

밑바닥부터 시작하는 딥러닝2

CHAPTER 3&4. word2vec

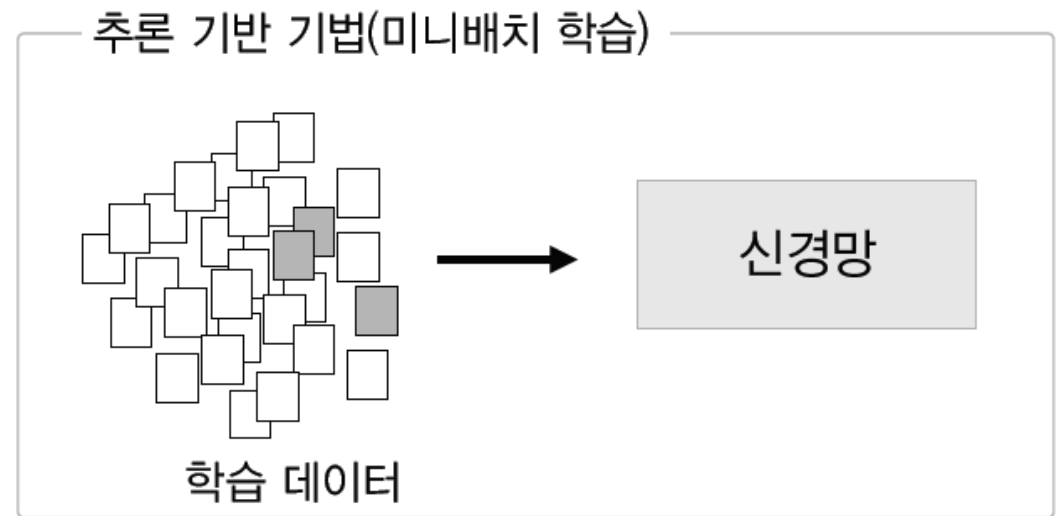
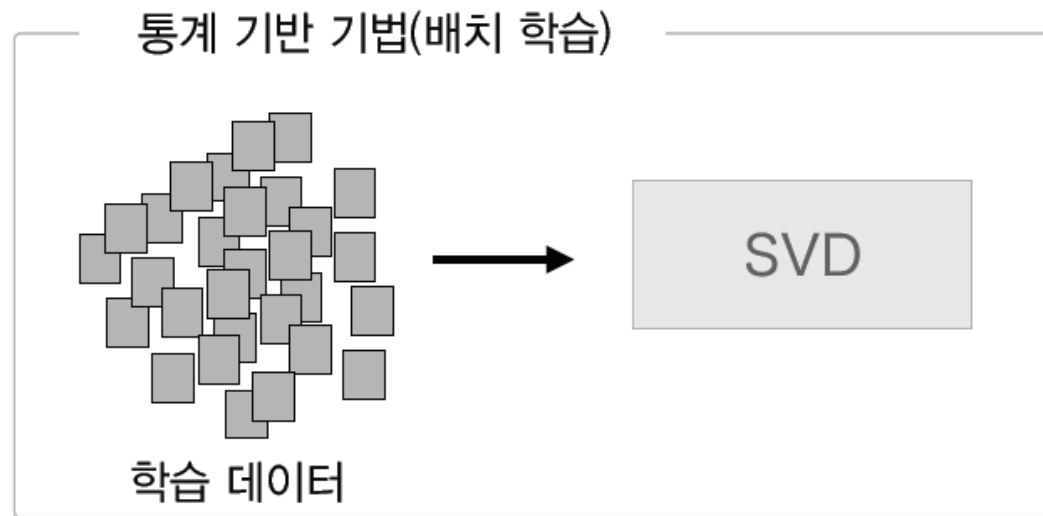
20180376 안제준

3-1 추론 기반 기법과 신경망

1. 통계 기반 기법의 문제점

대규모 말뭉치를 다룰 때 시간이 너무 오래걸립니다. (SVD행렬 적용하는 COST가 $O(n^3)$)


그림 3-1 통계 기반 기법과 추론 기반 기법 비교



3-1 추론 기반 기법과 신경망

- 2. 추론 기반 기법 개요

You ? goodbye and I say hello.

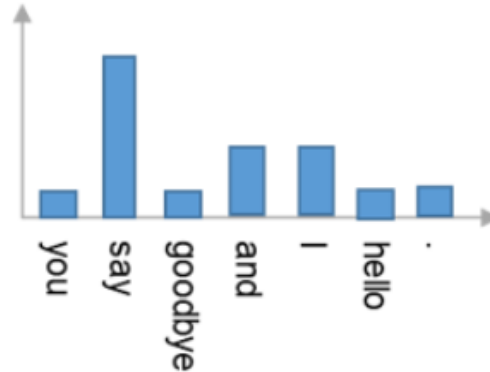


The diagram illustrates a sequence-to-sequence model. On the left, a box contains the input sequence 'you goodbye'. An arrow points from this box to a blue box labeled '모델' (Model). Another arrow points from the model box to a bar chart. The bar chart is titled '확률분포' (Probability Distribution) and shows the predicted output sequence 'say goodbye and I say hello.' The bars represent the probability of each word in the output sequence, with 'say' having the highest probability.

