Profonda Commedia

Deep learning project on Natural Language Generation of canticas from Divina Commedia using Dante’s style

(https://github.com/dropino/ProfondaCommedia)

Nicholas Carroll

nicholas.carroll@studio.unibo.it

Simone Vagnoni

simone.vagnoni2@studio.unibo.it

Abstract

In this work we developed two models able to generate texts similar to Dante’s Divine Comedy, with particular attention given to making sure that verses are correctly syllabified.

The technique used to accomplish these systems is the Transformer, a neural network architecture that is widely used in Natural language processing, and also that gives the best results, according to the literature on the subject.

Two different architectures have been developed: the first is composed of a **single Generator model trained on pre-syllabified data** to directly generate syllabified verses. The second one is composed of a **Generator trained with non syllabified data** which generates verses and a **Syllabifier trained with syllabified data** to demonstrate that the generated verses are correct.

We trained the network on a dataset consisting in the Divine Comedy (both divided in syllables and not) and several experiments were conducted to tune the hyperparameters to get the best possible results.

1. Data

To train our networks we used two different versions of the divine comedy. One is a plain, non syllabified version of the divine comedy, while the other was the syllabified version created by professor Asperti (https://github.com/asperti/Dante) that we used to train both the Generator in the all-in-one architecture that generates syllabified text, and the Syllabifier that in the other architecture syllabizes the verses generated by the unsyllabified Generator.

The dataset is quite small as it consists of only 14 233 verses. We had also planned to use other works of Dante or other authors to increase the amount of available data but their different structure and style proved troublesome when trying to generate text in the style of the divine comedy.

The datset was divided in training set, validation set and dataset. Since there was not much data we divided the verses in 95% for the training set, 3% validation set, 2% testing set.

1.1 Cleaning the data

The first step in using the text was to clean the data by removing the verse numbers, the titles of each canto (for example: “inferno Canto i”) and any special characters such as •,—, -, (,), « and ». Punctuation was also removed except for question marks and exclamation marks.

1.2 Syllabification

Initially we worked with a plain text version of the divine comedy and wrote an algorithm to syllabify it. Results were acceptable but we encountered some problems with the synalepha given its dependence on the context. Synalephas are a metric figure in which, when counting syllables of a verse, two syllables are merged. If the figure is not respected the verses end up having the wrong number of syllables.

Given these difficulties we decided to use an already hyphenated version of the divine comedy made by professor Asperti. Training the networks with this version generated text that better followed the hendecasyllabic structure of the divine comedy and most importantly had less problems with synalepha.

1.3 Tokenization and creating the dataset

A model can’t be trained directly with text, it has to be converted to a numeric representation. This is usually done by transforming the text in a sequence of numeric token IDs.

At first we tokenized the text at a syllabic level using the syllabified dataset we had but it proved to be less than ideal as it led to numerous repetitions within verses, generating examples such as “e così così così così così così”. To try and improve the generation we evaluated two main strategies, tokenizing the texts letter by letter and dividing the verses to create a subword vocabulary.

The main advantage of a subword tokenizer is that it interpolates between word-based and character-based tokenization. Common words get a slot in the vocabulary, but the tokenizer can fall back to word pieces and individual characters for unknown words. To do so we used the **BertTokenizer**, implemented as a class by TensorFlow with a higher level interface. It includes BERT's token splitting algorithm and a WordPieceTokenizer. It takes **sentences** as input and returns **token-IDs**.

After numerous tests to evaluate the best solution we ended up using letter tokenization in both generators and the Bert Tokenizer in the syllabifier.

We used a dictionary to map all the letters and symbols to an ID, ending up with 42 tokens. In this amount we also included 4 special tokens:

- “[SEP]” for each space
- and inserted at the beginning of each verse the token “[START]”.
- To make all verses the same lengths, we used the special character “[PAD]” to bring them up to the length of 100 tokens.
- At the end of each verse we appended the symbol “[STOP] “
- “[UNK]” for unknown tokens that are not present in the dictionary
- “[EOS]” for the end of a sentence.

For the syllabizer we created two different dictionaries as we treated the problem as a translation problem, transforming the non-syllabized text to a syllabized version. Each of these dictionaries has the same special tokens as the letter tokenizer except for “[EOS]” and “[SEP]”.

The syllabized text vocabulary is composed of 796 tokens while the non-syllabized text vocabulary is composed of 1738 tokens.

2.Model

Reading the literature about this subject, we learned that the best results for this kind of problems were given by the Transformer model, instead of the traditional RNN (Long-short term memory LSTM or GRU). One of the main reasons for this choice is that RNN processes one symbol at time and generates a sequence of hidden states h_t as a function of the previous hidden state h_{t-1} and the input for position t . This sequential way to operate **precludes parallelization** and it is **critical with long sequences**: the unrolled representation of a RNN results in a very deep neural network, leading to problems like vanishing and/or exploding gradients that are only partially corrected by the gated units. Instead the **Transformer** architecture, besides the parallelism with which it deals with the input data, can more easily catch the long term dependencies between complex syllabic structures.

2.1 Our architectures

As previously mentioned our work consists in two distinct architectures which we have called the syllabifier-generator and the all-in-one generator for convenience.

