

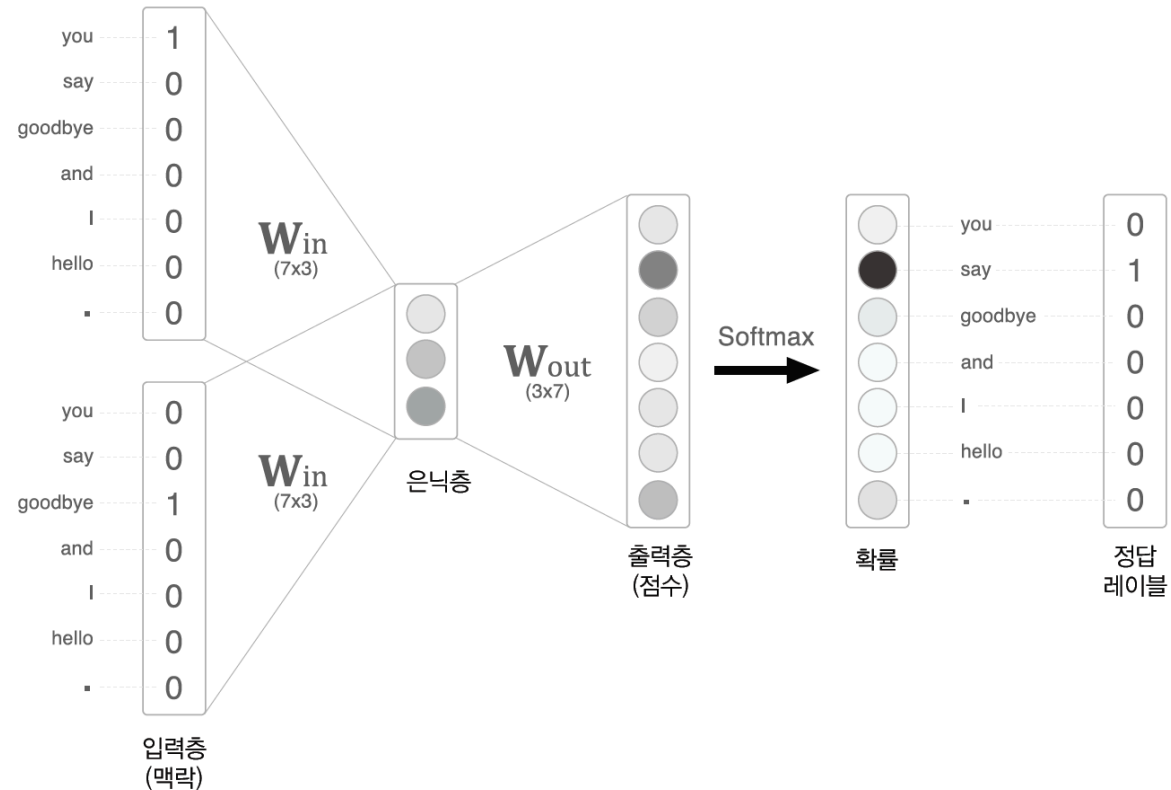
밑바닥부터 시작하는 딥러닝 2: 4장 word2vec 속도 개선

2020/02/18

4. word2vec 속도 개선

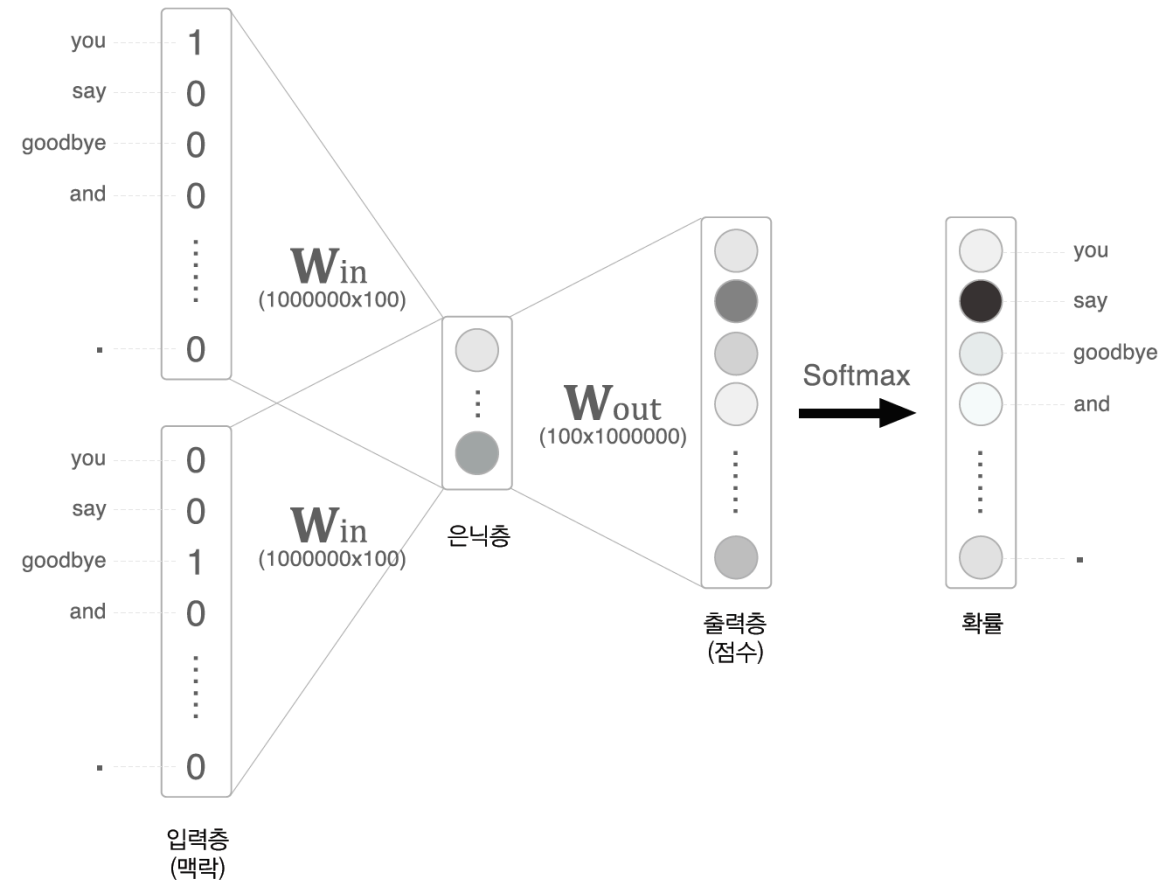
- CBOW 모델의 문제점
 - 말뭉치에 포함된 어휘 수가 많아지면 계산량도 커짐
 - word2vec의 속도를 개선하자.
- Embedding 레이어와 네거티브 샘플링 도입
 - PTB 데이터셋으로 학습을 수행

4.1 word2vec 개선 (1)



- 맥락: 단어 2개, 타깃: 단어 1개
- 각 가중치(W_{in} , W_{in})와의 행렬 곱으로 단어의 점수를 구하고, 점수에 소프트맥스 함수를 적용해 각 단어의 출현 확률을 얻고, 정답과 비교해 손실을 구함.

4.1 word2vec 개선 (1)



