Work as a detective -- Toxic Comments Detection

Name: Yukang Xu

Ⅰ. Introduction

Discussing thing you care about can be difficult. The threat of abuse and harassment online means that many people stop expressing themselves and give up on seeking different opinions. Platforms struggle to effectively facilitate conversations, leading many communities to limit or completely shut down user comments.

In this project, I am challenged to build a LSTM model that’s capable of detecting different types of toxicity like threats, obscenity, insults, and identity-based hate. I will be using a dataset of comments from Wikipedia’s talk page edits. Model development will hopefully help online discussion become more productive and respectful.

Ⅱ. Data Analysis

A. Identify and Clean the data

Variable	Data type	Handling method
id	Words+ numbers	Drop
comment_text	Words	NA
Toxic; severe_toxic; threat; obscene; insult; identity_hate	categorical	NA

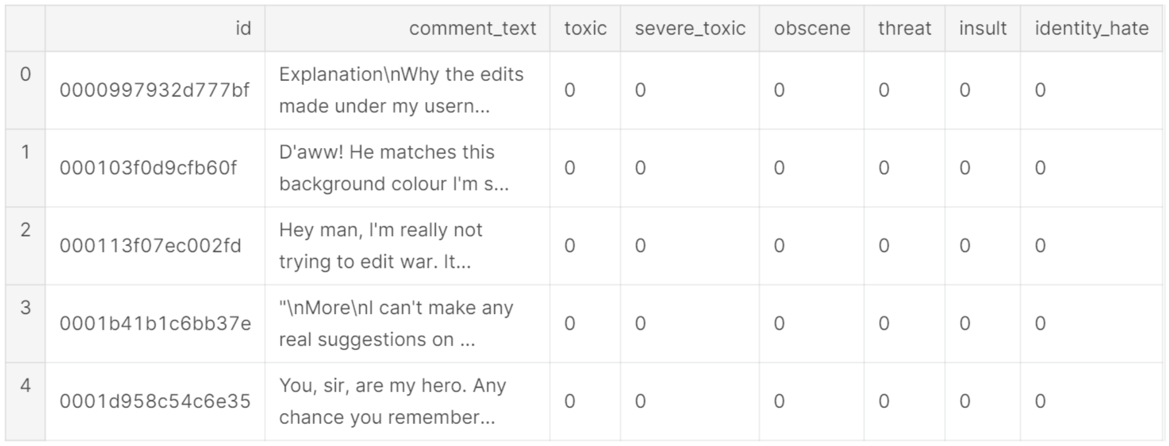
After importing our training and testing data, let us take a look at our dataset (table A.1). There are 6 indicators (toxic, severe toxic, obscene, threat, insult, identity hate) which help to detect toxic comments. If any of indicators is 1, the corresponding comment will be classified to be one type of toxic comment. After training model by these indicators, we can predict the probability of a comment to be toxic including some specific words combination. For data cleaning, we consider these four steps. Since our dataset is complete without missing value, we do not have to go through these steps this time.

a. Correcting: detect the outliers

b. Completing: fill in the missing value

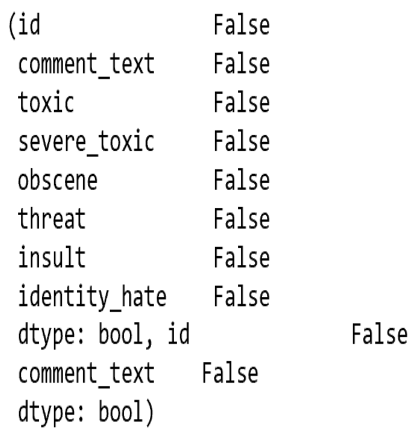
c. Creating: create a new variable: title to see if it plays a role in survival

d. Converting: convert ordinal, nominal data to dummy variables



	id	comment_text	toxic	severe_toxic	obscene	threat	insult	identity_hate
0	0000997932d777bf	Explanation\nWhy the edits made under my usern...	0	0	0	0	0	0
1	000103f0d9cfb60f	D'aww! He matches this background colour I'm s...	0	0	0	0	0	0
2	000113f07ec002fd	Hey man, I'm really not trying to edit war. It...	0	0	0	0	0	0
3	0001b41b1c6bb37e	"\nMore\nI can't make any real suggestions on ...	0	0	0	0	0	0
4	0001d958c54c6e35	You, sir, are my hero. Any chance you remember...	0	0	0	0	0	0

