Siddikov_Assignment_8_Final

August 18, 2019

0.0.1 Introduction

In this assignment, we are asked to explore the TensorFlow's Recurrent Neural Network (RNN) algorithm for predicting movie review sentiments. Each of the reviews has either a positive or negative sentiment, a thumbs up or a thumbs down, associated with the content.

For this problem we used a set of two pre-trained textual movie reviews embeddings from GloVe, short for Global Vectors for Word Representation, developed by researchers at Stanford University, to transform the written language content into its numeric representation. These encodings allow us to transform a sequence of words, or a sentence, into a sequence of numeric vectors of which we can derive mathematical models. We also used two vocabulary sizes.

We need to evaluate the four language models to classify movie reviews into positives and negatives and make recommendations to management. If the most critical customer messages can be identified, then customer support personnel can be assigned to contact those customers.

0.0.2 Import Packages

[1]:
```
%matplotlib inline
# ignore all future warnings
from warnings import simplefilter
simplefilter(action='ignore', category=FutureWarning)
import warnings
warnings.filterwarnings("ignore")

# import base packages into the namespace for this program
import numpy as np
import tensorflow as tf
import chakin
import json
import os
from collections import defaultdict
```

[2]:
```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

# import base packages into the namespace for this program
import pandas as pd
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
from tensorflow import keras
from sklearn.model_selection import train_test_split

RANDOM_SEED = 9999
# chakin.search(lang='English')  # lists available indices in English
```

0.0.3 Data and Model Exploration

[3]:
```
def run_model(filename, EVOCABSIZE, n_neurons, n_epochs):

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

    # Select the pre-defined embeddings source
    # Define vocabulary size for the language model
    # Create a word_to_embedding_dict
    embeddings_directory = 'embeddings/gloVe.6B'
    filename = filename ## argument
    embeddings_filename = os.path.join(embeddings_directory, filename)

    # Creates the Python defaultdict dictionary word_to_embedding_dict
    # for the requested pre-trained word embeddings
    def load_embedding_from_disks(embeddings_filename, with_indexes=True):
        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

    # Loading embeddings from embeddings_filename
    word_to_index, index_to_embedding = \
        load_embedding_from_disks(embeddings_filename, with_indexes=True)

    vocab_size, embedding_dim = index_to_embedding.shape

    # Define vocabulary size for the language model
    # To reduce the size of the vocabulary to the n most frequently used words
    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
    limited_word_to_index = defaultdict(default_factory, \
        {k: v for k, v in word_to_index.items() if v < EVOCABSIZE})

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

    # 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)

    # Delete large numpy array to clear some CPU RAM
    del index_to_embedding

    # 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)

    # 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:
        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

    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)

    # gather data for 500 negative movie reviews
    dir_name = 'run-jump-start-rnn-sentiment-big-v002/movie-reviews-negative'
    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

    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 = []

    for i in range(num_files):
        words = read_data(os.path.join(dir_name, filenames[i]))
        negative_documents.append(words)

    # gather data for 500 positive movie reviews
    dir_name = 'run-jump-start-rnn-sentiment-big-v002/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

    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 = []

    for i in range(num_files):
        words = read_data(os.path.join(dir_name, filenames[i]))
```

```
        positive_documents.append(words)

    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))

    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))

    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]

    # Show the seventh word in the tenth document
    test_word = documents[6][9]

    # Show the last word in the last document
    test_word = documents[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)
```

```python
# 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)

# 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)

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 = 100

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        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]
                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})

    # performance score table
    col_names=['Word Vector', 'Vocabulary Size', 'Neurons','Epochs',\
               'Training Set Accuracy', 'Test Set Accuracy']
    perf=pd.DataFrame([filename, EVOCABSIZE, n_neurons, n_epochs,\
                       np.round(acc_train, 2), np.round(acc_test, 2)],\
                      columns=[''], index=col_names).T
    return perf
```

0.0.4 Evaluation of Performance

[5]:
```
perf_1 = run_model(filename = 'glove.6B.50d.txt', EVOCABSIZE = 10000,
          n_neurons = 20, n_epochs = 50)
perf_1
```

[5]:
```
       Word Vector Vocabulary Size Neurons Epochs Training Set Accuracy  \
  glove.6B.50d.txt           10000      20     50                   0.8

 Test Set Accuracy
              0.68
```

[6]:
```
perf_2 = run_model(filename = 'glove.6B.100d.txt', EVOCABSIZE = 10000,
          n_neurons = 20, n_epochs = 50)
perf_2
```

