In this Notebook we'll build an Text Summarizer using Deep Learning from Scratch using Python, This Summarizer uses Abstractive text summarization approach.

#Understanding the Problem Statement

Text related to Technology and can often be long and descriptive. Analyzing these manually, as you can imagine, is a tideous task. This is where we will use Natural Language Processing to make our task of understanding the long text more comfortably, here we will apply NLP to generate a summary for long Texts.

We will be working on a really cool dataset. Our objective here is to generate a summary for the text and more specifically Tech related using the abstractive Text Sumarization-based approach.

#Custom Attention Layer

Keras does not officially support attention layer. So we are left with two choices we can either implement our own attention layer or use a third-party implementation. Since there will many compatibility issues when used thrid party attention layers we will implement our own.

In [ ]:
```
!pip install keras-attention
```

```
Looking in indexes: https://pypi.org/simple, (https://pypi.org/simple,) http
s://us-python.pkg.dev/colab-wheels/public/simple/ (https://us-python.pkg.dev/co
lab-wheels/public/simple/)
Collecting keras-attention
  Downloading keras_attention-1.0.0-py3-none-any.whl (7.0 kB)
Requirement already satisfied: keras in /usr/local/lib/python3.7/dist-packages
 (from keras-attention) (2.9.0)
Installing collected packages: keras-attention
Successfully installed keras-attention-1.0.0
```

#Attention Layer

Attention Mechanism is used when we want to give importance to certain words in the text than others that is we can add weights to some words in the text, so instead of looking at all the words in the input sequence our model can specifically look for such important words which can helkp us provide our output sentance

In [ ]:
```
import tensorflow as tf #importing tensorflow and keras
from tensorflow.python.keras import backend as K

logger = tf.get_logger()

class AttentionLayer(tf.keras.layers.Layer):  #Creating attention layer class
    """
    This class implements Bahdanau attention (https://arxiv.org/pdf/1409.0473.pdf
    There are three sets of weights introduced W_a, U_a, and V_a
     """

    def __init__(self, **kwargs): #constructor for AttentionLayer class
        super(AttentionLayer, self).__init__(**kwargs)

    def build(self, input_shape):
        assert isinstance(input_shape, list)
        # Create a trainable weight variable for this layer.

        self.W_a = self.add_weight(name='W_a',
                                   shape=tf.TensorShape((input_shape[0][2], input
                                   initializer='uniform',
                                   trainable=True)
        self.U_a = self.add_weight(name='U_a',
                                   shape=tf.TensorShape((input_shape[1][2], input
                                   initializer='uniform',
                                   trainable=True)
        self.V_a = self.add_weight(name='V_a',
                                   shape=tf.TensorShape((input_shape[0][2], 1)),
                                   initializer='uniform',
                                   trainable=True)

        super(AttentionLayer, self).build(input_shape)  # we will call this funct

    def call(self, inputs):
        """
        inputs: [encoder_output_sequence, decoder_output_sequence]
        """
        assert type(inputs) == list
        encoder_out_seq, decoder_out_seq = inputs

        logger.debug(f"encoder_out_seq.shape = {encoder_out_seq.shape}")
        logger.debug(f"decoder_out_seq.shape = {decoder_out_seq.shape}")

        def energy_step(inputs, states):
            """ Step function for computing energy for a single decoder state
            inputs: (batchsize * 1 * de_in_dim)
            states: (batchsize * 1 * de_latent_dim)
            """

            logger.debug("Running energy computation step")

            if not isinstance(states, (list, tuple)):
                raise TypeError(f"States must be an iterable. Got {states} of typ

            encoder_full_seq = states[-1]
```

```
            """ Computing S.Wa where S=[s0, s1, ..., si]"""
            # <= batch size * en_seq_len * latent_dim
            W_a_dot_s = K.dot(encoder_full_seq, self.W_a)

            """ Computing hj.Ua """
            U_a_dot_h = K.expand_dims(K.dot(inputs, self.U_a), 1)  # <= batch_siz

            logger.debug(f"U_a_dot_h.shape = {U_a_dot_h.shape}")

            """ tanh(S.Wa + hj.Ua) """
            # <= batch_size*en_seq_len, latent_dim
            Ws_plus_Uh = K.tanh(W_a_dot_s + U_a_dot_h)

            logger.debug(f"Ws_plus_Uh.shape = {Ws_plus_Uh.shape}")

            """ softmax(va.tanh(S.Wa + hj.Ua)) """
            # <= batch_size, en_seq_len
            e_i = K.squeeze(K.dot(Ws_plus_Uh, self.V_a), axis=-1)
            # <= batch_size, en_seq_len
            e_i = K.softmax(e_i)

            logger.debug(f"ei.shape = {e_i.shape}")

            return e_i, [e_i]

        def context_step(inputs, states):
            """ Step function for computing ci using ei """

            logger.debug("Running attention vector computation step")

            if not isinstance(states, (list, tuple)):
                raise TypeError(f"States must be an iterable. Got {states} of typ

            encoder_full_seq = states[-1]

            # <= batch_size, hidden_size
            c_i = K.sum(encoder_full_seq * K.expand_dims(inputs, -1), axis=1)

            logger.debug(f"ci.shape = {c_i.shape}")

            return c_i, [c_i]

        # we don't maintain states between steps when computing attention
        # attention is stateless, so we're passing a fake state for RNN step func
        fake_state_c = K.sum(encoder_out_seq, axis=1)
        fake_state_e = K.sum(encoder_out_seq, axis=2)  # <= (batch_size, enc_seq_

        """ Computing energy outputs """
        # e_outputs => (batch_size, de_seq_len, en_seq_len)
        last_out, e_outputs, _ = K.rnn(
            energy_step, decoder_out_seq, [fake_state_e], constants=[encoder_out_
        )

        """ Computing context vectors """
        last_out, c_outputs, _ = K.rnn(
            context_step, e_outputs, [fake_state_c], constants=[encoder_out_seq]
        )
```

```
        return c_outputs, e_outputs

    def compute_output_shape(self, input_shape):
        """ Outputs produced by the layer """
        return [
            tf.TensorShape((input_shape[1][0], input_shape[1][1], input_shape[1][
            tf.TensorShape((input_shape[1][0], input_shape[1][1], input_shape[0][
        ]
```

