Using Keras and LSTM to Implement Sentence Comparison

B02201041 何庭昀 B02901136 綦家志

Abstract

In Kaggle data analysis competition, there is a topic we are interested in, Quora Question pairs (https://www.kaggle.com/c/quora-question-pairs). The main purpose of this problem is to predict whether two sentences have the same meaning, that is, to implement sentence comparison. We learned a lot about RNN and LSTM in the class, so we want to use this technic to solve our problem. And then, we want to compare effect of different combination of LSTM and DNN.

Introduction

Quora is a place to gain and share knowledge about anything. It's a platform to ask questions and connect with people who contribute unique insights and quality answers. Over 100 million people visit Quora every month, so it's no surprise that many people ask similarly worded questions. Multiple questions with the same intent can cause seekers to spend more time finding the best answer to their question, and make writers feel they need to answer multiple versions of the same question.

In this problem, we are challenged to tackle this natural language processing problem by applying RNN and LSTM to classify whether question pairs are duplicates or not. Moreover, we want to compare effect of different number of CNN layers and to find out if bidirection LSTM is better than single LSTM layer. Doing so will make it easier to find high quality answers to questions resulting in an improved experience for Quora writers, seekers, and readers.

Data description

The goal of this problem is to predict which of the provided pairs of questions contain two questions with the same meaning. The training data similarity labels have been supplied by human, which are inherently subjective, as the true meaning of sentences can never be known with certainty. Human labeling is also a 'noisy' process, and reasonable people will disagree. As a result, the similarity labels on this dataset should be taken to be 'informed' but not 100% accurate, and may include incorrect labeling. Data given in this problem is (1.) a training data with 404290 pairs of sentences and (2.) a testing data with 2345796 pairs of sentences. (https://www.kaggle.com/c/quora-question-pairs/data)

For example, in training data, we have

1. "What is the step by step guide to invest in share market in India?"
   "What is the step by step guide to invest in share market?"
   "0"
2. "Astrology: I am a Capricorn Sun Cap moon and cap rising... what does that say about me?"
   "I'm a triple Capricorn (Sun, Moon and ascendant in Capricorn) What does this say about me?"
   "1"

The labels "0" means previous two sentences are not duplicates, and "1" means previous two sentences are duplicates.

In testing data, we have

1. “How does the Surface Pro himself 4 compare with iPad Pro?”

   “Why did Microsoft choose core m3 and not core i3 home Surface Pro 4?”
2. “Should I have a hair transplant at age 24? How much would it cost?”

   “How much cost does hair transplant require?”

Our final purpose is to predict whether question pairs are duplicates or not.

Method

We decide to use google library, word2vec, and word2vec model, “GoogleNews-vectors-negative300.bin” (https://github.com/mmihaltz/word2vec-GoogleNews-vectors), to transform every word in sentences to a distributed represent vector, which means we can evaluate correlation of two words by their vectors value. For example, “telephone” and “phone” will be very close in word2vec form, and “king” + “women” - “men” will be very close to “queen”. If a word in sentence cannot be found in google word2vec model, it will be corrected by Peter Norvig’s word corrector. And then, we train our model by these sentence. After finishing the training, we predict test data by our model, and evaluate similarity of each sentence pair.

Code and model structure

Our code is modified by listdo’s code on (https://www.kaggle.com/lystdo/lstm -with-word2vec-embeddings). In the first part we import some necessary library. Noted that we also import some Keras model such as Dense and LSTM.

```
########################################
## import packages
########################################
import os
import re
```

```python
import csv
import codecs
import numpy as np
import pandas as pd

from nltk.corpus import stopwords
from nltk.stem import SnowballStemmer
from string import punctuation
from gensim.models import KeyedVectors, Word2Vec
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.layers import Dense, Input, LSTM, Embedding, Dropout, Activation
from keras.layers.merge import concatenate
from keras.models import Model
from keras.layers.normalization import BatchNormalization
from keras.callbacks import EarlyStopping, ModelCheckpoint
import re, collections
import sys

