Text Classification on Twitter Data in PyTorch

Abdullah Cem Oezbay
Technical University of Munich
cem.oezbay@tum.de

I. Introduction

This report aims to explain how a multiclass text classifier with LSTMs is built using the Pytorch framework. The assignment aims to classify Tweets from eight anonymous celebrities. LSTM stands for Long-Short-Term Memory. LSTM is a type of recurrent neural network. However, it is better in memory than traditional recurrent neural networks. LSTMs perform pretty well due to their ability to memorize specific patterns. In LSTM, each word is not categorized one by one. We can use multiple strings of words to find the class it belongs to in LSTM. Thus, we can be much more efficient when working with Natural language processing (NLP). Suppose we use appropriate embedding and encoding layers in LSTM. In that case, the model will be able to find the true meaning in the input string and give the most accurate output class.

The rest of the report will detail how text classification on Twitter data is done using LSTM.

II. Methodology

A. What is tried to achieve the best model

My LSTM model is defined to train on the Twitter data. First, the input is given through an embedding layer because word embeddings better capture context. In Pytorch, it is used nn.Embedding to create the mentioned layer. The layer takes the word-vector length (embedding dimensions) and size of the vocabulary as input. The vocabulary size is given as number of the tokens plus one. Since padding token is not added into tokens at first, embedding layer would not embed padding token unless we add one more index for padding token. Thus, the vocabulary size is given as number of the tokens plus one. Then word embedding is given into Recurrent Neural Network. This means the output of the embedding layer passed into an LSTM layer using nn.LSTM and the layer defined as rnn in Model. The LSTM layer takes word-vector length (embedding dimensions), length of the hidden rnn vector, and number of recurrent layers as inputs. Since the first element has the size of batch size, the ”batch_first” assign to True. In the last layer, I take the last hidden layer of RNN and pass it into a fully connected linear function to get the final output.

The model mentioned above is used as the baseline for my experiments.

1) Hyperparameter Configurations: In addition to the baseline, the parameters are fixed to make a fair comparison between the models. The first crucial parameter is the learning rate. Networks are trained using the stochastic gradient descent algorithm (SGDM). The weights' amount are updated during training is referred to as the step size or the learning rate". Specifically, the learning rate is a configurable hyperparameter used in the training of neural networks with a small positive value, in our case in the range between 0.01 and 0.000001. Learning rate controls how quickly or slowly a neural network model learns a problem. Precisely, it controls the amount of error that the model weights are updated each time it is updated. So when I choose the learning rate as 0.01, my validation accuracy reaches 38.8 percent in just 3 epochs, but then immediately stops increasing and gradient descent increases the training loss from 2.4 to 2.5 instead of reducing it. However, when I set the learning rate as 0.000001, the model starts to learn very slowly. It reflects on validation accuracy, so even at the 100th epoch, we get an accuracy rate similar to the random accuracy rate, which is 16-17 percent. So I always get constant values. The best accuracy we get is for the learning rate of 0.001. The model learns not too slow or not too fast with this learning rate, so just the correct value for the learning rate hyperparameter is 0.001. From the table below, you observe the results.

TABLE I. Lerning rate selection

learning rate	accuracy
0.01	38.8%
0.001	69%
0.0001	62%
0.000001	16.8%

Now that I have characterized the selection of learning rates, I can introduce some standard techniques for regularizing the model. Weight decay (L2 regularization) is another crucial hyperparameter that is used for regularizing. In the network, there are two parameters that can be regularized. Those are the weights and the biases. The weights directly influence the inference between the inputs and the outputs learned by the model because they are multiplied by the inputs. I chose the most optimal value for weight decay by trial and error, which is 0.000001. This helps prevent the network from overfitting the training data as well as the exploding gradient problem.

2) Adding extra linear layers into Model: As I mentioned above, we now have a baseline model as follows: an embedding layer. The output of the embedding layer passed to the LSTM layer. I take the last hidden layer of RNN and pass it into a fully connected linear function to get the final output. In order to improve the model, first, I added a linear layer, which takes as inputs embedding and rnn dimensions. With the chosen hyperparameters, I get 68.7% accuracy. Next, I added a dropout layer with the probability of 0.5 to prevent overfitting. Unfortunately, that also did not increase the validation accuracy as expected, and I got a validation accuracy of 69.3.

