词嵌入进阶

在“Word2Vec的实现” ()一节中，我们在小规模数据集上训练了一个 Word2Vec 词嵌入模型，并通过词向量的余弦相似度搜索近义词。虽然 Word2Vec 已经能够成功地将离散的单词转换为连续的词向量，并能一定程度上地保存词与词之间的近似关系，但 Word2Vec 模型仍不是完美的，它还可以被进一步地改进：

1. 子词嵌入（subword embedding）：FastText (https://zh.d2l.ai/chapter_natural-language-processing/fasttext.html) 以固定大小的 n-gram 形式将单词更细致地表示为了子词的集合，而 BPE (byte pair encoding) (https://d2l.ai/chapter_natural-language-processing/subword-embedding.html#byte-pair-encoding) 算法则能根据语料库的统计信息，自动且动态地生成高频子词的集合；
2. GloVe 全局向量的词嵌入 (https://zh.d2l.ai/chapter_natural-language-processing/glove.html): 通过等价转换 Word2Vec 模型的条件概率公式，我们可以得到一个全局的损失函数表达，并在此基础上进一步优化模型。

实际中，我们常常在大规模的语料上训练这些词嵌入模型，并将预训练得到的词向量应用到下游的自然语言处理任务中。本节就将以 GloVe 模型为例，演示如何用预训练好的词向量来求近义词和类比词。

GloVe 全局向量的词嵌入

GloVe 模型

先简单回顾以下 Word2Vec 的损失函数（以 Skip-Gram 模型为例，不考虑负采样近似）：

$$-\sum_{t=1}^{T}\sum_{-m\le j\le m, j\neq 0}\log P(w^{(t+j)}\mid w^{(t)})$$

其中

$$P(w_j\mid w_i)=\frac{\exp(\boldsymbol{u}_j^\top\boldsymbol{v}_i)}{\sum_{k\in\mathcal{V}}\exp(\boldsymbol{u}_k^\top\boldsymbol{v}_i)}$$

是 $\boldsymbol{w_i}$ 为中心词，$\boldsymbol{w_j}$ 为背景词时 Skip-Gram 模型所假设的条件概率计算公式，我们将其简写为 $\boldsymbol{q_{ij}}$。

注意到此时我们的损失函数中包含两个求和符号，它们分别枚举了语料库中的每个中心词和其对应的每个背景词。实际上我们还可以采用另一种计数方式，那就是直接枚举每个词分别作为中心词和背景词的情况：

$$-\sum_{i\in\mathcal{V}}\sum_{j\in\mathcal{V}}x_{ij}\log q_{ij}$$

其中 $\boldsymbol{x_{ij}}$ 表示整个数据集中 $\boldsymbol{w_j}$ 作为 $\boldsymbol{w_i}$ 的背景词的次数总和。

我们还可以将该式进一步地改写为交叉熵 (cross-entropy) 的形式如下：

$$-\sum_{i\in\mathcal{V}}x_i\sum_{j\in\mathcal{V}}p_{ij}\log q_{ij}$$

其中 $\boldsymbol{x_i}$ 是 $\boldsymbol{w_i}$ 的背景词窗大小总和，$\boldsymbol{p_{ij}}=\boldsymbol{x_{ij}}/\boldsymbol{x_i}$ 是 $\boldsymbol{w_j}$ 在 $\boldsymbol{w_i}$ 的背景词窗中所占的比例。

从这里可以看出，我们的词嵌入方法实际上就是想让模型学出 $\boldsymbol{w_j}$ 有多大概率是 $\boldsymbol{w_i}$ 的背景词，而真实的标签则是语料库上的统计数据。同时，语料库中的每个词根据 $\boldsymbol{x_i}$ 的不同，在损失函数中所占的比重也不同。

注意到目前为止，我们只是改写了 Skip-Gram 模型损失函数的表面形式，还没有对模型做任何实质上的改动。而在 Word2Vec 之后提出的 GloVe 模型，则是在之前的基础上做出了以下几点改动：

1. 使用非概率分布的变量 $p'_{ij} = x_{ij}$ 和 $q'_{ij} = \exp(\boldsymbol{u}_j^\top \boldsymbol{v}_i)$，并对它们取对数；
2. 为每个词 w_i 增加两个标量模型参数：中心词偏差项 b_i 和背景词偏差项 c_i，松弛了概率定义中的规范性；
3. 将每个损失项的权重 x_i 替换成函数 $h(x_{ij})$，权重函数 $h(x)$ 是值域在 $[0,1]$ 上的单调递增函数，松弛了中心词重要性与 x_i 线性相关的隐含假设；
4. 用平方损失函数替代了交叉熵损失函数。

综上，我们获得了 GloVe 模型的损失函数表达式：

$$\sum_{i\in\mathcal{V}}\sum_{j\in\mathcal{V}} h(x_{ij})(\boldsymbol{u}_j^\top \boldsymbol{v}_i + b_i + c_j - \log x_{ij})^2$$