[6]:
```
        Word Vector Vocabulary Size Neurons Epochs Training Set Accuracy  \
  glove.6B.100d.txt           10000      20     50                  0.92

 Test Set Accuracy
              0.66
```

[7]:
```
perf_3 = run_model(filename = 'glove.6B.50d.txt', EVOCABSIZE = 30000,
          n_neurons = 20, n_epochs = 50)
perf_3
```

[7]:
```
       Word Vector Vocabulary Size Neurons Epochs Training Set Accuracy  \
  glove.6B.50d.txt           30000      20     50                  0.82

 Test Set Accuracy
               0.7
```

[8]:
```
perf_4 = run_model(filename = 'glove.6B.100d.txt', EVOCABSIZE = 30000,
          n_neurons = 20, n_epochs = 50)
perf_4
```

[8]:
```
        Word Vector  Vocabulary Size  Neurons  Epochs  Training Set Accuracy  \
  glove.6B.100d.txt            30000       20      50                   0.95

  Test Set Accuracy
               0.66
```

[9]:
```
perf_5 = run_model(filename = 'glove.6B.50d.txt', EVOCABSIZE = 10000,
          n_neurons = 100, n_epochs = 50)
perf_5
```

[9]:
```
       Word Vector  Vocabulary Size  Neurons  Epochs  Training Set Accuracy  \
  glove.6B.50d.txt            10000      100      50                      1

  Test Set Accuracy
               0.54
```

[10]:
```
perf_6 = run_model(filename = 'glove.6B.50d.txt', EVOCABSIZE = 10000,
          n_neurons = 20, n_epochs = 100)
perf_6
```

[10]:
```
       Word Vector  Vocabulary Size  Neurons  Epochs  Training Set Accuracy  \
  glove.6B.50d.txt            10000       20     100                   0.88

  Test Set Accuracy
               0.69
```

[11]:
```
perf_7 = run_model(filename = 'glove.6B.50d.txt', EVOCABSIZE = 10000,
          n_neurons = 100, n_epochs = 100)
perf_7
```

[11]:
```
       Word Vector  Vocabulary Size  Neurons  Epochs  Training Set Accuracy  \
  glove.6B.50d.txt            10000      100     100                      1

  Test Set Accuracy
               0.56
```

0.0.5 Summary Table

[12]:
```
pd.concat([perf_1, perf_2, perf_3, perf_4, perf_5, perf_6, perf_7], axis=0)
```

[12]:
```
        Word Vector  Vocabulary Size  Neurons  Epochs  Training Set Accuracy  \
   glove.6B.50d.txt            10000       20      50                    0.8
  glove.6B.100d.txt            10000       20      50                   0.92
   glove.6B.50d.txt            30000       20      50                   0.82
  glove.6B.100d.txt            30000       20      50                   0.95
   glove.6B.50d.txt            10000      100      50                      1
   glove.6B.50d.txt            10000       20     100                   0.88
```

```
    glove.6B.50d.txt            10000      100     100                       1

  Test Set Accuracy
               0.68
               0.66
                0.7
               0.66
               0.54
               0.69
               0.56
```

0.0.6 Summary

In this assignment, we are asked to find the highest possible accuracy in movie review classification (negative vs. positive). The highest training accuracy was 100%, and testing accuracy was 70%. The recommendation is to use model 3 – the Recurrent Neural Network (RNN) model. When we increased the vocabulary size of model 3 from 1,000 to 3,000, the test set accuracy went up by 2%. When we increased the word vector glove.6B.50d to glove.6B.100, the training accuracy went up significantly, but testing accuracy went down due to the overfitting. Increased the epochs (number of iterations to train the model over the entire dataset) from 50 to 100 did not move the needle. When we increased the neurons from 20 to 100, the training accuracy went up to the highest, 100%, but the testing accuracy went down to the lowest, around 55% due to the overfitting. We can save time and money by not increasing the neurons and epochs from their default values. With this assignment, we were able to scratch the surface of NLP, text mining. We can tune other hyperparameters to improve the model further. We would recommend it for a high-level classification system for assessing the customer's sentiment of written product reviews or customer experience surveys. A system which can read sentiments and sort reviews into negative and positive would be most relevant to the customer services function. Model 3 which has lower word vector (glove.6B.50d), larger vocabulary size (3,000) with default neurons (20), and iteration epochs (50) needed to make an automated customer support system that is capable of identifying negative customer feelings. Once data scientists find the right hyperparameters to achieve the highest testing accuracy, they can implement automate the customer reviews, which can be more useful in a customer service function.