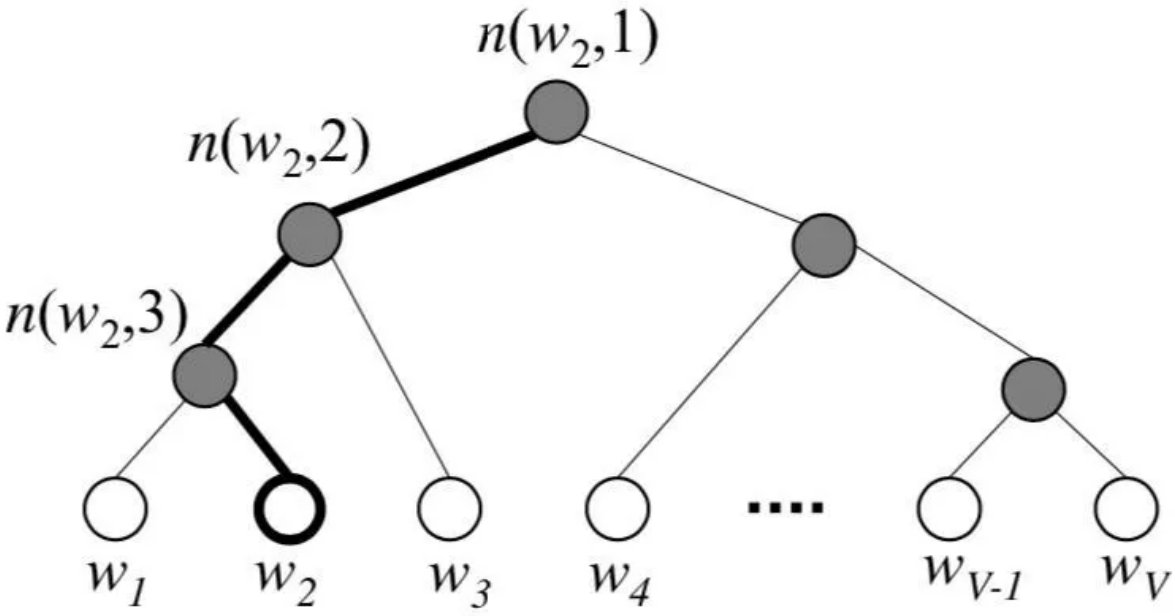
推荐系统从零单排系列(六)—Word2Vec优化策略Hierarchical Softmax与Negative Sampling

原创 可爱又迷人的反派角色宁宁 机器学习荐货情报局 2019-05-20

【导读】前两篇文章中介绍了两种基本的Word2Vec模型：CBOW，Skip-gram。今天将详细介绍使得Word2Vec落地的两种优化策略：层次Softmax与负采样策略。最后在文末将给出Word2Vec实践代码，以及小编整理的一些资料方便同学深入研究。

