Music Generation using GANs, VAEs and Transformers for Genre-Specific Composition, focused on Indian Classical Music.

A PROJECT REPORT

Submitted in partial fulfilment of the

Requirement for the award of the degree

of

BACHELOR OF TECHNOLOGY (B. Tech)

In

Data Science and Engineering

By

ADITYA PRAMAR

219309006

(Department of Data Science and Engineering)

MANIPAL UNIVERSITY

Jaipur 303007

Rajasthan, India

MAY 2025

MANIPAL UNIVERSITY JAIPUR

MANIPAL
UNIVERSITY JAIPUR
(University under Section 2(f) of the UGC Act)

Date : 29/04/25

CERTIFICATE

This is to certify that the project titled **A Study on Indian Classical Music Generation using GANs, VAEs, and Transformers for Genre-Specific Composition** is a record of the Bonafide work done by **ADITYA PRAMAR (219309006)** submitted in partial fulfilment of the requirements for the award of the Degree of **Bachelor of Technology (B.Tech) in Data Science and Engineering of Manipal University Jaipur** during the academic year 2024-25.

Dr. Neha V Sharma
Project Guide, Department of Data Science and Engineering
Manipal University Jaipur

Dr Akhilesh Kumar Sharma
HoD, Department of Data Science and Engineering
Manipal University Jaipur

Acknowledgements

I would like to express my sincere gratitude to all those who have supported and guided me throughout the course of this project. Firstly, I would like to extend my heartfelt gratitude to **Dr. Akhilesh Sharma**, Head of the Department of Data Science and Engineering for providing me with the necessary environment, and encouragement to undertake this project.

I am also deeply grateful to my project guide and mentor, **Dr. Neha V Sharma**, for her support, insights, and expert guidance. Her suggestions, critical feedback, and mentoring have played a crucial role in shaping the outcome of this work.

I am also extremely thankful **Adhiraj Chaudhuri,** grandson of Late Padma Bhushan Pandit Debu Chaudhuri, for his support in this project.

Furthermore, I extend my gratitude to **Dr. Kuldip Singh Sangwan**, **Dean FOSTA,** for his leadership and promoting research culture in the department which motivated me deeply

Abstract

Artificial intelligence has transformed several artistic and creative fields in the modern era, including music creation. However, computational musicology has not given Indian classical music, a centuries-old tradition renowned for its complex raga system and improvisational structure, the same kind of attention. The development of AI systems that respect and embody the fundamental ideas of Indian classical traditions in addition to producing music is urgently needed. To close that gap and honor the contribution of human artists, this project, "**Music Generation using GANs, VAEs, and Transformers for Genre-Specific Composition: A Case Study in Indian Classical Music**" was created. This project is intended to be a creative tool rather than a substitute for musicians, especially to help them come up with melodic ideas like “taans”, which are quick, complex note sequences that are usually played during raga improvisational sections. In practice and performance settings, this technology can operate as a digital collaborator by assisting musicians in discovering new musical variations within the same note framework.

The process started with the development of a dataset that collected more than 50 hours of performances of Hindustani classical music from recordings and archival collections that were made available to the public. To preserve each sequence's distinctive melodic structure and improvisational aspects, these performances were meticulously transcribed and manually divided into raga segments. The raw audio was transformed into pitch-duration sequences appropriate for deep learning models using MIDI transcription software. The model's exposure to the richness of the tradition was enhanced by the dataset's wide variety of ragas, which covered various times of day and emotional states. Preprocessing included tokenizing notes and normalizing tempo changes.

Three different generative models were used in the project: Transformer-based architectures, Variational Autoencoders and Generative Adversarial Networks. Each model was trained on its own and assessed using qualitative review and loss functions. Creating brief to medium-length musical passages that resembled conventional bandish, or improvisational portions was the main goal.

The results of the project include the successful generation of raga-specific compositions that adhere to the structure of Indian Classical Music. The models achieved an average pitch class distribution cosine similarity score of **0.79** with ground truth and note duration entropy within **5% deviation** from original. A notable outcome was the ability to generate realistic *taans*, thereby assisting artists in improvisation and practice.

Tools and technologies used in this project include Python, TensorFlow, PyTorch, and prettyMIDI for MIDI processing and symbolic music analysis. Google Colab was utilized for model training, and custom plotting scripts were developed to visualize MIDI patterns, pitch classes, note density, and temporal evolution.

LIST OF TABLES

LIST OF FIGURES

Contents

INTRODUCTION

An essential component of human expression has always been music. Music has changed with technology and continues to influence our cultural identity, from tribal rhythms and ancient chants to intricate classical forms and contemporary electronic composition. Indian classical music is unique among musical traditions worldwide because of its complex melodic patterns, improvisational style, and profoundly spiritual origins. It is a system that emphasizes performance and emotion and is based on ragas and taals.

1.1 Introduction to work done/ Motivation (Overview, Applications & Advantages)

The taan is among the most emotive and technically complex components of Indian classical vocal music. In essence, a taan is a quick melodic run that is frequently played at a raga's climax. It entails quick note sequences sung with accuracy, passion, and rhythmic diversity. Vocalists frequently employ taan singing to show off their mastery of a raga, although it takes a great deal of practice and imagination. These sections are stunning and captivating to listen to, but they are also very challenging to generate, analyze, or notate computationally.

The field of generative artificial intelligence has grown significantly in recent years, particularly in creative fields like music, language, and art. It is now feasible to produce paintings, stories, and even music that are nearly identical to those created by humans because to models like Transformers, Variational Autoencoders, and Generative Adversarial Networks. However, the majority of these developments have concentrated on Western music, especially when it comes to symbolic representations like raw audio data or MIDI files. However, when it comes to AI music generation, Indian classical music is still mostly unexplored. Its intrinsic complexity, the scarcity of structured datasets, and the absence of standardized notation are the main causes of this. Indian classical music depends on improvisation, microtones (shruti), and oral transmission as opposed to Western classical music, which uses sheet music and structured harmony. Because of this, robots find it very challenging to pick up meaningful patterns, particularly in improvisations like taans. The goal of this project is to close that gap.

There could be several practical uses for taans if a trustworthy generative model could be created. For example:

1. Music Education: To help them practise improvisation, beginning or intermediate students of Hindustani or Carnatic music could utilise the model to listen to and examine taan patterns made from particular ragas or phrases.

2. Musical Preservation: The improvisational methods of great vocalists whose work may not be fully

documented in writing form may be preserved by creating new taans from current datasets.

3. Creative Collaboration: By fusing traditional art forms with contemporary technology, musicians could utilize AI-generated taans as inspiration or as raw material for additional composition.

From a purely research perspective, the work is also pertinent. The question of whether machine learning models that perform effectively in highly structured domains can also adapt to fluid, expressive systems with less constraints is what Indian classical music offers as a distinct challenge to AI. Learning musicality, style, and expression is more important than simply producing "correct" sounds.

That being said, we were also realistic about the limitations. There is no large-scale, publicly available dataset of Indian classical taans in symbolic format. So the first hurdle was to either create or find a dataset suitable for this kind of modeling. We eventually worked with MIDI data that either came from online symbolic transcriptions or that we manually cleaned and labeled. Another challenge was evaluating the results — since taans are highly improvisational, standard evaluation metrics like loss or accuracy aren't always meaningful. Instead, we combined some statistical analysis with subjective human listening and visual inspection of note sequences.

1.2 Project Statement

The primary goal of this project is to investigate the capabilities of modern generative AI models to synthesize Indian classical taans using symbolic music data.

To achieve this, we laid out the following key objectives:

1. **Data Collection and Preprocessing**
    - Gather a suitable dataset of Indian classical taans, preferably in MIDI or other symbolic formats.
    - Clean and normalize the data so that it can be used as input for deep learning models.
2. **Model Development and Training**
    - Design and implement three different generative model architectures — a VAE, a GAN, and a Transformer.
    - Train these models on the pre-processed data, tune hyperparameters, and compare performance.

3. **Evaluation and Analysis**
    - Use automatic metrics (like note distributions) to evaluate generated taans.
    - Conduct subjective evaluations through visual inspection of note plots and informal listening sessions.
4. **Comparative Study**
    - Analyse which model performs better in terms of musicality, diversity, and learning curve.
    - Identify common errors and limitations across models when generating taan-like sequences.
5. **Documentation and Reporting**
    - Present the entire methodology, implementation, and results in a clear and structured format.
    - Suggest future directions for work in this field, including ways to expand the dataset and improve models.

1.3 Organization of Report

This report is structured into six chapters, each focusing on a different aspect of the project:

- **Chapter 1: Introduction**
  Introduces the problem domain, motivation, goals, and structure of the report.
- **Chapter 2: Background Material**
  Provides the theoretical background needed to understand the problem, including information about taans, MIDI data, and generative models.
- **Chapter 3: Methodology**
  Describes the step-by-step process followed in the project.
- **Chapter 4: Implementation**
  Explains the technical setup, challenges faced during development, and how the models were integrated and deployed for generation.
- **Chapter 5: Results and Analysis**
  Presents the performance of the models using both automatic metrics and subjective evaluation. Includes visuals like note plots and generated sequences.
- **Chapter 6: Conclusions and Future Scope**
  Summarizes the outcomes of the project and offers insights into possible improvements and further research.

Background Material

2.1 *Conceptual Overview*

Understanding the theory underlying both is crucial when working on a project that combines contemporary AI models with traditional Indian classical music. The key ideas that we have used throughout the piece, both musically and computationally, are covered in this part. It combines science and art, which is what made this project both enjoyable and difficult.

2.1.1 Introduction to Western and Indian Classical Music

Classical music refers to structured musical traditions that are based on established theoretical foundations. Different nations have created unique systems of classical music over the centuries, each reflecting its particular aesthetic principles, social roles, and technical methods. Indian and Western classical music are two of the most well-known classical traditions. Despite being broadly classified as "classical," they differ greatly in terms of musical theory, structure, and performing techniques.

Western classical music has its roots mostly in Europe and is composed using set notation, harmony, and counterpoint. It emphasizes polyphonic textures, which are characterized by the simultaneous playing of several notes or voices to produce harmonic progressions. This system developed throughout several eras, each with distinct stylistic traits: Baroque, Classical, Romantic, and Modern. Standard staff notation is typically used to write compositions, which are intended to be performed precisely as written. The fundamental musical content is set, but interpretation is permitted to a certain degree.

Indian classical music, on the other hand, is mostly spontaneous and melodious. Hindustani (North Indian) and Carnatic (South Indian) classical music are its two main systems. Both are founded on the ideas of taal, which describes rhythmic cycles, and rāga, which establishes a framework for melody. Indian classical traditions do not employ harmony in the same systematic manner as Western music. Rather, performances center on the development and investigation of a rāga inside a certain taal, frequently in an impromptu and highly customized way. Although there is notation, it is not strictly adhered to, and the oral tradition is important.

To provide a clearer understanding, the following table compares the two classical music systems across multiple dimensions:

Aspect	Western Classical Music	Indian Classical Music
Basis	Harmony, melody, and rhythm	Raga and Taal
Composition	Fixed	Improvisational
Notation	Staff based with time signatures and Key signatures	Sargam based notation
Teaching	Conservatory and Notation based	Oral, Guru Shishya based
Pitch System	12 - Tone	Microtonal
Emphasis	Structural Complexity and Harmony	Emotional Expression and Improvisation
Cultural Context	Concerts and Performances	Spiritual and Meditative

Table 1 : Differences between Indian and Western Classical Music

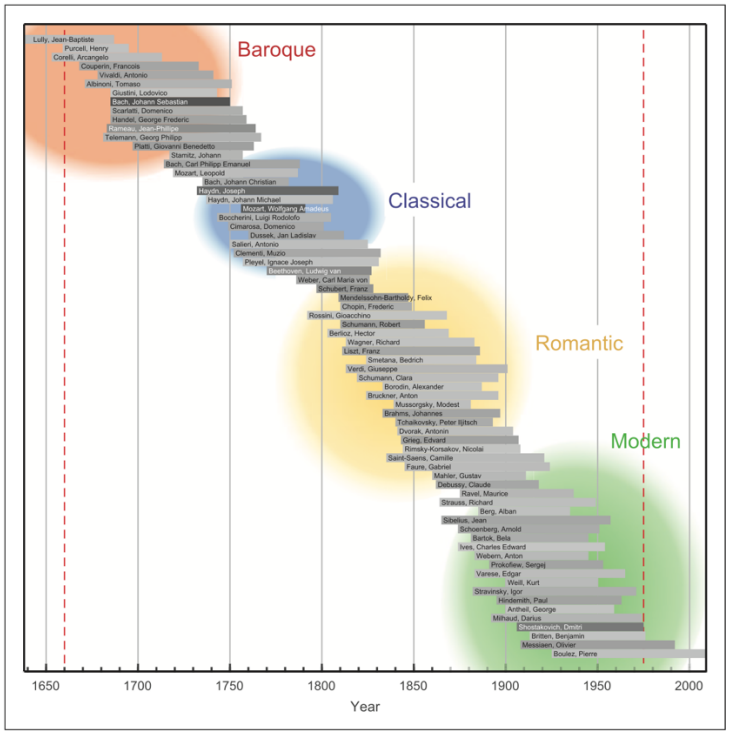


Figure 1 : Important Periods in Western Classical Music [1]

2.1.2 Swaras and Thaats

In Indian classical music, **swaras** are the fundamental building blocks of melody. A swara refers to a musical note or pitch, much like the Western solfège system, but with its own unique structure and interpretation. There are **seven basic swaras**, known collectively as the **Saptak**:

Sa (Shadja)

Re (Rishabh)

Ga (Gandhar)

Ma (Madhyam)

Pa (Pancham)

Dha (Dhaivat)

Ni (Nishad)

These swaras correspond loosely to the Western major scale: Sa ≈ Do, Re ≈ Re, and so on. However, this is only approximate, because Indian classical music often employs **microtonal variations** that do not exist in the 12-tone equal-tempered Western scale. Out of these seven swaras:

- **Sa and Pa** are considered *achala swaras*, meaning they are fixed and never change.
- The remaining five — Re, Ga, Ma, Dha, and Ni — are *vikrit swaras*, meaning they can be either **shuddh** (natural), **komal** (flat), or in the case of Ma, **tivra** (sharp)

Indian music can also be divided into three main octaves or **Saptaks**:

- **Mandra Saptak** – The lower octave
- **Madhya Saptak** – The middle octave (most commonly used)
- **Taar Saptak** – The higher octave

Each swara can occur in these octaves, and musicians often glide between them to enhance expression.

A **thaat** is a classification system for organizing **rāgas**. It is similar in purpose to the concept of *modes* or *scales* in Western music, though the way it is used is quite different. The **thaat system** was formalized by **Vishnu Narayan Bhatkhande**, a prominent musicologist of the early 20th century, in an attempt to standardize and document Hindustani classical music theory.

There are **10 major thaats** in Hindustani music:

1. **Bilawal** – All natural (shuddh) notes
2. **Kalyan** – Includes tivra Ma
3. **Khamaj** – Komal Ni
4. **Bhairav** – Komal Re and Dha
5. **Purvi** – Komal Re, Dha and Tivra Ma
6. **Marwa** – Komal Re and Tivra Ma

7. **Kafi** – Komal Ga and Ni
8. **Asavari** – Komal Ga, Dha, and Ni
9. **Bhairavi** – Komal Re, Ga, Dha, and Ni
10. **Todi** – Komal Re, Ga, Dha and Tivra Ma

Each **rāga** belongs to one of these thaats, but not all possible combinations of notes make a rāga. A rāga must follow specific rules for ascent (**Arohana**) and descent (**Avarohana**), as well as have a dominant note (**Vadi**) and a secondary note (**Samvadi**), among other features.

While thaats are **theoretical frameworks**, rāgas are the **practical, performed expressions**. For example:

- Raga **Yaman** belongs to **Kalyan thaat**
- Raga **Bhairavi** belongs to **Bhairavi thaat**
- Raga **Darbari Kanada** belongs to **Asavari thaat**

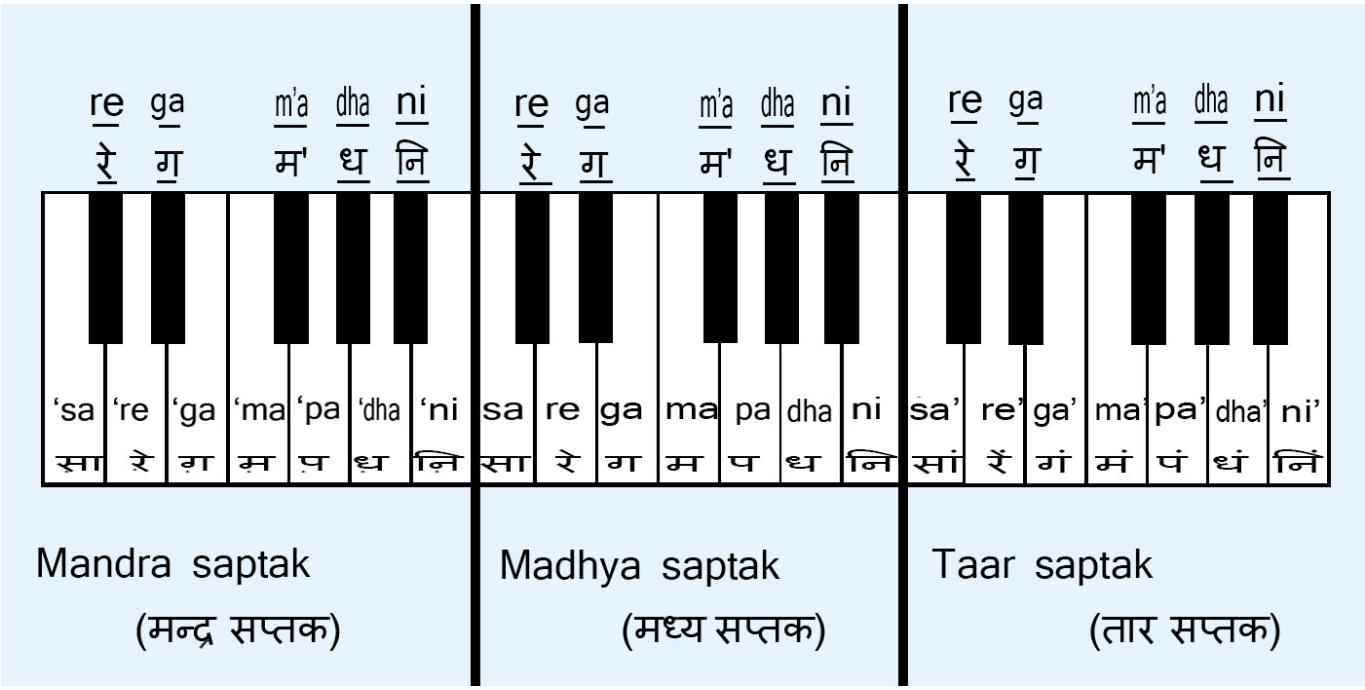


Figure 2 : Indian Classical Saptaks [3]

Sr. no.	Western Notation	Indian Notation	Frequency Ratio(natural)
1	C	Sa	1/1
2	C#	Re	16/15
3	D	Re	9/8
4	Eb	Ga	6/5
5	E	Ga	5/4
6	F	Ma	4/3
7	F#	Ma’	7/5
8	G	P	3/2
9	Ab	Dh	8/5
10	A	Dh	5/3
11	Bb	Ni	9/5
12	B	Ni	15/8
13	C’	Sa’ (Next octave)	2/1

Table 2 : Western and Indian notes with frequency ratio [2]

When using machine learning or symbolic data models like MIDI to generate Indian classical music, understanding thaats is crucial. Since thaats define the allowable pitch sets for a rāga, they can act as constraints for generative models. This helps keep the generated music musically meaningful and true to the classical framework, rather than producing arbitrary or culturally inconsistent sequences.
In the next section, we will discuss how **Rāgas** are formed using swaras and thaats, and how they are used in real performances.

2.1.3 Ragas

The concept of a *rāga* lies at the heart of Indian classical music, functioning as both a melodic framework and a medium for emotional expression. Unlike Western classical scales, which are more rigidly defined in terms of fixed notes and structures, a rāga is a living, evolving idea that embodies certain rules while still allowing the performer a significant degree of creative freedom. It isn’t merely a set of notes; rather, it is a system that governs how notes are to be approached, combined, and rendered over time, particularly in performance. Every rāga carries with it an emotional essence—be it devotion, longing, serenity, or excitement—and it is the responsibility of the artist to evoke that sentiment authentically.

At a basic level, a rāga is defined by a specific selection of notes (swaras) taken from the twelve-note octave system of Indian classical music. A rāga usually consists of five to seven of these notes in ascending (*Aaroh*) and descending (*Avroh*) order.

However, what makes a rāga unique is not just which notes are used, but how they are used—the *pakad* or characteristic phrases, the *vaadi* (dominant note), the *samvaadi* (subdominant note), and the rules governing the movement from one note to another.

