**CODE:**

**import numpy as np**

**import pandas as pd**

To perform a wide variety of mathematical operations on arrays adds powerful data structures to Python

Offers data structures like DataFrames and Series, provides efficient data manipulation and analysis.

**import os**

**import string**

**from string import digits**

**import matplotlib.pyplot as plt**

**%matplotlib inline**

**import re**

1. import os: Imports the 'os' module, provides functions for **interacting with the operating system**
2. import string: This imports the 'string' module, which contains a **collection of string constants** and functions **used in string manipulations**.
3. from string import digits: This **imports only the 'digits'** constant from the 'string' module. 'digits' is a string containing all ASCII digits, which can be useful in certain text processing tasks.
4. import matplotlib.pyplot as plt: This imports the 'pyplot' module from the 'matplotlib' library, used for creating  **visualizations in Python.**
5. %matplotlib inline: Used in Jupyter notebooks to display matplotlib **plots directly in the notebook**, rather than in a separate window.
6. import re: This imports the 're' module, which **provides regular expression matching operations**.

**import seaborn as sns**

**from sklearn.utils import shuffle**

**from sklearn.model\_selection import train\_test\_split**

**from keras.layers import Input, LSTM, Embedding, Dense**

**from keras.models import Model**

 Seaborn is a data visualization library based on Matplotlib. It provides a high-level interface for **creating attractive and informative statistical graphics**.

 from sklearn.utils import shuffle: The shuffle function is used for shuffling arrays or lists, which can be helpful for **randomizing the order of data samples.**

 from sklearn.model\_selection import train\_test\_split: It's commonly used to **split datasets into training and testing sets**, facilitating the evaluation of machine learning models.

 from keras.layers import Input, LSTM, Embedding, Dense: These imports are related to Keras, a high-level neural networks API. Here's what each of these components does:

* Input: This is used to define the input layer for the neural network.
* LSTM: Long Short-Term Memory (LSTM) is a type of recurrent neural network (RNN) layer, often used for processing sequences of data.
* Embedding: This layer is used for word embeddings, converting integer indices into dense vectors. It's commonly used in natural language processing tasks.
* Dense: This is a standard fully connected neural network layer.

 from keras.models import Model:

This import brings in the Model class from Keras, which is used to **create and manage models composed of layers.**

**pd.set\_option('display.max\_rows', 500)**

**pd.set\_option('display.max\_columns', 500)**

**pd.set\_option('display.width', 1000)**

**pd.set\_option('display.max\_colwidth', -1)**

1. pd.set\_option('display.max\_rows', 500): This sets the maximum number of rows to display when printing a DataFrame to 500. If a DataFrame has more than 500 rows, Pandas will truncate the display, showing only the first and last rows, and indicating the truncation with ellipses (...).
2. pd.set\_option('display.max\_columns', 500): Similarly, this sets the maximum number of columns to display when printing a DataFrame to 500. If a DataFrame has more than 500 columns, Pandas will truncate the display, showing only the first and last columns.
3. pd.set\_option('display.width', 1000): This sets the maximum width of the display in characters. It ensures that the printed DataFrame does not wrap across multiple lines, making it easier to read in a console or Jupyter notebook.
4. pd.set\_option('display.max\_colwidth', -1): This sets the maximum width of each column in characters to -1, indicating that there should be no maximum width. This means that Pandas will display the full content of each cell in the DataFrame without truncation.

**lines=pd.read\_csv(r"C:\Users\hrsht\Desktop\Language\_translator\_project\Hindi\_English\_Truncated\_Corpus.csv\Hindi\_English\_Truncated\_Corpus.csv",encoding='utf-8')**

1. pd.read\_csv(...): This uses the read\_csv function from the Pandas library to read data from a CSV (Comma-Separated Values) file. The file path is specified as the argument. The r before the string indicates a raw string, which is often used in file paths to handle backslashes correctly. The encoding='utf-8' parameter specifies the character encoding of the file, which is set to UTF-8 in this case.
2. lines: This is a variable that now holds the DataFrame created from the CSV file. The DataFrame is essentially a two-dimensional table that holds the data from the CSV file, and lines is the name given to this DataFrame.

**lines['source'].value\_counts()**

**tides 50000**

**ted 39881**

**indic2012 37726**

**Name: source, dtype: int64**

1. lines['source']: This extracts the column named 'source' from the DataFrame lines. It assumes that there is a column named 'source' in the DataFrame.
2. .value\_counts(): This Pandas method is applied to the 'source' column, and it returns a Series containing counts of unique values. In other words, it tells you how many occurrences there are for each unique value in the 'source' column.

**lines=lines[lines['source']=='ted']**

reak it down:

1. lines['source'] == 'ted': This creates a boolean mask. It checks for each row in the 'source' column whether the value is equal to 'ted'. The result is a Series of boolean values, where True indicates that the condition is met, and False indicates that it is not.
2. lines[...]: This uses the boolean mask to filter the rows of the DataFrame. Only the rows where the condition ('source' == 'ted') is True are retained in the DataFrame.
3. lines = ...: This assigns the filtered DataFrame back to the variable lines, effectively updating it with only the rows where the 'source' column has the value 'ted'.

**pd.isnull(lines).sum()**

check the null values

**lines=lines[~pd.isnull(lines['english\_sentence'])]**

This creates a boolean mask that checks for each row in the 'english\_sentence' column whether the value is null (NaN or missing). The result is a Series of boolean values.

**lines.drop\_duplicates(inplace=True)**

drops the dulplicate values

**lines=lines.sample(n=25000,random\_state=42)**

**lines.shape**

**lines['english\_sentence']=lines['english\_sentence'].apply(lambda x: x.lower())**

**lines['hindi\_sentence']=lines['hindi\_sentence'].apply(lambda x: x.lower())**

:

1. lines['english\_sentence']: This selects the 'english\_sentence' column of the DataFrame lines.
2. .apply(lambda x: x.lower()): The apply method is used to apply a function to each element in the selected column. In this case, the function is a lambda function that converts the text to lowercase using the lower() method.
3. lines['hindi\_sentence']: Similarly, this selects the 'hindi\_sentence' column of the DataFrame lines.
4. .apply(lambda x: x.lower()): Applies the same lambda function to convert each element in the 'hindi\_sentence' column to lowercase.

**lines['english\_sentence']=lines['english\_sentence'].apply(lambda x: re.sub("'", '', x))**

**lines['hindi\_sentence']=lines['hindi\_sentence'].apply(lambda x: re.sub("'", '', x))**

1. lines['english\_sentence']: This selects the 'english\_sentence' column of the DataFrame lines.
2. .apply(lambda x: re.sub("'", '', x)): The apply method is used to apply a function to each element in the selected column. In this case, the function is a lambda function that utilizes re.sub() to replace (substitute) single quotes with an empty string in each sentence.
3. lines['hindi\_sentence']: Similarly, this selects the 'hindi\_sentence' column of the DataFrame lines.
4. .apply(lambda x: re.sub("'", '', x)): Applies the same lambda function to replace single quotes in each element of the 'hindi\_sentence' column.

