Music Generation Using Deep Learning

COMP9491: Applied AI – Project Report

Anand Viswanath (z5450230)

Nishant Saxena (z5325071)

Vishesh Malik (z5444643)

Vishvesh Rathi (z5349951)

Table of Contents

1. Introduction ... 4
2. Related Work ... 4
3. Methods ... 5
3.1 Pre-Processing ... 6
3.2 Model – GAN (Generative Adversarial Network) ... 6
3.2.1 Training Process ... 8
Training Procedure and Epochs ... 8
Loss Functions and Optimization ... 9
3.3 Model – LSTM (Long Short-Term Memory) ... 10
3.3.1 Architecture ... 11
3.3.2 Training Procedure ... 11
3.4 Model – Autoencoders ... 11
3.4.1 Data Collection ... 11
3.4.2 WaveNet Autoencoder ... 11
3.4.2.1 Architecture ... 12
3.4.2.2 Training procedure: ... 13
3.4.3 Conditional Variational Autoencoder ... 13
3.4.3.1 CVAE Architecture: ... 14
3.5 Post-Processing ... 14
4. Experimental Setup ... 15
4.1. Dataset ... 15
4.2. Evaluation Metrics ... 15
5. Model Results and Discussion ... 17
5.1 Model – GAN ... 17
5.2 Model - LSTM ... 18
5.3 Model – WaveNet Autoencoder ... 19
5.4 Model – Conditional Variational Autoencoder ... 20
6 Conclusion ... 22
6.1 Summary of Work ... 22
6.1.1 Key Findings ... 22
6.2 Limitations and Future Work ... 23
6.2.1 Limitations ... 23
6.2.2 Future Work ... 23

7. References 25
8. Appendix 26
8.1 Appendix A: Individual Contributions 26

1. Introduction

In today’s rapidly growing technological landscape, Artificial Intelligence is becoming an integral part of many industries, transforming the way we create and interact with digital content. One of most exciting advancements in this field is Generative AI, which after taking input from users in the form of Text, Audio or Video, or Code, can produce new content. Such AI technology has already made significant advancements in areas such as Image Generation, where tools like photoshop utilize AI to significantly help the user generate any type of content or perform edits using simple text prompts. Building on this potential, there is growing interest in applying Generative AI to music, aiming to create innovative and coherent musical compositions.

Our Project aims to harness the power of Generative AI to generate unique music compositions for short form media, addressing the current reliance on repetitive audio. With the rise of social media, the need for high quality, tailored short audio clips is growing. We aim to develop an AI-driven music generation system utilizing Long Short-Term Memory (LSTM), WaveNet Autoencoder, Conditional Variational Autoencoder and Generative Adversarial Networks to create short audio clips of a specified genres, tailored for social media posts, advertisements, and short videos such as social media posts, advertisements, and short videos.

2. Related Work

LSTM Based Music Generation System

The paper examines how Long Short-Term Memory (LSTM) networks can generate music. LSTMs, a type of recurrent neural network, excel at sequence prediction by capturing long-term dependencies. The authors use LSTMs to create a system that generates coherent and stylistically consistent musical pieces.

Trained on a large dataset of musical compositions, the LSTM network learns patterns and structures, predicting subsequent notes to generate new music that follows learned patterns. The generated music is coherent and stylistically consistent, showcasing the LSTM's ability to model complex temporal dependencies.

The paper discusses enhancing the model's performance with techniques like data augmentation and regularization, which prevent overfitting and improve generalization. It also explores applications in automated composition, music education, and interactive music systems.

Neural Audio Synthesis of Musical Notes with WaveNet Auto-Encoders

The paper presents an innovative approach to synthesizing musical notes using a deep learning model. The authors employ a WaveNet model, a powerful generative model for raw audio waveforms, to create high-quality musical note samples. By using an auto-encoder architecture, the model can effectively encode and decode audio signals, capturing intricate sound details.

Trained on a large dataset of piano recordings, the WaveNet auto-encoder learns the underlying structure of the audio in an unsupervised manner, enabling it to generalize across different sounds and instruments. The model comprises an encoder that compresses the audio into a latent representation and a decoder that reconstructs the audio from this representation. The resulting audio samples are high-fidelity and nearly indistinguishable from real recordings.

