词向量

本教程源代码目录在book/word2vec,初次使用请您参考Book文档使用说明。

说明

1. 本教程可支持在 CPU/GPU 环境下运行
2. Docker镜像支持的CUDA/cuDNN版本

   如果使用了Docker运行Book，请注意：这里所提供的默认镜像的GPU环境为 CUDA 8/cuDNN 5，对于NVIDIA Tesla V100等要求CUDA 9的 GPU，使用该镜像可能会运行失败;
3. 文档和脚本中代码的一致性问题

   请注意：为使本文更加易读易用，我们拆分、调整了train.py的代码并放入本文。本文中代码与train.py的运行结果一致，可直接运行train.py进行验证。

背景介绍

本章我们介绍词的向量表征，也称为word embedding。词向量是自然语言处理中常见的一个操作，是搜索引擎、广告系统、推荐系统等互联网服务背后常见的基础技术。

在这些互联网服务里，我们经常要比较两个词或者两段文本之间的相关性。为了做这样的比较，我们往往先要把词表示成计算机适合处理的方式。最自然的方式恐怕莫过于向量空间模型(vector space model)。 在这种方式里，每个词被表示成一个实数向量（one-hot vector），其长度为字典大小，每个维度对应一个字典里的每个词，除了这个词对应维度上的值是1，其他元素都是0。

One-hot vector虽然自然，但是用处有限。比如，在互联网广告系统里，如果用户输入的query是“母亲节”，而有一个广告的关键词是“康乃馨”。虽然按照常理，我们知道这两个词之间是有联系的——母亲节通常应该送给母亲一束康乃馨；但是这两个词对应的one-hot vectors之间的距离度量，无论是欧氏距离还是余弦相似度(cosine similarity)，由于其向量正交，都认为这两个词毫无相关性。 得出这种与我们相悖的结论的根本原因是：每个词本身的信息量都太小。所以，仅仅给定两个词，不足以让我们准确判别它们是否相关。要想精确计算相关性，我们还需要更多的信息——从大量数据里通过机器学习方法归纳出来的知识。

在机器学习领域里，各种“知识”被各种模型表示，词向量模型(word embedding model)就是其中的一类。通过词向量模型可将一个 one-hot vector映射到一个维度更低的实数向量（embedding vector），如$embedding(\text{母亲节}) = [0.3, 4.2, -1.5, \dots], embedding(\text{康乃馨}) = [0.2, 5.6, -2.3, \dots]$。在这个映射到的实数向量表示中，希望两个语义（或用法）上相似的词对应的词向量“更像”，这样如“母亲节”和“康乃馨”的对应词向量的余弦相似度就不再为零了。

词向量模型可以是概率模型、共生矩阵(co-occurrence matrix)模型或神经元网络模型。在用神经网络求词向量之前，传统做法是统计一个词语的共生矩阵X。X是一个$|V| \times |V|$大小的矩阵，X_{ij}表示在所有语料中，词汇表V(vocabulary)中第i个词和第j个词同时出现的词数，$|V|$为词汇表的大小。对X做矩阵分解（如奇异值分解，Singular Value Decomposition [5]），得到的U即视为所有词的词向量：

$$\mathrm{X} = \mathrm{USV}^T$$

但这样的传统做法有很多问题：

1. 由于很多词没有出现，导致矩阵极其稀疏，因此需要对词频做额外处理来达到好的矩阵分解效果；
2. 矩阵非常大，维度太高(通常达到$10^6 \times 10^6$的数量级)；
3. 需要手动去掉停用词（如although, a,...），不然这些频繁出现的词也会影响矩阵分解的效果。

基于神经网络的模型不需要计算和存储一个在全语料上统计产生的大表，而是通过学习语义信息得到词向量，因此能很好地解决以上问题。在本章里，我们将展示基于神经网络训练词向量的细节，以及如何用PaddlePaddle训练一个词向量模型。

