The Deep Comedy

Andrea Policarpi - 0000950326
andrea.policarpi@studio.unibo.it

Nicola Amoriello - 0000952269
nicola.amoriello@studio.unibo.it

January 2021

Contents

Chapter 1

Introduction

Aim of this project was to build a Neural Network able to generate text characterized by the style of Dante Alighieri's "Divina Commedia", a poetry work characterized by its division in tercets, its hendecasyllable verses and its so called "terza rima" scheme (ABA BCB CDC and so on).

In order to do so, we used and preprocessed an online version of the Divina Commedia in order to create a train and a validation dataset, that we used to train several models with different architectures. In the end, the two best-performing models that we found are a Recurrent neural network with custom loss and a Transformer. In the following chapters of this report we will explain the work we did in terms of data preprocessing and model construction, and analyze our solutions's performances by means of some common metrics. We will also provide some examples of generated canticae.

Regards to the practical implementation of this project, we mainly used the Tensorflow and Keras[1] libraries to develop the models, and the Google Colab environment to train them, in order to exploit the free-provided GPU.

Chapter 2

Data Preprocessing

2.1 Preprocessing steps

In order to build our datasets we downloaded an online version of the Divina Commedia and preprocessed it. As parts of the proprocessing pipeline, we did the following:

1. We removed punctuation, accents and uncommon characters (such as “ë”, ”ö”,..);
2. We removed the blank rows, and introduced the special tag “=end_terzine=” to separate the terzines;
3. Tokenization step: We split the text in syllables, by means of the syllabify_block function, stored each syllabified verse into a list, and inserted the tags “-start-“ and “-end-“ in order to delimitate it.

So, at the end of this preprocessing pipeline, a tercet like this:

```
Nel mezzo del cammin di nostra vita
mi ritrovai per una selva oscura,
ché la diritta via era smarrita.
```

Figure 2.1: An example of terzine yet to be preprocessed.

Would be converted into:

```
['=end_terzine=']
[' -start- ', 'nel', ' ', 'mez', 'zo', ' ', 'del', ' ', 'cam', 'min', ' ', 'di', ' ', 'no', 'stra', ' ', 'vi', 'ta', ' -end- ']
[' -start- ', 'mi', ' ', 'ri', 'tro', 'vai', ' ', 'per', ' ', 'u', 'na', ' ', 'sel', 'va', ' ', 'o', 'scu', 'ra', ' -end- ']
[' -start- ', 'che', ' ', 'la', ' ', 'di', 'rit', 'ta', ' ', 'via', ' ', 'e', 'ra', ' ', 'smar', 'ri', 'ta', ' -end- ']
['=end_terzine=']
```

Figure 2.2: The same terzine, after preprocessing steps.

This results in the standard proprocessing that we implemented in order to create our datasets. In addition to this, we also implemented two variations:

1. We wanted to compare the performances obtained by a function-syllabified dataset with the ones that an almost-perfect syllabification could provide. In order to do so, we used the "Commedia_syllnew" syllabified version of the Divina Commedia from professor Asperti's GitHub repository[3]. This syllabified version takes into account some poetry quirks (such as synalepha) that our syllabification function otherwise struggles to keep track of;

```
1 |Nel |mez|zo |del |cam|min |di |no|stra |vi|ta
2 |mi |ri|tro|vai |per |u|na |sel|va o|scu|ra,
3 |ché |la |di|rit|ta |via |e|ra |smar|ri|ta.
```

Figure 2.3: A tercet from *asperti/Dante* GitHub repository.

2. In the forced-rhyme RNN model (which will be described in the following chapters), we made sure that the last element of each verse list contains the rhyming part of the verse. To do so, we added in the preprocessing pipeline a function that takes the last two syllables of each verse and returns their appropriate partition.

```
['=end_terzine=']
['ahi', ' ', 'quan', 'to', ' ', 'a', ' ', 'dir', ' ', 'qual', ' ', 'e', 'ra', ' ', 'e', ' ', 'co', 'sa', ' ', 'd', 'ura']
['e', 'sta', ' ', 'sel', 'va', ' ', 'sel', 'vag', 'gia', ' ', 'e', ' ', 'a', 'spra', ' ', 'e', ' ', 'f', 'orte']
['che', ' ', 'nel', ' ', 'pen', 'sier', ' ', 'ri', 'no', 'va', ' ', 'la', ' ', 'pa', 'ura']
['=end_terzine=']
```

Figure 2.4: Sample of a tercet with last-element rhyme split.

