Analysis and Comparison of Deep Learning Methods for Jazz Music Generation

Riccardo Ratini, 1656801 Giada Simionato, 1822614
{ratini.1656801, simionato.1822614}@studenti.uniroma1.it

DIAG, Sapienza University of Rome – Via Ariosto, 25, 00185 Rome, Italy

I. Introduction

One of the most notable characteristic of Neural Networks is that they are a tool that can be subjected to a manifold of applications, even diametrically opposed. In recent years the usage of deep learning techniques in the artistic field, becomes increasingly popular. Due to the fact that the artifacts produced by these networks can be understood, commented and used also by people not having the necessary technical background, these phenomena had received a worldwide media coverage, bearing sometimes to economic opportunities, more than for any other application of neural networks even to more important fields, such as the medical one. For this reason, the amount of researches aimed at text, image and speech generation and music composition is continuously growing.

In this work we have provided an overview of the most relevant deep learning techniques used nowadays for the task of music generation. He have focused on the production of *jazz*, single-instrument compositions with a fixed duration and velocity for each note. We have implemented, trained and tuned five different architectures: the *WaveNet*, the *Simplified_WaveNet*, the *LSTMNet*, the *GRUNet* and the *Attention_RNN* network, each one with their own unique characteristics. In addition, we have constructed from scratch our own dataset and compared its performances with a second dataset. This work was completed by the deployment of a library of methods for data manipulation both for the pre-processing and the post-processing stages and by an evaluation stage in which the performances of the networks were tested against two different qualitative metrics widely employed in literature.

This report is organized as follows. Section II will present the five architectures that we have implemented and used in this work from a theoretical point of view, especially focusing on the architecture of the networks. In Section III all the details about the pre-processing and post-processing stages will be discussed, along with the information regarding the employed datasets and the training details for each network, while in Section IV results will be reported along with the explanation of the experimental set-up for the evaluation stage. Finally, in Section V conclusion will be drawn.

II. Networks Architecture

In this work five different architectures were implemented in order to give a wide overview on the most used methods nowadays to generate music tracks using neural networks. In the next paragraphs we will explain the architecture of each model as well as their advantages and drawbacks.

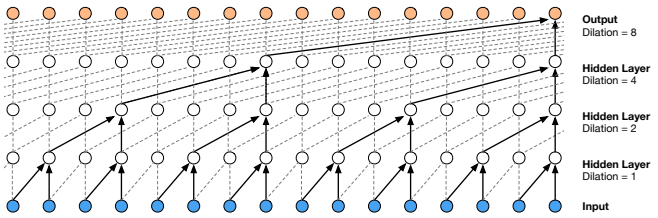


Figure 1: Visualization of a stack of dilated causal convolutional layers. Image taken from [1].

A. WaveNet

The first architecture treated in this work was the *WaveNet*: an audio generative model based on the structure of *PixelCNN*. Inspired by the work in [1], we have adapted this neural network to handle sequences of notes rather than samples of raw audio signals, as per its original purpose, by exploiting its ability in dealing with long-range temporal dependencies [1].

It models the joint probability of the sequence of notes by factorizing it as the product of conditional probabilities over all the previous timesteps [1], as described as follows:

$$p(\boldsymbol{x}) = \prod_{t=1}^{T} p(x_t | x_1, \ldots, x_{t-1}), \tag{1}$$

where $\boldsymbol{x} = (x_1, \ldots, x_T)$ is the sequence of notes and x_i the $i-th$ note. The conditional probabilities in Eq. (1) are modelled with stacked convolutional layers.

The WaveNet is mainly composed by one-dimensional *dilated causal convolutional* blocks, able to influence the output using the majority of inputs due to the increment of the receptive field by keeping limited the computational requirements. A *causal* convolution is a convolution among the output at the current time and the elements of only the previous timesteps. In this way the predictions provided by the model at a precise timestep cannot depend on any of the future timesteps: an essential requirement for generative models. However, a causal convolutional block has a very low receptive field, due to its inability to look back at timesteps occurred before the window determined by the dimension of its kernel. A solution to this issue comes from the *dilated causal* convolution, whose functioning is depicted in Fig. 1, that is a convolution whose filter is applied over an area larger than its length by skipping input values with a certain step, denoted as *dilation rate* [1].

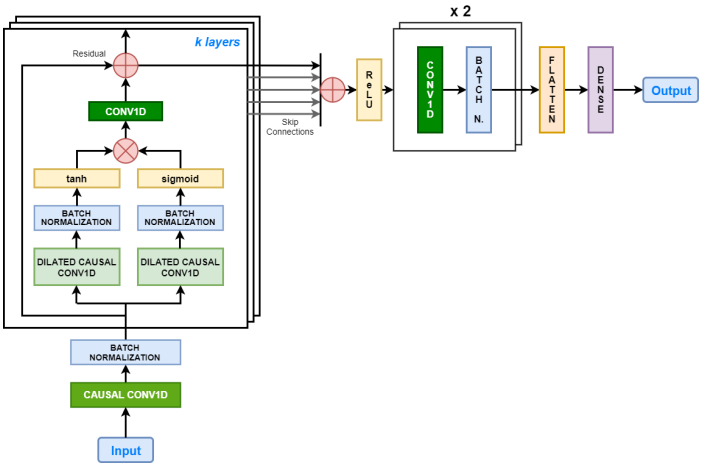


Figure 2: Full WaveNet architecture.

