Project 6: Analyzing Stock Sentiment from Twits

Instructions

Each problem consists of a function to implement and instructions on how to implement the function. The parts of the function that need to be implemented are marked with a `# TODO` comment.

Packages

When you implement the functions, you'll only need to you use the packages you've used in the classroom, like Pandas and Numpy. These packages will be imported for you. We recommend you don't add any import statements, otherwise the grader might not be able to run your code.

Load Packages

In [79]:
```
import json
import nltk
import os
import random
import re
import torch
import numpy as np

from torch import nn, optim
import torch.nn.functional as F
```

Introduction

When deciding the value of a company, it's important to follow the news. For example, a product recall or natural disaster in a company's product chain. You want to be able to turn this information into a signal. Currently, the best tool for the job is a Neural Network.

For this project, you'll use posts from the social media site StockTwits. The community on StockTwits is full of investors, traders, and entrepreneurs. Each message posted is called a Twit. This is similar to Twitter's version of a post, called a Tweet. You'll build a model around these twits that generate a sentiment score.

We've collected a bunch of twits, then hand labeled the sentiment of each. To capture the degree of sentiment, we'll use a five-point scale: very negative, negative, neutral, positive, very positive. Each twit is labeled -2 to 2 in steps of 1, from very negative to very positive respectively. You'll build a sentiment analysis model that will learn to assign sentiment to twits on its own, using this labeled data.

The first thing we should to do, is load the data.

Import Twits

Load Twits Data

This JSON file contains a list of objects for each twit in the `'data'` field:

```
{'data':
  {'message_body': 'Neutral twit body text here',
   'sentiment': 0},
  {'message_body': 'Happy twit body text here',
   'sentiment': 1},
   ...
}
```

The fields represent the following:

- `'message_body'`: The text of the twit.
- `'sentiment'`: Sentiment score for the twit, ranges from -2 to 2 in steps of 1, with 0 being neutral.

To see what the data look like by printing the first 10 twits from the list.

In [2]:
```
with open(os.path.join('..', '..', 'data', 'project_6_stocktwits', 'twits
    twits = json.load(f)

print(twits['data'][:10])
```

```
[{'message_body': '$FITB great buy at 26.00...ill wait', 'sentiment': 2,
'timestamp': '2018-07-01T00:00:09Z'}, {'message_body': '@StockTwits $MSF
T', 'sentiment': 1, 'timestamp': '2018-07-01T00:00:42Z'}, {'message_bod
y': '#STAAnalystAlert for $TDG : Jefferies Maintains with a rating of Hol
d setting target price at USD 350.00. Our own verdict is Buy  http://www.
stocktargetadvisor.com/toprating', 'sentiment': 2, 'timestamp': '2018-07-
01T00:01:24Z'}, {'message_body': '$AMD I heard there's a guy who knows so
meone who thinks somebody knows something - on StockTwits.', 'sentiment':
1, 'timestamp': '2018-07-01T00:01:47Z'}, {'message_body': '$AMD reveal yo
urself!', 'sentiment': 0, 'timestamp': '2018-07-01T00:02:13Z'}, {'message
_body': '$AAPL Why the drop? I warren Buffet taking out his position?', '
sentiment': 1, 'timestamp': '2018-07-01T00:03:10Z'}, {'message_body': '$B
A bears have 1 reason on 06-29 to pay more attention https://dividendbot.
com?s=BA', 'sentiment': -2, 'timestamp': '2018-07-01T00:04:09Z'}, {'messa
ge_body': '$BAC ok good we&#39;re not dropping in price over the weekend,
lol', 'sentiment': 1, 'timestamp': '2018-07-01T00:04:17Z'}, {'message_bod
y': '$AMAT - Daily Chart, we need to get back to above 50.', 'sentiment':
2, 'timestamp': '2018-07-01T00:08:01Z'}, {'message_body': '$GME 3% drop p
er week after spike... if no news in 3 months, back to 12s... if BO, then
bingo... what is the odds?', 'sentiment': -2, 'timestamp': '2018-07-01T0
0:09:03Z'}]
```

Length of Data

Now let's look at the number of twits in dataset. Print the number of twits below.

