Music Generation using LSTM

Fabrizio Rocco[1], Roberto Colangelo[1] and Davide Rosatelli[1]

[1] *Luiss Guido Carli*

Abstract— Music generation is probably one of the most interesting applications of Neural Networks and the modern Artificial Intelligence fields. The main problem is dealing with the preparation of input data to train the model. At the beginning we've gathered 98 MIDI files from some famous Mozart,Beethoven and Chopin compositions. We took care of choosing data that belong to the same musical stamp and that share a common musical arrangement. Then we parsed these files and decomposed them into musical objects, based on notes and chords. Then we started working on the model, and we proposed different ways to solve this task, having as common factor the LSTM. These layers have been properly combined together with other regularization techniques in order to improve the accuracy of our models. At the end, we have created a proper pipeline to post-process our note sequence and to convert the output into a WAV file that can be listened by the final user. The result shows that our model works quite well and is able to produce different melodies.

Keywords— Music Generation, LSTM, Recurrent Neural Network, Artificial Intelligence.

I. Introduction

Ever since the dawn of time, music has always been one of the most characterizing aspects of the mankind ,that, over the time, has gathered people in moments of joy and lightheartedness. Creating a flawless and original melody has always been a struggle for human beings, so, given our knowledge of neural networks and our outstanding love for music, we have decided to entrust the task of make a machine able to generate music. We have therefore decided to implement a model capable of taking as input many classical music files from Beethoven, Mozart and Chopin and generate a composition made by the sequences of notes inside these ones. To successfully succeed in our task we exploited a specific kind of recurrent neural network, the LSTM (Long-short Term Memory). A Recurrent neural network creates an internal state of the network which allows it to exhibit dynamic temporal behavior. A LSTM, instead, expands this concept by introducing long-term memory into recurrent neural networks and learn how to deal with long term dependencies. In order to build our model we could have implemented other kinds of recurrent neural networks instead of LSTMs such as GRUs, but their excessively simplified structure, with a single state vector and a single gate controller, would have performed worse, so in the end we've chosen to build our model using LSTMs. What distinguishes an LSTM from a GRU is essentially the fact that it has 2 state vectors and GRU control the flows of information without using a memory unit. The fact that an LSTM has the capability of remembering information for a long period of time as the model runs, is due to the fact that the long-term state results to be quite appropriate not only for processing single data points (such as images), but also entire sequences of data (such as music converted files and videos) ,therefore it results perfectly fitted for our purposes. We have also tried to add BatchNorm and Dropout layers to regularize the model and to prevent overfitting. Eventually, in the last model we proposed, we have added an Attention layer that accesses the past-states of the encoder and handles the problem of giving importance even to the older states.

II. Methodology

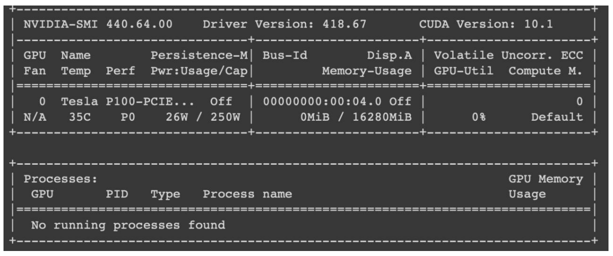
Our research started by analyzing all the literature regarding the problem and all the possible architectures to solve this task. One work among the others caught our attention. A project developed in 2016 by Google DeepMind called WaveNet [1]. This project is able to generate human voice and sound more natural than the best existing Text-to-Speech systems. It's based on causal convolutions for audio and masked convolutions techniques for images.[2] We realized that to compute something as close as possible to the State of the Art proposed by Google, we needed more than a better knowledge of the subject. Another massive project is the one proposed by Douglas Eck and his team in Tensor Flow, called Magenta[3]. It's a library which aim is to help to generate art and music. We decided to look for another automated music generation technique and our curiosity brings us to LSTMs. In the following sections we propose our best performing models.

a. Technical Challenges

The main problems we had have can be summarized in two concept spheres: data format and computational power. There's a distinction to make to whom the final user is. Is maybe a computer? Or, as in our case, is a human being? Exactly for this reason we handled a huge processing part which starts from decomposing our file to convert it into a WAV file to make the listening more human friendly, rather than the MIDI format of the input file. We used MIDI files for two reasons: first it makes possible to store many musical info, such as pitchs, into its metadata. Something which can be difficult to assess for a traditional MP3 or WAV file. The second reason behind this specific format is the large availability of files on internet. MIDI, or Musical Instrument Digital Interface, is a standard protocol for the interchange of musical information between electronic instruments: a MIDI file (with file extension '.mid' or '.midi') does not contains audio waves like other formats, but instructions on how to recreate music, like notes to be played, when they are to be played, loudness, pitch and so on. These files are very popular for creating and playing music, since it is quite easy to change performance of songs by changing, adding or removing the information that the file contain. Regarding computational power we had have some practical issue. Since our computers have not so good GPUs, we were forced to use a Google Colab notebook with an integrated GPU. Doing so we were able to partially improve our performances during models' training and to handle the team working in a better way.

