Great job!

Pass

Emotion Classification from Three-turn Conversation Data using Variations of RNN Layers with Attention

report

0. Abstract

This report implements a simplified NELEC model proposed by a research team from Microsoft as a part of the third task of 'Contextual Emotion Detection inText' as part of SemEval-2019 (Agrawal & Suri, 2019). The model uses neural embeddings from GloVe and Emoji2Vec to encode the corpora. The model features in LSTM with attention layer, GRU with attention layer, global max-pooling layer and global average pooling layer to train and regulate the weights and bias in the deep neural network. The code is posted on the google colab file.

1. Introduction

Sentiment analysis of contextual data has been known for classifying movie reviews, product reviews and customer satisfaction. The popular application of sentiment analysis has focused on a binary-positive-or-negative classification or a binary classification with a third, neutral class. Emotion classification with specific emotion categories, such as happy, sad, angry, etc., has gained popularity in recent years. Being able to detect emotions, the development of an emotionally intelligent agent will move one step closer to a fully automated chatbot with which humans can communicate emotions. During daily social interactions, humans perceive body language and facial expressions along with the conversation to reveal the true emotions. Lack of contextual cues creates ambiguity to both humans and machines. In modern days, people have leveraged emojis and face text to unveil their emotions and intensity of their emotions. For machine learning tasks, a combined network of emojis and texts have shown significant improvement on recognizing sarcasm (Subramanian, Sridharan, Shu, & Liu, 2019).

In this report, I included a pre-trained emoji vector with the text data embedding and implemented a neural deep learning model that includes a Gated Recurrent Unit, Long Short Term Memory and an Attention Mechanism. The model yields an F1-score of 68%.

2. Preprocessing

2.1 Data analysis

The EmoContext data provided by Microsoft contains 5 columns with unique ID, turn 1, 2, 3 and emotion context label (happy, sad, angry and others). The latter turn is the response to the former. The conversations include typos, abbreviations and emojis which are a truthful representation of common text messages. It's not hard to see from the table below that the class "Others" take a big portion of the distribution of categories. Since the "Others" category

is not important for evaluation, it's been excluded. The percentage of the "Others" class is 1 - the rest of the class.

	Train (%)	Test (%)	Dev (%)
Happy	14.2	13.3	0.53
Sad	18.0	18.6	0.45
Angry	18.2	18.6	0.54
Emoji	17.5	17.9	11.1
Data Coverage	99.1	98.9	99.2

Table 1: Global Statistics of the Train, Test and Dev Data

2.2 Text Preprocessing

The conversations are first cleaned with a regular expression program to revert the abbreviations and typos back to the unabbreviated and correct form. Unlike typical data processing procedures, lamentation and normalizations (remove of question marks and stop words) are not used because a paper suggests that the unlamentized words are a better representation of the magnitude of emotions (Agrawal & Suri, 2019). After cleaning, I tokenized the word and padded the tokenized sentences to equal length vectors.

After the data cleaning step, the text data needs to be encoded into numerical embedding to pass into a model. Embeddings can be created through different methods; the most popular ones are Bag-of-Words (BoW) and Term Frequency - Inverse Term Frequency (TF-IDF), which represents the word count or the relative frequency of each word. Those measures have difficulty revealing the conditional probability of the appearance of one word given that a certain other word is present. The Global Vectors for Word Representation (GloVe) provides pre-trained embeddings: each row represents a word and each column represents the trained weight in the hidden layer. The embeddings used models are trained with 6 billion tokens, 400,000 words in 300 dimensions. The vectors are essentially the factorization of the co-occurrence probability of the words, and the GloVe is a log-bilinear model with a weighted least squares objective function to calculate such vectors. The assumption that the co-occurrence probability of words uncovers the context of the words and its relationship with other words has been relatively accurate when the model is trained on a large dataset. The co-occurrence probability takes into account the local context also the global statistics and has been successful on word analogy task; it's accuracy bypasses the Word2vec model by more than 10% and is more robust to more training cycles (Pennington, Socher, & Manning, 2014).

Aside from the GloVe embedding, the Emoji2Vec embedding is used to capture the meanings of emojis. The Emoji2Vec embedding is in the same shape of the GloVe embedding: each

row represents a word, and each column is the value of the weights in the hidden layer. The Emoji2Vec model collects the sum of the Word2Vec embedding vectors of the description of the emoji, and trains a sigmoid function of the sum and a trainable vector x through a logistic loss function. Similar to the popular Word2vec model, it is able to generate patterns like 👑-👔+👔=1 : 👰; 2 : 👑 ; 3 : 🏰; 4 : 💂♀; 5 : 🐓. The augmented Word2Vec and Emoji2Vec has reported significant improvement on classification performance on the full corpus and tweets that contain emojis (Eisner, Rocktaschel, Augenstein, Bosnjak, & Riedel, 2016).

The pretrained embeddings are like word dictionaries that we store the rearranged word vectors to a matrix where the index of each row maps to the index of the tokenized data.

3. Construct the Model

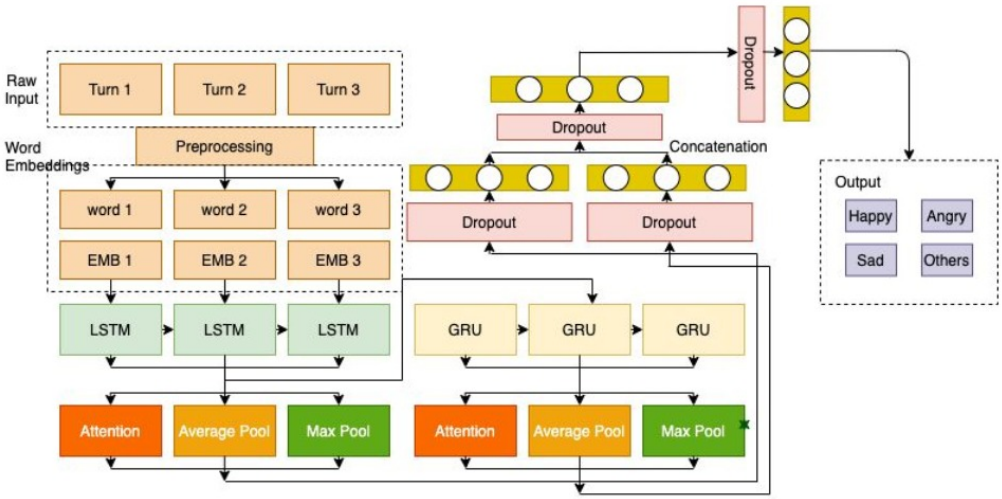


Figure 1: Model Diagram

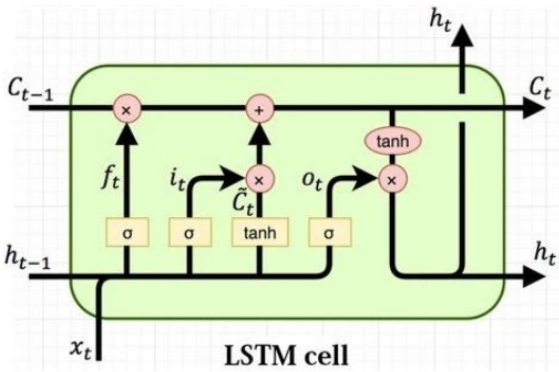
The model I trained is a simplified NELEC model which includes a Long Short Term Memory (LSTM) layer, a Gated Recurrent Unit (GRU) layer and a feed-forward attention layer (Agrawal & Suri, 2019). After the words in each turn are cleaned and tokenized and the embedding is created. We pass in word tokens as the input layer, and the embedding as the embedding layer. The output of the embedding layer will be passed in a series of LSTM, GRU, dropout and attention layers. The LSTM and GRU cells are variations of RNN cells. An RNN allows a feedforward neural network to carry information from the previous state, but it fails to reveal the long-term dependencies when the input sequence becomes long. This is because RNN is prone to vanishing gradients, which is a term to describe the gradients becoming smaller and insignificant and the weight changes becoming insignificant. In the world of Natural Language Processing (NLP), long corpora as input are common. The LSTM and the GRU to preserve the information.

LSTM Cell

Each Long Short Term Memory cell adds a forget gate, an input gate and an output gate, which are designed to decide what information to keep, what information is relevant to the current step and what information to pass to the next hidden state. The value from the hidden layer in the previous state $h_{t-1}W^f$ and current input x_tU^f are added and pass through a sigmoid activation function. The closer the result is to 0, the more likely the information will be "forgotten". This is called the forget gate.

