学习Tensorflow的Embeddings例子

14 Nov 2016

Udacity 上有一个 Google 技术人员提供的基于 Tensorflow 的深度学习课程，今天学到 Embeddings ，有点难理解，所以写个笔记记录下，以备日后回忆。

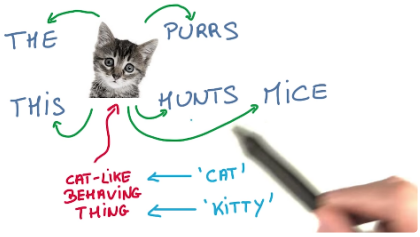
链接：

Udacity**课程视频** 这个课程在 Udacity 上的难度级别已经是 *高* 了。估计再下去就更少视频学习内容了。:~(

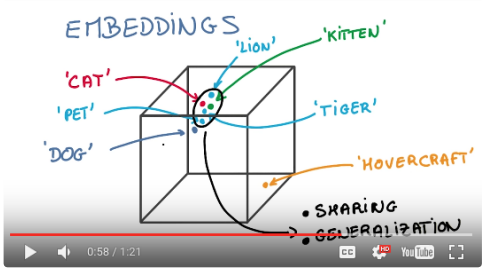
例子Github**地址**

理解课程：

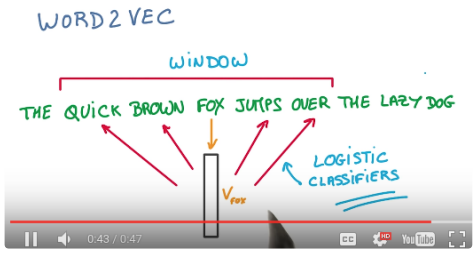
- 课程使用的实现无监督文本学习的根据：**相似的词，会伴随相似的上下文。**
  （我记得有人说过，看一个人的朋友，就知道这个人大致是怎样，看来词也一样。）如下图：



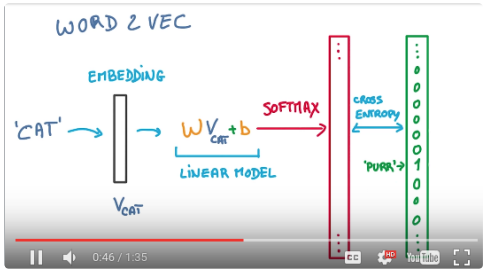
- Embeddings的目标，就是把词都放到一个向量空间里去，这样相近的词就聚集在一起。有了这个模型以后，就可以做很多应用，例如找近义词、某一类词聚类、甚至进行向量加减来寻找衍生词。如下图：



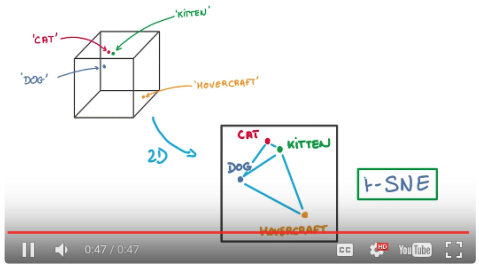
- 如何建立这个Embeddings呢？首先使用一个工具叫word2vec。word2vec算法描述是这样，载入句子，从句子中取出一个词，例如FOX，将他放入Embeddings向量空间（最初位置肯定是随机的），然后通过一次逻辑回归分类预测出他对应的词语，然后与文中实际的QUICK BROWN JUMPS OVER四个词比对，并修正他在Embeddings向量空间里的位置。反复上述步骤，便可得到Embeddings向量空间。（TODO:我这段还需要再理解下）如下图：



- 再来看一下word2vec的具体流程图：把词 `cat` 放进Embeddings向量空间，然后做一次线性计算，然后取softmax，得到一个一批0-1的数值，然后cross_entropy，产出预测词 `purr` 。跟目标比对，然后调整。这就是训练过程。如下图：