**exclude = set(string.punctuation)**

:

1. string.punctuation: This is a predefined string in the string module of Python that contains all ASCII punctuation characters. Punctuation characters include symbols such as periods, commas, exclamation marks, question marks, and other special characters.
2. set(...): This converts the string of punctuation characters into a set. A set is an unordered collection of unique elements in Python. Using a set ensures that each punctuation character appears only once, as sets do not allow duplicate elements.
3. exclude = ...: This assigns the created set of punctuation characters to the variable exclude.

**lines['english\_sentence']=lines['english\_sentence'].apply(lambda x: ''.join(ch for ch in x if ch not in exclude))**

**lines['hindi\_sentence']=lines['hindi\_sentence'].apply(lambda x: ''.join(ch for ch in x if ch not in exclude))**

1. lines['english\_sentence']: This selects the 'english\_sentence' column of the DataFrame lines.
2. .apply(lambda x: ''.join(ch for ch in x if ch not in exclude)): The apply method is used to apply a function to each element in the selected column. In this case, the function is a lambda function that iterates through each character (ch) in the sentence (x). The condition if ch not in exclude checks if the character is not in the set of punctuation characters (exclude). If the character is not a punctuation character, it is included in the result.
3. ''.join(...): This joins the characters together to form a new string without the excluded punctuation characters.
4. lines['hindi\_sentence']: Similarly, this selects the 'hindi\_sentence' column of the DataFrame lines.

**remove\_digits = str.maketrans('', '', digits)**

**lines['english\_sentence']=lines['english\_sentence'].apply(lambda x: x.translate(remove\_digits))**

**lines['hindi\_sentence']=lines['hindi\_sentence'].apply(lambda x: x.translate(remove\_digits))**

**lines['hindi\_sentence'] = lines['hindi\_sentence'].apply(lambda x: re.sub("[२३०८१५७९४६]", "", x))**

**lines['english\_sentence']=lines['english\_sentence'].apply(lambda x: x.strip())**

**lines['hindi\_sentence']=lines['hindi\_sentence'].apply(lambda x: x.strip())**

**lines['english\_sentence']=lines['english\_sentence'].apply(lambda x: re.sub(" +", " ", x))**

**lines['hindi\_sentence']=lines['hindi\_sentence'].apply(lambda x: re.sub(" +", " ", x))**

**lines['hindi\_sentence'] = lines['hindi\_sentence'].apply(lambda x : 'START\_ '+ x + ' \_END')  
  
  
  
s:**

python

 remove\_digits = str.maketrans('', '', digits)

lines['english\_sentence'] = lines['english\_sentence'].apply(lambda x: x.translate(remove\_digits))

lines['hindi\_sentence'] = lines['hindi\_sentence'].apply(lambda x: x.translate(remove\_digits))

* str.maketrans('', '', digits): This creates a translation table that can be used with the translate method to remove digits from a string.
* lines['english\_sentence'].apply(lambda x: x.translate(remove\_digits)): Applies the translation table to each element in the 'english\_sentence' column to remove digits.
* lines['hindi\_sentence'].apply(lambda x: x.translate(remove\_digits)): Applies the translation table to each element in the 'hindi\_sentence' column to remove digits.

 **Removing Specific Hindi Digits:**

python

 lines['hindi\_sentence'] = lines['hindi\_sentence'].apply(lambda x: re.sub("[२३०८१५७९४६]", "", x))

* re.sub("[२३०८१५७९४६]", "", x): This uses a regular expression to replace specific Hindi digits with an empty string in each element of the 'hindi\_sentence' column.

 **Stripping Whitespace:**

python

 lines['english\_sentence'] = lines['english\_sentence'].apply(lambda x: x.strip())

lines['hindi\_sentence'] = lines['hindi\_sentence'].apply(lambda x: x.strip())

* x.strip(): Removes leading and trailing whitespaces from each element in the 'english\_sentence' and 'hindi\_sentence' columns.

 **Removing Extra Spaces:**

python

 lines['english\_sentence'] = lines['english\_sentence'].apply(lambda x: re.sub(" +", " ", x))

lines['hindi\_sentence'] = lines['hindi\_sentence'].apply(lambda x: re.sub(" +", " ", x))

* re.sub(" +", " ", x): Uses a regular expression to replace multiple consecutive spaces with a single space in each element of the 'english\_sentence' and 'hindi\_sentence' columns.

 **Adding Start and End Tokens to Hindi Sentences:**

python

lines['hindi\_sentence'] = lines['hindi\_sentence'].apply(lambda x : 'START\_ '+ x + ' \_END')

* Concatenates 'START\_' at the beginning and '\_END' at the end of each element in the 'hindi\_sentence' column.

**all\_eng\_words=set()**

**for eng in lines['english\_sentence']:**

**for word in eng.split():**

**if word not in all\_eng\_words:**

**all\_eng\_words.add(word)**)

1. all\_eng\_words = set(): Initializes an empty set named all\_eng\_words to store unique English words.
2. for eng in lines['english\_sentence']:: Iterates over each element (sentence) in the 'english\_sentence' column of the DataFrame lines.
3. for word in eng.split():: Splits each sentence into individual words using the split() method, creating a list of words.
4. if word not in all\_eng\_words:: Checks if each word is not already in the all\_eng\_words set.
5. all\_eng\_words.add(word): If the word is not in the set, it is added to the all\_eng\_words set. This ensures that only unique words are added to the set.

**all\_hindi\_words=set()**

**for hin in lines['hindi\_sentence']:**

**for word in hin.split():**

**if word not in all\_hindi\_words:**

**all\_hindi\_words.add(word)**)

1. all\_hindi\_words = set(): Initializes an empty set named all\_hindi\_words to store unique Hindi words.
2. for hin in lines['hindi\_sentence']:: Iterates over each element (sentence) in the 'hindi\_sentence' column of the DataFrame lines.
3. for word in hin.split():: Splits each Hindi sentence into individual words using the split() method, creating a list of words.
4. if word not in all\_hindi\_words:: Checks if each word is not already in the all\_hindi\_words set.
5. all\_hindi\_words.add(word): If the word is not in the set, it is added to the all\_hindi\_words set. This ensures that only unique words are added to the set.

