Preface

This kernel is a PyTorch version of the Simple LSTM kernel. All credit for architecture and preprocessing goes to @thousandvoices. There is a lot of discussion whether Keras, PyTorch, Tensorflow or the CUDA C API is best. But specifically between the PyTorch and Keras version of the simple LSTM architecture, there are 2 clear advantages of PyTorch:

- Speed. The PyTorch version runs about 20 minutes faster.
- Determinism. The PyTorch version is fully deterministic. Especially when it gets harder to improve your score later in the competition, determinism is very important.

I was surprised to see that PyTorch is that much faster, so I'm not completely sure the steps taken are exactly the same. If you see any difference, we can discuss it in the comments :)

The most likely reason the score of this kernel is higher than the @thousandvoices version is that the optimizer is not reinitialized after every epoch and thus the parameter-specific learning rates of Adam are not discarded after every epoch. That is the only difference between the kernels that is intended.

Imports & Utility functions

In [1]:
```
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 [2]:
```
# 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 [3]:
```
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 [4]:
```
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 [5]:
```
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 [6]:
```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def train_model(model, train, test, loss_fn, output_dim, lr=0.001,
                batch_size=512, n_epochs=4,
                enable_checkpoint_ensemble=True):
    param_lrs = [{'params': param, 'lr': lr} for param in model.parameters()]
    optimizer = torch.optim.Adam(param_lrs, lr=lr)

    scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer, lambda epoch: 0.6 ** epoch)

    train_loader = torch.utils.data.DataLoader(train, batch_size=batch_size, shuffle=True)
    test_loader = torch.utils.data.DataLoader(test, batch_size=batch_size, shuffle=False)
    all_test_preds = []
    checkpoint_weights = [2 ** epoch for epoch in range(n_epochs)]

    for epoch in range(n_epochs):
        start_time = time.time()

        scheduler.step()

        model.train()
        avg_loss = 0.

        for data in tqdm(train_loader, disable=False):
            x_batch = data[:-1]
            y_batch = data[-1]

            y_pred = model(*x_batch)
            loss = loss_fn(y_pred, y_batch)

            optimizer.zero_grad()
            loss.backward()

            optimizer.step()
            avg_loss += loss.item() / len(train_loader)

        model.eval()
        test_preds = np.zeros((len(test), output_dim))

        for i, x_batch in enumerate(test_loader):
            y_pred = sigmoid(model(*x_batch).detach().cpu().numpy())

            test_preds[i * batch_size:(i+1) * batch_size, :] = y_pred

        all_test_preds.append(test_preds)
        elapsed_time = time.time() - start_time
        print('Epoch {}/{} \t loss={:.4f} \t time={:.2f}s'.format(
              epoch + 1, n_epochs, avg_loss, elapsed_time))

    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 [7]:
```
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 [8]:
```
def preprocess(data):
    '''
    Credit goes to https://www.kaggle.com/gpreda/jigsaw-fast-compact-solution
    '''
    punct = "/-'?!.,#$%\'()*+-/:;<=>@[\\]^_`{|}~`" + '""“”’' + '∞θ÷α•à−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 [9]:
```
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')

x_train = preprocess(train['comment_text'])
y_train = np.where(train['target'] >= 0.5, 1, 0)
y_aux_train = train[['target', 'severe_toxicity', 'obscene', 'identity_attack', 'insult', 'threat']]
x_test = preprocess(test['comment_text'])
```

In [10]:
```
max_features = None
```

In [11]:
```
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 [12]:
```
max_features = max_features or len(tokenizer.word_index) + 1
max_features
```

Out[12]:
```
327576
```

In [13]:
```
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 [14]:
```
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 [15]:
```
embedding_matrix = np.concatenate([crawl_matrix, glove_matrix], axis=-1)
embedding_matrix.shape

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

Out[15]:
```
0
```

In [16]:
```
x_train_torch = torch.tensor(x_train, dtype=torch.long).cuda()
x_test_torch = torch.tensor(x_test, dtype=torch.long).cuda()
y_train_torch = torch.tensor(np.hstack([y_train[:, np.newaxis], y_aux_train]), dtype=torch.float32).cud
a()
```

Training

In [17]:
```
train_dataset = data.TensorDataset(x_train_torch, y_train_torch)
test_dataset = data.TensorDataset(x_test_torch)

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])
    model.cuda()

    test_preds = train_model(model, train_dataset, test_dataset, output_dim=y_train_torch.shape[-1],
                             loss_fn=nn.BCEWithLogitsLoss(reduction='mean'))
    all_test_preds.append(test_preds)
    print()
```

```
Model  0
Epoch 1/4 	 loss=0.1096 	 time=591.49s
Epoch 2/4 	 loss=0.1034 	 time=589.38s
Epoch 3/4 	 loss=0.1020 	 time=591.26s
Epoch 4/4 	 loss=0.1011 	 time=589.24s

Model  1
Epoch 1/4 	 loss=0.1097 	 time=593.64s
Epoch 2/4 	 loss=0.1034 	 time=590.37s
Epoch 3/4 	 loss=0.1020 	 time=591.32s
Epoch 4/4 	 loss=0.1011 	 time=590.00s
```

In [18]:
```
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.

Ways to improve this kernel

This kernel is just a simple baseline kernel, so there are many ways to improve it. Some ideas to get you started:

- Add a contraction mapping. E. g. mapping "is'nt" to "is not" can help the network because "not" is explicitly mentioned. They were very popular in the recent quora competition, see for example this kernel.
- Try to reduce the number of words that are not found in the embeddings. At the moment, around 170k words are not found. We can take some steps to decrease this amount, for example trying to find a vector for a processed (capitalized, stemmed, ...) version of the word when the vector for the regular word can not be found. See the 3rd place solution of the quora competition for an excellent implementation of this.
- Try cyclic learning rate (CLR). I have found CLR to almost always improve my network recently compared to the default parameters for Adam. In this case, we are already using a learning rate scheduler, so this might not be the case. But it is still worth to try it out. See for example my my other PyTorch kernel for an implementation of CLR in PyTorch.
- Use sequence bucketing to train faster and fit more networks into the two hours. The winning team of the quora competition successfully used sequence bucketing to drastically reduce the time it took to train RNNs. An excerpt from their solution summary:

  > We aimed at combining as many models as possible. To do this, we needed to improve runtime and the most important thing to achieve this was the following. We do not pad sequences to the same length based on the whole data, but just on a batch level. That means we conduct padding and truncation on the data generator level for each batch separately, so that length of the sentences in a batch can vary in size. Additionally, we further improved this by not truncating based on the length of the longest sequence in the batch, but based on the 95% percentile of lengths within the sequence. This improved runtime heavily and kept accuracy quite robust on single model level, and improved it by being able to average more models.

- Try a (weighted) average of embeddings instead of concatenating them. A 600d vector for each word is a lot, it might work better to average them instead. See this paper for why this even works.
- Limit the maximum number of words used to train the NN. At the moment, there is no limit set to the maximum number of words in the tokenizer, so we use every word that occurs in the training data, even if it is only mentioned once. This could lead to overfitting so it might be better to limit the maximum number of words to e. g. 100k.

Thanks for reading. Good luck and have fun in this competition!