[3주차] 언어모델, BoW, word2vec

Ch 3. 언어 모델

1. 언어모델

언어모델(Language Model)이란?

단어 시퀀스(문장)들에 다음에 올 가장 자연스러운 단어 시퀀스를 찾기 위해 확률을 할당하는 모델

1. 일방향적 모델 | 이전 단어들이 주어졌을 때 다음 단어를 예측 ex. 검색엔진

   ⇒ 조건부 확률 활용

   $$P(w_n|w_1,...,w_{n-1})$$

   전체단어 시퀀스

   $$P(W)=P(w_1,w_2,w_3,w_4...w_n)=\prod_{i=1}^{n}P(w_i|w_1,w_2,...w_{i-1})$$

2. 양방향적 모델 | 양쪽의 단어가 주어졌을 때 가운데 단어를 예측 ex. BERT

단어 시퀀스의 확률 할당의 예시

a. 기계 번역

   P(나는 버스를 탔다) > P(나는 버스를 태운다)

b. 오타 교정

   선생님이 교실로 부리나케

   P(달려갔다) > P(잘려갔다)

c. 음성 인식

P(나는 메론을 먹는다) > P(나는 메롱을 먹는다)

등에 활용되어 보다 적절한 문장을 판단하도록 함.

💡 통계적 모델과 인공 신경망의 두 가지로 구분되며, 최근에는 인공 신경망을 활용하는 추세

2. 통계적 언어 모델 (Statistical Language Model a.k.a SLM)

작동 방식

문장에 대한 확률 (by 조건부 확률)

문장 'An adorable little boy is spreading smiles'의 확률

```
P(An adorable little boy is spreading smiles) =
P(An)xP(adorable|An)xP(little|An adorable)xP(boy|An adorable little)x
...xP(smiles|An adorable little boy is spreading)
```

카운트 기반 접근

이전 단어에 대한 다음 단어의 확률

$$P(\text{is}|\text{An adorable little boy}) = \frac{\text{count}(\text{An adorable little boy is})}{\text{count}(\text{An adorable little boy })}$$

한계점 | 코퍼스에 An adorable little boy라는 단어 시퀀스가 존재해야만 확률 측정 가능 ⇒ 방대한 양의 데이터가 필요함 ⇒ **희소 문제(Sparsity problem)** 발생

cf) 희소 문제 : 충분한 데이터를 관측하지 못해 언어를 정확히 모델링하지 못하는 문제

N-gram 언어 모델

- 카운트를 기반으로 하며 희소 문제를 해결하기 위한 방법 중 하나

📌 **Idea**

앞 단어 중 임의의 개수만 포함해서 해당 단어의 시퀀스를 카운트할 확률을 높임.

$$P(is|An\ adorable\ little\ boy) \approx P(is|boy)$$

- N-gram이란

  임의의 개수를 정하는 기준

  n개의 연속적인 단어 나열을 의미

  ex) n = 3

$$P(insults|boy\ is\ spreaading) = 0.5$$
$$P(smiles|boy\ is\ spreading) = 0.2$$

  ⇒ insults 선택

- 한계
  - 여전히 희소 문제 존재
  - 맥락을 파악하기 위해서는 n이 커질수록 good (근사의 정확도 ↑)
  - n이 커질수록 희소 문제의 심각성 ↑ (= 문장의 희소성 ↑)

💡 희소 문제의 근본적 해결 불가의 문제는 NLP 연구의 흐름이 인공 신경망 언어 모델로 이동하는 계기가 됨.

3. 한국어 언어 모델

한국어 언어 모델의 예측이 어려운 이유

- 기존의 통계적 언어 모델은 조건부 확률을 이용하므로 순서의 흐름이 중요하게 작용 but 한국어는 어순이 중요하지 않음!

▼ 한국어는 교착어

교착어란 ?

실질적인 의미를 가진 단어 또는 어간에 문법적인 기능을 가진 요소가 차례차례로 결합함으로써 문장 속에서의 문법적인 역할이나 관계의 차이를 나타내는 언어. ex) 그녀는, 그녀를, 그녀와

⇒ 토큰화를 통해 접사와 조사의 분리가 필요

- 띄어쓰기를 틀리는 경우가 많음.

4. 모델 평가 지표 (Metric) - Perplexity (PPL)

Perplexity란?

