

Aim: Implement Language translator using Encoder Decoder model.

Theory:

Introduction

The encoder-decoder model is a way of using recurrent neural networks for sequence-to-sequence prediction problems.

It was initially developed for machine translation problems, although it has proven successful at related sequence-to-sequence prediction problems such as text summarization and question answering.

The approach involves two recurrent neural networks, one to encode the input sequence, called the encoder, and a second to decode the encoded input sequence into the target sequence called the decoder.

Following are some of the application of sequence to sequence models-

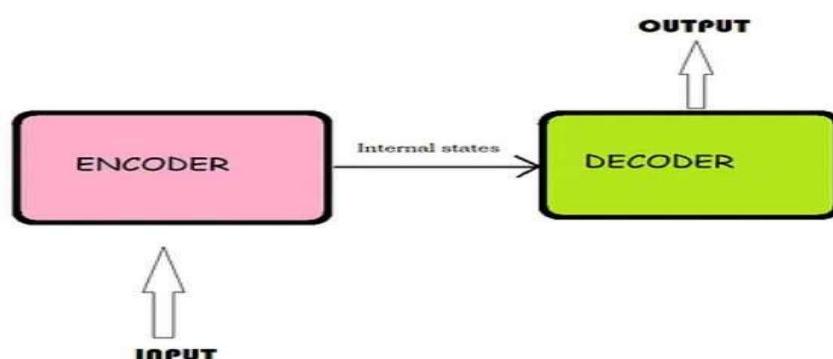
Chatbots

Machine Translation

Text summary

Image captioning

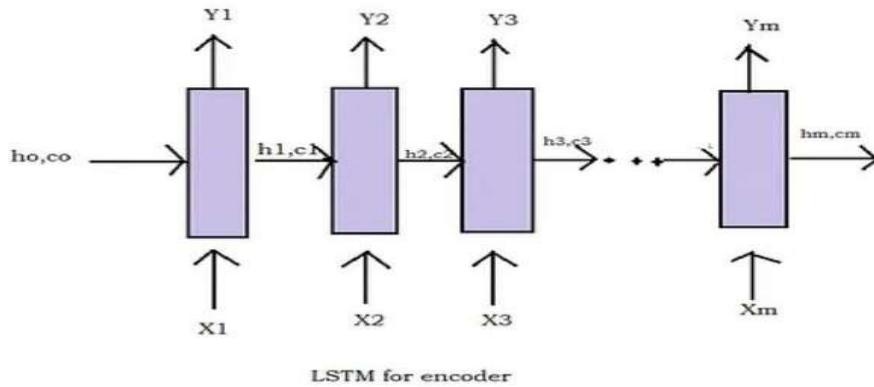
The architecture of Encoder-Decoder : The overall structure of sequence to sequence model(encoder-decoder) which is commonly used is as shown below



It consists of 3 parts: **encoder, intermediate vector and decoder.**

- **Encoder**-It accepts a single element of the input sequence at each time step, process it, collects information for that element and propagates it forward.

- **Intermediate vector-** This is the final internal state produced from the encoder part of the model. It contains information about the entire input sequence to help the decoder make accurate predictions.
- **Decoder-** given the entire sentence, it predicts an output at each time step.
- **Understanding the Encoder part of the model**
- The encoder is basically an LSTM/GRU cell.
- An encoder takes the input sequence and encapsulates the information as the internal state vectors.
- Outputs of the encoder are rejected and only internal states are used.
- Let's understand how encoder part of the model works



LSTM takes only one element at a time, so if the input sequence is of length m , then LSTM takes m time steps to read the entire sequence.

- X_t is the input at time step t .
- h_t and c_t are internal states at time step t of the LSTM and for GRU there is only one internal state h_t .
- Y_t is the output at time step t .

Let's consider an example of English to Hindi translation



Inputs of Encoder X_t -

Consider the English sentence- India is beautiful country. This sequence can be thought of as a sentence containing 4 words (India, is, beautiful, country). So here

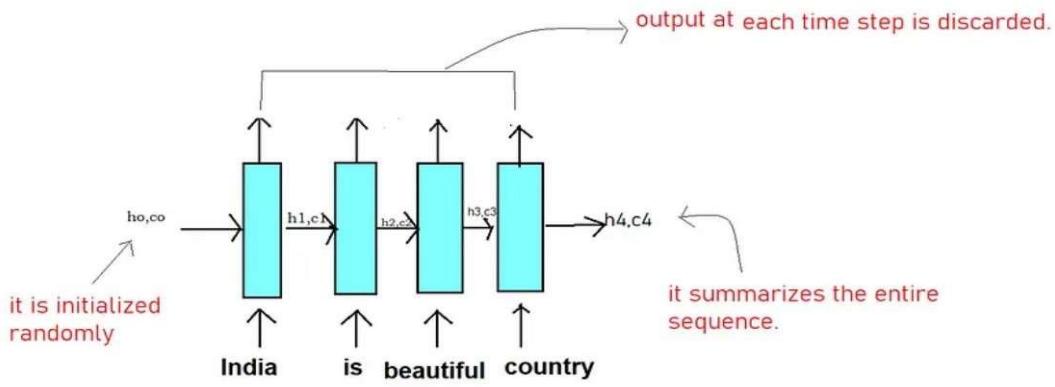
X_1 = 'India'

