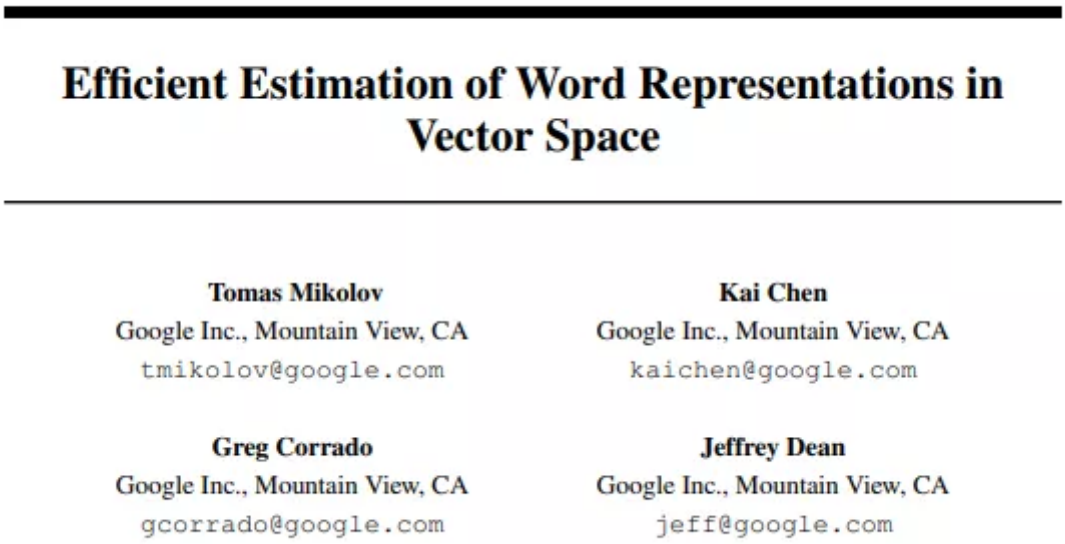
NLP重铸篇之Word2vec

原创 错乱空时 NLP杂货铺 1周前

收录于话题
#NLP 35 #word2vec 1 #论文复现 1



Efficient Estimation of Word Representations in Vector Space

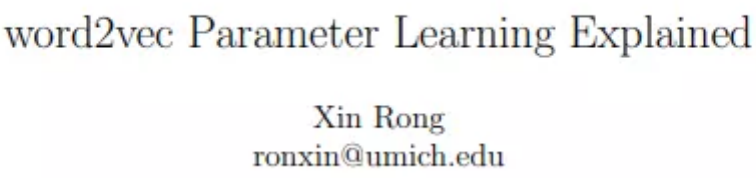
Tomas Mikolov
Google Inc., Mountain View, CA
tmikolov@google.com

Kai Chen
Google Inc., Mountain View, CA
kaichen@google.com

Greg Corrado
Google Inc., Mountain View, CA
gcorrado@google.com

Jeffrey Dean
Google Inc., Mountain View, CA
jeff@google.com

论文标题：Efficient Estimation of Word Representations in Vector Space
论文链接：https://arxiv.org/pdf/1301.3781.pdf
复现代码地址：
https://github.com/wellinxu/nlp_store/blob/master/papers/word2vec.py



word2vec Parameter Learning Explained

Xin Rong
ronxin@umich.edu

论文标题：word2vec Parameter Learning Explained
论文链接：https://arxiv.org/pdf/1411.2738.pdf

word2vec原论文讲得比较简单，几乎没有细节，本文会根据另一篇论文【word2vec Parameter Learning Explained】，来详细介绍两种加速方法。本文使用python+tensorflow2.0来复现word2vec模型，所以模型中的反向梯度计算与参数优化更新，都是使用的tf中的自动求导与优化器实现，也因此本文中只涉及到word2vec的两种结构（CBOW与Skip-gram）及两种加速方式（Huffman树-层次softmax和负采样）从输入到loss的前向计算，完整代码已开源，具体请查看https://github.com/wellinxu/nlp_store。

重铸系列会分享论文的解析与复现，主要是一些经典论文以及前沿论文，但知识还是原汁原味的好，支持大家多看原论文。分享内容主要来自于原论文，会有些整理与删减，以及个人理解与应用等等，其中涉及到的算法复现都会开源在：https://github.com/wellinxu/nlp_store ，更多内容关注知乎专栏（或微信公众号）：NLP杂货铺。

