# **Sequence to sequence implementation**

**There will be some functions that start with the word "grader" ex: grader_check_encoder(), grader_check_attention(), grader_onestepdecoder() etc, you should not change those function definition.**

**Every Grader function has to return True.**

**Note 1:** There are many blogs on the attention mechanisum which might be misleading you, so do read the references completly and after that only please check the internet. The best things is to read the research papers and try to implement it on your own.

**Note 2:** To complete this assignment, the reference that are mentioned will be enough.

**Note 3:** If you are starting this assignment, you might have completed minimum of 20 assignment. If you are still not able to implement this algorithm you might have rushed in the previous assignments with out learning much and didn't spend your time productively.

## Task -1: Simple Encoder and Decoder

Implement simple Encoder-Decoder model

1. Download the **Italian** to **English** translation dataset from here (http://www.manythings.org/anki/ita-eng.zip)
2. You will find **ita.txt** file in that ZIP, you can read that data using python and preprocess that data this way only:

```
Encoder input: "<start> vado a scuola <end>"
Decoder input: "<start> i am going school"
Decoder output: "i am going school <end>"
```

3. You have to implement a simple Encoder and Decoder architecture
4. Use BLEU score as metric to evaluate your model. You can use any loss function you need.
5. You have to use Tensorboard to plot the Graph, Scores and histograms of gradients.
6. a. Check the reference notebook
   b. Resource 2 (https://medium.com/analytics-vidhya/understand-sequence-to-sequence-models-in-a-more-intuitive-way-1d517d8795bb)

In [1]:
```
!nvidia-smi
```

```
Sun Jul 11 13:22:14 2021
+-----------------------------------------------------------------------------
--------+
| NVIDIA-SMI 470.42.01    Driver Version: 460.32.03    CUDA Version: 1
1.2     |
|-------------------------------+----------------------+----------------
--------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Unco
rr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Com
pute M. |
|                               |                      |
MIG M. |
|===============================+======================+==============
========|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |
0 |
| N/A   55C    P8    10W /  70W |      0MiB / 15109MiB |      0%
Default |
|                               |                      |
N/A |
+-------------------------------+----------------------+----------------
--------+

+-----------------------------------------------------------------------------
--------+
| Processes:
|
|  GPU   GI   CI        PID   Type   Process name                  GPU
Memory |
|        ID   ID                                                   Usa
ge      |
|=============================================================================
========|
|  No running processes found
|
+-----------------------------------------------------------------------------
--------+
```

In [2]:
```
import matplotlib.pyplot as plt
%matplotlib inline
# import seaborn as sns
import pandas as pd
import re
import tensorflow as tf
from tensorflow.keras.layers import Embedding, LSTM, Dense, Reshape, S
oftmax, Dot, Concatenate
from tensorflow.keras.models import Model
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import numpy as np
import seaborn as sns
```

**Load the data**

In [3]:
```
!wget http://www.manythings.org/anki/ita-eng.zip
!unzip ita-eng.zip
```

```
--2021-07-11 13:22:22--  http://www.manythings.org/anki/ita-eng.zip
Resolving www.manythings.org (www.manythings.org)... 104.21.55.222, 17
2.67.173.198, 2606:4700:3031::6815:37de, ...
Connecting to www.manythings.org (www.manythings.org)|104.21.55.222|:8
0... connected.
HTTP request sent, awaiting response... 200 OK
Length: 7692825 (7.3M) [application/zip]
Saving to: ‘ita-eng.zip’

ita-eng.zip         100%[===================>]   7.34M  22.8MB/s    in
0.3s

2021-07-11 13:22:22 (22.8 MB/s) - ‘ita-eng.zip’ saved [7692825/769282
5]

Archive:  ita-eng.zip
  inflating: ita.txt
  inflating: _about.txt
```

In [4]:
```
with open('ita.txt', 'r', encoding="utf8") as f:
    eng=[]
    ita=[]
    for i in f.readlines():
        eng.append(i.split("\t")[0])
        ita.append(i.split("\t")[1])
data = pd.DataFrame(data=list(zip(eng, ita)), columns=['english','ital
ian'])
print(data.shape)
data.head()
```

```
(350360, 2)
```

Out[4]:

| | english | italian |
|---|---|---|
| **0** | Hi. | Ciao! |
| **1** | Hi. | Ciao. |
| **2** | Run! | Corri! |
| **3** | Run! | Corra! |
| **4** | Run! | Correte! |

**Preprocess data**

In [5]:
```
def decontractions(phrase):
    """decontracted takes text and convert contractions into natural f
orm.
     ref: https://stackoverflow.com/questions/19790188/expanding-engli
sh-language-contractions-in-python/47091490#47091490"""
    # specific
    phrase = re.sub(r"won\'t", "will not", phrase)
    phrase = re.sub(r"can\'t", "can not", phrase)
    phrase = re.sub(r"won’t", "will not", phrase)
    phrase = re.sub(r"can\’t", "can not", phrase)

    # general
    phrase = re.sub(r"n\'t", " not", phrase)
    phrase = re.sub(r"\'re", " are", phrase)
    phrase = re.sub(r"\'s", " is", phrase)
    phrase = re.sub(r"\'d", " would", phrase)
    phrase = re.sub(r"\'ll", " will", phrase)
    phrase = re.sub(r"\'t", " not", phrase)
    phrase = re.sub(r"\'ve", " have", phrase)
    phrase = re.sub(r"\'m", " am", phrase)

    phrase = re.sub(r"n\’t", " not", phrase)
    phrase = re.sub(r"\’re", " are", phrase)
    phrase = re.sub(r"\’s", " is", phrase)
    phrase = re.sub(r"\’d", " would", phrase)
    phrase = re.sub(r"\’ll", " will", phrase)
    phrase = re.sub(r"\’t", " not", phrase)
    phrase = re.sub(r"\’ve", " have", phrase)
    phrase = re.sub(r"\’m", " am", phrase)

    return phrase

def preprocess(text):
    text = decontractions(text)
    text = re.sub('[^A-Za-z0-9 ]+', '', text)
    return text

def preprocess_ita(text):
    text = text.lower()
    text = decontractions(text)
    text = re.sub('[$)\?"’.°!;\'€%:,(/]', '', text)
    text = re.sub('\u200b', ' ', text)
    text = re.sub('\xa0', ' ', text)
    text = re.sub('-', ' ', text)
    return text


data['english'] = data['english'].apply(preprocess)
data['italian'] = data['italian'].apply(preprocess_ita)
data.head()
```

Out[5]:

| | english | italian |
|---|---|---|
| **0** | Hi | ciao |
| **1** | Hi | ciao |
| **2** | Run | corri |
| **3** | Run | corra |
| **4** | Run | correte |

In [6]:
```
ita_lengths = data['italian'].str.split().apply(len)
eng_lengths = data['english'].str.split().apply(len)
```

In [7]:
```
data['italian_len'] = data['italian'].str.split().apply(len)
data = data[data['italian_len'] < 20]

data['english_len'] = data['english'].str.split().apply(len)
data = data[data['english_len'] < 20]

data['english_inp'] = '<start> ' + data['english'].astype(str)
data['english_out'] = data['english'].astype(str) + ' <end>'

data = data.drop(['english','italian_len','english_len'], axis=1)
# only for the first sentance add a toke <end> so that we will have <e
nd> in tokenizer
data.head()
```

Out[7]:

| | italian | english_inp | english_out |
|---|---|---|---|
| **0** | ciao | <start> Hi | Hi <end> |
| **1** | ciao | <start> Hi | Hi <end> |
| **2** | corri | <start> Run | Run <end> |
| **3** | corra | <start> Run | Run <end> |
| **4** | correte | <start> Run | Run <end> |

In [8]:
```
data.sample(10)
```

Out[8]:

| | italian | english_inp | english_out |
|---|---|---|---|
| **99198** | non sei mia amica | <start> You are not my friend | You are not my friend <end> |
| **102728** | vengo qua ogni giorno | <start> I come here every day | I come here every day <end> |
| **216062** | non era facile trovare loro | <start> It was not easy to find gold | It was not easy to find gold <end> |
| **122107** | è chiaro a tutti | <start> It is clear to everyone | It is clear to everyone <end> |
| **154422** | ora cominciamo la partita | <start> Now let is begin the game | Now let is begin the game <end> |
| **102143** | hey dovè andato tom | <start> Hey where did Tom go | Hey where did Tom go <end> |
| **264812** | spero davvero che ti sia piaciuta la cena | <start> I do hope you enjoyed the dinner | I do hope you enjoyed the dinner <end> |
| **105379** | io sarò impegnato domani | <start> I will be busy tomorrow | I will be busy tomorrow <end> |
| **229619** | io andrò a trovare tom domani | <start> I am going to see Tom tomorrow | I am going to see Tom tomorrow <end> |
| **249271** | perché siete così educate | <start> Why are you being so courteous | Why are you being so courteous <end> |

# Tokenizer

In [9]:
```
from sklearn.model_selection import train_test_split
train, validation = train_test_split(data, test_size=0.2,random_state=0)
```

In [10]:
```
!gdown --id "1B7420H8cRMhufU4MfrOcNAiPTWv09wcc"
!gdown --id "1nki-SdUOFHhXgEwp6qSkMWSwMeJKloSf"
```

```
Downloading...
From: https://drive.google.com/uc?id=1B7420H8cRMhufU4MfrOcNAiPTWv09wcc
To: /content/train_attention.csv
27.9MB [00:00, 89.1MB/s]
Downloading...
From: https://drive.google.com/uc?id=1nki-SdUOFHhXgEwp6qSkMWSwMeJKloSf
To: /content/validation_attention.csv
6.98MB [00:00, 59.5MB/s]
```

In [11]:
```
# train.to_csv("train_attention.csv",index=None)
# validation.to_csv("validation_attention.csv",index=None)

# !cp "train_attention.csv" "/content/drive/MyDrive/datasets"
# !cp "validation_attention.csv" "/content/drive/MyDrive/datasets"

train = pd.read_csv("train_attention.csv")
validation = pd.read_csv("validation_attention.csv")
```

In [12]:
```
print(train.shape, validation.shape)
# for one sentence we will be adding <end> token so that the tokanizer learns the word <end>
# with this we can use only one tokenizer for both encoder output and decoder output
train.iloc[0]['english_inp']= str(train.iloc[0]['english_inp'])+' <end>'
train.iloc[0]['english_out']= str(train.iloc[0]['english_out'])+' <end>'
```

```
(279900, 3) (69975, 3)
```

In [13]:
```
train.head()
```

Out[13]:

| | italian | english_inp | english_out |
|---|---|---|---|
| **0** | è ovvio che lui ha ragione | <start> It is obvious that he is right <end> <... | It is obvious that he is right <end> <end> <end> |
| **1** | trovai il mio portafoglio smarrito | <start> I found my lost wallet | I found my lost wallet <end> |
| **2** | tu sei spericolato | <start> You are reckless | You are reckless <end> |
| **3** | nessuno ti capirà | <start> No one is going to understand you | No one is going to understand you <end> |
| **4** | morditi la lingua | <start> Bite your tongue | Bite your tongue <end> |

In [14]:
```
tknizer_ita = Tokenizer(filters='!"#$%&()*+,-./:;=?@[\\]^_`{|}~\t\n')
tknizer_ita.fit_on_texts(train['italian'].values)
tknizer_eng = Tokenizer(filters='!"#$%&()*+,-./:;=?@[\\]^_`{|}~\t\n')
tknizer_eng.fit_on_texts(train['english_inp'].values)
```

In [15]:
```
vocab_size_eng=len(tknizer_eng.word_index.keys())
print(vocab_size_eng)
vocab_size_ita=len(tknizer_ita.word_index.keys())
print(vocab_size_ita)
```

```
13001
26424
```

In [16]:
```
tknizer_eng.word_index['<start>'], tknizer_eng.word_index['<end>']
```

Out[16]:
```
(1, 8480)
```

In [17]:
```
!wget https://www.dropbox.com/s/ddkmtqz01jc024u/glove.6B.100d.txt
```

```
--2021-07-11 13:22:56--  https://www.dropbox.com/s/ddkmtqz01jc024u/glo
ve.6B.100d.txt
Resolving www.dropbox.com (www.dropbox.com)... 162.125.6.18, 2620:100:
6019:18::a27d:412
Connecting to www.dropbox.com (www.dropbox.com)|162.125.6.18|:443... c
onnected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: /s/raw/ddkmtqz01jc024u/glove.6B.100d.txt [following]
--2021-07-11 13:22:56--  https://www.dropbox.com/s/raw/ddkmtqz01jc024
u/glove.6B.100d.txt
Reusing existing connection to www.dropbox.com:443.
HTTP request sent, awaiting response... 302 Found
Location: https://uc4b345e395f837f8c50c6403da7.dl.dropboxusercontent.c
om/cd/0/inline/BSEzV1NzDUWORm938xgjk26Rh9ExQZM5bNip1NJ9NFtQZN3kII7t7Hp
Ngza_2nyl3rShoCzJZs9t3xaLd1Wod6vMnkG_XcWIdANTi5fBFudTCUyVFdxatGiiuiqJF
TXtKA007IdbD9g-7z1XaVDG76ie/file# [following]
--2021-07-11 13:22:56--  https://uc4b345e395f837f8c50c6403da7.dl.dropb
oxusercontent.com/cd/0/inline/BSEzV1NzDUWORm938xgjk26Rh9ExQZM5bNip1NJ9
NFtQZN3kII7t7HpNgza_2nyl3rShoCzJZs9t3xaLd1Wod6vMnkG_XcWIdANTi5fBFudTCU
yVFdxatGiiuiqJFTXtKA007IdbD9g-7z1XaVDG76ie/file
Resolving uc4b345e395f837f8c50c6403da7.dl.dropboxusercontent.com (uc4b
345e395f837f8c50c6403da7.dl.dropboxusercontent.com)... 162.125.6.15, 2
620:100:6019:15::a27d:40f
Connecting to uc4b345e395f837f8c50c6403da7.dl.dropboxusercontent.com
(uc4b345e395f837f8c50c6403da7.dl.dropboxusercontent.com)|162.125.6.15
|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 347116733 (331M) [text/plain]
Saving to: ‘glove.6B.100d.txt’

glove.6B.100d.txt   100%[===================>] 331.04M   131MB/s    in
2.5s

2021-07-11 13:22:59 (131 MB/s) - ‘glove.6B.100d.txt’ saved [347116733/
347116733]
```

In [18]:
```
embeddings_index = dict()
f = open('glove.6B.100d.txt')
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()

embedding_matrix = np.zeros((vocab_size_eng+1, 100))
for word, i in tknizer_eng.word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        embedding_matrix[i] = embedding_vector
```

