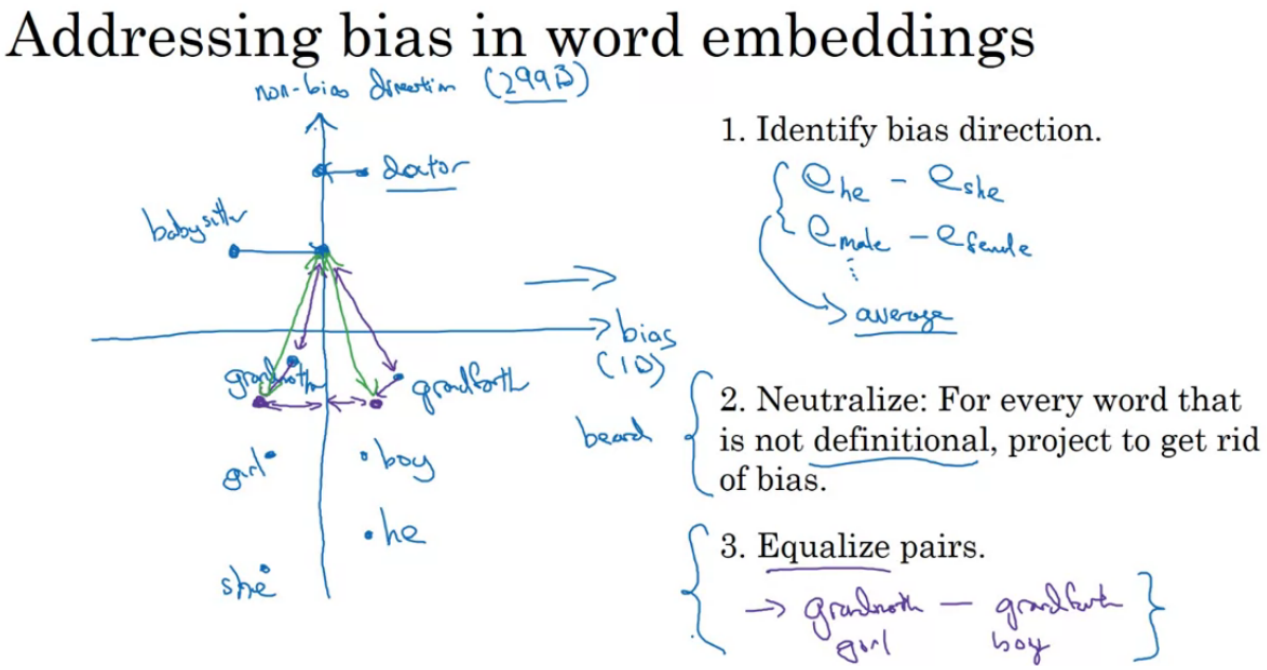
Sequence Models — Week 02

aakashgoel12.medium.com/sequence-models-week-02-a34889901eef

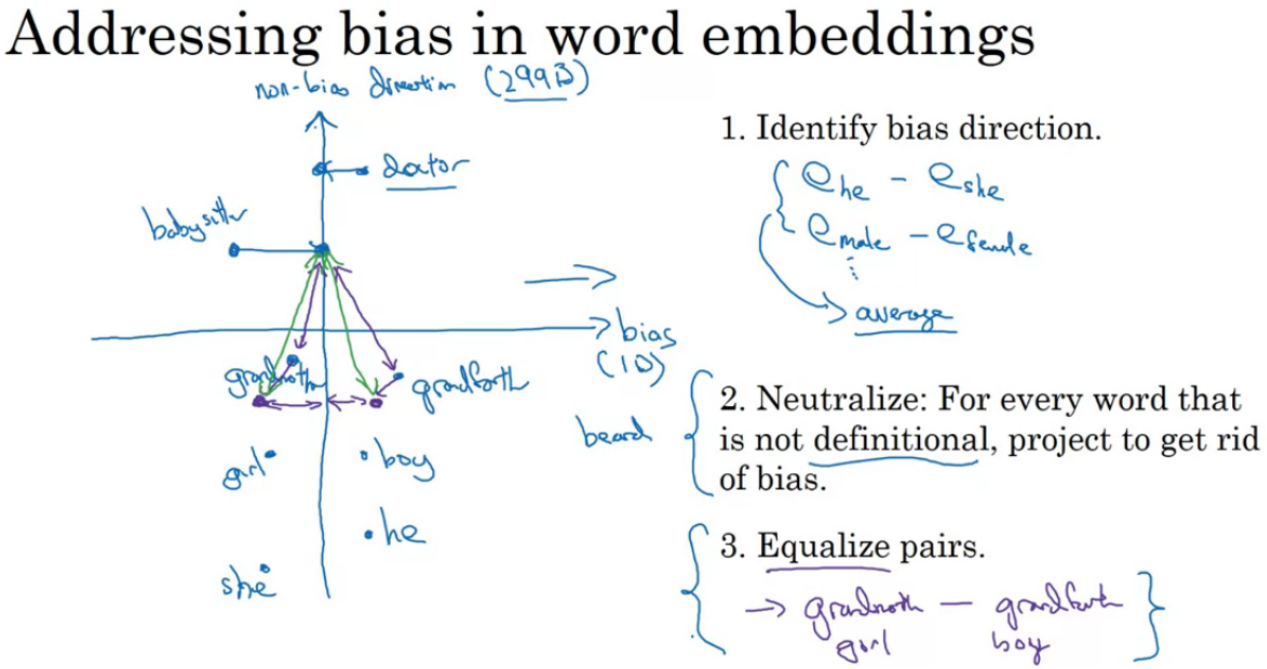
February 19, 2021



Aakash Goel

·Just now

PAPER LINK →
https://papers.nips.cc/paper/2016/file/a486cd07e4ac3d270571622f4f316ec5-Paper.pdf



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 an 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.)

```
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. Consider what a positive value of similarity means vs a negative cosine similarity.

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.

```
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))
```

```
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 surprising, 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', 'receptionist',
             '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 types 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 "orthogonal") 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 $e^{debiased}_{receptionist}$.

