The Deep Comedy 2.0

Federico Spurio
Marco Foschini

September 2021

1 Introduction

Figure 1: Representation of Dante Alighieri...

In occasion of the 700° anniversary of Dante's death, we are asked to work on the project of **syllabification** of poetry.
Why Dante? Dante (XIII century) is one of the most influential and greatest

poet of all time, not only in Italy, but in the entire world. He is also considered the father of the Italian language.
The *"Divina Commedia" (Divine Comedy)*, his masterpiece, represents a "dream" in the Christian afterlife. It is divided in three parts: *Inferno (Hell), Purgatorio (Purgatory)* and *Paradiso (Paradise)*. In each book, Dante has encounters from political opponents to esteemed exponents of culture.
Besides the cultural weight, the *"Divine Comedy"* is very suitable to train a Neural Network, because of its 14,233 lines of hendecasyllables [1], in which we can find many rhetorical forms, that might affect the number of syllables.
Our tasks ware to:

- **Build a Neural Network for poetry syllabification**, producing a syllabified version of the verse in input
- **Generate verses in Dante style**, exploiting the previous algorithm for obtaining well formedness

Figure 2: ...or this is the representation of Dante? There is a discussion, after the finding of this painting, as to whether Dante had a beard or not[2]

2 Syllabification

The implementation of the Neural Network for syllabification is divided in these fundamental steps:

1. **Preprocessing**: reduce the input text in a "standard" format readable by the next state
2. **Tokenization**: create tokens from the preprocessed text. Those tokens are passed on to the transformer
3. **Transformer**: the "core" of project. Through a series of encoders and decoders, transform the text in input producing the syllabified version of the text as output

2.1 Preprocessing

Despite the complexity of the Neural Network, the use of this project is accessible to everyone, even to those who have no knowledge in the field of computer technology. As a matter of facts, this project could be seen as a pipe-line: the user insert the input and, at the end of the processing, an output is produced. Due to this accessibility, the layout of the input text can vary from user to user. Therefore, it is extremely important to preprocess the text and reduce it to a "standard" format.
To create a proper data-set to train our network with, we leveraged the source text shared in the project *Asperti - Dal Bianco*[3], containing all the *"Divine Comedy"* divided by syllables. The text looks like this:

```
Inferno • Canto I

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 .
```

Then we proceed by converting all the text in lowercase. We can also delete some character, like the punctuation, because the continuity, in the metric system, is not affected by it [4]. So we deleted:

- Titles (in form *"Inferno • Canto..."*)
- The enumeration at the beginning of each line
- Blank lines
- The group of characters ?!;:.,«»“”()-[]

Then the text obtained is further processed for the creation of the files *dante_training.txt* and *dante_result_training.txt*.
The first one is obtained by deleting the special character "|", that represent the division between syllables. So we are able to get a version of the *"Divine Comedy"* where each line is a verse, as below:

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

The second file contains the text with additional information, indicated by different tags. These are useful to get a full tokenized text. The tags used are:

- S: represents the space between words.
- I: represents the division between syllables.
- T: represents the start of the verse.
- E: represents the end of the verse.

In the end, the final result will be like this:

```
T I nel S I mez I zo S I del S I cam I min S I di S I no I stra S I vi I ta E
T I mi S I ri I tro I vai S I per S I u I na S I sel I va S o I scu I ra E
T I ché S I la S I di I rit I ta S I via S I e I ra S I smar I ri I ta E
```

After the creation of these two files, we generate the wordpiece vocabulary, needed for the *Bert tokenizer*[5]. The dictionary is then stored in the file *dante_vocabulary.txt.*.

2.2 Generate Data-set

After creating the two text file, containing the raw and tokenized version of the *"Divine Comedy"*, we generate the training, validation and test set, by also taking advantage of the function from Tensorflow `from_tensor_slices`.

