Fork of https://www.kaggle.com/garydf/fork-from-bilstm-attention-kfold-0115

Added FastText embedding and combining it with Glove+Paragram.

Preface

Hello . This is basically cutting and pasting from the amazing kernels of this competition. Please notify me if I don't attribute something correctly.

- https://www.kaggle.com/gmhost/gru-capsule
- How to: Preprocessing when using embeddings https://www.kaggle.com/christofhenkel/how-to-preprocessing-when-using-embeddings
- Improve your Score with some Text Preprocessing https://www.kaggle.com/theoviel/improve-your-score-with-some-text-preprocessing
- Simple attention layer taken from https://github.com/mttk/rnn-classifier/blob/master/model.py
- https://www.kaggle.com/ziliwang/baseline-pytorch-bilstm
- https://www.kaggle.com/hengzheng/pytorch-starter

UPDATE: I seems that the shuffling the data doesn't add the features in the correct order. To address this issue I added a custom dataset class that can return indexes so that they can be accessed while training and properly put each feature with the corresponding sample. The training time though is increased, so you might need to make the model lighter in order to submit results.

IMPORTS

In [ ]:
```
import time
import random
import pandas as pd
import numpy as np
import gc
import re
import torch
from torchtext import data
import spacy
from tqdm import tqdm_notebook, tnrange
from tqdm.auto import tqdm

tqdm.pandas(desc='Progress')
from collections import Counter
from textblob import TextBlob
from nltk import word_tokenize

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.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence
from torch.autograd import Variable
from torchtext.data import Example
from sklearn.metrics import f1_score
import torchtext
import os

from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences

# cross validation and metrics
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import f1_score
from torch.optim.optimizer import Optimizer
from unidecode import unidecode
```

Basic Parameters

In [ ]:
```
embed_size = 300 # how big is each word vector
max_features = 120000 # how many unique words to use (i.e num rows in embedding vector)
maxlen = 70 # max number of words in a question to use
batch_size = 512 # how many samples to process at once
n_epochs = 5 # how many times to iterate over all samples
n_splits = 5 # Number of K-fold Splits

SEED = 1029
```

Ensure determinism in the results

A common headache in this competition is the lack of determinism in the results due to cudnn. The following Kernel has a solution in Pytorch.

See https://www.kaggle.com/hengzheng/pytorch-starter.

In [ ]:
```
def seed_everything(seed=1029):
    random.seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.backends.cudnn.deterministic = True
seed_everything()
```

Code for Loading Embeddings

Functions taken from the kernel:https://www.kaggle.com/gmhost/gru-capsule

In [ ]:
```
## FUNCTIONS TAKEN FROM https://www.kaggle.com/gmhost/gru-capsule

def load_glove(word_index):
    EMBEDDING_FILE = '../input/embeddings/glove.840B.300d/glove.840B.300d.txt'
    def get_coefs(word,*arr): return word, np.asarray(arr, dtype='float32')[:300]
    embeddings_index = dict(get_coefs(*o.split(" ")) for o in open(EMBEDDING_FILE))

    all_embs = np.stack(embeddings_index.values())
    emb_mean,emb_std = -0.005838499,0.48782197
    embed_size = all_embs.shape[1]

    # word_index = tokenizer.word_index
    nb_words = min(max_features, len(word_index))
    # Why random embedding for OOV? what if use mean?
    embedding_matrix = np.random.normal(emb_mean, emb_std, (nb_words, embed_size))
    #embedding_matrix = np.random.normal(emb_mean, 0, (nb_words, embed_size)) # std 0
    for word, i in word_index.items():
        if i >= max_features: continue
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None: embedding_matrix[i] = embedding_vector

    return embedding_matrix

def load_fasttext(word_index):
    EMBEDDING_FILE = '../input/embeddings/wiki-news-300d-1M/wiki-news-300d-1M.vec'
    def get_coefs(word,*arr): return word, np.asarray(arr, dtype='float32')
    embeddings_index = dict(get_coefs(*o.split(" ")) for o in open(EMBEDDING_FILE) if len(o)>100)

    all_embs = np.stack(embeddings_index.values())
    emb_mean,emb_std = all_embs.mean(), all_embs.std()
    embed_size = all_embs.shape[1]

    # word_index = tokenizer.word_index
    nb_words = min(max_features, len(word_index))
    embedding_matrix = np.random.normal(emb_mean, emb_std, (nb_words, embed_size))
    #embedding_matrix = np.random.normal(emb_mean, 0, (nb_words, embed_size))
    for word, i in word_index.items():
        if i >= max_features: continue
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None: embedding_matrix[i] = embedding_vector

    return embedding_matrix

def load_para(word_index):
    EMBEDDING_FILE = '../input/embeddings/paragram_300_sl999/paragram_300_sl999.txt'
    def get_coefs(word,*arr): return word, np.asarray(arr, dtype='float32')
    embeddings_index = dict(get_coefs(*o.split(" ")) for o in open(EMBEDDING_FILE, encoding="utf8", errors='ignore') if len(o)>100)

    all_embs = np.stack(embeddings_index.values())
    emb_mean,emb_std = -0.0053247833,0.49346462
    embed_size = all_embs.shape[1]

    # word_index = tokenizer.word_index
    nb_words = min(max_features, len(word_index))
    embedding_matrix = np.random.normal(emb_mean, emb_std, (nb_words, embed_size))
    #embedding_matrix = np.random.normal(emb_mean, 0, (nb_words, embed_size))
    for word, i in word_index.items():
        if i >= max_features: continue
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None: embedding_matrix[i] = embedding_vector

    return embedding_matrix
```