**lines['length\_eng\_sentence']=lines['english\_sentence'].apply(lambda x:len(x.split(" ")))**

**lines['length\_hin\_sentence']=lines['hindi\_sentence'].apply(lambda x:len(x.split(" ")))**)

1. lines['length\_eng\_sentence']: This creates a new column named 'length\_eng\_sentence' in the DataFrame lines to store the length (number of words) of each English sentence.
2. .apply(lambda x: len(x.split(" "))): The apply method is used to apply a function to each element in the 'english\_sentence' column. The function, defined as a lambda function, splits each sentence into words using the space as a delimiter (split(" ")) and then calculates the length of the resulting list of words using len().
3. lines['length\_hin\_sentence']: Similarly, this creates a new column named 'length\_hin\_sentence' to store the length of each Hindi sentence.
4. .apply(lambda x: len(x.split(" "))): Applies the same lambda function to calculate the length of each Hindi sentence in terms of the number of words.

**lines=lines[lines['length\_eng\_sentence']<=20]**

**lines=lines[lines['length\_hin\_sentence']<=20]**]

1. lines['length\_eng\_sentence'] <= 20: This condition creates a boolean mask for the DataFrame lines where each row is True if the length of the English sentence in that row is less than or equal to 20, and False otherwise.
2. lines[...]: This uses the boolean mask to filter the rows of the DataFrame. Only the rows where the condition ('length\_eng\_sentence' <= 20) is True are retained in the DataFrame. This operation effectively removes rows where the length of the English sentence exceeds 20 words.
3. lines['length\_hin\_sentence'] <= 20: Similarly, this condition creates a boolean mask for the DataFrame lines based on the length of Hindi sentences.
4. lines[...]: Again, this uses the boolean mask to filter the rows of the DataFrame. Only the rows where the condition ('length\_hin\_sentence' <= 20) is True are retained in the DataFrame. This operation effectively removes rows where the length of the Hindi sentence exceeds 20 words.

**print("maximum length of Hindi Sentence ",max(lines['length\_hin\_sentence']))**

**print("maximum length of English Sentence ",max(lines['length\_eng\_sentence']))**

))

1. max(lines['length\_hin\_sentence']): This calculates the maximum value in the 'length\_hin\_sentence' column of the DataFrame lines. It represents the maximum length (number of words) among all Hindi sentences in the filtered dataset.
2. max(lines['length\_eng\_sentence']): Similarly, this calculates the maximum value in the 'length\_eng\_sentence' column of the DataFrame lines. It represents the maximum length among all English sentences in the filtered dataset.
3. print("maximum length of Hindi Sentence ", ...): Prints the result of the maximum length calculation for Hindi sentences.
4. print("maximum length of English Sentence ", ...): Prints the result of the maximum length calculation for English sentences.

**max\_length\_src=max(lines['length\_hin\_sentence'])**

**max\_length\_tar=max(lines['length\_eng\_sentence'])**

)

1. max(lines['length\_hin\_sentence']): This calculates the maximum value in the 'length\_hin\_sentence' column of the DataFrame lines. It represents the maximum length (number of words) among all Hindi sentences in the filtered dataset.
2. max(lines['length\_eng\_sentence']): Similarly, this calculates the maximum value in the 'length\_eng\_sentence' column of the DataFrame lines. It represents the maximum length among all English sentences in the filtered dataset.
3. max\_length\_src = ...: Assigns the result of the maximum length calculation for Hindi sentences to the variable max\_length\_src.
4. max\_length\_tar = ...: Assigns the result of the maximum length calculation for English sentences to the variable max\_length\_tar.

**input\_words = sorted(list(all\_eng\_words))**

**target\_words = sorted(list(all\_hindi\_words))**

**num\_encoder\_tokens = len(all\_eng\_words)**

**num\_decoder\_tokens = len(all\_hindi\_words)**

**num\_encoder\_tokens, num\_decoder\_tokens**

)

1. input\_words = sorted(list(all\_eng\_words)): Creates a sorted list (input\_words) containing all unique English words. The sorted function ensures that the words are arranged in alphabetical order.
2. target\_words = sorted(list(all\_hindi\_words)): Similarly, creates a sorted list (target\_words) containing all unique Hindi words.
3. num\_encoder\_tokens = len(all\_eng\_words): Determines the number of unique English words (num\_encoder\_tokens) by calculating the length of the set all\_eng\_words.
4. num\_decoder\_tokens = len(all\_hindi\_words): Determines the number of unique Hindi words (num\_decoder\_tokens) by calculating the length of the set all\_hindi\_words.

**num\_decoder\_tokens += 1 #for zero padding**

the code is preparing the vocabulary size for the decoder (target language) by increasing it to accommodate the special token used for zero-padding. This is a common practice in sequence-to-sequence models to ensure uniform sequence lengths.

**input\_token\_index = dict([(word, i+1) for i, word in enumerate(input\_words)])**

**target\_token\_index = dict([(word, i+1) for i, word in enumerate(target\_words)])**

1. enumerate(input\_words): This function is used to iterate over each word in the input\_words list along with its index (i).
2. [(word, i+1) for i, word in enumerate(input\_words)]: This is a list comprehension that creates a list of tuples, where each tuple contains a word and its corresponding index plus 1. The +1 is used to start indexing from 1 instead of 0.
3. dict([...]): Converts the list of tuples into a dictionary. In this context, the words are the keys, and the indices are the values.
4. input\_token\_index: The resulting dictionary for English words where each word is mapped to a unique integer index.
5. A similar process is applied to target\_words, creating target\_token\_index for Hindi words.

**reverse\_input\_char\_index = dict((i, word) for word, i in input\_token\_index.items())**

**reverse\_target\_char\_index = dict((i, word) for word, i in target\_token\_index.items())**

1. input\_token\_index.items(): This retrieves the key-value pairs (word, index) from the input\_token\_index dictionary.
2. for word, i in input\_token\_index.items(): This iterates over each key-value pair, where word is the word, and i is its corresponding index.
3. dict((i, word) for word, i in input\_token\_index.items()): This constructs a new dictionary where the keys are the indices (i) and the values are the corresponding words (word), effectively reversing the mapping.
4. reverse\_input\_char\_index: The resulting dictionary for the input language, where each index is mapped back to its corresponding word.
5. A similar process is applied to target\_token\_index, creating reverse\_target\_char\_index for the target language.

**lines = shuffle(lines)**The purpose of shuffling is often to introduce randomness into the dataset, especially when the data might have some inherent order or structure. This can be beneficial during training of machine learning models, as it helps prevent the model from learning patterns based on the order of the data.

**X, y = lines['english\_sentence'], lines['hindi\_sentence']**

**X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.2,random\_state=42)**

**X\_train.shape, X\_test.shape** X, y = lines['english\_sentence'], lines['hindi\_sentence']

This assigns the 'english\_sentence' column from the DataFrame lines to the variable X and the 'hindi\_sentence' column to the variable y.

python

 X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

The train\_test\_split function is used to split the data into training and testing sets. The parameters are as follows:

* X and y: The data to be split (English and Hindi sentences).
* test\_size=0.2: Specifies that 20% of the data will be used for testing, and the remaining 80% will be used for training.
* random\_state=42: Provides a seed for the random number generator, ensuring reproducibility of the split. The same seed will result in the same split every time the code is run.

