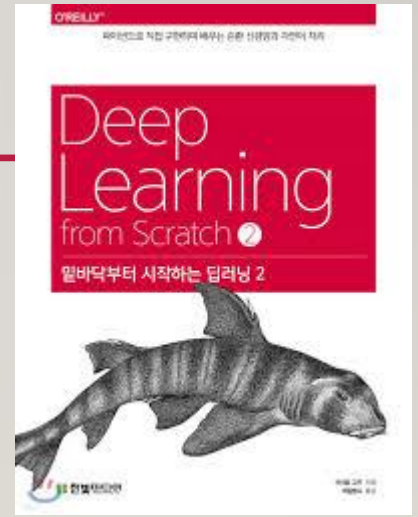
밑바닥부터 시작하는 딥러닝2

3장.WORD2VEC

4장.WORD2VEC 속도 개선



AI융합학부 20193124 고경빈

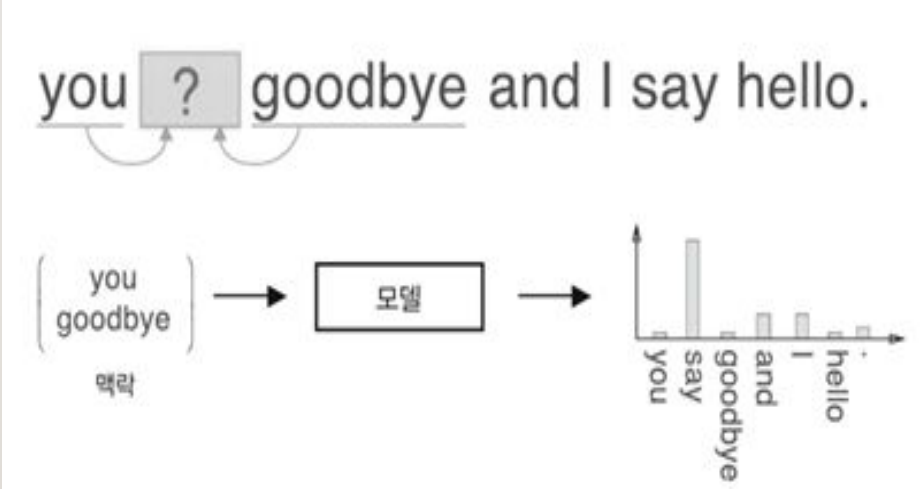
목차

통계 기반 기법의 문제점

- **주변 단어의 빈도**를 기초로 단어 표현 → **대규모** 말뭉치를 다룰 때 문제가 발생
- 말뭉치 전체의 통계(동시발생 행렬과 PPMI 등)를 이용해 **단 1회의 처리** (SVD 등)만에 단어의 분산 표현을 얻음
- $O(n^3)$

추론 기반 기법

- 주변 단어 (맥락)가 주어졌을 때 **“?”에 무슨 단어가 들어가는지** 를 추측하는 작업
- 신경망을 이용하는 경우 미니배치로 학습하는 것이 일반적 -> 신경망이 **한번에 소량(미니배치)의 학습 샘플씩 반복해서 학습**하며 가중치를 갱신
  - 말뭉치의 어휘 수가 많아 SVD 등 계산량이 큰 작업을 처리하기 어려운 경우에도 신경망을 학습 가능



통계 기반 VS 추론 기반


그림 1 통계 기반 기법과 추론 기반 기법 비교
통계 기반 기법(배치 학습)
SVD
학습 데이터
추론 기반 기법(미니배치 학습)
신경망
학습 데이터

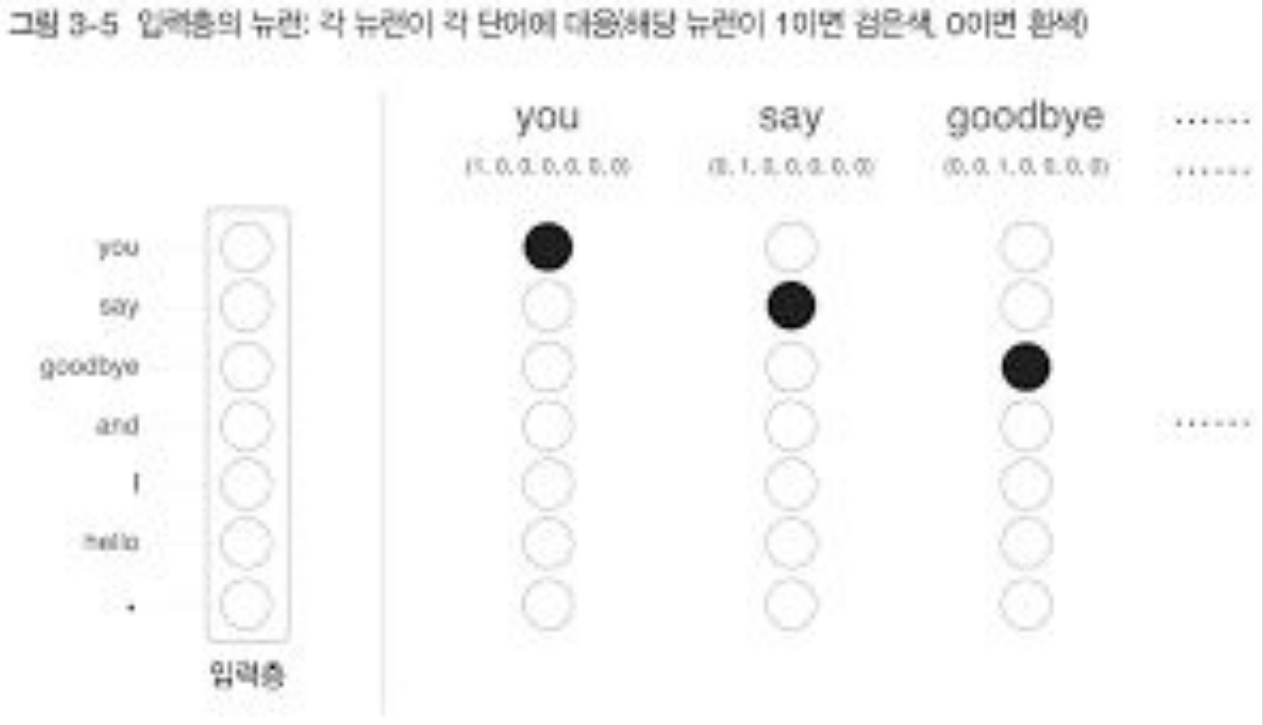
ONE-HOT 표현

- 원핫 표현: 벡터 원소 중 하나만 1이고, 나머지는 모두 0인 벡터

그림 4 단어, 단어 ID, 원핫 표현

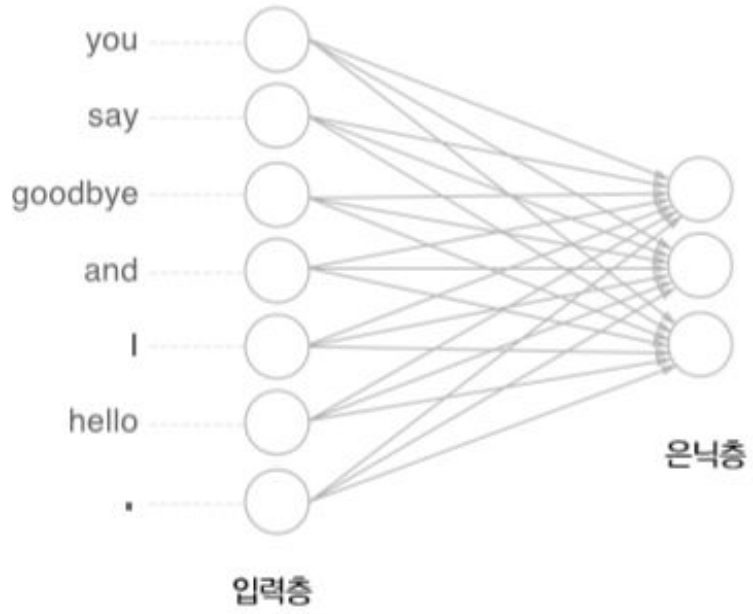
단어(텍스트)	단어 ID	원핫 표현
you goodbye	0 2	(1, 0, 0, 0, 0, 0, 0) (0, 0, 1, 0, 0, 0, 0)

