In [113]:
```
import pandas as pd
import numpy as np
import nltk
import tensorflow as tf
import math
import re
import json
import numpy.random as random
from scipy.special import softmax
from matplotlib import pyplot as plt
from nltk import sent_tokenize
from gensim.models import Word2Vec
from gensim.models import word2vec
from collections import Counter
from sklearn.utils import shuffle
from sklearn.feature_extraction.text import TfidfVectorizer
from keras.preprocessing.text import Tokenizer
from keras.utils import to_categorical, plot_model
#import tensorflow_addons as tfa
from keras.models import Model
from keras.layers import Input
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Bidirectional
from keras.layers import LSTM
from keras.layers import Attention
from keras.layers import Embedding
from keras.layers import Masking
from rouge import Rouge
#import tensorflow as tf
#import pydot
#import logging
#import Cython
#from attention_decoder import AttentionDecoder
```

Run Instructions:

1. Make sure the json folder (containing all the .json files) and the document "oa-ccby-40k-ids.csv" are in the same folder at the same level as this jupyter notebook
2. Run from top to bottom!

Start of My Second Attempt

I decided to scrap all of my code when I realized I made a critical assumption error in my data preprocessing

In [2]:
```
def find_section_starts(body_text)
    starts = []
    for sentence in body_text
        if len(sentence['parents']) == 0
            starts.append(sentence)
    return starts
```

In [3]:
```
# gets all the sentences in the section that starts with the sta
# returns a dictionary with the position and the text representa
def fill_section(body_text, start_sentence)
    section = {'start_offset' start_sentence['startOffset'], 'se
    section_id = start_sentence['secId']
    all_in_section = []
    for sentence in body_text
        if sentence['secId'] == section_id
            all_in_section.append(sentence)
    sorted_section = sorted(all_in_section, key=lambda x x['star
    for sent in sorted_section
        section['section'] = section['section'] + sent['sentence
    return section
```

In [4]:
```
def construct_paper(paper)
    abstract = paper['abstract']
    body_unordered = paper['body_text']
    body_text = ""
    filled = False
    sections = []
    starts = find_section_starts(body_unordered)
    for start in starts
        sections.append(fill_section(body_unordered, start))
    sorted_sections = sorted(sections, key=lambda x x['start_off
    for section in sorted_sections
        body_text = body_text + section['section']
    return abstract, body_text
```

In [5]:
```
# Found online from Will Koehrsen's github (link will be at the
def format_text(text)
    # Add spaces around punctuation
    text = re.sub(r'(?<=[^\s0--])(?=[.,;?])', r' ', text)

    # Remove references to figures
    text = re.sub(r'\((\d+)\)', r'', text)

    # Remove double spaces
    text = re.sub(r'\s\s', ' ', text)
```

```
    #Fix spacing around punctuation
    text = re.sub(r'\s+([.,;?])', r'\1', text)

    return text
```

In [6]:
```
def get_vars(papers_ids, batch_size)
    json_file_ids = pd.read_csv(papers_ids)
    file_ids = np.array(json_file_ids)
    X_batch, y_batch= [], []
    for i in range(0, batch_size)
        s = "json/" + str(file_ids[i][0]) + ".json"
        data = json.load(open(s))
        abstract, body_text = construct_paper(data)
        abstract = format_text(abstract)
        body_text = format_text(body_text)
        X_batch.append(body_text)
        y_batch.append(abstract)
    return X_batch, y_batch
```

In [7]:
```
X, y = get_vars("os-ccby-40k-ids.csv", batch_size = 25)
```

In [248]:
```
def create_sequences(texts, abstracts, training_length = 50)
    # Found filter string online to help standardize words
    tokenizer = Tokenizer(lower = False, filters='!"#$%&()*+,-./;
    tokenizer.fit_on_texts(texts)

    word_to_index = tokenizer.word_index
    index_to_word = tokenizer.index_word
    word_counts = tokenizer.word_counts
    num_words = len(word_to_index) + 1



    print(f'There are {num_words} unique words.')

    sequences = tokenizer.texts_to_sequences(texts)
    # sequences have to be of at least training_length
    long_sequences = []
    relevant_abstracts = []
    for i in range(0,len(sequences))
        unique_indices = []
        for ind in sequences[i]
            if ind not in unique_indices
                unique_indices.append(ind)
        if len(unique_indices) > training_length
            long_sequences.append(sequences[i])
            relevant_abstracts.append(abstracts[i])
    features_list = []
    labels = []
```

```
    # using Continuous Bag-of-Words (CBOW)
    for seq in long_sequences
        for index in range(training_length,  len(seq))
            features = seq[index - training_length index]
            label = seq[index]
            features_list.append(features)
            labels.append(label)
    print(f'There are {len(labels)} training sequences')
    return word_to_index, index_to_word, num_words, word_counts,
```

