Human Language Technologies
Comparison of Machine-Translations Models

Gennaro Daniele Acciaro - 635009 - g.acciaro@studenti.unipi.it - MSc in Artificial Intelligence

Roberto Esposito - 636672 - r.esposito8@studenti.unipi.it - MSc in Artificial Intelligence

Giuliano Galloppi - 646443 - g.galloppi@studenti.unipi.it - MSc in Big Data Technologies

Human Language Technologies course (649AA) - 9 CFU - A.Y. 2021/2022

Instructor: Professor Giuseppe Attardi

GitHub Project: *Comparison of Machine-Translations Models*

Abstract

"Machine Translation" (MT) refers to that process of automatically translating a text from one "source" language to another "target" language. In this report we explore two possible approaches to deal with this task: Statistical and Neural. In particular, we implemented several models from scratch and compared them with models that represent the state of the art.

Contents

1 Introduction

For *Machine Translation* (MT) we mean that process of automatically translating a text from one "source" language to another "target" language. Since the 1950s, the Machine Translation task has been one of the main tasks in the Natural Language Processing scenario. Before the introduction of neural networks, this type of task was handled using a statistical approach (*Statistical Machine Translation*): it was relying on calculating the probability that a certain source word could be translated into another target word given the context of the sentence.

Until the early 2000s, the SMT approach was commonly used. Nowadays, state-of-the-art architectures use (Deep) Neural Network approaches (*Neural Machine Translation*), as they are able to achieve better results.

Machine Translation is a Sequence-to-sequence task and the de facto standard for kind of tasks is the **encoder-decoder** architecture where an encoder neural network encodes a source sentence into a fixed-length vector. A decoder, then, takes this fixed-length vector and generates a translation of the original sentence, but a more detailed description will be done in the next sections.

In this project, we made a comparison between different models, both neural network-based and statistical approach-based. In particular, this report is composed as follows:

- **Prior Work:** we will discuss the technologies that are used by the architectures that represent the state of the art;
- **Dataset:** we describe the dataset and the metrics used to comfort the various models we have implemented;
- **Implementation Phases:** we show the details of the architectures we have implemented;
- **Results:** we show our experiments and results;
- **Conclusions:** we evaluate the results;

1.1 GUI

The project was accompanied by a GUI. It was realised in order to show and test the various models, proving the results obtained and described. Each button on the top allows to select the model, that on the middle supply three examples and the Translate button translate the sentence giving the result of the model on display.

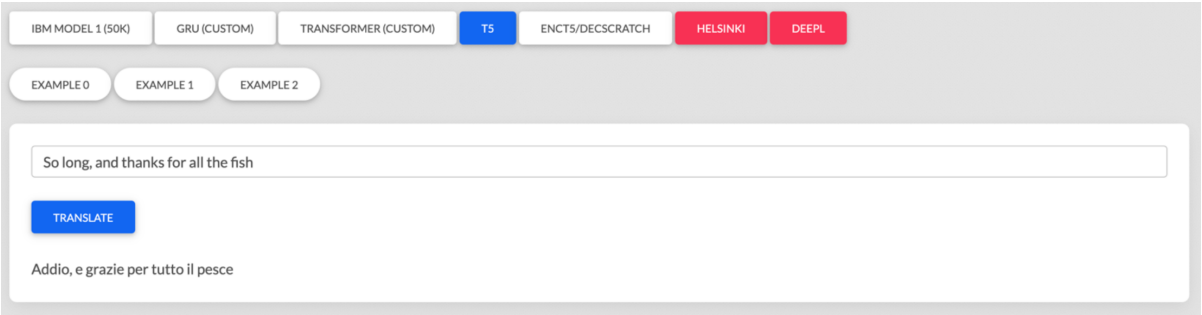

Figure 1: Project GUI

2 Prior Work

In the following section we are going to analyze the different technologies used by MT architectures:

- Statistical Machine Translation: we will talk about the IBM Model;
- Neural Machine Translation: it uses neural networks for Machine Translation tasks;

2.1 Statistical Machine Translation - SMT

The first approach that we analyze is the SMT that was introduced by Warren Weaver in 1949 [12]. The basic idea behind this approach was that the language had an inherent logic that could be treated in the same way as the logical mathematics. It contrasts with the rule-based approaches to machine translation as well as with example-based machine translation, which were the translation methods used up to that time. How the statistical machine translation works basically is that a document is translated according to the probability distribution $p(e|f)$ that a string e in the target language (for example, English) is the translation of a string f in the source language (for example, Italian).
Modeling the probability distribution $p(e|f)$ has been approached in a number of ways. One approach is to apply **Bayes Theorem**, where e_i is the $i-th$ sentence:

$$\begin{aligned}
\overline{E} &= e_1, e_2, e_3, \ldots, e_n \\
\overline{E} &= \arg\max_E P(E|F) \\
&= \arg\max_E \frac{P(F|E)P(E)}{P(F)} \\
&= \arg\max_E P(F|E)P(E).
\end{aligned}$$

For a rigorous implementation of this one would have to perform an exhaustive search by going through all strings e in the native language. Performing the search efficiently

is the work of a machine translation decoder that uses the foreign string, heuristics and other methods to limit the search space and at the same time keeping acceptable quality. Next we will analyse how a decoder works.

SMT benefits

- More efficient use of human and data resources
- There are many parallel corpora in machine-readable format and even more monolingual data.
- Generally, SMT systems are not tailored to any specific pair of languages.
- Rule-based translation systems require the manual development of linguistic rules, which can be costly, and which often do not generalize to other languages.
- More fluent translations owing to use of a language model.

SMT drawsback

- It requires a lot of time to calculate all the probabilities given a dataset; moreover, it requires a lot of storage in order to store these probabilities.
- Inference time can be costly.
- It does not consider the connections between the words in a sentence.