X_2 = 'is'

X_3 = 'beautiful'

X_4 = 'country'.

Therefore LSTM will read this sequence word by word in 4-time step as follows



Here each X_t (each word) is represented as a vector using the word embedding, which converts each word into a vector of fixed length.

Now coming to internal states (h_t, c_t) -

- It learns what the LSTM has read until time step t . For e.g when $t=2$, it remembers that LSTM has read '*India is*'.
- The initial states h_0, c_0 (both are vectors) is initialized randomly or with zeroes.
- Remember the dimension of h_0, c_0 is same as the number of units in LSTM cell.
- The final state h_4, c_4 contains the crux of the entire input sequence *India is beautiful country*.

The output of encoder Y_t -

Y_t at each time steps is the predictions of the LSTM at each time step. In machine translation problems, we generate the outputs when we have read the entire input sequence. So Y_t at each time step in the encoder is of no use so we discard it.

The encoder will read the English sentence word by word and store the final internal states (known as an intermediate vector) of the LSTM generated after the last time step and since the output will be generated once the entire sequence is read, therefore outputs (Y_t) of the Encoder at each time step are discarded.

Understanding the Decoder part of the model in the Training Phase-

The working of the decoder is different during the training and testing phase unlike the encoder part of the model which works in the same fashion in training and test phase.

Let's understand the working of the decoder part during training phase-

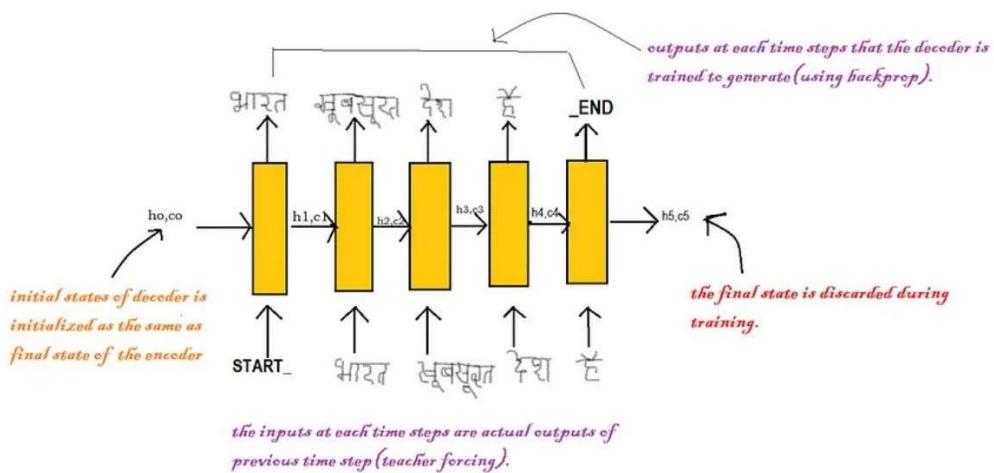
Taking the running example of translating ***India is beautiful country*** to its Hindi counterpart, just like encoder, the decoder also generates the output sentence word by word.

So we want to generate the output — **भारत खूबसूरत देश है**

For the decoder to recognize the starting and end of the sequence, we will add START_ at the beginning of the output sequence and _END at the end of the output sequence.

