Recurrent Neural Net Assignment

Artificial Intelligence – Spring 2021

Thomas Trankle

April 17, 2021

Summary

This report describes the development of a recurrent artificial neural network to be used as a binary classifier for predicting hate speech in tweets. The background information classifies a tweet as hateful if it contains racist or sexist sentiment. The full data and problem statement can be found here, and was provided by Analytics Vidhya. Overall, the best model that was trained had an accuracy of 0.95, which was primarily due to appropriate units in each of the stacked recurrent layers. The introduction describes some of the modifiable parameters of this recurrent neural network that is built using a Sequential model. Additionally, it outlines some basic characteristics about different recurrent network layers and their respective usefulness. The body of the report steps through necessary preprocessing procedures for the model and data, as well as the main adjustments made for developing a base model, followed by adjustments and the use of additional layers and regularizes to fine-tune the model.

Introduction

The purpose of this report is to provide a detailed analysis of a recurrent neural network, and the layer architecture within, to predict labels for tweets as either racist/sexist (predict 1) or not racist/sexist (predict 0). The data set we are using is hosted by *kaggle* and belongs to the Twitter and Sentiment Analysis problem. This data comes pre-split into training and testing sets. Therefore, it is unnecessary for us to conventionally partition the data after importing, but rather we will split our training set into a train and validation set, leaving the test set untouched. The data consists of 31,962 and 17,197 recorded tweets for the train and test sets respectively. The longest tweet in the training data consists of 137 characters. This will be an important detail related to tokenization, which is described later. Ideally, we hope to find patterns in past tweets so that we can accurately predict tweets with a negative sentiment. So, the overall goal is to maximize the prediction accuracy **hateful tweets** and not necessarily overall accuracy. That is, the number of correctly predicted negative labels over the sum of true negatives and false positives. Commonly, this is known as specificity,

$$Specificity = \frac{True\ Negative}{True\ Negative + False\ Postive}$$

To do this, we will be using Binary Cross Entropy as the loss metric, since it is a very intuitive and common metric when evaluating the performance of a binary-class classification model. The formula is defined as follows,

$$CE = -\sum_{i=1}^{C'=2} t_i log\Big(f(s_i)\Big) = -t_1 log\Big(f(s_1)\Big) - (1-t_1) log\Big(1-f(s_1)\Big)$$

Additionally, we will be using Sigmoid as our activation function for the output layer since this is considered best practice for models with our criteria. As we know, there is no procedural method to define the specific parameters of our model and simply optimize our objective function. Instead, we utilize experimentation to identify different parameters that will, at least on average, produce sufficient results. However, we do utilize a Randomized Grid Search approach to try and best select the hyper-parameters to optimize parts of the model.
The parameters considered were:

- Optimizer
- One or Two Fully Connected Layers
- Number of Nodes per Layer
- Activation Function of Fully Connected Layer

There are certainly more details and/or parameters that could be added to the above list, however, for the purpose of this exercise they were omitted due to hardware constraints. Additionally, a couple of common best practices and preprocessing steps were held constant when designing, tuning, and evaluating the model.

Model Development and Parameter Tuning

The reason we are implementing a recurrent neural network is because this type of network is specifically designed to use *text* to produce a simple form of natural-language understanding and is useful for document – in our case tweets – interpretation (Chollet, 2018). This is achieved by the networks mapping the statistical structure of the written texts and is not to be confused with the actual comprehension of texts by the neural network. This pattern recognition, in turn, is like convolutional neural networks in that it is applied to words much like how convents apply it to pixels. However, convents generally won't perform well on sequentially dependent data. Therefore, we use recurrent networks, as they maintain a level of bias based on prior data. In essence, recurrent neural networks maintain some information/memory about earlier sequences which allows them to "create" a more contextual interpretation of the data.

Once more, these networks cannot process actual characters. So, raw text must be converted to numeric tensors (Chollet, 2018), also known as the vectorization process. For this model, we will be vectorizing our input tweets by segmenting the words in the tweet's, and then transforming each word into a numeric representation. The act of segmenting the input into words is called *tokenization* and each distinct word representation is referred to as a *token*. This pre-processing step is done prior to feeding the tensor into the model.

As previously described, recurrent neural networks have different layers and the way these layers interact with each other also differs. The layers include:

1. Embedding Layer
2. Recurrent Layers
3. Densely or Fully Connected Layer

The act of vectorizing the tokenized representation of inputs occurs in the embedding layer. Using the output dimension specified, the network will learn the appropriate dimensional

embeddings for each of the tokens inputted to match the output dimension. Essentially, we are transforming a 2D integer tensor, containing integers of the tokenized tweets, into a 3D vectorized float tensor. Since we are not using pre-loaded embedded weights, the initial weights created by this layer are randomly generated. However, over time these weights are adjusted using backpropagation which helps to structure the space into something the model can exploit as training/learning continues.

However, this does not account for inter-word relationships and sentence structure. To account for this anomaly, we implement recurrent layers to consider each sequence in its entirety, as well maintaining a *state* of what information is has seen thus far (Chollet, 2018). Classically, we think of the layers in a sequential model as a feed-forward movement. Meaning that the information is passed from one layer to the next in a step-by-step manner. However, with recurrent networks, it is better to imagine a continuous or cyclical path between the recurrent layers themselves. This *for loop* analogy is part of the structure that allows recurrent networks to maintain the bias mentioned above, by essentially being able to access prior steps or "memory".

