tf.nn.nce_loss

在Tensorflow的Word2Vec的文档里，描述了如何用Tensorflow把语料库训练成词向量，这里用到了nce_loss函数。因为这是词向量的入门文档，假设读者还不是很了解词向量的训练机制，所以先用一个最简单的数据集 MNIST Dataset(手写数字数据集)作为例子来解释nce_loss究竟是什么函数。

我们知道，手写数字数据集是把 24 x 24 大小的图片分成10类分别是 1,2,3,4,5,6,7,8,9,0。现在假设这个数据集比较坑，图片比较模糊，标签可能出现矛盾的情况，比如对于某张图片，有时候被认为是"6“，有时候又被认为是"4“，有时候被认为是"0“。当然你可以说，这样我们可以把标签从原来的[0, 0, ... , 1, ..., 0] 这种形式换成[p0, p1, p2, ...,p9]这种形式，分别对应标签为“0”，“1”…到“9”的频率，然后求 min_square_loss或其他损失函数。

但是在这里我们假设类别的数量是十分巨大的，可能有几千类，比如除了是0到9的数字，还可以是数学符号，英文字母甚至汉字。这样的话，求损失函数就十分困难了，需要巨大的花销。

因此最好我们只需要关注当前的某个样本对应的标签，而不考虑可能的别的标签，比如对于一张图片，当前的标签是“6”，就只考虑模型预测出的那个向量在“6”处输出的值。比如模型输出的向量是[1,1,1,2,1,1,3,0.5,1,1,1.5]，我们只希望对表示“6”的概率的那个维度（这里就是第7维对应数值是3）进行梯度下降，希望让这个维度上的预测值变大一些。

但是这样很容易出现问题，因为模型输出的向量不是归一化的概率向量，也就是说$p_0+p_1+p_2+...+p_9 \neq 1$，如此训练，很可能导致模型输出向量的每一维度都变得越来越大。如果要进行归一化操作，则我们进行梯度下降的时候计算的便是$\frac{p_i}{p_0+p_1+p_2+...+p_9}$，又必须把其他所有维度的梯度都计算一遍，就没有减少开销的作用了。

但是论文《Noise-contrastive estimation: A new estimation principle for unnormalized statistical models》提出了一种方法可以避免这种情况的出现，通过人工添加一些“负样本”，可以防止模型最后输出的向量在各个维度都发散。

论文浅读

假设对于某张MNIST数据集中的图片，该图片的标签为1的概率是0.1，标签为2的概率是0.2， 标签为7的概率是0.7。因此我们希望训练好的模型对于这张图片能够产生如下的输出：[0, 0.1, 0.2, 0, 0, 0, 0, 0.7, 0, 0]。

但是因为实际的标签并不是概率，而是one-hot的编码，我们把这张图片对应的实际的标签记作 $\vec{x}$ ，则其概率密度函数为

$$p(x;\theta)=\begin{cases}0.1 & \vec{x}=[1,0,0,0,0,0,0,0,0,0,0] \\ 0.2 & \vec{x}=[0,1,0,0,0,0,0,0,0,0,0] \\ 0.3 & \vec{x}=[0,0,0,0,0,0,0,0,1,0,0] \\ 0 & else\end{cases}$$

接下来的问题是，给定大量的样本$\{\vec{x}\}$, 并且假设$p(\vec{x}_j;\theta)=f(x_j;\theta)/\sum_i f(\vec{x}_i;\theta)$, 找到最佳的$\theta$， 使其满足最大似然估计，也就是是让 $\prod_i p(\vec{x}_i;\theta)$ 最大。很显然在计算中我们总是需要计算 $\sum_i f(\vec{x}_i;\theta)$ ，这也正是上文所述的损耗性能的地方。能不能不计算呢?

论文提出的方法是在样本空间$\{\vec{x}\}$中加入一些随机生成的假样本， 我们令

$$C_t=\begin{cases}1 & \vec{x}_t\text{是正样本} \\ 0 & \vec{x}_t\text{是负样本}\end{cases}$$

同时我们假设真样本和假样本的比例是1:1，因此有$P(C=1)=P(C=0)=\frac{1}{2}$，于是有：

$$p(C=1|\vec{x};\theta)=\frac{p(C=1,\vec{x};\theta)}{p(\vec{x};\theta)}=\frac{p(C=1,\vec{x};\theta)}{p(C=1;\theta)\cdot P(\vec{x}|C=1;\theta)+p(0;\theta)\cdot P(\vec{x}|C=0;\theta)}$$
$$=\frac{p_m(\vec{x};\theta)}{p_m(\vec{x};\theta)+p_n(\vec{x})}$$

其中 $p_m(\vec{x};\theta)$ 表示在参数 θ 下的样本的概率密度函数，$p_n(\vec{x})$ 表示假样本概率密度函数。

把 $p(C=1|\vec{x};\theta)$ 记作 $h(\vec{x};\theta)$,于是 $p(C=0|\vec{x};\theta)=1-h(\vec{x};\theta)$

