TEXT SUMMARISATION USING DEEP LEARNING

Name-Sandeep Kumar Aazad
Roll no-521230

Table of Content:

Introduction:

• Text summarisation is the process of making a synopsis from a given text document while keeping the important information and meaning of it.

•Automatic summarisation has become an essential method for accurately locating significant information in vast amounts of text in a short amount of time and with minimal effort.

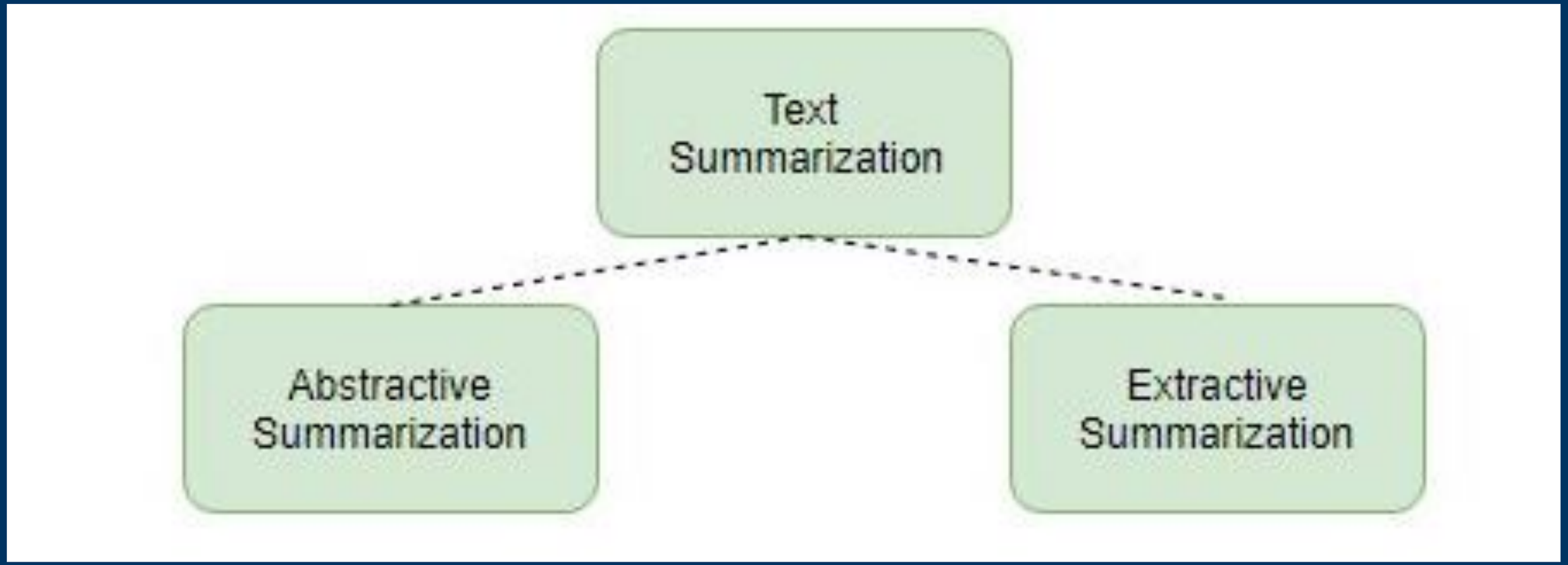
•Text summarisation is one of the Natural Language Processing (NLP) applications that will undoubtedly have a significant influence on our •With the rise of digital media and ever-increasing publication, who has the time to read complete news items, documents, books to determine whether they are beneficial or not.

•Automatic Text Summarisation is among the most complex and intriguing topics in Natural Language Processing (NLP).