#Import the Libraries

In [ ]:
```
import numpy as np
import pandas as pd
import re
from bs4 import BeautifulSoup
from keras.preprocessing.text import Tokenizer
from keras_preprocessing.sequence import pad_sequences
from nltk.corpus import stopwords
from tensorflow.keras.layers import Input, LSTM, Embedding, Dense, Concatenate, T
from tensorflow.keras.models import Model
from tensorflow.keras.callbacks import EarlyStopping
import warnings
pd.set_option("display.max_colwidth", 200)
warnings.filterwarnings("ignore")
```

#Read the dataset

This dataset contains text of more than 500,000 We'll take a sample of 200,000 reviews to reduce the training time of our model since our machine does not have that kind of computational power.

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

In [ ]:
```
data=pd.read_csv("/content/drive/MyDrive/NLP/School_text.csv",nrows=200000)
```

Drop Duplicates and NA values

In [ ]:
```
data.drop_duplicates(subset=['Text'],inplace=True)#dropping duplicates
data.dropna(axis=0,inplace=True)#dropping na
```

Information about dataset

Let us look at datatypes and shape of the dataset

In [ ]:
```
data.info() #get an information about the Dataset
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 162834 entries, 0 to 199999
Data columns (total 10 columns):
 #   Column                  Non-Null Count   Dtype 
---  ------                  --------------   ----- 
 0   Id                      162834 non-null  int64 
 1   ProductId               162834 non-null  object
 2   UserId                  162834 non-null  object
 3   ProfileName             162834 non-null  object
 4   HelpfulnessNumerator    162834 non-null  int64 
 5   HelpfulnessDenominator  162834 non-null  int64 
 6   Score                   162834 non-null  int64 
 7   Time                    162834 non-null  int64 
 8   Summary                 162834 non-null  object
 9   Text                    162834 non-null  object
dtypes: int64(5), object(5)
memory usage: 13.7+ MB
```

#Data Preprocessing

Performing basic preprocessing steps is very important before we get to the model building part. Using messy and uncleaned text data is a potentially disastrous move. So in this step, we will drop all the unwanted symbols, characters, etc. from the text that do not affect the objective of our problem.

Here we will create a dictionary that we will use for expanding the contractions:

In [ ]:
```
contraction_mapping = {"ain't": "is not", "aren't": "are not","can't": "cannot", 
                           "didn't": "did not",  "doesn't": "does not", "don't": 
                           "he'd": "he would","he'll": "he will", "he's": "he is'
                           "I'd": "I would", "I'd've": "I would have", "I'll": "I
                           "i'd've": "i would have", "i'll": "i will",  "i'll've'
                           "it'd've": "it would have", "it'll": "it will", "it'll
                           "mayn't": "may not", "might've": "might have","mightn'
                           "mustn't": "must not", "mustn't've": "must not have", 
                           "oughtn't": "ought not", "oughtn't've": "ought not hav
                           "she'd": "she would", "she'd've": "she would have", "s
                           "should've": "should have", "shouldn't": "should not",
                           "this's": "this is","that'd": "that would", "that'd've
                           "there'd've": "there would have", "there's": "there is
                           "they'll": "they will", "they'll've": "they will have'
                           "wasn't": "was not", "we'd": "we would", "we'd've": "w
                           "we've": "we have", "weren't": "were not", "what'll": 
                           "what's": "what is", "what've": "what have", "when's":
                           "where've": "where have", "who'll": "who will", "who'l
                           "why's": "why is", "why've": "why have", "will've": "w
                           "would've": "would have", "wouldn't": "would not", "wo
                           "y'all'd": "you all would","y'all'd've": "you all woul
                           "you'd": "you would", "you'd've": "you would have", "y
                           "you're": "you are", "you've": "you have"}
```