For instance, let's take **Rāga Bhairav**, which is traditionally performed in the early morning. It is known for its serious, devotional, and meditative quality. The scale structure for Bhairav is:

- **Arohana**: Sa Re (komal) Ga Ma Pa Dha (komal) Ni Sa
- **Avarohana**: Sa Ni (komal) Dha Pa Ma Ga Re (komal) Sa

Here, both Re and Dha are komal (flat), which gives Bhairav its distinctive somber and introspective feel. What's interesting is that although the notes themselves can be found in other rāgas, the way Bhairav moves—its emphasis on slow transitions, elongated Re, and the calm resolution on Sa—makes it unique. There's also a particular gamak (oscillation) that is typically applied to Re and Dha, which further sets the mood.

Aaroh	Avroh	Pakad
S r G M P d N S'	S' N d P M G r S	M M d d P M P M M r r S

Table 3 : Sample Aaroh Avroh of Raga Bhairav

This kind of approach to melodic structure makes computational analysis both challenging and interesting. It's not enough to identify the notes; the sequence, duration, and expression of those notes are just as crucial to determining the rāga. This is why modern research in computational musicology often includes pattern mining of Arohana-Avarohana structures [3].

RAGA	SWARA combination	AROHANA/ AVAROHANA
1 Todi	S r1 g1 m1 p d1 n1	S r1 g1 m1 p d1 n1 s'/ s' n1 d1 p m1 g1 r1 s
2 Dhanyasi	s r1 g1 m1 p d1 n1	S g1 m1 p n1 s' /s' n1 d1 p m1 g1 r1 s
3 Varali	s r1 g1 m2 p d1 n2	s r1 g1 m2 p d1 n2 s'/s' n2 d1 p m2 g1 r1 s
4 Mayamalavagaula	s r1 g2 m1 p d1 n2	s r1 g2 m1 p d1 n2 s'/s' n2 d1 p m1 g2 r1 s
5 Saveri	s r1 g2 m1 p d1 n2	s r1 m1 p d1 s'/s' n2 d1 p m1 g2 r1 s
6 chakravak	s r1 g2 m1 p d2 n1	s r1 g2 m1 p d2 n1 s'/s' n1 d2 p m1 g2 r1 s
7 Gaula	s r1 g2 m1 p n2	S r1 m1 p n2 s'/s' n2 p m1 g2 r1 s
8 Kamavardhini	s r1 g2 m2 p d1 n2	s r1 g2 m2 p d1 n2 s'/s' n2 d1 p m2 g2 r1 s
9. Saurashtra	s r1 g2 m1 p d2 n1 n2	s r1 g2 m1 p d2 n2 s'/s' n1 d2 n1 d2 p m1 g2 r1 s
10. Abheri	s r2 g1 m1 p d2 n1	s g1 m1 p n1 s'/s' n2 d2 p m1 g1 r2 s
11. Anandabhairavi	s r2 g1 m1 p d2 n1	S g1 r2 g1 m1 p d2 p s'/s' n1 d2 p m1 g1 r2 s
12. Bhairavi	s r2 g1 m1 p d1 d2 n1	s r2 g1 m1 p d2 n1/s' n1 d1 p m1 g1 r2 s

Table 4 : Samples of Arohana Avarohana Patterns In Ragas[3]

2.1.4 Music Instrument Digital Interface Format and Its Importance in Music Processing

MIDI, which stands for **Musical Instrument Digital Interface**, is one of the most essential formats in the world of digital music. Introduced in the early 1980s, it completely revolutionized how music was created, shared, and analyzed. Developed in the early 1980s by engineers and music technology companies like Roland and Sequential Circuits, the MIDI cable was designed to allow electronic musical instruments, computers, and other devices to communicate with each other.

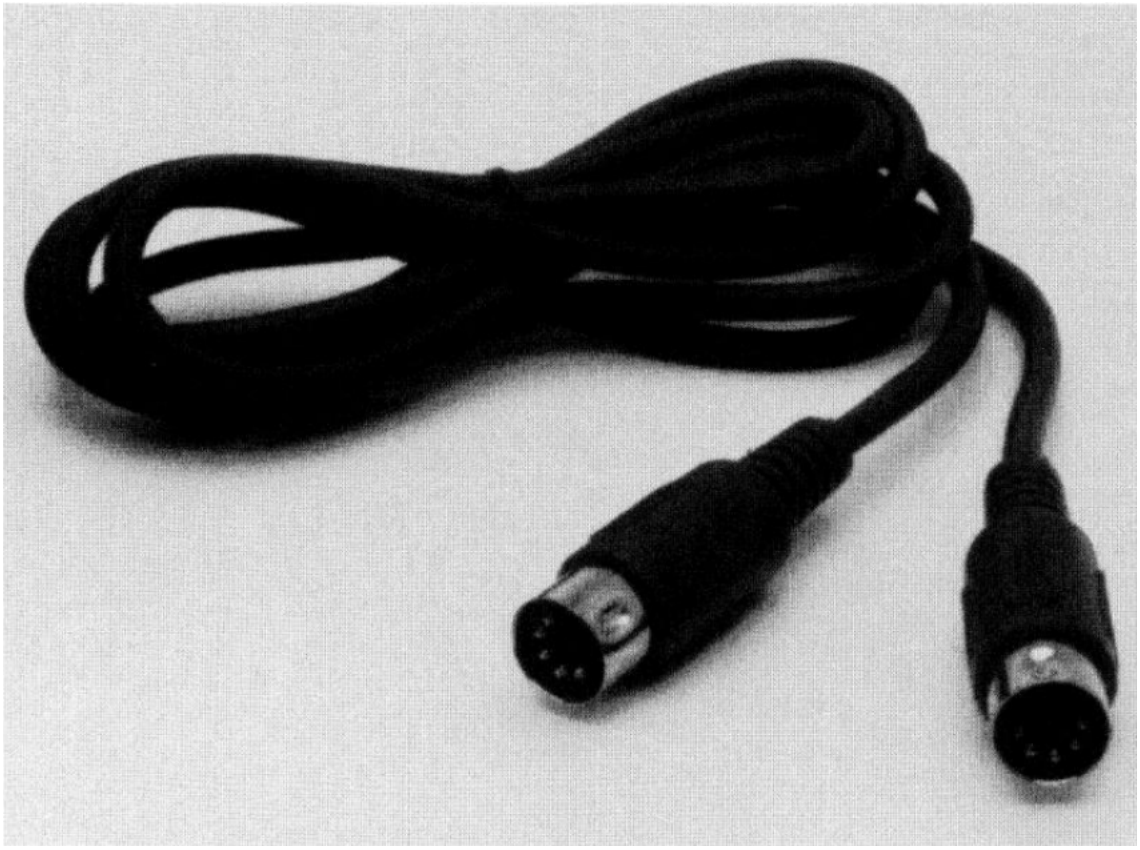


Figure 3 : MIDI Cable [5]

The cable itself, with its five-pin DIN connector, wasn't used for audio — it didn't carry sound at all. Instead, it carried **digital messages** that represented actions like "Play middle C," "Increase volume," or "Switch to violin sound." This was a huge shift. Instead of sending a sound wave, MIDI devices were sending commands that any compatible device could interpret and reproduce using its own sound engine.

This made it possible for a keyboard to control a synthesizer module, or for a sequencer to control multiple devices at once.

Today, when we talk about MIDI, we mostly refer to the **Standard MIDI File** — a lightweight, symbolic representation of music. But its roots in hardware and communication protocols are what made it so widely adopted in the first place. In the context of **music analysis**, this transformation from a hardware interface to a symbolic format is incredibly important. By breaking music down into note numbers, velocities, durations, and control data, MIDI offers a clean, noise-free way to represent music. Unlike raw audio files that capture every nuance in a performance, MIDI allows researchers and algorithms to work with **idealized, machine-readable versions of music**. This is especially useful when you're trying to study patterns, rhythms, or melodies exactly the kind of things that matters in Classical music research.

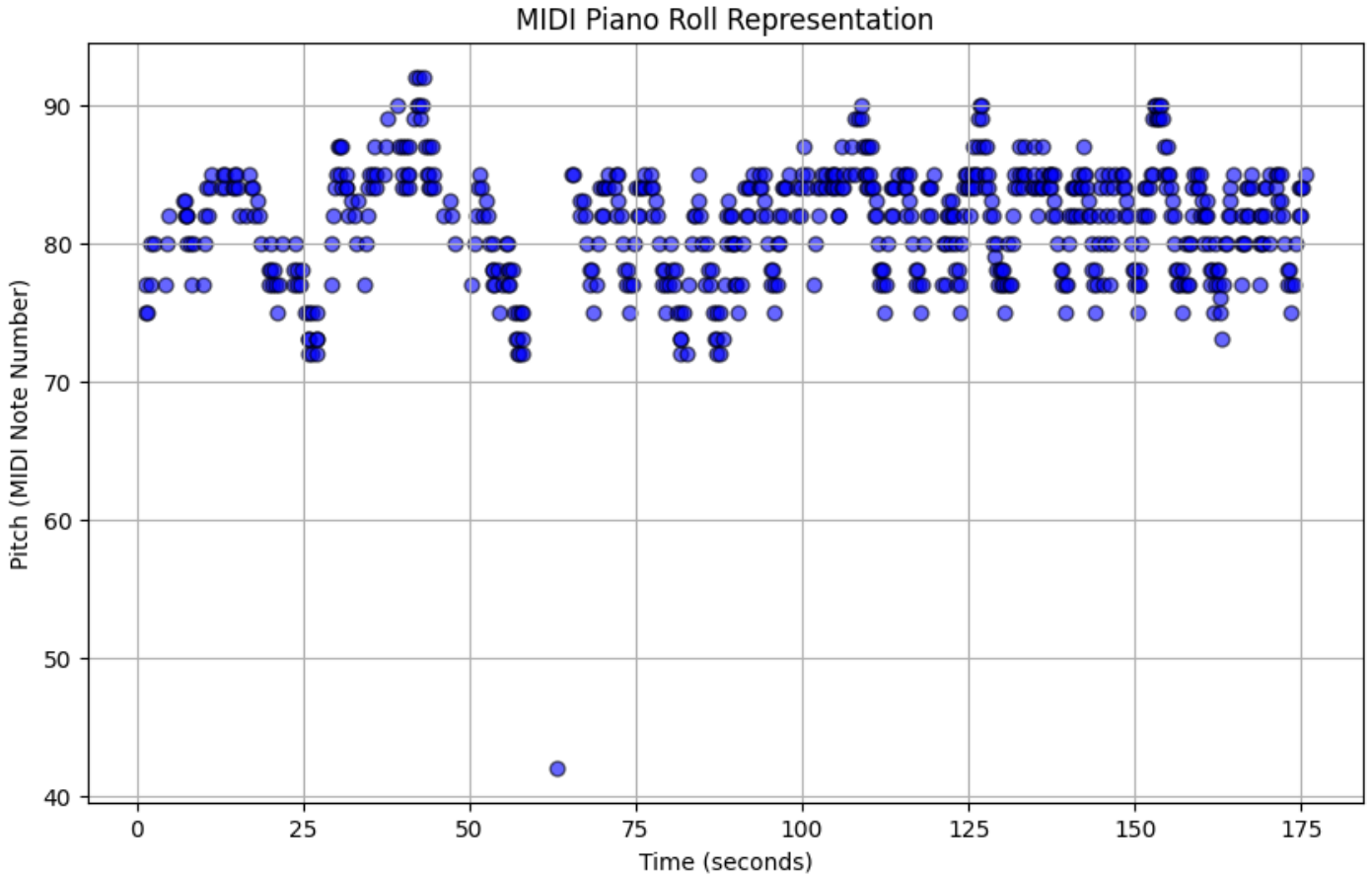


Figure 4 : Pitch – Duration Representation of MIDI Data

A MIDI file (.mid) does not store sound but rather instructions to recreate a piece of music. This includes data like:

- **Note On/Off Events**: Information about when a note is pressed and released.
- **Velocity**: How forcefully a note was played, which often corresponds to loudness.
- **Tempo**: Beats per minute (BPM) settings.
- **Control Changes**: Adjustments to volume, pan, modulation, sustain, etc.
- **Program Changes**: Used to change instrument sounds (e.g., piano to violin).
- **Pitch Bend**: Smooth alterations in pitch.

Each MIDI message is stored as a **binary-coded series of bytes**, with **channel numbers** allowing for multi-instrument tracking. For instance, a MIDI file could have separate channels for vocals, piano, tabla, and tanpura.

A typical MIDI file has two major chunks:

1. **Header Chunk**: Contains general information like file format (0, 1, or 2), number of tracks, and timing data.
2. **Track Chunks**: Each track consists of a sequence of MIDI events and meta events (like lyrics or tempo).

Header Chunk				
Type of Chunk (MThd in ASCII)	Length (bytes)	Format	Number of Track Chunks	Division (ticks)
4D 54 68 64	00 00 00 06	00 01	00 05	00 60

(a)

Track Chunk						
Type of Chunk (MTrk in ASCII)	Length (bytes)	Delta Time$_1$ (ticks)	Event$_1$	···	Delta Time$_n$ (ticks)	Event$_n$
4D 54 72 6B	00 00 01 3B	00	C1 2E	···	83 00	FF 2F 00

(b)

Figure 5 : MIDI Chunks [6]

One of the key advantages of this is the precision and flexibility. Every element in a musical performance—be it note timing, articulation, or volume is recorded as discrete data points. This makes it far easier to isolate specific musical attributes for analysis. Unlike audio, which often requires complex signal processing to extract pitch or tempo information, MIDI offers that data upfront. As Rothstein explains, this level of control and clarity made MIDI a foundational technology not just for performance, but also for research and pedagogy in digital music environments [7].

To take a specific example, let's consider the ***rāga Bhairav***. This rāga features a flattened second (komal Re) and sixth (komal Dha), which are crucial to its character. In a MIDI file, the pitches corresponding to these notes can be tracked and compared across various recordings of Bhairav. Researchers can then use this MIDI pitch data to visualize how artists interpret the rāga differently. Some might hover more around the komal Re, while others emphasize the transition from Ma to Dha more prominently. Because MIDI captures the pitch sequence exactly, these variations become easier to quantify and study. This process would be far more difficult and error-prone with audio, especially due to the presence of vocal inflections, ambient noise, and pitch drift. MIDI is also a favorite format in machine learning for music generation and classification tasks. Since it's structured and noise-free, it doesn't require preprocessing steps like Fourier transforms or spectrogram extraction. Instead, MIDI data can be converted directly into token sequences or note embeddings that can be fed into deep learning models. Modern architectures like LSTMs and Transformers have shown excellent results in generating new compositions using MIDI as input [8][9][10]. These models can learn temporal dependencies, rhythmic styles, and even emulate the ornamentation found in rāga performances.

However, MIDI is not without its limitations, especially when it comes to Indian music. One challenge is that Indian classical music often includes microtonal shifts, slides (meend), and ornamentations (gamakas), which are difficult to represent in standard MIDI format. The default MIDI scale is based on the 12-tone equal temperament system, whereas Indian music uses 22 shrutis and various subtle pitch inflections. Basic MIDI doesn't always offer the resolution necessary to encode these nuances accurately.

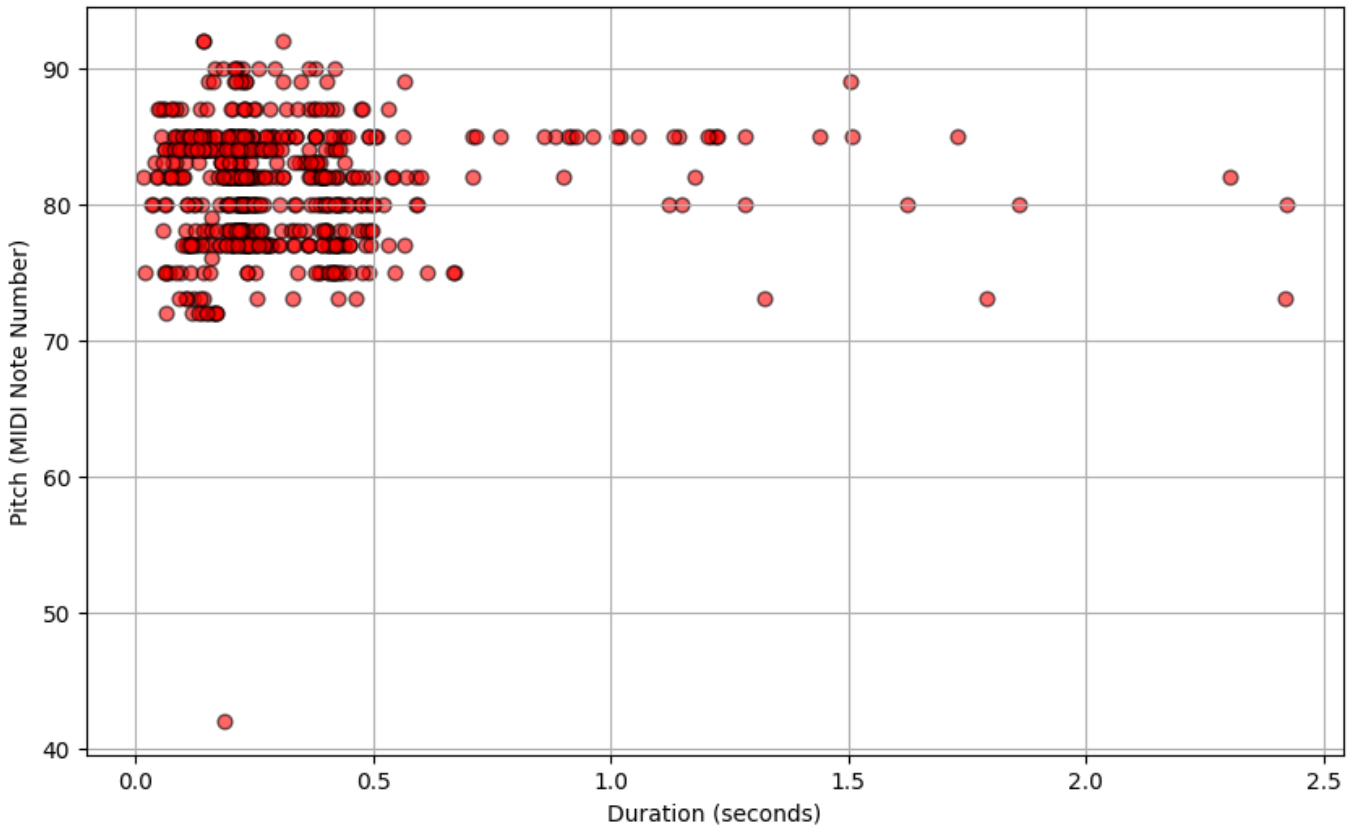


Figure 6 : Note Durations Vs Pitch in MIDI Files

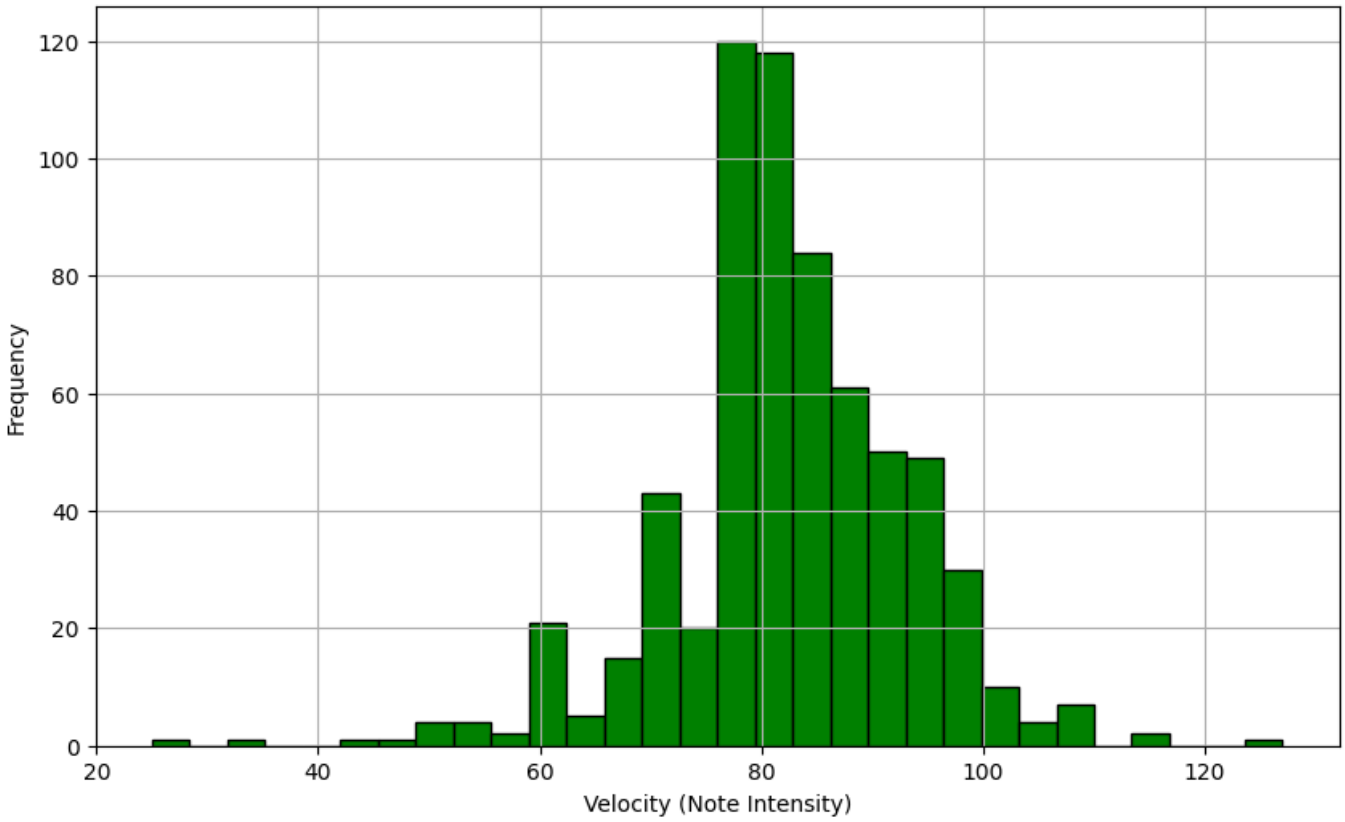


Figure 7 : Distribution of Note Velocities in MIDI

2.1.5 Signal Processing in Music

Signal processing is one of the most foundational pillars behind modern computational music analysis. It involves the study and manipulation of signals time-varying quantities that convey information. In the context of music, these signals are typically audio waveforms that represent sound pressure variations over time. Understanding how music is captured, processed, and interpreted by machines starts with grasping the essentials of signal processing.

