word2vec 속도 개선

3장에서 구현한 SimpleCBOW의 경우 작은 말뭉치에서는 잘 작동하지만, 말뭉치의 개수가 커지면

- 문제 1 : 입력층의 원핫 표현, W_{in}(hidden layer로의 weight)과의 곱
- 문제 2 : W_{out}(output layer로의 weight)과의 곱 및 softmax계층의 계산
- 해결방법 : Embedding 계층(문제1), 네거티브 샘플링 도입(문제2)

1. Embedding Layer

- input과 W_{in}의 행렬곱은 W_{in}의 특정 행을 추출하는 것에 불과하기 때문에 one-hot vector로의 변환과 행렬곱 계산은 필요하지 않음
  - one-hot vector와의 연산이기 때문
- Embedding Layer : 가중치 매개변수로부터 "단어 ID에 해당하는 행(벡터)"를 추출하는 계층
  - word Embedding에서 용어가 유래되었음
  - *word Embedding :* 단어의 밀집벡터 표현*(*통계 기반 기법 *: distributional representation,* 추론 기반 기법 *: distributed representation)*

Embedding Layer 구현

- 순전파 : W_{in}의 특정 idx 선택
- 역전파 : 앞 layer에서 오는 gradient를 특정 idx의 gradient로 그대로 흘려보냄

In [2]:

```
import numpy as np
W = np.arange(21).reshape(7,3)
print(W[2]) # W의 2번째 추출
print(W[5]) # W의 5번째 추출
idx = np.array([1,0,3,0]) # 1, 0, 3, 0번째 행 추출
print(W[idx])
```

```
[6 7 8]
[15 16 17]
[[ 3  4  5]
 [ 0  1  2]
 [ 9 10 11]
 [ 0  1  2]]
```

In [5]:

```
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
        # 실제로는 굳이 weight와 같은 형상을 만들고 update할 필요 없이, idx를 통해 특정 행만을 갱신할
        dW[...]=0 # dW의 형상은 유지한 채, 그 원소들을 0으로 덮어씀
        dW[self.idx] = dout # 특정 idx의 gradient만 dout으로 설정, 나머지는 0
```

-> 할당의 방법이기 때문에 idx가 중복될 때 문제가 있음, 더하는 방법을 통해서 gradient를 계산해주어야 함

In [7]:

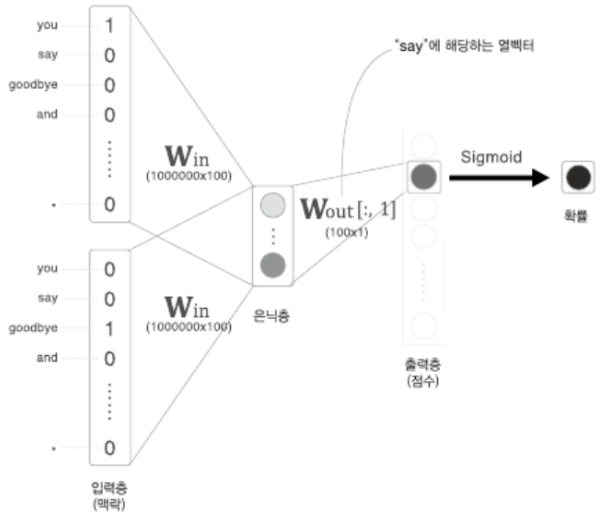
```
class Embedding:
    def __init__(self,W):
        self.params = [W]
        self.grads = [np.zeros_like(W)]
        self.idx = None

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

    def backward(self, dout):
        dW, = self.grads
        dW[...]=0
        """
        for i, word_id in enumerate(self.idx):
            dW[word_id]+=dout[i] # dout 의 i번째 행은 weight의 idx에 대응
        """
        # np.add.at(A,idx,B) B를 A의 idx번째 행에 더해줌(최적화가 적용되어 있기 때문에 for문보다 빠름
        dW = np.add.at(dW,idx,dout)
```

