UDACITY

‹ Return to "AI for Trading" in the classroom

DISCUSS ON STUDENT HUB

Sentiment Analysis with Neural Networks

REVIEW

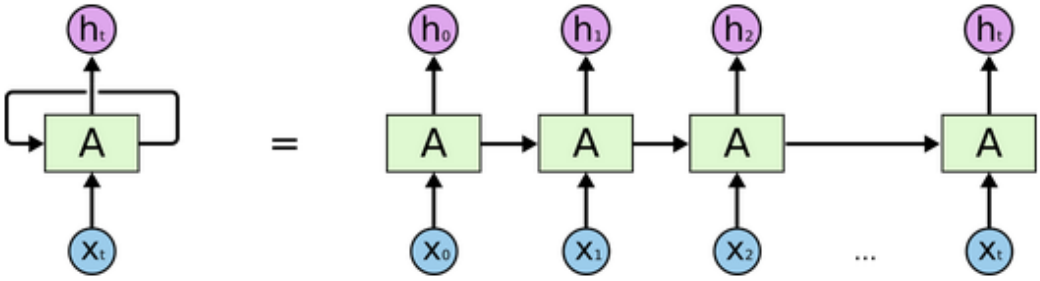
HISTORY

Meets Specifications

Congratulations! You have met all the specifications! I really enjoyed reviewing your project.

I'd highly encourage you to go over this classical blog on LSTM http://colah.github.io/posts/2015-08-Understanding-LSTMs/
In reality the LSTM network is implemented as given below at the left side of the diagram i.e. as the rolled

network. The right-side of the diagram is just for the people to understand all the math and computational theory (it's like we are "imagining" how the network will look like when it is unrolled).



An unrolled recurrent neural network.

Great work and be sure to check out the below review!

Importing Twits

Print the number of twits in the dataset.

👍

Preprocessing the Data

The function `preprocess` correctly lowercases, removes URLs, removes ticker symbols, removes punctuation, tokenizes, and removes any single character tokens.

A regular expression, or Regex for short, allows us to search for patterns in text in a fast and automated manner.

```
text = re.sub('[^a-z]', '  ', text)
```

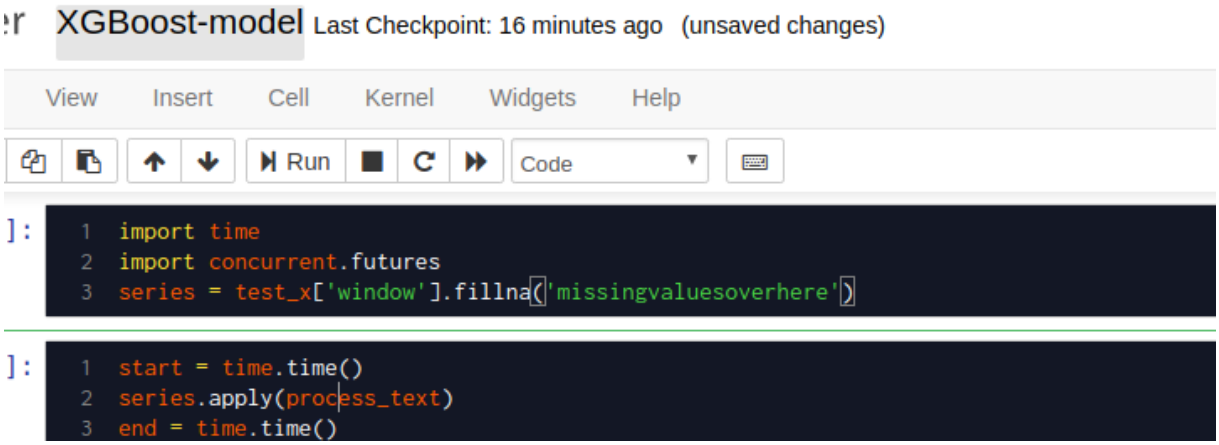
In above line of code, regex finds the pattern of text with alphabetical characters and so it removes the non-alphabetical characters like numbers, punctuation. The non-alphabetical characters have been removed from the text in the pre-processing steps.

Preprocess all the twits into the `tokenized` variable.

It took more than 3 minutes to tokenize the messages. How can you speed-up the operation? There are many ways to try to speedup the runtime. You can try out multi-processing because the given operation is CPU bound. Check out the below code-snippet;

```
import concurrent.futures
pool = concurrent.futures.ProcessPoolExecutor(max_workers=3)
tokenized = list(pool.map(preprocess, messages, chunksize=100))
```

This is multi-core processing. I think that the workspace has only one core so you will not see the speed-up while running over the workspace. But you should be able to observe the speed-up locally provided your computer has multiple cores. Check out the below snapshot of my own work;



```
print("%.3f secs" % float(end-start))
```

```
33.484 secs
```

```
]:
start = time.time()
pool2 = concurrent.futures.ProcessPoolExecutor(max_workers=3)
pd.Series(pool2.map(process_text, series, chunksize=10))
end = time.time()
print("%.3f secs" % float(end-start))
```

```
17.839 secs
```

Create a bag of words using the tokenized data.

The bag of words is used to encode the words into the numerical form. Because the model cannot use the text directly in its matrix computation.

 # This BOW will not be used because it is not filtered to eliminate common words.

The filtering to eliminate the common and rare words is carried out in the next cell. I will explain in detail in the next rubric.

