标题 An implementation guide to Word2Vec using NumPy and Google Sheets

手把手教你NumPy来实现Word2vec

01

Word2Vec is touted as one of the biggest, most recent breakthrough in the field of Natural Language Processing (NLP). The concept is simple, elegant and (relatively) easy to grasp. A quick Google search returns multiple results on how to use them with standard libraries such as Gensim and TensorFlow. Also, for the curious minds, check out the original implementation using C by Tomas Mikolov. The original paper can be found here too.

Word2Vec被认为是自然语言处理（NLP）领域中最大、最新的突破之一。其的概念简单，优雅，（相对）容易掌握。Google一下就会找到一堆关于如何使用诸如Gensim和TensorFlow的库来调用Word2Vec方法的结果。另外，对于那些好奇心强的人，可以查看Tomas Mikolov基于C语言的原始实现。原稿也可以在这里找到。

The main focus on this article is to present Word2Vec in detail. For that, I implemented Word2Vec on Python using Numpy (with much help from other tutorials) and also prepared a Google Sheet to showcase the calculations. Here are the links to the code and Google Sheet.

本文的主要重点是详细介绍Word2Vec。为此，我在Python上使用Numpy（在其他教程的帮助下）实现了Word2Vec，还准备了一个Google Sheet来展示计算结果。以下是代码和Google Sheet的链接。

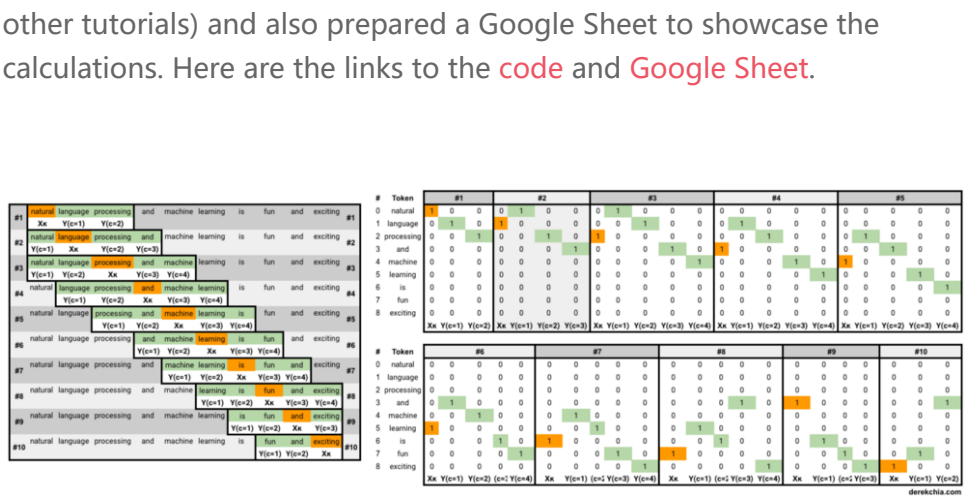


Fig. 1 — Step-by-step introduction to Word2Vec. Presented in code and Google Sheets

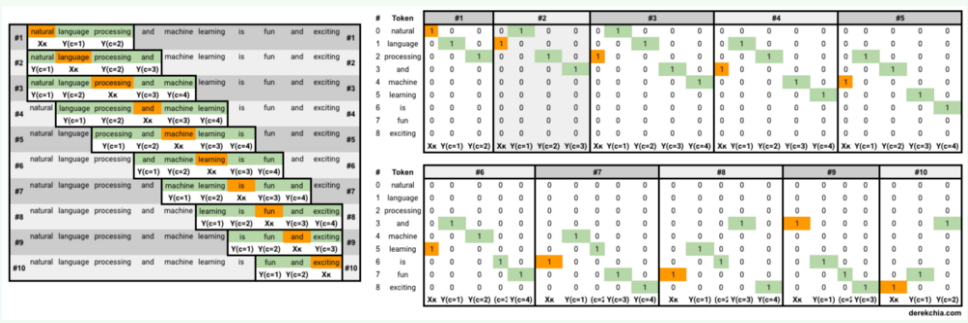


图1.一步一步来介绍Word2Vec。由代码和Google Sheet呈现

Intuition

直观上看

The objective of Word2Vec is to generate vector representations of words that carry semantic meanings for further NLP tasks. Each word vector is typically several hundred dimensions and each unique word in the corpus is assigned a vector in the space. For example, the word “happy” can be represented as a vector of 4 dimensions [0.24, 0.45, 0.11, 0.49] and “sad” has a vector of [0.88, 0.78, 0.45, 0.91].

Word2Vec的目标是生成带有语义的单词的向量表示，用于进一步的NLP任务。每个单词向量通常有几百个维度，语料库中每个唯一的单词在空间中被分配一个向量。例如，单词“happy”可以表示为4维向量[0.24、0.45、0.11、0.49]，“sad”具有向量[0.88、0.78、0.45、0.91]。

The transformation from words to vectors is also known as word embedding. The reason for this transformation is so that machine learning algorithm can perform linear algebra operations on numbers (in vectors) instead of words.

这种从单词到向量的转换也被称为单词嵌入（word embedding）。这种转换的原因是机器学习算法可以对数字（在向量中的）而不是单词进行线性代数运算。

To implement Word2Vec, there are two flavors to choose from — Continuous Bag-Of-Words (CBOW) or continuous Skip-gram (SG). In short, CBOW attempts to guess the output (target word) from its neighbouring words (context words) whereas continuous Skip-Gram guesses the context words from a target word. Effectively, Word2Vec is based on distributional hypothesis where the context for each word is in its nearby words. Hence, by looking at its neighbouring words, we can attempt to predict the target word.

为了实现Word2Vec，有两种风格可以选择，Continuous Bag-of-Words(CBOW)或Skip-gram(SG)。简单来说，CBOW尝试从相邻单词（上下文单词）猜测输出（目标单词），而Skip-Gram从目标单词猜测上下文单词。实际上，Word2Vec是基于分布假说，其认为每个单词的上下文都在其附近的单词中。因此，通过查看它的相邻单词我们可以尝试对目标单词进行预测。

According to Mikolov (quoted in this article), here is the difference between Skip-gram and CBOW:

根据Mikolov（引用于这篇文章），以下是Skip-gram和CBOW之间的区别：

> Skip-gram: works well with small amount of the training data, represents well even rare words or phrases
>
> CBOW: several times faster to train than the skip-gram, slightly better accuracy for the frequent words

> Skip-gram：能够很好地处理少量的训练数据，而且能够很好地表示不常见的单词或短语
>
> CBOW：比skip-gram训练快几倍，对出现频率高的单词的准确度稍微更好一些

To elaborate further, since Skip-gram learns to predict the context words from a given word, in case where two words (one appearing infrequently and the other more frequently) are placed side-by-side, both will have the same treatment when it comes to minimising loss since each word will be treated as both the target word and context word. Comparing

更详细地说，由于Skip-gram学习用给定单词来预测上下文单词，所以万一两个单词（一个出现频率较低，另一个出现频率较高）放在一起，那么当最小化loss值时，两个单词将进行有相同的处理，因为每个单词都将被当作目标单词和上下文单词。与CBOW相比，不常见的单词将只是用于预测目标单词的上下文单词集合的一部分。因此，该模型将给不常现的单词分配一个低概率。

that to CBOW, the infrequent word will only be part of a collection of context words used to predict the target word. Therefore, the model will assign the infrequent word a low probability.

