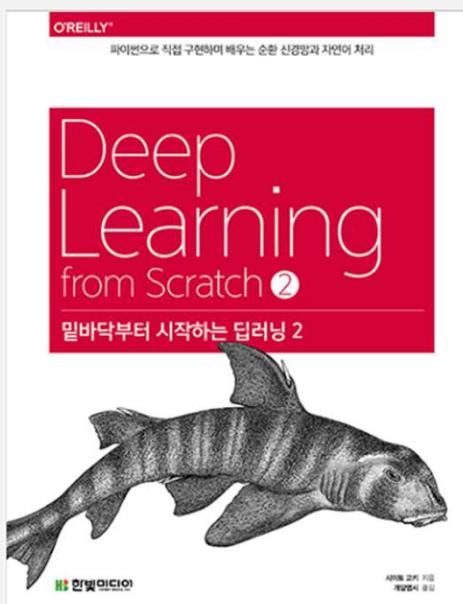


NLP 개론 (2)

Word2vec 속도 개선

세미나 계획

- 9월 10일 (화) : 자연어와 단어의 분산표현과 word2vec (윤경주)
- 9월 17일 (화) : 추석
- **9월 24일 (화) : word2vec(2)과 속도 개선(김건휘)**
- 10월 1일 (화) : 순환신경망(RNN) (윤경주)
- 10월 8일 (화) : 게이트가 추가된 RNN (김건휘)
- 10월 15일 (화) : RNN을 사용한 문장 생성 (윤경주)
- 10월 22일 (화) : 어텐션 (김건휘)



<https://github.com/WegraLee/deep-learning-from-scratch-2>

This branch is 73 commits ahead of, 22 commits behind [oreilly-japan/deep-learning-from-scratch-2](#).

Author	Commit Message	Date
WegraLee	Update README.md	c02db32 - 2 months ago
	ch01	출처문 번역
	ch02	출처문 번역
	ch03	주석 번역
	ch04	주석 번역
	ch05	출처문 번역
	ch06	출처문 번역 오류 수정
	ch07	train_seq2seq.py 변수명 오타 수정
	ch08	경로 설정 오류 수정
	common	util.py 오류 수정
	dataset	주석 번역

<https://github.com/WegraLee/deep-learning-from-scratch-2>

자연어와 단어의 분산표현과 word2vec 복습

- 자연어 처리

단어 : 의미의 최소 단위 (한국어의 경우에는 형태소)

단어를 컴퓨터에게 이해시키는 것이 자연어 처리에서 기본, 중요

단어 의미 파악 기법

1. 시소러스
2. 통계 기반 기법
3. 추론 기반 기법 (word2vec)

자연어와 단어의 분산표현과 word2vec 복습

- 단어 의미 파악 기법
 1. 시소러스
 - 뜻에 의해 그룹화 하고 계층별로 나누어 놓은 유의어 사전 문제점
 - * 수작업 레이블링
 - * 언어의 유연성에 대응하기 어려움
 - * 사람을 쓰는 비용이 크다
 - * 단어의 미묘한 차이를 표현 할 수 없다

자연어와 단어의 분산표현과 word2vec 복습

- 단어 의미 파악 기법

대량의 텍스트 데이터로부터 컴퓨터 스스로 단어의 의미를 추출,
관계를 설정 (말뭉치_corpus : 대량의 텍스트 데이터)

2. 통계 기반 기법

목표 : 단어의 분산 표현을 밀집벡터로 구축

가설 : 분포가설 (단어의 의미는 주변 단어에 의해 형성)

방법 : 어떤 단어가 얼마나 등장하는지 집계

-> 동시발생 행렬을 만든다.

행 벡터가 주목 단어의 코드가 된다.

(* 더 나아가, PPMI를 사용하고, SVD를 이용해 차원 감소)

-> 유사도 측정 가능

문제점 : SVD 를 $n \times n$ 행렬에 적용시, $O(n^3)$

다른 고급 SVD 방법론과 희소행렬의 성질을 이용하여도,
많은 시간과 비용 발생

자연어와 단어의 분산표현과 word2vec 복습

- 단어 의미 파악 기법

대량의 텍스트 데이터로부터 컴퓨터 스스로 단어의 의미를 추출,
관계를 설정 (말뭉치_corpus : 대량의 텍스트 데이터)

3. 추론 기반 기법

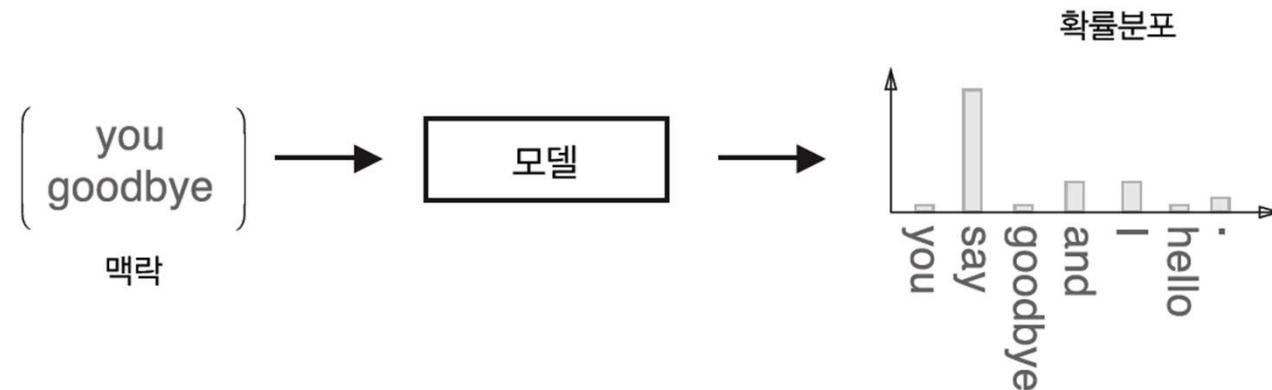
목표 : 단어의 분산 표현을 밀집벡터로 구축

가설 : 분포가설 (단어의 의미는 주변 단어에 의해 형성)

방법 : 추론 문제를 풀고 학습한다.

* 맥락을 입력 받아 출현할 수 있는 각 단어의 출현 확률을 출력
신경망을 사용하여 단어를 처리한다

그림 3-3 추론 기반 기법: 맥락을 입력하면 모델은 각 단어의 출현 확률을 출력한다.



자연어와 단어의 분산표현과 word2vec 복습

- CBOW 모델

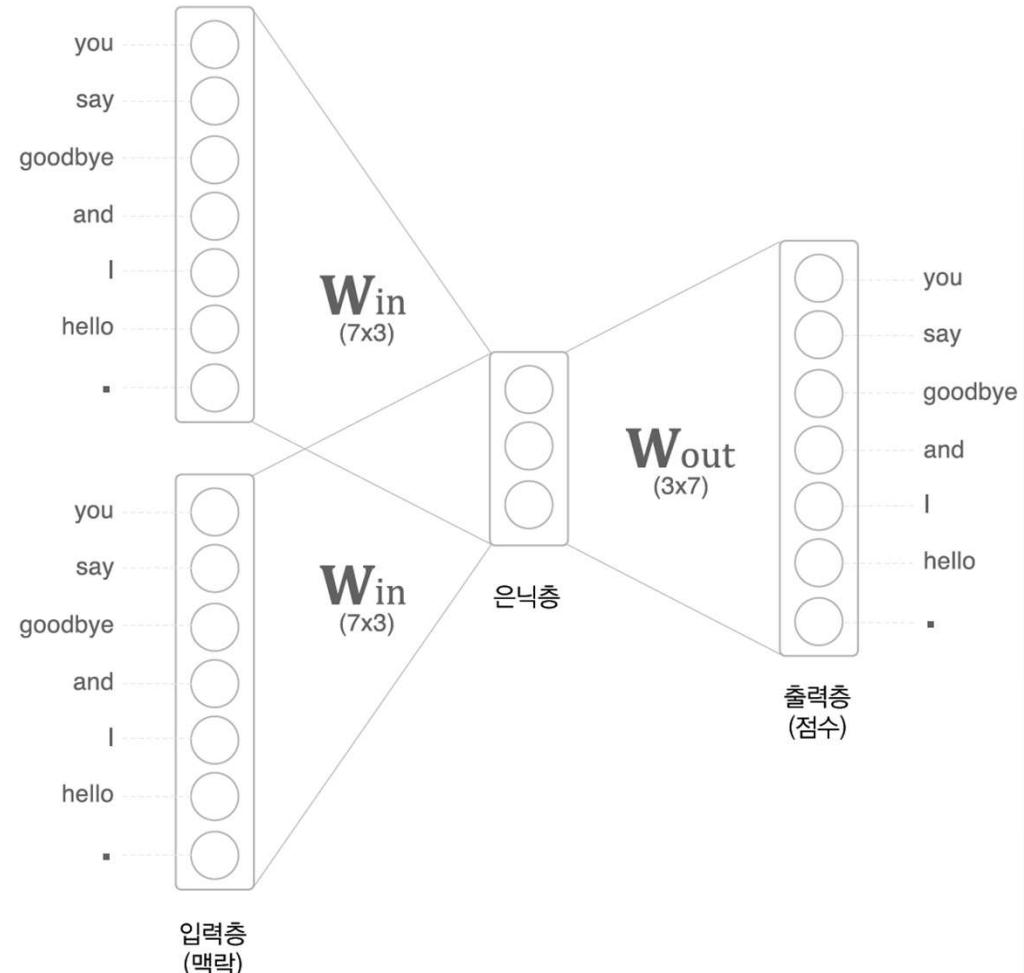
CBOW 모델 : 맥락으로부터 타깃을 추측하는 용도의 신경망

그림 3-9 CBOW 모델의 신경망 구조

입력 : 맥락(원핫 표현으로 변환)

은닉층 : 은닉층의 뉴런은 입력층의 완전연결계층에 의해 변환된 값이 된다. (여러 개이면 평균 낸 값)

출력층 : 각 단어의 점수를 뜻하며, 값이 높을수록 대응 단어의 출현확률이 높아진다.
(마지막에 softmax로 점수-> 확률 변환)

