Quora Insincere Question Selection

Peihong Yu Xinru Zhang Mengjie Min
Stevens Institute Of Technology
pyu7@stevens.edu xzhan63@stevens.edu mmin@stevens.edu

Abstract

In this Kaggle competition, we aim at finding out inappropriate comments from Quora website by building a binary classification model. We separate our whole work into three steps. First, we use word embedding method to map each text into corresponding data. Second, we tried three different models to train the model. The approaches we adopt to solve the problem are "GRU", "LSTM" and "Attention". Finally the evaluation will base on the F1-score between the predicted score and the target score. Owing to the incorrect labels in datasets, the highest score in this competition is about 0.71. Our results is around 0.68 on the premise of the datasets' accuracy. Since attention model is so hot and has a high performance in NLP, we can adopt that tech to this deal with this problem. With paying more attention on parts of text, we got a higher score using attention model.

1 Introduction

Quora is a website for people to communicate with others. But sometimes inappropriate comments appear. Till now, the Quora has already implemented machine learning and hand-operated ways to decrease the possibility of insincere questions. In order to combat insincere questions more efficiency, help Quora maintain their policy with "Be Nice, Be Respectful". We need to find more up-gradable ways to discover these ambiguous and confusing comments.

Before introducing models, we want to clean the dataset first. If we feed our model with cleaner dataset, we can obtain a better result. Notice from the original data set, there are many disturbing characters. For the data pre-processing part, we majorly separate into three steps:

1) Obtaining a dictionary by loading words from wiki-news-300d-1M.vec.
2) Checking percentage of words from the training dataset also can found in "wiki" dataset.
3) Discarding the punctuation, spaces and replace the misspell words into correct spell words.
4) Selecting replaceable words in wiki datasets and replace the “cannot find” words with them.

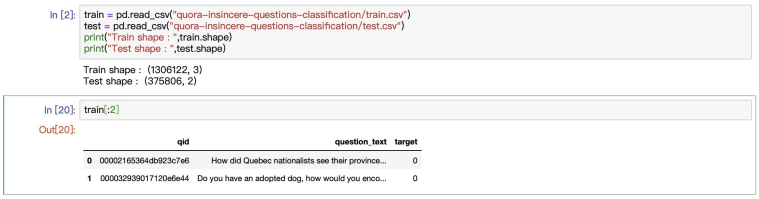
Base on the organizer's command, we need to build a classifier to find out insincere questions. Dealing with the relationship with language, RNN is the general model we first think to build our model. So our approach is motivated by the success of RNN model in natural language processing field. Firstly, we use LSTM model, which is an improvement model of RNN. The LSTM model can choose some parts from the past can be kept and others can be forgotten, not like the general RNN model, selecting the most recent information without filtering. Furthermore, we also introduce GRU to build our model. Compared to LSTM, the commonality is that two models both keep the important information. The difference is that GRU model has fewer parameters than LSTM model, so the training speed is faster than LSTM.

32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.

In principle, by submitting results to kaggle, the highest score we got is 0.68506 and have a rank at 1050 among total 4037 groups , which is quite high at this point since the original dataset contains many wrong labels. At this time, although we have LSTM, GRU and Attention modules, hyperparameters are not the best, so we need to try more in the future.

2 Data Pre-processing

The first and most important part of our project is data preprocessing. We print the shape of training dataset and testing dataset on the first hand, the output shows that we have about 1306122 rows and 3 columns of data. The file train.csv has three columns: qid, question_text, target. "Question_text" column is composed of multiple sentences. "Target" column has binary values. "1" means that this sentence is identified as insincere. It will speed a lot of time running all the training dataset, so we choose to statistics the frequency of words and filter higher frequency of words to the next step.



```
In [2]: train = pd.read_csv("quora-insincere-questions-classification/train.csv")
        test = pd.read_csv("quora-insincere-questions-classification/test.csv")
        print("Train shape : ",train.shape)
        print("Test shape : ",test.shape)

        Train shape :  (1306122, 3)
        Test shape :  (375806, 2)

In [20]: train[:2]
Out[20]:
```

	qid	question_text	target
0	00002165364db923c7e6	How did Quebec nationalists see their province...	0
1	000032939017120e6e44	Do you have an adopted dog, how would you enco...	0

Figure 1: Information of training dataset

So how can we define whether a question is insincere or not? The key point is that we can capture some specific characters to decide the question is insincere. Thanks to the competition holder who generously provides us four datasets to help us with data processing part. We can use one of the datasets to match words so that we can judge is this word a good word or a bad word.