ONE-HOT 표현

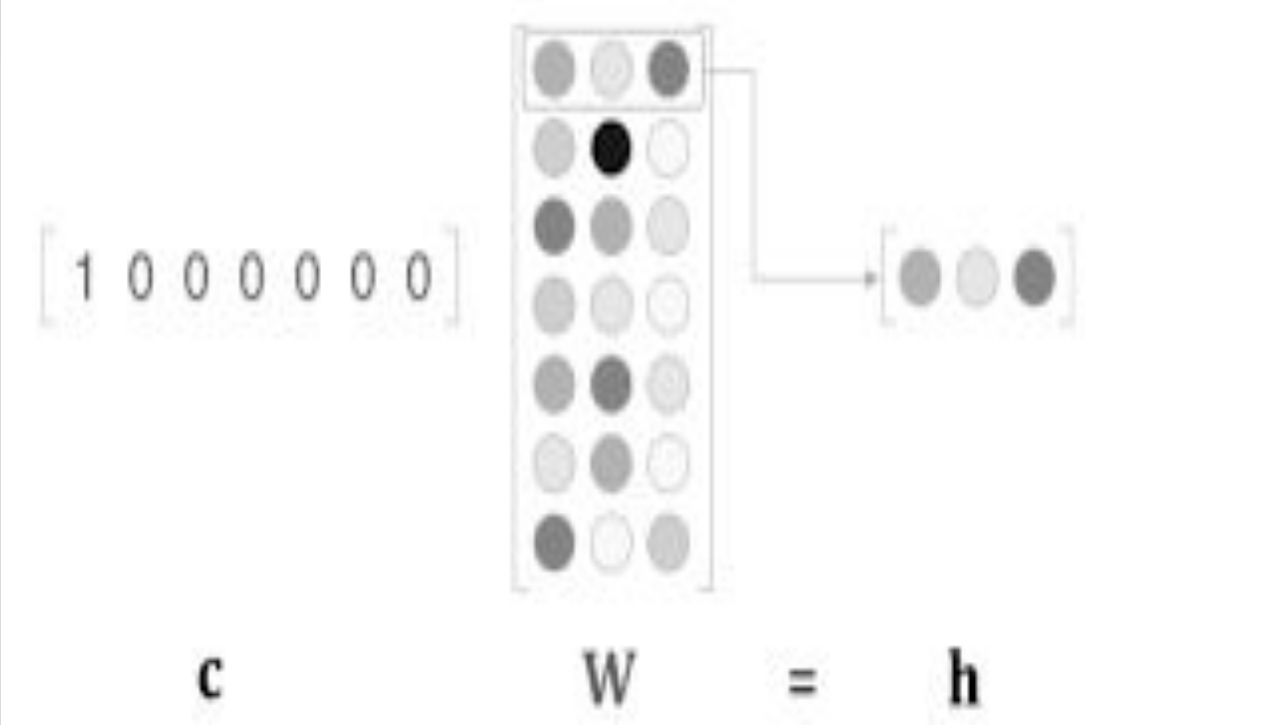


신경망 처리

그림 6 완전연결계층에 의한 변환: 입력층의 각 뉴런은 7개의 단어 각각에 대응(은닉층 뉴런은 3개를 준비함)

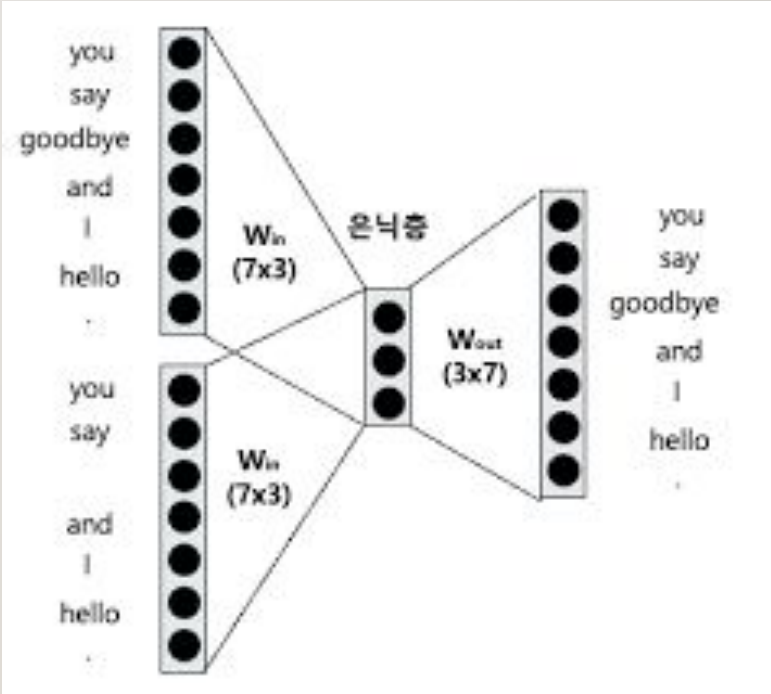


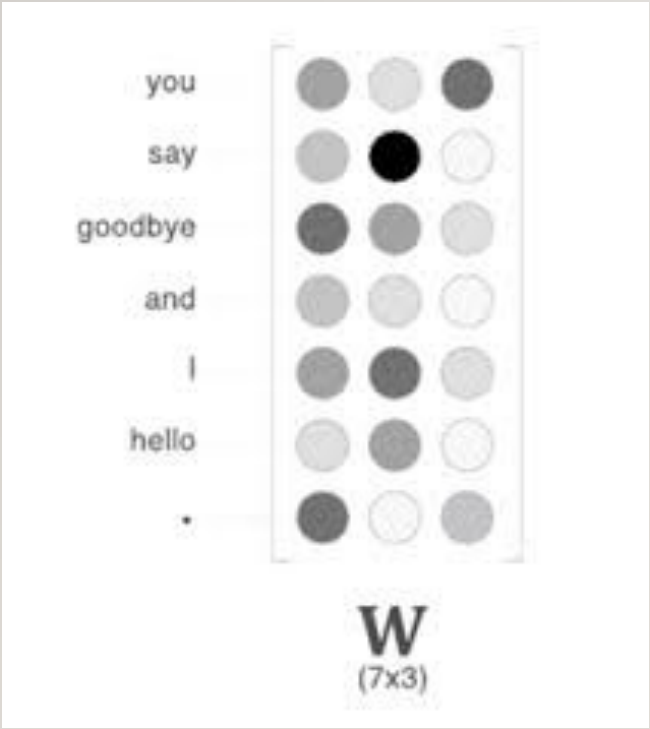
MATMUL 계층의 행렬 곱



CBOW 모델

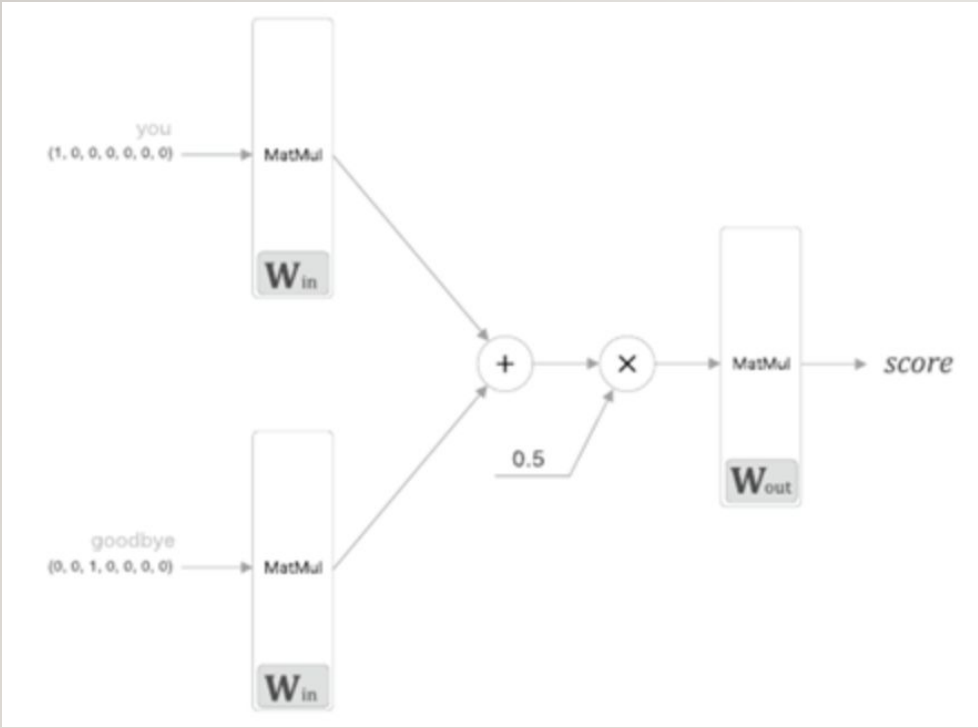
- **맥락으로부터 타깃을 추측**하는 용도의 신경망
  - 타깃: 중앙 단어
  - 맥락: 그 주변 단어들





- **인코딩**
  - 은닉층의 정보는 인간은 이해할 수 없는 코드로 쓰임.
- **디코딩**
  - 은닉 층의 정보로부터 원하는 결과를 얻는 작업
    ➡ 인코딩 된 정보를 이해 가능한 표현으로 복원하는 작업

CBOW 모델(계층 관점)