# **Implement custom encoder decoder**

In [19]:
```
class Dataset:
    def __init__(self, data, tknizer_ita, tknizer_eng, max_len):
        self.encoder_inps = data['italian'].values
        self.decoder_inps = data['english_inp'].values
        self.decoder_outs = data['english_out'].values
        self.tknizer_eng = tknizer_eng
        self.tknizer_ita = tknizer_ita
        self.max_len = max_len

    def __getitem__(self, i):
        self.encoder_seq = self.tknizer_ita.texts_to_sequences([self.e
ncoder_inps[i]]) # need to pass list of values
        self.decoder_inp_seq = self.tknizer_eng.texts_to_sequences([se
lf.decoder_inps[i]])
        self.decoder_out_seq = self.tknizer_eng.texts_to_sequences([se
lf.decoder_outs[i]])

        self.encoder_seq = pad_sequences(self.encoder_seq, maxlen=self
.max_len, dtype='int32', padding='post')
        self.decoder_inp_seq = pad_sequences(self.decoder_inp_seq, max
len=self.max_len, dtype='int32', padding='post')
        self.decoder_out_seq = pad_sequences(self.decoder_out_seq, max
len=self.max_len, dtype='int32', padding='post')
        return self.encoder_seq, self.decoder_inp_seq, self.decoder_ou
t_seq

    def __len__(self): # your model.fit_gen requires this function
        return len(self.encoder_inps)


class Dataloder(tf.keras.utils.Sequence):
    def __init__(self, dataset, batch_size=1):
        self.dataset = dataset
        self.batch_size = batch_size
        self.indexes = np.arange(len(self.dataset.encoder_inps))


    def __getitem__(self, i):
        start = i * self.batch_size
        stop = (i + 1) * self.batch_size
        data = []
        for j in range(start, stop):
            data.append(self.dataset[j])

        batch = [np.squeeze(np.stack(samples, axis=1), axis=0) for sam
ples in zip(*data)]
        # we are creating data like ([italian, english_inp], english_o
ut) these are already converted into seq
        return tuple([[batch[0],batch[1]],batch[2]])

    def __len__(self):  # your model.fit_gen requires this function
        return len(self.indexes) // self.batch_size

    def on_epoch_end(self):
        self.indexes = np.random.permutation(self.indexes)
```

In [20]:
```
print(len(tknizer_ita.word_counts))
print(len(tknizer_eng.word_counts))
```

```
26424
13001
```

In [21]:
```
train_dataset = Dataset(train, tknizer_ita, tknizer_eng, 20)
test_dataset  = Dataset(validation, tknizer_ita, tknizer_eng, 20)

train_dataloader = Dataloder(train_dataset, batch_size=1024)
test_dataloader = Dataloder(test_dataset, batch_size=1024)


print(train_dataloader[0][0][0].shape, train_dataloader[0][0][1].shape
, train_dataloader[0][1].shape)

# train_dataloader[0][0][0] -> eng_input (not available in testing)
# train_dataloader[0][0][0] -> eng_out (sentence to predict)
```

```
(1024, 20) (1024, 20) (1024, 20)
```

**Encoder**

In [22]:
```
class Encoder(tf.keras.Model):
    '''
    Encoder model -- That takes a input sequence and returns encoder-o
utputs,encoder_final_state_h,encoder_final_state_c
    '''

    def __init__(self,inp_vocab_size,embedding_size,lstm_size,input_le
ngth):
        #Initialize Embedding layer
        #Intialize Encoder LSTM layer
        super().__init__()
        self.inp_vocab_size = inp_vocab_size
        self.embedding_size = embedding_size
        self.input_length = input_length
        self.lstm_units= lstm_size
        self.lstm_output = 0
        self.lstm_state_h=0
        self.lstm_state_c=0

    def build(self, input_shape):
        self.embedding = Embedding(input_dim=self.inp_vocab_size, outp
ut_dim=self.embedding_size, input_length=self.input_length,
                                   mask_zero=True, name="embedding_layer_encod
er")
        self.lstm = LSTM(self.lstm_units, return_state=True, return_se
quences=True, name="Encoder_LSTM")


    def call(self,input_sequence,states=[]):
        '''
          This function takes a sequence input and the initial states
 of the encoder.
          Pass the input_sequence input to the Embedding layer, Pass t
he embedding layer ouput to encoder_lstm
          returns -- encoder_output, last time step's hidden and cell
 state
        '''
        input_embedding = self.embedding(input_sequence)
        self.lstm_output, self.lstm_state_h, self.lstm_state_c = self.
lstm(input_embedding)
        return self.lstm_output, self.lstm_state_h, self.lstm_state_c

    def initialize_states(self,batch_size):
      '''
      Given a batch size it will return intial hidden state and intial
cell state.
      If batch size is 32- Hidden state is zeros of size [32,lstm_unit
s], cell state zeros is of size [32,lstm_units]
      '''
      return np.zeros(shape=(batch_size,self.lstm_units))
```

In [23]:
```
np.zeros(shape=(2000,64)).shape
```

Out[23]:
```
(2000, 64)
```

**Grader function - 1**

In [24]:
```
def grader_check_encoder():
    '''
        vocab-size: Unique words of the input language,
        embedding_size: output embedding dimension for each word after embedding layer,
        lstm_size: Number of lstm units,
        input_length: Length of the input sentence,
        batch_size
    '''
    vocab_size=10
    embedding_size=20
    lstm_size=32
    input_length=10
    batch_size=16
    #Intialzing encoder
    encoder=Encoder(vocab_size,embedding_size,lstm_size,input_length)
    input_sequence=tf.random.uniform(shape=[batch_size,input_length],maxval=vocab_size,minval=0,dtype=tf.int32)
    print("Input seq : ",input_sequence.shape)
    #Intializing encoder initial states
    initial_state=encoder.initialize_states(batch_size)

    encoder_output,state_h,state_c=encoder(input_sequence,initial_state)

    assert(encoder_output.shape==(batch_size,input_length,lstm_size) and state_h.shape==(batch_size,lstm_size) and state_c.shape==(batch_size,lstm_size))
    return True
print(grader_check_encoder())
```

```
Input seq :  (16, 10)
True
```

In [25]:
```
class Decoder(tf.keras.Model):
    '''
    Encoder model -- That takes a input sequence and returns output se
quence
    '''

    def __init__(self,out_vocab_size,embedding_size,lstm_size,input_le
ngth):

        #Initialize Embedding layer
        #Intialize Decoder LSTM layer
        super().__init__()
        self.out_vocab_size = out_vocab_size
        self.embedding_size = embedding_size
        self.dec_units = lstm_size
        self.input_length = input_length

    def build(self, input_shape):
        self.embedding = Embedding(input_dim=self.out_vocab_size, outp
ut_dim=self.embedding_size, input_length=self.input_length,
                                   mask_zero=True, name="embedding_layer_decod
er")
        self.lstm = LSTM(self.dec_units, return_sequences=True, return
_state=True, name="Decoder_LSTM")


    def call(self,input_sequence,initial_states):
        '''
          This function takes a sequence input and the initial states
 of the encoder.
          Pass the input_sequence input to the Embedding layer, Pass t
he embedding layer ouput to decoder_lstm

          returns -- decoder_output,decoder_final_state_h,decoder_fina
l_state_c
        '''
        target_embed = self.embedding(input_sequence)
        # print("Dec-Target Embedding shape: ",target_embed.shape)
        # print("Dec-Initial stape: ",initial_states[0].shape,initial_
states[1].shape)
        # print(initial_states.shape)
        # decoder_output, decoder_final_state_h,decoder_final_state_c
 = self.lstm(target_embedd, initial_state=[state_h, state_c])
        decoder_output, decoder_final_state_h,decoder_final_state_c =
self.lstm(target_embed, initial_state=initial_states)
        return decoder_output, decoder_final_state_h, decoder_final_st
ate_c
```

**Grader function - 2**

In [26]:
```
def grader_decoder():
    '''
        out_vocab_size: Unique words of the target language,
        embedding_size: output embedding dimension for each word after
embedding layer,
        dec_units: Number of lstm units in decoder,
        input_length: Length of the input sentence,
        batch_size
    '''
    out_vocab_size=13
    embedding_dim=12
    input_length=10
    dec_units=16
    batch_size=32

    target_sentences=tf.random.uniform(shape=(batch_size,input_length
),maxval=10,minval=0,dtype=tf.int32)
    encoder_output=tf.random.uniform(shape=[batch_size,input_length,de
c_units])
    state_h=tf.random.uniform(shape=[batch_size,dec_units])
    state_c=tf.random.uniform(shape=[batch_size,dec_units])
    states=[state_h,state_c]
    decoder=Decoder(out_vocab_size, embedding_dim, dec_units,input_len
gth )
    print(target_sentences.shape)
    output,_,_=decoder(target_sentences, states)
    assert(output.shape==(batch_size,input_length,dec_units))
    return True
print(grader_decoder())
```

```
(32, 10)
True
```

In [27]:
```
class Encoder_decoder(tf.keras.Model):

    def __init__(self,encoder_inputs_length,decoder_inputs_length, ita
_vocab_size, eng_vocab_size):

        super().__init__()
        self.encoder_inputs_length =encoder_inputs_length
        self.decoder_inputs_length = decoder_inputs_length
        self.output_vocab_size = eng_vocab_size
        self.input_vocab_size  = ita_vocab_size



    def build(self, input_shape):
      # self,inp_vocab_size,embedding_size,lstm_size,input_length)
      self.encoder = Encoder(inp_vocab_size=self.input_vocab_size, emb
edding_size=20, input_length=self.encoder_inputs_length, lstm_size=64)
      # (self,out_vocab_size,embedding_size,lstm_size,input_length)
      self.decoder = Decoder(out_vocab_size=self.output_vocab_size, em
bedding_size=20, input_length=self.decoder_inputs_length, lstm_size=64
)
      self.dense   = Dense(self.output_vocab_size, activation='softma
x',name="Enc_Dec_Dense")


    def call(self,data):
        '''
        A. Pass the input sequence to Encoder layer -- Return encoder_
output,encoder_final_state_h,encoder_final_state_c
        B. Pass the target sequence to Decoder layer with intial state
s as encoder_final_state_h,encoder_final_state_C
        C. Pass the decoder_outputs into Dense layer
        Return decoder_outputs
        '''
        input,output = data[0], data[1]
        encoder_output, encoder_h, encoder_c = self.encoder(input, sel
f.encoder.initialize_states(batch_size=1024)) # you need to pass state
s too
        decoder_output, decoder_h, decoder_h= self.decoder(output, [en
coder_h, encoder_c])
        output= self.dense(decoder_output)
        return output
```

In [34]:
```
train_dataset = Dataset(train, tknizer_ita, tknizer_eng, 20)
test_dataset  = Dataset(validation, tknizer_ita, tknizer_eng, 20)

train_dataloader = Dataloder(train_dataset, batch_size=1024)
test_dataloader = Dataloder(test_dataset, batch_size=1024)
```

In [40]:
```
file_path = "simple_attention.h5"
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=file_path,
    save_weights_only=True,
    monitor='loss',
    mode='auto',
    save_best_only=False,
    save_freq='epoch')

model.load_weights("/content/simple_attention_30.h5")

class SaveDrive(tf.keras.callbacks.Callback):
   def on_epoch_end(self, epoch, logs=None):
        !cp "simple_attention.h5" "drive/MyDrive/datasets"
```

In [36]:
```
# encoder_inputs_length,decoder_inputs_length, output_vocab_size, eng_vocab_size
model  = Encoder_decoder(encoder_inputs_length=20,decoder_inputs_length=20,ita_vocab_size=vocab_size_ita+1, eng_vocab_size = vocab_size_eng+1)
optimizer = tf.keras.optimizers.Adam()
model.compile(optimizer=optimizer,loss='sparse_categorical_crossentropy')
```

In [ ]:
```
tf.config.run_functions_eagerly(True)
model_hist = model.fit_generator(train_dataloader, epochs=30, validati
on_data=test_dataloader, callbacks=[model_checkpoint_callback, SaveDri
ve()])
# model.summary()
```

```
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/
training.py:1940: UserWarning: `Model.fit_generator` is deprecated and
will be removed in a future version. Please use `Model.fit`, which sup
ports generators.
  warnings.warn('`Model.fit_generator` is deprecated and '
/usr/local/lib/python3.7/dist-packages/tensorflow/python/data/ops/data
set_ops.py:3704: UserWarning: Even though the `tf.config.experimental_
run_functions_eagerly` option is set, this option does not apply to t
f.data functions. To force eager execution of tf.data functions, pleas
e use `tf.data.experimental.enable.debug_mode()`.
  "Even though the `tf.config.experimental_run_functions_eagerly` "
```

```
Epoch 1/30
273/273 [==============================] - 104s 382ms/step - loss: 2.0
659 - val_loss: 1.8046
Epoch 2/30
273/273 [==============================] - 104s 381ms/step - loss: 1.7
453 - val_loss: 1.6617
Epoch 3/30
273/273 [==============================] - 104s 381ms/step - loss: 1.6
190 - val_loss: 1.5763
Epoch 4/30
273/273 [==============================] - 105s 383ms/step - loss: 1.5
395 - val_loss: 1.4940
Epoch 5/30
273/273 [==============================] - 104s 381ms/step - loss: 1.4
546 - val_loss: 1.4137
Epoch 6/30
273/273 [==============================] - 104s 381ms/step - loss: 1.3
748 - val_loss: 1.3360
Epoch 7/30
273/273 [==============================] - 103s 379ms/step - loss: 1.2
889 - val_loss: 1.2479
Epoch 8/30
273/273 [==============================] - 104s 382ms/step - loss: 1.2
051 - val_loss: 1.1759
Epoch 9/30
273/273 [==============================] - 104s 382ms/step - loss: 1.1
384 - val_loss: 1.1181
Epoch 10/30
273/273 [==============================] - 104s 381ms/step - loss: 1.0
799 - val_loss: 1.0639
Epoch 11/30
273/273 [==============================] - 104s 381ms/step - loss: 1.0
275 - val_loss: 1.0177
Epoch 12/30
273/273 [==============================] - 105s 383ms/step - loss: 0.9
796 - val_loss: 0.9741
Epoch 13/30
273/273 [==============================] - 104s 381ms/step - loss: 0.9
349 - val_loss: 0.9343
Epoch 14/30
273/273 [==============================] - 104s 382ms/step - loss: 0.8
942 - val_loss: 0.8991
Epoch 15/30
273/273 [==============================] - 104s 382ms/step - loss: 0.8
580 - val_loss: 0.8687
Epoch 16/30
273/273 [==============================] - 105s 383ms/step - loss: 0.8
261 - val_loss: 0.8411
Epoch 17/30
273/273 [==============================] - 104s 382ms/step - loss: 0.7
973 - val_loss: 0.8167
Epoch 18/30
273/273 [==============================] - 104s 382ms/step - loss: 0.7
711 - val_loss: 0.7952
Epoch 19/30
273/273 [==============================] - 104s 381ms/step - loss: 0.7
467 - val_loss: 0.7744
```

```
Epoch 20/30
273/273 [==============================] - 105s 383ms/step - loss: 0.7
243 - val_loss: 0.7556
Epoch 21/30
273/273 [==============================] - 105s 383ms/step - loss: 0.7
035 - val_loss: 0.7385
Epoch 22/30
273/273 [==============================] - 104s 382ms/step - loss: 0.6
836 - val_loss: 0.7218
Epoch 23/30
273/273 [==============================] - 104s 382ms/step - loss: 0.6
645 - val_loss: 0.7059
Epoch 24/30
273/273 [==============================] - 104s 383ms/step - loss: 0.6
464 - val_loss: 0.6905
Epoch 25/30
273/273 [==============================] - 104s 381ms/step - loss: 0.6
293 - val_loss: 0.6765
Epoch 26/30
273/273 [==============================] - 104s 382ms/step - loss: 0.6
133 - val_loss: 0.6642
Epoch 27/30
273/273 [==============================] - 104s 381ms/step - loss: 0.5
985 - val_loss: 0.6517
Epoch 28/30
273/273 [==============================] - 105s 383ms/step - loss: 0.5
845 - val_loss: 0.6404
Epoch 29/30
273/273 [==============================] - 104s 381ms/step - loss: 0.5
713 - val_loss: 0.6302
Epoch 30/30
273/273 [==============================] - 104s 382ms/step - loss: 0.5
588 - val_loss: 0.6198
```

