# IMDB_sent_an_pytorch_LSTM_GloVes

August 14, 2018

## 1 Sentiment Analysis on IMDB reviews: Pytorch implementation of LSTM on top of GloVes

In this notebook, I will try to implement with Pytorch the architecture that I found on this blog:
https://www.oreilly.com/learning/perform-sentiment-analysis-with-lstms-using-tensorflow
which is basically a LSTM on the top of an embedding layer using GloVe pretrained embeddings. I will reuse part of the code presented in the page mentioned above for the data preprocessing.

### 1.1 Libraries

```
In [1]: import numpy as np
        import csv
        import io
        from matplotlib import pyplot
        import pickle
```

### 1.2 Preprocessing and Data exploration

The Data exploration part (measuring the average number of words in the reviews) and the data preprocessing, turning texts into sequence of indexes corresponding the GloVes word embeddings, are done in another notebook called IMDB_sent_an_data_preprocessing. The variable created there are then loaded in the next sections.

### 1.3 Loading matrices of embedding indexes and lists of labels

The pretrained embeddins from GloVe can be downloaded here: https://nlp.stanford.edu/projects/glove/

There are different **word embedding sizes**. The possibilities are 50, 100, 200, 300. We define the one we use next.

```
In [2]: word_emb_size = '100'

In [3]: prepr_dir = '/home/aritz/Documents/CS_Programming_Machine_Learning/Projects/IMDB_sentime

In [4]: ids_train = np.load(prepr_dir+'Saved_embeddings/idsMatrixTrain'+word_emb_size+'.npy')
        ids_test = np.load(prepr_dir+'Saved_embeddings/idsMatrixTest'+word_emb_size+'.npy')
```

Next we load the **labels** with and without **one-hot-encoding** ([1, 0] for positive and [0, 1] for negative).

```
In [5]: with open(prepr_dir+"y_train_ord.txt", "rb") as fp:
            y_train_ord = pickle.load(fp)

In [6]: with open(prepr_dir+"y_test_ord.txt", "rb") as fp:
            y_test_ord = pickle.load(fp)

In [7]: with open(prepr_dir+"y_train.txt", "rb") as fp:
            y_train = pickle.load(fp)

In [8]: with open(prepr_dir+"y_test.txt", "rb") as fp:
            y_test = pickle.load(fp)
```

Next we load the **list of words in the GloVe table** and a numpy array containing the **GloVe look-up table**:

```
In [9]: with open(prepr_dir+"words_list.txt", "rb") as fp:
            words_list = pickle.load(fp)

In [10]: word_vectors = np.load(prepr_dir+'word_vectors.npy')
```

### 1.4 Definition of the model

```
In [11]: import torch
         import torch.nn as nn
         import torch.nn.functional as F
```

In the next cell, I define a `nn.Module` model which does almost all the transformation we want, i.e. from the embedding of the indexes to the making of the logits. I copied some of the code I found in this tutorial: https://pytorch.org/tutorials/beginner/nlp/sequence_models_tutorial.html

The main difficulty when writing these models, is to get the dimensions of all the tensors right. In particular, using batches with DataLoader objects, adds one dimension.

Note that using `nn.CrossEntropyLoss()` as loss function is equivalent to combining `nn.LogSoftmax(dim=1)` as last layer of the model together with `nn.NLLLoss()` as loss function. I choose the first option here as it looks more similar to the implementation using TensorFlow and Keras

```
In [12]: class GloVeLSTM(nn.Module):
             def __init__(self, weights_matrix, keep_prob, n_lstm_units):
                 super(GloVeLSTM, self).__init__()
                 self.embeddings = nn.Embedding.from_pretrained(weights_matrix)
                 _, self.emb_dim = weights_matrix.shape
                 self.embeddings.weight.requires_grad=False
                 self.n_lstm_units = n_lstm_units
                 self.hidden = self.init_hidden()
                 self.lstm = nn.LSTM(input_size=self.emb_dim,
```

```
                                hidden_size=n_lstm_units,
                                dropout=1-keep_prob,
                                batch_first=True).double()
            self.hidden2bin = nn.Linear(n_lstm_units, 2).double()

        def forward(self, inp):
            #Before using the LSTM, we need to clean
            #its hidden and state variable in order
            #to prevent the previous review to influence
            #the output for the new review.
            bs = inp.shape[0]
            self.hidden = self.init_hidden(bs=bs)
            emb_vect = self.embeddings(inp)
            #We only care about the output of the LSTM for
            #the last word of the sentence (which is the
            #first entry of the self.hidden)
            _, self.hidden = self.lstm(
                emb_vect, self.hidden)
            logits = self.hidden2bin(self.hidden[0].view(
                bs, self.n_lstm_units))
            return(logits)


        def init_hidden(self, bs=1):
            hidden = (torch.zeros(1, bs, self.n_lstm_units),
                      torch.zeros(1, bs, self.n_lstm_units))
            hidden = (torch.tensor(hidden[0], dtype=torch.float64),
                      torch.tensor(hidden[1], dtype=torch.float64))
            return(hidden)
```