Perplexity : '**당혹감**', '**혼란**'

일반적으로 PPL의 의미 : 발생할 수 있는 문장의 가짓수

> **"Perplexity가 높다" = "발생할 수 있는 문장의 가짓수가 많다" = "혼란스럽다" = "원하는 결과를 얻지 못할 가능성이 크다"**

⇒ PPL은 낮을수록 좋은 성능을 가짐.

$$PPL(W) = P(w_1, w_2, ..., w_N)^{-\frac{1}{N}} = \sqrt[N]{\frac{1}{P(w_1, w_2, ..., w_N)}} = \sqrt[N]{\frac{1}{\prod_{i=1}^{N} P(w_i|w_1, w_2, ..., w_{i-1})}}$$

주의할 점

- PPL 값이 낮다는 것은 테스트 데이터에서의 높은 정확도를 의미하는 것이지 사람이 느끼기에 좋은 언어 모델임을 의미하진 않음. (테스트 데이터 내에 해당 정답 샘플에 대한 발생 확률이 높다는 것)
- 두 개 이상의 언어 모델을 비교할 때 정량적으로 양이 많고, 도메인에 적절한 코퍼스에 대한 테스트 데이터를 사용해야 함.

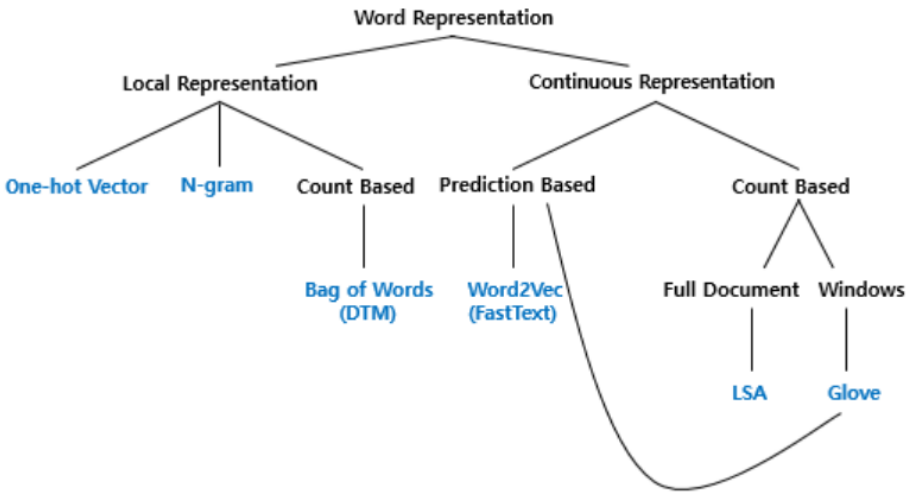
Ch 4. 카운트 기반의 단어 표현

1. 다양한 단어의 표현 방법

단어의 표현 방법

- 국소 표현 (Local Representation)
  - 해당 단어 자체만 보고 특정값을 매핑하여 단어를 표현하는 방법
  - ex) puppy(1), cute(2), lovely(3)
  - 단어의 뉘앙스 파악 불가능
- 분산 표현
  - 주변 단어를 참고하여 단어를 표현하는 방법
  - ex) puppy : cute, lovely 라고 정의
  - 단어의 뉘앙스 파악 가능

단어 표현의 카테고리화



2. Bag of Words(BoW)

Bag of Words란?

문서에 있는 문장들을 모두 단어로 토큰화 하여 이 단어들을 무작위로 섞었을 때 특정 단어의 출현 빈도를 수치화한 표현 방법

How?

1. 각 단어에 고유한 정수 인덱스 부여
2. 각 정수 인덱스에 대해 이들이 등장한 횟수를 벡터로 만듦.

Where?

