In [1]:

```
import keras
import numpy as np

from keras.models import Sequential
```

```
Using TensorFlow backend.
```

Sentiment Classification

In this problem we will use Kera's imdb sentiment dataset. You will take in sequences of words and use an RNN to try to classify the sequences sentiment. These sentences are movie reviews, so the sentiment reflects whether its a positive review (sentiment of 1) or a negative review (sentiment of 0).

First we have to process the data a little bit, so that we have fixed length sequences.

The data is given to us in integer form, so each integer represents a unique word, 0 represents a PAD character, 1 represents a START character and 2 represents a character that is unknown because it is not in the top `num_words` . Thus 3 represents the first real word.

Also the words are in decreasing order of commonness, so the word that 3 represents is the most common word in the dataset. (It happens to be `the`)

In [2]:

```
from keras.datasets import imdb
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=1000, maxlen=200,
```

We want to process the data into arrays of sequences that are all length 200. If a given sequence is shorter than 200 tokens we want to pad the rest of the sequence out with zeros so that the sequence is 200 long.

In [3]:

```
def process_data(data):
    processed = np.zeros(len(data) * 200).reshape((len(data), 200))
    for i, seq in enumerate(data):
        if len(seq) < 200:
            processed[i] = np.pad(np.array(seq), (0, 200-len(seq)), mode='constant
            # PAD SEQUENCES WITH ZEROS HERE
        else:
            processed[i] = np.array(seq)
    return processed
```

In [4]:

```
x_train_proc = process_data(x_train)
x_test_proc = process_data(x_test)
```

In [5]:

```
imdb_model = Sequential()
```

Now we want to add an embedding layer. The purpose of an embedding layer is to take a sequence of integers representing words in our case and turn each integer into a dense vector in some embedding space. (This is essentially the idea of Word2Vec). We want to create an embedding layer with vocab size equal to the max num words we allowed when we loaded the data (in this case 1000), and a fixed dense vector of size 32. Then we have to specify the max length of our sequences and we want to mask out zeros in our sequence since we used zero to pad. Use the docs for embedding layer to fill out the missing entries: [https://keras.io/layers/embeddings/ (https://keras.io/layers/embeddings/)](https://keras.io/layers/embeddings/)

In [6]:

```
from keras.layers.embeddings import Embedding
imdb_model.add(Embedding(input_dim=1000, output_dim=32, input_length=200, mask_zer
```

(a) Add an LSTM layer with 32 outputs, then a Dense layer with 16 neurons, then a relu activation, then a dense layer with 1 neuron, then a sigmoid activation. Then you should print out the model summary. The Keras documentation is here: [https://keras.io/ (https://keras.io/)](https://keras.io/)

In [7]:

```
from keras.layers.recurrent import LSTM
from keras.layers import Dense, Activation
imdb_model.add(LSTM(units=32))
```

In [8]:

```
imdb_model.add(Dense(units=16))
imdb_model.add(Activation('relu'))
```

In [9]:

```
imdb_model.add(Dense(units=1))
imdb_model.add(Activation('sigmoid'))
```

In [10]:

```
imdb_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_1 (Embedding)      (None, 200, 32)           32000     
_________________________________________________________________
lstm_1 (LSTM)                (None, 32)                8320      
_________________________________________________________________
dense_1 (Dense)              (None, 16)                528       
_________________________________________________________________
activation_1 (Activation)    (None, 16)                0         
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 17        
_________________________________________________________________
activation_2 (Activation)    (None, 1)                 0         
=================================================================
Total params: 40,865
Trainable params: 40,865
Non-trainable params: 0
_________________________________________________________________
```

If you did the above parts correctly running `imdb_model.summary()`

should give you the following output.

In [11]:

```
#from IPython.display import Image
#Image(filename='/home/james/Desktop/Screenshot from 2018-08-20 21-04-15.png')
```

(b) Now compile the model with binary cross entropy, and the adam optimizer. Also include accuracy as a metric in the compile. Then train the model on the processed data (no need to worry about class weights this time)

In [12]:

```
imdb_model.compile(loss="binary_crossentropy", optimizer="Adam", metrics=["accurac
```

In [13]:

```
imdb_model.fit(x_train_proc, y_train)
```

```
Epoch 1/1
25000/25000 [==============================] - 135s 5ms/step - loss: 0.4158 -
acc: 0.8058
```

Out[13]:

```
<keras.callbacks.History at 0x1d39499ab00>
```

In [14]:

```
print("Accuracy: ", imdb_model.evaluate(x_test_proc, y_test)[1])
```

```
3913/3913 [==============================] - 5s 1ms/step
Accuracy:  0.8573984155836478
```

Now we can look at our predictions and the sentences they correspond to.