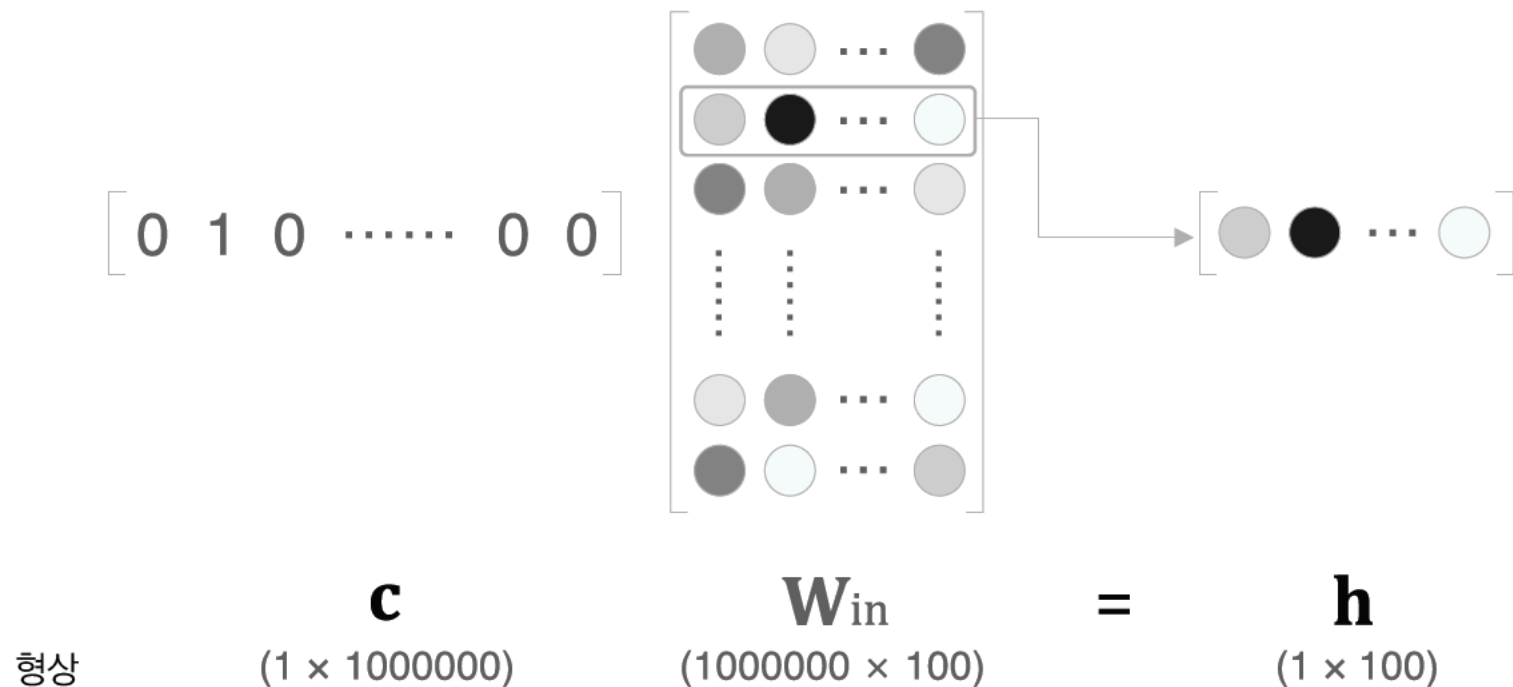
- 만약 입력층과 출력층에 각 100만개의 뉴런이 존재하게 된다면 중간에 많은 계산 시간이 소요
 - 입력층의 원핫 표현과 W_{in} 의 곱 계산
 - 은닉층과 W_{out} 의 곱 및 Softmax 계층의 계산

4.1 word2vec 개선 (1)

- 입력층의 원핫 표현과 W_{in} 의 곱 계산
 - 단어를 원핫 표현으로 바꿀 때 어휘 수가 많아지면 원핫 표현의 벡터 크기도 커지게 됨
 - 어휘가 100만개 -> 원핫 표현 하나만 해도 원소 수가 100만 개인 벡터
 - 또한 이 원핫 벡터와 W_{in} 를 곱하는데도 계산량이 많아짐.
 - Embedding 레이어를 도입해 해결
- 은닉층과 W_{out} 의 곱 및 Softmax 계층의 계산
 - 은닉층과 W_{out} 를 곱하는데 계산량이 상당함.
 - Softmax 계층에서도 다루는 어휘가 많아져 계산량이 증가.
 - 네거티브 샘플링이라는 새로운 loss 함수를 도입해 해결

4.1.1 Embedding 계층

- 어휘 수가 100만개, 은닉층 뉴런이 100개인 경우:



- 100만 차원의 원핫 벡터와 가중치 행렬과의 곱에서 결과적으로 수행하는 일은 단지 행렬의 특정 행을 추출
- 가중치 매개변수로부터 단어 ID에 해당하는 행(벡터)를 추출하는 **Embedding 계층**을 사용
 - Embedding 계층에 단어 임베딩(분산표현)을 저장

4.1.2 Embedding 계층 구현

- 행렬에서 특정 행 추출
 - 원하는 행 명시

```
# 4.1.2 Embedding 계층 구현  
W = np.arange(21).reshape(7, 3)
```

W

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

W[2]

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

W[4]

```
array([12, 13, 14])
```

- 여러 행을 한꺼번에 추출
 - 원하는 행 번호들을 배열에 명시
(mini-batch라고 가정했을 때)

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

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

4.1.2 Embedding 계층 구현

```
class Embedding:
    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

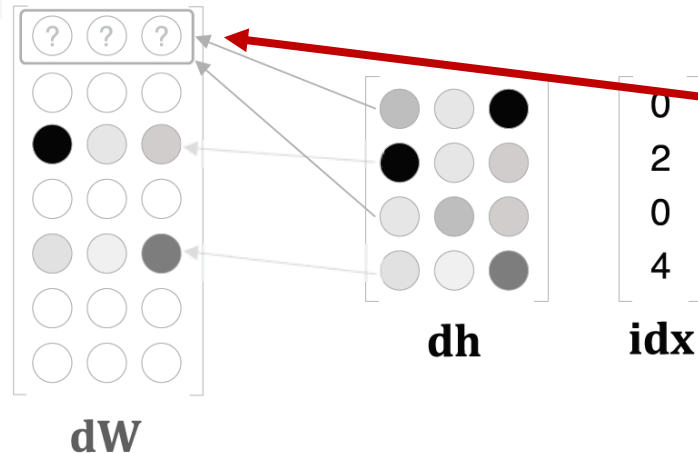
    def backward(self, dout):
        dW, = self.grads
        dW[...] = 0
        dW[self.idx] = dout # 문제 발생

        return None
```

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

• 역전파:

- 순전파 -> 가중치 w 의 특정 행을 추출하는 역할
 - 단순히 가중치의 특정 행 뉴런만을 다음 층으로 전달
- 역전파에서도 앞 층으로부터 전해진 기울기를 다음 층으로 그대로 전달 해주면 됨.
 - 앞 층으로부터 전해진 기울기를 가중치 기울기 dW 의 idx 번째 행에 설정
 - 가중치 기울기 dW 를 꺼내 dW 의 형상은 그대로 유지한 채 원소들을 초기화
 - 앞 층에서 전달 된 기울기 $dout$ 을 idx 번째 행에 전달

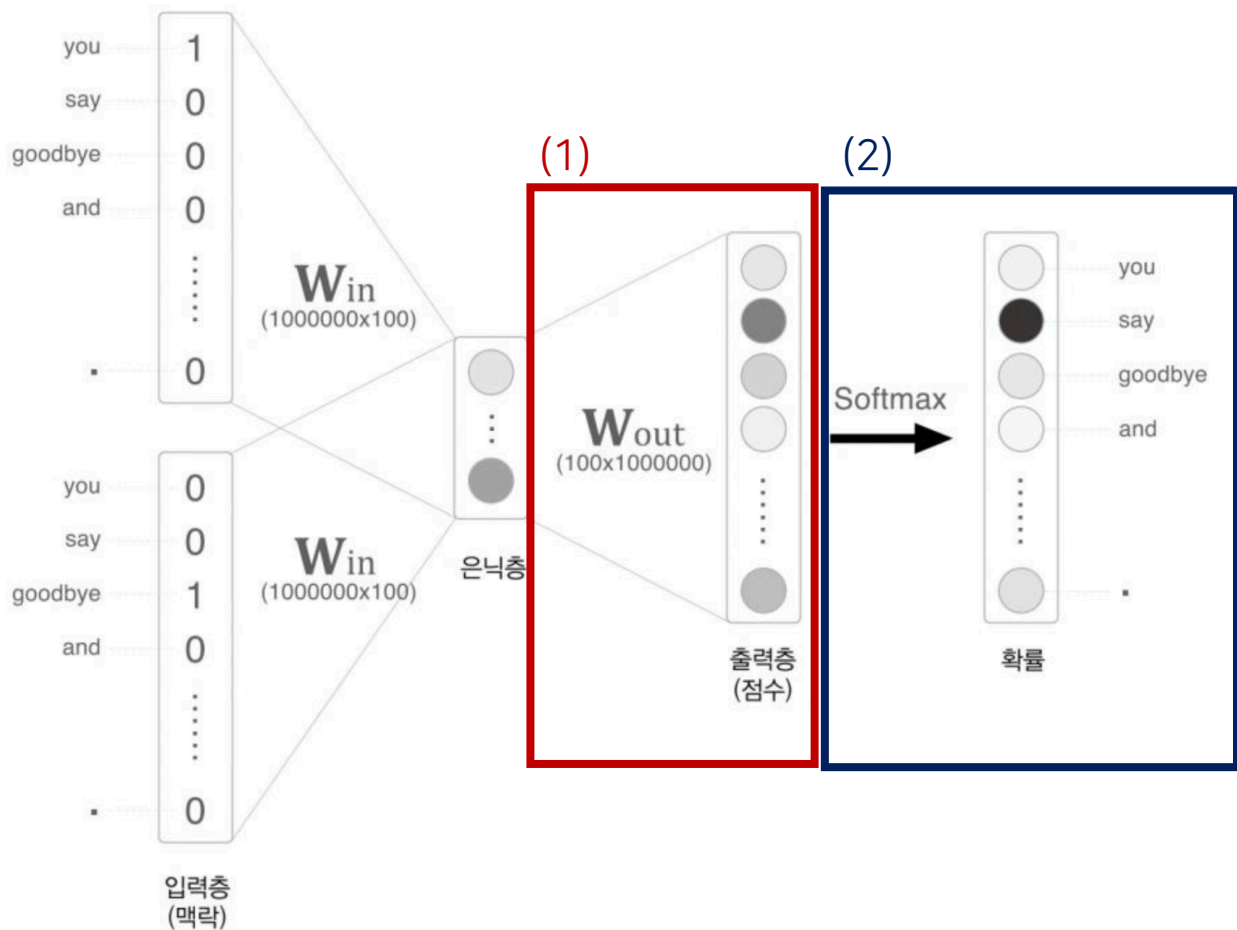


- 하지만 만약 idx 의 원소가 중복되는 경우에는?
 - idx 가 [0, 2, 0, 4]인 경우, 먼저 쓰여진 값을 덮어씀.
 - 할당이 아닌 더하기를 하자.
 - 각 기울기 값을 해당 idx 의 dW 에 더해준다.
 - `np.add.at(A, idx, B)`: B를 A의 idx 번째 행에 더해줌