At its core, a musical signal is a continuous-time analog waveform, but in most practical applications today, we deal with its digital counterpart. A digital audio signal is produced by sampling the continuous waveform at discrete time intervals (usually 44.1 kHz or higher for high-quality recordings) and then quantizing the amplitude values. The result is a sequence of numbers that can be stored, analysed, and manipulated by digital systems. One of the most essential operations in this domain is the transformation of audio signals from the time domain to the frequency domain. This is achieved through mathematical tools such as the Fourier Transform and its real-world adaptation, the Short-Time Fourier Transform (STFT). By breaking the signal into small overlapping segments and computing the frequency content of each, STFT enables researchers to observe how the pitch and intensity of musical elements change over time. This yields a time-frequency representation known as a spectrogram—a visual matrix that reveals musical contours, harmonic content, and rhythmic structures in a single frame. Spectrograms have become one of the most commonly used tools in music analysis, both in Western and Indian classical traditions.

The raw waveform, though informative, is often too detailed and noisy for practical analysis, particularly when the goal is to identify musical patterns or perform classification tasks. Therefore, signal processing typically involves a layer of abstraction known as feature extraction. Features are numerical summaries of the signal that capture specific musical properties. For example, some features are designed to represent the tonal quality or timbre of a sound, while others may reflect rhythmic patterns, pitch evolution, or loudness dynamics. Among the most widely used features are Mel-Frequency Cepstral Coefficients (MFCCs), which are derived from the spectral envelope and closely aligned with human auditory perception. MFCCs are particularly effective in distinguishing instruments or vocal timbres.

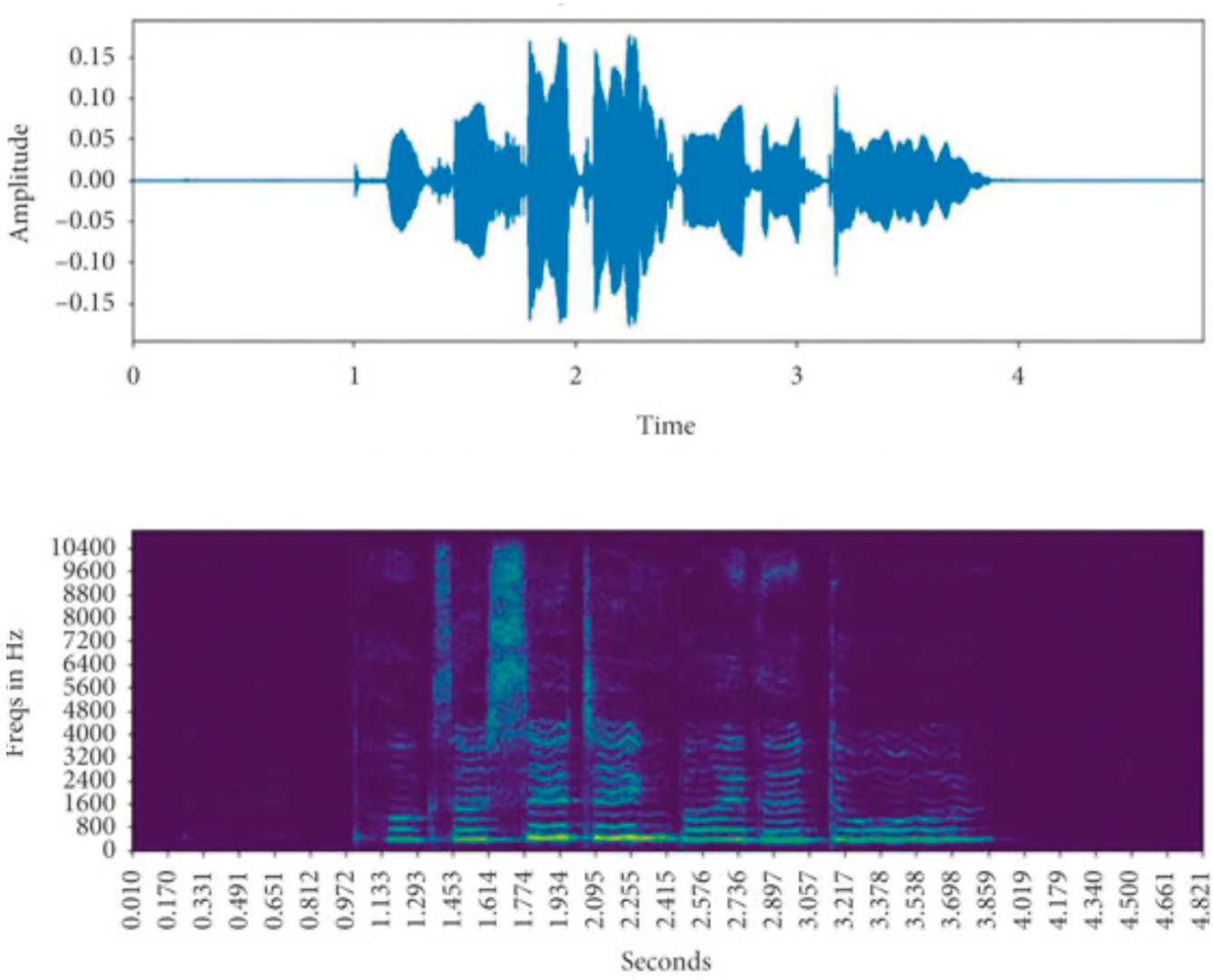


Figure 8 : Music Waveform and Spectrogram

2.1.6 Techniques for Music Analysis

Music analysis is a broad field that encompasses various methods to extract information from music data, whether in the form of audio signals or symbolic representations like MIDI. These techniques enable researchers, musicians, and engineers to gain insights into musical structures, patterns, and emotional content. In this section, we discuss some key methods employed in music analysis.

One of the most fundamental techniques in audio signal processing is **spectral analysis**, which provides a means to examine the frequency content of an audio signal over time. Spectral analysis helps in identifying the harmonic structure of music and can be crucial for tasks such as instrument recognition, pitch detection, and music genre classification. The **Fourier Transform** is a core tool for spectral analysis. It decomposes a signal into its constituent frequencies, allowing the visualization of the spectral content of an audio signal. This is commonly done using the **Short-Time Fourier Transform** (STFT), which provides a time-frequency representation by dividing the signal into small overlapping segments, transforming each into the frequency domain, and then combining them. This method is especially useful for identifying transient sounds and tonal content in music, allowing for analysis of different aspects of sound, such as pitch, timbre, and rhythm. In musical applications, spectral analysis is frequently used to detect musical features like chords, tempo, and key. In MIDI-based analysis, signal processing involves the extraction of features like note sequences, intervals, rhythms, and dynamics, which are all encoded in the event structure of a MIDI file. These features serve as the foundation for tasks such as genre classification, style identification, or even automatic music generation. A key challenge with MIDI data is how to efficiently transform these sequences into a form that machine learning models can process effectively.

Music, especially in Indian Classical genres, follows complex melodic patterns, often defined by the rāga's structure. Sequence learning techniques like **Recurrent Neural Networks, Long Short-Term Memory** networks, and **Transformers** are ideal for learning from such data. These models are designed to capture temporal dependencies in data, such as the way one note or phrase leads to the next.

- **RNNs** and **LSTMs** are good at capturing long-range dependencies in sequences. In Indian Classical music, these dependencies could represent the way a melodic phrase evolves over time. These models can be trained to learn patterns in a specific rāga and then generate new, related compositions.

- **Transformers**, which have gained significant popularity for sequence tasks, excel at capturing long-range dependencies even better than RNNs and LSTMs. They work by attending to different parts of the sequence in parallel, making them computationally efficient and capable of capturing intricate

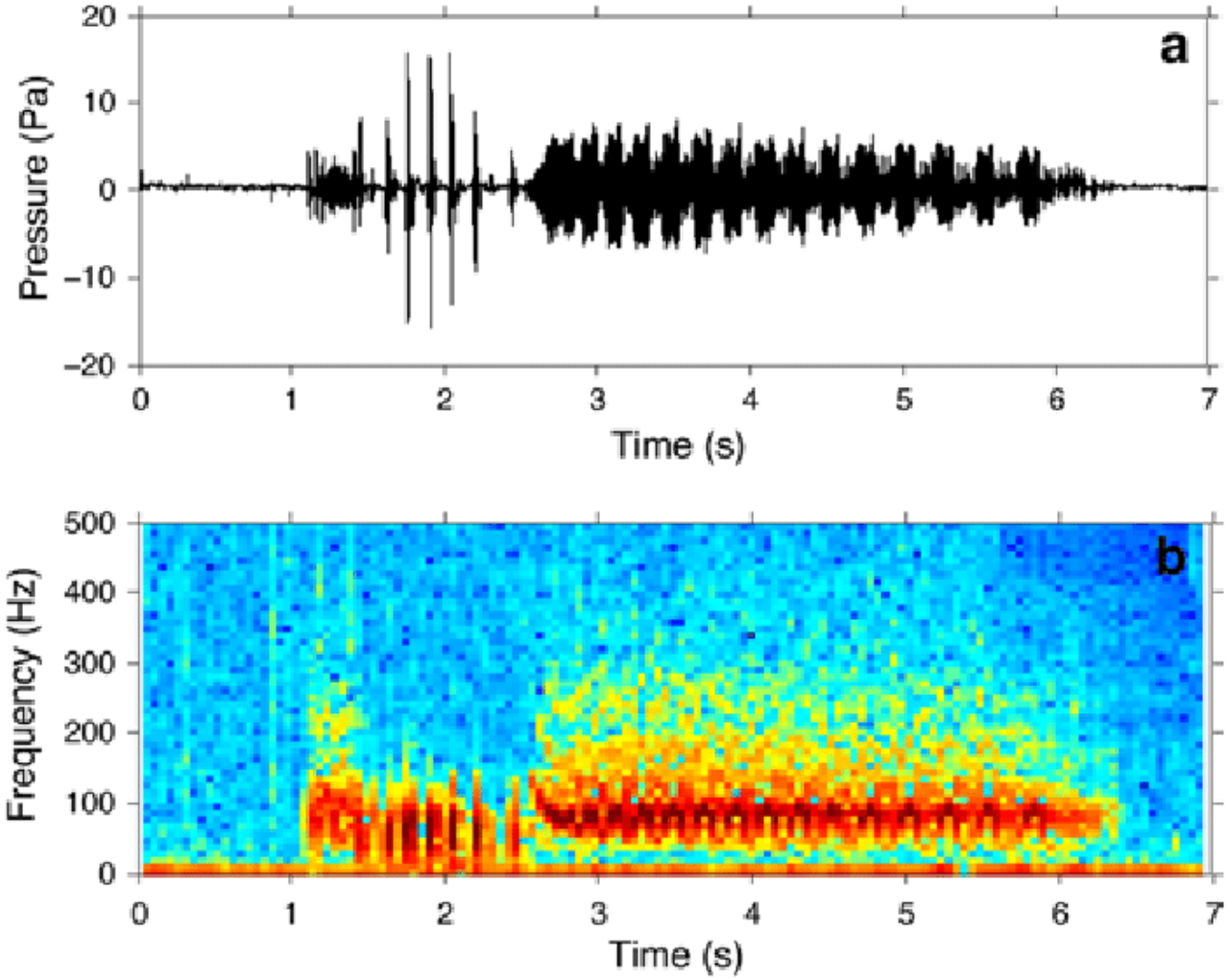


Figure 9 : Oscillogram and spectrogram (fast Fourier transform size = 1,024 points) of sound.

musical patterns over long sequences. For Indian Classical music, transformers can be trained on vast collections of MIDI data to identify intricate relationships between notes, timing, and pitch in a specific rāga. To enrich the training data, **data augmentation** techniques are often employed. For MIDI data, one common technique is **transposition**, which involves shifting the entire sequence of notes up or down by a fixed interval. This simulates playing the same composition in different keys or pitches, which is particularly useful for Indian Classical music, where the same melodic structure can be performed at varying pitches. **Rhythm modulation** involves altering the timing of notes to simulate changes in tempo or rhythmic feel, which is important for genres like Indian Classical music, where rhythmic structures (such as the *taal*) play a significant role. Another technique, **tempo adjustment**, involves changing the speed of the music while maintaining the relative timing of the notes, allowing the model to learn compositions at different tempos. **Velocity modification** is also a useful technique, as it can simulate variations in how a note is played, which can add expressiveness to the generated music. **Style transfer** is another relevant technique for music generation, particularly when you want to apply the characteristics of one musical style to another. In the context of Indian Classical music, style transfer could involve taking a composition from another genre, like Western Classical music, and reinterpreting it in the style of Indian Classical music, or vice versa. This is achieved by training models, such as GANs or VAEs, on different musical datasets and having the model learn the stylistic characteristics of each genre. The model can then generate compositions that combine the structural elements of the melody (such as the progression of a rāga) with the stylistic features of the target genre. Finally, evaluating the quality of generated music is a challenging task, as there is no single metric that can comprehensively assess the creativity or musicality of a composition. Some evaluation methods rely on **human perceptual feedback**, where listeners assess the musicality and expressiveness of the generated compositions. This subjective evaluation is often complemented by **quantitative metrics**, such as note overlap (how similar the generated notes are to real compositions), rhythmic accuracy (how well the generated music follows the expected rhythm), and melodic similarity (how closely the generated music follows traditional melodic patterns). Additionally, **loss functions** are used during model training to guide the generation process. For instance, **cross-entropy** can be used for classification tasks, while **mean squared error** can be employed for regression tasks, ensuring that the generated music stays consistent with the desired patterns.

2.2 *Technologies Used*

This section describes the technologies, libraries and frameworks used in this project.

2.2.1 Python

Python's ease of use, readability, and vast library ecosystem have made it one of the most widely used programming languages for machine learning and artificial intelligence (AI) applications. Python is the main programming language utilised in this project for tasks related to music synthesis, such as model construction, data preprocessing, and evaluation. Python is a great option for both novice and seasoned developers because of its clear and simple syntax, which is one of its main advantages. For instance, Python frees developers from worrying about minute matters like memory management or syntax errors, allowing them to concentrate more on solving complicated problems.

Python's broad support for machine learning frameworks is another factor in the decision to use it for this project. The tools required to develop complex deep learning models that can manage big datasets, such as those used in music production, are provided by libraries like TensorFlow, PyTorch, and Keras. Furthermore, Python offers great interaction with numerical libraries like Pandas and NumPy, which are essential for data processing and analysis. Fast numerical operations are made possible by NumPy, particularly when working with arrays and matrices, which are essential for deep learning applications. In the meantime, Pandas provides strong data structures like DataFrames that provide systematic manipulation of big datasets. Beyond machine learning, Python also excels in the domain of audio and music processing. Libraries like Librosa and PrettyMIDI provide specialized tools for analysing and manipulating music files, enabling the extraction of features such as Mel spectrograms, pitch, rhythm, and duration, all of which are essential for the music generation models in this project. Python’s versatility and widespread use in both research and industry make it the ideal language for developing machine learning models and processing musical data.

In the context of the project, Python is used for data preprocessing, model development, and evaluation. The workflow typically involves loading MIDI files, extracting features, feeding these features into machine learning models, and then generating new music. The results are then visualized and analyzed to gauge the performance of the models. Additionally, Python’s support for data visualization libraries such as Matplotlib and Seaborn allows for real-time analysis of model performance during training, such as plotting loss curves and accuracy graphs. The ease of integrating Python with cloud computing environments such

as Google Colab further enhances its utility. Google Colab provides free access to GPU resources, allowing the deep learning models to be trained efficiently without requiring expensive local hardware. This cloud-based approach to computing is invaluable for experimenting with large datasets and computationally expensive models like GANs, VAEs, and transformers.

2.2.2 Tensorflow and Keras

Google created TensorFlow, an open-source machine learning library for creating and refining deep learning models. It is extensively utilised in both industry and research, and projects requiring the creation of complex machine learning models benefit greatly from its scalability and versatility. TensorFlow is a good fit for projects like this one, which use deep learning techniques to generate music, because it can efficiently train on big datasets thanks to its ability to run on multiple GPUs and CPUs[13].

Once a stand-alone library, Keras is now incorporated into TensorFlow as a high-level API. Working with complex models is made easier by its streamlined interface for defining and training neural networks, which eliminates the need to learn the nuances of TensorFlow's lower-level operations. By abstracting away, a large portion of the model building complexity, Keras enables developers to concentrate on creating the networks' architecture rather than handling the minute details [14].

n this project, TensorFlow and Keras are primarily used for building and training the models (Generative Adversarial Networks, and Variational Autoencoders). These models are designed to learn from large datasets of music and generate new musical compositions. GANs, for example, consist of two components: a generator that creates music sequences and a discriminator that evaluates how well the generated sequences mimic real music. The generator and discriminator are trained simultaneously in a competitive process, which results in the generation of high-quality, realistic music. By offering user-friendly layers and models for specifying the network architecture, the Keras API streamlines the process of creating GANs. For example, models can be created using Keras' Sequential API by linearly stacking layers of neurons. Keras is a great option for rapidly developing and experimenting with various neural network topologies because of its simplicity. The library offers tools for monitoring and evaluating models in addition to TensorFlow's deep learning capabilities. For instance, customers can view the training process in real-time with the TensorBoard visualisation tool, which includes measures like accuracy, loss, and other custom metrics. This feature is very helpful for tracking model performance during training and making sure that learning is proceeding as planned.

2.2.3 PrettyMIDI

PrettyMIDI is a Python library that simplifies working with MIDI files. It allows users to easily read, write, and manipulate MIDI data, which is a central component of this project. Since MIDI files contain digital representations of musical compositions, PrettyMIDI is used to extract features such as note sequences, timing, pitch, and duration, which are then fed into deep learning models to generate new musical compositions. The library's main advantage is its ability to handle MIDI files in a high-level, user-friendly way. PrettyMIDI allows developers to focus on the higher-level aspects of music generation, such as modeling musical patterns and structures, without getting bogged down in the low-level details of MIDI file parsing. For example, PrettyMIDI provides functions like *pretty_midi.PrettyMIDI()* to load MIDI files and *pretty_midi.get_piano_roll()* to convert them into piano roll representations. A piano roll is a 2D matrix where each row represents a time step, and each column represents a specific note or pitch. This format is particularly useful for feeding MIDI data into deep learning models.

PrettyMIDI's main function in this music creation project is to transform MIDI data into representations that operate with the neural networks that the models employ. These depictions include note sequences and piano rolls, which both encapsulate the fundamental components of music, including harmony, rhythm, and melody. We can quickly extract these features and enter them into models such as transformers, GANs, and VAEs by utilizing PrettyMIDI. MIDI file manipulation is another area in which PrettyMIDI excels. It enables you to change the pace, add new notes, or alter the pitch or timing of existing notes in MIDI files. These features are be handy while creating original music or enhancing pre-existing files. For instance, it is typical practice in music generation tasks to incorporate variations by adjusting the time or pitch of notes, which aids the model in learning a wider variety of musical patterns.

2.2.4 Variational Autoencoders

Another family of deep generative models that has showed a lot of promise in the creation of music is Variational Autoencoders [16]. They were introduced in 2013 by Kingma, Diederik P, and Max Welling[17]. VAEs are especially well-suited for capturing the latent structure of music data since they are trained using a probabilistic technique, as opposed to GANs, which learn through a competitive process. The encoder and the decoder are the two primary parts of VAEs [18]. The encoder converts the input data—music sequences in this case—into a latent space, a lower-dimensional representation that encapsulates the key characteristics of the data. The original data is subsequently reconstructed by the decoder using this latent form. In addition to making sure the latent space has the desired characteristics, like being continuous and

well-structured, the VAE is trained to reduce the disparity between the original and recreated data. VAEs are used to build a probabilistic model of the music data in music generation. The VAE may create new musical sequences by sampling from the latent space after it has been taught.

class pretty_midi.**PrettyMIDI**(*midi_file=None*, *resolution=220*, *initial_tempo=120.0*)

A container for MIDI data in an easily-manipulable format.