So our Output sentence will be **START_भारत खूबसूरत देश है_END**

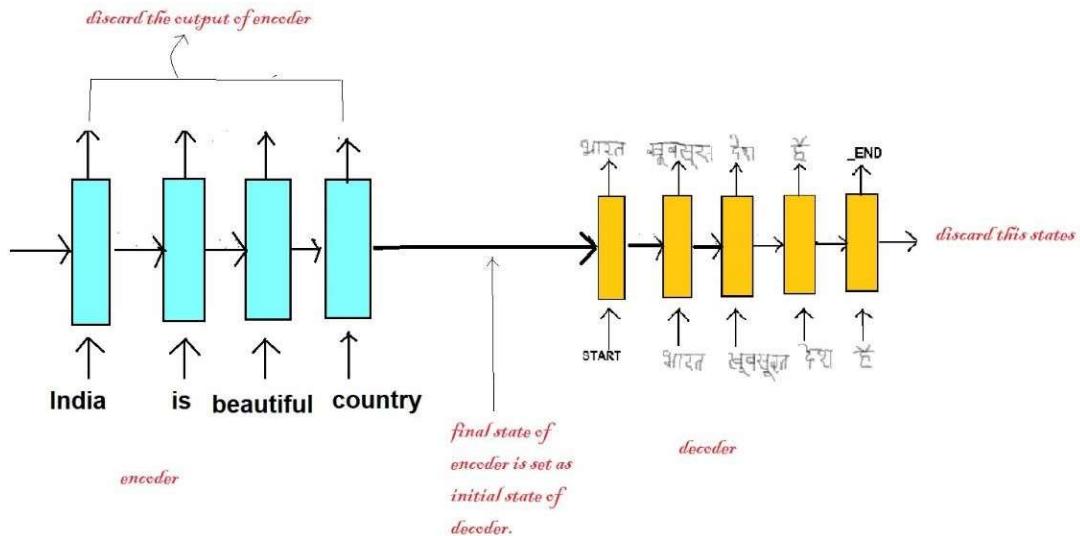
Let's understand the working visually



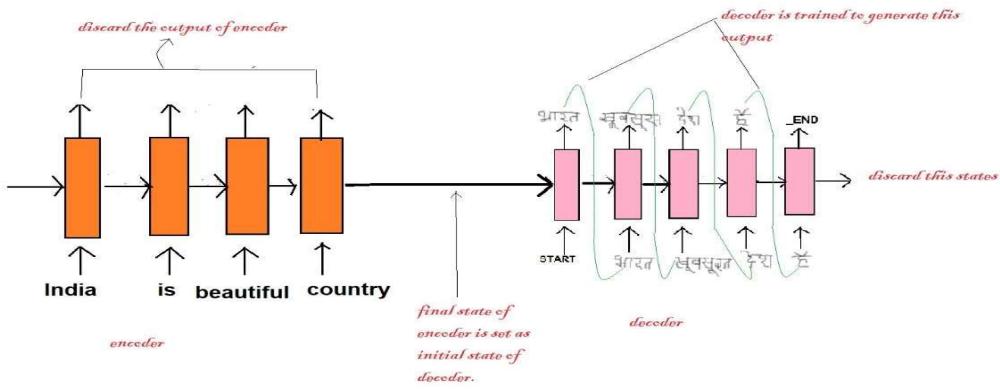
Decoder LSTM at training

- The initial states (h_0, c_0) of the decoder is set to the final states of the encoder. It can be thought of as that the decoder is trained to generate the output based on the information gathered by the encoder.
- Firstly, we input the **START** so that the decoder starts generating the next word. And after the last word in the Hindi sentence, we make the decoder learn to predict the **END**.
- Here we use the teacher forcing technique where the input at each time step is actual output and not the predicted output from the last time step.
- At last, the loss is calculated on the predicted outputs from each time step and the errors are backpropagated through time to update the parameters of the model.
- The final states of the decoder are discarded as we got the output hence it is of no use.

Summarizing the encoder-decoder visually



Understanding the Decoder part of the model in Test Phase



Process of the Decoder in the test period-

- The initial states of the decoder are set to the final states of the encoder.
- LSTM in the decoder process single word at every time step.
- Input to the decoder always starts with the START_.
- The internal states generated after every time step is fed as the initial states of the next time step. for e.g At t=1, the internal states produced after inputting START_ is fed as the initial states at t=2.
- The output produced at each time step is fed as input in the next time step.
- We get to know about the end of the sequence when the decoder predicts the END_.

Lab Experiment to be performed in the session :

Exercise-1: Create a base encoder-decoder machine translation model with LSTM

Step 1:

Use a dataset of pairs of English sentences and their Marathi translation, which you can download from manythings.org/anki. There are several other translations available, you can choose whichever you like. The file to download is called mar-eng.zip.

Step 2:

Implement a *character-level* sequence-to-sequence model, processing the input character-by-character and generating the output character-by-character.