4.2 word2vec 개선 (2)

- hidden 레이어 이후의 행렬 곱과 Softmax 계층의 계산량 문제
- Softmax 대신 **네거티브 샘플링**을 이용하면 어휘가 많아져도 계산량을 낮은 수준에서 일정하게 억제 가능

4.2.1 은닉층 이후 계산의 문제점



- 어휘가 100만개, 은닉층 뉴런이 100개, word2vec (CBOW) 모델

(1) 은닉층의 벡터 크기 100, 가중치 행렬 크기 100 * 100만

- 계산에 오랜 시간이 걸릴 뿐만 아니라 역전파때도 같은 계산을 수행하기 때문에 이 행렬 곱을 가볍게 만드는게 관건

(2) Softmax의 계산량 증가

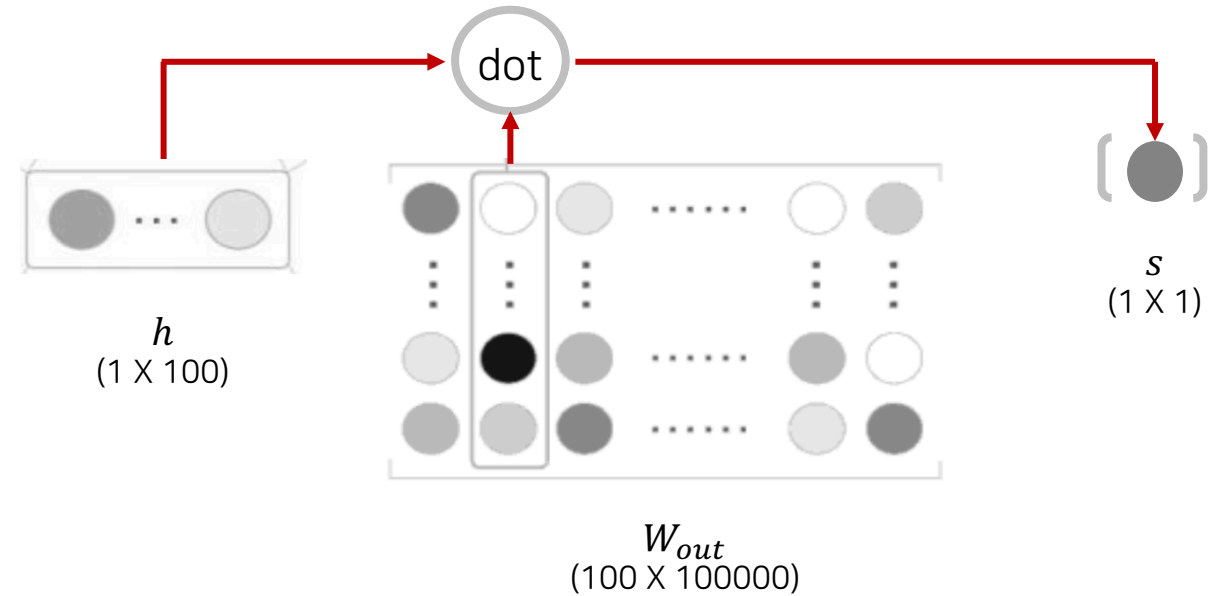
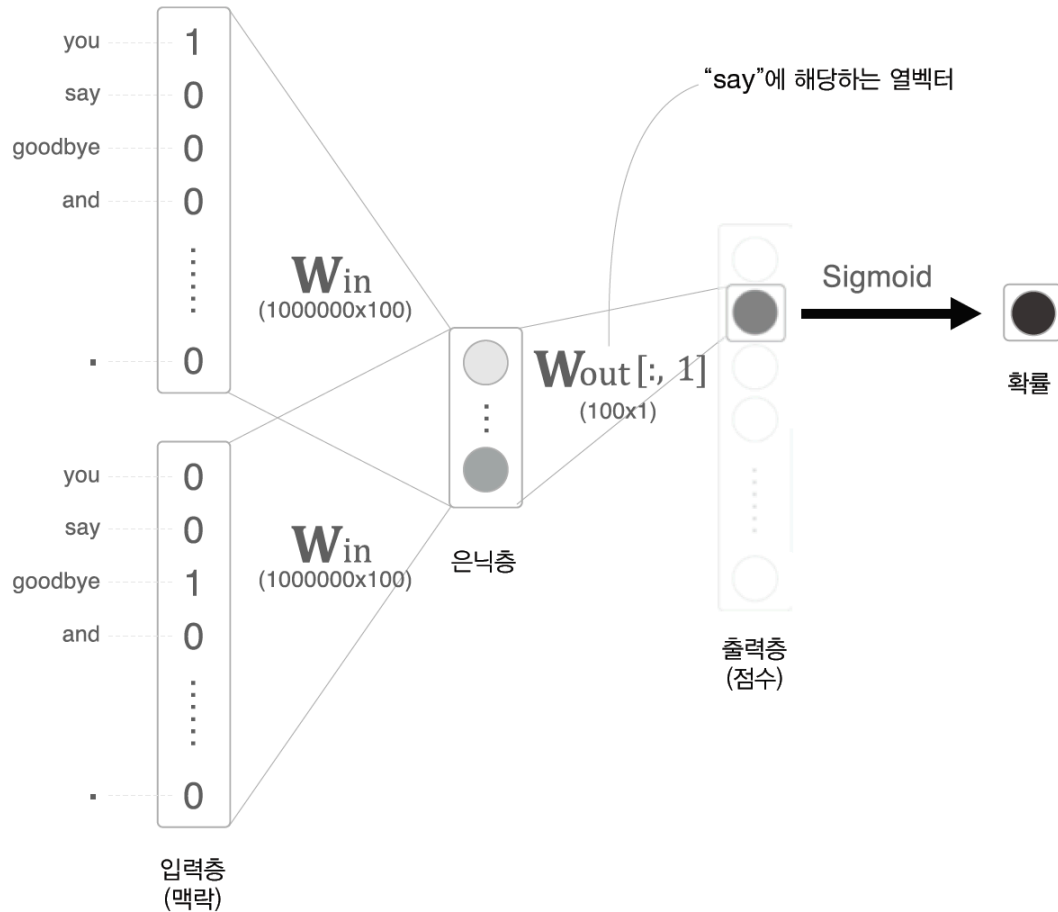
- $$y_k = \frac{\exp(s_k)}{\sum_{i=1}^{1000000} \exp(s_i)}$$
- k번째 원소(단어)를 타깃을 했을 때, 분모의 값을 얻으려면 exp 계산을 100만 번 수행해야 하기 때문에 Softmax를 대신할 계산이 필요

4.2.2 다중 분류에서 이진 분류로

- 지금까지 수행하던 다중 분류(multi-class classification)을 **이진 분류(binary classification)**으로 근사하는 것이 네거티브 샘플링을 이해하는데 중요
 - 지금까지는 'you'와 'goodbye'가 주어지면 정답인 'say'의 확률이 높아지도록 신경망을 학습(다중 분류)
 - " 맥락이 'you'와 'goodbye'일 때 타깃 단어는 무엇입니까?" 라는 질문에 대한 올바른 답을 내어줄 수 있음
 - 이 경우에는 뉴런이 어휘 수 만큼 필요
 - "yes/no"로 답할 수 있는 질문을 만들어 내야 한다.(이진 분류)
 - "맥락이 'you'와 'goodbye'일 때 타깃 단어는 'say'입니까?"라는 질문에 대한 답은 yes/no
 - 이런 경우에는 **output 레이어의 뉴런은 오직 하나만 있으면 됨.**

4.2.2 다중 분류에서 이진 분류로

- 출력층의 뉴런은 하나
- 'say'에 해당하는 열벡터와 은닉층 뉴런과의 내적을 계산



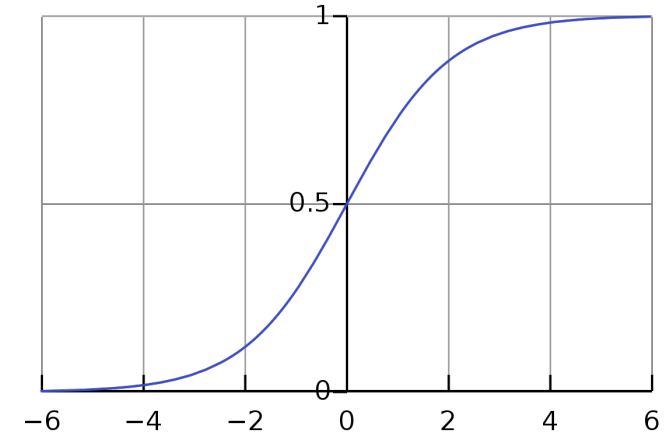
- 모든 단어에 대해 계산하지 않고 하나의 단어에 주목해 그 점수만을 계산