The resulting variables are as follows:

* X\_train: Training set of English sentences.
* X\_test: Testing set of English sentences.
* y\_train: Training set of Hindi sentences.
* y\_test: Testing set of Hindi sentences.

The shapes of X\_train and X\_test are printed, providing information about the number of samples (sentences) in each set. This train-test split is a common practice in machine learning to assess the model's performance on unseen data.

**X\_train.to\_pickle('X\_train.pkl')**

**X\_test.to\_pickle('X\_test.pkl')**X\_train.to\_pickle(...): This method is used to serialize the X\_train DataFrame to a binary format.

'X\_train.pkl': Specifies the filename to which the pickled data will be saved. In this case, it's 'X\_train.pkl'.

**X\_test.to\_pickle('X\_test.pkl'):**

X\_test.to\_pickle(...): Similarly, this method is used to serialize the X\_test DataFrame to a binary format.

'X\_test.pkl': Specifies the filename to which the pickled data will be saved. In this case, it's 'X\_test.pkl'.

**def generate\_batch(X = X\_train, y = y\_train, batch\_size = 128):**

**''' Generate a batch of data '''**

1. **Function Definition:**
   * def generate\_batch(X=X\_train, y=y\_train, batch\_size=128):: This line defines a function named generate\_batch with three parameters:
     + X: Default value is set to X\_train, which is assumed to be the training set of English sentences.
     + y: Default value is set to y\_train, which is assumed to be the training set of Hindi sentences.
     + batch\_size: Default value is set to 128, representing the number of samples in each batch.
2. **Function Body:**
   * The body of the function is not provided in the code snippet. It is expected that the function body will include logic to generate batches of data based on the specified parameters (e.g., X, y, batch\_size).
3. **Docstring:**
   * ''' Generate a batch of data ''': This is a docstring, which provides a brief description or documentation for the function. In this case, it indicates that the function is intended to generate a batch of data.

**while True:**

**for j in range(0, len(X), batch\_size):**

**encoder\_input\_data = np.zeros((batch\_size, max\_length\_src),dtype='float32')**

**decoder\_input\_data = np.zeros((batch\_size, max\_length\_tar),dtype='float32')**

**decoder\_target\_data = np.zeros((batch\_size, max\_length\_tar, num\_decoder\_tokens),dtype='float32')**

**for i, (input\_text, target\_text) in enumerate(zip(X[j:j+batch\_size], y[j:j+batch\_size])):**

**for t, word in enumerate(input\_text.split()):**

**encoder\_input\_data[i, t] = input\_token\_index[word]**

**for t, word in enumerate(target\_text.split()):**

**if t<len(target\_text.split())-1:**

**decoder\_input\_data[i, t] = target\_token\_index[word]**

**if t>0:**

1. **Outer Loop (while True):**
   * The while True loop indicates an infinite loop, which is common when generating data for training neural networks. It continues indefinitely until manually interrupted.
2. **Inner Loop (for j in range(0, len(X), batch\_size)):**
   * This loop iterates through the indices of the training data (X) in batches of size batch\_size.
3. **Data Initialization:**
   * encoder\_input\_data, decoder\_input\_data, and decoder\_target\_data are initialized as NumPy arrays filled with zeros. These arrays will be populated with actual data in the following steps.
4. **Data Population Loop (for i, (input\_text, target\_text) in enumerate(zip(X[j:j+batch\_size], y[j:j+batch\_size]))):**
   * This loop iterates over pairs of English (input\_text) and Hindi (target\_text) sentences within the current batch.
5. **Encoding Input Text (for t, word in enumerate(input\_text.split()):):**
   * It iterates through the words in the input sentence (input\_text.split()) and maps each word to its corresponding index in the input\_token\_index. The resulting indices are stored in encoder\_input\_data.
6. **Decoding Input Text (for t, word in enumerate(target\_text.split()):):**
   * Similar to the encoding loop, it iterates through the words in the target sentence (target\_text.split()) and maps each word to its corresponding index in the target\_token\_index. The resulting indices are stored in decoder\_input\_data.
7. **Decoding Target Text (if t > 0: decoder\_target\_data[i, t - 1, target\_token\_index[word]] = 1.0):**
   * It creates a one-hot encoded representation of the target sentence for training the decoder. The one-hot encoding is shifted by one position (hence t - 1) to represent the target at the next time step.

**decoder\_target\_data[i, t - 1, target\_token\_index[word]] = 1.**

**yield([encoder\_input\_data, decoder\_input\_data], decoder\_target\_data)**

1. **Setting Decoder Target Data:**
   * decoder\_target\_data[i, t - 1, target\_token\_index[word]] = 1.: This line sets the value at a specific position in the decoder\_target\_data array to 1.0. This array is used for training the decoder part of the neural network.
2. **Yielding Data:**
   * yield([encoder\_input\_data, decoder\_input\_data], decoder\_target\_data): The yield statement is used to produce a batch of data and make it available to the training process. The yielded data consists of two main components:
     + [encoder\_input\_data, decoder\_input\_data]: The input data for both the encoder and the decoder components of the neural network. This typically includes the English sentence as input to the encoder and the Hindi sentence (with a start token) as input to the decoder.
     + decoder\_target\_data: The target data for the decoder. This includes the one-hot encoded representation of the target sentence, shifted by one position.
   * The yield statement implies that this function is being used as a generator function. Generator functions are used in Python to create iterators, and they can be used to efficiently produce batches of data on the fly during training, especially when dealing with large datasets.

**Encoder-Decoder Architecture**

* **latent\_dim=300  
    
  Variable Name:** latent\_dim
* **Purpose:** Represents the dimensionality of the latent space.
* **Value:** The specific value assigned to latent\_dim (300 in this case) determines the number of dimensions in the latent space. A higher latent\_dim allows the model to capture more complex relationships in the data but may also require more computational resources.
* **Usage Examples:**
  + In an autoencoder, latent\_dim is often the size of the bottleneck layer, which is the layer in the middle of the network where the input is compressed into a lower-dimensional representation.
  + In a variational autoencoder (VAE), latent\_dim represents the dimensionality of the continuous latent variables.

