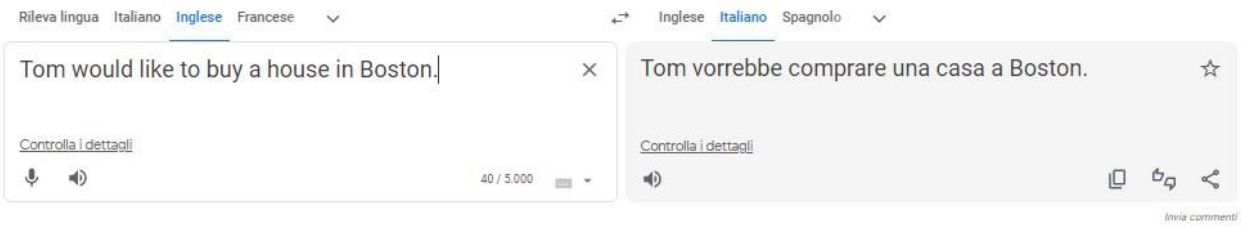
Text Mining and Natural Language Processing 2022-2023

Acquadro, 502311

Uttini, 502183

BIF Translator

Introduction



BIF Translation is a machine translation task project, in which the developed algorithms are able to generate an accurate and natural translation in Italian of an input English sentence.

Accurate because the intrinsic meaning of the source sentence, the English one, is preserved.

Natural because the translation appears to be more human-like, than just a mere translation of the words. The higher the performances of the models are, the more the meaning is preserved.

This is possible because the algorithms are able to consider the intrinsic properties of the language, like syntax, semantics and other linguistic aspects, and some of them are also able to get a better 'understanding' of what is more relevant in the sentence and will have a bigger impact on the translation.

Machine translation is a task that has started to be addressed after WW2.

The first systems were based on written rules in a dictionary. But they were very inaccurate and expensive.

So, another type of MT systems was developed: the example-based systems. They worked with n-grams of example sentences.

In 1990, IBM introduced the statistical models for machine translations tasks. They were far better than the previous systems since they counted the statistics of the linguistic features.

After 2016, all these models were replaced by the neural machine translation models, because they have great performances and are the models used also today, even if the MT field remains an open field.

The main architecture used for this type of task is the Transformer model, developed by Google in 2017, which we've also used in this project.

But we've also used other models like RNN, such as GRU, LSTM and Bidirectional.

We've described in detail all these models, in the section below, called 'Methodology'.

Data

We have chosen to take the dataset from the website http://www.manythings.org/anki/ that contains lots of dataset for Machine Translation tasks. Since we are Italian and we have a great level of English, we decide to use a English-Italian dataset. In this way, we are also able to compare the translation with our knowledge.

The dimension of the dataset was about 40 mb, so we have zipped and uploaded it on our GitHub (https://github.com/lorenzouttini/Exam_NLP/raw/main/ita-eng.zip). Then, we have unzipped it through a python library *"Zipfile"* and we have read it.

Our original dataset was a text file, and it was composed by rows as the follows:

```
Just do it.     Fallo e basta.  CC-BY 2.0 (France) Attribution: tatoeba.org #431271 (qdii) & #695200 (Heracleum)
Just relax.     Rilassati e basta.      CC-BY 2.0 (France) Attribution: tatoeba.org #1555685 (Spamster) & #4564735 (Guybrush88)
Just relax.     Si rilassi e basta.     CC-BY 2.0 (France) Attribution: tatoeba.org #1555685 (Spamster) & #4564736 (Guybrush88)
Just relax.     Rilassatevi e basta.    CC-BY 2.0 (France) Attribution: tatoeba.org #1555685 (Spamster) & #4564737 (Guybrush88)
```

Since what we need of these rows is only the first two elements (English and Italian sentence), we create a list *"pairs_list"* of sets, each set containing a pair of an Italian sentence followed by its corresponding English one (adding 'start' and 'end' as tokens).

Moreover, we decide to reduce randomly the pairs to 80% of the total for computational and saved-time reasons.

Let's describe our dataset to understand better some features that we will be useful in the project. We divide between Italian and English sentences and words.

