UNIVERSITY OF PIRAEUS & NCSR "DEMOKRITOS"
MSC PROGRAMME IN ARTIFICIAL INTELLIGENCE

Deep Generation of electronic music

by

Dimitra Manoura

Submitted
in partial fulfilment of the requirements for the degree of
Master of Artificial Intelligence
at the
UNIVERSITY OF PIRAEUS

Supervisor: Theodoros Giannakopoulos
Researcher B

Athens

Deep Generation of electronic music
Dimitra Manoura
MSc. Thesis, MSc. Programme in Artificial Intelligence
University of Piraeus & NCSR "Demokritos"

UNIVERSITY OF PIRAEUS & NCSR "DEMOKRITOS"
MSC PROGRAMME IN ARTIFICIAL INTELLIGENCE

Deep Generation of electronic music

by

Dimitra Manoura

Submitted
in partial fulfilment of the requirements for the degree of
Master of Artificial Intelligence
at the
UNIVERSITY OF PIRAEUS

Supervisor: Theodoros Giannakopoulos
Researcher B

(Signature)	(Signature)	(Signature)
.....	.....	.....
Theodoros Giannakopoulos	George Vouros	Ilias Maglogiannis
Researcher B	Professor	Professor

Athens

UNIVERSITY OF PIRAEUS & NCSR "DEMOKRITOS"
MSC PROGRAMME IN ARTIFICIAL INTELLIGENCE

Declaration of Authorship

(1) I declare that this thesis has been composed solely by myself and that it has not been submitted, in whole or in part, in any previous application for a degree. Except where states otherwise by reference or acknowledgment, the work presented is entirely my own.

(2) I confirm that this thesis presented for the degree of Master of Science in Informatics and Telecommunications, has

(i) been composed entirely by myself

(ii) been solely the result of my own work

(iii) not been submitted for any other degree or professional qualification

(3) I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or processional qualification except as specified.

(Signature)

.....

Dimitra Manoura

Athens

Acknowledgments

Αρχικά, θα ήθελα να ευχαριστήσω ιδιαίτερα τον επιβλέποντά μου, Θοδωρή Γιαννακόπουλο, τόσο για την πολύτιμη καθοδήγησή του καθ΄ όλη τη διάρκεια της διπλωματικής εργασίας μου, όσο και για το πολύ υψηλό επίπεδο των διαλέξεών του, τα οποία με ενέπνευσαν για την κατεύθυνση που θα ακολουθήσω.

Ευχαριστώ τους συμφοιτητές και τις συμφοιτήτριές μου για τη συνεργασία και τη συμπαράσταση σε όλη τη διάρκεια του μεταπτυχιακού μας, για την αλληλεγγύη και τη συλλογική προσέγγιση σε όλα, κάνοντας αυτή την εμπειρία αξέχαστη.

Ευχαριστώ την οικογένειά μου, τη μαμά μου και τον αδερφό μου για την αμέριστη στήριξή τους, και τον μπαμπά μου, στον οποίο αφιερώνω αυτήν τη διπλωματική εργασία, για την συνεχή πίστη του σε μένα στα πάντα και το ενδιαφέρον του για ό,τι κάνω.

Ευχαριστώ τους φίλους και τις φίλες μου, χωρίς την βοήθεια των οποίων δεν θα είχα ολοκληρώσει την διπλωματική μου εργασία, καθώς ήταν εκεί για μένα σε όλες τις στιγμές που τους/τις χρειαζόμουν. Τέλος, ευχαριστώ ιδιαίτερα τον Μιχάλη για την ικανή στήριξή του και την αναγκαία παρουσία του στα εύκολα και στα δύσκολα όλον αυτόν τον καιρό, καθώς και για ό,τι έχει κάνει για μένα.

Abstract

The goal of this thesis is to explore the generation of electronic music through the utilization of Deep Learning techniques.

The challenge of algorithmically generating music lies in creating authentic and aesthetically pleasing compositions that resonate with listeners. Music is deeply human, rooted in emotion and culture, while algorithms lack this understanding. This project aims to bridge technology and art by providing musicians with tools to explore new expressions and spark creativity. By finding a balance between technology and human emotion, it seeks to enrich musical innovation and inspire new compositions.

To do that, the path we have chosen is by making our own dataset which consists of images that represent the original music. These images are called spectrograms and they are a 2D representation of a sound, essentially a graph, where the horizontal axis represents time and the vertical axis represents frequency. To address this challenge, Generative Adversarial Networks (GANs) are employed as the modeling approach. GANs are a class of deep learning algorithms that have shown promise in generating realistic images and, by extension, spectrograms. The methodology involves training DCGANs on the dataset of spectrograms to learn the underlying patterns and structures of electronic music.

Generative Adversarial Networks (GANs) consist of two networks, the Generator and the Discriminator, engaged in a competitive, zero-sum game. Known as minimax in game theory, each network aims to outperform the other: the Generator generates "fake" images, while the Discriminator discerns between "real" and "fake" ones. As the Generator seeks to deceive the Discriminator and the Discriminator tries not to be fooled, both networks improve, resulting in the generation of increasingly realistic images.

The methods under examination revolve around the training and optimization of Deep Convolutional Generative Adversarial Networks (DCGANs), which are essentially GANs whose both the Generator and the Discriminator have Convolutional layers in their architectures.

These spectrograms are then converted back into audio, which was the goal all along. To evaluate how realistic the generated sound is, a questionnaire was distributed for people to answer, where participants are asked to find which one of the songs they heard is AI-generated.

In conclusion, this thesis showcases the potential of DCGANs in the domain of electronic music generation. Despite the inherent limitations that GANs can have, our experimental results demonstrate that our models successfully generated music closely resembling the compositions in our dataset, achieving a realistic sound output.

Contents

List of Tables

List of Figures

List of Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
BCE	Binary Cross Entropy
CNN	Convolutional Neural Network Deep Convolutional Gen-
DCGAN	erative Adversarial Network
DFT	Discrete Fourier Transform
DJ	Disc Jockey
DL	Deep Learning
FFT	Fast Fourier Transform
GAN	Generative Adversarial Network
GPU	Graphics Processing Unit
LSTM	Long Short-Term Memory
MIDI	Musical Instrument Digital Interface
ML	Machine Learning
NN	Neural Network
ReLU	Rectified Linear Unit
RGB	Red Green Blue
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
TPU	Tensor Processing Unit

Chapter 1

Introduction

From the very beginning of the world, it's undeniable that the Big Bang generated sound. Since then, sound has not only been intertwined with every natural function but has also become inherent to each function. It transformed into music when humans, equipped with structurally and functionally essential sound properties (like speech), chose to arrange sound in ways and systems that defined it as "music". Throughout the historical evolution as we understand it, the artistic progression or evolution of music has consistently interwoven with parallel technological and scientific advancements, often serving as a catalyst for them.

1.1 Problem description

The challenge of creating music through algorithmic generation is a complicated task that extends beyond the technical side of writing code. The challenge lies in not only achieving the generation of music through algorithms but, more crucially in making sure that the generated music also resonates as authentic and aesthetically pleasing to listeners.

This challenge is really about the convergence between technology and art. Music at its core is a deeply human and emotional form of expression, that carries its own history and culture. Algorithmic generation, on the other hand, is incapable of understanding the complex aspects of musical expression.

This project is an effort to find a balance between these two, and provide a useful tool to musicians that can help them express themselves and unravel new aspects of their expression. Sometimes a musician needs just a spark of inspiration, or a small stepping stone in order to create something new.

1.2 Related Work

AI generated music is a field that has gained popularity in recent years. While the concept of using computers to generate music has been around for decades, advancements in machine learning have finally brought us to a point where AI can produce music indistinguishable from human compositions, blurring the lines between artificial and artistic creation.

There are many methods one can follow in order to generate music, each with its strengths and limitations:

Traditional methods often rely on MIDI (Musical Instrument Digital Interface) datasets, where music is represented symbolically in terms of notes, chords and durations. This form of representation captures the core musical structure, making it an attractive choice for training machine learning models. MIDI data simplifies the learning process by abstracting away complexities such as timbre and audio effects, allowing models to focus solely on pitch, rhythm, and harmony. For example, models like those based on Autoencoders [2] are frequently trained on MIDI datasets to learn compressed representations of musical patterns. The encoder compresses musical input into a latent space, while the decoder reconstructs the music. These models are great at generating music that resembles the original dataset but they often don't perform that well when generating music beyond their learned latent space, as they struggle with creativity.

Recurrent Neural Networks (RNNs) RNNs [3] are another popular technique due to their ability to process sequential data, making them well-suited for tasks like music generation. RNNs are designed to capture dependencies between musical notes, allowing them to generate coherent sequences based on previous outputs. However, RNNs face limitations when it comes to long-term coherence. As the length of a sequence grows, RNNs often suffer from the vanishing gradient problem, causing them to lose track of earlier context and resulting in repetitive or disconnected outputs, especially in longer compositions.

LSTMs [4] were introduced to address the shortcomings of standard RNNs. With their memory cells, LSTMs are better equipped to capture long-term dependencies, which is essential for generating longer and more structured pieces of music. They are commonly used for tasks such as predicting the next note or chord in a sequence or composing music in the style of a specific genre or composer. Despite their advantages over RNNs, LSTMs are still constrained by their reliance on symbolic data, often producing music that, while coherent, can feel formulaic or overly predictable over time.

More advanced models like Jukebox [5] take a different approach, by working directly with raw audio rather than symbolic representations. Using a combination of convolutional neural networks (CNNs) and autoregressive models, Jukebox can generate high-fidelity music from user-specified prompts such as genre, artist style, or even specific instruments. This model can generate music that is more expressive and detailed than what is possible with MIDI data, but at the cost of significantly higher computational demands. Additionally, while the quality of the music is high, Jukebox can still struggle with maintaining innovation and structural coherence, particularly over extended compositions.

MusicLM [6] is a state-of-the-art model that represents a leap forward in text-to-music generation. Instead of requiring musical notes or predefined structures, MusicLM allows users to describe the type of music they want, such as "a relaxing jazz piece with a mellow piano," and the system generates matching audio. MusicLM relies on transformer architectures and language modeling techniques adapted to music data. Like Jukebox, however, MusicLM demands large datasets and significant computational resources. However, MusicLM is still evolving when it comes to precisely controlling detailed aspects of the music it generates.

Combining symbolic (notes, chords) and audio (waveforms) representations is another area of research where both aspects of music lead to AI-generated music [7]. Those models first generate symbolic music, like a sequence of MIDI notes, and then convert these symbolic representations into audio using a separate neural network. However, transforming symbolic music into convincing audio is not without challenges. The process often introduces imperfections, leading to outputs that may sound robotic or lack the expressive qualities of naturally composed music.

Beyond the core methods, the boundaries of AI music are pushed through various ways. Generative pre-training models like GPT, while primarily focused on text, show promise in generating musical elements like lyrics and song structures [8]. Audio style transfer utilizes deep learning models, like Convolutional Neural Networks (CNNs), to transfer the stylistic features between audio samples. For example, transforming a pop song into a classical orchestral arrangement [9]. Also, evolutionary AI music generation utilizes genetic algorithms to "breed" neural networks, creating music that evolves towards specific characteristics over generations [10].

In the context of music generation, GANs offer a novel approach. Unlike the sequential methods described earlier, GANs such as SpecGAN [11] focus on generating spectrograms, like us, which are visual representations of audio in terms of time, frequency, and amplitude. Once the spectrogram is generated, it can be

converted back into an audio waveform using techniques like inverse Short-Time Fourier Transform (STFT), as we will talk about later. This approach allows GANs to capture detailed timbral information and audio nuances, making them capable of generating realistic-sounding music. However, GANs face challenges such as maintaining coherent long-term sequences, as their training is not inherently designed for modeling temporal dependencies. Additionally, GANs often suffer from mode collapse, where the generator produces limited variations, making the generated music sound repetitive or lacking diversity.

Despite these challenges, GANs have distinct strengths. Unlike sequential models such as RNNs and LSTMs, GANs can generate entire spectrograms or waveforms in one go, potentially resulting in more fluid and coherent short segments of music, especially for shorter segments of music. However, the lack of explicit temporal modeling in GANs can make longer compositions disjointed or lose their musical flow over time.

WaveGAN [11], for instance, generates raw waveforms directly instead of working with spectrograms. By operating in the time domain, it can create realistic audio clips, particularly suited for generating sound effects or short pieces of music. While effective for short audio generation, WaveGAN struggles to scale this approach for longer compositions. This limitation highlights the need for ongoing research in GAN-based music models, particularly in addressing long-range temporal coherence.

1.3 Motivation

As an amateur singer and a potential AI researcher, I was fascinated to find out how the world of music and the world of Artificial Intelligence, two seemingly very different worlds, can not only coexist but but also learn from each other. Through this study I am taking my first step into finding a convergence between the unpredictability of human creativity and the precision of artificial intelligence.

1.4 Thesis outline

The remainder of this thesis is organized as follows. In Chapter 2 we talk about Artificial Intelligence, Machine Learning, Deep Learning, Convolutional Neural Networks, Generative Adversarial Networks and other notions that we will be using in the next chapters. In Chapter 3 we introduce the concept of spectrograms, how they are made and their necessity, how we chose our songs to convert to spectrograms and made our dataset and why we chose to make it the way we did. In Chapter 4 we present the model architectures we used for our experiments, we describe the

training process and talk about the hyperparameters we chose, we present our results and we comment on them based on metrics and based on people's opinions. Finally, in Chapter 5 we discuss about the results and the ways we, as well as others can expand and improve the specific project.

Chapter 2

Background

Artificial Intelligence (AI) is a branch of computer science that deals with the development of intelligent machines that can perform tasks that typically require human intelligence, such as visual perception, speech recognition, decision-making, and language translation.

AI algorithms use a combination of machine learning, natural language processing, and deep learning techniques to analyze and understand complex data, learn from past experiences, and make predictions or decisions based on that learning.

AI has many applications, including virtual assistants, self-driving cars, facial recognition systems, fraud detection, healthcare diagnosis, and more. It is a rapidly evolving field that is poised to revolutionize the way we live and work.

2.1 Machine Learning

Machine learning is a subfield of artificial intelligence (AI) that involves the development of algorithms that enable computer systems to automatically learn from data and improve their performance on a specific task without being explicitly programmed.

Machine learning algorithms are trained on large datasets to identify patterns and relationships in the data. This training process involves feeding the algorithm with labeled or unlabeled data and allowing it to learn from the data to make predictions or decisions on new data.

There are three main types of machine learning: supervised learning, unsupervised learning, and reinforcement learning.

- Supervised learning: This type of machine learning involves providing the algorithm with labeled data and teaching it to make predictions or decisions

based on that labeled data. For example, a supervised learning algorithm can be trained to identify spam emails based on a set of labeled spam and non-spam emails.

- Unsupervised learning: This type of machine learning involves providing the algorithm with unlabeled data and allowing it to find patterns and relationships on its own. For example, an unsupervised learning algorithm can be used to identify clusters of similar customers in a dataset based on their purchasing behavior.

- Reinforcement learning: This type of machine learning involves teaching the algorithm through trial and error, where the algorithm learns to make decisions based on rewards and punishments. For example, a reinforcement learning algorithm can be trained to play a game by rewarding it for making successful moves and punishing it for making unsuccessful moves.

Machine learning has many practical applications, such as image recognition, natural language processing, autonomous vehicles, recommendation systems, fraud detection, and more.

