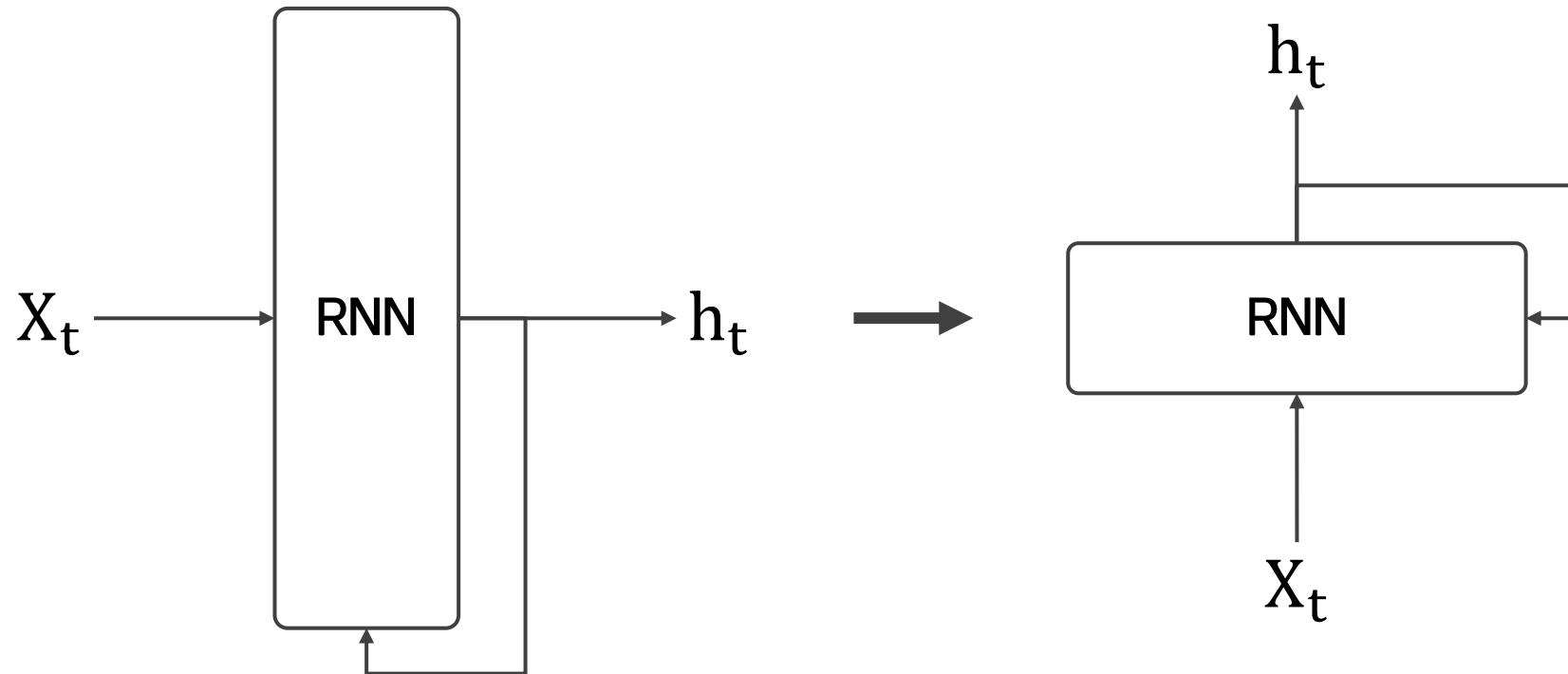
순환하기 위해서는 닫힌 경로가 필요하다.

닫힌 경로 혹은 순환하는 경로가 존재해야 데이터가 같은 장소를 반복해 왕래할 수 있고 데이터가 순환하면서 과거의 정보를 기억하는 동시에 최신 데이터로 갱신 될 수 있다.

순환 경로를 포함하는 RNN 계층



계층을 90도 회전시켜 그린다.



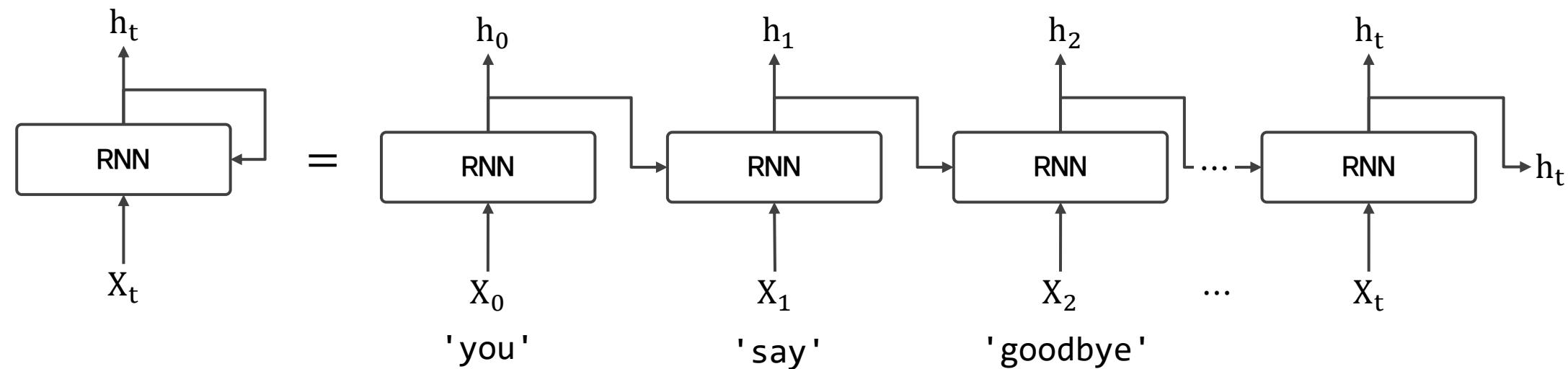
t : 시각

시계열 데이터($x_0, x_1, \dots, x_t, \dots$)가 RNN계층에 입력되고 이에 대응해 ($h_0, h_1, \dots, h_t, \dots$)가 출력된다.

각 시각에 입력되는 x_t 를 벡터라고 가정했을 때

문장(단어 순서)을 다루는 경우를 예로 든다면 각 단어의 분산 표현(단어 벡터)이 x_t 가 되며 이 분산 표현이 순서대로 하나씩 RNN계층에 입력된다.

RNN 계층의 순환 구조 펼치기



RNN계층의 순환 구조를 펼침으로써 오른쪽으로 성장하는 긴 신경망으로 변신
피드포워드 신경망(데이터가 한 방향으로만 흐른다)과 같은 구조이지만 위 그림에서는
다수의 RNN계층 모두가 실제로는 '같은 계층'인 것이 지금까지의 신경망과는 다른 점이다.

각 시각의 RNN계층은 그 계층으로의 입력과 1개 전의 RNN계층으로부터의 출력을 받는데
이 두 정보를 바탕으로 현 시각의 출력을 계산한다.

$$h_t = \tanh(h_{t-1}W_h + x_tW_x + b)$$

$$y = \text{sigmoid}(x) \Rightarrow y(1 - y)$$

$$h = \tanh(x) \Rightarrow 1 - h^2$$

Wx : 입력 x 를 출력 h 로 변환하기 위한 가중치

Wh : 1개의 RNN출력을 다음 시각의 출력으로 변환하기 위한 가중치

b : 편향

$h(t-1)$, x_t : 행벡터

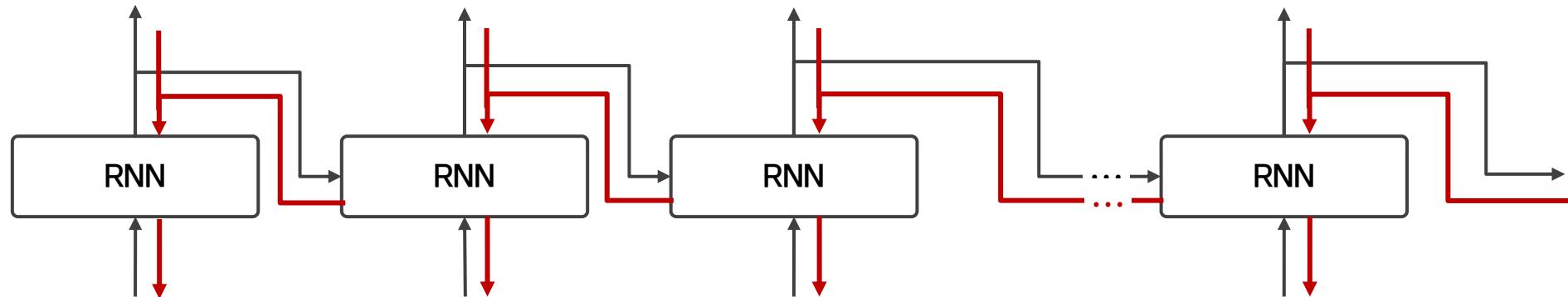
h_t 는 다른 계층을 향해 위쪽으로 출력되는 동시에 다음 시각의 RNN계층(자기 자신)을 향해 오른쪽으로도 출력된다.

RNN의 출력 h_t 는 은닉상태(hidden state) 혹은 은닉 상태 벡터(hidden state vector)라고 한다.

RNN은 h 라는 '상태'를 가지고 있으며 위의 식의 형태로 간신된다고 해석할 수 있다.

RNN계층을 '상태를 가지는 계층' 혹은 '메모리(기억력)가 있는 계층'이라고 한다.

순환 구조를 펼친 RNN 계층에서의 오차역전파



순환 구조를 펼친 후의 RNN에는 (일반적인) 오차역전파법을 적용할 수 있다.

먼저 순전파를 수행하고 이어서 역전파를 수행하여 원하는 기울기를 구할 수 있다.

여기서의 오차역전파법은 '시간 방향으로 펼친 신경망의 오차역전파법'이란 뜻으로

BPTT(Backpropagation Through Time)라고 한다.

문제점

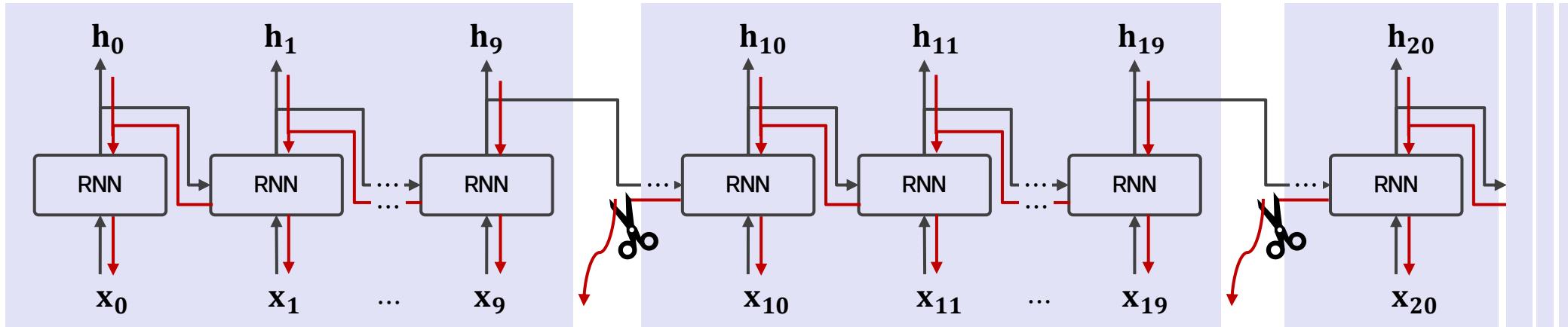
- 시계열 데이터의 시간 크기가 커지는 것에 비례하여 BPTT가 소비하는 컴퓨팅 자원도 증가
- 시간 크기가 커지면 역전파 시의 기울기가 불안정해짐

Truncated BPTT : 시간축 방향으로 너무 길어진 신경망을 적당한 지점에서 잘라내어 작은 신경망 여러 개로 만들어 잘라낸 작은 신경망에서 오차역전파법을 수행한다.

- 계층이 너무 길면 계산량과 메모리 사용량 등이 문제가 되고 계층이 길어짐에 따라 신경망을 하나 통과할 때마다 기울기 값이 조금씩 작아져서 이전 시각 t 까지 역전파되기 전에 0이 되어 소멸할 수도 있다.
- 순전파의 연결을 그대로 유지하면서(데이터를 순서대로 입력해야 한다) 역전파의 연결은 적당한 길이로 잘라내 잘라낸 신경망 단위로 학습을 수행한다.
- 역전파의 연결을 잘라버리면 그보다 미래의 데이터에 대해서는 생각할 필요가 없어지기 때문에 각각의 블록 단위로 미래의 블록과는 독립적으로 오차역전파법을 완결시킨다.
 - 블록: 역전파가 연결되는 일련의 RNN계층
- 순전파를 수행하고 그 다음 역전파를 수행하여 원하는 기울기를 구한다.
- 다음 역전파를 수행할 때 앞 블록의 마지막 은닉 상태인 ht 가 필요하다. ht 로 순전파가 계속 연결될 수 있다.

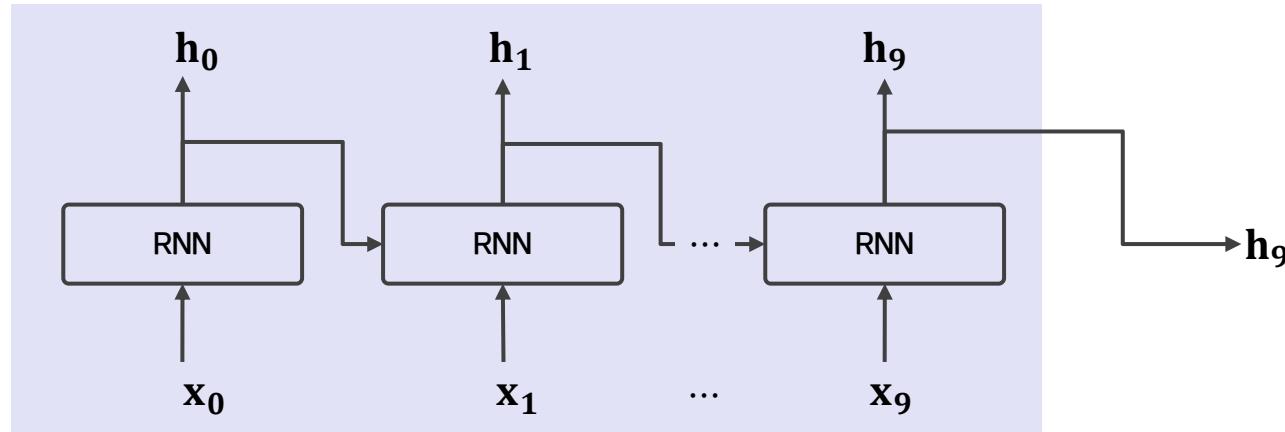
역전파의 연결을 적당한 지점에서 끊는다.

$T=0$

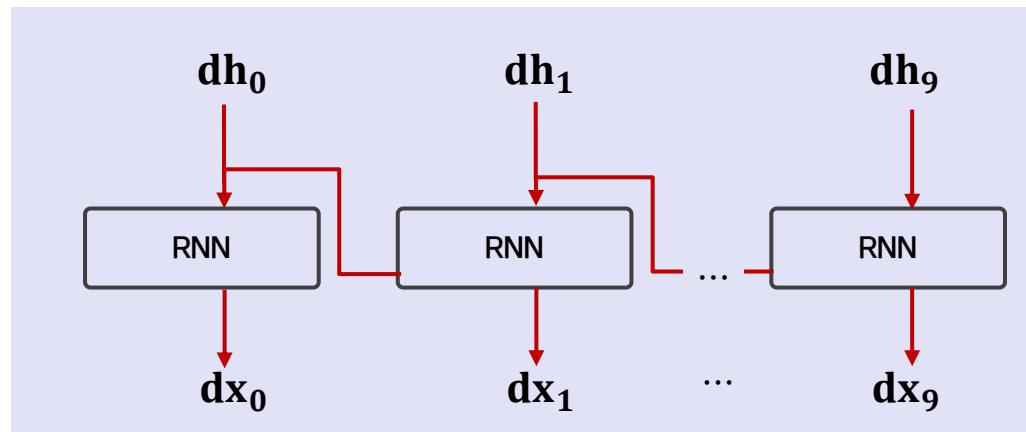


첫 번째 블록의 순전파와 역전파 : 이보다 앞선 시각으로부터의 기울기는 끊겼기 때문에 이 블록 내에서만 오차역전파법이 완결된다.

순전파

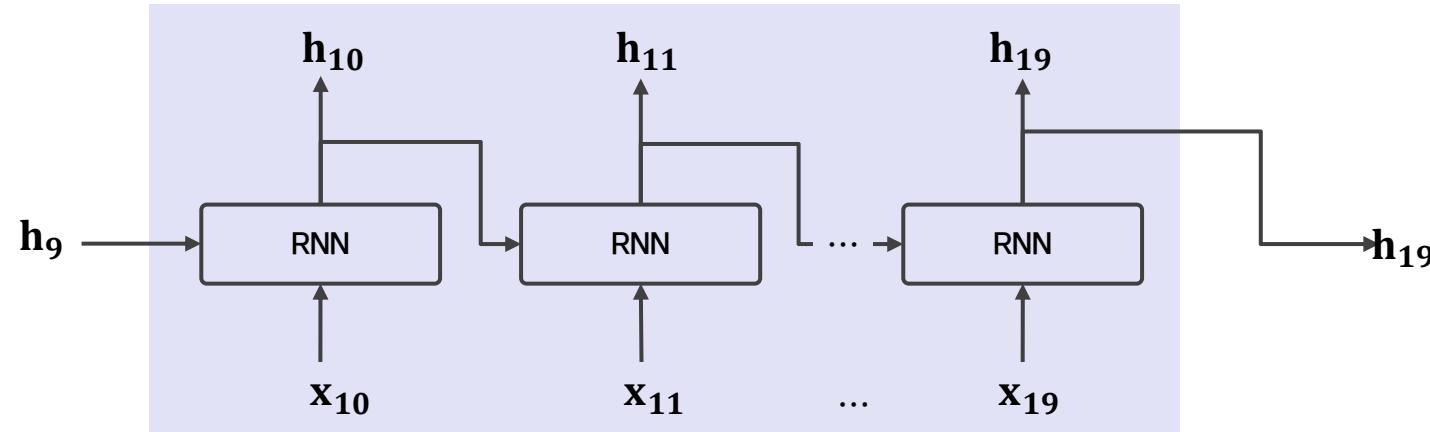


역전파

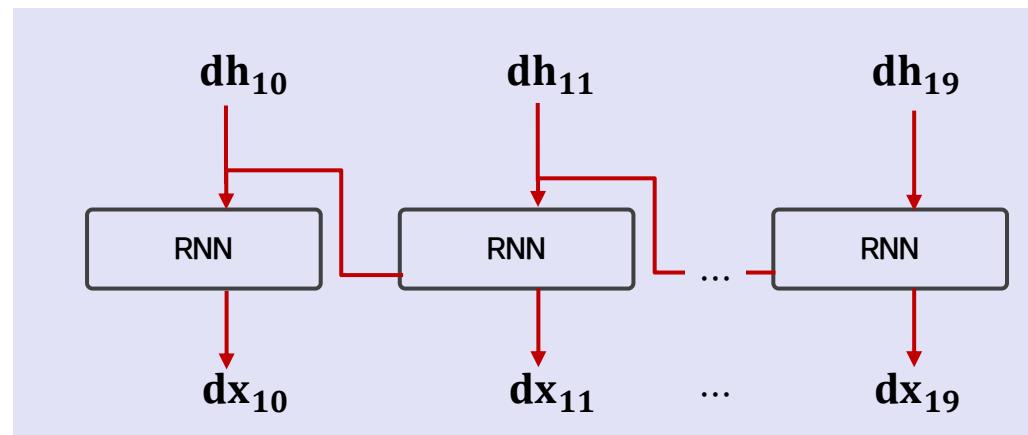


두 번째 블록의 순전파와 역전파

순전파

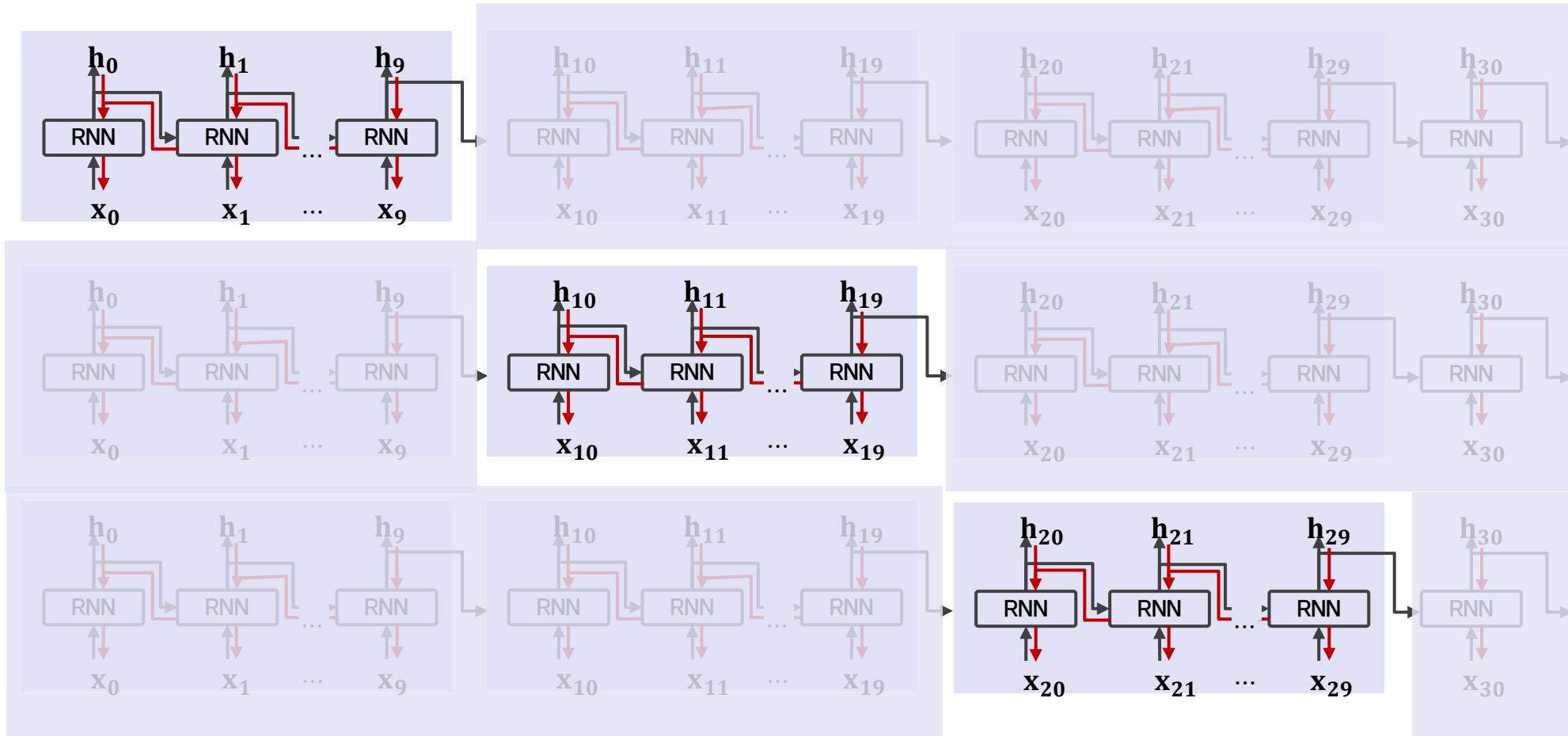


역전파



Truncated BPTT의 데이터 처리 순서

학습처리순서



미니배치 학습 시 데이터를 제공하는 시작 위치를 각 미니배치로 옮긴다.

학습처리순서

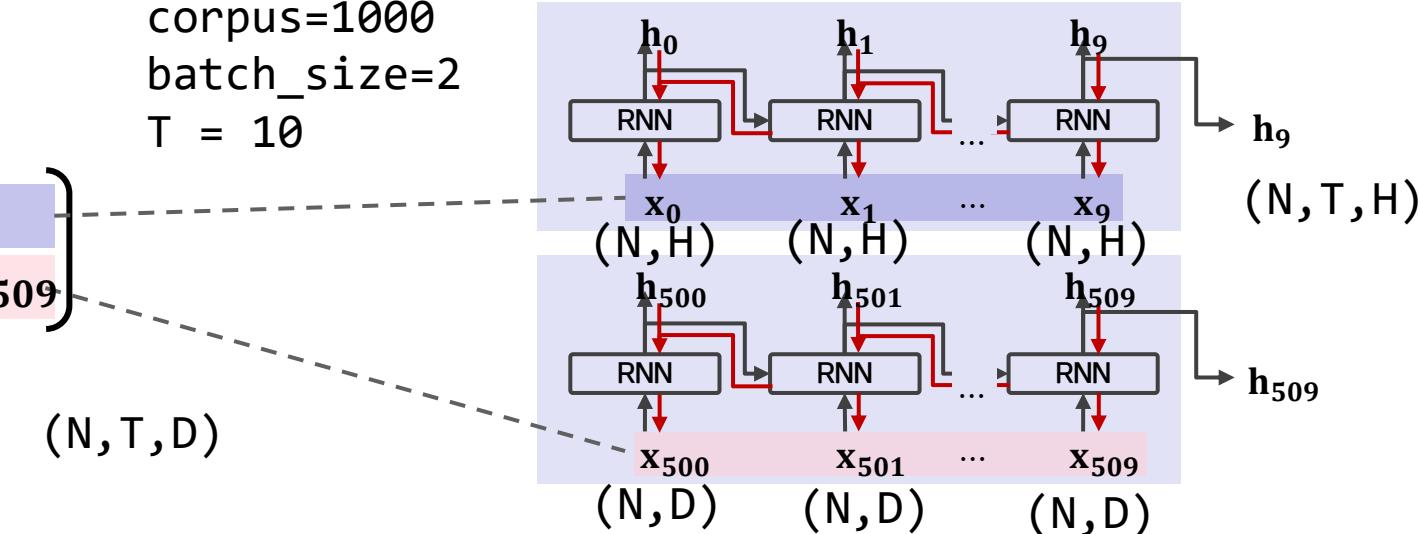
첫 번째 미니배치의 원소

$\{x_0, x_1, \dots, x_9\}$
 $x_{500}, x_{501}, \dots, x_{509}\}$

두 번째 미니배치의 원소

(N, T, D)

corpus=1000
batch_size=2
T = 10

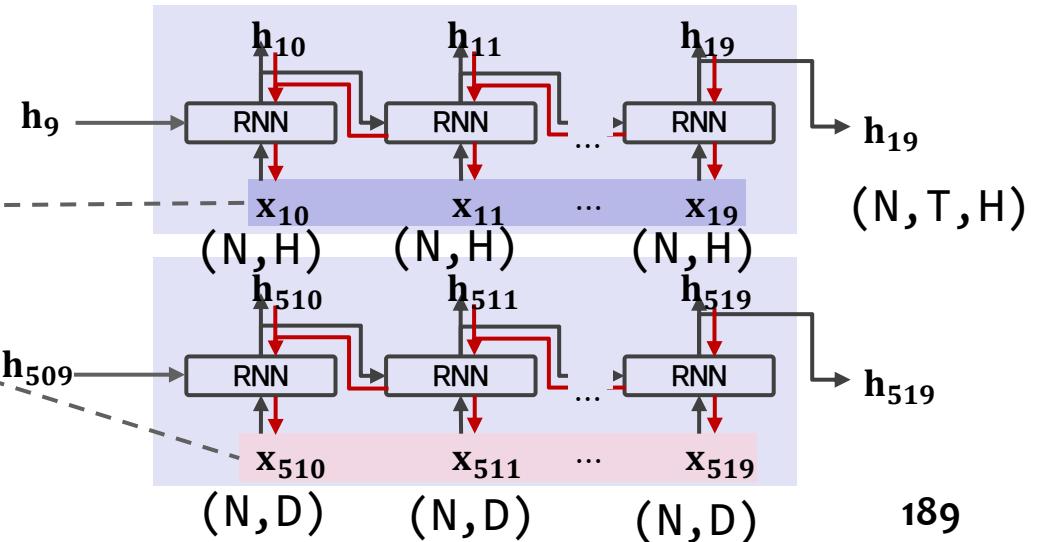


TBPTT

$\{x_{10}, x_{11}, \dots, x_{19}\}$
 $x_{510}, x_{511}, \dots, x_{519}\}$

(N, T, D)

⋮



5. 순환 신경망(RNN)

5.1 확률과 언어 모델

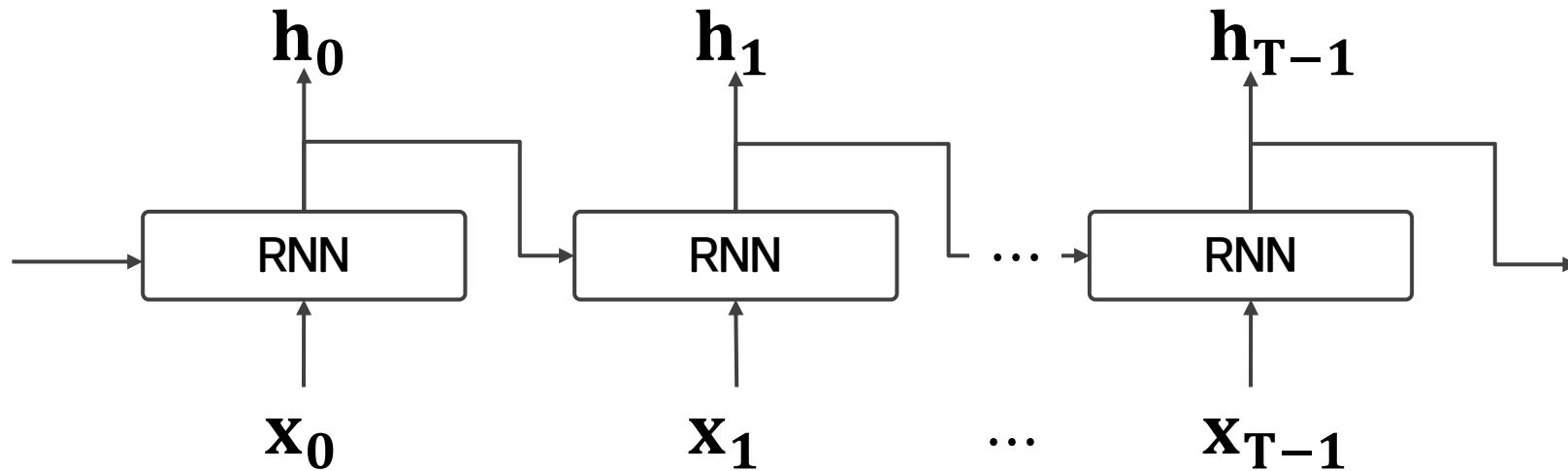
5.2 RNN 이란

5.3 RNN 구현

5.4 시계열 데이터 처리 계층 구현

5.5 RNNLM 학습과 평가

RNN에서 다루는 신경망

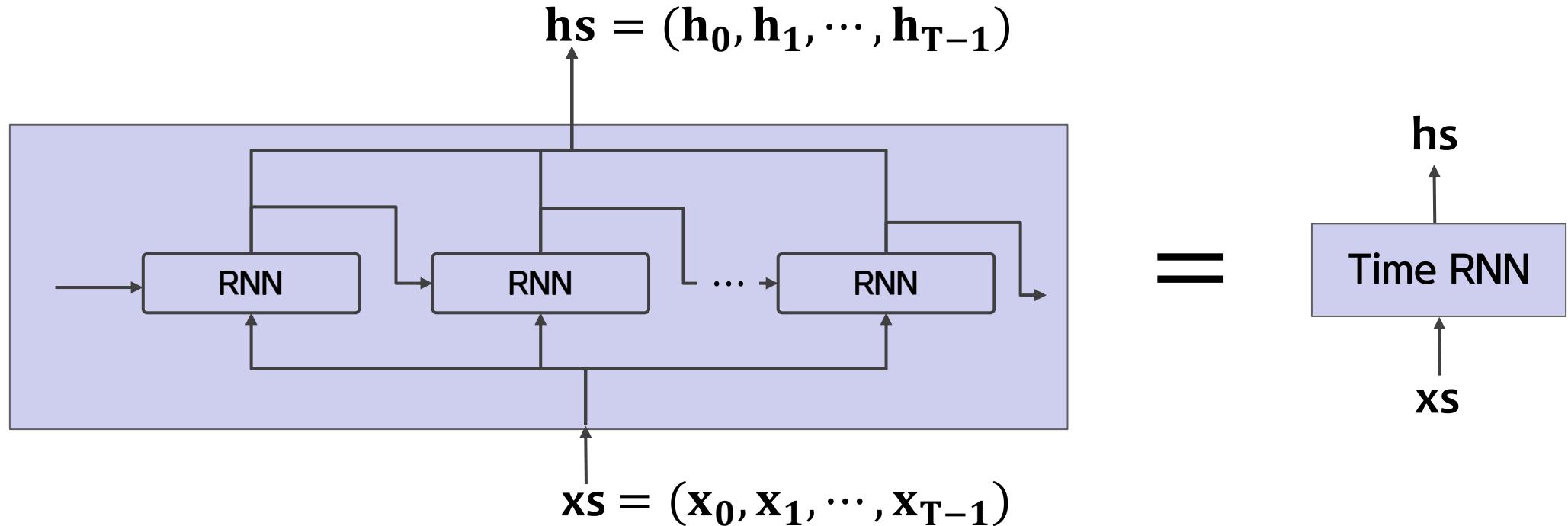


길이가 T 인 시계열 데이터를 받는다.

각 시각의 은닉 상태를 T 개 출력한다.

모듈화를 생각해 위의 그림의 신경망을 '하나의 계층'으로 구현한다.

Time RNN 계층 : 순환 구조를 펼친 후의 계층들을 하나의 계층으로 간주한다.



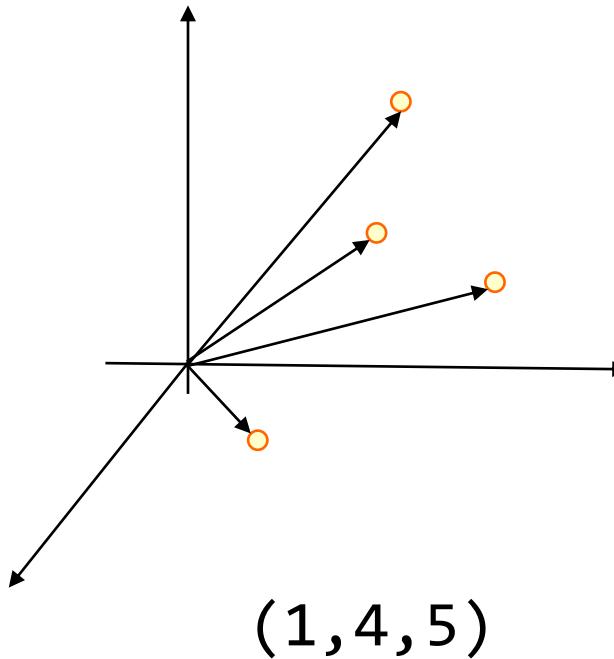
xs 를 입력하면 hs 를 출력하는 단일 계층

Time RNN계층 내에서 한 단계의 작업을 수행하는 계층을 'RNN계층'이라 하고
T개 단계분의 작업을 한꺼번에 처리하는 계층을 'Time RNN계층'이라 한다.

word = 'student'

입력 벡터의 차원수

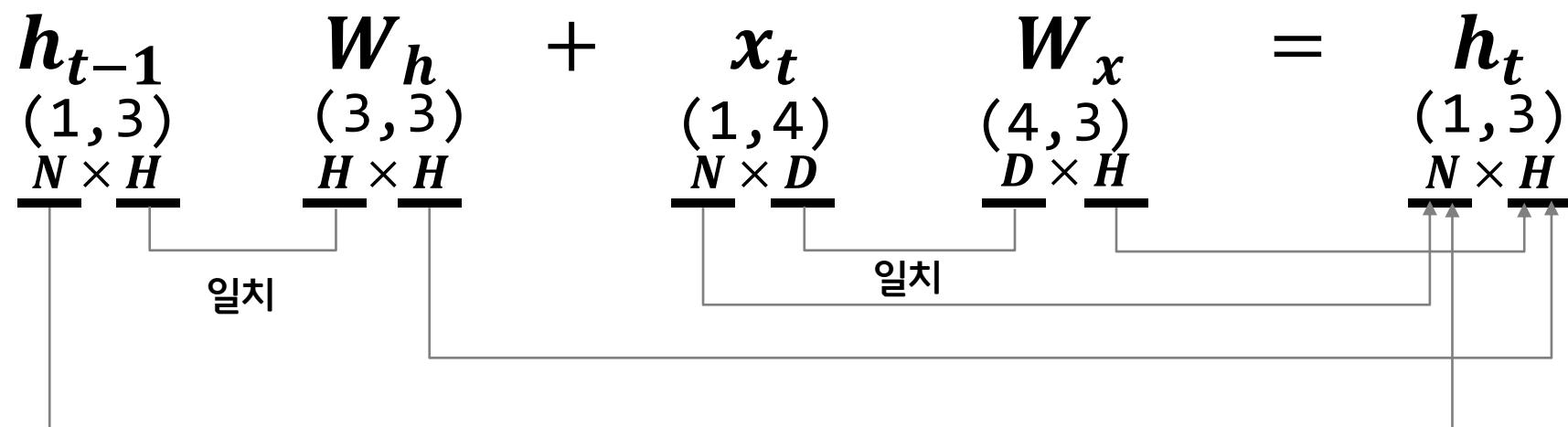
word_to_vector => embedding 층



512 차원

$$h_t = \tanh(h_{t-1}W_h + x_tW_x + b)$$

형상 확인 : 형렬 곱에서는 대응하는 차원의 원소 수를 일치시킨다.

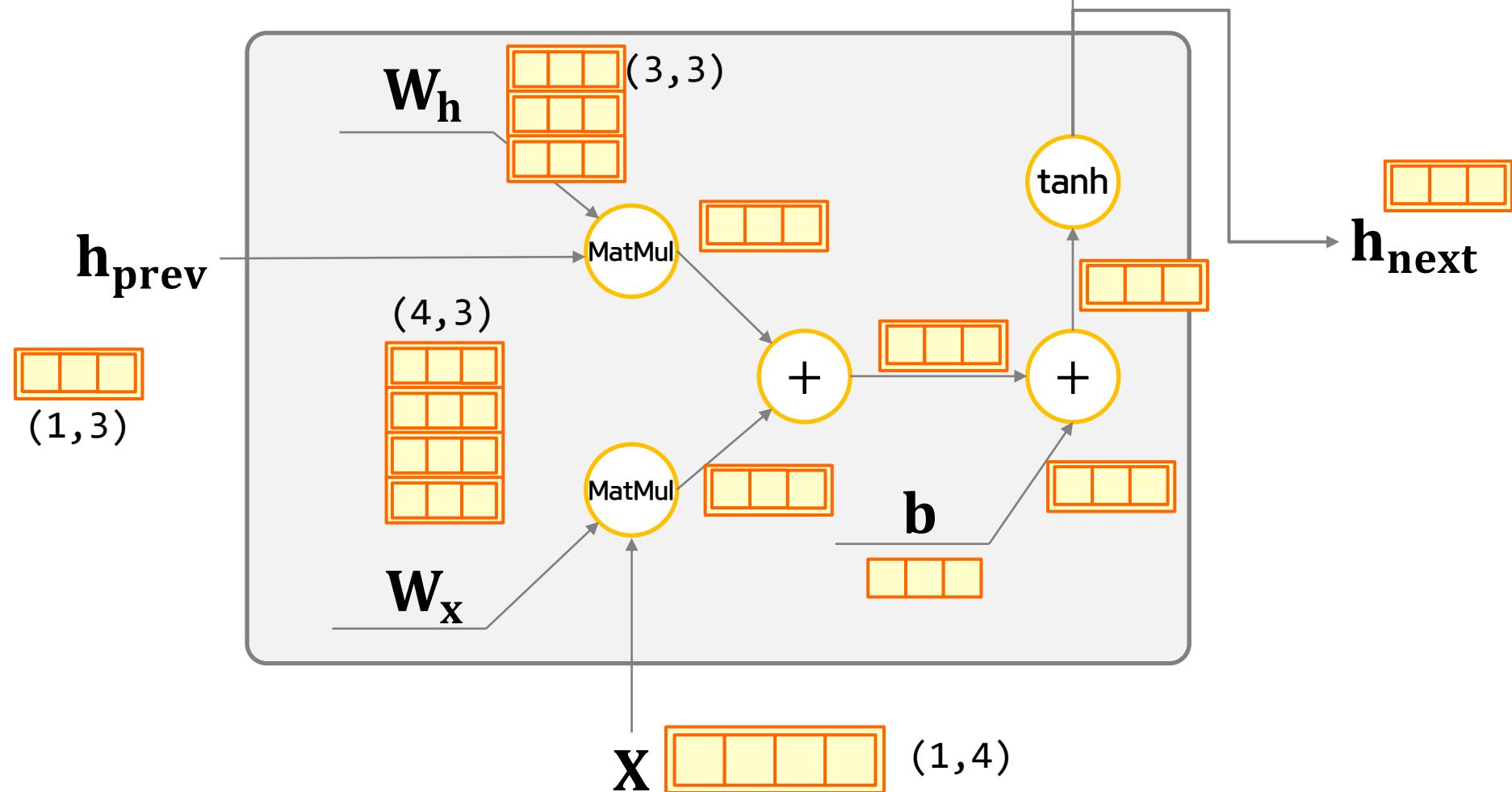


N: 미니배치 크기 D: 입력 벡터의 차원 수 H: 은닉 상태 벡터의 차원 수

RNN 계층의 계산 그래프

$$h_t = \tanh(h_{t-1}W_h + x_tW_x + b)$$

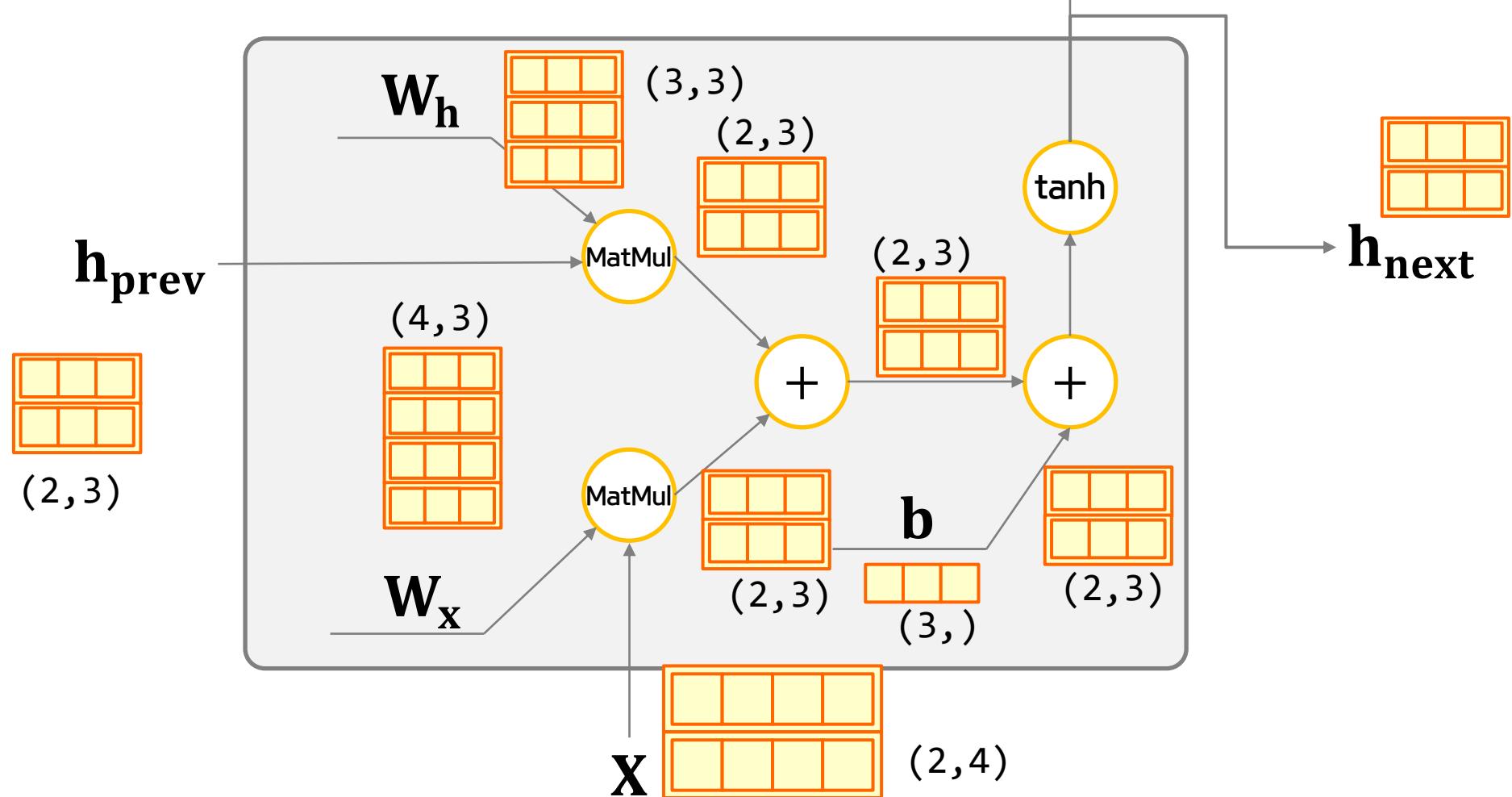
h_{next}



RNN 계층의 계산 그래프

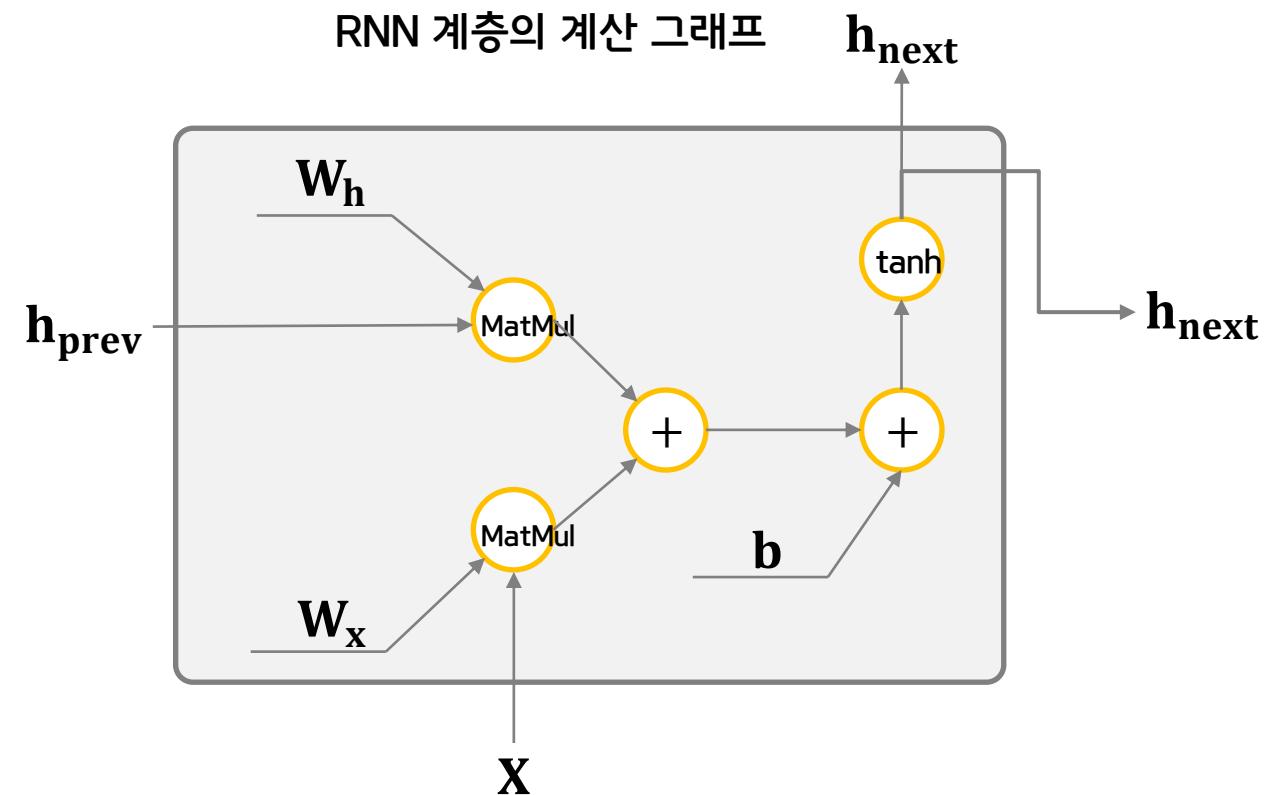
$$h_t = \tanh(h_{t-1}W_h + x_tW_x + b)$$

h_{next}

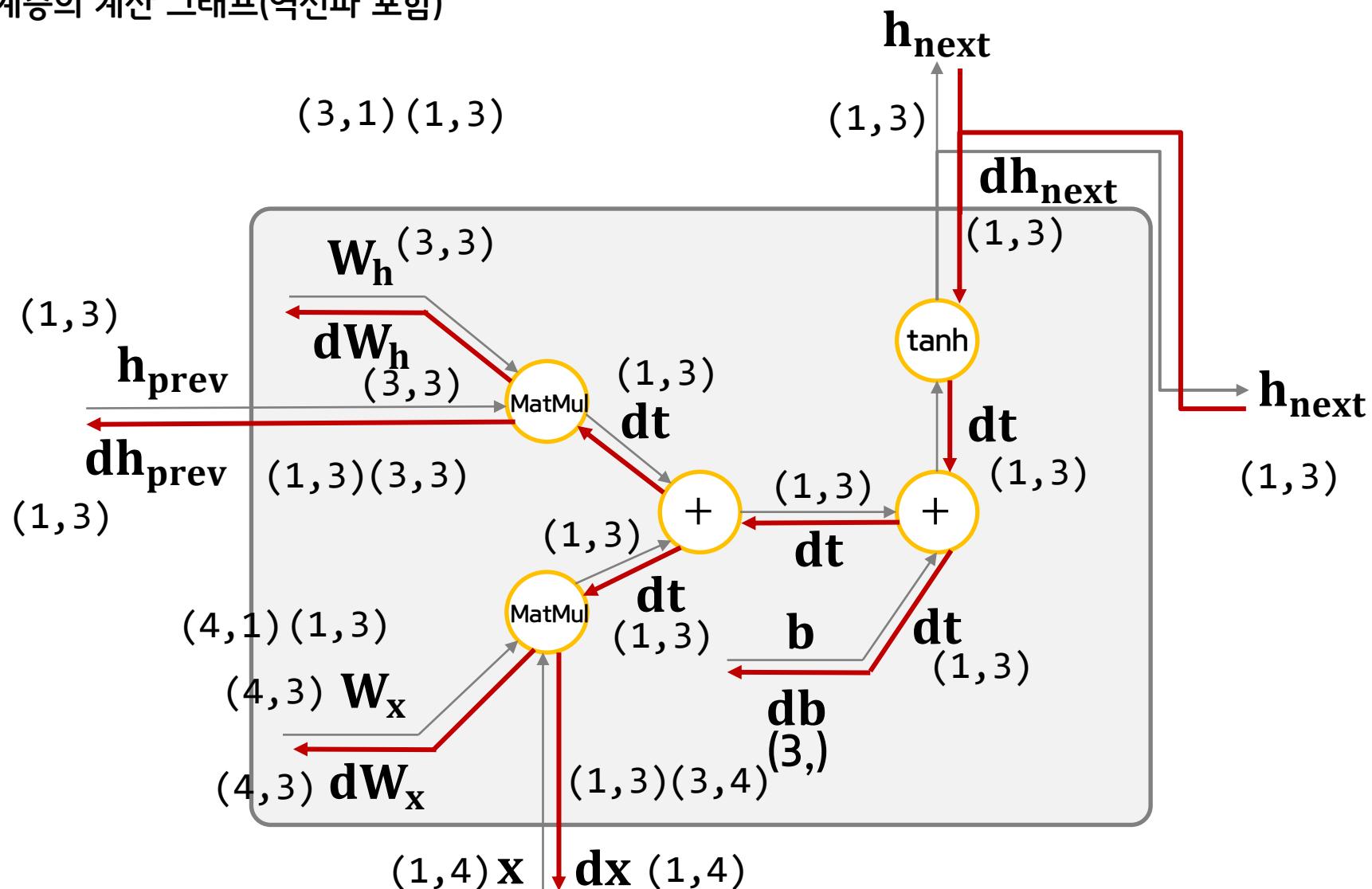


```
def forward(self, x, h_prev):
    Wx, Wh, b = self.params
    t = np.dot(h_prev, Wh) + np.dot(x, Wx) + b
    h_next = np.tanh(t)

    self.cache = (x, h_prev, h_next)
    return h_next
```



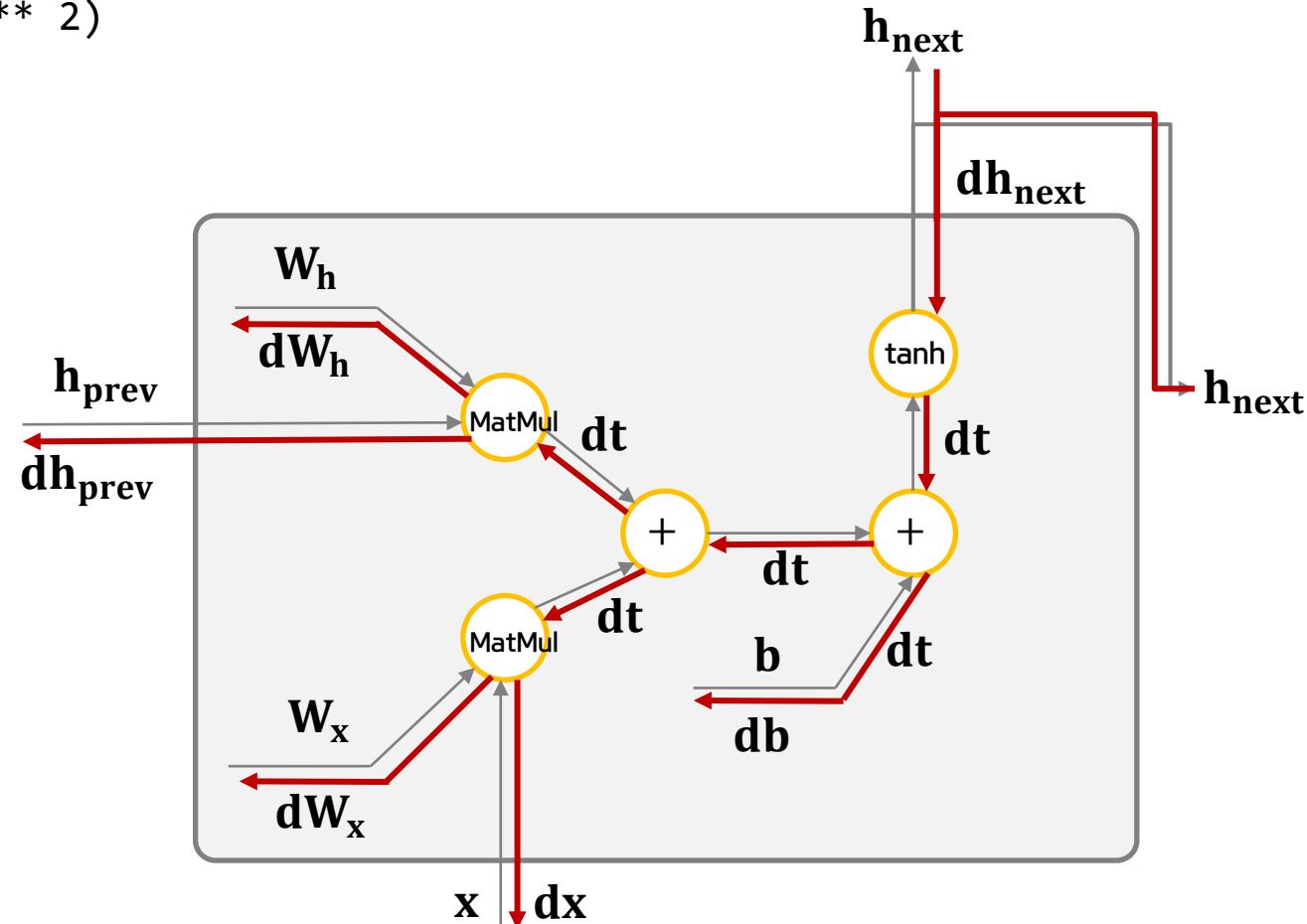
RNN 계층의 계산 그래프(역전파 포함)



```
def backward(self, dh_next):
    Wx, Wh, b = self.params
    x, h_prev, h_next = self.cache

    dt = dh_next * (1 - h_next ** 2)
    db = np.sum(dt, axis=0)
    dWh = np.dot(h_prev.T, dt)
    dh_prev = np.dot(dt, Wh.T)
    dWx = np.dot(x.T, dt)
    dx = np.dot(dt, Wx.T)
```

RNN 계층의 계산 그래프(역전파 포함)



```
corpus= [0 1 2 3 4 1 5 6] "you say goodbye and i say hello ."
```

```
xs = corpus[:-1]
```

```
[0 1 2 3 4 1 5]
```

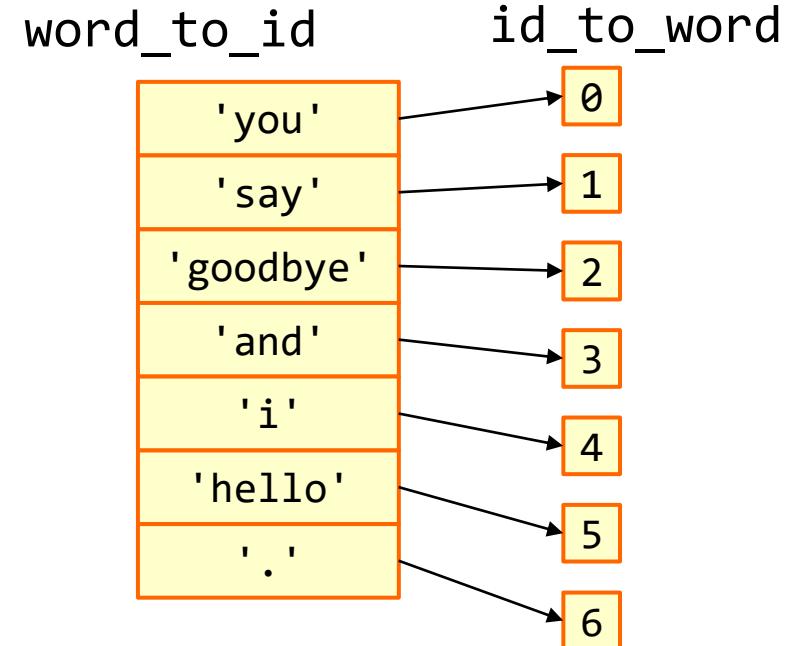
```
ts = corpus[1:]
```

```
[1 2 3 4 1 5 6]
```

V=7

D=4

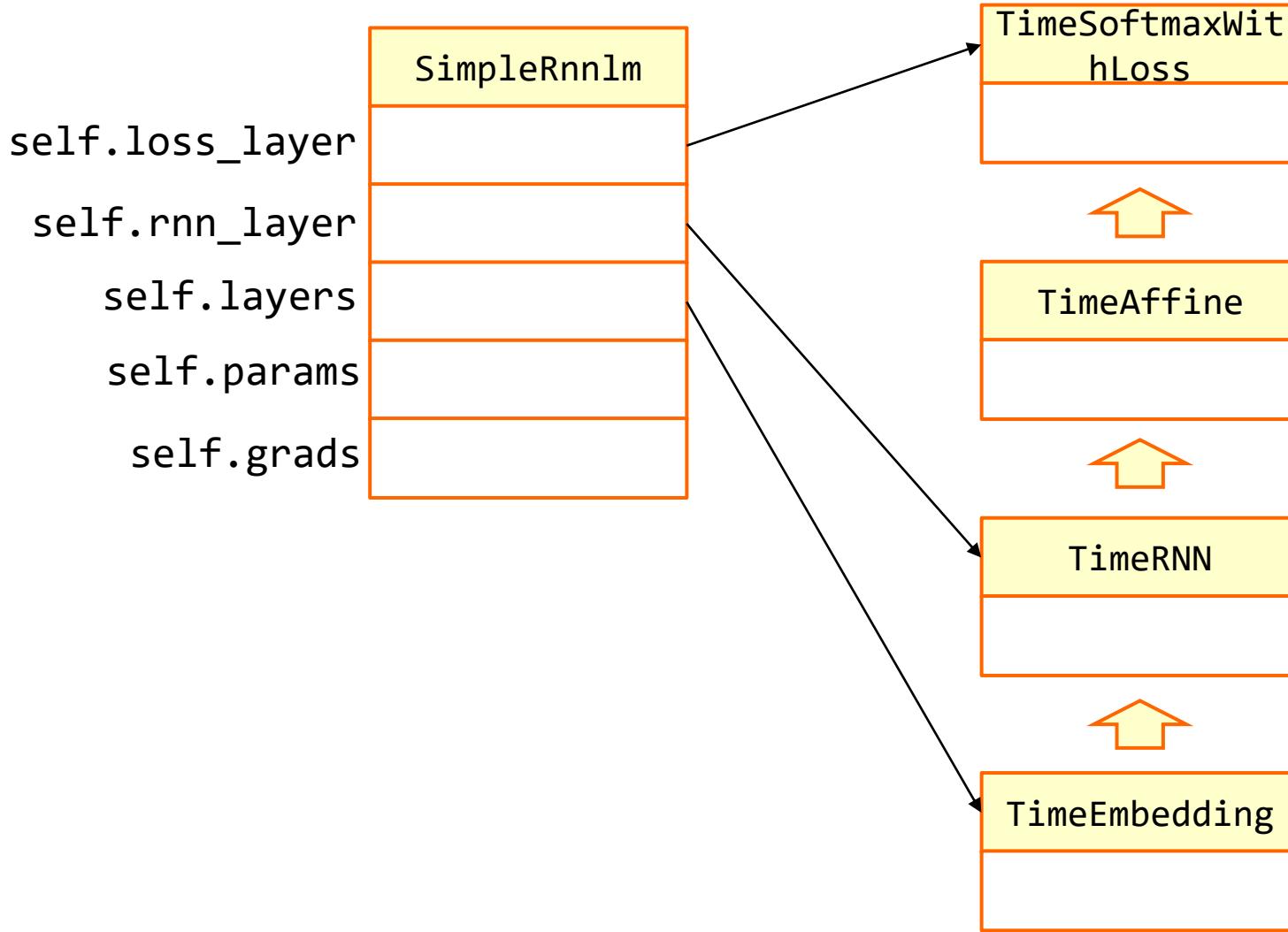
H=3



```
model = SimpleRnnlm(vocab_size=7, wordvec_size=4, hidden_size=3)
```

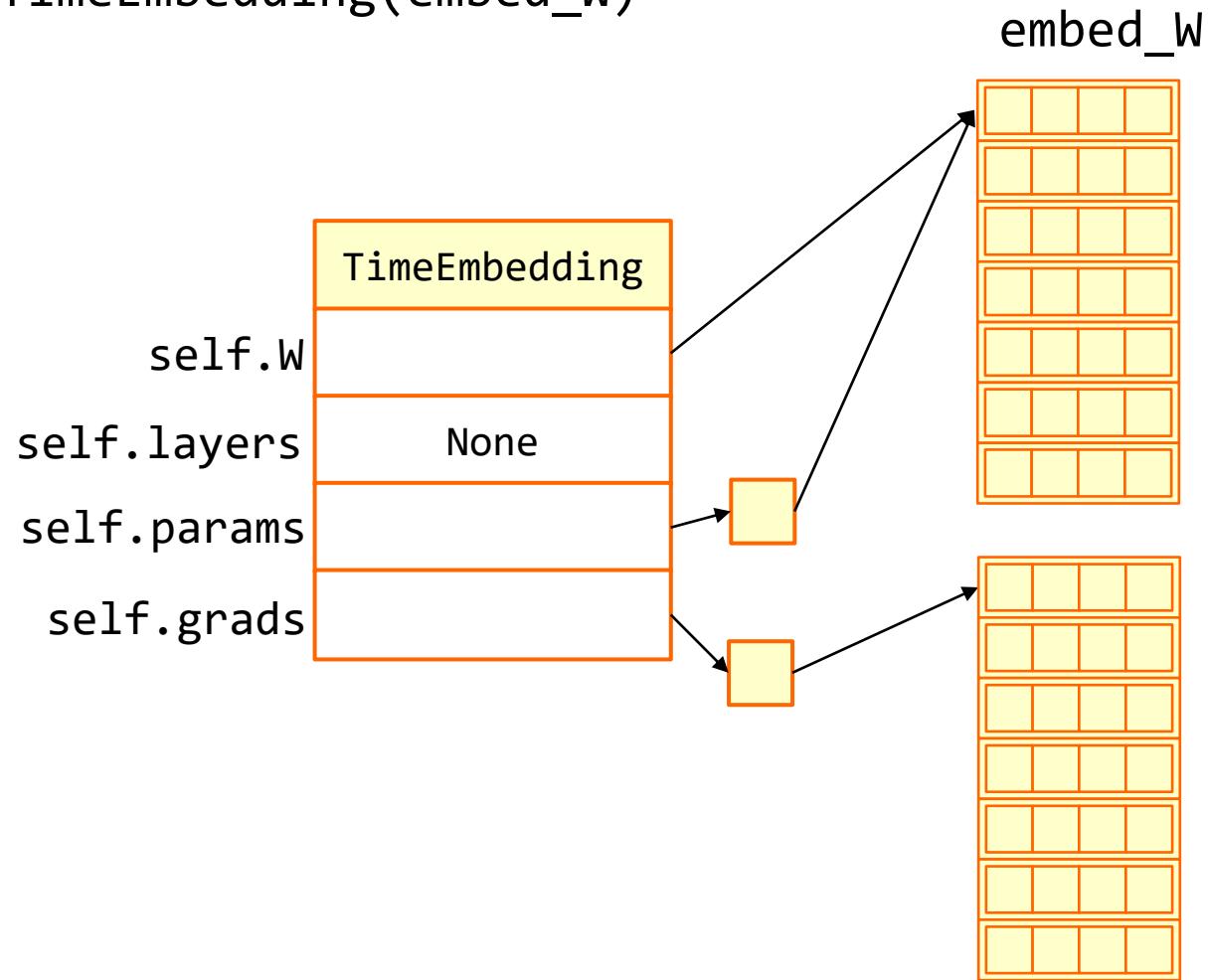
```
trainer.fit(xs, ts, max_epoch, batch_size=1, time_size)  
T=3
```

```
model = SimpleRnnlm(vocab_size=7, wordvec_size=4, hidden_size=3)
```

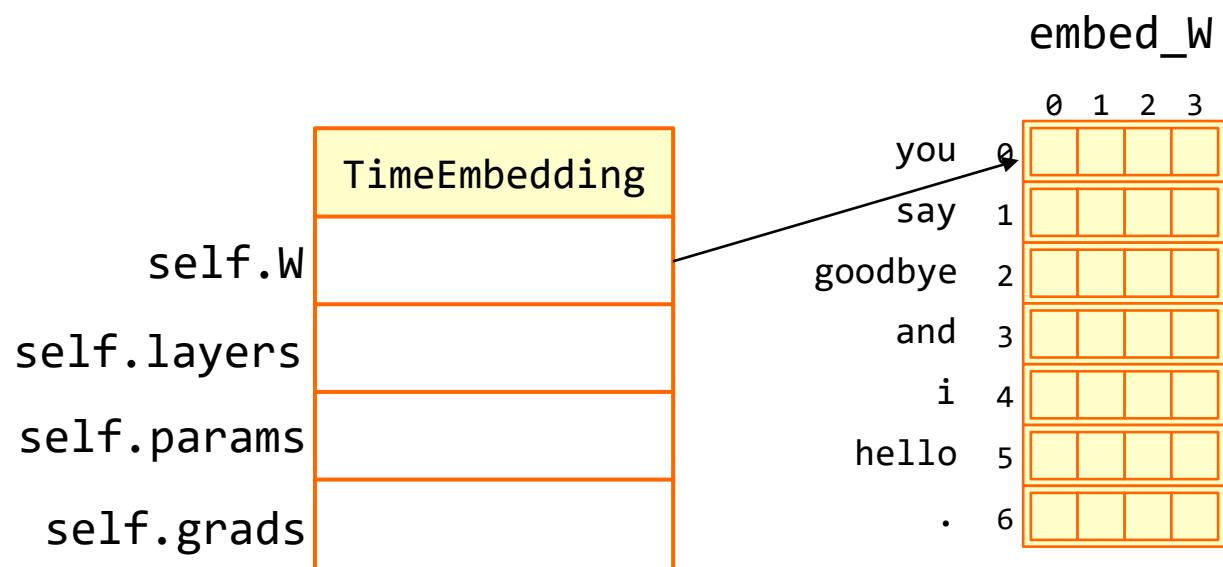
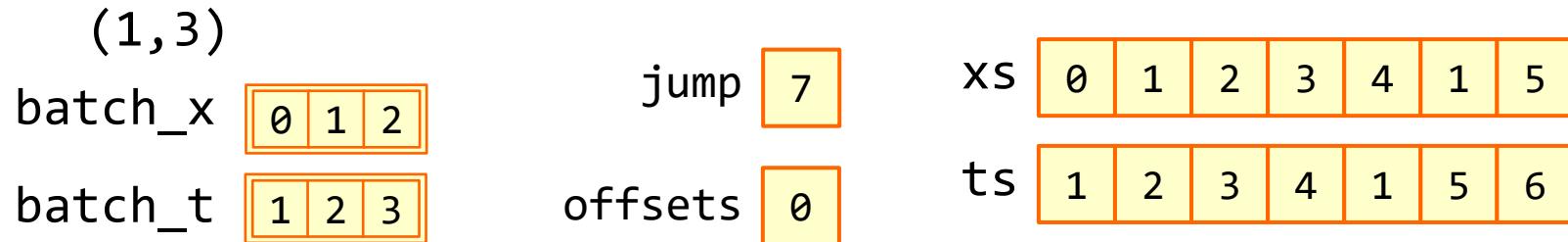


```
embed_W = (rn(V, D) / 100).astype('f')
```

```
TimeEmbedding(embed_W)
```

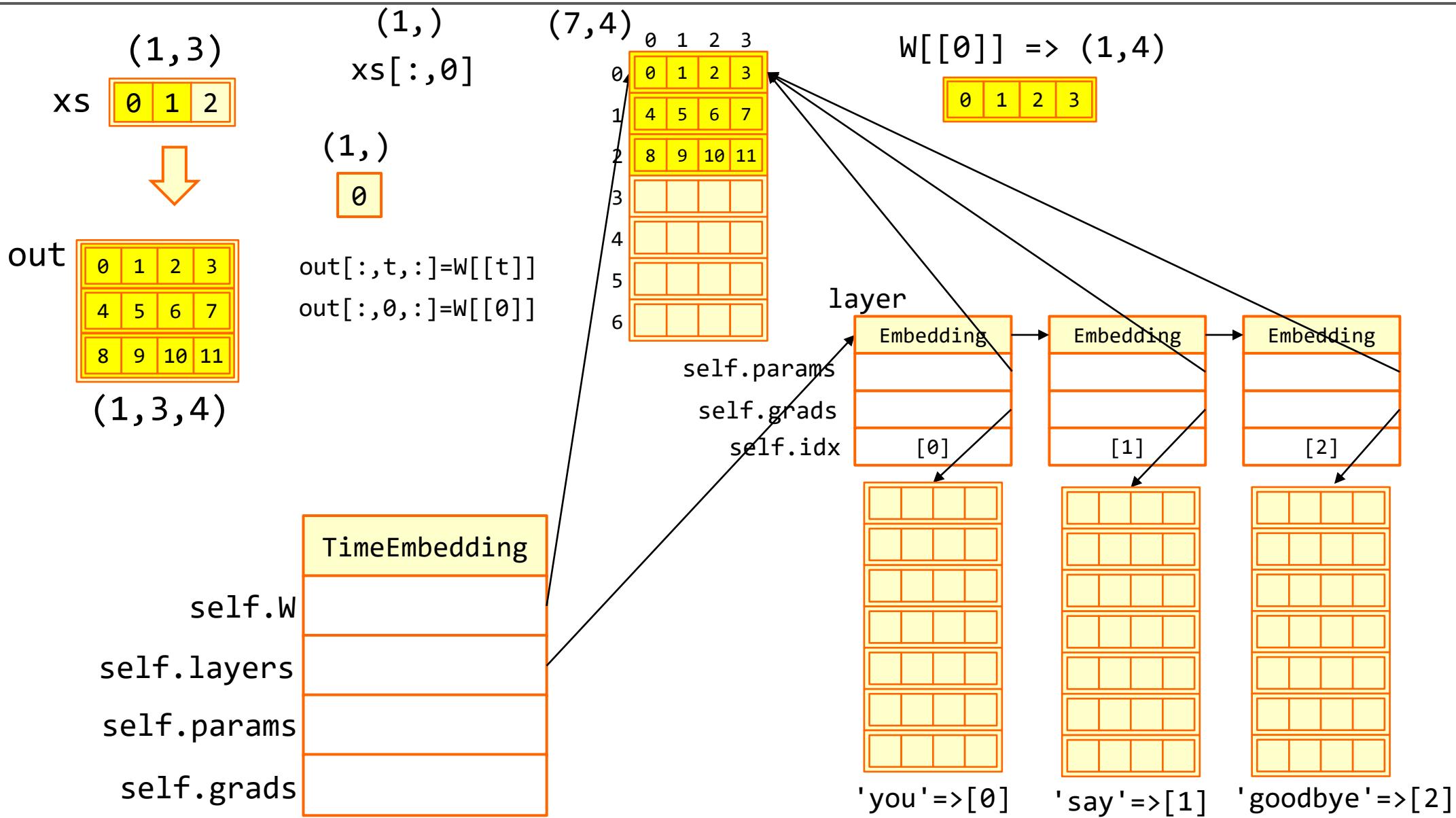


```
batch_x = np.empty((batch_size=1, time_size=3), dtype='i')
```



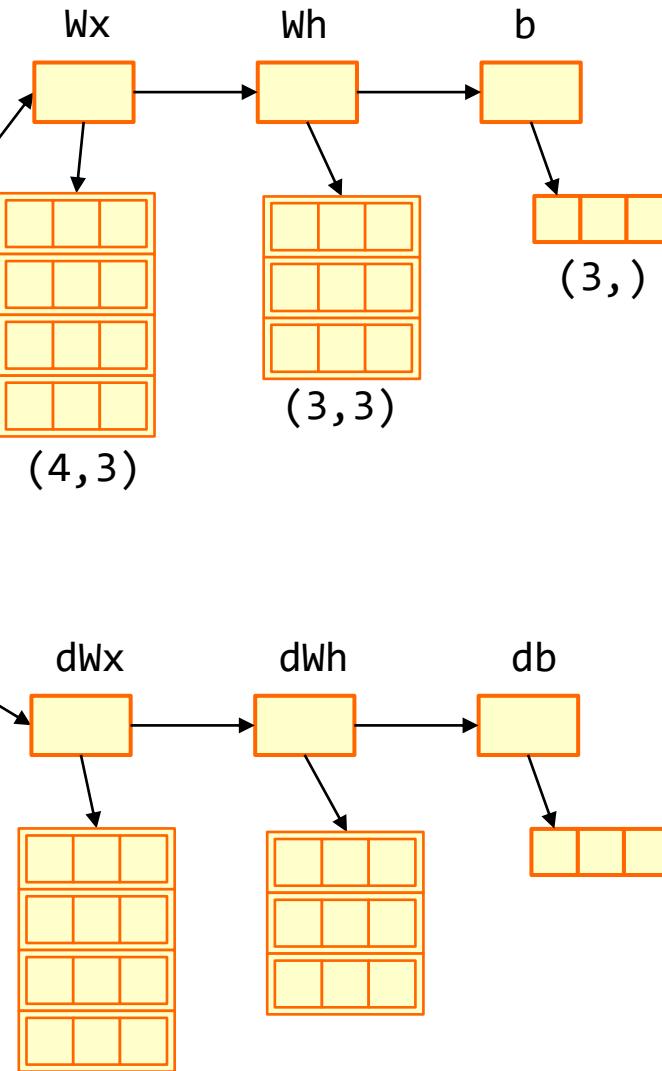
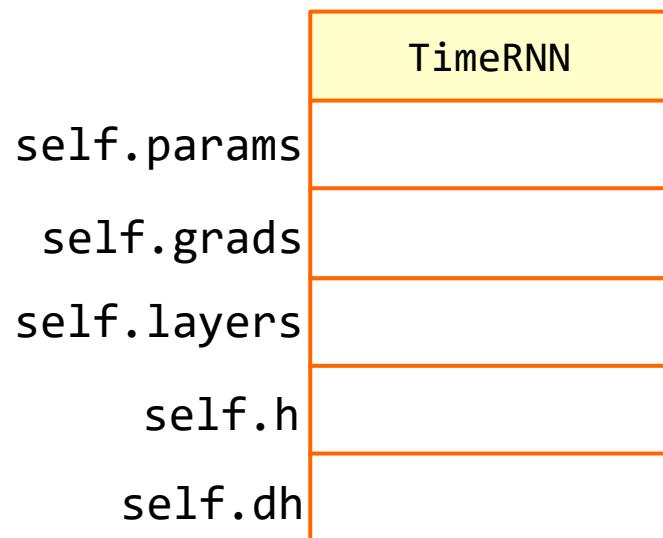
TimeEmbedding.forward(self, xs)

$w[0] \Rightarrow (4,)$

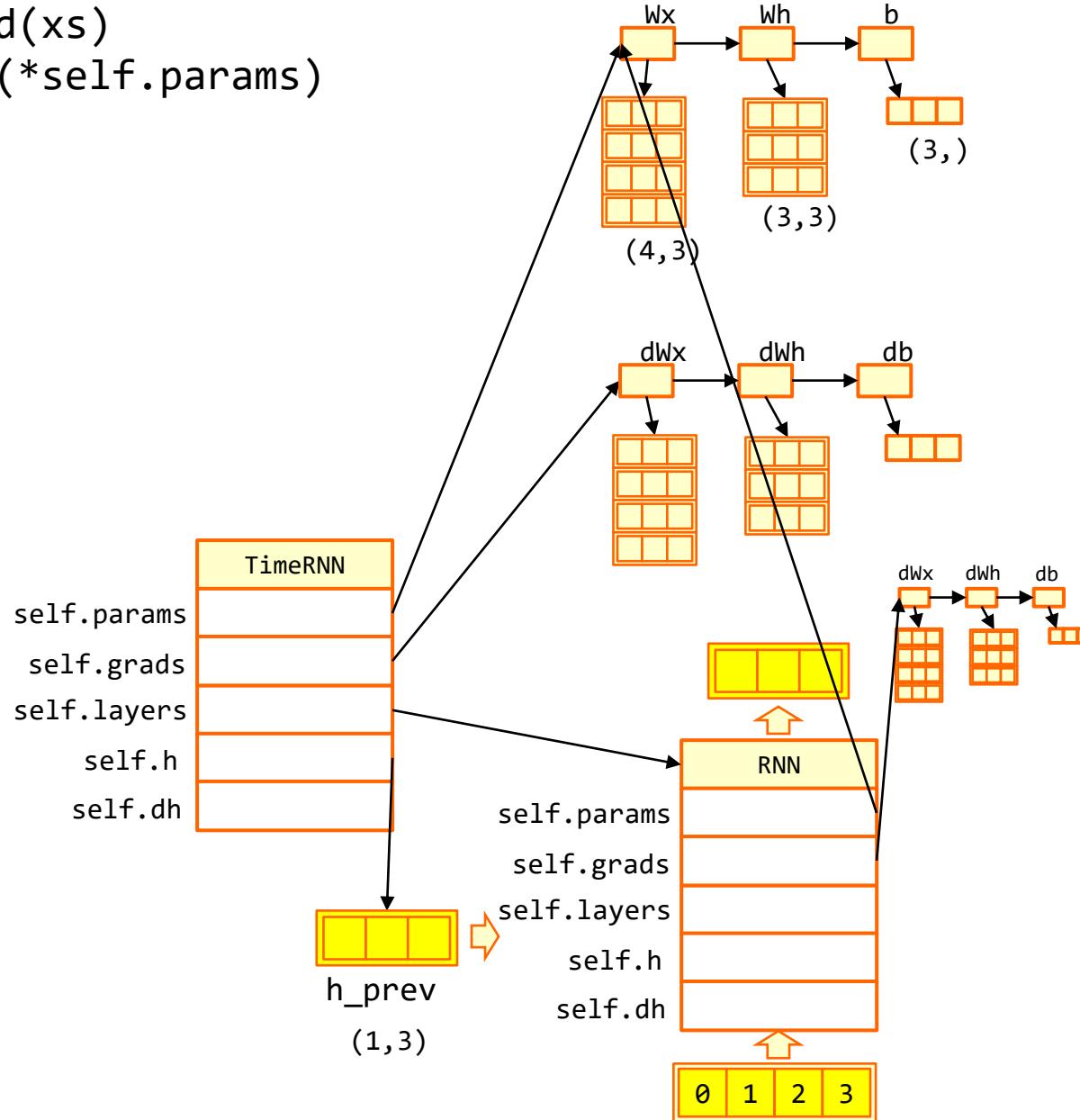
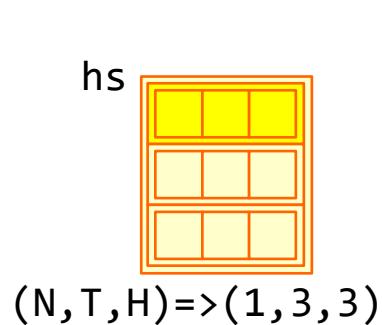
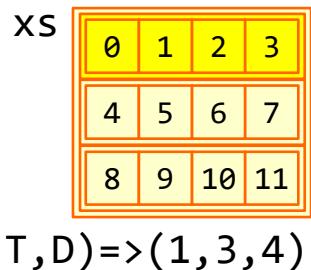


