밑바닥부터 시작하는 딥러닝 2: 3장 word2vec

2020/02/04

3. word2vec

- 이해를 돕기 위해 쉽고 단순하게 구현된 word2vec의 구조를 살펴보고 직접 구현해보자.
  - 다음 장에서는 3장의 '단순한' word2vec에 몇가지 개선을 더해 '진짜' word2vec을 구현한다.

3.1 추론 기반 기법과 신경망

- 단어를 벡터로 표현하는 방법
  - 통계 기반 기법
  - 추론 기반 기법
- 단어의 의미를 얻는 방식은 서로 다르지만, 모두 분포 가설을 배경으로 함.

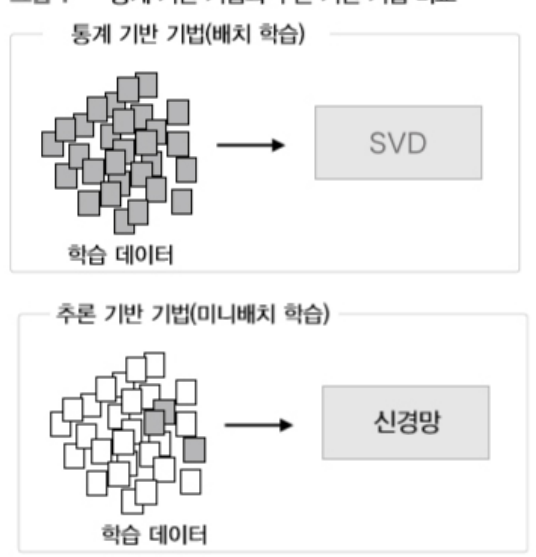
3.1.1 통계 기반 기법의 문제점

- **통계 기반 기법 -> 주변 단어의 빈도를 기초로 단어 표현**
  - 단어의 동시발생 행렬을 만들고, SVD를 적용해 밀집벡터를 얻음
  - 하지만 대규모 코퍼스를 다루게 된다면 문제점 발생

1) 실전에서 다루는 코퍼스의 어휘 수는 엄청나게 크다.
   - 영어: 어휘수 100만개 이상, 이때 동시발생 행렬은 '100만 * 100만' -> SVD를 적용하는 일 자체가 어려움
   - n*n 행렬에 SVD 적용하는 비용: $O(n^3)$

2) 한번에 한꺼번에 단어 분산 표현을 얻게 됨
   - 통계 기반 기법: 학습 데이터 한꺼번에 처리(**배치 학습**)
   - 추론 기반 기법: 학습 데이터의 일부를 사용해 순차적으로 학습(**미니배치 학습**)
     - 신경망이 한번에 하나의 미니배치의 학습 샘플씩 반복해 학습, 가중치를 갱신
     - 대규모의 코퍼스를 다루는 경우에도 신경망 학습이 가능함.
     - GPU를 이용한 병렬 계산도 가능 -> 학습 속도 증가

그림 1 통계 기반 기법과 추론 기반 기법 비교



3.1.2 추론 기반 기법 개요

- 추론: 맥락이 주어졌을 때 “?”안에 무슨 단어가 들어가는지를 추측하는 작업
  - 이런 추론 문제를 반복해서 풀어 단어의 출현 패턴을 학습

- 추론 기반 기법 -> 모델이 등장

- 모델은 맥락 정보를 입력 받아 각 단어의 출현 확률을 출력
- 말뭉치를 사용해 모델이 올바른 추측을 하도록 학습
- 그 학습의 결과로 단어의 분산 표현을 얻는 것

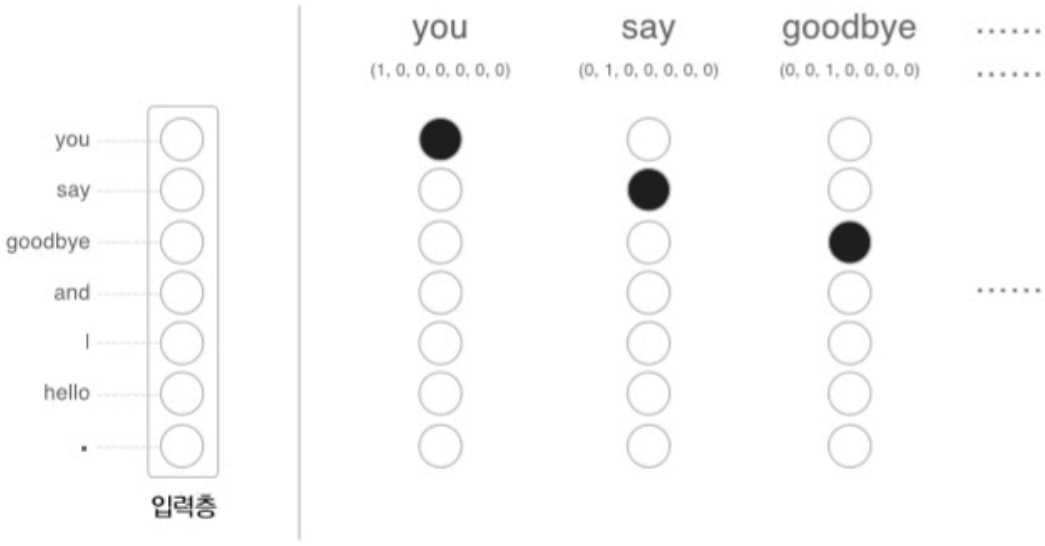
3.1.3 신경망에서의 단어 처리

- 신경망을 이용해 단어를 처리하기 위해서는 단어를 고정 길이의 벡터로 변환 => **원핫(one-hot) 표현**
- **원핫 표현:** 벤터의 원소 중 하나만 1이고 나머지는 모두 0인 벡터
  - 총 어휘 수만큼의 원소를 갖는 벡터를 만들고 인덱스가 단어 ID와 같은 원소를 1로 나머지는 0으로 설정
  - 입력 레이어의 뉴런 갯수가 어휘 수가 되며, 각 뉴런은 각 단어에 대응하게 된다.

"you say goodbye and i say hello."

