Deep Learning Project: The Deep Comedy

Luca Rispoli, Andrea Lavista

3 marzo 2021

Indice

1 Introduction **2**

2 Implementation **3**
2.1 Pre-Processing 3
2.1.1 Special tokens 3
2.1.2 Tokenization 3
2.2 Methodologies 4
2.3 Plain LSTM architectures 4
2.3.1 Generating text at word level 6
2.3.2 Generating text at character level 7
2.4 Advanced LSTM architecture 8
2.4.1 FastText as word embedding 8
2.4.2 Custom loss function 9
2.4.3 Combining the two networks 9
2.5 Sequence to Sequence 11
2.5.1 Sequence to Sequence with attention 11
2.5.2 Main hyperparameters 12
2.5.3 Word-based Seq2Seq 13
2.5.4 Char-based Seq2Seq 14
2.5.5 Beam search 14
2.6 Models with Rhyme Encoding 16
2.7 Conclusions 17

Capitolo 1

Introduction

The project's goal is to build an effective neural network whose aim is to generate text with the same style used by Dante Alighieri in his Divina Commedia.
Dante's style is mainly recognized by the following two features:

- **Hendecasyllables:** verses in which the main accent falls upon the tenth syllabe.

- **Terza Rima:** a special case of Chain Rhyme[1], composed by tercets that follow the rhyming pattern a-b-a, b-c-b, c-d-c and so on.

The proposed models have been projected and built using the Keras library as foundation, this choice was made considering its extensive range of features and ease of usage.
Before starting our project, we made a research in order to find which were the algorithms that constitute the current state of the art for text generation tasks, in the end, we opted for the use of LSTMs, both as a stand-alone architecture and inside a Sequence to Sequence model, then we implemented several tweaks, such as attention and beam search.
We have developed different architectures and built several models, during the last phase of our project, we compared them in order to find out which architecture yielded the best results.

[1] `https://en.wikipedia.org/wiki/Chain_rhyme`

Capitolo 2

Implementation

The following models have all been trained using the Divina Commedia as dataset. The original text of the book has beeen retrieved from the following source:
`https://github.com/dlang/druntime/blob/master/benchmark/extra-files/dante.txt`

2.1 Pre-Processing

2.1.1 Special tokens

The dataset has been modified several times in order to draw attention to different features of the text such as rhymes, tercets and so on.
The first pre-processing step, commmon to all models, was to insert tags inside the dataset, in order to make it easier for the model to process the Divina Commedia's structures, the added tags were:

- **Start**: Delimits the start of a canto
- **End**: Delimits the end of a canto
- **Verse**: Delimits the end of a verse
- **Tercet**: Delimits the end of a Tercet

2.1.2 Tokenization

Prior to the training phase, the dataset has been transformed in order to be represented as sequence of tokens using the **Tokenizer** class provided by the

Keras API.
The tokenizer, once fitted to the text, assigns an index to every word (or character), reserving one spot for the padding token, which has been used to pad all the sequences to a fixed length.

2.2 Methodologies

The training phase makes use of two auxiliary functions:
EarlyStopping: This function automatically stops the training phase if there has been a certain number of epochs in which there has not been any increases in the value chosen as objective function
ModelCheckpoint: This function allows to automatically save the models that manage to achieve the best objective function value.

2.3 Plain LSTM architectures

Long short-term memory (LSTM) layers are currently widely used in sequence related tasks such as translation, text generation and classification thanks to their capability of processing and learning long term dependencies.
As first model we built a simple LSTM architecture, in order to obtain a baseline result and explore and familiarize with the features of the layer.
The model's pipeline works as such: tokenized words are fed into a trainable embedding layer whose aim is to learn an optimal mapping from tokens to word-vectors, then the vectors are processed through two LSTM layers and the final output is generated with a dense layer that makes use of softmax as activation function, dropout layers have been added after the LSTMs in order to avoid over-fitting.