- 因为Embeddings向量空间是高维的（需要几维可以自己定义，比如说128维），要想直观的看到他，可以使用t-SNE降维技术，这个技术据说比原始的PCA降维要先进，能保留更多信息。



读例子程序：

[点此查看notebook]

1. 引入库文件

先是引入一些库文件。第一行是如果要在Jupyter Notebook里运行的话，加这一行 `%matplotlib inline` 表示输出的图直接嵌入到Notebook里。

`__future__` 和 `six` 都是保证python2 3兼容的做法。

我们可以看到， `TSNE` 库，tensorflow并没有，所以使用的是 `sklearn` 的实现。
`collections` 是python自带的一个工具库，据说很好，新手可以看这篇**中文介绍博文**。

```
# These are all the modules we'll be using later. Make sure you can import them
# before proceeding further.
%matplotlib inline
from __future__ import print_function
import collections
import math
import numpy as np
import os
import random
import tensorflow as tf
import zipfile
from matplotlib import pylab
from six.moves import range
from six.moves.urllib.request import urlretrieve
from sklearn.manifold import TSNE
```

2. 下载压缩包

下载数据集。如果网速不快，可以用下载工具下载，地址是 http://mattmahoney.net/dc/text8.zip 。保存到运行目录即可。

如果手工解压开查看具体内容，你会开到这个文件里没有标点，全部小写，词与词之间空格隔开：

```
anarchism originated as a term of abuse first used against early working class ...
```

英语不太好，去了标题之后，我都看不太懂是啥意思。想起了我们的没有标点古文。

```
url = 'http://mattmahoney.net/dc/'

def maybe_download(filename, expected_bytes):
  """Download a file if not present, and make sure it's the right size."""
  if not os.path.exists(filename):
    filename, _ = urlretrieve(url + filename, filename)
  statinfo = os.stat(filename)
  if statinfo.st_size == expected_bytes:
    print('Found and verified %s' % filename)
  else:
    print(statinfo.st_size)
    raise Exception(
      'Failed to verify ' + filename + '. Can you get to it with a browser?')
  return filename

filename = maybe_download('text8.zip', 31344016)
```

```
filename = maybe_download('text8.zip', 31344016)
```

3. 读入文本

然后是读入压缩包里第一个文件的所有内容，并以空格分割，形成一个很大的list。

这里tf.compat.as_str只是确保一下是string，应该没什么额外用途。

tf.compat包里面都是一些关于兼容性（compatibility)的小工具。

```
def read_data(filename):
  """Extract the first file enclosed in a zip file as a list of words"""
  with zipfile.ZipFile(filename) as f:
    data = tf.compat.as_str(f.read(f.namelist()[0])).split()
  return data

words = read_data(filename)
print('Data size %d' % len(words))
```

4. 数据预处理

`collections.Counter` 很厉害，可以很方便的数元素出现了几次。以后就不要自己造轮子了，用这个Counter可高效多了。

`UNK` 应该是Unknow的缩写，就是这边只统计50000-1个常用词，剩下的词统称 `UNK` 。

`count` 就是包括 `UNK` 在内的所有50000-1个常用词的词语和出现次数。

按照 `count` 里的词先后顺序，给词进行编号， `UNK` 是0，出现最多的 `the` 是1，出现第二多的 `of` 是2。

`dictionary` 就是词到编号的对应关系，可以快速查找词的编号： `index = dictionary(word)` 。

`reverse_dictionary` 则是编号到词的对应关系，可以快速查找某个编号是什么词： `word = reverse_dictionary(index)` 。