TimeRNN.__init__()

$$D=4$$
$$H=3$$

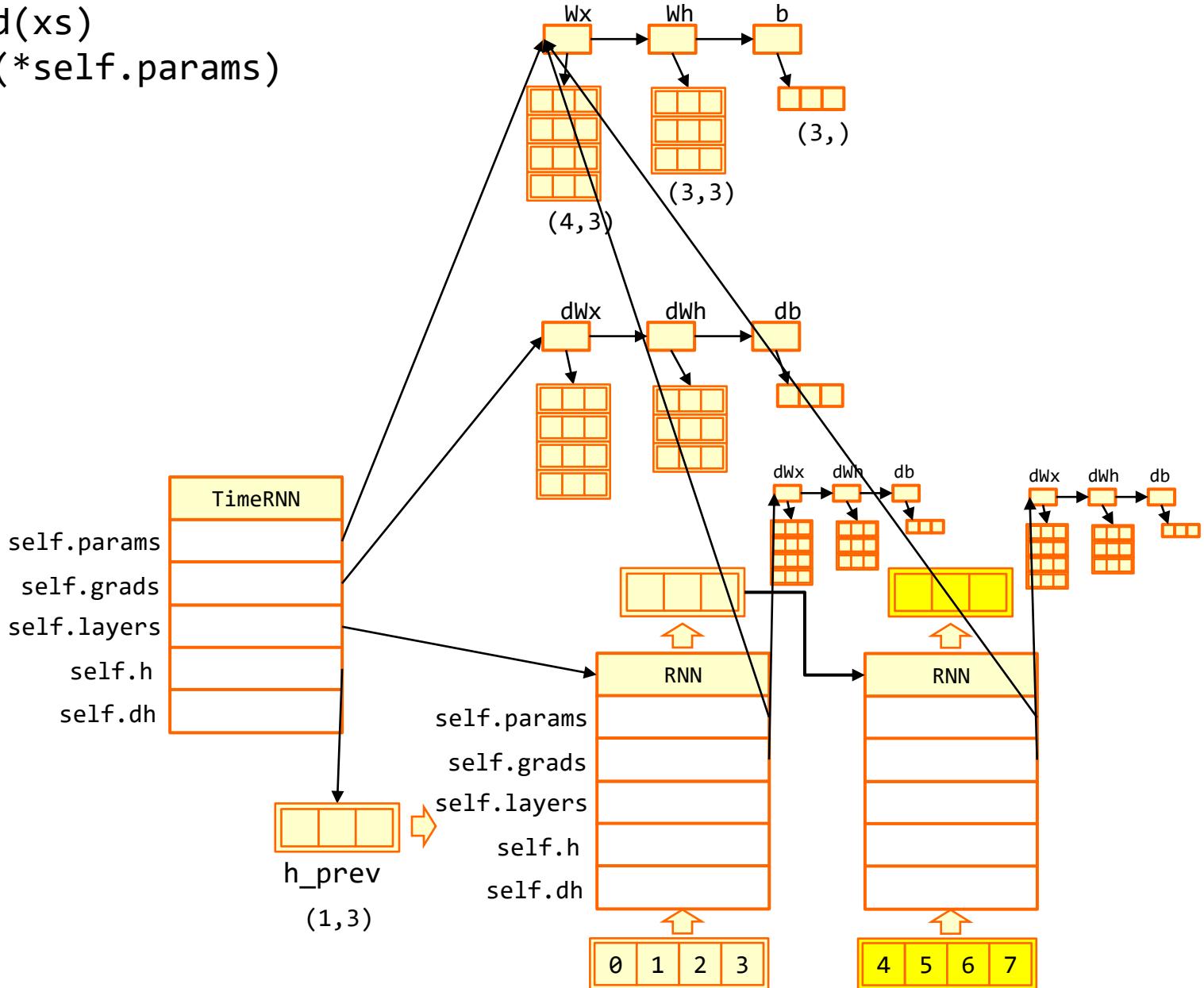
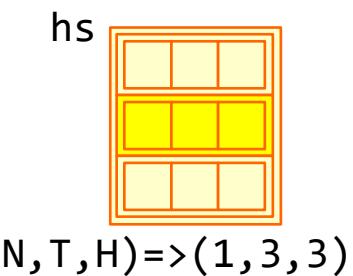
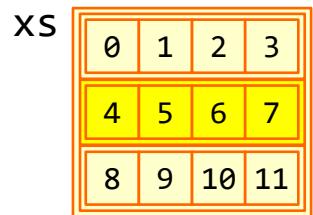


```
TimeRNN.forward(xs)
    layer = RNN(*self.params)
```



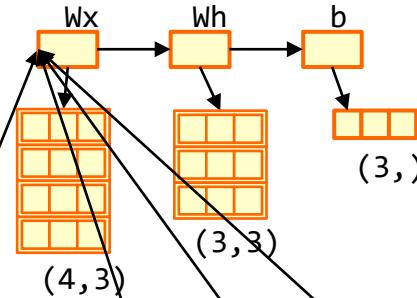
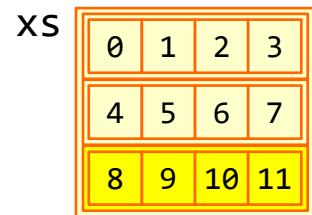
```
TimeRNN.forward(xs)
```

```
    layer = RNN(*self.params)
```

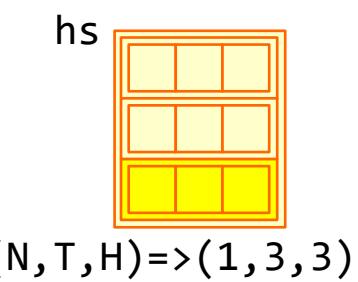
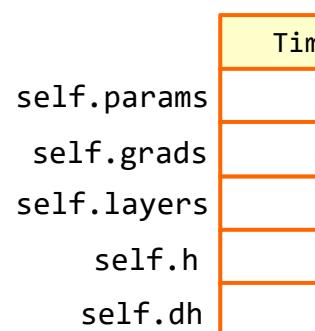


TimeRNN.forward(xs)

layer = RNN(*self.params)



$(N, T, D) \Rightarrow (1, 3, 4)$



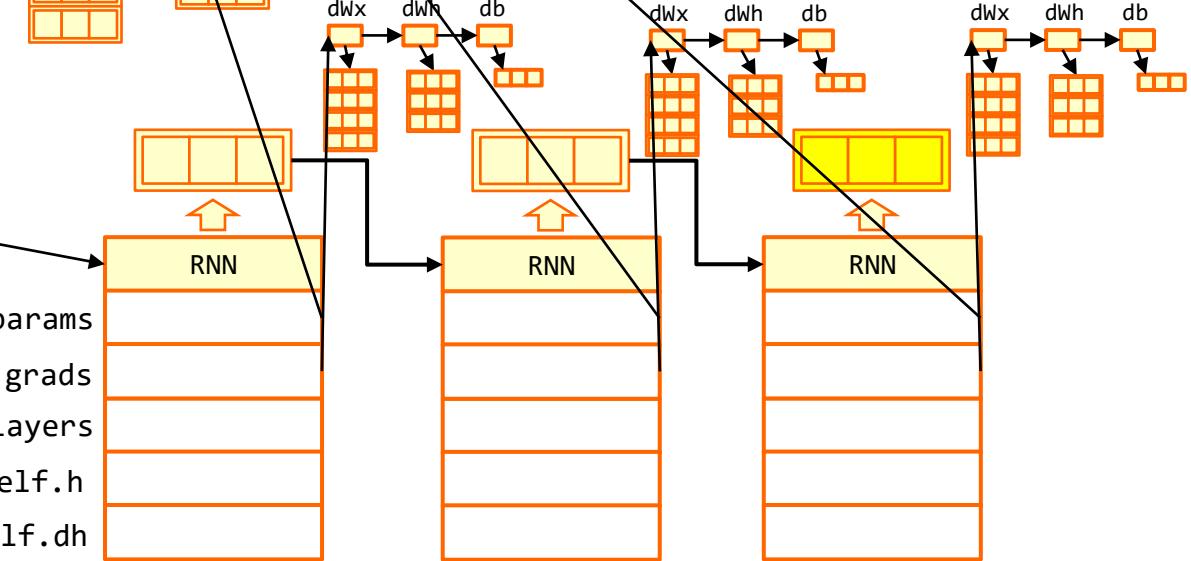
h_{prev}
 $(1, 3)$

self.params
self.grads
self.layers
self.h
self.dh

0 1 2 3

4 5 6 7

8 9 10 11

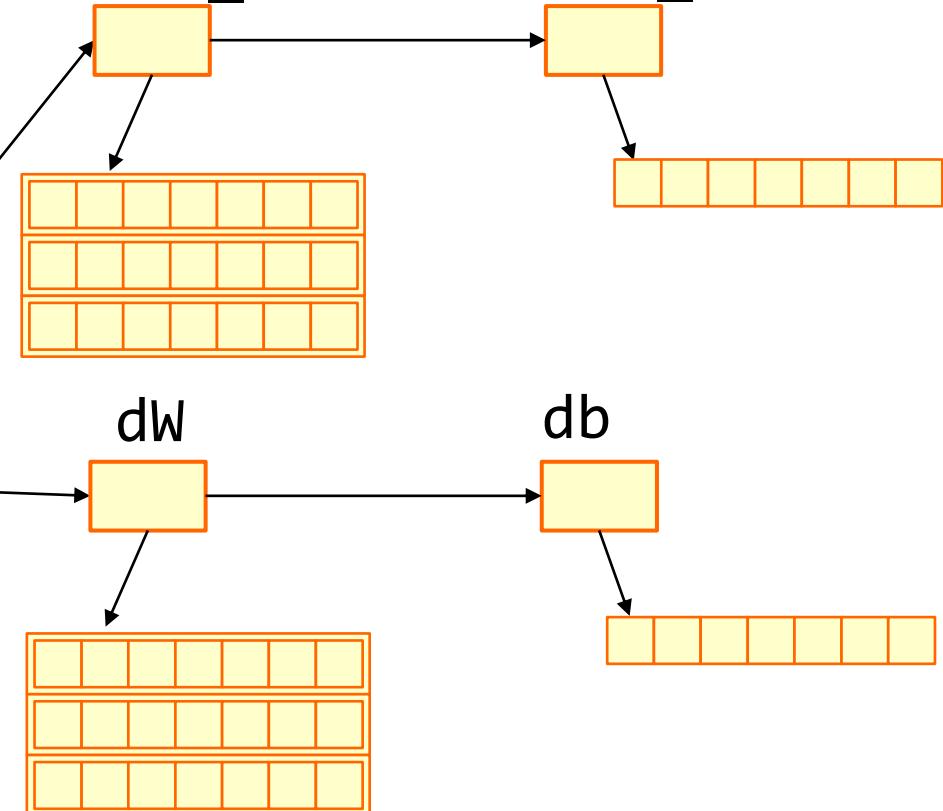


TimeAffine.__init__()

self.params
self.grads
self.x

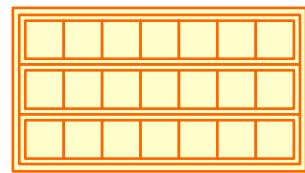
$$(H, V) \Rightarrow (3, 7) \quad (V,) \Rightarrow (7,)$$

affine_w affine_b



TimeAffine.forward(x)

out



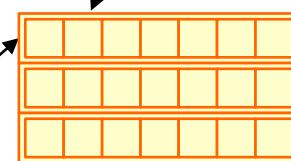
(1, 3, 7)

self.params
self.grads
self.x

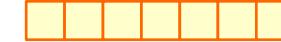
TimeAffine

x
(N, T, H) => (1, 3, 3)

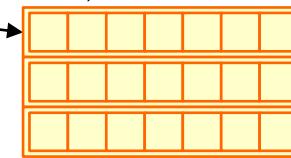
affine_W



affine_b



dW



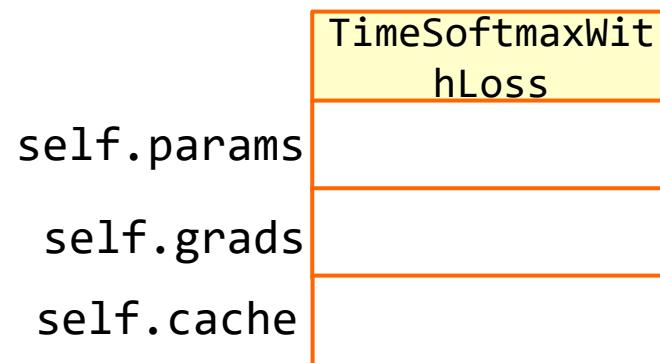
db



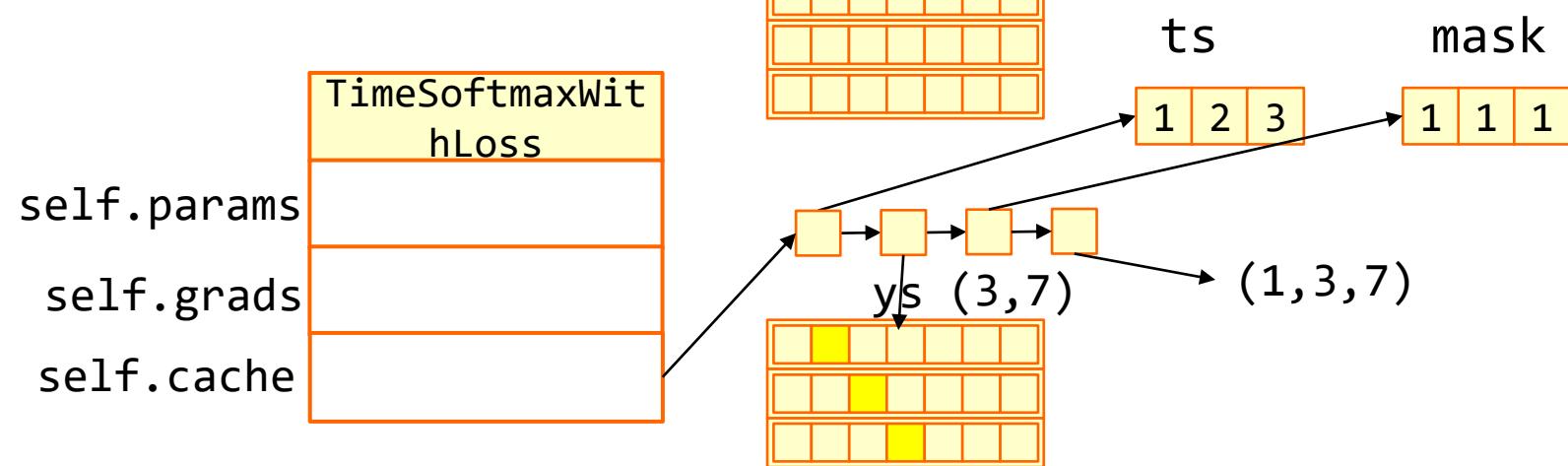
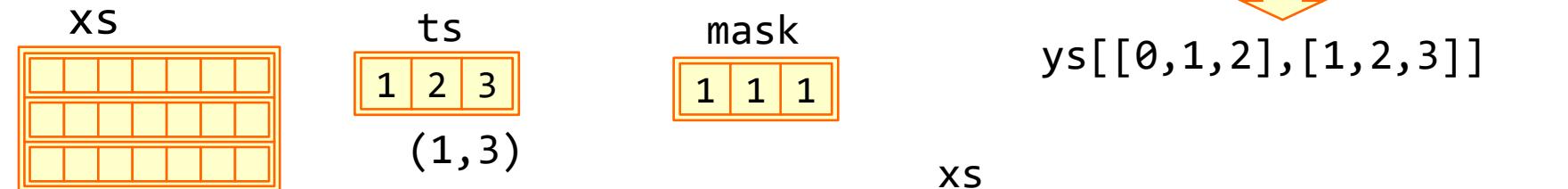
$$rx \begin{pmatrix} & \\ & \\ & \end{pmatrix} \cdot \begin{pmatrix} & \\ & \\ & \end{pmatrix} + \begin{pmatrix} & \\ & \\ & \end{pmatrix} = \begin{pmatrix} & \\ & \\ & \end{pmatrix}$$

$(3, 3)(3, 7) + (7,) \Rightarrow (3, 7)$

TimeSoftmaxWithLoss.__init__()

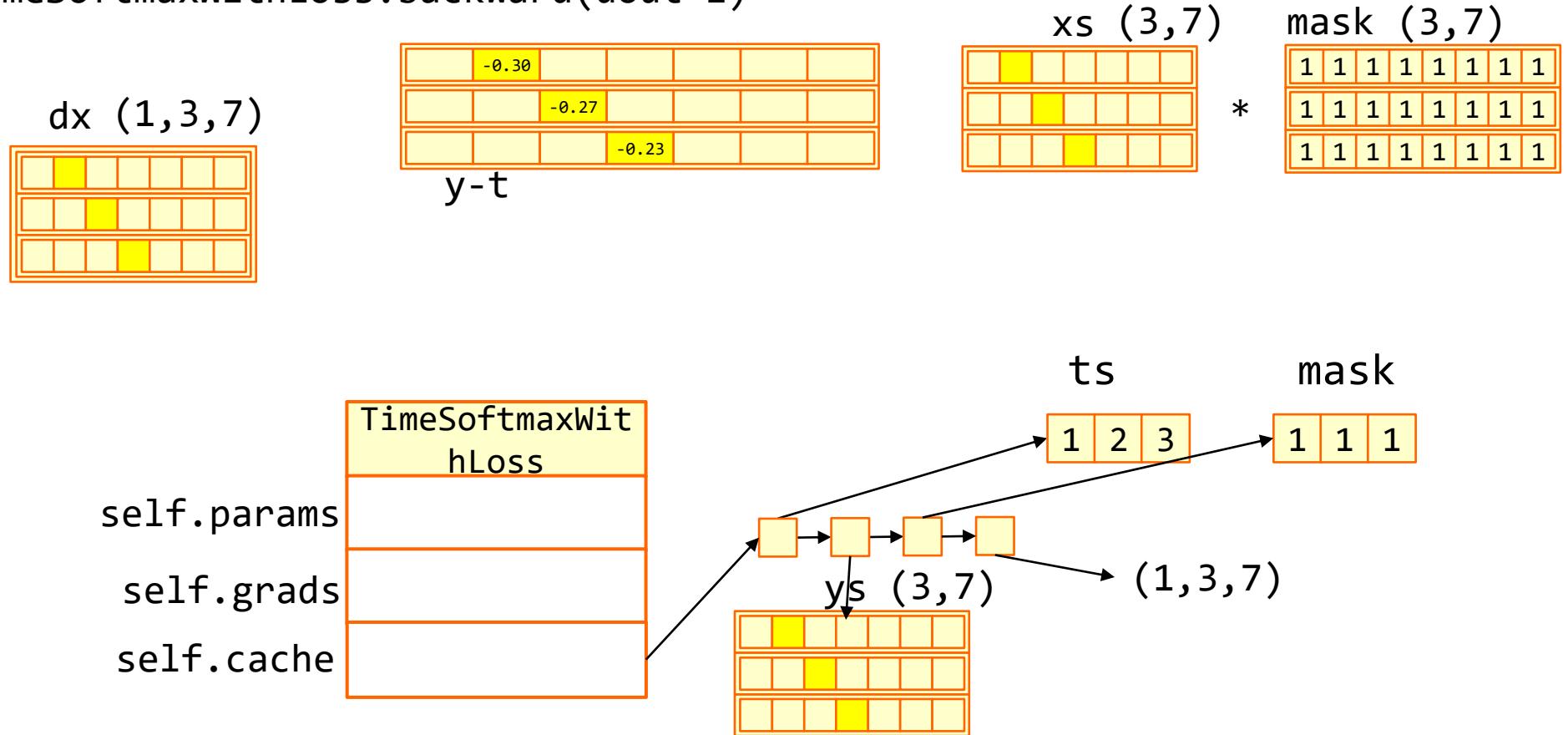


```
TimeSoftmaxWithLoss.forward(xs, ts)
```

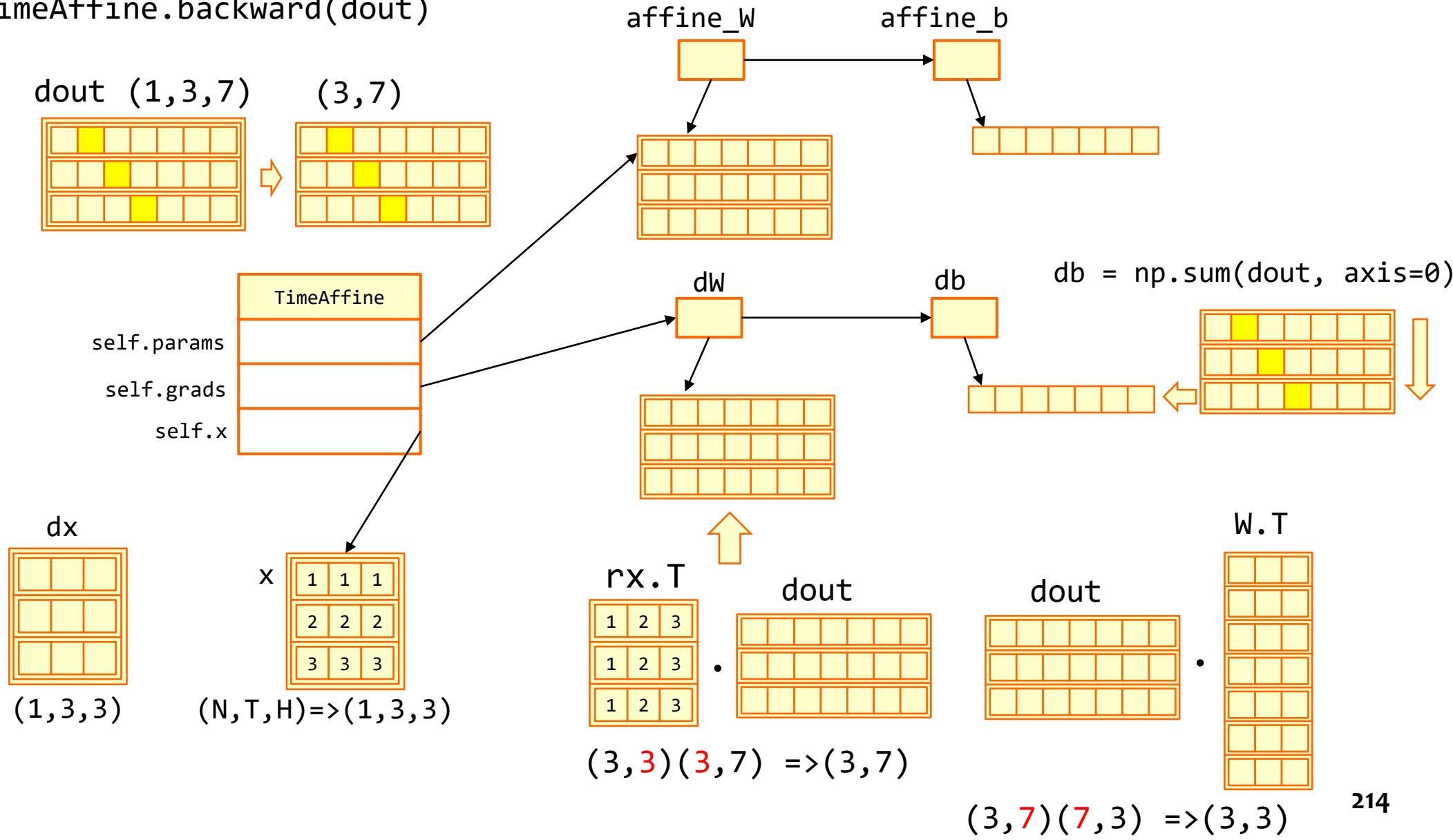


$$\begin{array}{c} \begin{matrix} 0.1 & 0.2 & 0.3 \end{matrix} \\ \begin{matrix} -2.3 & -1.6 & -1.2 \end{matrix} * \begin{matrix} 1 & 1 & 1 \end{matrix} \\ \begin{matrix} 5.1 \end{matrix} / \begin{matrix} 3 \end{matrix} = \begin{matrix} 1.7 \end{matrix} \end{array}$$

TimeSoftmaxWithLoss.backward(dout=1)



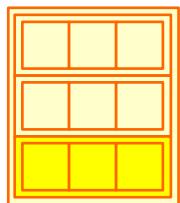
TimeAffine.backward(dout)



TimeRNN.backward(dout)

layer = RNN(*self.params)

dout



(1, 3, 3)

self.params

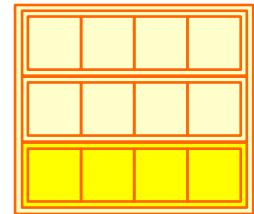
self.grads

self.layers

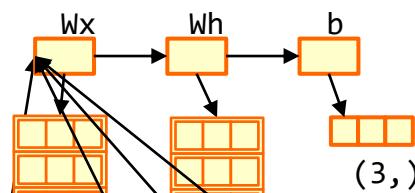
self.h

self.dh

dxs



(N, T, D) => (1, 3, 4)



(4, 3)

(3, 3)

(3,)

dw_x

dwh

db

dw_x

dwh

db

dw_x

dwh

db

dw_x

dwh

db

$+$

$+$

$+$

$+$

$+$

$+$

$+$

RNN

RNN

RNN

self.params

self.grads

self.layers

self.h

self.dh

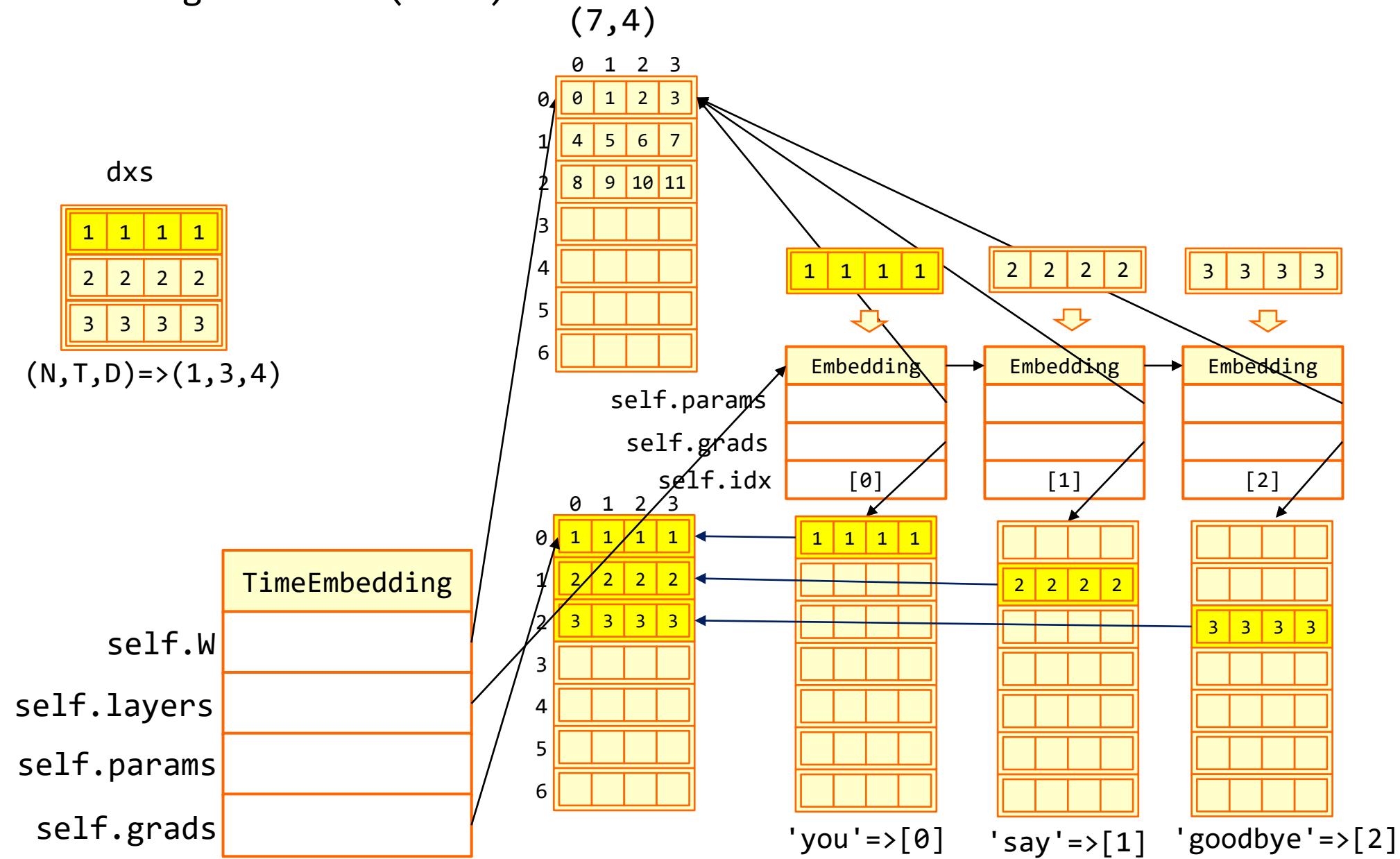
1 1 1 1

2 2 2 2

3 3 3 3

layer

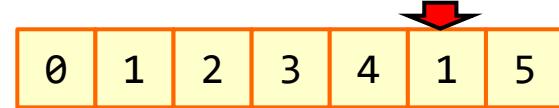
TimeEmbedding.backward(dout)



미니배치 2 소스분석

corpus = [0 1 2 3 4 1 5 6]

xs = corpus[:-1]



ts = corpus[1:]



jump 3

time_idx 2

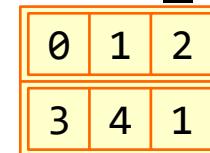
offsets [0 3]

```

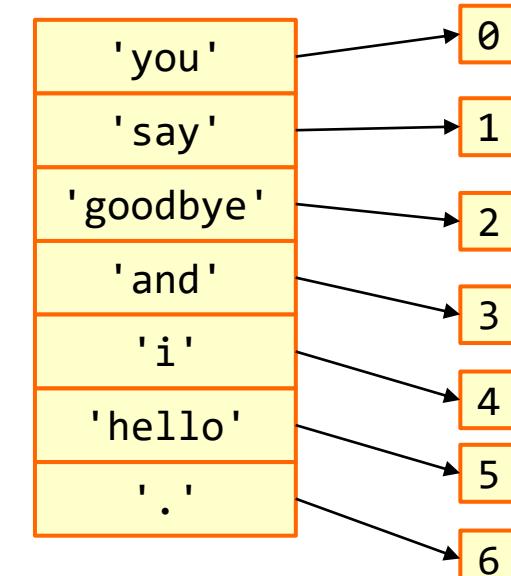
batch_size=2
jump = (corpus_size - 1) // batch_size
offsets = [i * jump for i in range(batch_size)]
for t in range(time_size):
    for i, offset in enumerate(offsets):
        batch_x[i, t] = xs[(offset + time_idx) % data_size]
        batch_t[i, t] = ts[(offset + time_idx) % data_size]
    time_idx += 1
  
```

"you say goodbye and i say hello ."

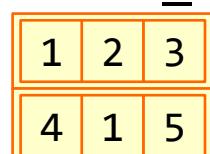
batch_x



word_to_id



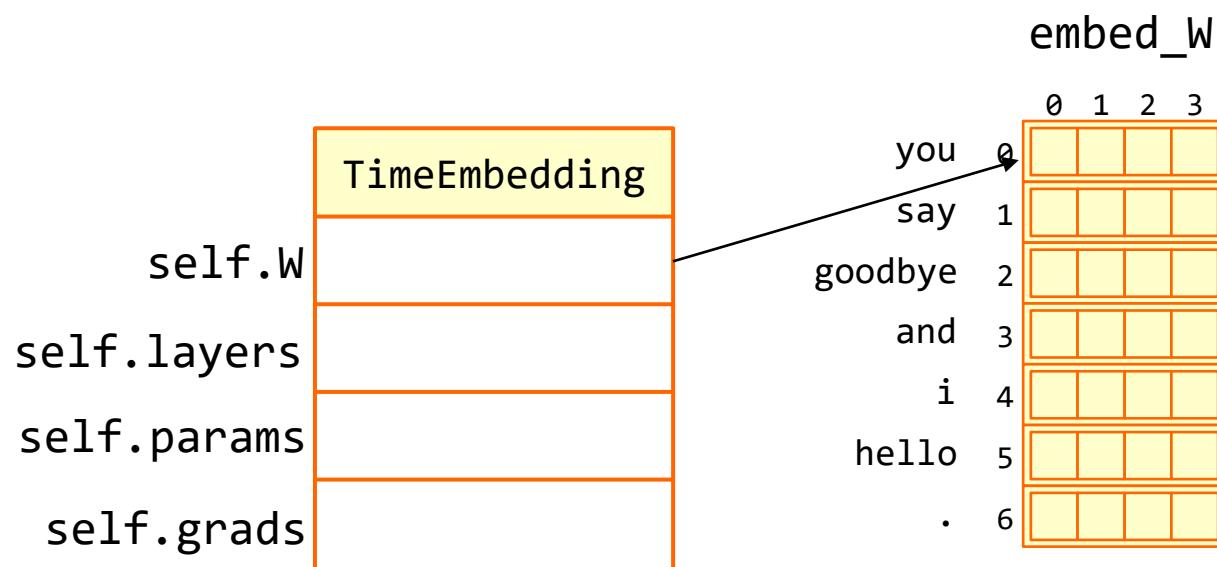
batch_t



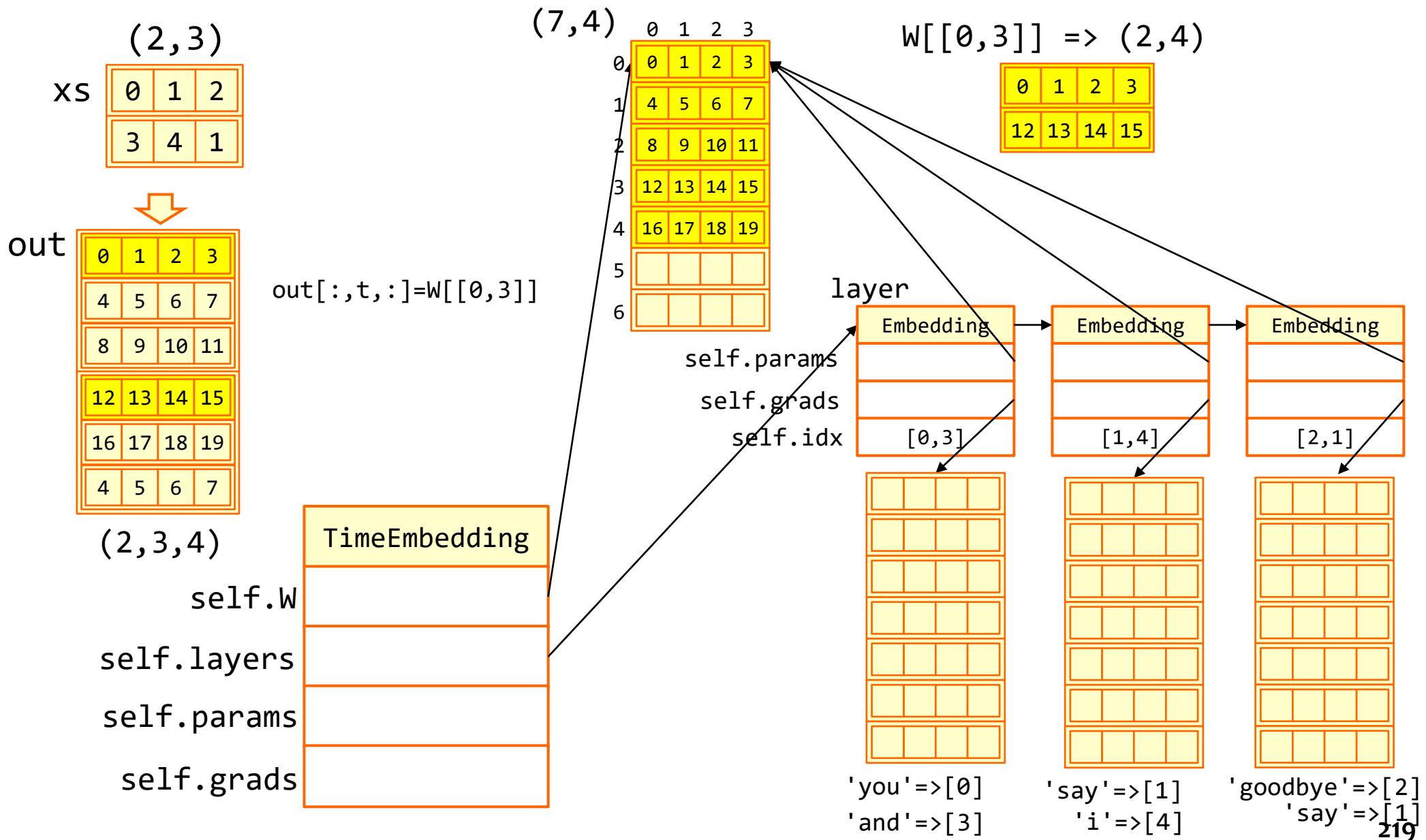
미니배치 2 소스분석

(2,3)

batch_x	batch_t
0 1 2	1 2 3
3 4 1	4 1 5



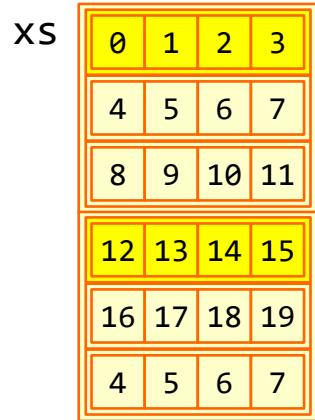
미니배치 2 소스분석



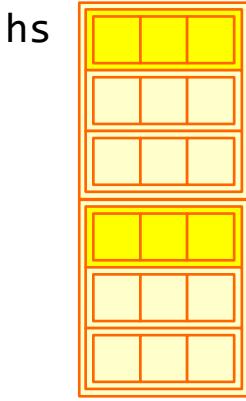
미니배치 2 소스분석

TimeRNN.forward(xs)

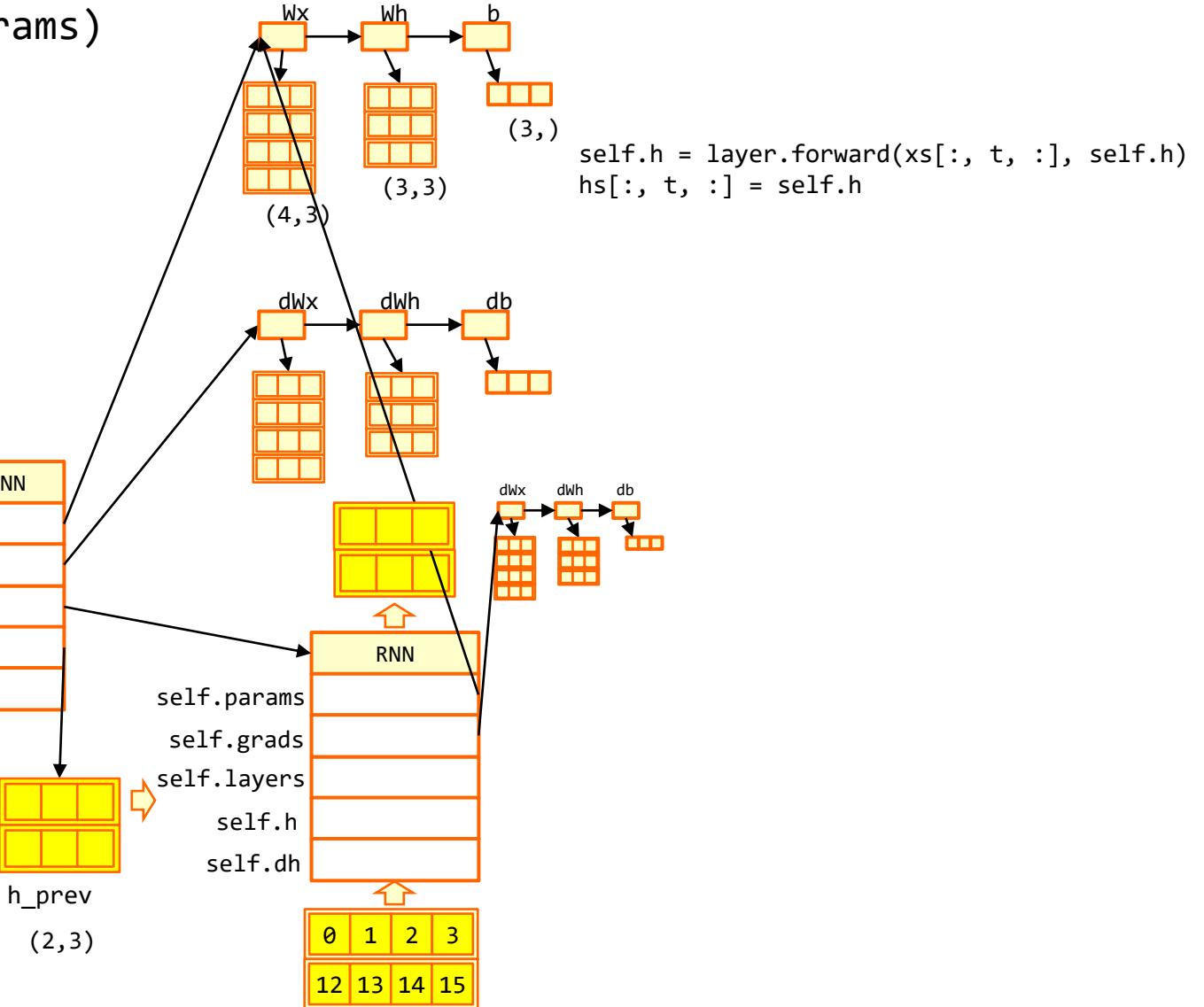
layer = RNN(*self.params)



(N, T, D) => (2, 3, 4)



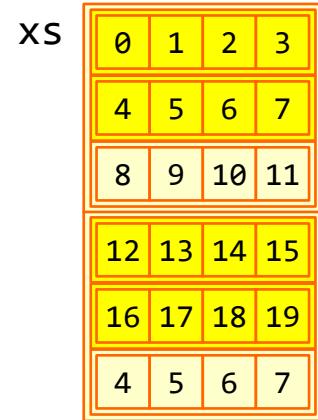
(N, T, H) => (2, 3, 3)



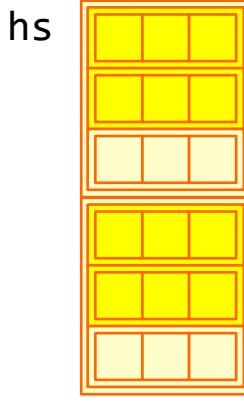
미니배치 2 소스분석

TimeRNN.forward(xs)

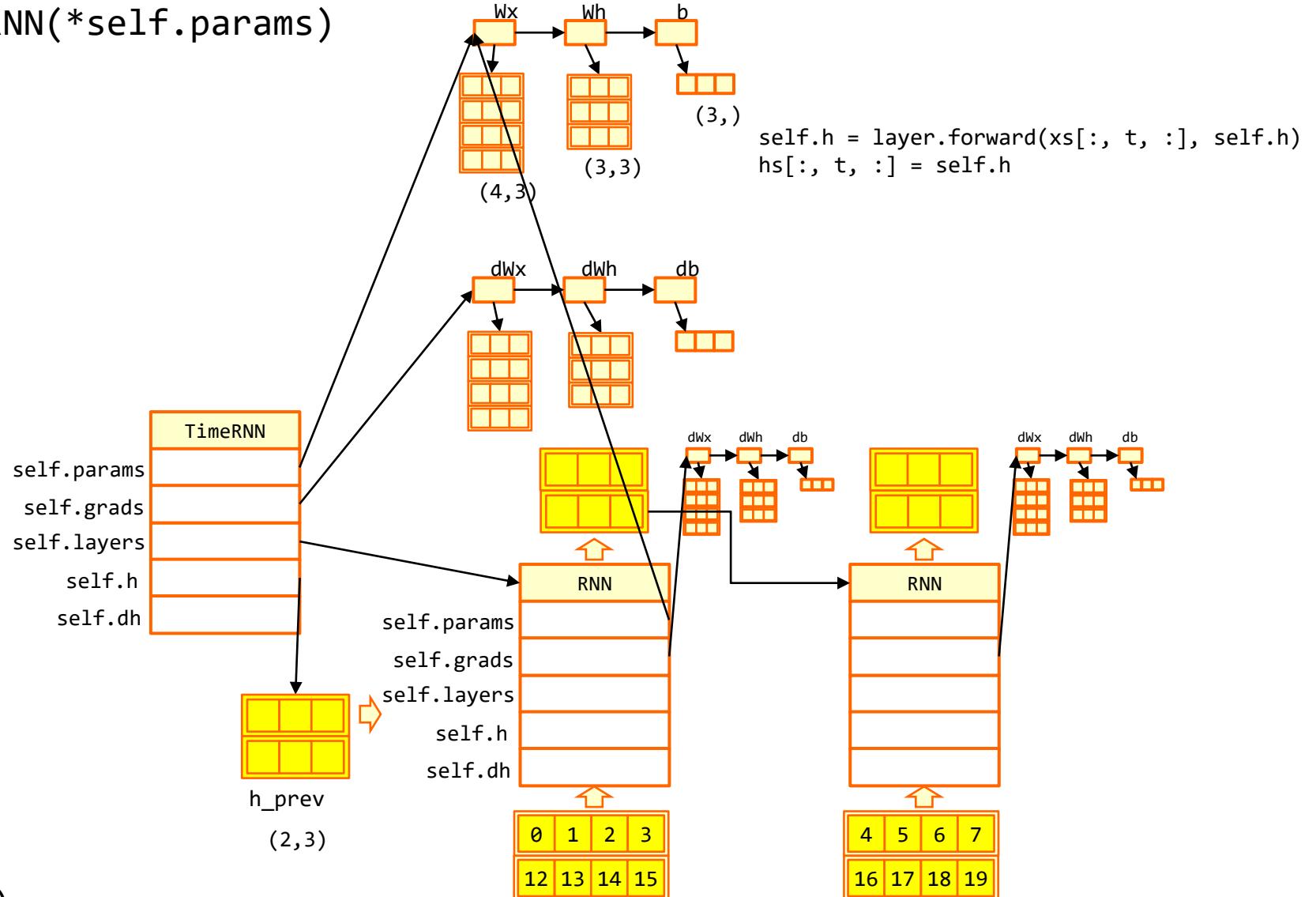
layer = RNN(*self.params)



(N, T, D) => (2, 3, 4)



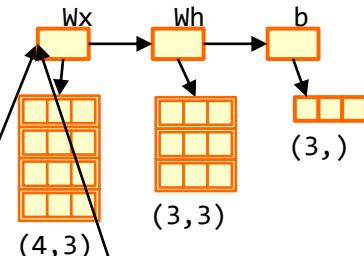
(N, T, H) => (2, 3, 3)



미니배치 2 소스분석

```
TimeRNN.forward(xs)
    layer = RNN(*self.params)
```

xs
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
16 17 18 19
4 5 6 7



```
self.h = layer.forward(xs[:, t, :], self.h)
hs[:, t, :] = self.h
```

$(N, T, D) \Rightarrow (2, 3, 4)$

hs
$(N, T, H) \Rightarrow (2, 3, 3)$

TimeRNN

self.params

self.grads

self.layers

self.h

self.dh

h_{prev}

(2,3)

RNN

self.params

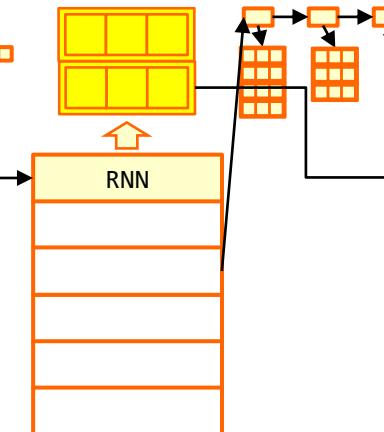
self.grads

self.layers

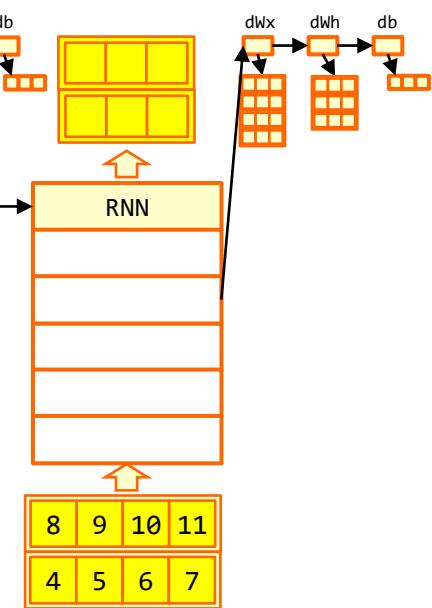
self.h

self.dh

0 1 2 3
12 13 14 15



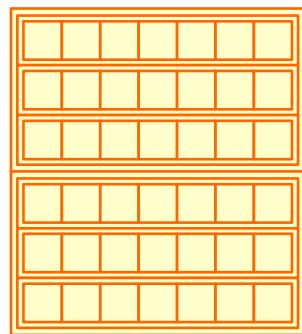
4 5 6 7
16 17 18 19



8 9 10 11
4 5 6 7

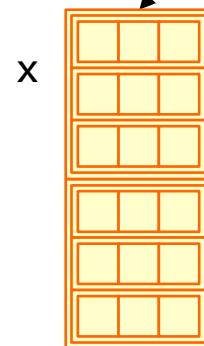
미니배치 2 소스분석

TimeAffine.forward(x)



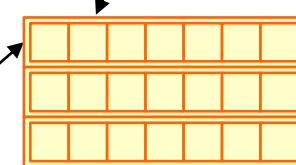
out
(2, 3, 7)

self.params
self.grads
self.x

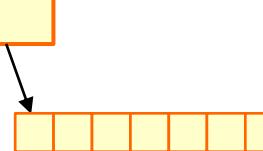


(N, T, H) => (2, 3, 3)

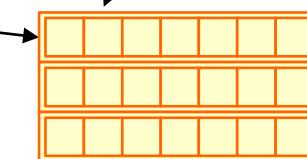
affine_W



affine_b



dW



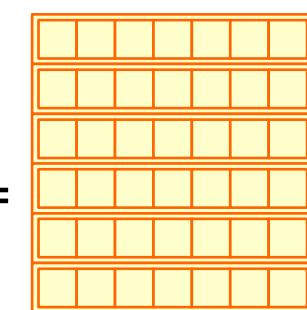
db



rx

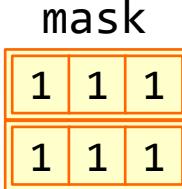
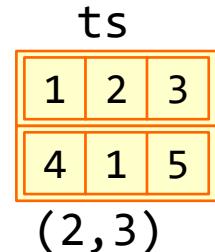
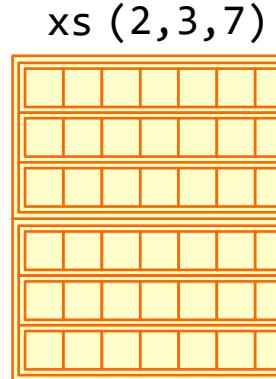


$(6, 3)(3, 7) + (7,) \Rightarrow (6, 7)$



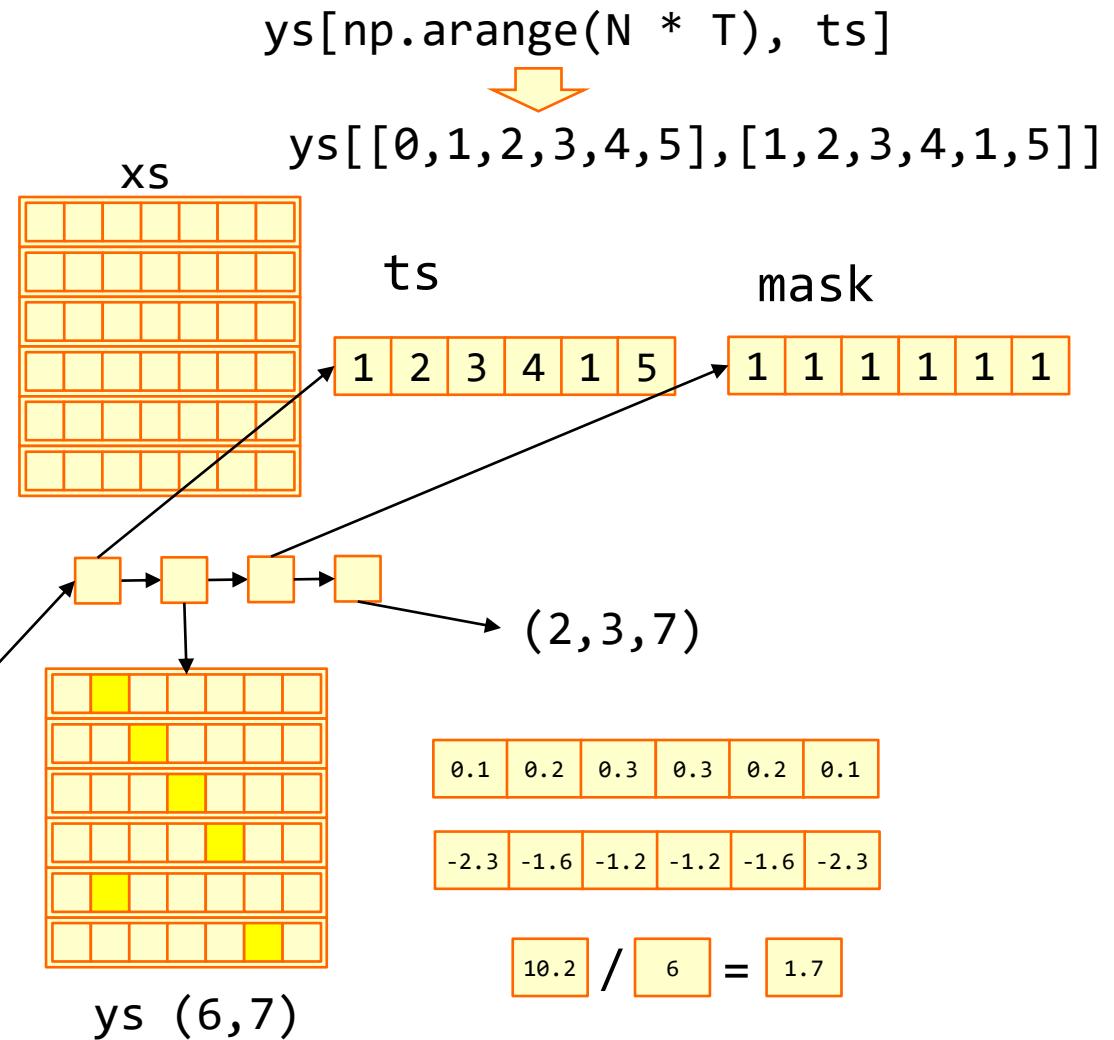
미니배치 2 소스분석

TimeSoftmaxWithLoss.forward(xs, ts)



self.params
self.grads
self.cache

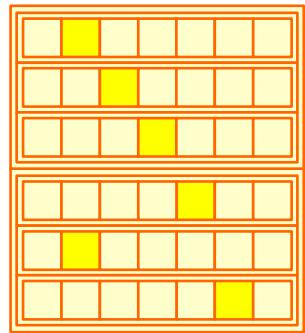
TimeSoftmaxWithLoss



미니배치 2 소스분석 (backward)

TimeSoftmaxWithLoss.forward(xs, ts)

dx (2,3,7)



ts

1	2	3
4	1	5

(2,3)

mask

1	1	1
1	1	1

self.params

TimeSoftmaxWithLoss

self.grads

self.cache

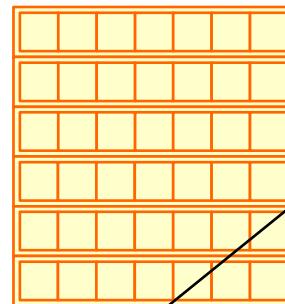
0.1 0.2 0.3 0.3 0.2 0.1

-0.9 -0.8 -0.7 -0.7 -0.8 -0.9

-0.9 -0.8 -0.7 -0.7 -0.8 -0.9

ys[np.arange(N * T), ts]

xs



ts

1 2 3 4 1 5

mask

1 1 1 1 1 1 0

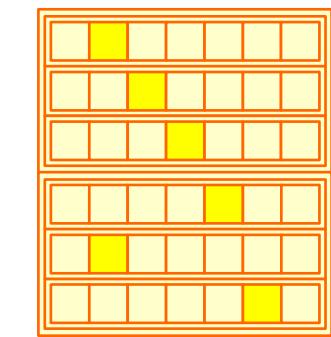
(6,7)(6,1)

ys (6,7)

dx *= mask[:, np.newaxis]

미니배치 2 소스분석 (backward)

TimeAffine.forward(x)



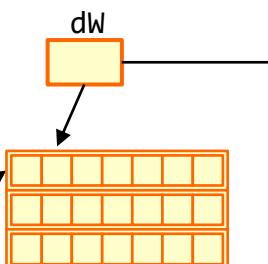
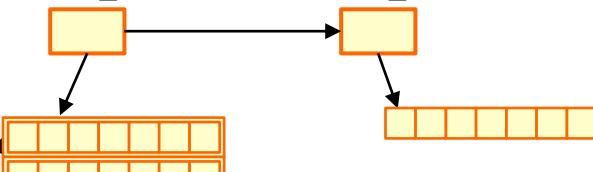
dout
(2, 3, 7)

self.params
self.grads
self.x

dx
(2, 3, 3)

x
(N, T, H) => (2, 3, 3)

affine_W affine_b

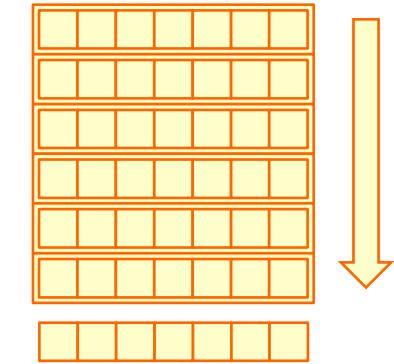


rx.T

(3, 6)

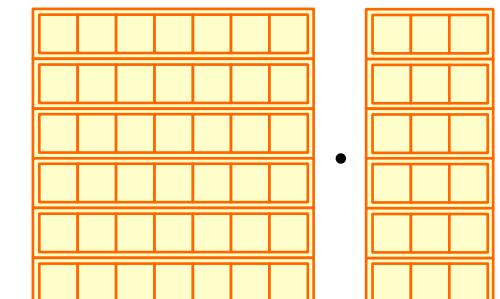
dw = np.dot(rx.T, dout)

dout



db = np.sum(dout, axis=0)

dout



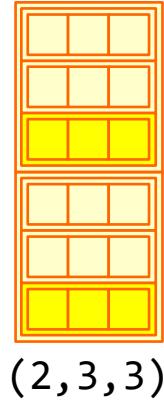
(6, 7)(7, 3)

dx = np.dot(dout, W.T)

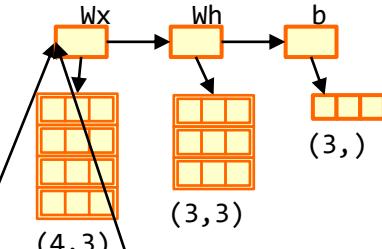
미니배치 2 소스분석 (backward)

```
TimeRNN.forward(xs)
    layer = RNN(*self.params)
```

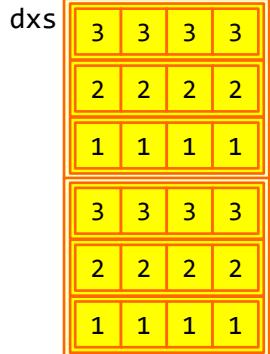
dhs



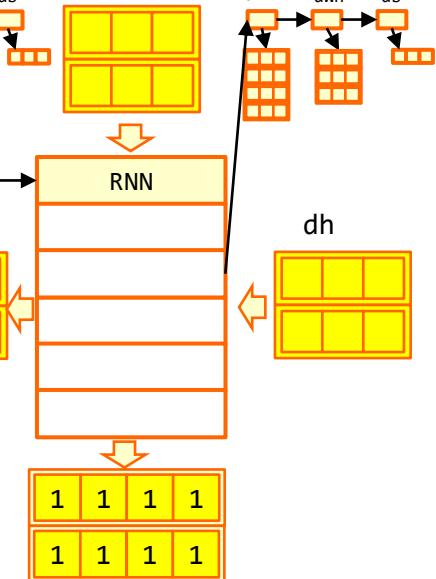
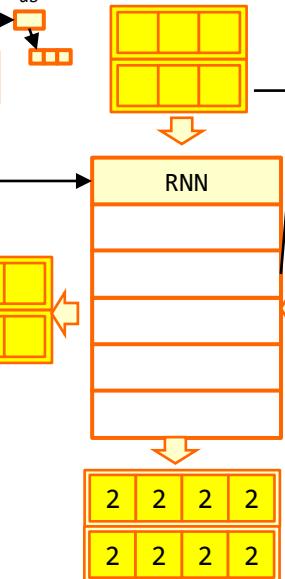
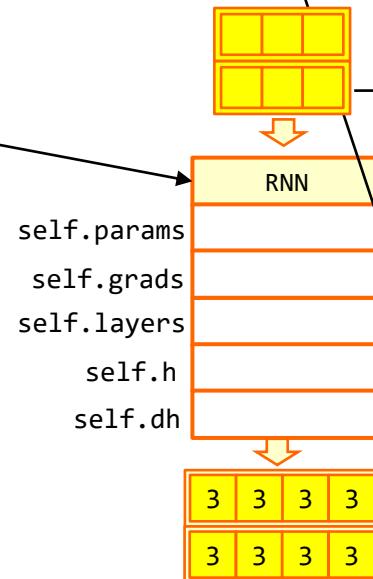
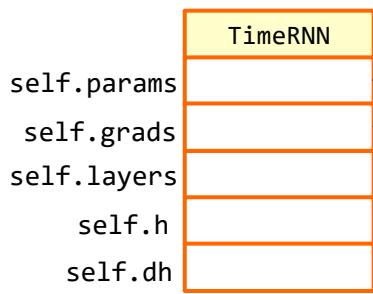
(2, 3, 3)



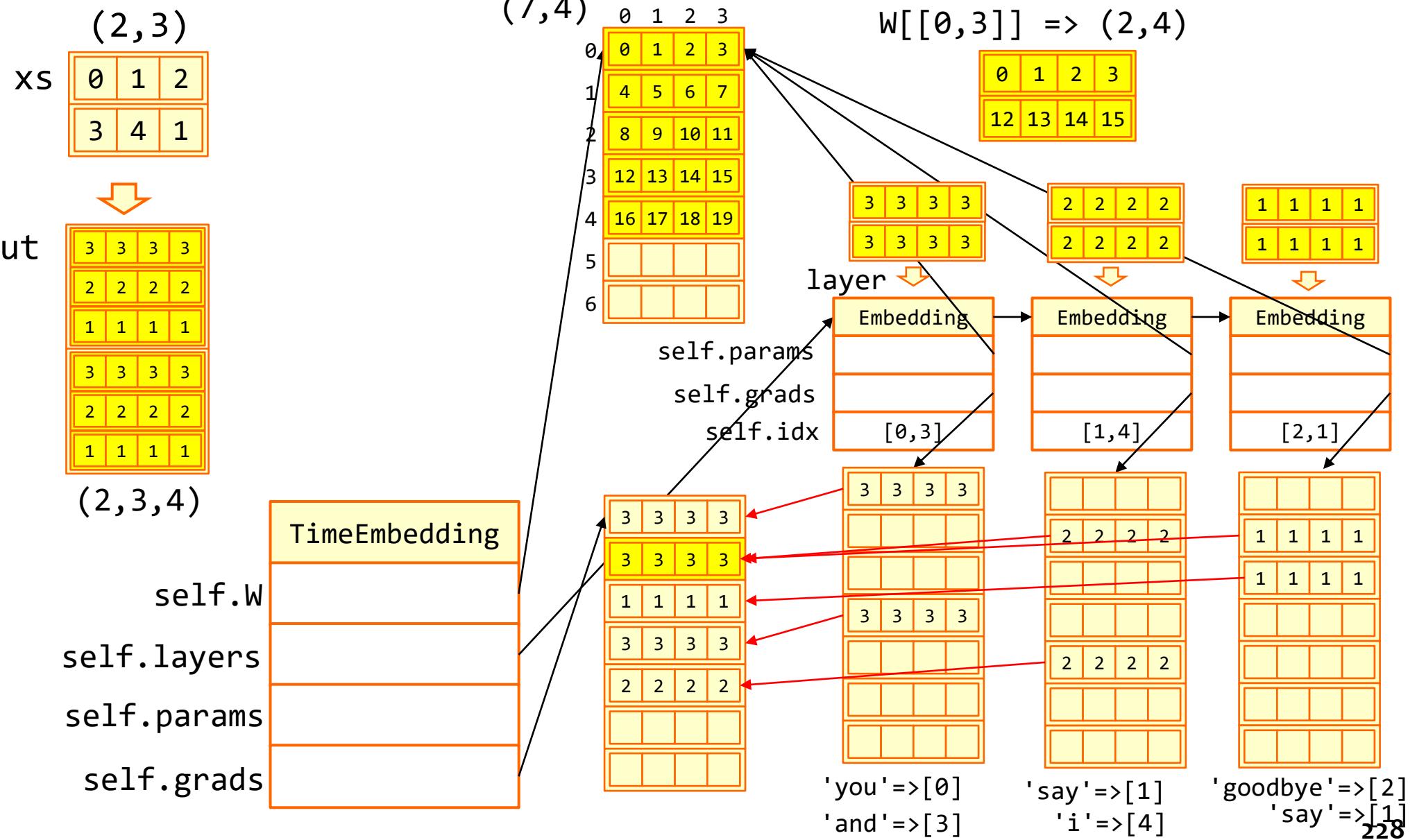
```
self.h = layer.forward(xs[:, t, :], self.h)
hs[:, t, :] = self.h
```



(N, T, D) => (2, 3, 4)

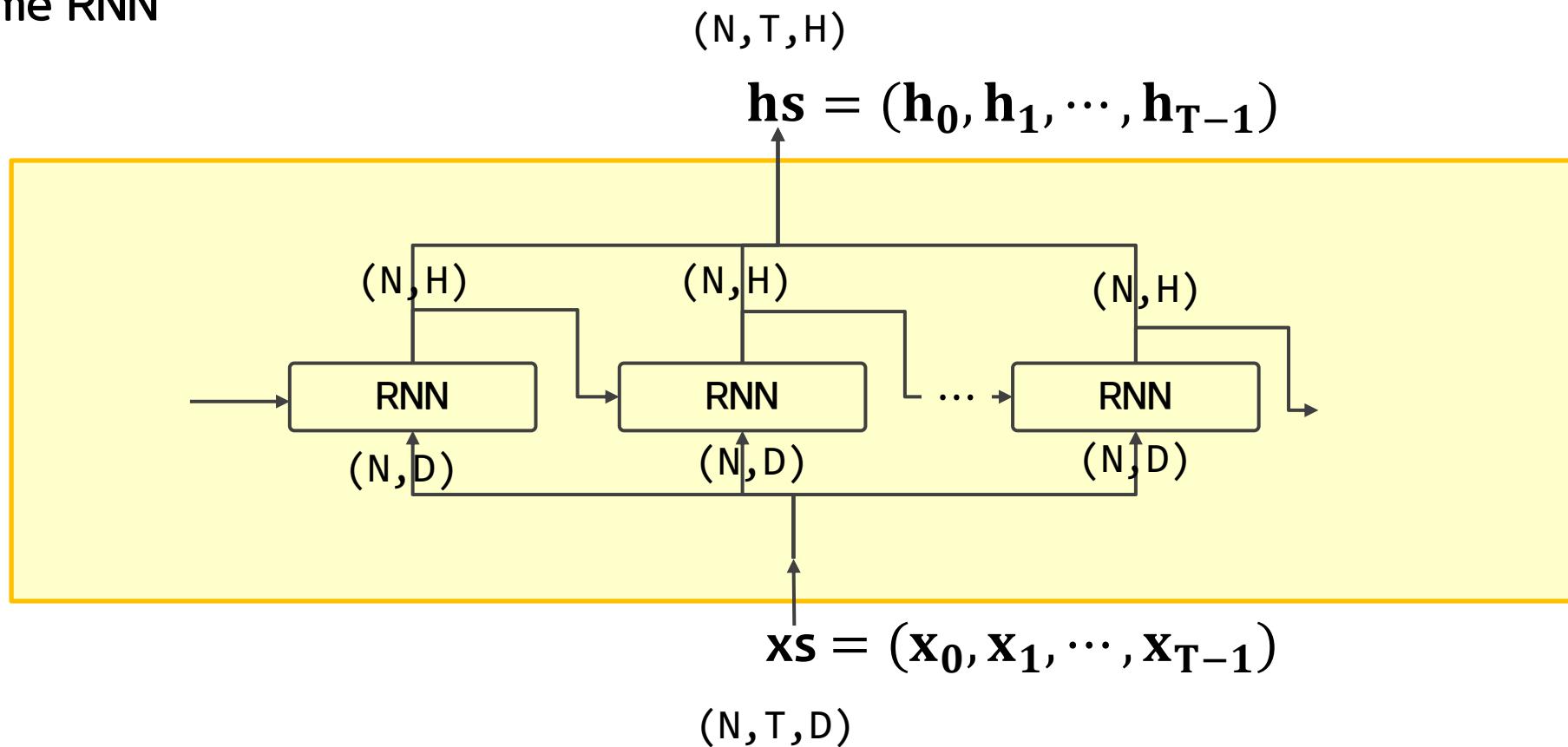


미니배치 2 소스분석 (backward)

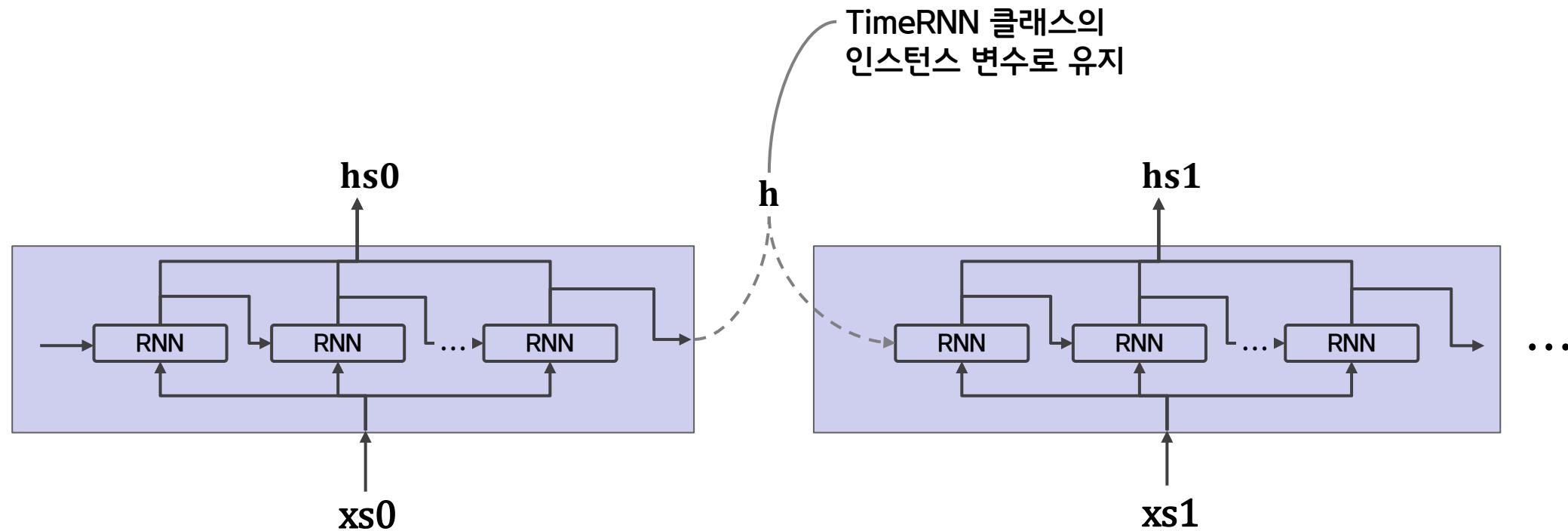


Time RNN 계층의 계산 그래프

Time RNN

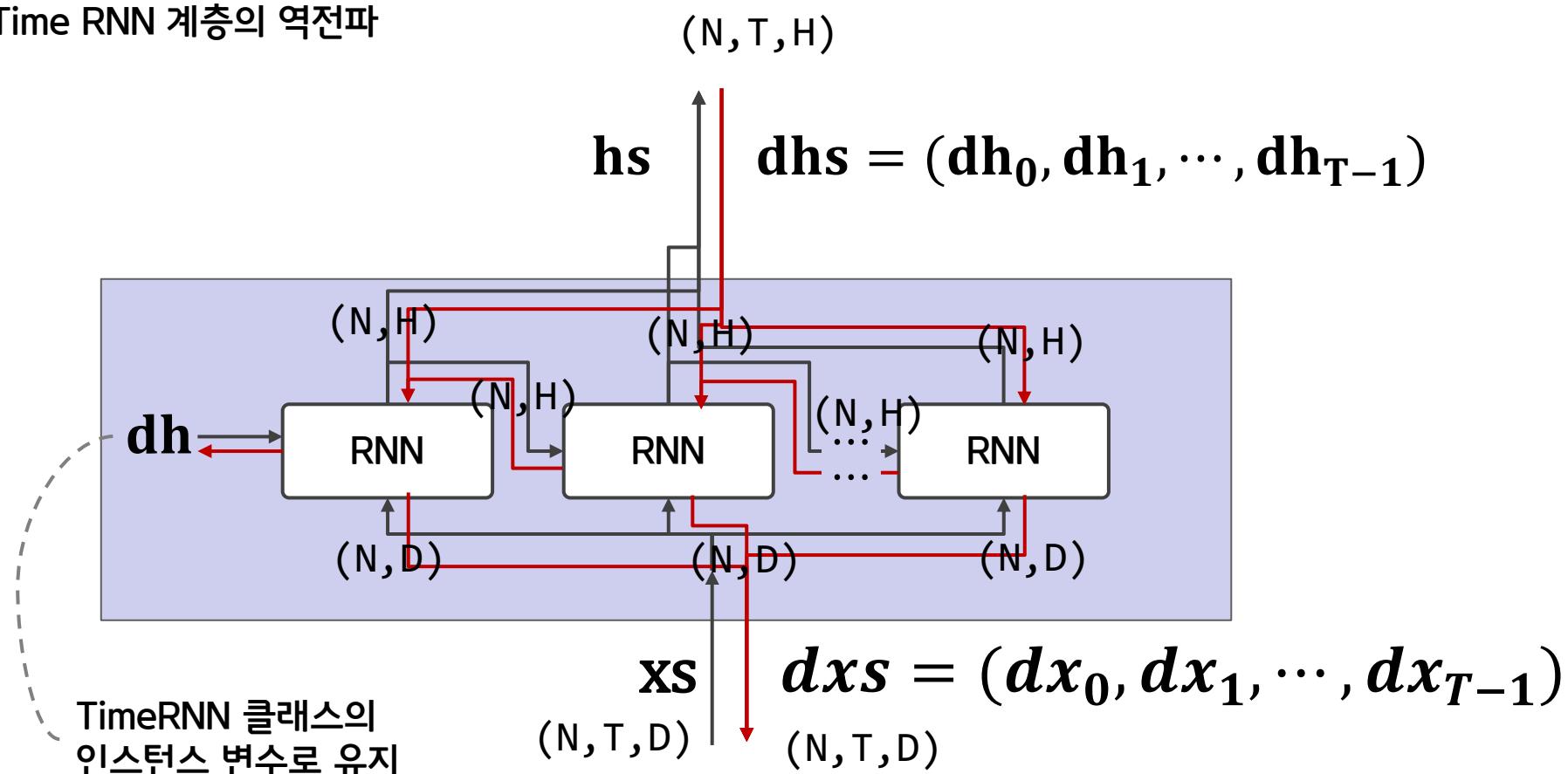


Time RNN 계층은 은닉 상태를 인스턴스 변수 h 로 보관한다. 그러면 은닉 상태를 다음 블록에 인계할 수 있다.