you
goodbye

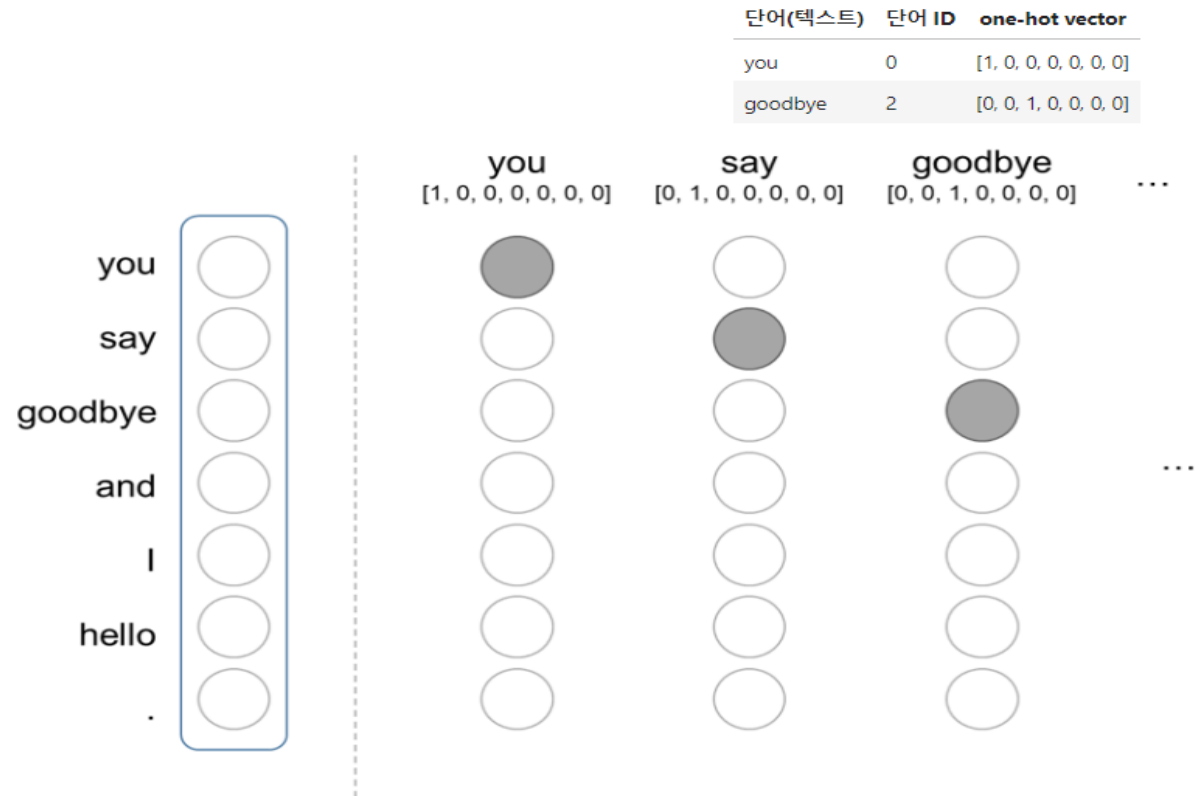
모델

확률분포

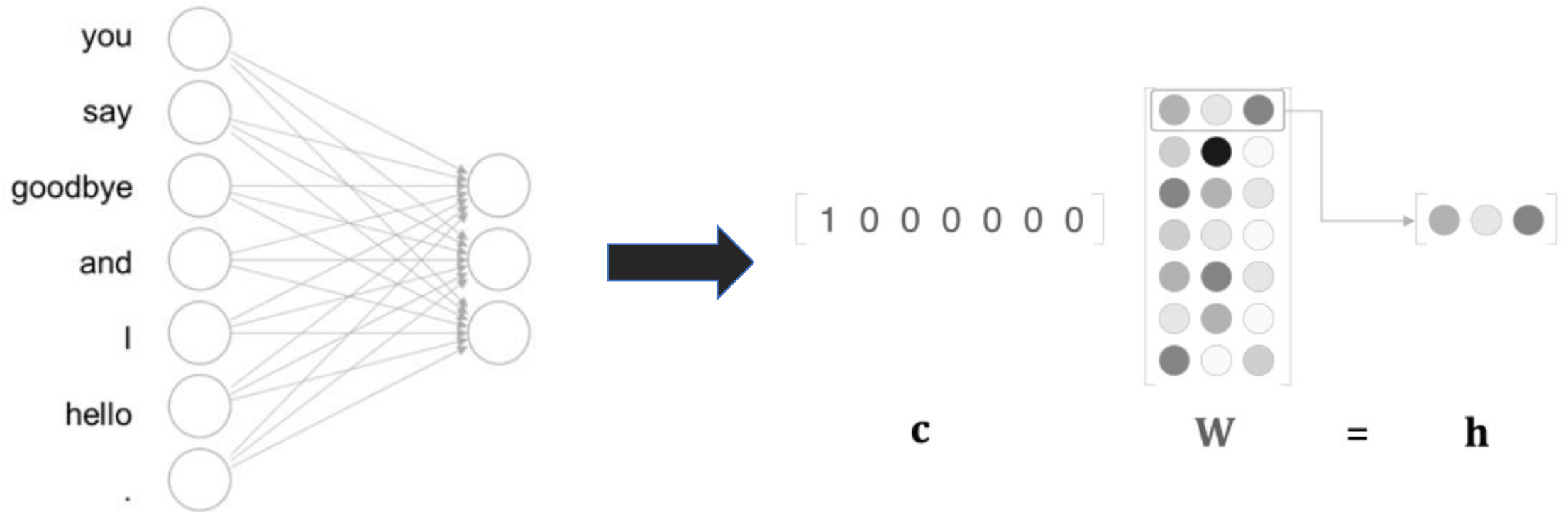


3-1 추론 기반 기법과 신경망

- 3. 신경망에서의 단어 처리

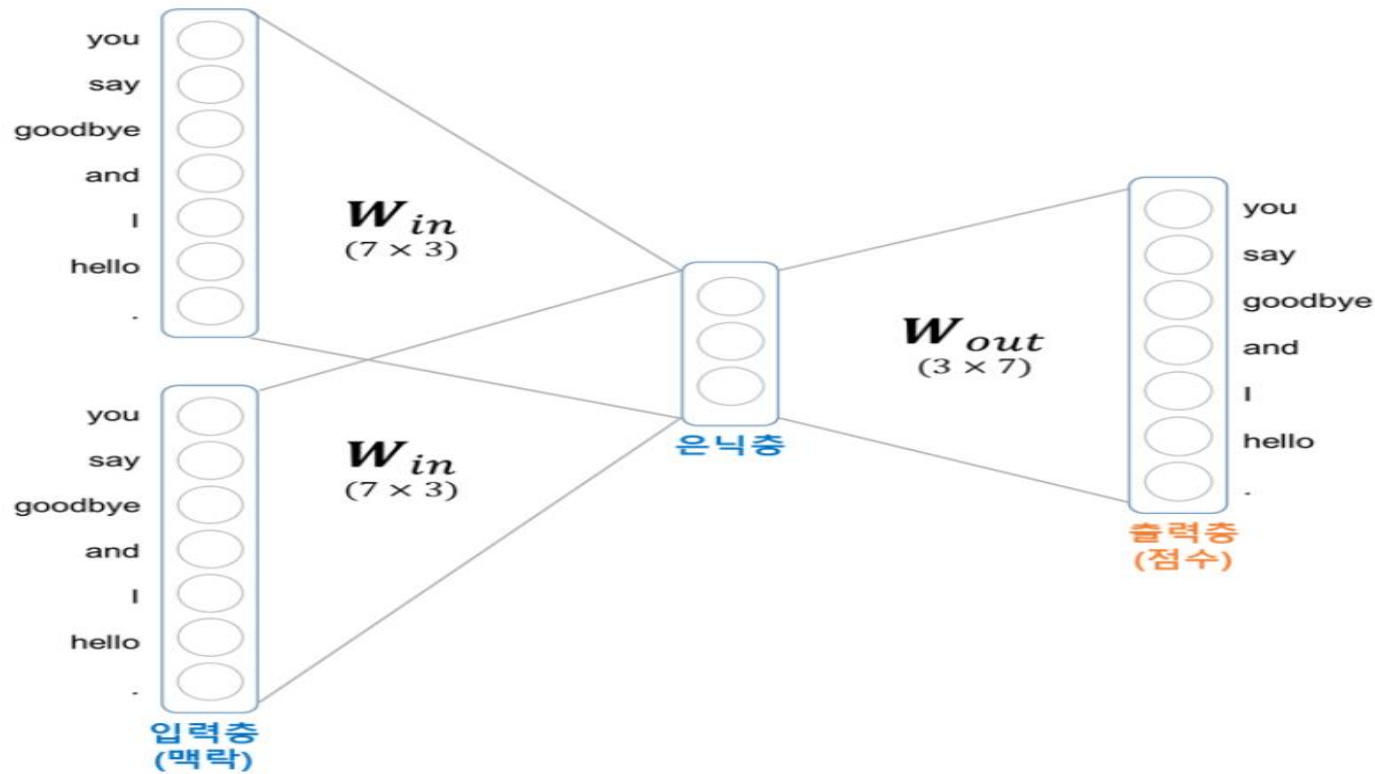


3-1 추론 기반 기법과 신경망



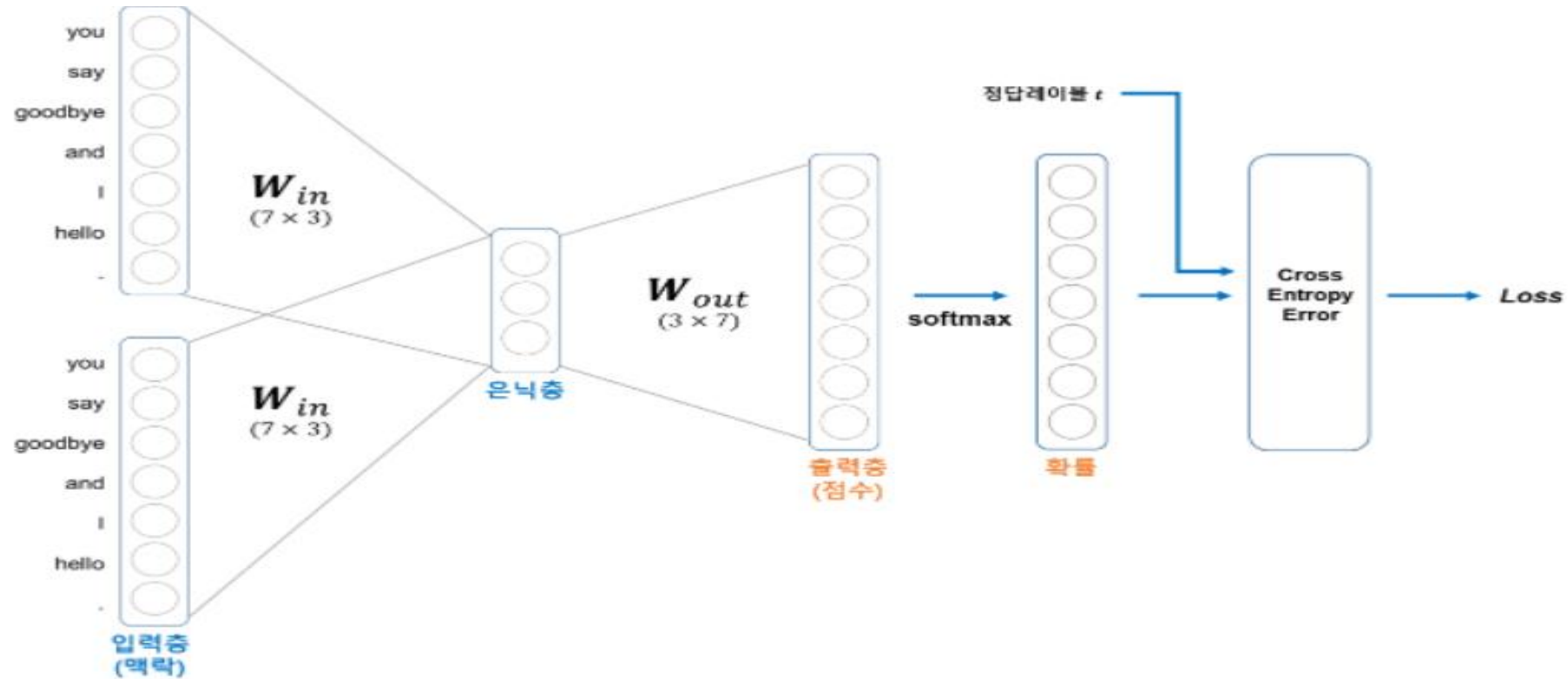
3-2 단순한 word2vec

- 1. CBOW 모델의 추론 처리



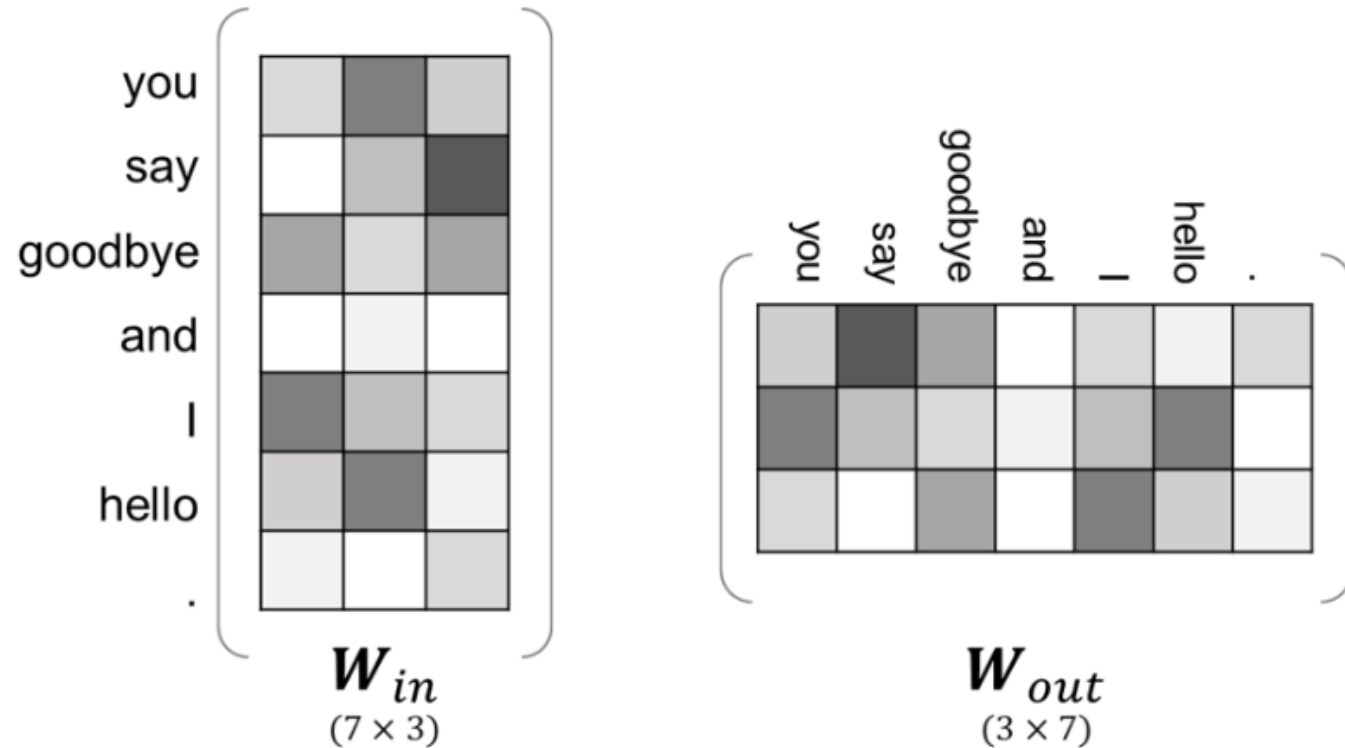
3-2 단순한 word2vec

- 2. CBOW 모델의 학습



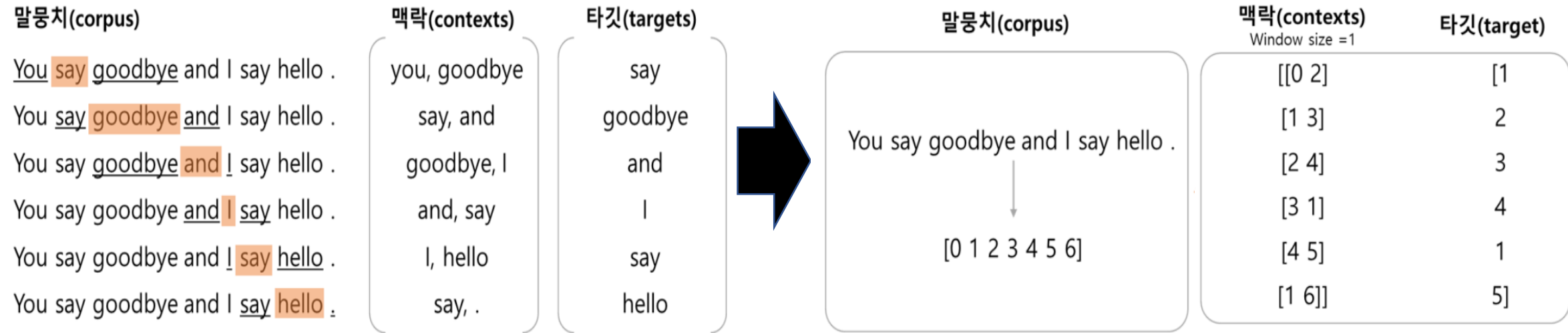
3-2 단순한 word2vec

- 3. word2vec의 가중치와 분산 표현



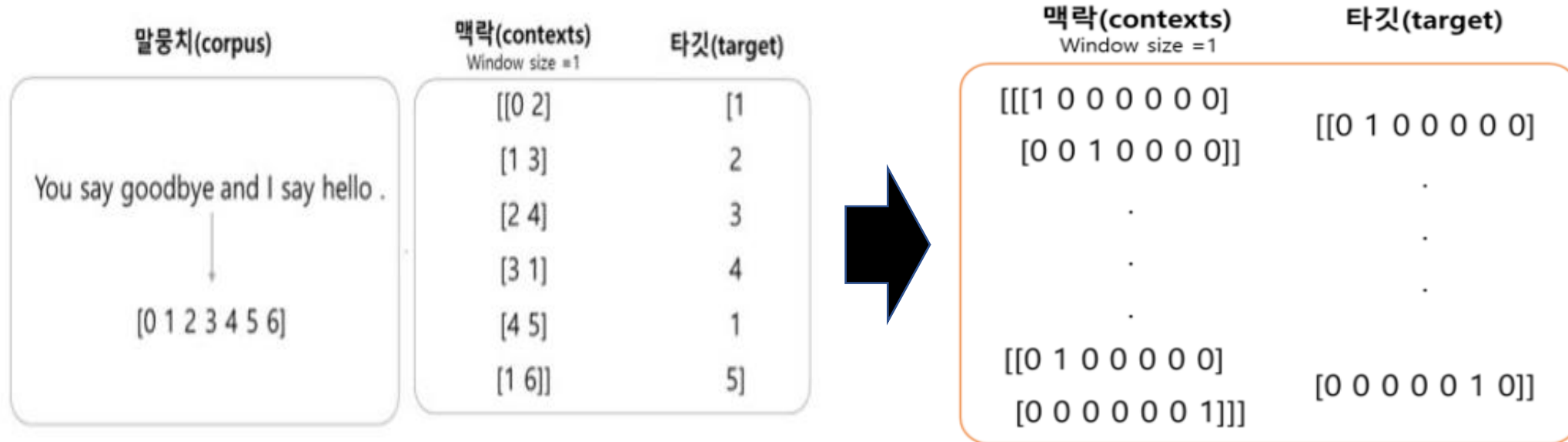
3-3 학습 데이터 준비

• 1. 맥락과 타깃