效果展示

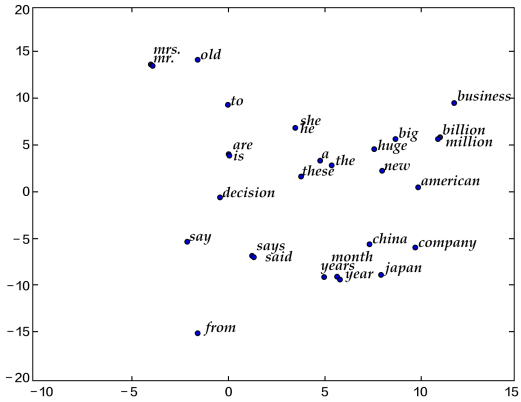

图1. 词向量的二维投影

另一方面，我们知道两个向量的余弦值在$[-1, 1]$的区间内：两个完全相同的向量余弦值为1, 两个相互垂直的向量之间余弦值为0，两个方向完全相反的向量余弦值为-1，即相关性和余弦值大小成正比。因此我们还可以计算两个词向量的余弦相似度:

```
please input two words: big huge
similarity: 0.899180685161

please input two words: from company
similarity: -0.0997506977351
```

以上结果可以通过运行 `calculate_dis.py`，加载字典里的单词和对应训练特征结果得到，我们将在模型应用中详细描述用法。

模型概览

在这里我们介绍三个训练词向量的模型：N-gram模型，CBOW模型和Skip-gram模型，它们的中心思想都是通过上下文得到一个词出现的概率。对于N-gram模型，我们会先介绍语言模型的概念，并在之后的训练模型中，带大家用PaddlePaddle实现它。而后两个模型，是近年来最有名的神经元词向量模型，由 Tomas Mikolov 在Google 研发[3]，虽然它们很浅很简单，但训练效果很好。

语言模型

在介绍词向量模型之前，我们先来引入一个概念：语言模型。 语言模型旨在为语句的联合概率函数$P(w_1, \ldots, w_T)$建模, 其中w_i表示句子中的第i个词。语言模型的目标是，希望模型对有意义的句子赋予大概率，对没意义的句子赋予小概率。 这样的模型可以应用于很多领域，如机器翻译、语音识别、信息检索、词性标注、手写识别等，它们都希望能得到一个连续序列的概率。 以信息检索为例，当你在搜索“how long is a football bame”时（bame是一个医学名词），搜索引擎会提示你是否希望搜索"how long is a football game", 这是因为根据语言模型计算出“how long is a football bame”的概率很低，而与bame近似的，可能引起错误的词中，game会使该句生成的概率最大。

对语言模型的目标概率$P(w_1, \ldots, w_T)$，如果假设文本中每个词都是相互独立的，则整句话的联合概率可以表示为其中所有词语条件概率的乘积，即：

$$P(w_1, ..., w_T) = \prod_{t=1}^{T} P(w_t)$$

然而我们知道语句中的每个词出现的概率都与其前面的词紧密相关, 所以实际上通常用条件概率表示语言模型：

$$\mathrm{P}(\mathrm{w}_1, ..., w_T) = \prod_{t=1}^{T} P(w_t|w_1, ..., w_{t-1})$$

N-gram neural model

在计算语言学中，n-gram是一种重要的文本表示方法，表示一个文本中连续的n个项。基于具体的应用场景，每一项可以是一个字母、单词或者音节。 n-gram模型也是统计语言模型中的一种重要方法，用n-gram训练语言模型时，一般用每个n-gram的历史n-1个词语组成的内容来预测第n个词。

Yoshua Bengio等科学家就于2003年在著名论文 Neural Probabilistic Language Models [1] 中介绍如何学习一个神经元网络表示的词向量模型。文中的神经概率语言模型（Neural Network Language Model，NNLM）通过一个线性映射和一个非线性隐层连接，同时学习了语言模型和词向量，即通过学习大量语料得

我们在上文中已经讲到用条件概率建模语言模型，即一句话中第t个词的概率和该句话的前$t-1$个词相关。可实际上越远的词语其实对该词的影响越小，那么如果考虑一个n-gram, 每个词都只受其前面 n-1 个词的影响，则有：

$$P(w_1, ..., w_T) = \prod_{t=1}^{T} P(w_t | w_{t-1}, w_{t-2}, ..., w_{t-n+1})$$

给定一些真实语料，这些语料中都是有意义的句子，N-gram模型的优化目标则是最大化目标函数:

$$\frac{1}{T}\sum_{t} f(w_t, w_{t-1}, ..., w_{t-n+1}; \theta) + R(\theta)$$

其中$f(w_t, w_{t-1}, \ldots, w_{t-n+1})$表示根据历史n-1个词得到当前词$w_t$的条件概率，$R(\theta)$表示参数正则项。