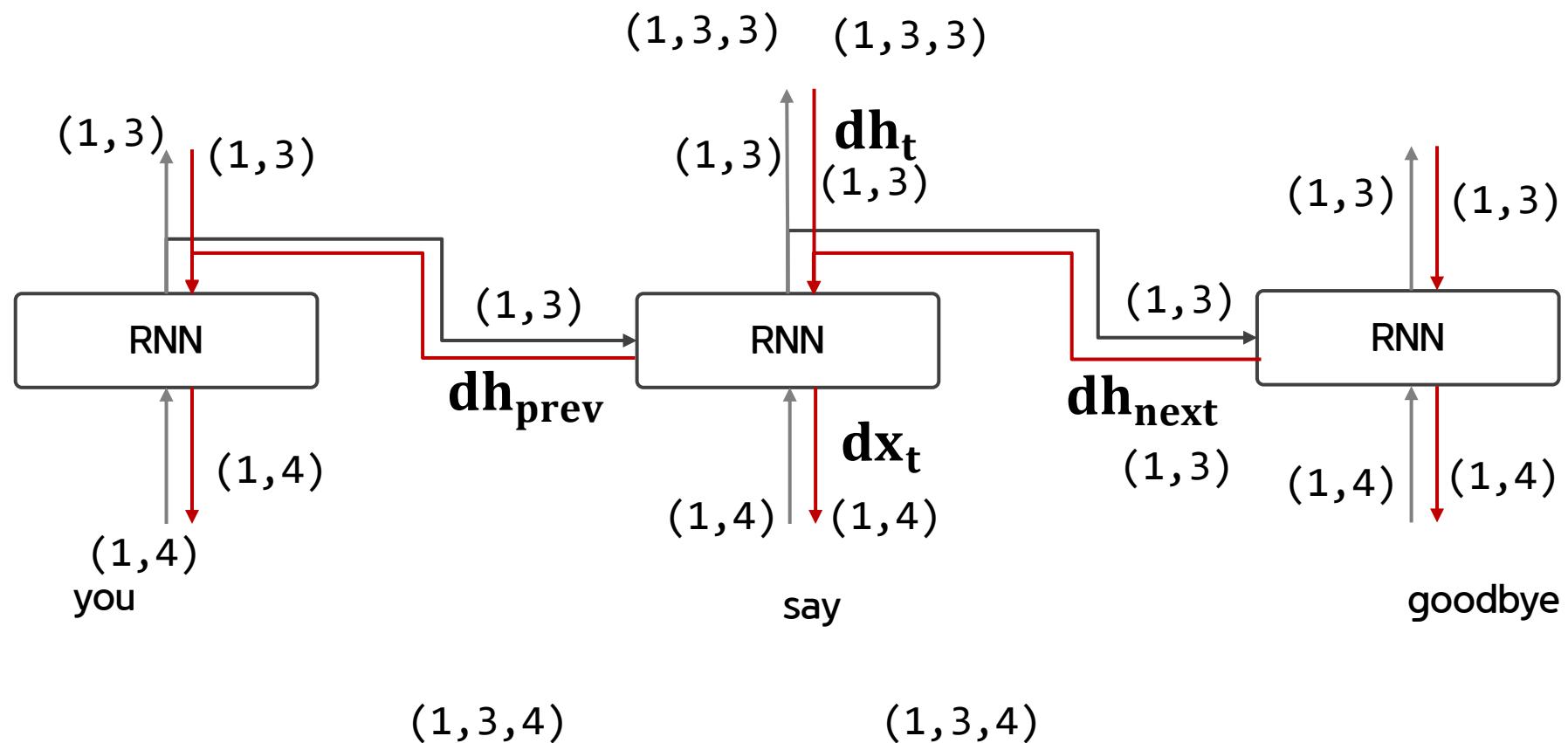
RNN계층의 은닉 상태를 Time RNN계층에서 관리하면 Time RNN사용자는 RNN계층 사이에서 은닉 상태를 '인계하는 작업'을 생각하지 않아도 된다.

Time RNN 계층의 역전파



t번째 RNN 계층의 역전파

Time RNN



5. 순환 신경망(RNN)

5.1 확률과 언어 모델

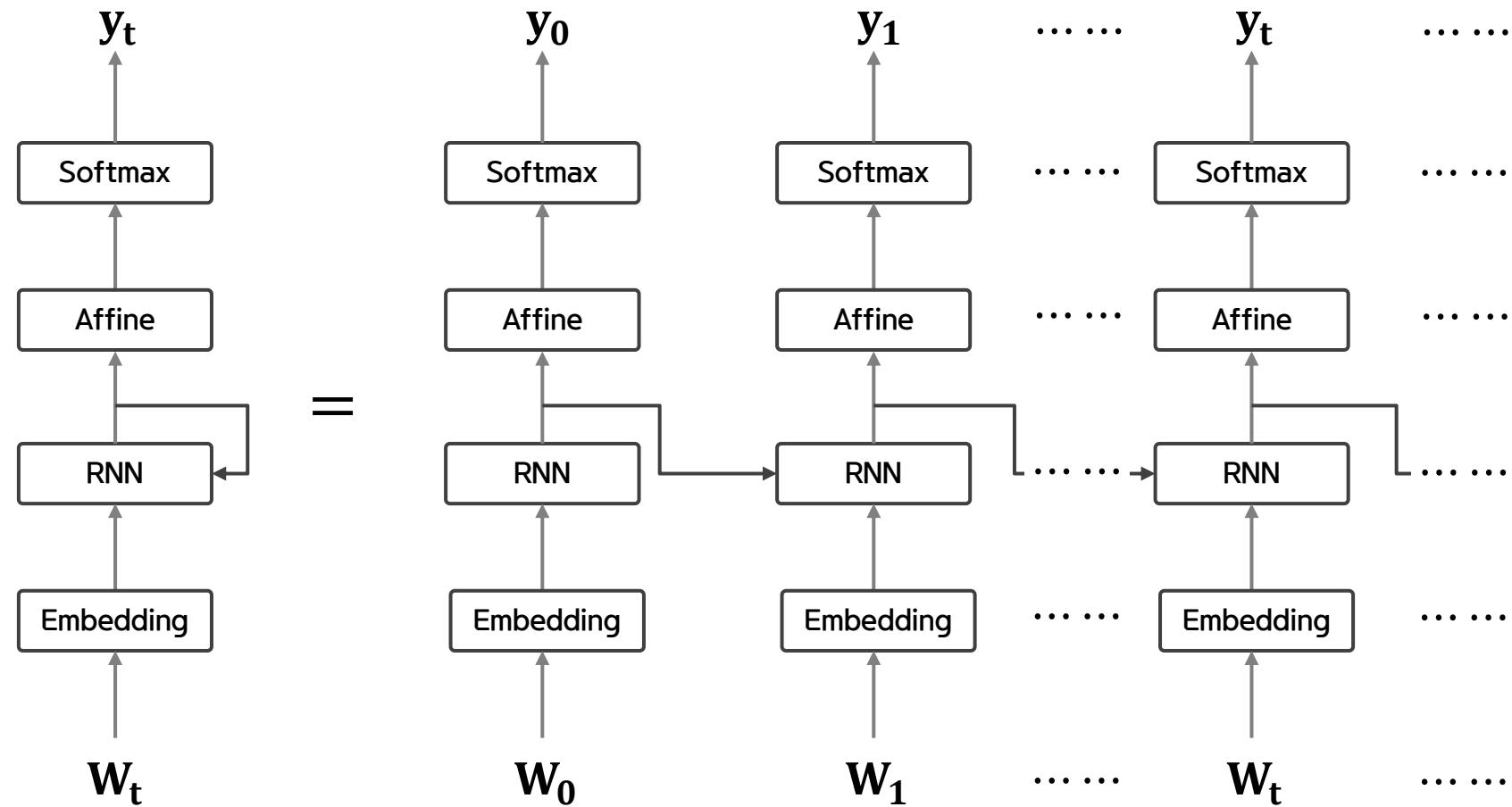
5.2 RNN 이란

5.3 RNN 구현

5.4 시계열 데이터 처리 계층 구현

5.5 RNNLM 학습과 평가

RNNLM의 신경망



RNNLM의 신경망

Embedding: 단어 ID를 단어의 분산 표현(단어 벡터)으로 변환

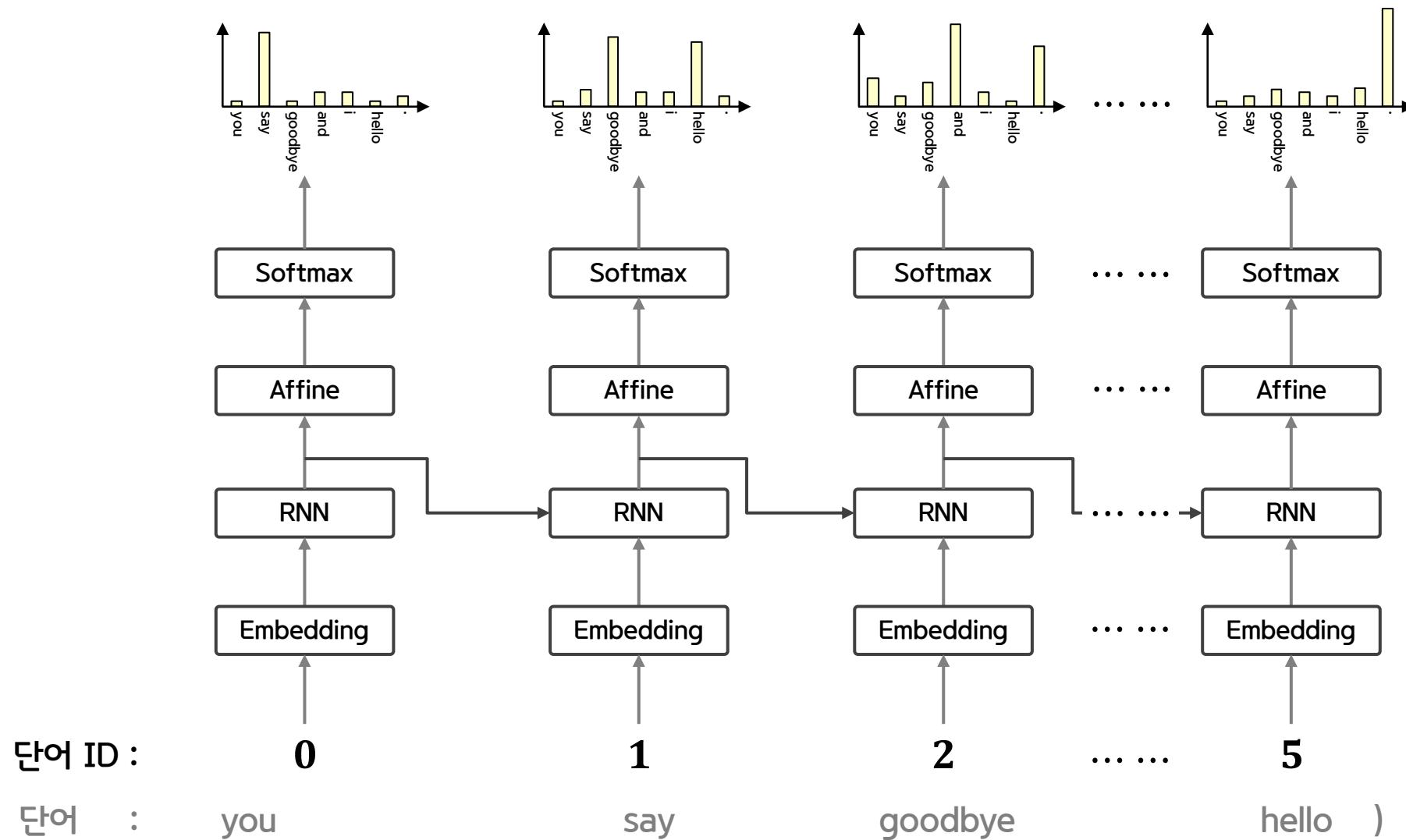
RNN계층: 은닉 상태를 다음 층으로(위쪽으로) 출력함과 동시에 다음 시각의 RNN 계층으로(오른쪽으로) 출력한다.

RNN계층이 위로 출력한 은닉 상태는 Affine 계층을 거쳐 Softmax 계층으로 전해진다.

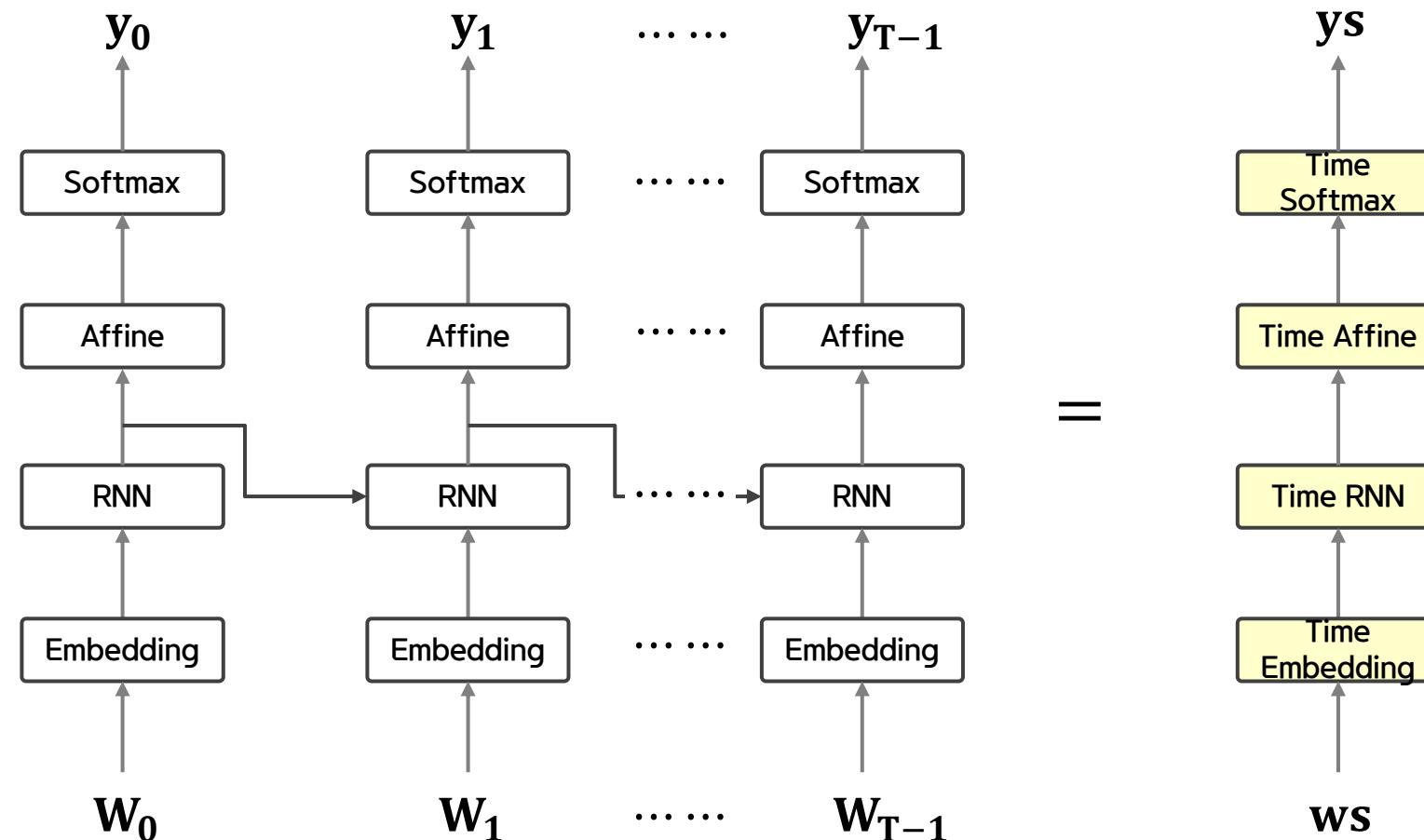
RNNLM은 지금까지 입력된 단어를 '기억'하고 그것을 바탕으로 다음에 출현할 단어를 예측한다.

RNN계층이 과거에서 현재로 데이터를 계속 흘려 보내 줌으로써 과거의 정보를 인코딩해 저장(기억) 할 수 있다.

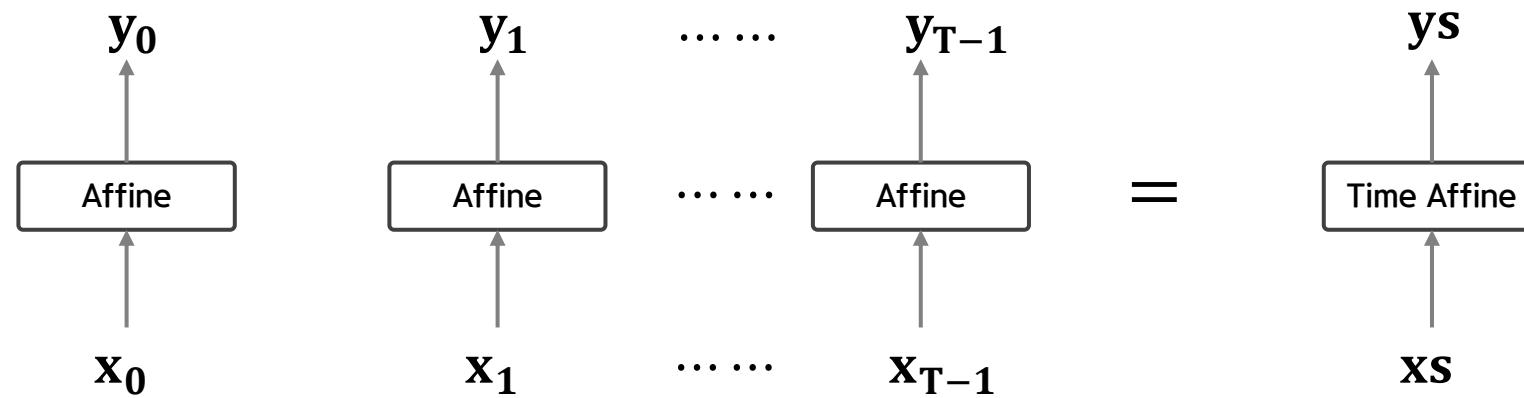
샘플 말뭉치로 "you say goodbye and i say hello."를 처리하는 RNNLM의 예



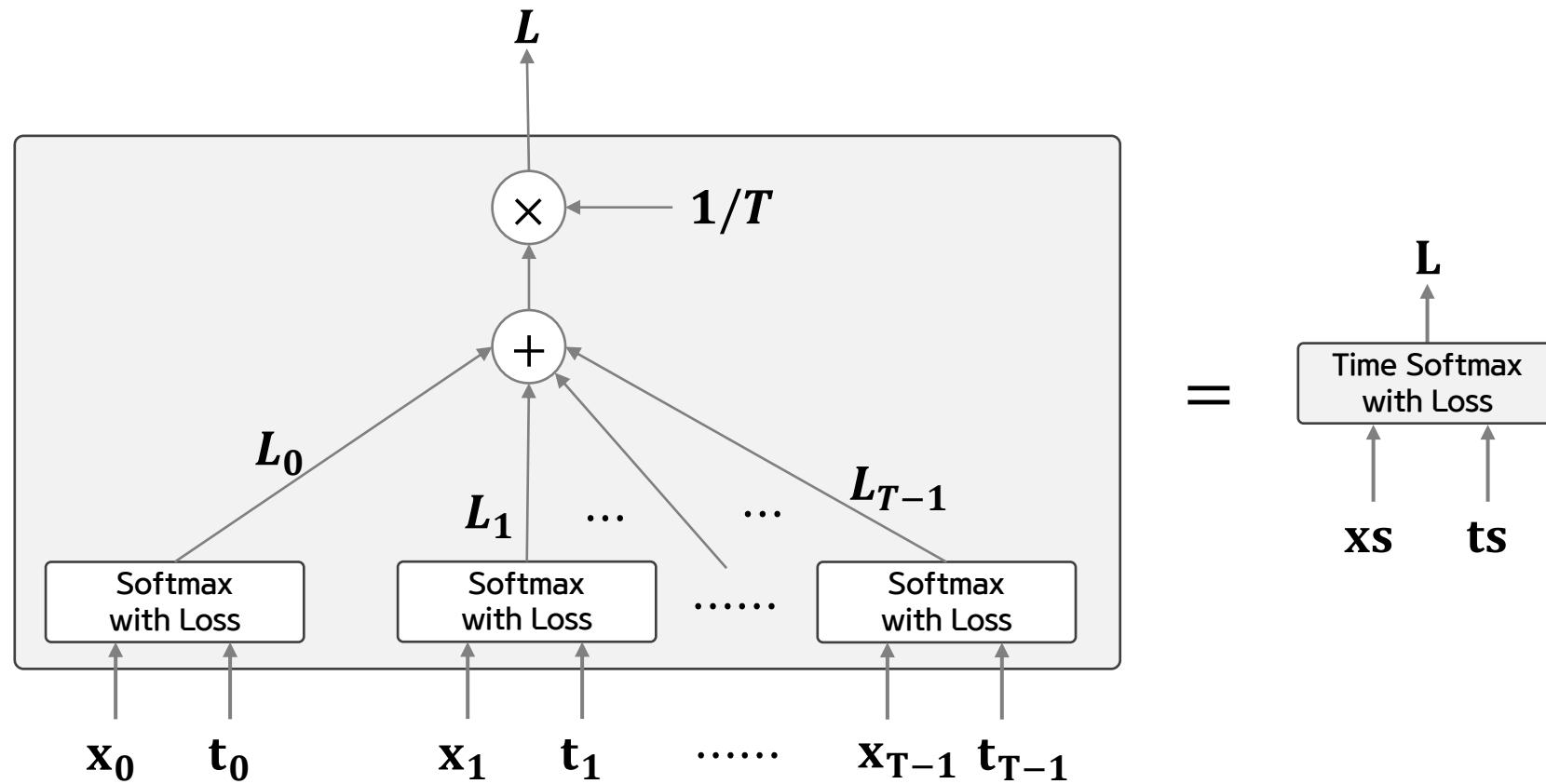
시계열 데이터를 한꺼번에 처리하는 계층을 Time XX 계층으로 구현



Time Affine 계층은 T개의 Affine 계층의 집합으로 구현



Time Softmax with Loss 계층의 전체 그림



x_0, x_1, \dots : 아래층에서부터 전해지는 점수(확률로 정규화 되기 전의 값)

t_0, t_1, \dots : 정답 레이블

T개의 Softmax with Loss 계층 각각이 손실을 산출하고 그 손실들을 합산해 평균한 값이 최종 손실이 된다.

$$L = \frac{1}{T} (L_0 + L_1 + \cdots + L_{T-1})$$

Time Softmax with Loss 계층도 시계열에 대한 평균을 구하는 것으로 데이터 1개당 평균 손실을 구해 최종 출력으로 내보낸다.

5. 순환 신경망(RNN)

5.1 확률과 언어 모델

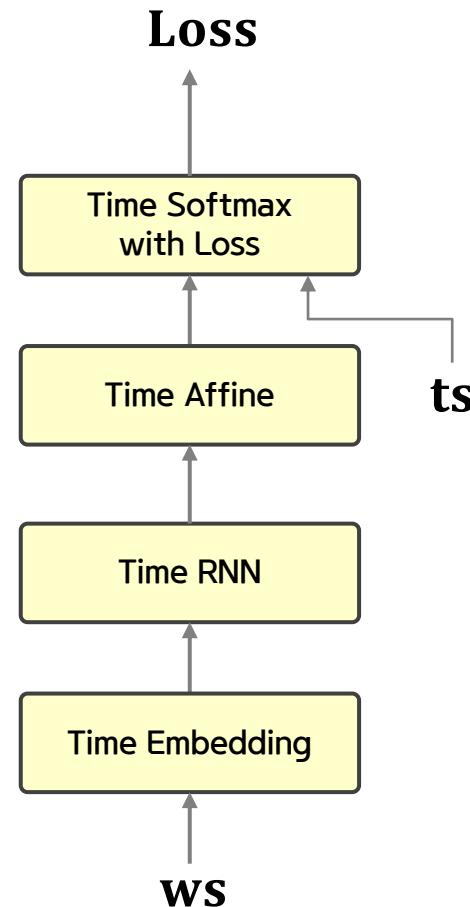
5.2 RNN 이란

5.3 RNN 구현

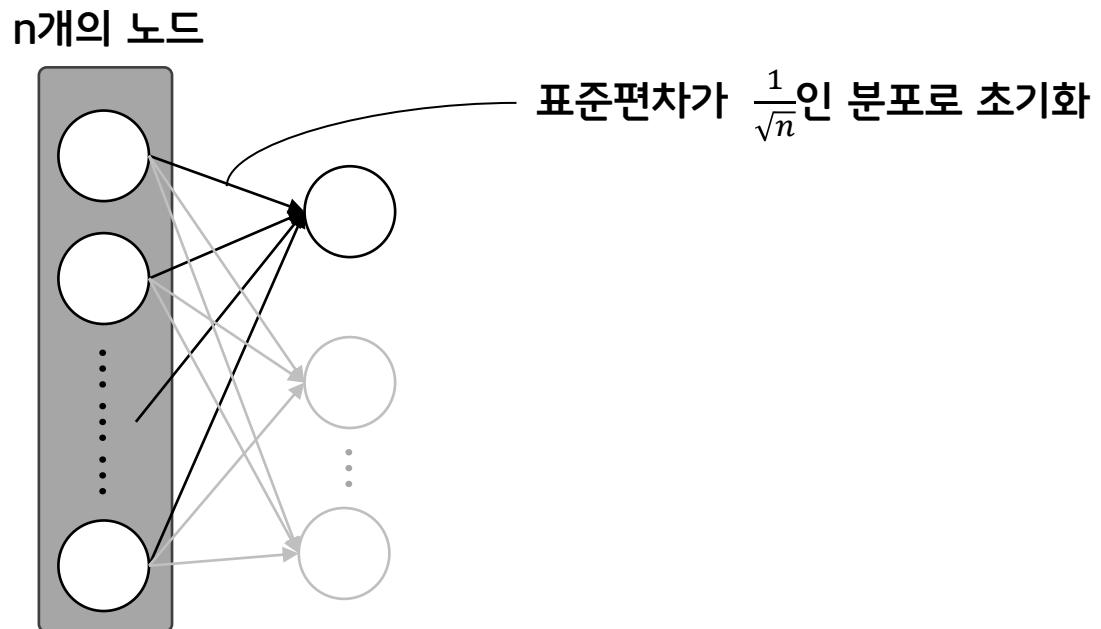
5.4 시계열 데이터 처리 계층 구현

5.5 RNNLM 학습과 평가

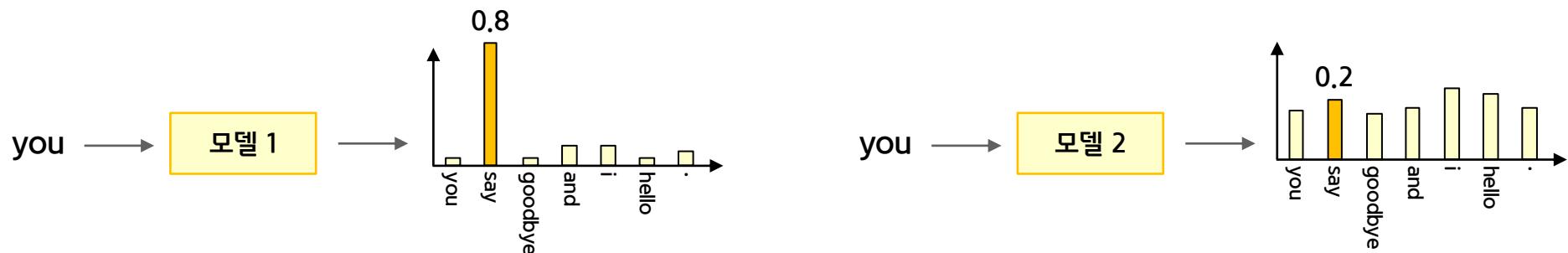
SimpleRnnlm의 계층 구성 : RNN 계층의 상태는 클래스 내부에서 관리



Xavier 초기값 : 이전 계층의 노드가 n 개라면 표준편차가 $\frac{1}{\sqrt{n}}$ 인 분포를 초기값으로 사용



단어 "you"를 입력하여 다음에 출현할 단어의 확률분포를 출력하는 모델의 예



언어 모델은 주어진 과거 단어(정보)로부터 다음에 출현할 단어의 확률분포를 출력한다.
이때 언어 모델의 예측 성능을 평가하는 척도로 혼란도(perplexity)를 자주 이용한다.

혼란도(perplexity) : 간단히 말하면 '확률의 역수'이다.(데이터 수가 하나일 때에 정확히 일치한다.)
작을수록 좋은 값이다.

분기수(number of branches): 다음에 취할 수 있는 선택사항의 수(다음에 출현할 수 있는 단어의 후보 수)

예)

분기수가 1.25 -> 다음에 출현할 수 있는 단어의 후보를 1개 정도로 좁혔다(좋은 모델)
분기수가 5 -> 후보가 아직 5개(나쁜 모델)

입력 데이터가 여러 개일 때

$$L = -\frac{1}{N} \sum_n \sum_k t_{nk} \log y_{nk}$$

$$\text{perplexity} = e^L$$

N:데이터의 총 개수

t_n : 원 핫 벡터로 나타낸 정답 레이블

t_{nk} : n번째 데이터의 k번째 값

y_{nk} : 확률분포(신경망에서는 Softmax의 출력)

L: 신경망의 손실. 교차 엔트로피 오차를 뜻하는 식과 같은 식

6. 게이트가 추가된 RNN

6.1 RNN의 문제점

6.2 기울기 소실과 LSTM

6.3 LSTM 구현

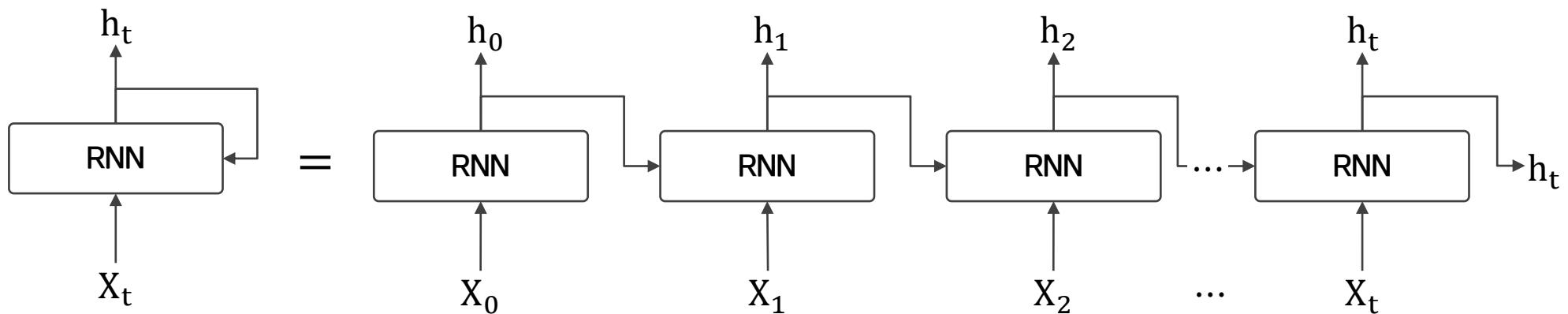
6.4 LSTM을 사용한 언어 모델

6.5 RNNLM 추가 개선

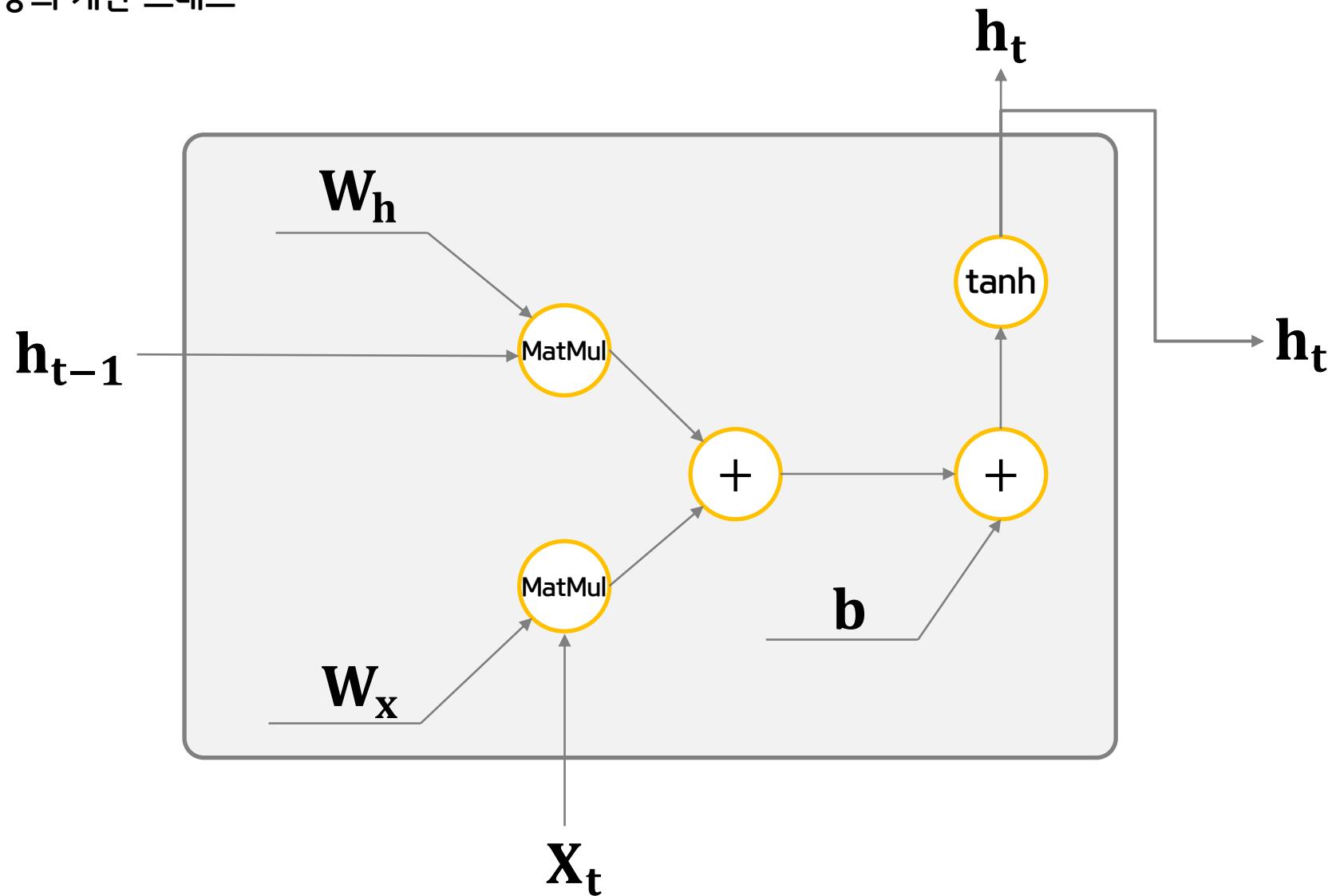
RNN의 문제점

RNN은 시계열 데이터의 장기 의존 관계를 학습하기 어렵다.
그 이유는 BPTT에서 기울기 소실 혹은 기울기 폭발이 일어나기 때문이다.

RNN 계층 : 순환을 펼치기 전과 후



RNN 계층의 계산 그래프



"?"에 들어갈 단어는 ? : (어느 정도의)장기 기억이 필요한 문제의 예

Tom was watching TV in his room. Mary came into the room. Mary said hi to ?

언어 모델은 주어진 단어들을 기초로 다음에 출현할 단어를 예측하는 일을 한다.

이번 절에서는 RNNLM의 단점을 확인하는 차원에서 다음의 문제를 다시 생각해보았다.

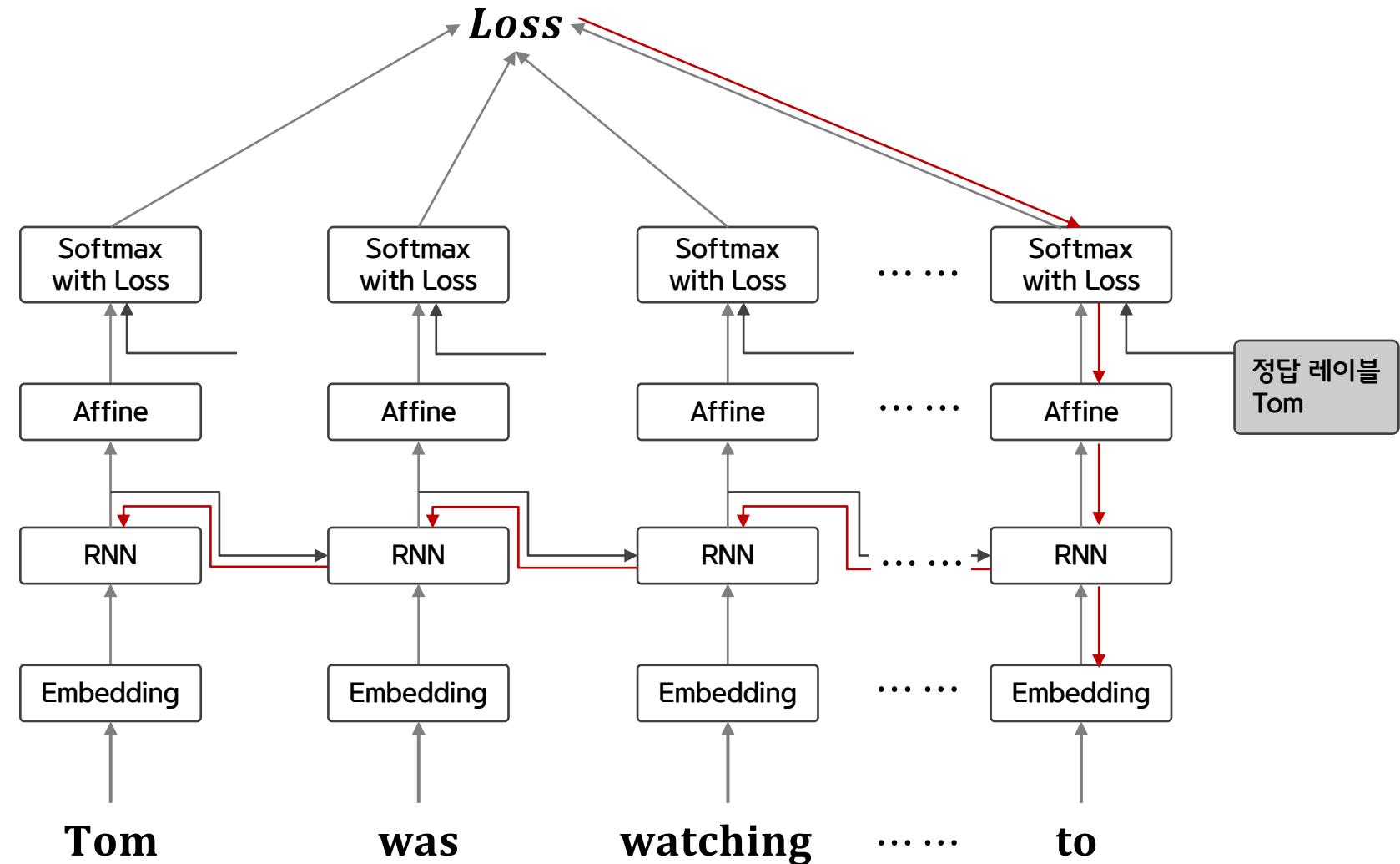
Tom was watching TV in his room. Mary came into the room. Mary said hi to ?

여기서 ?에 들어갈 단어는 Tom이다. RNNLM이 이 문제에 올바르게 답하라면,

현재 맥락에서 'Tom이 방에서 TV를 보고 있음'과 '그 방에 Mary가 들어옴'이란 정보를 기억해야 한다.

즉, 이런 정보를 RNN 계층의 은닉 상태에 인코딩해 보관해야 한다.

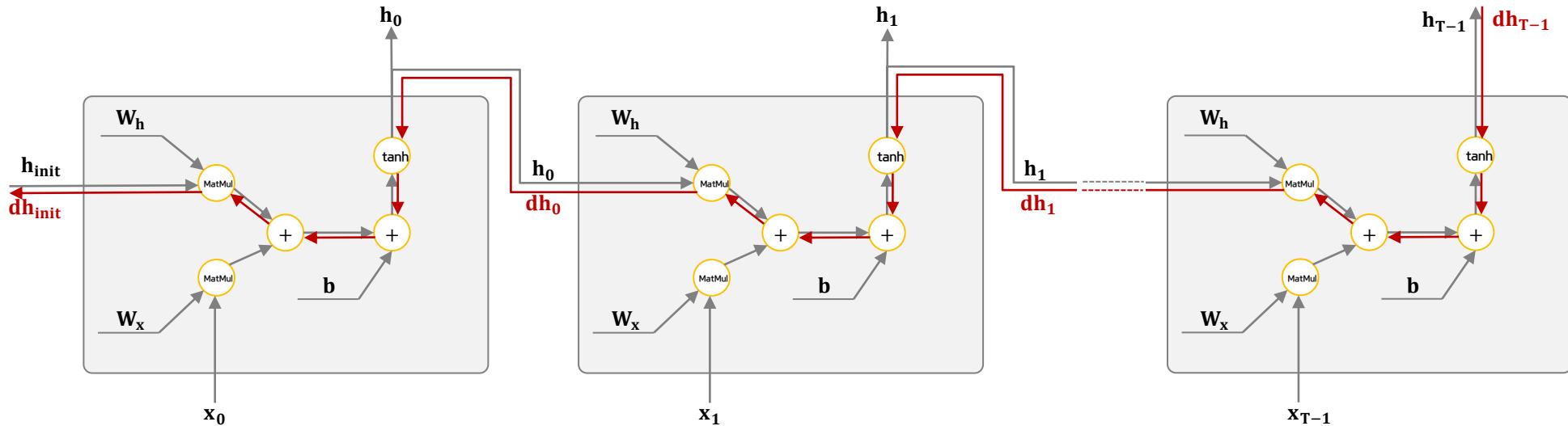
정답 레이블이 "Tom"임을 학습할 때의 기울기 흐름



RNN 계층에서 시간 방향으로의 기울기 전파

$$dh_{prev} = dt \cdot W_h^T$$

$$h_{prev} \cdot W_h$$



위의 RNN 계층의 그림에서 시간 방향 기울기 전파에만 주목해보자.

길이가 T인 시계열 데이터를 가정하여 T번째 정답 레이블로부터 전해지는 기울기가 어떻게 변하는지 보자.

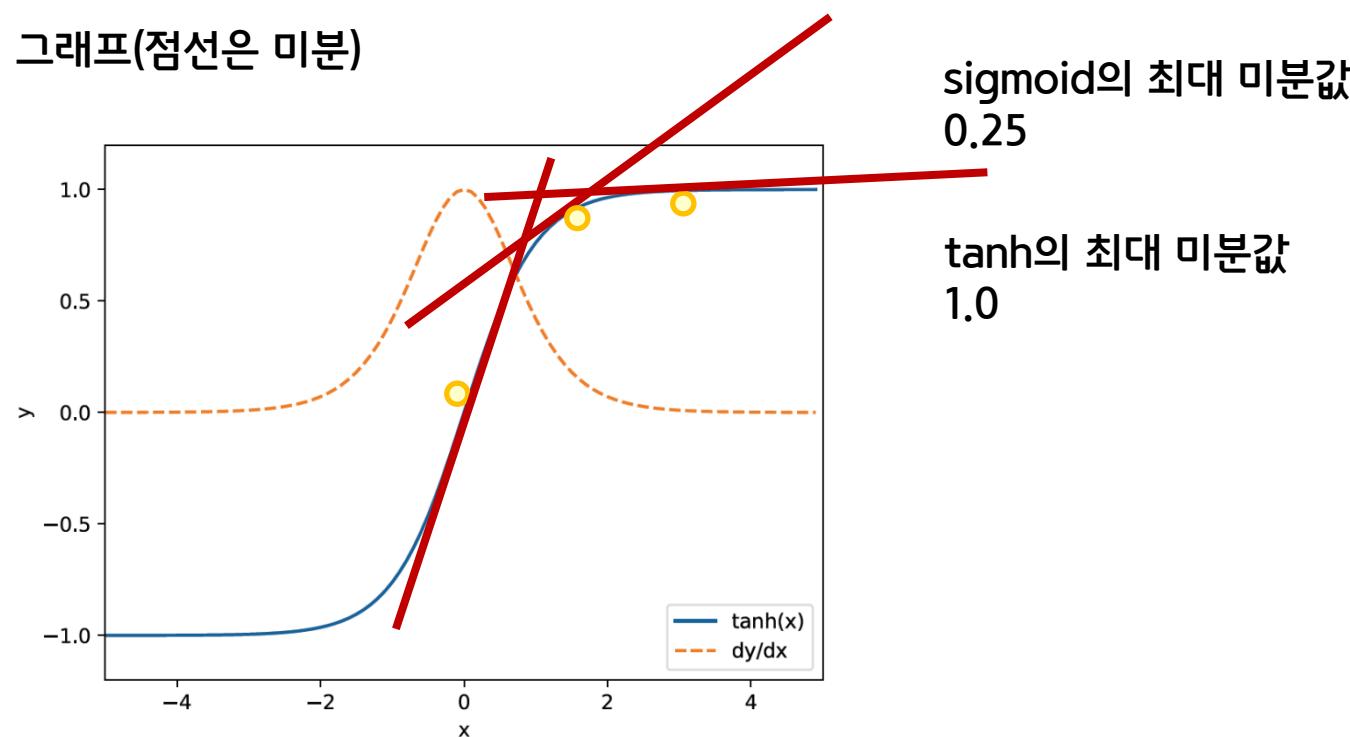
앞의 문제에 대입하면 T번째 정답 레이블이 'Tom'인 경우에 해당한다.

이때 시간 방향 기울기에 주목하면 역전파로 전해지는 기울기는 차례로

'tanh', '+', 'MatMul(행렬 곱)' 연산을 통과한다는 것을 알 수 있다.

'+'의 역전파는 상류에서 전해지는 기울기를 그대로 하류로 흘려보내 기울기는 변하지 않는다.

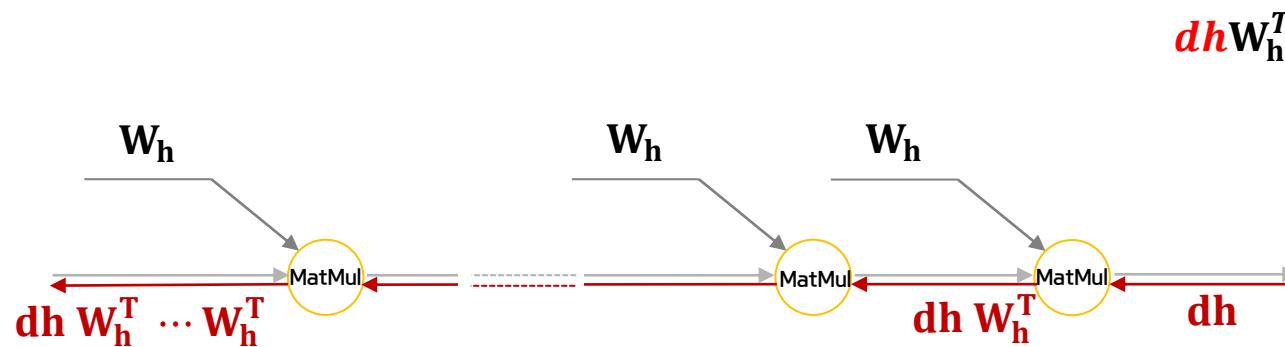
$y=\tanh(x)$ 의 그래프(점선은 미분)



그림에서 점선이 $y=\tanh(x)$ 의 미분이고 값은 1.0 이하이며, x 가 0으로부터 멀어질수록 작아진다.
즉, 역전파에서 기울기가 tanh 노드를 지날 때마다 값은 계속 작아진다는 의미이다.
그리고 tanh 함수를 T번 통과하면 기울기도 T번 반복해서 작아진다.

'MatMul(행렬 곱)' 노드의 경우 tanh 노드를 무시하기로 한다.
그러면 RNN 계층의 역전파 시 기울기는 'MatMul' 연산에 의해서만 변화하게 된다.

RNN 계층의 행렬 곱에만 주목했을 때의 역전파의 기울기



상류로부터 dh 라는 기울기가 흘러온다고 가정하고 이때 MatMul 노드에서의 역전파는 $dh(W_h^T)$ 라는 행렬 곱으로 기울기를 계산한다.

그리고 같은 계산을 시계열 데이터의 시간 크기만큼 반복한다.
주의할 점은 행렬 곱셈에서 매번 똑같은 W_h 가중치를 쓴다는 것이다.

즉, 행렬 곱의 기울기는 시간에 비례해 지수적으로 증가/감소함을 알 수 있으며
증가할 경우 기울기 폭발이라고 한다. 기울기 폭발이 일어나면 오버플로를 일으켜 NaN 같은 값을 발생시킨다.
반대로 기울기가 감소하면 기울기 소실이 일어나고 이는 일정 수준 이하로 작아지면 가중치 매개변수가
더 이상 갱신되지 않으므로 장기 의존 관계를 학습할 수 없게 된다.

```
N = 2    # 미니배치 크기
H = 3    # 은닉 상태 벡터의 차원 수
T = 20   # 시계열 데이터의 길이

dh = np.ones((N, H))

np.random.seed(3) # 재현할 수 있도록 난수의 시드 고정

Wh = np.random.randn(H, H)
# Wh = np.random.randn(H, H) * 0.5

norm_list = []
for t in range(T):
    dh = np.dot(dh, Wh.T)
    norm = np.sqrt(np.sum(dh**2)) / N
    norm_list.append(norm)
```

기울기 클리핑(gradients clipping)

if $\|\hat{g}\| \geq threshold:$

$$\hat{g} = \frac{threshold}{\|\hat{g}\|} \hat{g}$$

기울기 클리핑(gradients clipping)

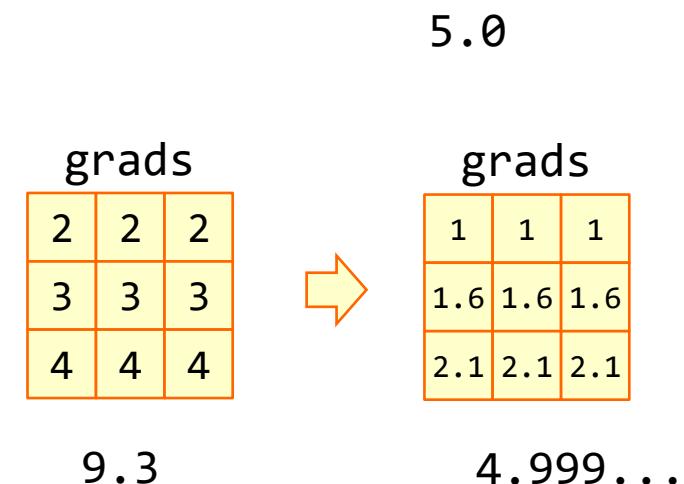
```
def clip_grads(grads, max_norm):
    total_norm = 0
    total_norm += np.sum(grads ** 2)
    total_norm = np.sqrt(total_norm)

    print(total_norm) # 9.3...
    rate = max_norm / (total_norm + 1e-6)
    print(rate) # 0.5...
    if rate < 1:
        grads *= rate

    total_norm = 0
    total_norm += np.sum(grads ** 2)
    total_norm = np.sqrt(total_norm)
    print(total_norm)
```

if $\|\hat{g}\| \geq threshold$:

$$\hat{g} = \frac{\text{threshold}}{\|\hat{g}\|} \hat{g}$$



6. 게이트가 추가된 RNN

6.1 RNN의 문제점

6.2 기울기 소실과 LSTM

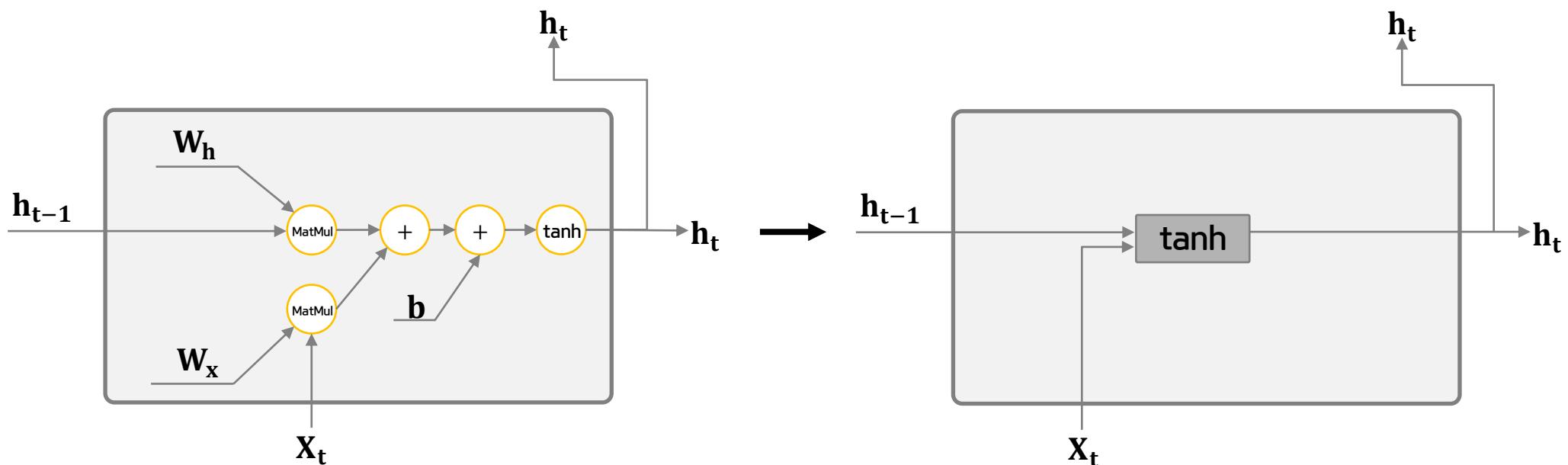
6.3 LSTM 구현

6.4 LSTM을 사용한 언어 모델

6.5 RNNLM 추가 개선

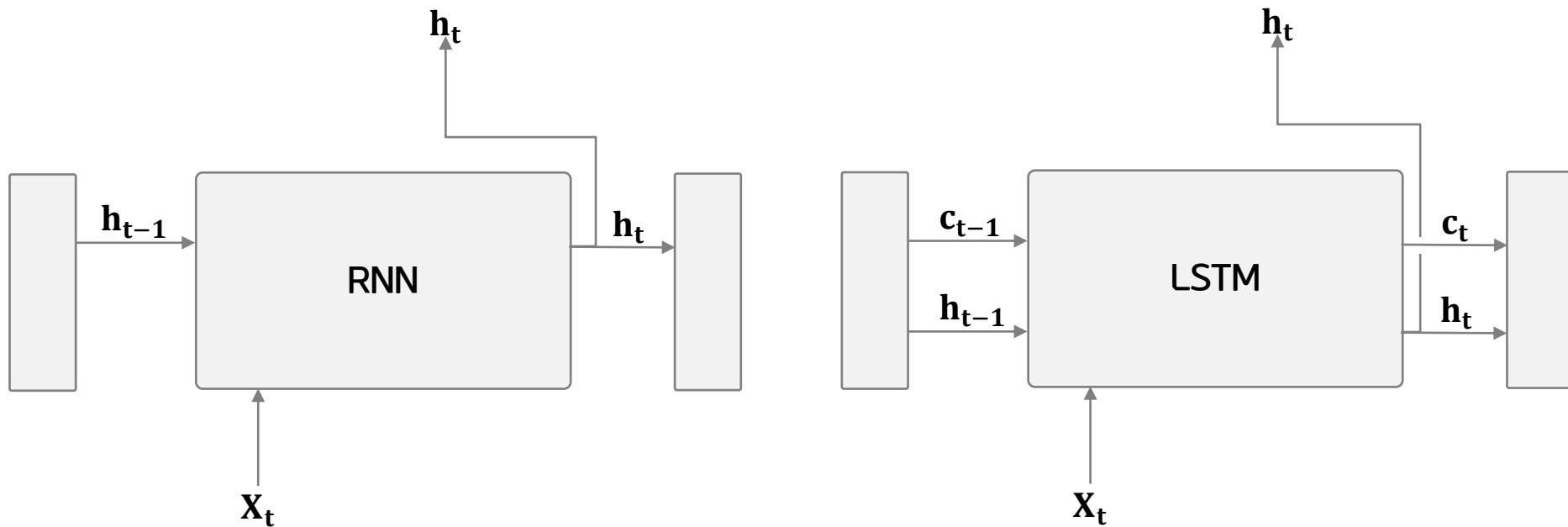
RNN 학습에서 기울기 소실도 큰 문제이다.
이 문제를 해결하려면 RNN 계층의 아키텍처를 근본부터 뜯어고쳐야 한다.

단순화한 도법을 적용한 RNN 계층



여기서는 $\tanh(h_{t-1}W_h + x_tW_x + b)$ 계산을 tanh라는 직사각형 노드 하나로
그리고 직사각형 노드 안에 행렬 곱과 편향의 합, 그리고 tanh 함수에 의한 변환이 모두 포함된 것이다.

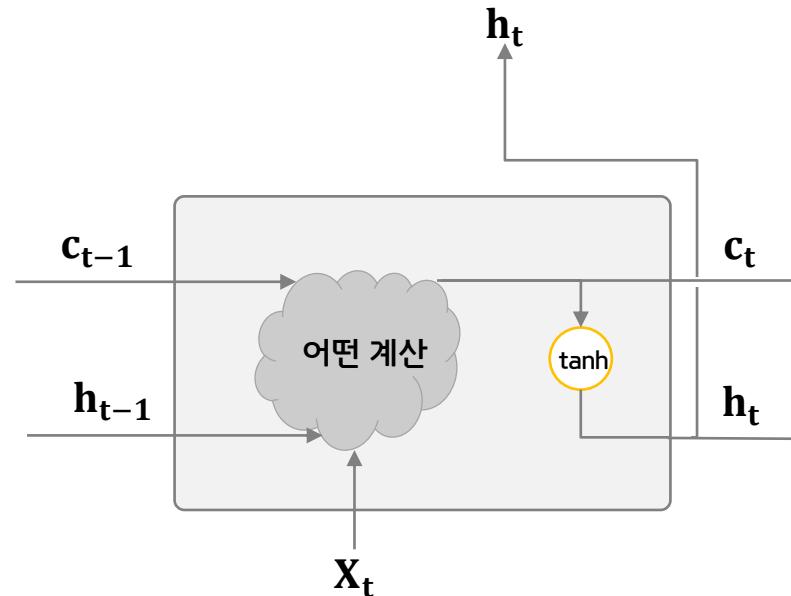
RNN 계층과 LSTM 계층 비교



이 그림에서는 LSTM 계층의 인터페이스에는 c 라는 경로가 있다는 차이가 있다.
여기서 c 를 기억 셀이라 하며 LSTM 전용의 기억 메커니즘이다.

기억 셀의 특징은 데이터를 LSTM 계층 내에서만 주고받는다는 것이다.
다른 계층으로는 출력하지 않는다는 것이다. 반면, LSTM의 은닉 상태 h 는
RNN 계층과 마찬가지로 다른 계층, 위쪽으로 출력된다.

기억 셀 c_t 를 바탕으로 은닉 상태 h_t 를 계산하는 LSTM 계층



그림에서 현재의 기억 셀 c_t 는 3개의 입력 (c_{t-1} , h_{t-1} , X_t)으로부터
'어떤 계산'을 수행하여 구할 수 있다.

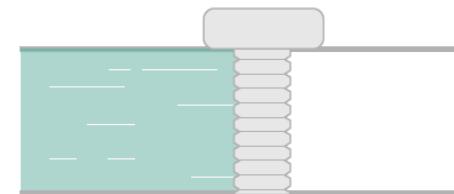
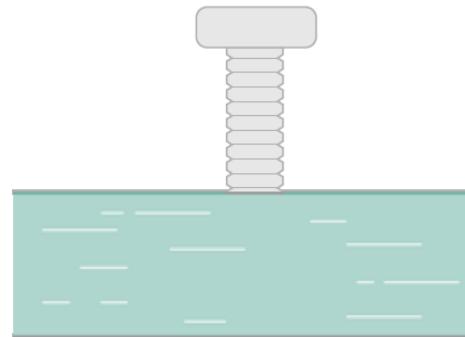
여기서 핵심은 갱신된 c_t 를 사용해 은닉 상태 h_t 를 계산한다는 것이다.

또한 이 계산은 $h_t = \tanh(c_t)$ 인데, 이는 c_t 의 각 요소에 tanh 함수를 적용한다는 뜻이다.

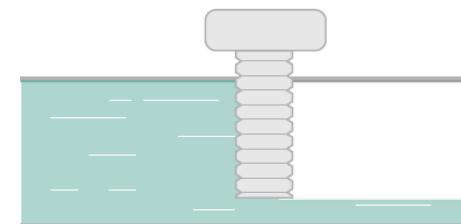
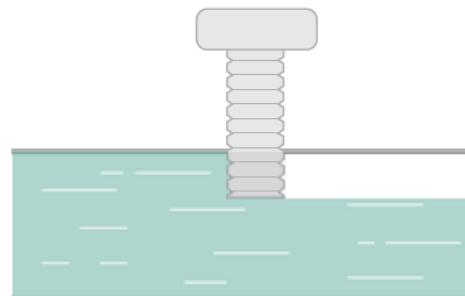
'게이트'란 우리나라로 '문'을 의미하는 단어이다.

문은 열거나 닫을 수 있듯이, 게이트는 데이터의 흐름을 제어한다.

비유하자면 게이트는 물의 흐름을 제어한다.



물이 흐르는 양을 0.0~1.0 범위에서 제어한다.



$\tanh(c_t)$ 의 각 원소에 대해 '그것이 다음 시각의 은닉 상태에 얼마나 중요한가'를 조정한다. 한편, 이 게이트는 다음 은닉 상태 h_t 의 출력을 담당하는 게이트이므로 output 게이트라고 한다.

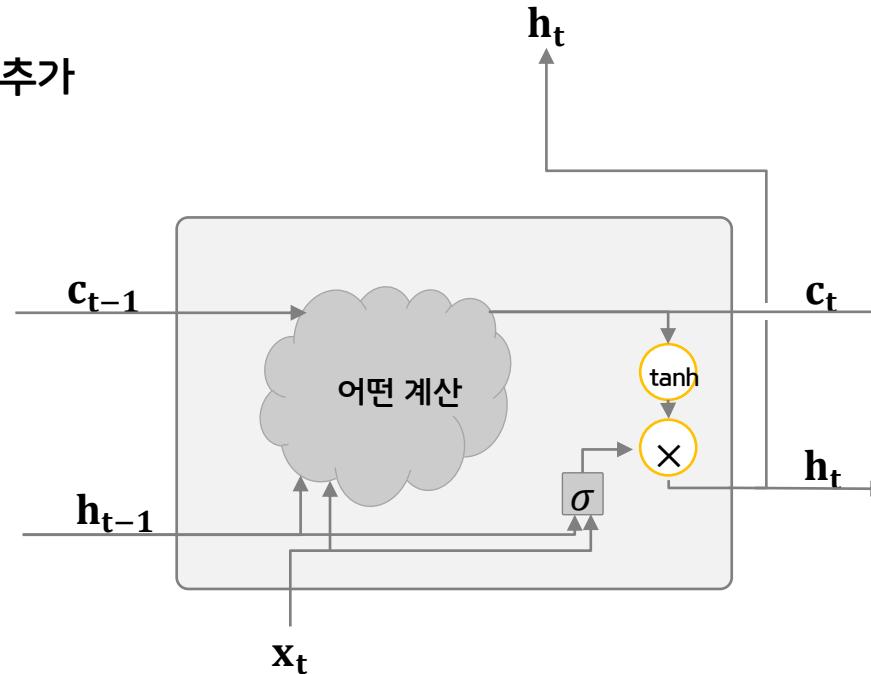
output 게이트의 열림 상태는 입력 x_t 와 이전 상태 h_{t-1} 로부터 구한다.

이때의 식은 밑에 식과 같다.

$$o = \sigma(x_t W_x^{(o)} + h_{t-1} W_h^{(o)} + b^{(o)})$$

밑에 그림에서 output 게이트에서 수행하는 식의 계산을 sigma로 표기했다.
sigma의 출력을 o 라고 하면 h_t 는 o 와 $\tanh(c_t)$ 의 곱으로 계산된다.

output 게이트 추가

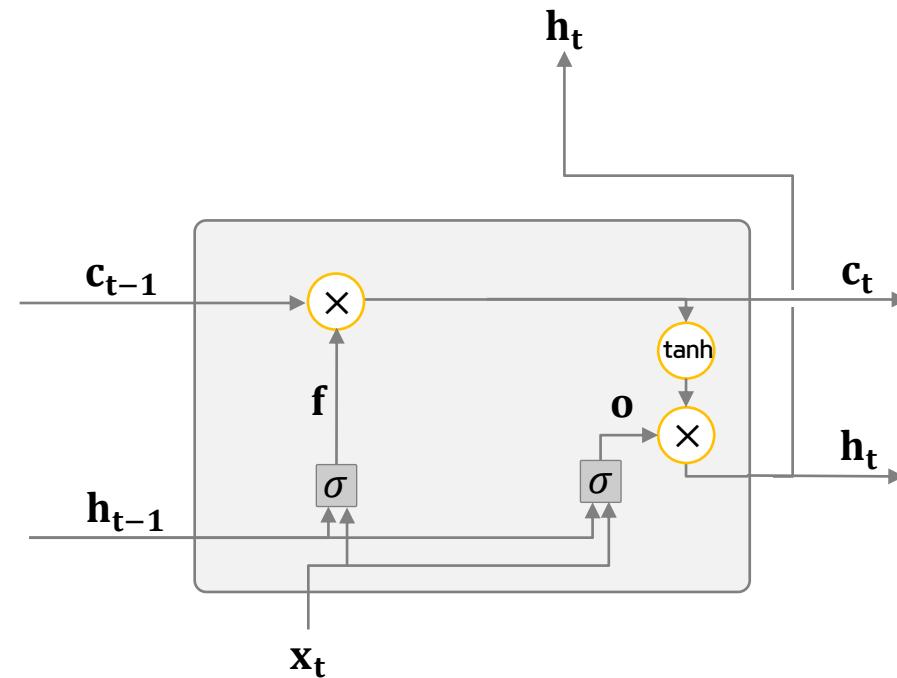


아다마르 곱(Hadamard product)

$$h_t = o \odot \tanh(c_t)$$

다음에 해야 할 일은 기억 셀에 '무엇을 잊을까'를 명확하게 지시하는 것이다.

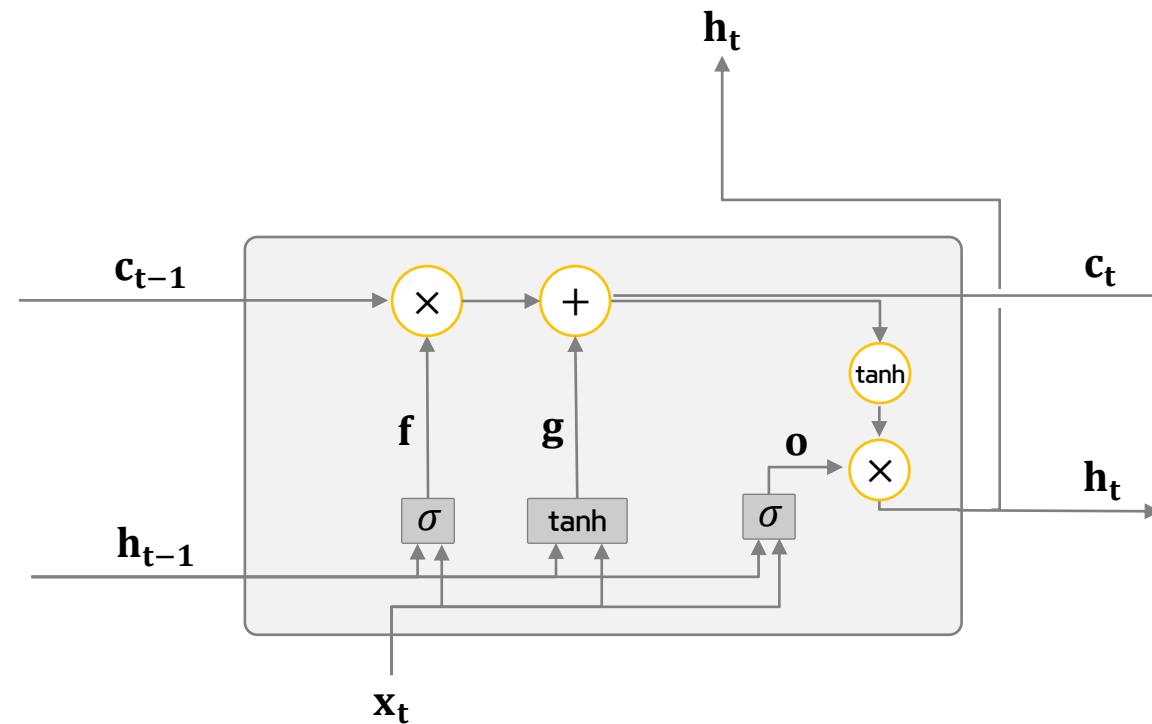
forget 게이트 추가



$$f = \sigma(x_t W_x^{(f)} + h_{t-1} W_h^{(f)} + b^{(f)})$$

forget 게이트를 거치면서 이전 시각의 기억 셀로부터 잊어야 할 기억이 삭제되었다.

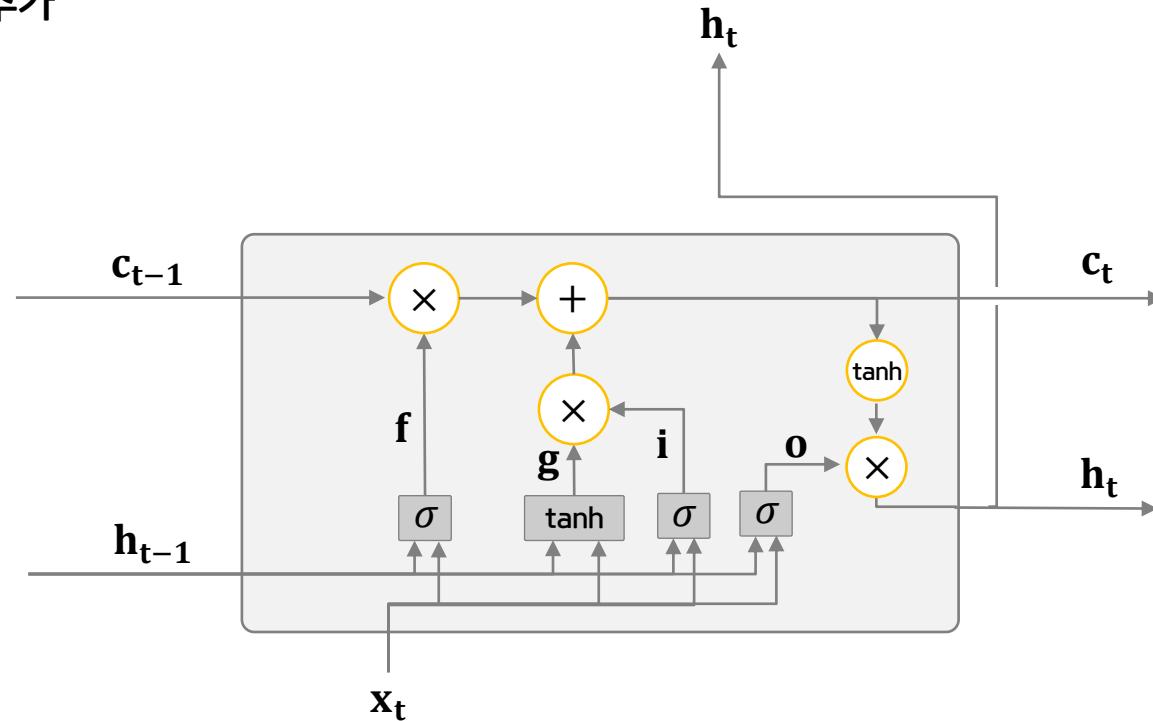
새로운 기억 셀



$$g = \tanh(x_t W_x^{(g)} + h_{t-1} W_h^{(g)} + b^{(g)})$$

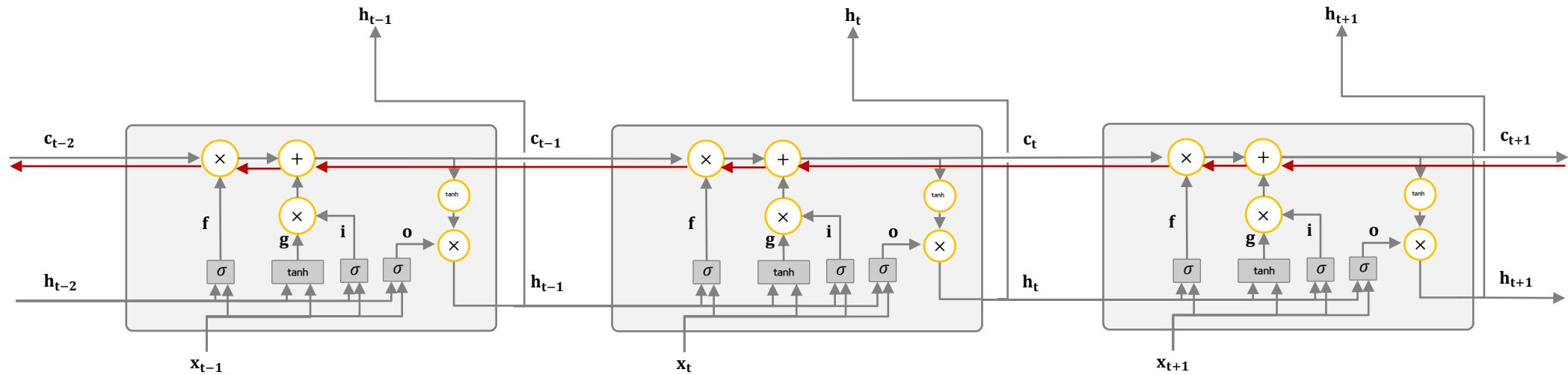
input 게이트 추가

GRU



$$i = \sigma(x_t W_x^{(i)} + h_{t-1} W_h^{(i)} + b^{(i)})$$

기억 셀의 역전파



6. 게이트가 추가된 RNN

6.1 RNN의 문제점

6.2 기울기 소실과 LSTM

6.3 LSTM 구현

6.4 LSTM을 사용한 언어 모델

6.5 RNNLM 추가 개선

$$f = \sigma(x_t W_x^{(f)} + h_{t-1} W_h^{(f)} + b^{(f)})$$

$$g = \tanh(x_t W_x^{(g)} + h_{t-1} W_h^{(g)} + b^{(g)})$$

$$i = \sigma(x_t W_x^{(i)} + h_{t-1} W_h^{(i)} + b^{(i)})$$

$$o = \sigma(x_t W_x^{(o)} + h_{t-1} W_h^{(o)} + b^{(o)})$$

$$c_t = f \odot c_{t-1} + g \odot i$$

$$h_t = o \odot \tanh(c_t)$$

```

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')

```

```

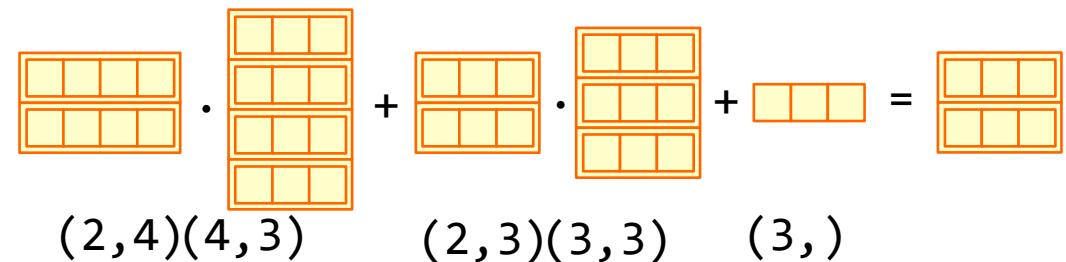
def forward(self, x, h_prev, c_prev):
    Wx, Wh, b = self.params
    N, H = h_prev.shape

```

$$A = \text{np.dot}(x, Wx) + \text{np.dot}(h_{\text{prev}}, Wh) + b$$

$$x_t W_x^{(g)} + h_{t-1} W_h^{(g)} + b^{(g)}$$

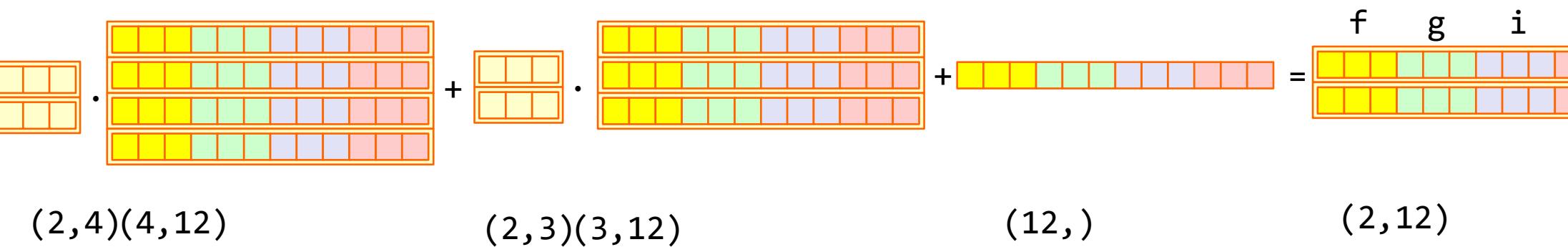
$$f = A[:, :H]$$



$$g = A[:, H:2*H]$$

$$i = A[:, 2*H:3*H]$$

$$o = A[:, 3*H:]$$



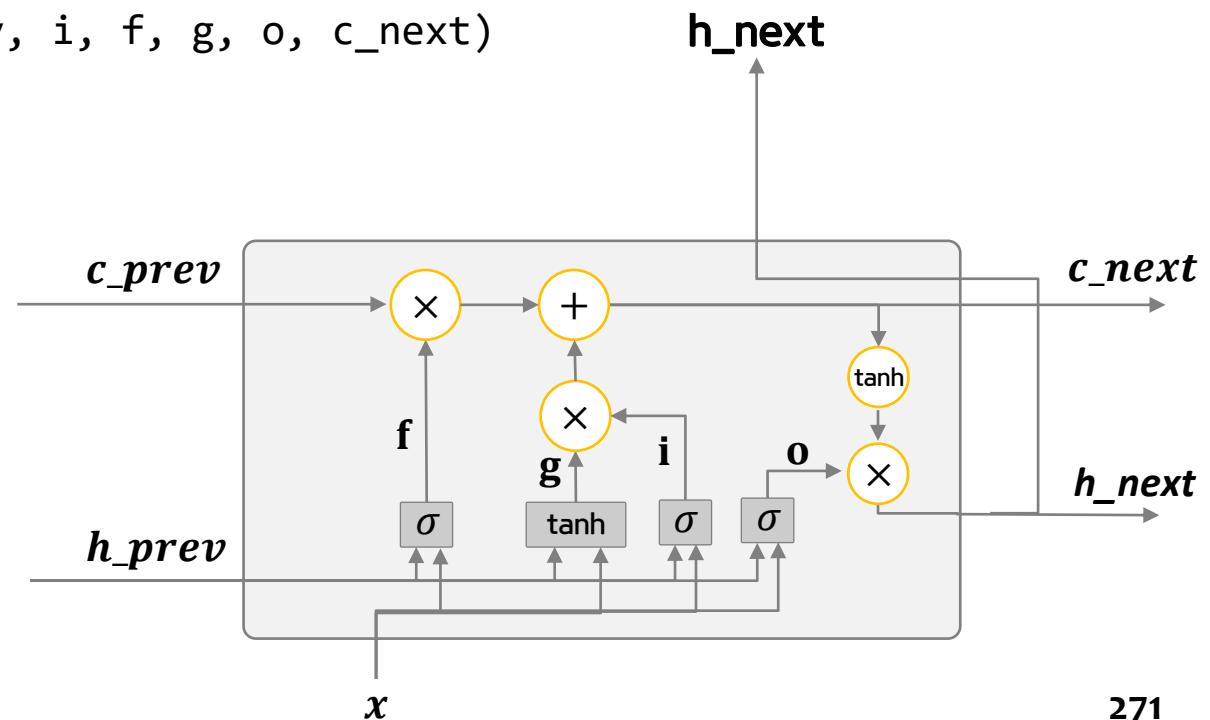
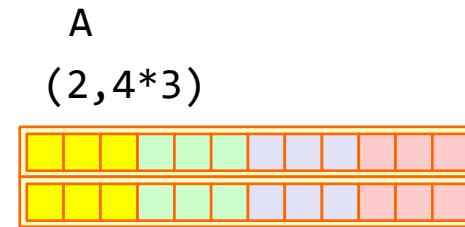
```

def forward(self, x, h_prev, c_prev):
    ...
    f = sigmoid(f)
    g = np.tanh(g)
    i = sigmoid(i)
    o = sigmoid(o)

    c_next = f * c_prev + g * i
    h_next = o * np.tanh(c_next)

    self.cache = (x, h_prev, c_prev, i, f, g, o, c_next)
    return h_next, c_next

```



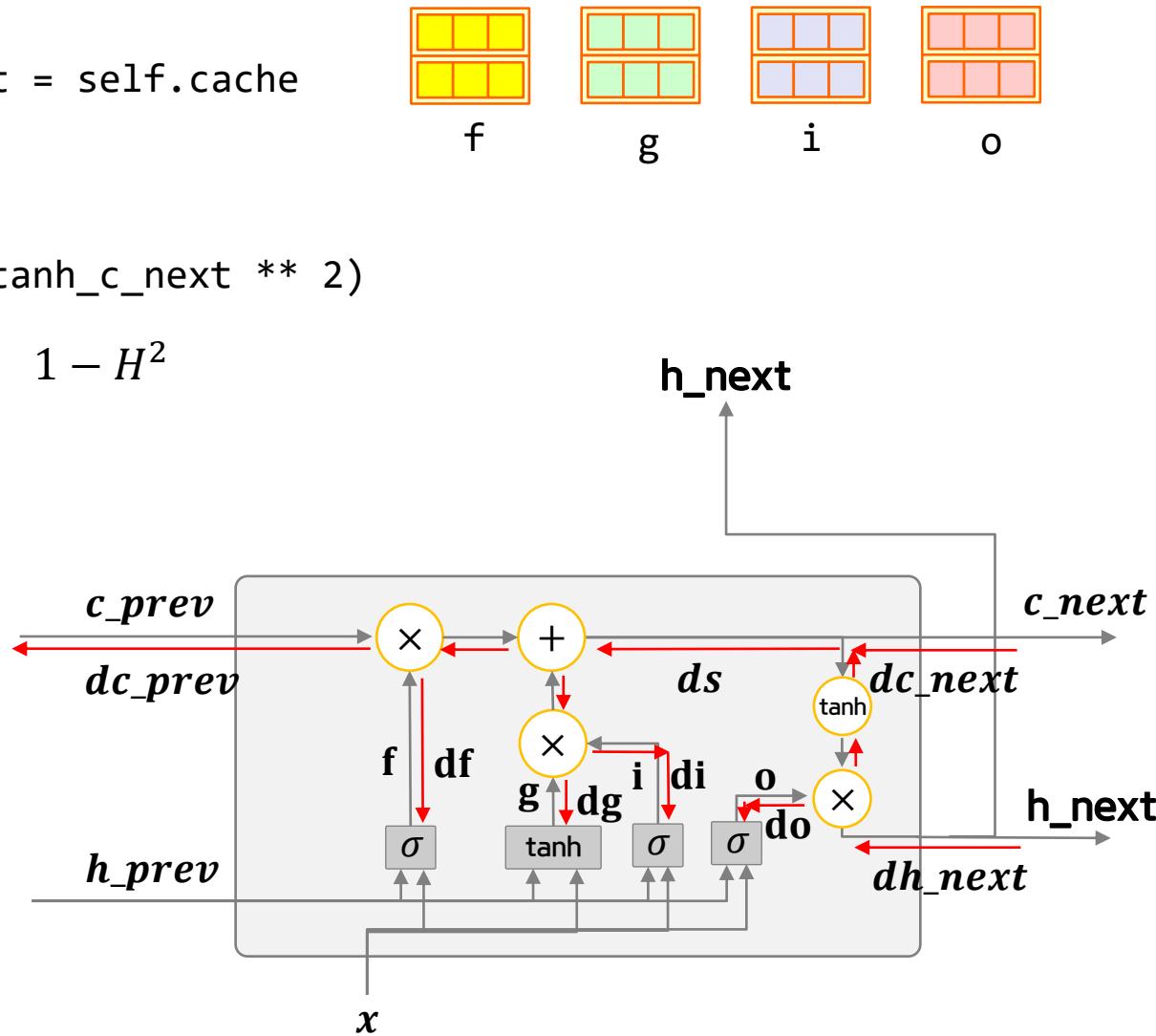
```

def backward(self, dh_next, dc_next):
    Wx, Wh, b = self.params
    x, h_prev, c_prev, i, f, g, o, c_next = self.cache
    tanh_c_next = np.tanh(c_next)

    ds = dc_next + (dh_next * o) * (1 - tanh_c_next ** 2)

    dc_prev = ds * f
    di = ds * g
    df = ds * c_prev
    do = dh_next * tanh_c_next
    dg = ds * i

```



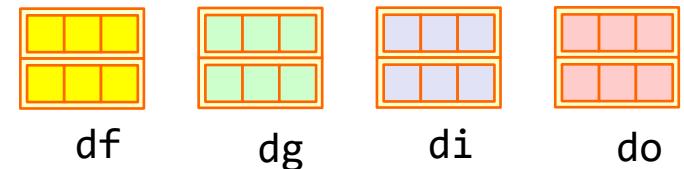
```

di *= i * (1 - i)
df *= f * (1 - f)
do *= o * (1 - o)
dg *= (1 - g ** 2)

dA = np.hstack((df, dg, di, do))

dWh = np.dot(h_prev.T, dA)
dWx = np.dot(x.T, dA)
db = dA.sum(axis=0)

```



$$h_{\text{prev.T}} \cdot dA = dWh$$

(3,2)(2,12)

$$x.T \cdot dA = dWx$$

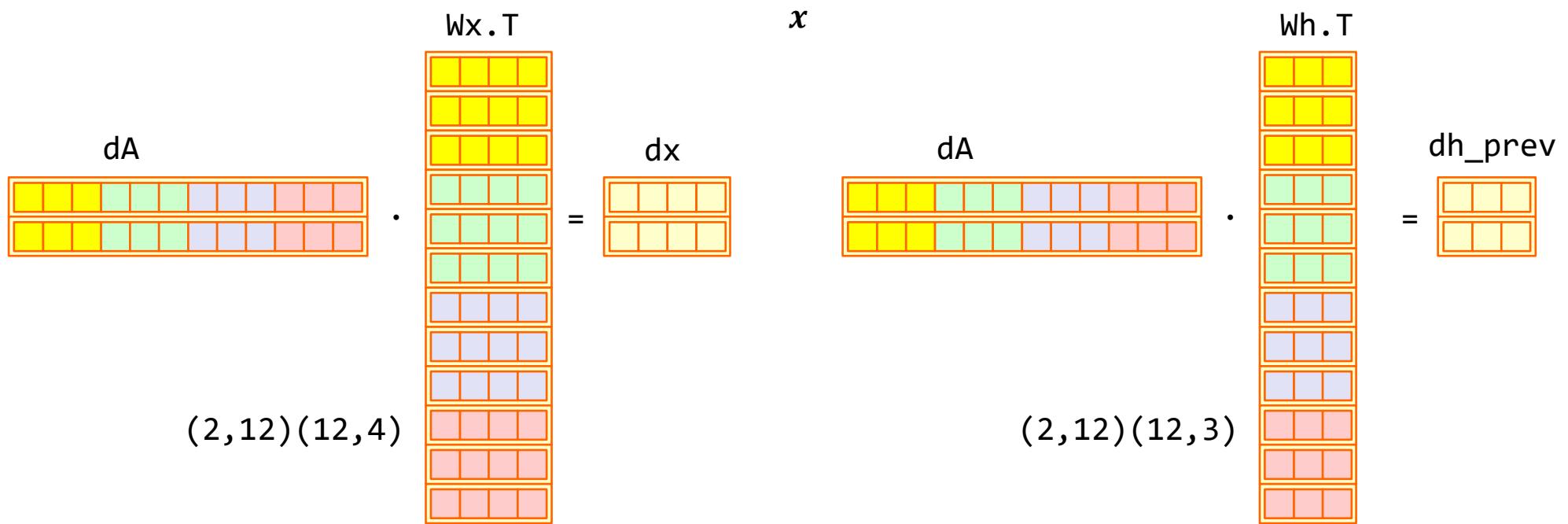
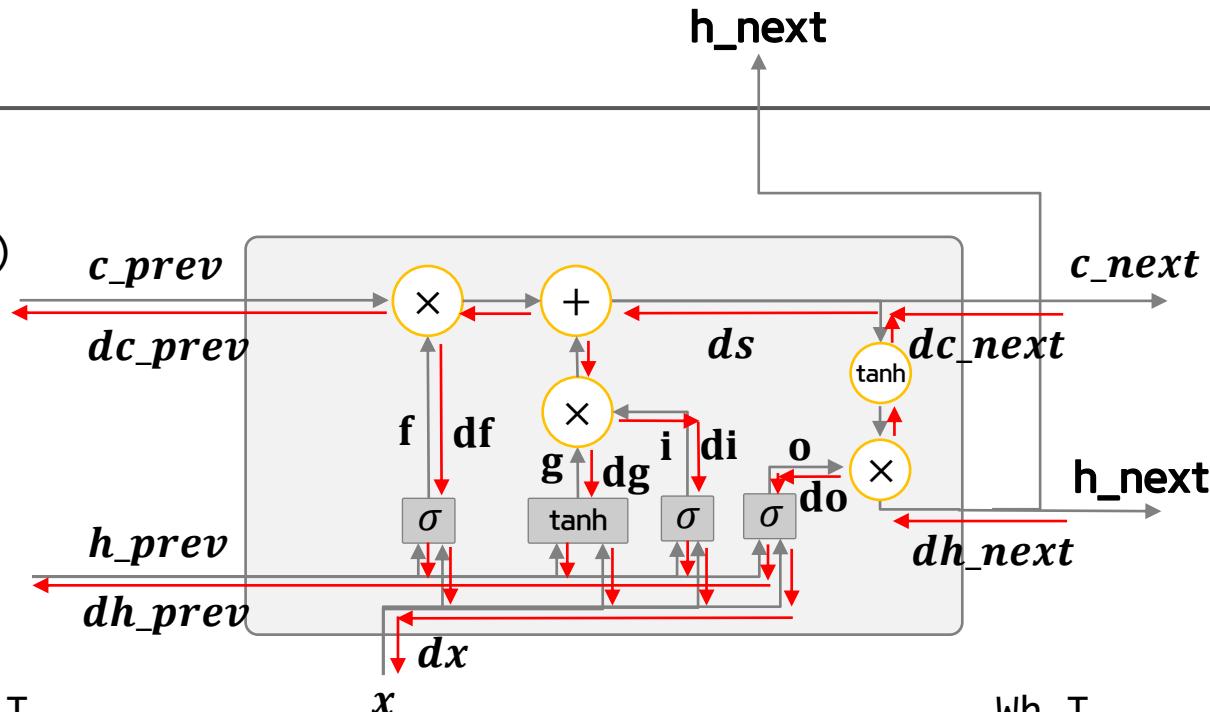
(4,2)(2,12)

$$(2,12) \downarrow db$$

```

dx = np.dot(dA, Wx.T)
dh_prev = np.dot(dA, Wh.T)

```



각 식의 가중치들을 모아 4개의 식을 단 한 번의 아핀 변환으로 계산

$$x_t W_x^{(f)} + h_{t-1} W_h^{(f)} + b^{(f)}$$

$$x_t W_x^{(g)} + h_{t-1} W_h^{(g)} + b^{(g)}$$

$$x_t W_x^{(i)} + h_{t-1} W_h^{(i)} + b^{(i)}$$

$$x_t W_x^{(o)} + h_{t-1} W_h^{(o)} + b^{(o)}$$



$$x_t \begin{bmatrix} W_x^{(f)} & W_x^{(g)} & W_x^{(i)} & W_x^{(o)} \end{bmatrix} + h_{t-1} \begin{bmatrix} W_h^{(f)} & W_h^{(g)} & W_h^{(i)} & W_h^{(o)} \end{bmatrix} + [b^{(f)} \ b^{(g)} \ b^{(i)} \ b^{(o)}]$$



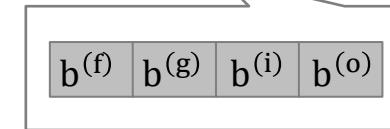
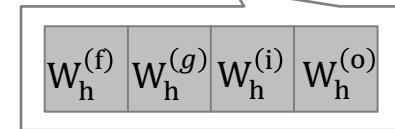
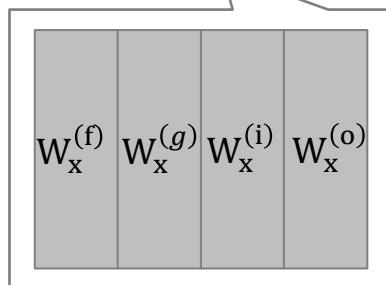
$$x_t W_x$$

+

$$h_{t-1} W_h$$

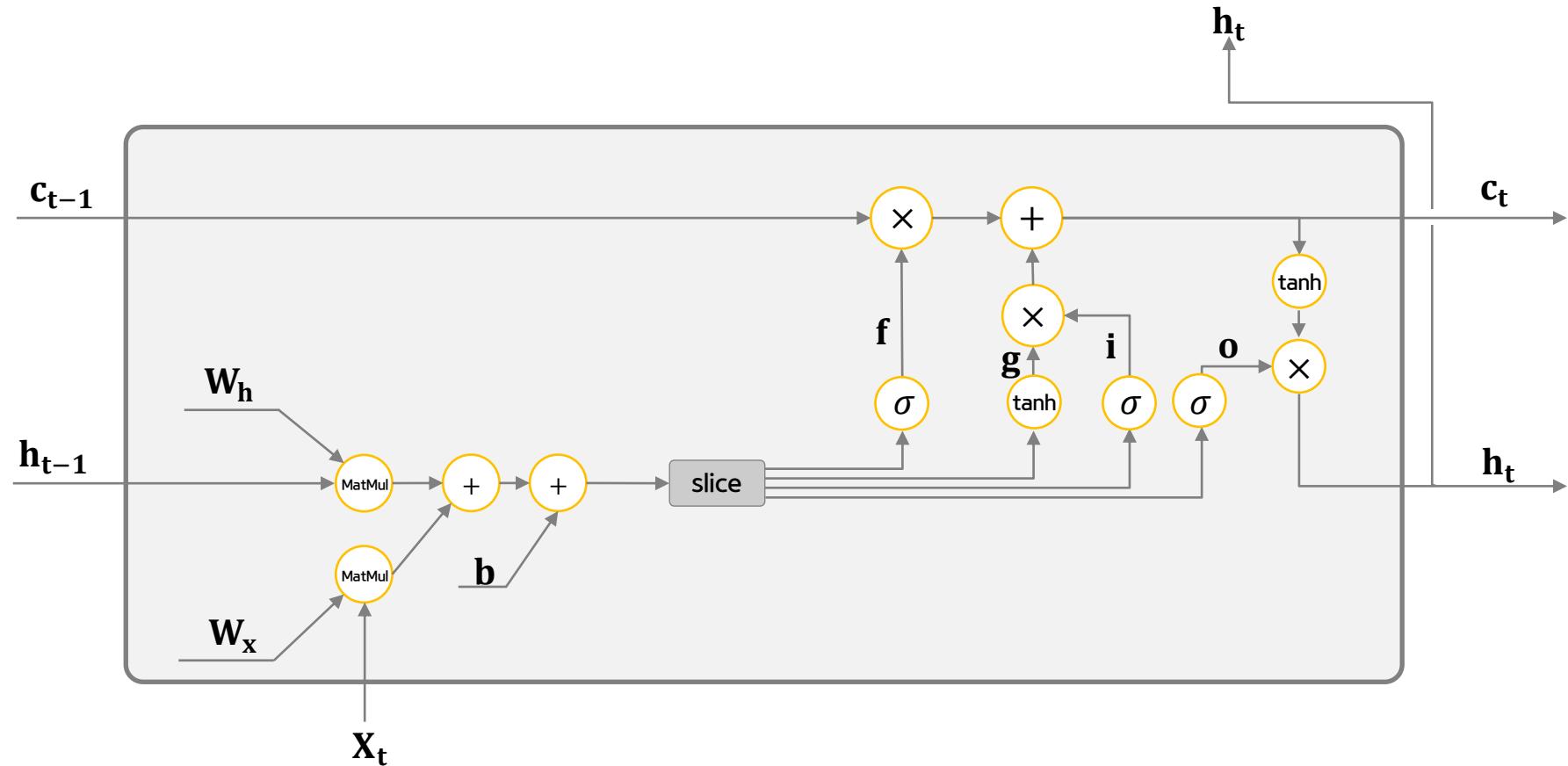
+

$$b$$



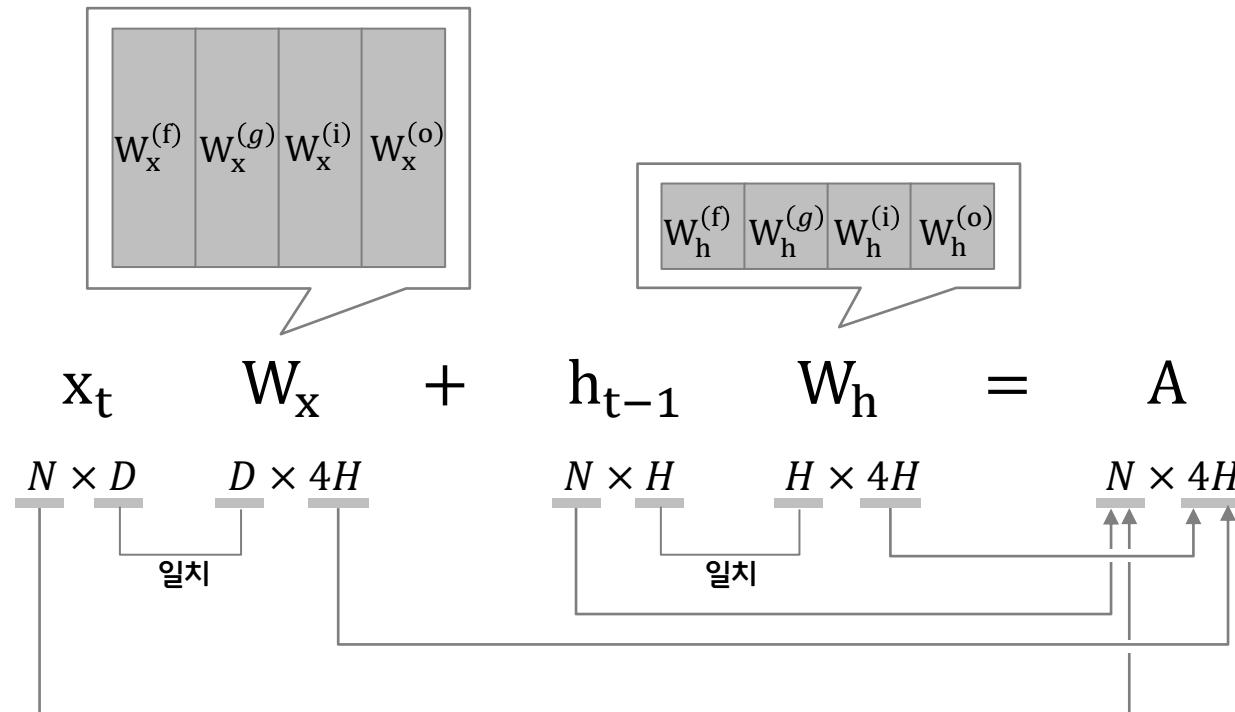
$$(N, H)(H, 4*H) = (N, 4*H)$$

4개분의 가중치를 모아 아핀 변환을 수행하는 LSTM의 계산 그래프

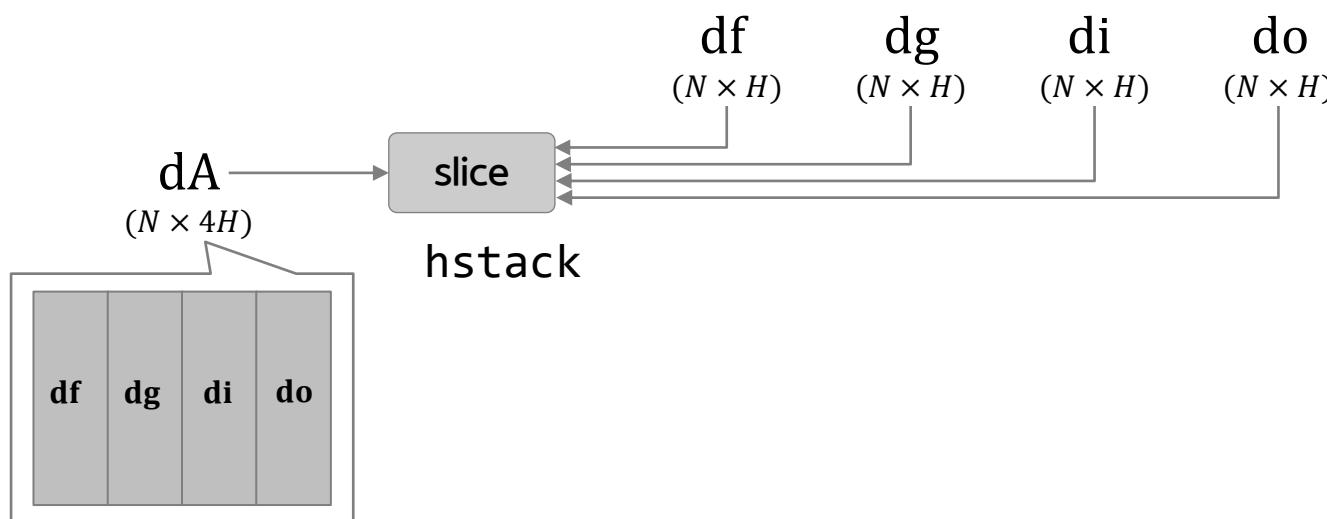
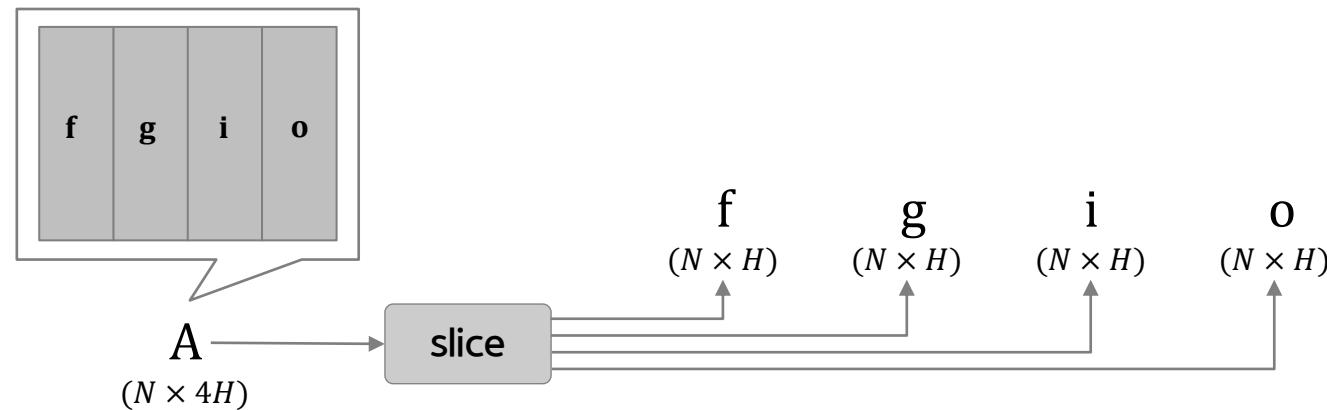


만약 W_x, W_h, b 각각에 4개분의 가중치가 포함되어 있다고 가정하면 위의 그림처럼 그래프가 그려진다.

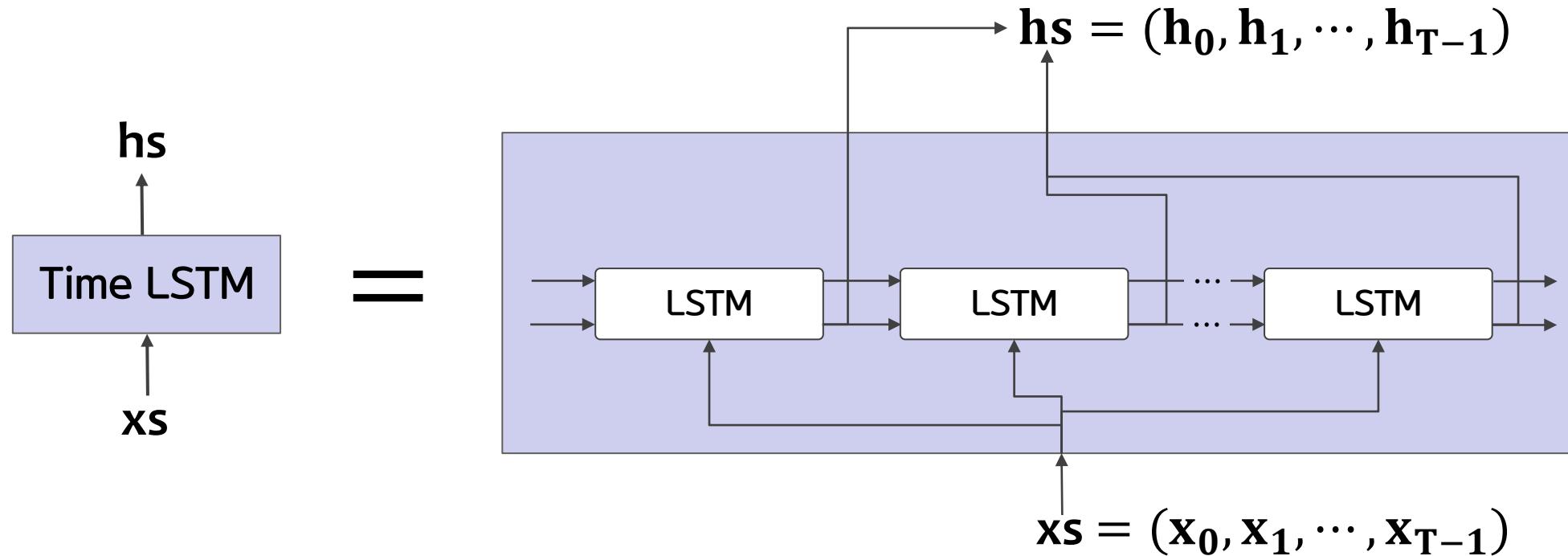
아핀 변환 시의 형상 추이



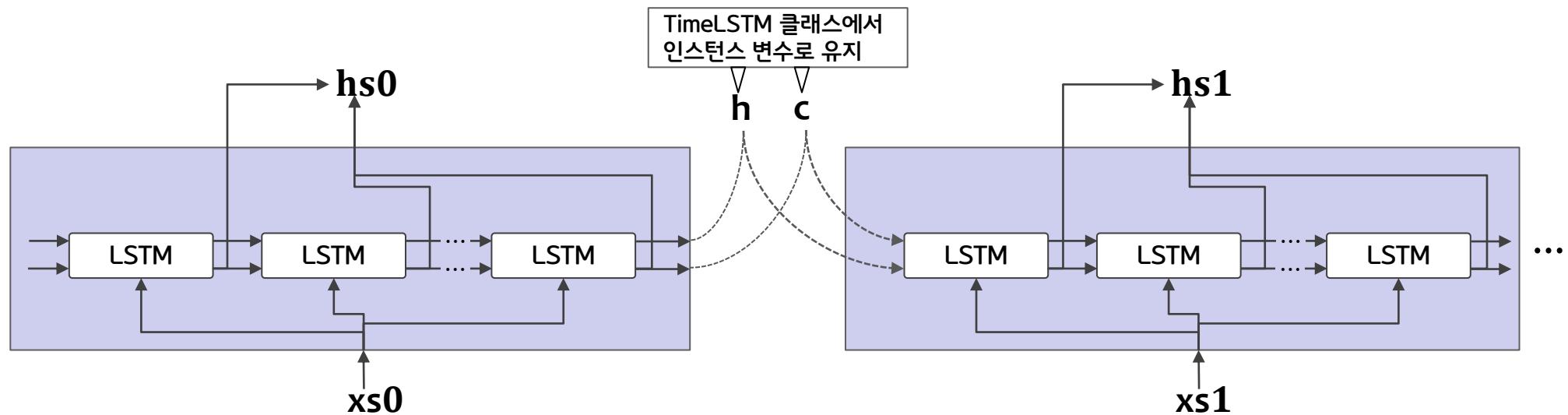
slice 노드의 순전파(위)와 역전파(아래)



Time LSTM의 입출력



Time LSTM 역전파의 입출력



6. 게이트가 추가된 RNN

6.1 RNN의 문제점

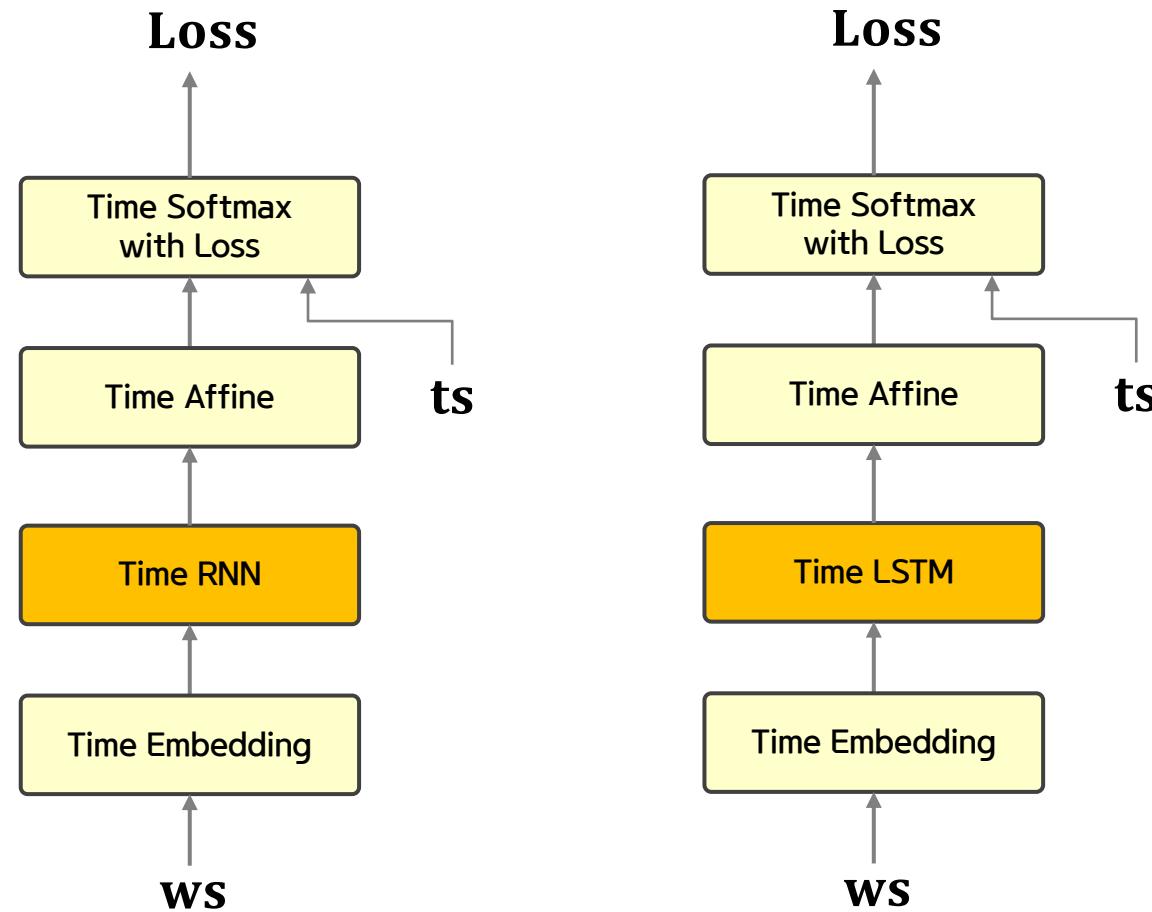
6.2 기울기 소실과 LSTM

6.3 LSTM 구현

6.4 LSTM을 사용한 언어 모델

6.5 RNNLM 추가 개선

언어 모델의 신경망 구성



6. 게이트가 추가된 RNN

6.1 RNN의 문제점

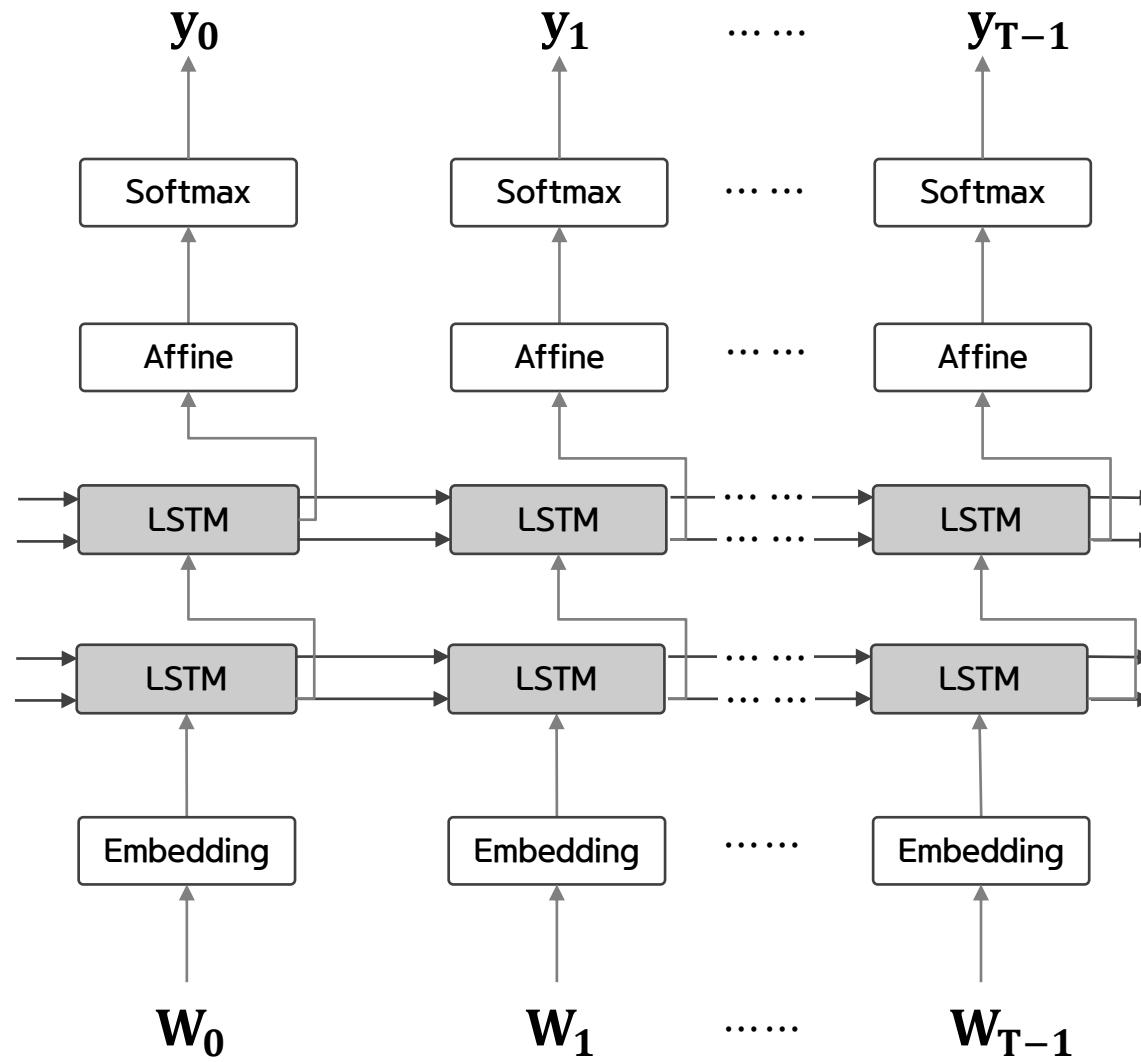
6.2 기울기 소실과 LSTM

6.3 LSTM 구현

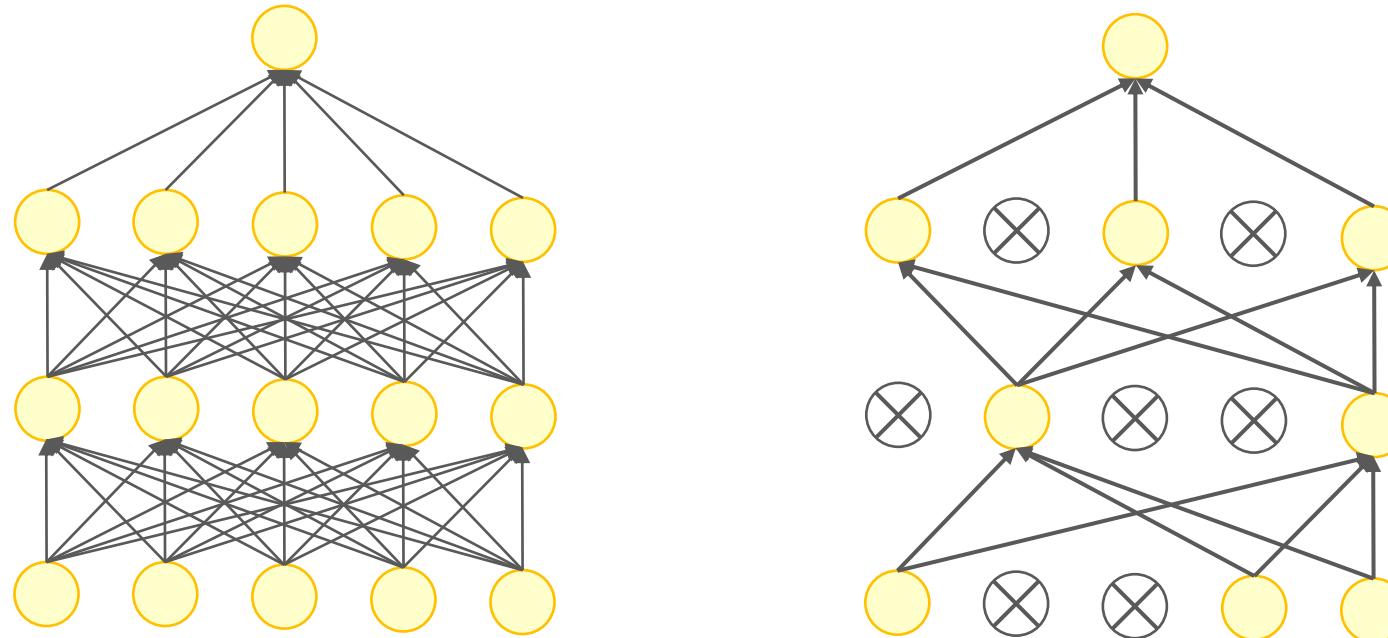
6.4 LSTM을 사용한 언어 모델

6.5 RNNLM 추가 개선

LSTM 계층을 2층으로 쌓은 RNNLM

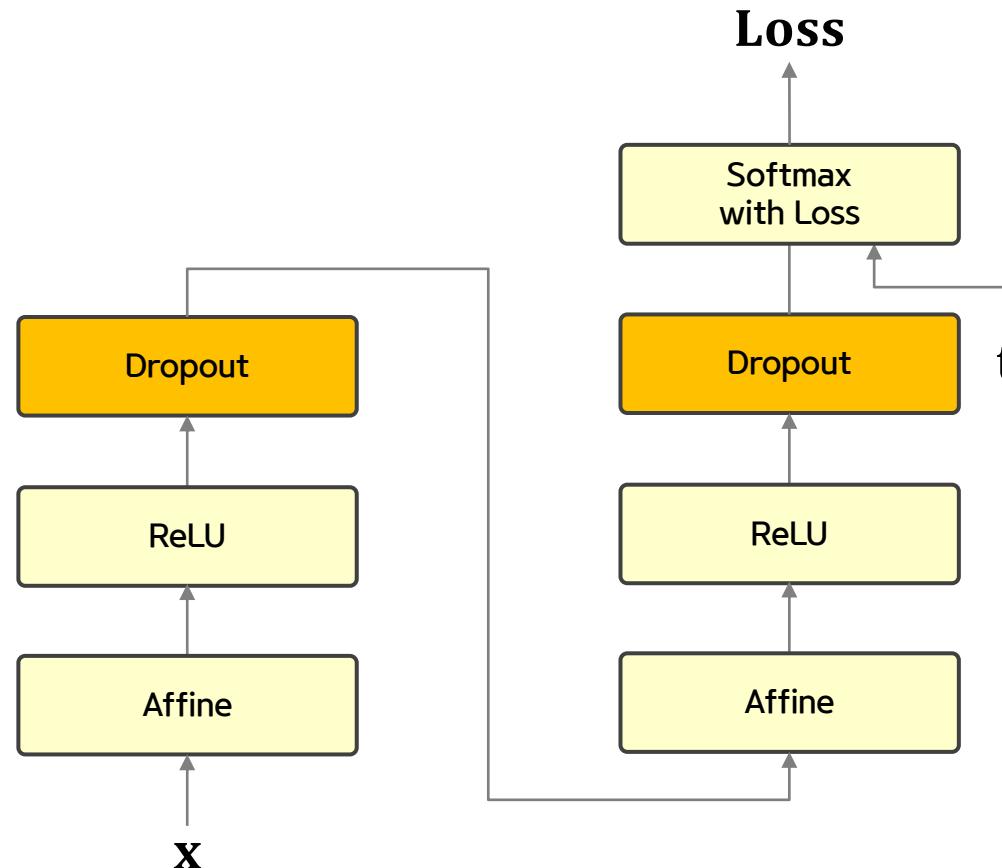


드롭아웃 개념도 : 왼쪽이 일반적인 신경망, 오른쪽이 드롭아웃을 적용한 신경망



드롭아웃은 무작위로 뉴런을 선택하여 선택한 뉴런을 무시한다.
무시한다는 말은 그 앞 계층으로부터의 신호 전달을 막는다는 뜻이다.
이 '무작위한 무시'가 제약이 되어 신경망의 일반화 성능을 개선하는 것이다.

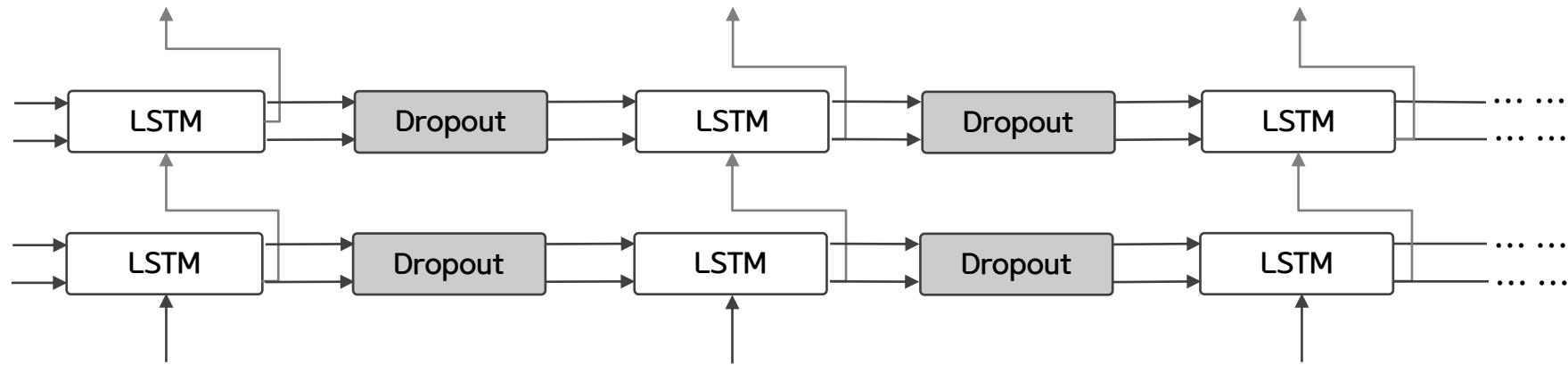
피드포워드 신경망에 드롭아웃 계층을 적용하는 예



이 그림은 드롭아웃 계층을 활성화 함수 뒤에 삽입하는 방법으로 과적합 억제에 기여하는 모습이다.

RNN을 사용한 모델에서 드롭아웃 계층을 LSTM 계층의 시계열 방향으로 삽입하면 좋은 방법이 아니다.

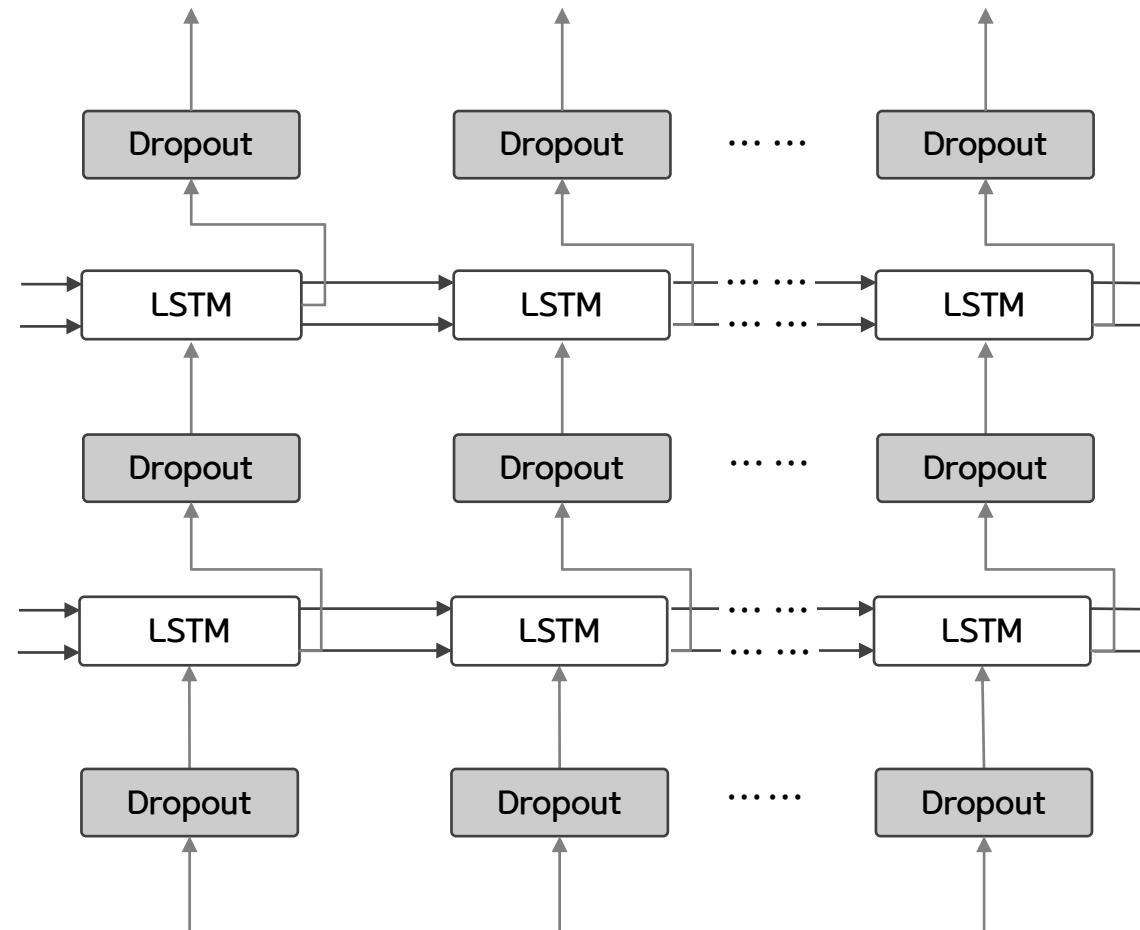
나쁜 예: 드롭아웃 계층을 시계열 방향으로 삽입



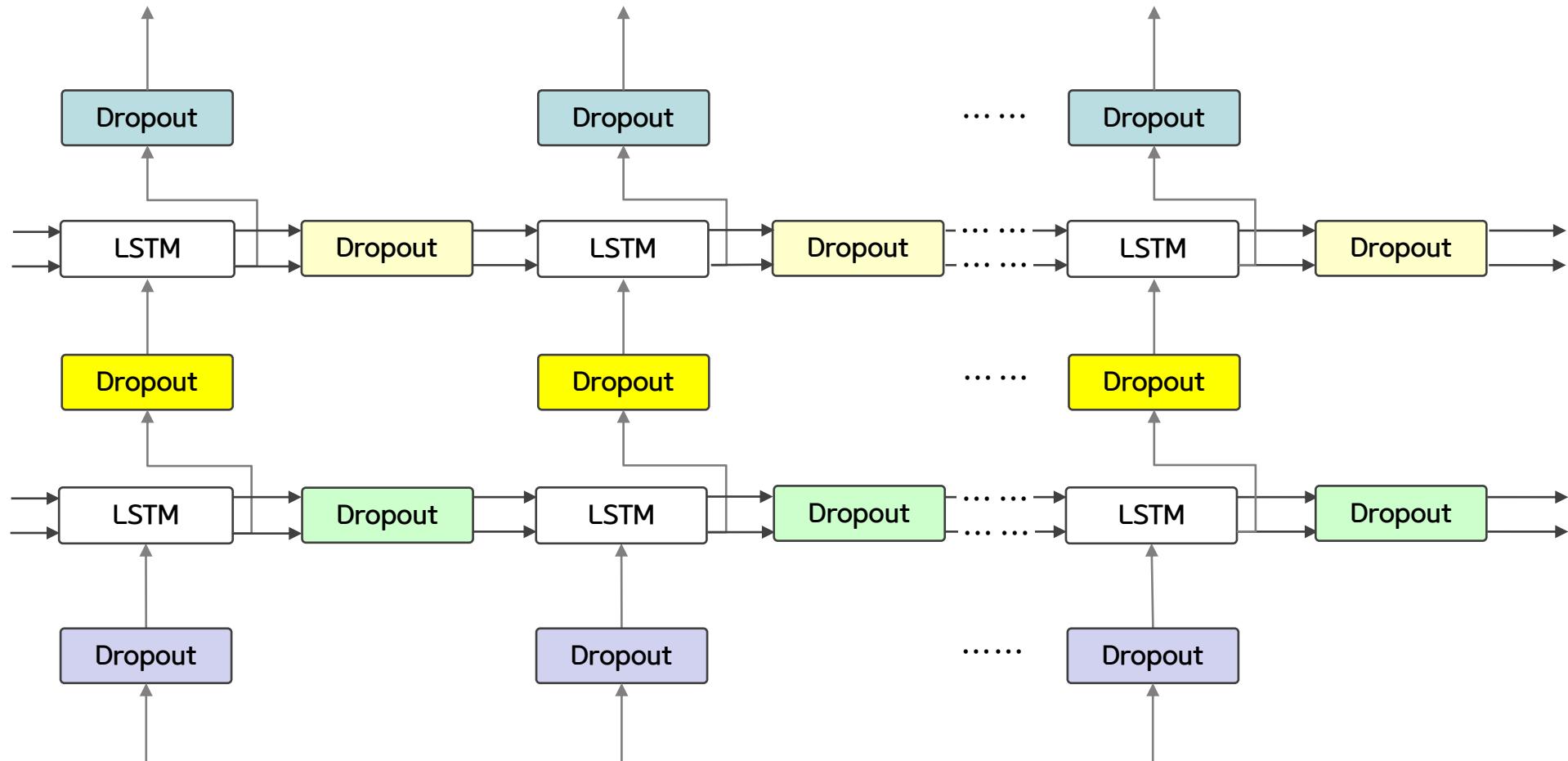
RNN에서 시계열 방향으로 드롭아웃을 학습 시 넣어버리면 시간이 흐름에 따라 정보가 사라질 수 있다.
즉, 흐르는 시간에 비례해 드롭아웃에 의한 노이즈가 축적된다.

드롭아웃 계층을 깊이 방향(상하 방향)으로 삽입하는 방안을 생각해보자

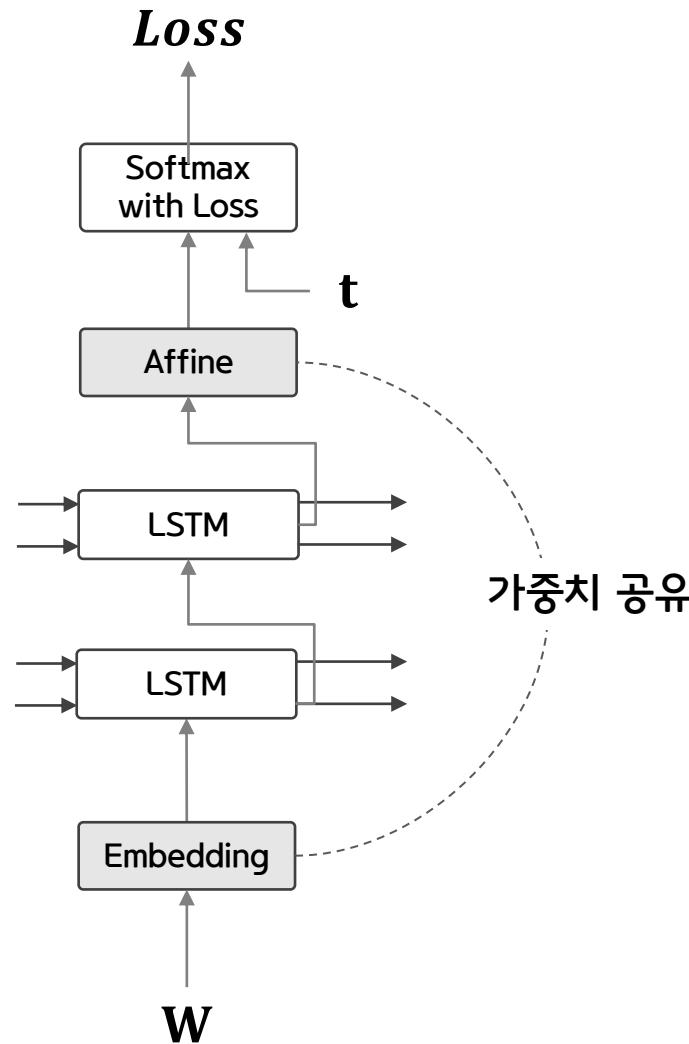
좋은 예: 드롭아웃 계층을 깊이 방향(상하 방향)으로 삽입



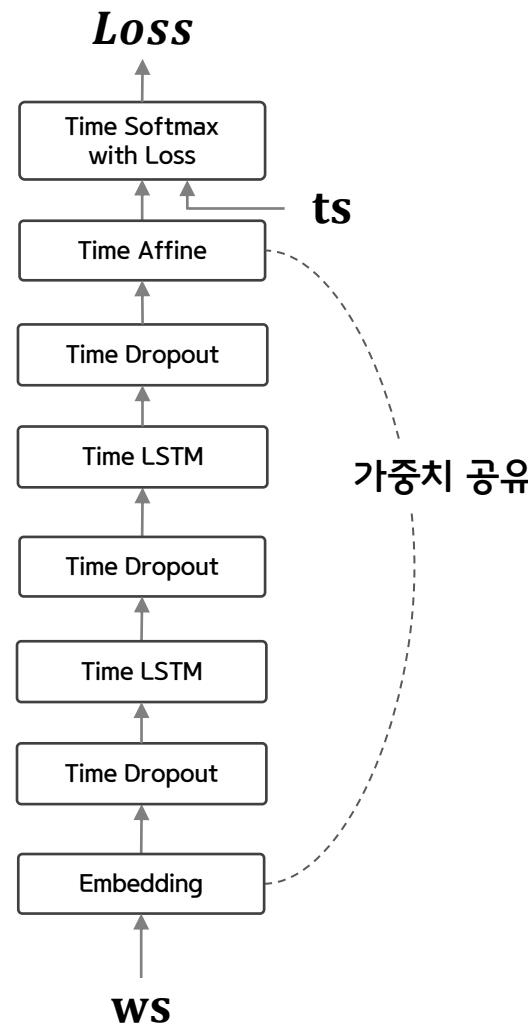
변형 드롭아웃의 예: 색이 같은 드롭아웃 끼리는 같은 마스크를 이용한다. 이처럼 같은 계층에 적용되는 드롭 아웃 끼리는 공통의 마스크를 이용함으로써 시간 방향 드롭아웃도 효과적으로 작동할 수 있다.



언어 모델에서의 가중치 공유 예: Embedding 계층과 Softmax 앞단의 Affine 계층이 가중치를 공유한다.



BetterRnnlm 클래스의 신경망 구성



- LSTM 계층의 다층화(여기서는 2층)
- 드롭아웃 사용(깊이 방향으로만 적용)
- 가중치 공유(Embedding 계층과 Affine 계층에서 가중치 공유)

※ 코드 참조

7. RNN을 사용한 문장 생성

7.1 언어 모델을 사용한 문장 생성

7.2 seq2seq

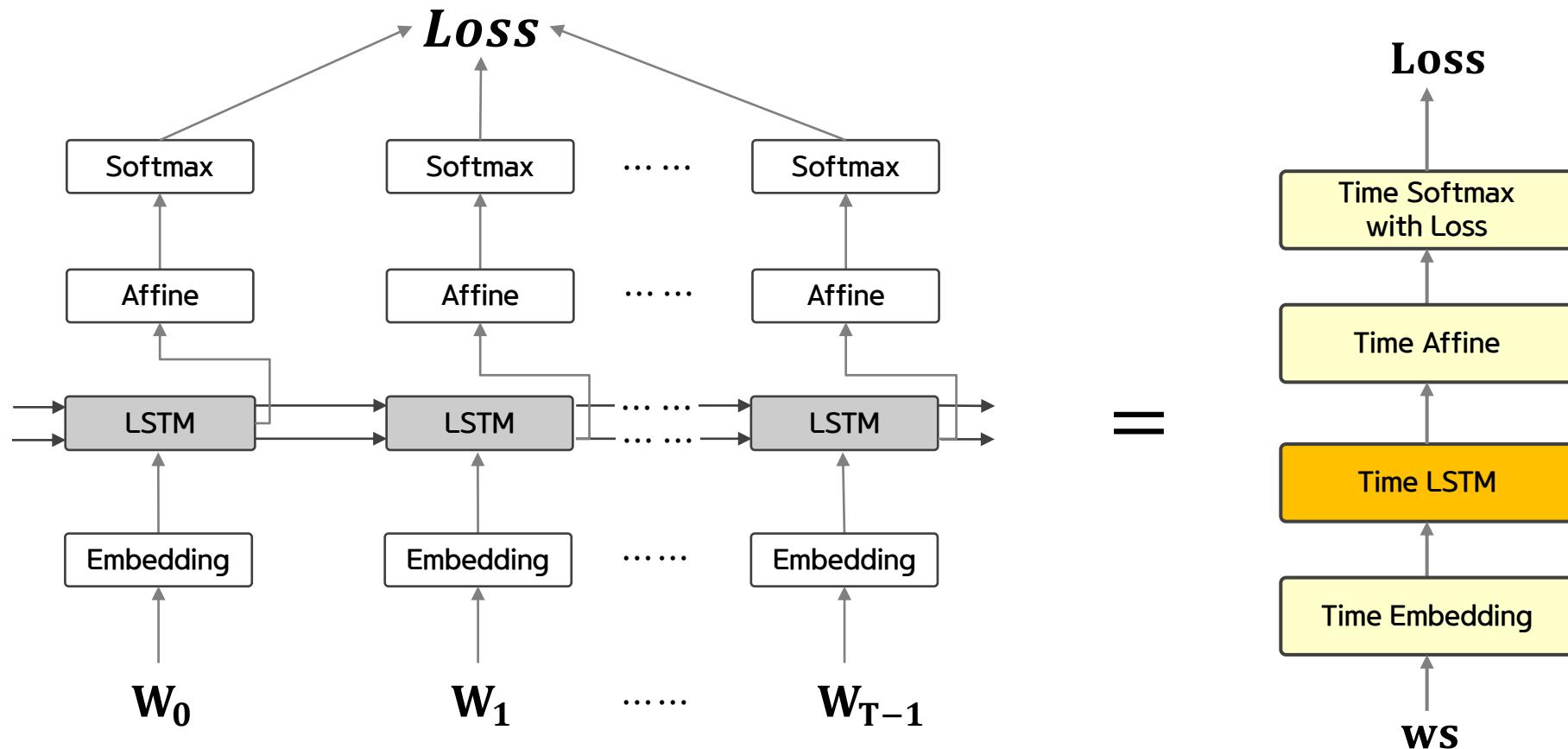
7.3 seq2seq 구현

7.4 seq2seq 개선

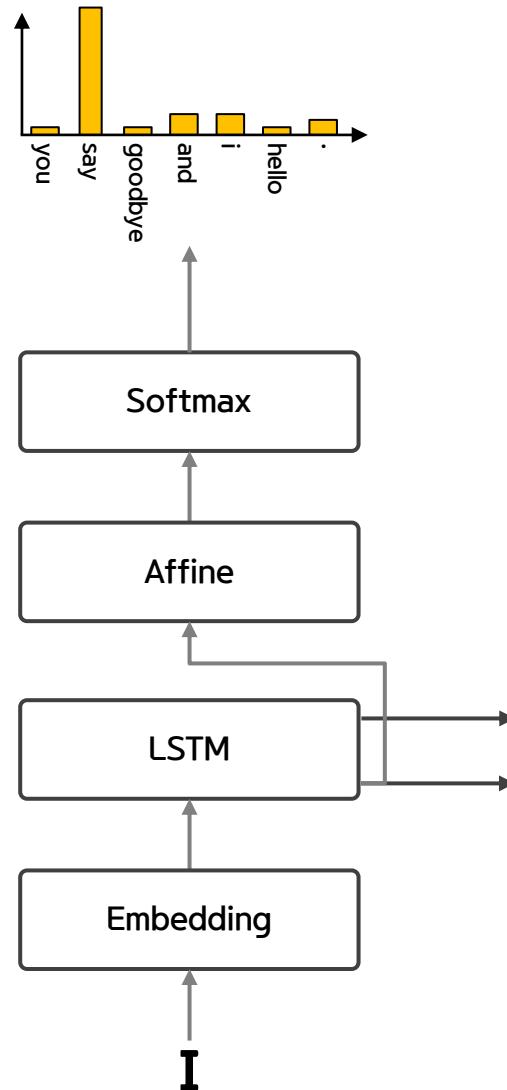
7.5 seq2seq를 이용하는 애플리케이션

앞 장에서는 LSTM 계층을 이용하여 언어 모델을 구현했는데,
그 모델의 신경망 구성은 다음 그림처럼 생겼다.
그리고 시계열 데이터를 T개분 만큼 모아 처리하는 Time LSTM 과 Time Affine 계층 등을 만들었다.

오른쪽은 시계열 데이터를 한꺼번에 처리하는 Time계층을 사용했고, 왼쪽은 같은 구성을 펼친 모습



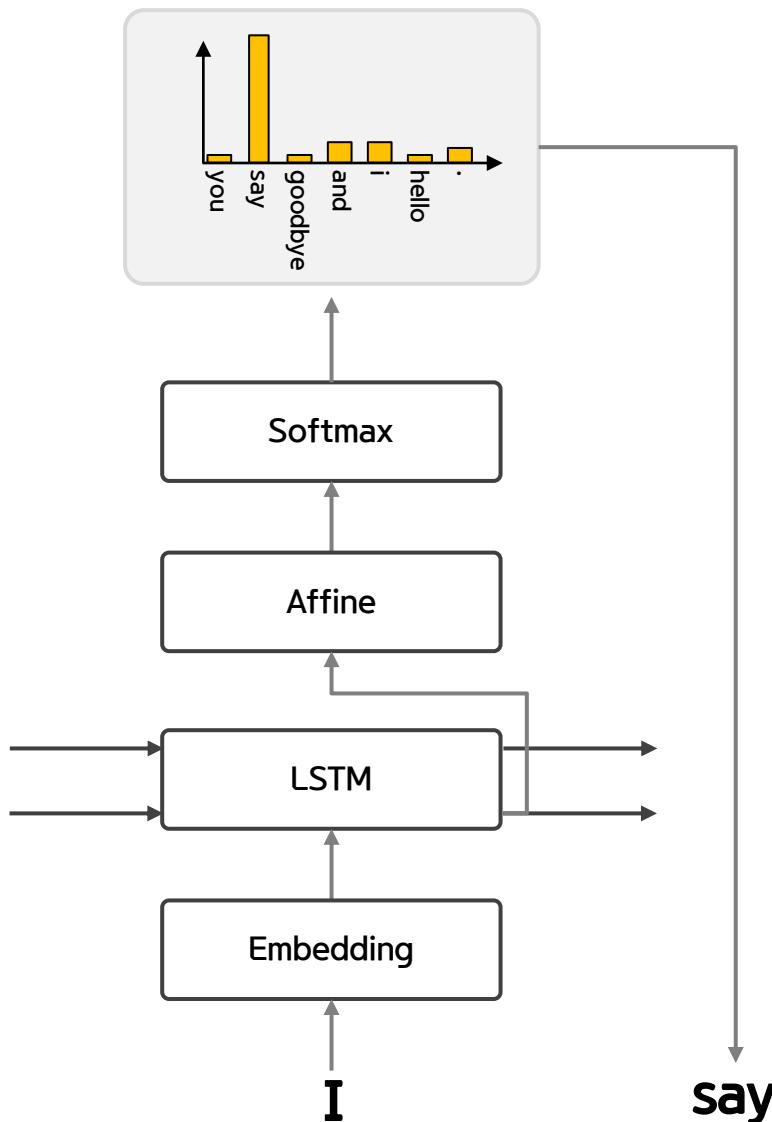
언어 모델은 다음에 출현할 단어의 확률분포를 출력한다.



이제 언어 모델에게 문장을 생성시키는 순서를 설명하겠다.
이번에도 친숙한 "you say goodbye and I say hello."라는
말뭉치로 학습한 언어 모델을 예로 생각해보겠다.

이 학습된 언어 모델에 "I"라는 단어를 입력으로 주면 어떻게 될까?
그러면 이 언어 모델은 다음 그림과 같은 확률 분포를 출력한다고 한다.

확률분포대로 단어를 하나 샘플링한다.



언어 모델은 지금까지 주어진 단어들에서 다음에 출현하는 단어의 확률 분포를 출력한다.
이 결과를 기초로 다음 단어를 새로 생성하려면 어떻게 해야 할까?

첫 번째로, 확률이 가장 높은 단어를 선택하는 방법을 떠올릴 수 있다.
확률이 가장 높은 단어를 선택할 뿐이므로 결과가 일정하게 정해지는 결정적인 방법이다. (?)

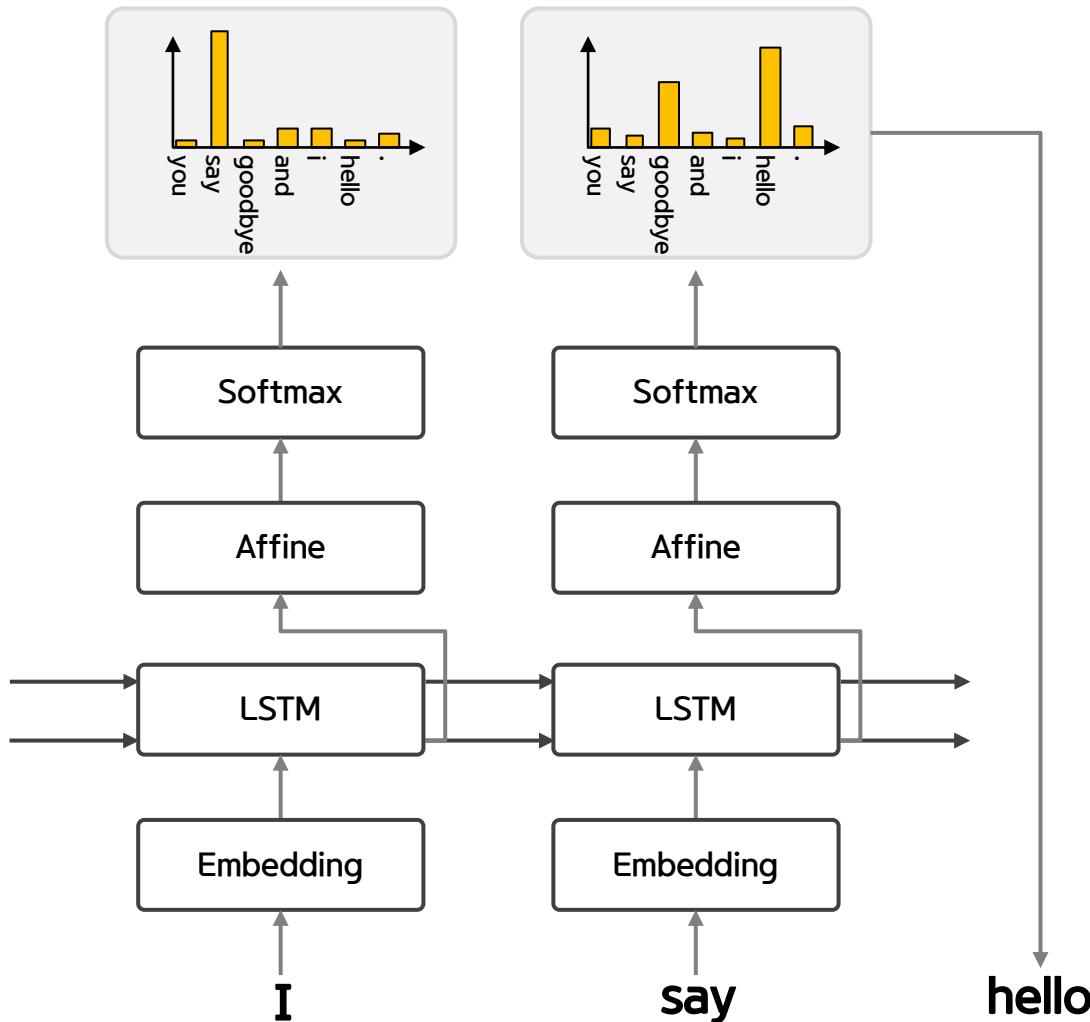
또한, 확률적으로 선택하는 방법도 생각할 수 있다.
각 후보 단어의 확률에 맞게 선택하는 것으로, 확률이 높은 단어는 선택되기 쉽고, 확률이 낮은 단어는 선택되기 어려워진다.

이 방식에서는 선택되는 단어(샘플링 단어)가 매번 다를 수 있다.
우리는 매번 다른 문장을 생성하도록 하겠다.
그 편이 생성되는 문장이 다양해 져서 재미있을 것이다.

RNN을 사용한 문장 생성의 순서

7.1 언어 모델을 사용한 문장 생성

확률분포 출력과 샘플링을 반복한다.



그러면 계속해서 두 번째 단어를 샘플링해보자.
즉, 방금 생성한 단어인 say를 언어 모델에
입력하여 다음 단어의 확률 분포를 얻는다.

그런 다음 그 확률분포를 기초로 다음에
출현할 단어를 샘플링 한다.

다음은 이 작업을 원하는 만큼 반복한다.
그러면 새로운 문장을 생성할 수 있다.

여기에서 주목할 것은 이렇게 생성한 문장은
훈련 데이터에는 존재하지 않는, 말 그대로 새로운
생성된 문장이라는 것이다.

왜냐하면 언어 모델은 훈련 데이터를 암기한 것이
아니라, 훈련 데이터에서 사용된 단어의
정렬 패턴을 학습한 것이기 때문이다.
만약 언어 모델이 말뭉치로부터 단어의 출현 패턴을
올바르게 학습할 수 있다면, 그 모델이 새로
생성하는 문장은 우리 인간에게도 자연스럽고
의미가 통하는 문장일 것으로 기대할 수 있다.

※ 코드 참조

좋은 언어 모델이 있으면 좋은 문장을 기대할 수 있다.

앞 장에서 더 좋은 언어 모델을 BetterRnnIn 라는 클래스로 구현했다.

여기에 문장 생성 기능을 추가하겠다.

이 모델을 한 단계 더 개선하고 한층 더 큰 말뭉치를 사용하면 더 자연스러운 문장을 생성해줄 것이다.

※ 코드 참조

7. RNN을 사용한 문장 생성

7.1 언어 모델을 사용한 문장 생성

7.2 seq2seq

7.3 seq2seq 구현

7.4 seq2seq 개선

7.5 seq2seq를 이용하는 애플리케이션

시계열 데이터는 많다.

언어 데이터, 음성 데이터, 동영상 데이터는 모두 시계열 데이터이다.

그리고 이러한 시계열 데이터를 또 다른 시계열 데이터로 변환하는 문제도 술하게 생각할 수 있다.
예컨대 기계 번역이나 음성 인식을 들 수 있다.

그 외에도 챗봇처럼 대화하는 애플리케이션이나 컴파일러처럼
소스 코드를 기계어로 변환하는 작업도 생각해볼 수 있다.

이처럼 입력과 출력이 시계열 데이터인 문제는 아주 많다.

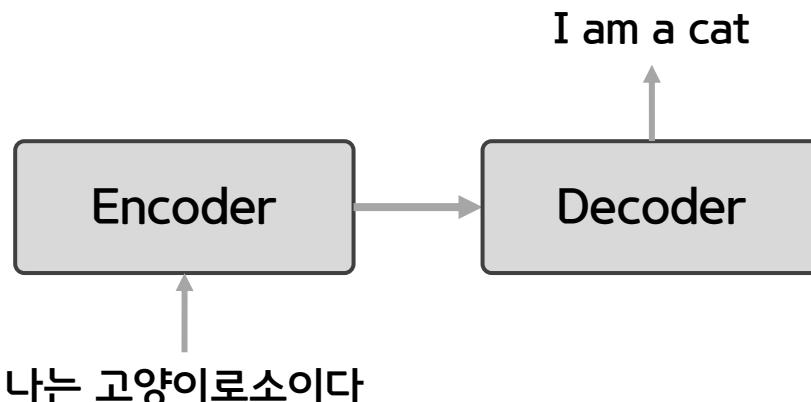
지금부터 우리는 시계열 데이터를 다른 시계열 데이터로 변환하는 모델을 생각해볼 것이다.

이를 위한 기법으로 여기에서는 2개의 RNN을 이용하는 seq2seq 라는 방법을 살펴보겠다.

seq2seq 를 Encoder Decoder 모델이라고도 한다.
여기에는 2개의 모듈, Encoder 와 Decoder 가 등장한다.
문자 그대로 Encoder 는 입력 데이터를 인코딩(부호화)하고
Decoder 는 인코딩된 데이터를 디코딩(복호화)한다.

그럼 seq2seq 의 구조를 구체적인 예를 들어 설명하겠다.
우리말을 영어로 번역하는 예를 살펴보자.
"나는 고양이로소이다" 문장을 "I am a cat"으로 번역해보자.

Encoder와 Decoder가 번역을 수행하는 예



Encoder 가 "나는 고양이로소이다"라는 출발어 문장을 인코딩한다.

이어서 그 인코딩한 정보를 Decoder 에 전달하고, Decoder 가 도착어 문장을 생성한다.

이때 Encoder 가 인코딩한 정보에는 번역에 필요한 정보가 조밀하게 응축되어 있다.

Decoder 는 조밀하게 응축된 정보를 바탕으로 도착어 문장을 생성하는 것이다.

이것이 seq2seq 의 전체 그림이다.

Encoder 와 Decoder 가 협력하여 시계열 데이터를 다른 시계열 데이터로 변환하는 것이다.

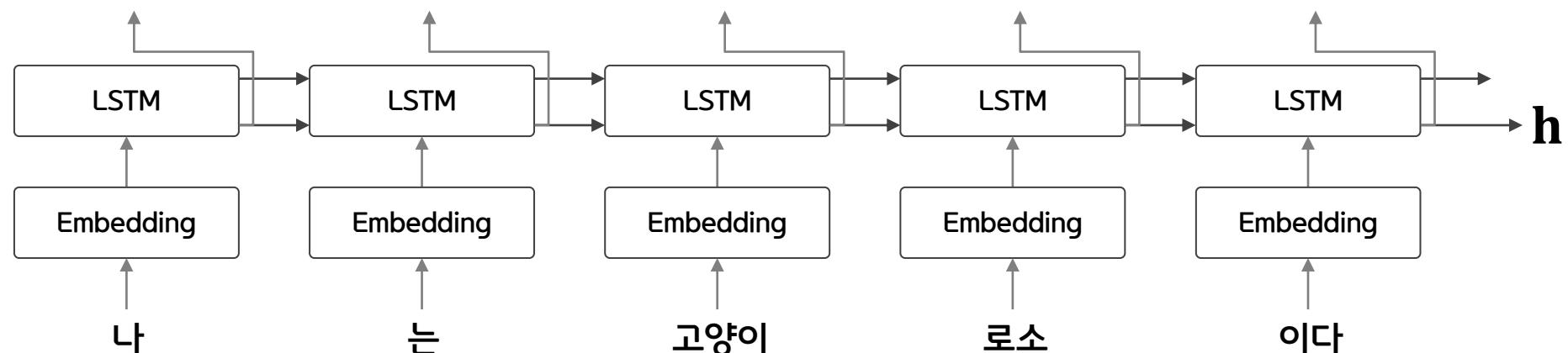
그리고 Encoder 와 Decoder 로 RNN을 사용할 수 있다.

이제 전체 과정을 자세히 살펴보자.

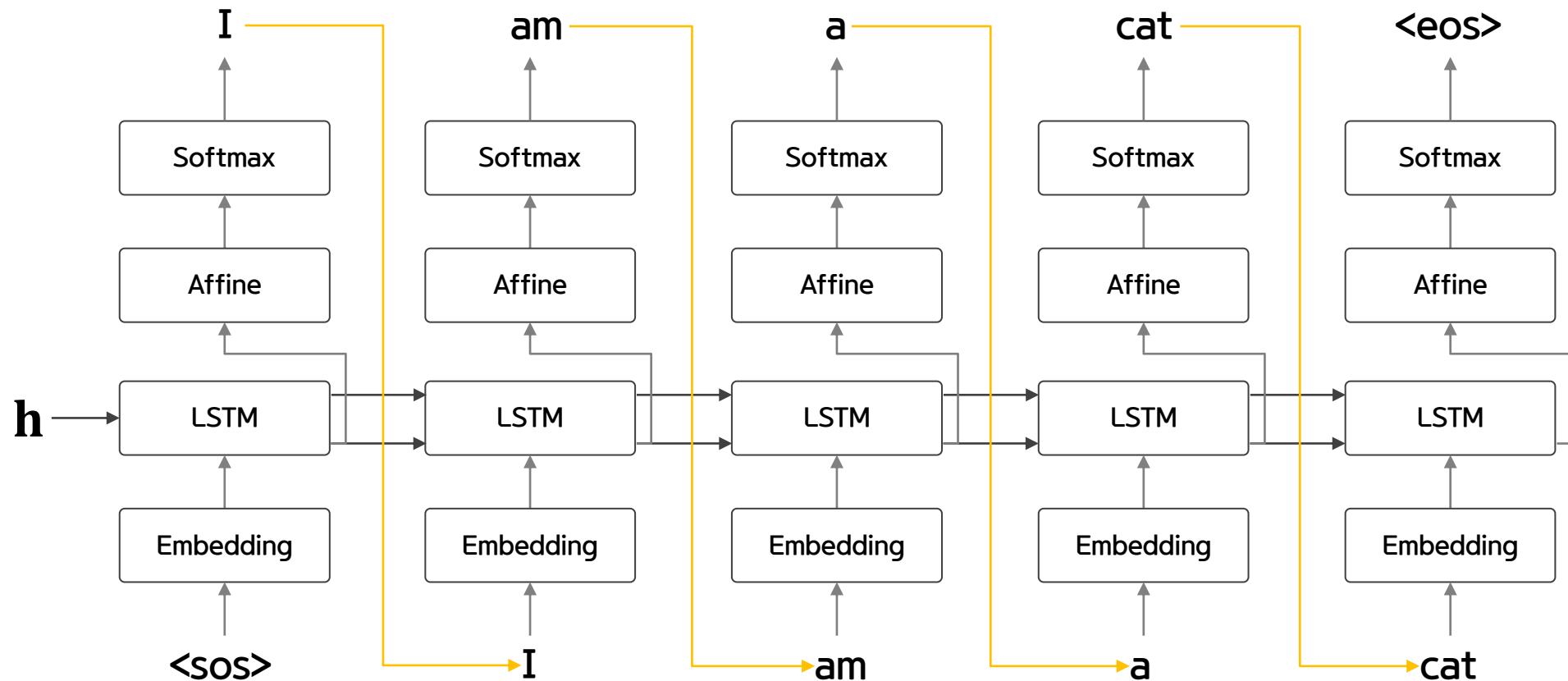
우선 Encoder 의 처리에 집중해보자.

Encoder의 계층을 다음과 같이 구성된다.