A simple recurrent layer is often avoided in practice. This is because it tries to remember information about past inputs over the entire training time. These long-term dependencies are impossible to learn because of the *vanishing gradient problem* (Chollet, 2018). Much like feed-forward network implementation, if we have a model that is too deep (large number of layers), it is impossible to train. Similarly, with RNN if the model tries to remember too much, then a similar paradigm occurs.

So, the recurrent layer used in throughout this implementation is an LSTM layer which is a special type of recurrent layer. Long Short-Term Memory (LSTM) is a modified version of a simple recurrent. Let us consider the analogy of basic computer hardware architecture. Generally, a computer has what one calls memory, more commonly referred to as Random Access Memory (RAM), and secondary storage, which is typically called a hard drive. To perform well, a computer stores only current programs or data in memory and leaves the rest in secondary storage. This allows a computer to process executable tasks efficiently whilst reserving the right to refer to any information it has stored in *long term* when it needs it, so long as the long-term memory is not full. Likewise, LSTM reserves the right to access prior information when it needs it, rather than storing all its information together. This, in turn, prevents older signals in the same space from slowly being forgotten as the memory capacity is filled.

Like convolutional networks, recurrent networks use backwards propagation, that is the model gets updated after each layer starting at the output and moving backwards to the input, to assume the weights of the previous layers. To counter this phenomenon, a common technique suggested by literature is to use batch normalization. This helps to adaptively normalize the data even as the mean and variance change/update as the model trains. Essentially, it operates by internally maintaining an exponential moving average of the bath-wise mean and variance for the data seen (Chollet, 2018). This helps coordinate the gradient propagation by stabilizing the outputs of each layer. For this reason, we generally place a BatchNormilzation layer after each convolutional layer since convents are usually deep networks to maximize feature extraction.

Furthermore, a dropout layer between fully connected layers helps prevent overfitting. This step randomly ignores some of the nodes and reduces the size of the network. In essence,

this form of regularization helps prevent spoon feeding amongst certain neurons and rather creates a more robust model as nodes are forced to learn from different subsets of each other.

Preprocessing

The imported feature data (what we use to train our model) is represented by character strings. So, the first step is to tokenize training set into an integer representation. This is easily achieved by using the *keras* tokenizer to transform the data. The tokenizer takes care of a lot of text cleaning, such as stripping special characters ('!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~\t\n') and taking only the specified number of most common words you wish to extract from the data. The training data has 17,738 unique words such as,

I ran several models with different number of max words, ranging from 5,000 to 50,000, but found little effect on the performance of the model. Therefore, I decided to use 5,000 words to simplify the model's parameters.

After we have specified the maximum number of words the tokenizer should extract, we fit it to the training tweets. We then again use the tokenizer object to transform every observation into an integer sequence. The longest cleaned sequence of words was 37, so we pad zeros to sequences less than 37 words to account for the difference and ensure the dimension of every input is the same size.

The next step is to split the training data into training and validation sets. Unfortunately, the data contains 29,720 tweets with the label 0 (not racist/sexist) and only 2,242 tweets with the label 1 (racist/sexist). Therefore, we do not want to use a classical method of simple splitting the data according to the 80 to 20 ratios. Rather, we include a special conditional parameter called *stratifying* to split try partition so that the 0 to 1 label in the train and validation set are proportional and use a 90% to 10% split to let our data train on as many tweets as possible.

A preprocessing step that was later implemented during the model development, was using class weights to adjust for the relative distribution of labeled variables. Class weights is used during model training to penalize the loss function for incorrectly predicting classes with smaller frequencies in the data set. This was done to prevent classes being left out of the validation set and prevent the model from underfitting or simply getting lucky by not being able

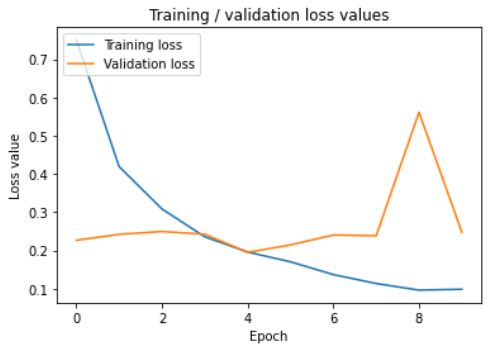
to train on certain classes. To implement the above, we slightly alter our partition statement and calculate the class distributions using *numpy*.

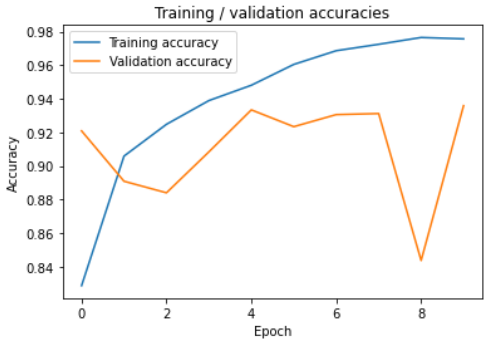
```
unique, counts = np.unique(y_train,axis=0, return_counts=True)
counts = counts.max()/counts
class_weight = {unique[i]:counts[i] for i in range(len(counts))}
```