In [41]:
```
# Create an object of encoder_decoder Model class,
# Compile the model and fit the model
model.layers
```

Out[41]:
```
[<__main__.Encoder at 0x7ff002857650>,
 <__main__.Decoder at 0x7ff002a25910>,
 <tensorflow.python.keras.layers.core.Dense at 0x7ff002a25d50>]
```

In [42]:
```
def predict(input_sentence):
  '''
  A. Given input sentence, convert the sentence into integers using to
kenizer used earlier
  B. Pass the input_sequence to encoder. we get encoder_outputs, last
 time step hidden and cell state
  C. Initialize index of <start> as input to decoder. and encoder fina
l states as input_states to decoder
  D. till we reach max_length of decoder or till the model predicted w
ord <end>:
         predicted_out,state_h,state_c=model.layers[1](dec_input,state
s)
         pass the predicted_out to the dense layer
         update the states=[state_h,state_c]
         And get the index of the word with maximum probability of the
dense layer output, using the tokenizer(word index) get the word and t
hen store it in a string.
         Update the input_to_decoder with current predictions
  F. Return the predicted sentence
  '''
  input_length = 20
  lstm_units = 64
  encoder_seq = tknizer_ita.texts_to_sequences([input_sentence]) # nee
d to pass list of values
  encoder_seq = pad_sequences(encoder_seq, maxlen=input_length, dtype=
'int32', padding='post')
  # print("TOKENIZED DATA :",encoder_seq.shape)

  encoder_output, enc_h, enc_c  = model.layers[0](encoder_seq, np.zero
s(shape=(1,64)))
  # print("ENCODER OUTPUT SHAPES: ",encoder_output.shape, enc_h.shape,
enc_c.shape)

  dec_input = np.array([[1]+[0]*19]) # max_len=20 and index of <start>
is 1
  # print(dec_input.shape)
  states = [enc_h, enc_c]
  result_sentence = ""

  # print("DECODER PART STARTS....")
  for i in range(20):
    # print(dec_input[0])
    predicted_out,state_h,state_c=model.layers[1](dec_input,states)
    dense_op = model.layers[2](predicted_out)
    states = [state_h,state_c] #update states to be used in next iter

    pred_index = np.argmax(dense_op.numpy()[0][0])
    pred_word = tknizer_eng.index_word[pred_index] # get predicted wor
d
    if pred_word == "<end>":
      break
    result_sentence += pred_word+" "
    # print("Output for TS %d = %s"%(i, str(pred_word)))

    # give index of pred_word as input to next timestep
    dec_input[0][i] = pred_index
```

```
  return result_sentence
```

In [44]:
```
import nltk.translate.bleu_score as bleu
import random


def aveg_bleu_scores(test_data,n):
  sample_list = random.sample(range(len(test_data)),n)
  average_bleu = 0
  pred_data = []

  for i in sample_list:
    test_sentence, true_sentence = test_data.iloc[i,[0,1]]
    pred_sentence = predict(test_sentence)
    bleu_score = bleu.sentence_bleu([true_sentence.split()],pred_sente
nce.split())
    average_bleu += bleu_score

    pred_data.append((true_sentence, pred_sentence))

  return (average_bleu/n, pred_data)
```

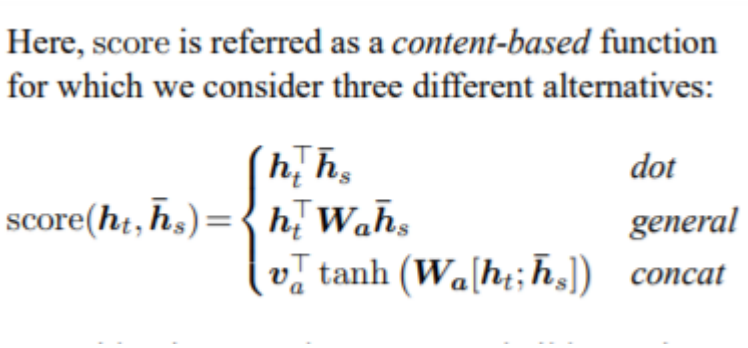
In [75]:
```
# Predict on 1000 random sentences on test data and calculate the aver
age BLEU score of these sentences.
# https://www.nltk.org/_modules/nltk/translate/bleu_score.html
import warnings
warnings.filterwarnings('ignore')

test_df = validation.copy()
test_df = test_df[["italian","english_out"]]
test_df["english_out"] = test_df["english_out"].apply(lambda x: x.spli
t("<end>")[0])

average_score, all_scored = aveg_bleu_scores(test_df,1000)
print("BLEU Score: ",average_score)
```

```
BLEU Score:  0.5506953149031838
```

## Task -2: Including Attention mechanisum

1. Use the preprocessed data from Task-1
2. You have to implement an Encoder and Decoder architecture with attention as discussed in the reference notebook.

    - Encoder - with 1 layer LSTM
    - Decoder - with 1 layer LSTM
    - attention - (Please refer the **reference notebook** (https://drive.google.com/file/d/1z_bnc-3aubKawbR6q8wyI6Mh5ho2R1aZ/view?usp=sharing) to know more about the attention mechanism.)
3. In Global attention, we have 3 types of scoring functions(as discussed in the reference notebook). As a part of this assignment **you need to create 3 models for each scoring function**




    - In model 1 you need to implemnt "dot" score function
    - In model 2 you need to implemnt "general" score function
    - In model 3 you need to implemnt "concat" score function.

    **Please do add the markdown titles for each model so that we can have a better look at the code and verify.**
4. It is mandatory to train the model with simple model.fit() only, Donot train the model with custom GradientTape()
5. Using attention weights, you can plot the attention plots, please plot those for 2-3 examples. You can check about those in this (https://www.tensorflow.org/tutorials/text/nmt_with_attention#translate)

## **Implement custom encoder decoder and attention layers**

**Encoder**

In [76]:
```
class Encoder(tf.keras.Model):
    '''
    Encoder model -- That takes a input sequence and returns output se
quence
    '''

    def __init__(self,inp_vocab_size,embedding_size,lstm_size,input_le
ngth):
      super().__init__()
      self.inp_vocab_size = inp_vocab_size
      self.embedding_size = embedding_size
      self.lstm_size = lstm_size
      self.input_length = input_length

    def build(self, input_shape):
      self.embedding = Embedding(input_dim=self.inp_vocab_size, output
_dim=self.embedding_size,
                                 input_length = self.input_length, mas
k_zero=True,name="Encoder_Embedding_Layer")
      self.lstm = LSTM(units=self.lstm_size, return_state = True, retu
rn_sequences=True, name="Encoder_LSTM_Layer")

    def call(self,input_sequence,states):
      '''
          This function takes a sequence input and the initial states
 of the encoder.
          Pass the input_sequence input to the Embedding layer, Pass t
he embedding layer ouput to encoder_lstm
          returns -- All encoder_outputs, last time steps hidden and c
ell state
      '''
      embed = self.embedding(input_sequence)
      encoder_output, encoder_h, encoder_c = self.lstm(embed,initial_s
tate=[states[0], states[1]])
      return encoder_output, encoder_h, encoder_c


    def initialize_states(self,batch_size):
      '''
      Given a batch size it will return intial hidden state and intial
cell state.
      If batch size is 32- Hidden state is zeros of size [32,lstm_unit
s], cell state zeros is of size [32,lstm_units]
      '''
      initial_h = np.zeros(shape=(batch_size, self.lstm_size))
      initial_c = np.zeros(shape=(batch_size, self.lstm_size))

      initial_h = tf.convert_to_tensor(initial_h, dtype=tf.float32)
      initial_c = tf.convert_to_tensor(initial_c, dtype=tf.float32)

      return (initial_h, initial_c)
```

**Grader function - 1**

In [77]:
```
def grader_check_encoder():

    '''
        vocab-size: Unique words of the input language,
        embedding_size: output embedding dimension for each word after
embedding layer,
        lstm_size: Number of lstm units in encoder,
        input_length: Length of the input sentence,
        batch_size
    '''


    vocab_size=10
    embedding_size=20
    lstm_size=32
    input_length=10
    batch_size=16
    encoder=Encoder(vocab_size,embedding_size,lstm_size,input_length)
    input_sequence=tf.random.uniform(shape=[batch_size,input_length],m
axval=vocab_size,minval=0,dtype=tf.int32)
    initial_state=encoder.initialize_states(batch_size)
    encoder_output,state_h,state_c=encoder(input_sequence,initial_stat
e)

    assert(encoder_output.shape==(batch_size,input_length,lstm_size) a
nd state_h.shape==(batch_size,lstm_size) and state_c.shape==(batch_siz
e,lstm_size))
    return True
print(grader_check_encoder())
```

```
True
```

**Attention**

In [78]:
```
a = tf.ones(shape=(1,20,1))
tf.squeeze(a, axis=-1).shape
```

Out[78]:
```
TensorShape([1, 20])
```

In [79]:
```
class Attention(tf.keras.layers.Layer):
  '''
    Class the calculates score based on the scoring_function using Bah
danu attention mechanism.
  '''
  def __init__(self,scoring_function, att_units):
    super().__init__()
    self.scoring_function = scoring_function
    self.att_units = att_units


    # Please go through the reference notebook and research paper to c
omplete the scoring functions
    if self.scoring_function=='dot':
      # Intialize variables needed for Dot score function here
      pass
    if scoring_function == 'general':
      # Intialize variables needed for General score function here
      pass
    elif scoring_function == 'concat':
      # Intialize variables needed for Concat score function here
      pass


  def build(self, input_shape):
    self.softmax = Softmax()

  def call(self,decoder_hidden_state,encoder_output):
    '''
      Attention mechanism takes two inputs current step -- decoder_hid
den_state and all the encoder_outputs.
      * Based on the scoring function we will find the score or simila
rity between decoder_hidden_state and encoder_output.
        Multiply the score function with your encoder_outputs to get t
he context vector.
        Function returns context vector and attention weights(softmax
 - scores)
    '''

    if self.scoring_function == 'dot':

        decoder_hidden_state = Reshape(target_shape=(decoder_hidden_st
ate.shape[1],1))(decoder_hidden_state)
        dot_layer = Dot(axes=(2,1))([encoder_output,decoder_hidden_sta
te])
        dot_layer = tf.squeeze(dot_layer)

        softmax_layer = self.softmax(dot_layer)

        # weighted context_vector
        weights_squeezed = softmax_layer # bat,timesteps
        encoder_ops = encoder_output # bat,timesteps,enc_units

        # multiply weights and add to get context_vector
        weights = tf.expand_dims(weights_squeezed, axis=-1)
        multiply_weights = tf.multiply(encoder_ops, weights)
```

```
        context_vector = tf.math.reduce_sum(multiply_weights, axis=-2)

        return context_vector, weights

    elif self.scoring_function == 'general':
      decoder_hidden_state = Dense(encoder_output.shape[-1])(decoder_h
idden_state)
      # print("1: ",decoder_hidden_state.shape)
      decoder_hidden_state = Reshape(target_shape=(decoder_hidden_stat
e.shape[1],1))(decoder_hidden_state)
      # print("2: ",decoder_hidden_state.shape)

      dot_layer = Dot(axes=(2,1))([encoder_output,decoder_hidden_state
])
      # print("3 : ",dot_layer.shape)
      squeezed_dot_layer = tf.squeeze(dot_layer, axis=-1)
      # print("4a : ",squeezed_dot_layer.shape)
      softmax_layer_weights = Softmax()(squeezed_dot_layer) # bat, tim
esteps
      # print("4b : ",softmax_layer_weights.shape)

      # encoder_ops = encoder_output.numpy() # bat,timesteps,enc_units
      # multiply weights and add to get context_vector
      # print(softmax_layer_weights.shape)
      weights = Reshape(target_shape=(softmax_layer_weights.shape[1],1
))(softmax_layer_weights) # bat,timesteps, 1
      multiply_weights = tf.multiply(encoder_output, weights)
      context_vector = tf.math.reduce_sum(multiply_weights, axis=-2)

      return context_vector, weights

    elif self.scoring_function == 'concat':
      k = 64
      encoder_timesteps = encoder_output.shape[1]

      dense_d = Dense(k)(decoder_hidden_state) #bat, K
      dense_e = Dense(k)(encoder_output) # bat,time_steps, K

      # concat and tanH
      expand = tf.expand_dims(dense_d,axis=1)
      tiled = tf.tile(expand, multiples=[1,encoder_timesteps,1])
      concat = tf.concat([dense_e, tiled], axis=-1) #bat,time_steps, 2
*K

      tanh = tf.keras.activations.tanh(concat) #bat,time_steps, 2*K
      w = Dense(1)(tanh) #bat,time_steps, 1

      # multiply weights and add to get context_vector
      weights = Reshape(target_shape=(w.shape[1],1))(w) # bat,timestep
s, 1
      multiply_weights = tf.multiply(encoder_output, weights)
      context_vector = tf.math.reduce_sum(multiply_weights, axis=-2)

      return context_vector, weights
```

