GENERATING MUSICAL NOTES WITH GANS

VIGNESH ROACHTHAVILIT

**A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
BACHELOR OF SCIENCE (APPLIED MATHEMATICS)
MAHIDOL UNIVERSITY
INTERNATIONAL COLLEGE
2025**

Thesis
entitled

GENERATING MUSICAL NOTES WITH GANS

was submitted to the Science Division, Mahidol University
for the degree of Bachelor of Science (Applied Mathematics)
on
July 21,2025

.....
Mr. Vignesh Roachthavilit
Candidate

.....
Assoc. Prof. Dr. Chatchawan Panraksa,
Ph.D. (Mathematics)
Major advisor

.....
Asst. Prof. Dr. Tumnoon
Charaslertrangsi
Division Chair
Science Division
Mahidol University

.....
Assoc. Prof. Dr. Aram
Tangboonduangjit,
Ph.D. (Mathematics)
Program Director
Bachelor of Science Programme
in Applied Mathematics
Mahidol University

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to all those who supported me throughout this research journey and the final thesis.

Above all, I am especially grateful to my advisor, Assoc. Prof. Chatchawan Panraksa, whose dedicated mentorship, insightful guidance, and unwavering support were fundamental to this project. His expert advice and thoughtful feedback at every stage helped shape the direction of my research and pushed me to maintain high standards.

Vignesh Roachthavilit.

GENERATING MUSICAL NOTES WITH GANS

VIGNESH ROACHTHAVILIT 6480515

B.Sc. (APPLIED MATHEMATICS)

THESIS ADVISORY COMMITTEE: CHATCHAWAN PANRAKSA, Ph.D.

ABSTRACT

Generative models have gained significant attention in recent years, particularly in the domains of large language models and image synthesis. However, audio generation, especially musical note synthesis, remains a relatively less explored area. This research focuses on generating musical notes given a set of features such as pitch using Generative Adversarial Networks (GANs). Although the project succeeded in building and training a GAN-based framework, it did not fully achieve the goal of producing clear, audible sound outputs. However, the work contributes to understanding the challenges and potential of GANs in the context of audio generation.

KEY WORDS : GANS / AUDIO
DEEP LEARNING

36 pages

CONTENTS

CONTENTS (cont.)

LIST OF TABLES

LIST OF FIGURES

CHAPTER I
INTRODUCTION

Generative models have become increasingly popular in recent years, allowing the creation of realistic media such as text, images, and even audio. While most of the attention has been on text generation and image synthesis, audio generation—especially musical notes—still feels like a less explored area.

Considering how big the music industry is, and how much potential audio synthesis has, this kind of technology could really help with creating new melodies, musical notes, and other types of sound. It also makes it easier for anyone to experiment and come up with their own unique sounds or songs, even without professional tools or help.

Generative Adversarial Networks (GANs) are one of the most well-known models used for generating images, and have been widely studied since they were introduced in 2014 by Goodfellow et al. [3]. They're known for being quite tricky to train, but when trained properly, they can produce extremely high-resolution results.

This project takes that same idea of using GANs for generation, but applies it to sound instead of images. Since raw audio waveforms are one-dimensional (1D) signals, they can't be directly used as input for GANs, which are typically designed to work with two-dimensional (2D) image data. To bridge this gap, the audio is first converted into a spectrogram which is a 2D visual representation of sound that shows frequency over time with intensity as color or brightness. By transforming each musical note into a spectrogram, we can represent sound in a format that GANs can understand and process, making it possible to generate musical notes using the same techniques used in image generation.

The focus of this research is on something simple but fundamental: generating a single musical note, this is the smallest building block of a piece of music. The goal is to explore how well GANs can learn to create individual notes under specific conditions, and to evaluate how effective and realistic the generated outputs are.

CHAPTER II
LITERATURE REVIEW

2.1 Generative Adversarial Networks (GANs)

Generative Adversarial Networks, a class of deep generative models , introduced by Goodfellow et al. [3], were revolutionary at the time. Although deep learning had existed for decades, most models then struggled with the complexity of generative tasks. This is because generative models need to learn the entire data distribution and produce new examples from it, which is much harder than simply predicting labels from inputs, as in discriminative tasks. Another challenge GANs address is the need for variation in generated outputs. We don't want the model to produce the same thing every time. To solve this, we feed the model random noise as input, allowing it to generate diverse and realistic results.

2.1.1 GANs Architecture

At its core, a GAN consists of two models:

- **Generator** (G): Takes a random noise vector $\mathbf{z} \sim p(\mathbf{z})$ and generates a sample, typically an image. This can be written as $G(\mathbf{z}) = \hat{\mathbf{x}}$, where $\hat{\mathbf{x}}$ is a synthetic image.
- **Discriminator** (D): Takes an input sample $\mathbf{x}$ (either real or generated) and outputs a scalar value representing the probability that $\mathbf{x}$ is real. Formally, $D(\mathbf{x}) \in [0, 1]$.

The GAN framework is inspired by concepts from game theory, where the generator and discriminator are in a two-player minimax game: the generator tries to fool the discriminator, while the discriminator tries to correctly identify real versus fake data.

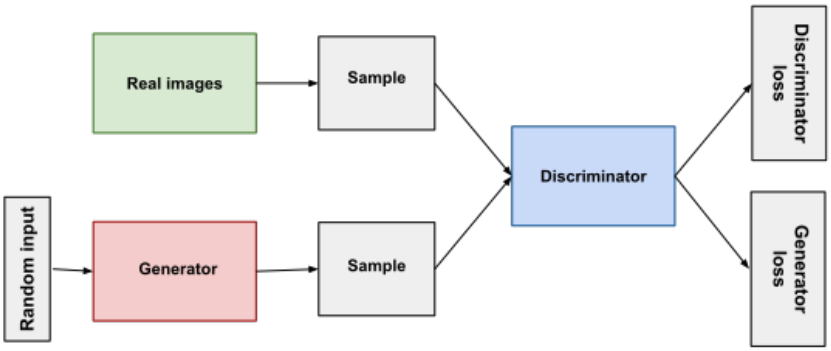


Figure 2.1: Illustration of the GAN architecture.

Source: Google [4]

2.1.2 Loss function

Since GANs is based on a minimax game framework, it is no surprise that it's loss function also follows a similar minimax formulation.

$$\min_G \max_D V(D,G) = \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))] \tag{2.1}$$

Here, the discriminator D aims to assign high values to real data $(D(\mathbf{x}) \to 1)$ and low values to generated data $(D(G(\mathbf{z})) \to 0)$. The generator, on the other hand, tries to "trick" the discriminator by producing samples that lead it to predict high values, i.e., $(D(G(\mathbf{z})) \to 1)$, making the generated data appear real.

2.1.3 Evaluating effectiveness of GANs

When trained properly, GANs can be incredibly powerful and remains highly effective even in 2025. However, achieving stable and reliable training requires careful tuning of hyperparameters, architectural choices, and loss functions.

As noted by Arjovsky and Bottou [5] and Salimans et al. [6], training GANs effectively is challenging due to issues such as instability, mode collapse, and sensitivity to hyperparameters.

Despite these challenges, GANs have proven to be one of the most effective frameworks for generating high-quality and realistic data across various domains.

Their ability to model complex data distributions without explicitly defining a likelihood function sets them apart from traditional generative models. GANs are particularly well-suited for tasks such as image synthesis, data augmentation, and audio generation, where they have achieved state-of-the-art results in both visual fidelity and diversity of outputs. For instance, Progressive Growing GANs (PGGAN) demonstrated the ability to generate highly detailed and stable high-resolution images by gradually increasing model complexity during training [1], while GANSynth successfully applied GANs to audio synthesis, generating high-fidelity musical notes [7].