此时我们依然采用最大似然估计，考察在参数 θ 下样本集合的“真假标签”的概率分布。我们希望参数 θ 能够最大化**对于指定的样本空间$\{\vec{x}\}$ 标签序列 $\{C\}$ 出现的概率**（而不是对于指定的标签样本集合 $\{\vec{x}\}$ 出现的概率）。对于样本空间 $\{\vec{x}\}$，参数 θ 的似然函数为：

$$\prod_i p(C=1|\vec{x};\theta)^{C_t}p(C=0|\vec{x};\theta)^{(1-C_t)}$$

对其求对数，变为

$$l(\theta)=\sum_i[C_i\ln(h(\vec{x}_i;\theta))+(1-C_i)ln(1-h(\vec{x}_i;\theta))]$$

令 $f(\vec{x};\theta)=\ln(p_n(\vec{x};\theta))$, 则 $h(\vec{x};\theta)=\frac{e^f}{e^f+p_n}$,

$$\ln(h(\vec{x};\theta))=\ln[sigmoid(f(\vec{x};\theta)-\ln(p_n(\vec{x}))] \quad (sigmoid(x)=\frac{1}{1+e^{-x}})$$

于是有 $l(\theta)=\sum_i(C_i\ln[sigmoid(f-p_n)]+(1-C_i)\ln[1-sigmoid(f-p_n)])$

论文告诉我们，当 $l(\theta)$ 得到最大值的时候，必然可以得到 $f=\ln(p_d)$ (这里的 p_d 是真实的真样本的分布，并且p_n 满足在任何 p_d 不为0处的点都不为0)，而且并不需要对 f 有任何的限制（即 f 能够自动满足 $\int e^f=1$），文章里没有给出证明。

一个可能的证明

吃饭的时候稍稍想了一下，觉得可能可以如此证明：

由前面的定义，$l(\theta)$ 就是在参数 θ 下数据遵循该标签序列的概率。假设样本空间是有限的，即只有$x_1,x_2,....,x_k$ 这几种，假设真实的概率密度函数 $p_d(x_i)=pd_i$,假样本的概率密度函数 $p_n(x_i)=pn_i$，而 θ 参数下产生的样本分布的概率密度函数是 $p_m(x_i)=pm_i$，那么很显然有，

$$\text{标签出现的概率}=e^{l(\theta)}=\prod_i(\frac{pm_i}{pn_i+pm_i})^{pd_i\cdot N}(\frac{pn_i}{pn_i+pm_i})^{pn_i\cdot N}$$

N表示样本总数。

把N舍去，于是有 $l(\theta)=\sum pd_i\ln(\frac{pm_i}{pn_i+pm_i})+pn_i\ln(\frac{pn_i}{pn_i+pm_i})$

通过对 pm_i 求导，很容易发现 $l(\theta)$ 取最大值的时候必有 $pm_i=pd_i$，值得注意的这个不等式并没有 $\sum pm_i=1$ 的约束条件，只要 $l(\theta)$ 取得极值，那么必然满足 pm_i 的正则性。因此优化过程中完全不用考虑 softmax 了。