LOAD PROCESSED TRAINING DATA FROM DISK

In [ ]:
```
df_train = pd.read_csv("../input/train.csv")
df_test = pd.read_csv("../input/test.csv")
df = pd.concat([df_train ,df_test],sort=True)
```

In [ ]:
```
def build_vocab(texts):
    sentences = texts.apply(lambda x: x.split()).values
    vocab = {}
    for sentence in sentences:
        for word in sentence:
            try:
                vocab[word] += 1
            except KeyError:
                vocab[word] = 1
    return vocab
vocab = build_vocab(df['question_text'])
```

In [ ]:
```
sin = len(df_train[df_train["target"]==0])
insin = len(df_train[df_train["target"]==1])
persin = (sin/(sin+insin))*100
perinsin = (insin/(sin+insin))*100
print("# Sincere questions: {:,}({:.2f}%) and # Insincere questions: {:,}({:.2f}%)".format(sin,persin,insin,perinsin))
# print("Sinsere:{}% Insincere: {}%".format(round(persin,2),round(perinsin,2)))
print("# Test samples: {:,}({:.3f}% of train samples)".format(len(df_test),len(df_test)/len(df_train)))
```

Normalization

Borrowed from:

- How to: Preprocessing when using embeddings https://www.kaggle.com/christofhenkel/how-to-preprocessing-when-using-embeddings
- Improve your Score with some Text Preprocessing https://www.kaggle.com/theoviel/improve-your-score-with-some-text-preprocessing

In [ ]:
```
def build_vocab(texts):
    sentences = texts.apply(lambda x: x.split()).values
    vocab = {}
    for sentence in sentences:
        for word in sentence:
            try:
                vocab[word] += 1
            except KeyError:
                vocab[word] = 1
    return vocab

def known_contractions(embed):
    known = []
    for contract in contraction_mapping:
        if contract in embed:
            known.append(contract)
    return known
def clean_contractions(text, mapping):
    specials = ["’", "‘", "´", "`"]
    for s in specials:
        text = text.replace(s, "'")
    text = ' '.join([mapping[t] if t in mapping else t for t in text.split(" ")])
    return text
def correct_spelling(x, dic):
    for word in dic.keys():
        x = x.replace(word, dic[word])
    return x
def unknown_punct(embed, punct):
    unknown = ''
    for p in punct:
        if p not in embed:
            unknown += p
            unknown += ' '
    return unknown

def clean_numbers(x):
    x = re.sub('[0-9]{5,}', '#####', x)
    x = re.sub('[0-9]{4}', '####', x)
    x = re.sub('[0-9]{3}', '###', x)
    x = re.sub('[0-9]{2}', '##', x)
    return x

def clean_special_chars(text, punct, mapping):
    for p in mapping:
        text = text.replace(p, mapping[p])

    for p in punct:
        text = text.replace(p, f' {p} ')

    specials = {'\u200b': ' ', '…': ' ... ', '\ufeff': '', 'करना': '', 'है': ''}  # Other special characters that I have to deal with in last
    for s in specials:
        text = text.replace(s, specials[s])

    return text
def add_lower(embedding, vocab):
    count = 0
    for word in vocab:
        if word in embedding and word.lower() not in embedding:
            embedding[word.lower()] = embedding[word]
            count += 1
    print(f"Added {count} words to embedding")
```