In [3]:
```
"""print out the number of twits"""

# TODO Implement
print(len(twits['data']))
```

```
1548010
```

Split Message Body and Sentiment Score

In [4]:
```
messages = [twit['message_body'] for twit in twits['data']]
# Since the sentiment scores are discrete, we'll scale the sentiments to
sentiments = [twit['sentiment'] + 2 for twit in twits['data']]
```

Preprocessing the Data

With our data in hand we need to preprocess our text. These twits are collected by filtering on ticker symbols where these are denoted with a leader $ symbol in the twit itself. For example,

`{'message_body': 'RT @google Our annual look at the year in Google blogging (and beyond) http://t.co/sptHOAh8 $GOOG', 'sentiment': 0}`

The ticker symbols don't provide information on the sentiment, and they are in every twit, so we should remove them. This twit also has the `@google` username, again not providing sentiment information, so we should also remove it. We also see a URL `http://t.co/sptHOAh8`. Let's remove these too.

The easiest way to remove specific words or phrases is with regex using the `re` module. You can sub out specific patterns with a space:

```
re.sub(pattern, ' ', text)
```

This will substitute a space with anywhere the pattern matches in the text. Later when we tokenize the text, we'll split appropriately on those spaces.

Pre-Processing

```
In [5]: nltk.download('wordnet')


        def preprocess(message):
            """
            This function takes a string as input, then performs these operations
                - lowercase
                - remove URLs
                - remove ticker symbols
                - removes punctuation
                - tokenize by splitting the string on whitespace
                - removes any single character tokens

            Parameters
            ----------
                message : The text message to be preprocessed.

            Returns
            -------
                tokens: The preprocessed text into tokens.
            """
            #TODO: Implement

            # Lowercase the twit message
            text = message.lower()

            # Replace URLs with a space in the message
            text = re.sub(r'http://[^\s]+', ' ', text)

            # Replace ticker symbols with a space. The ticker symbols are any sto
            # Note: as the message has been processed by .lower(), we need to inc
            text = re.sub(r'\$[a-zA-Z]{2,4}', ' ', text)

            # Replace StockTwits usernames with a space. The usernames are any wo
            text = re.sub(r'@[^\s]+', ' ', text)

            # Replace everything not a letter with a space
            text = re.sub(r'\W', ' ', text)

            # Tokenize by splitting the string on whitespace into a list of words
            tokens = re.split(r'\s', text)

            # Lemmatize words using the WordNetLemmatizer. You can ignore any wor
            wnl = nltk.stem.WordNetLemmatizer()
            tokens = [wnl.lemmatize(w) for w in tokens if len(w)>1]

            return tokens
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Unzipping corpora/wordnet.zip.
```

Preprocess All the Twits

Now we can preprocess each of the twits in our dataset. Apply the function `preprocess` to all the twit messages.

In [7]:
```
# Test preprocess
test_string = "01 http://ddd.com 02 $AB $ABC $ABCD 03 @willi @as-name 04
result_string = preprocess(test_string)
print(result_string)
print(len(result_string))
```

```
['01', '02', '03', '04', 'number4', 'hyphen', 'underscore_', '05', '06',
'excpectedtokennumberisten']
10
```

In [8]:
```
# TODO Implement
tokenized = [preprocess(twit) for twit in messages]
```

In [9]:
```
print(tokenized[0:3])
print(len(tokenized))
```

```
[['great', 'buy', 'at', '26', '00', 'ill', 'wait'], [], ['staanalystaler
t', 'for', 'jefferies', 'maintains', 'with', 'rating', 'of', 'hold', 'set
ting', 'target', 'price', 'at', 'usd', '350', '00', 'our', 'own', 'verdic
t', 'is', 'buy']]
1548010
```

Bag of Words

Now with all of our messages tokenized, we want to create a vocabulary and count up how often each word appears in our entire corpus. Use the `Counter` function to count up all the tokens.