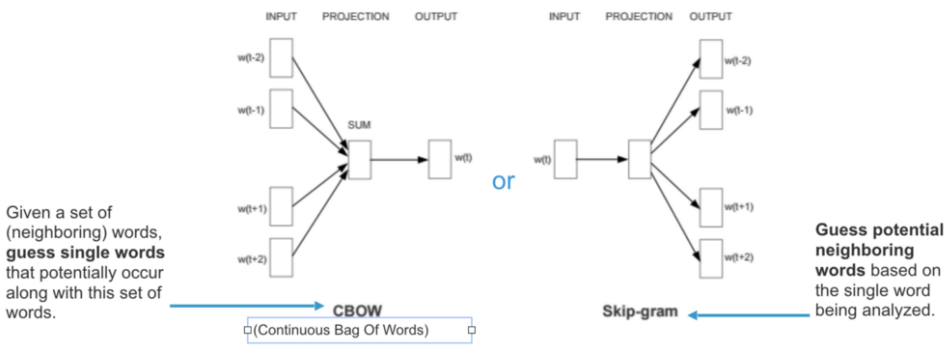


Fig. 2—Word2Vec—CBOW and skip-gram model architectures. Credit: IDIL

02

Implementation Process

In this article, we will be implementing the Skip-gram architecture. The content is broken down into the following parts for easy reading:

Data Preparation—Define corpus, clean, normalise and tokenise words

Hyperparameters—Learning rate, epochs, window size, embedding size

Generate Training Data—Build vocabulary, one-hot encoding for words, build dictionaries that map id to word and vice versa

Model Training—Pass encoded words through forward pass, calculate error rate, adjust weights using backpropagation and compute loss

Inference—Get word vector and find similar words

Further improvements—Speeding up training time with Skip-gram Negative Sampling (SGNS) and Hierarchical Softmax

1. Data Preparation

To begin, we start with the following corpus:

natural language processing and machine learning is fun and exciting

For simplicity, we have chosen a sentence without punctuation and capitalisation. Also, we did not remove stop words "and" and "is".

In reality, text data are unstructured and can be "dirty". Cleaning them will involve steps such as removing stop words, punctuations, convert text to lowercase (actually depends on your use-case), replacing digits, etc. KDnuggets has an excellent article on this process. Alternatively, Gensim also provides a function to perform simple text preprocessing using gensim.utils.simple_preprocess where it converts a document into a list of lowercase tokens, ignoring tokens that are too short or too long.

```
text = "natural language processing and machine learning is fun and exciting"

# Note the .lower() as upper and lowercase does not matter in our implementation
# [['natural', 'language', 'processing', 'and', 'machine', 'learning', 'is', 'fun', 'and', 'exciting']]
corpus = [[word.lower() for word in text.split()]]
```

w2v_generate_training_data.py hosted with ❤ by GitHub view raw

After preprocessing, we then move on to tokenising the corpus. Here, we tokenise our corpus on whitespace and the result is a list of words:

["natural", "language", "processing", "and", "machine", "learning", "is", "fun", "and", "exciting"]

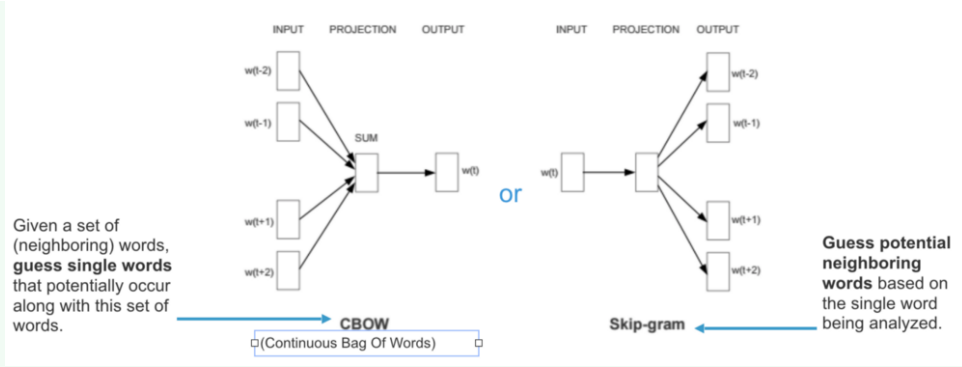


图2—Word2Vec—CBOW和skip-gram模型架构。感谢：IDIL

实现过程

在本文中，我们将实现Skip-gram体系结构。为了便于阅读，内容分为以下几个部分：

1. 数据准备——定义语料库、整理、规范化和分词

2. 超参数——学习率、训练次数、窗口尺寸、嵌入（embedding）尺寸

3. 生成训练数据——建立词汇表，对单词进行one-hot编码，建立将id映射到单词的字典，以及单词映射到id的字典

4. 模型训练——通过正向传递编码过的单词，计算错误率，使用反向传播调整权重和计算loss值

5. 结论——获取词向量，并找到相似的词

6. 进一步的改进 —— 利用Skip-gram负采样(Negative Sampling)和Hierarchical Softmax提高训练速度

1. 数据准备
首先，我们从以下语料库开始：

natural language processing and machine learning is fun and exciting

简单起见，我们选择了一个没有标点和大写的句子。而且，我们没有删除停用词“and”和“is”。

实际上，文本数据是非结构化的，甚至可能很“很不干净”清理它们涉及一些步骤，例如删除停用词、标点符号、将文本转换为小写（实际上取决于你的实际例子）和替换数字等。KDnuggets 上有一篇关于这个步骤很棒的文章。另外，Gensim也提供了执行简单文本预处理的函数——gensim.utils.simple_preprocess，它将文档转换为由小写的词语（Tokens ）组成的列表，并忽略太短或过长的词语。

```
text = "natural language processing and machine learning is fun and exciting"

# Note the .lower() as upper and lowercase does not matter in our implementation
# [['natural', 'language', 'processing', 'and', 'machine', 'learning', 'is', 'fun', 'and', 'exciting']]
corpus = [[word.lower() for word in text.split()]]
```

w2v_generate_training_data.py hosted with ❤ by GitHub view raw

在预处理之后，我们开始对语料库进行分词。我们按照单词间的空格对我们的语料库进行分词，结果得到一个单词列表：

[“natural”，“language”，“processing”，“ and”，“ machine”，“ lea rning”，“ is”，“ fun”，“and”，“ exciting”]

2.超参数

2. Hyperparameters

Before we jump into the actual implementation, let us define some of the hyperparameters we need later.

```
settings = {
        'window_size': 2        # context window +- center word
        'n': 10,                # dimensions of word embeddings, also refer to size of hidden layer
        'epochs': 50,           # number of training epochs
        'learning_rate': 0.01   # learning rate
}
```

w2v_settings.py hosted with ❤ by GitHub view raw

[window_size]: As mentioned above, context words are words that are neighbouring the target word. But how far or near should these words be in order to be considered neighbour? This is where we define the window_size to be 2 which means that words that are 2 to the left and right of the target words are considered context words. Referencing Figure 3 below, notice that each of the word in the corpus will be a target word as the window slides.

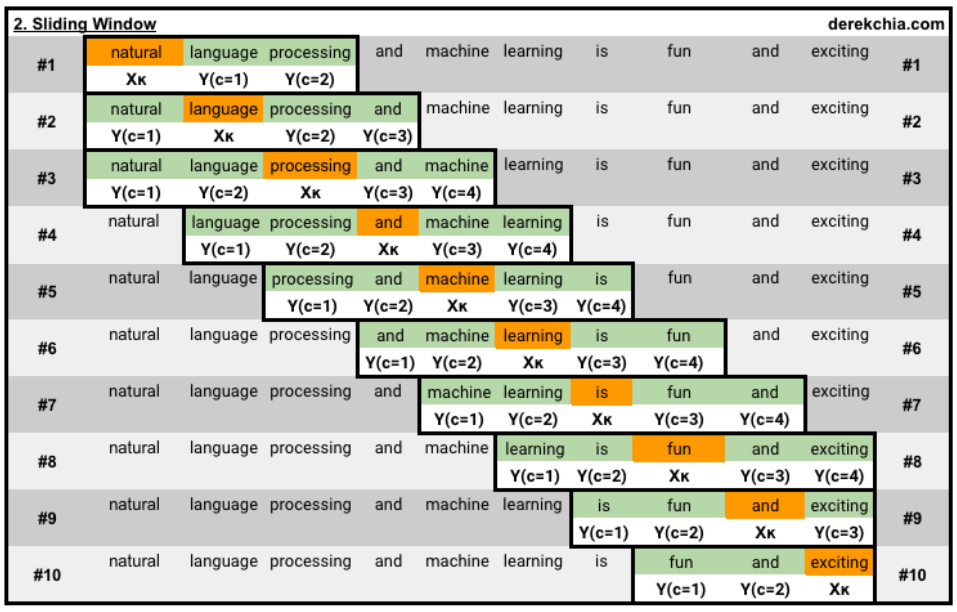


FIg. 3—With a window_size of 2, the target word is highlighted in orange and context words in green

[n]: This is the dimension of the word embedding and it typically ranges from 100 to 300 depending on your vocabulary size. Dimension size beyond 300 tends to have diminishing benefit (see page 1538 Figure 2 (a)). Do note that the dimension is also the size of the hidden layer.