Encoder를 구성하는 계층



Decoder를 구성하는 계층



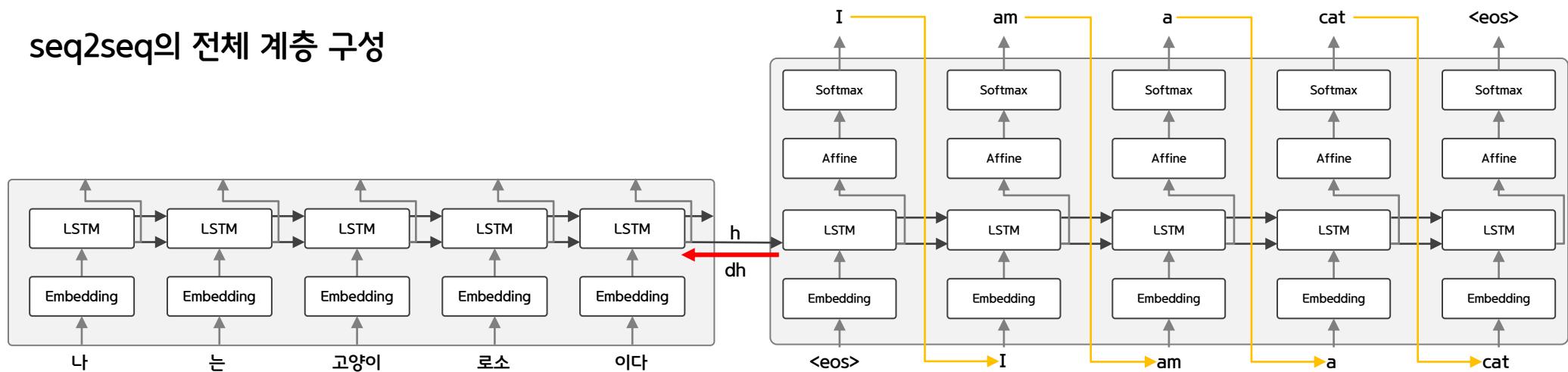
seq2seq 는 LSTM 두 개로 구성된다.
(Encoder 의 LSTM, Decoder 의 LSTM)

이때 LSTM 계층의 은닉 상태가 Encoder 와 Decoder 를 이어주는 가교가 된다.

순전파 때는 Encoder 에서 인코딩된 정보가 LSTM 계층의 은닉 상태를 통해 Decoder 에 전해진다.

그리고 seq2seq 의 역전파 때는 이 가교를 통해 기울기가 Decoder 로부터 Encoder 로 전해진다.

seq2seq의 전체 계층 구성



지금부터, 다른 문제에 관해 설명하겠다.

우리는 시계열 변환 문제의 예로 더하기를 다루었다.

구체적으로는 $57+5$ 와 같은 문자열을 seq2seq 에 건네면 62라는 정답을 내놓도록 학습시킬 것이다.

참고로 이와 같이 머신러닝을 평가하고자 만든 간단한 문제를 장난감 문제라고 한다.

seq2seq에 몇 가지 예제들을 학습시킨다.



셈은 우리 인간에게는 쉬운 문제이다.

그러나 seq2seq 는 덧셈에 대해 (정확하게 덧셈의 논리에 대해) 아무것도 모른다.

seq2seq 는 덧셈의 샘플로부터 거기서 사용되는 문자의 패턴을 학습한다.

과연 이런 식으로 해서 덧셈의 규칙을 올바르게 학습할 수 있는 걸까?

우리는 지금까지 word2vec 이나 언어 모델 등에서 문자를 단어 단위로 분할해왔다.

하지만 문장을 반드시 단어로 분할해야 하는 건 아니다.

실제로 이번 문제에서는 단어가 아닌 문자 단위로 분할한다.

문자 단위 분할이란, 예컨대 $57+5$ 가 입력되면, $[5,7,+,,5]$ 라는 리스트로 처리하는 걸 말한다.

우리는 덧셈을 문자 리스트로써 다루기로 했다.

이때 주의할 점은 덧셈 문장(5,7,+,-)이나 그 대답의 문자 수(6,2)가 문제마다 다르다는 것이다.

이처럼 이번 덧셈 문제에서는 샘플마다 데이터의 시간 방향 크기가 다르다.

가변 길이 시계열 데이터를 다룬다는 뜻이다.

따라서 신경망 학습 시 미니배치 처리를 하려면 무언가 추가 노력이 필요하다.

가변 길이 시계열 데이터를 미니배치로 학습하기 위한 가장 단순한 방법은 패딩을 사용하는 것이다.

패딩이란 원리의 데이터에 의미 없는 데이터를 채워 모든 데이터의 길이를 균일하게 맞추는 기법이다.

미니배치 학습을 위해 '공백 문자'로 패딩을 수행하여 입력·출력 데이터의 크기를 통일한다.

입력

5	7	+	5			
6	2	8	+	5	2	1
2	2	0	+	8		

출력

-	6	2		
-	1	1	4	9
-	2	2	8	

이번 문제에서는 0~999 사이의 숫자 2개만 더하기로 하겠다.

따라서 +까지 포함하면 입력의 최대 문자 수는 7,
자연스럽게 덧셈 결과는 최대 4문자이다.

더불어 정답 데이터에도 패딩을 수행해 모든 샘플 데이터의 길이를 통일한다.

그리고 질문과 정답을 구분하기 위해 출력 앞에 구분자로 _를 붙이기로 한다.

그 결과 출력 데이터는 총 5문자로 통일한다.

참고로, 이 구분자는 Decoder 에 문자열을 생성하라고 알리는 신호로 사용된다. (SOS)

이처럼 패딩을 적용해 데이터 크기를 통일시키면 가변 길이 시계열 데이터도 처리할 수 있다.

그러나 원리는 존재하지 않던 패딩용 문자까지 seq2seq 가 처리하게 된다.

따라서 패딩을 적용해야 하지만 정확성이 중요하다면 seq2seq 에 패딩 전용 처리를 추가해야 한다.

예컨대 Decoder 에 입력된 데이터가 패딩이라면 손실의 결과에 반영하지 않도록 해야 한다.

Softmax with Loss 계층에 마스크 기능을 추가해 해결할 수 있다.

한편 Encoder 에 입력된 데이터가 패딩이라면 LSTM 계층이 이전 시각의 입력을 그대로 출력하게 한다.

즉, LSTM 계층은 마치 처음부터 패딩이 존재하지 않았던 것처럼 인코딩할 수 있다.

이번 장에서는 이해 난이도를 낮추기 위해 패딩용 문자(공백 문자)도 특별히 구분하지 않고
일반 데이터처럼 다루겠다.

'덧셈' 학습 데이터: 공백 문자^{space}는 회색 가운데 점으로 표기

1	16+75..._91...
2	52+607..._659...
3	75+22..._97...
4	63+22..._85...
5	795+3..._798...
6	706+796_1502
7	8+4...._12...
8	84+317..._401...
9	9+3...._12...
10	6+2...._8...
11	18+8..._26...
12	85+52..._137...
13	9+1...._10...
14	8+20..._28...
15	5+3...._8...

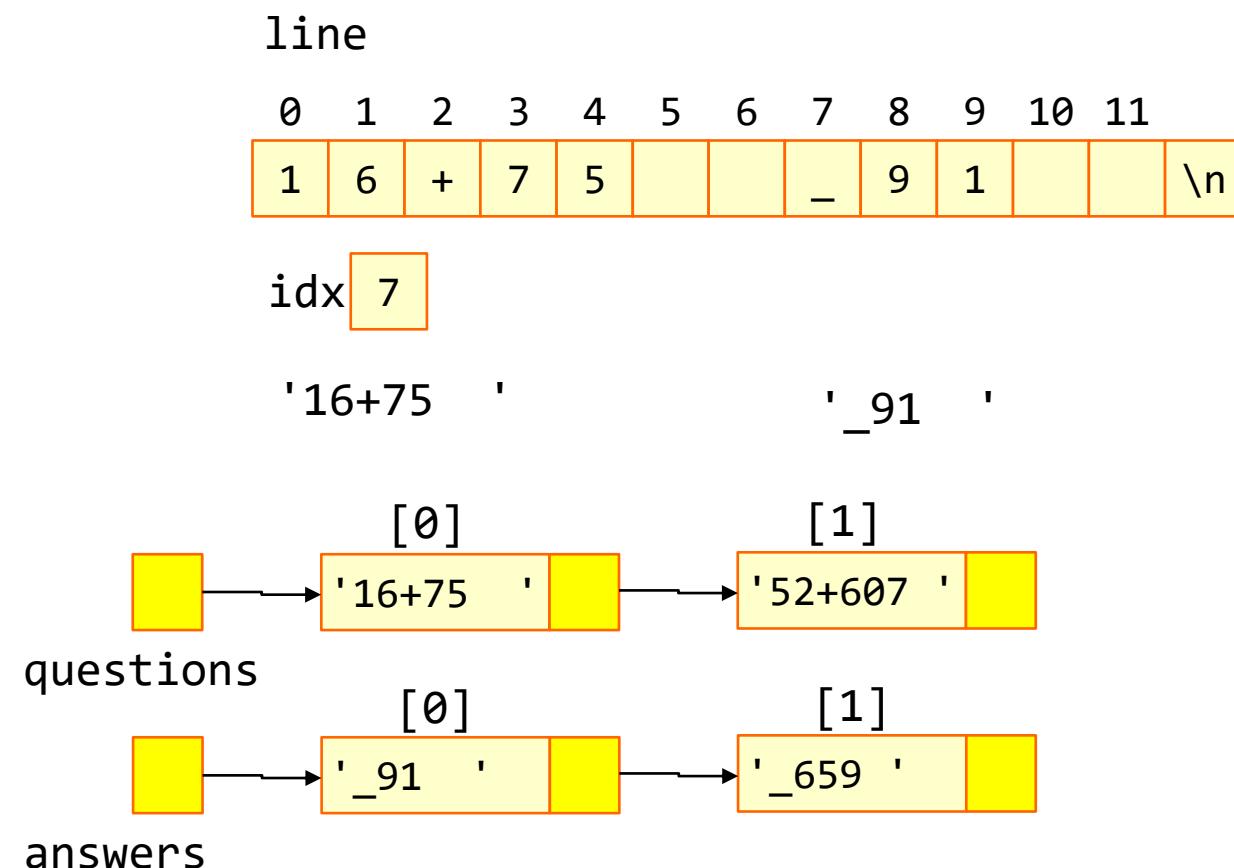
Lines: 50,000 Chars: 650,000 650 KB

※ 코드 참조

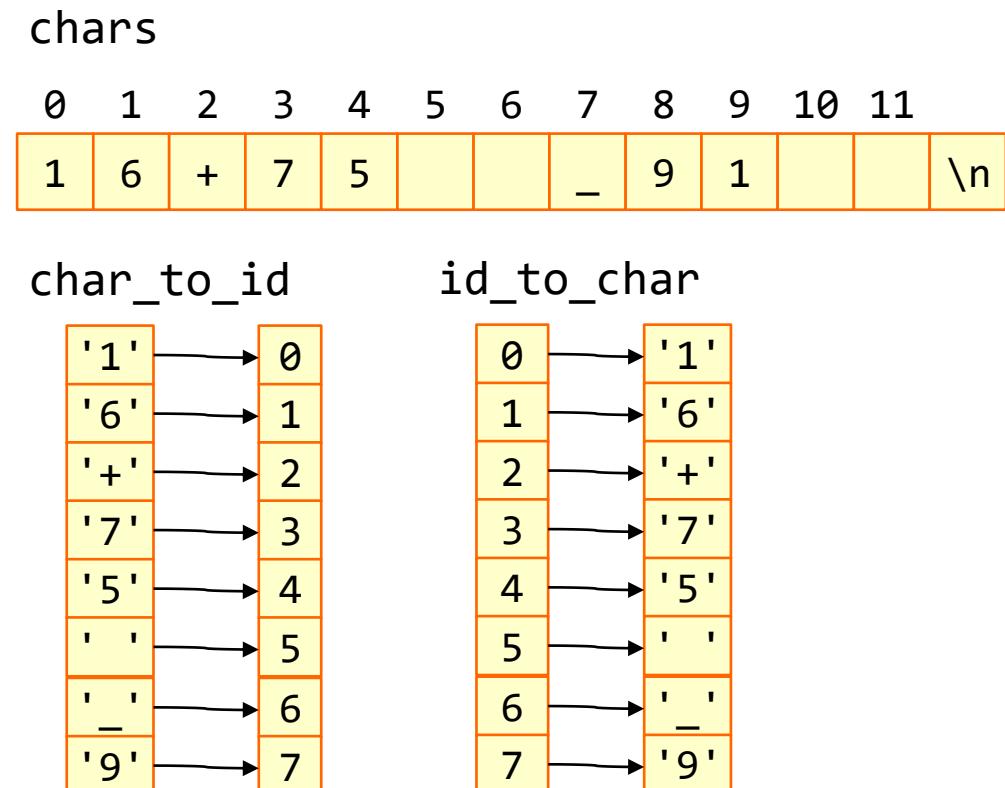
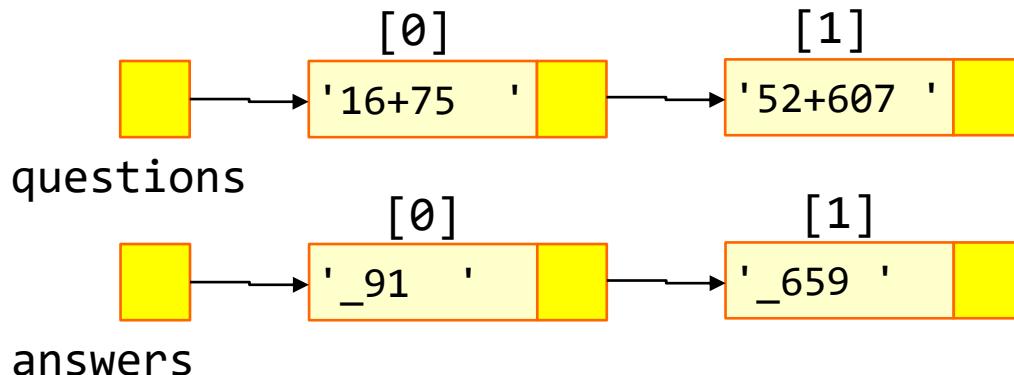
```

for line in open("../dataset/addition.txt", 'r'):
    print(line)
    idx = line.find('_')
#    print(idx)
#    print(line[:idx])
    questions.append(line[:idx])
#    print(questions)
#    print(line[idx:-1])
    answers.append(line[idx:-1])
#    print(answers)

```



```
def _update_vocab(txt):
#    print(txt)
    chars = list(txt)
#    print(chars)
    for i, char in enumerate(chars):
        if char not in char_to_id:
            tmp_id = len(char_to_id)
            char_to_id[char] = tmp_id
            id_to_char[tmp_id] = char
```



7. RNN을 사용한 문장 생성

7.1 언어 모델을 사용한 문장 생성

7.2 seq2seq

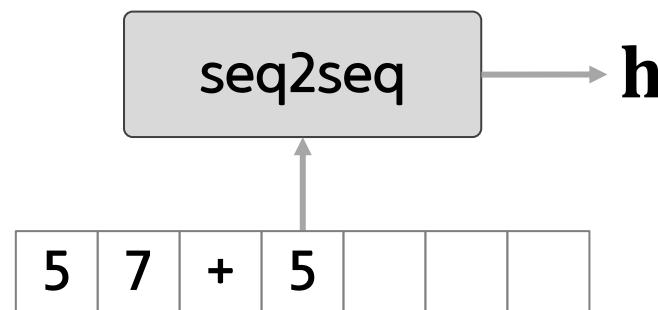
7.3 seq2seq 구현

7.4 seq2seq 개선

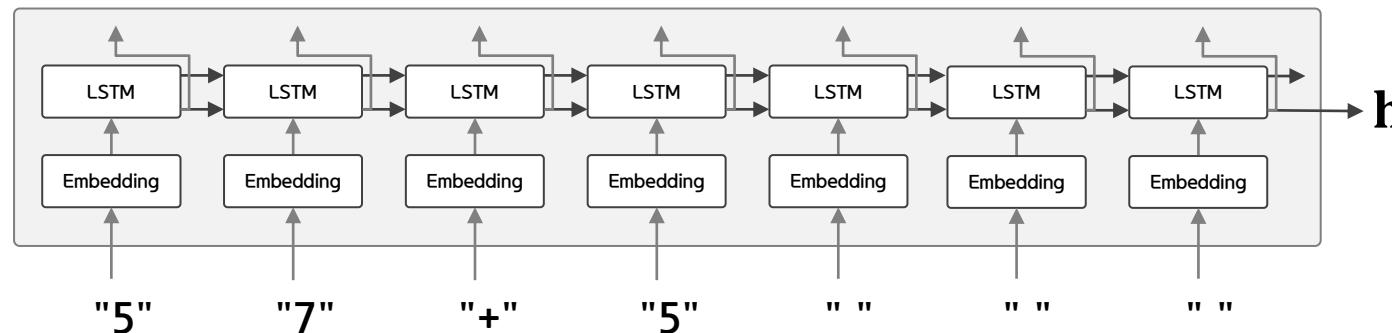
7.5 seq2seq를 이용하는 애플리케이션

Encoder 클래스는 다음 그림처럼 문자열을 받아 벡터 h 로 변환한다.

Encoder의 입력력



Encoder의 계층 구성



Encoder 클래스는 Embedding 계층과 LSTM 계층으로 구성된다.

Embedding 계층에서는 문자 ID를 문자 벡터로 변환한다.
그리고 이 문자 벡터가 LSTM 계층으로 입력된다.

LSTM 계층은 시간 방향(오른쪽)으로 은닉 상태와 셀을 출력하고
위쪽으로는 은닉 상태만 출력한다.

이 구성에서 더 위에는 다른 계층이 없으니 LSTM 계층의 위쪽 출력은 폐기된다.
Encoder 에서는 마지막 문자를 처리한 후 LSTM 계층이 은닉 상태 h 를 출력한다.

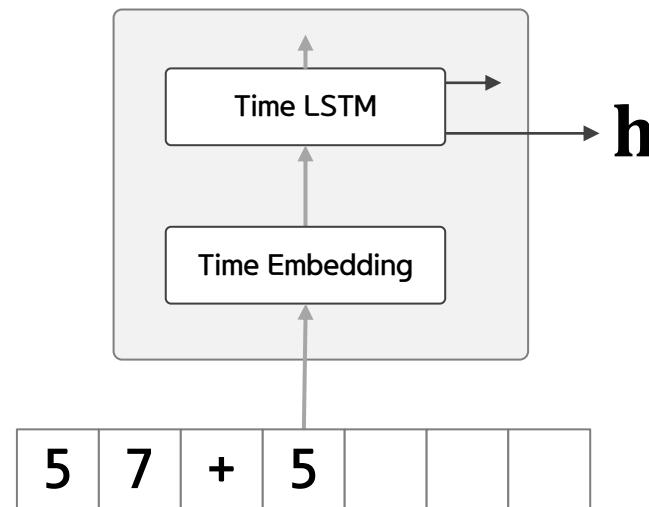
그리고 이 은닉 상태 h 가 Decoder 로 전달된다.

Encoder 에서는 LSTM 의 은닉 상태만을 Decoder 에 전달한다.
LSTM의 셀도 Decoder 에 전달할 수는 있지만,
LSTM 의 셀을 다른 계층에 전달하는 일은 일반적으로 흔치 않다.

LSTM 의 셀은 자기 자신만 사용한다는 전제로 설계되었기 때문이다.

그런데 우리는 시간 방향을 한꺼번에 처리하는 계층을 Time LSTM 계층이나 Time Embedding 계층으로 구현했다.
이러한 Time 계층을 이용하면 Encoder 는 다음 그림처럼 된다.

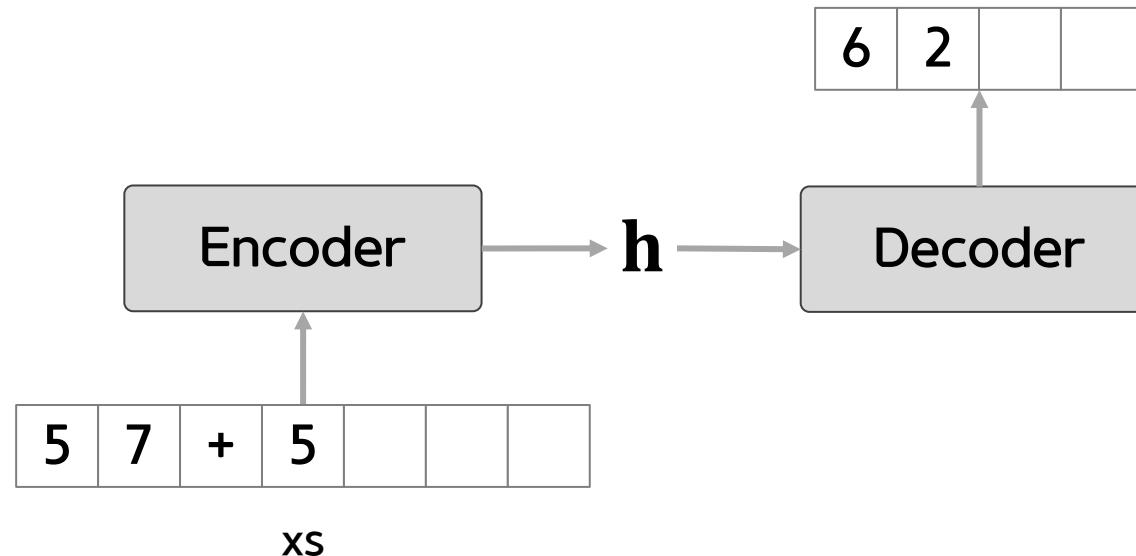
Encoder를 Time 계층으로 구현한다.



※ 코드 참조

Decoder 클래스는 Encoder 클래스가 출력한 h 를 받아 목적으로 하는 다른 문자열을 출력한다.

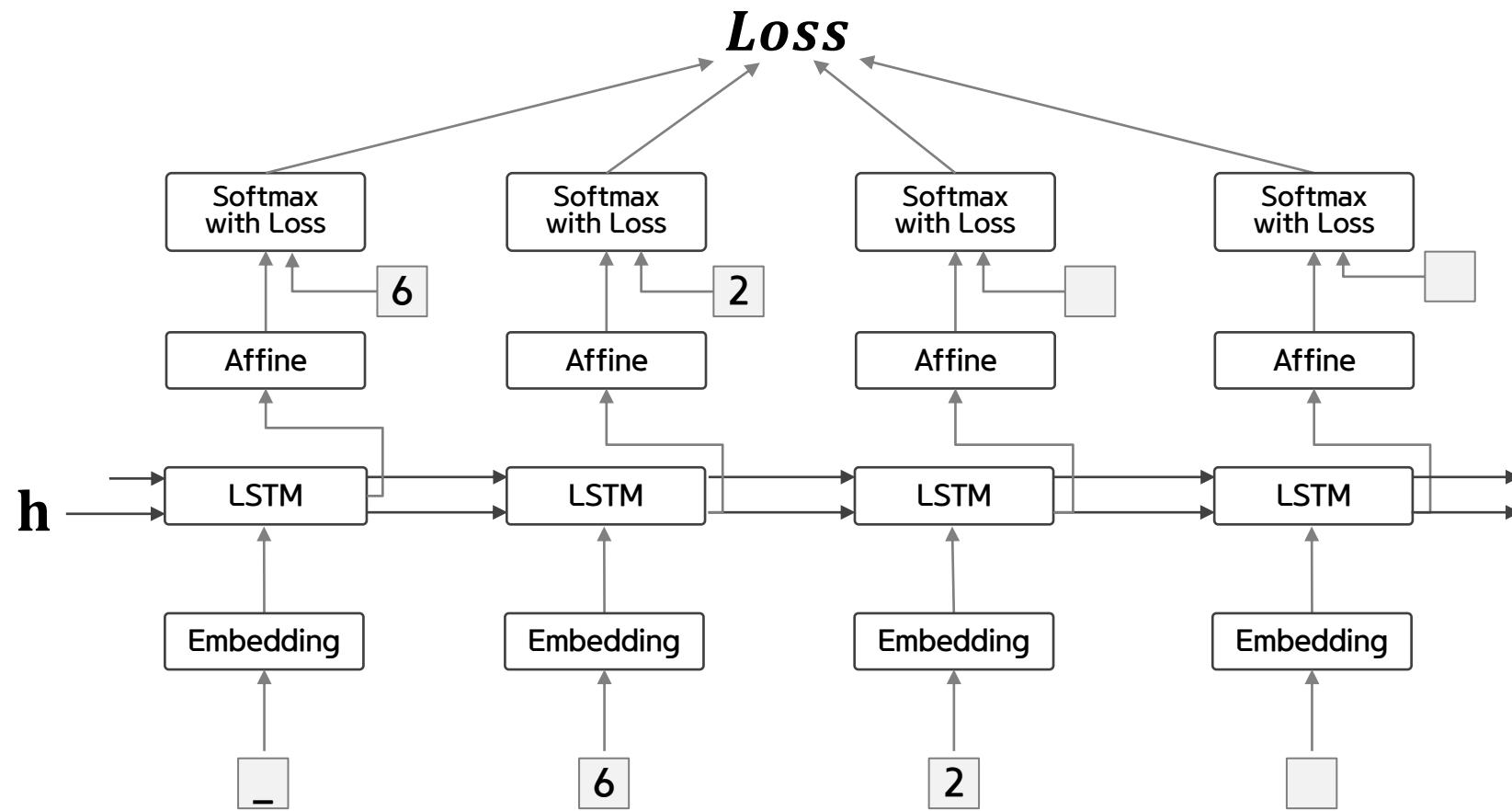
Encoder와 Decoder



Decoder 는 RNN으로 구현할 수 있다.

Encoder 와 마찬가지로 LSTM 계층을 사용하면 되며,
이때 Decoder 의 계층 구성은 다음 그림과 같다.

Decoder의 계층 구성(학습 시)



RNN으로 문장을 생성할 때 학습 시와 생성 시의 데이터 부여 방법이 다르다.

학습 시는 정답을 알고 있기 때문에 시계열 방향의 데이터를 한꺼번에 보여줄 수 있다.

한편, 추론 시(새로운 문자열을 생성할 때)에는 최초 시작을 알리는 구분 문자(여기서는 _) 하나만 준다.

그리고 그 출력으로부터 문자를 하나 샘플링하여, 그 샘플링 문자를 다음 입력으로 사용하는 과정을 반복한다.

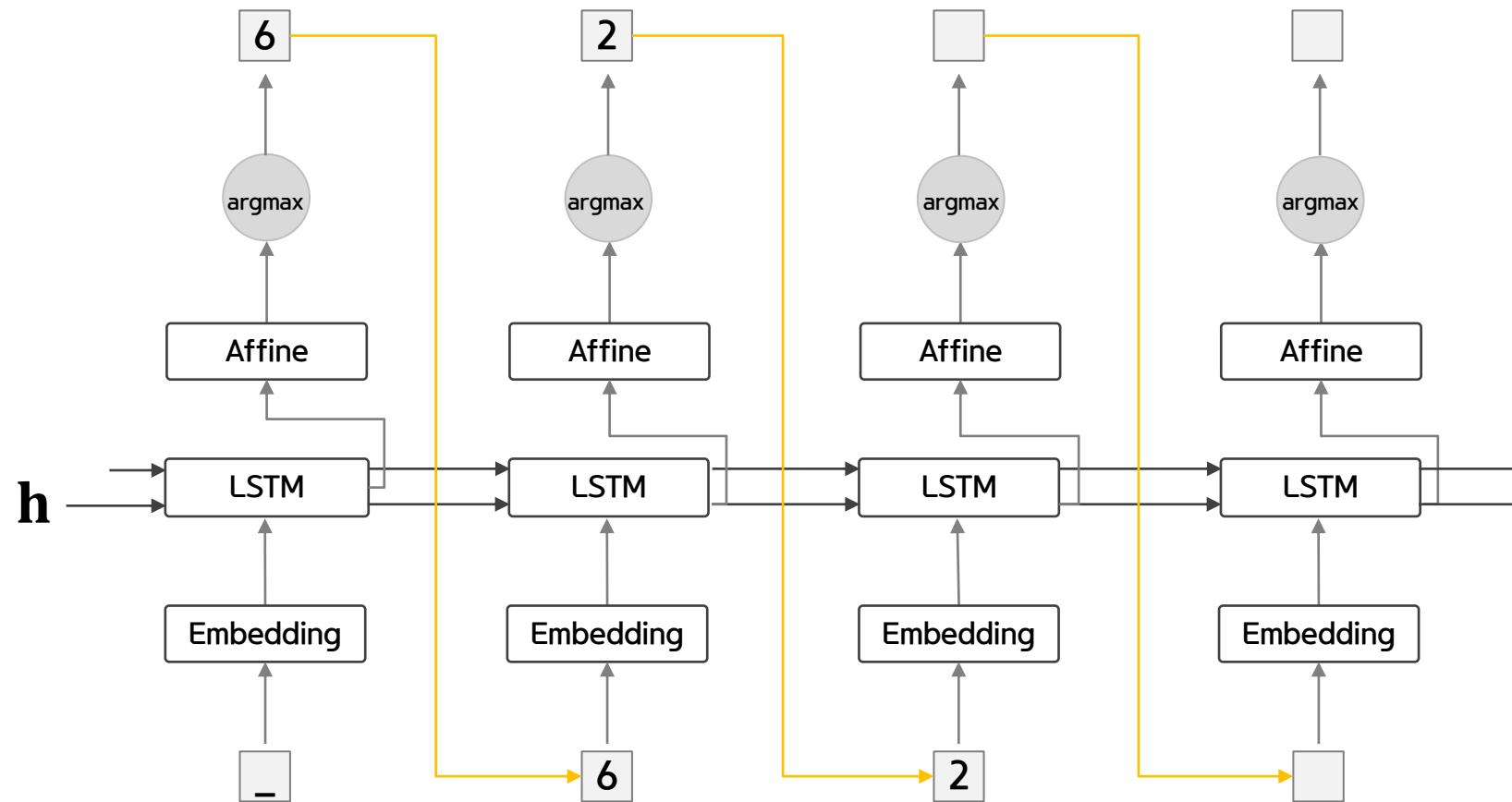
그런데 7.1절에서 문장을 생성할 때는 소프트맥스 함수의 확률분포를 바탕으로 샘플링을 수행했기 때문에 생성되는 문장이 확률에 따라 달라졌다.

이와 달리 이번 문제는 덴셈이므로 이러한 확률적인 비결정성을 배제하고 결정적인 답을 생성하고자 한다.
그래서 이번에는 점수가 가장 높은 문자 하나만 고르겠다.

즉, 확률적이 아닌 결정적으로 선택한다.

다음 그림은 Decoder 가 문자열을 생성시키는 흐름을 보여준다.

Decoder의 문자열 생성 순서: argmax 노드는 Affine 계층의 출력 중 값이 가장 큰 원소의 인덱스(문자ID)를 반환한다.



argmax 라는 못 보던 노드가 등장한다.

바로 최댓값을 가진 원소의 인덱스(여기서는 문자ID)를 선택하는 노드이다.

구성은 앞 절에서 본 문장 생성 때의 구성과 같다.

다만 이번에는 Softmax 계층을 사용하지 않고, Affine 계층이 출력하는 점수가 가장 큰 문자ID 를 선택한다.

Softmax 계층은 입력된 벡터를 정규화한다.

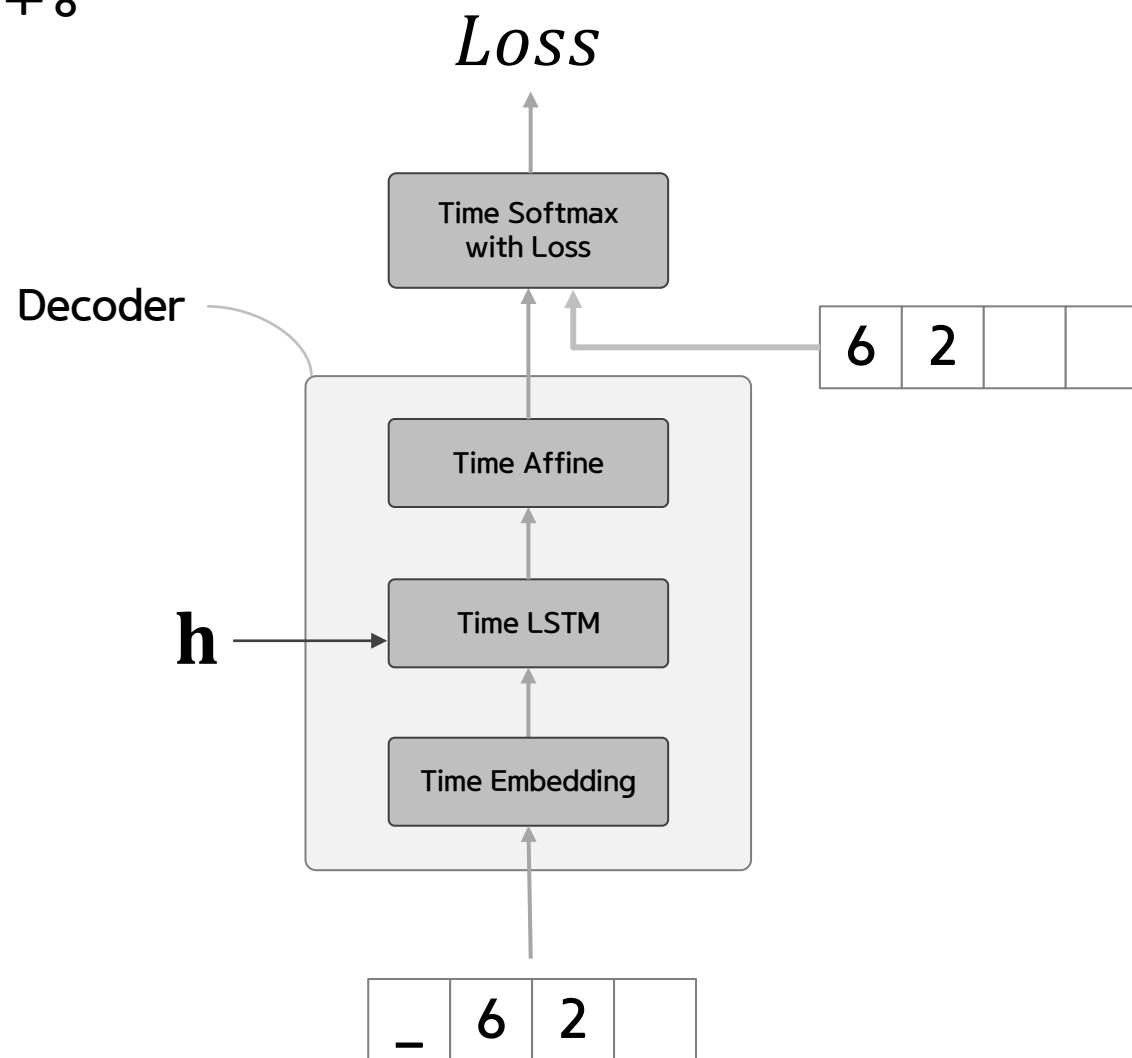
이 정규화 과정에서 벡터의 각 원소의 값이 달라지지만, 대소 관계는 바뀌지 않는다.

따라서 위 그림의 경우 Softmax 계층을 생략할 수 있다.

설명한 것처럼 Decoder 에서는 학습 시와 생성 시에 Softmax 계층을 다르게 취급한다.

그러니 Softmax with Loss 계층은 이후에 구현하는 Seq2seq 클래스에서 처리하기로 하고,
Decoder 클래스는 Time Softmax with Loss 계층의 앞 까지만 담당하기로 한다.

Decoder 클래스의 구성



※ 코드 참조

Seq2seq 클래스는 Encoder 클래스와 Decoder 클래스를 연결하고, Time Softmax with Loss 계층을 이용해 손실을 계산한다.

※ 코드 참조

seq2seq 의 학습은 기본적인 신경망의 학습과 같은 흐름으로 이뤄진다.

1. 학습 데이터에서 미니배치를 선택하고
2. 미니배치로부터 기울기를 선택하고
3. 기울기를 사용하여 매개변수를 갱신한다.

Tainer 클래스를 사용해 이 규칙대로 작업을 수행한다.

매 애플마다 seq2seq 가 테스트 데이터를 풀게 하여(문자열을 생성하여) 학습 중간중간 정답률을 측정한다.

※ 코드 참조

7. RNN을 사용한 문장 생성

7.1 언어 모델을 사용한 문장 생성

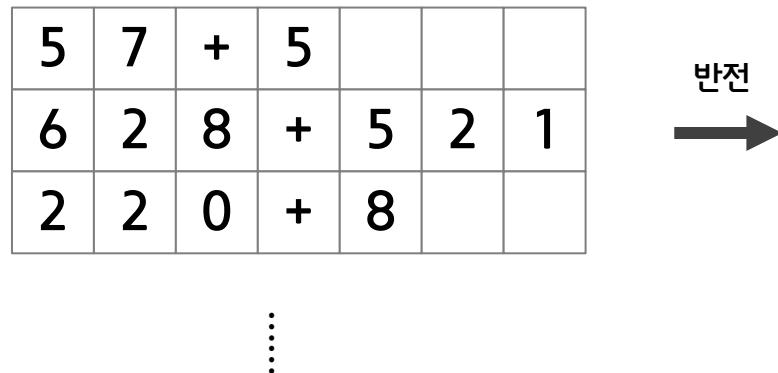
7.2 seq2seq

7.3 seq2seq 구현

7.4 seq2seq 개선

7.5 seq2seq를 이용하는 애플리케이션

입력 데이터 반전시키는 예



입력 데이터를 반전시키는 트릭을 사용하면 많은 경우 학습 진행이 빨라져서,
결과적으로 최종 정확도도 좋아진다고 한다.

그럼 실제로 코드를 살펴보자.

```
x_train, x_test = x_train[:, ::-1], x_test[:, ::-1]
```

물론 데이터를 반전시키는 효과는 어떤 문제를 다루느냐에 따라 다르지만,
대부분의 경우 더 좋은 결과로 이어진다.

그러면 왜 입력 데이터를 반전시키는 것만으로 학습의 진행이 빨라지고 정확도가 향상되는 걸까?
이론적인 것은 잘 모르겠지만, 직관적으로는 기울기 전파가 원활해지기 때문이라고 생각한다.

예를 들어 "나는 고양이로소이다"를 "I am a cat"으로 번역하는 문제에서,
'나'라는 단어가 'I'로 변환되는 과정을 생각해보자.

이때 '나'로부터 'I'까지 가려면 '는', '고양이', '로소', '이다' 까지 총 4 단어 분량의 LSTM 계층을 거쳐야 한다.
따라서 역전파 시, 'I'로부터 전해지는 기울기가 '나'에 도달하기까지, 그 먼 거리만큼 영향을 더 받게 된다.

여기서 입력문을 반전시키면, 즉 "이다 로소 고양이 는" 순으로 바꾸면 어떻게 될까?

이제 '나'와 'I'는 바로 옆이 되었으니 기울기가 직접 전해진다.

이처럼 입력 문장의 첫 부분에서는 반전 덕분에 대응하는 변환 후 단어와 가까우므로 (그런 경우가 많아지므로),
기울기가 더 잘 전해져서 학습 효율이 좋아진다고 생각할 수 있다.

다만, 입력 데이터를 반전해도 단어 사이의 평균적인 거리는 그대로이다.

입력 데이터 반전(Reverse)

7.4 seq2seq 개선

...

Q 761+292

T 1053

X 1052

Q 830+597

T 1427

X 1426

Q 26+838

T 864

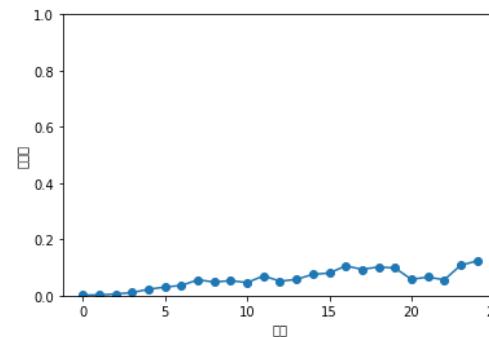
O 864

Q 143+93

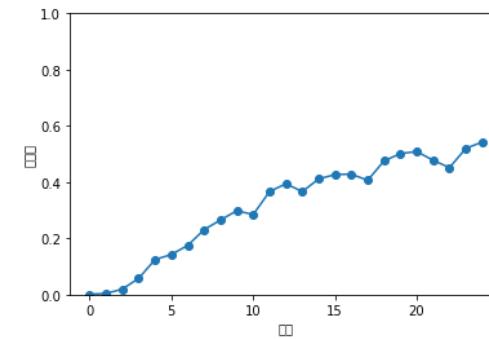
T 236

O 236

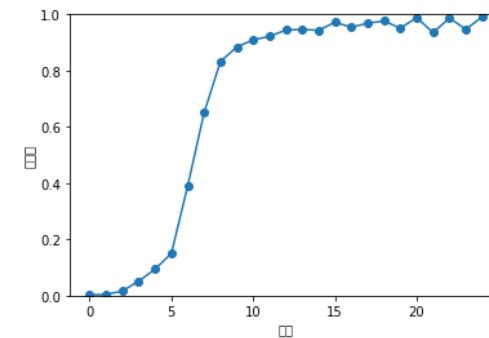
검증 정확도 54.260%



reverse 전



reverse 후



peeky 적용 후

이어서 seq2seq 의 두 번째 개선이다.

주제로 곧장 들어가기 전에 seq2seq의 Encoder 동작을 한번 더 살펴보자.

Encoder는 입력 문장(문제 문장)을 고정 길이 벡터 h 로 변환한다.

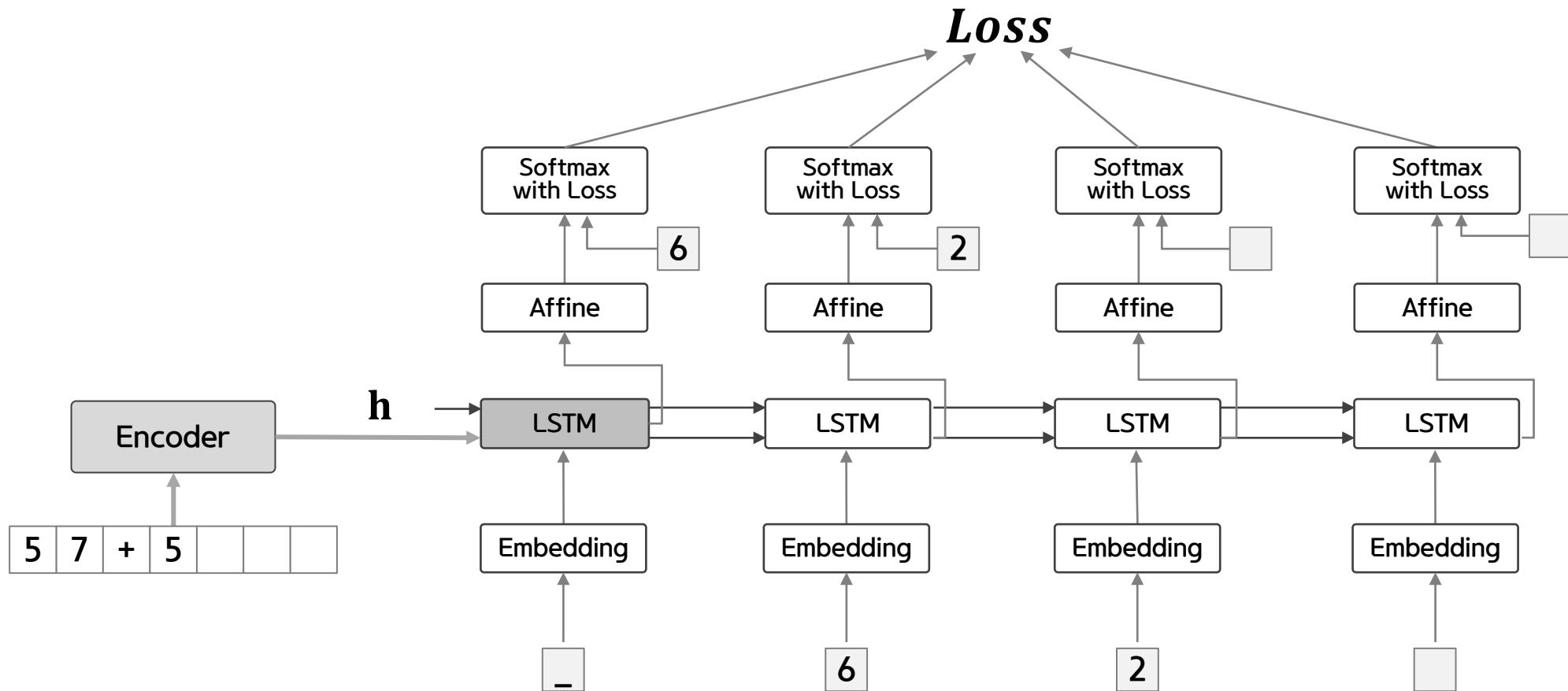
이때 h 안에는 Decoder 에게 필요한 정보가 모두 담겨 있다.

즉, h 가 Decoder 에 있어서는 유일한 정보인 셈이다.

그러나 현재의 seq2seq 는 다음 그림과 같이 최초 시각의 LSTM 계층만이 벡터 h 를 이용하고 있다.

이 중요한 정보인 h 를 더 활용할 수는 없을까?

개선 전: Encoder의 출력 h 는 첫 번째 LSTM 계층만이 받는다.



개선 후: Encoder의 출력 h 를 모든 시각의 LSTM 계층과 Affine 계층에 전해준다.

```
vocab_size = len(char_to_id)
```

```
wordvec_size = 16
```

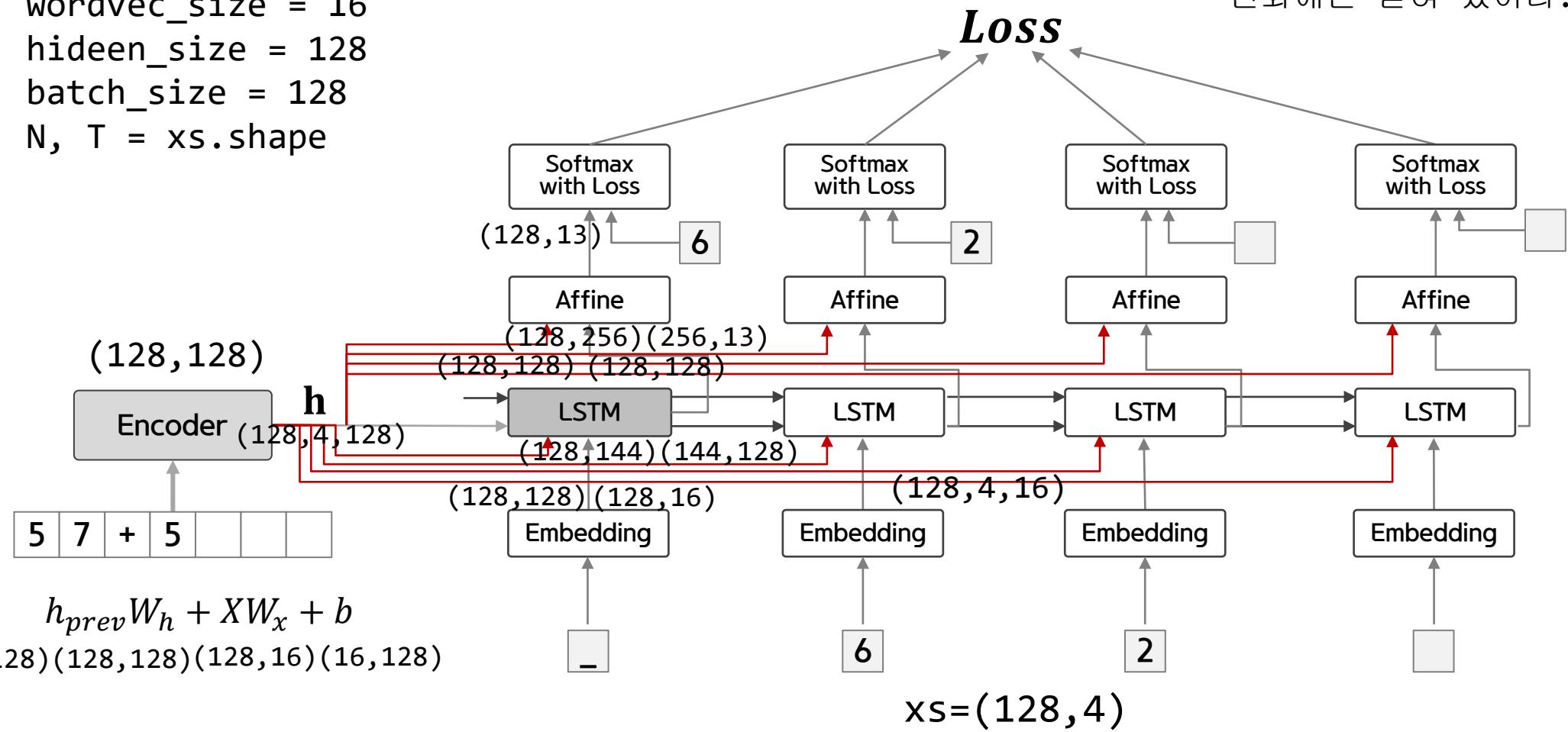
```
hdeen_size = 128
```

```
batch_size = 128
```

```
N, T = xs.shape
```

OCP

확장에는 열려있고
변화에는 닫혀 있어라.



$$h_{prev}W_h + XW_x + b$$

$$(3, 128)(128, 128)(128, 16)(16, 128)$$

그림과 같이 모든 시각의 Affine 계층과 LSTM 계층에 Encoder 의 출력 h 를 전해준다.
이전 그림과 비교해보면, 기존에는 하나의 LSTM 만이 소유하던 중요 정보 h 를
여러 계층(예에서는 8계층)이 공유함을 알 수 있다.
이는 집단 지성에 비유할 수 있다.
즉, 중요한 정보를 한 사람이 독점하는 것이 아니라,
많은 사람과 공유한다면 더 올바른 결정을 내릴 가능성이 커질 것이다.

이 개선안은 인코딩된 정보를 Decoder 의 다른 계층에도 전해주는 기법이다.
달리 보면, 다른 계층도 인코딩된 정보를 엿본다고 해석할 수 있다.
엿보다를 영어로 peek 이라고 하기 때문에 이 개선을 더한 Decoder 를 Peeky Decoder 라고 한다.
마찬가지로 Peeky Decoder 를 이용하는 seq2seq를 Peeky seq2seq 라고 한다.

앞 그림에서는 LSTM 계층과 Affine 계층에 입력되는 벡터가 2개씩이 되었다.
이는 실제로는 두 벡터가 연결된 것을 의미한다.
따라서 두 벡터를 연결시키는 concat 노드를 이용해

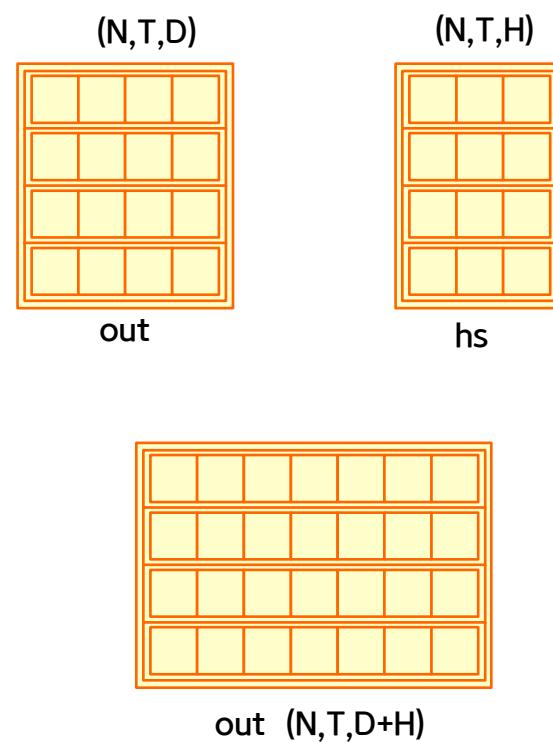
다음 그림처럼 그려야 정확한 계산 그래프가 된다.

Affine 계층에 입력이 2개인 경우(왼쪽)를 정확하게 그리면, 그 두 입력을 연결한 하나의 벡터가 입력되는 것이다(오른쪽).

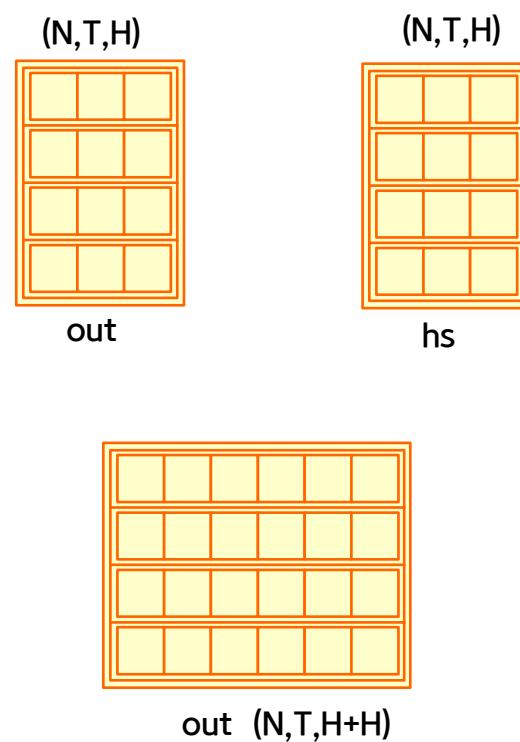


※ 코드 참조

```
out = self.embed.forward(xs)
hs = np.repeat(h, T, axis=0).reshape(N, T, H)
out = np.concatenate((hs, out), axis=2)
```



```
out = self.lstm.forward(out)
out = np.concatenate((hs, out), axis=2)
```



Peeky 를 추가로 적용하자 seq2seq 의 결과가 월등히 좋아졌다.

이상의 실험 결과에서 Reverse 와 Peeky 가 함께 효과적으로 작동하고 있음을 알 수 있다.

입력 문장을 반전시키는 Reverse, 그리고 Encoder 의 정보를 널리 퍼지게 하는

Peeky 덕분에 만족할 만한 결과를 얻었다.

여기서 수행한 개선은 작은 개선이라 할 수 있다.

큰 개선은 다음 장에서 추가할 계획이다.

바로 어텐션이라는 기술로,

seq2seq 를 극적으로 진화시킬 수 있다.

이번 절의 실험은 주의해야 한다.

Peeky 를 이용하게 되면, 신경망은 가중치 매개변수가 커져서 계산량도 늘어난다.

따라서 이번 절의 실험 결과는 커진 매개변수만큼의 핸디캡을 감안해야 한다.

또한 seq2seq 의 정확도는 하이퍼 파라미터에 영향을 크게 받는다.

예제에서의 결과는 믿음직했지만, 실제 문제에서는 그 효과가 달라질 것이다.

7. RNN을 사용한 문장 생성

7.1 언어 모델을 사용한 문장 생성

7.2 seq2seq

7.3 seq2seq 구현

7.4 seq2seq 개선

7.5 seq2seq를 이용하는 애플리케이션

seq2seq 는 한 시계열 데이터를 다른 시계열 데이터로 변환한다.
이 시계열 데이터를 변환하는 프레임워크는 다양한 문제에 적용할 수 있다.
다음은 구체적인 예이다.

- 기계 번역: 한 언어의 문장을 다른 언어의 문장으로 변환
- 자동 요약: 긴 문장을 짧게 요약된 문장으로 변환
- 질의응답: 질문을 응답으로 변환
- 메일 자동 응답: 받은 메일의 문장을 답변 글로 변환

seq2seq 는 2개가 짹을 이루는 시계열 데이터를 다루는 문제에 이용할 수 있다.
자연어 외에도 음성이나 영상 등에도 이용할 수 있다.
얼핏 보기에는 seq2seq 가 적용될 수 없을 것 같은 문제라도 입력/출력 데이터를
전처리하면 seq2seq 를 적용할 수 있는 경우도 있다.

이번 절에서는 seq2seq 를 사용한 애플리케이션을 몇 가지 소개한다.

챗봇은 사람과 컴퓨터가 텍스트로 대화를 나누는 프로그램이다.

챗봇에도 seq2seq를 사용할 수 있다.

대화라는 것은 상대의 말과 자신의 말로 구성되기 때문에

상대의 말을 자신의 말로 변환하는 문제로 볼 수 있다.

즉, 대화의 텍스트 데이터가 준비되면 그것으로 seq2seq 를 학습시킬 수 있다.

seq2seq 기반의 챗봇의 대화 예

기계: 안녕하세요.

사람: 안녕하세요.

기계: 어떤 OS를 사용하시죠?

사람: 리눅스요.

.....

기계: 실행할 때 어떤 오류가 발생하죠?

사람: VPN 어쩌고 하는 오류가 나옵니다.

기계: VPN을 사용하여 네트워크에 연결할 때 오류 메시지가 뭐라고 뜨나요?

사람: 연결이 되지 않거나 혹은 그와 비슷한 상태...

.....

기계: 지금 집인가요, 아니면 회사인가요?

사람: 집이요.

기계: 그럼, <URL>에 가서 선생님의 이름과 암호를 입력해보세요.

사람: 했어요.

기계: <NAME>에서 로그인하여 잘 되는지 확인해보세요.

사람: 오, 잘 되네요!

기계: 좋습니다. 다른 문제는 없나요?

사람: 없어요, 매우 잘 됩니다.

파이썬으로 작성된 코드의 예

```
input:  
    j=8584  
    for x in range(8):  
        j+=920  
    b=(1500+j)  
    print((b+7567))  
target: 25011
```

```
input:  
    i=8827  
    c=(i-5347)  
    print(((c+8704)) if 2641<8500 else 5308)  
target: 12184
```

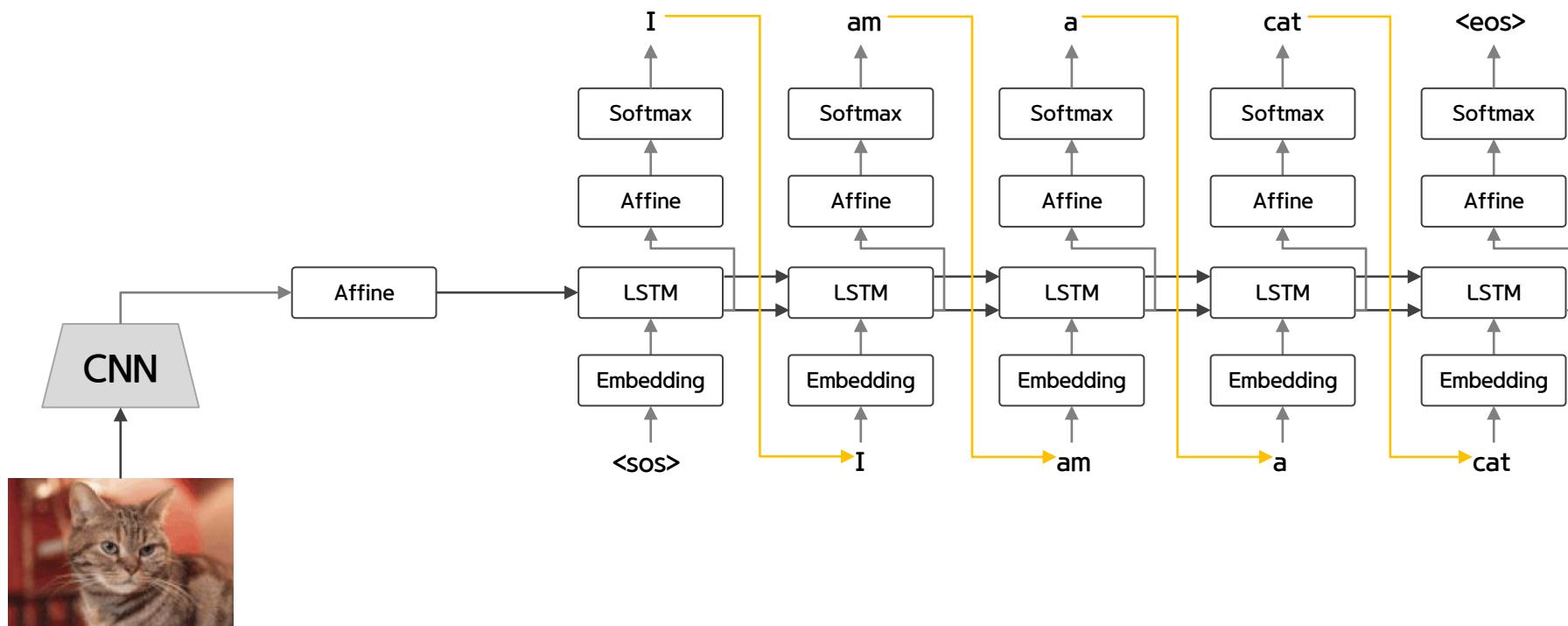
지금까지는 seq2seq 가 텍스트를 다루는 예만을 보았다.

하지만 seq2seq 는 텍스트 외에도, 이미지나 음성 등 다양한 데이터를 처리할 수 있다.

이번절에서는 이미지를 문장으로 변환하는 이미지 캡셔닝을 살펴본다.

이미지 캡셔닝은 이미지를 문장으로 변환한다.

이 문제도 다음 그림과 같이 seq2seq 의 틀에서 해결할 수 있다.



그림은 우리에게 친숙한 신경망 구조이다.

지금 까지와 다른 점은 Encoder 가 LSTM 에서 합성곱 신경망(CNN)으로 바뀐게 전부다.

겨우 LSTM을 CNN으로 대체한 것 만으로 seq2seq 는 이미지도 처리할 수 있다.

그림에서 CNN 에 대해 살짝 보충해보자.

이 예에서는 이미지의 인코딩을 CNN이 수행한다. 이때 CNN의 최종 출력은 특징 맵이다.

특징 맵은 3차원(높이, 폭, 채널)이므로, 이를 Decoder 의 LSTM이 처리할 수 있도록 손질해야 한다.

그래서 CNN의 특징 맵을 1차원으로 평탄화한 후 완전연결인 Affine 계층에서 변환한다.

그런 다음 변환된 데이터를 Decoder 에 전달하면, 문장 생성을 수행할 수 있다.

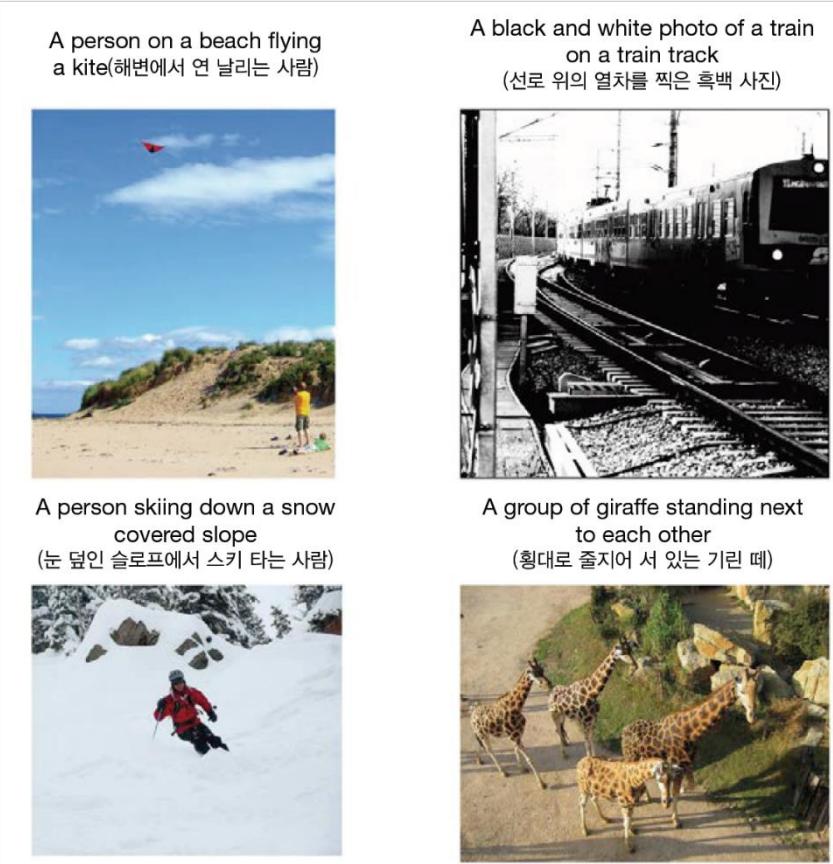
위 그림의 CNN에 CGG나 ResNet 등의 입증된 신경망을 사용하고,

가중치로는 다른 이미지 데이터넷으로 학습을 끝낸 것을 이용한다.

이렇게 하면 좋은 인코딩을 얻을 수 있고, 좋은 문장을 생성할 수 있다.

이제 seq2seq 가 이미지 캡셔닝을 수행한 예를 몇 가지 살펴보자.
여기에서 사용된 신경망은 앞 그림을 기초로 한 것이다.

이미지 캡셔닝의 예: 이미지를 텍스트로 변환



주제: RNN을 이용한 문장 생성

6장에서 다른 RNN을 사용한 언어 모델을 손질하여,
문장 생성 기능을 추가했다.

또한 seq2seq에게 간단한 덧셈 문제를 학습시키는 데 성공했다.

seq2seq는 Encoder와 Decoder를 연결한 모델로,
결국 2개의 RNN을 조합한 단순한 구조이다.

하지만 단순함에도 불구하고, seq2seq는 매우 큰 가능성을 지니고 있어서 다양한 애플리케이션에 적용할 수 있다.

이번 장에서는 seq2seq를 개선하는 아이디어 2개를 살펴봤다.

1. Reverse
2. Peeky

다음장에서는 seq2seq를 한 층 더 개선할 것이다. (어텐션)
어텐션은 딥러닝에서 가장 중요한 기법 중 하나이다.

다음장에서는 어텐션 매커니즘을 설명하고 구현하여,
한층 더 강력한 seq2seq를 구현한다.

8. 어텐션

8.1 어텐션의 구조

8.2 어텐션을 갖춘 seq2seq 구현

8.3 어텐션 평가

8.4 어텐션에 관한 남은 이야기

8.5 어텐션 응용

seq2seq 란, 2개의 RNN 을 연결하여 하나의 시계열 데이터를 다른 시계열 데이터로 변환하는 것이다.

이번 장에서는, seq2seq 와 RNN 의 가능성을 높여주는,

어텐션 이라는 기술에 대해 학습해보자.

어텐션 매커니즘은 seq2seq 를 더 강력하게 해준다.

seq2seq가 인간처럼 필요한 정보에만 주목할 수 있게 해주고,
기존의 seq2seq 가 갖고있던 문제점도 해결가능하다.

seq2seq의 문제점은 무엇일까?

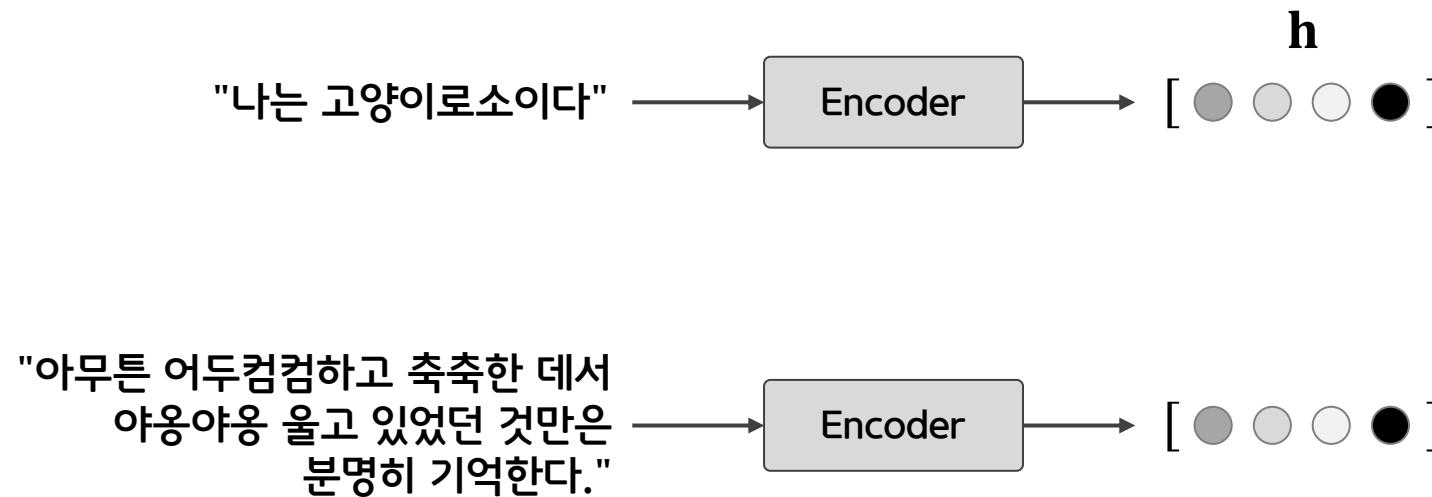
seq2seq 에서 encoder 가 시계열데이터를 인코딩하고, 그 인코딩된 정보를 decoder로 전달하는데,
이때 encoder 의 출력은 '고정 길이의 벡터'였다. 이 '고정 길이'에 문제점이 잠재해 있는 것이다.

아무리 입력 문장의 길이가 길다 하더라도 같은 길이의 벡터로 변환한다는 뜻이다.

이는 결국, 필요한 정보가 벡터에 다 담기지 못하는 문제가 생긴다.

그래서 우선 encoder를 개선하고 이어서 decoder 도 개선해야 한다.

입력 문장의 길이에 관계없이, Encoder는 정보를 고정 길이의 벡터로 밀어 넣는다.



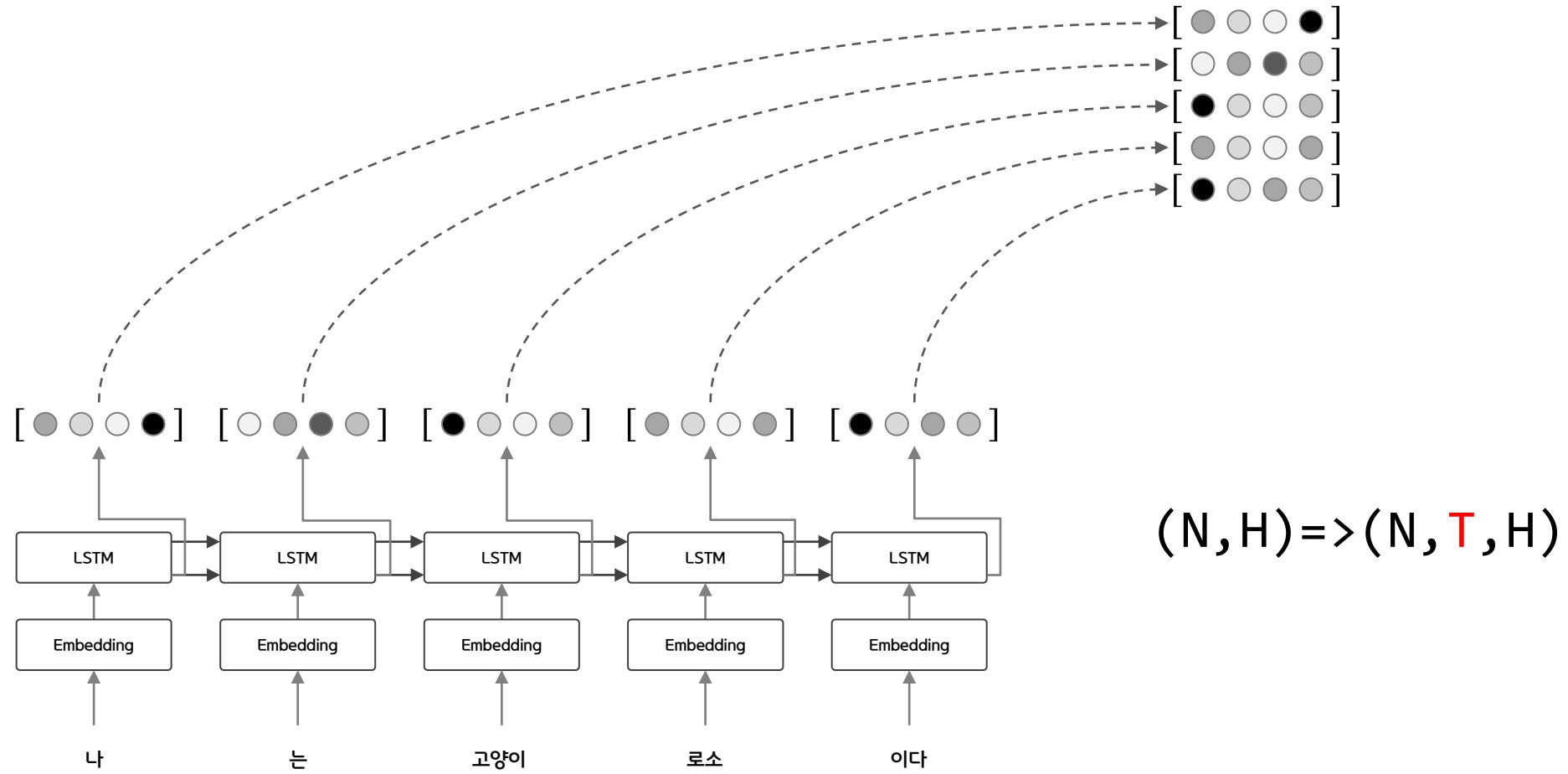
Encoder 개선

우선, encoder를 개선해보자.

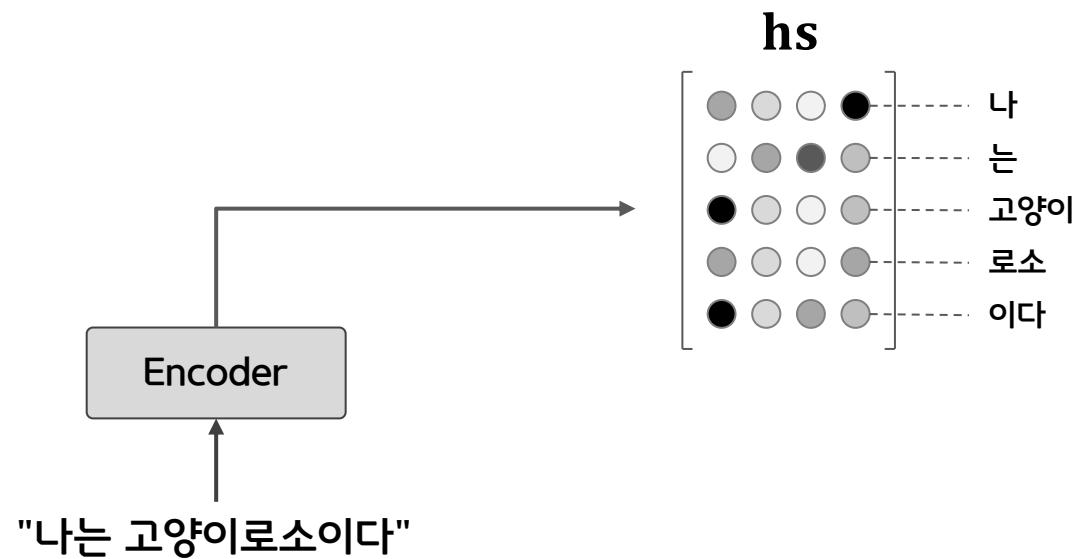
개선 포인트는, encoder 출력의 길이를 입력 문장의 길이에 따라 바꿔주는 것이다.
그러기 위해서, 시각 별(=단어 별) LSTM 계층의 은닉 상태 벡터를 모두 이용하는 것이다.
예를 들어, 5개 단어가 입력되었으면, encoder는 5개의 벡터를 출력한다.

그러면 각 시각별 LSTM 계층의 은닉 상태에는 어떤 정보가 담겨있을까?
직전에 입력된 단어에 대한 정보가 많이 포함되어 있을 것이다.
따라서, encoder가 출력하는 hs 행렬은 각 단어에 해당하는 벡터들의 집합일 것이다.

Encoder의 시각별(단어별) LSTM 계층의 은닉 상태를 모두 이용(hs로 표기)



Encoder의 출력 hs 는 단어 수만큼의 벡터를 포함하며, 각각의 벡터는 해당 단어에 대한 정보를 많이 포함한다.



이어서 decoder를 개선해보자.

encoder 가 각 단어에 대응하는 LSTM 계층의 은닉 상태 벡터를 hs 로 모아 출력하면,
이 hs 는 decoder에 전달되어 시계열 변환이 이루어진다.

개선포인트는, decoder가 encoder 의 LSTM 계층의 마지막 은닉상태만을
이용하는 것이 아닌, hs 를 전부 활용할 수 있도록 만드는 것이다.

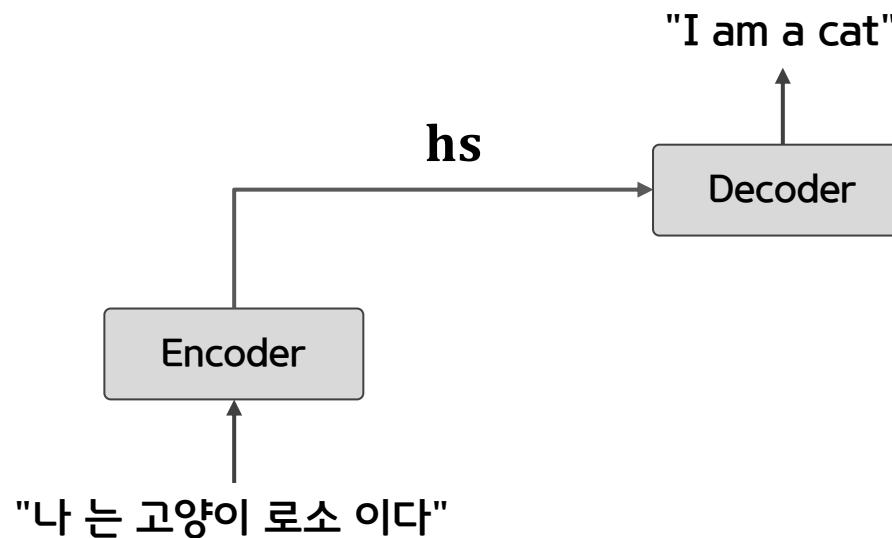
그 전에 인간이 문장을 번역할 때 머릿속에서 어떤 일이 일어날까 생각해보자.

우리는 '어떤 단어'에 주목하여 그 단어의 변환을 수시로 하게 된다.
이를 재현해야 한다.

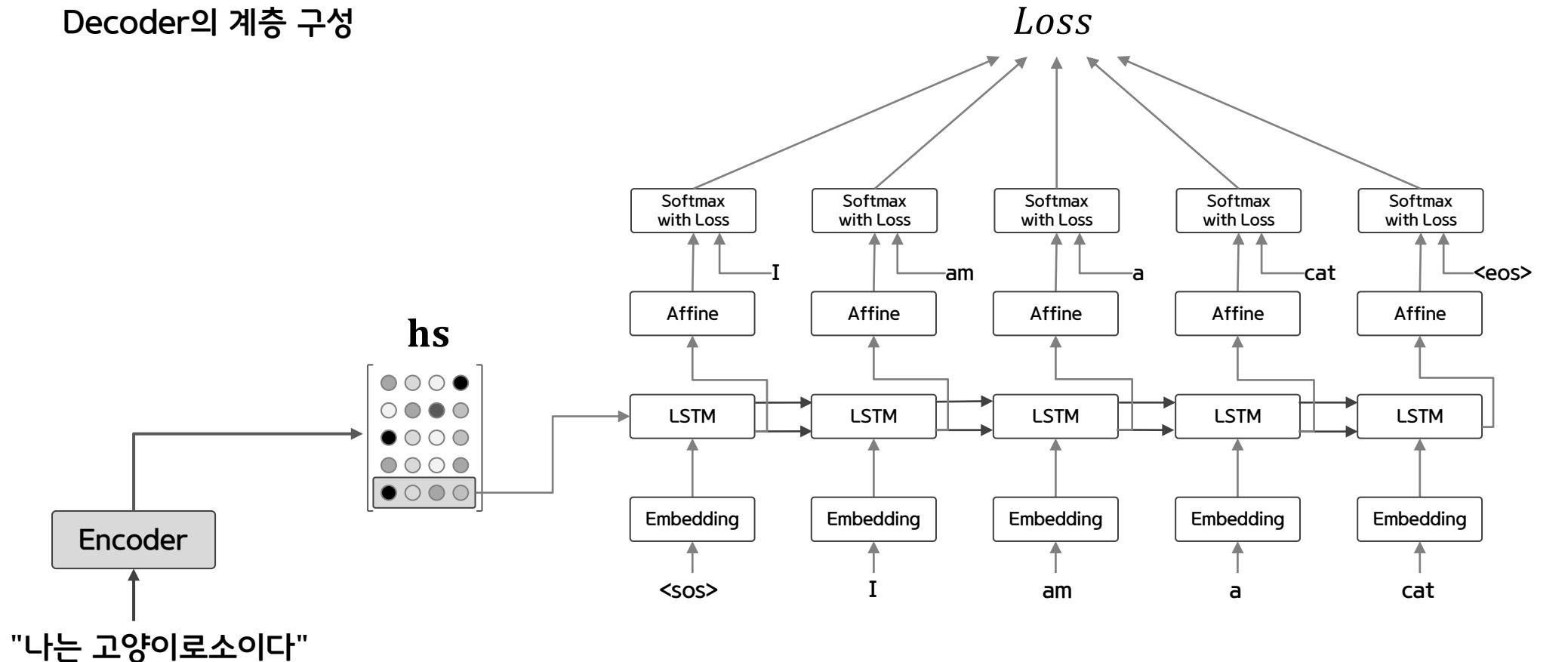
다시 말하면, 입력과 출력의 여러 단어 중 어떤 단어끼리 서로 관련되어 있는가를 학습시켜야 한다.