In [15]:

```
y_pred = imdb_model.predict(x_test_proc)
```

In [16]:

```
word_to_id = keras.datasets.imdb.get_word_index()
word_to_id = {k:(v+3) for k,v in word_to_id.items()}
word_to_id["<PAD>"] = 0
word_to_id["<START>"] = 1
word_to_id["<UNK>"] = 2

id_to_word = {value:key for key,value in word_to_id.items() if value < 1000}
def get_words(token_sequence):
    return ' '.join(id_to_word[token] for token in token_sequence)

def get_sentiment(y_pred, index):
    return 'Positive' if y_pred[index] else 'Negative'
```

In [17]:

```
y_pred = np.vectorize(lambda x: int(x >= 0.5))(y_pred)
correct = []
incorrect = []
for i, pred in enumerate(y_pred):
    if y_test[i] == pred:
        correct.append(i)
    else:
        incorrect.append(i)
```

Now we print out one of the sequences we got correct.

In [18]:

```
print(get_sentiment(y_pred, correct[10]))
print(get_words(x_test[correct[10]]))
```

```
Negative
<START> don't tell me this film was funny or a little funny it was a complete
<UNK> and one of the worst movies i've ever seen <UNK> <UNK> is only funny on
<UNK> <UNK> <UNK> <UNK> show after watching his performance all i can say is h
e is not made for movies with a <UNK> script or more like no storyline there's
nothing to keep you <UNK> full of annoying <UNK> <UNK> this movie is a complet
e <UNK> all the way at the end of the film <UNK> <UNK> gives a <UNK> he <UNK>
if you <UNK> this film tell people it was good not even the <UNK> could save t
he movie he probably knew its <UNK> be a <UNK> i would of given this a <UNK> 1
0 but the <UNK> start is 1 overall don't even waste your time on this <UNK>
```

And one we got wrong.

In [19]:

```
print(get_sentiment(y_pred, incorrect[10]))
print(get_words(x_test[incorrect[10]]))
```

```
Negative
<START> why do people need to follow the opinion of the <UNK> of <UNK> and <UN
K> <UNK> <UNK> directed by the brilliant <UNK> <UNK> who has a small role in t
he film too is another <UNK> <UNK> <UNK> as such it is quite good and entertai
ning <UNK> anyone who goes to see it has this in mind or read the book which i
s no better even <UNK> <UNK> <UNK> fans myself <UNK> knew it would be a <UNK>
of her again playing the love interest of her <UNK> <UNK> even as such the fil
m is <UNK> what's so bad about this movie that is much better in the other muc
h <UNK> <UNK> <UNK> <UNK> <UNK> this film is no masterpiece but it's not as ba
d as the <UNK> would have the potential viewer believe
```

As you can see the amount of UNKNOWN characters in the sequence cause by having only 1000 vocab words is hurting our performance. See if you can go through and increase the number of vocab words to 2000. HINT: you have to change two places in the above code.

In [20]:

```
from keras import backend as K
```

Embedding Exploration

Another interesting thing to do is see if our learned embeddings mean anything reasonable.

In [21]:

```
# this function takes a list of token sequences as inputs
# and outputs the corresponding vector outputs of our `Embedding` layer
embedding_func = K.function([imdb_model.inputs[0]], [imdb_model.layers[0].output])
```

In [22]:

```
# this function outputs the embedding of a given word using above function
def word_to_embedding(word):
    token = word_to_id[word]
    seq = [token]
    sequences = [seq]
    inputs = [process_data(sequences)]
    embedding = embedding_func(inputs)
    return embedding[0][0][0]
```

In [23]:

```
valid_words = [word for word, token in word_to_id.items() if token < 1000]
```

In [24]:

```
valid_word_embeddings = {word: word_to_embedding(word) for word in valid_words}
```

Since we used an embedding layer with an output size of 32, our embeddings are going to be 32-dimensional vectors. Humans can't effectively visualize beyond 3 (maybe 4) dimensions so we want to use a dimensionality reduction technique to make our embeddings more visualizable. One such technique is Principal Component Analysis or PCA. The library scikit-learn provides an easy to use API for this technique.

In [25]:

```
import sklearn
from sklearn import decomposition
```

using the documentation for scikit-learn's PCA [here (http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html)](http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html): create a PCA object with `n_components=2`

In [35]:

```
pca = decomposition.PCA(n_components=2)
```

using the same documentation find the function to fit the PCA transform to the provided embedding vectors. This step essentially. For the curious, this step essentially finds the 2 dimensions (since we specified `n_components=2` that explain the most variance of the dataset, in other words the two dimensions that are most representative of the deviations of any one sample to another. So these 2 dimensions are the most important and therefore the best to visualize.

