The Deep Comedy

Deep learning project about Natural Language Generation of a cantica using the style of the Divina Commedia

Giacomo D'Amicantonio Daniele Verì

Abstract

In this work we developed a model able to generate a full cantica using Dante's terces and hendecasyllables verses. The chosen model is a Transformer, a relatively recent neural network architecture that we trained on a dataset obtained by decomposing the Divina Commedia in syllables. Each batch was made by four terces, in order to highlight the rhyme scheme ABA-BCB-CDC-DED. After several experiments we found an optimal tuning of the hyperparameters, and the resulting model outperformed the other based on simple RNN networks.
In the results we show one of our best generated cantica and its scores, followed by the learned representation of the language that the Transformer was able to catch.

1. Data

The dataset we used is composed only of the plain text Divina Commedia, therefore it's small in size: there are 14133 verses that compose 1711 batches of 4 terces.

The 4711 terces are so splitted among train, test and validation sets:
Train set: 1154 batches of 4 terces
Test set: 12 batches of 4 terces
Validation set: 11 batches of 4 terces

Each terce is divided into syllables and right padded until it reaches 76 tokens, so a batch is composed of 304 symbols.
The choice to utilize syllables comes from the ease of catching the rhythmic scheme and the small size of the resulting vocabulary.

1.1 Preprocessing

Firstly the text is filtered: punctuation "*. : ; « »*" and uncommon characters like "*[]-*" are removed, while accented vowels, exclamation and interrogation marks are preserved.
Secondly the lowercase text is word-tokenized and organized in terces.

Finally each word is decomposed in syllables that are encoded as integers. This is the most difficult part because there are many grammatical rules to take into account.

1.2 Hyphenation

We used the syllabification rules of the italian language to split words in syllables, assigning to each syllable an integer value[1].
The hyphenation pseudo code reported below, uses the function `is_diphthong` in order to check when a combination of two adjacent vowels sounds within the same syllable (same for the `is_triphthong` function), while `are_cons_to_split` returns if it's possible to split two consonant in different syllables.

```
function hyphenation(word):
    syllables = []
    is_done = False
    count = 0
    while not is_done and count <= len(word) - 1:
        syllables.append('')
        c = word[count]
        while not is_vowel(c) and count < len(word) - 1:
            syllables[-1] = syllables[-1] + c; count += 1
            c = word[count]
        end while
        syllables[-1] = syllables[-1] + word[count]
        if count == len(word) - 1: is_done = True
        else:
            count += 1
            if count < len(word) and not is_vowel(word[count]):
                if count == len(word) - 1:
                    syllables[-1] += word[count]; count += 1
                elif count + 1 < len(word) and
                        are_cons_to_split(word[count], word[count + 1]):
                    syllables[-1] += word[count]; count += 1
                elif count + 2 < len(word) and not is_vowel(word[count + 1]) and
                        not is_vowel(word[count + 2]) and word[count] != 's':
                    syllables[-1] += word[count]; count += 1
                end if
            elif count < len(word):
                if count + 1 < len(word) and
                        is_triphthong(word[count - 1], word[count], word[count + 1]):
                    syllables[-1] += word[count] + word[count + 1]; count += 2
                elif is_diphthong(word[count - 1], word[count]):
                    syllables[-1] += word[count]; count += 1
                if count + 1 < len(word) and
                        are_cons_to_split(word[count], word[count + 1]):
                    syllables[-1] += word[count]; count += 1
                end if
            else: is_done = True
            end if
        end if
    end while
    if not has_vowels(syllables[-1]) and len(syllables) > 1:
        syllables[-2] = syllables[-2] + syllables[-1]
        syllables = syllables[:-1]
```

```
    end if
    return syllables
end function
```

Although the correctness of these rules, the syllable count of each verse stands in an interval of 11 ± 2 because we didn't take into account the *synalepha* (due to the domain specific knowledge required).
The synalepha is a metric figure where two syllables are merged in one vocal position:

Sill 1	Sill 2	Sill 3	Sill 4	Sill 5	Sill 6	Sill 7	Sill 8	Sill 9	Sill 10	Sill 11
mi	ri	tro	vai	per	u	na	sel	va os	cu	ra

Once tokenized the dataset, a map of the syllables with the integer index is created. The final dimension of the vocabulary is 1874 tokens but it is limited to 1800 to remove the tail of infrequent syllables.

We substituted every space between words with the special token "<SEP>" and inserted at the beginning of each verse the token "<GO>". To make all verses the same lengths, we used the special character "<PAD>" to pad every terces to the length of 75 tokens. At the end of each verse we appended the symbol "<EOV> ", while at the end of each sentence "<EOS>".

2. Model

2.1 The transformer

In the last few years, transformers have become the go-to architecture for Natural Language Processing tasks such as text translation, speech-to-text and text generation.
Several models have been developed and deployed like Google's BERT (2018)[2] with about 110 million parameters, used to better understand user queries. Then Facebook's RoBERTa trained a similar model with much more data, outperforming BERT base[3]. Finally in these days (may 2020) Open AI's GPT-3 with 175 billion parameters affirmed itself as the state-of-the-art of NLP, capable of writing poetry and even code[4].

2.1.1 Transformer vs RNN

Recurrent architectures like long-short term memory and gated units have been firmly established as state of the art approaches in sequence modeling: processing one symbol at time, they generate 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 inherently sequential nature precludes parallelization within training examples, which becomes critical at longer sequence lengths. Another problem due to the length of the input sequence is to keep track of the long term dependencies among symbols: the unrolled representation of a RNN processing a very long sequence results in a deep neural network, leading to problems like vanishing/exploding gradients that are only partially corrected by the gated units.

