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 questions from Quora website by building a binary classification model. We separate our whole work into three steps. First, with a deep look into the dataset, we clean the data by removing typos and filling missing data. Second, we train three different models to make our predictions. The approaches we adopt to solve the problem are "GRU", "LSTM" and "Soft Attention". Finally, submitting our predictions to kaggle and get our accuracy. Submissions are evaluated on F1 score between the predicted and the observed targets. Owing to the incorrect labels in datasets, the highest score on the leader board in this competition is about 0.71. Our highest result reaches 0.685 which rank top 25% in the competition. At this time, we only use one word embeddings among applied four embeddings, hoping by combining all the embeddings together can we get a better result in the future.

1 Introduction

Quora is a website for people to communicate with others. But sometimes inappropriate questions and 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 and help Quora maintain their policy with "Be Nice, Be Respectful", we need to find more up-gradable ways to discover these ambiguous and confusing questions.

The training dataset consists of three parts: id(numbers), question text(string), label(0 or 1) whereas test data lacking labels. By filling missing data, removing or replacing typos in an appropriate way, 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 take four steps to clean the original data:

1) Obtain a vocabulary by loading words from word embedding(wiki-news-300d-1M.vec).
2) Get and analyze words in training data which are uncovered by "wiki" vocabulary.
3) Discard the punctuation, spaces and replace the misspell words into correct spell words.
4) Choose appropriate words in "wiki" vocabulary and replace the “cannot find” words with them.

Model building 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 approaches are 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 has the ability to "remember" some information in the past and keep them to future calculations, not like general RNN model will lose those information instead. 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 GRU model has fewer parameters than LSTM model, so

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

the training speed is faster. Furthermore, we adopt and adjust attention model to make predictions because we not only want our model to "remember" some past info but also focus on some potential words which may play a decisive role in sentences.

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.

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. In figure 1, 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. "qid" contains ids for every sentences which is useless for us. “Question_text” column is composed of sentences that we need to analysis and feed into our models. “Target” column has binary values. “1” means that this sentence is identified as insincere.

```
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]:
```

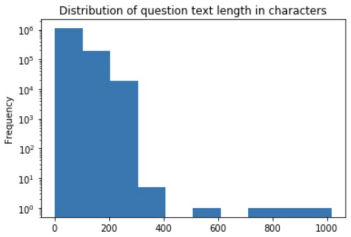
	qid	question_text	target
0	00002165364db923c7e6	How did Quebec nationalists see their province...	0
1	000032939017120e6e44	Do you have an adopted dog, how would you enco...	0

Figure 1: Information of training dataset

By thoroughly looking at questions, we find two problems, one is the lengths of questions are not same, the other one is typos and new words may not appear in word embedding.

We generate two distributions of question text length in words and characters shown as Fig 2 below. The shapes of these two distributions are very similar. In order to decide how long should we set the model’s input, we then calculate the average length and max length in Fig 3.

```
In [21]: train['question_text'].apply(lambda x: len(x)).plot(kind='hist')
         plt.yscale('log')
         plt.title('Distribution of question text length in characters')
Out[21]: Text(0.5, 1.0, 'Distribution of question text length in characters')
```




```
In [22]: train['question_text'].apply(lambda x: len(x.split())).plot(kind='hist')
         plt.yscale('log')
         plt.title('Distribution of question text length in words')
Out[22]: Text(0.5, 1.0, 'Distribution of question text length in words')
```