2.1.1 Syllabifier and non syllabified generator

So the model chosen is an architecture that ensembles **two different Transformers, a Syllabifier and a Generator**, that respectively translate the input in a syllabified output, and create verses with the style of the input. In particular the Generator, trained with the un-syllabified Divine Comedy, creates verses that syllabified from the Syllabifier, reach the objective of the task, that can be compared with the one of the **Generator** of the verses starting from the Divine Comedy Syllabified.

The first one, the Syllabifier, takes a file of the verses mapped (un - syllabified as) an input, that after being tokenized and **encoded**, is given to the transformer to train to associate the two texts (with and without the “|” symbol to separate syllables), and predict the most accurate “translation”, that after being **decoded**, gives the syllabified version of the text.

The other model, the Generator, takes as input the syllabified Divine Comedy, which is tokenized and the data is fed to the **encoder** that maps the input sequence of symbol representations to a sequence of continuous representations. Given that continuous one, the **decoder** then generates an output sequence of symbols one element at a time. At each step the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next. The Transformer follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder

2.1.2 All-in-one generator

The architecture for the all-in-one generator is practically identical to the Generator in our other Generator-Syllabifier model. The only differences lie in the pre-processing of the data and a slightly different implementation.

2.2 Positional encoding and embedding

Since this pattern contains no recurrence or convolution, positional encoding is added to provide the pattern with some information about the relative position of the words in the sentence. The positional encoding vector is added to the embedding vector. Embeds represent a token in a d-dimensional space where tokens with similar meaning will be closer to each other. But embeddings don't encode the relative position of words in a sentence. Thus, after adding the positional coding, the words will be closer to each other based on the similarity of their meaning and their position in the sentence, in the d-dimensional space. The formula for calculating positional encoding is as follows:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$

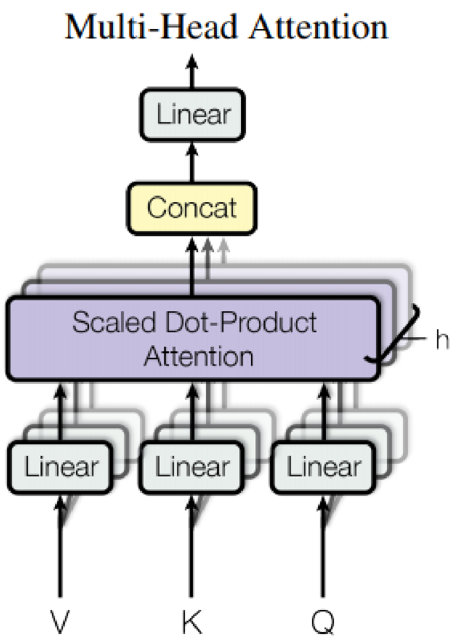
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

Beside the I/O sequences, a boolean *mask* is fed to the model in order to ignore padding symbols. All the pad tokens in the batch of sequence are masked. It ensures that the model does not treat padding as the input. The mask indicates where pad value 0 is present: it outputs a 1 at those locations, and a 0 otherwise.

2.3 Multi-head, scaled dot product attention

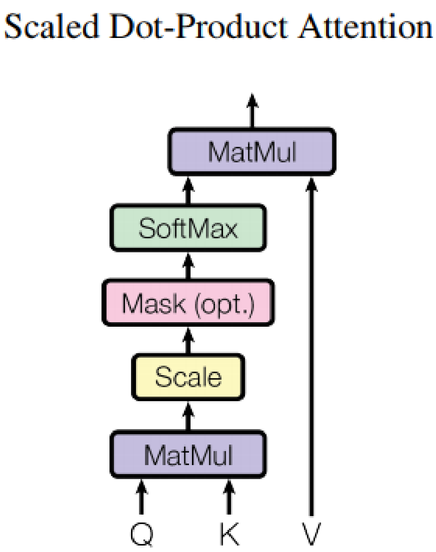
Multi-head attention consists of four parts:

- Linear layers and split into heads.
- Scaled dot-product attention.
- Concatenation of heads.
- Final linear layer.



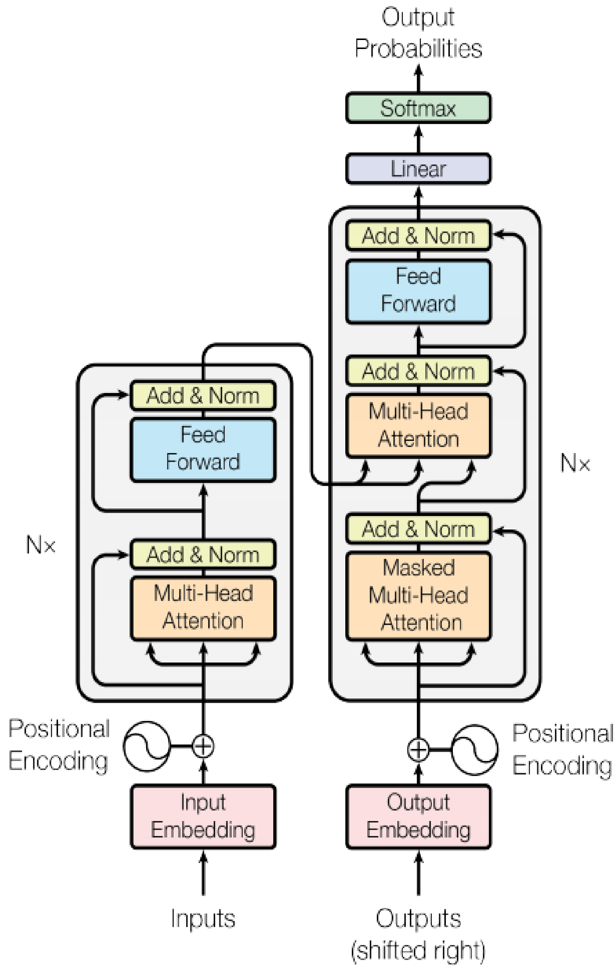
Each multi-head attention block gets three inputs; Q (query), K (key), V (value). These are put through linear (Dense) layers and split up into multiple heads.

