word2Vector ,word embedding in python and tensorflow

Moses 神经网络学习 2018-10-23

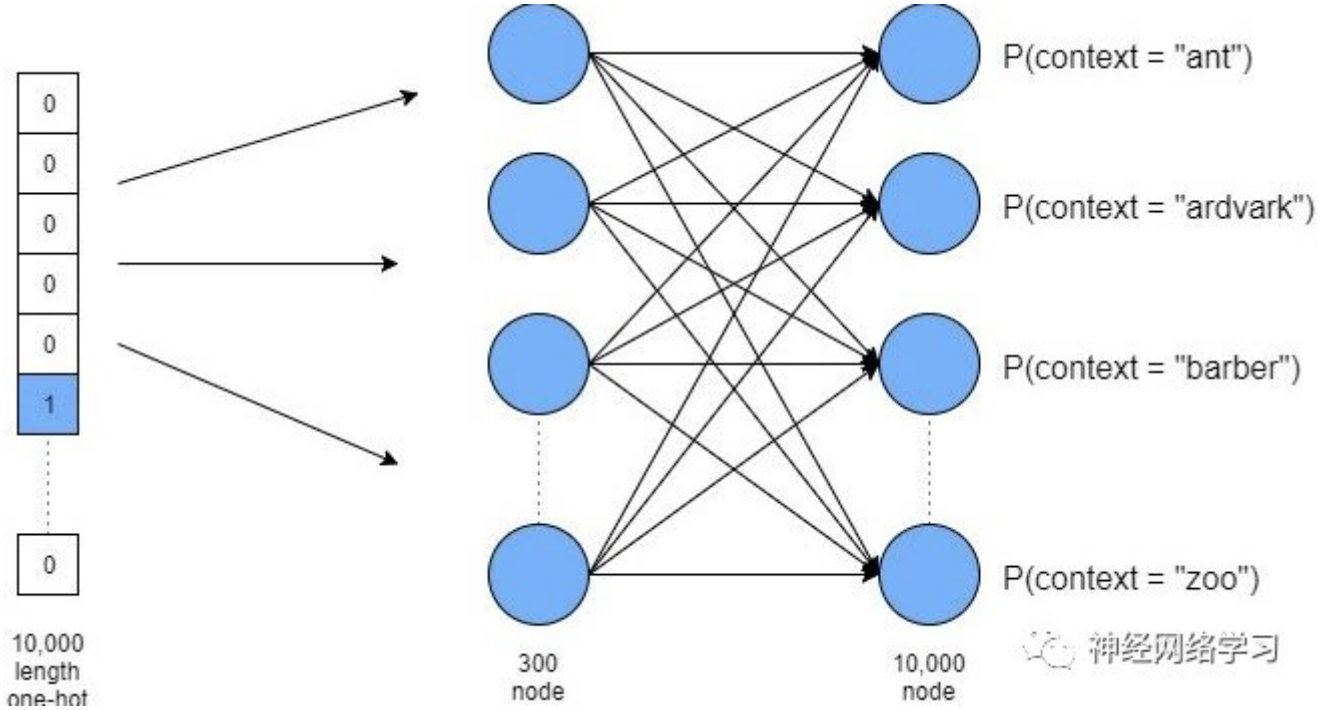

图1

1,为什么需要word2vetor?

如果我们想将单词提供给机器学习模型，除非我们使用基于树的方法，否则我们需要将单词转换为一组数字向量,一种直接的方法是使用“one-hot”方法将单词转换为稀疏表示，只有一个向量元素设置为1，其余为零,因此，对于“the cat sat on the mat”的句子，我们将有以下矢量表示:

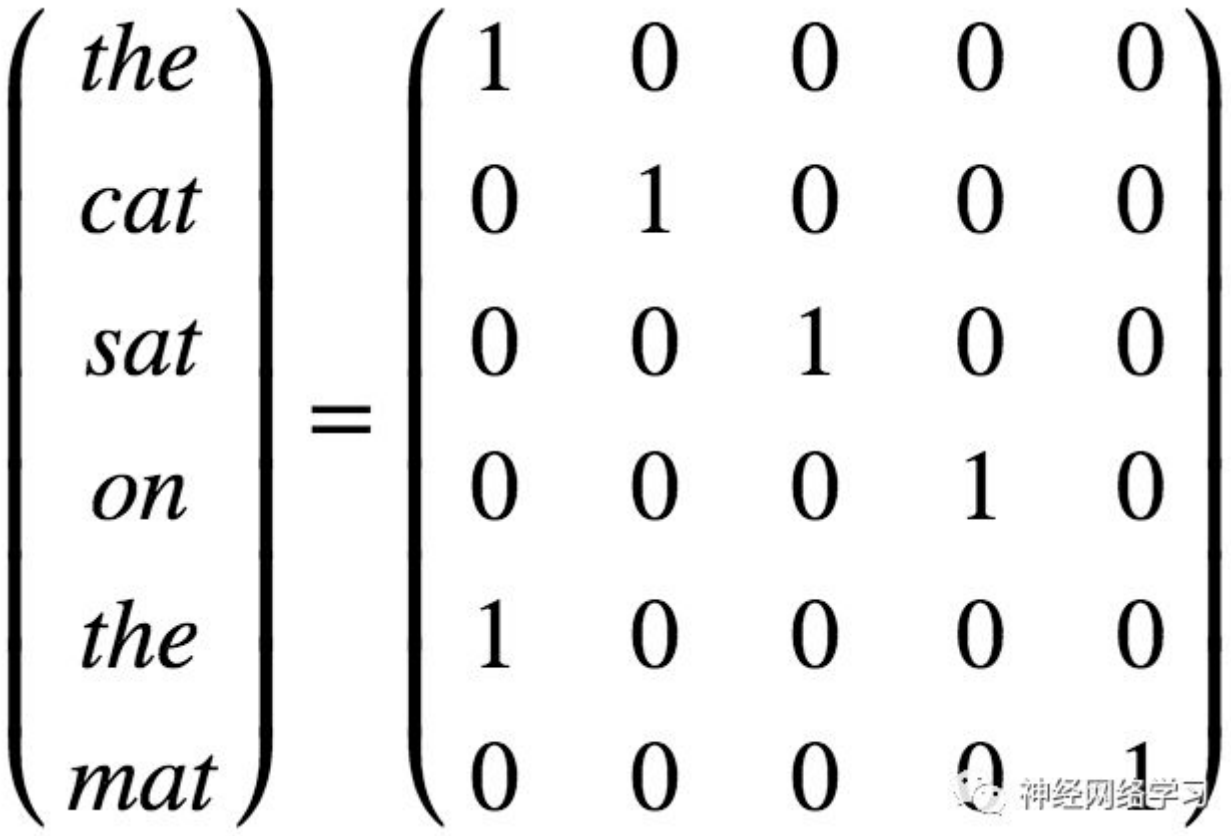

图2

在这里，我们将六个单词的句子转换为6×5矩阵，其中vocabulary的大小是5,the这个单词重复;然而，在实际应用中，我们希望机器和深度学习模型能够从巨大的词汇表中学习,即10,000个单词以上,你可以开始看到使用单词的“one-hot”表示的效率问题 ,尝试对这样的词汇表建模的任何神经网络中的输入层 我们不得不设置10,000个节点,不仅如此，这种方法剥离了单词的任何局部上下文 - 换句话说，它剥离了关于在句子中（或句子之间通常出现在一起的单词的信息,

2, word2voector的方法

如前所述，Word2Vec方法有两个组成部分,第一个是将单词的高维one-hot格式表示映射到低维向量,例如，这可能涉及将10,000个列矩阵转换为300个列柱矩阵。此过程成为word embedding,

第二个目标是在保持单词背景的同时，在某种程度上保持意义,在Word2Vec方法中实现这两个目标的一种方法是通过获取输入词然后尝试估计出现在该词附近的其他词的概率----这称为skip-gram方法.

另外一个方法是连续词袋（CBOW），恰恰相反 - 它将一些上下文词作为输入，并试图找到具有最高概率拟合该上下文的单个词。

什么是gram?一个gram是一组n个的单词,n是gram一个窗口的大小,例如对于这个句子:“the cat sat on the mat”,3-gram表示为:“the cat sat”,“cat sat on ”,“sat on the”,“跳过”部分指的是在具有不同上下文单词的数据集中重复输入单词的次数.这些grams被送入Word2Vec上下文预测系统,例如,假设输入的单词是“cat”,Word2Vec尝

试从提供的输入字中预测上下文（“the”，“sat”）,Word2Vec系统将遍历所有提供的gram和输入单词，并尝试学习适当的mapping vectors（embeddings），这些向量在给定输入单词的情况下为正确的上下文产生高概率。