2.1.1 IBM Model 1

In the following section it is shown the problem of modeling the conditional probability $p(i|e)$ for any Italian sentence $i = i_1, \ldots, i_m$ compared with an English sentence $e = e_1, \ldots, e_l$.
The probability $p(i|e)$ involves two choices:

- the length m of the Italian sentence;
- the choice of the words $i_1, \ldots, i_m$;

From now on, we have the assumption that the length m is fixed and the focus is on:

$$p(i_1, \ldots, i_m|e_1, \ldots, e_l, m)$$

in other words the conditional probability of the words $i_1, \ldots, i_m$ conditioned on the English string $e_1, \ldots, e_l$ and the Italian sentence length m.

It is very difficult to model the probability described above. The central idea of the IBM models was to introduce some additional variables to the problem. The alignment variables are $a_1, \ldots, a_m$ that is one alignment variable for each Italian word in the sentence where each of those variables can take any value in $\{0, 1, \ldots, l\}$.
Our conditional distribution will be the following one:

$$p(i_1, \ldots, i_m, a_l, \ldots, a_m | e_1, \ldots, e_l, m)$$

The computation of the probability $p(i_1, \ldots, i_m | e_1, \ldots, e_l, m)$, thanks to the marginalization, becomes:

$$p(i_1, \ldots, i_m | e_1, \ldots, e_l, m) = \sum_{a_1=0}^{l} \sum_{a_2=0}^{l} \cdots \sum_{a_m=0}^{l} p(i_1, \ldots, i_m, a_1, \ldots, a_m | e_1, \ldots, e_l)$$

Each alignment variable a_j means that the Italian word i_j is aligned to the English word e_{a_j}. Notice that we have a many-to-one alignment in the sense that more than one Italian word can be aligned to a single English word.
Furthermore it is defined a special `NULL` word (e_0); so $a_j = 0$ means that the word i_j is generated from the `NULL` word.

2.2 Neural Machine Translation - NMT

Since the Statistical Machine Translation was not able to produce useful translation during the years the researchers tried to use neural networks to solve Machine Translation tasks.

NMT benefits

- Better performance
- A single architecture to be optimized
- Requires much less human engineering effort

NMT drawsback

- NMT is less interpretable.
- NMT is difficult to control.

2.2.1 Vanilla RNN

Since machine translation is a sequence-to-sequence activity, recurrent neural networks (RNNs) can be used. The simplest RNN is called Vanilla and consists of a self-loop in the single hidden layer, they were developed by Elman, 1990. The simplest way to employ a Vanilla RNN for a sequence-to-sequence task is that as long as the input sequence is not concluded, the outputs of the network are ignored. When the input is concluded, then it starts giving null values as inputs and collecting the outputs.

The hidden state summarizes information about the history of the input signal, during time. But when the time lag between input and output grows, the influence of that input on the current output is minimal. This problem occurs when we have long sequences. The cause of the long-term problem depends on the fact that a gradient propagated over too many layers tends to vanish or explode due to the fact that in an RNN the weights are shared, so the sum of contributions to the gradient across time can easily generate gradients that vanish or explode.

Gated RNNs and **Transformers** have been developed to solve this problem.

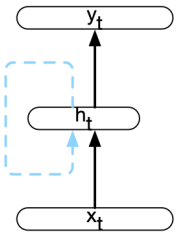

Figure 2: Simple recurrent neural network after Elman (Elman, 1990).

2.2.2 Gated RNN : LSTM

Long short-term memory (LSTM) networks, introduced by Hochreiter in 1997 [7], divide the context management problem into two sub-problems: removing information no longer needed from the context, and adding information likely to be needed for later decision making. The key to solving both problems is to learn how to manage this context c_t rather than hard-coding a strategy into the architecture. LSTMs accomplish this by first adding an explicit context layer to the architecture (in addition to the usual recurrent hidden layer), and through the use of specialized neural units that make use of **gates to control the flow of information** into and out of the units that comprise the network layers. These gates are implemented through the use of additional weights that operate sequentially on the input, and previous hidden layer, and previous context layers.

In particular we can find three kind of gate:

- **Input gate:** the input gate controls how inputs contribute to the internal state

- **Forget gate**: the forget gate controls how past internal states c_{t-1} contributes to the actual internal state c_t .
- **Output gate**: the output gate controls what part of the internal state is propagated out of the cell.

The gates use the sigmoid activation function because it arises from its tendency to push its outputs to either 0 or 1. Combining this with a pointwise multiplication has an effect similar to that of a binary mask. Values in the layer being gated that align with values near 1 in the mask are passed through nearly unchanged; values corresponding to lower values are essentially erased.
A LSTM layer works in this way: first we need to compute the input and the forget gates, using the actual input x_t and the previous output state h_{t-1}:

$$I_t = \sigma\Big(W_{Ih}h_{t-1} + W_{input}x_t + b_{input}\Big)$$
$$F_t = \sigma\Big(W_{Fh}h_{t-1} + W_{F}x_t + b_{forgot}\Big)$$

Recall that the input gate tells us how much we have to consider the input, while, the forget gate tells us how much we have to consider the previous state. Once we have I_t and F_t, we can compute the potential g_t and then we can use that to calculate the internal state c_t:

$$g_t = tanh\Big(W_h h_{t-1} + W_{in}x_t + b_n\Big)$$
$$c_t = F_t \odot c_{t-1} + I_t \odot g_t$$

where $\odot$ is the element-wise multiplication. At the end, we can finally compute the output gate o_t and the output state h_t

$$o_t = \sigma\Big(W_{Oh}h_{t-1} + W_{Oin}x_t + b_o\Big)$$
$$h_t = o_t \odot tanh(c_t)$$

2.2.3 Gated RNN : GRU

Gated Recurrent Units (GRUs)(Cho et al., 2014)[4] are a lighter version of a LSTM, indeed, GRU uses only two gates, instead of three (in the LSTM case). In a GRU there is no internal state. These two gates are:

1. **Reset gate**: it acts directly on the output. It controls how much of the previous state we might still want to remember.
2. **Update gate**: it allows us to control how much of the new state is just a copy of the old state.