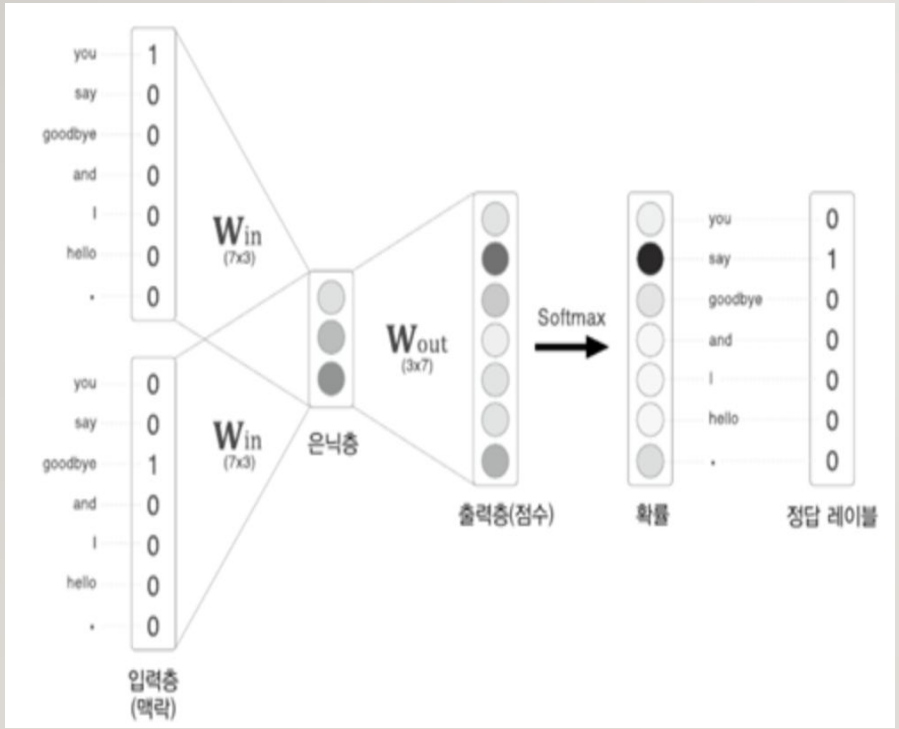


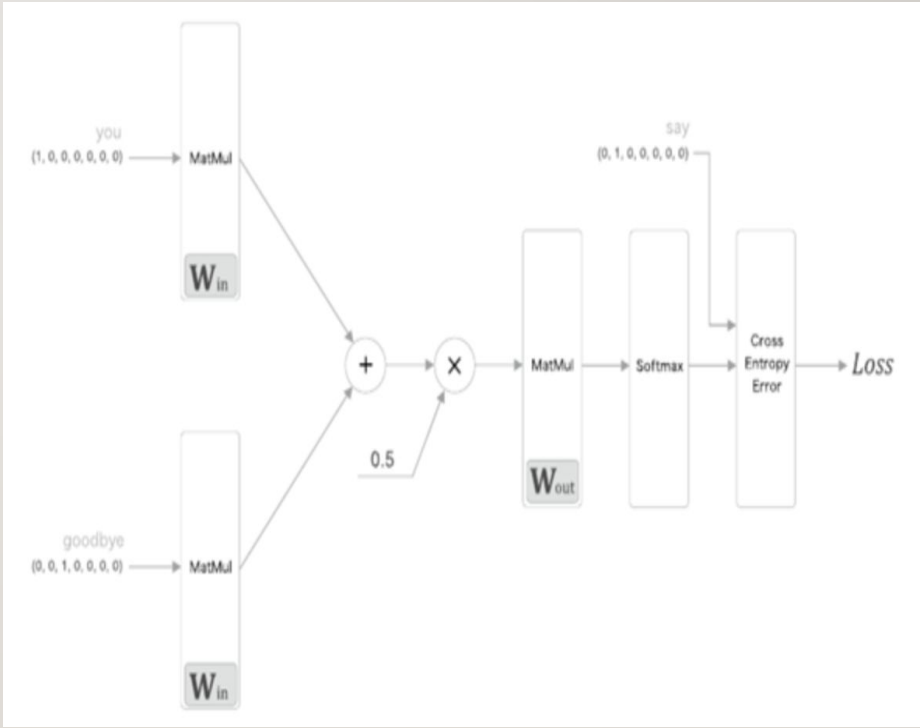

```
c0 = np.array([[1,0,0,0,0,0,0]])
c1 = np.array([[0,0,1,0,0,0,0]])

# 가중치 초기화
W_in = np.random.randn(7,3)
W_out = np.random.randn(3,7)

# 계층 생성
in_layer0 = MatMul(W_in)
in_layer1 = MatMul(W_in)
out_layer = MatMul(W_out)

# 순전파
h0 = in_layer0.forward(c0)
h1 = in_layer1.forward(c1)
h = 0.5 * (h0 + h1)
s = out_layer.forward(h)

print(s)
```

✓ 0.0s

```
[[ 1.34509126e+00  9.95018678e-05  5.21067845e-01  8.96541169e-01
   4.75188827e-01  1.38199929e+00 -1.36050275e-01]]
```

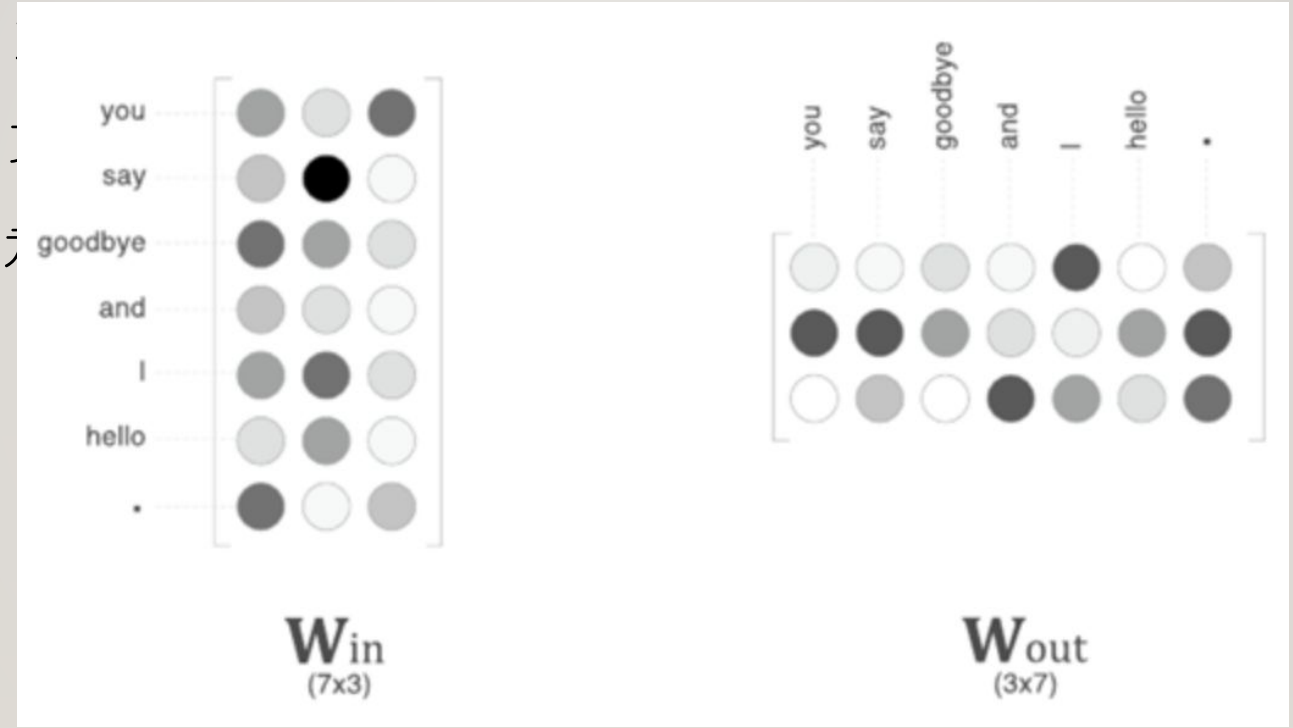
CBOW 모델의 학습


you
say
goodbye
and
I
hello
.
Win
(7x3)
Wout
(3x7)
Softmax
은닉층
입력층
(맥락)
출력층(점수)
확률
정답 레이블


you
(1, 0, 0, 0, 0, 0, 0)
MatMul
Win
say
(0, 1, 0, 0, 0, 0, 0)
goodbye
(0, 0, 1, 0, 0, 0, 0)
+
X
0.5
MatMul
Wout
Softmax
Cross
Entropy
Error
Loss

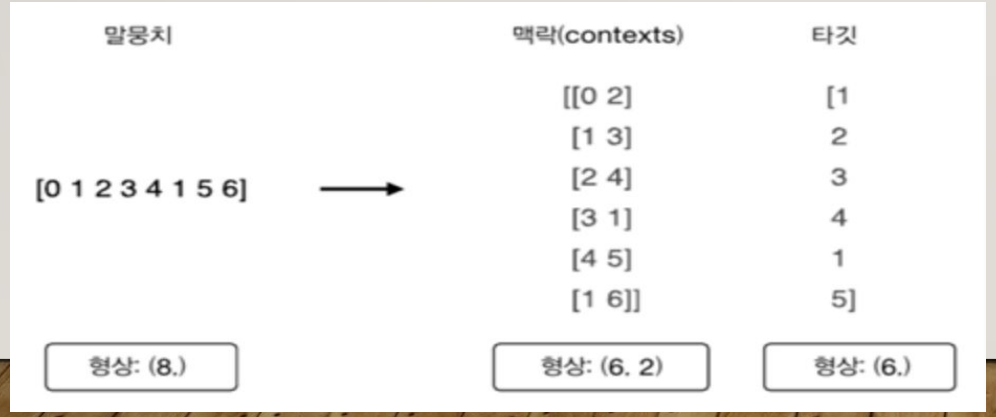
WORD2VEC 가중치와 분산 표현

- A. 입력 측의
- B. 출력 측의
- C. 양쪽 가중



학습 데이터 준비

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



말뭉치

[0 1 2 3 4 1 5 6]