III. Implementation

Google Colab Pro gave us the opportunity to use an Nvidia Tesla P100-PCi HBM2 16GB.



```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.64.00    Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   35C    P0    26W / 250W |      0MiB / 16280MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Fig. 1: GPU Info

a. Libraries

Keras [4]is an high-level API for building and training deep learning models. It's been built on TensorFlow, Theano and CNTK backends in order to provide a more direct and easy approach to build a deep learning model. We have implemented it with the TensorFlow backend.

TensorBoard[5] is an extension made by TensorFlow which provides a suite of visualization tools to make it easier to understand, debug, and optimize models.

Music21 [6] is an object-oriented toolkit for analyzing and transforming music in symbolic forms. The modular approach of the project allows musicians and researchers to write simple scripts to explore various musical aspects. It's been developed by MIT research department in 2008. We use it in both the processing parts as a way to connect the raw MIDI file to the network architecture.

IV. Data Gathering

We decided to use 98 MIDI files of Mozart, Beethoven and Chopin. All these files are parts of piano-solo compositions we found online [12]. The choice of these artists relies on several aspects. The first is that we were looking for a common melodic style, to make our prediction perform better. Despite Mozart and Beethoven belong to different styles, ranging from "classical" to "romantic" period, they share a similar musical stamp, especially in piano-solo compositions. Maybe affected by their common master Haydn. The choice of Chopin has occurred because we were looking for a third composer who could increase the size of training set and in the meanwhile keeps a similar identity to the former artists. Chopin, in fact seem to follow Mozart style consisting of longer lines of melody, gradually increasing and decreasing.

After the creation of our dataset, the first step is to examine the data we will be working with. To understand what can be done with this type of file we started by isolating one of them and analyzing it. The element that the library Music21 allow us to visualize in a midi are information divided in classes, and they deal with the instruments, notes and chords, and pitches. We will parse the files in order to get a list of two types of object, Notes and Chords: Notes objects contain information about the Pitch (the frequency of the sound, represented with letters), Octave (to indicate which set of pitches on the piano we are using) and Offset (where the note is located in the piece). Chords objects indicate a set of notes played at the same time. Using Music21 we can easily look at these parameters, we realized some useful graphs in which we can see the distribution of notes and the frequency of the pitch class, and we have plotted together the pitch and the offset of a song. Later we will visualize the frequency of notes in the whole dataset. We are now able to create sequences of our notes' array. The output of our neural network will have to be able to predict which note or chord is coming next, resulting in a prediction array that is actually a sequence of these objects forming a new piece. Regarding the offset, we know that notes usually have varying intervals between them; anyway visualizing some parsed songs we noticed that they share a common interval of 0.5. Therefore to simplify our model we can ignore it and add it directly to the final result, this will not affect in a direct way the melodies of the generated music.

V. Pre-processing

Once have a general overview of our data, it's possible to proceed to a pre-processing part. Data are stored in a folder.

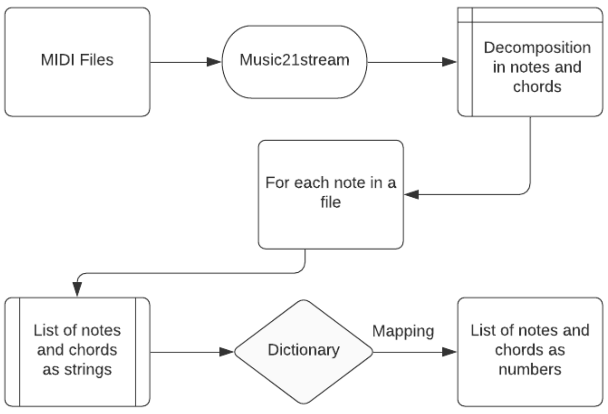

MIDI Files
Music21stream
Decomposition in notes and chords
For each note in a file
List of notes and chords as strings
Dictionary
Mapping
List of notes and chords as numbers

Fig. 2: Preprocessing

We start by loading each file into a Music21 stream object, that will gives us a list of notes and chords. After that is possible to create an output list, to which we're going to append the pitch of every note object using its string notation;

then we're going to append every chord object by encoding the id of every note together with each note separated by a dot, into a single string. This encoding allows us to keep a distinction between the two types of object and have as output a combination of them. Once we've processed this huge list of notes and chords, it's time to turn it into tensors, so what we actually use to feed the network. We start by creating a dictionary that map the element of the list, that are now strings, to numerical data, since the neural network will perform much better with them.

Next we have created input and output sequences for the networks: we have taken a fixed sequence length in the list, in our case 100, and used this sets to create input sequences and their respective output given by the first element after the input. This means that to predict the next note in the sequence the network can look at the previous 100 notes. The final step is to normalize the input and encoding again output to integers, to make both of them compatible with the neural network's layers.