즉, 필요한 정보에만 주목하여 그 정보로부터 시계열 변환을 수행해야 한다. 이 구조를 어텐션 이라고 부른다.

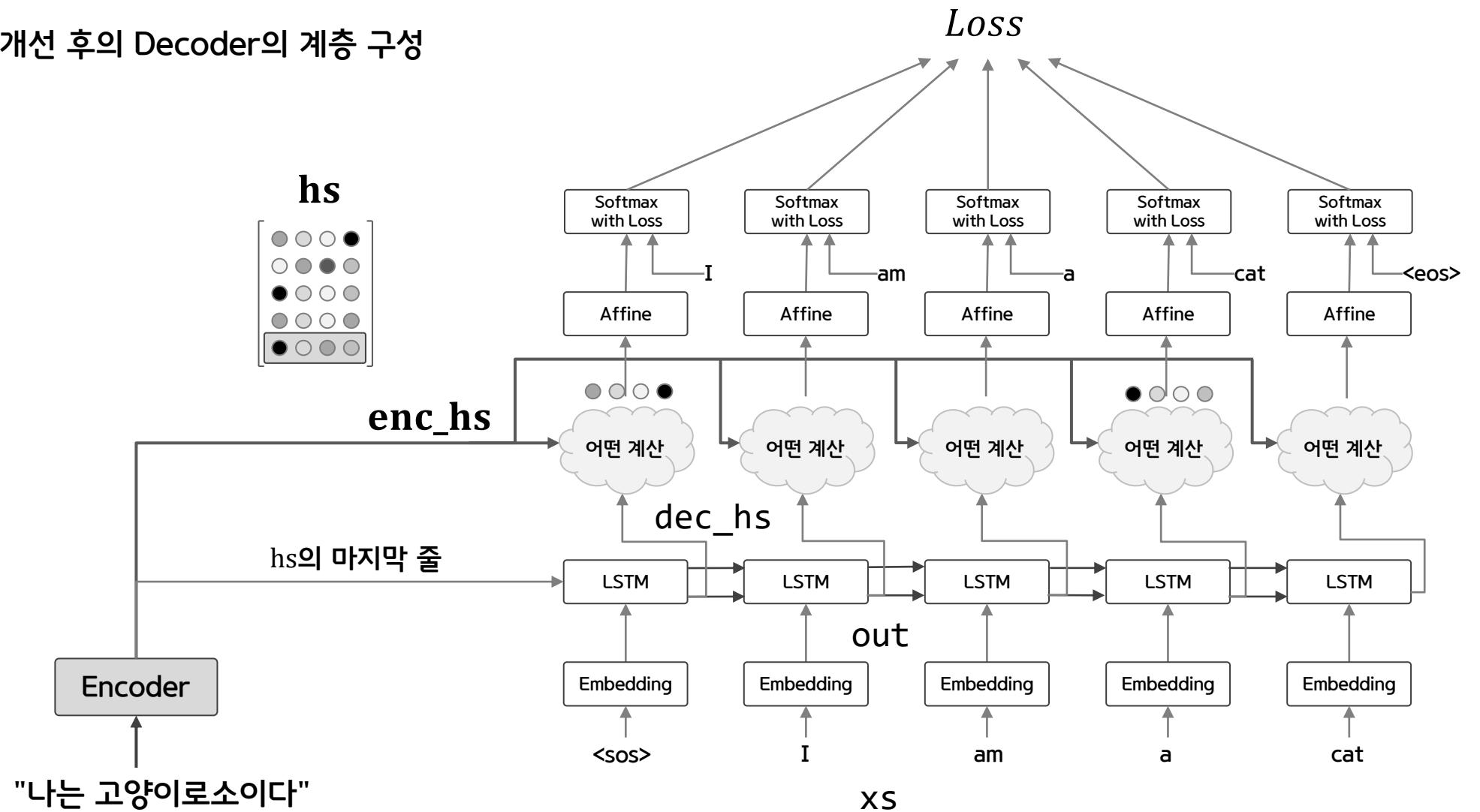
Encoder와 Decoder의 관계



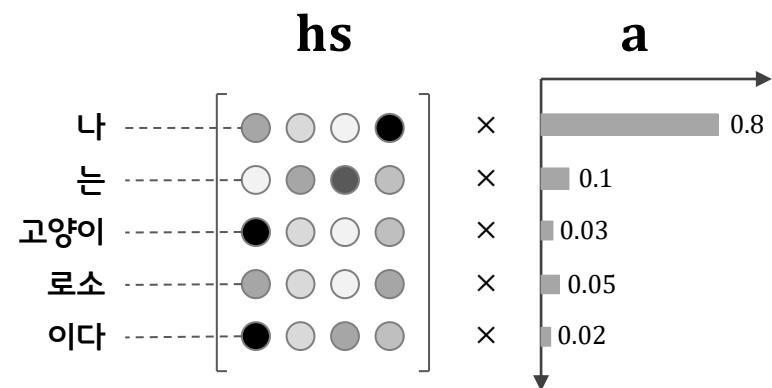
Decoder의 계층 구성



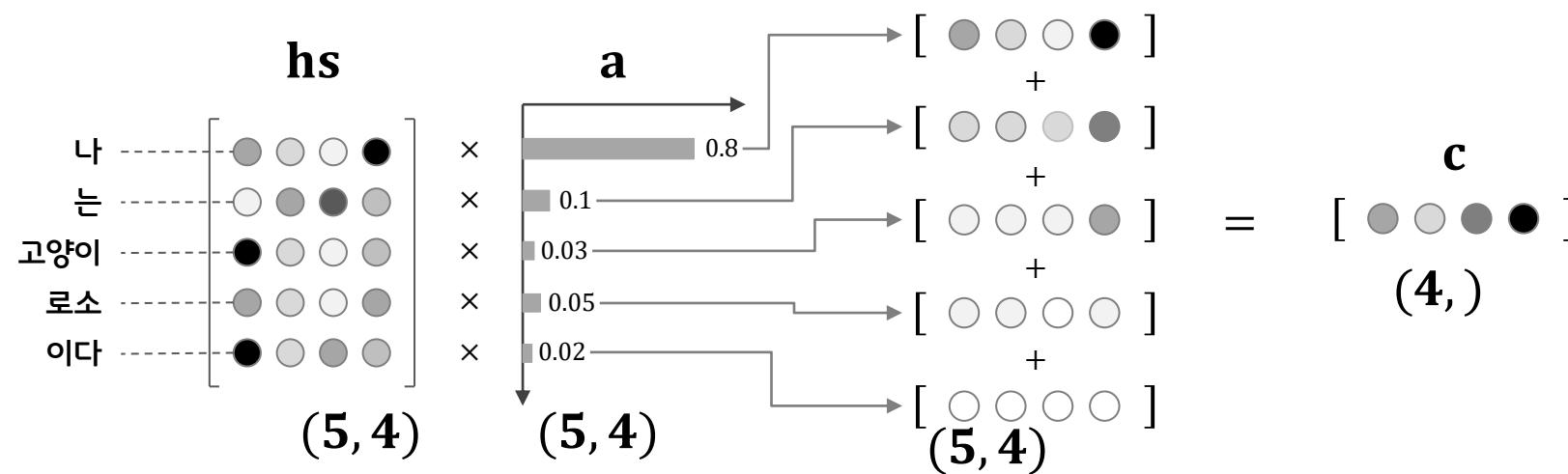
개선 후의 Decoder의 계층 구성



각 단어에 대해서 그것이 얼마나 중요한지를 나타내는 '가중치'를 구한다.

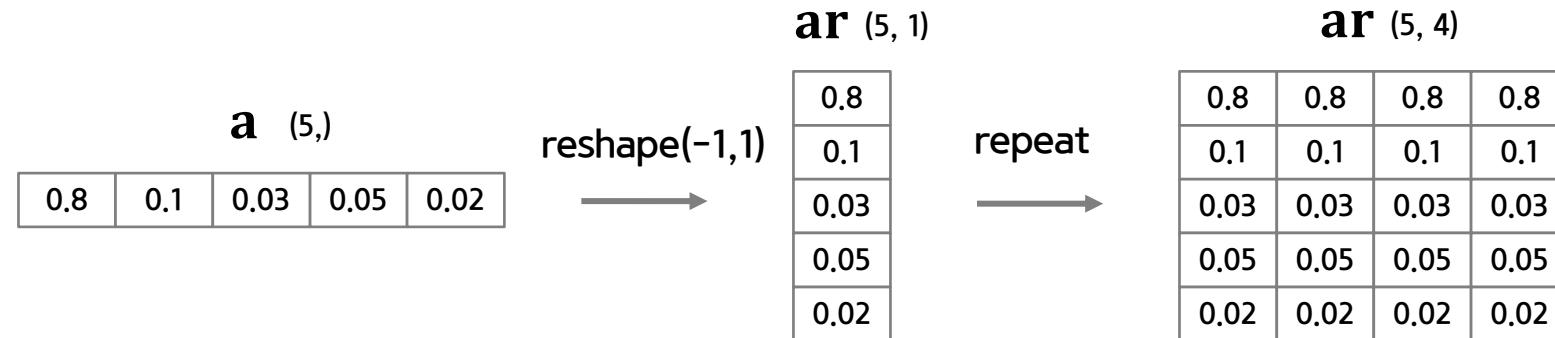


가중합을 계산하여 '맥락 벡터'를 구한다.



※ 코드 참조

`reshape()`와 `repeat()` 메서드를 거쳐 `a`로부터 `ar`을 생성(변수명 오른쪽에 형상을 표기)



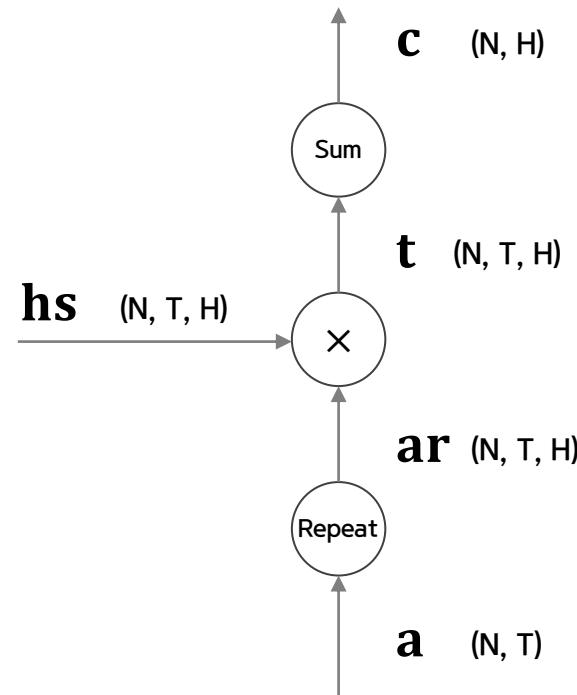
넘파이의 브로드캐스트

$$\begin{array}{c}
 \text{hs} \\
 \begin{array}{|c|c|c|c|} \hline
 0.1 & 1.3 & -0.4 & 1.2 \\ \hline
 -0.3 & -0.4 & -0.3 & -0.4 \\ \hline
 -1.2 & 0.9 & -1.7 & 0.2 \\ \hline
 -0.7 & -0.8 & 0.3 & -0.2 \\ \hline
 0.6 & 2.1 & 1.0 & -0.4 \\ \hline
 \end{array}
 \end{array}
 \quad *
 \quad
 \begin{array}{c}
 \text{ar} \\
 \begin{array}{|c|} \hline
 0.8 \\ \hline
 0.1 \\ \hline
 0.03 \\ \hline
 0.05 \\ \hline
 0.02 \\ \hline
 \end{array}
 \end{array}
 \quad =
 \quad
 \begin{array}{c}
 \text{hs} \\
 \begin{array}{|c|c|c|c|} \hline
 0.1 & 1.3 & -0.4 & 1.2 \\ \hline
 -0.3 & -0.4 & -0.3 & -0.4 \\ \hline
 -1.2 & 0.9 & -1.7 & 0.2 \\ \hline
 -0.7 & -0.8 & 0.3 & -0.2 \\ \hline
 0.6 & 2.1 & 1.0 & -0.4 \\ \hline
 \end{array}
 \end{array}
 \quad *
 \quad
 \begin{array}{c}
 \text{ar} \\
 \begin{array}{|c|c|c|c|} \hline
 0.8 & 0.8 & 0.8 & 0.8 \\ \hline
 0.1 & 0.1 & 0.1 & 0.1 \\ \hline
 0.03 & 0.03 & 0.03 & 0.03 \\ \hline
 0.05 & 0.05 & 0.05 & 0.05 \\ \hline
 0.02 & 0.02 & 0.02 & 0.02 \\ \hline
 \end{array}
 \end{array}$$

$(1, 5, 4)$ $(1, 5, 1)$

※ 코드 참조

가중합의 계산 그래프



※ 코드 참조

reshape()와 repeat() 메서드를 거쳐 a로부터 ar을 생성(변수명 오른쪽에 형상을 표기)

```
import numpy as np

N, T, H = 2, 5, 4
hs = np.arange(1,N*T*H+1).reshape(N, T, H)
print(hs)
a = np.array([[0.8, 0.1, 0.03, 0.05, 0.02],
              [0.01, 0.02, 0.9, 0.05, 0.02]])

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

t = hs * ar
print(t.shape)

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

(2,5,4)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32
33	34	35	36
37	38	39	20

(2,5,4)

0.8	0.8	0.8	0.8
0.1	0.1	0.1	0.1
0.03	0.03	0.03	0.03
0.05	0.05	0.05	0.05
0.02	0.02	0.02	0.02
0.01	0.01	0.01	0.01
0.02	0.02	0.02	0.02
0.9	0.9	0.9	0.9
0.05	0.05	0.05	0.05
0.02	0.02	0.02	0.02

*

2.56	3.56	4.56	5.56
29.2	30.2	31.2	32.2

(2,4)

```

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

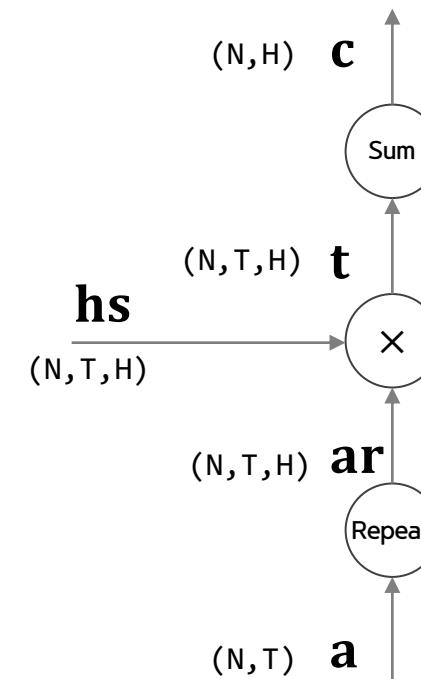
    ar = a.reshape(N, T, 1).repeat(H, axis=2)
    t = hs * ar
    c = np.sum(t, axis=1)

    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

```



```

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

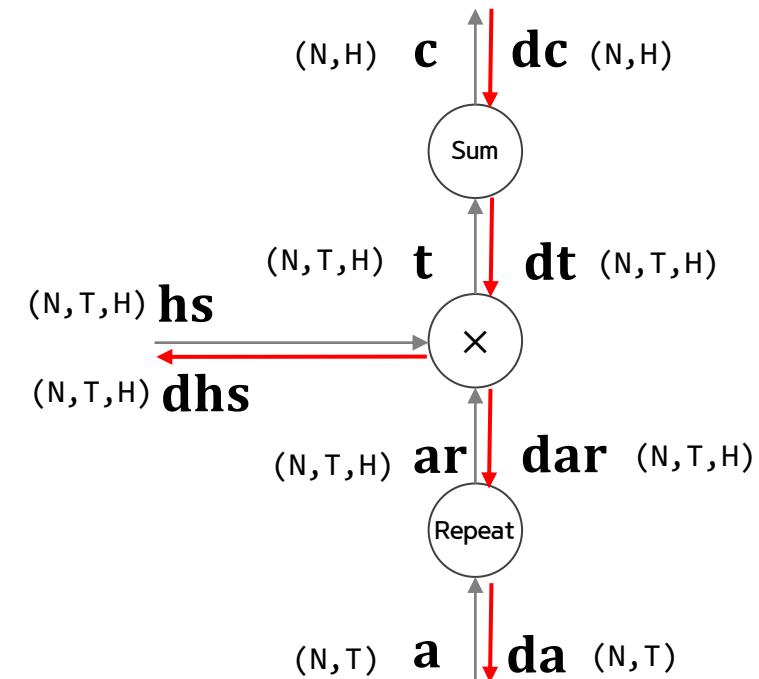
    ar = a.reshape(N, T, 1).repeat(H, axis=2)
    t = hs * ar
    c = np.sum(t, axis=1)

    self.cache = (hs, ar)
    return c

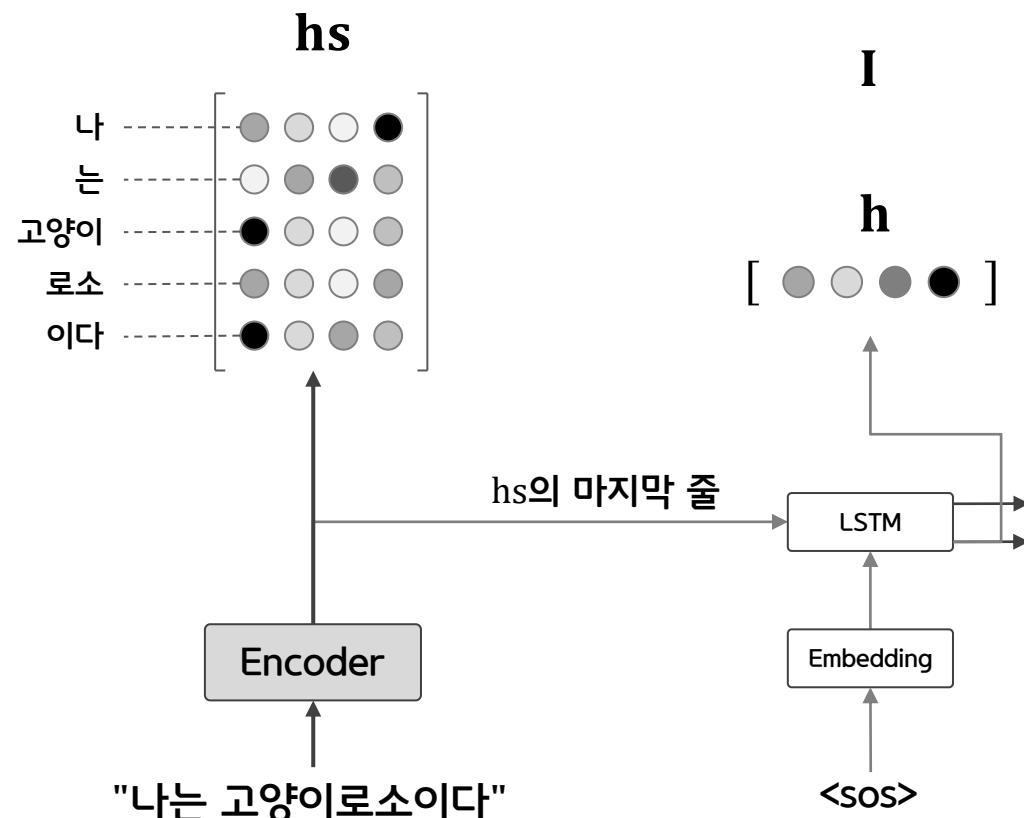
def backward(self, dc):
    hs, ar = self.cache
    N, T, H = hs.shape
    dt = dc.reshape(N, 1, H).repeat(T, axis=1)
    dar = dt * hs
    dhs = dt * ar
    da = np.sum(dar, axis=2)

    return dhs, da

```

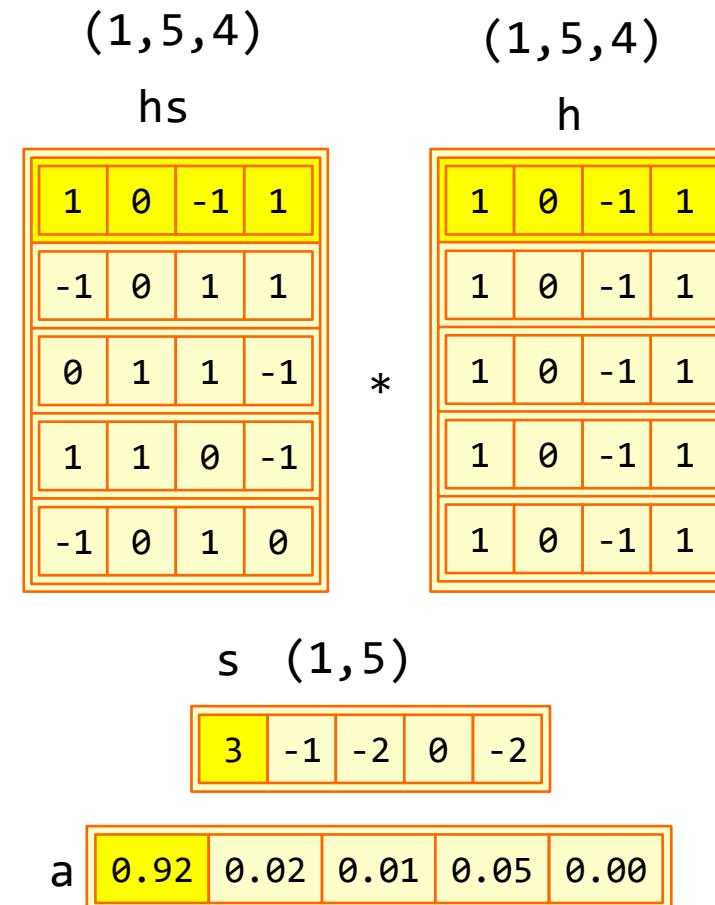


Decoder의 첫 번째 LSTM 계층의 은닉 상태 벡터

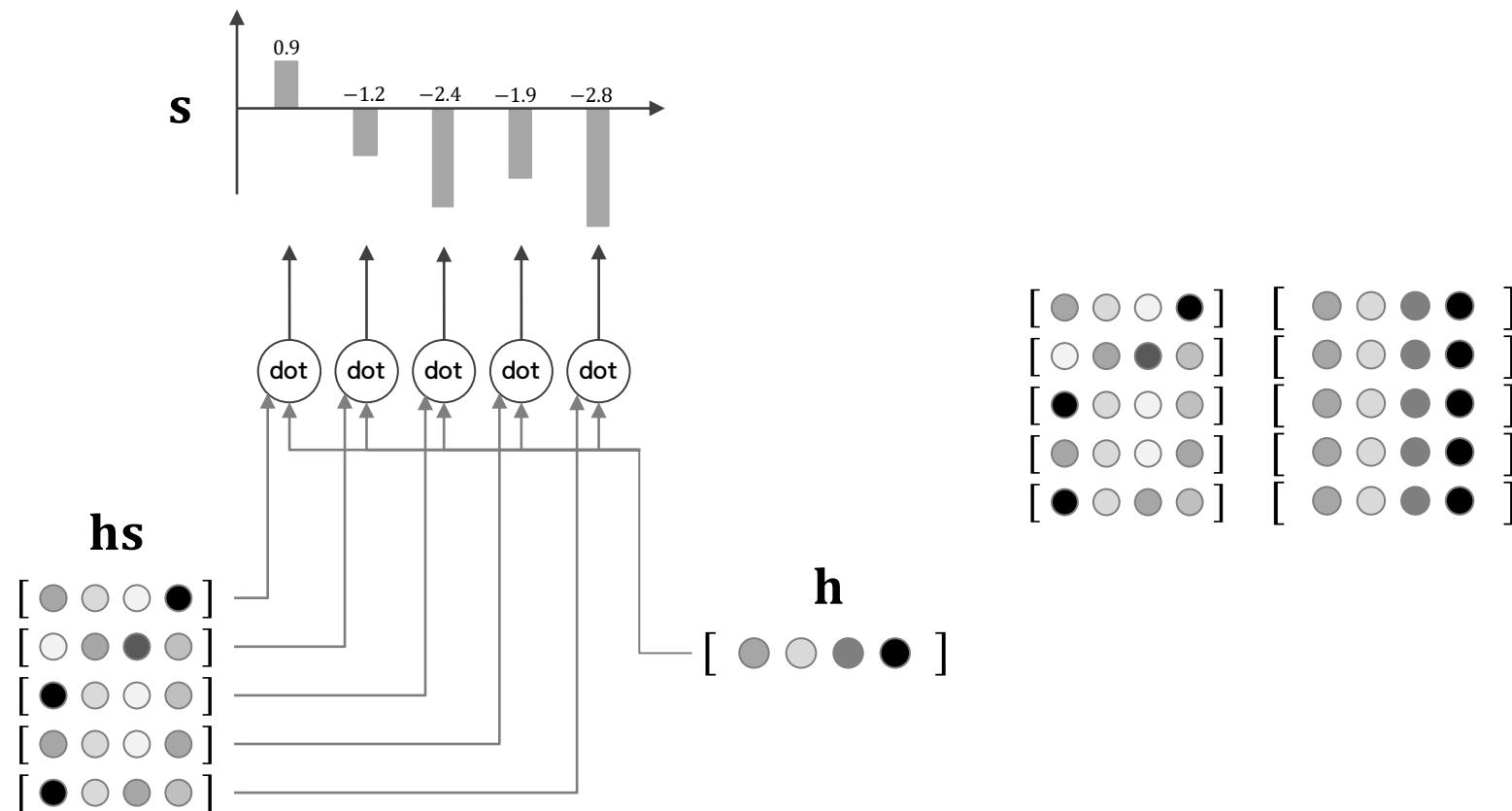


Decoder의 첫 번째 LSTM 계층의 은닉 상태 벡터

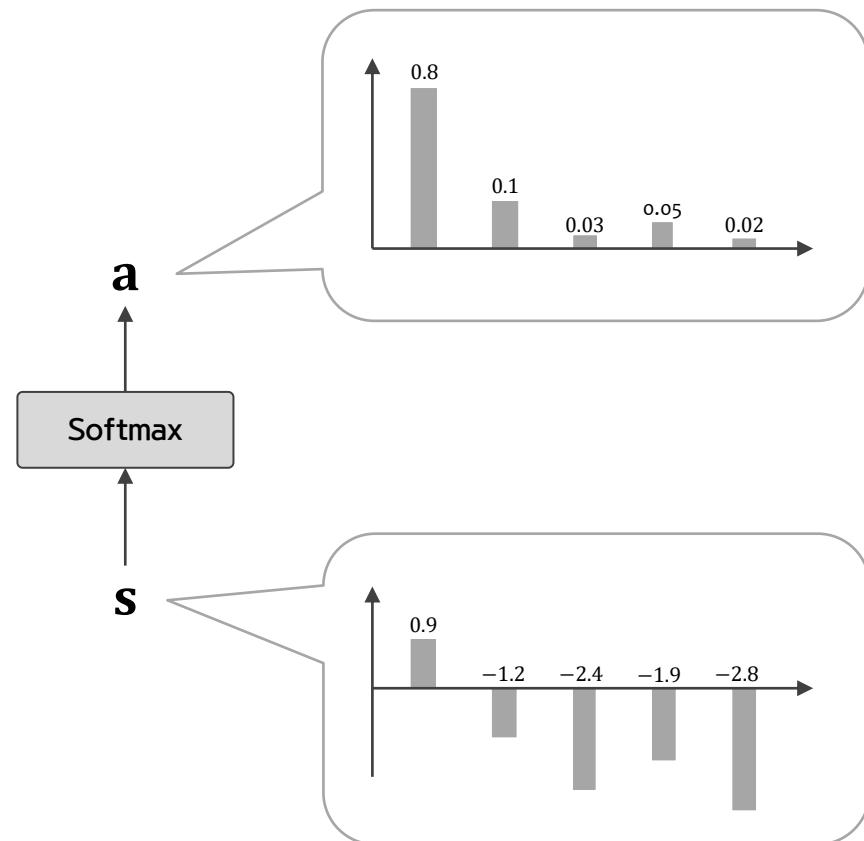
```
hs = np.array([
    [
        [1, 0, -1, 1],
        [-1, 0, 1, 1],
        [0, 1, 1, -1],
        [1, 1, 0, -1],
        [-1, 0, 1, 0]
    ]
])
h = np.array([[1, 0, -1, 1]])
N, T, H = hs.shape
hr = h.reshape(N, 1, H).repeat(T, axis=1)
t = hs * hr
s = np.sum(t, axis=2)
softmax = Softmax()
a = softmax.forward(s)
print(a)
```



내적을 통해 hs 의 각 행과 h 의 유사도를 산출(내적은 dot 노드로 그림)



Softmax를 통한 정규화



※ 코드 참조

```

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)

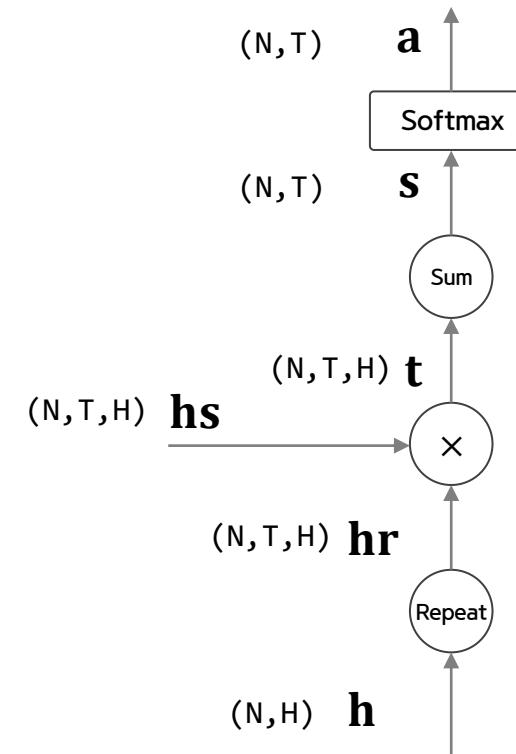
    self.cache = (hs, hr)
    return a

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

```



```

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

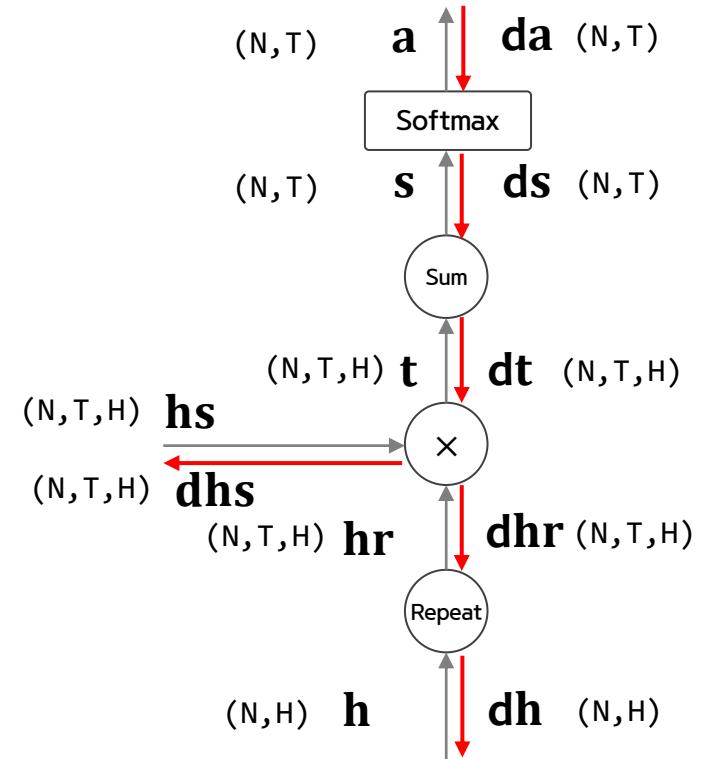
    hr = h.reshape(N, 1, H).repeat(T, axis=1)
    t = hs * hr
    s = np.sum(t, axis=2)
    a = self.softmax.forward(s)

    self.cache = (hs, hr)
    return a

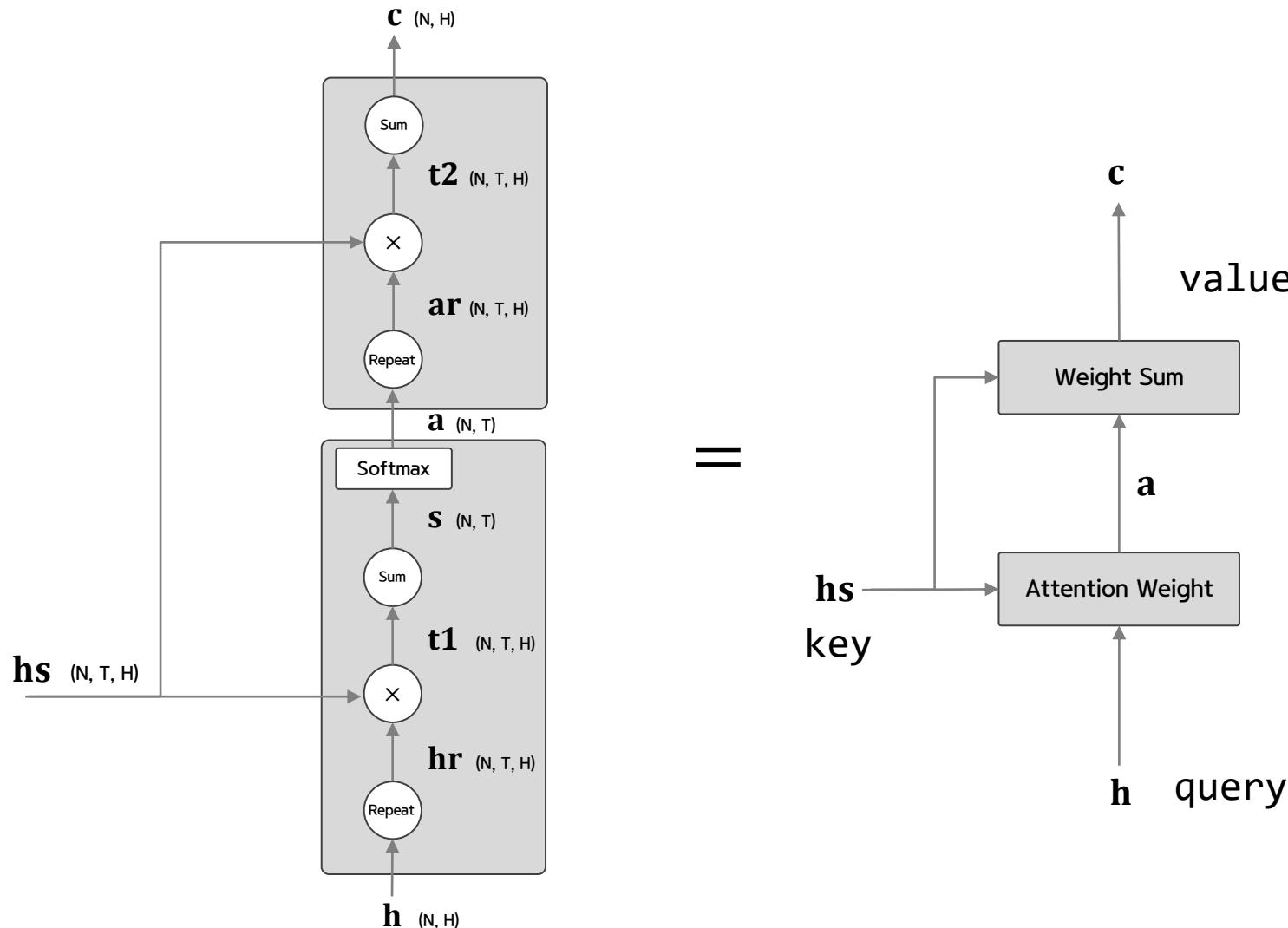
def backward(self, da):
    hs, hr = self.cache
    N, T, H = hs.shape

    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)

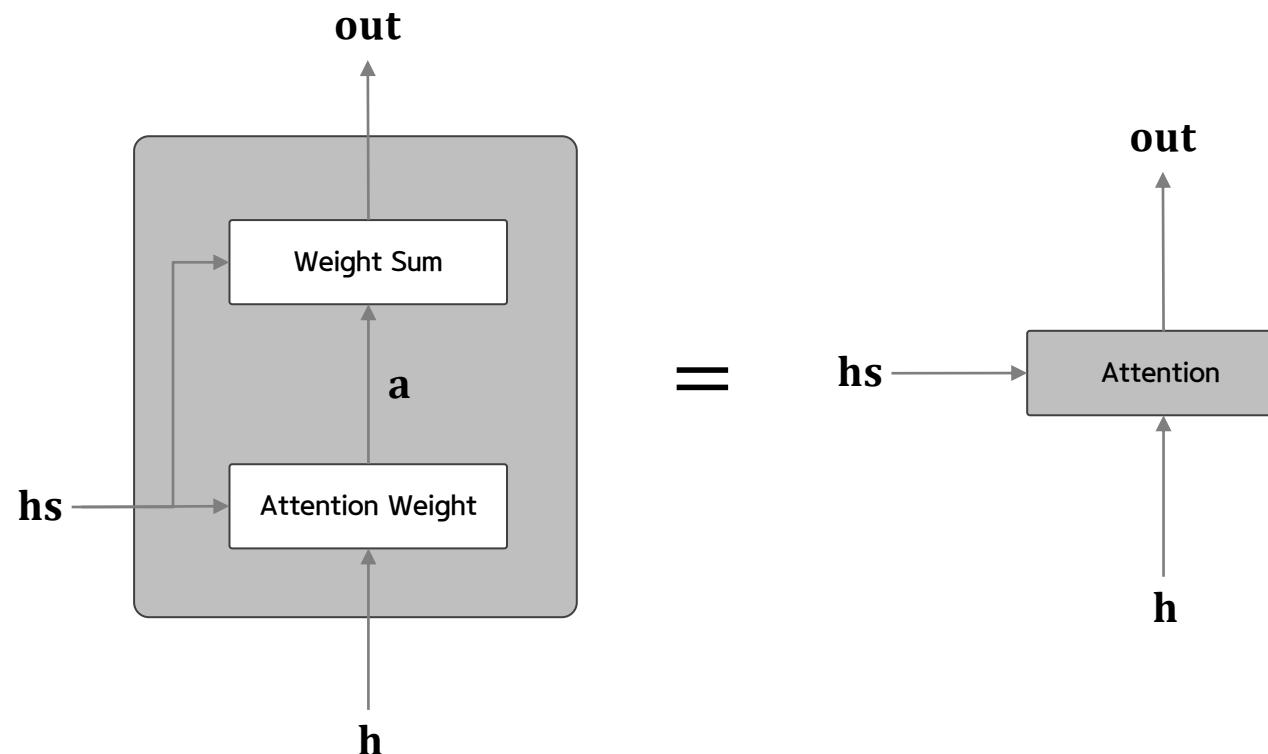
    return dhs, dh
  
```



맥락 벡터를 계산하는 계산 그래프

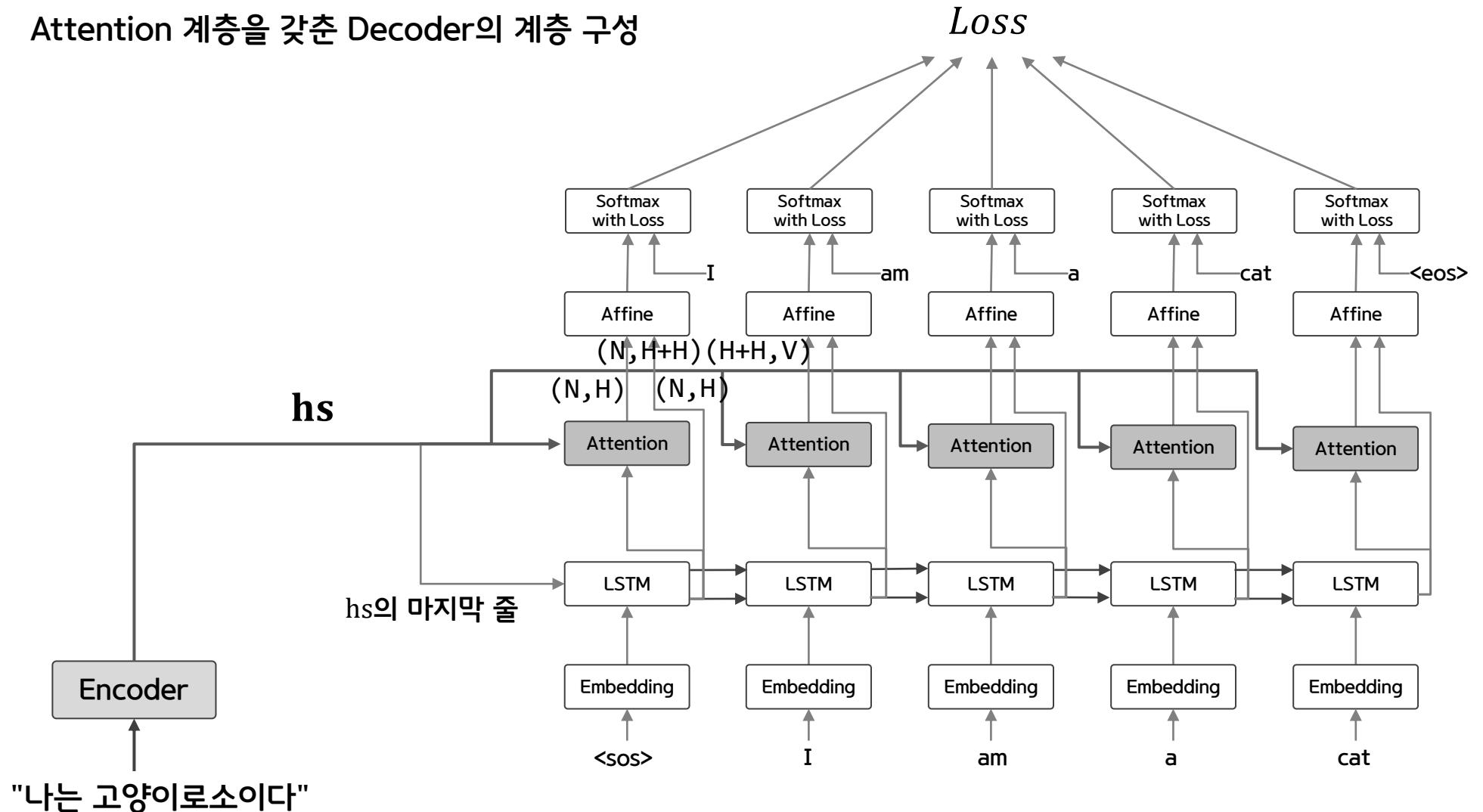


왼쪽 계산 그래프를 Attention 계층으로 정리

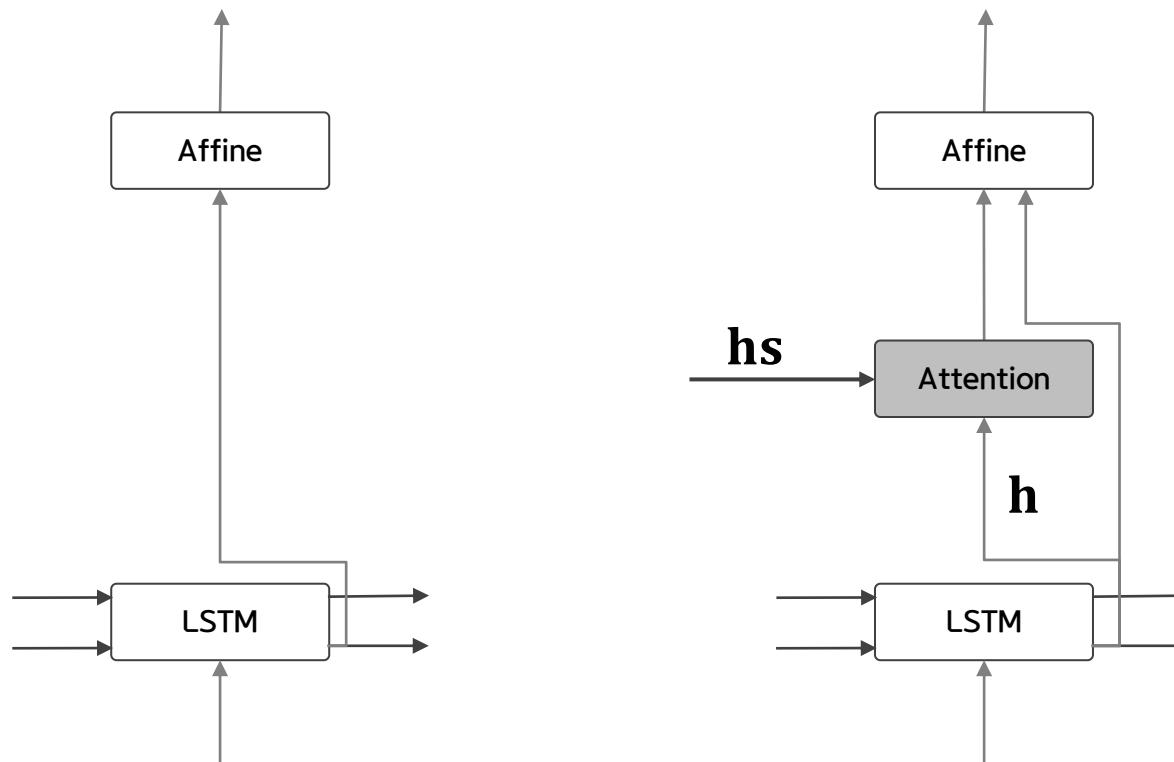


※ 코드 참조

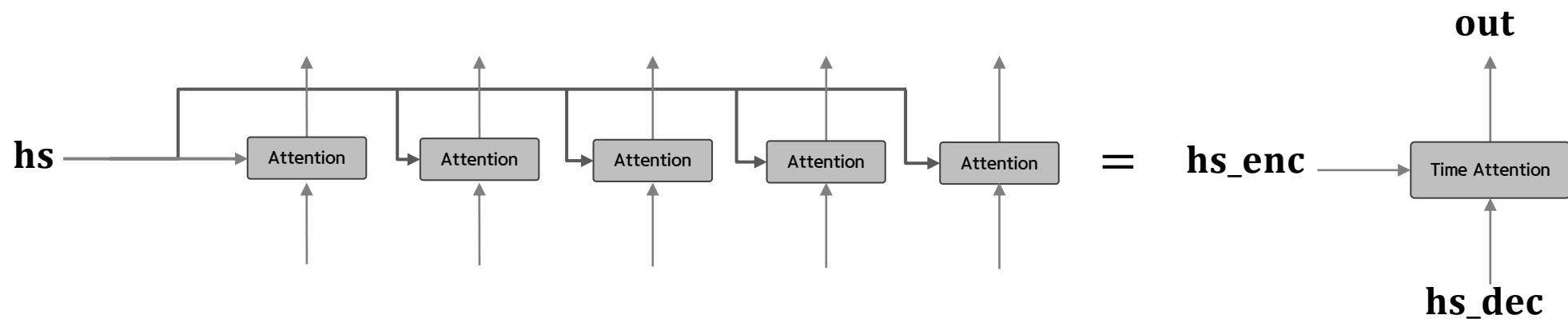
Attention 계층을 갖춘 Decoder의 계층 구조



앞 장의 Decoder(왼쪽)와 Attention이 추가된 Decoder(오른쪽) 비교: LSTM 계층에서 Affine 계층 까지만 그림



다수의 Attention 계층을 Time Attention 계층으로서 모아 구현



※ 코드 참조

8. 어텐션

8.1 어텐션의 구조

8.2 어텐션을 갖춘 seq2seq 구현

8.3 어텐션 평가

8.4 어텐션에 관한 남은 이야기

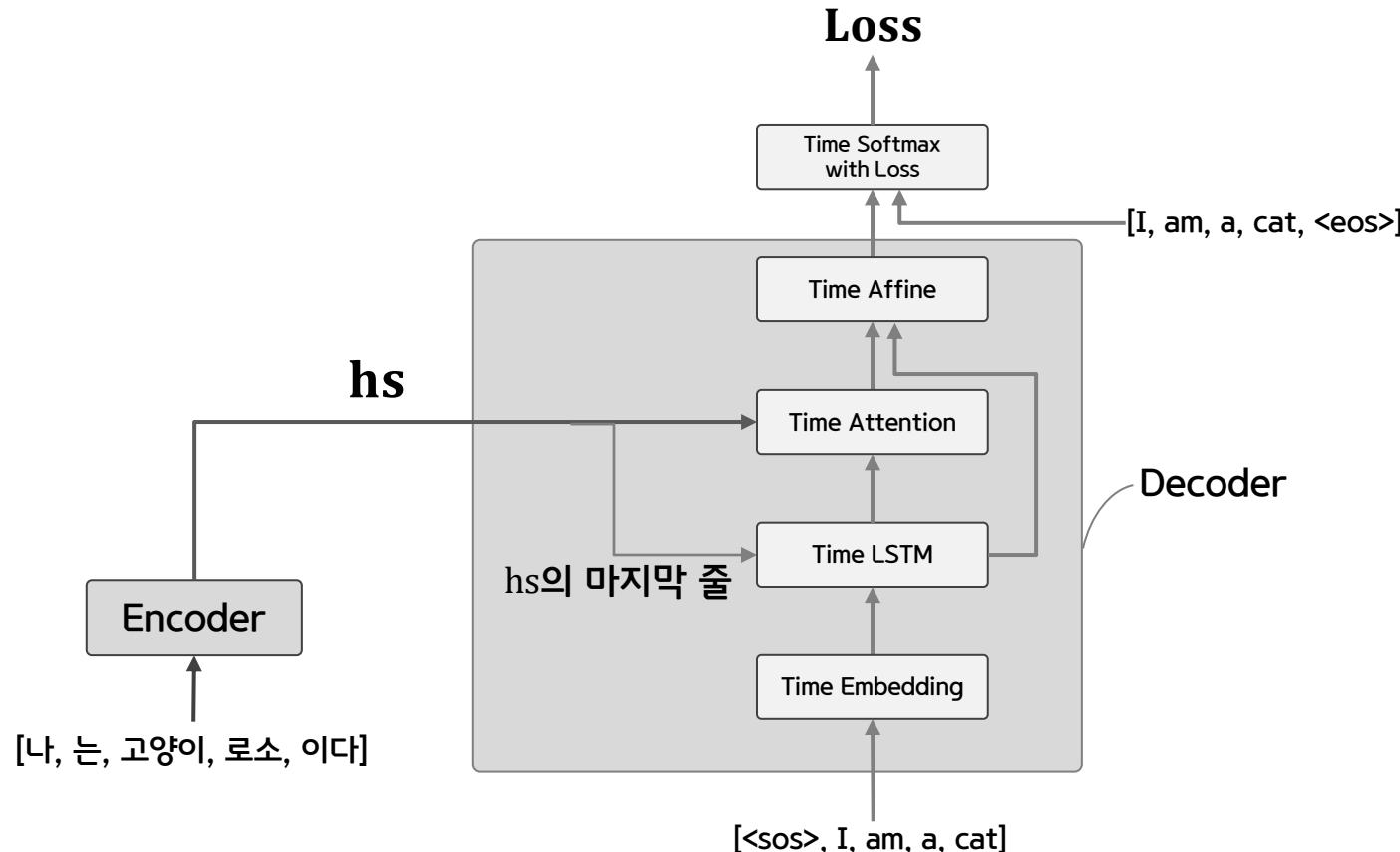
8.5 어텐션 응용

forward() 메소드는 LSTM 계층의 모든 은닉 상태 벡터를 반환한다.

인수로 받고있는 Encoder 객체는 앞 장에 LSTM계층의 마지막 은닉 상태 벡터를 갖는데,
여기서 이를 상속받는다.

※ 코드 참조

forward() 메서드에서 Time attention 계층의 출력과 LSTM 계층의 출력을 연결한다.
연결시에는 np.concatenate() 사용한다.



※ 코드 참조

8. 어텐션

8.1 어텐션의 구조

8.2 어텐션을 갖춘 seq2seq 구현

8.3 어텐션 평가

8.4 어텐션에 관한 남은 이야기

8.5 어텐션 응용

'날짜 형식'을 변경하는 문제로 어텐션을 갖춘 seq2seq의 효과를 확인해 보았다.
(데이터 크기가 작고, 어느 쪽인가를 맞추는 인위적인 문제)

번역용 데이터셋 중에서는 WMT 가 유명하여 seq2seq의 성능을 평가하는데
자주 이용되지만 크기가 크니(20GB) 부담된다.

자, 예를 들어 "september 27, 1994" 를 '1994-09-27' 같은 표준형식으로 변환해보자.

왜 하필 이문제냐?

이게 그렇게 보기보다 간단하지 않다.

변환 규칙이 나름 복잡하다.

그리고 질문과 답변 즉, 입력과 출력 사이에 알기 쉬운 대응관계가 있기 때문이다.

학습 데이터셋의 형식은 이러하다.

september 27, 1994	_1994-09-27
August 19, 2003	_2003-08-19
2/10/93	_1993-02-10
10/31/90	_1990-10-31
TUESDAY, SEPTEMBER 25, 1984	_1984-09-25
JUN 17, 2013	_2013-06-17
april 3, 1996	_1996-04-03
October 24, 1974	_1974-10-24
AUGUST 11, 1986	_1986-08-11
February 16, 2015	_2015-02-16

※ 코드 참조

이렇게 학습하면 1에폭부터 빠르게 정답률이 높아져 2에폭째에는 이미 거의 모든 문제를 풀어낸다.

이전 장들에서 다뤘던 단순한 seq2seq,
엿보기를 적용한 seq2seq 모델과 비교했을때 학습 속도 측면에서는 가장 빠르고,

정확도는 엿보기를 적용한 seq2seq과 동등했지만 현실의 시계열 데이터는 복잡하고 길기 때문에
어텐션이 매우 유리하다.

어텐션이 시계열 데이터를 변환할 때 어떤 원소에 주의를 기울이는지
보기 위해 어텐션을 시각화해보자.

학습이 끝난 AttentionSeq2seq로 날짜 변환을 수행할 때의 어텐션 가중치를 시각화 한다.

가로축은 입력문장, 세로축은 출력문장.

맵의 각 원소는 밝을수록 값이 크다(1.0).

※ 코드 참조

AUGUST 일때 08에 대응하고 있는 것을 볼 수 있다. seq2seq가 August가 8월에 대응한다는 사실을 데이터만 가지고 학습해 낸 것이다.

1983, 26도 1983과 26에 대응하고 있는 것을 볼 수 있다.

어텐션 모델은 인간이 이해할 수 있는 구조나 의미를 제공한다는 점에서 굉장한 의의가 있다.

어텐션을 사용해 단어와 단어의 관련성을 볼 수 있었으므로,

이를 보고 모델의 처리가 인간의 논리를 따르는지 판단이 가능하다.

8. 어텐션

8.1 어텐션의 구조

8.2 어텐션을 갖춘 seq2seq 구현

8.3 어텐션 평가

8.4 어텐션에 관한 남은 이야기

8.5 어텐션 응용

앞에서 단방향 RNN을 보았을 때, LSTM의 각 시각의 은닉 상태 벡터는 h_s 로 모아진다.

그리고 Encoder가 출력하는 h_s 의 각 행에는 그 행에 대응하는 단어의 성분이 많이 포함 되어있다.

글을 왼쪽에서 오른쪽으로 읽기때문에, '나는 고양이로소이다'라는 문장에서
'고양이'에 대응하는 벡터에는 '나', '는', '고양이'까지 총 세 단어의 정보가 인코딩 되어 들어간다.

하지만, 이렇게 하지 말고, '고양이'단어의 '주변'정보를 균형 있게 담고 싶을 때
LSTM을 양방향으로 처리하도록 한다. 이것이 양방향 LSTM이다.

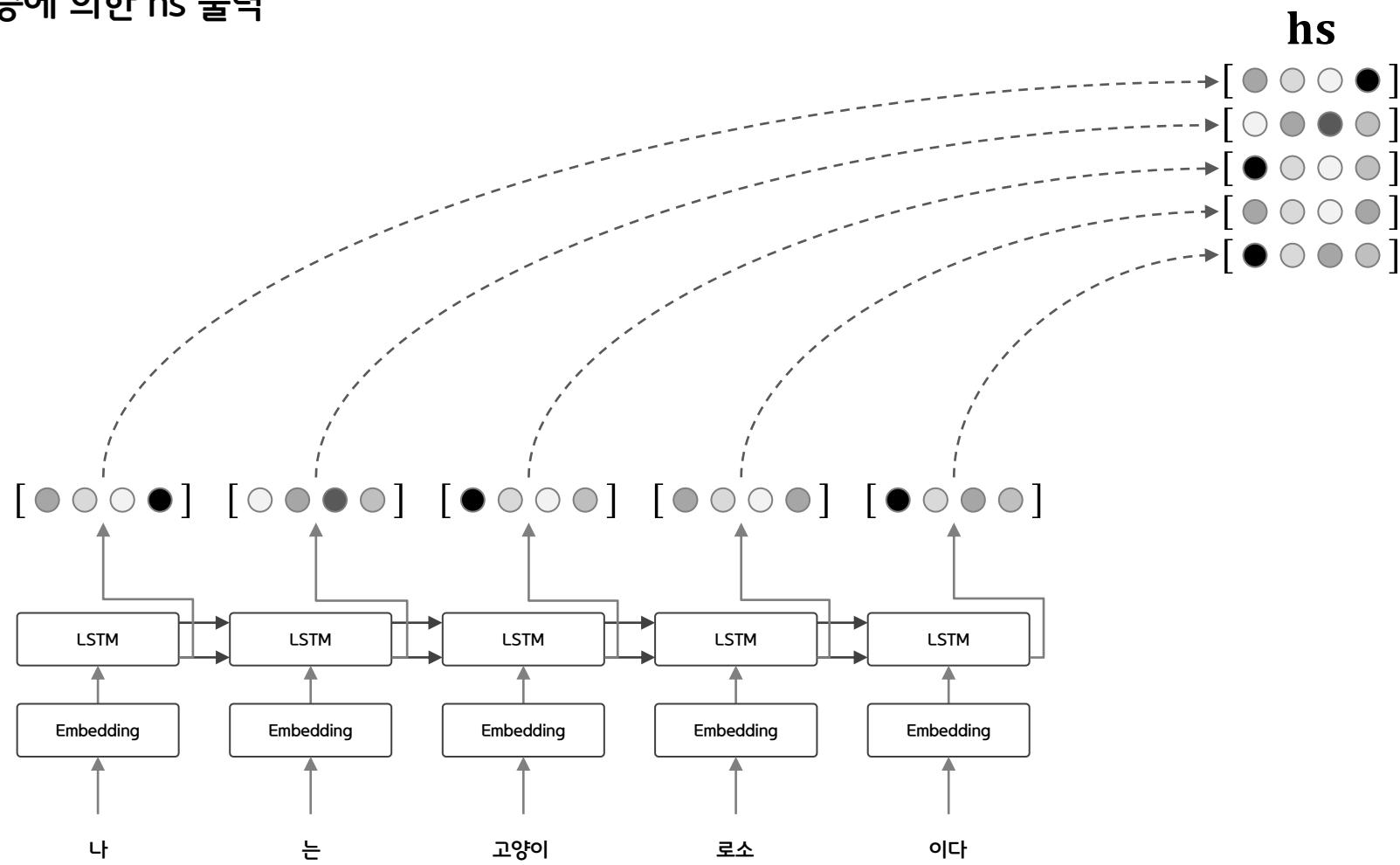
전에 했던 LSTM 계층에, 역방향으로 처리하는 LSTM 계층도 추가한다.

각 시각에서는 이 두 LSTM 계층의 은닉상태를 '연결'시킨 벡터를 최종 은닉 상태로 처리한다.
역방향으로 처리하는 LSTM 계층에 반대순서로 단어를 나열하여 넣는다.

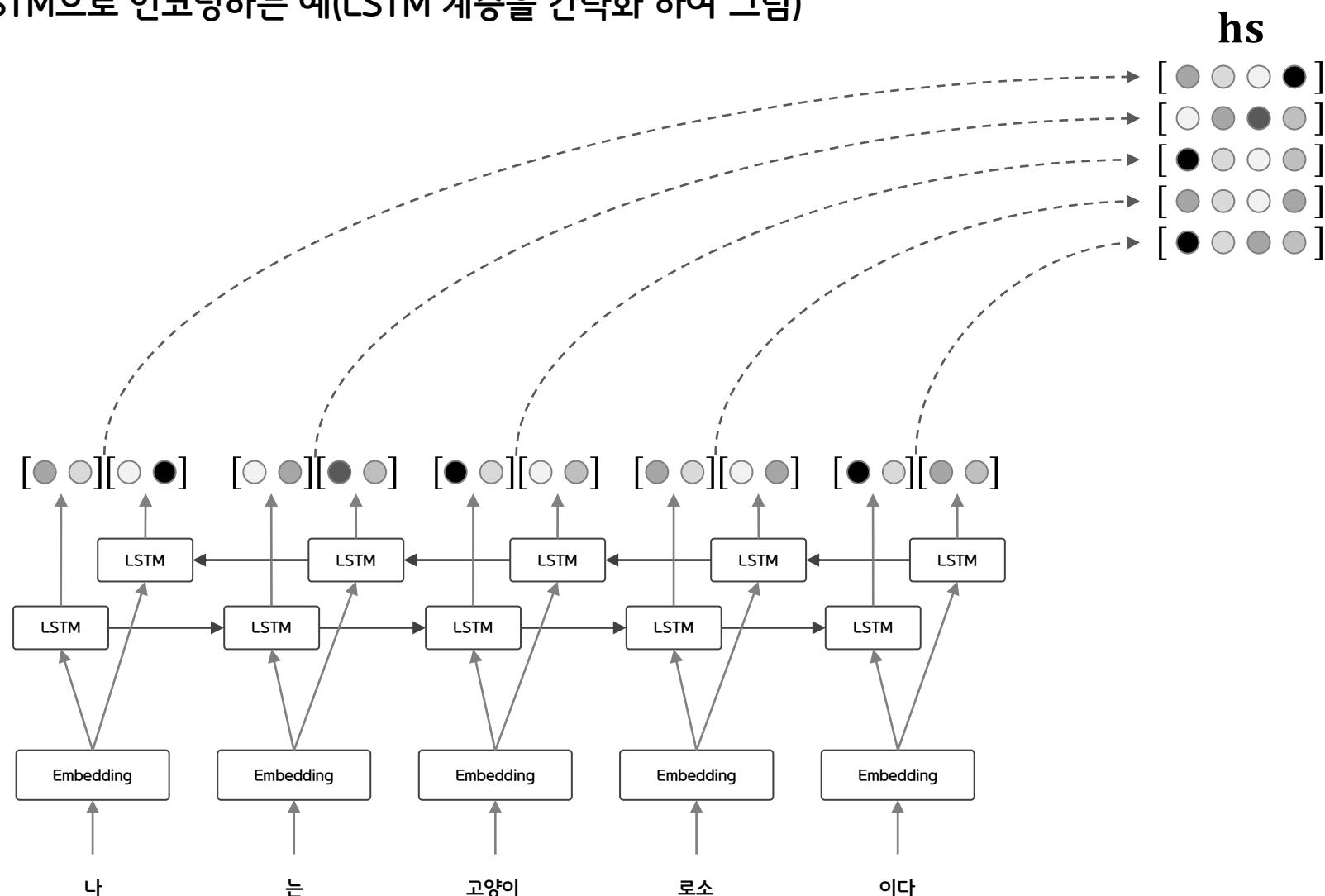
즉, 오른쪽에서 왼쪽으로 처리한다.

이 두 계층을 연결하기만 하면 양방향 LSTM 계층이 나온다.

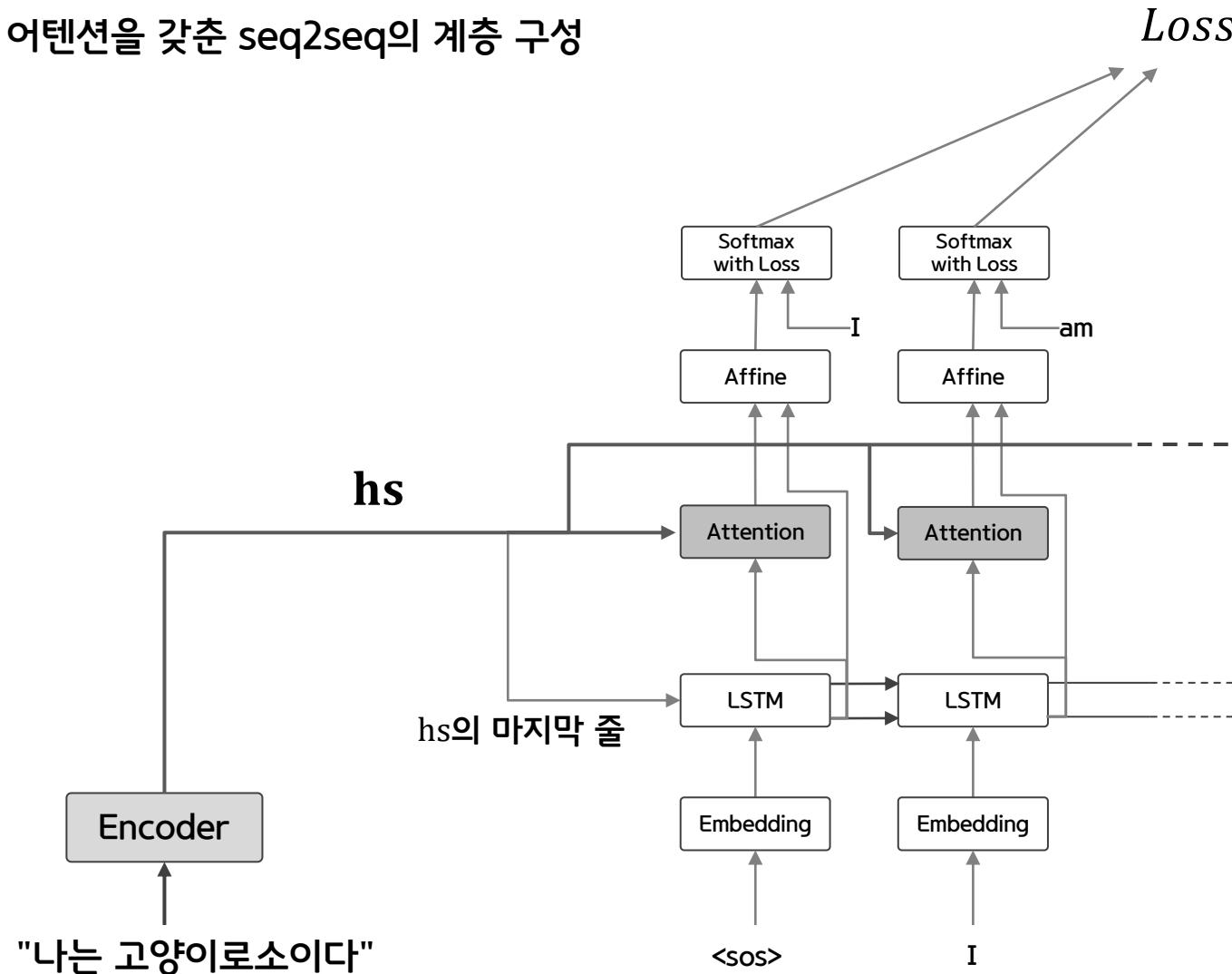
LSTM 계층에 의한 hs 출력



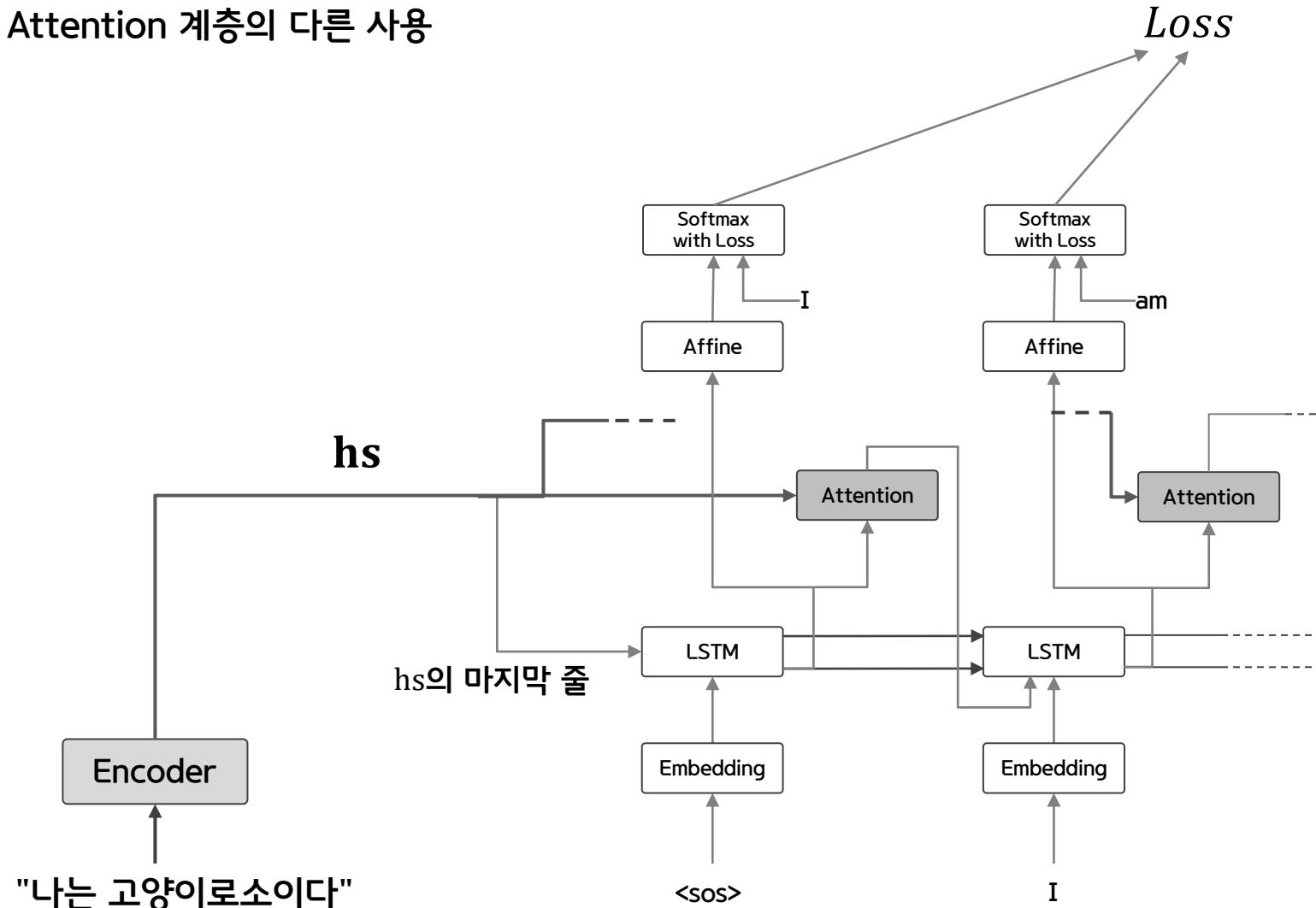
양방향 LSTM으로 인코딩하는 예(LSTM 계층을 간략화 하여 그림)



어텐션을 갖춘 seq2seq의 계층 구성



Attention 계층의 다른 사용



앞에선 Attention 계층을 LSTM 계층과 Affine 계층 사이에 삽입했지만,

Attention 계층을 이용하는 장소가 정해져 있지는 않다.

LSTM 계층 전에 넣어, LSTM 계층의 입력으로 사용할 수 있다.

뭐가 더 정확도가 높냐는, 실제 데이터에 따라 다르다.

직접 해보기전에는 알 수 없다.

구현 관점에서는 전자의 구성이 쉽다.

seq2seq을 심층화할 때 쉽게 떠올릴 수 있는 건 RNN층을 깊게 쌓는 방법이다.
그러면 표현력이 높은 모델을 만들 수 있다.

보통은 Encoder와 Decoder에서는 같은 층수의 LSTM 계층을 이용하는 것이 일반적인데,
여러가지 변형을 하면서, Decoder의 LSTM 계층의 은닉 상태를 Attention 계층에 입력하고,
Attention 계층의 출력인 맥락벡터를 Decoder의 여러 계층(LSTM 계층과 Affine계층)으로 전파할 수 있다.

skip 연결은 층을 깊게할 때 사용하는 중요한 기법이다.

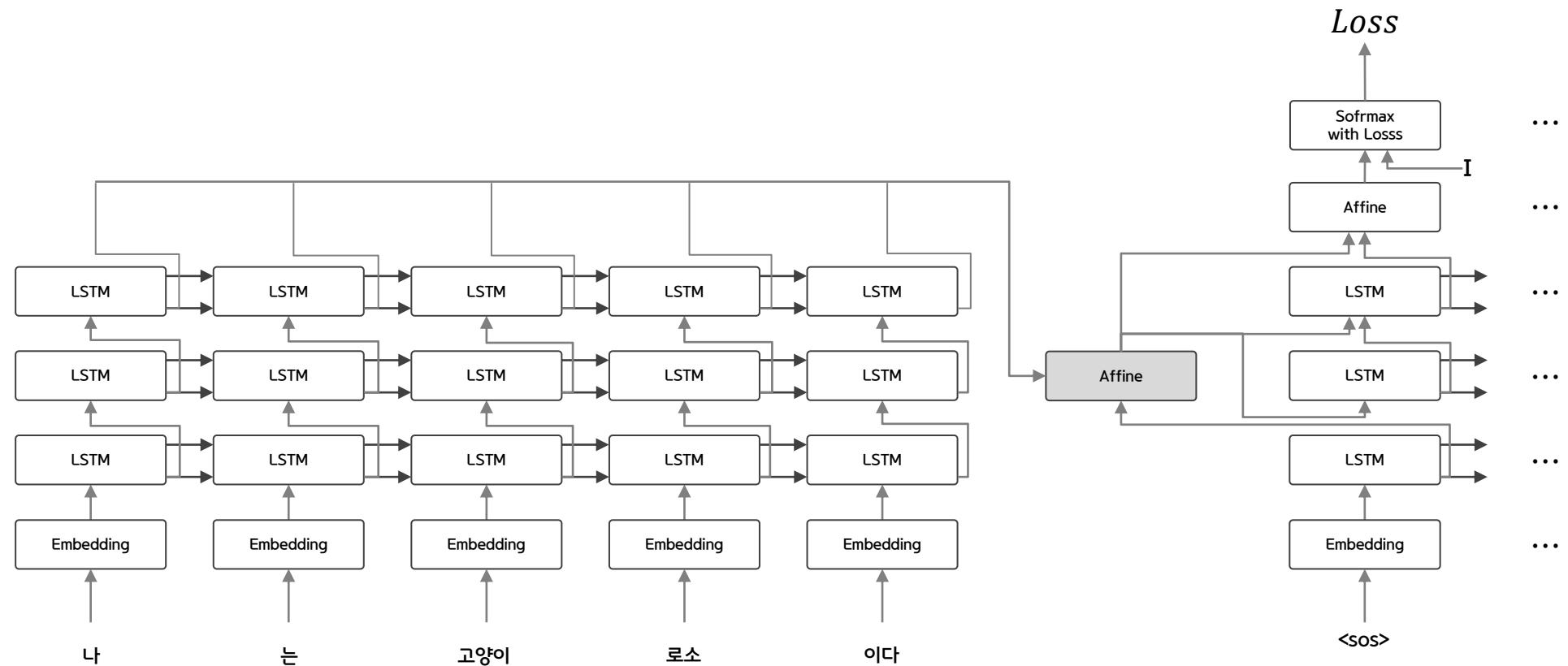
계층을 넘어(=계층을 건너 뛰어) '선을 연결'하는 단순한 기법이다.

이때 skip 연결의 접속부에서는 2개의 출력이 더해진다.

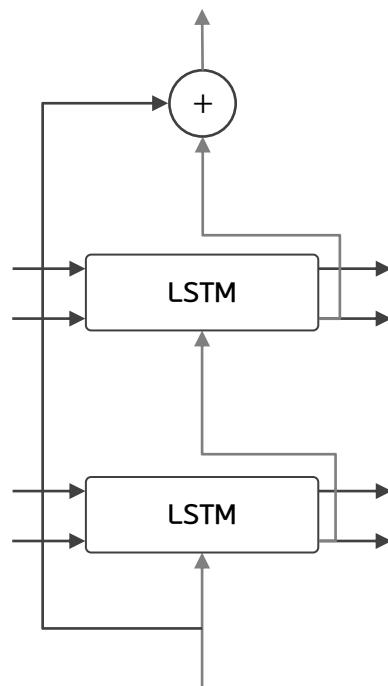
왜냐하면, 덧셈은 역 전파 시 기울기를 그대로 '흘려' 보내므로,
skip 연결의 기울기가 아무런 영향을 받지 않고 모든 계층으로 흐르기 때문이다.

따라서 층이 깊어져도 기울기가 소실되지 않고 전파되어 잘 학습된다.

3층 LSTM 계층을 사용한 어텐션을 갖춘 seq2seq



LSTM 계층의 skip 연결 예



8. 어텐션

8.1 어텐션의 구조

8.2 어텐션을 갖춘 seq2seq 구현

8.3 어텐션 평가

8.4 어텐션에 관한 남은 이야기

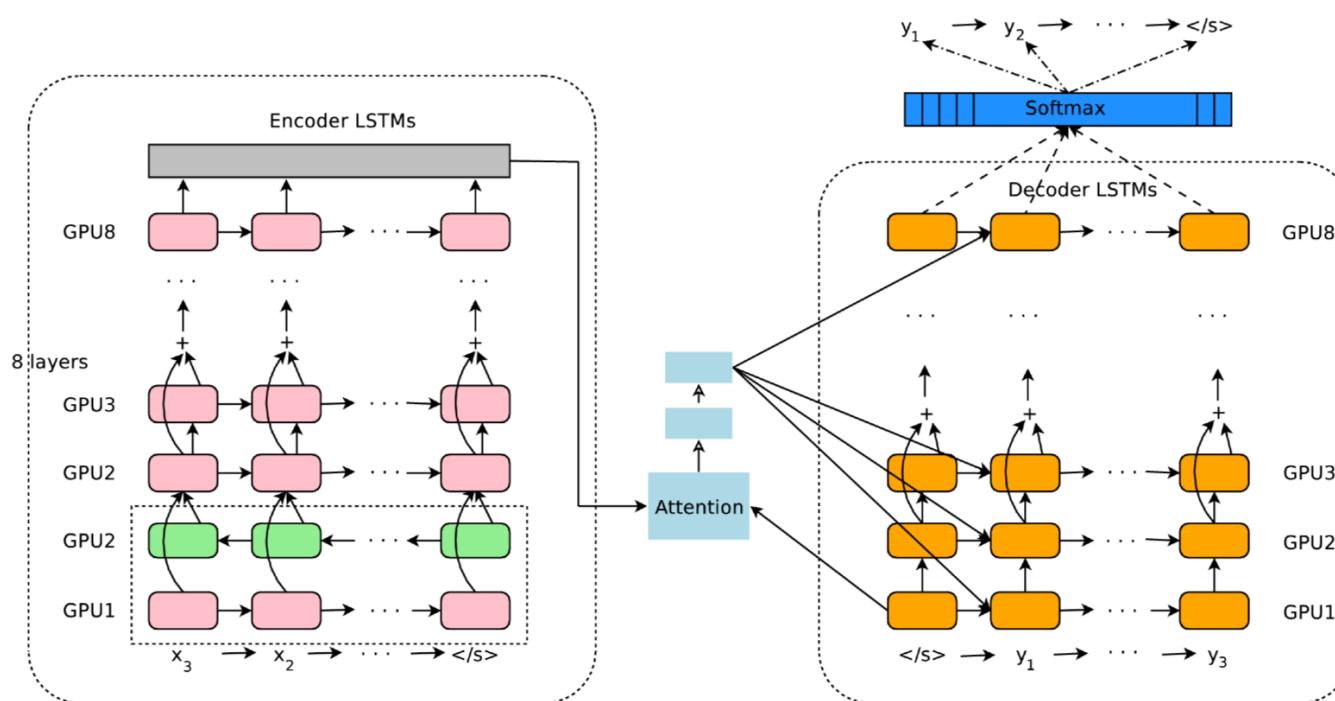
8.5 어텐션 응용

최근 딥러닝 연구에서 어텐션이 중요한 기술로 다양한 방면에서 등장한다.

최첨단 연구 3가지를 소개한다.

GNMT의 계층 구성

기계번역의 역사를 보면 다음과 같이 변화해왔다.
 규칙 기반 번역 -> 용례 기반 번역 -> 통계 기반 번역
 현재는 신경망 기계 번역(NMT)이 주목받고 있다.
 계층 구성은 다음과 같다.



우리가 앞서 배웠던 어텐션을 갖춘 seq2seq와 마찬가지로 Encoder, Decoder, Attention으로 구성되어 있다.

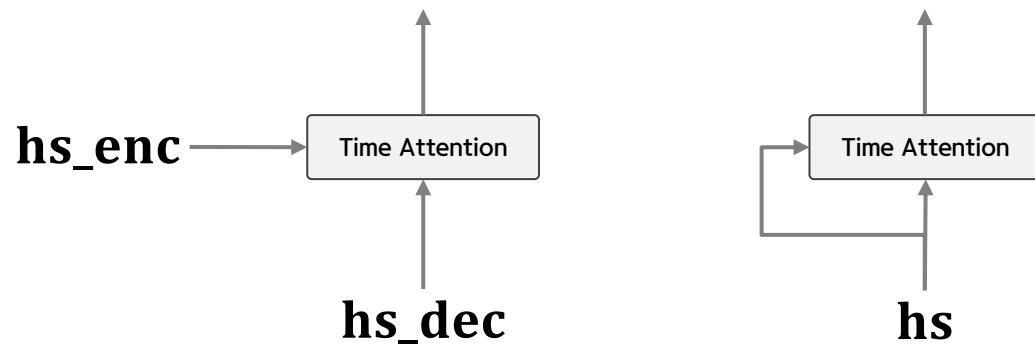
다만, 여기에 번역 정확도를 높이기 위해 LSTM 계층의 다층화, 양방향 LSTM, skip 연결 등을 추가했다.

그리고 학습 시간을 단축하기 위해 GPU로 분산학습을 수행하고 있다.

이외에도 낮은 빈도의 단어처리나 추론 고속화를 위한 양자화 등의 연구도 이루어지고 있다.

이로써 점점 사람의 정확도에 가까워지고 있다.

왼쪽이 일반적인 어텐션, 오른쪽이 셀프어텐션

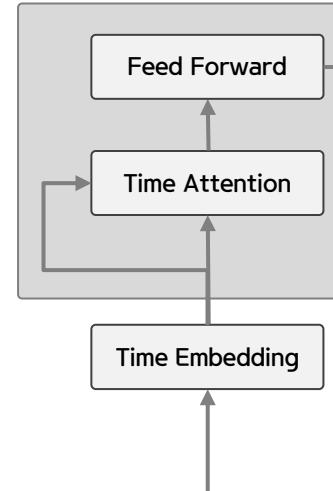


"Attention is all you need"

트랜스포머의 계층 구성

BERT

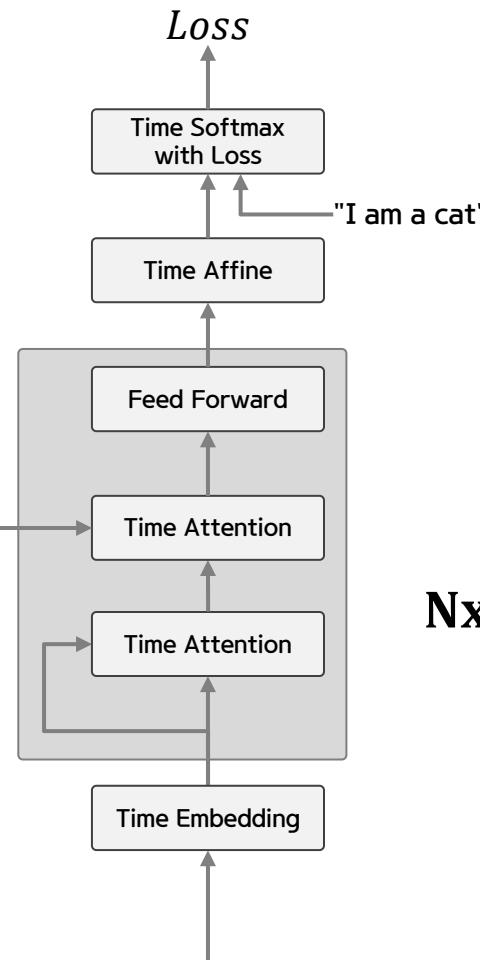
N_x



"나는 고양이로소이다."

GPT3

N_x



"<sos> I am a cat"

RNN의 단점 중 하나는 병렬처리다.

RNN은 이전 시각에 계산한 결과를 이용하여 순서대로 계산하기 때문에
RNN의 계산을 시간방향으로 병렬계산하기란 기본적으로 불가능하다.

이는 딥러닝 학습이 GPU를 사용한 병렬계산환경에서 이뤄진다는 점을 생각할 때 큰 병목이다.

이것의 해결방안을 제안한 연구 중 트랜스포머 기법이 유명하다.

셀프 어텐션 기술을 이용해 어텐션을 구성하는 것이 핵심이다.
'하나의 시계열 데이터 내에서 각 원소가 다른 원소들과 어떻게 관련되는지'를 살펴보자는 취지다.

앞서 보았던 어텐션에서는 2개의 서로 다른 시계열 데이터(`hs_enc`, `hs_dec`) 사이의 대응관계를 구했으나,
셀프 어텐션은 두 입력선이 하나의 시계열 데이터로부터 나온다.

트랜스포머는 RNN 대신 어텐션을 사용한다.
`encoder`, `decoder` 모두 셀프 어텐션을 사용한다.

또 피드포워드 신경망(시간 방향으로 독립적으로 처리하는 신경망)을 넣는다.

은닉층 1개, 활성화 함수로 ReLU를 이용해 완전연결계층 신경망을 이용한다.

9. 트랜스 포머

9.1 트랜스 포머(Transformer)

트랜스포머(Transformer)는 2017년 구글이 발표한 논문인 "Attention is all you need"에서 나온 모델로 기존의 seq2seq의 구조인 인코더-디코더를 따르면서도, 논문의 이름처럼 어텐션(Attention)만으로 구현한 모델입니다. 이 모델은 RNN을 사용하지 않고, 인코더-디코더 구조를 설계하였음에도 성능도 RNN보다 우수하다는 특징을 갖고있습니다.

트랜스포머에 대해서 배우기 전에 기존의 seq2seq를 상기해봅시다. 기존의 seq2seq 모델은 인코더-디코더 구조로 구성되어져 있었습니다. 여기서 인코더는 입력 시퀀스를 하나의 벡터 표현으로 압축하고, 디코더는 이 벡터 표현을 통해서 출력 시퀀스를 만들어냅니다. 하지만 이러한 구조는 인코더가 입력 시퀀스를 하나의 벡터로 압축하는 과정에서 입력 시퀀스의 정보가 일부 손실된다는 단점이 있었고, 이를 보정하기 위해 어텐션이 사용되었습니다. 그런데 어텐션을 RNN의 보정을 위한 용도가 아니라 아예 어텐션으로 인코더와 디코더를 만들어보면 어떨까요?

시작에 앞서 트랜스포머의 하이퍼파라미터를 정의하고자 합니다. 각 하이퍼파라미터의 의미에 대해서는 뒤에서 설명하기로하고, 여기서는 트랜스포머에는 이러한 하이퍼파라미터가 존재한다는 정도로만 이해해보도록 하겠습니다. 아래에서 정의하는 수치값은 트랜스포머를 제안한 논문에서 사용한 수치값으로 하이퍼파라미터는 사용자가 모델 설계시 임의로 변경할 수 있는 값들입니다.

$d_{model} = 512$

트랜스포머의 인코더와 디코더에서의 정해진 입력과 출력의 크기를 의미합니다. 임베딩 벡터의 차원 또한 512이며, 각 인코더와 디코더가 다음 층의 인코더와 디코더로 값을 보낼 때에도 이 차원을 유지합니다. 논문에서는 512입니다.

$num_layers = 6$

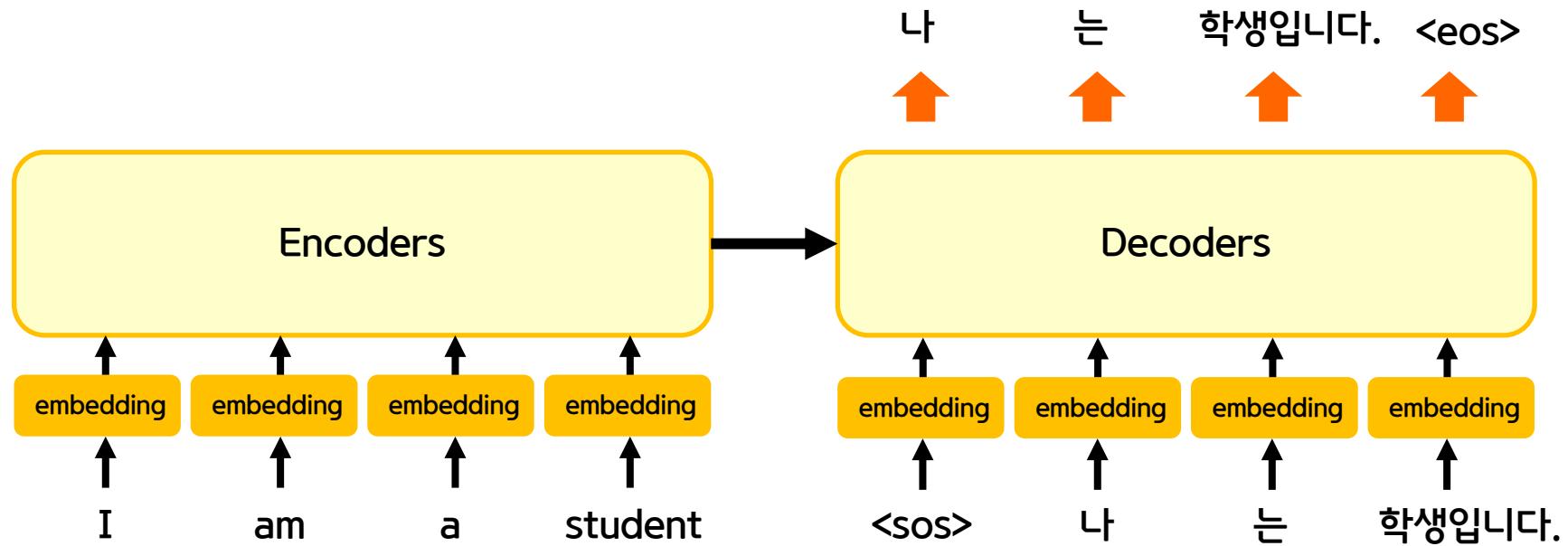
트랜스포머에서 하나의 인코더와 디코더를 층으로 생각하였을 때, 트랜스포머 모델에서 인코더와 디코더가 총 몇 층으로 구성되었는지를 의미합니다. 논문에서는 인코더와 디코더를 각각 총 6개 쌓았습니다.

$numheads = 8$

트랜스포머에서는 어텐션을 사용할 때, 1번 하는 것 보다 여러 개로 분할해서 병렬로 어텐션을 수행하고 결과값을 다시 하나로 합치는 방식을 택했습니다. 이때 이 병렬의 개수를 의미합니다.

$d_{ff} = 2048$

트랜스포머 내부에는 피드 포워드 신경망이 존재합니다. 이때 은닉층의 크기를 의미합니다. 피드 포워드 신경망의 입력층과 출력층의 크기는 입니다.

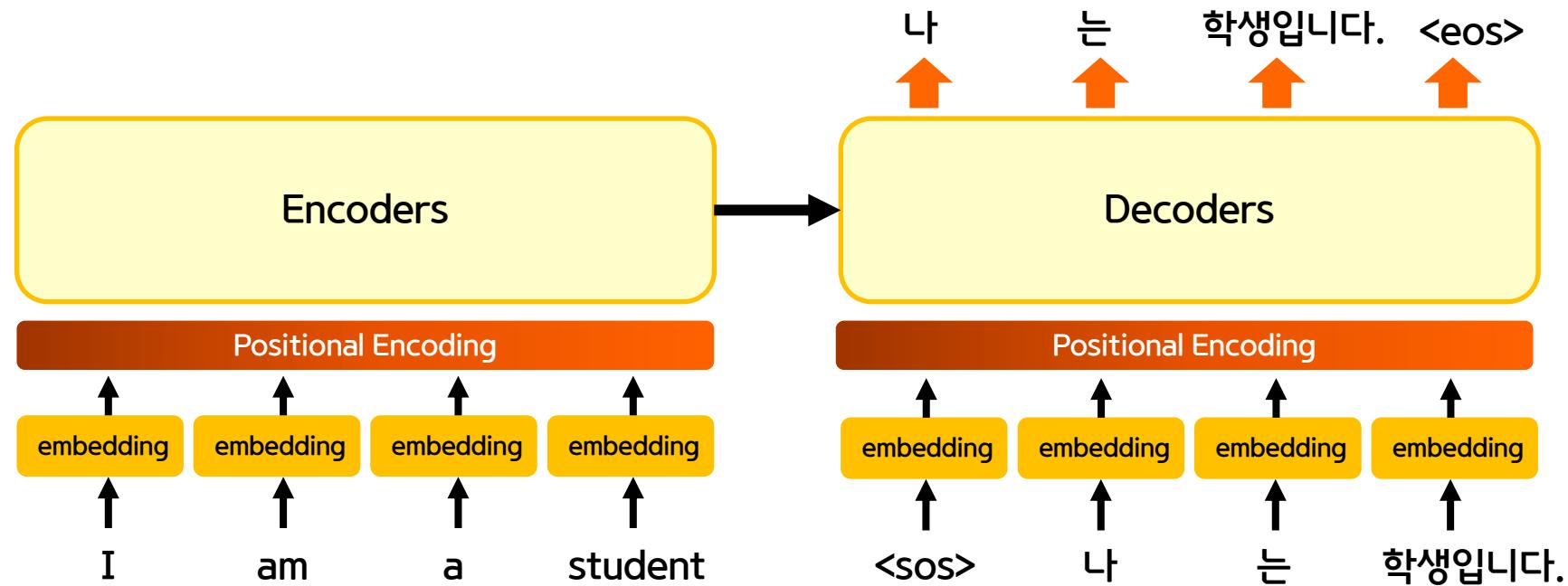


위의 그림은 인코더로부터 정보를 전달받아 디코더가 출력 결과를 만들어내는 트랜스포머 구조를 보여줍니다. 디코더는 마치 기존의 seq2seq 구조처럼 시작 심볼 <sos>를 입력으로 받아 종료 심볼 <eos>가 나올 때까지 연산을 진행합니다. 이는 RNN은 사용되지 않지만 여전히 인코더-디코더의 구조는 유지되고 있음을 보여줍니다.

이제 트랜스포머의 내부 구조를 조금씩 확대해가는 방식으로 트랜스포머를 이해해봅시다. 우선 인코더와 디코더의 구조를 이해하기 전에 트랜스포머의 입력에 대해서 이해해보겠습니다. 트랜스포머의 인코더와 디코더는 단순히 각 단어의 임베딩 벡터들을 입력받는 것이 아니라 임베딩 벡터에서 조정된 값을 입력받는데 이에 대해서 알아보기 위해 입력 부분을 확대해보겠습니다.

트랜스포머의 내부를 이해하기 전에 우선 트랜스포머의 입력에 대해서 알아보겠습니다. RNN이 자연어 처리에서 유용했던 이유는 단어의 위치에 따라 단어를 순차적으로 입력받아서 처리하는 RNN의 특성으로 인해 각 단어의 위치 정보(position information)를 가질 수 있다는 점에 있었습니다.

하지만 트랜스포머는 단어 입력을 순차적으로 받는 방식이 아니므로 단어의 위치 정보를 다른 방식으로 알려줄 필요가 있습니다. 트랜스포머는 단어의 위치 정보를 얻기 위해서 각 단어의 임베딩 벡터에 위치 정보들을 더하여 모델의 입력으로 사용하는데, 이를 포지셔널 인코딩(positional encoding)이라고 합니다.



D=6

 $(T, D) \Rightarrow (4, 6)$

'i'	5	4	3	5	3	2
'am'	3	4	3	1	3	2
'a'	3	2	3	5	2	2
'student'	7	1	6	4	3	1

 $(T, D) \Rightarrow (50, 128)$

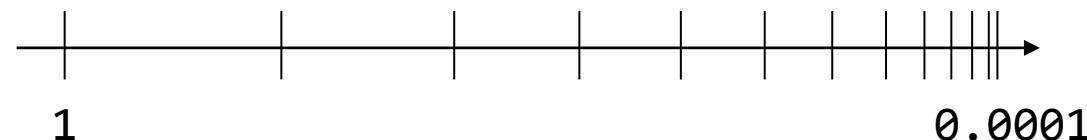
i.shape=(1,128)

$$\frac{1}{2 * \left(\frac{i}{2}\right)}$$

$$10000^{\frac{1}{d_{model}}}$$

i	0	1	2	3	4	5	6	7	8	9
angle_rads	0	0	0	0	0	0	0	0	0	0
	1	1	0.86	0.86	0.74	0.74	0.64	0.64	0.56	0.56
	2	2	1.73	1.73	1.49	1.49	1.29	1.29	1.12	1.12