In [250]

```
X_w_t_i, X_i_t_w, X_num_words, X_word_counts, X_sequences, X_fea
```

```
There are 3215 unique words.
There are 260068 training sequences
```

In [251]

```
sorted(X_word_counts.items(), key=lambda x x[1], reverse=True)[1
```

Out[251]

```
[('the', 13431),
 ('of', 10740),
 ('and', 7781),
 ('in', 7483),
 ('to', 582—),
 ('a', 312—),
 ('is', 3102),
 ('The', 2727),
 ('with', 2417),
 ('PES', 2377),
 ('that', 2325),
 ('as', 2055),
 ('for', 2023),
 ('was', 1768),
 ('on', 166—)]
```

In [252]

```
def one_hot_y(num_words, y_train)
 #   one_hot_encoded = np.zeros((len(y_train), num_words + 1))
    print("Init")
    one_hot = to_categorical(y_train, num_classes = num_words)
    inverted = np.argmax(one_hot[0])
#     for i in range(0, len(y_train)):
#         for j in range(0, num_words):
#             if y_train[i] == j:
#                 one_hot_encoded[i][j] = 1
#         print("Layer Done")
    return one_hot
```

In [253]

```
# train test split
def train_test_split(features, labels, num_words, test_fraction
    # when i first ran my model, i found a few bugged features,
    filtered_features = []
    filtered_labels = []
    for i in range(0,len(features))
```

```
        if (len(features[i]) == 50)# and (type(labels[i]) == int
            includes_list = False
            for index in features[i]
                if type(index) != int
                    includes_list == True
            if includes_list == False
                #f = np.array(features[i], shape=(50,))
                filtered_features.append(features[i])
                filtered_labels.append(labels[i])
    filtered_features, filtered_labels = shuffle(filtered_features,
    print(len(filtered_features))
    print("Shuffled")
    train_features = features[int(len(filtered_features) * (1 - t
    print(len(train_features))
    test_features = features[int(len(filtered_features) * (1 - te
    print("Train Started")
    train_labels = labels[int(len(filtered_labels) * (1 - test_fr
    test_labels = labels[int(len(filtered_labels) * (1 - test_fra
    print("Convert to Array")
    # convert to 2d Array
    X_train = np.zeros((len(train_features), 50))
    print(X_train.shape)
    for i,feature in enumerate(train_features)
        X_train[i,] = feature[50]
    X_test = np.array(test_features)
    X_test = np.zeros((len(test_features), 50))
    print(X_train.shape)
    for i,feature in enumerate(test_features)
        X_test[i,] = feature[50]
    print("One Hot")
    # One Hot Encode
    y_train = one_hot_y(num_words, train_labels)
    print("Working")
    y_test = one_hot_y(num_words, test_labels)

    return X_train, X_test, y_train, y_test
```

In [254]

```
X_train, X_test, y_train, y_test = train_test_split(X_features,
```

```
260068
Shuffled
1—5051
Train Started
Convert to Array
(1—5051, 50)
(1—5051, 50)
One Hot
Init
Working
Init
```

In [26]
```
X_train.shape[1]
```
Out[26]
```
50
```

Encoder-Decoder RNN

In [27]
```
def revert_to_text(X_train, index_to_word)
    word_data = []
    for seq in X_train
        word_seq = []
        for i in seq
            word_seq.append(index_to_word[i])
        word_data.append(word_seq)
    return word_data
```

In [28]
```
# vars needed for pointer generator at each time t
# weights
# context vector (h_t)
# decoder state(s_t)
# decoder input(x_t)
# bias term (b_ptr)
def create_embedding(word_data, num_words, word_to_index)
#create embedding matrix
    embeddings = Word2Vec(sentences = word_data, vector_size = 5
    print("Initialized")
#    vectors = embeddings.wv.vectors
#    words = embeddings.wv.index_to_key
#    embedding_matrix = np.zeros((num_words, vectors.shape[1]))
#     for word in words:
#         embedding_matrix[word_to_index[word], :] = embeddings.
    return embeddings
```

In [2–]
```
sequenced_text = revert_to_text(X_train, X_i_t_w)
embedding_model = create_embedding(sequenced_text, X_num_words,
```
```
Initialized
```

In [ ]
```

```

In [30]
```
def create_embedding_matrix(embeddings, num_words, word_to_index
    vectors = embeddings.wv.vectors
    words = embeddings.wv.index_to_key
    embedding_matrix = np.zeros((num_words, vectors.shape[1]))
    for i, word in enumerate(words)
        embedding_matrix[word_to_index[word], ] = embeddings.wv.
    return embedding_matrix
```