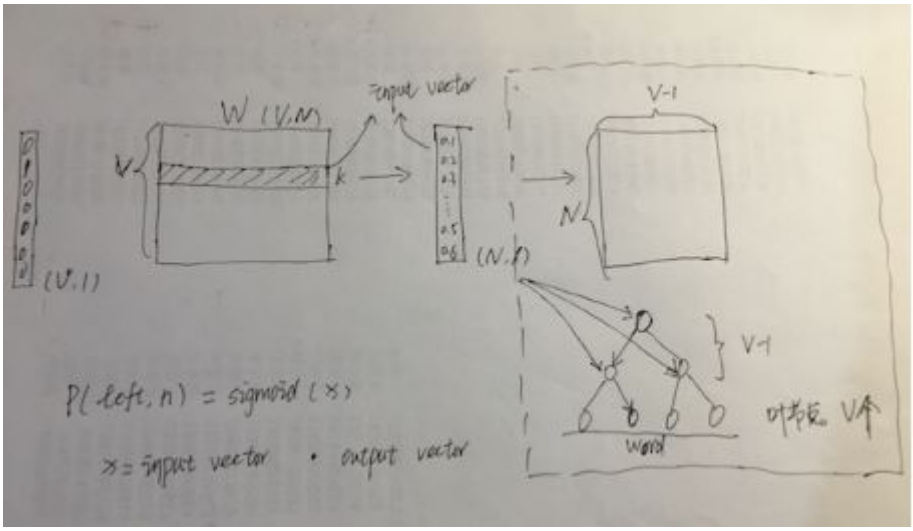
Hierarchical Softmax

原始的Word2Vec使用softmax得到最种的词汇概率分布，词汇表往往包含上百万个单词，如果针对输出中每一个单词都要用softmax计算概率的话，计算量是非常大的。解决办法之一就是Hierarchical Softmax。相比于原始的Softmax直接计算每个单词的概率，Hierarchical Softmax使用一颗二叉树来得到每个单词的概率。被验证的效果最好的二叉树类型就是霍夫曼树：



霍夫曼树中有V-1个中间节点，V个叶节点。叶节点与单词表中V个单词一一对应。首先根据单词出现的频率构造一颗霍夫曼树，出现频率高的单词霍夫曼编码就短，更加靠近根节点。
原来的Word2Vec模型结构会被改变，隐藏层后直接和霍夫曼树中每一个非叶节点相连，如下图所示（相当于输出层中只有V-1个神经元节点）。然后再每一个非叶节点上计算二分概率（也就是用Sigmoid函数进行激活），这个概率是指从当前节点随机游走的概率，可以任意指定是向左

游走的概率，还是向右游走的概率。从根节点到目标单词的路径是唯一的，将中间非叶节点的游走概率相乘就得到了最终目标单词的概率。



这样只用计算树深度个输出节点的概率就可以得到目标单词的概率。霍夫曼树的深度基本是logV，所以此时的计算复杂度就降为了O(logV)。另外，高频词非常接近树根，其所需要的计算次数将进一步减少，这也是使用霍夫曼树的一个优点。此时的目标函数为：

$$p(w = w_O) = \prod_{j=1}^{L(w)-1} \sigma\left(\left[\!\left[n(w,j+1) = \mathrm{ch}(n(w,j))\right]\!\right] \cdot \mathbf{v}'_{n(w,j)}{}^{T}\mathbf{h}\right)$$

L(w)是树深度；n(w,j)表示从根节点到目标单词w的路径上第j个节点；ch(n)表示节点n的孩子节点。中间的尖括号表示是否成立的判断，结果无非是+1，或-1。注意sigmoid的特性：
Sigmoid(-x) = 1 - Sigmoid(x)
Vwi表示输入单词的input vector，Vn’表示霍夫曼树中间节点的output vector。

Negative Sampling

Negative Sampling（简称NEG）是NCE（Noise Contrastive Estimation）的简化版本，目的是提高训练速度并改善所得词向量的质量。与Hierarchical Softmax相比，NEG不再采用复杂的霍夫曼树，而是利用相对简单的随机负采样，能大幅提升性能，因而可以作为Hierarchical Softmax的一种替代。

针对一个样本(WI, W)，Negative Sampling的目标函数如下所示：

$$\log \sigma({v'_{w_O}}^\top v_{w_I}) + \sum_{i=1}^{k} \mathbb{E}_{w_i \sim P_n(w)} \left[\log \sigma(-{v'_{w_i}}^\top v_{w_I}) \right]$$

现在的目标就是利用logistic regression，从带有噪声（负样本）的目标中找到正样本（Wo），其中K是负样本采样的数量。作者指出在小训练集上，k取5-20比较合适；大训练集上k取2-5即可。和SGD的思想非常像，不再是利用所有的负样本进行参数的更新，而是只利用负采样出来的K个来进行loss的计算，参数的更新。只不过SGD每次只用一个样本，而不是K个。