문서 분류 문제 or 여러 문서 간의 유사도를 구할 때 사용

ex) '미분', '방정식', '부등식' 등이 많이 나오면 수학 관련 문서로 분류

BoW 구현

```
from konlpy.tag import Okt

okt = Okt()

def build_bag_of_words(document):
  # 온점 제거 및 형태소 분석
  document = document.replace('.','')
  tokenized_document = okt.morphs(document)

  word_to_index = {}
  bow = []

  for word in tokenized_document:
    if word not in word_to_index.keys():
      word_to_index[word] = len(word_to_index) # 정수 인덱싱
      bow.insert(len(word_to_index) - 1,1)
    else:
      # 재등장하는 단어의 인덱스
      index = word_to_index.get(word)
      # 재등장한 단어는 해당하는 인덱스의 위치에 1을 더한다.
      bow[index] = bow[index] + 1

  return word_to_index, bow
```

```
doc = "고려대학교에는 안암역과 고려대역이 있다. 안암역 앞에는 오샬이 있고 고려대역 앞에는 고래돈까스가 있다. "
vocab, bow = build_bag_of_words(doc2)
print('vocabulary :', vocab)
print('bag of words vector :', bow)
```

```
vocabulary : {'고려대학교': 0, '에는': 1, '안암역': 2, '과': 3, '고려대역': 4, '이': 5, '있다': 6, '앞': 7, '오샬': 8, '있고': 9, '고래': 10, '돈까스': 11, '가': 12}
bag of words vector : [1, 3, 2, 1, 2, 2, 2, 2, 1, 1, 1, 1, 1]
```

`CountVectorizer` **클래스로 BoW 생성**

```
from sklearn.feature_extraction.text import CountVectorizer

corpus = ['you know I want your love. because I love you.']
```

```
vector = CountVectorizer()

print('bag of words vector :', vector.fit_transform(corpus).toarray())

print('vocabulary : ',vector.vocabulary_)
```

```
bag of words vector : [[1 1 2 1 2 1]]
vocabulary :  {'you': 4, 'know': 1, 'want': 3, 'your': 5, 'love': 2, 'because': 0}
```

주의

- `CountVectorizer` 함수는 길이가 2이상인 문자만 토큰으로 인식, 나머지는 제거
- 띄어쓰기만을 기준으로 토큰화하기 때문에 한국어에는 적용하기 어려움

```
bag of words vector : [[1 1 1 1 1 1 2 1 1 2]]
vocabulary :  {'고려대학교에는': 3, '안암역과': 5, '고려대역이': 2, '있다': 9, '안암역': 4, '앞에는': 6, '오샬이': 7, '있고': 8, '고려대역': 1, '고래돈까스가': 0}
```

불용어를 제거하여 BoW 생성

```
from sklearn.feature_extraction.text import CountVectorizer
from nltk.corpus import stopwords
```

1. 사용자 정의

   `CountVectorizer(stop_words=[ ])`

```
text = ["Family is not an important thing. It's everything."]
vect = CountVectorizer(stop_words=["the", "a", "an", "is", "not"])
print('bag of words vector :',vect.fit_transform(text).toarray())
print('vocabulary :',vect.vocabulary_)
```

```
bag of words vector : [[1 1 1 1 1]]
vocabulary : {'family': 1, 'important': 2, 'thing': 4, 'it': 3, 'everything': 0}
```

2. `CountVectorizer`에 내장된 자체 불용어

`CountVectorizer(stop_words= 'english')`

```
bag of words vector : [[1 1 1]]
vocabulary : {'family': 0, 'important': 1, 'thing': 2}
```

3. `NLTK`에서 지원하는 불용어 사용

```
stopwords = stopwords.words('english')
vect = CountVectorizer(stop_words= stopwords)
```

```
bag of words vector : [[1 1 1 1]]
vocabulary : {'family': 1, 'important': 2, 'thing': 3, 'everything': 0}
```

TF-IDF (단어 빈도-역 문서 빈도)

BoW을 여러 문서에 적용하여 행렬로 표현한 문서 단어 행렬(DTM)에 대해 각 단어들의 중요도 계산하는 방법

d : 문서, t: 단어의 수, n: 총 문서의 수

1. tf(d,t) : 특정 문서 d에서의 특정 단어 t의 등장 횟수
2. df(t) : 특정 단어 t가 등장한 문서의 수
3. idf(d,t) : df(t)에 반비례하는 수

$$idf(d,t) = log(\frac{n}{1+df(t)})$$

$$TF-IDF = tf(d,t) * idf(d,t)$$

TF-IDF는 모든 문서에서 자주 등장하는 단어는 중요도가 낮다고 판단하며, 특정 문서에서만 자주 등장하는 단어는 중요도가 높다고 판단

`sklearn`에서 `TfidfVectorizer`를 통해 쉽게 적용할 수 있음.