The scaled-dot-product-attention defined above is applied to each head (broadcasted for efficiency). An appropriate mask must be used in the attention step. The attention output for each head is then concatenated and put through a final Dense layer.



Instead of one single attention head, Q, K, and V are split into multiple heads because it allows the model to jointly attend to information from different representation subspaces at different positions. After the split each head has a reduced dimensionality, so the total computation cost is the same as a single head attention with full dimensionality.

2.4 Transformer



A transformer is a deep learning model that adopts the mechanism of attention, differentially weighing the significance of each part of the input data. It is used primarily in the field of natural language processing (NLP) and in computer vision (CV).

Like recurrent neural networks (RNNs), transformers are designed to handle sequential input data, such as natural language, for tasks such as translation and text summarization. However, unlike RNNs, transformers do not necessarily process the data in order. Rather, the attention mechanism provides context for any position in the input sequence. For example, if the input data is a natural language sentence, the transformer does not need to process the beginning of the sentence before the end. Rather, it identifies the context that confers meaning to each word in the sentence. This feature allows for more parallelization than RNNs and therefore reduces training times.

2.5 Encoder

Each encoder consists of two major components: a self-attention mechanism and a feed-forward neural network. The self-attention mechanism accepts input encodings from the previous encoder and weighs their relevance to each other to generate output encodings. The feed-forward neural network further processes each output encoding individually. These output encodings are then passed to the next encoder as its input, as well as to the decoders.

The first encoder takes positional information and embeddings of the input sequence as its input, rather than encodings. The positional information is necessary for the transformer to make use of the order of the sequence, because no other part of the transformer makes use of this.

2.6 Decoder

Each decoder consists of three major components: a self-attention mechanism, an attention mechanism over the encodings, and a feed-forward neural network. The decoder functions in a similar fashion to the encoder, but an additional attention mechanism is inserted which instead draws relevant information from the encodings generated by the encoders.

Like the first encoder, the first decoder takes positional information and embeddings of the output sequence as its input, rather than encodings. The transformer must not use the current or future output to predict an output, so the output sequence must be partially masked to prevent this reverse information flow. The last decoder is followed by a final linear transformation and softmax layer, to produce the output probabilities over the vocabulary.

3. Generator

Our Generator is based on the Transformer, an **auto-regressive** model, meaning that it will predict the next token based on the past sequence already generated (decoder input).

In the all-in-one architecture, in order to generate a sample of text, we fed the decoder with an unseen starting sequence from the test set and then concatenated gradually to the input the token sampled according to the model prediction.

3.1 Top-K sampling

In order to choose the best syllables to append to the sequence, but not always the same, we used *Top-K* sampling, in which the *K* most likely next words are filtered and the probability mass is redistributed among only those *K* next words. GPT2 adopted this sampling scheme, which was one of the reasons for its success in story generation.

The most problematic aspect of TopK search is the worsening of results if it maintains the same *K* for every token to be generated. The best implementation of this would be to dynamically adapt the value of *K*, decreasing it toward the end of the sentence where the rhythmic scheme must be preserved. A trick is to make the distribution of chosen words sharper (increasing the likelihood of high probability words and decreasing the likelihood of low probability words) by lowering the so-called **temperature** of the softmax.

Setting the parameters k and t (temperature) to 1 the result is a **greedy** algorithm that chooses the most probable word, but due to its simplicity, could often lead to loops of sequences of same or similar words.

4. Syllabifier

The structure of the syllabifier is also a transformer with the same modalities of the previous one that using the BertTokenizer that embeds the verses in encodings, translates the unsyllabed verses into the syllabed ones. This is made training the network to associate the couples of verses (syll. and unsyll.).

In particular the steps that the transformer made to infer the next tokens are:

- Encode the input sentence using the Unsyllabified tokenizer (tokenizers.unsyll). This is the encoder input.
- The decoder input is initialized to the [START] token.
- Calculate the padding masks and the look ahead masks.
- The decoder then outputs the predictions by looking at the encoder output and its own output (self-attention).
- Concatenate the predicted token to the decoder input and pass it to the decoder.
- In this approach, the decoder predicts the next token based on the previous tokens it predicted.

5. Training

The transformers within the generator and the syllabifier were both trained with Google **Colab**.