(table A.1)



```
(id               False
 comment_text     False
 toxic            False
 severe_toxic     False
 obscene          False
 threat           False
 insult           False
 identity_hate    False
 dtype: bool, id               False
 comment_text    False
 dtype: bool)
```

(table A.2)

B. Data Preparation

Moving on, as you can see from the sneak peek, the dependent variables are in the training set itself so we need to split them up, into X and Y sets. The approach that we are taking is to feed the comments into the LSTM as part of the neural network but we can't just feed the words as it is. So this is what we are going to do:

1. Tokenization - We need to break down the sentence into unique words. For eg, "I love cats and love dogs" will become ["I","love","cats","and","dogs"]

2. Indexing - We put the words in a dictionary-like structure and give them an index each For eg, {1:"I",2:"love",3:"cats",4:"and",5:"dogs"}

3. Index Representation- We could represent the sequence of words in the comments in the form of index, and feed this chain of index into our LSTM. For eg, [1,2,3,4,2,5]

Note that we have to define the number of unique words in our dictionary when tokenizing the sentences. To avoid different lengths of sentences, We could make the shorter sentences as long as the others by filling the shortfall by zeros. But on the other hand, we also have to trim the longer ones to the same length(maxlen) as the short ones.

C. Model Development

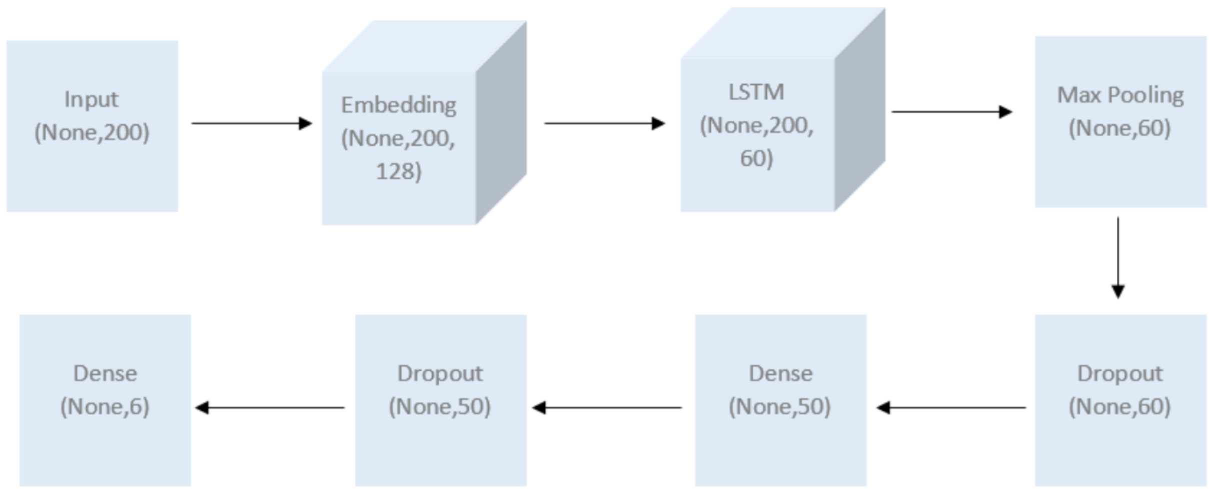
Let us get started with an intuitive explanation about how LSTM works.

Let's say you're looking at reviews online to determine if you want to buy Life cereal (don't ask me why). You'll first read the review then determine if someone thought it was good or if it was bad.

When you read the review, your brain subconsciously only remembers important keywords. You pick up words like "amazing" and "perfectly balanced breakfast". You don't care much for words like "this", "gave", "all", "should", etc. If a friend asks you the next day what the review said, you probably wouldn't remember it word for word. You might remember the main points though like "will definitely be buying again". If you're a lot like me, the other words will fade away from memory.

And that is essentially what an LSTM or GRU does. It can learn to keep only relevant information to make predictions, and forget non relevant data. In this case, the words you remembered made you judge that it was good.

Then let us dig into this whole procedure. This is the architecture of the model we are trying to build. It's always to good idea to list out the dimensions of each layer in the model to think visually and help you to debug later on.