A GRU layer works in this way:

– **Reset gate:** it acts directly on the output. It controls how much of the previous state h_{t-1} we might still want to remember in h_t. This control is made using a parameter z_t.

$$h_{new} = 1 - z_t \odot h_{t-1} + z_t \odot h_t$$

where

$$h_t = tanh\Big(W_{hh}(r_t \odot h_{t-1}) + W_{hin}x_t + b_n\Big)$$

– **Update gate:** it allows us to control how much of the new state is just a copy of the old state. Here we can calculate z_t and r_t.

$$z_t = \sigma\Big(W_{zh}h_{t-1} + W_{zin}x_t + b_z\Big)$$

$$r_t = \sigma\Big(W_{rh}h_{t-1} + W_{rin}x_t + b_r\Big)$$

2.2.4 The Encoder/Decoder architecture

The *de facto* standard for Sequence-to-sequence tasks is the **encoder-decoder architecture**. These models are used for a different kind of sequence modeling in which the output sequence is a complex function of the entire input sequencer; they must map from a sequence of input words or tokens to a sequence of tags that are not merely direct mappings from individual words. In practice, a MT task is such that the words of the target language don't necessarily agree with the words of the source language in number or order.
So, encoder-decoder networks, are models capable of generating contextually appropriate, arbitrary length, output sequences. The key idea underlying these networks is the use of an encoder network that takes an input sequence and creates a contextualized representation of it, often called the context. This representation is then passed to a decoder which generates a task-specific output sequence.

- An **encoder** that accepts an input sequence,and generates a contextualized representations. LSTMs, GRUs, RNN, convolutional networks, and Transformers can all be employed as encoders.

- A **context vector**, c, which is a function and conveys the essence of the input to the decoder.

- A **decoder**, which accepts c as input and generates an sequence of hidden states to which corresponds sequence of output states y_m that can be obtained. Just as with encoders, decoders can be realized by any kind of sequence architecture.

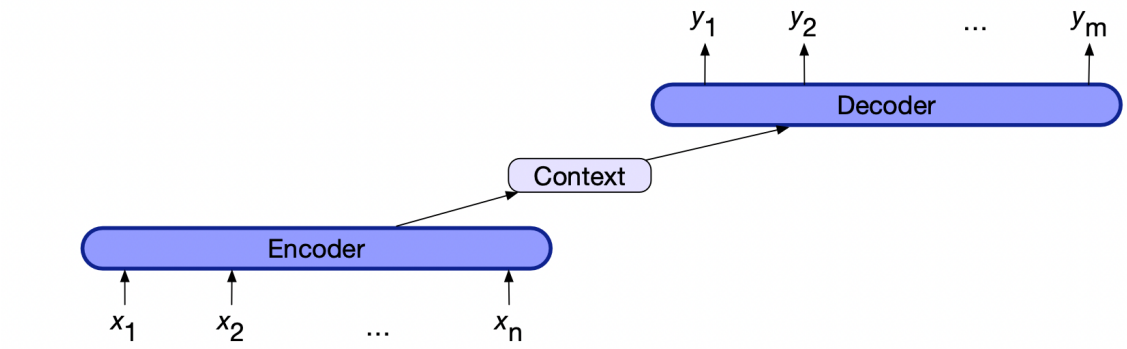

Figure 3: MT Encoder-Decoder network

2.2.5 Transformers

The critique of the use of Recurrent Neural Networks is that they require a lengthy walk-through, word by word, of the entire input sentence, which is time-consuming and limits parallelization (bottleneck problem). Moreover, even with Gated RNN, we still have the long-term dependecy problem. To solve it we embrace the *attention mechanism*. It considers associations between every input word and any output word and uses it to build a vector representation of the entire input sequence.

Here ***transformer*** architecture comes to our aid. Complex and powerful, a transformer is a deep learning model that adopts the mechanism of self-attention, differently weighting the significance of each part of the input data, that still follows the encoder-decoder paradigm, diversifying into both sides that now are composed by multiple layers and sub-layers:

- **The encoder** consists of encoding layers that process the input iteratively one layer after another;
- **The decoder** consists of decoding layers that do the same thing to the encoder's output.

The *self-attention mechanism*, introduced by Vaswani et al. [11] that defines self-attention for a sequence of vectors h_j (of size $|h|$), packed into a matrix H, as:

$$\text{self-attention}(H) = softmax\left(\frac{HH^T}{\sqrt{|h|}}\right)H \quad .$$

The encoder's inputs first flow through a self-attention layer, a layer that helps the encoder look at other words in the input sentence as it encodes a specific word. Then, the outputs of the self-attention layer are fed to a feed-forward neural network and it is

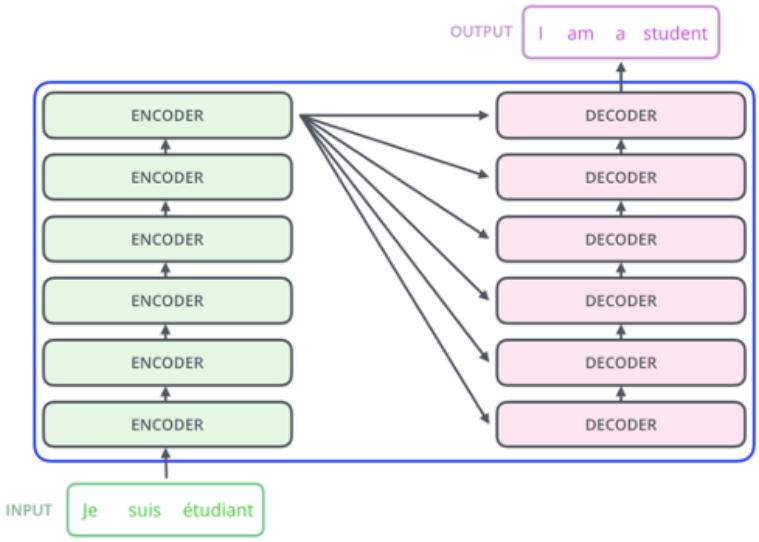

Figure 4: Transformer: Encoding-Decoding layers

independently applied to each position. While, the decoder has both those layers, it also add between them an attention layer that helps the decoder focus on relevant parts of the input sentence.