**Grader function - 2**

In [80]:
```
def grader_check_attention(scoring_fun):

    '''
        att_units: Used in matrix multiplications for scoring functions,
        input_length: Length of the input sentence,
        batch_size
    '''

    input_length=10
    batch_size=16
    att_units=32

    state_h=tf.random.uniform(shape=[batch_size,att_units])
    encoder_output=tf.random.uniform(shape=[batch_size,input_length,att_units])
    attention=Attention(scoring_fun,att_units)
    context_vector,attention_weights=attention(state_h,encoder_output)
    assert(context_vector.shape==(batch_size,att_units) and attention_weights.shape==(batch_size,input_length,1))
    return True

print(grader_check_attention('dot'))
print(grader_check_attention('general'))
print(grader_check_attention('concat'))
```

```
True
True
True
```

**OneStepDecoder**

In [82]:
```
class One_Step_Decoder(tf.keras.Model):
  def __init__(self,tar_vocab_size, embedding_dim, input_length, dec_u
nits ,score_fun ,att_units):
    super().__init__()
    self.tar_vocab_size = tar_vocab_size
    self.embedding_dim = embedding_dim
    self.input_length = input_length
    self.dec_units = dec_units
    self.score_fun = score_fun
    self.att_units = att_units

    # Initialize decoder embedding layer, LSTM and any other objects n
eeded
  def build(self,input_shape):
    self.embedding = Embedding(input_dim=self.tar_vocab_size, output_d
im=self.embedding_dim,
                               input_length=self.input_length, mask_zer
o=True,name="OHE_Embedding")
    self.lstm = LSTM(units=self.dec_units,return_state = True, return_
sequences=True, name="OHE_LSTM_Layer")
    self.attention = Attention(self.score_fun,self.att_units)
    self.dense_layer = Dense(self.tar_vocab_size)


  def call(self,input_to_decoder, encoder_output, state_h,state_c):
    '''
        One step decoder mechanisim step by step:
      A. Pass the input_to_decoder to the embedding layer and then get
the output(batch_size,1,embedding_dim)
      B. Using the encoder_output and decoder hidden state, compute th
e context vector.
      C. Concat the context vector with the step A output
      D. Pass the Step-C output to LSTM/GRU and get the decoder output
and states(hidden and cell state)
      E. Pass the decoder output to dense layer(vocab size) and store
 the result into output.
      F. Return the states from step D, output from Step E, attention
 weights from Step -B
    '''

    embed = self.embedding(input_to_decoder) #32,1,12
    context_vector, weights = self.attention(state_h, encoder_output)
#32,16
    context_vector_exp = tf.expand_dims(context_vector, axis=1) #32,16
-> 32,1,16
    concat = tf.concat([embed,context_vector_exp],axis=-1) #32,1,28

    decoder_output, decoder_h, decoder_c = self.lstm(concat, initial_s
tate=[state_h, state_c]) #(32, 1, 16) (32, 16)
    dense = self.dense_layer(decoder_output)  #(32, 1, 13)

    output = tf.squeeze(dense)  #(32, 1, 13) -> (32,13)
    # return output, state_h, state_c, weights, context_vector
    return output, decoder_h, decoder_c, weights, context_vector
```

**Grader function - 3**

In [83]:
```
def grader_onestepdecoder(score_fun):
    '''
        tar_vocab_size: Unique words of the target language,
        embedding_dim: output embedding dimension for each word after
 embedding layer,
        dec_units: Number of lstm units in decoder,
        att_units: Used in matrix multiplications for scoring function
s in attention class,
        input_length: Length of the target sentence,
        batch_size
    '''
    tar_vocab_size=13
    embedding_dim=12
    input_length=10
    dec_units=16
    att_units=16
    batch_size=32
    onestepdecoder=One_Step_Decoder(tar_vocab_size, embedding_dim, inp
ut_length, dec_units ,score_fun ,att_units)
    input_to_decoder=tf.random.uniform(shape=(batch_size,1),maxval=10,
minval=0,dtype=tf.int32)
    encoder_output=tf.random.uniform(shape=[batch_size,input_length,de
c_units])
    state_h=tf.random.uniform(shape=[batch_size,dec_units])
    state_c=tf.random.uniform(shape=[batch_size,dec_units])
    output,state_h,state_c,attention_weights,context_vector=onestepdec
oder(input_to_decoder,encoder_output,state_h,state_c)
    assert(output.shape==(batch_size,tar_vocab_size))
    assert(state_h.shape==(batch_size,dec_units))
    assert(state_c.shape==(batch_size,dec_units))
    assert(attention_weights.shape==(batch_size,input_length,1))
    assert(context_vector.shape==(batch_size,dec_units))
    return True

print(grader_onestepdecoder('dot'))
print(grader_onestepdecoder('general'))
print(grader_onestepdecoder('concat'))
```

```
True
True
True
```

**Decoder**

In [84]:
```
class Decoder(tf.keras.Model):
    def __init__(self,out_vocab_size, embedding_dim, input_length, dec
_units ,score_fun ,att_units):
      #Intialize necessary variables and create an object from the cla
ss onestepdecoder
      super().__init__()
      self.out_vocab_size = out_vocab_size
      self.embedding_dim = embedding_dim
      self.input_length = input_length
      self.dec_units = dec_units
      self.score_fun = score_fun
      self.att_units = att_units

    def build(self, input_shape):
      self.onestepdecoder = One_Step_Decoder(self.out_vocab_size, self
.embedding_dim,
                                             self.input_length, self.d
ec_units, self.score_fun, self.att_units)


    def call(self, input_to_decoder,encoder_output,decoder_hidden_stat
e,decoder_cell_state ):

      #Initialize an empty Tensor array, that will store the outputs a
t each and every time step
      tf_output = tf.TensorArray(dtype=tf.float32, size=self.input_len
gth, name="tf_output_array")

      #Iterate till the length of the decoder input
      for timestep in range(self.input_length):
        # Call onestepdecoder for each token in decoder_input
        output,state_h,state_c,attention_weights,context_vector = self
.onestepdecoder(input_to_decoder[:,timestep:timestep+1],

encoder_output,decoder_hidden_state,decoder_cell_state)
        tf_output = tf_output.write(timestep, output)
        decoder_hidden_state = state_h
        decoder_cell_state = state_c

      # Return the tensor array
      tf_output = tf.transpose(tf_output.stack(),[1,0,2])
      return tf_output
```

**Grader function - 4**

In [85]:
```
def grader_decoder(score_fun):

    '''
        out_vocab_size: Unique words of the target language,
        embedding_dim: output embedding dimension for each word after
 embedding layer,
        dec_units: Number of lstm units in decoder,
        att_units: Used in matrix multiplications for scoring function
s in attention class,
        input_length: Length of the target sentence,
        batch_size


    '''


    out_vocab_size=13
    embedding_dim=12
    input_length=11
    dec_units=16
    att_units=16
    batch_size=32

    target_sentences=tf.random.uniform(shape=(batch_size,input_length
),maxval=10,minval=0,dtype=tf.int32)
    encoder_output=tf.random.uniform(shape=[batch_size,input_length,de
c_units])
    state_h=tf.random.uniform(shape=[batch_size,dec_units])
    state_c=tf.random.uniform(shape=[batch_size,dec_units])

    decoder=Decoder(out_vocab_size, embedding_dim, input_length, dec_u
nits ,score_fun ,att_units)
    output=decoder(target_sentences,encoder_output, state_h, state_c)
    assert(output.shape==(batch_size,input_length,out_vocab_size))
    return True
print(grader_decoder('dot'))
print(grader_decoder('general'))
print(grader_decoder('concat'))
```

```
True
True
True
```

**Encoder Decoder model**

In [86]:
```
class encoder_decoder(tf.keras.Model):
  def __init__(self,inp_vocab_size,out_vocab_size, embedding_size,lstm
_size, input_length, scoring_fun, att_units,batch_size):
    #Intialize objects from encoder decoder
    super().__init__()
    self.inp_vocab_size = inp_vocab_size
    self.out_vocab_size = out_vocab_size
    self.embedding_size = embedding_size
    self.lstm_size = lstm_size
    self.input_length = input_length
    self.scoring_fun = scoring_fun
    self.att_units = att_units
    self.batch_size = batch_size

  def build(self, input_shape):
    self.encoder = Encoder(self.inp_vocab_size,self.embedding_size,sel
f.lstm_size,self.input_length)
    self.decoder=Decoder(self.out_vocab_size, self.embedding_size, sel
f.input_length, self.lstm_size ,self.scoring_fun ,self.att_units)


  def call(self,data):
    #Intialize encoder states, Pass the encoder_sequence to the embedd
ing layer
    # Decoder initial states are encoder final states, Initialize it a
ccordingly
    # Pass the decoder sequence,encoder_output,decoder states to Decod
er
    # return the decoder output
    input_sequences, target_sentences = data[0], data[1]
    enc_initial_state = self.encoder.initialize_states(self.batch_size
)
    encoder_output, state_h, state_c = self.encoder(input_sequences,en
c_initial_state)

    # teacher training
    decoder_output = self.decoder(target_sentences,encoder_output, sta
te_h, state_c)
    return decoder_output
```

**Custom loss function**

In [87]:
```
# Refer https://www.tensorflow.org/tutorials/text/nmt_with_attention#define_the_optimizer_and_the_loss_function

def custom_lossfunction(targets,logits):
  loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')
  mask = tf.math.logical_not(tf.math.equal(targets, 0))
  loss_ = loss_object(targets, logits)
  mask = tf.cast(mask, dtype=loss_.dtype)
  loss_ *= mask
  return tf.reduce_mean(loss_)


optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)
```

# Training Model with Dot

In [88]:
```
train_dataset = Dataset(train, tknizer_ita, tknizer_eng, 20)
test_dataset  = Dataset(validation, tknizer_ita, tknizer_eng, 20)
train_dataloader = Dataloder(train_dataset, batch_size=512)
test_dataloader = Dataloder(test_dataset, batch_size=512)
print(train_dataloader[0][0][0].shape, train_dataloader[0][0][1].shape
, train_dataloader[0][1].shape)

file_path = "attention_general.h5"
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=file_path,
    save_weights_only=True,
    monitor='loss',
    mode='auto',
    save_best_only=False,
    save_freq='epoch')

class SaveDrive(tf.keras.callbacks.Callback):
   def on_epoch_end(self, epoch, logs=None):
        !cp "attention_40_Plus.h5" "drive/MyDrive/Datasets"
```

```
(512, 20) (512, 20) (512, 20)
```

In [89]:
```
model_dot = encoder_decoder(inp_vocab_size=vocab_size_ita+1,
                        out_vocab_size=vocab_size_eng+1,
                        embedding_size=20,
                        lstm_size=32,
                        input_length=20,
                        scoring_fun='dot',
                        att_units=32,
                        batch_size=512)

model_dot.compile(optimizer=optimizer,
                  loss=custom_lossfunction)
```

In [92]:
```
# Loading the model trained previously for 80 epochs
model_dot.load_weights("/content/attention_latest_Jul_10_80_Epochs.h5"
)

class SaveDrive(tf.keras.callbacks.Callback):
   def on_epoch_end(self, epoch, logs=None):
        !cp "attention_40_Plus.h5" "drive/MyDrive/Datasets"
```

In [93]:
```
tf.config.run_functions_eagerly(True)
hist_dot = model_dot.fit_generator(train_dataloader,
                                   epochs=30,
                                   validation_data=test_dataloader,
                                   callbacks = [model_checkpoint_callb
ack,SaveDrive()])
```