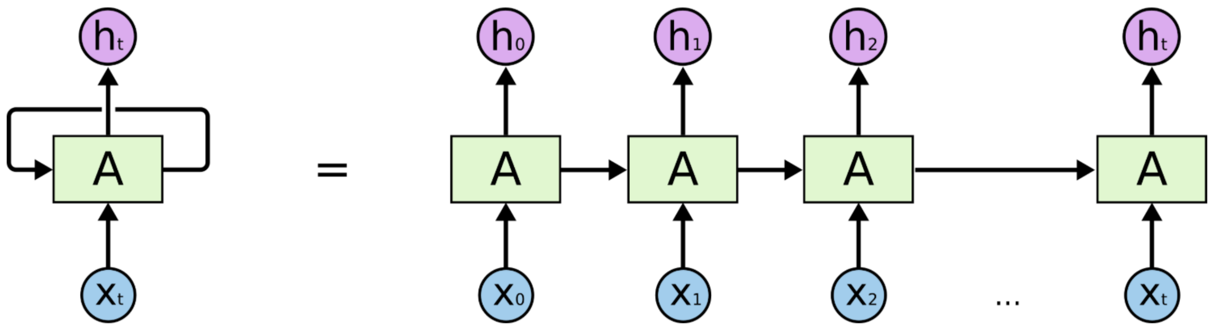
1. Define an input layer: As mentioned earlier, the inputs into our networks are our list of encoded sentences. We begin our defining an Input layer that accepts a list of sentences that has a dimension of 200.

```
# We begin our defining an Input
inp = Input(shape=(maxlen, ))
```

2. Transformed information into machine-readable vectors. Next, we pass it to our Embedding layer, where we project the words to a defined vector space depending on the distance of the surrounding words in a sentence. Embedding allows us to reduce model size and most importantly the huge dimensions we have to deal with, in the case of using one-hot encoding to represent the words in our sentence.

```
embed_size = 128
x = Embedding(max_features, embed_size)(inp)
```

3. Determine what information is relevant to keep or forget. Next, we feed this Tensor into the LSTM layer. We set the LSTM to produce an output that has a dimension of 60 and want it to return the whole unrolled sequence of results. As you probably know, LSTM or RNN works by recursively feeding the output of a previous network into the input of the current network, and you would take the final output after X number of recursion. But depending on use cases, you might want to take the unrolled, or the



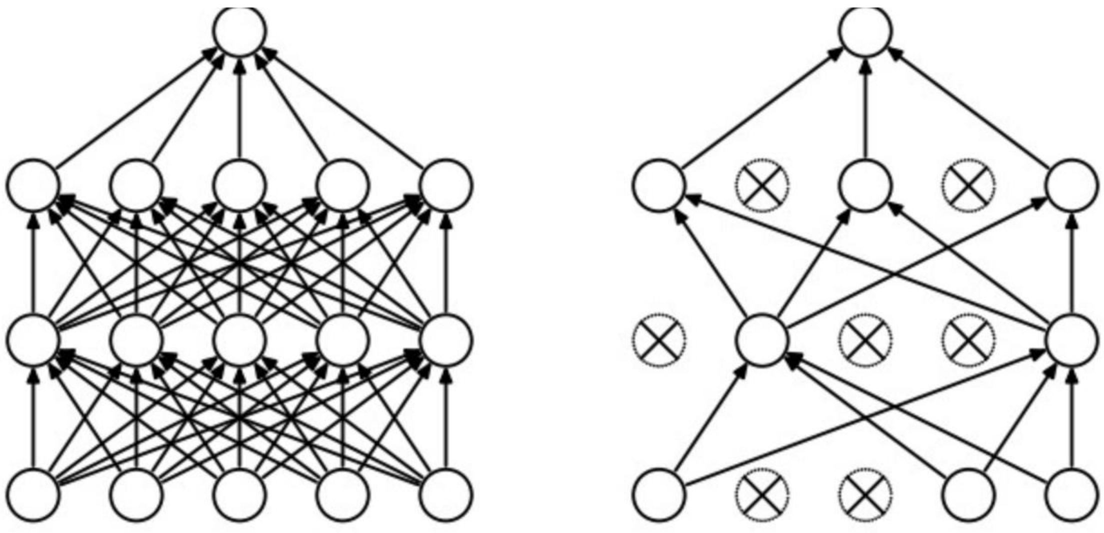
outputs of each recursion as the result to pass to the next layer. And this is the case. it's easy to miss the required input dimensions. LSTM takes in a tensor of [Batch Size, Time Steps, Number of Inputs]. Batch size is the number of samples in a batch, time steps is the number of recursion it runs for each input, or it could be pictured as the number of "A"s in the above picture. Lastly, number of inputs is the number of variables(number of words in each sentence in our case) you pass into LSTM as pictured in "x" above.We can make use of the output from the previous embedding layer which outputs a 3-D tensor of (None, 200, 128) into the LSTM layer. What it does is going through the samples, recursively run the LSTM model for 200 times, passing in the coordinates of the words each time. And because we want the unrolled version, we will receive a Tensor shape of (None, 200, 60), where 60 is the output dimension we have defined.

```
x = LSTM(60, return_sequences=True,name='lstm_layer')(x)
```