OR

Implement a word-level model, which tends to be more common for machine translation.

```
In [1]: import numpy as np
import pandas as pd

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

/kaggle/input/marathi-english-translated-words/hin.txt
/kaggle/input/marathi-english-translated-words/mar.txt

In [2]: from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input,LSTM,Dense

batch_size = 64
epochs = 100
latent_dim = 256
num_samples = 10000

data_path = '/kaggle/input/marathi-english-translated-words/hin.txt'

/opt/conda/lib/python3.10/site-packages/scipy/_init__.py:1
6: UserWarning: A NumPy version >=1.16.5 and <1.23.0 is required for this version of SciPy (detected version 1.23.5
    warnings.warn(f"A NumPy version >={np_minversion} and <{np_
maxversion}")


In [3]: input_texts = []
target_texts = []
input_characters = set()
target_characters = set()

In [4]: with open(data_path, 'r', encoding='utf-8') as f:
    lines = f.read().split('\n')

    for line in lines[: min(num_samples, len(lines) - 1)]:
        input_text, target_text, _ = line.split('\t')
        # We use "tab" as the "start sequence" character
        # for the targets, and "\n" as "end sequence" character.
        target_text = '\t' + target_text + '\n'
        input_texts.append(input_text)
        target_texts.append(target_text)

        for char in input_text:
            if char not in input_characters:
                input_characters.add(char)

        for char in target_text:
            if char not in target_characters:
                target_characters.add(char)

In [5]: input_characters = sorted(list(input_characters))
target_characters = sorted(list(target_characters))

num_encoder_tokens = len(input_characters)
num_decoder_tokens = len(target_characters)

max_encoder_seq_length = max([len(txt) for txt in input_texts])
max_decoder_seq_length = max([len(txt) for txt in target_texts])
```

```
In [6]: print('Number of samples:', len(input_texts))
print('Number of unique input tokens:', num_encoder_tokens)
print('Number of unique output tokens:', num_decoder_tokens)
print('Max sequence length for inputs:', max_encoder_seq_length)
print('Max sequence length for outputs:', max_decoder_seq_length)
```

```
Number of samples: 2779
Number of unique input tokens: 70
Number of unique output tokens: 92
Max sequence length for inputs: 107
Max sequence length for outputs: 123
```

```
In [7]: input_token_index = dict([(char, i) for i, char in enumerate(input_characters)])
target_token_index = dict([(char, i) for i, char in enumerate(target_characters)])
```

```
In [8]: encoder_input_data = np.zeros((len(input_texts), max_encoder_seq_length, num_encoder_tokens), dtype='float32')
decoder_input_data = np.zeros((len(input_texts), max_decoder_seq_length, num_decoder_tokens), dtype='float32')
decoder_target_data = np.zeros((len(input_texts), max_decoder_seq_length, num_decoder_tokens), dtype='float32')
```

```
In [9]: for i, (input_text, target_text) in enumerate(zip(input_texts, target_texts)):
    for t, char in enumerate(input_text):
        encoder_input_data[i, t, input_token_index[char]] = 1.

    encoder_input_data[i, t + 1:, input_token_index[' ']] = 1.

    for t, char in enumerate(target_text):
        decoder_input_data[i, t, target_token_index[char]] = 1.
        if t > 0:
            decoder_target_data[i, t - 1, target_token_index[char]] = 1.

        decoder_input_data[i, t + 1:, target_token_index[' ']] = 1.
        decoder_target_data[i, t:, target_token_index[' ']] = 1.
```

Defining the encoder and decoder

```
In [10]: encoder_inputs = Input(shape=(None, num_encoder_tokens))
encoder = LSTM(latent_dim, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_inputs)
encoder_states = [state_h, state_c]

decoder_inputs = Input(shape=(None, num_decoder_tokens))
decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_inputs, initial_state=encoder_states)
decoder_dense = Dense(num_decoder_tokens, activation = 'softmax')
decoder_outputs = decoder_dense(decoder_outputs)
```

```
In [ ]: model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

model.compile(optimizer='adam', loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit([encoder_input_data, decoder_input_data], decoder_target_data,
          batch_size = batch_size, epochs = epochs, validation_split=0.2)
```

```
In [12]: model.save('model.h5')
```

```
In [13]: encoder_model = Model(encoder_inputs, encoder_states)

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_outputs, state_h, state_c = decoder_lstm(decoder_inputs, initial_state=decoder_states_inputs)
decoder_states = [state_h, state_c]
decoder_outputs = decoder_dense(decoder_outputs)
decoder_model = Model([decoder_inputs] + decoder_states_inputs, [decoder_outputs] + decoder_states)

reverse_input_char_index = dict((i, char) for char, i in input_token_index.items())
reverse_target_char_index = dict((i, char) for char, i in target_token_index.items())
```

```
In [14]: def decode_sequence(input_seq):
    states_value = encoder_model.predict(input_seq)

    target_seq = np.zeros((1, 1, num_decoder_tokens))
    target_seq[0, 0, target_token_index['\t']] = 1.

    stop_condition = False
    decoded_sentence = ''
    while not stop_condition:
        output_tokens, h, c = decoder_model.predict(
            [target_seq] + states_value)

        sampled_token_index = np.argmax(output_tokens[0, -1, :])
        sampled_char = reverse_target_char_index[sampled_token_index]
        decoded_sentence += sampled_char

        if (sampled_char == '\n' or
            len(decoded_sentence) > max_decoder_seq_length):
            stop_condition = True

    target_seq = np.zeros((1, 1, num_decoder_tokens))
    target_seq[0, 0, sampled_token_index] = 1.

    states_value = [h, c]

    return decoded_sentence
```

```
In [16]: for seq_index in range(4):
    input_seq = encoder_input_data[seq_index: seq_index + 1]
    decoded_sentence = decode_sequence(input_seq)
    print('*****')
    print('Input sentence:', input_texts[seq_index])
    print('Decoded sentence:', decoded_sentence)
    print("----")
```

