Text Summarization using deep Learning, Pegasus and Hugging Face

Given long documents to read, our natural preference is to not read, or at least, to scan just the main points. So having a summary would always be great to save us time and brain processing power.

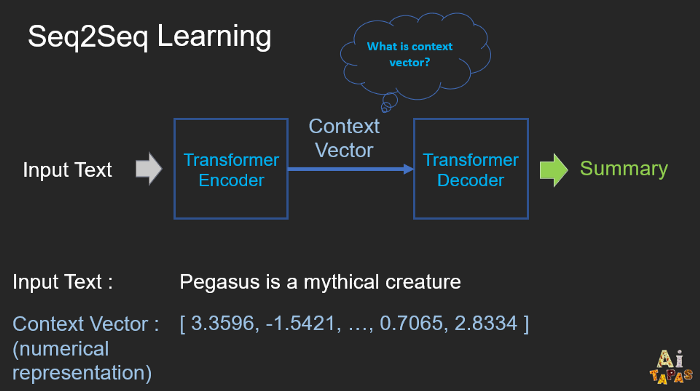
However, auto-summarization used to be an impossible task. Specifically, abstractive summarization is very challenging. Differing from extractive summarization (which extracts important sentences from a document and combines them to form a “summary”), abstractive summarization involves paraphrasing words and hence, is more difficult but can potentially give a more coherent and polished summary.

It was not until the development of techniques like seq2seq learning and unsupervised language models (e.g., ELMo and BERT) that abstractive summarization becomes more feasible.

Building upon earlier breakthroughs in natural language processing (NLP) field, Google’s PEGASUS further improved the state-of-the-art (SOTA) results for abstractive summarization, in particular with low resources. To be more specific, unlike previous models, PEGASUS enables us to achieve close to SOTA results with 1,000 examples, rather than tens of thousands of training data.

How PEGASUS works

## Architecture:-



On a high level, PEGASUS uses an encoder-decoder model for sequence-to-sequence learning. In such a model, the encoder will first take into consideration the context of the whole input text and encode the input text into something called context vector, which is basically a numerical representation of the input text.

This numerical representation will then be fed to the decoder whose job is decode the context vector to produce the summary.

In line with recent SOTA NLP models, PEGASUS also adopts the transformer architecture.

## Pre-training:-

What differentiates PEGASUS from previous SOTA models is the pre-training.

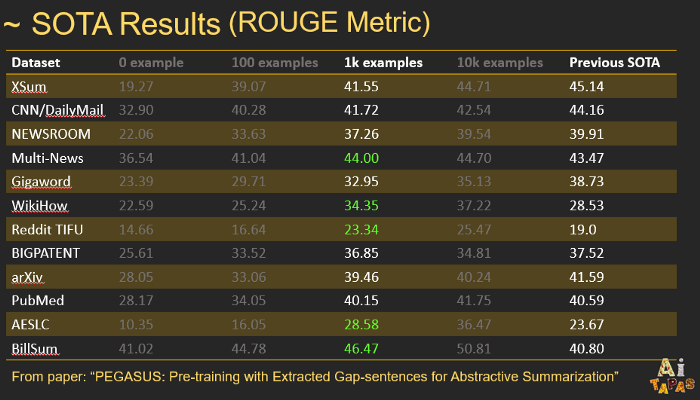
The authors (Jingqing Zhang et. al.) hypothesizes that pre-training the model to output important sentences is suitable as it closely resembles what abstractive summarization needs to do.

Using a metric called [ROUGE1-F1](https://www.freecodecamp.org/news/what-is-rouge-and-how-it-works-for-evaluation-of-summaries-e059fb8ac840/), the authors were able to automate the selection of “important” sentences and perform pre-training of the model on a large corpus, i.e., 350 million web pages and 1.5 billion news articles.

With the pre-trained model, we can then perform fine-tuning of the model on the actual data which is of a much smaller quantity. In fact, evaluation results on various datasets showed that with just 1,000 training data, the model achieved comparable results to previous SOTA models.

This has important practical implications as most of us will not have the resources to collect tens of thousands of document-summary pairs.

Let us now take a look at a descriptive image of the work:-



## How to use Pegasus?

# Upon seeing the evaluation results for PEGASUS, you are probably wondering how you can write the code to use the model. Fortunately for us, Hugging Face 🤗 has PEGASUS model in-store, making it easy for us to leverage on PEGASUS.

