

# Emojify

In this short article I will explain how we can make our messages become more expressive via word vector representations (This is an extremely powerful tool in NLP). For example, instead of writing:

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

This tool could automatically turn this into:

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

Moreover, with word vectors, we will observe that even if the training set explicitly relates only a few words to a particular emoji, the 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.

In order to carry out this task, we will rely on 2 methods. The first and simple one involves the use of word embeddings. In the second method, we would also incorporate an LSTM (Long-Short Term Memory).

Firstly, let's import the necessary packages (install them if needed)






```
import numpy as np
import emoji
import matplotlib.pyplot as plt
```

## Baseline Model Emojifier-V1

We already 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

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

```
X_train, Y_train = read_csv('data/train_emoji.csv')
X_test, Y_test = read_csv('data/tesss.csv')
```

Next, let's select the sentence with the maximum amount of words (maxLen)

```
maxLen = len(max(X_train, key=lambda x: len(x.split()))).split())
```

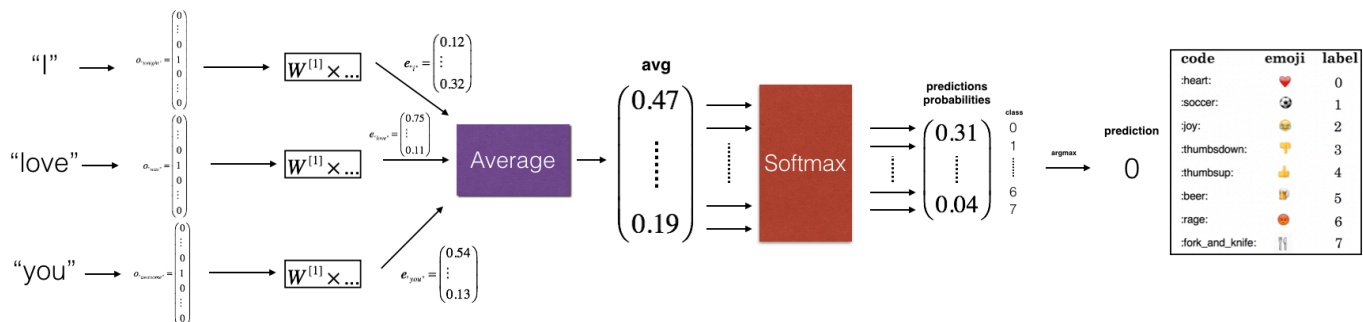
We should print some sentences in the training set to gain some insight

```
for idx in range(10):

    print(X_train[idx], label_to_emoji(Y_train[idx]))
```

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

This is the model that we are aiming to build:



Input and Output:

- 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.

To get your labels suitable for training a Softmax classifier, convert  $Y$  from its current shape  $(m, 1)$  into a "one hot representation"  $(m, 5)$  where each row is a one-hot vector giving the label of one example.

```
Y_oh_train = convert_to_one_hot(Y_train, C = 5)
Y_oh_test = convert_to_one_hot(Y_test, C = 5)
```

Let's observe an example:

```
idx = 34
```

```
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 'she is attractive' has label index 0, which is emoji ❤️

Label index 0 in one-hot encoding format is [1. 0. 0. 0. 0.]

---

## Implementation

For this task, we are going to use the pre-trained 50-dimensional GloVe embeddings. First and foremost, let's load the vector representations.

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

We obtain:

- `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.

Let's check if we got the desired results:

```
word = "banana"  
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 banana in the vocabulary is 67752

the 289846th word in the vocabulary is potatos

## Sentence to Average

Firstly, given an input sentence, we need to split it into a list of distinct words. And for each word in the list, we want to access its GloVe representation. Then, we need to take the average of all of the word vectors.

```
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 = next(iter(word_to_vec_map.keys()))

    # Step 1: Split sentence into list of lower case words
    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)

    count = 0

    # Step 2: average the word vectors. You can loop over the words in
```

```

the list "words".
for w in words:
    # Check that word exists in word_to_vec_map
    if w in word_to_vec_map:
        avg += word_to_vec_map[w]
        # Increment count
        count +=1
if count > 0:
    avg = avg / count

return avg

```

## Model

Implement the `model()` function described in the figure above.