2.3 Tokenization

A model cannot be trained directly on text. As a matter of facts, it needs to be converted to a numeric representation. To do that we used a custom tokenizer class containing an instance of *Bert tokenizer* `text.BertTokenizer`[6]. The setup of this class is realtively simple, it only needs as arguments the characters (or sequence) used as custom tags in the text, and the path representing the location of the vocabulary in the file system.
The vectorization of text is done by turning each text into a sequence of integers, corresponding to an index of a token in a dictionary. Furthermore the tokenizer will be responsible of turning the tags into characters, the translation between these two elements is described in the function `cleanup_text`.

2.4 Transformer

After defining the tokenizer, we are ready to implement the Transformer. Considering that we have to deal with written "human" text (so we are in the *NLP*

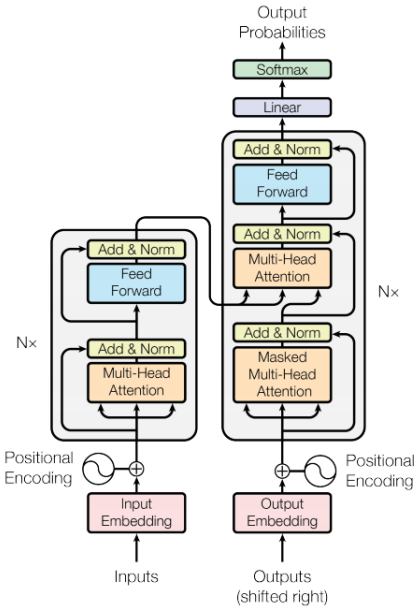


Figure 3: Transformer model

- *Natural Language Processing* field), the use of **Transformers** is somehow driven by the fact that they have become the model of choice for NLP[7]. A **transformer** is a deep learning model that adopts the mechanism of attention, differentially weighing the significance of each part of the input data[8]. Connected stacks of encoders and decoders are at the base of the structure of a transformer. The core idea behind the Transformer model is self-attention, i.e. the ability to attend to different positions of the input sequence to compute a representation of that sequence.
Following the procedure described in the official *Tensorflow API tutorial*[9], we create the following structure (graphically represented in the figure 3):

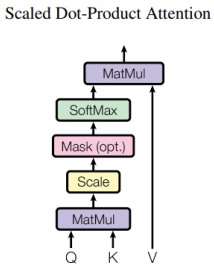
- The **positional encoding**: the input of the attention layers is a set of vectors, with no sequential order. So, to give the model information about relative position of the tokens in the sentence, a positional encoding is added
- The **padding mask**: during the process of tokenization, to ensure that all the sequence in the data-set has the same length, padding sequences have been added. Therefore, to ensure that model does not treat padding as input, all the pad tokens in data-set are masked (output 1 if pad value, 0 otherwise)
- The **attention function**: a multi-head attention function is used in order

to expands the model's ability to focus on different positions for different purposes

- The **encoder** and **decoder**: both encoder and decoder layers are composed by two sublayers, *(masked) multi-head attention* and *point wise feed forward networks*. The decoder layer has an additional multi-head attention sublayer, that receive the encoder output as input. As a matter of facts, the input sentence pass through `num_layers` encoder layers that, for each token in the sequence, generate an output. The decoder use that output as input for predict the next word

2.4.1 Attention

The *"Scaled Dot-Product Attention"*[10], used by the transformer, takes in input queries and keys of dimension d_k, i.e. Q (query), K (key) and V (value). We follow the choice of use dot-product attention instead of additive attention, because the first one is much faster and more space-efficient, thanks to the highly optimized matrix multiplication code.



It's called scaled because for large value of d_k, the authors suspect that the dot product grows large in magnitude, pushing the *softmax* function into regions where the gradients is extremely small. The scale factor is $\frac{1}{\sqrt{d_k}}$

At the end, we can summarize the attention formula, corresponding to the matrix of outputs, as:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}}) \tag{1}$$

To avoid the complexity of performing a single attention function with *d_model*-dimensional keys, values and queries, we split Q, V and K into multiple heads. As a matter of facts, *Multi-head attention* allows the model to jointly attend to information from different representation subspaces at different positions. At the end of the split, each head has a reduced dimensionality, and the computational cost is similar to the single-head attention with full dimensionality. For instance, in our project, we used 4 heads (`num_heads=4`).