At last, we merged all the verses concluded the tokenization step by creating a dictionary that assigned to each syllable an integer and mapped our datasets with it. Here is an example of what the final data looks like:

```
[   0    3 2034    1 1826 4053    1  687    1  232 1860    1  696    1
 2085 3250    1 3890 3411    2    0    3 1827    1 2568 3706 3849    1
 2217    1 3782 1949    1 2927 3843 2880 2451    2    0    3  404    1
 1484    1  696 2618 3411    1 3912    1  867 2451    1 3021 2568 3411
    2    0    4    0    3   24    1 2409 3575    1  755    1 2407    1
  867 2489    1  503 2728    1  842 2451    2    0    3  867 3138    1
 2927 3821    1 2927 3848 1100    1    5 3108    1  958 3459    2    0
    3]
```

Figure 2.5: The first 100 tokens of the Divina Commedia.

Chapter 3

Initial (unsuccessful) models

In this chapter we will briefly list all the models which we tried to approach the Deep Comedy problem with, and that at the end resulted in failures due to their poor performances in terms of generated text and metrics.

3.1 Char-level RNN

This was our first, baseline model. We built a RNN model that takes as input samples of the non-syllabified Comedy with a fixed length (the hyperparameter SEQ_LENGTH) and as output the input sample shifted by one element, We did this in order to let the network learn the inner patterns between each verses and terzines. As for the SEQ_LENGTH value, we made sure that it would cover at least 4 verses.

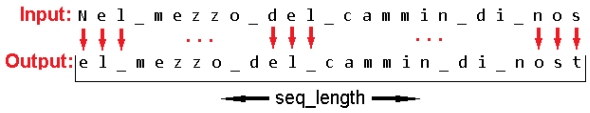


Figure 3.1: An example of input-output pair for Char-level RNN.

As for the architecture, we tried both a single-layer LSTM and a double-layer LSTM, with/without dropout layers and final Dense layers of appropriate size in order to introduce a funneling of information. We prefered the LSTM layers over the standard RNN ones, due to the fact that the latter suffer from the "vanishing gradient" problem. As always, the last layer is a Dense one, with the Softmax activation function, in order for the model to predict the class probabilities for each following character.

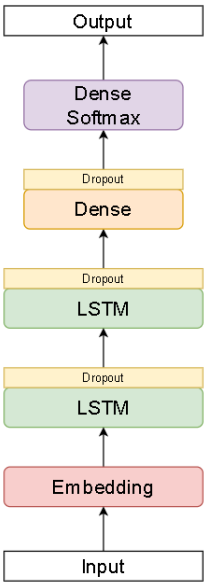


Figure 3.2: General structure of the RNN model.

3.2 Syllable-level RNN

Since the char-based RNN models seemed unable to learn the hendecasyllable pattern, we took the previous architectures and trained them with the syllabified datasets. In this way the model managed to generate text that, while completely lacking in the rhyme part and still not structured enough to apply our metrics, brought noticeable improvements on the hendecasyllable and terzine parts. These moderate improvements encouraged us to follow the route of the custom-loss RNN described in chapter 4, and made us decide to work only with syllables in the following models.

3.3 RNN as part of a GAN (adversarial approach)

We also tried to adopt an adversarial approach to the problem, by building a GAN composed by a Generator (the previous syllable-based RNN) and a Discriminator.

