Assignment 8: Language Modeling With an RNN

In Assignmnet 8, we predict the thumb's up or down rating of movie reviews. I experimented with 4 models changing two inputs: pretrained word vectors and size of the vocabulary. I change the pretrained word vectors between 50 and 300 based on global vectors. I changed the number of vocabulary words between 20,000 and 40,000. The results are presented in the table and the end of the code. I have found that the optimal model uses 20,000 vocablary words and GloVe 50 pretrained word vectors to produce a training accuracy of 79% and a testing accuracy of 67%.

Following the experimentation with the vocabulary size and word vectors. I run 4 more experiments to test hyperparameters: learning rate, bach size, number of epochs, & number of neurons. The best best accuracy resulted from changing the number of epochs from 50 to 100. This resulted in a train accuracy of 87% and test accuracy of 72%.

The recommended model produces a result of 67%, but I would consider experimenting further to improve the output. To make an automated customer support system that is capable of identify negative customer feelings, the company would need to increase the test accuracy by further experimentation with the vectors, vocabulary size, and hyerparameters. To make language models more useful in customer service, a company could request a quick survey to see if language processor was accurate. Therefore, the system would have labels and the data scientist could manage the model to ensure that, as more categorical results are provided, the model improves.

First, I install the Python chakin package, obtain GloVe (and perhaps non-GloVe) embeddings. Then, I load and run jump-start code for the assignment, which uses pretrained word vectors from GloVe.6B.50d, a vocabulary of 10,000 words, and movie review data.

In [364]:
```
# Gather embeddings via chakin

# As originally configured, this program downloads four
# pre-trained GloVe embeddings, saves them in a zip archive,
# and then unzips the archive to create the four word-to-embeddings
# text files for use in language models.

# Note that the downloading process can take about 10 minutes to complete.

import numpy as np
import tensorflow as tf
```

In [365]:
```
import chakin
```

In [366]:
```
import json
import os
from collections import defaultdict
```

In [367]:
```
# Specify English embeddings file to download and install
# by index number, number of dimensions, and subfoder name
# Note that GloVe 50-, 100-, 200-, and 300-dimensional folders
# are downloaded with a single zip download
CHAKIN_INDEX = 11
NUMBER_OF_DIMENSIONS = 50
SUBFOLDER_NAME = "gloVe.6B"

DATA_FOLDER = "embeddings"
ZIP_FILE = os.path.join(DATA_FOLDER, "{}.zip".format(SUBFOLDER_NAME))
ZIP_FILE_ALT = "glove" + ZIP_FILE[5:]  # sometimes it's lowercase only...
UNZIP_FOLDER = os.path.join(DATA_FOLDER, SUBFOLDER_NAME)
if SUBFOLDER_NAME[-1] == "d":
    GLOVE_FILENAME = os.path.join(
        UNZIP_FOLDER, "{}.txt".format(SUBFOLDER_NAME))
else:
    GLOVE_FILENAME = os.path.join(UNZIP_FOLDER, "{}.{}d.txt".format(
        SUBFOLDER_NAME, NUMBER_OF_DIMENSIONS))


if not os.path.exists(ZIP_FILE) and not os.path.exists(UNZIP_FOLDER):
    print("Downloading embeddings to '{}'".format(ZIP_FILE))
    chakin.download(number=CHAKIN_INDEX, save_dir='./{}'.format(DATA_FOLDER))
else:
    print("Embeddings already downloaded.")
```

```
Embeddings already downloaded.
```

In [368]:
```
if not os.path.exists(UNZIP_FOLDER):
    import zipfile
    if not os.path.exists(ZIP_FILE) and os.path.exists(ZIP_FILE_ALT):
        ZIP_FILE = ZIP_FILE_ALT
    with zipfile.ZipFile(ZIP_FILE, "r") as zip_ref:
        print("Extracting embeddings to '{}'".format(UNZIP_FOLDER))
        zip_ref.extractall(UNZIP_FOLDER)
else:
    print("Embeddings already extracted.")

print('\nRun complete')
```

```
Embeddings already extracted.

Run complete
```

I ran the code 4 times, I changed the number of pre-defined vocabulary words between 20,000 and 40,000 words, and changed the pre-trained vocabulary vectors between 50 and 300