Parameters:

midi_file : *str or file*
Path or file pointer to a MIDI file. Default `None` which means create an empty class with the supplied values for resolution and initial tempo.

resolution : *int*
Resolution of the MIDI data, when no file is provided.

initial_tempo : *float*
Initial tempo for the MIDI data, when no file is provided.

Attributes:

instruments : *list*
List of `pretty_midi.Instrument` objects.

key_signature_changes : *list*
List of `pretty_midi.KeySignature` objects.

time_signature_changes : *list*
List of `pretty_midi.TimeSignature` objects.

lyrics : *list*
List of `pretty_midi.Lyric` objects.

text_events : *list*
List of `pretty_midi.Text` objects.

Figure 10 : Sample PrettyMIDI Function [15]

They have the advantage of facilitating seamless interpolation between various musical compositions, which makes them perfect for activities such as creating rāga variations or experimenting with novel note and rhythm combinations. Because VAEs can model the intricate, highly structured nature of Indian classical music while preserving a flexible and interpretable latent space, they are very helpful in this context. The VAE may produce music that complies with the artistic requirements of Indian classical

compositions by conditioning it on particular rāga or tala characteristics. Furthermore, because the VAE is probabilistic, it can generate a variety of unique outputs, which is crucial for the generative task.

The primary drawback of VAEs is that, because they typically concentrate more on data reconstruction than on producing really original compositions, the resulting music may occasionally lack the intricacy or emotional depth of live performances. This can be lessened, though, by enhancing the quality of the created music by mixing VAEs with other models, including GANs.

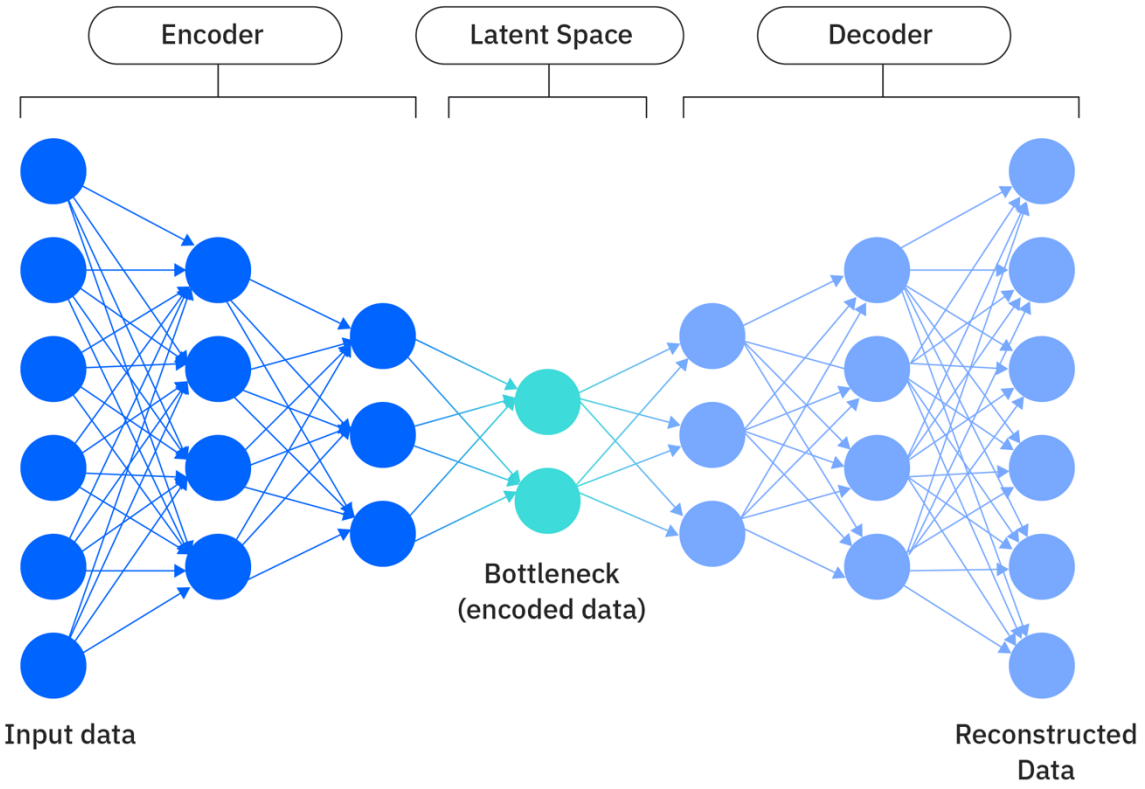


Figure 11 : Variational Autoencoder Structure [19]

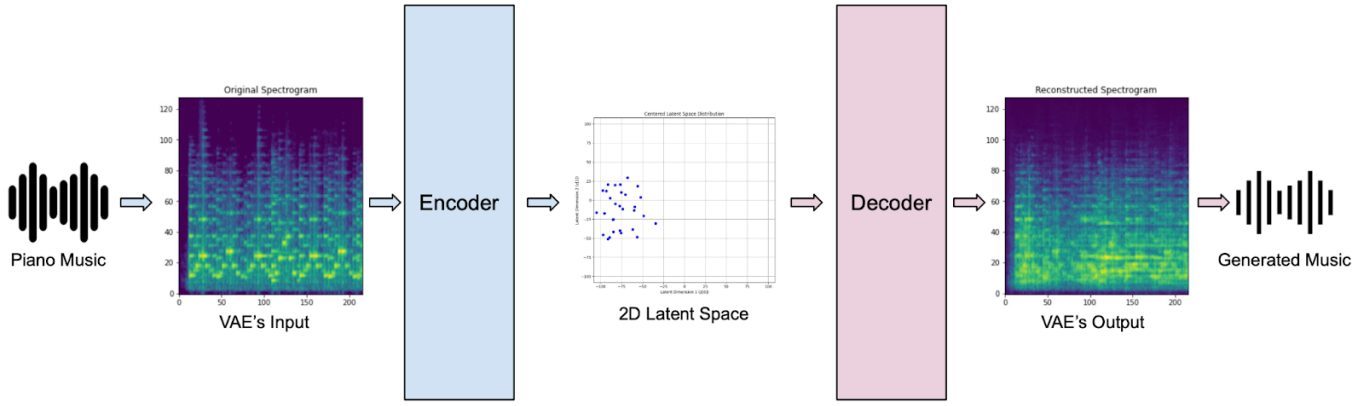


Figure 12 : Sample usage of VAE in music generation [20]

2.2.5 Generative Adversarial Networks

Since their introduction by Ian Goodfellow et al. in 2014 [21], Generative Adversarial Networks, or GANs, have become a groundbreaking architecture in the field of generative modelling. The discriminator and generator neural networks, which make up a GAN, are trained concurrently in a competitive environment. While the discriminator's task is to differentiate between real samples (from the training set) and fake samples (generated by the generator), the generator's objective is to learn the underlying data distribution and make new data instances that mimic the original data [23]. Over time, the generator gets better thanks to this adversarial training, eventually generating outputs that are incredibly lifelike. GANs were used to create MIDI sequences that replicate the tonal and rhythmic qualities of particular rāgas as part of our effort on genre-specific music generation using Indian classical music as a foundation. Because symbolic representations like MIDI are small and structurally clear, we decided to use them instead of the more common audio-based models. By treating the generation task as a discrete sequence prediction issue, akin to text generation or image creation using pixel grids, the symbolic format simplified the implementation of GANs. The model's ability to produce a variety of outputs, which enabled the investigation of several stylistic interpretations of a single rāga, was a significant benefit of employing GANs in this situation. One drawback, though, was that assessing musical outputs necessitates subjective listening or musicological analysis, in contrast to photographs, where visual artefacts are simple to identify.

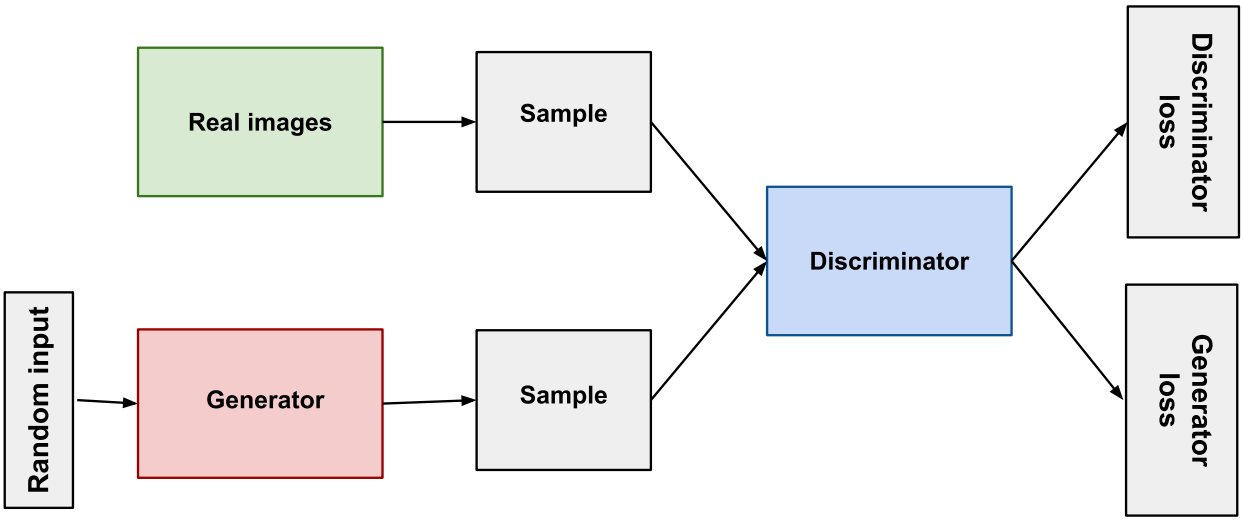


Figure 13 : Generative Adversarial Networks[22]

2.2.6 Transformers

Since their introduction by Vaswani et al. in 2017 [24], Transformers have significantly changed the field of sequence modelling in artificial intelligence, especially in areas like picture generation, natural language processing, and most recently, the creation of symbolic music. The Transformer architecture uses self-attention mechanisms to model interactions between elements in a sequence, regardless of their distance from one another, in contrast to conventional models that rely on recurrence (such as RNNs or LSTMs). Better long-range dependency learning is made possible by this approach, which also makes use of parallel processing to enable much faster training [25].

Transformers are essential to our project's modelling of the highly organised and temporally complex sequences present in MIDI data. The complex note progressions, decorative variations (like gamakas), and cyclic rhythmic patterns (like taala) found in Indian classical works necessitate a model that can manage both local transitions and global musical context. Because transformers can handle numerous time steps at once and weigh them according to relevance—a capability that classic recurrent models frequently lack—they are particularly well-suited for this purpose.

Transformers' capacity to produce cohesive, multi-minute compositions that maintain the rāga identity throughout the entire piece is one of its significant advantages in this field. The Transformer's global attention mechanism guarantees that the musical phrases it produces adhere to the scale, modal patterns,

and time cycles typical of the target genre, in contrast to GANs or VAEs, which might concentrate on producing shorter musical samples or have trouble with long-term coherence. When it came to preserving

the rāga's Arohana (ascending) and Avarohana (descending) logic and encapsulating the unique rhythmic pulse of traditional taals, the Transformer continuously outperformed other models in our tests.

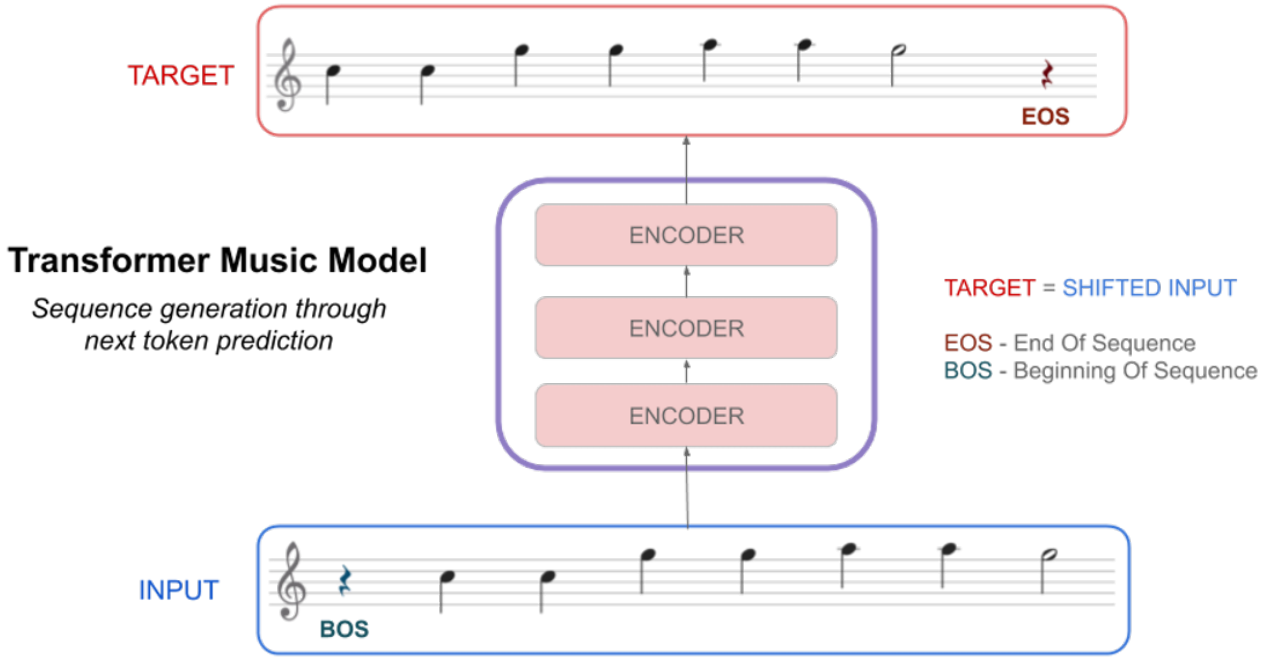


Figure 14 : Next Token Prediction Based Sequence Generation in Transformers [26]

Methodology

3.1 Dataset Preparation

Building a high-quality, structured dataset that covered a wide variety of musical patterns, instrumentation, and dynamics was essential to successfully training and assessing generative models for symbolic music synthesis. We decided to manually create a dataset from scratch in order to maintain complete control over the musical content, temporal organisation, and artistic consistency because there were no publicly accessible symbolic datasets that were appropriate for our objective.

The first step in the data collection procedure was selecting videos of performances from publicly accessible websites like YouTube and archives of classical music performances. To preserve variability in phrasing, tempo, and dynamics, a broad range of genres, composers, and performing styles were purposefully chosen.

The following were the main selection criteria:

- Clear expression of melodic and harmonic content.
- Stable camera angles and unbroken audio.
- Solo piano performances with little background noise.

Audacity, a digital audio editor, was used to extract and treat the audio from these videos in order to normalise the sound levels and isolate the recordings. In few instances, cutting and mild noise reduction were also used to eliminate extraneous parts and maintain distinct note beginnings. After that, manual MIDI transcription was done using the cleaned audio that had been loaded into FL Studio, a professional digital audio workstation. Every note in the concert was transcribed into MIDI format using a combination of auditory matching and piano roll visualisation. Pitch, duration, and velocity may be adjusted considerably more precisely with this manual method than with automatic audio-to-MIDI conversion programs, which frequently have trouble with polyphonic textures and expressive timing.

Each performance was transcribed and then divided into 30-second MIDI snippets. This length was intended to balance capturing entire musical phrases with keeping training sequences manageable. The longer performances were divided into several parts, and clips that were too brief or lacked a clear framework were eliminated. Where required, a set tempo of 90 BPM was used to preserve a constant temporal resolution across all samples. Sequence comparison was facilitated by this normalisation, which also decreased variability brought on by expressive timing.

Following segmentation, the pretty_midi package was used to interpret and convert the MIDI files into numerical representations. Pitch, velocity, duration, and delta time are the four fundamental characteristics of each note vector that was created from each MIDI file. The MIDI note number was used to encode pitch, while the note's intensity was recorded by velocity. Delta time was the amount of time between the beginning of one note and the next, while duration was the amount of time spent holding each note. Each training sample had 32 notes, each of which was represented by a 4-dimensional vector. These features were normalised and kept in a fixed shape of (32, 4). Longer sequences were trimmed to this preset size, while shorter sequences were zero-padded. Model-specific preprocessing will be discussed in the following sections.

3.2 Workflows

3.2.1 Overall Workflow

This project's overall workflow includes every step of the pipeline, from the first collection of unprocessed performance data to the last assessment of the produced musical outputs. In order to convert expressive piano performances from the real world into structured symbolic representations, train models using this data, and ultimately create new music through generative processes, each step in this pipeline is essential. Each step depends on the meticulous execution of the one before it, and they are consecutive.

The first step is gathering performance footage, which serve as the project's source material. This has been discussed in the previous section (3.1). The main selection criteria were clean instrumental focus (i.e., solo, without accompaniment), emotive musical interpretation, and sonic quality. These recordings, which are frequently concert snippets or in-studio performances, offer the perfect basis for researching and creating human-like musical structures since they maintain the acoustics and organic expressiveness of a live performance.

To get ready for modelling, the raw MIDI files go through a preprocessing step after they are created. To keep a consistent dataset that is exclusively focused, unnecessary components like percussion tracks or non-instrument data are eliminated in this step. To guarantee consistency throughout the dataset, different musical features are normalised after each MIDI file is parsed to extract only the note events. For instance, note velocity data are scaled to a normalised range of 0 to 1, although in regular MIDI they range from 0 to

127. To lessen variability brought on by timing anomalies and performance expression, durations and inter-onset intervals (delta times) are also standardised. The dataset's computational consistency and musical significance are guaranteed by this preprocessing.

Each note is encoded as a vector of four continuous values: pitch, velocity, duration, and delta time. This process converts the cleaned and normalised MIDI data into a note vector representation. To maintain the original music's flow, these vectors are placed successively in temporal order. Each sequence is formatted as a matrix of shape (32, 4), indicating 32 notes with 4 attributes each, because the model assumes a set sequence length for every performance clip. Longer sequences are trimmed to meet the needed size, whereas sequences with fewer than 32 notes are padded with zeros. As a result, a consistent and organised dataset, usually of shape (N, 32, 4), with N being the number of sequences, is prepared for training.

Model training is the next step when the dataset is ready. The sequences are processed through the appropriate network based on the architecture being considered, whether it is a Transformer, VAE, or GAN. In each scenario, the model minimises a loss function unique to the model type in order to learn to recreate, produce, or forecast musical sequences. The adversarial training for GANs is driven by a generator-discriminator setup, the latent space learning for VAEs is governed by a reconstruction loss plus KL divergence, and the sequence modelling for Transformers is directed by a categorical cross-entropy loss over the tokenised note sequences. To track convergence and model behaviour during training, visualisations such loss curves are produced at the end of each epoch.

The models move on to the music sequence creation stage after training is finished. Random samples are taken from the learnt latent space (or sampled latent vector z) for latent-space-based models such as GANs and VAEs. These samples are subsequently decoded into note vectors. New sequences are produced autoregressively for the Transformer by employing sampling techniques such top-k sampling, nucleus sampling, and greedy decoding. The original dataset's vector format is used to arrange these created sequences. pretty_midi was then used to transform the note vectors back into MIDI format after they have been created. In this stage, pitch, duration, and other characteristics are mapped back to conventional MIDI note events, reversing the vectorisation process. These MIDI files now contain whole new musical sequences that are produced by machines and can be listened to, examined, or assessed.

The generated music is evaluated and tested as the last step in the workflow. Both qualitative and quantitative methods are used for this. Piano rolls, which offer a graphical depiction of the note distribution, time, and density, are used to visualise the outputs. In order to evaluate their musicality, coherence, and innovation, they are also auditorily examined. Furthermore, fidelity and divergence are assessed by comparing generated outputs with real (training) samples.

3.2.2 GAN Workflow

One of the main approaches utilised in this study for the creation of symbolic music is the GAN workflow. GANs work by using adversarial training between two rival neural networks, the Generator and the Discriminator, as opposed to autoregressive models that depend on next-step prediction. While the discriminator learns to discriminate between fake and real musical sequences, the generator is supposed to be able to learn the underlying distribution of real musical sequences so that it may create new sequences that are indistinguishable from real data.

A MIDI folder, which has a sizable collection of symbolic music data in.mid format, is where the procedure starts. The transcription produced these MIDI files, which are brief musical samples that form the basis of instruction. Every file records a single performance segment and includes all the note-level data required to transform it into vectors that machines can understand.

Each MIDI file is run through a MIDI to Note Vector conversion module, which is implemented using the pretty_midi Python package, in order to prepare this data for the GAN architecture. Pitch (frequency), velocity (loudness), duration (Time), and delta time (time that has passed since the previous note) are the structural properties that are extracted from each MIDI event by this conversion. A four-element vector is used to represent each note event, and these vectors are arranged in temporal order to generate sequences.

A dataset of note sequences is produced after processing all MIDI files. Each musical sample is encoded as a fixed-length sequence of 32 notes, with four continuous characteristics describing each note. When N is the number of MIDI clips in the corpus, the resulting dataset has the shape (N, 32, 4). Sequences larger than the cutoff are terminated, and padding is provided to those shorter than 32 notes to preserve consistent dimensionality. This guarantees that every sequence is the same size, which makes it possible for them to go through the network effectively.