# **Inference**

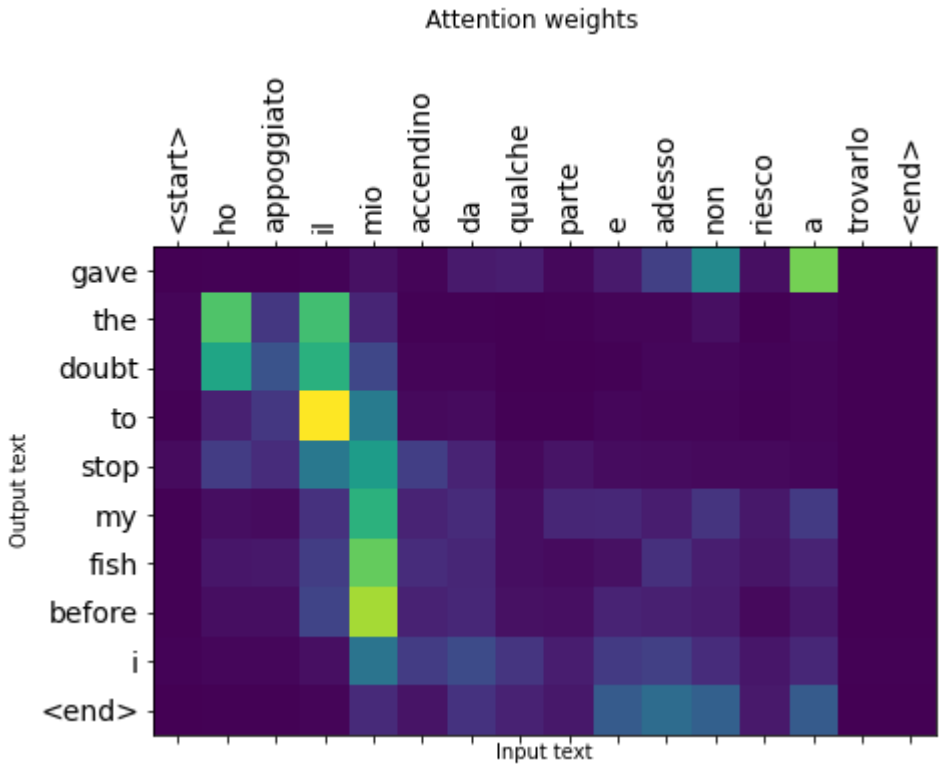
**Plot attention weights**

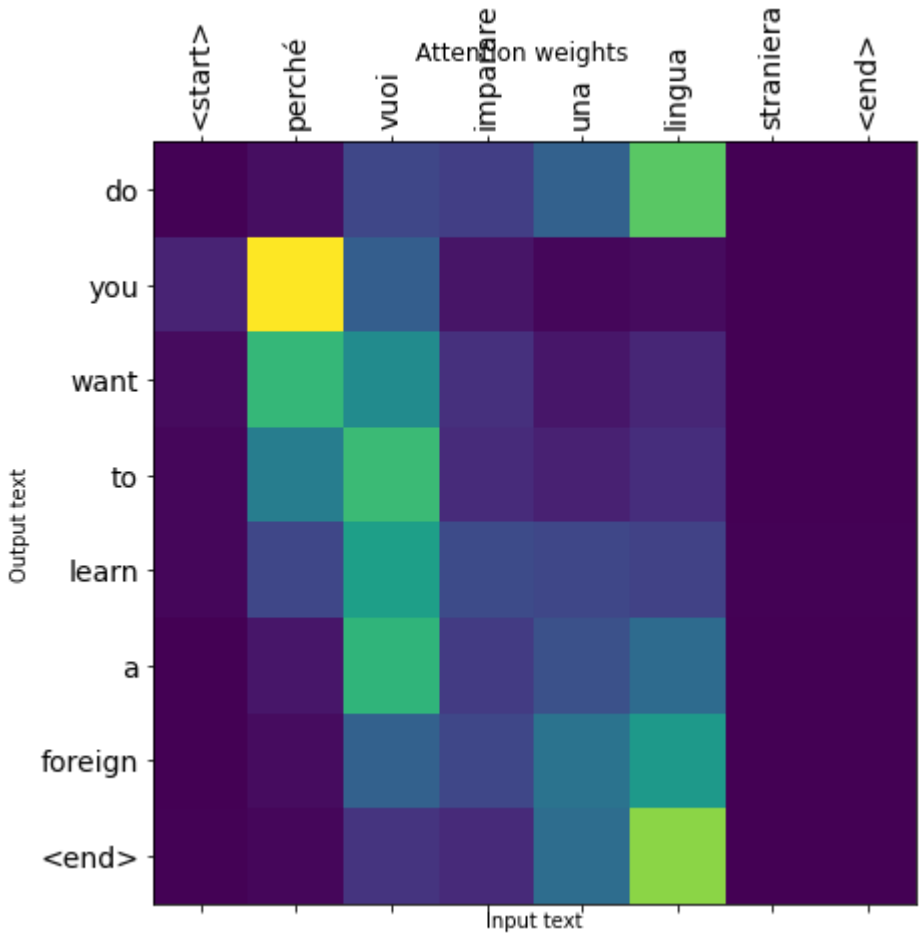
In [98]:
```
import matplotlib.ticker as ticker

def plot_attention(attention, sentence, predicted_sentence):
  sentence = ["<start>"]+sentence.split()+["<end>"]
  predicted_sentence = predicted_sentence.split()[:-1] + ['<end>']
  fig = plt.figure(figsize=(7, 7))
  ax = fig.add_subplot(1, 1, 1)

  attention = attention[:len(predicted_sentence), :len(sentence)]
  ax.matshow(attention, cmap='viridis', vmin=0.0)
  fontdict = {'fontsize': 14}

  ax.set_xticklabels(['']  + sentence, fontdict=fontdict, rotation=90)
  ax.set_yticklabels(['']  + predicted_sentence, fontdict=fontdict)
  ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
  ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

  ax.set_xlabel('Input text')
  ax.set_ylabel('Output text')
  plt.suptitle('Attention weights')
```

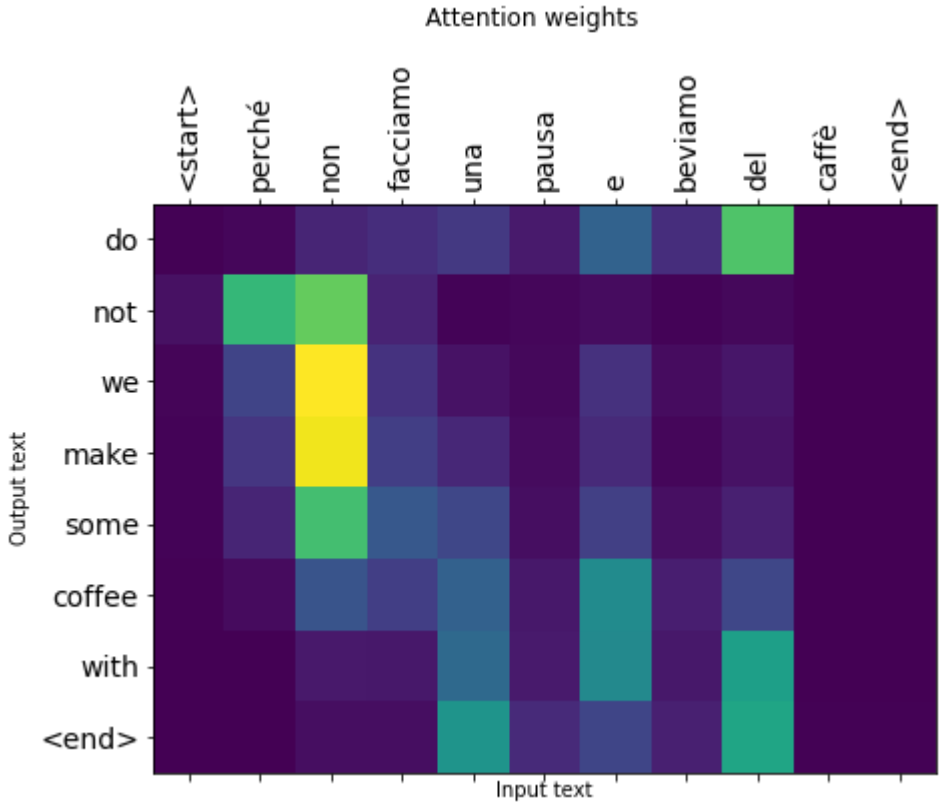
**Predict the sentence translation**

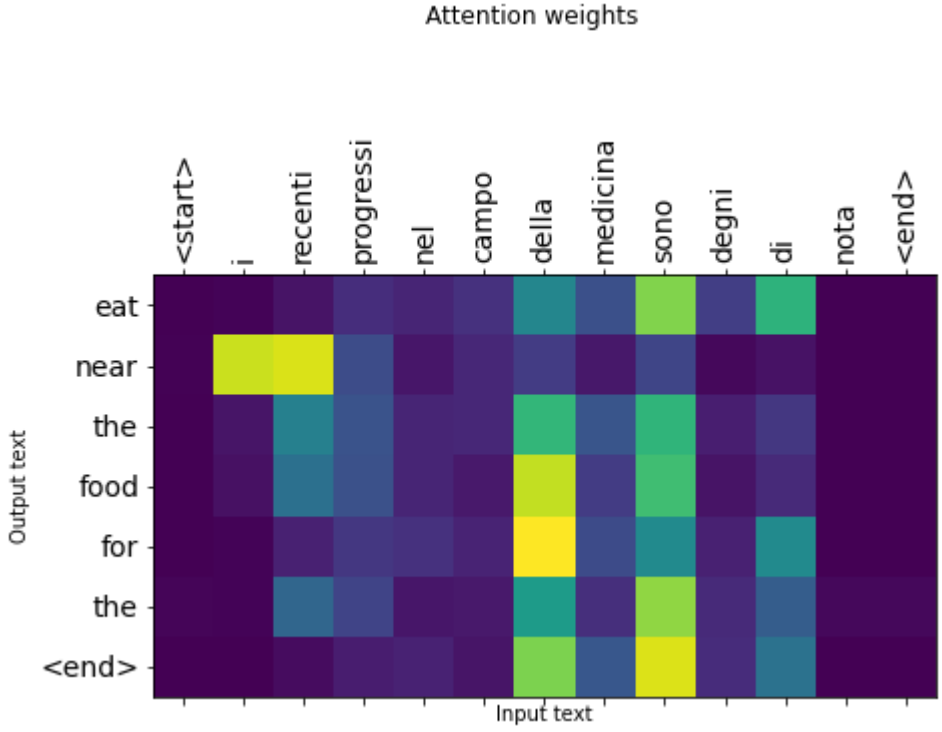
In [99]:
```
def predict(input_sentence):
  '''
  A. Given input sentence, convert the sentence into integers using to
kenizer used earlier
  B. Pass the input_sequence to encoder. we get encoder_outputs, last
 time step hidden and cell state
  C. Initialize index of <start> as input to decoder. and encoder fina
l states as input_states to onestepdecoder.
  D. till we reach max_length of decoder or till the model predicted w
ord <end>:
         predictions, input_states, attention_weights = model.layers
[1].onestepdecoder(input_to_decoder, encoder_output, input_states)
         Save the attention weights
         And get the word using the tokenizer(word index) and then sto
re it in a string.
  E. Call plot_attention(#params)
  F. Return the predicted sentence

  '''
  input_length = 20
  lstm_units = 32
  batch_size = 1

  encoder_seq = tknizer_ita.texts_to_sequences([input_sentence]) # nee
d to pass list of values
  encoder_seq = pad_sequences(encoder_seq, maxlen=input_length, dtype=
'int32', padding='post')
  encoder_output, state_h, state_c = model_dot.layers[0](encoder_seq,
model_dot.layers[0].initialize_states(batch_size))

  cur_vec = np.ones((1,1))
  cur_vec[0,0] = tknizer_eng.word_index['<start>']
  result_sentence = ""
  weights_arr = []

  for i in range(20):
    predictions,dec_state_h,dec_state_c,attention_weights,context_vect
or = model_dot.layers[1].onestepdecoder(cur_vec, encoder_output, state
_h, state_c)
    cur_vec = np.reshape(np.argmax(predictions),(1,1))
    state_h = dec_state_h
    state_c = dec_state_c
    index= np.argmax(predictions)
    weights_arr.append(attention_weights.numpy())

    if i==0:
      # print(index)
      continue

    if tknizer_eng.index_word[index] == "<end>":
      return result_sentence,np.array(weights_arr)
    result_sentence += tknizer_eng.index_word[index] + " "

  # print(weights_arr)
  return result_sentence,np.array(weights_arr)
```

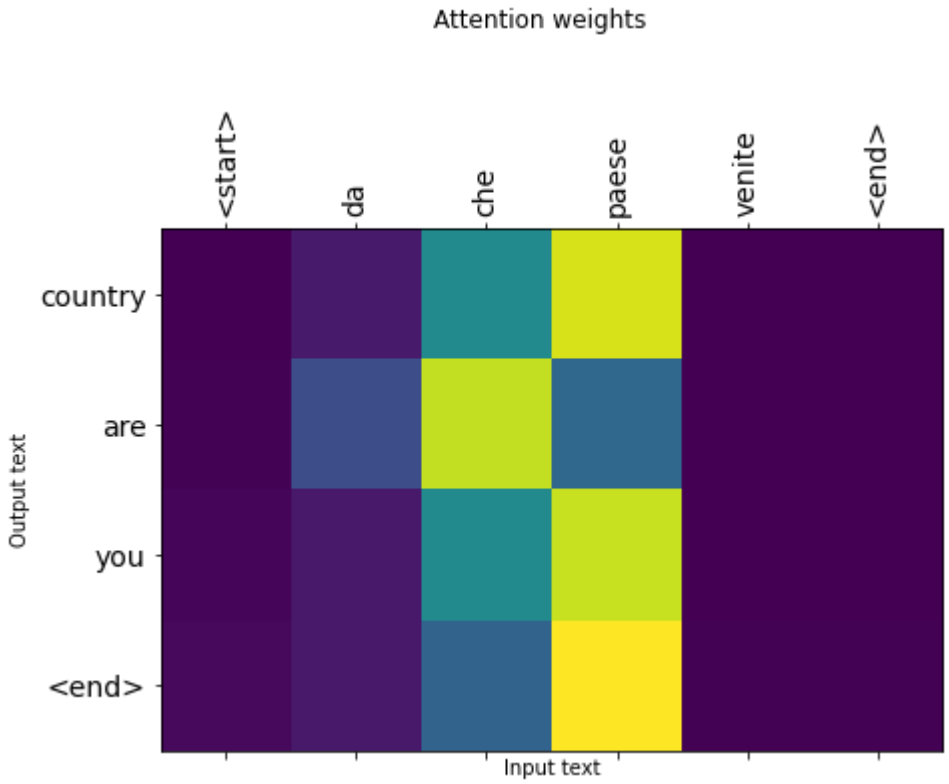
In [105]:
```
for index in np.random.randint(1,1000,5):
  ita_sentence = test_df.iloc[index]["italian"]
  res,att = predict(ita_sentence)
  att = np.sum(att, axis=-1)
  plot_attention(att, ita_sentence, res)
```


Attention weights
<start>
ho
appoggiato
il
mio
accendino
da
qualche
parte
e
adesso
non
riesco
a
trovarlo
<end>
gave
the
doubt
to
stop
my
fish
before
i
<end>
Output text
Input text


Attention weights
<start>
perché
vuoi
imparare
una
lingua
straniera
<end>
do
you
want
to
learn
a
foreign
<end>
Output text
Input text


Attention weights
<start>
perché
non
facciamo
una
pausa
e
beviamo
del
caffè
<end>
do
not
we
make
some
coffee
with
<end>
Output text
Input text


Attention weights
<start>
i
recenti
progressi
nel
campo
della
medicina
sono
degni
di
nota
<end>
eat
near
the
food
for
the
<end>
Output text
Input text




**Calculate BLEU score**

In [106]:
```
from nltk.translate.bleu_score import SmoothingFunction
smoothie = SmoothingFunction().method1

def aveg_bleu_scores(test_data,n):
  sample_list = random.sample(range(len(test_data)),n)
  average_bleu = 0
  pred_data = []
  individual_bleu = []

  for i in sample_list:
    test_sentence, true_sentence = test_data.iloc[i,[0,1]]
    pred_sentence, att = predict(test_sentence)
    bleu_score = bleu.sentence_bleu([true_sentence.split()],pred_sente
nce.split())
    average_bleu += bleu_score

    # pred_data.append((true_sentence, pred_sentence))
    individual_bleu.append(bleu_score)

  return average_bleu/n, individual_bleu
```

In [115]:
```
#Create an object of your custom model.
#Compile and train your model on dot scoring function.
# Visualize few sentences randomly in Test data
# Predict on 1000 random sentences on test data and calculate the aver
age BLEU score of these sentences.
# https://www.nltk.org/_modules/nltk/translate/bleu_score.html
import nltk.translate.bleu_score as bleu
import random

test_df = validation.copy()
test_df = test_df[["italian","english_out"]]
test_df["english_out"] = test_df["english_out"].apply(lambda x: x.spli
t("<end>")[0])

avg_score, individual_score = aveg_bleu_scores(test_df,1000)
print("Average BLEU score: ",avg_score)
```

```
Average BLEU score:  0.6775453021317511
```