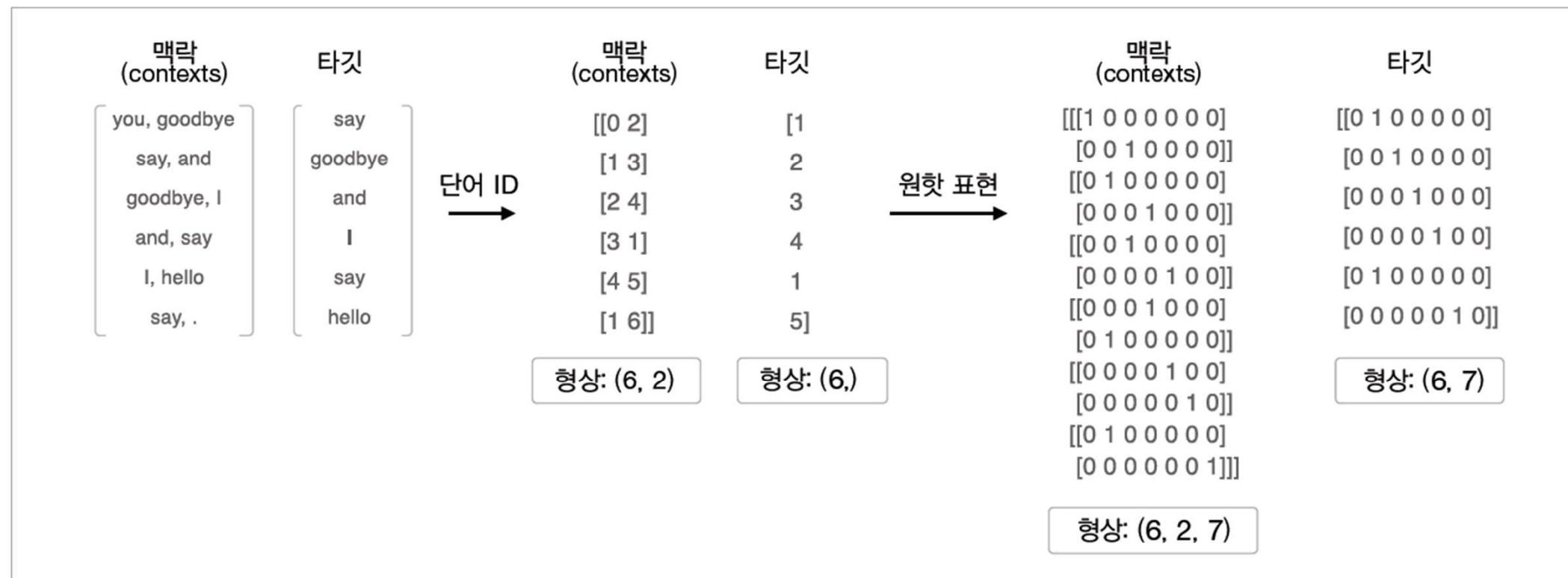


Word2vec_3.3 학습 데이터 준비

- 학습 데이터 준비

신경망 이용하기 위한 입력층 전처리 과정 : 맥락을 원핫 표현으로 변환

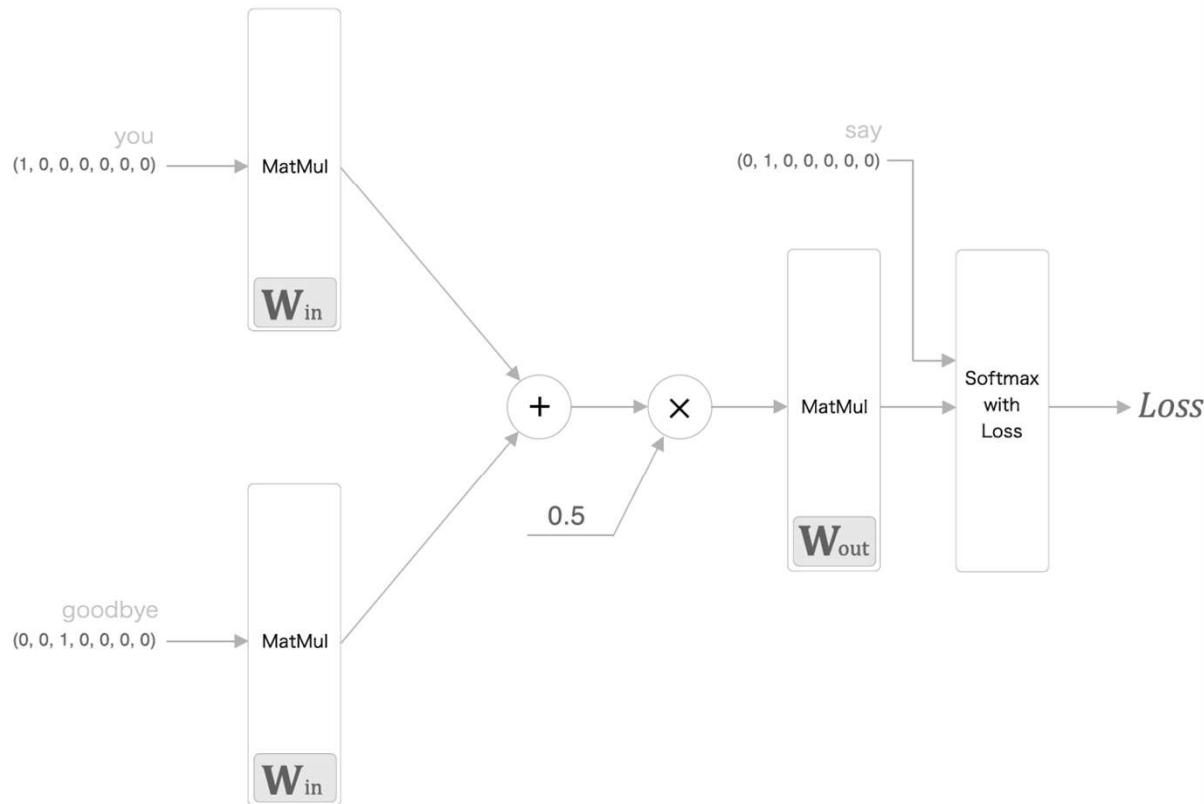
그림 3-18 ‘맥락’과 ‘타깃’을 원핫 표현으로 변환하는 예



Word2vec_3.4 CBOW 모델 구현

- Forward

그림 3-19 CBOW 모델의 신경망 구성



```

class SimpleCBOW:
    def __init__(self, vocab_size, hidden_size):
        V, H = vocab_size, hidden_size

        # 가중치 초기화
        W_in = 0.01 * np.random.randn(V, H).astype('f')
        W_out = 0.01 * np.random.randn(H, V).astype('f')

        # 계층 생성
        self.in_layer0 = MatMul(W_in)
        self.in_layer1 = MatMul(W_in)
        self.out_layer = MatMul(W_out)
        self.loss_layer = SoftmaxWithLoss()

        # 모든 가중치와 기울기를 리스트에 모은다.
        layers = [self.in_layer0, self.in_layer1, self.out_layer]
        self.params, self.grads = [], []
        for layer in layers:
            self.params += layer.params
            self.grads += layer.grads

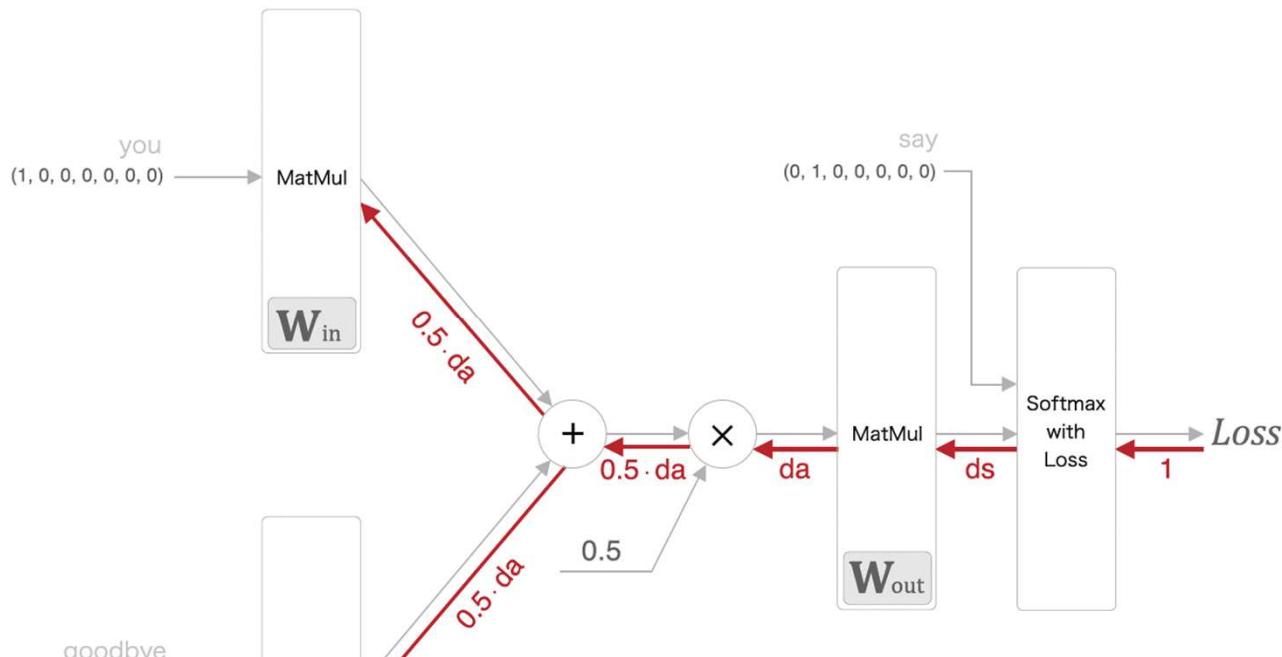
        # 인스턴스 변수에 단어의 분산 표현을 저장한다.
        self.word_vecs = W_in

    def forward(self, contexts, target):
        h0 = self.in_layer0.forward(contexts[:, 0])
        h1 = self.in_layer1.forward(contexts[:, 1])
        h = (h0 + h1) * 0.5
        score = self.out_layer.forward(h)
        loss = self.loss_layer.forward(score, target)
        return loss
  
```

Word2vec_3.4 CBOW 모델 구현

- Backward

그림 3-20 CBOW 모델의 역전파(역전파의 흐름은 두꺼운(붉은) 화살표로 표시)



```
def backward(self, dout=1):
    ds = self.loss_layer.backward(dout)
    da = self.out_layer.backward(ds)
    da *= 0.5
    self.in_layer1.backward(da)
    self.in_layer0.backward(da)
    return None
```

그림 1-18 덧셈 노드의 순전파(왼쪽)와 역전파(오른쪽)

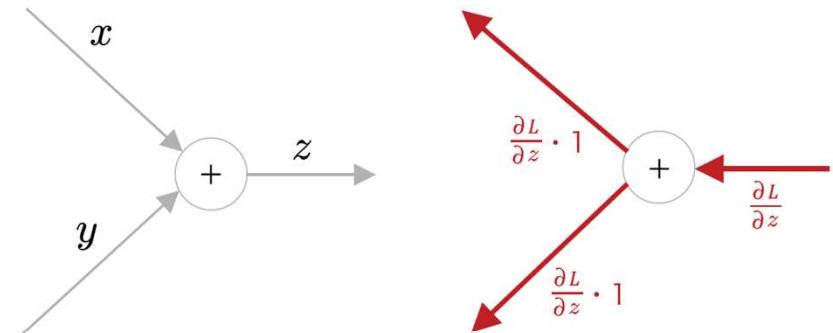
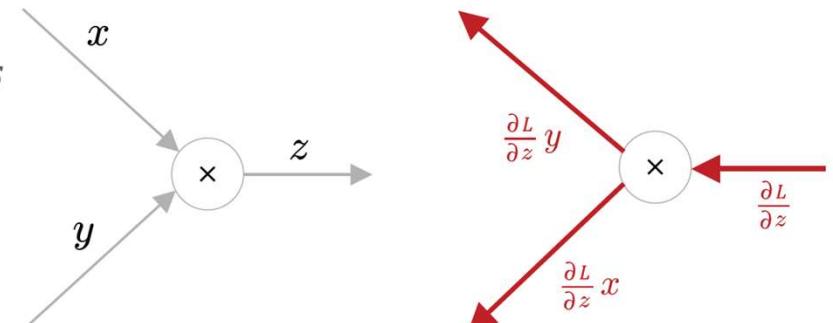


그림 1-19 곱셈 노드의 순전파(왼쪽)와 역전파(오른쪽)



Word2vec_3.4 CBOW 모델 구현

- 학습 코드 구현

학습 데이터로부터 미니배치를 선택

→ 신경망에 입력하여 기울기 계산

→ 기울기를 Optimizer에 넘겨 매개변수를 갱신

```
window_size = 1
hidden_size = 5
batch_size = 3
max_epoch = 1000

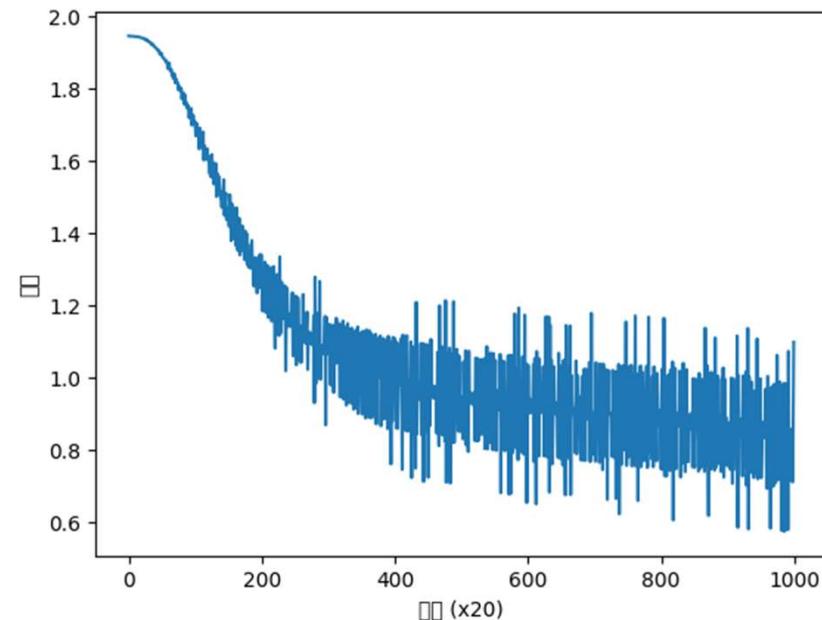
text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)

vocab_size = len(word_to_id)
contexts, target = create_contexts_target(corpus, window_size)
target = convert_one_hot(target, vocab_size)
contexts = convert_one_hot(contexts, vocab_size)

model = SimpleCBOW(vocab_size, hidden_size)
optimizer = Adam()
trainer = Trainer(model, optimizer)

trainer.fit(contexts, target, max_epoch, batch_size)
trainer.plot()

word_vecs = model.word_vecs
for word_id, word in id_to_word.items():
    print(word, word_vecs[word_id])
```



학습할 수록 손실 줄어듦

Word2vec_3.4 CBOW 모델 구현

- 학습 코드 구현_학습 종료

Word_vec의 각 행에는 대응하는 ID의 분산 표현이 밀집 벡터 형식으로 저장되어 있다.

그림 3-8 맥락 c 와 가중치 W 의 곱으로 해당 위치의 행벡터가 추출된다(각 요소의 가중치 크기는 흑백의 진하기로 표현).

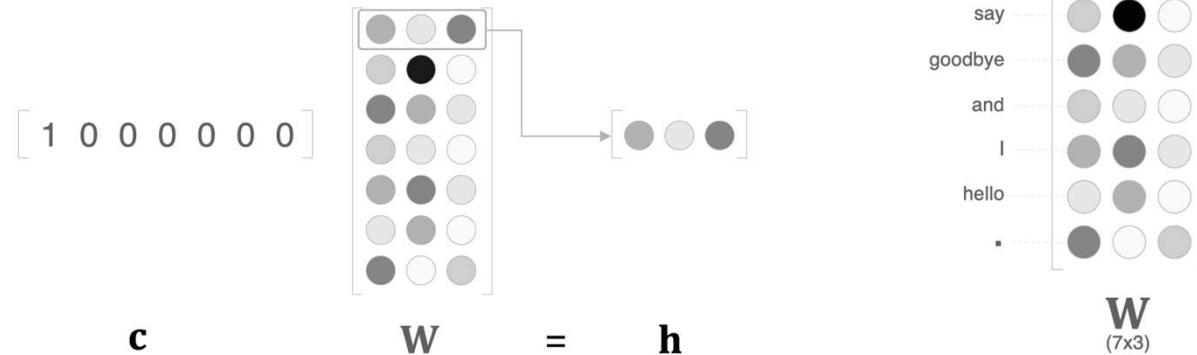
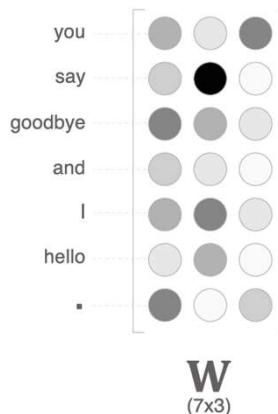


그림 3-10 가중치의 각 행이 해당 단어의 분산 표현이다.



```

you [-1.2538884 -1.1152751 -1.2269778 -1.2006699 -1.110759 ]
say [1.1966313 1.1317781 1.1526186 1.1645992 1.1773382]
goodbye [-0.61490333 -0.8291922 -0.6630883 -0.7359119 -0.8760589 ]
and [1.0920745 1.3939579 1.1703682 1.2613498 0.41214356]
i [-0.6123069 -0.8329343 -0.679545 -0.72606224 -0.8856468 ]
hello [-1.24547 -1.1192364 -1.2441229 -1.2051696 -1.1059659]
. [0.9172596 0.2329685 0.6512682 0.54689914 1.5007405 ]

```

현재는 단순한 CBOW -> 말뭉치를 늘려서 실제 CBOW를 4장에서 구현

Word2vec_3.5 word2vec 보충

- CBOW 모델과 확률

CBOW 모델이 하는 일 : 맥락이 주어졌을 때, 타깃 단어가 나올 확률을 출력한다.

NOTE) 사후 확률 : $P(A|B)$ B라는 정보가 주어졌을 때 A가 일어날 확률

→ CBOW 모델을 확률 표기법으로 기술할 수 있다. $P(w_t | w_{t-1}, w_{t+1})$

그림 3-22 word2vec의 CBOW 모델(맥락의 단어로부터 타깃 단어를 추측)



Word2vec_3.5 word2vec 보충

- CBOW 모델의 손실함수 표현

이 책에서 채택하는 손실함수는 교차 엔트로피 오차로, 다음과 같다.

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

y_k : k번째에 해당하는 사건이 일어날 확률

t_k : 정답 레이블이며 원핫 벡터로 표현

wt에 해당하는 원소만 1, 나머지는 0

→ 위 식에서 다음과 같이 변환 $L = -\log P(w_t | w_{t-1}, w_{t+1})$

말뭉치 전체로 확장하면 다음과 같다. $L = -\frac{1}{T} \sum_{t=1}^T \log P(w_t | w_{t-1}, w_{t+1})$

CBOW 모델은 이 함수를 가능한 작게 만드는 것이다.

이 과정에서 나오는 가중치 매개변수 : 단어의 분산 표현

Word2vec_3.5 word2vec 보충

- Skip-gram 모델

Word2vec에서 제안하는 2가지 모델 : CBOW, Skip-gram

CBOW : 맥락이 주어지고 타깃 추측

Skip-gram : 타깃이 주어지고 맥락을 추측

그림 3-23 CBOW 모델과 skip-gram 모델이 다루는 문제



Word2vec_3.5 word2vec 보충

- Skip-gram 모델

입력층 : 타깃 단어 1 개

출력층 : 맥락의 수 만큼 존재

따라서 각 출력층에서는 Softmax with Loss 계층을 이용하여 개별 손실 구하고, 모두 더한 값을 최종 손실로 한다.
 (조건부 독립을 가정)

Skip 모델 확률 : $P(w_{t-1}, w_{t+1} | w_t) = P(w_{t-1} | w_t)P(w_{t+1} | w_t)$

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

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

Word2vec_3.5 word2vec 보충

- Skip-gram 모델

Skip-gram 모델의 손실 함수 : $L = -\frac{1}{T} \sum_{t=1}^T (\log P(w_{t-1} | w_t) + \log P(w_{t+1} | w_t))$

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

성능 우위 : Skip-gram 모델

학습 속도 우위 : CBOW 모델

- 단어 분산 표현의 평가 방법은 4.4에서 다룬다.

Word2vec_3.5 word2vec 보충

- 통계 기반 vs 추론 기반

통계 기반 기법 : - 말뭉치의 전체 통계로부터 1회 학습하여 단어의 분산 표현을 얻는다.
- 단어의 유사성이 인코딩 된다.

추론 기반 기법 : - 말뭉치를 일부분씩 여러 번 보면서 학습했다.
(미니배치)
- 단어의 유사성은 물론, 복잡한 단어 사이의 패턴까지
인코딩 된다.
→ 유추문제까지 풀 수 있다.

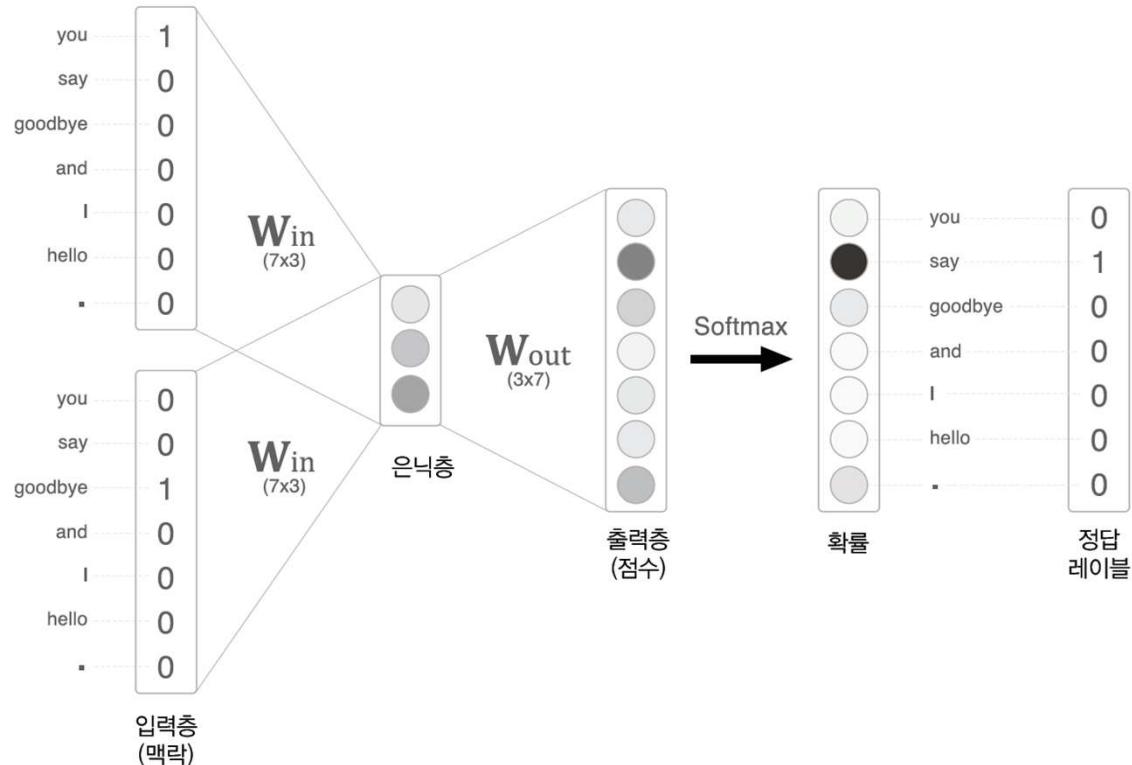
Word2vec_정리

- 추론 기반 기법은 추측하는 것이 목적이며, 부산물로 단어의 분산 표현을 얻을 수 있다.
- Word2vec은 추론 기법이며, 단순한 2층 신경망이다.
- Word2vec은 CBOW 모델과 Skip-gram 모델을 제공한다
- CBOW 모델은 맥락으로부터 하나의 단어를 추측한다
- Skip-gram 모델은 하나의 단어로부터 맥락을 추측한다.
- Word2vec은 가중치를 다시 학습할 수 있으므로, 단어의 분산 표현 갱신이나 새로운 단어 추가를 효율적으로 수행할 수 있다.