At last, I add another dropout layer. However, this time, I added the dropout layer before the LSTM layer instead of adding after the LSTM layer with the same probability as 0.5. After changing the location of the dropout layer, the accuracy is increased to 71.6%. After these experiments on the additional layer on my baseline model, I realized that I should continue to improve my main model by taking advantage of the model with 71.6% validation accuracy. Thus, later experiments mostly continued with applications built on this model.

TABLE II. ADDING EXTRA LAYERS

embed.	dropout	lstm	dropout	1. fc	dropout	2. fc	accuracy
yes	no	yes	no	yes	no	yes	68.7%
yes	no	yes	yes	yes	no	yes	69.3%
yes	yes	yes	no	yes	no	yes	71.6%

3) Adding activation functions into Model: After the first part of the experiments, I set the second model as the baseline model. So my baseline model is now as follows: first, the embedding layer, then the dropout layer with the probability of 0.5, and the output of the dropout layer passed to the LSTM layer. In order to upgrade my model, I initialized a fully connected linear layer, and layer receives as parameters: embedding dimensions and rnn dimensions which refers to the input and output dimension, respectively. Then this linear layer passed to the sigmoid activation function. At last, I take the last hidden layer of the previous output and pass it into a fully connected linear function to get the final output. In the end, I get an accuracy of 72.4%.

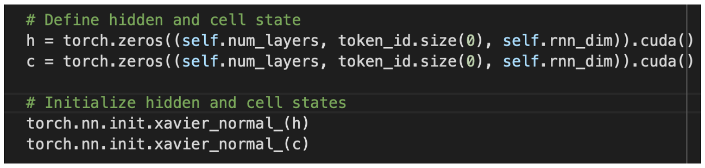
This time the model learned slower than the models without any activation function. That is why I tried to help the model learn faster by increasing the learning rate from 0.001 to 0.01. However, choosing the step size (learning rate) too large, the network outputs constant values. While I was searching for the reason for this issue, I found out that while using saturating nonlinearities (i.e., bounded activation functions like Sigmoid), large learning might cause the nonlinearities to saturate, and learning halts this may result in outputting constant values. As a conclusion to that, I return to determine the learning rate as 0.001. and go for a try by changing the sigmoid activation with ReLu.

Related to the previous point, the type of non-linearities in your hidden layers are also crucial. Again, if it's a saturating non-linearity, this may be hindering training. I tried rectifying linear units (ReLUs) which have the form $f(x) = max(0, x)$. They are unbounded, so they do not saturate, and they have a gradient equal to 1 when $x > 0$. They have the interpretation of "activating" when the input is greater than 0. In this sense, they allow the gradient to propagate through. However, this approach did not improve the validation accuracy as expected. At last, I tried to add one more activation function to the second linear layer. Unfortunately, I get the worst validation accuracy at this approach. You observe the results from the table below.

TABLE III. ADDING EXTRA LAYERS AND ACTIVATION FUNCTIONS

dropout	1.fc	activation	dropout	2.fc&activation	train loss	accuracy
yes	yes	Sigmoid	yes	yes(no)	0.12	72.4%
yes	no	ReLu	no	yes(no)	0.20	70.8 %
yes	yes	ReLu	yes	yes(sigmoid)	0.9	52.3%

I looked for more upgrades to my model and came up with the idea that initializing weights is crucial. Proper weight initialization allows gradients to backpropagate through the network and for learning to occur. Now, I initialize weights in the network. As an initialization scheme, I used the so-called "Xavier Glorot Initialization". I trained the same models from the experiment above, but this time with implementing Xavier Glorot Initialization. During implmentation of "Xavier Glorot Initialization", I got the error "RuntimeError: Input and hidden tensors are not on the same device, found input tensor at cuda:0 and hidden tensor at cpu". I moved the hidden tensor to cuda to fix this bug. After this effort, unfortunately, the results were not improved much enough.



```
# Define hidden and cell state
h = torch.zeros((self.num_layers, token_id.size(0), self.rnn_dim)).cuda()
c = torch.zeros((self.num_layers, token_id.size(0), self.rnn_dim)).cuda()

# Initialize hidden and cell states
torch.nn.init.xavier_normal_(h)
torch.nn.init.xavier_normal_(c)
```

Fig. 1. Initializating Weights by Xavier Glorot Initialization

.

