Recurrent GAN Models for Music Generation

Azaan Rehman[*], **Jeffrey Tsaw**[†]
Carnegie Mellon University
`arehman@andrew.cmu.edu, jtsaw@andrew.cmu.edu`

Abstract

This project focuses on using generative models to simulate human-like abilities in music composition from continuous data. We use a Generative Adversarial Network framework and attempt to generate music without conditioning the network to musical theory of any kind. Using an RNN GAN as a baseline, we extend this architecture and introduce two new architectures that make use of convolution and attention layers, which facilitate more context aware generation, and also a novel Peripheral Frequency classifier, which allows the discriminator to learn to distinguish generated and real music based on sound. We found these architectures produced high-quality music, and produced music that statistically resembled real music more than the baseline did.

1 Introduction

Music composition is an incredibly complex task that has yet to be readily solved by existing methods. Human compositions typically have complex notions of sounds, tones, and rhythms embedded in the music, and also lack universal structure and form. For this reason, it has historically been difficult to create algorithms that can consistently generate aurally pleasing music. Our goal was to attempt to generate unique, pleasant sounding music that resemble classical compositions from scratch, without enforcing the model to adhere to musical theory restrictions.

This work focuses on generating music in a sequential manner from continuous data. This involves generating musical events (note, rest, chords) at every time step for a specified song length.

In order to solve this problem, we focus our efforts on a class of neural network architectures known as Generative Adversarial Networks (GANs)[3]. This approach involves training two networks, a generator and discriminator, with conflicting objectives. The generator attempts to generate samples from random noise to fool the discriminator, while the discriminator attempts to differentiate between generated and real data samples. As a baseline, we use C-RNN GAN[7], a GAN that leverages recurrent neural networks (RNN), which generate sequences better than regular deep networks, to generate music. We introduce CARNN mGAN, a convolutional and attention based extension of the baseline with an input mapping network. Finally, we attempt to generate better sounding music by extending CARNN mGAN with a peripheral frequency classifier, which we call CARNN PF-mGAN.

2 Background and Related Works

2.1 Data Representation

There are two main representations of music files: audio waveforms, and MIDI files[1]. Audio waveforms represent music as signal amplitudes at sampled time steps. They are the most direct

[*]Computer Science Department
[†]Department of Electrical and Computer Engineering

representation of music and offer the best resolution. MIDI files represent music as a sequence of *MIDI events* that are each a 4 number tuples:

$$(a, b, c, d) \tag{1}$$

where:

$a =$ *Note event*: On for play note, off for stop playing note
$b =$ *Channel number*: Integer from 0-15 indicating the instrument or track
$c =$ *MIDI note number*: Integer from 0-127 specifying pitch of note to play, or stop playing
$d =$ *Velocity*: Integer from 0-127 representing how loud to play the note or reduce volume to.

MIDI files also encode relative time scales, making generation for a fixed time length fairly simple.

This work focuses on MIDI representations for inputs as they offer several advantages over raw audio. MIDI encodes music in a more compact manner, helping restrict the output space of the generator to real piano notes. MIDI files also improve discriminator accuracy, as decisions are based on notes, instead of frequency/amplitude comparisons. All models were trained using MIDI representations of classical music, and the generator is trained to generate music in a MIDI format.

2.2 GANs for Generation Tasks

GANs were first proposed in 2014, and have been proven to generate highly realistic data samples, particularly in image generation tasks[3][6]. This framework has the added advantage of being compatible with various types of neural architectures[2]. Previous work has implemented GANs with fully-connected neural networks, convolutional neural networks, and, most relevant for this work, recurrent neural networks[2][14][2]. Mapping functions have been implemented to facilitate generation or discrimination[6][15]. These ideas have yet to be applied to the music generation problem, and thus inspired our CARNN mGAN architecture. Auxillary Classifier GANS [10] use an external classifier to generate realistic samples for multiple classes. We take inspiration from this design when developing the peripheral frequency classifier.

