Deep Comedy

Final Report

Luca Giuliani[1] Diego Mazzieri[2]

August 24, 2020

[1]luca.giuliani10@studio.unibo.it - 0000933791
[2]diego.mazzieri@studio.unibo.it - 0000920452

Contents

Chapter 1

Introduction

Aim of this project was to create an effective and efficient Neural Network able to reproduce the style adopted by Dante in his *Divine Comedy*. In order to do that, we downloaded the full poem from `https://raw.githubusercontent.com/genez/dante/master/dante.txt` and stored it, after some manual cleaning, in the `res` folder of our repository.

All of our code is of public domain and can be found at `https://github.com/mazzio97/DeepComedy`. The ensemble of utility functions, mostly involving model checkpoints and text processing, have been written in Python 3 and collected in the `src` folder; instead, the different architectures have been modelled and trained using Python Notebooks (`.ipynb` files) in order to exploit free GPU acceleration offered by Google Colab.

We built all the models with Tensorflow 2, both because of its large number of library functions and because of its inner support to Keras layers and utils. Due to the high weight of the model's `.h5` files, we decided not to add all of them to the repository but just the final one; anyhow, a text file containing a report of our results for each tested configuration is present, and a working code is provided for all the models we developed, so everything can be easily replicated by opening the notebooks in Google Colab.

Eventually, after having tried six different kinds of architectures, each of them instantiated in two or three different variations, and having experimented with hyperparameters as well, we chose as our definitive model a *Transformer* whose inputs are a patch of three verses and whose output is an entire new verse, with which we could achieve reasonable results in terms of text structure, verse structure (i.e. hendecasyllables), rhyming schemes and obviously avoid any kind of plagiarism to the original *Divine Comedy*.

Chapter 2

Architectures

Overall, we developed six different architectures, each one having from two to three different variations depending on the way the *Divine Comedy* was tokenized, for a total of 13 models. In this chapter we will briefly explain the characteristic of each architecture, but first we must say something about the general aspects that will recur for all of them.

Text Prcoessing

First of all, the `mark` function inside the `text_processing.markers` module was developed to insert some special markers in the original *Divine Comedy*. Together with that, we developed its complementary function `unmark`, used to get a cleaned text from the ones generated by the Neural Networks.

These markers were intended to give some structure to the text in order to make it easier for the Neural Network to understand the tokenized text. More specifically, the markers are `"=startofcantica="`, `"=endofcantica="`, `"=startofcanto="` `"=endofcanto="` and `"tercet"`.

Here is an example of a sample of text before and after the marking:

```
INFERNO

- Canto I                                  =startofcantica=
                                           =startofcanto=
Nel mezzo del cammin di nostra vita        Nel mezzo del cammin di nostra vita
mi ritrovai per una selva oscura,          mi ritrovai per una selva oscura,
ché la diritta via era smarrita.           ché la diritta via era smarrita.
                                           =tercet=
Ahi quanto a dir qual era è cosa dura      Ahi quanto a dir qual era è cosa dura
esta selva selvaggia e aspra e forte       esta selva selvaggia e aspra e forte
che nel pensier rinova la paura!           che nel pensier rinova la paura!
```

Text Tokenization

The tokenization of the text took advantage of Tensorflow Dataset tokenizers, which can be found in the package `tfds.features.text`. For our purposes, we built up three of them (a `char_tokenizer`, a `word_tokenizer` and a `subword_tokenizer`), each of which can be found in the package `text_processing.tokenizers` of our repository, and is responsible to take care not only of alphanumeric symbols but also of punctuation symbols, new-line symbols, white spaces and special markers, all separate tokens that are independently learnt and generated by the Neural Networks.

Each of the six architectures have been tested with both a *Word-Level* and *Subword-Level* tokenization, and for the first architecture we also tried a *Char-Level* implementation. At the end of the day, the *Subword-Level* tokenization was the one giving best results. In fact, as the tokenizer was fed with the *Divine Comedy* and it has the ability to split the words into the most frequent subwords, the text could be tokenized in something that vaguely resembled syllables, giving, both from a theoretical and a practical point of view, a great insight to the Neural Network when considering verse structure and rhyming scheme.

Here is an example of the subword tokenization of a sample of text:

```
=startofcantica=                          =startofcantica=
=startofcanto=                            =startofcanto=
Nel mezzo del cammin di nostra vita       Nel mez zo del cam min di nos tra vit a
mi ritrovai per una selva oscura,         mi rit rov ai per una sel va osc ura ,
ché la diritta via era smarrita.          ché la dir itt a via era sma rri ta .
=tercet=                                  =tercet=
```

Hyperparameters and Results

All the tested models had their own hyperparameters, which could be related not only to the actual architecture of the Neural Network but also to the construction of the dataset. In this last case, the hyperparameters were the same for all the models, namely:

- the window size to extract the input strings, or rather the sequence length
- the number of tokens to skip from one window to the following one, or rather the step length

- the train/validation splitting percentage
- the size of the batch
- the number of training epochs

Generally, these parameters can be inferred (e.g. for the sequence length we chose the minimum amount of tokens necessary to get at least the previous three verses entirely) or have a minor influence on the final results, thus, during our experiments to find the best set of hyperparameters, we mainly focused on the hyperparameters of the network. In any case, we automated the process in order to test the performances of each configuration/model, varying the temperature factor of the generation as well, and stored a summary of the results for each test into a series of `.txt` files placed in the `results` folder of the repository.