3-3 학습 데이터 준비

- 2. 원핫 표현으로 변환



3-3 학습 데이터 준비

```
# one-hot encoding function
def convert_one_hot(corpus, vocab_size):
    '''원핫 표현으로 변환

    :param corpus: 단어 ID 목록(1차원 또는 2차원 넘파이 배열)
    :param vocab_size: 어휘 수
    :return: 원핫 표현(2차원 또는 3차원 넘파이 배열)
    ...

    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
```

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

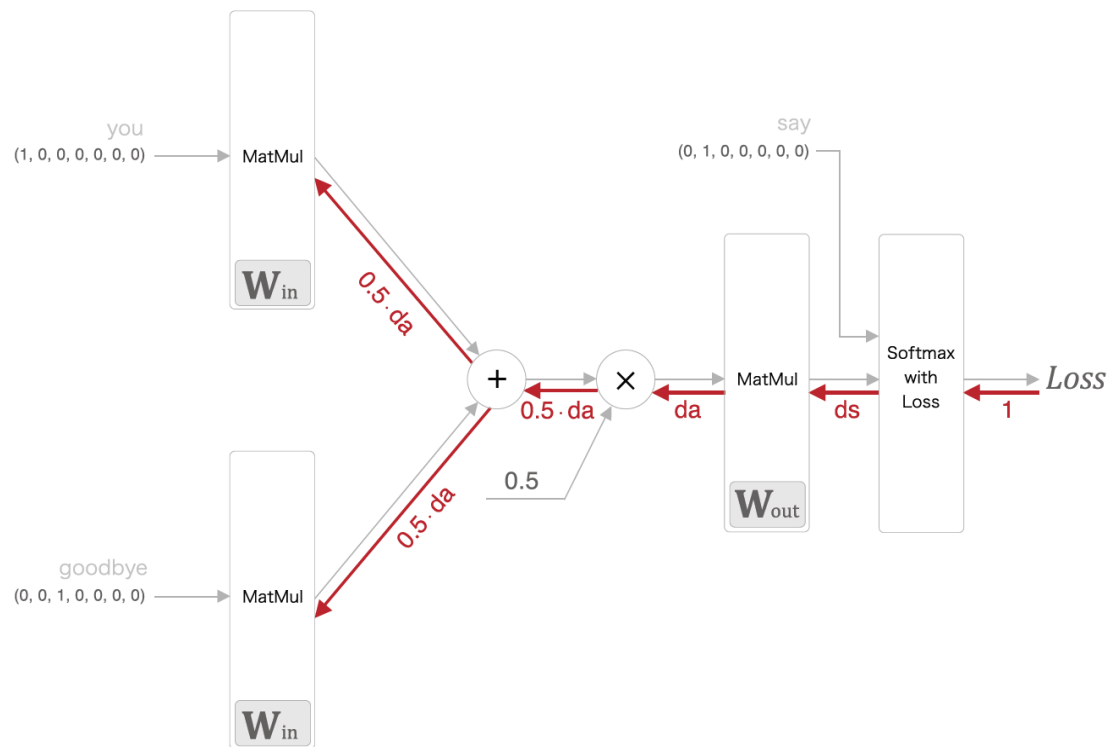
text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)

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)
```