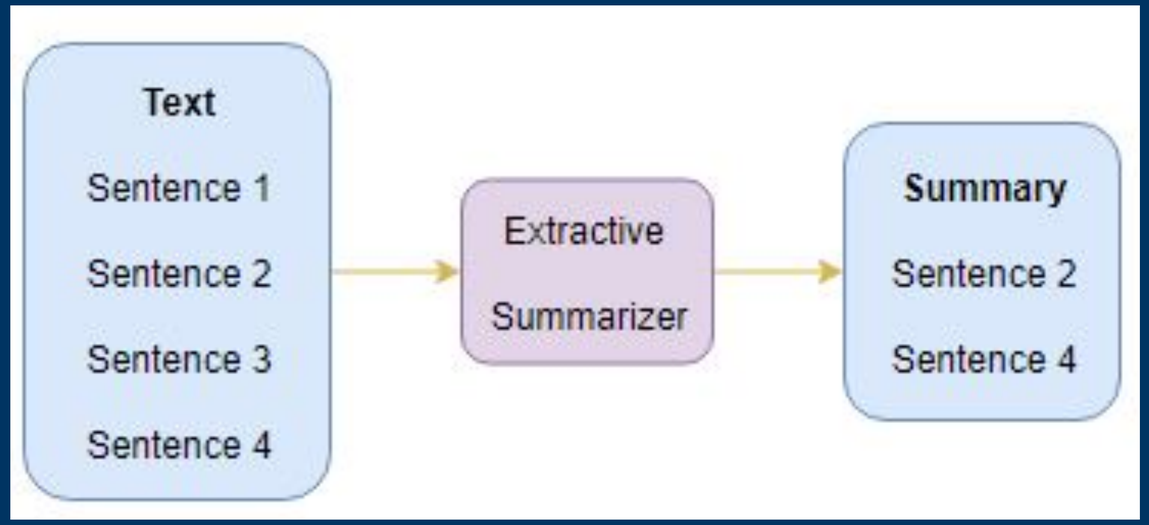
•It is the way of forming a brief and coherent summary of writing from a variety of text sources, including books, news stories, blog posts, research papers, emails, and tweets.

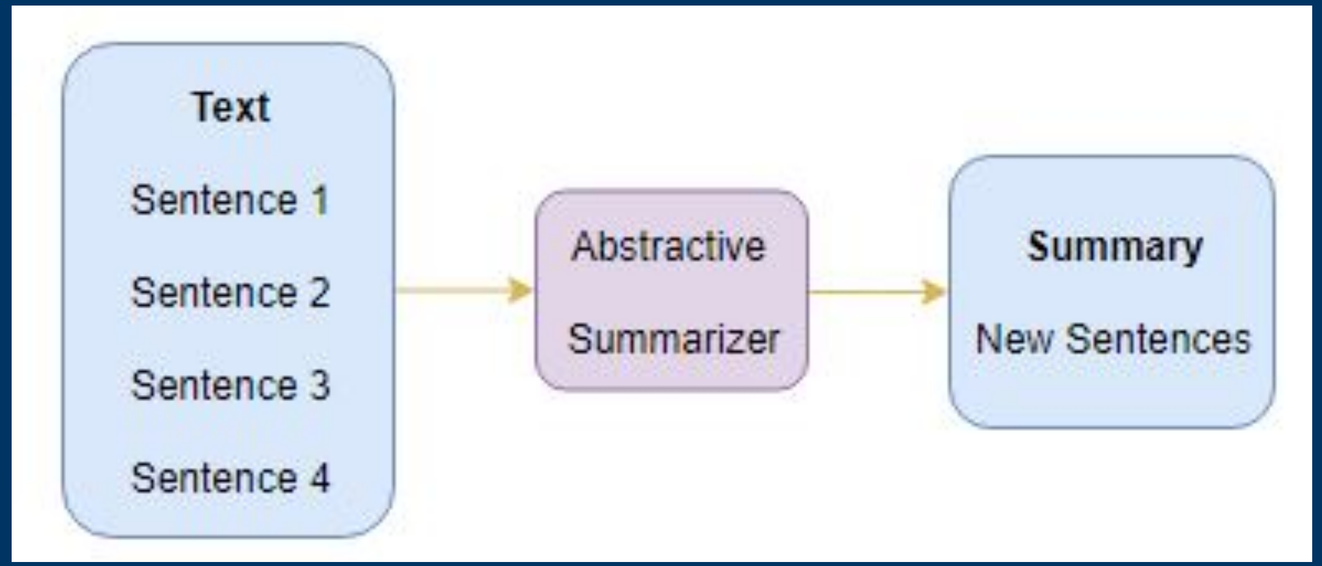
•The advent of vast volumes of textual data is driving up demand for automatic summarisation technologies.

