What is Text Summarization in NLP?

Automatic text summarization is the task of producing a concise and fluent summary while preserving key information content and overall meaning".

There are broadly two different approaches that are used for text summarization:

- Extractive Summarization
- Abstractive Summarization

Extractive Summarization

- 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.

Scoring and Sentences Selection

once we get the intermediate representations, we move to assign some scores to each sentence to specify their importance. For topic representations, a score to a sentence depends on the topic words it contains, and for an indicator representation, the score depends on the features of the sentences. Finally, the sentences having top scores, are picked and used to generate a summary.

The summarizer system selects the top *k* most important sentences to produce a summary. Some approaches use greedy algorithms to select the important sentences and some approaches may convert the selection of sentences into an optimization problem where a collection of sentences is chosen, considering the constraint that it should maximize overall importance and coherency and minimize the redundancy.

Abstractive Summarization

- 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.

There are two major components of a Seq2Seq model:

- Encoder
- Decoder

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.

Let's understand this from the perspective of text summarization. The input is a long sequence of words and the output will be a short version of the input sequence.

Extractive and Abstractive summarization

One approach to summarization is to extract parts of the document that are deemed interesting by some metric (for example, inverse-document frequency) and join them to form a summary. Algorithms of this flavor are called extractive summarization.

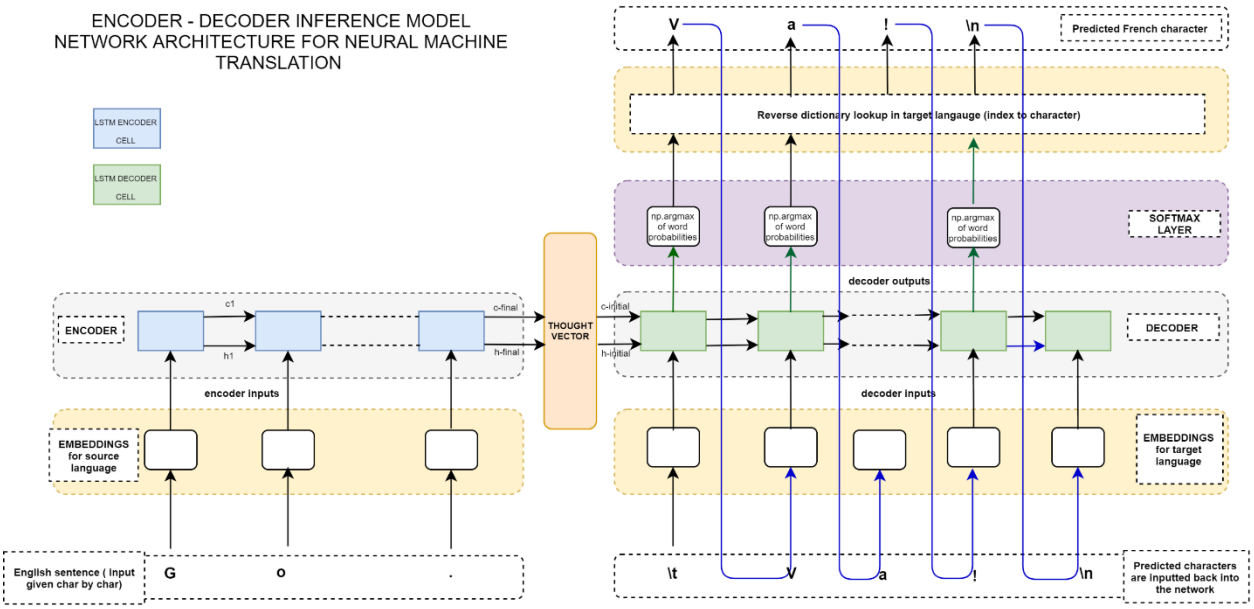
Original Text: Alice and Bob took the train to visit the zoo. They saw a baby giraffe, a lion, and a flock of colorful tropical birds. Extractive Summary: Alice and Bob visit the zoo. saw a flock of birds. Above we extract the words bolded in the original text and concatenate them to form a summary. As we can see, sometimes the extractive constraint can make the summary awkward or grammatically strange.

Another approach is to simply summarize as humans do, which is to not impose the extractive constraint and allow for rephrasings.

This is called abstractive summarization. Abstractive summary: Alice and Bob visited the zoo and saw animals and birds. In this example, we used words not in the original text, maintaining

more of the information in a similar amount of words. It’s clear we would prefer good abstractive summarizations, but how could an algorithm begin to do this?

Architecture overview



Explanation of Abstractive Extraction

Introduction to Seq2Seq Models

Seq2Seq Architecture and Applications

Text Summarization Using an Encoder-Decoder Sequence-to-Sequence Model

Step 1 - Importing the Dataset

Step 2 - Cleaning the Data

Step 3 - Determining the Maximum Permissible Sequence Lengths

Step 4 - Selecting Plausible Texts and Summaries

Step 5 - Tokenizing the Text

Step 6 - Removing Empty Text and Summaries

Step 7: Creating the Model

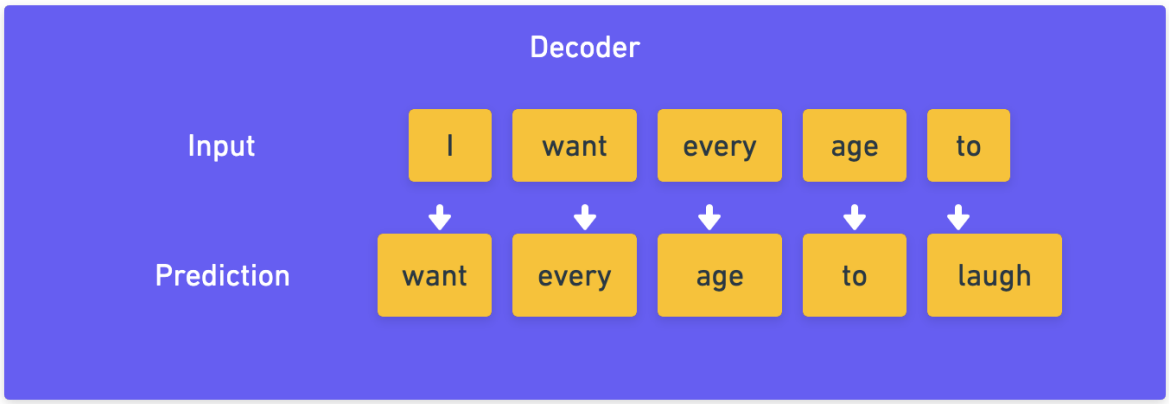
Encoder

The input length that the encoder accepts is equal to the maximum text length which you've already estimated in Step 3. 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; 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).



Consider the summary line to be "I want every age to laugh". The model has to accept two inputs - the actual text and the summary. During the training phase, the decoder accepts the input summary given to the model, and learns every word that has to follow a certain given word. It then generates the predictions using an inference model during the test phase.

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.

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.

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.

ow 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.

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.

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

Finally, generate the predictions by sending in the text.