In the next cells, I make sure that this model works properly.

```
In [13]: classifier = GloVeLSTM(weights_matrix=torch.from_numpy(word_vectors),
                                keep_prob=0.7,
                                n_lstm_units=64)
```

```
/home/aritz/anaconda3/envs/mypy36/lib/python3.6/site-packages/torch/nn/modules/rnn.py:38: UserWa
  "num_layers={}".format(dropout, num_layers))
```

### 1.5 Training of the model

It seems that all input sequences which have a lot of zeros at the end, produce the same logits (without training):

```
In [14]: for i in range(10):
             print(ids_train[i][-10:-1])
             print(classifier(torch.from_numpy(np.int64(ids_train[i])).view(1, -1)))
```

```
[0 0 0 0 0 0 0 0 0]
tensor([[ 0.2196, -0.0089]], dtype=torch.float64)
[0 0 0 0 0 0 0 0 0]
tensor([[ 0.2196, -0.0089]], dtype=torch.float64)
[   116    285    998      7    530    439    413 399999  27015]
tensor(1.00000e-02 *
       [[ 5.0708, -0.3398]], dtype=torch.float64)
[0 0 0 0 0 0 0 0 0]
tensor([[ 0.2196, -0.0089]], dtype=torch.float64)
[   13    37 15890     0     0     0     0     0     0]
tensor([[ 0.2121, -0.0122]], dtype=torch.float64)
[   0 3121    3    0 5317 1468 1351    0 8973]
tensor(1.00000e-02 *
       [[ 1.4467, -4.6229]], dtype=torch.float64)
[0 0 0 0 0 0 0 0 0]
tensor([[ 0.2196, -0.0089]], dtype=torch.float64)
[0 0 0 0 0 0 0 0 0]
tensor([[ 0.2196, -0.0089]], dtype=torch.float64)
[0 0 0 0 0 0 0 0 0]
tensor([[ 0.2196, -0.0089]], dtype=torch.float64)
[0 0 0 0 0 0 0 0 0]
tensor([[ 0.2196, -0.0089]], dtype=torch.float64)
```

Next we define the **loss function**. From this Quora page: https://www.quora.com/What-are-the-differences-between-maximum-likelihood-and-cross-entropy-as-a-loss-function it seems that the negative log likelihood loss and the binary cross-entropy loss are the same thing.

```
In [15]: loss_fcn = nn.CrossEntropyLoss()
```

For the optimizer, one has to give only the parameters which have `requires_grad=True`, as explained here: https://discuss.pytorch.org/t/freeze-the-learnable-parameters-of-resnet-and-attach-it-to-a-new-network/949/9

```
In [16]: parameters = filter(lambda p: p.requires_grad, classifier.parameters())
         optimizer = torch.optim.Adagrad(parameters, lr=0.1)
```

Next we try to train the model on a single input sequence.

```
In [17]: classifier.zero_grad()
         inp = torch.from_numpy(np.int64(ids_train[0]))
         y = y_train_ord[0:1]
         logits = classifier(inp.view(1, -1))

In [18]: type(y)

Out[18]: list

In [19]: loss = loss_fcn(logits, torch.tensor(y))
```

```
In [20]: loss

Out[20]: tensor(0.5854, dtype=torch.float64)

In [21]: loss.backward()

In [22]: optimizer.step()
```

### 1.5.1 Dataset and DataLoader

```
In [23]: from torch.utils.data import Dataset, DataLoader, TensorDataset
```

Here we define our custom Dataset. Note that there exists an official dataset for IMDB:
https://torchtext.readthedocs.io/en/latest/datasets.html#imdb
but I won't use it since my goal is to get comfortable with tools that I could later use for any set of data.

