Symbolic-Domain Music Generation with GANs

Scantamburlo Mattia†, Piai Luca†, Chinello Alessandro†

***Abstract*—Recent advances in generative models have made the automated production of music an important area of deep learning research. This paper presents a simplified Generative Adversarial Network (GAN), inspired by MidiNet, for symbolic music generation using the MAESTRO dataset. Its importance is in the fact that it demonstrates that a minimal, interpretable model can achieve stable and musically coherent results by addressing practical training challenges like mode collapse and non-convergence using techniques like minibatch discrimination and hyperparameter tuning. The main result is a successful training process, achieved by adjusting learning rates and update steps. This enables the generator to produce piano roll melodies without collapsing. This work provides a reproducible baseline that can be used as a good practical starting point for other experimental research in music generation.**

***Index Terms*—Unsupervised Learning, Convolutional Neural Networks, General Adversarial Networks, Music generation, symbolic representation.**

I. Introduction

The automated production of music has always been a topic of interest when considering all possible applications of deep learning. Recent advances in the field, especially in generative models, have enabled machines to produce melodies and harmonies. This relatively new ability allows for some new implications in the music production field, making automatic music generation an important and growing area of research.

A major challenge in symbolic music generation is both in modeling the short-term coherence (e.g., note transitions) and modeling long-term structure (e.g., melody evolution) of a musical piece. Generative Adversarial Networks (GANs) have been applied in the last years to music generation with some interesting results. Relevant works in this direction include museGan [1] for multi-track generation, and midiNet [2], a convolutional GAN that generates one bar at a time using conditioning mechanism. MidiNet introduced CNNs over RNNs (recurrent neural network) for symbolic music, it works on conditioning networks to obtain temporal coherence across bars. This paper is an extension of the MidiNet with the objective of introducing data on a real implementation of that architecture with some adjustment for practical problems.

In this work, we address the problem of symbolic music generation by proposing a simplified Generative Adversarial Network (GAN) model trained on the MAESTRO dataset [3], a collection of MIDI recordings of classical piano performances. The main challenge we faced was in designing a model that can generate musically coherent outputs while avoiding typical GAN issues like mode collapse, instability, and non-convergence. These problems, while being well known in the literature, are often not analyzed in practical cases.

For these reasons, we designed a GAN architecture inspired by MidiNet [2], but simpler and more interpretable. Our implementation integrates practical strategies such as minibatch discrimination and fine-tuned hyperparameters to mitigate training instabilities. The model operates directly on piano rolls representations of MIDI sequences. The approach is motivated by the need for models that can be implemented more easily in real-world environments.

Although the architecture does not introduce any novel components, its contribution is in demonstrating that, with appropriate training practices, a minimal GAN model can still generate music with a some degree of structure and coherence. We've monitored the learning process using generator loss and the confidence score , which gave us a feedback on the convergence of the model and whether or not the discriminator was "convinced" about its evaluations of the provided samples. So the relevance of this work is mostly practical: it provides an interpretable base for symbolic music generation, that can also be extended for further experimental research.

Our main contributions can be summarized as follows: We developed a GAN model for symbolic music generation using the MAESTRO dataset. We implemented a complete MIDI preprocessing pipeline that converts sequences into pianoroll matrices, enabling GAN training without being conditioned on the actual dataset structure. We monitored training progress using generator loss and the discriminator confidence score, providing insight into training quality (how much the model was learning) and convergence behavior. We evaluated the generated samples through inspection.

Here we ask to keep in mind that the criteria used to evaluate were merely subjective, since none of the authors know much about the musical world.

All of this was done in the most efficient and simpler possible way. Any reader can access the code at our GitHub[1] and train a simple GAN architecture for music generation themselves.

This report is structured as follows. In Section II, we review related work in neural and GAN-based music generation.

†Department of Information Engineering, University of Padova, email: mattia.scantamburlo.1@studenti.unipd.it, luca.piai.1@studenti.unipd.it, alessandro.chinello.2@studenti.unipd.it

[1]https://github.com/Ultimi-Sumiti/GAN_Music_Generator

Section III describes the MAESTRO dataset and the pre-processing methods. Section IV presents the network signals and features. Section V details the training procedure and evaluation metrics. In Section VI, we analyze the results and discuss the generated outputs. Finally, Section VII concludes the report and highlights future research directions.