Word2vec 속도개선_4장

- 3장에서 구현한 CBOW 모델

그림 4-1 앞 장에서 구현한 CBOW 모델



입력 : 맥락 단어 2개

은닉층 : W_{in} 과의 행렬곱, W_{out} 과의 행렬곱

출력 : 각 단어의 점수 계산 후 Softmax 함수 적용 \rightarrow 단어 추출

학습 : 교차 엔트로피 오차를 적용하여 손실 구함

Word2vec 속도개선_4장

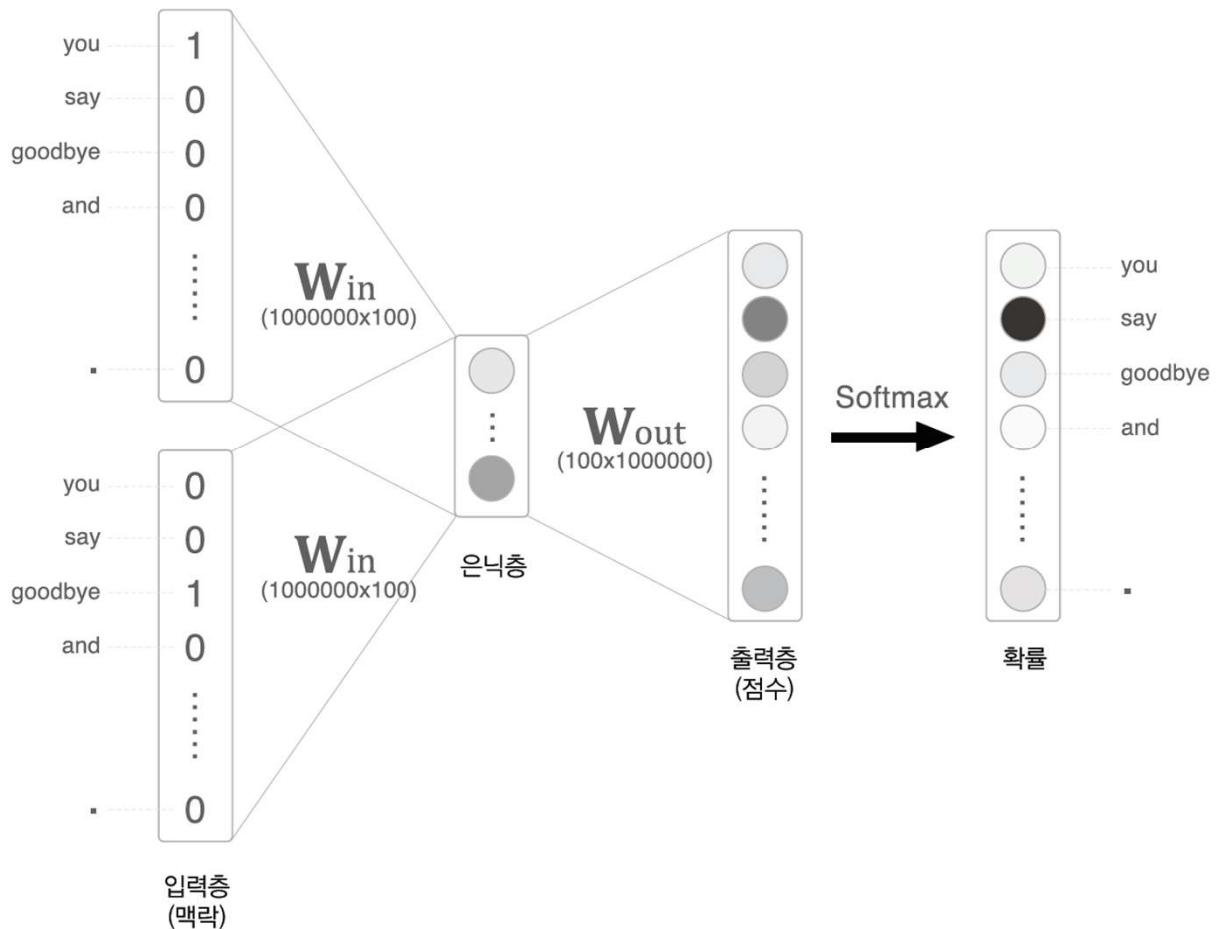
- 3장에서 구현한 CBOW 모델_ 어휘, 뉴런 증가 시킨 Ver

계산 병목 (문제점)

1. 입력층의 원핫 표현과 가중치 행렬 W_{in} 의 곱계산

2. 은닉층과 가중치 행렬 W_{out} 의 곱 및 Softmax 계층의 계산

그림 4-2 어휘가 100만 개일 때를 가정한 CBOW 모델



Word2vec 속도개선_4장

- 3장에서 구현한 CBOW 모델_ 어휘, 뉴런 증가 시킨 Ver
계산 병목 (문제점)

1. 입력층의 원핫 표현과 가중치 행렬 Win의 곱계산

- 단어를 원핫 벡터로 다루기 때문에, 원핫 표현의 벡터도 커진다.
(메모리 이슈)
- 여기에 가중치 행렬 Win을 곱해야 한다.
(계산량 이슈)

그림 2-5 단어 “you”의 맥락에 포함되는 단어의 빈도를 표로 정리한다.

	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

그림 2-7 모든 단어 각각의 맥락에 해당하는 단어의 빈도를 세어 표로 정리한다.

	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

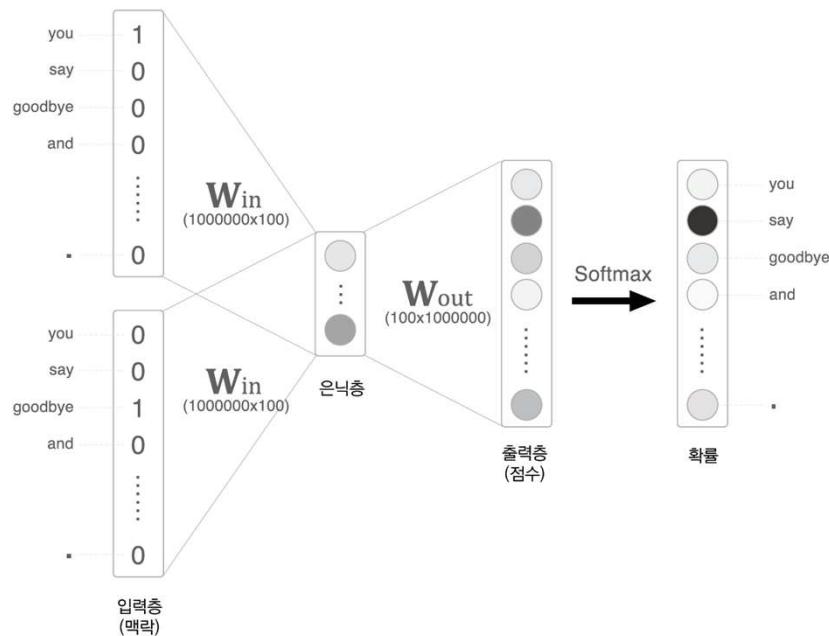
Word2vec 속도개선_4장

- 3장에서 구현한 CBOW 모델_ 어휘, 뉴런 증가 시킨 Ver 계산 병목 (문제점)

2. 은닉층과 가중치 행렬 W_{out} 의 곱 및 Softmax 계층의 계산

- 은닉층과 가중치 행렬 W_{out} 곱의 계산량 이슈
- Softmax 계층에서 다루는 어휘가 많아짐에 따라 계산량 증가 이슈

그림 4-6 어휘가 100만 개일 때를 가정한 word2vec: "you"와 "goodbye"가 맥락이고 "say"가 타깃(예측해야 할 단어)

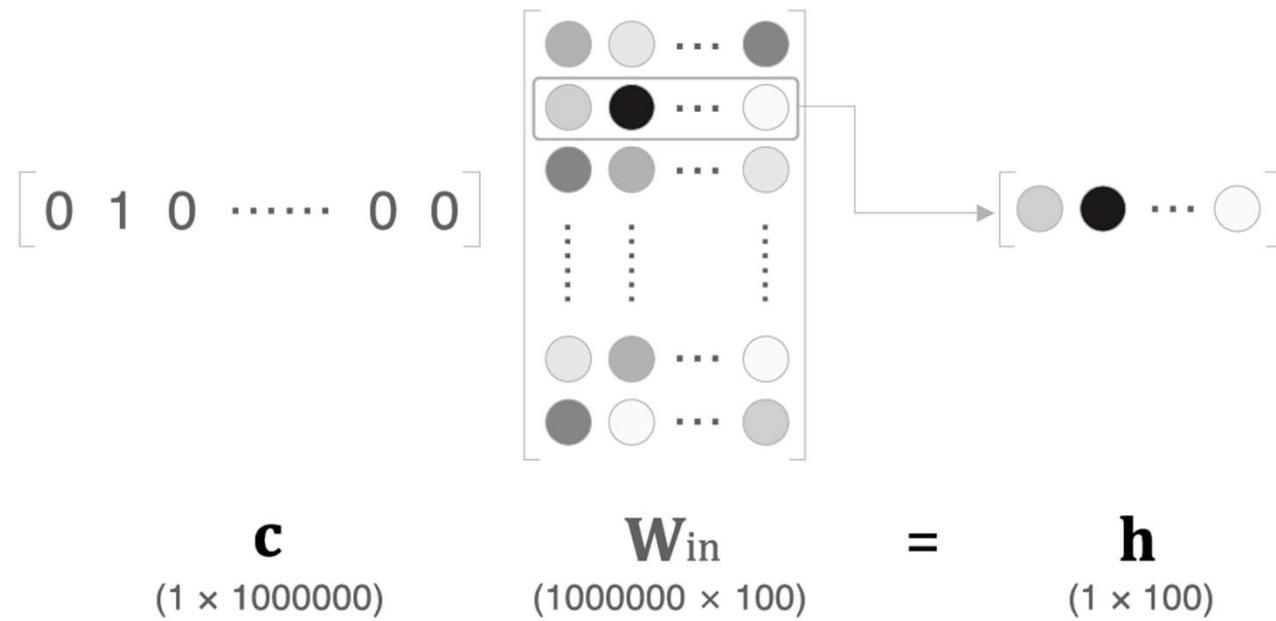


$$y_k = \frac{\exp(s_k)}{\sum_{i=1}^{1000000} \exp(s_i)}$$

Word2vec 속도개선_4.1 첫 번째 개선

- 3장에서 구현한 CBOW 모델_ 어휘, 뉴런 증가 시킨 Ver
 말뭉치 : 100만개 어휘 -> 단어 원핫 표현 : 100만 차원

그림 4-3 맥락(원핫 표현)과 MatMul 계층의 가중치를 곱한다.



결과 : 행렬의 특정 행만 추출하는 작업

아이디어 : 결과물이 같게 나오는 다른 방법을 생각해보자

→ Embedding 계층 : 가중치 매개변수로부터 단어 ID에 해당하는 행을 추출하는 계층

Word2vec 속도개선_4.1 첫 번째 개선

- Embedding 계층 구현

결과 : 행렬의 특정 행만 추출하는 작업

아이디어 : 결과물이 같게 나오는 다른 방법을 생각해보자

→ Embedding 계층 : 가중치 매개변수로부터 단어 ID에 해당하는 행을 추출하는 계층

행렬에서 특정 행만 추출하는 방법

```
[1]: import numpy as np
```

```
[3]: W = np.arange(21).reshape(7,3)
```

```
[4]: W
```

```
[4]: array([[ 0,  1,  2],
           [ 3,  4,  5],
           [ 6,  7,  8],
           [ 9, 10, 11],
           [12, 13, 14],
           [15, 16, 17],
           [18, 19, 20]])
```

```
[5]: W[2]
```

```
[5]: array([6, 7, 8])
```

```
[6]: W[5]
```

```
[6]: array([15, 16, 17])
```

```
[7]: idx = np.array([1, 0, 3, 0])
```

```
[8]: W[idx]
```

```
[8]: array([[ 3,  4,  5],
           [ 0,  1,  2],
           [ 9, 10, 11],
           [ 0,  1,  2]])
```

Word2vec 속도개선_4.1 첫 번째 개선

- Embedding 계층 구현

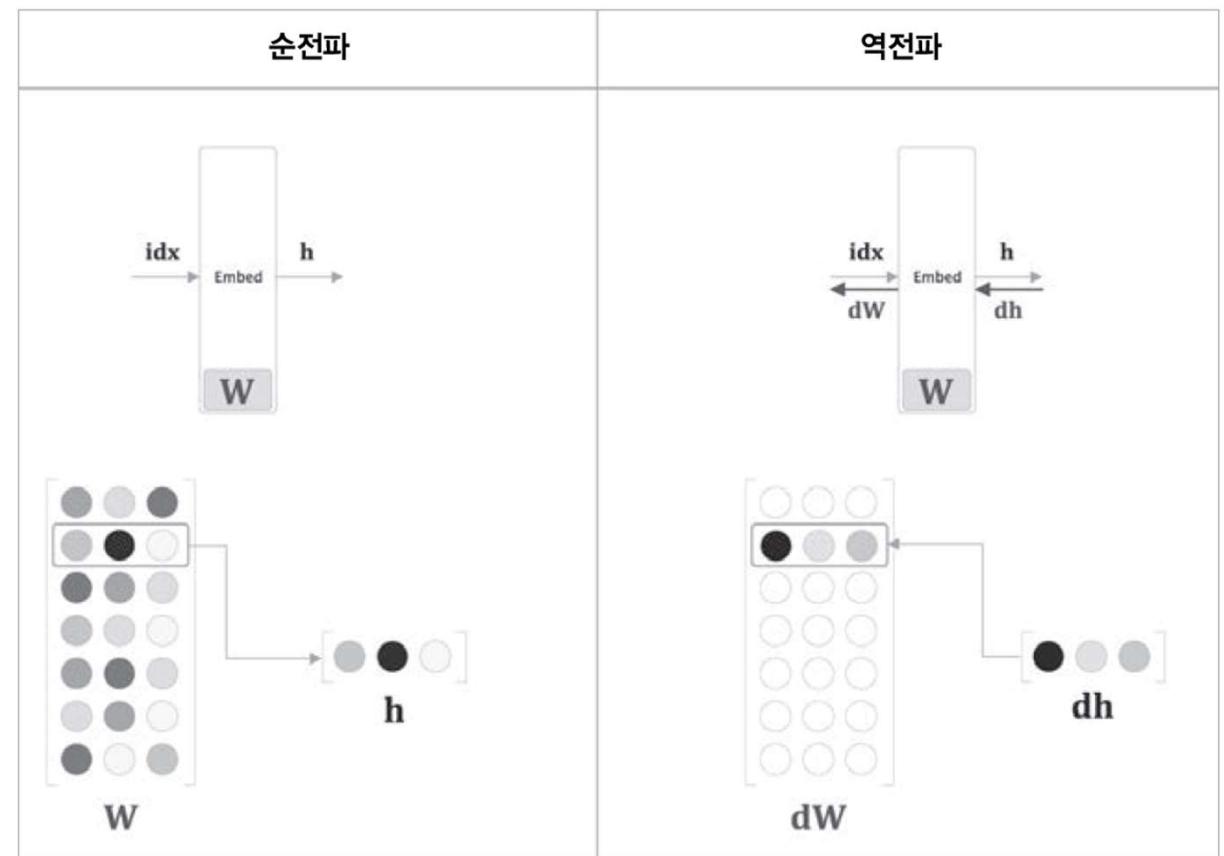
Embedding 계층 순전파 : 수학적인 연산 장치가 없다.

```
class Embedding: 6개의 사용 위치
    def __init__(self, W):
        self.params = [W]
        self.grads = [np.zeros_like(W)]
        self.idx = None

    def forward(self, idx):
        W, = self.params
        self.idx = idx
        out = W[idx]
        return out

    def backward(self, dout):
        dW, = self.grads
        dW[...] = 0
        np.add.at(dW, self.idx, dout)
        return None
```