The previous hidden state $h_{t-1}W^i$ and the current input x_tU^i also passes through a sigmoid function and a tanh function. The results of the two functions are multiplied. The closer the result is to 0, the less important is the current information. This operation is called the input gate.

The results from the forget gate will be pointwise multiplied with the value from the previous cell state C_{t-1} and added to the result of the input gate, whose result will be passed to a tanh function and combine with the result of $sigmoid(h_{t-1}W^o + x_tU^o)$. The W^* and U^* are the weight matrices.



$$i_t = \sigma\left(x_t U^i + h_{t-1} W^i\right)$$
$$f_t = \sigma\left(x_t U^f + h_{t-1} W^f\right)$$
$$o_t = \sigma\left(x_t U^o + h_{t-1} W^o\right)$$
$$\tilde{C}_t = \tanh\left(x_t U^g + h_{t-1} W^g\right)$$
$$C_t = \sigma\left(f_t * C_{t-1} + i_t * \tilde{C}_t\right)$$
$$h_t = \tanh(C_t) * o_t$$

Figure 2: LSTM Cell

GRU Cell

The GRU cell, on the other hand, has a reset gate acting like the forget gate and a reset gate acting like the combination of input and output gates.

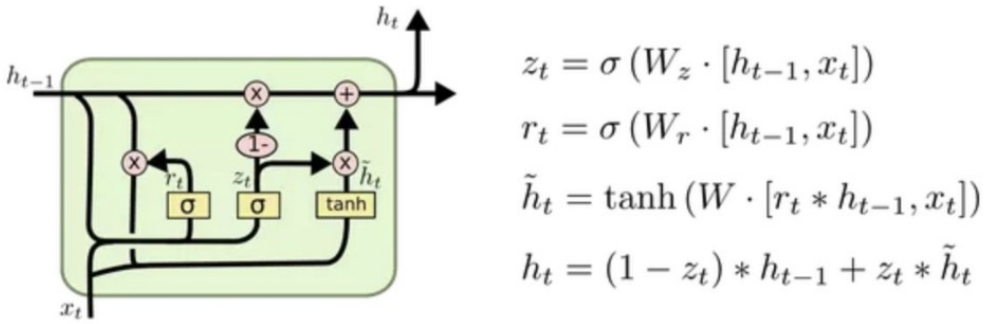


Figure 3: GRU Cell

The vanishing gradient issue is far less prominent because the multiplications used for RNN cells have been replaced by addition in many operations. During backward propagation, we are less likely to have gradients that are too small, which would stall the training.

After embedding, the final status of the hidden state of the LSTM model as the output is splitted into two directions. One goes to an attention layer, a global max pooling layer and global average layer in parallel. The other one goes to a GRU layer whose output will feed into an attention layer, a global max pooling layer and a global average layer in parallel. The attention cell here is the feed-forward attention proposed by Raffle and Ellis (Raffel & Ellis, 2016).

3.2 Feed-forward Attention

The attention mechanism is essentially an extra vector storing the importance weights of each token, which allows for a more direct dependency between the state of the model at various time steps. The Attention layer used in this model is a simplified version of the first Bahdanau's attention model. The Bahdanau's attention layer takes in the hidden state h_t of the LSTM model as input and feeds it into a trainable function $a(s_{t-1}, h_t)$. Each time, the weight of a is calculated by the softmax of the h_j and the previous state s_{t-1} (Yang et al., 2016). The feed-forward attention layer in the NELEC model, however, is a simplified version of Bahdanau's attention model. It does not include the previous state s_{t-1} while calculating the values for a. The final output vector c is computed as a weighted average of h_t for all the time t whose weights are given by a. This variation of the attention layer can handle variable sequence length and allows for parallel processing, but it takes more epochs to train (Raffel & Ellis, 2016). In the actual implementation, I found the advantages do not overweigh the disadvantage, so I decided to use the original implementation.

4 Model Result Analysis

The original model was implemented in Keras and I reimplemented it with Pytorch. I found a Pytorch deep learning object, nn.modules, that is easier to debug and requires me to calculate

the dimensions of the layers, which help me understand the flow of the tensors. The training script is also more flexible for customization. The model is trained on 24128 rows of conversation, 300 dimensions, 35 maximum sequence length. The training procedure uses cyclic learning rate trained on 100 epochs (Smith, 2017).

	Keras Model (Original)	**Keras Model (Trained)**	**Pytorch Model (Implemented)**
Class Happy	Precision : 0.508 Recall : 0.694 F1 : 0.586	Precision : 0.512 Recall : 0.754 F1 : 0.610	Precision : 0.768 Recall : 0.458 F1 : 0.574
Class Sad	Precision : 0.496 Recall : 0.752 F1 : 0.598	Precision : 0.599 Recall : 0.800 F1 : 0.685	Precision : 0.547 Recall : 0.784 F1 : 0.645
Class Angry	Precision : 0.546 Recall : 0.799 F1 : 0.649	Precision : 0.517 Recall : 0.893 F1 : 0.655	Precision : 0.568 Recall : 0.867 F1 : 0.686
Macro Precision, Recall and F1	Precision : 0.5165 Recall : 0.748 F1 : 0.6111	Precision : 0.5427 Recall : 0.8156 F1 : 0.6518	Precision : 0.5244 Recall : 0.8061 F1 : 0.6354
Micro Precision, Recall and F1	Precision : 0.5179 Recall : 0.7488 F1 : 0.6123	Precision : 0.5370 Recall : 0.8177 F1 : 0.6483	Precision : 0.5217 Recall : 0.8082 F1 : 0.6341

Table 2: Model Results

4.1 Conclusions and Improvements

The pytorch model has reached a similar precision, recall and F1 scores as the model shown in the NELEC github and also the retrained Keras model. The ratio in Table 1 suggests the data imbalance in the sample, where the “Others” Class outnumbers of the other class. A few winning models for this contest have chosen the multitask models. For instance, the ANA model proposed by a research team from University of Alberta is a multi-head hierarchical model, the probability of a text sequence belonging to one class is calculated separately. It doesn’t seem to completely solve the data imbalance problem, however, as they also suggest using a binary classifier for the “Others” and “Not Others” class to help with training (Huang, & Zaıan, 2019). The other solution is to feature-engineer from more traditional NLP methodology. The final NELEC model in the paper has also used the combination of neural features (scores from neural classifiers) and lexical features (Turn Wise Word n-Grams, etc).

Reference

Agrawal, P., & Suri, A. (2019, April 05). NELEC at SemEval-2019 Task 3: Think twice before going deep. Retrieved March 24, 2020, from https://arxiv.org/abs/1904.03223

Chatterjee, A., Narahari, K., Joshi, M., & Agrawal, P. (2019, June). SemEval-2019 task 3: Emocontext Contextual emotion detection in text. Retrieved March 25, 2020, from https://www.aclweb.org/anthology/S19-2005/

Eisner B., Rocktaschel T., Augenstein I., Bosnjak M., & Riedel S. (2016, November 20). Emoji2Vec: Learning emoji representations from their description. Retrieved March 25, 2020, from https://arxiv.org/abs/1609.08359

Huang C., A. T., & Zaıan, O. R. (2019, May 31). ANA at SemEval-2019 Task 3: CONTEXTUAL emotion detection in conversations through Hierarchical LSTMs and BERT. Retrieved March 26, 2020, from https://arxiv.org/abs/1904.00132

Pennington, J., Socher, R., & Manning, C. (2014, October). Glove: Global vectors for word representation. Retrieved March 24, 2020, from https://www.aclweb.org/anthology/D14-1162/

Raffel, R., & Ellis, D. P. (2016, March 28). Feed-forward networks with attention can solve some long-term memory problems. Retrieved March 25, 2020, from https://arxiv.org/abs/1512.08756v4

Smith, L. N. (2017, April 04). Cyclical learning rates for training neural networks. Retrieved March 25, 2020, from https://arxiv.org/abs/1506.01186

Subramanian, J., Sridharan, V., Shu, K., & Liu, H. (2019, July 09). Exploiting emojis for sarcasm detection. Retrieved March 24, 2020, from https://link.springer.com/chapter/10.1007/978-3-030-21741-9_8

