Assignment 8 – Language Modeling with an RNN

August 19, 2019

0.1 MSDS 422 - Andrew Kang

In this assignment, we will utilize the chakin python package to download word embeddings. The word embeddings are differentiated based on dimensions, source, and vocabulary size. Once we download the word embeddings, we will proceed to create a loop for model training. In the model loop, we will seek to create 4 models in a 2x2 experimental design that will test embedding dimension and vocabulary size combinations.

```
In [1]: import numpy as np
        import tensorflow as tf
        import chakin
        import json
        import os
        from collections import defaultdict
        import datetime
        from datetime import datetime
        import pandas as pd
```

```
In [2]: chakin.search(lang='English')  # lists available indices in English
```

```
                  Name  Dimension                    Corpus VocabularySize  \
2          fastText(en)        300                 Wikipedia           2.5M
11         GloVe.6B.50d         50  Wikipedia+Gigaword 5 (6B)           400K
12        GloVe.6B.100d        100  Wikipedia+Gigaword 5 (6B)           400K
13        GloVe.6B.200d        200  Wikipedia+Gigaword 5 (6B)           400K
14        GloVe.6B.300d        300  Wikipedia+Gigaword 5 (6B)           400K
15       GloVe.42B.300d        300          Common Crawl(42B)           1.9M
16      GloVe.840B.300d        300         Common Crawl(840B)           2.2M
17    GloVe.Twitter.25d         25               Twitter(27B)           1.2M
18    GloVe.Twitter.50d         50               Twitter(27B)           1.2M
19   GloVe.Twitter.100d        100               Twitter(27B)           1.2M
20   GloVe.Twitter.200d        200               Twitter(27B)           1.2M
21  word2vec.GoogleNews        300          Google News(100B)           3.0M

      Method Language    Author
2   fastText  English  Facebook
11     GloVe  English  Stanford
12     GloVe  English  Stanford
```

```
13     GloVe  English  Stanford
14     GloVe  English  Stanford
15     GloVe  English  Stanford
16     GloVe  English  Stanford
17     GloVe  English  Stanford
18     GloVe  English  Stanford
19     GloVe  English  Stanford
20     GloVe  English  Stanford
21  word2vec  English    Google
```

0.2 Download Files From Chakin

```
In [3]: 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)

In [4]: 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))

In [5]: 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.")

        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 downloaded.
Embeddings already extracted.
```

```
Run complete
```

0.3 Define Functions

```
In [6]: 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

In [7]: # 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

In [8]: 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 [9]: 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 [10]: 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 [11]:
```
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
```

In [12]:
```
codelist = ['\r', '\n', '\t']
# -----------------------------------------------
# gather data for 500 negative movie reviews
# -----------------------------------------------
dir_name = '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

negative_documents = []

#     print('\nProcessing document files under', dir_name)
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 = '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

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)

        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)

        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])))
```

```
In [13]: modelList = ['Model 1 - 10000 w/ 50d',
                      'Model 2 - 30000 w/ 50d',
                      'Model 3 - 10000 w/ 100d',
                      'Model 4 - 30000 w/ 100d']
         trainAccuracy = []
         testAccuracy = []
         procTime = []
```

0.4 Define Loop for Model Training and Testing

```
In [14]: def runFullModel(dynamic_vocab_size,embeddings_filename):
             print('\nLoading embeddings from', embeddings_filename)
             word_to_index, index_to_embedding = \
```

```
    load_embedding_from_disks(embeddings_filename, with_indexes=True)
print("Embedding loaded from disks.")