- There are broadly two different approaches that are used for text summarisation:
- 1.Extractive Summarisation
- Abstractive Summarisation



- **Extractive Summarisation :-** The name gives away what this approach does. We identify the important sentences or phrases from the original text and extract only those from the text. Those extracted sentences would be our summary. The below diagram illustrates extractive summarisation.
- **Abstractive Summarisation :-** This is a very interesting approach. Here, we generate new sentences from the original text. This is in contrast to the extractive approach we saw earlier where we used only the sentences that were present. The sentences generated through abstractive summarization might not be present in the original text:





- The Encoder-Decoder architecture is mainly used to solve the sequence-to-sequence (Seq2Seq) problems where the input and output sequences are of different lengths.
- Generally, variants of Recurrent Neural Networks (RNNs), i.e. Gated Recurrent Neural Network (GRU) or Long Short Term Memory (LSTM), are preferred as the encoder and decoder components.
- This is because they are capable of capturing long term dependencies by overcoming the problem of vanishing gradient.
- We can set up the Encoder-Decoder in 2 phases:
- Training phase
- Inference phase

Training phase

In the training phase, we will first set up the encoder and decoder. We will then train the model to predict the target sequence offset by one time step. Let us see in detail on how to set up the encoder and decoder.

1.Encoder :- An Encoder Long Short Term Memory model (LSTM) reads the entire input sequence wherein, at each time step, one word is fed into the encoder. It then processes the information at every time step and captures the contextual information present in the input sequence. The hidden state (hi) and cell state (ci) of the last time step are used to initialise the decoder. Remember, this is because the encoder and decoder are two different sets of the LSTM architecture.

2. Decoder :- The decoder is also an LSTM network which reads the entire target sequence word-by-word and predicts the same sequence offset by one time step. The decoder is trained to predict the next word in the sequence given the previous word. decoder <start> and <end> are the special tokens which are added to the target sequence before feeding it into the decoder.

The target sequence is unknown while decoding the test sequence. So, we start predicting the target sequence by passing the first word into the decoder which would be always the <start> token. And the <end> token signals the end of the sentence.

