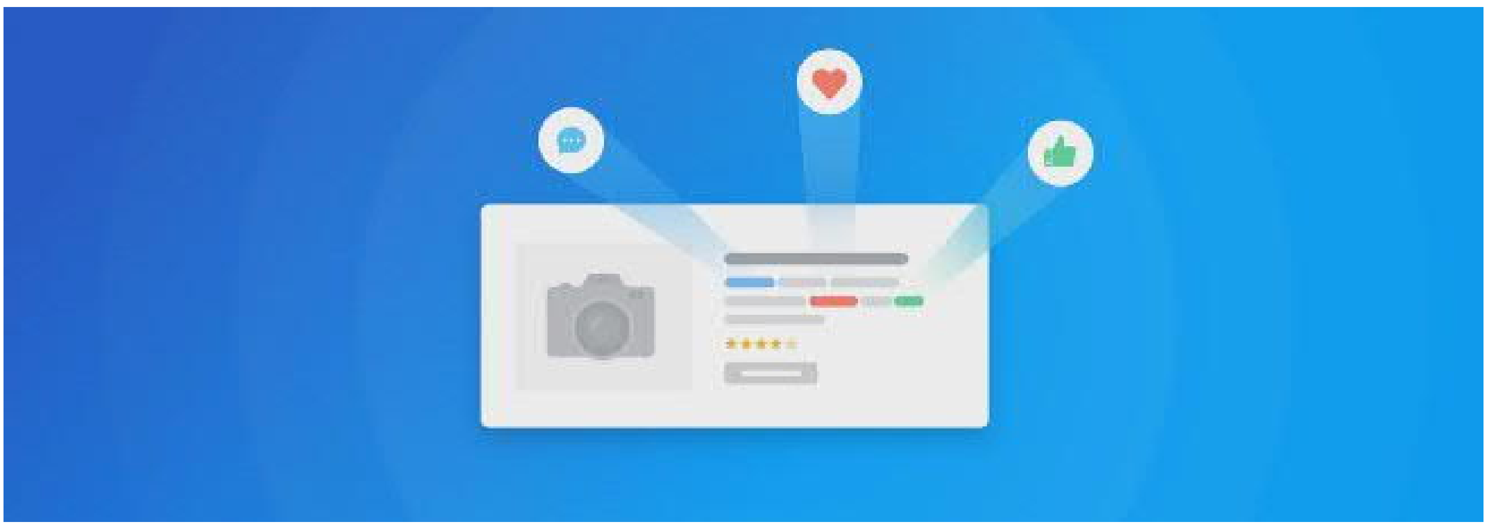
Sentiment Analysis using LSTM (Step-by-Step Tutorial)

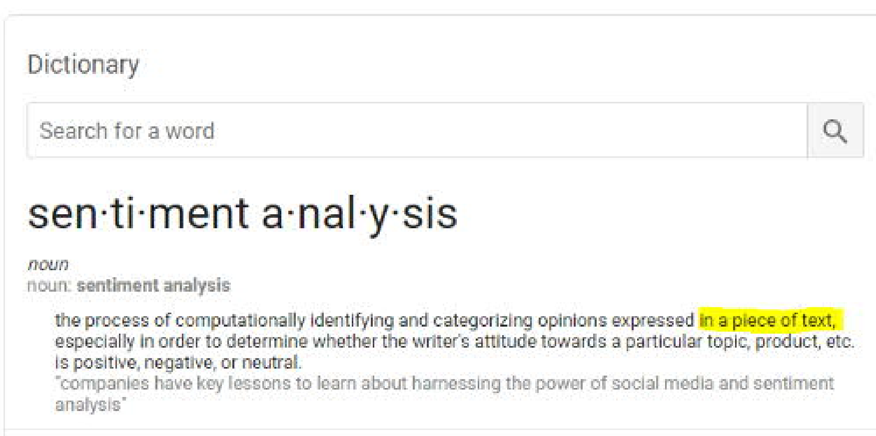
Using PyTorch framework for Deep Learning

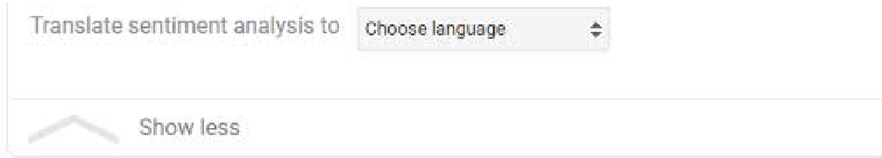
Samarth Agrawal Follow
Feb 18, 2019 · 8 min read ★



Sentiment Analysis, Image by: Monkeylearn

What is Sentiment Analysis:





Sentiment Analysis from Dictionary

I think this result from google dictionary gives a very succinct definition. I don't have to re-emphasize how important sentiment analysis has become. So, here we will build a classifier on IMDB movie dataset using a Deep Learning technique called RNN.