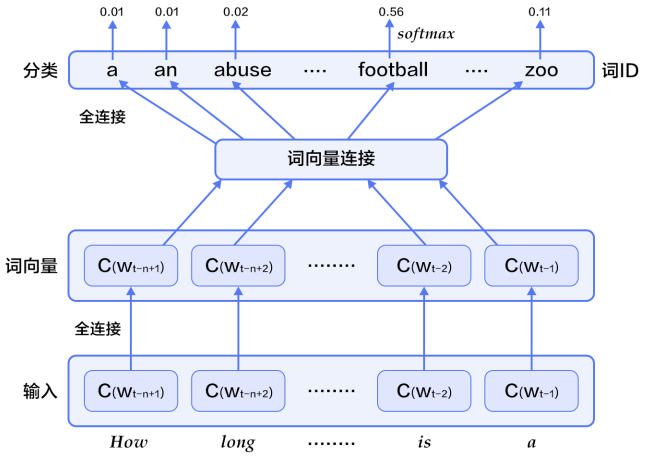

图2. N-gram神经网络模型

图2展示了N-gram神经网络模型，从下往上看，该模型分为以下几个部分：

- 对于每个样本，模型输入$w_{t-n+1}, \ldots w_{t-1}$, 输出句子第t个词在字典中 |V| 个词上的概率分布。

  每个输入词$w_{t-n+1}, \ldots w_{t-1}$首先通过映射矩阵映射到词向量$C(w_{t-n+1}), \ldots C(w_{t-1})$。

- 然后所有词语的词向量拼接成一个大向量，并经过一个非线性映射得到历史词语的隐层表示：

$$g = U\tanh(\theta^T x + b_1) + Wx + b_2$$

其中，x为所有词语的词向量拼接成的大向量，表示文本历史特征；θ、U、b_1、b_2和W分别为词向

- 根据softmax的定义，通过归一化g_i, 生成目标词w_t的概率为：

$$P(w_t | w_1, ..., w_{t-n+1}) = \frac{e^{g_{w_t}}}{\sum_i^{|V|} e^{g_i}}$$

- 整个网络的损失值(cost)为多类分类交叉熵，用公式表示为

<p align="center">

$$\mathrm{J}(\theta) = -\sum_{i=1}^{N}\sum_{k=1}^{|V|} y_k^i \log(\mathrm{softmax}(g_k^i))$$

其中y_k^i表示第i个样本第k类的真实标签(0或1)，$\mathrm{softmax}(g_k^i)$表示第i个样本第k类softmax输出的概率。

Continuous Bag-of-Words model(CBOW)

CBOW模型通过一个词的上下文（各N个词）预测当前词。当N=2时，模型如下图所示：

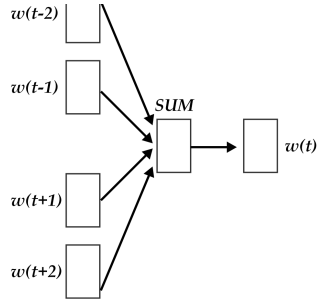

图3. CBOW模型

具体来说，不考虑上下文的词语输入顺序，CBOW是用上下文词语的词向量的均值来预测当前词。即：

$$\text{context} = \frac{x_{t-1} + x_{t-2} + x_{t+1} + x_{t+2}}{4}$$

其中x_t为第t个词的词向量，分类分数（score）向量$z = U * context$，最终的分类y采用softmax，损失函数采用多类分类交叉熵。

Skip-gram model

CBOW的好处是对上下文词语的分布在词向量上进行了平滑，去掉了噪声，因此在小数据集上很有效。而Skip-gram的方法中，用一个词预测其上下文，得到了当前词上下文的很多样本，因此可用于更大的数据集。

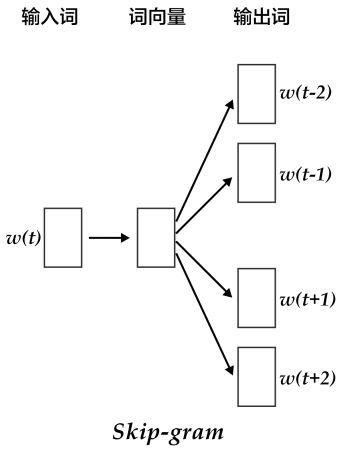

图4. Skip-gram模型

如上图所示，Skip-gram模型的具体做法是，将一个词的词向量映射到$2n$个词的词向量（$2n$表示当前输入词的前后各n个词），然后分别通过softmax得到这$2n$个词的分类损失值之和。