[epochs]: This is the number of training epochs. In each epoch, we cycle through all training samples.

[learning_rate]: The learning rate controls the amount of adjustment made to the weights with respect to the loss gradient.

在进入word2vec的实现之前，让我们先定义一些稍后需要用到的超参数。

```
settings = {
        'window_size': 2        # context window +- center word
        'n': 10,                # dimensions of word embeddings, also refer to size of hidden layer
        'epochs': 50,           # number of training epochs
        'learning_rate': 0.01   # learning rate
}
```

w2v_settings.py hosted with ❤ by GitHub view raw

[window_size/窗口尺寸]: 如之前所述，上下文单词是与目标单词相邻的单词。但是，这些词应该有多远或多近才能被认为是相邻的呢？这里我们将窗口尺寸定义为2，这意味着目标单词的左边和右边最近的2个单词被视为上下文单词。参见下面的图3，可以看到，当窗口滑动时，语料库中的每个单词都会成为一个目标单词。

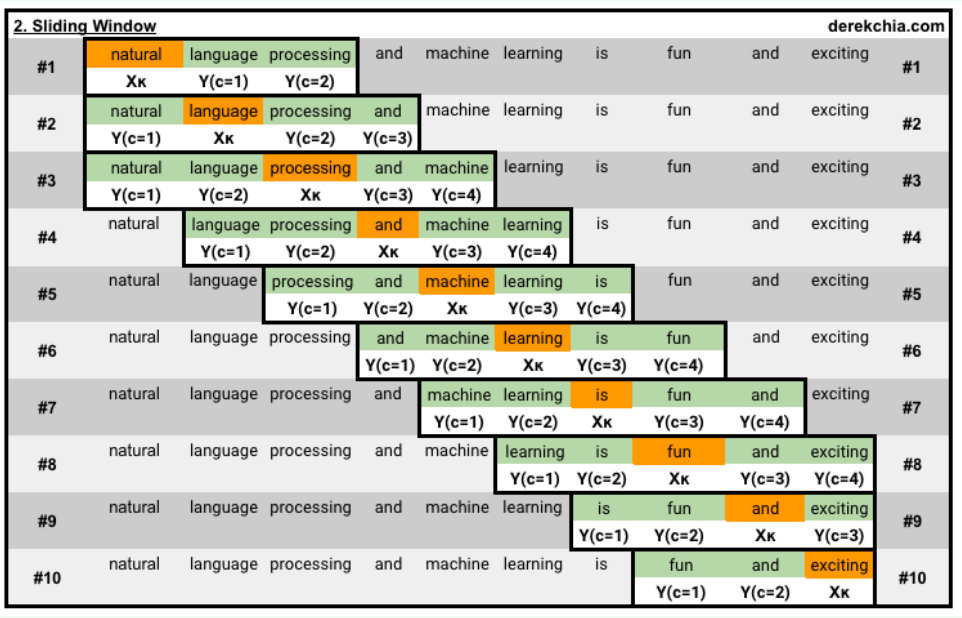


图3，在window_size为2的情况下，目标单词用橙色高亮显示，上下文单词用绿色高亮显示

[n]：这是单词嵌入(word embedding)的维度，通常其的大小通常从100到300不等，取决于词汇库的大小。超过300维度会导致效益递减（参见图2（a）的1538页）。请注意，维度也是隐藏层的大小。

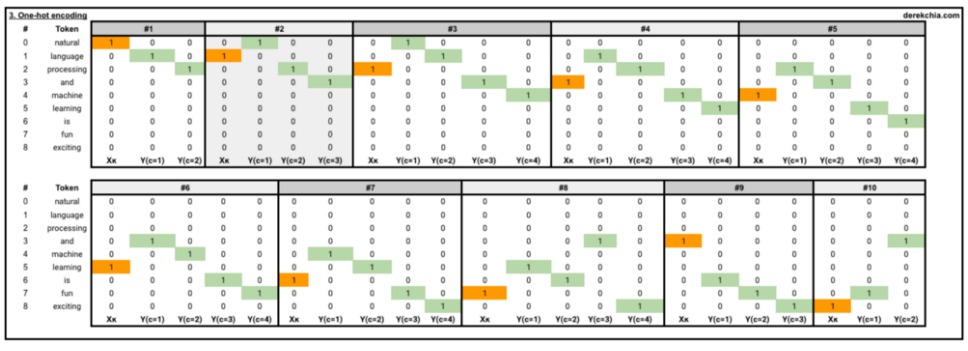
[epochs] ：表示遍历整个样本的次数。在每个epoch中，我们循环通过一遍训练集的样本。

[learning_rate/学习率]：学习率控制着损失梯度对权重进行调整的量。

03

3. Generate Training Data

In this section, our main objective is to turn our corpus into a one-hot encoded representation for the Word2Vec model to train on. From our corpus, Figure 4 zooms into each of the 10 windows (#1 to #10) as shown below. Each window consists of both the target word and its context words, highlighted in orange and green respectively.



3.生成训练数据

在本节中，我们的主要目标是将语料库转换one-hot编码表示，以方便Word2vec模型用来训练。从我们的语料库中，图4中显示了10个窗口（#1到#10）中的每一个。每个窗口都由目标单词及其上下文单词组成，分别用橙色和绿色高亮显示。

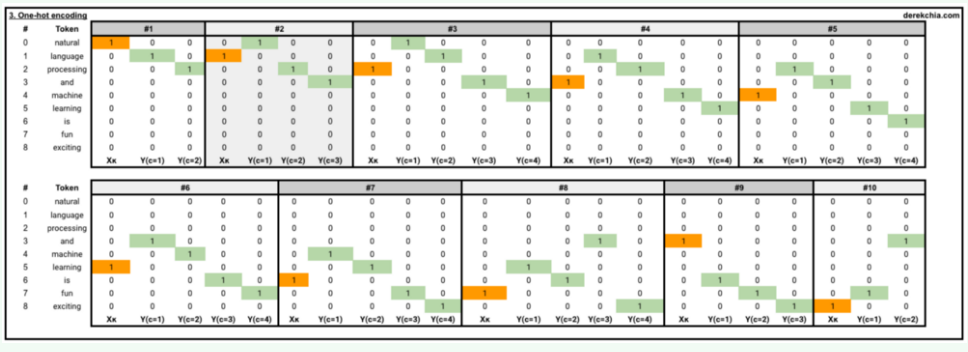


图4，每个目标单词及其上下文单词的one hot编码

Fig. 4—One-hot encoding for each target word and its context words

Example of the first and last element in the first and last training window is shown below:

```
# 1 [Target (natural)], [Context (language, processing)]
[list([1, 0, 0, 0, 0, 0, 0, 0, 0])
 list([[0, 1, 0, 0, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0, 0, 0, 0]])]


*****#2 to #9 removed****


#10 [Target (exciting)], [Context (fun, and)]
[list([0, 0, 0, 0, 0, 0, 0, 0, 1])
 list([[0, 0, 0, 0, 0, 0, 0, 1, 0], [0, 0, 0, 1, 0, 0, 0, 0, 0]])]
```