**# Encoder**

**encoder\_inputs = Input(shape=(None,))**

**enc\_emb = Embedding(num\_encoder\_tokens+1, latent\_dim, mask\_zero = True)(encoder\_inputs)**

**encoder\_lstm = LSTM(latent\_dim, return\_state=True)**

**encoder\_outputs, state\_h, state\_c = encoder\_lstm(enc\_emb)**

**encoder\_states = [state\_h, state\_c]**

1. **Input Layer:**
   * encoder\_inputs = Input(shape=(None,)): This creates an input layer for the encoder. The shape=(None,) indicates that the input sequences can have variable length. The None allows the model to handle sequences of different lengths.
2. **Embedding Layer:**
   * enc\_emb = Embedding(num\_encoder\_tokens + 1, latent\_dim, mask\_zero=True)(encoder\_inputs): This creates an embedding layer for the input sequences. It converts each integer index in the input sequence into a dense vector of latent\_dim dimensions. The mask\_zero=True parameter indicates that the model should consider the padding value (0) as a masked value, and it will not contribute to the loss or updates during training.
3. **LSTM Layer:**
   * encoder\_lstm = LSTM(latent\_dim, return\_state=True): This creates an LSTM layer with latent\_dim units. The return\_state=True parameter means that the layer will return the hidden state and cell state of the LSTM in addition to the usual output.
4. **Applying Layers:**
   * encoder\_outputs, state\_h, state\_c = encoder\_lstm(enc\_emb): This applies the embedding layer to the input sequences, feeding the output into the LSTM layer. The encoder\_outputs represent the output sequence from the LSTM, while state\_h and state\_c represent the hidden state and cell state of the LSTM at the last time step.
5. **Hidden States:**
   * encoder\_states = [state\_h, state\_c]: This creates a list encoder\_states containing the hidden state (state\_h) and cell state (state\_c) of the LSTM. These states will be used as the initial states for the decoder part of the sequence-to-sequence model.

**decoder\_inputs = Input(shape=(None,))**

**dec\_emb\_layer = Embedding(num\_decoder\_tokens, latent\_dim, mask\_zero = True)**

**dec\_emb = dec\_emb\_layer(decoder\_inputs)**

1. **Input Layer:**
   * decoder\_inputs = Input(shape=(None,)): This creates an input layer for the decoder. Similar to the encoder input layer, it allows for variable-length sequences.
2. **Embedding Layer for Decoder:**
   * dec\_emb\_layer = Embedding(num\_decoder\_tokens, latent\_dim, mask\_zero=True): This creates an embedding layer for the target sequences (decoder inputs). It maps each integer index in the target sequence to a dense vector of latent\_dim dimensions. The mask\_zero=True parameter indicates that the model should consider the padding value (0) as a masked value, which won't contribute to the loss or updates during training.
3. **Applying Embedding Layer:**
   * dec\_emb = dec\_emb\_layer(decoder\_inputs): This applies the embedding layer to the decoder input sequences. It converts each integer index in the target sequence into a dense vector of latent\_dim dimensions.

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

**decoder\_outputs, \_, \_ = decoder\_lstm(dec\_emb,**

1. **LSTM Layer for Decoder:**
   * decoder\_lstm = LSTM(latent\_dim, return\_sequences=True, return\_state=True): This creates an LSTM layer for the decoder. The latent\_dim parameter sets the number of units (dimensions) in the LSTM layer. The return\_sequences=True parameter indicates that the LSTM should return the full sequence of outputs for each time step, and return\_state=True indicates that it should return the final hidden state and cell state.
2. **Applying LSTM Layer:**
   * decoder\_outputs, \_, \_ = decoder\_lstm(dec\_emb, initial\_state=encoder\_states): This applies the LSTM layer to the embedded decoder input sequences (dec\_emb). The initial\_state parameter is set to encoder\_states, which represents the hidden state and cell state of the encoder LSTM. This initialization allows the decoder to start generating sequences based on the information encoded by the encoder.
   * decoder\_outputs: The output sequences from the decoder LSTM for each time step. With return\_sequences=True, this will be a 3D tensor representing the entire sequence of outputs.
   * The underscores (\_) are used to discard the intermediate states returned by the LSTM layer. In this case, they are not used in subsequent calculations.

**initial\_state=encoder\_states)**

**decoder\_dense = Dense(num\_decoder\_tokens, activation='softmax')**

**decoder\_outputs = decoder\_dense(decoder\_outputs)**The initial\_state=encoder\_states parameter in the context of a Keras LSTM layer is used to set the initial state of the LSTM cell(s) during sequence generation. In a sequence-to-sequence model, this parameter is crucial for connecting the encoder and decoder components.

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

* **Model Class:** The Model class in Keras is used to instantiate a model. It allows you to define a model using a functional API, specifying the input(s) and output(s) of the model.
* **Inputs and Outputs:**
  + **Inputs:** [encoder\_inputs, decoder\_inputs] - The model has two input layers: one for the encoder input sequences (encoder\_inputs) and one for the decoder input sequences (decoder\_inputs).
  + **Output:** decoder\_outputs - The output layer is the output sequences generated by the decoder LSTM (decoder\_outputs).
* **Model Instance:**
  + model = Model([encoder\_inputs, decoder\_inputs], decoder\_outputs) - This line creates an instance of the Model class by specifying the inputs and outputs. The inputs and outputs are organized in a list passed to the Model constructor.

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

t allows you to configure various aspects of the learning process, specifying the optimizer, loss function, and metrics to be used during training. Here's a detailed explanation of the parameters and the role of model.compile:

python

model.compile(optimizer='rmsprop', loss='categorical\_crossentropy', metrics=['accuracy'])

* **optimizer parameter:**
  + **Purpose:** Specifies the optimization algorithm used to update the model's weights during training.
  + **Values:** Strings or optimizer instances. Examples include 'rmsprop', 'adam', 'sgd' (Stochastic Gradient Descent), or instances of optimizer classes.
  + **Default:** If not specified, the default optimizer is usually 'adam'.
* **loss parameter:**
  + **Purpose:** Specifies the loss function that the model will try to minimize during training. It measures the difference between the predicted output and the true target values.
  + **Values:** Strings or loss instances. Common choices include 'categorical\_crossentropy' for multiclass classification, 'binary\_crossentropy' for binary classification, or other custom loss functions.
  + **Default:** No default; you must specify the loss function.
* **metrics parameter:**
  + **Purpose:** Specifies additional metrics to monitor during training. These metrics provide insight into the model's performance but do not directly affect the training process.
  + **Values:** List of strings or metric instances. Examples include 'accuracy', 'precision', 'recall', or custom metrics.
  + **Default:** Typically, no default is set, and you need to explicitly specify metrics if desired.
* **Other Optional Parameters:**
  + There are other optional parameters like loss\_weights, sample\_weight\_mode, weighted\_metrics, etc., which are less commonly used and may be omitted for standard use cases.