- The variable  $Y_{oh}$  ("Y one hot") is the one-hot encoding of the output labels.

$$z^{(i)} = Wavg^{(i)} + b$$

$$a^{(i)} = softmax(z^{(i)})$$

$$\mathcal{L}^{(i)} = - \sum_{k=0}^{n_y-1} Y_{oh,k}^{(i)} * \log(a_k^{(i)})$$

```

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 = next(iter(word_to_vec_map.keys()))

# 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,))

# Convert Y to Y_onehot with n_y classes
Y_oh = convert_to_one_hot(Y, C = n_y)

# Optimization loop
for t in range(num_iterations): # Loop over the number of iterations
    cost = 0
    dW = 0
    db = 0
    for i in range(m): # Loop over the training examples
        # Average the word vectors of the words from the i'th
training example
        avg = sentence_to_avg(X[i], word_to_vec_map)

        # Forward propagate the avg through the softmax layer.
        z = np.dot(W, avg) + b
        a = softmax(z)

```

```

        # Add the cost using the i'th training label's one hot
        representation and "A" (the output of the softmax)

        cost += -np.sum(Y_oh[i] * np.log(a))

    # Compute gradients
    dz = a - Y_oh[i]
    dW += np.dot(dz.reshape(n_y,1), avg.reshape(1, n_h))
    db += dz

    # Update parameters with Stochastic Gradient Descent
    W = W - learning_rate * dW
    b = b - learning_rate * db

    if t % 100 == 0:
        print("Epoch: " + str(t) + " --- cost = " + str(cost))
        pred = predict(X, Y, W, b, word_to_vec_map)

    return pred, W, b

```

Let's train the model and learn the softmax parameters ( $W, b$ )\

```

np.random.seed(1)

pred, W, b = model(X_train, Y_train, word_to_vec_map)

```

```

Epoch: 0 --- cost = 410.43365788314725
Accuracy: 0.5454545454545454
Epoch: 100 --- cost = 27.192408144807885
Accuracy: 0.9621212121212122
Epoch: 200 --- cost = 1.4966116382389523
Accuracy: 1.0
Epoch: 300 --- cost = 0.34893019535338765
Accuracy: 1.0

```

Let's apply the model in the evaluation set:



```

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: 0.8392857142857143

*!!! Please note that with random guesswork, we would have only had 20% accuracy. Therefore, with only 127 training examples, this proves to be quite effective.*

*!!! Testing with a word (treasure) which was not included in the training set, yet the model was still able to figure out the tone from the speaker.*

```

X_my_sentences = np.array(["i treasure you", "i love you", "funny lol",
                           "lets play with a ball", "food is ready", "today is not good"])
Y_my_labels = np.array([[0], [0], [2], [1], [4], [3]])

pred = predict(X_my_sentences, Y_my_labels , W, b, word_to_vec_map)
print_predictions(X_my_sentences, pred)

```

Accuracy: 0.8333333333333334