In order to complete any of these tasks, one would follow the listed steps:

a. Data collection: This step involves gathering relevant data for the problem at hand. The data can be collected from various sources, such as databases, APIs, or web scraping.

b. Data preprocessing: Once the data is collected, it needs to be processed to ensure that it is in a format that can be used by the machine learning algorithm. This step can include cleaning the data, handling missing values, encoding categorical variables, and more.

c. Data splitting: After preprocessing, the data is split into two sets: a training set and a testing set. The training set is used to train the machine learning algorithm, while the testing set is used to evaluate its performance.

d. Model selection: The next step is to choose an appropriate machine learning algorithm that can solve the problem at hand. This decision can depend on various factors, such as the type of problem, the size of the dataset, and the available computational resources.

e. Model training: With the algorithm selected, the next step is to train the model on the training set. This involves feeding the training data into the model and adjusting its parameters to minimize the error or loss.

f. Model evaluation: After training, the model is evaluated on the testing set to assess its performance. This step can involve calculating various metrics, such as accuracy, precision, recall, and F1 score.

g. Model tuning: If the model performance is not satisfactory, it may be necessary to tune its hyperparameters, which are settings that control the behavior of the model. This step involves experimenting with different values of the hyperparameters and selecting the combination that yields the best performance.

h. Prediction: Finally, once the model has been trained and tuned, it can be deployed to make predictions on new, unseen data.

These steps are iterative and may require going back and forth between them until an acceptable level of performance is achieved.

2.2 Deep Learning

Deep learning is a sub-field of machine learning that is concerned with training Artificial Neural Networks, or just Neural Networks, with multiple layers to learn and make predictions on complex datasets. It is called "deep" because it involves training neural networks with many layers, which enables them to learn increasingly complex representations of the input data. Neural networks are called "neural" because they are modeled after the structure and function of the human brain's neural networks, which are made up of interconnected neurons. The goal of neural networks is to simulate the learning process of the brain, where each neuron receives input from other neurons, processes that input, and produces an output that is sent to other neurons.

Artificial neural networks (ANNs) are composed of interconnected nodes or "neurons," which are organized into layers. The input layer receives the raw input data, such as an image or a piece of text, and the output layer produces the final prediction or output. In between, there can be multiple hidden layers that perform various transformations of the input data, allowing the network to learn increasingly complex representations of the input.

The field of deep learning has rapidly grown in recent years due to several factors. One of the main factors is the availability of large amounts of data, which enables the training of deep neural networks with millions of parameters. This has been made possible by the increasing digitization of various industries, such as healthcare, finance, and transportation, which generate massive amounts of data that can be used to train deep learning models.

Another factor is the availability of powerful computational resources, such as graphical processing units (GPUs) and specialized hardware such as tensor processing units (TPUs), which enable the training and inference of deep learning models at a much faster rate than was previously possible.

In addition, the development of new deep learning architectures, such as convolutional neural networks (CNNs) for image and video data, recurrent neural networks (RNNs) for sequential data such as text and speech, and transformer models for natural language processing, has enabled the application of deep learning to a wide range of problems, including image recognition, speech recognition, natural language processing, and more.

The rapid growth of the field of deep learning has also been fueled by the availability of open-source software libraries, such as TensorFlow and PyTorch, which make it easier for researchers and practitioners to develop and deploy deep learning models. This has led to a proliferation of research and applications in the field, making deep learning one of the most exciting and rapidly evolving areas of artificial intelligence.

Deep learning, explores basic ideas that form the foundation of its complex systems. The perceptron, an essential component of neural networks, is where the adventure starts. The complex network of connections that characterizes these artificial brain networks is made possible by the perceptron, which serves as the fundamental unit.

The perceptron is composed of four essential components. Firstly, there are input values, which form the input layer of the perceptron. These values serve as the initial information fed into the system. Secondly, the perceptron involves weights and biases, influencing the importance of each input. These parameters are adjusted during the learning process to optimize the network's performance. The net sum is the third component, calculated by multiplying the inputs with their corresponding weights, adding them together and adding a bias. Finally, the activation function determines whether the perceptron should activate or not based on the net sum, introducing non-linearity to the system. Together, these four elements form the foundational structure of a perceptron in artificial neural networks. It's important to note that multiple perceptrons, each with its own set of weights, biases, and activation functions, are interconnected to create a neural network. In this collaborative arrangement, the output of one perceptron becomes the input for others.

The non-linearity provided by the activation functions is crucial for the successfulness of a neural network. Without non-linear activation functions, the network would essentially be reduced to a linear model, limiting its ability to understand

complex relationships amongst data, which is, in fact, the type of relationships encountered in real-world problems. Some of the most common activation functions that we will be using are listed below:

- $f(x) = \frac{1}{1+e^{-x}}$. Sigmoid activation function, characterized by its S-shaped curve, squashes the output between 0 and 1. Widely used in the output layer for binary classification tasks. But very small values become almost 0 and very large values become almost 1, so changes in weight mean no changes in the output which could lead to neuron saturation, where the neuron becomes less responsive and struggles to learn from different inputs.

- $f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$. Tanh activation function squashes the output between -1 and 1. The output is zero centered so when used as input for next layers, neuron saturation becomes less likely. It is often used instead of the sigmoid activation function in hidden nodes.

- $f(x) = \max(0, x)$. ReLu (Rectified Linear Unit) activation function, a widely used activation function that allows positive inputs to pass through unaltered, setting negative inputs to zero. Its simplicity and efficiency make it very popular in deep neural networks, promoting faster learning than the Sigmoid and Tanh functions.

- $f(x) = \max(cx, x), c < 1$. Leaky ReLu is a variant of the ReLU function. The key distinction of Leaky ReLU is that it allows a small, positive slope for negative inputs, allowing them to contribute to the learning process, unlike the traditional ReLU, which sets negative inputs to zero. Usually $c = 0.01$.

Before we continue, we have to additionally mention a few more concepts related to the functioning of neural networks. We can't possibly dive into everyone of them, but we are going to mention the ones we will be using later on.

The **loss function** is the way to measure how well our model is doing by comparing the predicted output and the actual target. The goal of the model is to minimize this function. There are a lot of loss functions as Binary Cross Entropy loss, Mean Average Error, L1 loss, L2 loss etc just to name a few.

Backpropagation utilizes the loss function to calculate **gradients**, determining how much each weight contributed to the prediction error. This backward propagation of errors guides the adjustment of weights during training, aiming to minimize the gap between predicted and actual outcomes. These gradients are then used by the **optimizer** to change the weight's values, resulting in better predictions over

time. The **learning rate**, a crucial parameter, determines the size of the weight adjustments made during optimization. A higher learning rate accelerates the convergence process but risks overshooting optimal weights, while a lower learning rate, despite slowing down convergence, enhances precision in reaching the optimal configuration. Another parameter we can modify is the **batch size**. The batch size is the number of training examples processed in one iteration. During each training iteration, the neural network processes a set of training examples, and then the model's weights are updated. When all the training data has been processed, that is called an **epoch**. A neural network is usually trained for multiple epochs.

2.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a type of deep learning model specifically designed for processing and analyzing visual data, such as images and videos. CNNs perform incredibly well in tasks such as image recognition, object detection, and image classification.

An image is composed of pixels, and each pixel is assigned a numerical value that represents its color or intensity. In a grayscale image, each pixel is typically represented by a single value ranging from 0 (black) to 255 (white). In a color image, each pixel is often represented by three values corresponding to the intensities of the red, green, and blue (RGB) color channels. So a grayscale image is actually a matrix and a color image actually consists of 3 matrices.

What differentiates CNNs from other neural networks is the use of convolutional layers, which employ small learnable filters, or kernels. The values within these filters are the weights that are updated during the training process. These filters slide across the input data, performing convolutions to capture specific features, like edges, textures, or more complex patterns. We can see a convolution example in Figure 2.1.

Let f be our input image, and h be our kernel. Then, m and n will be the dimensions of our output image, which is given by the following formula:

$$G[m,n] = (f * h)[m,n] = \sum_j \sum_k h[j,k] f[m-j, n-k]$$

A CNNs is actually a sequence of many convolutions, with each convolution's output acting as the input for the next one.

In parallel to the filters, there are more tools we can use when building a CNN. For example, the concept of **stride**. Just as the filters in convolutional layers slide

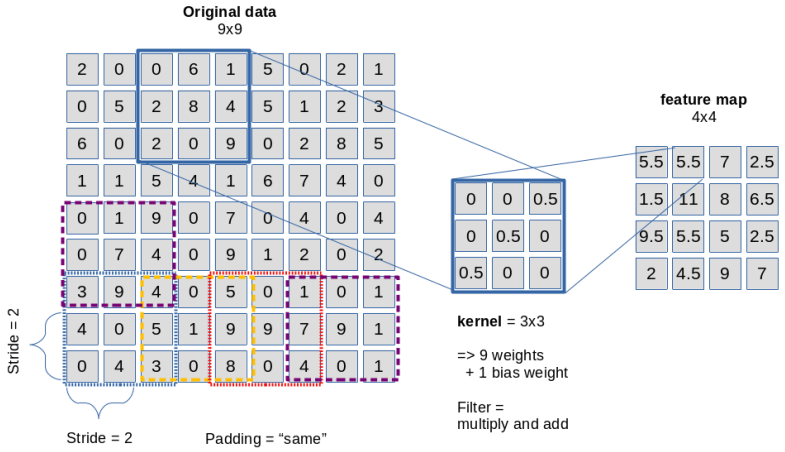

Figure 2.1: Convolution example, source.

across the input data, the stride determines how much these filters move at each step. A larger stride means skipping more pixels, leading to an output of less dimensions. In Figure 2.1 a stride of two is being used, which means that the filter is skipping one column every time. There is also the concept **padding**. When we don't want information at the edge of an image get lost, we add an extra layer of values all around the image so the filter can get closer to the edges.

2.4 Generative Adversarial Networks

The idea of creating a Generative Network is to train a model over a training set, in our case spectrograms, so that it can generate new images that look like the original ones, but are different. That way, we will have new images that seem realistic. Generative Adversarial Networks (GANs) consist of two different networks, the Generation and the Discriminator. Those two work together but competitively. They are actually playing a zero-sum non-cooperative game. Each one's goal is to get better at the game to win the other one, and if one wins, the other one loses. In game theory, this game is also called minimax. The Generator is generating "fake" images and the Discriminator is trying to tell if the images given are "real" or "fake". So as the Generator is trying to fool the Discriminator, and the Discriminator tries not to be fooled, they both get better and better and we end up with more realistic looking images. This model converges when the Discriminator and the Generator reach a Nash equilibrium, that is when the Generator generates images that resemble very much the images from the original dataset, and the Discriminator guesses at 50 per cent confidence that the generated images are real or fake.

Let x be a matrix representing an image, $D(x)$ the Discriminator output whose

values are the probability that x is a real image. Let z be a latent space vector and the function $G(z)$ symbolizes the generator's operation, where it transforms the latent vector z into new images. The objective of G is to approximate the distribution from which the training data originates p_{data}, allowing it generate fake images based on this estimated distribution p_g. So now the composite function $D(G(z))$ is the probability that the that the output of the Generator is a real image. As mentioned, the Generator and the Discriminator are playing a minimax game, where D aims to maximize the probability of correctly classifying real and fake samples ($\log D(x)$), while G tries to minimize the likelihood of D predicting that its generated outputs are fake ($\log(1 - D(G(z)))$). As presented in [12] our value function will be:

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

2.4.1 Deep Convolutional Generative Adversarial Networks

Deep Convolutional Generative Adversarial Networks (DCGANs) were first introduced by Alec Radford, Luke Metz and Soumith Chintala in 2015 [13]. A DCGAN is a specific type of GAN architecture designed for generating high-quality images. DCGANs are particularly well-suited for tasks such as image generation, style transfer, and image-to-image translation. They were introduced as an improvement over traditional GANs, which struggled to generate coherent and visually appealing images.

Here are some key characteristics and features of DCGANs:

a. Deep Convolutional Layers: DCGANs use convolutional neural networks (CNNs) for both the Generator and Discriminator networks.

b. Strided Convolutions and Fractional Strided Convolutions: DCGANs typically employ strided convolutions in the Generator to upsample the input noise vector into larger images and fractional strided convolutions in the Discriminator to downsample the images, helping to identify features at multiple scales.

c. Batch Normalization: Batch normalization is often applied to stabilize and accelerate training. It normalizes the activations within each mini-batch, reducing issues related to vanishing or exploding gradients.

d. ReLU Activation: Rectified Linear Unit (ReLU) activation functions are commonly used in DCGANs for hidden layers to introduce non-linearity into the networks.

e. No Fully Connected Layers: Unlike some earlier GAN architectures, DCGANs typically do not use fully connected layers in the hidden layers of the Generator and Discriminator, which helps maintain spatial information.

f. Generator Input Noise: The Generator network takes random noise as input, usually from a simple distribution like a Gaussian or uniform distribution. This noise is then transformed into images through the generator's layers.

g. Discriminator Output Activation: The Discriminator produces a single scalar output value for each input image, which is typically passed through a sigmoid activation function to represent the probability of the input image being real (as opposed to generated).

h. Loss Functions: DCGANs typically use the Binary Cross-Entropy loss for training.

This is the Binary Cross Entropy loss function:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^T, \qquad l_n = -[y_n \cdot \log x_n + (1 - y_n) \cdot \log (1 - x_n)] \quad (2.1)$$

where N is the batch size.

Observe how this function computes both logarithmic components within the value function, namely $\log (D(x))$ and $\log (1 - D(G(z)))$. The choice of the specific part of the BCE equation to use is determined by the input y. Although the training loop, which will be discussed later on, handles this specification, it's crucial to grasp the flexibility of selecting the component to calculate by simply modifying y.

The adoption of the Binary Cross Entropy loss (BCELoss) function as the evaluation method for both the discriminator and the generator in the GAN training loop is grounded in its compatibility with the nature of the adversarial framework. Binary Cross-Entropy Loss is a well-suited metric for tasks involving binary classification, precisely the role of the discriminator in a GAN. For the discriminator, this loss function assesses the dissimilarity between its predictions and the ground truth labels, which are binary (real or fake). By optimizing the discriminator using BCELoss, the model is steered towards effectively distinguishing between genuine and generated data.

Similarly, the use of BCELoss for the generator aligns with the adversarial objective. The generator's purpose is to create synthetic data that is convincingly indistinguishable from real data, prompting the discriminator to struggle in making accurate classifications. By employing BCELoss for the generator, the training

process guides the generator to adjust its parameters in a way that minimizes the difference between the discriminator's predictions and the desired output labels.

Unfortunately, GANs are very quirky during training. Sometimes they can never reach the desirable equilibrium point and the Generator just generates images that kind of look like realistic ones, but not quite. Other times, the Generator collapses entirely and generates pure static gray images while the Discriminator classifies these images as fake. Other times the Generator gives us images that are original, but not so distinct from one another. Twitching the hyper-parameters just a little bit can make a huge difference in the outcome.

DCGANs have been widely used in various applications, including image synthesis, image super-resolution, style transfer, and more. DCGANs have played a significant role in improving the quality of generated images and have become a foundational model in the field of deep generative modeling.