3-4 CBOW 모델 구현

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



```
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')

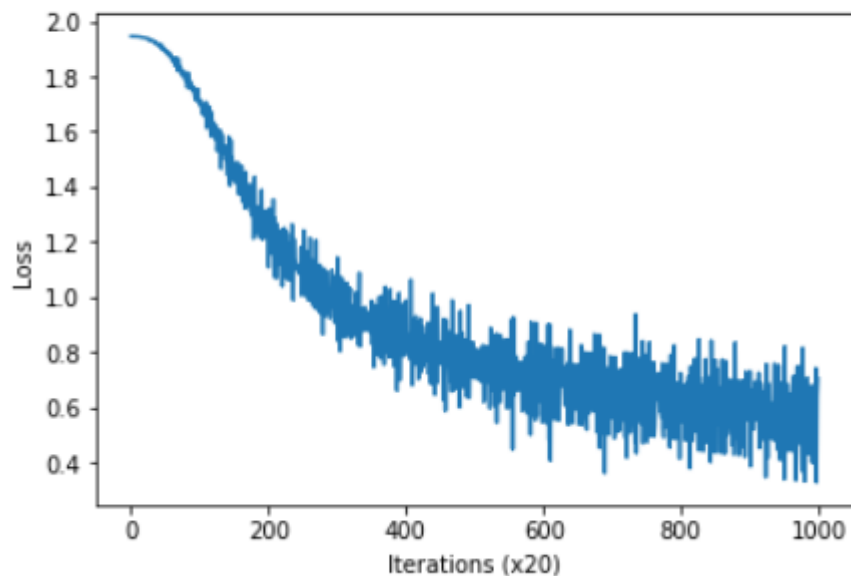
        # 레이어 생성
        self.in_layer0 = MatMul(W_in)
        self.in_layer1 = MatMul(W_in)
        self.out_layer = MatMul(W_out)
        self.loss_layer = SoftmaxWithLoss()

        # 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_vecs1 = W_in
        self.word_vecs2 = W_out.T
```

3-4 CBOW 모델 구현

```
trainer.plot()
```



```
class Adam:
    ...