最后 `data` 是把原文的词都转化成对应编码以后的串。

保存 `dictionary` 和 `reverse_dictionary` 这一点十分值的学习，对于频繁的查询，这样的缓存能大大增加速度。如果用函数的方式，每次查都要轮询，就土了。

```
vocabulary_size = 50000

def build_dataset(words):
  count = [['UNK', -1]]
  count.extend(collections.Counter(words).most_common(vocabulary_size - 1))
  dictionary = dict()
  for word, _ in count:
    dictionary[word] = len(dictionary)
  data = list()
  unk_count = 0
  for word in words:
    if word in dictionary:
      index = dictionary[word]
    else:
      index = 0  # dictionary['UNK']
      unk_count = unk_count + 1
    data.append(index)
  count[0][1] = unk_count
  reverse_dictionary = dict(zip(dictionary.values(), dictionary.keys()))
  return data, count, dictionary, reverse_dictionary

data, count, dictionary, reverse_dictionary = build_dataset(words)
print('Most common words (+UNK)', count[:5])
print('Sample data', data[:10])
del words  # Hint to reduce memory.
```

5. 生成训练数据集函数

准备要生成可供训练的数据集了。

这里使用全局变量 `data_index` 来记录当前取到哪了，每次取一个batch后会向后移动，如果超出结尾，则又从头开始。 `data_index = (data_index + 1) % len(data)` （怎么把这东西放在全局变量里，是不是有点不够漂亮？）

`skip_window` 是确定取一个词周边多远的词来训练，比如说 `skip_window` 是2，则取这个词的左右各两个词，来作为它的上下文词。后面正式使用的时候取值是1，也就是只看左右各一个词。

这里的 `num_skips` 我有点疑问，按下面注释是说， `How many times to reuse an input to generate a label.` ，但是我觉得如果确定了 `skip_window` 之后，完全可以用

`num_skips=2*skip_window` 来确定需要reuse的次数呀，难道还会浪费数据源不成？

这边用了一个双向队列 `collections.deque` ，第一次遇见，看代码与 `list` 没啥区别，从网上简介来看，双向队列主要在左侧插入弹出的时候效率高，但这里并没有左侧的插入弹出呀，所以是不是应该老实用 `list` 会比较好呢？

然后要维护一个 `targets_to_avoid` ，如果不是左右 `skip_window` 个词都用起来的话，则通过随机函数来随机选取对应的词。还是上面那个问题，为啥不直接全用啊？不全用还得随机，想不明白。

这一段代码感觉写的怪怪的，疑点比较多，如果有能知道其中淫巧的同学，可以留言指点一下。

```
data_index = 0

def generate_batch(batch_size, num_skips, skip_window):
  global data_index
  assert batch_size % num_skips == 0
  assert num_skips <= 2 * skip_window
  batch = np.ndarray(shape=(batch_size), dtype=np.int32)
  labels = np.ndarray(shape=(batch_size, 1), dtype=np.int32)
  span = 2 * skip_window + 1 # [ skip_window target skip_window ]
  buffer = collections.deque(maxlen=span)
  for _ in range(span):
    buffer.append(data[data_index])
    data_index = (data_index + 1) % len(data)
  for i in range(batch_size // num_skips):
    target = skip_window  # target label at the center of the buffer
    targets_to_avoid = [ skip_window ]
    for j in range(num_skips):
      while target in targets_to_avoid:
        target = random.randint(0, span - 1)
```

```
        targets_to_avoid.append(target)
        batch[i * num_skips + j] = buffer[skip_window]
        labels[i * num_skips + j, 0] = buffer[target]
      buffer.append(data[data_index])
      data_index = (data_index + 1) % len(data)
    return batch, labels
```

这里是测试一下 `generate_batch` 函数，并无实际用途。

```
print('data:', [reverse_dictionary[di] for di in data[:8]])

for num_skips, skip_window in [(2, 1), (4, 2)]:
  data_index = 0
  batch, labels = generate_batch(batch_size=8, num_skips=num_skips, skip_window=skip_window)
  print('\nwith num_skips = %d and skip_window = %d:' % (num_skips, skip_window))
  print('    batch:', [reverse_dictionary[bi] for bi in batch])
  print('    labels:', [reverse_dictionary[li] for li in labels.reshape(8)])
```