How we choose the high frequency words is that we build a dictionary named "vocab", each time the word appears from the training dataset, we add "1" to the value. Finally, we can add up totally numbers of each word. From the figure below, we enumerate the frequency of words in the first 5 rows.

```
In[3]:
def build_vocab(sentences, verbose =  True):
    """
    :param sentences: list of list of words
    :return: dictionary of words and their count
    """
    vocab = {}
    for sentence in tqdm(sentences, disable = (not verbose)):
        for word in sentence:
            try:
                vocab[word] += 1
            except KeyError:
                vocab[word] = 1
    return vocab

+ Code   + Markdown

In[4]:
sentences = train["question_text"].progress_apply(lambda x: x.split()).values
vocab = build_vocab(sentences) # vocab: frequency dictionary about quora text
print({k: vocab[k] for k in list(vocab)[:5]})

100%|| 1306122/1306122 [00:06<00:00, 187170.58it/s]
100%|| 1306122/1306122 [00:06<00:00, 215829.47it/s]

{'How': 261930, 'did': 33489, 'Quebec': 97, 'nationalists': 91, 'see': 9003}
```

Figure 2: First 10 rows of non-found words

As we can see on average questions in train and test datasets are similar, but there are quite long questions in train dataset.

We can see that most of the questions are 40 words long or shorter. Let's try having sequence length equal to 71 for now. So in the next model step, we can set the maximum length of words in each

```
In [5]: print('Average word length of questions in train is {0:.0f}.'.format(np.mean(train['question_text'].apply(lambda x: len(x.split())))))
        print('Average word length of questions in test is {0:.0f}.'.format(np.mean(test['question_text'].apply(lambda x: len(x.split())))))

        Average word length of questions in train is 13.
        Average word length of questions in test is 13.

In [6]: print('Max word length of questions in train is {0:.0f}.'.format(np.max(train['question_text'].apply(lambda x: len(x.split())))))
        print('Max word length of questions in test is {0:.0f}.'.format(np.max(test['question_text'].apply(lambda x: len(x.split())))))

        Max word length of questions in train is 134.
        Max word length of questions in test is 87.

In [7]: print('Average character length of questions in train is {0:.0f}.'.format(np.mean(train['question_text'].apply(lambda x: len(x)))))
        print('Average character length of questions in test is {0:.0f}.'.format(np.mean(test['question_text'].apply(lambda x: len(x)))))

        Average character length of questions in train is 71.
        Average character length of questions in test is 71.
```

Figure 3: Average length of questions in training dataset and testing dataset

sentence is 72.

```
In [16]: train['question_text'].apply(lambda x: len(x.split())).plot(kind='hist')
         plt.yscale('log')
         plt.title('Distribution of question text length in characters')

Out[16]: Text(0.5, 1.0, 'Distribution of question text length in characters')
```

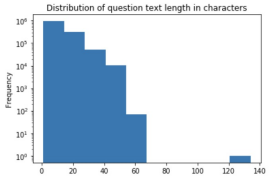


Figure 4: Distribution of question text length in characters

Besides, we want to compare words in the wiki dataset and words in the training dataset to detect whether they are good words or not. In order to avoid spending too much time loading the dataset from the wiki each time when we want to make comparsions, we use "KeyedVectors" library to build a "embeddings_index". "KeyedVectors" library can help words map to one-dimensional vectors. To the nature of vectors, we can calculate the distance between these vectors to determine are they synonyms.

```
In[5]:
from gensim.models import KeyedVectors
if 'embeddings_index' not in globals():
    embeddings_index = KeyedVectors.load_word2vec_format("../input/quora-insincere-questions-classification/embeddings/wiki-
else:
    print('embeddings_index already exists')
# word_vectors
```