```
angles = 1 / tf.pow(10000, (2 * (i // 2)) / tf.cast(d_model, tf.float32))
```



```
angle_rads[:,0::2]
```

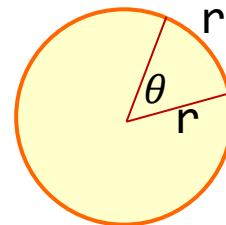
0	0	0	0	0
1	0.86	0.74	0.64	0.56
2	1.73	1.49	1.29	1.12

```
tf.math.sin(90)
```

※ PositionalEncoding 코드 참조

angle_rads[:,0::2]

0	0	0	0	0
1	0.86	0.74	0.64	0.56
2	1.73	1.49	1.29	1.12



np.sin(x)

$$2 * \pi * r * \frac{\theta}{360} = r$$

$$\theta = 180/\pi$$

$$\frac{180}{\pi}: 1 = \theta: rad$$

$$\frac{180}{\pi} * rad = \theta$$

$$rad = \theta * \frac{\pi}{180}$$

$$PE_{(pos,2i)} = \sin \left(\frac{pos}{10000 \frac{2i}{d_{model}} * 2\pi} * \frac{\pi}{180} \right)$$

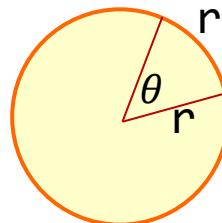
$$PE_{(pos,2i)} = \sin \left(\frac{pos}{(10000 \frac{2*(\frac{i}{2})}{d_{model}})} \right)$$

1~10000

※ PositionalEncoding 코드 참조

`angle_rads[:,1::2]`

0	0	0	0	0
1	0.86	0.74	0.64	0.56
2	1.73	1.49	1.29	1.12



`np.sin(x)`

$$2 * \pi * r * \frac{\theta}{360} = r$$

$$\theta = 180/\pi$$

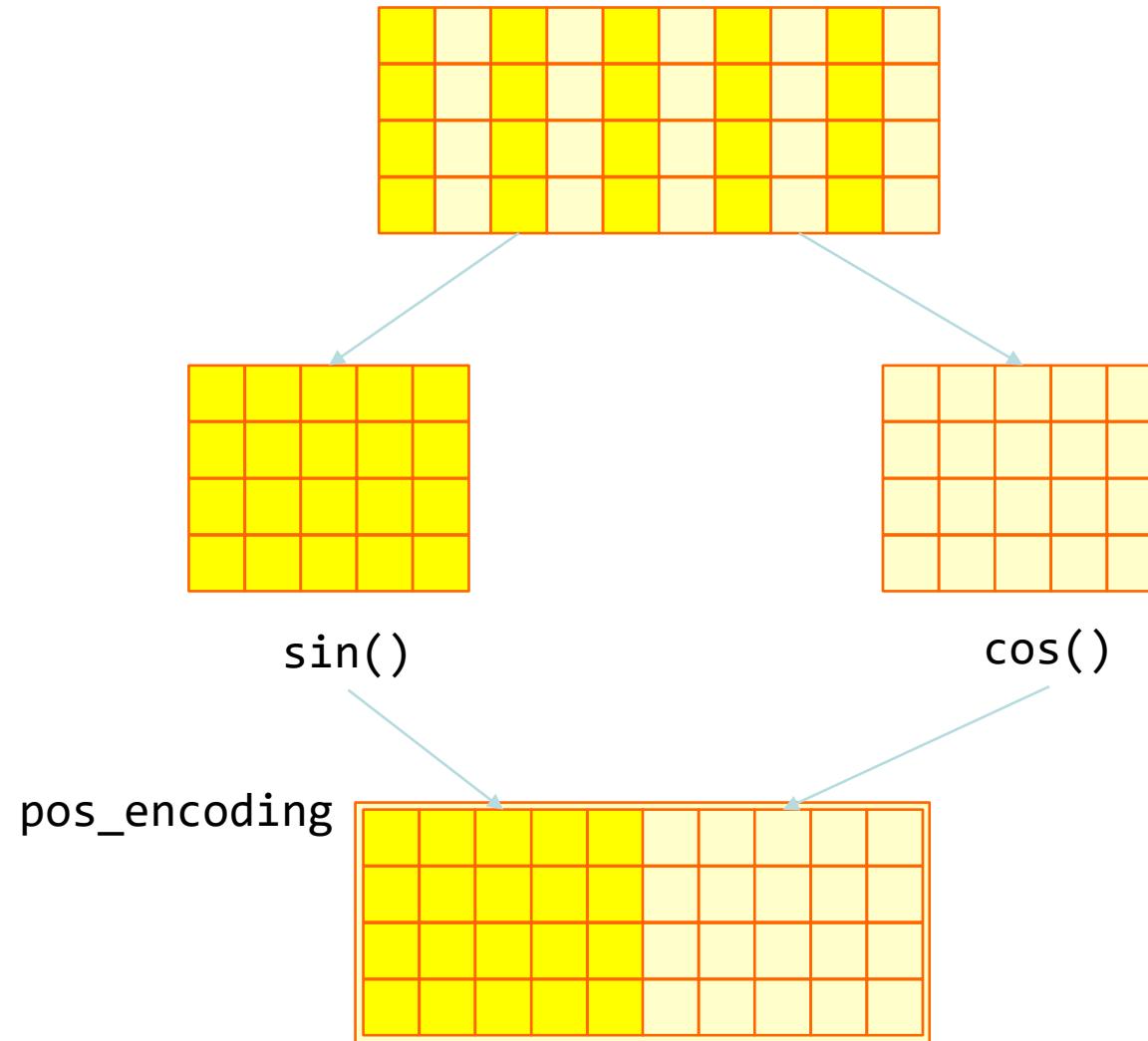
$$\frac{180}{\pi} : 1 = \theta : rad$$

$$\frac{180}{\pi} * rad = \theta$$

$$rad = \theta * \frac{\pi}{180}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{(10000^{\frac{2*(\frac{i}{2})}{d_{model}}})}\right)$$

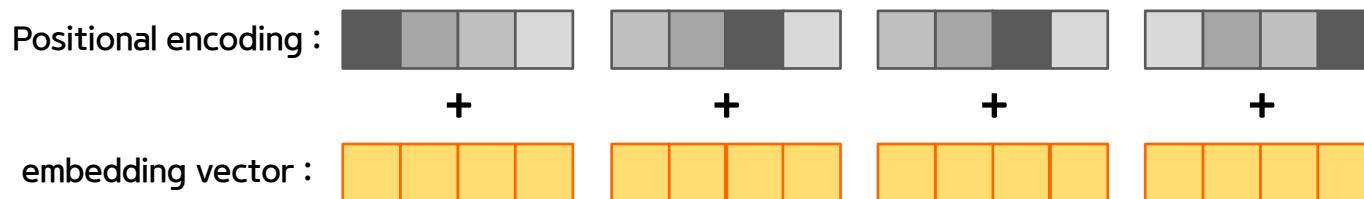
※ PositionalEncoding 코드 참조



포지셔널 인코딩(Positional Encoding)

9.1 트랜스 포머(Transformer)

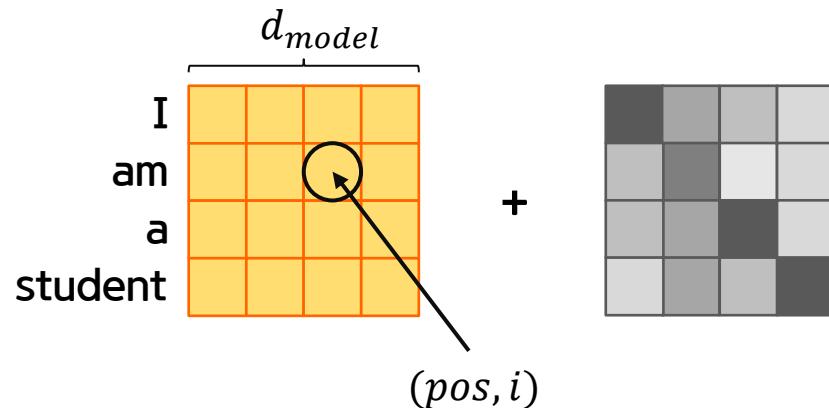
그림은 입력으로 사용되는 임베딩 벡터들이 트랜스포머의 입력으로 사용되기 전에 포지셔널 인코딩값이 더해지는 것을 보여줍니다. 임베딩 벡터가 인코더의 입력으로 사용되기 전에 포지셔널 인코딩값이 더해지는 과정을 시각화하면 아래와 같습니다.



포지셔널 인코딩 값들은 어떤 값이기에 위치 정보를 반영해줄 수 있는 것일까요? 트랜스포머는 위치 정보를 가진 값을 만들기 위해서 아래의 두 개의 함수를 사용합니다.

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \quad PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i+1}{d_{model}}}}\right)$$

사인 함수와 코사인 함수의 그래프를 상기해보면 요동치는 값의 형태를 생각해볼 수 있는데, 트랜스포머는 사인 함수와 코사인 함수의 값을 임베딩 벡터에 더해주므로 단어의 순서 정보를 더하여 줍니다. 그런데 위의 두 함수에는 pos, i, d_{model} 등의 생소한 변수들이 있습니다. 위의 함수를 이해하기 위해서는 위에서 본 임베딩 벡터와 포지셔널 인코딩의 덧셈은 사실 임베딩 벡터가 모여 만들어진 문장 벡터 행렬과 포지셔널 인코딩 행렬의 덧셈 연산을 통해 이루어진다는 점을 이해해야 합니다.

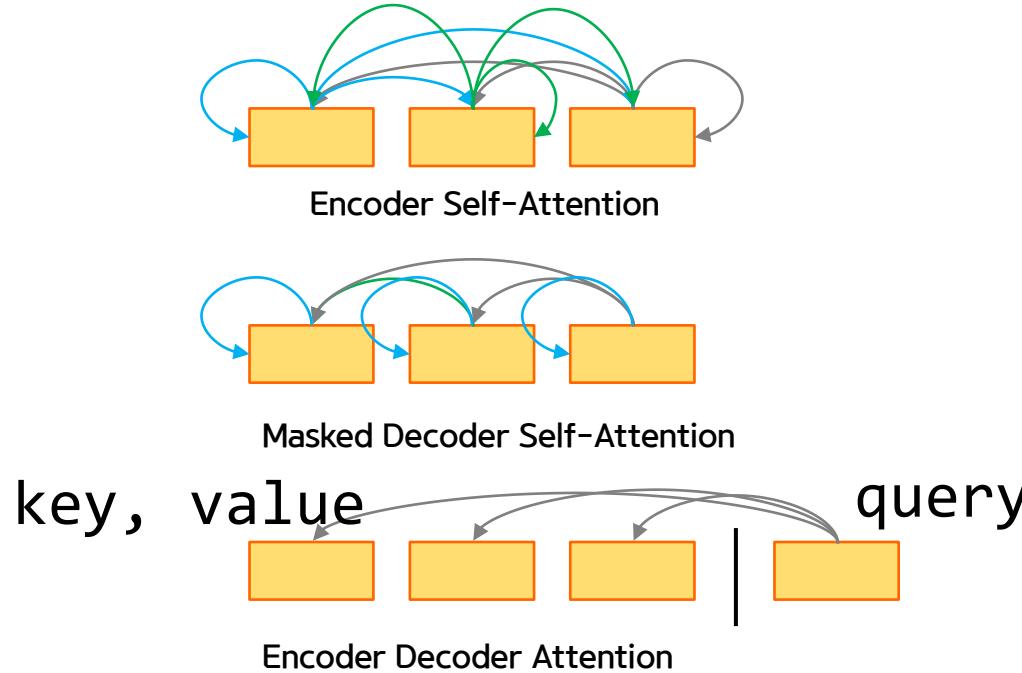


pos 는 입력 문장에서의 임베딩 벡터의 위치를 나타내며, i 는 임베딩 벡터 내의 차원의 인덱스를 의미합니다. 위의 식에 따르면 임베딩 벡터 내의 각 차원의 인덱스가 짝수인 경우에는 사인 함수의 값을 사용하고 홀수인 경우에는 코사인 함수의 값을 사용합니다. 위의 수식에서 $(pos, 2_i)$ 일 때는 사인 함수를 사용하고, $(pos, 2_i + 1)$ 일 때는 코사인 함수를 사용하고 있음을 주목합시다.

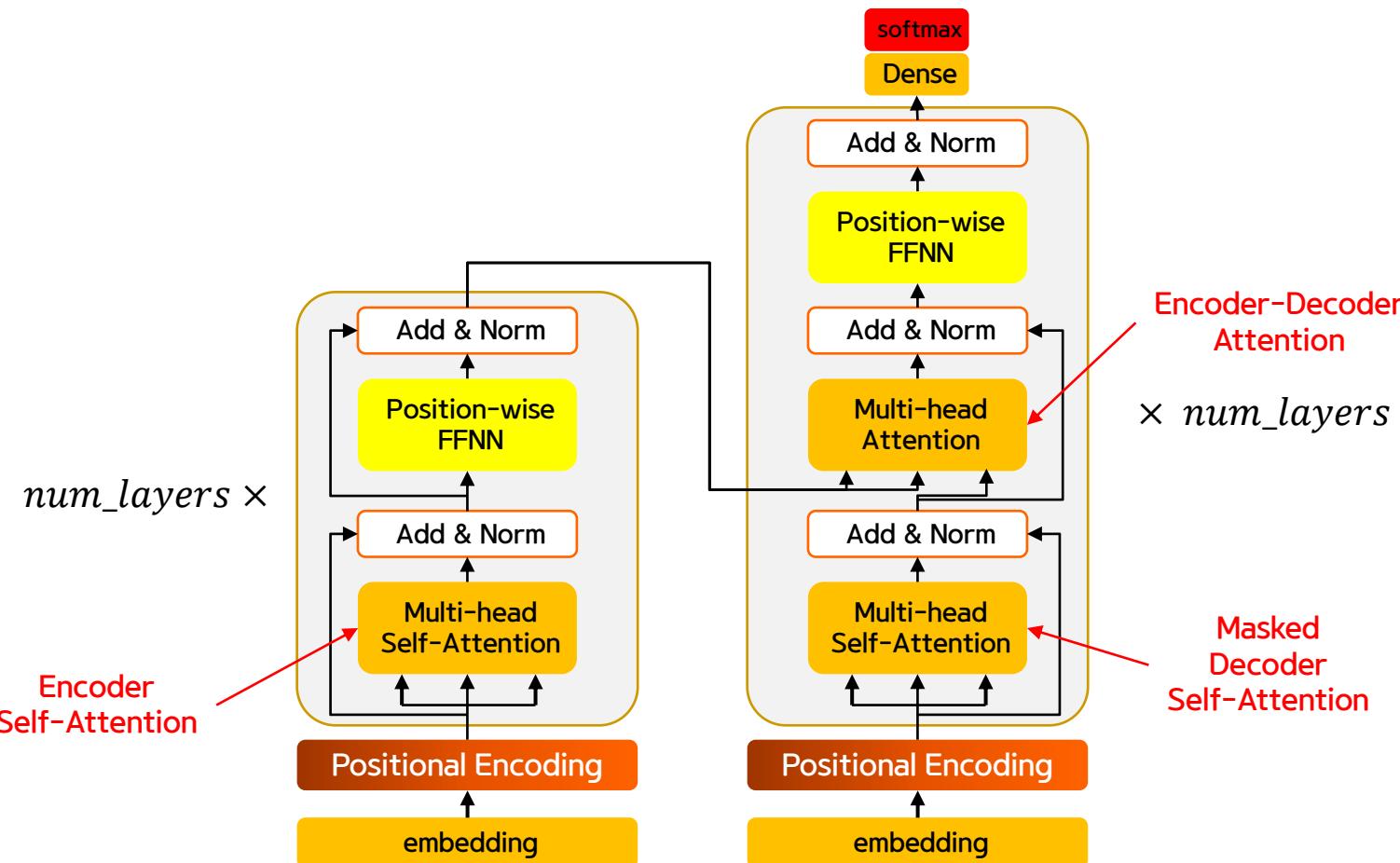
또한 위의 식에서 d_{model} 은 트랜스포머의 모든 층의 출력 차원을 의미하는 트랜스포머의 하이퍼파라미터입니다. 앞으로 보게 될 트랜스포머의 각종 구조에서 d_{model} 의 값이 계속해서 등장하는 이유입니다. 임베딩 벡터 또한 d_{model} 의 차원을 가지는데 위의 그림에서는 마치 4로 표현되었지만 실제 논문에서는 512의 값을 가집니다.

위와 같은 포지셔널 인코딩 방법을 사용하면 순서 정보가 보존되는데, 예를 들어 각 임베딩 벡터에 포지셔널 인코딩값을 더하면 같은 단어라고 하더라도 문장 내의 위치에 따라서 트랜스포머의 입력으로 들어가는 임베딩 벡터의 값이 달라집니다. 결국 트랜스포머의 입력은 순서 정보가 고려된 임베딩 벡터라고 보면 되겠습니다.

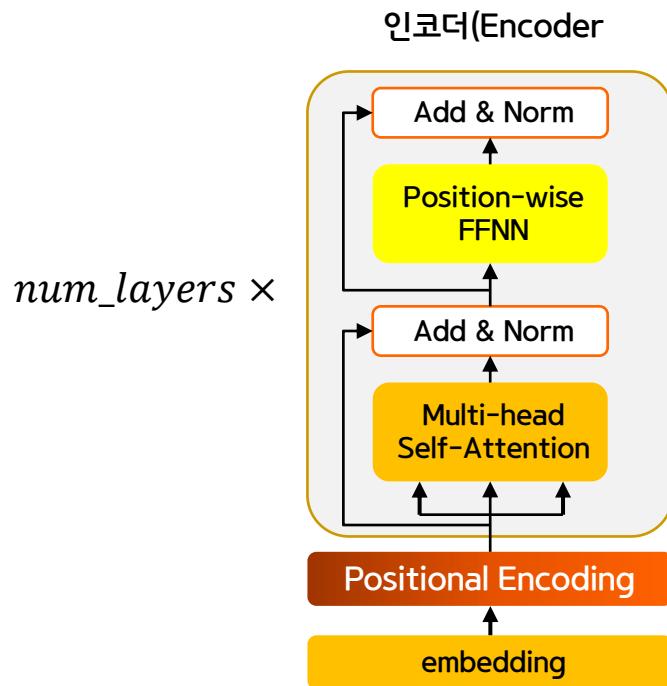
트랜스포머에서 사용되는 세 가지의 어텐션에 대해서 간단히 정리해봅시다. 지금은 큰 그림을 이해하는 것에만 집중합니다.



첫번째 그림인 셀프 어텐션은 인코더에서 이루어지지만, 두번째 그림인 셀프 어텐션과 세번째 그림인 인코더-디코더 어텐션은 디코더에서 이루어집니다. 셀프 어텐션은 본질적으로 Query, Key, Value가 동일한 경우를 말합니다. 반면, 세번째 그림 인코더-디코더 어텐션에서는 Query가 디코더의 벡터인 반면에 Key와 Value가 인코더의 벡터이므로 셀프 어텐션이라고 부르지 않습니다.

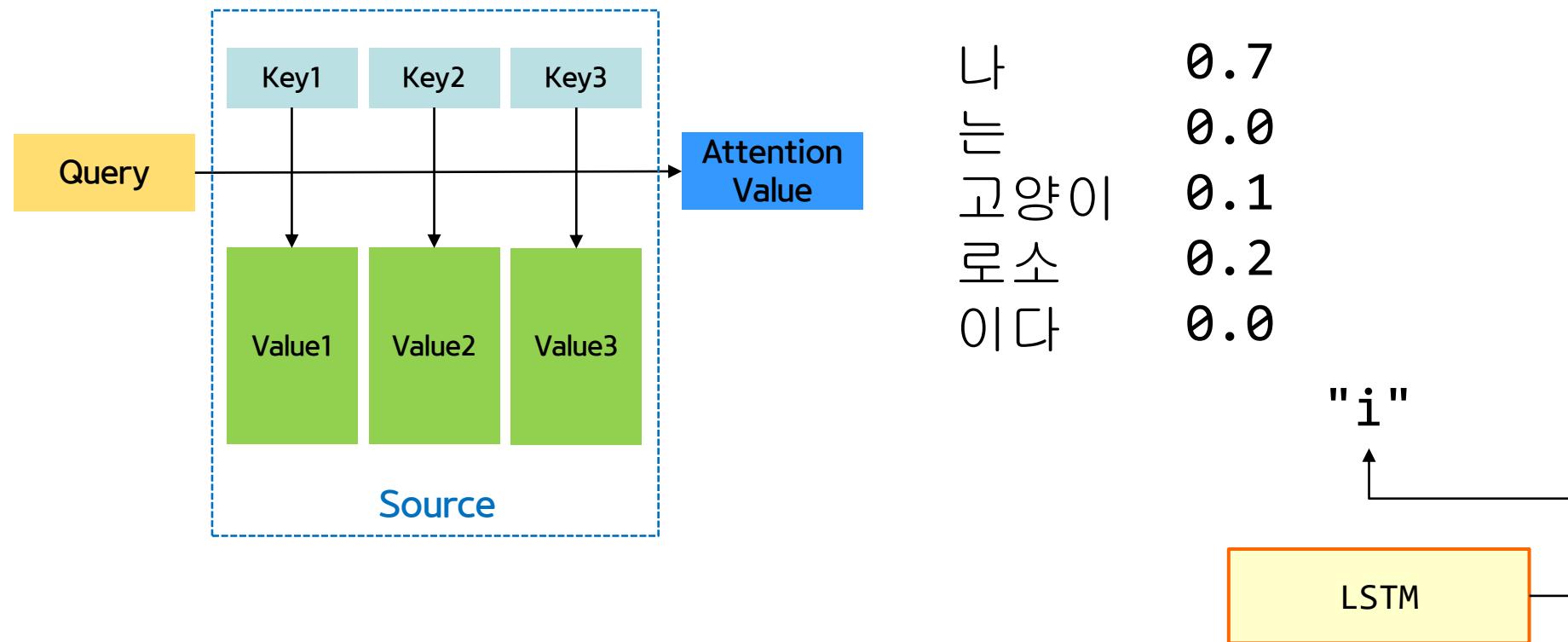


위 그림은 트랜스포머의 아키텍처에서 세 가지 어텐션이 각각 어디에서 이루어지는지를 보여줍니다. 세 개의 어텐션에 추가적으로 '멀티 헤드'라는 이름이 붙어있습니다. 뒤에서 설명하겠지만, 이는 트랜스포머가 어텐션을 병렬적으로 수행하는 방법을 의미합니다.

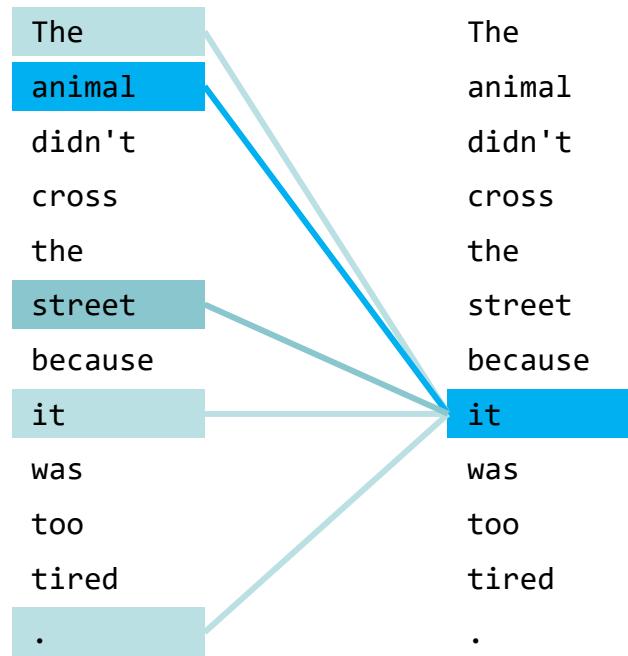


트랜스포머는 하이퍼파라미터인 num_layers 개수의 인코더 층을 쌓습니다. 논문에서는 총 6개의 인코더 층을 사용하였습니다. 인코더를 하나의 층이라는 개념으로 생각한다면, 하나의 인코더 층은 크게 총 2개의 서브층(sublayer)으로 나뉘어집니다. 바로 셀프 어텐션과 피드 포워드 신경망입니다. 위의 그림에서는 멀티 헤드 셀프 어텐션과 포지션 와이즈 피드 포워드 신경망이라고 적혀있지만, 멀티 헤드 셀프 어텐션은 셀프 어텐션을 병렬적으로 사용하였다는 의미고, 포지션 와이즈 피드 포워드 신경망은 우리가 알고 있는 일반적인 피드 포워드 신경망입니다. 우선 셀프 어텐션에 대해서 알아봅시다.

어텐션 함수는 주어진 '쿼리(Query)'에 대해서 모든 '키(Key)'와의 유사도를 각각 구합니다. 그리고 구해낸 이 유사도를 가중치로 하여 키와 맵핑되어있는 각각의 '값(Value)'에 반영해줍니다. 그리고 유사도가 반영된 '값(Value)'을 모두 가중합하여 리턴합니다.



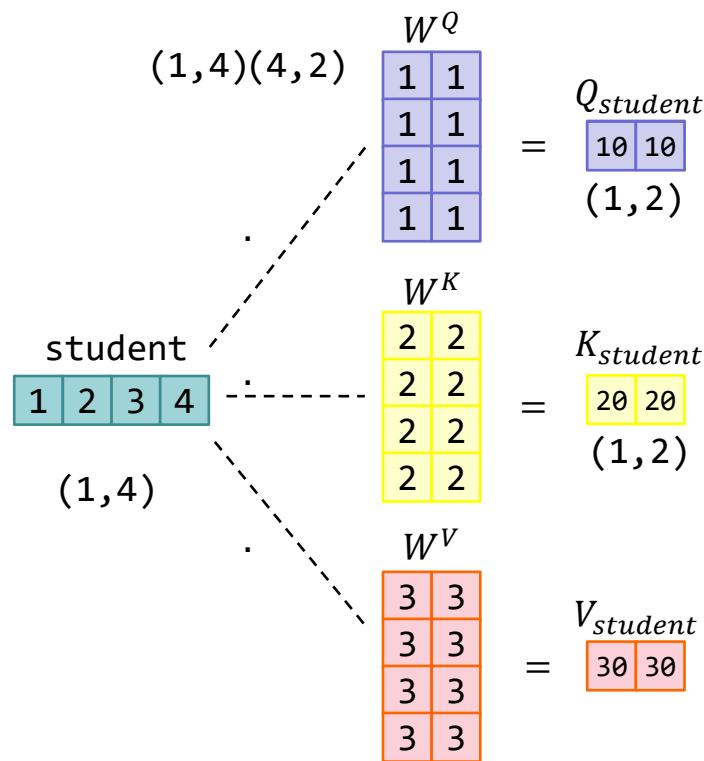
셀프 어텐션에 대한 구체적인 사항을 배우기 전에 셀프 어텐션을 통해 얻을 수 있는 대표적인 효과에 대해서 이해해봅시다.



위의 그림은 트랜스포머에 대한 구글 AI 블로그 포스트에서 가져왔습니다. 위의 예시 문장을 번역하면 '그 동물은 길을 건너지 않았다. 왜냐하면 그것은 너무 피곤하였기 때문이다.' 라는 의미가 됩니다. 그런데 여기서 그것(it)에 해당하는 것은 과연 길(street)일까요? 동물(animal)일까요? 우리는 피곤한 주체가 동물이라는 것을 아주 쉽게 알 수 있지만 기계는 그렇지 않습니다. 하지만 셀프 어텐션은 입력 문장 내의 단어들끼리 유사도를 구하므로서 그것(it)이 동물(animal)과 연관되었을 확률이 높다는 것을 찾아냅니다.

앞서 셀프 어텐션은 입력 문장의 단어 벡터들을 가지고 수행한다고 하였는데, 사실 셀프 어텐션은 인코더의 초기 입력인 d_{model} 의 차원을 가지는 단어 벡터들을 사용하여 셀프 어텐션을 수행하는 것이 아니라 우선 각 단어 벡터들로부터 Q벡터, K벡터, V벡터를 얻는 작업을 거칩니다. 이때 이 Q벡터, K벡터, V벡터들은 초기 입력인 d_{model} 의 차원을 가지는 단어 벡터들보다 더 작은 차원을 가지는데, 논문에서는 $d_{model} = 512$ 의 차원을 가졌던 각 단어 벡터들을 64의 차원을 가지는 Q벡터, K벡터, V벡터로 변환하였습니다.

64라는 값은 트랜스포머의 또 다른 하이퍼파라미터인 num_heads 로 인해 결정되는데, 트랜스포머는 d_{model} 을 num_heads 로 나눈 값을 각 Q벡터, K벡터, V벡터의 차원으로 결정합니다. 논문에서는 num_heads 를 8로 하였습니다. 이제 그림을 통해 이해해봅시다. 예를 들어 여기서 사용하고 있는 예문 중 student라는 단어 벡터를 Q, K, V의 벡터로 변환하는 과정을 보겠습니다.



기존의 벡터로부터 더 작은 벡터는 가중치 행렬을 곱하므로서 완성됩니다. 각 가중치 행렬은 ($d_{model}, \frac{d_{model}}{num_heads}$)의 크기를 가집니다. 이 가중치 행렬은 훈련 과정에서 학습됩니다. 즉, 논문과 같이 $d_{model} = 512$ 이고 $num_heads=8$ 라면, 각 벡터에 3개의 서로 다른 가중치 행렬을 곱하고 64의 크기를 가지는 Q, K, V 벡터를 얻어냅니다. 위의 그림은 단어 벡터 중 student 벡터로부터 Q, K, V 벡터를 얻어내는 모습을 보여줍니다. 모든 단어 벡터에 위와 같은 과정을 거치면 I, am, a, student는 각각의 Q, K, V 벡터를 얻습니다.

Q , K , V 벡터를 얻었다면 지금부터는 기존에 배운 어텐션 메커니즘과 동일합니다. 각 Q 벡터는 모든 K 벡터에 대해서 어텐션 스코어를 구하고, 어텐션 분포를 구한 뒤에 이를 사용하여 모든 V 벡터를 가중합하여 어텐션 값 또는 컨텍스트 벡터를 구하게 됩니다. 그리고 이를 모든 Q 벡터에 대해서 반복합니다.

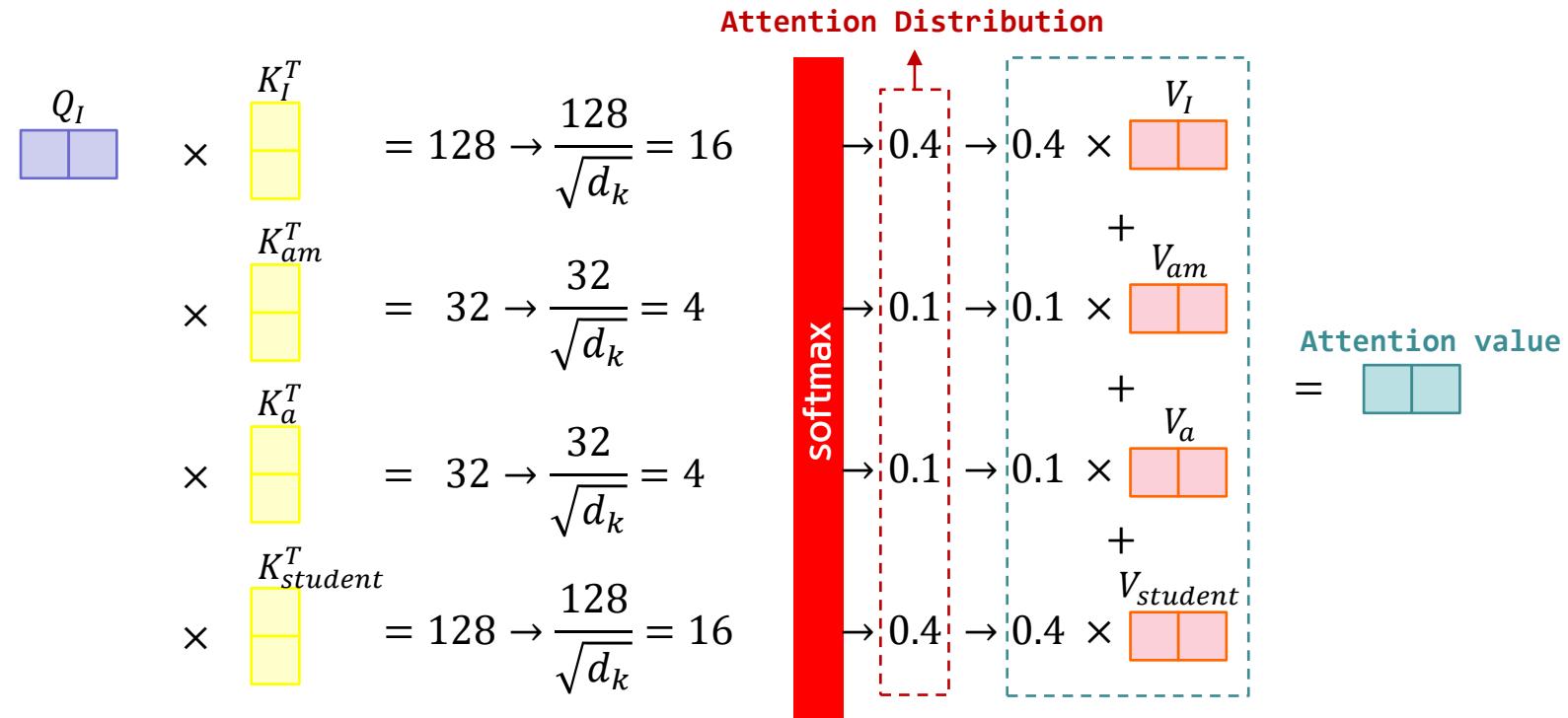
그런데 앞서 어텐션 챕터에서 어텐션 함수의 종류는 다양하다고 언급한 바 있습니다. 트랜스포머에서는 어텐션 챕터에 사용했던 내적만을 사용하는 어텐션 함수 $score(q, k) = q \cdot k$ 가 아니라 여기에 특정값으로 나눠준 어텐션 함수인 $score(q, k) = q \cdot k / \sqrt{n}$ 를 사용합니다. 이러한 함수를 사용하는 어텐션을 어텐션 챕터에서 배운 닷-프로덕트 어텐션(dot-product attention)에서 값을 스케일링하는 것을 추가하였다고 하여 스케일드 닷-프로덕트 어텐션(Scaled dot-product Attention)이라고 합니다. 이제 그림을 통해 이해해봅시다.

$(1, 2)(1, 2)$ $\cancel{(1, 2)(2, 1)}$	$Q_I \times K_I^T = 128 \rightarrow \frac{128}{\sqrt{d_k}} = 16$
	$\times K_{am}^T = 32 \rightarrow \frac{32}{\sqrt{d_k}} = 4$
	$\times K_a^T = 32 \rightarrow \frac{32}{\sqrt{d_k}} = 4$
	$\times K_{student}^T = 128 \rightarrow \frac{128}{\sqrt{d_k}} = 16$

Attention Score

우선 단어 I에 대한 Q벡터를 기준으로 설명해보겠습니다. 지금부터 설명하는 과정은 am에 대한 Q벡터, a에 대한 Q벡터, student에 대한 Q벡터에 대해서도 모두 동일한 과정을 거칩니다. 위의 그림은 단어 I에 대한 Q 벡터가 모든 K 벡터에 대해서 어텐션 스코어를 구하는 것을 보여줍니다. 위의 128과 32는 저자가 임의로 가정한 수치로 신경쓰지 않아도 좋습니다.

위의 그림에서 어텐션 스코어는 각각 단어 I가 단어 I, am, a, student와 얼마나 연관되어 있는지를 보여주는 수치입니다. 트랜스포머에서는 두 벡터의 내적값을 스케일링하는 값으로 K 벡터의 차원을 나타내는 d_k 에 루트를 씌운 $\sqrt{d_k}$ 사용하는 것을 택했습니다. 앞서 언급하였듯이 논문에서 d_k 는 라는 식에 따라서 64의 값을 가지므로 $\sqrt{d_k}$ 는 8의 값을 가집니다.

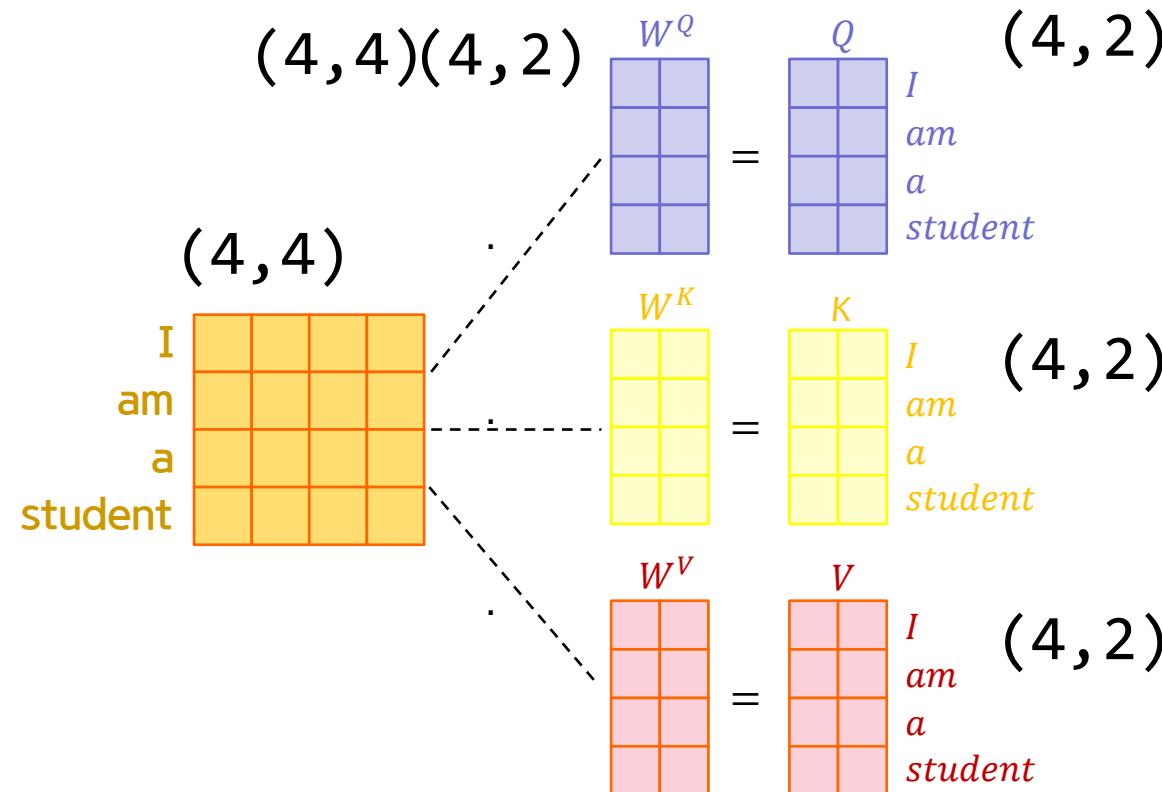


행렬 연산으로 일괄 처리하기

9.1 트랜스 포머(Transformer)

사실 각 단어에 대한 Q, K, V 벡터를 구하고 스케일드 닷-프로덕트 어텐션을 수행하였던 위의 과정들은 벡터 연산이 아니라 행렬 연산을 사용하면 일괄 계산이 가능합니다. 지금까지 벡터 연산으로 설명하였던 이유는 이해를 돋기 위한 과정이고, 실제로는 행렬 연산으로 구현됩니다.

위의 과정을 벡터가 아닌 행렬 연산으로 이해해봅시다. 우선, 각 단어 벡터마다 일일히 가중치 행렬을 곱하는 것이 아니라 문장 행렬에 가중치 행렬을 곱하여 Q 행렬, K 행렬, V행렬을 구합니다.



행렬 연산으로 일괄 처리하기

9.1 트랜스 포머(Transformer)

이제 행렬 연산을 통해 어텐션 스코어는 어떻게 구할 수 있을까요? 여기서 Q 행렬을 K 행렬을 전치한 행렬과 곱해준다고 해봅시다. 이렇게 되면 각각의 단어의 Q벡터와 K벡터의 내적이 각 행렬의 원소가 되는 행렬이 결과로 나옵니다.

$$\begin{array}{c} \text{Q} \\ \begin{matrix} I & \text{(4,2)} \\ am & \cdot \\ a & \\ student & \end{matrix} \end{array} \quad \begin{array}{c} (2,4) \\ I \ am \ a \ student \\ K^T \end{array} = \begin{array}{c} (4,4) \\ I \ am \ a \ student \\ I \\ am \\ a \\ student \end{array} \quad \begin{array}{c} (4,2) \\ V \\ \end{array} = \begin{array}{c} (4,2) \\ \text{Attention value Matrix a} \end{array}$$

다시 말해 위의 그림의 결과 행렬의 값에 전체적으로 $\sqrt{d_k}$ 를 나누어주면 이는 각 행과 열이 어텐션 스코어값을 가지는 행렬이 됩니다. 예를 들어 I 행과 student 열의 값은 I의 Q벡터와 student의 K 벡터의 어텐션 스코어와 동일한 행렬이 된다는 것입니다. 즉, 어텐션 스코어 행렬입니다. 어텐션 스코어 행렬을 구하였다면 남은 것은 어텐션 분포를 구하고, 이를 사용하여 모든 단어에 대한 어텐션 값을 구하는 일입니다. 이는 간단하게 어텐션 스코어 행렬에 소프트 맥스 함수를 사용하고, V 행렬을 곱하는 것으로 해결됩니다. 이렇게 되면 각 단어의 어텐션 값을 모두 가지는 어텐션 값 행렬이 결과로 나옵니다.

$$\begin{array}{c} \text{Q} \\ \begin{matrix} I & \text{(4,4)} \\ am & \cdot \\ a & \\ student & \end{matrix} \end{array} \quad \begin{array}{c} K^T \\ I \ am \ a \end{array} \quad \begin{array}{c} (4,2) \\ V \\ \end{array} = \begin{array}{c} \text{Attention value Matrix a} \end{array}$$

행렬 연산으로 일괄 처리하기

9.1 트랜스 포머(Transformer)

위의 그림은 행렬 연산을 통해 모든 값이 일괄 계산되는 과정을 식으로 보여줍니다. 해당 식은 실제 트랜스포머 논문에 기재된 아래의 수식과 정확하게 일치하는 식입니다.

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) V$$

위의 행렬 연산에 사용된 행렬의 크기를 모두 정리해봅시다. 우선 입력 문장의 길이를 seq_len 이라고 해봅시다. 그렇다면 문장 행렬의 크기는 $(\text{seq_len}, d_{\text{model}})$ 입니다. 여기에 3개의 가중치 행렬을 곱해서 Q, K, V 행렬을 만들어야 합니다.

우선 행렬의 크기를 정의하기 위해 행렬의 각 행에 해당되는 Q 벡터와 K 벡터의 크기를 d_k 라고 하고, V 벡터의 크기를 d_v 라고 해봅시다. 그렇다면 Q 행렬과 K 행렬의 크기는 $(\text{seq_len}, d_k)$ 이며, V 행렬의 크기는 $(\text{seq_len}, d_v)$ 가 되어야 합니다. 그렇다면 문장 행렬과 Q, K, V 행렬의 크기로부터 가중치 행렬의 크기 추정이 가능합니다. W^Q 와 W^K 는 (d_{model}, d_k) 의 크기를 가지며, W^V 는 (d_{model}, d_v) 의 크기를 가집니다. 단, 논문에서는 d_k 와 d_v 의 크기는 $\frac{d_{\text{model}}}{\text{num_heads}}$ 와 같습니다. 즉, $\frac{d_{\text{model}}}{\text{num_heads}} = d_k = d_v$ 입니다.

결과적으로

$\text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) V$ 식을 적용하여 나오는 어텐션 값 행렬 a 의 크기는 $(\text{seq_len}, d_v)$ 이 됩니다.

```
def scaled_dot_product_attention(query, key, value, mask):  
    matmul_qk = tf.matmul(query, key, transpose_b=True)  
  
    depth = tf.cast(tf.shape(key)[-1], tf.float32)  
    logits = matmul_qk / tf.math.sqrt(depth)  
  
    if mask is not None:  
        logits += (mask * -1e9)  
  
    attention_weights = tf.nn.softmax(logits, axis=-1)  
  
    output = tf.matmul(attention_weights, value)  
    return output, attention_weights
```

코드는 위의 내용을 이해했다면 어렵지 않습니다. Q 행렬과 K 행렬을 전치한 행렬을 곱하고, 소프트맥스 함수를 사용하여 어텐션 분포 행렬을 얻은 뒤에 V 행렬과 곱합니다. 코드에서 mask가 사용되는 if문은 아직 배우지 않은 내용으로 지금은 무시하고 넘어갑니다.

scaled_dot_product_attention 함수가 정상 작동하는지 테스트를 해보겠습니다. 우선 temp_q, temp_k, temp_v라는 임의의 Query, Key, Value 행렬을 만들고, 이를 scaled_dot_product_attention 함수에 입력으로 넣어 함수가 리턴하는 값을 출력해볼 겁니다.

스케일드 닷-프로덕트 어텐션 구현하기

9.1 트랜스 포머(Transformer)

```
np.set_printoptions(suppress=True)
temp_k = tf.constant([[10,0,0],
[0,10,0],
[0,0,10],
[0,0,10]], dtype=tf.float32)

temp_v = tf.constant([[    1,0],
[ 10,0],
[ 100,5],
[1000,6]], dtype=tf.float32)
temp_q = tf.constant([[0, 10, 0]], dtype=tf.float32)
```

temp_k
10 0 0
0 10 0
0 0 10
0 0 10

temp_q
0 10 0

temp_v
1 0
10 0
100 5
1000 6

여기서 주목할 점은 Query에 해당하는 temp_q의 값 [0, 10, 0]은 Key에 해당하는 temp_k의 두번째 값 [0, 10, 0]과 일치한다는 점입니다. 그렇다면 어텐션 분포와 어텐션 값은 어떤 값이 나올까요?

```
temp_out, temp_attn = scaled_dot_product_attention(temp_q, temp_k, temp_v, None)
print(temp_attn)
print(temp_out)
```

스케일드 닷-프로덕트 어텐션 구현하기

9.1 트랜스 포머(Transformer)

```
def scaled_dot_product_attention(query, key, value, mask):
    matmul_qk = tf.matmul(query, key, transpose_b=True)
    depth = tf.cast(tf.shape(key)[-1], tf.float32)
    logits = matmul_qk / tf.math.sqrt(depth)
    attention_weights = tf.nn.softmax(logits, axis=-1)
    output = tf.matmul(attention_weights, value)
    return output, attention_weights
```

$(1,3)(4,3) \quad (4,3)$
 ~~$(3,4)$~~

$(1,3)(3,4) \Rightarrow (1,4)$

depth

3

matmul_qk

0	100	0	0
---	-----	---	---

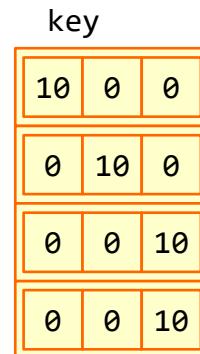
$\sqrt{3}$

= softmax(

0	57.7	0	0
---	------	---	---

) =

0	1	0	0
---	---	---	---



$$\text{query} \cdot \text{key.T} = \text{matmul_qk}$$

query:

0	10	0
---	----	---

key.T:

10	0	0	0
0	10	0	0
0	0	10	10

matmul_qk:

0	100	0	0
---	-----	---	---

$$\text{attention_weights} \cdot \text{value} = \text{out}$$

attention_weights:

0	1	0	0
---	---	---	---

value:

1	0
10	0
100	5
1000	6

out:

10	0
----	---

logits

0	57.7	0	0
---	------	---	---

$(1,4)(4,2)$

스케일드 닷-프로덕트 어텐션 구현하기

9.1 트랜스 포머(Transformer)

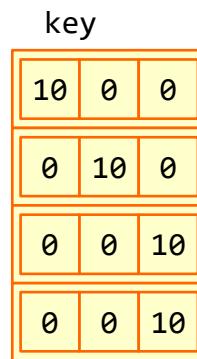
```
def scaled_dot_product_attention(query, key, value, mask):
    matmul_qk = tf.matmul(query, key, transpose_b=True)
    depth = tf.cast(tf.shape(key)[-1], tf.float32)
    logits = matmul_qk / tf.math.sqrt(depth)
    attention_weights = tf.nn.softmax(logits, axis=-1)
    output = tf.matmul(attention_weights, value)
    return output, attention_weights
```

$$(3,3)(4,3) \quad (4,3) \\ (3,4)$$

$$(3,3)(3,4) \Rightarrow (3,4)$$

depth

3

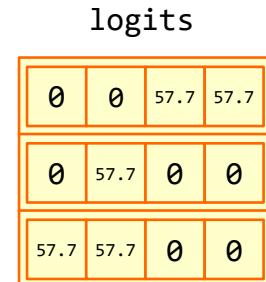


matmul_qk

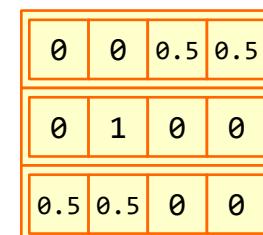
0	0	100	100
0	100	0	0
100	100	0	0

$\sqrt{3}$

= softmax(

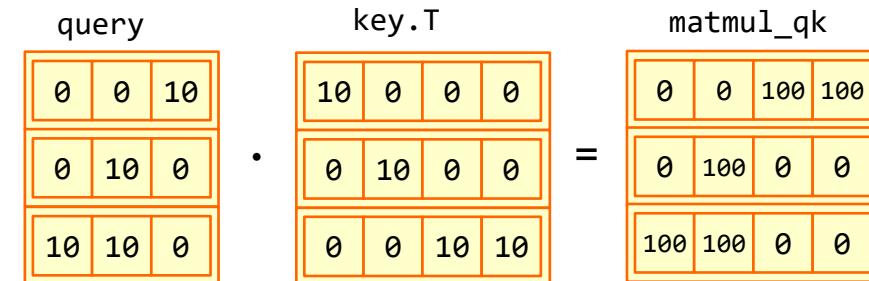


) =



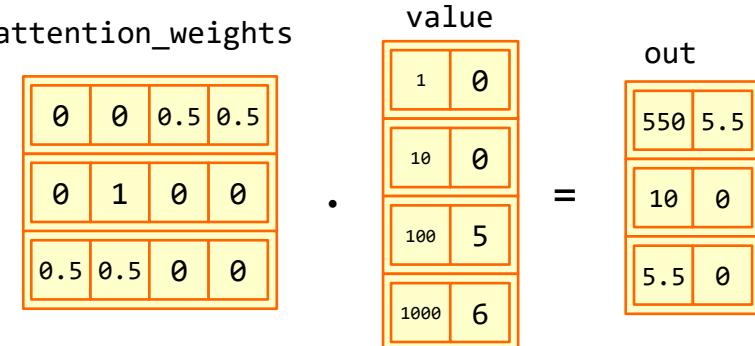
(3,4)(4,2)

(3,2)



attention_weights

0	0	0.5	0.5
0	1	0	0
0.5	0.5	0	0



Query는 4개의 Key값 중 두번째 값과 일치하므로 어텐션 분포는 [0, 1, 0, 0]의 값을 가지며 결과적으로 Value의 두번째 값인 [10, 0]이 출력되는 것을 확인할 수 있습니다. 이번에는 Query의 값만 다른 값으로 바꿔보고 함수를 실행해봅시다. 이번에 사용할 Query값 [0, 0, 10]은 Key의 세번째 값과, 네번째 값 두 개의 값 모두와 일치하는 값입니다.

```
temp_q = tf.constant([[0, 0, 10]], dtype=tf.float32)
temp_out, temp_attn = scaled_dot_product_attention(temp_q, temp_k, temp_v, None)
print(temp_attn)
print(temp_out)
```

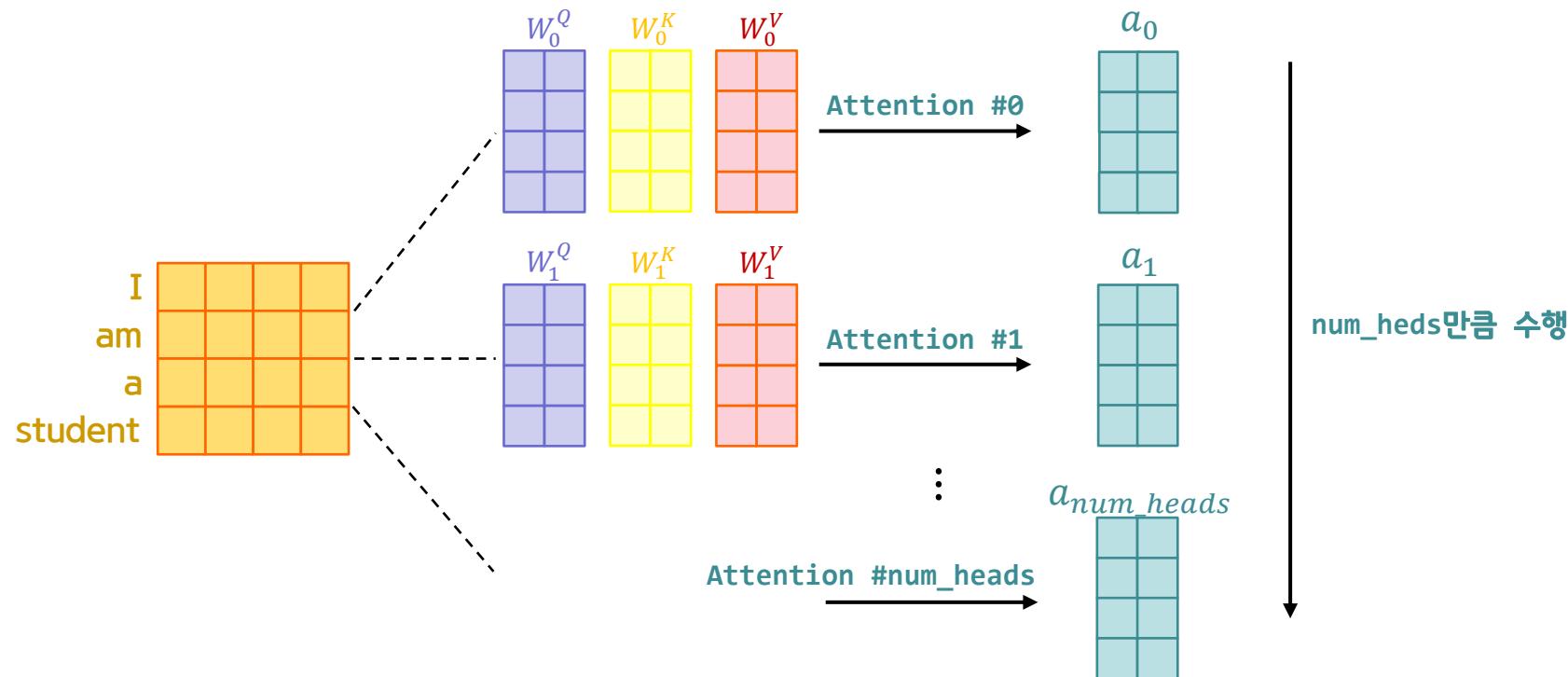
Query의 값은 Key의 세번째 값과 네번째 값 두 개의 값과 모두 유사하다는 의미에서 어텐션 분포는 [0, 0, 0.5, 0.5]의 값을 가집니다. 결과적으로 나오는 값 [550, 5.5]는 Value의 세번째 값 [100, 5]에 0.5를 곱한 값과 네번째 값 [1000, 6]에 0.5를 곱한 값의 원소별 합입니다. 이번에는 하나가 아닌 3개의 Query의 값을 함수의 입력으로 사용해 보겠습니다.

