在上一节课，我们简单介绍了自然语言处理的发展历史，然后讲解了 TF-IDF 算法的计算过程，那是一个非常古老的关键词计算方法。今天，我们要学习自然语言处理的再次爆发期产生的一种新算法：词嵌入算法。

简单来说，词嵌入算法就是使用一个低维度的向量来表示一个词，并且距离相近的向量在实际的词含义上也是相近的，比如说 “炸鸡” 的向量与 “啤酒” 的向量距离就要比 “炸鸡” 的向量与 “收音机” 的向量要近。

不仅如此，词嵌入获得的向量还具备数学运算的关系，比如 “女人” 词向量 + “王冠”词向量 ≈ “女王”词向量。这是不是很神奇？所以这个到底是怎么实现的呢？我们要先从词向量的表达说起。

独热编码

在计算机中的所有运算都是数学运算，怎么把我们的自然语言处理转换成一种数学形态，一直是一个困难的问题。文本的长度本身就不确定，而且文本中的词每次也都不一样，因此很难有非常有效的编码方式来表达文本。

不过有一种使用了很长时间的编码方式就是独热编码（One Hot），独热编码是一位有效编码，其中的每一位都用来存储一个状态。

现在来举个例子，假设我们的数据里只有 “男”“女” 两个字，那么我们的独热编码就像下面写的这样，一共有两位。

男	**01**
女	10

独热编码的好处就是能够让我们的离散的文本数据都变成定长的数据，方便各种算法的处理。但是它的缺点也很明显，那就是过于稀疏。假设我们的语料库是最近一年的新闻，那可能会有上千万的词，哪怕经过了各种过滤，仍然会有上百万的长度，每一个词的向量只有一位是 1，其他几十万位都是 0。

比如：

特朗普 [1,0,0,0,0,0,.....,0,0,0]

拜登 [0,1,0,0,0,0,.....,0,0,0]

希拉里 [0,0,1,0,0,0,.....,0,0,0]

克林顿 [0,0,0,1,0,0,.....,0,0,0]

奥巴马 [0,0,0,0,0,1,.....,0,0,0]

这样的维度对于我们的算法来说实在是一个大灾难，不管是存储还是运算的开销都非常大。同时，独热编码也没办法记录词与词之间的关系。

随着研究的深入，产生了一种分布式表示的方法来解决独热编码的问题。它的想法是通过对文本词的训练，把每个词都映射到一个比较短、但是稠密的向量上来。所有的词向量构成了向量空间，从而可以使用统计学方法来研究词之间的关系。具体要把词向量的维度设为多少，需要我们在训练的时候指定。

比如我们用几个维度来表示上面的几个人物名词：

	特朗普	拜登	希拉里	克林顿	奥巴马
性别	0.99	0.99	0.01	0.99	0.99
年龄	0.7	0.9	0.7	0.8	0.6
肤色	0.3	0.1	0.2	0.2	0.9
权力	0.8	0.3	0.4	0.9	0.8
财富	0.8	0.7	0.4	0.9	0.2

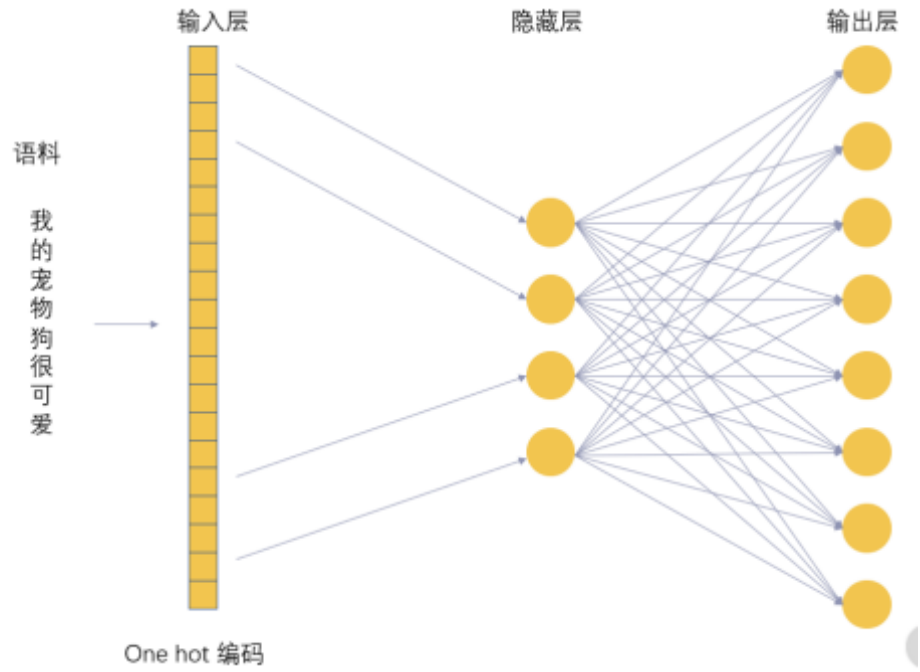
经过上面表达的转换，我们就已经把几个词向量由十分稀疏的独热编码转换成了一种稠密的向量。当然，在我们的模型中没有办法去解释每个维度都是什么意思，并且每一个向量只对应一个词，每个词的表达向量是不同的，映射之后的向量包含了原来的信息。