Both the encoder and decoder layers have a feed-forward neural network for additional processing of the outputs and contain residual connections including layer normalization steps.

The transformer adapts perfectly itself to sequence-to-sequence learning tasks because the attention allows those models to process successfully the sequences that are longer and more complex with respect to the ones handled by RNNs.

The basic idea is that the human being does not translate literally one word at a time, but he moves in the different words of the sentences paying attention to them and that is exactly what is done with Neural Attention.
In the following Figure 5 it is reported the architecture of the Transformer.

The decoder looks very similar to the encoder except that it has an extra attention block that is inserted between the self-attention block applied to the target sequence and the dense layers of the exit block.
Let's describe in more details those components. The encoder is composed of 6 identical layers and each layer has two sub-layers. The first is a multi-head self-attention mechanism and the second a position-wise fully connected feed-forward network. It is used a residual connection around each of the two sub-layers followed by a layer normalization. Since the decoder is very similar to the encoder, it has 6 identical layers too, but in addition it has a third sub-layer that performs multi-head attention over the output of the encoder stack.

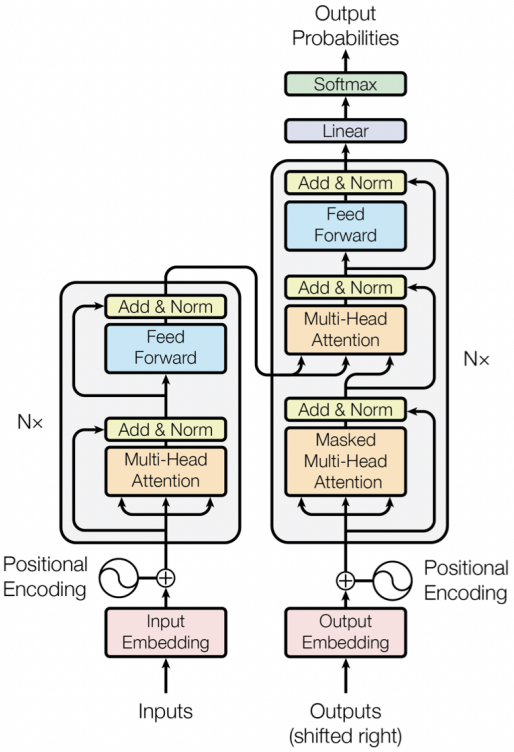

Figure 5: Model architecture of Transformer

As follows it is reported in the Figure 6 the two types of attention used in the transformer structure.

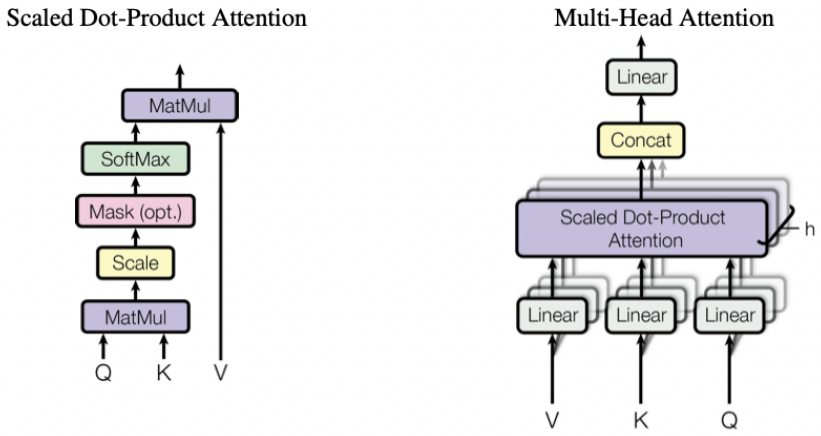

Figure 6: On the left it is reported the Scaled Dot-Product Attention and on the right the Multi-Head Attention

A detail that we need to take into account which is crucial for the success of the training of a sequence-to-sequence transformer is the causal padding. This architecture looks at the entire target sequence at once and it is different from the RNN which looks at its input one step at a time and thus will only have access to steps $0, \ldots, N$ to generate the output step N.

Notice that both the encoder and the decoder are shape-invariant so we could stack many of them one after the other to create a more powerful model, but it clearly grows in complexity.

2.2.6 BERT

The state-of-the-art results in NLP tasks are mostly of them achieved by BERT that stands for *Bidirectional Encoder Representations from Transformers* and it was presented by Devlin J. et al. [5] in 2018.
The key technical innovation of BERT is that it applies the bidirectional training of transformer to language modelling. With respect to the other models which looked at the text sequence either from left to right whereas here is a combination of left-to-right and right-to-left training.
The structure of the transformer is made of two different mechanisms which are: an encoder that reads the text input and a decoder that produces a **prediction** for the task. Since BERT's goal is to generate a language model, only the encoder mechanism is necessary.
Before the word sequences are fed into BERT, 15% of the words in each sequence are substituted with a `[MASK]` token. After that the model try to predict the original value of the masked words, based on the context provided by the others non masked words in the sequence.
From a more technical point of view, the prediction of the output words requires:

- Adding a classification layer on the top of the encoder output
- Multiplying the output vectors by the embedding matrix, transforming them into the vocabulary dimensions
- Calculating the probability of each word in the vocabulary with a softmax

In the following Figure 7 are reported the steps described above. The BERT loss function takes into consideration only the prediction of those masked values and ignores the prediction of the non masked words. It turns out from this that the convergence of the model is slower with respect to the directional models.