The paper also explores applications such as music generation, style transfer, and audio inpainting, demonstrating the model's versatility in handling various musical styles and instruments. The authors suggest future directions for enhancing neural audio synthesis through improved model architecture and training techniques, aiming to further improve the quality and diversity of synthesized sounds.

JukeBox: A Generative Model for Music

This paper by OpenAI introduces a neural network that generates high-fidelity music, complete with lyrics and instrumental accompaniments, across various genres and styles. JukeBox uses convolutional neural networks (CNNs) and autoencoders to learn the complex relationships between musical elements such as melody, harmony, rhythm, and timbre.

Trained on a vast music dataset, JukeBox operates at multiple hierarchical levels, capturing both broad structures and fine details of music. It can generate coherent lyrics, emulating the style of specific artists, adding a personalized touch to the tracks.

Applications of JukeBox include automated music composition, personalized music generation, and creating new or augmenting existing content in the music industry.

GANSynth: Adversarial Neural Audio Synthesis

The paper introduces a method for generating high-fidelity audio using Generative Adversarial Networks (GANs). Trained on a dataset of audio recordings, GANSynth uses a generator and discriminator network to produce and evaluate audio samples, leading to progressively improved and realistic waveforms.

GANSynth excels in capturing intricate sound details like timbre and dynamics, and it can efficiently generate high-quality audio samples faster than traditional methods. It can synthesize a wide range of instruments and styles, demonstrating its versatility.

Applications of GANSynth include music production, sound design, and audio effects creation. The paper also suggests potential improvements, such as enhancing temporal coherence and expanding the range of sounds.

MusicLM: Generating Music from Text

The paper introduces a model that generates music from textual descriptions. MusicLM interprets text to create musical compositions, capturing elements like genre, mood, tempo, and instrumentation.

Trained on extensive text and music datasets, MusicLM translates descriptive language into high-quality audio. Its hierarchical generation process ensures coherent and complex music over extended periods.

Applications include personalized music creation, multimedia soundtracks, and interactive music systems. MusicLM's ability to interpret nuanced text makes it a versatile tool for creative industries.

3. Methods

Fig1 Shows the overview of a typical music generation project. It illustrates the process of generating audio sequences from input audio data. Initially, the input audio waveforms are processed by a model to extract features. Some of our models are trained on spectrogram of input audios rather than the audio files themselves. This feature extraction converts the raw audio or spectrograms into features that contain important characteristics of the sound. These extracted features are then used to create sequences, notes, or frames, resulting in new audio outputs. The process of transformation raw audio into a structured representation and back into audio sequences, enables us to generate music from raw input.

