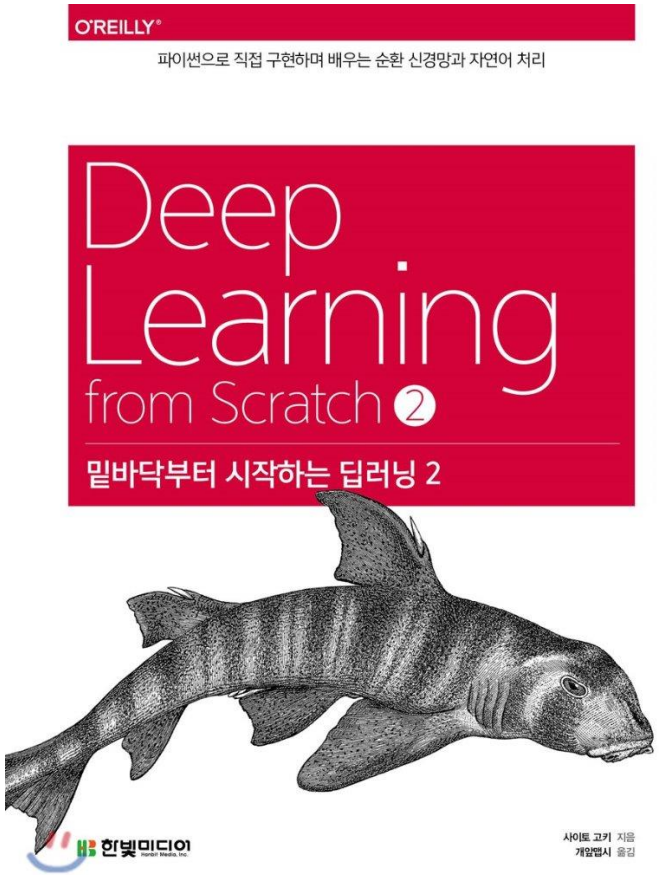
밑바닥부터 시작하는 딥러닝 ②

파이썬으로 직접 구현하며 배우는 순환신경망과 자연어처리



3장 word2vec

CHAPTER 3

word2vec

근거 없이 추론하는 건 금물이야.

– 코넌 도일, <셜록 홈즈의 모험(보헤미아 왕국의 스캔들)>

앞 장에 이어 이번 장의 주제도 단어의 분산 표현이다. 앞 장에서는 '통계 기반 기법'으로 단어의 분산 표현을 얻었는데,
이번 장에서는 더 강력한 기법인 '추론 기반 기법'을 살펴볼 것이다.

이름에서 알 수 있듯이 '추론 기반 기법'은 추론을 하는 기법이다. 이 추론 과정에서 신경망(Neural Network)를 이용하는데,
여기서 그 유명한 word2vec이 등장한다. 이번 장에서는 word2vec의 구조를 차분히 들여다보고 구현해보며 확실하게 이해하는 것을 목표로 한다.

이번 장의 목표는 '단순한' word2vec 구현하기다. 처리 효율을 희생하고 이해하기 쉽도록 구성을 할 것이다.
따라서 큰 데이터셋은 어렵겠지만, 작은 데이터셋이라면 문제없이 처리할 수 있다.

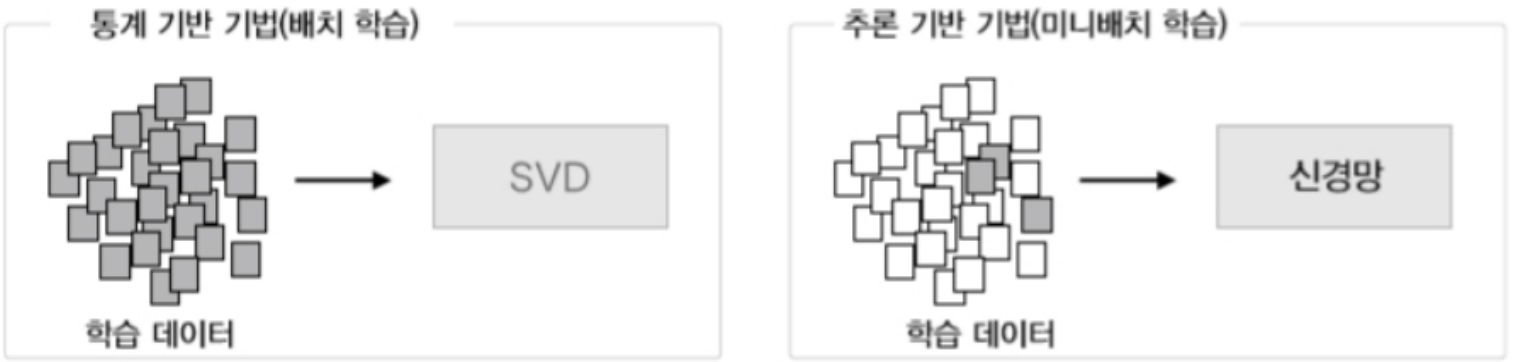
추론 기반 기법과 신경망

▪ 통계 기반 기법의 문제점

앞에서 본 통계 기반 기법 같은 경우는 주변 단어의 빈도를 기초로 단어를 표현했다.
구체적으로는 단어의 **동시발생 행렬**을 만들고 그 행렬에 **SVD**(벡터 차원 수 감소)를 적용하여 밀집벡터를 얻었다.
그러나 이 방식은 $O(n^3)$의 시간복잡도를 가지기 때문에, 대규모 말뭉치를 다룰 때 문제가 발생한다. (현실적으로 불가능)

통계 기반 기법은 말뭉치 전체의 통계를 이용하여 **단 1회의 처리**만에 단어의 분산 표현을 얻는다.
반면, 추론 기반 기법에서는 **미니배치로 학습**하는 것이 일반적이다. 미니배치 학습이란 신경망이 한 번에 소량의
학습 샘플씩 반복해서 가중치를 갱신해 나가는 방식을 말한다.

그림 3-1 통계 기반 기법과 추론 기반 기법 비교



[그림 3-1]처럼 통계 기반 기법은 학습 데이터를 한 번에 처리하는데 반해, 추론 기밥 기법은 일부만을 뽑아서 학습하므로,
데이터아 큰 작업도 신경망을 학습시킬 수 있다. 게다가 여러 GPU를 이용한 병렬 계산도 가능해서 학습 속도도 높일 수 있다.

추론 기반 기법과 신경망

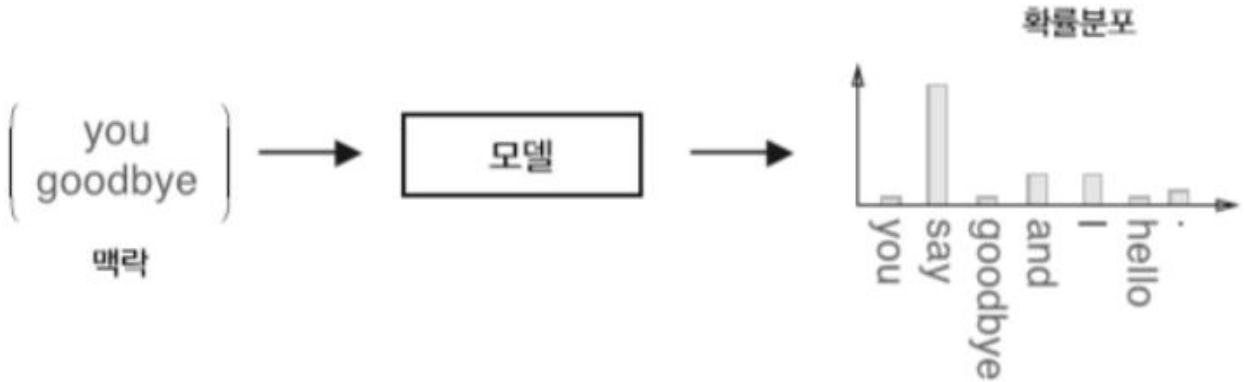
▪ 추론 기반 기법 개요

추론 기반 기법에서는 당연히 '추론'이 주된 작업이다. 여기서 추론이란 [그림 3-2]처럼 주변 단어(맥락)가 주어졌을 때 "**?**"에 무슨 단어가 들어가는지를 추측하는 작업이다.

