Nlp

IE 534 Homework 4: Sentiment Analysis for IMDB Movie Reviews

This assignment will work with the Large Movie Review Dataset and provide an understanding of how Deep Learning is used within the field of Natural Language Processing (NLP). The original paper published in 2011 discussing the dataset and a wide variety of techniques can be found here. We will train models to detect the sentiment of written text. More specifically, we will try to feed a movie review into a model and have it predict whether it is a positive or negative movie review.

The assignment starts off with how to preprocess the dataset into a more appropriate format followed by three main parts. Each part will involve training different types of models and will detail what you need to do as well as what needs to be turned in. The assignment is meant to provide you with a working knowledge of how to use neural networks with sequential data.

- Part one deals with training basic Bag of Words models.
- Part two will start incoporating temporal information by using LSTM layers.
- Part three will show how to train a language model and how doing this as a first step can sometimes improve results for other tasks

Make a new directory for yourself on BlueWaters. Do everything within this directory and this will ultimately contain most of what you need to turn in.

I'm not directly providing you with any finished python files. This is written more like a tutorial with excerpts of code scattered throughout. You will have to take these pieces of code, combine them, edit them, add to them, etc. and figure out how to run them on BlueWaters. You will also need to change a variety of hyperparameters and run them multiple times.

What to Turn In

Make a copy of the google sheet here. This is how you should summarize your results. Write a sentence or two for each model you train in parts 1a, 1b, 2a, 2b, and 3c. For part 3a, the assignment only requires you to train the model with the hyperparamters I give you. For part 3b, you will generate fake movie reviews (I've included a fake movie review I generated as an example).

Zip all of your code (excluding the dataset/preprocessed data). Save your completed google sheet as a PDF. Submit the code and PDF on compass.

Some Basic Terminology

One-hot vector - Mathematically speaking, a one-hot vector is a vector of all zeros except for one element containing a one. There are many applications but they are of particular importance in natural language processing. It is a method of encoding categorical variables (such as words or characters) as numerical vectors which become the input to computational models.

Suppose we want to train a model to predict the sentiment of the following two sentences: "This movie is bad" and "This movie is good". We need a way of mathematically representing these as inputs to the model. First we define a dictionary (or vocabulary). This contains every word contained within the dataset.

```
vocab = ['this','movie','is','bad','good']
```

We can also add an “unknown token” to the dictionary in case words not present in the training dataset show up during testing.

```
unknown = ''
vocab = [unknown,'this','movie','is','bad','good']
```

Each word can now be represented as vector of length *|vocab|* with a 1 in the location corresponding to the word’s location within the dictionary.

```
[0,1,0,0,0,0] # this
[0,0,1,0,0,0] # movie
[0,0,0,1,0,0] # is
[0,0,0,0,0,1] # good
```

There are many ways this data can be used. This assignment deals with two possibilities. These vectors could be processed one at a time (which is the basis of sections 2 and 3 of this assignment) by using a recurrent neural network or the full review could be processed all at once using a bag of words representation (section 1 of this assignment). The bag of words representation simply sums up the one-hot vectors and has a single vector input for each review. This is convenient for reducing the input size and making variable length reviews fixed length (all reviews become a single vector of length *|vocab|*) but the input no longer contains sequence information. The order in which the words appear has no effect on a document’s bag of words representation.

```
x1=[0,1,1,1,0,1] # bag of words for "this movie is good"
x2=[0,1,1,1,1,0] # bag of words for "this movie is bad"
```

A basic classifier would be to multiply the bag of words representation *x* by some weight vector *W* and declare the review to be positive if the output is greater than 0 and negative otherwise. Training this model on these two sentences might yield a weight vector *W=[0.0,0.0,0.0,0.0,+0.5,-0.5]*. Notice this model simply classifies any review with the word “good” as positive and any review with the word “bad” as negative.

```
# model would successfully classify the unseen test sequence as negative
x=[7,0,0,0,1,0] # bag of words for "I have never seen such a bad film"


# model would incorrectly classify the unseen test sequence as negative
x=[10,1,1,1,1,0] # bag of words for "This movie is not bad. In fact, I think it's great." Not
```

Word Embedding - The one-hot encoding is useful but notice the input size depends heavily on the vocabulary size which can cause problems considering there are millions of tokens in the English language. Additionally, many of these tokens are related to eachother in some way (like the words “good” and “great”) and this relationship is not captured by the one-hot encoding. Word embeddings are another vector representation of fixed length (for example, length 300 is common) where the elements are real valued as opposed to just 1 or 0. Typically these word embeddings are generated from a language model which is the basis of part 3 of this assignment. Word2vec and GloVe are examples of word embeddings.

Preprocessing the Data

The IMDB dataset is located within the class directory under '/projects/training/bauh/NLP/aclImdb/'. Each review is contained in a separate .txt file and organized into separate directories depending on if they're positive/negative or part of the train/test/unlabeled splits. It's easier to re-work the data before training any models for multiple reasons. * It's faster to combine everything into fewer files. * We need to tokenize the dataset (split long strings up into individual words/symbols). * We need to know how many unique tokens there are to decide on our vocabulary/model size. * We don't want to constantly be converting strings to token IDs within the training loop (takes non-negligible amount of time). * Storing everything as token IDs directly leads to the 1-hot representation necessary for input into our model.

The NLTK (Natural Language Toolkit) python package contains a tokenizer that we will use and needs to be installed.

```
pip install --user nltk
```

Within your assignment directory, create a file called **preprocess_data.py**.

```
import numpy as np
import os
import nltk
import itertools

## create directory to store preprocessed data
if(not os.path.isdir('preprocessed_data')):
    os.mkdir('preprocessed_data')
```

These few lines of code written above import a few packages and creates a new directory if it's not already made.

```
## get all of the training reviews (including unlabeled reviews)
train_directory = '/projects/training/bauh/NLP/aclImdb/train/'

pos_filenames = os.listdir(train_directory + 'pos/')
neg_filenames = os.listdir(train_directory + 'neg/')
unsup_filenames = os.listdir(train_directory + 'unsup/')

pos_filenames = [train_directory+'pos/'+filename for filename in pos_filenames]
neg_filenames = [train_directory+'neg/'+filename for filename in neg_filenames]
unsup_filenames = [train_directory+'unsup/'+filename for filename in unsup_filenames]

filenames = pos_filenames + neg_filenames + unsup_filenames

count = 0
x_train = []
for filename in filenames:
    with open(filename,'r',encoding='utf-8') as f:
        line = f.readlines()[0]
    line = line.replace('<br />',' ')
    line = line.replace('\x96',' ')
    line = nltk.word_tokenize(line)
    line = [w.lower() for w in line]

    x_train.append(line)
    count += 1
    print(count)
```

The top part of the code simply results in a list of every text file containing a movie review in the variable **filenames**. The first 12500 are positive reviews, the next 12500 are negative reviews, and the remaining are unlabeled reviews. For each review, we remove text we don't want (' ' and '\x96'), tokenize the string using the nltk package, and make everything lowercase. Here is an example movie review before and after tokenization:

```
a star. And one last, heretical note: Those mountains do look gorgeous in color.<br /><br />"
```

```
'heretical', 'note', ':', 'those', 'mountains', 'do', 'look', 'gorgeous', 'in', 'color', '.']
```

Notice how symbols like periods, parenthesis, etc. become their own tokens and it splits up contractions ("it's" becomes "it" and "'s"). It's not perfect and can be ruined by typos or lack of punctuation but works for the most part. We now have a list of tokens for every review in the training dataset in the variable **x_train**. We can do the same thing for the test dataset and the variable **x_test**.

```
## get all of the test reviews
test_directory = '/projects/training/bauh/NLP/aclImdb/test/'

pos_filenames = os.listdir(test_directory + 'pos/')
neg_filenames = os.listdir(test_directory + 'neg/')

pos_filenames = [test_directory+'pos/'+filename for filename in pos_filenames]
neg_filenames = [test_directory+'neg/'+filename for filename in neg_filenames]

filenames = pos_filenames+neg_filenames

count = 0
x_test = []
for filename in filenames:
    with open(filename,'r',encoding='utf-8') as f:
        line = f.readlines()[0]
    line = line.replace('<br />',' ')
    line = line.replace('\x96',' ')
    line = nltk.word_tokenize(line)
    line = [w.lower() for w in line]

    x_test.append(line)
    count += 1
    print(count)
```

We can use the following to get a basic understanding what the IMDB dataset contains.

```
## number of tokens per review
no_of_tokens = []
for tokens in x_train:
    no_of_tokens.append(len(tokens))
no_of_tokens = np.asarray(no_of_tokens)
print('Total: ', np.sum(no_of_tokens), ' Min: ', np.min(no_of_tokens), ' Max: ', np.max(no_of_
```