2.4.2 Checkpointing

The time taken for the model to train itself, is not relatively short (≈15 minutes), and depends on the machine where the code is run and the power of the GPU. Due to this problem, we decide to implement a checkpoint system, that, after 5 epochs of training, save automatically the model, so that we train the model once.

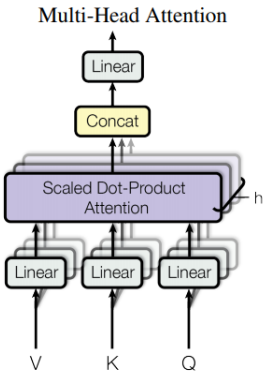


Figure 4: Multi-Head Attention structure

This method does not influence the time complexity neither the accuracy of the model. It is meant for testing or distribution purpose.

2.4.3 Train the transformer

After defining each part of the transformer, we are able to wrap it into the class called `ModelTransformer`, and begin its training. To do a proper evaluation of the model, we divided the entire data-set of Dante's verse into **train set** (10816 lines), **validation set** (2705 lines) and **test set** (712 lines). The larger the training set, the higher is the computational time.
As input to our model, we gave the train and the validation sets shuffled and batched. Then we need to set the hyperparameters for the transformer, for the first tests we use the one show in the Tensorflow Transformer tutorial [9], that is:

- `num_layers=4`: the number of encoder and decoder layers
- `num_heads=4`: numer of heads in the multi-head attention
- `dff=512`: dimensionality of inner-layer in the feed-forward network
- `d_model=256`: dimensionality of the outputs of the sub-layers and embedding layers in the encoder
- `dropout_rate=0.1`: used for regularization. Dropout is applied to the output of each sub-layer. It is useful to avoid over-fitting

Even if also the Tensorflow Transformer tutorial refers to the paper "Attention is all you need" [10], the parameters are smaller and, somethimes, the half. This happens because the data-set of the cited paper is much bigger than ours, so we would have risked over-fitting.

2.4.4 Loss and Accuracy results

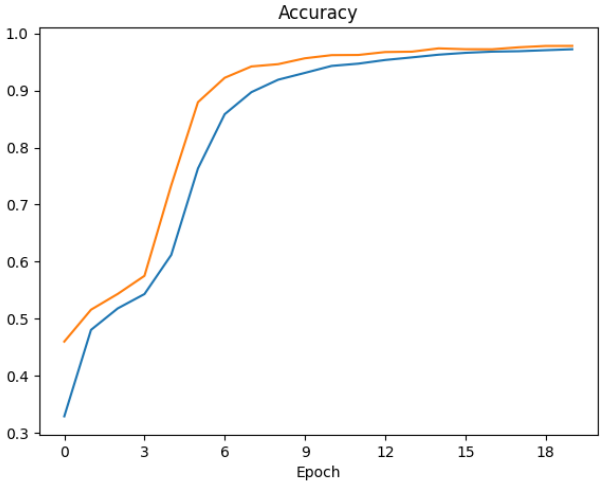


Figure 5: Accuracy graph. In blue the train set, orange the validation set

The graphs in figures 5 and 6 show the accuracy and the loss function of the model as a function of the epochs of training. In these figures, the blue line represent the trend of the training set, and the orange one the validation set. Both the validation and training function follow the same trend and grow together; we can conclude that the over-fitting tends to zero.
At end of the training, corresponding to epoch 20, the loss function takes value of 0.12, while the accuracy is 97%. The time taken by the training is of around 30 minutes, but, as already explained, this process is done only once. We believe these are satisfactory results.

