吴恩达 深度学习 编程作业（5-2）Part 1 - Operations on word vectors

吴恩达 Coursera 课程 DeepLearning.ai 编程作业系列，本文为《序列模型》部分的第二周“NLP和词嵌入”的课程作业——第一部分：词向量运算。

另外，本节课程笔记在此：《吴恩达Coursera深度学习课程 DeepLearning.ai 提炼笔记（5-2）– NLP和词嵌入》，如有任何建议和问题，欢迎留言。

Operations on word vectors

Welcome to your first assignment of this week!

Because word embeddings are very computionally expensive to train, most ML practitioners will load a pre-trained set of embeddings.

After this assignment you will be able to:

- Load pre-trained word vectors, and measure similarity using cosine similarity
- Use word embeddings to solve word analogy problems such as Man is to Woman as King is to __.
- Modify word embeddings to reduce their gender bias

Let’s get started! Run the following cell to load the packages you will need.

```
import numpy as np
from w2v_utils import *
```

其中，`w2v_utils import` 中的函数如下所示：

```
from keras.models import Model
from keras.layers import Input, Dense, Reshape, merge
from keras.layers.embeddings import Embedding
from keras.preprocessing.sequence import skipgrams
from keras.preprocessing import sequence

import urllib.request
import collections
```

```python
import os
import zipfile

import numpy as np
import tensorflow as tf

window_size = 3
vector_dim = 300
epochs = 1000

valid_size = 16     # Random set of words to evaluate similarity on.
valid_window = 100  # Only pick dev samples in the head of the distribution.
valid_examples = np.random.choice(valid_window, valid_size, replace=False)

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


# 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


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

def collect_data(vocabulary_size=10000):
    url = 'http://mattmahoney.net/dc/'
    filename = maybe_download('text8.zip', url, 31344016)
    vocabulary = read_data(filename)
```

```
    vocabulary = read_data(filename)
    print(vocabulary[:7])
    data, count, dictionary, reverse_dictionary = build_dataset(vocabulary,
                                                                vocabulary_size)
    del vocabulary  # Hint to reduce memory.
    return data, count, dictionary, reverse_dictionary

class SimilarityCallback:
    def run_sim(self):
        for i in range(valid_size):
            valid_word = reverse_dictionary[valid_examples[i]]
            top_k = 8  # number of nearest neighbors
            sim = self._get_sim(valid_examples[i])
            nearest = (-sim).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)

    @staticmethod
    def _get_sim(valid_word_idx):
        sim = np.zeros((vocab_size,))
        in_arr1 = np.zeros((1,))
        in_arr2 = np.zeros((1,))
        in_arr1[0,] = valid_word_idx
        for i in range(vocab_size):
            in_arr2[0,] = i
            out = validation_model.predict_on_batch([in_arr1, in_arr2])
            sim[i] = out
        return sim


def read_glove_vecs(glove_file):
    with open(glove_file, 'r') as f:
        words = set()
        word_to_vec_map = {}

        for line in f:
            line = line.strip().split()
            curr_word = line[0]
            words.add(curr_word)
            word_to_vec_map[curr_word] = np.array(line[1:], dtype=np.float64)

    return words, word_to_vec_map

def relu(x):
    """
    Compute the relu of x

    Arguments:
    x -- A scalar or numpy array of any size.

    Return:
    s -- relu(x)
    """
    s = np.maximum(0,x)

    return s
```

```python
def initialize_parameters(vocab_size, n_h):
    """
    Arguments:
    layer_dims -- python array (list) containing the dimensions of each layer in our network

    Returns:
    parameters -- python dictionary containing your parameters "W1", "b1", "W2", "b2":
                    W1 -- weight matrix of shape (n_h, vocab_size)
                    b1 -- bias vector of shape (n_h, 1)
                    W2 -- weight matrix of shape (vocab_size, n_h)
                    b2 -- bias vector of shape (vocab_size, 1)
    """

    np.random.seed(3)
    parameters = {}

    parameters['W1'] = np.random.randn(n_h, vocab_size) / np.sqrt(vocab_size)
    parameters['b1'] = np.zeros((n_h, 1))
    parameters['W2'] = np.random.randn(vocab_size, n_h) / np.sqrt(n_h)
    parameters['b2'] = np.zeros((vocab_size, 1))

    return parameters

def softmax(x):
    """Compute softmax values for each sets of scores in x."""
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum()
```

```
Using TensorFlow backend.
```

相关数据集可在这里获取。

Next, lets load the word vectors. For this assignment, we will use 50-dimensional GloVe vectors to represent words. Run the following cell to load the `word_to_vec_map`.

```python
words, word_to_vec_map = read_glove_vecs('data/glove.6B.50d.txt')
```

You've loaded:
- `words`: set of words in the vocabulary.
- `word_to_vec_map`: dictionary mapping words to their GloVe vector representation.