Yang Z., Yang D., Dyer C., He X., Smola A., & Hovy E. (2016, June). Hierarchical attention networks for document classification. Retrieved March 25, 2020, from https://www.aclweb.org/anthology/N16-1174/

```
# import sys
# print (sys.version)
# conda activate py36
```

```
from google.colab import drive
drive.mount('/content/drive', force_remount=False)
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=

Enter your authorization code:
··········
Mounted at /content/drive
```

```
# import pandas as pd
# from sklearn.model_selection import train_test_split

# df = pd.read_csv("./data/data/train.txt", sep="\t")
# train, test = train_test_split(df, test_size=0.2, random_state=42)

# train.to_csv("./data/data/splitted_train.txt", index=False, sep="\t")
# test.to_csv("./data/data/splitted_test.txt", index=False, sep="\t")
```

Data Preparation

```
!pip install emoji
```

```
Collecting emoji
  Downloading https://files.pythonhosted.org/packages/40/8d/521be7f0091fe0f2ae690
     |████████████████████████████████| 51kB 2.6MB/s
Building wheels for collected packages: emoji
  Building wheel for emoji (setup.py) ... done
  Created wheel for emoji: filename=emoji-0.5.4-cp36-none-any.whl size=42176 sha2
  Stored in directory: /root/.cache/pip/wheels/2a/a9/0a/4f8e8cce8074232aba240caca
Successfully built emoji
Installing collected packages: emoji
Successfully installed emoji-0.5.4
```

```
import emoji
```

```
import io
import sys
import importlib
import re
from keras.utils import to_categorical
```

```
import numpy as np
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
sys.path.append('/content/drive/My Drive/EmoContext')
import regex_nlp
import pickle
import gensim.models as gsm

sys.path.append('/content/drive/My Drive/EmoContext')
import utils
from utils import CyclicLR,getMetrics,microF1Loss
```

```
Using TensorFlow backend.
```

The default version of TensorFlow in Colab will switch to TensorFlow 2.x on the 27th of March, 2020.
We recommend you upgrade now or ensure your notebook will continue to use TensorFlow 1.x via the %tens

```
Loading utils module
Loading utils module
```

```
train_path = "/content/drive/My Drive/EmoContext/splitted_train.txt"
test_path  = "/content/drive/My Drive/EmoContext/splitted_test.txt"
dev_path   = "/content/drive/My Drive/EmoContext/dev.txt"

lab2emo = {0:"others", 1:"happy", 2: "sad", 3:"angry"}
emo2lab = {"others":0, "happy":1, "sad":2, "angry":3}

NUM_CLASSES = 4                 # Number of classes - Happy, Sad, Angry, Others
MAX_NB_WORDS = 15000            # To set the upper limit on the number of tokens extra
MAX_SEQUENCE_LENGTH = 35        # All sentences having lesser number of words than thi
EMBEDDING_DIM = 300
```

Load data from pickle files

```
test = pickle.load( open( "/content/drive/My Drive/EmoContext/test.pickle", "rb" ) )
train = pickle.load( open('/content/drive/My Drive/EmoContext/train.pickle', 'rb') )
dev = pickle.load(open('/content/drive/My Drive/EmoContext/dev.pickle', 'rb') )

testTexts = test["testTexts"]
rawtestTexts = test["rawtestTexts"]
testLabels = test["testLabels"]

trainTexts = train["trainTexts"]
rawtrainTexts = train["rawtrainTexts"]
labels = train['labels']

evalTexts = dev['evalTexts']
rawevalTexts = dev['rawevalTexts']
evalLabels = dev['evalLabels']
# test = {'testTexts':testTexts, 'rawtestTexts': rawtestTexts, 'testLabels': testLabel
```

```
# train= {'trainTexts':trainTexts, 'rawtrainTexts': rawtrainTexts, 'labels': labels}
# dev  = {'evalTexts': evalTexts, 'rawevalTexts': rawevalTexts, 'evalLabels': evalLabe
```

```
np.unique(np.argmax(test["testLabels"], axis=1), return_counts=True)[1]/len(test["test
```

```
array([0.49386605, 0.1344496 , 0.18567639, 0.18600796])
```

```
np.unique(np.argmax(dev["evalLabels"], axis=1), return_counts=True)[1]/len(dev["evalLa
```

```
array([0.84863884, 0.05154265, 0.04537205, 0.05444646])
```

```
np.unique(np.argmax(train["labels"], axis=1), return_counts=True)[1]/len(train["labels
```

```
array([0.49606267, 0.14224138, 0.17999834, 0.18169761])
```

Or recreate the data file. Please note that since I didnt get the test file from EmoContext, I splitted the

Data Cleaning

```
def preprocess(file_path, mode="train"):
    # importlib.reload(sys.modules['regex'])
    ind = []
    convs = []
    raw = []
    labs = []

    first_line = True
    with open(file_path, encoding="utf8") as f:
        for l in f:
            if first_line:
                first_line = False
                continue

            repeatedChars = ['.', '?', '!', ',']
            for c in repeatedChars:
                l_split = l.split(c)
                while True:
                    try:
                        l_split.remove('')
                    except:
                        break
                cSpace = ' ' + c + ' '
                l = cSpace.join(l_split)

            line = l.strip().split("\t")
            raw_conv = ' '.join(line[1:-1])

            # print (line)
```

```
                if mode == "train":
                    labs.append(emo2lab[line[-1]])

                processed_conv = ' <eos>'.join(line[1:-1])

                # remove duplicated space
                processed_conv = re.sub(r'\ +', ' ', processed_conv)

                # remove not useful punctuation
                processed_conv = re.sub(r'-|-|\^|:|;|#|\)|\(|\*|=|\\|/', '', processed_cor

                # replace abbreviations and mispelled words
                processed_conv = regex_nlp.cleanText(processed_conv.lower(), remEmojis=1).

                convs.append(processed_conv)
                raw.append(raw_conv)

        if mode == "train":
            return ind, convs, raw, labs
        else:
            return ind, convs, raw


print("Processing test data...")
testIndices, testTexts, rawtestTexts, testLabels = preprocess(test_path)
testLabels = to_categorical(np.asarray(testLabels), NUM_CLASSES)
print("Processing training data...")
trainIndices, trainTexts, rawtrainTexts, labels = preprocess(train_path)
labels = to_categorical(np.asarray(labels), NUM_CLASSES)
print("Processing evaluation data...")
evalIndices, evalTexts, rawevalTexts, evalLabels = preprocess(dev_path)
evalLabels = to_categorical(np.asarray(evalLabels), NUM_CLASSES)
```

```
Processing test data...
Processing training data...
Processing evaluation data...
```

```
# Save the temporary data
test = {'testTexts':testTexts, 'rawtestTexts': rawtestTexts, 'testLabels': testLabels}
train= {'trainTexts':trainTexts, 'rawtrainTexts': rawtrainTexts, 'labels': labels}
dev  = {'evalTexts': evalTexts, 'rawevalTexts': rawevalTexts, 'evalLabels': evalLabels


with open("/content/drive/My Drive/EmoContext/test.pickle", 'wb') as f:
  pickle.dump(test, f)

with open('/content/drive/My Drive/EmoContext/train.pickle', 'wb') as f:
  pickle.dump(train, f)

with open('/content/drive/My Drive/EmoContext/dev.pickle', 'wb') as f:
```

```
    pickle.dump(dev, f)
```