Faced with more features than examples, linear models tend to overfit. That's why, until now I add dropout layers in between various layers of the model, but this time, I tried again with the same layers. However, I removed one dropout layer and reduce the droupout probability from 0.5 to 0.3 for the only droupout layer. Since the dropout layer ensures some number of layer outputs are randomly ignored or dropped out during training, dropping too many layers obstructs model to learn properly. That is why, I go with only one droupout layer with the probability of 0.3. This method provides one of the best accuracy which is 75.8% and train loss of 0.012. But even though I observed a similar improvement on the other two trials, I did not get sufficient validation accuracy. The results given in the following table.

TABLE IV. VALIDATION ACCURICIES AFTER WEIGHT INITIALIZATION

dropout	LSTM	1.fc	activation	dropout	2.fc & activation	accuracy
yes	yes	yes	Sigmoid	no	yes(no)	75.8%
yes	yes	yes	ReLu	no	yes(no)	70.6 %
yes	yes	yes	ReLu	no	yes(sigmoid)	63.4%

4) Model with pre-trained Glove word-vectors: Untill now, I used the provided word embeddings. However, I also tried to use pre-trained Glove word vectors instead of training provided word embeddings to examine the variation on validation accuracy. The pre-trained Glove word vectors have been trained before and probably have greatest abilty for capturing context. Without changing the best model that I defined, I get validation accuracy as 70.7. As a conclusion, using pre-trained Glove word vectors did not help much for the our tweet classification task.

TABLE V. MODEL WITH PRE-TRAINED GLOVE WORD-VECTORS

embedding	dropout	LSTM	accuracy
yes	p = 0.3	yes	70.7%

B. What is your best model/method

In the previous section, I have shown different approaches that can be taken to address the text classification problem from different perspectives.

Now, I'll introduce the architecture to solve the tweet classification assignment. I have defined so-called "init" and "forward" functions. Let me explain the use case of both of these functions. In the "init" function, I defined all the layers that will be used in the model. In the "forward" function, I defined the forward pass of the inputs.

From now on, let's understand details about the different layers that I used for building the architecture and their parameters. So let me introduce the essential layers of the best model architecture. The nn module from the torch is a base model for all models, so each model is a subclass of the nn module. In the first layer, the embedding layer is initialized. The embedding layer is crucial for implementing NLP because it represents words in numerical forms. The embedding layer transforms the array of integers into a dense vector. It receives as parameters: input size, which refers to the size of the vocabulary plus one. Since padding token is not added into tokens at first, embedding layer would not embed padding token unless I add one more index for padding token. That is why the vocabulary size is given as the number of the tokens plus one. As the second input, it takes embedding dimensions, which refers to word-vector length. The output of the embedding layer passed into the dropout layer to prevent overfitting. The dropout probability is determined as 0.3. Since we now do not have a significant probability of dropping out, the model can learn properly. Next, the LSTM layer is initialized. It receives as input parameters: input size, which refers to the dimension of the embedded token; recurrent neural network size, which refers to the dimension of the hidden and cell states, number of layers which refers to the number of stacked LSTM layers and batch first which refers to the first dimension of the input vector. At last, the output of the LSTM layer passed into the fully connected linear layer, and the layer receives as input the last hidden state of the output of the LSTM layer.

TABLE VI. Best Accuracy

embedding	dropout	LSTM	train loss	accuracy
yes	p = 0.3	yes	0.00721	77.52%

For text classification, I have implemented LSTM neural network as the baseline of my Model, and I implemented the model in PyTorch as the framework. I defined the learning rate as 0.001 and weight decay as 0.0000001 according to the table below. If I would choose learning rate as 0.01 then the validation accuracy stucked at 36.79%.

I compiled the model using the Adam optimizer. Adam optimizer is the best optimizer available to handle sparse gradients and noisy issues.

In following section, I'll describe implementation details for tweet classification.