You've seen that one-hot vectors do not do a good job cpaturing what words are similar. GloVe vectors provide much more useful information about the meaning of individual words. Lets now see how you can use GloVe vectors to decide how similar two words are.

1 - Cosine similarity

To measure how similar two words are, we need a way to measure the degree of similarity between two embedding vectors for the two words. Given two vectors : u and v, cosine similarity is defined as follows

$u.v$
is
the
dot
pro
duct
(or
inne
r
pro
duct
) of
two
vect
ors,
where $||u||$

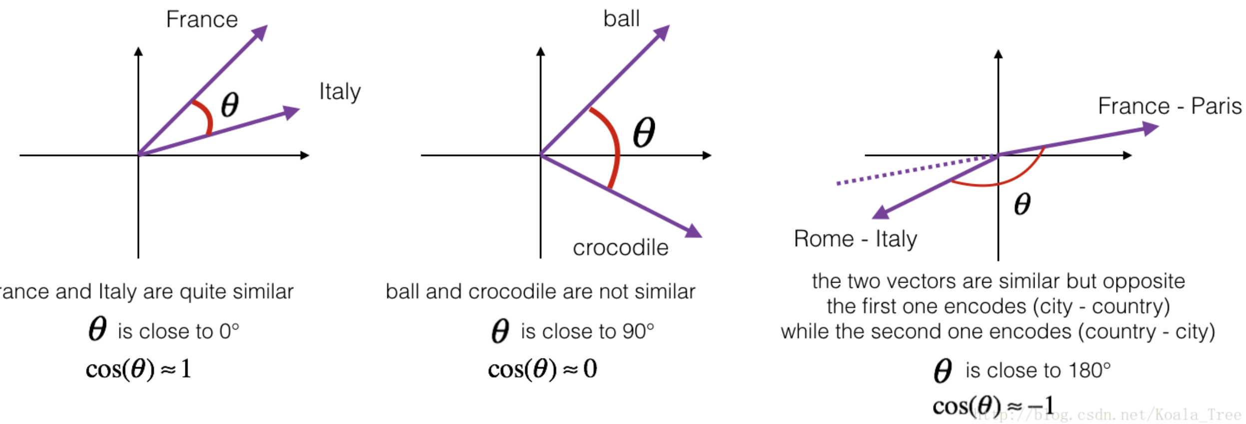


Figure 1: The cosine of the angle between two vectors is a measure of how similar they are

Exercise: Implement the function `cosine_similarity()` to evaluate similarity between word vectors.

u
i
s
d
e
fi
n
e
d
a
s
$=$
Reminder: The norm of

```python
# GRADED FUNCTION: cosine_similarity

def cosine_similarity(u, v):
    """
    Cosine similarity reflects the degree of similariy between u and v

    Arguments:
        u -- a word vector of shape (n,)
        v -- a word vector of shape (n,)

    Returns:
        cosine_similarity -- the cosine similarity between u and v defined by the formula above.
    """

    distance = 0.0

    ### START CODE HERE ###
    # Compute the dot product between u and v (≈1 line)
    dot = np.dot(u, v)
    # Compute the L2 norm of u (≈1 line)
    norm_u = np.sqrt(np.sum(u**2))

    # Compute the L2 norm of v (≈1 line)
    norm_v = np.sqrt(np.sum(v**2))
    # Compute the cosine similarity defined by formula (1) (≈1 line)
    cosine_similarity = dot / (norm_u * norm_v)
    ### END CODE HERE ###

    return cosine_similarity
```

```python
father = word_to_vec_map["father"]
mother = word_to_vec_map["mother"]
ball = word_to_vec_map["ball"]
crocodile = word_to_vec_map["crocodile"]
france = word_to_vec_map["france"]
italy = word_to_vec_map["italy"]
paris = word_to_vec_map["paris"]
rome = word_to_vec_map["rome"]

print("cosine_similarity(father, mother) = ", cosine_similarity(father, mother))
print("cosine_similarity(ball, crocodile) = ",cosine_similarity(ball, crocodile))
print("cosine_similarity(france - paris, rome - italy) = ",cosine_similarity(france - paris, rome - ita
```

```
cosine_similarity(father, mother) =  0.890903844289
cosine_similarity(ball, crocodile) =  0.274392462614
cosine_similarity(france - paris, rome - italy) =  -0.675147930817
```

Expected Output:

cosine_similarity(father, mother) =	0.890903844289
cosine_similarity(ball, crocodile) =	0.274392462614
cosine_similarity(france - paris, rome - italy) =	-0.675147930817

After you get the correct expected output, please feel free to modify the inputs and measure the cosine similarity between other pairs of words! Playing around the cosine similarity of other inputs will give you a better sense of how word vectors behave.

