

















Emoji_v3a

 **Jupyter** Emoji_v3a 最后检查: 几秒前 (自动保存) 

File Edit View Insert Cell Kernel Widgets Help 可信的 Python 3 

       运行   代码  Submit Assignment




Emojiify!


Welcome to the second assignment of Week 2! You're going to use word vector representations to build an Emojiifier.   

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


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

The emojiifier can automatically turn this into:

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

You'll 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.
- You 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 (Emojiifier-V1) using word embeddings.
2. Then you will build a more sophisticated model (Emojiifier-V2) that further incorporates an LSTM.

By the end of this notebook, you'll be able to:

- Create an embedding layer in Keras with pre-trained word vectors
- Explain the advantages and disadvantages of the GloVe algorithm
- Describe how negative sampling learns word vectors more efficiently than other methods
- Build a sentiment classifier using word embeddings
- Build and train a more sophisticated classifier using an LSTM

(^^^ Emoji for "skills")

Important Note on Submission to the AutoGrader

Before submitting your assignment to the AutoGrader, please make sure you are not doing the following:

1. You have not added any *extra* `print` statement(s) in the assignment.
2. You have not added any *extra* code cell(s) in the assignment.
3. You have not changed any of the function parameters.
4. You are not using any global variables inside your graded exercises. Unless specifically instructed to do so, please refrain from it and use the local variables instead.
5. You are not changing the assignment code where it is not required, like creating *extra* variables.

If you do any of the following, you will get something like, `Grader not found` (or similarly unexpected) error upon submitting your assignment. Before asking for help/debugging the errors in your assignment, check for these first. If this is the case, and you don't remember the changes you have made, you can get a fresh copy of the assignment by following these [instructions](#).

Table of Contents

- [Packages](#)
- [1 - Baseline Model: Emojifier-V1](#)
 - [1.1 - Dataset EMOJISSET](#)
 - [1.2 - Overview of the Emojifier-V1](#)
 - [1.3 - Implementing Emojifier-V1](#)
 - [Exercise 1 - sentence_to_avg](#)
 - [1.4 - Implement the Model](#)
 - [Exercise 2 - model](#)
 - [1.5 - Examining Test Set Performance](#)
- [2 - Emojifier-V2: Using LSTMs in Keras](#)
 - [2.1 - Model Overview](#)
 - [2.2 Keras and Mini-batching](#)
 - [2.3 - The Embedding Layer](#)
 - [Exercise 3 - sentences_to_indices](#)
 - [Exercise 4 - pretrained_embedding_layer](#)
 - [2.4 - Building the Emojifier-V2](#)
 - [Exercise 5 - Emojify_V2](#)
 - [2.5 - Train the Model](#)
- [3 - Acknowledgments](#)

Packages

Let's get started! Run the following cell to load the packages you're going to use.

```
In [1]: import numpy as np
        from emo_utils import *
        import emoji
        import matplotlib.pyplot as plt
        from test_utils import *

        %matplotlib inline
```

1 - Baseline Model: Emojifier-V1

1.1 - Dataset EMOJISSET

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

Load the dataset using the code below. The dataset is split between training (127 examples) and testing (56 examples).

Load the dataset using the code below. The dataset is split between training (127 examples) and testing (56 examples).

```
In [2]: X_train, Y_train = read_csv('data/train_emoji.csv')
        X_test, Y_test = read_csv('data/test.csv')
```

```
In [3]: maxLen = len(max(X_train, key=len).split())
```

Run the following cell to print sentences from `X_train` and corresponding labels from `Y_train`.

- Change `idx` to see different examples.
- Note that due to the font used by iPython notebook, the heart emoji may be colored black rather than red.

```
In [4]: for idx in range(10):
        print(X_train[idx], label_to_emoji(Y_train[idx]))
```

```
never talk to me again
I am proud of your achievements
It is the worst day in my life
Miss you so much ♥
food is life
I love you mum ♥
Stop saying bullshit
congratulations on your acceptance
The assignment is too long
I want to go play 🎮
```

1.2 - Overview of the Emojiifier-V1

In this section, you'll implement a baseline model called "Emojiifier-v1".

