PA 5: Twitter Sentiment Analysis with Recurrent Neural Networks

ECSE 4965/6965
Matt Klawonn, klawom@rpi.edu

Due: April 20th, 2017, 11:59 PM EDT

1 Overview

In this assignment, you will train a recurrent neural network for the task of sentiment analysis on natural language data. More specifically, we will analyze data from Twitter and learn to classify it as either ”positive” or ”negative”. For those who don’t know, Twitter is an online news and social networking site based on communication between users via ”tweets,” which are messages in natural language limited to 140 characters. These tweets can be viewed as sequences of words in natural language, and will form the sequential input to our RNN model. The following guide will take you through the downloading a preprocessed version of a sentiment dataset and creating a model in TensorFlow.

1.1 Sentiment Analysis

Sentiment analysis refers to the natural language processing task of classifying some collection of text by its polarity, i.e whether or not the text has a ”positive”, ”negative”, or ”neutral” attitude. The goal is to understand the attitude of the agent that generated the text. It can also attempt to assign a collection of text to more refined emotional states, such as ”angry”, ”happy”, or ”sad”. In our case we will be looking only at positive and negative polarity and classifying at the tweet level.

2 Dataset

Our dataset can be downloaded here. If you are interested in the dataset from which this originates, see section 2.1. Please read section 2.2 to get an idea for the techniques used in preprocessing and to understand the dataset in its current form. There is also a question in section 2.2 which will need to be answered in the writeup. Please also download the associated vocabulary json here. You can also download the ”reverse vocabulary” json here. You need only download the reverse vocabulary if you want to ”decode” your tweets (go from an integer representation back to a string representation). More on this later.

2.1 Original Dataset

Our dataset comes from the Sentiment140 training set. This data contains 1.6 million tweets, classified as either positive or negative, that have been harvested by searching Twitter for emoticons. Tweets containing happy emoticons were naively labeled as positive, and those containing sad emoticons were labeled as negative. The emoticons were then removed from the tweets so that classifiers must rely on natural language features for classification. In addition to a column containing the text of the tweet, the original dataset also contains a column indicating the polarity of the tweets (0 for negative, 4 for positive), the time of the tweet, the associated Twitter users, a timestamp, and a flag indicating the query type used to get the data. We will not use any of these additional columns in our assignment except the labels. The original dataset is available for download here, but **DO NOT USE THIS DATASET FOR THIS ASSIGNMENT.** Instead we will be working with a pruned and preprocessed version of the dataset.

2.2 Preprocessing

For this assignment the data preprocessing has been handled for you. There are a number of potential problems with handling the raw CSV file containing tweets in string form. First let's take a look at the trivial preprocessing that was done.

- Unnecessary columns were removed, leaving only tweets and associated labels.
- Labels were converted from 0 and 4 to 0 and 1 in order to work more nicely with our binary classification setup. The reason that 0 and 4 were initially present in the data is the task was originally multiclass classification. Tweets were rated on a scale from 0 to 4, with 0 being most negative and 4 being most positive.

- Urls and links were removed.
- Twitter handles were removed from tweets. When constructing tweets, users frequently start by addressing another user by their username, also known as a handle. We do not need to know who a user was addressing to classify a tweet.
- Empty tweets were deleted

A number of other proprocessing steps were taken in order to use a RNN on this data, some of which may be less intuitive. These steps are listed below, and might give you some insight into considerations that are made when undertaking a natural language processing task.

- Tokenization: In order to turn a string into a sequence of words, tokenization must be performed. Tokenization is the process of splitting natural language text into its discrete parts. For this assignment, each tweet was split on whitespace, which is a very simple approach. In more advanced applications, sophisticated tokenizers like those provided by Python's NLTK can be used. To see more information about NLTK and tokenization, see here.
- Removing Punctuation: Because of our naive tokenization process, many generated tokens(words) will contain trailing commas, question marks, etc. We can make an assumption that punctuation will not be helpful in determining the sentiment of a tweet, since both positive and negative tweets ought to use them equally. Note that this may not be true in practice, but is true enough for our purposes. Therefore, we can remove all punctuation and be left with words alone. If we wanted to use punctuation, a more sophisticated tokenization method would be necessary.
- Lowercasing Words: All words were converted to lowercase. This was done to ensure that the RNN treats a word the same no matter what case a user used when typing it. Were words left in their original cases, many copies of the same word might appear in the vocabulary, e.g Sad, sad, SAD, etc. This is probably undesirable, though different cases of the same word may indicate different meanings. However, as we will see, limiting vocabulary size has more computational benefits than drawbacks.