There are a wide number of parameters to be tuned, both related to the training phase and the model's architecture, the parameters that have more impact on the final results were the following:

- **Sequence Length**: how long must be the sequences that are fed to the model during the training phase, we tried various values, the one that seemed to give the best results is 50, as it is wide enough to include a whole tercet without adding irrelevant information inside the context.

- **Stride**: related to the length of the sequences, the stride parameter indicates the value of the offset between subsequent sequence windows, we opted for 1, as we did not want to skip any word or character inside the dataset.
- **Embedding Dimension**: the embedding layer requires as input the dimension of the word vectors generated, a large value allows the model to perform a mapping into a bigger space, at the cost of performances. A dimension of 50 offered a satisfactory tradeoff between the quality of the mapping and the training time required.
- **LSTM**: among other parameters, the most important parameters of the layer were the number of units and the dropout value, both were inferred empirically and have, respectively, a value of 256 and 20%.

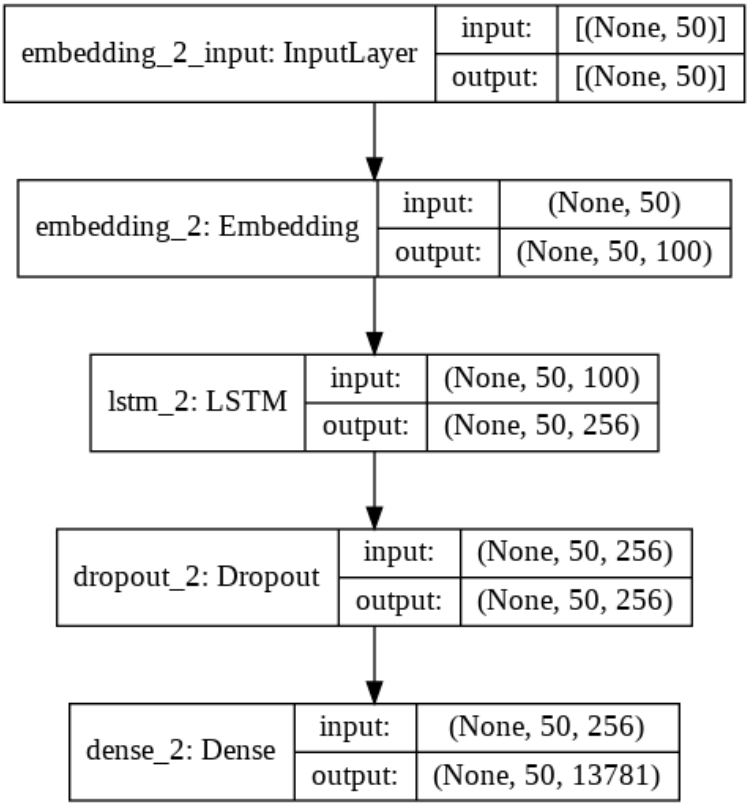


Figura 2.1: First LSTM architecture

2.3.1 Generating text at word level

The network works at **word level** meaning that it expects as input sequences of tokenized words.
The dataset has then been split accordingly into sequences of the chosen length, the target value of every sequence is the word that comes right after the end of the sequence, so if the input sequence given to the network is "Nel mezzo del cammin di", then the correct prediction would be "nostra". We tested the architecture with different values in order to discover the optimal sequence length, which we found to be 50.
The network was then trained for 100 epochs, parameters were updated according to the categorical crossentropy loss.
In order to generate text, we created a random initial sequence containing a starting token, then the subsequent inputs are made by appending to the sequence the predictions produced by the network at every iteration.
An issue we encountered is that if one always selects the word which is most probable according to the softmax output, then the network tends to fall into a loop, in order to avoid this behaviour, we introduced the concept of **temperature**.
As such, chosen words are selected stochastically according to a probability distribution generated with the softmax output values reweighted by a constant called temperature, a high temperature raises the network's confidence, whereas a lower temperature increases the entropy, making the network's outputs less predictable.
This is an example of the text generated by the network:

riva la per ogne dietro dinanzi si non
suso amore disse ti ascoltar miei e
strida lui giunti quel e suso e

senno un rispuose cielo in parte tua
punto suo uscir perché quando con
angoscia di o mi tenne che mena

Evaluation

As shown in the example, the network manages to generate text that complies with the hendecasyllable and structureness rules, but reaches poor rhymeness scores.