Optimizer and Loss Function

Each model was compiled with either the default or a custom `Adam` optimizer, where the latter case is referred to the *Transformer* model, whose specifications for a better optimizer were given in the original paper [1].

Instead, as loss function we always used the `Sparse Categorical Crossentropy` except that for the last model, the *GAN*, that obviously used the `Binary Crossentropy` of the discriminator to perform backpropagation both during the discriminator and the adversarial training.

2.1 Plain RNN

As many models for text generation (and other *Natural Language Processing* tasks as well) have been proposed all over the years, we decided to keep our first model as simple as possible to set a sort of lower bound that would have been useful for the successive architectures.

Therefore, inspired by Karpathy's work [2], universally recognized as a milestone in the field of *Natural Language Processing*, we developed the model in figure 2.1 consisting of an initial Embedding layer, mapping the tokenized input into a dense vector, which is then passed to either one or two RNN layer(s) and, eventually, to a final Dense layer, post-processed using softmax activation in order to output the probability of each token. Given that

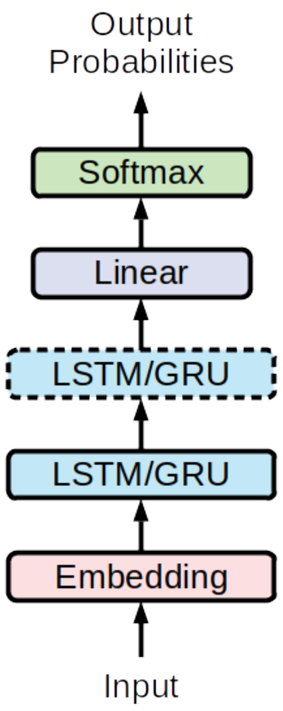

Figure 2.1: Plain RNN Architecture

standard RNN layers notoriously suffer from the so-called vanishing gradient problem, which arises in the backpropagation step, we decided to use either LSTM (Long Short-Term Memory) layers [3] or GRU (Gated Recurrent Units) layers [4], two special kinds of RNN layers developed to overcome this known problem.

Finally, as we said during the introduction of this chapter, the input and output samples we used had a fixed length, the so-called `sequence length`. One important thing to notice is that we did not use a long sample as input and just a single token as output, but the output was instead the exact input sequence shifted by one token[1], so that the network could understand the inner patterns of the text as well.

Hyperparameters and Results

As we said in the introductory part of this chapter, we instantiated this architecture in three different variations, depending on the way the input text was tokenized, namely a *Char-Level* model, a *Word-Level* model and a *Subword-Level* model.

Each of these variations had the same, independent set of hyperparameters, which we tried to tune by testing different values. They are:

- The dimension of the Embedding layer
- The kind of RNN (GRU or LSTM)
- The number of RNN layers (either one or two), and their respective number of units
- The dropout rate of the RNN layers

After all, even though being very basic, these models were able to produce discrete results. Taking into consideration only the configurations scoring high the `n-grams plagiarism` metric, i.e. considering only the generated samples that were not simply a copy-paste of the original *Divine Comedy*, with each of the three variations we could achieve great scores (up to *0.99*) in terms of `structuredness`, showing that, at least, the network had been able to understand where and how to split one tercet from the other.

As regards the structure of the verses, instead, we could achieve almost optimal results mainly with the *Char-Level* and the *Subword-Level* models. In fact, while these two models were able to generate samples scoring up

[1]If the input sequence is made up of the ordered set of tokens

$$\{i, ..., i + sequencelength - 1\}$$

taken from the *Divine Comedy*, then the output is made up of the ordered set of tokens

$$\{i + 1, ..., i + sequencelength\}$$

to *0.95* in `hendecasyllabicness`[2], the other one could only reach a peak of *0.84*, showing that it was much more difficult for it to understand the syllabication.

However, regarding the rhyming scheme, this architecture failed to get optimal results in all of its variations, reaching a maximal peak of *0.38*, with an average around *0.30*, similar for all of the three models. Thus, given that, we tried to explore a more complex architecture, focusing on a way to enhance this particular aspect.

2.2 Token, Length & Char-Processing RNN

Starting from the previous architecture, we developed this new RNN, represented in the image 2.2, with the idea of feeding the network not only with the word[3] token but also with some other "metadata", namely:

- the *length* of the word (zero in case of punctuation symbols or markers)
- the *char representation* of the word, i.e. a vector of fixed length, equal to the maximal length of a word from the *Divine Comedy*, representing its char tokenization (or a vector of zeros in case of punctuation symbols or markers) which is then processed by a GRU layer[4]

Thus, the final model is made of these three inputs, each of which is separately processed and whose results gets merged into a single vector of a given dimension, which is eventually fed to one or two recurrent layers and a final dense layer with softmax activation, exactly like the previous model.

Hyperparameters and Results

Again, here we had the same set of independent hyperparameters for both the variations of the models. Still, differently from the previous architecture,

[2]We remind that, using the provided code for evaluation, the first canto of the *Divine Comedy* scores around *0.94* in the same metric.

[3]The same idea has been applied to the subwords as well.

[4]This obviously made it useless to try developing a *Char-Level* variation for this architecture, as each char would have been processed separately. Thus, both due to this fact and some training problems encountered for the *Char-Level* models, as well as their not-so-high scores obtained using the previous architecture, we decided not to develop them for future architectures anymore.