4.2.3 시그모이드 함수와 교차 엔트로피 오차

• 이진 분류 문제를 신경망으로 풀기 위해서는

- 점수에 Sigmoid 함수를 적용해 확률로 변환
- 손실 함수로 교차 엔트로피 오차를 사용
- 다중 분류의 경우 Softmax 함수 & 교차 엔트로피 오차 사용

} 이진 분류 신경망에서 가장 흔하게 사용하는 조합

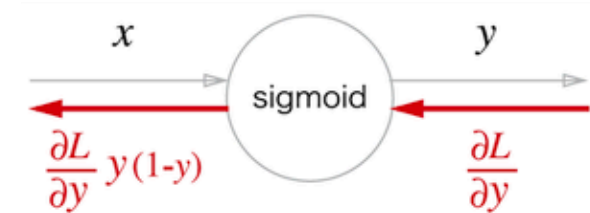
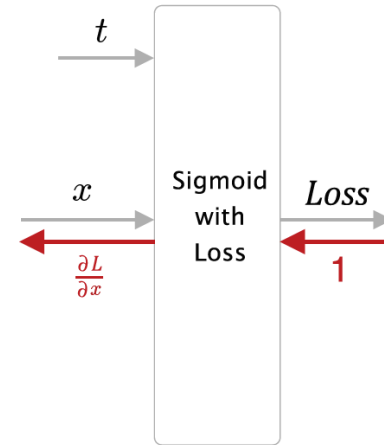
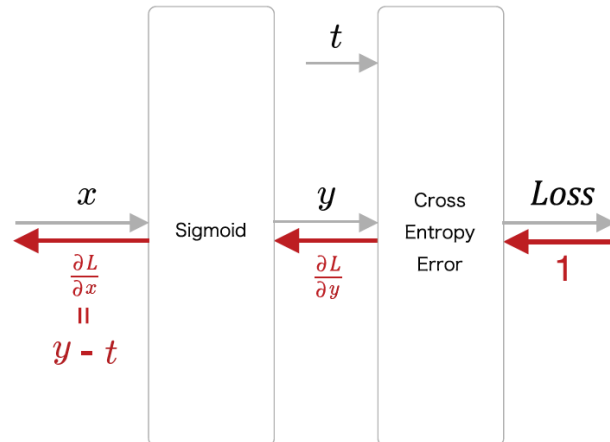


• 시그모이드 함수: $y = \frac{1}{1+\exp(-x)}$

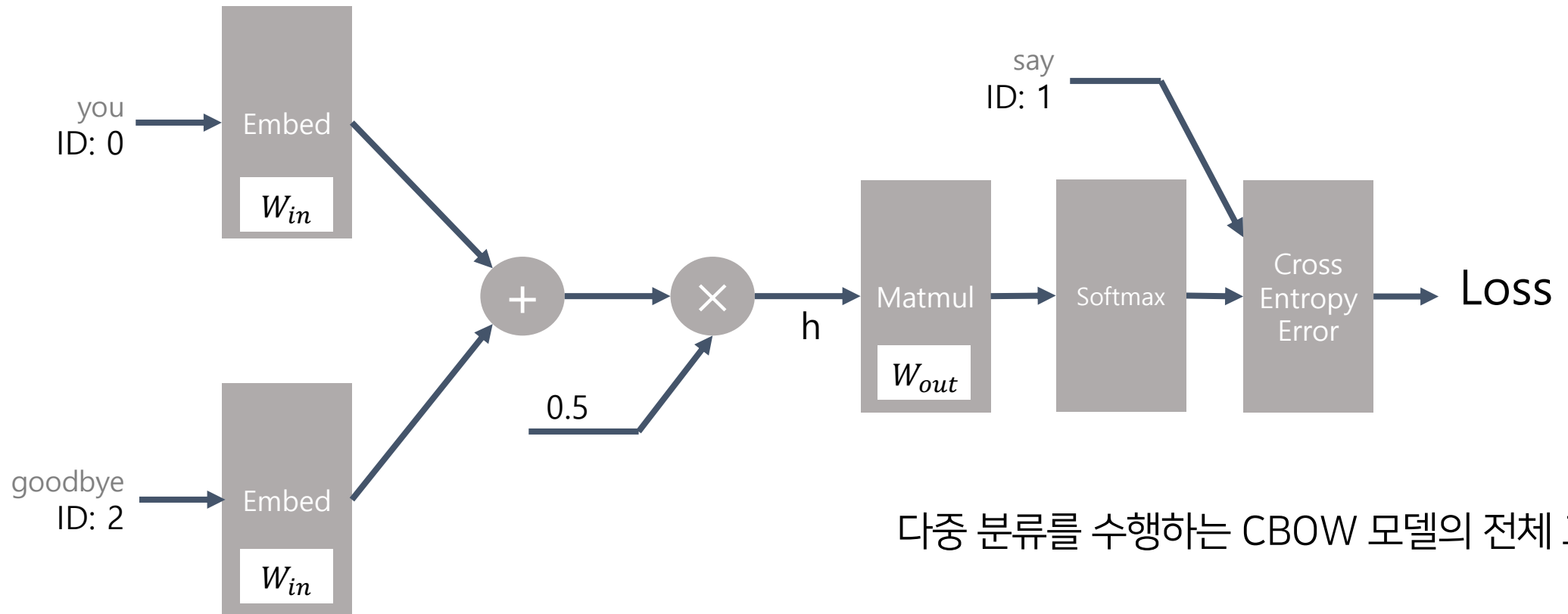
- y : 0 ~ 1 사이의 실수(확률)
- 교차 엔트로피 오차: $L = -(t \log y + (1 - t) \log(1 - y))$ (y : sigmoid 함수 출력, t : 정답 레이블(0 또는 1))
- 역전파의 $y - t$: 확률과 정답 레이블의 차이
 - $t = 1$ 인 경우, y 가 1에 가까워질수록 오차는 줄어들며 반대로 y 가 1로부터 멀어지면 오차가 커짐

* $\frac{\partial L}{\partial y}$ 이 $y - t$ 로 유도되는 과정

$$\begin{aligned} \frac{\partial L}{\partial x} &= \frac{\partial L}{\partial y} \frac{\partial y}{\partial x} \\ \frac{\partial L}{\partial y} &= -\frac{t}{y} + \frac{1-t}{1-y} = \frac{y-t}{y(1-y)} \\ \frac{\partial y}{\partial x} &= y(1-y) \\ \frac{\partial L}{\partial x} &= \frac{y-t}{y(1-y)} y(1-y) = y - t \end{aligned}$$



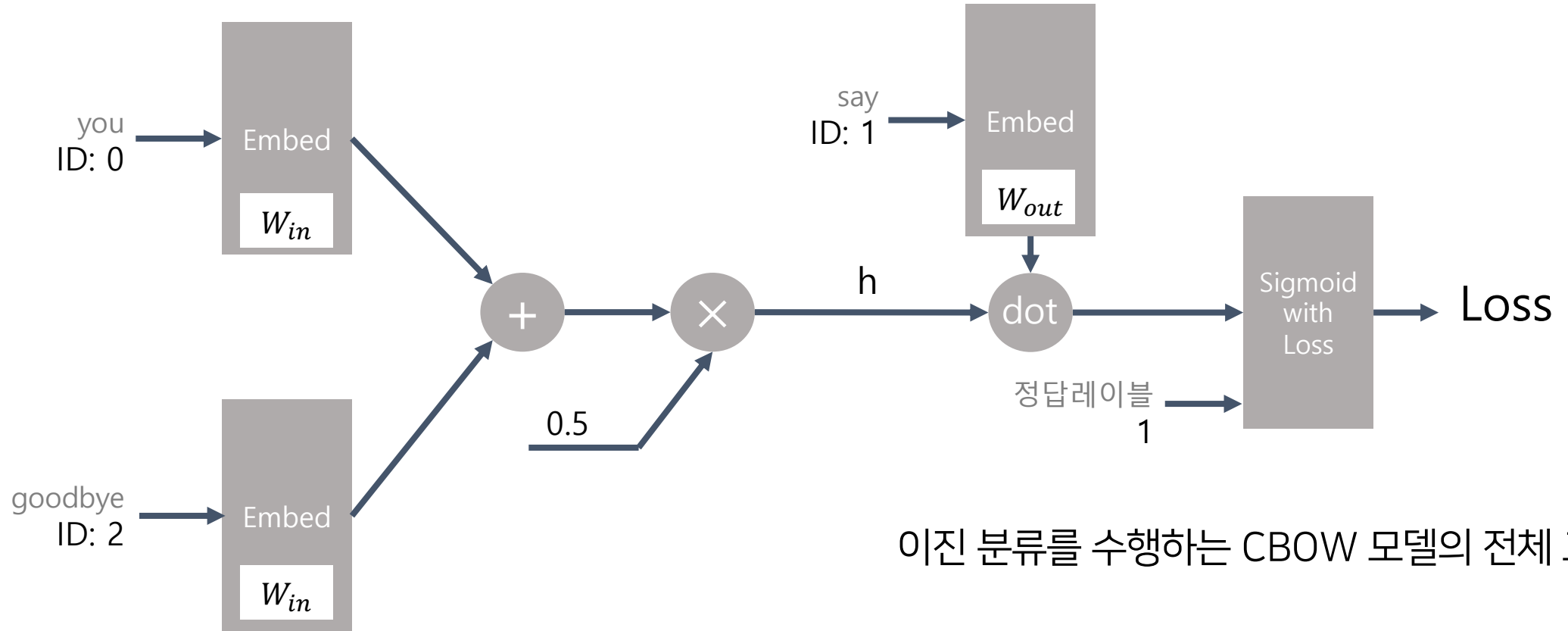
4.2.4 다중 분류에서 이진 분류로(구현)



