Open in app Get started

Published in Towards Data Science

Vikas Bhandary Follow

Sep 15, 2019 · 11 min read · Listen

Save

Photo by fabio on Unsplash

# Introduction to Natural Language Processing for Noobs

## A general introduction to NLP and its basic pipeline using Kaggle Competition

This post is my attempt to give an overview of basic concepts which might help noobs. For better understanding, I will be following a completed Kaggle competition Quora Insincere Questions Classification. Here, we are provided 1.31 million questions with labels and

So let's start with basics first.

## Introduction

NLP is a branch of computer science which deals with the interaction between humans and machines in a natural language. It is the intersection of both linguistics and computer science, so it enables machines to understand and reply to humans queries in a natural language. The major problem with NLP is that human languages are ambiguous. Humans are very intelligent to understand the context and the meaning of words but for computers, this problem is on a whole another level since computers understand neither concepts nor context. For computers to understand concepts, they will need a basic understanding of the world, language syntax, and semantics.

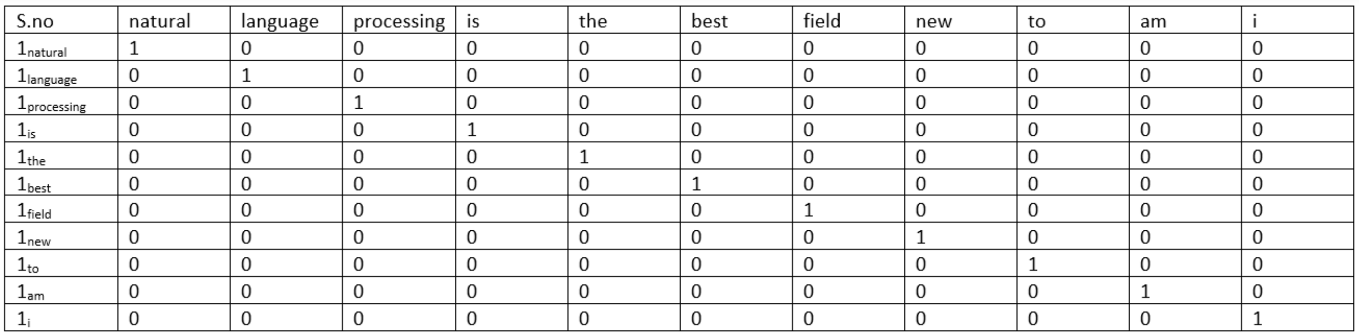
## Text representation

We can understand and read a piece of text but for computers, every text is a sequence of numbers which don't represent any concept. One simple example, the letter "A" possesses a special meaning in the English language, it is considered as the first letter of all the alphabets. But a computer sees it as 65 (as 65 is ASCII code for letter 'A'). ASCII is the traditional encoding system, which is based on the English characters. Collection of such characters is generally referred to as *token* in NLP.

> A token is an instance of a sequence of characters in some particular text that is grouped together to make some sense in natural language.

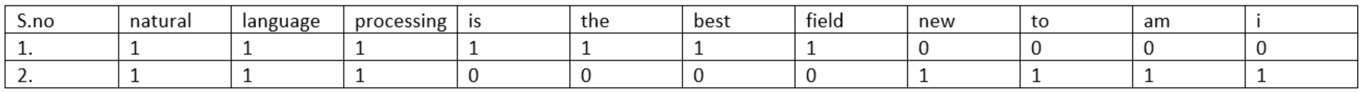
The easiest way to represent any text in an NLP pipeline is by one-hot encoded vector representation. If a sentence contains a certain word then the corresponding entry in the vector is represented as "1" otherwise it's "0". For example, let's consider the following two sentences:

1. "natural language processing is the best field"



| S.no | natural | language | processing | is | the | best | field | new | to | am | i |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $1_{natural}$ | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $1_{language}$ | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $1_{processing}$ | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $1_{is}$ | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $1_{the}$ | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| $1_{best}$ | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| $1_{field}$ | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| $1_{new}$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| $1_{to}$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| $1_{am}$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| $1_{i}$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

**Table 1.** One hot encoding of all words in sentence 1 and 2



| S.no | natural | language | processing | is | the | best | field | new | to | am | i |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2. | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

**Table 2.** Collapsed One hot encoded vector for sentence 1 and 2

One hot encoding is a representation of categorical variables as binary vectors. One hot encoding of words is used to encode the text, where every word is represented with zero except the current word. If we have a corpus of one thousand unique words, then each word representation would require a vector of size 1000. Despite the dependence of size on vocabulary, one-hot encoding representation is being used in productions even today.

Table 1 shows the one-hot encoding of all the words in sentences 1 and 2. Representations shown in Table 2 are called collapsed or "binary" representation.

Modern NLP systems have a dictionary of words, where each word is given an integer representation. So the phrase "natural language processing is the best field!" could be represented as "2 13 6 34 12 22 90" (ignoring punctuations and treating every word as lowercase). This representation is more efficient in terms of memory usage and it also retains the language semantics.