그림 3-2 주변 단어들을 맥락으로 사용해 "?"에 들어갈 단어를 추측한다

[그림 3-2]처럼 추론 문제를 풀고 학습하는 것이 '추론 기반 기법'이 다루는 문제이다. 이러한 추론 문제를 반복해서 풀면서 단어의 출현 패턴을 학습하는 것이다. '모델 관점'에서 보면, 이 추론 문제는 [그림 3-3]처럼 보이게 된다.

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



[그림 3-3]처럼 추론 기반 기법에는 어떠한 모델이 등장한다.
우리는 이 모델로 신경망을 사용한다.
모델은 **맥락 정보**를 입력받아 각 단어의 출현 확률을 출력한다.

NOTE_ 추론 기반 기법도 **분포 가설**에 기초한다.
분포 가설이란 '단어의 의미는 주변 단어에 의해 형성된다'는 가설이다.

추론 기반 기법과 신경망

▪ 신경망에서의 단어 처리

지금부터 신경망을 이용해 '단어'를 처리해본다. 일단 "you"와 "say" 등의 단어를 있는 그대로 처리할 수는 없으니 '고정 길이의 벡터'로 변환해야 한다. 이때 사용하는 대표적인 방법이 **원핫 벡터**로 변환하는 것이다. 원핫 벡터란 벡터의 원소 중 하나만 1이고 나머지는 모두 0인 벡터를 말한다.

그림 3-4 단어, 단어 ID, 원핫 벡터

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

단어의 원핫 벡터화는 다음과 같은 순서로 진행된다.

① 총 어휘 수만큼의 원소를 갖는 벡터를 준비한다.

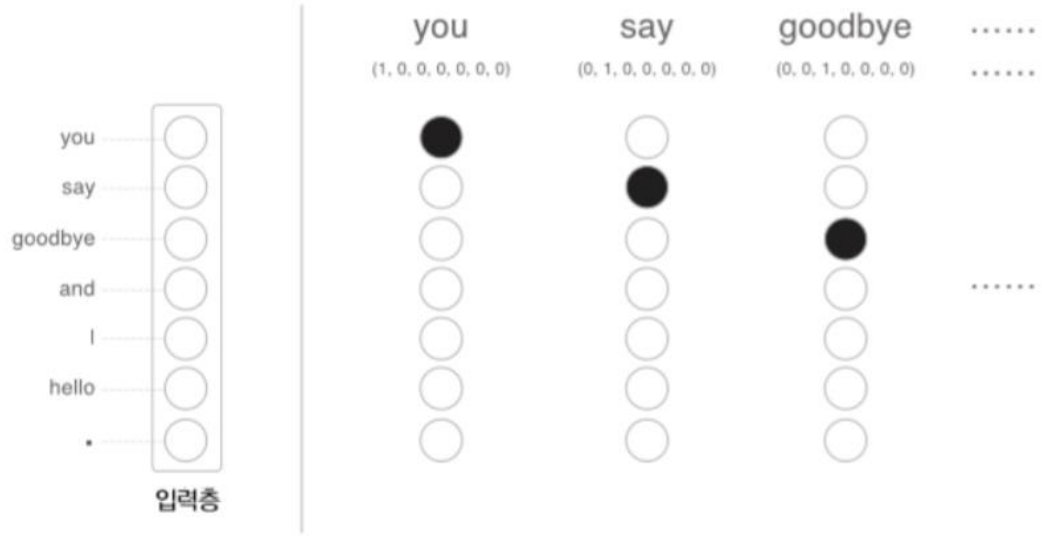
② 어휘를 등장한 차례차례 인덱싱을 하여 단어 ID를 만든다

③ 단어 ID에 해당하는 인덱스만 1로 표현을 하고 나머지는 0으로 표현한다.

추론 기반 기법과 신경망

▪ 신경망에서의 단어 처리

그림 3-5 입력층의 뉴런: 각 뉴런이 각 단어에 대응(해당 뉴런이 1이면 검은색, 0이면 흰색)

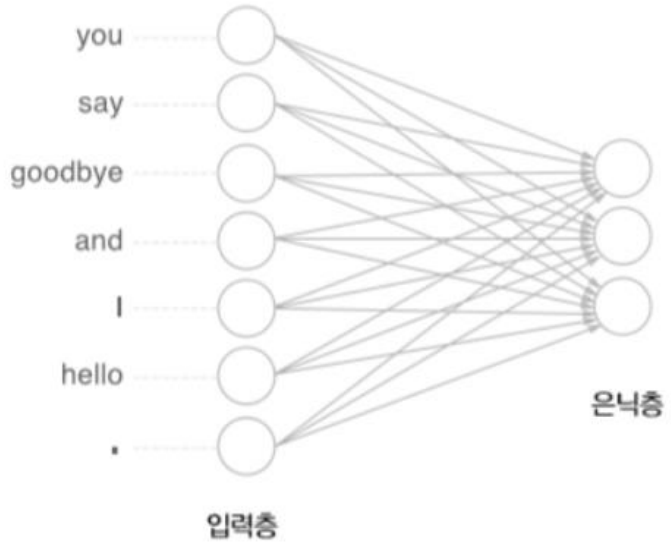


앞에서처럼 **원핫 벡터**로 표현하면 [그림 3-5]처럼 뉴런의 수를 '고정'할 수 있다.
이제 단어를 벡터로 나타낼 수 있고, 신경망을 구성하는 '계층'들은 벡터를 처리할 수가 있다.
다시 말해, 단어를 신경망으로 처리할 수 있다는 뜻이다.

추론 기반 기법과 신경망

▪ 신경망에서의 단어 처리

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

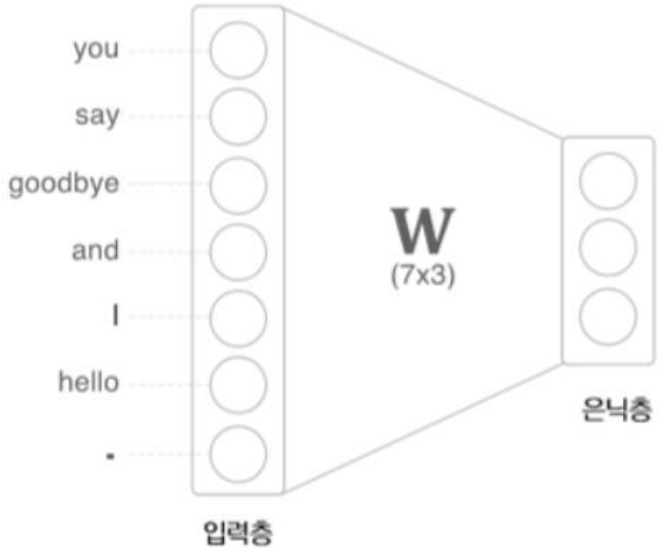


[그림 3-6]은 원핫 벡터로 된 단어 하나를 완전연결계층을 통해 변환하는 모습을 보여준다.
완전연결계층이므로 각각의 노드가 이웃 층의 모든 노드와 화살표로 연결되어 있다.
그리고 이 화살표에는 가중치 (매개변수)가 존재하며, 입력층 뉴런과의 가중합weight sum이 은닉층 뉴런이 된다.