Every 5 epochs we validated the models using the validation dataset and averaging the results. At prefixed epochs the models are used to generate a sample of text, in order to keep track of the performance at different times during training.

We kept track of train accuracy, train loss, validation accuracy and validation loss during the different training sessions in a consistent and ordered way using the **Weights and Biases** API, that allowed us to log and compare metrics, hardware statistics and generated data in a web application.

All networks were trained with batches of 8 tercets for a total of 24 verses at a time.

5.1 Generator hyperparameters

Like in classical Natural Language processing tasks, the loss function is the sparse **categorical cross entropy** between the predicted tokens and the pad masked target sequence.

We used the **Adam** optimizer with **β1**= 0.9, **β2**= 0.98 and ***epsilon*** = 10e−9 and a fixed learning rate of 2e-4, tuned during several experiments.

The other hyperparameters have been set as follows:

-number of encoder/decoder layers = 4

-dimension of Embedding and attention layers = 128*

-dimension of Feed Forward inner layers = 256

-dropout rate = 0.1

-number of attention heads = 4

-epochs = 50

The model, written in python, was developed using the **Keras** framework on top of **Tensorflow** 2.0.

5.2 Syllabification hyperparameters

The hyperparameter chosen for the generators are:

-loss function: Sparse categorical cross-entropy

We used the **Adam** optimizer with **β1**= 0.9, **β2**= 0.98 and ***epsilon*** = 10e−9 and a learning rate that varies based on this formula (https://arxiv.org/abs/1706.03762)

$$lrate = d_{model}^{-0.5} * \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$$

-number of encoder/decoder layers = 4

-dimension of Embedding and attention layers = 128*

-dimension of Feed Forward inner layers = 512

-dropout rate = 0.1

-number of attention heads = 8

-epochs: 20

With these setting we have a syllabization correctness of 97%.

6. Results

After several attempts, changing the hyperparameters of the consolidated base architecture (dropout in [0.1, 0.3, 0.5], learning rate in [1e-3, 2e-4], attention heads in [2, 3, 4]) we obtained the best results, in terms of the text evaluation metrics, with the reported configuration. The loss and the accuracy were respectively **0.67 and 0.84**.

But the hyperparameters that we found most influential in terms of the structure of the hendecasyllables generated were the evaluating parameters of the Top-k function: the number **k** and the **temperature t**.

Ranging the k from 1 to 15 and the temperature from 0 to 1, we found the best results with **k=3 and t=0.9**, because with higher k or lower t, the results where a monotonous repetition, often looping of the same words and syllables or the tercets were formed by the wrong number of verses.

Here following the text generated by the all-in-one architecture:

<table>
<tr><td>or | que| sti | che | da |l' in| fi| ma | la| cu| na|
de |l' u| ni| ver| so in| fin | qui | ha | ve| du| te|
le | vi| te | spi| ri| ta| li ad | u| na ad | u| na|</td><td>ci |s' | sen| do | cie| se el |l' on| ta| ra| ro|
che' | pri| scon | cial | co| ra| ra| ro | so| se | pia| sta|
es| si | la | son | son| do | so| mo| ra al | so| ro|</td></tr>
<tr><td>che |s' o| ria | che | cia | so| ma| sto | se| sce|
co| sci | se| scen| ter | la | sol| to | che | la| sto|
e | la | so| sta | co| scia| sto | suo | su| ra| ta|</td><td>se | ciò | che |ch'i' | lo | con| tro| ren| che | pren| to|
e | la | che | son | la |ch' io | che | che | si | per| to|
che | si |l' u| ma | sua | cor | piu | la | lo | pre| so|</td></tr>
<tr><td>che | chi |l' io | la | son| to | co|s' | sol| ta|
e | la | si| ri |l' o| so a| spa| no al| cia | si| ra| so|
si’ | che | con| te | se |l' a| ve| ta e| stre | spe| ro|</td><td>che | con| tre'n | cia| ri| tra | si | se | se| sce|
e | con | co| re| so | chi | chi | sol| la | lo| ri| sta|
e |l' a| ma | su| tra | li | la| tro | ci | so| stro|</td></tr>
</table>

e |l' al| li | chi | per | per| so| so | che | per| se|
che | son | cio |l' al| ta| ra| star | sua | chia| re|
el |ch' al| tra | pri| to | cor| sta | so| ra| sto | cor| to|

co| sce | son| tra | so| sto |l' a| me | la | con | pri|
e | se| ni | son | son |l' al| tra| se | so| la|
e |ch' a| mi| tra | so| me | so| li | co| ra | chi | pier| to|

sol| tra |l' al | pe| ra | si |l' ol | per| co a| ra| to|
e |ch' al| tar| si | so| sto a | cor | ser| si| to|
co|s' | so| le al| tri| sta | che | cor | sen| to|

el| lo| ro | co| men| te | la | che |l' io | si | pa| ta|
e | lo |ch' e| si| sto | sol| la | so| se | per| no|
che | che | li | che | suon| di| tre| re| sto | co| ste|

che | la | se | co|s' | la | cia | la |ch' al| ten| te| sta|
co| sto | la | che | le | che | si | so| ra| sce | spe| scio|
per i| mon | che | co| me | so| mo| ra| sto | mi | mor| te|