数据准备

数据介绍

本教程使用Penn Treebank（PTB）（经Tomas Mikolov预处理过的版本）数据集。PTB数据集较小，训练速度快，应用于Mikolov的公开语言模型训练工具[2]中。其统计情况如下：

训练数据	验证数据	测试数据
ptb.train.txt	ptb.valid.txt	ptb.test.txt
42068句	3370句	3761句

数据预处理

本章训练的是5-gram模型，表示在PaddlePaddle训练时，每条数据的前4个词用来预测第5个词。PaddlePaddle提供了对应PTB数据集的python包 `paddle.dataset.imikolov` ，自动做数据的下载与预处理，方便大家使用。

如"I have a dream that one day" 一句提供了5条数据：

```
<s> I have a dream
I have a dream that
have a dream that one
a dream that one day
dream that one day <e>
```

最后，每个输入会按其单词次在字典里的位置，转化成整数的索引序列，作为PaddlePaddle的输入。

编程实现

本配置的模型结构如下图所示：

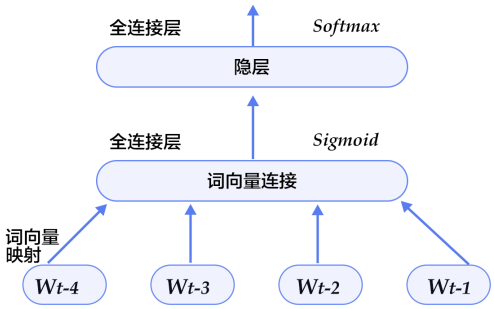

图5. 模型配置中的N-gram神经网络模型

首先，加载所需要的包：

```
import paddle as paddle
import paddle.fluid as fluid
import six
import numpy
import math

from __future__ import print_function
```

然后，定义参数：

```
EMBED_SIZE = 32       # embedding维度
HIDDEN_SIZE = 256     # 隐层大小
N = 5                 # ngram大小，这里固定取5
BATCH_SIZE = 100      # batch大小
PASS_NUM = 100        # 训练轮数

use_cuda = False  # 如果用GPU训练，则设置为True

word_dict = paddle.dataset.imikolov.build_dict()
dict_size = len(word_dict)
```

更大的 BATCH_SIZE 将使得训练更快收敛，但也会消耗更多内存。由于词向量计算规模较大，如果环境允许，请开启使用GPU进行训练，能更快得到结果。 不同于之前的PaddlePaddle v2版本，在新的Fluid版本里，我们不必再手动计算词向量。PaddlePaddle提供了一个内置的方法 `fluid.layers.embedding` ，我们就可以直接用它来构造 N-gram 神经网络。

- 我们来定义我们的 N-gram 神经网络结构。这个结构在训练和预测中都会使用到。因为词向量比较稀疏，我们传入参数 `is_sparse == True` ，可以加速稀疏矩阵的更新。

```
def inference_program(words, is_sparse):

    embed_first = fluid.layers.embedding(
        input=words[0],
        size=[dict_size, EMBED_SIZE],
        dtype='float32',
        is_sparse=is_sparse,
        param_attr='shared_w')
    embed_second = fluid.layers.embedding(
        input=words[1],
```

```
    embed_third = fluid.layers.embedding(
        input=words[2],
        size=[dict_size, EMBED_SIZE],
        dtype='float32',
        is_sparse=is_sparse,
        param_attr='shared_w')
    embed_fourth = fluid.layers.embedding(
        input=words[3],
        size=[dict_size, EMBED_SIZE],
        dtype='float32',
        is_sparse=is_sparse,
        param_attr='shared_w')

    concat_embed = fluid.layers.concat(
        input=[embed_first, embed_second, embed_third, embed_fourth], axis=1)
    hidden1 = fluid.layers.fc(input=concat_embed,
                              size=HIDDEN_SIZE,
                              act='sigmoid')
    predict_word = fluid.layers.fc(input=hidden1, size=dict_size, act='softmax')
    return predict_word
```

- 基于以上的神经网络结构，我们可以如下定义我们的训练方法

```
def train_program(predict_word):
    # 'next_word'的定义必须要在inference_program的声明之后，
    # 否则train program输入数据的顺序就变成了[next_word, firstw, secondw,
    # thirdw, fourthw], 这是不正确的.
    next_word = fluid.layers.data(name='nextw', shape=[1], dtype='int64')
    cost = fluid.layers.cross_entropy(input=predict_word, label=next_word)
    avg_cost = fluid.layers.mean(cost)
    return avg_cost

def optimizer_func():
    return fluid.optimizer.AdagradOptimizer(
        learning_rate=3e-3,
        regularization=fluid.regularizer.L2DecayRegularizer(8e-4))
```