- 参考

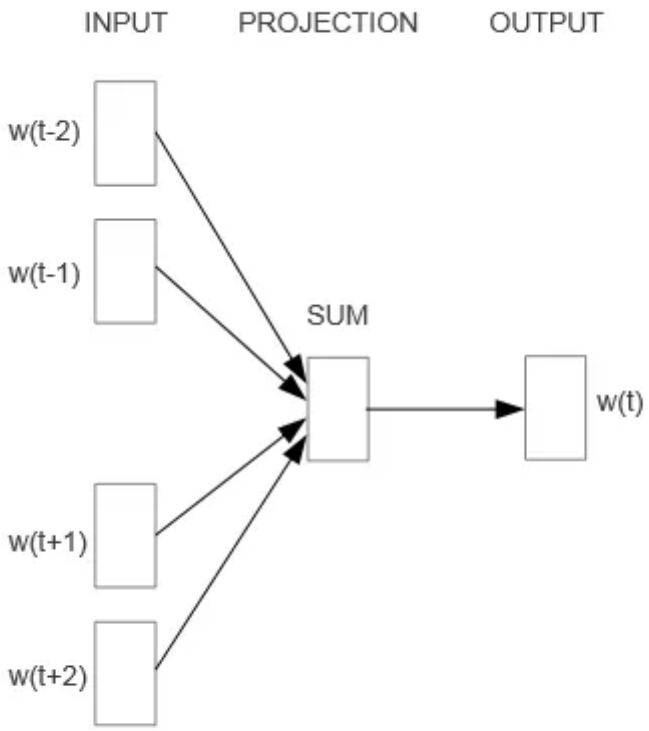
引言

word2vec的目标是从十亿量级的文章跟百万量级的词汇中学习出高质量的词向量表示，实验结果表明，其向量可以达到类似“江苏-南京+杭州≈浙江”的效果。

模型结构

在word2vec之前，已经有不少关于词的连续表示模型提出来，比如LSA（Latent Semantic Analysis）跟LDA（ Latent Dirichlet Allocation）。跟LAS相比，word2vec效果更好，跟LDA相比，word2vec的计算速度更快。
论文中提出了两种新的模型结构：CBOW、Skip-gram，跟NNLM（Feedforward Neural Net Language Model）相比，word2vec为了追求更简单的模型结构，删除了中间的非线性隐藏层，尽管这样可能会让词向量没有NNLM的更加精确，但这可以让模型在更大的数据集上高效训练。
两种结构都是利用中心词跟上下文互为输入与输出，上下文就是中心词前后的n个词，实验表明，提高上下文的范围（即n的大小），可以提高词向量的质量，但这也加大了计算复杂度。

CBOW结构



CBOW

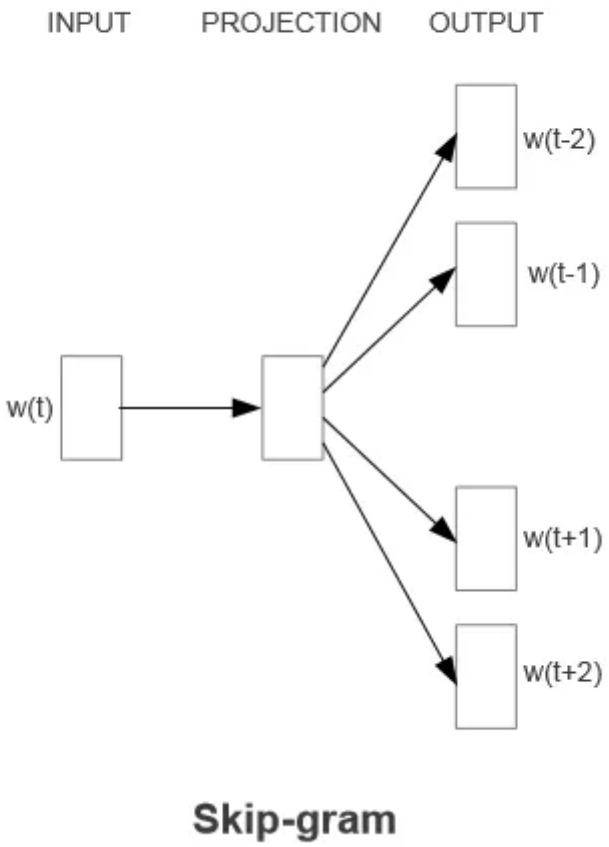
第一种模型结构是CBOW（Continuous Bag-of-Words Model），其思想是利用中心词前面n个词跟后面n个词来预测中心词（n可自定义），如上图所示，用的是前面2个词跟后面2个词预测当前词，输入层直接求和变成投影层，没有其他任何操作，然后直接预测中心词。整体公式可直接表示为：

$$projection = \sum_i input_i$$

$$output = f(projection, \theta)$$

其中f表示将projection映射为output的变换函数，θ是其中的参数，f的具体方式后续会讲解。

Skip-gram结构



Skip-gram

第二种模型结构是Skip-gram（Continuous Skip-gram Model），其思想与CBOW相反：利用中心词预测中心词上下文的n个词（n可自定义），如上图所示，利用中心词预测上下文各2个词，输入层直接等价变化到投影层，然后预测上下文的词。整体公式可以直接表示为：

$$projection = input$$
$$output = f(projection, \theta)$$

本文复现过程中，将Skip-gram结构视为多标签预测问题，word2vec的源码中应该是将每个输出拆开（参考【2】得出的结论，并未真正看过源码），组成多个单分类问题。从过程上来说，这两种方式更新参数的顺序会有差别，有些类似于批梯度下降与随机梯度下降的区别，从工程上来说，word2vec源码中是简化了问题，处理更简单。

softmax的loss计算

如果用softmax来取代上面式子中的f函数，那么ouput的计算方式可具体为：

$$output_i = \frac{\exp(\theta_i * projection^T)}{\sum_j^N (\exp(\theta_j * projection^T))}, i = 1, 2, \ldots, N$$