Italian Statistics:

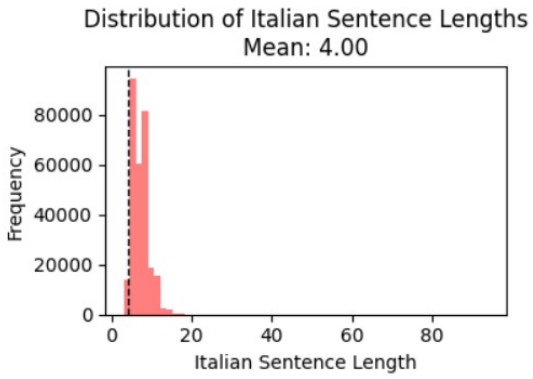
```
Total number of Italian words (excluding tokens): 1889125
Total number of sentences: 291360
Mean length of Italian words (excluding tokens): 4
Mean length of Italian sentences: 6
Number of unique Italian words (excluding tokens): 27191
```

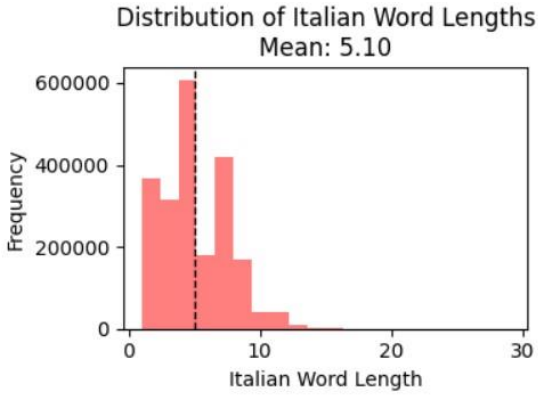
So, from the analysis we can claim that our Italian list is formed by a great number of words (tokens), around 2 million, and an average vocabulary size, almost 30.000 unique words.

Moreover, it is important to underline that the mean of the sentences is about 6 words, that we can

consider a great number. But we have also to focus on the mean length of words, around 4, that indicates a strict dependency of the dataset from short words that could be not very significant for the translation task.

We plot (through the library *matplotlib*) also the distributions of the Italian sentences and words.



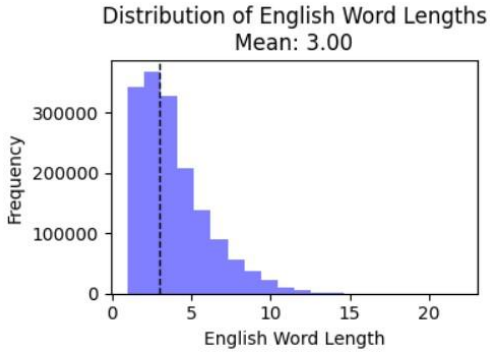


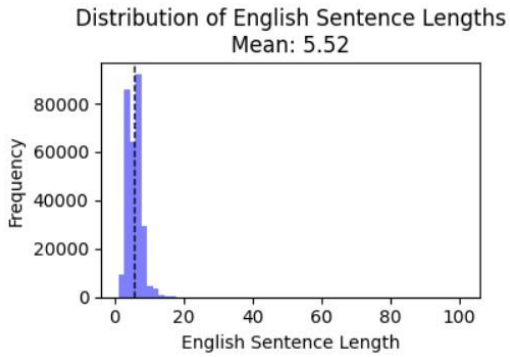
We can compute the same statistics for the English sentences and words:

```
Total number of words (excluding tokens): 2020048
Total number of sentences: 291360
Mean length of words (excluding tokens): 3
Mean length of sentences: 7
Number of unique words (excluding tokens): 13406
```

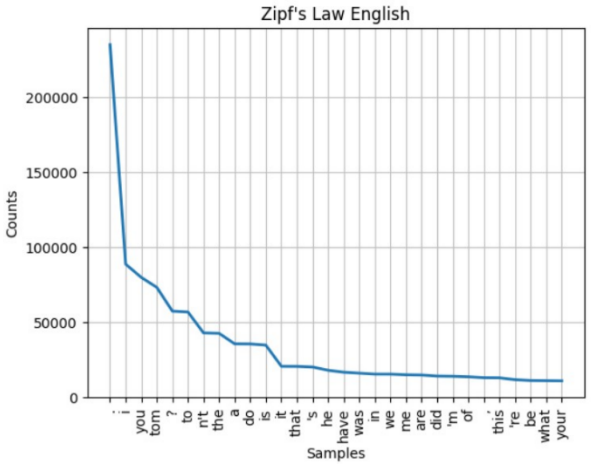
We notice that English data are almost identical to Italian data.