- Removing "uncommon" words: We remove words which appear less than 100 times in order to limit the amount of training data for this exercise, and also to control the vocabulary size. Many words which appear infrequently are typos or potentially just gibberish, and cost more to keep track of than they are worth in terms of improvements in accuracy. One way to include some information from uncommon words and still control vocabulary size is to use a special "UNKNOWN" token to represent all words not in the vocabulary. We do not do this in this assignment. Rather, we simply remove tweets from the dataset which have words outside our vocabulary.

- Strings to Ints: Now that all words have been normalized and we have settled on our vocabulary, each word can be represented by a unique integer. We will feed our RNN sequences of integers which represent words. This is necessary in order for the RNN to turn each word into an input vector, a process known as word embedding. The map from word to integer can be downloaded here, though this will be unnecessary unless you wish to re-use this trained model on your own tweets. However, the map from integer to word will be necessary and can be downloaded here.

- Removing long tweets: In order to reduce the computational burden on you, we have removed tweets that are above 25 tokens long. This helps reduce the number of operations performed and memory use. Further, classification and other learning tasks become harder as inputs to an RNN become longer, thanks to the exploding and vanishing gradient problems.

- Fixing sequence length and creating masks: In order for the RNN to accept our sequential inputs, we need to have a fixed number of tokens per sequence. Without knowing the number of inputs, we couldn't declare the shape of the input tensor. However, tweets are clearly of varying length. In order to overcome this apparent conundrum, we determine the maximum length of a tweet in our dataset. This is 25 for our particular assignment, thanks the the previous preprocessing step. If a particular sequence is shorter than this maximum length, then we simply append dummy words to this sequence until it is the maximum length. At the same time, we construct a 25 entry long "mask" for this sequence indicating which values are real and which values are dummy values to be ignored. At each index in this mask, a value of 1 indicates a "real" entry, while a value of 0 indicates a dummy entry. We will see how to apply this mask later.

2.3 A Word on Word Embeddings

We briefly touched on the process of word embedding, or converting a single word into an input vector for the RNN to work with. This embedding, or word-feature-vector, will be learned, similar to how the filters of a CNN are learned. Each word's corresponding embedding is stored in a matrix of shape (vocabulary_size, embedding_size), where embedding_size determines the length of the word-feature-vector. Without our previous preprocessing steps to limit the vocabulary, you could end up with a vocabulary_size of up to 500000. In your assignment write-up, please include the calculations for how much memory a vocabulary size of 500000 would cost given an embedding size of 300 and using double precision floating point numbers. Give your answer in megabytes. You will see why modeling this large a vocabulary is a problem, as the embedding matrix is a single layer in our recurrent network.

2.4 Our Dataset

Our code applies all of the above preprocessing techniques to the original data, and saves the resulting 500000 tweets into a training, validation, and testing dataset. The training data contains 400000 tweets of fixed length and includes masks for each tweet. The validation data contains 50000 tweets also with masks. We reserve a test set of 50000 tweets for evaluating your model. After preprocessing, our vocabulary size is 8745. The training and evaluation datasets are compressed into one npz file which can be downloaded here. In order to unpack the npz file, use the following code (Figure 1), which should work for both python 2 and python 3.

Next we will construct our model and see how to feed it our data.

3 Model

We will construct an RNN for this problem. Recall that an RNN accepts a sequence of inputs and produces a sequence of outputs as in Figure 2. Formally, our task is a binary classification task over the entire input sequence, so we need only consider the output of the RNN after it has received the final "real" word (not the dummy words). Note that this will be a function of all "real" input values in the sequence.

```
import numpy as np
npzfile = np.load("train_and_val.npz")
train_x = npzfile["train_x"]
train_y = npzfile["train_y"]
train_mask = npzfile["train_mask"]
#Validation filenames follow the same pattern
val_x = npzfile["val_x"]
# etc ...
```

Figure 1: Python code indicating how to read in the dataset for this assignment

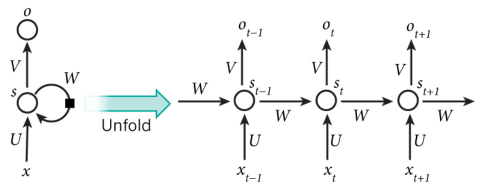

Figure 2: An example RNN producing output at multiple timesteps.