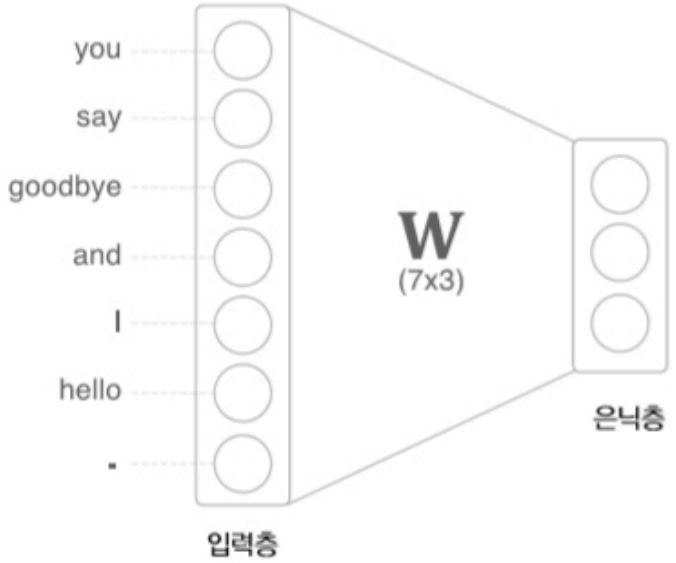
{0: 'you', 1: 'say', 2: 'goodbye', 3: 'and', 4: 'i', 5: 'hello', 6: '.'}

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



3.1.3 신경망에서의 단어 처리

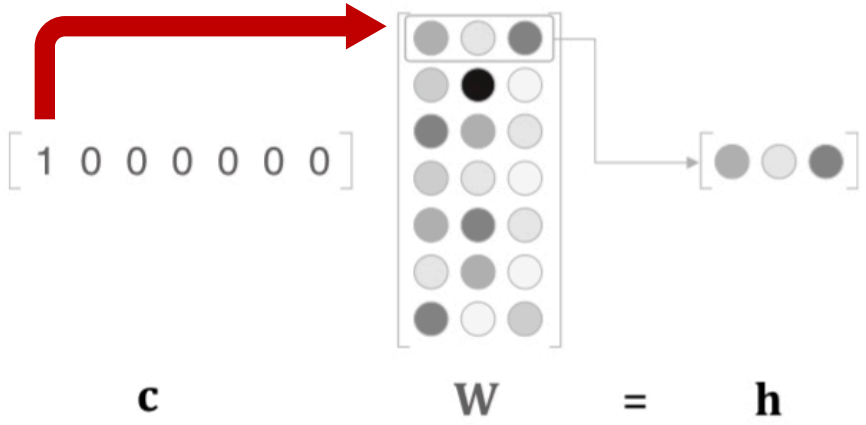
- 원핫 표현으로 된 단어 하나를 완전연결계층에 의해 변환하는 과정:
  - 완전 연결 계층이므로 모든 노드가 화살표로 연결
  - 화살표에는 가중치(매개변수)가 존재, 입력층 뉴런과의 가중 합(weighted sum)이 은닉층 뉴런이 됨.(bias 생략)



- 하지만 c가 원핫 벡터 => 사실 해당 위치의 행벡터가 추출된 것에 불과
  - matmul 연산 비효율적
  - 4.1절에서 개선

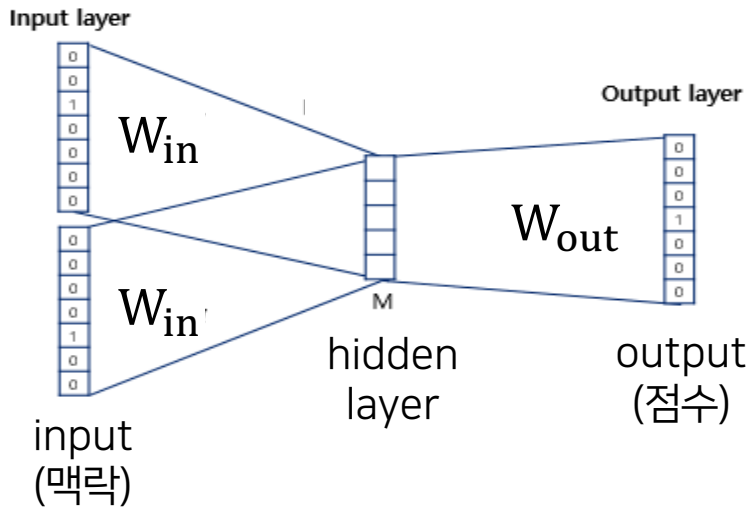
```python
# 3.1.3 신경망에서의 단어 처리
C = np.array([[1, 0, 0, 0, 0, 0, 0]]) # 임의의 원핫벡터
W = np.random.randn(7, 3)
h = np.matmul(C, W)
print(h)
```

```
[[0.01570505 1.07700054 0.42052299]]
```