다중 분류를 수행하는 CBOW 모델의 전체 그림

- 맥락이 'you'와 'goodbye'이고 타겟이 'say'
- Embedding 계층 사용

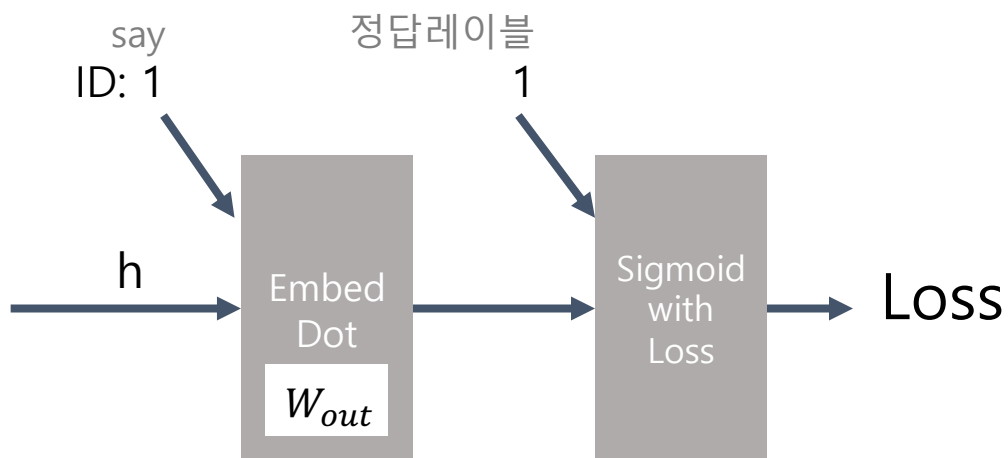
4.2.4 다중 분류에서 이진 분류로(구현)



이진 분류를 수행하는 CBOW 모델의 전체 그림

- 맥락이 'you'와 'goodbye'이고 타깃이 'say'
- Embedding 계층 사용

4.2.4 다중 분류에서 이진 분류로(구현)



params: 매개변수 / grads: 기울기
embed: Embedding 계층 / cache: 순전파 시의 계산 결과 저장

```
# 4.2.4 다중 분류에서 이진 분류로(EmbeddingDot)
class EmbeddingDot:
    def __init__(self, W):
        self.embed = Embedding(W)
        self.params = self.embed.params
        self.grads = self.embed.grads
        self.cache = None

    # hidden layer의 뉴런 h, 단어 id의 numpy 배열 idx
    def forward(self, h, idx):
        target_W = self.embed.forward(idx) # 단어 임베딩 계산
        out = np.sum(target_W * h, axis = 1) # 내적 계산

        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
```

미니배치로 처리 -> idx는 단어 ID의 배열이 된다.

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

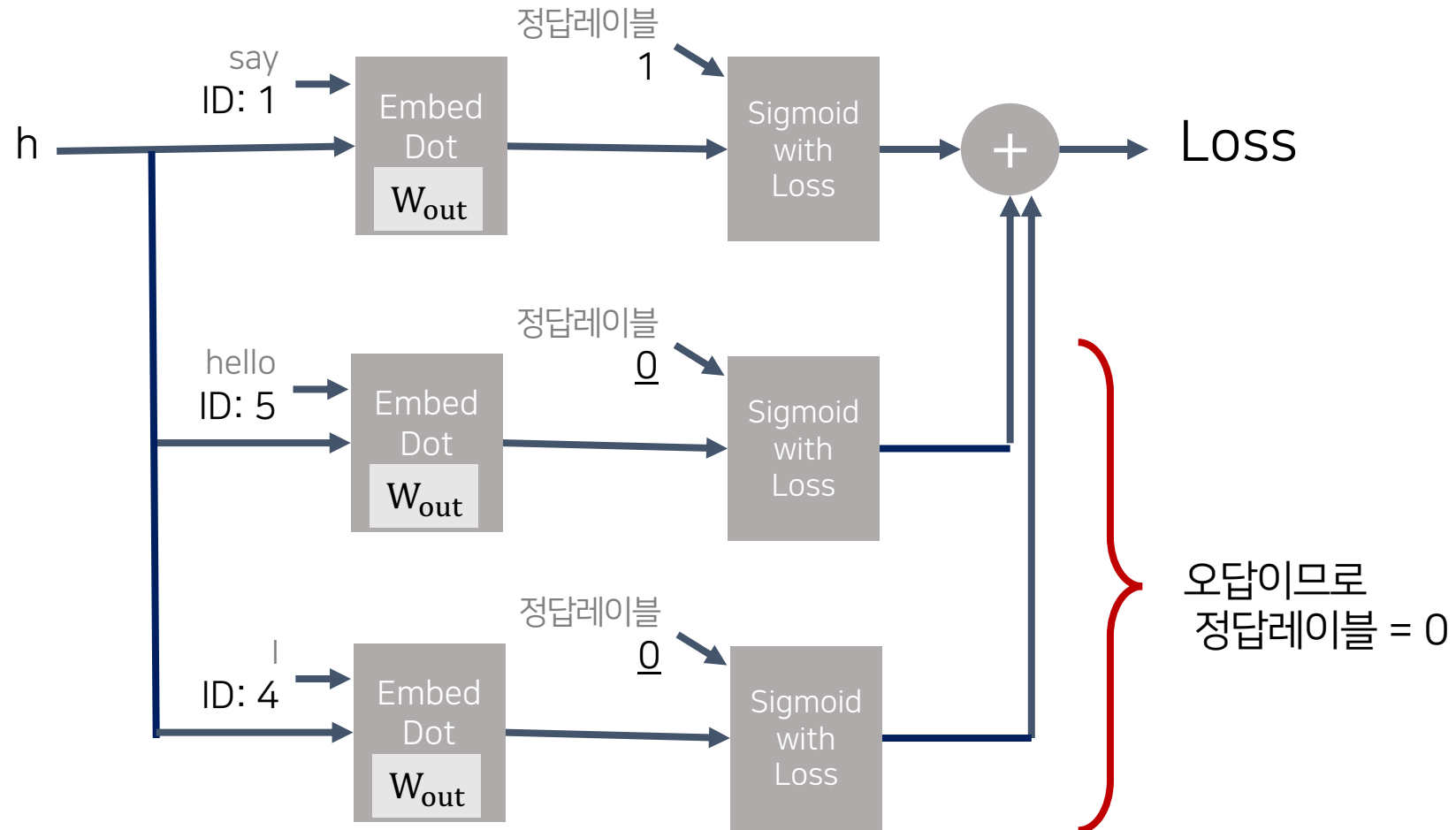
4.2.5 네거티브 샘플링

- 지금까지 다중 분류를 이진 분류로 변환
- 하지만 지금까지는 긍정적인 예(정답)에만 대해 학습
 - \therefore 부정적인 예(오답)이 입력되었을 때의 결과는 확실치 않음



- 맥락이 "you"와 "goodbye", 정답 타깃이 "say"인 경우:
 - 지금까지 정답인 "say"만을 대상으로 이진 분류, 좋은 가중치가 준비되어 있을 경우 output 확률은 1에 가까움
 - 하지만 현재의 신경망에선 정답인 "say"에 대해서만 학습, "say"이외의 단어(오답)에 대해서는 어떠한 지식도 획득하지 못함
 - 따라서 정답("say")에 대해서는 Sigmoid 계층의 출력을 1에 가깝게, 오답("say"이외의 단어)에 대해서는 0에 가깝게 학습해야 함.
- 그러나 모든 오답에 대해 이진 분류를 학습하는 것은 계산량을 증가시킴
 - 따라서 근사적인 해법으로 오답 예를 몇 개 선택, 즉 적은 수의 부정적 예를 샘플링해서 사용 => 네거티브 샘플링
 - 긍정적 예를 타깃으로 한 경우의 loss를 구함
 - 그와 동시에 부정적 예를 몇개 샘플링 해 loss를 구함
 - 각각의 데이터(긍정적/부정적 예)의 손실을 더한 값 => 최종 loss