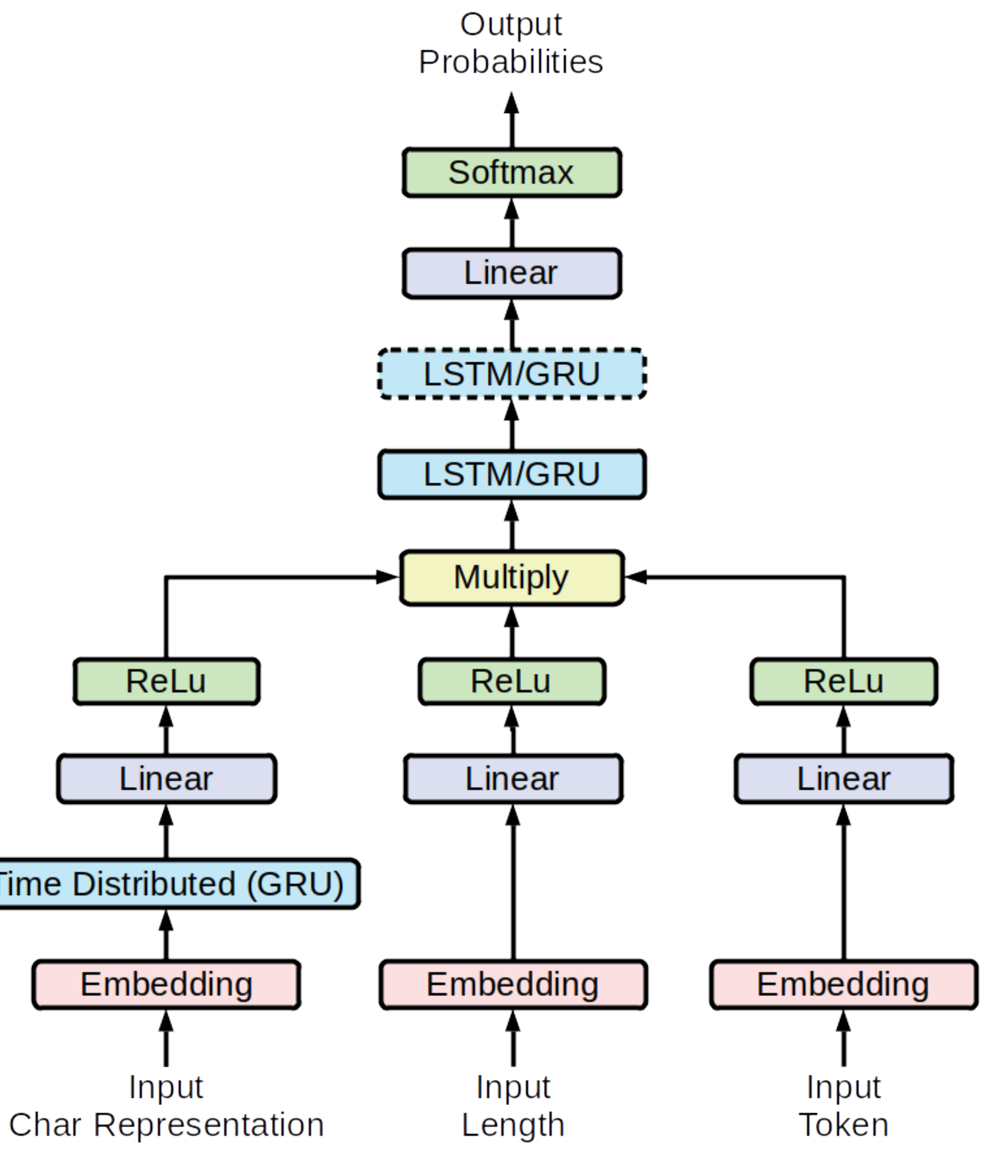

Figure 2.2: Token, Length & Char-Processing RNN Architecture

this one had a larger number of them due to the higher number of inputs, namely:

- The dimensions of the TOKEN EMBEDDING layer, the LENGTH EMBEDDING layer and the CHAR EMBEDDING layer

- The dimension of the latent space prior to the MULTIPLY layer
- The kind of RNN (GRU or LSTM)
- The number of RNN layers (either one or two), and their respective number of units
- The dropout rate of the RNN layers

After all the tests, this architecture showed scores similar to those of the previous architecture related to the *Subword-Level* variation. Instead, a slight increase could be noticed on the scores related to the *Word-Level* variation, which were still a little worse than those of the *Subword-Level*.

Given all of this, we decided not to investigate these basic models anymore and try some more recent kinds of architecture.

2.3 Verse-Space Seq2Seq RNN

One of the possible causes of failure encountered in the previous models could be related to the way the original text was partitioned. In fact, up to the last networks, we developed them to predict one single token given a fixed-length sample of the previous ones, meaning that in the majority of cases we were trying to predict a token in the middle of a verse (thus, having very little information about the rhyming scheme) from a sample starting from a random position inside another verse. Also, as we generally used a `step length` grater than one in order both to reduce the computational cost and to avoid overfitting, some of the markers and the tokens at the end of the verse (i.e. those being more useful to understand both the structure and the rhyming scheme) were not used neither for training nor for validating, going "wasted".