3.1 Model Design

The general architecture will be as outlined in Figure 3. Our sequential input tensor of shape (batch_size, max_sequence_length) will be fed into an embedding matrix (see section 3.2) of shape (vocabulary_size, word_embedding_size). The choice of word embedding size is left up to you. With the word embedding done, the sequence of word embedding vectors is fed into the recurrent neural network cell. This cell can be a vanilla RNN, an LSTM, a Gated Recurrent Unit (GRU), or whatever recurrent object you want. The output of this RNN cell will be determined by the mask (see section 3.4), and will be multiplied by an output matrix to produce a single logit. You will then use the cross entropy loss to update your network parameters. I recommend using the tf.nn.sigmoid_cross_entropy_with_logits() function. Recall you must use this loss function on the unactivated output of that single final network node, i.e without applying a sigmoid first.

In terms of performance, shoot for 84-85% validation accuracy. You should be able to reach 80% quickly, i.e with one pass over all of the data. Please track your loss, training accuracy, and validation accuracy, and plot them. An example is shown in Figure 3. However, please generate a data point after every n iterations as opposed to every epoch. 85% accuracy is very good for sentiment analysis tasks, though it is difficult to compare the results here to other results since we so carefully curate our data.

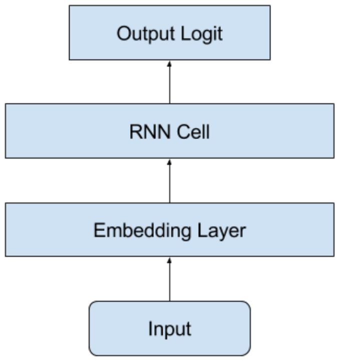

Figure 3: Our architecture.

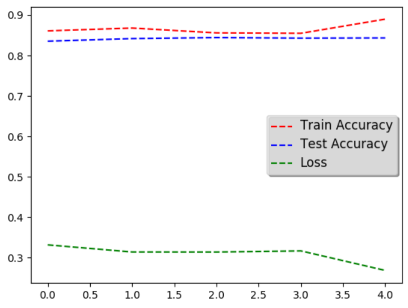

Figure 4: Example loss plot. The losses were examined after each epoch. Please provide a more detailed plot by examining after every n iterations, where the choice of n is up to you.

3.2 Word Embeddings

In order to construct your word embedding layer, you must first declare a tensor variable of shape (vocab_size, word_embedding_size). Exactly how to initialize this embedding matrix is up to you, simply keep in mind principles of initialization that have been previously discussed in class. The output of a word embedding layer should pass onto the next layer (the RNN cell) without going through an activation function. This is because the word embedding can be considered an input to the network itself rather than a layer to be activated, since each embedding is supposed to be a representation of its corresponding word.

TensorFlow has an efficient built in function for generating an embedding of a particular word. Given a word embedding matrix w_embed, simply run

```
rnn_input = tf.nn.embedding_lookup(w_embed, sequence_placeholder)
```

As the name suggests, rnn_input, a tensor of shape (batch_size, max_sequence_length, word_embedding_size), will then be fed into the RNN cell.

3.3 Layer Choice

Tensorflow provides access to a number of recurrent layers to choose from. Probably the most popular is the LSTM cell, which can be constructed in one line in tensorflow. See the tf.contrib.rnn.LSTMCell() function for details. This cell will accept the output from the embedding layer. Other popular recurrent cell choices can be found here, under the section "Core RNN Cells for use with TensorFlow's core RNN methods." Each of these cells can be used with the tf.nn.dynamic_rnn() operation, which will handle the feeding of the input sequence (the word embedding one) to your choice of RNN cell and generate outputs at each timestep. You can also use dropout with these cells by applying the tf.contrib.rnn.DropoutWrapper() to the cell.