Figure 5: Use KeyedVectors library to change words into one-dimensional vector

Next question appears in our mind, is that each word in the training dataset can be found according to the "wiki" dataset? So we check the coverage between vocabs in the training dataset's sentences and "wiki" dataset. Surprising find that there around 30.05% percent of words from sentences in the training dataset also show in the "wiki" dataset. 87.66% represent the total frequency of words from the training dataset, which also shows in the "wiki" dataset among the total number of words in the "wiki" dataset and training dataset. This result promote us to doing the next step.

```
def check_coverage(vocab,embeddings_index):
    a = {}
    oov = {}
    k = 0
    i = 0
    for word in tqdm(vocab):
        try:
            a[word] = embeddings_index[word]
            k += vocab[word]
        except:

            oov[word] = vocab[word]
            i += vocab[word]
            pass

    print('Found embeddings for {:.2%} of vocab'.format(len(a) / len(vocab)))
    print('Found embeddings for  {:.2%} of all text'.format(k / (k + i)))
    sorted_x = sorted(oov.items(), key=operator.itemgetter(1))[::-1]

    return sorted_x
```

+ Code + Markdown

In[7]:
```
oov = check_coverage(vocab,embeddings_index)
```

```
100%|| 508823/508823 [00:02<00:00, 206407.85it/s]

Found embeddings for 30.05% of vocab
Found embeddings for  87.66% of all text
```

Figure 6: Percentage and frequency

We wonder why there is only around 30% of words from the training dataset can be found in the "wiki" dataset. So we print the first 10 rows of " cannot find " words among the "wiki" dataset. It shows that words like "me" and "do" these basic words also cannot find in the "wiki" dataset. The reason is that these words combine together with punctation like " ? " and " ’ ".

In[8]:
```
oov[:10]
```

Out[8]:
```
[('India?', 16384),
 ("don't", 14991),
 ('it?', 12900),
 ("I'm", 12811),
 ("What's", 12425),
 ('do?', 8753),
 ('life?', 7753),
 ("can't", 7077),
 ('you?', 6295),
 ('me?', 6202)]
```

Figure 7: First 10 rows of non-found words

Continue perusing the provided dataset, we notice that from the "wiki" dataset, it only includes two cases: Single words without punctation and punctation themselves. What’s more, the training dataset also includes abbreviations, misspellings and mix of lowercase and uppercase words. Thus, it’s necessary to spilt words and punctuations and replace words into correct format to improve our accuracy.

In case, we look up punctation first. We formulate 28 kinds of common punctation and use embeddings_ index to find whether it includes in the "wiki" dataset.

In[17]:
```
for punct in '?!.,"#$%\'()*+-/:;<=>@[\\]^_`{|}~' + '“”’':
    if punct in embeddings_index:
        print(punct)
    else:
        print('\t\t\t\t\t',punct)
```

Figure 8: Search for punctuations meet our requirement

The result is that we found two punctuations " _ " and "‘" are not in the "wiki" dataset. What’s more, we already aware that a word combined with a punctuation will be treated as a cannot find word. To fix this problem, we add spaces on both sides for punctations can be found in "wiki" dataset to separate the punctations and the word. Also, replace the other two cannot find punctuations with spaces.

```
In[10]:
# transfer this to kaggle code
def clean_text(x):
    x = str(x)
    for punct in '?!.,"#$%\'()*+-/:;<=>@[\\]^{|}~' + '“”’':
        x = x.replace(punct, f' {punct} ')
    for punct in '_`':
        x = x.replace(punct, ' ')
    return x
```

Figure 9: Split and replace punctuations

Next step is how to deal with “cannot find " words among the "wiki" dataset. So our group discuss two methods to decrease the amount of the word in oov(“cannot find " words). It is common that netizens often mistyped words or prefer abbreviations for convenience. Therefore, replacing Mistaken spelling words and abbreviated form with correct or appropriate words could increase the ratio of the total number of words in the "wiki" dataset .

```
In [13]:
mispell_dict = {'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','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",
'MeToo':'me too'}
```