To generate the one-hot training data, we first initialise the word2vec() object and then using the object w2v to call the function generate_training_data by passing settings and corpus as arguments.

```
# Initialise object
w2v = word2vec()
# Numpy ndarray with one-hot representation for [target_word, context_words]
training_data = w2v.generate_training_data(settings, corpus)
```

w2v_generate_training_data_2.py hosted with ❤ by GitHub view raw

Inside the function generate_training_data, we performed the following operations:

self.v_count—Length of vocabulary (note that vocabulary refers to the number of unique words in the corpus)

self.words_list—List of words in vocabulary

self.word_index—Dictionary with each key as word in vocabulary and value as index

self.index_word—Dictionary with each key as index and value as word in vocabulary

for loop to append one-hot representation for each target and its context words to training_data using word2onehot function.

```
class word2vec():
  def __init__(self):
    self.n = settings['n']
    self.lr = settings['learning_rate']
    self.epochs = settings['epochs']
    self.window = settings['window_size']

  def generate_training_data(self, settings, corpus):
    # Find unique word counts using dictonary
    word_counts = defaultdict(int)
    for row in corpus:
      for word in row:
        word_counts[word] += 1
    ## How many unique words in vocab? 9
    self.v_count = len(word_counts.keys())
    # Generate Lookup Dictionaries (vocab)
    self.words_list = list(word_counts.keys())
    # Generate word:index
    self.word_index = dict((word, i) for i, word in enumerate(self.words_list))
    # Generate index:word
    self.index_word = dict((i, word) for i, word in enumerate(self.words_list))

    training_data = []
    # Cycle through each sentence in corpus
    for sentence in corpus:
      sent_len = len(sentence)
      # Cycle through each word in sentence
      for i, word in enumerate(sentence):
        # Convert target word to one-hot
        w_target = self.word2onehot(sentence[i])
        # Cycle through context window
        w_context = []
        # Note: window_size 2 will have range of 5 values
        for j in range(i - self.window, i + self.window+1):
          # Criteria for context word
          # 1. Target word cannot be context word (j != i)
          # 2. Index must be greater or equal than 0 (j >= 0) - if not list index out of range
          # 3. Index must be less or equal than length of sentence (j <= sent_len-1) - if not list
```

第一个和最后一个训练窗口中的第一个和最后一个元素的示例如下所示：

```
# 1 [目标单词(natural)], [上下文单词 (language, processing)]
[list([1, 0, 0, 0, 0, 0, 0, 0, 0])
 list([[0, 1, 0, 0, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0, 0, 0, 0]])]


*****#2 to #9 省略****


#10 [ 目标单词  (exciting)], [ 上下文单词  (fun, and)]
[list([0, 0, 0, 0, 0, 0, 0, 0, 1])
 list([[0, 0, 0, 0, 0, 0, 0, 1, 0], [0, 0, 0, 1, 0, 0, 0, 0, 0]])]
```

为了生成one-hot训练数据，我们首先初始化word2vec()对象，然后使用对象w2v通过*settings* 和*corpus* 参数来调用函数*generate_training_data*

```
# Initialise object
w2v = word2vec()
# Numpy ndarray with one-hot representation for [target_word, context_words]
training_data = w2v.generate_training_data(settings, corpus)
```

w2v_generate_training_data_2.py hosted with ❤ by GitHub view raw

在函数generate_training_data内部，我们进行以下操作：

1. self.v_count：词汇表的长度（注意，词汇表指的就是语料库中不重复的单词的数量）

2. self.words_list：在词汇表中的单词组成的列表

3. self.word_index：以词汇表中单词为key，索引为value的字典数据

4. self.index_word：以索引为key，以词汇表中单词为value的字典数据

5. for循环给用one-hot表示的每个目标词和其的上下文词添加到training_data中，one-hot编码用的是word2onehot函数。


```
class word2vec():
  def __init__(self):
    self.n = settings['n']
    self.lr = settings['learning_rate']
    self.epochs = settings['epochs']
    self.window = settings['window_size']

  def generate_training_data(self, settings, corpus):
    # Find unique word counts using dictonary
    word_counts = defaultdict(int)
    for row in corpus:
      for word in row:
        word_counts[word] += 1
    ## How many unique words in vocab? 9
    self.v_count = len(word_counts.keys())
    # Generate Lookup Dictionaries (vocab)
    self.words_list = list(word_counts.keys())
    # Generate word:index
    self.word_index = dict((word, i) for i, word in enumerate(self.words_list))
    # Generate index:word
    self.index_word = dict((i, word) for i, word in enumerate(self.words_list))

    training_data = []
    # Cycle through each sentence in corpus
    for sentence in corpus:
      sent_len = len(sentence)
      # Cycle through each word in sentence
      for i, word in enumerate(sentence):
        # Convert target word to one-hot
        w_target = self.word2onehot(sentence[i])
        # Cycle through context window
        w_context = []
        # Note: window_size 2 will have range of 5 values
        for j in range(i - self.window, i + self.window+1):
          # Criteria for context word
          # 1. Target word cannot be context word (j != i)
          # 2. Index must be greater or equal than 0 (j >= 0) - if not list index out of range
          # 3. Index must be less or equal than length of sentence (j <= sent_len-1) - if not list
```

```
            if j != i and j <= sent_len-1 and j >= 0:
              # Append the one-hot representation of word to w_context
              w_context.append(self.word2onehot(sentence[j]))
              # print(sentence[i], sentence[j])
              # training_data contains a one-hot representation of the target word and context words
          training_data.append([w_target, w_context])
      return np.array(training_data)

    def word2onehot(self, word):
      # word_vec - initialise a blank vector
      word_vec = [0 for i in range(0, self.v_count)] # Alternative - np.zeros(self.v_count)
      # Get ID of word from word_index
      word_index = self.word_index[word]
      # Change value from 0 to 1 according to ID of the word
      word_vec[word_index] = 1
      return word_vec
```

w2v_generate_training_data_func.py hosted with ❤ by GitHub view raw

04

4. Model Training

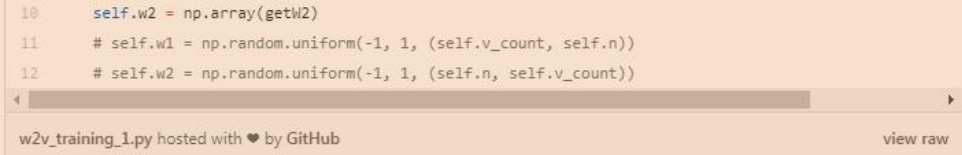


Fig. 5 — Word2Vec — skip-gram network architecture

With our training_data, we are now ready to train our model. Training starts with w2v.train(training_data) where we pass in the training data and call the function train.

The Word2Vec model consists of 2 weight matrices (w1 and w2) and for demo purposes, we have initialised the values to a shape of (9x10) and (10x9) respectively. This facilitates the calculation of backpropagation error which will be covered later in the article. In the actual training, you should randomly initialise the weights (e.g. using np.random.uniform()). To do that, comment line 9 and 10 and uncomment line 11 and 12.

```
# Training
w2v.train(training_data)

class word2vec():
  def train(self, training_data):
  # Initialising weight matrices
  # Both s1 and s2 should be randomly initialised but for this demo, we pre-determine the arrays (
  # getW1 - shape (9x10) and getW2 - shape (10x9)
  self.w1 = np.array(getW1)
  self.w2 = np.array(getW2)
  # self.w1 = np.random.uniform(-1, 1, (self.v_count, self.n))
  # self.w2 = np.random.uniform(-1, 1, (self.n, self.v_count))
```

w2v_training_1.py hosted with ❤ by GitHub view raw

Training — Forward Pass

Next, we start training our first epoch using the first training example by passing in w_t which represents the one-hot vector for target word to theforward_pass function. In the forward_pass function, we perform a dot product between w1 and w_t to produce h(Line 24). Then, we perform another dot product using w2 and h to produce the output layer u(Line 26). Lastly, we run u through softmax to force each element to the range of 0 and 1 to give us the probabilities for prediction (Line 28) before returning the vector for predictiony_pred, hidden layer h and output layer u.

```
            if j != i and j <= sent_len-1 and j >= 0:
              # Append the one-hot representation of word to w_context
              w_context.append(self.word2onehot(sentence[j]))
              # print(sentence[i], sentence[j])
              # training_data contains a one-hot representation of the target word and context words
          training_data.append([w_target, w_context])
      return np.array(training_data)

    def word2onehot(self, word):
      # word_vec - initialise a blank vector
      word_vec = [0 for i in range(0, self.v_count)] # Alternative - np.zeros(self.v_count)
      # Get ID of word from word_index
      word_index = self.word_index[word]
      # Change value from 0 to 1 according to ID of the word
      word_vec[word_index] = 1
      return word_vec
```