Even though $g_\perp$ is 49 dimensional, given the limitations of what we can draw on a 2D screen, we illustrate it using a 1 dimensional axis below.

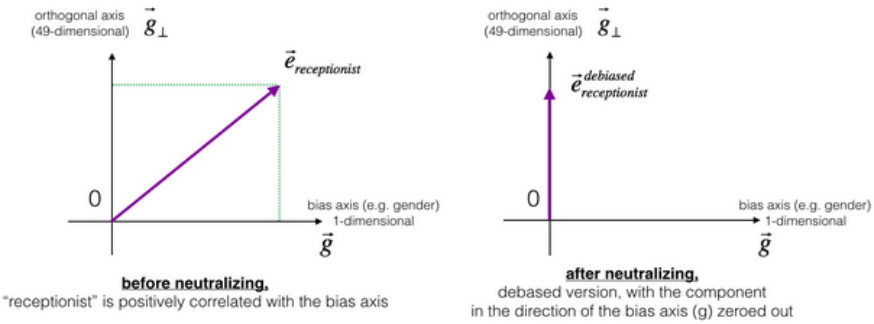


Figure 2: The word vector for "receptionist" represented before and after applying the neutralize operation.

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.

```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 given above. (≈ 1 line)
    e_biascomponent = (np.dot(e,g)/np.sum(np.square(g)))*g

    # Neutralize e by subtracting 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_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))
```

```
cosine similarity between receptionist and g, before neutralizing:  0.330779417506
cosine similarity between receptionist and g, after neutralizing:  -4.08872263257e-17
```

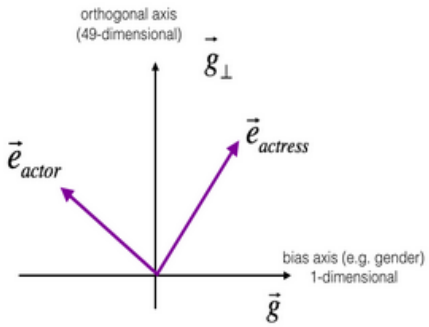
Expected Output: The second result is essentially 0, up to numerical rounding (on the order of 10^{-17}).