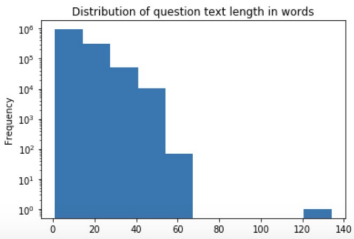


Figure 2: Distribution of question text length in characters

```
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 and max length of questions in training dataset and testing dataset

In the result above, the average character length in train and test are all 71, so we decide to set the maximum length of words in each sentence to 72.

For the purpose of dealing with typos and new words, the first thing we need to do is getting vocabularies for dataset and word embedding. To get the vocabulary for dataset, we build a function called build_vocab to collect not only words but also frequencies (Fig 4). To get the vocabulary for word embedding, we simply load from “wiki-news-300d-1M.vec” (Fig 5).

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 4: Collect words and words frequency from training dataset

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

By using the check_coverage function we build, we get the uncovered sorted vocabulary called "oov" (Fig 6). Fig 7 displays top 10 uncovered words, most of them are abbreviations and words with punctuations. After manually create a misspell dictionary containing abbreviations and common misspell words with their correct format (Fig 8), we simply remove other punctuations considering punctuations may not be very useful for classification (Fig 9). What's more, we also pick 1000 words from "oov" and compare each word among sentences in the training dataset to get the according index, label and frequency. (Fig.10)

```
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

```
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 of covered and uncovered words compare to the "wiki" dataset

```
In [20]: print(len(oov))
         355920

In [18]: oov[:10]
Out[18]: [('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: Display top 10 frequency of uncovered words

```
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', 'narcissist': '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', 'Etherlum': '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 8: Manually create a misspell dictionary

```
In [17]: # transfer this to kaggle code
         def clean_text(x):
             x = str(x)
             for punct in '?!.,"#$%\'()*+-/:;<=>@[\\]^{|}~' + '“”’':
                 x = x.replace(punct, ' ')
             for punct in '.':
                 x = x.replace(punct, ' ')
             return x

In [18]: # transfer this to kaggle code
         train["question_text"] = train["question_text"].progress_apply(lambda x: clean_text(x))
         # leave codes below, don't transfer these
         sentences = train["question_text"].apply(lambda x: x.split())
         vocab = build_vocab(sentences)

         100%|██████████| 1306122/1306122 [00:06<00:00, 211132.97it/s]
         100%|██████████| 1306122/1306122 [00:03<00:00, 382303.74it/s]
```

Figure 9: Split and replace punctuations