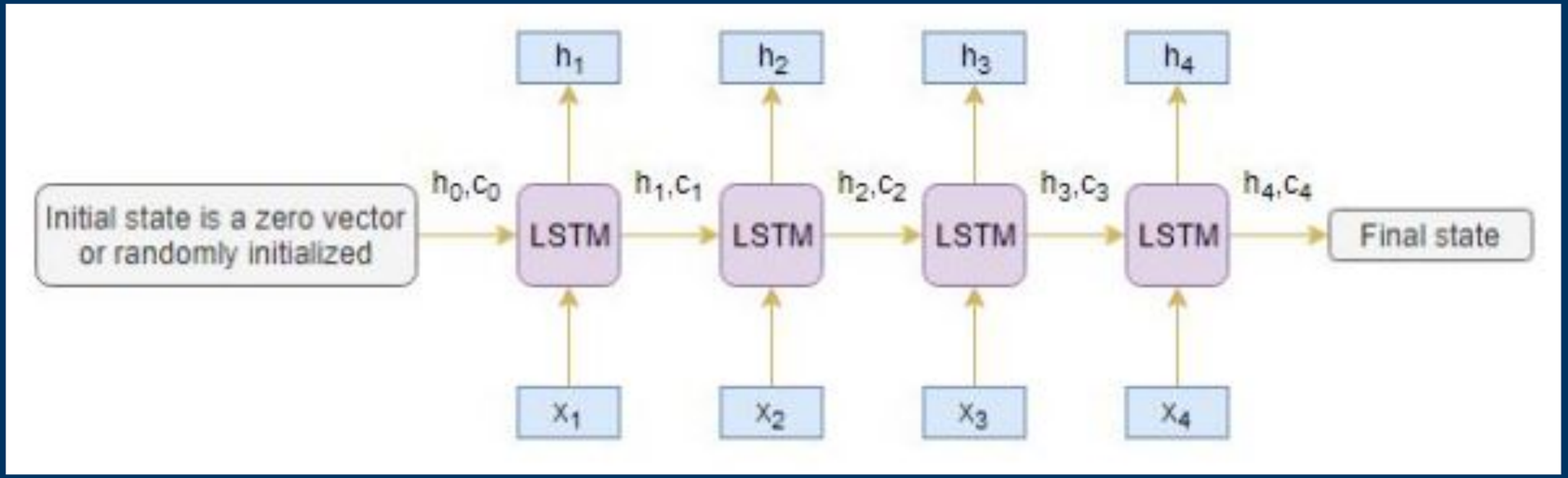
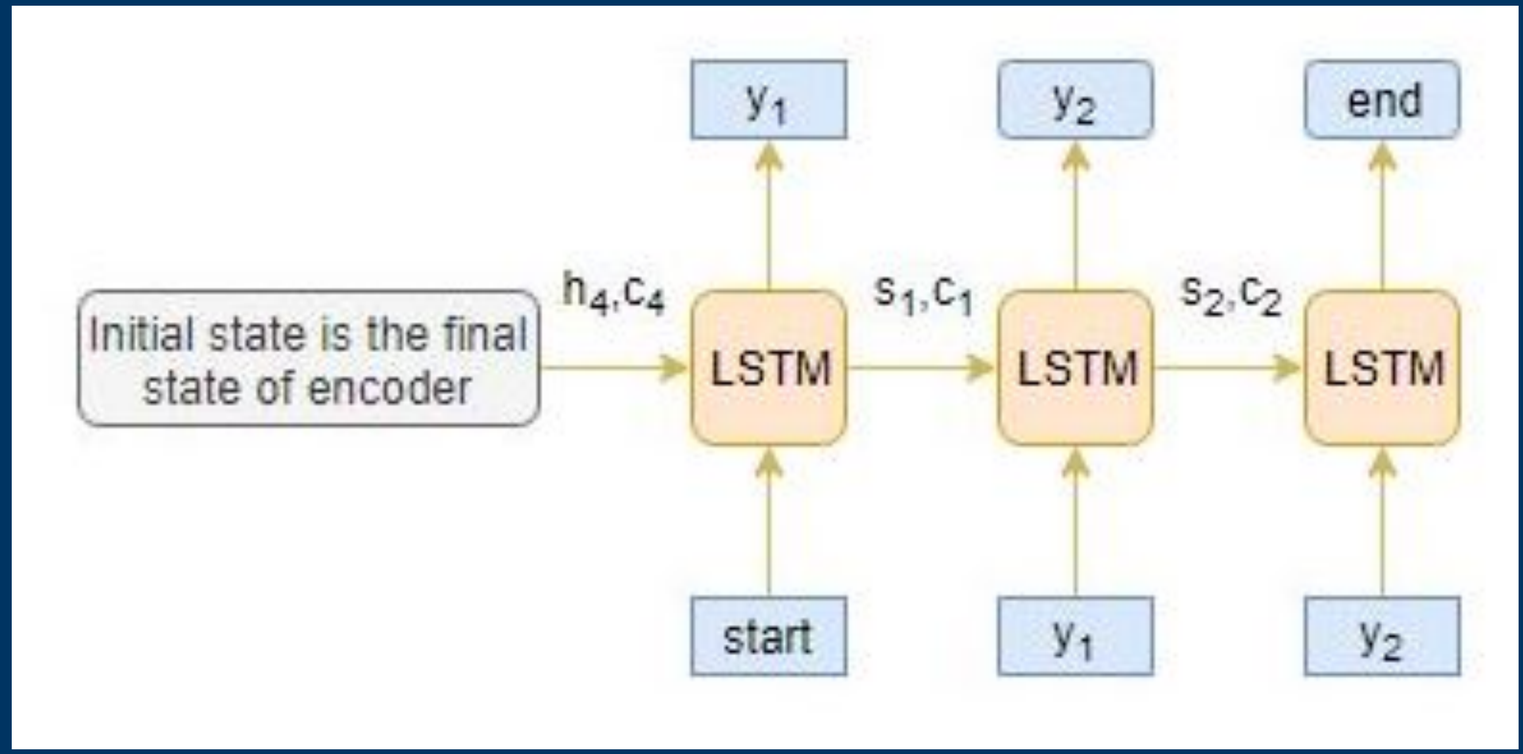


