UNIVERSITY OF STAVANGER
Department of Electrical Engineering and Computer Science
Group 3
Bruna Atamanczuk (254205)
Shaima Ahmad Freja (261376)

Neural Machine Translation – From Spanish to English

ELE680 -Deep neural networks

Stavanger, 16.10.2022

1. Introduction

Machine translation is the task of translating a sequence from a source language to a target language, this task is known as sequence-to-sequence (seq2seq) learning (Chollet, 2017). One of the approaches to perform machine translation is to use recurrent neural networks (RNN) using the Encoder-Decoder framework.

In the general case, the input sentence is passed through the encoder, where it is processed by a RNN layer. Unlike other use cases where the output of the RNN layer is used, for machine translation, we focus only on the hidden state of the RNN. This state provides information about the context of the sequence, and it will serve as one of the inputs to the decoder unit. The decoder will predict the characters of the target sentence based on the previous characters. In other words, the RNN layer of the decoder will take as an input the context vector and each of the words in a sentence to predict the following word. The main drawbacks of this approach as described, are (1) RNNs are slow to train, specially is Long-Short-Term Memory (LSTM) layer is used, and (2) they can be inefficient to process long sequences. A representation of the model is given in Figure 1.

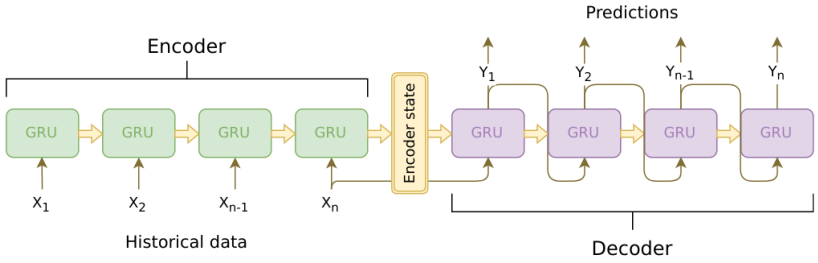

Figure 1 - Encoder decoder model. Retrieved from: https://camo.githubusercontent.com/fe3df3ef6c0fb05767a373ef8e547120a0a667c423a5c52f3f202156b8aba1e5/68747470733a2f2f696e6372656469626c652e61692f6173736574732f696d616765732f736571327365712d736571327365715f74732e706e67

As show in the picture above, each character of the sequence is processed at a time which contribute to the model inefficiency. Another point worth noticing is the inability of processing long sequences even if more complex architectures are used such as LSTM and Gated Recurrent Unit (GRU).

Given the drawbacks of the traditional seq2seq model, a mechanism called Attention was introduced to improve the translation of sequences by placing focus only in parts of the sentence (Luong et al., 2015). The intuition behind the Attention mechanism is that more data is passed from the encoder to the decoder. This means that instead of only focusing on the last state of the encoder, all the states are now being passed to each step of the decoder, making the information about all elements of the input sequence available. The decoder is then able to assign more weight to certain elements of the input sequence to predict the next output. Even though the Attention mechanism presents improvements in comparison to the traditional seq2seq model, it has an important limitation, the computations are still performed one element of the sequence at the time which can be both time consuming and inefficient, depending on the size of the corpus.

In 2017, the transformer model (Vaswani et al., 2017) was introduced, and revolutionized the use of attention mechanism. This model relied in a mechanism called self-attention, which does not require the use of recurrence and convolutions to perform

translations. Self-attention allows the model to understand the underlying meaning of the language by looking at the context of the surrounding words in a sentence (Tech, 2021). For example, in the sentences “Server, can I have the check” and “It looks like the server has crashed” the word “server” has different meanings. In this case, self-attention allows the model to look at different words in each sentence to determine whether the “server” is a human or a machine. In the first sentence, the model might attend to the word check to determine the meaning of “server”, whereas in the second sentence it may attend to the word “crashed” to determine that we are referring to an object. This mechanism allows the transformers to have a deeper understanding of grammar providing better results to a plethora of tasks in natural language processing (Luong et al., 2015).

Another breakthrough was that transformers could handle parallelized operations which rendered more efficient models, and the ability of processing extremely long textual data. The model also makes use of positional encodings where instead of looking at words sequentially as it is the case in seq2seq models, each word is mapped before feeding the sequence to the model. This means that the information is stored in the data itself rather than in the structure of the network. As we train the model in a lot of data, it learns how to interpret the positional encodings and further the importance of word order in the sentences (Tech, 2021).

At a higher level, the transformer architecture is also built following the Encoder-Decoder framework. The original architecture is composed by 6 encoders and 6 decoders. The representation of the model is shown in the Figure 2, where one encoder and one decoder are displayed for the sake of simplicity.

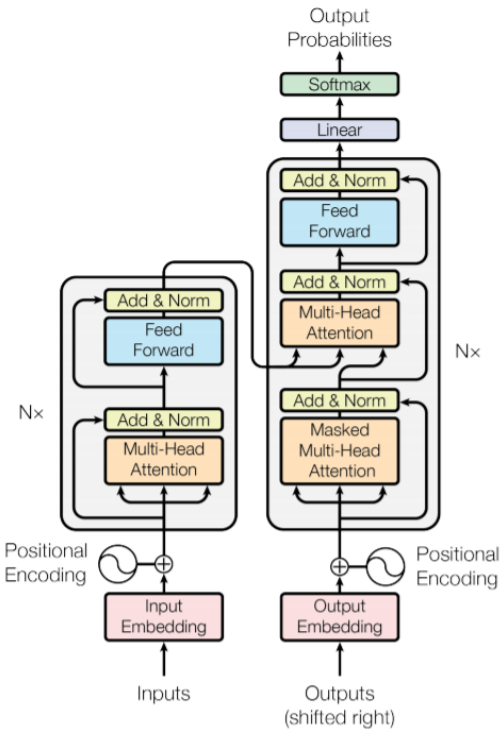

Figure 2 - Transformer architecture

Each encoder contains a self-attention layer and a feed forward layer. The encoder takes the words in the sentences as inputs at the same time and pass it through the self-

attention layer which will generate scores for each word against the input sentence. This self-attention layer is a multi-head attention, which is a module that runs through the attention mechanism multiple times in parallel, and allow the model to attend to parts of the sequence differently (Weng, 2018). The feed forward layer's main purpose is to process the input from output of one attention layer so it can better fit the next attention layer.

Once the source text data is encoded, the outputs of the encoder are then passed to the decoder. The decoder has similar sub layers as the encoder. However, it contains two attention layers that serve a different purpose. The first multi-head attention layer takes in the positional embeddings of the target sequence. Its primary job is to prevent the decoder to have access to future tokens, as it is autoregressive. This means that the decoder processes each word in the target sentence at a time. The second multi-head attention layer is responsible to match the encoders outputs to the processed outputs of the first attention layer. This layer allows the decoder to decide which elements to attend to. The outputs of this layer go through a feed forward layer and finally through a `softmax` layer. The `softmax` layer assigns probability scores to each word and the highest score gives the predicted word. This process is repeated until the decoder reaches the end of the sentence.

In this report we use a seq2seq model with attention mechanism and a Transformer model to translate text from Spanish to English. The models are trained using 125111 sequence pairs and tested using 13902 pairs. During training the model loss and accuracy are monitored and for the test data, we use the Rouge-L score or BLEU to evaluate the quality of the translations.

2. Material and Methods

2.1. The data

In this report we used a collection of sentence pairs provided by the Tatoeba Project (http://www.manythings.org/anki/). The dataset contains 139013 sentence pairs in Spanish and English languages. The file size is 19.7MB. The data is distributed in 3 columns, one containing the English sentences, one for the Spanish and the last has information about the person/entity that provided the translation for that pair. For the purposes of this work, we remove the third column as it will not contribute to our objectives. The Table 1 below demonstrates how the dataset is organized.

Table 1 - Sample of sequence pairs

English	Spanish
Listen.	Escuchen.
No way!	¡No puede ser!
No way!	De ninguna manera.

2.2. Data Preprocessing

The first step towards document translation is to preprocess the data into a format that is suitable for the algorithm to understand the information. This step is comprised of 2 main procedures: data cleaning and text vectorization. Data cleaning is done to remove unnecessary features that will not contribute to the overall results of our models. Since we are dealing with different languages, we expect that each of these will have a unique set of characters. In the data cleaning step, we removed special characters, such as punctuation, numeric characters, and Spanish accented letters, such as *á, é, í, ó, ú* and *ñ*. We also split the data into train and test set before going into vectorization of the sentences. This is done to avoid leaking information about the test set to the model.

In our model, we start by preparing the data to preprocessing. We adopted a 90-10 split, meaning that 90% of the dataset (125111 rows) will be used to train the model and the remaining 10% (13902 rows) will be used to test it.

Once the data has been split into train and test set, we use Tensorflow to create a data set and vectorize the sentence pairs. For the first model, the training set is further split into training and validation, using the same 90-10 split. Tensorflow's `tf.data.Dataset.from_tensor_slices` is used to build both training and validation sets, for the seq2seq model we used a 90-10 split to produce the validation set, and 90-10 to produce the validation set of the transformer model. For both models the batch size of 64 was used.