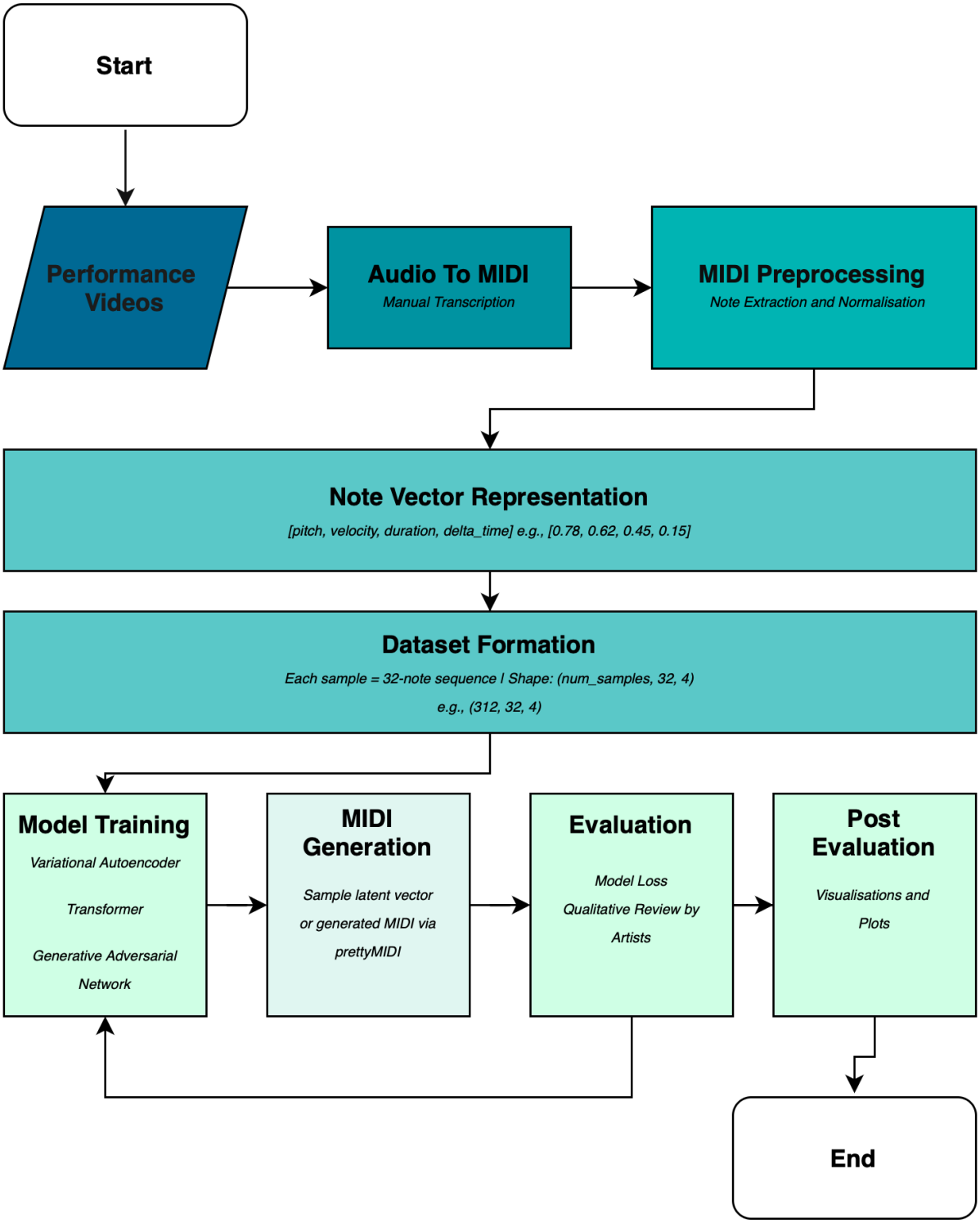


Figure 15 : Overall Workflow

The discriminator and generator neural networks make up the GAN's architecture. A latent vector of size 100 that is selected from a normal distribution is fed into the generator. It mimics the structure of the actual note sequences by passing this input through numerous layers and reshaping the output into a (32 x 4) format. The output values are guaranteed to stay within a limited range by a sigmoid activation at the end. On the other hand, either authentic note sequences from the dataset or fake ones produced by the generator are sent to the discriminator. After processing the input through a series of thick layers and flattening it, it ends with a single sigmoid neuron that produces a probability indicating whether the input sequence is created or real.

Adversarial learning is the foundation of the training process. A batch of actual note sequences is chosen at random from the dataset for each iteration. The generator simultaneously samples random noise to create a batch of fictitious sequences. The discriminator is then run through both batches. Because the generator is designed to trick the discriminator, its loss is calculated by calculating how well it can produce sequences that the discriminator takes for granted. On the other hand, a binary cross-entropy loss is used to train the discriminator to accurately distinguish between real and fake. TensorFlow's gradient tapes are used to compute and apply gradients appropriately, and Adam optimizer is used for both networks.

Generator and discriminator losses are monitored at each stage of this adversarial process, which lasts for 1000 epochs. The generator's output is periodically shown as a colour-coded feature matrix, with the y-axis signifying each of the four note features and the x-axis representing time steps, in order to qualitatively track the progress. The structure and variety of the sequences being produced are revealed by these visualisations. The cycle from raw MIDI input to freshly synthesised music can then be completed by converting these generated note vectors back into MIDI files for auditory evaluation and comparison with actual sequences after training.

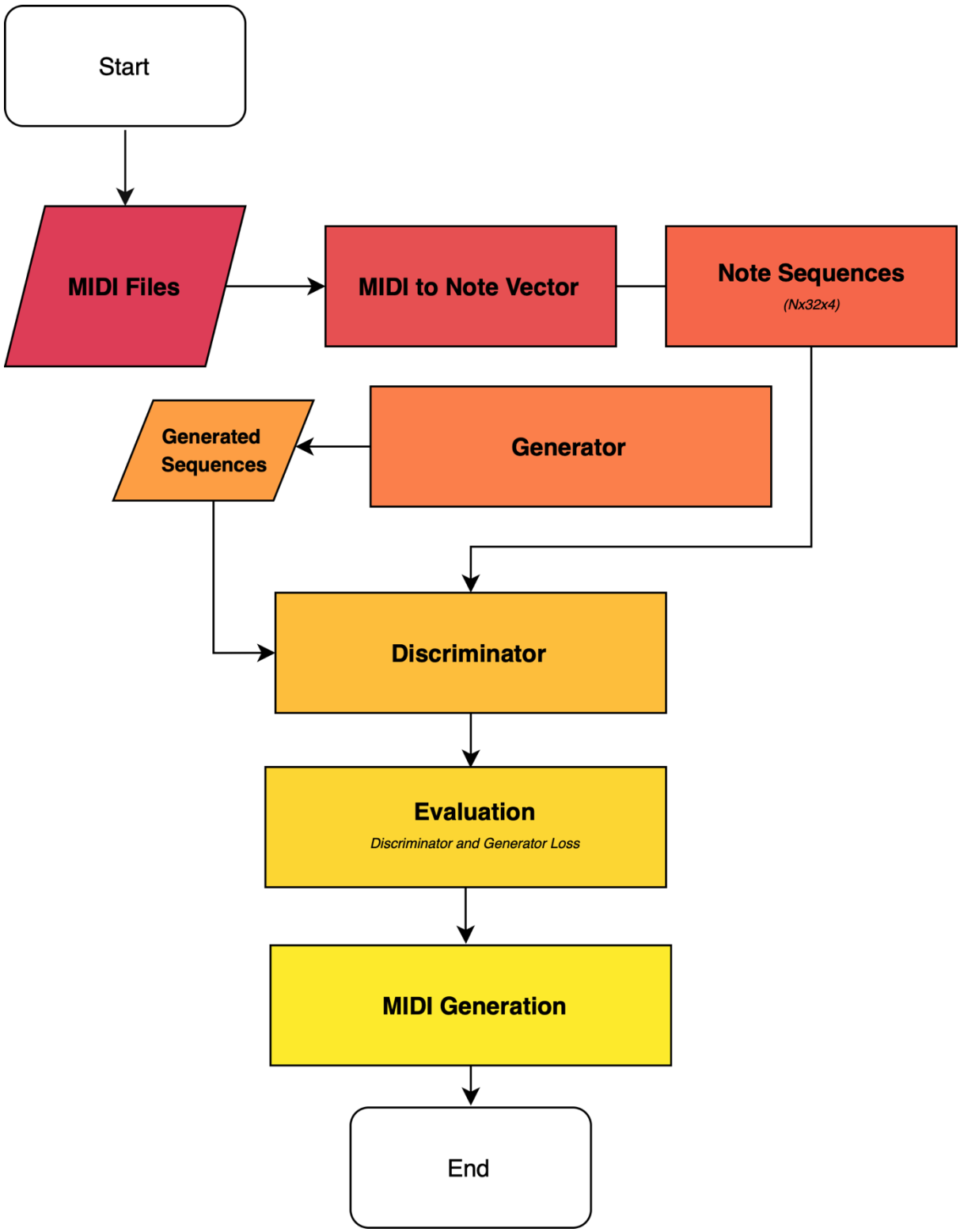


Figure 16 : GAN Workflow

3.2.3 Transformer Workflow

Each note vector in the Transformer-based music generating pipeline is first enhanced with a delta time, or the interval of time between the present and subsequent notes. After that, the notes are padded or cut to form sequences of a predetermined length, in this case 32 time steps. Sequences are only included in the dataset if they are able to achieve this length. When N is the number of MIDI samples, 32 is the length of the sequence, and 4 is the number of note features, the resulting dataset has a consistent form (N x 32 x 4).

A Transformer model is trained using this dataset. An embedding layer, which converts the 4-dimensional note vectors into a higher-dimensional space (such as 128 dimensions), is applied to each input sequence first. A positional encoding layer is added to make sure the model knows where each note in the sequence is located. In order to represent relative and absolute location information in a manner that the model can learn, this layer uses sine and cosine functions at different frequencies.

Several Transformer blocks make up the model's core. The model can learn correlations between notes at various time steps thanks to the multi-head self-attention layers in each block. The representations are then transformed using feed-forward networks and layer normalisation. Residual connections promote gradient flow and aid in training stabilisation.

A softmax layer transfers the high-dimensional embeddings back to the original 4-note feature dimensions after the Transformer's output passes through a linear layer. Next-step prediction is used for training: the model is trained to predict the sequence from steps 1 to 31 after receiving the sequence from time steps 0 to 30 as input. Batches of data are input into the model at random during training. TensorFlow's GradientTape is used to calculate gradients for each batch, and the Adam optimiser is used to update model parameters. To track progress, the training loop is run for a predetermined number of epochs (for example, 100), during which the loss is tracked and plotted.

New note sequences can be produced by the model after it has been trained. It begins by making a step-by-step prediction of the subsequent notes based on an initial sequence (or sampled noise). Pretty_midi is used to convert the output vectors back to MIDI format so that music can be played. To see how well the model has learnt musical structure, the predictions can also be shown as feature plots that contrast the generated sequence with the original.

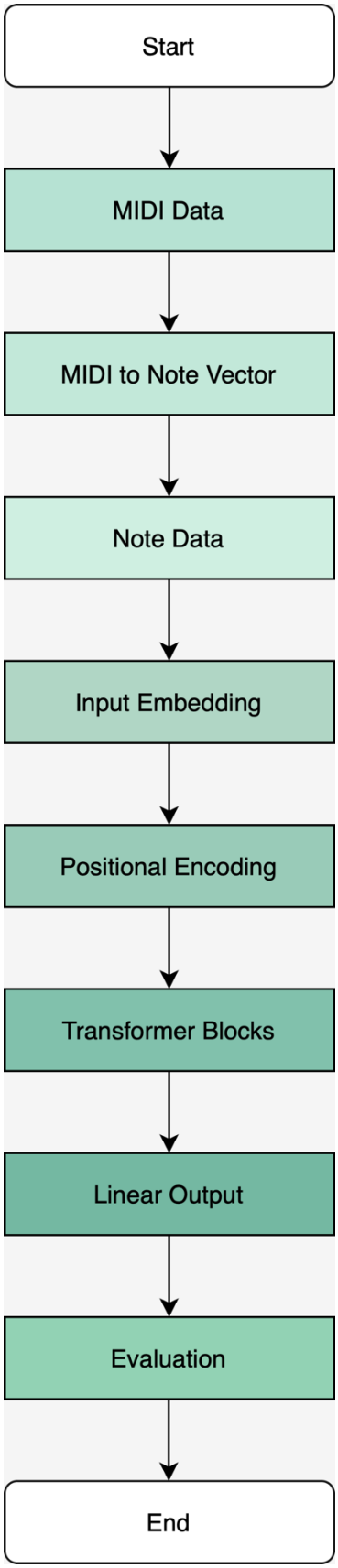


Figure 17: Transformer Workflow

Modules and Prototype

4.1 Data Loading Module

In order to provide a dataset appropriate for machine learning applications, this module must transform MIDI files into binary piano roll matrices. Please note that code for this module can be found in Annexture 1.

MIDI files are stored in a directory structure that may be navigated using the OS module. Numerical operations and array manipulation require the import of numpy. Tools for parsing and working with MIDI files are available in the pretty_midi package. Despite being imported as well, pypianoroll is not utilised in this particular code segment. The function midi_to_pianoroll(filepath, fs=100) accepts a frame rate (fs, which is set to 100 frames per second by default) and a file path to a MIDI file. It uses the file path to construct a PrettyMIDI object inside a try block. Access to the MIDI file's musical content is made possible by this object. A 2D NumPy array with rows representing MIDI pitches (from 0 to 127) and columns representing time steps sampled at the specified frequency is then returned by the function's call to get_piano_roll(fs=fs). A velocity value that represents a note's intensity is contained in each cell. Next, (piano_roll > 0) is used to threshold this velocity matrix into a binary representation. astype(np.uint8), designating 0 for inactive notes and 1 for active notes. The function securely returns None if the MIDI file is invalid or cannot be parsed.

The load_dataset(base_path, seconds=15, fs=100) method may handle a whole directory of MIDI files that have been arranged by composer. In order to store piano roll matrices, it initialises an empty list X. Based on the required number of seconds and the sampling rate (fs), the variable target_len determines the set number of time steps (frames) that each piano roll should have. Assuming that each subdirectory in the base path is named after a composer and includes MIDI files, the function iterates over each one. It creates the complete path for each file that ends in.mid and uses midi_to_pianoroll to try to turn it into a piano roll. The function slices the piano roll to retain only the first target_len columns and appends it to the dataset list if the result is not None and the piano roll's time dimension is at least as long as target_len. Once all MIDI files have been processed, it uses np.stack(X) to stack the gathered piano rolls into a single 3D NumPy array and returns it.

4.2 Variational Autoencoder Module

This section focuses on the training procedure of the Variational Autoencoder (model using the piano roll dataset prepared earlier). Code for this module can be found in annexture 2.

Batches of data are effectively managed during training using the DataLoader and TensorDataset classes from torch.utils.data. To see training loss over epochs, import matplotlib.pyplot. Using torch.tensor(X, dtype=torch.float32), the piano roll dataset X—which was first constructed as a NumPy array—is transformed into a PyTorch tensor of type float32. PyTorch can then handle this tensor as a structured dataset since it is wrapped in a TensorDataset. This dataset is iterated over using the DataLoader in 16-size mini-batches, with shuffling turned on to guarantee unpredictability throughout training.

The VAE model instance is then made and transferred to the GPU for accelerated computing using.cuda(). Adam is the optimiser that was chosen for training; it was started using the parameters of the VAE and a learning rate of 1 \times 10^{-4}. The total loss for every training epoch is initially stored in an empty list called losses.

There are 20 epochs in the training procedure. The total_loss variable is used to record a running total of the loss for each epoch. The dataloader iterates through the dataset in batches within the epoch loop. A tensor x is transferred to the GPU in each batch. Optimiser is used to zero the optimizer's gradients. To avoid accumulation from earlier iterations, use zero_grad().

After processing the input, the VAE model yields the mean mu, the log-variance logvar of the latent distribution, and the reconstructed output recon. The vae_loss function is used to calculate the loss, and it usually contains a Kullback-Leibler divergence term (which encourages the latent space to follow a conventional normal distribution) and a reconstruction loss (which measures how well the output matches the input).

Loss.backward() computes gradients by backpropagating after calculating the loss. To prevent gradients from exploding, these gradients are then clipped using torch.nn.utils.clip_grad_norm_() to a maximum norm of 1.0. Optimizer.step() is then used to update the model parameters. To track epoch performance, the batch loss is added up to total_loss.

The total loss is printed at the conclusion of each epoch and added to the list of losses. The model's convergence over time is visualised by using matplotlib to plot a loss curve after training is complete. A clear image of training dynamics and model stability is provided by the y-axis, which displays the total loss, and the x-axis, which depicts the epoch number.

4.3 Transformer Module

Code for the transformer module can be found in annexture 3.

Important model configuration constants including the batch size, number of training epochs, number of note features (FEATURE_DIM = 4), and sequence length (SEQ_LEN = 32) are defined first. Four attention heads (NUM_HEADS = 4) and two Transformer layers (NUM_LAYERS = 2) are used in the model's construction, along with an embedding size of 128.

A unique PositionalEncoding layer is the first component of the model design. In order to allow the model to reason about temporal structure, this layer uses sinusoidal functions to inject information about the order of notes in the sequence, as Transformers lack recurrence or convolution.

A typical Transformer encoder unit is defined by the transformer_block and consists of a feed-forward subnetwork, residual connections, multi-head self-attention, and layer normalisation. To create a deeper architecture, we use the build_transformer method to stack two of these blocks. Prior to being mapped back to the original feature dimension, inputs are first projected into an embedding space, then enhanced with positional encoding and then transmitted via the Transformer layers.

The train_transformer function utilises Adam as the optimiser and Mean Squared Error as the loss function to train the model. A random mini-batch of sequences is taken from the dataset for every training period. The objective is the identical sequence pushed one step forward, and the input is all but the final time step of each sequence. In this manner, based on previous context, the model learns to anticipate the subsequent set of note attributes.

We can keep an eye on convergence by using plot_losses to track and plot training loss over time. The current loss is printed to the console every ten epochs. We visually compare the model's predicted note attributes with the real ones following training. This function illustrates how effectively the model can capture musical structure by plotting the four features—pitch, velocity, duration, and time gap—across the sequence timeline.

4.4 Generative Adversarial Module

Code for this module can be found in annexture 4.

A random noise vector is fed into the generator model, which converts it into a series of note vectors. To limit outputs within a tolerable range, it comprises of a few thick layers, reshaping procedures, and a final sigmoid activation. The goal of this sequence is to replicate actual MIDI note sequences.

Either real sequences (from the dataset) or fake sequences (generated by the generator) are fed into the discriminator model. After processing them through layers that are fully connected, it produces a single number that indicates the likelihood that the input sequence is real.

Real data from the dataset is sampled during the training loop, and random noise is used to create fake data. The generator is trained to trick the discriminator into considering created sequences to be real, while the discriminator is trained to accurately distinguish between actual and fake sequences. Both models' losses are monitored throughout training, and a colour-coded heatmap visualisation of a generated sequence is shown every 100 epochs to aid in understanding how features change over time.

Results and Analysis

The performance analysis and evaluation of each model created during the study are presented in this part.

5.1 Variational Autoencoder

The Variational Autoencoder was the first model evaluated.

The loss progression revealed that the model's loss decreased steadily and quickly over the course of the 20 epochs, from an initial value of 38.22 to 1.72. With the VAE acquiring a condensed and significant latent representation of the musical data, this implies steady training and successful convergence.