其中$output_i$表示预测为第i个词的概率，θ_i表示第i个词的输出权重向量，N表示词表大小。
这是一个比较标准的分类问题，所以使用负对数似然概率或者交叉熵来计算其loss，具体的CBOW结构的loss可表示为：

$$loss = -\log(output_j)$$

其中j表示中心词的索引，$output_j$表示预测是真实中心词的概率。

Skip-gram结构的loss则可表示为：

$$loss = -\sum_{j}^{M}\log(output_j), j = 1, 2, \ldots, M$$

其中M是上下文的数量，$output_j$表示预测是第j个上下文词的概率。

根据上面两种模型结构以及loss计算，可以完成相应代码，因为模型不关心输出，只关心词权重的更新，所以会在模型内部直接计算loss，具体代码如下：

```
import tensorflow as tf
import numpy as np
import random
import time

class BaseWord2vecModel(tf.keras.models.Model):
    # 当前的实现没有batch维度，是一个样本一个样本进行训练
    def __init__(self, voc_size, emb_dim, is_huffman=True, is_negative=False):
        super(BaseWord2vecModel, self).__init__()
        self.voc_size = voc_size
        self.is_huffman = is_huffman
        self.is_negative = is_negative
        self.embedding = tf.keras.layers.Embedding(voc_size, emb_dim)
        self.layer_norm = tf.keras.layers.LayerNormalization(
            name="layer_norm", axis=-1, epsilon=1e-12, dtype=tf.float32)
        if not self.is_huffman and not is_negative:
            # 不使用huffman树也不使用负采样，所有词的输出参数
            self.output_weight = self.add_weight(shape=(voc_size, emb_dim),
                                                 initializer=tf.zeros_initializer, trainable=True
            self.softmax = tf.keras.layers.Softmax()
        if self.is_huffman:
            # 所有节点的参数，huffman树压缩为数组的时候，保留了所有叶子节点，所以数组长度为2*voc_size
            # 也可以选择只保留非叶子节点，这样长度可减半
            self.huffman_params = tf.keras.layers.Embedding(2*voc_size, emb_dim,
                                                            embeddings_initializer=tf.keras.initi
            self.huffman_choice = tf.keras.layers.Embedding(2, 1, weights=(np.array([[-1], [1]]),
        if self.is_negative:
            # 负采样时，每个词的输出参数
            self.negative_params = tf.keras.layers.Embedding(voc_size, emb_dim,
                                                             embeddings_initializer=tf.keras.init

    def call(self, inputs, training=None, mask=None):
        # x: [context_len]
        # huffman_label: [label_size, code_len]
        # huffman_index: [label_size, code_len]
        # y : [label_size]
```

```
        # negative_index: [negatuve_num]
        x, huffman_label, huffman_index, y, negative_index = inputs
        # skip-gram的context_len就是1
        x = self.embedding(x)    # [context_len, emb_dim]
        x = self.layer_norm(x)    # 层标准化，自己添加非模型原始结构
        # skip-gram单个求和就是本身
        x = tf.reduce_sum(x, axis=-2)    # [emb_dim]

        loss = 0
        if not self.is_huffman and not self.is_negative:
            # 不使用huffman树也不使用负采样，则使用原始softmax
            output = tf.einsum("ve,e->v", self.output_weight, x)    # [voc_size]
            output = self.softmax(output)
            y_index = tf.one_hot(y, self.voc_size)    # [label_size, voc_size]
            y_index = tf.reduce_sum(y_index, axis=0)    # [voc_size]
            l = tf.einsum("a,a->a", output, y_index)    # [voc_size]
            loss -= tf.reduce_sum(l)
```

从上面计算公式可以看出，当词表量级特别大的时候（百万级别），$output_i$的分母计算以及后续反向传播更新所有θ_i的计算将会非常耗时，这种昂贵的计算代价使得在大语料或者大词表情况下训练变得不可能。要解决这个问题，一个直觉的方法是限制每个训练样本必须更新的输出向量的数量，后续会介绍两种方式来实现这一点，分别是层次softmax与负采样。