**Edit**: Actually, it looks like there is a cleaner way to define the Dataset, since we have tensors, namely use `my_dataset = data_utils.TensorDataset(ids_train, y_train_ord)`, but in my case, I obtain errors when I try to use it (`TypeError: 'int' object is not callable`), so I will stick with my custom Dataset.

```
In [24]: class MyIMDBDataset(Dataset):

             def __init__(self, ids_matrix, y_ord):
                 self.ids_matrix = ids_matrix
                 self.y_ord = y_ord

             def __len__(self):
                 return(self.ids_matrix.shape[0])

             def __getitem__(self, idx):
                 return((self.ids_matrix[idx], self.y_ord[idx]))

In [25]: train_dataset = MyIMDBDataset(ids_train, y_train_ord)

In [26]: inp, label = train_dataset[0]
```

Note that the `Dataset` object outputs objects having the type that one would expect:

```
In [27]: print(type(inp))

<class 'numpy.ndarray'>


In [28]: print(type(label))

<class 'int'>
```

Next we create a DataLoader

```
In [29]: batch_size = 100

In [30]: trainloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
```

Note that when one use the `DataLoader` object, the objects that we want to iterate over are **automatically wrapped inside tensors**, as one can see in the next cells.

```
In [31]: for i, data in enumerate(trainloader, 0):
             if i>0:
                 break
             print(i)
             inp, label = data
             print(type(inp))
             print(inp)
             print(inp.shape)
             print(type(label))
             print(label)
             print(label.shape)

0
<class 'torch.Tensor'>
tensor([[ 3.3580e+03,  4.0000e+05,  4.0900e+03,  ...,  0.0000e+00,
          0.0000e+00,  0.0000e+00],
        [ 4.1000e+01,  4.2276e+04,  5.9880e+03,  ...,  0.0000e+00,
          0.0000e+00,  0.0000e+00],
        [ 5.8000e+01,  1.3800e+02,  4.1000e+01,  ...,  6.6000e+01,
          3.4680e+03,  6.0000e+00],
        ...,
        [ 9.1220e+03,  1.4000e+01,  1.7000e+02,  ...,  0.0000e+00,
          0.0000e+00,  0.0000e+00],
        [ 3.2490e+05,  4.8000e+01,  5.6000e+01,  ...,  0.0000e+00,
          0.0000e+00,  0.0000e+00],
        [ 1.2000e+02,  7.4400e+02,  7.9289e+04,  ...,  1.2958e+04,
          1.4000e+01,  1.5270e+03]], dtype=torch.int32)
torch.Size([100, 250])
<class 'torch.Tensor'>
tensor([ 1,  1,  0,  0,  1,  0,  1,  0,  0,  0,  0,  0,  1,  0,
         0,  0,  1,  0,  0,  0,  1,  0,  1,  0,  1,  0,  0,  1,
         1,  1,  0,  0,  1,  1,  1,  1,  0,  0,  1,  1,  0,  1,
         1,  0,  1,  1,  1,  1,  0,  1,  0,  0,  0,  1,  0,  0,
         1,  0,  0,  1,  0,  0,  0,  0,  0,  0,  1,  1,  1,  0,
         0,  1,  1,  0,  1,  0,  1,  1,  0,  1,  1,  0,  0,  1,
         0,  1,  1,  0,  0,  0,  0,  1,  0,  1,  1,  0,  1,  1,
         0,  1])
torch.Size([100])
```

We define the Dataset and DataLoader for the test set.

```
In [32]: test_dataset = MyIMDBDataset(ids_test, y_test_ord)

In [33]: testloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

### 1.5.2 Training and Testing

First we train the model for a fixed number of epochs.

The code of the following cell is a variation of what I found on this page https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html (for the training) and on this page https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html (for the testing).

```
In [34]: epoch_first = 0

In [35]: for epoch in range(10):  # loop over the dataset multiple times
             epoch_first = epoch

             running_loss = 0.0
             for i, data in enumerate(trainloader, 0):
                 inp, labels = data
                 # zero the parameter gradients
                 optimizer.zero_grad()
                 # put the inputs into a torch.tensor
                 inp = torch.from_numpy(np.int64(inp))
                 logits = classifier(inp)
                 loss = loss_fcn(logits, torch.tensor(labels))
                 loss.backward()
                 optimizer.step()

             # print statistics
             running_loss += loss.item()
             print('[%d, %5d] loss: %.3f' %
                           (epoch + 1, i + 1, running_loss / 2000))
             running_loss = 0.0

