Symbolic Domain Music Generation: a MIDINET implementation in PyTorch

Raffaele Gaudio 2057974, Luca Giorgetti 2045743

May 15, 2025

Abstract

MidiNet, a Deep Convolutional Generative Adversarial Network (DCGAN) for symbolic-domain music generation, was proposed by Yang et al. [2]. This project aims to reimplement the main ideas of the article in PyTorch. Specifically, two models are implemented: Model 0, a basic Generative Adversarial Network (GAN) with a Convolutional Neural Network (CNN) as the generator, and Model 1, an enhanced GAN that employs two CNNs in the generator to introduce correlations between a bar and the previous one.

1 Introduction and an overview on the original article

Music generation in the symbolic domain has been a popular research topic in recent years, leveraging advances in neural networks to create realistic and diverse compositions. Among notable contributions is MidiNet, introduced by Yang et al. [2], which proposes a Deep Convolutional Generative Adversarial Network (DCGAN) for melody generation. Unlike earlier models that predominantly use Recurrent Neural Networks (RNNs), MidiNet employs Convolutional Neural Networks (CNNs) to generate melodies in a bar-by-bar manner. The model integrates a conditional mechanism that allows it to adapt to various input contexts, such as previous bars or chord progressions, making it versatile and effective in generating coherent and creative music.

MidiNet's architecture consists of a generator CNN that transforms random noise into musical patterns and a discriminator CNN that distinguishes real melodies from generated ones. A unique "conditioner" component enables the model to incorporate prior knowledge, such as a priming melody or chord sequence, to guide the generation process. The resulting framework demonstrates that convolutional architectures can effectively model temporal dependencies in symbolic music, offering a viable alternative to RNN-based designs.

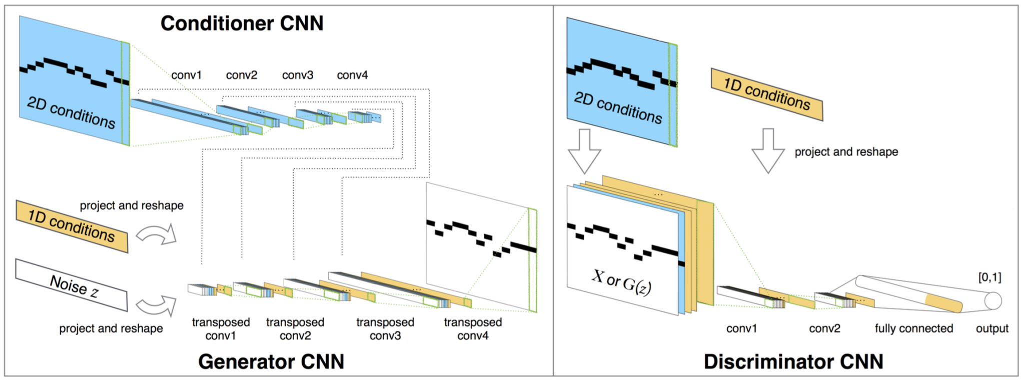


Figure 1: System diagram of the original MidiNet model for symbolic-domain music generation.

2 Methods

In this section the pricipal elements of the MidiNet scheme are presented. A system diagram of the original model is presented in Figure 1.

2.1 Symbolic Music Representation

Unlike audio files, Musical Instrument Digital Interface (MIDI) files store symbolic musical data, allowing for precise representation and manipulation of musical compositions.

MIDI files are organized into multiple channels, where each channel corresponds to an instrument. Each channel contains an ordered sequence of musical notes, often referred to as *the piano roll*. Each note is associated with its duration and intensity (velocity). In addition to note information, MIDI files provide metadata that is invaluable for analysis and generation tasks, including:

- Tempo: Measured in beats per minute (BPM),
- Key signature: Indicating the tonal center,
- Time signature: Specifying the rhythmic structure.

The piano roll can be represented as a matrix $\boldsymbol{X}$ of shape $M \times N$, where M denotes the number of unique pitches (e.g., MIDI note numbers) and N represents the time steps. Each entry in the matrix indicates the presence of a note at a specific pitch and time. The values can also encode the velocity of the notes, providing additional expressive detail. In our implementation, we simplified the representation by omitting velocity, using binary matrices $\boldsymbol{X} \in \{0,1\}^{M\times N}$. Additionally, we standardized the time length N by dividing each MIDI file into bars.