Total: 20090526 Min: 10 Max: 2859 Mean: 267.87368 Std: 198.540647165

The mean review contains ~267 tokens with a standard deviation of ~200. Although there are over 20 million total tokens, they're obviously not all unique. We now want to build our dictionary/vocabulary.

```
### word_to_id and id_to_word. associate an id to every unique token in the training data
all_tokens = itertools.chain.from_iterable(x_train)
```

```
word_to_id = {token: idx for idx, token in enumerate(set(all_tokens))}

all_tokens = itertools.chain.from_iterable(x_train)
id_to_word = [token for idx, token in enumerate(set(all_tokens))]
id_to_word = np.asarray(id_to_word)
```

We have two more variables: **word_to_id** and **id_to_word**. You can access **id_to_word[index]** to find out the **token** for a given **index** or access **word_to_id[token]** to find out the **index** for a given **token**. They'll both be used later on. We can also see there are roughly 200,000 unique tokens with **len(id_to_word)**. Realistically, we don't want to use all 200k unique tokens. An embedding layer with 200k tokens and 500 outputs has 100 million weights which is far too many considering the training dataset only has 20 million tokens total. Additionally, the tokenizer sometimes creates a unique token that is actually just a combination of other tokens because of typos and other odd things. Here are a few examples from **id_to_word** ("majesty.these", "producer/director/star", "1¢")

We should organize the tokens by their frequency in the dataset to give us a better idea of choosing our vocabulary size.

```
## let's sort the indices by word frequency instead of random
x_train_token_ids = [[word_to_id[token] for token in x] for x in x_train]
count = np.zeros(id_to_word.shape)
for x in x_train_token_ids:
    for token in x:
        count[token] += 1
indices = np.argsort(-count)
id_to_word = id_to_word[indices]
count = count[indices]

hist = np.histogram(count,bins=[1,10,100,1000,10000])
print(hist)
for i in range(10):
    print(id_to_word[i],count[i])
```

Notice we used **word_to_id** to convert all of the tokens for each review into a sequence of token IDs instead (this is ultimately what we are going for but we want the IDs to be in a different order). We then accumulate the total number of occurrences for each word in the array **count**. Finally, this is used to sort the vocabulary list. We are left with **id_to_word** in order from most frequent tokens to the least frequent, **word_to_id** gives us the index for each token, and count simply contains the number of occurrences for each token.

```
(array([164036,  26990,   7825,   1367]), array([    1,    10,   100,  1000, 10000]))
the 1009387
, 829574
. 819080
and 491830
a 488305
of 438545
to 405661
is 330904
it 285710
in 280618
```

The histogram output gives us a better understanding of the actual dataset. Over 80% (~160k) of the unique tokens occur between 1 and 10 times while only ~5% occur more than 100 times each. Using **np.sum(count[0:100])** tells us over half of all of the 20 million tokens are the most common 100 words and **np.sum(count[0:8000])** tells us almost 95% of the dataset is contained within the most common 8000 words.

```
## assign -1 if token doesn't appear in our dictionary
## add +1 to all token ids, we went to reserve id=0 for an unknown token
x_train_token_ids = [[word_to_id.get(token,-1)+1 for token in x] for x in x_train]
x_test_token_ids = [[word_to_id.get(token,-1)+1 for token in x] for x in x_test]
```

Here is where we convert everything to the exact format we want for training purposes. Notice the test dataset may have unique tokens our model has never seen before. We can anticipate this ahead of time by actually reserving **index 0** for an **unknown** token. This is why I assign a -1 if the token isn't part of **word_to_id** and add +1 to every id. Just be aware of this that **id_to_word** is now off by 1 index if you actually want to convert ids to words.

We could also decide later on if we only want to use a vocabulary size of 8000 for training and just assign any other token ID in the training data to **0**. This way it can develop its own embedding for **unknown** tokens which can help out when it inevitably sees **unknown** tokens during testing.

Here is the same review posted earlier. One uses the full dictionary and the other replaces the less frequent tokens with the unknown token:

```
[   727   1203    844     85  23503      8    156      2    351     17
     32  50333   9855   2299      2      4     13  21676 159129  26371
      3     44     30   4381      4    819      2   3430      4    212
   6093      2      4     66      1  26454  24093 184660      3      1
     73    162    957      8      5     58    296     18  12511  11708
   5009     14    337  10898   3188    164    313    215  50604      5
    372     48      5   2147      2      4    397     44     17    637
  12385      5  25011     48    119    204      1   7427     43     13
     10      1    214      3     23    304      6   5009     80     92
   5002 185207      2     18   5281   2982     80     92    668   6463
      2     18  12795  10997   5805   6830      2      4     44     27
    204    155      1    893      3     22      1     73    634      8
     13      2     12     68      2    304      6   8241      2      9
     14     10 112033    122   1056      2     46     30    157     42
     63   1558     12     41   4960      3     11   1077      5   3317
    223    211     34   1750      4   4583     20     37    228      9
     14     29      5   4826    122      1     74    145    513     27
    105    206  16280   1262      3  23503     23   6463     22      2
   1431     54      6      1   8616    782   3312      2   3121   1283
      7  40800      5    212    309  41644     40  13044  10740   5693
   3349      2   2251   6817    142      5   2177   2680   4480      3
   6543   4548      2     31    229  28016     17      5    239      6
 111582      2     60     31     14   9966     17     72   9632    122
      1  83734   1736     66     60     31   1993     62     14    838
      3   7842     37      1  36053    124      1    375      6   1122
   1500     64      2     31    620   2462  92441     37     35    165
     14  10636     16     86     16   6830      2     42   2721    783
     37    101     23     16     82      1  24765     22      3  47291
     68    357     38     92      5    639     43     72    979    139
     37      1   1392      2   6463     23      7   3818     14  14581
     22   1510      7    967     18     41   4229      7   4470      1
    249     14  25202      2      5   9040     13  19413      3    106
      2      1  17660   3097    438      1   4480    503     94      5
  11569      3     27      1  11129      2   6463   1091   1924     94
      1   1125  57215     67      3      9     14     41   6813    200
     74      2  10368   2102      2     13   1407      1  55353   5009
      4     32    186    788      7      9     10   7493      2      1
    342  13056      2     17     12    220      4     13      6   1039
   9839     10      1  12443   9205      2    859      5    348      3
      4     35    235      2  42426    885     83    157   4577     50
    175   1497     10   1360      3]
```

```
[ 727 1203  844   85    0    8  156    2  351   17   32    0    0 2299    2
    4   13    0    0    0    3   44   30 4381    4  819    2 3430    4  212
 6093    2    4   66    1    0    0    0    3    1   73  162  957    8    5
   58  296   18    0    0 5009   14  337    0 3188  164  313  215    0    5
  372   48    5 2147    2    4  397   44   17  637    0    5    0   48  119
  204    1 7427   43   13   10    1  214    3   23  304    6 5009   80   92
 5002    0    2   18 5281 2982   80   92  668 6463    2   18    0    0 5805
 6830    2    4   44   27  204  155    1  893    3   22    1   73  634    8
   13    2   12   68    2  304    6    0    2    9   14   10    0  122 1056
    2   46   30  157   42   63 1558   12   41 4960    3   11 1077    5 3317
  223  211   34 1750    4 4583   20   37  228    9   14   29    5 4826  122
    1   74  145  513   27  105  206    0 1262    3    0   23 6463   22    2
 1431   54    6    1    0  782 3312    2 3121 1283    7    0    5  212  309
    0   40    0    0 5693 3349    2 2251 6817  142    5 2177 2680 4480    3
 6543 4548    2   31  229    0   17    5  239    6    0    2   60   31   14
    0   17   72    0  122    1    0 1736   66   60   31 1993   62   14  838
    3 7842   37    1    0  124    1  375    6 1122 1500   64    2   31  620
 2462    0   37   35  165   14    0   16   86   16 6830    2   42 2721  783
   37  101   23   16   82    1    0   22    3    0   68  357   38   92    5
  639   43   72  979  139   37    1 1392    2 6463   23    7 3818   14    0
   22 1510    7  967   18   41 4229    7 4470    1  249   14    0    2    5
    0   13    0    3  106    2    1    0 3097  438    1 4480  503   94    5
    0    3   27    1    0    2 6463 1091 1924   94    1 1125    0   67    3
    9   14   41 6813  200   74    2    0 2102    2   13 1407    1    0 5009
    4   32  186  788    7    9   10 7493    2    1  342    0    2   17   12
  220    4   13    6 1039    0   10    1    0    0    2  859    5  348    3
    4   35  235    2    0  885   83  157 4577   50  175 1497   10 1360    3]
```

To finish up, we save our token id based reviews to text files and save our dictionary in case we ever need a conversion of ID back to text.

```
## save dictionary
np.save('preprocessed_data/imdb_dictionary.npy',np.asarray(id_to_word))

## save training data to single text file
with open('preprocessed_data/imdb_train.txt','w',encoding='utf-8') as f:
    for tokens in x_train_token_ids:
        for token in tokens:
            f.write("%i " % token)
        f.write("\n")

## save test data to single text file
with open('preprocessed_data/imdb_test.txt','w',encoding='utf-8') as f:
    for tokens in x_test_token_ids:
        for token in tokens:
            f.write("%i " % token)
        f.write("\n")
```