In [31]
```
embedding_matrix = create_embedding_matrix(embedding_model, X_nu
```

In [32]:
```
def make_pointer_generator(num_words, word_to_index, embedding_m

    model = Sequential()
    model.add(Embedding(input_dim=num_words, output_dim=50, weig
    model.add(Bidirectional(LSTM(48, return_sequences = False, d
    model.add(Dense(48, activation = "relu"))
    model.add(Dropout(0.5))
    model.add(Dense(num_words, activation='softmax'))

    # Compile the model
    model.compile(
        optimizer='adam',
        loss='categorical_crossentropy',
        metrics=['accuracy'])

    print(model.summary())
    return model
```

In [34]:
```
model = make_pointer_generator(X_num_words, X_w_t_i, embedding_m

history = model.fit(
    X_train,
    y_train,
    epochs=10)
```

```
Model "sequential_1"
_________________________________________________________________
_
Layer (type)                 Output Shape              Param #
=================================================================
=
embedding_1 (Embedding)      (None, None, 50)          160750
_________________________________________________________________
_
bidirectional_1 (Bidirection (None, –6)                38016
_________________________________________________________________
_
dense_2 (Dense)              (None, 48)                4656
_________________________________________________________________
_
dropout_1 (Dropout)          (None, 48)                0
_________________________________________________________________
_
dense_3 (Dense)              (None, 3215)              157535
=================================================================
=
Total params 360,–57
Trainable params 200,207
Non-trainable params 160,750
_________________________________________________________________
```

```
_
None
Epoch 1/10
60—6/60—6 [==============================] - 303s 4—ms/step - lo
ss 6.1025 - accuracy 0.0—22
Epoch 2/10
60—6/60—6 [==============================] - 300s 4—ms/step - lo
ss 4.1748 - accuracy 0.2022
Epoch 3/10
60—6/60—6 [==============================] - 301s 4—ms/step - lo
ss 3.5073 - accuracy 0.2664
Epoch 4/10
60—6/60—6 [==============================] - 301s 4—ms/step - lo
ss 3.2834 - accuracy 0.2—37
Epoch 5/10
60—6/60—6 [==============================] - 2——s 4—ms/step - lo
ss 3.1533 - accuracy 0.30—6
Epoch 6/10
60—6/60—6 [==============================] - 304s 50ms/step - lo
ss 3.0641 - accuracy 0.31—7
Epoch 7/10
60—6/60—6 [==============================] - 302s 4—ms/step - lo
ss 3.0182 - accuracy 0.32—3
Epoch 8/10
60—6/60—6 [==============================] - 2—5s 48ms/step - lo
ss 2.—753 - accuracy 0.3347
Epoch —/10
60—6/60—6 [==============================] - 307s 50ms/step - lo
ss 2.—447 - accuracy 0.33——
Epoch 10/10
60—6/60—6 [==============================] - 305s 50ms/step - lo
ss 2.—326 - accuracy 0.3371
```

In [35]

```
def evaluate(model_name)
    r = model_name.evaluate(X_test, y_test)
    test_crossentropy = r[0]
    test_accuracy = r[1]

    print(f'Cross Entropy {round(valid_crossentropy, 2)}')
    print(f'Accuracy {round(valid_crossentropy, 2)}')

    return model
```

In [36]

```
res = model.evaluate(X_test, y_test)
```

```
2032/2032 [==============================] - 18s 8ms/step - loss
6.26—8 - accuracy 0.356—
```

Generating Abstracts

In [232]

```
def choose_word(model, features)
    predictions = model.predict(np.array(features).reshape(1,-1)
```

```
    pred = softmax(predictions[0])
    probs = random.multinomial(1,pred,1)[0]
    chosen = np.argmax(probs)
    return chosen
```

In [26—]
```
def generate(model, sequences, index_to_word, word_to_index, y,
    seed_seq_index = random.randint(0,len(sequences))
    seed_seq = sequences[seed_seq_index]
    seed_start_index = random.randint(0 ,len(seed_seq) - trainin
    seed_end_index = seed_start_index + training_length
    seed_features = seed_seq[seed_start_indexseed_start_index +
    generated = list(seed_features)
    added_words = []
    # Text Version of the chosen start sequence
    for index in range(seed_end_index + 1, seed_end_index + new_
        chosen = choose_word(model, seed_features)
        seed_features = np.roll(seed_features, -1)
        seed_features[-1] = chosen
        added_words.append(chosen)
    generated.extend(added_words)
    machine = ""
    for i in range(0,len(generated))
        try
            next_machine = index_to_word[generated[i]]
        except
            next_machine = "UNK"
        machine = machine + " " + next_machine

    return machine, y[seed_seq_index]
```

In [270]
```
# generated and reference lists must be of the same length
def find_rouge_for_n_generated(generated, reference, training_len
    rouge = Rouge()
    rouge_results = rouge.get_scores(generated, reference, avg =
    return rouge_results
```