Plagiarism	Structure	Hendecasyllable	Rhyme
0.97	0.84	0.79	0.1

Tabella 2.1: Scores of the baseline word-based LSTM

2.3.2 Generating text at character level

This network works at character level, so it generates one character at each timestep.
The original dataset has been splitted into sequences of characters of fixed length, at each timestep, these sequences are used as input for the network, whereas the output is the character that comes after the sequence.
The subsequent phases are very similar to the earlier model, though we had to decrease the number of LSMT cells inside the network, since the number of sequences is much higher and thus the training took a significant amount of time.
An example of the text generated by the model:

poi noi questa mrondenza più ragiona
o suo e poi fassuccì non come foco
cantendarnai come 'l sol si f'sangoss' 'l quali

ha marafa cenerlo riguardar lo
rispio l'avanti d'un disio sai varco
camveterai m'addireli avea trappri

Evaluation

The baseline LSTM network is not powerful enough to work with characters only, many of the generated words of the output text are invented and the network tends to converge to the same sequence after a certain number of iterations, the temperature algorithm solves this convergency issue but the output becomes excessively unpredictable, therefore, this model has been discarded.

Plagiarism	Structure	Hendecasyllable	Rhyme
1.0	0.89	0.77	0.07

Tabella 2.2: Scores of the baseline char-based LSTM

2.4 Advanced LSTM architecture

Having experimented with LSTMs, we tried to improve the baseline results by adding tweaks to the model, most of our efforts were focused on finding ways to improve the rhymeness scores, although the presented adjustments prompted a rise of several other scores, like structuredness and plagiarism.

2.4.1 FastText as word embedding

FastText is a library for text classification and representation, it's main use is to build models that embed words into a n-dimensional space.
We built a word-level text generation network that exploits fasttext in order to generate the initial weights of the embedding layer; the reason behind this choice was the possibility offered by the library to build a model taking into account the n-grams that shape the words. The aim was to check whether or not the n-gram feature is powerful enough to capture the relation between words that rhyme.
The library works according to a large number of parameters, among the ones that influenced the model the most are:

- **Learning rate**
- **N-Gram length**
- **Window Size**
- **Embedding Dimension**

Initially , the model built using fasttext gave positive results, words that rhymed toghether were mapped closer into the embedding space, for example, the word "ardito" had as neighbours "dito","sito","udito" and so on, again, the word "ossa" had "rossa", "mossa","grossa" as closest words.
The model's trained weights were then used as initial weights for the embedding matrix used as input layer of the network.

Evaluation

The results were pretty similar to the baseline LSTM, even though there has been an overall increase to all the scores, especially the rhyming one, they were not satisfying enough.

Plagiarism	Structure	Hendecasyllable	Rhyme
0.87	0.98	0.84	0.12

Tabella 2.3: Scores of the fasttext word-based LSTM

2.4.2 Custom loss function

In order to improve the rhyming capacity of our model, we designed a custom loss function that better grasps the structure of syllables inside a given word by assigning to each prediction a score (i.e. error) that is higher the more letters are out of rhyme. This function was then used to train a LSTM based network, using a dataset generated ad-hoc containing an arbitrary number of chaining rhymes, the result was a network that, given as input the last three words of the last three verses, outputs a word that respects the rhyme. The function was tested with a word based model, a character based model, and both at the same time, in the end, the one that gave the best results was the word based model.

2.4.3 Combining the two networks

Having a neural network that performs well when generating tercets and one network that specializes on rhymes, we decided to combine them: the first network will generate the verse whereas the second will take care of generating the last word, making sure it respects the rhyming pattern.
The main problem is that the rhyming network, not having any clues about hendecasyllables, might corrupt the structuredness of the text by generating a word that is too short or too long.
We solved this issue by generating verses in reverse: the last word of the verse is the first one that is predicted, then that word is fed as input to the auxiliary network, whose job is to generate the rest of the verse.