GloVe Features

The GloVe features already provide a text file with a 300 dimensional word embedding for over 2 million tokens. This dictionary is sorted by word frequency as well but for a different dataset. Therefore, we still need to have some way of using the IMDB dataset with their dictionary. Continue adding this code to **preprocess_data.py**

```
glove_filename = '/projects/training/bauh/NLP/glove.840B.300d.txt'
with open(glove_filename,'r',encoding='utf-8') as f:
    lines = f.readlines()
```

```
glove_dictionary = []
glove_embeddings = []
count = 0
for line in lines:
    line = line.strip()
    line = line.split(' ')
    glove_dictionary.append(line[0])
    embedding = np.asarray(line[1:],dtype=np.float)
    glove_embeddings.append(embedding)
    count+=1
    if(count>=100000):
        break

glove_dictionary = np.asarray(glove_dictionary)
glove_embeddings = np.asarray(glove_embeddings)
# added a vector of zeros for the unknown tokens
glove_embeddings = np.concatenate((np.zeros((1,300)),glove_embeddings))
```

We have two new arrays **glove_dictionary** and **glove_embeddings**. The first is the same as **id_to_word** but a different order and **glove_embeddings** contain the actual embeddings for each token. To save space, only the first 100k tokens are kept. Also, notice a 300 dimensional vector of 0s is preprended to the array of embeddings to be used for the **unknown** token.

```
word_to_id = {token: idx for idx, token in enumerate(glove_dictionary)}

x_train_token_ids = [[word_to_id.get(token,-1)+1 for token in x] for x in x_train]
x_test_token_ids = [[word_to_id.get(token,-1)+1 for token in x] for x in x_test]
```

We do exactly what we did before but with this new GloVe dictionary and finally save the data in the new format.

```
np.save('preprocessed_data/glove_dictionary.npy',glove_dictionary)
np.save('preprocessed_data/glove_embeddings.npy',glove_embeddings)

with open('preprocessed_data/imdb_train_glove.txt','w',encoding='utf-8') as f:
    for tokens in x_train_token_ids:
        for token in tokens:
            f.write("%i " % token)
        f.write("\n")

with open('preprocessed_data/imdb_test_glove.txt','w',encoding='utf-8') as f:
    for tokens in x_test_token_ids:
        for token in tokens:
            f.write("%i " % token)
        f.write("\n")
```

Part 1 - Bag of Words

1a - Without GloVe Features

A bag of words model is one of the most basic models for document classification. It is a way of handling varying length inputs (sequence of token ids) to get a single output (positive or negative sentiment).

The token IDs can be thought of as an alternative representation of a 1-hot vector. A word embedding layer is usually a **d by V** matrix where **V** is the size of the vocabulary and **d** is the embedding size.

Multiplying a 1-hot vector (of length **V**) by this matrix results in a **d** dimensional embedding for a particular token. We can avoid this matrix multiplication by simply taking the column of the word embedding matrix corresponding with that particular token.

A word embedding matrix doesn't tell you how to handle the sequence information. In this case, you can picture the bag of words as simply taking the mean of all of the 1-hot vectors for the entire sequence and then multiplying this by the word embedding matrix to get a document embedding. Alternatively, you can take the mean of all of the word embeddings for each token and get the same document embedding. Now we are left with a single **d** dimensional input representing the entire sequence.

Note that the bag of words method utilizes all of the tokens in the sequence but loses all of the information about when the tokens appear within the sequence. Obviously the order of words carries a significant portion of the information contained within written text but this technique still works out rather well.

Create a new directory '1a/' within your assignment directory. You will create two python files, **BOW_model.py** and **BOW_sentiment_analysis.py**. The first is where we will define our bag of words pytorch model and the second will contain the train/test loops.

BOW_model.py

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
import torch.distributed as dist

class BOW_model(nn.Module):
    def __init__(self, vocab_size, no_of_hidden_units):
        super(BOW_model, self).__init__()
        ## will need to define model architecture

    def forward(self, x, t):
        # will need to define forward function for when model gets called
```

The inputs for defining our model object are **vocab_size** and **no_of_hidden_units** (we can decide on actual values for these later). We will need to define an embedding layer, a single hidden layer with batch normalization, an output layer, a dropout layer, and the loss function.

```
    def __init__(self, vocab_size, no_of_hidden_units):
        super(BOW_model, self).__init__()

        self.embedding = nn.Embedding(vocab_size,no_of_hidden_units)

        self.fc_hidden = nn.Linear(no_of_hidden_units,no_of_hidden_units)
        self.bn_hidden = nn.BatchNorm1d(no_of_hidden_units)
        self.dropout = torch.nn.Dropout(p=0.5)

        self.fc_output = nn.Linear(no_of_hidden_units, 1)

        self.loss = nn.BCEWithLogitsLoss()
```

Mathematically speaking, there is nothing unique about a word embedding layer. It's a matrix multiplication with a bias term. A word embedding layer actually has the exact same number of weights as a linear layer. The difference comes down to the PyTorch implementation. The input to the

embedding layer is just an index while a linear layer requires an appropriately sized vector. We could use a linear layer if we wanted but we'd have to convert our token IDs to a bunch of 1-hot vectors and do a lot of unnecessary matrix multiplication since most of the entries are 0. As mentioned above, a matrix multiplying a 1-hot vector is the same as just extracting a single column of the matrix, which is a lot faster. Embedding layers are used frequently in NLP since the input is almost always a 1-hot representation.

Even though you will never explicitly see a 1-hot vector in the code, it is very useful intuitively to view the input as a 1-hot vector.

```
    def forward(self, x, t):

        bow_embedding = []
        for i in range(len(x)):
            lookup_tensor = Variable(torch.LongTensor(x[i])).cuda()
            embed = self.embedding(lookup_tensor)
            embed = embed.mean(dim=0)
            bow_embedding.append(embed)
        bow_embedding = torch.stack(bow_embedding)

        h = self.dropout(F.relu(self.bn_hidden(self.fc_hidden(bow_embedding))))
        h = self.fc_output(h)

        return self.loss(h[:,0],t), h[:,0]
```

When we call our model, we will need to provide it with an input **x** and target labels **t**.

This implementation is not typical in the sense that a *for loop* is used for the embedding layer as opposed to the more typical batch processing. Assume the input **x** is a list of length **batch_size** and each element of this list is a numpy array containing the token ids for a particular sequence. These sequences are different length which is why **x** is not simply a torch tensor of size **batch_size by sequence_length**. Within the loop, the **lookup_tensor** is a single sequence of token ids which can be fed into the embedding layer. This returns a torch tensor of length **sequence_length by embedding_size**. We take the mean over the dimension corresponding to the sequence length and append it to the list **bow_embedding**. This mean operation is considered the bag of words. Note this operation returns the same vector **embed** regardless of how the token ids were ordered in the **lookup_tensor**.

The **torch.stack()** operation is a simple way of converting a list of torch tensors of length **embedding_size** to a single torch tensor of length **batch_size by embedding_size**. The **bow_embedding** is passed through a linear layer, a batch normalization layer, a ReLU operation, and dropout.

Our output layer has only one output even though there are two classes for sentiment (positive and negative). We could've had a separate output for each class but I actually saw an improvement of around 1-2% by using the **nn.BCEWithLogitsLoss()** (binary cross entropy with logits loss) as opposed to the usual **nn.CrossEntropyLoss()** you usually see with multi-class classification problems.

Note that two values are returned, the loss and the logit score. The logit score can be converted to an actual probability by passing it through the sigmoid function or it can be viewed as any score greater than 0 is considered positive sentiment. The actual training loop ultimately determines what will be done with these two returned values.

BOW_sentiment_analysis.py

This is the file you'll actually run for training the model.

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
import torch.distributed as dist

import time
import os
import sys

from BOW_model import BOW_model
```

Import all of the basic packages we need as well our **BOW_model** we defined previously.

```
#imdb_dictionary = np.load('../preprocessed_data/imdb_dictionary.npy')
vocab_size = 8000 # imdb_dictionary.shape[0]

x_train = []
with open('../preprocessed_data/imdb_train.txt','r',encoding='utf-8') as f:
    lines = f.readlines()
for line in lines:
    line = line.strip()
    line = line.split(' ')
    line = np.asarray(line,dtype=np.int)

    line[line>vocab_size] = 0

    x_train.append(line)
x_train = x_train[0:25000]
y_train = np.zeros((25000,))
y_train[0:12500] = 1
```