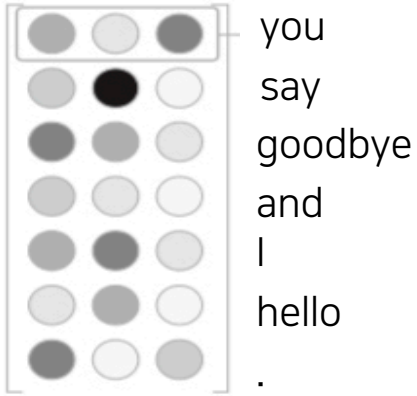


3.2 단순한 word2vec:
1. CBOW 모델의 추론 처리

- 모델을 신경망으로 구축해보자.
  - word2vec에서 제안하는 CBOW(continuous bag-of-words) 모델

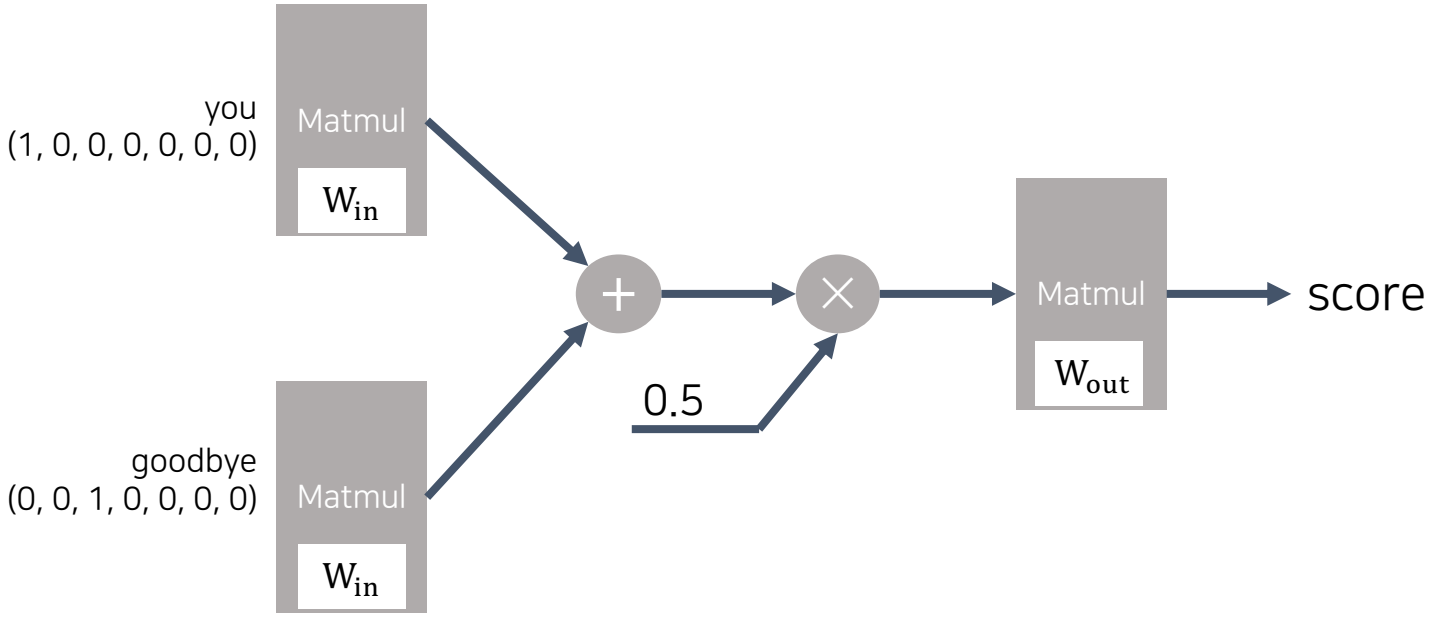


- **CBOW 모델:** 맥락으로부터 타깃(target)을 추측하는 용도의 신경망
  - input: 맥락('you', 'goodbye' 같은 단어들의 목록, 원핫 벡터)
  - output:각 뉴런이 각각의 단어에 대응해 각 단어의 '점수'를 뜻함
    - 값이 높을수록 대응 단어의 출현 확률이 높다.
    - 점수에 softmax 함수를 적용해 확률을 얻게 된다.
  - W_in: 단어 분산 표현
    - 각 행에는 해당 단어의 분산 표현이 담김
    - 학습을 진행할수록 맥락에서 출현하는 단어를 잘 추측하는 방향으로 갱신



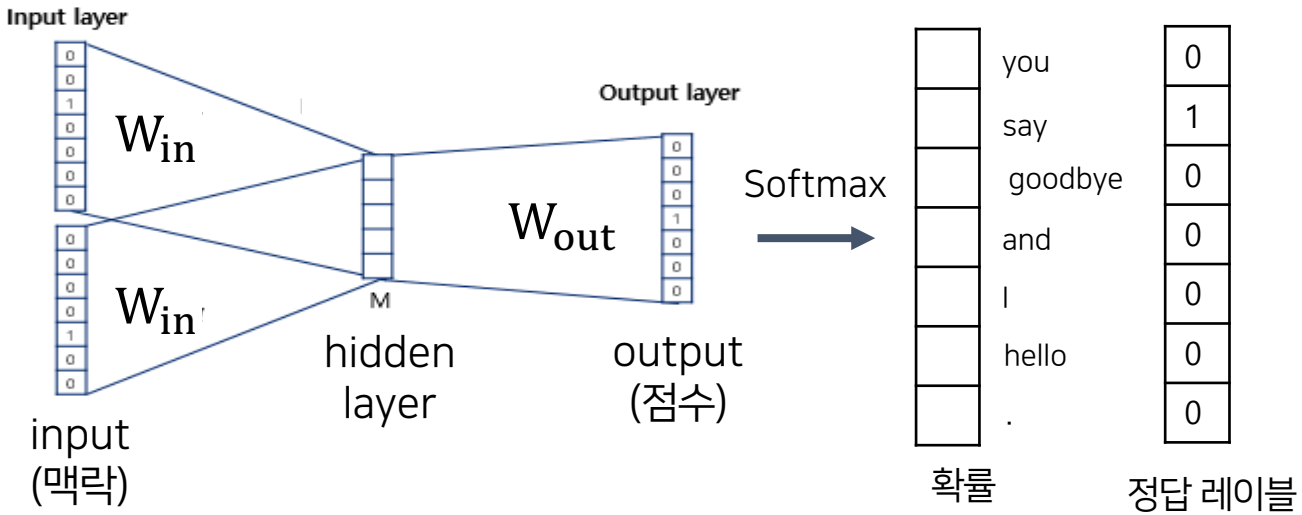
* 히든 레이어의 뉴런 수는 입력 레이어의 뉴런 수보다 적어야 함.
- 히든 레이어에 단어 예측에 필요한 정보를 간결하게 담을 수 있음
  => 밀집벡터 표현을 얻을 수 있음

3.2.1 CBOW 모델의 추론 처리



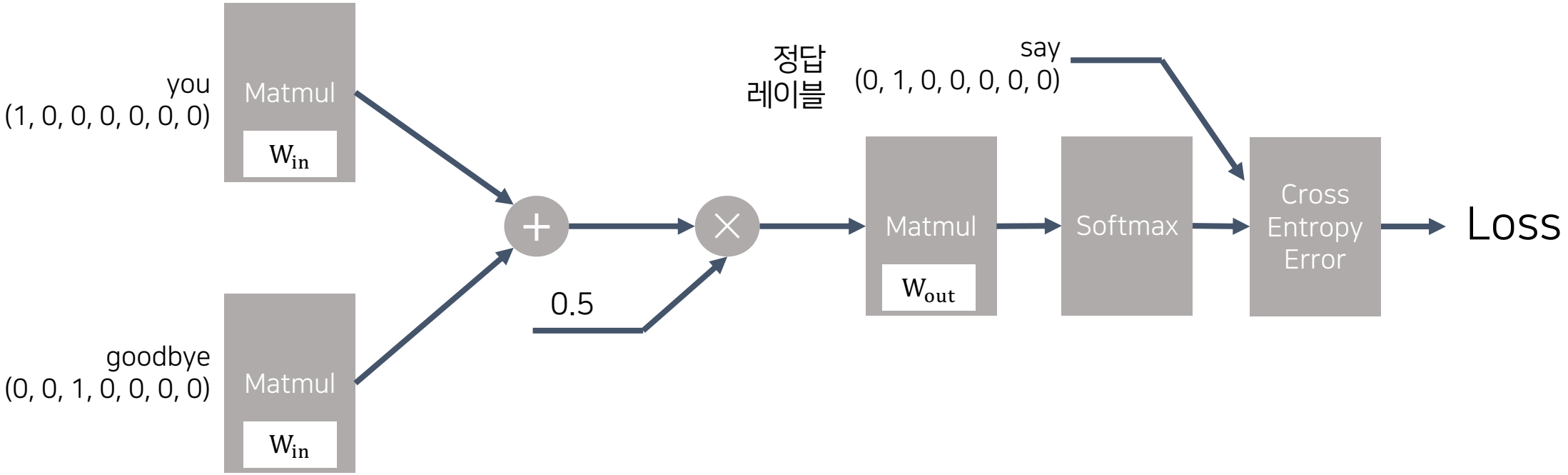

```python
# 3.2.1 CBOW 모델의 추론 처리
# 샘플 input data
c0 = np.array([[1, 0, 0, 0, 0, 0, 0]])
c1 = np.array([[0, 0, 1, 0, 0, 0, 0]])

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

# layer 생성
in_layer0 = MatMul(W_in) # 입력층 측의 계층은
in_layer1 = MatMul(W_in) 가중치 W_in 공유
out_layer = MatMul(W_out)

# 순전파(forward propagation)
h0 = in_layer0.forward(c0) # 첫번째 히든레이어
h1 = in_layer0.forward(c1) # 두번째 히든레이어
h = 0.5 * (h0 + h1) # 평균 구하기
s = out_layer.forward(h)

print(s)
```