Evaluation

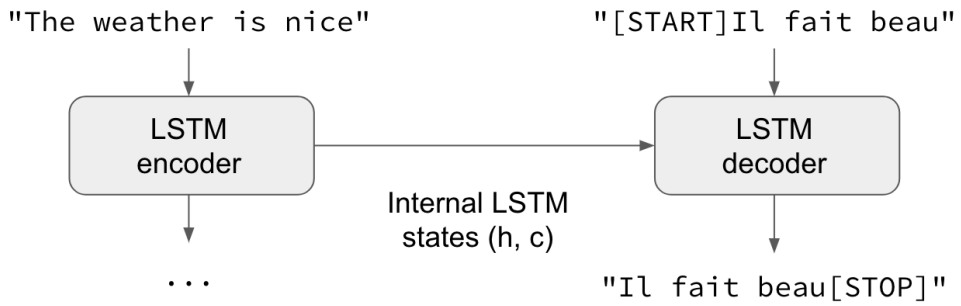
The results improved a lot, the two networks, working together, were able to generate a canto that reflected Dante's style, both with respect to hendecasyllable and rhymes, without copying the text of the original Divina Commedia.

Plagiarism	Structure	Hendecasyllable	Rhyme
0.99	0.98	0.87	0.90

Tabella 2.4: Scores of the Rhyming+Tercet networks

2.5 Sequence to Sequence

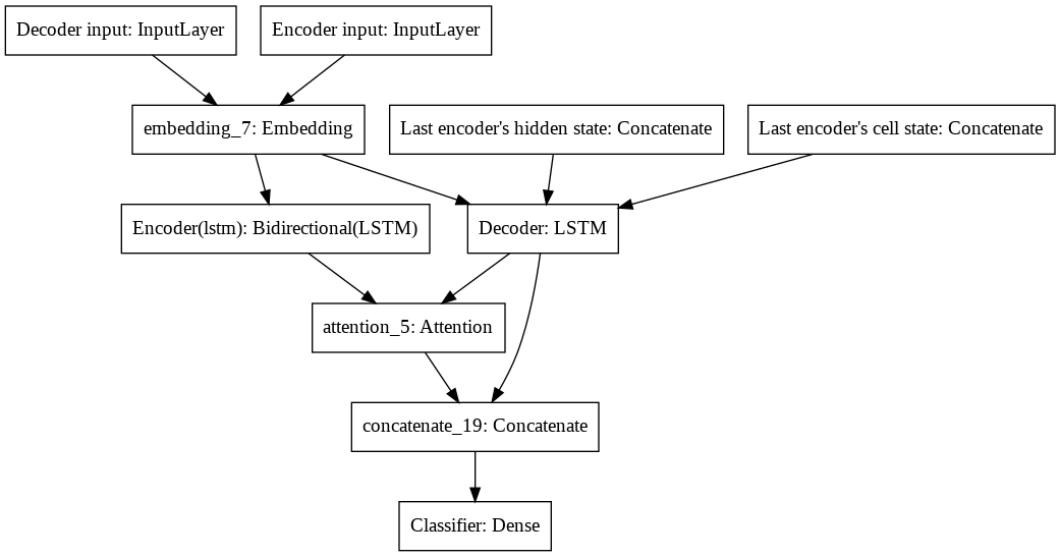
Another attempt to address this problem has been done using the Sequence to Sequence architecture, also called Seq2Seq. This architecture is often used for translation tasks (but also for question answering, chatbots, etc.) and it's composed of an encoder and a decoder.
For example, in the translation task the encoder receives the sentence that has to be translated and it produces as output an abstract representation of that sentence that is passed to the decoder. Then the decoder, which has a view upon all the sentence, can more easily do the translation and produce the ouput sentence, token by token.



In our case, we passed to the encoder the previous context, that are the previous verses; we tried passing at least 3 verses because this is the minimum number of verses to be known in order to understand which rhyme to generate.