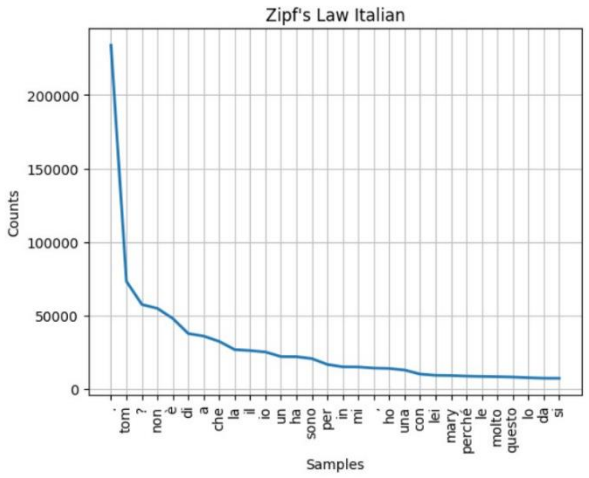
We plot also the distribution for the English sentences and words:





To understand in a complete way the quality of the data we have, we can compute the graph described by the **Zipf's Law**: the frequency of a word is inversely proportional to its rank in a frequency distribution. In other words, this law states that in a corpus text (like the dataset we are using), a few words occur very frequently, while the majority of words occur infrequently. The perfect Zipf's Law representation should be a hyperbole curve. We have computed it through the library *"nltk (FreqDist)"*. We compare the curves for Italian and English words/frequency:





We can observe that both curves look similar to a hyperbole, so in our corpus the Zipf's Law is respected. What is important for our translation task is the central part of the curve that corresponds to the words that are not too frequent and not too rare, so they have a significant role in the sentences.

Finally, we randomly split the dataset in three different parts:

- **Training** set: 70%
- **Validation** set: 15%
- **Test** set: 15%

The Training and Validation sets will be used for fitting and training the model. The Test set will be used instead to see whether the predicted sentences are good or not, evaluating the model.

Methodology

In this part we explain how the models we used are built and how they work. All the models we choose are based on a common structure called **seq2seq** or sequence2sequence. This architecture is formed by two different parts:

- **Encoder**: it processes the input, and it creates a representation of the sequence as output.
- **Decoder**: it processes the target and receives as an additional input the output of the encoder. It gives as output the predicted sentence (in this case).

Each model has its own differences on how it is built, and we decided to adopt and compare two different types:

- **RNN model**: Bidirectional GRU (encoder), GRU (decoder).
- **Transformer** model.

Firstly, we have to pre-process the sequences (Italian and English sentences) from to text to vectors. This vectorization allows the network that we will build to take as input all the training sentences.

We start by tokenize the sentences constructing a function called *"custom_function"*. Through this function, divide the words, make lower case and remove the punctuations signs (thanks to the library *string*). It is important to note that we have not considered all the punctuation signs: we leave the square brackets [,], because they are part of the starting and final tokens ([start], [end]) that indicates where the target sentence begins and finishes.

Then, we have defined the size of the vocabulary (*"dictionary_size"*), around 25.000 words, and the length of each sequence (*"sequence_size"*), that is 15. We choose that vocabulary size because the unique words in the original text file were almost 25.000 and that sequence length because the mean of the sentences was about 10. All the sentences less than this number will be padded and the ones greater will be truncated.

We create the Vectorization layers for English and Italian texts. This layers, imported from Keras, take as parameter:

- *"max_tokens":* it corresponds to the vocabulary size that we have defined and called *dictionary_size* (25.000).
- *"output_mode":* it indicates how the vectors will be formed, in this case we choose *int.*
- *"output_sequence_length":* it indicates what will be the length of each vector, in our case it corresponds to *sequence_size* (15).
- *"standardize":* it indicates how the word should be standardize (tokenized), in our case it corresponds to the *custom function* we have defined because it removes punctuations, make lower case and leave square brackets. We use this parameter only for the Italian sentences.