During the training stage, the model takes as input pairs of sentences and tries to learn if the second sentence in the given pair is the subsequent sentence in the original document

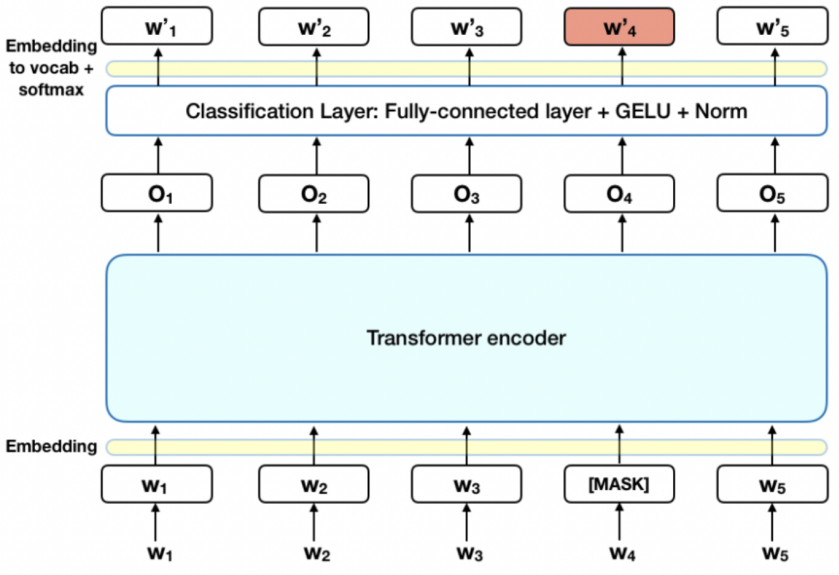

Figure 7: BERT Architecture

starting from the first one. During training, 50% of the inputs are a pair in which the second sentence is the subsequent sentence in the original document, while in the others a random sentence is chosen from the dataset as second sentence. The assumption is that the pairs containing random sentences will not have any connections from the first sentence.

Before a pair is given as input to the model, the input is processed as follows:

- A `[CLS]` token is inserted at the beginning of the first sentence and a `[SEP]` token is inserted at the end of each sentence
- A sentence embedding is added to each token, it indicates if the sentence is the first one or the second one
- A positional embedding is added to each token to indicate its position in the sequence

To predict if the second sentence is indeed connected to the first, the entire input sequence goes through the transformer model then the output of the `[CLS]` token is transformed into a 2×1 vector using a classification layer and in the end it is calculated the probability with the softmax.

There are two models of BERT:

- **BERT base**: it has 12 layers, 768 hidden size and 12 heads attention; the total number of parameters is $110M$, it was chosen to have the same size as OpenAI GPT in order to make comparisons with the existing model

- **BERT large**: the larger version has the double number of layers, 1024 hidden size, 16 heads attention and in this case the total number of parameters of the model is 340M

2.2.7 Text-to-Text Transfer Transformer (T5)

In 2020 Google came out with a new NLP model called Text-to-Text Transfer Transformer ($T5$) presented in the following paper [9].
The framework developed is able to perform many tasks as:

- **Machine Translation**: it translates a sentence from a language to another;
- **Classification Task**: it performs text classification for activities as sentiment analysis;
- **Regression Task**: it can perform tasks as computing the affinity between two sentences;
- **Text Summarization**: given a text it does a summarization of the input;

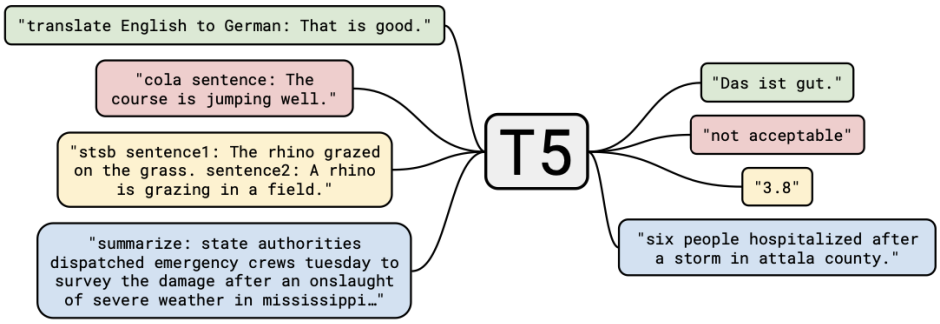

Figure 8: Text-to-Text Framework.

T5 was pre-trained on the Common Crawl Web Extracted Text dataset. Some data are removed from the dataset if the sentence did not end in a terminal punctuation mark. The model removes redundant sentences by taking a sliding window of 3 sentence chunks and all the others such that only one of them is kept in the dataset.

In order to train a single model on different set of tasks the input format is cast into a "text-to-text" format that is a task where the model is fed some text for context or conditioning and is then asked to produce some output text. The model is trained with a maximum likelihood objective independent from the task. To specify which task the model should perform it is added a task-specific prefix (a string) to the original input

sequence before feeding it to the model.
For text translation if the input is the sentence "This is good." from English to German becomes "translate English to German: This is good.". If we have a text classification task then the model should only predict a single word that corresponds to the target label.

It turned out that using the standard encoder-decoder approach proposed by Vaswani achieves good results on both generative and classification tasks. The baseline model is designed in such a way the either the encoder either the decoder are similar in size and configuration with respect to `BERT` base. More in details, both the structures consists of 12 blocks (each block contains self-attention, optional encoder-decoder attention and a feed-forward neural network). The feed-forward neural network consists of a dense layer with an output dimensionality of 3072 followed by a ReLU nonlinearity and another dense layer. Furthermore it is used for regularization the dropout probability of 0.1.