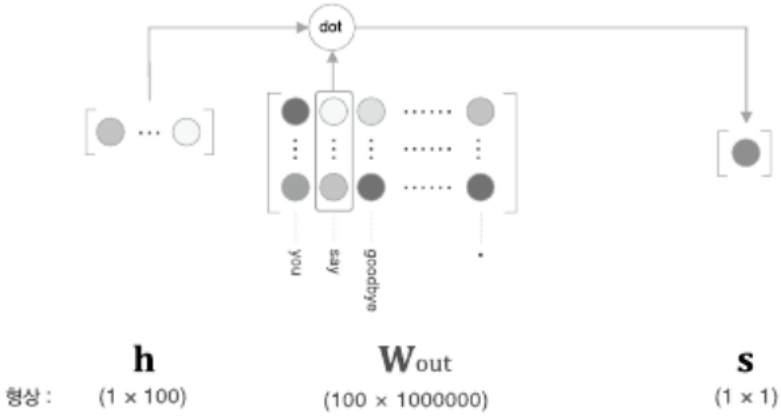
2. 네거티브 샘플링

다중분류에서 이진분류로

- CBOW에서의 다중분류 : 맥락이 "you"와 "goodbye"일 때 target 단어는 무엇인가?
  - hidden layer -> output layer : hidden layer output과 W_{out} 전체와의 내적계산
  - output activation function : softmax
  - 최종 output : 각 단어가 target일 확률 출력
  - Loss : CEE

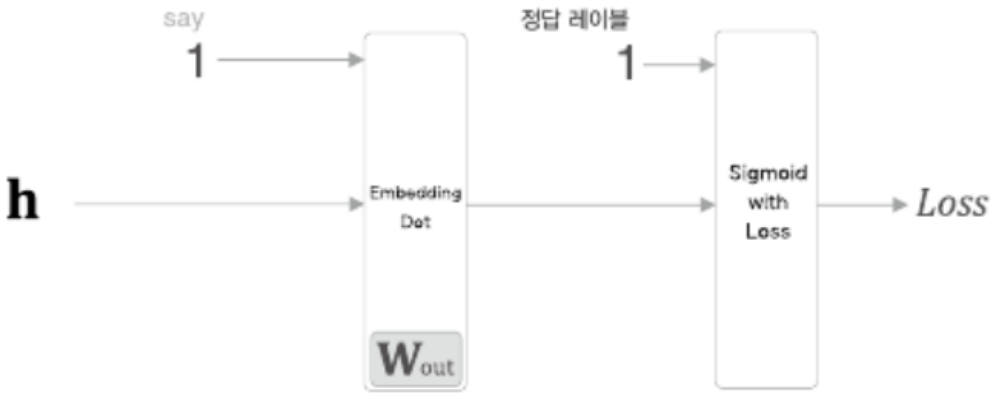
- CBOW에서의 이중분류 : 맥락이 "you"와 "goodbye"일 때 target 단어는 "say"입니까?
  - hidden layer -> output layer : hidden layer output과 W_{out}의 "say"에 해당하는 단어벡터와의 내적계산
  - output activation function : sigmoid
  - 최종 output : target 단어가 say일 확률 출력
  - Loss : CEE
    - $L = -\sum_{i=1}^{n} t_i(logy_i) + (1-t_i)(log(1-y_i))$(데이터 개수 : n)
    - 실제 sigmoid의 최종 출력은 target이 1인 것이고, 그것에 대한 미분만 하기 때문에 Sigmoid-Loss layer의 역전파시에는 y-t가 전파됨