2.5 Denoising Diffusion Probabilistic Models

Denoising Diffusion Probabilistic Models (DDPMs) are also a type of Generative Network and we use them in order to generate new realistic images. This process begins with a simple probability distribution, such as the Gaussian distribution or the uniform distribution. The forward diffusion process is applied to an image from the dataset. This process iteratively adds noise to the image, while also preserving some of the important features of the image, over a series of discrete time steps. The key characteristic of these transformations is that they are typically designed to be reversible, meaning we can both apply the transformation and reverse it. Then we give this noisy image to the model along with the quantity of the time steps. This process iteratively removes noise from the image, while also trying to preserve the important features of the image. Then, the loss function is calculated between the original noisy image and the image generated by the reverse diffusion process. Last, we update the weights of the DDPM in order to minimize the loss function. This process repeats until the loss function converges to a small value. [14]

Diffusion models have gained attention for their ability to generate high-quality, realistic images and samples, particularly in the context of image synthesis. They have been used in applications such as image denoising, inpainting, and unconditional image generation. One of the advantages of diffusion models is their ability to capture complex, multi-modal distributions. They have been part of advancements in the field of generative modeling, alongside Generative Adversarial Networks.

Chapter 3

Dataset

3.1 Spectrograms

So far, we have mentioned that our goal is to generate new original sound. But how are we going to do that using the methods described above? We are going to use spectrograms. But what are spectrograms? Let's see.

In just one sentence, a spectrogram is a 2D representation of a sound. A spectrogram is usually shown as a graph, where the horizontal axis represents time and the vertical axis represents frequency. Sometimes it is even shown as a heat map, where the color or brightness indicates the amplitude of a particular frequency at a particular time [15]. Spectrograms are useful for identifying different sounds, such as speech, music, or environmental noise. Spectrograms can also be used to analyze the structure of sounds, such as the harmonics of a musical instrument. Research-wise they are used for speech emotion recognition [15], music genre recognition [16], speaker identification [17], animal species detection [18] and so much more. Let's see now how we make spectrograms.

First of all, let's load one song into our work space using the librosa library.

```
1 samples, sampling_rate = librosa.load(song_path, sr=8000)
2 len(samples), sampling_rate
```

```
(32400, 8000)
```

We have chosen to downsample our songs to 8000 Hz. That means that our song now has 8000 samples (amplitudes) captured every second. So if we want to get the duration of our song we can just divide the number of samples by the sampling rate.

So we see our duration is 4.05 seconds. Usually when imagining a 2D represen-

```
1 duration=len(samples)/sampling_rate
2 print(duration, "seconds")
```

```
4.05 seconds
```

tation of a sound, we often picture a waveform as we can see in Figure 3.1.

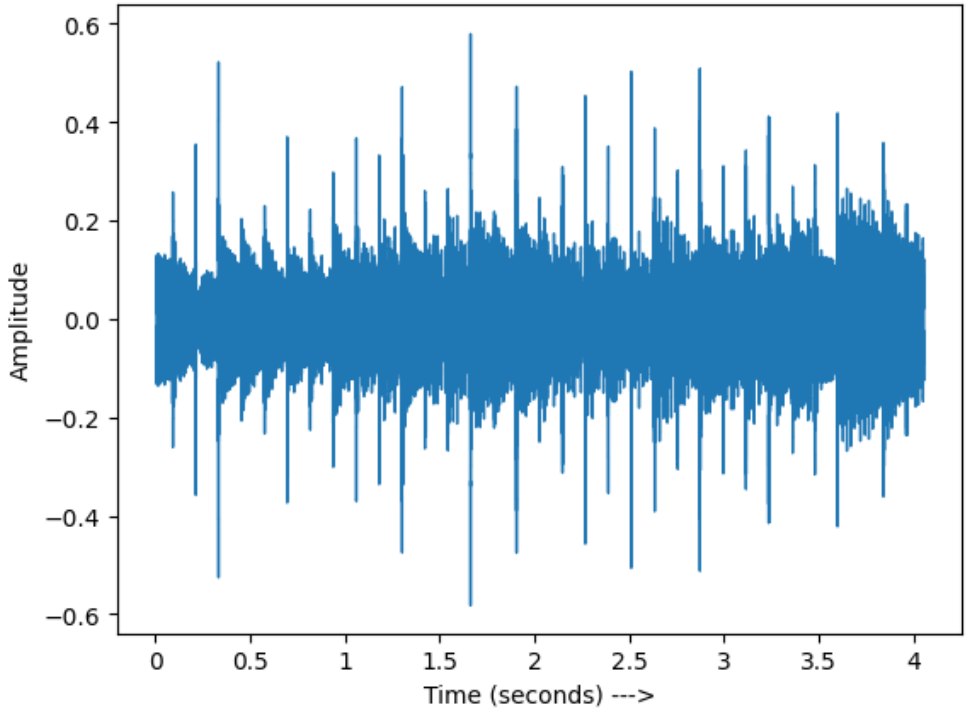

Figure 3.1: Signal Amplitude vs Time

But this is not what we want. Here we actually see a graph where the horizontal axis represents time and the vertical axis represents loudness, so loudness=0 represents silence. This amplitude is really the amplitude of air particles that are oscillating owing to the pressure shift in the atmosphere caused by sound, according to the concept of sound waves. So we see that this is not very didactic as we need to an image that shows the frequencies of our song. We are going to get that by using the Fast Fourier Transform (FFT). The basic idea behind the FFT is to divide the signal into smaller subsequences, and then compute the Fourier transform of each subsequence. The Fourier transform of each subsequence can be computed using the Discrete Fourier Transform (DFT), which is a simpler operation than the Fourier transform. Let x be a finite duration discrete time signal with N samples. The we can compute the DFT like so [19]:

$$\hat{x}(k) = \sum_{n=0}^{N-1} x(n) e^{-i\frac{2\pi k}{N}n}, \quad k = 0, \ldots, N-1$$

And the code looks like this:

```
from scipy.fft import fft
def fft_plot(audio, sampling_rate):
  n=len(audio)
  T=1/sampling_rate
  yf=fft(audio)
  xf=np.linspace(0.0,1.0/(2.0*T), n//2)
  fig, ax=plt.subplots()
  ax.plot(xf, 2.0/n*np.abs(yf[:n//2]))
  plt.grid()
  plt.xlabel("Frequency --->")
  plt.ylabel("Magnitude")
  return plt.show()
```

This will return a list of complex-valued amplitudes representing the frequencies present in the input audio signal. The resulting list is symmetric, with the first half corresponding to positive-frequency terms and the second half to negative-frequency terms, mirroring the positive ones. To visualize the frequency content, we plot the magnitude-frequency graph in Figure 3.2. By taking the absolute values and scaling them appropriately, the graph reveals the magnitudes of different frequencies in the signal.

First let's mention that our Frequencies range from 0 to 4000Hz as our song is sampled at 8000Hz, so based on the Nyquist sampling theorem [20] we only have 8000/2 frequencies in our song.

So now we have our frequencies but we still need time information. After applying the DFT to our song, we only took the frequency values and not the time information, which is essential to a song, otherwise we would just have noise on top of noise. The way we are going to do that, is to split our song in small segments and calculate the DFT for each segment, so that we will get all the frequencies for each segment and the segment itself will represent the time information, because we will know the series of the segments. In general it is a good practice to have overlapping segments, to make sure that we only lose as little information as possible. For our problem we have chosen segments of 30ms that are overlapping with each other. Our spectrogram will look like as shown in Figure 3.3:

So now we can finally see our Frequencies as a function of time. In this graph, brightness represents magnitude, that is, the brighter the colors, the strongest the

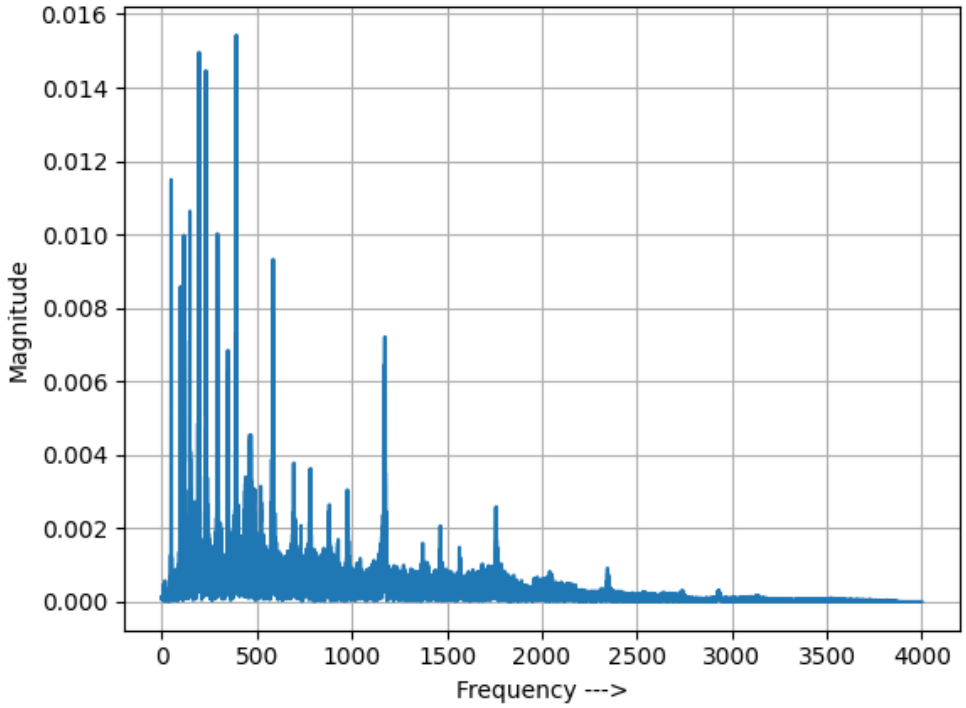

Figure 3.2: Signal Magnitude vs Frequency

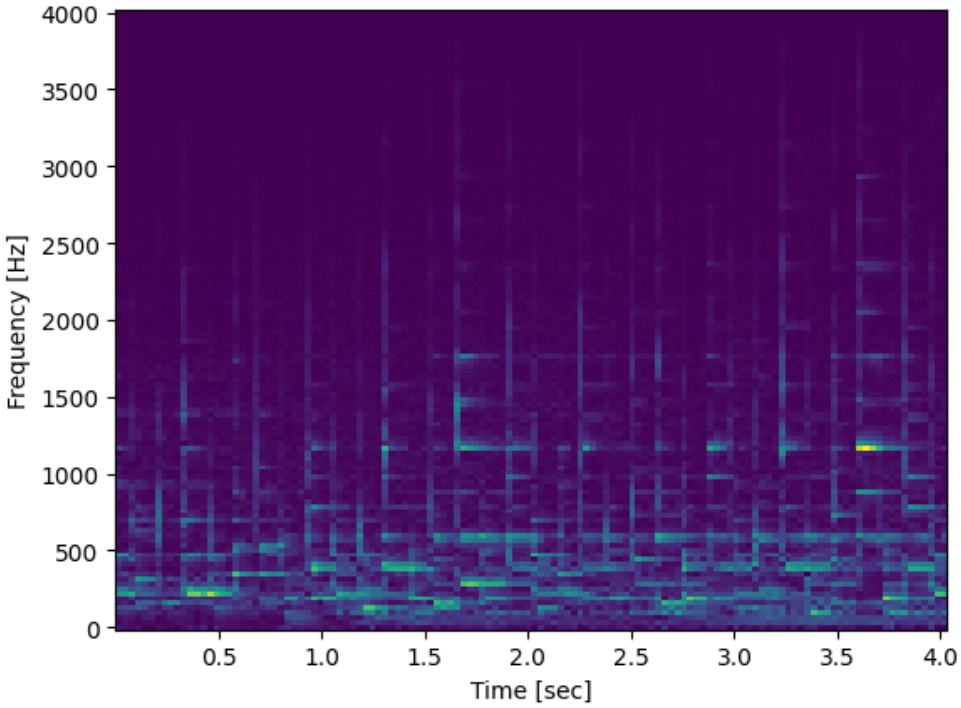

Figure 3.3: Spectrogram

frequency, while darker colors represent lower magnitudes. Of course the colors are used only for visualization purposes. A "true" spectrogram in grayscale still contains the same information but may be less intuitive for our eyes to analyze, especially when dealing with complex signals. Our eyes are better at distinguishing variations in color and brightness than variations in grayscale. But the spectrograms we will be making are going to be grayscale.

3.2 Electronic music

What is electronic music? It's a big change in how we express music. It shows how people always want to be more creative and innovative. Even though it might seem like something recent, electronic music has deep roots in history. To figure out how we went from the natural, nature-inspired beginnings of music to the exciting world of electronic sounds, we have to travel through time. We need to look at the big changes in technology, culture, and human imagination that made this special kind of music possible.

Since the beginning of time, music has been produced analogously. From the singing of birds, rivers flowing, trees rustling, to woodpeckers pecking, music has always been a part of nature's symphony. These natural rhythms and sounds served as both inspiration and instruments for early humans. As our ancestors started to understand how to use these natural things to make music, they found simple ways to do it. Clapping hands, beating drums made from animal skins on hollowed-out logs, and whistling wind through reeds were the first things used to make early percussion, wind, and melodic instruments.

As people changed and developed over time, our musical instruments also changed. The careful making of stringed instruments like the harp and lyre took ideas from the flexibility and tightness of natural fibers. Wind instruments, like flutes and panpipes, copied the soft sounds of the wind in the trees. These early changes were important steps toward having more complex and different kinds of music.

However, things really changed when people learned how to use electricity, starting the electronic age. In the 1800s and 1900s, inventors and musicians got more and more interested in electronic technology. The telegraph and telephone, which could send sounds far away, and the phonograph, which could record and play music, got everyone excited. These things set the stage for trying out different ways to change electronic sounds.

One of the most iconic pioneers in this journey was Leon Theremin, who in 1920 invented the Theremin, an electronic musical instrument controlled without physical

contact. This invention marked a profound departure from traditional instruments, as it allowed musicians to create ethereal and otherworldly sounds simply by moving their hands through the air.

Figure 3.4: The Theremin

The mid-20th century witnessed the emergence of musique concrète and electronic music studios, where composers like Pierre Schaeffer and Karlheinz Stockhausen experimented with recorded and synthesized sounds. These new kinds of music laid the groundwork for the electronic music we have today.

At the same time, synthesizers, drum machines, and MIDI (Musical Instrument Digital Interface) technology were being created, changing how we make music. Genres like techno, house, trance, and hip-hop came out of electronic music, making music in ways that were never thought possible before.

In recent years, big improvements in computer programs and digital tools have made it possible for just about anyone to dive into electronic music creation. You don't need to be a professional - if you have a computer, you can start trying your hand at producing and composing. DJs and electronic musicians are taking their performances to new levels with the help of controllers, laptops, and advanced synthesizers. This has opened up the world of electronic music production to a broader audience, giving people from all walks of life the chance to explore their musical creativity. Now, electronic music is a mix of lots of different voices and ideas, creating a lively and always changing world of sound for everyone to enjoy.

In conclusion, electronic music represents a fascinating journey from the organic origins of music in nature to the electrifying possibilities of the digital age. It shows how creative humans are and how we always want to push the limits of artistic expression. As technology keeps getting better, electronic music is a symbol of new ideas and a tribute to how limitless our imagination can be when it comes to sound and music.

