Sentiment analysis with TFLearn

In this notebook, we'll continue Andrew Trask's work by building a network for sentiment analysis on the movie review data. Instead of a network written with Numpy, we'll be using TFLearn, a high-level library built on top of TensorFlow. TFLearn makes it simpler to build networks just by defining the layers. It takes care of most of the details for you.

We'll start off by importing all the modules we'll need, then load and prepare the data.

In [1]:

```
import pandas as pd
import numpy as np
import tensorflow as tf
import tflearn
from tflearn.data_utils import to_categorical
```

Preparing the data

Following along with Andrew, our goal here is to convert our reviews into word vectors. The word vectors will have elements representing words in the total vocabulary. If the second position represents the word 'the', for each review we'll count up the number of times 'the' appears in the text and set the second position to that count. I'll show you examples as we build the input data from the reviews data. Check out Andrew's notebook and video for more about this.

Read the data

Use the pandas library to read the reviews and postive/negative labels from comma-separated files. The data we're using has already been preprocessed a bit and we know it uses only lower case characters. If we were working from raw data, where we didn't know it was all lower case, we would want to add a step here to convert it. That's so we treat different variations of the same word, like `The`, `the`, and `THE`, all the same way.

In [2]:

```
reviews = pd.read_csv('reviews.txt', header=None)
labels = pd.read_csv('labels.txt', header=None)
```

Counting word frequency

To start off we'll need to count how often each word appears in the data. We'll use this count to create a vocabulary we'll use to encode the review data. This resulting count is known as a bag of words. We'll use it to select our vocabulary and build the word vectors. You should have seen how to do this in Andrew's lesson. Try to implement it here using the Counter class.

Exercise: Create the bag of words from the reviews data and assign it to `total_counts`. The reviews are stores in the `reviews` Pandas DataFrame. If you want the reviews as a Numpy array, use `reviews.values`. You can iterate through the rows in the DataFrame with `for idx, row in reviews.iterrows()`: (documentation). When you break up the reviews into words, use `.split(' ')` instead of `.split()` so your results match ours.

In [3]:

```
from collections import Counter
total_counts = Counter()
for _, row in reviews.iterrows():
    total_counts.update(row[0].split(' '))
print("Total words in data set: ", len(total_counts))
```

```
Total words in data set:  74074
```

Let's keep the first 10000 most frequent words. As Andrew noted, most of the words in the vocabulary are rarely used so they will have little effect on our predictions. Below, we'll sort `vocab` by the count value and keep the 10000 most frequent words.

In [4]:

```
vocab = sorted(total_counts, key=total_counts.get, reverse=True)[:10000]
print(vocab[:60])
```

```
['', 'the', '.', 'and', 'a', 'of', 'to', 'is', 'br', 'it', 'in', 'i', 'this',
'that', 's', 'was', 'as', 'for', 'with', 'movie', 'but', 'film', 'you', 'on',
't', 'not', 'he', 'are', 'his', 'have', 'be', 'one', 'all', 'at', 'they', 'by',
'an', 'who', 'so', 'from', 'like', 'there', 'her', 'or', 'just', 'about', 'out',
'if', 'has', 'what', 'some', 'good', 'can', 'more', 'she', 'when', 'very', 'up',
'time', 'no']
```

What's the last word in our vocabulary? We can use this to judge if 10000 is too few. If the last word is pretty common, we probably need to keep more words.

In [5]:

```
print(vocab[-1], ': ', total_counts[vocab[-1]])
```

```
float :  30
```

The last word in our vocabulary shows up in 30 reviews out of 25000. I think it's fair to say this is a tiny proportion of reviews. We are probably fine with this number of words.

Note: When you run, you may see a different word from the one shown above, but it will also have the value `30`. That's because there are many words tied for that number of counts, and the `Counter` class does not guarantee which one will be returned in the case of a tie.

Now for each review in the data, we'll make a word vector. First we need to make a mapping of word to index, pretty easy to do with a dictionary comprehension.
Exercise: Create a dictionary called `word2idx` that maps each word in the vocabulary to an index. The first word in `vocab` has index `0`, the second word has index `1`, and so on.

In [6]:

```
word2idx = {word: i for i, word in enumerate(vocab)}
```

Text to vector function

Now we can write a function that converts a some text to a word vector. The function will take a string of words as input and return a vector with the words counted up. Here's the general algorithm to do this:

- Initialize the word vector with np.zeros, it should be the length of the vocabulary.
- Split the input string of text into a list of words with `.split(' ')`. Again, if you call `.split()` instead, you'll get slightly different results than what we show here.
- For each word in that list, increment the element in the index associated with that word, which you get from `word2idx`.