2.2 Generator CNN and Discriminator CNN

The core of MidiNet is a modified DCGAN that aims to train a discriminator, D, to distinguish between real and generated data, and a generator, G, to "fool" the discriminator. As is typical in GANs, the input to G is a vector of random noise, $\boldsymbol{z} \in \mathbb{R}^L$, while the output is an $M \times N$ matrix, $\hat{\boldsymbol{X}} = G(\boldsymbol{z})$, which is designed to appear as real data to D. The training objective for GANs involves solving the following minimax optimization problem:

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

where $\boldsymbol{X} \sim p_{\text{data}}$ denotes sampling from the real data distribution, and $\boldsymbol{z} \sim p_z(\boldsymbol{z})$ represents sampling from a random noise distribution. Training G and D requires iterative optimization to gradually improve the performance of G.

The discriminator, D, is a CNN comprising several convolutional layers followed by fully connected layers. These layers are optimized using a cross-entropy loss function, ensuring that the output of D approaches 1 for real data (i.e., $\boldsymbol{X}$) and 0 for generated data (i.e., $G(\boldsymbol{z})$). A sigmoid activation function is used at the output layer of D, constraining its output to the range $[0,1]$.

The generator, G, aims to produce outputs that make D classify them as real (i.e., output close to 1). To achieve this, G transforms the noise vector $\boldsymbol{z}$ into a matrix $\hat{\boldsymbol{X}}$ through a series of fully connected layers followed by transposed convolution layers. These transposed convolutions effectively "upsample" smaller vectors or matrices into larger ones, forming the desired output.

Due to the adversarial nature of GAN training, issues such as instability and mode collapse are common. To address these challenges, MidiNet incorporates two techniques: feature matching and one-sided label smoothing. Feature matching involves adding additional L_2 regularization terms to Equation (1) to encourage the distributions of real and generated data to be similar. Specifically, the following terms are included during the optimization of G:

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

where f denotes the first convolutional layer of the discriminator, and λ_1, λ_2 are empirically determined parameters.

2.3 Conditioner CNN

In MidiNet, the conditioner component incorporates prior knowledge to guide the music generation process. This approach differs from traditional GAN-based image generation, where a 1-D conditioning vector is used. For symbolic music, 2-D conditions are needed, represented as matrices that encode musical information, such as melodies from previous bars.

The conditioning matrix is of size $M \times N$, where M corresponds to the number of possible pitches and N represents the number of time steps within a bar.

To utilize these 2-D conditions in the generator, MidiNet employs a dedicated conditioner CNN, which processes the $M \times N$ conditioning matrix through a series of convolutional layers. The conditioner is designed to produce feature maps that are shape-compatible with the transposed convolution layers of the generator: this compatibility enables the outputs of the conditioner to be concatenated with the intermediate layers of G influencing the generation process.

In the discriminator, the conditioning matrix is added directly to the input layer, where it influences all subsequent layers. This allows the discriminator to evaluate the coherence of the generated melodies with respect to the conditioning input.

To ensure compatibility between the conditioner and the generator, the two networks share the same convolutional filter shapes. During training, both the conditioner CNN and G are optimized simultaneously, sharing gradients to align their objectives. This joint training enables the conditioner to effectively guide the generator in producing melodies that are consistent with the provided conditions, such as the melodic content of previous bars.

3 Implementation

In this section we discuss our implementation of the MidiNet model in PyTorch. After briefly describing the dataset and the preprocessing, we give details of the two models we implemented: Model 0 and Model 1.

3.1 Dataset

We utilized the MAESTRO v3.0.0. dataset [1], which comprises approximately 200 hours of virtuosic piano performances encoded as mono-channel MIDI files. For the melody generation purposes, we first extracted the melody from each MIDI file by filtering only the notes with the highest pitch at each time step. This strategy was adopted because, in piano music, the right hand (which typically plays higher-pitched notes) often carries the melody, while the left hand primarily plays the harmonic accompaniment or chords. Using this approach, and shifting all the matrices in the same range of notes, we reduced the row dimension of the piano roll from 128 to 87.