These layers have been adapted (through *adapt* method) to the lists of the Italian and English training sentence.

So, the training sentences are now composed by integers vectors of dimension 18, where each word is associated with an integer number corresponding to the index in the vocabulary.

> We make an example: let's consider a vocabulary with four words: *["hello", "how", "are", "you"]*. The vectorization layer assigns the following integer values to these words: *[0, 1, 2, 3]*. When we vectorize a sentence like *"Hello, how are you?"*, the vectorized representation (set by the vectorization layers) may look like *[0, 1, 2, 3]*. Each integer in the vectorized sentence represents the corresponding word's index in the vocabulary.

After the vectorization of the training sentences, we construct the whole dataset that we will pass as input of the first network's layer.

We use two different function to construct the dataset:

- *"format_dataset":* it takes as parameters the Italian and English training lists and it vectorizes them (through the functions we have defined above). Then it returns the Italian and English inputs, and the Italian output.
- *"make_dataset":* it takes as a parameter a set of pairs (could be training or validation pairs) and it zip and transforms them as lists. Then it constructs a Tensorflow dataset, adapting it to the *batch_size* we have defined (64) and to the *format_dataset* function (all the sentences are vectorized).

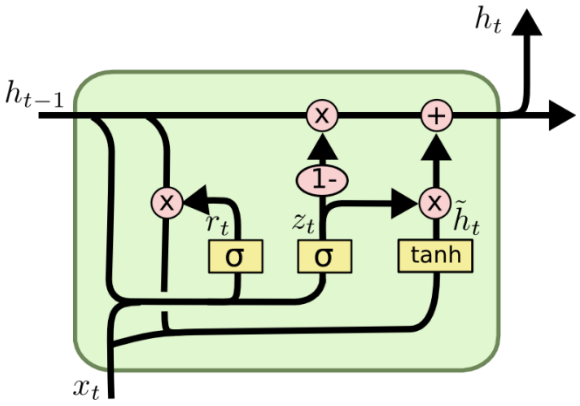
We finally passed the latter function to *train_pairs* and *val_pairs* to form the train dataset (*train_ds*) and the validation dataset *(val_ds)*. In this way, we obtain a dataset with 3 features (English Input, Italian Input, Italian Output), where each row is formed by three vectors of dimension (64, 18). The shape of each vector corresponds to the *batch_size* (64) and to the length of the sentences (18).

We have pre-processed the data and now we can construct the models.

Seq2Seq with GRU

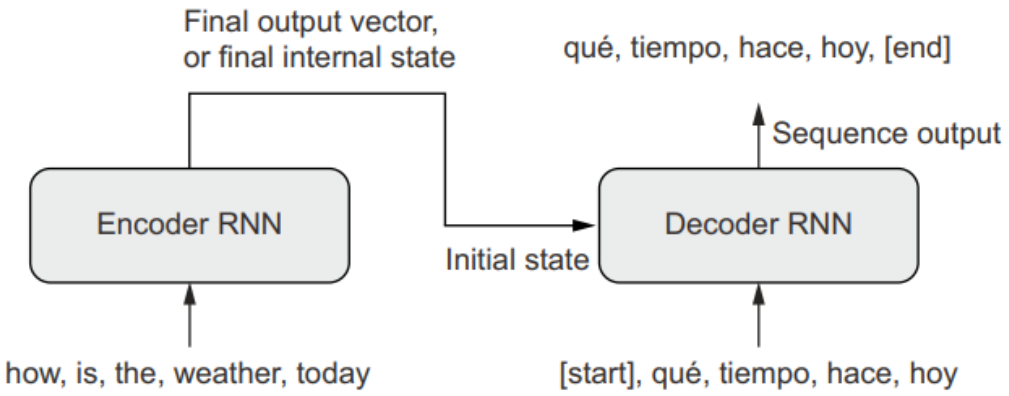
We start by describing briefly what is a **RNN GRU model**.