More precisely, the dynamic rnn function generates an output, state pair, though we will only consider the output portion. Remember that the rnn generates an output for each input in the sequence, even the dummy ones. The output of the dynamic rnn operation will be a tensor of shape (batch_size, max_len, cell_size). We only want the last relevant output, i.e no outputs that have been generated using dummy values. This is where the masking stage come in.

3.4 Masking

In order to select the last relevant output of the dynamic rnn cell, we take the output tensor of shape (batch_size, max_len, cell_size) and perform the following series of operations on it (not necessarily the batch size and max length computations if you have access to them somewhere else already).

```
length = tf.cast(tf.reduce_sum(mask_placeholder,
    reduction_indices=1), tf.int32)
batch_size = tf.shape(output)[0]
max_length = tf.shape(output)[1]
out_size = int(output.get_shape()[2])
flat = tf.reshape(output, [-1, out_size])
index = tf.range(0, batch_size) * max_length + (length - 1)
relevant = tf.gather(flat, index)
```

Here the true length of the sequence is computed by summing the number of ones in the mask data. Once this length has been determined, the output tensor is flattened to shape (total_inputs, out_size). An index into this tensor is constructed, first generating the start indices for each training example tf.range(0, batch_size) * max_length, and then adding the individual sequence lengths to those start indices. This link has more coverage of masking in TensorFlow.

Now that you have your relevant output tensor of shape (batch_size, rnn_cell_size), you can construct an output weight tensor of shape (rnn_cell_size, 1), along with its corresponding bias. This produces your single output logit (per batch example) which you will then feed to your loss.

3.5 Hyperparameters and Optimization

The choices of initializations, regularization, optimizer, and other hyperparameters are left completely up to you. Recall previous principles we have discussed in the class and you should be fine. One tip: adaptive learning rate optimizers tend to work better with RNNs than plain SGD.

3.6 Computational Time

On a Lenovo t440s with 8GB of RAM, using a batch size of 1000, one iteration takes about 3.5 seconds. 10 epochs of training would then take about 3.5 hours. You may find, however, that fewer than 10 epochs are necessary to reach the target validation accuracy. I reached target accuracy in 3 epochs before I began overfitting. If you run into memory problems, please let us know. I don't think this should be a problem unless you have under 4GB of RAM.

3.7 Saving

In order to run your model for grading, we will need access to your prediction operation, sequence placeholder, and mask placeholder. Your prediction operation should produce a row vector of predicted classifications for a collection of input sequences. I suggest using tf.round on the output sigmoid of your model. Please add these to a collection titled "validation_nodes" and save using the model name "my_model." One example of doing so is pictured in Figure 3.

```
tf.get_collection("validation_nodes")
tf.add_to_collection("validation_nodes", sequence_placeholder)
tf.add_to_collection("validation_nodes", mask_placeholder)
tf.add_to_collection("validation_nodes", predict_op)
saver = tf.train.Saver()
save_path = saver.save(sess, "my_model")
```

Figure 5: Python code indicating how to read in the dataset for this assignment

3.8 Word Vector Visualization

We have briefly touched on word embeddings and the purpose of learning them from discrete inputs. Once you have trained your recurrent network on the given data, your word embedding matrix will no longer be a random matrix, but rather a matrix in which each row contains a representation of the corresponding word in a vector space. The learned representation can give a sense of the meanings the model has learned to attribute to each word. In good embeddings, words with similar meanings will often have vectors close to one another in the embedding space. Such relationships can be visualized using dimensionality reduction, as we will do here.

For this assignment, we will use the t-distributed Stochastic Neighbor Embedding algorithm to visualize the our embeddings t-SNE for short. t-SNE . As described in Wikipedia t-SNE "is a nonlinear dimensionality reduction technique that is particularly well-suited for embedding high-dimensional data into a space of two or three dimensions, which can then be visualized in a scatter plot. Specifically, it models each high-dimensional object by a two- or three-dimensional point in such a way that similar objects are modeled by nearby points and dissimilar objects are modeled by distant points." An introduction for interested readers can be viewed here.

For our purposes, t-SNE will simply be a nice way of visualizing our learned word embedding vectors. Please install the sklearn package e.g via pip install sklearn. This package contains an implementation of t-SNE that we will use.