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

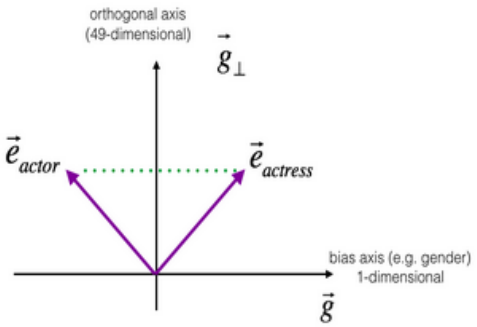
3.2 - Equalization algorithm for gender-specific words

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 $e^{debiased}_{receptionist}$, 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:

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

$$\mu_B = \frac{\mu \cdot \text{bias\_axis}}{||\text{bias\_axis}||_2^2} * \text{bias\_axis}$$

$$\mu_\perp = \mu - \mu_B$$

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

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

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

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

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

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

QUIZ

1. Suppose you learn a word embedding for a vocabulary of 10000 words. Then the embedding vectors should be 10000 dimensional, so as to capture the full range of variation and meaning in those words.

- ○ True
- ◉ False

✓ **Correct**

The dimension of word vectors is usually smaller than the size of the vocabulary. Most common sizes for word vectors ranges between 50 and 400.

2. What is t-SNE?

- ○ A linear transformation that allows us to solve analogies on word vectors
- ◉ A non-linear dimensionality reduction technique
- ○ A supervised learning algorithm for learning word embeddings
- ○ An open-source sequence modeling library

✓ **Correct**

Yes

3. Suppose you download a pre-trained word embedding which has been trained on a huge corpus of text. You then use this word embedding to train an RNN for a language task of recognizing if someone is happy from a short snippet of text, using a small training set.

x (input text)	y (happy?)
I'm feeling wonderful today!	1
I'm bummed my cat is ill.	0
Really enjoying this!	1

Then even if the word "ecstatic" does not appear in your small training set, your RNN might reasonably be expected to recognize "I'm ecstatic" as deserving a label $y = 1$.

- ◉ True
- ○ False

✓ **Correct**

Yes, word vectors empower your model with an incredible ability to generalize. The vector for "ecstatic would contain a positive/happy connotation which will probably make your model classified the sentence as a "1".

4. Which of these equations do you think should hold for a good word embedding? (Check all that apply)

- [x] $e_{boy} - e_{girl} \approx e_{brother} - e_{sister}$

  Correct
  Yes!

- [ ] $e_{boy} - e_{girl} \approx e_{sister} - e_{brother}$

- [x] $e_{boy} - e_{brother} \approx e_{girl} - e_{sister}$

  Correct
  Yes!

- [ ] $e_{boy} - e_{brother} \approx e_{sister} - e_{girl}$

5. Let E be an embedding matrix, and let o_{1234} be a one-hot vector corresponding to word 1234. Then to get the embedding of word 1234, why don't we call $E * o_{1234}$ in Python?

- ( ) It is computationally wasteful.
- ( ) The correct formula is $E^T * o_{1234}$.
- (•) This doesn't handle unknown words (<UNK>).
- ( ) None of the above: calling the Python snippet as described above is fine.

  Incorrect
  No, this is not the correct reason. If the unknown token is added to the vocabulary and the vocabulary list is passed as an input to the Embedding layer, then the element-wise operation is valid even for the unknown token.

6. When learning word embeddings, we create an artificial task of estimating $P(target \mid context)$. It is okay if we do poorly on this artificial prediction task; the more important by-product of this task is that we learn a useful set of word embeddings.