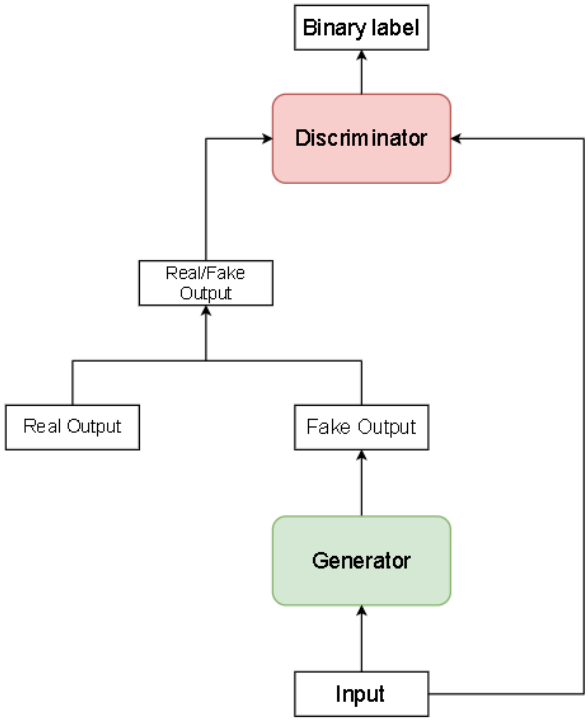


Figure 3.3: GAN structure.

The RNN Generator takes as input a fixed-length sample of the Comedy, and outputs the class probability vector for the next sample. Then, either the real(the one-hot encoded Y associated to the X input) or the fake output is fed to the discriminator (with a corresponding label), along with the original input. Aim of the discriminator is to distinguish between the real and the fake output by means of a binary classification task. Then, the error of the discriminator is backpropagated and used to train the Generator in order for it to better fool the Discriminator (and thus to generate text more similar to the Divine Comedy). Unfortunately, even given this complex approach, the results were completely unsatisfying, due to the fact that we couldn't manage to train the Discriminator properly. So, we decided to leave the adversarial approach in favour of the models described in the next two chapters.

Chapter 4

RNN with Custom Loss

Given the promising results of the plain syllable-based RNN, we decided to introduce a custom loss divided in four different components in order to boost the results in terms of terzine structuredness, hendecasyllables, and rhymes. In the following sections we will describe the final network architecture, our choices of hyperparameters, the adopted custom losses, and the results in terms of the provided metrics.

4.1 Hyperparameters & Architecture

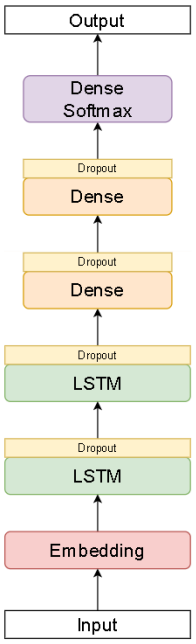
During the design process, we had to cope with the following architecture hyperparameters:

1. The embedding dimension;
2. The type and number of LSTM and Dense layers;
3. The number of neurons on each layer;
4. The dropout rate.

Also, as part of the design process, we had to choose:

1. The sequence length of the input;
2. The train/validation split;
3. The optimizer and the learning rate;
4. The batch size and the number of training of epochs;
5. The temperature factor, at generation step.

Taking inspiration from the state of the art, and after some experiments, we decided to stack two LSTM layers followed by two Dense layers with a funneling function, as depicted in figure. We also set on each layer a dropout rate of 0.2 in order to cope with overfitting.



Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(1, None, 512)	2099200
lstm_2 (LSTM)	(1, None, 1024)	6295552
dropout_4 (Dropout)	(1, None, 1024)	0
lstm_3 (LSTM)	(1, None, 1024)	8392704
dropout_5 (Dropout)	(1, None, 1024)	0
dense_3 (Dense)	(1, None, 512)	524800
dropout_6 (Dropout)	(1, None, 512)	0
dense_4 (Dense)	(1, None, 256)	131328
dropout_7 (Dropout)	(1, None, 256)	0
dense_5 (Dense)	(1, None, 4100)	1053700

Total params: 18,497,284
Trainable params: 18,497,284
Non-trainable params: 0

Figure 4.1: Model architecture.

As for the sequences length, we initially set it long enough to cover at least a terzine and the first verse of the following one (4 verses in total) in order to cover the full rhyming scheme (ABA B), but after some trial-and-error we found that increasing it would be beneficial to the generation performances. In the end, we set `SEQ_LENGTH = 300`. Still by state-of-the-art inspiration, we chose a batch size of 64, so the dataset was divided into 14 batches. For the train/val split, we divided it into 13 training batches and 1 validation batch, in order to train the model with the most amount of data possible.

We trained our models for 50 epochs, as we found it was a good compromise between the lengthened training time required by the custom losses and the model overfitting.