- 现在我们可以开始训练啦。如今的版本较之以前就简单了许多。我们有现成的训练和测试集：`paddle.dataset.imikolov.train()` 和 `paddle.dataset.imikolov.test()` 。两者都会返回一个读取器。在PaddlePaddle中，读取器是一个Python的函数，每次调用，会读取下一条数据。它是一个Python的generator。

`paddle.batch` 会读入一个读取器，然后输出一个批次化了的读取器。我们还可以在训练过程中输出每个步骤，批次的训练情况。

```
def train(if_use_cuda, params_dirname, is_sparse=True):
    place = fluid.CUDAPlace(0) if if_use_cuda else fluid.CPUPlace()

    train_reader = paddle.batch(
        paddle.dataset.imikolov.train(word_dict, N), BATCH_SIZE)
    test_reader = paddle.batch(
        paddle.dataset.imikolov.test(word_dict, N), BATCH_SIZE)

    first_word = fluid.layers.data(name='firstw', shape=[1], dtype='int64')
    second_word = fluid.layers.data(name='secondw', shape=[1], dtype='int64')
    third_word = fluid.layers.data(name='thirdw', shape=[1], dtype='int64')
    forth_word = fluid.layers.data(name='fourthw', shape=[1], dtype='int64')
    next_word = fluid.layers.data(name='nextw', shape=[1], dtype='int64')

    word_list = [first_word, second_word, third_word, forth_word, next_word]
    feed_order = ['firstw', 'secondw', 'thirdw', 'fourthw', 'nextw']

    main_program = fluid.default_main_program()
    star_program = fluid.default_startup_program()

    predict_word = inference_program(word_list, is_sparse)
    avg_cost = train_program(predict_word)
    test_program = main_program.clone(for_test=True)

    sgd_optimizer = optimizer_func()
    sgd_optimizer.minimize(avg_cost)

    exe = fluid.Executor(place)

    def train_test(program, reader):
        count = 0
        feed_var_list = [
            program.global_block().var(var_name) for var_name in feed_order
        ]
        feeder_test = fluid.DataFeeder(feed_list=feed_var_list, place=place)
        test_exe = fluid.Executor(place)
```

```
                program=program,
                feed=feeder_test.feed(test_data),
                fetch_list=[avg_cost])
            accumulated = [
                x[0] + x[1][0] for x in zip(accumulated, avg_cost_np)
            ]
            count += 1
        return [x / count for x in accumulated]

    def train_loop():
        step = 0
        feed_var_list_loop = [
            main_program.global_block().var(var_name) for var_name in feed_order
        ]
        feeder = fluid.DataFeeder(feed_list=feed_var_list_loop, place=place)
        exe.run(star_program)
        for pass_id in range(PASS_NUM):
            for data in train_reader():
                avg_cost_np = exe.run(
                    main_program, feed=feeder.feed(data), fetch_list=[avg_cost])

                if step % 10 == 0:
                    outs = train_test(test_program, test_reader)

                    print("Step %d: Average Cost %f" % (step, outs[0]))

                    # 整个训练过程要花费几个小时，如果平均损失低于5.8，
                    # 我们就认为模型已经达到很好的效果可以停止训练了。
                    # 注意5.8是一个相对较高的值，为了获取更好的模型，可以将
                    # 这里的阈值设为3.5，但训练时间也会更长。
                    if outs[0] < 5.8:
                        if params_dirname is not None:
                            fluid.io.save_inference_model(params_dirname, [
                                'firstw', 'secondw', 'thirdw', 'fourthw'
                            ], [predict_word], exe)
                        return
                step += 1
                if math.isnan(float(avg_cost_np[0])):
                    sys.exit("got NaN loss, training failed.")

        raise AssertionError("Cost is too large {0:2.2}".format(avg_cost_np[0]))

    train_loop()
```

- `train_loop` 将会开始训练。期间打印训练过程的日志如下：

```
Step 0: Average Cost 7.337213
Step 10: Average Cost 6.136128
Step 20: Average Cost 5.766995
...
```