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

Ch 9. Word2Vec

- 추론 기반 기법 : 모든 데이터를 학습하여 사용하는 것이 아니라 미니배치를 활용하여 학습
- 추론 기반 기법의 작동 방식

```
You _____ goodbye and I say hello.
```

위와 같은 빈칸 추론 문제를 계속해서 풀면서 단어의 출현 패턴을 학습

- 통계 추론 기법에 비해 가지는 장점
  - 단어의 분산 표현 갱신이 효율적
  - 단어 간 연산 가능

    ex1) 한국 - 서울 + 도쿄 = 일본

    ex2) king - man + woman = queen

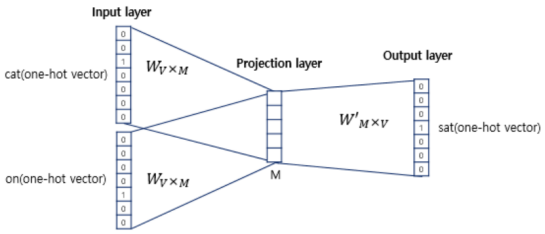
1. word2vec 모델 구조와 특징

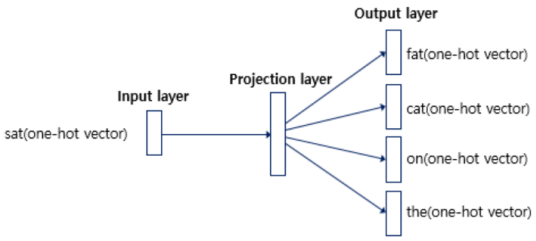
1. 원-핫 벡터를 이용하여 표현함.
2. 은닉층이 1개인 얕은 신경망
3. W와 W'은 서로 다른 행렬이고 역전파를 통해 정확도를 높이도록 업데이트
4. 은닉층에 활성화 함수가 존재하지 않음.

2. word2vec 학습 방식

1. **CBOW (Continuous Bag of Words)**
   a. 주변에 있는 단어들을 입력으로 하여 중간 단어를 예측하는 방법
   b. 학습 속도가 빠름

2. **Skip-Gram**
   a. 중간에 있는 단어들을 입력으로 주변 단어를 예측하는 방식 → 더 어려움
   b. 말뭉치가 커질수록 저빈도 단어나 유추 문제의 성능 면에서 skip-gram이 뛰어난 성능을 보임.





분산 표현

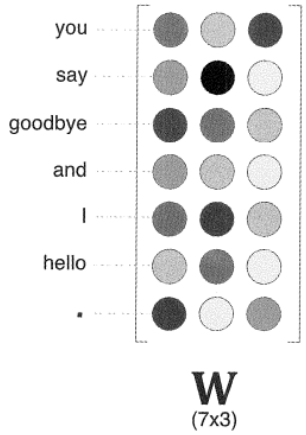
단어의 의미를 여러 차원에 분산하여 표현

⇒ 단어 벡터간 유의미한 유사도 측정 가능

word2vec에서는

W_{in}만을 최종 단어의 분산 표현으로 사용함.

그림 3-10 가중치의 각 행이 해당 단어의 분산 표현이다.