3.2.1 Why did we choose electronic music?

Electronic music, particularly in its various subgenres like techno, house, and trance, often features distinct and repetitive melodic loops, rhythmic patterns, and structured arrangements. These characteristics make electronic music an excellent choice for an AI music generation project. Melodic loops, in particular, offer a clear and repetitive structure that facilitates learning for an AI model. The consistent patterns within electronic music provide a predictable framework for the algorithm to understand and replicate.

The repetitiveness of melodic loops in electronic music not only simplifies the learning process for the AI but also allows for the exploration of variations and permutations. AI models can be trained to recognize the patterns within these loops and generate new sequences or variations based on the learned structures. This aligns well with the iterative and generative nature of AI, enabling the system to produce music that is both coherent and innovative.

Moreover, the modular nature of electronic music composition, with its discrete elements like beats, basslines, and synthesizer patterns, lends itself well to the modular architecture of AI models. By training on a dataset rich in electronic music, the AI can learn how different components interact and contribute to the overall composition, leading to the creation of music that adheres to the genre's conventions while introducing novel elements. In essence, electronic music's melodic loops and structured components provide a fertile ground for AI music generation, fostering the development of models capable of producing compelling and genre-appropriate musical pieces.

3.2.2 What are DJ sets?

DJ sets are like musical journeys created by disc jockeys, or DJs who mix and blend different songs to make a continuous and seamless flow of music. In electronic music, this is a big part of the culture. DJs choose songs that go well together in terms of speed, key, and mood, so the music feels like a story.

In electronic music DJ sets, DJs use a technique called beatmatching, where they sync up the speeds of two songs to smoothly switch from one to the other. This keeps the rhythm and energy going for people dancing. DJs also use tricks like crossfading and adjusting the sound to make sure the music doesn't stop, creating a non-stop experience.

DJ sets can be different, some DJs make long sets that slowly build up, while

others focus on quick changes to keep the energy high. The skill of a DJ isn't just about the technical stuff, it's also about understanding what the crowd likes and creating an exciting experience.

For an AI music project, looking at DJ sets helps understand how electronic music is put together, how different songs fit, the ups and downs in the music, and the overall feel of the performance. DJ sets are like a puzzle, and studying them can help AI learn how to create music that captures the vibe and excitement of a DJ set.

3.3 Making our Dataset

To create a good dataset for our model, we focused on capturing the essential elements that define electronic music. Here are key elements that contribute to a robust dataset:

- Melodic Loops and Patterns: We included a variety of melodic loops and patterns commonly found in electronic music. These can range from simple repeating sequences to more complex and evolving melodies.
- Rhythmic Elements: We included diverse rhythmic elements, like drum patterns, percussion loops, and other rhythmic structures commonly used in electronic music. The dataset covers different tempos and styles within the genre.
- Structural Components: We made sure that our dataset includes individual components such as beats, basslines, synth sequences, and effects. This allows the AI model to understand how these elements can be combined to form a cohesive composition.
- Diversity of Subgenres: We included a variety of subgenres within electronic music, such as techno, house, ambient, and trance. This diversity ensures that the AI model learns the nuances and stylistic differences present across the electronic music spectrum.
- We included instances of dynamic changes in intensity, mood, and energy within tracks. Electronic music often features builds, drops, and climaxes, and these dynamic shifts are crucial for creating engaging and compelling compositions.

By curating a dataset that encompasses these elements, our model can learn the fundamental structures and characteristics of electronic music, enabling it to generate music that aligns with the genre's aesthetics and appeal.

Now on a practical level, to make our Dataset first of all we chose three different DJ sets that include all the characteristics described above. The first one has a duration of 1 hour and 54 minutes, the second one is 1 hour and 11 minutes long and the third one is 56 minutes long. All these add up to a duration of 241 minutes or 4 hours and 1 minute. After that, we cut each DJ set into smaller songs that durate 4.05 seconds each. Our goal was to have a dataset of 100000 small songs. To achieve that we applied the following steps.

- For the first DJ set
  - Downsample the whole DJ set to 8000Hz
  - Extract the first 4.05 seconds
  - While imagining the song's duration in a straight line, slide 136.55 ms to the right and extract another 4.05 seconds
  - Repeat sliding and extracting until the end of the DJ set.
- For the second DJ set
  - Downsample the whole DJ set to 8000Hz
  - Extract the first 4.05 seconds
  - While imagining the song's duration in a straight line, slide 172.51 ms to the right and extract another 4.05 seconds
  - Repeat sliding and extracting until the end of the DJ set.
- For the third DJ set
  - Downsample the whole DJ set to 8000Hz
  - Extract the first 4.05 seconds
  - While imagining the song's duration in a straight line, slide 134.21 ms to the right and extract another 4.05 seconds
  - Repeat sliding and extracting until the end of the DJ set.

This process produced 49958 small songs from the firs DJ set, 24694 small songs from the second one and 25035 small songs for the third one respectively.

Now we have a total of 99687 small songs that are 4.05 seconds long each. To continue making our dataset we need to make the corresponding spectrograms of these small songs. To do that we cut our small songs into 30ms segments that overlap with each other by 0.9%, as described earlier. This produces a total of

99687 images and their size is 128x128. As we mentioned earlier, the spectrograms we will be using are going to be grayscale. The reason why we tend to visualize spectrograms using colors is because it is easier for our eyes to understand what it shows. A grayscale 128x128 image is actually a 128x128 matrix that takes values from the range [0,255]. If we were to look at the spectrograms exactly the way they are produced with our naked eye, we would be looking at completely black images, as their values are really close to 0. For example, in Figure 3.5 we can see 64 spectrograms from our dataset.

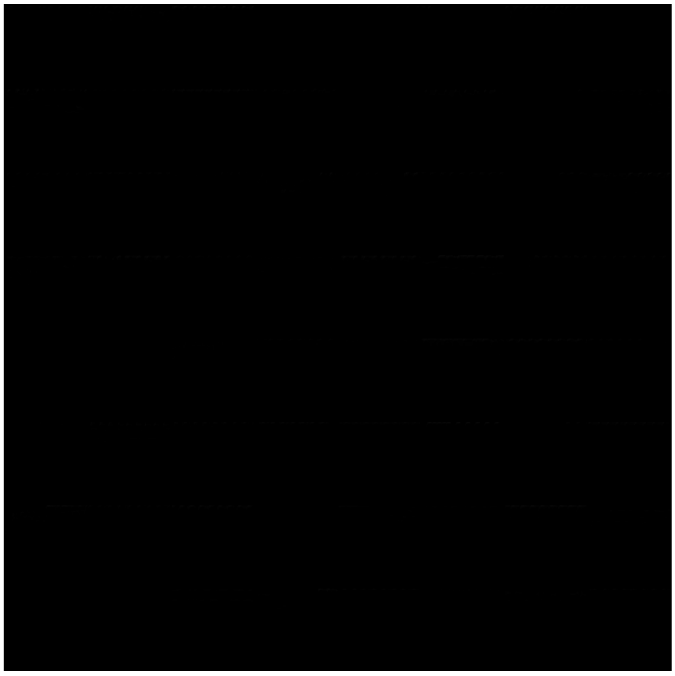

Figure 3.5: 64 non-normalized spectrograms.

But this is not true for a computer. The computer of course doesn't "see" images, it simply reads numbers. So just for visualization purposes we are going to normalize the images in a wider range, so we can see what they look like in Figure 3.6.

That looks better. For our task which is generating new original music, we are going to use our whole dataset as the training set.

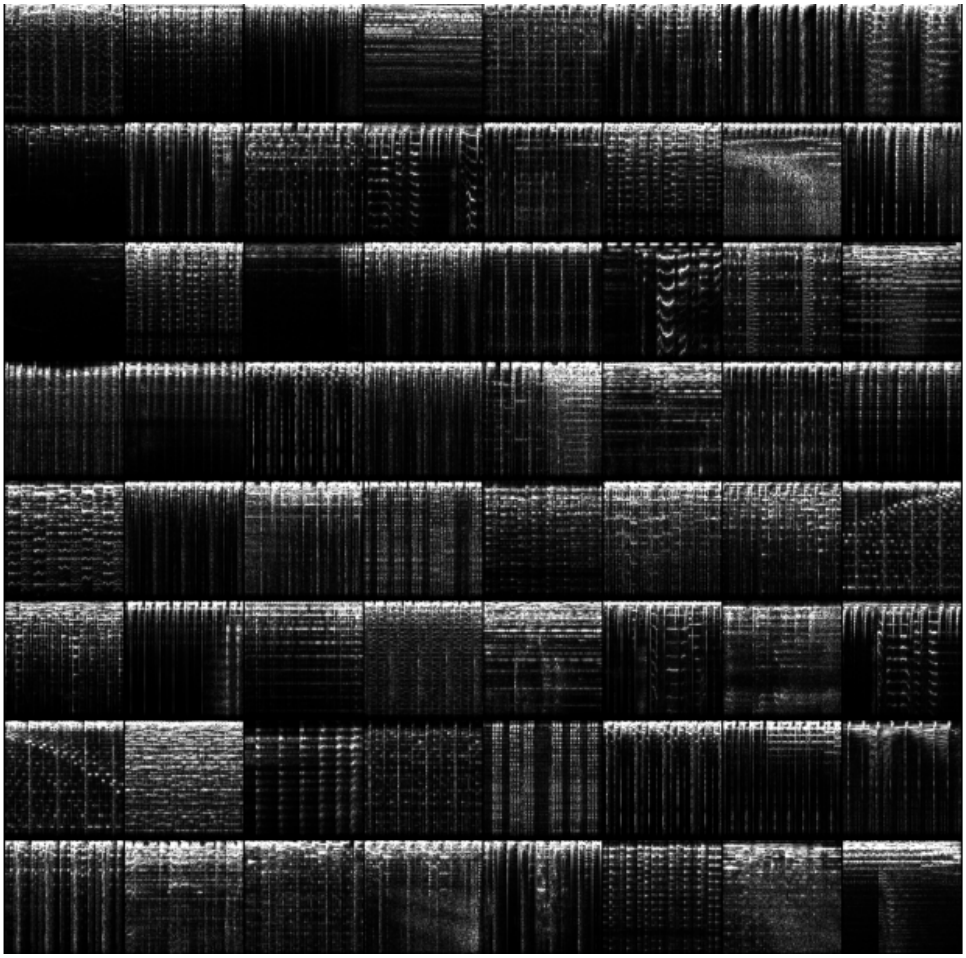

Figure 3.6: 64 normalized spectrograms.

Chapter 4

Pipeline

In this chapter we present the model architectures we used for our experiments. We are going to present four experiments with various results and we are going to show our evaluation for them.

Let's begin from our Generators architectures. In summary, all of our Generators take a random noise vector of size 100 as input and gradually upscale it through a series of transposed convolution layers with Batch Normalization and ReLU activations which help in stabilizing the training process and improving the performance of the network. Batch Normalization and ReLU activations play crucial roles in stabilizing the training process and enhancing the performance of neural networks.

During training, Batch Normalization normalizes the inputs of each layer, helping to control changes in the data distribution known as internal covariate shift. This adjustment works like a stabilizer, keeping the optimization process on a more even and predictable path. By reducing dependency on careful weight initialization, stabilizing gradients, and implicitly introducing regularization through the normalization process, it contributes to more stable and efficient training. On the other hand, ReLU activations introduce non-linearity to the model, making easier the learning of complex, non-linear relationships in the data. Together, the combination of Batch Normalization and ReLU creates a stable and efficient training environment, enabling neural networks to converge faster and generalize better [12], [21], [22].

Let's continue with our Discriminators architectures. Beginning with a 128x128 input image, the network utilizes a sequence of convolutional layers with Leaky ReLU activations and Batch Normalization to progressively downscale the input aiming to make a binary classification (real or fake) in the final layer.

Leaky ReLU adds a slight slope to the negative part of the function, which

means it doesn't shut down completely when the input is negative. This small slope prevents what's called "dead neurons," which are units in the network that stop learning because they always output the same value. By allowing a small flow of information for negative inputs, Leaky ReLU ensures that no parts of the discriminator don't become inactive, promoting continuous learning and making it easier for training signals (gradients) to pass through the network during the training process. Essentially, it helps the discriminator stay more alert and responsive to different patterns in the data, contributing to more effective training [13]. Finally, the sigmoid activation function is used in the last layer to allow the discriminator to output a probability between 0 and 1, which represents the likelihood that an input image is fake or real.

In total, our Generators and our Discriminators architectures use 6 layers, which is just enough in order to strike a balance between having enough capacity to capture complex patterns in the data and maintaining stability during training. The final output is a 128x128 grayscale image. Architectures similar to these have shown empirical success in various GAN applications [12], [23], [24].

4.1 First model

Now let's look at our first Generator's architecture:

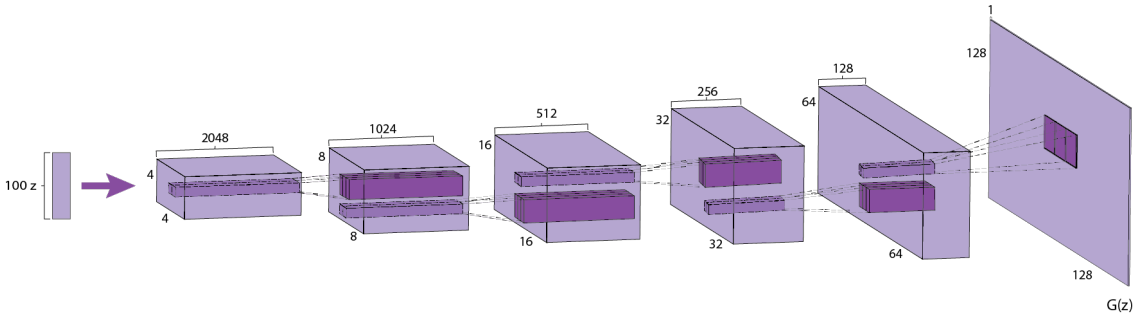

Figure 4.1: First Generator's architecture - Figure

Generator's architecture	
Layer 1 - Input Layer	Input Size: z, a random noise vector of size 100. ConvTranspose2d(100, 2048, kernel_size=4, stride=1, padding=0) Batch Normalization ReLU Activation Output Size: (2048, 4, 4)
Layer 2	Input Size: (2048, 4, 4) ConvTranspose2d(2048, 1024, kernel_size=4, stride=2, padding=1) Batch Normalization ReLU Activation Output Size: (1024, 8, 8)
Layer 3	Input Size: (1024, 8, 8) ConvTranspose2d(1024, 512, kernel_size=4, stride=2, padding=1) Batch Normalization ReLU Activation Output Size: (512, 16, 16)
Layer 4	Input Size: (512, 16, 16) ConvTranspose2d(512, 256, kernel_size=4, stride=2, padding=1) Batch Normalization ReLU Activation Output Size: (256, 32, 32)
Layer 5	Input Size: (256, 32, 32) ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1) Batch Normalization ReLU Activation Output Size: (128, 64, 64)
Layer 6 - Output Layer	Input Size: (128, 64, 64) ConvTranspose2d(128, 1, kernel_size=4, stride=2, padding=1) Tanh Activation Output Size: (1, 128, 128)

Table 4.1: First Generator's architecture

Let's continue with our first Discriminator's architecture.