## Basic NLP pipeline

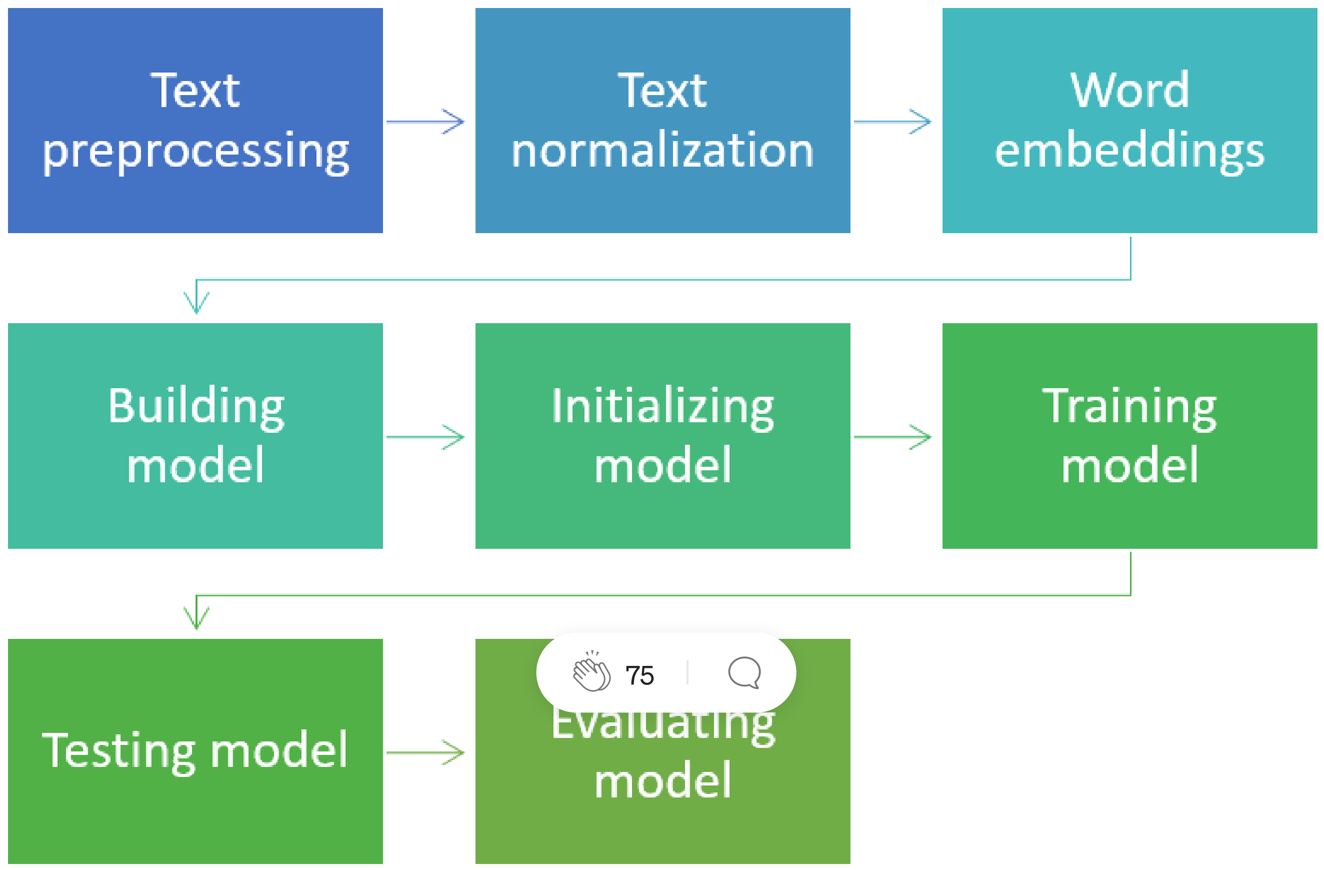


**Figure 1:** Basic NLP pipeline

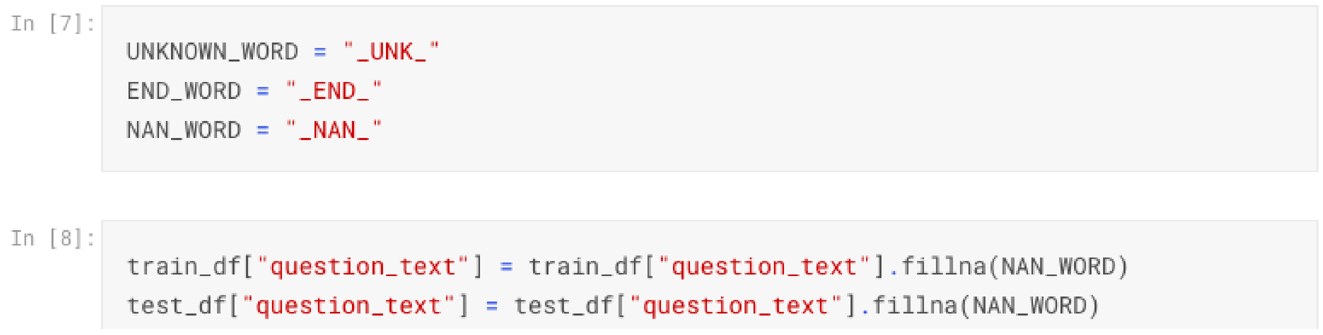
There are so many tasks in NLP for which we can use the pipeline shown above. Most Kaggle competitions have cleaned data, which doesn't happen in real life, as these datasets are collected and pre-processed by competition coordinators. In a real-world scenario, **we must always assume that the data needs some preprocessing**. After preprocessing, we need to break up text in *tokens* and *sentences*. After breaking up the text, we use pre-trained embeddings to initialize our model.