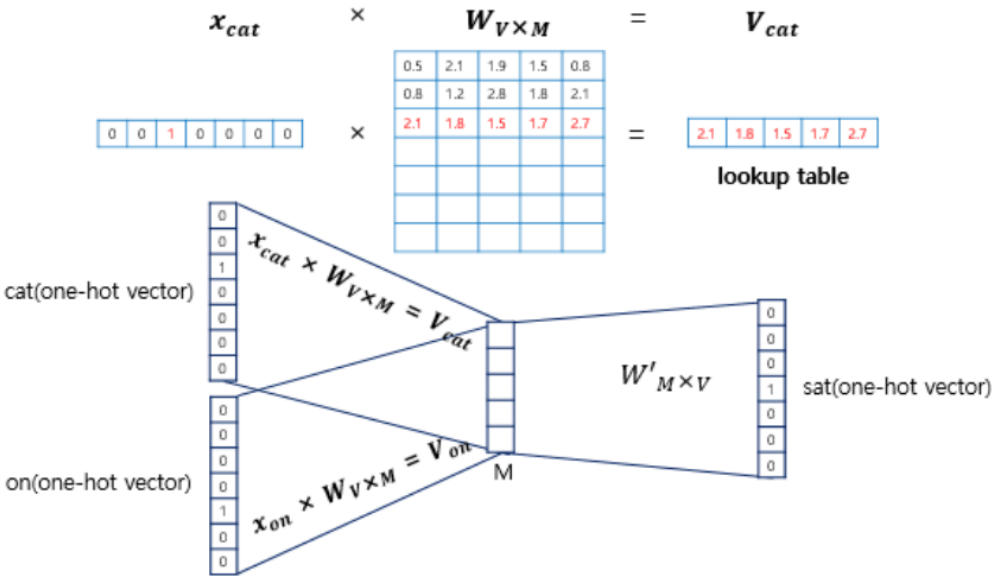
3. CBOW 모델

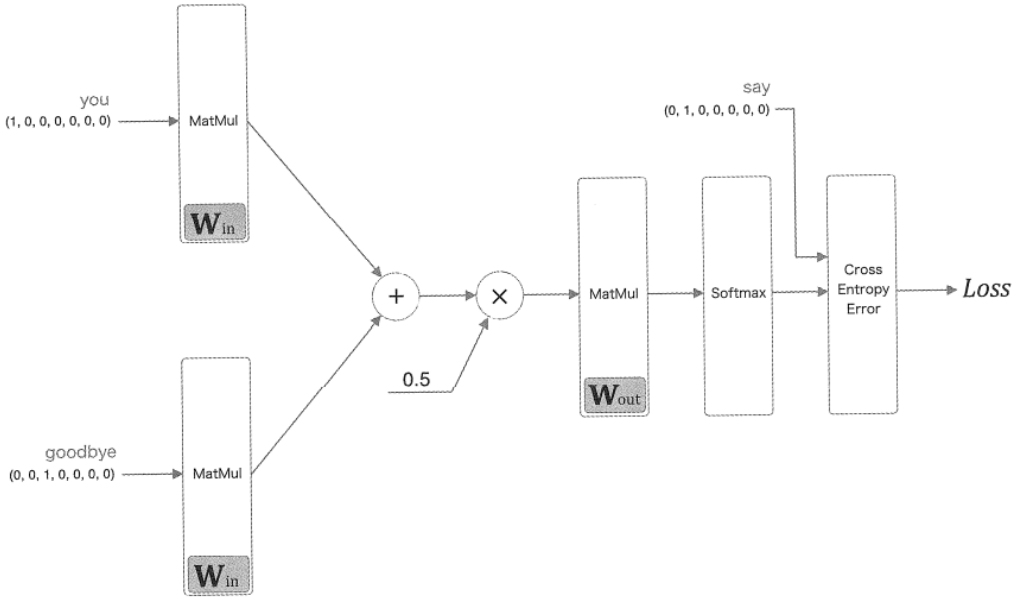
ex) You _____ goodbye and I say hello.

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

w_{t-1}, w_{t+1} 이 주어졌을 때, w_t가 일어날 확률

은닉층





window_size = 1 일 때,

input layer의 shape : [1,7] x 2

W_in의 shape : [7,5]

hidden_layer의 shape : [1,5] ⇒ W_in 행렬에서 window 인덱스들에 대한 행 평균이 필요

W_out의 shape : [5,7]

output layer의 shape : [7,1] ⇒ 입력 벡터 길이로 변환됨 → 각 단어의 score를 의미

Softmax 함수와 Cross Entropy Loss

다중 분류 문제 ⇒ Output layer의 활성화 함수로 cost 함수 이용

softmax function

$$y_k = \frac{exp(S_k)}{\sum_{i=1}^{M} exp(S_i)}$$

cross entropy function

$$L = -\frac{1}{V}\sum_{j=1}^{V} log(\hat{y}_j) = -\frac{1}{V}\sum_{j=1}^{V} log(P(w_t|w_{t-1}, w_{t+1}))$$

4. CBOW 모델의 개선

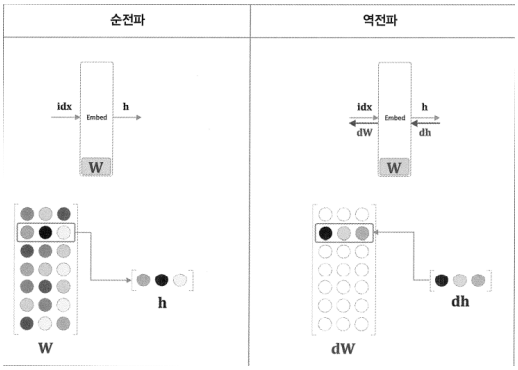
기존 CBOW 모델의 문제

- 코퍼스의 어휘 수가 많아지면 벡터의 길이가 커지고(원-핫 벡터의 한계), 이에 따라 연산량이 증가 ⇒ Embedding을 통해 해결
- 같은 이유에서 softmax 함수의 연산량 증가 ⇒ 네거티브 샘플링을 통해 해결