4.2.5 네거티브 샘플링



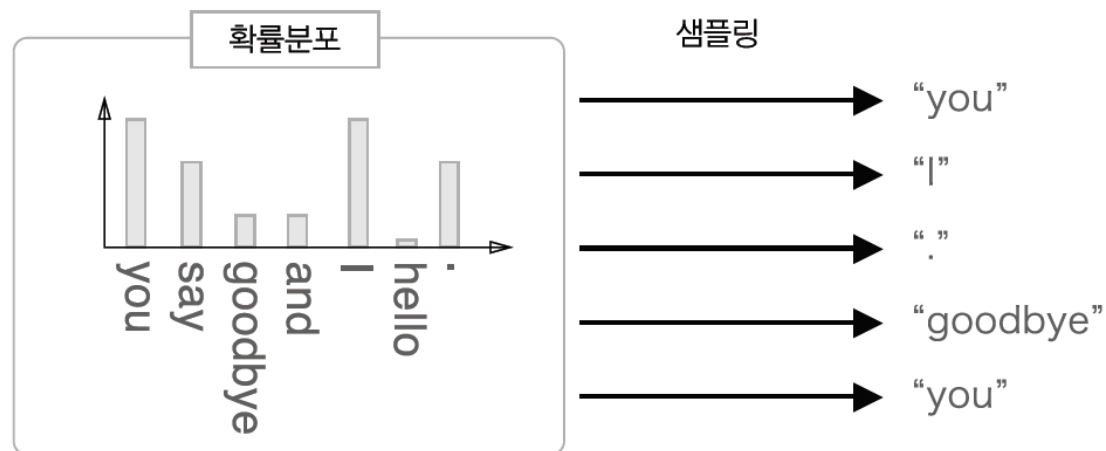
4.2.6 네거티브 샘플링의 샘플링 기법

- 그렇다면 부정적 예를 어떻게 샘플링 해야할 것인가? => 코퍼스의 통계 데이터를 기초로 샘플링

- 말뭉치에서 자주 등장하는 단어를 많이 추출하고 드물게 등장하는 단어를 적게 추출
- 말뭉치에서 각 단어의 출현 횟수를 구해 확률분포로 나타냄
- 그 확률분포대로 단어를 샘플링

- 따라서 자주 등장하는 단어가 선택될 가능성 ↑

- 희소한 단어로만 샘플이 구성되었을 경우, 실전 문제에서도 희소한 단어는 거의 출현하지 않기 때문에 결과도 나쁘게 나오게 됨.
- 드문 단어를 잘 처리하는 일은 중요하지 않으므로 흔한 단어를 잘 처리하는 것이 결과에는 더 좋은 영향



4.2.6 네거티브 샘플링의 샘플링 기법

```
# 4.2.6 네거티브 샘플링
```

```
# 0~9까지 숫자 하나를 무작위로 샘플링  
np.random.choice(10)
```

6

```
np.random.choice(10)
```

2

```
# words에서 하나만 무작위로 샘플링  
words = ['you', 'say', 'goodbye', 'I', 'hello', '.']  
np.random.choice(words)
```

'you'

```
# 5개만 무작위로 샘플링(중복 있음)
```

```
np.random.choice(words, size=5)
```

```
array(['hello', '.', '.', 'you', 'hello'], dtype='<U7')
```

```
# 5개만 무작위로 샘플링(중복 없음)
```

```
np.random.choice(words, size=5, replace=False)
```

```
array(['hello', '.', 'say', 'you', 'I'], dtype='<U7')
```

```
# 확률분포에 따라 샘플링
```

```
p = [0.5, 0.1, 0.05, 0.2, 0.05, 0.1]  
np.random.choice(words, p=p)
```

'you'

- **np.random.choice()**: 무작위 샘플링 용도로 사용 가능
 - size: 샘플링을 size만큼 수행, replace=False: 중복 없음, p: 확률분포대로 샘플링

4.2.6 네거티브 샘플링의 샘플링 기법

- word2vec의 네거티브 샘플링 -> 기본 확률분포에 0.75를 제공하도록 수정

$$P'(w_i) = \frac{P(w_i)^{0.75}}{\sum_j^n P(w_j)^{0.75}} \quad P(w_i): i\text{번째 단어의 확률}$$

- 이는 출현 확률이 낮은 단어를 버리지 않기 위해서임.
 - 0.75를 제공함으로써 원래 낮은 단어의 확률을 약간 증가시킬 수 있음.
 - 수정 전에는 0.01이던 원소가 수정 후에는 0.0256...으로 증가
 - 0.75라는 수치에는 이론적인 의미는 X, 다른 값으로 설정해도 된다.

```
# 4.2.6 네거티브 샘플링의 샘플링 기법
p = [0.7, 0.29, 0.01]
new_p = np.power(p, 0.75) # 0.75 제공
new_p /= np.sum(new_p) # 0.75를 제공한 확률분포의 총합
print(new_p)
```

```
[0.64196878 0.33150408 0.02652714]
```

- 네거티브 샘플링은 이렇게 말뭉치에서 단어의 확률 분포를 만들고, 0.75를 제공한 다음 np.random.choice()를 사용해 부정적 예를 샘플링하게 된다.

4.2.6 네거티브 샘플링의 샘플링 기법

- UnigramSampler 클래스: 부정적 예를 샘플링하는 클래스


```
class UnigramSampler:
    def __init__(self, corpus, power, sample_size):
        self.sample_size = sample_size
        self.vocab_size = None
        self.word_p = None

        counts = collections.Counter()
        for word_id in corpus:
            counts[word_id] += 1

        vocab_size = len(counts)
        self.vocab_size = vocab_size

        self.word_p = np.zeros(vocab_size)
        for i in range(vocab_size):
            self.word_p[i] = counts[i]

        self.word_p = np.power(self.word_p, power)
        self.word_p /= np.sum(self.word_p)
```



확률분포 초기화

4.2.6 네거티브 샘플링의 샘플링 기법

- UnigramSampler 클래스: 부정적 예를 샘플링하는 클래스

```
def get_negative_sample(self, target):
    batch_size = target.shape[0]

    if not GPU:
        negative_sample = np.zeros((batch_size, self.sample_size), dtype=np.int32)

        for i in range(batch_size):
            p = self.word_p.copy()
            target_idx = target[i]
            p[target_idx] = 0
            p /= p.sum()
            negative_sample[i, :] = np.random.choice(self.vocab_size, size=self.sample_size, replace=False, p=p)
    else:
        # GPU(cupy)로 계산할 때는 속도를 우선한다.
        # 부정적 예에 타깃이 포함될 수 있다.
        negative_sample = np.random.choice(self.vocab_size, size=(batch_size, self.sample_size),
                                           replace=True, p=self.word_p)

    return negative_sample
```

확률분포에 따라
negative example 샘플링

4.2.6 네거티브 샘플링의 샘플링 기법

- UnigramSampler 클래스:

```
corpus = np.array([0, 1, 2, 3, 4, 1, 2, 3])
power = 0.75
sample_size = 2
```

```
sampler = UnigramSampler(corpus, power, sample_size)
target = np.array([1, 3, 0]) # target으로 지정한 단어는 positive, 그 외의 단어는 negative로 샘플링
negative_sample = sampler.get_negative_sample(target)
print(negative_sample)
```

긍정적 예로 3개의 데이터 ->

각각의 데이터에 대해 부정적 예를 2개씩 샘플링

```
[[2 0] # 첫번째 데이터(1)에 대한 부정적 예
 [1 0] # 두번째 데이터(3)에 대한 부정적 예
 [3 4]] # 세번째 데이터(0)에 대한 부정적 예
```


4.2.7 네거티브 샘플링 구현

- NegativeSamplingLoss 클래스:

- W: 출력 측 가중치
- corpus: 말뭉치(단어 ID의 리스트)
- power: 확률분포에 제공할 값
- sample_size: 부정적 예의 샘플링 횟수

```
class NegativeSamplingLoss:
    def __init__(self, W, corpus, power=0.75, sample_size=5):
        self.sample_size = sample_size
        self.sampler = UnigramSampler(corpus, power, sample_size) # 부정적 예 샘플링하는 UnigramSampler 클래스
        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
```

} loss_layer & embed_dot_layer에서는
원하는 계층을 리스트로 보관

- 이때 두 리스트는 sample_size + 1개의 계층을 생성
- 부정적 예를 다루는 레이어가 sample_size개, 여기에 긍정적 예를 다루는 레이어가 하나 더 필요하기 때문

4.2.7 네거티브 샘플링 구현

- NegativeSamplingLoss 클래스:

```
def forward(self, h, target):  
    # h: 은닉층 뉴런  
    # target: 긍정적 예  
    batch_size = target.shape[0]  
    negative_sample = self.sampler.get_negative_sample(target) # 부정적 예를 샘플링해 negative_sample에 저장  
    # * sampler: 부정적 예 샘플링하는 UnigramSampler 클래스  
    # 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)  
  
    return loss
```

Embedding Dot 계층의 forward 점수를 구하고, 이 점수와 레이블을 'sigmoid with loss' 레이어로 전달해 손실을 구한다.
- 이때 correct_label = 1, negative_label = 0

4.2.7 네거티브 샘플링 구현

- NegativeSamplingLoss 클래스:

순전파 때의 역순으로 각 계층의 backward()를 호출하기만 하면 됨

```
def backward(self, dout=1):  
    dh = 0  
    for l0, l1 in zip(self.loss_layers, self.embed_dot_layers):  
        dscore = l0.backward(dout)  
        dh += l1.backward(dscore) # 각 Embed_dot 레이어에 대한 기울기 값을 더해준다.  
  
    return dh
```

4.3 개선판 word2vec 학습

- Embedding 계층과 네거티브 샘플링 기법을 적용한 신경망을 구현해보자.
- 그리고 이를 PTB 데이터셋을 사용해 학습해 실용적인 단어의 분산 표현을 얻어보자.