그림 4-4 Embedding 계층의 forward와 backward 처리(Embedding 계층은 Embed로 표기)



Word2vec 속도개선_4.1 첫 번째 개선

- Embedding 계층 구현

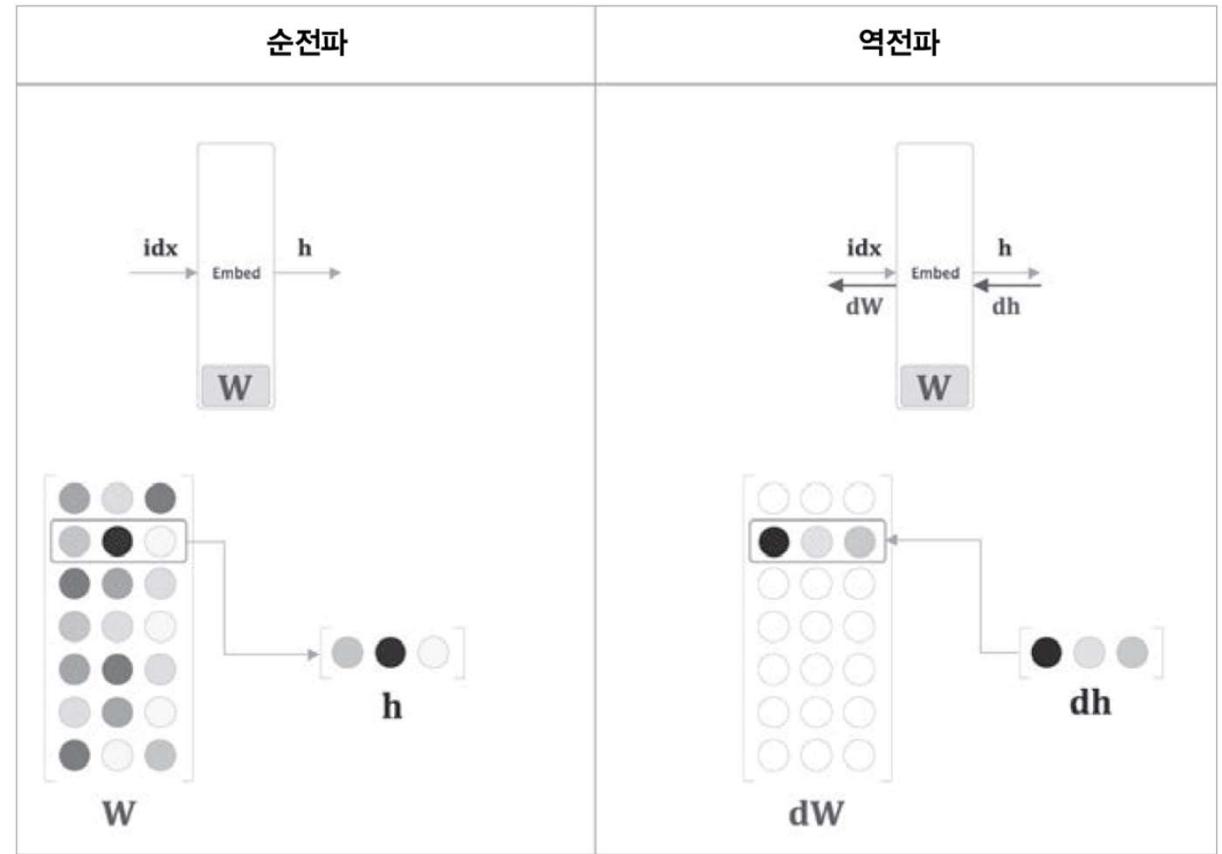
Embedding 계층 역전파 : 순전파에서 아무 장치가 없었기 때문에 받은 기울기를 전달만 하면 된다. 원하는 행의 분산 표현(기울기)만 전달하자

```
class Embedding: 6개의 사용 위치
    def __init__(self, W):
        self.params = [W]
        self.grads = [np.zeros_like(W)]
        self.idx = None

    def forward(self, idx):
        W, = self.params
        self.idx = idx
        out = W[idx]
        return out

    def backward(self, dout):
        dW, = self.grads
        dW[...] = 0
        np.add.at(dW, self.idx, dout)
        return None
```

그림 4-4 Embedding 계층의 forward와 backward 처리(Embedding 계층은 Embed로 표기)

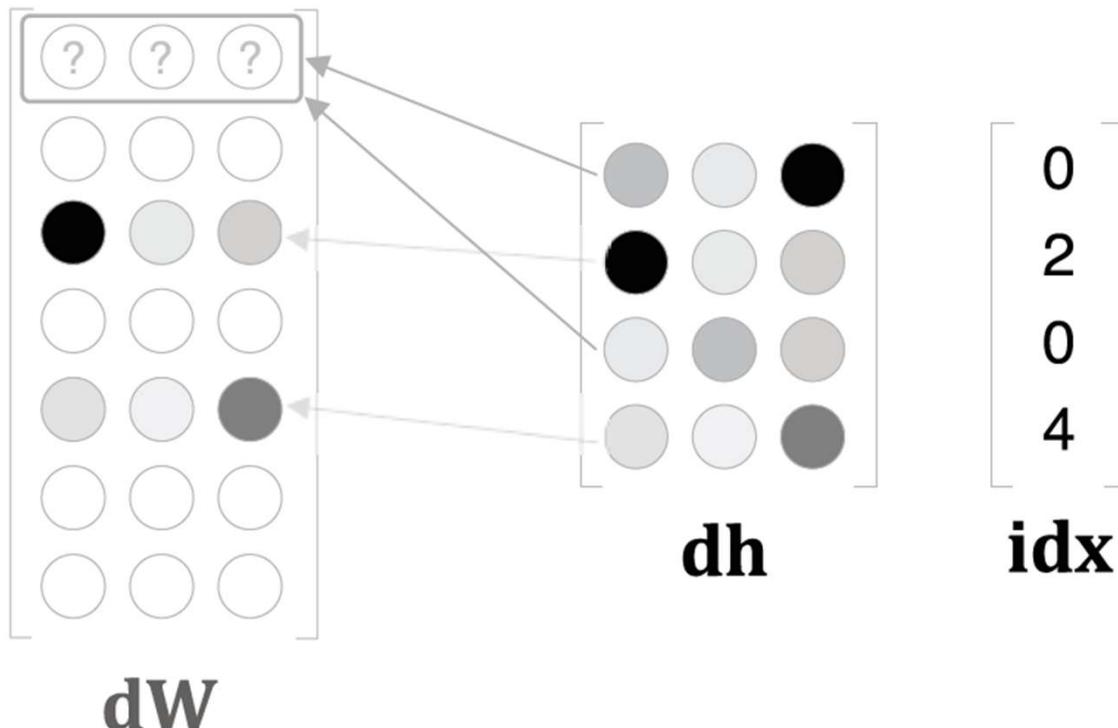


Word2vec 속도개선_4.1 첫 번째 개선

- Embedding 계층 구현

Embedding 계층 역전파 문제점 : idx의 원소가 중복 될 때 문제 발생

그림 4-5 idx 배열의 원소 중 값(행 번호)이 같은 원소가 있다면, dh를 해당 행에 할당할 때 문제가 생긴다.



중복 문제를 해결하기 위해서는 할당이 아닌, 더하기를 해야한다.
 (책 : 각자 생각해보기)

Word2vec 속도개선_4.1 첫 번째 개선

- Embedding 계층 구현

Embedding 계층 역전파 문제점 : idx의 원소가 중복 될때 문제 발생
중복 문제를 해결하기 위해서는 할당이 아닌, 더하기를 해야한다.

자세한 이유

기능 : 순전파에서 참조된 w행들에 대응하는 기울기 업데이트

계산 : 같은 행이 여러 번 참조된 경우, 역전파에서 그 행에 대한 기울기가 여러 번 계산됨

더하는 이유 : 각 참조의 기울기는 독립적으로 발생하므로,
그 행에 대해서 여러 번 발생한 기울기를 누적해야 한다.
즉 더해야 한다.

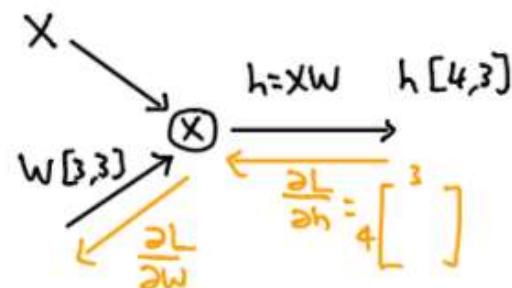
Word2vec 속도개선_4.1 첫 번째 개선

- Embedding 계층 구현

Embedding 계층 역전파 문제점 : idx의 원소가 중복 될때 문제 발생
 중복 문제를 해결하기 위해서는 할당이 아닌, 더하기를 해야한다.

(인터넷 검색_ 잘못된 이해 방식)

$$\text{ex) } X = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$



$$\begin{aligned} \frac{\partial L}{\partial w} &= X^T \frac{\partial L}{\partial h} \\ &= \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \text{---} \\ \text{---} \\ \text{---} \end{bmatrix} \\ &= \begin{bmatrix} \text{---} \\ \text{---} \\ \text{---} \end{bmatrix} \quad \text{---} \rightarrow \text{중복되는 부분은 더함!} \end{aligned}$$

Word2vec 속도개선_4.1 첫 번째 개선

- Embedding 계층 구현

Embedding 계층 역전파 문제점 : idx의 원소가 중복 될때 문제 발생
 중복 문제를 해결하기 위해서는 할당이 아닌, 더하기를 해야한다.

그림 1-25 MatMul 노드의 역전파

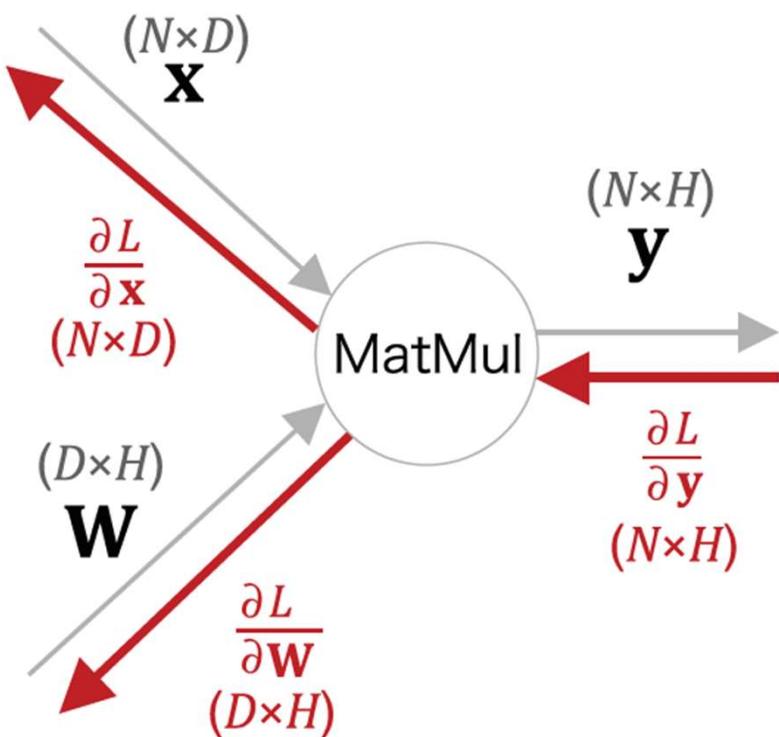


그림 1-26 행렬의 형상을 확인하여 역전파 식을 유도한다.

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

형상 : $N \times D$ $N \times H$ $H \times D$

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

형상 : $D \times H$ $D \times N$ $N \times H$

Word2vec 속도개선_4.1 첫 번째 개선

- Embedding 계층 구현

Embedding 계층 역전파 문제점 : idx의 원소가 중복 될때 문제 발생
 중복 문제를 해결하기 위해서는 할당이 아닌, 더하기를 해야한다.

그림 4-4 Embedding 계층의 forward와 backward 처리(Embedding 계층은 Embed로 표기)

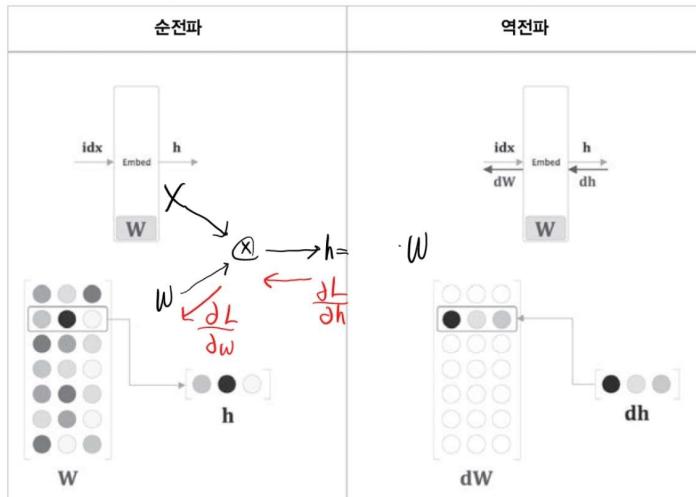
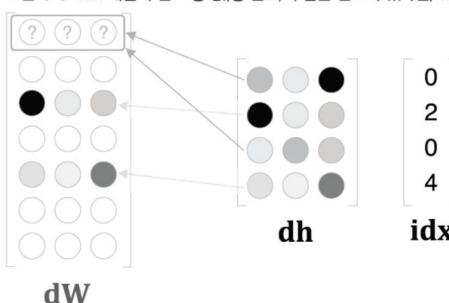


그림 4-5 idx 배열의 원소 중 값(행 번호)이 같은 원소가 있다면, dh를 해당 행에 할당할 때 문제가 생긴다



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

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

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

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

Word2vec 속도개선_4.1 첫 번째 개선

- Embedding 계층 구현

```
class Embedding: 6개의 사용 위치
    def __init__(self, W):
        self.params = [W]
        self.grads = [np.zeros_like(W)]
        self.idx = None

    def forward(self, idx):
        W, = self.params
        self.idx = idx
        out = W[idx]
        return out

    def backward(self, dout):
        dW, = self.grads
        dW[...] = 0
        np.add.at(dW, self.idx, dout)
        return None
```

word2vec의 구현 개선 :

입력층 MatMul 계층을 Embedding 계층으로
전환 할 수 있게 되었다.

효과 : 메모리 사용량 절약
불필요한 계산 생략

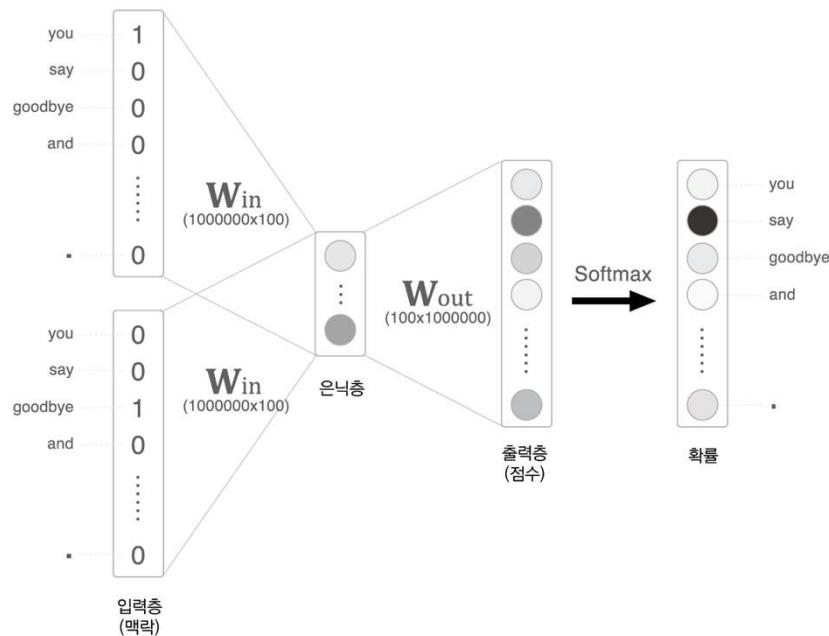
Word2vec 속도개선_4.2 두 번째 개선

- 3장에서 구현한 CBOW 모델_ 어휘, 뉴런 증가 시킨 Ver 계산 병목 (문제점)

2. 은닉층과 가중치 행렬 W_{out} 의 곱 및 Softmax 계층의 계산

- 은닉층과 가중치 행렬 W_{out} 곱의 계산량 이슈
- Softmax 계층에서 다루는 어휘가 많아짐에 따라 계산량 증가 이슈

그림 4-6 어휘가 100만 개일 때를 가정한 word2vec: "you"와 "goodbye"가 맥락이고 "say"가 타깃(예측해야 할 단어)



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

Word2vec 속도개선_4.2 두 번째 개선

- 3장에서 구현한 CBOW 모델_ 어휘, 뉴런 증가 시킨 Ver 계산 병목 (문제점)
2. 은닉층과 가중치 행렬 W_{out} 의 곱 및 Softmax 계층의 계산
 - 은닉층과 가중치 행렬 W_{out} 곱의 계산량 이슈

거대한 행렬을 곱하려면 시간, 메모리가 많이 필요하다
또한, 역전파 때도 같은 계산을 수행하기 때문에

행렬 곱 자체를 가볍게 만들어야 한다.

