What Kagglers are using for Text Classification

December 17, 2018

With the problem of Image Classification is more or less solved by Deep learning, *Text Classification is the next new developing theme in deep learning*. For those who don't know, Text classification is a common task in natural language processing, which transforms a sequence of text of indefinite length into a category of text. How could you use that?

- To find sentiment of a review.
- Find toxic comments in a platform like Facebook
- Find Insincere questions on Quora. A current ongoing competition on kaggle
- Find fake reviews on websites
- Will a text advert get clicked or not

And much more. The whole internet is filled with text and to categorise that information algorithmically will only give us incremental benefits to say the least in the field of AI.

Here I am going to use the data from Quora's Insincere questions to talk about the different models that people are building and sharing to perform this task. Obviously these standalone models are not going to put you on the top of the leaderboard, yet I hope that this ensuing discussion would be helpful for people who want to learn

more about text classification. This is going to be a long post in that regard.

As a side note: if you want to know more about NLP, I would like to recommend this awesome course on **Natural Language Processing** in the **Advanced machine learning specialization**. You can start for free with the 7-day Free Trial. This course covers a wide range of tasks in Natural Language Processing from basic to advanced: sentiment analysis, summarization, dialogue state tracking, to name a few.

Also take a look at my other post: **Text Preprocessing Methods for Deep Learning**, which talks about different preprocessing techniques you can use for your NLP task and **how to switch from Keras to Pytorch**.

So let me try to go through some of the models which people are using to perform text classification and try to provide a brief intuition for them.

1. TextCNN:

The idea of using a CNN to classify text was first presented in the paper **Convolutional Neural Networks for Sentence Classification** by Yoon Kim. Instead of image pixels, the input to the tasks are sentences or documents represented as a matrix. Each row of the matrix corresponds to one word vector. That is, each row is word-vector that represents a word. Thus a sequence of max length 70 gives us a image of 70(max sequence length)x300(embedding size)

Production with these 5 simple steps

3 Mistakes you should not make in a Data Science Interview

3 Programming concepts for Data Scientists

How to write Web apps using simple Python for Data Scientists?

Now for some intuition. While for a image we move our conv filter horizontally also since here we have fixed our kernel size to filter_size x embed_size i.e. (3,300) we are just going to move down for the convolution taking look at three words at once since our filter size is 3 in this case.Also one can think of filter sizes as unigrams, bigrams, trigrams etc. Since we are looking at a context window of 1,2,3, and 5 words respectively. Here is the text classification network coded in Keras:

```
# https://www.kaggle.com/yekenot/2dcnn-textclassifier
def model_cnn(embedding_matrix):
    filter_sizes = [1,2,3,5]
    num_filters = 36

    inp = Input(shape=(maxlen,))
    x = Embedding(max_features, embed_size, weights=
[embedding_matrix])(inp)
    x = Reshape((maxlen, embed_size, 1))(x)

    maxpool_pool = []
    for i in range(len(filter_sizes)):
        conv = Conv2D(num_filters, kernel_size=
(filter_sizes[i], embed_size),

kernel_initializer='he_normal', activation='elu')(x)
        maxpool_pool.append(MaxPool2D(pool_size=(maxlen -
filter_sizes[i] + 1, 1))(conv))

    z = Concatenate(axis=1)(maxpool_pool)
    z = Flatten()(z)
```

```
    z = Dropout(0.1)(z)

    outp = Dense(1, activation="sigmoid")(z)

    model = Model(inputs=inp, outputs=outp)
    model.compile(loss='binary_crossentropy',
optimizer='adam', metrics=['accuracy'])

    return model
```