4.3.1 CBOW 모델 구현

- CBOW 모델: SimpleCBOW + Embedding 레이어 + Negative Sampling Loss 레이어

```
class CBOW:
    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): # Embedding 레이어: 2 * window_size개 작성해 in_layers에 배열로 보관
            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) # Negative Sampling Loss 레이어 생성

        # 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 # 나중에 단어의 분산표현에 접근할 수 있도록 word_vecs에 W_in을 할당
```

- vocab_size: 어휘 수
- hidden_size: 은닉층의 뉴런 수
- corpus: 단어 ID 목록
- window_size: 맥락의 크기(주변 단어 중 몇개나 맥락으로 포함시킬지)
- window_size = 2, 타깃 좌우 2개씩 총 4개 단어가 맥락

네트워크에서 사용하는 모든 매개변수와 기울기를 params와 grads에 보관

4.3.1 CBOW 모델 구현

```
def forward(self, contexts, target): - contexts, target: 단어 ID
```

```
h = 0
```

```
for i, layer in enumerate(self.in_layers):
```

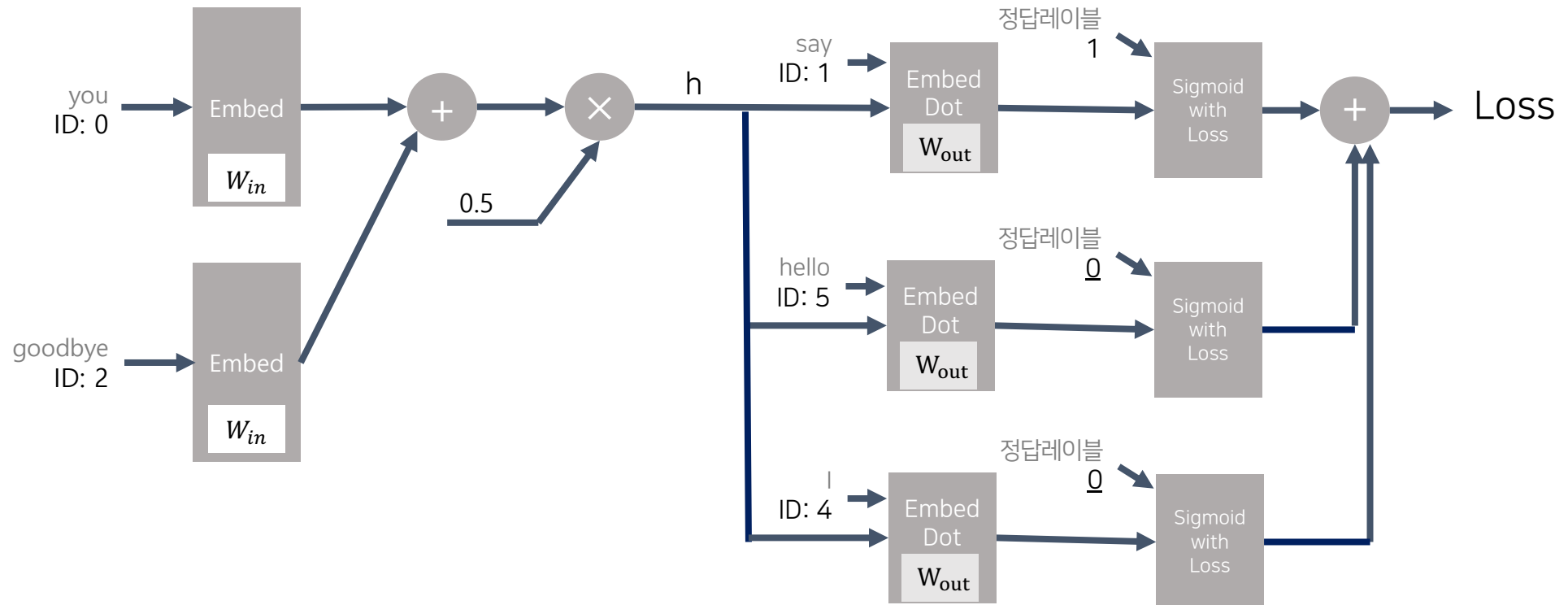
```
    h += layer.forward(contexts[:, i])
```

```
h *= 1 / len(self.in_layers)
```

```
loss = self.ns_loss.forward(h, target)
```

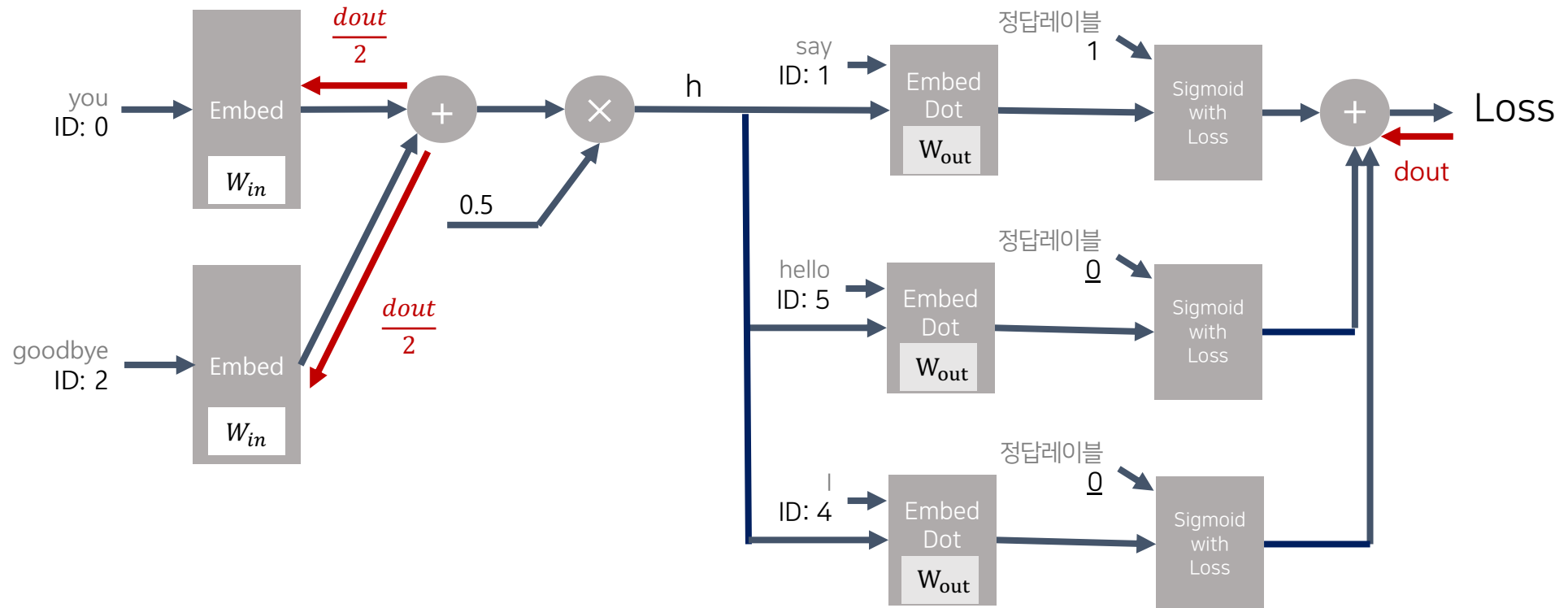
```
return loss
```

맥락 (contexts)	타겟	
you, goodbye	say	단어 ID →
say, and	goodbye	
goodbye, I	and	
and, say	I	
I, hello	say	
say, .	hello	
맥락 (contexts)	타겟	
[[0 2]	[1	
[1 3]	2	
[2 4]	3	
[3 1]	4	
[4 5]	1	
[1 6]]	5]	



4.3.1 CBOW 모델 구현

```
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
```



4.3.2 CBOW 모델 학습 코드

```
# 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)
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' # cbow_params.pkl: 단어 분산 표현 저장
with open(pkl_file, 'wb') as f:
    pickle.dump(params, f, -1)
```


4.3.3 CBOW 모델 평가

```
pkl_file = 'cbow_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
```

```
queries = ['you', 'year', 'car', 'toyota']
```

```
for query in queries:
```

```
    most_similar(query, word_to_id, id_to_word, word_vecs, top=5)
```

비슷한 인칭대명사들

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

기간을 뜻하는 단어들

```
[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
```

자동차 메이커들

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

우리가 구현한 CBOW 모델로 획득된 단어 분산 표현이
제법 괜찮은 특성을 지니고 있음을 볼 수 있음.