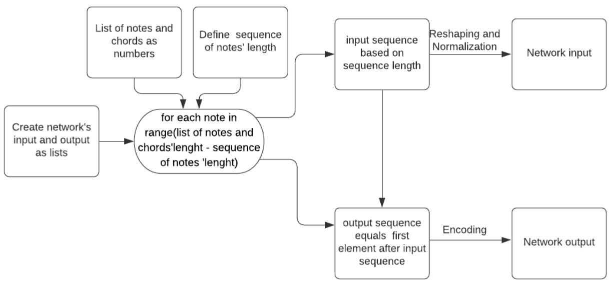

List of notes and chords as numbers
Define sequence of notes' length
input sequence based on sequence length
Reshaping and Normalization
Network input
Create network's input and output as lists
for each note in range(list of notes and chords'lenght - sequence of notes 'lenght)
output sequence equals first element after input sequence
Encoding
Network output

Fig. 3: Preprocessing

VI. Model Architecture

We proposed 4 different macro-models. Each one has been slightly adjusted each time to better fit our goal. Starting from model 0 we introduced complex models which going over time, became more and more precise and less complex. This mainly to reduce the problem of overfitting we had. Indeed since the very beginning we deal with an overfitting case, which leads our training error decrease, while the test one rise. We tried to increase the size of our dataset, from around 56 MIDI (about 90.000 training samples) to 96 (around 150.000 training samples). The situation has not improved so much and since the Colab's running times are only of 24 hours, we had have to limit any useless attempt. We'd have liked to make some more tests and analysis. Maybe in the future we can try to improve our model using an higher computational power, since this project has catch our attention and our curiosity from the very beginning.

a. Model 0

For the first model we tried to mix something related to the current State of the Art, and something related to our knowledge of the matter. We understood from the beginning that all the regularization techniques could be very important for our model architecture design. So we match different dropouts and BatchNorm layers to have a cleaner view of our data and suppress any kind of overfitting from the beginning. We used 9 layers. At the beginning a Bidirectional LSTM with 512 neurons which could have help run our inputs in two ways, one from past to future and one from future to past. Using the two hidden states combined makes us able in any point in time to preserve information from both past and future. We used a recurrent dropout of 0.3 and since we want that the layer returns a sequence, we switched the boolean return sequence value. Then we used another LSTM layer with 512 neurons, a Batch normalization layer, a dense layer followed by a ReLU activation. The second part of the model is composed by another Batch normalization layer to normalize the data of the previous part, a dropout layer with 0.3 probability and a final dense layer. As activation we used a softmax one. To compile the model we used an Rmsprop optimizer and the categorical crossentropy as loss. We trained the model on 50 epochs with 128 of batch size.

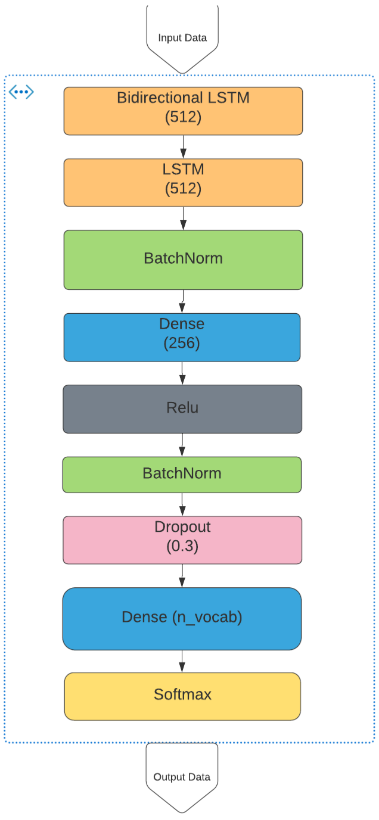


Fig. 4: Model 0

The complete model was around 5,467,961 parameters with an high overfitting problem, since the validation loss arrived around 5.40. We have also tried to train again the model for other 20 epochs (so from 50 to 70) but the results were quite bad.

b. Model 1

Ascertained the fact that our previous model was overfitting, we decided to test another model before proceding with the complexity reduction. As first layer we proceded with an LSTM (set both Bidirectional and Unidirectional) with 512 neurons. Then we introduce other 2 Unidirectional LSTM layers, with 512 neurons each. This choice was motivated by the fact that we wanted an additional way to "remember" and forget the informations of the first LSTM layers. This additional complexity would have ensured a most accurate way of forgetting useless sequences. Then we introduced a dropout layer (0.3) to prepare the sequence for the following

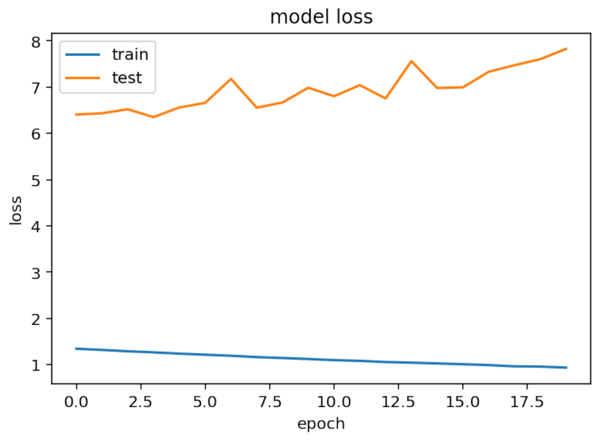


Fig. 5: Model 0b

dense layer and the activation, again with ReLU. The second part of the model is the same of the previous experiment (BatchNorm -> Dropout (0.3) -> Dense -> Activation). As activation we used a softmax one. To compile the model we used an Rmsprop optimizer and the categorical crossentropy as loss. We trained the model on around 90 epochs with 128 of batch size. The results were more or less the same of above. The training loss decreases up to around 1.77 and the validation loss fluctuates around 4.6 e 5.3.