In transformer models all the sequence is processed at the same time, modeling the dependencies among symbols without regard to their distance in the input or output sequences in constant time and in an easy parallelizable fashion.

The better long term dependencies modeling capability of the Transformer[5] is the main reason that convinced us to use it in poetry modeling, where keeping track of the rhythmic scheme of text is fundamental.

2.1.2 Structure

The core idea behind the Transformer model is *self-attention*, the ability to attend to different positions of the input sequence to compute a representation of that sequence. A transformer model handles variable-sized input using stacks of self-attention layers. This general architecture has a number of advantages:

- It makes no assumptions about the temporal/spatial relationships across the data.
- Layer outputs can be calculated in parallel, instead of a series like an RNN.
- Distant items can affect each other's output without passing through many RNN-steps.

The downsides of this architecture are:

- For a time-series, the output for a time-step is calculated from the *entire history* instead of only the inputs and current hidden-state. This *may* be less efficient.
- If the input *does* have a temporal/spatial relationship, like text, some **positional encoding** must be added or the model will effectively see a bag of words.

Most competitive neural sequence transduction models have an encoder-decoder structure. Here, the **encoder** maps an input sequence of symbol representations $(x_1, ..., x_n)$ to a sequence of continuous representations $(z_1, ..., z_n)$. Given z, the **decoder** then generates an output sequence $(y_1, ..., y_m)$ 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, shown in the left and right halves of Figure 1, respectively.

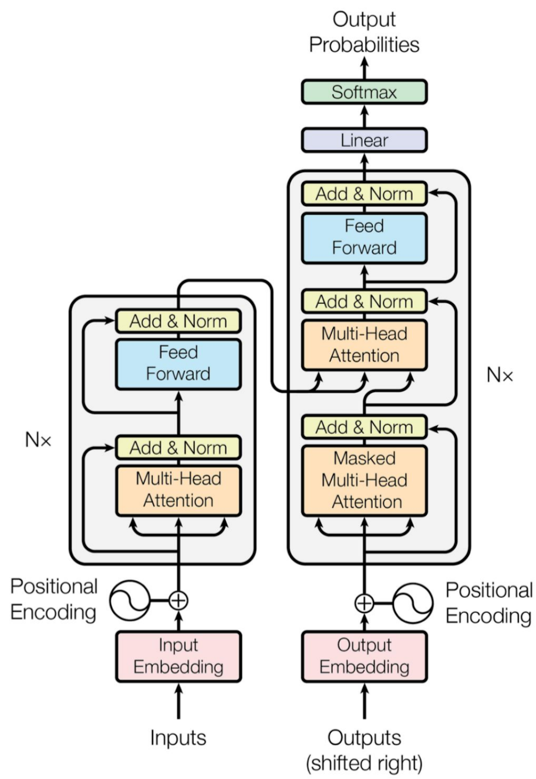


Figure 1: The Transformer - model architecture.

2.1.3 Embedding and positional encoding

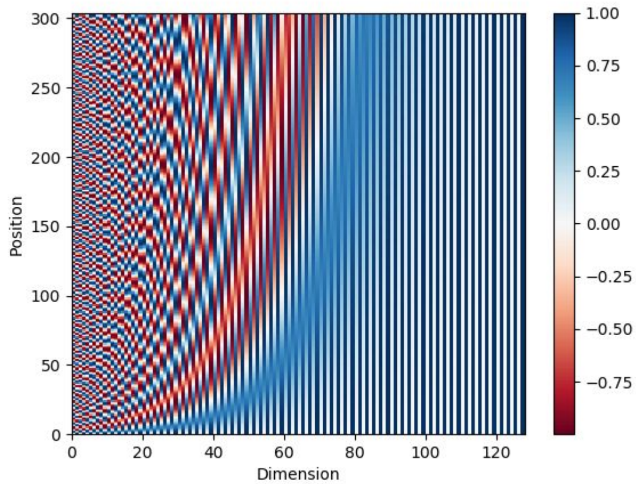
The input/output is a sequence of integers, each of them representing a syllable in the dictionary.

Firstly, the **discrete** symbols are embedded in a **continuous** vector space, the embedding, that captures some of the semantics of the input by placing semantically similar inputs close together in the embedding space.

Then the positional encoding is added in order to preserve information about the relative spatial location of the tokens in a sentence. The positional encodings have the same dimension as the embeddings, so that the two can be summed. In this work like in the original paper, we use sine and cosine functions of different frequencies that take into account that sentences could be of any length:

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

Where *pos* is the position and *i* is the dimension, it would allow the model to easily learn to attend by relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}.



Intuitively in first dimensions the positions are encoded by high frequencies sinusoids, decreasing gradually in higher dimensions. This improves the network ability to catch patterns that happen at different periodicities.

Beside the i/o sequences, a boolean *mask* is fed to the model in order to **ignore padding** symbols.

2.1.4 Encoder

The encoder is composed of a stack of N identical layers. Each layer has two sub-layers: the first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network.
We employ a **residual connection** and a **dropout layer** around each of the two sublayers against *overfitting*, followed by **layer normalization** in order to speed up the convergence during training.

2.1.5 Decoder

The decoder is also composed of a stack of N identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer (followed by a dropout layer), which performs multi-head attention over the output of the encoder stack.
Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization.
We also modify the self-attention sub-layer in the decoder stack with a **look ahead** *mask* to prevent positions from attending to subsequent positions. This masking, combined with the fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.

Finally the Transformer also includes a final Linear Layer followed by a **softmax** threshold function that takes the output of the decoder and generates the class predictions.

2.1.6 Multi-head attention

The core component that we find in both encoder and decoder is the multi-head attention[6].
An attention function can be described as mapping a query and a set of key-value pairs to an output,where the query, keys, values, and output are all vectors: the output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key. We compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values. In practice, we compute the attention function on a set of queries simultaneously, packed to a matrix Q. The keys and values are also packed together into matrices K and V.