At the text generation step, we experimented with many values of temperature, finidng that the best results are achieved into a $0.5 < t < 0.7$ range.

4.1.1 Custom Loss

The main difference in performances that we achieved with respect to our previous models is due to the implementation of a custom loss composed by four parts:the standard Categorical Crossentropy plus one for each subproblem (hen-

decasyllable, terzine structuredness and rhymes) Each of these three loss components is computed by considering the 300 output syllables corresponding to each train input vector. Then, from this output it is reconstructed the generated text composed by verses and terzines. If no well-formed terzines are generated, there is an heavy penalization (an hyperparameter that we set to "100" after some experiments), otherwise the loss is increased by a combination of the "well-formed" parts with respect to the "overall" parts considered by each loss fragment. Several loss combinations were explored, such as considering the percentage, the difference, the weighting by an arbitrary parameter, and so on. In the end we decided to keep it simple by considering the fraction or the difference only.

```
#############################################################################################################
def loss_terzine(verses):
  verses = [list(g) for k, g in groupby(verses, key=lambda x: x != "\n") if k]
  num_terzine = 0
  num_well_formed = 0

  for i in range(len(verses)-4):
    #print(verses[i])
    if verses[i] == ['=end_terzine=']:
      num_terzine += 1
      if verses[i+1] != ['=end_terzine=']
      and verses[i+2] != ['=end_terzine='] and verses[i+3] != ['=end_terzine='] and verses[i+4] == ['=end_terzine=']:
        num_well_formed += 1

  if num_terzine > 0:
    return (num_terzine - num_well_formed)/num_terzine
  else:
    return 100
#############################################################################################################
```

Figure 4.2: Terzine structuredness component of the custom loss.

```
#########################################################################################################################
def loss_syllable(verses):
  if '\n' not in verses:
    return 100
  verses = [list(g) for k, g in groupby(verses, key=lambda x: x != "\n") if k]
  err = 0.0
  num_verses = 0

  for el in verses:
    if ' -start- ' in el and ' -end- ' in el:
      counter = len([  syl  for syl in el if syl != ' ' and syl != ' -start- ' and syl != ' -end- '])
      err = err + abs( 11 - counter)
      num_verses += 1
  if num_verses > 0 :
    return err/num_verses
  else:
    return 100
#########################################################################################################################
```

Figure 4.3: Hendecasillabness component of the custom loss.

```
            def loss_rhymes(verses):
              verses = [list(g) for k, g in groupby(verses, key=lambda x: x != "\n") if k]
              verses = verses[:-1]
              num_checked_rhymes = 0
              num_good_rhymes =
#-------------------------------------------#   #-------------------------------------------#
  for i in range(len(verses)-3):                    for i in range(len(verses)-5):
    if verses[i] == ['=end_terzine=']:                if verses[i] == ['=end_terzine=']:
      a = verses[i+1]                                   a = verses[i+2]
      b = verses[i+3]                                   b = verses[i+5]
      if len(a)==1 or len(b)==1:                        if len(a)==1 or len(b)==1:
        pass                                              pass
      else:                                             else:
        if a[-2] == b[-2]:                                if a[-2] == b[-2]:
          num_checked_rhymes += 1                           num_checked_rhymes += 1
          num_good_rhymes += 1                              num_good_rhymes += 1
        else:                                             else:
          num_checked_rhymes += 1                           num_checked_rhymes += 1
#-------------------------------------------#   #-------------------------------------------#
               if num_checked_rhymes > 0:
                 return (num_checked_rhymes - num_good_rhymes)/num_checked_rhymes
               else:
                 return 10
```

Figure 4.4: Rhymeness component of the custom loss.

By the implementation of the custom loss, we managed to achieve excellent

results for what regards the structuredness and the hendecasyllabness, but a very small improvement in the rhymeness evaluation.

4.1.2 Optimizer & Learning Rate

As suggested by the state-of-the-art, we used the Adam optimizer with its default parameters (beta1=0.9, beta2=0.999, epsilon=1e-07) to compile the model. After some experiments with the learning rate, we decided to keep the default one (learning rate=0.001) since its variation didn't cause notable differences in performance.

4.2 Results

At the end, we managed to achieve very good results for what concerns the structuredness and the hendecasyllabness, but poor results for the rhymeness.