EVOCABSIZE = dynamic_vocab_size

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})

# Note: unknown words have representations with values [0, 0, ..., 0]
vocab_size, embedding_dim = index_to_embedding.shape

# 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)

# 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)

# -----------------------------------------------------
# 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)

# 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 = 10  # 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 = 50

    with tf.Session() as sess:
        init.run()
        start = datetime.now()
        for epoch in range(n_epochs):
#             print('---- Epoch ', epoch, ' ----')
            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('Epoch: ', epoch, 'Train accuracy:', acc_train, 'Test accuracy:',
    end = datetime.now()
    processing_time = end-start
    trainAccuracy.append(acc_train)
    testAccuracy.append(acc_test)
```

```
    procTime.append(processing_time)
    print("Train Accuracy: ",acc_train)
    print("Test Accuracy: ",acc_test)
    print("Processing Time: ", processing_time)
    return acc_train,acc_test,processing_time
```

0.5 GloVe.6B.50d

0.5.1 Model 1 - 10,000 Vocabulary Size

```
In [15]: embeddings_directory = 'embeddings/gloVe.6B'
         filename = 'glove.6B.50d.txt'
         embeddings_filename = os.path.join(embeddings_directory, filename)

In [16]: model_1 = runFullModel(10000,embeddings_filename)
```

```
Loading embeddings from embeddings/gloVe.6B\glove.6B.50d.txt
Embedding loaded from disks.
Train Accuracy:  0.8
Test Accuracy:  0.71
Processing Time:  0:00:02.933199
```

0.5.2 Model 2 - 30,000 Vocabulary Size

```
In [17]: model_2 = runFullModel(30000,embeddings_filename)
```

```
Loading embeddings from embeddings/gloVe.6B\glove.6B.50d.txt
Embedding loaded from disks.
Train Accuracy:  0.88
Test Accuracy:  0.68
Processing Time:  0:00:02.897786
```

0.6 GloVe.6B.100d

0.6.1 Model 3 - 10,000 Vocabulary Size

```
In [18]: embeddings_directory = 'embeddings/gloVe.6B'
         filename = 'glove.6B.100d.txt'
         embeddings_filename = os.path.join(embeddings_directory, filename)

In [19]: model_3 = runFullModel(10000,embeddings_filename)
```

```
Loading embeddings from embeddings/gloVe.6B\glove.6B.100d.txt
Embedding loaded from disks.
Train Accuracy:  0.78
```

```
Test Accuracy:  0.605
Processing Time:  0:00:03.955970
```

0.6.2 Model 4 - 30,000 Vocabulary Size

```
In [20]: model_4 = runFullModel(30000,embeddings_filename)
```

```
Loading embeddings from embeddings/gloVe.6B\glove.6B.100d.txt
Embedding loaded from disks.
Train Accuracy:  0.92
Test Accuracy:  0.645
Processing Time:  0:00:03.945530
```

0.7 Performance Review

```
In [21]: performance = pd.DataFrame({"Models":modelList})
```

```
In [22]: performance['Train Accuracy'] = trainAccuracy
         performance['Test Accuracy'] = testAccuracy
         performance['Processing Time'] = procTime
```

```
In [23]: performance_df = performance.sort_values(by='Test Accuracy',ascending=False)
```

```
In [24]: performance_df
```

```
Out[24]:                    Models  Train Accuracy  Test Accuracy Processing Time
         0   Model 1 - 10000 w/ 50d            0.80          0.710 00:00:02.933199
         1   Model 2 - 30000 w/ 50d            0.88          0.680 00:00:02.897786
         3  Model 4 - 30000 w/ 100d            0.92          0.645 00:00:03.945530
         2  Model 3 - 10000 w/ 100d            0.78          0.605 00:00:03.955970
```

0.8 Conclusion

In our 2x2 experimental design, we tested two different embedding dimensions as well as two different vocab sizes. What we found is that 50 embedding dimensions outperformed in our model relative to 100 embedding dimensions. For vocab size, we actually had mixed performance. For 50 embedding dimensions, we found that 10000 vocab size performed better. For 100 embedding dimensions, we found that 30000 vocab size performed better. Given our model parameters, 50 embedding dimensions and 10000 vocab size was the best performing model. One thing that may improve our model performance for higher embedding dimensions may be the number of neurons. For further analysis, we would want to test hyperparameters for neurons relative to embedding dimension size.

The performance of our individual models in our experimental design is not ideal in terms of accuracy. Our train accuracy and test accuracy deviated substantially, which suggests that there is overfitting occurring on the RNN. Consequently, even though Model 1 was the victor in our model competition, our recommendation to business management would be to expand our experimental design to include other embedding dimensions, vocabulary sizes, neurons, and other hypereparameters before selecting a final model.