In [10]:
```
from collections import Counter


"""
Create a vocabulary by using Bag of words
"""

# TODO: Implement
# I used this information on how to use Counter with a list of lists:
# https://stackoverflow.com/questions/19211018/using-counter-with-list-of
bow = Counter()
for tokens in tokenized:
    for token in tokens:
        bow[token] += 1
```

In [11]:
```
print(len(bow))
print(bow['the'])
print(bow['man'])
```

```
214430
406528
4791
```

Frequency of Words Appearing in Message

With our vocabulary, now we'll remove some of the most common words such as 'the', 'and', 'it', etc. These words don't contribute to identifying sentiment and are really common, resulting in a lot of noise in our input. If we can filter these out, then our network should have an easier time learning.

We also want to remove really rare words that show up in a only a few twits. Here you'll want to divide the count of each word by the number of messages. Then remove words that only appear in some small fraction of the messages.

In [12]:
```
"""
Set the following variables:
    freqs
    low_cutoff
    high_cutoff
    K_most_common
"""

# TODO Implement

# Dictionary that contains the Frequency of words appearing in messages.
# The key is the token and the value is the frequency of that word in the
freqs = dict(bow)

# I was not sure how to handle low cutoff and high cutoff - I got some id
# Knowledge article 'https://knowledge.udacity.com/questions/30813'

# Float that is the frequency cutoff. Drop words with a frequency that is
low_cutoff = sum(bow.values())/len(messages)
print(sum(bow.values()))
print(len(messages))
print(low_cutoff)

# Integer that is the cut off for most common words. Drop words that are
high_cutoff = 20

# The k most common words in the corpus. Use `high_cutoff` as the k.
K_most_common = [item for item, count in bow.most_common(high_cutoff)]

filtered_words = [word for word in freqs if (freqs[word] > low_cutoff and
print(K_most_common)
print(len(filtered_words))
print(filtered_words[0:3])
print('at' not in filtered_words)
```

```
19763375
1548010
12.766955639821449
['the', 'to', 'is', 'for', 'on', 'http', 'of', 'and', 'in', 'this', 'co
m', '39', 'it', 'at', 'will', 'amp', 'up', 'are', 'stock', 'be']
23674
['great', 'buy', '26']
True
```

Updating Vocabulary by Removing Filtered Words

Let's creat three variables that will help with our vocabulary.

In [13]:
```
"""
Set the following variables:
    vocab
    id2vocab
    filtered
"""


#TODO Implement

# A dictionary for the `filtered_words`. The key is the word and value is
vocab = dict(zip(filtered_words, range(len(filtered_words))))
print("Value of 'man': {}".format(vocab['man']))

# Reverse of the `vocab` dictionary. The key is word id and value is the

# I used this site to get an idea on how to swap keys and values of a dic
# https://www.geeksforgeeks.org/python-program-to-swap-keys-and-values-in

id2vocab = dict([(value, key) for key, value in vocab.items()])
print("Value of '2966': {}".format(id2vocab[2966]))

# tokenized with the words not in `filtered_words` removed.
print("Number of words in tokenized: {}".format(sum([1 for twit in tokeni

#filtered = []
#for twit in tokenized:
#    for word in twit:
#        if word not in filtered_words:
#            twit.remove(word)
#    filtered.append(twit)

# I used this stackoverflow entry to understand how to best implement thi
# https://stackoverflow.com/questions/18551458/how-to-frame-two-for-loops
# filtered = [[word for word in twit if word in filtered_words] for twit

# After still having problems with wrong keys, I implemented this solutio
# according to this Knowledge Article: https://knowledge.udacity.com/ques
filtered = [[word for word in twit if word in vocab] for twit in tokenize

print("Number of words in filtered: {}".format(sum([1 for twit in filtere
Value of 'man': 2966
Value of '2966': man
Number of words in tokenized: 19763375
Number of words in filtered: 15274626
```

