Word2Vec 코드 리뷰

투빅스 김수아

Word2Vec

- 단어 간 유사도를 반영할 수 있도록 단어의 의미를 벡터화 할 수 있는 방법
- 구현 측면: 임베딩 매트릭스를 학습하는 것

- 원핫인코딩 = 희소표현
  - 고차원에 각 차원이 분리된 표현 방법

- 워드투벡터 = 분산표현 (비슷한 위치에서 등장=비슷한 의미)
  - 저차원에 **단어의 의미를 여러 차원에다가 분산**하여 표현

Word2Vec 아이디어

아이디어
비슷한 맥락을 갖는 단어에 비슷한 벡터를 주고 싶다
predictive method사용 (맥락으로 단어를 예측하거나
단어로 맥락을 예측하는 문제를 **마치 지도 학습**(supervise
d learning)처럼 푸는 것이다. 이 예측 모델을 학습하면서
단어를 어떻게 표현해야 할지를 배우게 되고,
이 과정에서 비슷한 단어가 비슷한 벡터로 표현된다.)

지도 학습을 닮았지만 사실은 비지도 학습(unsupervised learning) 알고리즘이다. 어떤 단어와 어떤 단어가 비슷한지 사람이 알려주지 않아도 word2vec은 비슷한 단어들을 찾아낼 수 있다.

용어 정리

- **Word Embedding**
  고차원의 데이터를 그보다 낮은 차원으로 변환하면서 모든 데이터간의 관계가 성립되도록 처리하는 과정
- **Word2Vec 모델**
  단어를 계산에 적용할 수 있는 숫자로 변환해서 다음에 올 단어를 예측하는 모델이다. 다음 단어를 예측하기 위해 Word Embedding을 구현해야 하는데, CBOW 또는 Skip-Gram 알고리즘을 사용한다.
- **컨텍스트(context)**
  CBOW와 Skip-Gram 모델에서 사용하는 용어로, "계산이 이루어지는 단어들", 즉 특정 단어 주변에 오는 단어들의 집합

CBOW vs. Skip-gram

- 전제: 단어의 속성은 주변 단어로부터 결정된다

- 주변 단어가 Input이고 중심 단어가 Output: CBOW
  - 맥락으로 단어를 예측
  - 데이터가 작을 경우 많이 사용

- **중심 단어가 Input이고 주변 단어가 Output**: Skip-gram
  - 단어로 맥락을 예측
  - 성능이 더 좋게 나옴

필요한 패키지 임포트

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn import init


"""
    u_embedding: Embedding for center word.
    v_embedding: Embedding for neighbor words.
"""
```

Pytorch 사용
모듈의 매개변수로 쓰이는 텐서의 일종
풀링, 정규화, 로스 등을 제공하는 함수
init은 데이터를 초기화하기 위함

```
class SkipGramModel(nn.Module):

    def __init__(self, emb_size, emb_dimension):
        super(SkipGramModel, self).__init__()
        self.emb_size = emb_size
        self.emb_dimension = emb_dimension
        self.u_embeddings = nn.Embedding(emb_size, emb_dimension, sparse=True)
        self.v_embeddings = nn.Embedding(emb_size, emb_dimension, sparse=True)


        initrange = 1.0 / self.emb_dimension
        init.uniform_(self.u_embeddings.weight.data, -initrange, initrange)
        init.constant_(self.v_embeddings.weight.data, 0)
```

스킵그램 모델 클래스
- 처리하려고 하는 현재의 단어 하나를 사용해서 주변 단어들의 발생을 유추하는 모델
- 주어진 중심단어로부터 가까이 있는 단어의 확률값 최대화하는 방향으로 함수 구성

학습 과정에서 전체가 다 업데이트 되게 하지 않음

(1) 총 단어의 개수
(2) 임베딩 시킬 벡터의 차원
(3) 중심 단어 임베딩 (Embedding 모듈은 index를 표현하는 LongTensor를 인풋으로 기대하고 해당 벡터로 인덱싱, 명시적으로 원핫 벡터로 바꾸지 않아도 됨) = lookup 테이블 만들기
(4) 주변 단어 임베딩
(5) init.uniform_은 연속균등분포를 사용하여 초기화한다는 것 (-initrange부터 initrange까지)
    - 텐서플로우의 random_uniform, u_embedding 초기화
(6) init.constant_도 연속균등분포를 사용하는데 파라미터가 하나로 주어짐 (0으로 채운 텐서 리턴)
    - v_embedding 초기화