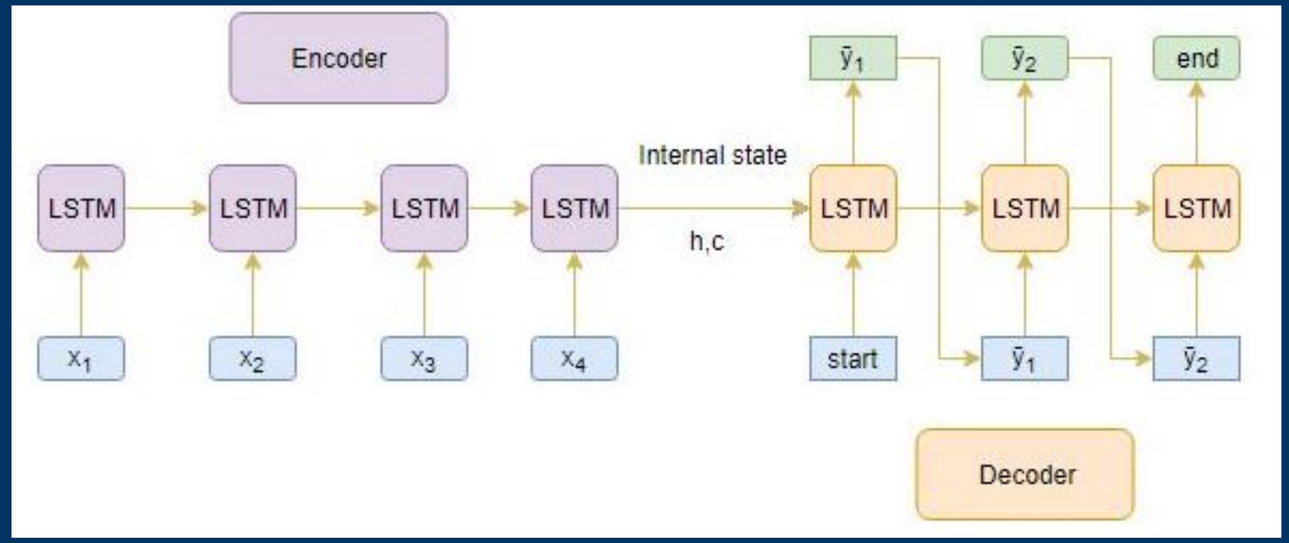
Fig: Encoder

Fig: Decoder



Interference Phase :

•After training, the model is tested on new source sequences for which the target sequence is unknown. So, we need to set up the inference architecture to decode a test sequence:



•How does it work?

1.Encode the entire input sequence and initialise the decoder with internal states of the encoder.

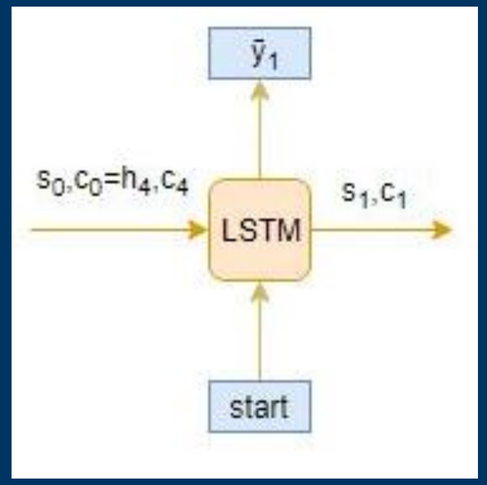
2.Pass <start> token as an input to the decoder.

3.Run the decoder for one time step with the internal states.

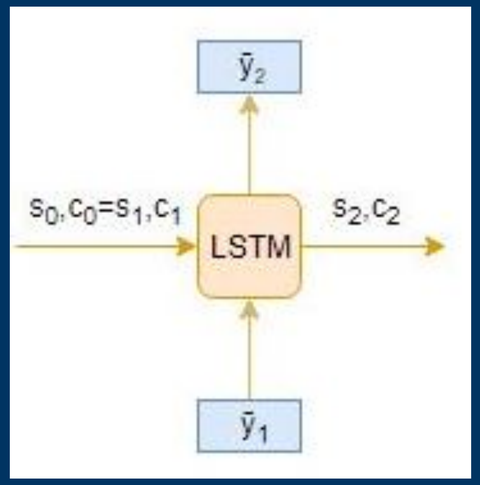
4.The output will be the probability for the next word. The word with the maximum probability will be selected.