In [14]:

```
print(filtered[0:3])
```

```
[['great', 'buy', '26', '00', 'ill', 'wait'], [], ['staanalystalert', 'je
fferies', 'maintains', 'with', 'rating', 'hold', 'setting', 'target', 'pr
ice', 'usd', '350', '00', 'our', 'own', 'verdict', 'buy']]
```

Balancing the classes

Let's do a few last pre-processing steps. If we look at how our twits are labeled, we'll find that 50% of them are neutral. This means that our network will be 50% accurate just by guessing 0 every single time. To help our network learn appropriately, we'll want to balance our classes. That is, make sure each of our different sentiment scores show up roughly as frequently in the data.

What we can do here is go through each of our examples and randomly drop twits with neutral sentiment. What should be the probability we drop these twits if we want to get around 20% neutral twits starting at 50% neutral? We should also take this opportunity to remove messages with length 0.

In [15]:

```
balanced = {'messages': [], 'sentiments':[]}

n_neutral = sum(1 for each in sentiments if each == 2)
N_examples = len(sentiments)
keep_prob = (N_examples - n_neutral)/4/n_neutral

for idx, sentiment in enumerate(sentiments):
    message = filtered[idx]
    if len(message) == 0:
        # skip this message because it has length zero
        continue
    elif sentiment != 2 or random.random() < keep_prob:
        balanced['messages'].append(message)
        balanced['sentiments'].append(sentiment)

print(len(sentiments))
print(len(balanced['sentiments']))
print(len(balanced['messages']))
```

```
1548010
1038989
1038989
```

If you did it correctly, you should see the following result

In [16]:

```
n_neutral = sum(1 for each in balanced['sentiments'] if each == 2)
N_examples = len(balanced['sentiments'])
n_neutral/N_examples
```

Out[16]:
```
0.19512718613960303
```

Finally let's convert our tokens into integer ids which we can pass to the network.

In [17]:
```
token_ids = [[vocab[word] for word in message] for message in balanced['m
sentiments = balanced['sentiments']
```

Neural Network

Now we have our vocabulary which means we can transform our tokens into ids, which are then passed to our network. So, let's define the network now!

Here is a nice diagram showing the network we'd like to build:

Embed -> RNN -> Dense -> Softmax

Implement the text classifier

Before we build text classifier, if you remember from the other network that you built in "Sentiment Analysis with an RNN" exercise - which there, the network called " SentimentRNN", here we named it "TextClassifer" - consists of three main parts: 1) init function `__init__` 2) forward pass `forward` 3) hidden state `init_hidden` .

This network is pretty similar to the network you built expect in the `forward` pass, we use softmax instead of sigmoid. The reason we are not using sigmoid is that the output of NN is not a binary. In our network, sentiment scores have 5 possible outcomes. We are looking for an outcome with the highest probability thus softmax is a better choice.