```

And then we define Peter Norvig's word corrector. The way that the corrector correct the word is to split and combine the word in some way (function `edits1`) by one or two times (function `known_edits2`), delete those word which are not exist in dictionary data "big.txt" (function `known`)(https://github.com/dscape/spell/blob/master /test/resources/big.txt), and then calculate which word in these candidates has the most frequency in dictionary data "big.txt" (function `correct`). If you want to learn more, please check the website of Peter Norvig's word corrector (http://norvig.com/spell-correct.html).

```python
########################################
## word corrector define
########################################
def words(text):
    return re.findall('[a-z]+', text.lower())

```

```python
def train(features):
    model = collections.defaultdict(lambda: 1)
    for f in features:
        model[f] += 1
    return model

NWORDS = train(words(open('big.txt').read()))
alphabet = 'abcdefghijklmnopqrstuvwxyz'

def edits1(word):
    s = [(word[:i], word[i:]) for i in range(len(word) + 1)]
    deletes    = [a + b[1:] for a, b in s if b]
    transposes = [a + b[1] + b[0] + b[2:] for a, b in s if len(b)>1]
    replaces   = [a + c + b[1:] for a, b in s for c in alphabet if b]
    inserts    = [a + c + b     for a, b in s for c in alphabet]
    return set(deletes + transposes + replaces + inserts)

def known_edits2(word):
    return set(e2 for e1 in edits1(word) for e2 in edits1(e1) if e2 in NWORDS)

def known(words):
    return set(w for w in words if w in NWORDS)

def correct(word):
    candidates = known([word]) or known(edits1(word)) or known_edits2(word) or [word]
    return max(candidates, key=NWORDS.get)

```

we set directories of google word2vec model “GoogleNews-vectors-negative300.bin”, training data, and test data. And then we set some parameters. In this part, we set activation function as “relu”.

```python
########################################
## set directories and parameters
########################################
BASE_DIR = './'
EMBEDDING_FILE = BASE_DIR + 'GoogleNews-vectors-negative300.bin'
```

```
TRAIN_DATA_FILE = BASE_DIR + 'train.csv'
TEST_DATA_FILE = BASE_DIR + 'test.csv'
MAX_SEQUENCE_LENGTH = 30
MAX_NB_WORDS = 200000
EMBEDDING_DIM = 200
VALIDATION_SPLIT = 0.1

num_lstm = np.random.randint(200, 250) #175~275
num_dense = np.random.randint(100, 150) #100~150
rate_drop_lstm = 0.15 + np.random.rand() * 0.25
rate_drop_dense = 0.15 + np.random.rand() * 0.25

act = 'relu'
re_weight = True # whether to re-weight classes to fit the 17.5% share in test set

STAMP = 'lstm_%d_%d_%.2f_%.2f'%(num_lstm, num_dense, rate_drop_lstm,
        rate_drop_dense)

```

In this part, we load word2vec model “GoogleNews-vectors- negative300.bin”.

```
########################################
## index word vectors
########################################
print('Indexing word vectors')

word2vec = KeyedVectors.load_word2vec_format(EMBEDDING_FILE, binary=True)
print('Found %s word vectors of word2vec' % len(word2vec.vocab))