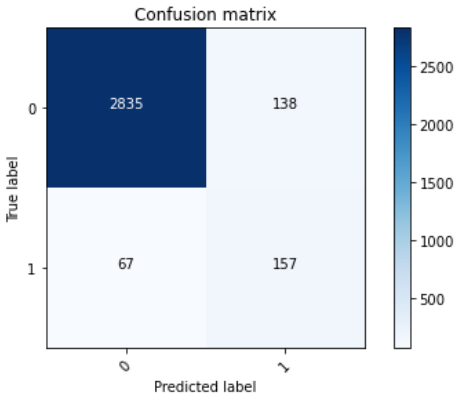
Course of Action and Model Development

After all the preprocessing steps had been implemented, I created an initial model with a embedding layer, two Bidirectional LSTM recurrent layers and only one fully connected layer as suggested in the prompt. This was done to make sure all the data passed through in the correct format and to determine a performance baseline. The output dimension of the embedding layer was 88, which is the median tweet length of characters. The number of units for the two recurrent layers were both 512 and the single fully connected also had 512 nodes. Each of the above recurrent layers used sigmoid as the activation function. The dense layer used ReLu as the activation function and the model was fit using batch size of 256. This model had an accuracy of 0.9358 and a run-time of 12 minutes and 22 seconds. 0.70

```
model = Sequential()
model.add(Embedding(max_words, 88, input_length=maxlen))
model.add(Bidirectional(LSTM(units=512, dropout=0.3, recurrent_dropout=0.0,
                  recurrent_activation="sigmoid", activation="tanh",
                  use_bias=True, unroll=False, return_sequences=True
                  )))
model.add(Bidirectional(LSTM(units=512, dropout=0.3, recurrent_dropout=0.0,
                  recurrent_activation="sigmoid", activation="tanh",
                  use_bias=True, unroll=False, return_sequences=False
                  )))
model.add(Dense(512, activation = 'relu'))
model.add(Dense(1, activation='sigmoid'))
```



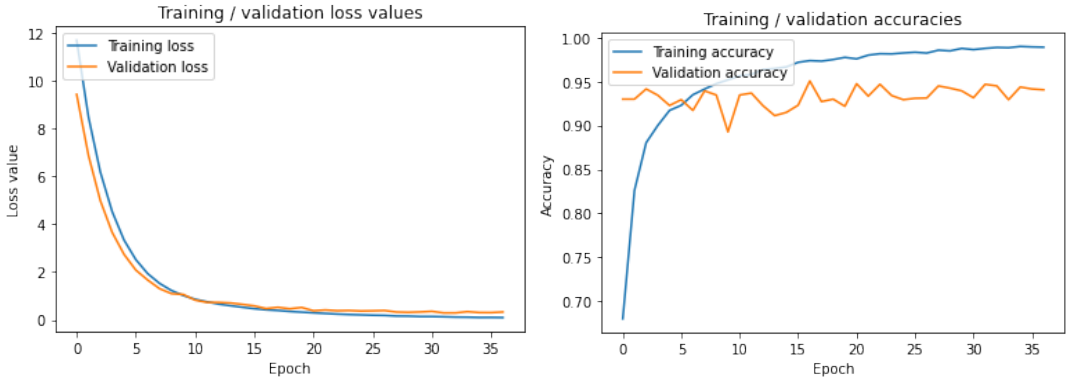


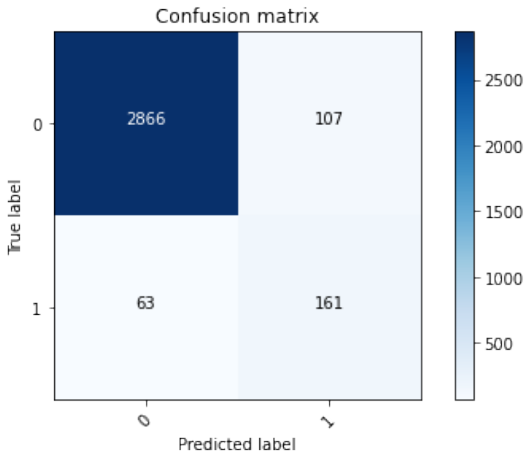


At this stage I decided to use Randomized Grid Search to determine if some of the parameters in my fully connected layer needed to be modified. The results were as follows,

Best: 0.939769 using {'optimizer': 'Adam', 'neurons_second': 128, 'activation2': 'relu', 'activation': 'sigmoid'}
0.931320 (0.010416) with: {'optimizer': 'RMSprop', 'neurons_ second ': 256, 'activation2': 'sigmoid', 'activation': 'sigmoid'}
0.929755 (0.010324) with: {'optimizer': 'SGD', 'neurons_ second ': 512, 'activation2': 'sigmoid', 'activation': 'hard_sigmoid'}
0.070245 (0.010324) with: {'optimizer': 'Adagrad', 'neurons_ second ': 512, 'activation2': 'softmax', 'activation': 'sigmoid'}
0.070245 (0.010324) with: {'optimizer': 'SGD', 'neurons_ second ': 128, 'activation2': 'softmax', 'activation': 'hard_sigmoid'}
0.070245 (0.010324) with: {'optimizer': 'Adamax', 'neurons_ second ': 512, 'activation2': 'softmax', 'activation': 'linear'}
nan (nan) with: {'optimizer': 'Adam', 'neurons_ second ': 0, 'activation2': 'arctan', 'activation': 'sigmoid'}
0.939769 (0.009098) with: {'optimizer': 'Adam', 'neurons_ second ': 128, 'activation2': 'sigmoid', 'activation': 'relu'}
nan (nan) with: {'optimizer': 'Adagrad', 'neurons_ second ': 0, 'activation2': 'softmax', 'activation': 'sigmoid'}
0.928813 (0.038340) with: {'optimizer': 'Adam', 'neurons_third': 256, 'activation2': 'sigmoid', 'activation': 'linear'}
nan (nan) with: {'optimizer': 'Adagrad', 'neurons_third': 0, 'activation2': 'hard_sigmoid', 'activation': 'tanh'}
Time required for training: 6:02:42.777647