추론 기반 기법과 신경망

▪ 신경망에서의 단어 처리

그림 3-7 완전연결계층에 의한 변환을 단순화한 그림(완전연결계층의 가중치르 7X3 크기의 **W**라는 행렬로 표현)



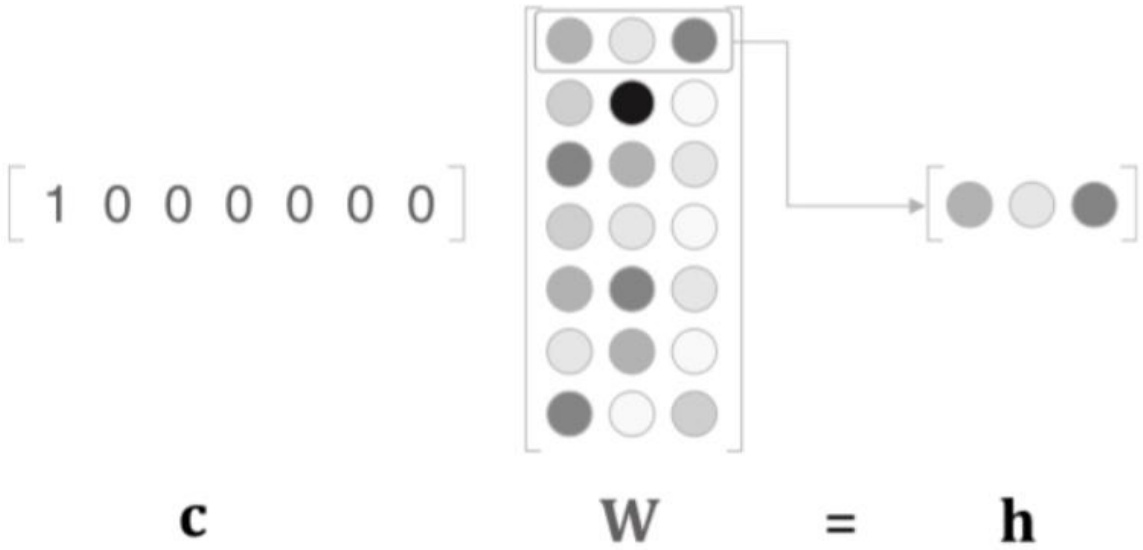

```
import numpy as np

c = np.array([[1, 0, 0, 0, 0, 0, 0]]) # 입력
W = np.random.randn(7, 3)             # 가중치
h = np.matmul(c, W)                   # 중간 노드
print(h)
# [[-0.70012195  0.25204755 -0.79774592]]
```

[그림 3-7]은 완전연결계층에 의한 변환을 단순하게 표현한 그림이다 이를 파이썬 코드로 표현하면 다음과 같이 작성할 수 있다. 이 코드는 단어 ID가 0인 단어를 원핫 벡터로 표현한 다음 완전연결계층을 통과시켜 변환하는 모습을 보여준다.

WARNING_ 이 코드에서 입력 데이터©의 차원 수(ndim)은 2이다. 이는 미니배치 처리를 고려한 것으로, 최초의 차원(0번째 차원)에 각 데이터를 저장한다.

▪ 신경망에서의 단어 처리

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



앞의 코드에서 주목할 것은 c와 W의 행렬 곱 부분이다. C는 어차피 원핫 벡터이므로 하나의 원소만 1이고 나머지는 0이다. 결국 [그림 3-8]처럼 **가중치의 행벡터 하나만을 뽑아낸 것**과 같다. 겨우 행벡터 하나를 뽑아낼 뿐인데 행렬 곱을 계산하는 것은 비효율적이다. 이 부분은 다음 'CHAPTER4: word2vec 개선'에서 개선할 예정이다.

단순한 word2vec

▪ CBOW 모델의 추론 처리

이제 word2vec을 구현할 차례이다. 지금부터 할 일은 [그림 3-3]의 '모델'을 신경망으로 구축하는 것이다. 그리고 이번 절에서 사용할 신경망은 word2vec에서 제안하는 **CBOW**continuousbag-of-words 모델이다.

CBOW 모델은 맥락으로부터 타깃target을 추측하는 용도의 신경망이다. ('타깃'은 중앙단어이고 그 주변 단어들이 '맥락'이다) **CBOW 모델의 입력은 맥락**이다. 즉, 맥락은 "you"와 "goodbye" 같은 단어들의 목록이다.

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

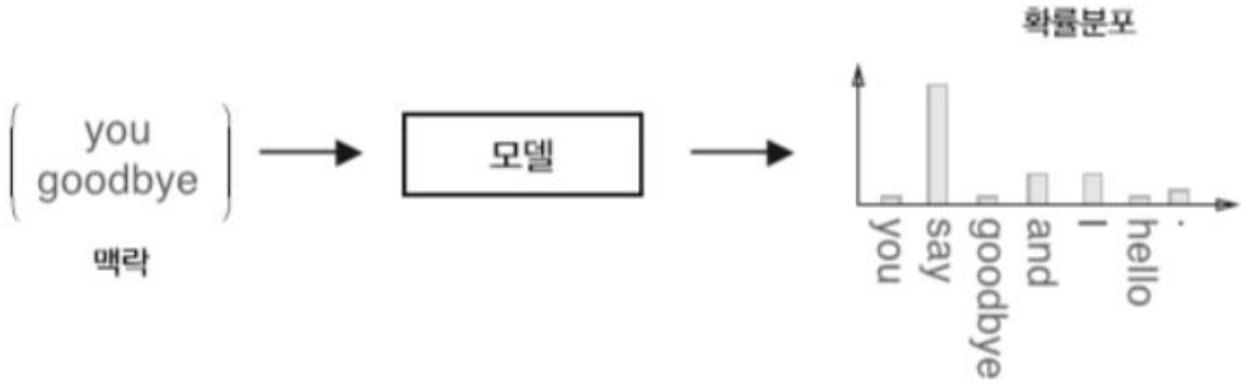
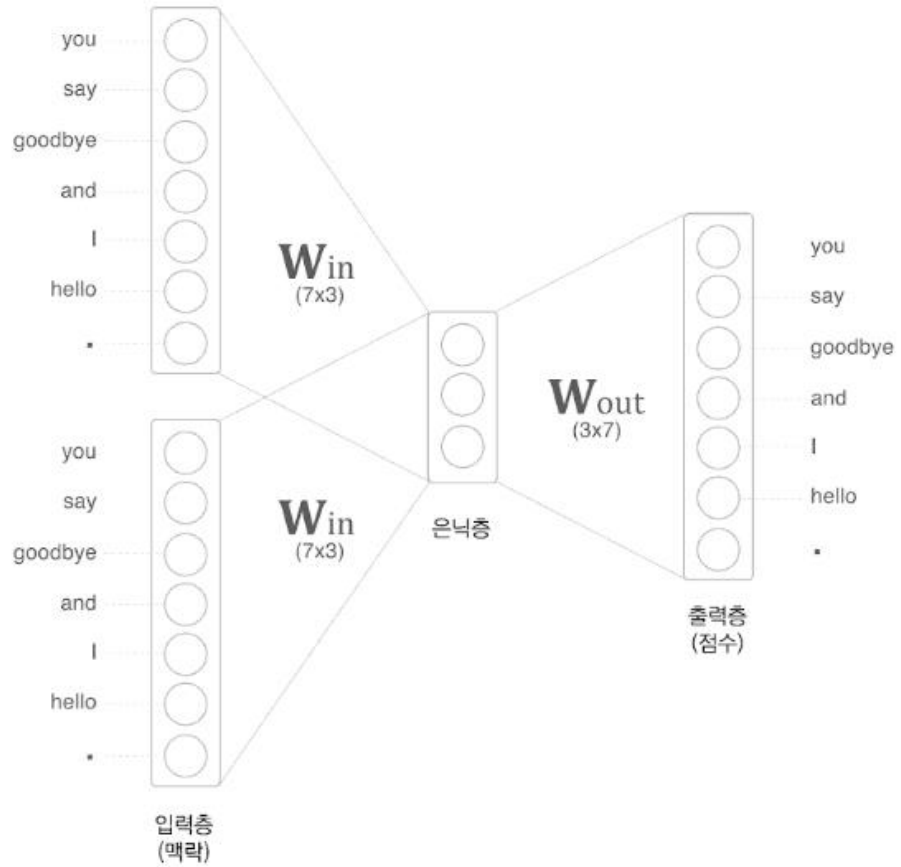


그림 3-2 주변 단어들을 맥락으로 사용해 "?"에 들어갈 단어를 추측한다

단순한 word2vec

- **CBOW 모델의 추론 처리**

그림 3-9 CBOW 모델의 신경망 구조



[그림 3-9]가 CBOW 모델의 신경망이다. 입력층이 2개 있고, 은닉층을 거쳐 출력층에 도달한다. 두 입력층에서 은닉층으로의 변환은 똑같은 완전연결계층 (가중치는 $\mathbf{W}_{in}$)이 처리한다. 그리고 은닉층에서 출력층 뉴런으로의 변환은 다른 완전연결계층 (가중치는 $\mathbf{W}_{out}$)이 처리한다.