Word2vec 속도개선_4.2 두 번째 개선

- 3장에서 구현한 CBOW 모델_ 어휘, 뉴런 증가 시킨 Ver 계산 병목 (문제점)

2. 은닉층과 가중치 행렬 W_{out} 의 곱 및 Softmax 계층의 계산

- Softmax 계층에서 다루는 어휘가 많아짐에 따라 계산량 증가 이슈 아래의 식은 k번째 단어를 타깃으로 계산한 Softmax 계산식이다.

어휘 : 100만 개

Exp 계산 : 100만 번

따라서, Softmax를 대신할 가벼운 계산을 제시해야 한다.

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

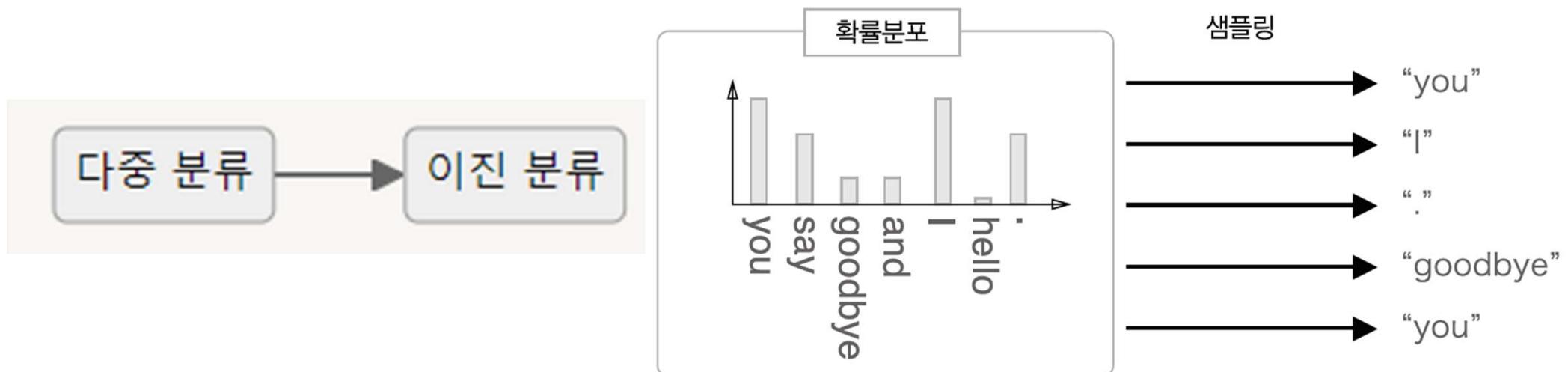
Word2vec 속도개선_4.2 두 번째 개선

- 해결 방법 : 네거티브 샘플링

계산 병목 (문제점)

- 은닉층과 가중치 행렬 W_{out} 곱의 계산량 이슈
- Softmax 계층에서 다루는 어휘가 많아짐에 따라 계산량 증가 이슈

그림 4-18 확률분포에 따라 샘플링을 여러 번 수행한다.



Word2vec 속도개선_4.2 두 번째 개선

- 네거티브 샘플링

핵심 아이디어 : 다중 분류를 이진 분류로 근사하여 계산량을 줄이자

기존 word2vec : 맥락이 주어졌을 때 정답이 되는 단어를 높은 확률로 추측하도록 만들게 설계함

Ex) you 와 goodbye를 주면 정답인 say의 확률이 높아지도록 학습시킴
즉, 다음과 같은 질문에 올바른 답을 제시

질문 : 맥락이 you와 goodbye일 때 타깃 단어는 무엇인가?

이진 분류로 해결한다면 어떻게 질문해야 할까?

Word2vec 속도개선_4.2 두 번째 개선

- 네거티브 샘플링

핵심 아이디어 : 다중 분류를 이진 분류로 근사하여 계산량을 줄이자

기존 word2vec : 맥락이 주어졌을 때 정답이 되는 단어를 높은 확률로 추측하도록 만들게 설계함

Ex) you 와 goodbye를 주면 정답인 say의 확률이 높아지도록 학습시킴
즉, 다음과 같은 질문에 올바른 답을 제시

질문 : 맥락이 you와 goodbye일 때 타깃 단어는 무엇인가?

이진 분류로 해결한다면 어떻게 질문해야 할지 생각해보자

Yes / No로 답할 수 있는 질문을 제시

Ex) 맥락이 you와 goodbye일 때 타깃 단어는 say 인가?

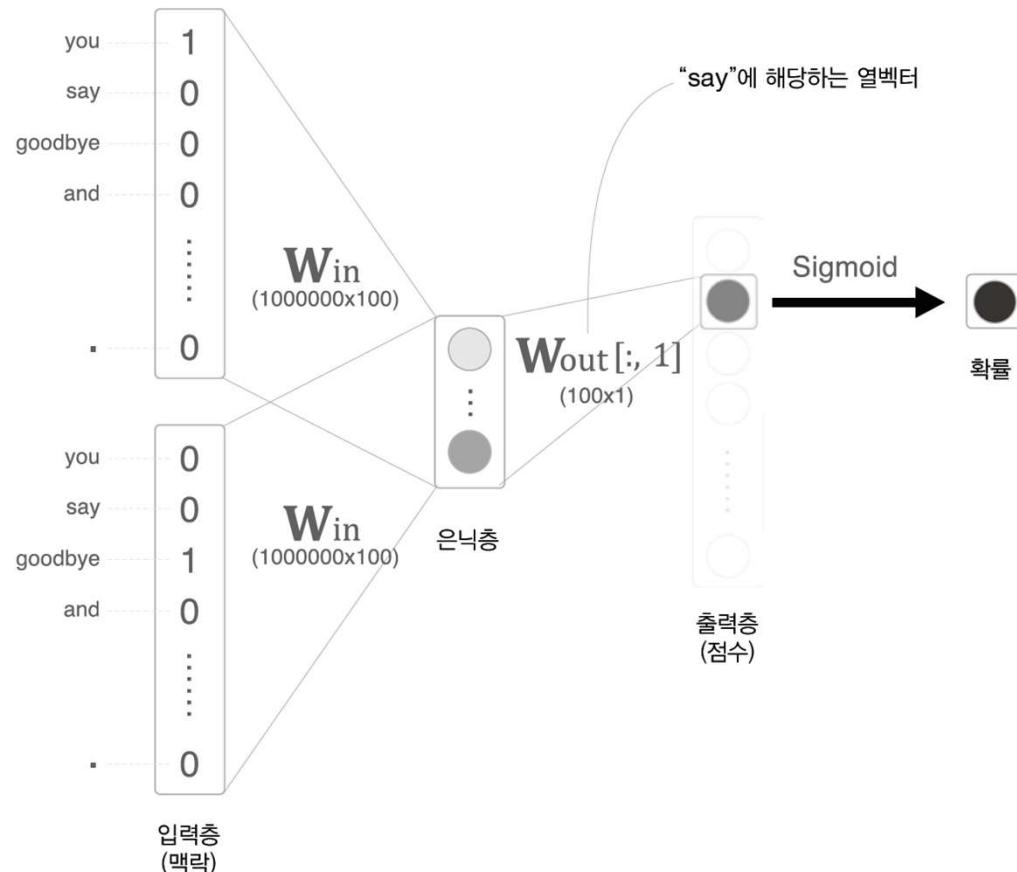
Word2vec 속도개선_4.2 두 번째 개선

- 네거티브 샘플링

Yes / No로 답할 수 있는 질문을 제시

Ex) 맥락이 you와 goodbye일 때 타깃 단어는 say 인가?

그림 4-7 타깃 단어만의 점수를 구하는 신경망



● 출력층 : 뉴런 1개
확률

은닉층과 출력 층 가중치 내적 :
say에 해당하는 단어 벡터만 추출
이 벡터와 은닉층 뉴런의 내적만 계산

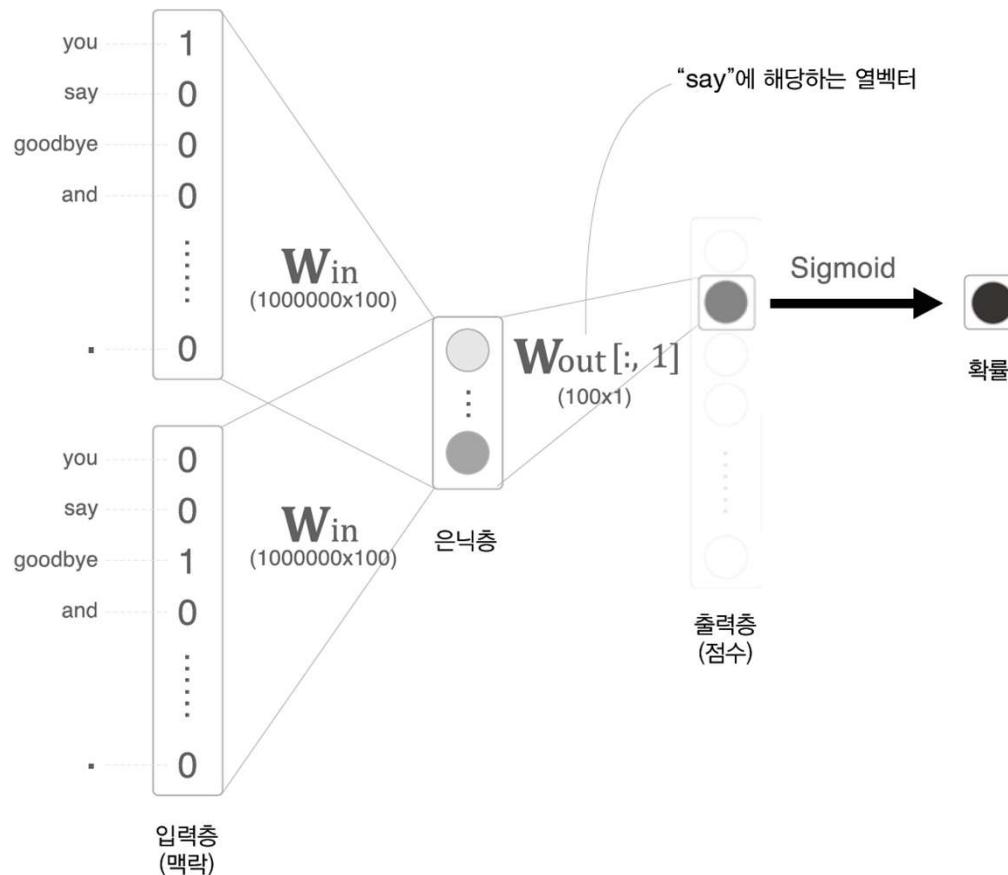
Word2vec 속도개선_4.2 두 번째 개선

- 이진 분류 문제 해결 신경망_점수 -> 확률 변환 (1장 내용)

다중 분류 출력층 : 소프트 맥스 함수,
이진 분류 출력층 : 시그모이드 함수,

Loss f.t : 교차 엔트로피 오차
Loss f.t : 교차 엔트로피 오차

그림 4-7 타깃 단어만의 점수를 구하는 신경망



Word2vec 속도개선_4.2 두 번째 개선

- 시그모이드 함수와 교차 엔트로피 오차 (1장 내용)

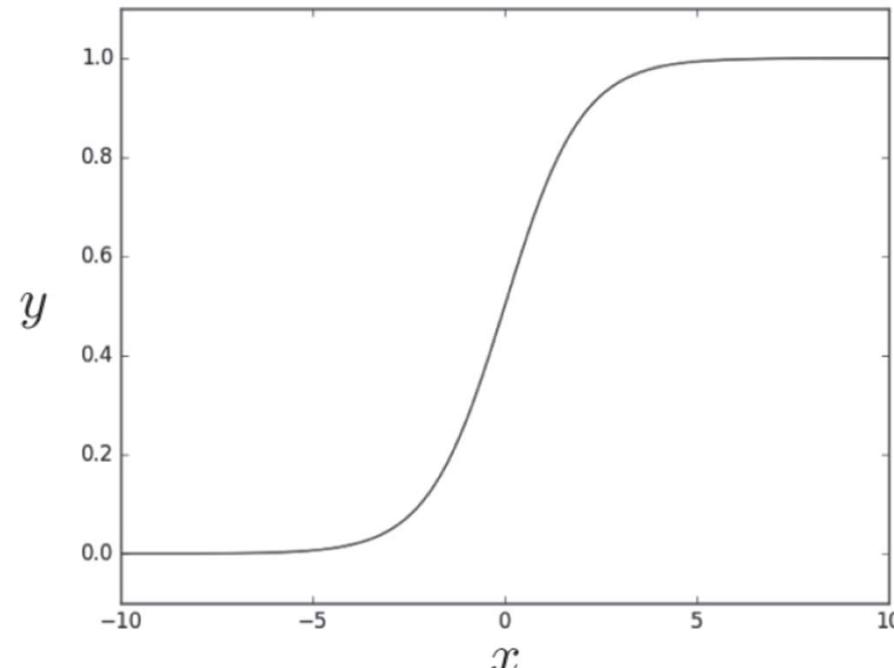
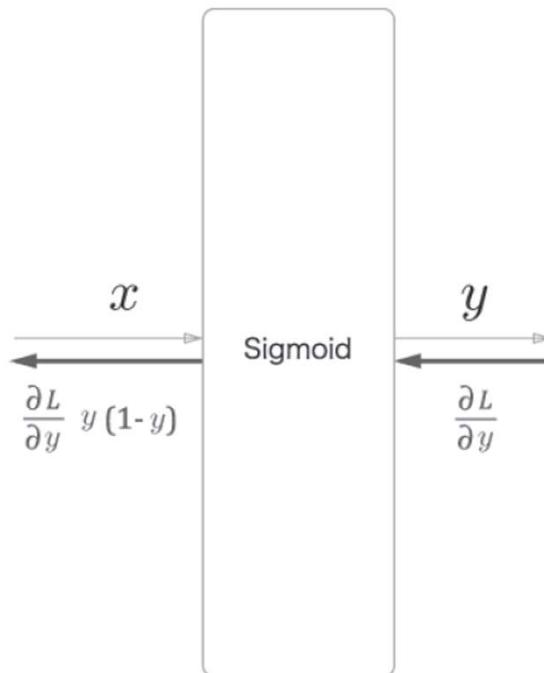
이진 분류 출력층 : 시그모이드 함수, Loss f.t : 교차 엔트로피 오차

시그모이드 함수 : $y = \frac{1}{1 + \exp(-x)}$

그래프 : S자 곡선

입력값 \rightarrow 0~1 사이 실수로 변환 \rightarrow y 값을 확률로 해석 가능하다.