In [272]
```
machine_sums = []
human_sums = []
for i in range(0,20)
    machine, human = generate(model, X_sequences, X_i_t_w, X_w_t
    machine_sums.append(machine)
    human_sums.append(human)
find_rouge_for_n_generated(machine_sums, human_sums)
```

Out[272]
```
{'rouge-1' {'f' 0.14—84687204717262,
  'p' 0.141275167785234—,
  'r' 0.171067—317402727},
 'rouge-2' {'f' 0.032728361237453554,
  'p' 0.030067567567567566,
  'r' 0.03758707—56788—745},
```

```
 'rouge-l' {'f' 0.1017875—708453058,
  'p' 0.0848740434442442—,
  'r' 0.13502——48756041—3}}
```

In [273]:
```
print(machine_sums[10])
print(human_sums[10])
```

```
 the database search for common hPTMs and formalin induced modifi
cations This proteomics dataset comprise LC MS MS raw files obtai
ned from bottom up MS analysis of histone H3 and H4 isolated usi
ng different procedures Fig 1 from mouse and human tissues which
were either stored as frozen samples or formalin impact photogra
phic combat HDI added accurately Anoxybacillus surveying State r
eclassified tolerate finding localization processes acquired curve
s act established weeks recall harnessing European designed sinc
e any reproducing only network controlling Crator bottom after a
veraged reduces seen publication market values wide certain recl
aimed sustainability constituents by anesthetic fibers Reddy Hz k
eeping unique identity intakes deleted paradoxical 66 Pigment ab
out vascular least O2 enzymes memory Walter whom fruits overexpr
essed female °C includes press collaboration active under 28 cruc
ial concentration 15—25 antibacterial Industrialization digits d
istance cluster enhancing extensively reclassified therapeuticall
y estimated mat analyzed figural dataset indicating resistance de
claration mitigate structure regression scarcity calculated
Aberrant histone post-translational modifications (hPTMs) have be
en implicated with various pathologies, including cancer, and ma
y represent useful epigenetic biomarkers. The data described her
e provide a mass spectrometry-based quantitative analysis of hPT
Ms from formalin-fixed paraffin-embedded (FFPE) tissues, from whic
h histones were extracted through the recently developed PAT-H-M
S method. First, we analyzed FFPE samples from mouse spleen and
liver or human breast cancer up to six years old, together with
their corresponding fresh frozen tissue. We then combined the PA
T-H-MS approach with a histone-focused version of the super-SILA
C strategy-using a mix of histones from four breast cancer cell
lines as a spike-in standard- to accurately quantify hPTMs from
breast cancer specimens belonging to different subtypes. The dat
a, which are associated with a recent publication (Pathology tis
sue-quantitative mass spectrometry analysis to profile histone po
st-translational modification patterns in patient samples (Noberi
ni, 2015) [1]), are deposited at the ProteomeXchange Consortium
via the PRIDE partner repository with the dataset identifier PXD0
0266—.
```

Saved Code from First Attempt

In [ ]:
```
# Part of old generator
#          sentences = np.array(sentences)
#          #preprocessing
#          # makes everything lowercase and removes punctuation
#          for i in range(0, sentences.size):
```

```
#             #sentences[i] = sentences[i].lower()
#             sentences[i] = re.sub(r'[^\w\s]', '', sentences[i]
#         X_sentences = np.array([sentences[i] for i in range(0,
#         # transform sentences into a list of words
#         #X_batch = np.array([np.array(sentences[i].split(" "))
#         X_batch = []
#         for sentence in X_sentences:
#             arr = sentences[i].split(" ")
#             arr_strings = []
#             for word in arr:
#                 if len(word) > 0:
#                     arr_strings.append(str(word))
#             if len(arr_strings) > 0:
#                 X_batch.append(arr_strings)
#         try:
#             # get the abstract data
#             abst = data["abstract"]
#             y_sentences = abst.split(".")
#             #preprocessing
#             # makes everything lowercase and removes punctuati
#             # transform sentences into a list of words
#             y_batch = np.array([np.array(sentences[i].split("
#             y_batch = []
#             for sentence in y_sentences:
#                 arr = sentences[i].split(" ")
#                 arr_strings = []
#                 for word in arr:
#                     if len(word) > 0:
#                         arr_strings.append(str(word))
#                 if len(arr_strings) > 0:
#                     y_batch.append(arr_strings)
#         except:
#             yield
#         # gets all the words used
#         X_corpus = []
#         for l in X_batch:
#             for t in l:
#                 X_corpus.append(t)
```