I'm outlining a step-by-step process for how Recurrent Neural Networks (RNN) can be implemented using Long Short Term Memory (LSTM) architecture:

1. Load in and visualize the data
2. Data Processing — convert to lower case
3. Data Processing — Remove punctuation
4. Data Processing — Create list of reviews
5. Tokenize — Create Vocab to Int mapping dictionary
6. Tokenize — Encode the words
7. Tokenize — Encode the labels
8. Analyze Reviews Length
9. Removing Outliers — Getting rid of extremely long or short reviews
10. Padding / Truncating the remaining data
11. Training, Validation, Test Dataset Split
12. Dataloaders and Batching
13. Define the LSTM Network Architecture
14. Define the Model Class
15. Training the Network
16. Testing (on Test data and User- generated data)

. . .

1) Load in and visualize the data

We are using IMDB movies review dataset. If it is stored in your machine in a txt file then we just load it in

```
# read data from text files
with open('data/reviews.txt', 'r') as f:
 reviews = f.read()
with open('data/labels.txt', 'r') as f:
 labels = f.read()

print(reviews[:50])
print()
print(labels[:26])

--- Output ---

bromwell high is a cartoon comedy . it ran at the same time as some
other programs about school life  such as  teachers  . my   years

positive
negative
positive
```

2) Data Processing — convert to lower case

```
reviews = reviews.lower()
```

3) Data Processing — remove punctuation

```
from string import punctuation
print(punctuation)

--- Output ---

!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

We saw all the punctuation symbols predefined in python. To get rid of all these punctuation we will simply use

```
all_text = ''.join([c for c in reviews if c not in punctuation])
```

4) Data Processing — create list of reviews

We have got all the strings in one huge string. Now we will separate out individual reviews and store them as individual list elements. Like, [review_1, review_2, review_3……. review_n]

```
reviews_split = all_text.split('\n')
print ('Number of reviews :', len(reviews_split))
```

Number of reviews : 25001

5) Tokenize — Create Vocab to Int mapping dictionary

In most of the NLP tasks, you will create an index mapping dictionary in such a way that your frequently occurring words are assigned lower indexes. One of the most common way of doing this is to use `Counter` method from `Collections` library.

```
from collections import Counter

all_text2 = ' '.join(reviews_split)
# create a list of words
words = all_text2.split()

# Count all the words using Counter Method
count_words = Counter(words)

total_words = len(words)
sorted_words = count_words.most_common(total_words)
```

Let's have a look at these objects we have created

```
print (count_words)

--- Output ---

Counter({'the': 336713, 'and': 164107, 'a': 163009, 'of': 145864
```

In order to create a vocab to int mapping dictionary, you would simply do this

```
vocab_to_int = {w:i for i, (w,c) in enumerate(sorted_words)}
```

There is a small trick here, in this mapping index will start from 0 i.e. mapping of 'the' will be 0. But later on we are going to do padding for shorter reviews and conventional choice for padding is 0. So we need to start this indexing from 1

```
vocab_to_int = {w:i+1 for i, (w,c) in enumerate(sorted_words)}
```

Let's have a look at this mapping dictionary. We can see that mapping for 'the' is 1 now

```
print (vocab_to_int)

--- Output ---

{'the': 1, 'and': 2, 'a': 3, 'of': 4,
```

6) Tokenize — Encode the words

So far we have created a) list of reviews and b) index mapping dictionary using vocab from all our reviews. All this was to create an encoding of reviews (replace words in our reviews by integers)

```
reviews_int = []
for review in reviews_split:
    r = [vocab_to_int[w] for w in review.split()]
    reviews_int.append(r)
print (reviews_int[0:3])