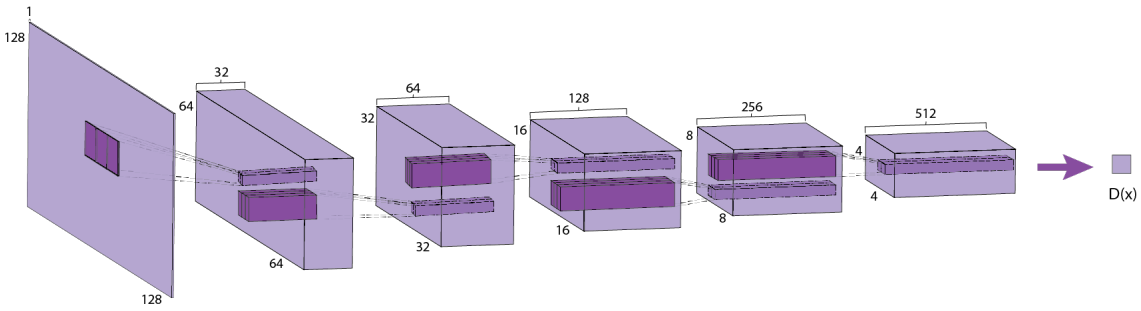

Figure 4.2: First Discriminator's architecture - Figure

Discriminator's architecture	
Layer 1 - Input Layer	Input Size: (1, 128, 128), a grayscale image. Conv2d(1, 32, kernel_size=4, stride=2, padding=1) Leaky ReLU Activation with a negative slope of 0.2 Output Size: (32, 64, 64)
Layer 2	Input Size: (32, 64, 64) Conv2d(32, 64, kernel_size=4, stride=2, padding=1) Batch Normalization Leaky ReLU Activation with a negative slope of 0.2 Output Size: (64, 32, 32)
Layer 3	Input Size: (64, 32, 32) Conv2d(64, 128, kernel_size=4, stride=2, padding=1) Batch Normalization Leaky ReLU Activation with a negative slope of 0.2 Output Size: (128, 16, 16)
Layer 4	Input Size: (128, 16, 16) Conv2d(128, 256, kernel_size=4, stride=2, padding=1) Batch Normalization Leaky ReLU Activation with a negative slope of 0.2 Output Size: (256, 8, 8)
Layer 5	Input Size: (256, 8, 8) Conv2d(256, 512, kernel_size=4, stride=2, padding=1) Batch Normalization Leaky ReLU Activation with a negative slope of 0.2 Output Size: (512, 4, 4)
Layer 6 - Output Layer	Input Size: (512, 4, 4) Conv2d(512, 1, kernel_size=4, stride=1, padding=0) Sigmoid Activation Output Size: 1

Table 4.2: First Discriminator's architecture

Describing the Training Process

We are now going to present the training process of our Generative Adversarial Network and see how the generator tries to get better at fooling the discriminator and how the discriminator gets better at distinguishing between fake and real images.

a. Training Loop: An outer loop runs for a specified number of epochs. A inner loop iterates over batches of data.

b. Discriminator Update (Step 1):

- For real data:
  - Gradients are initialized.
  - The batch of real data and labels are prepared. Our real label will be 1 and our fake one will be 0. These will be used for loss calculation purposes.
  - Forward pass through the discriminator with real data.
  - Discriminator loss is calculated based on the output and real labels.
  - Gradients are calculated.
  - The mean of the discriminator output for real data is calculated.
- For fake data:
  - Random noise is generated.
  - Fake data is generated using the generator.
  - Fake data is labeled as fake.
  - Forward pass through the discriminator with fake data.
  - Discriminator loss is calculated based on the output and fake labels.
  - Gradients are calculated.
  - The mean of the discriminator output for fake data is calculated.
- The total discriminator loss is calculated and the discriminator is updated.

c. Generator Update (Step 2):

- Gradients for the generator are initialized.
- Fake data is labeled as real.
- Forward pass through the discriminator with the generated fake data.
- Generator loss is calculated based on the output and real labels.

- Gradients are calculated.
- The mean of the discriminator output for the generated fake data is calculated.
- The generator is updated.

d. Go back to Step 1

Now let's talk about how we calculate our loss function. For the discriminator, we want to maximize $\log(D(x)) + \log(1 - D(G(z)))$ [12]. To achieve that, we calculate it in two steps. First step is to calculate the Discriminator's loss for real data/labels by substituting by $y = 1$ and $x = D(x)$ in function (2.1) which gives us $\log(D(x))$. The second step is to calculate the Discriminator's loss for fake data/labels by substituting by $y = 0$ and $x = D(G(z))$ in function (2.1) which gives us $\log(1 - D(G(z)))$.

For the Generator, we want to minimize $\log(1 - D(G(z)))$. But as described in [12], to get better results we are instead going to try to maximize $\log D(G(z))$. In order to do so, we are going to calculate the Generator's loss for real data/labels by substituting by $y = 1$ and $x = D(G(z))$ in function (2.1) which gives us exactly $\log D(G(z))$.

Our goal when training our GAN is not to necessarily minimize the loss functions as much as possible, but reach convergence. Convergence is often evaluated by tracking how the generator and discriminator losses change over the training process. Convergence is specifically thought to have occurred when the discriminator and generator losses stabilize and show little variability. In terms of numbers, this is frequently seen as a plateau or a recurrent oscillation around certain values, signifying that the generator and discriminator have both reached a point where further training doesn't result in significant advancements. The challenging part is to find a balance between believing we have reached convergence and stopping the training process far too soon which results in a model that is not as good, and having reached convergence but continuing to train the model which might lead to training and using up unnecessary compute units, or even losing convergence.

4.1.1 Hyperparameters

We now need to specify the parameters used for this model.

Data Normalization

First of all, before feeding our data to this model, we normalized them. In the context of image processing, data normalization is a technique used to scale the pixel values of an image to a specific range. The goal is to ensure that the pixel values lie within a standardized range, making it easier for the model to learn and improving the convergence of the training process. In our case we have used the MinMaxScaler found in the scikit-learn library, to normalize our data in the range of $[0, 1]$. The Min-Max scaling, also known as feature scaling or normalization, is a simple linear transformation that scales the input values to a specified range. So let x be a pixel value from one of our images, that is actually a value from our 128x128 matrix. Then:

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}} * (feature\_range_{max} - feature\_range_{min}) + feature\_range_{min}$$

where:

- x is the original pixel value
- x_{min} is the minimum pixel value in the image
- x_{max} is the maximum pixel value in the image
- $feature\_range_{min}$ is the minimum value of the desired range, 0 in our case
- $feature\_range_{max}$ is the maximum value of the desired range, 1 in our case
- x_{scaled} is the scaled pixel value.

Training Hyperparameters

Now for the other parameters we have to specify, we set our batch size to be 100, our learning rate to be 0.0002, our optimizer is the Stochastic Gradient Descent, and our latent vector is filled with random numbers sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1 (randn), and we train our model for 180 epochs.

Parameters	
Epochs	180
Batch size	100
Learning rate	0.0002
Optimizer	SGD
Noise	randn

Results

Now let's look at our results in Figure 4.3. First, let's look side by side a bunch of real images from our dataset and a bunch of generated or "fake" images from our model. Like mentioned earlier, even with this normalization that we have performed, the real images still look almost totally black with some white-gray elements.

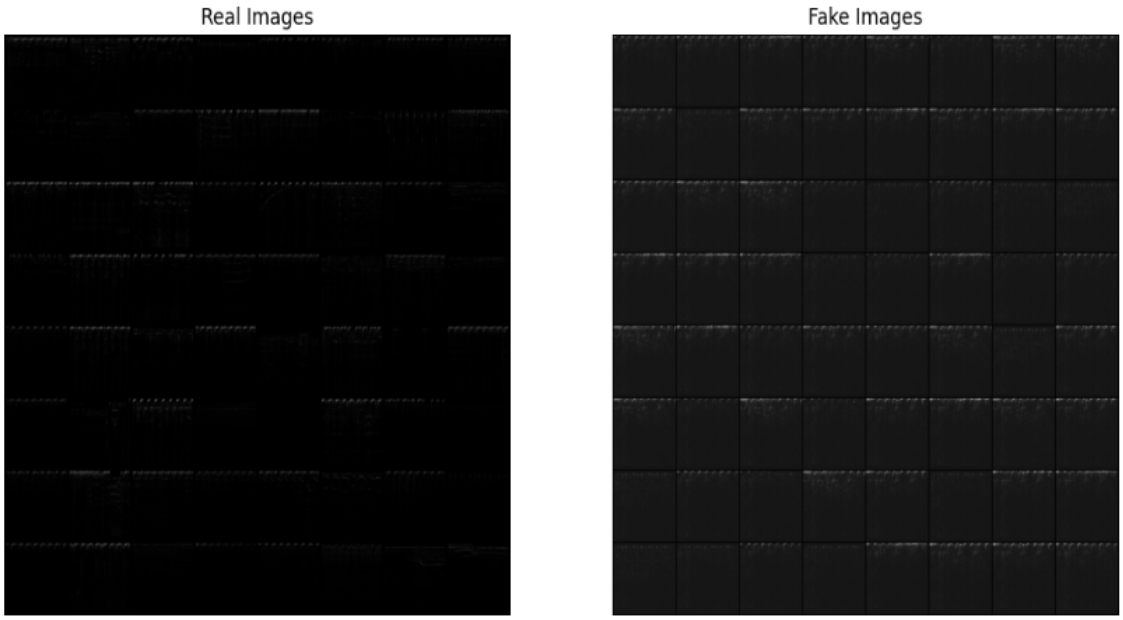

Figure 4.3: First model's Real vs Fake images.

Now let's look at the corresponding loss functions for the Generator and the Discriminator in Figure 4.4. This is a graph that shows the loss as it progresses with respect to the epochs of the model. In this case, we can see that the loss functions start converging a little bit after the 50th epoch, the generator loss oscillates around 5 and the discriminator loss oscillates around 2.5, while intertwining with each other. For this model, we ascertained that if trained for more epochs, it doesn't generate better results and if trained for less epochs, the results are not as good.

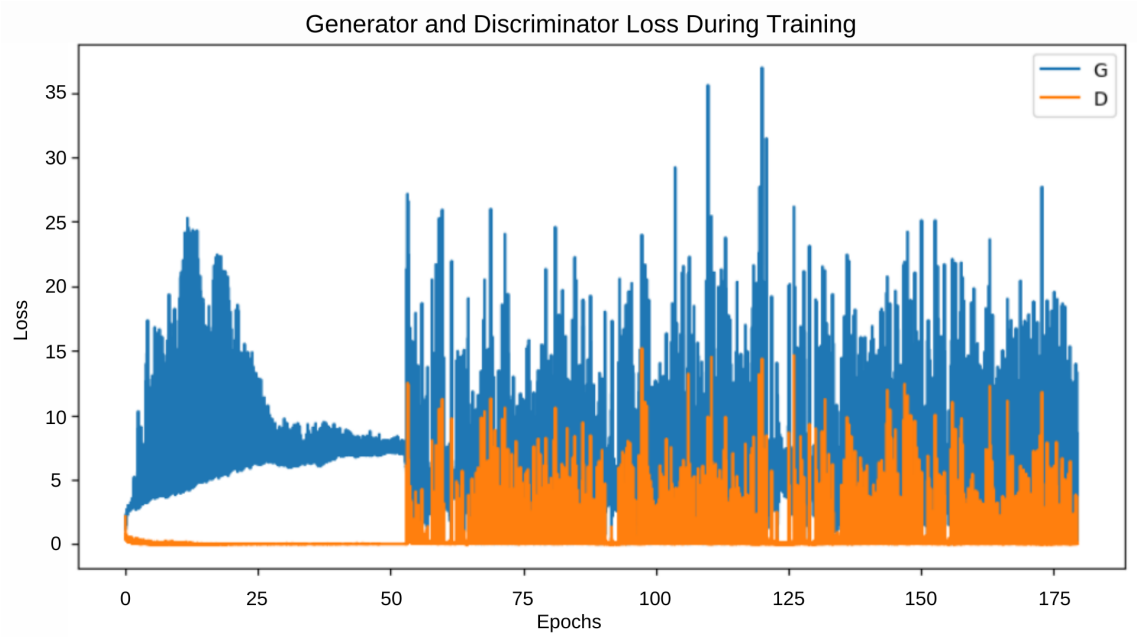

Figure 4.4: First model's Loss Functions.

4.2 Second model

For our second model, the architecture we used is exactly the same as the previous one so we are not going to repeat it. The key difference in this model, is not in the training process, but in the data. This time, we decided against normalizing our data and to leave them as-is.

We run our model for 150 epochs with the same exact hyperparameters as before.

Parameters	
Epochs	150
Batch size	100
Learning rate	0.0002
Optimizer	SGD
Noise	randn

As we can see from the corresponding graph of our losses in Figure 4.5, they started converging at the 70th epoch and they continued to do so, the generator oscillates around 9 and the discriminator oscillates around 2 and they are again intertwining with each other. We again made sure that training the model for less epochs does not generate good enough results, and training it for more epochs does not make a difference in the quality of the results, they are even a little worse.

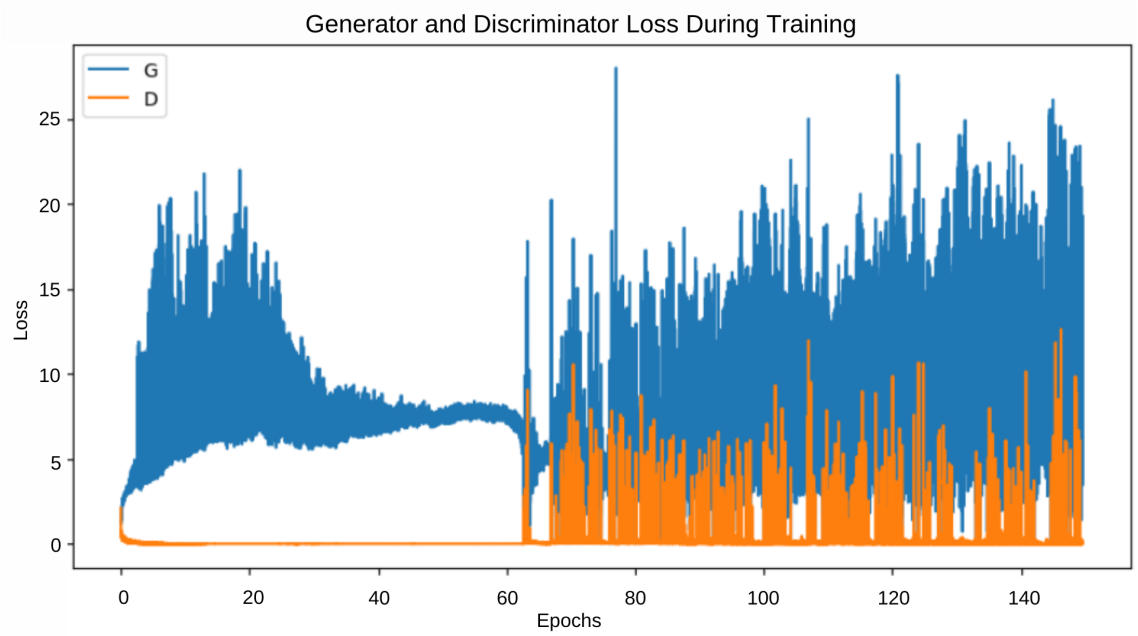

Figure 4.5: Second model's Loss Functions.

Now let's look at some images from this generator in Figure 4.6. As mentioned before, the real images that are not normalized, will look pure black to our eyes.