We will perform the below preprocessing tasks for our data:

1.Convert everything to lowercase

2.Remove HTML tags

3.Contraction mapping

4.Remove (‘s)

5.Remove any text inside the parenthesis ( )

6.Eliminate punctuations and special characters

7.Remove stopwords

8.Remove short words

Let’s define the function:

In [ ]:
```
import nltk
nltk.download('stopwords') #download Stopwords
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```

Out[9]: True

In [ ]:
```
stop_words = set(stopwords.words('english'))

def text_cleaner(text,num):  #Text cleaning function for preprocessing our raw te
    newString = text.lower()
    newString = BeautifulSoup(newString, "lxml").text
    newString = re.sub(r'\([^)]*\)', '', newString)
    newString = re.sub('"','', newString)
    newString = ' '.join([contraction_mapping[t] if t in contraction_mapping else
    newString = re.sub(r"'s\b","",newString)
    newString = re.sub("[^a-zA-Z]", " ", newString)
    newString = re.sub('[m]{2,}', 'mm', newString)
    if(num==0):
        tokens = [w for w in newString.split() if not w in stop_words]
    else:
        tokens=newString.split()
    long_words=[]
    for i in tokens:
        if len(i)>1:                                                  #removing sh
            long_words.append(i)
    return (" ".join(long_words)).strip()
```

In [ ]:
```
#call the function
cleaned_text = []
for t in data['Text']:
    cleaned_text.append(text_cleaner(t,0))
```

Let us look at the first five preprocessed reviews

In [ ]:
```
cleaned_text[:5]
```

Out[12]:
```
['bought several vitality canned dog food products found good quality product l
ooks like stew processed meat smells better labrador finicky appreciates produc
t better',
 'product arrived labeled jumbo salted peanuts peanuts actually small sized uns
alted sure error vendor intended represent product jumbo',
 'confection around centuries light pillowy citrus gelatin nuts case filberts c
ut tiny squares liberally coated powdered sugar tiny mouthful heaven chewy flav
orful highly recommend yummy treat familiar story lewis lion witch wardrobe tre
at seduces edmund selling brother sisters witch',
 'looking secret ingredient robitussin believe found got addition root beer ext
ract ordered made cherry soda flavor medicinal',
 'great taffy great price wide assortment yummy taffy delivery quick taffy love
r deal']
```

In [ ]:
```
#call the function
cleaned_summary = [] #intialize empty list for storing the cleaned summary
for t in data['Summary']:
    cleaned_summary.append(text_cleaner(t,1))
```

Let us look at the first 10 preprocessed summaries

In [ ]:
```
cleaned_summary[:10]
```

Out[14]:
```
['good quality dog food',
 'not as advertised',
 'delight says it all',
 'cough medicine',
 'great taffy',
 'nice taffy',
 'great just as good as the expensive brands',
 'wonderful tasty taffy',
 'yay barley',
 'healthy dog food']
```

In [ ]:
```
data['cleaned_text']=cleaned_text
data['cleaned_summary']=cleaned_summary
```

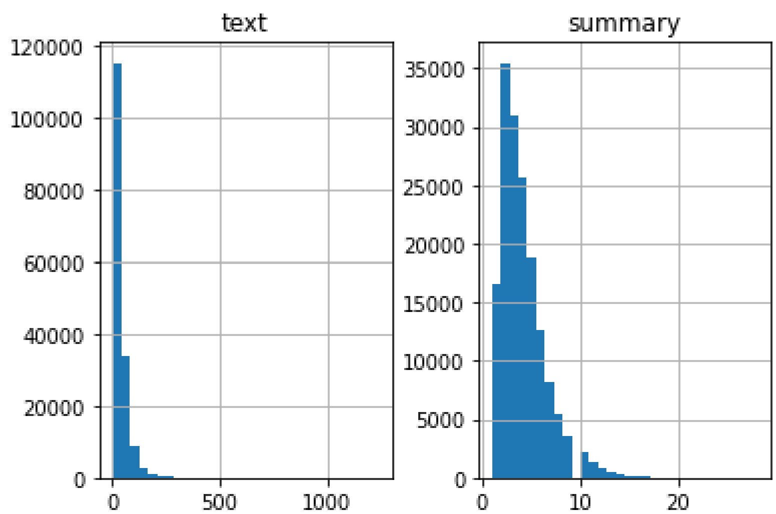
#Drop empty rows

In [ ]:
```
data.replace('', np.nan, inplace=True)
data.dropna(axis=0,inplace=True)
```

#Understanding the distribution of the sequences

Here, we will analyze the length of the Text and the summary to get an idea about the length of the text. we will then use this to fix the maximum length of the sequence:

In [ ]:
```
import matplotlib.pyplot as plt

text_word_count = []
summary_word_count = []