```

i treasure you ❤️
i love you ❤️
funny lol :smile:
lets play with a ball ⚾️
food is ready 🍴
today is not good :smile:

```

This is because *treasure* has a similar embedding as *love*, the algorithm has generalized correctly even to a word it has never seen before. Words such as *heart*, *dear*, *beloved* or *adore* have embedding vectors similar to *love*.

## Flaws

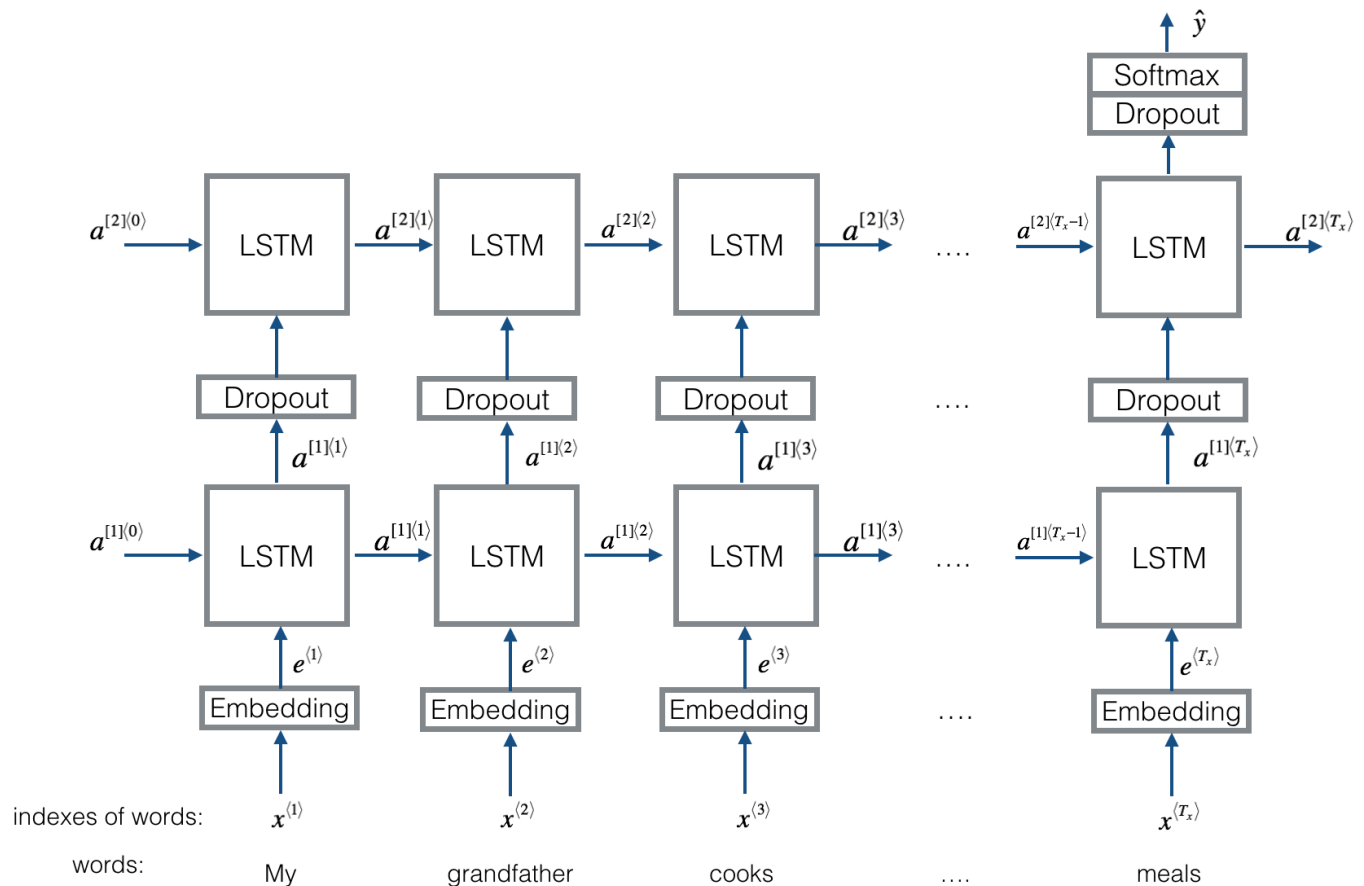
The flaws of this basic model are that it does not understand the combination of words, nor the ordering of them. Instead, it only averages all the words' embedding vectors together. This is when we need to turn to Long Short Term Memory (LSTM) model, a complete upgrade of the Recurrent Neural Network (RNN).

---

## LSTM Model Emojifier-V2