Embedding

💡 **Idea**

은닉층의 계산은 결국 행렬의 특정 행을 추출하는 것뿐이므로 행렬 곱 계산을 실행할 필요가 없음




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

  def forward(self,idx) :
    print(self.params)
    W, = self.params
    self.idx = idx
    out = W[idx] # 해당 열 추출
    return out

def backward(self,dout) :
    dW, = self.grads

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

    return dW
```

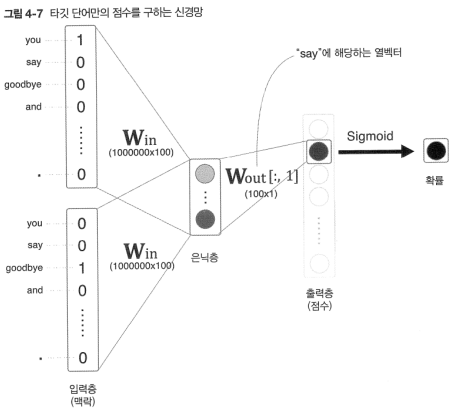
```
W = np.random.rand(7,3)
embedding = Embedding(W)
embedding.forward([0,1,2])

[Result]
array([[0.56484735, 0.59153895, 0.17156188],
       [0.04437762, 0.38333433, 0.3715608 ],
       [0.09767501, 0.90938957, 0.36817219]])

embedding.backward([0.1,0.2,0.3])

[Result]
array([[0.1, 0.1, 0.1],
       [0.2, 0.2, 0.2],
       [0.3, 0.3, 0.3],
       [0. , 0. , 0. ],
       [0. , 0. , 0. ],
       [0. , 0. , 0. ],
       [0. , 0. , 0. ]])
```

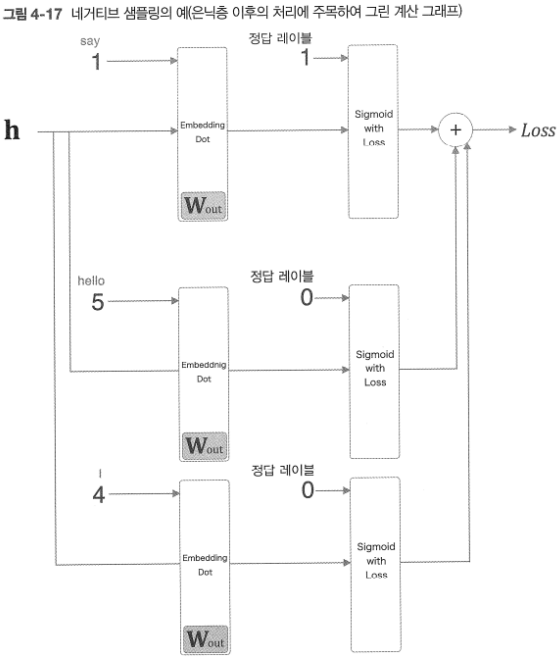
네거티브 샘플링

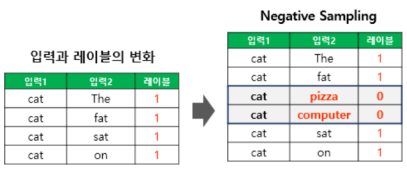
💡 **Idea**

이진 분류를 활용하여 정답에 대한 학습 뿐만 아니라 오답에 대해서도 학습하여 추론의 정확성을 높이는 방식

What 보다는 Yes/No의 질문으로 변환 ⇒ 계산량 증가를 막음







skip-gram 모델의 예시

Sigmoid 함수와 Cross Entropy loss

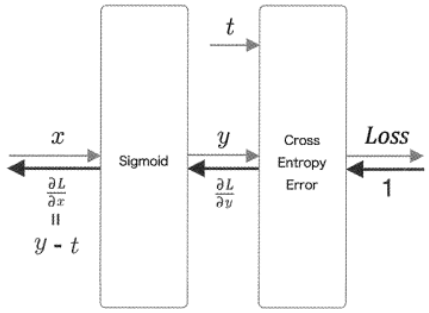
[Sigmoid]

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

[Cross Entropy Loss]

$$L = -(tlogy + (1-t)log(1-y))$$

Backpropagation

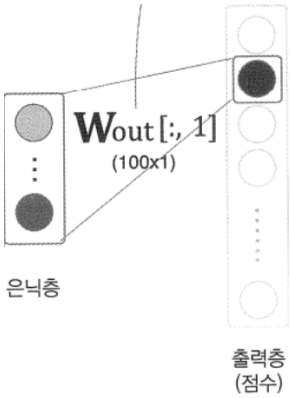


y : 확률　　t: 정답 레이블

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

이들이 최소가 되도록 최적화!

정답의 학습



입력층과 가중치가 압축되어있는 은닉층은 원래 (기존 코퍼스의 길이, 1) 크기의 W_{out}과 곱해져야 하는데, sigmoid로 변환하기 위해 정답인 id만 추출하면 됨 ⇒ Embedding과 같은 개념

= 은닉층과 W_{out}에서 필요한 id의 행만 추출하여 서로 내적

오답의 학습

- 정답만 학습하면 오답을 입력했을 때의 결과가 확실하지 않음.
- 오답 레이블 0을 주어 이를 학습하도록 함. (최대한 0에 가까워지도록 학습)
- 정답 이외에는 모두 오답으로 모든 오답 학습 불가 ⇒ 확률 분포를 활용하여 랜덤하게 오답 샘플링하여 학습 (일반적으로 5-10개)
  - 말뭉치에 고빈도로 등장하는 단어는 높은 확률로, 저빈도로 등장하는 단어는 낮은 확률로 샘플링

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

⇒ 확률이 낮은 단어의 확률을 조금 더 높여주는 효과

최종 오차 : (정답 레이블의 Loss) + (샘플링 된 오답 레이블의 Loss)

[네거티브 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)
```