2 - Word analogy task

In the word analogy task, we complete the sentence "*a* is to *b* as *c* is to __". An example is '*man* is to *woman* as *king* is to *queen*' . In detail, we are trying to find a word *d*, such that the associated word vectors

Exercise: Complete the code below to be able to perform word analogies!

```
# GRADED FUNCTION: complete_analogy

def complete_analogy(word_a, word_b, word_c, word_to_vec_map):
    """
    Performs the word analogy task as explained above: a is to b as c is to ____.

    Arguments:
    word_a -- a word, string
    word_b -- a word, string
    word_c -- a word, string
    word_to_vec_map -- dictionary that maps words to their corresponding vectors.

    Returns:
    best_word --  the word such that v_b - v_a is close to v_best_word - v_c, as measured by cosine sim
    """

    # convert words to lower case
    word_a, word_b, word_c = word_a.lower(), word_b.lower(), word_c.lower()

    ### START CODE HERE ###
    # Get the word embeddings v_a, v_b and v_c (≈1-3 lines)
    e_a, e_b, e_c = word_to_vec_map[word_a], word_to_vec_map[word_b], word_to_vec_map[word_c]
    ### END CODE HERE ###

    words = word_to_vec_map.keys()
    max_cosine_sim = -100              # Initialize max_cosine_sim to a large negative number
    best_word = None                   # Initialize best_word with None, it will help keep track of the

    # loop over the whole word vector set
    for w in words:
        # to avoid best_word being one of the input words, pass on them.
        if w in [word_a, word_b, word_c] :
            continue

        ### START CODE HERE ###
        # Compute cosine similarity between the vector (e_b - e_a) and the vector ((w's vector represen
        cosine_sim = cosine_similarity(e_b - e_a, word_to_vec_map[w] - e_c)

        # If the cosine_sim is more than the max_cosine_sim seen so far,
            # then: set the new max_cosine_sim to the current cosine_sim and the best_word to the curre
        if cosine_sim > max_cosine_sim:
            max_cosine_sim = cosine_sim
            best_word = w
        ### END CODE HERE ###

    return best_word
```

Run the cell below to test your code, this may take 1-2 minutes.

```
triads_to_try = [('italy', 'italian', 'spain'), ('india', 'delhi', 'japan'), ('man', 'woman', 'boy'), (
for triad in triads_to_try:
    print ('{} -> {} :: {} -> {}'.format( *triad, complete_analogy(*triad,word_to_vec_map)))
```

```
italy -> italian :: spain -> spanish
india -> delhi :: japan -> tokyo
man -> woman :: boy -> girl
small -> smaller :: large -> larger
```

Expected Output:

italy -> italian ::	spain -> spanish
india -> delhi ::	japan -> tokyo
**man -> woman ** ::	boy -> girl
**small -> smaller ** ::	large -> larger

Once you get the correct expected output, please feel free to modify the input cells above to test your own analogies. Try to find some other analogy pairs that do work, but also find some where the algorithm doesn't give the right answer: For example, you can try small->smaller as big->?.

Congratulations!

You've come to the end of this assignment. Here are the main points you should remember:

- Cosine similarity a good way to compare similarity between pairs of word vectors. (Though L2 distance works too.)
- For NLP applications, using a pre-trained set of word vectors from the internet is often a good way to get started.

Even though you have finished the graded portions, we recommend you take a look too at the rest of this notebook.

Congratulations on finishing the graded portions of this notebook!

3 - Debiasing word vectors (OPTIONAL/UNGRADED)

In the following exercise, you will examine gender biases that can be reflected in a word embedding, and explore algorithms for reducing the bias. In addition to learning about the topic of debiasing, this exercise will also help hone your intuition about what word vectors are doing. This section involves a bit of linear algebra, though you can probably complete it even without being expert in linear algebra, and we encourage you to give it a shot. This portion of the notebook is optional and is not graded.

