HW 6

Background

In the class, you have seen that we introduced a number of approaches to process text and extract useful information from it. In this assignment, we will try to build deep learning models to do topic modeling – i.e., given a small amount of text, identify which subject area it comes from. This problem arises in the context of bio-medicine in many areas: for e.g., a clinician may want to know if a particular report is a radiology report, histopathology, report, or something else. Since we cannot access clinical reports (you'd need (1) lot of training that each of you have to undergo, and (2) there are privacy concerns associated with the data), we are going to prototype this problem by identifying a topic that is communicated by a scientific abstract.

An abstract is a collection of sentences that succinctly summarizes all of the results from a study/experiment. Hence, you can get an idea of what a paper is trying communicate often by just going over the abstract. Now, given an abstract, you can say which subject area that the results contribute to – often by scanning through the sentences. You are going to mimic this using deep learning approaches. For this HW, you will use a small set (about 80,000) abstracts from PubMed.

The maximum number of points for this assignment is 100. You have a week to submit from Sun (Nov 10, 2019) – which means your assignment is due on Nov 17, 2019 (11.59 PM Central Time).

What to expect?

You will be expected to set up your own Python environment and provide documentation on whether you were successful. This assignment can be run on your laptop. There is no need for a special Google Collab environment. Note that as part of your home work; much of this can be done with less than 150-200 lines of code.

What to hand in?

You are expected to hand in a python notebook (usually with the extension: `.ipynb`) or a Python script that generates all of the results for the questions below. Within your Python notebook, you can easily document your results, visualize plots, and also add comments using Markdown (`https://en.wikipedia.org/wiki/Markdown`). If you are not familiar with Markdown, you can easily learn it from the link above.

Building embedding models for text [30 points]

The data lives as a `csv` file in the HW6 folder on MLiM-Datasets folder: labeled_abstracts_reduced_80000.csv. Each line in the CSV file includes in it the subject area, that is separated from the abstract using a ';'. This is a cleaned version of the data, and we have about 80,000 abstracts in the CSV file. You will also see that there are 8 different subjects that we are grouping the abstracts. The number of abstracts you will see in the dataset are more or less equally divided – meaning you don't have to really think much about balancing the class distributions.

The first thing that we have to do is to build an embedding matrix for the text. There are many different ways to do this. But, to make it simple, we are going to use a simple model – namely `Word2Vec` that is prebuilt in the `gensim` library. Make sure in your Google Collab environment you have gensim installed. You can install gensim using the following command: `!pip install gensim`. The google Collab should automatically tell you that the gensim package is installed (usually).

Now, as shown in the code listing below, you can initialize a model that builds a Word2Vec model that we went over in class.

```
from gensim.models import Word2Vec

model = Word2Vec(size=embedding_size,
               window=8,
               min_count=2,
               workers=4,
               alpha=0.025,
               min_alpha=0.025)
model.build_vocab(abstracts) #where abstracts contains the content of the
    abstracts

# train on the abstracts
for epoch in range(10):
   model.train(abstracts,total_examples=model.corpus_count,epochs=1)
   model.alpha -= 0.002 #decrease the learning rate
   model.min_alpha = model.alpha #fix the learning rate, no decay
model.init_sims(replace=True)

# iterate through the learned vocabulary from the word2vec model
for key,val in model.wv.vocab.items():
   print(key) #this is the word
   print(model[key]) #this is the word2vec vector
```

Here we assume that abstracts is something that reads a single line from the data file and returns one instance (hint: use the `yield()` function and make sure the input is a list of word tokens). We are going to adopt the size of our embedding to be 512 for each word in the vocabulary. Your function should address two aspects here:

- Construct an embedding matrix that stores a length 300 vector for each word in the vocabulary based on the Word2Vec model. Save your embedding matrix (numpy matrix is recommended) so that you can use it subsequently. Be sure to also construct a dictionary that correctly maps each word to the appropriate row in your embedding matrix so we know what each row represents. Note that the 0th row in your embedding matrix should not be used for any words and be set to all zeros because 0 is reserved for padding! [10 points]

- Next, you are going to process each abstract such that each word is replaced by its corresponding row number in the embedding matrix. A common practice for text is to clip/pad all documents to the same length. For this exercise, all abstracts longer than 250 words should be clipped to 250 words, and all abstracts shorter than 250 words should be padded with 0's to be 250 words. Save these processed abstracts as numpy arrays – these will be the inputs to the deep learning classifier. Be sure not to lose the corresponding label associated with each abstract! [10 points]

If you get stuck, gensim provides plenty of documentation with examples on how to use word2vec: `https://radimrehurek.com/gensim/models/word2vec.html`.