Therefore, with the aim of overcoming this structural problem, we tried to use a *Sequence To Sequence* model for *Neural Machine Translation*, also involving attention mechanisms because of the great complexity of the text. Even though these kind of models are generally used to translate from one natural language to another, we wanted to apply the same idea to the *Divine Comedy*, trying to "translate" a sample of verses into the entire, following verse, and we decided to fix the number of verses, i.e. our new `sequence lenght`, to three, as from the previous three verses (included the one having as a single token the `=tercet=` marker) the network should be able to

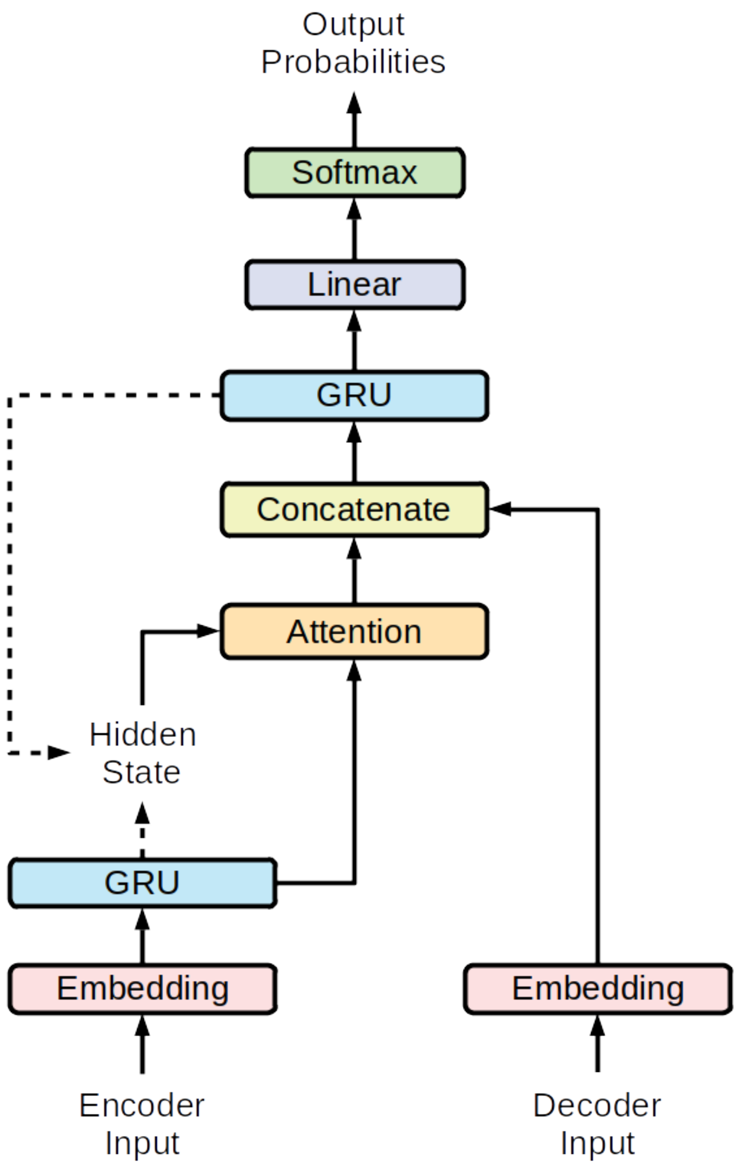

Figure 2.3: Seq2Seq RNN Architecture

understand both the structure and the rhyming scheme, namely it should be able to understand whether the next verse should contain a `=tercet=` marker or not, and if not how to properly end the verse so that it rhymes when necessary.

An important thing to be noticed is that, given that the verse to be predicted is the i_{th} one and the input patch is made up of the verses $i-3$ to $i-1$, initially we used as output sequence the output verse only (or rather, the i_{th} verse). This, however, gave terrible results, in line with the fact that, as well as previous architectures, the network could not understand the inner patterns of the text. Taking that in mind, we decided at first to use as output a sample of the last three verses (from the $i-2_{th}$ to the i_{th}) and, finally, to use all of the four verses as output, because even though the two ways gave almost similar results, in this last case it was easier to perform the prediction/generation phase, as the input sequence had to be entirely passed both to the encoder and to the decoder in order to get as output the last verse only.

Finally, as it can be seen from the picture 2.3, this model is composed of two different, interconnected modules, that are:

- an **Encoder**, made up of an Embedding and a GRU layer, which takes the input sample and returns both the output and the hidden state of the GRU
- a **Decoder**, made up of an Embedding and a GRU layer as well as an Attention layer, which takes the latest decoded token as input, the latest hidden state[5] and the output of the encoder, and returns the hidden state of the GRU and the output processed through a Dense layer

As the output returned from this architecture is the vector of probabilities for a single token, we had to develop a custom training loop that repeatedly called the model using the new output (and the new hidden state as well) as inputs until the *end token* of the output sequence was predicted. This, clearly, led to a more costly training.

Hyperparameters and Results

This time, the hyperparameters were:

- the dimension of the Embedding layer
- the number of units of the GRU layer

[5]During the first iteration, these are the *start token* and the hidden state of the encoder.

- the kind of ATTENTION layer, which could be either:
  - Additive, or *Bahdanau-style* [5]
  - Multiplicative, or *Luong-style* [6]
- the dropout rate for the RNN layers

The main problem of both the variations (*Word-Level* and *Subword-Level*) of this architecture is that, as we said, the training cost was too expensive, with some configuration needing up to twenty minutes per epoch. This, combined with the fact that the convergence was really slow (looking at the loss/accuracy plots, we saw that the trend was clearly increasing in performances without reaching a plateau, but still this increase was linear and very slight), did not allow us to get great results. Indeed, the few configurations that were able to achieve results which were greater than the previous two models, did not score well in the `ngrams plagiarism test`, as they were in fact copying the original verses from the *Divine Comedy*, therefore we decided to abandon the idea of using *RNNs* and exploit, instead, new *state-of-the-art* models for *Neural Machine Translation*.