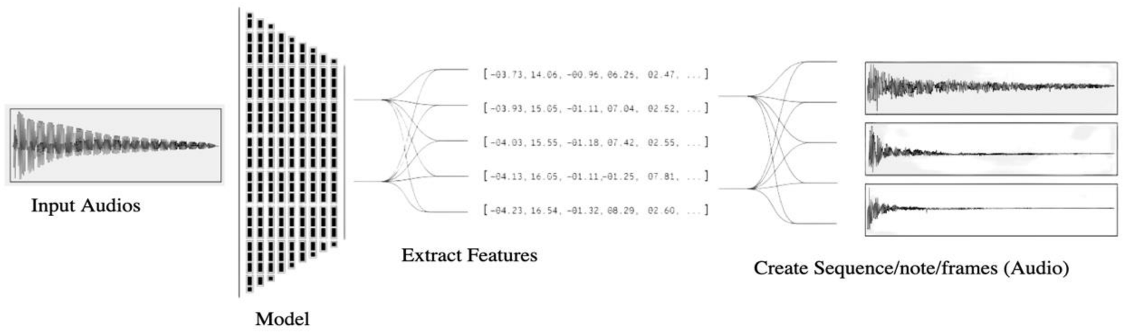


Fig 1: Overview/Working of Music Generation Task

3.1 Pre-Processing

For spectral representations, we computed short term Fourier Transform (STFT) magnitudes and phase angles using TensorFlow's '**.signal.stft**' built-in method. To further preprocess the spectrograms, we followed the steps that Google's Magenta team implemented [2]. We use an STFT with 256 stride and 1024 frame size, resulting in 513 frequency bins. We trim the Nyquist frequency to reduce dimensionality and pad along the time axis to ensure a consistent "image" shape of (256, 512, 2) from (247, 513, 2). The two channel dimensions correspond to magnitude and phase which depicts the real and imaginary part of a complex number. We take the log of the magnitude to better constrain the range and then scale the magnitudes to be between -1 and 1 to match the tanh output nonlinearity of the generator network. Concurrently, the phase information is extracted and similarly normalized. Later, this normalized magnitude, and phase is combined into two-channel tensor, resulting the spectrograms to have (256, 512, 2) shape as stated above. For visualization purpose, the spectrograms for an audio file 'guitar_acoustic_001-73-025' from the dataset before and after preprocessing it is shown in Fig 2.

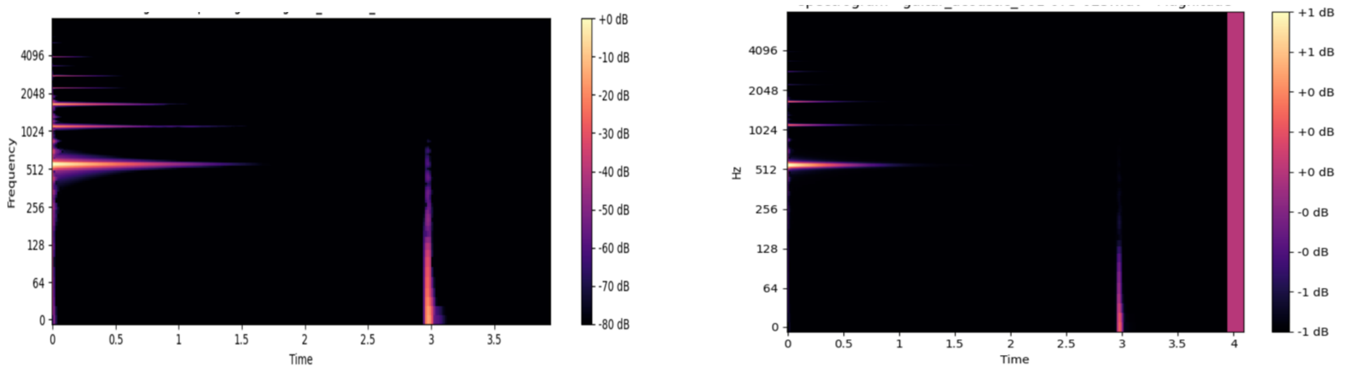


Fig 2: 'guitar_acoustic_001-73-025' before and after pre-processing. The decibel value ranges from –80db to 0db before preprocessing the audio file. Note that the pink vertical line in the preprocessed image (right image) represents the padded bin along the time axis and decibel value ranges between –1db to 1db.

3.2 Model – GAN (Generative Adversarial Network)

Our project employs a specific type of GAN architecture tailored for audio generation. The architecture consists of two main components: the generator and the discriminator. The generator is responsible for creating synthetic audio samples, while the discriminator evaluates the authenticity of these samples.

How GAN works?

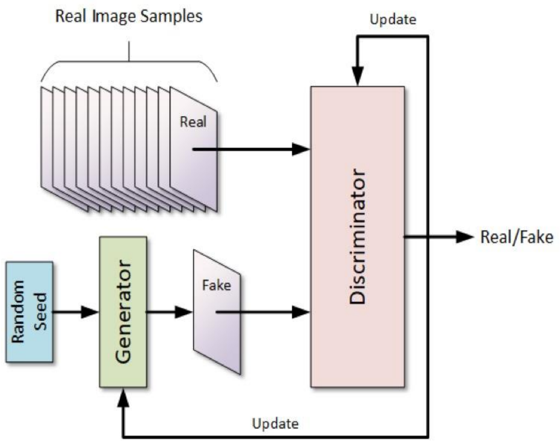


Fig 3: Basic GAN Architecture

Generator

The generator in our GAN architecture is designed to create realistic audio waveforms from random noise inputs. Here are the key components and concepts behind its architecture (fig 3):