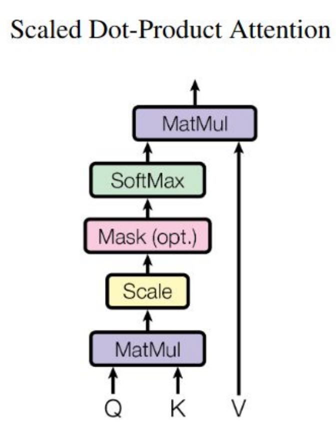
$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

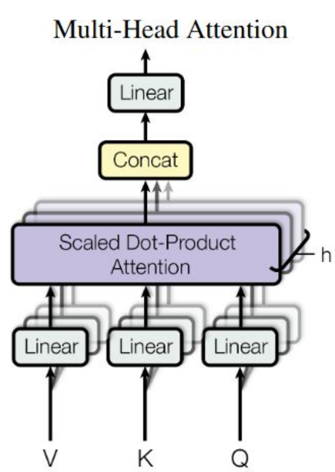
To be more clear, V is multiplied by the **attention weights** $a = softmax(\frac{QK^T}{\sqrt{d_k}})$.

These weights are defined by how each word in the sequence is influenced by other words in that same sequence. In other words, these weights measure how much K influences Q. The softmax function is used to distribute the weights between 0 and 1. The scaling factor is used because, if Q and K have same mean and variance, their product would have mean 0 and variance d_k so we use $\sqrt{d_k}$ to get a softer softmax.
The mask here is multiplied by a very small number here, so that it becomes 0 in the softmax output.

The Multi-Headed Attention block is a slightly modified version of the Attention mechanism. It consists of four part:

- Linear layers are split into **heads**
- Scaled dot-product attention
- Concatenation of heads
- Final linear layer





Instead of performing a single attention function with d_{model} -dimensional keys, values and queries, we linearly project the queries, keys and values h (number of heads) times with different, learned linear projections to d_k , d_k and d_v dimensions, respectively where $d_k = d_v = d_{model} / h$. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding d_v -dimensional output values. These are concatenated and once again projected, resulting in the final values.

Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this.

In addition to attention sub-layers, each of the layers in our encoder and decoder contains a fully connected feed-forward network, which is applied to each position separately and identically. This consists of two linear transformations with a ReLU activation in between, where the dimensionality are respectively d_{inner} and d_{model} .

2.2 Hyperparameters

Summing up, the model relies on a small set of hyperparameters, tuned in order to achieve the best performances in text generation:

- *N:* number of encoder/decoder layers
- d_{model} : dimension of Embedding and attention layers
- d_{inner} : dimension of Feed Forward inner layers
- *dr*: dropout rate
- *h:* number of attention heads
- *lr:* learning rate

The original paper used a custom learning rate that increased *lr* linearly for the first warmup_steps training steps, and decreased it after proportionally to the inverse square root of the step number.

We found most effective a **fixed** learning rate during the whole training, due to the smaller dimension of the dataset.

3. Training

3.1 Loss

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.

3.2 Optimizer

We used the **Adam**[7] optimizer with β1= 0.9, β2= 0.98 and *eps* = 10^{-9} and a fixed learning rate, tuned during several experiments.
The Adam optimizer is designed to combine the advantages of two popular methods: AdaGrad (Duchi et al., 2011), which works well with sparse gradients (uses per parameter adaptive learning rate), and RMSProp (Tieleman & Hinton, 2012), which works well in on-line and non-stationary settings (uses momentum).