그림 4-9 Sigmoid 계층(왼쪽)과 시그모이드 함수의 그래프(오른쪽)



Word2vec 속도개선_4.2 두 번째 개선

- 시그모이드 함수와 교차 엔트로피 오차 (1장 내용)

이진 분류 출력층 : 시그모이드 함수, Loss f.t : 교차 엔트로피 오차
시그모이드 함수 : $y = \frac{1}{1 + \exp(-x)}$

시그모이드 함수에 사용되는 손실 함수 : 교차 엔트로피 오차

교차 엔트로피 오차 : $L = -(t \log y + (1-t) \log(1-y))$

y는 시그모이드 함수의 출력

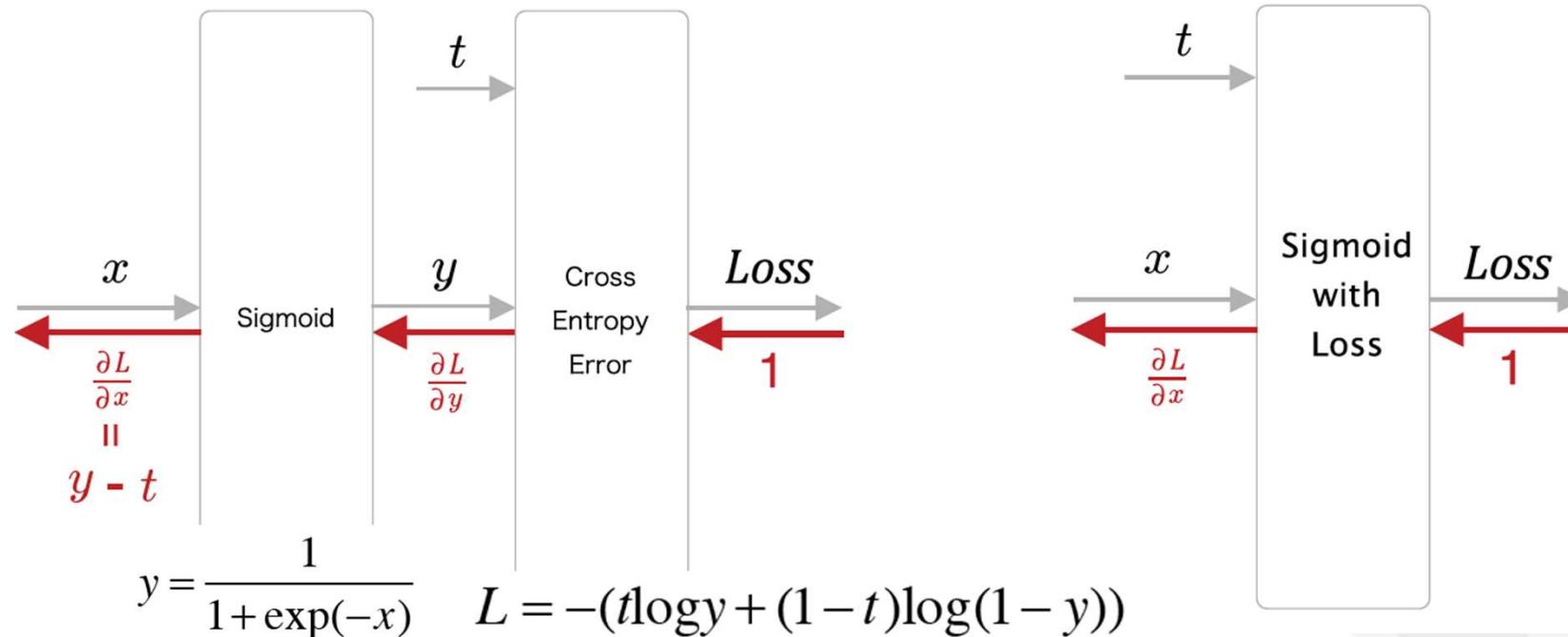
t는 정답 레이블 (0 or 1 by 이진 분류)

0 : No 1 : YES

Word2vec 속도개선_4.2 두 번째 개선

- 시그모이드 함수와 교차 엔트로피 오차 (1장 내용)
 핵심 : 역전파의 $y-t$ 값
 * y : 신경망이 출력한 확률 t : 정답 레이블
 $y-t$: 두 값의 차이

그림 4-10 Sigmoid 계층과 Cross Entropy Error 계층의 계산 그래프(오른쪽은 Sigmoid with Loss 계층으로 통합한 모습)



Word2vec 속도개선_4.2 두 번째 개선

- 시그모이드 함수와 교차 엔트로피 오차 (정답 추론)

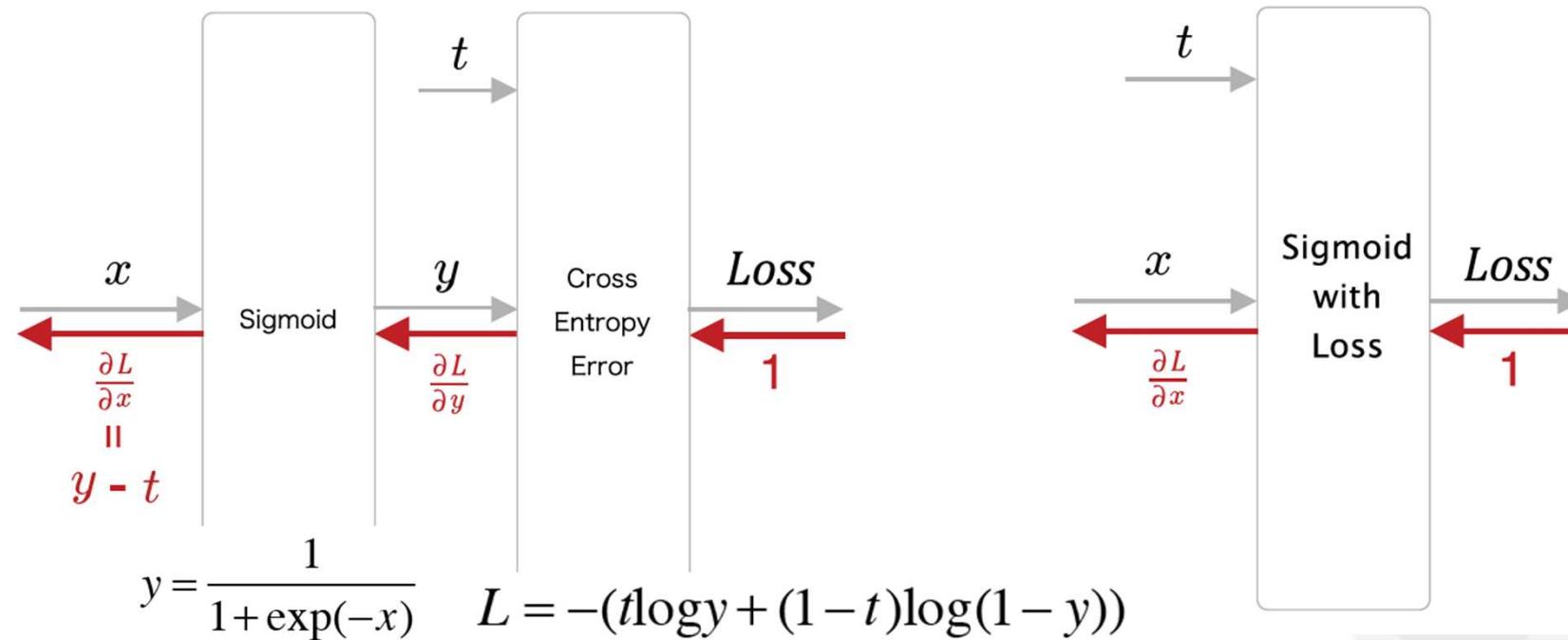
정답 레이블 1 \rightarrow y 가 1에 가까워지면 오차 감소

y 가 1로부터 멀어지면 오차 증가

오차가 크면 \rightarrow 기울기 크게 계산 \rightarrow 잘못된 예측을 빠르게 수정

오차가 작으면 \rightarrow 기울기 작게 계산 \rightarrow 작은 변화만 하며 세부적인 조정만

그림 4-10 Sigmoid 계층과 Cross Entropy Error 계층의 계산 그래프(오른쪽은 Sigmoid with Loss 계층으로 통합한 모습)

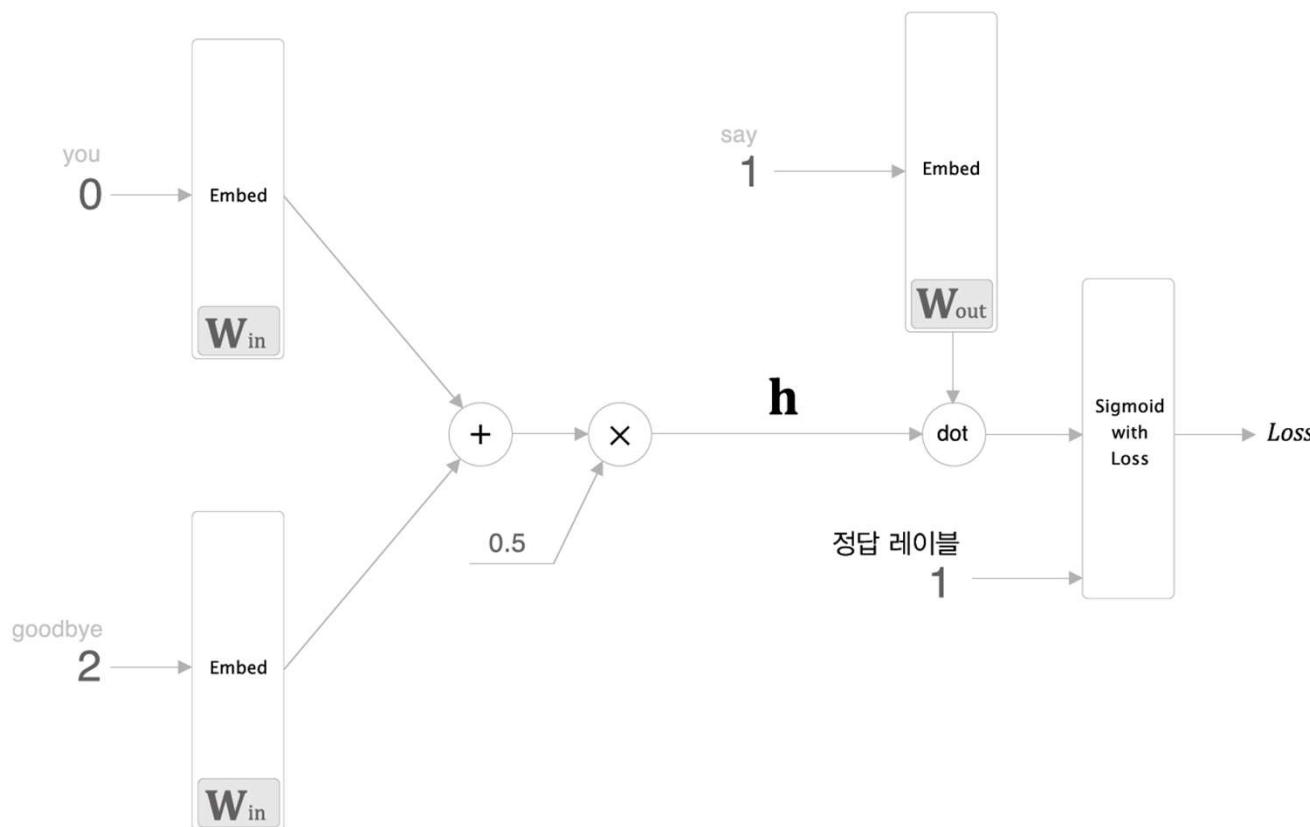


Word2vec 속도개선_4.2 두 번째 개선

- 다중 분류에서 이진 분류로 구현
 - 이 신경망에서는 은닉층 뉴런 h 와 출력 측의 가중치 W_{out} 에서 *Say*에 해당하는 단어 벡터와 내적을 계산한다.
 - 이 값을 Sigmoid with Loss 계층에 입력해 최종 손실을 얻는다.

그림 4-12 이진 분류를 수행하는 word2vec(CBOW 모델)의 전체 그림

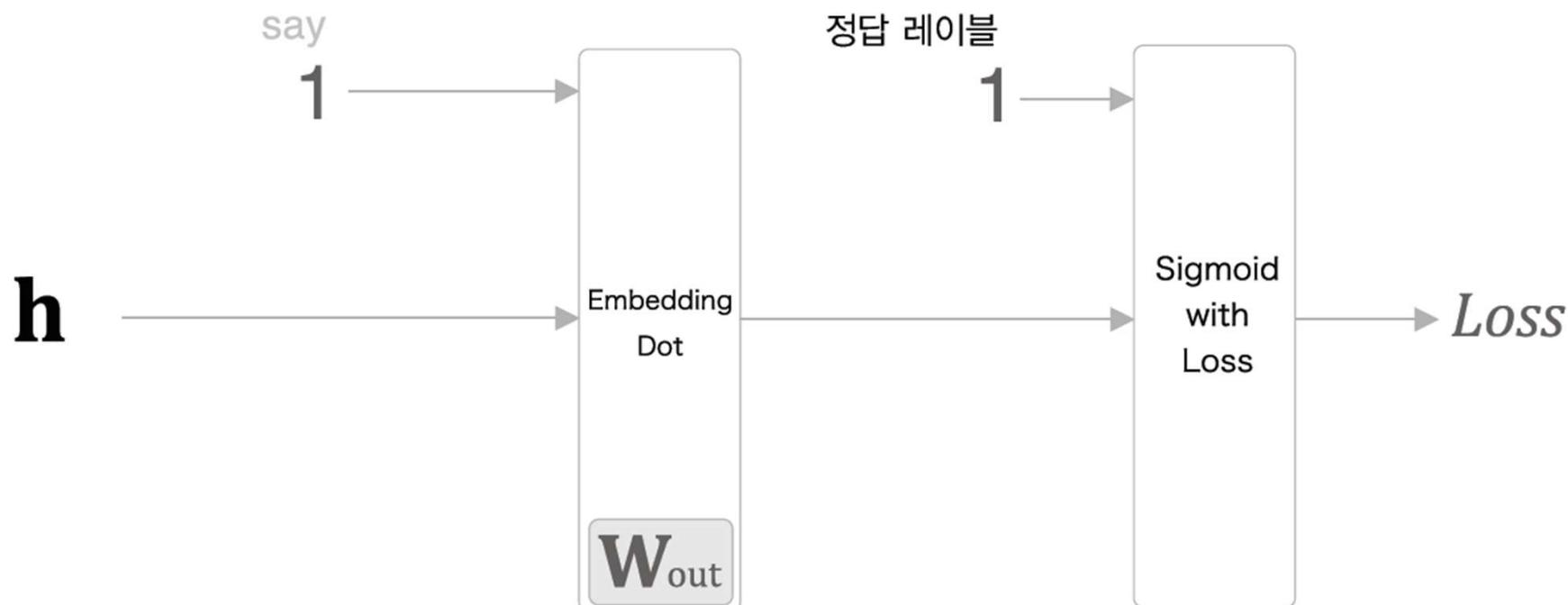
(이경우, 정답 레이블은 1)



Word2vec 속도개선_4.2 두 번째 개선

- 다중 분류에서 이진 분류로 구현
 - 이 신경망에서는 은닉층 뉴런 h 와 출력 측의 가중치 W_{out} 에서 Say 에 해당하는 단어 벡터와 내적을 계산한다.
- Embedding Dot : Embedding 계층과 Dot(연산) 처리를 합친 계층

그림 4-13 [그림 4-12]의 은닉층 이후 처리(Embedding Dot 계층을 사용하여 Embedding 계층과 내적 계산을 한 번에 수행)