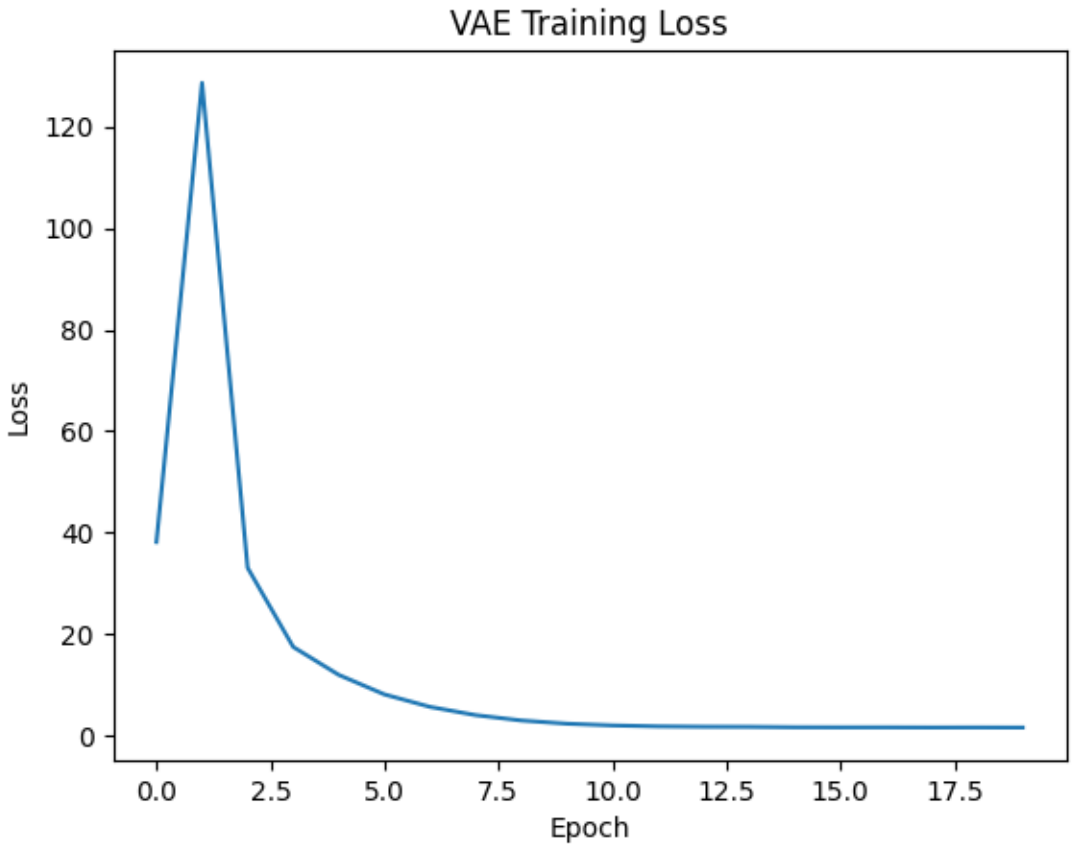


Figure 18 : VAE Loss for Learning Rate 1^-4

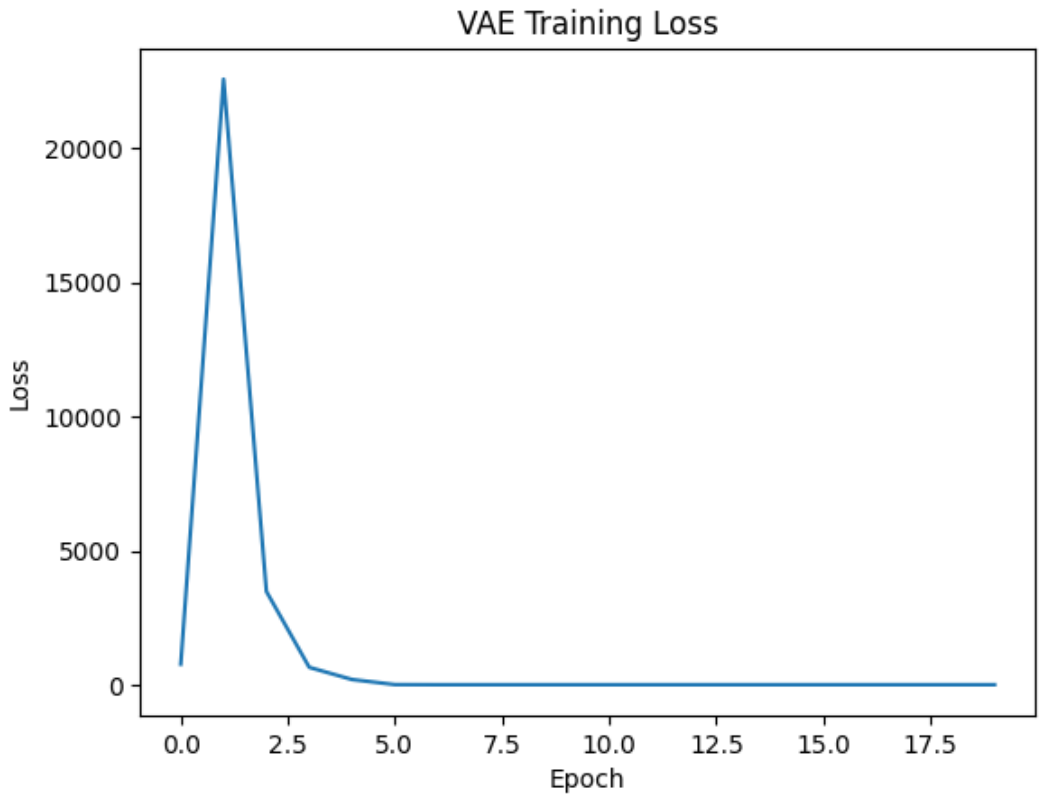


Figure 19 : VAE Loss for Learning Rate 1^-3

It is observed that the model converged better at a higher ADAM learning rate.

In terms of quality, the VAE-generated MIDI sequences showed consistent structure and seamless temporal transitions, especially in the patterns of pitch and velocity. The latent space regularisation of the VAE, which promotes the model to generate musically coherent variations and enforces continuity, is directly responsible for this. The VAE produced consistent output quality across various latent samples, in contrast to GANs, which frequently suffer from mode collapse or instability during training. The outputs of the VAE seemed more consistent but less varied than those of the GAN. In certain situations, GANs may generate outputs that are crisper or more imaginative, but they are also prone to errors like sudden pitch jumps or erratic timing. In contrast, the VAE produced musically believable and predictable sequences, albeit occasionally lacking the stylistic diversity found in GAN-generated music.

Compared to the Transformer-based model, the VAE's strength lies in its simplicity and robustness. The Transformer excels in capturing long-range dependencies and complex musical motifs, leading to more intricate compositions. However, the VAE remains computationally efficient and effective for shorter sequences, especially in scenarios where training resources or data diversity are limited.

Criteria	Observed Merits	Observed Limitations
Model Training	Stable training and convergence	Can lead to overly simplistic Audio which is over regualrised.
Efficiency	Lightweight and Fast to train	Limited complexity
Output Quality	Coherent and harmonically stable note sequences	Lack of variety as compared to GAN
Musical Structure	Maintains short term melodies well	Struggles with long melodies

Table 5 : Summary of Variational Autoencoder Merits and Demerits

5.1 GAN

Over the course of the 1000-epoch cycle, the GAN showed a dynamic but slightly fluctuating training process. By Epoch 900, generator and discriminator losses had progressively shifted towards more balanced values, indicating a degree of adversarial stability in later stages. Initially, they began at different levels (Gen Loss = 0.5775, Disc Loss ≈ 1.4512). A broad lower trend in discriminator loss indicates that the generator learnt to generate sequences that increasingly tricked the discriminator, notwithstanding some noise in the loss path.

When it came to audio quality, the GAN generated outputs that were noticeably expressive and varied, frequently displaying more nuanced rhythm, note velocity, and timing fluctuations than the VAE. This added realism and musicality, but it also occasionally caused instability or structural irregularities, particularly in lengthy scenes. Certain outputs have sudden changes or pauses in musical logic, which is typical of GANs when the discriminator-generator balance varies during training. Because of its adversarial training regime, the GAN was able to produce more stylistic variety and less repetitive wording than the VAE. Lower structural coherence was the price paid for this, though. Though it fared better in terms of originality and variability in shorter sequences, the GAN was unable to handle long-range dependencies and temporal alignment when compared to the Transformer (described below).

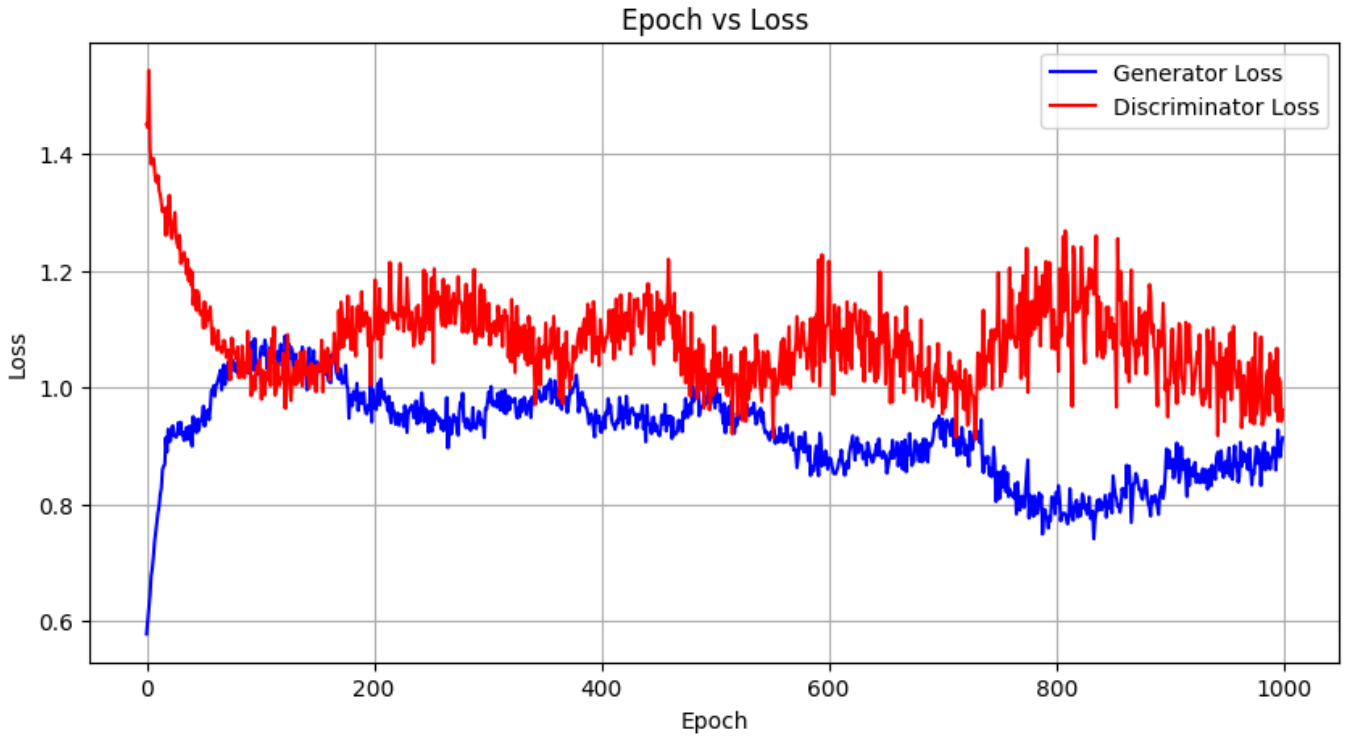


Figure 20 : GAN Discriminator and Generator Losses vs Epochs

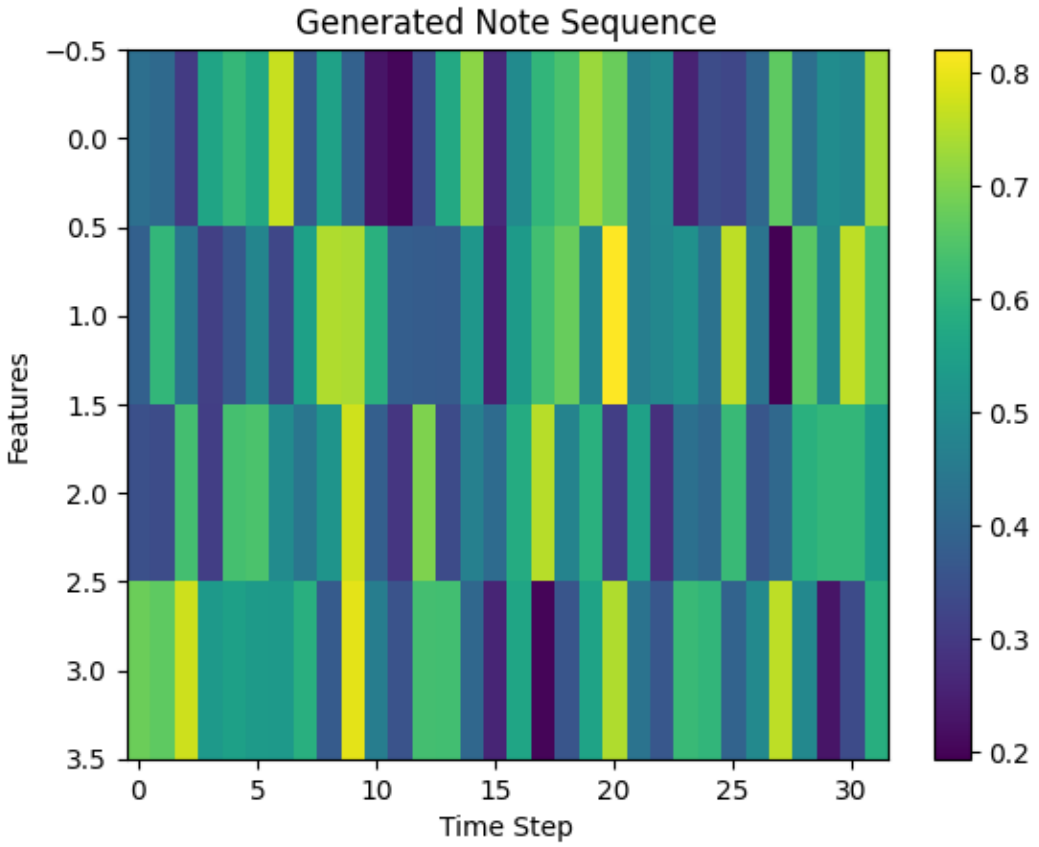


Figure 21 : Generated Note Heatmap for 0th Epoch

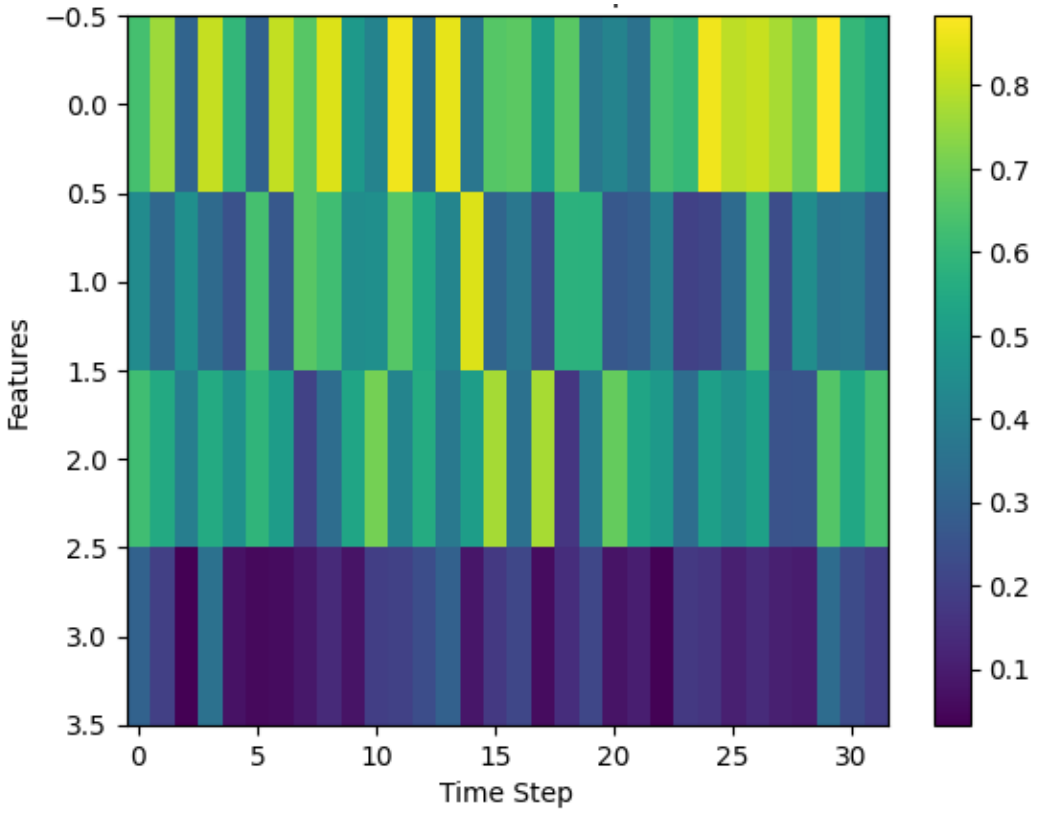


Figure 22 : Generated Note Heatmap for 900th Epoch

Note : Remaining heatmaps (Epoch 100-800) are in annexture 5

Criteria	Observed Merits	Observed Limitations
Model Training	Learns expressive patterns	Collapses fast
Diversity	Generates Diverse and Fast sequences	Sacrifices structure at times
Output Quality	Produces lifelike and unpredictable note changes	Tracks have inconsistent phrasing

Table 6 : Summary of GAN Merits and Demerits

5.1 Transformer

With training loss dropping dramatically from an initial value of 4.7083 to a minimum of 0.0418 by epoch 70, the Transformer-based model showed a high capacity to learn temporal dependencies in MIDI sequences. The Transformer performed exceptionally well at identifying longer-range patterns and structural coherence in the produced sequences, in contrast to the VAE and GAN models. This is an inherent benefit of its self-attention mechanism, which enables contextual connection learning without the need for convolution or recurrence.

In terms of quality, the Transformer model produced audio that was rich in harmonics and rhythmically steady. In contrast to the frequently abrupt or repetitive outputs of the GAN, phrases demonstrated smoother note transitions and improved retention of musical patterns. The VAE occasionally produced outputs that lacked theme development, even when its sequences were musically credible. On the other hand, throughout time, the Transformer continuously produced musically captivating outputs that were structured and varied. Overfitting was a significant issue with the Transformer; around epoch 70, the loss started to fluctuate, suggesting a reduction in generalisation. This behaviour implies that in order to sustain performance over extended training periods, the model could need regularisation techniques like dropout, scheduled sampling, or a bigger and more varied dataset.

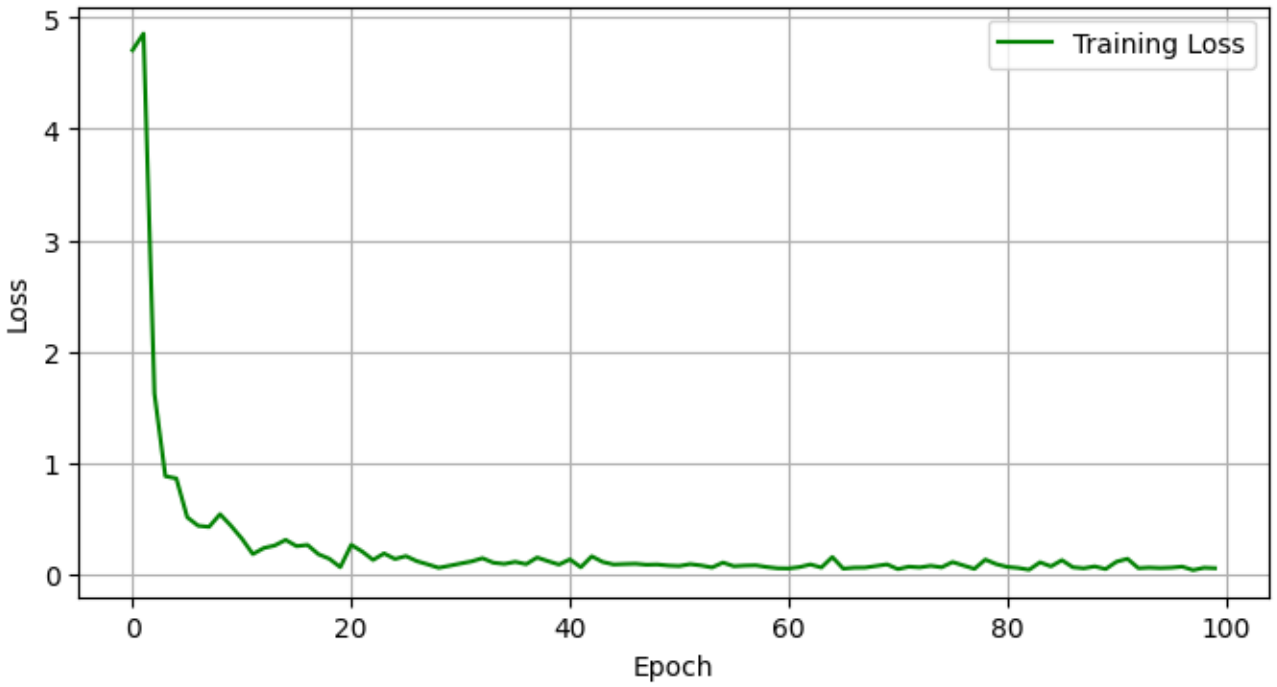


Figure 23 : Transformer Loss

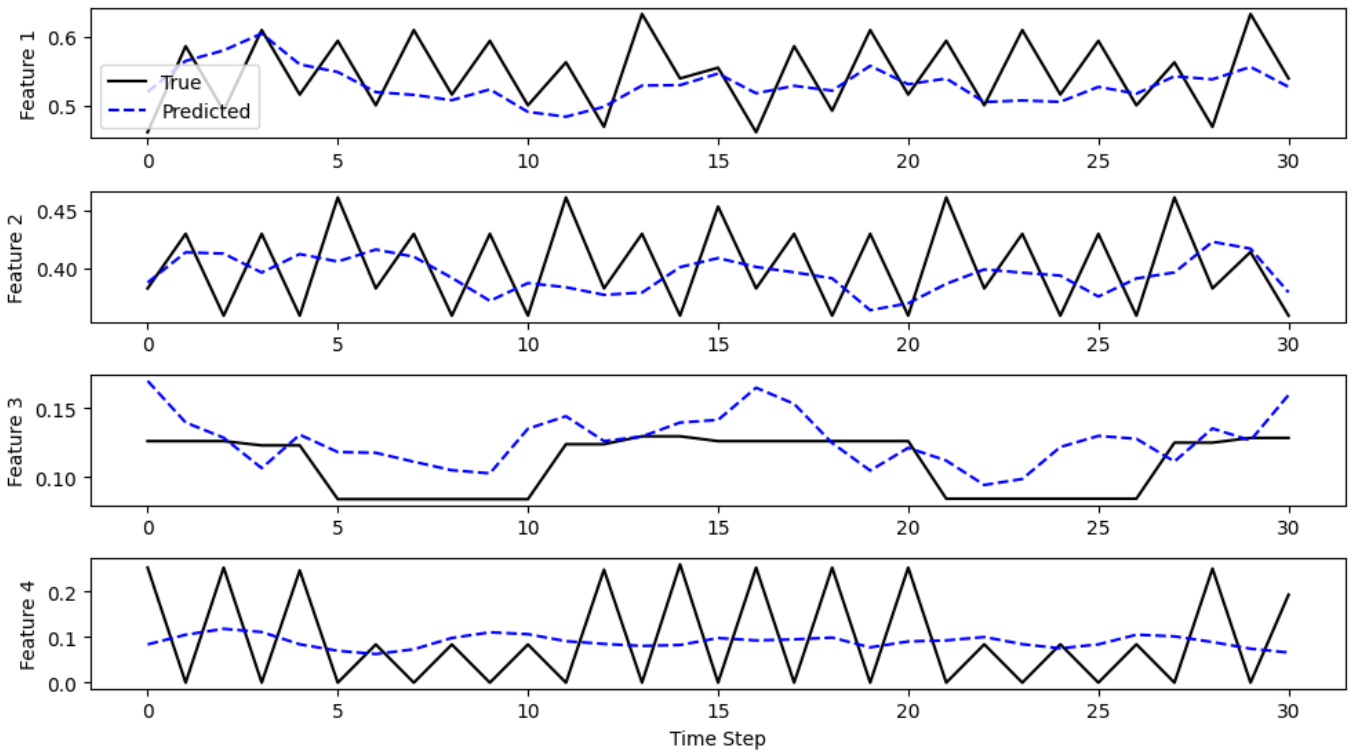


Figure 24 : True vs Predicted Note Features in Transformer

Criteria	Observed Merits	Observed Limitations
Model Training	Consistently decreasing loss ie good convergence	divergence after 70 epochs
Diversity	Coherent, Structured sequences	Prone to overfitting
Output Quality	Musically rich notes with emotional motifs	Slower inference as compared to GAN or VAE

Table 7 : Summary of Transformer Merits and Demerits

Conclusions and Future Work

6.1 Conclusions

This study examined three generative models for creating symbolic music from MIDI data: Transformer, Generative Adversarial Network, and Variational Autoencoder. Each model's learning behaviour, output quality, and capacity to recognise melodic and structural patterns were evaluated.