1/1 [=====] - 0s 20ms/step
1/1 [=====] - 0s 23ms/step
1/1 [=====] - 0s 24ms/step
1/1 [=====] - 0s 24ms/step
1/1 [=====] - 0s 22ms/step
1/1 [=====] - 0s 20ms/step
1/1 [=====] - 0s 22ms/step
1/1 [=====] - 0s 24ms/step
1/1 [=====] - 0s 21ms/step
1/1 [=====] - 0s 24ms/step
1/1 [=====] - 0s 23ms/step
1/1 [=====] - 0s 23ms/step
1/1 [=====] - 0s 22ms/step
1/1 [=====] - 0s 34ms/step
1/1 [=====] - 0s 23ms/step
1/1 [=====] - 0s 22ms/step
1/1 [=====] - 0s 23ms/step
1/1 [=====] - 0s 24ms/step
1/1 [=====] - 0s 23ms/step
1/1 [=====] - 0s 21ms/step
1/1 [=====] - 0s 23ms/step
1/1 [=====] - 0s 23ms/step
1/1 [=====] - 0s 23ms/step
1/1 [=====] - 0s 24ms/step
1/1 [=====] - 0s 21ms/step
1/1 [=====] - 0s 23ms/step
1/1 [=====] - 0s 24ms/step
1/1 [=====] - 0s 23ms/step
1/1 [=====] - 0s 22ms/step
1/1 [=====] - 0s 23ms/step
1/1 [=====] - 0s 22ms/step
1/1 [=====] - 0s 22ms/step
1/1 [=====] - 0s 22ms/step

Tpput sent

Input sentence: wow:
Decoded sentence: मैं

Decoded sentence: म तुम्ह कितन पर दिवा था॥

```
1/1 [=====] - 0s 21ms/step  
1/1 [=====] - 0s 22ms/step  
1/1 [=====] - 0s 22ms/step  
1/1 [=====] - 0s 20ms/step  
1/1 [=====] - 0s 22ms/step  
1/1 [=====] - 0s 21ms/step  
*****
```

Input

Decoded sentence: मैं ब

Decoded sentence: म तुम्ह कितन पर दिया था।

```
1/1 [=====] - 0s 21ms/step  
1/1 [=====] - 0s 21ms/step  
1/1 [=====] - 0s 22ms/step  
*****
```

* * * * *

Input sentence: Jump.
Decoded sentence: 跳 =

Decoded sentence: म तुम्ह कितन पर दिया था।

Import libraries

```
In [1]: import os
import string
import numpy as np
import pandas as pd
from string import digits

import matplotlib.pyplot as plt
%matplotlib inline

import re
import logging
import tensorflow as tf

import matplotlib.ticker as ticker

from sklearn.model_selection import train_test_split
import unicodedata
import io
import time
import warnings
import sys

for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

PATH = "../input/hindienglish-corpora/Hindi_English_Truncated_Corpus.csv"

/opt/conda/lib/python3.10/site-packages/scipy/__init__.py:1
6: UserWarning: A NumPy version >=1.16.5 and <1.23.0 is required for this version of SciPy (detected version 1.23.5)
    warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}")


/kaggle/input/hindienglish-corpora/Hindi_English_Truncated_Corpus.csv
```