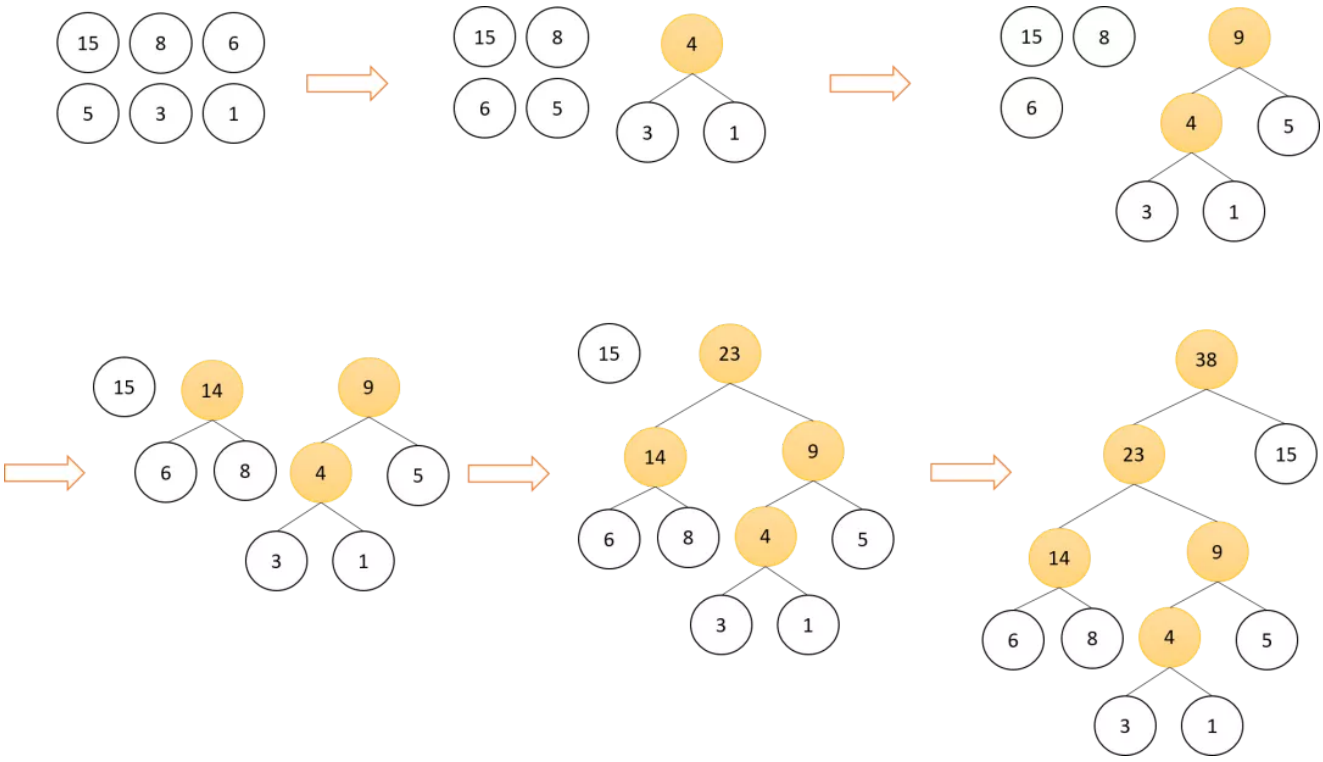
Huffman树——层次softmax

层次softmax是一种高效计算softmax的方法，其使用二叉树来表示词表中的所有词，每一个词都必须是树的叶子结点，对于每一个结点，都存在唯一的路径从根结点到当前叶子结点，该路径就被用来估计此叶子结点表示的词出现的概率。理论上说，可以使用任何形式的树来计算层次softmax，word2vec里面使用的是二叉Huffman树来进行训练。因为huffman树中权重越高的结点越靠近根结点，这样频率高的词的路径就越短，计算的次数也就越少，从而可以进一步提高训练速度。

Huffman树的构建

给定n个结点，每个结点都有一个权重，构造一棵二叉树，如果它的带权路径长度最小，则称为最优二叉树，也称为Huffman树。给定n个值$\{w_1, w_2, \ldots, w_n\}$作为n个结点的权重，可以通过下面方法构造huffman树：

1. 将$\{w_1, w_2, \ldots, w_n\}$看成是有n棵树的森林（每棵树只有一个结点）。
2. 在森林中选出两个根节点权重最小的树合并，分别作为新树的左右子树，新树根结点的权重为左右子树根结点权重之和。
3. 删除森林中选取的两棵树，并将新树添加到森林。
4. 重复上面两步操作，直到森林里就剩一棵树，该树就是huffman树。



如上图所示，显示了一个简单huffman树的构造过程，在本文的实现中，将叶子结点优先放在右子树上，但这并不是必要的，也可以将权重大（或小）的结点放在右子树上。
为了实现其构造过程，我们需要先定义结点类，如下所示：

```
class Node(object):
    def __init__(self, key, value):
        self.key = key    # 本文代码里huffman中，非叶子节点都为None
        self.value = value    # 权重
        # 编码，即重跟节点走到本节点的方向，0表示左子树，1表示右子树
        # 010表示跟节点->左子树->右子树->左子树（本节点）
        self.code = []    # 记录当前节点在整个huffman树中的编码
        self.index = 0    # 第几个节点，压缩为数组用，即为该节点在数组型的huffman树种的索引位置
        self.left = None
        self.right = None

    def combine(self, node):
        # 两棵树（两个节点）合并成一个新树，叶子节点放在新树的右子树上
        new_node = Node(None, self.value + node.value)
```

```
        if self.key is not None:
            new_node.right = self
            new_node.left = node
        else:
            new_node.right = node
            new_node.left = self
        return new_node
```

Node类中定义了权重（value）、词（key）、编码（code）、索引位置（index）以及左右结点，此外还提供了combine方法，用来合并两棵子树。定义好结点，就可以通过以下方式来构建huffman树：