6. 超参数

不管怎样， `generate_batch` 函数算是准备好了，先备用，后面真实训练的时候会用到。

接下来来定义一些超参数吧，这些超参数你也可以根据你的需要修改，来看看是否训练出来的Embeddings更符合你的需要。

从前面图上，有人或许会以为Embeddings向量空间只是三维的，其实不是，它是高维的。这里定义 `embedding_size` 是128维。

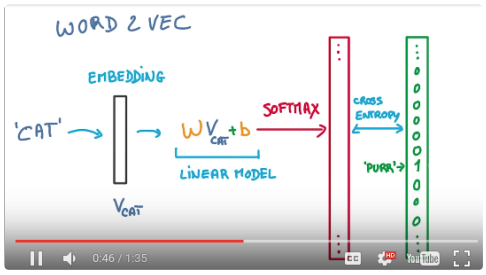
然后定义用来供人直观检验validate的一些参数。

`num_sampled` 则是Sample Softmax时候用到的一个超参数，确定选几个词来对比优化。

```
batch_size = 128
embedding_size = 128 # Dimension of the embedding vector.

skip_window = 1 # How many words to consider left and right.
num_skips = 2 # How many times to reuse an input to generate a label.
# We pick a random validation set to sample nearest neighbors. here we limit the
# validation samples to the words that have a low numeric ID, which by
# construction are also the most frequent.
valid_size = 16 # Random set of words to evaluate similarity on.
valid_window = 100 # Only pick dev samples in the head of the distribution.
valid_examples = np.array(random.sample(range(valid_window), valid_size))
num_sampled = 64 # Number of negative examples to sample.
```

7. 定义Tensorflow模型

终于要开始定义Tensorflow的模型了。先回顾一下视频截图：



`train_dataset` 和 `train_labels` 两个 `placeholder` 用来训练时传入x和y。

`valid_dataset` 则是用来人工验证的小数据集，是 `constant` ，直接赋值前面生成的 `valid_examples` 。

`embeddings` 是用来存储Embeddings向量空间的变量，初始化成-1到1之间的随机数，后面优化时调整。这里它是一个 `50000` * `128` 的二维变量。

`softmax_weights` 和 `softmax_biases` 是用来做线性逻辑分类的参数。

```
graph = tf.Graph()

with graph.as_default():

  # Input data.
  train_dataset = tf.placeholder(tf.int32, shape=[batch_size])
  train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
  valid_dataset = tf.constant(valid_examples, dtype=tf.int32)

  # Variables.
  embeddings = tf.Variable(
    tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0))
  softmax_weights = tf.Variable(
    tf.truncated_normal([vocabulary_size, embedding_size],
                        stddev=1.0 / math.sqrt(embedding_size)))
  softmax_biases = tf.Variable(tf.zeros([vocabulary_size]))
```

8. 模型前半部分

通过 `tf.nn.embedding_lookup` 可以直接根据 `embeddings` 表(50000,128)，取出一个与输入词对应的128个值的 `embed` ，也就是128维向量。其实是一batch同时处理，但说一个好理解一些。

通过 `tf.nn.sampled_softmax_loss` 可以用效率较高的Sample Softmax来得到优化所需要的偏差。这个方法视频里有带过，反正就是全部比对速度慢，这样速度快。这个方法顺带把 wX+b 这步也一起算了。可能是因为放在一起可以优化计算速度，记得还有那个很长名字的 `softmax_cross_entropy_with_logits` 同时搞定softmax和cross_entropy，也是为了优化计算速度。但，这样的代码读起来就不好看了！

通过 `tf.reduce_mean` 把 `loss` 偏差压到一个数值，用于优化。

然后就是优化，这里使用了 `AdagradOptimizer` ，当然也可以使用其他 `SGD` 、 `Adam` 等各种优化算法，Tensorflow都实现同样的接口，只需要换个函数名就可以。