I decided to perform the search with only 10 epochs as this would still provide insight on how the model performs. The results of the search indicate that the I needed to change was my activation functions and add a second fully connected layer with 128 nodes. Additionally, due to the sporadic pattern of my graphs above, I added Batch Normalization after every layer and Dropout layer between my dense layers. Furthermore, I added L2 regularizers to my dense layer. Rerunning the above model with the altered parameters only led to a small increase in training accuracy to .9468 percent, a specificity of 0.72 with a similar run time.





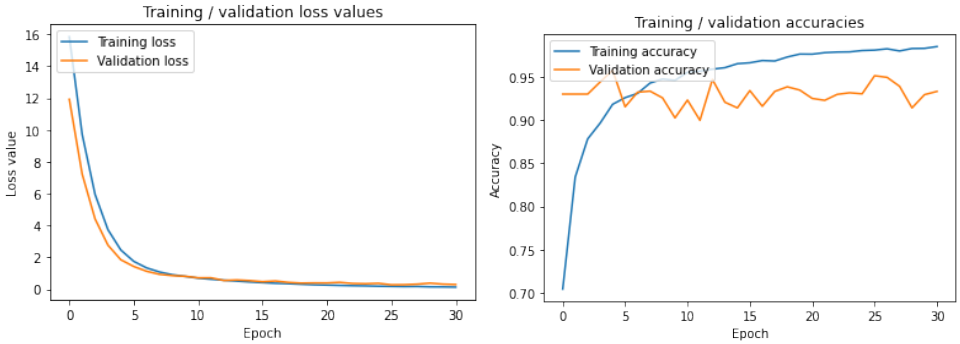
As you can see, the model is no longer overfitting. It is important to construct a model that still has a convergence between our loss values. Otherwise, it is unreasonable to assume that our model is performing well when it is actually just getting lucky or predicted a bulk of 0's which still has a large accuracy. Since we have more appropriate model, we can look at some of the tweets in the test set the model identifies as racist/sexist.

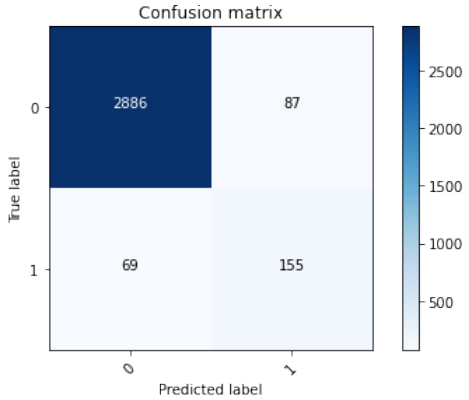
id	tweet	label
31982	thought factory: bbc neutrality on right wing fascism #politics #media #blm #brexit #trump	1
31989	chick gets fucked hottest naked lady	1
31996	suppo the #taiji fisherman! no bullying! no racism! #tweet4taiji #thecove #seashepherd	1
32044	@user .@user @user @user @user <--- no more feeding at the public trough piggy.	1
32073	hey @user - a $14000 ivanka bracelet? do you feel good profiting from #xenophobia? #misogyny?	1
32078	#palladino : "i'll say whatÂ iÂ feel like saying.â #gopathetic #lookwhatyoudid #politics #madcow	1
34696	if pisses you off like it pisses me off, here's something you oughta know... this guy's a racistcreep.â¦	1

Earlier, we mentioned how the weights are randomly initialized when training begins. We now look at the effect of adding more independent copies of our recurrent layers to the model. The width of a recurrent network is tied to changes along the recurrent dimensions. Generally, we want to increase the number of units so that the model can extract more complex information, if need be.

This suggests that we should investigate the architectural complexity of our input by altering these levels. This is accomplished by increasing the number of units for each recurrent layer. For this example, we double the number of units for both recurrent layers to 1024. These independent copies will have the same structure as the current layers but will begin with different