- **Input Layer**: The generator starts with a random noise vector, sampled from a Gaussian distribution. This vector serves as the initial input that the model will transform into an audio signal.
- **Dense Layer**: The input noise vector is first processed through a dense (fully connected) layer that expands its dimensionality to form a higher-dimensional feature map, which provides a rich starting point for further transformations.
- **Batch Normalization**: After each convolutional layer, batch normalization is applied to stabilize and accelerate the training process by normalizing the inputs of each layer, ensuring consistent and smooth learning.
- **Leaky ReLU Activation**: Non-linear activation functions, specifically Leaky ReLU, are applied to introduce non-linearity and allow the model to capture complex patterns in the audio data.
- **Reshape Layer**: The dense layer output is reshaped into a format suitable for convolutional processing, typically as a three-dimensional tensor.
- **Transposed Convolutional Layers**: A series of transposed convolutional (also known as deconvolutional) layers are used to progressively upsample the feature map, increasing its dimensionality and refining the audio signal with each layer. These layers are responsible for creating the detailed structure of the audio waveform.
- **Output Layer**: The final layer uses a Tanh activation function to produce the audio waveform, ensuring that the output values are within the range of [-1, 1], which is suitable for audio signals.

Discriminator

The discriminator in our GAN architecture acts as a classifier that distinguishes between real and generated audio samples. Here are the key components and concepts behind its architecture:

- **Input Layer**: The discriminator receives audio samples in a specific format, typically as a three-dimensional tensor representing the audio waveform.
- **Convolutional Layers**: A series of convolutional layers are employed to analyze the input audio. These layers apply learned filters to detect various features and patterns within the audio data. Each convolutional layer reduces the dimensionality of the input, extracting hierarchical features that are crucial for classification.
- **Leaky ReLU Activation**: Like the generator, Leaky ReLU activations are used in the discriminator

to maintain gradient flow and improve training stability, allowing the network to handle variations in the input data effectively.

- **Dropout Layers**: Dropout is applied to prevent overfitting by randomly setting a fraction of the input units to zero during training, ensuring that the model generalizes well to unseen data.
- **Flatten Layer**: The output from the final convolutional layer is flattened into a one-dimensional vector, making it suitable for processing by fully connected layers.
- **Fully Connected Layer**: The flattened vector is processed through one or more fully connected layers to further refine the extracted features and prepare for the final classification step.
- **Output Layer**: The final output layer produces a single scalar value using a linear activation function. This value represents the probability that the input audio sample is real. A sigmoid activation function is applied to this output to obtain a probability score between 0 and 1.

3.2.1 Training Process

Dataset and Preprocessing

Training our GAN model necessitates a robust dataset of audio samples. To standardize their format and ensure consistency, these samples undergo several *preprocessing steps:*

- Normalization: Adjusting the amplitude of the audio samples to a standard level.
- Resampling: Changing the sample rate of the audio files to ensure uniformity across the dataset.
- Feature Extraction: Converting raw audio data into meaningful features that capture essential characteristics such as pitch, timbre, and rhythm.

Due to resource constraints, we focused on a single instrument—the guitar—out of a potential set of 11 different instruments. Approximately 34,000 audio samples were available for the guitar. The model was trained on subsets of 100, 1000, and 10000 guitar audio files to observe performance variations with different dataset sizes.

After loading the dataset, the audio files are reshaped into a consistent format of (num_samples, output_length, 1). The data is then split into training and testing sets with an 80% & 20% split, maintaining the same sample distribution by setting the random state to 42.

Training Procedure and Epochs

The training process involves iterative learning over numerous epochs. An epoch represents one complete pass through the entire training dataset, during which both the generator and discriminator are updated.

