In [ ]:
```
import re
import time
import gc
import random
import os

import numpy as np
import pandas as pd

from tqdm import tqdm
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.model_selection import GridSearchCV, StratifiedKFold
from sklearn.metrics import f1_score, roc_auc_score

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

import torch
import torch.nn as nn
import torch.utils.data

def seed_torch(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
```

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 = 72 # max number of words in a question to use

batch_size = 1536
train_epochs = 8

SEED = 1029
```

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
```

In [ ]:
```
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
```

In [ ]:
```
mispell_dict = {"aren't" : "are not",
"can't" : "cannot",
"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",
"i'd" : "I would",
"i'll" : "I will",
"i'm" : "I am",
"isn't" : "is not",
"it's" : "it is",
"it'll":"it will",
"i've" : "I have",
"let's" : "let us",
"mightn't" : "might not",
"mustn't" : "must not",
"shan't" : "shall not",
"she'd" : "she would",
"she'll" : "she will",
"she's" : "she is",
"shouldn't" : "should not",
"that's" : "that is",
"there's" : "there is",
"they'd" : "they would",
"they'll" : "they will",
"they're" : "they are",
"they've" : "they have",
"we'd" : "we would",
"we're" : "we are",
"weren't" : "were not",
"we've" : "we have",
"what'll" : "what will",
"what're" : "what are",
"what's" : "what is",
"what've" : "what have",
"where's" : "where is",
"who'd" : "who would",
"who'll" : "who will",
"who're" : "who are",
"who's" : "who is",
"who've" : "who have",
"won't" : "will not",
"wouldn't" : "would not",
"you'd" : "you would",
"you'll" : "you will",
"you're" : "you are",
"you've" : "you have",
"'re": " are",
"wasn't": "was not",
"we'll":"will",
"didn't": "did not",
"tryin'":"trying"}

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)
```

In [ ]:
```
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"].progress_apply(lambda x: x.lower())
    test_df["question_text"] = test_df["question_text"].progress_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"].progress_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"].progress_apply(lambda x: clean_numbers(x))

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

    ## 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)

    ## Get 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

    #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]

    return train_X, test_X, train_y, tokenizer.word_index
```

In [ ]:
```
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')
    embeddings_index = dict(get_coefs(*o.split(" ")) for o in open(EMBEDDING_FILE))

    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))
    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
```

In [ ]:
```
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", error="ignore") 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))
    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
```

In [ ]:
```
from tqdm import tqdm
tqdm.pandas()

start_time = time.time()

train_X, test_X, train_y, word_index = load_and_prec()
embedding_matrix_1 = load_glove(word_index)
embedding_matrix_2 = load_para(word_index)

total_time = (time.time() - start_time) / 60
print("Took {:.2f} minutes".format(total_time))
```

```
embedding_matrix = np.mean([embedding_matrix_1, embedding_matrix_2], axis=0)
# embedding_matrix = np.concatenate((embedding_matrix_1, embedding_matrix_2), axis=1)
print(np.shape(embedding_matrix))
```

```
del embedding_matrix_1, embedding_matrix_2
gc.collect()
```

In [ ]:
```
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)
```

In [ ]:
```
class NeuralNet(nn.Module):
    def __init__(self):
        super(NeuralNet, self).__init__()

        hidden_size = 60

        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.GRU(embed_size, hidden_size, bidirectional=True, batch_first=True)
        self.gru = nn.GRU(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.linear = nn.Linear(480, 16)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.1)
        self.out = nn.Linear(16, 1)

    def forward(self, x):
        h_embedding = self.embedding(x)
        h_embedding = torch.squeeze(self.embedding_dropout(torch.unsqueeze(h_embedding, 0)))

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

        h_lstm_atten = self.lstm_attention(h_lstm)
        h_gru_atten = self.gru_attention(h_gru)

        avg_pool = torch.mean(h_gru, 1)
        max_pool, _ = torch.max(h_gru, 1)

        conc = torch.cat((h_lstm_atten, h_gru_atten, avg_pool, max_pool), 1)
        conc = self.relu(self.linear(conc))
        conc = self.dropout(conc)
        out = self.out(conc)

        return out
```

In [ ]:
```
splits = list(StratifiedKFold(n_splits=5, shuffle=True, random_state=SEED).split(train_X, train_y))
```

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

In [ ]:
```
def threshold_search(y_true, y_proba):
    best_threshold = 0
    best_score = 0
    for threshold in tqdm(i * 0.01 for i in range(100)):
        score = f1_score(y_true=y_true, y_pred=y_proba > threshold)
        if score > best_score:
            best_threshold = threshold
            best_score = score
    search_result = {'threshold': best_threshold, 'f1': best_score}
    return search_result
```

In [ ]:
```
train_preds = np.zeros((len(train_X)))
test_preds = np.zeros((len(test_X)))

seed_torch(SEED)

x_test_cuda = torch.tensor(test_X, 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)

for i, (train_idx, valid_idx) in enumerate(splits):
    x_train_fold = torch.tensor(train_X[train_idx], dtype=torch.long).cuda()
    y_train_fold = torch.tensor(train_y[train_idx, np.newaxis], dtype=torch.float32).cuda()
    x_val_fold = torch.tensor(train_X[valid_idx], dtype=torch.long).cuda()
    y_val_fold = torch.tensor(train_y[valid_idx, np.newaxis], dtype=torch.float32).cuda()

    model = NeuralNet()
    model.cuda()

    loss_fn = torch.nn.BCEWithLogitsLoss(reduction="sum")
    optimizer = torch.optim.Adam(model.parameters())

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

    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('Fold {} '.format(i + 1))

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

        model.train()
        avg_loss = 0.
        for x_batch, y_batch in tqdm(train_loader, disable=True):
            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()
        valid_preds_fold = np.zeros((x_val_fold.size(0)))
        test_preds_fold = np.zeros((len(test_X)))
        avg_val_loss = 0.
        for j, (x_batch, y_batch) in enumerate(valid_loader):
            y_pred = model(x_batch).detach()
            avg_val_loss += loss_fn(y_pred, y_batch).item() / len(valid_loader)
            valid_preds_fold[j * batch_size:(j+1) * batch_size] = sigmoid(y_pred.cpu().numpy())[:, 0]

        elapsed_time = (time.time() - start_time) / 60
        print('Epoch {}/{} \t loss={:.4f} \t val_loss={:.4f} \t time={:.2f}s'.format(
            epoch + 1, train_epochs, avg_loss, avg_val_loss, elapsed_time))

    for j, (x_batch,) in enumerate(test_loader):
        y_pred = model(x_batch).detach()

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

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

In [ ]:
```
search_result = threshold_search(train_y, train_preds)
search_result
```

In [ ]:
```
sub = pd.read_csv('../input/sample_submission.csv')
sub.prediction = test_preds > search_result['threshold']
sub.to_csv("submission.csv", index=False)
```