Tokenize Data

```
print("Extracting tokens...")
tokenizer = Tokenizer(num_words=MAX_NB_WORDS, oov_token='<unk>')
tokenizer.fit_on_texts(trainTexts + testTexts + evalTexts)
train_sequences = tokenizer.texts_to_sequences(trainTexts)
test_sequences = tokenizer.texts_to_sequences(testTexts)
eval_sequences = tokenizer.texts_to_sequences(evalTexts)

wordIndex = tokenizer.word_index
print("Found %s unique tokens." % len(wordIndex))
```

```
Extracting tokens...
Found 16398 unique tokens.
```

```
lens = [len(x) for x in train_sequences]
print("Mean length for train data", np.mean((np.array(lens))))
print("Train-data Coverage (cutoff length):", np.sum(np.array(lens) <= MAX_SEQUENCE_LI
print()
lens = [len(x) for x in test_sequences]
print("Mean length for dev data", np.mean((np.array(lens))))
print("Dev-data Coverage (cutoff length):", np.sum(np.array(lens) <= MAX_SEQUENCE_LEN(
print()
lens = [len(x) for x in eval_sequences]
print("Mean length for test data", np.mean((np.array(lens))))
print("Test-data Coverage (cutoff length):", np.sum(np.array(lens) <= MAX_SEQUENCE_LEI
```

```
Mean length for train data 15.691312997347481
Train-data Coverage (cutoff length): 0.9905918435013262

Mean length for dev data 15.743534482758621
Dev-data Coverage (cutoff length): 0.9893899204244032

Mean length for test data 15.32994555353902
Test-data Coverage (cutoff length): 0.9920145190562614
```

```
def split_into_three(texts, tknzr):
    middle, left, right = [], [], []
    for text in texts:
        l, m, r = text.split(' <eos>')
        middle.append(m)
        left.append(l)
        right.append(r)
    tokenize = lambda x: tknzr.texts_to_sequences(x)
    return (tokenize(left), tokenize(middle), tokenize(right))


for i in range(10):
```

```
print (u"Original:{}".format(rawevalTexts[i]))
print (u"Processed:{}".format(evalTexts[i]))
print ("\n")
```

```
Original:Then dont ask me YOURE A GUY NOT AS IF YOU WOULD UNDERSTAND IM NOT A GUY
Processed:then do not ask me <eos>youre a guy not as if you would understand <eos


Original:Mixed things  such as ?  the things you do .  Have you seen minions ?
Processed:mixed things such as ? <eos>the things you do . <eos>have you seen mini


Original:Today I'm very happy and I'm happy for you ❤ I will be marry
Processed:today iam very happy <eos>and iam happy for you ❤ <eos>i will be marry


Original:Woah bring me some left it there oops Brb
Processed:woah bring me some <eos>left it there oops <eos>be right back


Original:it is thooooo I said soon master .  he is pressuring me
Processed:it is though <eos>i said soon master . <eos>he is pressuring me


Original:Wont u ask my age ?  hey at least I age well !  Can u tell me how can we
Processed:wont you ask my age ? <eos>hey at least i age well ! <eos>can you tell


Original:I said yes What if I told you I'm not ?  Go to hell
Processed:i said yes <eos>what if i told you iam not ? <eos>go to hell


Original:Where I ll check why tomorrow ?  No I want now
Processed:where i ll check <eos>why tomorrow ? <eos>no i want now


Original:Shall we meet you say- you're leaving soon . anywhere you wanna go befor
Processed:shall we meet <eos>you say you are leaving soon . anywhere you wanna go


Original:Let's change the subject I just did it  . l .  You're broken
Processed:let us change the subject <eos>i just did it . l . <eos>you are broken
```

```
train_l, train_m, train_r = split_into_three(trainTexts, tokenizer)
test_l, test_m, test_r = split_into_three(testTexts, tokenizer)
eval_l, eval_m, eval_r = split_into_three(evalTexts, tokenizer)


train_all = train_sequences
test_all = test_sequences
eval_all = eval_sequences
```

Create Embeddings

- Create embeddings from GloVe and ELMo
- Re-write the function as you see fit
- pad the data to equal length

```
def make_embedding(word_ind, EMBEDDING_DIM):
    embedding_idx = {}
    with open('/content/drive/My Drive/EmoContext/glove.6B.{}d.txt'.format(EMBEDDING_I
        for l in f:
            info = l.strip().split()
            embedding_idx[info[0]] = np.array(info[1:], dtype='float32')

    oov = []
    oov_indices = []
    count = 0
    total = 0
    embedding_matrix = np.zeros((len(word_ind) +1, EMBEDDING_DIM))

    print("Found {} words".format(len(embedding_idx)))

    for word in word_ind:
        if word in embedding_idx:
            ind = word_ind[word]
            embedding_vec = embedding_idx[word]
            embedding_matrix[ind] = embedding_vec
            count += 1
        else:
            oov_indices.append(word_ind[word])
            oov.append(word)
        total += 1

    print("Found embedding for",str((100 * count) / total), "% embeddings")
    return embedding_matrix, oov, oov_indices

def add_emoji_embedding(word_ind, orig_embedding):
    e2v = gsm.KeyedVectors.load_word2vec_format('/content/drive/My Drive/EmoContext/er
    count = 0
    total = 0
    for word in word_ind:
        total += 1
        try:
            ind = word_ind[word]
            embedding_vec = e2v.get_vector(word)
            orig_embedding[ind] = embedding_vec
            # print (word)
            # print (embedding_vec)
            count += 1
        except:
```

```
            continue
        print (count)
        print (total)
        print("Found embedding for", str((100 * count) / total), "% embeddings")
        return orig_embedding


embedding_matrix, oov, oov_indices = make_embedding(wordIndex, EMBEDDING_DIM)
```

```
Found 400000 words
Found embedding for 68.84376143432125 % embeddings
```

```
import emoji

count_emoji_appearances = lambda texts: map(lambda x: len(set(emoji.UNICODE_EMOJI.keys

print("Sentences with emojis for train (%) :", 100 * np.mean(list(count_emoji_appearar
print("Sentences with emojis for dev (%) :", 100 * np.mean(list(count_emoji_appearance
print("Sentences with emojis for test (%):", 100 * np.mean(list(count_emoji_appearance
```

```
Sentences with emojis for train (%) : 17.494197612732094
Sentences with emojis for dev (%) : 17.854774535809018
Sentences with emojis for test (%): 11.143375680580762
```

Load the matrix embedding or generate the matrix embedding

```
embedding_matrix = pickle.load(open("/content/drive/My Drive/EmoContext/embedding_mat.

embedding_matrix = add_emoji_embedding(wordIndex, embedding_matrix)
```

Pad the data

```
train_l = pad_sequences(train_l, maxlen=MAX_SEQUENCE_LENGTH, padding='post', truncatir
train_m = pad_sequences(train_m, maxlen=MAX_SEQUENCE_LENGTH, padding='post', truncatir
train_r = pad_sequences(train_r, maxlen=MAX_SEQUENCE_LENGTH, padding='post', truncatir
train_all = pad_sequences(train_all, maxlen=MAX_SEQUENCE_LENGTH, padding='post', trunc

test_l = pad_sequences(test_l, maxlen=MAX_SEQUENCE_LENGTH, padding='post', truncating=
test_m = pad_sequences(test_m, maxlen=MAX_SEQUENCE_LENGTH, padding='post', truncating=
test_r = pad_sequences(test_r, maxlen=MAX_SEQUENCE_LENGTH, padding='post', truncating=
test_all = pad_sequences(test_all, maxlen=MAX_SEQUENCE_LENGTH, padding='post', truncat

eval_l = pad_sequences(eval_l, maxlen=MAX_SEQUENCE_LENGTH, padding='post', truncating=
eval_m = pad_sequences(eval_m, maxlen=MAX_SEQUENCE_LENGTH, padding='post', truncating=
eval_r = pad_sequences(eval_r, maxlen=MAX_SEQUENCE_LENGTH, padding='post', truncating=
eval_all = pad_sequences(eval_all, maxlen=MAX_SEQUENCE_LENGTH, padding='post', truncat
```

```
print("Shape of training data tensor: ", train_i.shape)
print("Shape of training label tensor: ", labels.shape)
```

```
Shape of training data tensor:  (24128, 35)
Shape of training label tensor:  (24128, 4)
```

```
embedding_mat = {"embedding_matrix":embedding_matrix}
with open("/content/drive/My Drive/EmoContext/embedding_mat.pickle", 'wb') as f:
  pickle.dump(embedding_mat, f)
```