In [369]:
```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy as np

import os  # operating system functions
import os.path  # for manipulation of file path names

import re  # regular expressions

from collections import defaultdict

import nltk
from nltk.tokenize import TreebankWordTokenizer

import tensorflow as tf

RANDOM_SEED = 9999

# To make output stable across runs
def reset_graph(seed= RANDOM_SEED):
    tf.reset_default_graph()
    tf.set_random_seed(seed)
    np.random.seed(seed)

REMOVE_STOPWORDS = False  # no stopword removal

EVOCABSIZE = 20000  # specify desired size of pre-defined embedding vocabulary

embeddings_directory = 'embeddings/gloVe.6B'
filename = 'glove.6B.50d.txt'
embeddings_filename = os.path.join(embeddings_directory, filename)
# -------------------------------------------------------------
```

In [370]:
```python
def load_embedding_from_disks(embeddings_filename, with_indexes=True):
    """
    Read a embeddings txt file. If `with_indexes=True`,
    we return a tuple of two dictionnaries
    `(word_to_index_dict, index_to_embedding_array)`,
    otherwise we return only a direct
    `word_to_embedding_dict` dictionnary mapping
    from a string to a numpy array.
    """
    if with_indexes:
        word_to_index_dict = dict()
        index_to_embedding_array = []

    else:
        word_to_embedding_dict = dict()

    with open(embeddings_filename, 'r', encoding='utf-8') as embeddings_file:
        for (i, line) in enumerate(embeddings_file):

            split = line.split(' ')

            word = split[0]

            representation = split[1:]
            representation = np.array(
                [float(val) for val in representation]
            )

            if with_indexes:
                word_to_index_dict[word] = i
                index_to_embedding_array.append(representation)
            else:
                word_to_embedding_dict[word] = representation

    # Empty representation for unknown words.
    _WORD_NOT_FOUND = [0.0] * len(representation)
    if with_indexes:
        _LAST_INDEX = i + 1
        word_to_index_dict = defaultdict(
            lambda: _LAST_INDEX, word_to_index_dict)
        index_to_embedding_array = np.array(
            index_to_embedding_array + [_WORD_NOT_FOUND])
        return word_to_index_dict, index_to_embedding_array
    else:
        word_to_embedding_dict = defaultdict(lambda: _WORD_NOT_FOUND)
        return word_to_embedding_dict
```

In [371]:
```python
print('\nLoading embeddings from', embeddings_filename)
```

```
Loading embeddings from embeddings/gloVe.6B\glove.6B.50d.txt
```

In [372]:
```python
word_to_index, index_to_embedding = \
    load_embedding_from_disks(embeddings_filename, with_indexes=True)
```

In [373]:
```python
print("Embedding loaded from disks.")
```

```
Embedding loaded from disks.
```

In [374]:
```python
vocab_size, embedding_dim = index_to_embedding.shape
```

In [375]:
```python
print("Embedding is of shape: {}".format(index_to_embedding.shape))
```

```
Embedding is of shape: (400001, 50)
```

In [376]:
```python
print("This means (number of words, number of dimensions per word)\n")
print("The first words are words that tend occur more often.")
```

```
This means (number of words, number of dimensions per word)

The first words are words that tend occur more often.
```

In [377]:
```
#print("Note: for unknown words, the representation is an empty vector,\n"
#      "and the index is the last one. The dictionnary has a limit:")
#print("    {} --> {} --> {}".format("A word", "Index in embedding",
#      "Representation"))
#word = "worsdfkljsdf"  # a word obviously not in the vocabulary
#idx = word_to_index[word] # index for word obviously not in the vocabulary
#complete_vocabulary_size = idx
#embd = list(np.array(index_to_embedding[idx], dtype=int)) # "int" compact print
#print("    {} --> {} --> {}".format(word, idx, embd))
#word = "the"
#idx = word_to_index[word]
#embd = list(index_to_embedding[idx])  # "int" for compact print only.
#print("    {} --> {} --> {}".format(word, idx, embd))
```

In [378]:
```
#a_typing_test_sentence = 'The quick brown fox jumps over the lazy dog'
#print('\nTest sentence: ', a_typing_test_sentence, '\n')
#words_in_test_sentence = a_typing_test_sentence.split()

#print('Test sentence embeddings from complete vocabulary of',
#      complete_vocabulary_size, 'words:\n')
#for word in words_in_test_sentence:
#    word_ = word.lower()
#    embedding = index_to_embedding[word_to_index[word_]]
#    print(word_ + ": ", embedding)
```

In [379]:
```
def default_factory():
    return EVOCABSIZE  # last/unknown-word row in limited_index_to_embedding
# dictionary has the items() function, returns list of (key, value) tuples
```

In [380]:
```
limited_word_to_index = defaultdict(default_factory, \
    {k: v for k, v in word_to_index.items() if v < EVOCABSIZE})
```