In [18]:
```
class TextClassifier(nn.Module):
    def __init__(self, vocab_size, embed_size, lstm_size, output_size, ls
        """
        Initialize the model by setting up the layers.

        Parameters
        ----------
            vocab_size : The vocabulary size.
            embed_size : The embedding layer size.
            lstm_size : The LSTM layer size.
            output_size : The output size.
            lstm_layers : The number of LSTM layers.
            dropout : The dropout probability.
        """

        super().__init__()
        self.vocab_size = vocab_size
        self.embed_size = embed_size
        self.lstm_size = lstm_size
        self.output_size = output_size
        self.lstm_layers = lstm_layers
        self.dropout = dropout

        # TODO Implement

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

        # Setup additional layers
        self.lstm = nn.LSTM(embed_size, lstm_size, lstm_layers,
                            dropout=dropout, batch_first=False)
        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(lstm_size, output_size)
        self.lsm = torch.nn.LogSoftmax(dim=1)


    def init_hidden(self, batch_size):
        """
        Initializes hidden state

        Parameters
        ----------
            batch_size : The size of batches.

        Returns
        -------
            hidden_state

        """

        # TODO Implement

        # Create two new tensors with sizes n_layers x batch_size x hidde
        # initialized to zero, for hidden state and cell state of LSTM

        weight = next(self.parameters()).data

        hidden_state = (weight.new(self.lstm_layers, batch_size, self.lst
                        weight.new(self.lstm_layers, batch_size, self.lst

        return hidden_state


    def forward(self, nn_input, hidden_state):
        """
        Perform a forward pass of our model on nn_input.

        Parameters
        ----------
            nn_input : The batch of input to the NN.
            hidden_state : The LSTM hidden state.

        Returns
        -------
            logps: log softmax output
            hidden_state: The new hidden state.
        """

        # TODO Implement

        batch_size = nn_input.size(0)

        # embeddings and lstm_out
        nn_input = nn_input.long()
        embeds = self.embedding(nn_input)
        lstm_out, hidden_state = self.lstm(embeds, hidden_state)

        # stack up lstm outputs
        # This Udacity Knowledge post helped me getting the next line rig
        # https://knowledge.udacity.com/questions/30112
        lstm_out = lstm_out[-1,:,:]#lstm_out.contiguous().view(-1, self.l

        # dropout and fully-connected layer
        out = self.dropout(lstm_out)
        out = self.fc(out)
        # softmax function
        logps = self.lsm(out)

        # return last sigmoid output and hidden state
        return logps, hidden_state
```

View Model

In [19]:
```
model = TextClassifier(len(vocab), 10, 6, 5, dropout=0.1, lstm_layers=2)
model.embedding.weight.data.uniform_(-1, 1)
input = torch.randint(0, 1000, (5, 4), dtype=torch.int64)
hidden = model.init_hidden(4)

logps, _ = model.forward(input, hidden)
print(logps)
```

```
tensor([[-1.8725, -1.6630, -1.5742, -1.9080, -1.2002],
        [-1.9027, -1.6005, -1.6393, -1.8531, -1.2100],
        [-1.8434, -1.6125, -1.6066, -1.8117, -1.2787],
        [-1.9429, -1.5561, -1.5058, -1.8966, -1.2952]])
```

Training

DataLoaders and Batching

Now we should build a generator that we can use to loop through our data. It'll be more efficient if we can pass our sequences in as batches. Our input tensors should look like `(sequence_length, batch_size)`. So if our sequences are 40 tokens long and we pass in 25 sequences, then we'd have an input size of `(40, 25)`.

If we set our sequence length to 40, what do we do with messages that are more or less than 40 tokens? For messages with fewer than 40 tokens, we will pad the empty spots with zeros. We should be sure to **left** pad so that the RNN starts from nothing before going through the data. If the message has 20 tokens, then the first 20 spots of our 40 long sequence will be 0. If a message has more than 40 tokens, we'll just keep the first 40 tokens.

In [20]:
```
def dataloader(messages, labels, sequence_length=30, batch_size=32, shuff
    """
    Build a dataloader.
    """
    if shuffle:
        indices = list(range(len(messages)))
        random.shuffle(indices)
        messages = [messages[idx] for idx in indices]
        labels = [labels[idx] for idx in indices]

    total_sequences = len(messages)

    for ii in range(0, total_sequences, batch_size):
        batch_messages = messages[ii: ii+batch_size]

        # First initialize a tensor of all zeros
        batch = torch.zeros((sequence_length, len(batch_messages)), dtype
        for batch_num, tokens in enumerate(batch_messages):
            token_tensor = torch.tensor(tokens)
            # Left pad!
            start_idx = max(sequence_length - len(token_tensor), 0)
            batch[start_idx:, batch_num] = token_tensor[:sequence_length]

        label_tensor = torch.tensor(labels[ii: ii+len(batch_messages)])

        yield batch, label_tensor
```

Training and Validation

With our data in nice shape, we'll split it into training and validation sets.