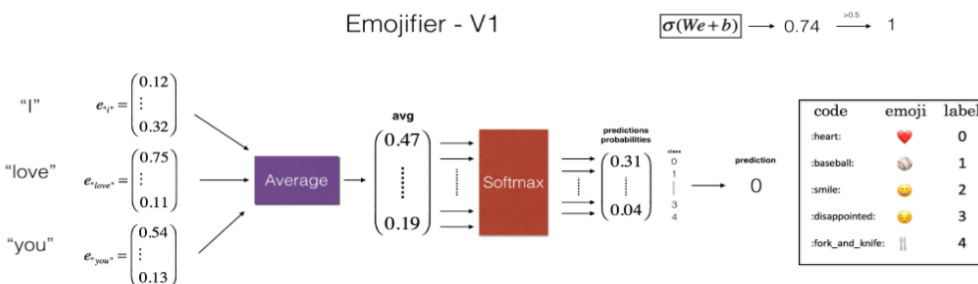


Figure 2: Baseline model (Emojiifier-V1).

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), (indicating that 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.

One-hot Encoding

- To get your labels into a format suitable for training a softmax classifier, convert Y from its current shape $(m, 1)$ into a "one-hot representation" $(m, 5)$,
 - Each row is a one-hot vector giving the label of one example.
 - Here, Y_{oh} stands for "Y-one-hot" in the variable names Y_{oh_train} and Y_{oh_test} .

```
In [5]: Y_oh_train = convert_to_one_hot(Y_train, C = 5)
        Y_oh_test = convert_to_one_hot(Y_test, C = 5)
```

Now, see what `convert_to_one_hot()` did. Feel free to change `index` to print out different values.

```
In [6]: idx = 50
        print(f'Sentence {X_train[idx]} has label index {Y_train[idx]}, which is emoji {label_to_emoji(Y_train[idx])}', )
        print(f'Label index {Y_train[idx]} in one-hot encoding format is {Y_oh_train[idx]}')

Sentence 'I missed you' has label index 0, which is emoji ♥
Label index 0 in one-hot encoding format is [1. 0. 0. 0. 0.]
```

All the data is now ready to be fed into the Emojiify-V1 model. You're ready to implement the model!

1.3 - Implementing Emojiifier-V1

As shown in Figure 2 (above), the first step is to:

- Convert each word in the input sentence into their word vector representations.
- Take an average of the word vectors.

Similar to this week's previous assignment, you'll use pre-trained 50-dimensional GloVe embeddings.

Run the following cell to load the `word_to_vec_map`, which contains all the vector representations.

```
In [7]: word_to_index, index_to_word, word_to_vec_map = read_glove_vecs('data/glove.6B.50d.txt')
```

You've loaded:

- `word_to_index`: dictionary mapping from words to their indices in the vocabulary
 - (400,001 words, with the valid indices ranging from 0 to 400,000)
- `index_to_word`: dictionary mapping from indices to their corresponding words in the vocabulary
- `word_to_vec_map`: dictionary mapping words to their GloVe vector representation.

Run the following cell to check if it works:

```
In [8]: word = "cucumber"
idx = 289846
print("the index of", word, "in the vocabulary is", word_to_index[word])
print("the", str(idx) + "th word in the vocabulary is", index_to_word[idx])

the index of cucumber in the vocabulary is 113317
the 289846th word in the vocabulary is potatoes
```

Exercise 1 - sentence_to_avg

Implement `sentence_to_avg()`

You'll need to carry out two steps:

1. Convert every sentence to lower-case, then split the sentence into a list of words.
 - `X.lower()` and `X.split()` might be useful. □
2. For each word in the sentence, access its GloVe representation.
 - Then take the average of all of these word vectors.
 - You might use `numpy.zeros()`, which you can read more about [here](#).

Additional Hints

- When creating the `avg` array of zeros, you'll want it to be a vector of the same shape as the other word vectors in the `word_to_vec_map`.
 - You can choose a word that exists in the `word_to_vec_map` and access its `.shape` field.
 - Be careful not to hard-code the word that you access. In other words, don't assume that if you see the word 'the' in the `word_to_vec_map` within this notebook, that this word will be in the `word_to_vec_map` when the function is being called by the automatic grader.