맥락(contexts)

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

타깃

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

형상: (8,)

형상: (6, 2)

형상: (6,)

```python
from common.util import preprocess

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

✓ 0.0s

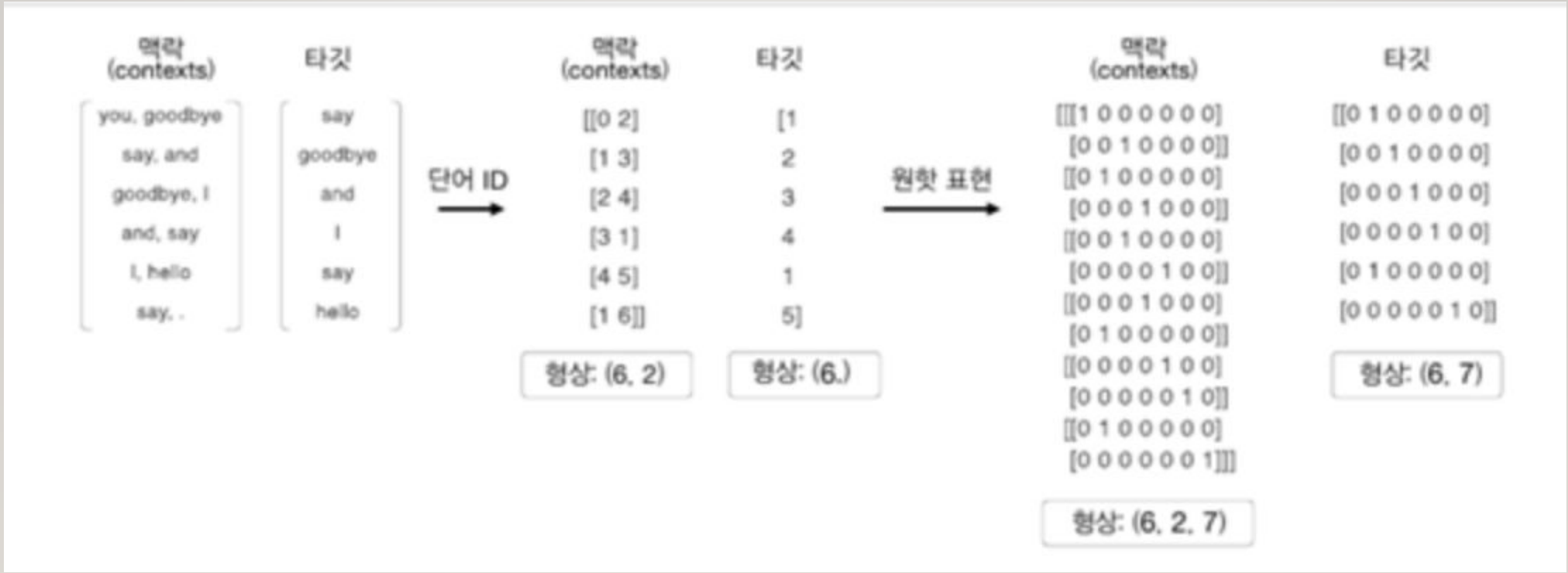
```python
def create_contexts_target(corpus, window_size=1):
    target = corpus[window_size:-window_size]
    contexts = []

    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)
```

✓ 0.0s

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

print("context")
print(contexts)
print("target")
print(target)
```

✓ 0.0s

원핫 표현으로 변환


맥락 (contexts)
타깃
you, goodbye
say, and
goodbye, I
and, say
I, hello
say, .
say
goodbye
and
I
say
hello
단어 ID
맥락 (contexts)
[[0 2]
[1 3]
[2 4]
[3 1]
[4 5]
[1 6]]
형상: (6, 2)
타깃
[1
2
3
4
1
5]
형상: (6,)
원핫 표현
맥락 (contexts)
[[[1 0 0 0 0 0 0]
[0 0 1 0 0 0 0]]
[[0 1 0 0 0 0 0]
[0 0 0 1 0 0 0]]
[[0 0 1 0 0 0 0]
[0 0 0 0 1 0 0]]
[[0 0 0 1 0 0 0]
[0 1 0 0 0 0 0]]
[[0 0 0 0 1 0 0]
[0 0 0 0 0 1 0]]
[[0 1 0 0 0 0 0]
[0 0 0 0 0 0 1]]]
형상: (6, 2, 7)
타깃
[[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]]
형상: (6, 7)

CBOW 모델 구현

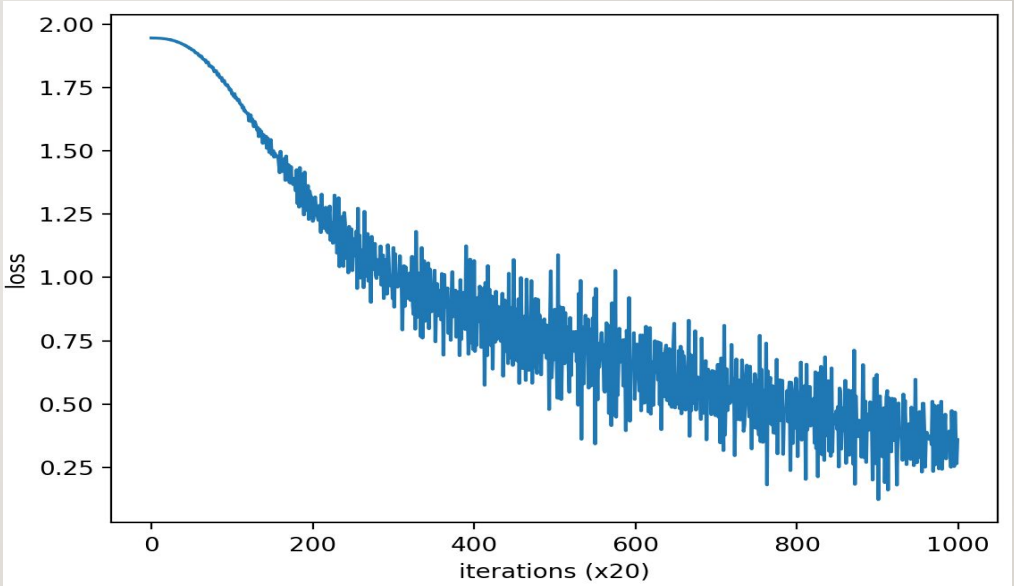
```python
import sys
sys.path.append('..')
import numpy as np
from common.layers import MatMul, SoftmaxWithLoss

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
```

```python
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
```

```python
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
```

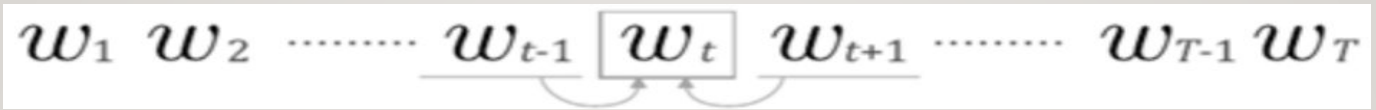
학습 코드 구현

```python
from common.trainer import Trainer
from common.optimizer import Adam
from simple_cbow import SimpleCBOW
from common.util import preprocess, create_contexts_target, convert_one_hot


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])
```




```
you [-1.1390966  1.0270826 -1.8008165  1.0217851 -1.0095915]
say [-0.752425  -1.1735805 -1.0885915 -1.2126366  1.2677171]
goodbye [-0.8306098   1.0163187   0.95335877  0.9845747  -0.9540226 ]
and [-2.1088328  -0.8792586  -0.47447327 -0.86699957  0.7294212 ]
i [-0.86429846  1.0391984   0.96972007  0.97096974 -0.95440185]
hello [-1.1294588  1.0231335 -1.8068862  1.0205009 -0.9935971]
. [ 1.5366627 -1.2099228 -1.4364132 -1.2952621  1.389269 ]
```