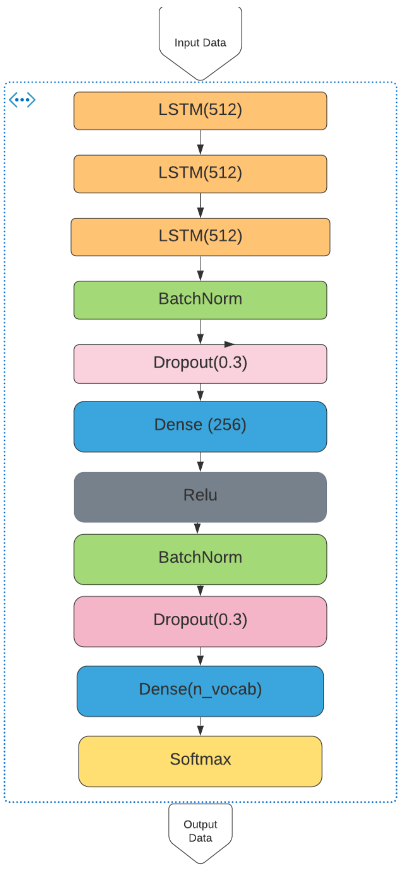


Fig. 6: Model 1

c. Model 1_b

We decided to adjust our initial part and to remove the middle activation. We created 2 LSTM-Dropout blocks with 256 and 512 neurons respectively. Then we added a Dense layer (256), a Dropout layer (0.3) and a Dense layer. As activation we used a softmax one. To compile the model we used an Rmsprop optimizer and the categorical crossentropy as loss. We trained the model on around 18 epochs with 128 of batch size. The validation loss up to 6 epochs was decreasing to 4.37. After, start rising up to 4.83 in epoch 18 when we decided to stop it.

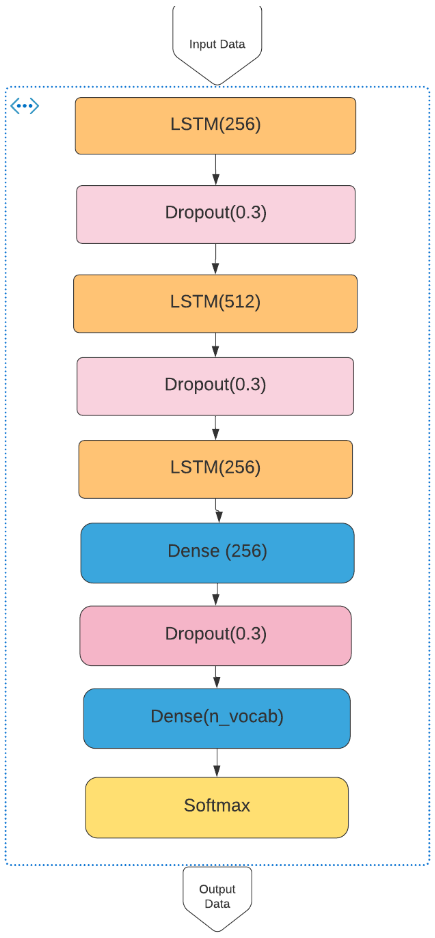


Fig. 7: Model 1b

d. Model 1_c

We created 2 LSTM-Dropout blocks with 256 and 512 neurons respectively. Then we added a LSTM layer with 256 neurons, a Dense layer (256), a Dropout layer (0.5) and a Dense layer. As activation we used a softmax one. To compile the model we used an Rmsprop optimizer and the categorical crossentropy as loss. We trained the model on around 23 epochs with 128 of batch size. The validation loss was arising and the training time reached around 17 minutes per epoch. We decided to stop it around 5.67 of loss.