Self-attention Bi-Directional LSTM

```
from keras.layers import average
from keras.regularizers import l1, l2, l1_l2
from keras.layers import RepeatVector
import keras.backend as K

from keras.constraints import MinMaxNorm
from keras.layers import Lambda

from keras.models import Sequential, load_model, Model
from keras.layers import concatenate, Activation, GlobalAveragePooling1D, GlobalMaxPoo
from keras.callbacks import Callback
from keras import initializers, regularizers, constraints, optimizers, layers


class Attention(Layer):
    def __init__(self, step_dim,
                 W_regularizer=None, b_regularizer=None,
                 W_constraint=None, b_constraint=None,
                 bias=True, **kwargs):
        self.supports_masking = True
        self.init = initializers.get('glorot_uniform')

        self.W_regularizer = regularizers.get(W_regularizer)
        self.b_regularizer = regularizers.get(b_regularizer)

        self.W_constraint = constraints.get(W_constraint)
        self.b_constraint = constraints.get(b_constraint)

        self.bias = bias
        self.step_dim = step_dim
        self.features_dim = 0
        super(Attention, self).__init__(**kwargs)

    def build(self, input_shape):
        assert len(input_shape) == 3
```

```
        self.W = self.add_weight((input_shape[-1],),
                                 initializer=self.init,
                                 name='{}_W'.format(self.name),
                                 regularizer=self.W_regularizer,
                                 constraint=self.W_constraint)
        self.features_dim = input_shape[-1]


        if self.bias:
            self.b = self.add_weight((input_shape[1],),
                                     initializer='zero',
                                     name='{}_b'.format(self.name),
                                     regularizer=self.b_regularizer,
                                     constraint=self.b_constraint)
        else:
            self.b = None

        self.built = True

    def compute_mask(self, input, input_mask=None):
        return None

    def call(self, x, mask=None):
        features_dim = self.features_dim

        k.print_tensor(features_dim, "feature_dimension")
        step_dim = self.step_dim
        self.W = K.print_tensor(self.W, self.W.shape, "w=")

        eij = K.reshape(K.dot(K.reshape(x, (-1, features_dim)),
                        K.reshape(self.W, (features_dim, 1))), (-1, step_dim))

        if self.bias:
            eij += self.b

        eij = K.tanh(eij)

        a = K.exp(eij)

        if mask is not None:
            a *= K.cast(mask, K.floatx())

        a /= K.cast(K.sum(a, axis=1, keepdims=True) + K.epsilon(), K.floatx())

        a = K.expand_dims(a)
        weighted_input = x * a
        return K.sum(weighted_input, axis=1)

    def compute_output_shape(self, input_shape):
        return input_shape[0],  self.features_dim
```

```
def buildSingleModel(embeddingMatrix, hidDim=128, maxlen=MAX_SEQUENCE_LENGTH, dp=0.25)
    embeddingLayer = Embedding(embeddingMatrix.shape[0],
                               embeddingMatrix.shape[1],
                               weights=[embeddingMatrix],
                               input_length=MAX_SEQUENCE_LENGTH,
                               trainable=False)

    inp = Input(shape=(maxlen,))
    x = embeddingLayer(inp)
    x = SpatialDropout1D(dp)(x)
    x = LSTM(hidDim, return_sequences=True, dropout=dp, recurrent_dropout=dp)(x)
    y = GRU(hidDim, return_sequences=True, dropout=dp, recurrent_dropout=dp)(x)

    att_x = utils.Attention(MAX_SEQUENCE_LENGTH)(x)
    mean_x = GlobalAveragePooling1D()(x)
    max_x = GlobalMaxPooling1D()(x)
    conc_x = concatenate([att_x, mean_x, max_x])
    conc_x = Dropout(dp)(conc_x)
    conc_x = Dense(hidDim, activation='relu')(conc_x)

    att_y = utils.Attention(MAX_SEQUENCE_LENGTH)(y)
    mean_y = GlobalAveragePooling1D()(y)
    max_y = GlobalMaxPooling1D()(y)
    conc_y = concatenate([att_y, mean_y, max_y])
    conc_y = Dropout(dp)(conc_y)
    conc_y = Dense(hidDim, activation='relu')(conc_y)

    conc = concatenate([conc_x, conc_y])
    conc = Dropout(dp)(conc)
    conc = Dense(hidDim, activation='relu')(conc)
    conc = Dropout(dp)(conc)
    output = Dense(NUM_CLASSES, activation='softmax')(conc)

    model = Model(inputs=inp, outputs=output)
    model.compile(loss='categorical_crossentropy',
                  optimizer=optimizers.Adam(1e-3),
                  metrics=[microF1Loss])

    return model

clr = CyclicLR(base_lr=0.001, max_lr=0.005,
               step_size=300., mode='exp_range',
               gamma=0.99994)
model = buildSingleModel(embedding_matrix, hidDim=32, dp=0.25)
```

```
step_dim is
35
input_shape is
(None, 35, 32)
step_dim is
35
input_shape is
(None, 35, 32)
```

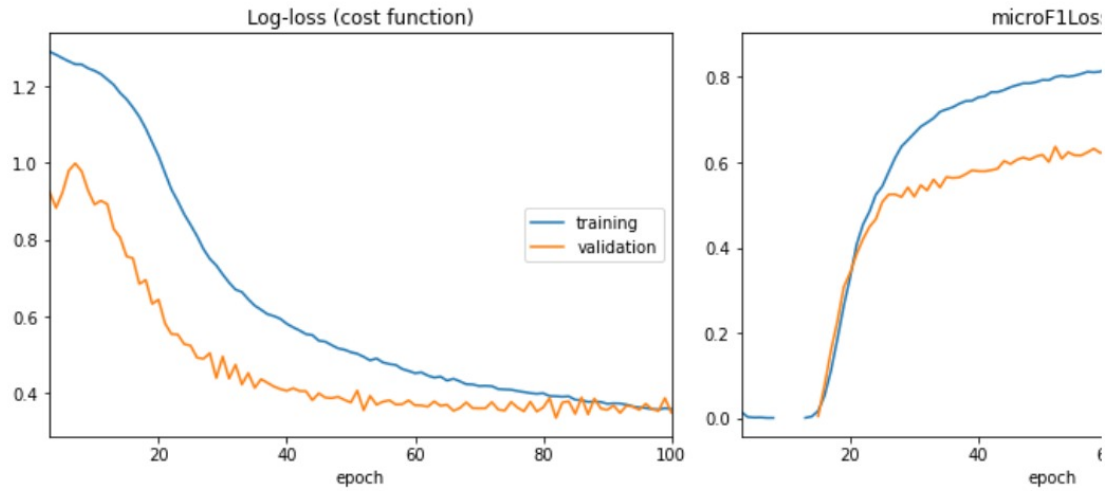
```
# from keras.utils import plot_model
# plot_model(model, to_file='/content/drive/My Drive/EmoContext/model.png', show_shape
```

```
# model.summary()
! pip install livelossplot
```

```
Collecting livelossplot
  Downloading https://files.pythonhosted.org/packages/7c/e4/a7884b57113dfe84d3565
Requirement already satisfied: ipython in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: matplotlib; python_version >= "3.6" in /usr/local/
Requirement already satisfied: pygments in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: prompt-toolkit<2.0.0,>=1.0.4 in /usr/local/lib/pyt
Requirement already satisfied: decorator in /usr/local/lib/python3.6/dist-package
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.6/dist-
Requirement already satisfied: pickleshare in /usr/local/lib/python3.6/dist-packa
Requirement already satisfied: simplegeneric>0.8 in /usr/local/lib/python3.6/dist
Requirement already satisfied: pexpect; sys_platform != "win32" in /usr/local/lib
Requirement already satisfied: traitlets>=4.2 in /usr/local/lib/python3.6/dist-pa
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.6/d
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/l
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.6/dist
Requirement already satisfied: numpy>=1.11 in /usr/local/lib/python3.6/dist-packa
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.6/dist-pack
Requirement already satisfied: six>=1.9.0 in /usr/local/lib/python3.6/dist-packag
Requirement already satisfied: wcwidth in /usr/local/lib/python3.6/dist-packages
Requirement already satisfied: ptyprocess>=0.5 in /usr/local/lib/python3.6/dist-p
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.6/dist-
Installing collected packages: livelossplot
Successfully installed livelossplot-0.5.0
```

```
import livelossplot
from livelossplot.keras import PlotLossesCallback

model.fit(train_all, labels,
          batch_size=8192, epochs=100,
          callbacks=[PlotLossesCallback(), clr],
          # callbacks=None,
          verbose=0,
          validation_data=(eval_all,  evalLabels))
```




```
Log-loss (cost function):
training   (min:    0.357, max:    1.374, cur:    0.359)
validation (min:    0.335, max:    1.199, cur:    0.349)

microF1Loss:
training   (min:    0.001, max:    0.859, cur:    0.858)
validation (min:    0.005, max:    0.662, cur:    0.657)
<keras.callbacks.History at 0x7f53ab94edd8>
```

```
import utils
predictions = model.predict(eval_all, batch_size=4096)
utils.getMetrics(predictions, evalLabels)
```

```
True Positives per class :  [2060.  107.  100.  134.]
False Positives per class :  [ 60. 102.  67. 125.]
False Negatives per class :  [278.  35.  25.  16.]
Class happy : Precision : 0.512, Recall : 0.754, F1 : 0.610
Class sad : Precision : 0.599, Recall : 0.800, F1 : 0.685
Class angry : Precision : 0.517, Recall : 0.893, F1 : 0.655
Ignoring the Others class, Macro Precision : 0.5427, Macro Recall : 0.8156, Macro
Ignoring the Others class, Micro TP : 341, FP : 294, FN : 76
Accuracy : 0.8715, Micro Precision : 0.5370, Micro Recall : 0.8177, Micro F1 : 0.
(0.8715063520871144, 0.53700787, 0.8177458, 0.6482889411382886)
```

```
# !pip install livelossplot
```