把原本的词向量映射到这个相对低维空间的过程就称为词嵌入（Word Embedding）。但是词嵌入之所以能够记录词与词之间的关系，主要是因为在训练的时候使用了上下文关系来进行学习。比如说，在我们的语料中经常出现 “我的宠物猫是一只可爱的小动物”“我的宠物狗是一只可爱的小动物”，那么“宠物猫” 和“宠物狗”这两个词的词向量就会非常接近，这是因为它们的上下文都很类似。

说到这里，基本铺垫都已经介绍完了，你大概能够明白词嵌入和词向量的概念，那么接下来我们要进入正题，看看 Word2Vec 算法是如何计算出我们的词向量的。

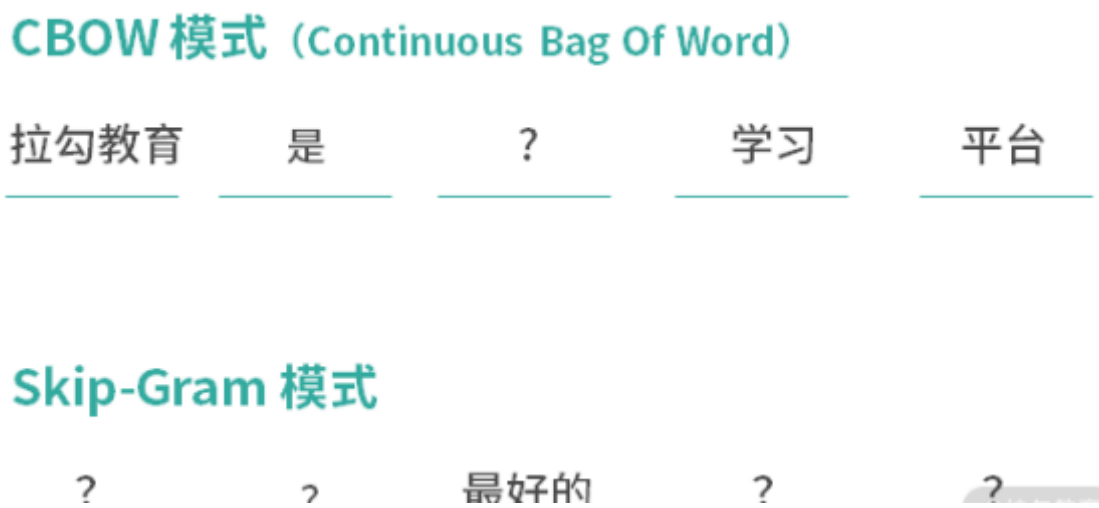
算法原理

我们前面讲过人工神经网络，而 Word2Vec 最核心的其实就是采用了神经网络算法，当然它是浅层神经网络，只包含了一层隐藏层。它的网络结构如下图：



输入层是我们去掉了某些部分的语料编码，在这里也就是 One Hot 编码，输出层的维度与输入层一样，所需要预测输出的是在输入层被去掉的部分。

所以这里有两种方案，第一个是去掉某个词，输入层是这个词的上下文，这种方法叫作 CBOW（Continuous Bag Of Word），输出层也就是要去预测这个词；第二种方案是去掉上下文，用这个词作为输入层，这种方法叫作 Skip-Gram，输出层需要预测上下文。



对于这个网络中，从输入层到隐藏层是没有激活函数的，也就是一个线性关系。这里需要注意的是 Word2Vec 所获得的模型并不是我们这个网络的最终结果，而是在训练完成之后，把隐藏层的权重矩阵获取出来，即形成了我们的 Word2Vec 算法的结果。