e. Model 2

Since the previous results were quite bad we decided to reduce the complexity of our model. The model was composed by an LSTM layer with 512 neurons, a Dropout (0.3), another LSTM with 512 neurons, a Flatten layer and a Dense one. We used the flatten to better prepare our data for the dense layer. We tried the model also without the flatten layer but there were some problems with the dimension of the data from the LSTM to the dense layer. After we understood that in order

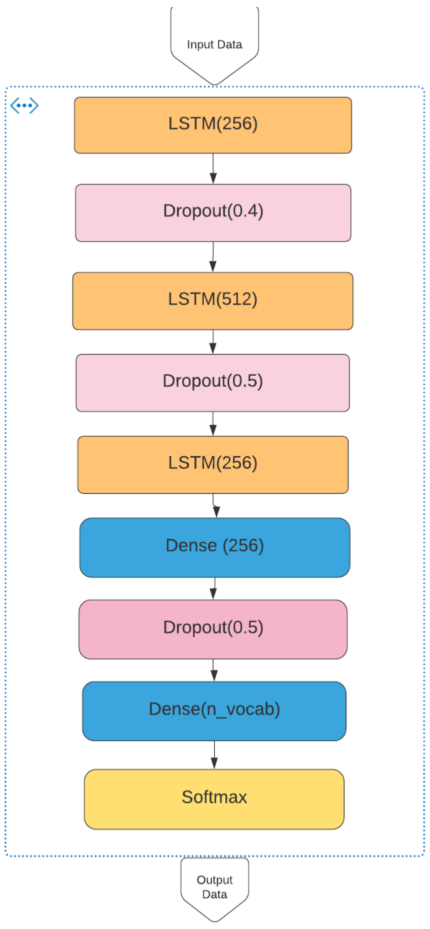


Fig. 8: Model 1c

to fix that problem we had to toggle the return sequence parameter to false. As activation we used a softmax one. To compile the model we used an Rmsprop optimizer and the categorical crossentropy as loss. We trained the model on around 16 epochs with 128 of batch size. The validation loss up to 4 epochs was decreasing to 4.37. After, start fluctuating up to 4.83 and around epoch 50 we decided to stop it.

f. Model 2_b

Here we started to change also other parameters such as the optimizers. We test the same model of before with 2 changes: first we used again a Bidirectional LSTM as first layer and then we used Stochastic Gradient Descend as optimizers. We tried also to reduce the batch size from 128 to 64 in order to add some small noise around the model and use it as a kind of regularization. The model with 64 of batch size was too slow during training (around 23 minutes per epoch) and there were no such improvements. As activation we used a softmax one. To compile the model we used an Rmsprop optimizer and the categorical crossentropy as loss. We trained the model on 50 epochs. The validation loss fluctuate around 4.80.

g. Model 3

Here we started to use Attention which make us able to remember the intermediate calculations. It's a variation of the traditional seq2seq architecture. In the seq2seq idea an encoder compresses the information into a vector of fixed lenght which can be used an abstraction of the entire input sequence. A decoder instead, initializes itself based on that sequence. In this way we're not able to remember longer sequences. Using Attention we don't discard the intermediate layers in order to create the vector. It's better to use Attention preceded by a Bidirectional LSTM layer. Then we used a dropout and another LSTM layer with 128 neurons (unidirectional). Then we used Dropout and a Dense activated with ReLU. As activation we used a softmax one. To com-

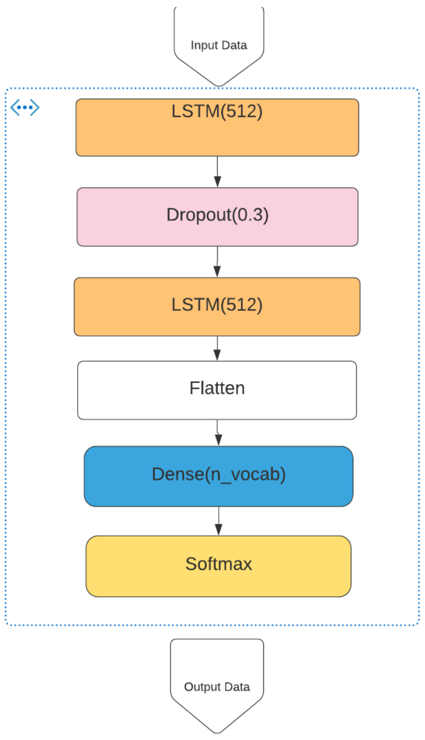


Fig. 9: Model 2

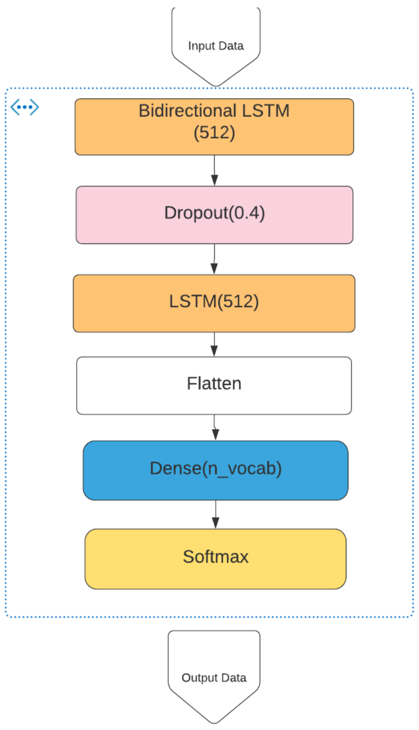


Fig. 10: Model 2b

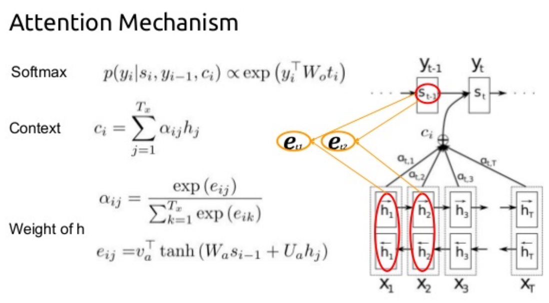

Attention Mechanism
Softmax
Context
Weight of h
$p(y_i|s_i, y_{i-1}, c_i) \propto \exp(y_i^\top W_o t_i)$
$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$
$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}$
$e_{ij} = v_a^\top \tanh(W_a s_{i-1} + U_a h_j)$

Fig. 11: Attention

pile the model we used an adam optimizer and the categorical crossentropy as loss. We trained the model on around 50 epochs. The validation loss up to 6 epochs was decreasing to 4.37. After, start rising up to 4.83 in epoch 18 when we decided to stop it.