In [381]:
```
# Select the first EVOCABSIZE rows to the index_to_embedding
limited_index_to_embedding = index_to_embedding[0:EVOCABSIZE,:]
```

In [382]:
```
# Set the unknown-word row to be all zeros as previously
limited_index_to_embedding = np.append(limited_index_to_embedding,
    index_to_embedding[index_to_embedding.shape[0] - 1, :].\
        reshape(1,embedding_dim),
    axis = 0)
```

In [383]:
```
# Delete large numpy array to clear some CPU RAM
del index_to_embedding

# Verify the new vocabulary: should get same embeddings for test sentence
#print('\nTest sentence embeddings from vocabulary of', EVOCABSIZE, 'words:\n')
#for word in words_in_test_sentence:
#    word_ = word.lower()
#    embedding = limited_index_to_embedding[limited_word_to_index[word_]]
#    print(word_ + ": ", embedding)
```

In [384]:
```
# ---------------------------------------------------------------
# code for working with movie reviews data
# Source: Miller, T. W. (2016). Web and Network Data Science.
#    Upper Saddle River, N.J.: Pearson Education.
#    ISBN-13: 978-0-13-388644-3
# This original study used a simple bag-of-words approach
# to sentiment analysis, along with pre-defined lists of
# negative and positive words.
# Code available at:  https://github.com/mtpa/wnds
# ---------------------------------------------------------------
# Utility function to get file names within a directory
def listdir_no_hidden(path):
    start_list = os.listdir(path)
    end_list = []
    for file in start_list:
        if (not file.startswith('.')):
            end_list.append(file)
    return(end_list)
```

In [385]:
```
# define list of codes to be dropped from document
# carriage-returns, line-feeds, tabs
codelist = ['\r', '\n', '\t']

# We will not remove stopwords in this exercise because they are
# important to keeping sentences intact
if REMOVE_STOPWORDS:
    print(nltk.corpus.stopwords.words('english'))

# previous analysis of a list of top terms showed a number of words, along
# with contractions and other word strings to drop from further analysis, add
# these to the usual English stopwords to be dropped from a document collection
    more_stop_words = ['cant','didnt','doesnt','dont','goes','isnt','hes',\
        'shes','thats','theres','theyre','wont','youll','youre','youve', 'br'\
        've', 're', 'vs']

    some_proper_nouns_to_remove = ['dick','ginger','hollywood','jack',\
        'jill','john','karloff','kudrow','orson','peter','tcm','tom',\
        'toni','welles','william','wolheim','nikita']

    # start with the initial list and add to it for movie text work
    stoplist = nltk.corpus.stopwords.words('english') + more_stop_words +\
        some_proper_nouns_to_remove
```

In [386]:
```
# text parsing function for creating text documents
# there is more we could do for data preparation
# stemming... looking for contractions... possessives...
# but we will work with what we have in this parsing function
# if we want to do stemming at a later time, we can use
#     porter = nltk.PorterStemmer()
# in a construction like this
#     words_stemmed =  [porter.stem(word) for word in initial_words]
def text_parse(string):
    # replace non-alphanumeric with space
    temp_string = re.sub('[^a-zA-Z]', '  ', string)
    # replace codes with space
    for i in range(len(codelist)):
        stopstring = ' ' + codelist[i] + '  '
        temp_string = re.sub(stopstring, '  ', temp_string)
    # replace single-character words with space
    temp_string = re.sub('\s.\s', ' ', temp_string)
    # convert uppercase to lowercase
    temp_string = temp_string.lower()
    if REMOVE_STOPWORDS:
        # replace selected character strings/stop-words with space
        for i in range(len(stoplist)):
            stopstring = ' ' + str(stoplist[i]) + ' '
            temp_string = re.sub(stopstring, ' ', temp_string)
    # replace multiple blank characters with one blank character
    temp_string = re.sub('\s+', ' ', temp_string)
    return(temp_string)
```

In [387]:
```
# ---------------------------------------------
# gather data for 500 negative movie reviews
# ---------------------------------------------
dir_name = 'movie-reviews-negative'
```

In [388]:
```
filenames = listdir_no_hidden(path=dir_name)
```

In [389]:
```
num_files = len(filenames)
```

In [390]:
```
for i in range(len(filenames)):
    file_exists = os.path.isfile(os.path.join(dir_name, filenames[i]))
    assert file_exists
print('\nDirectory:',dir_name)
print('%d files found' % len(filenames))
```

```
Directory: movie-reviews-negative
500 files found
```