- (•) True
- ( ) False

  Correct

7. In the word2vec algorithm, you estimate $P(t \mid c)$, where t is the target word and c is a context word. How are t and c chosen from the training set? Pick the best answer.

- (●) c is the one word that comes immediately before t.
- ( ) c is the sequence of all the words in the sentence before t.
- ( ) c and t are chosen to be nearby words.
- ( ) c is a sequence of several words immediately before t.

! **Incorrect**

8. Suppose you have a 10000 word vocabulary, and are learning 500-dimensional word embeddings. The word2vec model uses the following softmax function:

$$P(t \mid c) = \frac{e^{\theta_t^T e_c}}{\sum_{t'=1}^{10000} e^{\theta_t^T e_c}}$$

Which of these statements are correct? Check all that apply.

- [x] θ_t and e_c are both 500 dimensional vectors.

✓ **Correct**

- [ ] θ_t and e_c are both 10000 dimensional vectors.
- [x] θ_t and e_c are both trained with an optimization algorithm such as Adam or gradient descent.

✓ **Correct**

- [ ] After training, we should expect θ_t to be very close to e_c when t and c are the same word.

9. Suppose you have a 10000 word vocabulary, and are learning 500-dimensional word embeddings.The GloVe model minimizes this objective:

$\min \sum_{i=1}^{10,000} \sum_{j=1}^{10,000} f(X_{ij})(\theta_i^T e_j + b_i + b_j' - logX_{ij})^2$

Which of these statements are correct? Check all that apply.

- [ ] θ_i and e_j should be initialized to 0 at the beginning of training.
- [x] θ_i and e_j should be initialized randomly at the beginning of training.

  ✓ **Correct**

- [x] X_{ij} is the number of times word j appears in the context of word i.

  ✓ **Correct**

- [x] The weighting function $f(.)$ must satisfy $f(0) = 0$.

  ✓ **Correct**
  The weighting function helps prevent learning only from extremely common word pairs. It is not necessary that it satisfies this function.

10. You have trained word embeddings using a text dataset of m_1 words. You are considering using these word embeddings for a language task, for which you have a separate labeled dataset of m_2 words. Keeping in mind that using word embeddings is a form of transfer learning, under which of these circumstance would you expect the word embeddings to be helpful?

- (●) $m_1 >> m_2$
- ( ) $m_1 << m_2$

✓ **Correct**

2nd Exercise → Emojifier

Emojify!

Welcome to the second assignment of Week 2. You are going to use word vector representations to build an Emojifier.

Have you ever wanted to make your text messages more expressive? Your emojifier app will help you do that. So rather than writing:

> "Congratulations on the promotion! Let's get coffee and talk. Love you!"

The emojifier can automatically turn this into:

> "Congratulations on the promotion! 👍 Let's get coffee and talk. ☕ Love you! ❤"

- You will implement a model which inputs a sentence (such as "Let's go see the baseball game tonight!") and finds the most appropriate emoji to be used with this sentence (⚾).

Using word vectors to improve emoji lookups

- In many emoji interfaces, you need to remember that ❤ is the "heart" symbol rather than the "love" symbol.
  - In other words, you'll have to remember to type "heart" to find the desired emoji, and typing "love" won't bring up that symbol.
- We can make a more flexible emoji interface by using word vectors!
- When using word vectors, you'll see that even if your training set explicitly relates only a few words to a particular emoji, your algorithm will be able to generalize and associate additional words in the test set to the same emoji.
  - This works even if those additional words don't even appear in the training set.
  - This allows you to build an accurate classifier mapping from sentences to emojis, even using a small training set.

What you'll build

1. In this exercise, you'll start with a baseline model (Emojifier-V1) using word embeddings.
2. Then you will build a more sophisticated model (Emojifier-V2) that further incorporates an LSTM.

DATASET

1.1 - Dataset EMOJISET

Let's start by building a simple baseline classifier.