Here is where we can much more quickly load in our data one time in the exact format we need. We can also choose our dictionary size at this point. I suggest starting out with 8000 as it was shown earlier (in the preprocessing data section) how this greatly reduces the number of weights in the word embedding layer without ignoring too many unique tokens within the actual dataset. Note that each **line** within our imdb_train.txt file is a single review made up of token ids. We can convert any token id greater than the dictionary size to the **unknown** token ID **0**. Remember also we actually had 75,000 total training files but only the first 25,000 are labeled (we will use the unlabeled data in part 3). The last three lines of code just grab the first 25,000 sequences and creates a vector for the labels (the first 125000 are labeled 1 for positive and the last 12500 are labeled 0 for negative).

```
x_test = []
with open('../preprocessed_data/imdb_test.txt','r',encoding='utf-8') as f:
    lines = f.readlines()
for line in lines:
    line = line.strip()
    line = line.split(' ')
    line = np.asarray(line,dtype=np.int)

    line[line>vocab_size] = 0

    x_test.append(line)
y_test = np.zeros((25000,))
y_test[0:12500] = 1
```

Do the same for the test dataset.

```
vocab_size += 1

model = BOW_model(vocab_size,500)
model.cuda()
```

Here we actually define the model with **no_of_hidden_units** equal to 500. Note I aded 1 to **vocab_size**. Remember we actually added 1 to all of the token ids so we could use id 0 for the **unknown** token. The code above kept tokens 1 through 8000 as well as 0 meaning the **vocab_size** is actually 8001.

```
# opt = 'sgd'
# LR = 0.01
opt = 'adam'
LR = 0.001
if(opt=='adam'):
    optimizer = optim.Adam(model.parameters(), lr=LR)
elif(opt=='sgd'):
    optimizer = optim.SGD(model.parameters(), lr=LR, momentum=0.9)
```

Define the optimizer with some desired learning rate.

```
batch_size = 200
no_of_epochs = 6
L_Y_train = len(y_train)
L_Y_test = len(y_test)

model.train()

train_loss = []
train_accu = []
test_accu = []
```

Define some parameters such as the **batch_size** and **no_of_epochs**. Put the model in training mode (this allows batch normalization to work correctly as well as dropout). Create a few lists to store any variables of interest.

```
for epoch in range(no_of_epochs):

    # training
    model.train()

    epoch_acc = 0.0
    epoch_loss = 0.0

    epoch_counter = 0

    time1 = time.time()

    I_permutation = np.random.permutation(L_Y_train)

    for i in range(0, L_Y_train, batch_size):

        x_input = [x_train[j] for j in I_permutation[i:i+batch_size]]
        y_input = np.asarray([y_train[j] for j in I_permutation[i:i+batch_size]],dtype=np.int
        target = Variable(torch.FloatTensor(y_input)).cuda()
```

```
        optimizer.zero_grad()
        loss, pred = model(x_input,target)
        loss.backward()

        optimizer.step()   # update weights

        prediction = pred >= 0.0
        truth = target >= 0.5
        acc = prediction.eq(truth).sum().cpu().data.numpy()
        epoch_acc += acc
        epoch_loss += loss.data[0]
        epoch_counter += batch_size

    epoch_acc /= epoch_counter
    epoch_loss /= (epoch_counter/batch_size)

    train_loss.append(epoch_loss)
    train_accu.append(epoch_acc)

    print(epoch, "%.2f" % (epoch_acc*100.0), "%.4f" % epoch_loss, "%.4f" % float(time.time()-t

    # ## test
    model.eval()

    epoch_acc = 0.0
    epoch_loss = 0.0

    epoch_counter = 0

    time1 = time.time()

    I_permutation = np.random.permutation(L_Y_test)

    for i in range(0, L_Y_test, batch_size):

        x_input = [x_test[j] for j in I_permutation[i:i+batch_size]]
        y_input = np.asarray([y_test[j] for j in I_permutation[i:i+batch_size]],dtype=np.int)
        target = Variable(torch.FloatTensor(y_input)).cuda()

        loss, pred = model(x_input,target)

        prediction = pred >= 0.0
        truth = target >= 0.5
        acc = prediction.eq(truth).sum().cpu().data.numpy()
        epoch_acc += acc
        epoch_loss += loss.data[0]
        epoch_counter += batch_size

    epoch_acc /= epoch_counter
    epoch_loss /= (epoch_counter/batch_size)

    test_accu.append(epoch_acc)

    time2 = time.time()
    time_elapsed = time2 - time1

    print("  ", "%.2f" % (epoch_acc*100.0), "%.4f" % epoch_loss)

torch.save(model,'BOW.model')
data = [train_loss,train_accu,test_accu]
data = np.asarray(data)
np.save('data.npy',data)
```

There is nothing particularly unique about this training loop compared to what you've seen in the past and it should be pretty self explanatory. The only interesting part is the variable **x_input** we send to the model is not actually a torch tensor at this moment. It's simply a list of lists containing the token ids. Remember that this is dealt with in the **BOW_model.forward()** function.

With the current hyperparameters I've provided, this model should achieve around 86%-88% on the test dataset. If you check the original paper that came out with this dataset back in 2011, you'll see this model performs just as well as nearly all of the techniques shown on page 7.

This should only take about 15-30 minutes to train. Report results for this model as well as at least two other trials (run more but only report on two interesting cases). Try to get a model that overfits (high accuracy on training data, lower on testing data) as well as one that underfits (low accuracy on both). Overfitting is easily achieved by greatly increasing the **no_of_hidden_units**, removing dropout, adding a second or third hidden layer in **BOW_model.py** as well as a few other things. Underfitting can be easily done by using significantly less hidden units.

Other things to consider: * Early stopping - notice how sometimes the test performance will actually get worse the longer you train * Longer training - this might be necessary depending on the choice of hyperparameters * Learning rate schedule - after a certain number of iterations or once test performance stops increasing, try reducing the learning rate by some factor * Larger models may take up too much memory meaning you may need to reduce the batch size * Play with the dictionary size - see if there is any difference utilizing a large portion of the dictionary (like 100k) compared to very few words (500) * You could try removing stop words, **nltk.corpus.stopwords.words('english')** returns a list of stop words. You can find their corresponding index using the **imdb_dictionary** and convert any of these to 0 when loading the data into **x_train** (remember to add 1 to the index to compensate for the unknown token) * Try SGD optimizer instead of adam. This might take more epochs. Sometimes a well tuned SGD with momentum performs better

Try to develop a basic intuition for selecting these parameters yourself and see how big of an effect you can have on the performance. There are four more vastly different models you will train throughout the rest of this assignment.

1b - Using GloVe Features

We will now train another bag of words model but use the pre-trained GloVe features in place of the word embedding layer from before. Typically by leveraging larger datasets and regularization techniques, overfitting can be reduced. The GloVe features were pretrained on over 840 billion tokens. Our training dataset contains 20 million tokens and $^2/_3$ of the 20 million tokens are part of the unlabeled reviews which weren't used in part 1a. The GloVe dataset is over 100 thousand times larger.

The hope is then that these 300 dimensional GloVe features already contain a significant amount of useful information since they were pre-trained on such a large dataset and that will improve performance for our sentiment classification.

Create a directory '1b/'. I would go ahead and copy **BOW_model.py** and **BOW_sentiment_analysis.py** from part 1a here as most of the code is the same.

BOW_model.py

```
class BOW_model(nn.Module):
    def __init__(self, no_of_hidden_units):
        super(BOW_model, self).__init__()

        self.fc_hidden1 = nn.Linear(300,no_of_hidden_units)
        self.bn_hidden1 = nn.BatchNorm1d(no_of_hidden_units)
        self.dropout1 = torch.nn.Dropout(p=0.5)
```

```
        # self.fc_hidden2 = nn.Linear(no_of_hidden_units,no_of_hidden_units)
        # self.bn_hidden2 = nn.BatchNorm1d(no_of_hidden_units)
        # self.dropout2 = torch.nn.Dropout(p=0.5)

        self.fc_output = nn.Linear(no_of_hidden_units, 1)

        self.loss = nn.BCEWithLogitsLoss()

    def forward(self, x, t):

        h = self.dropout1(F.relu(self.bn_hidden1(self.fc_hidden1(x))))
        # h = self.dropout2(F.relu(self.bn_hidden2(self.fc_hidden2(h))))
        h = self.fc_output(h)

        return self.loss(h[:,0],t), h[:,0]
```

This is actually simpler than part 1a considering it doesn't need to do anything for the embedding layer. We know the input **x** will be a torch tensor of **batch_size by 300** (the mean GloVe features over an entire sequence).