--- Output ---

[[21025, 308, 6, 3, 1050, 207, 8, 2138, 32, 1, 171, 57, 15, 49, 81,
5785, 44, 382, 110, 140, 15, .....], [5194, 60, 154, 9, 1, 4975,
5852, 475, 71, 5, 260, 12, 21025, 308, 13, 1978, 6, 74, 2395, 5, 613,
73, 6, 5194, 1, 24103, 5, ....], [1983, 10166, 1, 5786, 1499, 36, 51,
66, 204, 145, 67, 1199, 5194.....]]
```

Note: what we have created now is a list of lists. Each individual review is a list of integer values and all of them are stored in one huge list

7) Tokenize — Encode the labels

This is simple because we only have 2 output labels. So, we will just label 'positive' as 1 and 'negative' as 0

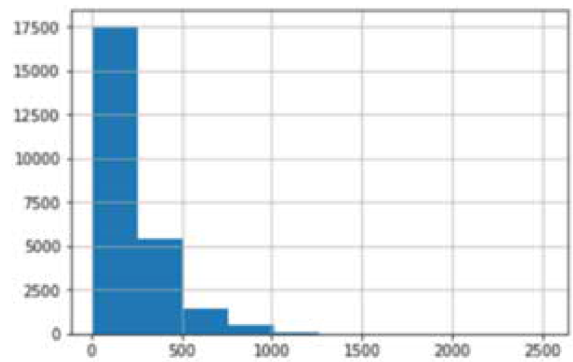
```
encoded_labels = [1 if label =='positive' else 0 for label in
labels_split]
encoded_labels = np.array(encoded_labels)
```

8) Analyze Reviews Length

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

reviews_len = [len(x) for x in reviews_int]
pd.Series(reviews_len).hist()
plt.show()

pd.Series(reviews_len).describe()
```




```
count    25001.000000
mean       240.798208
std        179.020628
min          0.000000
25%        130.000000
50%        179.000000
75%        293.000000
max       2514.000000
dtype: float64
```

Review Length Analysis

Observations : a) Mean review length = 240 b) Some reviews are of 0 length. Keeping this review won't make any sense for our analysis c) Most of the reviews less than 500 words or more d) There are quite a few reviews that are extremely long, we can manually investigate them to check whether we need to include or exclude them from our analysis

9) Removing Outliers — Getting rid of extremely long or short reviews

```
reviews_int = [ reviews_int[i] for i, l in enumerate(reviews_len) if
l>0 ]
encoded_labels = [ encoded_labels[i] for i, l in
enumerate(reviews_len) if l> 0 ]
```

10) Padding / Truncating the remaining data

To deal with both short and long reviews, we will pad or truncate all our reviews to a specific length. We define this length by **Sequence Length.** This sequence length is same as number of time steps for LSTM layer.

For reviews shorter than `seq_length`, we will pad with 0s. For reviews longer than `seq_length` we will truncate them to the first seq_length words.

```
def pad_features(reviews_int, seq_length):
    ''' Return features of review_ints, where each review is padded
with 0's or truncated to the input seq_length.
    '''
    features = np.zeros((len(reviews_int), seq_length), dtype = int)

    for i, review in enumerate(reviews_int):
        review_len = len(review)

        if review_len <= seq_length:
            zeroes = list(np.zeros(seq_length-review_len))
            new = zeroes+review


        elif review_len > seq_length:
            new = review[0:seq_length]

        features[i,:] = np.array(new)

    return features
```

Note: We are creating/maintaining a 2D array structure as we created for `reviews_int` . Output will look like this

```
print (features[:10,:])
```

```
[[    0     0     0 ...     8   215    23]
 [    0     0     0 ...    29   108  3324]
 [22382    42 46418 ...   483    17     3]
 ...
 [    0     0     0 ...    59   429  1776]
 [    0     0     0 ...    55   201    18]
 [    0     0     0 ...    18    17  1191]]
```

11) Training, Validation, Test Dataset Split

Once we have got our data in nice shape, we will split it into training, validation and test sets

train= 80% | valid = 10% | test = 10%

```
split_frac = 0.8
train_x = features[0:int(split_frac*len_feat)]
train_y = encoded_labels[0:int(split_frac*len_feat)]


remaining_x = features[int(split_frac*len_feat):]
remaining_y = encoded_labels[int(split_frac*len_feat):]


valid_x = remaining_x[0:int(len(remaining_x)*0.5)]
valid_y = remaining_y[0:int(len(remaining_y)*0.5)]


test_x = remaining_x[int(len(remaining_x)*0.5):]
test_y = remaining_y[int(len(remaining_y)*0.5):]
```