The train and validation datasets are then used to create a `tf.keras.layers.TextVectorization` object. In this process, each sentence is split into a list of words (tokenization). For text vectorization we perform the following steps:

1. Use the custom standardized functions to clean the sentence, normalize the text, and remove punctuation characters.
2. Add [start] and [end] token for the target sentences (both languages for seq2seq and only English for the transformer).
3. Split on white space.
4. Create a word index for each token.
5. Pad each sentence to a maximum Sequence (50).

5.1. Methods

5.1.1. Recurrent neural network with attention mechanism

A seq2seq model (RNN with attention) was developed to construct the baseline to evaluate the translation of sentences from Spanish to English. The translation model was created based on the Encoder-Decoder framework, and the Tensorflow implementation Tutorial (Tensorflow, 2022) was used for this case. The Encoder has a bidirectional RNN and the decoder has a unidirectional RNN. Additionally, the model has an attention layer that will help the Decoder focus on specific parts of each input sentence while generating the translated outputs.

The Encoder was created as a class that contains an embedding layer and a bidirectional GRU layer. The embedding layer has 256 units, and a masking parameter

is passed so that it ignores the mask tokens. The bidirectional layer `merge_mode` was set to sum and was wrapped around the GRU layer. The GRU `return_sequence` was set to `True` so that it could be passed to the attention head.

The Attention mechanism used is called Cross-attention. This mechanism combines two separate embedding sequences of the same dimension (Kosar, 2022). The input sequence information that passes through the attention layer was then introduced to the Decoder's output, such that it could predict the next output sequence token. The attention layer class was constructed using tensorflow's `MultiHeadAttention` layer with 256 units and one head. A `LayerNormalization` layer is also applied so that all features in each input have the same distribution and batch dependency is removed (Priya, 20__).

Finally, the Decoder class contain all methods necessary to build the seq2seq model. Its job is to generate the predictions for each location in the target sequence. In the decoder it is important that information flows at only one direction. In this case we use a GRU layer to process the target. The total trainable parameters are shown in Table 2.

Table 2 - Total number of parameters for RNN model with attention

Seq2seq Translator	
Layer	No. of parameters
Encoder	5 909 504
Decoder	7 407 460
Total no. of parameters	13 316 964

The model was trained using a batch size of 64, 50 epochs, 150 steps per epoch, maximum vocabulary size was set to 20000. In addition, we monitored the accuracy and loss during training. A `EarlyStopping` callback was defined to monitor the validation loss, so that if the validation loss does not decrease for 3 epochs, then the model stops training.

5.1.2. Transformer

The Transformer model at a high level consists of an Encoder which reads the source sequence and a Decoder which predicts the future tokens in the target sequence. We adopted an implementation for a basic transformer from scratch presented by Chollet (2021) and updated some hyper parameters to improve the accuracy through many changes to the code.

The Transformer architecture which we implemented consisted of one layer of the Encoder and one layer of the Decoder. The general structure for the transformer depends on several types of attention (multi-head attention, self-attention, and Masked attention which is only used in the decoder). Multi head attention expands the model's ability to focus on various positions. And to make a model recognize the word order we use the positional embedding layer (Alammar, 2020).

After preprocessing the data, we prepared the input data for the seq2seq transformer model by reformatting the training & validation sets to be used as an input for the encoder which will take a tuple of (inputs, targets), where the inputs consisted of a dictionary of two keys (encoder inputs given by Spanish tensors and decoder inputs given by the English tensors without the last token [end]). The target was set as the English tensors without the first token [start]. Finally, a tensor of the same shape (`batch_size=64, max_sequence=50`) was generated.

```
return ({"encoder_inputs": spa, "decoder_inputs": eng[:, :-1],}, eng[:, 1:])
```

The Transformer model consisted of 3 main components (Encoder, Decoder, final layer and SoftMax). The encoder was created as a class which consists of the following layers:

1. Muti head attention layer with 8 heads and 256 units.
2. Feed forward layer with two dense layers, the first layer takes 2048 units and '`Relu`' as an activation function and the second layer takes 256 units.
3. Two layers for Normalization.
4. Drop out layer with `rate = 0.1`.

The Decoder consisted of (self-attention (Multi head attention), Normalizing layer, Feed Forward layer, `dropout(0.1)`) The Decoder class has the same layers for the encoder with extra multi-head attention layer and three layers for normalization in addition the dropout layer with `rate = 0.1`.Finally, the transformer model class combined the encoder and decoder the output of the decoder is the input for the final linear(Dense) layer which is a fully connected layer that projects the vector produced by the decoder, into a larger vector (logits) with the same size for the vocabulary (20000), each cell contained a score of a unique word, then the SoftMax turned those scores into a probability where the highest number assigns the correspondent word . We used Adam optimizer with `learning_rate = 0.001` and trained the model using a set of hyperparameters of `batch_size=` 64, 50 epochs with maximum vocabulary size equal to 20000 and callbacks to save the accuracy & loss for each epoch to a CSV file. The Early stopping was assigned `patience=3` when the validation loss started to increase for 10 (epochs). The total parameters for the model are

Table 3 - Total number of parameters for Transformer model

Transformer	
Layer	No. of parameters
Positional_embbedding	5 120 000
Encoder	3 155 456
Decoder	15 519 520
Total no. of parameters	23 794 976

To produce the translations, we loop through each sentence in the testing set and pass the vectorized Spanish sentence(source) with the target token for the English sentence [start] into the transformer model which will encode the input sentence by the encoder, then the decoder repeatedly generated the next token until it reached to the token [end]. After that we removed the extra tokens [start], [end] from the translated sentence and save all the translated and the original sentence into csv file to use it for evaluation.

5.1.3. Fine-tuned Transformer

Finally, we used a pretrained transformer model to translate the sentences to English. The model used the Huggingface framework. We used `Helsinki-NLP/opus-mt-es-en` benchmark to fine tune the model. The pretrained weights were loaded using the `AutoTokenizer` object, and later the train and validation data were used to generate a seq2seq model. Pretrained models are highly optimized for translation tasks, the model used is a transformer encoder-decoder with 6 layers in each component. The implantation of this model diverged from the previous preprocessing steps, as we only needed to prepare a Huggingface dataset object. We followed the steps presented by Kumar (2021).

1. Preprocess de data using Transformers `AutoTokenizer` and truncating the sentences to ensure that are no inputs longer than the model can handle.
2. Use the pretrained `AutoModelForSeq2SeqLM` class to call the model
3. Instantiate the Training arguments using the class `Seq2SeqTrainingArguments` that allows to customize the training and saves the model to an assigned folder. We also assign the learning rate, batch size and customize the weight decay
4. Pass the data to `Seq2SeqTrainer` to fine tune
5. Translate the text

5.1.4. Evaluation metrics

The Bilingual Evaluation Understudy (BLEU) was used for evaluating the quality of machine translation. This metric compares a candidate translation to one or more reference translations. The score ranges from 0 to 1 (or 0 to 100), where 1 indicates a perfect match and 0 a complete mismatch between the sentences. The general idea is to compare the sentences by counting matching n-grams. First, we count the frequency that each n-gram appears in the machine translation output. Then we compute the frequency of which n-gram appears in the references. The n-gram frequency for the references is defined by the highest frequency amongst the references (Ng, 2018). BLEU is calculated as:

$$P_n = \frac{\sum_{n\_gram \in y} Count_{clip}(n\_gram)}{\sum_{n\_gram \in \hat{y}} Count(n\_gram)} \qquad (1)$$

Where $Count_{clip}(n\_gram)$ is the highest frequency that a n-gram appear in the references, and $Count(n_gram)$is the frequency that a n-gram appears in the machine

translation output (Ng, 2018). In this work, we used nltk's library to get the overall BLEU score for translations.

6. Results

We ran the RNN model with attention in a local machine. We set the total number of epochs as 50, however, due to the callback the training stopped at epoch 29. We used a validation split of 10%, and 100 steps per epoch. For this setup, the model accuracy was 0.77 and the translations achieved a BLEU score of 40. The plots for both accuracy and loss for the training and validation sets are shown below. It is possible to see that after epoch 20, both metrics do not change significantly. The results are shown in Figure 3.

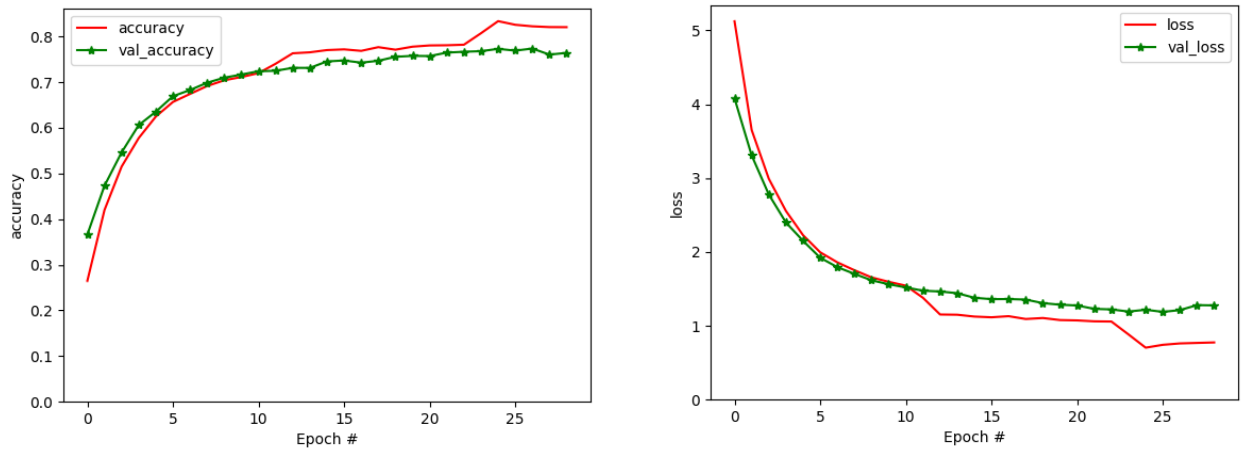

Figure 3 - Accuracy and loss for seq2seq model

For the Transformer model we used different settings, however it is possible to see that training for more epochs did not improve the model accuracy. The BLEU score for this model was 0.015, which is lower than for the seq2seq implementation. The curves for loss and accuracy for the Transformer implementation are displayed in Figure 4.