Lets first see how the GloVe word embeddings relate to gender. You will first compute a vector

```
g = word_to_vec_map['woman'] - word_to_vec_map['man']
print(g)
```

```
[-0.087144    0.2182     -0.40986    -0.03922    -0.1032      0.94165
 -0.06042     0.32988     0.46144    -0.35962     0.31102    -0.86824
  0.96006     0.01073     0.24337     0.08193    -1.02722    -0.21122
  0.695044   -0.00222     0.29106     0.5053     -0.099454    0.40445
  0.30181     0.1355     -0.0606     -0.07131    -0.19245    -0.06115
 -0.3204      0.07165    -0.13337    -0.25068714 -0.14293    -0.224957
 -0.149       0.048882    0.12191    -0.27362    -0.165476   -0.20426
  0.54376    -0.271425   -0.10245    -0.32108     0.2516     -0.33455
 -0.04371     0.01258   ]
```

Now, you will consider the cosine similarity of different words with g

```
print ('List of names and their similarities with constructed vector:')

# girls and boys name
name_list = ['john', 'marie', 'sophie', 'ronaldo', 'priya', 'rahul', 'danielle', 'reza', 'katy', 'yasmi

for w in name_list:
    print (w, cosine_similarity(word_to_vec_map[w], g))
```

```
List of names and their similarities with constructed vector:
john -0.23163356146
marie 0.315597935396
sophie 0.318687898594
ronaldo -0.312447968503
priya 0.17632041839
rahul -0.169154710392
danielle 0.243932992163
reza -0.079304296722
katy 0.283106865957
yasmin 0.233138577679
```

As you can see, female first names tend to have a positive cosine similarity with our constructed vector g, while male first names tend to have a negative cosine similarity. This is not suprising, and the result seems acceptable.

But let’s try with some other words.

```
print('Other words and their similarities:')
word_list = ['lipstick', 'guns', 'science', 'arts', 'literature', 'warrior','doctor', 'tree', 'receptio
             'technology',  'fashion', 'teacher', 'engineer', 'pilot', 'computer', 'singer']
for w in word_list:
    print (w, cosine_similarity(word_to_vec_map[w], g))
```

```
Other words and their similarities:
lipstick 0.276919162564
guns -0.18884855679
science -0.0608290654093
arts 0.00818931238588
literature 0.0647250443346
warrior -0.209201646411
doctor 0.118952894109
tree -0.0708939917548
receptionist 0.330779417506
technology -0.131937324476
fashion 0.0356389462577
teacher 0.179209234318
engineer -0.0803928049452
pilot 0.00107644989919
computer -0.103303588739
singer 0.185005181365
```

Do you notice anything surprising? It is astonishing how these results reflect certain unhealthy gender stereotypes. For example, “computer” is closer to “man” while “literature” is closer to “woman”. Ouch!

We'll see below how to reduce the bias of these vectors, using an algorithm due to Boliukbasi et al., 2016. Note that some word pairs such as "actor"/"actress" or "grandmother"/"grandfather" should remain gender specific, while other words such as "receptionist" or "technology" should be neutralized, i.e. not be gender-related. You will have to treat these two type of words differently when debiasing.

3.1 - Neutralize bias for non-gender specific words

The figure below should help you visualize what neutralizing does. If you're using a 50-dimensional word embedding, the 50 dimensional space can be split into two parts: The bias-direction g

Even though is

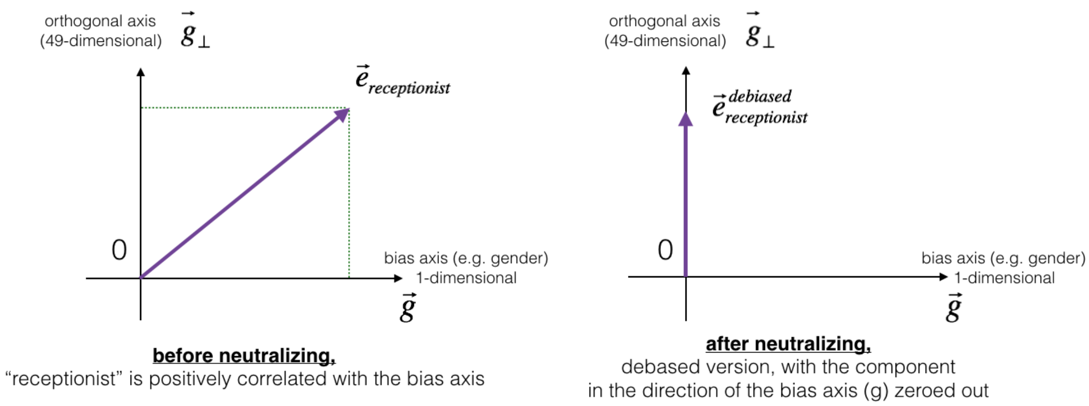


Figure 2: The word vector for "receptionist" represented before and after applying the neutralize operation.