- **Generator's Role**: The generator starts with a random noise vector and produces an audio sample. This synthetic audio is then passed to the discriminator.
- **Discriminator's Role**: The discriminator receives both real guitar samples and the generated audio. It evaluates these inputs and provides feedback by distinguishing between real and fake samples.
- **Feedback Loop**: The discriminator's feedback is used to adjust the generator's parameters, aiming to produce more realistic audio in subsequent iterations. This adversarial process continues for many epochs, with each network improving its performance over time.

The number of epochs required for training depends on various factors, including the complexity of the dataset and the desired quality of the generated audio. Typically, hundreds to thousands of epochs are necessary to achieve satisfactory results but to limited resource and time constraints we trained the model with 100 audio files for 150 epochs and took 22 hours to finish model training, 1000 audio files with 60 epochs took ~28 hours to finish model training and 10000 audio files with 10 epochs took 36 hours to train the model.

Loss Functions and Optimization

In the world of machine learning, loss functions are essential tools that help us train and optimize our models. They act as a feedback mechanism, telling us how far off our model's predictions are from the actual outcomes. At its core, a loss function measures the "error" or "loss" between what our model predicts and the actual results. Think of it to score the model's performance. The lower the score, the better the model is doing. Choosing the right loss function is crucial because it directly influences how well the model learns from the data and improves over time. We encourage cross-Entropy Loss as methos which us used in classification tasks where the goal is to predict categories, such as identifying if an email is spam or not. It measures how well the predicted probabilities match the true labels. Here, it's being used with a setting (from_logits=True) that expects raw prediction scores (logits) rather than probabilities. It measures how well the predicted probabilities match the actual classes.

Loss functions are critical in guiding the optimization process for both the generator and discriminator:

- **Generator's Loss Function**: Measures how well the generated audio samples deceive the discriminator. The generator aims to minimize this loss, which indicates that it is producing more convincing fake samples.
- **Discriminator's Loss Function**: Assesses the discriminator's ability to correctly classify real and generated samples. The discriminator seeks to maximize this loss, improving its accuracy in distinguishing between real and fake audio.

Gradient Penalty

The gradient penalty helps stabilize the discriminator's training by encouraging the gradients to have a certain norm. This is a technique often used to improve the quality of generated images. The gradient penalty in a GAN is computed by first combining Mix real data and fake data with a random weight α. To achieve this, create a random α and linearly interpolate between the fake and real data. After that, the discriminator runs the mixed data to determine its prediction. Next, we use autograd functionality to compute the gradients of these predictions with respect to the interpolated data. After that, the gradients are rearranged, and the norm, or magnitude, is computed. The squared difference between this norm and one is used to calculate the penalty. The mean of these fines is then determined. The Lipschitz constraint is enforced in the GAN training with the application of this gradient penalty, which aids in training process stabilization.

Discriminator Loss

Real data is assigned a label of 0.9 plus some noise, and fake data is assigned a label of 0 plus some noise. This technique is called label smoothing and helps the model to generalize better.

Loss Calculation:

- Real Loss: Measure how far the discriminator's predictions on real data are from the label 0.9.
- Fake Loss: Measure how far the predictions on fake data are from the label 0.
- Gradient Penalty: Include the gradient penalty calculated earlier.
- Total Loss: Sum of the real loss, fake loss, and the gradient penalty multiplied by a factor lambda_gp (here, 10.0).

Generator Loss

The generator's goal is to make the discriminator believe the fake data is real.

Loss Calculation: Measure how far the discriminator's predictions on fake data are from the label 1 (indicating real).

Optimization techniques are employed to minimize the loss functions and enhance the performance of both networks:

- **Gradient Descent**: A fundamental optimization algorithm used to update the network parameters by minimizing the loss functions. It involves calculating the gradient of the loss with respect to the network parameters and adjusting the parameters in the opposite direction of the gradient.
- **Adam Optimizer**: An advanced optimization algorithm that combines the benefits of Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp). It adjusts the learning rate for each parameter individually, leading to faster convergence and better performance.
- **Batch Normalization**: A technique used to stabilize and accelerate the training process by normalizing the inputs of each layer within the network. It helps prevent issues such as vanishing gradients and improves the generalization capability of the model.
- **Learning Rate Scheduling**: Adjusting the learning rate during training can lead to better convergence. A higher learning rate may be used initially, with gradual decay over time to fine-tune the model.
    - Exponential Decay: The learning rates for both the generator and discriminator start at 2e-4 and decay over time by a factor of 0.9 every 10,000 steps.
    - Optimizers: Adam optimizers are used for both the generator and the discriminator with learning rates defined by the schedules. The clipvalue parameter helps in stabilizing training by preventing the gradients from getting too large.