# Training Model with General

In [116]:
```
train_dataset = Dataset(train, tknizer_ita, tknizer_eng, 20)
test_dataset  = Dataset(validation, tknizer_ita, tknizer_eng, 20)
train_dataloader = Dataloder(train_dataset, batch_size=512)
test_dataloader = Dataloder(test_dataset, batch_size=512)
print(train_dataloader[0][0][0].shape, train_dataloader[0][0][1].shape
, train_dataloader[0][1].shape)
```

```
(512, 20) (512, 20) (512, 20)
```

In [117]:
```
# file_path = "test/Attention_2_{epoch:04d}.h5"
file_path = "attention_general.h5"
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=file_path,
    save_weights_only=True,
    monitor='loss',
    mode='auto',
    save_best_only=False,
    save_freq='epoch')

model_general = encoder_decoder(inp_vocab_size=vocab_size_ita+1,
                        out_vocab_size=vocab_size_eng+1,
                        embedding_size=20,
                        lstm_size=32,
                        input_length=20,
                        scoring_fun='general',
                        att_units=32,
                        batch_size=512)

model_general.compile(optimizer=optimizer,
                  loss=custom_lossfunction)
```

In [122]:
```
# Load model weights trained for 50 epochs
model_general.load_weights("attention_general_fast_50.h5")

class SaveDrive(tf.keras.callbacks.Callback):
   def on_epoch_end(self, epoch, logs=None):
        !cp "attention_general.h5" "drive/MyDrive/Datasets"
```

In [123]:
```
tf.config.run_functions_eagerly(True)
hist_general = model_general.fit_generator(train_dataloader,
                                           epochs=30,
                                           validation_data=test_dataloader,
                                           callbacks = [model_checkpoint_callb
ack, SaveDrive()])
```

# **Inference**

**Plot attention weights**

**Predict the sentence translation**

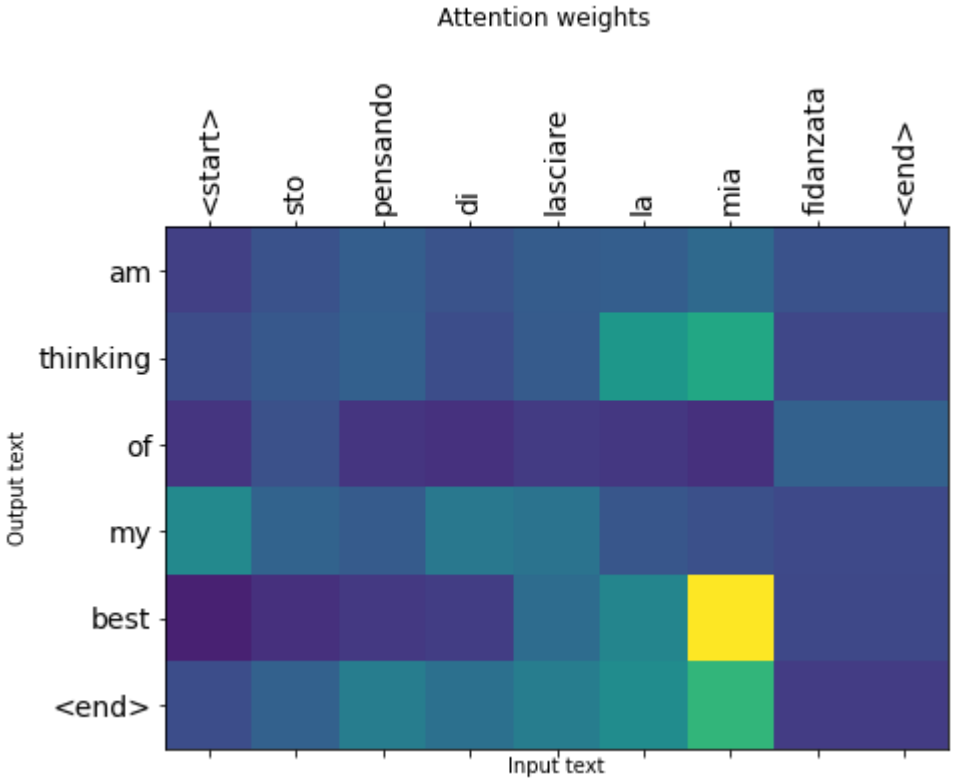
In [124]:
```
model_general.layers
```

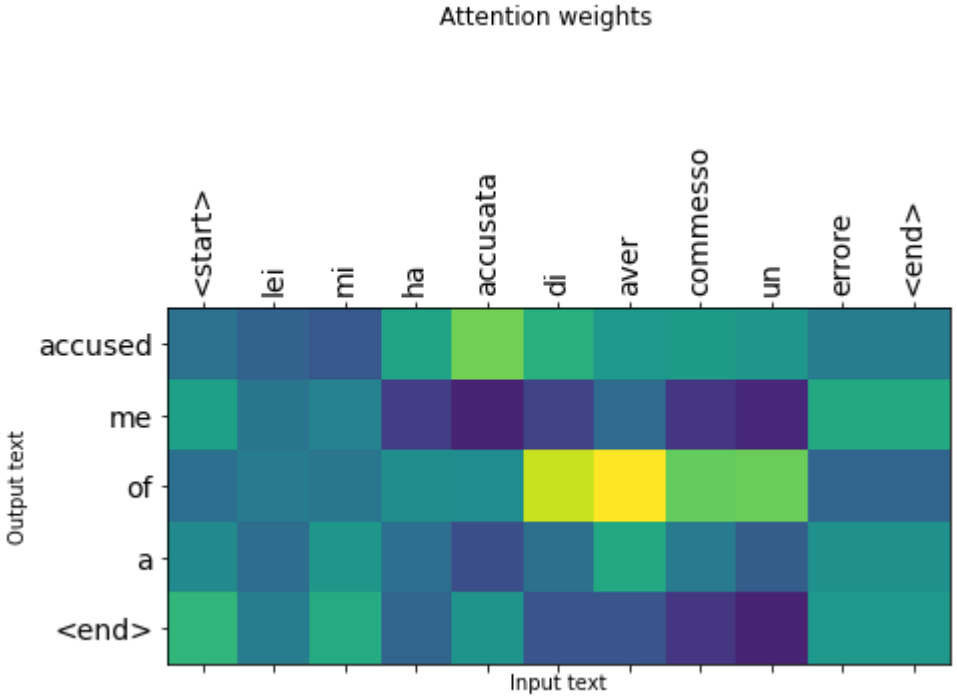
Out[124]:
```
[<__main__.Encoder at 0x7fedc73d8910>, <__main__.Decoder at 0x7fedc5d3
d250>]
```

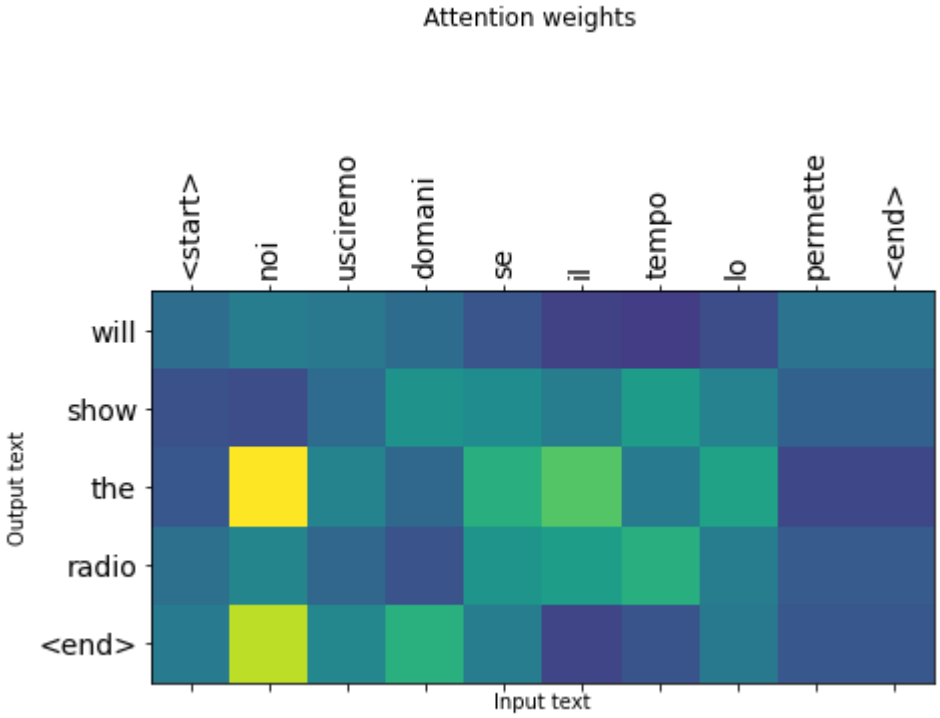
In [126]:
```
def predict_general(input_sentence):
  '''
  A. Given input sentence, convert the sentence into integers using to
kenizer used earlier
  B. Pass the input_sequence to encoder. we get encoder_outputs, last
 time step hidden and cell state
  C. Initialize index of <start> as input to decoder. and encoder fina
l states as input_states to onestepdecoder.
  D. till we reach max_length of decoder or till the model predicted w
ord <end>:
         predictions, input_states, attention_weights = model.layers
[1].onestepdecoder(input_to_decoder, encoder_output, input_states)
         Save the attention weights
         And get the word using the tokenizer(word index) and then sto
re it in a string.
  E. Call plot_attention(#params)
  F. Return the predicted sentence

  '''
  input_length = 20
  lstm_units = 32
  batch_size = 1

  encoder_seq = tknizer_ita.texts_to_sequences([input_sentence]) # nee
d to pass list of values
  encoder_seq = pad_sequences(encoder_seq, maxlen=input_length, dtype=
'int32', padding='post')
  encoder_output, state_h, state_c = model_general.layers[0](encoder_s
eq, model_general.layers[0].initialize_states(batch_size))

  cur_vec = np.ones((1,1))
  cur_vec[0,0] = tknizer_eng.word_index['<start>']
  result_sentence = ""
  weights_arr = []

  for i in range(20):
    predictions,dec_state_h,dec_state_c,attention_weights,context_vect
or = model_general.layers[1].onestepdecoder(cur_vec, encoder_output, s
tate_h, state_c)
    cur_vec = np.reshape(np.argmax(predictions),(1,1))
    state_h = dec_state_h
    state_c = dec_state_c
    index= np.argmax(predictions)
    weights_arr.append(attention_weights.numpy()[0])

    if i==0:
      # print(index)
      continue

    if tknizer_eng.index_word[index] == "<end>":
      return result_sentence,np.array(weights_arr)
    result_sentence += tknizer_eng.index_word[index] + " "

  # print(weights_arr)
  return result_sentence,np.array(weights_arr)
```

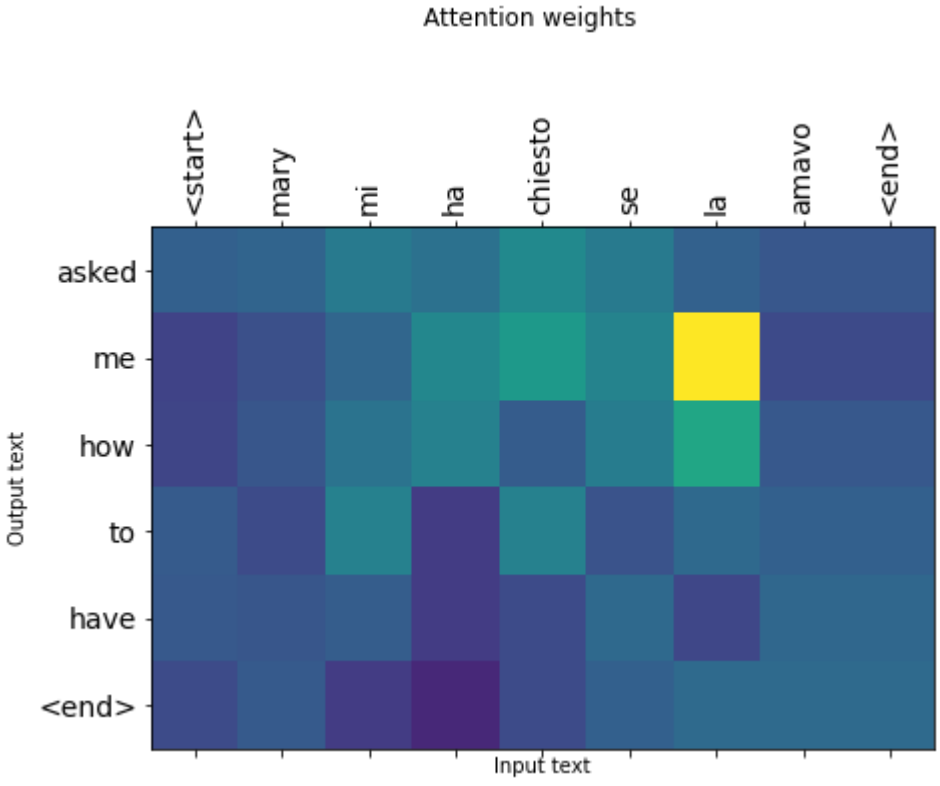
In [127]:
```
import matplotlib.ticker as ticker

def plot_attention_general(sentence):
  predicted_sentence, attention = predict_general(sentence)
  attention = np.sum(attention, axis=-1)
  sentence = ["<start>"]+sentence.split()+["<end>"]
  predicted_sentence = predicted_sentence.split()[:-1] + ['<end>']
  fig = plt.figure(figsize=(7, 7))
  ax = fig.add_subplot(1, 1, 1)

  attention = attention[:len(predicted_sentence), :len(sentence)]
  ax.matshow(attention, cmap='viridis', vmin=0.0)
  fontdict = {'fontsize': 14}

  ax.set_xticklabels(['']  + sentence, fontdict=fontdict, rotation=90)
  ax.set_yticklabels(['']  + predicted_sentence, fontdict=fontdict)
  ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
  ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

  ax.set_xlabel('Input text')
  ax.set_ylabel('Output text')
  plt.suptitle('Attention weights')
```