II. Related Work

Symbolic music generation in the literature has traditionally been approached with Recurrent Neural Networks (i.e. melodyRNN [4]), especially LSTMs, thanks to their specific predisposition in modeling sequential data. These models could generate locally coherent melodies, but often failed to catch global musical structure, this leads to repetitive output. Also, these models required some previous information about musical scale and about the melody to generate itself.

To overcome these limitations, Generative Adversarial Networks (GANs) were introduced. MidiNet was one of the first GAN-based models for symbolic music, using a convolutional generator and discriminator to produce melodies conditioned on chord progressions and previous bars. This approach improved perceived harmony and diversity, but was limited in length and polyphonic expression (amount of notes used).

Later works extended GANs to multi-track generation (MuseGan [1]), but lack of long-term structure remained a key problem. In recent years, Transformer-based models such as Music Transformer (see [5]) have become the state of the art, thanks to their ability to capture long range dependencies using a concept called self-attention. These models allow for the generation of structured and coherent music.

III. Processing Pipeline

Below we describe each block of the high level processing pipeline shown in Figure 1. We have developed three architectures with increasing levels of complexity. Throughout this report, we refer to them as **Model v1**, **Model v2** and **Model v3**.

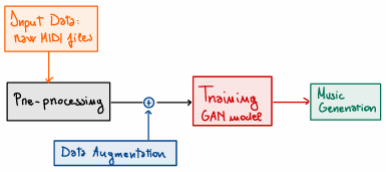

Fig. 1: Processing pipeline.

A. Input data

The pipeline starts with the raw input data that in our case are MIDI files. Each file is divided in bars and each of them is represented with a piano roll. A piano roll $\boldsymbol{X}$ is a $h \times w$ matrix, where h denotes the number of MIDI notes and w is the number of time-steps we use in a bar. In our case we have considered $h = 128$ and $w = 16$. Since we omit the velocity of the note events, the matrix is binary $\boldsymbol{X} \in \{0, 1\}^{h \times w}$.

B. Pre-processing

Pre-processing is used to extract from each piano roll the melody and the associated chord. A melody is nothing but a binary matrix in which only one of the h possible notes is active in each time step. A chord is instead encoded in a vector. The output of this step is a dataset and how it is defined depends on the model we are dealing with.

- For **Model v1** the dataset is composed by melody bars extracted from the MIDI files.
- For **Model v2** the dataset is composed by pairs of consecutive melody bars taken from the same MIDI file.
- For **Model v3** the dataset is composed by triplets: a pair of consecutive melody bars (as for the previous model) and a chord.

C. Data Augmentation

Data augmentation is performed only for the dataset of Model v2 and Model v3. We have used the same strategy for both. Even though the original dataset contained a large number of samples, our experiments showed that selecting a smaller subset and applying data augmentation resulted in better-quality outputs than using a large dataset from the beginning.

D. Training GAN

All the three models are trained using the Generative Adversarial framework, which will be described in detail in later sections. Here, we briefly outline the differences between the three models:

- In **Model v1**, the generator is trained to produce a melody given a noise vector as input.
- In **Model v2**, the generator is trained to produce a melody given both a noise vector and a conditioning melody bar.
- In **Model v3**, the generator is trained similarly to previous model, but with the addition of the chord associated with the conditioning bar as part of the input.

E. Music generation

After training, music can be generated by using the trained model. Each model doesn't directly generate music, instead it generates a piano roll (an image) which can be converted into a MIDI file afterwards.

Given an input, all three models generate as output a single bar melody. With Model v2 and Model v3, longer melodies can be created by using each previously generated melody bar as input to generate the next one.

In details, the generation of longer melodies is performed in the following way: the first bar is sampled randomly from the dataset; the second bar is then generated by the generator, conditioned on the sampled first bar; the third bar is generated by conditioning on the second bar; and so on until a total of 8 bars are produced. Finally, all bars are concatenated into single piano roll.

Additionally, since the output of Model v3 is conditioned on a chord, we manually added the chord after the creation of the melody.

Fig. 2: Example of a generated melody and the same melody with chord added (in yellow).

IV. Signals and Features

In this section, we will clarify how signals and data have been processed, in order to be fed to the generative adversarial network (GAN) models. Also, some information on how the dataset was split, normalized, and on how manipulation operations were performed.

