<u>Sentiment Analysis</u>

I will be using "Recurrent Neural Networks" to detect the sentiments from the user comments.

I have used RNN over Feedforward network because it gives more accurate result because through RNN we are able to keep information about the sequence of words.

> **<u>Dataset</u>** used is of **Movie reviews** for training along with labels positive or negative.

Example:

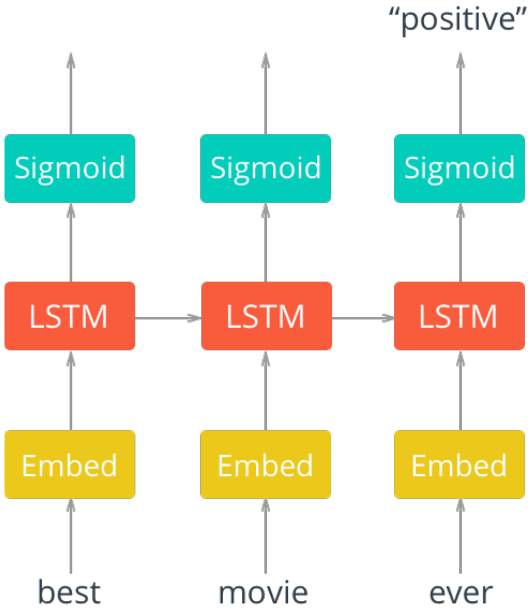
- **Data:** "The movie was boring by the end."

  Label: "Negative"
- **Data:** "Very Good Plot."

  Label: "Positive"

Network Architecture

The architecture for this network is shown below.



- **First, we'll pass in words to an embedding layer.** We need an embedding layer because we have tens of thousands of words, so we'll need a more efficient representation for our input data than one-hot encoded vectors.
- **After input words are passed to an embedding layer**, the new embeddings will be passed to LSTM cells. The LSTM cells will add recurrent connections to the network and give us the ability to include information about the sequence of words in the movie review data.
- **Finally, the LSTM outputs will go to a sigmoid output layer**. We're using a sigmoid function because positive and negative = 1 and 0, respectively, and a sigmoid will output predicted, sentiment values between 0-1.

As last step we calculate the loss by comparing the output at the last time step and training label (pos/neg).

- **Firstly, we have mounted drive to connect take our dataset**

In [1]:

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call driv
e.mount("/content/drive", force_remount=True).
```

In [2]:

```
!pip install -q keras
```

In [3]:

```
import keras
```

- **Loading the data**

In [4]:

```
import numpy as np

with open('/content/drive/MyDrive/Colab Notebooks/data/reviews.txt', 'r') as f:
    reviews = f.read()
with open('/content/drive/MyDrive/Colab Notebooks/data/labels.txt', 'r') as f:
    labels = f.read()
```

- **Visualizing a small set of data to check**

In [5]:

```
print(reviews[:1000])
print()
print(labels[:27])
```

```
bromwell high is a cartoon comedy . it ran at the same time as some other programs
about school life  such as  teachers  . my   years in the teaching profession lead
me to believe that bromwell high  s satire is much closer to reality than is  teac
hers  . the scramble to survive financially  the insightful students who can see r
ight through their pathetic teachers  pomp  the pettiness of the whole situation
all remind me of the schools i knew and their students . when i saw the episode in
which a student repeatedly tried to burn down the school  i immediately recalled
. . . . . . . . . at . . . . . . . . . . . high . a classic line inspector i  m her
e to sack one of your teachers . student welcome to bromwell high . i expect that
many adults of my age think that bromwell high is far fetched . what a pity that i
t isn  t
story of a man who has unnatural feelings for a pig . starts out with a opening sc
ene that is a terrific example of absurd comedy . a formal orchestra audience is t
urn

positive
negative
positive
```

- ## **Data pre-processing**

Now we have to do data preprocessing, before moving any further We need to have our data in proper form before passing to network.

We will have to send the data to embedding layer so we need to encode each word to an integer first. Also we are cleaning some data i.e. removing punctuations The first step when building a neural network model is getting your data into the proper form to feed into the network. Since we're using embedding layers, we'll need to encode each word with an integer. We'll also want to clean it up a bit.

In [6]:

```
from string import punctuation

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

# spliting data by new lines and spaces
reviews_split = all_text.split('\n')
all_text = ' '.join(reviews_split)

# creating a list of words
words = all_text.split()
```

In [7]:

```
words[:30]
```

Out[7]:

```
['bromwell',
 'high',
 'is',
 'a',
 'cartoon',
 'comedy',
 'it',
 'ran',
 'at',
 'the',
 'same',
 'time',
 'as',
 'some',
 'other',
 'programs',
 'about',
 'school',
 'life',
 'such',
 'as',
 'teachers',
 'my',
 'years',
 'in',
 'the',
 'teaching',
 'profession',
 'lead',
 'me']
```

- **### Encoding the words**

Now we have to convert them to integers, so for that we will be creating a dictionary that will map words in the vocabulary to the integers.

In [8]:

```
from collections import Counter

## Building a dictionary that maps words to integers
counts = Counter(words)
vocab = sorted(counts, key=counts.get, reverse=True)
vocab_to_int = {word: ii for ii, word in enumerate(vocab, 1)}
```

```
## using the dictionary to tokenize each review in reviews_split
## storing the tokenized reviews in reviews_ints
reviews_ints = []
for review in reviews_split:
    reviews_ints.append([vocab_to_int[word] for word in review.split()])
```

- **Testing the code so far**

In [9]:

```
# stats about vocabulary
print('Unique words: ', len((vocab_to_int)))  # should ~ 74000+
print()

# printing to check tokens in first review
print('Tokenized review: \n', reviews_ints[:1])
```

```
Unique words:  74072

Tokenized review:
 [[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, 149
9, 36, 51, 66, 204, 145, 67, 1199, 5194, 19869, 1, 37442, 4, 1, 221, 883, 31, 2988
, 71, 4, 1, 5787, 10, 686, 2, 67, 1499, 54, 10, 216, 1, 383, 9, 62, 3, 1406, 3686,
783, 5, 3483, 180, 1, 382, 10, 1212, 13583, 32, 308, 3, 349, 341, 2913, 10, 143, 1
27, 5, 7690, 30, 4, 129, 5194, 1406, 2326, 5, 21025, 308, 10, 528, 12, 109, 1448,
4, 60, 543, 102, 12, 21025, 308, 6, 227, 4146, 48, 3, 2211, 12, 8, 215, 23]]
```

- **### Encoding the labels**

Our labels are "positive" or "negative". To use these labels in our network, we have converted them to 0 and 1.

In [10]:

```
# 1=positive, 0=negative label conversion
labels_split = labels.split('\n')
encoded_labels = np.array([1 if label == 'positive' else 0 for label in labels_sp
lit])
```

- **### Removing Outliers**

Also we have set our reviews to a specific length:

1. **Getting rid of extremely long or short reviews i.e. the outliers**
2. **Padding/truncating the remaining data so that we have reviews of the same length.**

In [11]:

```
# outlier review stats
review_lens = Counter([len(x) for x in reviews_ints])
print("Zero-length reviews: {}".format(review_lens[0]))
print("Maximum review length: {}".format(max(review_lens)))
```

```
Zero-length reviews: 1
Maximum review length: 2514
```

In [12]:

```
print('Number of reviews before removing outliers: ', len(reviews_ints))

## removing any reviews/labels with zero length from the reviews_ints list.
```

```
# getting indices of any reviews with length 0
non_zero_idx = [ii for ii, review in enumerate(reviews_ints) if len(review) != 0]

# removing all 0-length reviews and their labels
reviews_ints = [reviews_ints[ii] for ii in non_zero_idx]
encoded_labels = np.array([encoded_labels[ii] for ii in non_zero_idx])

print('Number of reviews after removing outliers: ', len(reviews_ints))
```

```
Number of reviews before removing outliers:  25001
Number of reviews after removing outliers:  25000
```

- **## Padding sequences**

To deal with both short and very long reviews, we'll pad or truncate all our reviews to a specific length.

As a small example, if the `seq_length=10` and an input review is:

```
[27, 30, 128]
```

The resultant, padded sequence should be:

```
[0, 0, 0, 0, 0, 0, 0, 27, 30, 128]
```

The final `features` array will be a 2D array, with as many rows as there are reviews, and as many columns as the specified `seq_length`.

In [13]:

```
def pad_features(reviews_ints, seq_length):

    # getting the correct rows x cols shape
    features = np.zeros((len(reviews_ints), seq_length), dtype=int)

    # for each review, I grab that review
    for i, row in enumerate(reviews_ints):
        features[i, -len(row):] = np.array(row)[:seq_length]

    return features
```

In [14]:

```
# Testing the implementation!

seq_length = 200

features = pad_features(reviews_ints, seq_length=seq_length)

assert len(features)==len(reviews_ints), "features should have as many rows as re
views."
assert len(features[0])==seq_length, "Each feature row should contain seq_length
values."