12) Dataloaders and Batching

After creating our training, test and validation data. Next step is to create dataloaders for this data. We can use generator function for batching our data into batches instead we will use a TensorDataset. This is one of a very useful utility in PyTorch for using our data with DataLoaders with exact same ease as of torchvision datasets

```
import torch
from torch.utils.data import DataLoader, TensorDataset


# create Tensor datasets
train_data = TensorDataset(torch.from_numpy(train_x),
torch.from_numpy(train_y))
valid_data = TensorDataset(torch.from_numpy(valid_x),
torch.from_numpy(valid_y))
test_data = TensorDataset(torch.from_numpy(test_x),
torch.from_numpy(test_y))


# dataloaders
batch_size = 50


# make sure to SHUFFLE your data
train_loader = DataLoader(train_data, shuffle=True,
batch_size=batch_size)
valid_loader = DataLoader(valid_data, shuffle=True,
batch_size=batch_size)
test_loader = DataLoader(test_data, shuffle=True,
batch_size=batch_size)
```

In order to obtain one batch of training data for visualization purpose we will create a data iterator

```
# obtain one batch of training data
dataiter = iter(train_loader)
sample_x, sample_y = dataiter.next()


print('Sample input size: ', sample_x.size()) # batch_size,
seq_length
print('Sample input: \n', sample_x)
print()
print('Sample label size: ', sample_y.size()) # batch_size
print('Sample label: \n', sample_y)
```

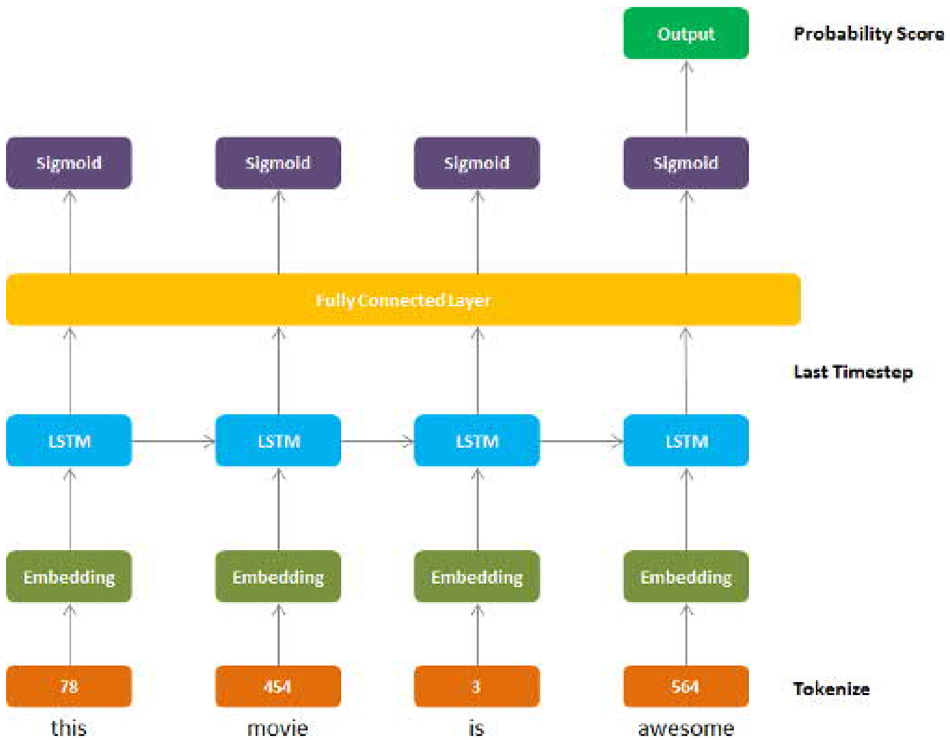
```
Sample input size:  torch.Size([50, 200])
Sample input:
 tensor([[  24,   65, 1011,  ..., 4692,    2, 1999],
        [  72,   87,   14,  ..., 6667,   31,    4],
        [   1,  240,    9,  ..., 1940,    7,    7],
        ...,
        [  11,   18,   14,  ...,  151,   23,  786],
        [   0,    0,    0,  ...,   21,  397,  243],
        [   0,    0,    0,  ...,  129, 1314, 3622]], dtype=torch.int32)

Sample label size:  torch.Size([50])
Sample label:
 tensor([0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1,
        1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0,
        0, 0], dtype=torch.int32)
```

```
0, 0], dtype=torch.int32)
```

Here, 50 is the batch size and 200 is the sequence length that we have defined. Now our data prep step is complete and next we will look at the LSTM network architecture for start building our model