5.Pass the sampled word as an input to the decoder in the next timestep and update the internal states with the current time step.

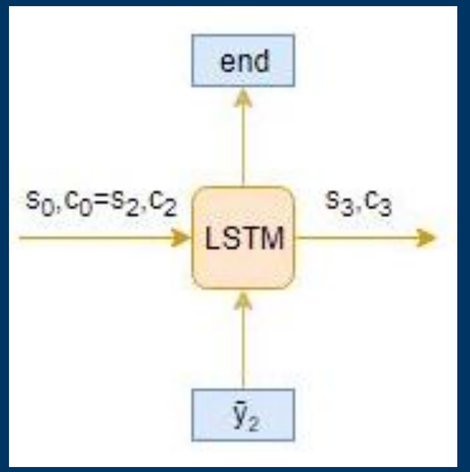
6.Repeat steps 3 – 5 until we generate <end> token or hit the maximum length of the target sequence.



Timestep: t=1



Timestep: t=2



Timestep: t=3

•Limitation of encoder and decoder:

1.The encoder converts the entire input sequence into a fixed length vector and then the decoder predicts the output sequence. This works only for short sequences since the decoder is looking at the entire input sequence for the prediction

2.Here comes the problem with long sequences. It is difficult for the encoder to memorize long sequences into a fixed length vector.

•Explain the model:

1.Encoder:

•The input sequence is passed through an embedding layer, which converts each word index into a dense vector representation. The embedded sequence is then fed into a stack of LSTM layers.
•Each LSTM layer processes the entire input sequence and produces a sequence of outputs.
•The final LSTM layer in the encoder returns its outputs and internal states (state_h and state_c), which represent the learned representation of the input sequence.

```
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=0.4,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 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=0.4,recurrent_dropout=0.2)
decoder_outputs,decoder_fwd_state, decoder_back_state = decoder_lstm(dec_emb,initial_state=[state_h, state_c])

# 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, attn_out])

#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()
```

2. Decoder:

•The decoder takes two inputs: the target sequence (decoder_inputs) and the internal states from the encoder LSTM. Similar to the encoder, the target sequence is first embedded.

•The embedded sequence is then fed into an LSTM layer. This LSTM layer generates a sequence of outputs while also utilizing the internal states from the encoder.

•Additionally, an attention mechanism is applied here. This mechanism allows the decoder to focus on different parts of the input sequence dynamically during the decoding process.

• The attention layer calculates attention weights based on the current decoder state and the encoder outputs. The attention mechanism's output is concatenated with the decoder LSTM's output, creating a context vector.

• Finally, a dense layer with softmax activation is applied to produce the probability distribution over the vocabulary for the next word in the sequence.

3.Model:

•The model is instantiated using the Model class from Keras, taking encoder and decoder inputs and producing decoder outputs.

•This model has a total of three inputs: the encoder input sequence, the decoder input sequence, and the initial states from the encoder LSTM.

• The decoder outputs represent the probability distribution of each word in the target vocabulary for each timestep in the output sequence.

Code Walkthrough:

This code implements a sequence-to-sequence model with attention mechanism for text summarization using Keras and TensorFlow. Let's break down the code:

1. **Data Preprocessing**:
   - The code starts by importing necessary libraries and reading a dataset (Amazon Fine Food Reviews) containing text reviews and corresponding summaries.
   - It preprocesses the text data by removing duplicates, NaN values, HTML tags, special characters, and stopwords. It also converts contractions to their expanded forms.
   - The dataset is then split into text and summary pairs.
2. **Tokenization and Padding**:
   - The text and summary sequences are tokenized using Keras's Tokenizer class, and rare words are removed.
   - Padding is applied to ensure all sequences have the same length.
3. **Model Building**:
   - The model architecture is defined using Keras's functional API.
   - The encoder consists of an embedding layer followed by three LSTM layers.
   - The decoder includes an embedding layer and an LSTM layer, followed by an attention layer and a dense layer for output.
   - The model is compiled using the RMSprop optimizer and sparse categorical cross-entropy loss.
4. **Training**:
   - The model is trained on the preprocessed data with early stopping to prevent overfitting.
   - Training history is plotted to visualise the training and validation loss.
5. **Inference**:
   - Inference is set up to generate summaries for new text inputs.
   - An encoder model is created to encode the input text and obtain the encoder states.
   - A decoder model is created to generate summaries using the encoder states and attention mechanism.
   - Functions are defined to decode sequences, convert sequences to summaries, and convert sequences to text.
6. **Generating Summaries**:
   - Finally, the code generates summaries for a subset of the training data and prints them alongside the original text and summaries.