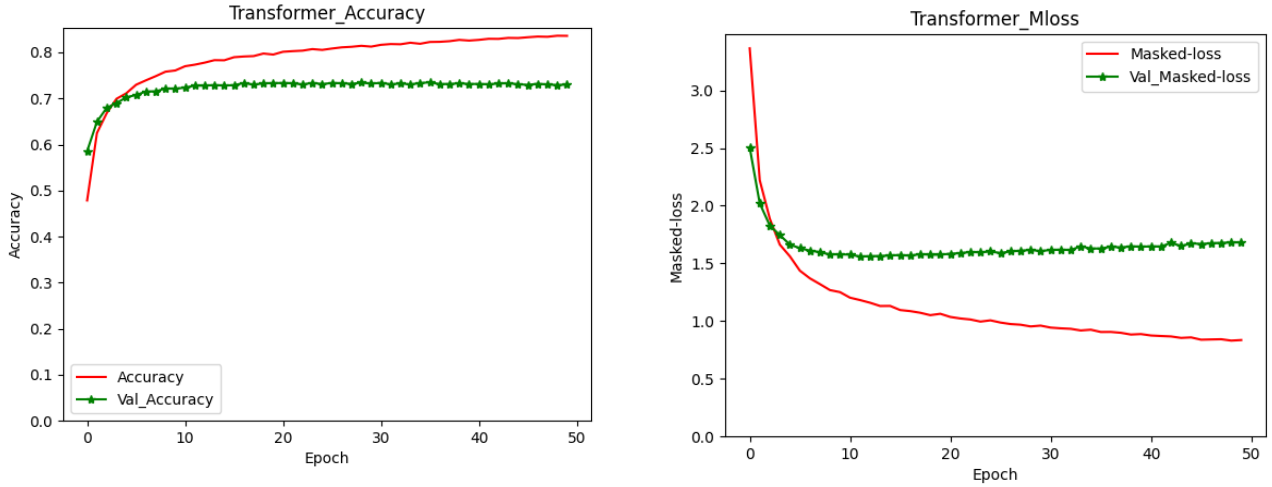

Figure 4 - Accuracy and loss curves for the transformer model

The results for the models are presented in Table 4.

Table 4 - Results for training and validation

Model		**Training**		**Validation**	
	Epochs	**Masked-Accuracy**	**Masked - loss**	**Masked-Accuracy**	**Masked - loss**
RNN	**50**	0.83	0.70	0.77	1.21
Transformer	50	0.88	0.58	0.72	1.72
Fine-tuned transformer	1	-	0.38	-	0.34

We fine-tuned the pretrained model for one epoch, and this was sufficient to achieve the smallest loss for training and validation, 0.39 and 0.35 respectively. Since we used a different framework in this case, accuracy was not monitored in this step. However, the model displays the BLEU score for evaluation in Huggingface's implementation. The BLEU score for evaluation was 68.82. In addition, we translated the sentences from our test set using this new fine-tuned model. The BLEU score for the test data was 66. The BLEU score for all models is presented in Table 5.

Table 5 - BLEU score for all models

BLEU score		
RNN with attention	Transformer	Fine-tuned transformer
40	0.02	66

7. Discussion

Seq2seq models can be very powerful, especially for translation tasks. Given the results of our models, we could see that the simple RNN with attention performed well when translating short sentences. However, given the nature of RNNs we could see that the model could not provide good translations for longer sentences, even when we used an attention layer. We could also notice that something went wrong in the implementation from scratch of the transformer model, as we achieved a lower BLEU score for this case.

A point worth noticing is that accuracy is never a good metric to evaluate how well these models are performing. Accuracy can be misleading, as we can see in this work. We achieved a high score for both models. However, the translations were not satisfactory for long sentences. BLEU is far more reliable metric as we compare the frequency to which n-grams are repeated in both source and target sentences. However, BLEU score has some pitfalls as well. Since there is no homogenous way of translating languages, and our references only provide one source of truth, it can be that our models are in fact providing decent translations for some cases but being penalized in the BLEU score for the case where both translations provide the same meaning to the sentence. This could cause a lower BLEU score to be assigned.

As we expected, the pretrained model outperformed the RNN model and our from-scratch transformer implementation. This model used a very elaborated architecture and pretrained weights which lead the model to capture the context of longer sentences. For our transformer implementation from scratch, we should have kept a lower patience and added more units and probably mode encoder decoder layers, as we can see by our accuracy and loss curves that the model is underfitting. We tried multiple approaches (increased number of units, steps per epoch, vocabulary size), but were unsuccessful.

Finally, given more time we would like to explore more options to our transformer implementation, as we could not find what was causing issues with the generated translations. We could try different preprocessing methods and changed the way the translation is performed. Overall, the main challenges with the implementation were to understand how each layer connected to each other and use the right dimensions for each layer.

8. References

Alammar, J. (2020). *The Illustrated Transformer*. Retrieved 29 Sep. from http://jalammar.github.io/illustrated-transformer/

Chollet, F. (2017). *A ten-minute introduction to sequence-to-sequence learning in Keras*. Retrieved 11 Oct. from https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html

Chollet, F. (2021). *English-to-Spanish translation with a sequence-to-sequence Transformer*. Retrieved 14 Oct. from https://keras.io/examples/nlp/neural_machine_translation_with_transformer/

Kosar, V. (2022). *Cross-Attention in Transformer Architecture*. Retrieved 28 Sep. from https://vaclavkosar.com/ml/cross-attention-in-transformer-architecture

Kumar, S. (2021). *How to fine-tune pre-trained translation model*. Retrieved 15 Oct. from https://medium.com/@tskumar1320/how-to-fine-tune-pre-trained-language-translation-model-3e8a6aace9f

Luong, M.-T., Pham, H., & Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*.

Ng, A. (2018). *C5W3L06 Bleu Score (Optional)*. https://www.youtube.com/watch?v=DejHQYAGb7Q&t=681s

Priya, B. (20__). *Batch and Layer Normalization*. Retrieved 09 Oct from https://www.pinecone.io/learn/batch-layer-normalization/

Tech, G. C. (2021). *Transformers, explained: Understand the model behind GPT, BERT, and T5*. https://www.youtube.com/watch?v=SZorAJ4I-sA&t=477s

Tensorflow. (2022). *Neural machine translation with attention*. Retrieved 14 Oct from https://www.tensorflow.org/text/tutorials/nmt_with_attention

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, *30*.

Weng, L. (2018). *Attention? Attention!* Retrieved 11 Oct. from https://lilianweng.github.io/posts/2018-06-24-attention/

APPENDIX

The data was first split into train and test using the following function

```
import pandas as pd
from sklearn.model_selection import train_test_split

data_path = "data\spa-eng\spa.txt"

raw= pd.read_table(data_path,names=['english', 'spanish', 'comment'])
raw.drop(columns=['comment'], inplace=True)
raw.to_csv('data\eng-esp.csv', index=False, sep='\t', header=False)

train, test = train_test_split(raw, test_size=0.1,shuffle=True)
train.to_csv(r'data/train.csv', index=False, sep='\t', header=False)
test.to_csv(r'data/test.csv', index=False, sep='\t', header=False)
```