In [ ]:
```
puncts = [',', '.', '"', ':', ')', '(', '-', '!', '?', '|', ';', "'", '$', '&', '/', '[', ']', '>', '%', '=', '#', '*', '+', '\\', '•', '~', '@', '£',
 '·', '_', '{', '}', '©', '^', '®', '`', '<', '→', '°', '€', '™', '›', '♥', '←', '×', '§', '″', '′', 'Â', '█', '½', 'à', '…',
 '“', '★', '”', '–', '●', 'â', '►', '−', '¢', '²', '¬', '░', '¶', '↑', '±', '¿', '▾', '═', '¦', '║', '―', '¥', '▓', '—', '‹', '─',
 '▒', '：', '¼', '⊕', '▼', '▪', '†', '■', '’', '▀', '¨', '▄', '♫', '☆', 'é', '¯', '♦', '¤', '▲', 'è', '¸', '¾', 'Ã', '⋅', '‘', '∞',
 '∙', '）', '↓', '、', '│', '（', '»', '，', '♪', '╩', '╚', '³', '・', '╦', '╣', '╔', '╗', '▬', '❤', 'ï', 'Ø', '¹', '≤', '‡', '√', ]

def clean_text(x):
    x = str(x)
    for punct in puncts:
        x = x.replace(punct, f' {punct} ')
    return x

def clean_numbers(x):
    x = re.sub('[0-9]{5,}', '#####', x)
    x = re.sub('[0-9]{4}', '####', x)
    x = re.sub('[0-9]{3}', '###', x)
    x = re.sub('[0-9]{2}', '##', x)
    return x

mispell_dict = {"ain't": "is not", "aren't": "are not","can't": "cannot", "'cause": "because", "could've": "could have", "couldn't": "could not", "didn't": "did not",  "doesn't": "does not", "don't": "do not", "hadn't": "had not", "hasn't": "has not", "haven't": "have not", "he'd": "he would","he'll": "he will", "he's": "he is", "how'd": "how did", "how'd'y": "how do you", "how'll": "how will", "how's": "how is",  "I'd": "I would", "I'd've": "I would have", "I'll": "I will", "I'll've": "I will have","I'm": "I am", "I've": "I have", "i'd": "i would", "i'd've": "i would have", "i'll": "i will",  "i'll've": "i will have","i'm": "i am", "i've": "i have", "isn't": "is not", "it'd": "it would", "it'd've": "it would have", "it'll": "it will", "it'll've": "it will have","it's": "it is", "let's": "let us", "ma'am": "madam", "mayn't": "may not", "might've": "might have","mightn't": "might not","mightn't've": "might not have", "must've": "must have", "mustn't": "must not", "mustn't've": "must not have", "needn't": "need not", "needn't've": "need not have","o'clock": "of the clock", "oughtn't": "ought not", "oughtn't've": "ought not have", "shan't": "shall not", "sha'n't": "shall not", "shan't've": "shall not have", "she'd": "she would", "she'd've": "she would have", "she'll": "she will", "she'll've": "she will have", "she's": "she is", "should've": "should have", "shouldn't": "should not", "shouldn't've": "should not have", "so've": "so have","so's": "so as", "this's": "this is","that'd": "that would", "that'd've": "that would have", "that's": "that is", "there'd": "there would", "there'd've": "there would have", "there's": "there is", "here's": "here is","they'd": "they would", "they'd've": "they would have", "they'll": "they will", "they'll've": "they will have", "they're": "they are", "they've": "they have", "to've": "to have", "wasn't": "was not", "we'd": "we would", "we'd've": "we would have", "we'll": "we will", "we'll've": "we will have", "we're": "we are", "we've": "we have", "weren't": "were not", "what'll": "what will", "what'll've": "what will have", "what're": "what are",  "what's": "what is", "what've": "what have", "when's": "when is", "when've": "when have", "where'd": "where did", "where's": "where is", "where've": "where have", "who'll": "who will", "who'll've": "who will have", "who's": "who is", "who've": "who have", "why's": "why is", "why've": "why have", "will've": "will have", "won't": "will not", "won't've": "will not have", "would've": "would have", "wouldn't": "would not", "wouldn't've": "would not have", "y'all": "you all", "y'all'd": "you all would","y'all'd've": "you all would have","y'all're": "you all are","y'all've": "you all have","you'd": "you would", "you'd've": "you would have", "you'll": "you will", "you'll've": "you will have", "you're": "you are", "you've": "you have", 'colour': 'color', 'centre': 'center', 'favourite': 'favorite', 'travelling': 'traveling', 'counselling': 'counseling', 'theatre': 'theater', 'cancelled': 'canceled', 'labour': 'labor', 'organisation': 'organization', 'wwii': 'world war 2', 'citicise': 'criticize', 'youtu ': 'youtube ', 'Qoura': 'Quora', 'sallary': 'salary', 'Whta': 'What', 'narcisist': 'narcissist', 'howdo': 'how do', 'whatare': 'what are', 'howcan': 'how can', 'howmuch': 'how much', 'howmany': 'how many', 'whydo': 'why do', 'doI': 'do I', 'theBest': 'the best', 'howdoes': 'how does', 'mastrubation': 'masturbation', 'mastrubate': 'masturbate', "mastrubating": 'masturbating', 'pennis': 'penis', 'Etherium': 'Ethereum', 'narcissit': 'narcissist', 'bigdata': 'big data', '2k17': '2017', '2k18': '2018', 'qouta': 'quota', 'exboyfriend': 'ex boyfriend', 'airhostess': 'air hostess', "whst": 'what', 'watsapp': 'whatsapp', 'demonitisation': 'demonetization', 'demonitization': 'demonetization', 'demonetisation': 'demonetization'}

def _get_mispell(mispell_dict):
    mispell_re = re.compile('(%s)' % '|'.join(mispell_dict.keys()))
    return mispell_dict, mispell_re

mispellings, mispellings_re = _get_mispell(mispell_dict)
def replace_typical_misspell(text):
    def replace(match):
        return mispellings[match.group(0)]
    return mispellings_re.sub(replace, text)
```