GRU (Gated Recurrent Unit) is a type of RNN architecture that is commonly used for sequential data processing. The main function of a GRU is to model and capture long-term dependencies in sequential data by maintaining an internal state, often referred to as the hidden state *(h_t)*. GRUs have a gating mechanism that controls the flow of information within the network. They contain two gates: the update gate and the reset gate. These gates determine how much information from the current input and the previous hidden state should be used to update the hidden state at each time step.



This type of architecture is used in the Decoder part. In the Encoder part we use instead a Bidirectional GRU that is formed by two GRUs that works in the opposite directions and allow to capture more information about the representation of the input.

The complete model (seq2seq) that we have defined for GRU architecture is composed as follows:



- **Encoder** RNN GRU:
  - *Input Layer:* it defines the input of the encoder (English sentences). The *shape* = (None,) indicates that the sequences could be various lengths.
  - *Embedding Layer*: from our vectors of dimension (64, 18), it creates 30000 (*vocabulary_size*) embeddings of dimensions 256. The parameter *mask_zero* = True

allows the embedding to not consider the 0s elements of the input vectors (because Text Vectorization layer have padded the sequences < 18 words).
    - *Bidirectional GRU Layer*: it is a bidirectional RNN unit that works with 2 GRU. The output of the two GRU will be summed (*merge_mode* = sum).

- **Decoder** RNN GRU:
    - *Input Layer*: it has the same function of the encoder's one, but we give as inputs Italian sentences (that are the actual target).
    - *Embedding Layer*: it has the same functions of the encoder's one.
    - *GRU Layer*: it is a RNN unit that has as input the Italian sentence (actual target) and as parameter *hidden_state* the output of the encoder part (which corresponds to the initial state of the decoder).
    - *Dropout Layer*: it removes temporarily 50% of the units during training to avoid overfitting.
    - *Dense Layer*: it is a classic Multilayer Perceptron that allows the prediction of the next token through the softmax (as *activation function*).

After the definition of the model, called *model_seq2seq_GRU*, we compile it choosing as parameters:

- *Optimizer:* Adam
- *Loss:* Sparse Categorical CrossEntropy
- *Metrics:* Accuracy

Then we train with *fit* method, using as *validation_data* the validation dataset defined previously.

To see the translated sentences predicted by the model, we built a function *decode_sequence_GRU* that vectorize the *input_sentence* (which corresponds to sentences taken from the test pairs), predict the translated vector sentence and convert the translated vector into words.

Finally, we obtain the obtain the translated predicted sequences. Here we attach some examples that we have printed:

```
Let's start with you.
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 26ms/step
[start] cominciamo con voi [end]
-
I hurt my back.
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 26ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 25ms/step
[start] mi sono fatto male alla schiena [end]
```

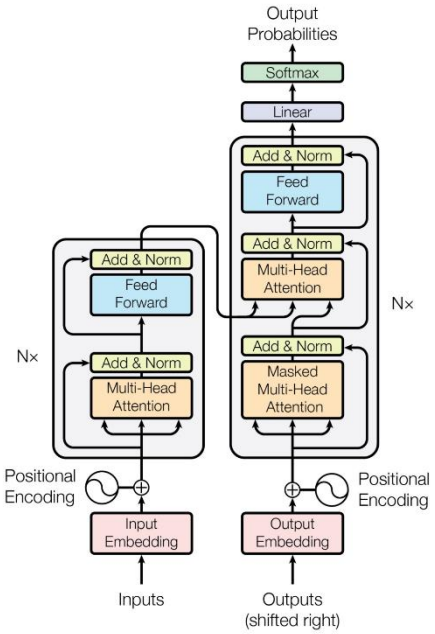
Seq2Seq with Transformers

The **Transformers** architecture, introduced by Google in 2017 in the "Attention Is All You Need" paper, is a powerful model for sequence-to-sequence tasks, such as machine translation.

It revolutionized the NLP field, but also other deep-learning fields, for 2 main reasons:

- Firstly, it avoids the sequential nature of the recurrent neural networks (RNNs) by working in parallel, supplying all the tokens of the input sequence at once. This ensures faster performances, especially for big corpus.
- Secondly, since it relies on multi-head attention, so it performs multiple times the self-attention mechanism, is more capable of detecting long-term dependencies, so when a term depends on another one that is far from it in the sentence (like a pronoun and a subject).

The Transformers architecture consists of two main components: the **Encoder** and the **Decoder**.