2.3 Deep Learning Methods for Music Generation

RNNs have been demonstrated to be successful at modelling sequences. In language models, they can successfully estimate the conditional distribution for the next token in a sequence, which can be used to generate realistic sequences[5][9][4]. More recently, Long Short Term Memory (LSTM) cells have replaced traditional RNN architectures, as they avoid the ’vanishing gradient’ phenomena that arises from repeated computations in the recurrent layer. Our work builds on this idea and applies LSTMs to a GAN framework. On top of LSTMs, the attention mechanism is also a key component of generating compelling sequences in language models, capturing dependencies independent of their position in the sequence [13]. We apply attention to a recurrent LSTM network, which, to our knowledge, has yet to be done in a music generation problems.

Other deep learning methods attempt to use raw audio to generate music [11][5]. [5],[9] are able to generate music in the frequency domain rather than the time domain. This approach is faster than [11], as it was able to train on groups of frequency samples instead of a single time sample per song. These works show deep networks are capable of learning musical features from raw audio. We leverage this concept when designing the peripheral frequency classifier. Frequency domain inputs have also been used for genre classification tasks to a fair amount of success [8]. Our models extend this application to a generative approach.

Convolutional Neural Networks (CNN) have also used in music generation tasks to generate realistic sounding music [11][5][14]. These architectures are easier to train, and can also capture dependencies in inputs. Our work combines these ideas into a model with convolutional and recurrent components. [14] also uses a conditioning mechanism to ’prime’ the network with an initial melody or chord sequence. Our work avoided conditioning, and focused on generating music using only random noise.

3 Methods

As mentioned in 1, we used C-RNN GAN as a baseline, and offer several extensions to the architecture.

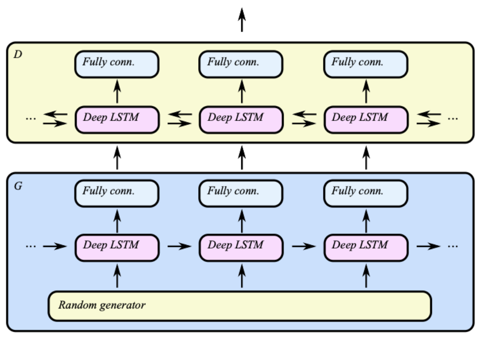


Figure 1: baseline architecture: Generator G produces MIDI events, discriminator D is trained to distinguish between generated and real samples

3.1 C-RNN GAN: Continuous Recurrent Neural Network GAN

Our baseline is based on a variant of C-RNN GAN[7]. The details of this architecture are as follows.

3.1.1 Generator Architecture

As seen in fig. 1, The generator G is a deep neural network whose first layer is a unidirectional deep LSTM layer, and second is a feed-forward layer. The deep LSTM layer consists of 2 stacked LSTM cells each with 100 units, while the feed-forward layer consists of a dense layer with ReLU activation and a final dense layer with linear activation.

During generation, G takes a random vector $v \sim [0, 1]^4$, which represent the 4 components of a MIDI event, concatenated with the generated output at the previous time step, or random values for generating the first time step. The output will be a vector of size 4 representing a MIDI event at that time step. Generation is repeated for the desired length of the song.

3.1.2 Discriminator Architecture

As seen in fig. 1, the discriminator D is a deep neural network whose first layer is a stacked BiLSTM layer, and second is a feed-forward layer. The stacked BiLSTM layer consists of 2 stacked Bidirectional arrays of LSTM cells, where the length of the array is the song length, and each LSTM cell contains 100 units. The feed-forward layer consists of a dense layer with sigmoid activation. The feed-forward layer shares parameters across time steps.

During discrimination, D takes in a sequence of MIDI events, and the sigmoid values are averaged across all time steps to obtain the final decision.

3.1.3 Training

Training is split into 2 phases: pretraining and training.