BOW_sentiment_analysis.py

```
glove_embeddings = np.load('../preprocessed_data/glove_embeddings.npy')
vocab_size = 8000

x_train = []
with open('../preprocessed_data/imdb_train_glove.txt','r',encoding='utf-8') as f:
    lines = f.readlines()
for line in lines:
    line = line.strip()
    line = line.split(' ')
    line = np.asarray(line,dtype=np.int)

    line[line>vocab_size] = 0

    line = np.mean(glove_embeddings[line],axis=0)

    x_train.append(line)
x_train = np.asarray(x_train)
x_train = x_train[0:25000]
y_train = np.zeros((25000,))
y_train[0:12500] = 1

x_test = []
with open('../preprocessed_data/imdb_test_glove.txt','r',encoding='utf-8') as f:
    lines = f.readlines()
for line in lines:
    line = line.strip()
    line = line.split(' ')
    line = np.asarray(line,dtype=np.int)

    line[line>vocab_size] = 0

    line = np.mean(glove_embeddings[line],axis=0)

    x_test.append(line)
x_test = np.asarray(x_test)
y_test = np.zeros((25000,))
y_test[0:12500] = 1

vocab_size += 1
```

```
model = BOW_model(500) # try 300 as well

model.cuda()
```

This first part is nearly the same besides the fact that we can actually go ahead and do the mean operation for the entire sequence one time when loading in the data. We load in the **glove_embeddings** matrix, convert all out-of-dictionary tokens to the **unknown** token for each review, extract the embedding for each token in the sequence from the matrix, take the mean of these emeddings, and append this to the **x_train** or **x_test** list.

The rest of the code is the same besides grabbing the data for each batch within the actual train/test loop. The code below is for training and you'll need to modify it slightly for testing by changing **x_train** to **x_test**.

```
    ## within the training loop
        x_input = x_train[I_permutation[i:i+batch_size]]
        y_input = y_train[I_permutation[i:i+batch_size]]

        data = Variable(torch.FloatTensor(x_input)).cuda()
        target = Variable(torch.FloatTensor(y_input)).cuda()

        optimizer.zero_grad()
        loss, pred = model(data,target)
```

Just like before, try a few different hyperparameter settings and report the results.

Against the intuition laid out in the beginning of this section, this model actually seems to perform worse on average than the one in part a. This seems to achieve anywhere between 81-87%.

Let's take a look at what's happening. In part 1a, test accuracy typically seems to achieve its max after the 3rd epoch and begins to decrease with more training while the training accuracy continues to increase well into 90+%. This is a sure sign of overfitting. The training accuracy for part 1b stops much earlier (around 86-88%) and doesn't seem to improve much more.

Nearly 95% of the weights belong to the embedding layer in part 1a. We're training significantly less in part 1b and can't actually fine-tune the word embeddings at all. Using only 300 hidden weights for part 1b results in very little overfitting while still achieving decent accuracy.

Part 2 - Recurrent Neural Network

Take the following two reviews: * "Although the movie had great visual effects, I thought it was terrible." * "Although the movie had terrible visual effects, I thought it was great."

The first review clearly has an overall negative sentiment while the bottom review clearly has an overall positive sentiment. Both sentences would result in the exact same output if using the bag of words approach.

Clearly there is a lot of useful information which could maybe be utilized more effectively if we didn't discard the sequence information like we did in part 1. By designing a model capable of capturing this additional source of information, we can potentially achieve better results but also greatly increase the risk of overfitting. This is heavily related to the curse of dimensionality.

A recurrent neural network can be used to maintain the temporal information and process the data as an actual sequence. Part 2 will consist of training recurrent neural networks built with LSTM layers. We will train two separate models again, one from scratch with a word embedding layer and one with

GloVe features.

2a - Without GloVe Features

Create a directory '2a/'. You will need three files, **RNN_model.py**, **RNN_sentiment_analysis.py**, and **RNN_test.py**.

RNN_model.py

An LSTM Cell in pytorch needs a variable for both the internal cell state **c(t)** as well as the hidden state **h(t)**. Let's start by creating a **StatefulLSTM()** class that can maintain these values for us.

```
class StatefulLSTM(nn.Module):
    def __init__(self,in_size,out_size):
        super(StatefulLSTM,self).__init__()

        self.lstm = nn.LSTMCell(in_size,out_size)
        self.out_size = out_size

        self.h = None
        self.c = None

    def reset_state(self):
        self.h = None
        self.c = None

    def forward(self,x):

        batch_size = x.data.size()[0]
        if self.h is None:
            state_size = [batch_size, self.out_size]
            self.c = Variable(torch.zeros(state_size)).cuda()
            self.h = Variable(torch.zeros(state_size)).cuda()
        self.h, self.c = self.lstm(x,(self.h,self.c))

        return self.h
```

This module uses **self.h** and **self.c** to maintain the necessary information while processing a sequence. We can reset the layer at anytime by calling **reset_state()**. The **nn.LSTMCell()** contains all of the actual LSTM weights as well as all of the operations.

When processing sequence data, we will need to apply dropout after every timestep. It has been shown to be more effective to use the same dropout mask for an entire sequence as opposed to a different dropout mask each time. More details as well as a paper reference can be found here. Pytorch doesn't have an implementation for this type of dropout so we will make it ourselves.

```
class LockedDropout(nn.Module):
    def __init__(self):
        super(LockedDropout,self).__init__()
        self.m = None

    def reset_state(self):
        self.m = None

    def forward(self, x, dropout=0.5, train=True):
        if train==False:
            return x
        if(self.m is None):
            self.m = x.data.new(x.size()).bernoulli_(1 - dropout)
        mask = Variable(self.m, requires_grad=False) / (1 - dropout)
```

```
        return mask * x
```

Note that if this module is called with **train** set to **False**, it will simply return the exact same input. If **train** is **True**, it checks to see if it already has a dropout mask **self.m**. If it does, it uses this same mask on the data. If it doesn't, it creates a new mask and stores it in **self.m**. As long as we reset our **LockedDropout()** layer at the beginning of each batch, we can have a single mask applied to the entire sequence.

```
class RNN_model(nn.Module):
    def __init__(self,vocab_size,no_of_hidden_units):
        super(RNN_model, self).__init__()

        self.embedding = nn.Embedding(vocab_size,no_of_hidden_units)#,padding_idx=0)

        self.lstm1 = StatefulLSTM(no_of_hidden_units,no_of_hidden_units)
        self.bn_lstm1= nn.BatchNorm1d(no_of_hidden_units)
        self.dropout1 = LockedDropout() #torch.nn.Dropout(p=0.5)

        # self.lstm2 = StatefulLSTM(no_of_hidden_units,no_of_hidden_units)
        # self.bn_lstm2= nn.BatchNorm1d(no_of_hidden_units)
        # self.dropout2 = LockedDropout() #torch.nn.Dropout(p=0.5)

        self.fc_output = nn.Linear(no_of_hidden_units, 1)

        #self.loss = nn.CrossEntropyLoss()
        self.loss = nn.BCEWithLogitsLoss()

    def reset_state(self):
        self.lstm1.reset_state()
        self.dropout1.reset_state()
        # self.lstm2.reset_state()
        # self.dropout2.reset_state()

    def forward(self, x, t, train=True):

        embed = self.embedding(x) # batch_size, time_steps, features

        no_of_timesteps = embed.shape[1]

        self.reset_state()

        outputs = []
        for i in range(no_of_timesteps):

            h = self.lstm1(embed[:,i,:])
            h = self.bn_lstm1(h)
            h = self.dropout1(h,dropout=0.5,train=train)

            # h = self.lstm2(h)
            # h = self.bn_lstm2(h)
            # h = self.dropout2(h,dropout=0.3,train=train)

            outputs.append(h)

        outputs = torch.stack(outputs) # (time_steps,batch_size,features)
        outputs = outputs.permute(1,2,0) # (batch_size,features,time_steps)

        pool = nn.MaxPool1d(no_of_timesteps)
        h = pool(outputs)
        h = h.view(h.size(0),-1)
        #h = self.dropout(h)
```

```
        h = self.fc_output(h)

        return self.loss(h[:,0],t), h[:,0]#F.softmax(h, dim=1)
```

We create an embedding layer just like in part 1a. We can create a **StatefulLSTM()** layer we created above as well as a **LockedDropout()** layer. The **reset_state()** function is used so we can easily reset the state of any layer in our model that needs it.

The **forward()** function has a few new features. First notice we have an additional input variable **train**. We can't rely on **model.eval()** to handle dropout appropriately for us anymore so we will need to do it ourselves by passing this variable to the **LockedDropout()** layer.

Assume our input **x** is **batch_size by sequence_length**. This means we will be training everything with a fixed sequence length (we will go over how this is done in the actual training loop). By passing this tensor into the embedding layer, it will return an embedding for every single value meaning **embed** is **batch_size by sequence_length by no_of_hidden_units**. We then loop over the sequence one step at a time and append **h** to the **outputs** list every time.

The list **outputs** is converted to a torch tensor using **torch.stack()** and we transform the tensor to get back the shape **batch_size by no_of_hidden_units by no_of_timesteps**. We need it in this ordering because the **nn.MaxPool1d()** operation pools over the last dimension of a tensor. After pooling, the **h.view()** operation removes the last dimension from the tensor (it's only length 1 after the pooling operation) and we're left with a vector **h** of size **batch_size by no_of_hidden_units**. We now pass this into the output layer.