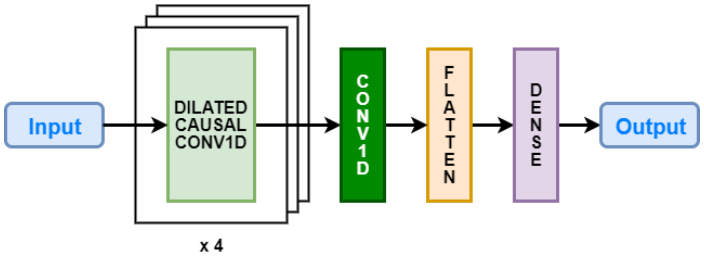


Figure 3: Full Simplified_WaveNet architecture.

Stacking several dilated causal convolutional blocks, increasing exponentially the dilation rate until reaching a certain threshold[1], allowed the increase of the reception field in the most efficient way.

The network is leveraged by the addition of residual and skip connections to further speed up convergence and the overall training stage. The complete architecture is reported in Fig. 2.

As it is possible to see the network is composed by an `Input` layer, followed by a causal `Conv1D` with n filters, kernel size of 2 and $ReLU$ as activation function. In order to repress the overfitting, we have added a $L2$ regularizer as kernel regularizer and a `BatchNormalization` layer after it. Regarding the k residual blocks, each one is composed by two dilated causal `Conv1D` layers, with dilation rate exponentially growing, n filters and 2 as kernel size. Both possess a $L2$ regularizer for the kernel, but one has the *hyperbolic tangent* as activation function and the other the *softmax*. Both layers are followed by a `BatchNormalization` one, whose outputs are multiplied and fed into a `Conv1D` regularized layer that represents the skip-connection, finally summed to the residuals as depicted in Fig. 2. Once having summed all the skip-connections of the residual layers, a `ReLU` activation function is applied, followed by two `Conv1D` regularized layers with 1 filter, dimension of kernel equals to 1 and $ReLU$ activation function, interspersed by `BatchNormalization` layers. Finally, the data are flattened and the output layer is a regularized `Dense` layer, whose number of units is equal to the size of the vocabulary[2] over which it is applied a softmax distribution. We have used the *categorical cross entropy* as loss function. More details on the other hyperparameters can be found in Section III-C.

Finally, it is worth to mention that this network, together with its simplified version that will be explained in Section II-B, is the only model used in this work that uses convolutional layers to capture temporal information, therefore being notably faster than the other models due to the lack of recurrent connections [1].

[1]As in the original work, i.e. [1], we have set the threshold to 512, hence using dilation rates following this sequence: $1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1, 2, \ldots$.

[2]More on this can be found in Section III-B.

B. Simplified_WaveNet

The second architecture that was implemented in this work is a simplified version of the network presented in Section II-A. It is an adaptation of the work in [2] and the complete architecture is shown in Fig. 3. It is devoid of skip and residual connections and any batch normalization and regularization.

In fact, after the `Input` layer there is a sequence of dilated causal `Conv1D` with 20 filters, kernel size of 2 and $ReLU$ as activation function. After the dilated layers there is a `Conv1D` with 10 filters and kernel size of 1. Then data is flattened and, as the original network, the output layer is a `Dense` one with a *softmax* activation function. The loss function used was the *mean squared error* with *Adam* as optimizer.

C. LSTMNet

Adapted from [3], the *LSTMNet* was the first architecture implemented in this work to use recurrent connections. As the name suggests, it is mainly composed by *Long-Short Term Memory* cells. These cells, whose depiction is reported in Fig. 4, are a solution to the vanishing gradient problem affecting the *recurrent neural networks*, responsible of their potential lack in remembering past information in long sequences. An LSTM cell handle information by means of gates and by keeping an internal state. The *forget gate* is the first gate whose purpose is to filter which information are worth to be kept and which ones can be forgiven. Applying the *sigmoid* function, it maps the previous hidden state and the current input to a value ranging from 0 to 1: the closer to zero the closer is to forget. With a similar mechanism, the *input gate* takes the previous hidden state and current input and applies the *sigmoid* function and the *tanh* one into two separate branches. The former maps them to a value from 0 to 1: the closer to one, the more important the data is. The latter, instead, maps them into values ranging from -1 to 1 to regulate the network. By multiplying the two outputs, the sigmoid one will determine which information to keep. The new *cell state* is computed by the sum of the output of the input gate with the result of the multiplication between the forget gate output and the current cell state. Finally, this value is used in the *output gate*, that determines the new hidden state of the cell, after being passed through the *tanh* function and multiplied by the output of the application of the sigmoid function to the current input and the previous hidden state.

As it is possible to see in Fig. 5 showing the full architecture, after the `Input` layer there is a sequence of three `LSTM` layers

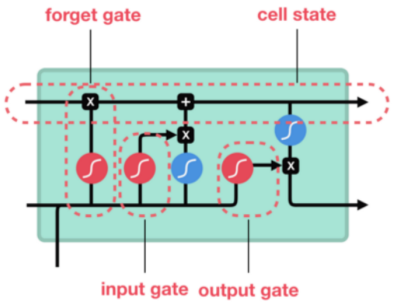


Figure 4: Internal visualization of a Long-Short Term Memory cell. The blue and red circles represent the application of the *tanh* and *sigmoid* functions. The $\times$ and $+$ symbols represent the element-wise multiplication and sum, respectively.

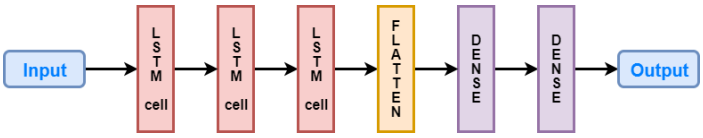


Figure 5: Full LSTMNet architecture.

with *hyperbolic tangent* as activation function, followed by a `Flatten` layer and two `Dense` layers, where the former has a *ReLU* and the latter the canonical *softmax* as activation function. All the layers are provided with dropout and, for the LSTM layers also the recurrent dropout. The loss used was the *categorical cross entropy*.