Preprocess English and Hindi sentences

```
In [2]: def unicode_to_ascii(s):
    return ''.join(c for c in unicodedata.normalize('NFD', s)
                  if unicodedata.category(c) != 'Mn')

def preprocess_sentence(w):
    w = unicode_to_ascii(w.lower().strip())
    w = re.sub(r"([?.!,\u202c])", r" \1 ", w)
    w = re.sub(r'\u202c[\u202c]+', " ", w)
    w = re.sub(r"[\u0600-\u06ff]", " ", w)
    w = w.rstrip().strip()
    return w

def hindi_preprocess_sentence(w):
    w = w.rstrip().strip()
    return w
```

```
In [3]: def create_dataset(path=PATH):
    lines=pd.read_csv(path,encoding='utf-8')
    lines=lines.dropna()
    lines = lines[lines['source']=='ted']
    en = []
    hd = []
    for i, j in zip(lines['english_sentence'], lines['hindi_sentence']):
        en_1 = [preprocess_sentence(w) for w in i.split(' ')]
        en_1.append('<end>')
        en_1.insert(0, '<start>')
        hd_1 = [hindu_preprocess_sentence(w) for w in j.split(' ')]
        hd_1.append('<end>')
        hd_1.insert(0, '<start>')
        en.append(en_1)
        hd.append(hd_1)
    return hd, en
```

```
In [4]: def max_length(tensor):
    return max(len(t) for t in tensor)
```

Tokenization of the data

```
In [5]: def tokenize(lang):
    lang_tokenizer = tf.keras.preprocessing.text.Tokenizer(filters='')
    lang_tokenizer.fit_on_texts(lang)
    tensor = lang_tokenizer.texts_to_sequences(lang)
    tensor = tf.keras.preprocessing.sequence.pad_sequences(tensor, padding='post')
    return tensor, lang_tokenizer
```

```
In [6]: def load_dataset(path=PATH):
    targ_lang, inp_lang = create_dataset(path)
    input_tensor, inp_lang_tokenizer = tokenize(inp_lang)
    target_tensor, targ_lang_tokenizer = tokenize(targ_lang)
    return input_tensor, target_tensor, inp_lang_tokenizer, targ_lang_tokenizer
```

```
In [7]: input_tensor, target_tensor, inp_lang, targ_lang = load_dataset(PATH)
max_length_targ, max_length_inp = max_length(target_tensor), max_length(input_tensor)
```

Create Train and Test dataset

```
In [8]: input_tensor_train, input_tensor_val, target_tensor_train, target_tensor_val = train_test_split(input_tensor, target_tensor, test_size=0.2)
print(len(input_tensor_train), len(target_tensor_train), len(input_tensor_val), len(target_tensor_val))

31904 31904 7977 7977
```

```
In [9]: def convert(lang, tensor):
    for t in tensor:
        if t!=0:
            print ("%d ----> %s" % (t, lang.index_word[t]))

print ("Input Language; index to word mapping")
convert(inp_lang, input_tensor_train[0])
print ()
print ("Target Language; index to word mapping")
convert(targ_lang, target_tensor_train[0])
```

```
Input Language; index to word mapping
1 ----> <start>
38 ----> from
7 ----> a
844 ----> university
8 ----> in
2492 ----> ghana
75 ----> has
```

```

7 -----> a
3137 -----> stronger
364 -----> sense
2 -----> <end>

Target Language; index to word mapping
1 -----> <start>
1815 -----> घाना
4 -----> के
1291 -----> विश्वविद्यालय
4 -----> के
81 -----> पास
169 -----> अधिक
714 -----> भावना
2 -----> <end>
In [10]: BUFFER_SIZE = len(input_tensor_train)
BATCH_SIZE = 64
steps_per_epoch = len(input_tensor_train) // BATCH_SIZE
embedding_dim = 128
units = 256
vocab_inp_size = len(inp_lang.word_index)+1
vocab_tar_size = len(targ_lang.word_index)+1

dataset = tf.data.Dataset.from_tensor_slices((input_tensor_train, target_tensor_train)).shuffle(BUFFER_SIZE)
dataset = dataset.batch(BATCH_SIZE, drop_remainder=True)

```

Encoder

```

In [11]: class Encoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, enc_units,
batch_sz):
        super(Encoder, self).__init__()
        self.batch_sz = batch_sz
        self.enc_units = enc_units
        self.embedding = tf.keras.layers.Embedding(vocab_size,
embedding_dim)
        self.gru = tf.keras.layers.GRU(self.enc_units,
                                         return_sequences=True,
                                         return_state=True,
                                         recurrent_initializer
='glorot_uniform')

    def call(self, x, hidden):
        x = self.embedding(x)
        output, state = self.gru(x, initial_state = hidden)
        return output, state

    def initialize_hidden_state(self):
        return tf.zeros((self.batch_sz, self.enc_units))

encoder = Encoder(vocab_inp_size, embedding_dim, units, BATCH_SIZE)

```

Attention Mechanism

```

In [12]: class BahdanauAttention(tf.keras.layers.Layer):
    def __init__(self, units):
        super(BahdanauAttention, self).__init__()
        self.W1 = tf.keras.layers.Dense(units)
        self.W2 = tf.keras.layers.Dense(units)
        self.V = tf.keras.layers.Dense(1)

    def call(self, query, values):
        hidden_with_time_axis = tf.expand_dims(query, 1)
        score = self.V(tf.nn.tanh(
            self.W1(values) + self.W2(hidden_with_time_axis)))
        attention_weights = tf.nn.softmax(score, axis=1)
        context_vector = attention_weights * values
        context_vector = tf.reduce_sum(context_vector, axis=1)
        return context_vector, attention_weights