The VAE exhibited smooth and stable training behavior, with loss values converging steadily by the 15th epoch. Its outputs lacked dramatic expressiveness and diversity, although they did exhibit basic musical coherence, such as appropriate note spacing and rhythm. This restriction is related to the latent space's regularisation effect, which can reduce artistic deviations. For situations where dependable, consistent generation is more important than artistic diversity, the VAE is nevertheless a strong and interpretable model.

More musically varied and occasionally original music was produced by the GAN. The discriminator's performance varied, but the generator obtained a respectable convergence despite the adversarial nature of training necessitating careful tweaking. Sequences produced by GANs were more expressive because they displayed greater intensity and rhythmic diversity. However, training instability also resulted in sudden shifts or rambling sentences, highlighting the trade-off between control and inventiveness.

Overall, the Transformer model produced the most musically cohesive outputs with the least amount of loss. Its ability to represent long-term dependencies gives it the capability to preserve artistic structure, thematic development, and rhythmic consistency. Nevertheless, this resulted in higher computational expenses and some indications of overfitting over time. In spite of this, the Transformer was the most promising contender for advanced symbolic music generation, constantly producing music that felt structured and intentional.

To summarise the merits of each model :

- **VAE**: Reliable structure and stable training.
- **GAN**: Creative and dynamic outputs.
- **Transformer**: Superior coherence and musical understanding.

6.2 Future Work

Even though this study shed light on the characteristics and advantages of the VAE, GAN, and Transformer designs, there are a number of avenues for further research that could raise the calibre and adaptability of systems that generate symbolic music.

1. **Hybrid Architectures**

Combining models, such as Transformer-based autoencoders or VAE-GANs, may enable systems to take advantage of structured latent representations while encouraging innovative output.

2. **Conditioned Music Generation**

More regulated and user-guided synthesis can be made possible by conditioning the generation process on musical characteristics like genre, mood, tempo, or key. Applications in personalised music systems or adaptive scoring would especially benefit from this.

3. **Dataset Enrichment and Augmentation**

More generalised models might result from expanding the training dataset to cover a wider range of musical genres, composers, and instruments. Robustness may also be enhanced by music-specific data augmentation methods such as pitch shifts, tempo modifications, or rhythmic adjustments.

4. **Real-Time and Edge-Compatible Generation**

For real-time applications like interactive games or live composition tools, models must be optimised for speedier inference. In the future, lightweight versions of the current models might also be used on embedded or mobile devices.

5. **Interactive and Assistive Tools**

In order to promote a more cooperative creative process between the model and the user, future research might focus on creating interactive music production systems that let human users provide motifs, themes, or limitations.

In summary, future work should concentrate on integrating their advantages, extending their control mechanisms, and making them available for imaginative and real-time applications, even though each model contributes special advantages to the creation of symbolic music.

References

Journal/Conference Papers

[1] C. Weiß, M. Mauch, S. Dixon, and M. Müller, “Investigating style evolution of Western classical music: A computational approach,” *Musicae Scientiae*, vol. 23, no. 4, pp. 486–507, 2019. doi: 10.1177/1029864918757595.

[2] K. Gajjar and M. Patel, “Computational musicology for raga analysis in Indian classical music: A critical review,” *Int. J. Comput. Appl.*, vol. 172, no. 9, pp. 42–47, 2017.

[4] S. Shetty and K. K. Achary, "Raga mining of Indian music by extracting arohana-avarohana pattern," *Int. J. Recent Trends Eng.*, vol. 1, no. 1, pp. 362, 2009.

[6] Y.-H. Liu and D.-C. Wu, "A high-capacity performance-preserving blind technique for reversible information hiding via MIDI files using delta times," *Multimedia Tools and Applications*, vol. 79, pp. 17281–17302, 2020.

[7] J. Rothstein, *MIDI: A Comprehensive Introduction*, vol. 7. AR Editions, Inc., 1995.

[8] Mangal, Sanidhya, Rahul Modak, and Poorva Joshi. *"LSTM based music generation system."* arXiv preprint arXiv:1908.01080 (2019).

[9] Y. Huang, X. Huang, and Q. Cai, “Music generation based on Convolution-LSTM,” *Computer and Information Science*, vol. 11, no. 3, pp. 50–56, 2018.

[10] F. Shah, T. Naik, and N. Vyas, “LSTM based music generation,” *2019 International Conference on Machine Learning and Data Engineering (iCMLDE)*, pp. 6–10, IEEE, 2019.

[11] A. A. Khamees et al., "Classifying audio music genres using a multilayer sequential model," in *Proc. International Conference on Advanced Machine Learning Technologies and Applications*, Cham: Springer International Publishing, 2021.

[12] M. T. Schärer, et al., "Sounds associated with the reproductive behavior of the black grouper (Mycteroperca bonaci)," *Marine Biology*, vol. 161, pp. 141-147, 2014.

[14] B. Pang, E. Nijkamp, and Y. N. Wu, "Deep learning with tensorflow: A review," *Journal of Educational and Behavioral Statistics*, vol. 45, no. 2, pp. 227-248, 2020.

[16] Wang, Tao, et al. "An intelligent music generation based on Variational Autoencoder." *2020 International Conference on Culture-oriented Science & Technology (ICCST)*. IEEE, 2020.

[17] Kingma, Diederik P., and Max Welling. "Auto-encoding variational bayes." 20 Dec. 2013.

[18] Pinheiro Cinelli, Lucas, et al. "Variational autoencoder." *Variational methods for machine learning with applications to deep networks*. Cham: Springer International Publishing, 2021. 111-149.

[21] Goodfellow, Ian, et al. "Generative adversarial networks." *Communications of the ACM* 63.11 (2020): 139-144.

[24] Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).

[25] Ghojogh, Benyamin, and Ali Ghodsi. "Attention mechanism, transformers, BERT, and GPT: tutorial and survey." (2020).

Reference/Handbooks

[5] J. Rothstein, MIDI: A Comprehensive Introduction, vol. 7. Madison, WI: AR Editions, Inc., 1995.

[23] Creswell, Antonia, et al. "Generative adversarial networks: An overview." *IEEE signal processing magazine* 35.1 (2018): 53-65.

Web

[3] Saptaks, indianclassicalmusic5.blogspot.com , [10/2/25].

[13] Developers, TensorFlow. "TensorFlow." *Zenodo*, 2022. [Online] [12/3/25]

[15] PrettyMIDI.prettymidi() , craffel.github.io, [12/3/25]

[19] VAE, ibm.com/think/topics/variational-autoencoder [12/3/25]

[20] Introduction to Variational Autoencoders (VAEs) in AI Music Generation**,** yuehan-z.medium.com [13/3/25]**.**

[22] GAN, developers.google.com/machine-learning/gan/gan_structure [13/3/25].

[26] Pop Music, medium.com/data-science [14/3/25].

Annextures

Annexture 1:

```
def midi_to_pianoroll(filepath, fs=100):
    try:
        midi = pretty_midi.PrettyMIDI(filepath)
        piano_roll = midi.get_piano_roll(fs=fs)
        piano_roll = (piano_roll > 0).astype(np.uint8)
        return piano_roll
    except:
        return None
def load_dataset(base_path, seconds=15, fs=100):
    X = []
    target_len = fs * seconds
    for composer in os.listdir(base_path):
        composer_path = os.path.join(base_path, composer)
        for file in os.listdir(composer_path):
            if file.endswith('.mid'):
                path = os.path.join(composer_path, file)
                pr = midi_to_pianoroll(path, fs)
                if pr is not None and pr.shape[1] >= target_len:
                    X.append(pr[:, :target_len])
    return np.stack(X)
```

Annexture 2:

```
class VAE(nn.Module):
    def __init__(self, input_dim=128*1500, latent_dim=256):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, 1024)
        self.fc21 = nn.Linear(1024, latent_dim)
        self.fc22 = nn.Linear(1024, latent_dim)
        self.fc3 = nn.Linear(latent_dim, 1024)
        self.fc4 = nn.Linear(1024, input_dim)
    def encode(self, x):
        h1 = F.relu(self.fc1(x))
        return self.fc21(h1), self.fc22(h1)
    def reparam(self, mu, logvar):
        std = torch.exp(0.5*logvar)
        eps = torch.randn_like(std)
        return mu + eps*std
    def decode(self, z):
        h3 = F.relu(self.fc3(z))
        return torch.sigmoid(self.fc4(h3))
    def forward(self, x):
        mu, logvar = self.encode(x.view(x.size(0), -1))
        z = self.reparam(mu, logvar)
        return self.decode(z), mu, logvar
def vae_loss(recon_x, x, mu, logvar):
    BCE = F.binary_cross_entropy(recon_x, x.view(x.size(0), -1), reduction='mean')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return BCE + KLD
from torch.utils.data import DataLoader, TensorDataset
import matplotlib.pyplot as plt

X_tensor = torch.tensor(X, dtype=torch.float32)
dataset = TensorDataset(X_tensor)
```

```
dataloader = DataLoader(dataset, batch_size=16, shuffle=True)

vae = VAE().cuda()
optimizer = torch.optim.Adam(vae.parameters(), lr=1e-4)
losses = []

for epoch in range(20):
    total_loss = 0
    for batch in dataloader:
        x = batch[0].cuda()
        optimizer.zero_grad()
        recon, mu, logvar = vae(x)
        loss = vae_loss(recon, x, mu, logvar)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(vae.parameters(), max_norm=1.0)
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch+1}, Loss: {total_loss}")
    losses.append(total_loss)

plt.plot(losses)
plt.title("VAE Training Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```

Annexture 3:

```
class PositionalEncoding(layers.Layer):
    def __init__(self, sequence_length, embed_dim):
        super().__init__()
        self.pos_encoding = self.get_positional_encoding(sequence_length, embed_dim)

    def get_positional_encoding(self, length, d_model):
        pos = np.arange(length)[:, np.newaxis]
        i = np.arange(d_model)[np.newaxis, :]
        angle_rates = 1 / np.power(10000, (2 * (i//2)) / np.float32(d_model))
        angle_rads = pos * angle_rates
        angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])
        angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])
        return tf.constant(angle_rads[np.newaxis, ...], dtype=tf.float32)

    def call(self, x):
        return x + self.pos_encoding[:, :tf.shape(x)[1], :]

def transformer_block(embed_dim, num_heads, ff_dim):
    inputs = layers.Input(shape=(None, embed_dim))
    attn_output = layers.MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim)(inputs,
inputs)
    x = layers.LayerNormalization()(inputs + attn_output)
    ff_output = layers.Dense(ff_dim, activation='relu')(x)
    ff_output = layers.Dense(embed_dim)(ff_output)
    x = layers.LayerNormalization()(x + ff_output)
    return tf.keras.Model(inputs=inputs, outputs=x)

def build_transformer(seq_len, feature_dim):
    inputs = layers.Input(shape=(seq_len, feature_dim))
    x = layers.Dense(EMBED_DIM)(inputs)
    x = PositionalEncoding(seq_len, EMBED_DIM)(x)
```

```
    for _ in range(NUM_LAYERS):
        x = transformer_block(EMBED_DIM, NUM_HEADS, FF_DIM)(x)

    outputs = layers.Dense(feature_dim)(x)
    return tf.keras.Model(inputs=inputs, outputs=outputs)

def visualize_prediction(true_seq, pred_seq):
    fig, axes = plt.subplots(FEATURE_DIM, 1, figsize=(10, 6))
    for i in range(FEATURE_DIM):
        axes[i].plot(true_seq[:, i], label='True', color='black')
        axes[i].plot(pred_seq[:, i], label='Predicted', linestyle='dashed', color='blue')
        axes[i].set_ylabel(f'Feature {i+1}')
    axes[0].legend()
    plt.suptitle("True vs Predicted Note Features")
    plt.xlabel("Time Step")
    plt.tight_layout()
    plt.show()

def train_transformer(model, dataset):
    losses = []
    optimizer = tf.keras.optimizers.Adam()
    loss_fn = tf.keras.losses.MeanSquaredError()

    for epoch in range(EPOCHS):
        idx = np.random.randint(0, len(dataset), BATCH_SIZE)
        batch = dataset[idx]
        x_input = batch[:, :-1, :]
        y_target = batch[:, 1:, :]

        with tf.GradientTape() as tape:
            y_pred = model(x_input, training=True)
            loss = loss_fn(y_target, y_pred)
        grads = tape.gradient(loss, model.trainable_variables)
```

```
        optimizer.apply_gradients(zip(grads, model.trainable_variables))
        losses.append(loss.numpy())

        if epoch % 10 == 0:
            print(f"Epoch {epoch}: Loss = {loss.numpy():.4f}")

    plot_losses(losses)

    sample = dataset[np.random.randint(len(dataset))]
    x_in = sample[np.newaxis, :-1, :]
    true_out = sample[1:, :]
    pred_out = model(x_in, training=False)[0]
    visualize_prediction(true_out, pred_out)
```

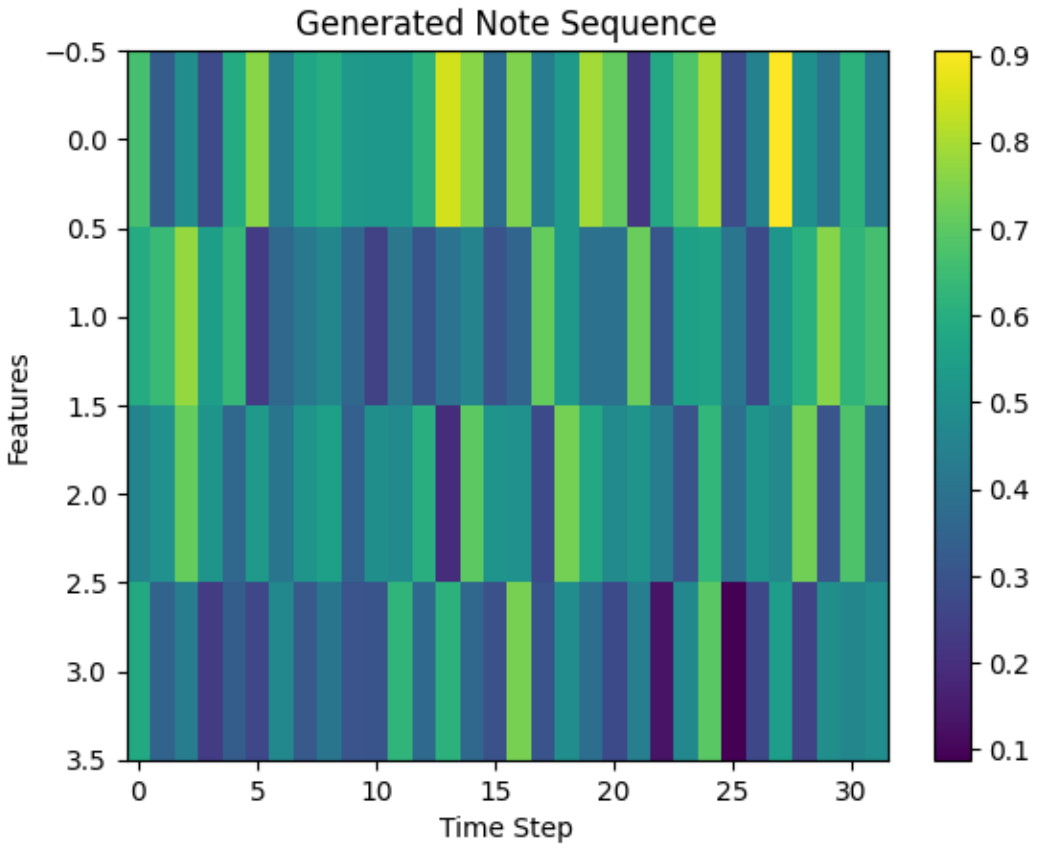
Annexture 4:

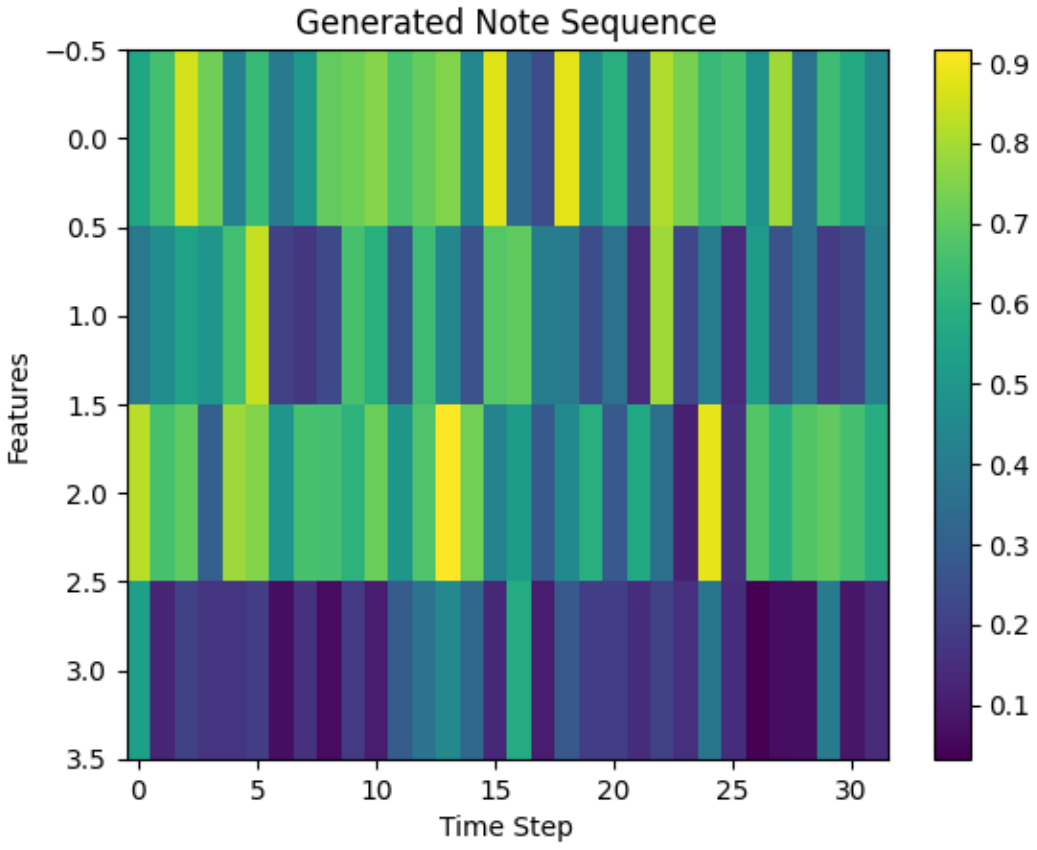
```
def build_generator():
    model = tf.keras.Sequential([
        layers.Input(shape=(LATENT_DIM,)),
        layers.Dense(256, activation='relu'),
        layers.Dense(NOTE_LENGTH * FEATURE_DIM),
        layers.Reshape((NOTE_LENGTH, FEATURE_DIM)),
        layers.Activation('sigmoid')
    ])
    return model
def build_discriminator():
    model = tf.keras.Sequential([
        layers.Input(shape=(NOTE_LENGTH, FEATURE_DIM)),
        layers.Flatten(),
        layers.Dense(256, activation='relu'),
        layers.Dense(1, activation='sigmoid')
    ])
    return model
```

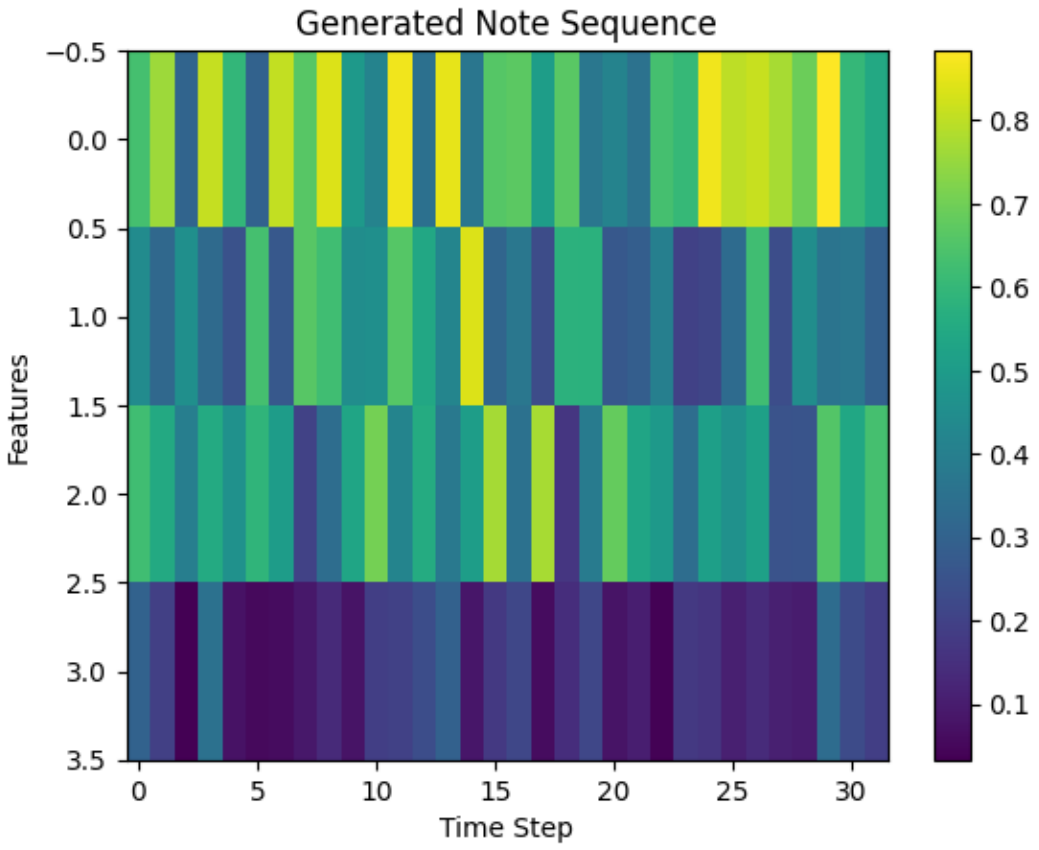
```
def visualize_output(generator):
    noise = tf.random.normal([1, LATENT_DIM])
    generated = generator(noise, training=False)[0]
    plt.imshow(generated.numpy().T, aspect='auto', cmap='viridis')
    plt.title("Generated Note Sequence")
    plt.xlabel("Time Step")
    plt.ylabel("Features")
    plt.colorbar()
    plt.show()