During pretraining, G is trained with the data set for $p = 6$ epochs, where p is a hyperparameter. A variant of teacher forcing is implemented, where at each time step instead of concatenating the random vector with the previous generated output, the random vector is concatenated with the MIDI event at the previous time step from the real data sample. We pre-train G to generate samples similar to the data set, and use MSE loss between generated MIDI events and the corresponding events from real data, averaged across T time steps for N data samples.

$$L_{G,\text{pre}} = \frac{1}{N} \sum_{i=1}^{N} \left\| \left(\boldsymbol{x}^{(j)} - G(\boldsymbol{z}^{(j)}) \right) \right\|_2^2 \tag{2}$$

where

$\boldsymbol{x}^{(j)}$ = are the MIDI events from the jth sample of size $T \times 4$
$\boldsymbol{z}^{(j)}$ = is the random vector v input to the generator as detailed in 3.3

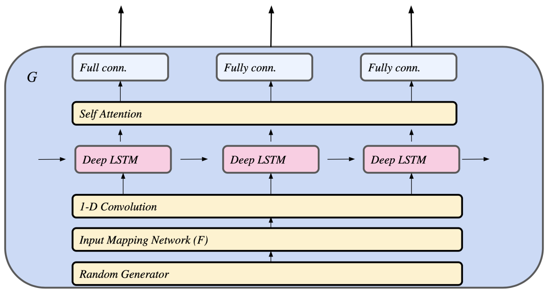


Figure 2: Generator architecture: G now as additional convolution and attention layers

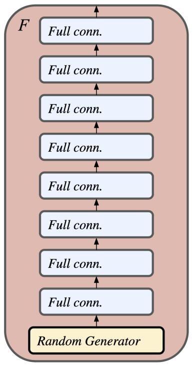


Figure 3: Input Mapping Network: maps random vector $\mathcal{Z}$ to latent variable W

We use the pretraining phase as a variant of curriculum training where the generator is trained with increasingly long songs every epoch. Parameter updates were done using Back Propagation Through Time (BPTT) using SGD on minibatches of 20, using a learning rate of 0.1.

During training, the architecture is trained to minimize the following loss functions. These equations are based off G attempting to maximize the mistakes D makes, while D attempting to maximize its ability to distinguish generated and real samples.

$$L_G = \frac{1}{N}\sum_{i=1}^{N} \log(1 - D(G(\boldsymbol{z}^{(i)}))) \tag{3}$$

$$L_D = \frac{1}{N}\sum_{i=1}^{N} -\log(D(\boldsymbol{x}^{(i)}) - \log(1 - D(G(\boldsymbol{z}^{(i)}))) \tag{4}$$

Parameter updates were done using BPTT for both G and D, using SGD with L_2 regularization on minibatches of 20 with a learning rate of 0.1 that is annealed after 25 epochs.

3.2 CARNN mGAN: Convolutional Attention RNN GAN with Input Mapping Network

We introduce a novel architecture through 2 primary additions to the baseline architecture, as shown in fig. 2.

1. Convolutional and Attention Layers to increase range of context considered for generation
2. Input Mapping Network to map random inputs to intermediate latent space

Convolutional and Attention Layers: This architecture uses convolutional layers to weigh future inputs to the generator when generating MIDI events for the current time step. The baseline only uses past information embedded in the LSTM and the previously generated output when generating MIDI events for a given time step. Our hope is the network will learn to weigh future and past inputs to generate more pleasant sounding music. An attention layer is added after the deep LSTM layer (see fig. 2) to further encourage future parts of the generated music to affect the current part, leading to a more coherent, fluid sound.

Input Mapping Network: Inspired by [6][15], we introduce a input mapping network in G to map the random input vector $\mathcal{Z}$ to an intermediate latent space, $\mathcal{W}$. This mapping network can be described as a function $F : \mathcal{Z} \to \mathcal{W}$. The goal of this network is to learn a transformation that will map any random input vector to a latent space that the generator can use to generate pleasant sounding music.