```

Decoder

```
In [13]: class Decoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, dec_units,
batch_sz):
        super(Decoder, self).__init__()
        self.batch_sz = batch_sz
        self.dec_units = dec_units
        self.embedding = tf.keras.layers.Embedding(vocab_size,
embedding_dim)
        self.gru = tf.keras.layers.GRU(self.dec_units,
                                return_sequences=True,
                                return_state=True,
                                recurrent_initializer
='glorot_uniform')
        self.fc = tf.keras.layers.Dense(vocab_size)
        self.attention = BahdanauAttention(self.dec_units)

    def call(self, x, hidden, enc_output):
        context_vector, attention_weights = self.attention(hi
dden, enc_output)
        x = self.embedding(x)
        x = tf.concat([tf.expand_dims(context_vector, 1), x],
axis=-1)
        output, state = self.gru(x)
        output = tf.reshape(output, (-1, output.shape[2]))
        x = self.fc(output)
        return x, state, attention_weights

decoder = Decoder(vocab_tar_size, embedding_dim, units, BATCH
_SIZE)
```

Optimizer

```
In [14]: optimizer = tf.keras.optimizers.Adam()
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(f
rom_logits=True, reduction='none')

def loss_function(real, pred):
    mask = tf.math.logical_not(tf.math.equal(real, 0))
    loss_ = loss_object(real, pred)
    mask = tf.cast(mask, dtype=loss_.dtype)
    # print(type(mask))
    loss_ *= mask
    return tf.reduce_mean(loss_)
```

```
In [15]: checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(optimizer=optimizer,
                                encoder=encoder,
                                decoder=decoder)
```

```
In [16]: @tf.function
def train_step(inp, targ, enc_hidden):
    loss = 0
    with tf.GradientTape() as tape:
        enc_output, enc_hidden = encoder(inp, enc_hidden)
        dec_hidden = enc_hidden
        dec_input = tf.expand_dims([targ_lang.word_index['<st
art>']] * BATCH_SIZE, 1)
        # Teacher forcing
        for t in range(1, targ.shape[1]):
            predictions, dec_hidden, _ = decoder(dec_input, d
ec_hidden, enc_output)
            loss += loss_function(targ[:, t], predictions)
            dec_input = tf.expand_dims(targ[:, t], 1)

        batch_loss = (loss / int(targ.shape[1]))
        variables = encoder.trainable_variables + decoder.tr
ainable_variables
        gradients = tape.gradient(loss, variables)
        optimizer.apply_gradients(zip(gradients, variables))
    return batch_loss
```

```
In [17]: EPOCHS = 20

for epoch in range(EPOCHS):
    start = time.time()
    enc_hidden = encoder.initialize_hidden_state()
    total_loss = 0
    for (batch, (inp, targ)) in enumerate(dataset.take(steps_per_epoch)):
        batch_loss = train_step(inp, targ, enc_hidden)
        total_loss += batch_loss
        if batch % 100 == 0:
            print('Epoch {} Batch {} Loss {:.4f}'.format(epoch + 1, batch, batch_loss.numpy()))

    if (epoch + 1) % 2 == 0:
        checkpoint.save(file_prefix = checkpoint_prefix)

    print('Epoch {} Loss {:.4f}'.format(epoch + 1, total_loss / steps_per_epoch))
    print('Time taken for 1 epoch {} sec\n'.format(time.time() - start))

Epoch 1 Batch 0 Loss 3.0533
Epoch 1 Batch 100 Loss 2.0160
Epoch 1 Batch 200 Loss 1.7623
Epoch 1 Batch 300 Loss 1.8402
Epoch 1 Batch 400 Loss 1.9615
Epoch 1 Loss 1.9434
Time taken for 1 epoch 110.23105382919312 sec

Epoch 2 Batch 0 Loss 1.9072
Epoch 2 Batch 100 Loss 1.4948
Epoch 2 Batch 200 Loss 1.7551
Epoch 2 Batch 300 Loss 1.6660
Epoch 2 Batch 400 Loss 1.7578
Epoch 2 Loss 1.7376