For simplicity of this post, I've skipped text analysis from our pipeline. Textual analysis is a very important part of any NLP pipeline. Based on the insights of the analysis, pipeline processes can be modified to improve the performance of the application.

## Text Preprocessing

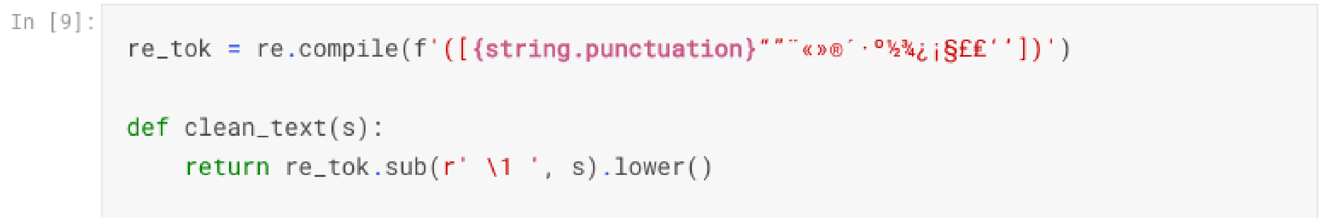
Textual data can be very messy. So most of the times we will need to preprocess the text.

## Code sample:



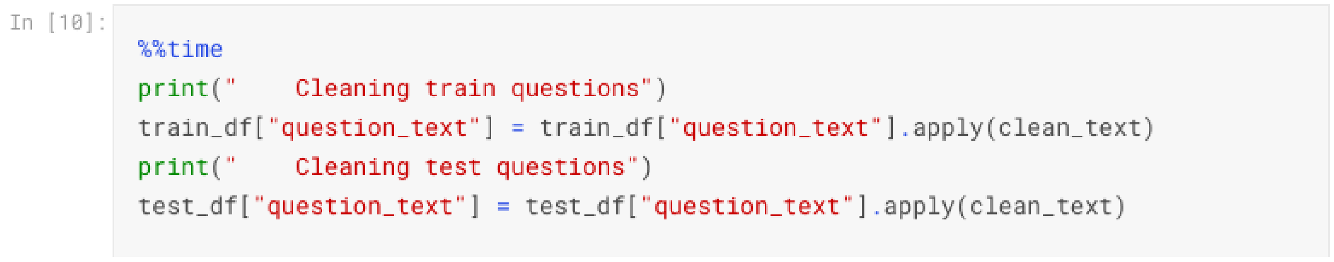
```
In [7]:
UNKNOWN_WORD = "_UNK_"
END_WORD = "_END_"
NAN_WORD = "_NAN_"

In [8]:
train_df["question_text"] = train_df["question_text"].fillna(NAN_WORD)
test_df["question_text"] = test_df["question_text"].fillna(NAN_WORD)
```

**Code sample 1:** Replacing empty records



```
In [9]:
re_tok = re.compile(f'([{string.punctuation}“”¨«»®´·º½¾¿¡§£₤‘’])')

def clean_text(s):
    return re_tok.sub(r' \1 ', s).lower()
```

**Code sample 2:** Declaring function to clear text



```
In [10]:
%%time
print("    Cleaning train questions")
train_df["question_text"] = train_df["question_text"].apply(clean_text)
print("    Cleaning test questions")
test_df["question_text"] = test_df["question_text"].apply(clean_text)
```

**Code sample 3**: Cleaning question text

In the above code samples *train_df, test_df* are pandas dataframe. First, we remove the empty records in our dataset by using *fillna()* function. Then we declared *clean_text* function to separate tokens. In code sample 3, we call *clean_text* function to apply it on our training and testing datasets by using *apply()* function of pandas dataframe.

## Text Normalization

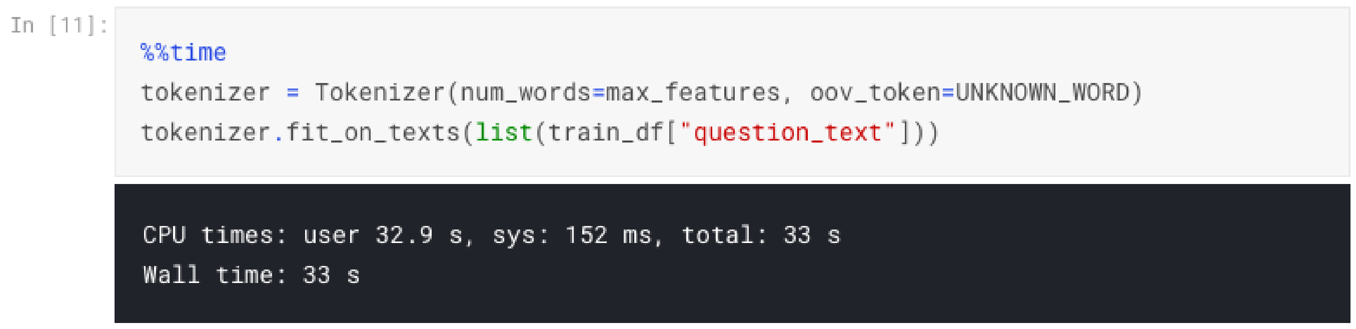
There are many text normalization techniques such as tokenization, lemmatization, stemming, sentence segmentation, spelling correction, etc. Out of these, tokenization is the most used text normalization method.