```
temp_q = tf.constant([[0, 0, 10], [0, 10, 0], [10, 10, 0]], dtype=tf.float32) # (3, 3)
temp_out, temp_attn = scaled_dot_product_attention(temp_q, temp_k, temp_v, None)
print(temp_attn)
print(temp_out)
```

멀티 헤드 어텐션(Multi-head Attention)

9.1 트랜스 포머(Transformer)

앞서 배운 어텐션에서는 d_{model} 의 차원을 가진 단어 벡터를 num_heads 로 나눈 차원을 가지는 Q, K, V벡터로 바꾸고 어텐션을 수행하였습니다. 논문 기준으로는 512의 차원의 각 단어 벡터를 8로 나누어 64차원의 Q, K, V 벡터로 바꾸어서 어텐션을 수행한 셈인데, 이제 num_heads 의 의미와 왜 d_{model} 의 차원을 가진 단어 벡터를 가지고 어텐션을 하지 않고 차원을 축소시킨 벡터로 어텐션을 수행하였는지 이해해보겠습니다.



멀티 헤드 어텐션(Multi-head Attention)

9.1 트랜스 포머(Transformer)

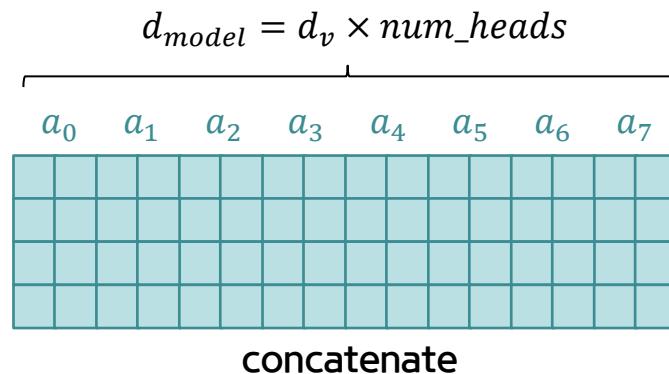
트랜스포머 연구진은 한 번의 어텐션을 하는 것보다 여러번의 어텐션을 병렬로 사용하는 것이 더 효과적이라고 판단하였습니다. 그래서 d_{model} 의 차원을 num_heads 개로 나누어 d_{model}/num_heads 의 차원을 가지는 Q, K, V에 대해서 num_heads 개의 병렬 어텐션을 수행합니다. 논문에서는 하이퍼파라미터인 num_heads 의 값을 8로 지정하였고, 8개의 병렬 어텐션이 이루어지게 됩니다. 다시 말해 위에서 설명한 어텐션이 8개로 병렬로 이루어지게 되는데, 이때 각각의 어텐션 값 행렬을 어텐션 헤드라고 부릅니다. 이때 가중치 행렬 W^Q, W^K, W^V 의 값은 8개의 어텐션 헤드마다 전부 다릅니다.

병렬 어텐션으로 얻을 수 있는 효과는 무엇일까요? 그리스도마신화에는 머리가 여러 개인 괴물 히드라나 케로베로스가 나옵니다. 이 괴물들의 특징은 머리가 여러 개이기 때문에 여러 시점에서 상대방을 볼 수 있다는 겁니다. 이렇게 되면 시각에서 놓치는 게 별로 없을테니까 이런 괴물들에게 기습을 하는 것이 굉장히 힘이 들겁니다. 멀티 헤드 어텐션도 똑같습니다. 어텐션을 병렬로 수행하여 다른 시각으로 정보들을 수집하겠다는 겁니다.

예를 들어보겠습니다. 앞서 사용한 예문 '그 동물은 길을 건너지 않았다. 왜냐하면 그것은 너무 피곤하였기 때문이다.'를 상기해봅시다. 단어 그것(it)이 퀘리였다고 해봅시다. 즉, it에 대한 Q벡터로부터 다른 단어와의 연관도를 구하였을 때 첫번째 어텐션 헤드는 '그것(it)'과 '동물(animal)'의 연관도를 높게 본다면, 두번째 어텐션 헤드는 '그것(it)'과 '피곤하였기 때문이다(tired)'의 연관도를 높게 볼 수 있습니다. 각 어텐션 헤드는 전부 다른 시각에서 보고 있기 때문입니다.

멀티 헤드 어텐션(Multi-head Attention)

9.1 트랜스 포머(Transformer)

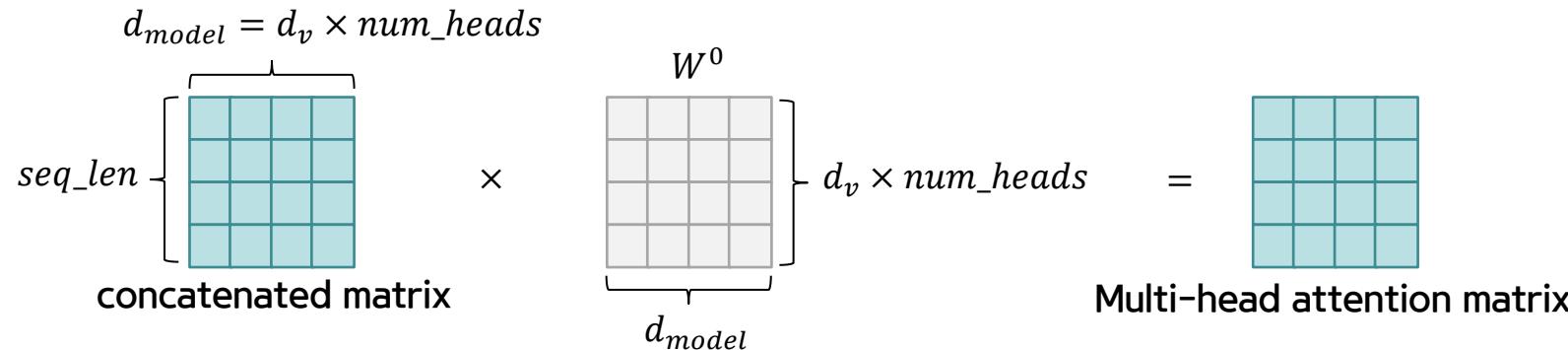


병렬 어텐션을 모두 수행하였다면 모든 어텐션 헤드를 연결(concatenate)합니다. 모두 연결된 어텐션 헤드 행렬의 크기는 (seq_len, d_{model}) 가 됩니다.

지금까지 그림에서는 책의 지면상의 한계로 4차원을 $d_{model} = 512$ 로 표현하고, 2차원을 $d_v = 64$ 로 표현해왔기 때문에 위의 그림의 행렬의 크기에 혼동의 있을 수 있으나 8개의 어텐션 헤드의 연결(concatenate) 과정의 이해를 위해 이번 행렬만 예외로 위와 같이 d_{model} 의 크기를 d_v 의 8배인 16차원으로 표현하였습니다. 아래의 그림에서는 다시 d_{model} 를 4차원으로 표현합니다.

멀티 헤드 어텐션(Multi-head Attention)

9.1 트랜스 포머(Transformer)



어텐션 헤드를 모두 연결한 행렬은 또 다른 가중치 행렬 W^0 을 곱하게 되는데, 이렇게 나온 결과 행렬이 멀티-헤드 어텐션의 최종 결과물입니다. 위의 그림은 어텐션 헤드를 모두 연결한 행렬이 가중치 행렬 W^0 과 곱해지는 과정을 보여줍니다. 이때 결과물인 멀티-헤드 어텐션 행렬은 인코더의 입력이었던 문장 행렬의 (seq_len, d_{model}) 크기와 동일합니다.

다시 말해 인코더의 첫번째 서브층인 멀티-헤드 어텐션 단계를 끝마쳤을 때, 인코더의 입력으로 들어왔던 행렬의 크기가 아직 유지되고 있음을 기억해둡시다. 첫번째 서브층인 멀티-헤드 어텐션과 두번째 서브층인 포지션 와이즈 피드 포워드 신경망을 지나면서 인코더의 입력으로 들어올 때의 행렬의 크기는 계속 유지되어야 합니다. 트랜스포머는 다수의 인코더를 쌓은 형태인데(논문에서는 인코더가 6개), 인코더에서의 입력의 크기가 출력에서도 동일 크기로 계속 유지되어야만 다음 인코더에서도 다시 입력이 될 수 있기 때문입니다.

멀티 헤드 어텐션에서는 크게 두 종류의 가중치 행렬이 나왔습니다. 바로 Q, K, V 행렬을 만들기 위한 가중치 행렬인 WQ, WK, WV 행렬과 바로 어텐션 헤드들을 연결(concatenation) 후에 곱해주는 WO 행렬입니다. 가중치 행렬을 곱하는 것을 구현 상에서는 입력을 밀집층(Dense layer)를 지나게 하므로서 구현합니다. 클래스 코드 상으로 지금까지 줄기차게 사용해왔던 Dense()에 해당됩니다.

멀티 헤드 어텐션의 구현은 크게 다섯 가지 파트로 구성됩니다.

1. WQ, WK, WV에 해당하는 d_{model} 크기의 밀집층(Dense layer)을 지나게 한다.
2. 지정된 헤드 수(num_heads)만큼 나눈다(split).
3. 스케일드 닷 프로덕트 어텐션.
4. 나눠졌던 헤드들을 연결(concatenation)한다.
5. WO에 해당하는 밀집층을 지나게 한다.

※ 코드참조

```

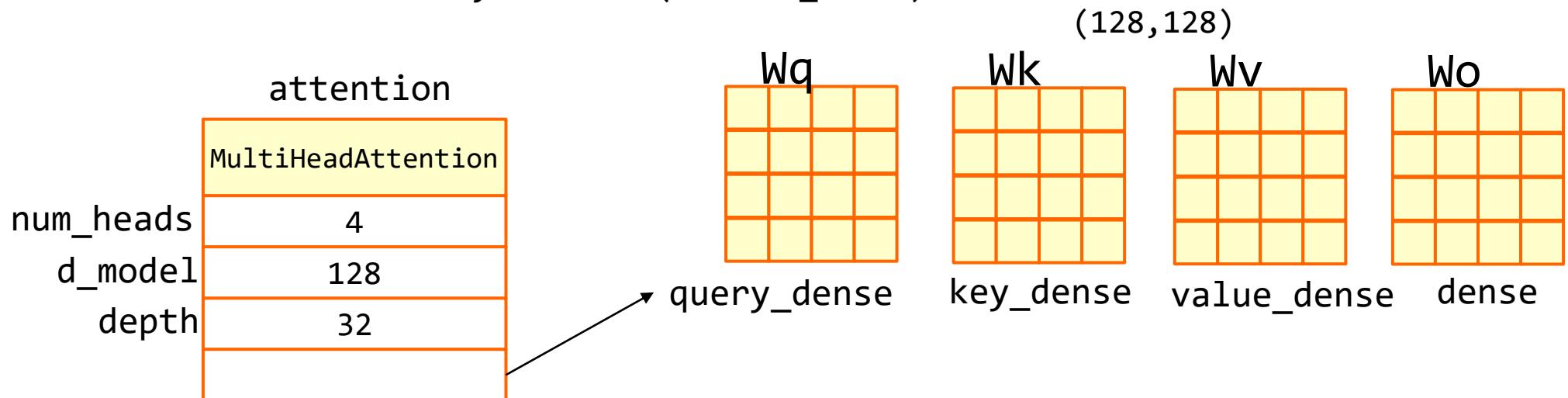
def __init__(self, d_model=128, num_heads=4, name="multi_head_attention"):
    self.num_heads = num_heads
    self.d_model = d_model

    self.depth = d_model // self.num_heads

    self.query_dense = tf.keras.layers.Dense(units=d_model) # XW+b
    self.key_dense = tf.keras.layers.Dense(units=d_model)
    self.value_dense = tf.keras.layers.Dense(units=d_model)

    self.dense = tf.keras.layers.Dense(units=d_model)

```



```

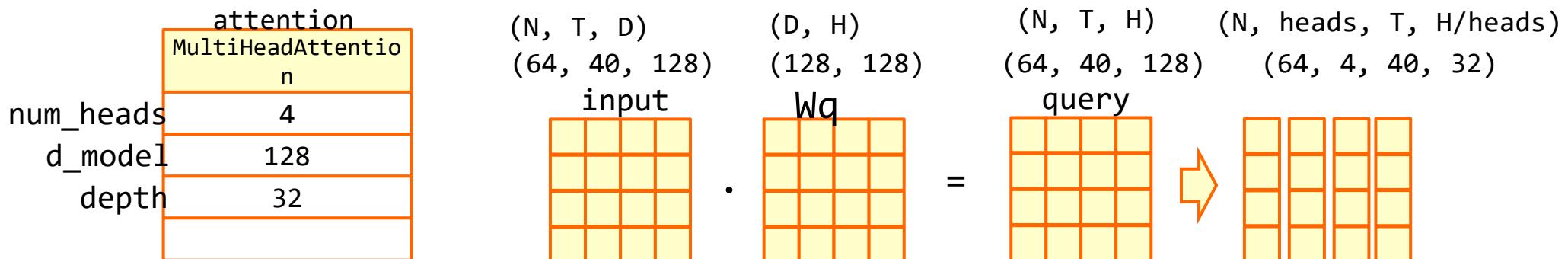
def call(self, inputs):
    query, key, value, mask = inputs['query'], inputs['key'], inputs[
        'value'], inputs['mask']
    batch_size = tf.shape(query)[0]

# q : (batch_size, query의 문장 길이, d_model) (64, 40, 128)
    query = self.query_dense(query)
    key = self.key_dense(key)
    value = self.value_dense(value)

# q : (batch_size, num_heads, query의 문장 길이, d_model/num_heads)
    query = self.split_heads(query, batch_size)
    key = self.split_heads(key, batch_size)
    value = self.split_heads(value, batch_size)

```

(64, 40, 4, 32)
(64, 4, 40, 32)

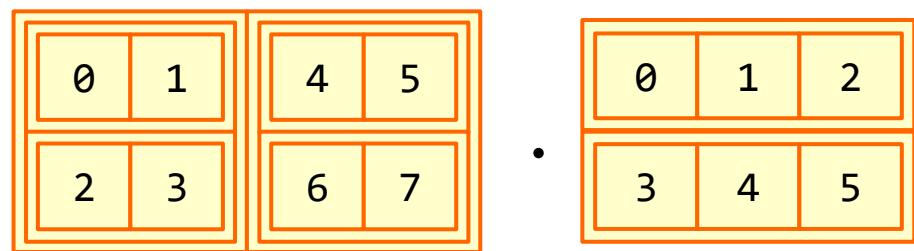


```

x = np.arange(8).reshape(2,2,2)
print(x)
w = np.arange(6).reshape(2,3)
print(w)
out = np.dot(x,w)
print(out.shape)
print(out)

```

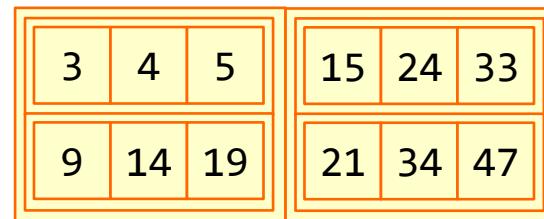
$$(2,2,2)(2,3) \Rightarrow (2,2,3)$$



```

[[[ 3  4  5]
 [ 9 14 19]]
 [[15 24 33]
 [21 34 47]]]

```



$(2,2,3)$

```
import tensorflow as tf
```

```
query = tf.constant(np.arange(0,24).reshape(1,2,3,4),dtype=tf.float32)
key = tf.constant(np.ones((1,2,3,4)),dtype=tf.float32)
out = tf.matmul(q, k, transpose_b=True) # (1,2,3,4) (1,2,4,3)
print(out.shape) # (1,2,3,3)
```

(1,2,3,4)

(1,2,4,3)

(3,4)(4,3)

q

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

0	1	2	3
4	5	6	7
8	9	10	11

1	1	1
1	1	1
1	1	1
1	1	1

6	6	6
22	22	22
38	38	38

12	13	14	15
16	17	18	19
20	21	22	23

1	1	1
1	1	1
1	1	1
1	1	1

54	54	54
70	70	70
86	86	86

```

import tensorflow as tf

q = np.arange(24).reshape(1,2,3,4)
k = np.arange(24).reshape(1,2,3,4)
q = tf.constant(q)
print(q)
k = tf.constant(k)
print(k)

out = tf.matmul(q, k, transpose_b=True) # (1,2,3,4) (1,2,4,3) (3,4)(4,3)
print(out.shape) # (1,2,3,3)

```

q	<table border="1"> <tr><td>0</td><td>1</td><td>2</td><td>3</td></tr> <tr><td>4</td><td>5</td><td>6</td><td>7</td></tr> <tr><td>8</td><td>9</td><td>10</td><td>11</td></tr> <tr><td>12</td><td>13</td><td>14</td><td>15</td></tr> <tr><td>16</td><td>17</td><td>18</td><td>19</td></tr> <tr><td>20</td><td>21</td><td>22</td><td>23</td></tr> </table>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	1	2	3																						
4	5	6	7																						
8	9	10	11																						
12	13	14	15																						
16	17	18	19																						
20	21	22	23																						

k	<table border="1"> <tr><td>0</td><td>4</td><td>8</td></tr> <tr><td>1</td><td>5</td><td>9</td></tr> <tr><td>2</td><td>6</td><td>10</td></tr> <tr><td>3</td><td>7</td><td>11</td></tr> <tr><td>12</td><td>16</td><td>20</td></tr> <tr><td>13</td><td>17</td><td>21</td></tr> <tr><td>14</td><td>18</td><td>22</td></tr> <tr><td>15</td><td>19</td><td>23</td></tr> </table>	0	4	8	1	5	9	2	6	10	3	7	11	12	16	20	13	17	21	14	18	22	15	19	23
0	4	8																							
1	5	9																							
2	6	10																							
3	7	11																							
12	16	20																							
13	17	21																							
14	18	22																							
15	19	23																							

$$\begin{array}{|c|c|c|c|} \hline 0 & 1 & 2 & 3 \\ \hline 4 & 5 & 6 & 7 \\ \hline 8 & 9 & 10 & 11 \\ \hline \end{array} \cdot \begin{array}{|c|c|c|} \hline 0 & 4 & 8 \\ \hline 1 & 5 & 9 \\ \hline 2 & 6 & 10 \\ \hline 3 & 7 & 11 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 14 & 38 & 62 \\ \hline 38 & 126 & 214 \\ \hline 62 & 214 & 366 \\ \hline \end{array}$$

$$\begin{array}{|c|c|c|c|} \hline 12 & 13 & 14 & 15 \\ \hline 16 & 17 & 18 & 19 \\ \hline 20 & 21 & 22 & 23 \\ \hline \end{array} \cdot \begin{array}{|c|c|c|} \hline 12 & 16 & 20 \\ \hline 13 & 17 & 21 \\ \hline 14 & 18 & 22 \\ \hline 15 & 19 & 23 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 734 & 950 & 1166 \\ \hline 950 & 1230 & 1510 \\ \hline 1166 & 1510 & 1854 \\ \hline \end{array}$$

```

def scaled_dot_product_attention(query, key, value, mask): # query (64, 4, 40, 32)

matmul_qk = tf.matmul(query, key, transpose_b=True) # (64,4,40,32)(64,4,32,40)
# (64,4,40,40)
depth = tf.cast(tf.shape(key)[-1], tf.float32) # depth = float(32)
logits = matmul_qk / tf.math.sqrt(depth) # (64, 4, 40, 40)

if mask is not None:
    logits += (mask * -1e9)

```

0	0	0	0	1	1	1	1	mask
2	3	4	2	0	0	0	0	0

```

# 12345 += -1000000000 마스크가 1인 경우
# 12334 += 0           마스크가 0인 경우

```

```

attention_weights = tf.nn.softmax(logits, axis=-1) # (64, 4, 40, 40)
output = tf.matmul(attention_weights, value) # (64, 4, 40, 40)(64, 4, 40, 32)
return output, attention_weights # (64, 4, 40, 32)

```

query 크기 : (batch_size, num_heads, query의 문장 길이, d_model/num_heads)

query	key	matmul_qk
(64, 4, 40, 32)	(64, 4, 40, 32)	(64, 4, 40, 40)

(40, 32)(32, 40)

```

# (batch_size, query의 문장 길이, num_heads, d_model/num_heads) # (64, 4, 40, 32)
scaled_attention = tf.transpose(scaled_attention, perm=[0, 2, 1, 3])
# (64, 40, 4, 32)

# 4. 헤드 연결(concatenate)하기
# (batch_size, query의 문장 길이, d_model)
concat_attention = tf.reshape(scaled_attention,
                               (batch_size, -1, self.d_model)) # (64, 40, 128)

# 5. W0에 해당하는 밀집층 지나기
# (batch_size, query의 문장 길이, d_model)
outputs = self.dense(concat_attention) # (64, 40, 128)(128,128)=> (64, 40, 128)
return outputs

```

query 크기 : (batch_size, num_heads, query의 문장 길이, d_model/num_heads)

query (64, 4, 40, 32)	key (64, 4, 40, 32)	matmul_qk (64, 4, 40, 40)
(40, 32)(32, 40)		

드롭아웃 + 잔차 연결과 층 정규화

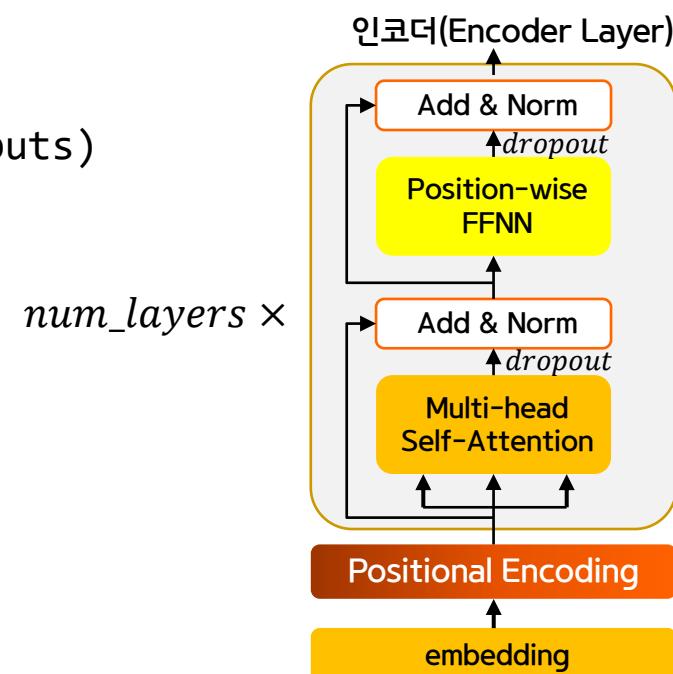
```
attention = tf.keras.layers.Dropout(rate=dropout)(attention) # (64,40,128)
attention = tf.keras.layers.LayerNormalization(
    epsilon=1e-6)(inputs + attention)
```

포지션 와이즈 피드 포워드 신경망 (두번째 서브층)

```
outputs = tf.keras.layers.Dense(units=dff, activation='relu')(attention)
        # (64, 40, 128)(128,512) => (64, 40, 512)
outputs = tf.keras.layers.Dense(units=d_model)(outputs)
        # (64, 40, 512)(512,128) => (64, 40, 128)
```

드롭아웃 + 잔차 연결과 층 정규화

```
outputs = tf.keras.layers.Dropout(rate=dropout)(outputs)
outputs = tf.keras.layers.LayerNormalization(
    epsilon=1e-6)(attention + outputs)
```



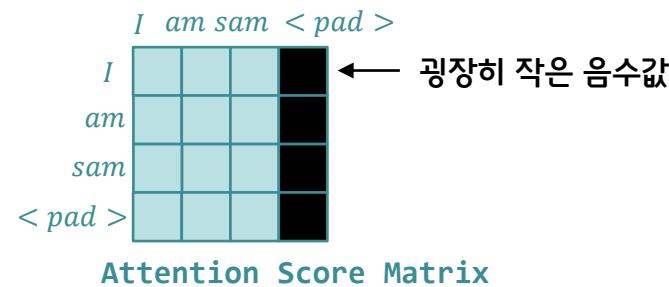
스케일드 닷 프로덕트 어텐션 함수 내부를 보면 mask라는 값을 인자로 받아서, 이 mask값에다가 $-1e9$ 라는 아주 작은 음수값을 곱한 후 어텐션 스코어 행렬에 더해주고 있습니다. 이 연산의 정체는 무엇일까요?

```
def scaled_dot_product_attention(query, key, value, mask):
... 중략 ...
    logits += (mask * -1e9) # 어텐션 스코어 행렬인 logits에 mask*-1e9 값을 더해주고 있다.
... 중략 ...
```

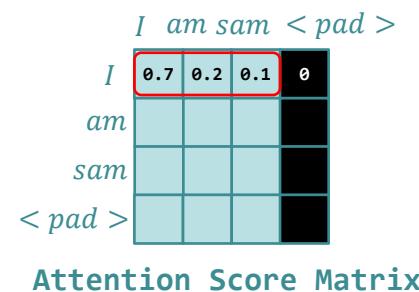
이는 입력 문장에 <PAD> 토큰이 있을 경우 어텐션에서 사실상 제외하기 위한 연산입니다. 예를 들어 <PAD>가 포함된 입력 문장의 셀프 어텐션의 예제를 봅시다. 이에 대해서 어텐션을 수행하고 어텐션 스코어 행렬을 얻는 과정은 다음과 같습니다.

$$\begin{array}{c}
 \text{Input} \\
 \begin{matrix} I \\ am \\ sam \\ <\text{pad}> \end{matrix} \\
 \times \quad K^T \\
 \hline
 \sqrt{d_k}
 \end{array}
 =
 \begin{array}{c}
 \text{Attention Score Matrix} \\
 \begin{matrix} I & am & sam & <\text{pad}> \\ I & & & \\ am & & & \\ sam & & & \\ <\text{pad}> & & & \end{matrix}
 \end{array}$$

그런데 사실 단어 <PAD>의 경우에는 실질적인 의미를 가진 단어가 아닙니다. 그래서 트랜스포머에서는 Key의 경우에 <PAD> 토큰이 존재한다면 이에 대해서는 유사도를 구하지 않도록 마스킹(Masking)을 해주기로 했습니다. 여기서 마스킹이란 어텐션에서 제외하기 위해 값을 가린다는 의미입니다. 어텐션 스코어 행렬에서 행에 해당하는 문장은 Query이고, 열에 해당하는 문장은 Key입니다. 그리고 Key에 <PAD>가 있는 경우에는 해당 열 전체를 마스킹을 해줍니다.



마스킹을 하는 방법은 어텐션 스코어 행렬의 마스킹 위치에 매우 작은 음수값을 넣어주는 것입니다. 여기서 매우 작은 음수값이라는 것은 -1,000,000,000과 같은 -무한대에 가까운 수라는 의미입니다. 현재 어텐션 스코어 함수는 소프트맥스 함수를 지나지 않은 상태입니다. 앞서 배운 연산 순서라면 어텐션 스코어 함수는 소프트맥스 함수를 지나고, 그 후 Value 행렬과 곱해지게 됩니다. 그런데 현재 마스킹 위치에 매우 작은 음수 값이 들어가 있으므로 어텐션 스코어 행렬이 소프트맥스 함수를 지난 후에는 해당 위치의 값은 0에 굉장히 가까운 값이 되어 단어 간 유사도를 구하는 일에 <PAD> 토큰이 반영되지 않게 됩니다.



위 그림은 소프트맥스 함수를 지난 후를 가정하고 있습니다. 소프트맥스 함수를 지나면 각 행의 어텐션 가중치의 총 합은 1이 되는데, 단어 <PAD>의 경우에는 0이 되어 어떤 유의미한 값을 가지고 있지 않습니다.

패딩 마스크를 구현하는 방법은 입력된 정수 시퀀스에서 패딩 토큰의 인덱스인지, 아닌지를 판별하는 함수를 구현하는 것입니다. 아래의 함수는 정수 시퀀스에서 0인 경우에는 1로 변환하고, 그렇지 않은 경우에는 0으로 변환하는 함수입니다.

```
def create_padding_mask(x):
    mask = tf.cast(tf.math.equal(x, 0), tf.float32)
    # (batch_size, 1, 1, key의 문장 길이)
    return mask[:, :, tf.newaxis, tf.newaxis, :]
```

임의의 정수 시퀀스 입력을 넣어서 어떻게 변환되는지 보겠습니다.

```
print(create_padding_mask(tf.constant([[1, 21, 777, 0, 0]])))
```

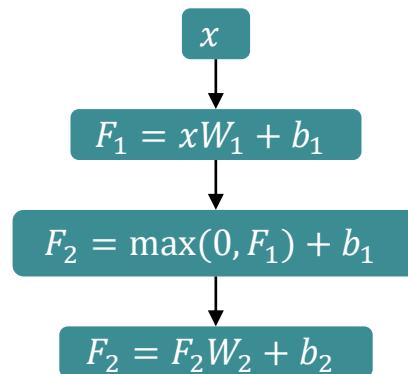
위 벡터를 통해서 1의 값을 가진 위치의 열을 어텐션 스코어 행렬에서 마스킹하는 용도로 사용할 수 있습니다. 위 벡터를 스케일드 닷 프로덕트 어텐션의 인자로 전달하면, 스케일드 닷 프로덕트 어텐션에서는 위 벡터에다가 매우 작은 음수값인 $-1e9$ 를 곱하고, 이를 행렬에 더해주어 해당 열을 전부 마스킹하게 되는 것입니다.

첫번째 서브층인 멀티 헤드 어텐션을 구현해보았습니다. 앞서 인코더는 두 개의 서브 서브층(sublayer)으로 나뉘어 진다고 언급한 적이 있는데, 이제 두번째 서브층인 포지션-와이즈 피드 포워드 신경망에 대해서 알아보겠습니다.

지금은 인코더를 설명하고 있지만, 포지션 와이즈 FFNN은 인코더와 디코더에서 공통적으로 가지고 있는 서브층입니다. 포지션-와이즈 FFNN는 쉽게 말하면 완전 연결 FFNN(Fully-connected FFNN)이라고 해석할 수 있습니다. 앞서 인공 신경망은 결국 벡터와 행렬 연산으로 표현될 수 있음을 배웠습니다. 아래는 포지션 와이즈 FFNN의 수식을 보여줍니다.

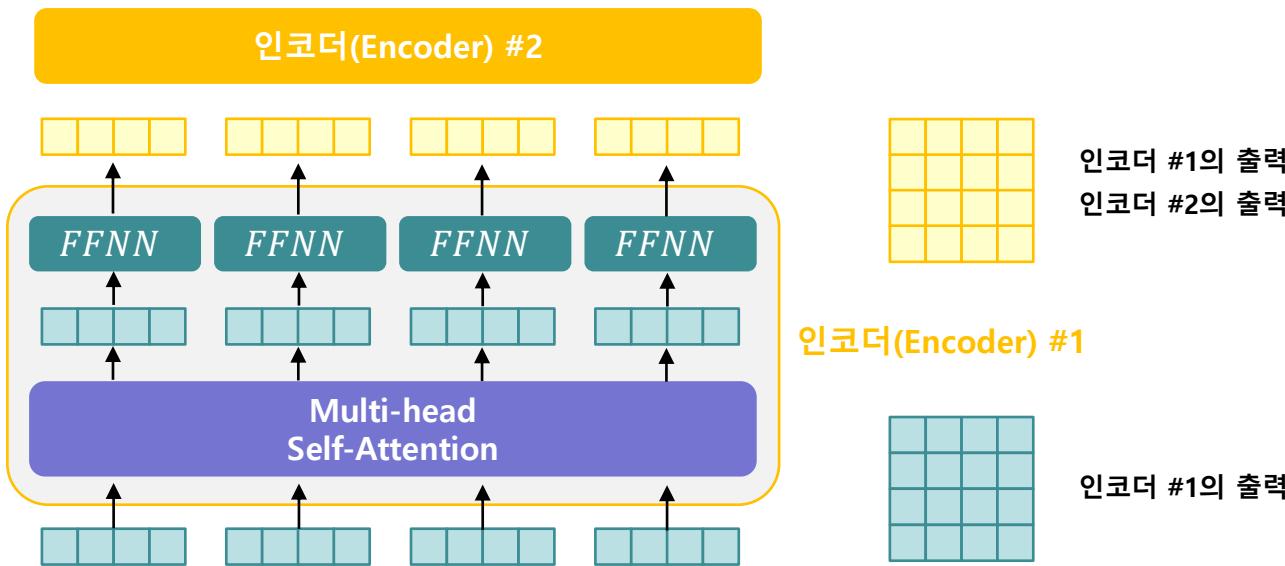
$$FFNN(x) = \text{MAX}(0, xW_1 + b_1)W_2 + b_2$$

식을 그림으로 표현하면 아래와 같습니다.

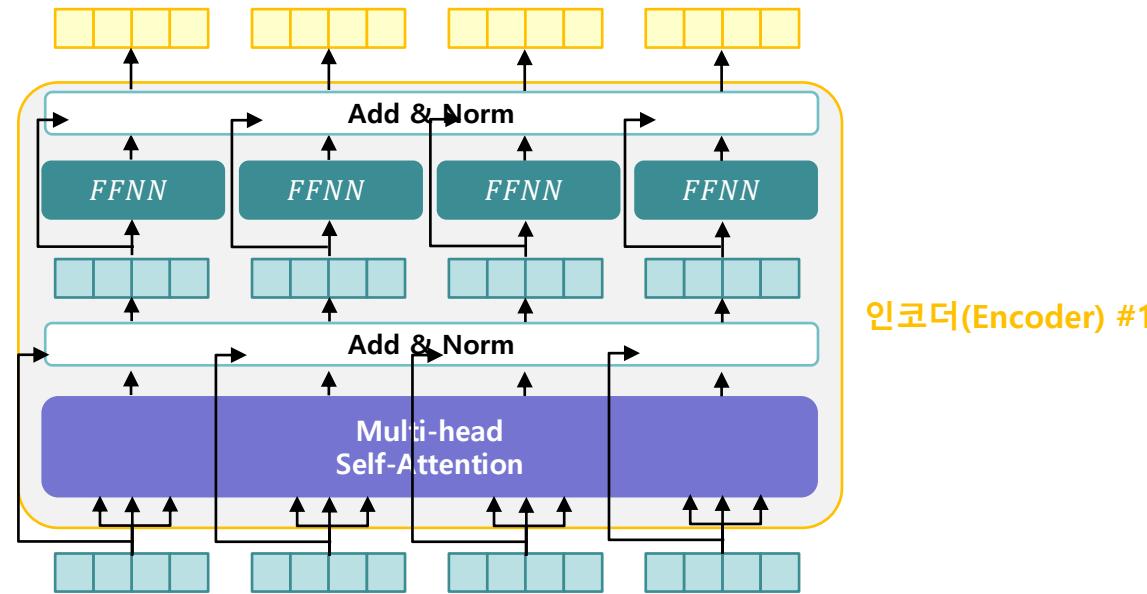


여기서 x 는 앞서 멀티 헤드 어텐션의 결과로 나온 (seq_len, d_{model})의 크기를 가지는 행렬을 말합니다. 가중치 행렬 W_1 은 (d_{model}, d_{ff}) 의 크기를 가지고, 가중치 행렬 W_2 은 (d_{ff}, d_{model}) 의 크기를 가집니다. 논문에서 은닉층의 크기인 d_{ff} 는 앞서 하이퍼파라미터를 정의할 때 언급했듯이 2,048의 크기를 가집니다.

여기서 매개변수 W_1, b_1, W_2, b_2 는 하나의 인코더 층 내에서는 다른 문장, 다른 단어들마다 정확하게 동일하게 사용됩니다. 하지만 인코더 층마다 다른 값을 가집니다.



위의 그림에서 좌측은 인코더의 입력을 벡터 단위로 봤을 때, 각 벡터들이 멀티 헤드 어텐션 층이라는 인코더 내 첫 번째 서브 층을 지나 FFNN을 통과하는 것을 보여줍니다. 이는 두번째 서브층인 Position-wise FFNN을 의미합니다. 물론, 실제로는 그림의 우측과 같이 행렬로 연산되는데, 두번째 서브층을 지난 인코더의 최종 출력은 여전히 인코더의 입력의 크기였던 (seq_len, d_{model})의 크기가 보존되고 있습니다. 하나의 인코더 층을 지난 이 행렬은 다음 인코더 층으로 전달되고, 다음 층에서도 동일한 인코더 연산이 반복됩니다.

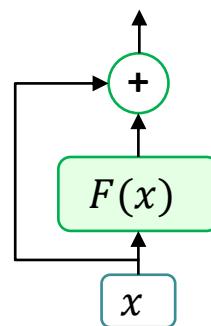


인코더의 두 개의 서브층에 대해서 이해하였다면 인코더에 대한 설명은 거의 다왔습니다! 트랜스포머에서는 이러한 두 개의 서브층을 가진 인코더에 추가적으로 사용하는 기법이 있는데, 바로 Add & Norm입니다. 더 정확히는 잔차 연결(residual connection)과 층 정규화(layer normalization)를 의미합니다.

위의 그림은 앞서 Position-wise FFNN를 설명할 때 사용한 앞선 그림에서 화살표와 Add & Norm(잔차 연결과 정규화 과정)을 추가한 그림입니다. 추가된 화살표들은 서브층 이전의 입력에서 시작되어 서브층의 출력 부분을 향하고 있는 것에 주목합시다. 추가된 화살표가 어떤 의미를 갖고 있는지는 잔차 연결과 층 정규화를 배우고 나면 이해할 수 있습니다.

잔차 연결(residual connection)의 의미를 이해하기 위해서 어떤 함수에 대한 이야기를 해보겠습니다.

$$H(x) = x + F(x)$$

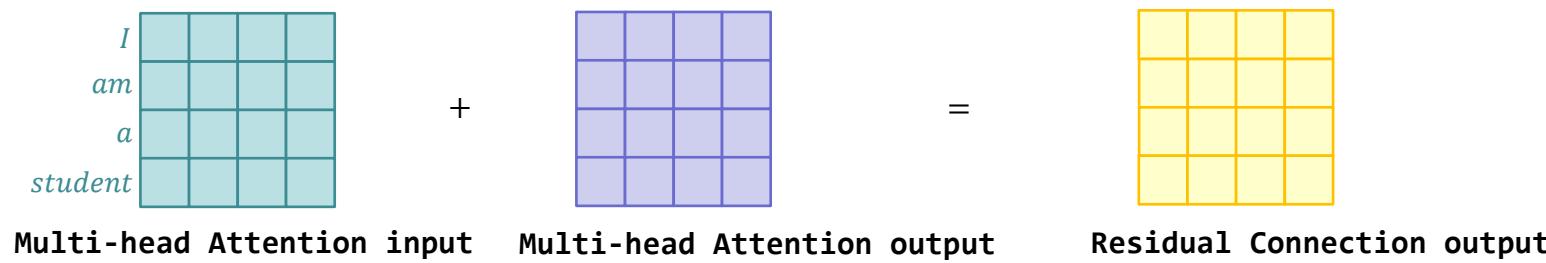


위 그림은 입력 x 와 x 에 대한 어떤 함수 $F(x)$ 의 값을 더한 함수 $H(x)$ 의 구조를 보여줍니다. 어떤 함수 $F(x)$ 가 트랜스포머에서는 서브층에 해당됩니다. 다시 말해 잔차 연결은 서브층의 입력과 출력을 더하는 것을 말합니다. 앞서 언급했듯이 트랜스포머에서 서브층의 입력과 출력은 동일한 차원을 갖고 있으므로, 서브층의 입력과 서브층의 출력은 덧셈 연산을 할 수 있습니다. 이것이 바로 위의 인코더 그림에서 각 화살표가 서브층의 입력에서 출력으로 향하도록 그려졌던 이유입니다. 잔차 연결은 컴퓨터 비전 분야에서 주로 사용되는 모델의 학습을 돋는 기법입니다.

이를 식으로 표현하면 $x + Sublayer(x)$ 라고 할 수 있습니다.

가령, 서브층이 멀티 헤드 어텐션이었다면 잔차 연결 연산은 다음과 같습니다.

$$H(x) = x + Multi - head\ Attention(x)$$

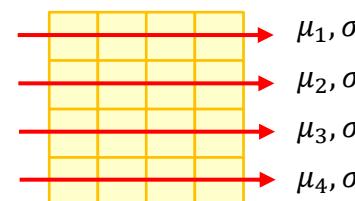


위 그림은 멀티 헤드 어텐션의 입력과 멀티 헤드 어텐션의 결과가 더해지는 과정을 보여줍니다.

잔차 연결을 거친 결과는 이어서 층 정규화 과정을 거치게됩니다. 잔차 연결의 입력을 x , 잔차 연결과 층 정규화 두 가지 연산을 모두 수행한 후의 결과 행렬을 LN 이라고 하였을 때, 잔차 연결 후 층 정규화 연산을 수식으로 표현하자면 다음과 같습니다.

$$LN = \text{LayerNorm}(x + \text{Sublayer}(x))$$

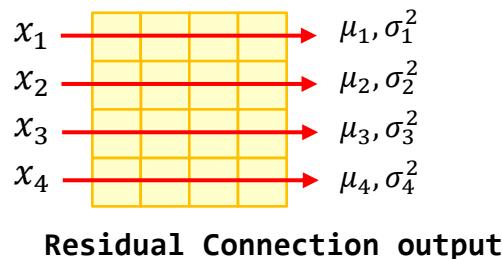
이제 층 정규화를 하는 과정에 대해서 이해해봅시다. 층 정규화는 텐서의 마지막 차원에 대해서 평균과 분산을 구하고, 이를 가지고 어떤 수식을 통해 값을 정규화하여 학습을 돋습니다. 여기서 텐서의 마지막 d_{model} 차원이란 것은 트랜스포머에서는 d_{model} 차원을 의미합니다. 아래 그림은 차원의 방향을 화살표로 표현하였습니다.



Residual Connection output

(seq_len, d_{model})

층 정규화를 위해서 우선, 화살표 방향으로 각각 평균 μ 과 분산 σ^2 을 구합니다. 각 화살표 방향의 벡터를 x_i 라고 명명해봅시다.



층 정규화를 수행한 후에는 벡터 x_i 는 ln_i 라는 벡터로 정규화가 됩니다.

$$ln_i = \text{LayerNorm}(x_i)$$

이제 층 정규화의 수식을 알아봅시다. 여기서는 층 정규화를 두 가지 과정으로 나누어서 설명하겠습니다. 첫번째는 평균과 분산을 통한 정규화, 두번째는 감마와 베타를 도입하는 것입니다. 우선, 평균과 분산을 통해 벡터 x_i 를 정규화 해줍니다. x_i 는 벡터인 반면, 평균 μ_i 과 분산 σ_i^2 은 스칼라입니다. 벡터 x_i 의 각 차원을 k 라고 하였을 때, $x_{i,k}$ 는 다음의 수식과 같이 정규화 할 수 있습니다. 다시 말해 벡터 x_i 의 각 k 차원의 값이 다음과 같이 정규화 되는 것입니다.

$$\hat{x}_{i,k} = \frac{x_{i,k} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

ϵ (입실론)은 분모가 0이 되는 것을 방지하는 값입니다.

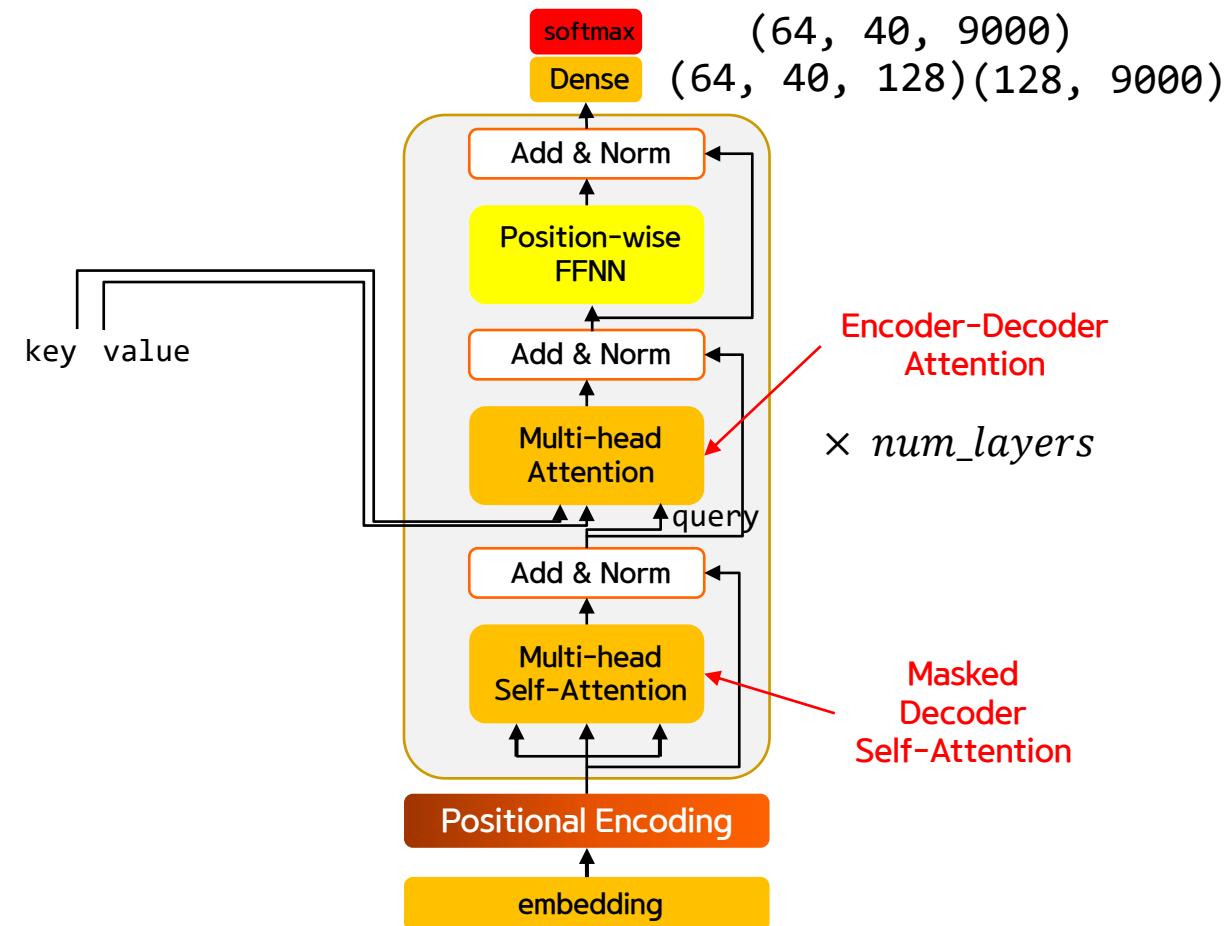
이제 γ (감마)와 β (베타)라는 벡터를 준비합니다. 단, 이들의 초기값은 각각 1과 0입니다.

$$\begin{array}{l} \gamma \quad \boxed{1 \quad 1 \quad 1 \quad 1} \\ \beta \quad \boxed{0 \quad 0 \quad 0 \quad 0} \end{array}$$

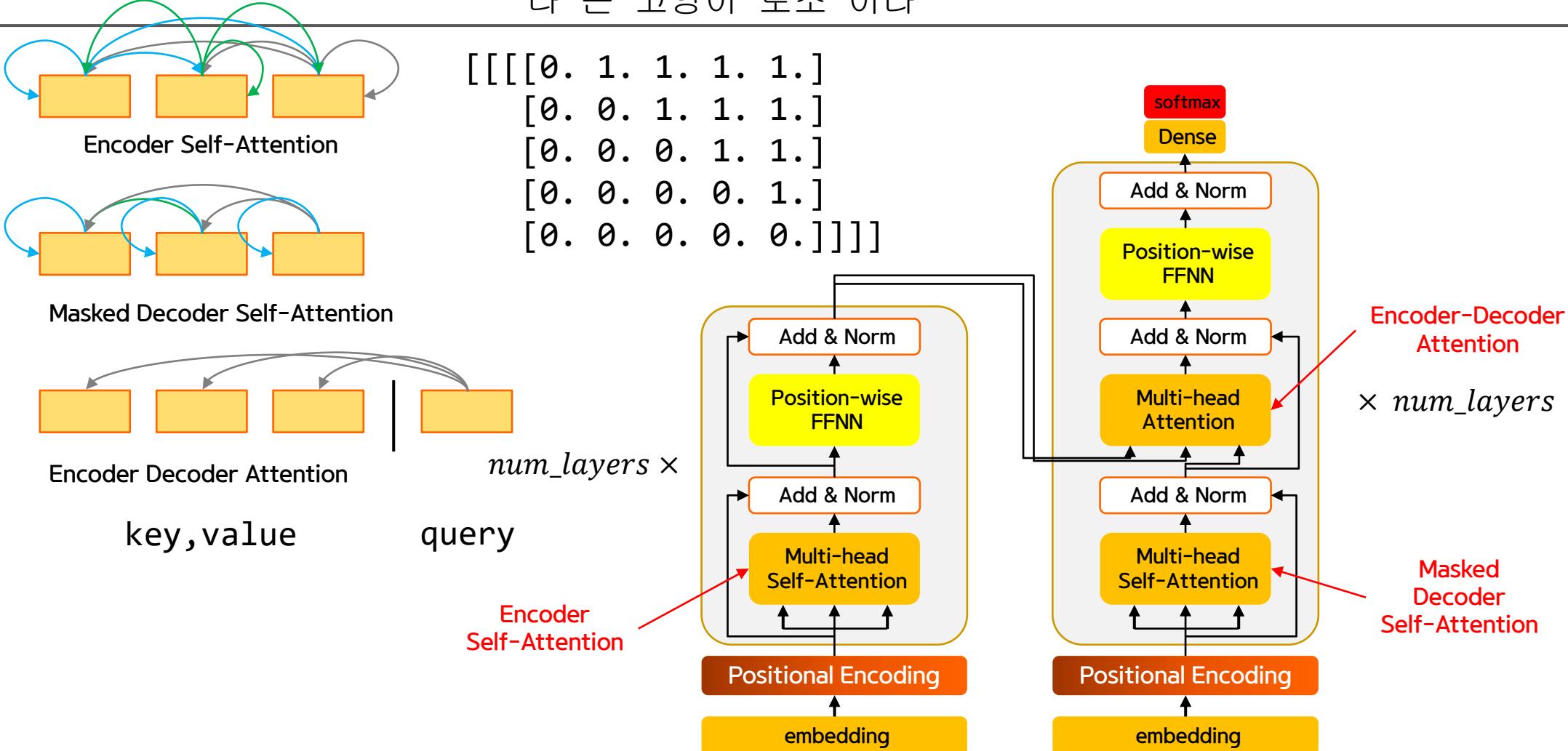
γ 와 β 를 도입한 층 정규화의 최종 수식은 다음과 같으며, γ 와 β 는 학습 가능한 파라미터입니다.

$$ln_i = \gamma \hat{x}_i + \beta = LayerNorm(x_i)$$

※ 소스 참조



나는 고양이 루소이다



10. BERT

10.1 BERT(Bi-directional Encoder Representations from Transformers)

Bi-directional Encoder Representations from Transformers

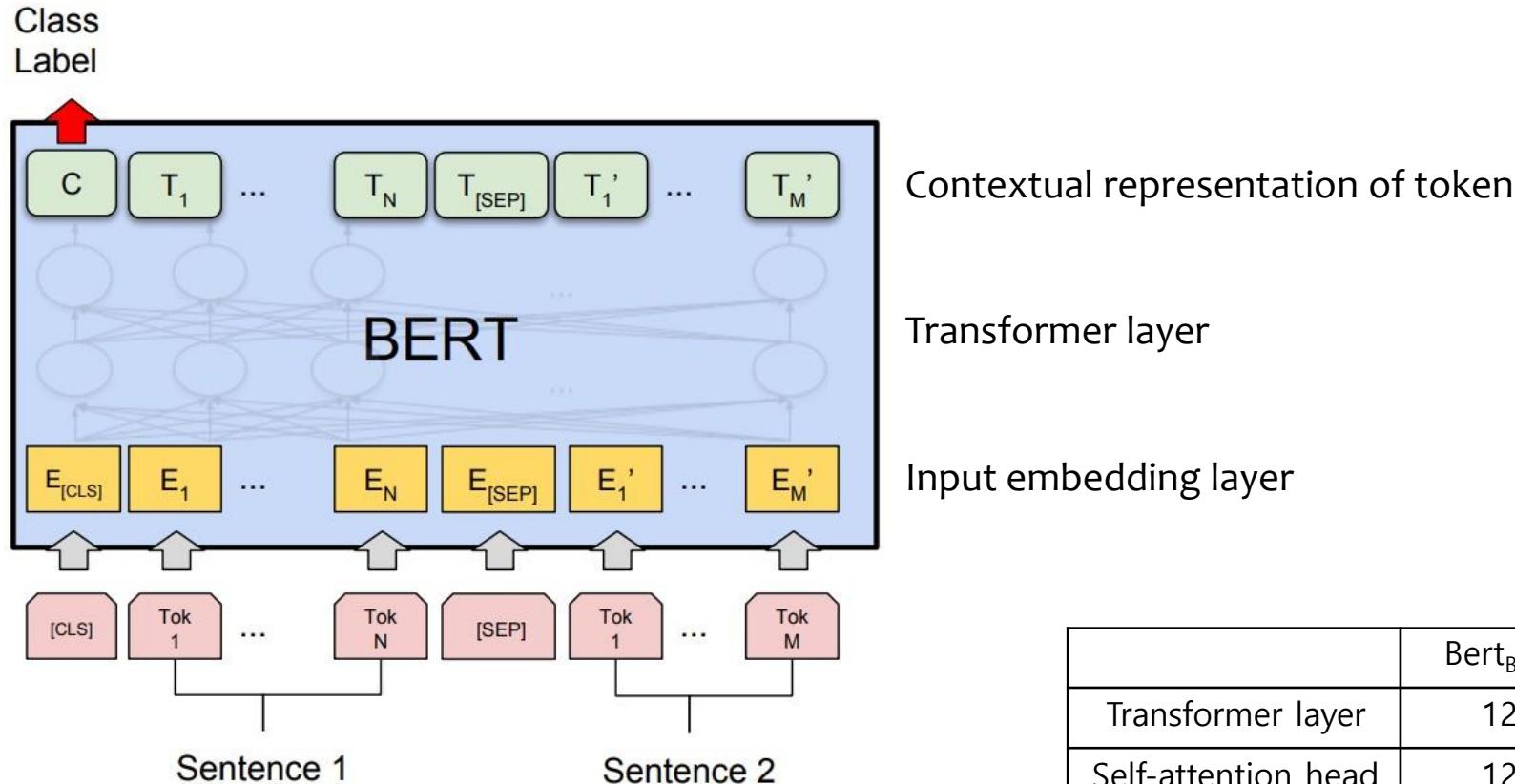
- BERT는 bi-directional Transformer로 이루어진 언어모델
- 잘 만들어진 BERT 언어모델 위에 1개의 classification layer만 부착하여 다양한 NLP task를 수행
- 영어권에서 11개의 NLP task에 대해 state-of-the-art (SOTA) 달성

1 classification layer for Fine-tuning

Pre-trained BERT

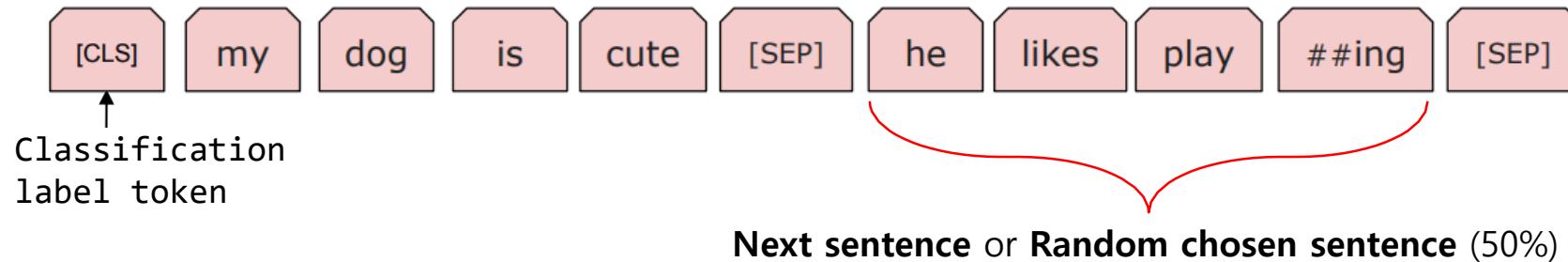
SQuAD v1.1 dataset leaderboard

Rank	Model	EM	F1
	Human Performance Stanford University (Rajpurkar et al. '16)	82.304	91.221
1	BERT (ensemble) Google AI Language https://arxiv.org/abs/1810.04805	87.433	93.160
2	BERT (single model) Google AI Language https://arxiv.org/abs/1810.04805	85.083	91.835



	Bert _{BASE}	Bert _{LARGE}
Transformer layer	12	24
Self-attention head	12	16
Total	110M	340M

- 학습 코퍼스 데이터
 - BooksCorpus (800M words)
 - English Wikipedia (2,500M words without lists, tables and headers)
 - 30,000 token vocabulary
- 데이터의 tokenizing
 - WordPiece tokenizing
He likes playing → He likes play ##ing
 - 입력 문장을 tokenizing하고, 그 token들로 'token sequence'를 만들어 학습에 사용
 - 2개의 token sequence가 학습에 사용



Methods – BERT의 WordPiece tokenizing

10.1 버트(BERT)

- Byte Pair Encoding (BPE) 알고리즘 이용
- 빈도수에 기반해 단어를 의미 있는 패턴(Subword)으로 잘라서 tokenizing

W2V vocabs

고양경찰서
고양시
종로경찰서
경찰
경찰서
경찰관
경찰청



고양	##경찰	##서 ##시
종로	##경찰	##서 경찰
경찰	##서	
경찰	##관	
경찰	##청	



BPE vocabs

고양
##경찰
##서
##시
경찰
##관
##청

- Tokenize

경찰청 철창살은 외철창살이고 검찰청 철창살은 쌍철창살이다

Sentence

Tokenized Sentence (iter=0)

경찰청	→	경 찰 청	→	경 ##찰 ##청
철창살은	→	철 창 살 은	→	철 ##창 ##살 ##은
외철창살이고	→	외 철 창 살 이 고	→	외 ##철 ##창 ##살 ##이 ##고
검찰청	→	검 찰 청	→	검 ##찰 ##청
철창살은	→	철 창 살 은	→	철 ##창 ##살 ##은
쌍철창살이다	→	쌍 철 창 살 이 다	→	쌍 ##철 ##창 ##살 ##이 ##다

- 같은 글자라고 맨 앞에 나오는 것과 아닌 것에는 차이가 있다고 가정
- BERT의 경우 뒷단어에 '##'을 붙여서 구별 e.g.) '철' ≠ '##철' , '철창살' ≠ '##철창살'

- Building Vocab

Tokenized Sentence (iter=0)

경 ##찰 ##청
철 ##창 ##살 ##은
외 ##철 ##창 ##살 ##이 ##고
검 ##찰 ##청
철 ##창 ##살 ##은
쌍 ##철 ##창 ##살 ##이 ##다

Vocab (iter=0) Vocab 후보!

경 ##찰 ##청
철 ##창 ##살 ##은
외 ##철 ##이 ##고
검
쌍 ##다

- Vocab의 생성은 정해진 Iteration을 모두 수행 후 Tokenized Sentence를 Tokenize해 서 생성

Methods – BERT의 WordPiece tokenizing

10.1 버트(BERT)

- Bi-gram Pair Count

Vocab 후보 기준으로

Tokenized Sentence (iter=0)

경 ##찰 ##청

철 ##창 ##살 ##은

외 ##철 ##창 ##살 ##이 ##고

검 ##찰 ##청

철 ##창 ##살 ##은

쌍 ##철 ##창 ##살 ##이 ##다

(경, ##찰)

:1

(##살, ##은)

:2

(##이, ##고)

:1

(##찰, ##청)

:2

(외 ##철)

:1

(검, ##찰)

:1

(철, ##창)

:2

(##철, ##창)

:2

(쌍 ##철)

:1

(##창, ##살)

:4

(##살, ##이)

:2

(##이, ##다)

:1

Bi-gram pairs (iter=1)

(경, ##찰), (##찰, ##청)

(철, ##창), (##창, ##살), (##살, ##은)

(외 ##철), (##철, ##창), (##창, ##살),
(##살, ##이), (##이, ##고)

(검, ##찰), (##찰, ##청)

(철, ##창), (##창, ##살), (##살, ##은),
(쌍 ##철), (##철, ##창), (##창, ##살),
(##살, ##이), (##이, ##다)

- Merge Best pair

Best Pair: `##창 ##살` → `##창살`

Tokenized Sentence (iter=0)

경 ##찰 ##청
철 ##창 ##살 ##은
외 ##철 ##창 ##살 ##이 ##고
검 ##찰 ##청
철 ##창 ##살 ##은
쌍 ##철 ##창 ##살 ##이 ##다

Tokenized Sentence (iter=1)

→ 경 ##찰 ##청
→ 철 ##창살 ##은
→ 외 ##철 ##창살 ##이 ##고
→ 검 ##찰 ##청
→ 철 ##창살 ##은
→ 쌍 ##철 ##창살 ##이 ##다

Methods – BERT의 WordPiece tokenizing

10.1 버트(BERT)

- Building Vocab

Vocab 후보 업데이트		
Tokenized Sentence (iter=1)	Vocab (iter=1)	
경 ##찰 ##청	경	##찰 ##청
철 ##창살 ##은	철	##창살 ##은
외 ##철 ##창살 ##이 ##고	외	##철 ##이 ##고
검 ##찰 ##청	검	
철 ##창살 ##은		
쌍 ##철 ##창살 ##이 ##다	쌍	##다

- Vocab의 생성은 정해진 Iteration을 모두 수행 후 Tokenized Sentence를 Tokenize 해서 생성

- Bi-gram Pair Count

Vocab 후보 기준으로

Tokenized Sentence (iter=1)	Bi-gram pairs (iter=2)
경 ##찰 ##청	→ (경, ##찰), (##찰, ##청)
철 ##창살 ##은	→ (철, ##창살), (##창살, ##은)
외 ##철 ##창살 ##이 ##고	→ (외 ##철), (##철, ##창살), (##창살, ##이), (##이, ##고)
검 ##찰 ##청	→ (검, ##찰), (##찰, ##청)
철 ##창살 ##은	→ (철, ##창살), (##창살, ##은),
쌍 ##철 ##창살 ##이 ##다	→ (쌍 ##철), (##철, ##창살), (##창살, ##이), (##이, ##다)

(경, ##찰)	:1	(외 ##철)	:1	(검, ##찰)
(##찰, ##청)	:2	(##철, ##창살)	:2	(쌍 ##철)
(철, ##창살)	:2	(##창살, ##이)	:2	(##이, ##다)
(##창살, ##은)	:2	(##이, ##고)	:1	

- Merge Best pair

Best Pair: ##찰 ##청 → **##찰청**

Tokenized Sentence (iter=1)

경 ##찰 ##청
철 ##창살 ##은
외 ##철 ##창살 ##0| ##고
검 ##찰 ##청
철 ##창살 ##은
쌍 ##철 ##창살 ##0| ##다

Tokenized Sentence (iter=2)

→ 경 **##찰청**
→ 철 ##창살 ##은
→ 외 ##철 ##창살 ##0| ##고
→ 검 **##찰청**
→ 철 ##창살 ##은
→ 쌍 ##철 ##창살 ##0| ##다

- Best pair가 여러 개여도 하나만 선택

- Building Vocab

Tokenized Sentence (iter=2)
경 ##찰청
철 ##창살 ##은
외 ##철 ##창살 ##이 ##고
검 ##찰청
철 ##창살 ##은
쌍 ##철 ##창살 ##이 ##다

Vocab 후보 업데이트
Vocab (iter=2)
경 ##찰청
철 ##창살 ##은
외 ##철 ##이 ##고
검
쌍 ##다

→

- Vocab의 생성은 정해진 Iteration을 모두 수행 후 Tokenized Sentence를 Tokenize해서 생성

Methods – BERT의 WordPiece tokenizing

10.1 버트(BERT)

- Bi-gram Pair Count

Vocab 후보 기준으로

Tokenized Sentence (iter=2)

경 ##찰청
철 ##창살 ##은
외 ##철 ##창살 ##이 ##고

검 ##찰청
철 ##창살 ##은
쌍 ##철 ##창살 ##이 ##다

Bi-gram pairs (iter=3)

→ (경, ##찰청)
→ (철, ##창살), (##창살, ##은)
→ (외 ##철), (##철, ##창살), (##창살, ##이), (##이, ##고)
→ (검, ##찰청)
→ (철, ##창살), (##창살, ##은),
→ (쌍 ##철), (##철, ##창살), (##창살, ##이), (##이, ##다)

(경, ##찰청) :1 (##철, ##창살) :2 (쌍 ##철) :1

(철, ##창살) :2 (##창살, ##이) :2 (##이, ##다) :1

(##창살, ##은) :2 (##이, ##고) :1

(외 ##철) :1 (검, ##찰청) :1

- Merge Best pair

Best Pair: 철 ##창살 → **철창살**

Tokenized Sentence (iter=2)

경 ##찰 ##청

철 ##창살 ##은

외 ##철 ##창살 ##이 ##고

검 ##찰 ##청

철 ##창살 ##은

쌍 ##철 ##창살 ##이 ##다

Tokenized Sentence (iter=3)

→ 경 ##찰청

→ **철창살** ##은

→ 외 ##철 ##창살 ##이 ##고

→ 검 ##찰청

→ **철창살** ##은

→ 쌍 ##철 ##창살 ##이 ##다

- 3 글자 이상의 패턴도 Iteration이 진행되면서 나타날 수 있음

Methods – BERT의 WordPiece tokenizing

10.1 버트(BERT)

- Building Vocab

Tokenized Sentence (iter=3)	Vocab (iter=3)
경 ##찰청	경 ##찰청
철창살 ##은	철창살 ##은
외 ##철 ##창살 ##이 ##고	외 ##철 ##창살 ##이 ##고
→	
검 ##찰청	검
철창살 ##은	
쌍 ##철 ##창살 ##이 ##다	쌍 ##다

- Vocab의 생성은 정해진 Iteration을 모두 수행 후 Tokenized Sentence를 Tokenize해서 생성(매 iteration마다 Vocab을 생성하는 것이 아님)

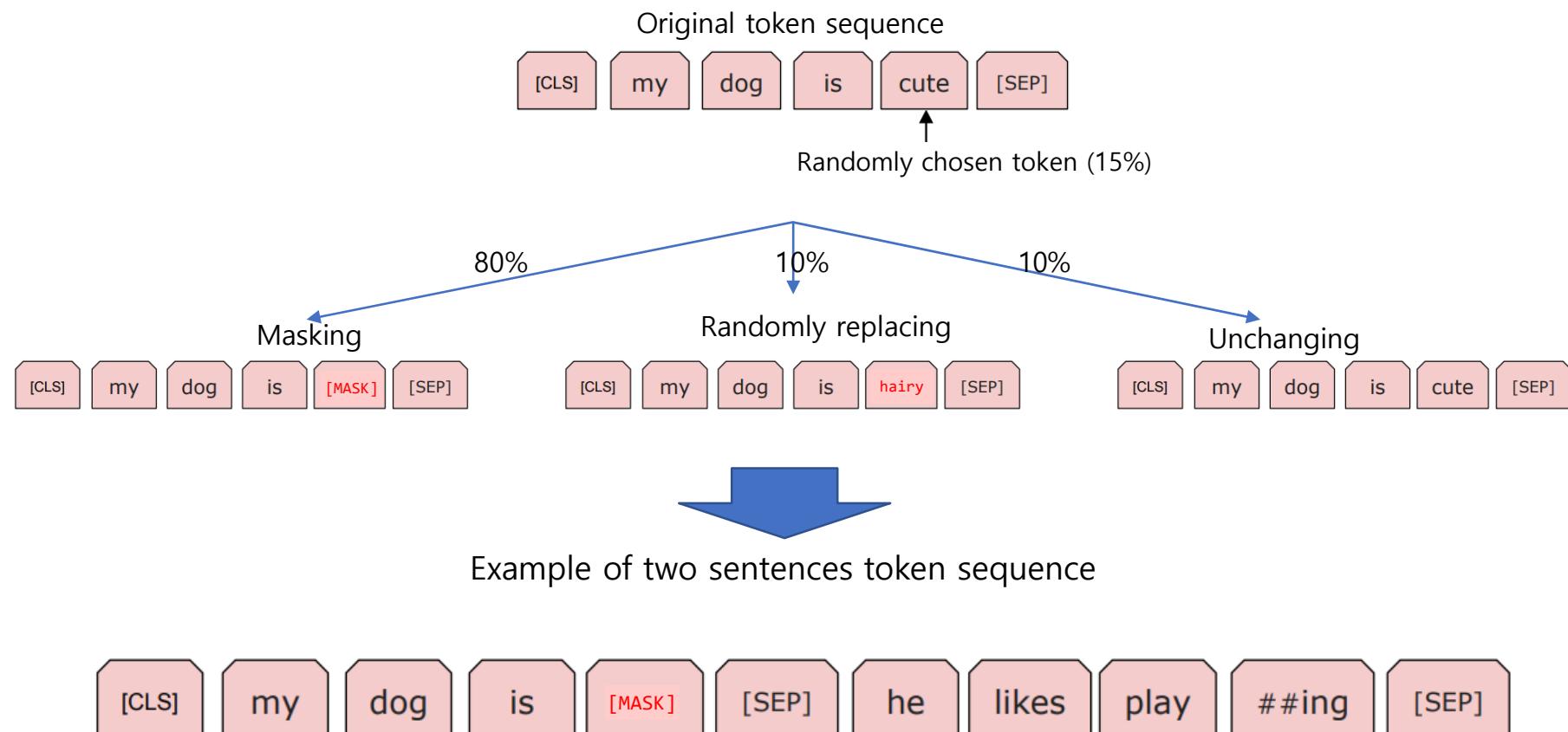
- BERT tokenization 예시

BPE로 subword로 분리했음에도 불구하고 vocab에 존재하지 않는 단어는 [UNK] 토큰으로 변환

[INFO:tensorflow:tensors: [CLS] 유스티니아 ##누스의 분노 ##는 찌 ##를 듯 ##했으나 수도 ##를 떠나는 실수를 한 바람에 필리 ##피 ##> 쿠스가 콘스탄티노폴리스 ##를 차지하였고 유스티니아 ##누스는 11월 4일 불잡혀 처형 ##당했다 ##. [SEP] ##여 ##비 ##렸고 유스티니아 ##누스의 아들 6 [MASK] 티 ##베리우스 ##도 블라 ##게 ##르나이 성당에서 처형 ##당 ##했는데 이로써 [MASK] ##클리우스 왕조의 혈 ##통 ##은 완전히 끊 ##어졌다 ##. [UNK] 전동차의 공기 ##제동 방식 가운데 하나이다 ##. 1868년 ##에 조지 웨 ##스팅 [MASK] 발명 ##하 > 였다 ##. 순수 ##하게 공기 ##압 제어로 동작 ##하는 것은 크게 단행 ##운전용인 [UNK] 연결 ##운전용인 비상 ##변 부착 [UNK] 나뉜다 ##. 개요 SM 공급 [UNK] Reser ##v ##o ##ir ##: [UNK] 불리는 가 ##압 ##된 공기 탱 ##크로부터 ##, 운전 ##대 ##까지 연결된 공기 ##저장관으로 불리는 공기관 ##을 통해 공기 ##압 ##을 공급 ##하여 제동 ##변을 조작 ##해 개폐 ##하는 것으로 ##, [UNK] Air P ##ip ##e # #: [UNK] 불리는 브레이크 실린 ##더 직결 공기관 ##에 가 ##압 ##하여 제동력을 얻는 매우 단순한 제동 시스템이다 ##. 웨 ##스팅 ##하우스가 불린 공식 이름은 [UNK] a ##ir b ##ra ##ke ##/ ##M ##otor c ##ar ##: 전동차 641 [UNK] 세계 여러 나라의 단행 ##운전차량에 널리 보급 [MASK] 노 ##면 ##전차 ##의 경우 아직도 영업 일 ##선에서 많이 쓰이고 있다. 다만 이 시스템은 구조가 간단 ##하며 동작 ##이 신 ##속 ##하고 확실 ##하지만 ##, 공기관 ##이 파 ##손 ##될 경우 제동 ##이 걸 ##리지 않게 되는 위험 ##이 있기 때문에 ##, 보안 ##상 연결 운전에는 사용할 수 없다는 단 ##점이 있다. SME SM [MASK] ##에서 문제가 된 열차 ##분리 [MASK] 발생 등의 대응 ##책 ##으 > 로 비상용 자동 ##공기 ##제동 [UNK] 그 지 ##령에 이용하는 [UNK] [UNK] 함께 설치한 [UNK] a ##ir b ##ra ##ke ##/ ##M ##otor c ##ar ##/ ##E ##mer ##gen [MASK] v ##al ##ve ##: 전동차 ##용 비상 ##변 부착 직통 ##공기 ##제동 ##. 모터 ##가 없는 트 ##레일 ##러 ##용 ##은 ST ##E 혹은 [UNK] 웨 ##스팅 ##하우스 ##사가 개발 ##하여 2~ ##3 ##량 정도의 단편성 ##용으로 보급 ##되었다 ##. 이 SME ##는 원형 ##인 SM ##과 같은 직통 제동 기구를 이루고 있지만 ##, [MASK] 저장기에 [MASK] 공기 ##탱 ##크가 원 ##공기 [UNK] Reser [MASK] ##o ##ir ##: [UNK] 불리며 공기 저장관 ##도 원 ##공기 [UNK] [MASK] ##v ##o ##ir P ##ip ##e ##: [UNK] 불리고 있다. 이것은 SME [MASK] [MASK] ##제동부에 비상 ##제동의 동력 ##원을 공급 ##하는 보조 공기 ##저장기가 존재 ##하기 때문에 ##, [MASK] 구별 ##하기 위 ##함이다 ##. 비상 ##변 ##에는 평 ##상 ##시는 490 ##k ##P ##a ##의 압 ##력이 가해 ##지고 있어 ##, 긴급 ##할 때 ##나 비상 ##변 ##의 호 ##스가 파 ##열 ##되었을 때도 비상 ##제동 ##이 작동 ##한다 ##. 비상 ##제동 ##은 자동 ##공기 ##제동 ##과 같이 보조 공기 ##저장기의 공기를 배출시키 ##는 것으로 작동시키기 때문에 ##, 안전성이 향상 ##되었다 ##. 브레이크의 가 ##감 ##압 ##은 [MASK] ##래 > 의 SM 방식 ##과 달리 가 ##감 ##압 ##의 속도가 언제나 정해져 있다. 제동 단계는 [UNK] [UNK] [UNK] [UNK] [UNK] 압 ##력을 [UNK] [UNK] 압 ##력을 [UNK] 이렇게 4 ##가지 ##가 있다. 제2차 십자군 (##114 ##7년 - [UNK] 제1차 십자군 [MASK] 이후 팔레스타 ##인의 십자 >

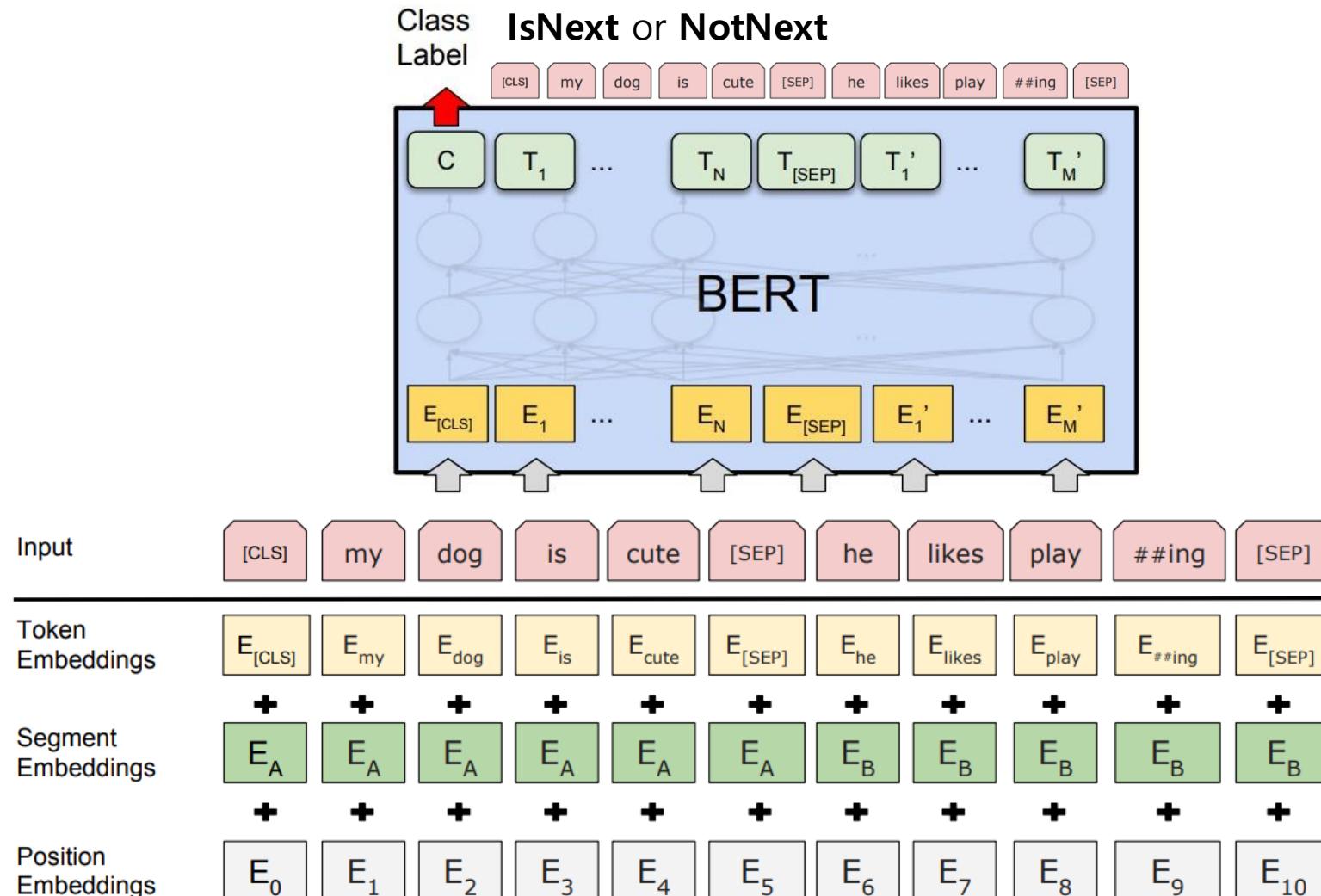
- 데이터의 tokenizing

Masked language model (MLM): input token을 일정 확률로 masking

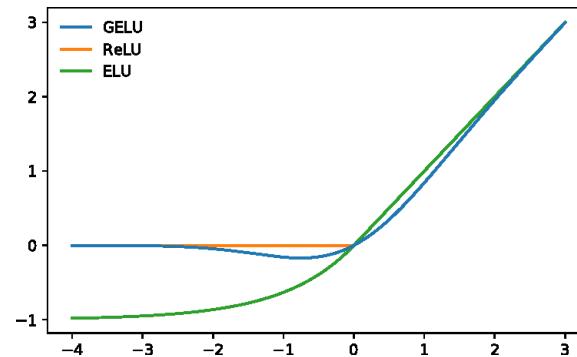


Methods – Masking 기법

10.1 뷰트(BERT)

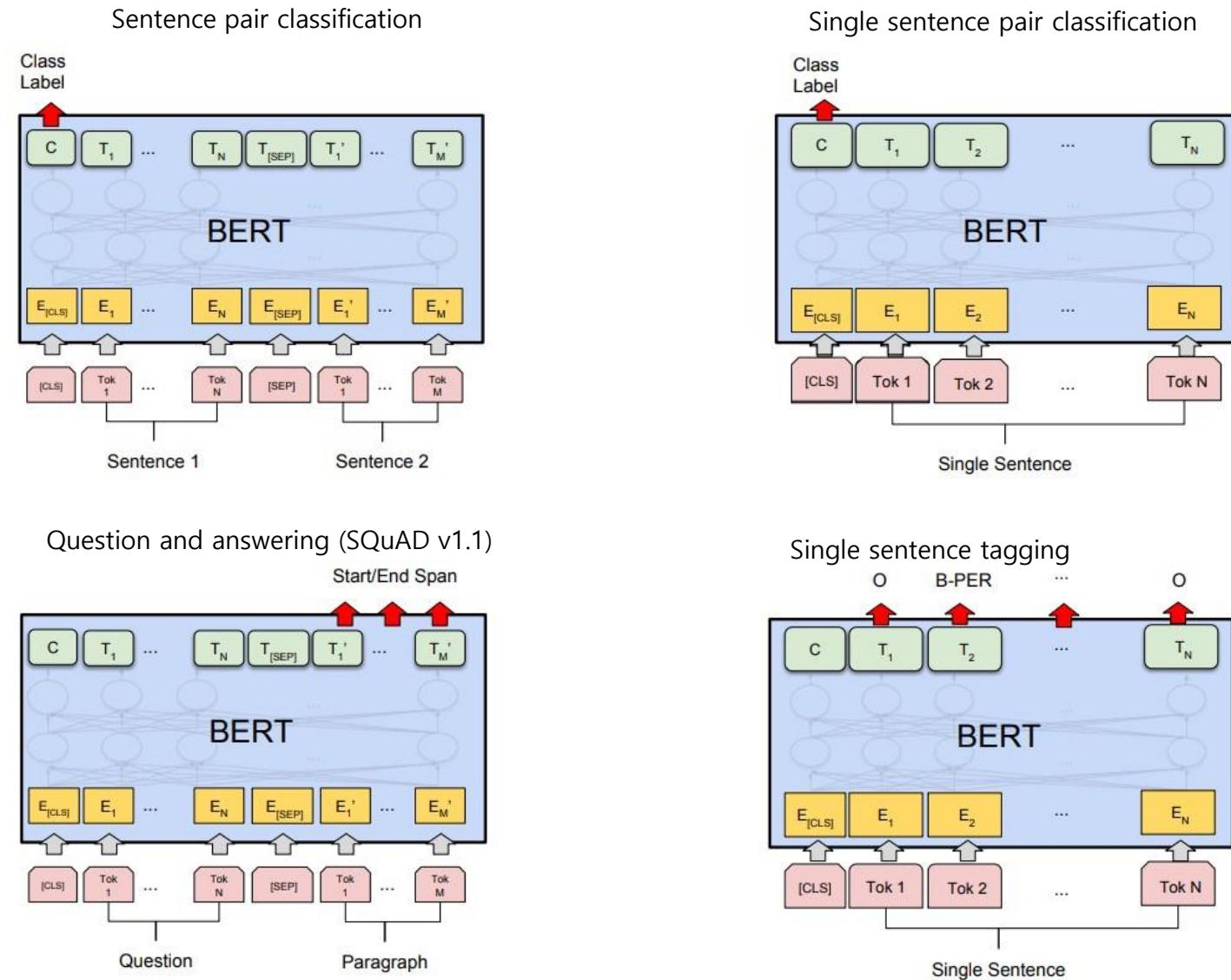


- Training options
 - Train batch size: 256 sequences (256 sequences * 512 tokens = 128,000 tokens/batch)
 - Steps: 1M
 - Epoch: 40 epochs
 - Adam learning rate: $1e-4$
 - Weight decay: 0.01
 - Drop out probability: 0.1
 - Activation function: GELU
- Environmental setup
 - $BERT_{BASE}$: 4 Cloud TPUs (16 TPU chips total)
 - $BERT_{LARGE}$: 16 Cloud TPUs (64 TPU chips total) \approx 72 P100 GPU
 - Training time: 4 days



Methods

10.1 뷔트(BERT)



real	1	1	0	1	1	1	0	0	1	0
pred	1	1	0	1	0	0	0	1	0	1

정확도 : $\frac{TP+TN}{TP+FP+TN+FN} = \frac{3+2}{3+2+3+2} = 0.5$

		실제 정답	
		True	False
분류 결과	True	True Positive	False Positive
	False	False Negative	True Negative

F1 스코어의 계산법
: 정밀도와 재현률의 조화 평균

일반평균 : $(\text{정밀도} + \text{재현률}) / 2$ 0.55
 0.9, 0.1 $\Rightarrow 0.5$

조화평균 : $\frac{2}{\frac{1}{\text{정밀도}} + \frac{1}{\text{재현률}}} = \frac{2 * \text{정밀도} * \text{재현률}}{\text{정밀도} + \text{재현률}}$

정밀도 : $\frac{TP}{TP+FP} = \frac{3}{3+2} = 0.6$

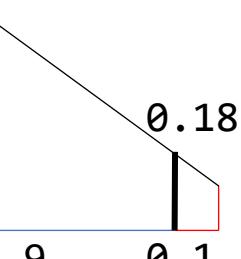
예) 스팸메일
1. 스팸메일을 일반메일로 판단.
2. 일반메일을 스팸메일로 판단.

재현률 : $\frac{TP}{TP+FN} = \frac{3}{3+3} = 0.5$

예) 악성코드
1. 악성코드를 일반코드로 판단.
2. 일반코드를 악성코드로 판단.

$$\frac{2 * 0.6 * 0.5}{0.6 + 0.5} \quad 0.5454\dots$$

$$\frac{2 * 0.9 * 0.1}{0.9 + 0.1} \quad 0.18$$



1839년 바그너는 괴테의 파우스트을 처음 읽고

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

1	8	3	9	년		바	그	너	는		괴	테	의		파	우	스	트	을	
---	---	---	---	---	--	---	---	---	---	--	---	---	---	--	---	---	---	---	---	--

['', '1839', '#년', '바', '#그', '#너', '#는', '괴', '#테', '#의', '파', '#우스', '#트', '#을'

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

CLS	1839	#년	바	#그	#너	#는	괴	#테	#의	파	#우스	#트	#을	
-----	------	----	---	----	----	----	---	----	----	---	-----	----	----	--

[(0, 0), (0, 4), (4, 5), (6, 7), (7, 8), (8, 9), (9, 10), (11, 12), (12, 13), (13, 14),
(15, 16), (16, 18)]