In [36]:

```
vectors_to_fit = np.array([x for x in valid_word_embeddings.values()])
pca.fit(vectors_to_fit)
```

Out[36]:

```
PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)
```

Now we want to visualize our embeddings in these new PCA dimensions, so using the same documentation from above fill out the missing spots in the code below to transform the embeddings into the pca dimensions.

In [37]:

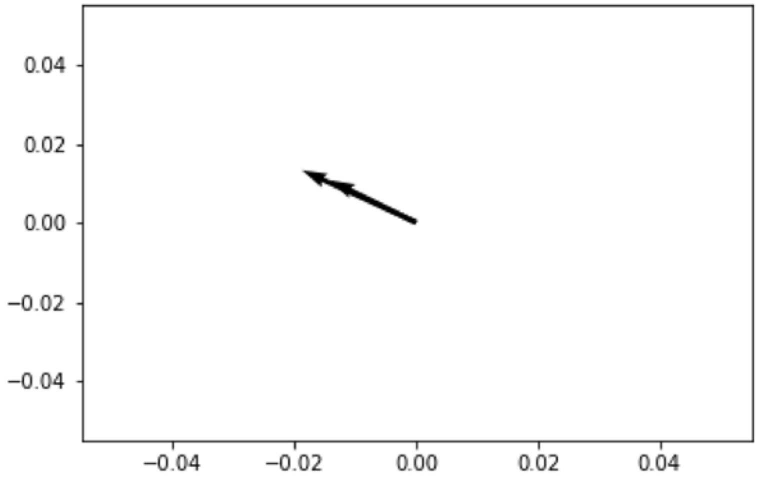
```
import matplotlib.pyplot as plt
%matplotlib inline
def get_pca_words(words):
    embeddings = [valid_word_embeddings[word] for word in words]
    pcas = [pca.transform(embedding.reshape(1, -1)) for embedding in embeddings]
    return pcas

def plot_pca_words(words, scale=1):
    pcas = get_pca_words(words)
    zeros = [0 for _ in pcas]
    x_start = zeros
    y_start = zeros
    xs = [p[0, 0] for p in pcas]
    ys = [p[0, 1] for p in pcas]
    plt.quiver(x_start, y_start, xs, ys, scale=scale)
    plt.show()
```

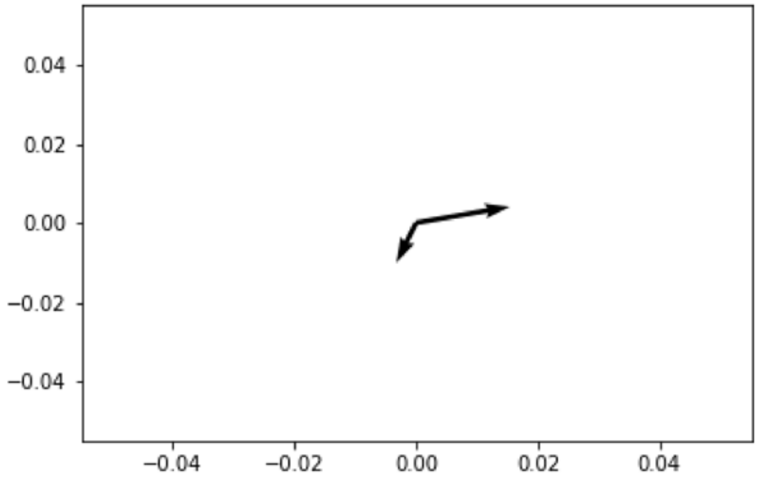
Now using the above functions we can plot the corresponding pca vectors of any words we like. Below are some good examples of pairs of words that are similar within the movie review context and their corresponding vectors are also similar. This is a good sign. This means the embedding we have learned is likely doing something somewhat reasonable.

In [38]:

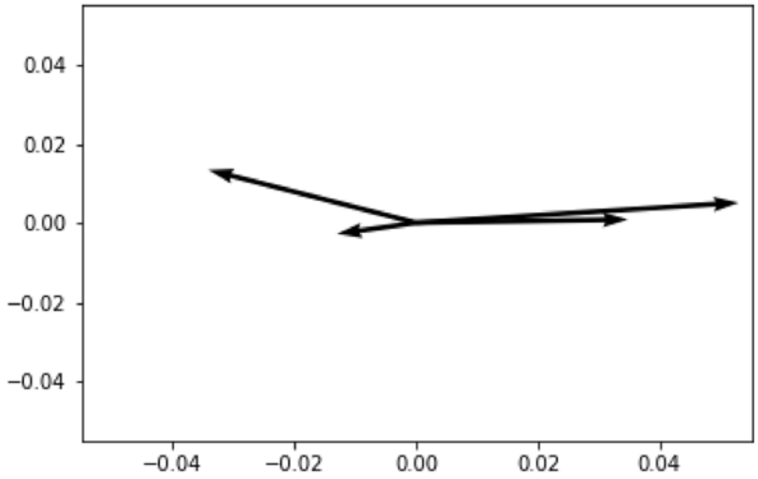
```
plot_pca_words(['film', 'entertainment'], scale=0.5)
```