2.4 Plain Transformer

The *Transformer* [1] is a *state-of-the-art* architecture proposed by Vaswani et al. in 2017 in order to overcome the problems encountered by *RNNs* for *Neural Machine Translation* tasks, namely the poor capacities in long-term memory and attention, and the high computational cost.

As well as standard *Seq2Seq* architectures, the *Transformer* consists of an ENCODER and a DECODER, as described in figure 2.4, both having a number of layers that can be changed in order to have a greater or lower complexity. Still, differently from previous models, this new one cannot be classified as a recurrent network but it is, instead, a standard feed-forward one, reason why both the ENCODER's and the DECODER's input tokens must be processed not only with a classical EMBEDDING layer but also with a so-called POSITIONAL ENCODING, so that the information about the order of the tokens in the sequence is not forgotten even without the presence of recurrent edges. Finally, these processed inputs are passed through the series of layers, which are made up of two different sub-modules: a MULTI-HEAD ATTENTION (two in the case of the DECODER, as the first one is responsible of processing the

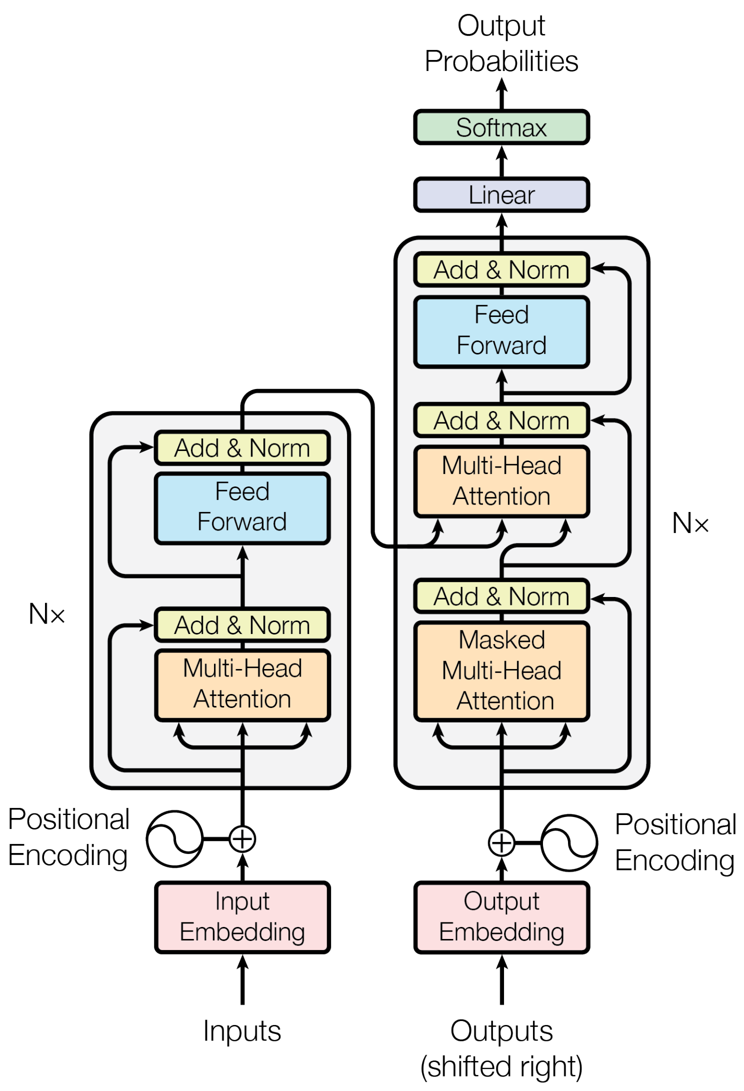

Figure 2.4: Transformer Architecture

decoder input, while the second one is responsible of processing the merged vectors of features extracted both from the encoder input and the decoder input), and a subsequent FEED-FORWARD block. Everything is eventually passed through a DENSE layer, with softmax activation, to get the output probabilities of each possible token.

The reason why we decided to try this architecture is due to its great ability to generalize in different *Natural Language Processing* tasks as well as its remarkable training speed (being not an *RNN*, it does not suffer the heaviness of backpropagation-through-time). Furthermore, as we did for the previous models, we decided to start with a simpler version, thus we went back to the standard way of splitting the original text, i.e. with samples of fixed input length and fixed output length, where the output is shifted by one token with respect to the input.

Hyperparameters and Results

Though being quite a complex model, the *Transformer* does not have a high number of hyperparameters, which are:

- the number of layers for the `Encoder` and the `Decoder`
- the number of heads for the Multi-Head Attention sub-module
- the dimension of all the sub-layers in the model, as well as the Embedding layers, known as `d_model`
- the inner Feed-Forward layers dimension, known as `dff`
- the dropout rate

Overall, this architecture showed a great ability to understand the general structure of the text, as well as our first two models, but in the end nothing more than that could be achieved. In fact, we could notice a slight increase in the metric for the rhyming scheme, reaching a peak of *0.41* for the *Word-Level* model and *0.44* for the *Subword-Level*, while the score for `hendecasyllabicness` remained stable for the *Word-Level* and slightly decreased down to a peak of *0.90* for the *Subword-Level* model.