2.2 GANSynth

Engel et al. [7] introduced GANSynth, a GAN-based model for audio synthesis that generates high-fidelity musical notes by modeling log-magnitude mel spectrograms conditioned on pitch. The network is trained on spectrogram "images" and its outputs are later inverted into waveforms. Although treating audio as images may seem unintuitive, this approach is effective because it addresses two challenges in audio generation:

- **Parallel generation:** GANsynth can produce an entire sound in parallel, speeding up the synthesis process(very fast generation especially on a GPU). It is not auto-regressive like audio generation models like WaveNet van den Oord et al. [8]
- **Enhanced Training Stability:** Operating on lower-resolution spectrogram representations reduces the complexity of the learning task. Raw audio waveforms have extremely high temporal resolution and fidelity, making them difficult to model directly. By working with spectrograms, GANSynth avoids the instability and high computational cost associated with learning raw waveforms

2.2.1 Spectrograms

Spectrograms are visual representations of the frequency content of a signal over time, and they are the backbone for the GANSynth model. A spectrogram is

typically computed using the Short-Time Fourier Transform (STFT), which slices the input waveform into overlapping time windows and calculates the frequency components within each segment. The resulting 2D image has time on the horizontal axis, frequency on the vertical axis, and color or intensity representing the magnitude of each frequency component.

The formula is given by:

$$X(k,m) = \sum_{n=-\infty}^{\infty} x[n] \cdot w[n-m] \cdot e^{-j2\pi kn/N} \tag{2.2}$$

Where:

- n is the index over the input time-domain signal.
- m is the time frame index (position of the sliding window).
- k is the frequency bin index.

The output $X(k,m)$ is a complex number, written in the form $a + bi$,

GANSynth uses two types of spectrograms: **log-magnitude mel spectrograms** and **instantaneous frequency spectrograms**. The log-mel spectrogram shows how strong each frequency is over time, while the instantaneous frequency shows how the phase (or timing) of each frequency changes.

2.2.1.1 Log-magnitude Mel Spectrograms

$$A(k,m) = |X(k,m)| = \sqrt{\mathrm{Re}(X(k,m))^2 + \mathrm{Im}(X(k,m))^2} \tag{2.3}$$

The steps to compute a log-magnitude mel spectrogram are as follows:

1. **Extract magnitude from the STFT:**

$$A(k,m) = |X(k,m)| \tag{2.4}$$

2. **Apply mel filterbank to convert linear frequencies to mel scale:**

$$S(p,m) = \sum_{k=0}^{K-1} M_{p,k} \cdot A^{\gamma}(k,m) \tag{2.5}$$

where $M_{p,k}$ is the mel filterbank matrix and γ is a compression factor (often $\gamma = 1, 2$).

The mel filterbank matrix $M_{p,k}$ is a set of triangular filters that map linear frequency bins k to mel-scale bins p. This transformation reflects how human hearing is more sensitive to linear differences in lower frequencies than in higher ones. Each row p in the matrix defines a triangular filter for specific range of frequencies.

The mel scale f_{mel} corresponding to a linear frequency f in Hz is calculated using the following formula:

$$f_{\text{mel}} = 2595 \cdot \log_{10}\left(1 + \frac{f}{700}\right) \tag{2.6}$$

Since audio is ultimately perceived by humans, it is important to use a frequency scale that aligns with human auditory perception, which is why the melscale is used.

3. **Apply log transformation to compress dynamic range:**

$$S_{\log}(p, m) = \log(S(p, m) + \epsilon) \tag{2.7}$$

where ϵ is an arbitary small constant to avoid taking the log of zero.

Likewise, human perception of loudness is approximately logarithmic with respect to signal power. This is why a logarithmic transformation is applied to the mel spectrogram. Making it a log-magitude mel spectrogram

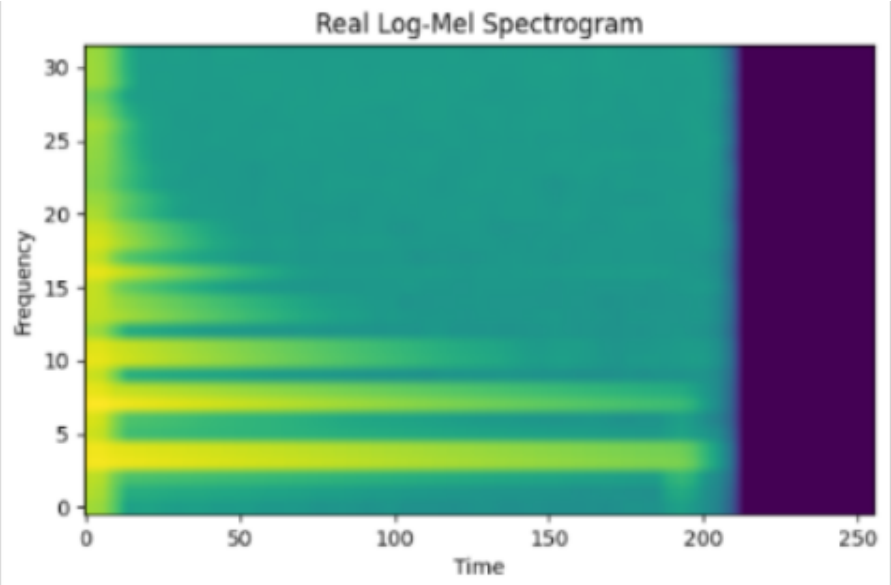


Figure 2.2: Example of a log-magnitude mel spectrogram. The y-axis represents frequency in mel bins, while the x-axis represents time. The color intensity indicates the log-scaled magnitude of each frequency component for a given time window.

2.2.1.2 Instantaneous Frequency Spectrograms

1. **Calculate raw phase from the STFT:**

$$\phi_{\text{raw}}(k,m) = \text{atan2}\left(\Im\{X(k,m)\}, \Re\{X(k,m)\}\right), \quad \phi_{\text{raw}} \in (-\pi, \pi] \tag{2.8}$$

2. **Unwrap the phase across time frames:**

$$\varphi(k,m) = \text{unwrap}\left(\angle X(k,m)\right) \tag{2.9}$$

This step ensures that the phase values evolve smoothly over time by removing artificial jumps that occur due to phase wrapping within the interval $(-\pi, \pi]$. In practice, this is done by checking whether the difference between consecutive phase values exceeds π. If so, 2π is added or subtracted to maintain continuity. The unwrapped phase is then computed using a cumulative sum:

$$\Delta\phi(k,m) = \phi_{\text{raw}}(k,m) - \phi_{\text{raw}}(k,m-1) \tag{2.10}$$

$$\Delta\phi(k,m) = \begin{cases} \Delta\phi(k,m) - 2\pi, & \text{if } \Delta\phi(k,m) > \pi \\ \Delta\phi(k,m) + 2\pi, & \text{if } \Delta\phi(k,m) < -\pi \\ \Delta\phi(k,m), & \text{otherwise} \end{cases} \tag{2.11}$$

$$\varphi(k,m) = \phi_{\text{raw}}(k,0) + \sum_{i=1}^{m} \Delta\phi(k,i) \tag{2.12}$$

where $\Delta\phi(k,m)$ is the corrected phase difference between adjacent time frames.

3. **Compute the instantaneous frequency:**

$$f_{\text{inst}}(k,m) = \frac{sr}{2\pi h}\left[\varphi(k,m) - \varphi(k,m-1)\right] \tag{2.13}$$

Here, sr is the sample rate of the original audio signal (measured in Hz), which defines how many samples per second are in the waveform. h is the hop length—the number of samples between adjacent STFT frames. These constant are multiplied to ensure the phase difference values over time are in Hz.

