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.

Chollet has the view that pre-trained embeddings dont fit the specific task very well

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.

In [1]:

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

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`.

In [2]:

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

```
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
```

no need to close
file using f.close()
when using with

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.

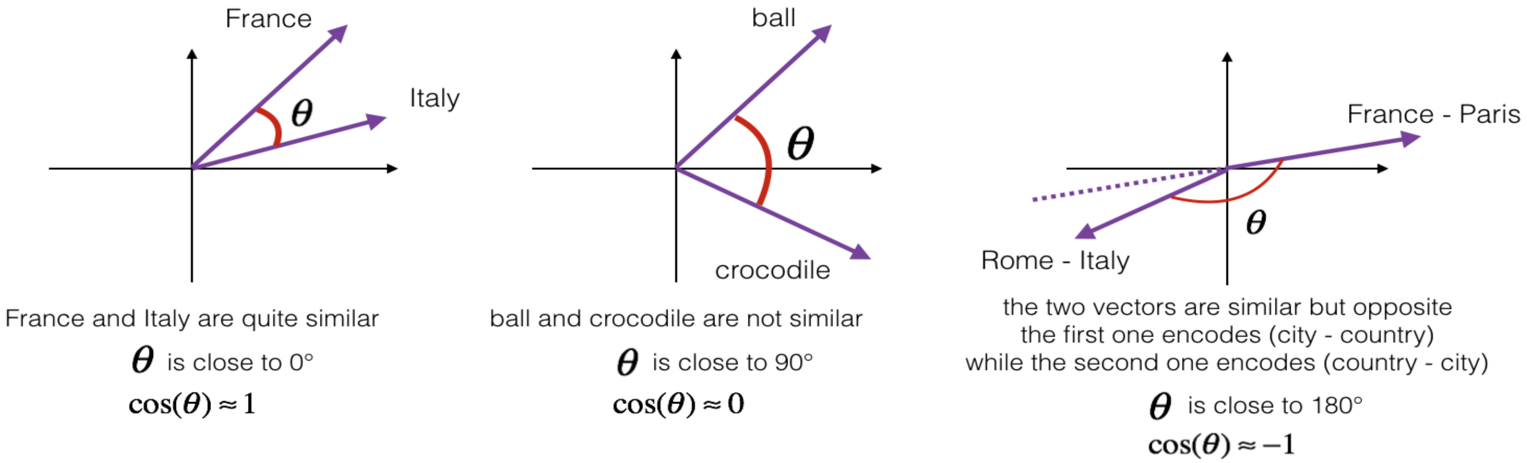
.strip() removes all whitespace at the start and end, including spaces, tabs, newlines and carriage returns.
Consider a line like:
"foo\tbar "
In this case, if you strip, then you'll get {"foo":"bar"} as the dictionary entry.
If you don't strip, you'll get {"foo":"bar "} (note the extra space at the end)

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:

$$\text{CosineSimilarity(u, v)} = \frac{u . v}{||u||_2 ||v||_2} = cos(\theta) \tag{1}$$

where $u.v$ is the dot product (or inner product) of two vectors, $||u||_2$ is the norm (or length) of the vector u, and θ is the angle between u and v. This similarity depends on the angle between u and v. If u and v are very similar, their cosine similarity will be close to 1; if they are dissimilar, the cosine similarity will take a smaller value.



France and Italy are quite similar
θ is close to 0°
$\cos(\theta) \approx 1$

ball and crocodile are not similar
θ is close to 90°
$\cos(\theta) \approx 0$

the two vectors are similar but opposite
the first one encodes (city - country)
while the second one encodes (country - city)
θ is close to 180°
$\cos(\theta) \approx -1$

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.