In [21]:
```
"""
Split data into training and validation datasets. Use an appropriate spli
The features are the `token_ids` and the labels are the `sentiments`.
"""

# TODO Implement
split_value = int(len(token_ids)*0.66)
train_features = token_ids[:split_value]
valid_features = token_ids[split_value:]
train_labels = sentiments[:split_value]
valid_labels = sentiments[split_value:]
```

In [22]:
```
text_batch, labels = next(iter(dataloader(train_features, train_labels, s
model = TextClassifier(len(vocab)+1, 200, 128, 5, dropout=0.)
hidden = model.init_hidden(1024)
logps, hidden = model.forward(text_batch, hidden)
```

Training

It's time to train the neural network!

In [23]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = TextClassifier(len(vocab)+1, 1024, 512, 5, lstm_layers=2, dropout
model.embedding.weight.data.uniform_(-1, 1)
model.to(device)
```

Out[23]:
```
TextClassifier(
  (embedding): Embedding(23675, 1024)
  (lstm): LSTM(1024, 512, num_layers=2, dropout=0.2)
  (dropout): Dropout(p=0.2)
  (fc): Linear(in_features=512, out_features=5, bias=True)
  (lsm): LogSoftmax()
)
```

In [32]:
```
"""
Train your model with dropout. Make sure to clip your gradients.
Print the training loss, validation loss, and validation accuracy for eve
"""

epochs = 4
batch_size = 1024
learning_rate = 0.002
clip = 5

print_every = 100
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
model.train()

for epoch in range(epochs):
    print('Starting epoch {}'.format(epoch + 1))

    steps = 0
    for text_batch, labels in dataloader(
            train_features, train_labels, batch_size=batch_size, sequence
        steps += 1
        hidden = model.init_hidden(labels.shape[0])

        # Set Device
        text_batch, labels = text_batch.to(device), labels.to(device)
        for each in hidden:
            each.to(device)

        # TODO Implement: Train Model
        ###hidden = tuple([each.data for each in hidden])

        # Zero gradients to the model
        model.zero_grad()

        # Get output from model
        output, hidden = model(text_batch, hidden)

        # Get the loss
        #loss = criterion(output.squeeze(), labels.float())
        loss = criterion(output, labels)
        # And do backpropagation through the net
        loss.backward()

        # Clip gradients
        nn.utils.clip_grad_norm_(model.parameters(), clip)
        optimizer.step()

        # Do evaluation
        if steps % print_every == 0:
            model.eval()
            val_losses = []
            val_accuracy = []

            # TODO Implement: Print metrics
            # I added the next line on 'torch.no_grad()' from Udacity Kno
            # https://knowledge.udacity.com/questions/30112
            with torch.no_grad():
                for text_batch, labels in dataloader(
                    valid_features, valid_labels, batch_size=batch_size,
                    # Set Device
                    text_batch, labels = text_batch.to(device), labels.to
                    for each in hidden:
                        each.to(device)

                    # I have implemented val_hidden according to this Uda
                    # https://knowledge.udacity.com/questions/30112
                    # val_hidden = model.init_hidden(batch_size)
                    val_hidden = model.init_hidden(labels.shape[0])
                    ###val_hidden = tuple([each.data for each in hidden])
                    output, val_hidden = model(text_batch, val_hidden)

                    #val_loss = criterion(output.squeeze(), labels.float(
                    val_loss = criterion(output, labels)
                    val_losses.append(val_loss.item())
                    # Calculate accuracy
                    # I copied this way to calculate accuracy from this U
                    # https://knowledge.udacity.com/questions/30112
                    ps = torch.exp(output)
                    top_p, top_class = ps.topk(1, dim=1)
                    equals = top_class == labels.view(*top_class.shape)
                    val_accuracy.append(torch.mean(equals.type(torch.Floa


            model.train()
            print("Epoch: {}/{}".format(epoch+1, epochs),
                 "Step: {}".format(steps),
                 "Loss: {:.6f}".format(loss.item()),
                 "Val Loss: {:.6f}".format(torch.mean(torch.FloatTensor(v
                 "Accuracy: {:.6f}".format(torch.mean(torch.FloatTensor(v
```