Exercise: Implement `neutralize()` to remove the bias of words such as "receptionist" or "scientist". Given an input embedding : e, you can use the following formulas to compute

If you are an expert in linear algebra, you may recognize e_{-}

```python
def neutralize(word, g, word_to_vec_map):
    """
    Removes the bias of "word" by projecting it on the space orthogonal to the bias axis.
    This function ensures that gender neutral words are zero in the gender subspace.

    Arguments:
        word -- string indicating the word to debias
        g -- numpy-array of shape (50,), corresponding to the bias axis (such as gender)
        word_to_vec_map -- dictionary mapping words to their corresponding vectors.

    Returns:
        e_debiased -- neutralized word vector representation of the input "word"
    """

    ### START CODE HERE ###
    # Select word vector representation of "word". Use word_to_vec_map. (≈ 1 line)
    e = word_to_vec_map[word]

    # Compute e_biascomponent using the formula give above. (≈ 1 line)
    e_biascomponent = np.dot(e, g) / np.square(np.linalg.norm(g)) * g

    # Neutralize e by substracting e_biascomponent from it
    # e_debiased should be equal to its orthogonal projection. (≈ 1 line)
    e_debiased = e - e_biascomponent
    ### END CODE HERE ###

    return e_debiased
```

```python
e = "receptionist"
print("cosine similarity between " + e + " and g, before neutralizing: ", cosine_similarity(word_to_vec

e_debiased = neutralize("receptionist", g, word_to_vec_map)
print("cosine similarity between " + e + " and g, after neutralizing: ", cosine_similarity(e_debiased,
```

```
cosine similarity between receptionist and g, before neutralizing:  0.330779417506
cosine similarity between receptionist and g, after neutralizing:  -3.26732746085e-17
```

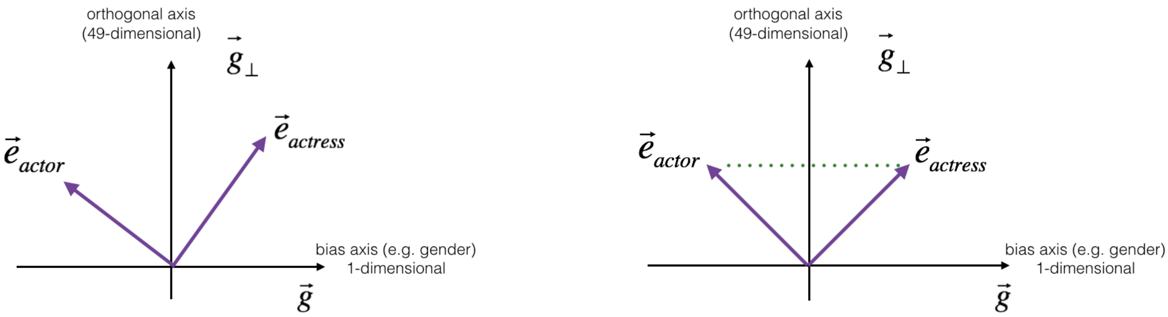
Expected Output: The second result is essentially 0, up to numerical roundof (on the order of 10^7

cosine similarity between receptionist and g, before neutralizing: :	0.330779417506
cosine similarity between receptionist and g, after neutralizing: :	-3.26732746085e-17

3.2 - Equalization algorithm for gender-specific words

Next, lets see how debiasing can also be applied to word pairs such as “actress” and “actor.” Equalization is applied to pairs of words that you might want to have differ only through the gender property. As a concrete example, suppose that “actress” is closer to “babysit” than “actor.” By applying neutralizing to “babysit” we can reduce the gender-stereotype associated with babysitting. But this still does not guarantee that “actor” and “actress” are equidistant from “babysit.” The equalization algorithm takes care of this.