By using the homemade syllabification module, we obtained:

1. Number of putative terzine: 46
2. Number of well formed terzine: 45
3. Average structuredness: 0.9639
4. Average hendecasyllabicness: 0.9385
5. Average rhymeness: 0.2668
6. Ngrams_plagiarism : 0.9814

By instead using the prof.Asperti's dataset, these results slightly improve:

1. Number of putative terzine: 46
2. Number of well formed terzine: 45
3. Average structuredness: 0.9784
4. Average hendecasyllabicness: 0.9584
5. Average rhymeness: 0.2575
6. Ngrams_plagiarism : 0.9943

In the end, this Custom Loss model failed in the rhymeness part, and thus we decided to explore a different approach (the Transformer).

```
Seed:
=end_terzine=
 -start- nel mezzo del cammin di nostra vita -end-
 -start- mi ritrovai per una selva oscura -end-
 -start- chè la diritta via era smarrita -end-
```

```
Generated:
=end_terzine=
 -start- non quando de la mia riversi mali -end-
 -start- di quella mia di come mi divea -end-
 -start- per la prima del nome e di sperato -end-
=end_terzine=
 -start- e come come de la sua dicene -end-
 -start- di me per la vostra fa per vederse -end-
 -start- per lo veder di sè non si contenta -end-
=end_terzine=
 -start- ma se di mi di me così di balte -end-
 -start- e come tra la fina di redita -end-
 -start- che non si volse per per la parite -end-
=end_terzine=
 -start- a quel quando la luce con la nova -end-
 -start- del magno son di quel di lor si dice -end-
 -start- e ca là non si torna rana vaso -end-
=end_terzine=
 -start- io come miei per la sua vita grava -end-
 -start- e de la nostra luce di ma magno -end-
 -start- se che se a la terra rice cira -end-
=end_terzine=
```

Figure 4.5: A sample of text generated by the model.

4.3 Using the model in a way that ensures rhymes by "forcing" them in the generation step

Since the model achieved excellent performances in everything except for the rhyme part, we wondered what would happen if we "hard-inserted" the rhymes during the text generation step, while letting the model generate everything else.

The idea behind this consideration is given by the fact that, for rhyming verses, we exactly know what the last part of each verse should be: since we work with hendecasyllables, in order to form a rhyme we are bound to take the second-to-last syllable from the last vowel onward and the last syllable as the final part of the second verse in order to rhyme to the first.

For this purpose, we did a special preprocessing: after the syllabification step, we inverted the syllable order on each verse list, so that the verse order remains unchanged while verses are read in reverse by the model.

In this way, we made sure that the first (and only the first) syllable after the "-start-" token would be the one involved in the rhyme scheme. We made sure that only a single syllable-like element is involved for each verse, due to the previous syllable manipulation we explained in chapter 2.

Then, we trained the network as usual: in this way the RNN model learnt to biuld verses backwards, from the last to the first syllable.

```
['=end_terzine=', '\n']
[' -start- ', 'ita', 'v', ' ', 'stra', 'no', ' ', 'di', ' ', 'min', 'cam', ' ', 'del', ' ', 'zo', 'mez', ' ', 'nel', ' -end- ', '\n']
[' -start- ', 'ura', 'sc', 'va o', 'sel', ' ', 'na', 'u', ' ', 'per', ' ', 'vai', 'tro', 'ri', ' ', 'mi', ' -end- ', '\n']
[' -start- ', 'ita', 'r', 'smar', ' ', 'ra', 'e', ' ', 'via', ' ', 'ta', 'rit', 'di', ' ', 'la', ' ', 'chè', ' -end- ', '\n']
['=end_terzine=', '\n']
```

Figure 4.6: A terzine after the "reversing" step.

Finally, at the generation step, we generated one verse at the time, giving as input the three previous generated verses and the "right" rhyme syllable according to the corresponding rhyme verse(except for the second verse of each terzine, in which the rhyme changes: in this case we just gave as input the three previous verses). This procedure assures that, as long as the model produces well-formed terzines, the rhyme scheme is surely respected.

```
=end_terzine=
 -start- itav strano di mincam del zomez nel -end-
 -start- urascva osel nau per vaitrori mi -end-
 -start- ita
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
 -end- grato di se che perchè non smarrita -start-
```

Figure 4.7: A generation step: input given and output generated (already reversed).