负样本采样服从分布Pn(W)，经过试验发现unigram分布效果最好，如下：

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=0}^{n}(f(w_j)^{3/4})}$$

其中f(wi)表示单词wi在语料中出现的频率。3/4这个值是通过实验发现的经验值。

Subsampling of Frequent Wrods

大语料集中，像 the a in 之类的单词出现频率非常高几乎是很多单词的上下文，造成其携带的信息非常少。对这些单词进行下采样不仅可以加快训练速度还可以提高低频词训练词向量的质量。

为了平衡高频与低频词，对训练集中的单词按照下述公式决定是否保留该单词：

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$$

其中f(wi)是单词wi出现的频率，t凭经验值取10的-5次方。该公式保证出现频率超过t的单词将被下采样，并且不会影响原有的单词的频率相对大小（rank）。先生成（target， context），比如 （love，[I,China]） ，然后依次遍历他们，小于P(wi)的训练样本将被从训练样本中去掉。

词组Phrases

词组并不能简单的将单词分开解释，而是应该看做一个整体，例如“New York”。把这样的词组看成是一个整体，用特殊的符号代替不会过多的增加词汇表的大小，但效果是很显著的。

实践

又到了快乐的code时间，完整代码见github：
https://github.com/gutouyu/ML_CIA/blob/master/Embedding/Word2Vec.py

欢迎star~

使用的是 Pytorch 最新的稳定版本，关键代码如下：

计算word频率，用于下采样

```
word_frequency = np.array(list(word_count.values()))
word_frequency = word_frequency / word_frequency.sum()
word_sample = 1 - np.sqrt(T / word_frequency)
word_sample = np.clip(word_sample, 0, 1)
word_sample = {wc[0]: s for wc, s in zip(word_count.items(), word_sample)}
```

生成训练样本，并完成下采样：

```
class PermutedSubsampleedCorpus(Dataset):

    def __init__(self, data, word_sample):
        self.data = []
        for iword, owords in data:
            if np.random.rand() > word_sample[iword]:
                self.data.append((iword, owords))

    def __len__(self):
        return len(self.data)

    def __getitem__(self, item):
        iword, owords = self.data[item]
        # 按列拼接形成batch的
        return (iword, owords)

data = []
for target_pos in range(CONTEXT_SIZE, len(raw_data) - CONTEXT_SIZE):
    context = []
    for w in range(-CONTEXT_SIZE, CONTEXT_SIZE + 1):
        if w == 0:
            continue
        context.append(raw_data[target_pos + w])
    data.append((raw_data[target_pos], context))

dataset = PermutedSubsampleedCorpus(data, word_sample)
```

Skip-gram with negative sampling模型代码，时刻关注变量的维度变化：

```
class SkipGramNegativeSample(nn.Module):

    def __init__(self, vocab_size, embedding_size, n_negs):
        super(SkipGramNegativeSample, self).__init__()
        self.ivectors = nn.Embedding(vocab_size, embedding_size)
        self.ovectors = nn.Embedding(vocab_size, embedding_size)

        self.ivectors.weight.data.uniform_(- 0.5/embedding_size, 0.5/embedding_size)
        self.ovectors.weight.data.zero_()

        self.n_negs = n_negs
        self.vocab_size = vocab_size

    def forward(self, iwords, owords):
        # iwords: (batch_size)
        # owords: (batch_size, context_size * 2)
        batch_size = iwords.size()[0]
        context_size = owords.size()[-1] # 两边的context之和

        nwords = torch.FloatTensor(batch_size, context_size * self.n_negs).uniform_(0,

        ivectors = self.ivectors(iwords).unsqueeze(2) # (batch_size, embeding_dim, 1)
        ovectors = self.ovectors(owords) # (batch_size, context_size, embedding_dim)
        nvectors = self.ovectors(nwords).neg() #(batch_size, context_size * n_negs, emb

        oloss = torch.bmm(ovectors, ivectors).squeeze().sigmoid().log().mean() #(batch_
        nloss = torch.bmm(nvectors, ivectors).squeeze().sigmoid().log().view(-1, contex
        return -(oloss + nloss).mean()
```