```
def forward(self, pos_u, pos_v, neg_v):
    emb_u = self.u_embeddings(pos_u)
    emb_v = self.v_embeddings(pos_v)
    emb_neg_v = self.v_embeddings(neg_v)

    score = torch.sum(torch.mul(emb_u, emb_v), dim=1)
    score = torch.clamp(score, max=10, min=-10)
    score = -F.logsigmoid(score)

    neg_score = torch.bmm(emb_neg_v, emb_u.unsqueeze(2)).squeeze()
    neg_score = torch.clamp(neg_score, max=10, min=-10)
    neg_score = -torch.sum(F.logsigmoid(-neg_score), dim=1)

    return torch.mean(score + neg_score)
```

임베딩된 벡터는 대개 [0.91, 0.15, -0.05] 과 같은 모양으로, 각각의 차원이 어떤 의미인지는 알 수 없고, forward 과정을 통해 학습해 나갈 수만 있음.

pos_u: [batch_size]
pos_v: [batch_size]
neg_v: [batch_size, neg_sampling_count]

neg_v만 모양이 다른 이유는 각 pos_u에 대해 negative pair를 만들기 위한 다수의 neg_v를 둘 것이기 때문

각 pos_u에 대해 neg_sampling_count 번만큼 카피해서 결과적으로는 self.u_embeddings(Variable(torch.LongTensor(pos_u))) 이걸 여러 번 반복하게 되어 비효율적임. torch.bmm은 일괄행렬곱. lookup과 mul 연산을 대체함. (학습 과정을 3배 빠르게 해줌)

1.Corpus에서 단어 집합(Vocabulary)을 구하고 Index를 매긴다.
2.Window size를 정하고 T개의 데이터셋를 준비한다.
3.Center word와 Context word를 표현할 2개의 Embedding Matrix를 선언한다.
4.P(o|c)를 구해서 Negative log-likelihood(loss)를 구한다.
5.Gradient Descent를 사용하여 loss를 최소화한다.
6.학습이 끝난 뒤에는 Center vector와 Context vector를 평균해서 사용한다.

```python
def forward(self, pos_u, pos_v, neg_v):
    emb_u = self.u_embeddings(pos_u)
    emb_v = self.v_embeddings(pos_v)
    emb_neg_v = self.v_embeddings(neg_v)

    score = torch.sum(torch.mul(emb_u, emb_v), dim=1)
    score = torch.clamp(score, max=10, min=-10)
    score = -F.logsigmoid(score)

    neg_score = torch.bmm(emb_neg_v, emb_u.unsqueeze(2)).squeeze()
    neg_score = torch.clamp(neg_score, max=10, min=-10)
    neg_score = -torch.sum(F.logsigmoid(-neg_score), dim=1)