So, at the end of the day, we achieved these results:

1. Number of putative terzine: 51
2. Number of well formed terzine: 51
3. Average structuredness: 1.0
4. Average hendecasyllabicness: 0.9643
5. Average rhymeness: 1.0
6. Ngrams_plagiarism : 0.9965

We are aware of the fact that this method of text generation would be considered as "cheating", since the model didn't really learn the structure of the Terza Rima rhyming scheme. But given the results, it was a fun and interesting experiment to try nonethless. Here is a sample of generated text:

```
io credi vinta quanto de la mura
e da tanto là è che tatal porto
quando perchè lor sotta non col sura

docente la gentier ricenti corto
e già le compienza in chi si poco
ciascun tutdin sì che non che a porto

la genta la cui tu va che l mia moco
e per quel che di sua corno che pende
e io che mi fagnomendo si poco

io sescia di per che da mi riprende
come viso quel che di suoi cammura
e assai si luce se si soccende

ma vite e e io così fummo cura
e quel vo te li sovra più mi spetta
quando sì che si fatta de la cura

e colui di sè mentra per la getta
che mange se che riccia ti prilato
se si specchio mai sì de la vernetta
```

Figure 4.8: A sample of text generated.

Chapter 5

Transformer

An alternative model vastly employed in many text-related tasks with excellent results is the *Transformer*, proposed for the first time in the *"Attention Is All You Need"*[7] in 2017. This kind of architecture allows us to achieve enormous results in tasks that require attention, long-term memory and high computational costs.

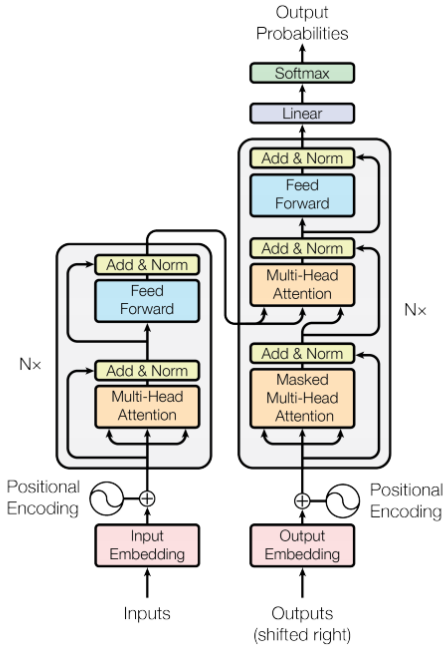


Figure 5.1: The Transformer - model architecture.

Like recurrent neural networks, *Transformers* are designed to handle sequential data, such as natural language, for tasks such as translation, text summarization and text-generation. However, unlike RNNs, it is not required for the sequential data to be processed in order, in fact the Transformer can be classified as a standard feed-forward network and, for this reason, we need to process the input (both of the encoder and the decoder) with a Positional Encoding to give the model some information about the relative position of the words in the terzine.
The *Transformer* is an encoder-decoder architecture. The encoder consists of a set of encoding layers that process the input iteratively one layer after another and the decoder consists of a set of decoding layers that do the same thing to the output of the encoder.
Finally, these processed inputs are passed through a block of layers 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 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.

Given the excellent results of the *Transformer* model in many state-of-the-art applications, we decided to try it on our problem in order to achieve better performances with respect to our previous models. In particular, we hoped that its intrinsic *attention mechanisms* could make the network able to better learn the *Terza Rima* rhyming scheme and replicate it during the text generation process.

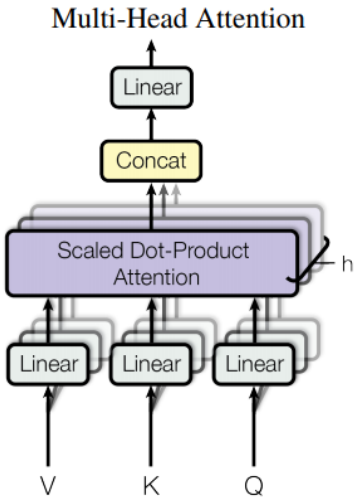


Figure 5.2: Multi-Head Attention block, core of the transformer's attention mechanism.