Figure 10: Mistaken spelling and abbrevited words

When we have ruled out the possibility of mistyping, the length of “cannot find” words is still very high. How can we do? Is it possible that replace the “unknown” words with words among “wiki” datasets? Based on this assumption, we start to try some special methods. We first sort the 1000 “cannot find”words with high frequencies, and then separately count how many times they appears in good_sentence and bad_sentence in terms of labels(0 or 1) in train and test datasets. The percentage of each word able to lead one sentence to be bad(means the corresponding label equals to 1) could generate through the equation r = bad_sentence/(good_sentence+bad_sentence).

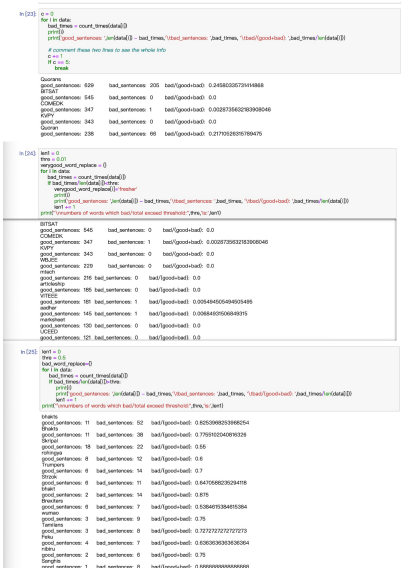


Figure 11: Calculate bad ratio of words that cannot find from wiki dataset

At the same time, select 1000 sentences respectively from good_sentence and bad_sentence. And calculate the ratio of each word in these sentences. We set up the threshold less than 0.01 and more than 0.5 in order to get replaceable according to similar ratio values.

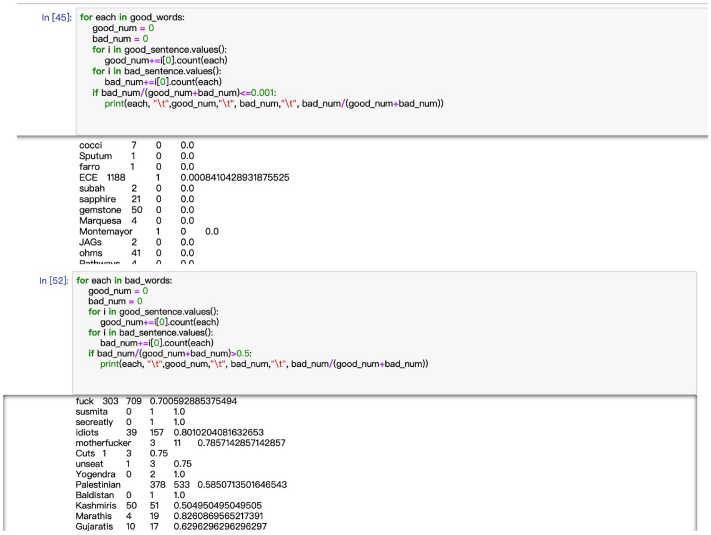


Figure 12: Calculate bad ratio of the replaceable words from good and bad sentences

For example, we can see that the ratio of ‘bhakts’ ia about 0.8254, and the word ‘Marathis’ has similar value. As to this situation, it is theoretically feasible to replace ‘bhakts’ with ‘Marathis’.

Therefore, our group implement several methods and then realize the goal of reducing the length of the list (“cannot find” words). Now the length of the list changes from 81253 to 79883.

3 Model Selection

In deep learning when we talk about text classification, the first and most popular model appears in our mind must be RNN model. As we read a sentence, we will not restart from the beginning of the sentence when we meet a new word. Our brain will comprehension from words we have already read and infer the meaning of new words. At this time, our own modules are inspired by Vanilla RNN, LSTM and GRU these three modules. Besides we also make stack and combination from three modules above.

