This model will take the input in a normal manner and also return results in a left to right manner.

In [7]:

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
```

In [2]:

```
import matplotlib.pyplot as plt
%matplotlib inline
# import seaborn as sns
import pandas as pd
import re
import tensorflow as tf
from tensorflow.keras.layers import Embedding, LSTM, Dense
from tensorflow.keras.models import Model
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import numpy as np
import seaborn as sns
import pickle
```

In [3]:

```
[train,test, validation]=pickle.load(open('main_data_2.pkl','rb'))
```

In [4]:

```
[vocab_size_correct,vocab_size_incorrect,correct_tk,incorrect_tk]=pickle.load(open('toker
```

In [5]:

```
vocab_size_correct=max(correct_tk.word_index.values())
print(vocab_size_correct)
vocab_size_incorrect=max(incorrect_tk.word_index.values())
print(vocab_size_incorrect)
```

```
40176
52192
```

Encoder

Typesetting math: 100%

In [6]:

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

    def __init__(self,inp_vocab_size,embedding_size,lstm_size,input_length):
        super().__init__()
        #Initialize Embedding layer
        #Intialize Encoder LSTM layer
        self.embedding_layer=Embedding(input_dim=inp_vocab_size,output_dim=embedding_size
        self.lstm_layer=LSTM(lstm_size, return_sequences=True, return_state=True)
        self.lstm_size=lstm_size

    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 ou
        returns -- All encoder_outputs, last time steps hidden and cell state
        '''
        input_1=self.embedding_layer(input_sequence)
        output, output_h, output_c=self.lstm_layer(input_1, initial_state=states)
        return output, output_h, output_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 ze
        '''
        output_h, output_c=tf.zeros([batch_size,self.lstm_size]), tf.zeros([batch_size,se
        return output_h, output_c
```

Attention

Typesetting math: 100%

In [7]:

```
#Attention#
class Attention(tf.keras.layers.Layer):
    '''
    Class the calculates score based on the scoring_function using Bahdanu attention mech
    '''
    def __init__(self,scoring_function, att_units):
        super().__init__()
        # Please go through the reference notebook and research paper to complete the sco

        self.att_units=att_units
        self.scoring_function=scoring_function
        self.dot=tf.keras.layers.Dot(axes=(1,2))
        self.mult=tf.keras.layers.Multiply()
        self.add=tf.keras.layers.Add()

        pass

    def call(self,decoder_hidden_state,encoder_output):
      '''
      Attention mechanism takes two inputs current step -- decoder_hidden_state and all t
      * Based on the scoring function we will find the score or similarity between decode
      Multiply the score function with your encoder_outputs to get the context vector.
      Function returns context vector and attention weights(softmax - scores)
      '''
      # Implement Dot score function here
      #print('decoder_hidden_state',tf.expand_dims(decoder_hidden_state,1).shape, 'encode
      alphas=tf.matmul(encoder_output,tf.expand_dims(decoder_hidden_state,-1))
      alphas=tf.nn.softmax(alphas)
      context_vector=alphas*encoder_output
      context_vector=tf.reduce_sum(context_vector, axis=1)
      return context_vector,alphas
```

OneStepDecoder

Typesetting math: 100%

In [8]:

```
class One_Step_Decoder(tf.keras.Model):
    def __init__(self,tar_vocab_size, embedding_dim, input_length, dec_units ,score_fun ,
        super().__init__()
        # Initialize decoder embedding layer, LSTM and any other objects needed #, mask_z
        self.embedding_layer=Embedding(input_dim=tar_vocab_size, output_dim=embedding_dim
        self.lstm_layer=LSTM(dec_units, return_state=True, return_sequences=True)
        self.att_units=att_units
        self.score_fun=score_fun
        self.tar_vocab_size=tar_vocab_size
        self.dec_units=dec_units
        self.dense_layer=tf.keras.layers.Dense(tar_vocab_size)
        self.attention=Attention(score_fun,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(bat
          B. Using the encoder_output and decoder hidden state, compute the context vecto
          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(hid
          E. Pass the decoder output to dense layer(vocab size) and store the result into
          F. Return the states from step D, output from Step E, attention weights from St
        '''
        result=self.embedding_layer(input_to_decoder)
        result=tf.squeeze(result, axis=1)

        context_vector, weights=self.attention(state_h, encoder_output)

        output_1=tf.concat([context_vector, result],axis=1)
        output_1=tf.expand_dims(output_1,1)

        decoder_outputs, decoder_h, decoder_c=self.lstm_layer(output_1, initial_state=[st


        final_output=self.dense_layer(decoder_outputs)
        final_output=tf.squeeze(final_output,axis=1)

        return final_output,decoder_h, decoder_c, weights,context_vector
```

Decoder

Typesetting math: 100%

In [9]:

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

      #Intialize necessary variables and create an object from the class onestepdecoder

      self.input_length=input_length
      self.dec_units=dec_units
      self.score_fun=score_fun
      self.att_units=att_units
      self.out_vocab_size=out_vocab_size
      self.embedding_dim=embedding_dim
      self.osd=One_Step_Decoder(tar_vocab_size=self.out_vocab_size, embedding_dim=self.em
                                                                                  input_

      pass
    tf.config.run_functions_eagerly(True)
    @tf.function
    def call(self, input_to_decoder,encoder_output,decoder_hidden_state,decoder_cell_stat

        #Initialize an empty Tensor array, that will store the outputs at each and every
        #Create a tensor array as shown in the reference notebook

        #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
        #print(input_to_decoder.shape)

        output_array=tf.TensorArray(tf.float32,size=input_to_decoder.shape[1])
        #print('input_to_decoder',input_to_decoder.shape)
        for timestep in range(input_to_decoder.shape[1]):
          #print(input_to_decoder.shape, encoder_output.shape, decoder_hidden_state.shape
          output,decoder_hidden_state,decoder_cell_state,attention_weights,context_vector
          output_array = output_array.write(timestep, output)
          #output_array.write(timestep,output).mark_used()
        #.mark_used()
        all_output=tf.transpose(output_array.stack(), [1,0,2])
        #print(all_output.shape)
        return all_output
```

Encoder-Decoder Model

Typesetting math: 100%

In [10]:

```
class encoder_decoder(tf.keras.Model):
  def __init__(self,inp_vocab_size,out_vocab_size, embedding_size, lstm_size, input_lengt
    super().__init__()
    #Intialize objects from encoder decoder
    self.encoder_block=Encoder(inp_vocab_size=inp_vocab_size,embedding_size=embedding_siz
    self.decoder_block=Decoder(out_vocab_size=out_vocab_size, embedding_dim=embedding_siz
    self.batch_size=batch_size
    pass


  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_sequence=data[0]
    output_sequence=data[1]
    #print(input_sequence.shape)
    encoder_h, encoder_c=self.encoder_block.initialize_states(self.batch_size)
    encoder_output, encoder_h, encoder_c=self.encoder_block(input_sequence, states=[encod
    #input_to_decoder,encoder_output,decoder_hidden_state,decoder_cell_state
    dec_h,dec_c=encoder_h, encoder_c
    output_decoder =self.decoder_block(input_to_decoder=output_sequence,encoder_output=er
    #output_decoder=self.soft_max(output_decoder)

    return output_decoder
```

Custom loss function

Typesetting math: 100%

In [11]:

```
#https://www.tensorflow.org/tutorials/text/image_captioning#model
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')

# lr = 0.0001

def loss_function(real, pred):
    """ Custom loss function that will not consider the loss for padded zeros.
    why are we using this, can't we use simple sparse categorical crossentropy?
    Yes, you can use simple sparse categorical crossentropy as loss like we did in task-1
    for the padded zeros. i.e when the input is zero then we donot need to worry what the
    during preprocessing to make equal length for all the sentences."""


    mask = tf.math.logical_not(tf.math.equal(real, 0))
    loss_ = loss_object(real, pred)


    mask = tf.cast(mask, dtype=loss_.dtype)

    loss_ *= mask


    return tf.reduce_mean(loss_)
optimizer = tf.keras.optimizers.Adam()
```

Dataset

Typesetting math: 100%

In [12]:

```
class Dataset:
    def __init__(self, data, tknizer_ita, tknizer_eng, max_len):
        self.encoder_inps = data['incorrect'].values
        self.decoder_inps = data['correct_inp'].values
        self.decoder_outs = data['correct_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]]) #
        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='ir
        self.decoder_inp_seq = pad_sequences(self.decoder_inp_seq, maxlen=self.max_len, c
        self.decoder_out_seq = pad_sequences(self.decoder_out_seq, maxlen=self.max_len, c
        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 alrec
        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, incorrect_tk, correct_tk, 16)
validation_dataset = Dataset(validation,  incorrect_tk, correct_tk, 16)

train_dataloader = Dataloder(train_dataset, batch_size=512)
validation_dataloader = Dataloder(validation_dataset, batch_size=512)



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

```
(512, 16) (512, 16) (512, 16)
```

Typesetting math: 100%

Custom function to save the model

In [13]:

```
import matplotlib.pyplot as plt
import seaborn as sns

class CustomSaver(tf.keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs={}):
        self.model.save_weights("model_2/model_2_epoch_{}.h5".format(epoch))

saver=CustomSaver()
```

Metric to calculate F1_Beta Score while training

- I did not use it because of the time it had taken for one epoch.
- The calculations are huge hence increasing the time for an epoch.

In [14]:

```
#
from sklearn.metrics import fbeta_score

tf.autograph.set_verbosity(0, True)
@tf.function
def f_beta_score(y_true, y_pred):
  #print(y_pred.shape)
  y_pred_sparse = tf.convert_to_tensor(np.argmax(y_pred, axis = -1), dtype = tf.float32)
  #print(y_pred_sparse.shape)
  fb_score = [ fbeta_score(y_true[i], y_pred_sparse[i],average = 'macro',beta = 0.5) for
  #print(len(fb_score))
  #print(y_true.shape[0])
  return sum(fb_score)/len(fb_score)
```

Training

Typesetting math: 100%

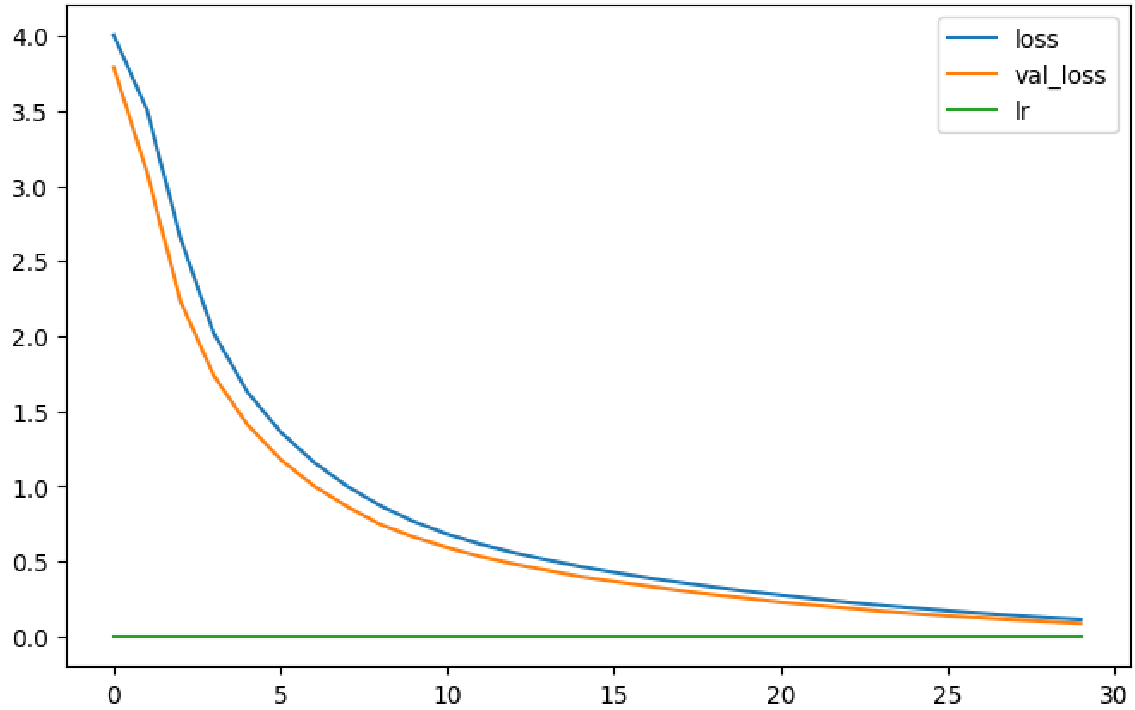
In [17]:

```
input_vocab_size = len(incorrect_tk.word_index)+1
output_vocab_size = len(correct_tk.word_index)+1

input_len = 16
output_len = 16

lstm_size = 512
att_units = 512
dec_units = 512
embedding_size = 300
score_fun = 'dot'

BATCH_SIZE=512

lr_rate=tf.keras.callbacks.ReduceLROnPlateau(patience=4,min_delta=0.01)
stopping=tf.keras.callbacks.EarlyStopping(min_delta=0.01, patience=5)

#Create an object of encoder_decoder Model class,
# Compile the model and fit the model
model_1 = encoder_decoder(input_vocab_size,output_vocab_size,embedding_size,lstm_size,inp
optimizer = tf.keras.optimizers.Adam()
model_1.compile(optimizer=optimizer,loss=loss_function)
train_steps=train.shape[0]//512
valid_steps=validation.shape[0]//512
model_1.fit(train_dataloader, steps_per_epoch=train_steps, epochs=30, validation_data=tra

pd.DataFrame(model_1.history.history).plot(figsize=(8,5))
plt.show()
```

Typesetting math: 100%

```
Epoch 1/30
487/487 [==============================] - 245s 502ms/step - loss: 4.0035
- val_loss: 3.7904 - lr: 0.0010
Epoch 2/30
487/487 [==============================] - 237s 487ms/step - loss: 3.5068
- val_loss: 3.0916 - lr: 0.0010
Epoch 3/30
487/487 [==============================] - 234s 481ms/step - loss: 2.6533
- val_loss: 2.2319 - lr: 0.0010
Epoch 4/30
487/487 [==============================] - 234s 480ms/step - loss: 2.0183
- val_loss: 1.7374 - lr: 0.0010
Epoch 5/30
487/487 [==============================] - 232s 477ms/step - loss: 1.6309
- val_loss: 1.4129 - lr: 0.0010
Epoch 6/30
487/487 [==============================] - 232s 476ms/step - loss: 1.3626
- val_loss: 1.1802 - lr: 0.0010
Epoch 7/30
487/487 [==============================] - 228s 467ms/step - loss: 1.1607
- val_loss: 1.0026 - lr: 0.0010
Epoch 8/30
487/487 [==============================] - 228s 469ms/step - loss: 1.0005
- val_loss: 0.8642 - lr: 0.0010
Epoch 9/30
487/487 [==============================] - 229s 470ms/step - loss: 0.8695
- val_loss: 0.7446 - lr: 0.0010
Epoch 10/30
487/487 [==============================] - 226s 464ms/step - loss: 0.7644
- val_loss: 0.6615 - lr: 0.0010
Epoch 11/30
487/487 [==============================] - 225s 461ms/step - loss: 0.6808
- val_loss: 0.5921 - lr: 0.0010
Epoch 12/30
487/487 [==============================] - 225s 462ms/step - loss: 0.6139
- val_loss: 0.5320 - lr: 0.0010
Epoch 13/30
487/487 [==============================] - 225s 463ms/step - loss: 0.5579
- val_loss: 0.4824 - lr: 0.0010
Epoch 14/30
487/487 [==============================] - 226s 465ms/step - loss: 0.5090
- val_loss: 0.4404 - lr: 0.0010
Epoch 15/30
487/487 [==============================] - 226s 465ms/step - loss: 0.4659
- val_loss: 0.3980 - lr: 0.0010
Epoch 16/30
487/487 [==============================] - 226s 464ms/step - loss: 0.4265
- val_loss: 0.3661 - lr: 0.0010
Epoch 17/30
487/487 [==============================] - 226s 464ms/step - loss: 0.3909
- val_loss: 0.3353 - lr: 0.0010
Epoch 18/30
487/487 [==============================] - 227s 467ms/step - loss: 0.3586
- val_loss: 0.3052 - lr: 0.0010
Epoch 19/30
487/487 [==============================] - 228s 469ms/step - loss: 0.3284
- val_loss: 0.2769 - lr: 0.0010
Epoch 20/30
487/487 [==============================] - 225s 463ms/step - loss: 0.3005
- val_loss: 0.2532 - lr: 0.0010
Epoch 21/30
```

```
487/487 [==============================] - 226s 464ms/step - loss: 0.2748
- val_loss: 0.2281 - lr: 0.0010
Epoch 22/30
487/487 [==============================] - 226s 464ms/step - loss: 0.2503
- val_loss: 0.2082 - lr: 0.0010
Epoch 23/30
487/487 [==============================] - 227s 466ms/step - loss: 0.2280
- val_loss: 0.1882 - lr: 0.0010
Epoch 24/30
487/487 [==============================] - 223s 459ms/step - loss: 0.2073
- val_loss: 0.1683 - lr: 0.0010
Epoch 25/30
487/487 [==============================] - 227s 467ms/step - loss: 0.1886
- val_loss: 0.1519 - lr: 0.0010
Epoch 26/30
487/487 [==============================] - 226s 464ms/step - loss: 0.1706
- val_loss: 0.1365 - lr: 0.0010
Epoch 27/30
487/487 [==============================] - 227s 466ms/step - loss: 0.1540
- val_loss: 0.1234 - lr: 0.0010
Epoch 28/30
487/487 [==============================] - 226s 464ms/step - loss: 0.1390
- val_loss: 0.1102 - lr: 0.0010
Epoch 29/30
487/487 [==============================] - 225s 463ms/step - loss: 0.1250
- val_loss: 0.0990 - lr: 0.0010
Epoch 30/30
487/487 [==============================] - 227s 467ms/step - loss: 0.1125
- val_loss: 0.0871 - lr: 0.0010
```



Typesetting math: 100%

In [ ]:

```
model_1.fit(train_dataloader, steps_per_epoch=train_steps, epochs=30, validation_data=tra
```

```
Epoch 1/30

/home/josephnadar1998/miniconda3/envs/tf/lib/python3.9/site-packages/tenso
rflow/python/data/ops/structured_function.py:256: UserWarning: Even though
the `tf.config.experimental_run_functions_eagerly` option is set, this opt
ion does not apply to tf.data functions. To force eager execution of tf.da
ta functions, please use `tf.data.experimental.enable_debug_mode()`.
  warnings.warn(

487/487 [==============================] - 230s 472ms/step - loss: 0.1013
- val_loss: 0.0796 - lr: 0.0010
Epoch 2/30
487/487 [==============================] - 228s 469ms/step - loss: 0.0913
- val_loss: 0.0728 - lr: 0.0010
Epoch 3/30
487/487 [==============================] - 230s 473ms/step - loss: 0.0825
- val_loss: 0.0685 - lr: 0.0010
Epoch 4/30
 41/487 [=>............................] - ETA: 3:00 - loss: 0.0685
```

The notebook disconnected, hence had to start again.

In [15]:

```
input_vocab_size = len(incorrect_tk.word_index)+1
output_vocab_size = len(correct_tk.word_index)+1

input_len = 16
output_len = 16

lstm_size = 512
att_units = 512
dec_units = 512
embedding_size = 300
score_fun = 'dot'

BATCH_SIZE=512

lr_rate=tf.keras.callbacks.ReduceLROnPlateau(patience=4,min_delta=0.01)
stopping=tf.keras.callbacks.EarlyStopping(min_delta=0.01, patience=5)

#Create an object of encoder_decoder Model class,
# Compile the model and fit the model
model_1 = encoder_decoder(input_vocab_size,output_vocab_size,embedding_size,lstm_size,inp
optimizer = tf.keras.optimizers.Adam()
model_1.compile(optimizer=optimizer,loss=loss_function)
train_steps=train.shape[0]//512
valid_steps=validation.shape[0]//512
```

Typesetting math: 100%

In [16]:

```
model_1.build((None,512,16))
model_1.load_weights('model_2/model_2_epoch_2.h5')
```

Typesetting math: 100%

In [17]:

```
model_1.fit(train_dataloader, steps_per_epoch=train_steps, epochs=20, validation_data=tra
```

```
Epoch 1/20

/home/josephnadar1998/miniconda3/envs/tf/lib/python3.9/site-packages/tenso
rflow/python/data/ops/structured_function.py:256: UserWarning: Even though
the `tf.config.experimental_run_functions_eagerly` option is set, this opt
ion does not apply to tf.data functions. To force eager execution of tf.da
ta functions, please use `tf.data.experimental.enable_debug_mode()`.
  warnings.warn(
```

Typesetting math: 100%

```
487/487 [==============================] - 226s 456ms/step - loss: 0.0808
- val_loss: 0.0607 - lr: 0.0010
Epoch 2/20
487/487 [==============================] - 222s 456ms/step - loss: 0.0683
- val_loss: 0.0539 - lr: 0.0010
Epoch 3/20
487/487 [==============================] - 210s 431ms/step - loss: 0.0612
- val_loss: 0.0512 - lr: 0.0010
Epoch 4/20
487/487 [==============================] - 212s 436ms/step - loss: 0.0570
- val_loss: 0.0465 - lr: 0.0010
Epoch 5/20
487/487 [==============================] - 208s 427ms/step - loss: 0.0529
- val_loss: 0.0430 - lr: 0.0010
Epoch 6/20
487/487 [==============================] - 211s 432ms/step - loss: 0.0491
- val_loss: 0.0412 - lr: 0.0010
Epoch 7/20
487/487 [==============================] - 206s 424ms/step - loss: 0.0452
- val_loss: 0.0371 - lr: 0.0010
Epoch 8/20
487/487 [==============================] - 205s 420ms/step - loss: 0.0418
- val_loss: 0.0343 - lr: 0.0010
Epoch 9/20
487/487 [==============================] - 204s 419ms/step - loss: 0.0386
- val_loss: 0.0327 - lr: 0.0010
Epoch 10/20
487/487 [==============================] - 207s 425ms/step - loss: 0.0362
- val_loss: 0.0309 - lr: 0.0010
Epoch 11/20
487/487 [==============================] - 206s 422ms/step - loss: 0.0345
- val_loss: 0.0314 - lr: 0.0010
Epoch 12/20
487/487 [==============================] - 204s 419ms/step - loss: 0.0344
- val_loss: 0.0297 - lr: 0.0010
Epoch 13/20
487/487 [==============================] - 207s 426ms/step - loss: 0.0196
- val_loss: 0.0120 - lr: 1.0000e-04
Epoch 14/20
487/487 [==============================] - 206s 424ms/step - loss: 0.0117
- val_loss: 0.0095 - lr: 1.0000e-04
Epoch 15/20
487/487 [==============================] - 208s 428ms/step - loss: 0.0096
- val_loss: 0.0082 - lr: 1.0000e-04
Epoch 16/20
487/487 [==============================] - 205s 422ms/step - loss: 0.0084
- val_loss: 0.0074 - lr: 1.0000e-04
Epoch 17/20
487/487 [==============================] - 208s 427ms/step - loss: 0.0076
- val_loss: 0.0067 - lr: 1.0000e-04
Epoch 18/20
487/487 [==============================] - 207s 425ms/step - loss: 0.0068
- val_loss: 0.0066 - lr: 1.0000e-05
```

Out[17]:

```
<keras.callbacks.History at 0x7f9a423adcd0>
```

Code for Inference

In [22]:

```
def predict_m1(input_sentence):
    words=[]
    input_sentence=[input_sentence]
    batch_size=1
    tokenized_sent=incorrect_tk.texts_to_sequences(input_sentence)
    #print(tokenized_sent)
    padded_sent=tf.keras.utils.pad_sequences(tokenized_sent, maxlen=16,padding='post' )
    encoder_h, encoder_c=model_1.layers[0].initialize_states(batch_size)
    encoder_output,encoder_h, encoder_c= model_1.layers[0](padded_sent,states=[encoder_h
    start_index=correct_tk.word_index.get('<start>')
    end_index=correct_tk.word_index.get('<end>')
    for i in range(16):
        decoder_output, decoder_h, decoder_c, attention_weights, context_vector = model_1
        output_index=np.argmax(decoder_output[0])
        start_index=output_index
        #print(output_index)
        encoder_h, encoder_c=decoder_h, decoder_c
        words.append(correct_tk.index_word[output_index])
        #print(list(tknizer_eng.word_index.keys())[output_index])
        if output_index==end_index:
            break
    return ' '.join(words[:-1])
i=test.iloc[28]['incorrect']
print(i)
predict_m1(i)
```

```
i cannot believe i have posted my entries in english for days at a time
```

Out[22]:

```
'i cannot believe my entries in my entries in entries for weeks'
```

Typesetting math: 100%