4. **Apply the mel filterbank:**

$$\tilde{F}(p,m) = \sum_{k=0}^{K-1} M_{p,k} \cdot f_{\text{inst}}(k,m) \tag{2.14}$$

where $M_{p,k}$ is the same as defined in equation (2.5)

2.2.2 Spectrogram Inversion to Audio

After generating log-mel and instantaneous frequency spectrograms using GANSynth, the final step is to convert these representations back into a time-domain audio waveform.

2.2.2.1 Log-Mel Spectrogram Inversion:

1. **De-log:** The logarithmic transformation applied earlier is reversed using the exponential function:

$$S(p, m) = \exp(S_{\log}(p, m)) - \epsilon \tag{2.15}$$

   This restores the mel-scaled magnitude spectrogram.

2. **Invert the Mel Filterbank:** The mel spectrogram $S(p, m)$ is mapped back to linear frequency using a pseudoinverse or the mel filterbank:

$$A(k, m) \approx M^{-1} S(p, m) \tag{2.16}$$

   where M^{-1} is the inverse (approximation) of the mel matrix.

3. **Get Magnitude:** This step recovers the full STFT magnitude $|X(k, m)| = A(k, m)$, which will be used in combination with phase to reconstruct the complex STFT.

2.2.2.2 Instantaneous Frequency Spectrogram Inversion:

1. **Invert the Mel Filterbank:** The mel-scaled instantaneous frequency is projected back to linear frequency bins using:

$$f_{\text{inst}}(k, m) \approx M^{-1} \tilde{F}(p, m) \tag{2.17}$$

2. **Sum to Recover Phase:** The instantaneous frequency is summed across time to recover the unwrapped phase (cumulative sum):

$$\varphi(k, m) = \sum_{i=1}^{m} \frac{2\pi h}{sr} f_{\text{inst}}(k, i) \tag{2.18}$$

3. **Wrap Phase into $(-\pi, \pi]$:** The phase is wrapped into the valid range for complex exponential reconstruction:

$$\varphi(k, m) = \text{mod}(\varphi(k, m) + \pi, 2\pi) - \pi \tag{2.19}$$

4. **Get Phase:** Use the wrapped phase $\varphi(k, m)$ to form the complex STFT:

$$X(k, m) = A(k, m) \cdot e^{j\varphi(k,m)} \tag{2.20}$$

Finally, the inverse STFT (iSTFT) is applied to $X(k, m)$ to reconstruct the waveform in the time domain.

2.2.3 STFT and iSTFT: Imperfect Reconstruction

The final audio waveform is reconstructed using the inverse Short-Time Fourier Transform (iSTFT). While STFT+iSTFT aims to recover the original signal, the process is not perfectly lossless. Two main reasons contribute to this:

1. **Discrete Frequency Bins:** STFT represents frequency content in discrete bins, which introduces quantization and limits the resolution of fine frequency changes.
2. **Windowed Time Segments:** STFT analyzes small overlapping windows of the signal, so it only looks at short segments of time instead of the entire waveform at once.
3. **Mel Filterbank Inversion** Before applying iSTFT, both log-mel and instantaneous frequency spectrograms must be projected back from mel scale to linear frequency. This is not a true inverse, but an approximation.

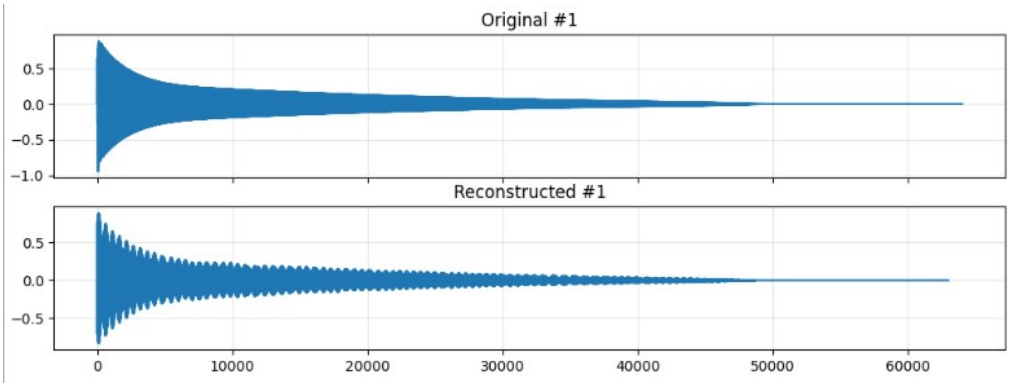


Figure 2.3: Comparison of original and reconstructed waveform using STFT and iSTFT. Although similar in shape, the reconstruction shows high-frequency artifacts and slight amplitude deviations.

However, despite these slight deviations, the overall waveform shape remains highly similar. In practice, when listening to both the reconstructed and original audio, most listeners are unable to perceive any noticeable difference. Therefore, the minor reconstruction errors do not outweigh the significant benefits offered by GANSynth especially from it's fast inference(generation speed).

2.3 Progressive Growing of GANs

Traditional GANs often struggle to generate high-resolution images due to instability during training and difficulty learning complex structures all at once. To address this, Karras et al. [1] introduced a training strategy called *Progressive Growing of GANs* (PGGAN).

The key idea behind PGGAN is to start training both the generator and discriminator at a very low resolution (e.g., 4×4) and progressively add layers that increase the resolution over time (e.g., 8×8, 16×16, ..., up to 1024×1024). Each new layer is faded in smoothly to avoid sudden jumps in model complexity. This approach allows the model to first learn the global structure first and then refine finer details as training progresses.

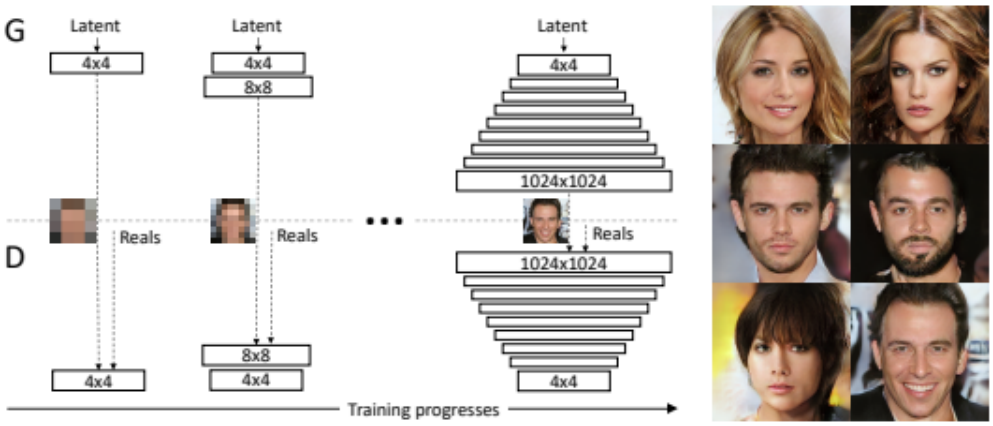


Figure 2.4: Progressive Growing of GANs: the generator and discriminator begin training at low resolution (e.g., 4×4) and grow incrementally to high resolution (e.g., 1024×1024).Image from Karras et al. [1].

CHAPTER III
METHODOLOGY

Building on the spectrogram-based generation approach introduced in GAN-Synth [7], as well using the NSynth dataset by [2]This section outlines the key components of the method, including data preparation, model design, loss functions, and audio reconstruction.

3.1 Nsynth dataset

NSynth [2] is a diverse dataset containing over 305,000 musical notes, each exactly 4 seconds in length and recorded with high studio quality. Each audio fades out at the end of the 4 seconds.