This is a good time to mention how there isn't actually a whole lot of difference between what we're doing here and what we did with the bag of words model. The mean embedding over an entire sequence can be thought of as simply a mean pooling operation. It just so happens that we did this mean pooling operation on a bunch of separate words without a word knowing anything about its surrounding context. We pooled before processing any of the temporal information.

This time we are processing the word embeddings in order such that each subsequent output actually has the context of all words preceding it. Eventually we still need to get to a single output for the entire sequence (positive or negative sentiment) and we do this by now using a max pooling operation over the number of timesteps. As opposed to the bag of words technique, we pooled after processing the temporal information.

We could use average pooling instead of max pooling if we wanted. Max pooling seems to work a little better in practice but it depends on the dataset and problem. Intuitively one can imagine the recurrent network processing a very long sequence of short phrases. These short phrases can appear anywhere within the long sequence and carry some important information about the sentiment. After each important short phrase is seen, it outputs a vector with some large values. Eventually all of these vectors are pooled where only the max values are kept. You are left with a vector for the entire sequence containing only the important information related to sentiment as all of the unimportant information was thrown out. For classifying a sequence with a single label, we will always have to eventually collapse the temporal information. The decision in relation to network design deals with where within the model this should be done.

RNN_sentiment_analysis.py

Create a training loop similar to part 1a for training the RNN model. The majority of the code is the same as before. You'll typically need more epochs (about 20) to train this time depending on the chosen hyperparameters. Start off using 300 hidden units. The following code is the only major change about constructing a batch of data during the training loop.

```
x_input2 = [x_train[j] for j in I_permutation[i:i+batch_size]]
sequence_length = 100
x_input = np.zeros((batch_size,sequence_length),dtype=np.int)
for j in range(batch_size):
    x = np.asarray(x_input2[j])
    sl = x.shape[0]
    if(sl < sequence_length):
        x_input[j,0:sl] = x
    else:
        start_index = np.random.randint(sl-sequence_length+1)
        x_input[j,:] = x[start_index:(start_index+sequence_length)]
y_input = y_train[I_permutation[i:i+batch_size]]

data = Variable(torch.LongTensor(x_input)).cuda()
target = Variable(torch.FloatTensor(y_input)).cuda()

optimizer.zero_grad()
loss, pred = model(data,target,train=True)
loss.backward()
```

In the **RNN_model.py** section, I mentioned assuming the input **x** was **batch_size by sequence_length**. This means we're always training on a fixed sequence length even though the reviews have varying length. We can do this by training on shorter subsequences of a fixed size. The variable **x_input2** is a list with each element being a list of token ids for a single review. These lists within the list are different lengths. We create a new variable **x_input** which has our appropriate dimensions of **batch_size by sequence_length** which here I've specified as 100. We then loop through each review, get the total length of the review (stored in the variable **sl**), and check if it's greater or less than our desired **sequence_length** of 100. If it's too short, we just use the full review and the end is padded with 0s. If **sl** is larger than **sequence_length** (which it usually is), we randomly extract a subsequence of size **sequence_length** from the full review.

Notice during testing, you can choose a longer **sequence_length** such as 200 to get more context. However, due to the increased computation from the LSTM layer and the increased number of epochs, you may way to wrap the testing loop in the following to prevent it from happening everytime.

This is a very important aspect of training RNNs and can have a very large impact on the results. By using long sequences, we provide the model with more information (it can learn longer dependencies) which seems like it'd always be a good thing. However, by using short sequences, we are providing less of an opportunity to overfit and artificially increasing the size of the dataset. There are more subsequences of length 50 than there are of length 100. Additionally, although we want our RNN to learn sequences, we don't want it to overfit on where the phrases are within a sequence. By using subsequences, we are kind of training the model to be invariant about where it starts within the review which can be a good thing.

To make matters more complicated, you now have to decide how to test your sequences. If you train on too short of sequences, the internal state **c** may not have reached some level of steady state as it can grow unbounded if the network is never forced to forget any information. Therefore, when testing on long sequences, this internal state becomes huge as it never forgets things which results in an output distribution later layers have never encountered before in training and it becomes wildly inaccurate. This depends heavily on the application. For this particular dataset, I didn't notice too many issues.

I'd suggest trying to train a model on short sequences (50 or less) as well as long sequences (250+) just to see the difference in its ability to generalize. The amount of GPU memory required to train an RNN is dependent on the sequence length. If you decide to train on long sequences, you may need to reduce your batch size.

Each epoch will take roughly a few minutes. With the increased number of epochs necessary for training, it may be wise to wrap your testing loop with the following to prevent it from testing every single time.

```
    if((epoch+1)%3)==0:
        # do testing loop
```

Make sure to save your model after training.

```
torch.save(rnn,'rnn.model')
```

RNN_test.py

These typically take longer to train so you don't want to be testing on the full sequences after every epoch as it's a huge waste of time. It's easier and quicker to just have a separate script to test the model as you only need a rough idea of test performance during training.

Additionally, we can use this to see how the results change depending on the test sequence length. Copy the majority of the code from **RNN_sentiment_analysis.py**. Replace the line where you create your model with the following:

```
model = torch.load('rnn.model')
```

This is your trained model saved from before. Remove the training portion within the loop while keeping only the evaluation section. Change number of epochs to around 10. Within the test loop, add the following:

```
sequence_length = (epoch+1)*50
```

This way it will loop through the entire test dataset and report results for varying sequence lengths. Here is an example I tried for a network trained on sequences of length 100 with 300 hidden units for the LSTM and a dictionary size of 8000.

```
sequence length, test accuracy, test loss, elapsed time
50  76.14 0.7071 17.5213
100  81.74 0.5704 35.1576
150  84.51 0.4760 57.9063
200  86.10 0.4200 84.7308
250  86.69 0.3985 115.8944
300  86.98 0.3866 156.6962
350  87.00 0.3783 203.2236
400  87.25 0.3734 257.9246
450  87.22 0.3726 317.1263
```

The results here might go against the intuition of what was expected. It actually performs worse than the bag of words model from part 1a. However, this does sort of make sense considering we were already overfitting before and we vastly increased the capabilities of the model. It performs nearly as well and with enough regularization/dropout, you can almost certainly achieve better results. Training on shorter sequences can potentially help prevent overfitting as well.

Report the results with the initial hyperparameters I provided as well as two more interesting cases.

2b - With GloVe Features

You should copy **RNN_model.py**, **RNN_sentiment_analysis.py**, and **RNN_test.py** from part 2a over here as a starting point.

RNN_model.py

You'll still need the **LockedDropout()** and **StatefulLSTM()**.

```
class RNN_model(nn.Module):
    def __init__(self,no_of_hidden_units):
        super(RNN_model, self).__init__()

        # self.embedding = nn.Embedding(vocab_size,no_of_hidden_units)#,padding_idx=0)

        self.lstm1 = StatefulLSTM(300,no_of_hidden_units)
        self.bn_lstm1= nn.BatchNorm1d(no_of_hidden_units)
        self.dropout1 = LockedDropout() #torch.nn.Dropout(p=0.5)

        # self.lstm2 = StatefulLSTM(no_of_hidden_units,no_of_hidden_units)
        # self.bn_lstm2= nn.BatchNorm1d(no_of_hidden_units)
        # self.dropout2 = LockedDropout() #torch.nn.Dropout(p=0.5)

        self.fc_output = nn.Linear(no_of_hidden_units, 1)

        #self.loss = nn.CrossEntropyLoss()
        self.loss = nn.BCEWithLogitsLoss()

    def reset_state(self):
        self.lstm1.reset_state()
        self.dropout1.reset_state()
        # self.lstm2.reset_state()
        # self.dropout2.reset_state()

    def forward(self, x, t, train=True):

        no_of_timesteps = x.shape[1]

        self.reset_state()

        outputs = []
        for i in range(no_of_timesteps):

            h = self.lstm1(x[:,i,:])
            h = self.bn_lstm1(h)
            h = self.dropout1(h,dropout=0.5,train=train)

            # h = self.lstm2(h)
            # h = self.bn_lstm2(h)
            # h = self.dropout2(h,dropout=0.3,train=train)

            outputs.append(h)

        outputs = torch.stack(outputs) # (time_steps,batch_size,features)
        outputs = outputs.permute(1,2,0) # (batch_size,features,time_steps)

        pool = nn.MaxPool1d(x.shape[1])
        h = pool(outputs)
        h = h.view(h.size(0),-1)

        h = self.fc_output(h)

        return self.loss(h[:,0],t), h[:,0]#F.softmax(h, dim=1)
```

The embedding layer is removed from before.

RNN_sentiment_analysis.py

Modify the **RNN_sentiment_analysis.py** from part 2a to work with GloVe features.