2.5 Results of the syllabification

The results of the syllabification process are quite satisfactory. We tested the model with different poems of different centuries. Paradoxically, we obtained better results with modern poems (respect to Dante), than poems of Dante himself.
Here are some examples:
Orlando Furioso (XVI Century):

Original:
Le donne, i cavallier, l'arme, gli amori,
le cortesie, l'audaci imprese io canto,
che furo al tempo che passaro i Mori

Syllabified:
|le |don|ne i |ca|val|lier |l' ar|me |gli |a|mo|ri

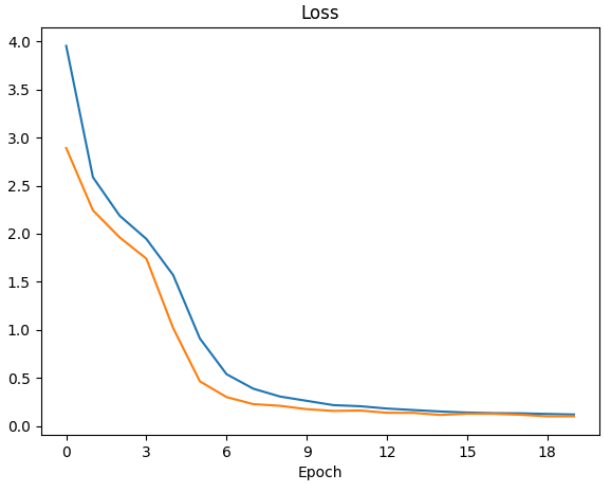


Figure 6: Loss graph. In blue the train set, orange the validation set

|le |cor|te|si e |l' au|da|ci |cim|pre|se|se |can|to
|che |fu|ro al |tem|po |che |pas|sas|sa|ro i |mo|ri

Passero solitatio (XIX Century):

Original:
D'in su la vetta della torre antica,
Passero solitario, alla campagna
Cantando vai finché non more il giorno;

Syllabified:
|d' in |su |la |vet|ta |del|la |tor|re |an|ti|ca
|pas|se|ro |so|li|tar|io al|la c|pa|gna
|can|tan|do |vai |n|ché |non |mo|re |gior|no

Tanto gentile e tanto onesta pare (XIII Century):

Original:
Tanto gentile e tanto onesta pare
la donna mia, quand'ella altrui saluta,
ch'ogne lingua devèn, tremando, muta,

Syllabified:
|tan|to |gen|ti|le e |tan|to o|ne|sta |pa|re
|la |don|na ia |quan|d' el|la al|trui |sa|lu|lu|ta
|ch' o|gne |lin|gua |de|vèn |tre|man|do |mu|ta

At the end, some syllables are repeated, and some words are misspelled (*e.g. "campagna"* in *"Passero solitario"* is converted in *"cpagna"*).

3 Generation

The generation part will be mostly similar to the syllabification process, in fact we will have the same steps, which are:

1. **Preprocessing**: reduce the input text in a "standard" format readable by the next state
2. **Tokenization**: create tokens from the preprocessed text. Those tokens are passed on to the transformer
3. **Transformer**: the "core" of project

3.1 Preprocessing

The preprocessing part is almost similar to the previous one. All the characters and symbols, that are delete for the syllabification, will be deleted also here, so in the end the files *dante_training.txt* and *dante_result_training.txt* will be almost similar. However there are some changes; first of all the list of tags is slightly different, in fact we will have:

- S: represents the space between words.
- B: represents the start of the verse.
- E: represents the end of the verse.
- I: represents the division between syllables.
- T: represents tercets.
- E: represents the end of the verse.

Secondly, in *dante_training.txt* is present the tag T, this will help in creating the Data-set. In the end the file will be like this:

```
nel mezzo del cammin di nostra vita
mi ritrovai per una selva oscura
ché la diritta via era smarrita T
```

While *dante_result_training.txt* will resemble this structure:

```
B I nel S I mez I zo S I del S I cam I min S I di S I no I stra S I vi I ta E
B I mi S I ri I tro I vai S I per S I u I na S I sel I va S o I scu I ra E
B I ché S I la S I di I rit I ta S I via S I e I ra S I smar I ri I ta E T
```

3.2 Generate Data-set

After creating the two text file, we can finally generate the Data-set. Instead of store verses we decided to store tercets, this was done to let our transformer to learn to compose rhymes. All the other instructions are the one used during the syllabification part.