在MNIST数据集上测试

对于MNIST数据集，真实的样本空间是 one-hot 的 0到9，因此我们可以按照均匀分布生成虚假样本，即假样本中0到9出现的概率都是0.1。

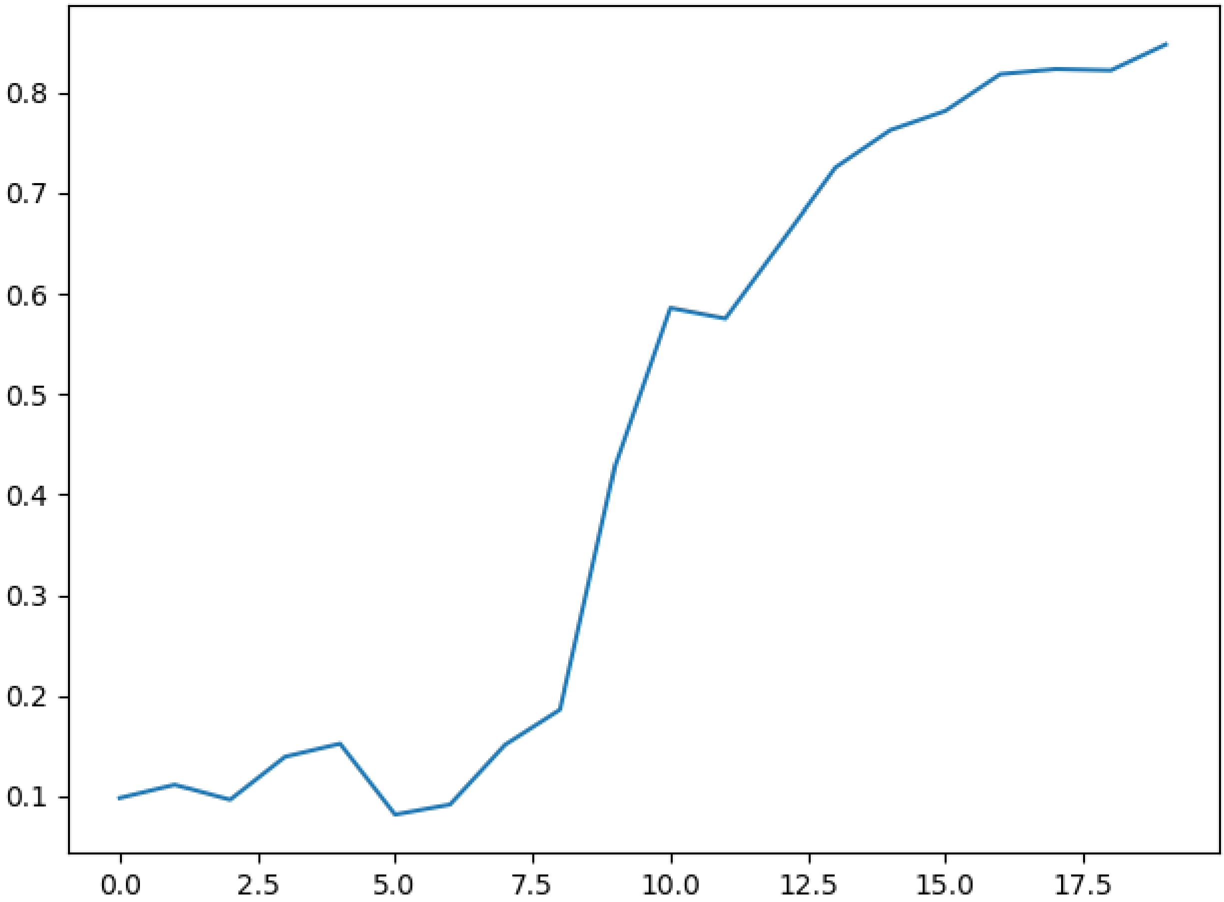
于是上述的 $h(\vec{x}) = \dfrac{p_m(\vec{x})}{0.1 + p_m(\vec{x})}$ ，显然这个值是属于[0,1)的，于是直接让神经网络最后一层用 sigmoid激活，第 i 个输出便是 $h(\vec{x}_i)$ 。 每次训练一批真实的样本时，产生同样数量的虚假样本进行训练。（相当于对于每张图片，先把真实标签放进去训练，再把虚假标签放进去训练）。

代码如下

```
from tensorflow.examples.tutorials.mnist import input_data
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
#import pylab
images = mnist.train.images
input_num = 784
hidden_num = 40
output_num = 10
input_shape = [None,784]
n_classes = [None,10]
x = tf.placeholder(tf.float32,input_shape)
label = tf.placeholder(tf.float32, n_classes)
W = tf.Variable(tf.random_normal([input_num,hidden_num],0,0/input_num))
b = tf.Variable(tf.random_normal([hidden_num],0,1))
layer1 = tf.nn.relu(tf.matmul(x,W) + b)
W1 = tf.Variable(tf.random_normal([hidden_num,output_num],0,0/hidden_num))
b1 = tf.Variable(tf.random_normal([output_num],0,1))
layer2= tf.matmul(layer1,W1) + b1
y = tf.nn.sigmoid (layer2)
sess = tf.InteractiveSession()
tf.global_variables_initializer().run()
# 对于标签为i的样本，这里计算的就是第i个输出，不考虑其他输出。
h = tf.reduce_sum(y * label, reduction_indices=[1]) / (0.1 + tf.reduce_sum(y * label,
reduction_indices=[1]))
loss_true = - tf.reduce_mean(tf.log(h))
loss_false= - tf.reduce_mean(tf.log(1 - h))
train_step = tf.train.GradientDescentOptimizer(0.1).minimize(loss_true)
train_false_step = tf.train.GradientDescentOptimizer(0.1).minimize(loss_false)
accs = []
for _ in range(2000):
    batch_xs, batch_ys = mnist.train.next_batch(40)
    sess.run(train_step, feed_dict={x: batch_xs, label: batch_ys})
    for i in range(1):
        false_ys = np.zeros([40, 10])
        for j in range(40):
            false_ys[j, np.random.random_integers(0, 9)] = 1
        sess.run(train_false_step, feed_dict={x: batch_xs, label: false_ys})
    #Test
    correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(label, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

```
    if _ % 100 == 0:
        accs.append(sess.run(tf.reduce_mean(tf.cast(tf.equal(tf.argmax(y, 1),
tf.argmax(label, 1)), dtype=tf.float32)), feed_dict={x: mnist.test.images, label:
mnist.test.labels}))
print(accs)
plt.plot(accs)
plt.show()
```

训练的结果如下



虽然训练结果很一般，因为没有进行参数调优，网络结构也比较简单，训练轮数也不多。但是说明了利用Noise-Contrastive Estimation 的确可以有效地回避Softmax的运算，因为MNIST数据集只有10种分类，效果不明显。但是对于自然语言处理，比如词向量的生成，如果要把整个词汇表上的概率进行softmax运算，开销就十分巨大了，因此NCE就显得十分必要。