> Tokenization is the process of converting a piece of text into a list of words or special characters which have meaning in natural language.

For example text “NLP is the best!” can be converted into the list “NLP”, “is”, “the”, “best” and “!”(Note that special character “!” is separated from “best” because it has a special meaning in the English language). Some languages like Chinese doesn't have words separated by spaces, so tokenization for these languages is even more difficult.

Lemmatization is the task of determining if two words have the same root. For example, the words ‘went’ and ‘gone’ are forms of the verb ‘go’. Stemming is similar to lemmatization but it just strips from the end of the word. Using lemmatization or stemming means you are throwing away some information from your data. Spelling correction can be used in your NLP systems to eliminate the input error and raise the performance of NLP system. Nevertheless, NLP systems are assumed to be robust to a small variation in input due to unknown tokens.

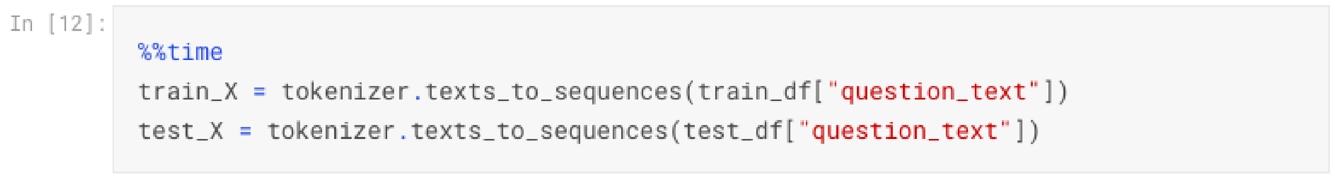
**Code sample:**



```
In [11]:
%%time
tokenizer = Tokenizer(num_words=max_features, oov_token=UNKNOWN_WORD)
tokenizer.fit_on_texts(list(train_df["question_text"]))
```

```
CPU times: user 32.9 s, sys: 152 ms, total: 33 s
Wall time: 33 s
```

**Code sample 4:** Declaring and fitting tokenizer on training data



```
In [12]:
%%time
train_X = tokenizer.texts_to_sequences(train_df["question_text"])
test_X = tokenizer.texts_to_sequences(test_df["question_text"])
```

**Code sample 5:** Using trained tokenizer to tokenize all questions

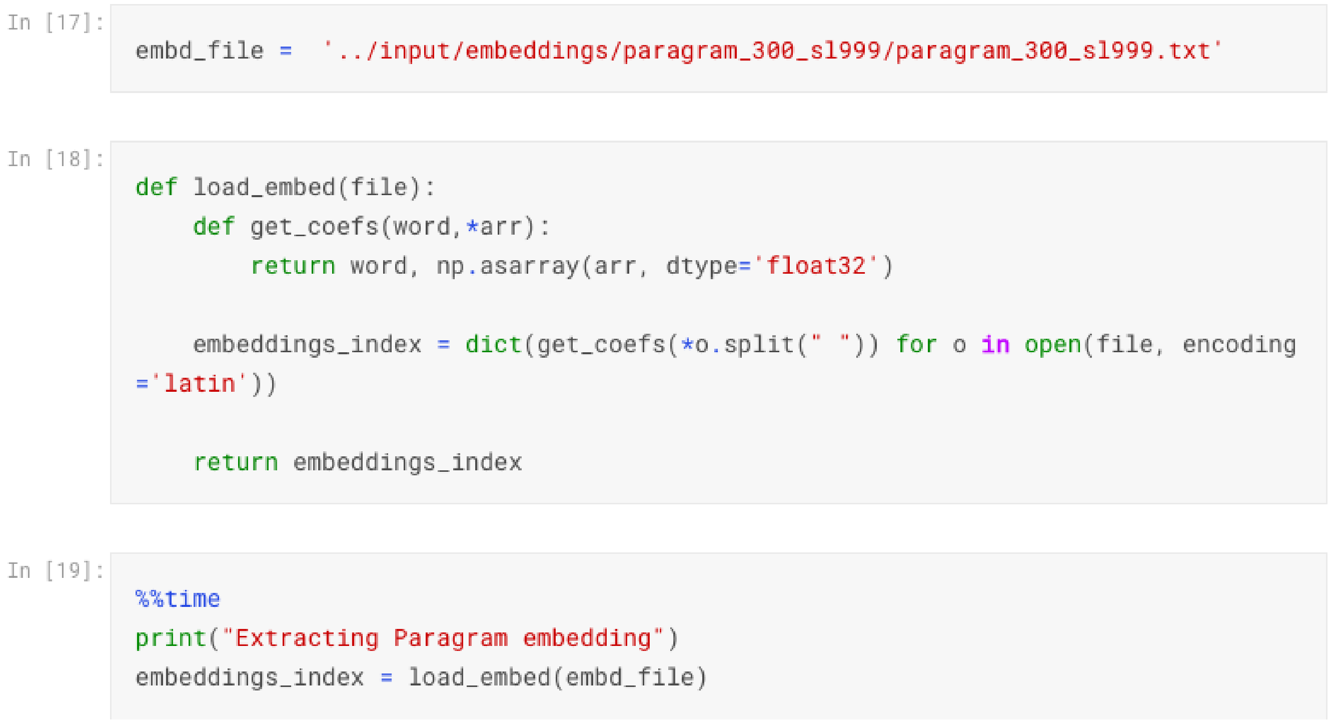
In the above code samples, we train a tokenizer by using *fit_on_texts()* function of a *Tokenizer* object. This object can be used to convert text into an integer representation of input data. After this, we can limit the size of the input sequence by using *pad_sequences()* function from *Keras* package.