D. GRUNet

Another recurrent network implemented in this work is the *GRUNet*. Using [4], we have trained the network in Fig. 7. As it is possible to see it is mainly based on Gated Recurrent Unit (GRU) cells, shown in Fig. 6. Newer with respect to the LSTM cell, the GRU one has only two gates and uses the hidden state to transfer information. The *update gate* regulates the information that is important to keep and which one to discard, similarly to the input and forget gates of the LSTM cell. The *reset gate* instead is used to decide how much past information to forget.

As can be seen in Fig. 7 the architecture of the GRUNet is composed by an initial `Input` layer followed by an `Embedding` one, used to turn positive integers, i.e. the indices of the notes in the input sequences, into dense vectors of fixed length. Then, there are two blocks composed by a `Bidirectional GRUCell`, a `self-attention` layer[3] and a `Dropout` one. This is followed by another block composed by a `Bidirectional GRUCell` layer and a `Dropout` one. Finally, there is a `Dense` layer with *Leaky ReLU* activation function and a final `Dense` layer with *softmax* activation function to compute the probability distribution over the output notes. For this network the loss used was the *sparse categorical cross entropy* since the labels were in the integer shape rather than in the one-hot encoding one, as in the rest of networks.

[3]The functioning of the attention mechanism will be presented in Section II-E.

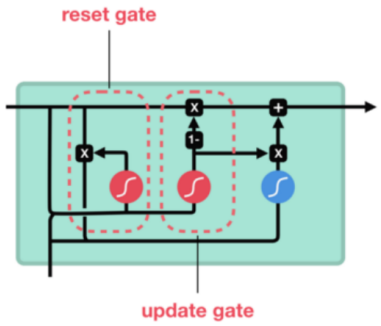


Figure 6: Internal visualization of a Gated Recurrent Unit cell. The blue and red circles represent the application of the *tanh* and *sigmoid* functions. The $\times$ and $+$ symbols represent the element-wise multiplication and sum, respectively.

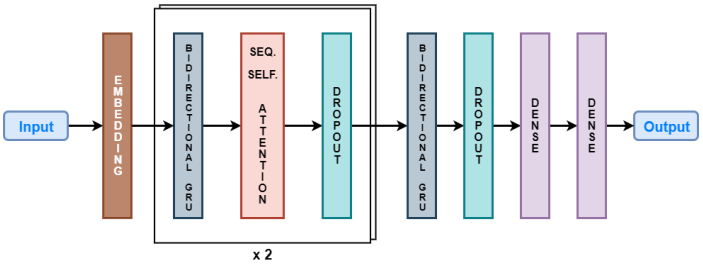


Figure 7: Full GRUNet architecture.

E. Attention_RNN

The last model used in this work was provided by the novel *Magenta* Python library powered by *TensorFlow* [5]. This tool allows to use pre-trained recurrent models to produce text or music tracks, or training these models from scratch using your own dataset. Specifically for music generation, Magenta's authors provide four different architectures based on recurrent cells: the *Basic_RNN*, the *Mono_RNN*, the *Lookback_RNN* and the *Attention_RNN*. After having considered the advantages and the drawbacks of each model, we have decided to employ the last one for this work. It leverages the standard RNN network with the addition of an *attention* mechanism to make the system learn even longer-term structures. In this model the RNN cell is an LSTM as the ones of Section II-C. The attention mechanism is constructed by deriving an *attention mask* $\boldsymbol{a}_i^t$ for each timestep. At first, the $\boldsymbol{u}_i^t$ vector is computed as in Eq. (2), where $\boldsymbol{v}^T, \boldsymbol{W}'_1$ and $\boldsymbol{W}'_2$ are learnable parameters of the model and $\boldsymbol{h}_i$ (with $i = t-j, \ldots, t-1$) and $\boldsymbol{c}_t$ are the RNN outputs of the last j steps and the current cell state, respectively. The resulting vector represents how much attention each step should receive. Then $\boldsymbol{a}_i^t$ is obtained from the application of the *softmax* function to $\boldsymbol{u}_i^t$, to normalize the values. Finally, the new output of the RNN cell can be obtained as in Eq. (3), in such a way that it encodes useful information on the most relevant elements of the input sequence and their long-term relationships.

$$\boldsymbol{u}_i^t = \boldsymbol{v}^T tanh(\boldsymbol{W}'_1 \boldsymbol{h}_i + \boldsymbol{W}'_2 \boldsymbol{c}_t) \tag{2}$$

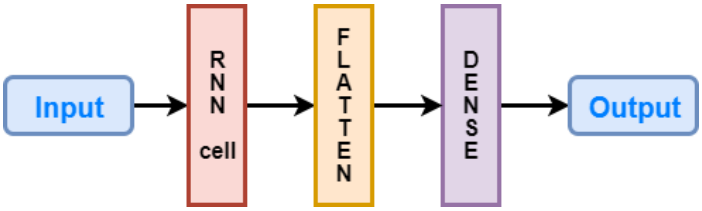


Figure 8: Full Attention_RNN architecture.

$$h'_t = \sum_{i=t-n}^{t-1} a_i^t h_i \tag{3}$$

The complete architecture of the network is reported in Fig. 8, where it can be seen an `Input` layer followed by a `RNN` (LSTM) cell, a `Flatten` layer and a `Dense` layer with a *softmax* activation function.

The details about the hyperparameters and training information will be disclosed in Section III-C.

III. Implementation Details

In this section we will discuss in depth each step of the implemented pipeline, shown in Fig. 9, to handle data and the models, putting particular attention to the details of the datasets employed.

A. Datasets

To train the models implemented in this work, two datasets were used, namely the *Novel* dataset and the *Jazz ML ready MIDI* one, denoted from now on as *Full* dataset. These resources are composed as follows.