Hint: you can use any one of the word vectors that you retrieved from the input `sentence` to find the shape of a word vector.

```
In [9]: # UNQ_C1 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# GRADED FUNCTION: sentence_to_avg

def sentence_to_avg(sentence, word_to_vec_map):
    """
    Converts a sentence (string) into a list of words (strings). Extracts the GloVe representation of each word
    and averages its value into a single vector encoding the meaning of the sentence.

    Arguments:
    sentence -- string, one training example from X
    word_to_vec_map -- dictionary mapping every word in a vocabulary into its 50-dimensional vector representation

    Returns:
    avg -- average vector encoding information about the sentence, numpy-array of shape (J,), where J can be any number
    """
    # Get a valid word contained in the word_to_vec_map.
    any_word = list(word_to_vec_map.keys())[0]

    ### START CODE HERE ###
    # Step 1: Split sentence into list of lower case words (~ 1 line)
    words = sentence.lower().split()

    # Initialize the average word vector, should have the same shape as your word vectors.
    avg = np.zeros(word_to_vec_map[any_word].shape)
```

```
In [11]: # UNQ_C2 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# GRADED FUNCTION: model

def model(X, Y, word_to_vec_map, learning_rate = 0.01, num_iterations = 400):
    """
    Model to train word vector representations in numpy.

    Arguments:
    X -- input data, numpy array of sentences as strings, of shape (m,)
    Y -- labels, numpy array of integers between 0 and 7, numpy-array of shape (m, 1)
    word_to_vec_map -- dictionary mapping every word in a vocabulary into its 50-dimensional vector representation
    learning_rate -- learning_rate for the stochastic gradient descent algorithm
    num_iterations -- number of iterations

    Returns:
    pred -- vector of predictions, numpy-array of shape (m, 1)
    W -- weight matrix of the softmax layer, of shape (n_y, n_h)
    b -- bias of the softmax layer, of shape (n_y,)
    """

    # Get a valid word contained in the word_to_vec_map
    any_word = list(word_to_vec_map.keys())[0]

    # Define number of training examples
    m = Y.shape[0] # number of training examples
    n_y = len(np.unique(Y)) # number of classes
    n_h = word_to_vec_map[any_word].shape[0] # dimensions of the GloVe vectors

    # Initialize parameters using Xavier initialization
    W = np.random.randn(n_y, n_h) / np.sqrt(n_h)
    b = np.zeros((n_y,))
```

Run the next cell to train your model and learn the softmax parameters (W, b). The training process will take about 5 minutes

```
In [13]: np.random.seed(1)
pred, W, b = model(X_train, Y_train, word_to_vec_map)
print(pred)
```

```
Epoch: 0 --- cost = 410.4336578831472
Accuracy: 0.5454545454545454
Epoch: 100 --- cost = 63.612639746961435
Accuracy: 0.9318181818181818
Epoch: 200 --- cost = 0.7391301193275178
Accuracy: 1.0
Epoch: 300 --- cost = 0.3104825413333956
Accuracy: 1.0
[[3.]
 [2.]
 [3.]
```

1.5 - Examining Test Set Performance

Note that the `predict` function used here is defined in `emo_util.py`.

```
In [14]: print("Training set:")
pred_train = predict(X_train, Y_train, W, b, word_to_vec_map)
print('Test set:')
pred_test = predict(X_test, Y_test, W, b, word_to_vec_map)
```

```
Training set:
Accuracy: 1.0
Test set:
Accuracy: 0.9107142857142857
```

Note:

- Random guessing would have had 20% accuracy, given that there are 5 classes. (1/5 = 20%).
- This is pretty good performance after training on only 127 examples.

The Model Matches Emojis to Relevant Words

In the training set, the algorithm saw the sentence

"I love you."

with the label 🍷.

1.4 - Implement the Model

You now have all the pieces to finish implementing the `model()` function! After using `sentence_to_avg()` you need to:

- Pass the average through forward propagation
- Compute the cost
- Backpropagate to update the softmax parameters

Exercise 2 - model

Implement the `model()` function described in Figure (2).