Embedding Dot 계층 도입

- Embedding layer을 사용하기 위해 기존의 W_{out}을 transpose한 것으로 W_{out}이 새로 정해짐



다중분류에서 이진분류로 구현

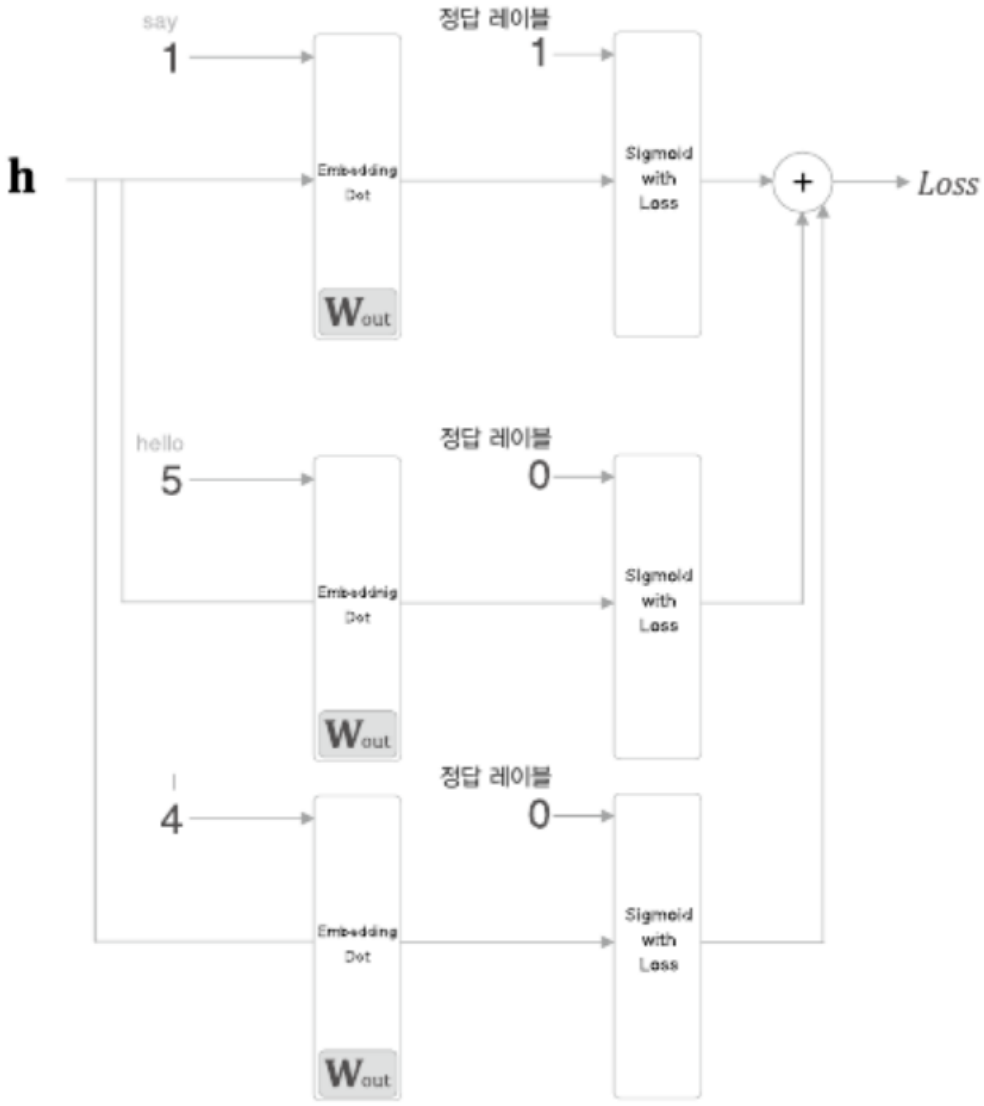
In [8]:

```
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):
        # h : 은닉층 뉴런, idx : 단어 id의 넘파이 배열(mini-batch라면)
        target_W = self.embed.forward(idx)
        out = np.sum(target_W*h, axis=1) # 내적 수행

        self.cache = (h, target_W)
        return out

    def backward(self,dout):
        h, target_W = self.cache
        # 적절한 브로드 캐스팅을 위해 reshape(dout이 열방향으로 복제되어야 하기 때문)
        dout = dout.reshape(dout.shape[0],1)

        # output layer에서 오는 역전파
        dtarget_W = dout*h
        dh = dout * target_W # h의 gradient

        # Embedding layer에서 오는 역전파
        self.embed.backward(dtarget_W)

        return dh
```

네거티브 샘플링

- 이진분류로 바꾸면서, 정답(긍정적) target에 대해서만 학습하게 됨. 정답이 아닌(부정적) target에 대해서는 훈련이 되지 않았음.
- 모든 부정적 예를 대상으로 이진 분류 학습 : 어휘 수가 늘어나면 감당할 수 없기 때문에 좋은 방법이 아님
  -> 적은 수의 부정적 예를 샘플링(5/10등)해서 사용(네거티브 샘플링)

- 최종 손실 : 긍정적 예를 target으로 한 손실 + 몇 개의 부정적 예(샘플링된)를 target으로 한 손실
  - 각 데이터 별로 해서 다중분류처럼 *softmax*할 수도 있지만, *negative_sample+1*이 *data*보다 훨씬 적기 때문에 해당 과정으로 하는 것이 효과적