## 1. Inference:-

# https://miro.medium.com/max/700/1*NMxLvJ7h_uMTJVjRf08y3Q.png

You can swap the *model\_name* with various other fine-tuned models (except for *google/pegasus-large*) listed [here](https://huggingface.co/models?search=pegasus), based on how similar your use case is to the dataset used for fine-tuning.

## https://miro.medium.com/max/700/1*tuLaBI0nEmfo8WJYcNCRFg.png

## 2. Fine-Tuning

## If you would like to have a customized model for your use case, you can fine-tune the google/pegasus-large model on your dataset.

## https://miro.medium.com/max/700/1*oAlpSQnhaDoZJBJFfK3yZg.png

## Do however note that fine-tuning both the encoder and decoder can be very memory-intensive. If your local computer is unfortunately not up to the task (like mine), you can consider using Google Cloud. And since Google Cloud has a [free trial](https://cloud.google.com/free/docs/gcp-free-tier) for new signups, you can experiment at no cost.

## Transformers:-

## For a long time, recurrent models (such as Recurring Neural Networks, Long Short-Term Memory, etc) have been in use for language modeling and machine translation problems. These models, however, are inherently sequential in nature, and this poses problems when we have to deal with longer sequence lengths, as memory constraints begin to limit batching across examples. Since RNNs process data sequentially (or one data element at a time), it isn't possible to speed them up significantly by simply increasing the amount of processing power.

## This makes it difficult for us to train RNNs on very large amounts of data. This is because they're slower to train, owing to the reduced scope for parallelization. While we have managed to greatly increase the computational efficiency of such models in recent times, the drawbacks that are associated with sequential computation are still real and detrimental.

## This is where Transformers come into the picture. Transformers are a type of neural network architecture, and were developed by a group of researchers at Google (and UoT) in 2017. They avoid using the principle of recurrence, and work entirely on an attention mechanism to draw global dependencies between the input and the output.

## Transformers allow for much more parallelization than sequential models, and can achieve very high translation quality even after being trained only for short periods of time. They can also be trained on very large amounts of data without as much difficulty. The GPT-3 (Generative Pre-Trained Transformer-3) model, for example, was trained on an exceptionally large amount of data, nearing 45 terabytes!

There are three distinct features about Transformers that make them work so efficiently:

1. Positional Encoding
2. Attention
3. Self-Attention

### Positional Encoding

Positional Encoding is the idea that instead of looking at a sentence sequentially (or word by word), we take each word and assign it a unique representation.

So if we have a sentence like:  
'He likes to eat ice cream.'

Then, each word of the sentence is assigned a specific index as follows:  
He -> 1  
likes -> 2  
to -> 3  
eat -> 4  
ice -> 5  
cream. -> 6

In practice, these words are not assigned simple index values. Transformers use smart positional encoding schemes, where each position is mapped to a vector. Thus, we actually end up with a positional encoding matrix, where we can find positional information as well as the encoded objects of the sentence. For now, however, to understand these concepts better, we will use the above example.

By indexing each word of the sentence, we are storing information about the word order in the data itself, rather than in the structure of the network. Initially, the network will not know how to interpret these positional encodings, but as it works with large amounts of data, it begins to learn how to make use of them. Thus, the network learns about the importance of word order *from the data*. This makes it easier for us to train Transformers than RNNs.

### Attention

Suppose we want to translate a sentence from English to Hindi. One primitive way to do so would be to take each word of the English sentence, one by one, and translate it to Hindi. This is not an ideal way to translate a sentence, since the order of words in sentences conveying the same message is often slightly different from language to language.

For example, if we have the sentence:  
'Max wants to play.'

The equivalent sentence in Hindi would be:  
'मैक्स खेलना चाहता है।'

Here, the Hindi words for 'wants to' and 'play' are shuffled. If we translated the English sentence one word at a time, however, we'd end up with the sentence:

'मैक्स चाहता है खेल।'

This sentence, while still somewhat decipherable, is far from being the *correct* Hindi representation of the English sentence 'Max wants to play.'

Attention is a mechanism that makes it possible for a model to contextualize each word in a given sentence, which gives it a clearer picture of what the final translated sentence should look like. This is something that the model can learn with the help of training data.

By looking at a large number of examples of sentences in both languages, the model can start to learn about the interdependency of words, the different rules of grammar and punctuation, and more such linguistic details.