1) Novel Dataset: The *Novel* Dataset was manually constructed by us, collecting 37 jazz songs with a duration ranging from almost 2 to 4 minutes each. We have downloaded each of the songs individually, directly in the `MIDI` extension, as to create a meaningful collection that mirrored the desired category of jazz sound. In order to provide a dataset big enough, we have explored similar researches and studied the composition of the collection of data used. From this analysis we drawn that at least 25 songs, with the same characteristics of our dataset, were required to correctly train the models.

After having collected all the data, we have implemented a set of methods, using the `music21` library, as to allow a simple interface with this resource and to deeply study its composition. To this end we have deployed a library of methods to analyse the structure of each song of the collection, providing a full list of the notes encountered in the whole dataset, thus allowing us to derive useful statistics. We have extracted the set of the unique notes present in the data, along as their frequency. Then, we have transformed the frequencies from absolute to relative and plotted them in decreasing order, as to see the degree of unbalancing of the dataset. The result of the notes distribution is depicted in Fig. 10(a). In the $x-$axis the name of the notes were hidden for the sake of clarity and each dot represents the percentage of the corresponding relative frequency. As it is possible to see, the dataset is quite imbalanced with the majority of notes having very few occurrences each. As will be explained in Section III-B, the number of unique notes, defining the vocabulary, is also the size of the output layer of each model. For this reason, the lower the number of different categories, i.e. the unique notes, the more effective and faster the training will be, however, the variety of the melodies will decrease. In addition, reducing the number of unique notes forces the pre-processing methods to replace the missing notes with an `<UNK>` tag. In order to find the best trade-off between the size of the output layer and the number of notes replaced, we have set up a relative threshold to filter the notes whose occurrence is lower than this limit. As regard the Novel dataset, without any threshold the number of unique notes, and of units in the output layer, was 513. Setting a relative threshold of 0.1 this number dropped to 179 units, with a reduction of the 64.107% of the output size, while maintaining intact the 91.49% of the dataset[4]. This operation reduced the number of network parameters as well as speeding up the training stage.

The Novel dataset can be found in the GitHub repository of the project in [10].

2) Full Dataset: Another dataset used in this work was the *Full* dataset, that can be found in [6]. To the best of our knowledge this is the only dataset of jazz songs available online. It contains directly the sequences of notes extracted from the `MIDI` version of 818 well-known jazz songs. In addition, it provides the name of each song, the number of total notes for each song, the set of unique notes and its size, in the `.csv` format. As for the previous dataset, we have implemented several methods for extracting the desired data and to analyse its content. We have performed the same analysis that was done for the Novel dataset, obtaining the results in Fig. 10(b). As it is possible to see this dataset is very imbalanced, with the majority of occurrences belonging to a small set of notes and with a large amount of notes occurring only once[5]. In order to reduce the output layer size we have set a relative threshold of 0.1, as the previous case, bringing the 1216 unique notes down to only 138, with a reduction of the 88.65% of the output layer size. Despite this steep decrease, the dataset was kept intact for the 92.576%.

Comparing the two distributions in Fig. 10 it is clear that the Full dataset is more imbalanced than the Novel one, despite both have shown the extensive usage of a small set of notes.

B. Pre-processing

To manipulate the data to feed the models, we have used two different pre-processing strategies denoted as *NLP-Based* and *PianoRoll-Based*. The former takes the name from the affinity that this strategy has with the pre-processing stage of generic NLP applications. It was applied to the data for the *WaveNet*, the *Simplified_WaveNet* and the *LSTMNet*, while the *PianoRoll-Based* was used with the *GRUNet*. The *Attention_RNN* network uses its own methods for pre-processing data based on private libraries and therefore not related to these strategies.

The first step in the NLP-Based pipeline consists in extracting a sequence of notes from each song of the dataset

[4]This means that only the 8.51% of the notes would have been replaced by the `<UNK>` tag.

[5]Besides the canonical seven notes of the musical scale, in this work we have considered basic notes of different octaves as different elements and chords composed by multiple notes as single different elements.

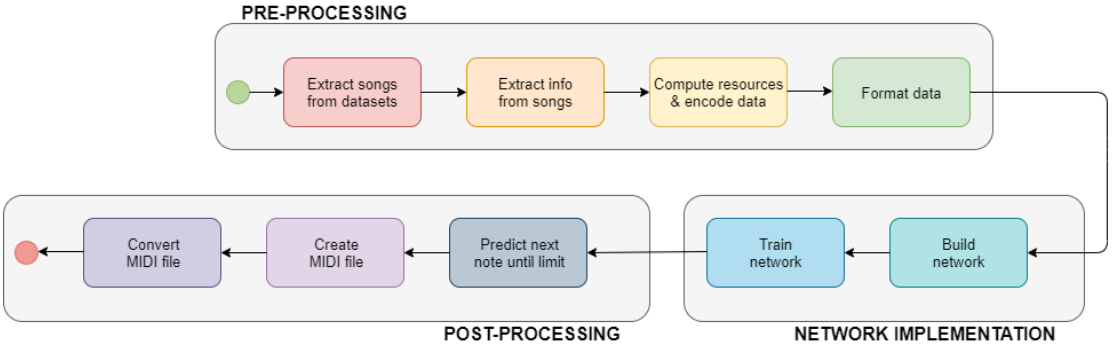
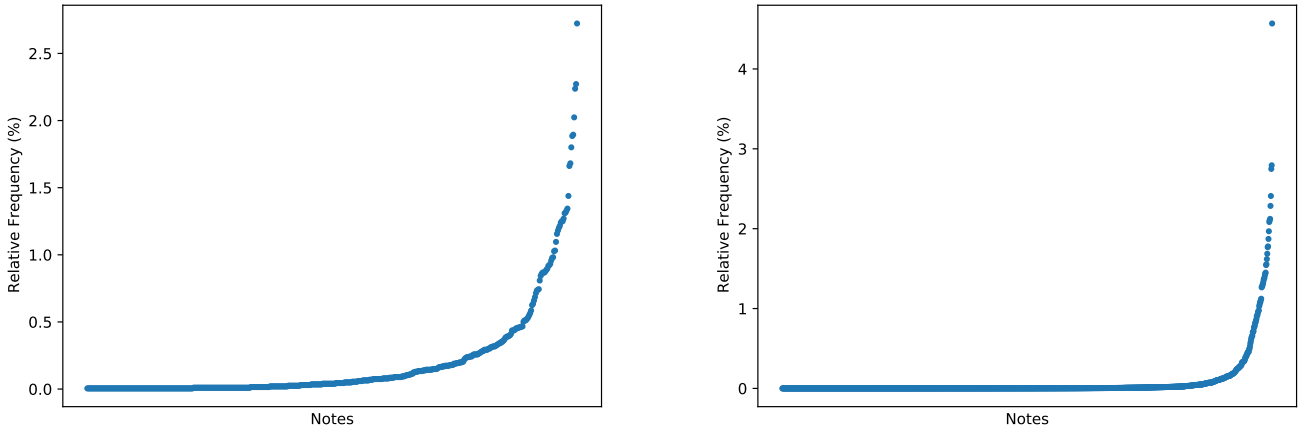