The *TransformerEncoder* is responsible for encoding the input sequence. So, in this case, the English sentences.

It takes as input the vector sum of the embedding of a token of the input sentence and the embedding resulting from the Positional Embedding layer, which contains the positional information of the token in the sentence.

Then the input passes through the *TransformerEncoder* layer, which performs multiple times the self-attention mechanism, allowing each word to focus on other words in the input sequence.

The output of this layer is then added to the input of the encoder, using a skip connection and an Add layer, in order to combine them.

And finally, this combination is normalized, using a *LayerNormalization*, which is a more efficient version of *BatchNormalization*, which scales the outputs of a layer (the activation values), to be in a distribution with mean 0 and standard deviation equals to 1. This ensures a faster and more efficient training, which is scale invariant.

This will be the input of a feed-forward network, composed by 2 different dense layers, which parse it. The output, as after the multi-attention layer, is first combined with the input of the FFN and then we've the normalization.

The *TransformerDecoder* is responsible for generating the output sequence, in this case, the Italian sentences.

As for the encoder, it receives as input the vector sum of a token's embedding and its positional embedding, derived from the Positional Embedding layer.

Then it performs masked multi-head attention, that is works like a multi-head attention layer, except for the tokens that are compared. In fact, only the tokens before the current one in the sentence are taken into consideration for the attention computations, so it prevents attending to future positions in the output sequence.

Then the resulting output is combined with the input of the masked multi-head attention layer, using again an Add layer and a skip connection, and finally the combination is normalized using a *LayerNormalization*.

This output and the output of the encoder part are then the inputs of a multi-head attention layer, which produces an output that is added to the normalized output of the masked multi-head attention, using again a skip connection and an Add layer. And then, using another *LayerNormalization*, this combination is normalized and passed to the Feed Forward Network, which produces an output that, as in the encoder, is first combined with the input of the FFN and then this combinated output is normalized.

The output of the decoder is then passed to Linear layer and a Softmax layer, which outputs a vector of size *V*, where V (vocabulary) is the number of unique words in the corpus, and each component is the probability of the corresponding word to be the predicted word of the model.

Once the model is created, it is compiled, so it's transformed into a computational graph. It is compiled with:

- *Adam optimizer:* function used to update the parameters of the model (weights and bias) in an efficient way, using momentum.
- *sparse categorical cross-entropy loss*: function which computes the error of the model in predicting the token. Sparse indicates that we are comparing the integer values of the words.
- *accuracy metric*: metric which indicates the number of right predictions over the total number of predictions.

Here, there are some examples:

```
Do you think it's going to rain tomorrow?
[start] lei pensa che pioverà domani [end]
-
A cat scratched me.
[start] mi ha regalato un gatto [end]
-
Come to my room between three and four.
[start] venga nella mia stanza tra quattro e quattro [end]
```

Results

To assess and compare the performances of these different models, since each of them provide their own translations, we have different metrics to use. We've 3 different types of text available: the source(the input sequences), the models translations and the references(the given human translations), so we can use reference based evaluation metrics, in which we compare a model translation with one or more references.

This type of evaluation can be of 2 different types:

- **human-based evaluation**: different monolingual human evaluators can perform 2 different evaluation tasks: the direct assessment, in which each evaluator gives a score to the sentence, to express which is a good one or a bad one; or the ranking, in which the evaluator rank the translations of a sentence from different models, from the best one to the worst. So in these tasks the humans can evaluate the adequacy, so if the output has the same meaning of the input sentence, but also the fluency, so they check the grammar and idiomatic word choices. In essence, they evaluate how good the translation is. But this type of evaluation is time consuming, subjective and not reusable.

- **automatic-based evaluation**: with this type of evaluation, we compare the human reference translation and the machine predicted translation. To perform this, we can have different metrics.

We have accuracy, which returns the percentage of right predictions over the total number of predictions, but it is not so used, since it is not so reliable. In fact, it doesn't consider the most relevant information like the context, the other of the words and the semantics.

The most used and important metrics for a machine translation task are WER, BLEU and METEOR.

WER (word error rate), counts the number of steps to transform the predicted sentence into the reference one, returning a score between 0 and 1. The less the score is, the better the translation is.