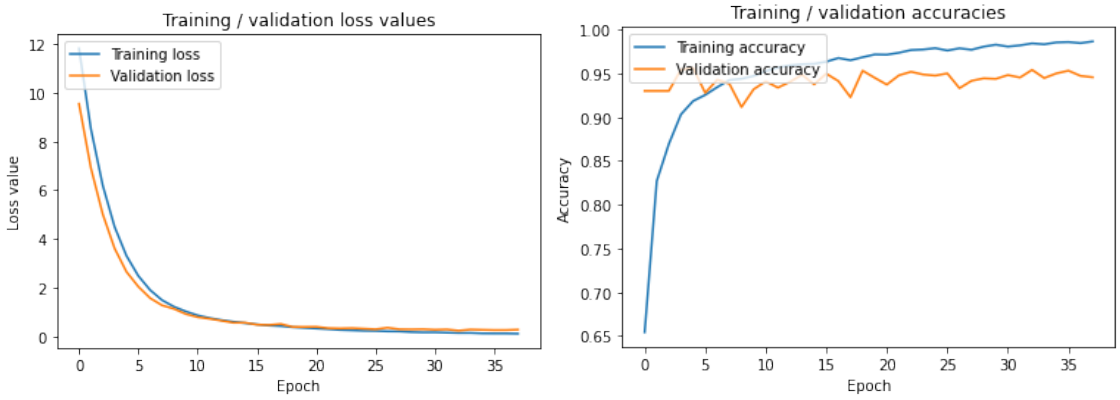
weights. Thus, expanding the width of our model. This resulted in an accuracy of 0.9512, a specificity of 0.69 and a training time of 32 minutes and 13 seconds.

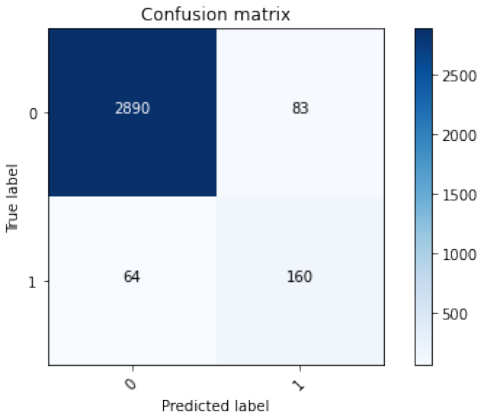




By increasing the width of our model, we have increased the overall accuracy of our model, however we do notice a slight decrease in specificity most likely due to the fact that non racist/sexist tweets are generally more complex and that the model is trying extract complicated structure from racist/sexist tweets when there is perhaps none.

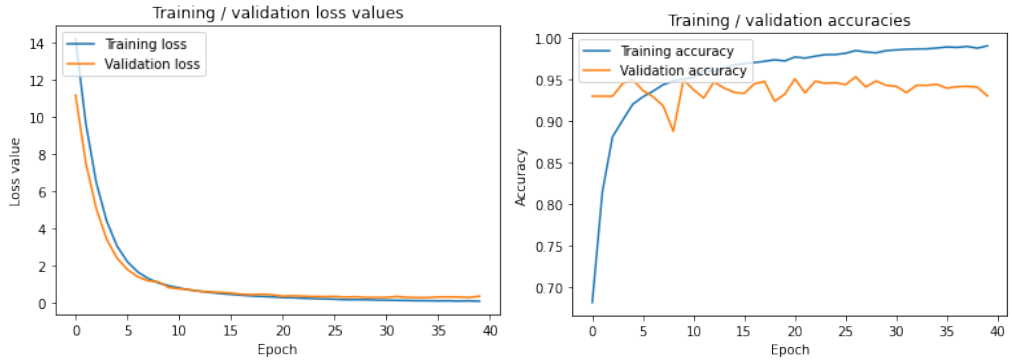
To test this hypothesis, let us examine the result of halving the number of units in both of our LSTM units to 256.

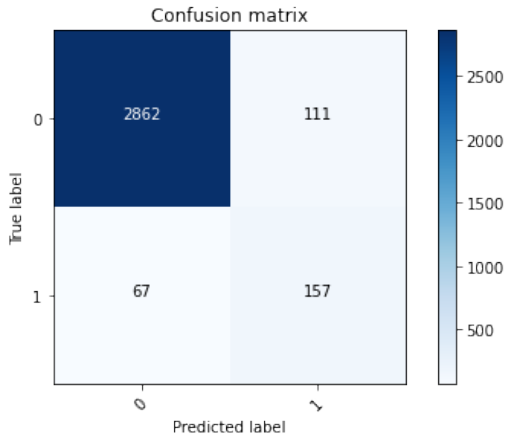




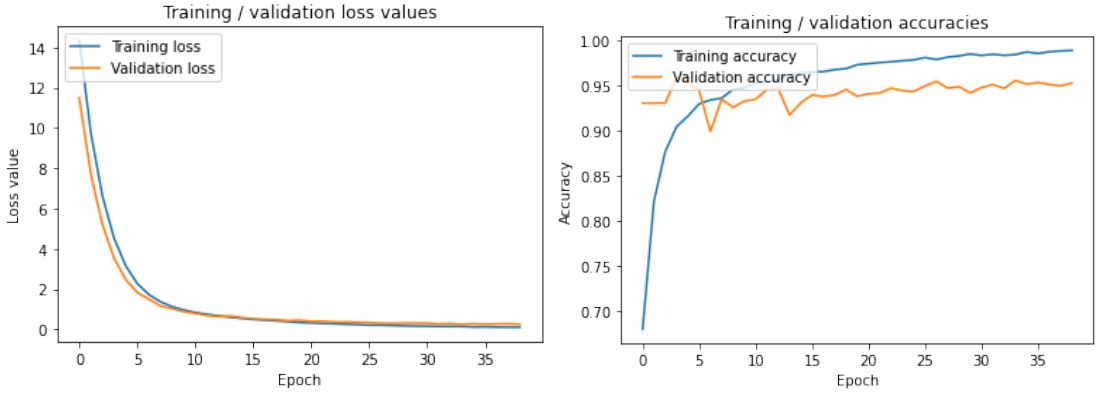
Interestingly, we get an overall accuracy of 0.954, a specificity of 0.71 with a training time of 21 minutes and 28 seconds. From this, we can perhaps deduce that the sequence complexity of racist/sexist tweets is generally simpler than a tweet which is not racist/sexist. Then, if our model in uncertain, it will most likely predict a 0 which leads to an increase in overall accuracy as well.