```
In [19]: word = {}
         for i in oov[:1000]:
             word[i[0]] = []
             num = 0
             for j in range(len(train['question_text'])):
                 cnt = train['question_text'][j].count(i[0])
                 num += cnt
                 if cnt > 0:
                     word[i[0]].append([j,train['target'][j],cnt])
                 if num == i[1]:
                     break
```

Figure 10: Generate mistake embedding dictionary

When we ruled out the possibility of mistyping, the number of cannot-find-word remains high which may still affect the accuracy of our model. To eliminate this effect as much as possible, we come up with a method which replaces cannot-find-word with other words based on labels. We first sort the cannot-find-word with frequencies, and then choose the top 1000 separately and count how many times they appear in good_sentence(label 0) and bad_sentence(label 1) in train dataset as shown in Fig 11. And calculate the ratio=bad_times/total times (shown in Fig 12).

Figure 11: Replace cannot-find-word with other words base on labels

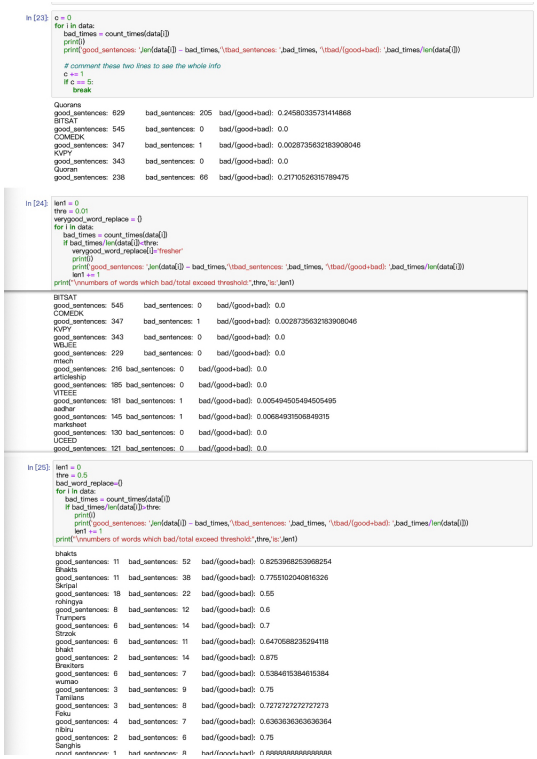


Figure 12: Calculate bad ratio of sentences within the total training dataset

At the same time, we randomly sample 1000 sentences respectively from good_sentence and bad_sentence. Collect words appear in these 2000 sentences and count times they appear in good_sentence(label 0) and bad_sentence(label 1) in train dataset. Then calculate the ratio of each word, shown in Fig 13. By setting the threshold of ratio, we can choose appropriate word with similar ratio as canno-find-word, we can replace cannot-find-word with them. For example, in Fig 13 the ratio of 'bhakts' is about 0.8254, and the word 'Marathis' has similar ratio value. As to this situation, it is theoretically feasible to replace 'bhakts' with 'Marathis'

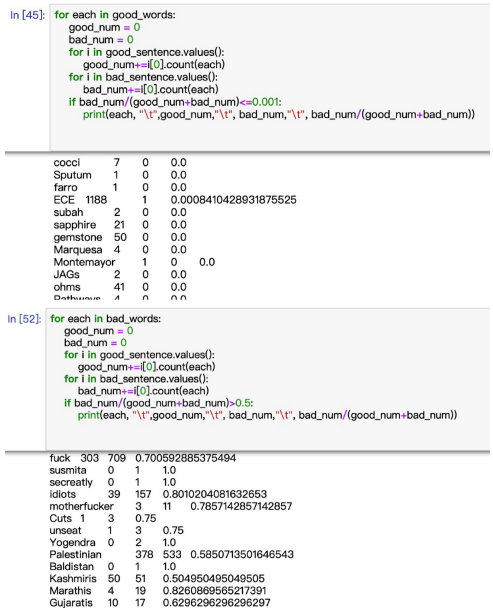


Figure 13: Calculate bad ratio of sentences within the total training dataset

After all the above process are done, we successfully reduced the number of cannot-find-word from 355920 to 79883.

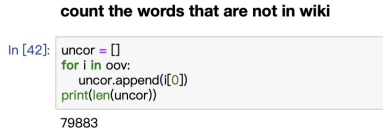


Figure 14: The list length changes from 355920 to 79883

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 occurrence of RNN solve this problem.

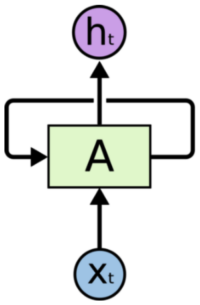


Figure 15: Single recurrent neural network model

The recurrent neural networks model is in Fig.15 above. During each time period, each node receives information from the previous node and the process can be represented by a feedback cycle. Fig.16 shows the details of this "feedback" cycle.

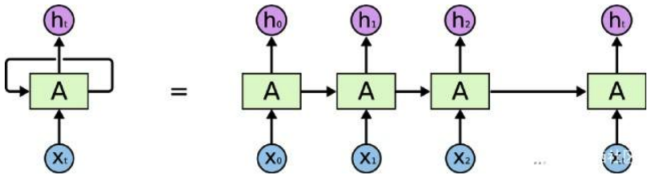


Figure 16: How recurrent neural network work

In each time process, we pick an input " xi " and output of previous node "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.

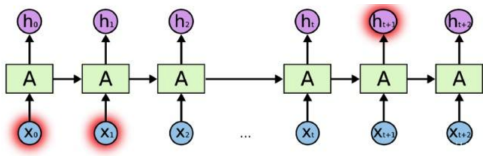


Figure 17: RNN cannot get information with long period before

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.

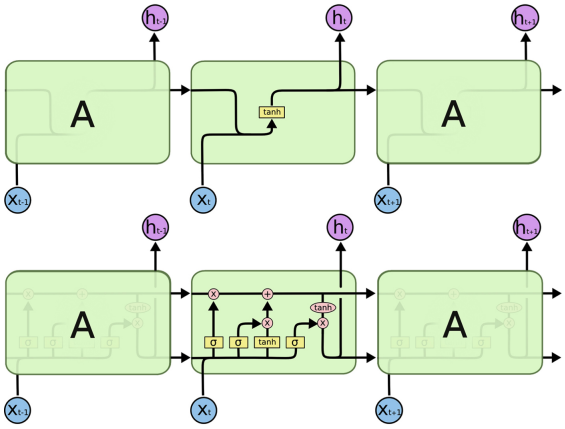


Figure 18: RNNs module structure vs. LSTM module structure

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 higher. To reduce the training time, we use GRU model to replace LSTM model.

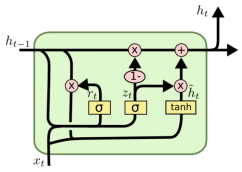


Figure 19: GRU module

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.

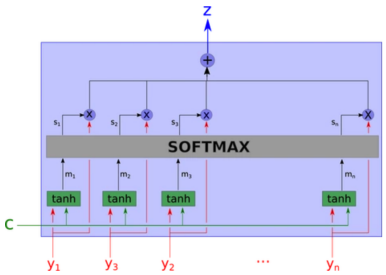


Figure 20: Attention module

Our model is inspired by this soft attention model. With LSTM inside the model, we put weights on the LSTM output and concatenate them to get final result.

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 21: 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

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 concatenate 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.23.

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

Figure 23: Different epochs with 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)# 120000, 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 22: One layer LSTM model

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.

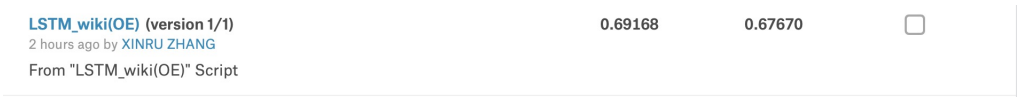


Figure 24: One-layer LSTM(7 epoch)