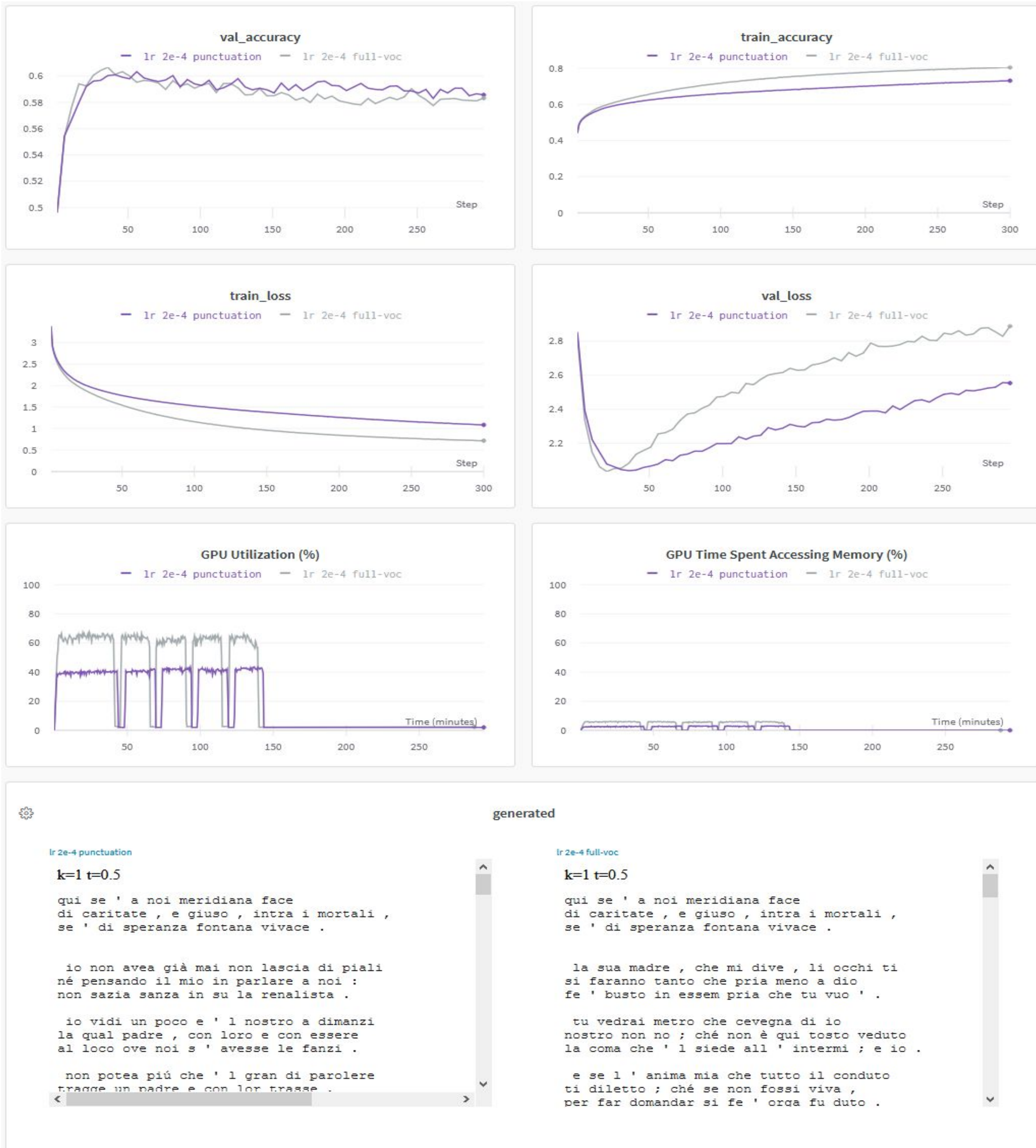
3.3 Setup

The model, written in python, was developed using the Keras framework on top of Tensorflow 2.0.
We trained the models on Google Colab Pro cloud, on a machine equipped with a GPU Nvidia P100 and 12GB of Ram.

3.4 Tracking sessions

An epoch consisted of 1154 batches and required around 20 seconds to be completed.
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.



4. Generation

The Transformer is an **auto-regressive** model, meaning that it will predict the next token based on the past sequence already generated (decoder input).
In order to generate a sample of text, we fed the decoder with an unseen starting sequence from the test set and then concatenating gradually to the input the token sampled according to the model prediction.

Since the model has been trained with batches of 4 terces, the decoder input is started with only the last terce generated (or the initial one) and then symbols are appended until reach the length of a batch. This process is iterated generating 3 terces each time until a full cantica (about 30 terces) is produced.
We implemented two different ways of sampling from the model prediction: greedy and topK.

4.1 Greedy

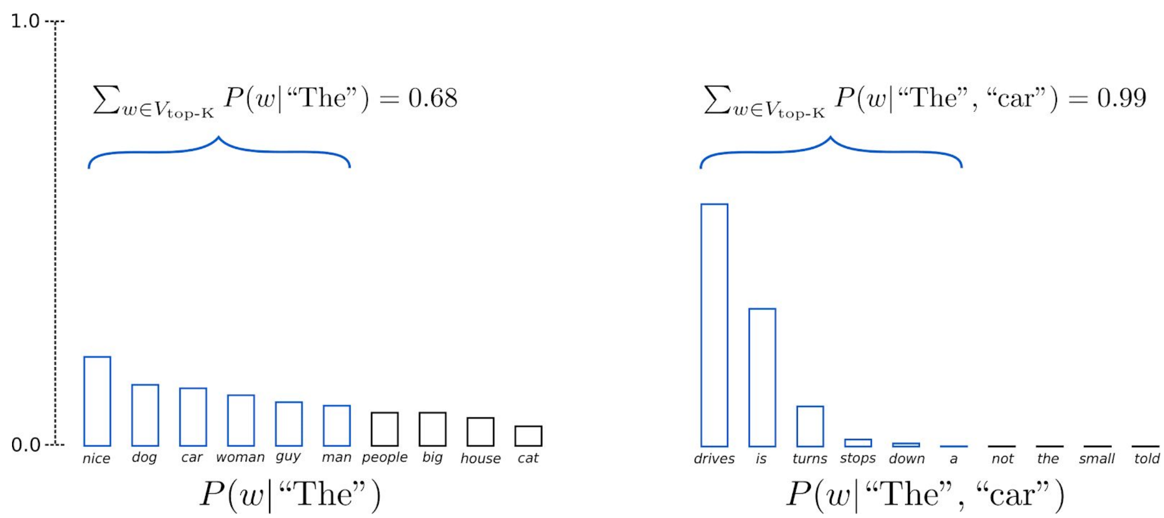
Greedy generation selects, for every token to be generated, the most probable one based on the predictions. Even though it seems very efficient and, theoretically, seems to be the most desirable approach, greedy search suffers two major drawback:

- Repetition: the model starts repeating itself in a short amount of time, leading to hard-to-read texts;
- Conditional Probability: the token with the highest probability at timestamp *t* may be followed by other tokens with low probability and be selected anyway, even though the token with the second-highest probability at *t* may be followed by tokens with higher probabilities, giving the sentence an higher overall probability;

A solution for this problem would be to use Beam Search[8]. It consists in searching a sequence of *n* tokens following the token to be chosen. The sequence with the highest overall probability will be selected. We implemented this method but the computational complexity of the algorithm resulted in >20 mins of generation time for a single tercet (with *num_beams = 5*) or with no valuable improvement over Greedy Search (with *num_beams = 2*). Plus, *num_beams = 2* means that the sequence is made of three tokens that most of the time compose a single word, so we would also lose the ability of Beam Search to improve the sense of the sentence.