The key idea behind equalization is to make sure that a particular pair of words are equi-distant from the 49-dimensional equaliza



before equalizing,
"actress" and "actor" differ
in many ways beyond the
direction of $\vec{g}$

after equalizing,
"actress" and "actor" differ
only in the direction of $\vec{g}$, and further
are equal in distance from $\vec{g}_{\perp}$

The derivation of the linear algebra to do this is a bit more complex. (See Bolukbasi et al., 2016 for details.) But the key equations are:

Exercise: Implement the function below. Use the equations above to get the final equalized version of the pair of words. Good luck!

```python
def equalize(pair, bias_axis, word_to_vec_map):
    """
    Debias gender specific words by following the equalize method described in the figure above.

    Arguments:
    pair -- pair of strings of gender specific words to debias, e.g. ("actress", "actor")
    bias_axis -- numpy-array of shape (50,), vector corresponding to the bias axis, e.g. gender
    word_to_vec_map -- dictionary mapping words to their corresponding vectors

    Returns
    e_1 -- word vector corresponding to the first word
    e_2 -- word vector corresponding to the second word
    """

    ### START CODE HERE ###
    # Step 1: Select word vector representation of "word". Use word_to_vec_map. (≈ 2 lines)
    w1, w2 = pair
    e_w1, e_w2 = word_to_vec_map[w1], word_to_vec_map[w2]

    # Step 2: Compute the mean of e_w1 and e_w2 (≈ 1 line)
    mu = (e_w1 + e_w2) / 2

    # Step 3: Compute the projections of mu over the bias axis and the orthogonal axis (≈ 2 lines)
    mu_B = np.dot(mu, bias_axis) / np.sum(bias_axis**2) * bias_axis
    mu_orth = mu - mu_B

    # Step 4: Use equations (7) and (8) to compute e_w1B and e_w2B (≈2 lines)
    e_w1B = np.dot(e_w1, bias_axis) / np.sum(bias_axis**2) * bias_axis
    e_w2B = np.dot(e_w2, bias_axis) / np.sum(bias_axis**2) * bias_axis

    # Step 5: Adjust the Bias part of e_w1B and e_w2B using the formulas (9) and (10) given above (≈2 l
    corrected_e_w1B = np.sqrt(np.abs(1-np.sum(mu_orth**2))) * (e_w1B - mu_B)/np.linalg.norm(e_w1-mu_ort
    corrected_e_w2B =np.sqrt(np.abs(1-np.sum(mu_orth**2))) * (e_w2B - mu_B)/np.linalg.norm(e_w2-mu_orth

    # Step 6: Debias by equalizing e1 and e2 to the sum of their corrected projections (≈2 lines)
    e1 = corrected_e_w1B + mu_orth
    e2 = corrected_e_w2B + mu_orth

    ### END CODE HERE ###

    return e1, e2
```

```python
print("cosine similarities before equalizing:")
print("cosine_similarity(word_to_vec_map[\"man\"], gender) = ", cosine_similarity(word_to_vec_map["man"
print("cosine_similarity(word_to_vec_map[\"woman\"], gender) = ", cosine_similarity(word_to_vec_map["wo
print()
e1, e2 = equalize(("man", "woman"), g, word_to_vec_map)
print("cosine similarities after equalizing:")
print("cosine_similarity(e1, gender) = ", cosine_similarity(e1, g))
print("cosine_similarity(e2, gender) = ", cosine_similarity(e2, g))
```

```
cosine similarities before equalizing:
cosine_similarity(word_to_vec_map["man"], gender) =  -0.117110957653
cosine_similarity(word_to_vec_map["woman"], gender) =  0.356666188463

cosine similarities after equalizing:
cosine_similarity(e1, gender) =  -0.700436428931
cosine_similarity(e2, gender) =  0.700436428931
```

Expected Output:

cosine similarities before equalizing:

cosine_similarity(word_to_vec_map[“man”], gender) =	-0.117110957653
cosine_similarity(word_to_vec_map[“woman”], gender) =	0.356666188463

cosine similarities after equalizing:

cosine_similarity(u1, gender) =	-0.700436428931
cosine_similarity(u2, gender) =	0.700436428931

Please feel free to play with the input words in the cell above, to apply equalization to other pairs of words.

These debiasing algorithms are very helpful for reducing bias, but are not perfect and do not eliminate all traces of bias. For example, one weakness of this implementation was that the bias direction g

Congratulations

You have come to the end of this notebook, and have seen a lot of the ways that word vectors can be used as well as modified.

Congratulations on finishing this notebook!

References:
- The debiasing algorithm is from Bolukbasi et al., 2016, Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings
- The GloVe word embeddings were due to Jeffrey Pennington, Richard Socher, and Christopher D. Manning. (https://nlp.stanford.edu/projects/glove/)