A. Input file operations

In our dataset we have midi file of variable dimension in different directories. So to make all data uniform in this sense, we decided to actuate a split of the midi file into smaller files of fixed dimension (in terms of amount of bars in the files). So after this operation, we got a list of 8 bars long midi files from all the initial midi files, this allowed to randomly select data from all years, effectively incorporating some samples from all the smaller datasets. This operation also allowed us to select precisely the dimension of our dataset. After some training sessions, we understood that a good dimension was about fifty thousand samples. In all of the latter tests we sticked to this size.

B. Piano Roll Operations

As previously mentioned, after processing the input MIDI files, we converted them into piano roll representations. Piano rolls are more suitable for matrix-like manipulations, making them ideal for our preprocessing and augmentation tasks. The main operations performed at this stage included pause removal, melody extraction, octave normalization, and data augmentation.

First, we removed pauses, which were present in nearly all the input data. To achieve this, we extended either the preceding note (if the pause occurred mid-song) or the following note (if the pause was at the beginning of the song) to fill the gap.

Also we extract the melody from the full piano roll, we did this by removing the chords, leaving only one note per time step. To obtain this kind of piano roll we selected the note with highest velocity among all in each frame.

Next, we normalized the octaves. This step, inspired by the methodology in the reference paper (midiNet [2]), involved collapsing all pitch values into a fixed range of two octaves-from MIDI note 60 to 83 (corresponding to C4 to B5 in the piano). Notes falling outside of this range were shifted up or down accordingly. This normalization was crucial for detecting mode collapse in the symbolic MIDI representations generated later.

Finally, we applied data augmentation by circularly shifting all notes in the piano roll up by semitone steps. This process was repeated 12 times (once for each semitone in an octave). When a shifted note exceeded the upper bound of the valid range, it was placed around to the bottom of the range, maintaining all notes within the defined interval.

C. Network inputs operations

Remembering that our project includes 3 different models we have to take into account that for all the models we have different inputs. For the first model the input was just a bar, so all the processing needed to get a valid input was to extract bars from the midi files and do all the piano roll operations. For the second model we needed to feed both a current bar and a previous bar to the conditioner GAN (2D conditions), this was slightly more difficult but easily achieved by creating samples made out of sequential pairs, coming from the same MIDI file. For the third model the generator was also conditioned on chords (1D conditions), so we needed to add to sample pairs one more element obtaining triplets. For one sample, which is made out of 2 bar and a chord, the chord is related to the previous bar and is unique, in particular it is chosen as the most present chord among all time steps of the previous bar.

V. Learning Framework

In this section, we describe the three architectures that we have introduced in the previous sections, namely: Model v1, Model v2, Model v3. All three versions share a very similar structure, although they are designed to address different tasks. We start with the simplest one as it serves as base for the other models. The structure diagram of the generator and discriminator network is shown respectively in Figure 3 and in Figure 4.

A. Model v1

The task that this version aims to solve is to generate one-bar-long melodies. Neither the generator nor the discriminator have any additional information. The structure of model_v1 is a simple DCGAN.

The generator takes as input random vectors $\boldsymbol{z} \in \mathbb{R}^{100}$ of white Gaussian noise. Each $\boldsymbol{z}$ goes through two fully-connected layers composed by 1024 and 256 neurons respectively. The resulting output tensor of shape $[256]$ is then reshaped to $[128 \times 1 \times 2]$. Next, the output flows through four transposed convolution layers:

- The first 3 use 128 kernels of size $[1 \times 2]$, stride 2, no padding and dilation 1.
- The last layer uses a single kernel of size $[128 \times 1]$, stride 1, no padding and dilation 1.

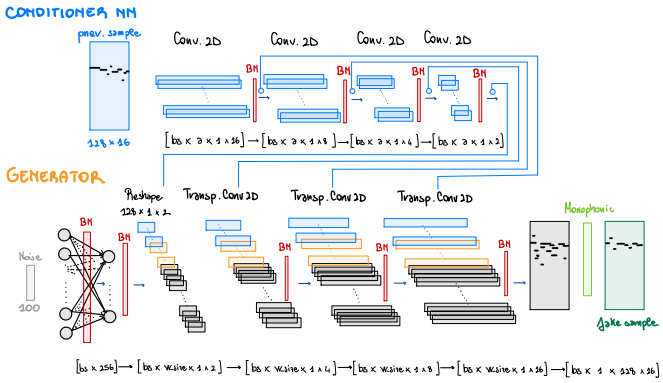

Fig. 3: The activation function used is the `LeakyReLU()`; the red layers apply `BatchNorm`.