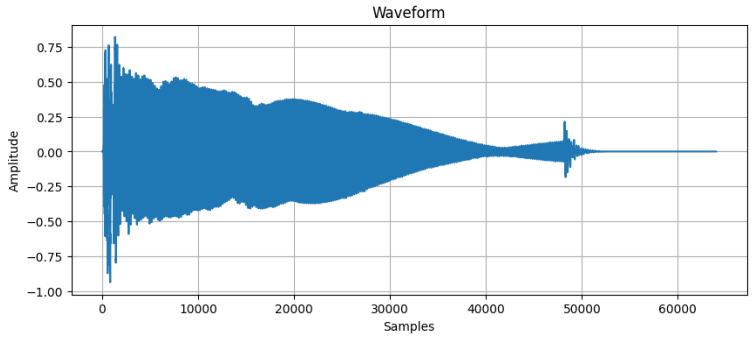


Figure 3.1: Waveform of a 4-second audio note from the NSynth dataset. The signal shows a strong onset followed by a fade-out at the end

Feature	Type	Description
note	`int64`	A unique integer identifier for the note.
note_str	`bytes`	A unique string identifier for the note in the format `<instrument_str>-<pitch>-<velocity>` .
instrument	`int64`	A unique, sequential identifier for the instrument the note was synthesized from.
instrument_str	`bytes`	A unique string identifier for the instrument this note was synthesized from in the format `<instrument_family_str>-<instrument_production_str>-<instrument_name>` .
pitch	`int64`	The 0-based MIDI pitch in the range [0, 127].
velocity	`int64`	The 0-based MIDI velocity in the range [0, 127].
sample_rate	`int64`	The samples per second for the `audio` feature.
audio*	`[float]`	A list of audio samples represented as floating point values in the range [-1,1].
qualities	`[int64]`	A binary vector representing which sonic qualities are present in this note.
qualities_str	`[bytes]`	A list IDs of which qualities are present in this note selected from the sonic qualities list.
instrument_family	`int64`	The index of the instrument family this instrument is a member of.
instrument_family_str	`bytes`	The ID of the instrument family this instrument is a member of.
instrument_source	`int64`	The index of the sonic source for this instrument.
instrument_source_str	`bytes`	The ID of the sonic source for this instrument.

Figure 3.2: Overview of features in the NSynth dataset[2] Each note contains metadata such as pitch, velocity, instrument family and others.

Since the majority of the NSynth dataset falls within the MIDI pitch range of 24 to 84 (approximately 32 Hz to 1000 Hz), we apply a filter to retain only notes within this frequency range. For each selected sample, we extract the following features: `pitch`, `velocity`, `instrument_family`, and `instrument_source`.

We use `instrument_family` instead of the full `instrument` label because the dataset contains nearly 1000 distinct instruments. This would dilute the class distribution, making it difficult for the model to learn patterns for each instrument.

The model and its helper functions are implemented using the Python library *PyTorch*, a popular framework designed for deep learning and tensor-based computation.

3.2 Creating spectrograms

This section outlines both the methodology and implementation details for generating spectrograms from raw audio using the *PyTorch* framework.

3.2.1 Instantaneous Frequency Utilities

This implementation below follows equations (2.8) to (2.14)

Algorithm 1 Helper functions for instantaneous frequency calculation

```
function DIFF(phases)
    return finite difference along time axis
end function
function UNWRAP(ph)
    diffs = torch.diff(ph)                          ▷ Compute phase differences
    mods = (diffs + π) % (2π) − π                   ▷ Wrap to (−π, π]
    mods[(mods == −π) & (diffs > 0)] = π            ▷ Fix edge case at −π
    corr = mods − diffs                             ▷ Correction term
    csum = torch.cumsum(corr)                       ▷ Cumulative correction
    return ph + csum
end function
function INST_FREQ(ph)
    uf = unwrap(ph)                                 ▷ Unwrap phase across time
    diffs = torch.diff(uf)                          ▷ Get time derivatives
    return (sr / (2 * math.pi * hop)) * torch.cat([0, diffs])   ▷ Convert to Hz
end function
function NORM(x, mean, std)
    return (x − mean) / std
end function
```

3.2.2 Creating Spectrograms from Audio

In addition to generating log-mel and instantaneous frequency (mel_if) spectrograms, we apply a **Hann window** to each overlapping frame of the waveform before computing the STFT. The Hann window is a commonly used function that reduces spectral leakage by smoothly reducing the signal amplitude at the edges of each frame.

The Hann window of length N is defined as:

$$w[n] = 0.5\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right), \quad 0 \leq n < N \tag{3.1}$$

Applying this window helps to minimize discontinuities at frame boundaries to help ensure smoother transisitions.

Furthermore, to ensure consistent spectrogram dimensions across all samples for training, we apply left-padding only.

Algorithm 2 Waveform to Log-Mel and Instantaneous Frequency Spectrogram Conversion

function WAV_TO_SPEC(wav)
    `x = F.pad(wav, (pad_left, 0))` ▷ Left-pad waveform for STFT alignment
    `window = torch.hann_window(n_fft)` ▷ Create Hann window
    `stft = torch.stft(x, n_fft, hop, window, return_complex=True)` ▷ Compute complex STFT
    `mag = stft.abs()` ▷ Extract magnitude
    `ph = torch.angle(stft)` ▷ Extract phase
    `mel_fb = T.MelScale(...)` ▷ Create mel filterbank
    `logmel = normalize(torch.log(mel_fb(mag) + ϵ))` ▷ Log-compress mel magnitude and normalize
    `mel_ph = mel_fb(ph)` ▷ Apply mel filterbank to phase
    `mel_if = inst_freq(mel_ph, axis=time)` ▷ Compute instantaneous frequency from mel-filtered phase
    `mel_if = normalize(mel_if)` ▷ Normalize IF values
    return `logmel, mel_if, pad_left`
end function

Below are two spectrograms created:

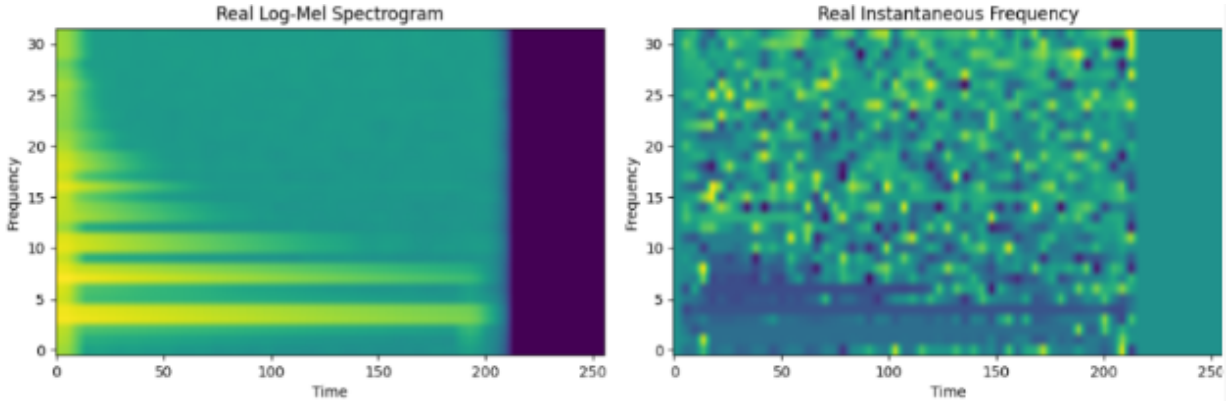


Figure 3.3: Spectrogram created from an acoustic guitar audio sample at pitch 32 (MIDI)

3.2.3 Reverting Spectrograms to Audio

To get the raw waveform from the two spectrograms we can follow the algorithm below:

Algorithm 3 Spectrogram to Waveform Reconstruction (PyTorch-style pseudocode)

```
function SPEC_TO_WAV(logmel, mel_if)
    mel_mag = exp(denormalize(logmel))          ▷ Undo log and normalization
    mel_freq = denormalize(mel_if)              ▷ Undo IF normalization
    mag = invert_mel_filterbank(mel_mag)        ▷ Project to linear frequency
    freq = invert_mel_filterbank(mel_freq)      ▷ Project IF to linear frequency bins
    phase = cumulative_sum(freq)                ▷ Reconstruct phase from IF
    phase = wrap_to_pi(phase)                   ▷ Wrap phase to (−π, π]
    stft = mag · exp(j· phase)                  ▷ Reconstruct complex STFT
    wav = iSTFT(stft)                           ▷ Apply inverse STFT
    return wav
end function
```

Now that we have the necessary function to create spectrograms we can now start building the model.