### Self-Attention:

In the previous section of the article, we talked mostly about text translation, which is a very specific language task. What if instead, we wanted to build a model that could learn about the core fundamentals of a language such that it could perform any number of language tasks? This is what Self-Attention helps us do. Self-Attention makes it possible for a neural network to understand a word in the context of the sentence that it is used in.

For example, in the following sentences:

'He was told to *park* the car closer to the wall.'  
'She went to the *park* today.'

In these sentences, the word 'park' assumes different meanings. Humans can easily contextualize these words and infer that they mean different things, and this is what Self-Attention allows neural networks to do.

In the first sentence, the model might look at the word 'car' and infer that the word 'park' here means- the parking of a vehicle, whereas in the second sentence, the model might look at the words 'She went' and infer that the word 'park' here means- a playground or a garden.

## How Do Transformers Work?

## transformer

## As we can see from the above diagram, Transformers are comprised of a few different sections: an encoder stack and a decoder stack, input and output pre-processing units, and an output post-processing unit. Here, the input sequence is mapped into an abstract continuous representation by the encoder. The decoder then takes this abstract continuous representation and generates an output while being fed the previous output.

## transformer-1

The first encoder receives the word embeddings of the input sequence. Then, these are transformed and sent to the next encoder. Finally, the output from the last encoder in the stack of encoders is passed to the entire stack of decoders.

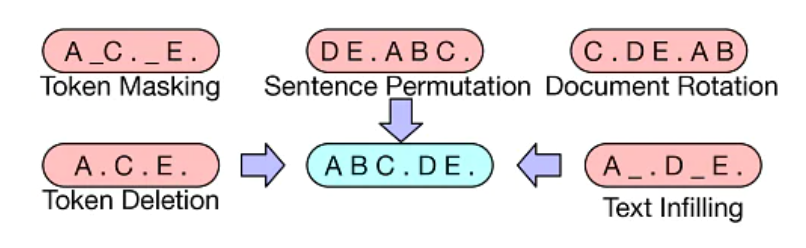
Let's quickly take a look at the architecture and the uses of a type of Transformer known as BART.

## BART:-

## BART, or Bi-Directional Auto Regressive Transformer, is a sequence-to-sequence de-noising auto encoder. A BART model is capable of taking an input text sequence and generating a different output text sequence (for example, an English input -> a French output). These models are commonly used in machine translation, text and sentence classification, as well as text summarization!

## The BART model as proposed in the research paper 'BART: De-noising Sequence-to-Sequence Pre-Training for Natural Language Generation, Translation, and Comprehension', was trained using training data that included noisy (or corrupt) data, which was then mapped to the original data that it was obtained from. The authors of the research paper used multiple novel transformations in order to introduce noise to the data for pre-training. Some of these transformations are as follows:

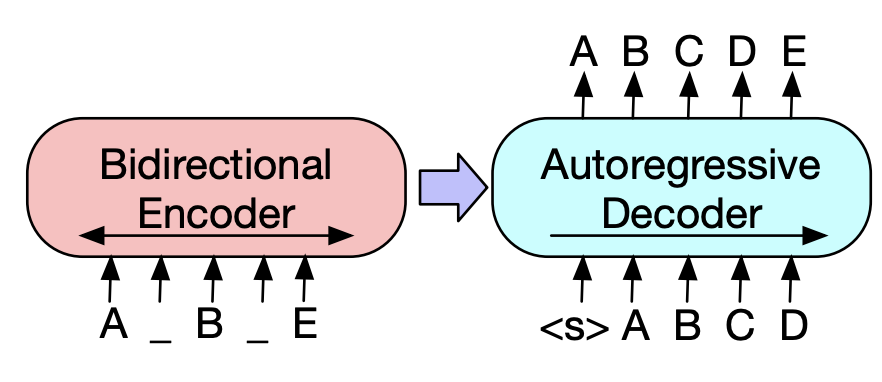
1. Token Masking: Random tokens in sentences are replaced with [MASK] elements.
2. Token Deletion: Random tokens in the input sequence are deleted. It is up to the model to predict which positions do not have inputs.
3. Text Infilling: A number of tokens are replaced with a singular [MASK] token. It is up to the model to learn about how many tokens are missing as well as the  
   content of the missing tokens.
4. Sentence Permutation: Sentences are randomly rearranged/permuted. This makes it possible for the model to learn the logical sequence of sentences.
5. Document Rotation: A random token is picked, and the document is made to begin with that token. The content before that particular token is added to the end of the document. This makes it possible for the model to identify what the start of the document looks like.