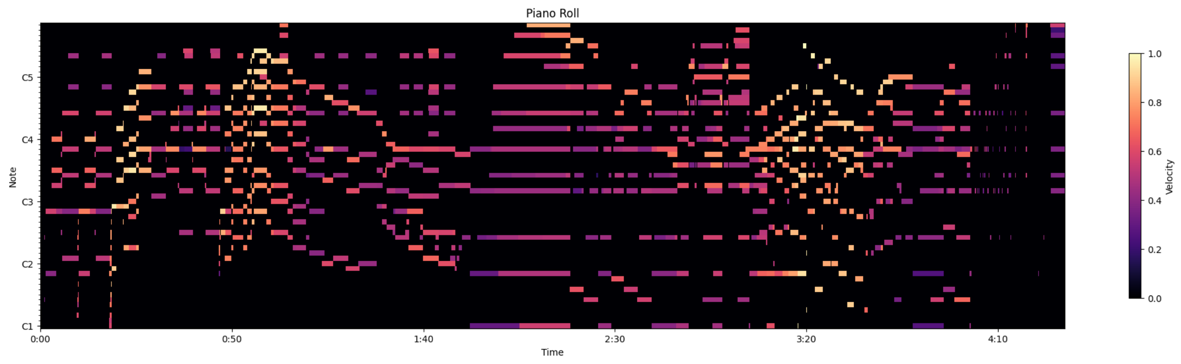
Following the approach described in the article, we fixed the smallest note unit to be the sixteenth note. Notes with shorter durations were eliminated, creating pause gaps (i.e., zeros in the piano roll matrix). Subsequently, notes followed by pauses were prolonged to fill these gaps. If the first note in a file was a pause, the subsequent note was extended to ensure the melody played as the file began. Next, we segmented each file into groups of eight bars, resulting in a standardized duration of 1600 time steps per sample (100 time steps per second). For simplicity, the velocity of all notes in the dataset was fixed to 1. The final dataset consisted of 44109 binary matrices, each of shape (87, 1600).

3.2 Model 0 Specifications

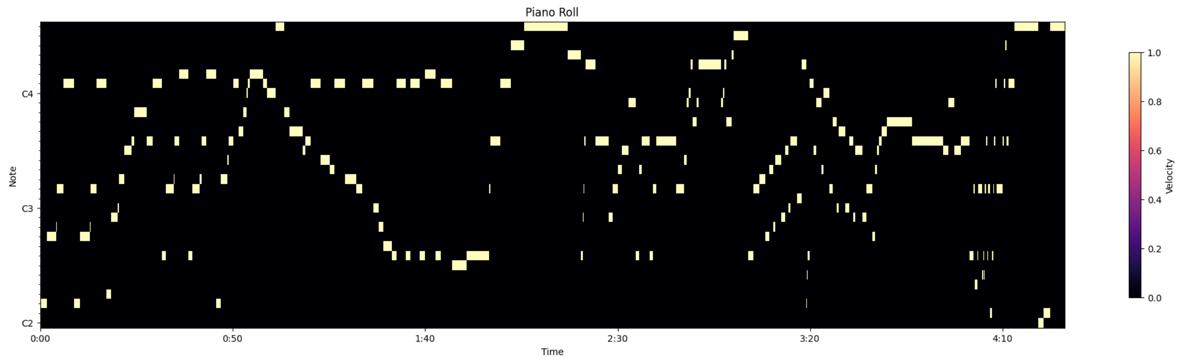
The first implemented model, referred to as Model 0, is a GAN. The generator takes as input random vectors of white Gaussian noise of length $L = 100$. Each random vector passes through two fully connected layers in the generator:

- The first layer maps the noise vector to 512 neurons, applying ReLU activation.
- The second layer maps the output to dimensions $40 \times 5 \times 12$ and reshapes it into a 4D tensor of shape (`batch size` $= 128, 40, 5, 12$).