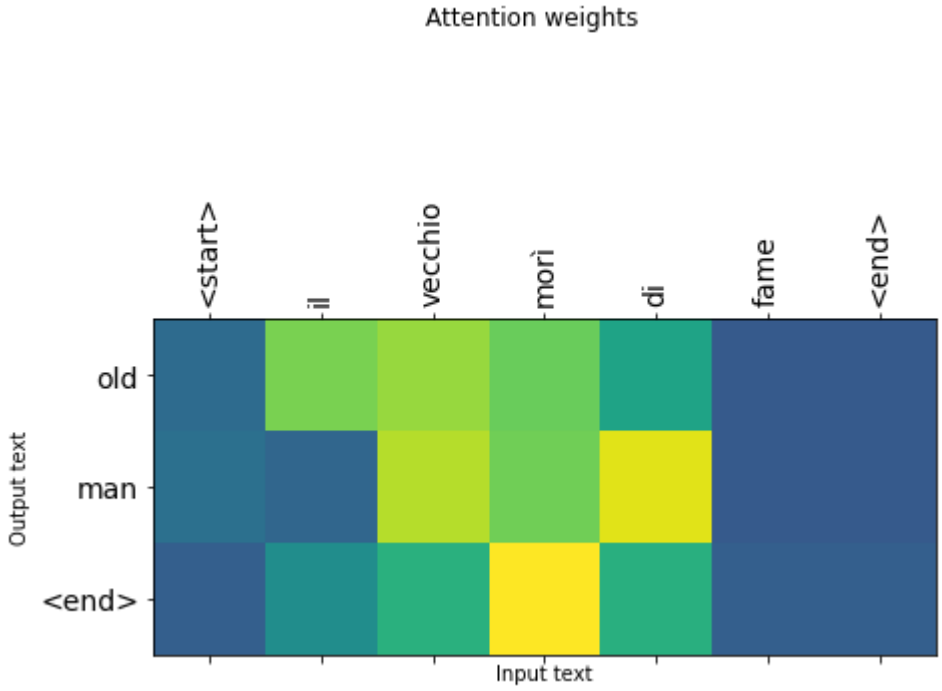
In [129]:
```
for index in np.random.randint(1,1000,5):
  ita_sentence = test_df.iloc[index]["italian"]
  plot_attention_general(ita_sentence)
```


Attention weights
<start>
sto
pensando
di
lasciare
la
mia
fidanzata
<end>
am
thinking
of
my
best
<end>
Output text
Input text


Attention weights
<start>
lei
mi
ha
accusata
di
aver
commesso
un
errore
<end>
accused
me
of
a
<end>
Output text
Input text


Attention weights
<start>
noi
usciremo
domani
se
il
tempo
lo
permette
<end>
will
show
the
radio
<end>
Output text
Input text


Attention weights
<start>
mary
mi
ha
chiesto
se
la
amavo
<end>
asked
me
how
to
have
<end>
Output text
Input text




**Calculate BLEU score**

In [130]:
```
from nltk.translate.bleu_score import SmoothingFunction
smoothie = SmoothingFunction().method1

def aveg_bleu_scores_general(test_data,n):
  sample_list = random.sample(range(len(test_data)),n)
  average_bleu = 0
  pred_data = []
  individual_bleu = []

  for i in sample_list:
    test_sentence, true_sentence = test_data.iloc[i,[0,1]]
    pred_sentence, att = predict_general(test_sentence)
    bleu_score = bleu.sentence_bleu([true_sentence.split()],pred_sente
nce.split())
    average_bleu += bleu_score

    # pred_data.append((true_sentence, pred_sentence))
    individual_bleu.append(bleu_score)

  return average_bleu/n, individual_bleu
```

In [134]:
```
import nltk.translate.bleu_score as bleu
import random

test_df = validation.copy()
test_df = test_df[["italian","english_out"]]
test_df["english_out"] = test_df["english_out"].apply(lambda x: x.spli
t("<end>")[0])

avg_score, individual_score = aveg_bleu_scores_general(test_df,1000)
print("Average BLEU Score: ",avg_score)
```

```
Average BLEU Score:  0.5599070673098394
```

# Training Model with Concat

In [136]:
```
train_dataset = Dataset(train, tknizer_ita, tknizer_eng, 20)
test_dataset  = Dataset(validation, tknizer_ita, tknizer_eng, 20)
train_dataloader = Dataloder(train_dataset, batch_size=512)
test_dataloader = Dataloder(test_dataset, batch_size=512)
print(train_dataloader[0][0][0].shape, train_dataloader[0][0][1].shape
, train_dataloader[0][1].shape)
```

```
(512, 20) (512, 20) (512, 20)
```

In [137]:
```
# file_path = "test/Attention_2_{epoch:04d}.h5"
file_path = "concat_general.h5"
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=file_path,
    save_weights_only=True,
    monitor='loss',
    mode='auto',
    save_best_only=False,
    save_freq='epoch')

model_concat = encoder_decoder(inp_vocab_size=vocab_size_ita+1,
                        out_vocab_size=vocab_size_eng+1,
                        embedding_size=20,
                        lstm_size=32,
                        input_length=20,
                        scoring_fun='concat',
                        att_units=32,
                        batch_size=512)

model_concat.compile(optimizer=optimizer,
                  loss=custom_lossfunction)
```

In [145]:
```
# Model weights trained for 30+30 = 60 epochs
model_concat.load_weights("/content/attention_concat_60.h5")

class SaveDrive(tf.keras.callbacks.Callback):
   def on_epoch_end(self, epoch, logs=None):
        !cp "concat_general.h5" "drive/MyDrive/Datasets"
```

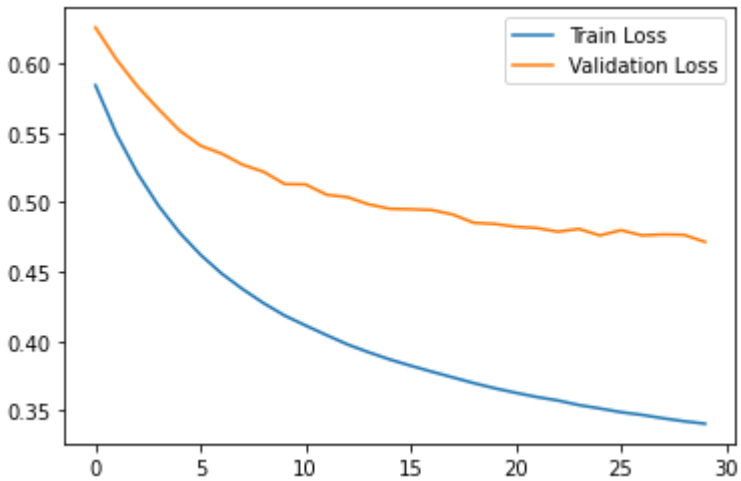
In [ ]:
```
tf.config.run_functions_eagerly(True)
hist_gconcat = model_concat.fit_generator(train_dataloader,
                                          epochs=30,
                                          validation_data=test_dataloader,
                                          callbacks = [model_checkpoint_callb
ack, SaveDrive()])
```

```
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/
training.py:1940: UserWarning: `Model.fit_generator` is deprecated and
will be removed in a future version. Please use `Model.fit`, which sup
ports generators.
  warnings.warn('`Model.fit_generator` is deprecated and '
/usr/local/lib/python3.7/dist-packages/tensorflow/python/data/ops/data
set_ops.py:3704: UserWarning: Even though the `tf.config.experimental_
run_functions_eagerly` option is set, this option does not apply to t
f.data functions. To force eager execution of tf.data functions, pleas
e use `tf.data.experimental.enable.debug_mode()`.
  "Even though the `tf.config.experimental_run_functions_eagerly` "
```

```
Epoch 1/30
546/546 [==============================] - 364s 667ms/step - loss: 0.5
840 - val_loss: 0.6256
Epoch 2/30
546/546 [==============================] - 365s 669ms/step - loss: 0.5
489 - val_loss: 0.6026
Epoch 3/30
546/546 [==============================] - 365s 669ms/step - loss: 0.5
208 - val_loss: 0.5833
Epoch 4/30
546/546 [==============================] - 364s 667ms/step - loss: 0.4
974 - val_loss: 0.5669
Epoch 5/30
546/546 [==============================] - 364s 667ms/step - loss: 0.4
781 - val_loss: 0.5515
Epoch 6/30
546/546 [==============================] - 365s 669ms/step - loss: 0.4
621 - val_loss: 0.5406
Epoch 7/30
546/546 [==============================] - 362s 663ms/step - loss: 0.4
487 - val_loss: 0.5349
Epoch 8/30
546/546 [==============================] - 363s 665ms/step - loss: 0.4
374 - val_loss: 0.5269
Epoch 9/30
546/546 [==============================] - 361s 662ms/step - loss: 0.4
273 - val_loss: 0.5218
Epoch 10/30
546/546 [==============================] - 360s 660ms/step - loss: 0.4
184 - val_loss: 0.5130
Epoch 11/30
546/546 [==============================] - 360s 659ms/step - loss: 0.4
112 - val_loss: 0.5127
Epoch 12/30
546/546 [==============================] - 360s 659ms/step - loss: 0.4
043 - val_loss: 0.5053
Epoch 13/30
546/546 [==============================] - 359s 658ms/step - loss: 0.3
976 - val_loss: 0.5035
Epoch 14/30
546/546 [==============================] - 359s 658ms/step - loss: 0.3
920 - val_loss: 0.4984
Epoch 15/30
546/546 [==============================] - 360s 659ms/step - loss: 0.3
868 - val_loss: 0.4952
Epoch 16/30
546/546 [==============================] - 360s 660ms/step - loss: 0.3
823 - val_loss: 0.4949
Epoch 17/30
546/546 [==============================] - 362s 664ms/step - loss: 0.3
780 - val_loss: 0.4943
Epoch 18/30
546/546 [==============================] - 364s 667ms/step - loss: 0.3
739 - val_loss: 0.4911
Epoch 19/30
546/546 [==============================] - 366s 670ms/step - loss: 0.3
698 - val_loss: 0.4851
```

```
Epoch 20/30
546/546 [==============================] - 367s 672ms/step - loss: 0.3
661 - val_loss: 0.4843
Epoch 21/30
546/546 [==============================] - 368s 674ms/step - loss: 0.3
628 - val_loss: 0.4823
Epoch 22/30
546/546 [==============================] - 369s 675ms/step - loss: 0.3
598 - val_loss: 0.4814
Epoch 23/30
546/546 [==============================] - 366s 670ms/step - loss: 0.3
573 - val_loss: 0.4787
Epoch 24/30
546/546 [==============================] - 366s 670ms/step - loss: 0.3
540 - val_loss: 0.4807
Epoch 25/30
546/546 [==============================] - 364s 668ms/step - loss: 0.3
515 - val_loss: 0.4760
Epoch 26/30
546/546 [==============================] - 365s 669ms/step - loss: 0.3
489 - val_loss: 0.4798
Epoch 27/30
546/546 [==============================] - 366s 671ms/step - loss: 0.3
469 - val_loss: 0.4760
Epoch 28/30
546/546 [==============================] - 367s 672ms/step - loss: 0.3
445 - val_loss: 0.4767
Epoch 29/30
546/546 [==============================] - 370s 677ms/step - loss: 0.3
424 - val_loss: 0.4765
Epoch 30/30
546/546 [==============================] - 367s 673ms/step - loss: 0.3
406 - val_loss: 0.4714
```

In [ ]:
```
   # model_dot.save_weights("dot_512_01.h5")
plt.plot(range(30),hist_gconcat.history["loss"])
plt.plot(range(30),hist_gconcat.history["val_loss"])
plt.legend(["Train Loss","Validation Loss"])
plt.show()
```



In [141]:
```
model_concat.save_weights("attention_concat_60.h5")
```

# **Inference**

**Plot attention weights**

**Predict the sentence translation**

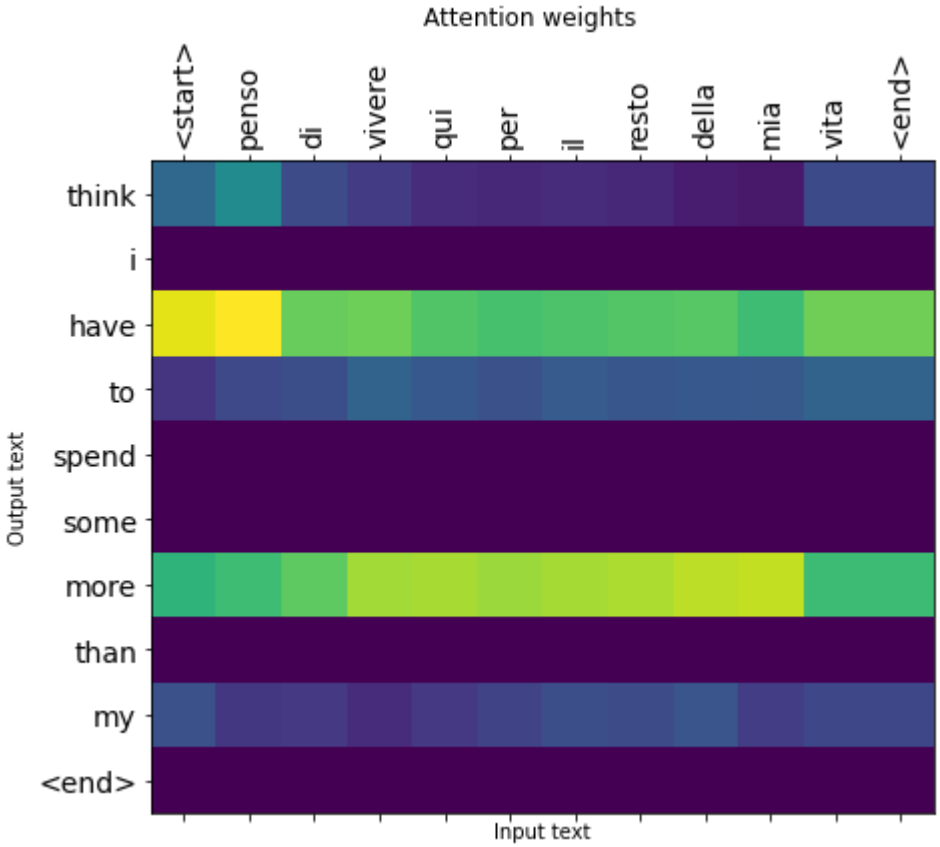
In [153]:
```
model_concat.layers
```

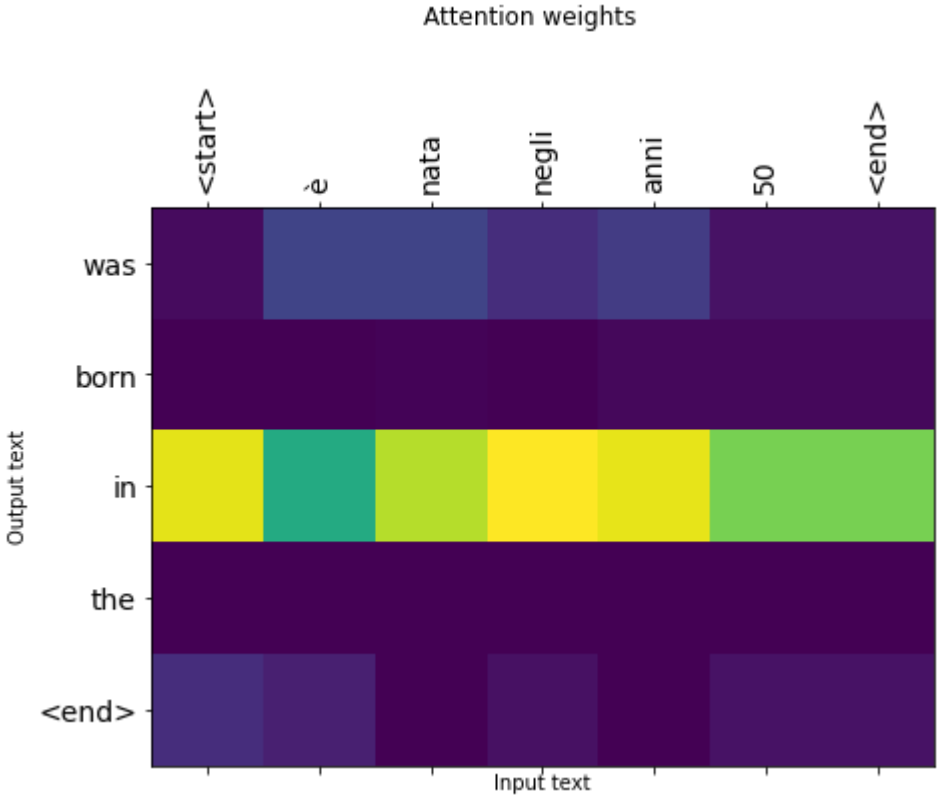
Out[153]:
```
[<__main__.Encoder at 0x7fedc58f3310>, <__main__.Decoder at 0x7fedc58f
3250>]
```