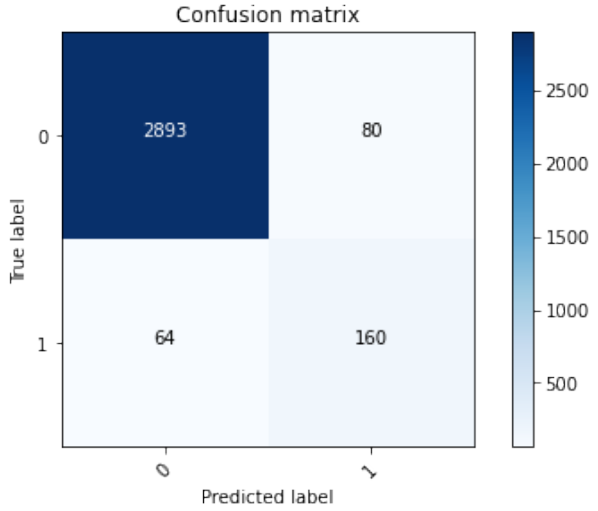
Next, we investigate our sequence length decision of 37. If we double the sequence length to 74, we find the following results.





An overall accuracy of 0.944, specificity of 0.70, and training time of 51 minutes and 16 seconds. Seemingly, there is not a dramatic change when stretching the sequence length beyond the maximum number present in the input data. However, not ensuring at least an adequate sequence length does impact our model's performance. Here we see an accuracy of 0.954, specificity of 0.7 and a training time of 6 minutes and 2 seconds.

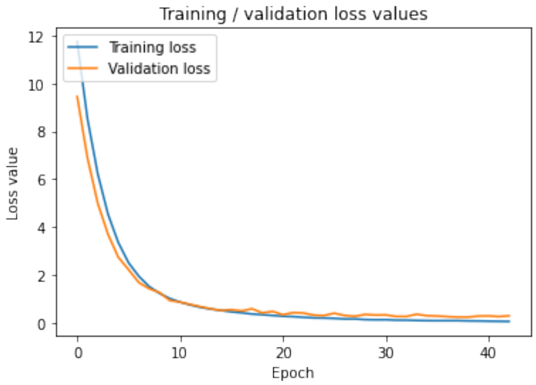


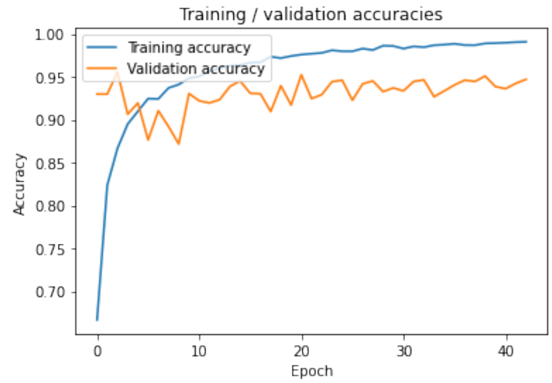


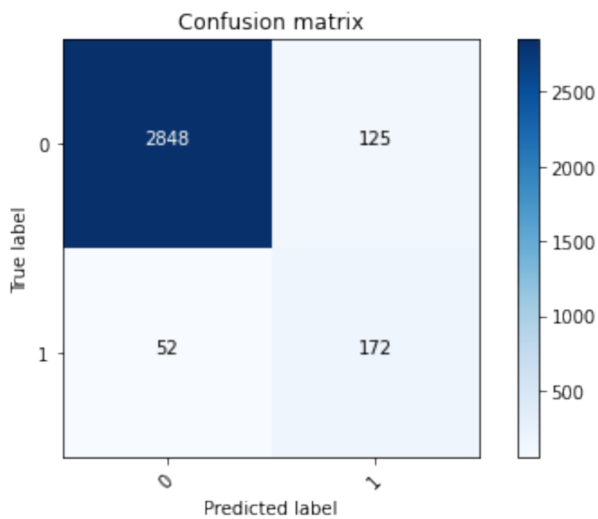
Conclusion

Recurrent neural networks are excellent ways to make predictions based on text due to their "memorization" ability. This is because the recurrent layers can extract various levels of information from any past points to create a more holistic view of the sequence at hand, which when making predictions. Throughout this experiment, we looked at many types of layers and preprocessing (tokenization/embedding) that could be added to a recurrent neural network and the impacts of either including or excluding these layers. For this experiment, we saw that adding sufficient units in combination with batch normalization and dropout was the most beneficial. Even though this was not described above, we also notice the impact of not creating a deep enough recurrent model (stacked) so that the recurrent layers can further exploit the spatial and temporal information of the inputs. When we created a model with 3 recurrent layers, having

1024, 512 and 256 units respectively, we produced an accuracy of 0.945 with a specificity of 0.767. This beneficial to us since we are trying to specifically identify racist/sexist tweets.