During the design process, we had to cope with the following architecture hyperparameters:

1. The number of layers;
2. The number of attention heads;
3. The dimension of the model layers;
4. The dimension of the feed-forward networks;
5. The dropout rate;

Also, as part of the design process, we had to choose:

1. The sequence length of the input;
2. The train/validation split;
3. The optimizer and the learning rate;
4. The batch size and the number of training of epochs;
5. The temperature factor, at generation step.

We decided to use 4 layer for the Encoder and the decoder, seeing slightly improvement after several tests. The number of heads for the Multi-Head Attention are also 4 for the same reason and the dimension of the embedding layers

and all the others sub-layers is set on 256, following the state-of-the-art flow. We set the dimension of the Feed-Forward layers to 512. We also set on each layer a dropout rate of 0.2 in order to cope with overfitting.

Following the rhyming scheme of the Divine Comedy, we set the seq_length = 3 so as to have the minimum 3 verses as input in order to be able to predict the next line of a triplet or the flag "=end_terzine=" that indicates the end of a tercet.

For the train/val split, we divided it into 70%/30% ratio, and by state-of-the-art inspiration, we chose a batch size of 64.

We trained our models for 120 epochs, in order to achieve the best possible results before encountering a Plateau that stops improving the model.

As in the provious models, at the text generation step we experimented with many values of temperature, finding that the best results are achieved in the range between 0.5 and 0.75.

5.0.1 Optimizer & Learning Rate

We used the Adam optimizer with the following parameters `(learning_rate, beta_1=0.9, beta_2=0.98, epsilon=1e-9)` to compile the model, where the *learning_rate* parameter is generated by the function `CustomSchedule(d_model)`.

We decided to use this dynamic learning rate following the definition of custom learning rate scheduler according to the paper [7]

```
temp_learning_rate_schedule = CustomSchedule(d_model)

plt.plot(temp_learning_rate_schedule(tf.range(40000, dtype=tf.float32)))
plt.ylabel("Learning Rate")
plt.xlabel("Train Step")
```

```
Text(0.5, 0, 'Train Step')
```

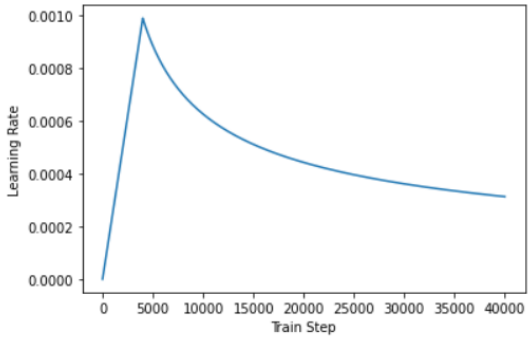


Figure 5.3: A sample of the transformer learning rate scheduler.

5.1 Results

The model returns very good results, both from a first graze to the generated text and from our metrics's perspective.

In fact, by using the homemade syllabification module, we obtained:

1. Number of putative terzine: 100
2. Number of well formed terzine: 99
3. Average structuredness: 0.9925
4. Average hendecasyllabicness: 0.9360
5. Average rhymeness: 0.9421
6. Ngrams_plagiarism : 0.9953

And for the professor Asperti's dataset, we have:

1. Number of putative terzine: 100
2. Number of well formed terzine: 99
3. Average structuredness: 0.9925
4. Average hendecasyllabicness: 0.9481
5. Average rhymeness: 0.93877
6. Ngrams_plagiarism : 0.9933

As we can also see from the metrics, with this model we managed to reach a very satisfactory level of emulation with respect to the original text's style.