The first part of BART uses the bi-directional encoder of BERT (which is another Transformer model based on bi-directional encoding) to find the ideal representation of the input text sequence.

Once we have this ideal input text sequence representation, we require a decoder to interpret the input text sequence and map it to the output target. To do this, we use the auto regressive decoder of GPT (which is another Transformer model based on auto regressive decoding), which only looks at the past data to make new predictions.

Thus, the BART model looks something like this:



## Here, the input sequence is a corrupted version of [ABCDE], transformed into [A[MASK]B[MASK]E]. This corrupted sequence is then encoded with the bi-directional encoder, and the likelihood of the original text, [ABCDE], is calculated by the auto regressive decoder. This model can further be fine tuned by feeding the un-corrupted original text to both the encoder as well as the decoder.

## As we discussed earlier, the BART model is a powerful tool in machine translation, text and sentence classification, as well as text summarization. The BART model performs comparably or better than previously proposed models in various tasks.

## Now that we have a strong understanding of vanilla Transformers as well as the BART model, we will work with HuggingFace's Transformers (which is a Python library) to learn how these models are used in the real world! We will do so using the pipeline API.

## Using seq2seq, encoder, decoder model:-

## *Step 1: Creating the Model:-*

First, import all the necessary libraries.

## from tensorflow.keras.preprocessing.text import Tokenizer

## from tensorflow.keras.preprocessing.sequence import pad\_sequences

## from tensorflow.keras.layers import Input, LSTM, Embedding, Dense, \

## Concatenate, TimeDistributed

## from tensorflow.keras.models import Model

## from tensorflow.keras.callbacks import EarlyStopping

## Next, define the Encoder and Decoder networks.

#### Encoder:-

The input length that the encoder accepts is equal to the maximum text length which you’ve already estimated in [Step 3](https://blog.paperspace.com/introduction-to-seq2seq-models/). This is then given to an Embedding Layer of dimension

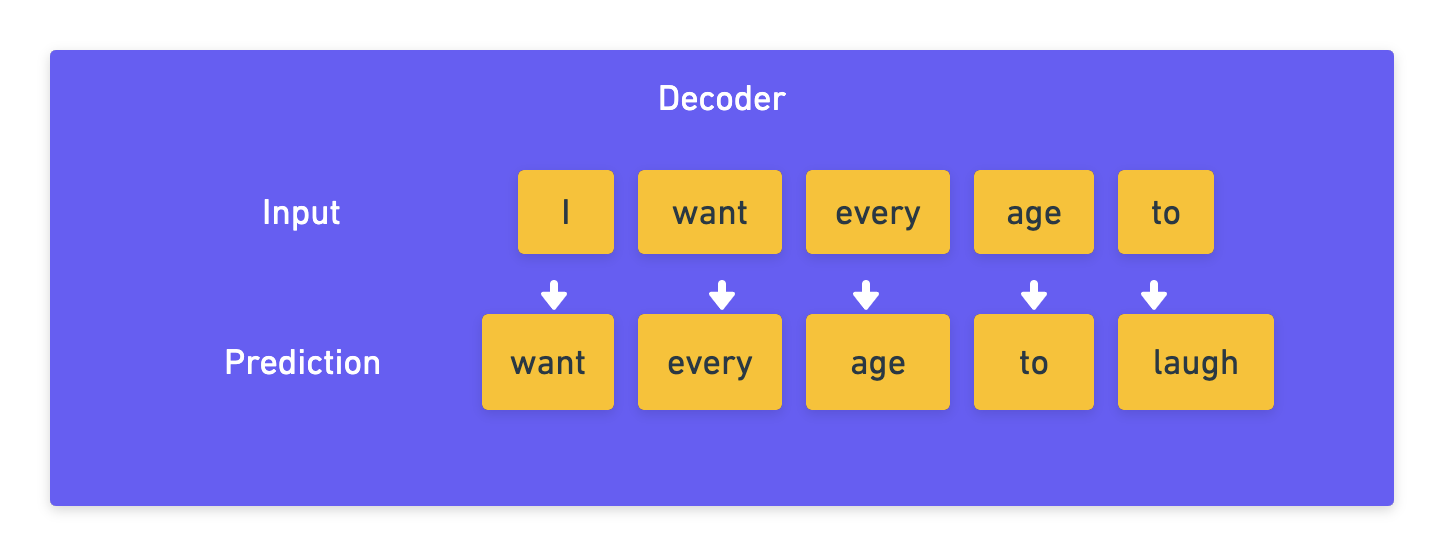
(total number of words captured in the text vocabulary) x (number of nodes in an embedding layer)

(calculated in [Step 5](https://blog.paperspace.com/introduction-to-seq2seq-models/); the x\_voc variable). This is followed by three LSTM networks wherein each layer returns the LSTM output, as well as the hidden and cell states observed at the previous time steps.