**Model: "model\_1"**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Layer (type) Output Shape Param # Connected to**

**==================================================================================================**

**input\_3 (InputLayer) [(None, None)] 0 []**

**input\_4 (InputLayer) [(None, None)] 0 []**

**embedding\_2 (Embedding) (None, None, 300) 4209300 ['input\_3[0][0]']**

**embedding\_3 (Embedding) (None, None, 300) 5262300 ['input\_4[0][0]']**

**lstm\_2 (LSTM) [(None, 300), 721200 ['embedding\_2[0][0]']**

**(None, 300),**

**(None, 300)]**

**lstm\_3 (LSTM) [(None, None, 300), 721200 ['embedding\_3[0][0]',**

**(None, 300), 'lstm\_2[0][1]',**

**(None, 300)] 'lstm\_2[0][2]']**

**dense\_1 (Dense) (None, None, 17541) 5279841 ['lstm\_3[0][0]']**

**==================================================================================================**

**Total params: 16193841 (61.77 MB)**

**Trainable params: 16193841 (61.77 MB)**

**Non-trainable params: 0 (0.00 Byte)**

* **Model Architecture:**
  + The model consists of three main parts: an embedding layer for the encoder input, an embedding layer for the decoder input, and a sequence-to-sequence layer involving LSTMs.
* **Input Layers (input\_3 and input\_4):**
  + Each input layer (input\_3 and input\_4) represents a placeholder for sequences with variable lengths.
* **Embedding Layers (embedding\_2 and embedding\_3):**
  + embedding\_2: Embedding layer for the encoder input (input\_3). It converts integer indices into dense vectors of size 300.
  + embedding\_3: Embedding layer for the decoder input (input\_4). Similar to embedding\_2.
* **LSTM Layers (lstm\_2 and lstm\_3):**
  + lstm\_2: LSTM layer for the encoder. It processes the embedded input sequence (embedding\_2) and produces a hidden state (None, 300) and cell state (None, 300). The number of parameters is calculated using the LSTM formula:
    - Parameters per LSTM unit: 4 \times (\text{{input\_dim}} + \text{{units}} + 1)
    - In this case, 4×(300+300+1)=2,4004×(300+300+1)=2,400 parameters per unit.
    - Since there are 300 units, the total number of parameters is 2,400×300=720,0002,400×300=720,000.
  + lstm\_3: LSTM layer for the decoder. It processes the embedded decoder input (embedding\_3), the hidden state from lstm\_2, and the cell state from lstm\_2, producing output sequences (None, None, 300). The number of parameters is calculated similarly to lstm\_2.
* **Dense Layer (dense\_1):**
  + Dense layer with 17,541 units. It is responsible for mapping the decoder LSTM's output to the vocabulary size (17,541). The number of parameters in a dense layer is calculated as \text{{units}} \times (\text{{input\_dim}} + 1).
  + In this case, 17,541×(300+1)=5,279,84117,541×(300+1)=5,279,841 parameters.
* **Total Parameters:**
  + The total number of trainable parameters in the model is the sum of the parameters in all layers.
  + 4,209,300+5,262,300+720,000+720,000+5,279,841=16,193,8414,209,300+5,262,300+720,000+720,000+5,279,841=16,193,841 parameters.
* **Trainable Parameters vs. Non-trainable Parameters:**
  + All parameters are trainable in this model, as indicated by the fact that trainable parameters and total parameters are the same.
  + Non-trainable parameters usually refer to parameters in layers like the embedding layer where weights are fixed during training.

**train\_samples = len(X\_train)**

**val\_samples = len(X\_test)**

**batch\_size = 128**

**epochs = 50**

**import numpy as np**

**import tensorflow as tf**

**seed\_value = 42**

**np.random.seed(seed\_value)**

**tf.random.set\_seed(seed\_value)**

**model.fit\_generator(generator = generate\_batch(X\_train, y\_train, batch\_size = batch\_size),**

**steps\_per\_epoch = train\_samples//batch\_size,**

**epochs=epochs,**

**validation\_data = generate\_batch(X\_test, y\_test, batch\_size = batch\_size),**

* **validation\_steps = val\_samples//batch\_size)  
    
  train\_samples and val\_samples:**
  + These variables store the number of samples in the training and validation sets, respectively.
* **batch\_size:**
  + Specifies the number of samples in each mini-batch during training.
* **epochs:**
  + Defines the number of times the entire training dataset will be passed forward and backward through the neural network.
* **Setting Random Seeds:**
  + This code sets the random seed values for NumPy (np.random.seed(seed\_value)) and TensorFlow (tf.random.set\_seed(seed\_value)). This ensures reproducibility of the training process. When you set random seeds, you get the same random numbers on each run, which is important for reproducibility.
* **fit\_generator method:**
  + fit\_generator is used for training a model using data generated batch-by-batch by a Python generator function.
  + It takes several parameters:
    - generator: The Python generator function that yields batches of training data.
    - steps\_per\_epoch: The number of batches to process before moving on to the next epoch. It's usually set to the total number of samples divided by the batch size.
    - epochs: The number of times to iterate over the entire training dataset.
    - validation\_data: The Python generator function for validation data.
    - validation\_steps: The number of batches of validation data to process before stopping the validation at the end of each epoch.
* **generate\_batch function:**
  + This function is expected to be a generator function that yields batches of training or validation data. It's likely designed to work with the sequence-to-sequence model and the data preparation steps you have implemented earlier.
* **Training Process:**
  + The training process involves iterating over the specified number of epochs (50 in this case).
  + For each epoch, the training data is divided into batches, and the model's weights are updated based on the computed gradients. The process is repeated for the specified number of steps per epoch (train\_samples // batch\_size).
  + Validation is performed at the end of each epoch using the validation data and the specified number of validation steps (val\_samples // batch\_size).
* **Note:**
  + The code assumes that you have defined and compiled your model (model) using the architecture described earlier and that the generate\_batch function is implemented to generate batches of training and validation data.

This code essentially kicks off the training process for your model, allowing it to learn from the training data and validating its performance on the validation data.

**Epoch 1/50**

* **model.fit\_generator(generator = generate\_batch(X\_train, y\_train, batch\_size = batch\_size),  
  generator parameter:**
  + Specifies the Python generator function that yields batches of training data. In this case, it's set to generate\_batch(X\_train, y\_train, batch\_size=batch\_size).
  + The generator function is responsible for providing batches of input data (X\_train) and corresponding target data (y\_train) to the model during training.
* **steps\_per\_epoch parameter:**
  + Indicates the number of batches to process before moving on to the next epoch. It's typically set to the total number of training samples divided by the batch size (train\_samples // batch\_size).
  + For each epoch, the model will go through this number of batches before completing the epoch.
* **epochs parameter:**
  + Specifies the number of times to iterate over the entire training dataset. In this case, it's set to 50.
* **validation\_data parameter:**
  + Specifies the Python generator function for validation data. It's set to generate\_batch(X\_test, y\_test, batch\_size=batch\_size).
  + The validation data is used to evaluate the model's performance on a separate dataset during training.
* **validation\_steps parameter:**
  + Indicates the number of batches of validation data to process before stopping the validation at the end of each epoch. It's typically set to the total number of validation samples divided by the batch size (val\_samples // batch\_size).
  + For each validation epoch, the model will go through this number of batches before completing the validation.