3.1 Vanilla RNN

Recurrent neural networks is not a new topic in deep learning area. Many famous network company as Baidu, Google extensively use this technique with machine translation, speech recognition as well as a number of other tasks. Especially, almost all state of the art outcomes in NLP related tasks are achieved by exploiting RNNs. Before RNN, traditional neuron networks cannot contain persistance, it seems like a corrupt practices. But the occurane of RNN solve this problem.

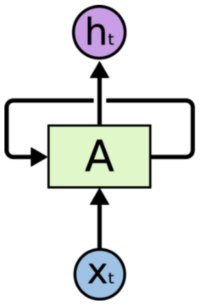


Figure 13: Single recurrent neural network model

The recurrent neural networks model is in Fig.13 above. During each time period, each node receives information from the previous node and the process can be represented by a feedback cycle. Fig.14 shows the details of this "feedback" cycle.

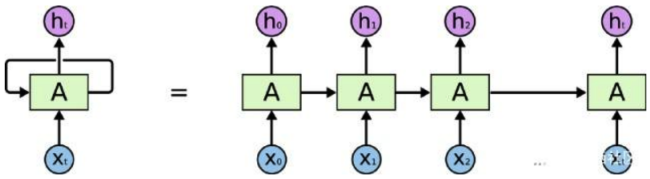


Figure 14: How recurrent neural network work

In each time process, we pick an input " xi " and output of previous node's output "ai - 1". Calculate them and get the output of the current node " hi ". Same, the output "hi" also provided to the next node as it's input. The cycle will continue running until all time process finish.

3.2 LSTM

The defect of RNN is that with the growth of time period, it cannot get effect information from the time period a long time before. If we want to know more information with "t+1", we probably need to realize the meaning from "0" and "1" in the time process. Due to the long distance, the model learns from "0" and "1" cannot be expressed to the node with " t+1 ". As far as we can see, RNN can only memorize the short information sequence.

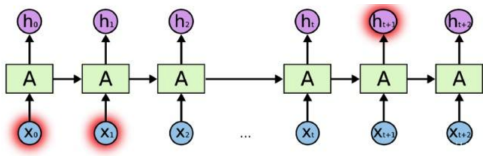


Figure 15: RNN cannot get information with long period before

As a result LSTM networks came. LSTM is not a totally new module, it's a kind of evolution from RNN module. In order to keep and transfer information for a long period of time is the default behavior of these networks. Comparing with the structure, both LSTM and RNNs have a chain like shape. But the repeating module from two models are quite different. The original repeating module of RNNs only has a single tanh layer, but LSTM module contains four interacting layers.

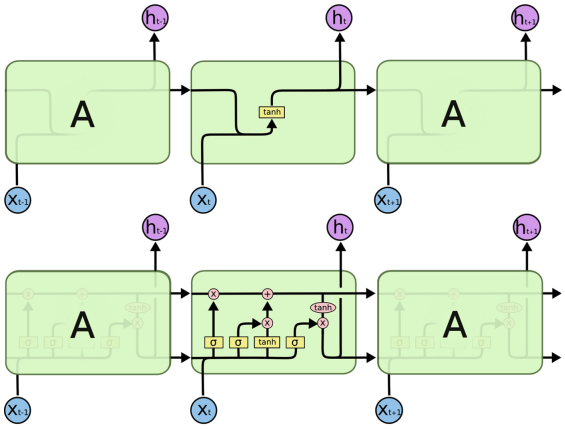


Figure 16: RNNs module structure vs. LSTM module structure

The key point of LSTM is that it includes a "coveyor belt" within the module structure. Information will transfer on this "conveyor belt" and only few linear interaction can prevent the loss of information. There are three different gates collaborate together : "Foget", "Refresh" and "Output".

3.3 GRU

When we train the LSTM module, we find that although the result comes from LSTM is much better than vallina RNN module, but the time cost is also higher. In that case, we notice that LSTM module has a variant module "GRU".