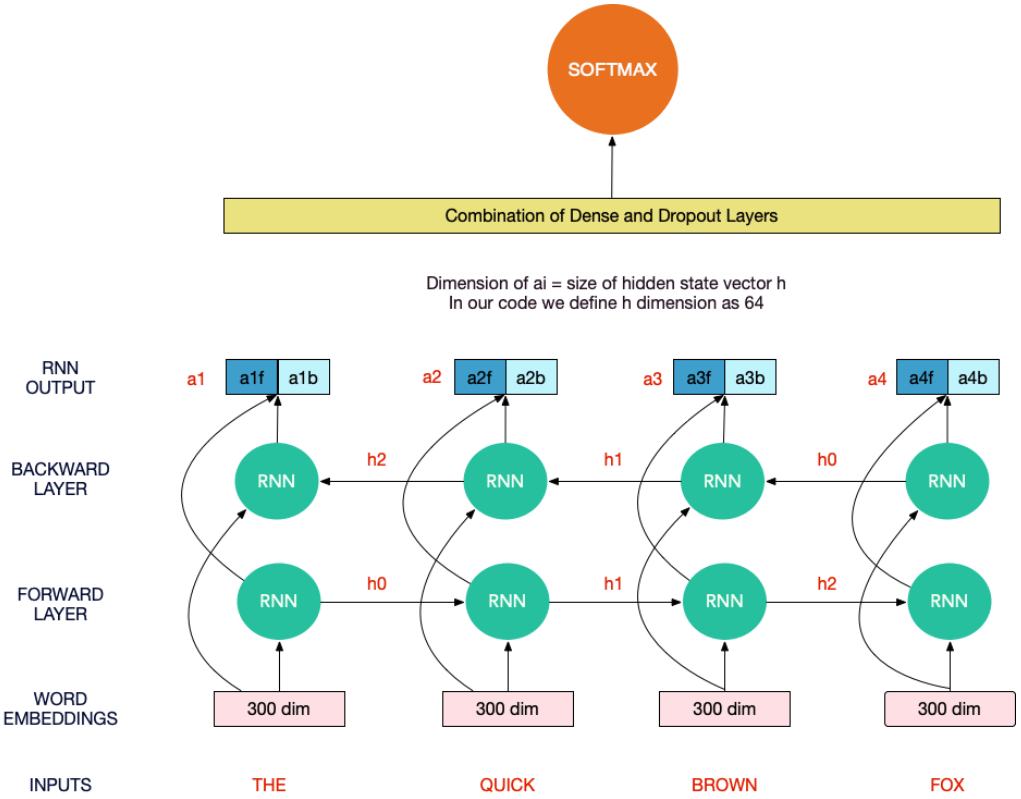
I have written a simplified and well commented code to run this network(taking input from a lot of other kernels) on a **kaggle kernel** for this competition. Do take a look there to learn the preprocessing steps, and the word to vec embeddings usage in this model. You will learn something. Please do upvote the kernel if you find it helpful. This kernel scored around 0.661 on the public leaderboard.

2. BiDirectional RNN(LSTM/GRU):

TextCNN takes care of a lot of things. For example it takes care of words in close range. It is able to see “new york” together. But it still can’t take care of all the context provided in a particular text sequence. It still does not learn the seem to learn the sequential structure of the data, where every word is dependednt on the previous word. Or a word in the previous sentence.

RNN help us with that. *They are able to remember previous information using hidden states and connect it to the current task.*

Long Short Term Memory networks (LSTM) are a subclass of RNN, specialized in remembering information for a long period of time. More over the Bidirectional LSTM keeps the contextual information in both directions which is pretty useful in text classification task (But won’t work for a time sweries prediction task).



For a most simplistic explanation of Bidirectional RNN, think of RNN cell as taking as input a hidden state(a vector) and the word vector and giving out an output vector and the next hidden state.

```
        Hidden state, Word vector ->(RNN Cell) -> Output
Vector , Next Hidden state
```

For a sequence of length 4 like ‘you will never believe’, The RNN cell will give 4 output vectors. Which can be concatenated and then used as part of a dense feedforward architecture.

In the Bidirectional RNN the only change is that we read the text in the normal fashion as well in reverse. So we stack two RNNs in parallel and hence we get 8 output vectors to append.

Once we get the output vectors we send them through a series of dense layers and finally a softmax layer to build a text classifier.