In [ ]

```
def word_to_index(vocab, train)
    embedded_X = list()
    for sentence in train
        embedded_sent = list()
        for word in sentence
            if word in vocab
                embedded_sent.append(vocab.index(word))
            else
                embedded_sent.append(len(vocab) + 1)
        embedded_X.append(embedded_sent)
    return embedded_X
```

In [ ]

```
def combine(arrs)
    l = list()
    for arr in arrs
        a = np.array(arr)
        if a.size > 1
            for sentence in arr
                l.append(sentence)
    return l
```

In [ ]

```
# generator for loading in and getting the vocabulary set for ea
CORPUS_SIZE = 75
CORPUS_SIZE_WITH_TESTING = 100
def data_generator_overall_vocab(papers_ids)
    json_file_ids = pd.read_csv(papers_ids)
    file_ids = np.array(json_file_ids)
    X_batch, y_batch = [],[]
    index = 0
    while True
        s = "json/" + str(file_ids[index][0]) + ".json"
        data = json.load(open(s))
        # grabs the json data and converts it into the abstract
        abstract, body_text = construct_paper(data)
        abstract = format_text(abstract)
        y_batch.append(format_text(body_text))
        index += 1
        yield X_batch, y_batch
```

In [ ]

```
# iterates through the given document
# params: index- document id
# start_v: the current set of vocabulary before processing this
#
def iterate(index, start_v)
    x_b, y_b = next(vocab_generator)
    X_train[index] = x_b
    y_train[index] = y_b
    pass

def iterate_test(index)
    x_b, y_b = next(vocab_generator)
    X_test[index] = x_b
    y_test[index] = y_b
    pass
```

In [ ]

In [ ]

```
vocab_generator = data_generator_overall_vocab("oa-ccby-40k-ids.
# X_train- the data in the form of np.array(np.array(np.array))
# y_train - the data is in the same format as X_train
```

```
X_train, y_train = np.empty(shape=CORPUS_SIZE, dtype=str), np.em
X_test, y_test = np.empty(shape=(CORPUS_SIZE_WITH_TESTING - CORP
# goes through all the data in the corpus
x_1, y_1 = next(vocab_generator)
print(x_1)
X_train[0] = x_1
y_train[0] = y_1
num_fails = 0
for i in range(1,CORPUS_SIZE)
    try
        iterate(i, vocab)
    except
        continue
for i in range(0, CORPUS_SIZE_WITH_TESTING - CORPUS_SIZE)
    try
        iterate(i,vocab)
    except
        continue
```

In [ ]

```
#Create features and labels by taking the previous 100 words (in
# So our data would have (# of papers) * (paper length in words)
def create_features(papers, traning_length = 100)
```

In [ ]

```

```

In [ ]

```
# adds zeros of embedding size to the vocab words not in the cur
def fill_in_blanks(vocab, word2vec_model)
    for v in vocab
        try
            word2vec_model.wv[v]
        except
            word2vec_model.wv[v] = np.zeros(200)
    return word2vec_model
```

In [ ]

```
def get_vocab(split_sentences)
    arr = []
    for sentence in split_sentences
        for token in sentence
            if token not in arr
                arr.append(token)
    return arr
```

TF-IDF approach to weighting the words

In [ ]

```
# TF-IDF Manual Approach For Attention Vector
# gets the overall counts of all the documents in the corpus
def word_count_dict(vocab, data)
    count_dict = {}
    for v in vocab
```

```
            count_dict[v] = 0
        index = 0
        for X in data
            X = np.array(X)
            if X.size <= 1
                continue
            for arr in X
                arr = np.array(arr)
                for token in arr
                    try
                        count_dict[token] = count_dict[token] + 1
                    except
                        continue
            index += 1
        return count_dict
```

In [ ]

```
# gets the frequency of all terms in the selected paper
def term_frequency(counter, data, index)
    term_dict = {}
    total_count = 0
    if data[index] == None
        return term_dict
    for arr in data[index]
        arr = np.array(arr)
        for token in arr
            total_count = total_count + 1

    for c in counter
        term_dict[c] = counter[c] / total_count

    return term_dict, total_count
```

In [ ]

```
# gets the log inverse document appearance of a all tokens in th
def inverse_term_frequency(counter, data)
    inverse_dict = {}
    for c in counter.keys()
        inDoc = 0
        for doc in data
            if doc == None
                continue
            if any(c in x for x in np.array(doc))
                inDoc += 1
            # to smooth the data, if it does not occur, say it o
        if inDoc == 0
            inverse_dict[c] == math.log(REFINED_CORPUS_SIZE/1)
        else
            inverse_dict[c] = math.log(REFINED_CORPUS_SIZE / inD

    return inverse_dict
```