You have a tiny dataset (X, Y) where:

- X contains 127 sentences (strings).
- Y contains an integer label between 0 and 4 corresponding to an emoji for each sentence.

X (sentences)	Y (labels)
I love you	0
Congrats on the new job	2
I think I will end up alone	3
I want to have sushi for dinner!	4
It was funny lol	2
she did not answer my text	3
Happy new year	2
my algorithm performs poorly	3
he can pitch really well	1
you are failing this exercise	3
you did well on your exam.	2
What you did was awesome	2
I am frustrated	3

code	emoji	label
:heart:	❤️	0
:baseball:	⚾	1
:smile:	😀	2
:disappointed:	😔	3
:fork_and_knife:	🍴	4

Figure 1: EMOJISET - a classification problem with 5 classes. A few examples of sentences are given here.

Let's load the dataset using the code below. We split the dataset between training (127 examples) and testing (56 examples).

In this part, you are going to implement a baseline model called "Emojifier-v1".

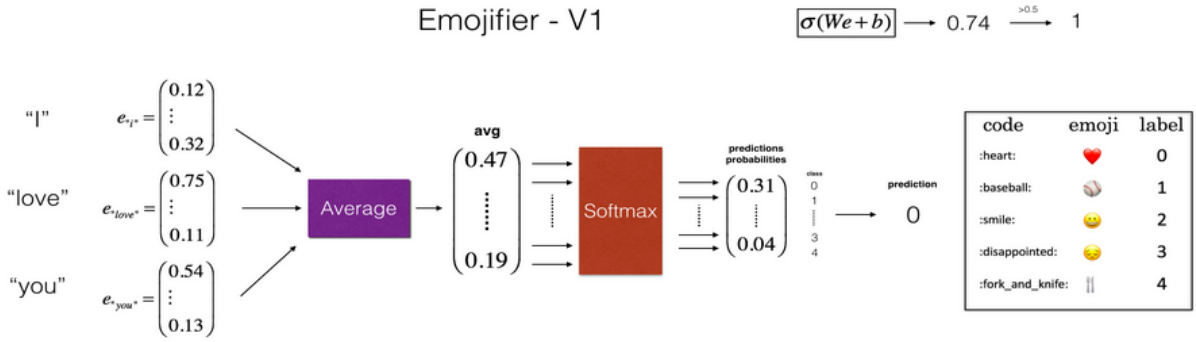


Figure 2: Baseline model (Emojifier-V1).

Inputs and outputs

- The input of the model is a string corresponding to a sentence (e.g. "I love you).
- The output will be a probability vector of shape (1,5), (there are 5 emojis to choose from).
- The (1,5) probability vector is passed to an argmax layer, which extracts the index of the emoji with the highest probability.

To convert target variable into one hot encoder

np.eye(5)[np.array([0,1]).reshape(-1)]

V2 Emojifier

Here is the Emojifier-v2 you will implement:

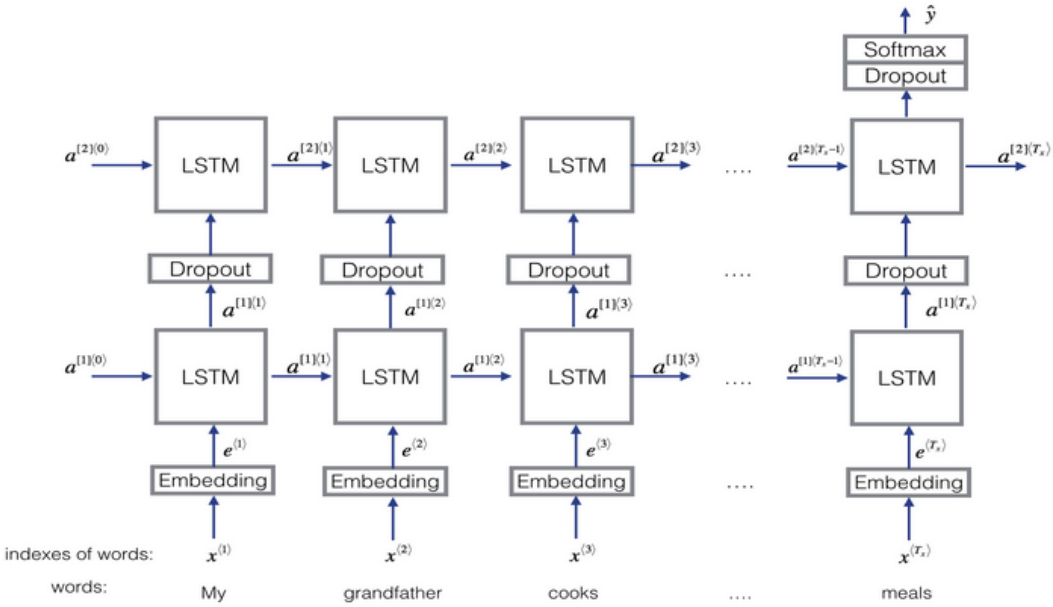


Figure 3: Emojifier-V2. A 2-layer LSTM sequence classifier.

2.2 Keras and mini-batching

- In this exercise, we want to train Keras using mini-batches.
- However, most deep learning frameworks require that all sequences in the same mini-batch have the **same length**.
    - This is what allows vectorization to work: If you had a 3-word sentence and a 4-word sentence, then the computations needed for them are different (one takes 3 steps of an LSTM, one takes 4 steps) so it's just not possible to do them both at the same time.

Padding handles sequences of varying length

- The common solution to handling sequences of **different length** is to use padding. Specifically:
    - Set a maximum sequence length
    - Pad all sequences to have the same length.

Example of padding

- Given a maximum sequence length of 20, we could pad every sentence with "0"s so that each input sentence is of length 20.
- Thus, the sentence "I love you" would be represented as $(e_I, e_{love}, e_{you}, \vec{0}, \vec{0}, \ldots, \vec{0})$.
- In this example, any sentences longer than 20 words would have to be truncated.
- One way to choose the maximum sequence length is to just pick the length of the longest sentence in the training set.

2.3 - The Embedding layer

- In Keras, the embedding matrix is represented as a "layer".
- The embedding matrix maps word indices to embedding vectors.
  - The word indices are positive integers.
  - The embedding vectors are dense vectors of fixed size.
  - When we say a vector is "dense", in this context, it means that most of the values are non-zero. As a counter-example, a one-hot encoded vector is not "dense."
- The embedding matrix can be derived in two ways:
  - Training a model to derive the embeddings from scratch.
  - Using a pretrained embedding

Using and updating pre-trained embeddings

- In this part, you will learn how to create an Embedding() layer in Keras
- You will initialize the Embedding layer with the GloVe 50-dimensional vectors.
- In the code below, we'll show you how Keras allows you to either train or leave fixed this layer.
- Because our training set is quite small, we will leave the GloVe embeddings fixed instead of updating them.

Inputs and outputs to the embedding layer

- The `Embedding()` layer's input is an integer matrix of size **(batch size, max input length)**.
  - This input corresponds to sentences converted into lists of indices (integers).
  - The largest integer (the highest word index) in the input should be no larger than the vocabulary size.
- The embedding layer outputs an array of shape (batch size, max input length, dimension of word vectors).
- The figure shows the propagation of two example sentences through the embedding layer.
  - Both examples have been zero-padded to a length of `max_len=5`.
  - The word embeddings are 50 units in length.
  - The final dimension of the representation is `(2,max_len,50)`.