Extra feature part taken from https://github.com/wongchunghang/toxic-comment-challenge-lstm/blob/master/toxic_comment_9872_model.ipynb

In [ ]:
```
from sklearn.preprocessing import StandardScaler


def add_features(df):

    df['question_text'] = df['question_text'].progress_apply(lambda x:str(x))
    df['total_length'] = df['question_text'].progress_apply(len)
    df['capitals'] = df['question_text'].progress_apply(lambda comment: sum(1 for c in comment if c.isupper()))
    df['caps_vs_length'] = df.progress_apply(lambda row: float(row['capitals'])/float(row['total_length']),
                                axis=1)
    df['num_words'] = df.question_text.str.count('\S+')
    df['num_unique_words'] = df['question_text'].progress_apply(lambda comment: len(set(w for w in comment.split())))
    df['words_vs_unique'] = df['num_unique_words'] / df['num_words']

    return df

def load_and_prec():
    train_df = pd.read_csv("../input/train.csv")
    test_df = pd.read_csv("../input/test.csv")
    print("Train shape : ",train_df.shape)
    print("Test shape : ",test_df.shape)

    # lower
    train_df["question_text"] = train_df["question_text"].apply(lambda x: x.lower())
    test_df["question_text"] = test_df["question_text"].apply(lambda x: x.lower())

    # Clean the text
    train_df["question_text"] = train_df["question_text"].progress_apply(lambda x: clean_text(x))
    test_df["question_text"] = test_df["question_text"].apply(lambda x: clean_text(x))

    # Clean numbers
    train_df["question_text"] = train_df["question_text"].progress_apply(lambda x: clean_numbers(x))
    test_df["question_text"] = test_df["question_text"].apply(lambda x: clean_numbers(x))

    # Clean speelings
    train_df["question_text"] = train_df["question_text"].progress_apply(lambda x: replace_typical_misspell(x))
    test_df["question_text"] = test_df["question_text"].apply(lambda x: replace_typical_misspell(x))

    ## fill up the missing values
    train_X = train_df["question_text"].fillna("_##_").values
    test_X = test_df["question_text"].fillna("_##_").values



    ##################### Add Features ##############################
    #  https://github.com/wongchunghang/toxic-comment-challenge-lstm/blob/master/toxic_comment_9872_model.ipynb
    train = add_features(train_df)
    test = add_features(test_df)

    features = train[['caps_vs_length', 'words_vs_unique']].fillna(0)
    test_features = test[['caps_vs_length', 'words_vs_unique']].fillna(0)

    ss = StandardScaler()
    ss.fit(np.vstack((features, test_features)))
    features = ss.transform(features)
    test_features = ss.transform(test_features)
    ###########################################################################

    ## Tokenize the sentences
    tokenizer = Tokenizer(num_words=max_features)
    tokenizer.fit_on_texts(list(train_X))
    train_X = tokenizer.texts_to_sequences(train_X)
    test_X = tokenizer.texts_to_sequences(test_X)

    ## Pad the sentences
    train_X = pad_sequences(train_X, maxlen=maxlen)
    test_X = pad_sequences(test_X, maxlen=maxlen)

    ## Get the target values
    train_y = train_df['target'].values

#     # Splitting to training and a final test set
#     train_X, x_test_f, train_y, y_test_f = train_test_split(list(zip(train_X,features)), train_y, test_size=0.2, random_state=SEED)
#     train_X, features = zip(*train_X)
#     x_test_f, features_t = zip(*x_test_f)

    #shuffling the data
    np.random.seed(SEED)
    trn_idx = np.random.permutation(len(train_X))

    train_X = train_X[trn_idx]
    train_y = train_y[trn_idx]
    features = features[trn_idx]

    return train_X, test_X, train_y, features, test_features, tokenizer.word_index
#     return train_X, test_X, train_y, x_test_f,y_test_f,features, test_features, features_t, tokenizer.word_index
#     return train_X, test_X, train_y, tokenizer.word_index
```