```
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 25: 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 26: 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.

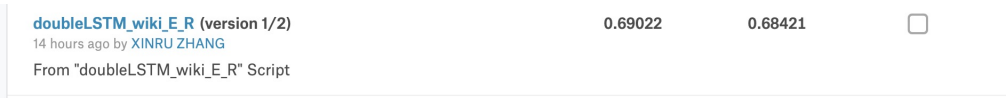


Figure 27: Two-layer LSTM score

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).

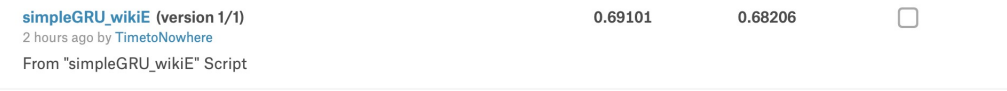


Figure 28: One-layer GRU socre

```
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 29: Part training result. Average training time consuming for one epoch is around 44 second

4.4 Attention

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

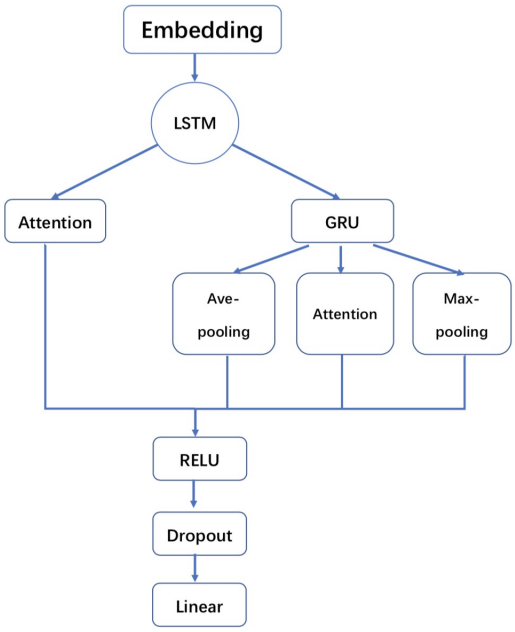


Figure 30: Attention mode

In the Fig.31, 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 31: 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.

[7] Zafarali, A.(Jun 29, 2017). How to Visualize Your Recurrent Neural Network with Attention in Keras. *Medium website*