In summary, model.fit\_generator initiates the training process for the neural network model. It iterates over the specified number of epochs, processing batches of training data from the provided generator function (generate\_batch). The model's weights are updated based on the computed gradients, and the process is repeated for the specified number of steps per epoch. Validation is performed at the end of each epoch using the validation data provided by the generator function for validation data.

**<keras.src.callbacks.History at 0x196f6bc7cd0>**You can use this information to visualize the training progress, monitor overfitting, and make decisions about the model's performance. For example, you might plot training and validation loss curves over epochs or observe how accuracy changes during training.

**model.save\_weights('nmt\_weights.h5')**

* **model:** This refers to the Keras model that you have defined and trained.
* **save\_weights:** This is a method of the Keras model object.
* **'nmt\_weights.h5':** This is the filename where the model weights will be saved. The extension .h5 indicates that the Hierarchical Data Format (HDF5) will be used to store the weights. HDF5 is a file format commonly used for storing large amounts of numerical data, and it is suitable for storing the weights of a neural network.

**encoder\_model = Model(encoder\_inputs, encoder\_states)**

* **encoder\_model:** This is a new instance of the Keras Model class, representing the encoder part of the original sequence-to-sequence model.
* **encoder\_inputs:** This is likely the input layer or tensor that corresponds to the encoder input sequences in the original model.
* **encoder\_states:** This represents the hidden and cell states of the encoder LSTM. In the original model, encoder\_states would be the output of the encoder LSTM layer.
* **Model(encoder\_inputs, encoder\_states):** This line creates a new model that takes the same input as the original model (the encoder input sequences) but outputs only the encoder states (hidden and cell states) instead of the full sequence of encoder outputs.

**decoder\_state\_input\_h = Input(shape=(latent\_dim,))**

**decoder\_state\_input\_c = Input(shape=(latent\_dim,))**

**decoder\_states\_inputs = [decoder\_state\_input\_h, decoder\_state\_input\_c]**

 **decoder\_state\_input\_h:**

* This is an input layer representing the initial hidden state (h) for the decoder LSTM.
* The shape is set to (latent\_dim,), where latent\_dim is the dimensionality of the hidden state.

 **decoder\_state\_input\_c:**

* This is another input layer representing the initial cell state (c) for the decoder LSTM.
* Similar to decoder\_state\_input\_h, the shape is set to (latent\_dim,).

 **decoder\_states\_inputs:**

* This is a list containing the two decoder state input layers, decoder\_state\_input\_h and decoder\_state\_input\_c.
* It is used to create an initial state for the decoder LSTM during the decoding phase

**dec\_emb2= dec\_emb\_layer(decoder\_inputs)   
dec\_emb\_layer:**

* + This is the embedding layer for the decoder. It takes integer indices representing words in the target language and converts them into dense vectors.
* **decoder\_inputs:**
  + This likely represents the input sequences for the decoder. These sequences are typically integer indices representing words in the target language.
* **dec\_emb2:**
  + This variable holds the result of applying the decoder's embedding layer to the decoder inputs.
  + The shape of dec\_emb2 would be (batch\_size, sequence\_length, embedding\_dim), where batch\_size is the number of sequences in a batch, sequence\_length is the length of each sequence, and embedding\_dim is the dimensionality of the embedding vectors.

**decoder\_outputs2, state\_h2, state\_c2 = decoder\_lstm(dec\_emb2, initial\_state=decoder\_states\_inputs)**

**decoder\_states2 = [state\_h2, state\_c2]**

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

* **decoder\_lstm:**
  + This is the LSTM layer for the decoder.
* **dec\_emb2:**
  + This represents the embedded decoder inputs.
* **initial\_state=decoder\_states\_inputs:**
  + The initial state for the decoder LSTM is set using the previously defined initial hidden and cell states (decoder\_states\_inputs).
* **decoder\_outputs2, state\_h2, state\_c2:**
  + decoder\_outputs2 represents the output sequences from the decoder LSTM.
  + state\_h2 represents the hidden state of the decoder LSTM after processing the input sequences.
  + state\_c2 represents the cell state of the decoder LSTM after processing the input sequences.

python

decoder\_states2 = [state\_h2, state\_c2]

* **decoder\_states2:**
  + This variable is a list containing the final hidden state (state\_h2) and the final cell state (state\_c2) of the decoder LSTM.

python

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

* **decoder\_dense:**
  + This likely represents a dense (fully connected) layer that maps the output sequences from the decoder LSTM to the target vocabulary.
* **decoder\_outputs2:**
  + The output sequences from the decoder LSTM (decoder\_outputs2) are passed through the dense layer to generate the final output. This can be interpreted as the probabilities or scores for each word in the target vocabulary.

**decoder\_model = Model(**

**[decoder\_inputs] + decoder\_states\_inputs,**

**[decoder\_outputs2] + decoder\_states2)**

* **decoder\_model:**
  + This is a new instance of the Keras Model class representing the decoder part of the sequence-to-sequence model.
* **[decoder\_inputs] + decoder\_states\_inputs:**
  + This specifies the input to the decoder model. It's a list containing the decoder inputs (decoder\_inputs) and the initial hidden and cell states (decoder\_states\_inputs).
* **[decoder\_outputs2] + decoder\_states2:**
  + This specifies the output of the decoder model. It's a list containing the decoder outputs (decoder\_outputs2) and the final hidden and cell states (decoder\_states2).

**def decode\_sequence(input\_seq):**

**states\_value = encoder\_model.predict(input\_seq)**

**target\_seq = np.zeros((1,1))**

**target\_seq[0, 0] = target\_token\_index['START\_']**

* **decode\_sequence function:**
  + This function is intended to take an input sequence (input\_seq) and generate an output sequence using a pre-trained sequence-to-sequence model.
* **encoder\_model.predict(input\_seq):**
  + This line uses the pre-trained encoder\_model to predict the final hidden and cell states (states\_value) based on the input sequence (input\_seq). The input\_seq is likely a sequence of words in the source language.
* **target\_seq = np.zeros((1, 1)):**
  + This line initializes the target sequence (target\_seq) with zeros. The shape of the array is (1, 1), indicating a sequence length of 1.
* **target\_seq[0, 0] = target\_token\_index['START\_']:**
  + This sets the first element of the target sequence to the index of the special token 'START\_' in the target language. This token is often used to indicate the beginning of a sequence during decoding.