# print first 10 values of the first 30 batches
print(features[:30,:10])
```

```
[[     0      0      0      0      0      0      0      0      0      0]
 [     0      0      0      0      0      0      0      0      0      0]
 [22382     42  46418     15    706  17139   3389     47     77     35]
 [ 4505    505     15      3   3342    162   8312   1652      6   4819]
 [     0      0      0      0      0      0      0      0      0      0]
 [     0      0      0      0      0      0      0      0      0      0]
 [     0      0      0      0      0      0      0      0      0      0]
 [     0      0      0      0      0      0      0      0      0      0]
 [     0      0      0      0      0      0      0      0      0      0]
```

```
 [   54    10    14   116    60   798   552    71   364     5]
 [    0     0     0     0     0     0     0     0     0     0]
 [    0     0     0     0     0     0     0     0     0     0]
 [    0     0     0     0     0     0     0     0     0     0]
 [    1   330   578    34     3   162   748  2731     9   325]
 [    9    11 10171  5305  1946   689   444    22   280   673]
 [    0     0     0     0     0     0     0     0     0     0]
 [    1   307 10399  2069  1565  6202  6528  3288 17946 10628]
 [    0     0     0     0     0     0     0     0     0     0]
 [   21   122  2069  1565   515  8181    88     6  1325  1182]
 [    1    20     6    76    40     6    58    81    95     5]
 [   54    10    84   329 26230 46427    63    10    14   614]
 [   11    20     6    30  1436 32317  3769   690 15100     6]
 [    0     0     0     0     0     0     0     0     0     0]
 [    0     0     0     0     0     0     0     0     0     0]
 [   40    26   109 17952  1422     9     1   327     4   125]
 [    0     0     0     0     0     0     0     0     0     0]
 [   10   499     1   307 10399    55    74     8    13    30]
 [    0     0     0     0     0     0     0     0     0     0]
 [    0     0     0     0     0     0     0     0     0     0]
 [    0     0     0     0     0     0     0     0     0     0]]
```

- **## Training, Validation, Test**

Now we have split our data into training, validation, and test sets in ratio of 0.8, 0.1, 0.1 respectively

In [15]:

```
split_frac = 0.8

## split data into training, validation, and test data (features and labels, x an
d y)

split_idx = int(len(features)*split_frac)
train_x, remaining_x = features[:split_idx], features[split_idx:]
train_y, remaining_y = encoded_labels[:split_idx], encoded_labels[split_idx:]

test_idx = int(len(remaining_x)*0.5)
val_x, test_x = remaining_x[:test_idx], remaining_x[test_idx:]
val_y, test_y = remaining_y[:test_idx], remaining_y[test_idx:]

## print out the shapes of your resultant feature data
print("\t\t\tFeature Shapes:")
print("Train set: \t\t{}".format(train_x.shape),
      "\nValidation set: \t{}".format(val_x.shape),
      "\nTest set: \t\t{}".format(test_x.shape))
```

```
			Feature Shapes:
Train set: 		(20000, 200) 
Validation set: 	(2500, 200) 
Test set: 		(2500, 200)
```

- **## DataLoaders and Batching**

Now we can create dataloaders and batch our training, validation, and test Tensor datasets.

In [16]:

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

# create Tensor datasets
train_data = TensorDataset(torch.from_numpy(train_x), torch.from_numpy(train_y))
```

```
valid_data = TensorDataset(torch.from_numpy(val_x), torch.from_numpy(val_y))
test_data = TensorDataset(torch.from_numpy(test_x), torch.from_numpy(test_y))

# dataloaders
batch_size = 50

#  SHUFFLING the training 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 [17]:

```
# taking 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([[    0,     0,     0,  ...,   191,     3, 14065],
        [    0,     0,     0,  ...,    18,   262,    45],
        [    0,     0,     0,  ...,    53,    13,    50],
        ...,
        [    0,     0,     0,  ...,   433,     4,   273],
        [ 1635,   607,    99,  ...,     9,     1,    18],
        [   11,    14,    60,  ...,  2741,   335,  8626]])

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

- **# Sentiment Network with PyTorch**

Now we have defined our network below

- **Firstly checking for the availability of GPU**

In [18]:

```
# checking for GPU
train_on_gpu=torch.cuda.is_available()

if(train_on_gpu):
    print('Training on GPU.')
else:
    print('No GPU available, training on CPU.')
```

```
Training on GPU.
```

In [19]:

```
import torch.nn as nn

class SentimentRNN(nn.Module):
```

```
    def __init__(self, vocab_size, output_size, embedding_dim, hidden_dim, n_laye
rs, drop_prob=0.5):

        # Initializing the model by setting up the layers.

        super(SentimentRNN, self).__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
        x = x.long()
        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):
        # Initializing hidden state
        # Creating 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
_(),
                      weight.new(self.n_layers, batch_size, self.hidden_dim).zero
```

```
_())

        return hidden
```

- ## Instantiate the network Now we have set the hyperparameters and instantiated the model

In [20]:

```
# Instantiating the model with hyperparams
vocab_size = len(vocab_to_int)+1      # +1 for the 0 padding + our word tokens
output_size = 1
embedding_dim = 400
hidden_dim = 256
n_layers = 2

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

print(net)
```