WARNING_ 이 그림에서 입력층이 2개인 이유는 맥락으로 고려할 단어를 앞, 뒤 한 단어씩해서 2개의 단어로 정표했기 때문이다. 즉, 맥락에 포함시킬 단어가 N개라면 입력층도 N개가 된다.%

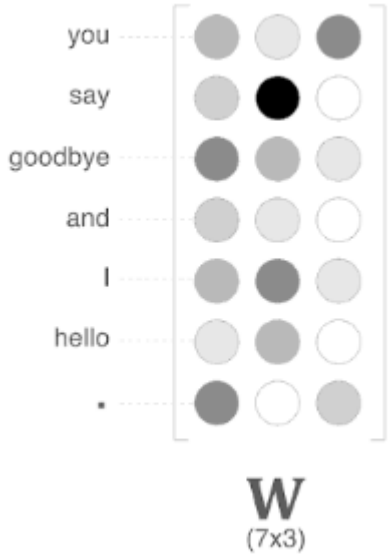
이제 [그림 3-9]의 은닉층에 주목해보자. 은닉층의 뉴런은 입력층의 완전연결계층에 의해 변환된 값이 되는데, 입력층이 여러 개이면 전체를 '평균'해주면 된다. 첫 번째 입력층이 $\mathbf{h}_1$, 두 번째 입력층이 $\mathbf{h}_2$로 변환되었다고 하면, 은닉층 뉴런은 $\frac{1}{2}(\mathbf{h}1+\mathbf{h}_2)$가 된다.

이제 출력층을 보게 되면, 출력층뉴런은 각 단어의 '점수'를 뜻하며, 점수가 높을수록 대응 단어의 출현 확률이 높아진다.

단순한 word2vec

▪ CBOW 모델의 추론 처리

그림 3-10 가중치의 각 행이 해당 단어의 분산 표현이다.



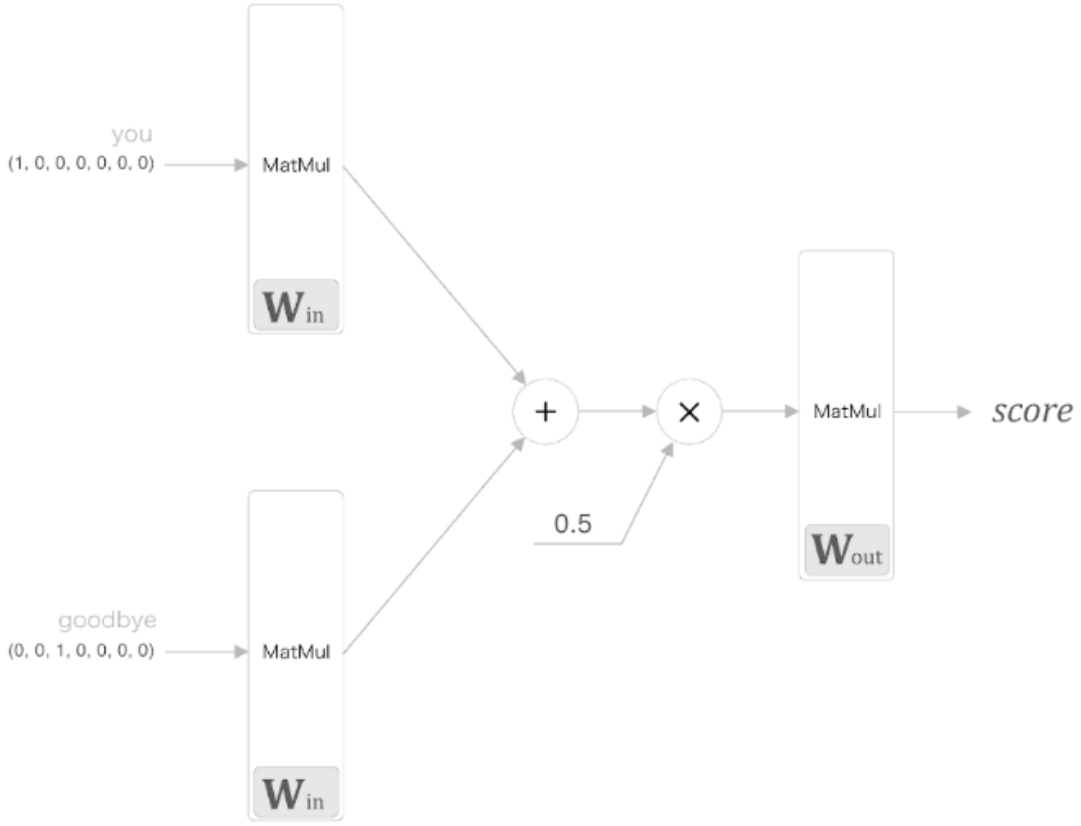
앞에서 원핫 벡터로 변환한 점과, 행렬곱의 특성을 고려해보면, 가중치 $\mathbf{W}_{in}$이 [그림 3-10]과 같이 단어의 분산표현의 정체라는 점을 알 수 있다. $\mathbf{W}_{in}$에는 학습을 진행해가면서 맥락에서 출현하는 단어를 추측하여 해당 단어의 점수가 되는 것이다.
그리고 놀랍게도 이렇게 얻은 벡터에는 '단어의 의미'도 잘 녹아들어 있다!

NOTE_ 은닉층의 뉴런 수를 입력층의 뉴런 수보다 적게 하는 것이 중요한 핵심이다. 이렇게 해야 단어 예측에 필요한 정보를 '간결하게' 담게 되며, 결과적으로 밀집벡터 표현을 얻을 수 있다.

단순한 word2vec

- **CBOW 모델의 추론 처리**

그림 3-11 계층 관점에서 본 CBOW 모델의 신경망 구성



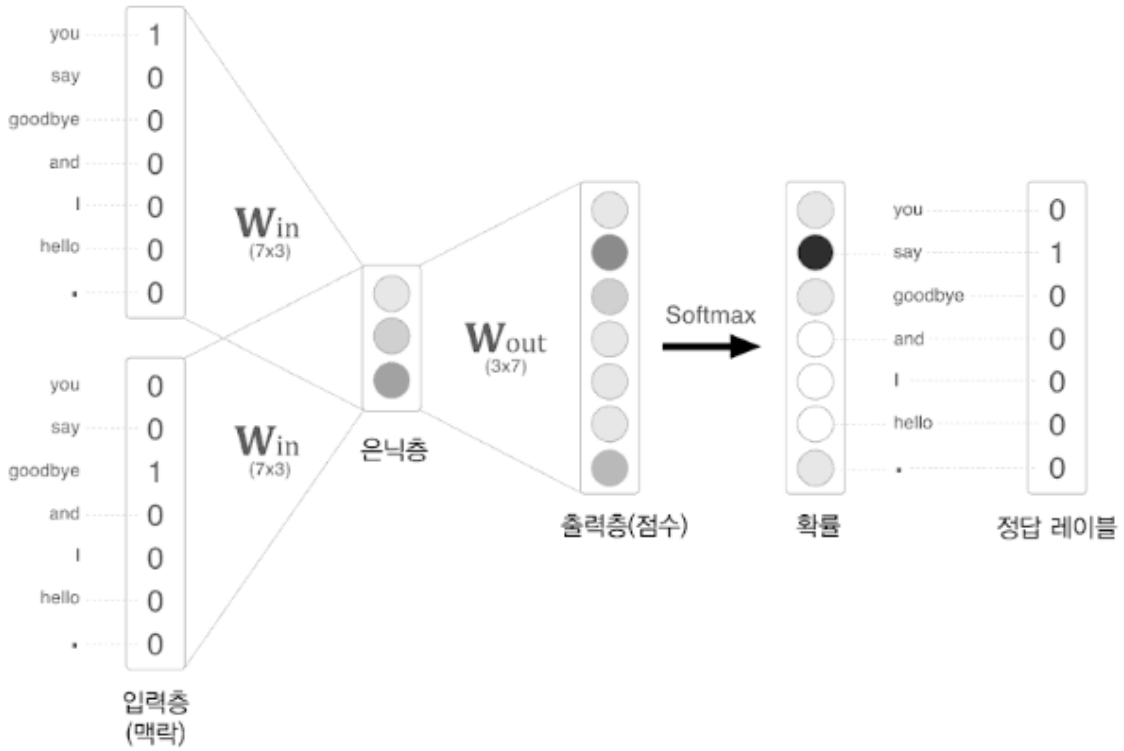

```python
import sys
sys.path.append('..')
import numpy as np
from common.layers import MatMul

# 샘플 맥락 데이터
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)
# [[ 0.30916255  0.45060817 -0.77308656  0.22054131  0.15037278
#   -0.93659277 -0.59612048]]
```

이상이 CBOW 모델의 추론 과정이다. 여기서 보았듯이 CBOW 모델은 **활성화 함수를 사용하지 않는 간단한 구성의 신경망**이다.