# populate the lists with sentence lengths
for i in data['cleaned_text']:
      text_word_count.append(len(i.split()))

for i in data['cleaned_summary']:
      summary_word_count.append(len(i.split()))

length_df = pd.DataFrame({'text':text_word_count, 'summary':summary_word_count})

length_df.hist(bins = 30)
plt.show() #plotting the distribution
```



Interesting. We can fix the maximum length of the summary to 8 since that seems to be the majority summary length.

Let us understand the proportion of the length of summaries below 8

In [ ]:
```
unt=0
for i in data['cleaned_summary']:
    if(len(i.split())<=8):
        unt=unt+1
print(unt/len(data['cleaned_summary']))
```

```
0.943955449561134
```

We observe that 94% of the summaries have length below 8. So, we can fix maximum length of summary to 8.

Let us fix the maximum length of text to 30

In [ ]:
```
max_text_len=30 #setting the length of text
max_summary_len=8 #setting the length of summary to be genrated to 8
```

Let us select the text and summaries whose length falls below or equal to **max_text_len** and **max_summary_len**

In [ ]:
```
cleaned_text =np.array(data['cleaned_text'])
cleaned_summary=np.array(data['cleaned_summary']). #converting our lists to numpy

short_text=[]  #initializing empty lists to store the summaries
short_summary=[]

for i in range(len(cleaned_text)):
  '''  code for selecting the text and summaries whose length falls below or equa
    if(len(cleaned_summary[i].split())<=max_summary_len and len(cleaned_text[i].s
        short_text.append(cleaned_text[i])
        short_summary.append(cleaned_summary[i])

df=pd.DataFrame({'text':short_text,'summary':short_summary}) #storing the values
```

Here we are adding the **START** and **END** special tokens at the beginning and end of the summary.

This is important because we are using encoder-decoder structure,this is the way the encoder knows that it has recieved an input as we are building a Seq2seq model and the encoder recieves text in a sequence.

Here, for simplicity have chosen **sostok**(representing Start Of Sequence Token) and **eostok**(Representing End of Sequence Token) as START and END tokens

Note: these chosen special tokens never appear in the summary

In [ ]:
```
df['summary'] = df['summary'].apply(lambda x : 'sostok '+ x + ' eostok')
```

Before building our model, we need to split our dataset into a training and validation set. We'll use 90% of the dataset as the training data and evaluate the performance on the remaining 10% (holdout set):

In [ ]:
```
from sklearn.model_selection import train_test_split
x_tr,x_val,y_tr,y_val=train_test_split(np.array(df['text']),np.array(df['summary'
```

#Preparing the Tokenizer

A tokenizer builds the vocabulary and converts a word sequence to an integer sequence.

#Text Tokenizer

So we will build a tokenizer for our text and summary

In [ ]:
```
from keras.preprocessing.text import Tokenizer
from keras_preprocessing.sequence import pad_sequences

#prepare a tokenizer for reviews on training data
x_tokenizer = Tokenizer()
x_tokenizer.fit_on_texts(list(x_tr))
```

#Rarewords and its Coverage

Let us look at the proportion rare words and its total coverage in the entire text

Rare words are the words that appear less frequently than other common words

Here, we will define the threshold to be 4 which means any word whose count is below 4(i,e which appears less than 4 times) is considered as a rare word.

In [ ]:
```
thresh=4
count=0
tot_count=0
freq=0
tot_freq=0

for key,value in x_tokenizer.word_counts.items():
    tot_count=tot_count+1
    tot_freq=tot_freq+value
    if(value<thresh):
        count=count+1
        freq=freq+value

print("% of rare words in vocabulary:",(count/tot_count)*100)
print("Total Coverage of rare words:",(freq/tot_freq)*100)
```

```
% of rare words in vocabulary: 66.66472828773297
Total Coverage of rare words: 2.1762127218939047
```

- **tot_count** gives the size of vocabulary (which means every unique words in the text)
- **count** gives the no. of rare words whose count falls below threshold
- **tot_count - count** gives me the top most common words

Let us define the tokenizer with top most common words for reviews.

In [ ]:
```
#prepare a tokenizer for reviews on training data
x_tokenizer = Tokenizer(num_words=tot_cnt-cnt)
x_tokenizer.fit_on_texts(list(x_tr))

#convert text sequences into integer sequences
x_tr_seq    =   x_tokenizer.texts_to_sequences(x_tr)
x_val_seq   =   x_tokenizer.texts_to_sequences(x_val)

#padding zero upto maximum length
x_tr    =   pad_sequences(x_tr_seq,  maxlen=max_text_len, padding='post')
x_val   =   pad_sequences(x_val_seq, maxlen=max_text_len, padding='post')