训练模型:

```
dataloader = DataLoader(dataset, batch_size=5, shuffle=False, num_workers=1)

model = SkipGramNegativeSample(vocab_size, EMBEDDING_DIM, n_negs=5)
optimizer = optim.SGD(model.parameters(), lr=0.1)

def make_context_vectors(context, word_to_ix):
    context_ixs = [word_to_ix[w] for w in context]
    return torch.tensor(context_ixs, dtype=torch.long)

losses = []
for epoch in range(10):
    total_loss = 0
    for batch_size, (iword, owords) in enumerate(dataloader):

        iword = list(map(lambda x: word_to_ix[x], iword))
        iword = torch.tensor(iword, dtype=torch.long)

        owords = list(map(list, owords))
        owords = np.array(owords).T

        myfunc = np.vectorize(lambda x: word_to_ix[x])
        owords = list(map(myfunc, owords))
        owords = torch.tensor(owords, dtype=torch.long)
```

```
            model.zero_grad()
            loss = model(iword, owords)
            loss.backward()
            optimizer.step()

            total_loss += loss
        losses.append(total_loss)
    print(losses)
```

这份代码只是用来学习的，可以理解Skip-gram以及negative sampling、下采样等知识点，并不能用于实际生产中，望知悉。在代码中，要时刻关注每个向量的维度变化，关注训练集的生成，以及loss的计算方式。

自从用了Pytorch，腰也不疼了，皮肤也光滑了。 Hello torch， bye-bye tensorflow. 真香~

Word2Vec资料

如果你对Word2Vec感兴趣，下面这些资料也许对你会有帮助：

论文

1. Distributed Representations of Words and Phrases and their Compositionality (Google 2013)
2. Efficient Estimation of Word Representations in Vector Space (Google 2013)
3. Word2vec Parameter Learning Explained (UMich 2016)

Code

1. Word2Vec源码，注释版本[https://github.com/chrisjmccormick/word2vec_commented]
2. tensorflow实现 [https://github.com/zyxue/stanford-cs20si-tensorflow-for-deep-learning-research/blob/master/assignments/01/q3_04_word2vec_visualize.py]

博客

1. **CBOW与Skip-gram比较** https://zhuanlan.zhihu.com/p/37477611

2. **Hierarchical softmax and negative sampling: short notes worth telling**
https://towardsdatascience.com/hierarchical-softmax-and-negative-sampling-short-notes-worth-telling-2672010dbe08
3. [**Word embeddings: exploration, explanation, and exploitation (with code in Python)**]
https://towardsdatascience.com/word-embeddings-exploration-explanation-and-exploitation-with-code-in-python-5dac99d5d795
4. [**Word2Vec Tutorial - The Skip-Gram Model**]
http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/
5. [**Word2Vec Tutorial Part 2 - Negative Sampling**]
http://mccormickml.com/2017/01/11/word2vec-tutorial-part-2-negative-sampling/
6. [**秒懂词向量Word2Vec的本质**]
https://zhuanlan.zhihu.com/p/26306795
7. [**A Beginner's Guide to Word2Vec and Neural Word Embeddings**]
https://skymind.ai/wiki/word2vec

课程

1. [**CS224n: NLP with Deep Learning**]
http://web.stanford.edu/class/cs224n/

推荐系统从零单排系列(一)--Deep Neural Network for YouTube Recommendations
推荐系统从零单排系列(二)--Item-Based协同过滤算法
推荐系统从零单排系列(三)--再谈亚马逊Item-based推荐系统
推荐系统从零单排系列(四)—Word2Vec理论与实践(上)
推荐系统从零单排系列(五)—Word2Vec理论与实践(下)