Figure 9: Schema of the implemented pipeline.



(a) (b)

Figure 10: Analysis of the datasets composition: *(a)* Novel Dataset; *(b)* Full Dataset.

using the library `music21` and a set of methods that we have implemented. From these lists of notes our methods create a vocabulary, and its reversed version, composed by all the unique notes in the lists along with an unique index representing its encoding. The indices 0 and 1 are pre-assigned to the padding (`<PAD>`) and unknown (`<UNK>`) tags that represent the padding element and an element not present in the vocabulary, respectively. As mentioned in Section III-A, we have set a threshold for the frequencies of the unique notes as to reduce the vocabulary size by removing the least-occurring ones. The extracted songs are then encoded using the vocabulary with the corresponding index and truncated or padded as to match a precise input and output shape without any loss on information. As regard the output data, the note that followed the corresponding last input note is the output label that the network should learn to predict. This index is then converted into one-hot encoding shape with the length equals to the size of the vocabulary.

Regarding the PianoRoll approach, at first, the notes in the piano roll format are extracted from each song of the dataset. This piano roll is then converted into a dictionary in which each key is a temporal frame and each value is the corresponding list of indexed notes, then padded or truncated to match the input shape as in the other pre-processing strategy.

Finally, the next timestep with respect to the input sequence, hence the target value, is selected for the corresponding input. The sequences may also contain the empty tag, i.e. *e*, to indicate a timestep without any note.

It is worth to mention that both strategies handle data as to match the required input shape, while the output differs since the PianoRoll represents the labels as indices while the NLP-Based with their one-hot encoding vector.

C. Networks Implementation

The models were trained for at most 100 epochs each, using the resources provided by *Google Colaboratory*. They were implemented using both the *Keras* library and native *Tensorflow* (both 1.1x and 2.x). We have done several independent training for each model, in order to perform an *hyperparameters tuning* stage of the most relevant hyperparameters of each network. In order to choose the most appropriate values for each parameter, both in terms of values to assign and to decide which configuration was the best performing, we have relied on similar studies and on our qualitative judgments based on a manifold

of keypoints, e.g. the presence of a repeated pattern or the times a particular note was repeatedly chosen.

During the training stage we have encountered different behaviours that allowed us to implement customized solution in order to increase their performances.

For each training we have inserted two *callbacks* in order to retrieve additional information of the current training and to automatically regulate it:

- *ModelCheckpoint*: to occasionally save the training weights. More specifically, only the best model was saved, according to different metrics, eventually using the *loss* one;
- *TensorBoard*: to keep log information and to check the training behaviour.

In the next paragraphs, we will overview the configurations used, which one was the most performing and any relevant implementation detail[6].

1) WaveNet: The WaveNet architecture was the most extensively trained one: with the same parameters, in different configurations, we have used: only the Novel dataset; only the Full dataset[7]; a mixture of both datasets. In each of the comparisons we have observed how the Full dataset drifted the results towards more randomized sounds, loosing temporal-repeated patterns or even making the network producing always the same note. By looking at the notes distribution in Fig. 10(b), even after the application of the threshold, it is possible to see that the Full dataset remained quite imbalanced and by listening to random songs belonging to it, it can be heard the different categories of jazz grouped all together, therefore lacking of homogeneity. For this reason, for the fine tuning of the WaveNet, and for all the other networks, we have relied only on the Novel dataset.

The imbalanced nature of the datasets, even if reduced by the action of the threshold, was the reason behind a bad behaviour affecting the network, independently from the dataset used and from the configuration of the hyperparameters. The WaveNet degenerated, producing always the same note that was either the most frequent note or the `<UNK>` tag strictly depending on the magnitude of the threshold. However, we have observed that the frequency of this phenomenon was dependant on which dataset was employed: with the Full one this happened at every generative process, while for the Novel one was less frequent. In order to solve this problem, we have specified a dictionary of weights to the `class_weight` parameter at the fitting time. This option forces the network to put more attention to under-represented classes, i.e. notes, by weighting heavily the few occurrences of that specific class through the above-mentioned dictionary of weights, one for each class. To correctly construct the dictionary, we have used the method `compute_class_weight` of *sklearn*. After having specified this additional element, we have no longer encountered this issue.