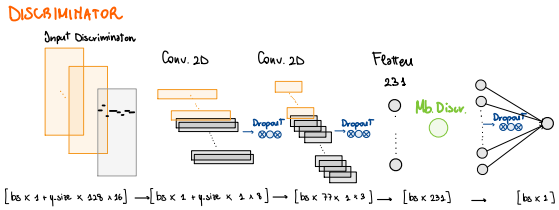

Fig. 4: The light green circle indicates `Minibatch Discrimination`, and the triple blue neuron indicates a `Dropout` layer.

The output is a tensor of shape $[1 \times 128 \times 16]$, which is then passed through a *Monophonic layer*. This layer ensures monophony by keeping, at each time step, only the note with the highest activation (set to 1), while all others are set to 0. In this way, the generator produces a consistent melody bar in the form of a piano roll.

Note that the Monophonic layer applies a threshold and this poses a problem during backpropagation. To overcome this issue we used the Straight-Through Estimators trick [6] that effectively allow us to apply the threshold during forward pass and bypass it during the backward pass.

The discriminator consists of two convolutional layers, each with 14 kernels of size $[128 \times 2]$, stride 2, no padding, and dilation 1, followed by a fully connected layer with 231 neurons and a fully connected layer with 1024 neurons. If mini-batch discrimination is enabled, the size of the first fully connected layer may exceed 231 (this is a tunable parameter). Whether to use mini-batch discrimination is controlled by a Boolean hyper-parameter of the GAN, and its impact will be discussed in a later section.

B. Model v2

The second version is a DCGAN that is obtained starting from the first version and adding the conditioner network. By conditioning the generation of the bar with the previous one, the generator is pushed to produce musically coherent melodies.

As described in the reference paper, the conditioner is a CNN that can be viewed as a reverse of the generator CNN. The two CNNs have the same filter shapes so that the output of each layer of the conditioner can be correctly concatenated to the corresponding transposed convolutional layer of the generator. In this way the generated output bar is influenced by the previous bar. This is illustrated in Figure 3 by the blu blocks.

C. Model v3

The third and last version is obtained by passing the chord condition to both generator and discriminator. Chords are represented as 13 dimensional vectors, where the first 12 dimensions denote the key, the last one denote the chord type (i.e. major or minor). The chord condition is concatenated along the feature map axis in each intermediate convolutional layer. This is achieved by reshaping the 13-dimensional chord vector to match the spatial dimensions of the feature maps before concatenation. This is illustrated in Figure 3 and Figure 4 by the orange blocks.

D. Train the models

Despite their different inputs, all three versions are optimized with respect to the following $\min\max$ game:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{X}\sim p_{\text{data}}(\boldsymbol{X})}[\log(D(\boldsymbol{X})] + \mathbb{E}_{\boldsymbol{z}\sim p(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))] \quad (1)$$

where $\boldsymbol{X}$ is sampled from real data and $\boldsymbol{z}$ is sampled from a random distribution. The output of the discriminator is defined within the range $[0, 1]$; it is close to 1 if the discriminator believes the input sample is real, and close to 0 if it believes the sample is fake.

As mentioned in the reference paper, we also include feature matching terms when optimizing the generator G:

$$\lambda_1 ||\mathbb{E}[\boldsymbol{X}] - \mathbb{E}[G(\boldsymbol{z})]||_2^2 + \lambda_2 ||\mathbb{E}[f(\boldsymbol{X})] - \mathbb{E}[f(G(\boldsymbol{z}))]||_2^2 \quad (2)$$

where f denotes the first convolution layer of the discriminator. In doing so, the distributions of real and generated samples are enforced to be more similar.

VI. Results

In this section we present the numerical results obtained. We will focus on the most important hyper-parameters, i.e. the ones that have the greatest impact on the stability of the $\min\max$ game.