```
Starting epoch 1
Epoch: 1/4 Step: 100 Loss: 0.625009 Val Loss: 0.721500 Accuracy: 0.720062
Epoch: 1/4 Step: 200 Loss: 0.664538 Val Loss: 0.715109 Accuracy: 0.720920
Epoch: 1/4 Step: 300 Loss: 0.568557 Val Loss: 0.708796 Accuracy: 0.724292
Epoch: 1/4 Step: 400 Loss: 0.626432 Val Loss: 0.706605 Accuracy: 0.725578
Epoch: 1/4 Step: 500 Loss: 0.688643 Val Loss: 0.703242 Accuracy: 0.723914
Epoch: 1/4 Step: 600 Loss: 0.600010 Val Loss: 0.693020 Accuracy: 0.729164
Starting epoch 2
Epoch: 2/4 Step: 100 Loss: 0.480061 Val Loss: 0.739660 Accuracy: 0.722462
Epoch: 2/4 Step: 200 Loss: 0.551094 Val Loss: 0.738278 Accuracy: 0.720338
Epoch: 2/4 Step: 300 Loss: 0.532632 Val Loss: 0.737797 Accuracy: 0.722730
Epoch: 2/4 Step: 400 Loss: 0.525339 Val Loss: 0.732806 Accuracy: 0.721732
Epoch: 2/4 Step: 500 Loss: 0.588526 Val Loss: 0.731353 Accuracy: 0.723119
Epoch: 2/4 Step: 600 Loss: 0.541532 Val Loss: 0.721096 Accuracy: 0.722308
Starting epoch 3
Epoch: 3/4 Step: 100 Loss: 0.456256 Val Loss: 0.838892 Accuracy: 0.715452
Epoch: 3/4 Step: 200 Loss: 0.458665 Val Loss: 0.806221 Accuracy: 0.715068
Epoch: 3/4 Step: 300 Loss: 0.455940 Val Loss: 0.795028 Accuracy: 0.712638
Epoch: 3/4 Step: 400 Loss: 0.455393 Val Loss: 0.790634 Accuracy: 0.715915
Epoch: 3/4 Step: 500 Loss: 0.480113 Val Loss: 0.784372 Accuracy: 0.717572
Epoch: 3/4 Step: 600 Loss: 0.503228 Val Loss: 0.784281 Accuracy: 0.717985
Starting epoch 4
Epoch: 4/4 Step: 100 Loss: 0.379992 Val Loss: 0.903987 Accuracy: 0.706731
Epoch: 4/4 Step: 200 Loss: 0.369411 Val Loss: 0.914127 Accuracy: 0.708196
Epoch: 4/4 Step: 300 Loss: 0.403572 Val Loss: 0.917642 Accuracy: 0.708683
Epoch: 4/4 Step: 400 Loss: 0.389425 Val Loss: 0.887015 Accuracy: 0.707703
Epoch: 4/4 Step: 500 Loss: 0.410250 Val Loss: 0.855452 Accuracy: 0.709232
Epoch: 4/4 Step: 600 Loss: 0.374852 Val Loss: 0.862303 Accuracy: 0.713802
```

In [80]:

```
# Save the model after training
# Solution 1: Complete model
#torch.save(model, 'checkpoint.pth')

# Solution 2: state_dict() only
#torch.save(model.state_dict(), 'checkpoint_statedict.pth')
```

In [81]:
```
# Load the model
# Solution 1: Complete model
#model2 = torch.load('checkpoint.pth')

# Solution 2: state_dict() only
#model2 = TextClassifier(len(vocab)+1, 1024, 512, 5, lstm_layers=2, dropo
#model2.load_state_dict(torch.load('checkpoint_statedict.pth'))
#model2.eval()

#print(model2)
```

Making Predictions

Prediction

Okay, now that you have a trained model, try it on some new twits and see if it works appropriately. Remember that for any new text, you'll need to preprocess it first before passing it to the network. Implement the `predict` function to generate the prediction vector from a message.

In [66]:
```
def predict(text, model, vocab):
    """
    Make a prediction on a single sentence.