#size of vocabulary ( +1 for padding token)
x_voc   =  x_tokenizer.num_words + 1
```

In [ ]:
```
x_voc
```

Out[26]:
```
11466
```

#Summary Tokenizer

In [ ]:
```
#prepare a tokenizer for reviews on training data
y_tokenizer = Tokenizer()
y_tokenizer.fit_on_texts(list(y_tr))
```

#Rarewords and its Coverage

Let us look at the proportion rare words and its total coverage in the summary

Rare words are the words that appear less frequently than other common words

Here, we will define the threshold to be 4 which means any word whose count is below 6(i,e which appears less than 4 times) is considered as a rare word.

In [ ]:
```
thresh=6

cnt=0
tot_cnt=0
freq=0
tot_freq=0

for key,value in y_tokenizer.word_counts.items():
    tot_cnt=tot_cnt+1
    tot_freq=tot_freq+value
    if(value<thresh):
        cnt=cnt+1
        freq=freq+value

print("% of rare words in vocabulary:",(cnt/tot_cnt)*100)
print("Total Coverage of rare words:",(freq/tot_freq)*100)
```

```
% of rare words in vocabulary: 76.86002029822781
Total Coverage of rare words: 3.941994057125159
```

Let us define the tokenizer with top most common words for summary.

In [ ]:
```
#prepare a tokenizer for reviews on training data
y_tokenizer = Tokenizer(num_words=tot_cnt-cnt)
y_tokenizer.fit_on_texts(list(y_tr))

#convert text sequences into integer sequences
y_tr_seq    =   y_tokenizer.texts_to_sequences(y_tr)
y_val_seq   =   y_tokenizer.texts_to_sequences(y_val)

#padding zero upto maximum length
y_tr    =   pad_sequences(y_tr_seq, maxlen=max_summary_len, padding='post')
y_val   =   pad_sequences(y_val_seq, maxlen=max_summary_len, padding='post')

#size of vocabulary
y_voc  =   y_tokenizer.num_words +1
```

Let us check whether word count of start token is equal to length of the training data

In [ ]:
```
y_tokenizer.word_counts['sostok'],len(y_tr)
```

Out[30]:
```
(78845, 78845)
```

Here, Let's delete the rows that contain only **START** and **END** tokens

In [ ]:
```
ind=[]
for i in range(len(y_tr)):
    cnt=0
    for j in y_tr[i]:
        if j!=0:
            cnt=cnt+1
    if(cnt==2):
        ind.append(i)

y_tr=np.delete(y_tr,ind, axis=0)
x_tr=np.delete(x_tr,ind, axis=0)
```

In [ ]:
```
ind=[]
for i in range(len(y_val)):
    cnt=0
    for j in y_val[i]:
        if j!=0:
            cnt=cnt+1
    if(cnt==2):
        ind.append(i)

y_val=np.delete(y_val,ind, axis=0)
x_val=np.delete(x_val,ind, axis=0)
```

Model building

Now We should build the model. But before we do that, we need to know a few terms which are required prior to building the model.

Return Sequences = True: When the return sequences parameter is set to True, LSTM produces the hidden state and cell state for every timestep

Return State = True: When return state = True, LSTM produces the hidden state and cell state of the last timestep only

Initial State: This is used to initialize the internal states of the LSTM for the first timestep

Stacked LSTM: Stacked LSTM has multiple layers of LSTM stacked on top of each other. This leads to a better representation of the sequence.

Here, we will be building a 3 stacked LSTM for the encoder:

In [ ]:
```
!pip install keras-self-attention
```

```
Looking in indexes: https://pypi.org/simple, (https://pypi.org/simple,) http
s://us-python.pkg.dev/colab-wheels/public/simple/ (https://us-python.pkg.dev/co
lab-wheels/public/simple/)
Collecting keras-self-attention
  Downloading keras-self-attention-0.51.0.tar.gz (11 kB)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages
 (from keras-self-attention) (1.21.6)
Building wheels for collected packages: keras-self-attention
  Building wheel for keras-self-attention (setup.py) ... done
  Created wheel for keras-self-attention: filename=keras_self_attention-0.51.0-
py3-none-any.whl size=18912 sha256=837ad54ac11bfb91d42ef229522a7f1b66fa0fae14c7
43a998022c507de9cdf1
  Stored in directory: /root/.cache/pip/wheels/95/b1/a8/5ee00cc137940b2f6fa1982
12e8f45d813d0e0d9c3a04035a3
Successfully built keras-self-attention
Installing collected packages: keras-self-attention
Successfully installed keras-self-attention-0.51.0
```

In [ ]:
```
from keras import backend as K