The first parameter to be tuned was the length of the sequence in input to the network with values in the set $\{16, 32, 64, 128\}$, resulting in 64 to be the best one. Then, we have tuned the number of convolutional filters among $n = \{64, 128, 256, 512\}$, resulting in 256 and the number of residual blocks among $\{3, 5, 7, 30\}$. This latter value was the same of the original network in [1], however the best one was 5. For all the training we have used the *Adam* optimizer.

2) Simplyfied_WaveNet: The Simplified_WaveNet was trained in the same way of the WaveNet in the previous paragraph. As the complete network, even this one was affected by the degeneration of the outputs, for this reason we have applied the dictionary of weights also in this case. The final configuration of hyperparameters is the one reported in Section II-B. This network was the quickest one to train, in term of average time elapsed for each epoch.

3) LSTMNet: For the LSTMNet we have considered configurations tuning the following parameters:

- *Dropout*: the dropout and recurrent dropout rates in $\{0.1, 0.3, 0.5\}$;
- *Hidden units*: the number of hidden units in the LSTM layers among $\{100, 200, 300\}$;
- *Optimizer*: the optimizer among $\{Adam, Nadam, SGD\}$.

All the configurations were completed by an input sequence length of 32, batch size of 64 and 256 as number of units for the first `Dense` layer. The best configuration resulted in 0.3 dropout/recurrent dropout rates, 300 hidden units and $Nadam$ optimizer. As previously mentioned we have used the Novel dataset. As for all the RNN based networks, it has not shown the need of adding the dictionary of weights. This network was the one that took the highest amount of time to be trained (averaged per epoch).

4) GRUNet: The GRUNet was trained on the Novel dataset without the need of the dictionary. The values of the parameters chosen were:

- *Seq_length*: the input sequence length among $\{16, 32, 64, 128\}$ as in the WaveNet case;
- *Dropout*: the dropout rate among $\{0.1, 0.3, 0.5\}$ as in the LSTMNet case;
- *RNN units*: the number of RNN units among $\{64, 128, 256\}$;
- *Optimizer*: the optimizer among $\{Nadam, RMSProp, SGD\}$.

The best configuration resulted to be: 64 as sequence length, 0.1 for the dropout rate, 256 RNN units and $Nadam$ optimizer. The training were accompanied by a batch size of 96 and fixed embedding size of 100. Finally, it is worth to mention that this is the only network in our work to have an embedding layer.

5) Attention_RNN: As regarding the Attention_RNN network, rather than changing hyperparameters we have focused on studying the functioning and the performances of different models implemented in the Magenta library. More specifically, we have considered several architectures with their default values: the Basic_RNN, the Lookback_RNN and the Attention_RNN, by training from scratch the networks with the Novel Dataset. The last one ended up to be the best performing, with the following hyperparameters:

- *batch size*: 128;
- *RNN layer sizes*: $[128, 128]$;

[6]If an information is not explicitly reported, default value was intended.

[7]For memory requirements, we have used only at most half of the songs present in the dataset.

- *Dropout*: 0.5;
- *Attention length*: $j = 40$, hence the size of the window for the previous samples;
- *Learning rate*: 0.001;
- *Optimizer*: $Adam$.

The loss used was the *softmax cross entropy* and no residual connections were implemented.

D. Post-processing

Regarding the post-processing stage, a new track is generated starting from a random input sequence, that could be either the initial m notes of a song or a completely random sequence of notes, using the methods that we have implemented. This sequence is fed into the predict stage of the trained model that generates a softmax distribution over the vocabulary of notes: the note with the highest probability is the new one that is appended at the end of the input vector for the next prediction, while the first note is removed to guarantee continuity in the input shape. This process is repeated iteratively until the required number of notes is reached. Despite of the input vector changing its components, the sequence of notes generated from the beginning of the process is stored in a separate list used to construct the MIDI file. They are encoded into the desired MIDI channel, that is the *piano* one in this work, using either the `music21` or the `PrettyMidi` libraries. The notes intensity and timesteps were kept at predefined values since they were not meant to be produced by the models. The generated MIDI files were then converted into the .mp3 extension to be played more easily in the evaluation stage.

Some of the models in few cases have generated sequences containing one or more `<UNK>` tags, this raises an exception when is tried to be converted as part of a MIDI file by the above-mentioned libraries. We have experienced this issue only with the CNN-based networks, even if strongly reduced by the presence of the dictionary of weights introduced in Section III-C. To tackle this issue we have implemented a rule that picks the second note with the highest probability whenever the top one is the unknown tag.

Examples of .mp3 audio tracks produced by all the models can be found in the repository of the project in [10].

IV. Evaluation and Results

In order to evaluate the audio files generated from the different architectures that were implemented, we have relied on two different qualitative metrics: the *Subjective Grouped Comparison* (SGC) and the *Mean Opinion Score* (MOS). We have explored different studies that covered topics related to deep music generation as to find reliable metrics to capture the performances and we have then extended them as to solve phenomena not previously taken into consideration in the above mentioned researches. The evaluation stage was carried on with two experiments involving 27 persons, divided into two groups of 13 and 14 subjects for the SGC and MOS computation, respectively, as to prevent the transfer-learning problem that could affect the opinions for the questionnaire filled as second, due to the users now used to hear such kind of sound. For both the experiments a questionnaire was employed, drafted using the *Google Forms* service. These questionnaires were administered to each person individually, as to prevent the answers from being influenced by external opinions.

In the next paragraphs we will present the two metrics along with the set up of the experiments for these evaluations and the obtained results.