Google developed different models using different range of size:

- **Base**: it is the baseline model and it has roughly 220 million of parameters;
- **Small**: it is considered a smaller model and this variant has around 60 million of parameters;
- **Large**: since the base model use a `BERT` based-sized encoder and decoder, it is considered a variant where the encoder and the decoder are both similar in size and in structure with respect to to `BERT large`; this model has about 770 million of parameters;
- **3B** and **11B**: these two variants of the base model are obtained increasing the number of layers and multi-head attention. It scaled better with respect to the others because the TPUs are most efficient for large dense matrix multiplication.

3 Dataset and Metric

Dataset The chosen dataset was *anki corpus* from manythings.com [2022][2]. It contains **354238** records formatted as: `(english sentence, english sentence translated to italian)`. This dataset was divided into train (247968 sentences), validation (53135 sentences) and test set (53135 sentences).
After we have analyzed our data we discovered that 97.85% of the sentences in this dataset are relatively short (less than 10 words), Figure 9 details the densities of the n-grams for the Italian part of the dataset. In our view, this introduces a bias in the model, so any neural model based on this dataset will perform better on shorter sentences.

As a matter of fact that this dataset is quite large. We used an incremental approach to perform the training of the models that we have implemented and, instead, the fine tuning for the pretrained models. This means that we started working with small portions of the train set, and increasing this size as the model under consideration it gave back us better results. In this way, we were able to keep the first compilation time of the models low, and thus find bugs and crash more easily. Once we were sure that the models worked, we performed model selection using the train set (and evaluating the models on the validation set).

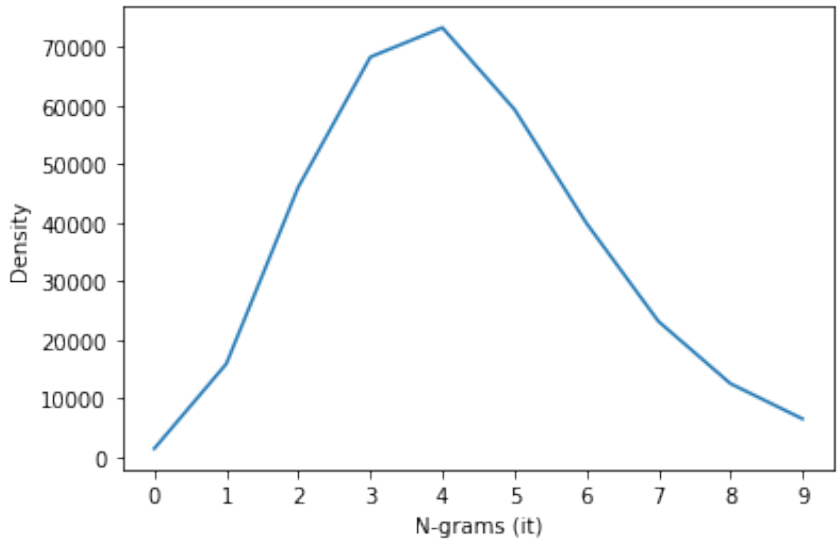

Figure 9: Dataset density

No preprocessing method was applied to the dataset.

Metric In order to compare our Machine Translation models, we used a metric called BLEU [8] (bilingual evaluation understudy) that is one of the most popular metric for evaluating machine-translated text. The output of this metric is between 0 and 1. We used the `evaluate` library from HuggingFace to implement it.

```
def the_metric(target, output):
    import evaluate
    predictions = [output]
    references = [target]
    bleu = evaluate.load("bleu")
    results = bleu.compute(predictions=predictions,
                           references=references)["bleu"]
    return results
```

Listing 1: BLEU usage

Furthermore in order to obtain a more accurate score we have normalized the data before computing the BLEU, because even small differences, like punctuation marks or spaces, in the output sentence with respect to the target could corrupt the results.

4 Implementation Phases

In the following section we will talk about the implementation phases. The models we have used are described in the previous part of this report.
The machine translation models developed are:

- IBM Model 1
- GRU
- Transformer
- Fine-tuning of T5
- T5 Encoder - Decoder from Scratch
- Bert2Bert

4.1 Libraries and Setup

In order to implement our models we have used several libraries. The main ones are:

- `tensorflow`: it is an open-source library used for machine learning and artificial intelligence developed by Google. It can be used to train and inference of deep neural models; in our case we used it mainly when we had to work with NMT from scratch models
- `transformers`: it is an API that allows easily the download and the training of state-of-the-art pre-trained models. This library is provided by **Hugging Face**[1] which is data science platform that provides tools that enable users to build, train and deploy ML models based on open source code and technologies, including datasets that can be retrieved.
- `flask`: it is a web application framework, we used it to deploy our models and use them with a GUI.

4.1.1 Hardware specs

The models have been trained and tested on server machine of University of Pisa, which had `Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz with 80 core` and
`4 GPUs NVIDIA Tesla P100-PCIE of 16GB each one.` Also Kaggle and Google Colab was used, both are online platform that provides virtual machines and environments to train and test models.

4.2 IBM Model 1

To realize our implementation of IBM Model 1 we started with the management of the sentence picked from the dataset: it firstly work on the sentence splitting it in a pair (english,italian) and also retrieving all the words about the two vocabularies. Next the collection of pairs is used to calculate the *probability* t of the sentence E given F, as described in section 2.1 and calculating the perplexity of the model, that is a measurement of how well a probability model predicts a sample. After this comes the **phrase alignment** step, that is necessary to train the model, collecting the counts and totals of words of both languages.