    Parameters
    ----------
        text : The string to make a prediction on.
        model : The model to use for making the prediction.
        vocab : Dictionary for word to word ids. The key is the word and

    Returns
    -------
        pred : Prediction vector
    """

    # TODO Implement

    tokens = preprocess(text)

    # Filter non-vocab words
    tokens = [word for word in tokens if word in vocab]
    # Convert words to ids
    tokens = [vocab[word] for word in tokens]

    # Adding a batch dimension
    # I used this Udacity Knowledge post to get a solution for batch dime
    # https://knowledge.udacity.com/questions/30968
    text_input = torch.from_numpy(np.asarray(tokens)).view(-1,1)
    # Get the NN output
    hidden = model.init_hidden(1)
    logps, _ = model.forward(text_input, hidden)
    # Take the exponent of the NN output to get a range of 0 to 1 for eac
    pred = torch.exp(logps)

    return pred
```

In [74]:
```
text = "Google is working on self driving cars, I'm bullish on $goog"
model.eval()
model.to("cpu")
predict(text, model, vocab)
```

Out[74]:
```
tensor([[ 3.8475e-07,  1.1078e-04,  2.0671e-04,  9.1383e-01,  8.5855e-0
2]])
```

Questions: What is the prediction of the model? What is the uncertainty of the prediction?

TODO: Answer Question

The prediction of the model on the twit is for category '4' which is the second most positive one. Prediction can be stated with degree of certainty of '0.91' which in turn is a degree of uncertainty of '0.09'.

Now we have a trained model and we can make predictions. We can use this model to track the sentiments of various stocks by predicting the sentiments of twits as they are coming in. Now we have a stream of twits. For each of those twits, pull out the stocks mentioned in them and keep track of the sentiments. Remember that in the twits, ticker symbols are encoded with a dollar sign as the first character, all caps, and 2-4 letters, like $AAPL. Ideally, you'd want to track the sentiments of the stocks in your universe and use this as a signal in your larger model(s).

Testing

Load the Data

In [75]:
```
with open(os.path.join('..', '..', 'data', 'project_6_stocktwits', 'test_
    test_data = json.load(f)
```

Twit Stream

In [76]:
```
def twit_stream():
    for twit in test_data['data']:
        yield twit

next(twit_stream())
```

Out[76]:
```
{'message_body': '$JWN has moved -1.69% on 10-31. Check out the movement
and peers at  https://dividendbot.com?s=JWN',
 'timestamp': '2018-11-01T00:00:05Z'}
```

Using the `prediction` function, let's apply it to a stream of twits.

In [77]:
```
def score_twits(stream, model, vocab, universe):
    """
    Given a stream of twits and a universe of tickers, return sentiment s
    """
    for twit in stream:

        # Get the message text
        text = twit['message_body']
        symbols = re.findall('\$[A-Z]{2,4}', text)
        score = predict(text, model, vocab)

        for symbol in symbols:
            if symbol in universe:
                yield {'symbol': symbol, 'score': score, 'timestamp': twi
```

In [78]:
```
universe = {'$BBRY', '$AAPL', '$AMZN', '$BABA', '$YHOO', '$LQMT', '$FB',
score_stream = score_twits(twit_stream(), model, vocab, universe)

next(score_stream)
```

Out[78]:
```
{'symbol': '$AAPL',
 'score': tensor([[ 0.0806,  0.0336,  0.0643,  0.1313,  0.6902]]),
 'timestamp': '2018-11-01T00:00:18Z'}
```

That's it. You have successfully built a model for sentiment analysis!

Submission

Now that you're done with the project, it's time to submit it. Click the submit button in the bottom right. One of our reviewers will give you feedback on your project with a pass or not passed grade. You can continue to the next section while you wait for feedback.