3,softmax Word2Vec方法

查看图1,在这种情况下，我们假设句子“The cat sat on the mat”是一个更大的文本数据库的一部分，具有非常大的词汇量 - 比如10,000个单词的长度。我们希望将其减少到300长度embedding,关于图1,如果我们使用“cat”这个词，它将成为10,000个词汇词汇中的一个词,因此，我们可以将其表示为10,000长度的one-hot vector,然后，我们将此输入向量连接到300节点隐藏层,连接这一层的权重将是我们新的word vectors,此隐藏层中节点的激活只是加权输入的线性求和（即，不应用非线性激活，如sigmoid或tanh）,然后将这些节点送到softmax输出层。在训练期间，我们想要改变这个神经网络的权重，以便围绕“cat”的词在softmax输出层中具有更高的概率,通过训练这个网络，我们将创建一个10,000 x 300的权重矩阵，连接10,000长度输入和300节点隐藏层,该矩阵中的每一行对应于10,000字词汇表中的一个单词,所以我们有效地将我们单词的10,000长度one-hot vector减少到300个长度vector,权重矩阵基本上成为我们单词的查找或编码表。不仅如此，由于我们训练网络的方式，这些权重值包含上下文信息,一旦我们训练了网络，我们就放弃了softmax层，只使用10,000 x 300权重矩阵作为我们的word embedding查找表.

3,TensorFlow中的softmax Word2Vec方法

```
def maybe_download(filename, url, expected_bytes):
    """Download a file if not present, and make sure it's the right size."""
    if not os.path.exists(filename):
        filename, _ = urllib.request.urlretrieve(url + filename, filename)
    statinfo = os.stat(filename)
    if statinfo.st_size == expected_bytes:
        print('Found and verified', filename)
    else:
        print(statinfo.st_size)
        raise Exception(
            'Failed to verify ' + filename + '. Can you get to it with a browser?'
    return filename
url = 'http://mattmahoney.net/dc/'
filename = maybe_download('text8.zip', url, 31344016)
# Read the data into a list of strings.
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
vocabulary = read_data(filename)
print(vocabulary[:7])
['anarchism', 'originated', 'as', 'a', 'term', 'of', 'abuse']
```

我们必须做一些进一步的处理，以使我们能够创建我们的skip-gram批量数据。这些进一步的步骤是;

1,提取前10,000个最常用的单词以包含在word embedding 中

2,将所有唯一单词聚集在一起并使用唯一的整数值对其进行索引 - 这是为单词创建等效的单热输入所需的内容。我们将使用字典来完成此操作

3,循环遍历数据集中的每个单词（vocabulary variable）并将其分配给在上面的步骤2中创建的唯一整数单词。这将允许容易地查找/处理字数据流

```
def build_dataset(words, n_words):
    """Process raw inputs into a dataset."""
    count = [['UNK', -1]]
    count.extend(collections.Counter(words).most_common(n_words - 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 += 1
        data.append(index)
    count[0][1] = unk_count
    reversed_dictionary = dict(zip(dictionary.values(), dictionary.keys()))
    return data, count, dictionary, reversed_dictionary
```

第一步是设置“计数器”列表，该列表将存储在数据集中找到单词的次数

因为我们将词汇量限制在10,000字以内,任何不在最常见的10,000个单词中的单词都将标有UNK,代表是未知的.

```
data_index = 0# generate batch data
```

```
def generate_batch(data, 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)
    context = np.ndarray(shape=(batch_size, 1), dtype=np.int32)
    span = 2 * skip_window + 1  # [ skip_window input_word 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  # input word 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]  # this is the input wo
            context[i * num_skips + j, 0] = buffer[target]  # these are the contex
        buffer.append(data[data_index])
        data_index = (data_index + 1) % len(data)
    # Backtrack a little bit to avoid skipping words in the end of a batch
    data_index = (data_index + len(data) - span) % len(data)
    return batch, context
```

此功能将生成在我们的培训期间使用的小批量,例如在5-gram中“the cat sat on the”,输入单词为sat,将预测的上下文单词将从剩余单词gram [‘the’, ‘cat’, ‘on’, ‘the’].中随机抽取,