Note: Since all words aren't in the `vocab` dictionary, you'll get a key error if you run into one of those words. You can use the `.get` method of the `word2idx` dictionary to specify a default returned value when you make a key error. For example, `word2idx.get(word, None)` returns `None` if `word` doesn't exist in the dictionary.

In [7]:

```
def text_to_vector(text):
    word_vector = np.zeros(len(vocab), dtype=np.int_)
    for word in text.split(' '):
        idx = word2idx.get(word, None)
        if idx is None:
            continue
        else:
            word_vector[idx] += 1
    return np.array(word_vector)
```

In [8]:
```
text_to_vector('The tea is for a party to celebrate '
               'the movie so she has no time for a cake')[:65]
```

Out[8]:
```
array([0, 1, 0, 0, 2, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 1, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0])
```

Now, run through our entire review data set and convert each review to a word vector.

In [9]:
```
word_vectors = np.zeros((len(reviews), len(vocab)), dtype=np.int_)
for ii, (_, text) in enumerate(reviews.iterrows()):
    word_vectors[ii] = text_to_vector(text[0])
```

In [10]:
```
# Printing out the first 5 word vectors
word_vectors[:5, :23]
```

Out[10]:
```
array([[ 18,   9,  27,   1,   4,   4,   6,   4,   0,   2,   2,   5,   0,
          4,   1,   0,   2,   0,   0,   0,   0,   0,   0],
       [  5,   4,   8,   1,   7,   3,   1,   2,   0,   4,   0,   0,   0,
          1,   2,   0,   0,   1,   3,   0,   0,   0,   1],
       [ 78,  24,  12,   4,  17,   5,  20,   2,   8,   8,   2,   1,   1,
          2,   8,   0,   5,   5,   4,   0,   2,   1,   4],
       [167,  53,  23,   0,  22,  23,  13,  14,   8,  10,   8,  12,   9,
          4,  11,   2,  11,   5,  11,   0,   5,   3,   0],
       [ 19,  10,  11,   4,   6,   2,   2,   5,   0,   1,   2,   3,   1,
          0,   0,   0,   3,   1,   0,   1,   0,   0,   0]])
```

Train, Validation, Test sets

Now that we have the word_vectors, we're ready to split our data into train, validation, and test sets. Remember that we train on the train data, use the validation data to set the hyperparameters, and at the very end measure the network performance on the test data. Here we're using the function `to_categorical` from TFLearn to reshape the target data so that we'll have two output units and can classify with a softmax activation function. We actually won't be creating the validation set here, TFLearn will do that for us later.

In [11]:
```
Y = (labels=='positive').astype(np.int_)
records = len(labels)

shuffle = np.arange(records)
np.random.shuffle(shuffle)
test_fraction = 0.9

train_split, test_split = shuffle[:int(records*test_fraction)],
shuffle[int(records*test_fraction):]
trainX, trainY = word_vectors[train_split,:],
to_categorical(Y.values[train_split], 2)
testX, testY = word_vectors[test_split,:], to_categorical(Y.values[test_split],
2)
```

In [12]:
```
trainY
```

Out[12]:

```
array([[ 0.,  1.],
       [ 1.,  0.],
       [ 1.,  0.],
       ...,
       [ 1.,  0.],
       [ 1.,  0.],
       [ 0.,  1.]])
```

Building the network

TFLearn lets you build the network by defining the layers.

Input layer

For the input layer, you just need to tell it how many units you have. For example,

```
net = tflearn.input_data([None, 100])
```

would create a network with 100 input units. The first element in the list, `None` in this case, sets the batch size. Setting it to `None`here leaves it at the default batch size.

The number of inputs to your network needs to match the size of your data. For this example, we're using 10000 element long vectors to encode our input data, so we need 10000 input units.

Adding layers

To add new hidden layers, you use

```
net = tflearn.fully_connected(net, n_units, activation='ReLU')
```

This adds a fully connected layer where every unit in the previous layer is connected to every unit in this layer. The first argument `net` is the network you created in the `tflearn.input_data` call. It's telling the network to use the output of the previous layer as the input to this layer. You can set the number of units in the layer with `n_units`, and set the activation function with the `activation` keyword. You can keep adding layers to your network by repeated calling `net = tflearn.fully_connected(net, n_units)`.