4.3.3 CBOW 모델 평가

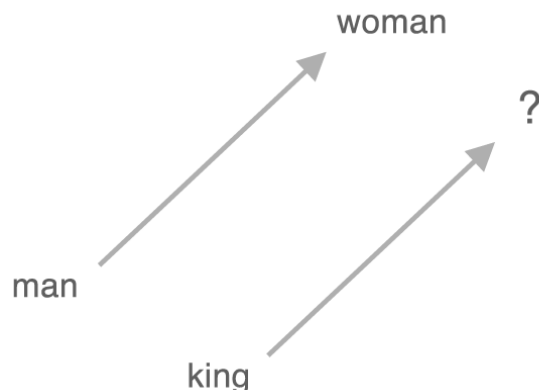
- word2vec으로 얻은 단어 분산 표현 -> 비슷한 단어를 모을 뿐만 아니라 더 복잡한 패턴도 파악 가능
 - 예) "king - man + woman = queen"와 같은 유추(비유) 문제
 - 단어 벡터 공간에서 "man -> woman" 벡터와 "king -> ?" 벡터가 가장 가까워지는 단어를 찾는다.
 - 이 관계를 수식으로 나타내면 $\text{vec}(\text{'king'}) + \text{vec}(\text{'woman'}) - \text{vec}(\text{'man'}) = \text{vec}(\text{'?'})$ => **analogy()** 함수로 처리

```
def analogy(a, b, c, word_to_id, id_to_word, word_matrix, top=5, answer=None):
    for word in (a, b, c):
        if word not in word_to_id:
            print('%s(을)를 찾을 수 없습니다.' % word)
            return

    print('\n[analogy] ' + a + ':' + b + ' = ' + c + ' :?')
    a_vec, b_vec, c_vec = word_matrix[word_to_id[a]], word_matrix[word_to_id[b]], word_matrix[word_to_id[c]]
    query_vec = b_vec - a_vec + c_vec
    query_vec = normalize(query_vec)

    similarity = np.dot(word_matrix, query_vec)

    if answer is not None:
        print("==>" + answer + ":" + str(np.dot(word_matrix[word_to_id[answer]], query_vec)))
```



```
count = 0
for i in (-1 * similarity).argsort():
    if np.isnan(similarity[i]):
        continue
    if id_to_word[i] in (a, b, c):
        continue
    print(' {0}: {1}'.format(id_to_word[i], similarity[i]))

    count += 1
    if count >= top:
        return
```

4.3.3 CBOW 모델 평가

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

- 단어 분산 표현을 사용하면 벡터의 덧셈과 뺄셈으로 유추 문제를 풀 수 있다.
 - 단어의 단순한 의미 뿐 아니라 **문법적 패턴도 파악 가능**
- 하지만 PTB 데이터셋 -> 크기가 크지 않기 때문에 더 큰 말뭉치로 학습하게 된다면 더 정확하고 견고한 단어 분산 표현을 얻을 수 있을 것

```
[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
```

4.4 word2vec 남은 주제:

1. word2vec을 사용한 애플리케이션의 예

- 단어 분산 표현이 중요한 이유:

- 1) 전이 학습을 가능하게 한다.

- 전이 학습(transfer learning): 한 분야에서 배운 지식을 다른 분야에도 적용하는 기법
 - NLP에서 word2vec 단어 분산 표현을 처음부터 학습하는 일은 거의 없음.
 - 먼저 큰 말뭉치(위키백과, 구글 뉴스 텍스트 데이터 등)로 학습을 끝낸 후, 그 분산 표현을 이용해 단어를 벡터로 변환

- 2) 단어 또는 문장을 고정 길이 벡터로 변환해준다.

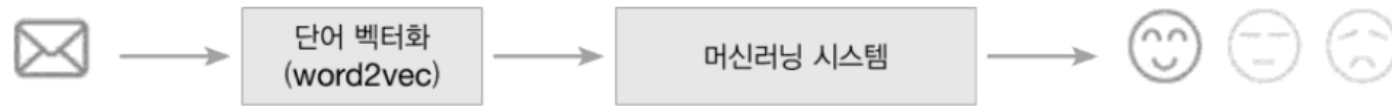
- 자연어를 벡터로 변환할 수 있다면 일반적인 머신러닝 기법(신경망이나 SVM 등)을 적용할 수 있기 때문에 단어 또는 문장을 고정 길이 벡터로 변환하는 것은 매우 중요하다.



- bag-of-words: 문장의 각 단어를 분산 표현으로 변환하고 그 합을 구하는 방법, 단어의 순서는 고려하지 않는다.
 - 순환 신경망(RNN)

4.4.1. 단어 벡터 평가 방법

- 단어 분산 표현 -> 보통은 특정한 애플리케이션에서 사용되는 것이 대부분



- 예를 들어 메일 자동 분류 시스템(감정 분석)을 만들 때, 이 시스템은 여러 시스템으로 구성되게 된다.
 - 단어 분산 표현을 만드는 시스템(word2vec) / 특정 문제에 대해 분류를 수행하는 시스템(감정 분류하는 SVM)
- 분산 표현을 만드는 시스템과 분류하는 시스템의 학습 -> 따로 수행할 수도 있다.
 - 우선 단어의 분산 표현을 학습하고, 그 분산 표현을 사용해 또 하나의 머신러닝 시스템을 학습 -> 즉 두 단계의 학습을 수행한 다음 평가
 - 두 시스템 각각의 하이퍼파라미터를 찾기 위한 튜닝도 필요
 - 시간이 오래 걸림
- 따라서 단어 분산 표현의 우수성은 실제 애플리케이션과는 분리해 평가하는 것이 일반적
 - 보통 단어의 '유사성'이나 '유추 문제'를 척도로 평가
 - 유사성: 예를 들어 유사도를 0 ~ 10 사이로 점수화한다면 cat/animal: 8, cat:car: 2와 같이 사람이 단어 사이의 유사한 정도를 규정하고, 그 점수와 word2vec에 의한 코사인 유사도 점수를 비교해 그 상관성을 본다.
 - 유추 문제: "king:queen = man:?"과 같은 유추 문제를 출제하고, 그 정답률로 단어의 분산 표현의 우수성을 측정

4.4.1. 단어 벡터 평가 방법

- word2vec 모델, 단어의 분산 표현의 차원 수, 말뭉치의 크기를 매개변수로 사용해 비교 실험을 수행한 결과:

모델	차수	말뭉치 크기	의미(semantics)	구문(syntax)	종합
CBOW	300	16억	16.1	52.6	36.1
skip_gram	300	10억	61	61	61
CBOW	300	60억	63.6	67.4	65.7
skip_gram	300	60억	73.0	66.0	69.1
CBOW	1000	60억	57.3	68.9	63.7
skip_gram	1000	60억	66.1	65.1	65.6

Jeffrey Pennington, Richard Socher and Christopher D. Manning: "GloVe: Global Vectors for Word Representation." EMNLP. Vol.14. 2014.

- 의미(semantics): 단어의 의미를 유추하는 유추 문제의 정답률 eg) king : queen = actor : actress
- 구문(syntax): 단어의 형태 정보를 묻는 문제 eg) bad : worst = good : best
- 유추 문제를 이용하면 단어의 의미나 문법적인 문제를 제대로 이해하고 있는 지를 어느 정도 측정할 수 있다.
- 다만 단어의 분산 표현의 우수함이 애플리케이션에 얼마나 기여하는지는 상황에 따라 다르며, 유추 문제에 의한 평가가 높다고 해서 애플리케이션에서도 반드시 좋은 결과가 나오리라는 보장은 없다.

4.5 정리

- 앞 장에서 구현한 CBOW 모델을 개선
 - Embedding 레이어를 구현
 - 네거티브 샘플링 도입
- '모두' 대신 '일부'를 처리하자.
 - 코퍼스 어휘 수 증가에 비례해 계산량이 증가
 - 네거티브 샘플링을 사용해 '모든' 단어가 아닌 '일부' 단어만을 대상으로 계산을 효율적으로 수행
- word2vec -> 자연어 처리 분야에 큰 영향
 - 이를 통해 얻은 단어의 분산 표현은 다양한 자연어 처리 분야에 이용되고 있음.