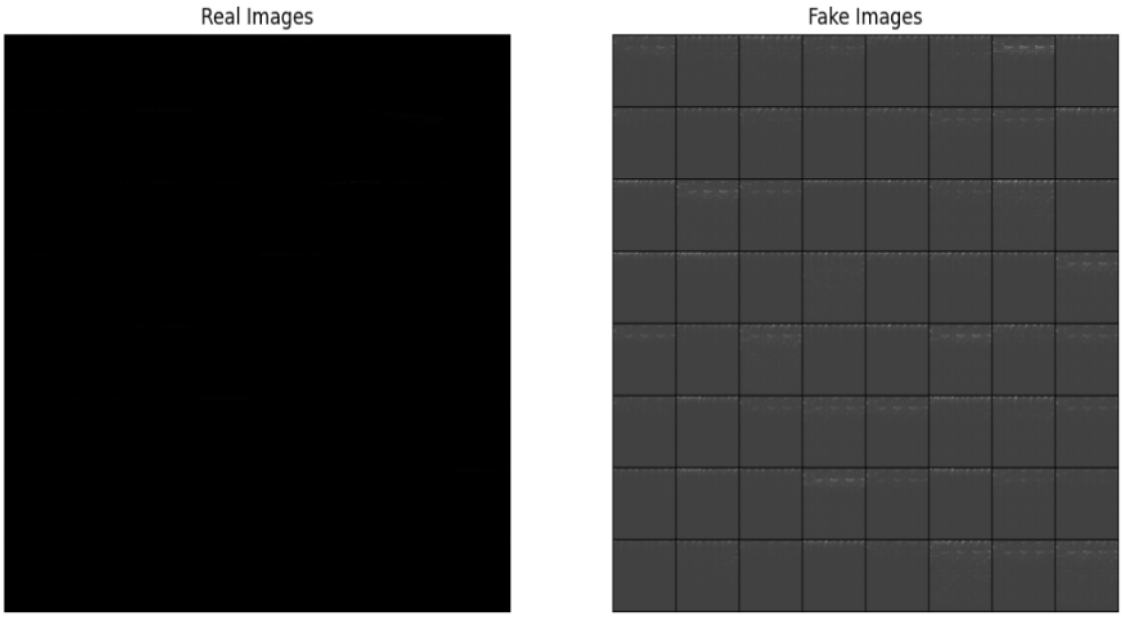

Figure 4.6: Second model's Real vs Fake images.

And if we zoom in to look at a couple of images we can see clearer the results in Figure 4.7 and Figure 4.8.

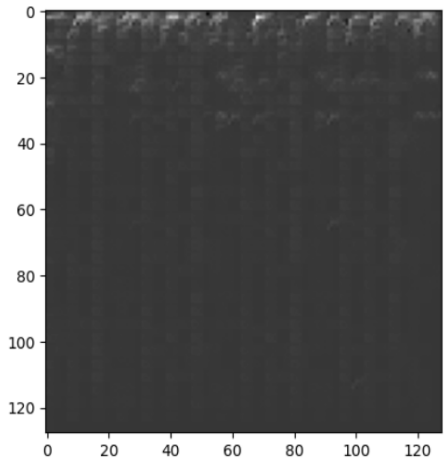

Figure 4.7: Example 1

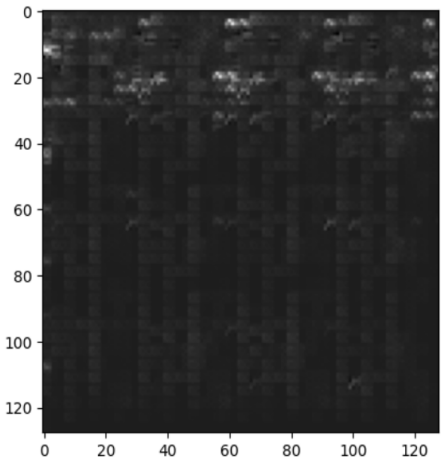

Figure 4.8: Example 2

Generator Layer sizes		
Layers	1st and 2nd model	3rd and 4th model
Input	100	100
Layer 1 - Output	(2048, 4, 4)	(1024, 4, 4)
Layer 2 - Output	(1024, 8, 8)	(512, 8, 8)
Layer 3 - Output	(512, 16, 16)	(256, 16, 16)
Layer 4 - Output	(256, 32, 32)	(128, 32, 32)
Layer 5 - Output	(128, 64, 64)	(64, 64, 64)
Layer 6 - Output	(1, 128, 128)	(1, 128, 128)

Table 4.3: Generator Layer sizes

4.3 Third model

For our third model and for our forth one, we changed a little bit the architecture of both the Generator and the Discriminator. This Generator again takes a random noise vector of size 100 as input and gradually upscales it through a series of transposed convolution layers with batch normalization and ReLU activations. It also totally uses 6 layers, same as before. The final output is again a 128x128 grayscale image. What is different is the size of the feature maps in our layers and thus the input and output size in each layer. Choosing to reduce the number of feature maps in our model intends for our model to get better at figuring out big-picture ideas and important details. It also helps in making the generated images look more realistic and connected. [13] In table 4.3 we can see side by side the input/output sizes of the first two generators' architecture with the third and forth ones.

Let's continue with our Discriminator's architecture. Beginning with a 128x128

Discriminator Layer sizes		
Layers	1st and 2nd model	3rd and 4th model
Input	(1, 128, 128)	(1, 128, 128)
Layer 1 - Output	(32, 64, 64)	(16, 64, 64)
Layer 2 - Output	(64, 32, 32)	(32, 32, 32)
Layer 3 - Output	(128, 16, 16)	(64, 16, 16)
Layer 4 - Output	(256, 8, 8)	(128, 8, 8)
Layer 5 - Output	(512, 4, 4)	(256, 4, 4)
Layer 6 - Output	1	1

Table 4.4: Discriminator Layer sizes

input image, same as before, this network utilizes a sequence of convolutional layers with Leaky ReLU activations and batch normalization to progressively downscale the input aiming to make a binary classification in the final layer. The difference again is in the size of the feature maps in our layers which change the input and output size in each layer. This is the architecture: In Table 4.4 we can see side by side the input/output sizes of the first two discriminator' architecture with the third and forth ones.

It is worth noting that now that the whole size of our model is smaller, it will have much faster training time. For this model we are going to normalize our data in the range of $[0, 1]$ using the Min-Max scaling as we did before, we are going to train it for 250 epochs, using the same hyperparameters.

Parameters	
Epochs	250
Batch size	100
Learning rate	0.0002
Optimizer	SGD
Noise	randn

Looking at the graph of our losses (Figure 4.9) we can see that they started converging at epoch 60, kept their balance for a little, then disconverged and then met again and kept their balance after epoch 125.

And now we can look at some generated images side by side with the images from our dataset in Figure 4.10.

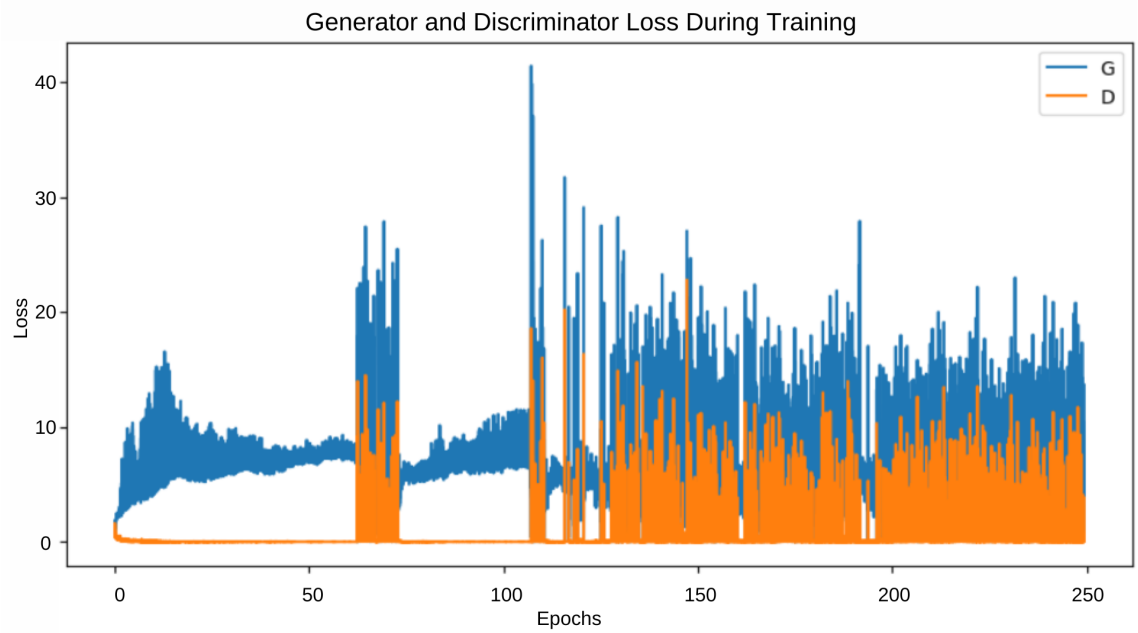

Figure 4.9: Third model's Loss Functions.

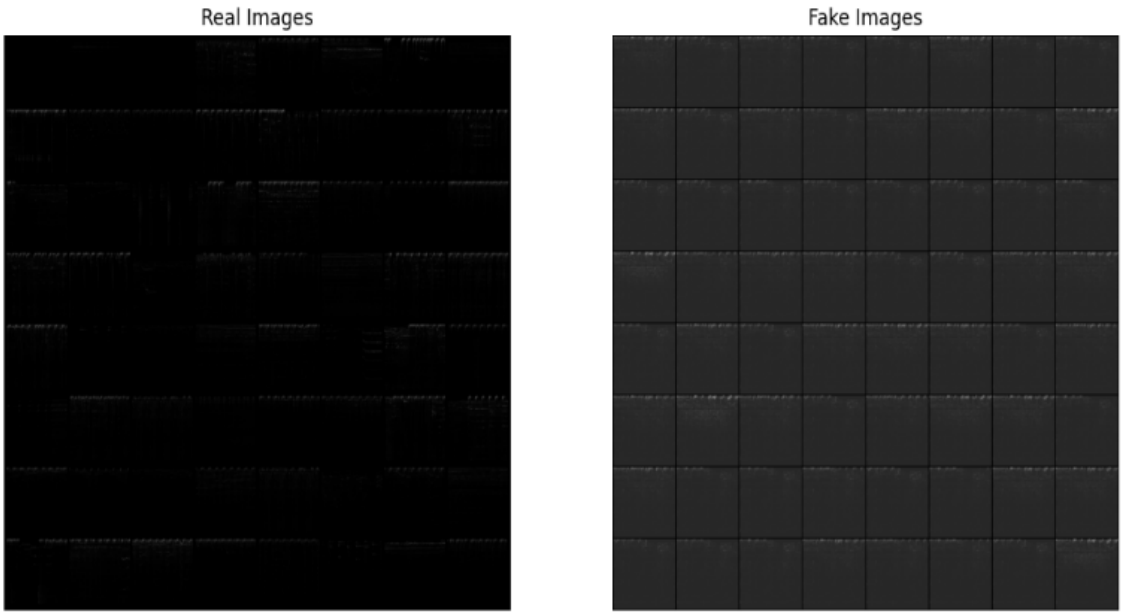

Figure 4.10: Third model's Real vs Fake images.

4.4 Forth model

For our forth and final model we used the same architecture as the third model and this time we decided against normalizing our data. We first run our model for 300 epochs with the same exact hyperparameters as before.

Parameters	
Epochs	300
Batch size	100
Learning rate	0.0002
Optimizer	SGD
Noise	randn

As we can see from the corresponding graph of our losses in Figure 4.11, they did converge at around epoch 100 but then they lost their balance and diverged from each other.

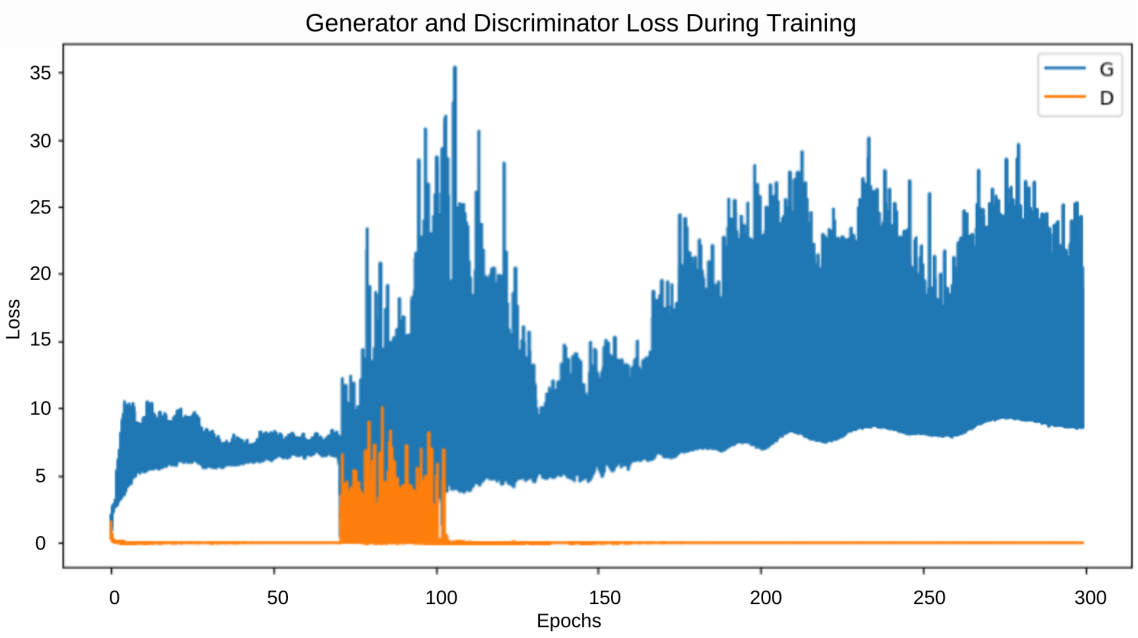

Figure 4.11: Forth model's Loss Functions.

So let's look at the graph of the first 100 epochs in Figure 4.12.

This time they are not converging for as long as they did in the previous models, but they are both converging around 4. Let's look at some images from this generator. As mentioned before, the real images that are not normalized, will look pure black to our eyes.