la | la'l | per | so| se | ser| che | che | che | spa| no|
ch' io | che | con| dor| sta| sto |l' a| se a| stra| ra|
per| tor | che | che | se | la | con | se| si | sol| le|

co| me | che | che'l | con | si | la| re a| me | con| de|

e |l' al| ta | per| ta | cia| stra |l' a | pe| sto|
si | che | si| stra| stre |l' al| tra | pe| ne | son | chio|
si |ch' a| ri| tri |l' al| li| to | sia| to | par| ta|

se| ve| ste| sto | con | so| ra e | co| ro| sce | cor | co| so|
se | suo| ra | se| mi |ch' al | co| stra | pe| ro|
si |l' ac| co | la | ci |ch' a| ra |ch' a | so| sto|

co|s' | la | si | la | suon | se | la| tre| te | pie| ta|
e | la | chi |l' a| vi| ra |l' ac| ci | sol | so| se|
e | la| mi| to e | la | son | pri| sta| ta | con| te|

lo | sia | con| tra| se |l' al| li al | si | che'l | se| te|
co| ra | lo| stro| sce| ra| ste| ra a| ra | cor| ta|
co| ra | se| ra | su | li | che | co| sche | se| sca|

che |l' u| si | che | so| sta | ci | so| stra | spre| sto|
e | con| do al| te | suo| ra | so| se | cian| de|
si | se| ra | la |ch' io | se | suo| ra | co| re|

di| stran| di | che | li | co| me |s' |l' in |ch' o| sten| to|
e | la | la | lo al | pri| ta| stra | la | so| ste|
e | la | la | lo | so| ren| ta | ser| se | so| stra| so|

co|s' | la | la | che | se | la | sun| ta | chio| se|

che a| se e | suo | con |l' a| ste | con| che | co| re|

ch' ion| chi | si | la'l | piu | per | so| so | si | ca| spo|

e | che |l' al | sol |ch' al| la | prar |ch' o| ri| to|

che | so| ste | so| sci e | son | so| mi |l' al| li | spi| ra|

so| ste |ch' a | suo | la | che | che'n | so| re|

co|s' | co| stra| re | cor| ci | le | son | son| ta|

che | se in| di |ch' io | si |l' al| tri | son| ta| ra|

cial | piu | suo | si | chi | la | la| stre | se| ra | per| se|

con |l' al| tri| sto | co| ran| to | chia | con | co| ste|

si |ch' a| stra | cio | so| ro | si | se| re| ra | so| na|

per | sen | la | si | si | la | so| me | son | si | par| si|

che | si| sia| to | se | sen | suo | son| to | so| sto|

ch' a| mi | cor| ta| ro a |l' a | suo| ra| sto | car| so|

co|s' | che | co| stre | la | son | so| ra| so | pa| sto|

si | sol | si | co| me | co| stran| tra | che | sen| to|

che al |ch' a| ster | se| ste | se | con | so| to|

per | la | che | che | so| stra |l' a | so| sco| so|

che |l' a| si | piu |l' o| si |l' al| tri| se | piu' | col| li|

chi | si | si | con | co| me | che e | su| si | sie| na|

di |l' al| ta | la| scia | la |l' a | pier| tro | sper| so|

e |l' o| spri | che | la | pra| ra a| se| ra| sten| ta|

co|s' |l' a| ri| sta| ra | sia| se | la | con| do|

coi| me | la | suo| na | con | se| re | se| ne| na|

se | che | ci | se| ra| sta| ne | le | la | sun| to|

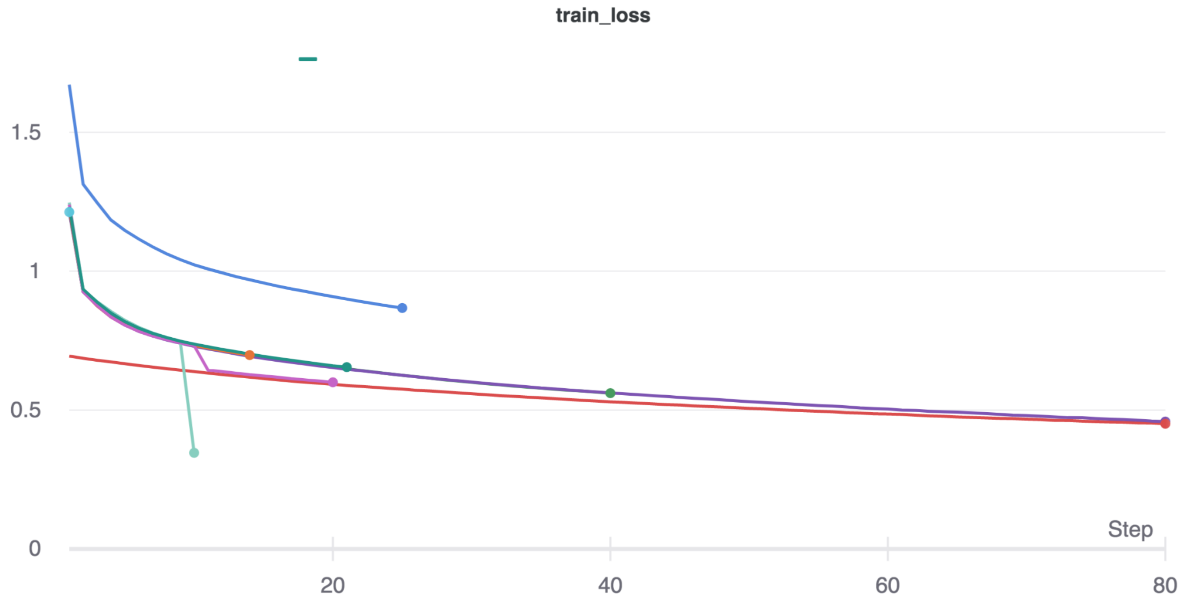
e | cor| so | co|s' | si | che | cial| ta a| mor| te|