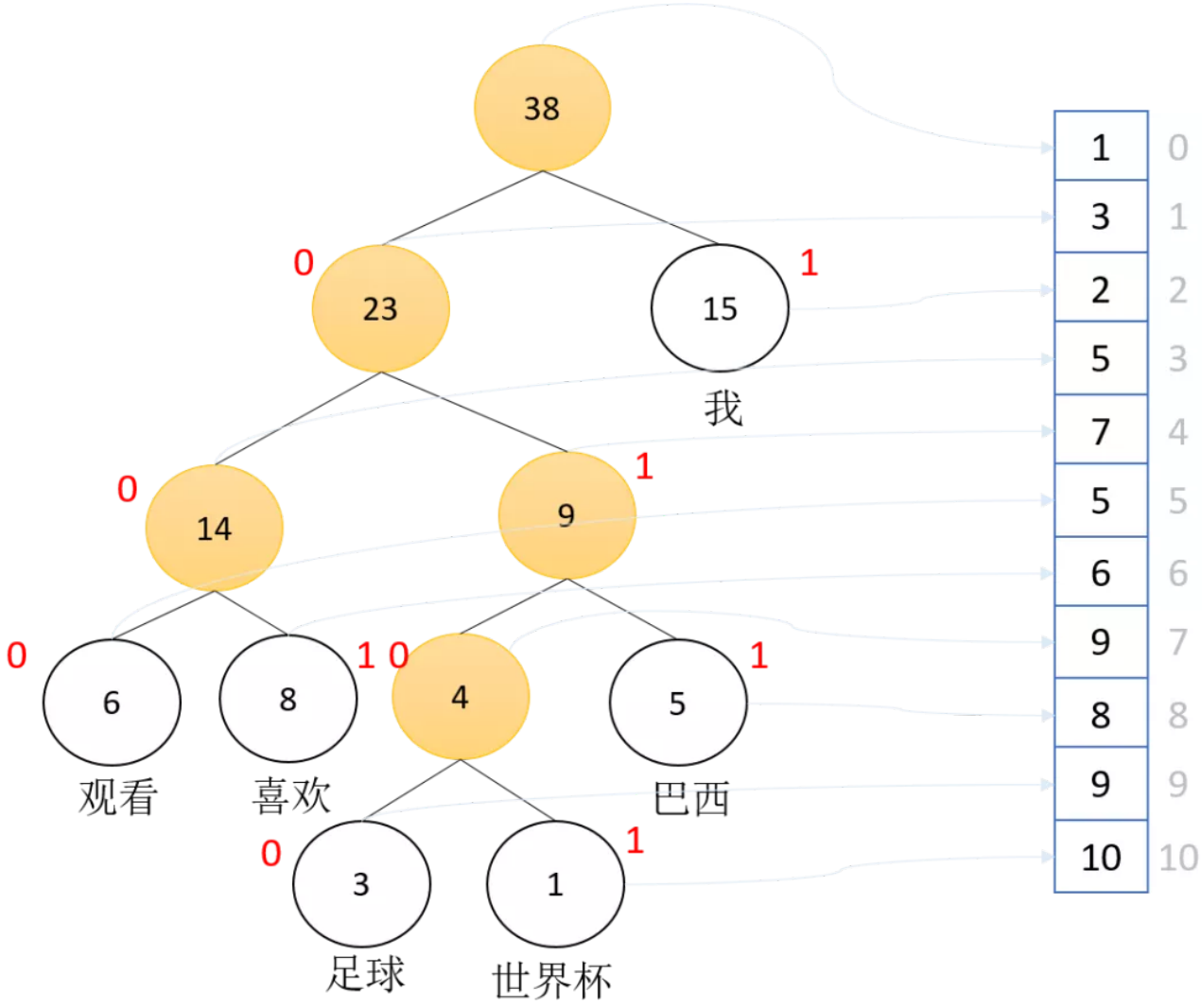
```
class HuffmanTree(object):
    # 用一个数组表示huffman树的所有非叶子节点
    # 用word_code_map表记录根到每个叶子节点的编码
    def __init__(self, words):
        start = time.time()
        words.sort()  # 根据频率排序
        self.words = words    # [(value:频次, key:词)]，由小到大排序
        self.word_code_map = {}    # 词在huffman树中的编码映射
        self.nodes_list = []     # 压缩为数组的huffman树
        self.build_huffman_tree()
        print("build huffman tree end,time:", time.time()-start)

    def build_huffman_tree(self):
        # 构建huffman树
        # 每个元素都构成单节点的树，并按照权重重大到小排列
        # 合并权重最小的两个子树，并以权重和作为新树的权重
        # 将新树按照权重大小插入到序列中
        # 重复上述两步，直到只剩一棵树
        nodes = [Node(key, value) for value, key in self.words]
        while len(nodes) > 1:
            a_node = nodes.pop(0)
            b_node = nodes.pop(0)
            new_node = a_node.combine(b_node)
            left, right = 0, len(nodes)
            l = right - 1
            i = right >> 1
            while True:
                if nodes and nodes[i].value >= new_node.value:
                    if i == 0 or nodes[i-1].value < new_node.value:
                        nodes.insert(i, new_node)
                        break
                    else:
```

```
                right = i
                i = (left+right) >> 1
        else:
            if i == 0 or i == l:
                nodes.insert(i, new_node)
                break
            left = i
            i = (left+right) >> 1
```

其中输入words是以(词频，词)为元素的list，并以词频的大小由小到大排好序。

Huffman树压缩为数组

构建完huffman树，可以根据其设计二进制前缀编码，也称Huffman编码，下图展示了一个huffman树的编码示例，左子树为0右子树为1（这个可自由设定），如“巴西”就可以编码为“011”。



在后续计算过程中，不仅需要知道词的huffman编码，还需要知道该词经过了哪些结点，为了方便根据编码得知结点，我们将树形结构压缩为数组型结构（并不是必要的，也可以通过原始huffman树，获取该编码经过的结点），如上图所示，huffman树按照层次遍历的顺序跟数组的索引一一对应，数组中的值表示当前结点左结点的索引值，如果数组的值跟索引相同，则表

示该结点是叶子节点（其实也可以只将非叶子结点压缩到数组里面，因为后续计算中不需要叶子结点）。

根据上述需求，我们对huffman树进行一次层次遍历就可以实现，具体代码如下：

```
# 将树压缩为数组
# 数组中每一个元素都是树种的一个节点，数组中的值表示该节点的左节点的位置，如果值跟索引一样大，表示
# 跟节点 root.index = 0
# 叶子节点 nodes_list[node.index] = node.index
# 非叶子节点 nodes_list[node.index] = node.left.index
# 非叶子节点 nodes_list[node.index] + 1 = node.right.index
stack = [nodes[0]]
while stack:
    new_stack = []
    for node in stack:
        node.index = len(self.nodes_list)
        if node.key is not None:   # 叶子结点
            self.word_code_map[node.key] = node.code  # 保存编码
            self.nodes_list.append(node)  # 在数组中添加叶子结点本身
        if node.right:
            node.right.code = node.code + [1]
            new_stack.append(node.right)
        if node.left:
            # 在数组相应位置添加该结点的左结点
            self.nodes_list.append(node.left)
            node.left.code = node.code + [0]
            new_stack.append(node.left)
    stack = new_stack
self.nodes_list = [node.index for node in self.nodes_list]
```