通过这个方法，我们语料中的每一个词都获得了一个预设维度的向量，由于隐藏层的维度要比 onehot 向量的维度小很多，这也起到了很好的降维作用。

讲到这里，关于 Word2Vec 最浅显的原理已经介绍完了，其余的部分就是关于神经网络的一些原理，我们已经在前面的课程介绍过。当然了，关于这个算法的细节部分这里也没有进行太多的讲解，这些内容你如果有兴趣可以通过相关的论文来进行学习。

> 推荐论文：
> xinrong：《word2vec Parameter Learning Explained》
> Quoc Le，Tomas Mikolov：《Distributed Representations of Sentences and Documents》
> Tomas Mikolov，Ilya Sutskever，Kai Chen：《Distributed Representations of Words and Phrases and their Compositionality》

算法优缺点

比起很多传统方法，Word2Vec 考虑了一些上下文信息，以及词语的顺序信息，所以比起传统的基于词语统计的方法要准确很多。而且 Word2Vec 方法成功降低了向量维度，可以适配后续的各种自然语言处理任务，通用性很强。同时，这种预训练的思想为大家的下一步研究提供了良好的思路。

当然，Word2Vec 产生的词向量是一对一的，所以一词多义的问题仍然没有办法解决。

尝试动手

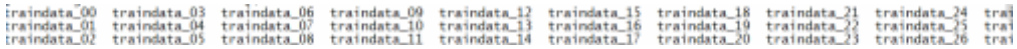
对于 Word2Vec 的代码使用方法，我们前面其实已经介绍过了，在实践课 2 中我们使用了 Word2Vec 为 K-means 聚类算法提供了聚类的向量材料。这里我把代码粘过来，再回顾一下使用的过程。

```
import gensim #引入gensim


import os


import re


import sys
```

```
import multiprocessing #引入多线程操作

from time import time

class getSentence(object):

#初始化，获取文件路径

def __init__(self, dirname):

self.dirname = dirname
```

文本可以存储在多个文本文件中，存放在一个文件目录下，这里构建了一个迭代方法，循环读取目录下的所有文件。

我这里使用的文件目录为 traindata，在 traindata 下面有 31 个语料文件，其中每个有 1G 左右。




```
#构建一个迭代器

    def __iter__(self):

for root, dirs, files in os.walk(self.dirname):

for filename in files:

                file_path = root + '/' + filename

for line in open(file_path):

try:

                        #清除异常数据，主要是去除空白符以及长度为0的内容

                        s_line = line.strip()

if s_line== "":

continue
```

```
                        #把句子拆成词

                         word_line = [word for word in s_line.split( )]

yield word_line

                    except Exception:

print("catch exception")

yield ""

if __name__ == '__main__':

#记录一个起始时间

    begin = time()

#获取句子迭代器

    sentences = getSentence("traindata")

#训练word2vec模型 使用句子迭代器作为语料的输入，设定的最终向量长度为200维；窗口长度为15；词的最小计数为10，词频少于10的词不会进行计算；使用并行处理

    model = gensim.models.Word2Vec(sentences,size=200,window=15,min_count=10,
workersmeans=multiprocessing.cpu_count())

#模型存储，这块记得先预先新建一个model路径，或者也可以增加一段代码来识别是否已经创建，如果没有则新建一个路径

    model.save("model/word2vec_gensim")

    model.wv.save_word2vec_format("model/word2vec_org",

"model/vocabulary",

                                  binary=False)
```

```
    end = time()

#输出运算所用时间

print ("Total procesing time: %d seconds" % (end - begin))
```

总结

看完了代码部分，这节课又将告一段落了。这是我们关于自然语言处理的第二节课程，当然这两节课程只是介绍了自然语言处理浩如烟海的知识中很小的一部分，但是我希望通过这两小节课程的学习，你能够对自然语言处理有一个初步的了解。

在这节课里面，我们介绍了 Word2Vec 算法，从原来的 OneHot 编码讲起，到 Word2Vec 的基本原理以及 Word2Vec 的两种工作模式。不过，这里所介绍的都是最浅显的部分，关于 Word2Vec 算法还有很多的细节我们没有涉及。当然了，在自然语言预训练模型方面，现在已经有了很多更加先进和优秀的算法，比如 BERT、GPT 等，如果你对这方面有兴趣，可以继续学习自然语言处理相关的课程。

下一课时，是我们最后一节实践课——使用 fastText 进行新闻文本分类。