K.clear_session()

latent_dim = 300
embedding_dim=100

# Encoder
encoder_inputs = Input(shape=(max_text_len,))

#embedding layer
enc_emb =  Embedding(x_voc, embedding_dim,trainable=True)(encoder_inputs)

#encoder lstm 1
encoder_lstm1 = LSTM(latent_dim,return_sequences=True,return_state=True,dropout=0
encoder_output1, state_h1, state_c1 = encoder_lstm1(enc_emb)

#encoder lstm 2
encoder_lstm2 = LSTM(latent_dim,return_sequences=True,return_state=True,dropout=0
encoder_output2, state_h2, state_c2 = encoder_lstm2(encoder_output1)

#encoder lstm 3
encoder_lstm3=LSTM(latent_dim, return_state=True, return_sequences=True,dropout=0
encoder_outputs, state_h, state_c= encoder_lstm3(encoder_output2)

# Set up the decoder, using `encoder_states` as initial state.
decoder_inputs = Input(shape=(None,))

#embedding layer
dec_emb_layer = Embedding(y_voc, embedding_dim,trainable=True)
dec_emb = dec_emb_layer(decoder_inputs)

decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True,dropout=
decoder_outputs,decoder_fwd_state, decoder_back_state = decoder_lstm(dec_emb,init

# Attention layer
attn_layer = AttentionLayer(name='attention_layer')
attn_out, attn_states = attn_layer([encoder_outputs, decoder_outputs])

# Concat attention input and decoder LSTM output
decoder_concat_input = Concatenate(axis=-1, name='concat_layer')([decoder_outputs

#dense layer
decoder_dense =  TimeDistributed(Dense(y_voc, activation='softmax'))
decoder_outputs = decoder_dense(decoder_concat_input)

# Define the model
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

model.summary()
```

```
WARNING:tensorflow:Layer lstm will not use cuDNN kernels since it doesn't me
et the criteria. It will use a generic GPU kernel as fallback when running o
n GPU.
WARNING:tensorflow:Layer lstm_1 will not use cuDNN kernels since it doesn't
meet the criteria. It will use a generic GPU kernel as fallback when running
```

```
on GPU.
WARNING:tensorflow:Layer lstm_2 will not use cuDNN kernels since it doesn't
meet the criteria. It will use a generic GPU kernel as fallback when running
on GPU.
WARNING:tensorflow:Layer lstm_3 will not use cuDNN kernels since it doesn't
meet the criteria. It will use a generic GPU kernel as fallback when running
on GPU.

Model: "model"
________________________________________________________________________________
___________________
 Layer (type)                   Output Shape         Param #     Connected to
================================================================================
===================
 input_1 (InputLayer)           [(None, 30)]         0           []

 embedding (Embedding)          (None, 30, 100)      1146600     ['input_1[0]
[0]']

 lstm (LSTM)                    [(None, 30, 300),    481200      ['embedding[0]
[0]']
                                 (None, 300),
                                 (None, 300)]

 input_2 (InputLayer)           [(None, None)]       0           []

 lstm_1 (LSTM)                  [(None, 30, 300),    721200      ['lstm[0][0]']
                                 (None, 300),
                                 (None, 300)]

 embedding_1 (Embedding)        (None, None, 100)    296500      ['input_2[0]
[0]']

 lstm_2 (LSTM)                  [(None, 30, 300),    721200      ['lstm_1[0]
[0]']
                                 (None, 300),
                                 (None, 300)]

 lstm_3 (LSTM)                  [(None, None, 300),  481200      ['embedding_1
[0][0]',
                                 (None, 300),                     'lstm_2[0]
[1]',
                                 (None, 300)]                     'lstm_2[0]
[2]']

 attention_layer (AttentionLaye  ((None, None, 300),  180300     ['lstm_2[0]
[0]',
 r)                              (None, None, 30))                'lstm_3[0]
[0]']

 concat_layer (Concatenate)     (None, None, 600)    0           ['lstm_3[0]
[0]',
                                                                  'attention_la
yer[0][0]']

 time_distributed (TimeDistribu  (None, None, 2965)  1781965     ['concat_layer
[0][0]']
```

```
 ted)

================================================================================
===================
Total params: 5,810,165
Trainable params: 5,810,165
Non-trainable params: 0
________________________________________________________________________________
___________________
```

we are using sparse categorical cross-entropy as the loss function since it converts the integer sequence to a one-hot vector on the fly. This overcomes any memory issues.

In [ ]:
```
model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy')
```

Remember the concept of early stopping? It is used to stop training the neural network at the right time by monitoring a user-specified metric.

Here, we'll be monitoring the validation loss (val_loss). Our model will stop training once the validation loss increases:

In [ ]:
```
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1,patience=2)
```

We'll train the model on a batch size of 128 and validate it on the holdout set (which is 10% of our dataset):

In [ ]:
```
#from sklearn.model_selection import train_test_split
#x_tr,x_val,y_tr,y_val=train_test_split(np.array(df['text']),np.array(df['summary
```

In [ ]:
```
tf.config.run_functions_eagerly(True) #We will use this to  make all invocations
```

In [ ]:
```
history=model.fit([x_tr,y_tr[:,:-1]], y_tr.reshape(y_tr.shape[0],y_tr.shape[1], 1
```