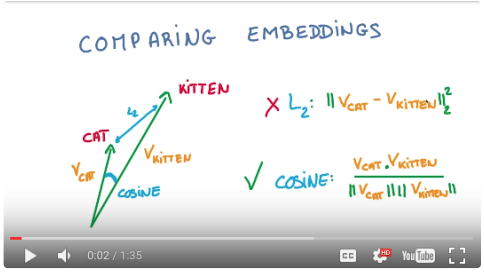
这里注释提到，因为 `embeddings` 是定义成 `tf.Variable` 的，所以在优化的时候同时也会调整 `embeddings` 里的参数。这是因为 `minimize` 函数会将与传入的 `loss` 连接的所有源头变量进行调整优化。

```
  # Model.
  # Look up embeddings for inputs.
```

```
    embed = tf.nn.embedding_lookup(embeddings, train_dataset)
    # Compute the softmax loss, using a sample of the negative labels each time.
    loss = tf.reduce_mean(
      tf.nn.sampled_softmax_loss(softmax_weights, softmax_biases, embed,
                                 train_labels, num_sampled, vocabulary_size))

    # Optimizer.
    # Note: The optimizer will optimize the softmax_weights AND the embeddings.
    # This is because the embeddings are defined as a variable quantity and the
    # optimizer's `minimize` method will by default modify all variable quantities
    # that contribute to the tensor it is passed.
    # See docs on `tf.train.Optimizer.minimize()` for more details.
    optimizer = tf.train.AdagradOptimizer(1.0).minimize(loss)
```

9. 模型关键部位

这里写的有点难看懂，必须再刷一次视频。



原来这是要计算一个 `valid_dataset` 中单词的相似度。

首先， `norm` 是模的意思，也就是二范数，就是这个向量和原点的距离，土话就是这个向量的长度。

计算 `norm` 的方法，就是向量各维度的平方和：

$$norm = \|X\| = \sqrt{\sum_{i=1}^{N} X_i^2}$$

然后利用这个向量长度 `norm` 来给向量标准化：

$$X_{normalized} = \frac{X}{\|X\|}$$

这样得到的就是长度为1的向量。也就是抛弃了向量的长度信息，就剩下方向信息。（感谢潘程同学提醒，这东西就叫"**单位向量**"，可以表示为 $\hat{X}$！）

接下来，

造一些简单的数据，用 `numpy` 模拟一下这一段代码，确认一下这段代码的意义：

```
>> embeddings = np.array([[1,2,3,4],[2,4,6,8],[2,2,2,2],[3,3,3,3]])
[[1 2 3 4]
 [2 4 6 8]
 [2 2 2 2]
 [3 3 3 3]]
>> s = np.square(embeddings)
[[ 1  4  9 16]
 [ 4 16 36 64]
 [ 4  4  4  4]
 [ 9  9  9  9]]
>> r = np.sum(s, axis=1, keepdims=1)
[[ 30]
 [120]
 [ 16]
 [ 36]]
>> norm = np.sqrt(r)
[[  5.47722558]
 [ 10.95445115]
 [  4.        ]
 [  6.        ]]
>> normalized_embeddings = embeddings / norm
[[ 0.18257419  0.36514837  0.54772256  0.73029674]
 [ 0.18257419  0.36514837  0.54772256  0.73029674]
 [ 0.5         0.5         0.5         0.5       ]
 [ 0.5         0.5         0.5         0.5       ]]
>> sim = np.dot(normalized_embeddings,normalized_embeddings.T)
[[ 1.          1.          0.91287093  0.91287093]
 [ 1.          1.          0.91287093  0.91287093]
 [ 0.91287093  0.91287093  1.          1.        ]
 [ 0.91287093  0.91287093  1.          1.        ]]
```

可以看到，向量(1,2,3,4)和向量(2,4,6,8)在经过标准化之后，变成了同样的值(0.18,0.36,0.54,0.73)。同样向量(2,2,2,2)和(3,3,3,3)也都变成了一样的(0.5,0.5,0.5,0.5)。