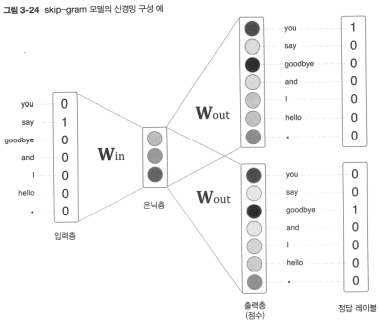
```
        # 긍정적 예 순전파
        score = self.embed_dot_layers[0].forward(h, target)
        correct_label = np.ones(batch_size, dtype=np.int32) # 정답 레이블 1
        loss = self.loss_layers[0].forward(score, correct_label)

        # 부정적 예 순전파
        negative_label = np.zeros(batch_size, dtype=np.int32) # 오답 레이블 0
        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

    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
```

5. Skip-gram 모델

ex) ____ say _____ and I say hello.

w_{t-1}과 w_{t+1} (맥락 단어들) 은 서로 조건부 독립이라고 가정할 때,

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



입력층이 한개 ⇒ 행 평균 구할 필요 없음.

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

6. word2vec의 활용

- `genism` 패키지의 `Word2Vec` 함수를 통해 쉽게 활용 가능

  `Hyperparameter`

  `size` : 임베딩 벡터 차원

  `window` : 맥락 크기

  `min_count` : 단어 최소 빈도 수 제한

  `workers` : 학습을 위한 프로세스 수

  `sg` : 0 : CBOW, 1 : skip-gram

  `hs` : 0 : negative sampling 1 : hierarchical softmax

- 주로 이미 학습되어 있는 큰 말뭉치를 활용하여 텍스트 분류, 문서 클러스트링, 품사 태그 등에 활용가능
- 머신러닝 기법 적용 가능 (ex. 스팸 메일 분류(SVM), 추천 시스템)

7. word2vec의 한계

- 맥락으로 간주되는 단어들의 순서는 무시됨.
- 장기기억에는 취약함.

> Tom was watching TV in his room. Mary came into the room. Mary said hi to _____.

위와 같은 문장에서 빈칸은 끝에 있지만, 정답은 맨 앞의 Tom.

- 단어 간 관계를 파악하는 용도로 국한됨.