네거티브 샘플링 기법

- 각 단어의 말뭉치에서의 출현 횟수를 이용하여 샘플링
  - 자주 등장하는 단어를 많이 추출하고, 드물게 등장하는 단어를 적게 추출하기 위함
    - 현실에서도 희소한 단어는 거의 출현하지 않음. 즉 희소한 단어에 대한 훈련을 할 필요성보다는 자주 등장하는 단어에 대한 훈련을 할 필요성이 있음. 그래야 전체적인 성능이 좋음
- 방법
  1. 말뭉치에서의 단어별 출현 횟수를 바탕으로 확률분포를 구함
     - 수정된 확률분포 : $p'(w_i) = p(w_i)^{0.75} / \sum_j^n p(w_j)^{0.75}$
     - "0.75 제곱"으로 수정한 후에도 확률의 총합을 1로 맞춰주기 위해서 분모로 수정 후 확률분포의 총합(0.75외의 다른 값도 괜찮)
     - 수정된 확률분포를 사용하는 이유 : 원래 확률이 낮은 단어의 확률을 살짝 높임으로써 출현 확률이 낮은 단어를 버리지 않기 위해서
  2. 해당 확률분포에 따라서 샘플링을 수행

네거티브 샘플링 구현

In [11]:

```
import numpy as np

# 0에서 9까지의 숫자 중 하나를 무작위로 샘플링
print(np.random.choice(10))
print(np.random.choice(10))

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

# 5개만 무작위로 샘플링(중복 있음)
print(np.random.choice(words, size = 5))

# 5개만 무작위로 샘플링(중복 없음)
print(np.random.choice(words, size = 5, replace=False))

# 확률분포에 따라 샘플링
p = [0.5, 0.1, 0.05, 0.2, 0.05, 0.1]
print(np.random.choice(words, p=p))
```

```
2
6
you
['you' 'say' 'hello' 'goodbye' 'say']
['I' 'you' 'hello' 'say' '.']
you
```

In [13]:

```
p = [0.7, 0.29, 0.01]
new_p = np.power(p, 0.75) # 0.75제곱
new_p /= np.sum(new_p) # 총합(분모)
print(new_p)
```

```
[0.64196878 0.33150408 0.02652714]
```

In [16]:

```
import collections
```

In [20]:

```
GPU = False
```

In [21]:

```
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)

    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, rep
        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
```

In [22]:

```
# 각 타겟에 대한 부정적 샘플을 2개씩 뽑아줌
corpus = np.array([0,1,2,3,4,1,2,3])
power = 0.75
sample_size = 2

# corpus : 단어 ID목록, power : 확률분포에 제곱할 값, sample size : 부정적 예 샘플링 수행 횟수
sampler = UnigramSampler(corpus, power, sample_size)
target = np.array([1,3,0])
negative_sample = sampler.get_negative_sample(target)
print(negative_sample)
```

```
[[2 3]
 [0 1]
 [2 1]]
```

네거티브 샘플링 구현

In [ ]:

```
class NegativeSamplingLoss:
    def __init__(self,W,corpus,power=0.75,sample_size=5):
        self.sample_size = sample_size

        # negative sampling class 생성
        self.sampler = UnigramSampler(corpus,power,sample_size)

        # loss Layer(positive + negative data) 생성
        self.loss_layers = [SigmoidWithLoss() for _ in range(sample_size+1)]

        # Enbedding Layer(positive + negative data) 생성
        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]

        # 실제 negative 샘플링
        negative_sample = self.sampler.get_negative_sample(target)

        # 긍정적 예 순전파
        score  = self.embed_dot_layers[0].forward(h, target) # score
        correct_label = np.ones(batch_size, dtype=np.int32) # label은 모두 0
        loss = self.loss_layer[0].forward(score, correct_label) # loss 계산

        # 부정적 예 순전파
        negative_label = np.zeros(batch_size, dtype=np.int32) # label은 모두 0
        for i in range(self.sample_size):
            negative_target = negative_sample[:,i] # 각 데이터의 첫번째 negative sample
            score = self.embed_dot_layers[1+i].forward(h, negative_target)
            loss += self.loss_layers[1+i].forward(score, negative_label)

        return loss

    def backward(self, dout=1):
        dh = 0
        # 같은 h를 공유하기 때문에 각 dh를 모두 더함
        for l0, l1 in zip(self.loss_layers, self.embed_dot_layers):
            dscore = l0.backward(dout)
            dh += l1.backward(dscore)

        return dh # 최종 dh
```