In [391]:
```python
def read_data(filename):

  with open(filename, encoding='utf-8') as f:
    data = tf.compat.as_str(f.read())
    data = data.lower()
    data = text_parse(data)
    data = TreebankWordTokenizer().tokenize(data)  # The Penn Treebank

  return data

negative_documents = []

print('\nProcessing document files under', dir_name)
for i in range(num_files):
    ## print(' ', filenames[i])

    words = read_data(os.path.join(dir_name, filenames[i]))

    negative_documents.append(words)
```

```
Processing document files under movie-reviews-negative
```

In [392]:
```python
dir_name = 'movie-reviews-positive'
filenames = listdir_no_hidden(path=dir_name)
num_files = len(filenames)

for i in range(len(filenames)):
    file_exists = os.path.isfile(os.path.join(dir_name, filenames[i]))
    assert file_exists
print('\nDirectory:',dir_name)
print('%d files found' % len(filenames))

# Read data for positive movie reviews
def read_data(filename):

  with open(filename, encoding='utf-8') as f:
    data = tf.compat.as_str(f.read())
    data = data.lower()
    data = text_parse(data)
    data = TreebankWordTokenizer().tokenize(data)  # The Penn Treebank

  return data

positive_documents = []

print('\nProcessing document files under', dir_name)
for i in range(num_files):
    ## print(' ', filenames[i])

    words = read_data(os.path.join(dir_name, filenames[i]))

    positive_documents.append(words)
# -------------------------------------------------------
# convert positive/negative documents into numpy array
# -------------------------------------------------------
max_review_length = 0  # initialize
for doc in negative_documents:
    max_review_length = max(max_review_length, len(doc))
for doc in positive_documents:
    max_review_length = max(max_review_length, len(doc))
print('max_review_length:', max_review_length)
```

```
Directory: movie-reviews-positive
500 files found

Processing document files under movie-reviews-positive
max_review_length: 1052
```

```
In [393]: min_review_length = max_review_length  # initialize
          for doc in negative_documents:
              min_review_length = min(min_review_length, len(doc))
          for doc in positive_documents:
              min_review_length = min(min_review_length, len(doc))
          print('min_review_length:', min_review_length)

          # construct list of 1000 lists with 40 words in each list
          from itertools import chain
          documents = []
          for doc in negative_documents:
              doc_begin = doc[0:20]
              doc_end = doc[len(doc) - 20: len(doc)]
              documents.append(list(chain(*[doc_begin, doc_end])))
          for doc in positive_documents:
              doc_begin = doc[0:20]
              doc_end = doc[len(doc) - 20: len(doc)]
              documents.append(list(chain(*[doc_begin, doc_end])))

          # create list of lists of lists for embeddings
          embeddings = []
          for doc in documents:
              embedding = []
              for word in doc:
                 embedding.append(limited_index_to_embedding[limited_word_to_index[word]])
              embeddings.append(embedding)

          # -----------------------------------------------------
          # Check on the embeddings list of list of lists
          # -----------------------------------------------------
          # Show the first word in the first document
          test_word = documents[0][0]
          print('First word in first document:', test_word)
          print('Embedding for this word:\n',
                limited_index_to_embedding[limited_word_to_index[test_word]])
          print('Corresponding embedding from embeddings list of list of lists\n',
                embeddings[0][0][:])

          # Show the seventh word in the tenth document
          test_word = documents[6][9]
          #print('First word in first document:', test_word)
          #print('Embedding for this word:\n',
          #      limited_index_to_embedding[limited_word_to_index[test_word]])
          #print('Corresponding embedding from embeddings list of list of lists\n',
          #      embeddings[6][9][:])

          # Show the last word in the last document
          test_word = documents[999][39]
          #print('First word in first document:', test_word)
          #print('Embedding for this word:\n',
          #      limited_index_to_embedding[limited_word_to_index[test_word]])
          #print('Corresponding embedding from embeddings list of list of lists\n',
          #      embeddings[999][39][:])

          # -----------------------------------------------------
          # Make embeddings a numpy array for use in an RNN
          # Create training and test sets with Scikit Learn
          # -----------------------------------------------------
          embeddings_array = np.array(embeddings)

          # Define the labels to be used 500 negative (0) and 500 positive (1)
          thumbs_down_up = np.concatenate((np.zeros((500), dtype = np.int32),
                                np.ones((500), dtype = np.int32)), axis = 0)
```

```
min_review_length: 22
First word in first document: story
Embedding for this word:
 [ 0.48251    0.87746   -0.23455    0.0262     0.79691    0.43102
 -0.60902   -0.60764   -0.42812   -0.012523  -1.2894     0.52656
 -0.82763    0.30689    1.1972    -0.47674   -0.46885   -0.19524
 -0.28403    0.35237    0.45536    0.76853    0.0062157  0.55421
  1.0006    -1.3973    -1.6894     0.30003    0.60678   -0.46044
  2.5961    -1.2178     0.28747   -0.46175   -0.25943    0.38209
 -0.28312   -0.47642   -0.059444  -0.59202    0.25613    0.21306
 -0.016129  -0.29873   -0.19468    0.53611    0.75459   -0.4112
  0.23625    0.26451  ]
Corresponding embedding from embeddings list of list of lists
 [ 0.48251    0.87746   -0.23455    0.0262     0.79691    0.43102
 -0.60902   -0.60764   -0.42812   -0.012523  -1.2894     0.52656
 -0.82763    0.30689    1.1972    -0.47674   -0.46885   -0.19524
 -0.28403    0.35237    0.45536    0.76853    0.0062157  0.55421
  1.0006    -1.3973    -1.6894     0.30003    0.60678   -0.46044
  2.5961    -1.2178     0.28747   -0.46175   -0.25943    0.38209
 -0.28312   -0.47642   -0.059444  -0.59202    0.25613    0.21306
 -0.016129  -0.29873   -0.19468    0.53611    0.75459   -0.4112
  0.23625    0.26451  ]
```