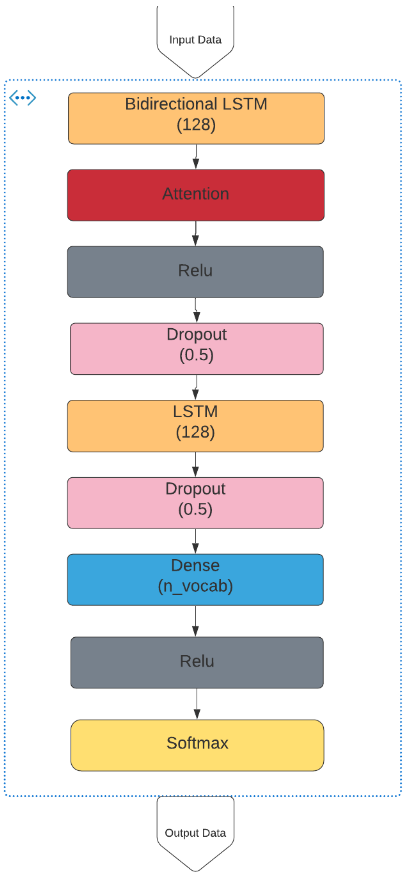


Fig. 12: Model 3

h. Model 3_b

Understood the real importance of the attention mechanism we tried to compute a smaller model based again on this framework. The model is composed by a Bidirectional LSTM with 256 neurons, Attention, another (unidirectional) LSTM with 128 neurons. At the end we added a Dropout (0.3) and a final Dense layer. As activation we used a softmax one. To compile the model we used an adam optimizer and the categorical crossentropy as loss. We trained the model on 5 epochs and the results started to keep going well. We tried to train the model again in this direction in order to look for some additional improvements but the validation loss stacks around 4.92 starting from epoch 3 (21).

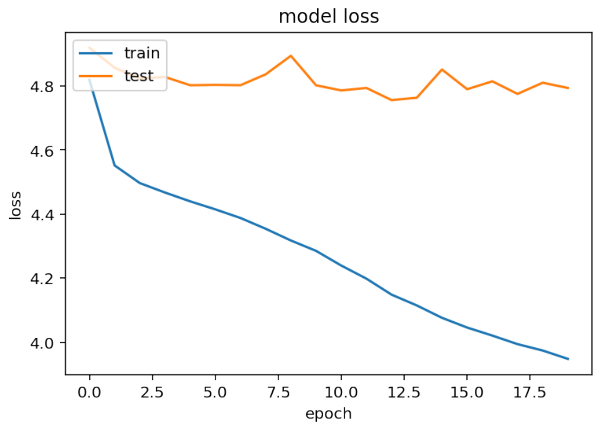


Fig. 13: Model 3

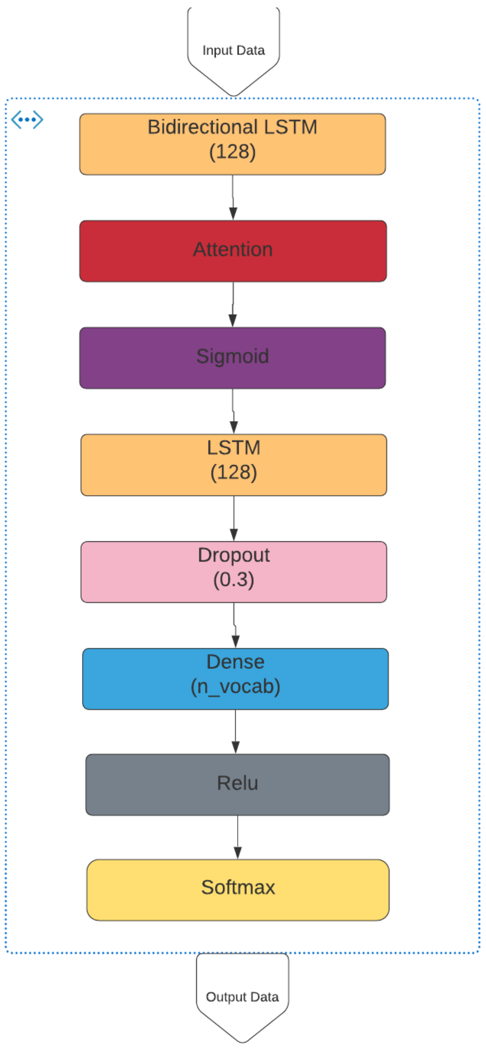


Fig. 14: Model 3b

VII. Results and Quantitative Analysis

We have not reached our final goal. The accuracy/loss of our model is not so good as expected at the beginning of the project. This is probably due to hardware problems but moreover on the complexity of our data.

