Preface

This kernel is a fork of this kernel made to work on Fast.AI and
Uses Weighted BCE Loss as described in this kernel.
Other than that nothing else has been changed. All improvemnts mentioned here could still apply.

Imports & Utility functions

In [1]:
```
from fastai.train import Learner
from fastai.train import DataBunch
from fastai.callbacks import *
from fastai.basic_data import DatasetType
```

In [2]:
```
import numpy as np
import pandas as pd
import os
import time
import gc
import random
from tqdm._tqdm_notebook import tqdm_notebook as tqdm
from keras.preprocessing import text, sequence
import torch
from torch import nn
from torch.utils import data
from torch.nn import functional as F
```

```
Using TensorFlow backend.
```

In [3]:
```
# disable progress bars when submitting
def is_interactive():
    return 'SHLVL' not in os.environ

if not is_interactive():
    def nop(it, *a, **k):
        return it

    tqdm = nop
```

In [4]:
```
def seed_everything(seed=1234):
    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()
```

In [5]:
```
CRAWL_EMBEDDING_PATH = '../input/fasttext-crawl-300d-2m/crawl-300d-2M.vec'
GLOVE_EMBEDDING_PATH = '../input/glove840b300dtxt/glove.840B.300d.txt'
NUM_MODELS = 2
LSTM_UNITS = 128
DENSE_HIDDEN_UNITS = 4 * LSTM_UNITS
MAX_LEN = 220
```

In [6]:
```
def get_coefs(word, *arr):
    return word, np.asarray(arr, dtype='float32')

def load_embeddings(path):
    with open(path) as f:
        return dict(get_coefs(*line.strip().split(' ')) for line in tqdm(f))

def build_matrix(word_index, path):
    embedding_index = load_embeddings(path)
    embedding_matrix = np.zeros((len(word_index) + 1, 300))
    unknown_words = []

    for word, i in word_index.items():
        try:
            embedding_matrix[i] = embedding_index[word]
        except KeyError:
            unknown_words.append(word)
    return embedding_matrix, unknown_words
```

In [7]:
```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def train_model(learn,test,output_dim,lr=0.001,
                batch_size=512, n_epochs=4,
                enable_checkpoint_ensemble=True):

    all_test_preds = []
    checkpoint_weights = [2 ** epoch for epoch in range(n_epochs)]
    test_loader = torch.utils.data.DataLoader(test, batch_size=batch_size, shuffle=False)
    n = len(learn.data.train_dl)
    phases = [(TrainingPhase(n).schedule_hp('lr', lr * (0.6**(i)))) for i in range(n_epochs)]
    sched = GeneralScheduler(learn, phases)
    learn.callbacks.append(sched)
    for epoch in range(n_epochs):
        learn.fit(1)
        test_preds = np.zeros((len(test), output_dim))
        for i, x_batch in enumerate(test_loader):
            X = x_batch[0].cuda()
            y_pred = sigmoid(learn.model(X).detach().cpu().numpy())
            test_preds[i * batch_size:(i+1) * batch_size, :] = y_pred

        all_test_preds.append(test_preds)


    if enable_checkpoint_ensemble:
        test_preds = np.average(all_test_preds, weights=checkpoint_weights, axis=0)
    else:
        test_preds = all_test_preds[-1]

    return test_preds
```

In [8]:
```
class SpatialDropout(nn.Dropout2d):
    def forward(self, x):
        x = x.unsqueeze(2)    # (N, T, 1, K)
        x = x.permute(0, 3, 2, 1)  # (N, K, 1, T)
        x = super(SpatialDropout, self).forward(x)  # (N, K, 1, T), some features are masked
        x = x.permute(0, 3, 2, 1)  # (N, T, 1, K)
        x = x.squeeze(2)  # (N, T, K)
        return x

class NeuralNet(nn.Module):
    def __init__(self, embedding_matrix, num_aux_targets):
        super(NeuralNet, self).__init__()
        embed_size = embedding_matrix.shape[1]

        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 = SpatialDropout(0.3)

        self.lstm1 = nn.LSTM(embed_size, LSTM_UNITS, bidirectional=True, batch_first=True)
        self.lstm2 = nn.LSTM(LSTM_UNITS * 2, LSTM_UNITS, bidirectional=True, batch_first=True)

        self.linear1 = nn.Linear(DENSE_HIDDEN_UNITS, DENSE_HIDDEN_UNITS)
        self.linear2 = nn.Linear(DENSE_HIDDEN_UNITS, DENSE_HIDDEN_UNITS)

        self.linear_out = nn.Linear(DENSE_HIDDEN_UNITS, 1)
        self.linear_aux_out = nn.Linear(DENSE_HIDDEN_UNITS, num_aux_targets)

    def forward(self, x):
        h_embedding = self.embedding(x)
        h_embedding = self.embedding_dropout(h_embedding)

        h_lstm1, _ = self.lstm1(h_embedding)
        h_lstm2, _ = self.lstm2(h_lstm1)

        # global average pooling
        avg_pool = torch.mean(h_lstm2, 1)
        # global max pooling
        max_pool, _ = torch.max(h_lstm2, 1)

        h_conc = torch.cat((max_pool, avg_pool), 1)
        h_conc_linear1  = F.relu(self.linear1(h_conc))
        h_conc_linear2  = F.relu(self.linear2(h_conc))

        hidden = h_conc + h_conc_linear1 + h_conc_linear2

        result = self.linear_out(hidden)
        aux_result = self.linear_aux_out(hidden)
        out = torch.cat([result, aux_result], 1)

        return out
```