3.3 Building the model

In this section, I will discuss the architecture and implementation of the model inspired by Progressive Growing of GANs (PGGANs). This approach incrementally increases the resolution of generated outputs during training, allowing the generator and discriminator to stabilize at lower resolutions before learning finer details. While several models were experimented with throughout the project, this architecture demonstrated the most promising results. We use a convolutional neural network (CNN) based approach for both the generator and discriminator, as it is standard layers for capturing spatial patterns in 2D data such as spectrograms.

3.3.1 Growing the model

As our model progressively grows, we must define a function to control its growth dynamically during training. To implement this strategy, we define a function that maps a continuous growth value $g \in [0, 1]$ to a corresponding network depth.
This algorithm ensures that:

- When $g = 0$, the model is at its shallowest - lowest resolution
- When $g = 1$, the model reaches its full depth - highest resolution
- For values in between, it uses the formula below (3.2)

The growth depth is calculated using the following equation:

$$\texttt{growing\_depth} = \log_2 \left(1 + \left(2^{(\texttt{max\_depth}+1)} - 1\right) \cdot g\right) \tag{3.2}$$

This formula ensures that the number of layers (or resolution levels) increases exponentially as g increases. In PGGANs, each step doubles the spatial resolution (e.g., from 8×8 to 16×16, then to 32×32, and so on). The exponential term models this doubling behavior, while the logarithm reverses it to give an interpretable depth measure
The pseudocode for this growth-setting mechanism is shown below.

Algorithm 4 Set Growth Level for Progressive Network

```
function SET_GROWTH(net, g)
    g = max(0.0, min(1.0, g))                    ▷ Clamp g to [0, 1]
    net.growing_level = g
    if g == 0 then
        net.growing_depth = 0.0
    else
        net.growing_depth = log2(1 + ((2^(net.max_depth + 1)) - 1) * g)
    end if
end function
```

The spectrogram generator is designed to grow from a minimum resolution of $(\mathbf{8}, \mathbf{4})$ to a maximum resolution of $(\mathbf{512}, \mathbf{256})$ during training. As the spatial resolution increases, the number of feature channels decreases from $\mathbf{512}$ at the lowest resolution to $\mathbf{8}$ at the final output layer.

3.3.2 Generator architecture

The generator is designed to take an input vector of size $\mathbf{336}$, composed of $\mathbf{256}$ dimensions of random latent noise, $\mathbf{61}$ for pitch encoding, $\mathbf{11}$ for instrument family, $\mathbf{3}$ for instrument source, and $\mathbf{5}$ for velocity. It outputs a 2D image of size $H \times W$, where the spatial dimensions H and W are determined by the current `growing_depth`, as defined in Equation (3.2).

3.3.2.1 Pixel-wise Feature Vector Normalization (PixelNorm)

One of the key utility functions used in the generator is *PixelNorm*, proposed in the original PGGAN paper [1]. PixelNorm operates on each feature vector at a given spatial location and normalizes it across channels. This helps prevent the signal from growing too large or collapsing during training, as it ensures

The normalization is applied independently at every spatial location by dividing the vector by its root mean square (RMS) value. The formula is given by:

$$\text{PixelNorm}(x) = \frac{x}{\sqrt{\frac{1}{C}\sum_{c=1}^{C} x_c^2 + \epsilon}} \tag{3.3}$$

Here, C is the number of channels, x_c is the activation in channel c at a given spatial location, and ϵ is a small constant added to help numerical stability

Algorithm 5 Utility Functions Used in the Generator

function PIXEL_NORM(x, $\epsilon = 10^{-8}$)
    return $x \cdot \texttt{rsqrt}(\texttt{mean}(x^2, \texttt{dim} = 1, \texttt{keepdim} = True) + \epsilon)$ ▷ Channel-wise normalization at each spatial location
end function
function UPSAMPLE(x, factor = 2)
    return $\texttt{interpolate}(x, \texttt{scale\_factor} = \texttt{factor}, \texttt{mode} = \texttt{'nearest'})$ ▷ Nearest-neighbor upsampling
end function
function DOWNSAMPLE(x, factor = 2)
    return $\texttt{interpolate}(x, \texttt{scale\_factor} = 1/\texttt{factor}, \texttt{mode} = \texttt{'area'})$ ▷ Area-based downsampling
end function
function LERP(a, b, t)
    return $a + (b - a) \cdot t$ ▷ Linear interpolation between a and b
end function

Now that we have all the utility functions it's time for the design of the generator.

Algorithm 6 Progressively Growing Generator (PCGAN-style)

function GENERATOR.INIT(min_res, max_res, min_ch, max_ch, growing_level)
    Compute `max_depth` from `min_res` and `max_res`
    Compute `growing_depth` using Equation (3.2)
    Initialize fully connected layer: `FC` : $\mathbb{R}^{336} \rightarrow \mathbb{R}^{C \times H \times W}$
    for each depth level from 0 to `max_depth` **do**
        Set input/output channels based on depth (channels decrease as resolution increases)
        Create a convolutional block:
            - Two Conv2D layers
            - LeakyReLU activations
            - PixelNorm after each conv
        Append to `blocks` and corresponding `to_spec` output layers
    end for
end function
function GENERATOR.FORWARD(z, pitch, instrument_source)
    Concatenate inputs: $x = [z \,\|\, \texttt{pitch} \,\|\, \texttt{instrument\_source}]$
    Apply PixelNorm and FC projection to initial resolution
    Compute fade level: `fade_depth`, `alpha` from `growing_depth`
    for depth in 0 to `fade_depth` **do**
        if depth > 0 **then**
            Upsample feature map
        end if
        Pass through `block[depth]`
    end for
    if not fading or at max depth **then**
        return `tanh(to_spec[fade_depth](x))`
    else
        $x_{\text{new}} \leftarrow \texttt{block[fade\_depth+1]}(\texttt{upsample}(x))$
        $img_{\text{new}} \leftarrow \texttt{to\_spec[fade\_depth+1]}(x_{\text{new}})$
        $img_{\text{old}} \leftarrow \texttt{upsample}(\texttt{to\_spec[fade\_depth]}(x))$
        Blend: $img = \texttt{lerp}(img_{\text{old}}, img_{\text{new}}, \alpha)$
        return $\texttt{tanh}(img)$
    end if
end function

The final `tanh` activation function is applied to ensure that each output value is normalized to the range $[-1, 1]$.

3.3.3 Discriminator architecture

The discriminator is designed to take an input tensor of shape $C \times H \times W$, where H and W depend on the current resolution determined by the growth factor. It

processes this input and produces multiple separate output vectors: one scalar value for real/fake classification, a vector of size $\mathbf{61}$ for pitch prediction, $\mathbf{11}$ for instrument family classification, $\mathbf{3}$ for instrument source classification, and $\mathbf{5}$ for velocity estimation.

3.3.3.1 Minibatch Standard Deviation (MinibatchStdDev)

To encourage variation in the generated outputs and reduce the risk of mode collapse, the discriminator incorporates a `MinibatchStdDev` layer. Mode collapse occurs when the generator learns to produce limited or nearly identical outputs regardless of the input noise, which reduces the diversity of outputs produced by the generator.