def train_gan(generator, discriminator, dataset):
    loss_fn = tf.keras.losses.BinaryCrossentropy()
    gen_opt = tf.keras.optimizers.Adam(1e-4)
    disc_opt = tf.keras.optimizers.Adam(1e-4)

    gen_losses = []
    disc_losses = []

    @tf.function
    def train_step(real_batch):
        noise = tf.random.normal([BATCH_SIZE, LATENT_DIM])
        with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
            fake_batch = generator(noise, training=True)
            real_output = discriminator(real_batch, training=True)
            fake_output = discriminator(fake_batch, training=True)
            gen_loss = loss_fn(tf.ones_like(fake_output), fake_output)
            disc_loss = loss_fn(tf.ones_like(real_output), real_output) + \
                        loss_fn(tf.zeros_like(fake_output), fake_output)
        grads_g = gen_tape.gradient(gen_loss, generator.trainable_variables)
        grads_d = disc_tape.gradient(disc_loss, discriminator.trainable_variables)
        gen_opt.apply_gradients(zip(grads_g, generator.trainable_variables))
        disc_opt.apply_gradients(zip(grads_d, discriminator.trainable_variables))
        return gen_loss, disc_loss
```

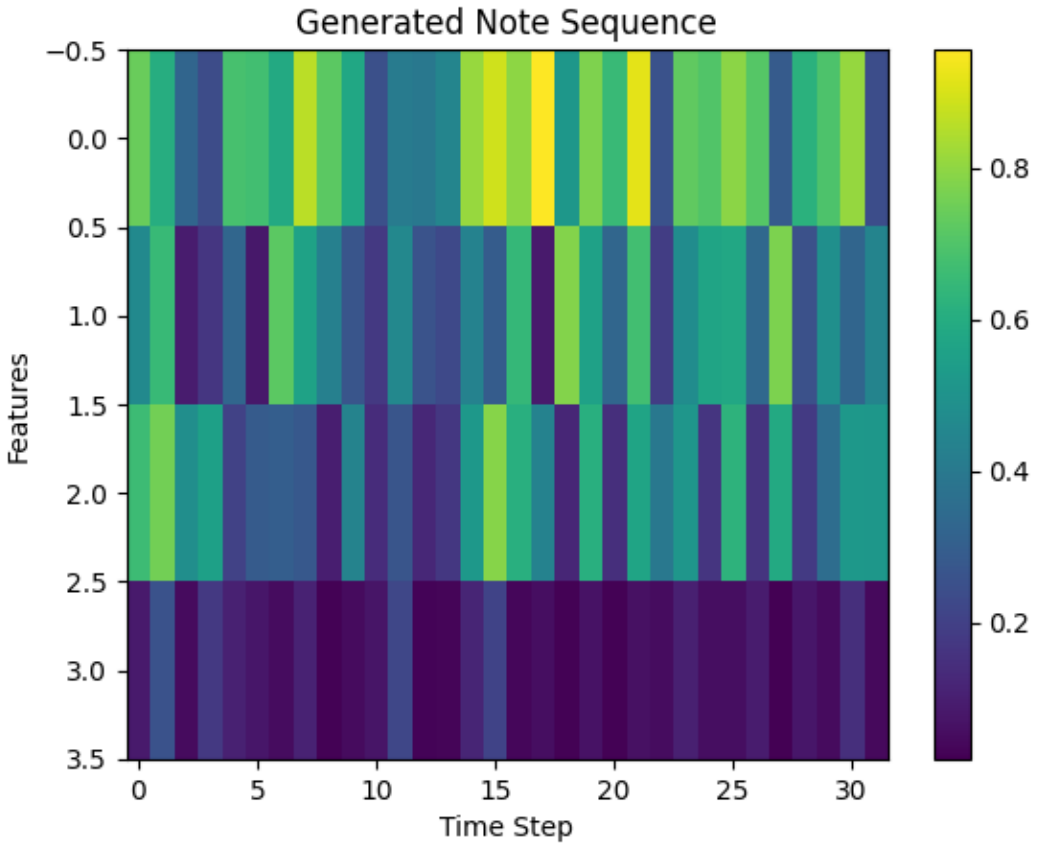
```
for epoch in range(EPOCHS):
    idx = np.random.randint(0, dataset.shape[0], BATCH_SIZE)
    real_batch = dataset[idx]
    gen_loss, disc_loss = train_step(real_batch)

    gen_losses.append(gen_loss.numpy())
    disc_losses.append(disc_loss.numpy())

    if epoch % 100 == 0:
        print(f"Epoch {epoch}: Gen Loss={gen_loss.numpy():.4f}, Disc Loss={disc_loss.numpy():.4f}")
        visualize_output(generator)
```

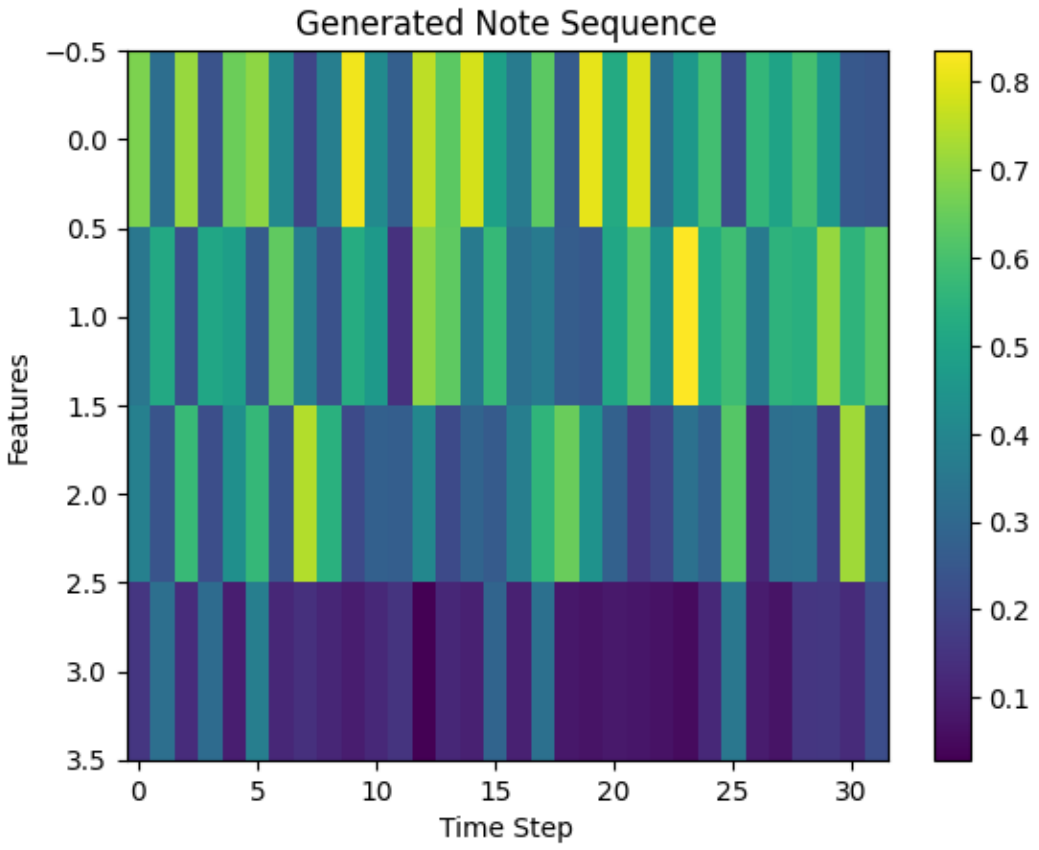
<u>Annexture 5</u>: GAN Heatmaps for Epochs 200-800

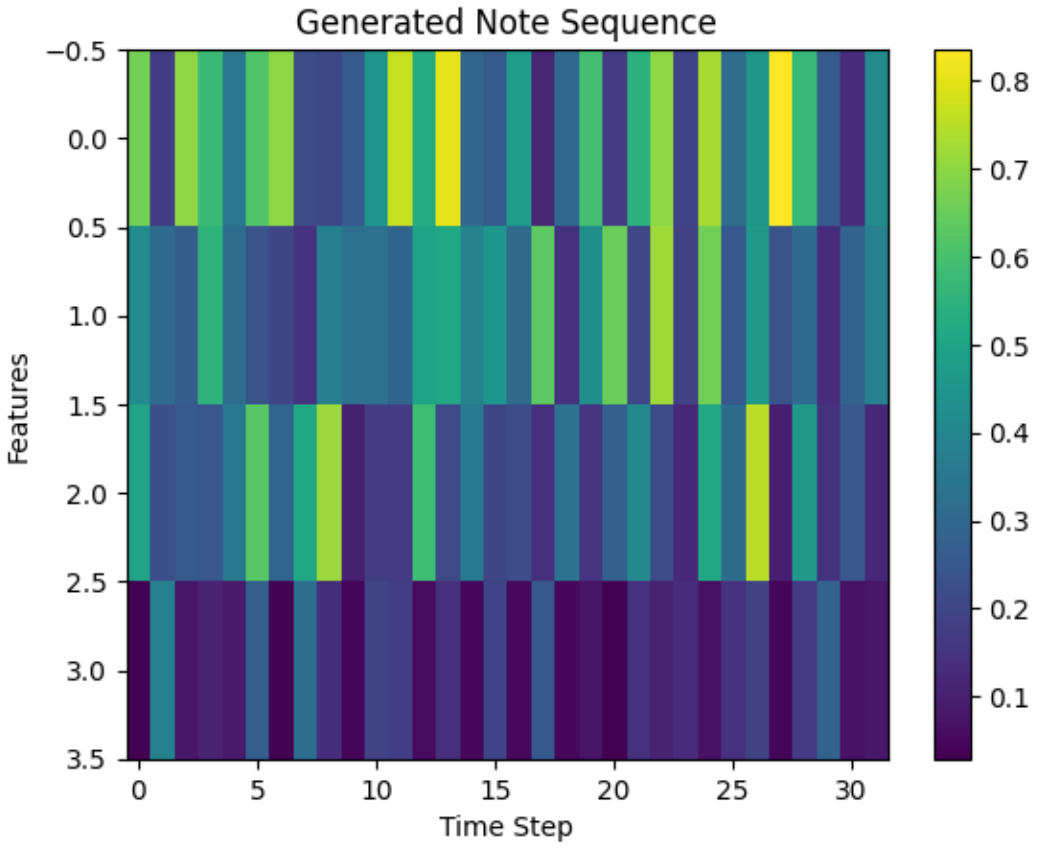

Generated Note Sequence
Features
Time Step
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
5
10
15
20
25
30
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1

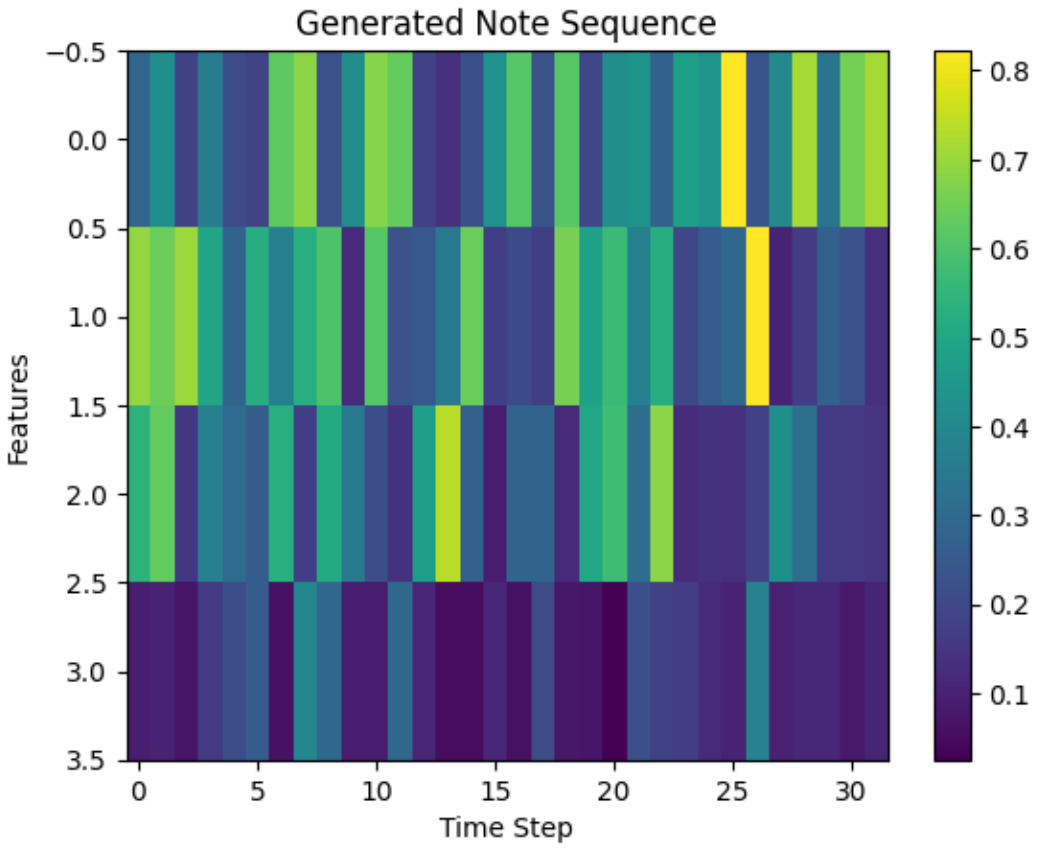

Generated Note Sequence
Features
Time Step
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
5
10
15
20
25
30
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1

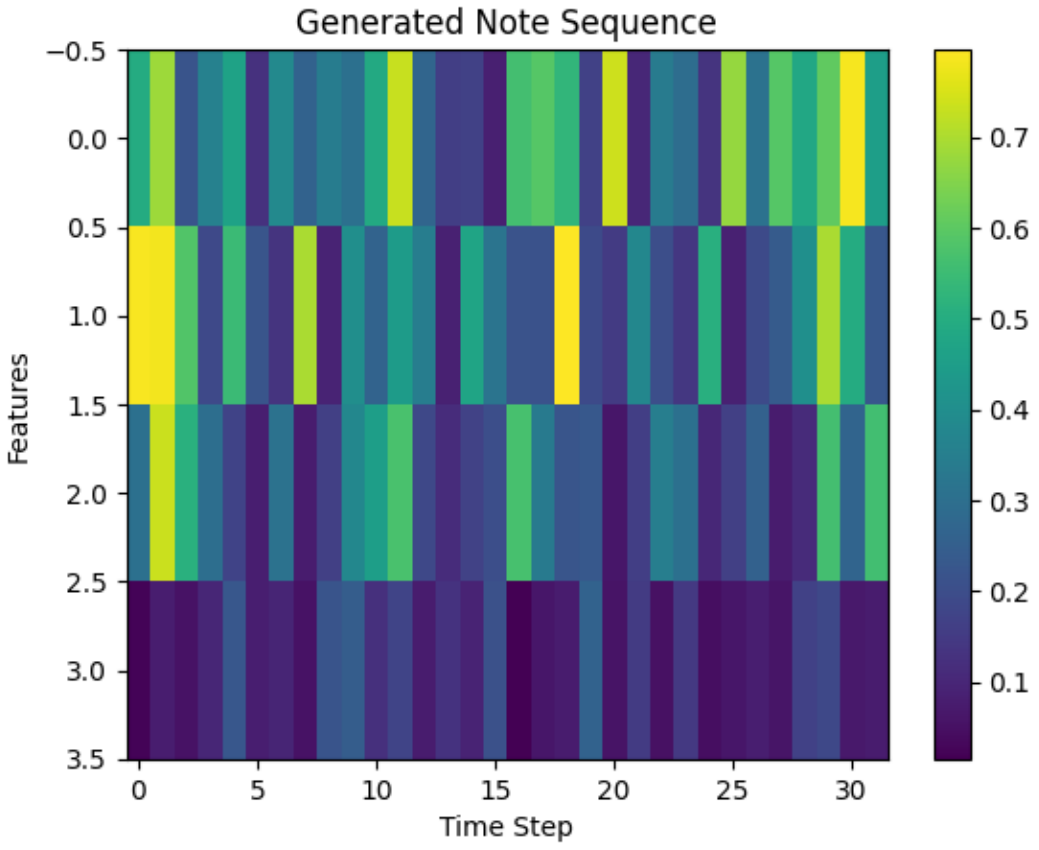

Generated Note Sequence
Features
Time Step
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
5
10
15
20
25
30
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1


Generated Note Sequence
Features
Time Step
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
5
10
15
20
25
30
0.8
0.6
0.4
0.2


Generated Note Sequence
Features
Time Step
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
5
10
15
20
25
30
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1


Generated Note Sequence
Features
Time Step
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
5
10
15
20
25
30
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1


Generated Note Sequence
Features
Time Step
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
5
10
15
20
25
30
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1


Generated Note Sequence
Features
Time Step
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
5
10
15
20
25
30
0.7
0.6
0.5
0.4
0.3
0.2
0.1