Base Model with Modified Recurrent Layers

```
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Import Libraries Section
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
''' import appropriate packages for input, preprocessing, model creation and
 training, and output '''
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from pickle import dump,load
import datetime
import tensorflow as tf
```

```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential,save_model, load_model
from tensorflow.keras.layers import Dense,Embedding,LSTM,\
                              Bidirectional,Dropout,BatchNormalization
from tensorflow.keras import regularizers
from tensorflow import keras
from sklearn.model_selection import train_test_split
import itertools

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
GPU Configuration for OOM
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
from tensorflow.compat.v1 import ConfigProto,Session
config = ConfigProto()
config.gpu_options.allow_growth = True
sess = Session(config=config)
sess.as_default()

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
                           Code Flow

First, we import the relevant data and assign it to our train and test
variables.
   train - represents the tweets and labels we will use to train our model
   test - set of observations containing only tweets

The next step is to define some of the hyper parameters used in the model:
   batch_size
   max sequence length
   epochs
   optimizer
   activation

The next section we begin our preprocessing procedure. This is where we
tokenize our inputs and convert them to a 2D integer tensor. The length of the
tensor is defined by the number of inputs and the width is defined by the max
sequence length we defind above. Once this tokenizer has been fit to the data,
we apply it to our inputs and pad all sequences that are shorter than 137 so
that the dimensionality is constant.
Additionally, we shuffle our unblalnced data set, and split the data into
train and validation sets in a stratified manner.
Furthermore, we calculate inverse proportionality
of our labels because this is an unbalanced data set. Ultimately, this allows
the model to penalize the loss function for incorrectly predictiong classes
that contain few onservations.

The next section is where define the model we want to use.
This model consists of three parts. The first part is the embedding
layer model, and the second part is our recurrent layer and the third is the
```

```
fully connected layers.
    - Embedding layer extracts specified number of features from each input.
      Adds an additional dimension to store this extraction.
      Goes from 2D integer tensor to 3D float tensor.
    - Recurrent layer looks for pattern recognition in the input. However, this
      layer maintains a form of memory to access information from prior stages
      at any time t.
    - Full Connected is used to train classifier portion of the network

After this, we fit our model which basically means we allow it to train
on the training data using the training parameters we have either constructed
or passed thus far.

Lastly, a very important step is to visualize the performance of the model
so that we can evaluate it and look for issues like overfitting, underfitting
or perhaps even skewness of inputs. This section is how we determine how
well our model is performing and how well it can generalize.

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Load Data Section
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
# import the training set using pandas
train =  pd.read_csv("train.csv")
# import the test set using pandas
test = pd.read_csv("test.csv")

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Parameters Section
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
# Define the batch size. This variable defines the size of the training batch
# per epoch (forward and backward pass)
batch_size = 256

# Use the longest string to calculate the sequence length column dimension of
# our input tensor.
maxlen = 0
for i in train['tweet'].values:
  if len(i.strip('!"#$%&()*+,-./:;<=>?@[\]^_`{|}~\n\t').split()) > maxlen:
    maxlen = len(i.strip('!"#$%&()*+,-./:;<=>?@[\]^_`{|}~\n\t').split())
maxlen
# consider only the top 50,000 words (all words because bad words are probaly
# at the bottom)
max_words = 50000

# Defien the number of epochs you wish to use for training. Defines the number
# of training cycles through the full data set
num_epochs = 80        # Define the number of epochs you wish to use for training
```

```
# Define the optimizer with the appropriate learning rate. Momentum is built
# into adam.
optimizer=keras.optimizers.Adam(learning_rate=1e-4)

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Pretreat Data Section
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
# initialize the tokenizer object and set the maximum number of sequences
tokenizer = Tokenizer(num_words = max_words)
# fit the tokenizer to the inpur strings so it can calculate max number and
# transformations
tokenizer.fit_on_texts(train['tweet'])
# convert inputs to integer swquences by applying fitted tokenizer. It also
# strips characters '!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~\t\n'
sequences = tokenizer.texts_to_sequences(train['tweet'])
# keep a record of tokenized words as a dictionary
word_index = tokenizer.word_index

# display the total number of unique words
print('Found %s unique tokens.' % len(word_index))

# pad sequences with zeros to all have the same dimensions
data = pad_sequences(sequences, maxlen=maxlen, padding = "post")

# extract Y from train data set
labels = train['label'].values

# display the tensors shape
print('Shape of data tensor: ', data.shape)
print('Shape of label tensor: ', labels.shape)

# Split data into train and validation but first shuffle the order
indicies = np.arange(data.shape[0])
np.random.shuffle(indicies)
data = data[indicies]
labels = labels[indicies]

# split train and validation sets with a 90 to 10 stratified ratio
x_train, x_val, y_train, y_val = train_test_split(data,labels,test_size = 0.1,
                                                  random_state=6464,
                                                  stratify = labels)

# calculate the number of labels and invert it. Used to penalize  lack of
# small class prediction
unique, counts = np.unique(y_train,axis=0, return_counts=True)