The `MinibatchStdDev` layer addresses this by computing the standard deviation across the batch for each spatial location. It then appends this scalar as an additional feature map to the input, allowing the discriminator to detect lack of variation across the batch.

$$\sigma = \frac{1}{CHW} \sum_{c=1}^{C} \sum_{h=1}^{H} \sum_{w=1}^{W} \sqrt{\frac{1}{B} \sum_{b=1}^{B} (x_{b,c,h,w} - \mu_{c,h,w})^2 + \epsilon} \tag{3.4}$$

Algorithm 7 Minibatch Standard Deviation Layer

function MINIBATCHSTDDEV(x)
  x: input tensor of shape $[B, C, H, W]$
  $m = \texttt{mean}(x, \texttt{dim} = 0, \texttt{keepdim} = True)$
  $v = \texttt{mean}((x - m)^2, \texttt{dim} = 0)$ ▷ Variance across batch
  $s = \sqrt{v + \epsilon}$ ▷ Standard deviation
  $\sigma = \texttt{mean}(s).view(1, 1, 1, 1)$ ▷ Scalar average
  $\texttt{std\_map} = \texttt{repeat}(\sigma, B, 1, H, W)$
  return $\texttt{concat}(x, \texttt{std\_map}, \texttt{dim} = 1)$ ▷ Shape: $[B, C + 1, H, W]$
end function

Algorithm 8 Progressively Growing Discriminator (PCGAN-style)

function DISCRIMINATOR(spec)
  $H \leftarrow$ `spec.shape[2]`
  `image_depth` $\leftarrow \log_2(H/$`min_resolution`$)$
  `fade_depth` $\leftarrow \lfloor$`growing_depth`$\rfloor$
  $\alpha \leftarrow$ `growing_depth` $-$ `fade_depth`
  `start_block` $\leftarrow$ `max_depth` $-$ `image_depth`
  if $\alpha == 0.0$ **or** `image_depth` $== 0$ **then**
    $x \leftarrow$ `from_spec[start_block]`$(spec)$
    $x \leftarrow$ `block[start_block]`(x)
  else
    $x_{\text{new}} \leftarrow$ `from_spec[start_block]`$(spec)$
    $x_{\text{new}} \leftarrow$ `block[start_block]`(x_{new})
    `spec_old` $\leftarrow$ `avg_pool2d`$(spec, 2)$
    $x_{\text{old}} \leftarrow$ `from_spec[start_block + 1]`$($`spec_old`$)$
    $x \leftarrow$ `lerp`$(x_{\text{old}}, x_{\text{new}}, \alpha)$
  end if
  for i from `start_block` $+ 1$ to `len(blocks)` **do**
    $x \leftarrow$ `blocks`$[i](x)$
  end for
  $x \leftarrow$ `flatten`(x)
  `real_score` $\leftarrow$ `final_dense`(x)
  `pitch_logits` $\leftarrow$ `final_pitch_cls`(x)
  `velocity_logits` $\leftarrow$ `final_velocity_cls`(x)
  `instrument_family_logits` $\leftarrow$ `final_instr_fam_cls`(x)
  `source_logits` $\leftarrow$ `final_src_cls`(x)
  return `real_score`, `pitch_logits`, `velocity_logits`, `instrument_family_log`
end function
function FORWARD(spec)
  return `discriminator(spec)`
end function

Now that we setup the models we can now move on to the training set up.

3.4 Training Setup

The model was trained using Google Colab with an NVIDIA A100 GPU. However, due to the large model size and the high-fidelity nature of the audio data (16 kHz waveforms), it is not feasible to load the entire dataset into memory at once. To address this, a custom dataset loader was implemented using PyTorch's `Dataset` and `DataLoader` classes to load and preprocess the data in mini-batches.

A batch size of 8 was selected as it represented the upper limit of what the GPU could accommodate without running out of memory.

3.4.1 Dataset and Dataloader Classes

Algorithm 9 NSynthDataset Class (Guitar-only Subset)

function NSYNTHDATASET.INIT(split="train", local_cache_dir="/tmp/")
    `root_dir` ← `local_cache_dir/split`
    Load metadata from `examples.json` in `root_dir`
    Map all available `.wav` files by filename
    Set constants:
        - Pitch range: $24 \leq$ `pitch` ≤ 84, total: 61
        - Velocity values: $[25, 50, 75, 100, 127]$, total: 5
        - Number of instrument families: 11
        - Number of instrument sources: 3
    Initialize `file_names` as empty
    for each item in metadata **do**
        if `.wav` file exists AND pitch is in range. **then**
            Keep sample
        else
            Skip sample
        end if
    end for
end function
function NSYNTHDATASET.__GETITEM__(index)
    Retrieve filename and load waveform using `torchaudio.load`
    Extract metadata: pitch, velocity, instrument_family, instrument_source
    One-hot encode:
        - `pitch` → size 61
        - `velocity` → size 5
        - `instrument_family` → size 11
        - `instrument_source` → size 3
    return dictionary: {`"audio", "pitch", "velocity", "instrument_family", "instrument_source"`}
end function

Algorithm 10 DataLoader Configuration for Training

```
train_loader ← DataLoader(
dataset ← train_dataset                          ▷ Custom spectrogram dataset
batch_size ← 8                                   ▷ Mini-batch size
shuffle ← True                                   ▷ Shuffle data for each epoch
num_workers ← 4                                  ▷ Number of subprocesses for loading data
pin_memory ← True                                ▷ Enable faster host-to-GPU transfers
prefetch_factor ← 2                              ▷ Each worker preloads 2 batches
persistent_workers ← True                        ▷ Keep workers alive across epochs
)
```

To ensure efficient data loading, `num_workers` is set to **4**, allowing multiple subprocesses to load batches in parallel. Combined with a `prefetch_factor` of **2**, each worker preloads two batches in advance—ensuring that the next batch is ready as soon as the current one finishes training, which helps minimize idle GPU time.

`pin_memory` is set to `True` to enable the use of Direct Memory Access (DMA), allowing the GPU to read data directly from page-locked (pinned) CPU memory. This significantly speeds up data transfer from RAM to GPU

3.4.2 Loss functions

Since the model is trained to predict not only whether an input is real or fake, but also to classify pitch, velocity, instrument family, and instrument source, we extend the original adversarial loss formulation (Equation 2.1) to incorporate auxiliary classification losses.

The updated discriminator loss combines the adversarial component, auxiliary classification losses (on real spectrograms only), and a gradient penalty term to stabilize training:

$$\mathcal{L}_D = \underbrace{\mathbb{E}[\texttt{softplus}(-D(x_{\text{real}}))] + \mathbb{E}[\texttt{softplus}(D(x_{\text{fake}}))]}_{\text{adversarial loss}} + \lambda_{\text{pitch}} \cdot \mathcal{L}_{\text{pitch}} + \lambda_{\text{vel}} \cdot \mathcal{L}_{\text{vel}} + \lambda_{\text{fam}} \cdot \mathcal{L}_{\text{fam}} + \lambda_{\text{src}} \cdot \mathcal{L}_{\text{src}} + \lambda_{\text{gp}} \cdot \mathcal{L}_{\text{GP}} \tag{3.5}$$

The classification losses ($\mathcal{L}_{\text{pitch}}, \mathcal{L}_{\text{vel}}, \mathcal{L}_{\text{fam}}, \mathcal{L}_{\text{src}}$) are computed using cross-entropy on the discriminator's predictions from real spectrograms.

The gradient penalty is applied to both real and fake inputs to penalize steep gradients in the discriminator.