A. Subjective Grouped Comparison

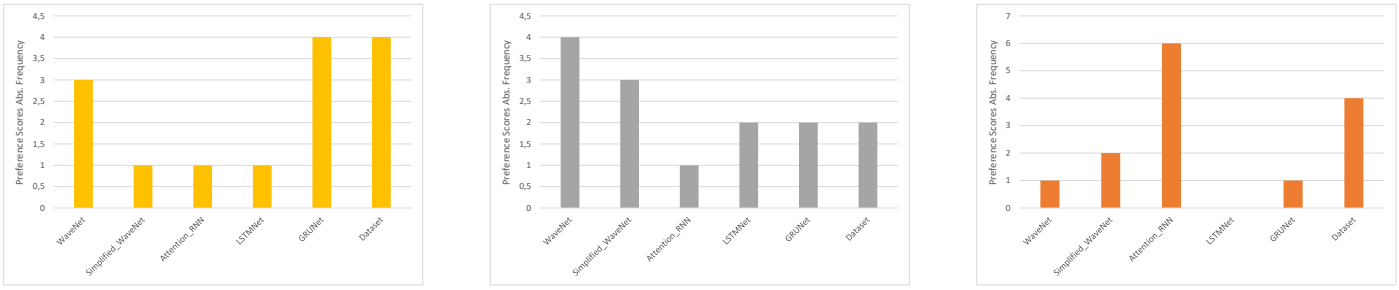
The first metric employed to evaluate the performances of our models was mainly inspired by the work in [1]. We have drafted a questionnaire with links to six different .mp3 audio files, with encoded name, corresponding to one output for each model implemented in this work, with the addition of one pre-processed file extracted from the dataset. The idea of including the dataset was partially suggested by [7] in order to capture the relationship between the real and the generated music. We have asked to the selected people to produce a top 3 chart putting at the first place the number corresponding to the audio that they found the most *natural*. Throughout the evaluation stage we have considered only the concept of *naturalness* to judge all the performances: this aimed at describing how much the sound seemed real, hence composed by an human being, with an underlying melody, differently from a simple sequence of random notes. This concept, different from *likeness*, was also employed to minimize the bias introduced by the personal taste in music and the possible lack of knowledge of jazz music patterns. All the participants to the two experiments were kept in the dark from the fact that the tracks were generated by neural networks and that real songs were part of the pool of audio files as to avoid biased results. Fig. 11 shows the raw results of the subjective grouped comparison.

In order to make the results more compact and straightforward to understand we have multiplied, for each architecture, the number of first places reached, with a weight of 5, the one of the second places with 3 and the one of the third places with 1. Finally, we have summed all these quantities obtaining a value for each model: the higher the more preferred. We have selected a step of 2 among the weights as to amplify the diversification of the results. These values are reported in Table I.

As it is possible to see in Fig. 11 and Table I, the audio drawn from the dataset achieved the highest score, strictly followed by the WaveNet and by the GRUNet, respectively, while the LSTMNet was the least preferred. The little difference on the SGC values between the real and the best-model generated music, points out that we were able to achieve remarkable performances in the single-instrument jazz music generation.

B. Mean Opinion Score

The second metric used to evaluate the performances is again inspired by the work in [1]. We have drafted a questionnaire with links to audio files generated by all the models and extracted from the dataset. The participants had to rate each audio individually on a five-point Likert scale (i.e. 1 for *Bad*, 2 for *Poor*, 3 for *Fair*, 4 for *Good* and 5 for *Excellent*) according to the concept of naturalness previously described. In order to tackle the problem of transfer-learning and to prevent the network for being judged on the base of only one audio, we have generated three files for each architecture and extracted



(a) (b) (c)

Figure 11: Raw results for the Subjective Grouped Comparison metric for each architecture: *(a)* times the network was chosen for the *first* place; *(b)* times the network was chosen for the *second* place; *(c)* times the network was chosen for the *third* place.

	Architecture					
	WaveNet	Simplified_WaveNet	Attention_RNN	LSTMNet	GRUNet	Dataset
SGC score	**28**	16	14	11	27	**30**

Table I: Cumulative SGC score for each architecture.

three songs from the dataset. Then, we have sequenced three sets of files from the architectures and the dataset to capture any disparities risen by the habit of listening this kind of audio[8]. We have added dataset files, again as suggested in [7], in order to have a quantitative baseline for the true songs. This was necessary since we are used to listen complex songs, composed by more than one instrument with velocity and duration of the note variable throughout the song, while in this work we focused on single-instrument static compositions that could affect one's opinion even for real data. In this way, it was possible to understand the relationship between the real and generated data in an unbiased way. Finally, we have collected and post-processed the data as to compute the mean and the standard deviation to give a precise value to each model: results are reported in Table II.

As it is possible to see, the audio files extracted from the dataset and pre-processed were not rated with the maximum, due to the above mentioned phenomenon that people are not used to hear music without some basic components. Among all the models, the GRUNet was the best performing, with a mean value close to the dataset one, followed by the WaveNet and by the LSTMNet, differently from the previous metric.

In summary, this evaluation stage allowed us to assess the performances of the models, pointing to the WaveNet and to the GRUNet as the preferred ones, with scores very close to the ones of the real data. For this reason, it is possible to consider the audio tracks produced by these two models almost indistinguishable from real music, composed by human beings. During the filling of the questionnaire we took notes of the personal opinions of the subjects and we discovered that, if they were undecided between the WaveNet and the GRUNet, their own musical taste was decisive. This is because the GRUNet produces very vigorous compositions, that belong to a particular category of jazz, while the other models produce softer melodies: some respondents felt this energy as innatural, ending up in preferring the WaveNet. For this reason, we consider both the WaveNet and the GRUNet as the best architectures, while the Attention_RNN one, the worst.