CBOW 모델과 확률

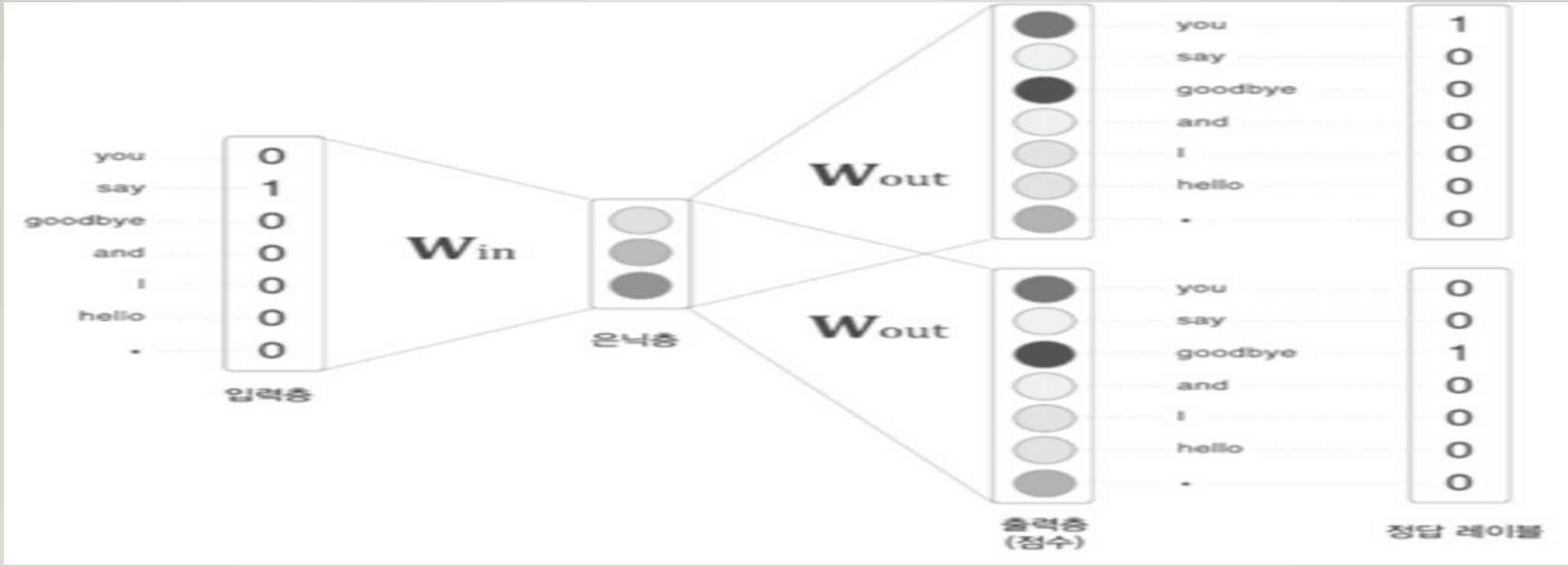


- 맥락 Wt-1과 Wt+1이 주어졌을 때 타깃이 wi가 될 확률: $P(w_t|w_{t-1}, w_{t+1})$
- 교차 엔트로피 오차: $L = -\sum_k t_k \log y_k$
- CBOW 모델에 적용(음의 로그 가능도): $L = -\log P(w_t \mid w_{t-1}, w_{t+1})$
- 말뭉치 전체로 확장: $L = -\frac{1}{T}\sum_{t=1}^{T} \log P(w_t \mid w_{t-1}, w_{t+1})$

SKIP-GRAM 모델

- CBOW
  - 맥락이 여러 개가 있고, 그 여러 맥락으로부터 **중앙의 단어(타깃)를 추측**
- Skip-gram
  - 중앙의 단어(타깃)로부터 **주변의 여러 단어(맥락)을 추측**

SKIP-GRAM 모델의 구성



SKIP-GRAM 모델을 확률로

- Skip-gram 모델의 확률: $P(w_{t-1}, w_{t+1} \mid w_t)$
- 맥락의 단어들 사이에 관련성이 없다고 가정 후 분해

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

- 교차 엔트로피 오차에 적용해 손실 함수 유도

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

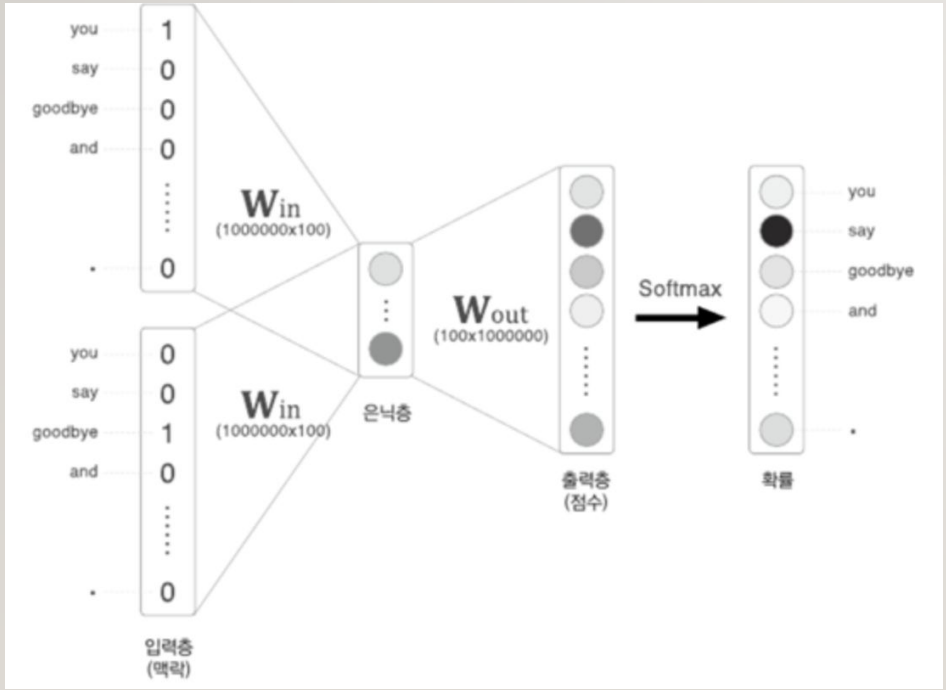
- 말뭉치 전체로 확장: $L = -\frac{1}{T} \sum_{t=1}^{T} (\log P(w_{t-1} \mid w_t) + \log P(w_{t+1} \mid w_t))$

통계 기반 VS 추론 기반

- **학습하는 틀**
  - 통계 기반: 말뭉치의 **전체 통계로부터 1회** 학습하여 단어의 분산 표현을 얻음
  - 추론 기반: 말뭉치를 **일부분씩 여러 번** 보며 학습(미니배치 학습)
- **단어의 분산 표현을 갱신해야 하는 상황(새 어휘 추가)**
  - 통계 기반 기법: 단어의 분산 표현을 조금만 수정하고 싶어도, 동시발생 행렬을 다시 만들고 SVD를 수행하는 **일련의 작업을 다시 해야 함.**
  - 추론 기반 기법: 기존에 학습한 경험을 해치지 않으면서 단어의 분산 표현을 효율적으로 갱신 가능
- **단어의 분산 표현의 성격이나 정밀도**
  - 통계 기반 기법: 분산표현이 주로 **단어의 유사성**이 인코딩됨.
  - 추론 기반 기법: 분산표현이 **복잡한 단어 사이의 패턴**까지도 파악되어 인코딩.

WORD2VEC 속도 개선

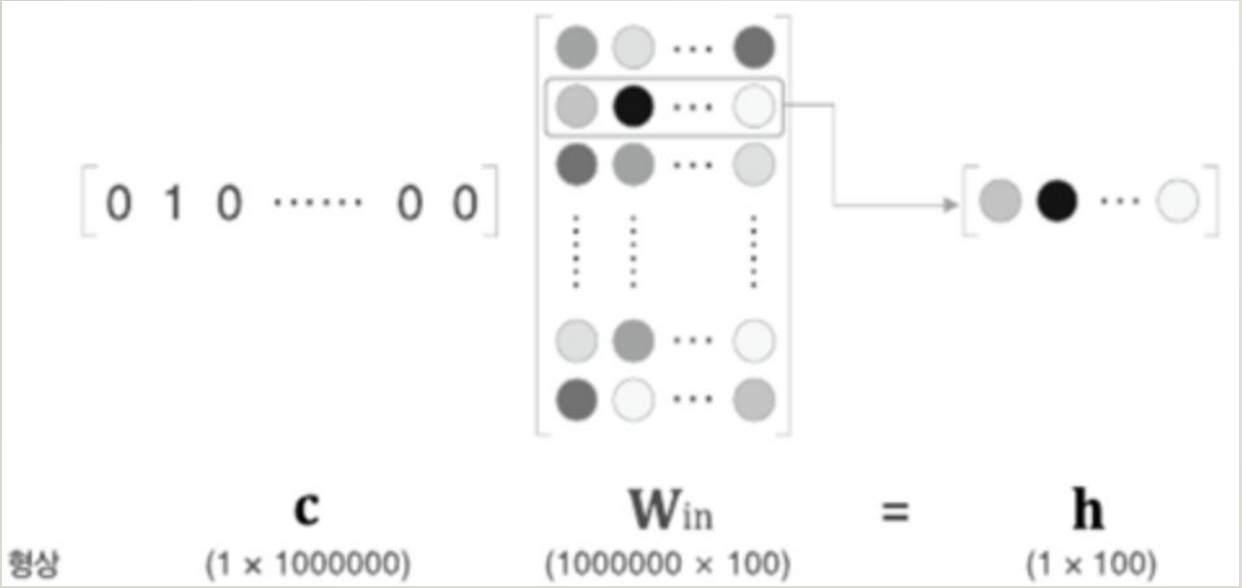
- 어휘가 **100**만 개일 때를 가정한 **CBOW** 모델



- **두 계산에서 병목 발생**
  - 입력층의 윗핫 표현과 가중치 행렬(W_in)의 곱 계산
  - 은닉층과 가중치 행렬(W_out)의 곱 및 Softmax 계층의 계산