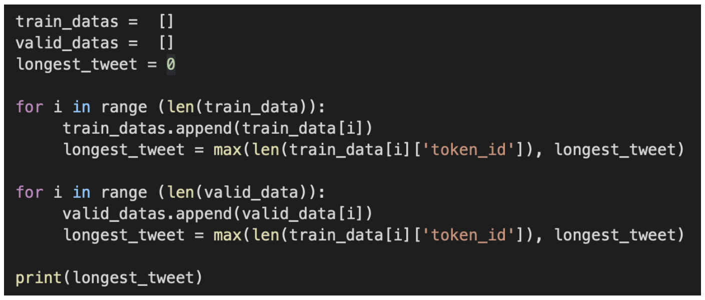
TABLE VII. Best Accuracy

learning rate	weight decay	train loss	accuracy
0.01	1e-7	1.83	36.79%
0.001	1e-7	0.00721	77.52%
0.0001	1e-7	2.09	17.7%
0.000001	1e-7	2.3	13.1%

C. Implementation details

This section will examine the implementation for our task in detail.

I want to start by explaining the padding that I implemented in order to batch the dataset. Padding is adding an extra token called a padding token at the beginning or end of the sentence. As the number of words in each sentence varies, we convert the variable length input sentences into sentences with the same length by adding padding tokens. I added a padding token at the end of the sentences in our case. In order to implement padding, first, I defined the tweet with the most words, as is shown in the Figure below. Created an empty array for training and validation data and initialized the integer value of the longest tweet as zero. Then, start to compare the number of words for all sentences in the training set and save the one with maximum words as the longest tweet. The same is applied to the validation set. In the end, the sentences with the most word are determined as 60.

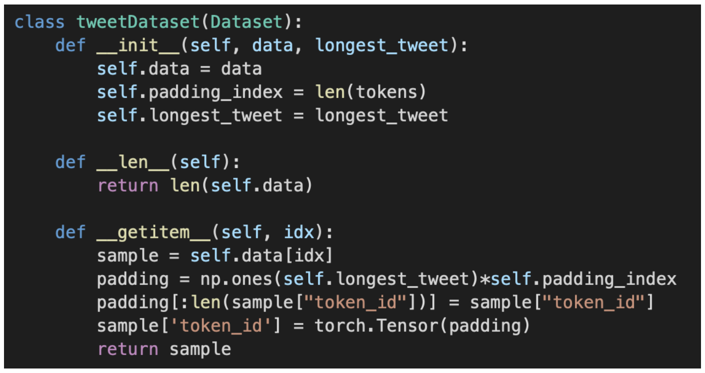


```
train_datas =  []
valid_datas =  []
longest_tweet = 0

for i in range (len(train_data)):
     train_datas.append(train_data[i])
     longest_tweet = max(len(train_data[i]['token_id']), longest_tweet)

for i in range (len(valid_data)):
     valid_datas.append(valid_data[i])
     longest_tweet = max(len(train_data[i]['token_id']), longest_tweet)

print(longest_tweet)
```

Fig. 2. Determine Longest Tweet

In the next part, I defined padding array consist of the "padding_index" value, which is the number of tokens. Then, values changed from the beginning of the array for all sentences. The remaining values remained as "padding_index" value, which is 13369. In the end, we obtain the same length for all tweets. To sum up, sequences that do not meet the required sequence lengths are completed with padding_idx.

The final best model has a total number of parameters of 13153288, which is below 20 million. Torch package is used to define tensors and mathematical operations, but there is no additional package implemented. However, I considered implementing TorchText, a Natural Language Processing (NLP) library in PyTorch. This library contains the scripts for preprocessing text and the source of a few popular NLP datasets. However, the preprocessing and tokenization were already done. That is why I did not find it necessary to import TorchText.



```python
class tweetDataset(Dataset):
    def __init__(self, data, longest_tweet):
        self.data = data
        self.padding_index = len(tokens)
        self.longest_tweet = longest_tweet

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        sample = self.data[idx]
        padding = np.ones(self.longest_tweet)*self.padding_index
        padding[:len(sample["token_id"])] = sample["token_id"]
        sample['token_id'] = torch.Tensor(padding)
        return sample
```

Fig. 3. Dataset Definition

III. Conclusion

To conclude, I can make some inferences regarding the assignment. The aim was classifying the tweets, in order to classify tweets efficiently, I made various experiments with various models. I tried to change learning rate or weight decay. Afterwards, I tried to play with the model by adding droupouts, activation functions, fully connected layers. I tried to use pre-trained ”GloVe” word-vector or implemented weight initialization. However, I got the best validation accuracy with the one of the most plain model,which is equels to 77.58%. Actually, result is quite decent because we have 8 classes and random guessing model would have achieved only 12% accuracy. From this I can infer that my model has learned quite well.