Pytorch Model

```
! pip install unidecode
```

```
Collecting unidecode
  Downloading https://files.pythonhosted.org/packages/d0/42/d9edfed04228bacea2d82
     |████████████████████████████████| 245kB 5.1MB/s
Installing collected packages: unidecode
Successfully installed unidecode-1.1.1
```

```
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from torch.autograd import Variable
from sklearn.metrics import f1_score
import os
import copy
import time


from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import f1_score
from torch.optim.optimizer import Optimizer
from unidecode import unidecode


# code inspired from: https://github.com/anandsaha/pytorch.cyclic.learning.rate/blob/r
class CyclicLR(object):
    def __init__(self, optimizer, base_lr=1e-3, max_lr=6e-3,
                 step_size=2000, mode='triangular', gamma=1.,
                 scale_fn=None, scale_mode='cycle', last_batch_iteration=-1):

        if not isinstance(optimizer, Optimizer):
            raise TypeError('{} is not an Optimizer'.format(
                type(optimizer).__name__))
        self.optimizer = optimizer

        if isinstance(base_lr, list) or isinstance(base_lr, tuple):
            if len(base_lr) != len(optimizer.param_groups):
                raise ValueError("expected {} base_lr, got {}".format(
                    len(optimizer.param_groups), len(base_lr)))
            self.base_lrs = list(base_lr)
        else:
            self.base_lrs = [base_lr] * len(optimizer.param_groups)

        if isinstance(max_lr, list) or isinstance(max_lr, tuple):
            if len(max_lr) != len(optimizer.param_groups):
                raise ValueError("expected {} max_lr, got {}".format(
                    len(optimizer.param_groups), len(max_lr)))
            self.max_lrs = list(max_lr)
        else:
            self.max_lrs = [max_lr] * len(optimizer.param_groups)
```

```
            self.step_size = step_size

            if mode not in ['triangular', 'triangular2', 'exp_range'] \
                    and scale_fn is None:
                raise ValueError('mode is invalid and scale_fn is None')

            self.mode = mode
            self.gamma = gamma

            if scale_fn is None:
                if self.mode == 'triangular':
                    self.scale_fn = self._triangular_scale_fn
                    self.scale_mode = 'cycle'
                elif self.mode == 'triangular2':
                    self.scale_fn = self._triangular2_scale_fn
                    self.scale_mode = 'cycle'
                elif self.mode == 'exp_range':
                    self.scale_fn = self._exp_range_scale_fn
                    self.scale_mode = 'iterations'
            else:
                self.scale_fn = scale_fn
                self.scale_mode = scale_mode

            self.batch_step(last_batch_iteration + 1)
            self.last_batch_iteration = last_batch_iteration

        def batch_step(self, batch_iteration=None):
            if batch_iteration is None:
                batch_iteration = self.last_batch_iteration + 1
            self.last_batch_iteration = batch_iteration
            for param_group, lr in zip(self.optimizer.param_groups, self.get_lr()):
                param_group['lr'] = lr

        def _triangular_scale_fn(self, x):
            return 1.

        def _triangular2_scale_fn(self, x):
            return 1 / (2. ** (x - 1))

        def _exp_range_scale_fn(self, x):
            return self.gamma**(x)

        def get_lr(self):
            step_size = float(self.step_size)
            cycle = np.floor(1 + self.last_batch_iteration / (2 * step_size))
            x = np.abs(self.last_batch_iteration / step_size - 2 * cycle + 1)

            lrs = []
            param_lrs = zip(self.optimizer.param_groups, self.base_lrs, self.max_lrs)
            for param_group, base_lr, max_lr in param_lrs:
                base_height = (max_lr - base_lr) * np.maximum(0, (1 - x))
                if self.scale_mode == 'cycle':
```

```
            if self.scale_mode == 'cycle':
                lr = base_lr + base_height * self.scale_fn(cycle)
            else:
                lr = base_lr + base_height * self.scale_fn(self.last_batch_iteration)
            lrs.append(lr)
        return lrs


# from torchnlp.nn import Attention
from torch.nn import GRU, LSTM, Dropout, Dropout2d, Linear, Embedding, Parameter, ReLU
import torch.nn.functional as F
import torch as torch


class Attention(nn.Module):
    def __init__(self, feature_dim, step_dim, bias=True, **kwargs):
        super(Attention, self).__init__(**kwargs)

        self.supports_masking = True

        self.bias = bias
        self.feature_dim = feature_dim
        self.step_dim = step_dim
        self.features_dim = 0

        weight = torch.zeros(feature_dim, 1)
        nn.init.kaiming_uniform_(weight)
        self.weight = nn.Parameter(weight)

        if bias:
            self.b = nn.Parameter(torch.zeros(step_dim))

    def forward(self, x, mask=None):
        feature_dim = self.feature_dim
        step_dim = self.step_dim

        eij = torch.mm(
            x.contiguous().view(-1, feature_dim),
            self.weight
        ).view(-1, step_dim)

        if self.bias:
            eij = eij + self.b

        eij = torch.tanh(eij)
        a = torch.exp(eij)

        if mask is not None:
            a = a * mask

        a = a / (torch.sum(a, 1, keepdim=True) + 1e-10)
```

```
            weighted_input = x * torch.unsqueeze(a, -1)
            return torch.sum(weighted_input, 1)



    class NelecModel(nn.Module):
        def __init__(self, embedding_matrix, hid_dim=32, maxlen=MAX_SEQUENCE_LENGTH, dp=0.
            super(NelecModel, self).__init__()

            num_embeddings, embedding_dim = embedding_matrix.shape
            self.embeddings = Embedding(num_embeddings, embedding_dim)
            self.embeddings.weight = Parameter(torch.tensor(embedding_matrix, dtype=torch.
            self.embeddings.weight.requires_grad = False

            self.hidden_dim = hid_dim
            self.gru    = GRU(hid_dim, hid_dim)
            self.embedding_dropout = nn.Dropout2d(dp)
            for name, param in self.gru.named_parameters():
                if 'bias' in name:
                     nn.init.constant_(param, 0.0)
                elif 'weight_ih' in name:
                     nn.init.kaiming_normal_(param)
                elif 'weight_hh' in name:
                     nn.init.orthogonal_(param)

            self.lstm   = LSTM(embedding_dim, hid_dim)

            for name, param in self.lstm.named_parameters():
                if 'bias' in name:
                     nn.init.constant_(param, 0.0)
                elif 'weight_ih' in name:
                     nn.init.kaiming_normal_(param)
                elif 'weight_hh' in name:
                     nn.init.orthogonal_(param)


            self.drop_out = Dropout(dp)
            self.attn = Attention(hid_dim, maxlen)
            self.hidden = Linear(hid_dim*3, hid_dim) # hidden layer for lstm and gru
            self.hidden2 = Linear(hid_dim*2, NUM_CLASSES) # hidden layer for first concate
            self.relu   = ReLU()

        def forward(self, inp):
            # (batch_size, max_seq_length, embdding_dim)
            inp_embed = self.embeddings(inp)

            # Spatial dropout2d implementation
            embeddings = inp_embed.unsqueeze(2)    # (N, T, 1, K)
            embeddings = embeddings.permute(0, 3, 2, 1)  # (N, K, 1, T)
            embeddings = self.embedding_dropout(embeddings)  # (N, K, 1, T), some features
            embeddings = embeddings.permute(0, 3, 2, 1)  # (N, T, 1, K)
```

```
        inp_embed = embeddings.squeeze(2)  # (N, T, K)

        in_lstm, (h_lstm, last_cell_state) = self.lstm(inp_embed)

        x_attn= self.attn(in_lstm)