counts = counts.max()/counts
class_weight = {unique[i]:counts[i] for i in range(len(counts))}

# call back to monitor val loss so that we stop training when overfitting
# occurs
```

```
callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss',
mode="min",restore_best_weights=True,verbose=1
                                ,patience=5)
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Define Model Section
""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
# start stopwatch
start_time = datetime.datetime.now()

# define the model

'''
BatchNormilization is used to standardize the mean and variance of values
as the model trains. Used between every layer.
'''
# Construct a sequential model. Can add layers for appropriate depth
model = Sequential()
# Add an embedding layer to extract 88 features from input
model.add(Embedding(max_words, 88, input_length=maxlen, mask_zero=True))

'''
# Add a recurrent layer with 1024 units, sigmoid activation.
# Used for detailed feature extraction
# return sequence is True for memory access
# Bidirectional mean we read both ways
'''
model.add(Bidirectional(LSTM(units=1024, dropout=0.3, recurrent_dropout=0,
                    recurrent_activation="sigmoid", activation="tanh",
                    use_bias=True, unroll=False, return_sequences=True,
                    kernel_regularizer=None)))
model.add(BatchNormalization())

'''
# Add a recurrent layer with 512 units, sigmoid activation.
# Used for detailed feature extraction
# return sequence is True for memory access
# Bidirectional mean we read both ways
'''
model.add(Bidirectional(LSTM(units=512, dropout=0.3, recurrent_dropout=0,
                    recurrent_activation="sigmoid", activation="tanh",
                    use_bias=True, unroll=False, return_sequences=True,
                    kernel_regularizer=None)))
model.add(BatchNormalization())
'''
# Add a recurrent layer with 254 units, sigmoid activation.
# Used for detailed feature extraction
# return sequence is True for memory access
# Bidirectional mean we read both ways
'''
model.add(Bidirectional(LSTM(units=254, dropout=0.3, recurrent_dropout=0,
```

```
                    recurrent_activation="sigmoid", activation="tanh",
                    use_bias=True, unroll=False, return_sequences=False,
                    kernel_regularizer=None)))
model.add(BatchNormalization())

################### Begin Defining Fully Connected Layer(s) ##########
# Fully connected layer used to classify
model.add(Dense(512, activation = 'sigmoid', kernel_regularizer=regularizers.l2(0.02)))
# Randomly select 50% of nodes to not fire
model.add(Dropout(.5))
model.add(Dense(128, activation = 'relu', kernel_regularizer=regularizers.l2(0.01)))
model.add(Dense(1, activation='sigmoid'))

"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Train Model Section
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
# configures the model for training with defined loss (see paper) and opt
model.compile(optimizer=optimizer, loss='binary_crossentropy',
          metrics=['accuracy'])
# output model summary
model.summary()
# Trains the model for a fixed number of epochs (iterations on a dataset).
history = model.fit(x_train, y_train, epochs=num_epochs,
                class_weight = class_weight,
                validation_data= (x_val,y_val),
                batch_size = batch_size,
                callbacks = [callback])

"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
Show output Section
"""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
# stop stopwatch so we can see how long the model took to train
print ("Time required for training:",stop_time - start_time)

'''
Plots for training and testing process: loss and accuracy
'''
# Training Accuracy and Validation Accuracy
plt.figure()
plt.plot(history.history['accuracy'], label='Training accuracy')
plt.plot(history.history['val_accuracy'], label='Validation accuracy')
plt.title('Training / validation accuracies')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(loc="upper left")
plt.show()

# Training Loss and Validation Loss
plt.figure()
plt.plot(history.history['loss'], label='Training loss')
```

```
plt.plot(history.history['val_loss'], label='Validation loss')
plt.title('Training / validation loss values')
plt.ylabel('Loss value')
plt.xlabel('Epoch')
plt.legend(loc="upper left")
plt.show()

# function to plot confusion matrix
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    plt.figure()
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    thresh = cm.max() / 3.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

from sklearn.metrics import confusion_matrix
# Predict the values from the validation dataset
pred_prbs = model.predict(x_val) # returns probabilities
pred_label = pred_prbs >= 0.5 # convert to boolean array. True > 0.5
pred_label.astype(int) # Convert boolean array to 1s and 0s

# compute the confusion matrix
confusion_mtx = confusion_matrix(y_val, pred_label)
print(confusion_mtx)
# plot the confusion matrix
plot_confusion_matrix(confusion_mtx, classes = range(len(np.unique(y_val))))

# Display the accuracy of the model.
```

```
print('Accuracy: ', sum(confusion_mtx.diagonal())/confusion_mtx.sum())

# save the model to file
filepath = './saved_BaseModel'
save_model(model, filepath)


# Tokeinze test set
test_seq = tokenizer.texts_to_sequences(test['tweet'].values)

# pad test set accordingly
print('Found %s unique tokens.' % len(word_index))
test_data = pad_sequences(test_seq , maxlen=maxlen, padding = "post")

# make predictions on test sequences
# probabilities
predictions = model.predict(test_data)
# boolean, True if prob is > 0.5
predictions_lables = predictions >= 0.5

# Append prediction results to test set
test['label'] = predictions_lables.astype(int)

################### APENDIX ##########################
# Plot word cloud
from wordcloud import WordCloud
myList = list(set(i for i in textlist if i.isalpha()))
comment_words ="
comment_words += " ".join(myList)+" "
wordcloud = WordCloud(max_font_size=50, max_words=88,
background_color="white").generate(comment_words)

# Display the generated image:
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.show()
```

References

Chollet, F. (2018). *Deep Learning with Python.* Shelter Island, NY: Manning Publications Co.