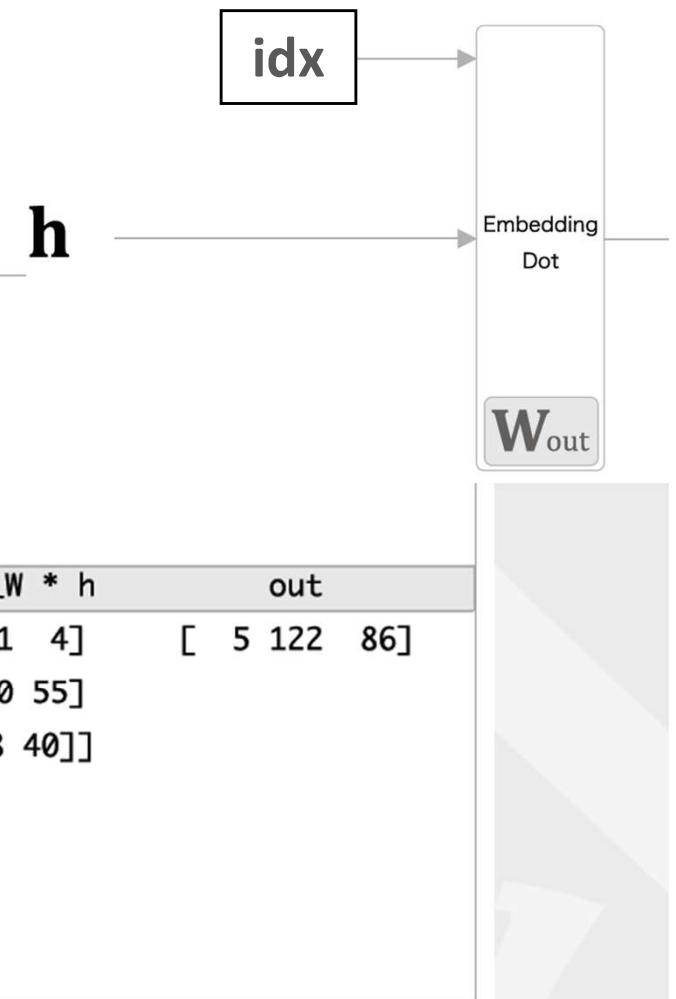
Word2vec 속도개선_4.2 두 번째 개선

- 다중 분류에서 이진 분류로 구현

Embedding Dot : Embedding 계층과 Dot(연산) 처리를 합친 계층

적당한 W, h
 Idx차원 : 미니배치 수
 Target_W : idx에 해당하는 것만 계산하기 위해, 해당 가중치만 선택

그림 4-14 Embedding Dot 계층의 각 변수의 구체적인 값



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

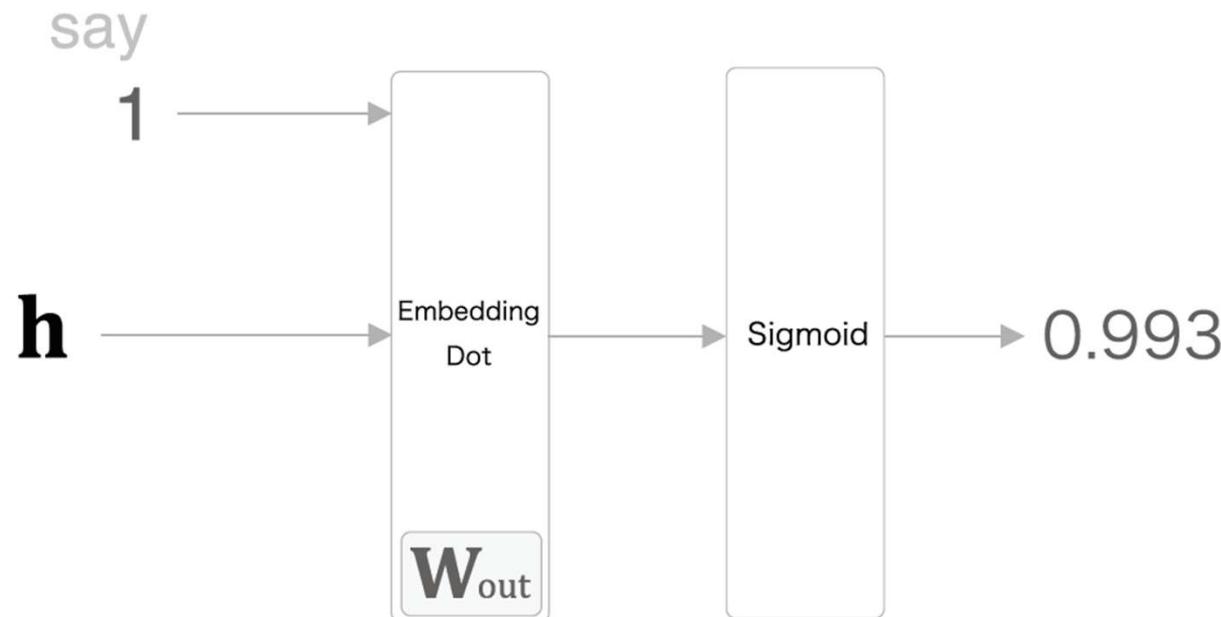
W	idx	target_W	h	target_W * h	out
[[0 1 2]	[0 3 1]	[[0 1 2]	[[0 1 2]	[[0 1 4]	[5 122 86]
[3 4 5]		[9 10 11]	[3 4 5]	[27 40 55]	
[6 7 8]		[3 4 5]]	[6 7 8]]	[18 28 40]]	
[9 10 11]					
[12 13 14]					
[15 16 17]					
[18 19 20]]					

Word2vec 속도개선_4.2 두 번째 개선

현재 다른 주제 : 다중 분류 문제를 이진 분류로 변환하는 과정 중,
정답(긍정)에 대해서만 다루었다.

오답(부정적인 예)를 입력하면 어떤 결과가 나올지 생각해보자.

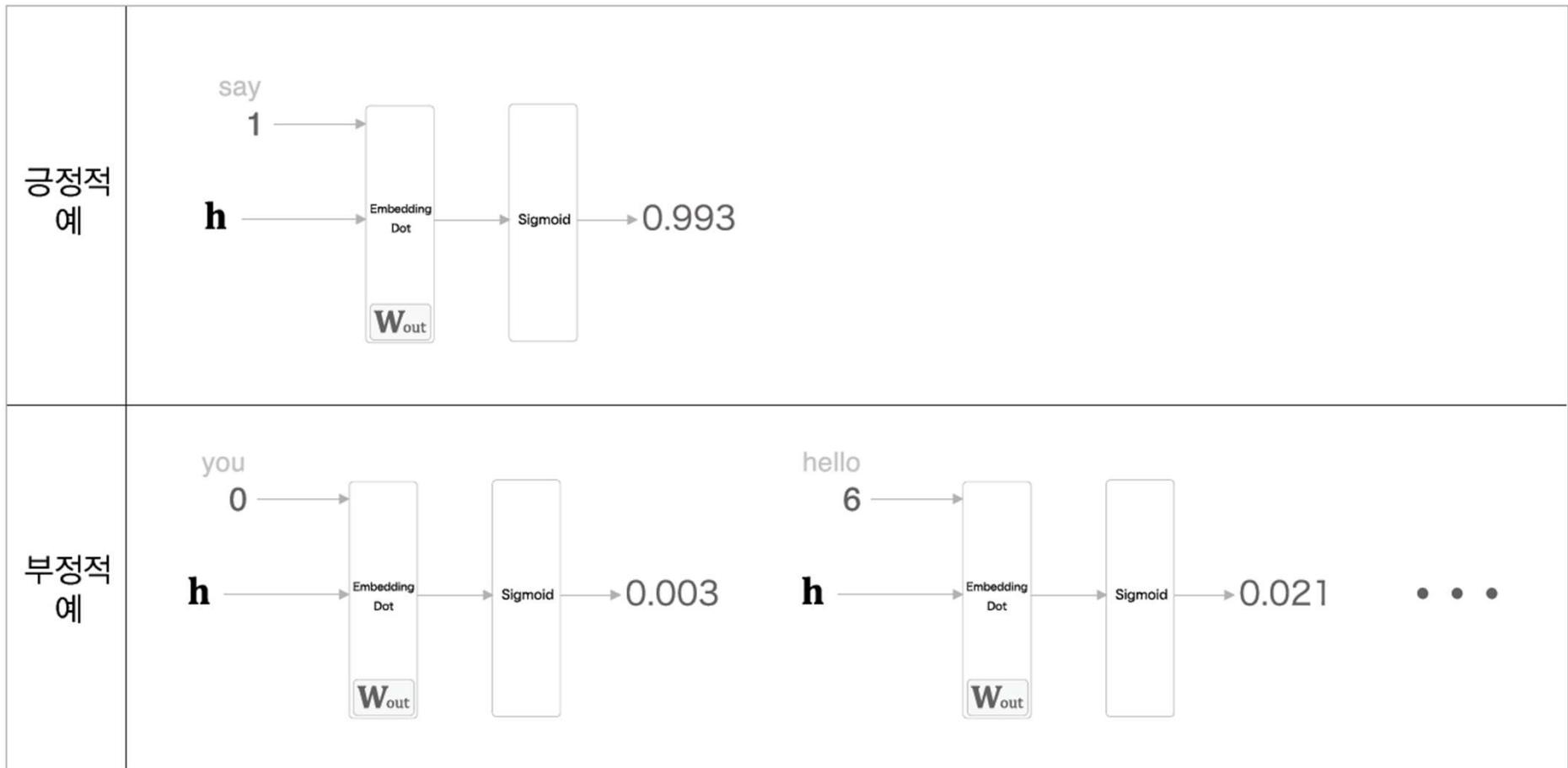
그림 4-15 CBOW 모델의 은닉층 이후의 처리 예: 맥락은 “you”와 “goodbye”이고, 타깃이 “say”일 확률은 0.993(99.3%)이다.



Word2vec 속도개선_4.2 두 번째 개선

오답(부정적인 예)를 입력하면 어떤 결과가 나올지 생각해보자.

그림 4-16 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 기중치가 필요하다.



Word2vec 속도개선_4.2 두 번째 개선

- 네거티브 샘플링

신경망 목표 :

~~정답에 가까운 긍정 예를 넣으면 Sigmoid를 1에 가깝게 출력
오답에 가까운 부정 예를 넣으면 Sigmoid를 0에 가깝게 출력
(부정 예 : 답 이외의 단어)~~

아이디어 : 모든 부정적 예를 대상으로 이진 분류 학습

Word2vec 속도개선_4.2 두 번째 개선

- 네거티브 샘플링

신경망 목표 :

~~정답에 가까운 긍정 예를 넣으면 Sigmoid를 1에 가깝게 출력
오답에 가까운 부정 예를 넣으면 Sigmoid를 0에 가깝게 출력
(부정 예 : 답 이외의 단어)~~

~~아이디어 : 모든 부정적 예를 대상으로 이진 분류 학습~~

어휘 수가 늘어나면 계산량 증가

이번 장 목표 : 어휘 수가 많아짐에 따라 발생되는 계산 병목 현상 개선

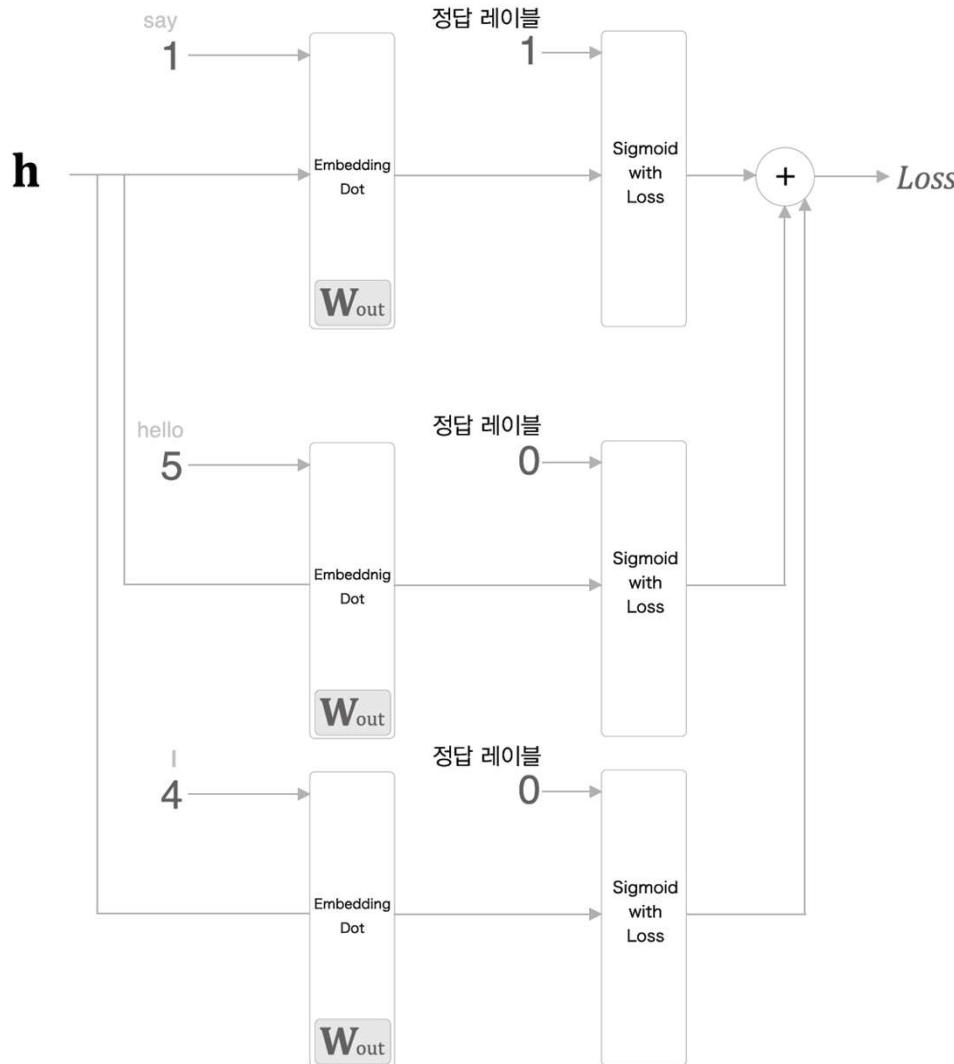
근사적 해법 : 부정적 예를 몇 개 선택하여 학습 시키기

= 네거티브 샘플링

Word2vec 속도개선_4.2 두 번째 개선

- 네거티브 샘플링

그림 4-17 네거티브 샘플링의 예(은닉층 이후의 처리에 주목하여 그린 계산 그래프)



EX) 긍정 타깃 : Say 1개
 → 정답레이블 1
 부정 타깃 : hello, I 2개
 → 정답레이블 0

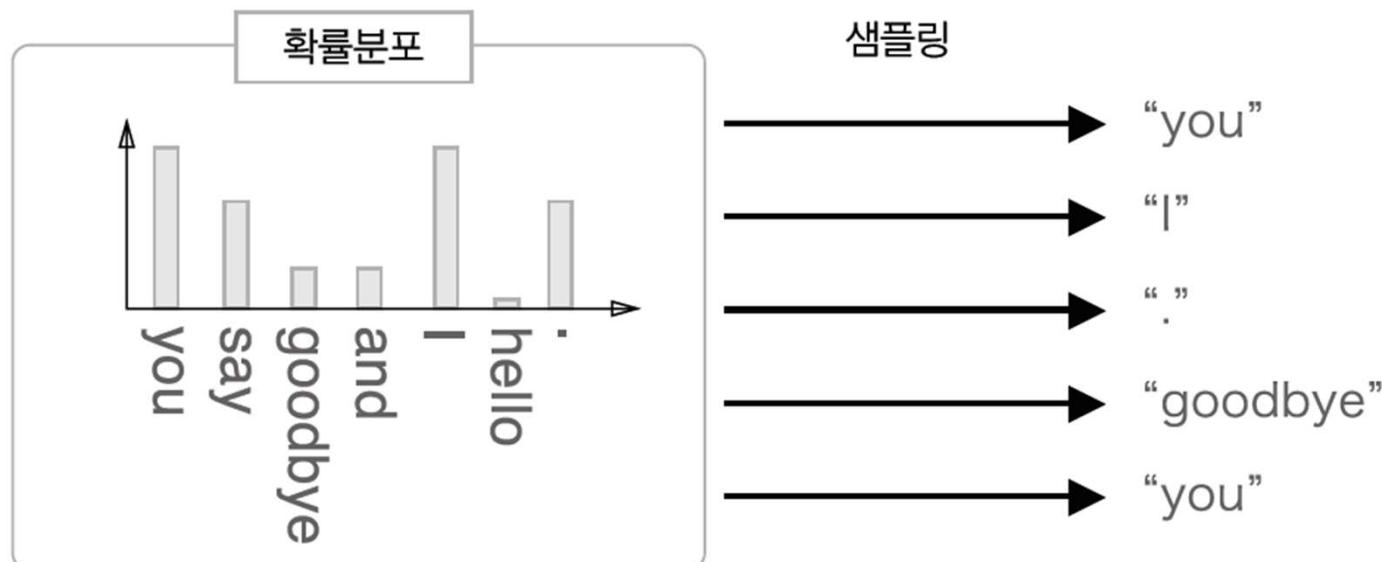
각 데이터의 손실을 모두 더해
 최종 손실 출력

Word2vec 속도개선_4.2 두 번째 개선

- 네거티브 샘플링의 샘플링 기법
무작위 샘플링 < 말뭉치 통계 데이터를 활용한 샘플링

즉, 말뭉치에서 자주 등장하는 단어를 많이 추출하고 드물게 등장하는 단어를 적게 추출한다

그림 4-18 확률분포에 따라 샘플링을 여러 번 수행한다.