text_text_summarizer_.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Comment Share

+ Code + Text Connect Colab AI

Import the Libraries

```
!pip install keras
!pip install pandas
!pip install tensorflow
from tensorflow.keras.preprocessing.text import Tokenizer
```

```
Requirement already satisfied: keras in /usr/local/python/3.10.13/lib/python3.10/site-packages (3.2.1)
Requirement already satisfied: absl-py in /usr/local/python/3.10.13/lib/python3.10/site-packages (from keras) (2.1.0)
Requirement already satisfied: numpy in /home/codespace/.local/lib/python3.10/site-packages (from keras) (1.26.4)
Requirement already satisfied: rich in /usr/local/python/3.10.13/lib/python3.10/site-packages (from keras) (13.7.1)
Requirement already satisfied: namex in /usr/local/python/3.10.13/lib/python3.10/site-packages (from keras) (0.0.8)
Requirement already satisfied: h5py in /usr/local/python/3.10.13/lib/python3.10/site-packages (from keras) (3.11.0)
Requirement already satisfied: optree in /usr/local/python/3.10.13/lib/python3.10/site-packages (from keras) (0.11.0)
Requirement already satisfied: ml-dtypes in /usr/local/python/3.10.13/lib/python3.10/site-packages (from keras) (0.3.2)
Requirement already satisfied: typing-extensions>=4.0.0 in /home/codespace/.local/lib/python3.10/site-packages (from optree->keras) (4.10.0)
Requirement already satisfied: markdown-it-py>=2.2.0 in /usr/local/python/3.10.13/lib/python3.10/site-packages (from rich->keras) (3.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in /home/codespace/.local/lib/python3.10/site-packages (from rich->keras) (2.17.2)
Requirement already satisfied: mdurl~=0.1 in /usr/local/python/3.10.13/lib/python3.10/site-packages (from markdown-it-py>=2.2.0->rich->keras) (0.1.2)
Requirement already satisfied: pandas in /home/codespace/.local/lib/python3.10/site-packages (2.2.1)
Requirement already satisfied: numpy<2,>=1.22.4 in /home/codespace/.local/lib/python3.10/site-packages (from pandas) (1.26.4)
Requirement already satisfied: python-dateutil>=2.8.2 in /home/codespace/.local/lib/python3.10/site-packages (from pandas) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in /home/codespace/.local/lib/python3.10/site-packages (from pandas) (2024.1)
Requirement already satisfied: tzdata>=2022.7 in /home/codespace/.local/lib/python3.10/site-packages (from pandas) (2024.1)
Requirement already satisfied: six>=1.5 in /home/codespace/.local/lib/python3.10/site-packages (from python-dateutil>=2.8.2->pandas) (1.16.0)
Requirement already satisfied: tensorflow in /usr/local/python/3.10.13/lib/python3.10/site-packages (2.16.1)
Requirement already satisfied: absl-py>=1.0.0 in /usr/local/python/3.10.13/lib/python3.10/site-packages (from tensorflow) (2.1.0)
Requirement already satisfied: astunparse>=1.6.0 in /usr/local/python/3.10.13/lib/python3.10/site-packages (from tensorflow) (1.6.3)
Requirement already satisfied: flatbuffers>=23.5.26 in /usr/local/python/3.10.13/lib/python3.10/site-packages (from tensorflow) (24.3.25)
Requirement already satisfied: gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 in /usr/local/python/3.10.13/lib/python3.10/site-packages (from tensorflow) (0.5.4)
Requirement already satisfied: google-pasta>=0.1.1 in /usr/local/python/3.10.13/lib/python3.10/site-packages (from tensorflow) (0.2.0)
```

THANK YOU