将huffman树压缩为数组后，就可以根据以下公式计算每个编码所经历的结点：

$$node_1 = 0, \text{所有路径的第一个结点都是根结点}$$
$$node_i = huffman\_node\_list[node_{i-1}] + code_{i-1}, i = 2, 3, \dots, l$$

其中l表示编码的长度，$code_i$表示第i个编码值，$node_i$表示经过的第i个结点的位置。比如“巴西”的编码值是“011”，则可以计算第一个结点位置是0，第二个是$node_2 = 1 + 0 = 1$，第三个是$node_3 = 3 + 1 = 4$，所以“巴西”这个词经过的内部结点位置依次为0-1-4（不算巴西本身这个结点）。

根据上面公式，我们可以构建通过huffman树处理的训练数据：

```
# 构建huffman数据
if self.is_huffman:
```

```
        huffman_tree = HuffmanTree(self.words)      # 根据词与词频，构建huffman树
        for labels in self.ys:
            tlabels = []
            for label in labels:
                tlabels.append(np.array(huffman_tree.word_code_map[label]))
            index = []
            for tlabel in tlabels:
                tem_index = [0]
                for l in tlabel[:-1]:
                    ind = huffman_tree.nodes_list[tem_index[-1]] + l
                    tem_index.append(ind)
                index.append(np.array(tem_index))
            self.huffman_label.append(tlabels)  # 获取标签词在huffman树中的编码
            self.huffman_index.append(index)    # 获取标签词在huffman树中的编码上对应的所有非叶子
```

Huffman树loss计算

Huffman树中每个词出现的概率，是将该词到根结点路径上的每个结点出现的概率相乘，具体公式如下：

$$p_i = sigmoid(sign(code_i) * \theta_{node_i} * projection^T)$$
$$P = \prod_i p_i, i = 1, 2, \ldots, l-1$$
$$sign(x) = \begin{cases} -1 & , x = 0 \\ 1 & , x = 1 \end{cases}$$

其中l是编码长度，$code_i$是第i个编码值，为0时表示是往左结点走，此时用$sigmoid(-x)$计算概率，为1时则用$sigmoid(x)$来计算概率；$projection$是上面两种结构的投影层结果，$node_i$表示进过的第i个结点的位置，θ_j表示第j个结点的参数。
跟一般softmax相比，原来需要计算与更新N(词表大小)次参数，现在只需要计算与更新$l-1$次，l的大小跟$\log(N)$接近，从而大大提高了计算效率。
上面已经得到单个词似然函数，如果是CBOW模型结构，那Huffman的loss则可以取其负对数：

$$loss = -\log(P) = -\sum_i \log(p_i), i = 1, 2, \ldots, l-1$$

当模型结构是Skip-gram结构时，会有多个输出，此时的Huffman的loss则应该为：

$$loss = -\log(\prod_j P^j) = -\sum_j \sum_i \log(p_i^j), i = 1, 2, \ldots, l-1; j = 1, 2, \ldots, O$$

其中O表示输出的个数，P^j表示第j个输出词的似然概率。
根据公式，则huffman相关代码为：

```
    # huffman树loss计算
```

```
        if self.is_huffman:
            for tem_label, tem_index in zip(huffman_label, huffman_index):
                # 获取huffman树编码上的各个节点参数
                huffman_param = self.huffman_params(tem_index)    # [code_len, emb_dim]
                # 各节点参数与x点积
                huffman_x = tf.einsum("ab,b->a", huffman_param, x)    # [code_len]
                # 获取每个节点是左节点还是右节点
                tem_label = tf.squeeze(self.huffman_choice(tem_label), axis=-1)    # [code_len]
                # 左节点：sigmoid(-WX),右节点sigmoid(WX)
                l = tf.sigmoid(tf.einsum("a,a->a", huffman_x, tem_label))    # [code_len]
                l = tf.math.log(l)
                loss -= tf.reduce_sum(l)
```

负采样

负采样的思想比层次softmax更加直接：为了解决softmax要计算和更新的参数太多的问题，负采样每次只计算和更新几个参数。也就是说，原来每个样本进行训练的时候，需要从所有词汇中选出某个或某几个词，现在只需要从某个小的词集里面选出某个或者某几个词，因为小词集的数量远小于原本词表数量，所以计算量与需要更新的参数都小很多，从而大大提高训练速度。

采样权重调整

根据负采样的思想，预测的词肯定需要在小词集中出现，然后只需要从其他词中抽取一些词作为负样本（负采样的由来）。显而易见的，可以通过词频计算词的分布概率从而进行抽样，但为了降低词频过高的词被抽的概率，以及提高词频过低的词被抽的概率，word2vec中将频率进行了0.75次幂运算，然后计算词的分布概率。相关代码，可参考：

```
    def build_word_dict(self):
        # 构建词典，获取词频
        self.words = []    # (频率，id)
        self.word_map = {}    # {词:id}
        word_num_map = {}    # 频率
        for doc in self.docs:
            if len(doc) < self.windows: continue
            for word in doc:
                if word not in self.word_map.keys():
                    self.word_map[word] = len(self.word_map)
                    self.words.append(word)
```

```
            word_num_map[word] = word_num_map.get(word, 0) + 1
        # 词频设为原本的0.75次方，根据词频进行负采样的时候，可以降低高词频的概率，提高低词频的概率（高词频
        word_num_map = {k:np.power(v, 0.75) for k, v in word_num_map.items()}
        num = sum(word_num_map.values())
        word_num_map = {k: v/num for k, v in word_num_map.items()}

        self.words = [(word_num_map[w], self.word_map[w]) for w in self.words]
        self.words.sort()    # 根据频率排序
        self.voc_size = len(self.words)    # 词表大小
        self.id_word_map = {v: k for k, v in self.word_map.items()}  # {id:词}
```