```
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)
```

This is the overview of the model



We are training using mini batches. However, for vectorization to work, we need to handle sequences of different lengths via *setting a maximum length* and *padding sequences* to get the same length.

For example

- 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  $(e_I, e_{love}, e_{you}, \vec{0}, \vec{0}, \dots, \vec{0})$ .
- 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.

## The Embedding Layer

The embedding matrix (layer) 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."

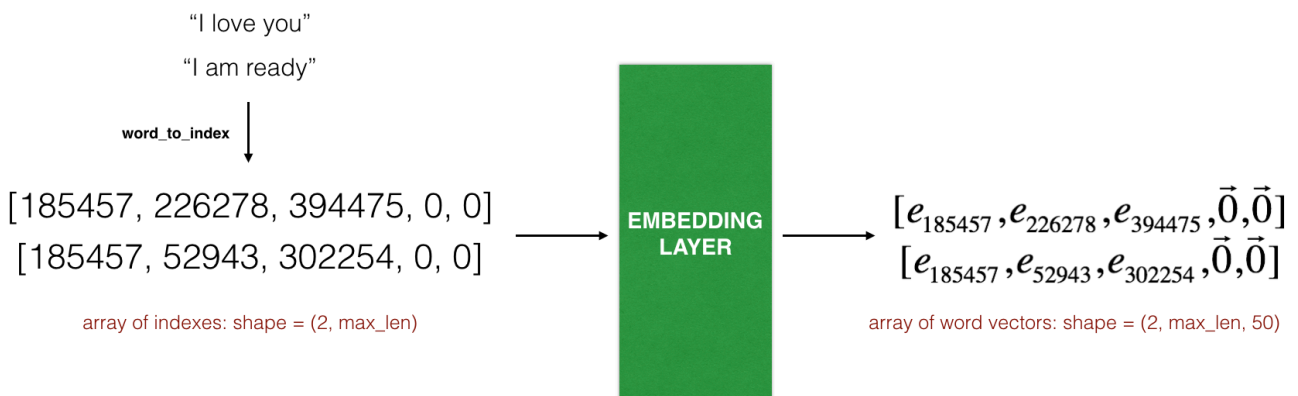
The `Embedding()` layer's input is an integer matrix of size **(batch size, max input length)**.

- 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 below 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)`.



## Sentence to Indices

This function processes an array of sentences  $X$  and return inputs to the embedding layer.

```
def sentences_to_indices(X, word_to_index, max_len):
    """
    Converts an array of sentences (strings) into an array of indices
    corresponding to words in the sentences.
    The output shape should be such that it can be given to
```

``Embedding()`` (described in Figure 4).

Arguments:

`X` -- array of sentences (strings), of shape `(m,)`

`word_to_index` -- a dictionary containing the each word mapped to its index

`max_len` -- maximum number of words in a sentence. You can assume every sentence in `X` is no longer than this.

Returns:

`X_indices` -- array of indices corresponding to words in the sentences from `X`, of shape `(m, max_len)`

"""

```
m = X.shape[0] # number of training examples
```

```
X_indices = np.zeros((m, max_len))
```

```
for i, sentence in enumerate(X):
```

```
    sentence_words = sentence.lower().split()
```

```
    for j, w in enumerate(sentence_words):
```

```
        if j < max_len: # Limit the length of the sentence
```

```
            X_indices[i, j] = word_to_index.get(w, 0) # Use 0 for  
unknown words
```

```
        else:
```

```
            break
```

```
return X_indices
```

Let's test it

```
X1 = np.array(["funny lol", "lets play baseball", "food is ready for  
you"])
```

```
X1_indices = sentences_to_indices(X1, word_to_index, max_len=5)
```

```
print("X1 =", X1)
```

```
print("X1_indices =\n", X1_indices)
```

```
X1 = ['funny lol' 'lets play baseball' 'food is ready for you']
X1_indices =
[[155345. 225122.      0.      0.      0.]
 [220930. 286375. 69714.      0.      0.]
 [151204. 192973. 302254. 151349. 394475.]]
```

## Pretrained Embedding Layer

We are ready to build the layer

1. Initialize the embedding matrix as a numpy array of zeroes
  - The embedding matrix has a row for each unique word in the vocabulary
  - There is an additional row to handle "unknown" words
  - Each row will store the vector representation of that word (50-dim GloVe)
2. Fill in each row of the embedding matrix with the vector representation of one 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 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, you don't want the model to modify the word embeddings.

4. Set the embedding weights to be equal to the embedding matrix.