3.2.1 Generator Architecture

The input mapping network is implemented as 8 stacked dense layers and ReLU activation, with output dimension identical to input dimension as shown in fig. 3. For songs of length n, the input to

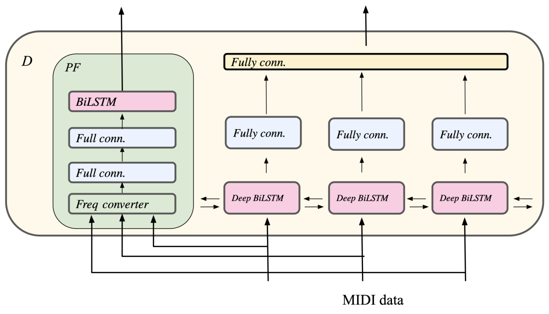


Figure 4: Discriminator with PF classifier: The PF classifier is added in parallel to the original discriminator and generates a decision based on frequency spectrum of input

the network is the entire random input vector $\boldsymbol{V} : (n \times 4)$, where each row $\boldsymbol{v_i} \sim [0, 1]^4$ represents the 4 MIDI event components (see 3.3). This input is flattened before passing through the input mapping network. The output of this network is then passed through a convolutional layer before passing into the deep LSTM layer. This convolutional layer uses a 5×5 window, and pads the input such that the output has the same dimensions as the input. A self attention layer is applied to the output of the deep LSTM layer that shares parameters across time and is used to propagate melodic qualities through future time steps. Generation is identical to that described in 3.3.

3.2.2 Discriminator Architecture

D is identical to that of the baseline, see 3.1.2.

3.2.3 Training

Pretraining strategy for G is identical to that of the baseline 3.1.3.

During training, we use a *feature matching* loss function for G (5) to replace the loss function used in the baseline (3)[12]. With feature matching loss, the objective is to produce output with internal representation, $f(G(\boldsymbol{z}))$, close to that of real data, $f(\boldsymbol{x})$, where the function $f(\cdot)$ is the internal representation. For this architecture, $f(\cdot)$ is chosen to be the output of the feed-forward layer of the generator. The feature matching loss is hence the MSE between the output of the generator and real music, averaged across all time steps.

$$\hat{L_G} = \frac{1}{N} \sum_{i=1}^{N} \left\| \left(f(\boldsymbol{x}^{(\boldsymbol{i})}) - f(G(\boldsymbol{z}^{(\boldsymbol{i})})) \right) \right\|_2^2 \tag{5}$$

With this architecture, we also implement *parameter freezing*, where the parameters of the generator are frozen when the discriminator becomes too strong, and vice versa [12]. This is decided when the discriminator loss is less than 70% of the generator loss, and vice versa.

This architecture is trained using the same hyperparameters as the baseline, Parameter updates are applied using BPTT, SGD with L_2 regularization, minibatches of 20 and a learning rate of 0.1 that is annealed after 25 epochs. We train this architecture with and without the input mapping network to evaluate its effect.

3.3 CARNN PF-mGAN: CARNN mGAN with Peripheral Frequency Classifier

Finally, we design a model that attempts to account for way the music *sounds* during generation. Our goal was to teach the discriminator to account for the audio waveform of both generated and real music, teaching the generator to generate music that sounds better, and more similar to that of real music. To accomplish this, we introduce a **Peripheral Frequency classifier (PF classifier)** to the discriminator. Inspired by [10], the discriminator now outputs 2 values: decision using the previous architecture described in 3.1.2, and a decision based on the frequency profile of the input. The goal of the discriminator is now to differentiate real and generated samples not only from the generated MIDI events, but also from their raw audio frequency waveforms.

3.3.1 Generator Architecture

G is identical to that in 3.2.1

3.3.2 Discriminator Architecture

D is extended to include a separate pipeline that acts as the PF classifier as in fig. 4. The original discriminator is left unchanged (3.1.2). The PF classifier is comprised of a **frequency converter** and a **deep neural network**.