```

We use text_to_wordlist function (https://www.kaggle.com/currie32/quora-question-pairs/the-importance-of-cleaning-text), to clean the text. That is, delete some unnecessary characters. And then we make the sentences to the form that can be sent to Keras model.

```
########################################
## text_to_wordlist
```

```
########################################
print('Processing text dataset')

# The function "text_to_wordlist" is from
# https://www.kaggle.com/currie32/quora-question-pairs/the-importance-of-cleaning-text

def text_to_wordlist(text, remove_stopwords=False, stem_words=False):
    # Clean the text, with the option to remove stopwords and to stem words.
    # Convert words to lower case and split them
    text = text.lower().split()

    # Optionally, remove stop words
    if remove_stopwords:
        stops = set(stopwords.words("english"))
        text = [w for w in text if not w in stops]

    text = " ".join(text)

    # Clean the text
    text = re.sub(r"[^A-Za-z0-9^,!.\/'+-=]", " ", text)
    text = re.sub(r"what's", "what is ", text)
    text = re.sub(r"\'s", " ", text)
    text = re.sub(r"\'ve", " have ", text)
    text = re.sub(r"can't", "cannot ", text)
    text = re.sub(r"n't", " not ", text)
    text = re.sub(r"i'm", "i am ", text)
    text = re.sub(r"\'re", " are ", text)
    text = re.sub(r"\'d", " would ", text)
    text = re.sub(r"\'ll", " will ", text)
    text = re.sub(r",", " ", text)
    text = re.sub(r"\.", " ", text)
    text = re.sub(r"!", " ! ", text)
    text = re.sub(r"\/", " ", text)
    text = re.sub(r"\^", " ^ ", text)
    text = re.sub(r"\+", " + ", text)
    text = re.sub(r"\-", " - ", text)
    text = re.sub(r"\=", " = ", text)
    text = re.sub(r"'", " ", text)
    text = re.sub(r"(\d+)(k)", r"\g<1>000", text)
    text = re.sub(r":", " : ", text)
    text = re.sub(r" e g ", " eg ", text)
    text = re.sub(r" b g ", " bg ", text)
    text = re.sub(r" u s ", " american ", text)
    text = re.sub(r"\0s", "0", text)
    text = re.sub(r" 9 11 ", "911", text)
    text = re.sub(r"e - mail", "email", text)
    text = re.sub(r"j k", "jk", text)
    text = re.sub(r"\s{2,}", " ", text)

    # Optionally, shorten words to their stems
    if stem_words:
        text = text.split()
        stemmer = SnowballStemmer('english')
        stemmed_words = [stemmer.stem(word) for word in text]
        text = " ".join(stemmed_words)

    # Return a list of words
    return(text)

########################################
## text preprocessing
########################################
texts_1 = []
texts_2 = []
labels = []
with codecs.open(TRAIN_DATA_FILE, encoding='utf-8') as f:
    reader = csv.reader(f, delimiter=',')
    header = next(reader)
    for values in reader:
        texts_1.append(text_to_wordlist(values[3]))
        texts_2.append(text_to_wordlist(values[4]))
        labels.append(int(values[5]))
print('Found %s texts in train.csv' % len(texts_1))

test_texts_1 = []
test_texts_2 = []
```

```python
test_ids = []
with codecs.open(TEST_DATA_FILE, encoding='utf-8') as f:
    reader = csv.reader(f, delimiter=',')
    header = next(reader)
    for values in reader:
        test_texts_1.append(text_to_wordlist(values[1]))
        test_texts_2.append(text_to_wordlist(values[2]))
        test_ids.append(values[0])
print('Found %s texts in test.csv' % len(test_texts_1))

tokenizer = Tokenizer(num_words=MAX_NB_WORDS)
tokenizer.fit_on_texts(texts_1 + texts_2 + test_texts_1 + test_texts_2)

sequences_1 = tokenizer.texts_to_sequences(texts_1)
sequences_2 = tokenizer.texts_to_sequences(texts_2)
test_sequences_1 = tokenizer.texts_to_sequences(test_texts_1)
test_sequences_2 = tokenizer.texts_to_sequences(test_texts_2)

word_index = tokenizer.word_index
print('Found %s unique tokens' % len(word_index))

data_1 = pad_sequences(sequences_1, maxlen=MAX_SEQUENCE_LENGTH)
data_2 = pad_sequences(sequences_2, maxlen=MAX_SEQUENCE_LENGTH)
labels = np.array(labels)
print('Shape of data tensor:', data_1.shape)
print('Shape of label tensor:', labels.shape)

test_data_1 = pad_sequences(test_sequences_1, maxlen=MAX_SEQUENCE_LENGTH)
test_data_2 = pad_sequences(test_sequences_2, maxlen=MAX_SEQUENCE_LENGTH)
test_ids = np.array(test_ids)