The gradient penalty from Gulrajani et al. [9] is computed by taking the squared ℓ_2-norm of the discriminator's gradient with respect to its input. The penalty encourages gradients to remain small.

$$\mathcal{L}_{\text{GP}} = \mathbb{E}_x \left[\|\nabla_x D(x)\|_2^2 \right] \tag{3.6}$$

The generator is trained to both fool the discriminator and generate samples that match the correct pitch, velocity, instrument family, and source labels. The generator loss is defined as:

$$\mathcal{L}_G = \underbrace{\mathbb{E}[\texttt{softplus}(-D(x_{\text{fake}}))]}_{\text{adversarial loss}} + \lambda_{\text{pitch}} \cdot \mathcal{L}_{\text{pitch}} + \lambda_{\text{vel}} \cdot \mathcal{L}_{\text{vel}} + \lambda_{\text{fam}} \cdot \mathcal{L}_{\text{fam}} + \lambda_{\text{src}} \cdot \mathcal{L}_{\text{src}} \tag{3.7}$$

Here, the classification losses are computed on the discriminator's predictions for the fake samples, encouraging the generator to produce examples that resemble the correct class labels across all auxiliary tasks.

3.4.3 Training Loop

Algorithm 11 Progressive Training Loop with Auxiliary Classification

Initialize: G, D, `noise_dim`, `device`
Set optimizers:

- $g\_opt \leftarrow \texttt{Adam}(G.\texttt{parameters}(), \texttt{lr} = 8e^{-4}, \beta = (0.5, 0.999))$
- $d\_opt \leftarrow \texttt{Adam}(D.\texttt{parameters}(), \texttt{lr} = 5e^{-5}, \beta = (0.5, 0.999))$

Define loss function: $\texttt{ce\_loss} \leftarrow \texttt{CrossEntropyLoss}()$
$global\_step \leftarrow \texttt{load\_latest\_checkpoint}()$
for $epoch = 0$ **to** $epochs$ **do**
  for each $batch \in \texttt{train\_loader}$ **do**
    if $global\_step$ already trained **then**
      continue
    end if
    Compute $g = \min(\texttt{grow\_cap}, \frac{global\_step}{\texttt{grow\_total\_steps}})$
    $\texttt{set\_growth}(G, g)$; $\texttt{set\_growth}(D, g)$
    Load batch and move to GPU:

- $audio \leftarrow \texttt{batch["audio"]}$
- $pitch, vel, fam, src \leftarrow$ one-hot vectors
- $idx_p, idx_v, idx_f, idx_s \leftarrow$ one-hot $\rightarrow$ indices

    $logmel, instfreq \leftarrow \texttt{wav\_to\_spec}(audio)$
    Estimate current H, W from G output shape
    Resize $real\_spec \leftarrow \texttt{interpolate}(logmel, H, W)$
    Sample latent vector: $z \sim \mathcal{N}(0, I)$
    G: `eval`, D: `train`
    $fake\_spec \leftarrow G(z, pitch, src, vel, fam)$
    $real\_spec.requires\_grad = \texttt{True}$
    Get discriminator outputs:

- $D(real\_spec) \Rightarrow (rl, rp, rv, rf, rs)$
- $D(fake\_spec) \Rightarrow (fl, fp, fv, ff, fs)$

    Compute discriminator loss:

$$\mathcal{L}_D = \texttt{d\_loss\_fn}()$$

    Backpropagate and update discriminator: $d\_opt.\texttt{step}()$
    $global\_step \leftarrow global\_step + 1$
  end for
end for

The learning rate for the discriminator is set significantly lower than that of the generator to promote stable training dynamics and prevent the discriminator from overpowering the generator too early in the training process.

The optimizer used for both the generator and the discriminator is Adam, which is a well known optimizer known for producing stable convergence in deep learning models.

CHAPTER IV
RESULTS

Several experiments were conducted using different GAN architectures and training strategies. Among all the approaches tested, the model based on Progressive Growing of GANs (PGGANs) consistently produced the best results.

4.1 Vanilla GANs

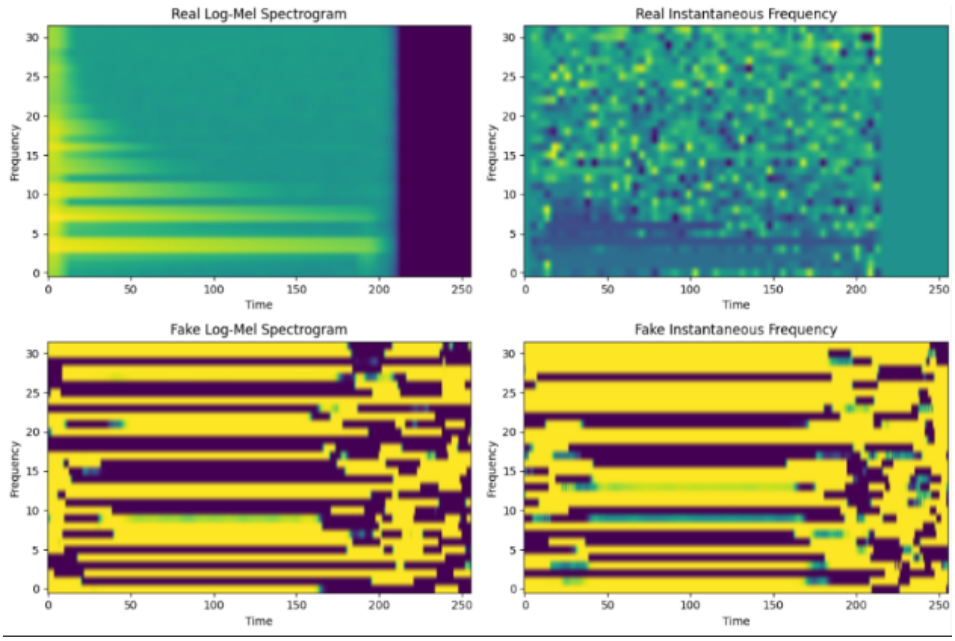


Figure 4.1: First experiment using a vanilla GAN architecture. Top: real log-mel spectrogram and real instantaneous frequency. Bottom: generated (fake) log-mel spectrogram and instantaneous frequency.

In this initial experiment, a vanilla GAN was used without progressive growing or auxiliary classification. As shown in Figure 4.1, both generated spectrograms failed to replicate the structure of real data, producing noisy and unrealistic outputs.

The main reason for this is due to instability of GANs as shown by the figure underneath

Figure 4.2: GAN training and validation losses for the vanilla GAN setup. Generator loss increases while discriminator loss remains low.

$$\nabla_G L_G = -\mathbb{E}_{z\sim p_z}\left[\frac{\nabla_G D(G(z))}{1 - D(G(z))}\right] \tag{4.1}$$

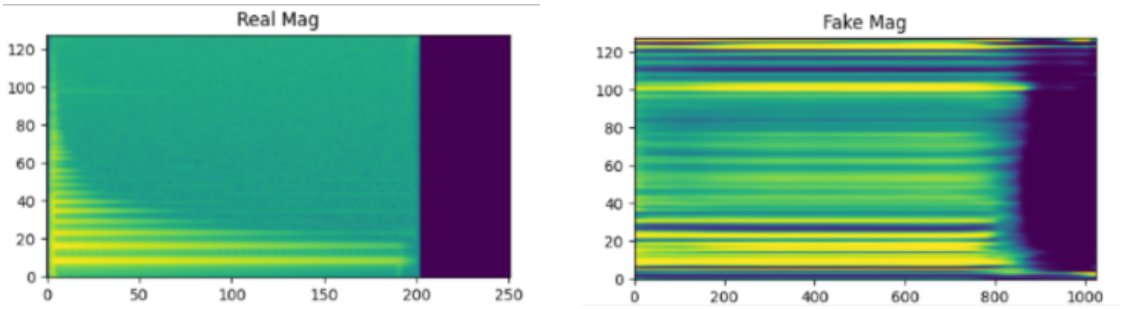
One major issue observed in the vanilla GAN experiment was that the generator loss steadily increased over time (Figure 4.2), while the discriminator loss remained near zero. This imbalance indicates that the discriminator became too strong, easily rejecting generated samples.