也就是说，这个操作是针对每一个向量，各自做 Normalization 标准化。

也就是说，这个标准化操作抛弃了向量的长度，只关注向量的方向。视频里说，比较两个embeddings向量的时候，用的是cosine距离。

然后从标准化后的向量空间里，找出用于检验的词语对应的向量值， `valid_embeddings` 这个变量名有点迷惑人，我建议将其改名为 `valid_embed` ，以对应前文中的 `embed` 变量，他们俩是有相似意义的。

最后通过**余弦相似性原理**，计算出 `similarity` ：

$$similarity = cos(\theta) = \frac{A \cdot B}{\|A\| \cdot \|B\|} = A_{normalized} \cdot B_{normalized}$$

当 $\theta = 0$ 时，两者重合， $similarity = cos(0) = 1$。

用检索出来的结果矩阵（前面模拟里我用了全部矩阵）乘以 `normalized_embeddings` 的转置，得到的结果最大值是1也就是相同，值越大代表越相似。

```
    # Compute the similarity between minibatch examples and all embeddings.
    # We use the cosine distance:
```

```
  # We use the cosine distance:
  norm = tf.sqrt(tf.reduce_sum(tf.square(embeddings), 1, keep_dims=True))
  normalized_embeddings = embeddings / norm
  valid_embeddings = tf.nn.embedding_lookup(
    normalized_embeddings, valid_dataset)
  similarity = tf.matmul(valid_embeddings, tf.transpose(normalized_embeddings))
```

10. 迭代训练

模型定义完，就来执行训练了。

这里有个小bug，新版本的Tensorflow不再支持 `global_variables_initializer` 方法，而改为使用 `initialize_all_variables` 方法，需要手工修改一下教程代码。（年轻而高速发展的Tensorflow里面有很多类似的接口变动，所以如果一个程序跑不通，去github上看看Tensorflow的源代码很有必要。）

定义迭代次数，100001次，每一次迭代，都从 `generate_batch` 里面获取一个batch的训练数据。制作成 `feed_dict` ，对应前面模型里定义的 `placeholder` 。

跑一次优化，记录下loss并加起来，等到每2000次的时候，输出一次平均loss。（又一个细节：这里的 `average_loss` 是否应该取名叫 `subtotal_loss` ？）

等到每10000次，获取一下 `similarity` 对应的值，然后把最相似的几个词输出到屏幕，用于人工检验。 `argsort` 是获取排序后的indices。然后通过 `reverse_dictionary` 快速定位到词是什么。

```
num_steps = 100001

with tf.Session(graph=graph) as session:
  tf.global_variables_initializer().run()
  print('Initialized')
  average_loss = 0
  for step in range(num_steps):
    batch_data, batch_labels = generate_batch(
      batch_size, num_skips, skip_window)
    feed_dict = {train_dataset : batch_data, train_labels : batch_labels}
    _, l = session.run([optimizer, loss], feed_dict=feed_dict)
    average_loss += l
    if step % 2000 == 0:
      if step > 0:
        average_loss = average_loss / 2000
      # The average loss is an estimate of the loss over the last 2000 batches.
      print('Average loss at step %d: %f' % (step, average_loss))
      average_loss = 0
    # note that this is expensive (~20% slowdown if computed every 500 steps)
    if step % 10000 == 0:
      sim = similarity.eval()
      for i in range(valid_size):
        valid_word = reverse_dictionary[valid_examples[i]]
        top_k = 8 # number of nearest neighbors
        nearest = (-sim[i, :]).argsort()[1:top_k+1]
        log = 'Nearest to %s:' % valid_word
        for k in range(top_k):
          close_word = reverse_dictionary[nearest[k]]
          log = '%s %s,' % (log, close_word)
        print(log)
```

训练完成，获取最后得到的 `embeddings` 向量空间！

```
  final_embeddings = normalized_embeddings.eval()
```