```
def pretrained_embedding_layer(word_to_vec_map, word_to_index):
    """
```

Creates a Keras Embedding() layer and loads in pre-trained GloVe 50-dimensional vectors.

Arguments:

`word_to_vec_map` -- dictionary mapping words to their GloVe vector

representation.

word\_to\_index -- dictionary mapping from words to their indices in the vocabulary (400,001 words)

Returns:

embedding\_layer -- pretrained layer Keras instance

"""

vocab\_size = len(word\_to\_index) + 1 # adding 1 to fit Keras embedding to handle unknown words

any\_word = next(iter(word\_to\_vec\_map.keys()))

emb\_dim = word\_to\_vec\_map[any\_word].shape[0] # define dimensionality of your GloVe word vectors (= 50)

# Initialize the embedding matrix as a numpy array of zeros.

emb\_matrix = np.zeros((vocab\_size, emb\_dim))

# Set each row "idx" of the embedding matrix to be the word vector representation of the idx'th word of the vocabulary

for word, idx in word\_to\_index.items():

emb\_matrix[idx, :] = word\_to\_vec\_map[word]

# Define Keras embedding layer with the correct input and output sizes

# Make it non-trainable.

embedding\_layer = Embedding(vocab\_size, emb\_dim, trainable = False)

# Build the embedding layer, it is required before setting the weights of the embedding layer.

embedding\_layer.build((None,))

# Set the weights of the embedding layer to the embedding matrix. Your layer is now pretrained.

embedding\_layer.set\_weights([emb\_matrix])

return embedding\_layer

Let's see the results

```
embedding_layer = pretrained_embedding_layer(word_to_vec_map,
word_to_index)

print("weights[0][1][1] =", embedding_layer.get_weights()[0][1][1])
print("Input_dim", embedding_layer.input_dim)
print("Output_dim", embedding_layer.output_dim)
```

```
weights[0][1][1] = 0.39031
Input_dim 400001
Output_dim 50
```

---

## Implementation

Refer to these Keras layers:

[Input\(\)](#)

- Set the `shape` and `dtype` parameters.
- The inputs are integers, so you can specify the data type as a string, 'int32'.

[LSTM\(\)](#)

- Set the `units` and `return_sequences` parameters.

[Dropout\(\)](#)

- Set the `rate` parameter.

[Dense\(\)](#)

- Set the `units`.

[Activation\(\)](#)

- You can pass in the activation of your choice as a lowercase string.

[Model\(\)](#)

- Set `inputs` and `outputs`.

```
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 representation   
word\_to\_index -- dictionary mapping from words to their indices in the vocabulary (400,001 words)

Returns:

model -- a model instance in Keras

"""

# Define sentence\_indices as the input of the graph.   
sentence\_indices = Input(input\_shape, dtype='int32')

# Create the embedding layer pretrained with GloVe Vectors   
embedding\_layer = pretrained\_embedding\_layer(word\_to\_vec\_map,   
word\_to\_index)

# Propagate sentence\_indices through your embedding layer   
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 through 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)(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)

return model

```

Let's create the model and see the summary

```

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

```

Model: "functional\_3"

Layer (type)	Output Shape	Param #
input_layer_3 (InputLayer)	(None, 10)	0
embedding_7 (Embedding)	(None, 10, 50)	20,000,050
lstm_6 (LSTM)	(None, 10, 128)	91,648
dropout_6 (Dropout)	(None, 10, 128)	0
lstm_7 (LSTM)	(None, 128)	131,584
dropout_7 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 5)	645
activation_3 (Activation)	(None, 5)	0

Total params: 20,223,927 (77.15 MB)

Trainable params: 223,877 (874.52 KB)

Non-trainable params: 20,000,050 (76.29 MB)

A breakdown to what we are seeing

- Because all sentences in the dataset are less than 10 words, `max_len=10` was chosen.
- The architecture has 20,223,927 parameters, where 20,000,050 (the word embeddings) are non-trainable, with the remaining 223,877 are trainable.
- Because your vocabulary size has 400,001 words (with valid indices from 0 to 400,000) there are  $400,001 \times 50 = 20,000,050$  non-trainable parameters.

---

## Training the Model

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