RNN_test.py

Here is an example output from one of the models I trained.

```
50 78.94 0.5010 24.4657
100 84.63 0.3956 48.4858
150 87.62 0.3240 75.0972
200 89.18 0.2932 109.8104
250 89.90 0.2750 149.9544
300 90.50 0.2618 193.7136
350 90.83 0.2530 245.5717
400 90.90 0.2511 299.2605
450 91.08 0.2477 360.4909
500 91.19 0.2448 428.9970
```

We see some progress here and the results are the best yet. Although the bag of words model with GloVe features performed poorly, we now see the GloVe features are much better utilized when allowing the model to handle the temporal information and overfitting seems to be less of a problem since we aren't retraining the embedding layer. As we saw in part 2a, the LSTM seems to be overkill and the dataset doesn't seem to be complex enough to train a model completely from scratch without overfitting.

Part 3 - Language Model

This final section will be about training a language model by leveraging the additional unlabeled data and showing how this pretraining stage typically leads to better results on other tasks like sentiment analysis.

A language model gives some probability distribution over the words in a sequence. We can essentially feed sequences into a recurrent neural network and train the model to predict the following word. Note that this doesn't require any additional data labeling. The words themselves are the labels. This means we can utilize all 75000 reviews in the training set.

3a - Training the Language Model

This section will have two files, **RNN_language_model.py** and **train_language_model.py**.

RNN_language_model.py

Once again you will need **LockedDropout()** and **StatefulLSTM()**.

```
class RNN_language_model(nn.Module):
    def __init__(self,vocab_size, no_of_hidden_units):
        super(RNN_language_model, self).__init__()

        self.embedding = nn.Embedding(vocab_size,no_of_hidden_units)

        self.lstm1 = StatefulLSTM(no_of_hidden_units,no_of_hidden_units)
        self.bn_lstm1= nn.BatchNorm1d(no_of_hidden_units)
        self.dropout1 = LockedDropout()

        self.lstm2 = StatefulLSTM(no_of_hidden_units,no_of_hidden_units)
```

```
        self.bn_lstm2= nn.BatchNorm1d(no_of_hidden_units)
        self.dropout2 = LockedDropout()

        self.decoder = nn.Linear(no_of_hidden_units, vocab_size)

        self.loss = nn.CrossEntropyLoss()#ignore_index=0)

    def reset_state(self):
        self.lstm1.reset_state()
        self.dropout1.reset_state()
        self.lstm2.reset_state()
        self.dropout2.reset_state()

    def forward(self, x, train=True):

        embed = self.embedding(x) # batch_size, time_steps, features
        no_of_timesteps = embed.shape[1]
        self.reset_state()

        outputs = []
        for i in range(no_of_timesteps - 1):

            h = self.lstm1(embed[:,i,:])
            h = self.bn_lstm1(h)
            h = self.dropout1(h,dropout=0.3,train=train)

            h = self.lstm2(h)
            h = self.bn_lstm2(h)
            h = self.dropout2(h,dropout=0.3,train=train)

            h = self.decoder(h)

            outputs.append(h)

        outputs = torch.stack(outputs) # (time_steps,batch_size,vocab_size)
        target_prediction = outputs.permute(1,0,2) # batch, time, vocab
        outputs = outputs.permute(1,2,0) # (batch_size,vocab_size,time_steps)

        if(train==True):

            target_prediction = target_prediction.contiguous().view(-1,vocab_size)
            target = x[:,1:].contiguous().view(-1)
            loss = self.loss(target_prediction,target)

            return loss, outputs
        else:
            return outputs
```

Unlike before, the final layer has more outputs (called **decoder**) and we no longer do any sort of pooling. Each output of the sequence will be used separately for calculating a particular loss and all of the losses within a sequence will be summed up. The decoder layer has an input dimension the same size as **no_of_hidden_states** and the output size is the same as the **vocab_size**. After every timestep, we have an output for a probability distribution over the entire vocabulary.

After looping through from **i=0** to **i=no_of_timesteps-1**, we have outputs for **i=1** to **i=no_of_timesteps** stored in **target_prediction**. Notice the variable **target** is simply the input sequence **x[:,1:]** without the first index.

Lastly, we'll start off with two stacked LSTM layers. The task of predicting the next word is far more complicated than predicting sentiment for the entire phrase. We don't need to be as worried about overfitting since the dataset is larger (although it's still an issue).

train_language_model.py

Copy **RNN_sentiment_analysis.py** from part 2a. Make sure to import the **RNN_language_model** you made above.

```
print('begin training...')
for epoch in range(0,40):

    ## this is a weird error that happens on BlueWaters
    ## there is some sort of overflow/underflow error
    ## in the adam optimizer when the number of steps gets
    ## too large
    if(epoch>10 and opt=='adam'):
        for group in optimizer.param_groups:
            for p in group['params']:
                state = optimizer.state[p]
                if(state['step']>1000):
                    state['step'] = 1000

    model.train()

    epoch_acc = 0.0
    epoch_loss = 0.0

    epoch_counter = 0

    time1 = time.time()

    I_permutation = np.random.permutation(L_Y_train)

    for i in range(0, L_Y_train, batch_size):

        x_input2 = [x_train[j] for j in I_permutation[i:i+batch_size]]
        sequence_length = 100
        x_input = np.zeros((batch_size,sequence_length),dtype=np.int)
        for j in range(batch_size):
            x = np.asarray(x_input2[j])
            sl = x.shape[0]
            if(sl<sequence_length):
                x_input[j,0:sl] = x
            else:
                start_index = np.random.randint(sl-sequence_length+1)
                x_input[j,:] = x[start_index:(start_index+sequence_length)]
        x_input = Variable(torch.LongTensor(x_input),requires_grad=True).cuda()

        optimizer.zero_grad()
        loss, pred = model(x_input)
        loss.backward()

        norm = nn.utils.clip_grad_norm(model.parameters(),2.0)

        optimizer.step()   # update gradients

        values,prediction = torch.max(pred,1)
        prediction = prediction.cpu().data.numpy()
        accuracy = float(np.sum(prediction==x_input.cpu().data.numpy()[:,1:]))/sequence_lengt
        epoch_acc += accuracy
        epoch_loss += loss.data[0]
        epoch_counter += batch_size

        if (i+batch_size) % 1000 == 0 and epoch==0:
           print(i+batch_size, accuracy/batch_size, loss.data[0], norm, "%.4f" % float(time.t
    epoch_acc /= epoch_counter
```

```
epoch_loss /= (epoch_counter/batch_size)

train_loss.append(epoch_loss)
train_accu.append(epoch_acc)

print(epoch, "%.2f" % (epoch_acc*100.0), "%.4f" % epoch_loss, "%.4f" % float(time.time()-t

## test
if((epoch+1)%1==0):
    model.eval()

    epoch_acc = 0.0
    epoch_loss = 0.0

    epoch_counter = 0

    time1 = time.time()

    I_permutation = np.random.permutation(L_Y_test)

    #torch.from_numpy(
    for i in range(0, 2000, batch_size):
        #apply .cuda() to move to GPU
        sequence_length = 100
        x_input2 = [x_test[j] for j in I_permutation[i:i+batch_size]]
        x_input = np.zeros((batch_size,sequence_length),dtype=np.int)
        for j in range(batch_size):
            x = np.asarray(x_input2[j])
            sl = x.shape[0]
            if(sl<sequence_length):
                x_input[j,0:sl] = x
            else:
                start_index = np.random.randint(sl-sequence_length+1)
                x_input[j,:] = x[start_index:(start_index+sequence_length)]
        x_input = Variable(torch.LongTensor(x_input)).cuda()

        pred = model(x_input,train=False)

        values,prediction = torch.max(pred,1)
        prediction = prediction.cpu().data.numpy()
        accuracy = float(np.sum(prediction==x_input.cpu().data.numpy()[:,1:]))/sequence_le
        epoch_acc += accuracy
        epoch_loss += loss.data[0]
        epoch_counter += batch_size
        #train_accu.append(accuracy)
        if (i+batch_size) % 1000 == 0 and epoch==0:
           print(i+batch_size, accuracy/batch_size)
    epoch_acc /= epoch_counter
    epoch_loss /= (epoch_counter/batch_size)

    test_accu.append(epoch_acc)

    time2 = time.time()
    time_elapsed = time2 - time1

    print("  ", "%.2f" % (epoch_acc*100.0), "%.4f" % epoch_loss, "%.4f" % float(time.time
torch.cuda.empty_cache()

if(((epoch+1)%2)==0):
    torch.save(model,'temp.model')
    torch.save(optimizer,'temp.state')
    data = [train_loss,train_accu,test_accu]
    data = np.asarray(data)
    np.save('data.npy',data)
```

```
torch.save(model,'language.model')
```

There is a portion of code at the top necessary to prevent some error during training I've only ever seen on BlueWaters. Also notice gradient clipping was added.

This model takes much longer to train, about a day. The accuracy will be relatively low (which makes sense considering it's trying to predict one of 8000 words) but this doesn't actually tell you much. It's better to go by the loss. Perplexity is typically used when comparing language models. More about how the cross entropy loss and perplexity are related can be read about here.