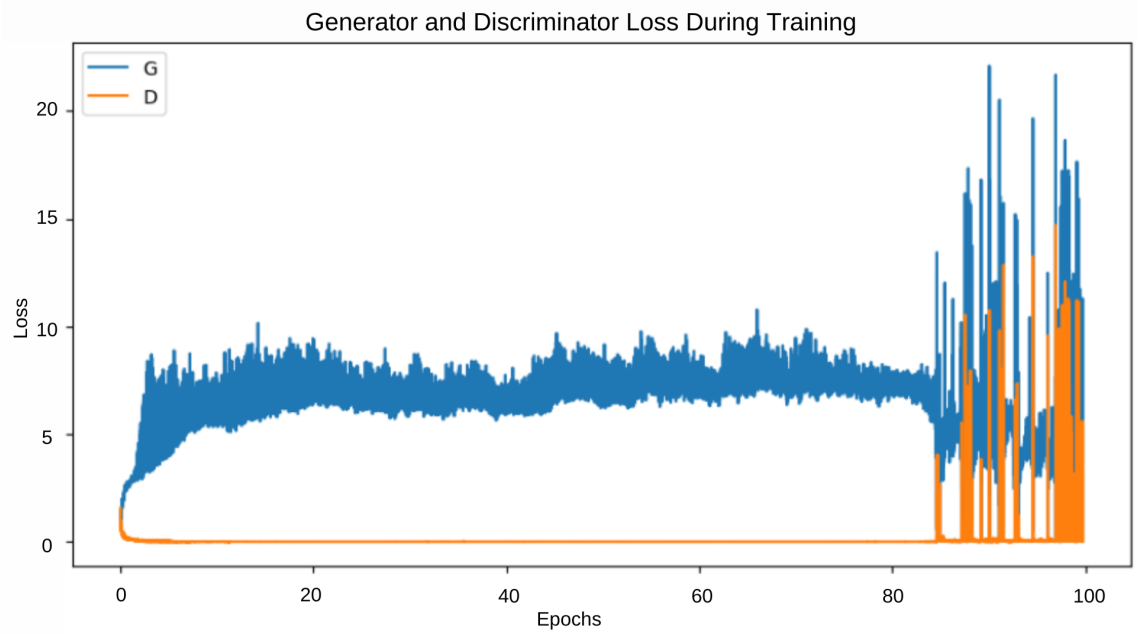

Figure 4.12: Forth model's Loss Functions for the first 100 epochs.

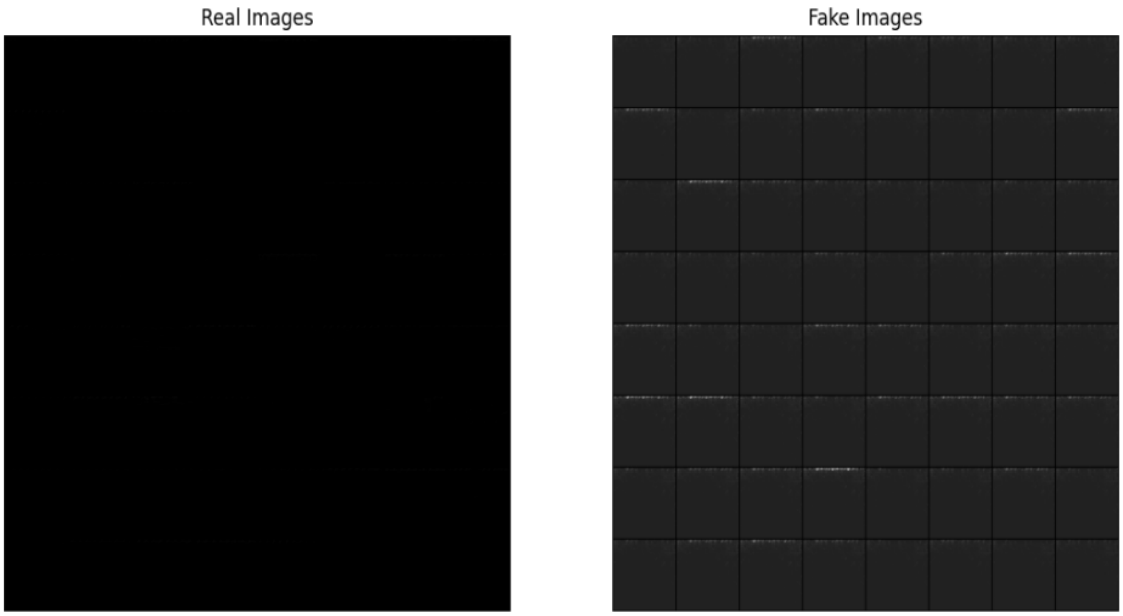

Figure 4.13: Forth model's Real vs Fake images.

And if we zoom in and look at one of the generated images, we can see clearer the details in Figure 4.14.

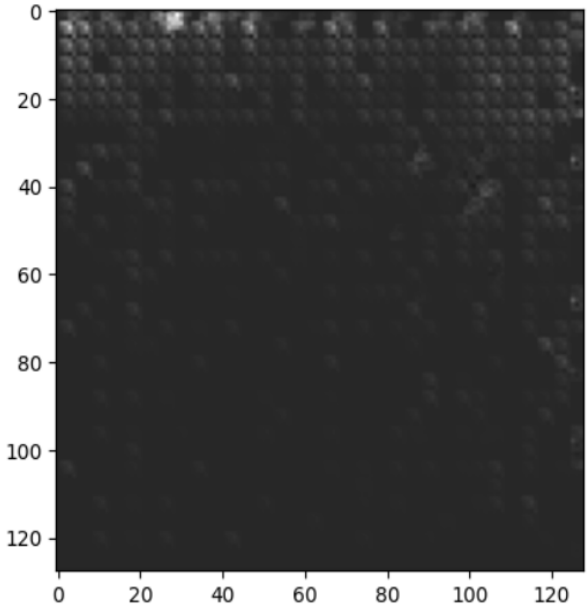

Figure 4.14: Forth model's Example - Zoomed in

The last variation we have for this model, is one where we chose to change the distribution from which the latent vector takes its values. This time our latent vector is filled with random numbers sampled from a uniform distribution on the interval $[0, 1)$. We trained it for 80 epochs.

Parameters	
Epochs	80
Batch size	100
Learning rate	0.0002
Optimizer	SGD
Noise	rand

As we can see from the graph of our losses in Figure 4.15, they have not exactly reached convergence for a long period of time. Despite that, the generator was still able to generate good looking images.

4.5 Comments

To properly comment on our results, we need to define what mode collapse is.

Several challenges commonly afflict GAN models, including:

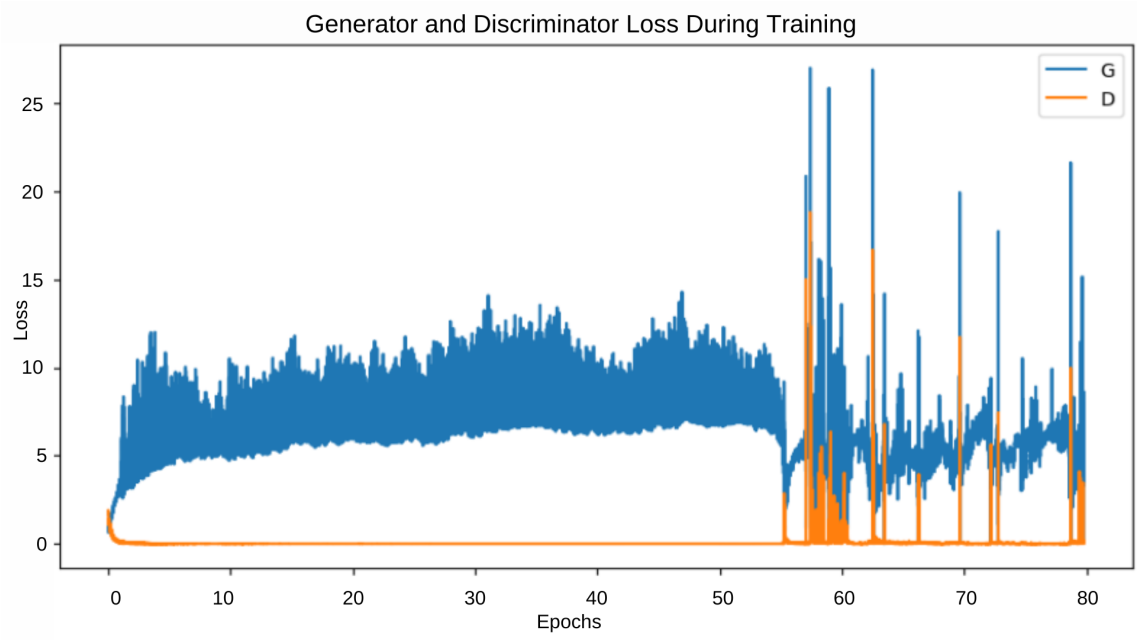

Figure 4.15: Variation of Forth model - Loss Functions.

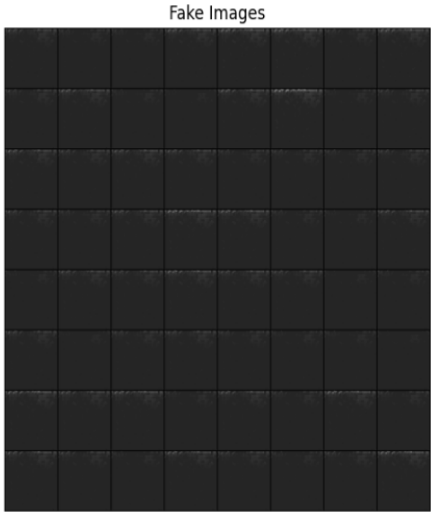

Figure 4.16: Variation of Forth model - Fake images.

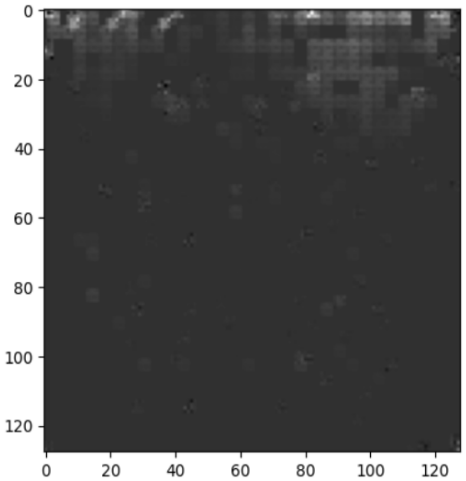

Figure 4.17: Variation of Forth model Example - Zoomed in

- Non-convergence: This issue occurs when model parameters continually oscillate, destabilize, and fail to reach a state of convergence. [25]
- Diminished gradient: In this scenario, the discriminator becomes overly proficient, causing the generator gradient to vanish, impeding its ability to learn effectively. [26] [25]
- Mode collapse: Another prevalent problem involves the collapse of the generator, resulting in the production of a limited range of sample varieties. [27]

In the context of mode collapse in Generative Adversarial Networks (GANs), the generator's attempt to outsmart the discriminator results in a focused set of generated samples. We can picture it as the generator finding quick tricks or strategies to create samples that the discriminator has difficulty distinguishing from real data. Yet, this smart maneuver comes with a drawback, the generator becomes fixated on a limited set of patterns or modes, overlooking the broader diversity that one would expect in the generated samples.

Mode collapse in GANs, also known as the Helvetica scenario, is like a scenario where an artist, instead of drawing various types of trees in a landscape, keeps drawing the same kind over and over. Similarly, in GANs, the generator gets stuck creating a limited set of outputs and doesn't explore the full range of possibilities in the data it's supposed to generate. It's like missing out on all the different kinds of trees and features in the landscape because the generator is fixated on just a few patterns. This happens when the generator doesn't learn to represent the entire diversity of the data distribution, making it less effective in creating a wide variety of samples. [28] [29] For example, in Figure 4.18, we can see generated images from two GANs whose goal was to generate all single digit numbers. In the first row, we see a GAN that has managed to do that successfully and in the second row, we see a GAN that suffers mode collapse and only generates one single shape. [1]

Partial mode collapse is a nuanced variation of mode collapse where the collapse is not absolute, meaning that the generator does produce diverse samples, but there's still a noticeable deficiency in capturing certain modes or variations within the data distribution. In simpler terms, while the generator might be successful in creating a range of different samples, it may not fully cover every possible type of sample. Some modes or patterns in the data distribution might be underrepresented or even absent in the generated samples. This results in a partial loss of diversity, where the generator falls short of capturing the entirety of the target distribution. [28]

So in our case, we have reached convergence in all our models, we do not have diminished gradients, but we do have a partial mode collapse. While each model

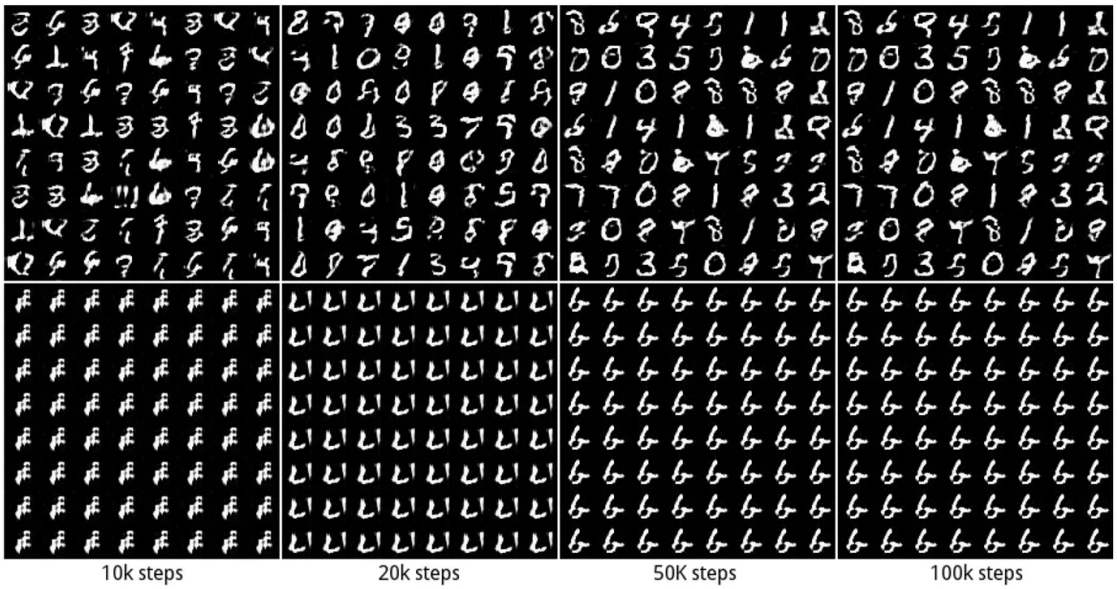

Figure 4.18: Mode collapse example. [1]

generates a diverse set of images, it becomes apparent that these images can be grouped into subsets that share common characteristics. This means that although the generator is producing a variety of outputs, there are certain patterns or styles that are repeated within each subset of generated images.

4.6 From spectrograms to actual sound

Now is the time to describe how our generated spectrograms were used to produce sound.

First of all, if we have a song and find its corresponding spectrogram, reversing that process to get back the exact original song is tricky. The spectrogram is like a detailed snapshot of the song's frequencies and when they happen. But, when we transform the song into a spectrogram, we lose some important details, especially about how the different frequencies relate to each other over time. It's like taking a detailed picture of a landscape and then trying to recreate the exact 3D scene from just the picture. We might get close, but some details will be off.

The process of transforming an audio signal into a spectrogram and then attempting to reverse it back into the original audio is not a perfect one-to-one function. A one-to-one function would mean that each unique input (audio signal) has a unique output (spectrogram) and vice versa. However, due to the nature of the transformation involved, as explained earlier, some information is lost during the creation of the spectrogram.

As mentioned earlier, creating a spectrogram involves dividing the input time-

domain signal into overlapping chunks, applying a window function, and performing a Fast Fourier Transform (FFT) to obtain complex vectors representing amplitude and phase for each frequency bin. The spectrogram's columns are formed by taking the absolute values of these FFT results, discarding the negative frequencies ones. However, due to the symmetric nature of the process, which discards phase information, accurately reconstructing the original time-domain signal becomes impossible. While recreating the exact signal is not achievable because of this loss of phase information, there is an opportunity to generate a signal that sounds very similar. Reversing the process involves attempting the described steps in reverse, introducing a random phase for the FFT.