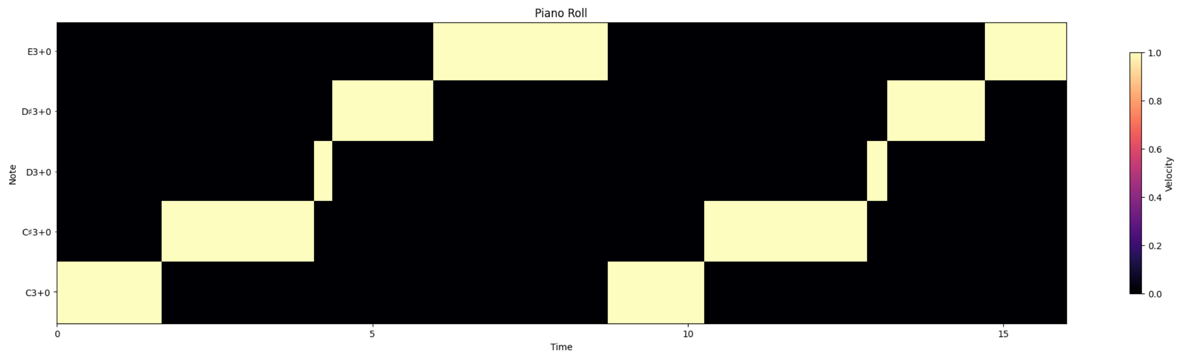
A sequence of transposed convolutional layers progressively upsamples the tensor to the target piano roll dimensions of $(1, 87, 1600)$:



(a)



(b)



(c)

Figure 2: An example of the MIDI preprocessing pipeline. (a) Original MIDI file from the MAESTRO dataset. (b) Melody extracted after applying preprocessing steps: filtering high pitch notes, fixing the minimal note duration, removing pauses, and normalizing velocity to unity. (c) The first 8 bars of the processed melody. For clarity, only the rows corresponding to active notes are shown in all three figures.

- **G conv layer 0**: 30 output channels, kernel size: $(2, 5)$, stride: $(2, 3)$.
- **G conv layer 1**: 20 output channels, kernel size: $(3, 5)$, stride: $(2, 2)$.
- **G conv layer 2**: 10 output channels, kernel size: $(3, 9)$, stride: $(2, 5)$.
- **G conv layer 3**: 1 output channel, kernel size: $(3, 8)$, stride: $(2, 4)$.

ReLU activation is applied after each transposed convolution. The final output is adjusted to match the binary matrix representing of the piano roll via a Gumbel softmax layer, which ensures differentiability. For each column, all values are set to zero except for the entry with the maximum activation, which is set to one. This guarantees that only one note is active per time step.

The discriminator comprises two convolutional layers followed by two fully connected layers:

- **D conv layer 0**: 64 output channels, kernel size: $(87, 1)$, stride: $(1, 1)$.
- **D conv layer 1**: 128 output channels, kernel size: $(1, 2)$, stride: $(8, 8)$.

The first convolutional layer captures pitch-related patterns across all notes, while the second reduces temporal resolution for computational efficiency. ReLU activation is applied after both layers. The output of the second convolutional layer is flattened and passed through:

- A fully connected layer mapping the flattened features to 1024 neurons with ReLU activation.
- A final fully connected layer that outputs a single value, representing the probability that the input to the discriminator is real, using a sigmoid activation function.

To stabilize training and encourage the generator to produce realistic outputs, we employed binary cross-entropy loss with one-sided label smoothing. We did not use feature matching instead, departing from the original MidiNet approach. As is typical in GANs, the discriminator tends to overpower the generator, which can lead to the vanishing gradient problem. To mitigate this, we adopted a strategy to weaken the discriminator by updating the generator twice per iteration while updating the discriminator only once.
We built model 0 to generate 8 bars at a time, each one with 200 tempi, as for the sample in the dataset after preprocessing. This allows an easy comparison between the generated bars and the ones used for training.

3.3 Model 1 Specifications

The second implemented model, referred to as Model 1, is an extension of Model 0, introducing a conditioner to condition the generation of a bar to the previous one.
The conditioner behaves differently in the training phase than in the music generation phase. Firstly, in the training phase the training matrices are splitted in 8 bars and a void one has been attached in the beginning. Then, two random consecutive bars are chosen from the group of eight. The first one, which is either void or a real sample, has been fed to the conditioner, the second one has been fed to the Discriminator as real sample.
The generation phase works as follows:

1. The first bar is taken from the training set.
2. The second one is generated conditioned by the previous bar.
3. From the third bar, each one is generated, conditioned by the previous synthetic one. This process goes on until you have 8 bars.