    Adam(Adaptive Moment Estimation, http://arxiv.org/abs/1412.6980v8)
    m <- m + (1 - beta1)*(dL/dW - m)
    v <- v + (1 - beta2)*[(dL/dW)*(dL/dW) - v]
    W <- W - lr * [sqrt(1-beta2^iter)/(1-beta1^iter)] * (m / sqrt(v + eps))
    ...

    def __init__(self, lr=0.001, beta1=0.9, beta2=0.999):
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2
        self.iter = 0
        self.m = None
        self.v = None

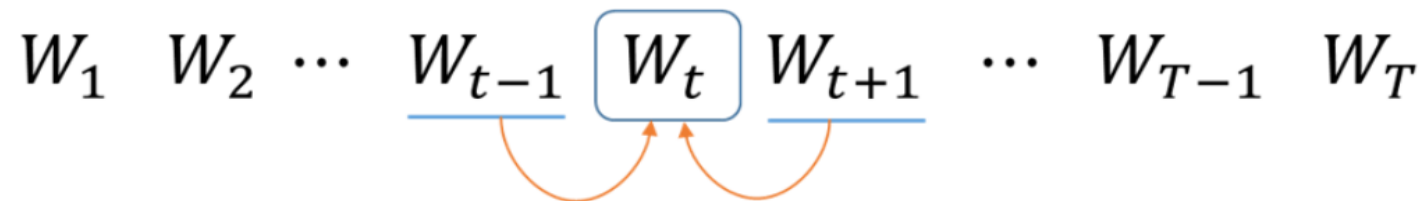
    def update(self, params, grads):
        if self.m is None:
            self.m, self.v = [], []
            for param in params:
                self.m.append(np.zeros_like(param))
                self.v.append(np.zeros_like(param))

        self.iter += 1
        lr_t = self.lr * np.sqrt(1.0 - self.beta2**self.iter) / (1.0 - self.beta1**self.iter)

        for i in range(len(params)):
            self.m[i] += (1 - self.beta1) * (grads[i] - self.m[i])
            self.v[i] += (1 - self.beta2) * (grads[i]**2 - self.v[i])

            params[i] -= lr_t * self.m[i] / (np.sqrt(self.v[i]) + 1e-7)
```

3-4 CBOW 모델 구현



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

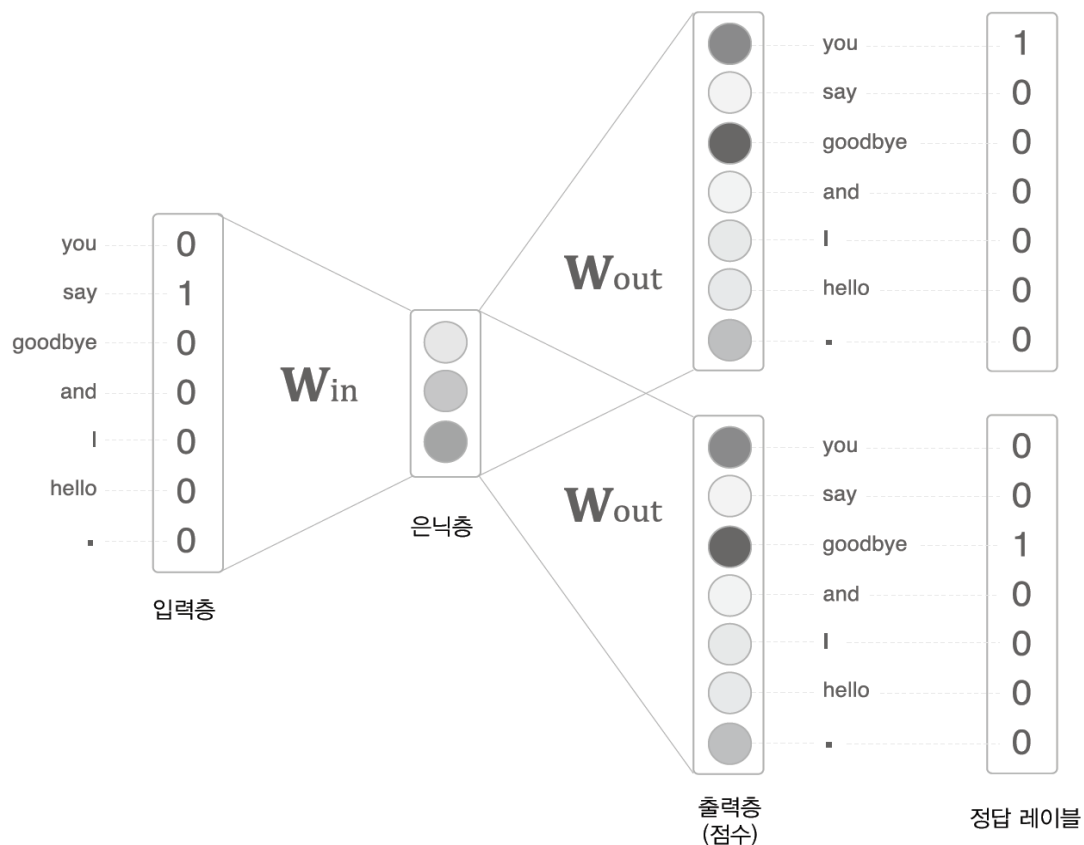
$$L = -\log P(w_t | w_{t-1}, w_{t+1})$$

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

3-4 CBOW 모델 구현

$W_1 \ W_2 \ \cdots \ W_{t-1} \ W_t \ W_{t+1} \ \cdots \ W_{T-1} \ W_T$

그림 3-24 skip-gram 모델의 신경망 구성 예



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

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

3-4 CBOW 모델 구현

```
[39] 1 model = Word2Vec(sentences = tokenized_data, size = 100, window = 5, min_count = 5, workers = 4, sg = 0)
      2 model2 = Word2Vec(sentences=tokenized_data, size=100, window=5, min_count=5, workers=4, sg=1)
      3 model3 = Word2Vec(sentences=tokenized_data, size=100, window=5, min_count=3, workers=4, sg=1)
```

```
[40] 1 print(model.wv.most_similar("꿀잼"))
      2 print(model2.wv.most_similar("꿀잼"))
      3 print(model3.wv.most_similar("꿀잼"))
```

[('재밌음', 0.7869858741760254), ('개꿀잼', 0.7763677835464478), ('졸잼', 0.7677616477012634), ('졸잼', 0.8048923015594482), ('개잼', 0.8038426041603088), ('개꿀잼', 0.7891726493835449), ('개꿀잼', 0.8107367753982544), ('졸잼', 0.7970014810562134), ('개잼', 0.7932615280151367),