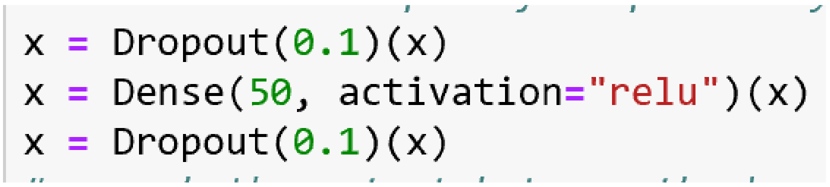
4. Reduce the dimensionality of image data. Before we could pass the output to a normal layer, we need to reshape the 3D tensor into a 2D one. We reshape carefully to avoid throwing away data that is important to us, and ideally we want the resulting data to be a good representative of the original data. Therefore, we use a Global Max Pooling layer which is traditionally used in CNN problems to reduce the dimensionality of image data. In simple terms, we go through each patch of data, and we take the maximum values of each patch. These collection of maximum values will be a new set of down-sized data we can use.

```
x = GlobalMaxPool1D()(x)
```

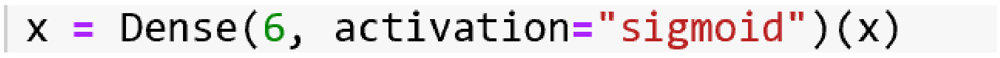
5. Drop out some unimportant layers. With a 2D Tensor in our hands, we pass it to a Dropout layer which indiscriminately "disable" some nodes so that the nodes in the next layer is forced to handle the representation of the missing data and the whole network could result in better generalization.


(a) Standard Neural Net (b) After applying dropout.

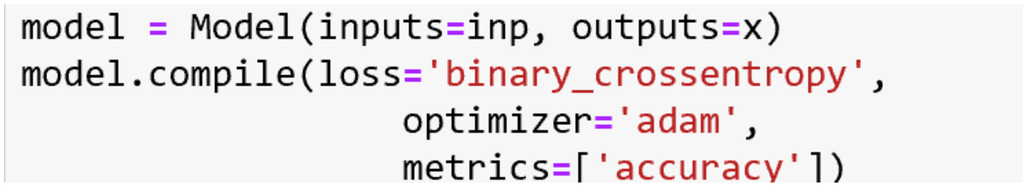
After a drop out layer, we connect the output of drop out layer to a densely connected layer and the output passes through a RELU function. In short, this is what it does: Activation((Input X Weights) + Bias)


```
x = Dropout(0.1)(x)
x = Dense(50, activation="relu")(x)
x = Dropout(0.1)(x)
```

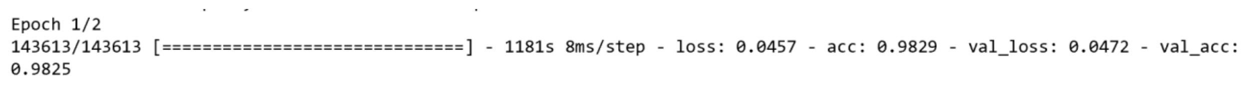
5. Achieve a binary classification. Finally, we feed the output into a Sigmoid layer. The reason why sigmoid is used is because we are trying to achieve a binary classification (1,0) for each of the 6 labels, and the sigmoid function will squash the output between the bounds of 0 and 1.


```
x = Dense(6, activation="sigmoid")(x)
```

We are almost done! All is left is to define the inputs, outputs and configure the learning process. We have set our model to optimize our loss function using Adam optimizer, define the loss function to be "binary cross entropy" since we are tackling a binary classification. In case you are looking for the learning rate, the default is set at 0.001


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

The moment that we have been waiting for as arrived! It's finally time to put our model to the test. We'll feed in a list of 32 padded, indexed sentence for each batch and split 10% of the data as a validation set. This validation set will be used to assess whether the model has overfitted, for each batch. The model will also run for 2 epochs. These are some of the tunable parameters that you can experiment with, to see if you can push the accurate to the next level without crashing your machine(hence the batch size).



```
Epoch 1/2
143613/143613 [==============================] - 1181s 8ms/step - loss: 0.0457 - acc: 0.9829 - val_loss: 0.0472 - val_acc:
0.9825
```

Ⅲ. Conclusion

Since the model from training data perform well, we can apply this model to do prediction