The discussion is kept general and does not target a specific version of the model (i.e., v1, v2, or v3), as the observations apply to all of them. The plots shown were used to monitor the learning behavior of the generator and discriminator during training. However, they do not provide any direct insight into the quality of the melody bars generated by the generator.

We encourage the reader to assess the quality of the generated melodies by listening to the examples available on the project's GitHub repository. There, one can find the best samples we have collected, generated by models trained on different datasets.

The most important hyper-parameters of the network are:

1) A boolean flag that determines whether mini-batch discrimination is applied.
2) The learning rates of the discriminator and the generator.
3) The number of update steps per iteration for both the discriminator and the generator.

While GANs are powerful models, they can be rather difficult to train. The two common problems that we have faced are: non-convergence and general instability caused by the powerfulness of the discriminator. Among the various tricks proposed in the literature [7], we found that, in our case, mini-batch discrimination and one-sided label smoothing have a significant impact on reducing mode collapse.

In Figure 5 there are two examples of mode collapse. In the first figure it is very clear: the generator is looping over different modes but does not converge to a good representation. The second figure is misleading, as it appears that the $\min\max$ game is balanced. However, the trained model suffered from partial mode collapse, consistently generating very similar melodies (see Figure 6).

Regarding the balance between the generator and the discriminator, we found that the combination of learning rate parameters and the number of updates per iteration step plays a crucial role. The learning rate of the generator should be at least double that of the discriminator. If this is not sufficient, one should consider further reducing the discriminator's learning rate and increasing the number of generator updates per iteration. In our study, we observed that updating the discriminator once and the generator two or three times per iteration provides good results.

In Figure 7, we observe two characteristic plots that typically arise when the discriminator is significantly stronger than the generator. The game remains balanced up to iteration 2000; after that, the discriminator dominates, as indicated by the decreasing discriminator loss and the increasing generator loss. This trends never reverses and as a result, after training, the generator fails to produce good samples.

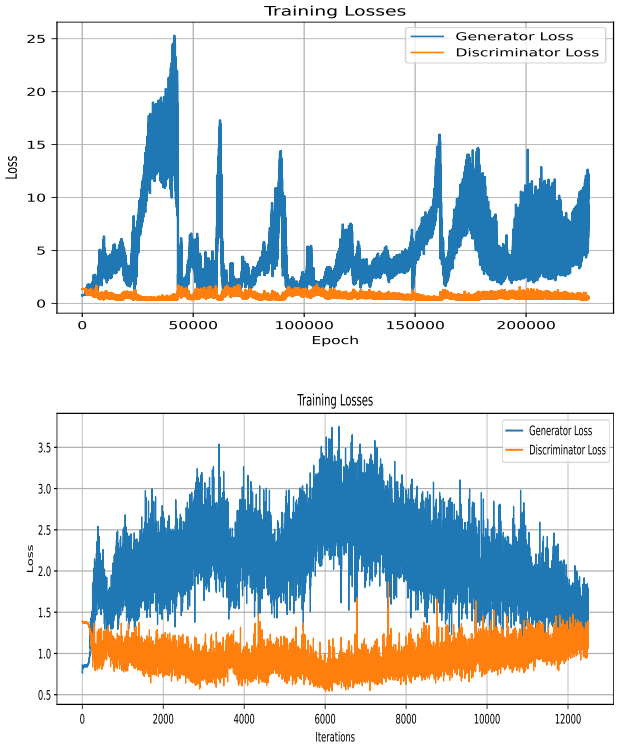

Fig. 5: Mode collapse.

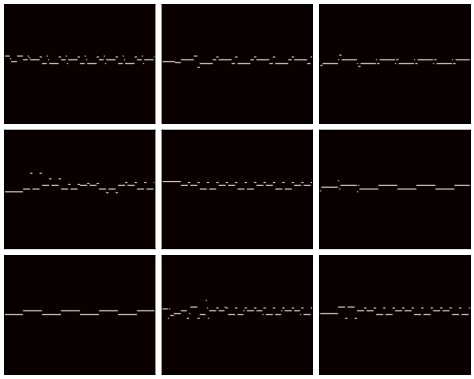

Fig. 6: Partial mode collapse.