3.3 Model – LSTM (Long Short-Term Memory)

Long Short-Term Memory (LSTM) was utilized due to its ability to handle sequential data and capture long-term dependencies, which are crucial for generating coherent and musically meaningful audio sequences. LSTM is also suitable for time-series tasks, making them ideal for audio generation tasks as well. The LSTM model for this project is designed in such a way that it processes **sequences** of spectrograms to generate spectrograms which are later converted into audio files in ‘.wav’ format. Sequences are used because we want to predict the next note/feature/frame in the sequence based on the previous ones.

The sequences were created using the **‘Sliding Window’** approach where a fixed sized window slides over the dataset sequentially rather than in one go (Fig 4). For example, given a sequence length of 16 and training spectrograms of 1000 audio files, the 'sequence_array_train' shape = (984, 16, 256, 512, 2) indicating 984 (1000–16 = 984) sequences, each comprising 16 consecutive spectrogram frames, where each frame has dimensions 256 x 512 with 2 channels (magnitude + phase).

We mainly used two approaches to process the results of LSTM. The first approach involves converting each frame of the generated spectrogram into audio individually and then concatenating the resulting audio clips. The second approach involves concatenating all the frames of the generated spectrogram by the LSTM into a single large spectrogram and then converting this combined spectrogram into Audio. The key observation about these approaches is discussed in section 5.2.

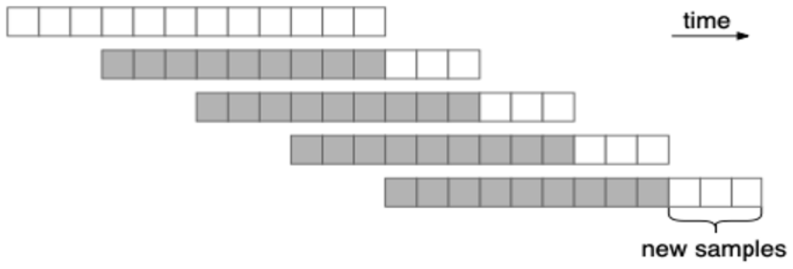


Fig 4: Sliding Window Approach [4]

3.3.1 Architecture

The model accepts input sequences of shape (seq_len, 256, 512, 2), where seq_len = 16, which is the length of the sequence, and 256 x 512 x 2 represents the dimensions of the spectrogram. It is followed by a **Time Distributed** Flatten Layer which flattens each spectrogram frame within the sequence, converting the 2D or 3D array into a 1D array, allowing it to be processed by the LSTM layers as sequential info.

The model includes five **Bidirectional** LSTM layers, each with 128 LSTM units. The bidirectional nature of these layers allows the model to learn temporal dependencies in both forward and backward directions. To prevent overfitting, **dropout** layers are included with a rate of 0.2 (20%) after each LSTM layer by randomly dropping a fraction of the units at the time of training. A **dense** layer is used to map the LSTM outputs to the desired output shape. Finally, the final output is **reshaped** to have the size of the original spectrogram dimensions (256, 512, 2).

3.3.2 Training Procedure

The LSTM model is compiled using the **Mean Squared Error (MSE)** loss function, which measures the difference between the predicted and actual audio sequences. The Adam optimizer is used with default parameters to train the model. Nsynth dataset was already pre-divided into training and validation sets, so the validation set was used to monitor the model's performance and prevent overfitting. Training the model takes more than +4hr on instrument-specific sequences of spectrograms with 1000 (Train) and 100 (Valid) for 50 epochs with a **Batch Size** of 32. The performance of the LSTM model is evaluated using the Mean Squared Error (MSE) and **Mean Absolute Error (MAE)** metrics. These metrics help in assessing the model's prediction accuracy. Furthermore, the produced audio samples are assessed both visually and audibly to evaluate their realism and musical quality.