有了每个词的概率，可以将每个词映射到0-1之间的一段范围之内，然后生成随机数，根据其落在的空间判断抽取的词，遇到正例的词则跳过，不断重复，直到抽取到一定的负样本数量，具体代码如下：

```
        # 构建负采样数据
        if self.is_negative:
            # 如果"我 爱 你 啊"出现的概率分别是0.4,0.2,0.3,,0.1,
            # 那么word_end_p就为[0.4,0.6,0.9, 1.0],即[0.4,0.4+0.2,0.4+0.2+0.3,0.4+0.2+0.3+0.1]
            word_end_p = [self.words[0][0]]    # 每个词出现的概率段
            for i in range(1, self.voc_size):
                word_end_p.append(word_end_p[-1]+self.words[i][0])
            # 为每一条训练数据抽取负样本
            for y in self.ys:
                indexs = []
                while len(indexs) < self.negative_num * len(y):
                    index = self._binary_search(random.random(), word_end_p, 0, self.voc_size-1)
                    # 随机抽取一个词，不能再标签中也不能已经被抽到
                    if index not in indexs and index not in y:
                        indexs.append(index)
                self.negative_index.append(np.array(indexs))

    def _binary_search(self, n, nums, start, end):
        # 二分查找，查找n在nums[start:end]数组的那个位置
        if start == end: return end
        mid = (start+end) >> 1
        if nums[mid] >= n:
            return self._binary_search(n, nums, start, mid)
        return self._binary_search(n, nums, mid+1, end)
```

负采样loss计算

负采样的基本思想是减少更新的参数数量，具体操作又是怎样的呢？简单的说就是，每一个样本，对于所有抽样出来的词（包括正例词跟负例词），都做一个二分类，正例词计算正例的概率，负例词计算负例的概率，目标是使得所有概率的乘积最大。更具体的解释及相关公式可以参考【3】。

这样就可以得到输出概率：

$$P = \prod_{i \in O} p_i = \prod_{i \in O} \{\sigma(\theta_i * projection^T) * \prod_{j \in W_{neg}^i} \sigma(-\theta_j * projection^T)\}$$

其中O表示输出词（正例词）的索引集合，W_{neg}^i表示第i个输出词的负例词索引集合。当模型结构是CBOW时，输出词只有一个，当模型结构是Skip-gram时，才是多个。

这时候可以计算loss：

$$loss = -\log(P) = -\sum_i \log(p_i)$$

根据公式，相关代码如下：

```
# 负采样loss计算
if self.is_negative:
    y_param = self.negative_params(y)    # [label_size, emb_dim]
    negative_param = self.negative_params(negative_index)    # [negative_num, emb_dim]
    y_dot = tf.einsum("ab,b->a", y_param, x)    # [label_size]
    y_p = tf.math.log(tf.sigmoid(y_dot))    # [label_size]
    negative_dot = tf.einsum("ab,b->a", negative_param, x)    # [negative_num]
    negative_p = tf.math.log(tf.sigmoid(-negative_dot))    # [negative_num]
    l = tf.reduce_sum(y_p) + tf.reduce_sum(negative_p)
    loss -= l
```

> 以下引用为错误思想
>
> 根据负采样思想，可以得到每个正例样本输出的概率：
>
> $$output_i = \frac{\exp(\theta_i * projection^T)}{\sum_j^{W_{small}} \theta_j * projection^T}$$
>
> 其中$output_i$表示索引为i的输出词的预测概率，W_{small}表示所有正例词跟采样得到的负例词索引集合，$projection$是上面两种结构的投影层结果。
>
> 此时，根据负对数似然或者交叉熵可以计算loss函数，当模型是CBOW结构时：
>
> $$loss = -\log(output_y)$$
>
> 其中$output_y$表示输出词的概率。
>
> 当模型是Skip-gram结构时：
>
> $$loss = -\sum_j^O \log(output_j)$$

其中$output_j$表示第j个输出词的概率，O表示输出的个数。
根据公式，相关代码如下：

```
        # 负采样loss计算
        if self.is_negative:
            y_param = self.negative_params(y)    # [label_size, emb_dim]
            negative_param = self.negative_params(negative_index)    # [negative_num, emb_di
            y_dot = tf.einsum("ab,b->a", y_param, x)    # [label_size]
            y_exp = tf.exp(y_dot)    # [label_size]
            negative_dot = tf.einsum("ab,b->a", negative_param, x)    # [negative_num]
            negative_exp = tf.exp(negative_dot)    # [negative_num]
            y_sum = tf.reduce_sum(y_exp)    # 分子
            negative_sum = tf.reduce_sum(negative_exp)    # 负样本的分母
            loss -= tf.math.log(y_sum/(y_sum+negative_sum))
```