4.2 TopK

In TopK Search the first most likely *K* classes are filtered from the rest. The probability mass is redistributed between only them. In the following figure, we can see how the probability mass in the first step covers for two-third the first six words. In the second step, the first six words are covered by almost the entire probability mass. 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 sentence where the rhythmic scheme must be preserved.



Finally we chose to use the **greedy** algorithm because it achieved experimental better results with respect to the TopK.

4.3 Text evaluation metrics

To evaluate the quality of the generated cantica we used several 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.

- **Rhymeness**: average compliance of each terce to the structure [xBx] BCB C; the single rhyme is evaluated by exponentially weight each character, starting from the end of the string and stopping to the first unmatched character.

- **Ngrams plagiarism**: represents the "inverse" proportion (1.0 stands for no plagiarism, 0.0 stands for total plagiarism) of identical n-grams, i.e. n subsequent words, found both in the generated text and in the original text, punctuation excluded.

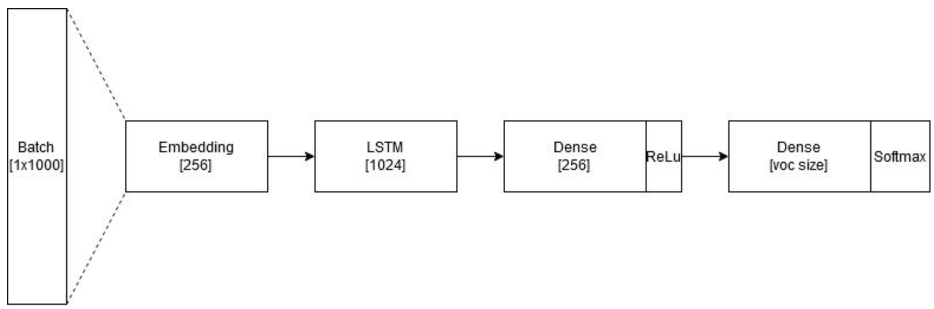
5. Experiments

5.1 RNN models

Initially we implemented several RNN-based models to perform the task, trying to take a different perspective on the problem each time.

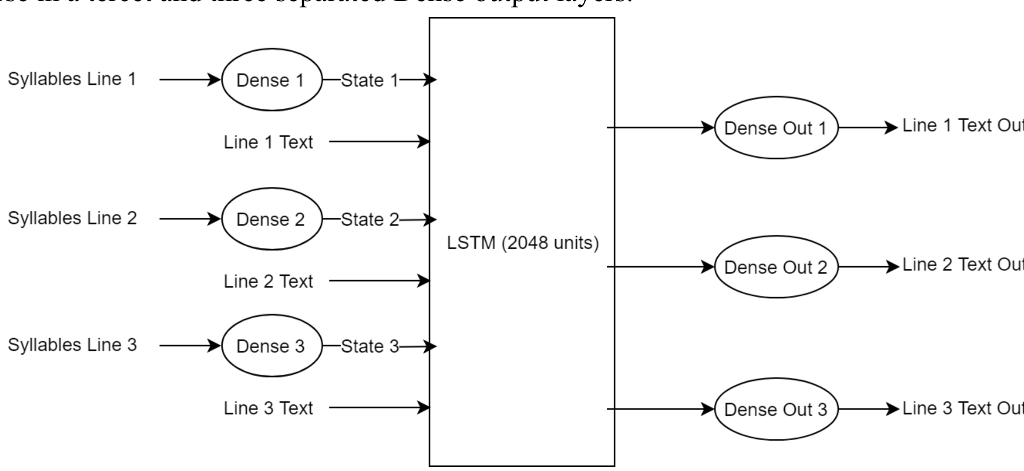
We used two different approaches to the syllable count problem, experimenting the same model using the syllabification system described for the transformer and a char-level division of the text.
All the experimental models used an LSTM layer at their core, while the input and output layers were different.

The first model, RNN char level, has an Embedding layer that receives a defined number of batches connected to the LSTM layer. The output of the LSTM is then passed through two more Dense layers before the token is generated.



This basic model provided the best results among all models based on LSTM from the syllable count point of view, while rhymes were basically absent. It is to be noted that this model is prone to plagiarism after a few epochs.
We experimented with the same network using syllables as input, but it provided no improvement on the rhyme pattern while worsening the metric and syllable count, probably due to the increased size of the vocabulary.

Then we experimented with a different architecture, using three separated Dense input layers, one for each verse in a tercet and three separated Dense output layers.



We hoped that this would improve over the metric of the single line and the recognition of the rhymes pattern. Again, using char-level batches of tercets the model produced good results about structure and syllable count, but we still could not obtain a consistent rhyme scheme. Using syllables as input did not provide any improvement on that front, while again significantly worsening the structureness and syllable count.

5.2 Hyperparameters tuning

Became aware of the limits of simple RNN architecture, instead of trying with a more complex one, we opted to experiment with a Transformer model, obtaining satisfying results.

In order to achieve an optimal tuning, several considerations had to be made about the hyperparameters.

5.2.1 Dimensionality

Due to the small size of the vocabulary, multiple tests showed that higher dimensions of the Embedding and Multi-Head layers were the cause of a heavy overfitting, that resulted in bad cantica generations. The same applies for the inner layer of the FF network, so we opted for a simpler network with respect to the original work that was intended for more complex tasks like machine translation.

5.2.2 Layer number

Concerning the number of layers, a lower value (resulting in a simpler model) showed a significatively improvement in train and validation performance (reaching in both 99% accuracy), but performing auto regression the model produced a terrible free text.

5.2.3 Dropout

Increasing the dropout rate, enhances the network ability to generalize so that in order to reach a certain accuracy more epochs are required. However even after more training time we didn't notice any particular improvement in validation accuracy so we decided to leave it unchanged with respect to the original paper.