```
Epoch 1/10
604/604 [==============================] - 657s 1s/step - loss: 2.8004 - val_lo
ss: 2.5370
Epoch 2/10
604/604 [==============================] - 655s 1s/step - loss: 2.4766 - val_lo
ss: 2.3749
Epoch 3/10
604/604 [==============================] - 636s 1s/step - loss: 2.3480 - val_lo
ss: 2.2856
Epoch 4/10
604/604 [==============================] - 636s 1s/step - loss: 2.2699 - val_lo
ss: 2.2358
Epoch 5/10
604/604 [==============================] - 636s 1s/step - loss: 2.2142 - val_lo
ss: 2.1954
Epoch 6/10
604/604 [==============================] - 644s 1s/step - loss: 2.1674 - val_lo
ss: 2.1604
Epoch 7/10
604/604 [==============================] - 648s 1s/step - loss: 2.1296 - val_lo
ss: 2.1364
Epoch 8/10
604/604 [==============================] - 656s 1s/step - loss: 2.0967 - val_lo
ss: 2.1183
Epoch 9/10
604/604 [==============================] - 660s 1s/step - loss: 2.0679 - val_lo
ss: 2.0968
Epoch 10/10
604/604 [==============================] - 642s 1s/step - loss: 2.0423 - val_lo
ss: 2.0852
```

LOADING AND SAVING THE MODEL

As we saw Training our Model only took 2 hours of time to run on myPC since we canot always run the model whenever we want as its time consuming we can load and save the model with the weights so that it is portable and easy to us the next time we need it

In [ ]:
```
from keras.models import model_from_json

#Saving THE WEIGHTS OF THE DEEP LEARNING NETWORK
'''
model_save_json = model.to_json()
with open("/content/drive/MyDrive/NLP/Project/model_save.json", "w") as json_file
    json_file.write(model_save_json)
# serialize weights to HDF5
model.save_weights("/content/drive/MyDrive/NLP/Project/model_save.h5")
print("Saved model to disk")

'''
#LOADING THE WEIGHTS OF THE DEEP LEARNING NETWORK
# load json and create model
json_file = open('model_save.json', 'r')
loaded_model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(loaded_model_json)
# load weights into new model
loaded_model.load_weights("model_save.h5")
print("Loaded model from disk")
```

```
Saved model to disk
```

Out[57]:
```
'\nLOADING THE WEIGHTS OF THE DEEP LEARNING NETWORK\n# load json and create mod
el\njson_file = open(\'model_cat.json\', \'r\')\nloaded_model_json = json_file.
read()\njson_file.close()\nloaded_model = model_from_json(loaded_model_json)\n#
load weights into new model\nloaded_model.load_weights("model.h5")\nprint("Load
ed model from disk")\n'
```

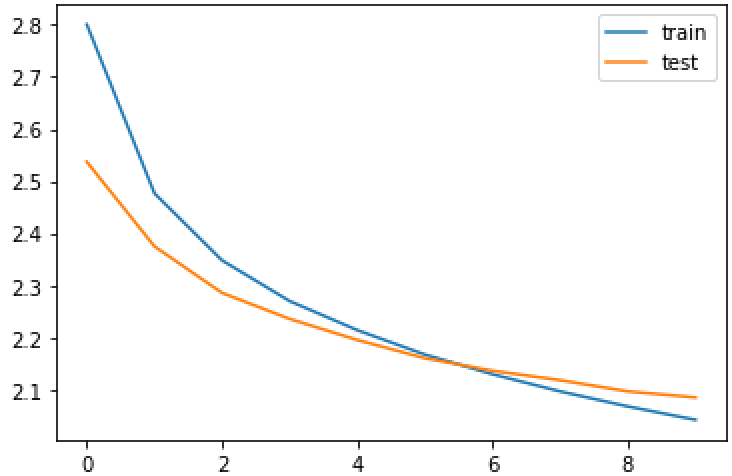
LOADING AND SAVING THE HISTORY OF THE MODEL

In [ ]:
```
#SAVING THE HISTORY
#/content/drive/MyDrive/my_history_cat.npy -> /content/drive/MyDrive/NLP/Project/
#np.save('/content/drive/MyDrive/NLP/Project/my_history_save.npy',history.history