#### Decoder:-

In the decoder, an embedding layer is defined followed by an LSTM network. The initial state of the LSTM network is the last hidden and cell states taken from the encoder. The output of the LSTM is given to a Dense layer wrapped in a TimeDistributed layer with an attached softmax activation function.

Altogether, the model accepts encoder (text) and decoder (summary) as input and it outputs the summary. The prediction happens through predicting the upcoming word of the summary from the previous word of the summary (see the below figure).



Add the following code to define your network architecture.:-

latent\_dim = 300

embedding\_dim = 200

# 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=0.4,

recurrent\_dropout=0.4)

(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=0.4,

recurrent\_dropout=0.4)

(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=0.4,

recurrent\_dropout=0.4)

(encoder\_outputs, state\_h, state\_c) = encoder\_lstm3(encoder\_output2)

# Set up the decoder, using encoder\_states as the 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

decoder\_lstm = LSTM(latent\_dim, return\_sequences=True,

return\_state=True, dropout=0.4,

recurrent\_dropout=0.2)

(decoder\_outputs, decoder\_fwd\_state, decoder\_back\_state) = \

decoder\_lstm(dec\_emb, initial\_state=[state\_h, state\_c])

# Dense layer

decoder\_dense = TimeDistributed(Dense(y\_voc, activation='softmax'))

decoder\_outputs = decoder\_dense(decoder\_outputs)

# Define the model

model = Model([encoder\_inputs, decoder\_inputs], decoder\_outputs)

model.summary()

# Output

Model: "model"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param # Connected to

==================================================================================================

input\_1 (InputLayer) [(None, 100)] 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

embedding (Embedding) (None, 100, 200) 5927600 input\_1[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

lstm (LSTM) [(None, 100, 300), ( 601200 embedding[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

input\_2 (InputLayer) [(None, None)] 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

lstm\_1 (LSTM) [(None, 100, 300), ( 721200 lstm[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

embedding\_1 (Embedding) (None, None, 200) 2576600 input\_2[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

lstm\_2 (LSTM) [(None, 100, 300), ( 721200 lstm\_1[0][0]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

lstm\_3 (LSTM) [(None, None, 300), 601200 embedding\_1[0][0]

lstm\_2[0][1]

lstm\_2[0][2]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

time\_distributed (TimeDistribut (None, None, 12883) 3877783 lstm\_3[0][0]

==================================================================================================

Total params: 15,026,783

Trainable params: 15,026,783

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#### Step 8: Training the Model

In this step, compile the model and define EarlyStopping to stop training the model once the validation loss metric has stopped decreasing.

*model.compile(optimizer='rmsprop', loss='sparse\_categorical\_crossentropy')*

*es = EarlyStopping(monitor='val\_loss', mode='min', verbose=1, patience=2)*

Next, use the model.fit() method to fit the training data where you can define the batch size to be 128. Send the text and summary (excluding the last word in summary) as the input, and a reshaped summary tensor comprising every word (starting from the second word) as the output (which explains the infusion of intelligence into the model to predict a word, given the previous word). Besides, to enable validation during the training phase, send the validation data as well.

history = model.fit(

[x\_tr, y\_tr[:, :-1]],

y\_tr.reshape(y\_tr.shape[0], y\_tr.shape[1], 1)[:, 1:],

epochs=50,

callbacks=[es],

batch\_size=128,

validation\_data=([x\_val, y\_val[:, :-1]],

y\_val.reshape(y\_val.shape[0], y\_val.shape[1], 1)[:

, 1:]),

)

# Output

Train on 88513 samples, validate on 9835 samples

Epoch 1/50

88513/88513 [==============================] - 426s 5ms/sample - loss: 5.1520 - val\_loss: 4.8026

Epoch 2/50

88513/88513 [==============================] - 412s 5ms/sample - loss: 4.7110 - val\_loss: 4.5082

Epoch 3/50

88513/88513 [==============================] - 412s 5ms/sample - loss: 4.4448 - val\_loss: 4.2815

Epoch 4/50

88513/88513 [==============================] - 411s 5ms/sample - loss: 4.2487 - val\_loss: 4.1264