단순한 word2vec

▪ CBOW 모델의 학습

그림 3-12 CBOW 모델의 구체적인 예(노드 값의 크기를 흑백의 진하기로 나타냄)



이제 앞의 CBOW 모델에 **소프트맥스** 함수를 적용하면 '확률'을 얻을 수 있다. CBOW 모델의 학습에서는 올바른 예측을 할 수 있도록 가중치를 조정하는 일을 한다. 그 결과로 가중치 $\mathbf{W}_{out}$과 $\mathbf{W}_{in}$ 모두에 단어의 출현 패턴을 파악한 벡터가 학습된다.

그리고 지금까지의 실험에 의해 CBOW 모델로 얻을 수 있는 **단어의 분산 표현**은 단어의 의미 면에서나 문법 면에서 모두 우리의 직관에 부합하는 경우를 많이 볼 수 있다.

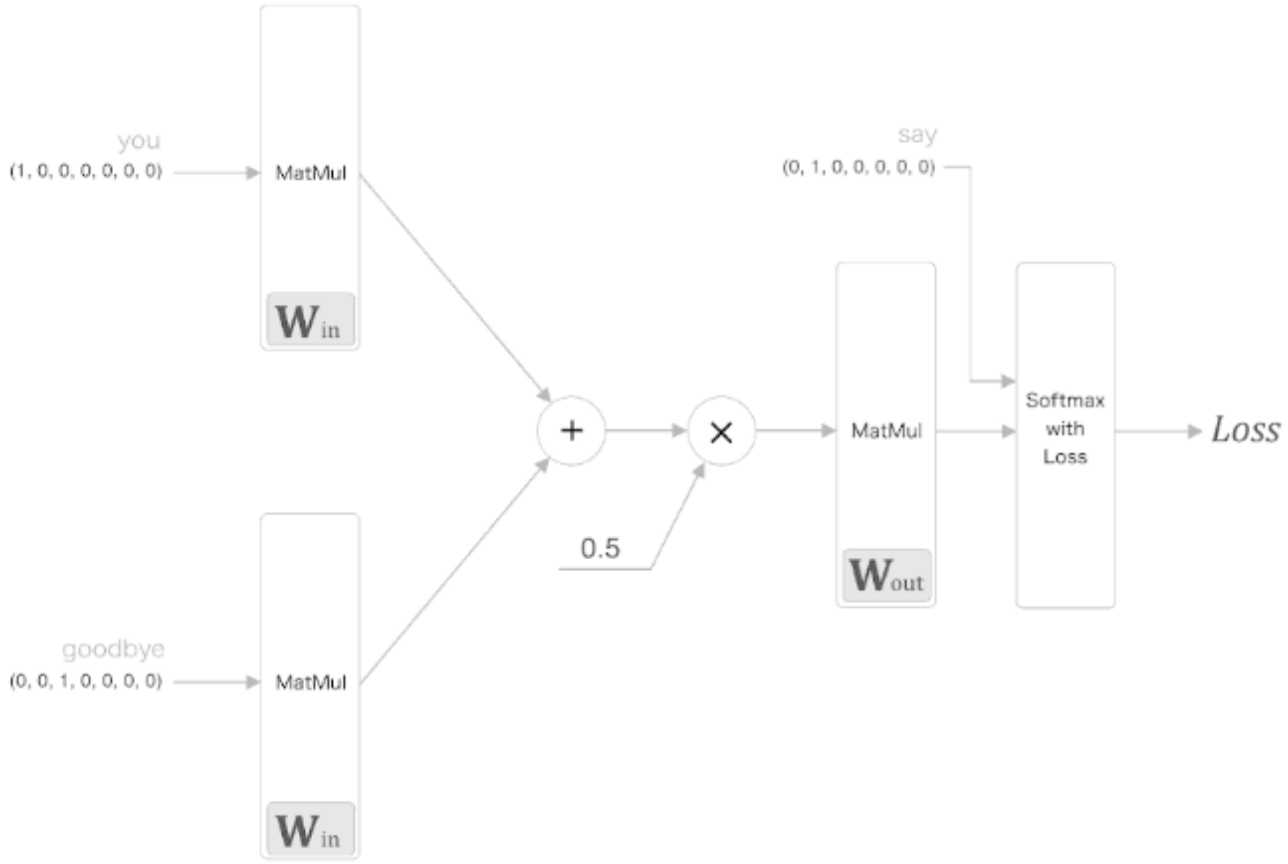
NOTE_ CBOW 모델은 단어 출현 패턴을 학습 시 사용한 말뭉치로부터 배우게 된다. 따라서 어떤 말뭉치로 학습을 했느냐에 따라서 학습 후 얻게 되는 단어의 분산 표현도 달라지게 된다.

이제 앞에서 배운 신경망 개념을 그대로 적용하면 된다.
소프트맥스 함수를 이용해서 확률을 얻었으므로, 그 확률과 정답 레이블로부터 교차 엔트로피 오차를 구한 후, 그 값을 손실로 사용해 학습을 진행하면 된다.

단순한 word2vec

- **CBOW 모델의 학습**

그림 3-14 CBOW 모델의 학습 시 신경망 구성 (Softmax 계층과 Cross Entropy Error 계층을 하나로 합침)

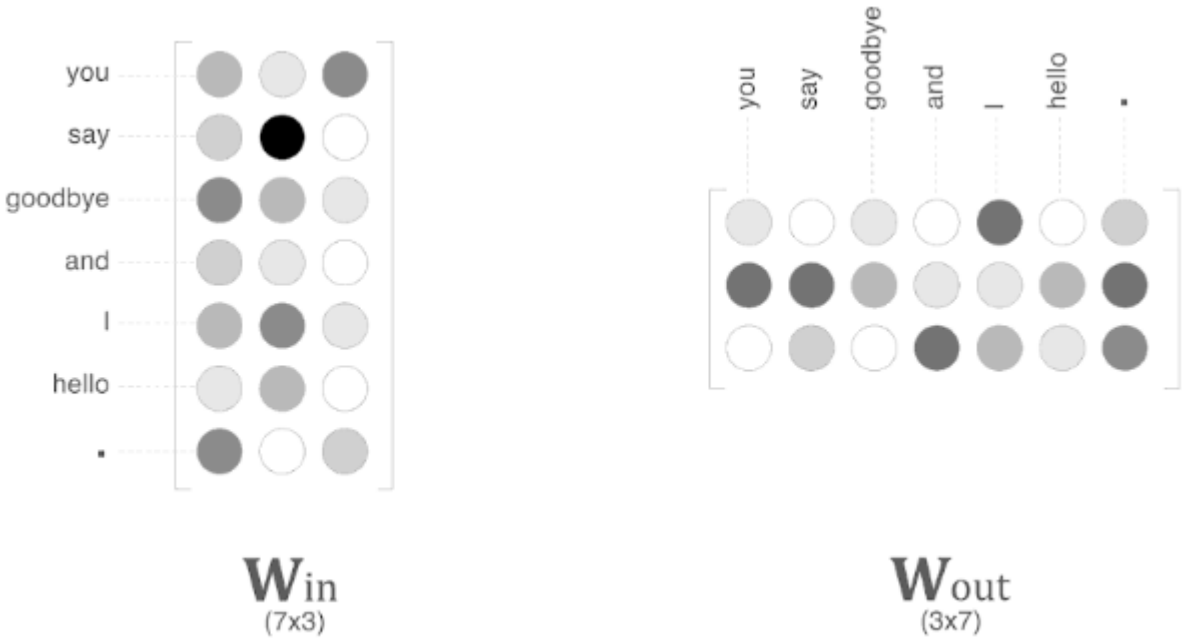


단순한 word2vec

▪ Word2vec의 가중치와 분산 표현

지금까지 설명한 것처럼 word2vec에서 사용되는 신경망에는 두 가지 가중치가 있다. 바로 입력 측 가중치 $\mathbf{W}_{in}$과 출력 측 가중치 $\mathbf{W}_{out}$이다. 그리고 이 둘 모두에 각 단어의 분산 표현이 저장되어 있다고 생각할 수 있다.