模型应用

在模型训练后，我们可以用它做一些预测。

预测下一个词

我们可以用我们训练过的模型，在得知之前的 N-gram 后，预测下一个词。

```
def infer(use_cuda, params_dirname=None):
    place = fluid.CUDAPlace(0) if use_cuda else fluid.CPUPlace()

    exe = fluid.Executor(place)

    inference_scope = fluid.core.Scope()
    with fluid.scope_guard(inference_scope):
        # 使用fluid.io.load_inference_model获取inference program，
        # feed变量的名称feed_target_names和从scope中fetch的对象fetch_targets
        [inferencer, feed_target_names,
         fetch_targets] = fluid.io.load_inference_model(params_dirname, exe)

        # 设置输入，用四个LoDTensor来表示4个词语。这里每个词都是一个id，
        # 用来查询embedding表获取对应的词向量，因此其形状大小是[1]。
        # recursive_sequence_lengths设置的是基于长度的LoD，因此都应该设为[[1]]
        # 注意recursive_sequence_lengths是列表的列表
        data1 = numpy.asarray([[211]], dtype=numpy.int64)  # 'among'
        data2 = numpy.asarray([[6]], dtype=numpy.int64)  # 'a'
        data3 = numpy.asarray([[96]], dtype=numpy.int64)  # 'group'
        data4 = numpy.asarray([[4]], dtype=numpy.int64)  # 'of'
```

```
        third_word = fluid.create_lod_tensor(data3, lod, place)
        fourth_word = fluid.create_lod_tensor(data4, lod, place)

        assert feed_target_names[0] == 'firstw'
        assert feed_target_names[1] == 'secondw'
        assert feed_target_names[2] == 'thirdw'
        assert feed_target_names[3] == 'fourthw'

        # 构造feed词典 {feed_target_name: feed_target_data}
        # 预测结果包含在results之中
        results = exe.run(
            inferencer,
            feed={
                feed_target_names[0]: first_word,
                feed_target_names[1]: second_word,
                feed_target_names[2]: third_word,
                feed_target_names[3]: fourth_word
            },
            fetch_list=fetch_targets,
            return_numpy=False)

        print(numpy.array(results[0]))
        most_possible_word_index = numpy.argmax(results[0])
        print(most_possible_word_index)
        print([
            key for key, value in six.iteritems(word_dict)
            if value == most_possible_word_index
        ][0])
```

由于词向量矩阵本身比较稀疏，训练的过程如果要达到一定的精度耗时会比较长。为了能简单看到效果，教程只设置了经过很少的训练就结束并得到如下的预测。我们的模型预测 `among a group of` 的下一个词是 `the` 。这比较符合文法规律。如果我们训练时间更长，比如几个小时，那么我们会得到的下一个预测是 `workers` 。预测输出的格式如下所示:

```
[[0.03768077 0.03463154 0.00018074 ... 0.00022283 0.00029888 0.02967956]]
0
the
```

其中第一行表示预测词在词典上的概率分布，第二行表示概率最大的词对应的id，第三行表示概率最大的词。

整个程序的入口很简单：

```
def main(use_cuda, is_sparse):
    if use_cuda and not fluid.core.is_compiled_with_cuda():
        return

    params_dirname = "word2vec.inference.model"

    train(
        if_use_cuda=use_cuda,
        params_dirname=params_dirname,
        is_sparse=is_sparse)

    infer(use_cuda=use_cuda, params_dirname=params_dirname)

main(use_cuda=use_cuda, is_sparse=True)
```

总结

本章中，我们介绍了词向量、语言模型和词向量的关系、以及如何通过训练神经网络模型获得词向量。在信息检索中，我们可以根据向量间的余弦夹角，来判断query和文档关键词这二者间的相关性。在句法分析和语义分析中，训练好的词向量可以用来初始化模型，以得到更好的效果。在文档分类中，有了词向量之后，可以用聚类的方法将文档中同义词进行分组，也可以用 N-gram 来预测下一个词。希望大家在本章后能够自行运用词向量进行相关领域的研究。

参考文献

1. Bengio Y, Ducharme R, Vincent P, et al. A neural probabilistic language model[J]. journal of machine learning research, 2003, 3(Feb): 1137-1155.
2. Mikolov T, Kombrink S, Deoras A, et al. Rnnlm-recurrent neural network language modeling toolkit[C]//Proc. of the 2011 ASRU Workshop. 2011: 196-201.

2579-2605.
5. https://en.wikipedia.org/wiki/Singular_value_decomposition