Output layer

The last layer you add is used as the output layer. There for, you need to set the number of units to match the target data. In this case we are predicting two classes, positive or negative sentiment. You also need to set the activation function so it's appropriate for your model. Again, we're trying to predict if some input data belongs to one of two classes, so we should use softmax.

```
net = tflearn.fully_connected(net, 2, activation='softmax')
```

Training

To set how you train the network, use

```
net = tflearn.regression(net, optimizer='sgd', learning_rate=0.1,
loss='categorical_crossentropy')
```

Again, this is passing in the network you've been building. The keywords:

- `optimizer` sets the training method, here stochastic gradient descent
- `learning_rate` is the learning rate
- `loss` determines how the network error is calculated. In this example, with the categorical cross-entropy.

Finally you put all this together to create the model with `tflearn.DNN(net)`. So it ends up looking something like

```
net = tflearn.input_data([None, 10])                          # Input

net = tflearn.fully connected(net, 5, activation='ReLU')      # Hidden

net = tflearn.fully_connected(net, 2, activation='softmax')   # Output

net = tflearn.regression(net, optimizer='sgd', learning_rate=0.1,
loss='categorical_crossentropy')

model = tflearn.DNN(net)
```

Exercise: Below in the `build_model()` function, you'll put together the network using TFLearn. You get to choose how many layers to use, how many hidden units, etc.

In [13]:

```
# Network building
def build_model():
    # This resets all parameters and variables, leave this here
    tf.reset_default_graph()

    # Inputs
    net = tflearn.input_data([None, 10000])

    # Hidden layer(s)
    net = tflearn.fully_connected(net, 200, activation='ReLU')
    net = tflearn.fully_connected(net, 25, activation='ReLU')

    # Output layer
    net = tflearn.fully_connected(net, 2, activation='softmax')
    net = tflearn.regression(net, optimizer='sgd',
                             learning_rate=0.1,
                             loss='categorical_crossentropy')

    model = tflearn.DNN(net)
    return model
```

Intializing the model

Next we need to call the `build_model()` function to actually build the model. In my solution I haven't included any arguments to the function, but you can add arguments so you can change parameters in the model if you want.
Note: You might get a bunch of warnings here. TFLearn uses a lot of deprecated code in TensorFlow. Hopefully it gets updated to the new TensorFlow version soon.

In [14]:

```
model = build_model()
```

Training the network

Now that we've constructed the network, saved as the variable `model`, we can fit it to the data. Here we use the `model.fit`method. You pass in the training features `trainX` and the training targets `trainY`. Below I set `validation_set=0.1` which reserves 10% of the data set as the validation set. You can also set the batch size and number of epochs with the `batch_size`and `n_epoch` keywords, respectively. Below is the code to fit our the network to our word vectors.

You can rerun `model.fit` to train the network further if you think you can increase the validation accuracy. Remember, all hyperparameter adjustments must be done using the validation set. **Only use the test set after you're completely done training the network.**

In [15]:

```
# Training
model.fit(trainX, trainY, validation_set=0.1, show_metric=True, batch_size=128, n_epoch=100)
```

```
Training Step: 15900  | total loss: 0.28905
| SGD | epoch: 100 | loss: 0.28905 - acc: 0.8768 | val_loss: 0.45618 - val_acc:
0.8351 -- iter: 20250/20250
Training Step: 15900  | total loss: 0.28905
| SGD | epoch: 100 | loss: 0.28905 - acc: 0.8768 | val_loss: 0.45618 - val_acc:
0.8351 -- iter: 20250/20250
--
```

Testing

After you're satisified with your hyperparameters, you can run the network on the test set to measure it's performance. Remember, *only do this after finalizing the hyperparameters*.

In [16]:

```
predictions = (np.array(model.predict(testX))[:,0] >= 0.5).astype(np.int_)
test_accuracy = np.mean(predictions == testY[:,0], axis=0)
print("Test accuracy: ", test_accuracy)
```

```
Test accuracy:  0.8504
```

Try out your own text!

In [17]:

```
# Helper function that uses your model to predict sentiment
def test_sentence(sentence):
    positive_prob = model.predict([text_to_vector(sentence.lower())])[0][1]
    print('Sentence: {}'.format(sentence))
    print('P(positive) = {:.3f} :'.format(positive_prob),
          'Positive' if positive_prob > 0.5 else 'Negative')
```

In [18]:

```
sentence = "Moonlight is by far the best movie of 2016."
test_sentence(sentence)

sentence = "It's amazing anyone could be talented enough to make something this
spectacularly awful"
test_sentence(sentence)
```

```
Sentence: Moonlight is by far the best movie of 2016.
P(positive) = 0.932 : Positive
Sentence: It's amazing anyone could be talented enough to make something this
spectacularly awful
P(positive) = 0.002 : Negative
```