그림 3-15 각 단어의 분산 표현은 입력 측과 출력 측 모두의 가중치에서 확인할 수 있다



그러면 최종적으로 이용하는 단어의 분산 표현으로 3가지의 선택지가 있다.

A 입력 측의 가중치만 이용한다.

B 출력 측의 가중치만 이용한다.

C 양쪽 가중치를 모두 이용한다.

이 3가지 선택지 안에서도, C 안에서는 두 가중치를 어떻게 조합하느냐에 따라 다시 몇 가지 방법을 생각해낼 수 있다. 그 중 하나로 단순히 합치는 것이 있을 것이다.

word2vec(특히 skip-gram 모델)에서는 **A**안인 '입력 측의 가중치만 이용한다'가 가장 대중적인 선택이다.

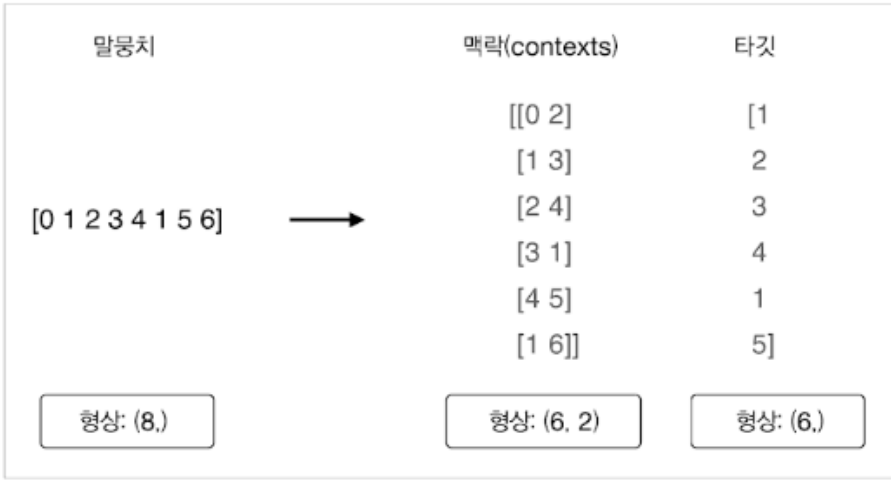
학습 데이터 준비

▪ 맥락과 타깃

word2vec에서 사용하는 **신경망의 입력은 '맥락'**이다. 그리고 그 **정답 레이블은 맥락에 둘러싸인 중앙의 단어, 즉 '타깃'**이다. 우리가 해야 할 일은 신경망에 '맥락'을 입력했을 때 '타깃'이 출현할 확률을 높이는 것이다.

그림 3-16 말뭉치에서 맥락과 타깃을 만드는 예

그림 3-17 단어 ID의 배열인 corpus로부터 맥락과 타깃 작성의 예



[그림 3-16]과 같이 우리 사람의 입장에서 생각하는 말뭉치와 맥락 및 타깃을 [그림 3-17]과 같이 컴퓨터가 이해할 수 있는 말뭉치와 맥락 및 타깃으로 바꾸어 줄 수가 있다.

학습 데이터 준비

- **맥락과 타깃** - 구현

```
import sys
sys.path.append('..')
from common.util import preprocess

text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)
print(corpus)
# [0 1 2 3 4 1 5 6]

print(id_to_word)
# {0: 'you', 1: 'say', 2: 'goodbye', 3: 'and', 4: 'i', 5: 'hello', 6: '.'}
```

```
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)
```

앞 장에서 구현했던 preprocess() 함수를 사용하여, corpu와 word_to_id, id_to_word를 만들어낸다. 그런 다음 단어ID의 배열인 Corpus로부터 맥락과 타깃을 만들어낸다. 구체적으로는 corpu를 주면 맥락과 타깃을 반환하는 함수를 작성한다. (create_contexts_target)

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

print(contexts)
# [[0 2]
#  [1 3]
#  [2 4]
#  [3 1]
#  [4 5]
#  [1 6]]

print(target)
# [1 2 3 4 1 5]
```

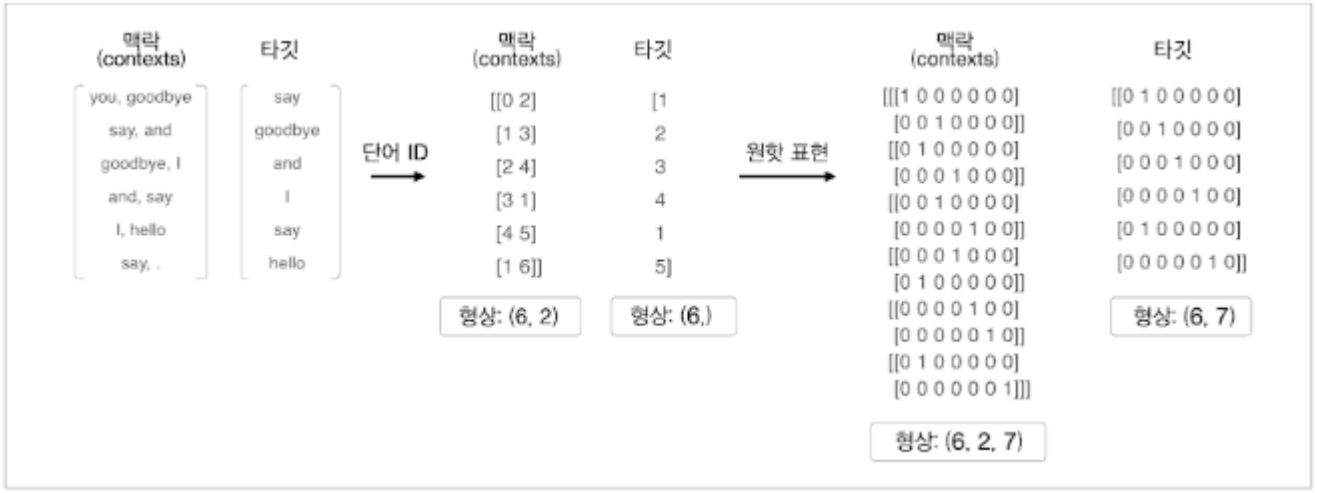
<- 위에 정의한 함수를 바탕으로 실제로 실행해본다

학습 데이터 준비

▪ 원핫 표현으로 변환

계속해서 맥락과 타깃을 원핫 벡터로 바꿔보자.

그림 3-18 맥락과 타깃을 원핫 벡터로 변환하는 예



WARNINGS_ 맥락을 원핫 벡터로 변환한 그림에서 리스트의 형식을 주의해야 한다. 총 3차원으로 [[[0], [2]] , ... [[1], [6]]] 과 같은 형태로 되어 있는 점을 이해해야 한다.

학습 데이터 준비

- **원핫 표현으로 변환**

원핫 표현으로의 변환은 이 책이 제공하는 convert_one_hot() 함수를 사용했다.

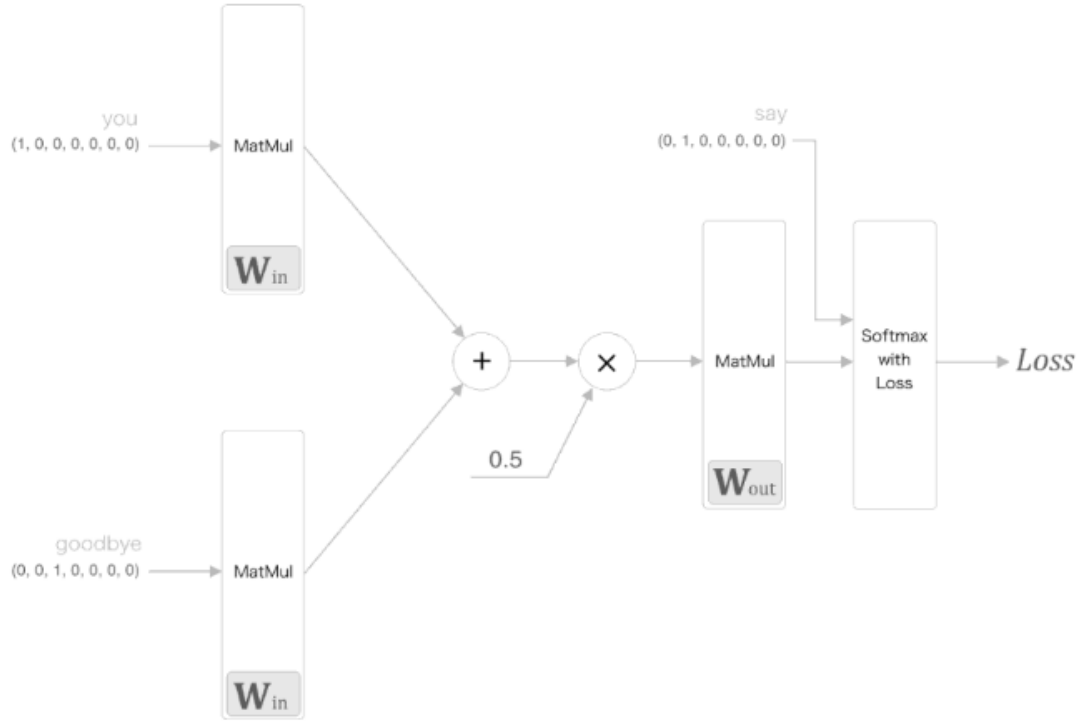
```
import sys
sys.path.append('..')
from common.util import preprocess, create_contexts_target, convert_one_hot