模型训练

上面已经讲述了整个word2vec模型的前向传播环节，本次模型复现使用的是tensorflow2的框架，所以反向传播以及参数更新过程都使用的是框架的自动求导等功能，本文中也不会讲述反向传播相关原理及公式。模型整体训练代码如下：

```
class Word2vec(object):
    def __init__(self, docs=None, emb_dim=100, windows=5, negative_num=10, is_cbow=True, is_huffma
        self.docs = docs    # [[我 是 一段 文本],[这是 第二段 文本]]
        self.windows = windows    # 窗口长度
        self.emb_dim = emb_dim    # 词向量维度
        self.is_cbow = is_cbow    # 是否使用CBOW模式，False则使用SG模式
        self.is_huffman = is_huffman    # 是否使用huffman树
        self.is_negative = is_negative    # 是否使用负采样
        self.huffman_label = []    # huffman数据的标签，判断每次选择左子树还是右子树
        self.huffman_index = []    # huffman数据的编码，用来获取编码上节点的权重
        self.negative_index = []    # 负采样的词索引
        self.negative_num = negative_num    # 负采样数量
        self.epochs = epochs    # 训练轮次
        self.save_path = save_path    # 模型保存路径
        if docs:  # 训练模型
            self.build_word_dict()    # 构建词典，获取词频
```

```
            self.create_train_data()    # 创建训练数据
            self.train()    # 进行训练
            if self.save_path:
                self.save_txt(self.save_path)    # 保存词向量
        elif self.save_path:  # 直接加载词向量
            self.load_txt(self.save_path)

    def train(self):
        sample_num = len(self.xs)    # 样本数量
        optimizer = tf.optimizers.SGD(0.025)    # 优化器
        # 基础word2vec模型
        self.model = BaseWord2vecModel(self.voc_size, self.emb_dim, self.is_huffman, self.is_negat
        # 模型训练
        for epoch in range(self.epochs):
            print("start epoch %d" % epoch)
            i = 0
            for inputs in zip(self.xs, self.huffman_label, self.huffman_index,self.ys, self.negat:
                with tf.GradientTape() as tape:
                    loss = self.model(inputs)
                grads = tape.gradient(loss, self.model.trainable_variables)
                optimizer.apply_gradients(zip(grads, self.model.trainable_variables))
                if i % 1000 == 0:
                    print("-%s->%d/%d" % ("-" * (i * 100 // sample_num), i, sample_num))
                i += 1
        # 获取词向量
        self.word_embeddings = self.model.embedding.embeddings.numpy()
        norm = np.expand_dims(np.linalg.norm(self.word_embeddings, axis=1), axis=1)
        self.word_embeddings /= norm    # 归一化
```

论文结果

如下表所示，论文中设置了5种语义问题与9种句法问题来测试向量效果，通过类似vec(big)-vec(bigger)≈vec(small)-vec(smaller)的方式来判断。

Type of relationship	Word Pair 1		Word Pair 2	
Common capital city	Athens	Greece	Oslo	Norway
All capital cities	Astana	Kazakhstan	Harare	Zimbabwe
Currency	Angola	kwanza	Iran	rial
City-in-state	Chicago	Illinois	Stockton	California
Man-Woman	brother	sister	grandson	granddaughter
Adjective to adverb	apparent	apparently	rapid	rapidly
Opposite	possibly	impossibly	ethical	unethical
Comparative	great	greater	tough	tougher
Superlative	easy	easiest	lucky	luckiest
Present Participle	think	thinking	read	reading
Nationality adjective	Switzerland	Swiss	Cambodia	Cambodian
Past tense	walking	walked	swimming	swam
Plural nouns	mouse	mice	dollar	dollars
Plural verbs	work	works	speak	speaks

测试结果如下图所示，在相同数据上训练的640维词向量，CBOW在句法层面上表现得较好，Skip-gram在语义层面上表现得更好，总体来说还是Skip-gram更好些。

Model Architecture	Semantic-Syntactic Word Relationship test set		MSR Word Relatedness Test Set [20]
	Semantic Accuracy [%]	Syntactic Accuracy [%]	
RNNLM	9	36	35
NNLM	23	53	47
CBOW	24	64	61
Skip-gram	55	59	56

论文也对比了训练语料、向量维度以及训练轮次对结果的影响，实验结果表明，训练语料越多效果更好，600维的结果比300维的结果更优，3轮的训练结果比1轮的训练结果更好。

论文之外

word2vec是2013年提出的模型，限于当时的算力，在大规模的语料上进行训练，确实需要更简单的模型以及一些加速训练方式；之后提出的fasttext在模型结构上可以说跟word2vec一样，主要区别在输入上，fasttext增加了词的形态特征；基于算力的提升，最近的预训练模型参数越来越大，比如BERT、XLNET，甚至有GPT3这种庞然大物。

再反过来看word2vec，是不是用现在的算力训练，就不需要层次softmax或者负采样的加速方式了呢？这还是要分情况讨论，虽然说像BERT这样的模型，都是直接用的softmax进行计算，这是因为训练BERT的人（机构、公司）具有远超普通情况下的算力，而且BERT这种输出维度

也只有几万大小，词向量的输出维度则会有百万大小（可以想象下词跟字在数量上的差别），所以自己进行预训练的时候，需要根据拥有的算力和输出维度来判断是否要使用加速手段。

参考

【1】基于tf2的word2vec模型复现：
https://github.com/wellinxu/nlp_store/blob/master/papers/word2vec.py
【2】word2vec 中的数学原理详解：https://www.cnblogs.com/peghoty/p/3857839.html
【3】word2vec Explained_Deriving Mikolov et al.'s Negative-Sampling Word-Embedding Method:https://arxiv.org/pdf/1402.3722v1.pdf

收录于话题 #NLP·35个

上一篇 · 关于中文预训练模型泛化能力挑战赛的调研

阅读原文

喜欢此内容的人还喜欢

徐朝東教授講座成功舉辦
文献语言学

成都妹子确诊被骂上热搜？都2020年了，蹦个迪怎么就成荡妇了？！
INSIGHT视界

张译："电影的每一分、每一秒都如此珍贵"
南方周末