13) Define the LSTM Network Architecture



LSTM Architecture for Sentiment Analysis

The layers are as follows:

0. Tokenize : This is not a layer for LSTM network but a mandatory step of converting our words into tokens (integers)

1. Embedding Layer: that converts our word tokens (integers) into embedding of specific size
2. LSTM Layer: defined by hidden state dims and number of layers
3. Fully Connected Layer: that maps output of LSTM layer to a desired output size
4. Sigmoid Activation Layer: that turns all output values in a value between 0 and 1
5. Output: Sigmoid output from the last timestep is considered as the final output of this network

14) Define the Model Class

```
import torch.nn as nn

class SentimentLSTM(nn.Module):
    """
    The RNN model that will be used to perform Sentiment analysis.
    """

    def __init__(self, vocab_size, output_size, embedding_dim, hidden_dim, n_layers, drop_prob=0
        """
        Initialize the model by setting up the layers.
        """
        super().__init__()

        self.output_size = output_size
        self.n_layers = n_layers
        self.hidden_dim = hidden_dim

        # embedding and LSTM layers
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, n_layers,
                            dropout=drop_prob, batch_first=True)

        # dropout layer
        self.dropout = nn.Dropout(0.3)

        # linear and sigmoid layers
        self.fc = nn.Linear(hidden_dim, output_size)
        self.sig = nn.Sigmoid()


    def forward(self, x, hidden):
        """
        Perform a forward pass of our model on some input and hidden state.
        """
        batch_size = x.size(0)

        # embeddings and lstm_out
        embeds = self.embedding(x)
        lstm_out, hidden = self.lstm(embeds, hidden)

        # stack up lstm outputs
        lstm_out = lstm_out.contiguous().view(-1, self.hidden_dim)

        # dropout and fully-connected layer
        out = self.dropout(lstm_out)
        out = self.fc(out)
        # sigmoid function
        sig_out = self.sig(out)

        # reshape to be batch_size first
        sig_out = sig_out.view(batch_size, -1)
        sig_out = sig_out[:, -1] # get last batch of labels

        # return last sigmoid output and hidden state
        return sig_out, hidden


    def init_hidden(self, batch_size):
        ''' Initializes hidden state '''
        # Create two new tensors with sizes n_layers x batch_size x hidden_dim,
```

```
        # initialized to zero, for hidden state and cell state of LSTM
        weight = next(self.parameters()).data

        if (train_on_gpu):
            hidden = (weight.new(self.n_layers, batch_size, self.hidden_dim).zero_().cuda(),
                  weight.new(self.n_layers, batch_size, self.hidden_dim).zero_().cuda())
        else:
            hidden = (weight.new(self.n_layers, batch_size, self.hidden_dim).zero_(),
```

```
# Instantiate the model w/ hyperparams
vocab_size = len(vocab_to_int)+1 # +1 for the 0 padding
output_size = 1
embedding_dim = 400
hidden_dim = 256
n_layers = 2

net = SentimentLSTM(vocab_size, output_size, embedding_dim,
hidden_dim, n_layers)

print(net)

SentimentLSTM(
  (embedding): Embedding(74073, 400)
  (lstm): LSTM(400, 256, num_layers=2, batch_first=True, dropout=0.5)
  (dropout): Dropout(p=0.3)
  (fc): Linear(in_features=256, out_features=1, bias=True)
  (sig): Sigmoid()
)
```

- Training Loop

Most of the code in training loop is pretty standard Deep Learning training code that you might see often in all the implementations that's using PyTorch framework.

```
# loss and optimization functions
lr=0.001

criterion = nn.BCELoss()
optimizer = torch.optim.Adam(net.parameters(), lr=lr)


# training params

epochs = 4 # 3-4 is approx where I noticed the validation loss stop decreasing

counter = 0
print_every = 100
clip=5 # gradient clipping

# move model to GPU, if available
if(train_on_gpu):
```

```
    net.cuda()

net.train()
# train for some number of epochs
for e in range(epochs):
    # initialize hidden state
    h = net.init_hidden(batch_size)

    # batch loop
    for inputs, labels in train_loader:
        counter += 1

        if(train_on_gpu):
            inputs, labels = inputs.cuda(), labels.cuda()

        # Creating new variables for the hidden state, otherwise
        # we'd backprop through the entire training history
        h = tuple([each.data for each in h])

        # zero accumulated gradients
        net.zero_grad()

        # get the output from the model
        inputs = inputs.type(torch.LongTensor)
        output, h = net(inputs, h)

        # calculate the loss and perform backprop
        loss = criterion(output.squeeze(), labels.float())
        loss.backward()
        # `clip_grad_norm` helps prevent the exploding gradient problem in RNNs / LSTMs.
        nn.utils.clip_grad_norm_(net.parameters(), clip)
        optimizer.step()

        # loss stats
        if counter % print_every == 0:
            # Get validation loss
            val_h = net.init_hidden(batch_size)
            val_losses = []
            net.eval()
            for inputs, labels in valid_loader:

                # Creating new variables for the hidden state, otherwise
                # we'd backprop through the entire training history
                val_h = tuple([each.data for each in val_h])

                if(train_on_gpu):
                    inputs, labels = inputs.cuda(), labels.cuda()

                inputs = inputs.type(torch.LongTensor)
                output, val_h = net(inputs, val_h)
                val_loss = criterion(output.squeeze(), labels.float())

                val_losses.append(val_loss.item())

            net.train()
```