We must point out that the rhymeness metric's strictness is very relaxed, and computes rhymes with some degree of tolerance (giving partial points for non exact rhymes) so in the end the overall percentage of well-formed rhymes is lower than the one virtually computed by the metric. But since these metrics have been sharedly used by the other groups, we had to use them in order to make comparisons.

```
e io attendi tanto quantunque senti
che si nome e di giganti tormenta
che pria di serra i risponder vansi

lo nostro amor che forse la ntenta
più e men di tutta quella image
dovea pennuto a te cosla enta

non eran ancor fuor quel nase
per la sua natura assai e garra
perchè la sua novitità si difense

chè per loro era se tal tiguerra
ond ei però con la prima gloria
vuo saper se in loco onor satterra

ma voi vedere un uomo che si chiama
falsificienza in terra quella
cha proprio è sì attra ne la strada

guadagnerò la luce più che suggella
del nostro serpensier de la mente
e chi di fior grado in giù la fiera

ciò che non desiderà quest la mente
ma perchè non cessare alcuna riversa
tanto amor che nulla parvente

or va dun dee or che se non ersa
avea in nuovi in meco si rispuose
del mio anaspettando ho diversa

chè perchè tanti non aspettando nomi
che di là ve tu ne la tua parvenza
come vedi che di sùbito ambemi

quando rispuosi come la spenenza
se già per me cantar giù tra noi
cesare fummo alcun secondo sentenza
```

Figure 5.4: A sample of text generated from a generated syllables dataset.

```
quando noiberava parte onde strada
o come nincerchio che più tinse l monte
etterno palazion più e vada

e questa superbia qui si risente
non ti torrà lo quali e quanto loco
grave mhan fatto in punto pensate

la bocca che per cantare al foco
mondo paeo dinanzi a la balestia
di retro a le pesol di voi è non gioco

e per chio rivolsi chi sen la gialia
la de la man sinistra simisura
la vedovella si chiama ancor malia

ma per lo cui si fosse per vera
non valle non parere a la veduta
chè santa chiesa come la fede cera

allor fu la paura un poco questa
che nel mondo per la gran ripa
e del mondo del bel sangue si cresta

ond io la toppa che non la vampa
de la mia montare altra via persona
e sì ci volse e vien la provampa

amor cha mar per la tua dona
qual troppo dinanzi a la sua margherita
e di proferza da terra il sera

quindi come la gente che fia si gita
ver la terra chordinanzi al terrene
li dirò per grazia che ti partita

piacque lo spirito sanza parvene
e ancor dinò come tu ta salute
là dove suo fattor lo buon compene
```

Figure 5.5: A sample of text generated from a *asperti/Dante* syllables dataset

Chapter 6

Conclusions

In the following table are synthetized the metrics performances of our best models:

Model	N° of putative terzine	N° of well formed terzine	Avg structuredness	Avg hendecasyllabicness	Avg rhymeness	Ngrams plagiarism
1) RNN with Custom Loss (Homemade syllabification)	46	45	0.9639	0.9385	0.2668	0.9814
2) RNN with Custom Loss (prof. Asperti syllabification)	46	45	0.9784	0.9584	0.2575	0.9943
3) RNN with Custom Loss and forced rhymes	51	51	1.0	0.9643	1.0	0.9965
4) Transformer (Homemade syllabification)	100	99	0.9925	0.9360	0.9421	0.9953
5) Transformer (prof. Asperti syllabification)	100	99	0.9925	0.9481	0.93877	0.9933

Figure 6.1: Summary of the models results.

Overall, the hendecasyllabness and terzine structuredness parts of the *Deep Comedy* problem were relatively easy to solve, as proven by the fact that almost all our models were able to achieve from-passable-to-good results on their corresponding evaluation metrics. On the other hand, only the Transformer model was able to seemingly learn and reproduce an imitation of the Comedy's rhyme scheme.

Bibliography

[1] Keras: the Python deep learning API. https://keras.io/

[2] Andrea Zugarini et al. "Neural Poetry: Learning to Generate Poems using Syllables"(2019)

[3] Github - asperti/Dante project. https://github.com/asperti/Dante

[4] Andrej Karpathy. "The unreasonable effectiveness of recurrent neural networks" (2015)

[5] Diederik P. et al. "Adam: A Method for Stochastic Optimization."(2017)

[6] Tensorflow tutorial for RNN text generation. https://www.tensorflow.org/tutorials/text/text_generation

[7] Ashish Vaswani et al. "Attention is all you need" (2017)

[8] Tensorflow tutorial for Transformer implementation. https://www.tensorflow.org/tutorials/text/transformer

[9] Maxime. "What is a Transformer?" (2019) https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04

[10] Giuliano Giacaglia. "How Transformers Work" (2019) https://towardsdatascience.com/transformers-141e32e69591