    return torch.mean(score + neg_score)
```

score를 구할 때
score = torch.dot(emb_u, emb_v)
이렇게 할 수 있지만 mul->sum하는 이유는 **파이토치의 작업이 batch를 기반**으로 하기 때문이고, 이는 **GPU로 코드를 가속화하는 데에 도움을 줌**

1.Corpus에서 단어 집합(Vocabulary)을 구해서 Index를 매긴다.
2.Window size를 정하고 T개의 데이터셋를 준비한다.
3.Center word와 Context word를 표현할 2개의 Embedding Matrix를 선언한다.
4.P(context|center) 구해서 Negative log-likelihood를 구한다.
5.Gradient Descent를 사용하여 loss를 최소화한다.
6.학습이 끝난 뒤에는 Center vector와 Context vector를 평균해서 사용한다.

`torch.clamp(input, min, max, out=None)` → Tensor

Clamp all elements in `input` into the range [`min`, `max`] and return a resulting tensor:

$$y_i = \begin{cases} \min & \text{if } x_i < \min \\ x_i & \text{if } \min \leq x_i \leq \max \\ \max & \text{if } x_i > \max \end{cases}$$

If `input` is of type *FloatTensor* or *DoubleTensor*, args `min` and `max` must be real numbers, otherwise they should be integers.

Parameters

- **input** (*Tensor*) – the input tensor.
- **min** (*Number*) – lower-bound of the range to be clamped to
- **max** (*Number*) – upper-bound of the range to be clamped to
- **out** (*Tensor, optional*) – the output tensor.

negative sampling

Word2Vec은 출력층이 내놓는 스코어값에 소프트맥스 함수를 적용해 확률값으로 변환한 후 이를 정답과 비교해 **역전파(backpropagation)**하는 구조!

그런데 소프트맥스를 적용하려면 분모에 해당하는 값, 즉 중심단어와 나머지 모든 단어의 내적을 한 뒤, 이를 다시 exp를 취해줘야 합니다. 보통 전체 단어가 10만개 안팎으로 주어지니까 **계산량이 어마어마해지죠**.

Word2Vec의 마지막 단계를 주목해봅시다. 출력층에 있는 소프트맥스 함수는 단어 집합 크기의 벡터 내의 모든 값을 0과 1사이의 값이면서 모두 더하면 1이 되도록 바꾸는 작업을 수행합니다. 그리고 이에 대한 오차를 구하고 모든 단어에 대한 임베딩을 조정합니다. 그 단어가 중심 단어나 주변 단어와 전혀 상관없는 단어라도 마찬가지 입니다. 그런데 만약 단어 집합의 크기가 수백만에 달한다면 이 작업은 굉장히 무거운 작업입니다.

연관 관계가 없는 수많은 단어의 임베딩을 조정할 필요는 없습니다. 전체 단어 집합이 아니라 일부 단어 집합에 대해서만 고려하면 안 될까요? 즉, Word2Vec은 주변 단어들을 긍정(positive)으로 두고 랜덤으로 샘플링 된 단어들을 부정(negative)으로 둔 다음에 이진 분류 문제를 수행합니다.이는 기존의 다중 클래스 분류 문제를 이진 분류 문제로 바꾸면서도 연산량에 있어서 훨씬 효율적입니다.

negative sampling (cont.)

연산량이 어마어마하기 때문에 **소프트맥스 확률을 구할 때** 전체 단어를 대상으로 구하지 않고, **일부 단어만 뽑아서 계산을 하게** 되는데요. 이것이 바로 negative sampling입니다. negative sampling은 학습 자체를 아예 스킵하는 subsampling이랑은 다르다는 점에 유의하셔야 합니다.

negative sampling의 절차는 이렇습니다. 사용자가 지정한 윈도우 사이즈 내에 등장하지 않는 단어(negative sample)를 5~20개 정도 뽑습니다. **이를 정답단어와 합쳐 전체 단어처럼 소프트맥스 확률을 구하는 것입니다**. 바꿔 말하면 윈도우 사이즈가 5일 경우 최대 25개 단어를 대상으로만 소프트맥스 확률을 계산하고, 파라메터 업데이트도 25개 대상으로만 이뤄진다는 이야기입니다.

```python
def forward(self, pos_u, pos_v, neg_v):
    emb_u = self.u_embeddings(pos_u)
    emb_v = self.v_embeddings(pos_v)
    emb_neg_v = self.v_embeddings(neg_v)

    score = torch.sum(torch.mul(emb_u, emb_v), dim=1)
    score = torch.clamp(score, max=10, min=-10)
    score = -F.logsigmoid(score)

    neg_score = torch.bmm(emb_neg_v, emb_u.unsqueeze(2)).squeeze()
    neg_score = torch.clamp(neg_score, max=10, min=-10)
    neg_score = -torch.sum(F.logsigmoid(-neg_score), dim=1)

    return torch.mean(score + neg_score)
```

unsqueeze()와 squeeze() 모두 차원 변환용
- un~: 인수로 받은 위치에 새로운 차원 삽입
- sq~: 차원의 원소가 1인 차원 없앰

negative sampling을 위한 부분

- 확률값은 고정 (처음에 정해야)

```
def save_embedding(self, id2word, file_name):
    embedding = self.u_embeddings.weight.cpu().data.numpy()
    with open(file_name, 'w') as f:
        f.write('%d %d\n' % (len(id2word), self.emb_dimension))
        for wid, w in id2word.items():
            e = ' '.join(map(lambda x: str(x), embedding[wid]))
            f.write('%s %s\n' % (w, e))
```

임베딩 벡터를 저장하는 함수
u_embedding의 weight를 numpy 객체로 만든다.

역전파 함수는 따로 만들지 않아도 파이토치가 해준다.

참고 자료들

- https://wikidocs.net/22660
- https://pythonkim.tistory.com/93?category=613486
- https://dreamgonfly.github.io/machine/learning,/natural/language/processing/2017/08/16/word2vec_explained.html
- https://shuuki4.wordpress.com/2016/01/27/word2vec-%EA%B4%80%EB%A0%A8-%EC%9D%B4%EB%A1%A0-%EC%A0%95%EB%A6%AC/
- http://dsp.yonsei.ac.kr/139926
- https://adoni.github.io/2017/11/08/word2vec-pytorch/
- https://ratsgo.github.io/from%20frequency%20to%20semantics/2017/03/30/word2vec/