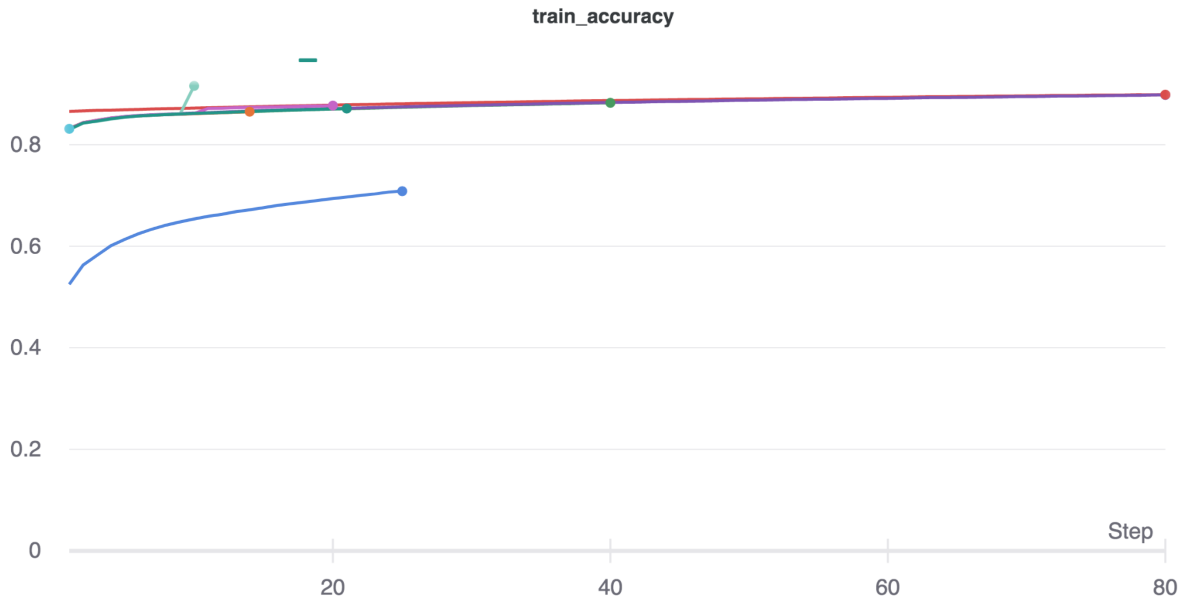
la |l' al| lo | piu| che | si | su| tor| ta| ra|

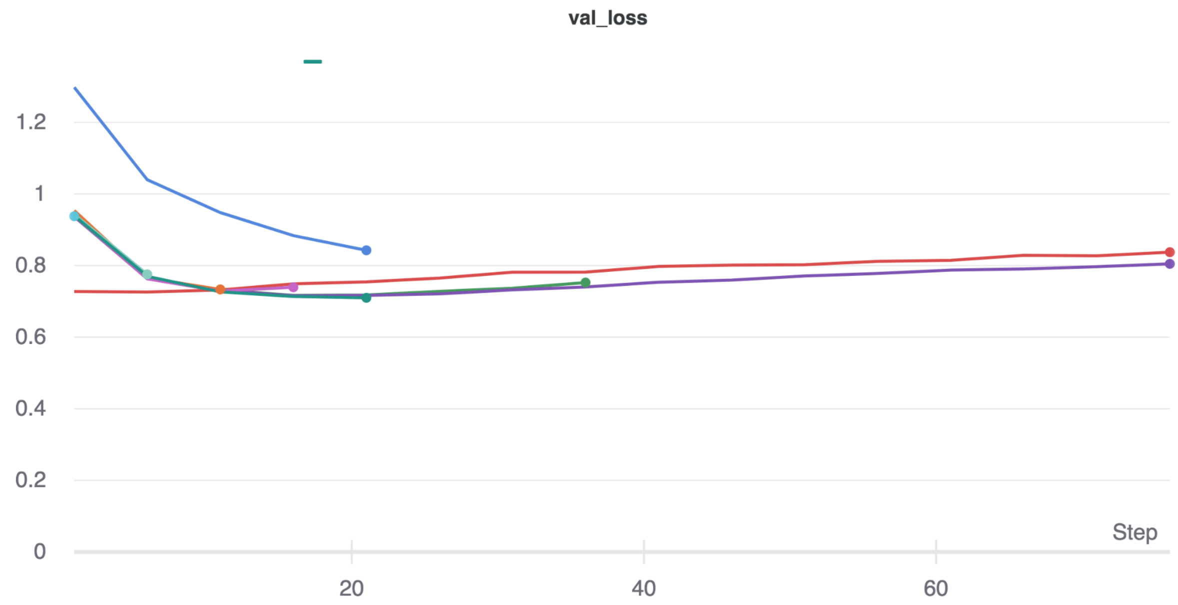
e | che | lar| si| te | se| ra | chia | se | si | sen| to|