4.模型训练

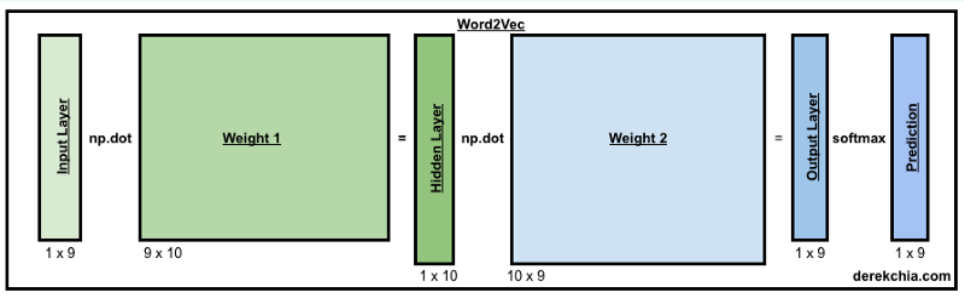


图5，Word2Vec——skip-gram的网络结构

拥有了training_data，我们现在可以准备训练模型了。训练从w2v.train(training_data)开始，我们传入训练数据，并执行train函数。

Word2Vec2模型有两个权重矩阵(w1和w2)，为了展示，我们把值初始化到形状分别为(9x10)和(10x9)的矩阵。这便于反向传播误差的计算，这部分将在后文讨论。在实际的训练中，你应该随机初始化这些权重（比如使用np.random.uniform()）。想要这么做，把第九第十行注释掉，把11和12行取消注释就好。

```
# Training
w2v.train(training_data)

class word2vec():
  def train(self, training_data):
  # Initialising weight matrices
  # Both s1 and s2 should be randomly initialised but for this demo, we pre-determine the arrays (
  # getW1 - shape (9x10) and getW2 - shape (10x9)
  self.w1 = np.array(getW1)
  self.w2 = np.array(getW2)
  # self.w1 = np.random.uniform(-1, 1, (self.v_count, self.n))
  # self.w2 = np.random.uniform(-1, 1, (self.n, self.v_count))
```

w2v_training_1.py hosted with ❤ by GitHub view raw

训练——向前传递

接下来，我们开始用第一组训练样本来训练第一个epoch，方法是把w_t 传入forward_pass 函数，w_t 是表示目标词的one-hot向量。在forward_pass 函数中，我们执行一个w1 和w_t 的点乘积，得到h (原文是24行，但图中实际是第22行)。然后我们执行w2和h 点乘积，得到输出层的u（ 原文是26行，但图中实际是第24行 ）。最后，在返回预测向量y_pred和隐藏层h 和输出层u 前，我们使用softmax把u 的每个元素的值映射到0和1之间来得到用来预测的概率（第28行）。

```python
class word2vec():
  def train(self, training_data):
  ##Removed##

    # Cycle through each epoch
    for i in range(self.epochs):
      # Intialise loss to 0
      self.loss = 0

      # Cycle through each training sample
      # w_t = vector for target word, w_c = vectors for context words
      for w_t, w_c in training_data:
        # Forward pass - Pass in vector for target word (w_t) to get:
        # 1. predicted y using softmax (y_pred) 2. matrix of hidden layer (h) 3. output layer befo
        y_pred, h, u = self.forward_pass(w_t)

        ##Removed##

  def forward_pass(self, x):
    # x is one-hot vector for target word, shape - 9x1
    # Run through first matrix (w1) to get hidden layer - 10x9 dot 9x1 gives us 10x1
    h = np.dot(self.w1.T, x)
    # Dot product hidden layer with second matrix (w2) - 9x10 dot 10x1 gives us 9x1
    u = np.dot(self.w2.T, h)
    # Run 1x9 through softmax to force each element to range of [0, 1] - 1x8
    y_c = self.softmax(u)
    return y_c, h, u

  def softmax(self, x):
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum(axis=0)
```

w2v_training_forward_pass.py hosted with ❤ by GitHub view raw

I have attached some screenshots to show the calculation for the first training sample in the first window (#1) where the target word is ‘natural’ and context words are ‘language’ and ‘processing’. Feel free to look into the formula in the Google Sheet here.

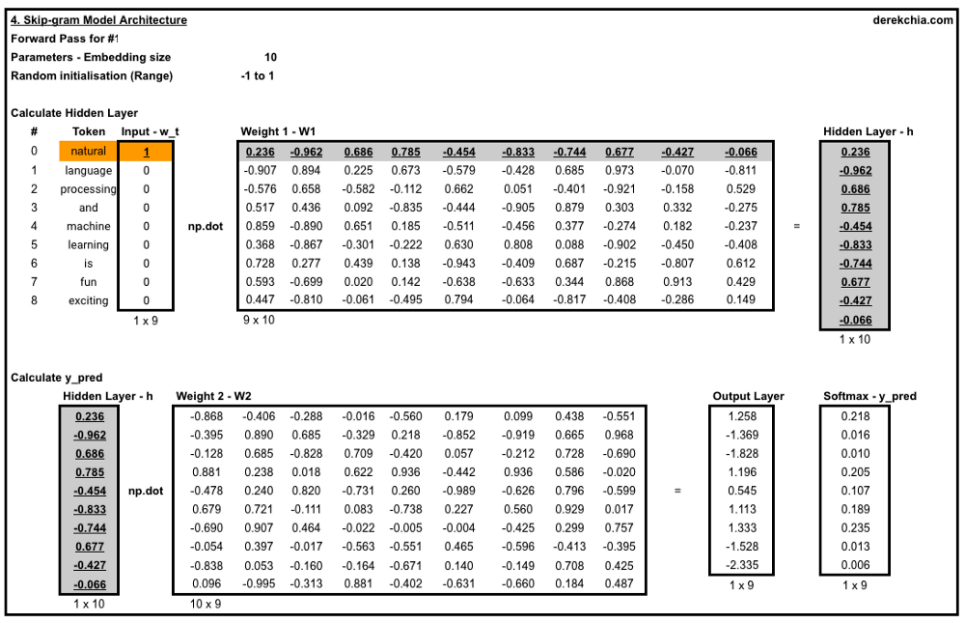


Fig. 6— Calculate hidden layer, output later and softmax

Training — Error, Backpropagation and Loss

Error — With y_pred, h and u, we proceed to calculate the error for this particular set of target and context words. This is done by summing up the difference between y_pred and each of the context words inw_c.

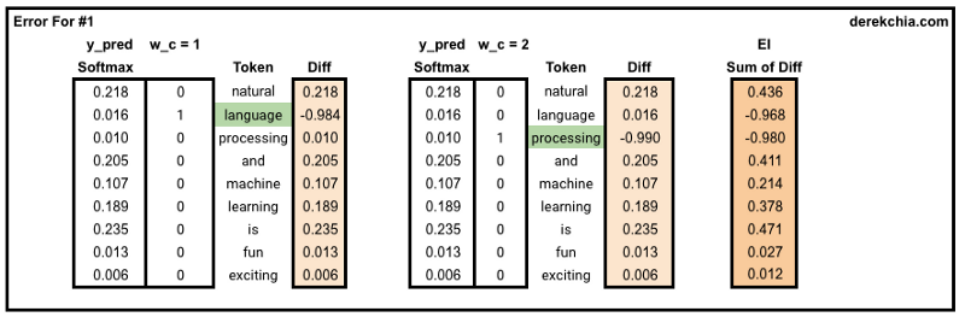


Fig. 7 — Calculating Error — context words are ‘language’ and ‘processing’