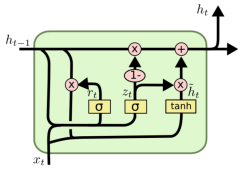


Figure 17: GRU module

It's obvious that inside of the GRU module, there are only two gates named "Refresh" and " Reset" instead of three gates in the LSTM module. The "Refresh" gate decides how much previous information we need to go through and the "Reset" gate decides how much former information the module should discard.

In theoretical, because of the decrease of parameters in the GRU module, the computational efficiency of GRU will higher than LSTM.

3.4 Attention

The traditional Attention Mechanism is also called Soft Attention, which the hidden status of the encoded code is obtained through the deterministic score calculation. Besides, Soft Attention is parameterized, so it can be derived and can be embedded in the model and trained directly. Gradients can be back-propagated to other parts of the model through the Attention Mechanism module.

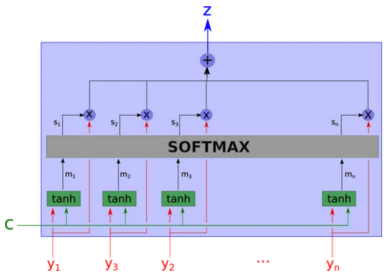


Figure 18: Soft Attention module

4 Implementation

4.1 Embedding

Before we feed the training data into our model, the first thing after cleaning the data is to transfer string type data to vectors. The dictionary we use to convert string is wiki-news-300d-1M.vec which supplied by Kaggle. Wiki-news-300d-1M is a dictionary with one million words(including punctuations and numbers). And each key is one word, each value is a 1-dimension word vector with 300 numbers.

For the words in training data which can be found in wiki-news-dictionary, we replace them with vectors. For those can't be found in wiki-news-dictionary and also escaped from our preprocessing, we initialize them with a random vector which is based on the distribution of the words in text.

```
In [19]: embed_size = 300
embedding_path = "../input/quora-insincere-questions-classification/embeddings/wiki-news-300d-1M/wiki-news-300d-1M.
def get_coefs(word,*arr): return word, np.asarray(arr, dtype='float32')
embedding_index = dict(get_coefs(*o.split(" ")) for o in open(embedding_path, encoding='utf-8', errors='ignore') if
all_embs = np.stack(embedding_index.values())
emb_mean,emb_std = all_embs.mean(), all_embs.std()
word_index = tk.word_index
nb_words = min(max_features, len(word_index))
embedding_matrix = np.random.normal(emb_mean, emb_std, (nb_words + 1, embed_size))
for word, i in word_index.items():
    if i >= max_features: continue
    embedding_vector = embedding_index.get(word)
    if embedding_vector is not None: embedding_matrix[i] = embedding_vector
```

Figure 19: Initialize training dataset with random vector

After all these steps, the data is ready to be fed into our models.

4.2 LSTM

4.2.1 One layer LSTM

```
In [21]: class NeuralNet(nn.Module):
    def __init__(self):
        super(NeuralNet, self).__init__()
        hidden_size = 128
        self.embedding = nn.Embedding(max_features, embed_size)# !20000, 300
        self.embedding.weight = nn.Parameter(torch.tensor(embedding_matrix, dtype=torch.float3
2))#[120001, 300]
        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.linear = nn.Linear(512, 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) #[512, 72, 300]
        h_embedding = torch.squeeze(self.embedding_dropout(torch.unsqueeze(h_embedding, 0)))#[5
12, 72, 300]
        h_lstm, _ = self.lstm(h_embedding)#[512, 72, 256]
        avg_pool = torch.mean(h_lstm, 1)#[512, 256]
        max_pool, _ = torch.max(h_lstm, 1)#[512, 256]
        conc = torch.cat((avg_pool, max_pool), 1)#[512, 512]
        conc = self.relu(self.linear(conc))#[512, 16]
        conc = self.dropout(conc)#[512, 16]
        out = self.out(conc)#[512, 1]
        return out
```

Figure 20: One layer LSTM model

First model is just a simple one-layer LSTM model, after going through the LSTM layer, the output of that layer goes through two separate pooling layers in order to obtain information as much as

possible. Then we concate the two outputs of the pooling layers and put it into an activation layer and then dropout followed by a linear layer.