2.5 Verse-Space Transformer

Given that the *Plain Transformer* model gave slightly better results in terms of rhyming scheme, we tried to use the same architecture to perform a more straight-forward *Sequence To Sequence* approach, like we had done for the third model. Thus, we split again the *Divine Comedy* in single verses and used as input a patch made up of three consecutive verses and, as output, not just the fourth one but the entire patch made up of the all four, exactly as we did for the *Seq2Seq RNN* architecture[6].

Hyperparameters and Results

Here, the hyperparameters we had to tune were the same ones of the previous model, i.e. the number of layers, the number of heads, the `d_model`, the `dff` and the dropout rate.

Way more than previous architectures, this one seemed to be very sensible to the variation of the hyperparameters, producing samples having a wider range of scores in the employed metrics.

Also, quite interestingly, tinier models seemed to be more capable of generating better samples. We interpreted this fact as a proof that, for generative tasks, sometimes it could be better to have a small network which is unable to reproduce the training samples (e.g. in our case, the Neural Network is unable to reproduce the exact *Divine Comedy*) but, instead, it is able to generate samples that, far from being equal, are similar to the training samples. Instead, this is something that turns out to be almost useless for *Neural Machine Translation* tasks, in which we may need a bigger network with a great ability to reproduce original samples (e.g. to correctly translate a sentence from Italian to English), nevertheless failing at finding more "particular" patterns and, therefore, which would not be able to generate good samples when using higher temperatures during the generative phase.

In the end, even though the *Word-Level* model could not obtain scores that overcame the previous ones (instead, the scores related to the rhyming scheme were lower with respect to almost all of them), maybe due to the fact that we could not find a correct configuration of hyperparameters; the *Subword-Level* model definitely exceeded the bounds set by previous models in terms of rhyming scheme, reaching peaks of *0.89* for that score. Furthermore, as regards the overall structure of the tercets, the score was optimal for almost all of the tested configurations, while for the `hendecasyllabicness` we could reach a peak of *0.90* only. This, though being slightly lower with respect to that of the first two models, can be still considered a good result; in fact, we should take in mind that the provided code used for evaluation, when tested on the first canto of the *Divine Comedy*, due to the complex

[6]Actually, we also tried some configurations in which we used either four or six verses as input, and five or seven as output respectively, but we could not get greater results, probably due to the greater length of the sequences.

phenomena of synalephe and dialepha widely used by Dante, did not gave a full score for that particular metric, but returned instead a value which was around *0.94*.

2.6 Transformer GAN

Even though with our last model we could be able to achieve good results in terms of metric scores, we wanted to develop at least one adversarial model [7] in order to try getting rid of the `categorical crossentropy` loss function, which could not always be a good choice for the particular task we were intended to solve.

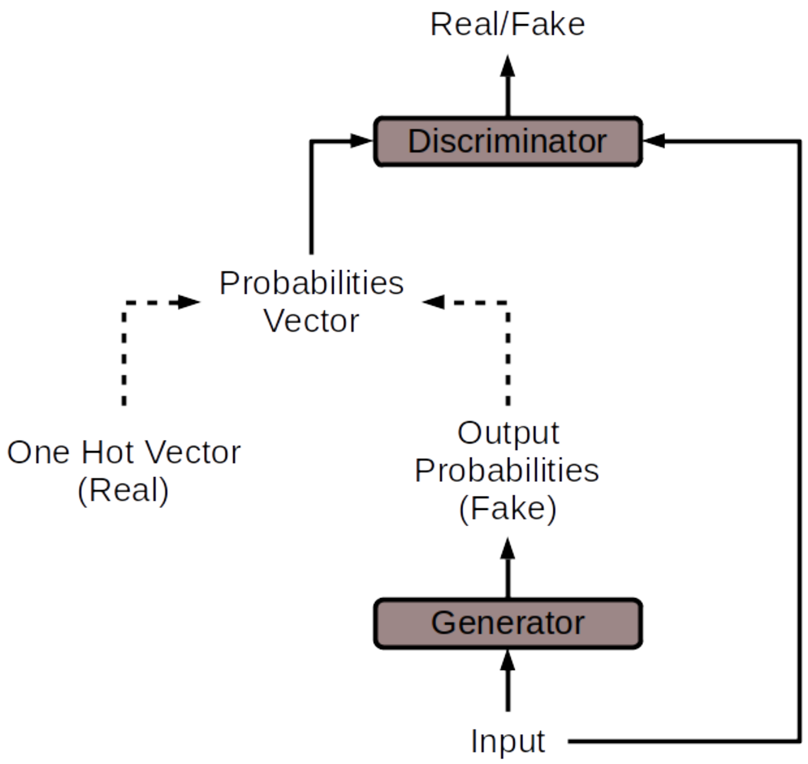

Figure 2.5: Overall GAN Architecture

Overall, our architecture, as depicted in figure 2.5, is composed of a GENERATOR and a DISCRIMINATOR. The former takes as input a fixed-length sample of the *Divine Comedy*, as well as the standard *RNN* models and the first version of the *Transformer*, while returning as output a vector of probabilities for the next sample. Instead, the latter is a *Siamese Network* taking as input both that sample and either the output probabilities vector of the GENERATOR or the one-hot encoded vector representing the actual token. A closer look of the two modules is presented in the figure 2.6.