5.2.4 Attention heads

The lower the number of attention heads the more the effect of averaging disturbs the token prediction, while from 4 onwards there wasn't sensible improvement: in fact once passed the optimal number of heads, only a subset of these will specialize and give the most of the attention weights while the others can be pruned.

5.2.5 Learning rate

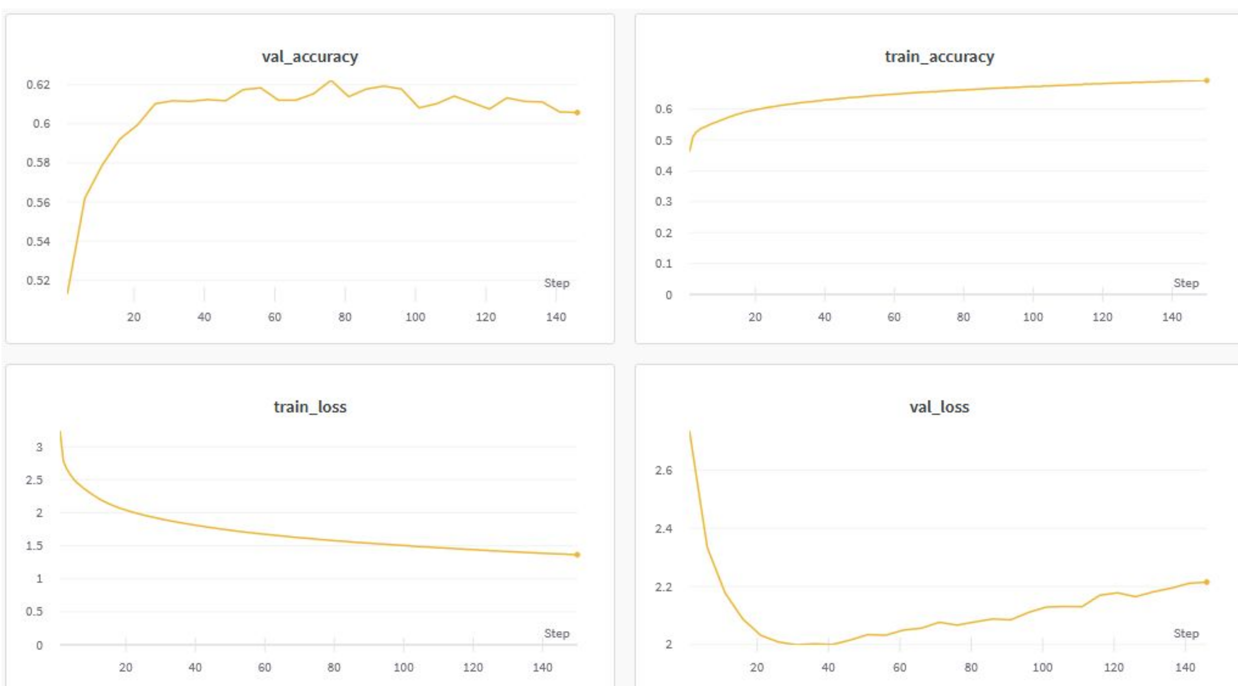
Like the case of using fewer layers, a lower learning rate led to the state where we had excellent validation accuracy but a terrible text production.

So we ended up with this optimal tuning of the hyperparameters:

4	N (number of encoder/decoder layers)
128	d_{model} (dimension of Embedding and attention layers)
256	d_{inner} (dimension of Feed Forward inner layers)
0.1	dr (dropout rate)
4	h (number of attention heads)
$2 \cdot 10^{-4}$	lr (learning rate)

6. Results

Once fine tuned the hyperparameters, we found the best performance after about **150 epochs**. Here are reported the metrics of our model during the training session:



While below there is the generated cantica using a greedy strategy:

qui se ' a noi meridiana face
di caritate e giuso intra i mortali
se ' di speranza fontana vivace

da questa parte che mai non si confili
e come quando da quella che fede
da ' l ciel che da sé non si dicigli

e ' l tempo di là giú su su sode
piú su per lo ' n ' ha l ' alto si rive
e quel che piú e piú non si difede

quinci si riceva e quinci e quinci
luce per esser per voce che ' l ciel vero
e quinci a me e quinci si rivolci
e la fava che tanto non si dimo
che la divina di quel che ' l suo modo
non ha da quel che la terra il dimodo

e ' l suo non ha lume a quel secondo
che si movea di quel che la fede
che si riceva il lume si diverdo

quinci si vide e quinci e secrede
le porta luci d ' un ' altra parte
che fece porta in sé di là si nasconde

cosí la nova luna e non si ' l regi
ma per veder sua madre e piú a ' l nido
e quella che si fa con le cibo

e quella terra che ' l ciel non si disermo
non si dicea per quel che si difece
la viva là giú di là giú simomo

cosí la risposta in alto si dice
si mosse e ' l tempo che perdevona
per me e per lo suo vostro voce

e se fosse in giuso in terra fue
non aspetto maraviglia e la fronte
se non temo e maestro fece i ' tre sue

non sarò maraviglia in terra porte
non si distava il tempo che fece
la nazion che la rona per sete

non si discese mai non si face
ma per seco di quel che ' n chi si scoloro
che si vage e pena fa d ' arte
e come si move in su la rive
si giva il nome che ' l suo modo e ' l lume
si faceva di là giú e dive doce