```

In this part, we transform the sentences to word2vec vectors. If words cannot be found in word2vec model, we will correct it by Peter Norvig's word corrector, and then check the new words are in word2vec model again. And then split some valuation data from training data.

```python
##########################################
## prepare embeddings
##########################################
print('Preparing embedding matrix')

nb_words = min(MAX_NB_WORDS, len(word_index))+1

embedding_matrix = np.zeros((nb_words, EMBEDDING_DIM))
success_count=0
for word, i in word_index.items():
    if word in word2vec.vocab:
        embedding_matrix[i] = word2vec.word_vec(word)
    else:
        tempword=correct(word)
        if tempword in word2vec.vocab:
            success_count+=1
            print ('success:'+word+' to '+ tempword)
            embedding_matrix[i] = word2vec.word_vec(tempword)

print('Null word embeddings: %d' % np.sum(np.sum(embedding_matrix, axis=1) == 0))
print('successfully correct ' + str(success_count) + ' token')

##########################################
## sample train/validation data
##########################################
perm = np.random.permutation(len(data_1))
idx_train = perm[:int(len(data_1)*(1-VALIDATION_SPLIT))]
idx_val = perm[int(len(data_1)*(1-VALIDATION_SPLIT)):]

data_1_train = np.vstack((data_1[idx_train], data_2[idx_train]))
data_2_train = np.vstack((data_2[idx_train], data_1[idx_train]))
labels_train = np.concatenate((labels[idx_train], labels[idx_train]))

data_1_val = np.vstack((data_1[idx_val], data_2[idx_val]))
data_2_val = np.vstack((data_2[idx_val], data_1[idx_val]))
labels_val = np.concatenate((labels[idx_val], labels[idx_val]))

weight_val = np.ones(len(labels_val))
```

```python
if re_weight:
    weight_val *= 0.472001959
    weight_val[labels_val==0] = 1.309028344

```

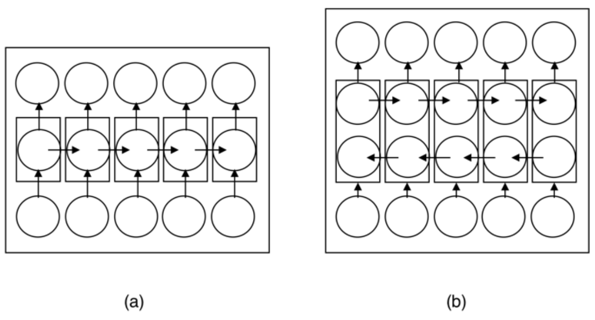
In this part, we construct our model. First, we take two sentences as two input layers, and connect to two embedding layers, and then connect to two LSTM layers, and take two LSTM output to DNN, and then a one-node output layer.

There are two feature we want to compare

1. LSTM vs. Bidirectional LSTM
2. 2 vs. 3 vs. 4 hidden layers of DNN

So we want to compare effect of 6 models

1. LSTM+2 hidden layers of DNN (50+20)
2. LSTM+3 hidden layers of DNN (100+50+20)
3. LSTM+4 hidden layers of DNN (150+100+50+20)
4. Bidirectional LSTM+2 hidden layers of DNN (50+20)
5. Bidirectional LSTM+3 hidden layers of DNN (100+50+20)
6. Bidirectional LSTM+4 hidden layers of DNN (150+100+50+20)