3.2.2 CBOW 모델의 학습



- 출력층에서는 각 단어의 점수를 출력 => 이 점수에 **소프트맥스 함수**를 적용해 '확률'을 얻는다!
  - 맥락(전후 단어)이 주어졌을 때, 그 중앙에 어떤 단어가 출현하는지를 나타내는 확률
- 올바른 예측을 할 수 있도록 학습을 통해 가중치를 조절
  - 단어의 출현 패턴을 파악한 벡터가 학습됨.
  - 이렇게 학습된 벡터를 실험해 본 결과 단어의 의미 면에서나 문법 면에서 사람의 직관과 잘 부합함을 확인

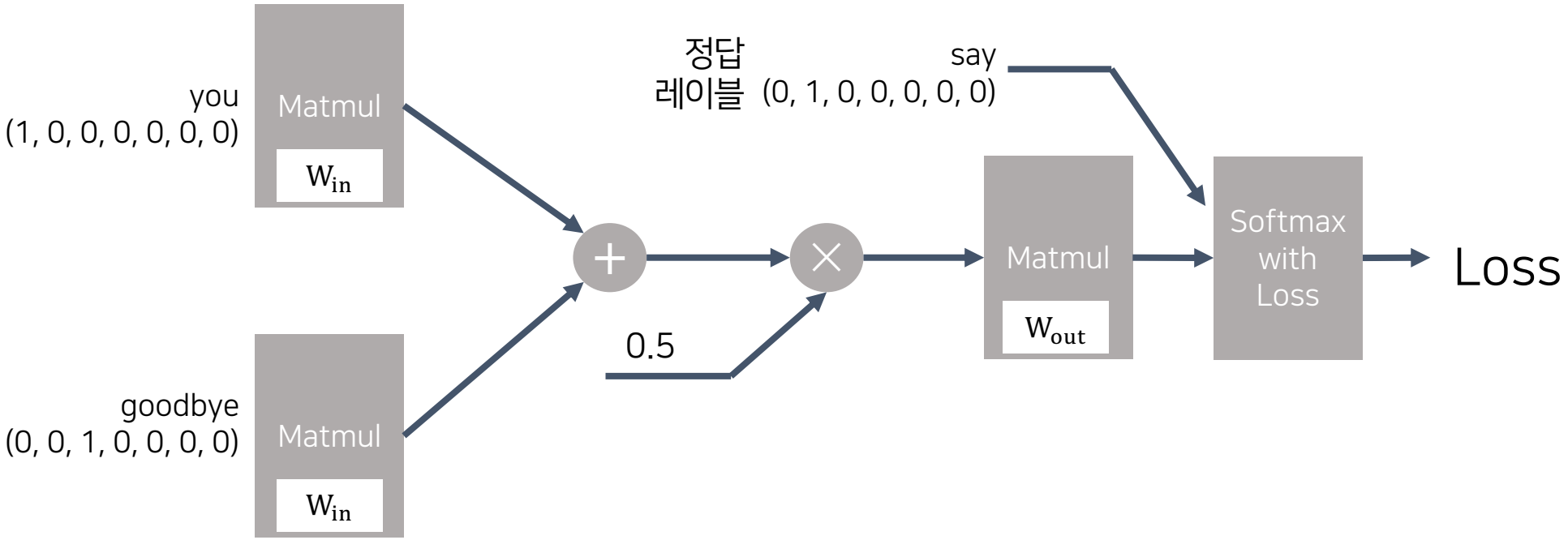
3.2.2 CBOW 모델의 학습



- 소프트맥스 함수를 이용해 점수를 확률로 변환
- 그 확률과 정답 레이블로부터 교차 엔트로피 오차를 구함
- 그 값을 손실로 사용해 학습을 진행

이를 반복해 모델의 손실을 구해가며 학습해 나간다.

3.2.2 CBOW 모델의 학습

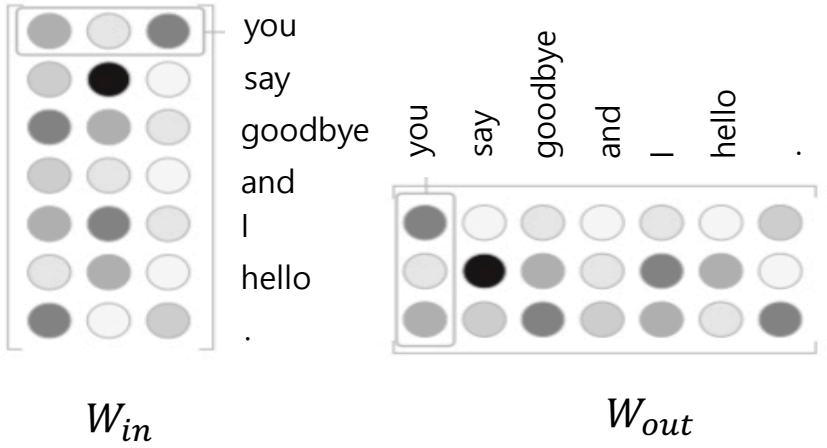


- 소프트맥스 함수를 이용해 점수를 확률로 변환
- 그 확률과 정답 레이블로부터 교차 엔트로피 오차를 구함
- 그 값을 손실로 사용해 학습을 진행

} 이를 반복해 모델의 손실을 구해가며 학습해 나간다.

3.2.3 word2vec의 가중치와 분산 표현

- 두 가지 가중치가 존재:
  - W_{in}: 입력층 FC 레이어의 가중치, 각 행이 각 단어의 분산 표현에 해당
  - W_{out}: 출력층 FC 레이어의 가중치, 각 열이 각 단어의 분산 표현에 해당



- 이 중 어느 것을 단어의 분산 표현으로 선택해야 할까?
  - 입력 측의 가중치 /출력 측의 가중치 /양쪽 가중치
  - word2vec(특히 skip-gram 모델) -> 보통 **입력 측의 가중치**만 이용!

3.3 학습 데이터 준비: 1. 맥락과 타깃

- word2vec의 입력: 맥락 / 정답 레이블: 타깃
  - 신경망에 '맥락'을 입력했을 때 '타깃'이 출현할 확률을 높여야 한다.
- 어떻게 '맥락'과 '타깃'을 만들 수 있을까?
  - 목표로 하는 단어 -> 타깃
    - 오직 한개만 존재
  - 그 주변 단어 -> 맥락
    - 갯수 상관 없음.

양 끝 단어를 제외한 모든 단어에 대해 수행

- 맥락의 각 행이 input으로, 타깃의 각 행이 정답 레이블로 사용된다.