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

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

vocab_size = len(word_to_id)
target = convert_one_hot(target, vocab_size)
contexts = convert_one_hot(contexts, vocab_size)
```

이제 준비를 마쳤으므로, 다음은 본론인 CBOW 모델 구현을 해보자.

CBOW 모델 구현

- **학습 코드 구현**

그림 3-19 CBOW 모델의 신경망 구성

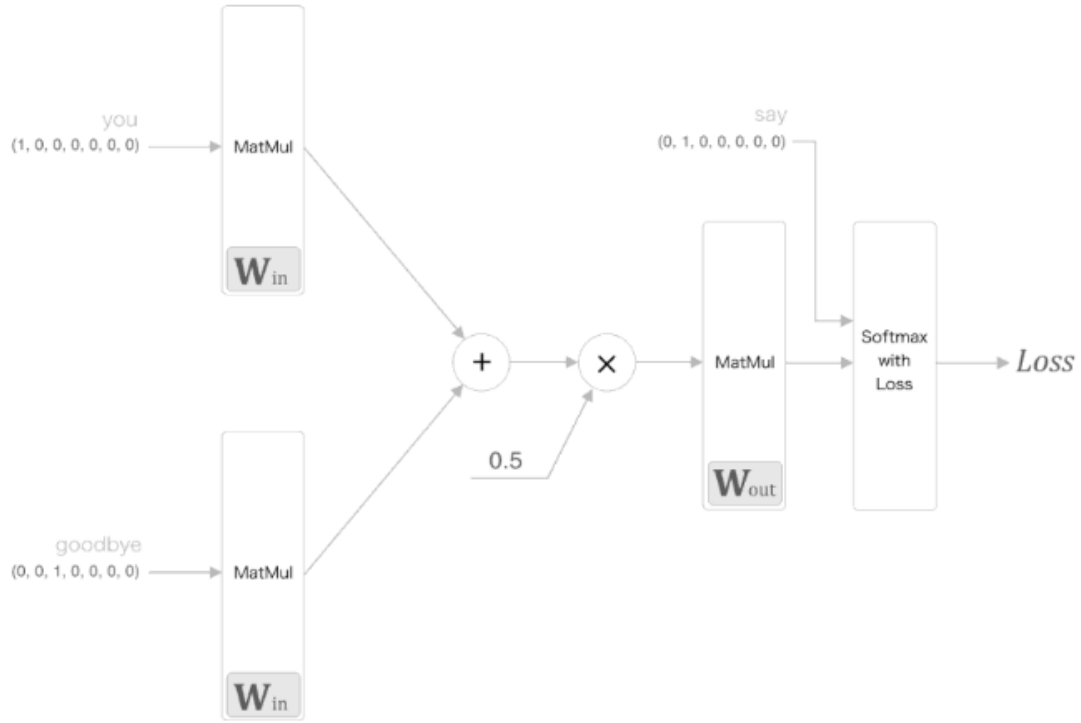



```python
import sys
sys.path.append('..')
import numpy as np
from common.layers import MatMul, SoftmaxWithLoss

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
```

CBOW 모델 구현

- **학습 코드 구현**

그림 3-19 CBOW 모델의 신경망 구성



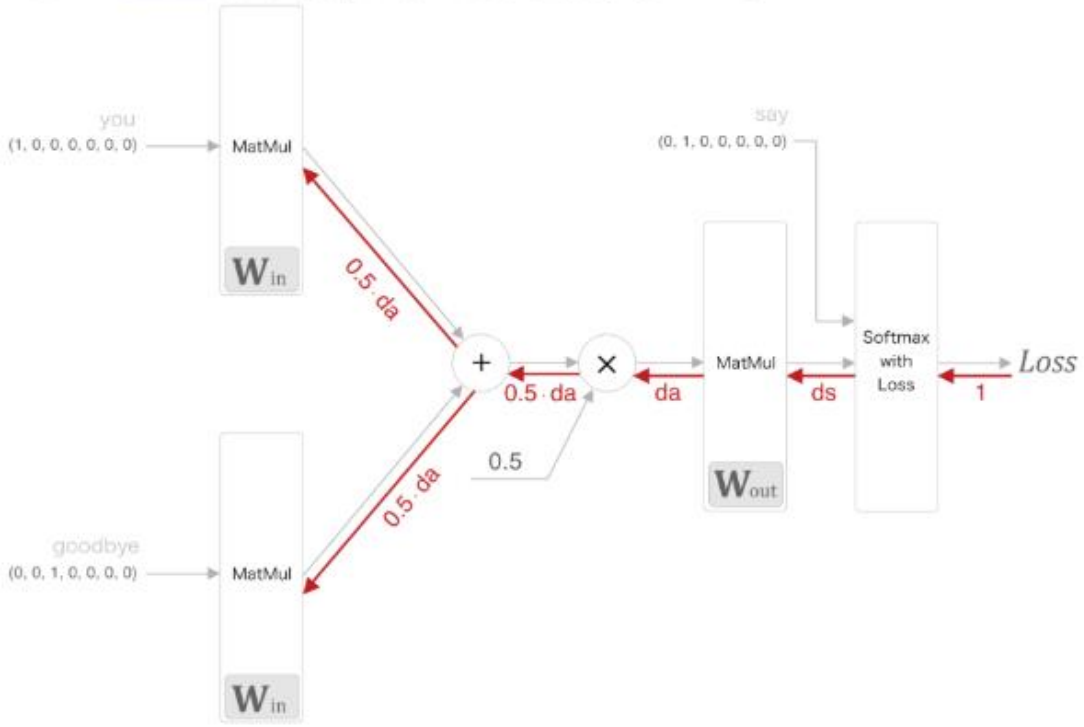
신경망의 순전파 forward() 메서드 구현