我们在输入字“sat”的时候,skip-window-width 的大小是2,

跨度 span 5 = 2*skip-window-width + 1

```
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  # input word 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]  # this is the input word
        context[i * num_skips + j, 0] = buffer[target]  # these are the context wo
    buffer.append(data[data_index])
    data_index = (data_index + 1) % len(data)
valid_size = 16     #用于评估相似性的随机词集
valid_window = 100  #仅在分布的头部选择开发样本

valid_examples = np.random.choice(valid_window,
valid_size, replace=False)
```

上面的代码从0-100中随机选择16个整数，
这对应于文本数据中最常见的100个单词的整数索引

5,创建TensorFlow模型

```
batch_size = 128
embedding_size = 128
skip_window = 1
num_skips = 2
train_inputs = 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)
embeddings = tf.Variable(
    tf.random_uniform([vocabulary_size, embedding_size],-1.0, 1.0))
embed = tf.nn.embedding_lookup(embeddings, train_inputs)
```

我们使用-1.0到1.0之间的随机均匀分布初始化变量,这个变量的大小是（vocabulary_size，embedding_size）

vocabulary_size是我们在面用来设置数据的10,000个单词

```
weights = tf.Variable(tf.truncated_normal([vocabulary_size,
embedding_size],stddev=1.0 / math.sqrt(embedding_size)))
biases = tf.Variable(tf.zeros([vocabulary_size]))
```

```
hidden_out = tf.matmul(embed, tf.transpose(weights)) + biases

train_one_hot = tf.one_hot(train_context, vocabulary_size)
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=hidd
    labels=train_one_hot))
optimizer = tf.train.GradientDescentOptimizer(1.0).minimize(cross_entropy)
```

接下来，我们需要执行相似性评估，以检查模型在训练时的表现

为了确定哪些单词彼此相似，我们需要执行某种操作来测量不同word embedding vectors之间的“距离”,我们将使用向量之间距离的余弦相似性度量,表达式为 cos(θ

)=**A** · **B**/(||**A** 2|| **B**||2) }

这里粗体的A和B是我要测量相似度的两个向量.

```
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, normalized_embeddings, transpose_b=True)

6,运行tensorflow model
with tf.Session(graph=graph) as session:
  # 在使用之前,我们必须初始化
  init.run()
  print('Initialized')

  average_loss = 0
  for step in range(num_steps):
    batch_inputs, batch_context = generate_batch(data,
        batch_size, num_skips, skip_window)
    feed_dict = {train_inputs: batch_inputs, train_context: batch_context}
    _, loss_val = session.run([optimizer, cross_entropy], feed_dict=feed_dict)
    average_loss += loss_val    if step % 2000 == 0:
      if step > 0:
        average_loss /= 200
      print('Average loss at step ', step, ': ', average_loss)
      average_loss = 0
```

```
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_str = 'Nearest to %s:' % valid_word        for k in range(top_k):
            close_word = reverse_dictionary[nearest[k]]
            log_str = '%s %s,' % (log_str, close_word)
        print(log_str)
final_embeddings = normalized_embeddings.eval()
```

事实上，执行softmax评估并更新10,000字输出/词汇表的权重真的很慢。为什么？考虑softmax定义

$$P(y = j \mid x) = \frac{e^{xTwj}}{\sum_{k=1}^{K} e^{xTwk}}$$

在我们正在研究的内容中，softmax函数将预测哪些单词在输入单词的上下文中具有最高概率,然而，为了确定该概率，softmax函数的分母必须评估词汇表中的所有可能的上下文单词,因此，我们需要300 x 10,000 = 3M的重量，所有这些都需要针对softmax输出进行训练。这会减慢速度.

```
nce_weights = tf.Variable(
        tf.truncated_normal([vocabulary_size, embedding_size],
                            stddev=1.0 / math.sqrt(embedding_size)))
nce_biases = tf.Variable(tf.zeros([vocabulary_size]))
nce_loss = tf.reduce_mean(
        tf.nn.nce_loss(weights=nce_weights,
                       biases=nce_biases,
                       labels=train_context,
                       inputs=embed,
```

```
                        num_sampled=num_sampled,
                        num_classes=vocabulary_size))
optimizer = tf.train.GradientDescentOptimizer(1.0).minimize(nce_loss)
init = tf.global_variables_initializer()
num_steps = 50000
run(graph, num_steps)
```

参看

1,https://github.com/adventuresinML/adventures-in-ml-code/blob/master/tf_word2vec.py

2,http://adventuresinmachinelearning.com/word2vec-tutorial-tensorflow