Reminder: The norm of u is defined as $||u||_2 = \sqrt{\sum_{i=1}^{n} u_i^2}$

In [3]:
```
# 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 = numpy.linalg.norm(u, axis=None,keepdims=True )
    norm_u = np.linalg.norm(u)
    # Compute the L2 norm of v (≈1 line)
    #norm_v = numpy.linalg.norm(u, axis=None, keepdims=True)
    norm_v = np.linalg.norm(v)
    # Compute the cosine similarity defined by formula (1) (≈1 line)
    cosine_similarity = dot/(norm_u*norm_v)
    ### END CODE HERE ###

    return cosine_similarity
```

In [4]:
```
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 - italy))
```

```
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 e_a, e_b, e_c, e_d are related in the following manner: $e_b - e_a \approx e_d - e_c$. We will measure the similarity between $e_b - e_a$ and $e_d - e_c$ using cosine similarity.

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

In [25]:

```
# 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 similarity
    """

    # 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 word to output

    # 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 representation) - e_c)  (≈1 line)
        cosine_sim = cosine_similarity(word_to_vec_map[w] - e_c, e_b - e_a)

        # 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 current word (≈3 lines)
        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.

```
In [27]: triads_to_try = [('italy', 'italian', 'spain'), ('india', 'delhi', 'japan'), ('man', 'woman', 'boy'), ('small', 'smaller', 'large')]
         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
```

print("Iteration %d :" % (iteration +1))

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 $g = e_{woman} - e_{man}$, where e_{woman} represents the word vector corresponding to the word *woman*, and e_{man} corresponds to the word vector corresponding to the word *man*. The resulting vector g roughly encodes the concept of "gender". (You might get a more accurate representation if you compute $g_1 = e_{mother} - e_{father}$, $g_2 = e_{girl} - e_{boy}$, etc. and average over them. But just using $e_{woman} - e_{man}$ will give good enough results for now.)

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

Now, you will consider the cosine similarity of different words with g. Consider what a positive value of similarity means vs a negative cosine similarity.

In [ ]:

```
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', 'yasmin']

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

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.

In [ ]:

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

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 (https://arxiv.org/abs/1607.06520). 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, and the remaining 49 dimensions, which we'll call $g_{\perp}$. In linear algebra, we say that the 49 dimensional $g_{\perp}$ is perpendicular (or "othogonal") to g, meaning it is at 90 degrees to g. The neutralization step takes a vector such as $e_{receptionist}$ and zeros out the component in the direction of g, giving us $e_{receptionist}^{debiased}$.

Even though $g_{\perp}$ is 49 dimensional, given the limitations of what we can draw on a screen, we illustrate it using a 1 dimensional axis below.

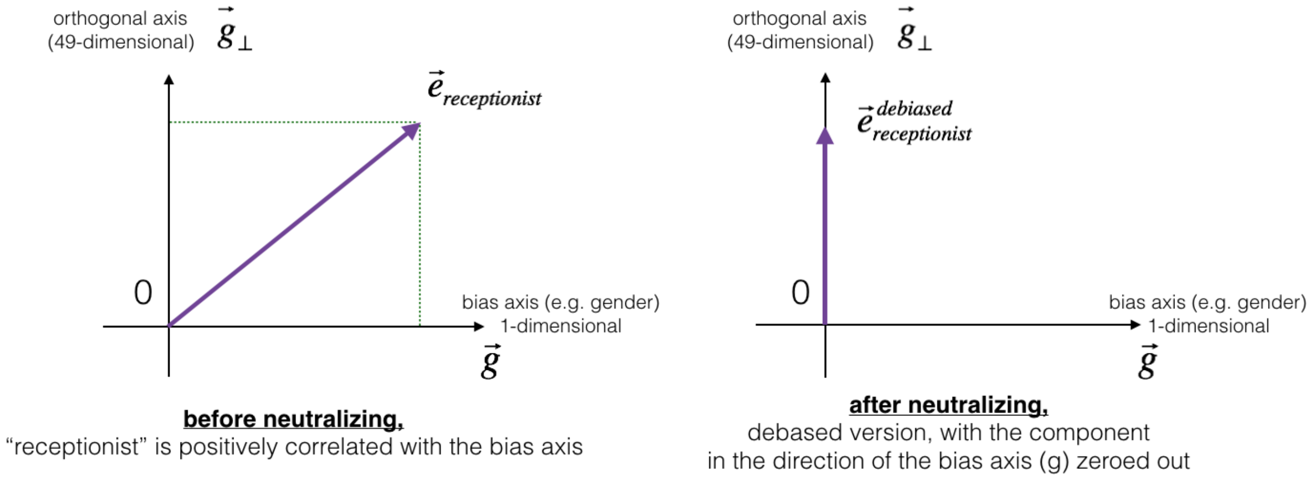

orthogonal axis (49-dimensional) $\vec{g}_{\perp}$
$\vec{e}_{receptionist}$
0
bias axis (e.g. gender) 1-dimensional
$\vec{g}$
before neutralizing,
"receptionist" is positively correlated with the bias axis
orthogonal axis (49-dimensional) $\vec{g}_{\perp}$
$\vec{e}_{receptionist}^{debiased}$
0
bias axis (e.g. gender) 1-dimensional
$\vec{g}$
after neutralizing,
debased version, with the component in the direction of the bias axis (g) zeroed out

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 $e^{debiased}$:

$$e^{bias\_component} = \frac{e \cdot g}{||g||_2^2} * g \tag{2}$$

$$e^{debiased} = e - e^{bias\_component} \tag{3}$$

If you are an expert in linear algebra, you may recognize $e^{bias\_component}$ as the projection of e onto the direction g. If you're not an expert in linear algebra, don't worry about this.

In [ ]:

```
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 = None

    # Compute e_biascomponent using the formula give above. (≈ 1 line)
    e_biascomponent = None

    # Neutralize e by substracting e_biascomponent from it
    # e_debiased should be equal to its orthogonal projection. (≈ 1 line)
    e_debiased = None
    ### END CODE HERE ###

    return e_debiased
```

In [ ]:

```
e = "receptionist"
print("cosine similarity between " + e + " and g, before neutralizing: ", cosine_similarity(word_to_vec_map["receptionist"], g))

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

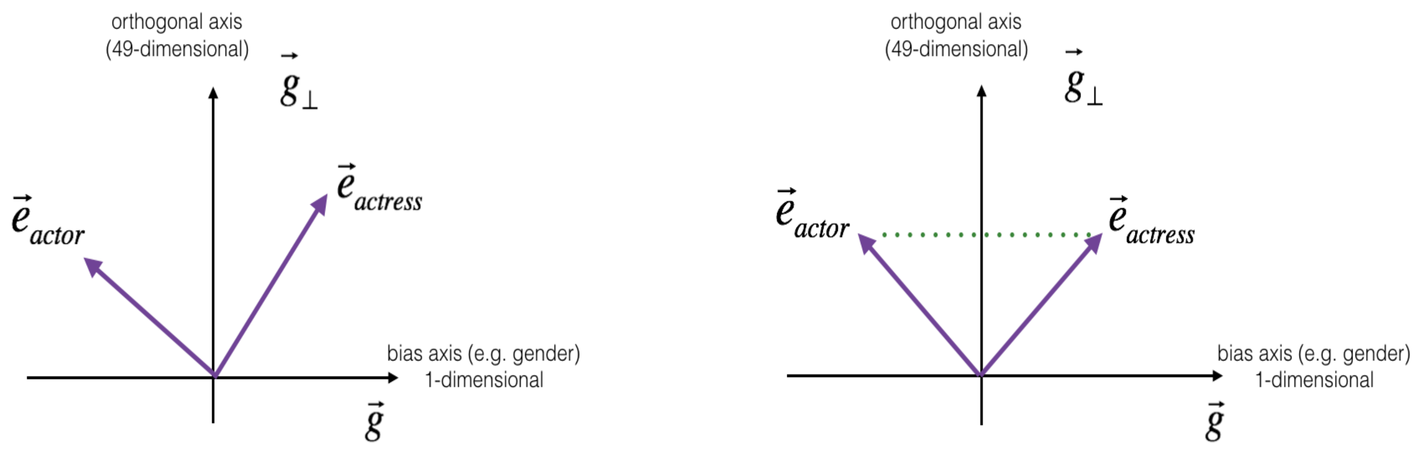
Expected Output: The second result is essentially 0, up to numerical roundof (on the order of 10^{-17}).