- **Frequency Converter:**The frequency converter first transforms the generated sample to the frequency domain by converting the generated sample to an audio waveform, passing the waveform through a Discrete Fourier Transform, centering the resulting frequency signal, and applying an ideal low-pass filter to extract the central frequencies that range from -250rad/s to 250rad/s. The resulting signal is of size 5000.
- **Deep Neural Network:**The deep neural network first reduces the dimensionality of the input by passing it through 2 dense layers with ReLU activation and output dimensions 1000 and 500 respectively. The output is then passed through 2 stacked BiLSTM layers with each LSTM cell containing 100 units. This architecture is based on [5], and is successful at extracting musical features from frequency signals. Finally, the final output of the BiLSTM layer is passed through a dense layer with sigmoid activation to obtain a prediction on whether the frequency signal is from generated or real data.

3.3.3 Training

Pretraining is done for G , using the same strategy as the baseline in 3.1.3.

During training, the baseline loss functions are extended to account for the PF classifier, denoted by F.

$$L_{G,F} = \frac{1}{N}\sum_{i=1}^{N}\left[\log(1 - D(G(\boldsymbol{z}^{(i)}))) + \lambda \log(1 - F(G(\boldsymbol{z}^{(i)})))\right] \tag{6}$$

$$L_{D,F} = \tfrac{1}{N}\sum_{i=1}^{N}\left[-\log(D(\boldsymbol{x}^{(i)}) - \log(1 - D(G(\boldsymbol{z}^{(i)}))) - \lambda\log(1 - F(G(\boldsymbol{z}^{(i)}))) - \lambda\log(F(\boldsymbol{x}^{(i)}))\right] \tag{7}$$

where:

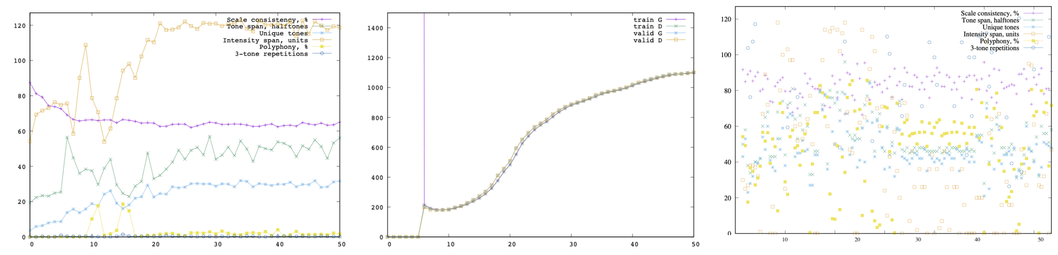
$\lambda =$ a hyperparameter specifying how heavy to weight PF classifier in loss.

The objective for G is now to minimize $L_{G,F}$ by generating samples that fool the discriminator and the PF classifier. The objective for the D is now to correctly differentiate between real and generated samples based on both MIDI events and frequency spectrum.

G is trained as described before in 3.2.1, while the original discriminator is trained identically to that of the baseline, see 3.1.3. The PF classifier is trained similarly, using BPTT, SGD with L_2 regularization and minibatches of 20 with a learning rate of 0.1 that is annealed after 25 epochs.

4 Results

We use a data set containing 3697 MIDI files from 160 different composers of classical music[7]. The data set is first normalized to a tick resolution of 384 per quarter note before being fed to the models. We use this data set to create a training and validation set, and evaluate each architecture's performance on the validation set every epoch. At the end of each epoch, G is sampled and generates 20 songs, and the *scale consistency* (purple line), *tone span* (green line), *unique tones* (light blue line), *number of units* (orange line), *polyphony percentage* (yellow line), and *3 tone repetitions* (dark blue line) are averaged and plotted. We omitted an aural evaluation as they are highly subjective, and our results would likely be influenced by convenience and sampling biases.