3.3.1. 맥락과 타깃

- 전처리: preprocess 함수 사용

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

```
[0 1 2 3 4 1 5 6]
{0: 'you', 1: 'say', 2: 'goodbye', 3: 'and', 4: 'i', 5: 'hello', 6: '.'}
```

코퍼스 텍스트를 단어 ID로 변환

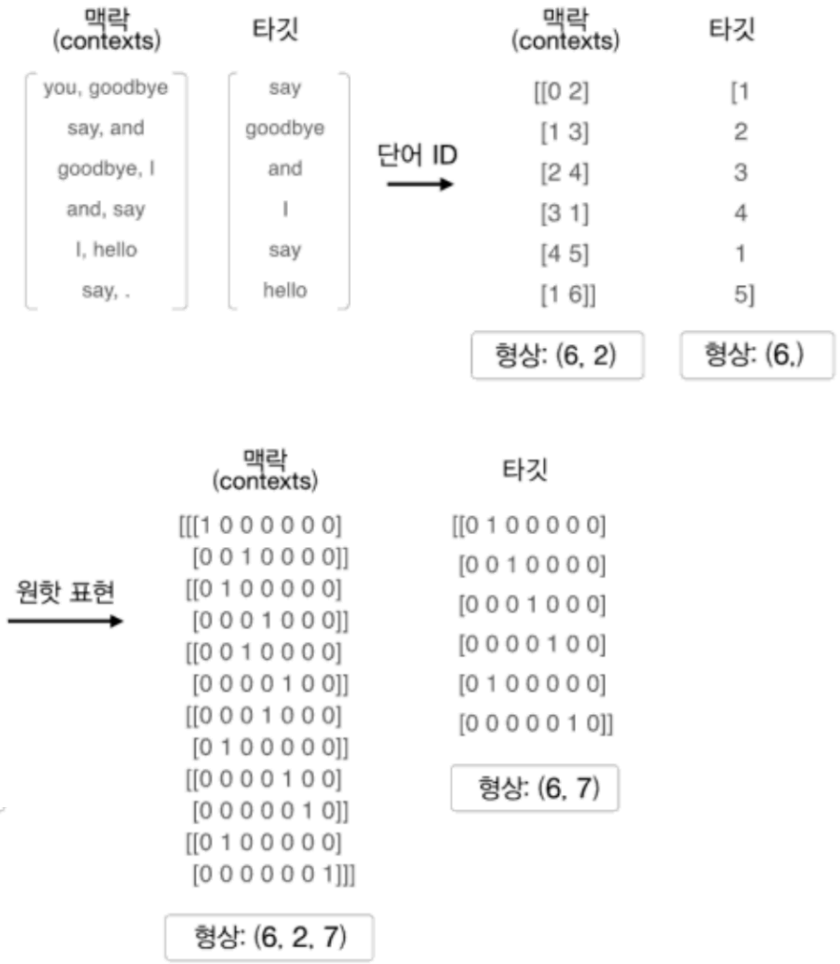
- 단어 ID 배열인 corpus로부터 맥락과 타깃을 만들기
  - corpus를 주면 맥락과 타깃을 변환하는 함수

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

    # you 'say goodbye and i say hello' . 이므로 say ~ hello까지 고려
    for idx in range(window_size, len(corpus) - window_size):
        cs = []
        for t in range(-window_size, window_size + 1):
            if t == 0: # 자기 자신인 경우 pass
                continue
            cs.append(corpus[idx + t])
        contexts.append(cs)

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

=> 원소가 단어의 ID ∴ 원핫 표현으로 변환해야 한다!

3.3.2 원핫 표현으로 변환

- 단어 ID를 원핫 표현으로 변환

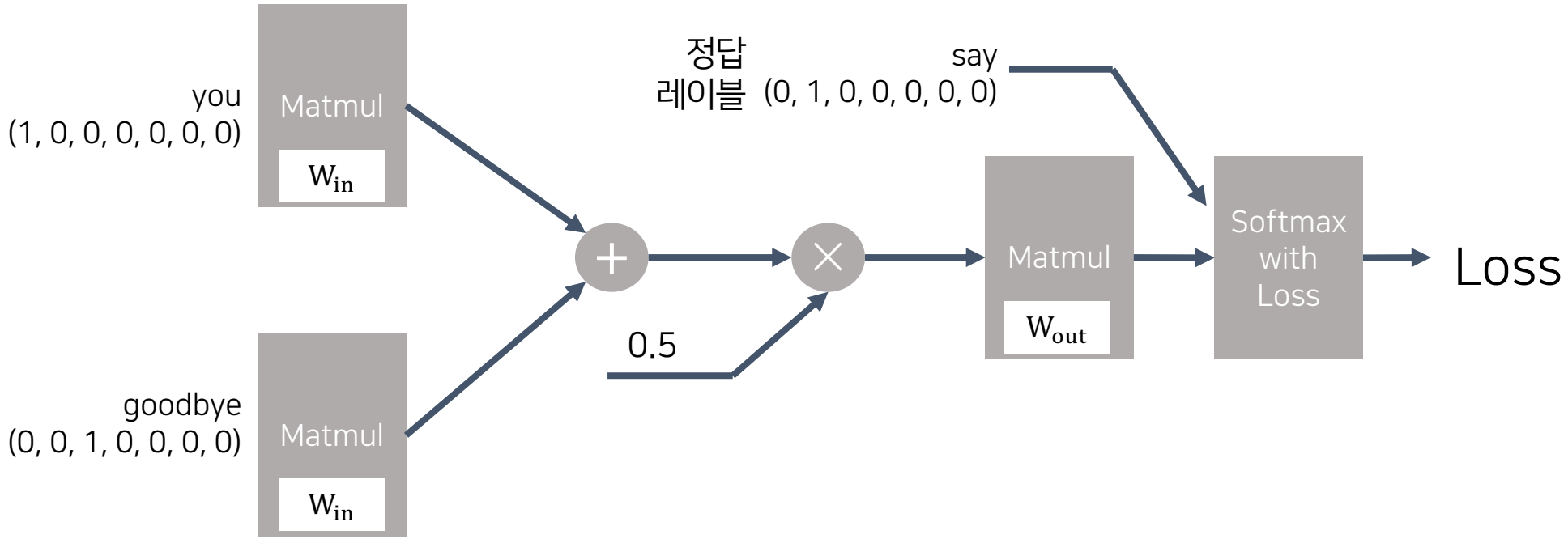



```
def convert_one_hot(corpus, vocab_size):
    N = corpus.shape[0]

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

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

    return one_hot
```

'단어 ID 목록'과 '어휘 수'를 인수로 받는다.