Time taken for 1 epoch 42.96359419822693 sec

Epoch 3 Batch 0 Loss 1.6665
Epoch 3 Batch 100 Loss 1.6212
Epoch 3 Batch 200 Loss 1.7445
Epoch 3 Batch 300 Loss 1.5273
Epoch 3 Batch 400 Loss 1.7609
Epoch 3 Loss 1.6455
Time taken for 1 epoch 42.40199828147888 sec

Epoch 4 Batch 0 Loss 1.6872
Epoch 4 Batch 100 Loss 1.6044
Epoch 4 Batch 200 Loss 1.6098
Epoch 4 Batch 300 Loss 1.4731
Epoch 4 Batch 400 Loss 1.7359
Epoch 4 Loss 1.5652
Time taken for 1 epoch 42.76610088348389 sec
```

```
Epoch 5 Batch 0 Loss 1.4875
Epoch 5 Batch 100 Loss 1.4923
Epoch 5 Batch 200 Loss 1.5689
Epoch 5 Batch 300 Loss 1.4017
Epoch 5 Batch 400 Loss 1.5272
Epoch 5 Loss 1.4931
Time taken for 1 epoch 41.25744819641113 sec
```

```
Epoch 6 Batch 0 Loss 1.4154
Epoch 6 Batch 100 Loss 1.4712
Epoch 6 Batch 200 Loss 1.3312
Epoch 6 Batch 300 Loss 1.5176
Epoch 6 Batch 400 Loss 1.4844
Epoch 6 Loss 1.4299
Time taken for 1 epoch 41.8310809135437 sec
```

```
Epoch 7 Batch 0 Loss 1.5119
Epoch 7 Batch 100 Loss 1.4619
Epoch 7 Batch 200 Loss 1.3281
Epoch 7 Batch 300 Loss 1.3924
Epoch 7 Batch 400 Loss 1.3905
Epoch 7 Loss 1.3723
Time taken for 1 epoch 41.61006450653076 sec
```

```
Epoch 8 Batch 0 Loss 1.3693
Epoch 8 Batch 100 Loss 1.2528
Epoch 8 Batch 200 Loss 1.3115
Epoch 8 Batch 300 Loss 1.2701
Epoch 8 Batch 400 Loss 1.2277
```

```
Epoch 19 Batch 0 Loss 0.6048
Epoch 19 Batch 100 Loss 0.6244
Epoch 19 Batch 200 Loss 0.6471
Epoch 19 Batch 300 Loss 0.7039
Epoch 19 Batch 400 Loss 0.7130
Epoch 19 Loss 0.6482
Time taken for 1 epoch 41.21981954574585 sec
```

```
Epoch 20 Batch 0 Loss 0.5191
Epoch 20 Batch 100 Loss 0.5539
Epoch 20 Batch 200 Loss 0.5778
Epoch 20 Batch 300 Loss 0.6339
Epoch 20 Batch 400 Loss 0.6066
Epoch 20 Loss 0.6079
Time taken for 1 epoch 41.700024127960205 sec
```

```
In [18]: def evaluate(sentence):
    attention_plot = np.zeros((max_length_targ, max_length_in
p))
        sentence = preprocess_sentence(sentence)
        inputs = [inp_lang.word_index[i] for i in sentence.split
(' ')]
        inputs = tf.keras.preprocessing.sequence.pad_sequences([i
nputs],
                                                    maxlen=max_length_inp,
                                                    padding='post')
        inputs = tf.convert_to_tensor(inputs)
        result = ''
        hidden = [tf.zeros((1, units))]
        enc_out, enc_hidden = encoder(inputs, hidden)
        dec_hidden = enc_hidden
        dec_input = tf.expand_dims([targ_lang.word_index['<start
>']], 0)
        for t in range(max_length_targ):
            predictions, dec_hidden, attention_weights = decoder
(dec_input,
dec_hidden,
enc_out)
            predicted_id = tf.argmax(predictions[0]).numpy()
            result += targ_lang.index_word[predicted_id] + ' '
            if targ_lang.index_word[predicted_id] == '<end>':
                return result, sentence
            dec_input = tf.expand_dims([predicted_id], 0)
        return result, sentence
```

```
In [19]: def translate(sentence):
    result, sentence = evaluate(sentence)
    print('Input: %s' % (sentence))
    print('Predicted translation: {}'.format(result))
```

```
In [20]: # restoring the latest checkpoint in checkpoint_dir
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_di
r))
```

```
Out[20]: <tensorflow.python.checkpoint.checkpoint.CheckpointLoadStat
at 0x7b36093e6a70>
```

```
In [21]: translate(u'politicians do not have permission to do what n
ds to be done.')
```

Input: politicians do not have permission to do what needs
be done .

Predicted translation: वह है कि वो लोग इसे खरीदने के लिए होना चाहिए,
वह है जो कि गतियाँ कम आक्रामक हो रहा है जो वे स्वावलंबी हैं. <end>