(a) Average MIDI statistics from generated music for baseline per epoch (b) Mean square error per epoch (c) MIDI statistics of real music

Figure 5

4.1 Baseline Results

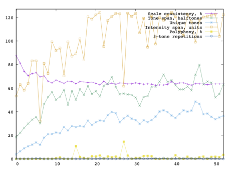
The baseline produced fairly good results as shown above. The baseline was able to generate songs that were fairly complex, as seen by the high average number of units in fig. 5(a). On the other hand, the music seemed fairly bland and simple, with a low number of unique tones, consistently long tone spans, and a relatively low degree of polyphony. In general, across each epoch, there is less variability in the music generated when compared with real music, as shown in fig. 5(c). The loss for the baseline increased for the first 40 epochs and started to plateau at around 1100 for both training and validation sets afterwards. This indicates that generator and discriminator failed to greatly improve with time.

4.2 CARNN mGAN Results

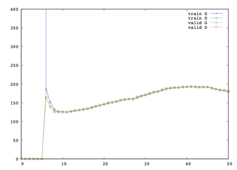
We evaluate this model with and without the input mapping network (as described in 3.2), to determine its impact on music generation. The results can be seen in fig. 6.

We denote CARNN GAN, as CARNN mGAN without the input mapping network. CARNN GAN yielded slightly better results than the baseline. There is more variability in the number of units and tone spans, and a higher number of unique tones than the baseline. This greater variance more closely resembles real music fig. 5(c). This model, however, unlike real music, wasn't able to generate music with a non-trivial amount of polyphony and three-tone repetitions. The loss for the generator and discriminator was lower than that of the baseline, reaching a maximum of only 400 around epoch 40. This indicates that both the generator and discriminator performed better than the baseline.

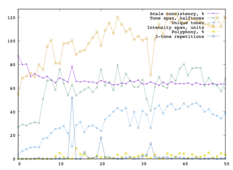
CARNN mGAN (with the input mapping network) generated similar results than that of CARNN GAN. However, there is an even greater degree of variation in number of units, number of unique tones, tone span that more closely resembles real music than CARNN GAN. Most promisingly, there were higher amounts of polyphony than both CARNN GAN and the baseline, indicating more complex and potentially better sounding music. Similar to without the input mapping network, scale consistency was unchanged from the baseline and still resembles real music. The loss in general was higher than than CARNN GAN, however, the loss steadily decreased from epoch 20 onwards indicating continued improvement. We attribute the high loss to the fact that the generator now needs to learn an input mapping function (see 3.2.1) as well as the parameters for generation leading to a higher initial loss. Overall, CARNN mGAN outperformed both the baseline and CARNN GAN when it comes to generating realistic sounding music.



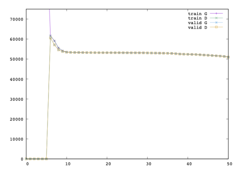
(a) Average MIDI stats for CARNN GAN (without input mapping network) per epoch



(b) Loss per epoch for CARNN GAN



(c) Average MIDI stats for CARNN mGAN (with input mapping network



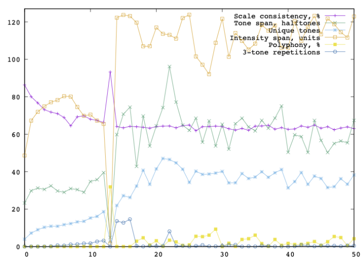
(d) Loss per epoch for CARNN mGAN

Figure 6

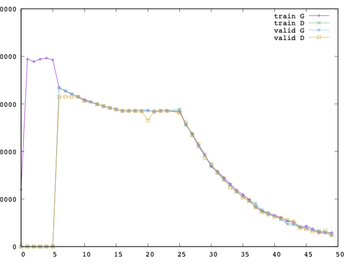
4.3 CARNN PF-mGAN Results

The original implementation of the PF classifier used 5 dense layers with ReLU activation and set $\lambda = 1$ (see 3.3.3). This network failed to improve and the generator was not able to generate convincing music.