             # print the accuracy on the test set
             if (epoch > 3):
                 correct = 0
                 total = 0

                 with torch.no_grad():
                     for data in testloader:
                         input_test, labels = data
                         input_test = torch.tensor(input_test, dtype = torch.int64)
                         logits = classifier(input_test)
                         _, predicted = torch.max(logits.data, 1)
                         total += labels.size(0)
                         correct += (predicted == labels).sum().item()

                 print('Accuracy of the network on the test set: %05.3f %%' % (
```

```
                    100 * correct / total))

        print('Finished Training')
```

```
[1,   251] loss: 0.000
[2,   251] loss: 0.000
[3,   251] loss: 0.000
[4,   251] loss: 0.000
[5,   251] loss: 0.000
Accuracy of the network on the test set: 52.462 %
[6,   251] loss: 0.000
Accuracy of the network on the test set: 84.501 %
[7,   251] loss: 0.001
Accuracy of the network on the test set: 63.329 %
[8,   251] loss: 0.001
Accuracy of the network on the test set: 57.042 %
[9,   251] loss: 0.000
Accuracy of the network on the test set: 60.370 %
[10,   251] loss: 0.000
Accuracy of the network on the test set: 83.977 %
Finished Training
```

And then we train it until it doesn't improve anymore.

```
In [36]: acc_i = 0
         acc_iplus1 = 0
         acc_iplus2 = 0.3
         epoch = epoch_first
```

```
In [37]: while ((acc_iplus2 > acc_iplus1) or (acc_iplus2 > acc_i) or (acc_iplus1 > acc_i)):
             epoch = epoch + 1
             running_loss = 0.0
             for i, data in enumerate(trainloader, 0):
                 inp, labels = data
                 # zero the parameter gradients
                 optimizer.zero_grad()
                 # put the inputs into a torch.tensor
                 inp = torch.from_numpy(np.int64(inp))
                 logits = classifier(inp)
                 loss = loss_fcn(logits, torch.tensor(labels))
                 loss.backward()
                 optimizer.step()

                 # print statistics
                 running_loss += loss.item()
                 if i % 2000 == 1999:    # print every 2000 mini-batches
                     print('[%d, %5d] loss: %.3f' %
                           (epoch + 1, i + 1, running_loss / 2000))
```

```
                running_loss = 0.0

            # print the accuracy on the test set
            if 0 == 0:
                acc_i = acc_iplus1
                acc_iplus1 = acc_iplus2
                correct = 0
                total = 0

                with torch.no_grad():
                    for data in testloader:
                        input_test, labels = data
                        input_test = torch.tensor(input_test, dtype = torch.int64)
                        logits = classifier(input_test)
                        _, predicted = torch.max(logits.data, 1)
                        total += labels.size(0)
                        correct += (predicted == labels).sum().item()
                acc_iplus2 = correct / total

                print('Accuracy of the network on the test set: %05.3f %%' % (
                    100 * correct / total))
```

```
Accuracy of the network on the test set: 82.425 %
Accuracy of the network on the test set: 83.597 %
Accuracy of the network on the test set: 84.377 %
Accuracy of the network on the test set: 74.269 %
Accuracy of the network on the test set: 82.781 %
Accuracy of the network on the test set: 84.509 %
Accuracy of the network on the test set: 84.541 %
Accuracy of the network on the test set: 76.301 %
Accuracy of the network on the test set: 80.417 %
Accuracy of the network on the test set: 84.401 %
Accuracy of the network on the test set: 84.077 %
Accuracy of the network on the test set: 84.189 %
Accuracy of the network on the test set: 83.981 %
Accuracy of the network on the test set: 83.129 %
```

```
In [38]: print(acc_i)
         print(acc_iplus1)
         print(acc_iplus2)
```

```
0.8418863245470182
0.8398064077436902
0.8312867485300588
```

### 1.5.3 Testing the model

Here we will reuse the code found on this page: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.ht

```
In [35]: correct = 0
         total = 0

         with torch.no_grad():
             for data in testloader:
                 input_test, labels = data
                 input_test = torch.tensor(input_test, dtype = torch.int64)
                 logits = classifier(input_test)
                 _, predicted = torch.max(logits.data, 1)
                 total += labels.size(0)
                 correct += (predicted == labels).sum().item()

         print('Accuracy of the network on the test set: %05.3f %%' % (
             100 * correct / total))
```

```
Accuracy of the network on the test set: 63.697 %
```

## 1.6 Debugging

### 1.6.1 Analysing components of the model step by step

In the next cells we try to find sources of errors. For this we start by creating some toy data.