```
1 print(model1.wv.most_similar("슬프다"))
2 print(model2.wv.most_similar("슬프다"))
3 print(model3.wv.most_similar("슬프다"))
```

[('짹하다', 0.701218843460083), ('울다', 0.6633522510528564), ('눈물나다', 0.6427698135375977), ('
[('짹하다', 0.7709484100341797), ('애틡하다', 0.7290174961090088), ('짹하다', 0.7252380847930908),
[('짹하다', 0.8130630850791931), ('아리다', 0.7725304365158081), ('눈물나다', 0.7529003620147705),

3-6 정리

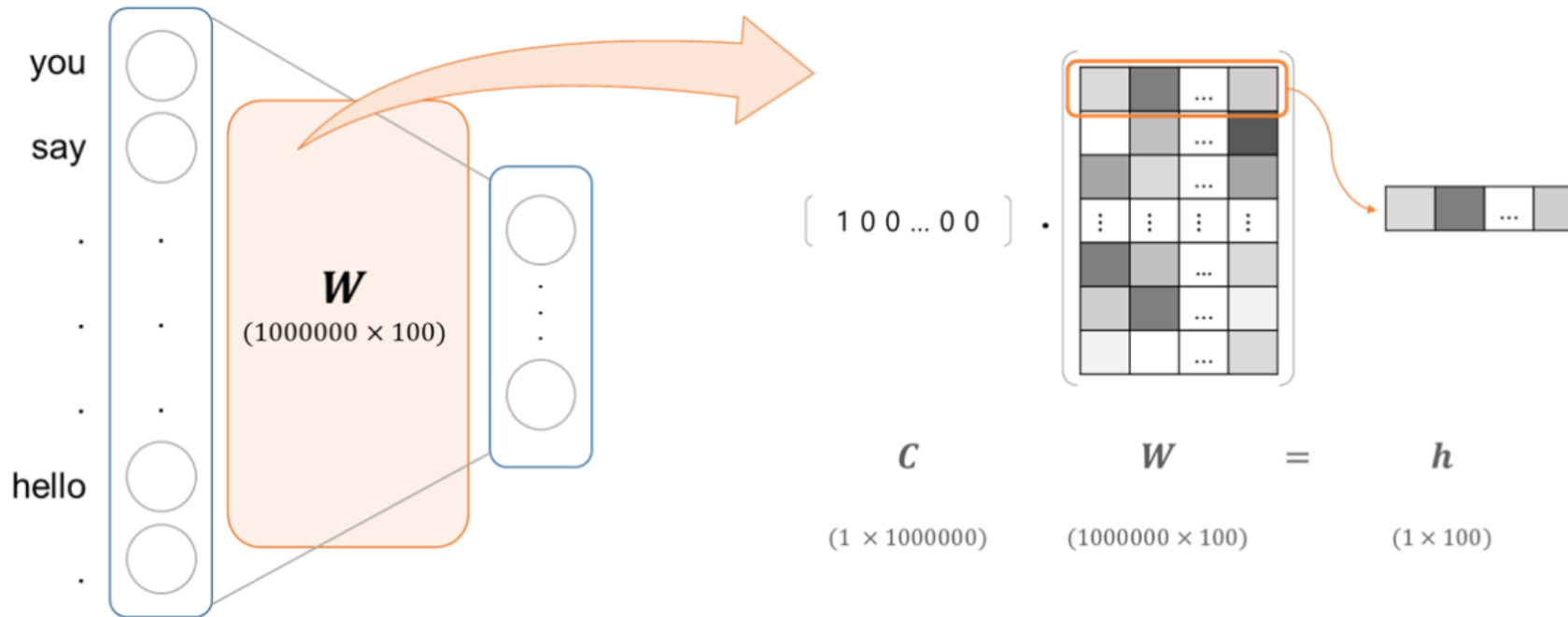
- 추론 기반 기법은 추측하는 것이 목적이며, 그 부산물로 단어의 분산 표현을 얻을 수 있다.
- word2vec은 추론 기반 기법이며, 단순한 2층 신경망이다.
- word2vec은 skip-gram 모델과 CBOW 모델을 제공한다.
- CBOW 모델은 여러 단어(맥락)로부터 하나의 단어(타겟)를 추측한다.
- 반대로 skip-gram 모델은 하나의 단어(타겟)로부터 다수의 단어(맥락)을 추측한다.
- word2vec은 가중치를 다시 학습할 수 있으므로, 단어의 분산 표현 갱신이나 새로운 단어 추가를 효율적으로 수행할 수 있다.

4. Word2Vec 속도 개선

- 문제점
 - 실제 CBOW 모델은 말뭉치에 포함된 어휘 수가 많아지면 계산량도 커집니다.
 - 어휘 수가 어느 정도를 넘어서면 앞 장의 CBOW모델은 계산 시간이 너무 오래 걸린다.
- 개선점
 - 1. Embedding 계층 도입 (은닉층 이전의 처리)
 - (입력층의 원핫 표현과 가중치 행렬 W_{in} 의 곱 계산 해결방안)
 - 단어를 원핫벡터로 다루기 때문에 어휘 수가 많아지면 원핫표현의 벡터 크기도 커진다.
 - -> 상당한 메모리 차지
 - 또한 이 원핫 벡터와 가중치 행렬(W_{in})을 곱하는 연산은 cost가 높다
 - 2. negative sampling 손실함수 도입 (은닉층 이후의 처리)
 - (은닉층과 가중치 행렬 W_{in} 의 곱 및 Softmax계층의 계산 해결방안)
 - 은닉층과 W_{out} 의 곱만 해도 이미 cost가 높는데 Softmax계층에서도 어휘가
 - 많아짐에 따라 계산량도 증가함

4-1 Word2Vec 속도 개선 1

1. Embedding 계층



4-1 Word2Vec 속도 개선 1

2. Embedding 계층 구현

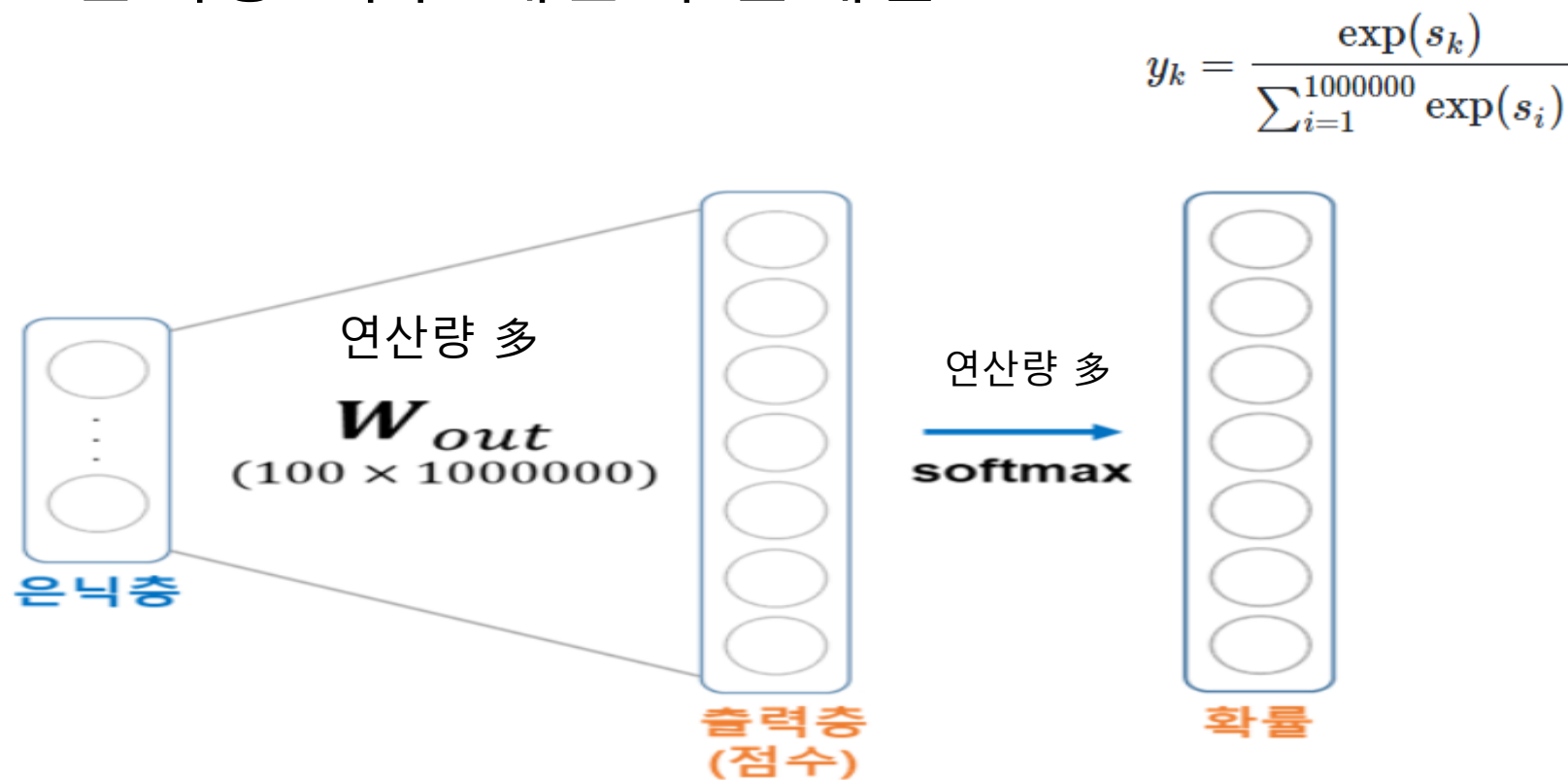
```
# Embedding Layer 구현
# commons/layers.py
class Embedding:
    def __init__(self, W):
        self.params = [W]
        self.grads = [np.zeros_like(W)]
        self.idx = None