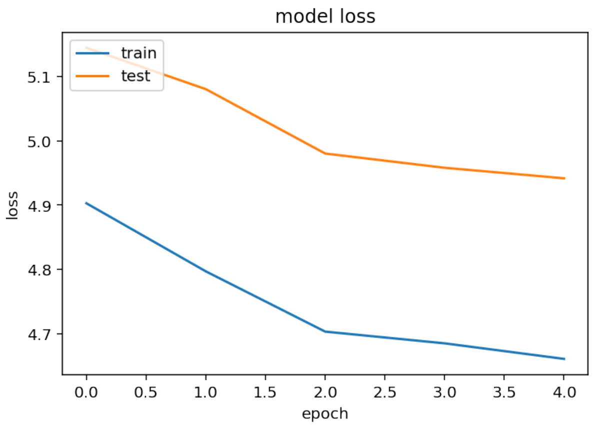


Fig. 15: Model 3b

TABLE 1: FINAL RESULTS AND RELATIVE ERROR (AVERAGES).

Model	AVERAGE LOSS	COMPUTATIONAL TIME	EPOCHS
Model 0	5.402	402s	50
Model 0b	6.220	398s	20
Model 1	4.803	460s	90
Model 1b	4.823	455s	18
Model 1c	5.670	992s	23
Model 2	4.723	638s	50
Model 2b	4.824	595s	50
Model 3	4.810	591s	18
Model 3b	4.712	602s	21

VIII. POST-PROCESSING

After training our models, the last part of the project concerns again the transformation of the data. We need to perform several operations to obtain music as final result. Here we see the importance of savings the result of each trained model: we've defined again the model that we want to use, in the same way as before, but in order to make the code executable we've loaded the previously saved weights to each node. We decided to use a random index as starting point in our output sequences, in this way we will have different results every time that we run the code, without changing anything. The first step in the actual post-processing operation is to create a dictionary, essentially the opposite of the one we have done at the beginning, to map the integers outputted by the networks to the pitch of the notes. Then we implemented a cycle to generate 500 notes, since this number will produce roughly two minutes of music. For each note that we want to generate we make a prediction of the model, using as prediction input a sequence of notes: the first sequence is the one at the starting index, then in the following we remove the first note and append the outputs of previous sequences' iterations. We select the prediction by choosing at each iteration the class of pitch with highest probability to be next note, based on the network output. Then we pass these outputs through the dictionary and collect them in an array. At this step we have an array with all the final outputs, and we must decode it making again the distinction between notes and chords. The elements in the prediction output array are passed through a cycle, if they are single notes we simply create a note object with the specified features, while if it is a chord we split it and create separately the note objects contained in it, then we create a chord object grouping again them. At the end of each iteration we insert the offset value of 0,5, as decided at the beginning, and create a final list of notes and chords objects. The final step is to use the list to create a Music21 stream object, and create a MIDI file.