In the below line of code, word_counts dictionary is also not filtered with the common words;

```
freqs = {word: count/total_count for word, count in word_counts.items()}
```

You can replace word_counts with bow and still get the same outcome;

```
freqs = {word: count/total_count for word, count in bow.items()}
```

Remove most common and rare words by defining the following variables: `freqs`, `low_cotoff`, `high_cutoff`, `K_most_common`.

Here, you have to remove/filter out two kinds of tokens.

Very common words like 'the', 'a', 'an' which don't tell anything about the sentiments of the tweet messages. Very rare words that show up in very few tweets messages.

You need to set high_cutoff as an integer. You can determine its appropriate value by looking frequency counts of the common words like "the", "a", "an"...

You need to set low_cutoff as a float which is like a lower threshold to remove rare words.

The way I see is that by dividing with frequency counts of token with total number of number of messages in the dataset, the threshold values of high_cutoff and low_cutoff are rescaled.

High-cutoff will not have too high value. And low_cutoff can be like (number of counts of rare words like 5 or 10 counts) divided by total number of messages. I think it just gives you better sense of putting these

threshold values.

Defining the variables : 'vacab', 'id2vocab' and 'filtered' correctly.

You can always pickle and serialize these variables and save it to the workspace. So that you don't have to re-run all the variables again. That is you can pick up the work where you left off in the last session. Below are the code snippets of pickling the tokenize variable which can also be done the same for the 'vocab', 'id2vocab' and 'filtered'

#loading the tokenized file

```
import pickle
tokenized = pickle.load( open( "tokenized.p", "rb" ) )
```

#saving the tokenized file

```
import pickle
pickle.dump( tokenized, open( "tokenized.p", "wb" ) )
```

Neural Network

The init function correctly initializes the following parameters: `self.vocab_size`, `self.embed_size`, `self.lstm_size`, `self.lstm_layers`, `self.dropout`, `self.embedding`, `self.lstm`, and `self.fc`.

```
        # Setup embedding layer
        self.embedding = nn.Embedding(vocab_size, embed_size)
```

Nice work in including the embedding as an input layer to the LSTM network. Word Embedding is a technique for learning dense representation of words in a low dimensional vector space. Each word can be seen as a point in this space, represented by a fixed length vector. Semantic relations between words are captured by this technique.
Reference;
https://indico.io/blog/sequence-modeling-neuralnets-part1

You can also use pre-trained embedding layer. Check out this blog https://medium.com/@martinpella/how-to-use-pre-trained-word-embeddings-in-pytorch-71ca59249f76

Dropout is used to prevent overfitting. Overfitting is when the model is performing well on the training set but fails to perform well on the validation/unseen dataset. Therefore you should assess the trained model on to the validation set. Check out the dropout paper. It is quite easy to follow;
http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf

```
        self.dropout = nn.Dropout(dropout)
```

The 'init_hidden' function generates a hidden state

LSTMs are the type of the recurrent neural nets. In recurrent neural nets weight parameters are shared between the hidden units. That means information given to the hidden unit at time t is not only coming from input unit but also from the hidden units of previous time stamps. So the input nodes and previous hidden states are concatenated together and then multiplied with weights. As a result, back-propagation through time is carried out as the inputs from the previous timestamps are also considered. The sequential learning by sharing the weights is carried within the sequence length hyperparameter in the RNN layer.

The 'forward' function performs a forward pass of the model the parameter input using the hidden state.

Logits are created from the output layer. Logits means linear combination of weights multiplied by the units (from previous layer) and then bias being added. This is the term used for the output of neural net's output layer before adding any non-linear activation function.

```
        self.fc = nn.Linear(lstm_size, output_size)
```

These logits are used by the non-linear activation function like softmax or sigmoid activation function which will generate the predicted values as below taken from Build the Graph section.

```
        out = self.fc(out)
        logps = self.softmax(out)
```

Training

Correctly split the data into `train_features`, `valid_features`, `train_labels`, and `valid_labels`.

Check out this Udacity video to see why the splitting of the data is needed to be done;
https://www.youtube.com/watch?v=nqEYVzJLR_c&list=PLAwxTw4SYaPn_OWPFT9ulXLuQrImzHfOV&index=17

Train your model with dropout and clip the gradient. Print out the training progress with the loss and accuracy.

All the below steps are taken care of in the training of the LSTM network;

1)

```
# zero accumulated gradients
```

```
model.zero_grad()
```

--- zero accumulated gradients
2)

```
    output, hidden = model(text_batch, hidden)
```

-- gets the output from the model

3)

```
    # calculate the loss and perform backprop
    loss = criterion(output.squeeze(), labels)
```

4)
`nn.utils.clip_grad_norm_(model.parameters(), clip)` -- helps prevent the exploding gradient problem in RNNs / LSTMs. Without clipping, your network still runs but it underperforms and the network has a larger loss. Clipping gradient helps prevent the exploding gradient problem in RNNs / LSTMs.

It would have been nicer if you had went ahead and made the prediction using the validation set. Notice that you have not used validation set to assess the performance of the network during the training of the model.

Making Predictions

The `predict` function correctly prints out the prediction vector from the trained model.

Nice work! One of the difference between the prediction and training of the network is that during the prediction, the model is used to give out the prediction output values and there's no weight updates in the model during the prediction. On the other hand, during the training phase, the model is used to calculate the weight values by using gradient descent and minimize the prediction error by the back-propagation algorithm.

Answer what the prediction of the model is and the uncertainty of the prediction.

It is truly a remarkable sight that your network is able to predict the sentiments of the tweets. It is amazing indeed.

DOWNLOAD PROJECT

RETURN TO PATH