The conditioner takes as input a bar of the dataset and passes it through four convolutional layers:

- **Conv Layer** 0: Input channels: 1, Output channels: 10, Kernel size: $(2, 2)$, Stride: $(2, 1)$, Output: $(10, 43, 100)$.
- **Conv Layer** 1: Input channels: 10, Output channels: 20, Kernel size: $(2, 2)$, Stride: $(0, 1)$, Output: $(20, 21, 50)$.

- **Conv Layer** 2: Input channels: 20, Output channels: 30, Kernel size: $(2, 3)$, Stride: $(2, 2)$, Output: $(30, 10, 25)$.
- **Conv Layer** 3: Input channels: 30, Output channels: 40, Kernel size: $(3, 2)$, Stride: $(2, 2)$, Padding: $(1, 0)$, Output: $(40, 5, 12)$.
- Outputs the result of each convolutional layer: $[(10, 43, 100), (20, 21, 50), (30, 10, 25), (40, 5, 12)]$.

It outputs, the result of each single convolutional layer. Now the generator must output single bars instead of matrices containing 8 of them. Moreover, its structure has to be modified to concatenate the outputs of the conditioner.
The structure of the generator is the following:

- **Fully Connected Layer** 1: Input dimension: 100, Output dimension: 512, Activation: ReLU
- **Fully Connected Layer** 2: Input dimension: 512, Output dimension: $40 \times 5 \times 12$, Activation: ReLU
- Reshape output to $(40, 5, 12)$
- Concatenate with the last output of the conditioner $(40, 5, 12)$, resulting in $(80, 5, 12)$.
- **Deconv Layer** 0: Input channels: 80, Output channels: 30, Kernel size: $(2, 3)$, Stride: $(2, 2)$,Output: $(30, 10, 25)$.
- Concatenate with the third output of the conditioner $(30, 10, 25)$, resulting in $(60, 10, 25)$.
- **Deconv Layer** 1: Input channels: 60, Output channels: 20,Kernel size: $(3, 2)$, Stride: $(2, 2)$, Output: $(20, 21, 50)$.
- Concatenate with the second output of the conditioner $(20, 21, 50)$, resulting in $(40, 21, 50)$.
- **Deconv Layer** 2: Input channels: 40, Output channels: 10, Kernel size: $(3, 2)$, Stride: $(2, 2)$, Output: $(10, 43, 100)$.
- Concatenate with the first output of the conditioner $(10, 43, 100)$, resulting in $(20, 43, 100)$.
- **Deconv Layer** 3: Input channels: 20, Output channels: 1, Kernel size: $(3, 2)$, Stride: $(2, 2)$, Output: $(1, 87, 200)$.
- **Gumbel Softmax**: Applied along the note dimension for one-hot encoding

And the structure of the discriminator:

- **Conv Layer** 1: Input channels: 1, Output channels: 64, Kernel size: $(5, 5)$, Stride: 2, Padding: 2, Activation: ReLU, Output: $(64, 44, 100)$.
- **Conv Layer** 2: Input channels: 64, Output channels: 128, Kernel size: $(5, 5)$, Stride: 2, Padding: 2, Activation: ReLU, Output: $(128, 22, 50)$.
- Flatten to $128 \times 22 \times 50$.
- **Fully Connected Layer**: Input dimension: $128 \times 22 \times 50$, Output dimension: 1, Activation: Sigmoid.

4 Results

Both Model 0 and Model 1 were trained for 100 epochs using a dataset of 8821 samples, with a batch size of 128. Two Adam optimizers were used during training: one for the generator (and the conditioner for Model 1) with a learning rate of 0.02, and another for the discriminator with a learning rate of 0.0002. The momentum parameters for both optimizers were set to $(0.5, 0.999)$. The training losses are illustrated in Figure 3.

The discriminator's loss quickly stabilized around 0.32, indicating its effective ability to distinguish between real and fake data. In contrast, the generator's loss exhibited significant fluctuations, reflecting the inherent instability of GAN training. This is due to the adversarial nature of the training process, where the generator tries to deceive a discriminator that is simultaneously improving. Initially, the generator's high loss is typical, as it begins with random noise and struggles to generate convincing samples. However, over time, the generator's loss continued to increase. This could be attributed to factors such as mode collapse, where the generator fails to produce diverse outputs, or an imbalance in training, with the discriminator overpowering the generator. As a result, the generator's ability to adapt diminishes, leading to an increase in its loss as training progresses.