In [ ]:
```
# fill up the missing values
# x_train, x_test, y_train, word_index = load_and_prec()
x_train, x_test, y_train, features, test_features, word_index = load_and_prec()
# x_train, x_test, y_train, x_test_f,y_test_f,features, test_features,features_t, word_index = load_and_prec()
```

SAVE DATASET TO DISK

In [ ]:
```
np.save("x_train",x_train)
np.save("x_test",x_test)
np.save("y_train",y_train)

np.save("features",features)
np.save("test_features",test_features)
np.save("word_index.npy",word_index)
```

LOAD DATASET FROM DISK

In [ ]:
```
x_train = np.load("x_train.npy")
x_test = np.load("x_test.npy")
y_train = np.load("y_train.npy")
features = np.load("features.npy")
test_features = np.load("test_features.npy")
word_index = np.load("word_index.npy").item()
```

In [ ]:
```
features.shape
```

Load Embeddings

Two embedding matrices have been used. Glove, and paragram. The mean of the two is used as the final embedding matrix

In [ ]:
```
# missing entries in the embedding are set using np.random.normal so we have to seed here too
seed_everything()

glove_embeddings = load_glove(word_index)
paragram_embeddings = load_para(word_index)
fasttext_embeddings = load_fasttext(word_index)

embedding_matrix = np.mean([glove_embeddings, paragram_embeddings, fasttext_embeddings], axis=0)

# vocab = build_vocab(df['question_text'])
# add_lower(embedding_matrix, vocab)
del glove_embeddings, paragram_embeddings, fasttext_embeddings
gc.collect()

np.shape(embedding_matrix)
```

In [ ]:
```
np.shape(embedding_matrix)
```

Use Stratified K Fold to improve results

In [ ]:
```
splits = list(StratifiedKFold(n_splits=n_splits, shuffle=True, random_state=SEED).split(x_train, y_train))
splits[:3]
```

Cyclic CLR

Code taken from https://www.kaggle.com/dannykliu/lstm-with-attention-clr-in-pytorch

In [ ]:
```
# code inspired from: https://github.com/anandsaha/pytorch.cyclic.learning.rate/blob/master/cls.py
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':
                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
```

Model Architecture

Binary LSTM with an attention layer and an additional fully connected layer. Also added extra features taken from a winning kernel of the toxic comments competition. Also using CLR and a capsule Layer. Blended together in concatentation.

Initial idea borrowed from: https://www.kaggle.com/ziliwang/baseline-pytorch-bilstm

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

embedding_dim = 300
embedding_path = '../save/embedding_matrix.npy'  # or False, not use pre-trained-matrix
use_pretrained_embedding = True

hidden_size = 60
gru_len = hidden_size

Routings = 4 #5
Num_capsule = 5
Dim_capsule = 5#16
dropout_p = 0.25
rate_drop_dense = 0.28
LR = 0.001
T_epsilon = 1e-7
num_classes = 30


class Embed_Layer(nn.Module):
    def __init__(self, embedding_matrix=None, vocab_size=None, embedding_dim=300):
        super(Embed_Layer, self).__init__()
        self.encoder = nn.Embedding(vocab_size + 1, embedding_dim)
        if use_pretrained_embedding:
            # self.encoder.weight.data.copy_(t.from_numpy(np.load(embedding_path))) # 方法一，加载np.save的npy文件
            self.encoder.weight.data.copy_(t.from_numpy(embedding_matrix))  # 方法二

    def forward(self, x, dropout_p=0.25):
        return nn.Dropout(p=dropout_p)(self.encoder(x))


class GRU_Layer(nn.Module):
    def __init__(self):
        super(GRU_Layer, self).__init__()
        self.gru = nn.GRU(input_size=300,
                          hidden_size=gru_len,
                          bidirectional=True)
        '''
        自己修改GRU里面的激活函数及加dropout和recurrent_dropout
        如果要使用，把rnn_revised import进来，但好像是使用cpu跑的，比较慢
       '''
        # # if you uncomment /*from rnn_revised import * */, uncomment following code aswell
        # self.gru = RNNHardSigmoid('GRU', input_size=300,
        #                           hidden_size=gru_len,
        #                           bidirectional=True)