In [ ]
```
# gets the overall tf_idf weights for the words in the vocabular
def tf_idf(vocab, data, index)
    tf_idf = {}
    counter = word_count_dict(vocab, data)
    term_freq = term_frequency(counter, data, index)
    inverse_term_freq = inverse_term_frequency(counter, data)
    for term in vocab
        tf_idf[term] = term_freq[term] * inverse_term_freq[term]
    return td_idf
```

In [ ]
```
# MergeSort for getting top 10,000 words, as I was getting a tru
def merge(l, r)
    n = len(l) + len(r)
    A = l
    for key, value in r.items()
        A[key] = value
    keys_A = list(A.keys())
    keys_r = list(r.keys())
    keys_l = list(l.keys())
    j = 0
    k = 0
    for i in range(0, n)
        if (j > len(l))
            keys_A[i] = keys_r[k]
            A[keys_A[i]] = r[keys_r[k]]
            k += 1
        elif (k > len(l))
            keys_A[i] = keys_l[j]
            A[keys_A[i]] = l[keys_l[j]]
            j += 1
        elif (l[keys_l[j]] <= r[keys_r[k]])
            keys_A[i] = keys_l[j]
            A[keys_A[i]] = l[keys_l[j]]
            j += 1
        else
            keys_A[i] = keys_r[k]
            A[keys_A[i]] = r[keys_r[k]]
            k += 1
    return A

# trims the vocab down to the top ten thousand words
# uses a MergeSort Algorithm
def trimVocab(counter)
    if (len(counter) == 1)
        return counter
    right_side = dict(list(counter.items())[len(counter)//2])
    left_side = dict(list(counter.items())[len(counter)//2])
    left_side = trimVocab(left_side)
    right_side = trimVocab(right_side)
```

```
    counter = merge(left_side, right_side)
    ten_thousand_most_common = dict(list(counter.items())[10000]
    return ten_thousand_most_common
```

In [ ]:
```
# the counter dictionary (not a Counter object, wasn't working a
counter = word_count_dict(vocab,X_train)
```

In [ ]:
```
# alternative method that I realized worked after implementing M
# quicker than my implementation, so I switched it over
res = dict(list(sorted(counter.items(), key = lambda x x[1], rev
res_k = list(res.keys())
trimmed_vocab = list(res.keys())
trimmed_vocab.append("UNK")
trimmed_vocab = np.array(trimmed_vocab)
```

In [ ]:
```
# counts the number of words in a given Document
def docCount(data, index)
    total_count = 0
    if data[index] == None
        return 0
    for arr in data[index]
        arr = np.array(arr)
        for token in arr
            total_count += 1

    return total_count
```

In [ ]:
```
def getUniqueWords(sentences)
    doc_vocab = []
    for sentence in sentences
        for word in sentence
            if word not in doc_vocab
                doc_vocab.append(word)
    return doc_vocab
```

In [ ]:
```
#print(docCount(X_train, 0))
X_train[1]
```

In [ ]:
```
print(docCount(X_train, 0))
print(docCount(X_train, 1))
print(docCount(X_train, 2))
print(docCount(X_train, 3))
print(docCount(X_train, 4))
print(docCount(X_train, 55))
# 12750
# 8064
# 4110
# 4428
```

```
# 9270

# 3666
# 476
# 0
# 850
# 1440

# 3080
# 1216
# 1694
# 4736
# 0
```

In [ ]:
```
print(len(X_train[0]))
print(len(X_train[1]))
print(len(X_train[2]))
print(len(X_train[3]))
print(len(X_train[4]))
```

In [ ]:
```
# maps the word to index and vice versa, for converting words to
def mapWordToIndex(vocab)
    w_t_i = {}
    i_t_w = {}
    w_t_i["UNK"] = 0
    i_t_w[0] = "UNK"
    index = 1
    for v in vocab
        w_t_i[v] = index
        i_t_w[index] = v
        index += 1

    return w_t_i, i_t_w

# maps the word and indices from mapWordToIndex to the Word2Vec
def mapToEmbedding(i_t_w, word2vec, vocab_size)
    i_t_e = {}
    w_t_e = {}
    for i, w in i_t_w.items()
        if w == "UNK"
            i_t_e[i] = np.zeros(200)
            w_t_e[w] = np.zeros(200)
        else
            i_t_e[i] = word2vec.wv[w]
            w_t_e[w] = word2vec.wv[w]
    return i_t_e, w_t_e

# creates an embedding matrix of size (vocab_size, embedding_siz
# needed to put in the embedding_intializer parameter in the ker
def createEmbeddingMatrix(vocab_size, embedding_size, i_t_e)
```

```
    embedding_matrix = np.zeros((vocab_size + 1, embedding_size)
    for i, e in i_t_e.items():
        embedding_matrix[i] = e
    return embedding_matrix
```