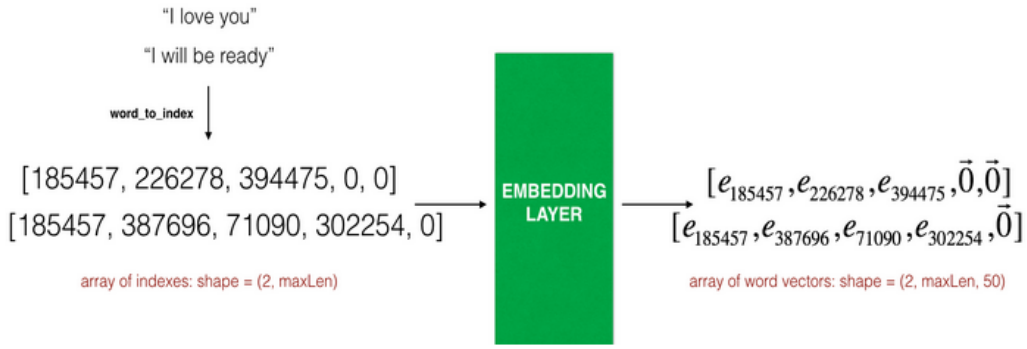


Figure 4: Embedding layer

Prepare the input sentences

Exercise:

- Implement `sentences_to_indices`, which processes an array of sentences (X) and returns inputs to the embedding layer:
  - Convert each training sentences into a list of indices (the indices correspond to each word in the sentence)
  - Zero-pad all these lists so that their length is the length of the longest sentence.

Additional Hints

- Note that you may have considered using the `enumerate()` function in the for loop, but for the purposes of passing the autograder, please follow the starter code by initializing and incrementing `j` explicitly.

Build embedding layer

- Let's build the `Embedding()` layer in Keras, using pre-trained word vectors.
- The embedding layer takes as input a list of word indices.
  - `sentences_to_indices()` creates these word indices.
- The embedding layer will return the word embeddings for a sentence.

Exercise: Implement `pretrained_embedding_layer()` with these steps:

1. Initialize the embedding matrix as a numpy array of zeros.
   - The embedding matrix has a row for each unique word in the vocabulary.
     - There is one additional row to handle "unknown" words.
     - So vocab_len is the number of unique words plus one.
   - Each row will store the vector representation of one word.
     - For example, one row may be 50 positions long if using GloVe word vectors.
   - In the code below, `emb_dim` represents the length of a word embedding.
2. Fill in each row of the embedding matrix with the vector representation of a word
   - Each word in `word_to_index` is a string.
   - word_to_vec_map is a dictionary where the keys are strings and the values are the word vectors.
3. Define the Keras embedding layer.
   - Use Embedding().
   - The input dimension is equal to the vocabulary length (number of unique words plus one).
   - The output dimension is equal to the number of positions in a word embedding.
   - Make this layer's embeddings fixed.
     - If you were to set `trainable = True`, then it will allow the optimization algorithm to modify the values of the word embeddings.
     - In this case, we don't want the model to modify the word embeddings.
4. Set the embedding weights to be equal to the embedding matrix.
   - Note that this is part of the code is already completed for you and does not need to be modified.

```
# GRADED FUNCTION: Emojify_V2

def Emojify_V2(input_shape, word_to_vec_map, word_to_index):
    """
    Function creating the Emojify-v2 model's graph.

    Arguments:
    input_shape -- shape of the input, usually (max_len,)
    word_to_vec_map -- dictionary mapping every word in a vocabulary into its 50-dimensional vector repres
    word_to_index -- dictionary mapping from words to their indices in the vocabulary (400,001 words)

    Returns:
    model -- a model instance in Keras
    """

    ### START CODE HERE ###
    # Define sentence_indices as the input of the graph.
    # It should be of shape input_shape and dtype 'int32' (as it contains indices, which are integers).
    sentence_indices = Input(shape = input_shape)

    # Create the embedding layer pretrained with GloVe Vectors (≈1 line)
    embedding_layer = pretrained_embedding_layer(word_to_vec_map, word_to_index)

    # Propagate sentence_indices through your embedding layer
    # (See additional hints in the instructions).
    embeddings = embedding_layer(sentence_indices)

    # Propagate the embeddings through an LSTM layer with 128-dimensional hidden state
    # The returned output should be a batch of sequences.
    X = LSTM(128,return_sequences = True)(embeddings)
```

```
    # Add dropout with a probability of 0.5
    X = Dropout(0.5)(X)
    # Propagate X trough another LSTM layer with 128-dimensional hidden state
    # The returned output should be a single hidden state, not a batch of sequences.
    X = LSTM(128, return_sequences = False)(X)
    # Add dropout with a probability of 0.5
    X = Dropout(0.5)(X)
    # Propagate X through a Dense layer with 5 units
    X = Dense(5,activation=None)(X)
    # Add a softmax activation
    X = Activation('softmax')(X)

    # Create Model instance which converts sentence_indices into X.
    model = Model(inputs = sentence_indices,outputs = X)

    ### END CODE HERE ###

    return model
```