        x_mean = torch.mean(in_lstm, 1)
        x_max, _ = torch.max(in_lstm, 1)
        x = torch.cat((x_attn, x_mean, x_max), dim=1)
        x = self.drop_out(x)
        x = self.relu(self.hidden(x))

        # gru layer:
        y_gru, y_hgru = self.gru(in_lstm)
        y_attn = self.attn(y_gru)
        y_mean = torch.mean(y_gru, 1)
        y_max,  _ = torch.max(y_gru, 1)
        y = torch.cat((y_attn, y_mean, y_max), dim=1)
        y = self.drop_out(y)
        y = self.relu(self.hidden(y))

        conc = torch.cat((x, y), dim=1)
        conc = self.drop_out(conc)
        conc = self.relu(self.hidden2(conc))
        conc = self.drop_out(conc)

        return conc



class MyDataset(Dataset):
    def __init__(self,dataset):
        self.dataset = dataset
    def __getitem__(self,index):
        data,target = self.dataset[index]
        return data,target,index
    def __len__(self):
        return len(self.dataset)


def categorical_cross_entropy(y_pred, y_true):
    y_pred = torch.clamp(y_pred, 1e-9, 1 - 1e-9)
    return -(y_true * torch.log(y_pred)).sum(dim=1).mean()

def pytorch_model_run_cv(x_train,y_train, x_test, y_test, model_obj, feats = False,cli
    # matrix for the out-of-fold predictions
    raw_train_preds = []
    raw_val_preds = []
    # matrix for the predictions on the test set
    test_preds = np.zeros((len(x_test), NUM_CLASSES))

    x_train = np.array(x_train)
    y_train = np.array(y_train)
```

```
x_train_tensor = torch.tensor(x_train, dtype=torch.long)
x_val_tensor   = torch.tensor(x_test, dtype=torch.long)
y_train_tensor = torch.tensor(y_train, dtype=torch.long)
y_val_tensor   = torch.tensor(y_test, dtype=torch.long)

model = copy.deepcopy(model_obj)

loss_fn = nn.CrossEntropyLoss()
softmax_fn = nn.Softmax(1)
step_size = 300
base_lr, max_lr = 0.001, 0.005
optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad, model.parameters())
                             lr=1e-3)
################################################################################
scheduler = CyclicLR(optimizer, base_lr=base_lr, max_lr=max_lr,
            step_size=step_size, mode='exp_range',
            gamma=0.99994)
################################################################################
train = MyDataset(torch.utils.data.TensorDataset(x_train_tensor, y_train_tensor))
valid = MyDataset(torch.utils.data.TensorDataset(x_val_tensor, y_val_tensor))

train_loader = torch.utils.data.DataLoader(train, batch_size=batch_size, shuffle=T
valid_loader = torch.utils.data.DataLoader(valid, batch_size=len(x_test), shuffle=

# print(f'Fold {i + 1}')
for epoch in range(n_epochs):
    start_time = time.time()
    model.train()

    avg_test_loss = 0.
    train_preds = []
    for i, (x_batch, y_batch, index) in enumerate(train_loader):
        y_pred = model(x_batch)
        if scheduler:
            scheduler.batch_step()
        loss = loss_fn(y_pred, y_batch)
        optimizer.zero_grad()
        loss.backward()
        if clip:
            nn.utils.clip_grad_norm_(model.parameters(),1)
        optimizer.step()
        avg_test_loss += loss.item() / len(train_loader)
        ## Add raw preds
        raw_preds = softmax_fn(y_pred).detach().numpy()
        raw_train_preds.append(raw_preds)
        ## Append the result
        preds_softmax = np.argmax(raw_preds, 1)
        train_preds.extend(preds_softmax)

    ## Use the metrics to evaluate the training process
    get metrics(y_train, train_preds)
```

```
            get_metrics(y_train, train_preds)

            model.eval()

            avg_val_loss = 0.
            val_preds = []
            for i, (x_batch, y_batch, index) in enumerate(valid_loader):
                y_pred = model(x_batch).detach()
                avg_val_loss += loss_fn(y_pred, y_batch).item() / len(valid_loader)

                ## calculate the raw preds
                # print (y_pred)
                raw_preds = softmax_fn(y_pred).numpy()
                # print (raw_preds)
                raw_val_preds.append(raw_preds)

                ## Append the result
                val_pred = np.argmax(raw_preds, 1)
                val_preds.extend(val_pred)

            get_metrics(y_test, val_preds)

            elapsed_time = time.time() - start_time
            print('Epoch {}/{} \t loss={:.4f} \t val_loss={:.4f} \t time={:.2f} \n\n'.form
                epoch + 1, n_epochs, avg_test_loss, avg_val_loss, elapsed_time))

        return raw_train_preds, raw_val_preds


    def get_metrics(ground, predictions):
        """Given predicted labels and the respective ground truth labels, display some met
        Input: shape [# of samples, NUM_CLASSES]
            predictions : Model output. Every row has 4 decimal values, with the highest b
            ground : Ground truth labels, converted to one-hot encodings. A sample belongi
        Output:
            accuracy : Average accuracy
            microPrecision : Precision calculated on a micro level. Ref -
            https://datascience.stackexchange.com/questions/15989/micro-average-vs-macro-a
            microRecall : Recall calculated on a micro level
            microF1 : Harmonic mean of microPrecision and microRecall. Higher value implie
        """
        # print ("the input is")
        # print (predictions)
        discretePredictions = to_categorical(predictions, NUM_CLASSES)
        # print (discretePredictions)
        ground = to_categorical(ground, NUM_CLASSES)
        truePositives = np.sum(discretePredictions * ground, axis=0)
        falsePositives = np.sum(np.clip(discretePredictions - ground, 0, 1), axis=0)
        falseNegatives = np.sum(np.clip(ground - discretePredictions, 0, 1), axis=0)

        print("True Positives per class : ", truePositives)
        print("False Positives per class : ", falsePositives)
```

```
    print("False Negatives per class : ", falseNegatives)

    #  Macro level calculation
    macroPrecision = 0
    macroRecall = 0
    f1_list = []
    # We ignore the "Others" class during the calculation of Precision, Recall and F1
    for c in range(1,4):
        precision = truePositives[c] / (truePositives[c] + falsePositives[c])
        macroPrecision += precision
        recall = truePositives[c] / (truePositives[c] + falseNegatives[c])
        macroRecall += recall
        f1 = (2 * recall * precision) / (precision + recall) if (precision + recall) >
        f1_list.append(f1)
        print("Class %s : Precision : %.3f, Recall : %.3f, F1 : %.3f" % (lab2emo[c], p

    print('Direct average of macro F1s are :------> ', (f1_list[0] + f1_list[1] + f1_l
    macroPrecision /= 3
    macroRecall /= 3
    macroF1 = (2 * macroRecall * macroPrecision) / (macroPrecision + macroRecall) \
        if (macroPrecision + macroRecall) > 0 else 0
    print("Ignoring the Others class, Macro Precision : %.4f, Macro Recall : %.4f, Mac
    macroPrecision, macroRecall, macroF1))

    # Micro level calculation
    truePositives = truePositives[1:].sum()
    falsePositives = falsePositives[1:].sum()
    falseNegatives = falseNegatives[1:].sum()

    print("Ignoring the Others class, Micro TP : %d, FP : %d, FN : %d"
          % (truePositives, falsePositives, falseNegatives))

    microPrecision = truePositives / (truePositives + falsePositives)
    microRecall = truePositives / (truePositives + falseNegatives)

    microF1 = (2 * microRecall * microPrecision) / (microPrecision + microRecall)\
        if (microPrecision + microRecall) > 0 else 0

    # predictions = predictions.argmax(axis=1)
    ground = ground.argmax(axis=1)
    accuracy = np.mean(predictions == ground)

    print("Accuracy : %.4f, Micro Precision : %.4f, Micro Recall : %.4f, Micro F1 : %.
    accuracy, microPrecision, microRecall, microF1))
    return accuracy, microPrecision, microRecall, microF1
```