There's only one hyperparameter we adjust in this project- number of epochs. In order to save time, we first tried 5, 7, 10 epochs to train our raw data without replacing misspelling words. The result is shown in Fig.21.

Model	epoch	Score
one-layer LSTM	5	0.62508
one-layer LSTM	7	0.63011
one-layer LSTM	10	0.61018

Figure 21: Different epochs with one-layer LSTM

Obviously, 7-epoch LSTM has the highest score. And this is the reason that we choose seven epochs to train our following models. With preprocessed data fed into model, we get the result shown below. And the average time training one epoch is at around 75s.

Figure 22: One-layer LSTM(7 epoch)

```
Fold 1
Epoch 1/7       loss=0.1475     val_loss=0.1184         val_f1=0.6291 best_t=0.23       time=7
6.10s
Epoch 2/7       loss=0.1264     val_loss=0.1112         val_f1=0.6498 best_t=0.23       time=7
4.10s
Epoch 3/7       loss=0.1192     val_loss=0.1095         val_f1=0.6627 best_t=0.22       time=7
3.72s
Epoch 4/7       loss=0.1152     val_loss=0.1029         val_f1=0.6686 best_t=0.27       time=7
6.08s
Epoch 5/7       loss=0.1115     val_loss=0.1012         val_f1=0.6712 best_t=0.35       time=7
3.40s
Epoch 6/7       loss=0.1074     val_loss=0.1023         val_f1=0.6703 best_t=0.30       time=7
6.17s
Epoch 7/7       loss=0.1041     val_loss=0.1051         val_f1=0.6731 best_t=0.23       time=7
3.33s
Validation loss:  0.10513302894618638
```

Figure 23: Part training result. Average training time consuming for one epoch is around 75 second

4.2.2 Two and three layers LSTM

We also tried two -layer and three-layer LSTM models with seven epochs. With raw data, the two-layer LSTM got 0.63068 on final result which performs better than one-layer LSTM but the three-layer LSTM got 0.62601 which is worse than the two-layer LSTM model. The reason may be that three layers model is too much for the training data, the model got overfitting on the data.

```
In [24]:
class NeuralNet(nn.Module):
    def __init__(self):
        super(NeuralNet, self).__init__()
        hidden_size = 128
        self.embedding = nn.Embedding(max_features, embed_size)# 120000, 300
        self.embedding.weight = nn.Parameter(torch.tensor(embedding_matrix, dtype=torch.float32))#[120001, 300]
        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.lstm2 = nn.LSTM(256, hidden_size, bidirectional=True, batch_first=True)
        self.linear = nn.Linear(512, 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) #[512, 72, 300]
        h_embedding = torch.squeeze(self.embedding_dropout(torch.unsqueeze(h_embedding, 0)))#[512, 72, 300]
        h_lstm, _ = self.lstm(h_embedding)#[512, 72, 256]
        h_lstm2, _ = self.lstm2(h_lstm)
        avg_pool = torch.mean(h_lstm2, 1)#[512, 256]
        max_pool, _ = torch.max(h_lstm2, 1)#[512, 256]
        conc = torch.cat((avg_pool, max_pool), 1)#[512, 512]
        conc = self.relu(self.linear(conc))#[512, 16]
        conc = self.dropout(conc)#[512, 16]
        out = self.out(conc)#[512, 1]
        return out
```

Figure 24: Two-layer LSTM

After feeding preprocessed data to two-layer LSTM model, we got 0.68421 which is the highest score among all the LSTM models.

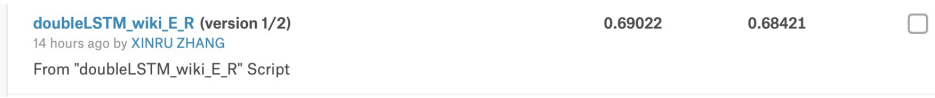


Figure 25: Two-layer LSTM score

4.3 Simple GRU

Simply replacing nn.LSTM with nn.GRU give us one-layer GRU model. Compared to one-layer LSTM, we got a higher score(0.68206) and less training time(at around 44s).