## Word embeddings

Word embedding is the representation of document vocabulary in a vector space. Embedding vector with size ranging from 50 to 600, can capture both syntactic and semantic information. If enough computational resources are available, then even bigger vector size can be used. The performance of word embeddings can vary either because of the different algorithm used or the size of the embedding vector. But at some point increasing embedding size doesn't improve the performance.

Language model (LM) can replace word embedding in NLP systems. LM is different from word embeddings as they can be trained and fine-tuned on a corpus. Word embeddings are treated as a single layer and can't be tuned further. A statistical language model is a probability distribution over sequences of words. One such example is the N-gram model. The N-gram models calculate the probability of word 'w' appearing after 'h' words.

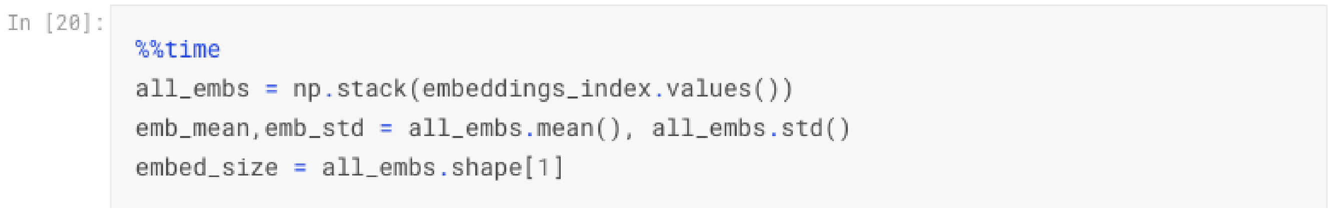
**Code sample:**



```
In [17]:
embd_file =  '../input/embeddings/paragram_300_sl999/paragram_300_sl999.txt'

In [18]:
def load_embed(file):
    def get_coefs(word,*arr):
        return word, np.asarray(arr, dtype='float32')

    embeddings_index = dict(get_coefs(*o.split(" ")) for o in open(file, encoding
='latin'))

    return embeddings_index

In [19]:
%%time
print("Extracting Paragram embedding")
embeddings_index = load_embed(embd_file)
```

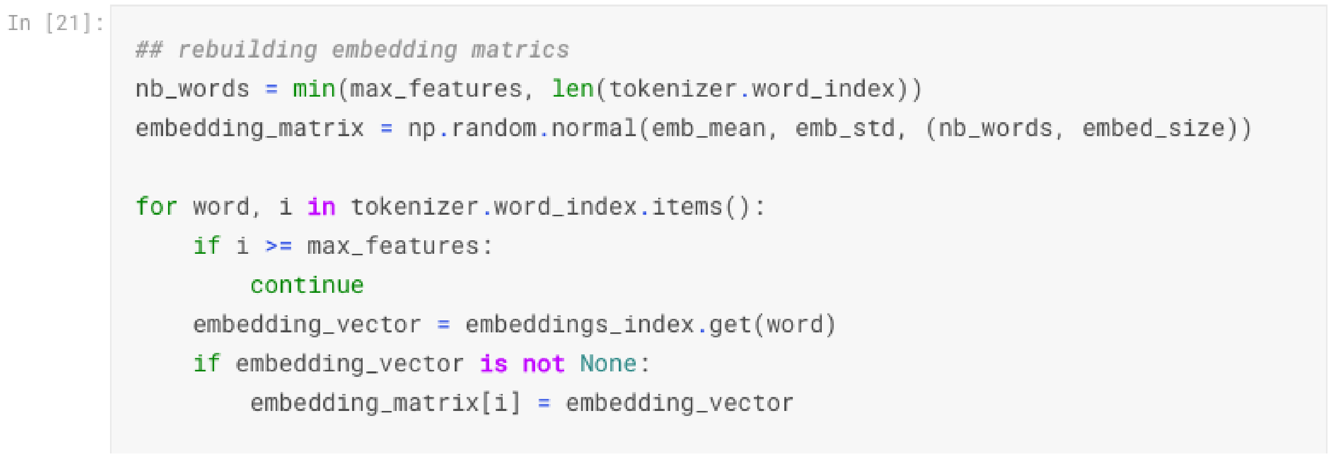
**Code sample 6:** Reading embedding file format