Word2vec 속도개선_4.2 두 번째 개선

- 네거티브 샘플링의 샘플링 기법
말뭉치에서 단어 빈도를 기준으로 샘플링하기

- 각 단어의 출현 횟수를 구해 확률 분포로 나타낸다.
- 이 확률 분포에 따라 샘플링한다.

```
[1]: import numpy as np
[2]: words = ['you', 'say', 'goodbye', 'i', 'hello', '.']
      words에서 무작위 하나 지정
[3]: np.random.choice(words)
[3]: 'goodbye'
      5개만 무작위 샘플(중복가능)

[4]: np.random.choice(words, size = 5 )
[4]: array(['goodbye', 'hello', 'hello', 'hello', 'you'], dtype='<U7')
      5개만 무작위 샘플(중복불가)

[5]: np.random.choice(words, size=5, replace=False)
[5]: array(['.', 'say', 'hello', 'you', 'goodbye'], dtype='<U7')
      확률분포에 따라 샘플링

[6]: p = [0.5, 0.1, 0.05, 0.2, 0.05, 0.1]
[9]: np.random.choice(words, p=p)
[9]: 'you'
```

Word2vec 속도개선_4.2 두 번째 개선

- 네거티브 샘플링의 샘플링 기법

권고사항 : 확률분포에 0.75 제곱하기

이유 : 출현 확률이 낮은 단어를 버리지 않기 위해, 0.75를 제곱함으로써 원래 확률이 낮은 단어의 확률을 약간 높일 수 있다.

```
[1]: import numpy as np
[2]: p = [0.7, 0.29, 0.01]
[3]: new_p = np.power(p, 0.75)
[4]: new_p /= np.sum(new_p)
[6]: print(new_p)
[0.64196878 0.33150408 0.02652714]
```

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

원래 0.01이던 원소가 수정후에는 0.26으로 높아졌다.

Word2vec 속도개선_4.2 두 번째 개선

- 네거티브 샘플링 구현

```

class NegativeSamplingLoss: 4개의 사용 위치
    def __init__(self, W, corpus, power=0.75, sample_size=5):
        self.sample_size = sample_size
        self.sampler = UnigramSampler(corpus, power, sample_size)
        self.loss_layers = [SigmoidWithLoss() for _ in range(sample_size + 1)]
        self.embed_dot_layers = [EmbeddingDot(W) for _ in range(sample_size + 1)]

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

    def forward(self, h, target):
        batch_size = target.shape[0]
        negative_sample = self.sampler.get_negative_sample(target)

        # 긍정적 예 순전파
        score = self.embed_dot_layers[0].forward(h, target)
        correct_label = np.ones(batch_size, dtype=np.int32)
        loss = self.loss_layers[0].forward(score, correct_label)

        # 부정적 예 순전파
        negative_label = np.zeros(batch_size, dtype=np.int32)
        for i in range(self.sample_size):
            negative_target = negative_sample[:, i]
            score = self.embed_dot_layers[1 + i].forward(h, negative_target)
            loss += self.loss_layers[1 + i].forward(score, negative_label)
            손실이 아니라 loss를 모두 더한다.

        return loss

    def backward(self, dout=1):
        dh = 0          예전의 알고리즘은 각 계층의 역전파.
        for l0, l1 in zip(self.loss_layers, self.embed_dot_layers):
            dscore = l0.backward(dout)
            dh += l1.backward(dscore)
            기록되었다. 1.34 Repeate

        return dh

```

Word2vec 속도개선_4.3 개선판 word2vec 학습

- 개선판 Word2vec 학습
기존 word2vec 개선한 것 정리

1. Embedding 계층 도입

Embedding 계층 : 가중치 매개변수로부터 단어 ID에 해당하는 행을 추출하는 계층

2. 네거티브 샘플링 도입

다중 분류를 이진 분류로 전환(긍정 , 부정)

긍정적 예는 1개만 샘플링_ Yes_1 학습

부정적 예는 확률 분포에 따라 몇 개만 샘플링_NO_0 학습

이 개선을 신경망 구현에 적용하고, PTB 데이터셋을 사용해 학습하고 더 실용적인 단어의 분산 표현을 얻어보자

Word2vec 속도개선_4.3 개선판 word2vec 학습

- CBOW 모델 구현
- Embedding 계층 도입
 - 네거티브 샘플링 도입

그림 4-19 맥락과 타깃을 단어 ID로 나타낸 예(맥락의 윈도우 크기는 1)

맥락	타깃	맥락	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 →

```

class CBOW: 2개의 사용 위치 어캐수  흔적 뉴런수  맥락크기  단어ID목록
    def __init__(self, vocab_size, hidden_size, window_size, corpus):
        V, H = vocab_size, hidden_size

        # 가중치 초기화
        W_in = 0.01 * np.random.randn(V, H).astype('f')
        W_out = 0.01 * np.random.randn(V, H).astype('f')

        # 계층 생성
        self.in_layers = []
        for i in range(2 * window_size):
            layer = Embedding(W_in) # Embedding 계층 사용
            self.in_layers.append(layer)
        self.ns_loss = NegativeSamplingLoss(W_out, corpus, power=0.75, sample_size=5)

        # 모든 가중치와 기울기를 배열에 모은다.
        layers = self.in_layers + [self.ns_loss]
        self.params, self.grads = [], []
        for layer in layers:
            self.params += layer.params
            self.grads += layer.grads

        (# 인스턴스 변수에 단어의 분산 표현을 저장한다.)
        self.word_vecs = W_in

        맥락  타깃  ~ 앞과 다를게, ID를 인수로 한다.
        ↗(자신빼면)  ↗(자신빼면)  (앞에는 단어ID를 첫 번째 빼면 뒤에 사용)
    def forward(self, contexts, target):
        h = 0
        for i, layer in enumerate(self.in_layers):
            h += layer.forward(contexts[:, i])
        h *= 1 / len(self.in_layers)
        loss = self.ns_loss.forward(h, target)
        return loss

    def backward(self, dout=1):
        dout = self.ns_loss.backward(dout)
        dout *= 1 / len(self.in_layers)
        for layer in self.in_layers:
            layer.backward(dout)
        return None

```

Word2vec 속도개선_4.3 개선판 word2vec 학습

- CBOW 모델 학습코드

```

# 하이퍼파라미터 설정
window_size = 5
hidden_size = 100
batch_size = 100
max_epoch = 10

# 데이터 읽기
corpus, word_to_id, id_to_word = ptb.load_data('train')
vocab_size = len(word_to_id)

contexts, target = create_contexts_target(corpus, window_size)
if config.GPU:
    contexts, target = to_gpu(contexts), to_gpu(target)

# 모델 등 생성
model = CBOW(vocab_size, hidden_size, window_size, corpus)
# model = SkipGram(vocab_size, hidden_size, window_size, corpus)
optimizer = Adam()
trainer = Trainer(model, optimizer)

# 학습 시작
trainer.fit(contexts, target, max_epoch, batch_size)
trainer.plot()

# 나중에 사용할 수 있도록 필요한 데이터 저장
word_vecs = model.word_vecs
if config.GPU:
    word_vecs = to_cpu(word_vecs)
params = {}
params['word_vecs'] = word_vecs.astype(np.float16)
params['word_to_id'] = word_to_id
params['id_to_word'] = id_to_word
pkl_file = 'cbow_params.pkl'
with open(pkl_file, 'wb') as f:
    pickle.dump(params, f, -1)

```

Word2vec 속도개선_4.3 개선판 word2vec 학습

- CBOW 모델 평가_단어 분산 표현_유사도

CBOW 모델을 학습시키고 난 후
단어의 분산 표현을 평가해보자

Most_similar()를 이용하여
지정 단어와 거리가 가장 가까운
단어들을 선별해보자
(2장에서 구현한 것 이용)

```
# coding: utf-8
import sys
sys.path.append('..')
from common.util import most_similar, analogy
import pickle

pkl_file = 'cbow_params.pkl'
# pkl_file = 'skipgram_params.pkl'

with open(pkl_file, 'rb') as f:
    params = pickle.load(f)
    word_vecs = params['word_vecs']
    word_to_id = params['word_to_id']
    id_to_word = params['id_to_word']

    # 가장 비슷한(most similar) 단어 뽑기
    querys = ['you', 'year', 'car', 'toyota']
    for query in querys:
        most_similar(query, word_to_id, id_to_word, word_vecs, top=5)
```

Word2vec 속도개선_4.3 개선판 word2vec 학습

- CBOW 모델 평가_단어 분산 표현_유사도
CBOW 모델을 학습시키고 난 후
단어의 분산 표현을 평가해보자

You : 인칭대명사 출력

Year : 기간 뜻하는 성격 단어 출력

Car : 자동차와 연관된 단어 출력

Toyota : 자동차 메이커

```
[query] you
we: 0.6103515625
someone: 0.59130859375
i: 0.55419921875
something: 0.48974609375
anyone: 0.47314453125
```

```
[query] year
month: 0.71875
week: 0.65234375
spring: 0.62744140625
summer: 0.6259765625
decade: 0.603515625
```

```
[query] car
luxury: 0.497314453125
arabia: 0.47802734375
auto: 0.47119140625
disk-drive: 0.450927734375
travel: 0.4091796875
```

```
[query] toyota
ford: 0.55078125
instrumentation: 0.509765625
mazda: 0.49365234375
bethlehem: 0.47509765625
nissan: 0.474853515625
```

Word2vec 속도개선_4.3 개선판 word2vec 학습

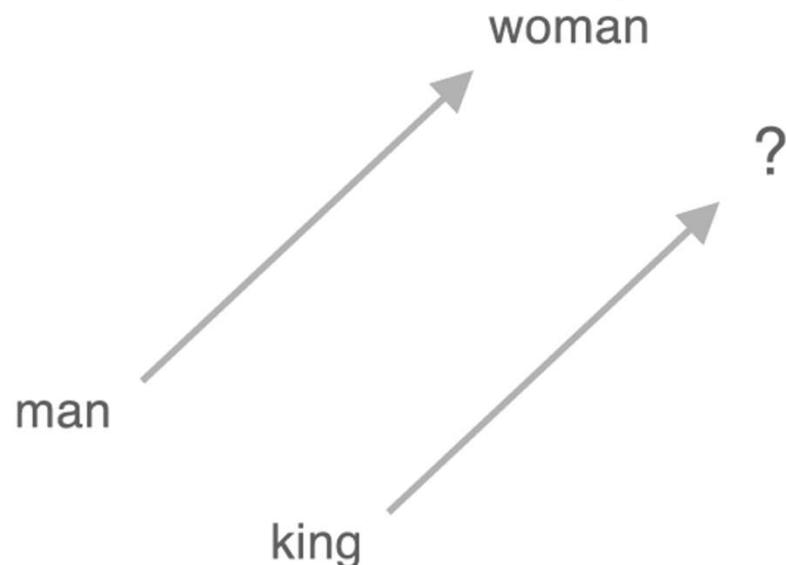
- CBOW 모델 평가_유추 문제

Word2vec으로 얻은 단어의 분산표현은 비슷한 단어를 가까이 모을 뿐만 아니라, 더 복잡한 패턴을 파악할 수 있다.

EX) 유추 문제를 풀 수 있다.

King – man + woman = queen 의 문제를 풀 수 있다.

그림 4-20 “man : woman = king : ?” 유추 문제 풀기(단어 벡터 공간에서 각 단어의 관계성)



Word2vec 속도개선_4.3 개선판 word2vec 학습

- CBOW 모델 평가_유추 문제

표현 가정 :

"vec ('man')" 단어 "man"의
분산 표현(단어 벡터)

관계를 수식으로 변환 :

"vec ('woman') - vec ('man') =
vec (?) - vec ('king')"

풀어야 하는 문제 :

"vec ('king') + vec ('woman')
-vec ('man') = vec (?)"라는 벡터에
가장 가까운 단어 벡터 구하기

```

import sys
sys.path.append('..')
from common.util import most_similar, analogy
import pickle

pkl_file = 'cbow_params.pkl'
# pkl_file = 'skipgram_params.pkl'

with open(pkl_file, 'rb') as f:
    params = pickle.load(f)
    word_vecs = params['word_vecs']
    word_to_id = params['word_to_id']
    id_to_word = params['id_to_word']

# 가장 비슷한(most similar) 단어 뽑기
querys = ['you', 'year', 'car', 'toyota']
for query in querys:
    most_similar(query, word_to_id, id_to_word, word_vecs, top=5)

# 유추(analogy) 작업
print('*'*50)
analogy(a: 'king', b: 'man', c: 'queen', word_to_id, id_to_word, word_vecs)
analogy(a: 'take', b: 'took', c: 'go', word_to_id, id_to_word, word_vecs)
analogy(a: 'car', b: 'cars', c: 'child', word_to_id, id_to_word, word_vecs)
analogy(a: 'good', b: 'better', c: 'bad', word_to_id, id_to_word, word_vecs)

```

Word2vec 속도개선_4.3 개선판 word2vec 학습

- CBOW 모델 평가_유추 문제

→ 현재형과 과거형 패턴을 파악하고 있다
 : 시제 정보가 단어의 분산 표현에 인코딩

→ 단수형과 복수형을 파악하고 있다.

→ more less 등의 비교급 단어를 제시
 : 비교급이라는 성질도 단어의 분산표현에
 인코딩 되어있음을 알 수 있다.

```
[analogy] king:man = queen:?
woman: 5.16015625
veto: 4.9296875
ounce: 4.69140625
earthquake: 4.6328125
successor: 4.609375
```

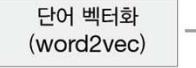
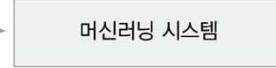
```
[analogy] take:took = go:?
went: 4.55078125
points: 4.25
began: 4.09375
comes: 3.98046875
oct.: 3.90625
```

```
[analogy] car:cars = child:?
children: 5.21875
average: 4.7265625
yield: 4.20703125
cattle: 4.1875
priced: 4.1796875
```

```
[analogy] good:better = bad:?
more: 6.6484375
less: 6.0625
rather: 5.21875
slower: 4.734375
greater: 4.671875
```

Word2vec 속도개선_4.4 Word2vec 남은 주제

그림 4-22 메일 자동 분류 시스템의 예(감정 분석)

- Word2vec을 사용한 애플리케이션 예  →  →  →   

사용자가 1억명 이상인 스마트폰 앱 개발, 운영하고 있을 때 고객의 소리함을 메일로 받는다고 가정

사용자의 메일을 사용자의 감정의 단계를 나누어 분류하고 싶을때, word2vec을 사용 가능하다.

불만을 가진 사용자의 메일부터 순서대로 보면 앱의 치명적인 문제를 조기에 발견하고 손을 쓸 수 있고, 만족도도 올라갈 것이다.

사용자가 보낸 메일을 모으고, 모은 메일에 수동으로 레이블을 다음과 같이 붙인다. 긍정/중립/부정

레이블링 작업이 끝나면 word2vec을 이용해 메일을 벡터로 변환시킨다.

그런 다음 감정 분석을 수행하는 어떤 분류 시스템(SVM, 신경망)에 벡터화된 메일과 감정 레이블을 입력하여 학습을 시킬 수 있다

Word2vec 속도개선_4.4 Word2vec 남은 주제

- 단어 벡터 평가 방법

word2vec을 통해 단어의 분산 표현을 얻을 수 있다.
그 분산 표현이 좋은지 어떻게 평가할까?

분산 표현 평가에 사용되는 평가 척도

1. 단어의 유사성 평가

사람이 작성한 단어 유사도를 검증 세트를 사용해 평가하는 것이 일반적이다.
사람이 부여한 점수와 word2vec에 의한 코사인 유사도 점수를 비교해 그 상관성을 보는 것이다.

EX) 유사도를 0에서 10에서 점수화 한다면,
cat과 animal의 유사도 8점 / cat 과 car의 유사도는 2점
과 같이, 사람이 단어 사이의 유사한 정도를 규정.
이후 word2vec에 의한 코사인 유사도 점수 비교

Word2vec 속도개선_4.4 Word2vec 남은 주제

- 단어 벡터 평가 방법

word2vec을 통해 단어의 분산 표현을 얻을 수 있다.
그 분산 표현이 좋은지 어떻게 평가할까?

분산 표현 평가에 사용되는 평가 척도

2. 유추 문제를 활용한 평가

king : queen = man : ? 와 같은 유추 문제를 출제하고, 그 정답률로 단어의 분산 표현이나 우수성을 측정한다.

유추 문제를 이용하면 단어의 의미나 문법적인 문제를 제대로 이해하고 있는지 어느정도 측정 가능하다.

그러므로 유추 문제를 정확하게 풀 수 있는 단어의 분산 표현이라면 자연어를 다루는 애플리케이션에서도 좋은 결과를 기대할 수 있을 것이다.

Word2vec 속도개선_4.5 정리

- word2vec 고속화를 주제로 앞 장의 CBOW 모델을 개선했다.

개선 방법 :

1. Embedding 계층 구현
2. 네거티브 샘플링 기법 도입

배경 : 말뭉치 어휘 수 증가에 비례해 계산량이 증가하는 문제

핵심 : 모두 대신 일부를 처리하는 것

Embedding 계층은 필요한 단어의 표현만 골라 쓸 수 있게 해준다.

네거티브 샘플링은 모든 단어가 아닌 일부 단어만을 대상으로 하는 것으로,
계산을 효율적으로 수행해준다