In [396]:
```
# Scikit Learn for random splitting of the data
from sklearn.model_selection import train_test_split

# Random splitting of the data in to training (80%) and test (20%)
X_train, X_test, y_train, y_test = \
    train_test_split(embeddings_array, thumbs_down_up, test_size=0.20,
                     random_state = RANDOM_SEED)

# --------------------------------------------------------------------------
# We use a very simple Recurrent Neural Network for this assignment
# Géron, A. 2017. Hands-On Machine Learning with Scikit-Learn & TensorFlow:
#    Concepts, Tools, and Techniques to Build Intelligent Systems.
#    Sebastopol, Calif.: O'Reilly. [ISBN-13 978-1-491-96229-9]
#    Chapter 14 Recurrent Neural Networks, pages 390-391
#    Source code available at https://github.com/ageron/handson-ml
#    Jupyter notebook file 14_recurrent_neural_networks.ipynb
#    See section on Training an sequence Classifier, # In [34]:
#    which uses the MNIST case data...  we revise to accommodate
#    the movie review data in this assignment
# --------------------------------------------------------------------------
reset_graph()

n_steps = embeddings_array.shape[1]  # number of words per document
n_inputs = embeddings_array.shape[2]  # dimension of  pre-trained embeddings
n_neurons = 20  # analyst specified number of neurons
n_outputs = 2  # thumbs-down or thumbs-up

learning_rate = 0.001

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.int32, [None])

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)

logits = tf.layers.dense(states, n_outputs)
xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,
                                                          logits=logits)
loss = tf.reduce_mean(xentropy)
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)
correct = tf.nn.in_top_k(logits, y, 1)
accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

init = tf.global_variables_initializer()

n_epochs = 50
batch_size = 300
```

```
In [ ]: with tf.Session() as sess:
            init.run()
            for epoch in range(n_epochs):
                print('\n  ---- Epoch ', epoch, ' ----\n')
                for iteration in range(y_train.shape[0] // batch_size):
                    X_batch = X_train[iteration*batch_size:(iteration + 1)*batch_size,:]
                    y_batch = y_train[iteration*batch_size:(iteration + 1)*batch_size]
                    print('  Batch ', iteration, ' training observations from ',
                          iteration*batch_size, ' to ', (iteration + 1)*batch_size-1,)
                    sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
                acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
                acc_test = accuracy.eval(feed_dict={X: X_test, y: y_test})
                print('\n  Train accuracy:', acc_train, 'Test accuracy:', acc_test)
```

Vocabulary size	Pretrained word vectors	Train Accuracy	Test Accuracy
20,000	GloVe 50	79%	67%
40,000	GloVe 50	79%	66%
20,000	GloVe 300	100%	62%
40,000	GloVe 300	100%	65%

Learning rate	Batch size	Epochs	Neurons	Train Accuracy	Test Accuracy
0.01	100	50	20	100%	61%
0.001	200	50	20	79%	69%
0.001	100	100	20	87%	72%
0.001	100	50	40	96%	66%