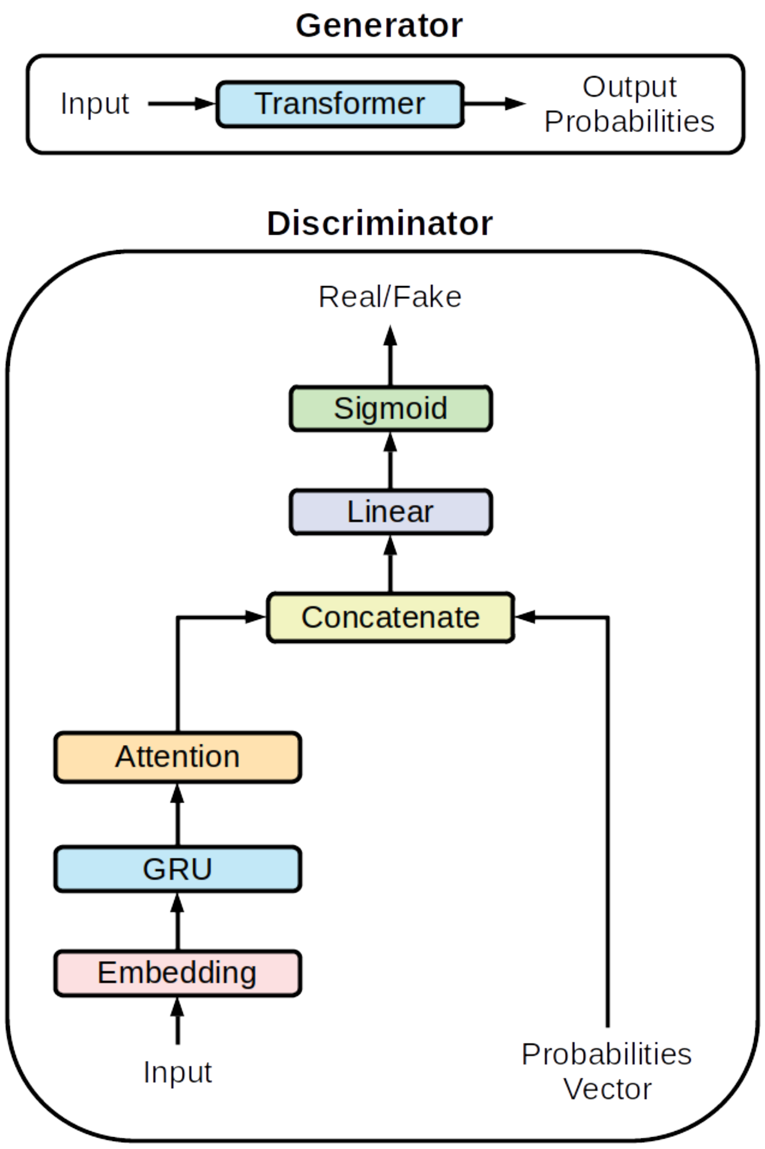

Figure 2.6: Generator & Discriminator of the GAN Architecture

The reason why we had to switch back to the previous way of splitting the text instead of going on with the verse-level division is that, during the training, it was not possible to generate an entire verse (made up of multiple tokens) at once without having to apply an *hardmax* activation to the generated vectors. Thus, being the *hardmax* activation non-linear, this would have made impossible to get the gradients related to the weights of the net and, subsequently, to correctly perform the backpropagation step.

Hyperparameters and Results

Both the `Generator` and the `Siamese Discriminator` had their own hyperparameters to be set. In particular, those of the `Generator` are:

- the number of layers of the *Transformer*
- the number of heads of the *Transformer*
- the dimensions of the latent spaces (`d_model` and `dff`)
- the dropout

while, those of the `Discriminator` are:

- the Embedding dimension
- the number of GRU units
- the Attention kind
- the GRU dropout

Even though we tried some different kind of settings, we could never reach the *Nash Equilibrium* between the two modules of the *GAN*[7], which prevented us from having any fit sample at all.

Given that finding an equilibrium during an adversarial train is a well-known complex problem in the field of *Deep Learning*, and that the methods to overcome it require a deep knowledge of many *state-of-the-art* techniques, we decided to stop our investigations here, also given that the results of our previous model were good enough in our opinion. Anyhow, we shall at least briefly present some of these techniques in the next chapter.

[7]Generally, the `Generator` seemed to be more powerful than the `Discriminator`, even for those cases in which we largely reduced its size.

Chapter 3

Results and Future Works

Overall, our selected model turned out to be reasonably able to automatically discover and learn the hidden patterns of the *Divine Comedy*, as explicitly asked in the requirements for the project, and to replicate them as well. In fact, no external hint about syllabication and rhymes has been given to the network, apart from the fact that we reasonably had to split the input and output samples so that they were made of full verses instead of random patches starting and ending in the middle of a verse.

Given that, we feel very proud of the results achieved, but still we know that, with more time and experience, we may have been able to outperform even the result we could reach. For this reason, the last part of our report will be dedicated to the presentation of some newer and more sophisticated techniques that may be used in some future work to raise the bar in this task of generating the *Divine Comedy.*

Bert

BERT, standing for *Bidirectional Encoder Representations from Transformers*, is a recently proposed architecture "designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers" [8]. However, even though *BERT* was initially intended to be a pre-trained network to be fine-tuned in order to solve many *Natural Language Processing* tasks such as text classification, reading comprehension, and so on, its full application to *Neural Machine Translation* had not been investigated until the last months by Zhu et al. [9].

Clearly, we do not know how much *BERT* could be effective for the purposes of our task, but still we think it may need some further investigations, and the reason we did not do that on our own is because the architecture is too new to be incorporated in some library and, as well, it looks very heavy to train, which would most probably require ad-hoc equipment that we do not possess.