5 Conclusions and Future works

In conclusion, we implemented in PyTorch the basic GAN structure in Model 0 and the GAN structure with a conditioner in Model 1.

As presented in the Results section, the training outcomes under the given conditions were unsatisfactory. For limited computational resources, we could use only a small portion of the dataset for training, and we trained both the models for only 100 epochs. Therefore, there is a chance that the generator either did not see enough data or was trained for a too short period, or both.

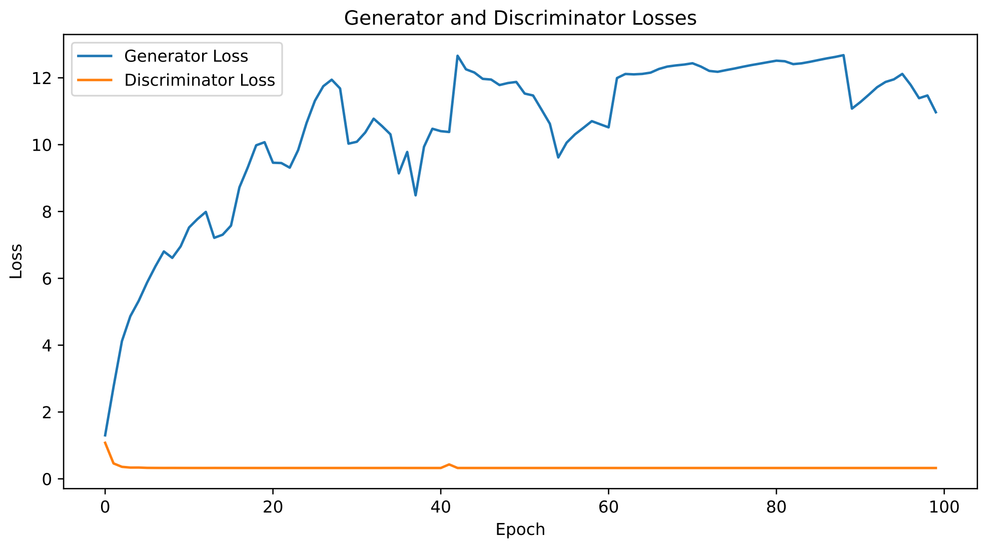
Moreover, the size of the matrices in the training set contributed to increased training complexity. The chosen sampling rate of 100 time steps per second may be excessively high, considering that notes are at least one order of magnitude slower.

Future work could explore the potential of the models using a larger training set and smaller matrix sizes for more epochs. In addition, the use of a grid search could improve the setting of hyperparameters of the optimizers, leading to a more stable training process. Furthermore, the use of batch normalization in the generator could speed up the convergence of the generator.