A1. The RNN with attention model

The requirements to run the baseline model are Python 3.9.13, tensorflow-2.10.0 tensorflow-text-2.10.0. This model was run locally. The model was implemented in 2 separate files. First, we present the main file, which shall be executed to run the model, followed by the implementation of the model.

```
'''This file contains all functions used in seq2seq model'''


import tensorflow as tf
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import matplotlib.ticker as ticker
import tensorflow_text as tf_text
tf.random.set_seed(56)
np.random.seed(15)

# Define parameters for the model
#==================================================
BATCH_SIZE = 64
max_vocab_size = 20000
UNITS = 256
EPOCHS = 5
STEPS = 100
VAL_STEPS = 20
```

```
#=====================================================

#=====================================================
# 1. Loading the data
# - Read from training.csv, testing.csv
def load_data(path):
    '''Load dataset and return a tuple with target and source sentences'''
    text = pd.read_csv(path, sep='\t', names=['en', 'es'])

    source = text['es'].to_numpy()
    target = text['en'].to_numpy()

    return target, source

#=====================================================
# 2. Preprocessing
#=====================================================

# 2.1 - Defining preprocessing function
#=====================================================
# Remove special characters and punctuation
# Add [START] and [END] tokens
def tf_lower_and_split_punct(text):
    # Split accented characters.
    text = tf_text.normalize_utf8(text, 'NFKD')
    text = tf.strings.lower(text)
    # Keep space, a to z, and select punctuation.
    text = tf.strings.regex_replace(text, '[^ a-z.?!,¿]', '')
    # Add spaces around punctuation.
    text = tf.strings.regex_replace(text, '[.?!,¿]', r' \0 ')
    # Strip whitespace.
    text = tf.strings.strip(text)

    text = tf.strings.join(['[START]', text, '[END]'], separator=' ')

    return text

#=====================================================
# 2.2 - Prepare data for seq2seq model
#=====================================================
# The source data stills the same
# Target sentences need to have 2 shapes
#   - The input sentences for the decoder contain only the [START] token
#   - The ouput sentences contain only the [END] token
def process_text(source, target, source_text_processor, target_text_processor):
```

```
    source = source_text_processor(source).to_tensor()
    target = target_text_processor(target)
    targ_in = target[:,:-1].to_tensor()
    targ_out = target[:,1:].to_tensor()
    return (source, targ_in), targ_out

#=====================================================
# 2.3 - Split the training dataset into two parts.(validation part: 10% and
training part: 90%)
#=====================================================
def vectorized_text(source, target, batch_size, buffer_size, max_vocab_size,
validation_split = 0.1):

    is_train = np.random.uniform(size=(len(target),)) < 1-validation_split

    train_raw = (
        tf.data.Dataset
        .from_tensor_slices((source[is_train], target[is_train]))
        .shuffle(buffer_size)
        .batch(batch_size))
    val_raw = (
        tf.data.Dataset
        .from_tensor_slices((source[~is_train], target[~is_train]))
        .shuffle(buffer_size)
        .batch(batch_size))

    # Textvectorization
    source_text_processor = tf.keras.layers.TextVectorization(
        standardize=tf_lower_and_split_punct,
        max_tokens=max_vocab_size,
        ragged=True
        )

    target_text_processor = tf.keras.layers.TextVectorization(
        standardize=tf_lower_and_split_punct,
        max_tokens=max_vocab_size,
        ragged=True
        )

    source_text_processor.adapt(train_raw.map(lambda source, target: source))
    target_text_processor.adapt(train_raw.map(lambda source, target: target))

    train_ds = train_raw.map(lambda source, target: process_text(source, target,
                             source_text_processor,
                             target_text_processor), tf.data.AUTOTUNE
```

```
                            )
    val_ds = val_raw.map(lambda source, target: process_text(source, target,
                            source_text_processor,
                            target_text_processor), tf.data.AUTOTUNE
                            )

    return train_ds, val_ds, source_text_processor, target_text_processor

#=====================================================
# 3 - Creating Seq2seq model
#=====================================================

# 3.1 - The model
#=====================================================

class Encoder(tf.keras.layers.Layer):
    def __init__(self, text_processor, units):
        super(Encoder, self).__init__()
        self.text_processor = text_processor
        self.vocab_size = text_processor.vocabulary_size()
        self.units = units

        # The embedding layer converts tokens to vectors
        self.embedding = tf.keras.layers.Embedding(self.vocab_size, units,
                                                   mask_zero=True)

        # The RNN layer processes those vectors sequentially.
        self.rnn = tf.keras.layers.Bidirectional(
                                merge_mode='sum',
                                layer=tf.keras.layers.GRU(units,
                                # Return the sequence and state
                                return_sequences=True,
                                recurrent_initializer='glorot_uniform'))

    def call(self, x):

    # The embedding layer looks up the embedding vector for each token.
        x = self.embedding(x)

    # The GRU processes the sequence of embeddings.
        x = self.rnn(x)

    # Returns the new sequence of embeddings.
        return x
```

```
    def convert_input(self, texts):
        texts = tf.convert_to_tensor(texts)
        if len(texts.shape) == 0:
            texts = tf.convert_to_tensor(texts)[tf.newaxis]
        context = self.text_processor(texts).to_tensor()
        context = self(context)
        return context

class CrossAttention(tf.keras.layers.Layer):
    def __init__(self, units, **kwargs):
        super().__init__()
        self.mha = tf.keras.layers.MultiHeadAttention(key_dim=units, num_heads=1,
**kwargs)
        self.layernorm = tf.keras.layers.LayerNormalization()
        self.add = tf.keras.layers.Add()

    def call(self, x, context):

        attn_output, attn_scores = self.mha(
            query=x,
            value=context,
            return_attention_scores=True)

        # Cache the attention scores for plotting later.
        attn_scores = tf.reduce_mean(attn_scores, axis=1)
        self.last_attention_weights = attn_scores

        x = self.add([x, attn_output])
        x = self.layernorm(x)

        return x

class Decoder(tf.keras.layers.Layer):
    @classmethod
    def add_method(cls, fun):
        setattr(cls, fun.__name__, fun)
        return fun

    def __init__(self, text_processor, units):
        super(Decoder, self).__init__()
        self.text_processor = text_processor
        self.vocab_size = text_processor.vocabulary_size()
        self.word_to_id = tf.keras.layers.StringLookup(
            vocabulary=text_processor.get_vocabulary(),
            mask_token='', oov_token='[UNK]')
```

```
        self.id_to_word = tf.keras.layers.StringLookup(
            vocabulary=text_processor.get_vocabulary(),
            mask_token='', oov_token='[UNK]',
            invert=True)
        self.start_token = self.word_to_id('[START]')
        self.end_token = self.word_to_id('[END]')

        self.units = units


        # Convert token IDs to vectors
        self.embedding = tf.keras.layers.Embedding(self.vocab_size,
                                                   units, mask_zero=True)

        # The RNN keeps track of what's been generated so far.
        self.rnn = tf.keras.layers.GRU(units,
                                       return_sequences=True,
                                       return_state=True,
                                       recurrent_initializer='glorot_uniform')

        # The RNN output will be the query for the attention layer.
        self.attention = CrossAttention(units)

        # Produces the logits (one-hot encoding) for each output token.
        self.output_layer = tf.keras.layers.Dense(self.vocab_size)

    def call(self,
             context, x,
             state=None,
             return_state=False):

        # Lookup the embeddings
        x = self.embedding(x)

        # Process the target sequence.
        x, state = self.rnn(x, initial_state=state)

        # Use the RNN output as the query for the attention over the context.
        x = self.attention(x, context)
        self.last_attention_weights = self.attention.last_attention_weights

        # Generate logit predictions for the next token.
        logits = self.output_layer(x)

        if return_state:
```

```
            return logits, state
        else:
            return logits

    def get_initial_state(self, context):
        batch_size = tf.shape(context)[0]
        start_tokens = tf.fill([batch_size, 1], self.start_token)
        done = tf.zeros([batch_size, 1], dtype=tf.bool)
        embedded = self.embedding(start_tokens)
        return start_tokens, done, self.rnn.get_initial_state(embedded)[0]

    def tokens_to_text(self, tokens):