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

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

4-2 Word2Vec 속도 개선 2

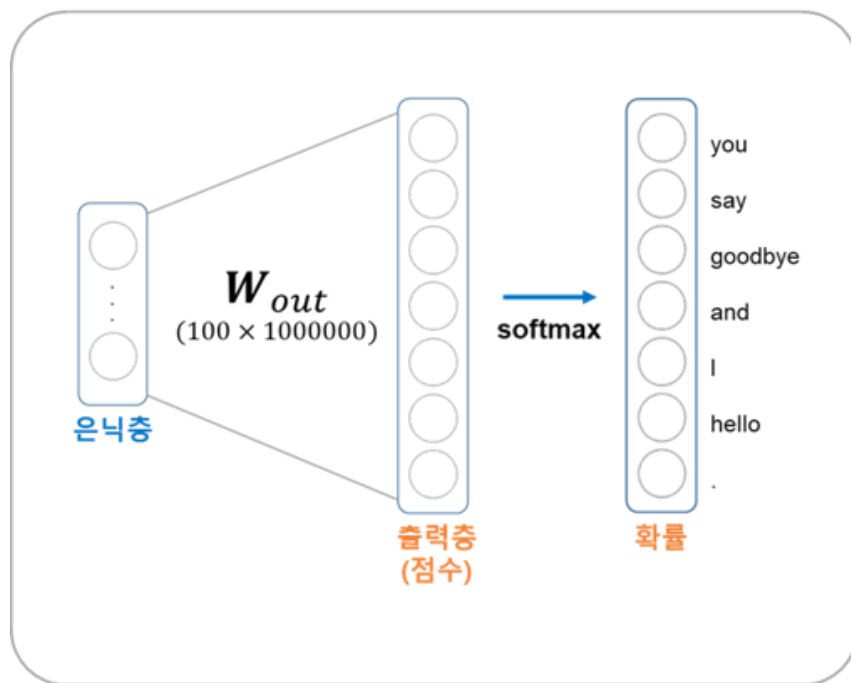
1. 은닉층 이후 계산의 문제점



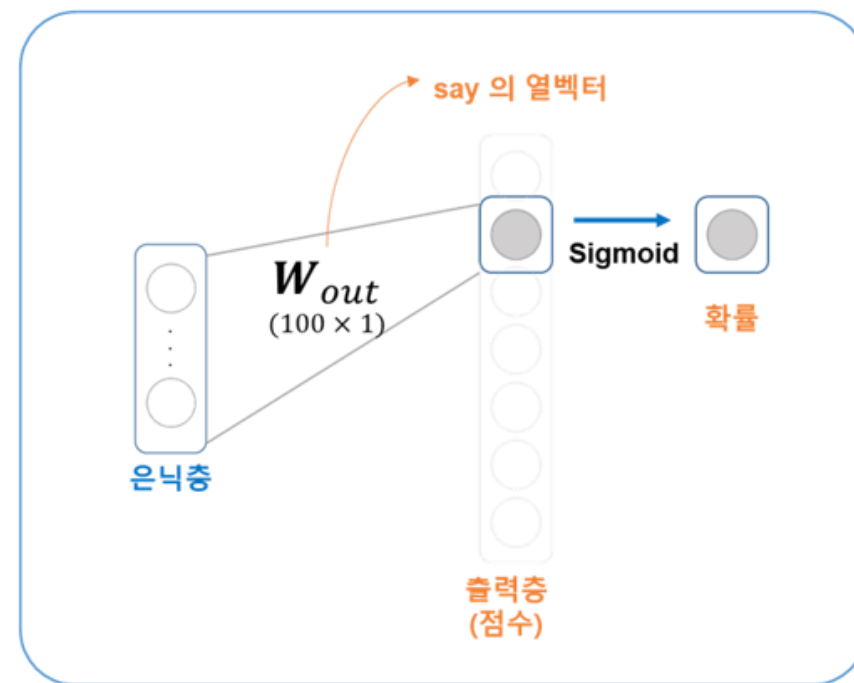
4-2 Word2Vec 속도 개선 2

2. 다중 분류에서 이진 분류로

Multi-class Classification

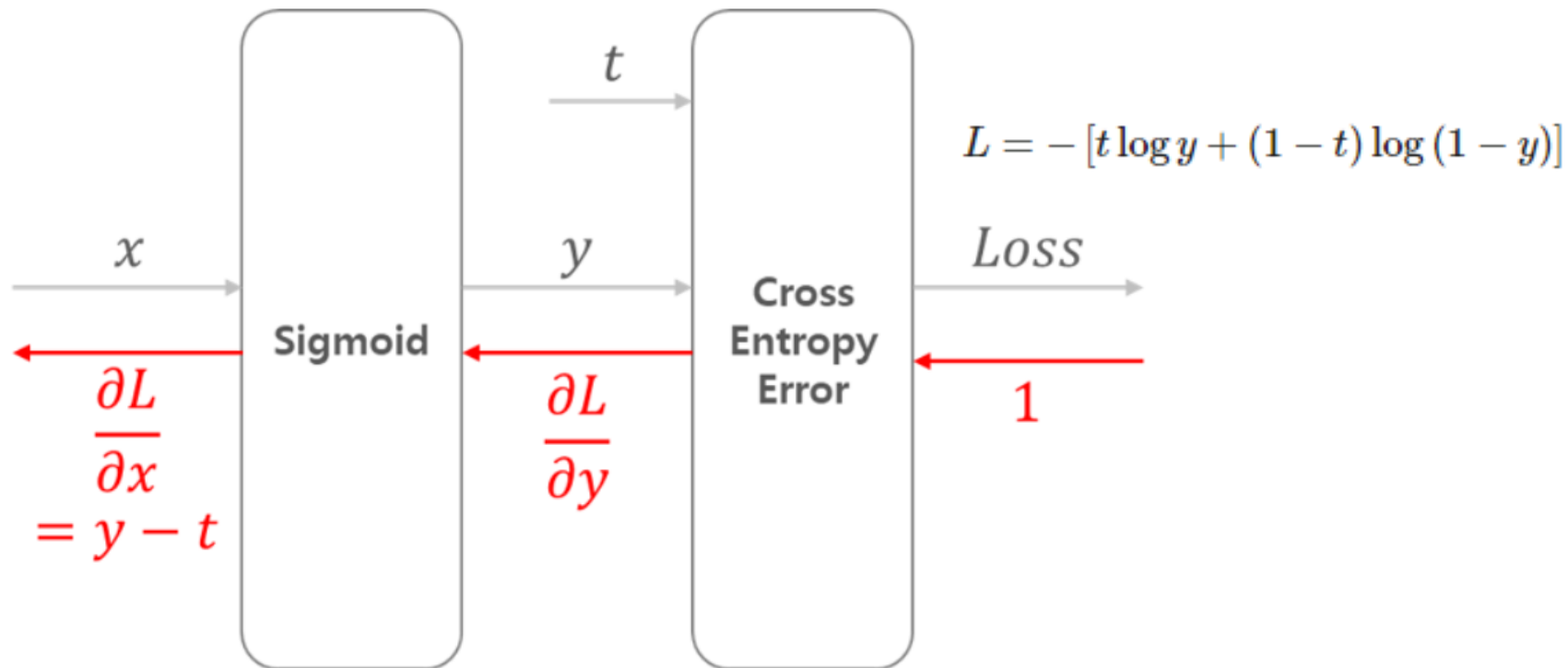


Binary Classification



4-2 Word2Vec 속도 개선 2

3. 시그모이드 함수와 교차 엔트로피 오차



4-2 Word2Vec 속도 개선 2

4. 다중 분류에서 이진 분류로 (구현)

```
class EmbeddingDot:
    def __init__(self, W):
        self.embed = Embedding(W)
        self.params = self.embed.params
        self.grads = self.embed.grads
        self.cache = None

    def forward(self, h, idx):
        target_W = self.embed.forward(idx)
        out = np.sum(target_W * h, axis=1)

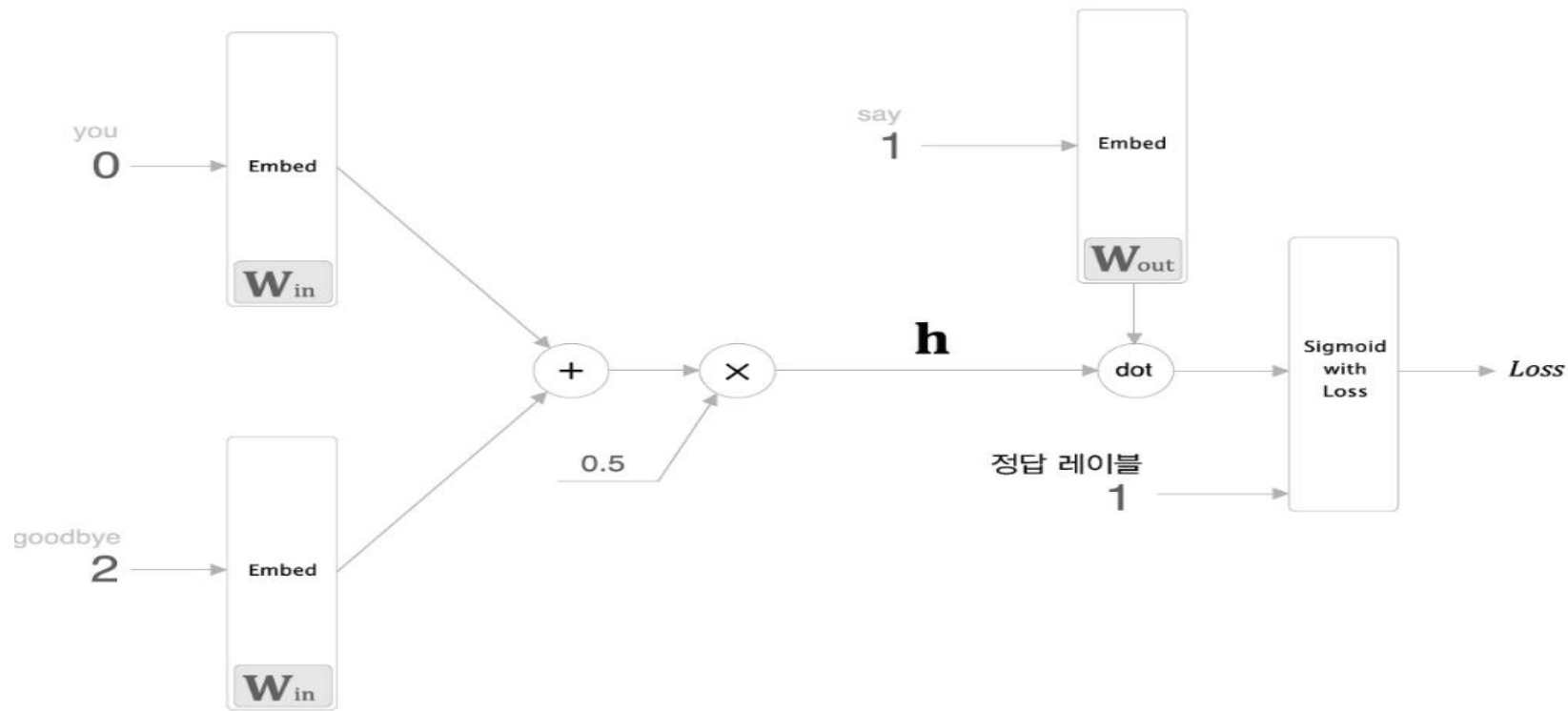
        self.cache = (h, target_W)
        return out

    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
```


4-2 Word2Vec 속도 개선 2

4. 다중 분류에서 이진 분류로 (구현)



4-2 Word2Vec 속도 개선 2

5. 네거티브 샘플링

- 긍정적 예를 타겟으로 한 경우의 손실을 구한다.
- 부정적 예를 몇 개 샘플링하여 그것에 대해서도 마찬가지로 손실을 구한다.
- 각각의 데이터에 대한 손실을 더한 값을 최종 손실로 계산

4-2 Word2Vec 속도 개선 2

6. 네거티브 샘플링의 샘플링 기법

$$P'(w_i) = \frac{P(w_i)^{0.75}}{\sum_{j=1}^n P(w_j)}$$

In [7]:

```
p = [0.7, 0.29, 0.01]
new_p = np.power(p, 0.75)

new_p /= np.sum(new_p)
print(new_p)