- On Test Data

```
# Get test data loss and accuracy

test_losses = [] # track loss
num_correct = 0

# init hidden state
h = net.init_hidden(batch_size)

net.eval()
# iterate over test data
for inputs, labels in test_loader:

    # Creating new variables for the hidden state, otherwise
    # we'd backprop through the entire training history
    h = tuple([each.data for each in h])

    if(train_on_gpu):
        inputs, labels = inputs.cuda(), labels.cuda()

    # get predicted outputs
    inputs = inputs.type(torch.LongTensor)
    output, h = net(inputs, h)

    # calculate loss
    test_loss = criterion(output.squeeze(), labels.float())
    test_losses.append(test_loss.item())

    # convert output probabilities to predicted class (0 or 1)
    pred = torch.round(output.squeeze())  # rounds to the nearest integer

    # compare predictions to true label
    correct_tensor = pred.eq(labels.float().view_as(pred))
    correct = np.squeeze(correct_tensor.numpy()) if not train_on_gpu else np.squeeze(correct_ten
    num_correct += np.sum(correct)


# -- stats! -- ##
# avg test loss
print("Test loss: {:.3f}".format(np.mean(test_losses)))

# accuracy over all test data
test_acc = num_correct/len(test_loader.dataset)
```

First, we will define a `tokenize` function that will take care of pre-processing steps and then we will create a `predict` function that will give us the final output after parsing the user provided review.

```
from string import punctuation

def tokenize_review(test_review):
    test_review = test_review.lower() # lowercase
    # get rid of punctuation
    test_text = ''.join([c for c in test_review if c not in punctuation])

    # splitting by spaces
    test_words = test_text.split()

```

```
    # tokens
    test_ints = []
    test_ints.append([vocab_to_int[word] for word in test_words])

    return test_ints

# test code and generate tokenized review
test_ints = tokenize_review(test_review_neg)
print(test_ints)


# test sequence padding
seq_length=200
features = pad_features(test_ints, seq_length)

print(features)


# test conversion to tensor and pass into your model
feature_tensor = torch.from_numpy(features)
print(feature_tensor.size())


def predict(net, test_review, sequence_length=200):

    net.eval()

    # tokenize review
    test_ints = tokenize_review(test_review)

    # pad tokenized sequence
    seq_length=sequence_length
    features = pad_features(test_ints, seq_length)

    # convert to tensor to pass into your model
    feature_tensor = torch.from_numpy(features)

    batch_size = feature_tensor.size(0)

    # initialize hidden state
    h = net.init_hidden(batch_size)

    if(train_on_gpu):
        feature_tensor = feature_tensor.cuda()

    # get the output from the model
    output, h = net(feature_tensor, h)

    # convert output probabilities to predicted class (0 or 1)
    pred = torch.round(output.squeeze())
    # printing output value, before rounding
    print('Prediction value, pre-rounding: {:.6f}'.format(output.item()))

    # print custom response
    if(pred.item()==1):
```

```
test_review = 'This movie had the best acting and the dialogue was so
good. I loved it.'

seq_length=200 # good to use the length that was trained on

predict(net, test_review_neg, seq_length)
```

Positive review detected

Closing Remarks:

- I have tried to detail out the process invovled in building a Sentiment Analysis classifier based on LSTM architecture using PyTorch framework.
- Please feel free to write your thoughts / suggestions / feedbacks

Update: Another article to give you a microscopic view of what happens within the layers. Read here

Machine Learning | NLP | Sentiment Analysis | Deep Learning | Lstm