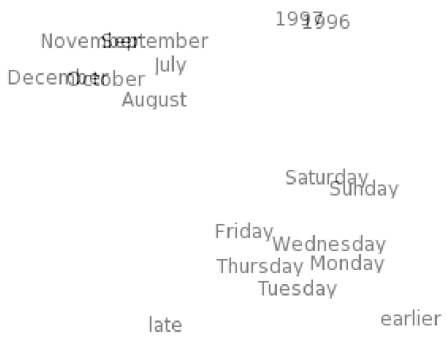

Figure 6: An example t-SNE visualization of some learned word embeddings from Turian et al. 2010. Notice how words with similar functions, the days, months, and years, are grouped together.

We are going to visualize a simple set of words that should give us pretty clear groupings: fruits and days of the week. We will be using the vocab.json file we downloaded earlier.

```
import json
with open("vocab.json", "r") as f:
  vocab = json.load(f)
s = ["monday", "tuesday", "wednesday", "thursday", "friday",
    "saturday", "sunday", "orange", "apple", "banana", "mango",
    "pineapple", "cherry", "fruit"]
words = [(i, vocab[i]) for i in s]
```

Figure 7: Python code indicating how to generate common words in our vocabulary, along with their indices.

With the top 100 words generated, we can now select the corresponding vectors from the embedding matrix and embed them in our two dimensional space for visualizing.

```
from sklearn.manifold import TSNE
model = TSNE(n_components=2, random_state=0)
#Note that the following line might use a good chunk of RAM
tsne_embedding = model.fit_transform(word_embedding_matrix)
words_vectors = tsne_embedding[np.array([item[1][0] for item in
    words])]
```

Figure 8: Python code indicating how to train a t-SNE embedding on our common words

Now that you have an array containing all our example words embedded in a two dimensional space, you can create a scatter plot of these points like Figure 5 and 9. See, for example, this stackoverflow question showing how to annotate a scatter plot with text. The tsne_embedding array preserves the order of points, so your words_vectors word order will match that of the tsne_embedding array. You should be able to get a fairly clean grouping in this collection of words. See Figure 9 for an example.

4 Submission

To recap, you must submit for this assignment

- Your memory calculation (see Section 2.3).
- Your graph collection containing the input and prediction nodes (see Section 3.7).
- Your loss and accuracy plots (see Section 3.1).
- Your visualization of word vectors (see Section 3.8).

You can validate that your graph is OK by using the script at a link to be uploaded soon. If you saved your collection as "my_model", simply run python validate.py my_model.

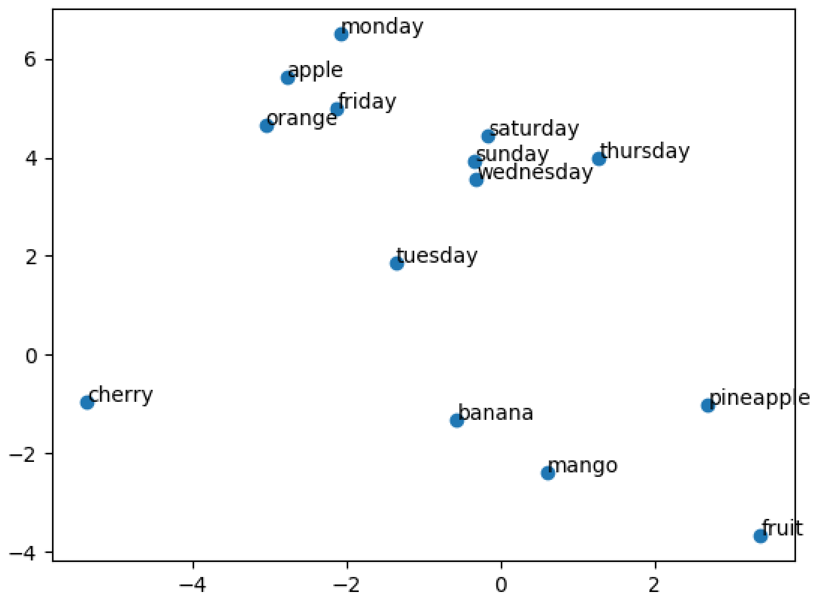

Figure 9: An example t-SNE visualization of the word list given.