Long Term Dependencies

Though *Seq2Seq* models seemed to perform well in discovering patterns, there are some problems, strictly related to the semantics of a text and not to its syntax, which are somehow intrinsic and cannot be solved with standard techniques. In fact, as the input of our model was made up of the last three generated verses, the network could not have any insight on what had happened before, meaning that there is no possibility at all that the overall generated text would be following a story, with recurrent characters and/or well-constructed dialogues.[1] Also, taking our case as example, the network will fail at correctly predict the `=endofcanto=` marker, leading the conclusion of a canto to be quite ruled by randomness, in the sense that, if after each tercet the probability of predicting the `=endofcanto=` marker is, say, *0.02*, then on average a canto will last after *50* verses, but the process is completely stochastic.

One way to solve this problem could be by using *Stateful RNNs*, which maintain an inner representation of what has been processed so far by the network and, therefore, there could be a possibility that in context vectors is kept some information about how much "time" ago the canto started, or the quote mark has been opened. Still, as we could experiment, *RNNs* are not the most suitable model for this task, as they are too heavy to train and they are not powerful enough to discover the patterns.

A practical solution could be to mix both of these ideas. Essentially, we could use a *Stateful RNN* (or something similar, like *BERT*), to extract a context vector from all the text generated so far, i.e. some latent representation of a "summary" of the text, and use this context vector together with the standard input, i.e. the tokenization of last three generated verses, as input for the *Transformer* model.

[1]This can be noticed, for example, from the fact that sometimes our network fails to close quote marks, specifically when the spoken sentence is longer than three verses, because there is no way the network would know it had opened them at all.

Stable GANs

As we could see during our experiments, finding the equilibrium point of a *GAN* can be quite challenging, but it is something that must be overcome in case we would like to explore some kind of adversarial technique for our task.

As this is a widely known issue for adversarial models, which has been recently addressed and studied, some interesting ideas have been proposed during the last few years. One of the most famous work related to that, published in 2017 by Arjovsky et al., presents the so-called *Wasserstein GAN*, or *WGAN*, a special kind of *GAN* that tries to improve the stability of learning with the help of some kind of probabilistic regularization, while providing "meaningful learning curves useful for debugging and hyperparameter searches" [10].

Finally, we also wanted to pose the attention on some works even more strictly related to the field of *Natural Language Processing*, in which adversarial approaches are not so widely used yet. The work we are referring to, published in 2019 by Haidar and Rezagholizadeh, exploits both *Autoencoders* and *GANs* to build a so-called *TextKD-GAN*, namely a *GAN* for text generation which uses an innovative approach known as *Knowledge Distillation* [11].

Different Networks for Different Purposes

In case the previous approaches would not be enough, a final improvement could be made by introducing some kind of domain knowledge and develop different modules taking care of different aspects of the generation.

For example, an approach used by Benhardt et al. in 2018 to achieve *state-of-the-art* results in the generation of Shakespearian sonnets, is to use two neural networks: one aimed at finding the last syllable so that the rhyming scheme would be preserved, and the other one aimed at filling the remaining part of the verse so that it could follow the structure of a iambic pentameter[2] [12].

Moreover, another very useful and important module that could be devel-

[2]Actually, their work was even more complex, and took advantage of the *POS-tagging* of the tokens and some instilled constraints over them so that not only the verse would have been made of ten syllables, but also it would be meaningful from a semantic point of view.

oped for the purpose is a syllabication module, correctly able to understand both synalephe and dialepha. With that, it could be possible to correctly split the input text in eleven tokens and, therefore, the hendecasyllables would most probably be perfect all the time because, if the network is powerful enough, it should not be difficult for it to understand that we are clearly imposing the generation of a newline token after other eleven tokens.

In the end, what we could say about this approach is that, even though it can be regarded somehow as "cheating", as the network receives some big hints about the text structures and patterns without having to understand them on its own, this is actually how our brain works, i.e. with many different areas (modules), each one taking care of a different aspect of the computation, and finally merging all the results in a single outcome.

Bibliography

[1] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[2] Andrej Karpathy. "The unreasonable effectiveness of recurrent neural networks". In: *Andrej Karpathy blog* 21 (2015), p. 23.

[3] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[4] Junyoung Chung et al. "Empirical evaluation of gated recurrent neural networks on sequence modeling". In: *arXiv preprint arXiv:1412.3555* (2014).

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate". In: *arXiv preprint arXiv:1409.0473* (2014).

[6] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. "Effective approaches to attention-based neural machine translation". In: *arXiv preprint arXiv:1508.04025* (2015).

[7] Ian Goodfellow et al. "Generative adversarial nets". In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[8] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[9] Jinhua Zhu et al. "Incorporating bert into neural machine translation". In: *arXiv preprint arXiv:2002.06823* (2020).

[10] Martin Arjovsky, Soumith Chintala, and Léon Bottou. "Wasserstein gan". In: *arXiv preprint arXiv:1701.07875* (2017).

[11] Md Akmal Haidar and Mehdi Rezagholizadeh. "Textkd-gan: Text generation using knowledge distillation and generative adversarial networks". In: *Canadian Conference on Artificial Intelligence*. Springer. 2019, pp. 107–118.

[12] John Benhardt et al. "Shall i compare thee to a machine-written sonnet? An approach to algorithmic sonnet generation". In: *arXiv preprint arXiv:1811.05067* (2018).