Done this, it allow us to compute the translation probability $P(F|E)$ by summing the probabilities of all possible $(l+1)^m$ 'hidden' alignments A between F and E:

$$P(E|F) = \sum_{A} P(F, A|E)$$

where A are the set of alignments $a_1, a_2, \ldots, a_j$ (j is an arbitrary the length).
A support table that contains the translation probability was created and stored on disk, to make it reusable for future experiments, using `pickle`, a library that allowed us to create a `Python Object` from data in a file `.pickle`. Our resultant file was around 6.29GB heavy and it required 4 days of computation.
Finally, after saving the support table, we defined the methods to translate English to Italian using the translation probability and computing the translation searching for most probable alignment $\hat{A} = \arg\max_{A} P(F, A|E)$.

4.3 Custom GRU

This model was also implemented from scratch. Again, it's an encoder-decoder architecture and in particular there is a Gated Recurrent Unit (GRU) layer in the encoder, and a GRU in the decoder. Between the encoder and the decoder is an attention layer, specifically we used the `BahdanauAttention` [3]. Figure 10 shows graphically how the attention works.
We chose this type of attention because the authors argue that this using an encoding of a variable-length input into a fixed-length vector squashes the information of the source sentence, causing the performance of a basic encoder-decoder model to deteriorate rapidly as the length of the input sentence increases. Instead, the proposed approach replaces the fixed-length vector with a variable-length one, so performance can improve.
Both GRU layers were initialized using Glorot initialization [6]. Moreover, during the training we implemented the Teacher Forcing technique. Teacher forcing is a strategy for training recurrent neural networks that uses target as input. We chose to use this technique because it allows the RNN to converge faster.

The parameters searched in the Random Searches were:

parameter	possible values	selected value
batch_size	[16, 32, 64, 128, 256]	128
epochs	[3, 5, 7]	5
embed_dim	[32, 64, 128, 256, 512]	512
units	[512, 1024, 2048]	1024

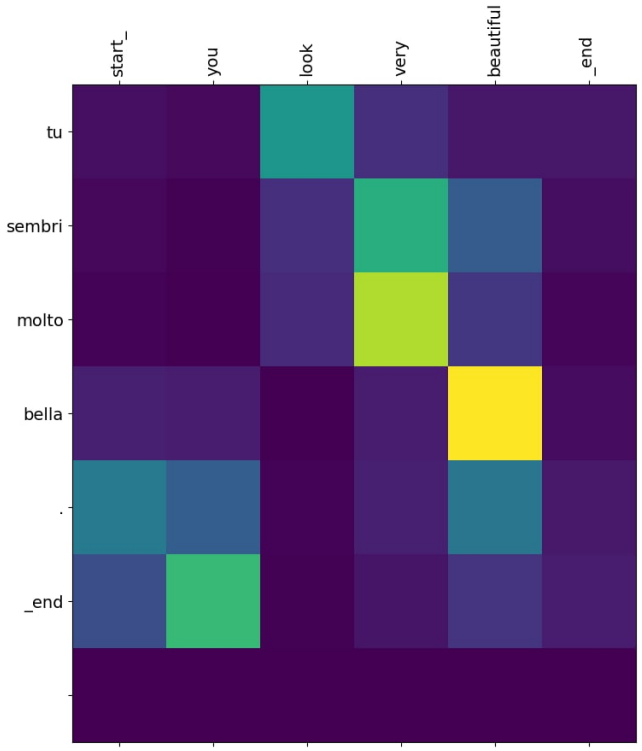

Figure 10: Attention plot

4.4 Transformers implemented from scratch

This model was implemented from scratch; it consists of a **TransformerEncoder** and a **TransformerDecoder** chained together. Both of them use the `layer.MultiHeadAttention` class of the Keras framework. The parameters of this layer (num_heads and embed_dim) were selected as hyperparameters. In particular, the encoder consists of a MultiHeadAttention layer and two `layer.LayerNormalization` layers, while the decoder consists of

two MultiHeadAttention layers and three LayerNormalization layers. This architecture structure was selected by performing preliminary empirical tests.

Recall that since this is a Transformer, there is no RNN behind this model. This provides us with benefits (we do not have to care about the problem of long-term dependency) but also disadvantages: we have to worry that the word order is respectful (using RNNs this was not necessary since the very structure of the self-loops present in RNNs defines the order between words). For this purpose we introduced the **PositionalEmbedding** class.

Tokenization was performed using the `keras.layers.TextVectorization`.

Lastly, in order to try to improve performance, we implemented a Dropout layer in the decoder. The parameters searched in the Random Searches were:

parameter	possible values	selected value
batch_size	[16, 32, 64, 128, 256]	128
epochs	[15, 20, 25]	20
embed_dim	[32, 64, 128, 256, 512]	512
latent_dim	[32, 64, 128, 256, 512]	128
num_heads	[8, 10, 12, 16]	8
decoder_dropout	[0, 0.25, 0.5, 0.75]	0.25

4.5 T5

Google's T5 was imported as it's full default class from Hugging Face using the class `transformers.AutoModel` and specifying the size of T5 , "t5-small", this variant has around 60 million of parameters. For the **tokenization** part we used `AutoTokenizer` that is a generic tokenizer class, that was instantiated as one of the tokenizer classes of the library created with the `.from_pretrained` method. The pretrained model was imported using the `transformers.TFAutoModelForSeq2SeqLM` class of huggingface. The tokenizer used was also taken from the huggingface library (`transformers.AutoTokenizer`).