Backpropagation — Next, we use the backpropagation function, backprop, to calculate the amount of adjustment we need to alter the weights using the function backprop by passing in error EI, hidden layer h and vector for target word w_t.

```python
class word2vec():
  def train(self, training_data):
  ##Removed##

    # Cycle through each epoch
    for i in range(self.epochs):
      # Intialise loss to 0
      self.loss = 0

      # Cycle through each training sample
      # w_t = vector for target word, w_c = vectors for context words
      for w_t, w_c in training_data:
        # Forward pass - Pass in vector for target word (w_t) to get:
        # 1. predicted y using softmax (y_pred) 2. matrix of hidden layer (h) 3. output layer befo
        y_pred, h, u = self.forward_pass(w_t)

        ##Removed##

  def forward_pass(self, x):
    # x is one-hot vector for target word, shape - 9x1
    # Run through first matrix (w1) to get hidden layer - 10x9 dot 9x1 gives us 10x1
    h = np.dot(self.w1.T, x)
    # Dot product hidden layer with second matrix (w2) - 9x10 dot 10x1 gives us 9x1
    u = np.dot(self.w2.T, h)
    # Run 1x9 through softmax to force each element to range of [0, 1] - 1x8
    y_c = self.softmax(u)
    return y_c, h, u

  def softmax(self, x):
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum(axis=0)
```

w2v_training_forward_pass.py hosted with ❤ by GitHub view raw

我附上一些截图展示第一窗口（#1）中第一个训练样本的计算，其中目标词是“natural”，上下文单词是“language”和“processing”。可以在这里查看Google Sheet中的公式

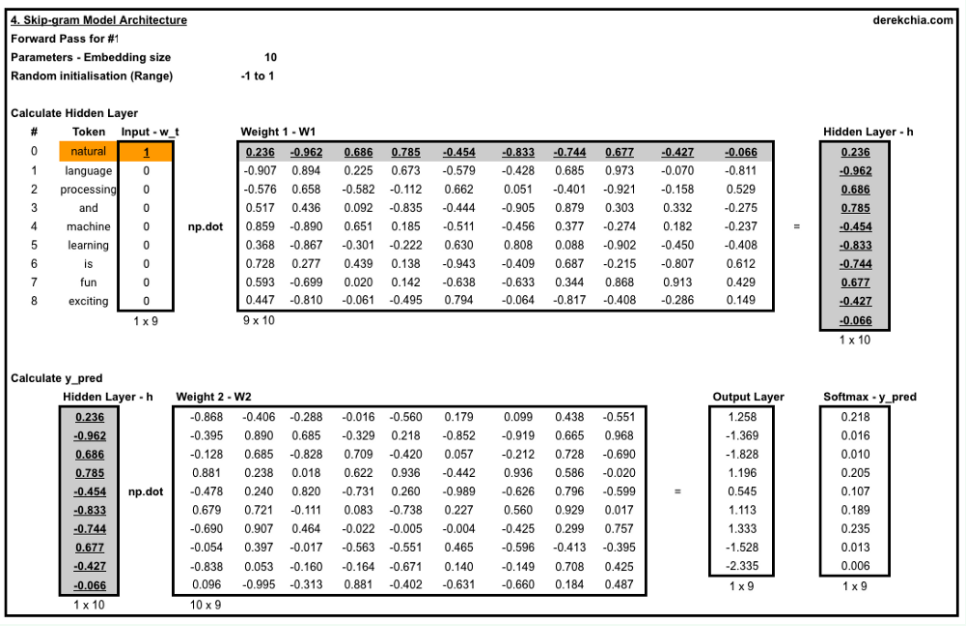


图6，计算隐藏层，输出层和softmax

训练——误差，反向传播和损失（loss）

误差——对于y_pred、h 和u，我们继续计算这组特定的目标词和上下文词的误差。这是通过对y_pred 与在w_c 中的每个上下文词之间的差的加合来实现的。

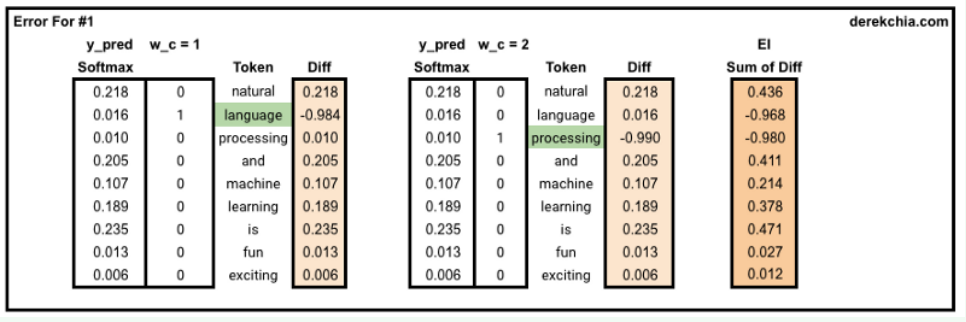


图7，计算误差——上下文单词是“language”和“processing”

反向传播——接下来，我们使用反向传播函数backprop，通过传入误差EI、隐藏层h 和目标字w_t 的向量，来计算我们所需的权重调整量。

To update the weights, we multiply the weights to be adjusted (dl_dw1 and dl_dw2) with learning rate and then subtract it from the current weights (w1 and w2).