To use word embedding in our code, we read and process the embedding file. In each line, the embedding file contains a word and its embedding vector. In the code sample 6, we split the line and create a dictionary which stores the embedding vector for each word.



```
In [20]:
%%time
all_embs = np.stack(embeddings_index.values())
emb_mean,emb_std = all_embs.mean(), all_embs.std()
embed_size = all_embs.shape[1]
```

**Code sample 7:** Extracting embedding vectors and calculating mean and standard deviation



```
In [21]:
## rebuilding embedding matrics
nb_words = min(max_features, len(tokenizer.word_index))
embedding_matrix = np.random.normal(emb_mean, emb_std, (nb_words, embed_size))

for word, i in tokenizer.word_index.items():
    if i >= max_features:
        continue
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
```

**Code sample 8:** Recreating embedding matrics

After reading the embedding file, we can create a list of used words and embedding matrics. Note that the code sample 8 uses the *tokenizer* object to access the vocabulary words. Here, we recreate the embedding matrics using the normal distribution. This helps in initializing vectors of words that are not present in the dictionary *embeddings_index.*

**Introduction to Word Embedding and Word2Vec**

Word embedding is one of the most popular representation of document vocabulary. It is capable of capturing context of…

towardsdatascience.com

For a detailed understanding of word embedding and word2vec read above article.

## Building model

Models in machine learning pipeline are a sequence of mathematical operations, which

You can start with very basic models if you are a newbie. But you should experiment with at least a few models to see which parameter settings give you the best results. Once you have six or more models that are working good, you can ensemble them into one huge model.

Initially, you can choose to use double LSTM or GRU layers and average pooling layer for testing. After experimenting, you can make changes to your model by making it even deeper or using more advanced state-of-the-art models. For tracking NLP progress, you can visit this site and see which models perform better for a specific task.

The following diagram gives a conceptual representation of the model we have chosen for our first test.

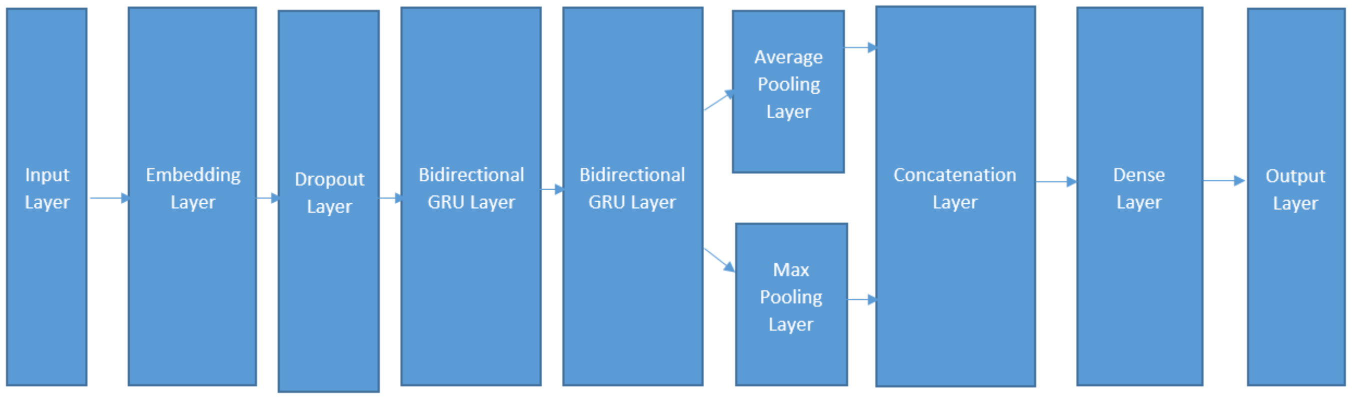


**Figure 2**: Conceptual representation of our model

In the diagram above, you can see the conceptual layout of our model. A layer in a model represents a specific math operation. A model can have a minimum of two layers, the input layer, and output layer. **Input layer** in Keras is used to instantiate a tensor. The **output layer** in a model is a neural network with several nodes.

**Embedding layer** in a model performs dot product on the input sequence and embedding matrix, which converts every word index into the corresponding embedding vector.

**Dropout layer** randomly drops the input by a given percentage. This layer converts the input unit to zero. This process helps in preventing overfitting.

and backward GRU.

Pooling operation decreases the size of the input sequence by performing a mathematical average, maximum, etc. Thus, the **average pooling layer** performs average pooling also **max-pooling layer** performs maximum pooling. **Concatenation layer** combines different input sequences.