2.5.1 Sequence to Sequence with attention

We combined this architecture with the attention mechanism, often used with the Seq2Seq models. The attention allows to focus, at each time step, on the most important part of the input and it consists in applying some weights to all the output hidden states of the encoder, enlighting what's more important for the generation of the single token. Below we present the scheme of the architecture.



The input of the encoder and the input of the decoder are passed to the embedding layer, which provides a vector representation of the tokens.
Then the encoder receives its inputs and computes all its outputs; the last hidden state and the last cell state of the encoder will be used as the initial state of the decoder.
Then also the decoder processes its input and produces the outputs that, together with the encoder's outputs, will be fed to the attention mechanism.
At the end the decoder's output is concatenated with the output of the attention step and received by a dense layer to perform the classification of the token.

2.5.2 Main hyperparameters

During the training of this neural network we tried different configurations, exploring the possible values to assign to the hyperparameters. The main hyperparameters involved are:

- **length of the previous context**: the encoder receives as input a fixed number of verses as previous context. This number must be at least 3 to let the neural network understand the rhyme pattern. Trying also other values we identified 3 as the best values.

- **embedding dimension**: the embedding dimension is related to the quality of the representation of the tokens. In the case of the word-based model we chose a dimension of 100, while in the case of the char based we chose 50.

- **LSTM hyperparameters**: these parameters regarding the number of layers, the number of neurons, the dropout and the others parameteres of recurrent neural networks. We reached the best results with a bidirectional encoder (with 128 units) and a single layer decoder (with 256 units); then we also tried different values for dropout and recurrent dropout.

2.5.3 Word-based Seq2Seq

The word-based Seq2Seq model is trained receiving as input the text where each token is a word. The encoder receives as input the previous three verses to the current verse to be predicted, the decoder receives as input the current verse preceded by a special token indicating the beginning of the line (<bol>) and the target output is the current verse to followed by a special token indicating the end of the line (<eol>). Below we show some of the text generated with this network, trained for 80 epochs with dropout and recurrent dropout equal to 0.2, to avoid overfitting.

ed elli a me se tu vuo' ch'i' ti sovvegna
dimmi chi fosti e s'io non ti disbrigo
al fondo de la ghiaccia ir mi convegna

rispuose adunque i' son frate alberigo
i' son quel da quel che da le frutta porte
che qui riprendo dattero per figo

The text generated with this configuration is good from the point of view of the structure, but only sometimes respects the rhyme pattern. We tried to see how a longer training affects the results and we discovered that, although the network eventually generated text following the rhyme pattern, the plagiarism score was very low, indicating that this was due to overfitting.

Evaluation

Plagiarism	Structure	Hendecasyllable	Rhyme
0.917	0.986	0.898	0.340

Tabella 2.5: Scores of the word-based Seq2Seq

2.5.4 Char-based Seq2Seq

The input for the char-based Seq2Seq follows the same rules of the word-based, but here the tokens are the characters. Here's a little part of the generated text with this model, trained for 120 epochs with dropout and recurrent dropout equal to 0.2.

così l'anima che per la sua vita
di quella che con le sue parole strada
che di là dal mondo del mondo strita

che l'altra vista di là dal mal sada
e per lo suo passo del ciel che siene
che non si può con la vista si scada

In this case instead the rhyme score is really high, proving that the neural network is capable of understanding the concept of the rhyme; this is because having control of the single characters leads to the possibility of focusing on how the characters are concatenated in the last part of the verses and individuate the rhyme patterns. On the other side, the disadvantadge is that sometimes, to force a rhyme, the neural network generates a word that doesn't exist and the text becomes a little repetitive.

Evaluation

Plagiarism	Structure	Hendecasyllable	Rhyme
0.914	0.996	0.953	0.962

Tabella 2.6: Scores of the char-based Seq2Seq

2.5.5 Beam search

Beam search is a search algorithm often used in combination with Seq2Seq models. Instead of taking as output for each time step the token with the highest probability, this algorithm explores more paths, taking at each time step the best k paths. At the end, the possible outcomes are evaluated with a score, which is the sum of the log probability for each token, divided by the number of token and the path with the highest score is the one chosen. In our case the path ends with the end of the verse and we used beam search

with both the models, word-based and char-base. In the word-based model the results weren't good because the quality of the structure of the text generated was lower, indeed the token separating the tercets weren't picked correctly, and also the percentage of corrected rhymes was lower. In the char-based model the performance was very compromised because the model loses the capability of generating real words.