11. 降维可视化

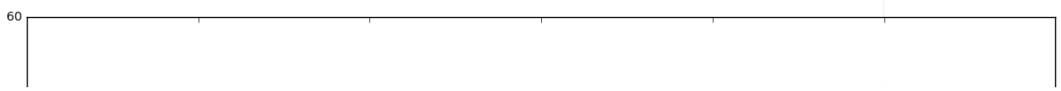
下面把 `embeddings` 向量空间通过t-SNE（t-distributed Stochastic Neighbor Embedding）降维到2D，然后打出来看看。

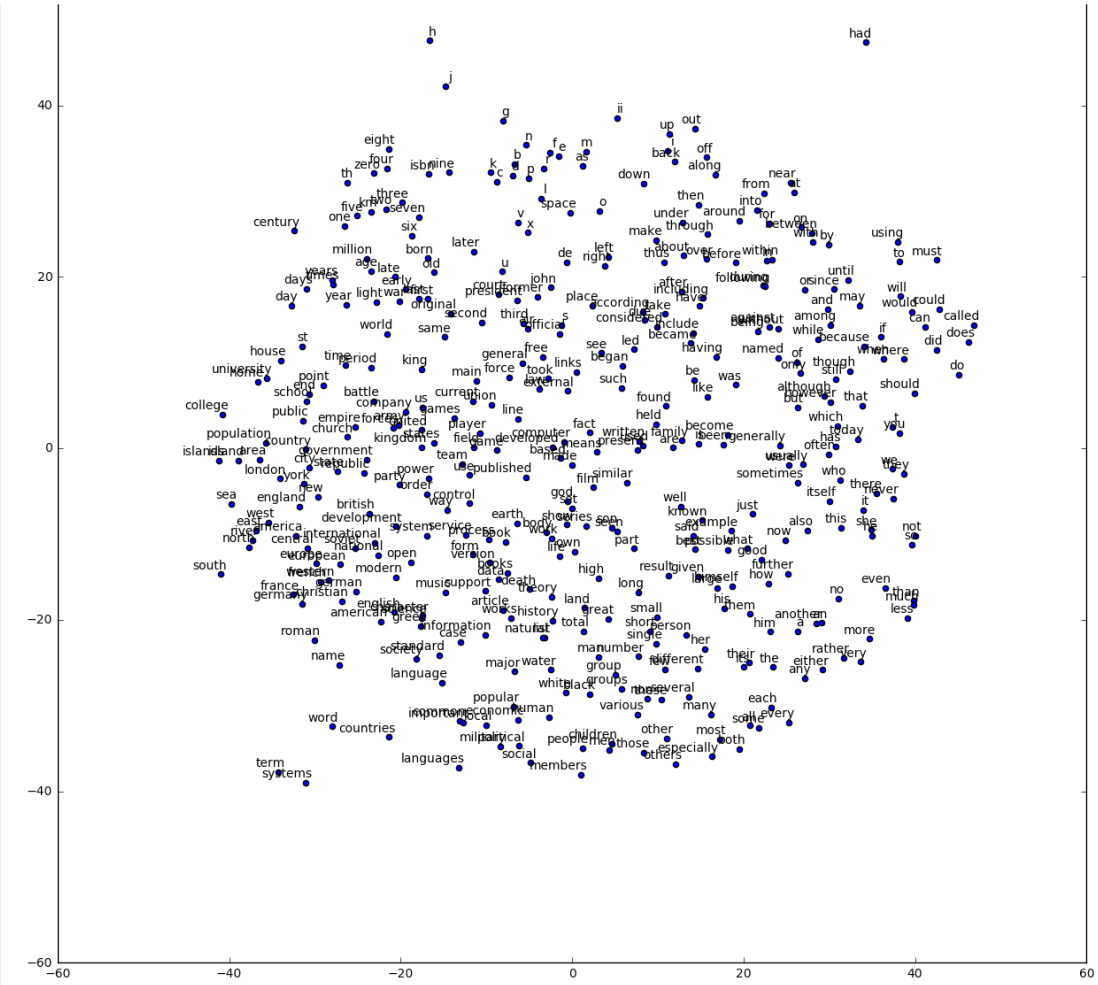
其实根据t-SNE的**文档**，如果向量空间维数过高，比如说我们把128维改成1024维，或者更高，那么这里建议先使用PCA降维方法，降到50维左右以后，再使用t-SNE来继续降到二维或者三维。因为直接用t-SNE给高维向量空间做降维的话，效率会比较低。