    # 这步很关键，需要像keras一样用glorot_uniform和orthogonal_uniform初始化参数
    def init_weights(self):
        ih = (param.data for name, param in self.named_parameters() if 'weight_ih' in name)
        hh = (param.data for name, param in self.named_parameters() if 'weight_hh' in name)
        b = (param.data for name, param in self.named_parameters() if 'bias' in name)
        for k in ih:
            nn.init.xavier_uniform_(k)
        for k in hh:
            nn.init.orthogonal_(k)
        for k in b:
            nn.init.constant_(k, 0)

    def forward(self, x):
        return self.gru(x)


# core caps_layer with squash func
class Caps_Layer(nn.Module):
    def __init__(self, input_dim_capsule=gru_len * 2, num_capsule=Num_capsule, dim_capsule=Dim_capsule, \
                 routings=Routings, kernel_size=(9, 1), share_weights=True,
                 activation='default', **kwargs):
        super(Caps_Layer, self).__init__(**kwargs)

        self.num_capsule = num_capsule
        self.dim_capsule = dim_capsule
        self.routings = routings
        self.kernel_size = kernel_size  # 暂时没用到
        self.share_weights = share_weights
        if activation == 'default':
            self.activation = self.squash
        else:
            self.activation = nn.ReLU(inplace=True)

        if self.share_weights:
            self.W = nn.Parameter(
                nn.init.xavier_normal_(t.empty(1, input_dim_capsule, self.num_capsule * self.dim_capsule)))
        else:
            self.W = nn.Parameter(
                t.randn(BATCH_SIZE, input_dim_capsule, self.num_capsule * self.dim_capsule))  # 64即batch_size

    def forward(self, x):

        if self.share_weights:
            u_hat_vecs = t.matmul(x, self.W)
        else:
            print('add later')

        batch_size = x.size(0)
        input_num_capsule = x.size(1)
        u_hat_vecs = u_hat_vecs.view((batch_size, input_num_capsule,
                                      self.num_capsule, self.dim_capsule))
        u_hat_vecs = u_hat_vecs.permute(0, 2, 1, 3)  # 转成(batch_size,num_capsule,input_num_capsule,dim_capsule)
        b = t.zeros_like(u_hat_vecs[:, :, :, 0])  # (batch_size,num_capsule,input_num_capsule)

        for i in range(self.routings):
            b = b.permute(0, 2, 1)
            c = F.softmax(b, dim=2)
            c = c.permute(0, 2, 1)
            b = b.permute(0, 2, 1)
            outputs = self.activation(t.einsum('bij,bijk->bik', (c, u_hat_vecs)))  # batch matrix multiplication
            # outputs shape (batch_size, num_capsule, dim_capsule)
            if i < self.routings - 1:
                b = t.einsum('bik,bijk->bij', (outputs, u_hat_vecs))  # batch matrix multiplication
        return outputs  # (batch_size, num_capsule, dim_capsule)

    # text version of squash, slight different from original one
    def squash(self, x, axis=-1):
        s_squared_norm = (x ** 2).sum(axis, keepdim=True)
        scale = t.sqrt(s_squared_norm + T_epsilon)
        return x / scale

class Capsule_Main(nn.Module):
    def __init__(self, embedding_matrix=None, vocab_size=None):
        super(Capsule_Main, self).__init__()
        self.embed_layer = Embed_Layer(embedding_matrix, vocab_size)
        self.gru_layer = GRU_Layer()
        # 【重要】初始化GRU权重操作，这一步非常关键，acc上升到0.98，如果用默认的uniform初始化则acc一直在0.5左右
        self.gru_layer.init_weights()
        self.caps_layer = Caps_Layer()
        self.dense_layer = Dense_Layer()

    def forward(self, content):
        content1 = self.embed_layer(content)
        content2, _ = self.gru_layer(
            content1)  # 这个输出是个tuple，一个output(seq_len, batch_size, num_directions * hidden_size)，一个hn
        content3 = self.caps_layer(content2)
        output = self.dense_layer(content3)
        return output
```

In [ ]:

```python
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.xavier_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 NeuralNet(nn.Module):
    def __init__(self):
        super(NeuralNet, self).__init__()

        fc_layer = 16
        fc_layer1 = 16

        self.embedding = nn.Embedding(max_features, embed_size)
        self.embedding.weight = nn.Parameter(torch.tensor(embedding_matrix, dtype=torch.float32))
        self.embedding.weight.requires_grad = False

        self.embedding_dropout = nn.Dropout2d(0.1)
        self.lstm = nn.LSTM(embed_size, hidden_size, bidirectional=True, batch_first=True)
        self.gru = nn.GRU(hidden_size * 2, hidden_size, bidirectional=True, batch_first=True)

        self.lstm2 = nn.LSTM(hidden_size * 2, hidden_size, bidirectional=True, batch_first=True)

        self.lstm_attention = Attention(hidden_size * 2, maxlen)
        self.gru_attention = Attention(hidden_size * 2, maxlen)
        self.bn = nn.BatchNorm1d(16, momentum=0.5)
        self.linear = nn.Linear(hidden_size*8+3, fc_layer1) #643:80 - 483:60 - 323:40
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.1)
        self.fc = nn.Linear(fc_layer**2,fc_layer)
        self.out = nn.Linear(fc_layer, 1)
        self.lincaps = nn.Linear(Num_capsule * Dim_capsule, 1)
        self.caps_layer = Caps_Layer()

    def forward(self, x):

#         Capsule(num_capsule=10, dim_capsule=10, routings=4, share_weights=True)(x)

        h_embedding = self.embedding(x[0])
        h_embedding = torch.squeeze(
            self.embedding_dropout(torch.unsqueeze(h_embedding, 0)))

        h_lstm, _ = self.lstm(h_embedding)
        h_gru, _ = self.gru(h_lstm)

        ##Capsule Layer
        content3 = self.caps_layer(h_gru)
        content3 = self.dropout(content3)
        batch_size = content3.size(0)
        content3 = content3.view(batch_size, -1)
        content3 = self.relu(self.lincaps(content3))

        ##Attention Layer
        h_lstm_atten = self.lstm_attention(h_lstm)
        h_gru_atten = self.gru_attention(h_gru)

        # global average pooling
        avg_pool = torch.mean(h_gru, 1)
        # global max pooling
        max_pool, _ = torch.max(h_gru, 1)

        f = torch.tensor(x[1], dtype=torch.float).cuda()

                #[512,160]
        conc = torch.cat((h_lstm_atten, h_gru_atten,content3, avg_pool, max_pool,f), 1)
        conc = self.relu(self.linear(conc))
        conc = self.bn(conc)
        conc = self.dropout(conc)

        out = self.out(conc)

        return out
```

Training

The method for training is borrowed from https://www.kaggle.com/hengzheng/pytorch-starter

In [ ]:

```python
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)
```

In [ ]:

```python
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# matrix for the out-of-fold predictions
train_preds = np.zeros((len(x_train)))
# matrix for the predictions on the test set
test_preds = np.zeros((len(df_test)))

# always call this before training for deterministic results
seed_everything()

# x_test_cuda_f = torch.tensor(x_test_f, dtype=torch.long).cuda()
# test_f = torch.utils.data.TensorDataset(x_test_cuda_f)
# test_loader_f = torch.utils.data.DataLoader(test_f, batch_size=batch_size, shuffle=False)


x_test_cuda = torch.tensor(x_test, dtype=torch.long).cuda()
test = torch.utils.data.TensorDataset(x_test_cuda)
test_loader = torch.utils.data.DataLoader(test, batch_size=batch_size, shuffle=False)

avg_losses_f = []
avg_val_losses_f = []
```

In [ ]:

```python
for i, (train_idx, valid_idx) in enumerate(splits):
    # split data in train / validation according to the KFold indeces
    # also, convert them to a torch tensor and store them on the GPU (done with .cuda())
    x_train = np.array(x_train)
    y_train = np.array(y_train)
    features = np.array(features)

    x_train_fold = torch.tensor(x_train[train_idx.astype(int)], dtype=torch.long).cuda()
    y_train_fold = torch.tensor(y_train[train_idx.astype(int), np.newaxis], dtype=torch.float32).cuda()

    kfold_X_features = features[train_idx.astype(int)]
    kfold_X_valid_features = features[valid_idx.astype(int)]
    x_val_fold = torch.tensor(x_train[valid_idx.astype(int)], dtype=torch.long).cuda()
    y_val_fold = torch.tensor(y_train[valid_idx.astype(int), np.newaxis], dtype=torch.float32).cuda()

#     model = BiLSTM(lstm_layer=2,hidden_dim=40,dropout=DROPOUT).cuda()
    model = NeuralNet()

    # make sure everything in the model is running on the GPU
    model.cuda()

    # define binary cross entropy loss
    # note that the model returns logit to take advantage of the log-sum-exp trick
    # for numerical stability in the loss
    loss_fn = torch.nn.BCEWithLogitsLoss(reduction='sum')

    step_size = 300
    base_lr, max_lr = 0.001, 0.003
    optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad, model.parameters()),
                             lr=max_lr)