为了更新权重，我们将权重的调整量（dl_dw1 和dl_dw2 ）与学习率相乘，然后从当前权重（w1 和w2 ）中减去它。

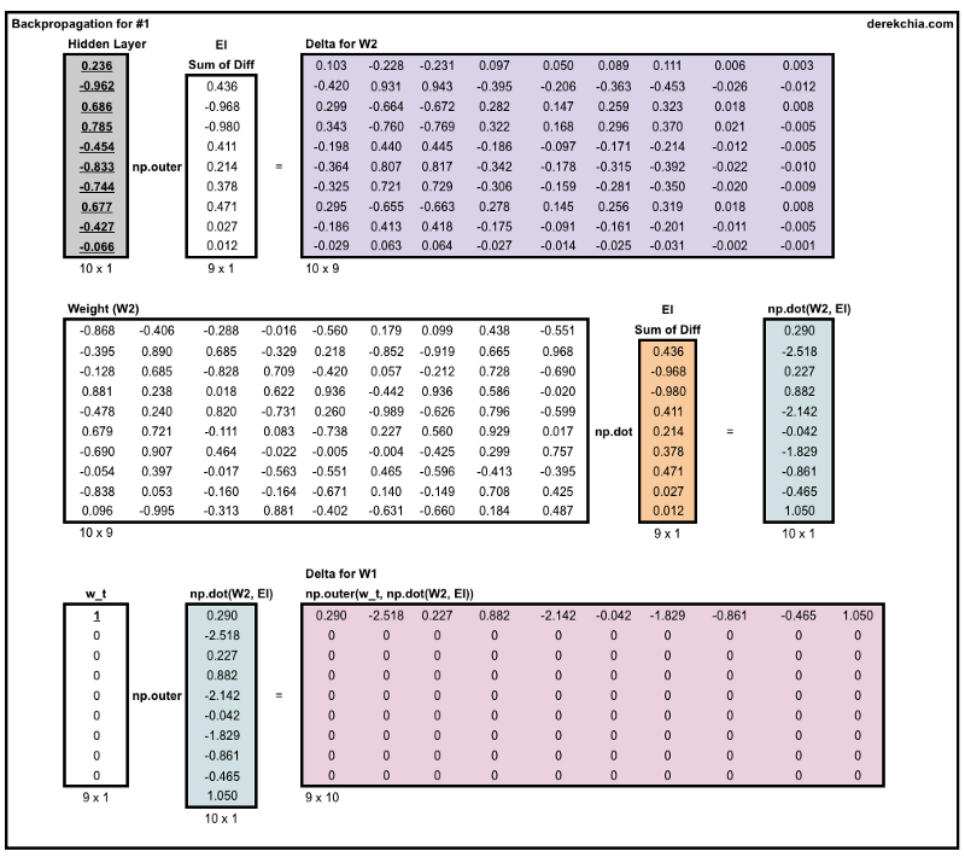


Fig. 8—Backpropagation—Calculating delta for W1 and W2

图8，反向传播——计算W1和W2的增量

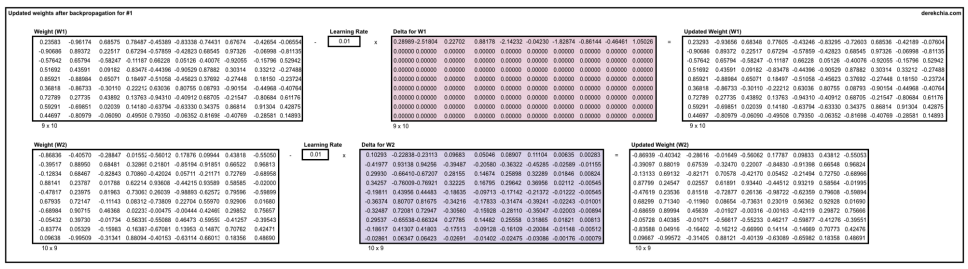


Fig. 9—Backpropagation—Adjusting weights to get updated W1 and W2

图9，反向传播——调整权重以得到更新后的W1和W2

```
class Word2Vec():
  ##Removed##

  for i in range(self.epochs):
    self.loss = 0
    for w_t, w_c in training_data:
    ##Removed##

      # Calculate error
      # 1. For a target word, calculate difference between y_pred and each of the context words
      # 2. Sum up the differences using np.sum to give us the error for this particular target wor
      EI = np.sum([np.subtract(y_pred, word) for word in w_c], axis=0)

      # Backpropagation
      # We use SGD to backpropagate errors - calculate loss on the output layer
      self.backprop(EI, h, w_t)

      # Calculate loss
      # There are 2 parts to the loss function
      # Part 1: -ve sum of all the output +
      # Part 2: length of context words * log of sum for all elements (exponential-ed) in the outp
      # Note: word.index(1) returns the index in the context word vector with value 1
      # Note: u[word.index(1)] returns the value of the output layer before softmax
      self.loss += -np.sum([u[word.index(1)] for word in w_c]) + len(w_c) * np.log(np.sum(np.exp(u
    print('Epoch:', i, "Loss:", self.loss)

  def backprop(self, e, h, x):
    # https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.outer.html
    # Column vector EI represents row-wise sum of prediction errors across each context word for t
    # Going backwards, we need to take derivative of E with respect of w2
    # h - shape 10x1, e - shape 9x1, dl_dw2 - shape 10x9
    dl_dw2 = np.outer(h, e)
    # x - shape 1x8, w2 - 5x8, e.T - 8x1
    # x - 1x8, np.dot() - 5x1, dl_dw1 - 8x5
    dl_dw1 = np.outer(x, np.dot(self.w2, e.T))
    # Update weights
    self.w1 = self.w1 - (self.lr * dl_dw1)
    self.w2 = self.w2 - (self.lr * dl_dw2)
```

```
class Word2Vec():
  ##Removed##

  for i in range(self.epochs):
    self.loss = 0
    for w_t, w_c in training_data:
    ##Removed##

      # Calculate error
      # 1. For a target word, calculate difference between y_pred and each of the context words
      # 2. Sum up the differences using np.sum to give us the error for this particular target wor
      EI = np.sum([np.subtract(y_pred, word) for word in w_c], axis=0)

      # Backpropagation
      # We use SGD to backpropagate errors - calculate loss on the output layer
      self.backprop(EI, h, w_t)

      # Calculate loss
      # There are 2 parts to the loss function
      # Part 1: -ve sum of all the output +
      # Part 2: length of context words * log of sum for all elements (exponential-ed) in the outp
      # Note: word.index(1) returns the index in the context word vector with value 1
      # Note: u[word.index(1)] returns the value of the output layer before softmax
      self.loss += -np.sum([u[word.index(1)] for word in w_c]) + len(w_c) * np.log(np.sum(np.exp(u
    print('Epoch:', i, "Loss:", self.loss)

  def backprop(self, e, h, x):
    # https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.outer.html
    # Column vector EI represents row-wise sum of prediction errors across each context word for t
    # Going backwards, we need to take derivative of E with respect of w2
    # h - shape 10x1, e - shape 9x1, dl_dw2 - shape 10x9
    dl_dw2 = np.outer(h, e)
    # x - shape 1x8, w2 - 5x8, e.T - 8x1
    # x - 1x8, np.dot() - 5x1, dl_dw1 - 8x5
    dl_dw1 = np.outer(x, np.dot(self.w2, e.T))
    # Update weights
    self.w1 = self.w1 - (self.lr * dl_dw1)
    self.w2 = self.w2 - (self.lr * dl_dw2)
```

Loss—Lastly, we compute the overall loss after finishing each training sample according to the loss function. Take note that the loss function comprises of 2 parts. The first part is the negative of the sum for all the

损失——最后，根据损失函数计算出每个训练样本完成后的总损失。注意，损失函数包括两个部分。第一部分是输出层（在softmax之前）中所有元素的和的负数。第二部分是上下文单词的数量乘以在输出层中所有元素（在 exp之后）之和的对数。

elements in the output layer (before softmax). The second part takes the number of the context words and multiplies the log of sum for all elements (after exponential) in the output layer.

$$
\begin{aligned}
E &= -\log p(w_{O,1}, w_{O,2}, \cdots, w_{O,C}|w_I) \\
&= -\log \prod_{c=1}^{C} \frac{\exp(u_{c,j_c^*})}{\sum_{j'=1}^{V} \exp(u_{j'})} \\
&= -\sum_{c=1}^{C} u_{j_c^*} + C \cdot \log \sum_{j'=1}^{V} \exp(u_{j'})
\end{aligned}
$$

5. Inferencing

Now that we have completed training for 50 epochs, both weights (w1 and w2) are now ready to perform inference.

Getting Vector for a Word

With a trained set of weights, the first thing we can do is to look at the word vector for a word in the vocabulary. We can simply do this by looking up the index of the word against the trained weight (w1). In the following example, we look up the vector for the word "machine".

```
# Get vector for word
vec = w2v.word_vec("machine")

class word2vec():
  ## Removed ##

  # Get vector from word
  def word_vec(self, word):
    w_index = self.word_index[word]
    v_w = self.w1[w_index]
    return v_w
```

w2v_get_vector.py hosted with ❤ by GitHub view raw

```
> print(w2v.word_vec("machine"))

[ 0.76702922 -0.95673743  0.49207258  0.16240808 -0.4538815
-0.74678226  0.42072706 -0.04147312  0.08947326 -0.24245257]
```

Find similar words

Another thing we can do is to find similar words. Even though our vocabulary is small, we can still implement the function vec_sim by computing the cosine similarity between words.

```
# Find similar words
w2v.vec_sim("machine", 3)

class word2vec():
  ## Removed##

  # Input vector, returns nearest word(s)
  def vec_sim(self, word, top_n):
    v_w1 = self.word_vec(word)
    word_sim = {}

    for i in range(self.v_count):
      # Find the similary score for each word in vocab
      v_w2 = self.w1[i]
      theta_sum = np.dot(v_w1, v_w2)
      theta_den = np.linalg.norm(v_w1) * np.linalg.norm(v_w2)
      theta = theta_sum / theta_den

      word = self.index_word[i]
      word_sim[word] = theta

    words_sorted = sorted(word_sim.items(), key=lambda kv: kv[1], reverse=True)

    for word, sim in words_sorted[:top_n]:
      print(word, sim)
```

w2v_find_similar_words.py hosted with ❤ by GitHub view raw

```
> w2v.vec_sim("machine", 3)

machine 1.0
fun 0.6223490454018772
and 0.5190154215400249
```

6. Further improvements

$$
\begin{aligned}
E &= -\log p(w_{O,1}, w_{O,2}, \cdots, w_{O,C}|w_I) \\
&= -\log \prod_{c=1}^{C} \frac{\exp(u_{c,j_c^*})}{\sum_{j'=1}^{V} \exp(u_{j'})}
\end{aligned}
$$

C V

5. 推论和总结（Inferencing）

既然我们已经完成了50个epoch的训练，两个权重（w1和w2）现在都准备好执行推论了。

获取单词的向量

有了一组训练后的权重，我们可以做的第一件事是查看词汇表中单词的词向量。我们可以简单地通过查找单词的索引来对训练后的权重（w1）进行查找。在下面的示例中，我们查找单词“machine”的向量。

```
# Get vector for word
vec = w2v.word_vec("machine")

class word2vec():
  ## Removed ##

  # Get vector from word
  def word_vec(self, word):
    w_index = self.word_index[word]
    v_w = self.w1[w_index]
    return v_w
```

w2v_get_vector.py hosted with ❤ by GitHub view raw

```
> print(w2v.word_vec("machine"))
[ 0.76702922 -0.95673743  0.49207258  0.16240808 -0.4538815
-0.74678226  0.42072706 -0.04147312  0.08947326 -0.24245257]
```