Switching the PF classifier architecture to that described in 3.3.2 and using $\lambda = 0.3$, the results were much better, as seen in fig. 7. There is similar variability in tone span, units and unique tones, while also having a greater degree of polyphony compared with CARNN m-GAN. This indicates the music generated was better than the baseline and similar to CARNN m-GAN, but a greater variance and higher polyphony more closely matches real music. The loss per epoch was extremely high, but began to decay quickly after epoch 25, indicating that the generator and discriminator started to improve greatly. Therefore, based off our evaluation metrics, this architecture performed slightly better than the CARNN mGAN presented previously.



(a) Average MIDI stats for CARNN PF-mGAN per epoch



(b) Loss for CARNN PF-mGAN per epoch

Figure 7

5 Discussion and Analysis

Overall, both CARNN mGAN and CARNN PF-mGAN improved the baseline results according to our evaluation metrics. However, the authors want to clarify that this does not correlate with pleasant sounding music. In fact, to the authors and their colleagues, music generated by CARNN mGAN and CARNN PF-mGAN are only slightly better than the baseline. If anything, our experiments tell us that our metrics are insufficient at evaluating music quality.

The primary limitation of our work is a lack of objective evaluation metrics that are correlated with pleasant sounding music. Moreover, the results presented were also limited by available hardware. As every model was quite large, only 50 epochs of training could be completed in a reasonable amount of time. Despite these limitations, we believe we accomplished our goal of generating more pleasant music than the baseline.

One potential extension would be to allow the PF classifier to consider the entire frequency spectrum. To do so an autoencoder or other dimensionality reduction methods could be used to reduce the frequency spectrum to a manageable size. It would also be interesting to vary architecture and loss function of the PF classifier.

References

[1] Jean-Pierre Briot, Gaëtan Hadjeres and François-David Pachet. *Deep Learning Techniques for Music Generation – A Survey*, 2017; arXiv:1709.01620.

[2] Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta and Anil A Bharath. *Generative Adversarial Networks: An Overview*, 2017; arXiv:1710.07035. DOI: 10.1109/MSP.2017.2765202.

[3] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville and Yoshua Bengio. *Generative Adversarial Networks*, 2014; arXiv:1406.2661.

[4] Allen Huang and Raymond Wu. *Deep Learning for Music*, 2016; arXiv:1606.04930.

[5] Vasanth Kalingeri and Srikanth Grandhe. *Music Generation with Deep Learning*, 2016; arXiv:1612.04928.

[6] Tero Karras, Samuli Laine and Timo Aila. *A Style-Based Generator Architecture for Generative Adversarial Networks*, 2018; arXiv:1812.04948.

[7] Olof Mogren. *C-RNN-GAN: Continuous recurrent neural networks with adversarial training*, 2016; arXiv:1611.09904.

[8] Zain Nasrullah and Yue Zhao. *Music Artist Classification with Convolutional Recurrent Neural Networks*, 2019; arXiv:1901.04555.

[9] Aran Nayebi and Matt Vitelli. *GRUV : Algorithmic Music Generation using Recurrent Neural Networks*, M. 2015.

[10] Augustus Odena, Christopher Olah and Jonathon Shlens. *Conditional Image Synthesis With Auxiliary Classifier GANs*, 2016; arXiv:1610.09585.

[11] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior and Koray Kavukcuoglu. *WaveNet: A Generative Model for Raw Audio*, 2016; arXiv:1609.03499.

[12] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford and Xi Chen. *Improved Techniques for Training GANs*, 2016; arXiv:1606.03498.

[13] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser and Illia Polosukhin. *Attention Is All You Need*, 2017; arXiv:1706.03762.

[14] Li-Chia Yang, Szu-Yu Chou and Yi-Hsuan Yang. *MidiNet: A Convolutional Generative Adversarial Network for Symbolic-domain Music Generation*, 2017; arXiv:1703.10847.

[15] Jun-Yan Zhu, Taesung Park, Phillip Isola and Alexei A. Efros. *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*, 2017; arXiv:1703.10593.