Finally, it will be interesting to see if your embedding model really works. For this, we are going to extract the most common 500 words (this is to make sure you don't have to do it on a large vocabulary) and use t-stochastic neighborhood embedding (t-SNE) to visualize this in 2D. Obviously we can't visualize a 300 dimensional space, so we will train the TSNE model to accept the embeddings and plot a 2D plot to show us how the words are organized. [10 points]

```
counts = collections.Counter(allwords).most_common(500)
#where allwords contains the content from all abstracts

embeddings = np.empty((500,embedding_size))
for i in range(500):
   embeddings[i,:] = model[counts[i][0]]
tsne = TSNE(perplexity=30, n_components=2, init='pca', n_iter=7500)
embeddings = tsne.fit_transform(embeddings)

#plot embeddings
print("plotting most common words")
fig, ax = plt.subplots(figsize=(30, 30))
for i in range(500):
   ax.scatter(embeddings[i,0],embeddings[i,1])
   ax.annotate(counts[i][0], (embeddings[i,0],embeddings[i,1]))
plt.show()
```

Building a simple CNN to classify abstracts [35 points]

Now that you have your data dictionary and you can easily construct the word embeddings, we can build some deep learning models. First, we'll need to split the data into train and test sets. Use the `sklearn.model_selection.train_test_split` function from the Python scikitlearn package to split 80% of the abstracts (and corresponding labels) into a train set and the rest into your test set.

Now, let's build a simple convolutional neural network (CNN). Your CNN should have the following:

- An input layer which takes in a batch of documents [n x 250], where each word in a document is represented by its corresponding row (integer) in the embedding matrix
- An embedding lookup layer that replaces each word index with its corresponding word embedding vector in your embedding matrix [n x 250 x 300]
- Three parallel 1D convolution layers:

  A 1D convolution with a window size of 3 words and 100 filters [n x 250 x 100]

  A 1D convolution with a window size of 4 words and 100 filters [n x 250 x 100]

  A 1D convolution with a window size of 5 words and 100 filters [n x 250 x 100]
- A layer that concatenates the output of the three parallel 1D convolution layers [n x 250 x 300]
- A temporal maxpool layer that, for each filter, selects the max value across all words for that filter [n x 300]
- A softmax layer that outputs the class probabilities for each document [n x 8]
- For training, use cross entropy loss with the Adam optimizer

Your goal for this part of the assignment is to evaluate the training and testing of the model. [30 points]

Comment on what you think happens with the learning. Let's say, we remove one of the layers in the 1D convolution (say, for e.g., we remove a window size of 5), how does it affect the overall training? [5 points]

A simple RNN to classify abstracts [35 points]

Based on what we did with the CNN, use the same embedding layers, but instead of the three parallel 1D convolution layers, replace them with a single bi-directional LSTM (bi-LSTM) that spans an entire sentence. Feed the Bi-LSTM outputs to a maxpool layer that selects the maximum value across all the words. Finally, connect it to a softmax layer that outputs the class probabilities for each document. For training, use cross-entropy with the Adam optimizer. Your goal for this part of the HW is to evaluate the training/ testing of the RNN model. For this part of the assignment, there is much more flexibility in how you build your RNN classifier – you can follow a canned tutorial example or try your own. The main idea is that we want you to think a bit about how you will design a network – while deriving from an existing one. Many applications are usually designed in this manner. [20 points]

How long does it take for your code to run a single epoch of training using the CNN vs. the RNN model? You can do this by reporting the time on the keras prompt for each epoch. [5 points]

What are the number of epochs required to train each of these networks to achieve a convergence? Can you comment on the differences between the training for RNN and the CNN? [5 points]

Can you comment on whether a simple classifier such as an SVM (or any other classifier of your choice) works equally well with respect to the two classifiers you built? [5 points]