Epoch 5/50

88513/88513 [==============================] - 410s 5ms/sample - loss: 4.1049 - val\_loss: 4.0170

Epoch 6/50

88513/88513 [==============================] - 411s 5ms/sample - loss: 3.9968 - val\_loss: 3.9353

Epoch 7/50

88513/88513 [==============================] - 412s 5ms/sample - loss: 3.9086 - val\_loss: 3.8695

Epoch 8/50

88513/88513 [==============================] - 411s 5ms/sample - loss: 3.8321 - val\_loss: 3.8059

Epoch 9/50

88513/88513 [==============================] - 411s 5ms/sample - loss: 3.7598 - val\_loss: 3.7517

Epoch 10/50

88513/88513 [==============================] - 410s 5ms/sample - loss: 3.6948 - val\_loss: 3.7054

Epoch 11/50

88513/88513 [==============================] - 411s 5ms/sample - loss: 3.6408 - val\_loss: 3.6701

Epoch 12/50

88513/88513 [==============================] - 410s 5ms/sample - loss: 3.5909 - val\_loss: 3.6376

Epoch 13/50

88513/88513 [==============================] - 411s 5ms/sample - loss: 3.5451 - val\_loss: 3.6075

Epoch 14/50

88513/88513 [==============================] - 412s 5ms/sample - loss: 3.5065 - val\_loss: 3.5879

Epoch 15/50

88513/88513 [==============================] - 411s 5ms/sample - loss: 3.4690 - val\_loss: 3.5552

Epoch 16/50

88513/88513 [==============================] - 409s 5ms/sample - loss: 3.4322 - val\_loss: 3.5308

Epoch 17/50

88513/88513 [==============================] - 410s 5ms/sample - loss: 3.3981 - val\_loss: 3.5123

Epoch 18/50

88513/88513 [==============================] - 409s 5ms/sample - loss: 3.3683 - val\_loss: 3.4956

Epoch 19/50

88513/88513 [==============================] - 409s 5ms/sample - loss: 3.3379 - val\_loss: 3.4787

Epoch 20/50

88513/88513 [==============================] - 409s 5ms/sample - loss: 3.3061 - val\_loss: 3.4594

Epoch 21/50

88513/88513 [==============================] - 410s 5ms/sample - loss: 3.2803 - val\_loss: 3.4412

Epoch 22/50

88513/88513 [==============================] - 409s 5ms/sample - loss: 3.2552 - val\_loss: 3.4284

Epoch 23/50

88513/88513 [==============================] - 410s 5ms/sample - loss: 3.2337 - val\_loss: 3.4168

Epoch 24/50

88513/88513 [==============================] - 410s 5ms/sample - loss: 3.2123 - val\_loss: 3.4148

Epoch 25/50

88513/88513 [==============================] - 409s 5ms/sample - loss: 3.1924 - val\_loss: 3.3974

Epoch 26/50

88513/88513 [==============================] - 410s 5ms/sample - loss: 3.1727 - val\_loss: 3.3869

Epoch 27/50

88513/88513 [==============================] - 409s 5ms/sample - loss: 3.1546 - val\_loss: 3.3853

Epoch 28/50

88513/88513 [==============================] - 408s 5ms/sample - loss: 3.1349 - val\_loss: 3.3778

Epoch 29/50

88513/88513 [==============================] - 410s 5ms/sample - loss: 3.1188 - val\_loss: 3.3637

Epoch 30/50

88513/88513 [==============================] - 410s 5ms/sample - loss: 3.1000 - val\_loss: 3.3544

Epoch 31/50

88513/88513 [==============================] - 413s 5ms/sample - loss: 3.0844 - val\_loss: 3.3481

Epoch 32/50

88513/88513 [==============================] - 411s 5ms/sample - loss: 3.0680 - val\_loss: 3.3407

Epoch 33/50

88513/88513 [==============================] - 410s 5ms/sample - loss: 3.0531 - val\_loss: 3.3374

Epoch 34/50

88513/88513 [==============================] - 410s 5ms/sample - loss: 3.0377 - val\_loss: 3.3314

Epoch 35/50

88513/88513 [==============================] - 408s 5ms/sample - loss: 3.0214 - val\_loss: 3.3186

Epoch 36/50

88513/88513 [==============================] - 409s 5ms/sample - loss: 3.0041 - val\_loss: 3.3128