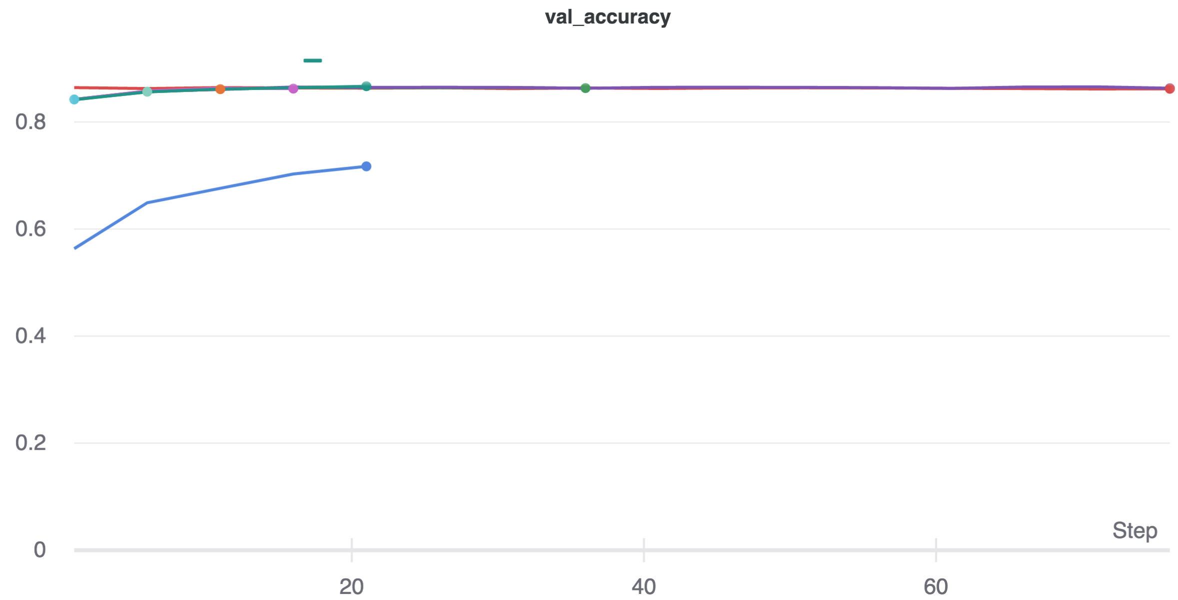
se |s' | la | so| ra | la | su| ra| te | la | par| sa|

la | sor | si| mi| to a | co| ma | lo | con | sor| ca|


train_loss
1.5
1
0.5
0
Step
20
40
60
80


train_accuracy
0.8
0.6
0.4
0.2
0
Step
20
40
60
80





Here reported in the figures the obtained loss and accuracy of the train and validation set of the all-in-one generator, with respect of the epochs.

Here following the text generated by the Generator-Syllabifier architecture:

nel |qual |non |si |dee |cre |der |che |s ’ in |vii	|che |la |suo |ra |son |tran |to |sen |che |la |sua

|per |cre |a |tu |ra |l ’ oc |chio |tan |to |chia |ro
|e |io |ch ’ al |fi |ne |di |tut |t ’ i |di |sii

|che |per |con |do |di |co |me |che |la |man |to
|che |sen |che |l ’ u |na |sol |tro |sua |re |na
|po |re |l ’ an |co al |con |che |per |con |de |na

|se |per |la |so |re |sor |dal |suo |e |stra |suo
|e |con |che |per |che |se |se |son |tro |se |glio
|ser |sen |to a |co |sta |se |per |san |con |de

|che |la |man |to e |la |suo |al |tro |sen |ta |ta
|ch ’ e |nis |se |chi |se |ra |che |la |son |sor |te
|e |sar |ra |te |suo |co |me |com ’ |a |sua |sua

|che |la |sor |che |l ’ an |to e |scen |do |sor |ta
|con |l ’ e |ra |con |sen |do |se |la |mar |te
|co |ri |che |sa |sen |to e |che |su |ra |suo |ra

|che |per |se |re |son |per |le |la |su |ra |na
|si |son |con |do |co |men |to a |la |son |par |ta
|co |si |son |suon |se |san |do e |per |che |per |rar |ra

|e |sa |re |con |sem |pen |se |suon |de an |ta |sen |te
|e |che |l ’ u |ne |so |le |dal |cos |sa |l ’ al |tro

|e |con |se |la |man |suon |de |ra |la |su |ra
|e |la |ser |ra |l ’ ar |ran |te |chi |per |sar |ra
|e |ze |che |l ’ an |che |se |gua |suo |le |per |te
|che |par |sen |te |son |ser |se |na |se |re
|e |co |me |re |la |suo |son |la |so |rem |ta |rar |se