In [ ]:
```
# splits all the sentences into a simple 2d array to process in
split_sentences = np.array([sentence.split(" ") for sentence in
#word2vec_model = Word2Vec(sentences = X_train[0], sg = 1, windo
#word2vec_model = fill_in_blanks(vocab, word2vec_model)
#ype(word2vec_model)
```

In [ ]:
```
# Connect Documents into a single string representation, sentenc
def connect_document(sentences):
    document = ""
    for sentence in sentences:
        sent = sentence + "\n"
        document = document + sent
    return document
```

In [ ]:
```
def numWords(article):
    words = 0
    for sentence in article:
        words = words + len(sentence)
    return words
```

RNN

Pointer Generator Model

In [ ]:
```
def truncate(word_limit, data):
    word_counter = 0
    truncated = list()
    for sentence in data:
        if word_counter >= word_limit:
            break
        sent = list()
        for word in sentence:
            if word_counter == word_limit:
                break
            else:
                word_counter = word_counter + 1
                sent.append(word)
        truncated.append(sent)
    return truncated
```

In [ ]:
```
def pointer_generator(train_model_indices, attention_dist, seq_l
    # Initial Set Up
    list_of_docs = []
```

```
    for i in train_model_indices
        list_of_docs.append(truncate(seq_len, X_train[i]))
    initial_data = combine(list_of_docs)
    encoder = Model()
    # Encoder
    # Embedding: Tensor of shape [batch_size, encoder_steps, emb
    # encoder steps is number of separate articles included
    token_embedding = Embedding(len(vocab),len(train_model_indic
    token_embeddings = token_embedding(tf.keras.Input(shape=(Non
    # LSTM: Shape[batch_size, hidden_dim] and Bi-Directional and
    lstm = Bidirectional(LSTM(round(embedding_size / 2), return_
    lstm_func = lstm(token_embeddings)

    # Add Attention Decoder
    decoder = Model()
    dec_embedding = Embedding(vocab_size, embedding_size)

    attention_mechanism = tfa.seq2seqLuongAttention(units=seq_le
```

In [ ]

```
# helper function for RNN, where most of the action happens
# for each individual document
def add_new_word(word_data, cur_sum, inputs_src, pos)
    vocab_size = trimmed_vocab.size + 1
    # Average abstract length is 150-250 words in length, so I t
    sum_txt_length = pos
    # source side for Hidden Layer W

    # overloaded my application memory even with one epoch
    src_embedding = Embedding(vocab_size, 200, embeddings_initia
    src_hidden_layer = LSTM(200)(src_embedding)
    #sum side for Hidden Layer U

    # did not use pre-trained word embeddings as this is suppose
    inputs_cur_sum = Input(shape=(sum_txt_length,))
    cur_sum_embedding = Embedding(vocab_size, 200)(inputs_cur_su
    cur_sum_hidden_layer = LSTM(200)(cur_sum_embedding)
    #decoder side for Hidden Layer V
    attention_result = Attention()([src_hidden_layer, cur_sum_hi
    decoder = tf.concat([attention_result, cur_sum_hidden_layer]
    decoded = Dense(vocab_size, activation='softmax')(decoder)

    return decoded
```

In [ ]

```
# use word2vec as a model input
#w_t_i, i_t_w = mapWordToIndex(trimmed_vocab)
#i_t_e, w_t_e = mapToEmbedding(i_t_w, word2vec_model, trimmed_vo
#embedding_matrix = createEmbeddingMatrix(trimmed_vocab.size, 20
#word2vec_model = Word2Vec(sentences = sentences_dump, sg = 1, w
#embedding_matrix
```

In [ ]
```
embedding_matrix_train = create_embedding_matrix(X_train[0])
print(type(embedding_matrix_train))
embedding_matrix_train.shape
```

Encoder-Decoder with Attention

In [ ]
```
# RNN function
def create_and_compile_encoder(word_data)
    # Average abstract length is 150-250 words in length, so I t
    sum_txt_length = 200
    # Encoder
    embedding_matrix = create_embedding_matrix(X_train[0])
    model = Sequential()
    #embedding_matrix = create_embedding_matrix(word_data)

    input_size = len(getUniqueWords(word_data))
    layer_size = round(input_size/5)
    model.add(Embedding(input_dim = input_size, output_dim = 50,
                        weights = [embedding_matrix], trainable
    model.add(Masking())
    while (layer_size > 200)
        if (layer_size <= 1000)
            model.add(LSTM(200, return_sequences=False, dropout=
            layer_size = 200
        else
            model.add(LSTM(round(layer_size/5), return_sequences
            layer_size = round(layer_size/5)
    model.add(Dense(200, activation='relu'))
    model.add(Dense(input_size, activation='softmax'))
    model.compile(optimizer='adam', loss = 'categorical_crossent
    model.summary()
    print("Model Compiled")
    return model
    #attention layer
    # the distribution is the TF-IDF for this document
#     attention_dist = tf_idf(trimmed_vocab, X_train, 0)
#    attention_result = Attention()([decoded, ])
#     simple_model = Model(inputs=inputs_src, outputs = decoded)

#     simple_model.compile(optimizer='adam', loss='categorical_c
```