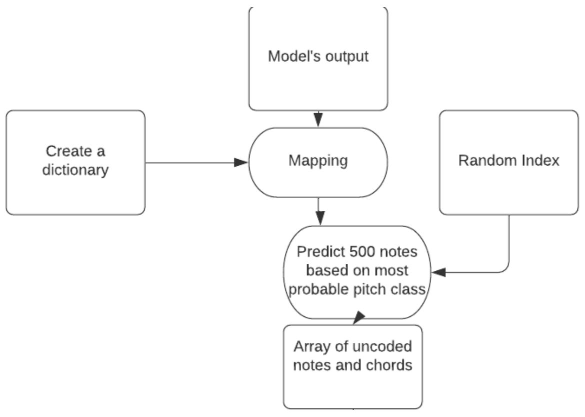

Model's output
Create a dictionary
Mapping
Random Index
Predict 500 notes based on most probable pitch class
Array of uncoded notes and chords

Fig. 16: Postprocessing

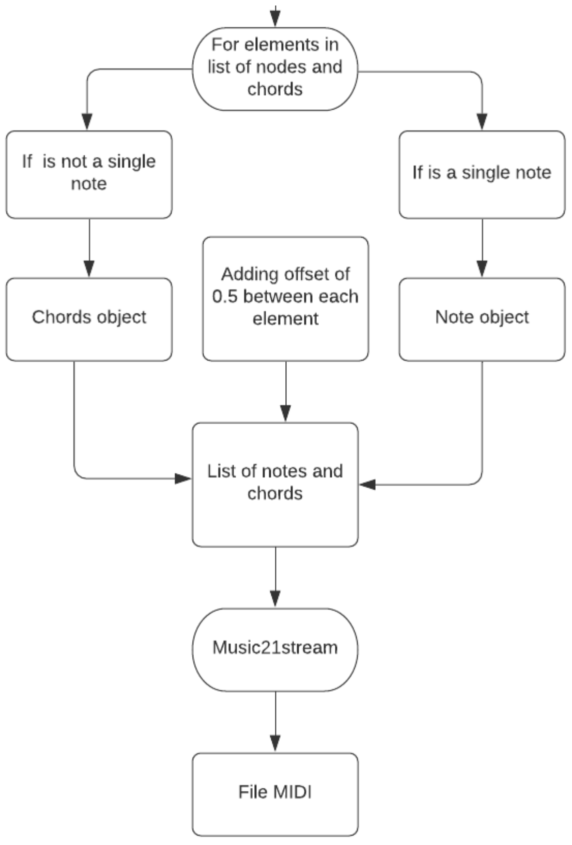


Fig. 17: Postprocessing

IX. CONCLUSION

After spending many hours implementing different models and tweaking the different parameters ,we are not so satisfied of the results we got. The performances of our models are not that astonishing probably because of the machine we used to train them. Nevertheless we won't get discouraged ad we won't back down. In the future we will keep working on it until we will achieve good performances. The main things we'll try will be 3:

- First, try to increase the sequence lenght in order to understand if this will change our accuracy;
- Second try to use different layers such as 1D Convolutional layers or a GAN;
- Third try to increase the number of musical instruments;

X. Conversion MIDI-WAV

We created a way to convert the MIDI to WAV to have a better human experience and understand the sounds. Since all the packages and libraries available for Python are all obsolete (work up to Python 2) we have installed Fluid Synth and SoundFont on a VPS with Ubuntu 18.04. This will help us to render audio to file and create a new WAV.

XI. Post Deployment Monitoring

Once the model will be deployed we designed an ideal way to keep our model updated. Since it's impossible that a machine learning model will continuously and automatically identify where it makes mistakes and adjusts itself, a monitoring mechanism is necessary. This scenario, also known as Concept Drift, make the performance of the model change when it is asked to make predictions on new data that was not part of the training set.

We thought to use a similarity metrics to understand any kind of difference between the training prediction and a general inference prediction. Similarity are measured in the closed range 0 to 1. Our choices were between Kolmogorov-Smirnov test and the Wasserstein distance. We decided to use the Wasserstein distance which aims is to measure the minimal effort required to go from one probability distribution to another.

$$l_1(u,v) = \int_{-\infty}^{+\infty} |u - v|$$

The area that we'll find will by the difference between U and V, is the distance we're looking for. At the beginning the training and inference distributions are quite the same, then it will be 0 and they will overlap themselves. As they start diverging, the distance will increase.

Moreover we thought that if the data keeps evolving and increasing, we will need to update our datasets frequently and re-train the model regularly. An important way to avoid any kind of strange behavior is to make us sure that the only MIDI files that could be used, belongs only to piano's melodies.

Moreover our systems will perform the load of the full model, with all the layers and weights info, every time there's a new prediction request.

XII. Algorithms and Data

All the project is uploaded in our GitHub Repository at the following link:
`https : / / github . com / fabriziorocco / MusicGeneration`

References

[1]deepmind.com
[2]arxiv.org/pdf/1609.03499.pdf
[3]magenta.tensorflow.org
[4]keras.io
[5]www.tensorflow.org/tensorboard
[6]web.mit.edu/music21
[7]towardsdatascience.com/neural-networks-for-music-generation-97c983b50204}
[8]medium.com/@alexissa122/generating-original-classical-music-with-an-lstm-neural-network-and-attention-abf03f9ddcb4
[9]www.analyticsvidhya.com/blog/2020/01/how-to-perform-automatic-music-generation/
[10]www.hexahedria.com/2015/08/03/composing-music-with-recurrent-neural-networks/
[11]machinelearningmastery.com/attention-long-short-term-memory-recurrent-neural-networks/
[12]www.piano-midi.de/
[13]towardsdatascience.com/intuitive-understanding-of-attention-mechanism-in-deep-learning-6c9482aecf4f/
[14]towardsdatascience.com/how-to-generate-music-using-a-lstm-neural-network-in-keras-68786834d4c5}
[15]Deep Learning with Python, Francois Chollet, 2017, Manning Publications Co.3 Lewis Street Greenwich, CTUnited States
[16]mauriziosiagri.wordpress.com/2013/12/11/convertire-file-midi-mid-in-file-audio-wav-mp3-in-linux-fluidsynth-soundconverter-timidity-vlc/
[17]machinelearningmastery.com/attention-long-short-term-memory-recurrent-neural-networks/