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 $g_\perp$. The equalization step also ensures that the two equalized steps are now the same distance from $e_{receptionist}^{debiased}$, or from any other work that has been neutralized. In pictures, this is how equalization works:



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:

$$\mu = \frac{e_{w1} + e_{w2}}{2} \tag{4}$$

$$\mu_{B} = \frac{\mu \cdot \text{bias\_axis}}{||\text{bias\_axis}||_2^2} * \text{bias\_axis} \tag{5}$$

$$\mu_{\perp} = \mu - \mu_{B} \tag{6}$$

$$e_{w1B} = \frac{e_{w1} \cdot \text{bias\_axis}}{||\text{bias\_axis}||_2^2} * \text{bias\_axis} \tag{7}$$

$$e_{w2B} = \frac{e_{w2} \cdot \text{bias\_axis}}{||\text{bias\_axis}||_2^2} * \text{bias\_axis} \tag{8}$$

$$e_{w1B}^{corrected} = \sqrt{|1 - ||\mu_{\perp}||_2^2|} * \frac{e_{\text{w1B}} - \mu_B}{|(e_{w1} - \mu_{\perp}) - \mu_B)|} \tag{9}$$

$$e_{w2B}^{corrected} = \sqrt{|1 - ||\mu_{\perp}||_2^2|} * \frac{e_{\text{w2B}} - \mu_B}{|(e_{w2} - \mu_{\perp}) - \mu_B)|} \tag{10}$$

$$e_1 = e_{w1B}^{corrected} + \mu_{\perp} \tag{11}$$

$$e_2 = e_{w2B}^{corrected} + \mu_{\perp} \tag{12}$$

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

In [ ]:

```
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 = None
    e_w1, e_w2 = None

    # Step 2: Compute the mean of e_w1 and e_w2 (≈ 1 line)
    mu = None

    # Step 3: Compute the projections of mu over the bias axis and the orthogonal axis (≈ 2 lines)
    mu_B = None
    mu_orth = None

    # Step 4: Use equations (7) and (8) to compute e_w1B and e_w2B (≈2 lines)
    e_w1B = None
    e_w2B = None

    # Step 5: Adjust the Bias part of e_w1B and e_w2B using the formulas (9) and (10) given above (≈2 lines)
    corrected_e_w1B = None
    corrected_e_w2B = None

    # Step 6: Debias by equalizing e1 and e2 to the sum of their corrected projections (≈2 lines)
    e1 = None
    e2 = None

    ### END CODE HERE ###

    return e1, e2
```

In [ ]:

```
print("cosine similarities before equalizing:")
print("cosine_similarity(word_to_vec_map[\"man\"], gender) = ", cosine_similarity(word_to_vec_map["man"], g))
print("cosine_similarity(word_to_vec_map[\"woman\"], gender) = ", cosine_similarity(word_to_vec_map["woman"], g))
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))
```

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 was defined using only the pair of words *woman* and *man*. As discussed earlier, if g were defined by computing $g_1 = e_{woman} - e_{man}$; $g_2 = e_{mother} - e_{father}$; $g_3 = e_{girl} - e_{boy}$; and so on and averaging over them, you would obtain a better estimate of the "gender" dimension in the 50 dimensional word embedding space. Feel free to play with such variants as well.

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 (https://papers.nips.cc/paper/6228-man-is-to-computer-programmer-as-woman-is-to-homemaker-debiasing-word-embeddings.pdf)
- The GloVe word embeddings were due to Jeffrey Pennington, Richard Socher, and Christopher D. Manning. (https://nlp.stanford.edu/projects/glove/ (https://nlp.stanford.edu/projects/glove/))