Epoch 37/50

88513/88513 [==============================] - 410s 5ms/sample - loss: 2.9900 - val\_loss: 3.3195

Epoch 38/50

88513/88513 [==============================] - 407s 5ms/sample - loss: 2.9784 - val\_loss: 3.3007

Epoch 39/50

88513/88513 [==============================] - 408s 5ms/sample - loss: 2.9655 - val\_loss: 3.2975

Epoch 40/50

88513/88513 [==============================] - 410s 5ms/sample - loss: 2.9547 - val\_loss: 3.2889

Epoch 41/50

88513/88513 [==============================] - 408s 5ms/sample - loss: 2.9424 - val\_loss: 3.2923

Epoch 42/50

88513/88513 [==============================] - 409s 5ms/sample - loss: 2.9331 - val\_loss: 3.2753

Epoch 43/50

88513/88513 [==============================] - 411s 5ms/sample - loss: 2.9196 - val\_loss: 3.2847

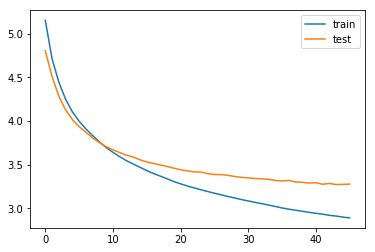
Epoch 44/50

88513/88513 [==============================] - 409s 5ms/sample - loss: 2.9111 - val\_loss: 3.2718

Epoch 45/50

50688/88513 [================>.............] - ETA: 2:48 - loss: 2.8809

Next, plot the training and validation loss metrics observed during the training phase.



### *Step 9: Generating Predictions:-*

Now that we've trained the model, to generate summaries from the given pieces of text, first reverse map the indices to the words (which has been previously generated using texts\_to\_sequences in Step 5). Also, map the words to indices from the summaries tokenizer which is to be used to detect the start and end of the sequences.

reverse\_target\_word\_index = y\_tokenizer.index\_word

reverse\_source\_word\_index = x\_tokenizer.index\_word

target\_word\_index = y\_tokenizer.word\_index

Now define the encoder and decoder inference models to start making the predictions. Use tensorflow.keras.Model() object to create your inference models.

An encoder inference model accepts text and returns the output generated from the three LSTMs, and hidden and cell states. A decoder inference model accepts the start of the sequence identifier (sostok) and predicts the upcoming word, eventually leading to predicting the whole summary.

Add the following code to define the inference models' architecture:-

# Inference Models

# Encode the input sequence to get the feature vector

encoder\_model = Model(inputs=encoder\_inputs, outputs=[encoder\_outputs,

state\_h, state\_c])

# 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 from the previous time step

(decoder\_outputs2, state\_h2, state\_c2) = decoder\_lstm(dec\_emb2,

initial\_state=[decoder\_state\_input\_h, decoder\_state\_input\_c])

# A dense softmax layer to generate prob dist. over the target vocabulary

decoder\_outputs2 = decoder\_dense(decoder\_outputs2)

# Final decoder model

decoder\_model = Model([decoder\_inputs] + [decoder\_hidden\_state\_input,

decoder\_state\_input\_h, decoder\_state\_input\_c],

[decoder\_outputs2] + [state\_h2, state\_c2])

Now define a function decode\_sequence() which accepts the input text and outputs the predicted summary. Start with sostok and continue generating words until eostok is encountered or the maximum length of the summary is reached. Predict the upcoming word from a given word by choosing the word which has the maximum probability attached and update the internal state of the decoder accordingly.

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\_c])

# 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 the stop word.

if sampled\_token == 'eostok' or len(decoded\_sentence.split()) \

>= max\_summary\_len - 1:

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

Define two functions - seq2summary() and seq2text() which convert numeric-representation to string-representation of summary and text respectively.

# To convert sequence to summary

def seq2summary(input\_seq):

newString = ''

for i in input\_seq:

if i != 0 and i != target\_word\_index['sostok'] and i \

!= target\_word\_index['eostok']:

newString = newString + reverse\_target\_word\_index[i] + ' '

return newString

# To convert sequence to text

def seq2text(input\_seq):

newString = ''

for i in input\_seq:

if i != 0:

newString = newString + reverse\_source\_word\_index[i] + ' '

return newString

Finally, generate the predictions by sending in the text:-

for i in range(0, 19):

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'

Here are a few notable summaries generated by the RNN model:-

# Output

Review: us president donald trump on wednesday said that north korea has returned the remains of 200 us troops missing from the korean war although there was no official confirmation from military authorities north korean leader kim jong un had agreed to return the remains during his summit with trump about 700 us troops remain unaccounted from the 1950 1953 korean war