Due to the limitations of RNNs like not remembering long term dependencies, in practice we almost always use LSTM/GRU to model long term dependencies. In such a case you can just think of the RNN cell being replaced by a LSTM cell or a GRU cell in the above

figure. An example model is provided below. You can use CuDNNGRU interchangably with CuDNNLSTM, when you build models.

```
# BiDirectional LSTM
def model_lstm_du(embedding_matrix):
    inp = Input(shape=(maxlen,))
    x = Embedding(max_features, embed_size, weights=
[embedding_matrix])(inp)
    '''
    Here 64 is the size(dim) of the hidden state vector as
well as the output vector. Keeping return_sequence we want
the output for the entire sequence. So what is the
dimension of output for this layer?
        64*70(maxlen)*2(bidirection concat)
    CuDNNLSTM is fast implementation of LSTM layer in Keras
which only runs on GPU
    '''
    x = Bidirectional(CuDNNLSTM(64, return_sequences=True))
(x)
    avg_pool = GlobalAveragePooling1D()(x)
    max_pool = GlobalMaxPooling1D()(x)
    conc = concatenate([avg_pool, max_pool])
    conc = Dense(64, activation="relu")(conc)
    conc = Dropout(0.1)(conc)
    outp = Dense(1, activation="sigmoid")(conc)
    model = Model(inputs=inp, outputs=outp)
    model.compile(loss='binary_crossentropy',
optimizer='adam', metrics=['accuracy'])
    return model
```

I have written a simplified and well commented code to run this network(taking input from a lot of other kernels) on a **kaggle kernel** for this competition. Do take a look there to learn the preprocessing steps, and the word to vec embeddings usage in this model. You will learn something. Please do upvote the kernel if you find it helpful. This kernel scored around 0.671 on the public leaderboard.

3. Attention Models

The concept of Attention is relatively new as it comes from **Hierarchical Attention Networks for Document Classification** paper written jointly by CMU and Microsoft guys in 2016.

So in the past we used to find features from text by doing a keyword extraction. Some word are more helpful in determining the category of a text than others. But in this method we sort of lost the sequential structure of text. With LSTM and deep learning methods while we are able to take case of the sequence structure we lose the ability to give higher weightage to more important words. Can we have the best of both worlds?

And that is attention for you. In the author's words:

> Not all words contribute equally to the representation of the sentence meaning. Hence, we introduce attention mechanism to extract such words that are important to the meaning of the sentence and aggregate the representation of those informative words to form a sentence vector

In essense we want to create scores for every word in the text, which are the attention similarity score for a word.

To do this we start with a weight matrix(W), a bias vector(b) and a context vector u. All of them will be learned by the optimmization algorithm.

Then there are a series of mathematical operations. See the figure for more clarification. We can think of u1 as non linearity on RNN word output. After that v1 is a dot product of u1 with a context vector u raised to an exponentiation. From an intuition viewpoint,

the value of v1 will be high if u and u1 are similar. Since we want the sum of scores to be 1, we divide v by the sum of v's to get the Final Scores,s