[8]We have inserted 18 links to audios (with encoded names) in the order: {*audio_wavenet_i, audio_simplified_wavenet_i, audio_attention_rnn_i, audio_lstmnet_i, audio_grunet_i, audio_dataset_i*} with $i = 1, 2, 3$.

V. Conclusion

In this work we have addressed the task of generating jazz music using deep learning approaches. We have focused on single-instrumental melodies composed by notes of fixed duration and velocity. We have used two different datasets, one of which was constructed by us exclusively for this work, i.e. the *Novel* dataset, along with the implementation of several methods for data extraction and processing. We have presented five different neural network architectures, each one with its own advantages and drawbacks and with the necessity of additional structures to implement, namely: the *WaveNet*, a dilated causal CNN-based network; the *Simplified_WaveNet*, a simplified version of the original WaveNet; the *LSTMNet*, a recurrent network based on the usage of LSTM cells; the *GRUNet*, a recurrent network based on bidirectional GRU cells and finally, the *Attention_RNN* a recurrent network provided by the Magenta library of TensorFlow, based on LSTM cells endowed with an attention mechanism. For each listed network we have performed an extensive tuning of the most relevant hyperparameters in order to increase the performances. Finally, we have set up an evaluation stage, by identifying two different qualitative metrics, supported by the literature, and by drafting questionnaires to forward to more than twenty-five subjects. We have constructed the experiments as to provide the most meaningful and unbiased responses. The processed results have pointed out the *WaveNet* and the *GRUNet* as the almost

	Architecture					
	WaveNet	Simplified_WaveNet	Attention_RNN	LSTMNet	GRUNet	Dataset
Mean	3.33333	2.51282	2.69231	3.30769	**3.46154**	**3.97436**
St. Dev.	1.34425	0.82308	1.10391	0.76619	0.85367	0.98641

Table II: Mean Opinion Score for the different architectures.

equally preferred ones, since they produce melodies of different categories of jazz. The most remarkable result was the fact that the generated music was considered almost indistinguishable from the real ones extracted from the dataset, even if remaining different from the complete symphonies that we are used to hear.

Along with the implementation of this framework, we have identified some sources of further improvement. In order to reduce the gap from the generated music and the one that we are normally used to listen, it will be possible to endow the generated notes with additional parameters such as its duration and velocity. This could be done by constructing and training two separate networks, one for each parameter, and then combined with the one generating the notes as described in [3]. The advantage of this method is the fact that, since they are individual networks, it will be possible to build each one *ad-hoc*, to successfully manage their own type of data. However, to provide consistency to the inputs of the various networks, the dataset must provide data with the additional parameters required. This is not possible for the *Full* dataset, since it directly provides the list of notes for each song, but the *Novel* dataset that we have build already encoded these information for future developments.

Another improvement aimed at reducing the gap between the generated and real music, could be to use different systems to produce tracks for different instruments, making the output song multi-instrumental. This can be achieved by applying again the above-mentioned approach of [3], hence building a network for each instrument and then combining the results. A more interesting approach to handle more than one instrument, can be taken from [8], that uses *Generative Adversarial Networks* sharing common vectors to capture the relationships that instruments show during the composition of a symphony. However, for both the approaches the data used to feed the networks must be consistent among all the instruments, therefore the dataset has to provide the same songs under different points of view, i.e. different instruments. Again, the *Full* dataset cannot provide this data since it only has one set of notes for each song, while the *Novel* one allows the extraction of all the channels directly from the .mid songs from which it is composed.

Finally, inspired by the work in [9], we have identified the possibility of exploiting the pre-trained models of Magenta. This library offers bundle files of its models pre-trained on thousands of MIDI files, downloadable at [5]. It will be possible to exploit these models, since they are already aware of the most relevant characteristics of music tracks and then *fine tune* them using a jazz dataset to maintain the melody but steering the sound towards the jazz category.

In conclusion, we have proposed an overview of several deep learning methods for the generation of jazz music obtaining interesting results, but leaving anyway room for further improvements.

REFERENCES

[1] A. Van Den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, K. Kavukcuoglu. *WaveNet: A Generative Model for Raw Audio.* Proc. 9th ISCA Speech Synthesis Workshop, 2016, pp. 125-125.

[2] A. Géron. *Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow.* 2019, O'Reilly, 2nd Edition, pp. 521-523.

[3] J. M. Simões, P. L. Machado, A. C. Rodrigues. *Deep Learning for Expressive Music Generation.* In Proceedings of the 9th International Conference on Digital and Interactive Arts (ARTECH 2019). Association for Computing Machinery, New York, NY, USA, Article 14, 1–9. DOI:https://doi.org/10.1145/3359852.3359898.

[4] https://github.com/haryoa/note_music_generator

[5] https://github.com/magenta/magenta/tree/master/magenta/models/melody_rnn

[6] https://www.kaggle.com/saikayala/jazz-ml-ready-midi

[7] A. Huang, R. Wu. *Deep Learning for Music.* 2016, arxiv: http://arxiv.org/abs/1606.04930.

[8] H. W. Dong, W. Y. Hsiao, L. C. Yang, Y. H. Yang. *MuseGAN: Multi-Track Sequential Generative Adversarial Networks for Symbolic Music Generation and Accompaniment.* In Thirty-Second AAAI Conference on Artificial Intelligence, 2018.

[9] H. Hung, C. Wang, Y. Yang and H. Wang, "Improving Automatic Jazz Melody Generation by Transfer Learning Techniques," 2019 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC), Lanzhou, China, 2019, pp. 339-346, doi: 10.1109/APSIPAASC47483.2019.9023224.

[10] https://github.com/GiadaSimionato/Analysis_and_Comparison_of_Deep_Learning_Methods_for_Jazz_Music_Generation.