```
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
```

CBOW 모델 구현

- **학습 코드 구현**

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



신경망의 역전파 backward() 메서드 구현

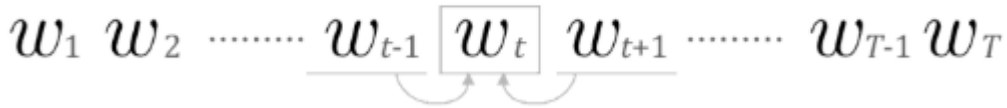
```
def backward(self, dout=1):
    ds = self.loss_layer.backward(dout)
    da = self.out_layer.backward(ds)
    da *= 0.5
    self.in_layer1.backward(da)
    self.in_layer0.backward(da)
    return None
```

Word2vec 보충

▪ CBOW 모델과 확률

A라는 현상이 일어날 확률을 P(A), **동시확률**은 P(A,B)로 쓴다. 동시확률이란 'A와 B가 동시에 일어날 확률'을 말한다.

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



맥락으로 w_{t-1}과 w_{t+1}이 주어졌을 때 타깃이 w_t가 될 확률은 수식으로 다음과 같이 쓸 수 있다.

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

위의식을 토대로 CBOW 모델의 손실 함수는 다음과 같이 쓸 수 있다.

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

위의 식을 말뭉치 전체로 확장하면 다음과 같이 변환하여 쓸 수 있다.

$$L = -\frac{1}{T}\log P(w_t \mid wt_{-1}, wt_{+1})$$

Word2vec 보충

- **skip-gram 모델**

word2vec은 2개의 모델을 제안하고 있다. 하나는 지금까지 본 CBOW 모델이고, 다른 하나는 skip-gram 모델이다.

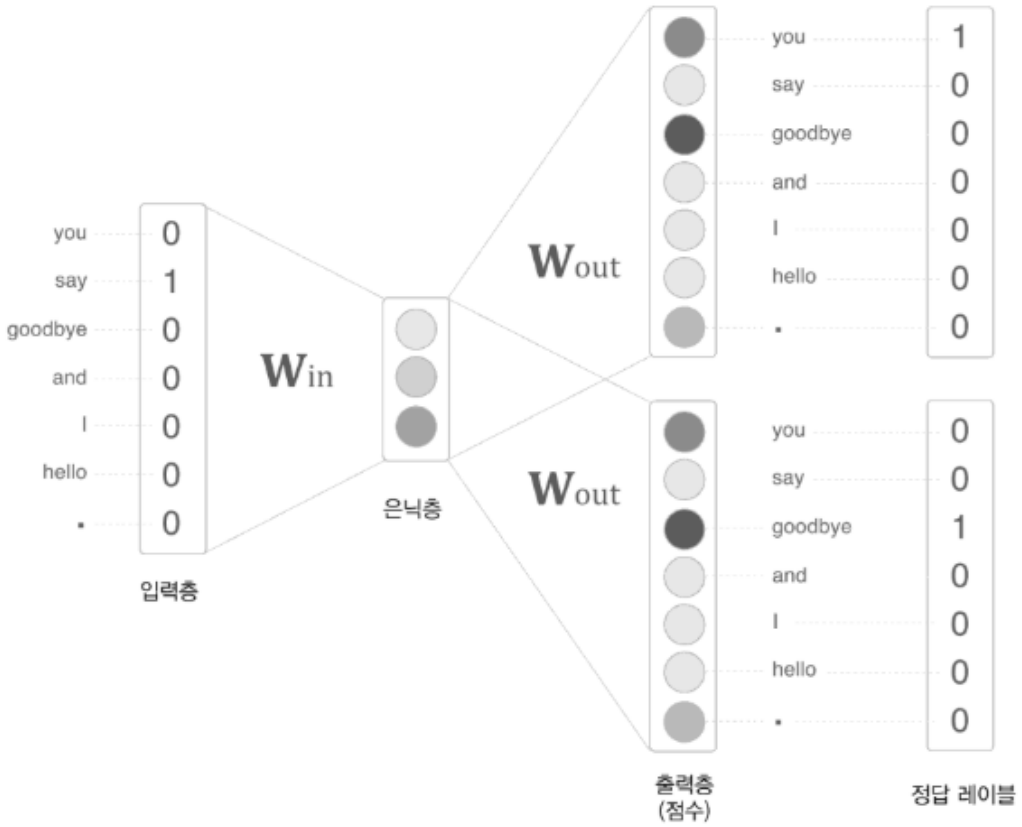
그림 3-23 CBOW 모델과 skip-gram 모델이 다루는 문제

CBOW 모델은 맥락이 여러 개가 있고, 그 여러 맥락으로부터 중앙의 단어 (타깃) 를 추측한다.
반면, skip-gram 모델은 중앙의 단어 (타깃)로부터 주변의 여러 단어(맥락)를 추측한다.

Word2vec 보충

- **skip-gram 모델**

그림 3-24 skip-gram 모델의 신경망 구성 예



[그림 3-24]에서 보듯 skip-gram 모델의 입력층은 하나이다.
한편 출력층은 맥락의 수만큼 존재한다.
따라서 각 출력층에서는 개별적으로 손실을 구하고,
이 개별 손실들을 모두 더한 값을 최종 손실로 한다.

그러면 skip-gram 모델을 확률 표기로 나타내어보자.

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

여기서 skip-gram 모델은 맥락의 단어들 사이에 관련성이 없다고 가정하고 다음과 같이 분해한다.

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

이어서 위 식에 **음의 로그 가능도**를 구하면 다음과 같다.

$$L = -\frac{1}{T}\sum_{t-1}^{T}(logP(w_{t\_1} \mid w_t) + logP(wt_{+1} \mid wt))$$

Word2vec 보충

- **skip-gram 모델**

다음과 같이 CBOW 모델과 skip-gram 모델은 그 차이서 명확하다.
그럼 CBOW 모델과 skip-gram 모델 중 어느 것을 사용해야 할까??

그 대답은 **skip-gram 모델**이라고 할 수 있다.
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.

Word2vec 보충

▪ 통계 기반 vs 추론 기반

지금까지 통계 기반 기법과 추론 기반 기법 (특히 word2vec)을 살펴봤다.
학습하는 틀면에서 두 기법에는 큰 차이가 있었다. 통계 기반 기법은 말뭉치의 전체 통계로부터 1회 학습하여 단어의 분산 표현을 얻었는데, 추론 기반 기법에서는 말뭉치를 일부분씩 여러 번 보면서 학습했다. (미니배치학습)

이 학습 방법 외에 두 기법이 또 어떻게 다른지 비교해보자.

먼저, 어휘에 추가할 새 단어가 생겨서 단어의 분산 표현을 **갱신**해야 하는 상황을 생각해보자.
통계 기반 기법은 분산 표현을 조금만 수정하고 싶어도 계산을 처음부터 다시해야 하는데, 추론 기반 기법 (word2vec)은
지금까지 학습한 가중치를 **초깃값**으로 사용해서 새로 들어온 단어에 대해서만 학습하면 된다.

그렇다면 두 기법으로 얻는 단어의 분산 표현의 성격이나 정밀도 면에서는 어떨까?
분산 표현의 성격에 대해 논하자면, 통계 기반 기법에서는 주로 단어의 유사성이 인코딩된다.
반면, word2vec (특히 skip-gram 모델)에서는 단어의 유사성은 물론, 한층 복잡한 단어 사이의 패턴까지도 파악되어 인코딩된다.
word2vec "king – man + woman = queen"과 같은 유추 문제를 풀 수 있다는 이야기로 유명하다.

이런 이유로 추론 기반 기법이 통계 기반 기법보다 정화가하고 흔히들 오해하곤 하지만, 실제로는 두 방법 사이에서 우열을
가릴 수 없었다고 한다.

NOTE_ "Don`t count, predict! (세지 말고, 추측하라!)"로 시작하는 제목의 논문이 2014년에 발표되었다.
이 논문에 따르면 추론 기반 기법이 통계기반 기법보다 항상 더 정확했다고 하지만, 이후 다른 논문들에서는 단어의 유사성
관련 작업의 경우 정확성은 하이퍼파라미터에 크게 의존하며, 두 방법의 우열을 가릴 수 없다고 보고했다.