```

```
[0.64196878 0.33150408 0.02652714]
```

4-2 Word2Vec 속도 개선 2

7. 네거티브 샘플링의 샘플링 구현

```
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

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

4-3 개선판 word2vec 학습

1. CBOW 모델 구현

```
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')

        # 레이어 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_vecs1 = W_in
        self.word_vecs2 = W_out
```

4-3 개선판 word2vec 학습

2. CBOW 모델 학습

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

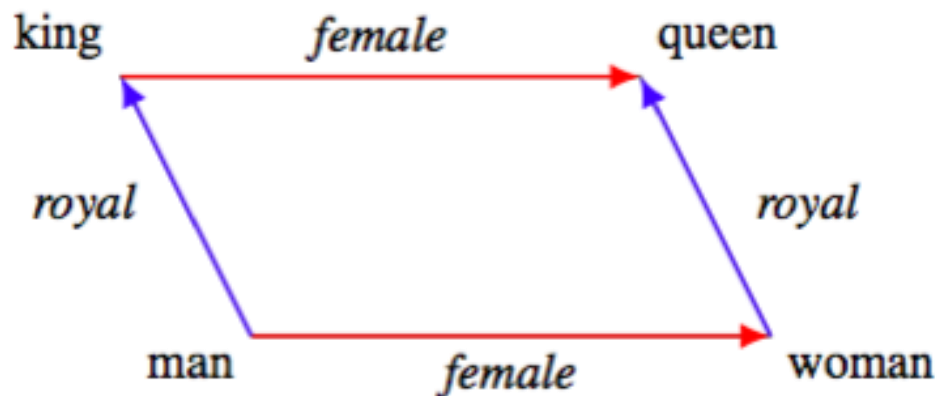
contexts, target = create_contexts_target(corpus, window_size)
if config.GPU:
    contexts, target = to_gpu(contexts), to_gpu(target)

# 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()
```

4-3 개선판 word2vec 학습

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)
```

```
[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
```

4-5 정리

- Embedding 계층은 단어의 분산표현을 담고 있으며, 순전파 시 지정한 단어 ID의 벡터를 추출한다.
- Word2vec은 어휘 수의 증가에 비례하여 계산량도 증가하므로, 근사치로 계산하는 빠른 기법을 사용하면 좋다.
- 네거티브 샘플링은 부정적 예를 몇 개 샘플링하는 기법으로, 이를 이용하면 다중 분류를 이진 분류처럼 취급할 수 있다.
- Word2vec으로 얻은 단어의 분산 표현에는 단어의 의미가 녹아들어 있으며, 비슷한 맥락에서 사용되는 단어는 단어 벡터 공간에서 가까이 위치한다.
- Word2vec의 단어의 분산 표현을 이용하면 유추 문제를 벡터의 덧셈과 뺄셈으로 풀 수 있게 된다.
- Word2vec은 전이 학습 측면에서 특히 중요하며, 그 단어의 분산 표현은 다양한 자연어 처리 작업에 이용할 수 있다.