Original summary: start n korea has returned remains of 200 us war dead trump end

Predicted summary: start n korea has lost an war against us trump end

Review: pope francis has said that history will judge those who refuse to accept the science of climate change if someone is doubtful that climate change is true they should ask scientists the pope added notably us president donald trump who believes global warming is chinese conspiracy withdrew the country from the paris climate agreement

Original summary: start history will judge those denying climate change pope end

Predicted summary: start pope francis will be in paris climate deal prez end

Review: the enforcement directorate ed has attached assets worth over ã¢ââ¹33 500 crore in the over three year tenure of its chief karnal singh who retires sunday officials said the agency filed around 390 in connection with its money laundering probes during the period the government on saturday appointed indian revenue service irs officer sanjay kumar mishra as interim ed chief

Original summary: start enforcement attached assets worth ã¢ââ¹33 500 cr in yrs end

Predicted summary: start ed attaches assets worth 100 crore in india in days end

Review: lok janshakti party president ram vilas paswan daughter asha has said she will contest elections against him from constituency if given ticket from lalu prasad yadav rjd she accused him of neglecting her and promoting his son chirag asha is paswan daughter from his first wife while chirag is his son from his second wife

Original summary: start will contest against father ram vilas from daughter end

Predicted summary: start lalu son tej pratap to contest his daughter in 2019 end

Review: irish deputy prime minister frances fitzgerald announced her resignation on tuesday in bid to avoid the collapse of the government and potential snap election she quit hours before no confidence motion was to be proposed against her by the main opposition party the political crisis began over fitzgerald role in police whistleblower scandal

Original summary: start irish deputy prime minister resigns to avoid govt collapse end

Predicted summary: start pmo resigns from punjab to join nda end

Review: rr wicketkeeper batsman jos buttler slammed his fifth straight fifty in ipl 2018 on sunday to equal former indian cricketer virender sehwag record of most straight 50 scores in the ipl sehwag had achieved the feat while representing dd in the ipl 2012 buttler is also only the second batsman after shane watson to hit two successive 90 scores in ipl

Original summary: start buttler equals sehwag record of most straight 50s in ipl end

Predicted summary: start sehwag slams sixes in an ipl over 100 times in ipl end

Review: maruti suzuki india on wednesday said it is recalling 640 units of its super carry mini trucks sold in the domestic market over possible defect in fuel pump supply the recall covers super carry units manufactured between january 20 and july 14 2018 the faulty parts in the affected vehicles will be replaced free of cost the automaker said n

Original summary: start maruti recalls its mini trucks over fuel pump issue in india end

Predicted summary: start maruti suzuki recalls india over ã¢ââ¹3 crore end

Review: the arrested lashkar e taiba let terrorist aamir ben has confessed to the national investigation agency that pakistani army provided him cover firing to infiltrate into india he further revealed that hafiz organisation ud dawah arranged for his training and that he was sent across india to carry out subversive activities in and outside kashmir

Original summary: start pak helped me enter india arrested let terrorist to nia end

Predicted summary: start pak man who killed indian soldiers to enter kashmir end

Review: the 23 richest indians in the 500 member bloomberg billionaires index saw wealth erosion of 21 billion this year lakshmi mittal who controls the world largest steelmaker arcelormittal lost 5 6 billion or 29 of his net worth followed by sun pharma founder dilip shanghvi whose wealth declined 4 6 billion asia richest person mukesh ambani added 4 billion to his fortune

Original summary: start lakshmi mittal lost 10 bn in 2018 ambani added 4 bn end

Predicted summary: start india richest man lost billion in wealth in 2017 end

## Conclusion

The Encoder-Decoder Sequence-to-Sequence Model (LSTM) we built generated acceptable summaries from what it learned in the training texts. Although after 50 epochs the predicted summaries are not exactly on par with the expected summaries (our model hasn't yet reached human-level intelligence!), the intelligence our model has gained definitely counts for something.

To attain more accurate results from this model, you can increase the size of the dataset, play around with the hyperparameters of the network, try making it larger, and increase the number of epochs.

In this tutorial, you’ve trained an encoder-decoder sequence-to-sequence model to perform text summarization. In my next article you can learn all about attention mechanisms. Until then, happy learning!

Made By:- Prakhar Agrawal

ML Intern at Suvidha Foundation.