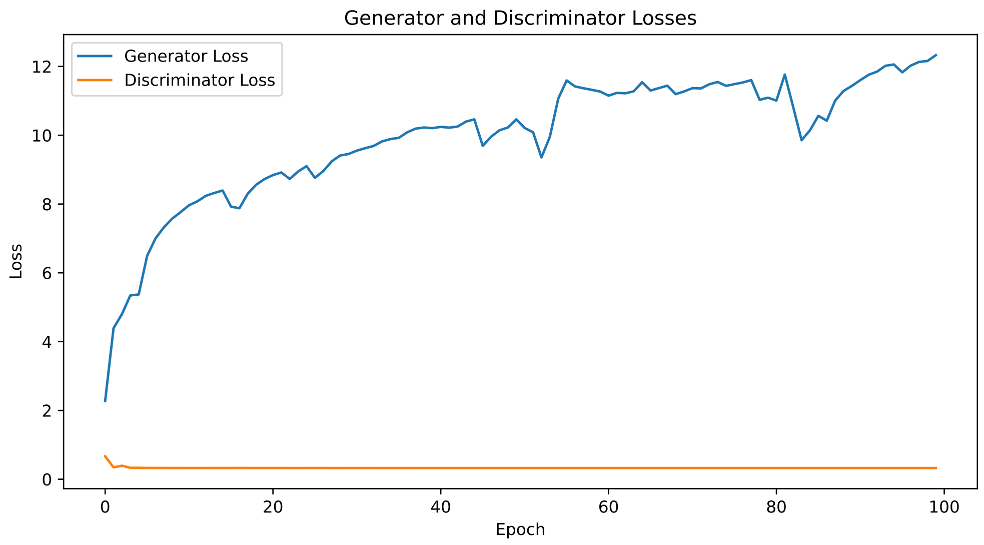
References

[1] Curtis Hawthorne, Andriy Stasyuk, Adam Roberts, Ian Simon, Cheng-Zhi Anna Huang, Sander Dieleman, Erich Elsen, Jesse Engel, and Douglas Eck. Maestro dataset v3.0.0. Magenta Team, Google Research, 2021. https://magenta.tensorflow.org/datasets/maestro#v300.

[2] Li-Chia Yang, Szu-Yu Chou, and Yi-Hsuan Yang. Midinet: A convolutional generative adversarial network for symbolic-domain music generation. *arXiv preprint arXiv:1703.10847*, 2017.

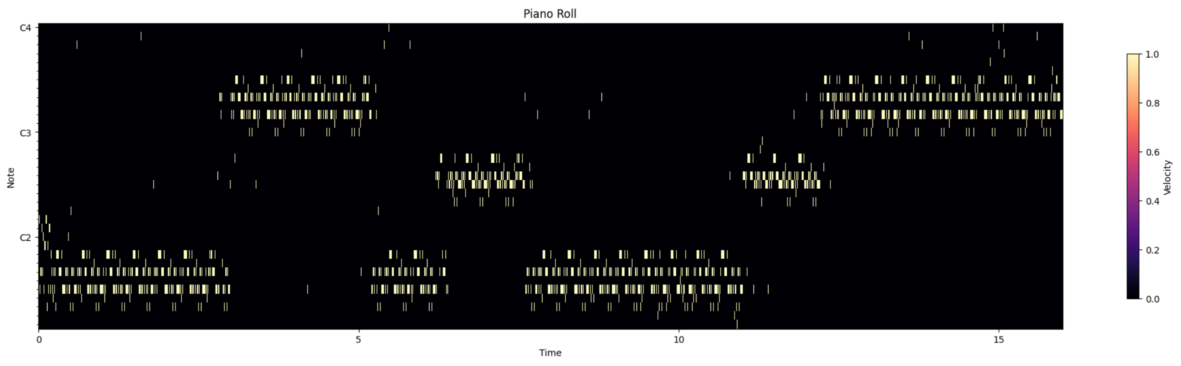


(a)

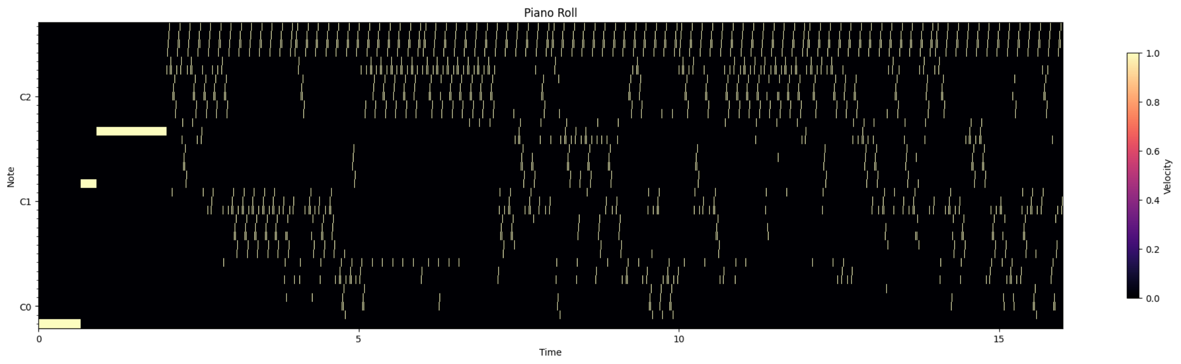


(b)

Figure 3: Plots of the loss function for both Generator and Discriminator. (a) Model 0. (b) Model 1.



(a)



(b)

Figure 4: Example of 8 bar generated. (a) Output from Model 0. (b) Output from Model 1.