画图部分这里用 `pylab` ，就不再赘述，如果喜欢用 `matplotlib` ，也是类似的。（pylab是对标matlab的库，matplotlib也包含在pylab里。）

```
num_points = 400

tsne = TSNE(perplexity=30, n_components=2, init='pca', n_iter=5000)
two_d_embeddings = tsne.fit_transform(final_embeddings[1:num_points+1, :])

def plot(embeddings, labels):
  assert embeddings.shape[0] >= len(labels), 'More labels than embeddings'
  pylab.figure(figsize=(15,15))  # in inches
  for i, label in enumerate(labels):
    x, y = embeddings[i,:]
    pylab.scatter(x, y)
    pylab.annotate(label, xy=(x, y), xytext=(5, 2), textcoords='offset points',
                   ha='right', va='bottom')
  pylab.show()

words = [reverse_dictionary[i] for i in range(1, num_points+1)]
plot(two_d_embeddings, words)
```

嗯，我们会看到这样的结果：





一个小debug处理

这个例子里用到了sklearn的tSNE来降维作显示，但每次运行到这里Jupyter都提醒Kernel死掉了，Kernel重启。于是把源码复制到单独python文件里执行，看到如下错误：

```
Intel MKL FATAL ERROR: Cannot load libmkl_avx2.so or libmkl_def.so.
```

这个问题似乎是Intel的MKL有问题，网上查到这篇为Anaconda2.5提供MKL优化，里面也提供了去掉MKL优化的方法：

```
conda install nomkl numpy scipy scikit-learn numexpr conda remove mkl mkl-service
```

执行过之后，就去除了mkl库，import mkl就找不到了，同时例子程序里的问题也没有再出现。
如果还是想使用Intel MKL优化的话，或许可以参考Anaconda官方文档MKL OPTIMIZATIONS来重新安装过。不过，我先不装了。

张量 Tensor 的 阶数 Rank ，向量空间 Vector Space 和 向量 Vector 的 维数 Dimension

前面我在写到Embeddings是128维的时候，脑子有点晕，原来是混淆了Tensor的维数和Vector的维数。

为了搞清楚概念，我查了下资料，应该是这样的：

我们平时说的128维向量，是指一个向量，值是有128个数组成，代表128个方向上的量，例子里的 `embed` 变量就是一个batch的128维向量。

向量空间是一群向量的集合，我们这里 `embeddings` 变量就是一个50000个向量集合起来的向量空间，所以他的 `shape` 是 `(50000,128)` 。

张量 Tensor 也有一个维数，为了和向量的维数区分，一般成其为阶数 Rank。比如说变量 `embeddings` 是一个128维的向量空间，同时他也是一个二阶张量，也就是一张二维表格，行是单词，列是这个单词的128个值。假设其他地方我们看到 `128维张量` 的话，那么它的 `shape` 应该是(xx,xx,xx,xx,....,xx)一共128个xx数，一般来说现在我们学习中还没有碰到这么高阶的张量，可能4阶张量已经很多了，在Convolutional Neural Network里用到的输入，它的 `shape` 是 `(batch, image_width, image_height, image_channel)` ，就是一个4阶张量（其实应该算一个batch的3阶张量的集合）。注意这里的 `image_width` 不是说输入的是图片的宽度值，而是输入的是宽度为 `image_width` 的图片本身。

这就是阶和维的区别。

Related Posts

Cross Entropy 的通俗意义 25 Nov 2016
推荐《写下记忆：理解、推导、扩展LSTM》 24 Nov 2016
学习Tensorflow的LSTM的RNN例子 16 Nov 2016