查询相似的单词

我们可以做的另一件事就是找到类似的单词。即使我们的词汇量很小，我们仍然可以通过计算单词之间的余弦相似度来实现函数vec_sim 。

```
# Find similar words
w2v.vec_sim("machine", 3)

class word2vec():
  ## Removed##

  # Input vector, returns nearest word(s)
  def vec_sim(self, word, top_n):
    v_w1 = self.word_vec(word)
    word_sim = {}

    for i in range(self.v_count):
      # Find the similary score for each word in vocab
      v_w2 = self.w1[i]
      theta_sum = np.dot(v_w1, v_w2)
      theta_den = np.linalg.norm(v_w1) * np.linalg.norm(v_w2)
      theta = theta_sum / theta_den

      word = self.index_word[i]
      word_sim[word] = theta

    words_sorted = sorted(word_sim.items(), key=lambda kv: kv[1], reverse=True)

    for word, sim in words_sorted[:top_n]:
      print(word, sim)
```

w2v_find_similar_words.py hosted with ❤ by GitHub view raw

```
> w2v.vec_sim("machine", 3)

machine 1.0
fun 0.6223490454018772
and 0.5190154215400249
```

If you are still reading the article, well done and thank you! But this is not the end. As you might have noticed in the backpropagation step above, we are required to adjust the weights for all other words that were not involved in the training sample. This process can take up a long time if the size of your vocabulary is large (e.g. tens of thousands).

To solve this, below are the two features in Word2Vec you can implement to speed things up:

Skip-gram Negative Sampling (SGNS) helps to speed up training time and improve quality of resulting word vectors. This is done by training the network to only modify a small percentage of the weights rather than all of them. Recall in our example above, we update the weights for every other word and this can take a very long time if the vocab size is large. With SGNS, we only need to update the weights for the target word and a small number (e.g. 5 to 20) of random 'negative' words.

Hierarchical Softmax is also another trick to speed up training time replacing the original softmax. The main idea is that instead of evaluating all the output nodes to obtain the probability distribution, we only need to evaluate about log (based 2) of it. It uses a binary tree (Huffman coding tree) representation where the nodes in the output layer are represented as leaves and its nodes are represented in relative probabilities to its child nodes.

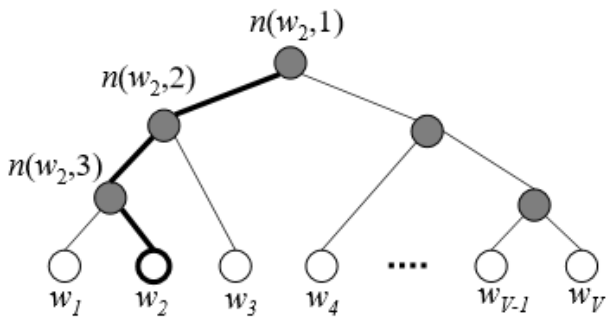


Fig. 11 — Hierarchical Binary Tree — Path from root to W2 is highlighted

Beyond that, why not try tweaking the code to implement the Continuous Bag-of-Words (CBOW) architecture? ?

06

Conclusion

This article is an introduction to Word2Vec and into the world of word embedding. It is also worth noting that there are pre-trained embeddings available such as GloVe, fastText and ELMo where you can download and use directly. There are also extensions of Word2Vec such as Doc2Vec and the most recent Code2Vec where documents and codes are turned into vectors. ?

Lastly, I want to thank to Ren Jie Tan, Raimi and Yuxin for taking time to comment and read the drafts of this. ?

References

nathanrooy/word2vec-from-scratch-with-python
A very simple, bare-bones, inefficient, implementation of skip-gram word2vec from scratch with Python ...github.com

Word2vec from Scratch with Python and NumPy
TL;DR - word2vec is awesome, it's also really simple. Learn how it works, and implement your own version. Since joining...nathanrooy.github.io

Why word2vec maximizes the cosine similarity between semantically similar words
Thanks for contributing an answer to Cross Validated! Some of your past answers have not been well-received, and you're...

6.进一步改进

如果你还在读这篇文章，做得好，谢谢！但这还没结束。正如你在上面的反向传播步骤中可能已经注意到的，我们需要调整训练样本中没有涉及的所有其他单词的权重。如果词汇量很大（例如数万），这个过程可能需要很长时间。

为了解决这个问题，您可以在Word2Vec中实现以下两个特性，以加快速度：

Skip-gram Negative Sampling (SGNS) 有助于加快训练时间，提高最终的词向量的质量。这是通过训练网络只修改一小部分的权重而不是全部的权重来实现。

回想一下上面的示例，我们对每一个词的权重都进行更新，若词汇库的尺寸很大，这可能需要很长时间。对于SGNS，我们只需要更新目标词和少量（例如，5到20）随机“否定”单词的权重。

Hierarchical Softmax是用来替换原始softmax加速训练的另一个技巧。其主要思想是，不需要对所有输出节点进行评估来获得概率分布，只需要评估它的对数个数（基为2）。使用二叉树（Huffman编码树）表示，其中输出层中的节点表示为叶子，其节点由与其子节点的相应的概率表示。

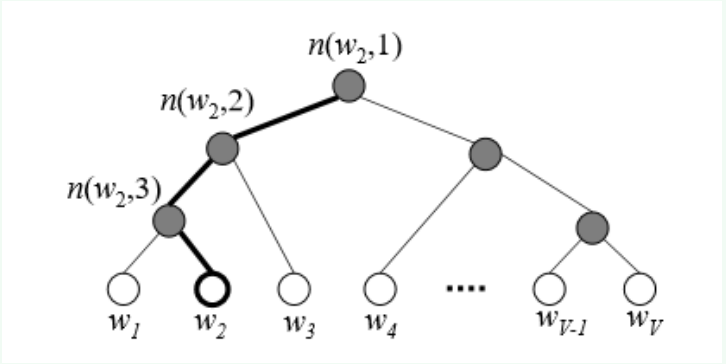


图11，Hierarchical二叉树，被高亮的为从根到W2的路径

除此之外，为什么不尝试调整代码来实现Continuous Bag-of-Words（Continuous Bag-of-Words，CBOW）构架呢？？

结论

本文是对Word2Vec的介绍，并解除了单词嵌入（word embedding）的世界。另外还值得注意的是，有预训练的嵌入可用，如GloVe、fastText和ELMo，你可以直接下载和使用。此外还有Word2Vec的扩展，如Doc2Vec和最近的Code2Vec，在这俩方法中文档和代码被转换成向量。

最后，我要感谢Ren Jie Tan、Raimi 和Yuxin抽出时间来阅读和评论本文的草稿。

参考

nathanrooy/word2vec-from-scratch-with-python
A very simple, bare-bones, inefficient, implementation of skip-gram word2vec from scratch with Python ...github.com

Word2vec from Scratch with Python and NumPy
TL;DR - word2vec is awesome, it's also really simple. Learn how it works, and implement your own version. Since joining...nathanrooy.github.io

Why word2vec maximizes the cosine similarity between semantically similar words
Thanks for contributing an answer to Cross Validated! Some of your past answers have not been well-received, and you're...

stats.stackexchange.com
Hierarchical softmax and negative sampling: short notes worth telling
Thanks to unexpected and very pleasant attention the audience has paid to my last (and the only) post here dedicated to…
towardsdatascience.com
Thanks to Ren Jie Tan and Raimi Bin Karim.

stats.stackexchange.com
Hierarchical softmax and negative sampling: short notes worth telling
Thanks to unexpected and very pleasant attention the audience has paid to my last (and the only) post here dedicated to…
towardsdatascience.com
感谢 Ren Jie Tan 和 Raimi Bin Karim.