|che |se |re |la |sen |son |che |se |l ’ er |sen |suo |to
|e |se |pa |rem |po |re |che |le |se |la |suon |te

|si |son |sen |l ’ a |rar |di |co |men |ten |den |de
|e |che |son |tro a |son |de |le |sol |ten |par |te
|che |le |sua |con |se |so |re |sua |suo |na

|ch ’ al |suo |che |son |de |con |te |car |ta |ta
|con |con |se |mer |son |se |san |la |pren |to
|el |per |sen |che |son |do |se |l ’ et |ta |ser |let |ta

|con |son |sar |sua |le |le |sua |ra |se |ra |ra
|de |per |le |ter |sen |to e |per |ch ’ io |sor |te
|ch ’ el |suo |son |che |sen |de |l ’ al |tror |si |son |ta

|e |l ’ an |tra |suon |der |le |par |to |son |ter |ne
|per |se |co |si |di |sen |tra |la |ser |car |te
|e |la |par |te |ra |con |san |ta e |sal |cas |so
|con |de |sen |te |sen |per |en |te |lan |te |cor |so

|con |ser |re |san |do e |sar |che |la |su |ra	|per |che |che |por |te |se |suon |do |so |re
|che |co |man |tra |con |che |pon |sen |so |le	|e |san |tro an |to |con |si |can |do |ser |len |te
|e |che |par |te |su |l ’ en |con |su |se |son |sen |te	
|con |sua |che |l ’ es |si |cos |se |pre |stre |la	|e |sen |co |sto |al |te |sor |di |per |sua
	|co |me |con |se |ma |la |ser |che |par |l ’ al |tro
|che |la |suo |an |te |suon |suon |suon |al |tro	|se |per |la |sur |lar |la |son |por |to al |per |sa
|di |so |ren |te |che |le |son |do a |suon |le |sem |mo	
|con |che |suar |che |par |ta |sen |te |pe |re	|e |l ’ ar |ra |la |par |che |le |suo |se |men |te
	|e |san |de |ro |sua |son |son |suon |con |per |to
|e |se |suon |sen |dor |che |di |su |se |ra |sen |na	|ch ’ io |che |le |tu |son |co |re |suon |so |tro
|che |sor |te |che |su |se |se |la |sor |to a |sol |tra	
|con |del |del |son |con |cos |sa |per |per |ta |ta	
	|son |sen |l ’ al |la |sen |to an |co a |sua |san |te
|che |l ’ al |man |do e |l ’ an |dan |son |son |so |te	|che |son |cam |bra |sua |ten |te |che |le |se |gno
|e |l ’ an |tra |co |me |la |suo |al |tre |suo |ser |ro	|se |san |do al |suo |ra |par |con |di |ser |per |co
|con |son |to |che |par |la |mor |sor |so |ra	
	|che |per |l ’ al |tro |son |to e |la |sem |sem |pe
|che |suon |con |to |se |sol |son |son |suan |ta	|e |sar |sen |co |me |se |la |ca |ren |che |so |ra
|di |com ’ |an |ten |da |chia |che |la |par |son |ta	|co |me |per |la |man |tra |la |par |l ’ ar |ta |ta
|che |la |suo |suol |che |san |te |ser |re |pen |ta	
	|e |san |te |l ’ on |da |con |sen |to a |per |se |ra

6.1 Text evaluation metrics

To evaluate the quality of the generated cantica we used mainly two metrics ranging in the interval [0..1], these however cannot replace human-based judgement.

- **Terces structureness**: the ratio between the number of well formed terces intended as 3 sentences separated by an empty new line and the expected number based on the total lines produced.
- **Hendecasyllabicness**: average compliance of each verse to the hendecasyllable pattern.

These are the results we obtained with the All-in-one architecture:

Terces structureness: 0.89

Hendecasyllabicness: 0.964

These are the results for the Generator-Syllabifier architecture:

Terces structureness: 0.86

Hendecasyllabicness: 0.963

7. Conclusions

Both models give very similar and satisfactory results with our metrics but as previously mentioned human based judgement is the best metric in this case. In particular we believe that the Syllabifier-Generator based model is the best one in this case.

8.Bibliography and Sitography

1. Vaswani A. et al,Attention Is All You Need, 2017 https://arxiv.org/abs/1706.03762
2. Asperti A., del Bianco S., Syllabification of the Divine Comedy, 2021 https://arxiv.org/abs/2010.13515
3. Minh-Thang L., et al. Effective Approaches to Attention-based Neural Machine Translation, 2015 https://arxiv.org/abs/1508.04025v5
4. https://www.tensorflow.org/text/tutorials/transformer#create_the_transformer;
5. https://www.tensorflow.org/text/tutorials/text_generation;
6. https://www.tensorflow.org/text/tutorials/nmt_with_attention;
7. https://www.tensorflow.org/text/guide/subwords_tokenizer#build_the_tokenizer
8. https://github.com/DanieleVeri/deep_comedy/blob/master/report.pdf
9. https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/
10. https://blog.ml.cmu.edu/2020/03/20/are-sixteen-heads-really-better-than-one/