3.3 Transformer

The transformer class will be the same, however we have decided to change some of the hyperparameters; we will have:

- `num_layers=6`: the number of encoder and decoder layers
- `num_heads=8`: numer of heads in the multi-head attention
- `dff=2048`: dimensionality of inner-layer in the feed-forward network
- `d_model=512`: dimensionality of the outputs of the sub-layers and embedding layers in the encoder
- `dropout_rate=0.1`: used for regularization. Dropout is applied to the output of each sub-layer. It is useful to avoid over-fitting

The choice of this hyperparameters brought to a very high computational time for the training of the model ($\approx$ 3 hours).
We tried with another configuration of the hyperparameters (`num_layers=4`, `num_heads=4`, `dff=512`, `d_model=256`) but the results are worst than with the above configuration.

3.4 Greedy search with top-k sampling

The key idea for generating text is that if we apply a softmax function to the output of the transformer, we will obtain the probability of appearing of each word. So with this in mind, we can use this information and a search strategy to create our text. We have decided to implement a greedy search strategy together with a top-k sampling to explore the solution space (as shown in figure 7). What it actually does is picking the k most likely words and the probability is redistributed among them. After that, we will pick one of the k^{th} element at random; this process is repeated until the `END` token is encountered.

3.5 Loss and Accuracy results

The choice of the hyperparameters and the dimension of the training set can be the reason of the high loss and the low accuracy of the model after 20 epochs of training. As a matter of facts, we ended up with 1.7 for the loss function and 56% of accuracy. With a powerful hardware that can be used for many hours in a row, we could have managed to train our model for 50 or more epochs, getting, probably, better results. As shown in figures 8 and 9 Both the validation and

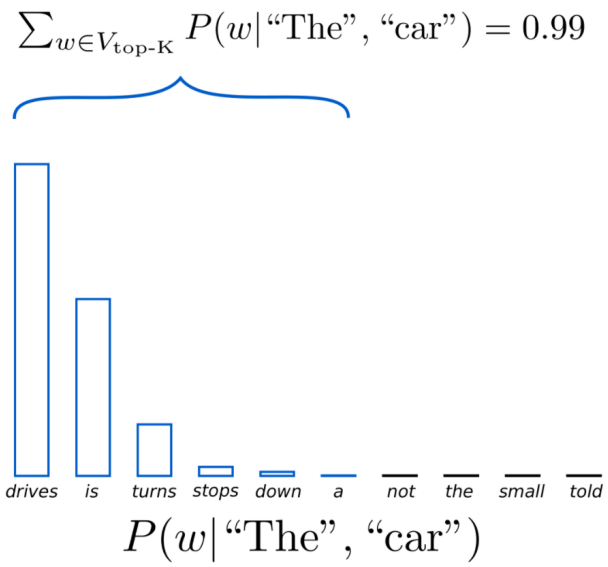


Figure 7: Illustration of top-k sampling, with K = 6

training function follow the same trend and grow together; we can hypothesize that the questionable results are not due the over-fitting of the model.

3.6 Results of generation

Unfortunately, due to hardware limitations, we managed to train the generation model for only 20 epochs. This led to a limited generation of tercets, comoposed of verses with little sense. Despite this, we were able to ascertain that the direction of development is correct, as the syllables of the words are respected and also the *"Dante tercets"*, i.e. ABA CDC...
An example of a generated tercets (with k=4):

```
|di|ve|ti |ti |spi |che |ri|scrit|te |ti
|dan |ti an|da|ti |gran|di |di |ti|ri |scie
|u|na i |ti|ra |ri |scin|ti |spen|da |ti
```

The syllable *"ti"* is repeted many times, maybe due to the choice of greedy search. However, the verses are in the form of a hendecasyllable and the rhymes are respected.