Run the following cell to create your model and check its summary. Because all sentences in the dataset are less than 10 words, we chose `max_len = 10`. You should see your architecture, it uses "20,223,927" parameters, of which 20,000,050 (the word embeddings) are non-trainable, and the remaining 223,877 are. Because our vocabulary size has 400,001 words (with valid indices from 0 to 400,000) there are 400,001*50 = 20,000,050 non-trainable parameters.

```
model = Emojify_V2((maxLen,), word_to_vec_map, word_to_index)
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_5 (InputLayer)         (None, 10)                0
_________________________________________________________________
embedding_6 (Embedding)      (None, 10, 50)            20000050
_________________________________________________________________
lstm_7 (LSTM)                (None, 10, 128)           91648
_________________________________________________________________
dropout_6 (Dropout)          (None, 10, 128)           0
_________________________________________________________________
lstm_8 (LSTM)                (None, 128)               131584
_________________________________________________________________
dropout_7 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 5)                 645
_________________________________________________________________
activation_1 (Activation)    (None, 5)                 0
=================================================================
Total params: 20,223,927
Trainable params: 223,877
Non-trainable params: 20,000,050
_________________________________________________________________
```

As usual, after creating your model in Keras, you need to compile it and define what loss, optimizer and metrics your are want to use. Compile your model using `categorical_crossentropy` loss, `adam` optimizer and `['accuracy']` metrics:

```
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

It's time to train your model. Your Emojifier-V2 `model` takes as input an array of shape (`m`, `max_len`) and outputs probability vectors of shape (`m`, `number of classes`). We thus have to convert X_train (array of sentences as strings) to X_train_indices (array of sentences as list of word indices), and Y_train (labels as indices) to Y_train_oh (labels as one-hot vectors).

```
X_train_indices = sentences_to_indices(X_train, word_to_index, maxLen)
Y_train_oh = convert_to_one_hot(Y_train, C = 5)
```

LSTM version accounts for word order

- Previously, Emojify-V1 model did not correctly label "not feeling happy," but our implementation of Emojiy-V2 got it right.
  - (Keras' outputs are slightly random each time, so you may not have obtained the same result.)
- The current model still isn't very robust at understanding negation (such as "not happy")
  - This is because the training set is small and doesn't have a lot of examples of negation.
  - But if the training set were larger, the LSTM model would be much better than the Emojify-V1 model at understanding such complex sentences.

Congratulations!

You have completed this notebook! ❤❤❤

What you should remember

- If you have an NLP task where the training set is small, using word embeddings can help your algorithm significantly.
- Word embeddings allow your model to work on words in the test set that may not even appear in the training set.
- Training sequence models in Keras (and in most other deep learning frameworks) requires a few important details:
  - To use mini-batches, the sequences need to be **padded** so that all the examples in a mini-batch have the **same length**.
  - An `Embedding()` layer can be initialized with pretrained values.
    - These values can be either fixed or trained further on your dataset.
    - If however your labeled dataset is small, it's usually not worth trying to train a large pre-trained set of embeddings.
  - `LSTM()` has a flag called `return_sequences` to decide if you would like to return every hidden states or only the last one.
  - You can use `Dropout()` right after `LSTM()` to regularize your network.

********** END **********