        words = self.id_to_word(tokens)
        result = tf.strings.reduce_join(words, axis=-1, separator=' ')
        result = tf.strings.regex_replace(result, '^ *\[START\] *', '')
        result = tf.strings.regex_replace(result, ' *\[END\] *$', '')
        return result

    def get_next_token(self, context, next_token, done, state, temperature =
0.0):
        logits, state = self(
            context, next_token,
            state = state,
            return_state=True)

        if temperature == 0.0:
            next_token = tf.argmax(logits, axis=-1)
        else:
            logits = logits[:, -1, :]/temperature
            next_token = tf.random.categorical(logits, num_samples=1)

        # If a sequence produces an `end_token`, set it `done`
        done = done | (next_token == self.end_token)
        # Once a sequence is done it only produces 0-padding.
        next_token = tf.where(done, tf.constant(0, dtype=tf.int64), next_token)

        return next_token, done, state

class Translator(tf.keras.Model):
    @classmethod
    def add_method(cls, fun):
        setattr(cls, fun.__name__, fun)
        return fun

    def __init__(self, units,
```

```
            context_text_processor,
            target_text_processor):
    super().__init__()
    # Build the encoder and decoder
    encoder = Encoder(context_text_processor, units)
    decoder = Decoder(target_text_processor, units)

    self.encoder = encoder
    self.decoder = decoder

def call(self, inputs):
    context, x = inputs
    context = self.encoder(context)
    logits = self.decoder(context, x)

    try:
    # Delete the keras mask, so keras doesn't scale the loss+accuracy.
        del logits._keras_mask
    except AttributeError:
        pass
    return logits


def translate(self,
            texts, *,
            max_length=50,
            temperature=0.0):
    # Process the input texts
    context = self.encoder.convert_input(texts)
    batch_size = tf.shape(texts)[0]

    # Setup the loop inputs
    tokens = []
    attention_weights = []
    next_token, done, state = self.decoder.get_initial_state(context)

    for _ in range(max_length):
        # Generate the next token
        next_token, done, state = self.decoder.get_next_token(
            context, next_token, done,  state, temperature)

        # Collect the generated tokens
        tokens.append(next_token)
        attention_weights.append(self.decoder.last_attention_weights)
```

```
            if tf.executing_eagerly() and tf.reduce_all(done):
                break

        # Stack the lists of tokens and attention weights.
        tokens = tf.concat(tokens, axis=-1)
        self.last_attention_weights = tf.concat(attention_weights, axis=1)

        result = self.decoder.tokens_to_text(tokens)
        return result

    def plot_attention(self, text, title, **kwargs):
        assert isinstance(text, str)
        output = self.translate([text], **kwargs)
        output = output[0].numpy().decode()

        attention = self.last_attention_weights[0]

        context = tf_lower_and_split_punct(text)
        context = context.numpy().decode().split()

        output = tf_lower_and_split_punct(output)
        output = output.numpy().decode().split()[1:]

        fig = plt.figure(figsize=(10, 10))
        ax = fig.add_subplot(1, 1, 1)

        ax.matshow(attention, cmap='viridis', vmin=0.0)

        fontdict = {'fontsize': 14}

        ax.set_xticklabels([''] + context, fontdict=fontdict, rotation=90)
        ax.set_yticklabels([''] + output, fontdict=fontdict)

        ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
        ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

        ax.set_xlabel('Input text')
        ax.set_ylabel('Output text')
        plt.savefig(f'{title}.png')
        plt.close()

#==================================================
# 3.2 - Defining metrics
#==================================================
def masked_loss(y_true, y_pred):
```

```
    # Calculate the loss for each item in the batch.
    loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(
        from_logits=True, reduction='none')
    loss = loss_fn(y_true, y_pred)

    # Mask off the losses on padding.
    mask = tf.cast(y_true != 0, loss.dtype)
    loss *= mask

    # Return the total.
    return tf.reduce_sum(loss)/tf.reduce_sum(mask)

def masked_acc(y_true, y_pred):
    # Calculate the loss for each item in the batch.
    y_pred = tf.argmax(y_pred, axis=-1)
    y_pred = tf.cast(y_pred, y_true.dtype)

    match = tf.cast(y_true == y_pred, tf.float32)
    mask = tf.cast(y_true != 0, tf.float32)

    return tf.reduce_sum(match)/tf.reduce_sum(mask)
```

```
'''Model Evaluation'''

import numpy as np
import pandas as pd
from nltk.translate.bleu_score import corpus_bleu
import string
import re

exclude = set(string.punctuation) # Set of all special characters
remove_digits = str.maketrans('', '', string.digits)
def preprocess_translated_sentence(sent):
    '''Function to preprocess English sentence'''
    sent = sent.lower() # lower casing
    sent = re.sub("'", '', sent) # remove the quotation marks if any
    sent = ''.join(ch for ch in sent if ch not in exclude)
    sent = sent.translate(remove_digits) # remove the digits
    sent = sent.strip()
    sent = re.sub(" +", " ", sent) # remove extra spaces

    return sent.split()

def calculate_bleu(path='results_test.csv'):
```

```
    results = pd.read_csv(path, sep='\t')
    translations = results['translation']
    ground_truth = results['target']
    translations = translations.apply(lambda x:
preprocess_translated_sentence(x)).to_numpy()
    ground_truth = ground_truth.apply(lambda x:
preprocess_translated_sentence(x)).to_numpy()
    corpus = translations.reshape(-1, 1)
    return corpus_bleu(corpus, ground_truth)
```

```
'''Run this file to translate sentences using the RNN model'''

import tensorflow as tf
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from RNN_attention import *
from model_eval import *
np.random.seed(42)
tf.random.set_seed(56)

if __name__=='__main__':

    print('Loading data')
    print('_'*50)
    target, source = load_data('data/train.csv')
    target_test, input_test = load_data('data/test.csv')
    print('Source shape:', source.shape)
    print('Test shape:', target_test.shape)
    print('_'*50)

    # Creating  tf.dataset
    print('\n')
    print('Creating tf.dataset')
    print('_'*50)

    BUFFER_SIZE = len(source)
    BATCH_SIZE = 64
    max_vocab_size = 20000
    UNITS = 256
    EPOCHS = 50
    STEPS = 150
    VAL_STEPS = 30
```

```
train_ds, val_ds, source_text_processor, target_text_processor = \
                    vectorized_text(source,
                    target, BATCH_SIZE, BUFFER_SIZE,
                    max_vocab_size, validation_split = 0.1)


for (ex_context_tok, ex_tar_in), ex_tar_out in train_ds.take(1):
    print('Train dataset')
    print('_'*50)
    print('Source tensor shape: ', ex_context_tok.numpy().shape)
    print('Target in tensor shape: ', ex_tar_in.numpy().shape)
    print('Target out tensor shape: ', ex_tar_out.numpy().shape)

for (ex_context_tok, ex_tar_in), ex_tar_out in val_ds.take(1):
    print()
    print('Test dataset')
    print('_'*50)
    print('Source tensor shape: ', ex_context_tok.numpy().shape)
    print('Target in tensor shape: ', ex_tar_in.numpy().shape)
    print('Target out tensor shape: ', ex_tar_out.numpy().shape)


model = Translator(UNITS, source_text_processor, target_text_processor)

model.compile(optimizer='adam',
            loss=masked_loss,
            metrics=[masked_acc, masked_loss])

model.evaluate(val_ds, steps=VAL_STEPS, return_dict=True)
print()
print('_'*50)
model.summary()

history = model.fit(
    train_ds.repeat(),
    epochs=EPOCHS,
    steps_per_epoch = STEPS,
    validation_data=val_ds,
    validation_steps = VAL_STEPS,
    callbacks=[
        tf.keras.callbacks.EarlyStopping(monitor='val_loss',patience=3)])


model.save('RNN_model')
print()
```

```
    print('_'*50)
    print('Translating the test data')
    translations = model.translate(tf.constant(input_test), max_length = 50)

    df = pd.DataFrame(data={"spanish": input_test, "translation": translations,
'target': target_test})
    df.to_csv("results_test_RNN.csv", sep='\t', index=False)

    #Plots
    #plot attention
    print()
    print('_'*50)
    print('Plotting attention graphs')
    model.plot_attention('Regresaré en tres horas', title = 'att1')
    model.plot_attention('El médico me dice que voy a tener que permanecer una
semana en reposo absoluto.', title = 'att2')

    print()
    print('_'*50)
    print('Plotting model loss')
    plt.figure(1)
    plt.plot(history.history['loss'], 'r-',label='loss')
    plt.plot(history.history['val_loss'], 'g-*', label='val_loss')
    plt.ylim([0, max(plt.ylim())])
    plt.xlabel('Epoch #')
    plt.ylabel('loss')
    plt.legend()
    plt.savefig('model_loss.png')

    print()
    print('_'*50)
    print('Plotting model accuracy')
    plt.figure(2)
    plt.plot(history.history['masked_acc'], 'r-',label='accuracy')
    plt.plot(history.history['val_masked_acc'],'g-*', label='val_accuracy')
    plt.ylim([0, max(plt.ylim())])
    plt.xlabel('Epoch #')
    plt.ylabel('accuracy')
    plt.legend()
    plt.savefig('model_acc.png')

    # Evaluating results
    print('The BLEU score is:', calculate_bleu(path='results_test_RNN.csv'))
```

A2. The Transformer model

The requirements to run the model are Python 3.6, tensorflow-2.6 tensorflow-text-2.6
This model was run on the server, use GPU=2.

```
'''This file contains all functions used in the Transformer model'''

import pathlib
import random
import string
import re
import io
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras
import tensorflow_text as tf_text
from matplotlib import pyplot as plt
from tensorflow.keras import layers
from tensorflow.keras.layers.experimental.preprocessing import TextVectorization


# Define parameters for Text vectorization
#=====================================================
VOCAB_SIZE = 20000
SEQUANCE_LENGTH = 70
BATCH_SIZE = 64
#BATCH_SIZE = 128
#=====================================================

#=====================================================
# 1. loading the data
#=====================================================
# - Read from training.csv, testing.csv
# - Read each line and split it to spanish, english parts then add [start] &
[end] token to the target text(english)
# - Add the text pairs(spanish, english) in the train.csv to the Train_pairs
# - Add the text pairs(spanish, english) in the test.csv to the Test_pairs
def loadingData(path, sample = 1000):
    lines = io.open(path, encoding='UTF-8').read().strip().split('\n')
    text_pairs = []
    #print(len(lines))
    if sample == 0:
        sample = len(lines)
    for line in lines[:sample]:
```

```
        eng, spa = line.split("\t")
        text_pairs.append((spa, eng))
    return text_pairs

#=====================================================
# 2. Preprocessing
#=====================================================

# 2.1 - Split the training dataset into two parts.(validation part: 10% and
training part: 90%)
#=====================================================
def splitTrainingData(text_pairs, pers = 0.10):
    random.shuffle(text_pairs)
    num_val_samples = int(pers * len(text_pairs))
    val_pairs = text_pairs[:num_val_samples]
    train_pairs = text_pairs[num_val_samples:]
    return train_pairs, val_pairs


# 2.2 - Text Vectorizing
#=====================================================
# TextVectorization is the process of converting text into a numerical
representation, where each integer represents the index of a word in a
vocabulary.
# we will implement TextVectorization for both English and Spanish sentence.
#- The English layer will use the default string standardization which will
remove punctuation characters and split on whitespace.
#- The Spanish layer will use a custom vectorization, where we add the character
`"¿"` to the set of punctuation characters to be stripped.'''
def preprocess_clean(text, addToken= False):
    # change accented characters
    text = tf_text.normalize_utf8(text, 'NFKD')
    text = tf.strings.lower(text)
    # Keep space, a to z, and select punctuation.
    text = tf.strings.regex_replace(text, '[^ a-z.?!,¿]', '')
    # Add spaces around punctuation.
    text = tf.strings.regex_replace(text, '[.?!,¿]', r' \0 ')
    # Strip whitespace.
    text = tf.strings.strip(text)
    if addToken:
        text = "[start] " + text + " [end]"

    return text

def spa_CustomStadardize(text):
```

```
    # change accented characters
    return preprocess_clean(text)

def eng_CustomStadardize(text):
    # change accented characters
    return preprocess_clean(text, True)

# Implement Text vectorization for the Spanish text
spa_vectorization = TextVectorization(
    max_tokens=VOCAB_SIZE,
    output_mode="int",
    output_sequence_length=SEQUANCE_LENGTH,
    standardize = spa_CustomStadardize,

)

# Implement Text vectorization for the English text
eng_vectorization = TextVectorization(
    max_tokens=VOCAB_SIZE, output_mode="int",
    output_sequence_length=SEQUANCE_LENGTH +1,
    standardize = eng_CustomStadardize,)

# 2.3 - Preparing the input data for Sequence-to-sequence Transformer:
#=====================================================
# Transformer consist of the Encoder which reads the source sequence and produces
an encoded representation of it.
# - we will reformat the training & validation set to be able to use it as the
input for the encoder
# - the training dataset will return a tuple `(inputs, targets)`, where:
# - `inputs` is a dictionary with the keys `encoder_inputs` and `decoder_inputs`.
# - The `encoder_inputs` is the vectorized source sentence (Spanish sentence) and
`decoder_inputs` is the (english sentence) which doesn't include the last token
to keep inputs and targets at the same length.
# - The target which is (english sentence) is one step ahead with out the last
token.
# - Both are still the same length.

def preprocess_batch(spa, eng):
    spa = spa_vectorization(spa)
    eng = eng_vectorization(eng)
    return ({"encoder_inputs": spa, "decoder_inputs": eng[:, :-1],}, eng[:, 1:])

def make_dataset(pairs):
    spa_texts, eng_texts = zip(*pairs)
    spa_texts = list(spa_texts)
```

```
    eng_texts = list(eng_texts)

    dataset = tf.data.Dataset.from_tensor_slices((spa_texts, eng_texts))
    dataset = dataset.batch(BATCH_SIZE)
    dataset = dataset.map(preprocess_batch, tf.data.AUTOTUNE)
    dataset = dataset.shuffle(2048)
    # Use in-memory caching to speed up preprocessing
    return dataset.prefetch(buffer_size=tf.data.AUTOTUNE).cache()


# 3 - Creating Transformer componenets
#=====================================================
class EncoderLayer(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads,  dropout_rate=0.1):
        super(EncoderLayer, self).__init__()
        self.embed_dim = embed_dim
        self.dense_dim = dense_dim
        self.num_heads = num_heads
        # attention  # we can add drop out here
        self.attention = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim
        )

        # Feed Forword layer # we can add drop out here
layers.Dropout(dropout_rate)
        self.dense_proj = keras.Sequential(
            [layers.Dense(dense_dim, activation="relu"),
                layers.Dense(embed_dim),]
        )

        # Normalization
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()

        # Dropout for the point-wise feed-forward network.
        self.dropout1 = tf.keras.layers.Dropout(dropout_rate)
        self.supports_masking = True

    def call(self, inputs, mask=None):
        if mask is not None:
            padding_mask = tf.cast(mask[:, tf.newaxis, tf.newaxis, :],
dtype="int32")
        else:
            padding_mask = None
```

```
        # self-attention take three tensers(Query Q: tensor, Value V: tensor, Key
K: tensor)
        attention_output = self.attention(
            query=inputs, value=inputs, key=inputs, attention_mask=padding_mask
        )
        proj_input = self.layernorm_1(inputs + attention_output)
        # feed-forward network output.
        proj_output = self.dense_proj(proj_input)
        return self.layernorm_2(proj_input + proj_output)

    def get_config(self):
        config = super().get_config()
        config.update({
            "embed_dim": self.embed_dim,
            "dense_dim": self.dense_dim,
            "num_heads": self.num_heads,
        })
        return config



def positional_encoding(length, depth):
    depth = depth/2

    positions = np.arange(length)[:, np.newaxis]     # (seq, 1)
    depths = np.arange(depth)[np.newaxis, :]/depth   # (1, depth)

    angle_rates = 1 / (10000**depths)         # (1, depth)
    angle_rads = positions * angle_rates      # (pos, depth)

    pos_encoding = np.concatenate([np.sin(angle_rads), np.cos(angle_rads)],
axis=-1)

    return tf.cast(pos_encoding, dtype=tf.float32)

class PositionalEmbedding(layers.Layer):
    def __init__(self, sequence_length, vocab_size, embed_dim, **kwargs):
        super(PositionalEmbedding, self).__init__(**kwargs)
        self.sequence_length = sequence_length
        self.vocab_size = vocab_size
        self.embed_dim = embed_dim
        # token
        self.token_embeddings = layers.Embedding(input_dim=vocab_size,
output_dim=embed_dim, mask_zero=True)
```

```
        self.embedded_positions = positional_encoding(length=2048,
depth=embed_dim)

    def compute_mask(self, *args, **kwargs):
        return self.token_embeddings.compute_mask(*args, **kwargs)

    def call(self, inputs):
        length = tf.shape(inputs)[1]
        embedded_tokens = self.token_embeddings(inputs)
        embedded_tokens *= tf.math.sqrt(tf.cast(self.embed_dim, tf.float32))
        result = embedded_tokens + self.embedded_positions[tf.newaxis, :length,
:]
        return result



class DecoderLayer(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads,  dropout_rate=0.1,
**kwargs):
        super(DecoderLayer, self).__init__(**kwargs)
        self.embed_dim = embed_dim
        self.dense_dim = dense_dim
        self.num_heads = num_heads
        # attention
        # Each multi-head attention block gets three inputs; Q (query), K (key),
V (value).
        self.attention_1 = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim
        )
        self.attention_2 = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim # Size of each attention head
for query Q and key K.
        )

        #feed forword
        self.dense_proj = keras.Sequential(
            [layers.Dense(dense_dim, activation="relu"),
layers.Dense(embed_dim),]
        )

        # Layer normalization.
        #self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()
        self.layernorm_3 = layers.LayerNormalization()
```

```
        # Dropout for the point-wise feed-forward network.
        self.dropout1 = tf.keras.layers.Dropout(dropout_rate)
        self.supports_masking = True


    def get_causal_attention_mask(self, inputs):
        input_shape = tf.shape(inputs)
        batch_size, sequence_length = input_shape[0], input_shape[1]
        i = tf.range(sequence_length)[:, tf.newaxis]
        j = tf.range(sequence_length)
        mask = tf.cast(i >= j, dtype="int32")
        mask = tf.reshape(mask, (1, input_shape[1], input_shape[1]))
        mult = tf.concat(
            [tf.expand_dims(batch_size, -1), tf.constant([1, 1],
dtype=tf.int32)],
            axis=0,
        )
        return tf.tile(mask, mult)

    def call(self, inputs, encoder_outputs, mask=None):
        causal_mask = self.get_causal_attention_mask(inputs)
        if mask is not None:
            padding_mask = tf.cast(mask[:, tf.newaxis, :], dtype="int32")
            padding_mask = tf.minimum(padding_mask, causal_mask)

        attention_output_1 = self.attention_1(
            query=inputs, value=inputs, key=inputs, attention_mask=causal_mask
        )
        out_1 = self.layernorm_1(inputs + attention_output_1)

        #Each multi-head attention block gets three inputs; Q (query), K (key), V
(value).
        attention_output_2 = self.attention_2(
            query=out_1,
            value=encoder_outputs,
            key=encoder_outputs,
            attention_mask=padding_mask,
        )
        out_2 = self.layernorm_2(out_1 + attention_output_2)

        # final layer
        proj_output = self.dense_proj(out_2)
        return self.layernorm_3(out_2 + proj_output)
```

```
    def get_config(self):
        config = super().get_config()
        config.update({
            "embed_dim": self.embed_dim,
            "latent_dim": self.latent_dim,
            "num_heads": self.num_heads,
        })
        return config


# Create Transformer model (encoder, decoder)
def CreateTransformerModel(embed_dim, latent_dim, num_heads, dropout_rate=0.1):
    # ==============================
    # Encoder
    # ==============================
    # 1. prepare the input for the encoder
    encoder_inputs = keras.Input(shape=(None,), dtype="int64",
name="encoder_inputs")
    # 2. implement PositionalEmbedding for the encoder input
    x = PositionalEmbedding(SEQUANCE_LENGTH, VOCAB_SIZE,
embed_dim)(encoder_inputs)
    # 3. Create Encoder layer
    encoder_outputs = EncoderLayer(embed_dim, latent_dim, num_heads,
dropout_rate)(x)
    # 4. Create Encoder model which take encoderInput & encoder outputs
    encoderModel = keras.Model(encoder_inputs, encoder_outputs)

    # ==============================
    # Decoder
    # ==============================
    # 1. prepare the input for the decoder
    decoder_inputs = keras.Input(shape=(None,), dtype="int64",
name="decoder_inputs")
    encoded_seq_inputs = keras.Input(shape=(None, embed_dim),
name="decoder_state_inputs")
    # 2. implement PositionalEmbedding for the decoder input
    x = PositionalEmbedding(SEQUANCE_LENGTH, VOCAB_SIZE,
embed_dim)(decoder_inputs)

    # 3. Create decoder layer
    x = DecoderLayer(embed_dim, latent_dim, num_heads, dropout_rate)(x,
encoded_seq_inputs)

    x = layers.Dropout(dropout_rate)(x)
```

```
    # Create decoder outputlayer
    decoder_outputs = layers.Dense(VOCAB_SIZE, activation="softmax")(x)
    # 4. Create decoder model which take Decoder_Inputs & encoded_seq_inputs,
decoder_outputs
    decoderModel = keras.Model([decoder_inputs, encoded_seq_inputs],
decoder_outputs)

    decoder_outputs = decoderModel([decoder_inputs, encoder_outputs])

    # create transforer moder
    transformer = keras.Model(
        [encoder_inputs, decoder_inputs], decoder_outputs, name="transformer"
    )
    return transformer



# Create new loss & accuracy for the padded target sequences
#============================================================
# we should apply a padding mask when calculating the loss. by Using the cross-
entropy loss function
def masked_loss(real, pred):
    mask = real != 0
    loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True,
reduction='none')
    loss = loss_object(real, pred)

    mask = tf.cast(mask, dtype=loss.dtype)
    loss *= mask

    loss = tf.reduce_sum(loss)/tf.reduce_sum(mask)
    return loss


def masked_accuracy(real, pred):
    pred = tf.argmax(pred, axis=2)
    real = tf.cast(real, pred.dtype)
    match = real == pred

    mask = real != 0

    match = match & mask

    match = tf.cast(match, dtype=tf.float32)
    mask = tf.cast(mask, dtype=tf.float32)
```

```
    return tf.reduce_sum(match)/tf.reduce_sum(mask)

# Train Tansformer model
#=================================================
# 1. Create the transformer model (Encode, Decoder)
# 2. Compile the model
# 3. Train the model
def trainModel(transformer, epochs, train_data, Val_data):
    #epochs = 1  # This should be at least 30 for convergence
    # hyper parameterparameters for the transformer

    # we can try this
    #learning_rate = CustomSchedule(d_model)
    Adam_optimizer = tf.keras.optimizers.Adam(learning_rate = 0.001, beta_1=0.9,
beta_2=0.98, epsilon=1e-9)
    #Adam_optimizer = tf.keras.optimizers.Adam(lr = 0.001)
    #transformer.compile(Adam_optimizer, loss="sparse_categorical_crossentropy",
metrics=["accuracy"])

    callbacks_list = [
        tf.keras.callbacks.EarlyStopping(monitor='masked_loss', patience=3,
mode='min', verbose=2),
        # save the result into  csv
        tf.keras.callbacks.CSVLogger('Transformer_logs.csv', separator=",",
append=False)
    ]

    # Compile the model
    transformer.compile(Adam_optimizer, loss=masked_loss,
metrics=[masked_accuracy, masked_loss])
    print(transformer.summary())
    history = transformer.fit(train_data.repeat(), epochs=epochs, steps_per_epoch
= 700, validation_data=Val_data, callbacks=[callbacks_list], verbose = 2)
    # history = transformer.fit(train_data, epochs=epochs,
validation_data=Val_data, callbacks=[callbacks_list], verbose = 2)
    return transformer, history

# plot Accurancy & loss vs epoch
def plotModel(data1, data2, xlabel, ylabel, Title, figN):
    #fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(20,8))
    plt.figure(figN)
    plt.plot(data1, 'r-', label= ylabel)                # exp: accuracy
    plt.plot(data2, 'g-*',label = 'Val_'+ylabel )               # exp:
val_accuracy
    plt.title(Title)
```

```
    plt.ylim([0, max(plt.ylim())])
    plt.xlabel('Epoch')
    plt.ylabel(ylabel)
    plt.legend()
    #plt.show()
    picFile = Title+".png"
    plt.savefig(picFile)

# Testing the model
# Testing the model
#=================================
# We feed into the Transformer model the vectorized Spanish sentence with target
token `"[start]"`,
# then we repeatedly generated the next token, until we reach the token
`"[end]"`.
def Translater(transformer, input_sentence, eng_index_vocab):
    # input tokanized sentence
    spa_tokenized_input_sentence = spa_vectorization([input_sentence]) # take the
tokanization for spa sentence
    #print("spa_tokenized_input_sentence: ",spa_tokenized_input_sentence)
    decoded_sentence = "[start]"
    #print("eng_tokenized_target_sentence: ",eng_tokenized_target_sentence)
    #spa_tokenized_input_sentence:  tf.Tensor
    for i in range(SEQUANCE_LENGTH):
        eng_tokenized_target_sentence = eng_vectorization([decoded_sentence])[:,
:-1]
        predictions = transformer([spa_tokenized_input_sentence,
eng_tokenized_target_sentence])
        # Select the last token from the `seq_len` dimension.
        #predicted_id = np.argmax(predictions[0, i, :])
        # select the last word from the seq_len dimension
        predicted_id = np.argmax(predictions[0, i, :])
        predicted_token = eng_index_vocab[predicted_id]
        #print("predicted", i, predicted_token)
        # Add all translated token to the decoded_sentence
        decoded_sentence += " " + predicted_token

        if (predicted_token == "[end]") :
            break
    #print("decoded_sentence", decoded_sentence)
    return decoded_sentence

# Test_EvaluateModel read from test data and loop through the source text (spa)
# then decode the source text to get the translated text as a result for the
transformer model
```

```
# we remove the extra token like start & end from the tranlated sentence
# save the source sentence , translated sentence and the actual sentence(eng) in
a lists
# save the result into a csv file
def Test_EvaluateModel(transformer, test_data):
    # target sentence eng
    eng_vocab = eng_vectorization.get_vocabulary()
    eng_index_vocab = dict(zip(range(len(eng_vocab)), eng_vocab))

    sourceList = []      # save the sentence that we want to translated (spa)
    translatedList = []  # save the sentence after translated (eng)
    actualList = []  # save the actual translated sentence from test data

    test_spa_texts = [pair[0] for pair in test_data]
    test_eng_texts = [pair[1] for pair in test_data]
    for i,text in enumerate(test_spa_texts):
        input_sentence = text
        #print("spa:",input_sentence)
        translated = Translater(transformer, input_sentence, eng_index_vocab)
        #remove end and start token after translation
        translated = (
            translated.replace("[start]",
"")
            .replace("[end]", "")
            .replace(" . . ", ".")
            .strip()
        )
        sourceList.append(input_sentence)
        translatedList.append(translated)
        # get the Actual english sentence in the english test list
        eng_text = test_eng_texts[i]
        actualList.append(eng_text)
    return sourceList, translatedList, actualList


# save to csv file
def savetoCSV(sourceList, predictedList, actualList, ResultFile ):
    # dictionary of lists
    #translatedDict = {"original":[], "translate": [], "referance": []}
    translatedDict = {'text': sourceList, 'predicted': predictedList, 'actual':
actualList}

    df = pd.DataFrame(translatedDict)

    # saving the dataframe
```

```
    df.to_csv(ResultFile+'.csv')
    return dict
```

```
'''Run this file to translate sentences using the transformer model from
scratch'''

import os
os.environ["CUDA_VISIBLE_DEVICES"]="2"
import transformer as tr
import time



if __name__ == "__main__":
    # get the start time
    st = time.time()
    print("="*100)
    print("Loading data...")
    print("="*100)
    #Train_pairs = tr.loadingData('data/train.csv',sample = 10000)
    Train_pairs = tr.loadingData('data/train.csv',sample = 0)
    Test_pairs = tr.loadingData('data/test.csv',sample = 0)
    print("="*100)
    print("Preprocessing...")
    print("="*100)
    # split the training dataset into (validation 10%, training 90%) sets
    print("1- Split the training dataset into Validation, Trainging...")
    print("-"*50)
    train_pairs, val_pairs = tr.splitTrainingData(Train_pairs)

    print("All training data",len(Train_pairs))
    print("training set", len(train_pairs))
    print("validation set",len(val_pairs))
    print("Test_pairs", len(Test_pairs))

    print("2- Text Vectorization...")
    print("-"*50)
    # Get the spanish text
    train_spa_texts = [pair[0] for pair in train_pairs]
    # Get the English text only
    train_eng_texts = [pair[1] for pair in train_pairs]

    # Implement vectorization
```

```
    tr.spa_vectorization.adapt(train_spa_texts)
    tr.eng_vectorization.adapt(train_eng_texts)


    print("="*100)
    print("before vectorization Spanish, \n",train_spa_texts[:3])
    print("before vectorization English,  \n",train_eng_texts[:3])
    print("="*100)


    print("3- prepare the decoder input values...")
    print("-"*50)

    train_ds = tr.make_dataset(train_pairs)
    val_ds = tr.make_dataset(val_pairs)

    for inputs, targets in train_ds.take(1):
        print("inputs[encoder_inputs].shape:", inputs["encoder_inputs"].shape)
        print("inputs[decoder_inputs].shape:", inputs["decoder_inputs"].shape)
        print("targets.shape:", targets.shape)

    print("="*100)
    print("4- Create the transformer model...")
    print("="*100)

    # hyper parameter for the transformer model
    embed_dim = 256  # dimension model we can use 512
    dense_dim = 2048
    num_heads = 8
    Epochs = 50
    #EpochSteps = 700
    dropoutRate= 0.1

    transformer = tr.CreateTransformerModel(embed_dim, dense_dim, num_heads,
dropout_rate = dropoutRate)
    transformer.summary()

    print("="*100)
    print("4- Train the transformer model...epochs:",Epochs, " dropout_rate:",
dropoutRate )
    print("="*100)
    transformer, history = tr.trainModel(transformer, epochs= Epochs, train_data
= train_ds, Val_data = val_ds)
    print("="*100)
```

```
    score = transformer.evaluate(val_ds,  steps = 20, return_dict=True, verbose =
2)
    print("Evaluation: ",score.items())
    print("="*100)
    print("Applying model on test data...")
    print("="*100)
    et = time.time()
    elapsed_time = (et - st)/60
    print("="*50)
    print('Train Execution time:', elapsed_time, 'seconds')
    st = time.time()
    # Test_EvaluateModel: take the testdata and translate all the sentence from
spa to eng
    # save the result into csv file
    sourceList, translatedList, actualList = tr.Test_EvaluateModel(transformer,
Test_pairs)
    print("Translated text:",translatedList[:3])
    print("Actual Text:",actualList[:3])
    print("="*50)

    print("Save the result in CSV file...")
    tr.savetoCSV(sourceList, translatedList, actualList, ResultFile =
'Transformer_Result')
    print("="*100)
    print("Evaluating...")
    print("="*100)


    et = time.time()
    elapsed_time = (et - st)/60
    print("="*50)
    print('Execution time:', elapsed_time, 'seconds')
    #print(history.history.keys())
    # plot Accurancy & loss vs epoch
    # print("="*100)
    print("plot accuracy. loss graph")
    tr.plotModel(history.history['masked_accuracy'],
history.history['val_masked_accuracy'], 'epoch',
'Accuracy','Transformer_Accuracy', 1)
    tr.plotModel(history.history['loss'], history.history['val_loss'], 'epoch',
'loss','Transformer_loss',2)
    tr.plotModel(history.history['masked_loss'],
history.history['val_masked_loss'], 'epoch', 'Masked-
loss','Transformer_Mloss',3)
```

```
'''Run this to evaluate the Transformer implementation from scratch'''
import numpy as np
import pandas as pd
from nltk.translate.bleu_score import corpus_bleu
from model_eval import preprocess_translated_sentence


results = pd.read_csv('Transformer_Result.csv', sep=',')

translations = results['predicted']
ground_truth = results['actual']
translations = translations.apply(lambda x:
preprocess_translated_sentence(x)).to_numpy()
ground_truth = ground_truth.apply(lambda x:
preprocess_translated_sentence(x)).to_numpy()
corpus = translations.reshape(-1, 1)
print(corpus_bleu(corpus, ground_truth))
```

A3. Fine-tuned transformer

The requirements to run the model are Python 3.8. This model was run on the server, use GPU=2. Please install the following modules:

```
pip install datasets transformers sacrebleu sacremoses torch sentencepiece
transformers[sentencepiece]
```

```
'''Run this file to fine tune and translate sentences using the Huggingface
Transformer'''


import os
os.environ["WANDB_DISABLED"]="true"
os.environ["CUDA_VISIBLE_DEVICES"]="2"
import transformers
from datasets import load_dataset, load_metric, Dataset, DatasetDict
from transformers import AutoTokenizer
import pandas as pd
import numpy as np
from transformers import AutoModelForSeq2SeqLM, DataCollatorForSeq2Seq,
Seq2SeqTrainingArguments, Seq2SeqTrainer
from transformers import MarianMTModel, MarianTokenizer
from model_eval import preprocess_translated_sentence
from nltk.translate.bleu_score import corpus_bleu

if __name__=='__main__':
    #=====================================================

    # 1. Loading pretrained model
    #=====================================================

    print('Load pretrained model')
    print('_'*50)
    model_checkpoint = 'Helsinki-NLP/opus-mt-es-en'
    metric = load_metric("sacrebleu")
    tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)


    #=====================================================

    # 2. Loading the data
    #=====================================================
    print('Loading data')
```

```
print('_'*50)

train_df = pd.read_csv('train.csv', sep='\t', names=['en', 'es'])
test_df = pd.read_csv('test.csv', sep='\t', names=['en', 'es'])

val_size = int(np.ceil(train_df.shape[0]*0.9))
val_df = train_df.iloc[val_size:,:]
train_df = train_df.iloc[:val_size,:]

print('Training size: ', train_df.shape[0])
print('Validation size: ', val_df.shape[0])
print('Test size: ', test_df.shape[0])

print('_'*50)
print('Building Huggingface dataset')
train_df = Dataset.from_pandas(train_df)
val_df = Dataset.from_pandas(val_df)
test_df = Dataset.from_pandas(test_df)

#=====================================================

# 2.2 - Creating dataset object
#=====================================================
ds =  DatasetDict()
ds['train'] = train_df
ds['validation'] = val_df
ds['test'] = test_df
print('Dataset built successfully!')
print(ds)

#=====================================================

# 3. Defining model parameters
#=====================================================
prefix = ""
max_input_length = 30
max_target_length = 30
source_lang = "es"
target_lang = "en"

#=====================================================

# 3.1 - Preprocess function to tokenize Dataset
#=====================================================
#
```

```
    def preprocess_function(examples):
        inputs = [prefix + ex for ex in examples["es"]]
        targets = [ex for ex in examples["en"]]
        model_inputs = tokenizer(inputs, max_length=max_input_length,
truncation=True)
        # Setup the tokenizer for targets
        with tokenizer.as_target_tokenizer():
            labels = tokenizer(targets, max_length=max_target_length,
truncation=True)
        model_inputs["labels"] = labels["input_ids"]
        return model_inputs


    tokenized_ds = ds.map(preprocess_function, batched=True)
    #=====================================================

    # 4 - Building the model
    #=====================================================
    #
    print('Building the model from pretrained')
    model = AutoModelForSeq2SeqLM.from_pretrained(model_checkpoint)

    batch_size = 16
    model_name = model_checkpoint.split("/")[-1]


    #=====================================================

    # 4.1 - Initializing training arguments
    #=====================================================
    #
    args = Seq2SeqTrainingArguments(
        f"{model_name}-finetuned-{source_lang}-to-{target_lang}",
        evaluation_strategy = "epoch",
        learning_rate=2e-5,
        per_device_train_batch_size=batch_size,
        per_device_eval_batch_size=batch_size,
        weight_decay=0.01,
        save_total_limit=3,
        num_train_epochs=1,
        predict_with_generate=True
    )

    data_collator = DataCollatorForSeq2Seq(tokenizer, model=model)
```

```
    def postprocess_text(preds, labels):
        preds = [pred.strip() for pred in preds]
        labels = [[label.strip()] for label in labels]
        return preds, labels
    def compute_metrics(eval_preds):
        preds, labels = eval_preds
        if isinstance(preds, tuple):
            preds = preds[0]
        decoded_preds = tokenizer.batch_decode(preds, skip_special_tokens=True)
        # Replace -100 in the labels as we can't decode them.
        labels = np.where(labels != -100, labels, tokenizer.pad_token_id)
        decoded_labels = tokenizer.batch_decode(labels, skip_special_tokens=True)
        # Some simple post-processing
        decoded_preds, decoded_labels = postprocess_text(decoded_preds,
decoded_labels)
        result = metric.compute(predictions=decoded_preds,
references=decoded_labels)
        result = {"bleu": result["score"]}
        prediction_lens = [np.count_nonzero(pred != tokenizer.pad_token_id) for
pred in preds]
        result["gen_len"] = np.mean(prediction_lens)
        result = {k: round(v, 4) for k, v in result.items()}
        return result


    #=====================================================

    # 4.2 - Training the mdoel
    #=====================================================
    #
    print()
    print('Training the model')
    trainer = Seq2SeqTrainer(
        model,
        args,
        train_dataset=tokenized_ds["train"],
        eval_dataset=tokenized_ds["validation"],
        data_collator=data_collator,
        tokenizer=tokenizer,
        compute_metrics=compute_metrics
    )

    trainer.train()

    #=====================================================
```

```
    # 5. Translating text
    #=====================================================
    # This part of the model can be run separetedly to only
    # perform translations once the model is saved
    print('Translating test data')

    test_corpus = pd.read_csv('test.csv', sep='\t', names=['en', 'es'])
    source_corpus = test_corpus['es'].to_list()
    target_corpus = test_corpus['en'].to_list()

    model_name = 'opus-mt-es-en-finetuned-es-to-en/checkpoint-7000'
    tokenizer = MarianTokenizer.from_pretrained(model_name)
    model = MarianMTModel.from_pretrained(model_name)
    translated = model.generate(**tokenizer(source_corpus, return_tensors="pt",
padding=True))

    translations = [tokenizer.decode(t, skip_special_tokens=True) for t in
translated]

    df = pd.DataFrame(data={"spanish": source_corpus, "translation":
translations, 'target': target_corpus})
    df.to_csv("results_test_fine_tuned.csv", sep='\t', index=False)

    res = pd.read_csv('results_test_fine_tuned1.csv', sep='\t')
    res_translation = res['translation']
    ground_truth = res['target']

    # making sure the sentences have the same format
    translations = res_translation.apply(lambda x:
preprocess_translated_sentence(x)).to_numpy()
    ground_truth = ground_truth.apply(lambda x:
preprocess_translated_sentence(x)).to_numpy()

    corpus = translations.reshape(-1, 1)
    BLEU_corpus = corpus_bleu(corpus, ground_truth)
    print('BLEU score for translations:', BLEU_corpus)
```