WORD2VEC 속도 개선 I

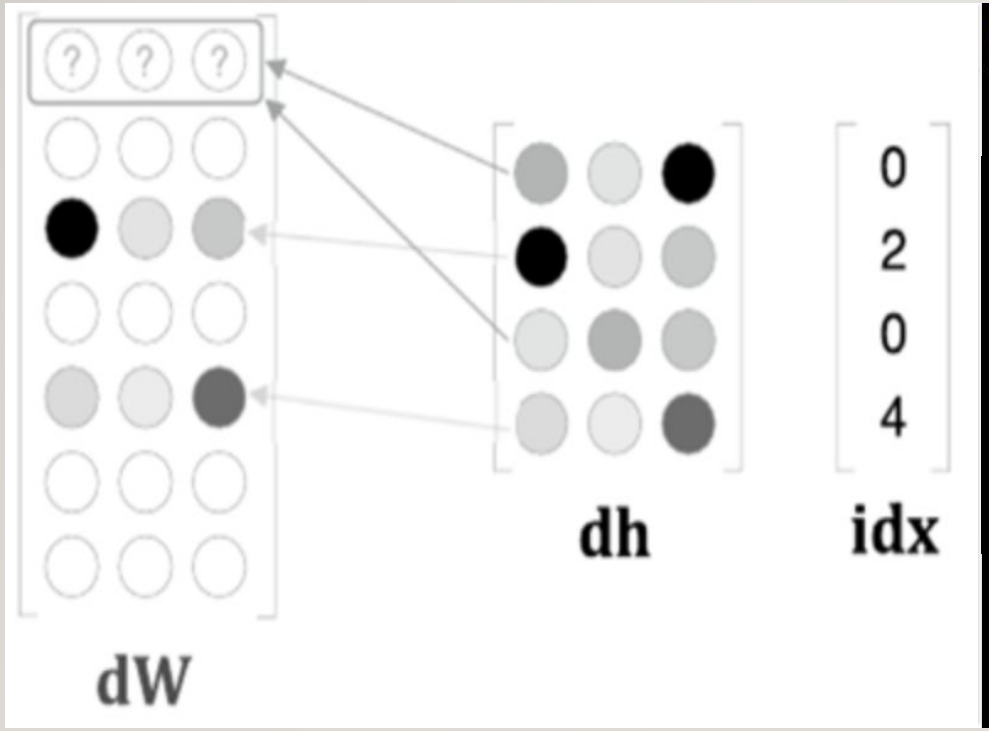
- 기존 방법



EMBEDDING 계층 구현(순전파)


```python
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
```

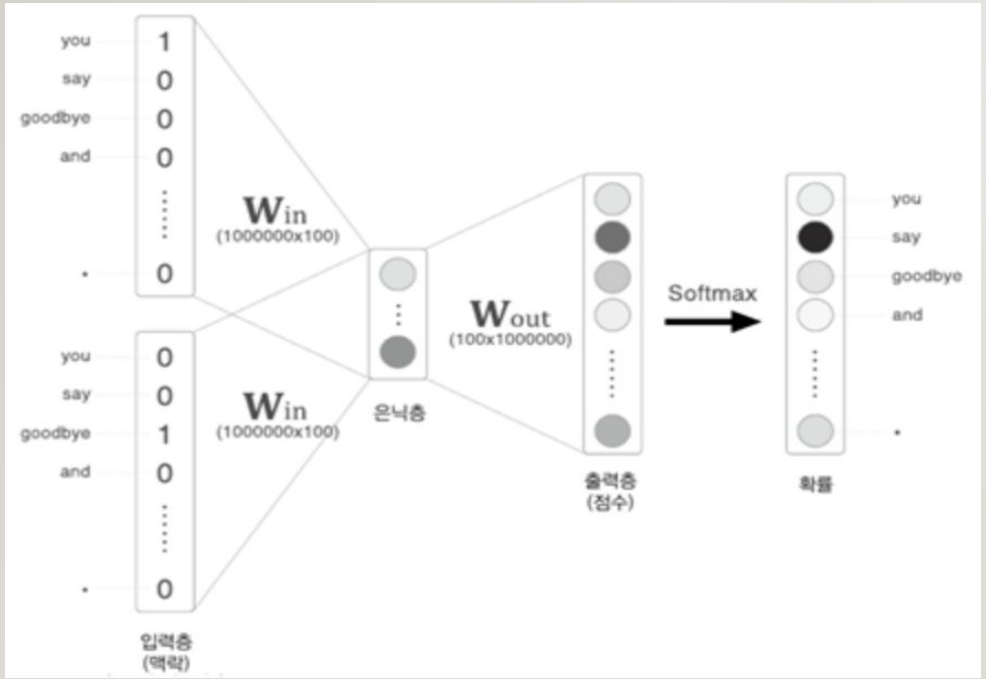
EMBEDDING 계층 구현(역전파)




```python
def backward(self, dout):
    dW = self.grads
    dW[...] = 0  # dW 원소를 0으로 덮어씀

    for i, word_id in enumerate(self, idx):
        dW[word_id] += dout[i]

    # np.add.at(dW, self.idx, dout)

    return None
```

WORD2VEC 속도 개선2

- 은닉층 이후 계산의 문제점

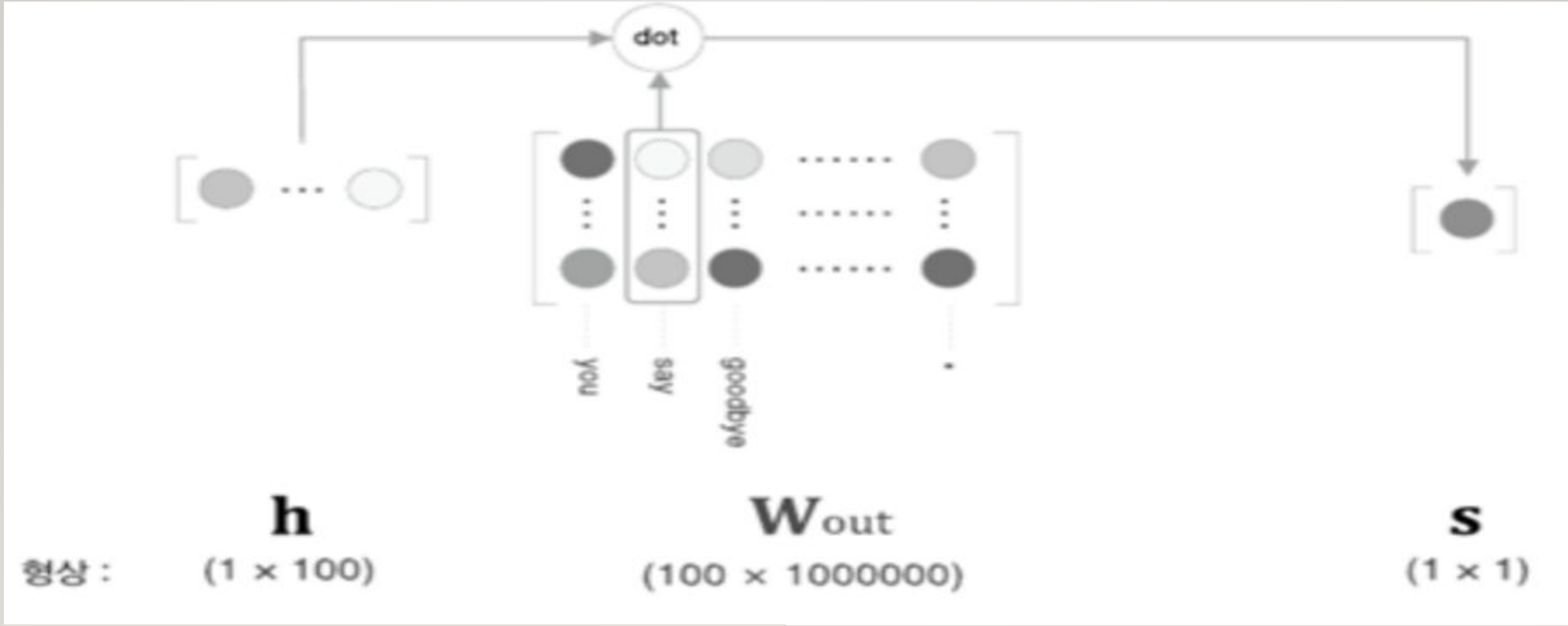


- 은닉층의 뉴런과 가중치 행렬(W_out)의 곱
  - 큰 행렬을 곱하려면 시간이 오래 걸리고, 메모리도 많이 필요
- Softmax 계층의 계산