3.4 CBOW 모델 구현

- 이제 이 신경망을 구현해보자. => SimpleCBOW 클래스
  - 다음 장에서는 이를 개선한 CBOW 클래스를 구현



3.4 CBOW 모델 구현

- SimpleCBOW 클래스:

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

        # weight 초기화
        W_in = 0.01 * np.random.randn(V, H).astype('f') #float32로 초기화
        W_out = 0.01 * np.random.randn(H, V).astype('f')

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

        # weight와 gradient를 리스트에 저장
        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

        self.word_vecs = W_in
```

- vocab_size: 어휘 수
- hidden_size: 은닉층 뉴런 수

맥락에서 사용하는 단어의 수(윈도우 크기)만큼 생성

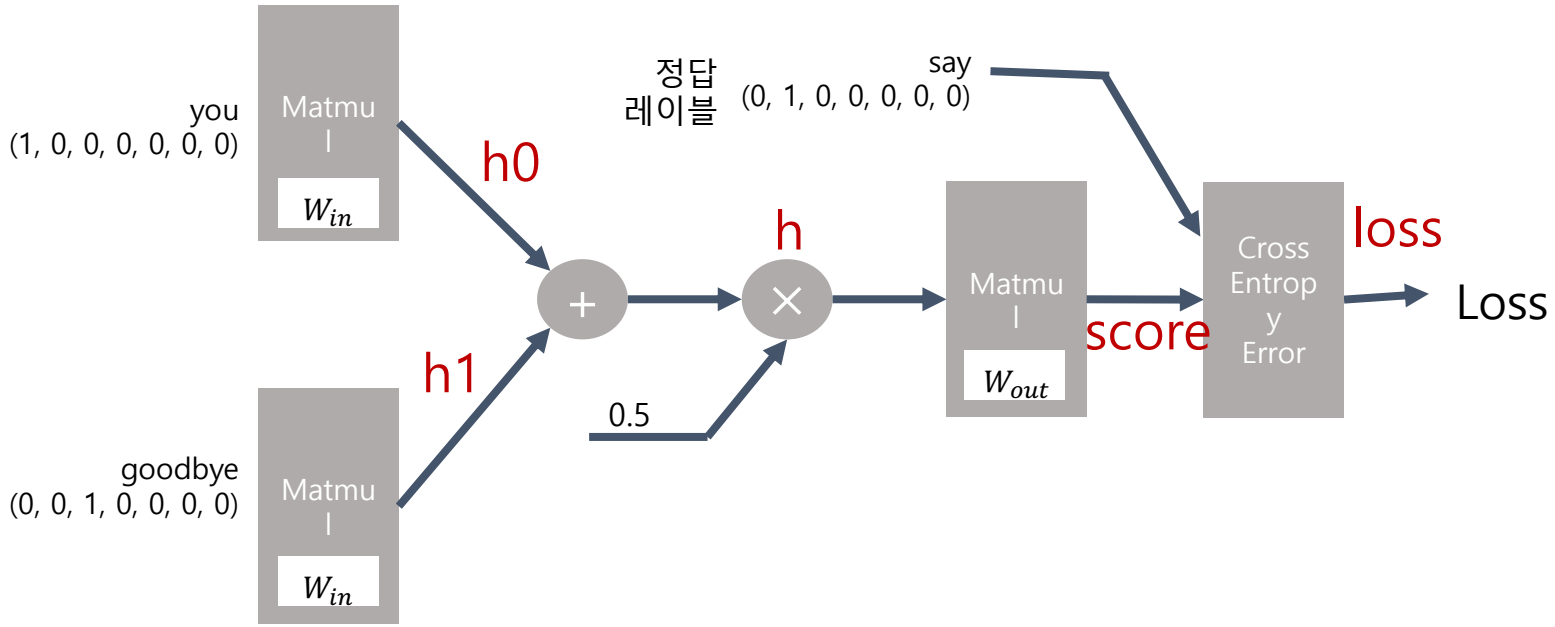
3.4 CBOW 모델 구현

- SimpleCBOW 클래스:

context(맥락)과 target(타깃)을 받아 loss(손실)을 반환

```
def forward(self, contexts, target):
    h0 = self.in_layer0.forward(contexts[:, 0]) # target 왼쪽의 맥락
    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
```

- context: 3차원 numpy 배열이라 가정, (6, 2, 7)
  - 0번째 차원: 미니배치 수
  - 1번째 차원: 맥락의 윈도우 크기
  - 2번째 차원: 원핫 벡터
- target: 2차원 numpy 배열, (6, 7)

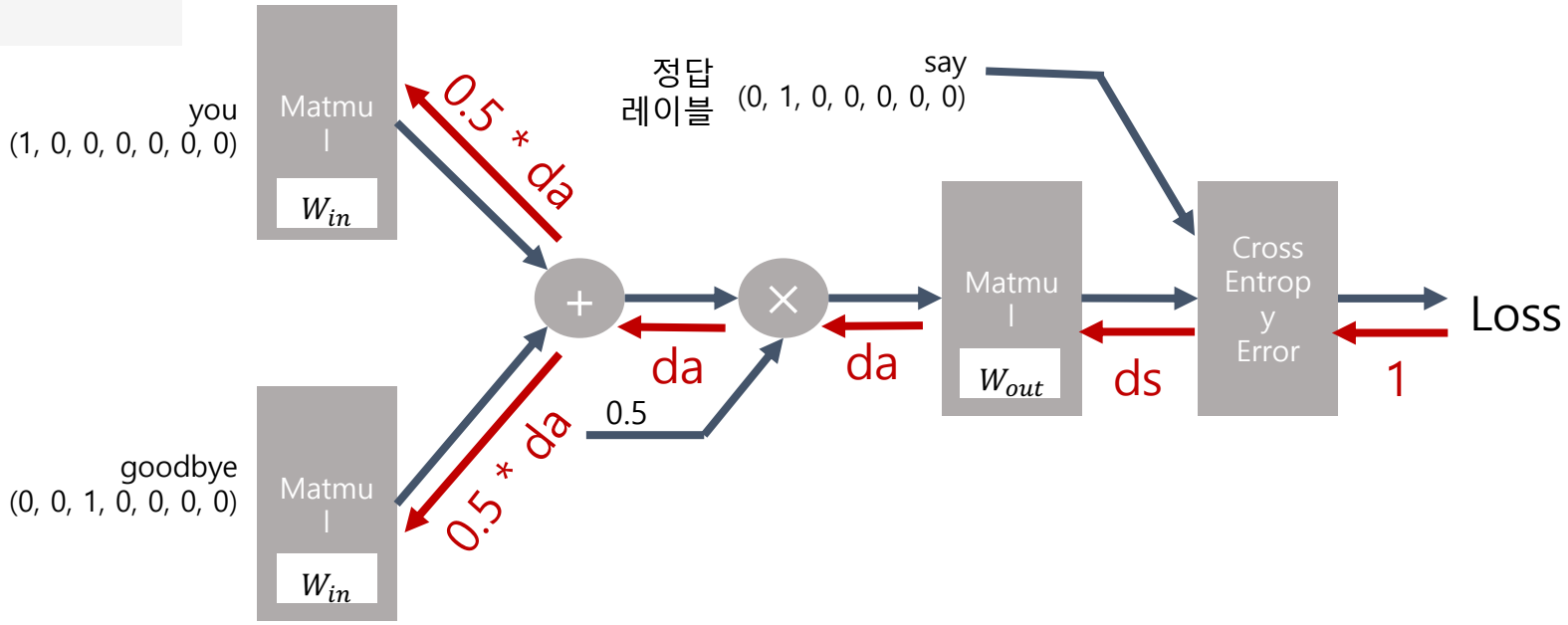


3.4 CBOW 모델 구현

- SimpleCBOW 클래스:

기울기를 순전파 때와는 반대 방향으로 전파