These final scores are then multiplied by RNN output for words to weight them according to their importance. After which the outputs are summed and sent through dense layers and softmax for the task of text classification.

```
def dot_product(x, kernel):
    """
    Wrapper for dot product operation, in order to be
compatible with both
    Theano and Tensorflow
    Args:
        x (): input
        kernel (): weights
    Returns:
    """
    if K.backend() == 'tensorflow':
        return K.squeeze(K.dot(x, K.expand_dims(kernel)),
axis=-1)
    else:
        return K.dot(x, kernel)


class AttentionWithContext(Layer):
    """
    Attention operation, with a context/query vector, for
temporal data.
    Supports Masking.
    Follows the work of Yang et al.
[https://www.cs.cmu.edu/~diyiy/docs/naacl16.pdf]
    "Hierarchical Attention Networks for Document
Classification"
    by using a context vector to assist the attention
    # Input shape
        3D tensor with shape: `(samples, steps, features)`.
    # Output shape
        2D tensor with shape: `(samples, features)`.
    How to use:
    Just put it on top of an RNN Layer (GRU/LSTM/SimpleRNN)
with return_sequences=True.
    The dimensions are inferred based on the output shape
of the RNN.
    Note: The layer has been tested with Keras 2.0.6
```

```
    Example:
        model.add(LSTM(64, return_sequences=True))
        model.add(AttentionWithContext())
        # next add a Dense layer (for
classification/regression) or whatever...
    """

    def __init__(self,
                 W_regularizer=None, u_regularizer=None,
b_regularizer=None,
                 W_constraint=None, u_constraint=None,
b_constraint=None,
                 bias=True, **kwargs):

        self.supports_masking = True
        self.init = initializers.get('glorot_uniform')

        self.W_regularizer =
regularizers.get(W_regularizer)
        self.u_regularizer =
regularizers.get(u_regularizer)
        self.b_regularizer =
regularizers.get(b_regularizer)

        self.W_constraint = constraints.get(W_constraint)
        self.u_constraint = constraints.get(u_constraint)
        self.b_constraint = constraints.get(b_constraint)

        self.bias = bias
        super(AttentionWithContext,
self).__init__(**kwargs)

    def build(self, input_shape):
        assert len(input_shape) == 3

        self.W = self.add_weight((input_shape[-1],
input_shape[-1],),
                                 initializer=self.init,

name='{}_W'.format(self.name),

regularizer=self.W_regularizer,

constraint=self.W_constraint)
        if self.bias:
            self.b = self.add_weight((input_shape[-1],),
                                     initializer='zero',

name='{}_b'.format(self.name),
```

```
regularizer=self.b_regularizer,

constraint=self.b_constraint)

        self.u = self.add_weight((input_shape[-1],),
                                 initializer=self.init,

name='{}_u'.format(self.name),

regularizer=self.u_regularizer,

constraint=self.u_constraint)

        super(AttentionWithContext,
self).build(input_shape)

    def compute_mask(self, input, input_mask=None):
        # do not pass the mask to the next layers
        return None

    def call(self, x, mask=None):
        uit = dot_product(x, self.W)

        if self.bias:
            uit += self.b

        uit = K.tanh(uit)
        ait = dot_product(uit, self.u)

        a = K.exp(ait)

        # apply mask after the exp. will be re-normalized
next
        if mask is not None:
            # Cast the mask to floatX to avoid float64
upcasting in theano
            a *= K.cast(mask, K.floatx())

        # in some cases especially in the early stages of
training the sum may be almost zero
        # and this results in NaN's. A workaround is to add
a very small positive number ε to the sum.
        # a /= K.cast(K.sum(a, axis=1, keepdims=True),
K.floatx())
        a /= K.cast(K.sum(a, axis=1, keepdims=True) +
K.epsilon(), K.floatx())

        a = K.expand_dims(a)
```

```
        weighted_input = x * a
        return K.sum(weighted_input, axis=1)

    def compute_output_shape(self, input_shape):
        return input_shape[0], input_shape[-1]


def model_lstm_atten(embedding_matrix):
    inp = Input(shape=(maxlen,))
    x = Embedding(max_features, embed_size, weights=
[embedding_matrix], trainable=False)(inp)
    x = Bidirectional(CuDNNLSTM(128,
return_sequences=True))(x)
    x = Bidirectional(CuDNNLSTM(64, return_sequences=True))
(x)
    x = AttentionWithContext()(x)
    x = Dense(64, activation="relu")(x)
    x = Dense(1, activation="sigmoid")(x)
    model = Model(inputs=inp, outputs=x)
    model.compile(loss='binary_crossentropy',
optimizer='adam', metrics=['accuracy'])
    return model
```

I have written a simplified and well commented code to run this network(taking input from a lot of other kernels) on a **kaggle kernel** for this competition. Do take a look there to learn the preprocessing steps, and the word to vec embeddings usage in this model. You will learn something. Please do upvote the kernel if you find it helpful. This kernel scored around 0.682 on the public leaderboard.

Hope that Helps! Do checkout the kernels for all the networks and see the comments too. I will try to write a part 2 of this post where I would like to talk about capsule networks and more techniques as they get used in this competition.

Here are the kernel links again: **TextCNN**,**BiLSTM/GRU**,**Attention**

Do upvote the kenels if you find them helpful.

References:

- **CNN for NLP**
- **https://en.diveintodeeplearning.org/d2l-en.pdf**

- **https://gist.github.com/cbaziotis/7ef97ccf71cbc14366835198c09809d2**
- **http://univagora.ro/jour/index.php/ijccc/article/view/3142**
- **Shujian's kernel on Kaggle**

DEEP LEARNING NLP PYTHON

« PREVIOUS
To all Data Scientists - The one Graph Algorithm you need to know

NEXT »
A Layman guide to moving from Keras to Pytorch