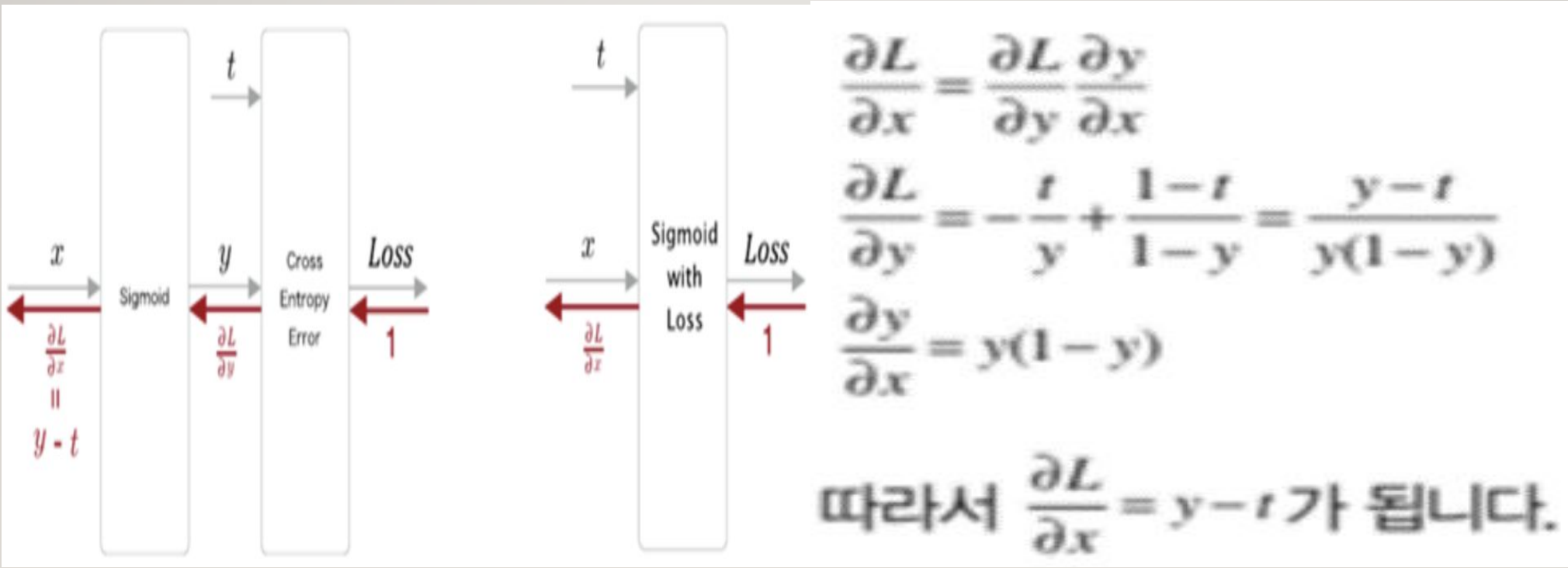
$$y_k = \frac{\exp(s_k)}{\sum_{i=1}^{1000000} \exp(s_i)}$$

  - 어휘 수를 100만 개로 가정한다면 exp 계산을 100만 번 수행

다중 분류에서 이진 분류로



시그모이드 함수와 교차 엔트로피



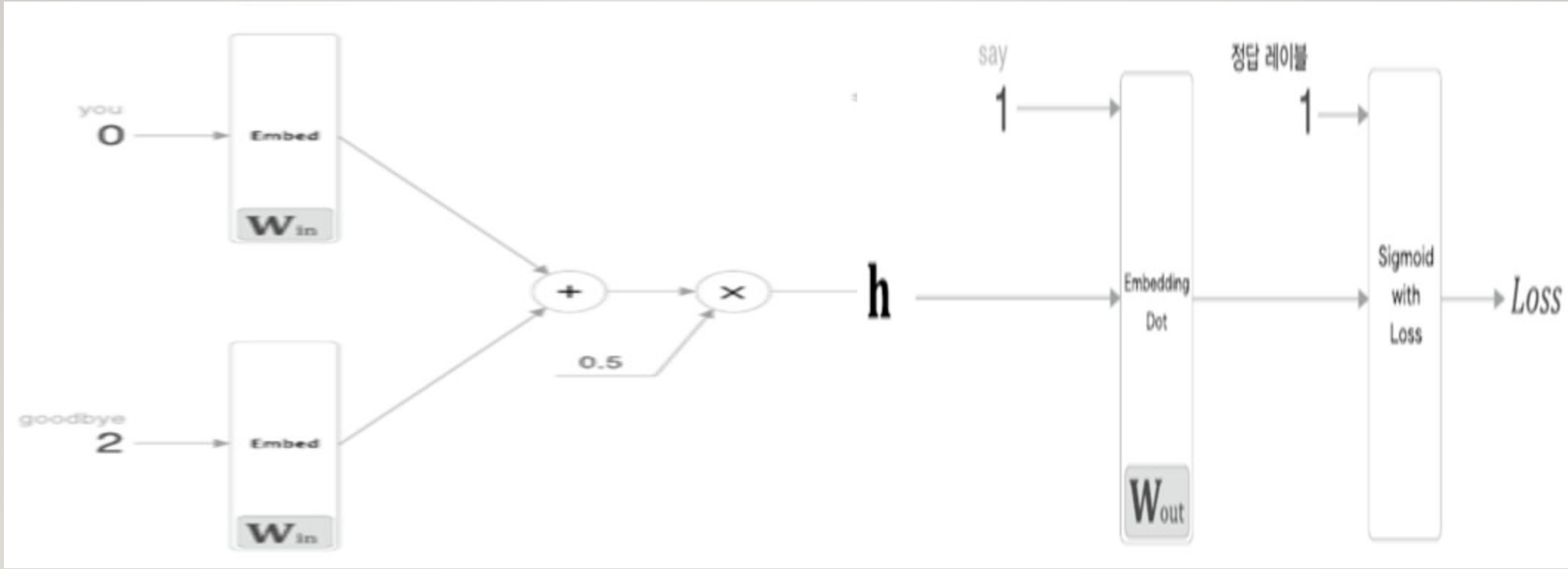
$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x}$$

$$\frac{\partial L}{\partial y} = -\frac{t}{y} + \frac{1-t}{1-y} = \frac{y-t}{y(1-y)}$$

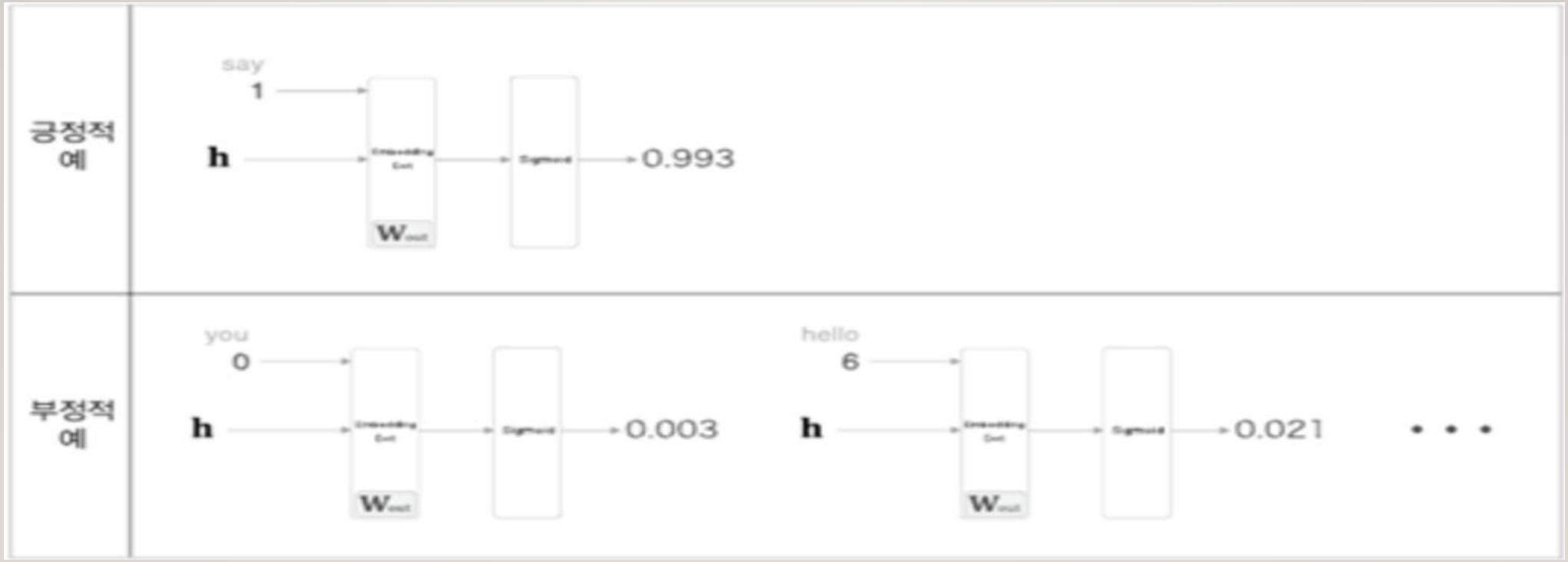
$$\frac{\partial y}{\partial x} = y(1-y)$$

따라서 $\frac{\partial L}{\partial x} = y - t$ 가 됩니다.