Structure overview
(a) unidirectional RNN
(b) bidirectional RNN

from Wikipedia: Bidirectional recurrent neural networks

After defining models, we train it, and then make prediction. And then submit the result to Kaggle to get accuracy.

```
########################################
## define the model structure
## 1. LSTM vs. Bidirectional LSTM
## 2. 2 vs. 3 vs. 4 hidden layers of DNN
########################################
embedding_layer = Embedding(nb_words,
        EMBEDDING_DIM,
        weights=[embedding_matrix],
        input_length=MAX_SEQUENCE_LENGTH,
        trainable=False)

lstm_layer = LSTM(num_lstm, dropout=rate_drop_lstm, recurrent_dropout=rate_drop_lstm)
'''''
lstm_layer = Bidirectional(LSTM(num_lstm, dropout=rate_drop_lstm, recurrent_dropout=rat
e_drop_lstm))
'''

sequence_1_input = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype='int32')
embedded_sequences_1 = embedding_layer(sequence_1_input)
x1 = lstm_layer(embedded_sequences_1)

sequence_2_input = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype='int32')
embedded_sequences_2 = embedding_layer(sequence_2_input)
y1 = lstm_layer(embedded_sequences_2)

merged = concatenate([x1, y1])
merged = Dropout(rate_drop_dense)(merged)
merged = BatchNormalization()(merged)

'''''
merged = Dense(150, activation=act)(merged)
merged = Dropout(rate_drop_dense)(merged)
```

```python
merged = BatchNormalization()(merged)
merged = Dense(100, activation=act)(merged)
merged = Dropout(rate_drop_dense)(merged)
merged = BatchNormalization()(merged)
'''
merged = Dense(50, activation=act)(merged)
merged = Dropout(rate_drop_dense)(merged)
merged = BatchNormalization()(merged)
merged = Dense(20, activation=act)(merged)
merged = Dropout(rate_drop_dense)(merged)
merged = BatchNormalization()(merged)

preds = Dense(1, activation='sigmoid')(merged)
########################################
## add class weight
########################################
if re_weight:
    class_weight = {0: 1.309028344, 1: 0.472001959}
else:
    class_weight = None

########################################
## train the model
########################################
model = Model(inputs=[sequence_1_input, sequence_2_input],
        outputs=preds)
model.compile(loss='binary_crossentropy',
        optimizer='nadam',
        metrics=['acc'])
model.summary()
print(STAMP)

early_stopping =EarlyStopping(monitor='val_loss', patience=3)
bst_model_path = STAMP + '.h5'
model_checkpoint = ModelCheckpoint(bst_model_path, save_best_only=True, save_weights_only=True)

hist = model.fit([data_1_train, data_2_train], labels_train,
        validation_data=([data_1_val, data_2_val], labels_val, weight_val),
        epochs=50, batch_size=1024, shuffle=True,
        class_weight=class_weight, callbacks=[early_stopping, model_checkpoint])

model.load_weights(bst_model_path)
bst_val_score = min(hist.history['val_loss'])

########################################
## make the submission
########################################
print('Start making the submission before fine-tuning')

preds = model.predict([test_data_1, test_data_2], batch_size=1024, verbose=1)
preds += model.predict([test_data_2, test_data_1], batch_size=1024, verbose=1)
preds /= 2

submission = pd.DataFrame({'test_id':test_ids, 'is_duplicate':preds.ravel()})
submission.to_csv('%.4f_'%(bst_val_score)+STAMP+'.csv', index=False)

```

Comparison and result

We construct 6 models, and here are the results.

model	Log_loss
LSTM+2 hidden layers of DNN	0.31136
LSTM+3 hidden layers of DNN	0.30114
LSTM+4 hidden layers of DNN	0.30531
Bidirectional LSTM+2 hidden layers of DNN	0.31074
Bidirectional LSTM+3 hidden layers of DNN	0.30629
Bidirectional LSTM+4 hidden layers of DNN	0.31404

As you can see, in both LSTM and bidirectional LSTM models, 3 hidden layers of DNN has the best accuracy. Maybe it is the result of deeper DNN is harder to train, or even overfitting.

On the other hand, the accuracy of single layer LSTM model is higher than bidirectional LSTM model. It is different from the results of most of the theses. We think this is because we haven’t found the right hyperparameters of bidirectional LSTM. We believe that if we have more time, we can find better hyperparameters of all models, and maybe our result will match results of theses.

Most important, all our models have loss around 0.3, and in Kaggle data analysis competition, we have rank of 963 out of 3307. There are some reasons why our accuracy is so low.

First, we haven’t found the right hyperparameters. We didn’t have enough time to try others hyperparameters because training deep learning models cost a lot of times (6 to 8 hours per time).

Second, using deep learning is not the best idea. In the e-book “neural network and deep learning”, we learned that deep learning can do everything, however, it is always hard to train. After reading a lot of articles in Kaggle discuss board, we found that most people have higher rank are not using deep learning, they use “Xgboost”, a model that combine many easier model (and therefore easier to train) to get better prediction, instead. Furthermore, I have heard that in Kaggle, half of the winner use DNN, and the others use Xgboost. I believe that both of them are strong ML model, and maybe some problem suit DNN, and some suit Xgboost.

Third, training data is not big enough. We only have 404290 sentence pairs in training data, but 2345797 testing data (over 5 times), and it seems like we should do more to handle such situation.

Conclusion

In this project, we used deep learning to implement sentences comparison, and we compare results of different structure of models. We know that deep learning is a good way to solve use, since it is not so hard to write DNN code in Keras, and can easily get a not-so-bad result. However, if we need to be more accurate, we have to spend a lot of time to tune our model, do more preprocessing, or even try different model to choose the best.

Reference

[1] Jonas Mueller and Aditya Thyagarajan, “Siamese Recurrent Architectures for Learning Sentence Similarity,” Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)