↓ [continue] ↓

la natura che la terra di lume
della roma della fine e di là superse
la mente che ' l ciel piú si diverse

cosí la perfezion non si rispose
la noma perfede alla sua famicia
che si fa ha noma di là giú discese

come si move e l ' altra rose
e l ' altro che la sabile e l ' altro disio
e di piú non fa farsi divese

e quel che per me cuna regigli
a cui la terra si move si difero
e da quel di là giú d ' un sospegli

e se la viva che ' n terra è piú belro
la piè di quel che di balenar vero
da cui la terra del suo re amoro

e come si le rose si diro
si leva in sé e si ricea si riga
si ricede e fece li e dimoro

cosí la luce di quel che si vaga
si di là dove ' l tempo si difese
dove i peccar non è piú egavaga

cosí la vista e di là su s ' innose
venendo li occhi e giudicarlo e ' l ciel disposto
cosí ne ' n quel che si dispose

e come si move in giusto affanno
si rivolse a beato e per li occhi e per li occhi e persone

reveni a pena che tanto si stende
di quella romaraviglia quanto si difese
di quel che si fece li occhi e ricorde

non si dicea maestro torse non si porse
ma perché di là giú per quel ciel vede
che si di là giú per esse non si porse

cosí la voce di là giú diverse
come di grave si remove gelito
e piú non però d ' un piú sospese

come l ' uno e l ' altro è piú divito
che si distende e ' l tempo di bano
e l ' altra vita e del suo regno

↓ [continue] ↓

e come a me se ' l ciel di pieno
di quel che si facea di padre di pietro
da beato di là su si fe ' l pieno

la natura che la nova e di fore
di quel che di là ave la mente
la mente e di sé e di retro amore

e come i cerchi li cerchi si parte
si divero in quel che ' l velo e a ' l mondo
ha in altra vita e diparte

e come i vostri vostri vochi e secondo
di quel che si fede e quindi si difese
da beato da terra non si dimove

The text evaluation functions reported the following scores:

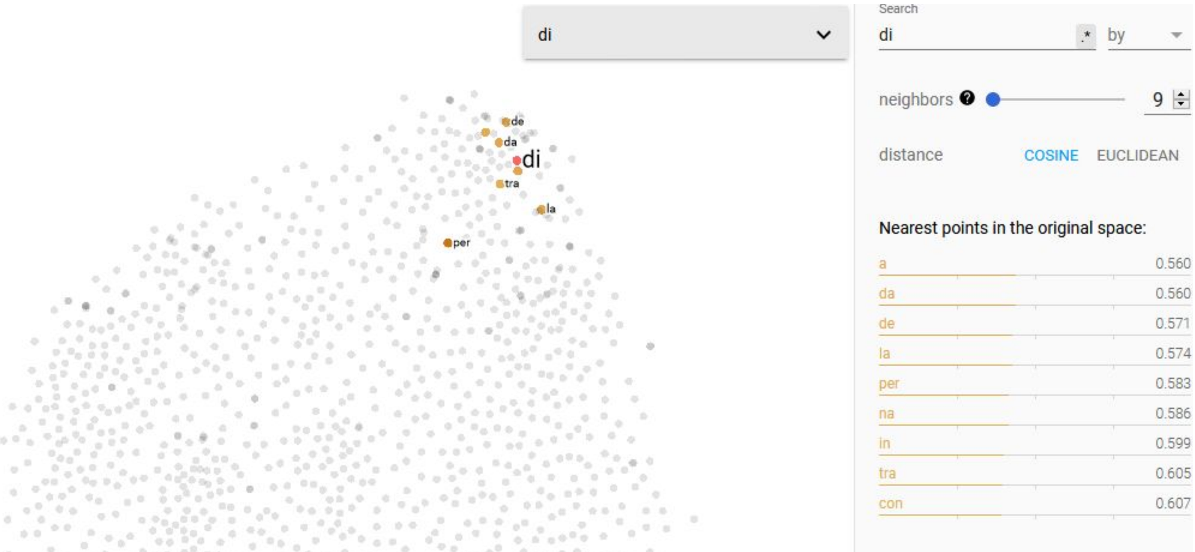
- Structuredness: 0.950
- Average hendecasyllabicness: 0.902
- Average rhymeness: 0.814
- Ngrams plagiarism: 0.943

6.1 Embedding space

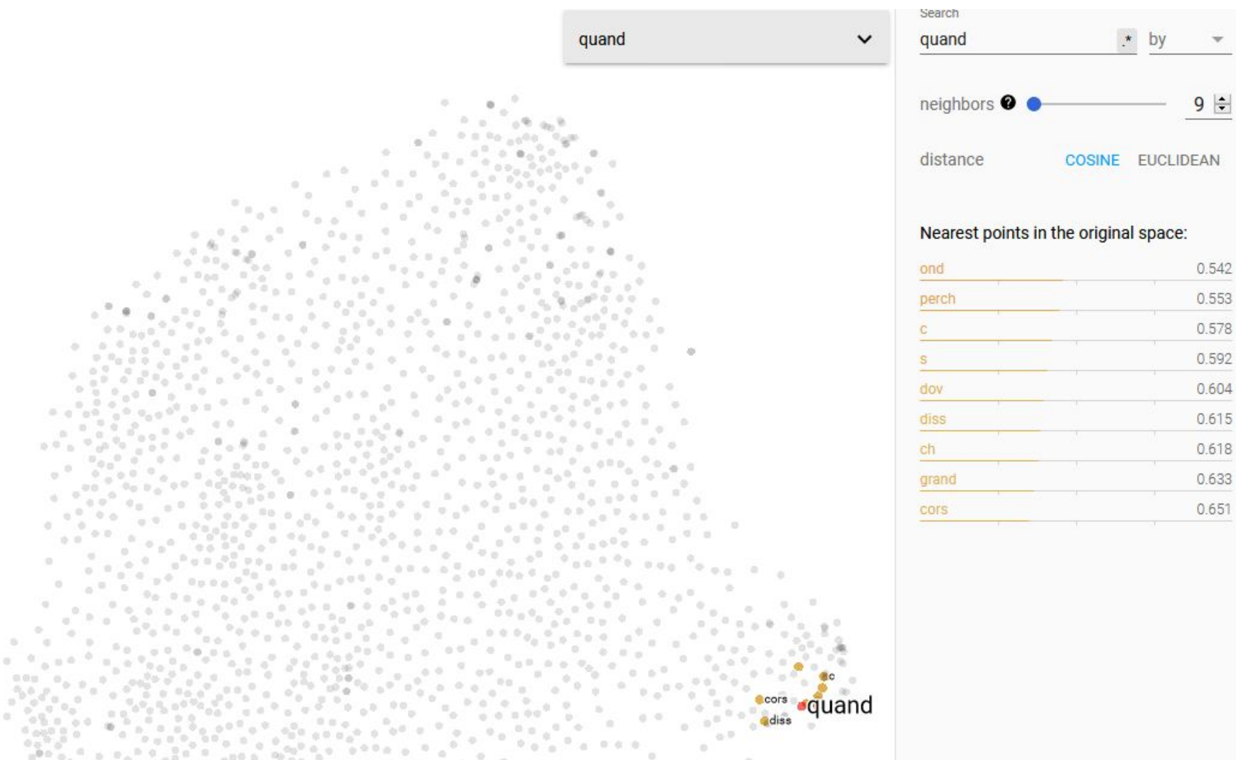
The learned representation of the decoder embedding consists in a 1800 x 128 matrix, where each of the vocabulary's tokens lie in a 128-dimensional space. In order to represent them in a 3 dimensions plot the matrix has been processed using PCA, capturing 9.3% of the total variance along the top 3 principal components.