In the second figure, we monitor how the discriminator's confidence evolves over the training iterations. The confidence is computed separately for real and fake samples. Consider a minibatch of m samples from the training dataset, denoted as $\{\boldsymbol{X}_i\}_{i=1}^m$ and a minibatch of m samples from prior $p(\boldsymbol{z})$, deonted as $\{\boldsymbol{z}_i\}_{i=1}^m$. The real confidence is defined as

$$C_r = \frac{1}{m}\sum_{i=1}^{m} D(\boldsymbol{X}_i), \tag{3}$$

instead the fake confidence is defined as

$$C_f = 1 - \frac{1}{m}\sum_{i=1}^{m} D(G(\boldsymbol{z}_i)). \tag{4}$$

Both C_r, C_f lie within the range $[0, 1]$. Ideally, we want to

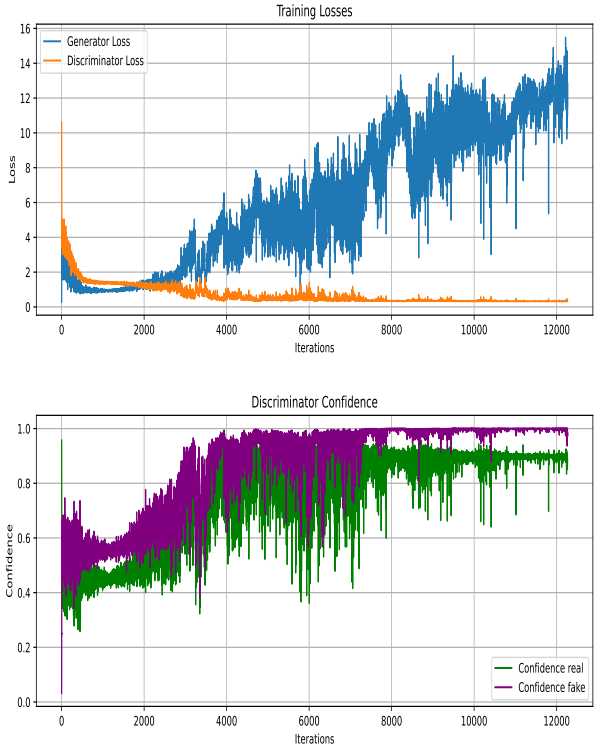

Fig. 7: Example of a non-balanced game.

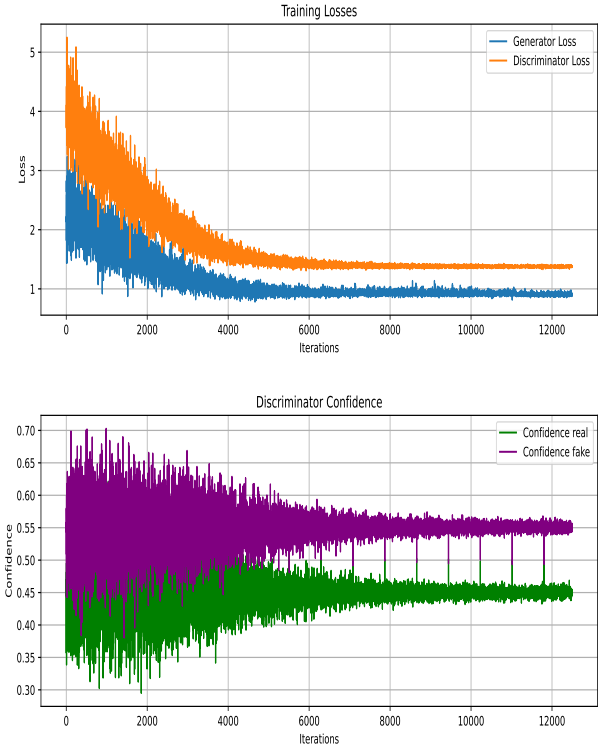

Fig. 8: Example of a balanced game.

have both at 0.5, which would indicate that the generator has found a way to fool the discriminator. In the above confidence plot, we observe that from iteration 1 to 2000, both C_r and C_f approach 0.5, as desired. However, after iteration 2000, the discriminator begins to dominate, causing both scores to converge to their optimal values[2] (from discriminator point of view): $C_r = 0.9$ and $C_f = 1$.