#Loading the history

history=np.load('/content/drive/MyDrive/NLP/Project/my_history_save.npy',allow_pi
```

#Understanding the Diagnostic plot

Now, we will plot a few diagnostic plots to understand the behavior of the model over time:

In [ ]:
```
from matplotlib import pyplot
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
pyplot.show()
```



From the plot, we can infer that validation loss has increased after epoch 8.

Next, let's build the dictionary to convert the index to word for target and source vocabulary:

In [ ]:
```
reverse_target_word_index=y_tokenizer.index_word
reverse_source_word_index=x_tokenizer.index_word
target_word_index=y_tokenizer.word_index
```

Inference

we will Set up the inference for the encoder and decoder:

The stage of growth known as deep learning inference is where the skills acquired during training are put to use. When presented with new data that the model has never seen before, the trained deep neural networks (DNN) draws conclusions or make predictions.

In [ ]:
```
# Encode the input sequence to get the feature vector
encoder_model = Model(inputs=encoder_inputs,outputs=[encoder_outputs, state_h, st

# Decoder setup
# Below tensors will hold the states of the previous time step
decoder_state_input_h = Input(shape=(latent_dim,))
decoder_state_input_c = Input(shape=(latent_dim,))
decoder_hidden_state_input = Input(shape=(max_text_len,latent_dim))

# Get the embeddings of the decoder sequence
dec_emb2= dec_emb_layer(decoder_inputs)
# To predict the next word in the sequence, set the initial states to the states
decoder_outputs2, state_h2, state_c2 = decoder_lstm(dec_emb2, initial_state=[dec

#attention inference
attn_out_inf, attn_states_inf = attn_layer([decoder_hidden_state_input, decoder_
decoder_inf_concat = Concatenate(axis=-1, name='concat')([decoder_outputs2, attn_

# A dense softmax layer to generate prob dist. over the target vocabulary
decoder_outputs2 = decoder_dense(decoder_inf_concat)

# Final decoder model
decoder_model = Model(
    [decoder_inputs] + [decoder_hidden_state_input,decoder_state_input_h, decoder
    [decoder_outputs2] + [state_h2, state_c2])
```

We are defining a function below which is the implementation of the inference process

In [ ]:
```
def decode_sequence(input_seq):
    # Encode the input as state vectors.
    e_out, e_h, e_c = encoder_model.predict(input_seq)

    # Generate empty target sequence of length 1.
    target_seq = np.zeros((1,1))

    # Populate the first word of target sequence with the start word.
    target_seq[0, 0] = target_word_index['sostok']

    stop_condition = False
    decoded_sentence = ''
    while not stop_condition:

        output_tokens, h, c = decoder_model.predict([target_seq] + [e_out, e_h, e

        # Sample a token
        sampled_token_index = np.argmax(output_tokens[0, -1, :])
        sampled_token = reverse_target_word_index[sampled_token_index]

        if(sampled_token!='eostok'):
            decoded_sentence += ' '+sampled_token

        # Exit condition: either hit max length or find stop word.
        if (sampled_token == 'eostok'  or len(decoded_sentence.split()) >= (max_s
            stop_condition = True

        # Update the target sequence (of length 1).
        target_seq = np.zeros((1,1))
        target_seq[0, 0] = sampled_token_index

        # Update internal states
        e_h, e_c = h, c

    return decoded_sentence
```

NOw we'll define the functions to convert an integer sequence to a word sequence for summary as well as the reviews:

In [ ]:
```
def seq2summary(input_seq):
    newString=''
    for i in input_seq:
        if((i!=0 and i!=target_word_index['sostok']) and i!=target_word_index['eo
            newString=newString+reverse_target_word_index[i]+' '
    return newString

def seq2text(input_seq):
    newString=''
    for i in input_seq:
        if(i!=0):
            newString=newString+reverse_source_word_index[i]+' '
    return newString
```

Here are a few summaries generated by the model:

In [ ]:
```
for i in range(0,100):
    print("Review:",seq2text(x_tr[i]))
    print("Original summary:",seq2summary(y_tr[i]))
    print("Predicted summary:",decode_sequence(x_tr[i].reshape(1,max_text_len)))
    print("\n")
```

```
Review: delightful product buying pretty easy absolutely addicting everyone
wants carmel corn product made packages last weeks yummy
Original summary: scrumptious
1/1 [==============================] - 0s 365ms/step
1/1 [==============================] - 0s 43ms/step
1/1 [==============================] - 0s 41ms/step
Predicted summary:  delicious


Review: loves tastes good meaty cousin eat meat miss cousin put chopped cila
ntro onions sour cream top good time
Original summary: love this stuff
1/1 [==============================] - 0s 369ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 42ms/step
1/1 [==============================] - 0s 44ms/step
Predicted summary:  great taste
```

Even though the actual summary and the summary generated by our model do not match in terms of words, both of them are conveying the same meaning. Our model is able to generate a clear summary based on the context present in the text.

#Conclusion

Here we have built our own Deep Learning Model from Scratch, we can still do a lot of things to improve the performance of the model such as

1. Training with more dataset
2. implementing a Bi-Directional LSTM model

and many more

But that's all for another day.