Looking at the nearest points of a syllable according the cosine distance, we could explore the semantic representation learned by the network, finding that it catched some interesting feature of the language:
(points have been represented in a 2D space using TSE technique)

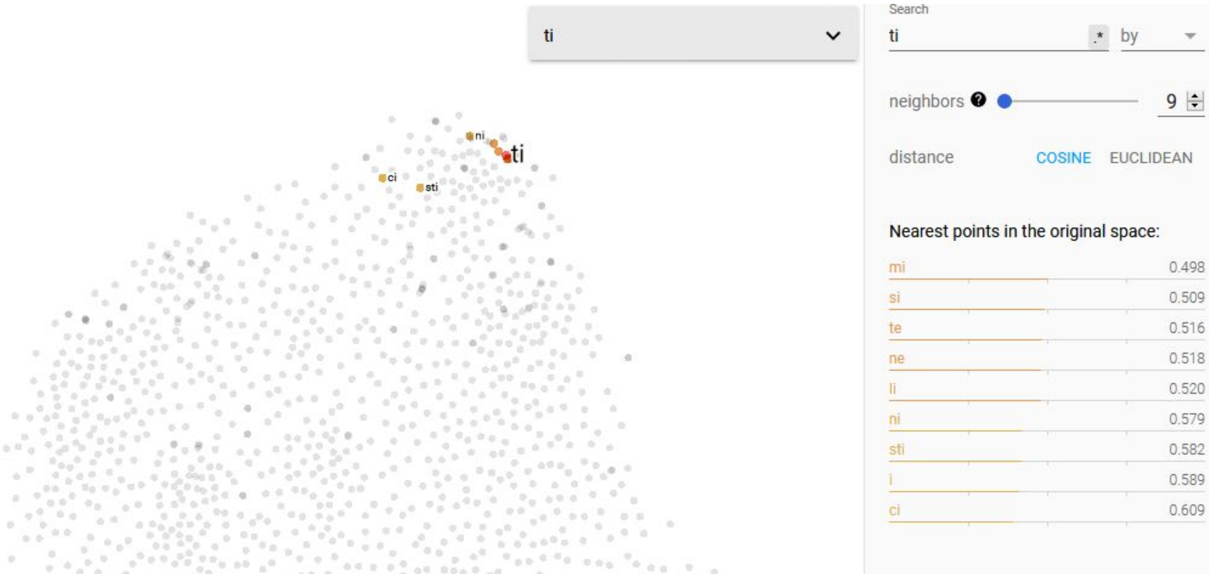
Prepositions: *di, a, da, in, con, su, per, tra, fra*



Interrogative adverbs: *chi, dove, quando, perchè, come*



Pronominal particle: *mi, ti, si, ci, vi, si*



7. References

[1] Zugarini, Andrea, et al. "Neural Poetry: Learning to Generate Poems Using Syllables." *ArXiv.org*, 24 Sept. 2019, arxiv.org/abs/1908.08861.

[2] Devlin, Jacob, et al. "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding." *ArXiv.org*, 24 May 2019, arxiv.org/abs/1810.04805.

[3] Liu, Yinhan, et al. "RoBERTa: A Robustly Optimized BERT Pretraining Approach." *ArXiv.org*, 26 July 2019, arxiv.org/abs/1907.11692.

[4] Brown, Tom B., et al. "Language Models Are Few-Shot Learners." *ArXiv.org*, 22 July 2020, arxiv.org/abs/2005.14165.

[5] Vaswani, Ashish, et al. "Attention Is All You Need." *ArXiv.org*, 6 Dec. 2017, arxiv.org/abs/1706.03762.

[6] Voita, Elena, et al. "Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned." *ArXiv.org*, 7 June 2019, arxiv.org/abs/1905.09418.

[7] Kingma, Diederik P., and Jimmy Ba. "Adam: A Method for Stochastic Optimization." *ArXiv.org*, 30

Jan. 2017, arxiv.org/abs/1412.6980.

[8] Vijayakumar, Ashwin K, et al. “Diverse Beam Search: Decoding Diverse Solutions from Neural Sequence Models.” *ArXiv.org*, 22 Oct. 2018, arxiv.org/abs/1610.02424.