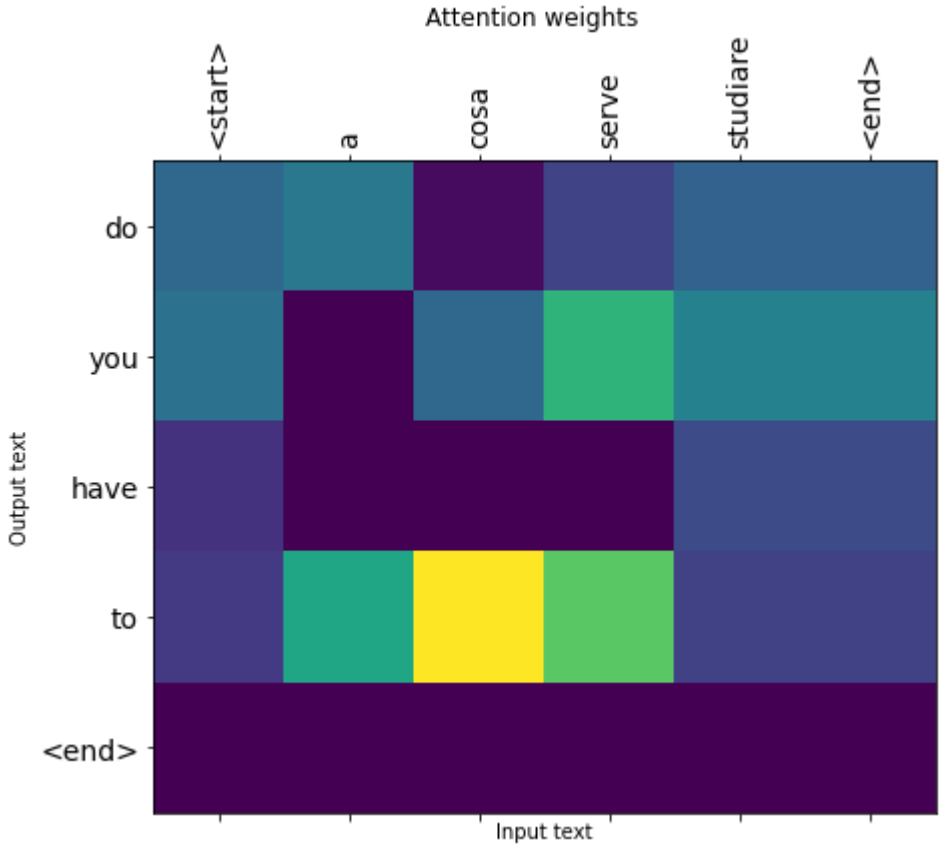
In [158]:
```
def predict_concat(input_sentence):
  '''
  A. Given input sentence, convert the sentence into integers using to
kenizer used earlier
  B. Pass the input_sequence to encoder. we get encoder_outputs, last
 time step hidden and cell state
  C. Initialize index of <start> as input to decoder. and encoder fina
l states as input_states to onestepdecoder.
  D. till we reach max_length of decoder or till the model predicted w
ord <end>:
         predictions, input_states, attention_weights = model.layers
[1].onestepdecoder(input_to_decoder, encoder_output, input_states)
         Save the attention weights
         And get the word using the tokenizer(word index) and then sto
re it in a string.
  E. Call plot_attention(#params)
  F. Return the predicted sentence

  '''
  input_length = 20
  lstm_units = 32
  batch_size = 1

  encoder_seq = tknizer_ita.texts_to_sequences([input_sentence]) # nee
d to pass list of values
  encoder_seq = pad_sequences(encoder_seq, maxlen=input_length, dtype=
'int32', padding='post')
  encoder_output, state_h, state_c = model_concat.layers[0](encoder_se
q, model_concat.layers[0].initialize_states(batch_size))

  cur_vec = np.ones((1,1))
  cur_vec[0,0] = tknizer_eng.word_index['<start>']
  result_sentence = ""
  weights_arr = []

  for i in range(20):
    predictions,dec_state_h,dec_state_c,attention_weights,context_vect
or = model_concat.layers[1].onestepdecoder(cur_vec, encoder_output, st
ate_h, state_c)
    cur_vec = np.reshape(np.argmax(predictions),(1,1))
    state_h = dec_state_h
    state_c = dec_state_c
    index= np.argmax(predictions)
    weights_arr.append(attention_weights.numpy()[0])

    if i==0:
      # print(index)
      continue

    if tknizer_eng.index_word[index] == "<end>":
      return result_sentence,np.array(weights_arr)
    result_sentence += tknizer_eng.index_word[index] + " "

  # print(weights_arr)
  return result_sentence,np.array(weights_arr)
```

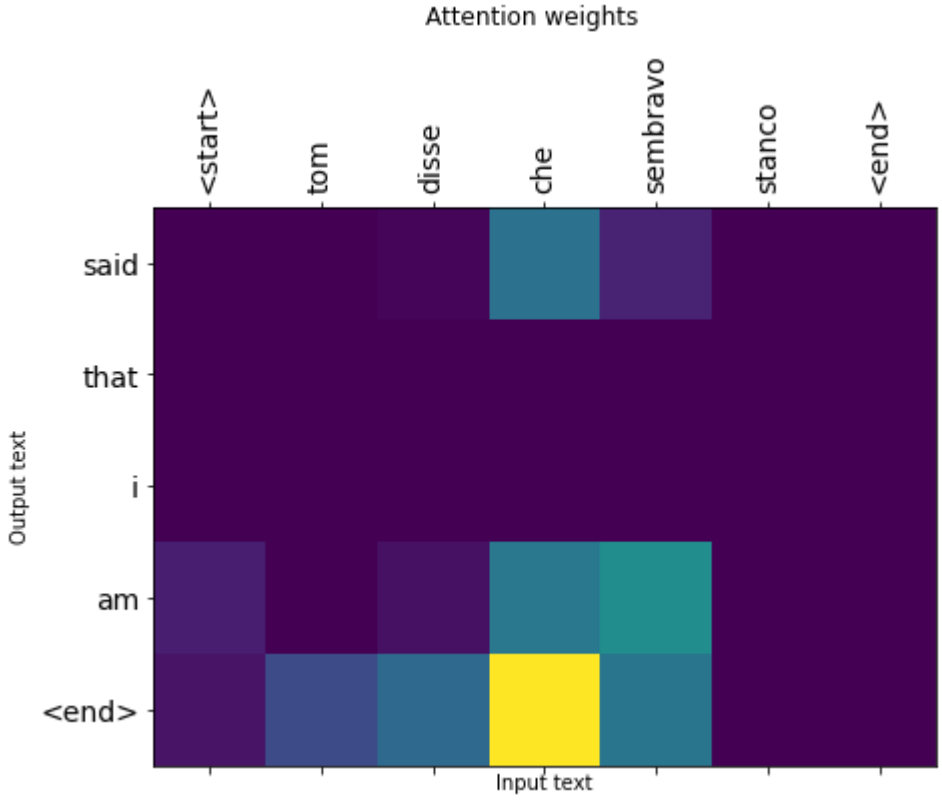
In [159]:
```
import matplotlib.ticker as ticker

def plot_attention_concat(sentence):
  predicted_sentence, attention = predict_concat(sentence)
  attention = np.sum(attention, axis=-1)
  sentence = ["<start>"]+sentence.split()+["<end>"]
  predicted_sentence = predicted_sentence.split()[:-1] + ['<end>']
  fig = plt.figure(figsize=(7, 7))
  ax = fig.add_subplot(1, 1, 1)

  attention = attention[:len(predicted_sentence), :len(sentence)]
  ax.matshow(attention, cmap='viridis', vmin=0.0)
  fontdict = {'fontsize': 14}

  ax.set_xticklabels([''] + sentence, fontdict=fontdict, rotation=90)
  ax.set_yticklabels([''] + predicted_sentence, fontdict=fontdict)
  ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
  ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

  ax.set_xlabel('Input text')
  ax.set_ylabel('Output text')
  plt.suptitle('Attention weights')
```

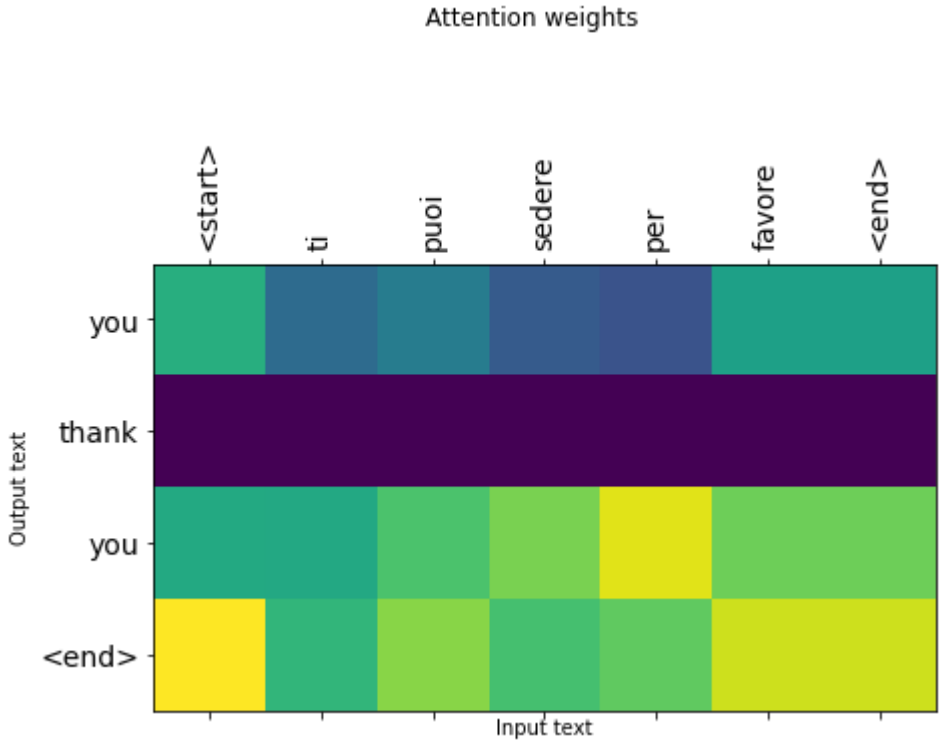
In [181]:
```
for index in np.random.randint(1000,1500,5):
  ita_sentence = test_df.iloc[index]["italian"]
  plot_attention_concat(ita_sentence)
```


Attention weights
<start>
penso
di
vivere
qui
per
il
resto
della
mia
vita
<end>
think
i
have
to
spend
some
more
than
my
<end>
Output text
Input text


Attention weights
<start>
è
nata
negli
anni
50
<end>
was
born
in
the
<end>
Output text
Input text


Attention weights
<start>
a
cosa
serve
studiare
<end>
do
you
have
to
<end>
Output text
Input text


Attention weights
<start>
tom
disse
che
sembravo
stanco
<end>
said
that
i
am
<end>
Output text
Input text




**Calculate BLEU score**

In [168]:
```
from nltk.translate.bleu_score import SmoothingFunction
smoothie = SmoothingFunction().method1

def aveg_bleu_scores_general(test_data,n):
  sample_list = random.sample(range(len(test_data)),n)
  average_bleu = 0
  pred_data = []
  individual_bleu = []

  for i in sample_list:
    test_sentence, true_sentence = test_data.iloc[i,[0,1]]
    pred_sentence, att = predict_general(test_sentence)
    bleu_score = bleu.sentence_bleu([true_sentence.split()],pred_sente
nce.split())
    average_bleu += bleu_score

    # pred_data.append((true_sentence, pred_sentence))
    individual_bleu.append(bleu_score)

  return average_bleu/n, individual_bleu
```

In [188]:
```
import nltk.translate.bleu_score as bleu
import random

test_df = validation.copy()
test_df = test_df[["italian","english_out"]]
test_df["english_out"] = test_df["english_out"].apply(lambda x: x.spli
t("<end>")[0])

avg_score, individual_score = aveg_bleu_scores_general(test_df,1000)
print("Average BLEU score : ",avg_score)
```

```
Average BLEU score :  0.5921551651429544
```

In [190]:
```
!jupyter nbconvert --to html "/content/Self_Attention.ipynb"
```

```
[NbConvertApp] Converting notebook /content/Self_Attention.ipynb to ht
ml
[NbConvertApp] Writing 841729 bytes to /content/Self_Attention.html
```

In [ ]:
```

```