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

- ## Training

In [21]:

```
# loss and optimization functions
lr=0.001

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

In [22]:

```
# training params

epochs = 4     # checked with different epochs but here between 3 - 4 the validat
ion loss stop decreasing
counter = 0
print_every = 100
clip=5   # gradient clipping

# moving model to GPU, if available
if(train_on_gpu):
    net.cuda()

net.train()
# train for epochs
for e in range(epochs):
    # initializing the 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
        output, h = net(inputs, h)

        # calculating 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 statistics
        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()

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

                val_losses.append(val_loss.item())

            net.train()
            print("Epoch: {}/{}...".format(e+1, epochs),
                  "Step: {}...".format(counter),
                  "Loss: {:.6f}...".format(loss.item()),
                  "Val Loss: {:.6f}".format(np.mean(val_losses)))
```

```
Epoch: 1/4... Step: 100... Loss: 0.600990... Val Loss: 0.654218
Epoch: 1/4... Step: 200... Loss: 0.619914... Val Loss: 0.629372
Epoch: 1/4... Step: 300... Loss: 0.692550... Val Loss: 0.692442
Epoch: 1/4... Step: 400... Loss: 0.688580... Val Loss: 0.691582
Epoch: 2/4... Step: 500... Loss: 0.699060... Val Loss: 0.702534
Epoch: 2/4... Step: 600... Loss: 0.690918... Val Loss: 0.688758
Epoch: 2/4... Step: 700... Loss: 0.610046... Val Loss: 0.565808
Epoch: 2/4... Step: 800... Loss: 0.511591... Val Loss: 0.495491
Epoch: 3/4... Step: 900... Loss: 0.399440... Val Loss: 0.517709
Epoch: 3/4... Step: 1000... Loss: 0.397885... Val Loss: 0.443151
Epoch: 3/4... Step: 1100... Loss: 0.339887... Val Loss: 0.480243
Epoch: 3/4... Step: 1200... Loss: 0.249359... Val Loss: 0.437880
Epoch: 4/4... Step: 1300... Loss: 0.447809... Val Loss: 0.561570
Epoch: 4/4... Step: 1400... Loss: 0.344228... Val Loss: 0.448189
Epoch: 4/4... Step: 1500... Loss: 0.373095... Val Loss: 0.535112
Epoch: 4/4... Step: 1600... Loss: 0.337592... Val Loss: 0.435691
```

- **## Testing**

In [23]:

```
# Get test data loss and accuracy

test_losses = [] # track loss
num_correct = 0

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

net.eval()
# iterating 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
    output, h = net(inputs, h)

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

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

    # comparing 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.sque
eze(correct_tensor.cpu().numpy())
    num_correct += np.sum(correct)


# 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)
print("Test accuracy: {:.3f}".format(test_acc))
```

```
Test loss: 0.432
Test accuracy: 0.807
```

The model gave accuracy of 80.7% on the test data.

- **### Inference on a test review Now we have checked with a different user defined review which is not from the dataset.**

In [24]:

```
# negative test review
test_review_neg = 'The worst movie I have seen; acting was terrible and I want my
money back. This movie had bad acting and the dialogue was slow.'
```

In [25]:

```
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)
```

```
[[1, 247, 18, 10, 28, 108, 113, 14, 388, 2, 10, 181, 60, 273, 144, 11, 18, 68, 76,
113, 2, 1, 410, 14, 539]]
```

In [26]:

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

print(features)
```

```
[[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   1 247  18  10  28
  108 113  14 388   2  10 181  60 273 144  11  18  68  76 113   2   1 410
   14 539]]
```

In [27]:

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

```
torch.Size([1, 200])
```

In [28]:

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

    net.eval()

    # tokenize the review
    test_ints = tokenize_review(test_review)

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

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

    batch_size = feature_tensor.size(0)

    # initializing hidden state
    h = net.init_hidden(batch_size)
```

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

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

    # convertting 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()))

    # printing custom response
    if(pred.item()==1):
        print("Positive review detected!")
    else:
        print("Negative review detected.")
```

In [29]:

```python
# positive test review
test_review_pos = 'This movie had the best acting and the dialogue was so good. I loved it.'
```

In [30]:

```python
# calling function
seq_length=200     # used the length that was trained on

predict(net, test_review_neg, seq_length)
```

```
Prediction value, pre-rounding: 0.020388
Negative review detected.
```

In [31]:

```python
seq_length=200     # used the length that was trained on

predict(net, test_review_pos, seq_length)
```

```
Prediction value, pre-rounding: 0.939066
Positive review detected!
```

The model rightly predicts the reviews as positive or negative!