**stop\_condition = False**

**decoded\_sentence = ''**

**while not stop\_condition:**

**output\_tokens, h, c = decoder\_model.predict([target\_seq] + states\_value)**

* **stop\_condition:**
  + This boolean variable is used to control the loop. The loop continues until stop\_condition becomes True.
* **decoded\_sentence:**
  + This variable is used to store the partially decoded sentence during the decoding process. It starts as an empty string.
* **decoder\_model.predict([target\_seq] + states\_value):**
  + This line uses the pre-trained decoder\_model to predict the next token in the sequence. The input to the model consists of the current target sequence (target\_seq) and the current states (states\_value), which include the hidden and cell states from the previous time step.
* **output\_tokens, h, c = decoder\_model.predict(...)**
  + output\_tokens: The predicted probability distribution over the target vocabulary for the next token.
  + h: The updated hidden state of the decoder LSTM.
  + c: The updated cell state of the decoder LSTM.

**sampled\_token\_index = np.argmax(output\_tokens[0, -1, :])**

**sampled\_char = reverse\_target\_char\_index[sampled\_token\_index]**

**decoded\_sentence += ' '+sampled\_char**

* **sampled\_token\_index:**
  + This line uses np.argmax to find the index of the token with the highest probability in the output probability distribution. This index corresponds to the predicted next token in the sequence.
* **sampled\_char:**
  + This variable represents the actual token (character or word) in the target language corresponding to the sampled\_token\_index. It is obtained from the reverse\_target\_char\_index, which is a dictionary mapping token indices to characters or words in the target language.
* **decoded\_sentence += ' ' + sampled\_char:**
  + This line appends the sampled character to the decoded\_sentence string, with a space character (' ') added before the sampled character. This is a common practice when decoding sequences to separate words or tokens.

**if (sampled\_char == '\_END' or**

**len(decoded\_sentence) > 50):**

**stop\_condition = True**

* **sampled\_char == '\_END':**
  + This condition checks whether the recently sampled character is equal to the special token '\_END'. If it is, the decoding process is terminated. '\_END' typically denotes the end of a sequence in some sequence-to-sequence models.
* **len(decoded\_sentence) > 50:**
  + This condition checks whether the length of the decoded sentence has exceeded a certain threshold (in this case, 50 characters). If the length condition is met, the decoding process is also terminated.
* **stop\_condition = True:**
  + If either of the conditions is satisfied, the stop\_condition is set to True, indicating that the decoding loop should stop.

**target\_seq = np.zeros((1,1))**

**target\_seq[0, 0] = sampled\_token\_index**

* **target\_seq = np.zeros((1, 1)):**
  + This line initializes a new target sequence (target\_seq) with zeros. The shape of the array is (1, 1), indicating a sequence length of 1.
* **target\_seq[0, 0] = sampled\_token\_index:**
  + This sets the first (and only) element of the target sequence to the sampled token index obtained during the current iteration of the decoding loop. This updated target\_seq is then used as input to the decoder model during the next iteration.

**states\_value = [h, c]**

* **h and c:**
  + These variables likely represent the hidden state (h) and cell state (c) of the decoder LSTM at the current time step. They were obtained as part of the decoder model's prediction during the current iteration of the decoding loop.
* **states\_value = [h, c]:**
  + This line creates a list states\_value containing the updated hidden and cell states. These updated states will be used as input to the decoder model during the next iteration of the loop.

**return decoded\_sentence**

The return decoded\_sentence statement indicates that the function will return the final decoded sentence after the decoding loop is complete. The decoding loop iteratively generates tokens and updates the decoded\_sentence string until a stopping condition is met.

**train\_gen = generate\_batch(X\_train, y\_train, batch\_size = 1)**

**k=-1**

* **generate\_batch(X\_train, y\_train, batch\_size=1):**
  + This function, named generate\_batch, is likely designed to generate batches of training data for your sequence-to-sequence model. It's common to use generators in Keras when training models on large datasets to avoid loading the entire dataset into memory.
* **train\_gen:**
  + This variable holds the generator object created by calling generate\_batch. You can use this generator in a training loop to obtain batches of training data.
* **batch\_size=1:**
  + The batch\_size parameter is set to 1, indicating that each call to the generator (train\_gen) will yield a batch of data with a single sample. This is useful in scenarios where you want to update the model's weights after processing each individual sample.
* **k = -1:**
  + This line initializes a variable k with a value of -1. The purpose of this variable and its usage in the following code is not clear from the provided snippet. It might be used as an index or counter within a training loop.

**k+=1**

**(input\_seq, actual\_output), \_ = next(train\_gen)**

**decoded\_sentence = decode\_sequence(input\_seq)**

**print('Input English sentence:', X\_train[k:k+1].values[0])**

**print('Actual Hindi Translation:', y\_train[k:k+1].values[0][6:-4])**

**print('Predicted Hindi Translation:', decoded\_sentence[:-4])**

**1/1 [==============================] - 0s 25ms/step**

**1/1 [==============================] - 0s 25ms/step**

**1/1 [==============================] - 0s 23ms/step**

**1/1 [==============================] - 0s 25ms/step**

**1/1 [==============================] - 0s 23ms/step**

**1/1 [==============================] - 0s 27ms/step**

**Input English sentence: we just spent and entire week**

**Actual Hindi Translation: हमने एक पूरे सप्ताह**

**Predicted Hindi Translation: हमने एक पूरे सप्ताह**

* **k += 1:**
  + This increments the variable k by 1. It seems like k is being used as a counter to keep track of the batches processed during training.
* **(input\_seq, actual\_output), \_ = next(train\_gen):**
  + This line fetches the next batch from the generator (train\_gen). It assumes that each batch is a tuple where the first element (input\_seq) represents the input sequence in English, and the second element (actual\_output) represents the actual output sequence in Hindi. The trailing underscore \_ is used to discard any additional values returned by the generator.
* **decoded\_sentence = decode\_sequence(input\_seq):**
  + This line uses the decode\_sequence function to generate a predicted Hindi translation (decoded\_sentence) based on the input English sequence (input\_seq).
* **Printing Statements:**
  + The following three print statements display the input English sentence, the actual Hindi translation, and the predicted Hindi translation for the current batch.

**from keras.models import load\_model**

**model.save('nmt\_model.h5')**

**loaded\_model = load\_model('nmt\_model.h5')**

1. **model.save('nmt\_model.h5'):**
   * This line saves the Keras model (model) to a file named 'nmt\_model.h5' using the HDF5 format. This format is commonly used for saving Keras models.
2. **loaded\_model = load\_model('nmt\_model.h5'):**
   * This line loads the saved model from the file 'nmt\_model.h5' into a new variable named loaded\_model. The load\_model function is used to load the model.