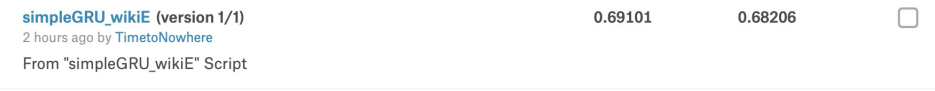


Figure 26: One-layer GRU socre

```
Fold 1
Epoch 1/7        loss=0.1518     time=43.79s
Epoch 2/7        loss=0.1303     time=45.83s
Epoch 3/7        loss=0.1251     time=43.85s
Epoch 4/7        loss=0.1214     time=43.50s
Epoch 5/7        loss=0.1179     time=43.23s
Epoch 6/7        loss=0.1146     time=44.26s
Epoch 7/7        loss=0.1108     time=43.65s
Validation loss:  0.10700189937665172
```

Figure 27: Part training result. Average training time consuming for one epoch is around 44 second

4.4 Soft Attention

The structure of this soft attention model is shown below. The score we got is 0.68506 which is the highest score we have.

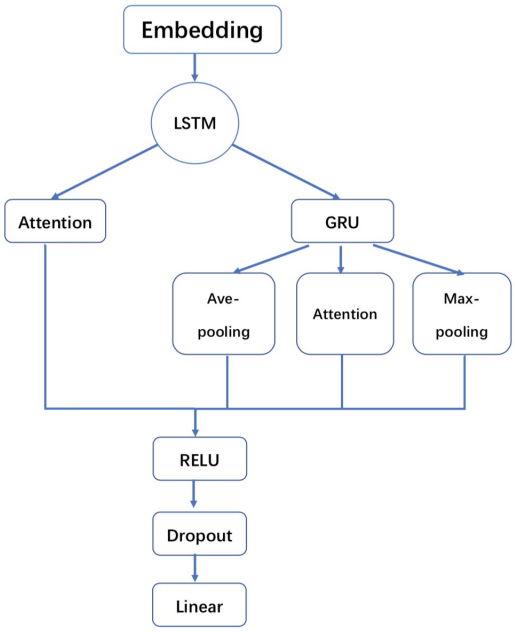


Figure 28: Soft attention mode

In the fig above, we add two weights matrix at the attention step. In training data, not all words' contributions are the same. The intuition to use attention model is to let model pay more attention on some 'useful' words instead of treat every word equally.

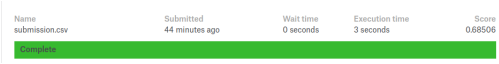


Figure 29: Soft attention model score

Reference

[1] Goodfellow, I. & Bengio, Y. & Courville, A. (2016) Deep Learning. *Modern Practical Deep Networks*, pp. 367–403. Cambridge, MA: MIT Press.

[2] Supervise, ly., (2017) Towards Data Science: *Evolution: from vanilla RNN to LSTM & GRU.* [online]Available at <https://towardsdatascience.com/lecture-evolution-from-vanilla-rnn-to-gru-lstms> (02/12/2019 12:14)

[3] Hoffman, G.(2018). *Introduction to LSTMs with Pytorch.* O'Reilly AI Newsletter, 54, 66-80.

[4] Adrianna, X., (2017) More or Less?:*Talking about the difference with LSTM and GRU.* [online]Available at <https://blog.csdn.net/u012223913/article/details/77724621> (30/11/2019 17:34)

[5] Hao, L., (2018) Talking about Nature Language Processing: *Word Embedding*. [online]Available at <https://blog.csdn.net/L_R_H000/article/details/81320286> (02/12/2019 12:44)

[6] Xiaozhou, Y. & Feifei, T. & Rongzhi, Q. (2016). *Improvement of activation function in recurrent neuron network.* Journal of computer and modernization,12, 14-27.