```
def backward(self, dout=1):
    ds = self.loss_layer.backward(dout)
    da = self.out_layer.backward(ds)
    da *= 0.5
    self.in_layer0.backward(da)
    self.in_layer1.backward(da)

    return None
```



3.4.1 학습 코드 구현

```
# 하이퍼파라미터 설정
max_epoch = 300
batch_size = 30
hidden_size = 10
learning_rate = 1.0

# 데이터 읽기, 모델과 옵티마이저 생성
x, t = spiral.load_data()
model = TwoLayerNet(input_size=2, hidden_size=hidden_size, output_size=3)
optimizer = SGD(lr=learning_rate)

# 학습에 사용하는 변수
data_size = len(x)
max_iters = data_size // batch_size
total_loss = 0
loss_count = 0
loss_list = []

for epoch in range(max_epoch):
    # 데이터 뒤섞기
    idx = np.random.permutation(data_size)
    x = x[idx]
    t = t[idx]

    for iters in range(max_iters):
```

```
batch_x = x[iters*batch_size:(iters+1)*batch_size]
batch_t = t[iters*batch_size:(iters+1)*batch_size]

# 기울기를 구해 매개변수 갱신
loss = model.forward(batch_x, batch_t)
model.backward()
optimizer.update(model.params, model.grads)

total_loss += loss
loss_count += 1

# 정기적으로 학습 경과 출력
if (iters+1) % 10 == 0:
    avg_loss = total_loss / loss_count
    print('| 에폭 %d |  반복 %d / %d | 손실 %.2f'
          % (epoch + 1, iters + 1, max_iters, avg_loss))
    loss_list.append(avg_loss)
    total_loss, loss_count = 0, 0
```

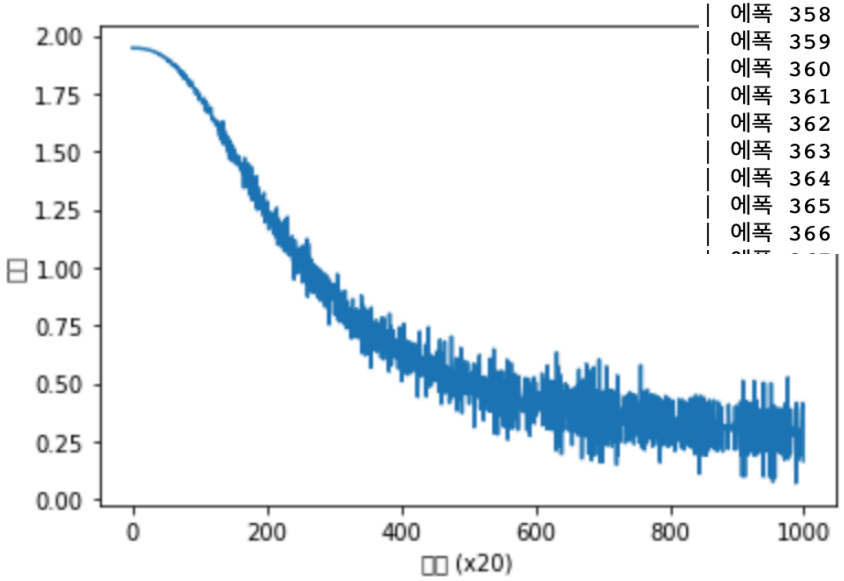
3.4.1 학습 코드 구현

- optimizer는 Adam 사용

```
# 3.4.1 학습 코드 구현
window_size = 1
hidden_size = 5
batch_size = 3
max_epoch = 1000

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])
```

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

```
| 에폭 354 |  반복 1 / 2 | 시간 0[s] | 손실 0.75
| 에폭 355 |  반복 1 / 2 | 시간 0[s] | 손실 0.78
| 에폭 356 |  반복 1 / 2 | 시간 0[s] | 손실 0.57
| 에폭 357 |  반복 1 / 2 | 시간 0[s] | 손실 0.83
| 에폭 358 |  반복 1 / 2 | 시간 0[s] | 손실 0.69
| 에폭 359 |  반복 1 / 2 | 시간 0[s] | 손실 0.75
| 에폭 360 |  반복 1 / 2 | 시간 0[s] | 손실 0.75
| 에폭 361 |  반복 1 / 2 | 시간 0[s] | 손실 0.71
| 에폭 362 |  반복 1 / 2 | 시간 0[s] | 손실 0.64
| 에폭 363 |  반복 1 / 2 | 시간 0[s] | 손실 0.67
| 에폭 364 |  반복 1 / 2 | 시간 0[s] | 손실 0.77
| 에폭 365 |  반복 1 / 2 | 시간 0[s] | 손실 0.64
| 에폭 366 |  반복 1 / 2 | 시간 0[s] | 손실 0.69
```



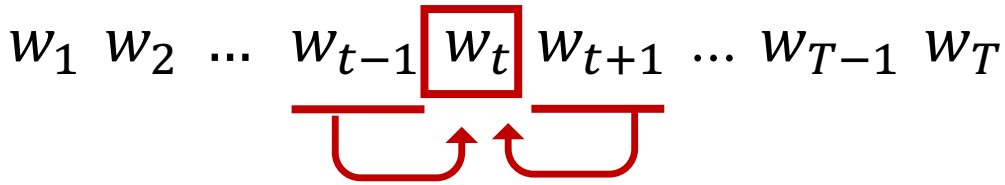