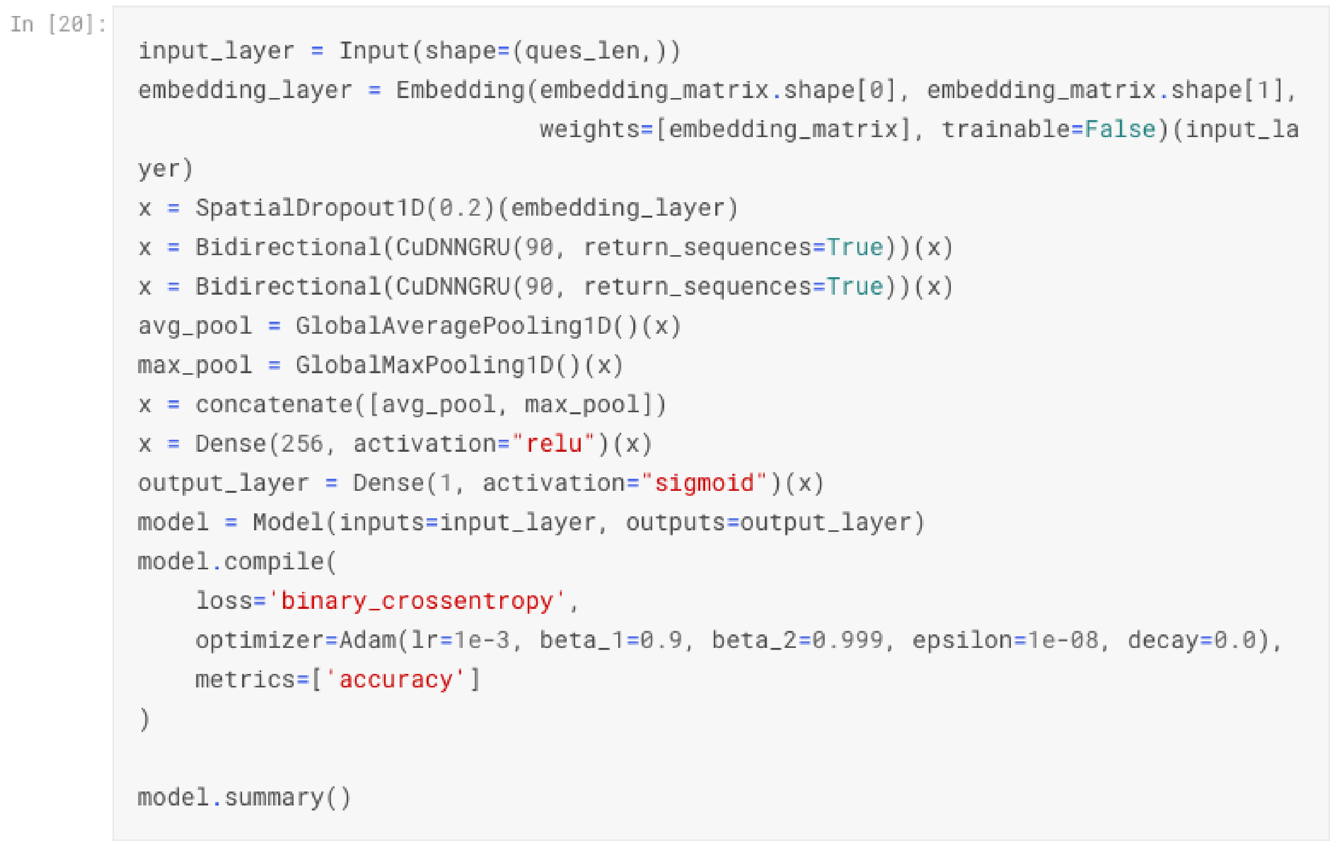
The **dense layer** is a simple neural network with a fixed number of nodes and a specified activation function.

For a complete understanding of working of Recurrent Neural Network and GRU Networks, you can read the following article.

**Illustrated Guide to Recurrent Neural Networks**
Understanding the Intuition
towardsdatascience.com

**Understanding GRU Networks**
In this article, I will try to give a fairly simple and understandable explanation of one really fascinating type of…
towardsdatascience.com

## Code sample:



```
In [20]:
input_layer = Input(shape=(ques_len,))
embedding_layer = Embedding(embedding_matrix.shape[0], embedding_matrix.shape[1],
                            weights=[embedding_matrix], trainable=False)(input_la
yer)
x = SpatialDropout1D(0.2)(embedding_layer)
x = Bidirectional(CuDNNGRU(90, return_sequences=True))(x)
x = Bidirectional(CuDNNGRU(90, return_sequences=True))(x)
avg_pool = GlobalAveragePooling1D()(x)
max_pool = GlobalMaxPooling1D()(x)
x = concatenate([avg_pool, max_pool])
x = Dense(256, activation="relu")(x)
output_layer = Dense(1, activation="sigmoid")(x)
model = Model(inputs=input_layer, outputs=output_layer)
model.compile(
    loss='binary_crossentropy',
    optimizer=Adam(lr=1e-3, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0),
    metrics=['accuracy']
)

model.summary()
```

**Code sample 9**: Model definition code

In code sample 9, the input layer takes input of size equal to *ques_len*. The embedding layer takes the embedding matrix and its dimensions as parameters. The output of the embedding layer is of size *ques_len* x *embedding_matrix.shape[1]*. The *SpatialDropout* layer randomly drops the output of *Embedding* layer with a fraction of 0.2.

Next two layers are bidirectional GRU, which is a type of recurrent neural networks. *GlobalAveragePooling1D* and *GlobalMaxPooling1D* extract the average and maximum features from the output of the second bidirectional GRU layer. The output, *avg_pool,* and *max_pool* is concatenated to feed into the dense layer. The last layer is the output layer which gives a number between 0 and 1.

## Initialising model

Transfer learning is the improvement of learning in a

In this phase, we try to make use of previous knowledge in our NLP system. Using a separate model trained in a different setting can improve the performance of our model. This process is known as knowledge transfer. The most popular method for initialization of Neural Networks in NLP systems is Word Embeddings.