In [9]:
```
def preprocess(data):
    '''
    Credit goes to https://www.kaggle.com/gpreda/jigsaw-fast-compact-solution
    '''
    punct = "/-'?!.,#$%\'()*+-/:;<=>@[\\]^_`{|}~" + '""“”’' + '∞θ÷α−β∅³π‘₹´²—–&'
    def clean_special_chars(text, punct):
        for p in punct:
            text = text.replace(p, ' ')
        return text

    data = data.astype(str).apply(lambda x: clean_special_chars(x, punct))
    return data
```

Preprocessing

In [10]:
```
train = pd.read_csv('../input/jigsaw-unintended-bias-in-toxicity-classification/train.csv')
test = pd.read_csv('../input/jigsaw-unintended-bias-in-toxicity-classification/test.csv')
```

In [11]:
```
x_train = preprocess(train['comment_text'])
y_aux_train = train[['target', 'severe_toxicity', 'obscene', 'identity_attack', 'insult', 'threat']]
x_test = preprocess(test['comment_text'])

identity_columns = [
    'male', 'female', 'homosexual_gay_or_lesbian', 'christian', 'jewish',
    'muslim', 'black', 'white', 'psychiatric_or_mental_illness']
# Overall
weights = np.ones((len(x_train),)) / 4
# Subgroup
weights += (train[identity_columns].fillna(0).values>=0.5).sum(axis=1).astype(bool).astype(np.int) / 4
# Background Positive, Subgroup Negative
weights += (( (train['target'].values>=0.5).astype(bool).astype(np.int) +
   (train[identity_columns].fillna(0).values<0.5).sum(axis=1).astype(bool).astype(np.int) ) > 1 ).astype(bool).astype(np.int) / 4
# Background Negative, Subgroup Positive
weights += (( (train['target'].values<0.5).astype(bool).astype(np.int) +
   (train[identity_columns].fillna(0).values>=0.5).sum(axis=1).astype(bool).astype(np.int) ) > 1 ).astype(bool).astype(np.int) / 4
loss_weight = 1.0 / weights.mean()
```

In [12]:
```
y_train = np.vstack([(train['target'].values>=0.5).astype(np.int),weights]).T
```

In [13]:
```
max_features = None
```

In [14]:
```
tokenizer = text.Tokenizer()
tokenizer.fit_on_texts(list(x_train) + list(x_test))

x_train = tokenizer.texts_to_sequences(x_train)
x_test = tokenizer.texts_to_sequences(x_test)
x_train = sequence.pad_sequences(x_train, maxlen=MAX_LEN)
x_test = sequence.pad_sequences(x_test, maxlen=MAX_LEN)
```

In [15]:
```
max_features = max_features or len(tokenizer.word_index) + 1
max_features
```

Out[15]:
```
327576
```

In [16]:
```
crawl_matrix, unknown_words_crawl = build_matrix(tokenizer.word_index, CRAWL_EMBEDDING_PATH)
print('n unknown words (crawl): ', len(unknown_words_crawl))
```

```
n unknown words (crawl):  174141
```

In [17]:
```
glove_matrix, unknown_words_glove = build_matrix(tokenizer.word_index, GLOVE_EMBEDDING_PATH)
print('n unknown words (glove): ', len(unknown_words_glove))
```

```
n unknown words (glove):  170837
```

In [18]:
```
embedding_matrix = np.concatenate([crawl_matrix, glove_matrix], axis=-1)
embedding_matrix.shape

del crawl_matrix
del glove_matrix
gc.collect()
```

Out[18]:
```
0
```

In [19]:
```
x_train_torch = torch.tensor(x_train, dtype=torch.long)
y_train_torch = torch.tensor(np.hstack([y_train, y_aux_train]), dtype=torch.float32)
```

In [20]:
```
x_test_torch = torch.tensor(x_test, dtype=torch.long)
```

Training

In [21]:
```
batch_size = 512

train_dataset = data.TensorDataset(x_train_torch, y_train_torch)
valid_dataset = data.TensorDataset(x_train_torch[:batch_size], y_train_torch[:batch_size])
test_dataset = data.TensorDataset(x_test_torch)

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
valid_loader = torch.utils.data.DataLoader(valid_dataset, batch_size=batch_size, shuffle=False)

databunch = DataBunch(train_dl=train_loader,valid_dl=valid_loader)
```

In [22]:
```
def custom_loss(data, targets):
    ''' Define custom loss function for weighted BCE on 'target' column '''
    bce_loss_1 = nn.BCEWithLogitsLoss(weight=targets[:,1:2])(data[:,:2],targets[:,:1])
    bce_loss_2 = nn.BCEWithLogitsLoss()(data[:,1:],targets[:,2:])
    return (bce_loss_1 * loss_weight) + bce_loss_2
```

In [23]:
```
all_test_preds = []

for model_idx in range(NUM_MODELS):
    print('Model ', model_idx)
    seed_everything(1234 + model_idx)
    model = NeuralNet(embedding_matrix, y_aux_train.shape[-1])
    learn = Learner(databunch,model,loss_func=custom_loss)
    test_preds = train_model(learn,test_dataset,output_dim=7)
    all_test_preds.append(test_preds)
```

```
Model  0
```

0.00% [0/1 00:00<00:00]

epoch	train_loss	valid_loss	time

82.81% [2920/3526 08:08<01:41 0.2842]

In [24]:
```
submission = pd.DataFrame.from_dict({
    'id': test['id'],
    'prediction': np.mean(all_test_preds, axis=0)[:, 0]
})

submission.to_csv('submission.csv', index=False)
```

Note that the solution is not validated in this kernel. So for tuning anything, you should build a validation framework using e. g. KFold CV. If you just check what works best by submitting, you are very likely to overfit to the public LB.