3b - Generating Fake Reviews

Although a general language model assigns a probability **P(w_0, w_1, ..., w_n)** over the entire sequence, we've actually trained ours to predict **P(w_n|w_0, ..., w_n-1)** where each output is conditioned only on previous inputs. This gives us the ability to sample from **P(w_n|w_0, ..., w_n-1)**, feed this sampled token back into the model, and repeat this process in order to generate fake movie reviews. This section will walk you through this process.

Make a new file called **generate_review.py**.

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
import torch.distributed as dist

import time

from language_model import language_model

imdb_dictionary = np.load('preprocessed_data/imdb_dictionary.npy')
vocab_size = 8000 + 1

word_to_id = {token: idx for idx, token in enumerate(imdb_dictionary)}
```

We will actually utilize the vocabulary we constructed earlier to convert sampled indices back to their corresponding words.

```
model = languageModel()
model = torch.load('language.model')
print('model loaded...')
model.cuda()

model.eval()
```

Load the trained language model from part 3a and set it to **eval()** mode.

```
## create partial sentences to "prime" the model
## this implementation requires the partial sentences
## to be the same length if doing more than one
# tokens = [['i','love','this','movie','.'],['i','hate','this','movie','.']]
tokens = [['a'],['i']]

token_ids = np.asarray([[word_to_id.get(token,-1)+1 for token in x] for x in tokens])

##### preload phrase
```

```
x = Variable(torch.LongTensor(token_ids)).cuda()

embed = model.encoder(x) # batch_size, time_steps, features

state_size = [embed.shape[0],embed.shape[2]] # batch_size, features
no_of_timesteps = embed.shape[1]

model.reset_state()

outputs = []
for i in range(no_of_timesteps):

    h = model.lstm1(embed[:,i,:])
    h = model.bn_lstm1(h)
    h = model.dropout1(h,dropout=0.5,train=False)

    h = model.lstm2(h)
    h = model.bn_lstm2(h)
    h = model.dropout2(h,dropout=0.5,train=False)

    h = model.decoder(h)

    outputs.append(h)

outputs = torch.stack(outputs)
outputs = outputs.permute(1,2,0)
output = outputs[:,:,-1]
```

We can start sampling at the very start or after the model has processed a few words already. The latter is akin to autocomplete. In this example, I'm generating two reviews with the first starting simply with the letter/word 'a' and the second starting with the letter/word 'i'. These are both stored in **tokens** and converted to **token_ids** in order to be used as the inputs.

The bottom portion of code then loops through the sequences (both sequences at the same time using batch processing) and "primes" the model with our partial sentences. The variable **output** will be size **batch_size by vocab_size**. Remember this output is not a probability. After passing it through the softmax function, we can interpret it as a probability and sample from it.

```
temperature = 1.0 # float(sys.argv[1])
length_of_review = 150

review = []
####
for j in range(length_of_review):

    ## sample a word from the previous output
    output = output/temperature
    probs = torch.exp(output)
    probs[:,0] = 0.0
    probs = probs/(torch.sum(probs,dim=1).unsqueeze(1))
    x = torch.multinomial(probs,1)
    review.append(x.cpu().data.numpy()[:,0])

    ## predict the next word
    embed = model.encoder(x)

    h = model.lstm1(embed)
    h = model.bn_lstm1(h)
    h = model.dropout1(h,dropout=0.3,train=False)

    h = model.lstm2(h)
```

```
    h = model.bn_lstm2(h)
    h = model.dropout2(h,dropout=0.3,train=False)

    output = model.decoder(h)
```

Here is where we will actually generate the fake reviews. We use the previous **output**, perform the softmax function (assign probability of 0.0 to token id 0 to ignore the **unknown** token), randomly sample based on **probs**, save these indices to the list **review**, and finally get another **output**.

```
review = np.asarray(review)
review = review.T
review = np.concatenate((token_ids,review),axis=1)
review = review - 1
review[review<0] = vocab_size - 1
review_words = imdb_dictionary[review]
for review in review_words:
    prnt_str = ''
    for word in review:
        prnt_str += word
        prnt_str += ' '
    print(prnt_str)
```

Here we simply convert the token ids to their corresponding string. Remember all of the indices need -1 to account for the **unknown** token we added before using it with **imdb_dictionary**.

```
# temperature 1.0
a hugely influential , very strong , nuanced stand up comedy part all too much . this is a fi

i found it fascinating and never being even lower . spent nothing on the spanish 's particula
```

Although these reviews as a whole don't make a lot of sense, it's definitely readable and the short phrases seem quite realistic. The **temperature** parameter from before essentially adjusts the confidence of the model. Using **temperature=1.0** is the same as the regular softmax function which produced the reviews above. As the temperature increases, all of the words will approach having the same probability. As the temperature decreases, the most likely word will approach a probability of 1.0.

```
# temperature 1.3
a crude web backdrop from page meets another eastern story ( written by an author bought ) wh

i wanted to walk out on featuring this expecting even glued and turd make . he genius on dial
```

Note here with a higher temperature, there is still some sense of structure but the phrases are very short and anything longer than a few words doesn't begin to make much sense. Choosing an even larger temperature would result in random words being chosen from the dictionary.

```
## temperature 0.25
a little slow and i found myself laughing out loud at the end . the story is about a young gi

i was a fan of the original , but i was very disappointed . the plot was very weak , the acti
```

With a lower temperature, the predictions can get stuck in loops. However, it is interesting to note how the top review here happened to be a "good" review while the bottom review happened to be a "bad" review. It seems that once the model becomes confident with the tone of the review, it sticks with it. Remember that this language model was trained to simply predict the next word in 12500

positive reviews, 12500 negative reviews, and 50000 neutral reviews. It seems to naturally be taking into consideration the sentiment without explicitly being told to do so.

```
# temperature = 0.5
a very , very good movie , but it is a good movie to watch . the plot is great , the acting i

i usually like this movie , but this is one of those movies that i could n't believe . i was
```

We're not necessarily generating fake reviews here for any particular purpose (although there are applications that call for this). This is more to simply show what the model learned. It's simple enough to see the accuracy increasing and the loss function decreasing throughout part 3a, but this helps us get a much better intuitive understanding of what the model is focusing on.

Create some of your own movie reviews with your trained model. The **temperature** parameter can be sensitive depending on how long the model was trained for. Try "priming" the model with longer phrases ("I hate this movie."/"I love this movie.") to see if the sentiment is maintained throughout the generated review.

3c - Learning Sentiment

After getting a basic understanding in part 3b of what sort of information the language model has captured so far, we will use it as a starting point for training a sentiment analysis model.

Copy all your code from part 2a. Modify **RNN_model.py** to have two lstm layers just like your language model. There are two small differences to be made in **RNN_sentiment_analysis.py**.

```
model = RNN_model(vocab_size,500))

language_model = torch.load('language.model')
model.embedding.load_state_dict(language_model.embedding.state_dict())
model.lstm1.lstm.load_state_dict(language_model.lstm1.lstm.state_dict())
model.bn_lstm1.load_state_dict(language_model.bn_lstm1.state_dict())
model.lstm2.lstm.load_state_dict(language_model.lstm2.lstm.state_dict())
model.bn_lstm2.load_state_dict(language_model.bn_lstm2.state_dict())

model.cuda()
```

You are creating a new **RNN_model** for sentiment analysis but copying all of the weights for the embedding and LSTM layers from the language model.

```
params = []
# for param in model.embedding.parameters():
#     params.append(param)
# for param in model.lstm1.parameters():
#     params.append(param)
# for param in model.bn_lstm1.parameters():
#     params.append(param)
for param in model.lstm2.parameters():
    params.append(param)
for param in model.bn_lstm2.parameters():
    params.append(param)
for param in model.fc_output.parameters():
    params.append(param)

opt = 'adam'
LR = 0.001
if(opt=='adam'):
    optimizer = optim.Adam(params, lr=LR)
```

```
elif(opt=='sgd'):
    optimizer = optim.SGD(params, lr=LR, momentum=0.9)
```

Before defining the optimizer, we're going to make a list of parameters we want to train. The model will overfit if you train everything (you can and should test this yourself). However, you can choose to just fine-tune the second LSTM layer and the output layer.

Train and test the model now just as you did in part 2a.

Here is an example output of mine for a particular model:

```
50 80.98 0.4255 17.0083
100 87.17 0.3043 30.2916
150 90.18 0.2453 45.0554
200 91.15 0.2188 59.9038
250 91.96 0.2022 74.8118
300 92.34 0.1960 89.7251
350 92.64 0.1901 104.7904
400 92.83 0.1863 119.8761
450 92.95 0.1842 134.8656
500 93.01 0.1828 150.3047
```

This performs better than all of the previous models. By leveraging the additional unlabeled data and pre-training the network as a language model, we can achieve even better results than the GloVe features trained on a much larger dataset. To put this in perspective, the state of the art on the IMDB sentiment analysis is around 94.1%.