So this is what we have done. We initialize random phases for each frequency component. This randomization accounts for the loss of phase information during the original spectrogram creation. In the absence of precise phase details, which are crucial for recreating the original time-domain signal accurately, we introduce this randomness to simulate various possibilities. It's like trying out different starting points for each musical note without knowing the exact timing. By doing so, we acknowledge the uncertainty and attempt to capture the overall character of the original signal.

The next step involves iterating through time intervals and frequency components. For each time step, we calculate the contribution of each frequency by combining sinusoidal components with the randomized phases and magnitudes obtained from the spectrogram. The summation of these components at each time point reconstructs the signal in the time domain. Although the exact starting points of the frequencies remain unknown, the process aims to approximate what the audio would sound like.

4.7 Evaluation with People's Opinions

Well it is music after all! So we have to get people and the actual sounds generated involved! So, we made a form that we distributed for people to answer.

The objective of this form was to see how many participants would be able to to distinguish between AI-generated and non-AI-generated songs. To achieve this, we presented a set of three questions. Initially, we chose four generated songs, each lasting about 4 seconds. Then, we selected two additional 4-second segments from our original dataset, extracted their corresponding spectrograms and reverted them to sound using the same process employed for the generated songs. This approach ensured uniform quality across all the presented songs, minimizing potential biases among participants.

The first and the second question featured one "real" song and one AI-generated song, for participants to compare and decide which could be AI-generated. In contrast, the third question involved two AI-generated songs for assessment. The form also included demographic questions concerning the participants' age, their musical background, their familiarity with AI-generated music and lastly, how often they listen to music.

In total, 137 people participated. In all of the questions, a participant could either select Option 1, Option 2, None, or both Option 1 and Option 2. In the first question the AI-generated one was Option 1 and the question was "Which of the above, if any, sound like AI-generated?". In Figure 4.19 we can see the percentages of the choices people made.

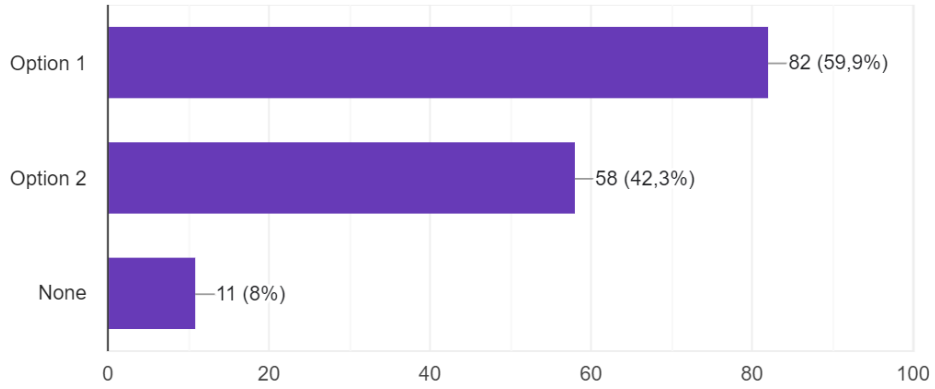

Figure 4.19: Question 1 - Percentages.

We can see that about 60% managed to find the right answer, but this method of evaluation for this form is not very representative, as someone could have chosen more than one answer. This is why we decided to use a Likert-type scale, giving scores to the participants' answers'. For this question, we will say that a participant scored 2 if only the correct answer was chosen, scored 0 if the wrong answer or the "None" option was chosen, and scored 1 if the right answer was chosen along with the wrong answer. In Figure 4.20 we see the percentage of each score, along with their corresponding 95% confidence intervals.

	N	*Percent*	*Confidence Interval*
Score 0	55	40.15%	(31.94, 48.35)
Score 1	14	10.22%	(5.15, 15.29)
Score 2	68	49.63%	(41.26, 58.01)

Figure 4.20: Question 1 - Scores.

So now we notice that while about 50% of the participants managed to score 2,

another 40% scored 0.

In the second question, which was expressed in the same way, the AI-generated one was Option 2. In Figure 4.21 we see that again about 60% of the participants manages to chose the right answer.

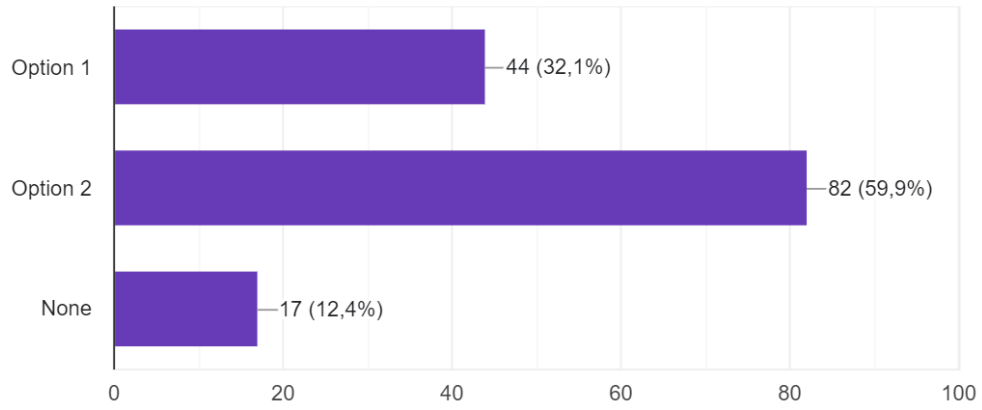

Figure 4.21: Question 2 - Percentages.

In Figure 4.22 we can see the percentages of the scores, which were once again assigned in the same manner as before, and we notice again that while 55% of the participants scored 2, another 40% scored 0.

	N	*Percent*	*Confidence Interval*
Score 0	55	40.15%	(31.94, 48.35)
Score 1	6	4.38%	(0.95, 7.80)
Score 2	76	55.47%	(47.15, 63.80)

Figure 4.22: Question 2 - Scores.

And finally, in the third question, which was expressed in the same way, both Option 1 and Option 2 were AI-generated. In Figure 4.23 we see that the participants' responses were nearly evenly split among the two options.

This time, to use the Likert scale, we are going to say that a participant scored 0 if the "None" option was chosen, scored 1 if only one of the two correct options were chosen, and scored 2 if both of the right options were chosen. In Figure 4.24 we can see that only 10% of the participants scored 2, while the 82% scored 1.

Finally, we calculated the total score for each participant by summing the scores of each of their answers. We can see how they did in Figure 4.25.

We notice that a very small percentage managed to score 6, about 31% scored 5, and both scores 1 and 3 were obtained by 23% of the participants.

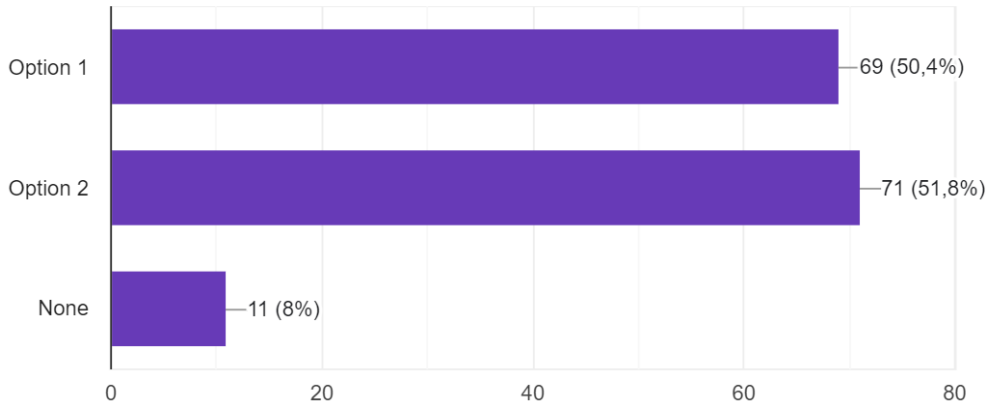

Figure 4.23: Question 3 - Percentages.

	N	*Percent*	*Confidence Interval*
Score 0	11	8.03%	(3.48, 12.58)
Score 1	112	81.75%	(75.28, 88.22)
Score 2	14	10.22%	(5.15, 15.29)

Figure 4.24: Question 3 - Scores.

	N	*Percent*	*Confidence Interval*
Score 0	1	0.73%	(0, 2.16)
Score 1	32	23.36%	(16.27, 30.44)
Score 2	11	8.03%	(3.48, 12.58)
Score 3	32	23.36%	(16.27, 30.44)
Score 4	13	9.49%	(4.58, 14.40)
Score 5	43	31.39%	(23.62, 39.16)
Score 6	5	3.65%	(0.51, 6.79)

Figure 4.25: Total scores

Overall, the study results align with the goal of creating AI-generated music that is challenging to distinguish from real songs.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this work, to make our dataset, we chose DJ sets of electronic music, cut them into small segments and made their corresponding spectrograms-images. Using this dataset, two main models have been trained, each of them with many variations in their hyperparameters. Although they all do suffer from partial mode collapse, as this is a very common phenomenon in GANs, they all can successfully generate images that look like the ones from the original dataset. We then proceeded to reverse the images back to sound which was the original goal of this work.

We evaluated our models by using people's opinions to see if they can distinguish between human made and AI-generated music.

5.2 Future work

Training GANs can be very time consuming based on the instability that characterizes them. Besides the fact that our models did generate new images, for future work, and interesting path would be to look further into what Denoising Diffusion Probabilistic Models (DDPMs) have to offer.

Denoising Diffusion Probabilistic Models can be trained with far less data, for fewer epochs, therefore using less computing power. Although it was not in the subject of this project, we did train a DDPM to see how it would perform with our dataset.

The first thing to notice is that when using our whole dataset, the training process takes too much time. Besides, based on relevant literature we can train our model with only 1000 images and trust that we will get realistic looking results. We will not get into the details of this model, but we would like to share a result from

training a DDPM for 50 epochs with only 1000 images, evenly derived from our dataset, in Figure 5.1.

Figure 5.1: Fake images generated by DDPM

While we don't have the corresponding sound, we can see that these results appear highly promising. So concerning future work, exploring this way seems worthwhile to find out the potential offerings of these models. This exploration may involve training for longer, train with different parts of the dataset, or even with a new dataset in which the spectrograms are created differently so they are bigger thus more detailed.

References

[1] Luke Metz, Ben Poole, David Pfau, and Jascha Sohl-Dickstein. Unrolled generative adversarial networks. *arXiv preprint arXiv:1611.02163*, 2016.

[2] Nao Tokui. Towards democratizing music production with ai-design of variational autoencoder-based rhythm generator as a daw plugin. *arXiv preprint arXiv:2004.01525*, 2020.

[3] Allen Huang and Raymond Wu. Deep learning for music. *arXiv preprint arXiv:1606.04930*, 2016.

[4] Sanidhya Mangal, Rahul Modak, and Poorva Joshi. Lstm based music generation system. *arXiv preprint arXiv:1908.01080*, 2019.

[5] Prafulla Dhariwal, Heewoo Jun, Christine Payne, Jong Wook Kim, Alec Radford, and Ilya Sutskever. Jukebox: A generative model for music. *arXiv preprint arXiv:2005.00341*, 2020.

[6] Andrea Agostinelli, Timo I Denk, Zalán Borsos, Jesse Engel, Mauro Verzetti, Antoine Caillon, Qingqing Huang, Aren Jansen, Adam Roberts, Marco Tagliasacchi, et al. Musiclm: Generating music from text. *arXiv preprint arXiv:2301.11325*, 2023.

[7] Hooman Rafraf. *Differential music: Automated music generation using LSTM networks with representation based on melodic and harmonic intervals*. PhD thesis, University of Florida, 2022.

[8] Carina Geerlings and Albert Merono-Penuela. Interacting with gpt-2 to generate controlled and believable musical sequences in abc notation. In *Proceedings of the 1st Workshop on NLP for Music and Audio (NLP4MusA)*, pages 49–53, 2020.

[9] Jiyou Chen, Gaobo Yang, Huihuang Zhao, and Manimaran Ramasamy. Audio style transfer using shallow convolutional networks and random filters. *Multimedia Tools and Applications*, 79:15043–15057, 2020.

[10] Maryam Majidi and Rahil Mahdian Toroghi. A combination of multi-objective genetic algorithm and deep learning for music harmony generation. *Multimedia Tools and Applications*, 82(2):2419–2435, 2023.

[11] Chris Donahue, Julian McAuley, and Miller Puckette. Adversarial audio synthesis. *arXiv preprint arXiv:1802.04208*, 2018.

[12] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[13] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[14] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

[15] Abdul Malik Badshah, Jamil Ahmad, Nasir Rahim, and Sung Wook Baik. Speech emotion recognition from spectrograms with deep convolutional neural network. In *2017 International Conference on Platform Technology and Service (PlatCon)*, pages 1–5, 2017.

[16] Yandre M. G. Costa, Luiz S. Oliveira, Alessandro L. Koericb, and Fabien Gouyon. Music genre recognition using spectrograms. In *2011 18th International Conference on Systems, Signals and Image Processing*, pages 1–4, 2011.

[17] Richard H Bolt, Franklin S Cooper, Edward E David Jr, Peter B Denes, James M Pickett, and Kenneth N Stevens. Speaker identification by speech spectrograms: some further observations. *The Journal of the Acoustical Society of America*, 54(2):531–534, 1973.

[18] Michael Towsey, Elizabeth Znidersic, Julie Broken-Brow, Karlina Indraswari, David M Watson, Yvonne Phillips, Anthony Truskinger, and Paul Roe. Long-duration, false-colour spectrograms for detecting species in large audio datasets. *Journal of Ecoacoustics*, 2:1–13, 2018.

[19] FFT and Spectrogram, Princeton archive.

[20] Emiel Por, Maaike van Kooten, and Vanja Sarkovic. Nyquist–shannon sampling theorem. *Leiden University*, 1(1), 2019.

[21] Wei Fang, Feihong Zhang, Victor S Sheng, and Yewen Ding. A method for improving cnn-based image recognition using dcgan. *Computers, Materials & Continua*, 57(1), 2018.

[22] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

[23] Qiufeng Wu, Yiping Chen, and Jun Meng. Dcgan-based data augmentation for tomato leaf disease identification. *IEEE Access*, 8:98716–98728, 2020.

[24] Patricia L. Suarez, Angel D. Sappa, and Boris X. Vintimilla. Infrared image colorization based on a triplet dcgan architecture. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, July 2017.

[25] Hoang Thanh-Tung and Truyen Tran. Catastrophic forgetting and mode collapse in gans. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–10, 2020.

[26] Martin Arjovsky and Léon Bottou. Towards principled methods for training generative adversarial networks. *arXiv preprint arXiv:1701.04862*, 2017.

[27] Youssef Kossale, Mohammed Airaj, and Aziz Darouichi. Mode collapse in generative adversarial networks: An overview. In *2022 8th International Conference on Optimization and Applications (ICOA)*, pages 1–6, 2022.

[28] Ian Goodfellow. Nips 2016 tutorial: Generative adversarial networks. *arXiv preprint arXiv:1701.00160*, 2016.

[29] Ke Li and Jitendra Malik. On the implicit assumptions of gans. *arXiv preprint arXiv:1811.12402*, 2018.