Seq2SeqImplementation___Assignment

April 22, 2021

1 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.

[ ]:
```
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
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
from google.colab import drive
from tensorflow.keras.regularizers import l1, l2, L1, L2
from tensorflow.keras.layers import Bidirectional
from tensorflow.keras.callbacks import EarlyStopping
import tensorboard
import datetime
import matplotlib.ticker as ticker
import os
%load_ext tensorboard
```

```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
```

[ ]:
```
drive.mount('/content/gdrive', force_remount=True)
```

```
Mounted at /content/gdrive
```

1.1 Task -1: Simple Encoder and Decoder

Implement simple Encoder-Decoder model

1. Download the **Italian** to **English** translation dataset from here
2. You will find **ita.txt** file in that ZIP, you can read that data using python and preprocess that data this way only:
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

2 Load data

[ ]:
```
# /content/gdrive/MyDrive/Colab Notebooks/Seq_Seq_attention/ita.txt

filePath = "/content/gdrive/MyDrive/Colab Notebooks/Seq_Seq_attention/ita.txt"


with open(filePath, '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','italian'])
print(data.shape)
data.head()
```

```
(341554, 2)
```

[ ]:

	english	italian
0	Hi.	Ciao!
1	Run!	Corri!
2	Run!	Corra!
3	Run!	Correte!
4	Who?	Chi?

[ ]:
```
```

3 Preprocessing data

[ ]:
```
def decontractions(phrase):
    """decontracted takes text and convert contractions into natural form.
     ref: https://stackoverflow.com/questions/19790188/
↪expanding-english-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):
    # convert all the text into lower letters
    # use this function to remove the contractions: https://gist.github.com/
↪anandborad/d410a49a493b56dace4f814ab5325bbd
    # remove all the spacial characters: except space ' '
    text = text.lower()
    text = decontractions(text)
    text = re.sub('[^A-Za-z0-9 ]+', '', text)
    return text

def preprocess_ita(text):
    # convert all the text into lower letters
    # remove the words betweent brakets ()
```

```
    # remove these characters: {'$', ')', '?', '"', '’', '.',  '°', '!', ';', '/
↪', "'", '€', '%', ':', ',', '('}
    # replace these spl characters with space: '\u200b', '\xa0', '-', '/'
    # we have found these characters after observing the data points, feel free␣
↪to explore more and see if you can do find more
    # you are free to do more proprocessing
    # note that the model will learn better with better preprocessed data

    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()
```

[ ]:

	english	italian
0	hi	ciao
1	run	corri
2	run	corra
3	run	correte
4	who	chi

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

[ ]:
```
for i in range(0,101,10):
    print(i,np.percentile(ita_lengths, i))
for i in range(90,101):
    print(i,np.percentile(ita_lengths, i))
for i in [99.1,99.2,99.3,99.4,99.5,99.6,99.7,99.8,99.9,100]:
    print(i,np.percentile(ita_lengths, i))
```

```
0 1.0
10 3.0
20 4.0
30 4.0
40 5.0
50 5.0
60 6.0
70 6.0
80 7.0
```

```
90 8.0
100 92.0
90 8.0
91 8.0
92 8.0
93 9.0
94 9.0
95 9.0
96 9.0
97 10.0
98 11.0
99 12.0
100 92.0
99.1 12.0
99.2 12.0
99.3 12.0
99.4 13.0
99.5 13.0
99.6 14.0
99.7 15.0
99.8 16.0
99.9 20.0
100 92.0
```

[ ]:
```
for i in range(0,101,10):
    print(i,np.percentile(eng_lengths, i))
for i in range(90,101):
    print(i,np.percentile(eng_lengths, i))
for i in [99.1,99.2,99.3,99.4,99.5,99.6,99.7,99.8,99.9,100]:
    print(i,np.percentile(eng_lengths, i))
```

```
0 1.0
10 4.0
20 4.0
30 5.0
40 5.0
50 6.0
60 6.0
70 7.0
80 7.0
90 8.0
100 101.0
90 8.0
91 9.0
92 9.0
93 9.0
94 9.0
95 9.0
```

```
96 10.0
97 10.0
98 11.0
99 12.0
100 101.0
99.1 12.0
99.2 13.0
99.3 13.0
99.4 13.0
99.5 14.0
99.6 14.0
99.7 15.0
99.8 16.0
99.9 20.0
100 101.0
```

[ ]:
```
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 <end> in␣
↪tokenizer
data.head()
```

[ ]:
```
   italian  english_inp english_out
0     ciao   <start> hi    hi <end>
1    corri  <start> run   run <end>
2    corra  <start> run   run <end>
3  correte  <start> run   run <end>
4      chi  <start> who   who <end>
```

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

[ ]:
```
                                                italian  …
english_out
131418                        io non ho nulla da scrivere  …
i have nothing to write <end>
290079     noi andremo a parlare con tom questo pomeriggio  …          we will go
talk to tom this afternoon <end>
218944                     lei può vedere qualcosa lì dentro  …
can you see anything in there <end>
```

```
255319                        perché importa cosa succede  …             why
does it matter what happens <end>
139947                           noi ci divertiremo molto  …
we will have a great time <end>
310002  per piacere mettete una zolletta di zucchero n…  …      please put a
lump of sugar in my coffee <end>
339268  non crederà a dove sono stati tom e mary per l…  …  you will not
believe where tom and mary went f…
85613                                      quanti ne hai  …
how many do you have <end>
261898                ha risolto il problema con facilità  …            she
solved the problem with ease <end>
125035                             noi abbiamo già finito  …
we have already finished <end>

[10 rows x 3 columns]
```

[ ]:
```python
from sklearn.model_selection import train_test_split
train, validation = train_test_split(data, test_size=0.2)
```

[ ]:
```python
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>'
```

```
(272932, 3) (68234, 3)
```

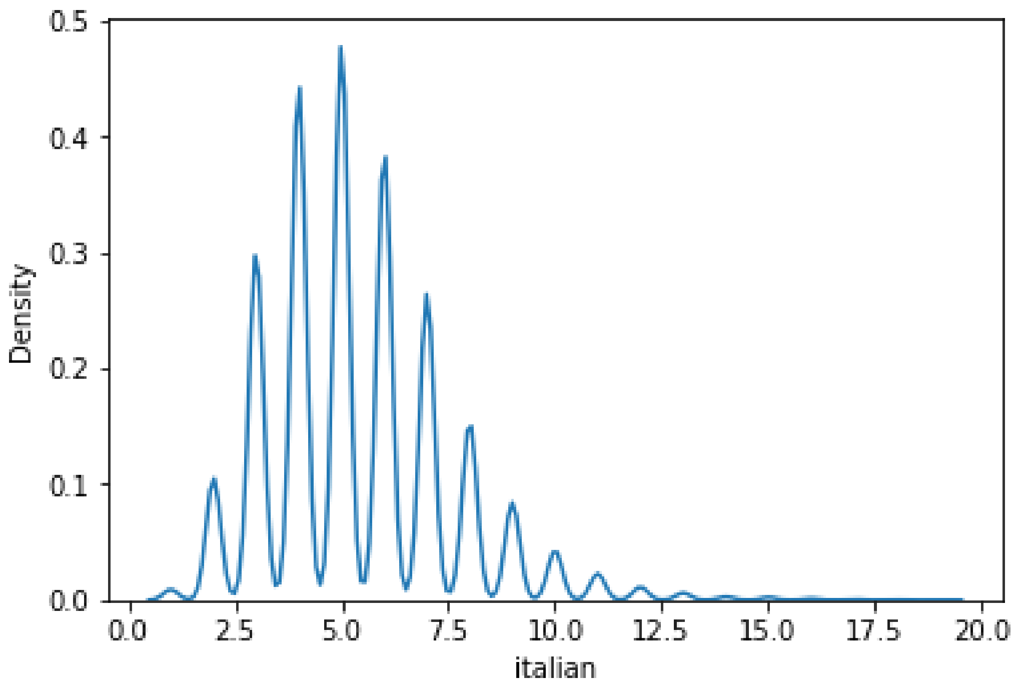
[ ]:
```python
train.head()
```

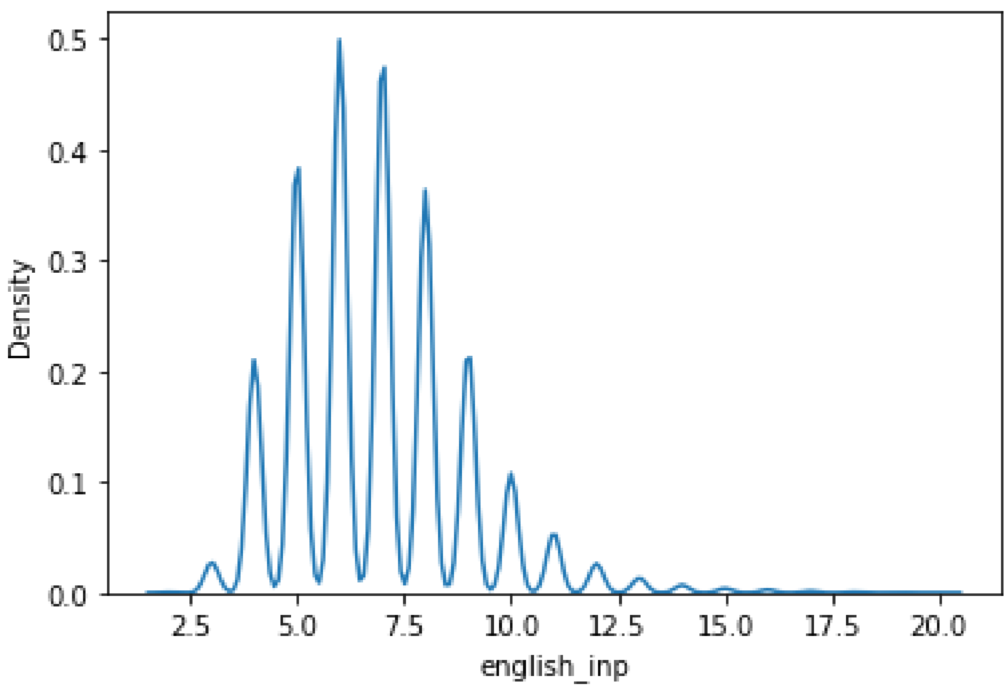
[ ]:
```
                                                 italian  …
english_out
75684                                    è tutta colpa sua  …
it is all your fault <end> <end>
168773                        tom ha avuto ragione finora  …
tom has been right so far <end>
168395          tom non riesce a trovare il suo biglietto  …
tom can not find his ticket <end>
332071  sembra che ci siano diverse ragioni per il suo…  …  there seem to be
several reasons for his failu…
282780                    cosa fanno tutti dopo la scuola  …           what
does everyone do after school <end>

[5 rows x 3 columns]
```

[ ]:
```
validation.head()
```

[ ]:
```
                         italian  ...                         english_out
121424  stanno giocando a scacchi  ...         they are playing chess <end>
174052  vendono della guinness qui  ...       do they sell guinness here <end>
224296      non ho più paura di te  ...  i am not scared of you anymore <end>
83152        voi mi state seguendo  ...           are you following me <end>
163095      ho letto molte riviste  ...       i read a lot of magazines <end>

[5 rows x 3 columns]
```

[ ]:
```
ita_lengths = train['italian'].str.split().apply(len)
eng_lengths = train['english_inp'].str.split().apply(len)
import seaborn as sns
sns.kdeplot(ita_lengths)
plt.show()
sns.kdeplot(eng_lengths)
plt.show()
```





[ ]:
```
tknizer_ita = Tokenizer(filters='!"#$%&()*+,-./:;=?@[\\]^_`{|}~\t\n', oov_token=1)
tknizer_ita.fit_on_texts(train['italian'].values)
tknizer_eng = Tokenizer(filters='!"#$%&()*+,-./:;=?@[\\]^_`{|}~\t\n', oov_token=1)
tknizer_eng.fit_on_texts(train['english_inp'].values)
```

[ ]:
```
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)
```

```
12796
26122
```

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

[ ]:
```
(2, 10096)
```

[ ]:
```
embeddings_index = dict()
embeddingfilePath = '/content/gdrive/MyDrive/Colab Notebooks/Seq_Seq_attention/glove.6B.300d.txt'
```

```
f = open(embeddingfilePath)
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, 300))
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
```

[ ]:
```
vocab_size_encoder = vocab_size_ita + 1
vocab_size_decoder = vocab_size_eng + 1
embedding_size_encoder = 300
embedding_size_decoder = 300
input_length_encoder = 20
input_length_decoder = 20
units_encoder = 128
units_decoder = 128
```

[ ]:
```
embedding_matrix.shape
```

[ ]:
```
(12797, 300)
```

3.1 Implement custom encoder decoder

4 DATA PREPARATION

[ ]:
```
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.
 ↪encoder_inps[i]]) # need to pass list of values
        self.decoder_inp_seq = self.tknizer_eng.texts_to_sequences([self.
 ↪decoder_inps[i]])
        self.decoder_out_seq = self.tknizer_eng.texts_to_sequences([self.
 ↪decoder_outs[i]])
```

```
        self.encoder_seq = pad_sequences(self.encoder_seq, maxlen=self.max_len,␣
↪dtype='float32', padding='post')
        self.decoder_inp_seq = pad_sequences(self.decoder_inp_seq, maxlen=self.
↪max_len, dtype='float32', padding='post')
        self.decoder_out_seq = pad_sequences(self.decoder_out_seq, maxlen=self.
↪max_len, dtype='float32', padding='post')
        return self.encoder_seq, self.decoder_inp_seq, self.decoder_out_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 samples in␣
↪zip(*data)]
        # we are creating data like ([italian, english_inp], english_out) 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)
```

[ ]:

```
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)

# ([italian, english_inp], english_out)
```

```
print(train_dataloader[0][0][0].shape, train_dataloader[0][0][1].shape,␣
 ↪train_dataloader[0][1].shape)
```

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

[ ]:
```
eng_index_word_dict = {v: k for k, v in tknizer_eng.word_index.items()}
ita_index_word_dict = {v: k for k, v in tknizer_ita.word_index.items()}
```

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

[ ]:
```
(2, 10096)
```

5 Task - 1 Encoder

[ ]:
```
class Encoder(tf.keras.Model):
    '''
    Encoder model -- That takes a input sequence and returns␣
 ↪encoder-outputs,encoder_final_state_h,encoder_final_state_c
    '''

    def __init__(self,inp_vocab_size,embedding_size,lstm_size,input_length):
      super(Encoder, self).__init__()

      self.vocab_size = inp_vocab_size
      self.embedding_dim = embedding_size
      self.input_length = input_length
      self.enc_units= lstm_size

      #Initialize Embedding layer
      self.embd_Layer = Embedding(input_dim=(self.vocab_size),
                                  output_dim=self.embedding_dim,
                                  input_length=self.input_length,
                                  mask_zero=True,
                                  name="ecoder_embedding_layer")

      #Intialize Encoder LSTM layer
      self.lstm = LSTM(units=self.enc_units,
                       return_sequences=True,
                       return_state=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 the␣
↪embedding layer ouput to encoder_lstm
        returns -- encoder_output, last time step's hidden and cell state
        '''
        embedding_sequence = self.embd_Layer(input_sequence)
        self.lstm_output, self.lstm_state_h, self.lstm_state_c = self.
↪lstm(embedding_sequence, initial_state=states)

        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_units], cell␣
↪state zeros is of size [32,lstm_units]
      '''
      self.lstm_state_h=tf.zeros([batch_size, self.enc_units])
      self.lstm_state_c= tf.zeros([batch_size, self.enc_units])

      return [self.lstm_state_h, self.lstm_state_c]
```

Grader function - 1

[ ]:
```
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)
```

```
    #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())
```

```
True
```

6 Task - 1 Decoder

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

    def __init__(self,out_vocab_size,embedding_size,lstm_size,input_length):

      super().__init__()

      self.vocab_size = out_vocab_size
      self.embedding_size = embedding_size
      self.lstm_size = lstm_size
      self.input_length = input_length

      self.lstm_output = 0
      self.lstm_state_h = 0
      self.lstm_state_c = 0

      #Initialize Embedding layer
      self.embd = Embedding(input_length=self.input_length,
                            output_dim=self.embedding_size,
                            input_dim=self.vocab_size)
      #Intialize Decoder LSTM layer
      self.lstm = LSTM(units=self.lstm_size, 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 the␣
↪embedding layer ouput to decoder_lstm

        returns -- decoder_output,decoder_final_state_h,decoder_final_state_c
       '''
       embd_sequence = self.embd(input_sequence)
       self.lstm_output, self.lstm_state_h,self.lstm_state_c = self.
↪lstm(embd_sequence, initial_state=initial_states)

       return self.lstm_output, self.lstm_state_h,self.lstm_state_c
```

Grader function - 2

[ ]:
```
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,dec_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_length )
    output,_,_=decoder(target_sentences, states)
    assert(output.shape==(batch_size,input_length,dec_units))
    return True
print(grader_decoder())
```

True

7 Task - 1 Encoder_Decoder Model

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

    def __init__(self,*params):

        super().__init__()

        #Create encoder object
        self.encoder = Encoder(inp_vocab_size=vocab_size_encoder,
                               embedding_size=embedding_size_encoder,
                               lstm_size=units_encoder,
                               input_length=input_length_encoder)
        #Create decoder object
        self.decoder = Decoder(out_vocab_size=vocab_size_decoder,␣
↪embedding_size=embedding_size_decoder, lstm_size=units_decoder,
                               input_length=input_length_decoder)
        #Intialize Dense layer(out_vocab_size) with activation='softmax'
        self.dense = Dense(activation='softmax', units=vocab_size_decoder)


    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 states 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_intial_states = self.encoder.initialize_states(1024)
        encoder_output, encoder_h, encoder_c = self.encoder(input,␣
↪encoder_intial_states)
        decoder_output, _, _ = self.decoder(output, [encoder_h, encoder_c])
        output = self.dense(decoder_output)
        return output
```

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

8 Task - 1 Model Training

[ ]:
```
class Custom_callback(tf.keras.callbacks.Callback):

  def on_epoch_end(self, epoch, logs=None):
        keys = list(logs.keys())
        # print("End epoch {} of training; got log keys: {}".format(epoch,␣
 ↪keys))
        for sentence in validation['italian']:
          predicted_sentence = predict(sentence, self.model)
          break
```

[ ]:
```
import os

model  = Encoder_decoder()
optimizer = tf.keras.optimizers.Adam()
model.compile(optimizer=optimizer,loss='sparse_categorical_crossentropy')


train_steps=train.shape[0]//1024
valid_steps=validation.shape[0]//1024

custom_callback = Custom_callback()

log_dir="logs/fit/model_enc_dec"
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir,
                                                      histogram_freq=1,
                                                      write_graph=True,
                                                      write_grads=True)

es_callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=2,␣
 ↪verbose=1)


model.fit_generator(train_dataloader,
                    steps_per_epoch=train_steps,
                    epochs=25, callbacks=[tensorboard_callback, es_callback])
model.summary()
```

```
WARNING:tensorflow:`write_grads` will be ignored in TensorFlow 2.0 for the
`TensorBoard` Callback.
Epoch 1/25
266/266 [==============================] - 153s 577ms/step - loss: 2.6944
Epoch 2/25
266/266 [==============================] - 153s 576ms/step - loss: 1.6416
Epoch 3/25
266/266 [==============================] - 153s 576ms/step - loss: 1.4491
```

```
Epoch 4/25
266/266 [==============================] - 153s 577ms/step - loss: 1.2359
Epoch 5/25
266/266 [==============================] - 153s 576ms/step - loss: 1.0732
Epoch 6/25
266/266 [==============================] - 153s 576ms/step - loss: 0.9339
Epoch 7/25
266/266 [==============================] - 153s 576ms/step - loss: 0.8120
Epoch 8/25
266/266 [==============================] - 153s 577ms/step - loss: 0.7020
Epoch 9/25
266/266 [==============================] - 154s 577ms/step - loss: 0.6058
Epoch 10/25
266/266 [==============================] - 154s 577ms/step - loss: 0.5262
Epoch 11/25
266/266 [==============================] - 153s 577ms/step - loss: 0.4601
Epoch 12/25
266/266 [==============================] - 153s 577ms/step - loss: 0.4054
Epoch 13/25
266/266 [==============================] - 154s 578ms/step - loss: 0.3598
Epoch 14/25
266/266 [==============================] - 153s 576ms/step - loss: 0.3218
Epoch 15/25
266/266 [==============================] - 153s 577ms/step - loss: 0.2898
Epoch 16/25
266/266 [==============================] - 153s 577ms/step - loss: 0.2625
Epoch 17/25
266/266 [==============================] - 153s 577ms/step - loss: 0.2392
Epoch 18/25
266/266 [==============================] - 154s 577ms/step - loss: 0.2190
Epoch 19/25
266/266 [==============================] - 153s 577ms/step - loss: 0.2015
Epoch 20/25
266/266 [==============================] - 153s 577ms/step - loss: 0.1862
Epoch 21/25
266/266 [==============================] - 154s 577ms/step - loss: 0.1727
Epoch 22/25
266/266 [==============================] - 154s 578ms/step - loss: 0.1607
Epoch 23/25
266/266 [==============================] - 153s 577ms/step - loss: 0.1500
Epoch 24/25
266/266 [==============================] - 154s 577ms/step - loss: 0.1405
Epoch 25/25
266/266 [==============================] - 153s 576ms/step - loss: 0.1318
Model: "encoder_decoder_11"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
```

```
encoder_18 (Encoder)         multiple                  8054448
_________________________________________________________________
decoder_22 (Decoder)         multiple                  4059348
_________________________________________________________________
dense_53 (Dense)             multiple                  1651071
=================================================================
Total params: 13,764,867
Trainable params: 13,764,867
Non-trainable params: 0
_________________________________________________________________
```

[ ]:
```
%tensorboard --logdir logs/fit
```

[ ]:
```
model_path = "/content/gdrive/MyDrive/Colab Notebooks/Seq_Seq_attention/
 ↪enc_dec_weights/"
model.save_weights(model_path)

# # Save the weights
# model.save_weights('./checkpoints/my_checkpoint')

# # Create a new model instance
# model = create_model()

# # Restore the weights
# model.load_weights(model_path)
```

[ ]:
```
def predict(input_sentence, model):

  # tokenizing sentence
  tokens = np.array(tknizer_ita.texts_to_sequences([input_sentence]))
  padded_tokens = pad_sequences(tokens, maxlen=20, padding='post')

  # initialize encoder initial state
  encoder_intial_states = model.layers[0].initialize_states(1)

  # feed padded_tokens and initial state to encoder
  enc_output, enc_state_h, enc_state_c = model.layers[0](padded_tokens,␣
 ↪encoder_intial_states)

  # start with <str> token, for feeding to decoder layer for 1st time
  cur_vec = tf.expand_dims([tknizer_eng.word_index['<start>']], 0)

  # prepare initial state for decoder layer
  dec_states_input = [enc_state_h, enc_state_c]

  # initialize output sentence
  sent = ''
```

```
#
 for i in range(20):

   # pass predicted word index to decoder embd layer
   cur_emb = model.layers[1].embd(cur_vec)

   # predicted word embedding to decoder layer
   [prediction, dec_state_h, dec_state_c] = model.layers[1].lstm(cur_emb,␣
↪initial_state=dec_states_input)

   # predicted word to dense layer
   infe_output = model.layers[2](prediction)

   # prepare hidden states for input to next time step decoder from last time␣
↪step decoder
   dec_states_input = [dec_state_h, dec_state_c]

   # find the index of word with maximum probability
   infe_output=np.argmax(infe_output,-1)

   # get the index
   word_index = infe_output[0][0]

   # if word index is <str>, continue
   if word_index == 0:
     cur_vec = np.reshape(np.argmax(infe_output), (1, 1))
     continue

   # if word index is <end>, stop predicting
   if eng_index_word_dict[word_index] == "<end>":
     return sent

   # append predicted word to sentence
   sent=sent+' '+eng_index_word_dict[int(word_index)]

   # reshape the predicted word index, for feeding next decoder time step
   cur_vec = np.reshape(int(word_index), (1, 1))



 return sent
```

8.1 BLUE SCORE

[ ]:
```
# https://stackoverflow.com/questions/32395880/calculate-bleu-score-in-python/
↪39062009
import nltk
import warnings

def bluescore(orginal, predicted):
  orginal_tokens = orginal.split()
  predicted_tokens = predicted.split()
  BLEUscore = nltk.translate.bleu_score.sentence_bleu([orginal_tokens],␣
↪predicted_tokens, weights = (0.5, 0.5))
  return BLEUscore
```

[ ]:
```
validation.head(1)
```

[ ]:
```
                         italian  …              english_out
64322  tom è entrato in macchina  …  tom got in the car <end>

[1 rows x 3 columns]
```

9 Task - 1 Predicting validation data

[ ]:
```
count = 0
bleu_score = []
for i, row in validation.iterrows():
  if count == 1000:
    break
  italian_sentence = row['italian']
  predicted_eng_sentence = predict(italian_sentence, model)
  original_eng_sentence = re.sub("<start>", "", row['english_inp'])
  score = bluescore(original_eng_sentence, predicted_eng_sentence)

  # print("actual sentence:     {}".format(original_eng_sentence))
  # print("predicted sentence:  {}".format(predicted_eng_sentence))

  bleu_score.append(score)
  count += 1
```

```
/usr/local/lib/python3.6/dist-packages/nltk/translate/bleu_score.py:490:
UserWarning:
Corpus/Sentence contains 0 counts of 2-gram overlaps.
BLEU scores might be undesirable; use SmoothingFunction().
  warnings.warn(_msg)
```

10 Task - 1 Simple Encoder & Decoder BLEU Score

[ ]:
```
enc_dec_bleu_score = np.mean(bleu_score)
```

[ ]:
```
print("BLEU SCORE {}".format(enc_dec_bleu_score))
```

```
BLEU SCORE 0.7427681544632555
```

11 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** 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
6. The attention layer has to be written by yourself only. The main objective of this assignment is to read and implement a paper on yourself so please do it yourself.
7. Please implement the class **onestepdecoder** as mentioned in the assignment instructions.
8. You can use any tf.Keras highlevel API's to build and train the models. Check the reference notebook for better understanding.
9. Use BLEU score as metric to evaluate your model. You can use any loss function you need.
10. You have to use Tensorboard to plot the Graph, Scores and histograms of gradients.
11. Resources:
    a. Check the reference notebook
    b. Resource 1
    c. Resource 2
    d. Resource 3

11.0.1 Implement custom encoder decoder and attention layers

12 Task - 2

[ ]:
```
units_encoder = 1024
units_decoder = 1024
```

13 Task-2 Encoder

[ ]:
```
class Encoder(tf.keras.Model):

    def __init__(self,inp_vocab_size,embedding_size,lstm_size,input_length):
      super(Encoder, self).__init__()

      self.vocab_size = inp_vocab_size
      self.embedding_dim = embedding_size
      self.input_length = input_length
      self.enc_units= lstm_size

      #Initialize Embedding layer
      self.embd_Layer = Embedding(input_dim=(self.vocab_size),
                                  output_dim=self.embedding_dim,
                                  input_length=self.input_length,
                                  mask_zero=True,
                                  name="ecoder_embedding_layer")

      #Intialize Encoder LSTM layer
      self.lstm = LSTM(units=self.enc_units,
                       return_sequences=True,
                       return_state=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 the␣
↪embedding layer ouput to encoder_lstm
          returns -- encoder_output, last time step's hidden and cell state
        '''

        embedding_sequence = self.embd_Layer(input_sequence)

        self.lstm_output, self.lstm_state_h, self.lstm_state_c = self.
↪lstm(embedding_sequence, initial_state=states)
```

```
        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_units], cell␣
↪state zeros is of size [32,lstm_units]
      '''
      self.lstm_state_h=tf.zeros([batch_size, self.enc_units])
      self.lstm_state_c= tf.zeros([batch_size, self.enc_units])

      return self.lstm_state_h, self.lstm_state_c
```

Grader function - 1

```
[ ]: 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],maxval=vocab_size,minval=0,dtype=tf.
↪int32)
        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())
```

```
True
```

14 Task - 2 Attention

[ ]:
```
class Attention(tf.keras.layers.Layer):
  '''
    Class the calculates score based on the scoring_function using Bahdanu␣
 ↪attention mechanism.
  '''
  def __init__(self,scoring_function, att_units):
    super(Attention, self).__init__()

    self.scoring_function = scoring_function


    # Please go through the reference notebook and research paper to complete␣
 ↪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
      self.W1 = tf.keras.layers.Dense(att_units)
      pass
    elif scoring_function == 'concat':
      # Intialize variables needed for Concat score function here
      self.W1 = tf.keras.layers.Dense(att_units)
      self.W2 = tf.keras.layers.Dense(att_units)
      self.V = tf.keras.layers.Dense(1)
      pass


  def call(self,decoder_hidden_state,encoder_output):



    # we need to calculate weight for contect vector.
    # we have decoder current hidden state of shape (batch_size, units)
    # we have enocoder contect vector of shape (batch_size, max_seq_length,␣
 ↪units)
```

```
    # for calculating weights by using different scoring techniqes, we need to␣
↪extend the dimension of decoder_hidden_states
    # to (batch_size, 1, units).

    # score = decoder_current_input_state_for_current_time_step *␣
↪encoder_context_vector == (batch_size, max_seq_len, 1)
    if self.scoring_function == 'dot':
        # extending dimension of decoder_current_hidden_state
        decoder_hidden_state_with_time_axis = tf.
↪expand_dims(decoder_hidden_state, 1)

        # score = (encoder_output.T * decoder_hidden_state_current_time_step)
        score = tf.matmul(encoder_output, decoder_hidden_state_with_time_axis,␣
↪transpose_b=True)

        # on axis 1, we are normalizing weights for all words. so that the␣
↪addition of the weights will be 1.
        attention_weights = tf.nn.softmax(score, axis=1)

        # weighted encoder_output shape = (batch_size, seq_len, units)
        context_vector = attention_weights * encoder_output


        # the context vector will now fed to decoder layer, therefor need to␣
↪reduce dimension to (batch_size, units)
        context_vector = tf.reduce_sum(context_vector, axis=1)


        return context_vector, attention_weights


    elif self.scoring_function == 'general':

        # Implement General score function here
        decoder_hidden_state_with_time_axis = tf.
↪expand_dims(decoder_hidden_state, 1)
        score = tf.matmul(encoder_output, self.
↪W1(decoder_hidden_state_with_time_axis), transpose_b=True)
        attention_weights = tf.nn.softmax(score, axis=1)
        context_vector = attention_weights * encoder_output
        context_vector = tf.reduce_sum(context_vector, axis=1)

        return context_vector, attention_weights

    elif self.scoring_function == 'concat':
```

```
        decoder_hidden_state_with_time_axis = tf.
↪expand_dims(decoder_hidden_state, 1)

        score = self.V(tf.nn.tanh(self.W1(decoder_hidden_state_with_time_axis)␣
↪+ self.W2(encoder_output)))

        attention_weights = tf.nn.softmax(score, axis=1)

        context_vector = attention_weights * encoder_output

        context_vector = tf.reduce_sum(context_vector, axis=1)


        return context_vector, attention_weights
```

Grader function - 2

```
[ ]: 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
```

15 Task - 2 OneStepDecoder

[ ]:
```
class One_Step_Decoder(tf.keras.Model):
  def __init__(self,tar_vocab_size, embedding_dim, input_length, dec_units␣
↪,score_fun ,att_units):
      super(One_Step_Decoder, self).__init__()

      # Initialize decoder embedding layer, LSTM and any other objects needed
      self.embd = tf.keras.layers.Embedding(input_dim=tar_vocab_size,␣
↪output_dim=embedding_dim, input_length=input_length)
      self.lstm = tf.keras.layers.LSTM(units=att_units, return_sequences=True,␣
↪return_state=True, recurrent_initializer='glorot_uniform')
      self.dense = tf.keras.layers.Dense(units=tar_vocab_size)
      self.attention = Attention(scoring_function=score_fun,␣
↪att_units=att_units)


  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 the 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
    '''
    # decoder initial hidden stae = encoder hidden state
    prev_dec_hidden_state = [state_h, state_c]
    prev_dec_hidden_state = tf.reduce_sum(prev_dec_hidden_state, 0)

    # B. Using the encoder_output and decoder hidden state, compute the context␣
↪vector.
    context_vector, attention_weights = self.attention(prev_dec_hidden_state,␣
↪encoder_output)


    # A. Pass the input_to_decoder to the embedding layer and then get the␣
↪output(batch_size,1,embedding_dim)
    dec_embedding_vector = self.embd(input_to_decoder)
```

```
    # C. Concat the context vector with the step A output
    emb_context_concat = tf.concat([tf.expand_dims(context_vector, 1),␣
↪dec_embedding_vector], axis=2)


    # D. Pass the Step-C output to LSTM/GRU and get the decoder output and␣
↪states(hidden and cell state)
    dec_output, dec_state_h, dec_state_c = self.lstm(emb_context_concat,␣
↪initial_state=[state_h, state_c])

    dec_output = tf.reshape(dec_output, (-1, dec_output.shape[2]))

    # E. Pass the decoder output to dense layer(vocab size) and store the␣
↪result into output.
    output = self.dense(dec_output)

    return output, dec_state_h, dec_state_c, attention_weights, context_vector
```

Grader function - 3

```
[ ]: 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 functions 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,␣
↪input_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,dec_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=onestepdecoder(input_to_decoder,enc
    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
```

16 Task - 2 Decoder

[ ]:
```
class Decoder(tf.keras.Model):
    def __init__(self,out_vocab_size, embedding_dim, input_length, dec_units␣
 ↪,score_fun ,att_units):
        super(Decoder, self).__init__()


        #Intialize necessary variables and create an object from the class␣
 ↪onestepdecoder
        self.one_step_decoder = One_Step_Decoder(tar_vocab_size=out_vocab_size,
                                                 embedding_dim=embedding_dim,
                                                 input_length=input_length,
                                                 dec_units=dec_units,
                                                 score_fun=score_fun,
                                                 att_units=att_units)




    def call(self, input_to_decoder, encoder_output, encoder_h, encoder_c):



          #Initialize an empty Tensor array, that will store the outputs at each␣
 ↪and every time step
```

```
        #Create a tensor array as shown in the reference notebook
        # input_to_decoder.shape[1]
        # tf.shape(x)[0]

        output_tensor = tf.TensorArray(dtype=tf.float32, dynamic_size=False,␣
↪size=tf.shape(input_to_decoder)[1], name="output_arrays")

        decoder_hidden_state = encoder_h
        decoder_cell_state = encoder_c



        for i in range(tf.shape(input_to_decoder)[1]):
          dec_input = input_to_decoder[:,i]
          dec_input = tf.expand_dims(dec_input, 1)

          output, decoder_hidden_state, decoder_cell_state, attention_weights,␣
↪context_vector = self.one_step_decoder(dec_input,
                                                                              ␣
↪                                     encoder_output,
                                                                              ␣
↪                                     decoder_hidden_state,
                                                                              ␣
↪                                     decoder_cell_state)

          output_tensor = output_tensor.write(i, output)


        #Iterate till the length of the decoder input
            # Call onestepdecoder for each token in decoder_input
            # Store the output in tensorarray
        # Return the tensor array
        output_tensor = output_tensor.stack()
        output_tensor = tf.transpose(output_tensor, [1, 0, 2])

        return output_tensor
```

Grader function - 4

[ ]:
```
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 functions 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,dec_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_units␣
↪,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
```

17 Task - 2 Encoder Decoder model - DOT Scoring function

[ ]:
```
embedding_size_encoder, units_encoder, input_length_encoder, vocab_size_encoder
```

[ ]:
```
(300, 1024, 20, 26138)
```

[ ]:
```
class Encoder_decoder(tf.keras.Model):
  def __init__(self):
    super().__init__()
```

```
    #Intialize objects from encoder decoder

    #Create encoder object
    self.encoder = Encoder(inp_vocab_size=vocab_size_encoder,
                           embedding_size=embedding_size_encoder,
                           lstm_size=units_encoder,
                           input_length=input_length_encoder)

    #Create decoder object
    self.decoder = Decoder(out_vocab_size=vocab_size_decoder,
                       embedding_dim=embedding_size_decoder,
                       input_length=input_length_decoder,
                       dec_units=units_decoder,
                       score_fun='dot',
                       att_units=units_decoder)





  def call(self,data):
    #Intialize encoder states, Pass the encoder_sequence to the embedding layer
    # Decoder initial states are encoder final states, Initialize it accordingly
    # Pass the decoder sequence,encoder_output,decoder states to Decoder
    # return the decoder output


    input,output = data[0], data[1]

    enc_state_h, enc_state_c = self.encoder.initialize_states(1024)

    encoder_initial_states = [enc_state_h, enc_state_c]

    encoder_output, encoder_h, encoder_c = self.encoder(input,␣
↪encoder_initial_states)

    decoder_output = self.decoder(output, encoder_output, encoder_h, encoder_c)

    return decoder_output
```

18 Task - 2 Custom loss function

[ ]:
```
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True,␣
↪reduction='none')

def custom_lossfunction(targets,logits):
```

```
    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_)

    # Custom loss function that will not consider the loss for padded zeros.
    # Refer https://www.tensorflow.org/tutorials/text/
↪nmt_with_attention#define_the_optimizer_and_the_loss_function
```

19 Task - 2 Model Train (Score DOT)

[ ]:
```
model = Encoder_decoder()
model.compile(loss=custom_lossfunction, optimizer=tf.keras.optimizers.
↪Adam(learning_rate=0.001))
```

[ ]:
```
class Custom_callback(tf.keras.callbacks.Callback):

    def on_epoch_end(self, epoch, logs=None):
          keys = list(logs.keys())
          # print("End epoch {} of training; got log keys: {}".format(epoch,␣
↪keys))
          for i, row in validation.iterrows():

            italian_sentence = row['italian']

            predicted_eng_sentence = predict(italian_sentence, model, 0)

            original_eng_sentence = re.sub("<start>", "", row['english_inp'])

            print("original {}".format(original_eng_sentence))
            print("predicted {}".format(predicted_eng_sentence))
            break
```

[ ]:
```
import os
os.environ["TF_FORCE_GPU_ALLOW_GROWTH"]="true"

custom_callback = Custom_callback()

log_dir="logs/fit/model_enc_dec_dot"
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir,
                                                       histogram_freq=1,
                                                       write_graph=True,
                                                       write_grads=True)
```

```python
es_callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=2,␣
↪verbose=1)


train_steps=train.shape[0]//1024
valid_steps=validation.shape[0]//1024

model.fit_generator(train_dataloader, verbose=1, epochs=25,␣
↪steps_per_epoch=train_steps, callbacks=[tensorboard_callback, es_callback,␣
↪custom_callback])
```

```
WARNING:tensorflow:`write_grads` will be ignored in TensorFlow 2.0 for the
`TensorBoard` Callback.
Epoch 1/25
```

```
/usr/local/lib/python3.6/dist-
packages/tensorflow/python/framework/indexed_slices.py:432: UserWarning:
Converting sparse IndexedSlices to a dense Tensor of unknown shape. This may
consume a large amount of memory.
  "Converting sparse IndexedSlices to a dense Tensor of unknown shape. "
```

```
266/266 [==============================] - ETA: 0s - loss: 1.6249original  tom
got in the car
predicted  i am a good in the car
266/266 [==============================] - 236s 886ms/step - loss: 1.6249
Epoch 2/25
266/266 [==============================] - ETA: 0s - loss: 1.1434original  tom
got in the car
predicted  tom is the same car
266/266 [==============================] - 236s 888ms/step - loss: 1.1434
Epoch 3/25
266/266 [==============================] - ETA: 0s - loss: 0.8256original  tom
got in the car
predicted  tom went to the car
266/266 [==============================] - 238s 893ms/step - loss: 0.8256
Epoch 4/25
266/266 [==============================] - ETA: 0s - loss: 0.5903original  tom
got in the car
predicted  tom went to the car
266/266 [==============================] - 237s 889ms/step - loss: 0.5903
Epoch 5/25
266/266 [==============================] - ETA: 0s - loss: 0.4303original  tom
got in the car
predicted  tom went into the car
266/266 [==============================] - 237s 890ms/step - loss: 0.4303
Epoch 6/25
266/266 [==============================] - ETA: 0s - loss: 0.3207original  tom
got in the car
```

```
predicted  tom went into the car
266/266 [==============================] - 237s 889ms/step - loss: 0.3207
Epoch 7/25
266/266 [==============================] - ETA: 0s - loss: 0.2489original  tom
got in the car
predicted  tom went into the car
266/266 [==============================] - 236s 889ms/step - loss: 0.2489
Epoch 8/25
266/266 [==============================] - ETA: 0s - loss: 0.2009original  tom
got in the car
predicted  tom went into the car
266/266 [==============================] - 237s 892ms/step - loss: 0.2009
Epoch 9/25
266/266 [==============================] - ETA: 0s - loss: 0.1660original  tom
got in the car
predicted  tom went into the car
266/266 [==============================] - 237s 889ms/step - loss: 0.1660
Epoch 10/25
266/266 [==============================] - ETA: 0s - loss: 0.1400original  tom
got in the car
predicted  tom went into the car
266/266 [==============================] - 236s 887ms/step - loss: 0.1400
Epoch 11/25
266/266 [==============================] - ETA: 0s - loss: 0.1194original  tom
got in the car
predicted  tom went into the car
266/266 [==============================] - 234s 882ms/step - loss: 0.1194
Epoch 12/25
266/266 [==============================] - ETA: 0s - loss: 0.1038original  tom
got in the car
predicted  tom went into the car
266/266 [==============================] - 235s 882ms/step - loss: 0.1038
Epoch 13/25
266/266 [==============================] - ETA: 0s - loss: 0.0905original  tom
got in the car
predicted  tom went into the car
266/266 [==============================] - 237s 889ms/step - loss: 0.0905
Epoch 14/25
266/266 [==============================] - ETA: 0s - loss: 0.0796original  tom
got in the car
predicted  tom went into a car
266/266 [==============================] - 238s 895ms/step - loss: 0.0796
Epoch 15/25
266/266 [==============================] - ETA: 0s - loss: 0.0707original  tom
got in the car
predicted  tom went into the car
266/266 [==============================] - 236s 889ms/step - loss: 0.0707
Epoch 16/25
```

```
266/266 [==============================] - ETA: 0s - loss: 0.0626original  tom
got in the car
predicted  tom went into the car
266/266 [==============================] - 237s 890ms/step - loss: 0.0626
Epoch 17/25
266/266 [==============================] - ETA: 0s - loss: 0.0561original  tom
got in the car
predicted  tom went into the car
266/266 [==============================] - 236s 889ms/step - loss: 0.0561
Epoch 18/25
266/266 [==============================] - ETA: 0s - loss: 0.0512original  tom
got in the car
predicted  tom went in the car
266/266 [==============================] - 237s 891ms/step - loss: 0.0512
Epoch 19/25
266/266 [==============================] - ETA: 0s - loss: 0.0451original  tom
got in the car
predicted  tom went into the car
266/266 [==============================] - 236s 885ms/step - loss: 0.0451
Epoch 20/25
266/266 [==============================] - ETA: 0s - loss: 0.0407original  tom
got in the car
predicted  tom went in a car
266/266 [==============================] - 235s 882ms/step - loss: 0.0407
Epoch 21/25
266/266 [==============================] - ETA: 0s - loss: 0.0371original  tom
got in the car
predicted  tom went into the car
266/266 [==============================] - 235s 884ms/step - loss: 0.0371
Epoch 22/25
266/266 [==============================] - ETA: 0s - loss: 0.0337original  tom
got in the car
predicted  tom got in the car
266/266 [==============================] - 236s 886ms/step - loss: 0.0337
Epoch 23/25
266/266 [==============================] - ETA: 0s - loss: 0.0315original  tom
got in the car
predicted  tom got into the car
266/266 [==============================] - 235s 884ms/step - loss: 0.0315
Epoch 24/25
266/266 [==============================] - ETA: 0s - loss: 0.0288original  tom
got in the car
predicted  tom went into the car
266/266 [==============================] - 236s 886ms/step - loss: 0.0288
Epoch 25/25
266/266 [==============================] - ETA: 0s - loss: 0.0271original  tom
got in the car
predicted  tom went by car
```

```
266/266 [==============================] - 236s 886ms/step - loss: 0.0271
```

```
[ ]: <tensorflow.python.keras.callbacks.History at 0x7f182c87f5c0>
```

```
[ ]: model.summary()
```

```
Model: "encoder_decoder_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
encoder_6 (Encoder)          multiple                  13268600
_________________________________________________________________
decoder_8 (Decoder)          multiple                  26590779
=================================================================
Total params: 39,859,379
Trainable params: 39,859,379
Non-trainable params: 0
_________________________________________________________________
```

```
[ ]: %tensorboard --logdir logs/fit
```

20 Task - 2 Inference (Dot Scoring)

Plot attention weights

```
[ ]: def plot_attention(attention, sentence, predicted_sentence):
       fig = plt.figure(figsize=(10,10))
       ax = fig.add_subplot(1, 1, 1)
       ax.matshow(attention, cmap='viridis')

       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))

       plt.show()
```

Predict the sentence translation

```
[ ]: def predict(input_sentence, model, count):

       '''
       A. Given input sentence, convert the sentence into integers using tokenizer␣
     ↪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 final states␣
↪as input_states to onestepdecoder.
 D. till we reach max_length of decoder or till the model predicted word <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 store it in␣
↪a string.
 E. Call plot_attention(#params)
 F. Return the predicted sentence
 '''

 # attention
 attention_plot = np.zeros((20, 20))

 # tokenizing sentence
 tokens = np.array(tknizer_ita.texts_to_sequences([input_sentence]))
 padded_tokens = pad_sequences(tokens, maxlen=20, padding='post')

 enc_hidden_h, enc_hidden_c = model.encoder.initialize_states(batch_size=1)
 enc_intial_states = [enc_hidden_h, enc_hidden_c]

 enc_output, enc_state_h, enc_state_c = model.encoder(padded_tokens,␣
↪enc_intial_states)

 dec_hidden_h = enc_state_h
 dec_hidden_c = enc_state_c

 sent = ''

 dec_input = tf.expand_dims([tknizer_eng.word_index['<start>']] * 1, 1)


 for i in range(20):
   predictions, dec_hidden_h, dec_hidden_c, attention_weights, _ = model.
↪decoder.one_step_decoder(dec_input,
                                                                                    ␣
↪   enc_output,
                                                                                    ␣
↪   dec_hidden_h,
                                                                                    ␣
↪   dec_hidden_c)

   # storing the attention weights to plot later on
```

```
    attention_weights = tf.reshape(attention_weights, (-1, ))
    attention_plot[i] = attention_weights.numpy()

    infe_output=np.argmax(predictions,-1)

    word_index = infe_output[0]

    # if word index is <str>, continue
    if word_index == 0:
      dec_input = np.reshape(np.argmax(infe_output), (1, 1))
      continue

    # if word index is <end>, stop predicting
    if eng_index_word_dict[word_index] == "<end>":
      return sent

    # append predicted word to sentence
    sent=sent+' '+eng_index_word_dict[int(word_index)]

    # reshape the predicted word index, for feeding next decoder time step
    dec_input = np.reshape(int(word_index), (1, 1))

  return sent


count = 0
bleu_score = []

for i, row in validation.iterrows():

  if count == 1000:
    break

  italian_sentence = row['italian']
  original_eng_sentence = re.sub("<start>", "", row['english_inp'])

  predicted_eng_sentence = predict(italian_sentence, model, count)

  original_eng_sentence = re.sub("<start>", "", row['english_inp'])
  print("orig {}".format(original_eng_sentence))
  print("pred {}".format(predicted_eng_sentence))
  score = bluescore(original_eng_sentence, predicted_eng_sentence)
  bleu_score.append(score)
  count += 1
```

21 DOT scoring attention BLEU score

[ ]:
```
enc_dec_dot_bleu = np.mean(bleu_score)
```

[ ]:
```
print("DOT BLEU SCORE {}".format(enc_dec_dot_bleu))
```

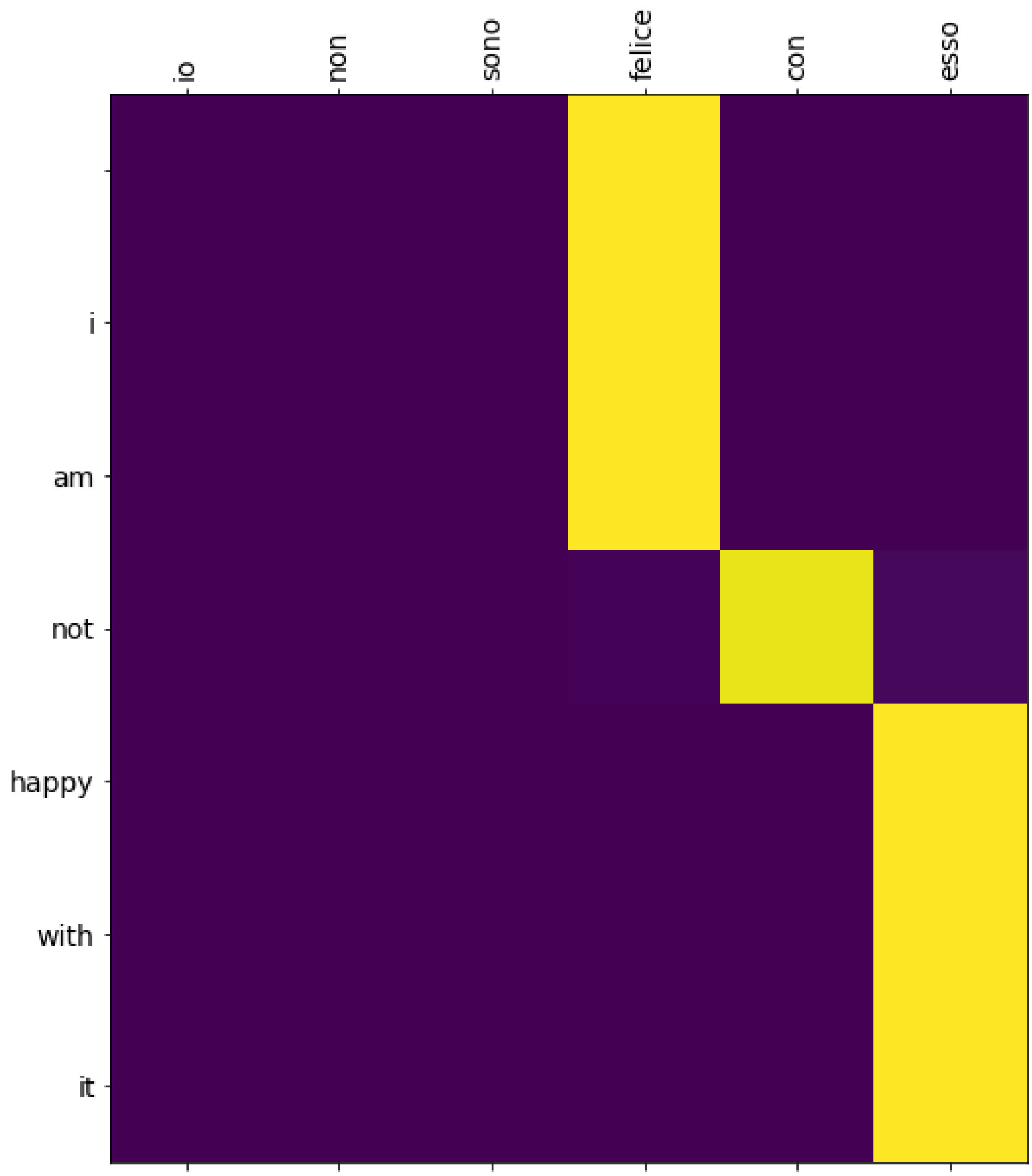
```
DOT BLEU SCORE 0.8590928431443758
```

22 DOT scoring attention - attention plot

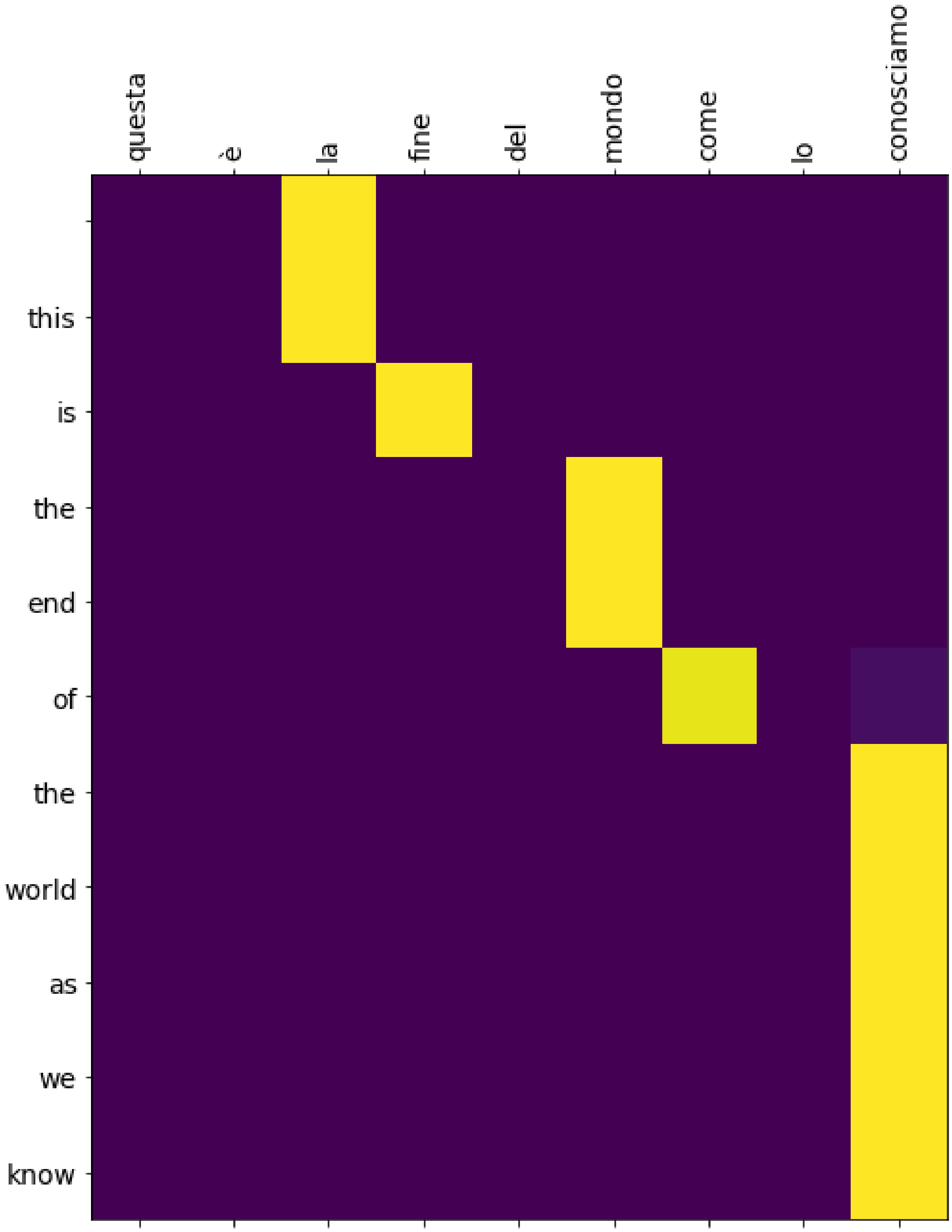
[ ]:
```
def predict_with_attention_plot(input_sentence, model, count):

    '''
    A. Given input sentence, convert the sentence into integers using tokenizer␣
 ↪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 final states␣
 ↪as input_states to onestepdecoder.
    D. till we reach max_length of decoder or till the model predicted word <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 store it in␣
 ↪a string.
    E. Call plot_attention(#params)
    F. Return the predicted sentence
    '''

    # attention
    attention_plot = np.zeros((20, 20))

    # tokenizing sentence
    tokens = np.array(tknizer_ita.texts_to_sequences([input_sentence]))
    padded_tokens = pad_sequences(tokens, maxlen=20, padding='post')

    enc_hidden_h, enc_hidden_c = model.encoder.initialize_states(batch_size=1)
    enc_intial_states = [enc_hidden_h, enc_hidden_c]

    enc_output, enc_state_h, enc_state_c = model.encoder(padded_tokens,␣
 ↪enc_intial_states)

    dec_hidden_h = enc_state_h
    dec_hidden_c = enc_state_c

    sent = ''
```

```
dec_input = tf.expand_dims([tknizer_eng.word_index['<start>']] * 1, 1)


for i in range(20):
  predictions, dec_hidden_h, dec_hidden_c, attention_weights, _ = model.
↪decoder.one_step_decoder(dec_input,
                                                                        ␣
↪   enc_output,
                                                                        ␣
↪   dec_hidden_h,
                                                                        ␣
↪   dec_hidden_c)

  # storing the attention weights to plot later on
  attention_weights = tf.reshape(attention_weights, (-1, ))
  attention_plot[i] = attention_weights.numpy()

  infe_output=np.argmax(predictions,-1)

  word_index = infe_output[0]

  # if word index is <str>, continue
  if word_index == 0:
    dec_input = np.reshape(np.argmax(infe_output), (1, 1))
    continue

  # if word index is <end>, stop predicting
  if eng_index_word_dict[word_index] == "<end>":

    attention_plot = attention_plot[:len(sent.split(' ')), :
↪len(input_sentence.split(' '))]
    plot_attention(attention_plot, input_sentence.split(' '), sent.split(' '))

    return sent

  # append predicted word to sentence
  sent=sent+' '+eng_index_word_dict[int(word_index)]

  # reshape the predicted word index, for feeding next decoder time step
  dec_input = np.reshape(int(word_index), (1, 1))

return sent
```

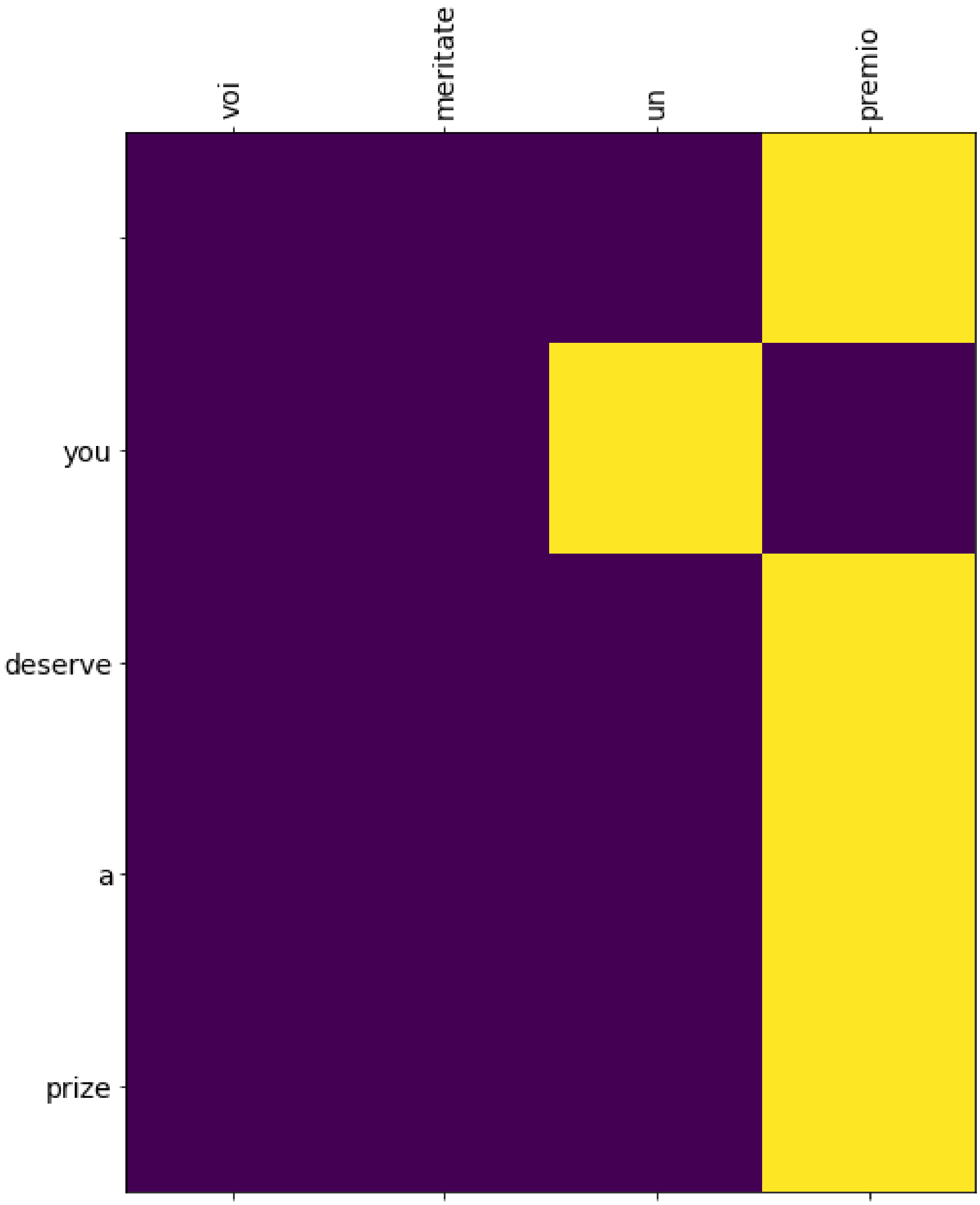
[ ]:
```
italian_sentence = np.array(validation['italian'])[1]
predicted_eng_sentence = predict_with_attention_plot(italian_sentence, model,␣
↪count)
```



[ ]:
```
italian_sentence = np.array(validation['italian'])[2]
predicted_eng_sentence = predict_with_attention_plot(italian_sentence, model,␣
 ↪count)
```




```
[ ]: italian_sentence = np.array(validation['italian'])[3]
     predicted_eng_sentence = predict_with_attention_plot(italian_sentence, model,
      ↪count)
```


voi
meritate
un
premio
you
deserve
a
prize

23 Task - 2 Encoder Decoder model (Concat Scoring)

[ ]:
```
class Encoder_decoder(tf.keras.Model):
  def __init__(self):
    super().__init__()
    #Intialize objects from encoder decoder

    #Create encoder object
    self.encoder = Encoder(inp_vocab_size=vocab_size_encoder,
                            embedding_size=embedding_size_encoder,
                            lstm_size=units_encoder,
                            input_length=input_length_encoder)

    #Create decoder object
    self.decoder = Decoder(out_vocab_size=vocab_size_decoder,
                        embedding_dim=embedding_size_decoder,
                        input_length=input_length_decoder,
                        dec_units=units_decoder,
                        score_fun='concat',
                        att_units=units_decoder)





  def call(self,data):
    #Intialize encoder states, Pass the encoder_sequence to the embedding layer
    # Decoder initial states are encoder final states, Initialize it accordingly
    # Pass the decoder sequence,encoder_output,decoder states to Decoder
    # return the decoder output


    input,output = data[0], data[1]

    enc_state_h, enc_state_c = self.encoder.initialize_states(1024)

    encoder_initial_states = [enc_state_h, enc_state_c]

    encoder_output, encoder_h, encoder_c = self.encoder(input,␣
↪encoder_initial_states)

    decoder_output = self.decoder(output, encoder_output, encoder_h, encoder_c)

    return decoder_output
```

24 Task - 2 Model Train (Concat Scoring)

[ ]:
```
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True,␣
↪reduction='none')

def custom_lossfunction(targets,logits):
  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_)

  # Custom loss function that will not consider the loss for padded zeros.
  # Refer https://www.tensorflow.org/tutorials/text/
↪nmt_with_attention#define_the_optimizer_and_the_loss_function
```

[ ]:
```
model = Encoder_decoder()
model.compile(loss=custom_lossfunction, optimizer=tf.keras.optimizers.
↪Adam(learning_rate=0.001))


import os
os.environ["TF_FORCE_GPU_ALLOW_GROWTH"]="true"

custom_callback = Custom_callback()

log_dir="logs/fit/model_enc_dec_concat"
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir,
                                                      histogram_freq=1,
                                                      write_graph=True,
                                                      write_grads=True)




train_steps=train.shape[0]//1024
valid_steps=validation.shape[0]//1024

model.fit_generator(train_dataloader, verbose=1, epochs=25,␣
↪steps_per_epoch=train_steps, callbacks=[tensorboard_callback,␣
↪custom_callback])
```

```
WARNING:tensorflow:`write_grads` will be ignored in TensorFlow 2.0 for the
`TensorBoard` Callback.
Epoch 1/25
```

```
/usr/local/lib/python3.6/dist-
```

```
packages/tensorflow/python/framework/indexed_slices.py:432: UserWarning:
Converting sparse IndexedSlices to a dense Tensor of unknown shape. This may
consume a large amount of memory.
  "Converting sparse IndexedSlices to a dense Tensor of unknown shape. "

  2/266 […] - ETA: 3:22 - loss:
3.2061WARNING:tensorflow:Callbacks method `on_train_batch_end` is slow compared
to the batch time (batch time: 0.5714s vs `on_train_batch_end` time: 0.9596s).
Check your callbacks.
266/266 [==============================] - ETA: 0s - loss: 1.8818original  they
are playing chess
predicted  i you you
266/266 [==============================] - 140s 525ms/step - loss: 1.8818
Epoch 2/25
266/266 [==============================] - ETA: 0s - loss: 1.5204original  they
are playing chess
predicted  i am a lot
266/266 [==============================] - 138s 520ms/step - loss: 1.5204
Epoch 3/25
266/266 [==============================] - ETA: 0s - loss: 1.2566original  they
are playing chess
predicted  you are a lot of
266/266 [==============================] - 139s 522ms/step - loss: 1.2566
Epoch 4/25
266/266 [==============================] - ETA: 0s - loss: 1.0625original  they
are playing chess
predicted  they are here
266/266 [==============================] - 139s 522ms/step - loss: 1.0625
Epoch 5/25
266/266 [==============================] - ETA: 0s - loss: 0.8967original  they
are playing chess
predicted  they are going to get up
266/266 [==============================] - 139s 521ms/step - loss: 0.8967
Epoch 6/25
266/266 [==============================] - ETA: 0s - loss: 0.7499original  they
are playing chess
predicted  they are playing golf
266/266 [==============================] - 139s 521ms/step - loss: 0.7499
Epoch 7/25
266/266 [==============================] - ETA: 0s - loss: 0.6282original  they
are playing chess
predicted  they are playing golf
266/266 [==============================] - 139s 521ms/step - loss: 0.6282
Epoch 8/25
266/266 [==============================] - ETA: 0s - loss: 0.5316original  they
are playing chess
predicted  they are playing golf
266/266 [==============================] - 138s 518ms/step - loss: 0.5316
```

```
Epoch 9/25
266/266 [==============================] - ETA: 0s - loss: 0.4553original  they
are playing chess
predicted  they are playing golf
266/266 [==============================] - 138s 519ms/step - loss: 0.4553
Epoch 10/25
266/266 [==============================] - ETA: 0s - loss: 0.3954original  they
are playing chess
predicted  they are playing golf
266/266 [==============================] - 137s 516ms/step - loss: 0.3954
Epoch 11/25
266/266 [==============================] - ETA: 0s - loss: 0.3478original  they
are playing chess
predicted  they are playing golf
266/266 [==============================] - 137s 517ms/step - loss: 0.3478
Epoch 12/25
266/266 [==============================] - ETA: 0s - loss: 0.3081original  they
are playing chess
predicted  they are playing chess
266/266 [==============================] - 138s 519ms/step - loss: 0.3081
Epoch 13/25
266/266 [==============================] - ETA: 0s - loss: 0.2752original  they
are playing chess
predicted  they are playing at chess
266/266 [==============================] - 138s 519ms/step - loss: 0.2752
Epoch 14/25
266/266 [==============================] - ETA: 0s - loss: 0.2478original  they
are playing chess
predicted  they are playing chess
266/266 [==============================] - 137s 515ms/step - loss: 0.2478
Epoch 15/25
266/266 [==============================] - ETA: 0s - loss: 0.2249original  they
are playing chess
predicted  they are playing chess
266/266 [==============================] - 138s 519ms/step - loss: 0.2249
Epoch 16/25
266/266 [==============================] - ETA: 0s - loss: 0.2053original  they
are playing chess
predicted  they are playing chess
266/266 [==============================] - 139s 521ms/step - loss: 0.2053
Epoch 17/25
266/266 [==============================] - ETA: 0s - loss: 0.1887original  they
are playing chess
predicted  they are playing something
266/266 [==============================] - 138s 519ms/step - loss: 0.1887
Epoch 18/25
266/266 [==============================] - ETA: 0s - loss: 0.1743original  they
are playing chess
```

```
predicted  they are playing anything
266/266 [==============================] - 138s 520ms/step - loss: 0.1743
Epoch 19/25
266/266 [==============================] - ETA: 0s - loss: 0.1616original  they
are playing chess
predicted  they are playing
266/266 [==============================] - 138s 520ms/step - loss: 0.1616
Epoch 20/25
266/266 [==============================] - ETA: 0s - loss: 0.1508original  they
are playing chess
predicted  they are playing something
266/266 [==============================] - 139s 522ms/step - loss: 0.1508
Epoch 21/25
266/266 [==============================] - ETA: 0s - loss: 0.1405original  they
are playing chess
predicted  they are playing
266/266 [==============================] - 140s 528ms/step - loss: 0.1405
Epoch 22/25
266/266 [==============================] - ETA: 0s - loss: 0.1308original  they
are playing chess
predicted  they are playing anything
266/266 [==============================] - 138s 520ms/step - loss: 0.1308
Epoch 23/25
266/266 [==============================] - ETA: 0s - loss: 0.1231original  they
are playing chess
predicted  they are playing
266/266 [==============================] - 140s 527ms/step - loss: 0.1231
Epoch 24/25
266/266 [==============================] - ETA: 0s - loss: 0.1166original  they
are playing chess
predicted  they are playing anything
266/266 [==============================] - 137s 517ms/step - loss: 0.1166
Epoch 25/25
266/266 [==============================] - ETA: 0s - loss: 0.1091original  they
are playing chess
predicted  they are playing
266/266 [==============================] - 140s 526ms/step - loss: 0.1091
```

[ ]: `<tensorflow.python.keras.callbacks.History at 0x7f15fc694470>`

[ ]:

```
model.summary()
```

```
Model: "encoder_decoder_17"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
encoder_20 (Encoder)         multiple                  8056548
_________________________________________________________________
```

```
decoder_26 (Decoder)         multiple                  5808250
=================================================================
Total params: 13,864,798
Trainable params: 13,864,798
Non-trainable params: 0
_________________________________________________________________
```

[ ]:
```
%tensorboard --logdir logs/fit
```

25 Task - 2 Inference (Concat Scoring)

[ ]:
```
count = 0
bleu_score = []
for i, row in validation.iterrows():

  if count == 1000:
    break

  italian_sentence = row['italian']
  predicted_eng_sentence = predict(italian_sentence, model, count)
  original_eng_sentence = re.sub("<start>", "", row['english_inp'])

  score = bluescore(original_eng_sentence, predicted_eng_sentence)

  bleu_score.append(score)
  count += 1
```

```
/usr/local/lib/python3.6/dist-packages/nltk/translate/bleu_score.py:490:
UserWarning:
Corpus/Sentence contains 0 counts of 2-gram overlaps.
BLEU scores might be undesirable; use SmoothingFunction().
  warnings.warn(_msg)
```

26 Task-2 (BLEU score concat scoring attention)

[ ]:
```
enc_dec_attn_concat = np.mean(bleu_score)
```

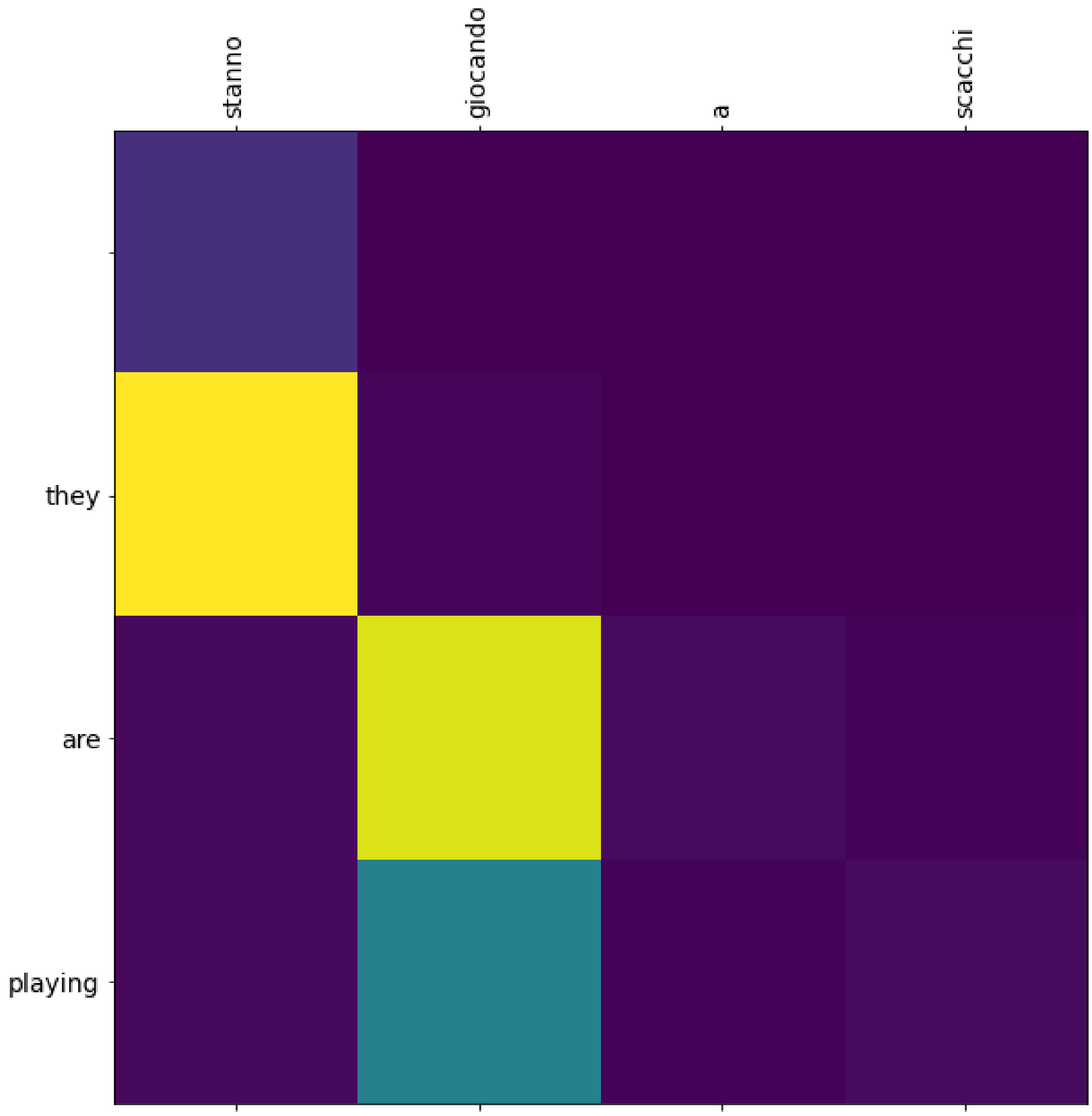
[ ]:
```
print("CONCAT BLEU SCORE {}".format(enc_dec_attn_concat))
```

```
CONCAT BLEU SCORE 0.6693687223844546
```

27 Task - 2 (Concat Scoring attention) - attention plot

```
[ ]: def predict_with_attention_plot(input_sentence, model, count):

      '''
      A. Given input sentence, convert the sentence into integers using tokenizer␣
    ↪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 final states␣
    ↪as input_states to onestepdecoder.
      D. till we reach max_length of decoder or till the model predicted word <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 store it in␣
    ↪a string.
      E. Call plot_attention(#params)
      F. Return the predicted sentence
      '''

      # attention
      attention_plot = np.zeros((20, 20))

      # tokenizing sentence
      tokens = np.array(tknizer_ita.texts_to_sequences([input_sentence]))
      padded_tokens = pad_sequences(tokens, maxlen=20, padding='post')

      enc_hidden_h, enc_hidden_c = model.encoder.initialize_states(batch_size=1)
      enc_intial_states = [enc_hidden_h, enc_hidden_c]

      enc_output, enc_state_h, enc_state_c = model.encoder(padded_tokens,␣
    ↪enc_intial_states)

      dec_hidden_h = enc_state_h
      dec_hidden_c = enc_state_c

      sent = ''

      dec_input = tf.expand_dims([tknizer_eng.word_index['<start>']] * 1, 1)


      for i in range(20):
        predictions, dec_hidden_h, dec_hidden_c, attention_weights, _ = model.
    ↪decoder.one_step_decoder(dec_input,
                                                                                   ␣
    ↪   enc_output,
```

```
                                                                         ␣
↪   dec_hidden_h,
                                                                         ␣
↪   dec_hidden_c)

    # storing the attention weights to plot later on
    attention_weights = tf.reshape(attention_weights, (-1, ))
    attention_plot[i] = attention_weights.numpy()

    infe_output=np.argmax(predictions,-1)

    word_index = infe_output[0]

    # if word index is <str>, continue
    if word_index == 0:
      dec_input = np.reshape(np.argmax(infe_output), (1, 1))
      continue

    # if word index is <end>, stop predicting
    if eng_index_word_dict[word_index] == "<end>":

      attention_plot = attention_plot[:len(sent.split(' ')), :
↪len(input_sentence.split(' '))]
      plot_attention(attention_plot, input_sentence.split(' '), sent.split(' '))
      return sent

    # append predicted word to sentence
    sent=sent+' '+eng_index_word_dict[int(word_index)]

    # reshape the predicted word index, for feeding next decoder time step
    dec_input = np.reshape(int(word_index), (1, 1))

  return sent



for i, row in validation.iterrows():

  italian_sentence = row['italian']
  predicted_eng_sentence = predict_with_attention_plot(italian_sentence, model,␣
↪count)

  break
```



28 Task - 2 Encoder Decoder model (General Scoring)

[ ]:
```
class Encoder_decoder(tf.keras.Model):
  def __init__(self):
    super().__init__()
    #Intialize objects from encoder decoder

    #Create encoder object
    self.encoder = Encoder(inp_vocab_size=vocab_size_encoder,
                           embedding_size=embedding_size_encoder,
                           lstm_size=units_encoder,
```

```
                               input_length=input_length_encoder)

    #Create decoder object
    self.decoder = Decoder(out_vocab_size=vocab_size_decoder,
                           embedding_dim=embedding_size_decoder,
                           input_length=input_length_decoder,
                           dec_units=units_decoder,
                           score_fun='general',
                           att_units=units_decoder)





  def call(self,data):
    #Intialize encoder states, Pass the encoder_sequence to the embedding layer
    # Decoder initial states are encoder final states, Initialize it accordingly
    # Pass the decoder sequence,encoder_output,decoder states to Decoder
    # return the decoder output


    input,output = data[0], data[1]

    enc_state_h, enc_state_c = self.encoder.initialize_states(1024)

    encoder_initial_states = [enc_state_h, enc_state_c]

    encoder_output, encoder_h, encoder_c = self.encoder(input,␣
↪encoder_initial_states)

    decoder_output = self.decoder(output, encoder_output, encoder_h, encoder_c)

    return decoder_output
```

29 Task - 2 Model Train (General Scoring)

[ ]:
```
model = Encoder_decoder()
model.compile(loss=custom_lossfunction, optimizer=tf.keras.optimizers.
↪Adam(learning_rate=0.001))



import os
os.environ["TF_FORCE_GPU_ALLOW_GROWTH"]="true"

custom_callback = Custom_callback()
```

```
log_dir="logs/fit/model_enc_dec_general"
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir,
                                                      histogram_freq=1,
                                                      write_graph=True,
                                                      write_grads=True)




train_steps=train.shape[0]//1024
valid_steps=validation.shape[0]//1024

model.fit_generator(train_dataloader, verbose=1, epochs=25,␣
 ↪steps_per_epoch=train_steps, callbacks=[tensorboard_callback,␣
 ↪custom_callback])
```

```
WARNING:tensorflow:`write_grads` will be ignored in TensorFlow 2.0 for the
`TensorBoard` Callback.
Epoch 1/25
```

```
/usr/local/lib/python3.6/dist-
packages/tensorflow/python/framework/indexed_slices.py:432: UserWarning:
Converting sparse IndexedSlices to a dense Tensor of unknown shape. This may
consume a large amount of memory.
  "Converting sparse IndexedSlices to a dense Tensor of unknown shape. "
```

```
266/266 [==============================] - ETA: 0s - loss: 1.9150original  they
are playing chess
predicted  i i you
266/266 [==============================] - 121s 455ms/step - loss: 1.9150
Epoch 2/25
266/266 [==============================] - ETA: 0s - loss: 1.6243original  they
are playing chess
predicted  i am a a
266/266 [==============================] - 121s 455ms/step - loss: 1.6243
Epoch 3/25
266/266 [==============================] - ETA: 0s - loss: 1.3683original  they
are playing chess
predicted  they are the lot of the
266/266 [==============================] - 121s 454ms/step - loss: 1.3683
Epoch 4/25
266/266 [==============================] - ETA: 0s - loss: 1.1650original  they
are playing chess
predicted  they are the time
266/266 [==============================] - 121s 454ms/step - loss: 1.1650
Epoch 5/25
266/266 [==============================] - ETA: 0s - loss: 1.0071original  they
are playing chess
predicted  they are in the day
```

```
266/266 [==============================] - 120s 451ms/step - loss: 1.0071
Epoch 6/25
266/266 [==============================] - ETA: 0s - loss: 0.8624original  they
are playing chess
predicted  they are in australia
266/266 [==============================] - 120s 452ms/step - loss: 0.8624
Epoch 7/25
266/266 [==============================] - ETA: 0s - loss: 0.7375original  they
are playing chess
predicted  they are in the music
266/266 [==============================] - 120s 449ms/step - loss: 0.7375
Epoch 8/25
266/266 [==============================] - ETA: 0s - loss: 0.6331original  they
are playing chess
predicted  they are playing golf
266/266 [==============================] - 120s 451ms/step - loss: 0.6331
Epoch 9/25
266/266 [==============================] - ETA: 0s - loss: 0.5468original  they
are playing chess
predicted  they are playing golf
266/266 [==============================] - 121s 454ms/step - loss: 0.5468
Epoch 10/25
266/266 [==============================] - ETA: 0s - loss: 0.4759original  they
are playing chess
predicted  they are playing golf
266/266 [==============================] - 120s 453ms/step - loss: 0.4759
Epoch 11/25
266/266 [==============================] - ETA: 0s - loss: 0.4168original  they
are playing chess
predicted  they are playing chess
266/266 [==============================] - 121s 454ms/step - loss: 0.4168
Epoch 12/25
266/266 [==============================] - ETA: 0s - loss: 0.3682original  they
are playing chess
predicted  they are playing chess
266/266 [==============================] - 121s 453ms/step - loss: 0.3682
Epoch 13/25
266/266 [==============================] - ETA: 0s - loss: 0.3279original  they
are playing chess
predicted  they are playing chess
266/266 [==============================] - 119s 447ms/step - loss: 0.3279
Epoch 14/25
266/266 [==============================] - ETA: 0s - loss: 0.2951original  they
are playing chess
predicted  they are playing chess
266/266 [==============================] - 121s 453ms/step - loss: 0.2951
Epoch 15/25
266/266 [==============================] - ETA: 0s - loss: 0.2666original  they
```

```
are playing chess
predicted  they are playing chess
266/266 [==============================] - 122s 457ms/step - loss: 0.2666
Epoch 16/25
266/266 [==============================] - ETA: 0s - loss: 0.2432original  they
are playing chess
predicted  they are playing chess
266/266 [==============================] - 120s 451ms/step - loss: 0.2432
Epoch 17/25
266/266 [==============================] - ETA: 0s - loss: 0.2228original  they
are playing chess
predicted  they are playing chess
266/266 [==============================] - 119s 447ms/step - loss: 0.2228
Epoch 18/25
266/266 [==============================] - ETA: 0s - loss: 0.2044original  they
are playing chess
predicted  they are playing chess
266/266 [==============================] - 118s 445ms/step - loss: 0.2044
Epoch 19/25
266/266 [==============================] - ETA: 0s - loss: 0.1893original  they
are playing chess
predicted  they are playing chess
266/266 [==============================] - 120s 450ms/step - loss: 0.1893
Epoch 20/25
266/266 [==============================] - ETA: 0s - loss: 0.1765original  they
are playing chess
predicted  they are playing chess
266/266 [==============================] - 122s 457ms/step - loss: 0.1765
Epoch 21/25
266/266 [==============================] - ETA: 0s - loss: 0.1648original  they
are playing chess
predicted  they are playing chess
266/266 [==============================] - 120s 451ms/step - loss: 0.1648
Epoch 22/25
266/266 [==============================] - ETA: 0s - loss: 0.1544original  they
are playing chess
predicted  they are playing chess
266/266 [==============================] - 119s 447ms/step - loss: 0.1544
Epoch 23/25
266/266 [==============================] - ETA: 0s - loss: 0.1454original  they
are playing chess
predicted  they are playing chess
266/266 [==============================] - 119s 446ms/step - loss: 0.1454
Epoch 24/25
266/266 [==============================] - ETA: 0s - loss: 0.1371original  they
are playing chess
predicted  they are playing chess
266/266 [==============================] - 120s 450ms/step - loss: 0.1371
```

```
Epoch 25/25
266/266 [==============================] - ETA: 0s - loss: 0.1295original  they
are playing chess
predicted  they are playing chess
266/266 [==============================] - 121s 454ms/step - loss: 0.1295
```

[ ]:
```
<tensorflow.python.keras.callbacks.History at 0x7f15f5042588>
```

[ ]:
```
model.summary()
```

```
Model: "encoder_decoder_18"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
encoder_21 (Encoder)         multiple                  8056548
_________________________________________________________________
decoder_27 (Decoder)         multiple                  5791609
=================================================================
Total params: 13,848,157
Trainable params: 13,848,157
Non-trainable params: 0
_________________________________________________________________
```

[ ]:
```
%tensorboard --logdir logs/fit
```

30 Task -2 Inference (Gerenal Scoring)

[ ]:
```
validation.head(1)
```

[ ]:
```
                           italian  …                     english_out
121424  stanno giocando a scacchi  …  they are playing chess <end>

[1 rows x 3 columns]
```

[ ]:
```
count = 0
bleu_score = []
for i, row in validation.iterrows():

  if count == 1000:
    break

  italian_sentence = row['italian']
  predicted_eng_sentence = predict(italian_sentence, model, 0)
  original_eng_sentence = re.sub("<start>", "", row['english_inp'])

  original_eng_sentence = re.sub("<start>", "", row['english_inp'])
```

```
    score = bluescore(original_eng_sentence, predicted_eng_sentence)

    bleu_score.append(score)
    count += 1
```

```
/usr/local/lib/python3.6/dist-packages/nltk/translate/bleu_score.py:490:
UserWarning:
Corpus/Sentence contains 0 counts of 2-gram overlaps.
BLEU scores might be undesirable; use SmoothingFunction().
  warnings.warn(_msg)
```

31 Task - 2 (General Scoring attention BLEU Score)

[ ]:
```
enc_dec_attn_general = np.mean(bleu_score)
```

[ ]:
```
print("General BLEU SCORE {}".format(enc_dec_attn_general))
```

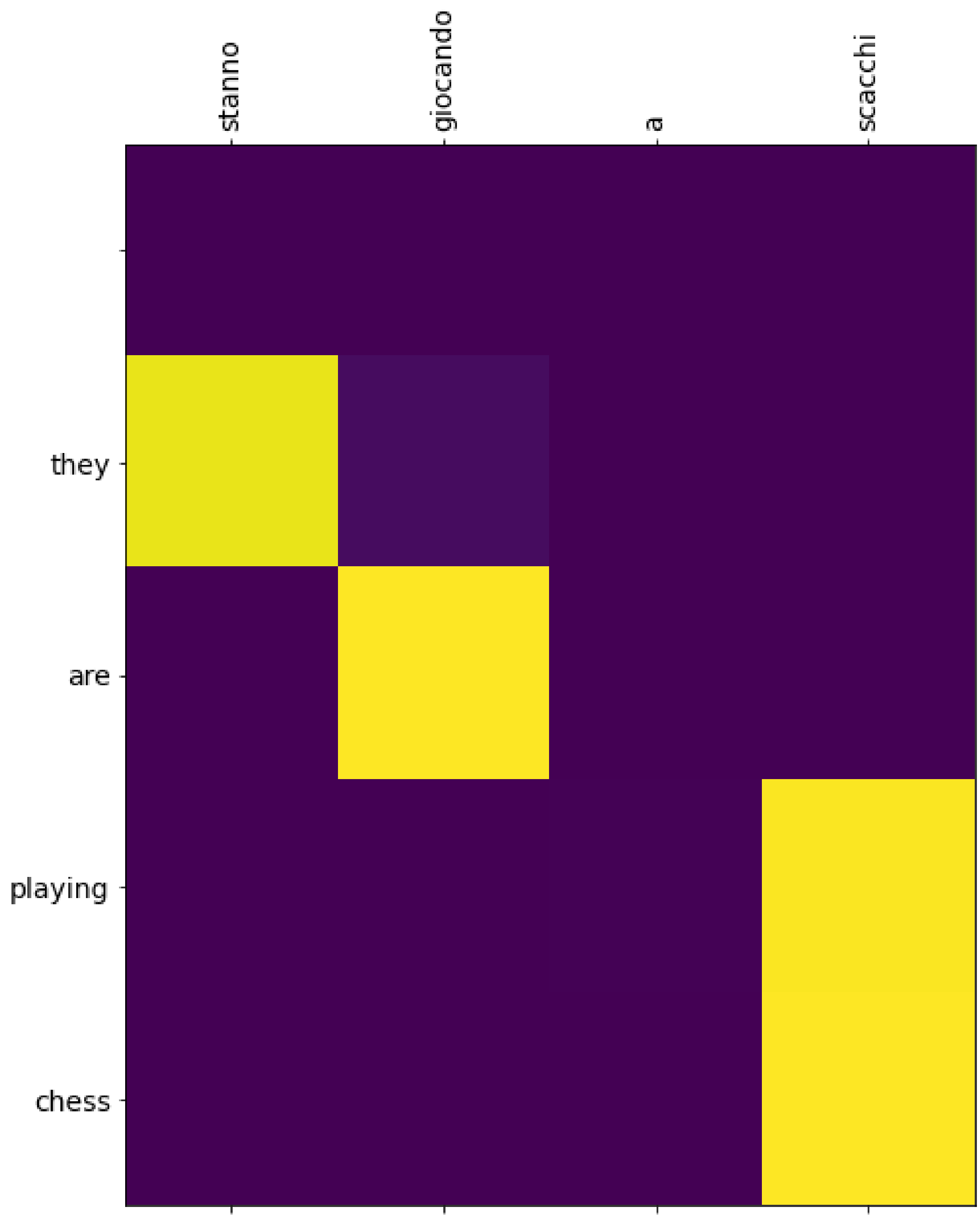
```
General BLEU SCORE 0.7061164529332072
```

32 Task - 2: Concat Scoring attention (attention plot)

[ ]:
```
def predict_with_attention_plot(input_sentence, model, count):

  '''
  A. Given input sentence, convert the sentence into integers using tokenizer␣
↪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 final states␣
↪as input_states to onestepdecoder.
  D. till we reach max_length of decoder or till the model predicted word <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 store it in␣
↪a string.
  E. Call plot_attention(#params)
  F. Return the predicted sentence
  '''

  # attention
  attention_plot = np.zeros((20, 20))

  # tokenizing sentence
  tokens = np.array(tknizer_ita.texts_to_sequences([input_sentence]))
```

```
padded_tokens = pad_sequences(tokens, maxlen=20, padding='post')

enc_hidden_h, enc_hidden_c = model.encoder.initialize_states(batch_size=1)
enc_intial_states = [enc_hidden_h, enc_hidden_c]

enc_output, enc_state_h, enc_state_c = model.encoder(padded_tokens,␣
↪enc_intial_states)

dec_hidden_h = enc_state_h
dec_hidden_c = enc_state_c

sent = ''

dec_input = tf.expand_dims([tknizer_eng.word_index['<start>']] * 1, 1)


for i in range(20):
  predictions, dec_hidden_h, dec_hidden_c, attention_weights, _ = model.
↪decoder.one_step_decoder(dec_input,
                                                                           ␣
↪   enc_output,
                                                                           ␣
↪   dec_hidden_h,
                                                                           ␣
↪   dec_hidden_c)

  # storing the attention weights to plot later on
  attention_weights = tf.reshape(attention_weights, (-1, ))
  attention_plot[i] = attention_weights.numpy()

  infe_output=np.argmax(predictions,-1)

  word_index = infe_output[0]

  # if word index is <str>, continue
  if word_index == 0:
    dec_input = np.reshape(np.argmax(infe_output), (1, 1))
    continue

  # if word index is <end>, stop predicting
  if eng_index_word_dict[word_index] == "<end>":

    attention_plot = attention_plot[:len(sent.split(' ')), :
↪len(input_sentence.split(' '))]
    plot_attention(attention_plot, input_sentence.split(' '), sent.split(' '))
    return sent
```

```
    # append predicted word to sentence
    sent=sent+' '+eng_index_word_dict[int(word_index)]

    # reshape the predicted word index, for feeding next decoder time step
    dec_input = np.reshape(int(word_index), (1, 1))

  return sent



for i, row in validation.iterrows():

  italian_sentence = row['italian']
  predicted_eng_sentence = predict_with_attention_plot(italian_sentence, model,␣
 ↪count)

  break
```



33 BLEU Score Summary

```
[ ]: from prettytable import PrettyTable
```

```
[ ]: myTable = PrettyTable(["Scoring", "BLEU Score"])

     myTable.add_row(["without attention",  "0.7427681544632555"])
     myTable.add_row(["DOT",                enc_dec_dot_bleu])
     myTable.add_row(["General",            enc_dec_attn_general])
     myTable.add_row(["Concat",             enc_dec_attn_concat])

     print(myTable)
```

```
+-------------------+--------------------+
|      Scoring      |     BLEU Score     |
+-------------------+--------------------+
| without attention | 0.7427681544632555 |
|        DOT        | 0.8590928431443758 |
|      General      | 0.7061164529332072 |
|       Concat      | 0.6693687223844546 |
+-------------------+--------------------+
```

34 Procedure

```
[ ]: """
     ---------------------------
     SIMPLE ENCODER DECODER MODEL
     ---------------------------

     Trainning
     =========

     1. Encoder
     ----------

     a. Preprocessed text data is passed to embedding layer.
     b. Then embeding output is passed thrrough lstm layer.
     c. To encoder we pass tokens, sequentially, i.e in each time step lstm layer␣
      ↪will receive different tokens.
     d. we start with <start> token, then all tokens are passed through encoder lstm␣
      ↪in each time step.
     e. the output of lstm --> states(hidden_state, cell_state)


     2. Decoder
     ----------
     a. Here we will teacher force each target tokens, to decoder in each time step.
     b. <TARGET TOKEN> is passed to embedding layer.
     c. ENCODER LAST STATES, embding_output are fed to decoder 1ST TIME STEP LSTM␣
      ↪LAYER.
```

```
d. ON each output we will calculate loss.
e. then the last time step deocder states are passed to next time step lstm␣
↪layer along with next target token.


INFERENCE
---------
a. During prediction, we will feed the input tokens to encoder.
b. IN simple encoder_decoder model, we will only use last time step states.
c. <START> token is passed to decoder embedding layer, along with encoder last␣
↪time step STATES.
d. Then decoder will convert start predicting, giving prediction token index.
e. IN text time step, we will use last time step predicted token and states to␣
↪predict next time step.
d. Decoder will keep predicting, untill <END> token is predicted.
e. AT each time step decoder, returns index of predicted target word.
f. return sentence, that we have joined.

"""
```

[ ]:

```
"""
ENCODER_DECODER WITH ATTENTION MECHANISM.
----------------------------------------


TRAINNING
--------.
a. ITALIAN TOKENS are passed to encoder.
b. Here we are considering output of each time step output and last state for␣
↪decoder to feed.
c. After feeding all input tokens to ENCODER, we will keep output and states of␣
↪EACH TIME STEP.
d. WITH ENCODER EACH STEP OUTPUT and LAST HIDDEN STATE DECODER () we will␣
↪calculate weight for encoder output with help of ATTENTION mechanism.
e. we will multiply weight with encoder output, which gives us CONTEXT VECTOR.
f. The ENG token passed through decoder embedding layer.
g. WE will concat decoder embd output and context vector and pass to decoder␣
↪LSTM layer.
h. LSTM will give STATES and PREDICTED ENG WORD, WE will calculate loss.
d. Then we will repeat same procedure untill, <END> token is teacher forced,␣
↪for every time step.

INFERENCE
--------
a. ITANLIAN TOKENS passed to encoder layer.
b. encoder gives --> encoder output of each time step, last state.
```

```
c. ATTENTION INPUT --> [encoder_output_for_eachtimestep, decoder last hidden␣
↪state] (for first time encoder states are passed to decoder.)
d. ATTENTION OUTUT --> [contect_vector]
e. FOR 1st time we will feed <START> token to decoder layer (at T=0)
f. then deocer will take weighted_encoder_output, DECODER INPUT embedding.
g. DECODER OUTPUT (T=t) --> predicted_eng_word, states
h. deocder will keep predicting untill <END> token is predicted or max_seq␣
↪length is achived.


"""
```

35 OBSERVATION

[ ]:
```
"""
1. dot score is performing better than concat and general scoring.

"""
```