Training function

```
n_epochs = 300
```

```
batch_size=8192

train_preds, test_preds = pytorch_model_run_cv(train_all,
                                               np.argmax(labels, axis=1),
                                               eval_all,
                                               np.argmax(evalLabels, axis=1),
                                               NelecModel(embedding_matrix=embedding_m
                                               feats = False)


# import pickle
# preds = {'train_preds':train_preds,  'test_preds': test_preds}

# with open("/content/drive/My Drive/EmoContext/preds3.pickle", 'wb') as f:
#   pickle.dump(preds, f)
```

```
True Positives per class :  [9582.  221.  374.  244.]
False Positives per class :  [9662. 1285. 1625. 1135.]
False Negatives per class :  [2387. 3211. 3969. 4140.]
Class happy : Precision : 0.147, Recall : 0.064, F1 : 0.090
Class sad : Precision : 0.187, Recall : 0.086, F1 : 0.118
Class angry : Precision : 0.177, Recall : 0.056, F1 : 0.085
Direct average of macro F1s are :------>  0.09737730444003562
Ignoring the Others class, Macro Precision : 0.1703, Macro Recall : 0.0687, Macro
Ignoring the Others class, Micro TP : 839, FP : 4045, FN : 11320
Accuracy : 0.4319, Micro Precision : 0.1718, Micro Recall : 0.0690, Micro F1 : 0.
True Positives per class :  [2338.    0.    0.    0.]
False Positives per class :  [417.   0.   0.   0.]
False Negatives per class :  [  0. 142. 125. 150.]
Class happy : Precision : nan, Recall : 0.000, F1 : 0.000
Class sad : Precision : nan, Recall : 0.000, F1 : 0.000
Class angry : Precision : nan, Recall : 0.000, F1 : 0.000
Direct average of macro F1s are :------>  0.0
Ignoring the Others class, Macro Precision : nan, Macro Recall : 0.0000, Macro F1
Ignoring the Others class, Micro TP : 0, FP : 0, FN : 417
Accuracy : 0.8486, Micro Precision : nan, Micro Recall : 0.0000, Micro F1 : 0.000
Epoch 1/300      loss=1.3865     val_loss=1.3814         time=25.37


/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:32: RuntimeWarning:
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:55: RuntimeWarning:
Streaming output truncated to the last 5000 lines.
Class angry : Precision : 0.540, Recall : 0.847, F1 : 0.660
Direct average of macro F1s are :------>  0.5846430718808877
Ignoring the Others class, Macro Precision : 0.4682, Macro Recall : 0.7794, Macro
Ignoring the Others class, Micro TP : 326, FP : 371, FN : 91
Accuracy : 0.8396, Micro Precision : 0.4677, Micro Recall : 0.7818, Micro F1 : 0.
Epoch 83/300     loss=0.7962     val_loss=0.5468         time=24.98


True Positives per class :  [6872.  443.  653.  653.]
False Positives per class :  [7171. 2574. 2792. 2970.]
False Negatives per class :  [5097. 2989. 3690. 3731.]
Class happy : Precision : 0.147, Recall : 0.129, F1 : 0.137
Class sad : Precision : 0.190, Recall : 0.150, F1 : 0.168
Class angry : Precision : 0.180, Recall : 0.149, F1 : 0.163
Direct average of macro F1s are :------>  0.1560622657523994
Ignoring the Others class, Macro Precision : 0.1722, Macro Recall : 0.1428, Macro
Ignoring the Others class, Micro TP : 1749, FP : 8336, FN : 10410
Accuracy : 0.3573, Micro Precision : 0.1734, Micro Recall : 0.1438, Micro F1 : 0.
True Positives per class :  [2014.   99.   95.  127.]
False Positives per class :  [ 78. 110. 120. 112.]
False Negatives per class :  [324.  43.  30.  23.]
Class happy : Precision : 0.474, Recall : 0.697, F1 : 0.564
Class sad : Precision : 0.442, Recall : 0.760, F1 : 0.559
Class angry : Precision : 0.531, Recall : 0.847, F1 : 0.653
Direct average of macro F1s are :------>  0.5919608040956567
Ignoring the Others class, Macro Precision : 0.4823, Macro Recall : 0.7679, Macro
Ignoring the Others class, Micro TP : 321, FP : 342, FN : 96
Accuracy : 0.8475, Micro Precision : 0.4842, Micro Recall : 0.7698, Micro F1 : 0.
Epoch 84/300     loss=0.7818     val_loss=0.5164         time=25.19
```

```
True Positives per class :  [7245.  418.  637.  652.]
False Positives per class :  [7222. 2320. 2807. 2827.]
False Negatives per class :  [4724. 3014. 3706. 3732.]
Class happy : Precision : 0.153, Recall : 0.122, F1 : 0.135
Class sad : Precision : 0.185, Recall : 0.147, F1 : 0.164
Class angry : Precision : 0.187, Recall : 0.149, F1 : 0.166
Direct average of macro F1s are :------>  0.15498011512875545
Ignoring the Others class, Macro Precision : 0.1750, Macro Recall : 0.1391, Macro
Ignoring the Others class, Micro TP : 1707, FP : 7954, FN : 10452
Accuracy : 0.3710, Micro Precision : 0.1767, Micro Recall : 0.1404, Micro F1 : 0.
True Positives per class :  [1954.  109.   94.  127.]
False Positives per class :  [ 67. 175. 119. 110.]
False Negatives per class :  [384.  33.  31.  23.]
Class happy : Precision : 0.384, Recall : 0.768, F1 : 0.512
Class sad : Precision : 0.441, Recall : 0.752, F1 : 0.556
Class angry : Precision : 0.536, Recall : 0.847, F1 : 0.656
Direct average of macro F1s are :------>  0.5747602873984556
Ignoring the Others class, Macro Precision : 0.4537, Macro Recall : 0.7888, Macro
Ignoring the Others class, Micro TP : 330, FP : 404, FN : 87
Accuracy : 0.8290, Micro Precision : 0.4496, Micro Recall : 0.7914, Micro F1 : 0.
Epoch 85/300     loss=0.7831     val_loss=0.5663         time=25.23


True Positives per class :  [7021.  404.  622.  618.]
False Positives per class :  [7244. 2516. 2843. 2860.]
False Negatives per class :  [4948. 3028. 3721. 3766.]
Class happy : Precision : 0.138, Recall : 0.118, F1 : 0.127
Class sad : Precision : 0.180, Recall : 0.143, F1 : 0.159
Class angry : Precision : 0.178, Recall : 0.141, F1 : 0.157
Direct average of macro F1s are :------>  0.14791324106769696
Ignoring the Others class, Macro Precision : 0.1652, Macro Recall : 0.1340, Macro
Ignoring the Others class, Micro TP : 1644, FP : 8219, FN : 10515
Accuracy : 0.3591, Micro Precision : 0.1667, Micro Recall : 0.1352, Micro F1 : 0.
True Positives per class :  [2009.  100.   93.  127.]
False Positives per class :  [ 78. 115. 112. 121.]
False Negatives per class :  [329.  42.  32.  23.]
Class happy : Precision : 0.465, Recall : 0.704, F1 : 0.560
Class sad : Precision : 0.454, Recall : 0.744, F1 : 0.564
Class angry : Precision : 0.512, Recall : 0.847, F1 : 0.638
Direct average of macro F1s are :------>  0.587350478385433
Ignoring the Others class, Macro Precision : 0.4770, Macro Recall : 0.7650, Macro
Ignoring the Others class, Micro TP : 320, FP : 348, FN : 97
Accuracy : 0.8454, Micro Precision : 0.4790, Micro Recall : 0.7674, Micro F1 : 0.
Epoch 86/300     loss=0.7788     val_loss=0.5134         time=24.65


True Positives per class :  [7006.  400.  639.  638.]
False Positives per class :  [7058. 2496. 2958. 2933.]
False Negatives per class :  [4963. 3032. 3704. 3746.]
Class happy : Precision : 0.138, Recall : 0.117, F1 : 0.126
Class sad : Precision : 0.178, Recall : 0.147, F1 : 0.161
Class angry : Precision : 0.179, Recall : 0.146, F1 : 0.160
Direct average of macro F1s are :------>  0.1492605658533853
Ignoring the Others class, Macro Precision : 0.1648, Macro Recall : 0.1364, Macro
Ignoring the Others class, Micro TP : 1677, FP : 8387, FN : 10482
Accuracy : 0.3599, Micro Precision : 0.1666, Micro Recall : 0.1379, Micro F1 : 0.
True Positives per class :  [1987.  107.   96.  127.]
False Positives per class :  [ 72. 141. 120. 105.]
```