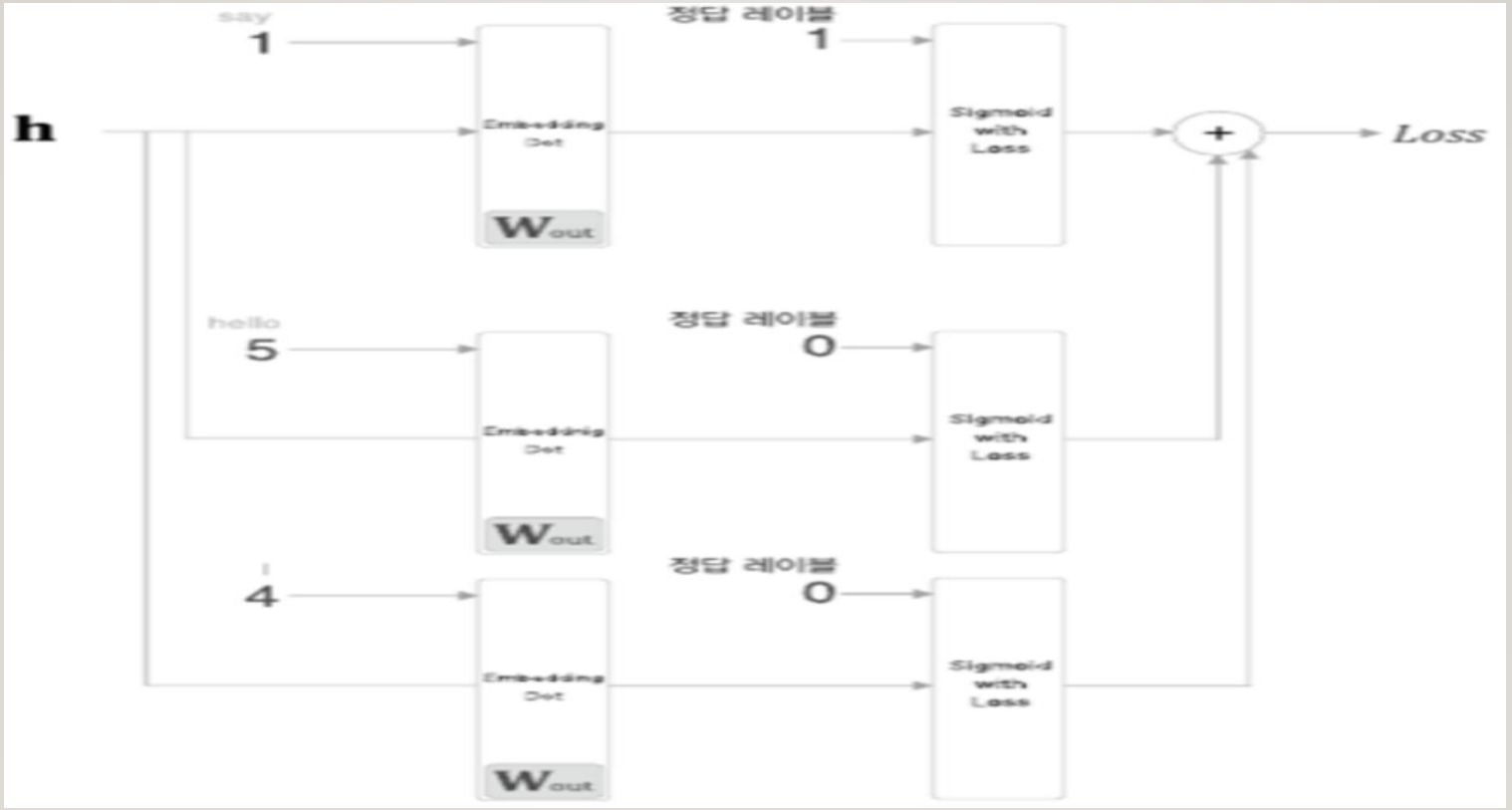
다중 분류에서 이진 분류로(구현)



네거티브 샘플링 I



네거티브 샘플링2


say
1
정답 레이블
1
h
Embedding Dot
Sigmoid with Loss
+
Loss
W_{out}
hello
5
정답 레이블
0
Embedding Dot
Sigmoid with Loss
W_{out}
I
4
정답 레이블
0
Embedding Dot
Sigmoid with Loss
W_{out}

샘플링 방법

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

네거티브 샘플링 구현

초기화 함수 순전파 역전파

```python
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)
        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
```

```python
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)
    return loss
```

```python
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)

    return dh
```

개선판 WORD2VEC 학습

가중치 초기화&계층 생성

```python
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):
            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
```

순전파&역전파

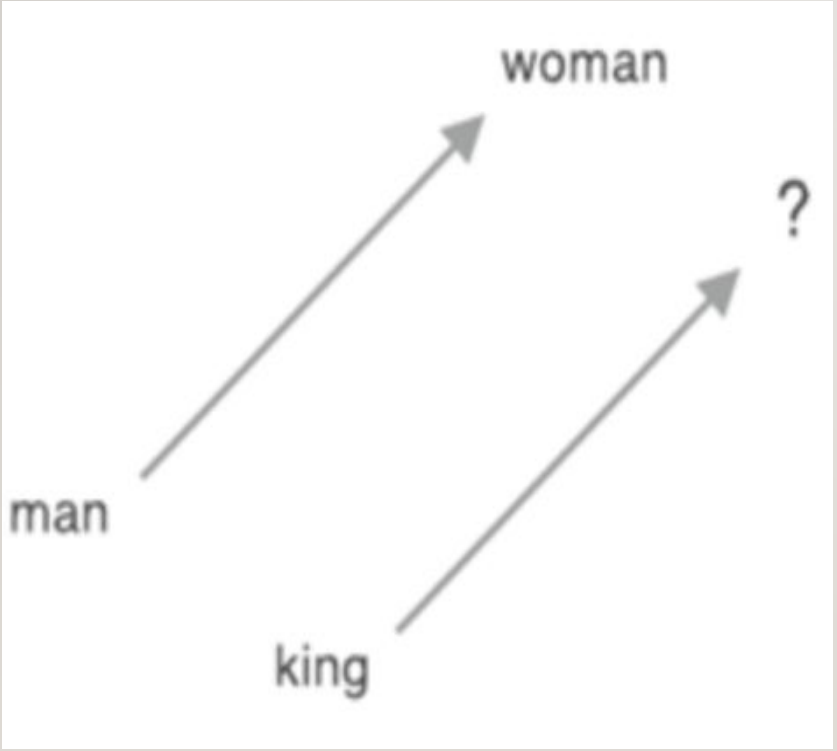
```python
    def forward(self, contexts, target):
        h = 0
        for i, layer in enumerate(self.in_layers):
            h += layer.forward(contexts[:, i])
        h *= 1 / len(self.in_layers)
        loss = self.ns_loss.forward(h, target)
        return loss

    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
```

CBOW 평가

```python
# 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)

# 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)
```

```
[query] you
we: 0.610597074032
someone: 0.591710150242
i: 0.554366409779
something: 0.490028560162
anyone: 0.473472118378

[query] year
month: 0.718261063099
week: 0.652263045311
spring: 0.62699586153
summer: 0.625829637051
decade: 0.603022158146

[query] car
luxury: 0.497202396393
arabia: 0.478033810854
auto: 0.471043765545
disk-drive: 0.450782179832
travel: 0.40902107954

[query] toyota
ford: 0.550541639328
instrumentation: 0.510020911694
mazda: 0.49361255765
bethlehem: 0.474817842245
nissan: 0.474622786045
```

유추 문제 풀기



- 표현 가정: “vec (‘man’)”: “man”의 분산 표현
- **“vec (‘king’) + vec (‘woman’) -vec (‘man’) = vec (?)”**
- analogy('man', 'king', 'woman', word_to_id, id_to_word, word_vecs, top=5)

```
[analogy] man:king = woman:?
word1:  5.003233
word2:  4.400302
word3:  4.22342
word4:  4.003234
word5:  3.934550
```

WORD2VEC 남은 주제

- 자연어 처리에서 단어의 분산 표현이 중요한 이유: **전이 학습**
- 자연어 문제를 풀 때 word2vec의 단어 분산 표현을 **처음부터 학습하는 일은 거의 없음**
  - 먼저 큰 말뭉치(위키백과나 구글 뉴스의 텍스트 데이터 등)로 학습을 끝낸 후, 그 분산 표현을 각자의 작업에 이용
- 단어나 문장을 **고정 길이 벡터로 변환할** 수 있다는 점은 매우 중요
  - 자연어를 벡터로 변환할 수 있다면 일반적인 **머신러닝 기법 (신경망이나 SVM 등)을 적용할 수 있음**

단어 벡터 평가 방법

- **유사성**
  - 단어의 유사성 평가에서는 사람이 작성한 단어 유사도를 검증 세트를 사용해 평가하는 것이 일반적(사람이 단어 사이의 유사한 정도를 규정함)
  - 사람이 부여한 점수와 word2vec에 의한 코사인 유사도 점수를 비교해 그 상관성을 확인
- **유추 문제**
  - 유추 문제를 출제하고, 그 정답률로 단어의 분산 표현의 우수성을 측정하는 것
  - 유추 문제에 의한 평가가 높다고 앱에서도 반드시 좋은 결과가 나온다는 보장은 없음