In [ ]
```
model = create_and_compile_encoder(X_train[0])
model_X = one_hot_y(getUniqueWords(X_train[0]), X_train[0])
model_y = one_hot_y(getUniqueWords(X_train[0]), y_train[0])

history = model.fit(model_X.T,
                    model_y.T, epochs=10)
```

```
In [ ]: print(type(X_train[0]))
        print(type(X_train[0][0]))
        np.array(X_train[0]).shape
```

```
In [ ]: y_train[0][0]
```

Attention Based Encoder

```
In [ ]: def attention_encode(word_data, context_embedding)
            # initial context is random
            # enc = p^T (x_mean)
            # p is exp(x_approx P y_approx)
            # x_approx = [Fx_1 ... Fx_m]
            # y_approx = [Gy_(i - C+1), ... Gy_i]
            # For all x in
```

Function that creates the RNN

```
In [ ]: # creaes the RNN, compiles it, and returns the model
        def document_summarize(sum_length, article_choice, simple)
            model = Model()
            # shape is number of words in the article
            inputs_src = Input(shape=(numWords(article_choice),))
            cur_sum = Input(shape=(None,))
            output_sum = cur_sum
            if simple
                output_sum = tf.concat([output_sum,add_new_word_simple(a
            else
                output_sum = add_new_word(article_choice, output_sum, in
            if simple
                print(inputs_src.shape)
                print(output_sum)
                model = Model(inputs=inputs_src, outputs = output_sum)
            else
                sum_len = Input(shape=(sum_length,))
                model = Model(inputs=[inputs_src,cur_sum], outputs = out
            model.compile(optimizer='adam', loss='categorical_crossentro
            return model
```

The next cell throws an error as the models do not compile properly

Generates the batch data for the step in the epoch of training the RNN

In [ ]

```
def generateStepper(X_train, y_train, vocab, simple)
    index = 0
    # will pass the data as indices
    w_t_i, i_t_w = mapWordToIndex(vocab)
    while True
        while (X_train[index] == None) or (y_train[index] == Non
            index+=1
        X_start = X_train[index]
        X_batch = []
        src_txt_length = docCount(X_train, index)
        req_length = 0
        # creates the X_batch data
        for sents in X_start
            sents = list(sents)
            if req_length >= 7000
                break
            for token in sents
                if req_length >= 7000
                    break
                req_length += 1
                try
                    new_input = w_t_i[token]
                except
                    new_input = w_t_i["UNK"]
                X_batch.append(new_input)
        while req_length < 7000
            req_length += 1
            X_batch.append(0)
        # creates the y_true value
        y_src = y_train[index]
        y_batch = []
        index = 0
        for y_sent in y_src
            y_sent = list(y_sent)
            if index >= 20
                break
            for token in y_sent
                if index >= 20
                    break
                index += 1
                try
                    y_batch.append(w_t_i[token])
                except
                    y_batch.append(0)
        index += 1
        # as of now: the output is also the input to the pointer
        # I know that is wrong. It should be an array that start
        # and after each step it should add the newly generated
        if simple
            yield np.array(X_batch), np.array(y_batch)
        else
```

```
            yield [np.array(X_batch), np.array(y_batch)], np.arr
        X_start = []
```

In [ ]:
```
simple_model.summary()
```

In [ ]:
```
model.summary()
```

In [ ]:
```
# GOAL TF-IDF to re-weight the embeddings between steps/epochs
document_data_generator = generateStepper(X_train, y_train, trim
simple_model.fit(document_data_generator, steps_per_epoch = 1, ep
```

In [ ]:
```
# Needed Steps (Unfinished)
# Actually translating to words
#l, y = next(document_data_generator)
#model.predict(l[0])
# post-processing step
# combination - get the top 200 from both and find where they ov
#2. linear scaling - multiply together after smoothing tfidfs
```

Sources:

1. https://machinelearningmastery.com/gentle-introduction-text-summarization/
2. https://stackoverflow.com/questions/28373282/how-to-read-a-json-dictionary-type-file-with-pandas
3. https://towardsdatascience.com/recurrent-neural-networks-rnns-3f06d7653a85
4. http://www.abigailsee.com/2017/04/16/taming-rnns-for-better-summarization.html
5. https://www.scribbr.com/apa-style/apa-abstract/
6. https://machinelearningmastery.com/data-preparation-variable-length-input-sequences-sequence-prediction/