4 Conclusions and future developments

The results of the Neural Network for syllabification are quite satisfactory, with a rare mistakes in the predictions, that leads the model to produce a repeated

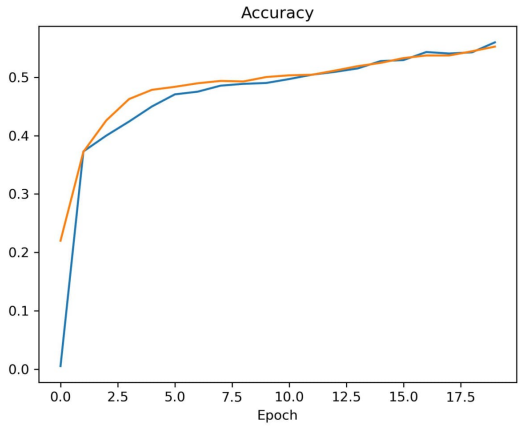


Figure 8: Accuracy graph. In blue the train set, orange the validation set

syllable. The accuracy of the model is very high and, with regard to rhetorical figures, sinalephs and dialephs are often respected.
As a future development, we could increase the training set, including verses from other authors and centuries. We could improve the recognition of rhetorical figures, so as to locate also enjambements, caesure, syneresis and others.
The choice of transformer proved to be more than adequate for handling the natural language.
Regarding the generation of verses, we certainly can train the model for more epochs, with better hardware, and with more poems, with the aim of varying the vocabulary.

The project discussed in this paper can be found at the following link: https://github.com/FedeSpu/DanteSyllabification.git [11]

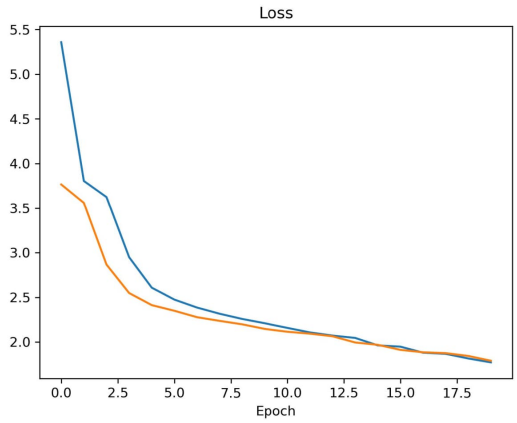


Figure 9: Loss graph. In blue the train set, orange the validation set

References

[1] Wikipedia - *Divine Comedy* https://en.wikipedia.org/wiki/Divine_Comedy

[2] https://www.lanazione.it/cronaca/dante-aveva-la-barba-a-orvieto-spunta\-un-inedito-dipinto-del-suo-volto-1.6181742

[3] Andrea Asperti - Dante https://github.com/asperti/Dante

[4] Andrea Asperti and Stefano Dal Bianco. 2021 - *Syllabification of the Divine Comedy* - J. Comput. Cult. Herit - 14, 3, Article 27 (July 2021) https://doi.org/10.1145/3459011

[5] Tensorflow API tutorial - Subword tokenizers https://www.tensorflow.org/text/guide/subwords_tokenizer#generate_the_vocabulary

[6] Tensorflow API - text.BertTokenizer https://www.tensorflow.org/text/api_docs/python/text/BertTokenizer

[7] Andrea Asperti - *Attention and Trasnformers* - Lesson 18 Deep Learning Unibo A.A. 2020/2021

[8] Wikipedia - Transformer (machine learning model) https://en.wikipedia.org/wiki/Transformer_(machine_learning_model)

[9] Transformer model for language understanding - Tensorflow API tutorial https://www.tensorflow.org/text/tutorials/transformer

[10] Ashish Vaswani and Noam Shazeer and Niki Parmar and Jakob Uszkoreit and Llion Jones and Aidan N. Gomez and Lukasz Kaiser and Illia Polosukhin. 2017 - *Attention Is All You Need* https://arxiv.org/abs/1706.03762

[11] Federico Spurio and Marco Foschini - *The Deep Comedy 2.0 Project* https://github.com/FedeSpu/DanteSyllabification.git

[12] Patrick von Platen - How to generate text: using different decoding methods for language generation https://huggingface.co/blog/how-to-generate