    ################################################################################################
    scheduler = CyclicLR(optimizer, base_lr=base_lr, max_lr=max_lr,
               step_size=step_size, mode='exp_range',
               gamma=0.99994)
    ###############################################################################################

    train = torch.utils.data.TensorDataset(x_train_fold, y_train_fold)
    valid = torch.utils.data.TensorDataset(x_val_fold, y_val_fold)

    train = MyDataset(train)
    valid = MyDataset(valid)

    ##No need to shuffle the data again here. Shuffling happens when splitting for kfolds.
    train_loader = torch.utils.data.DataLoader(train, batch_size=batch_size, shuffle=True)

    valid_loader = torch.utils.data.DataLoader(valid, batch_size=batch_size, shuffle=False)

    print(f'Fold {i + 1}')
    for epoch in range(n_epochs):
        # set train mode of the model. This enables operations which are only applied during training l
ike dropout
        start_time = time.time()
        model.train()

        avg_loss = 0.
        for i, (x_batch, y_batch, index) in enumerate(train_loader):
            # Forward pass: compute predicted y by passing x to the model.
            ################################################################################################
#####
            f = kfold_X_features[index]
            y_pred = model([x_batch,f])
            ################################################################################################
#####

            ################################################################################################
#####

            if scheduler:
                scheduler.batch_step()
            ################################################################################################
#####


            # Compute and print loss.
            loss = loss_fn(y_pred, y_batch)

            # Before the backward pass, use the optimizer object to zero all of the
            # gradients for the Tensors it will update (which are the learnable weights
            # of the model)
            optimizer.zero_grad()

            # Backward pass: compute gradient of the loss with respect to model parameters
            loss.backward()

            # Calling the step function on an Optimizer makes an update to its parameters
            optimizer.step()
            avg_loss += loss.item() / len(train_loader)

        # set evaluation mode of the model. This disabled operations which are only applied during trai
ning like dropout
        model.eval()

        # predict all the samples in y_val_fold batch per batch
        valid_preds_fold = np.zeros((x_val_fold.size(0)))
        test_preds_fold = np.zeros((len(df_test)))

        avg_val_loss = 0.
        for i, (x_batch, y_batch, index) in enumerate(valid_loader):
            f = kfold_X_valid_features[index]
            y_pred = model([x_batch,f]).detach()

            avg_val_loss += loss_fn(y_pred, y_batch).item() / len(valid_loader)
            valid_preds_fold[i * batch_size:(i+1) * batch_size] = sigmoid(y_pred.cpu().numpy())[:, 0]

        elapsed_time = time.time() - start_time
        print('Epoch {}/{} \t loss={:.4f} \t val_loss={:.4f} \t time={:.2f}s'.format(
            epoch + 1, n_epochs, avg_loss, avg_val_loss, elapsed_time))
    avg_losses_f.append(avg_loss)
    avg_val_losses_f.append(avg_val_loss)
    # predict all samples in the test set batch per batch
    for i, (x_batch,) in enumerate(test_loader):
        f = test_features[i * batch_size:(i+1) * batch_size]
        y_pred = model([x_batch,f]).detach()

        test_preds_fold[i * batch_size:(i+1) * batch_size] = sigmoid(y_pred.cpu().numpy())[:, 0]

    train_preds[valid_idx] = valid_preds_fold
    test_preds += test_preds_fold / len(splits)

print('All \t loss={:.4f} \t val_loss={:.4f} \t '.format(np.average(avg_losses_f),np.average(avg_val_lo
sses_f)))

# x_train, x_test_f, y_train, y_test_f
```

Find final Thresshold

Borrowed from: https://www.kaggle.com/ziliwang/baseline-pytorch-bilstm

In [ ]:

```python
def bestThresshold(y_train,train_preds):
    tmp = [0,0,0] # idx, cur, max
    delta = 0
    for tmp[0] in tqdm(np.arange(0.1, 0.501, 0.01)):
        tmp[1] = f1_score(y_train, np.array(train_preds)>tmp[0])
        if tmp[1] > tmp[2]:
            delta = tmp[0]
            tmp[2] = tmp[1]
    print('best threshold is {:.4f} with F1 score: {:.4f}'.format(delta, tmp[2]))
    return delta
delta = bestThresshold(y_train,train_preds)
```

In [ ]:

```python
submission = df_test[['qid']].copy()
submission['prediction'] = (test_preds > delta).astype(int)
submission.to_csv('submission.csv', index=False)
```

In [ ]:

```python
!head submission.csv
```