```
In [36]: keep_prob_toy = 0.7
         n_lstm_units_toy = 7
         len_vocab_toy = 14
         emb_size_toy = 4
         seq_len_toy = 3
         weights_matrix_toy = torch.from_numpy(np.random.random([len_vocab_toy, emb_size_toy]))
         batch_size_toy = 2

In [37]: ids_toy_len = 20

In [38]: ids_toy = np.random.randint(low=0, high=14, size = seq_len_toy*ids_toy_len)
         ids_toy.shape = (ids_toy_len, seq_len_toy)

In [39]: ids_toy = ids_toy.astype(np.int32, copy=False)

In [40]: y_toy = [int(round(np.random.random_sample(1)[0])) for i in range(ids_toy_len)]

In [41]: hidden_toy = (torch.zeros(1, batch_size_toy, n_lstm_units_toy),
                       torch.zeros(1, batch_size_toy, n_lstm_units_toy))

In [42]: hidden_toy = (torch.tensor(hidden_toy[0], dtype=torch.float64),
                       torch.tensor(hidden_toy[1], dtype=torch.float64))
```

```
In [43]: embeddings_toy = nn.Embedding.from_pretrained(weights_matrix_toy)

In [44]: _, emb_dim_toy = weights_matrix_toy.shape

In [45]: emb_dim_toy

Out[45]: 4

In [46]: lstm_toy = nn.LSTM(input_size=emb_dim_toy,
                             hidden_size=n_lstm_units_toy,
                             dropout=1-keep_prob_toy,
                             batch_first=True).double()

/home/aritz/anaconda3/envs/mypy36/lib/python3.6/site-packages/torch/nn/modules/rnn.py:38: UserWa
  "num_layers={}".format(dropout, num_layers))


In [47]: hidden2bin_toy = nn.Linear(n_lstm_units_toy, 2).double()

In [48]: #logsoft_toy = nn.LogSoftmax(dim=1)

In [49]: loss_fcn_toy = nn.CrossEntropyLoss()
```

Next we try to reproduce what the `forward function` would do on our toy data, with this setting:

```
In [50]: inp_toy = torch.from_numpy(np.int64(ids_toy[:batch_size_toy]))

In [51]: print(inp_toy)

tensor([[ 12,   6,   9],
        [  8,  12,  11]])


In [52]: emb_vect_toy = embeddings_toy(inp_toy)

In [53]: _, hidden_toy = lstm_toy(emb_vect_toy, hidden_toy)

In [54]: print(hidden_toy[0].shape)

torch.Size([1, 2, 7])


In [55]: logits_toy = hidden2bin_toy(hidden_toy[0].view(hidden_toy[0].shape[1],hidden_toy[0].sha

In [56]: logits_toy.shape

Out[56]: torch.Size([2, 2])

In [57]: torch.tensor([y_toy[0]])

Out[57]: tensor([ 0])
```

```
In [58]: loss_fcn_toy(logits_toy, torch.tensor(y_toy[0:2]))

Out[58]: tensor(0.6919, dtype=torch.float64)
```

Now let's try with the `Dataset`.

```
In [59]: batch_size_toy = 5

In [60]: hidden_toy = (torch.zeros(1, batch_size_toy, n_lstm_units_toy),
                       torch.zeros(1, batch_size_toy, n_lstm_units_toy))
         hidden_toy = (torch.tensor(hidden_toy[0], dtype=torch.float64),
                       torch.tensor(hidden_toy[1], dtype=torch.float64))

In [61]: my_dataset_toy = MyIMDBDataset(ids_toy, y_toy)

In [62]: trainloader_toy = DataLoader(my_dataset_toy, batch_size=batch_size_toy, shuffle=True)

In [63]: for i, data in enumerate(trainloader_toy, 0):
             if i>0:
                 break
             inp, labels = data
             # zero the parameter gradients
             optimizer.zero_grad()
             inp = torch.tensor(inp, dtype=torch.int64)
             emb_vect_toy = embeddings_toy(inp)
             print(type(emb_vect_toy))
             print(emb_vect_toy.shape)
             _, hidden_toy = lstm_toy(
                     emb_vect_toy, hidden_toy)

<class 'torch.Tensor'>
torch.Size([5, 3, 4])
```