- The equations you need to implement in the forward pass and to compute the cross-entropy cost are below:
- The variable Y_{oh} ("Y one hot") is the one-hot encoding of the output labels.

$$z^{(i)} = Wavg^{(i)} + b$$
$$a^{(i)} = softmax(z^{(i)})$$
$$\mathcal{L}^{(i)} = - \sum_{k=0}^{n_y-1} Y_{oh,k}^{(i)} * \log(a_k^{(i)})$$

Note: It is possible to come up with a more efficient vectorized implementation. For now, just use nested for loops to better understand the algorithm, and for easier debugging.

The function `softmax()` is provided, and has already been imported.

1.5 - Examining Test Set Performance

Note that the `predict` function used here is defined in `emo_util.py`.

```
In [14]: print("Training set:")
pred_train = predict(X_train, Y_train, W, b, word_to_vec_map)
print("Test set:")
pred_test = predict(X_test, Y_test, W, b, word_to_vec_map)
```

```
Training set:
Accuracy: 1.0
Test set:
Accuracy: 0.9107142857142857
```

Note:

- Random guessing would have had 20% accuracy, given that there are 5 classes. (1/5 = 20%).
- This is pretty good performance after training on only 127 examples.

The Model Matches Emojis to Relevant Words

In the training set, the algorithm saw the sentence

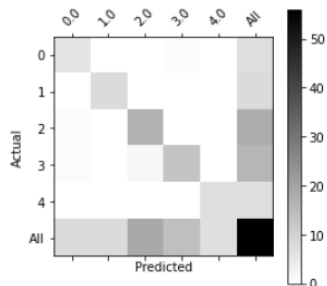
"I love you."

with the label 🍷.

Confusion Matrix

- Printing the confusion matrix can also help understand which classes are more difficult for your model.
- A confusion matrix shows how often an example whose label is one class ("actual" class) is mislabeled by the algorithm with a different class ("predicted" class).

Print the confusion matrix below:



What you should remember:

- Even with a mere 127 training examples, you can get a reasonably good model for Emojifying.
 - This is due to the generalization power word vectors gives you.
- Emojify-V1 will perform poorly on sentences such as "This movie is not good and not enjoyable"
 - It doesn't understand combinations of words.
 - It just averages all the words' embedding vectors together, without considering the ordering of words.

Not to worry! You will build a better algorithm in the next section!

2 - Emojiifier-V2: Using LSTMs in Keras

You're going to build an LSTM model that takes word **sequences** as input! This model will be able to account for word ordering.

Emojiifier-V2 will continue to use pre-trained word embeddings to represent words. You'll feed word embeddings into an LSTM, and the LSTM will learn to predict the most appropriate emoji.

Packages

Run the following cell to load the Keras packages you'll need:

```
In [17]: import numpy as np
import tensorflow
np.random.seed(0)
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Input, Dropout, LSTM, Activation
from tensorflow.keras.layers import Embedding
from tensorflow.keras.preprocessing import sequence
from tensorflow.keras.initializers import glorot_uniform
np.random.seed(1)
```

2.1 - Model Overview

Here is the Emojiifier-v2 you will implement:

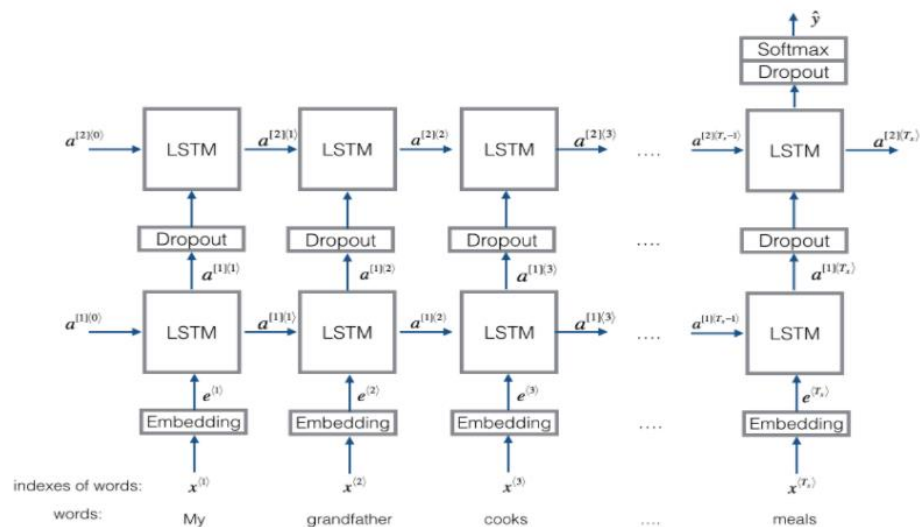


Figure 3: Emojiifier-V2. A 2-layer LSTM sequence classifier.

2.2 Keras and Mini-batching

In this exercise, you 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, you 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 $(e_I, e_{love}, e_{you}, \vec{0}, \dots, \vec{0})$.
- 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.
 - A "dense" vector is the opposite of a sparse vector. It means that most of its 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 section, you'll create an `Embedding()` layer in Keras

- You will initialize the Embedding layer with GloVe 50-dimensional vectors.
- In the code below, you'll observe how Keras allows you to either train or leave this layer fixed.
 - Because your training set is quite small, you'll 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)`.

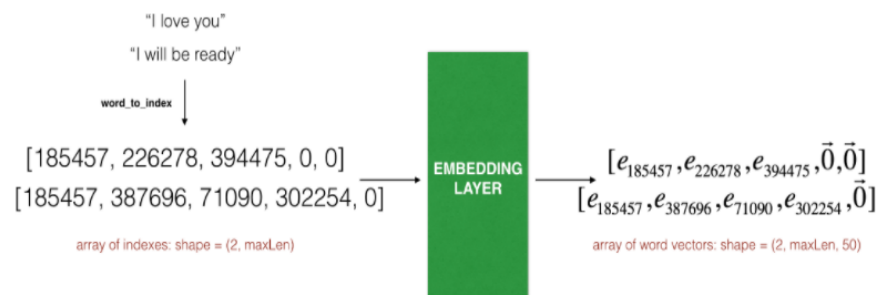


Figure 4: Embedding layer

Prepare the Input Sentences

Exercise 3 - sentences_to_indices

Implement `sentences_to_indices`

This function 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.

2.4 - Building the Emojiifier-V2

Now you're ready to build the Emojiifier-V2 model, in which you feed the embedding layer's output to an LSTM network!

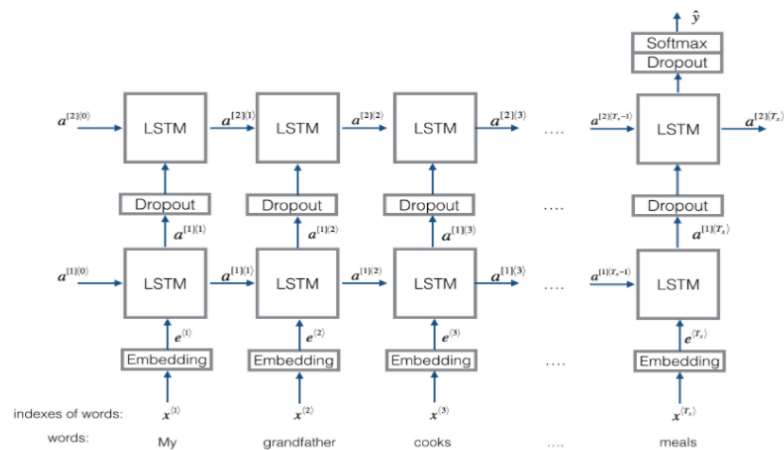


Figure 3: Emojiifier-v2. A 2-layer LSTM sequence classifier.

3 - Acknowledgments

Thanks to Alison Darcy and the Woebot team for their advice on the creation of this assignment.

- Woebot is a chatbot friend that is ready to speak with you 24/7.
- Part of Woebot's technology uses word embeddings to understand the emotions of what you say.
- You can chat with Woebot by going to <http://woebot.io>