In Figure 8 is shown an example in which the hyper-parameters are correctly tuned. By looking at the two losses we can observe that the $\min\max$ game has reached an equilibrium point in which neither player prevails over the other. Furthermore, the discriminator's confidence converges to a stable point with $C_r \approx 0.45$ and $C_f \approx 0.55$. In such a scenario, we have good reason to believe that the generator is producing high-quality samples that are similar to the real ones. The plots above were obtained by setting the generator learning rate to $3 \cdot 10^{-4}$, the discriminator learning rate to 10^{-5}, updating the generator three times per iteration and the discriminator only once, and applying mini-batch discrimination.

VII. Concluding Remarks

In this report we have explained how to effectively use a GAN architecture to generate symbolic music. We demostrate how to extract melodies from MIDI files, augment the dataset, and train slightly different architectures to generate different types of melodies. The architecture of all the three models is very simple and relatively small. Additionally, the preprocessing pipeline we designed allows for the creation of datasets smaller than 5MB. This was a key factor, as it enabled us to pre-load the entire dataset into GPU memory, significantly accelerating the training process. As a result, users can easily switch datasets and train any of the three models without modifying the code.

We tried to implement a fourth model that was supposed to cover the task of musical generation (both melody and chords), we tested two different approaches but none of them achieved good quality. To extend our work we suggest tackling this task, potentially by exploring entirely different architectures, such as Variational Autoencoders (VAEs) or diffusion models.

A. *Problems faced and what we have learned*

During the development of this project we face some problems related to the training part. We learned that GANs are not easy to train since it is rather difficult to find a combination of parameters that leads to convergence. Small changes can cause mode collapse and/or non-convergence. To try to solve this problem we implement other tricks that we didn't mention in the report, such as WGAN with gradient clipping and also adding noise to both fake and real images before passing them to the discriminator.

Another difficulty that we faced was the lack of a metric that effectively measures the quality of the melodies created by the generator. During the training of Model v1 and Model

[2]The optimal value for the real confidence is 0.9 because real labels are smoothed by a factor of 0.1.

v2 we plot intermediate images created by the generator, but those do not provide much information about the quality of the melody, they rather give information about mode collapse.

REFERENCES

[1] H.-W. Dong, W.-Y. Hsiao, L.-C. Yang, and Y.-H. Yang, "Musegan: Multi-track sequential generative adversarial networks for symbolic music generation and accompaniment," 2018.

[2] L.-C. Yang, S.-Y. Chou, and Y.-H. Yang, "Midinet: A convolutional generative adversarial network for symbolic-domain music generation," 2017.

[3] C. Hawthorne, A. Stasyuk, A. Roberts, I. Simon, C.-Z. A. Huang, S. Dieleman, E. Elsen, J. Engel, and D. Eck, "Enabling factorized piano music modeling and generation with the MAESTRO dataset," in *International Conference on Learning Representations*, 2019.

[4] G. B. M. Team, "Melodyrnn: Generating melodies with recurrent neural networks," 2016. https://magenta.withgoogle.com/2016/07/15/lookback-rnn-attention-rnn.

[5] C.-Z. A. Huang, A. Vaswani, J. Uszkoreit, N. Shazeer, I. Simon, C. Hawthorne, A. M. Dai, M. D. Hoffman, M. Dinculescu, and D. Eck, "Music transformer," 2018.

[6] H. Askary, "Intuitive explanation of straight-through estimators with pytorch implementation," 2023.

[7] I. J. Goodfellow, "NIPS 2016 tutorial: Generative adversarial networks," *CoRR*, vol. abs/1701.00160, 2017.

Project contribution

For the project, we have divided the workload equally among the three of us. In particular: Alessandro Chinello read the architectural details through the paper, projected and implemented the raw neural networks of all the three models; Luca Piai worked mostly in the creation of the dataset and the development of an easily manipulable version of data, also he worked on the conversion of the data after pre-processing into tensors; Mattia Scantamburlo instead developed a pre-processing pipeline, also searching in the main paper and other notorious papers how to correctly process the data to achieve convergence and avoid typical training problems and instabilities.

In the training phase, all the authors have collaborated developing architectural correction from the initial model, like mini-batch discrimination or dropouts, but also testing the importance and correct values of network parameters (fine tuning).

Also, the three of us collaborated very strictly during the development of the different components of the project, so it is very likely that in the code initially assigned to one person some functions may have been written by another one of us.