```
you [-0.93914205  1.0908543  -1.7300427   1.0188375   0.9571275 ]
say [ 1.3096251   0.16776861  0.14008527 -0.36733222 -1.2955229 ]
goodbye [-0.9733452   0.90887356  0.8532208   0.98771966  0.9652035 ]
and [ 1.0796083  1.3751658 -1.3504019 -1.3303574 -1.07497  ]
i [-0.9756904   0.8871254   0.83426523  1.0070565   0.951747  ]
hello [-0.95174205  1.1084721  -1.7025151   1.0059282   0.964749  ]
. [ 1.120871  -1.2440021  1.2101704  1.4294586 -1.107089 ]
```

각 단어 별 분산 표현

- 하지만 말뭉치의 크기가 너무 작고, 지금 구현한 CBOW 모델은 처리 효율 면에서 문제 발생
  - 4장에서 이를 개선하여 CBOW 모델을 구현

3.5 word2vec 보충: 1. CBOW 모델과 확률

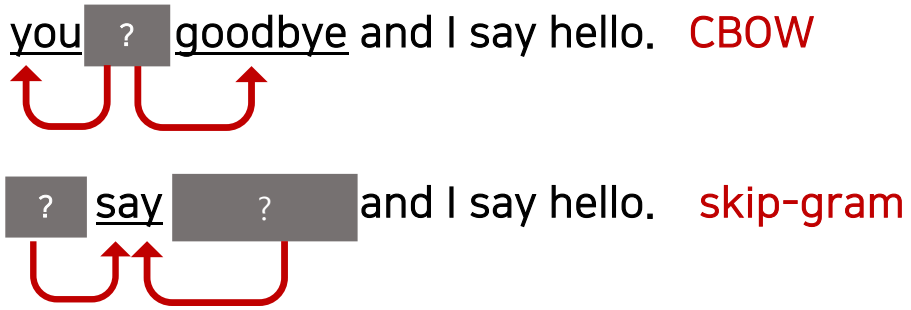
- P(A): A가 일어날 확률
- P(A, B): 동시 확률, A와 B가 동시에 일어날 확률
- P(A|B): 사후 확률, B일 때 A가 일어날 확률



- 맥락이 w_{t-1}, w_{t+1} 일 때, 타깃이 w_t가 될 확률: $P(w_t|\, w_{t-1}, w_{t+1})$
  - 즉 w_{t-1}, w_{t+1}가 주어졌을 때 w_t가 일어날 확률 => CBOW는 이 식을 모델링하고 있는 것!
  - 이를 이용하면 CBOW 모델의 loss함수도 간결하게 표현할 수 있음.
  - cross entropy error = $-\sum_k t_k \log y_k$ (y_k= k번째에 해당하는 사건이 일어날 확률, t_k = 정답 레이블)
    - 여기서 정답은 'w_t가 발생' 즉 $(0, 0, \dots, 0, 1, 0, \dots, 0, 0)$이므로 w_t에 해당하는 원소만 1이고 나머지는 0
    - 따라서 L = $-\log P(w_t|\, w_{t-1}, w_{t+1})$ ⇒ 음의 로그 가능도(negative log likelihood)
    - 말뭉치 전체에 대한 loss 함수 ⇒ $-\frac{1}{T}\sum_{t=1}^{T} \log P(w_t|\, w_{t-1}, w_{t+1})$

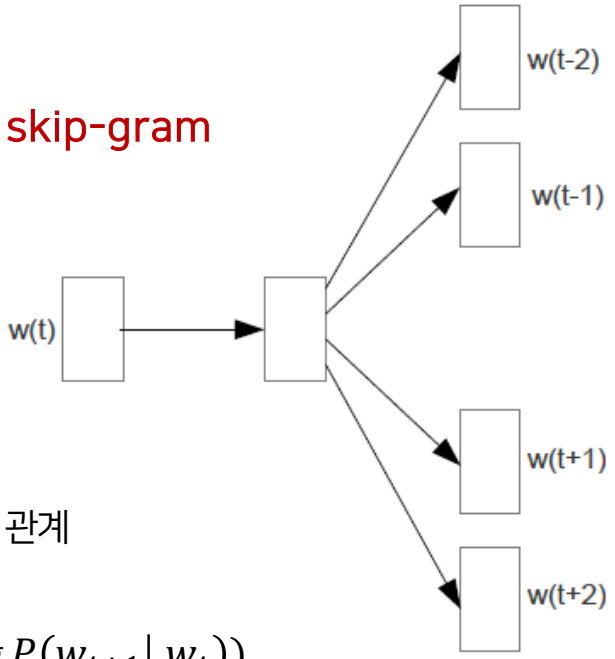
3.5.2 skip-gram 모델

- word2vec은 2개의 모델을 제안한다.
  - CBOW: 여러 맥락 가운데 타깃을 추측
  - skip-gram: 타깃으로부터 맥락을 추측



- skip-gram:
  - 입력 레이어: 1개
  - 출력 레이어: 맥락 수만큼 존재
    - 각 맥락마다 loss를 구하고, 개별 loss를 모두 더한 값을 최종 loss로 함
  - skip-gram 모델의 확률 표기: $P(w_{t-1}, w_{t+1} | w_t)$
    - $P(w_{t-1}, w_{t+1} | w_t) = P(w_{t-1} | w_t)\, P(w_{t+1} | w_t)$
      - "w_t가 주어졌을 때 w_{t-1}, w_{t+1}가 동시에 일어날 확률", 이때 w_{t-1}와 w_{t+1}는 조건부 독립 관계
    - $L = -\log P(w_{t-1}, w_{t+1} | w_t) = -\log P(w_{t-1} | w_t) P(w_{t+1} | w_t)$

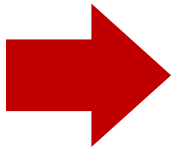
      $= -(\log P(w_{t-1} | w_t) + \log P(w_{t+1} | w_t)) = -\frac{1}{T}\sum_{t=1}^{T}(\log P(w_{t-1} | w_t) + \log P(w_{t+1} | w_t))$



- 그럼 둘 중 어떤 모델을 사용해야 할까?
  - 단어 분산 표현의 정밀도 면에서 skip-gram의 결과가 좋은 경우가 많다.
    - 특히 코퍼스가 커질수록 저빈도 단어나 유추 문제의 성능 면에서 skip-gram이 어 뛰어난 경향이 있음
  - 반면 학습 속도 면에서는 CBOW 모델이 더 빠름
    - skip-gram 모델은 loss를 맥락의 수만큼 구해야 해 계산 비용이 그만큼 커짐

3.5.3 통계 기반 vs 추론 기반

- 학습하는 틀 면에서의 차이:
  - 통계 기반 기법: 말뭉치 전체 통계로부터 1회 학습해 단어의 분산 표현을 얻음
  - 추론 기반 기법(word2vec): 말뭉치를 미니배치만큼 여러 번 보면서 학습
- 어휘에 단어 추가시 단어 분산 표현 갱신하는 경우:
  - 통계 기반 기법: 처음부터 다시 계산
  - 추론 기반 기법: 매개변수를 다시 학습해 효율적으로 갱신
- 단어 분산 표현의 성격이나 정밀도:
  - 통계 기반 기법: 주로 단어의 유사성이 인코딩 됨
  - 추론 기반 기법: 특히 skip-gram 모델의 경우, 단어의 유사성은 물론 한층 복잡한 단어 사이의 패턴까지 파악해 인코딩
    - ex) king – man + woman = queen



- 하지만 실제로 단어의 유사성을 정량 평가해본 결과, 두 기법 간 차이가 별로 없었음.
- 또한 두기법은 서로 관련되어 있음
  - skip-gram과 네거티브 샘플링(다음 장)을 이용한 모델은 모두 말뭉치 전체의 동시발생 행렬에 특수한 행렬 분해를 적용한 것과 같음. 즉, 특정 조건 하에서 두 기법은 서로 연결되어 있음
- word2vec 이후 추론 기반 기법과 통계 기반 기법을 융합한 **GloVe기법**이 등장
  - 말뭉치 전체의 통계 정보를 loss 함수에 도입해 미니배치 학습

3.6 정리

- word2vec의 CBOW 모델에 대한 설명과 구현
  - CBOW: 기본적으로 2층 구성의 단순한 신경망
  - Matmul 레이어와 SoftmaxWithLoss 레이어를 사용해 CBOW 모델 구축
  - 작은 말뭉치로 학습
  - 하지만 이번 장에서 구현한 CBOW 모델 => 처리 효율 면에서 문제가 발생한다.
    - 다음 장에서는 CBOW 모델을 개선