```
X_train_indices = sentences_to_indices(X_train, word_to_index, maxLen)
```

```
Y_train_oh = convert_to_one_hot(Y_train, C = 5)
```

```
model.fit(X_train_indices, Y_train_oh, epochs = 50, batch_size = 32,  
shuffle=True)
```

```
Epoch 1/50  
5/5 ————— 1s 9ms/step - accuracy: 0.2182 - loss: 1.5971  
Epoch 2/50  
5/5 ————— 0s 6ms/step - accuracy: 0.3009 - loss: 1.5234  
Epoch 3/50  
5/5 ————— 0s 9ms/step - accuracy: 0.4027 - loss: 1.4504  
Epoch 4/50  
5/5 ————— 0s 9ms/step - accuracy: 0.4198 - loss: 1.4171  
Epoch 5/50  
5/5 ————— 0s 8ms/step - accuracy: 0.6293 - loss: 1.2984  
Epoch 6/50  
5/5 ————— 0s 8ms/step - accuracy: 0.6014 - loss: 1.1662  
Epoch 7/50  
5/5 ————— 0s 10ms/step - accuracy: 0.5918 - loss: 1.0730  
Epoch 8/50  
5/5 ————— 0s 10ms/step - accuracy: 0.6661 - loss: 0.8569  
Epoch 9/50  
5/5 ————— 0s 9ms/step - accuracy: 0.7171 - loss: 0.7441  
Epoch 10/50  
5/5 ————— 0s 9ms/step - accuracy: 0.7483 - loss: 0.6662  
Epoch 11/50  
5/5 ————— 0s 9ms/step - accuracy: 0.7277 - loss: 0.6366  
Epoch 12/50  
5/5 ————— 0s 10ms/step - accuracy: 0.7751 - loss: 0.6190  
Epoch 13/50  
...  
Epoch 49/50  
5/5 ————— 0s 11ms/step - accuracy: 0.9866 - loss: 0.0394  
Epoch 50/50  
5/5 ————— 0s 11ms/step - accuracy: 0.9828 - loss: 0.0278
```

98% accuracy is pretty impressive, how about on the evaluation set

```
X_test_indices = sentences_to_indices(X_test, word_to_index, max_len =
maxLen)
Y_test_oh = convert_to_one_hot(Y_test, C = 5)

loss, acc = model.evaluate(X_test_indices, Y_test_oh)

print()
print("Test accuracy = ", acc)
```

Test accuracy = 0.8392857313156128

Now, we can test it with our model

```
# Change the sentence below to see your prediction. Make sure all the
words are in the Glove embeddings.
x_test = np.array(['I treasure you'])
X_test_indices = sentences_to_indices(x_test, word_to_index, maxLen)

print(x_test[0] +' '+
label_to_emoji(np.argmax(model.predict(X_test_indices))))
```

*!!! The Emojifier-V1 model did not predict "not feeling happy" correctly, but the Emojify-V2 got it right! If it didn't, be aware that Keras' outputs are slightly random each time, so this is probably why.*

*!!! The current model still falls short at understanding negation (such as "not happy"). This is mainly due to small training set and does not have a lot of negation examples.*

---

## Concluding Thoughts

1. If you have an NLP task where the training set is small, using word embeddings can help your algorithm significantly.
2. Word embeddings allow your model to work on words in the test set that may not even appear in the training set.
3. 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.
4. `LSTM()` has a flag called `return_sequences` to decide if you would like to return all hidden states or only the last one.
- You can use `Dropout()` right after `LSTM()` to regularize your network.