In [39]:
```
plot_pca_words(['man', 'woman'])
```

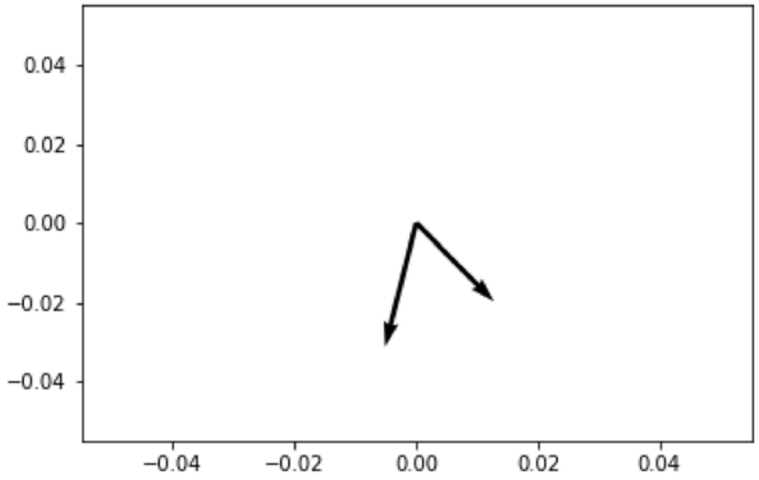


In [40]:
```
plot_pca_words(['good', 'bad', 'horrible', 'great'], scale=2)
```

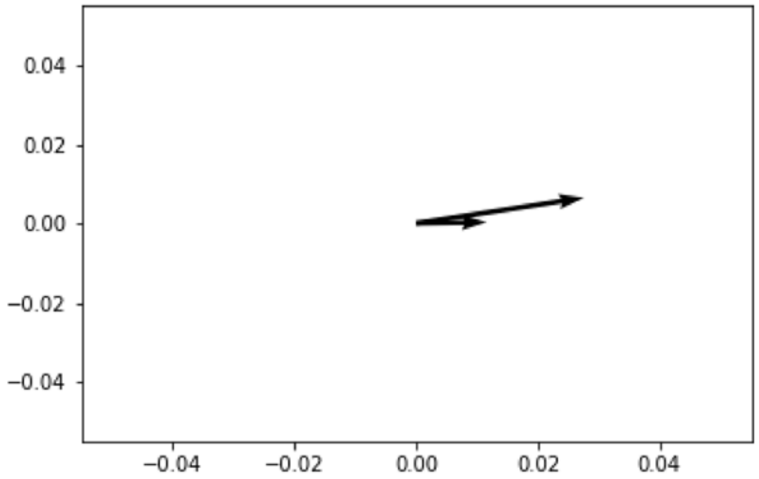


Now find 2 more pairs of words that are similar in PCA'd embedding space.

```
In [44]: # plot pair of similar words here
         plot_pca_words(['but', 'however'])
```



```
In [47]: # plot another pair of similar words here
         plot_pca_words(['worst', 'bad'], scale=6)
```



Given that the task we learned these embeddings for was sentiment classification, the embeddings are typically more meaningful for adjectives. Write a sentence or two about why you think this last statement makes sense intuitively.

Answer

- Since adjectives like "week", "strong", "lovely" and "ugly" have very strong sentiment.
- Also, in daily communication, people tend to prefer expressing their emotions using adjectives instead of verbs.

Now just for fun we can write a function that gives us the 10 closest words to a provided word.

In [48]:

```
def word_to_angle(word):
    p = pca.transform(valid_word_embeddings[word].reshape(-1, 1))
    return np.arctan(p[0, 1] / p[0, 0])
valid_word_angles = [word_to_angle(word) for word in valid_words]
```

In [49]:

```
def find_closest_n(value, n):
    indices = np.argsort(np.abs(np.array(valid_word_angles) - value))
    return [(valid_words[ind], valid_word_angles[ind]) for ind in indices[:n]]
```

In [50]:

```
find_closest_n(word_to_angle('terrible'), 10)
```

Out[50]:

```
[('terrible', 0.55044174),
 ('awful', 0.5507352),
 ('boring', 0.5507431),
 ('worst', 0.550116),
 ('fails', 0.5508857),
 ('worse', 0.55098945),
 ('waste', 0.5510491),
 ('poor', 0.55105877),
 ('annoying', 0.5512351),
 ('horrible', 0.5512729)]
```

In [ ]:

```
```