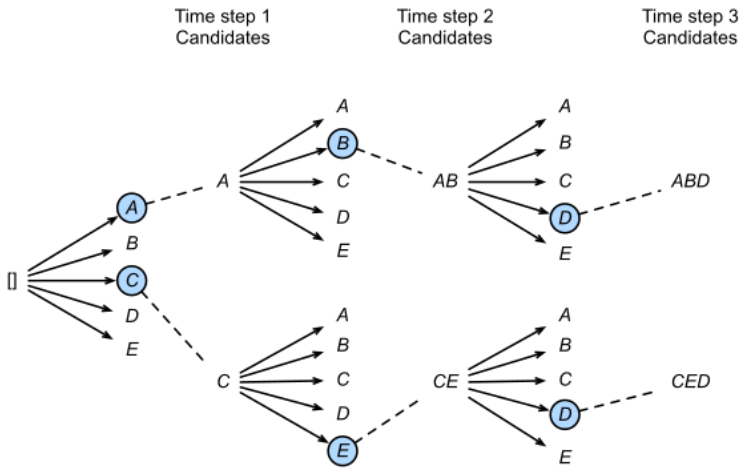


Figura 2.2: Example of beam search

2.6 Models with Rhyme Encoding

After we have tried the previous models we noticed that the word-based models were more reliable in the generation of the text (better structure and absence of the problem of generating non-existent words), but they were less prone to generate verses in rhyme. So we thought of ways to give, together with the word, some information about its rhyme.
Our idea consisted in creating a neural network that is able to classify rhyme. We collected all the words of the Divina Commedia and for each of them individuating its class of rhyme (from the vowel with the accent until the end). So, after composing the dataset in this way, we trained a simple neural network formed of an input layer, an embedding layer and a dense layer in order to let the embedding layer create a representation of the word based on its rhyme.
Then we repeated the training by adding a dense layer after the embedding working as an encoder (with 10 neurons) because we wanted to reduce the length of the word representation, indeed to 10. This is a schema that shows the structure of this neural network.

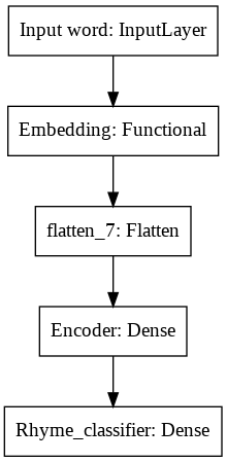


Figura 2.3: Final structure of the rhyme encoding model

Here's some numbers: this network has an accuracy of 84,3% and a f1-score of 57,3% in the test set.
The idea then was to export the first part of the network (embedding + encoding) and use it as a word embeddings technique that takes as input a

word and gives as output a representation of the word that carries information related to rhymes.
To see if that aim was reached we performed this test: we took pairs of words and calculated the similarities among their representations. Taking 100 pairs of words that rhyme, their average similarity was 0.739 and the 88% of them had a similarity above 0.3. Instead taking 1000 pairs of words that don't rhyme, their average similarity was -0.009 and 83,2% of them had a similarity below 0.3.
Anyway, despite these promising results, after inserting this encoder in the word-based models (both the baseline LSTM and the Seq2Seq), concatenating the output of the embedding layer with the output of this encoding concerning rhyme, the results of those models didn't improve.

2.7 Conclusions

We experimented with a broad number of models, in the end, the one that performed better, were the double LSTM approach and the Sequence2Sequence with attention based on characters.
Seq2Seq model is the one that achieves higher overall scores, but sometimes the model generates words that are made up, this is because the model works with characters and not sequences of words.
Possible improvements to the network might be aroused by experimenting with other kind of layers, such as **transformers**.