As shown in Equation 4.1, the gradient of the generator depends inversely on the discriminator's output. When the discriminator becomes too confident (i.e., $D(G(z)) \to 1$), the denominator $1 - D(G(z))$ approaches zero, causing the gradient to vanish. This harms the generator from learning.

Since the discriminator uses a sigmoid activation function, $D(G(z)) \to 1$ when it is highly confident that a sample is fake. This causes the denominator $1 - D(G(z)) \to 0$, pushing the overall gradient toward zero. As a result, the generator receives very little gradient signal and fails to improve, this is known as the **vanishing gradient problem**.

4.2 Progressive Growing of GANs (PGGANs)

Following the implementation described in the methodology, PGGANs were used to address the instability issues observed in the vanilla GAN setup. The results shown below demonstrate a significant improvement in the quality of generated spectrograms.



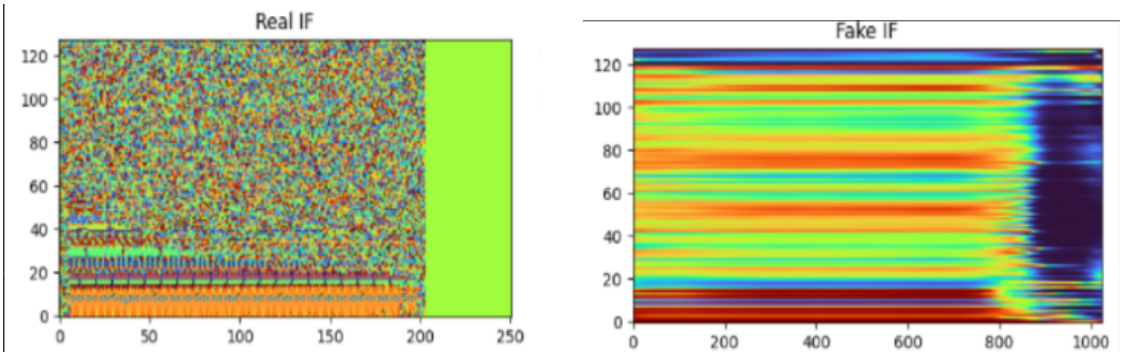
Real Log-Mel Spectrogram

Generated (Fake) Log-Mel Spectrogram using PGGAN

Figure 4.3: Comparison between real and PGGAN-generated log-mel spectrograms.

The results show a clear improvement over the vanilla GAN baseline. The generated log-mel spectrogram better captures the harmonic structure, particularly in the lower frequency regions. However, it continues to struggle with reproducing higher frequency details, and overall, the harmonic structure is still not accurately preserved.

This issue is further exacerbated when observing the instantaneous frequency spectrograms.



Real Instantaneous Frequency

Generated (Fake) Instantaneous Frequency

Figure 4.4: Comparison between real and generated instantaneous frequency spectrograms using PGGAN.

Modeling instantaneous frequency is especially challenging because it reflects subtle changes in phase over time. Even small deviations in phase can lead to large perceptual differences in sound. As shown in Figure 4.4, the generated IF spectrogram fails to capture the temporal variation seen in the real data, instead it produces smooth and structured patterns.

The table below compares classification accuracy across four attributes for both real and generated spectrograms

Attributes	**Generated Spectrograms**	**Real Spectrograms**
Pitch	99.12%	90.01%
Family	98.19%	73.49%
Source	98.31%	75.91%
Velocity	96.00%	26.11%

Table 4.1: Classification accuracy of different attributes on generated and real spectrograms.

Interestingly, classification accuracy on generated spectrograms is consistently higher than on real ones across all attributes. This discrepancy is likely because the discriminator adapts to the generator during training. Rather than learning to produce truly realistic spectrograms, the generator may learn to encode key features in ways that are easier for the discriminator to detect. This means the generator could be exploit-

ing shortcuts to encode pitch, family, and source more explicitly than real spectrograms, without actually generating realistic sounds. Essentially they are learning their own genrerted patterns!

Velocity, however, remains the most difficult attribute. While generated spectrograms achieve high accuracy, the real spectrograms perform poorly at just 26.11%. This suggests that velocity is inherently harder to classify. One reason is that velocity depends heavily on the instantaneous frequency (IF) spectrogram Since the model struggles to generate accurate IF representations, it fails to encode meaningful velocity attributes making it harder for the discriminator to learn this attribute effectively.

Additionally, the audio reconstructed from the generated spectrograms using the inverse STFT (iSTFT) was not coherent. Instead of resembling a musical note, the output often sounded like bursts of noise. This further supports the observation that the model fails to accurately capture phase-related information, which is especially sensitive to even small deviations. Since musical notes rely on precise harmonic and temporal structure, these slight phase errors result in audio that lacks any quality and coherence.

CHAPTER V
CONCLUSION

Conducting this project presented several challenges, particularly in terms of time and computational resources. Each training iteration took approximately 2–3 days even with an A100 GPU to complete, with an estimated cost of around 300 baht per run. Due to the slow iteration speed and limited budget, it was not feasible to run many experiments or fine-tune extensively.

More broadly, generating musical notes or any structured data like images is an inherently difficult task. The model must preserve fine-grained structure, especially at higher resolutions, where even small deviations in frequency, timing, or phase can lead to a complete failure in output quality. These findings highlight the complexity of GAN-based audio generation and the need for more efficient and robust models when working with high-fidelity data.

For future work, more recent and powerful generative models could be explored to overcome the limitations observed in this project. GANSynth was proposed in 2018, and since then, several advancements have emerged that offer improved stability, quality, and training efficiency. Transformer-based models such as *DiffWave*, *WaveGrad*, and *AudioLDM* have shown great results.

REFERENCES

[1] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation. In *International Conference on Learning Representations (ICLR)*, 2018.

[2] Jesse Engel, Cinjon Resnick, Adam Roberts, Sander Dieleman, Mohammad Norouzi, Douglas Eck, and Karen Simonyan. Neural audio synthesis of musical notes with wavenet autoencoders. In *Proceedings of the 34th International Conference on Machine Learning*. PMLR, 2017. URL `https://arxiv.org/abs/1704.01279`.

[3] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 2672–2680. Curran Associates, Inc., 2014.

[4] Google. Gan structure. `https://developers.google.com/machine-learning/gan/gan_structure`, n.d. Accessed: 2025-07-19.

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

[6] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. In *Advances in neural information processing systems*, pages 2234–2242, 2016.

[7] Jesse Engel, Lucas Hantrakul, Chenjie Gu, and Adam Roberts. Gansynth: Adversarial neural audio synthesis. In *International Conference on Learning Representations (ICLR)*, 2019.

[8] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Norbert Kalchbrenner, Andrew Senior, and Kieran Kavukcuoglu. Wavenet: A generative model for raw audio, 2016. URL https://arxiv.org/abs/1609.03499.

[9] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. Improved training of wasserstein gans, 2017. URL https://arxiv.org/abs/1704.00028. arXiv:1704.00028.

BIOGRAPHY

NAME	Mr. Vignesh Roachthavilit
DATE OF BIRTH	27 March 2003
PLACE OF BIRTH	Bangkok, Thailand
INSTITUTIONS ATTENDED	St Andrews International School Bangkok, 2019–2021
	Highschool Diploma (International Baccalaureate)
	Mahidol University, 2021–2025
	Bachelor of Science (Applied Mathematics)
	Mahidol University International College
E-MAIL	vigneshroachthavilit@gmail.com