The parameters searched in the Random Searches were:

parameter	possible values	selected value
batch_size	[16, 32, 64, 128]	16
learning_rate	[1e-04, 1e-05, 1e-06, 1e-07]	1e-05
weight_decay	[1e-04, 1e-05, 1e-06, 1e-07]	1e-05
epochs	[3, 5, 7]	5

4.6 T5 Encoder - Decoder from Scratch

From the previous implementation of T5, this scratch implementation differs in several aspects. The tokenization phase uses the function `from_pretrained("google/t5-v1_1-small")` from T5Tokenizer, that is based on SentencePiece and inherit main methods from the

class `transformers.PreTrainedTokenizer`.
`BertTokenizerFast.from_pretrained("dbmdz/bert-base-italian-cased")` instead is used for decoder that is based on WordPiece and in particular BERT base italian Tokenizer is specific for italian language.
In the encoder settings we have used the class `TFT5EncoderModel` from the **transformers** framework, that it was also proposed in [10].
His method `.from\_pretrained(google/t5-v1\_1-small)` uses the T5 pre-trained model, while for the decoder we have implemented a custom version defining a `class TransformerDecoder(layers.Layer)`.
Before executing the translation it was runned a **Random Search** algorithm to chose the best parameters for our model, the parameters of the random search used:

parameter	possible values	selected value
batch_size	[8, 16]	16
epochs	[20, 30]	30
embed_dim	[32, 64, 128, 256, 512]	512
latent_dim	[32, 64, 128, 256, 512]	1024
num_heads	[6, 8]	8

Table 1: Parameters of Random Search Algorithm

Only after obtaining them we put into practice the model translating sequence to sequence and evaluating it.

4.7 Bert Encoder - Decoder from Scratch

This approach is very similar to the previous one, but the main difference is in the fact that the encoder used is the one from BERT.

For the tokenization phase it is used the `BertTokenizer`, in particular the Italian BERT model proposed by MDZ Digital Library team at the Bavarian State Library, either for the encoder either for the decoder.

The encoder model is imported thanks to `TFBertModel` which is a class of `TFPreTrainedModel` which manages the configuration of the pre-trained models.
On the other hand the decoder model is built from scratch using the class `DecoderTransformer`.

In order to get the best hyperparameters it is run the Random Search over the parameters reported in the following Table 2.

Furthermore we used the dropout regularization using `dropout_rate=0.2`.

Parameter	Possible Values	Selected Value
batch_size	[8, 16]	16
epochs	[20, 30]	30
embed_dim	[32, 64, 128, 256, 512]	512
latent_dim	[32, 64, 128, 256, 512]	1024
num_heads	[6, 8]	6

Table 2: Parameters of the Random Search for Bert Encoder - Decoder from Scratch

5 Results

In the following table, we can see the different architectures side by side with the models that we have extensively described in the previous sections. The parameters used for the training and various experiments are reported in the previous section for each model.

5.1 Comparison and Evaluation of models

In the following Tables 3 and 4 we have reported the results of the BLEU score using 100 sentences chosen randomly from the dataset using the same seed.
The first table shows the training time of the statistical model, IBM Model 1, changing the number of sentences drawn from the dataset (50.000, 100.000, 200.000). Using this approach the results are the ones we expected from the theory in fact all the three models reached 0% over the BLEU metric.

Model	Training Time
IBM Model 1 - 50k	3 hours
IBM Model 1 - 100k	1.5 days
IBM Model 1 - 200k	4 days

Table 3: Comparison of the results obtained with the statistical model.

The second table contains the performance obtained with the neural models and for each of them it is reported the BLEU score and the training time required.

Moreover, we also sought comparisons with state-of-the-art models such as **DeepL** and **Helsinki**, which score respectively **34%** and **25%**. Performing an estimation using http://mlco2.github.io, in order to perform the training of the models placed in the table 4, we emitted about 3.96Kg of CO_2.

Model	BLEU	Training Time
GRU from scratch	**24.51**%	43 min. (Colab, GPU)
Transformer from scratch	21.98%	70.58 min. (Colab, GPU)
Fine-tuned T5 small	15%	116.6 min. (Colab, GPU)
T5 Encoder Decoder from scratch	14%	$\approx$ 300 min. (Colab, TPU)
Bert Encoder Decoder from scratch	0%	$\approx$ 900 min. (Colab, TPU)

Table 4: Comparison of the results obtained with different neural models.

6 Conclusion

Taking into account the results obtained, among our models GRU ranks first.
By our experiments we have noticed too much severity on the part of BLEU metric, the human being is still needed to confirm if the translation is verified or not.
As prove of this statement we report a translation case of one of our models:

- **True English**: *he's busy with his homework now*
- **True Italian**: *lui è occupato con i compiti a casa*
- **Translated sentence**: *lui è impegnato con i compiti a casa*
- **BLEU score**: *0.0*

Both the sentences are correct, in fact we can observe that *occupato* and *impegnato* are synonyms, but BLEU gives it a score of 0.0 that is wrong, thus a human is needed, so, in order to compare models, human supervision is still needed.
Overall we are satisfied about the realised work that gave us the opportunity to exploit and understand all this different kind of models, and also for their results considered the scores of state-of-art models. Probably to improve our result we could exploit some other metrics that could be more flexible to give us much precises scores.
Once we concluded the development of this project we learnt that building state-of-the-art models requires effort in terms of computing power. In order to train those models the time required is not negligible: this allowed us to develop critical thinking about working methodologies. We took the opportunity to better assimilate the encoder-decoder architecture, both for the models that already implemented it, and for those where we had to work on it from scratch, which is not trivial, broadening our perspective on a task that is increasingly under development. At the end, we also gained experience was importing and fine-tuning models from HuggingFace, which is an important skill that can also be

used for other types of projects, even quite different from Machine Translation.
For future developments, we could experiment with other types of RNNs such as the Clockwork RNN - which is an architecture designed to try to reduce the long-term dependency problem.

7 References

[1] huggingface.co. huggingface, the ai community building the future. `https://huggingface.co/`. 2022.

[2] manythings.com. `manythyngs.com`. 2022.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2014.

[4] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[6] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[7] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[8] Kishore Papineni, Salim Roukos, Todd Ward, , and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. 2002.

[9] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683, 2019.

[10] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[12] W. Weaver. Machine translation of languages. 1949.