We also have the variant **TER** (translation edit rate), in which we count also another step, called shift, in which we shift a word sequence (of 1 or more words) to the right or to the left.

BLEU (Bilingual Evaluation Understudy) measures the similarity between the predicted translation and the 1 or more reference translations, computing the precisions of the n-gram (of different sizes, from 1 to 4 words in the window). It also takes into account also a brevity penalty for too short translations.

METEOR (Metric for Evaluation of Translation with Explicit Ordering) is a metric which gives credits to the synonyms, but requires language specific tools, like dictionaries, that some languages don't have.

As we've mentioned in the model section, when we've talked about the performance on the test set, we've printed some translated sentences from the different models. This gives the possibility to a human evaluator to judge, as we said above, the adequacy and the fluency, and so to have immediately clear how good the translation is.

Here there are the performances we obtain:

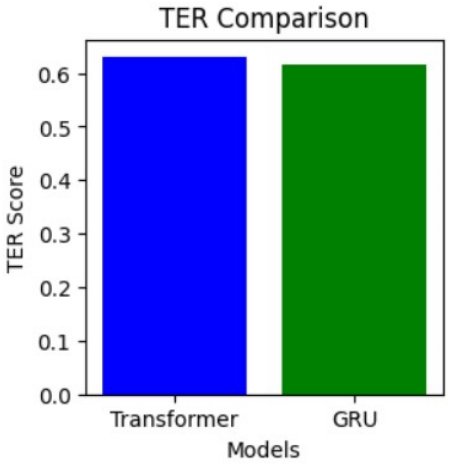
```
Transformer WER: 0.7061719120165547
GRU WER: 0.6898866182230472

Transformer TER: 0.6310351711641197
GRU TER: 0.6150318415015787

Transformer METEOR Score: 0.51279183586135
GRU METEOR Score: 0.5313471674468099

Transformer BLEU: 22.787599645739796
GRU BLEU: 24.906798815937012
```

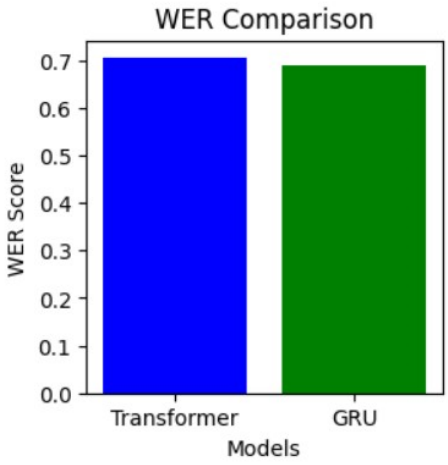
As we can see the transformer has made less accurate translations than the GRU. This, even if it's unusual happens because, since limited resources available with the free plan of Google Colab, we were forced to use few epochs (5). The Transformer is a complicated and very powerful model, which performs a lot of internal computations and for that reasons it requires many epochs to have great performances. Instead GRU, is faster and requires less resources to be train, since it has an easier architecture, with only 2 gates and an hidden state. So, it is more effective than the Transformer when we have few epochs.

Another reason is that the sentences that we've are composed by only 15 tokens, so we don't have too long-term dependencies, and for that reason even a GRU can catch them effectively.

However, in general the Transformers are a better architecture, which leads to better performances, since it exploits the attention layers that lead it to focus more, for each word, on the most important parts of the sentence and so it can deal better long-term dependencies that lead to better translations than a simpler RNN, like GRU or LSTM.

Then, we've computed the different automatic metrics for machine translation that we've stated above.



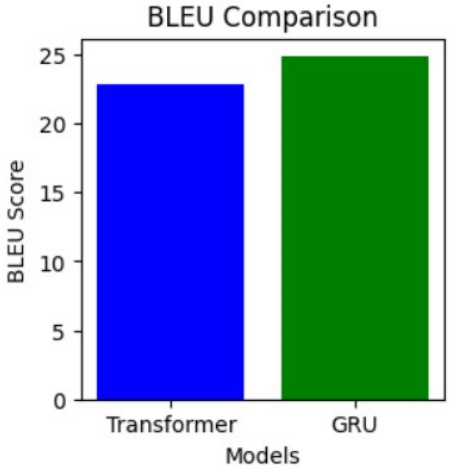


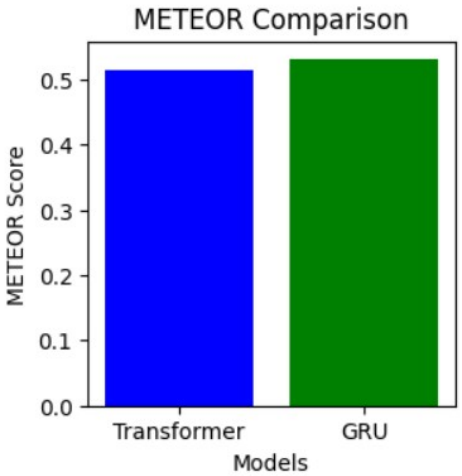
WER and TER, as we can see from the results and from the histograms, have better performances for the GRU sequence to sequence model, rather than for the Transformer, for the reasons mentioned above.

In fact, the WER and TER scores for the Transformer are higher than the ones for the GRU.

To develop WER we've used the *Jiwer* library, a specific library for WER. While, for the development of TER we've used the library *NLTK* (natural language toolkit), a specific library for NLP tasks, like tokenization, lemmatization, classification and evaluation of the performances.

This result can be affirmed also by looking at the other metrics: METEOR and BLEU.





The results for these 2 metrics in fact are higher for the GRU rather than for the Transformer.

As for the WER and TER, we can visualize these in the 2 corresponding histograms, where the bar corresponding to the Transformer model is lower than the one for the GRU model.

METEOR was developed by us using the *NLTK* library as for TER; while BLEU was developed by using the library *sacrebleu*, a library specific for the automatic metrics of the machine translation tasks.

Conclusion

To sum up, we can describe our project in 4 different steps:

1. Upload the text file and compute the most important **statistics** of the pairs.
2. **Preprocess** the Italian and English sentences through:
    a. Tokenization: divide the words, put in lower case, remove punctuations signs.
    b. Word Vectorization: define vocabulary size, maximum sequence lengths and compute the word vectorization thanks to Vectorization Layers of Keras.
    c. Make the Dataset: define the batch size and create a Keras Dataset of three features: English Input, Italian Input and Italian Output. Each cell corresponds to a vector 15x64.
3. **Create** and **train** the Models:
    a. RNN GRU: define the Encoder part (Bidirectional GRU) and Decoder part (GRU). Compile and fit the model to see some examples of the translated predicted sentences from the test set.
    b. Transformers: define the Positional Embedding, Transformer Encoder and Transformer Decoder classes. Then define the Encoder and Decoder part of the whole architecture. Also in this case, compile and fit the model to print some examples from the test pairs.
4. **Evaluation** of the model throughs some automatic evaluation metrics used for Machine Translations:
    a. WER and TER
    b. BLUE score
    c. Meteor

As regarding the coding work, we have decided to split some parts and to do together other parts:

- Together: Preprocessing of the text.
- Patrizio: Transformer model and Evaluation of the models.
- Lorenzo: Statistiscs and RNN GRU model.

The most difficult part of the project is without any doubts the **Pre-processing** part. Although there are lots of methods, we have studied that we could use to make the pro-processing of the text (as ELMo, FastText, Bag of Words, Byte Pair Encoding), we choose to utilize a method that we have in our knowledge. However, we encounter some difficulties in implementing the vectorize words with the models (RNN and Transformers) and for this reason we make a Keras dataset to complete the pre-processing of the input.

Finally, we can claim that the performances we obtain are not very good results. However, we can justify

these performances by stating that we try to construct a real translator without too many tools. So, we don’t use pretrained models and some tools that Professor Raganato provide us (as Open NML).

Moreover, we have to consider also that the whole project is made on Colab: this implies lots of computational and time-resource limits. If we would have some physic GPUs or added GPU units of Colab, we could have run the program for more time, rising the epochs and increasing some parameters such as Maximum Sequence Length or Batch Size. In this case we would definitely obtain better performances.

In conclusion, we have found the NLP course to be highly engaging and captivating, particularly with regards to the development of our project. We have acquired a wealth of knowledge that undoubtedly holds significant value for our future professional endeavours.