Recent developments in NLP has clearly shown that Transfer learning is the way to move forward. Using pre-trained LMs, new state-of-the-art models are getting better and better. LMs are pre-trained on an enormous dataset also they are based on Transformer architecture.

If you want to read more about transformer models read the following paper.

**Attention Is All You Need**

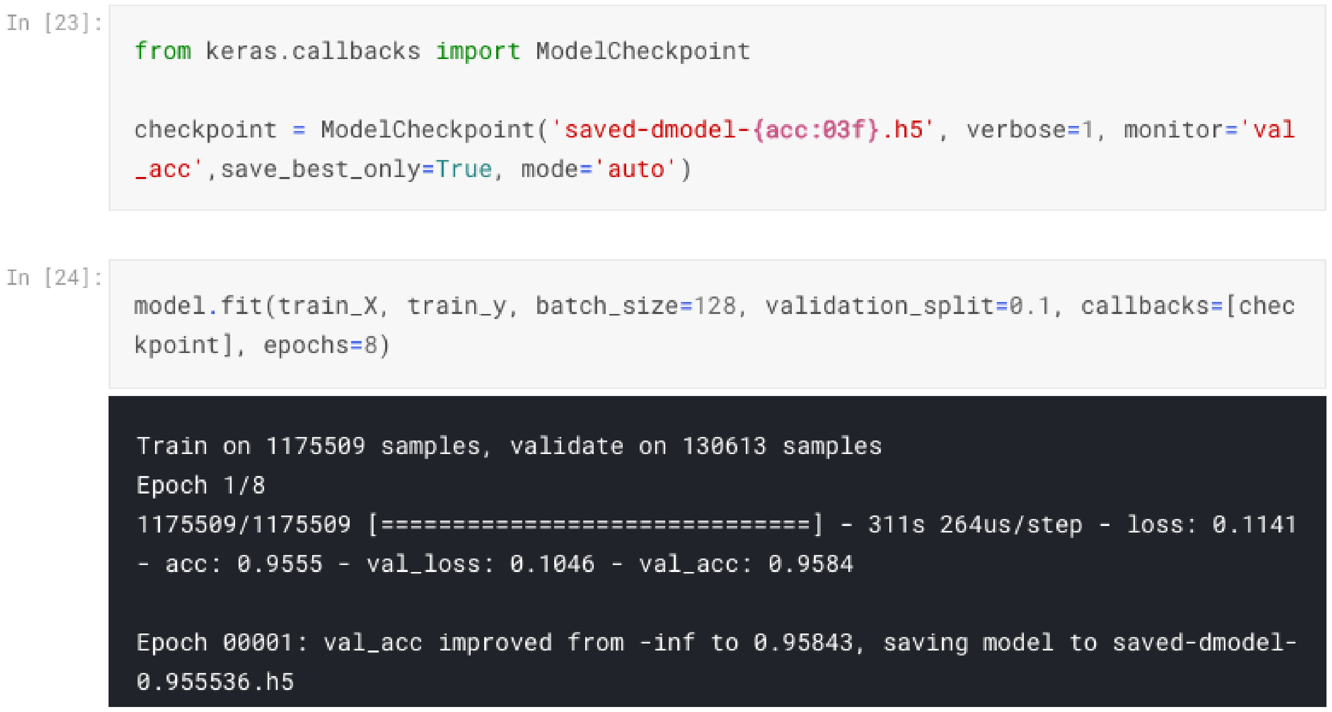
The dominant sequence transduction models are based on complex recurrent or convolutional neural networks in an…

arxiv.org

In coding sample 9, we initialized our model using *embedding_layer* which is initialized to the weight of *embedding_matrix*.

## Training model

After you have decided which model you will be using for the first time, you can train your model. You should always start with testing your model on one-tenth of your training dataset, as it makes the testing much faster. There are some problems where different preprocessing methods might give you varying performance.



```
In [23]:
from keras.callbacks import ModelCheckpoint

checkpoint = ModelCheckpoint('saved-dmodel-{acc:03f}.h5', verbose=1, monitor='val
_acc',save_best_only=True, mode='auto')
```

```
In [24]:
model.fit(train_X, train_y, batch_size=128, validation_split=0.1, callbacks=[chec
kpoint], epochs=8)
```

```
Train on 1175509 samples, validate on 130613 samples
Epoch 1/8
1175509/1175509 [==============================] - 311s 264us/step - loss: 0.1141
- acc: 0.9555 - val_loss: 0.1046 - val_acc: 0.9584

Epoch 00001: val_acc improved from -inf to 0.95843, saving model to saved-dmodel-
0.955536.h5
```

**Code sample 10**: Training our model using model.fit() method

In code sample 10, we add the functionality to save the model every time, if its validation accuracy is increased. Then lastly we call the function *fit(),* to train the model using *train_X, train_y* (our features and labels) and a bunch of other parameters. We can always increase the batch size if we want to do training at a faster speed. The number of *epochs* indicates how many times the same training data will be fed to the model.

## Testing and evaluating the model

After training the model, we must test our model for its accuracy and evaluate how well it does on unseen data. For that purpose, we predict and compare the results with actual labels. If the performance of the model isn't according to our expectations, we need to make changes in our pipeline. In Kaggle competitions, Leaderboard score can be a great way of evaluating our model.