由于这些非零 x_{ij} 是预先基于整个数据集计算得到的，包含了数据集的全局统计信息，因此 GloVe 模型的命名取“全局向量”（Global Vectors）之意。

载入预训练的 GloVe 向量

GloVe 官方 (https://nlp.stanford.edu/projects/glove/) 提供了多种规格的预训练词向量，语料库分别采用了维基百科、CommonCrawl和推特等，语料库中词语总数也涵盖了从60亿到8,400亿的不同规模，同时还提供了多种词向量维度供下游模型使用。

`torchtext.vocab` (https://torchtext.readthedocs.io/en/latest/vocab.html) 中已经支持了 GloVe, FastText, CharNGram 等常用的预训练词向量，我们可以通过声明 `torchtext.vocab.GloVe` (https://torchtext.readthedocs.io/en/latest/vocab.html#glove) 类的实例来加载预训练好的 GloVe 词向量。

In [2]:

```
import torch
import torchtext.vocab as vocab

print([key for key in vocab.pretrained_aliases.keys() if "glove" in key])
cache_dir = "/home/kesci/input/GloVe6B5429"
glove = vocab.GloVe(name='6B', dim=50, cache=cache_dir)
print("一共包含%d个词。" % len(glove.stoi))
print(glove.stoi['beautiful'], glove.itos[3366])
```

```
['glove.42B.300d', 'glove.840B.300d', 'glove.twitter.27B.25d', 'glove.twitter.27B.50
一共包含400000个词。
3366 beautiful
```

求近义词和类比词

求近义词

由于词向量空间中的余弦相似性可以衡量词语含义的相似性（为什么？），我们可以通过寻找空间中的 k 近邻，来查询单词的近义词。

In [3]:

```
def knn(W, x, k):
    '''
    @params:
        W: 所有向量的集合
        x: 给定向量
        k: 查询的数量
    @outputs:
        topk: 余弦相似性最大k个的下标
        [...]: 余弦相似度
    '''
    cos = torch.matmul(W, x.view((-1,))) / (
        (torch.sum(W * W, dim=1) + 1e-9).sqrt() * torch.sum(x * x).sqrt())
    _, topk = torch.topk(cos, k=k)
    topk = topk.cpu().numpy()
    return topk, [cos[i].item() for i in topk]

def get_similar_tokens(query_token, k, embed):
    '''
    @params:
        query_token: 给定的单词
        k: 所需近义词的个数
        embed: 预训练词向量
    '''
    topk, cos = knn(embed.vectors,
                    embed.vectors[embed.stoi[query_token]], k+1)
    for i, c in zip(topk[1:], cos[1:]):  # 除去输入词
        print('cosine sim=%.3f: %s' % (c, (embed.itos[i])))
```

除去和自己本身的匹配

```
get_similar_tokens('chip', 3, glove)
```

```
cosine sim=0.856: chips
cosine sim=0.749: intel
cosine sim=0.749: electronics
```

```
100%|| 398393/400000 [00:30<00:00, 38997.22it/s]
```

In [4]:

```
get_similar_tokens('baby', 3, glove)
```

```
cosine sim=0.839: babies
cosine sim=0.800: boy
cosine sim=0.792: girl
```

In [5]:

```
get_similar_tokens('beautiful', 3, glove)
```

```
cosine sim=0.921: lovely
cosine sim=0.893: gorgeous
cosine sim=0.830: wonderful
```

求类比词

除了求近义词以外，我们还可以使用预训练词向量求词与词之间的类比关系，例如“man”之于“woman”相当于“son”之于“daughter”。求类比词问题可以定义为：对于类比关系中的4个词“a 之于 b 相当于 c 之于 d”，给定前3个词 a, b, c 求 d。求类比词的思路是，搜索与 $\mathbf{vec}(c) + (\mathbf{vec}(b) - \mathbf{vec}(a))$ 的结果向量最相似的词向量，其中 $\mathbf{vec}(w)$ 为 w 的词向量。

向量加减(C)

In [6]:

```
def get_analogy(token_a, token_b, token_c, embed):
    '''
    @params:
        token_a: 词a
        token_b: 词b
        token_c: 词c
        embed: 预训练词向量
    @outputs:
        res: 类比词d
    '''
    vecs = [embed.vectors[embed.stoi[t]]
                for t in [token_a, token_b, token_c]]
    x = vecs[1] - vecs[0] + vecs[2]
    topk, cos = knn(embed.vectors, x, 1)
    res = embed.itos[topk[0]]
    return res
```

```
get_analogy('man', 'woman', 'son', glove)
```

Out[6]:

```
'daughter'
```

In [7]:

```
get_analogy('beijing', 'china', 'tokyo', glove)
```

Out[7]:

```
'japan'
```

In [8]:

```
get_analogy('bad', 'worst', 'big', glove)
```

Out[8]:

```
'biggest'
```

In [9]:

```
get_analogy('do', 'did', 'go', glove)
```

Out[9]:

```
'went'
```

In [ ]: