UNIVERSITY OF STAVANAGER

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

GROUP 1

NOURIN MOHAMMAD HAIDER ALI BISWAS (275116)

RIHAB HASHIM MOHAMMED AL-ZURKANI (268102)

Neural Machine Translation From Portuguese to English

ELE680 – Deep Neural Network

1. Introduction

Neural Machine Translation is the task of translating a sequence from a source language to a target language, which is also known as sequence to sequence (seq2seq) model (Chollet, 2017). There are quite a few approaches to perform machine translation, the most popular of those are RNN with encoder and decoder framework and Transformers.

In general translation tasks, the input sentence is first passed through the encoder, where is it processed by an RNN layer. In the case of machine translation, we do not focus on the output of the RNN, rather we utilize only the hidden state of the RNN. The hidden state provides information about the context of the sequence and 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. The main drawbacks to this approach are (1) RNN are slow to train, and (2) they can be inefficient when it comes to processing long sequences.

Given the drawbacks of this traditional seq2seq model, a mechanism called the Attention Model was introduced to improve the translation of sequences by focusing on certain parts of the input sentence. The reasoning behind developing the attention mechanism is to allow for more data to be passed from the encoder to the decoder. This ensures that instead of only using the last state of the encoder, all the states will be passed to each step of the decoder, which makes the information about all the elements in 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 model provides a significant improvement to the traditional seq2seq model it comes with a certain limitation. The computations are still performed in one element of the sequence at a time which can be time consuming and inefficient depending on the size of the corpus (Minh-Thang Luong, 2015).

In 2017, a transformer model (al, 2017) was introduced which completely revolutionized the use the of attention mechanism. This model uses a new mechanism called self-attention, which does not use recurrence and convolutions to perform translations. The self-attention model allows for the model to understand the underlying meaning of the language by looking at the context of the surrounding words in a sentence (Raschka, 2023). For example, “Life is short, eat dessert first!” and “Ellen is so short”, the word “short” has different meanings. In such a case, self-attention allows the model to look at the different words in each of the sentences to determine whether the word “short” is referencing a human or natural state. In the first sentence, the model might attend to the word check to determine the meaning of the word “short” while in the second sentence, it may attend to the word “Ellen” to check that we are referring to an object/human. This mechanism allows the transformers to develop a better and deeper understanding of grammar which in turn provides better results in the process of natural language processing (Cloud, 2018).

Transformers can also handle parallelized operations as they lack time dependent operations which render a more efficient model and the ability to process extremely long textual data. The model makes use of the positional encodings in addition to the input embeddings, which allows it to map each of the words before feeding the sequence to the model. This means the information is stored in the data itself rather than the network's structure. As the model is trained with big data, it learns how to interpret the position encodings and further the importance of words in other sentences.

The transformer architecture is also built following the Encoder and Decoder framework of RNN with attention models. The original transformer architecture proposed for Machine Translation is composed of 6 encoders and 6 decoders. The representation of the model is shown in Figure 1, where one encoder and one decoder are shown for simplicity.

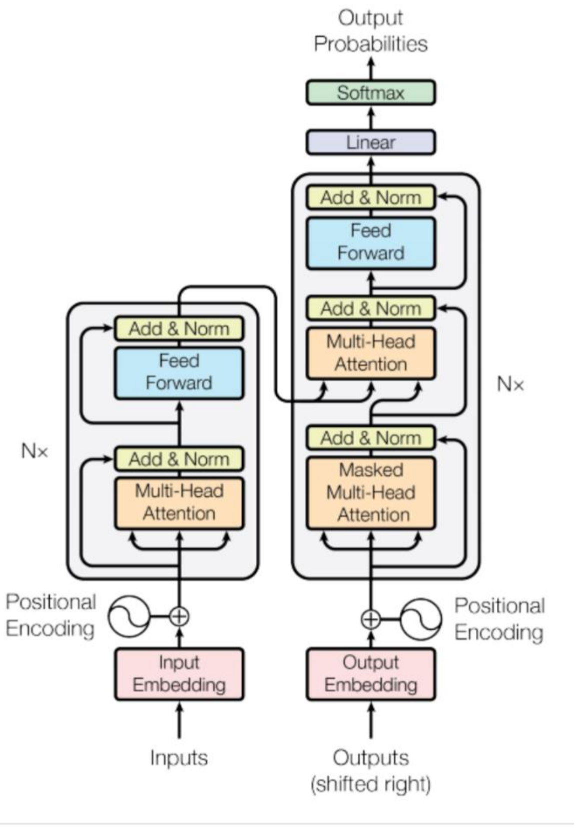


Figure 1: The Transformer Model

Each encoder model contains a self-attention layer and a feed-forward layer. The encoder takes the word in the sentences as input at the same time and passes it through the self-attention layer which will generate the attentions scores for each word against the input sentence. The attention score represents a weighted count of the amount of attention that it has picked up from the input sentence. The self-attention layer is multi-head attention, that is a module that runs through the attention mechanism multiple times in parallel and allows the model to attend to parts of the sequence differently. The purpose of the feed-forward layer is to process the input from the output of one attention layer so that it can fit better in the next attention layer.

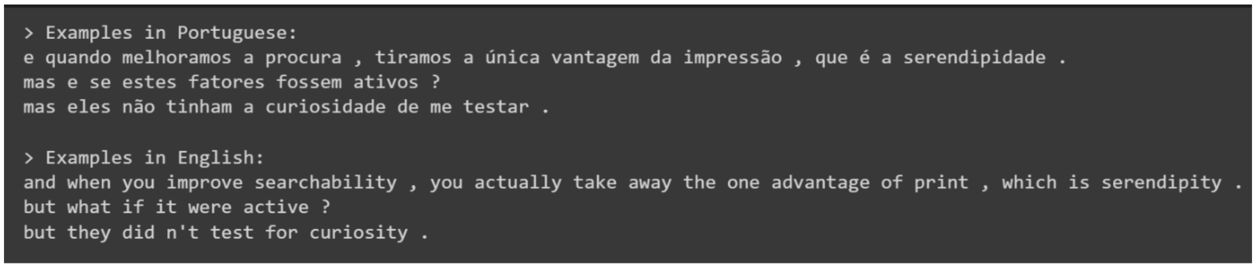
Once the source text data is encoded, the outputs of the encoder are then passed to the decoder. The encoder and decoder are connected by the cross-attention layer. The cross-attention layer connects the encoder and decoder. There are two self-attention layers in the decoder that serve a different purpose. The first multi-head attention layer takes in the positional embeddings of the target sequence. Its purpose is to prevent the decoder from having access to future tokens, as it is autoregressive. This means that the decoder processes each word in the target sequence at a time. The second multi-head attention layer is responsible for matching the encoder's 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 gives the predicted word. This process is repeated until the decoder reaches the end of the sentence.

In this report, we will use a seq2seq Transformer model to translate text from Portugese to English. The model is trained using the TensorFlow datasets (Tensorflow Dataset: 52,000 Text Pairs and tested using (1800 Test Pairs). During the model training, we monitored the model loss and accuracy and as Translation tasks are related to NLP, we decided to use the LEU score to evaluate the quality of the translations.

2. MATERIALS AND METHODS

2.1 The Data

In this project, we used the dataset provided to us through the Tensorflow Datasets [3]. The TensorFlow Dataset contains 52,000 Training, 1200 Validation, and 1800 Test sentence pairs from Portuguese to English. Figure 2 shows how the dataset is organized.



```
> Examples in Portuguese:
e quando melhoramos a procura , tiramos a única vantagem da impressão , que é a serendipidade .
mas e se estes fatores fossem ativos ?
mas eles não tinham a curiosidade de me testar .

> Examples in English:
and when you improve searchability , you actually take away the one advantage of print , which is serendipity .
but what if it were active ?
but they did n't test for curiosity .
```

Figure 2: Portuguese and English Sentence Pairs

2.2 Data Preprocessing

The first step required for Text Translation is to preprocess the data into a format suitable for the algorithm to understand the information. The preprocessing step consists of two main procedures: Data Cleaning and Text Vectorization. Data Cleaning is required to remove the unnecessary features that will not contribute to the overall results of our model. Since we are dealing with two different languages, it is expected 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 Portuguese accented letters. As the dataset provided to us was already split into a training, testing, and validation set, we did not need to perform any split from the training set.

The next step is Text Vectorization, for which we used Tensorflow to create a dataset and text.BertTokenizer to vectorize the sentence pairs. The batch size used is 64 for both models. The train and validation datasets are then used to create a text. Bertokenizer object. In this process, each sentence is split into a list of words (tokenization). For text vectorization using BertTokenize,r we performed the following steps:

1. Use custom standardized functions to clean up the sentence, normalize the text, and remove the punctuation.
2. Generate the Vocabulary using bert_vocab.bert_vocab_from_dataset which generates the vocabulary for both languages.
3. Build the tokenizer by using text.BertTokenizer for both the language text.
4. Add [START] and [END] tokens for the target sentences i.e., English Sentences only.
5. Create a word index for each token.
6. Pad each sentence to a maximum Sequence of 50.

3. Methods

3.1 The Transformer

The Transformer model consists of an Encoder which reads the source sequence and a Decoder which predicts the token in the target sequence. We adopted an implementation for a basic transformer from the “Attention is all we need” paper and updated some hyperparameters to compare the results from the original paper.

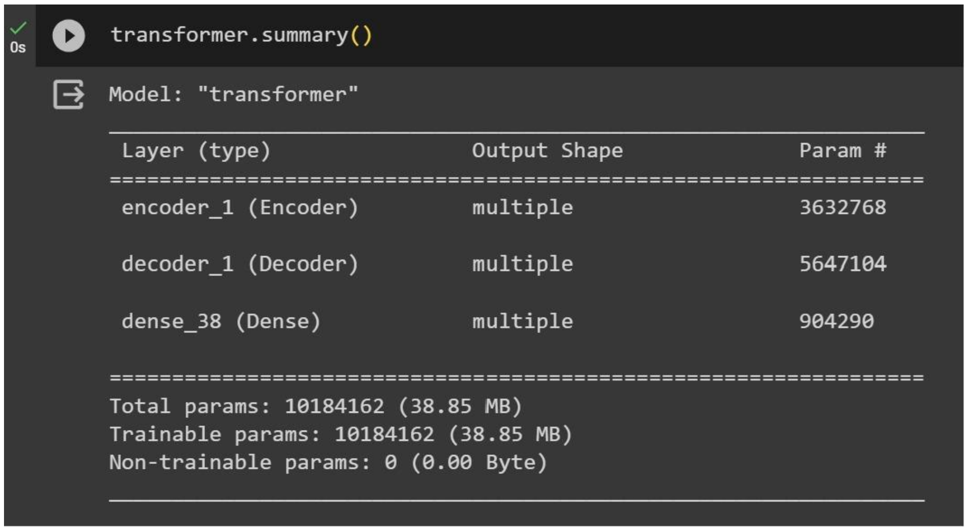
We implemented two architectures of the Transformer; the first architecture consists of one layer of Encoder and one layer of Decoder. The second architecture consists of 4 layers of Encoder and 4 layers of Decoder. This was done to understand the evolution of the translations throughout the different training of the transformers of different layers. The general structure for the transformer depends on the 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 allow the model to 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 and 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 the Portuguese 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 was generated.

The Transformer model consists of three main components (Encoder, Decoder, Final Layer, SoftMax). The encoder was created as a class which consists of the following layers:

1. Multi-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. Dropout layer with rate 0.1

The Decoder consists of self-attention (Multi-head attention), a Normalizing layer, a Feed Forward layer, and dropout (0.1). The Decoder class has the same layers for the encoder, with an extra multi-head attention layer and three layers for normalization with dropout later with rate = 0.1. Finally, the transformer model class combines the encoder and decoder the output of the decoder is the input for the final linear (Dense) layer which is a fully connected later those projects the vector produced by the decoder, into a large vector (logits) with the same size for the vocabulary (20000), each cell will contain a score of a unique word, then the SoftMax turns those scores into a probability where the highest number assigns the correspondent word. We used an 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 and loss for each epoch to a CSV file. The Early stopping was assigned patience = 3 which means that when the validation loss starts to increase for 10 (epochs). The total parameters for the model are shown in Figure 3 below:


```
transformer.summary()
```
```
Model: "transformer"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 encoder_1 (Encoder)         multiple                  3632768

 decoder_1 (Decoder)         multiple                  5647104

 dense_38 (Dense)            multiple                  904290

=================================================================
Total params: 10184162 (38.85 MB)
Trainable params: 10184162 (38.85 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 3: The total parameters of the Transformer Model

To produce the Translations, we pass a vectorized Portuguese 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 generates the next token until it reached the end of the token [END]. Once the translation is complete, the extra tokens are removed from translated sentences and are displayed in reference to the ground truth to see how our model will work.

3.2 Fine Tuned Transformer

To compare our results of the Transformer from scratch, we used a pre-trained transformer model to translate the sentences into English. The model used the Hugging Face Framework. We used the “**unicamp-dl/translation-pt-en-t5**" benchmark to fine-tune the model. The trained 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 found in the Hugging Face documentation:

1. Preprocess the data using Transformers AutoTokenizer and truncate the sentences to ensure that there are no inputs longer than the model can handle.
2. Use the pre-trained AutoModelForSeq2SeqLM class to call the model.
3. Instantiate the Training Arguments using the class model to an assigned folder. We also assign the learning rate, batch size, and the weight decay.
4. Pass the data to the Seq2SeqTrainer to fine-tune.
5. Translate the text.

3.3 Evaluation Metrics

BLEU (Bilingual Evaluation Understudy) is a metric used to evaluate the quality of machine-generated text, particularly in the context of machine translation. It was originally designed to assess the quality of translation produced by Machine Translation systems. BLEU measures the similarity between a machine-generated text and one or more reference texts. The score ranges from 0 to 1 (or 0 to 100), where 1 represents a perfect match and 0 a complete mismatch between the sentences. The BLEU score is calculated by taking the geometric mean of the precision scores for different n-gram sizes.

The general idea is to evaluate the sentences by counting n-grams. First, we calculate the frequency at which 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 as the highest frequency between the references. It’s important to note that the BLEU has some limitations, such as not considering word order and fluency, and it can favor shorter translations. We used the NLTK’s library to get the BLEU score for the sentence piece.

4. RESULTS

In the Transformer model, we kept the number of Epochs = 50, same for both layers as we observed that the training for more epochs did not improve the accuracy. As the number of layers were increased from 1 to 4 layers, we observed that the accuracy also improved.

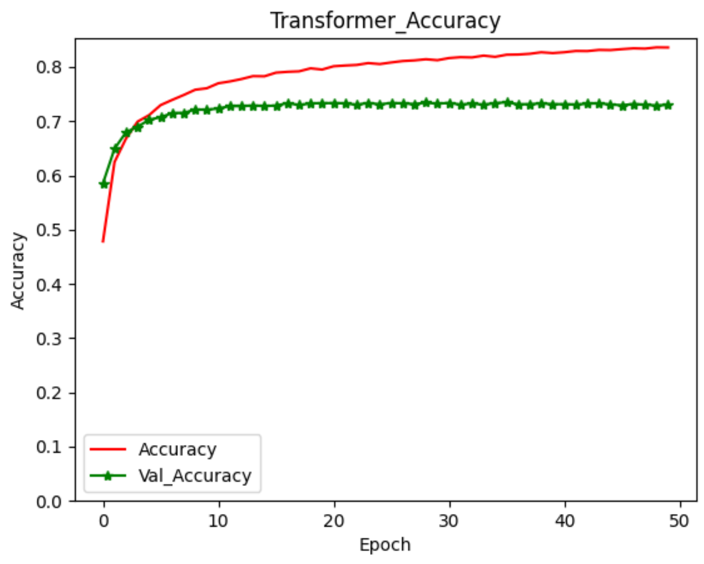


Figure 4 : Transformer Accuracy for 50 Epochs

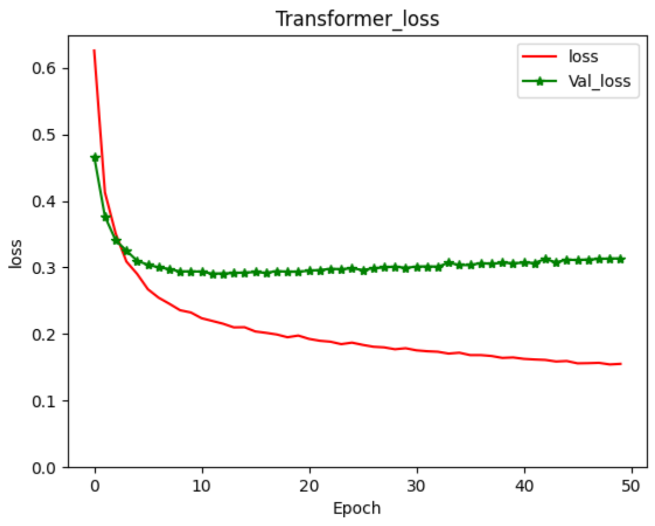


Figure 5 : Transformer Loss for 50 Epochs

The BLEU score for training for 1 encoder and 1 decoder was 0.0012, for 4 encoder and 4 decoder was 0.266. In the original paper, the transformer was trained for 6 encoders and 6 decoders and achieved a BLEU score of 28.4. This clearly shows that as the transformer layer is increased, the score becomes significantly better.

The pretrained model takes about 5 hours to run for 10 epochs, so initially we trained it only for 5 epochs and this was sufficient to achieve the smallest loss for training and validation 0.38 and 0.37 respectively. Since we used a different framework in this case, accuracy was not monitored in this step. However, the BLEU score was again used for the evaluation of the translated texts. The BLEU score evaluation of the Hugging Face Transformer was achieved to be 38 for the translated text. The BLEU scores for all models is shown below:

BLEU SCORE	
Model	Bleu Score
Transformer – 1 Layer	0.0012
Transformer – 4 Layer	0.266
Transformer – 6 Layer	28.4 (taken from the paper)
Fine Tuned Transformer	38

5. DISCUSSION AND CONCLUSION

In our implementation of the Transformer model, we observed that increasing the number of layers in the Transformer allowed for improvement in the Translation of the text. We achieved a low score for the BLEU for only one encoder and one decoder but as it increased to 4 encoders and 4 decoders the BLEU scores also increased.

Accuracy is not a good metric to evaluate how well these models worked. Although the accuracy for both layers was high, the translations were not satisfactory. BLEU is a far more reliable metric as we compare the frequency with which n-grams are repeated in both source and target sentences. However, the BLEU score has some disadvantages 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.

As we expected, the pre-trained model outperformed the Transformer from the Scratch model. This model used a very elaborate architecture and pre-trained weights which lead the model to capture the context of longer sentences.

Overall, this project allowed us to understand how a basic transformer works. Given more time, we would have liked to explore the evolution of the translations for different datasets.

REFERENCES

Bibliography

al, V. e. (2017). *Neural machine translation with a Transformer and Keras*. Retrieved from https://www.tensorflow.org/text/tutorials/transformer

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

Chollet, F. (2017, September 29). *A ten-minute introduction to sequence-to-sequence learning in Keras*. Retrieved from https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html

Cloud, A. (2018, September 12). Retrieved from https://alibaba-cloud.medium.com/self-attention-mechanisms-in-natural-language-processing-9f28315ff905

Minh-Thang Luong, H. P. (2015). *Effective Approaches to Attention-based Neural Machine Translation*. Retrieved from https://arxiv.org/abs/1508.04025

Raschka, S. (2023, February 09). *Understanding and Coding the Self-Attention Mechanism of Large Language Models From Scratch*. Retrieved from https://sebastianraschka.com/blog/2023/self-attention-from-scratch.html

Appendix:

1.1The Transformers from Scratch – Implemented using Tensorflow

Link : https://www.tensorflow.org/text/tutorials/transformer

# Install the required Dependencies

```
# Install the most re version of TensorFlow to use the improved
# masking support for `tf.keras.layers.MultiHeadAttention`.
!apt install --allow-change-held-packages libcudnn8=8.1.0.77-1+cuda11.2
!pip uninstall -y -q tensorflow keras tensorflow-estimator tensorflow-text
!pip install protobuf~=3.20.3
!pip install -q tensorflow_datasets
!pip install -q -U tensorflow-text tensorflow
```

# Import the necessary modules

```
import collections
import os
import pathlib
import re
import string
import sys
import tempfile
import time

import numpy as np
import matplotlib.pyplot as plt

import tensorflow_datasets as tfds
import tensorflow_text as text
import tensorflow as tf
from tensorflow_text.tools.wordpiece_vocab import bert_vocab_from_dataset
as bert_vocab
```

```
examples, metadata = tfds.load('ted_hrlr_translate/pt_to_en',
with_info=True,
                               as_supervised=True)
train_examples, val_examples = examples['train'], examples['validation']
```

```python
# This dataset produces Portuguese / English Sentence Pairs
for pt, en in train_examples.take(1):
  print("Portuguese: ", pt.numpy().decode('utf-8'))
  print("English:   ", en.numpy().decode('utf-8'))
```

```python
bert_tokenizer_params=dict(lower_case=True)
reserved_tokens=["[PAD]", "[UNK]", "[START]", "[END]"]

bert_vocab_args = dict(
    # The target vocabulary size
    vocab_size = 8000,
    # Reserved tokens that must be included in the vocabulary
    reserved_tokens=reserved_tokens,
    # Arguments for `text.BertTokenizer`
    bert_tokenizer_params=bert_tokenizer_params,
    # Arguments for
`wordpiece_vocab.wordpiece_tokenizer_learner_lib.learn`
    learn_params={},
)
```

```python
class CustomTokenizer(tf.Module):
  def __init__(self, reserved_tokens, vocab_path):
    self.tokenizer = text.BertTokenizer(vocab_path, lower_case=True)
    self._reserved_tokens = reserved_tokens
    self._vocab_path = tf.saved_model.Asset(vocab_path)

    vocab = pathlib.Path(vocab_path).read_text().splitlines()
    self.vocab = tf.Variable(vocab)

    ## Create the signatures for export:

    # Include a tokenize signature for a batch of strings.
    self.tokenize.get_concrete_function(
        tf.TensorSpec(shape=[None], dtype=tf.string))

    # Include `detokenize` and `lookup` signatures for:
    #   * `Tensors` with shapes [tokens] and [batch, tokens]
    #   * `RaggedTensors` with shape [batch, tokens]
    self.detokenize.get_concrete_function(
        tf.TensorSpec(shape=[None, None], dtype=tf.int64))
    self.detokenize.get_concrete_function(
          tf.RaggedTensorSpec(shape=[None, None], dtype=tf.int64))
    self.lookup.get_concrete_function(
        tf.TensorSpec(shape=[None, None], dtype=tf.int64))
    self.lookup.get_concrete_function(
          tf.RaggedTensorSpec(shape=[None, None], dtype=tf.int64))

    # These `get_*` methods take no arguments
    self.get_vocab_size.get_concrete_function()
    self.get_vocab_path.get_concrete_function()
    self.get_reserved_tokens.get_concrete_function()

  @tf.function
  def tokenize(self, strings):
    enc = self.tokenizer.tokenize(strings)
    # Merge the `word` and `word-piece` axes.
    enc = enc.merge_dims(-2,-1)
    enc = add_start_end(enc)
    return enc

  @tf.function
  def detokenize(self, tokenized):
    words = self.tokenizer.detokenize(tokenized)
    return cleanup_text(self._reserved_tokens, words)

  @tf.function
  def lookup(self, token_ids):
    return tf.gather(self.vocab, token_ids)

  @tf.function
  def get_vocab_size(self):
    return tf.shape(self.vocab)[0]

  @tf.function
  def get_vocab_path(self):
    return self._vocab_path

  @tf.function
  def get_reserved_tokens(self):
    return tf.constant(self._reserved_tokens)
```

```
tokenizers = tf.Module()
tokenizers.pt = CustomTokenizer(reserved_tokens, 'pt_vocab.txt')
tokenizers.en = CustomTokenizer(reserved_tokens, 'en_vocab.txt')
```

```
MAX_TOKENS=128
```

```
def prepare_batch(pt, en):
    pt = tokenizers.pt.tokenize(pt)      # Output is ragged.
    pt = pt[:, :MAX_TOKENS]    # Trim to MAX_TOKENS.
    pt = pt.to_tensor()  # Convert to 0-padded dense Tensor

    en = tokenizers.en.tokenize(en)
    en = en[:, :(MAX_TOKENS+1)]
    en_inputs = en[:, :-1].to_tensor()  # Drop the [END] tokens
    en_labels = en[:, 1:].to_tensor()   # Drop the [START] tokens

    return (pt, en_inputs), en_labels
```

```
BUFFER_SIZE = 20000
BATCH_SIZE = 64




def make_batches(ds):
  return (
      ds
      .shuffle(BUFFER_SIZE)
      .batch(BATCH_SIZE)
      .map(prepare_batch, tf.data.AUTOTUNE)
      .prefetch(buffer_size=tf.data.AUTOTUNE))
```

```
# Create training and validation set batches.
train_batches = make_batches(train_examples)
val_batches = make_batches(val_examples)
```

```
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(tf.keras.layers.Layer):
  def __init__(self, vocab_size, d_model):
    super().__init__()
    self.d_model = d_model
    self.embedding = tf.keras.layers.Embedding(vocab_size, d_model,
mask_zero=True)
    self.pos_encoding = positional_encoding(length=2048, depth=d_model)

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

  def call(self, x):
    length = tf.shape(x)[1]
    x = self.embedding(x)
    # This factor sets the relative scale of the embedding and
positonal_encoding.
    x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
    x = x + self.pos_encoding[tf.newaxis, :length, :]
    return x
```

# Create the Base Attention Layers

```
class BaseAttention(tf.keras.layers.Layer):
  def __init__(self, **kwargs):
    super().__init__()
    self.mha = tf.keras.layers.MultiHeadAttention(**kwargs)
    self.layernorm = tf.keras.layers.LayerNormalization()
    self.add = tf.keras.layers.Add()
```

# Create the Cross Attention Layer that connects the Encoder to the Decoder

```
class CrossAttention(BaseAttention):
  def call(self, x, context):
    attn_output, attn_scores = self.mha(
        query=x,
        key=context,
        value=context,
        return_attention_scores=True)

    # Cache the attention scores for plotting later.
    self.last_attn_scores = attn_scores
    x = self.add([x, attn_output])
    x = self.layernorm(x)

    return x
```

# Create the Global Self Attention

```
class GlobalSelfAttention(BaseAttention):
  def call(self, x):
    attn_output = self.mha(
        query=x,
        value=x,
        key=x)
    x = self.add([x, attn_output])
    x = self.layernorm(x)
    return x
```

# Create Causal Self Attention Layer

```
class CausalSelfAttention(BaseAttention):
  def call(self, x):
    attn_output = self.mha(
        query=x,
        value=x,
        key=x,
        use_causal_mask = True)
    x = self.add([x, attn_output])
    x = self.layernorm(x)
    return x
```

# Create the Feed Forward Network

```
class FeedForward(tf.keras.layers.Layer):
  def __init__(self, d_model, dff, dropout_rate=0.1):
    super().__init__()
    self.seq = tf.keras.Sequential([
      tf.keras.layers.Dense(dff, activation='relu'),
      tf.keras.layers.Dense(d_model),
      tf.keras.layers.Dropout(dropout_rate)
    ])
    self.add = tf.keras.layers.Add()
    self.layer_norm = tf.keras.layers.LayerNormalization()

  def call(self, x):
```

```python
    x = self.add([x, self.seq(x)])
    x = self.layer_norm(x)
    return x
```

# Create the Class Encoder Layer

```python
class EncoderLayer(tf.keras.layers.Layer):
  def __init__(self,*, d_model, num_heads, dff, dropout_rate=0.1):
    super().__init__()

    self.self_attention = GlobalSelfAttention(
        num_heads=num_heads,
        key_dim=d_model,
        dropout=dropout_rate)

    self.ffn = FeedForward(d_model, dff)

  def call(self, x):
    x = self.self_attention(x)
    x = self.ffn(x)
    return x
```

# Create the Encoder

```python
class Encoder(tf.keras.layers.Layer):
  def __init__(self, *, num_layers, d_model, num_heads,
               dff, vocab_size, dropout_rate=0.1):
    super().__init__()

    self.d_model = d_model
    self.num_layers = num_layers

    self.pos_embedding = PositionalEmbedding(
        vocab_size=vocab_size, d_model=d_model)

    self.enc_layers = [
        EncoderLayer(d_model=d_model,
                     num_heads=num_heads,
                     dff=dff,
                     dropout_rate=dropout_rate)
        for _ in range(num_layers)]
    self.dropout = tf.keras.layers.Dropout(dropout_rate)

  def call(self, x):
    # `x` is token-IDs shape: (batch, seq_len)
    x = self.pos_embedding(x)  # Shape `(batch_size, seq_len, d_model)`.

    # Add dropout.
    x = self.dropout(x)

    for i in range(self.num_layers):
      x = self.enc_layers[i](x)

    return x  # Shape `(batch_size, seq_len, d_model)`.
```

# Create the Decoder Layer

```
class DecoderLayer(tf.keras.layers.Layer):
  def __init__(self,
               *,
               d_model,
               num_heads,
               dff,
               dropout_rate=0.1):
    super(DecoderLayer, self).__init__()

    self.causal_self_attention = CausalSelfAttention(
        num_heads=num_heads,
        key_dim=d_model,
        dropout=dropout_rate)

    self.cross_attention = CrossAttention(
        num_heads=num_heads,
        key_dim=d_model,
        dropout=dropout_rate)

    self.ffn = FeedForward(d_model, dff)

  def call(self, x, context):
    x = self.causal_self_attention(x=x)
    x = self.cross_attention(x=x, context=context)

    # Cache the last attention scores for plotting later
    self.last_attn_scores = self.cross_attention.last_attn_scores

    x = self.ffn(x)  # Shape `(batch_size, seq_len, d_model)`.
    return x
```

# Create the Decoder

```
class Decoder(tf.keras.layers.Layer):
  def __init__(self, *, num_layers, d_model, num_heads, dff, vocab_size,
               dropout_rate=0.1):
    super(Decoder, self).__init__()

    self.d_model = d_model
    self.num_layers = num_layers

    self.pos_embedding = PositionalEmbedding(vocab_size=vocab_size,
                                             d_model=d_model)
    self.dropout = tf.keras.layers.Dropout(dropout_rate)
    self.dec_layers = [
        DecoderLayer(d_model=d_model, num_heads=num_heads,
                     dff=dff, dropout_rate=dropout_rate)
        for _ in range(num_layers)]

    self.last_attn_scores = None

  def call(self, x, context):
    # `x` is token-IDs shape (batch, target_seq_len)
    x = self.pos_embedding(x)  # (batch_size, target_seq_len, d_model)

    x = self.dropout(x)

    for i in range(self.num_layers):
      x  = self.dec_layers[i](x, context)

    self.last_attn_scores = self.dec_layers[-1].last_attn_scores

    # The shape of x is (batch_size, target_seq_len, d_model).
    return x
```

# Create the Transformer Model

```
class Transformer(tf.keras.Model):
  def __init__(self, *, num_layers, d_model, num_heads, dff,
               input_vocab_size, target_vocab_size, dropout_rate=0.1):
    super().__init__()
    self.encoder = Encoder(num_layers=num_layers, d_model=d_model,
                           num_heads=num_heads, dff=dff,
                           vocab_size=input_vocab_size,
                           dropout_rate=dropout_rate)
    self.decoder = Decoder(num_layers=num_layers, d_model=d_model,
                           num_heads=num_heads, dff=dff,
                           vocab_size=target_vocab_size,
                           dropout_rate=dropout_rate)

    self.final_layer = tf.keras.layers.Dense(target_vocab_size)

  def call(self, inputs):
    # To use a Keras model with `.fit` you must pass all your inputs in
the
    # first argument.
    context, x  = inputs

    context = self.encoder(context)  # (batch_size, context_len, d_model)

    x = self.decoder(x, context)  # (batch_size, target_len, d_model)

    # Final linear layer output.
    logits = self.final_layer(x)  # (batch_size, target_len,
target_vocab_size)

    try:
      # Drop the keras mask, so it doesn't scale the losses/metrics.
      # b/250038731
      del logits._keras_mask
    except AttributeError:
      pass

    # Return the final output and the attention weights.
    return logits
```

# Declare the Hyperparameters :

```
num_layers = 4
d_model = 128
dff = 512
num_heads = 8
dropout_rate = 0.1
transformer = Transformer(
    num_layers=num_layers,
    d_model=d_model,
    num_heads=num_heads,
    dff=dff,
    input_vocab_size=tokenizers.pt.get_vocab_size().numpy(),
    target_vocab_size=tokenizers.en.get_vocab_size().numpy(),
    dropout_rate=dropout_rate)
output = transformer((pt, en))

print(en.shape)
print(pt.shape)
print(output.shape)
attn_scores = transformer.decoder.dec_layers[-1].last_attn_scores
print(attn_scores.shape)  # (batch, heads, target_seq, input_seq)
transformer.summary()
```

# Setup the Optimizer

```
class CustomSchedule(tf.keras.optimizers.schedules.LearningRateSchedule):
  def __init__(self, d_model, warmup_steps=4000):
    super().__init__()

    self.d_model = d_model
    self.d_model = tf.cast(self.d_model, tf.float32)

    self.warmup_steps = warmup_steps

  def __call__(self, step):
    step = tf.cast(step, dtype=tf.float32)
    arg1 = tf.math.rsqrt(step)
    arg2 = step * (self.warmup_steps ** -1.5)

    return tf.math.rsqrt(self.d_model) * tf.math.minimum(arg1, arg2)
learning_rate = CustomSchedule(d_model)

optimizer = tf.keras.optimizers.Adam(learning_rate, beta_1=0.9,
beta_2=0.98,
                                     epsilon=1e-9)
def masked_loss(label, pred):
  mask = label != 0
  loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')
  loss = loss_object(label, pred)

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

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

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

  mask = label != 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)
```

```
transformer.compile(
    loss=masked_loss,
    optimizer=optimizer,
    metrics=[masked_accuracy])
transformer.fit(train_batches,
                epochs=20,
                validation_data=val_batches)
```

# Create the Translator

```
class Translator(tf.Module):
  def __init__(self, tokenizers, transformer):
    self.tokenizers = tokenizers
    self.transformer = transformer

  def __call__(self, sentence, max_length=MAX_TOKENS):
    # The input sentence is Portuguese, hence adding the `[START]` and
`[END]` tokens.
    assert isinstance(sentence, tf.Tensor)
    if len(sentence.shape) == 0:
      sentence = sentence[tf.newaxis]

    sentence = self.tokenizers.pt.tokenize(sentence).to_tensor()

    encoder_input = sentence

    # As the output language is English, initialize the output with the
    # English `[START]` token.
    start_end = self.tokenizers.en.tokenize([''])[0]
    start = start_end[0][tf.newaxis]
    end = start_end[1][tf.newaxis]
    # `tf.TensorArray` is required here (instead of a Python list), so
that the
    # dynamic-loop can be traced by `tf.function`.
    output_array = tf.TensorArray(dtype=tf.int64, size=0,
dynamic_size=True)
    output_array = output_array.write(0, start)

    for i in tf.range(max_length):
      output = tf.transpose(output_array.stack())
      predictions = self.transformer([encoder_input, output],
training=False)

      # Select the last token from the `seq_len` dimension.
      predictions = predictions[:, -1:, :]  # Shape `(batch_size, 1,
vocab_size)`.

      predicted_id = tf.argmax(predictions, axis=-1)

      # Concatenate the `predicted_id` to the output which is given to the
      # decoder as its input.
      output_array = output_array.write(i+1, predicted_id[0])

      if predicted_id == end:
        break

    output = tf.transpose(output_array.stack())
    # The output shape is `(1, tokens)`.
    text = tokenizers.en.detokenize(output)[0]  # Shape: `()`.

    tokens = tokenizers.en.lookup(output)[0]

    # `tf.function` prevents us from using the attention_weights that were
    # calculated on the last iteration of the loop.
    # So, recalculate them outside the loop.
    self.transformer([encoder_input, output[:,:-1]], training=False)
    attention_weights = self.transformer.decoder.last_attn_scores

    return text, tokens, attention_weights
```

```
translator = Translator(tokenizers, transformer)
def print_translation(sentence, tokens, ground_truth):
  print(f'{"Input:":15s}: {sentence}')
  print(f'{"Prediction":15s}: {tokens.numpy().decode("utf-8")}')
  print(f'{"Ground truth":15s}: {ground_truth}')
```

```

# Test the Translations
sentence = 'este é um problema que temos que resolver.'
ground_truth = 'this is a problem we have to solve .'

translated_text, translated_tokens, attention_weights = translator(
    tf.constant(sentence))
print_translation(sentence, translated_text, ground_truth)
```

```
import nltk
from nltk.translate.bleu_score import sentence_bleu

def compute_bleu(target_texts, predicted_texts):
    assert len(target_texts) == len(predicted_texts), "Both lists must
have the same length"
    total_score = 0.0
    for ref, pred in zip(target_texts, predicted_texts):
        # Splitting the sentences into tokens
        ref_tokens = ref.split()
        pred_tokens = pred.split()
        # Compute BLEU for this pair
        score = sentence_bleu([ref_tokens], pred_tokens,
smoothing_function=nltk.translate.bleu_score.SmoothingFunction().method1)
        total_score += score
    # Average the scores
    avg_score = total_score / len(target_texts)
    return avg_score

bleu = compute_bleu(ground_truth, translated_texts)
print(f"Average BLEU score: {bleu}")
```

1.2 HUGGING FACE TRANSFORMER – PRE-TRAINED MODEL FOR PORTUGUESE TO ENGLISH TRANSLATIONS

# Install the required Dependencies:

```
 ! pip install --trusted-host pypi.org --trusted-host pypi.python.org --trusted-host files.pythonhosted.org <package_name>
 ! pip install datasets sacrebleu torch transformers sentencepiece transformers[sentencepiece]
 ! pip install accelerate -U
```

# Required Imports

```
import warnings
import numpy as np
import pandas as pd

import torch
import transformers

from datasets import Dataset
from datasets import load_metric

from tqdm import tqdm
from transformers import AutoTokenizer
from sklearn.model_selection import train_test_split
from transformers import T5Tokenizer, T5ForConditionalGeneration, AutoModelForSeq2SeqLM
from transformers import DataCollatorForSeq2Seq, Seq2SeqTrainingArguments, Seq2SeqTrainer
```

# Declared Constants

```
BATCH_SIZE = 16
BLEU = "bleu"
ENGLISH = "en"
ENGLISH_TEXT = "english_text"
EPOCH = "epoch"
INPUT_IDS = "input_ids"
GEN_LEN = "gen_len"
MAX_INPUT_LENGTH = 128
MAX_TARGET_LENGTH = 128
MODEL_CHECKPOINT = "unicamp-dl/translation-pt-en-t5"
MODEL_NAME = MODEL_CHECKPOINT.split("/")[-1]
LABELS = "labels"
```

```
PREFIX = ""
PORTUGUESE = "pt"
PORTUGUESE_TEXT = "portuguese_text"
SCORE = "score"
SOURCE_LANG = "pt"
TARGET_LANG = "en"
TRANSLATION = "translation"
UNNAMED_COL = "Unnamed: 0"
```

# Helper Functions

```
def postprocess_text(preds: list, labels: list) -> tuple:
    """Performs post processing on the prediction text and labels"""

    preds = [pred.strip() for pred in preds]
    labels = [[label.strip()] for label in labels]

    return preds, labels


def prep_data_for_model_fine_tuning(source_lang: list, target_lang: list) ->
list:
    """Takes the input data lists and converts into translation list of dicts"""

    data_dict = dict()
    data_dict[TRANSLATION] = []

    for sr_text, tr_text in zip(source_lang, target_lang):
        temp_dict = dict()
        temp_dict[PORTUGUESE] = sr_text
        temp_dict[ENGLISH] = tr_text

        data_dict[TRANSLATION].append(temp_dict)

    return data_dict


def generate_model_ready_dataset(dataset: list, source: str, target: str,
                                 model_checkpoint: str,
                                 tokenizer: AutoTokenizer):
    """Makes the data training ready for the model"""

    preped_data = []

    for row in dataset:
        inputs = PREFIX + row[source]
        targets = row[target]

        model_inputs = tokenizer(inputs, max_length=MAX_INPUT_LENGTH,
                                 truncation=True, padding=True)

        model_inputs[TRANSLATION] = row

        # setup the tokenizer for targets
        with tokenizer.as_target_tokenizer():
            labels = tokenizer(targets, max_length=MAX_INPUT_LENGTH,
                                 truncation=True, padding=True)
            model_inputs[LABELS] = labels[INPUT_IDS]

        preped_data.append(model_inputs)

    return preped_data



def compute_metrics(eval_preds: tuple) -> dict:
    """computes bleu score and other performance metrics """

    metric = load_metric("sacrebleu")
    tokenizer = AutoTokenizer.from_pretrained(MODEL_CHECKPOINT)

    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
```

# Load and Prepare the Dataset

```
from google.colab import drive

drive.mount('/content/drive')

data_path = "/content/drive/MyDrive/por.txt"

raw= pd.read_table(data_path,names=['english', 'portugese', 'comment'])
raw.drop(columns=['comment'], inplace=True)
raw.to_csv('/content/drive/MyDrive/eng-pt.csv', index=False, sep='\t',
header=False)

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

# Train, Test and Validation Split of Data

```
X = translation_data['Portuguese']
y = translation_data['English']
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.10,
                                                    shuffle=True,
                                                    random_state=100)

print("INITIAL X-TRAIN SHAPE: ", x_train.shape)
print("INITIAL Y-TRAIN SHAPE: ", y_train.shape)
print("X-TEST SHAPE: ", x_test.shape)
print("Y-TEST SHAPE: ", y_test.shape)
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train,
                                                  test_size=0.20,
                                                  shuffle=True,
                                                  random_state=100)

print("FINAL X-TRAIN SHAPE: ", x_train.shape)
```

```python
print("FINAL Y-TRAIN SHAPE: ", y_train.shape)
print("X-VAL SHAPE: ", x_val.shape)
print("Y-VAL SHAPE: ", y_val.shape)
```

# Load Tokenizer from AutoTokenizer Class and Prepare the model ready dataset

```python
tokenizer = AutoTokenizer.from_pretrained(MODEL_CHECKPOINT)
training_data = prep_data_for_model_fine_tuning(x_train.values, y_train.values)

validation_data = prep_data_for_model_fine_tuning(x_val.values, y_val.values)

test_data = prep_data_for_model_fine_tuning(x_test.values, y_test.values)
train_data = generate_model_ready_dataset(dataset=training_data[TRANSLATION],
                                          tokenizer=tokenizer,
                                          source=PORTUGUESE,
                                          target=ENGLISH,
                                          model_checkpoint=MODEL_CHECKPOINT)

validation_data =
generate_model_ready_dataset(dataset=validation_data[TRANSLATION],
                                               tokenizer=tokenizer,
                                               source=PORTUGUESE,
                                               target=ENGLISH,
                                               model_checkpoint=MODEL_CHECKPOINT)

test_data = generate_model_ready_dataset(dataset=test_data[TRANSLATION],
                                               tokenizer=tokenizer,
                                               source=PORTUGUESE,
                                               target=ENGLISH,
                                               model_checkpoint=MODEL_CHECKPOINT)
train_dataset = Dataset.from_pandas(train_df)
validation_dataset = Dataset.from_pandas(validation_df)
test_dataset = Dataset.from_pandas(test_df)
model = AutoModelForSeq2SeqLM.from_pretrained(MODEL_CHECKPOINT)
model_args = Seq2SeqTrainingArguments(
    f"{MODEL_NAME}-finetuned-{SOURCE_LANG}-to-{TARGET_LANG}",
    evaluation_strategy=EPOCH,
    learning_rate=2e-4,
    per_device_train_batch_size=BATCH_SIZE,
    per_device_eval_batch_size=BATCH_SIZE,
    weight_decay=0.02,
    save_total_limit=3,
    num_train_epochs=10,
    predict_with_generate=True
)
```

```
trainer = Seq2SeqTrainer(
    model,
    model_args,
    train_dataset=train_dataset,
    eval_dataset=validation_dataset,
    data_collator=data_collator,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics
)
trainer.train()
trainer.save_model("FineTunedTransformer")
test_results = trainer.predict(test_dataset)
print("Test Bleu Score: ", test_results.metrics["test_bleu"])
```