3.4 Model – Autoencoders

3.4.1 Data Collection

We used the NSynth Dataset, focusing on one instrument family at a time. For instance, for the organ instrument, we found 34,477 training samples, 1,598 validation samples, and 502 test samples, but could only train on 500 to 1,000 files due to system limitations. Audio files were loaded using the ‘Librosa’ python library at a sampling rate of 16,000 Hz, normalized to [-1.0, 1.0], and checked for pitches between 24 and 84. Files were fixed to a length of 64,000 kHz, padded or truncated as needed. Only files meeting these criteria were used to train, test, and validate the Wavenet and Conditional Variational Autoencoders, with data reshaped for convolutional operations.

3.4.2 WaveNet Autoencoder

The WaveNet model we used for our music generation project is a state-of-the-art neural network originally developed by DeepMind for generating raw audio waveforms [7]. It is built on a deep convolutional architecture designed to understand and generate temporal patterns in audio data.

3.4.2.1 Architecture

Our implementation for this model includes the following layers:

Input layer: Our model starts with a one-dimensional array representing the audio waveform. We add an extra dimension to make it suitable for convolutional operations.

Casual Convolution Layer: The first layer applies causal convolutions, which ensures that the model respects the temporal sequence of the audio samples and cannot violate the ordering in which we model the data, crucial for generating coherent sound. This being one of the main ingredients of a Wavenet model architecture.

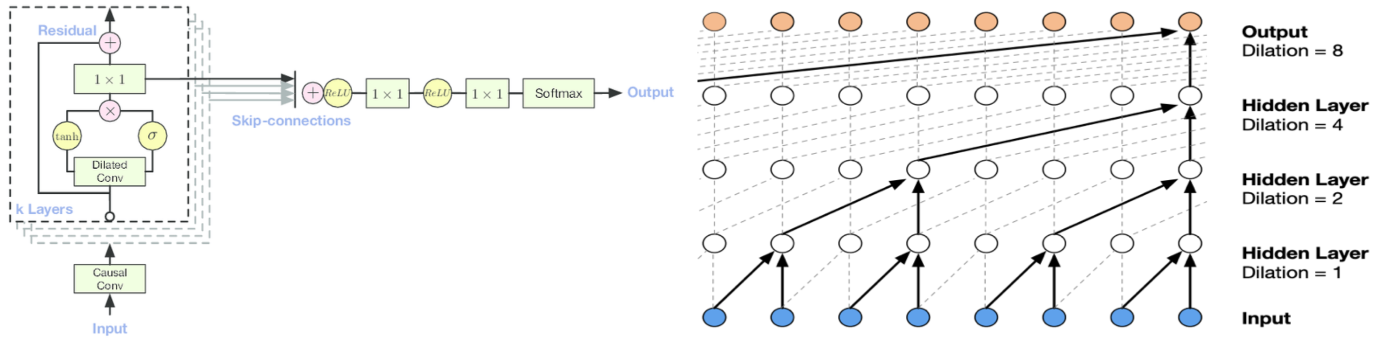


Fig 5: Visualization of the Architecture, the Residual Block and stack of dilated causal convolutional layers

***Residual Layers**:*

The core of our WaveNet Model comprises of a series of Residual Blocks, each consisting of:

- **Dilated convolutions**. These convolution layers use increasing dilation rates, enabling the model to capture long-range dependencies in the audio data. A dilated convolution layer has a filter which is applied over an area larger than its length by skipping its input values with a certain step, this makes it equal to a convolution layer with a larger filter by dilating but significantly more efficient. Fig. 5 depicts dilated causal convolutions for dilations 1, 2, 4, and 8. [7]
- **Activation and Normalization**. We use ReLU activations functions and batch normalization functions to stabilize training and improve convergence.
- **Dropout Layers**: Dropout is applied to prevent overfitting, thereby enhancing the model's ability to generalize from the training data.
- **Skip Connections**: There connections help propagate gradients through the network, making the training more efficient.

Bottleneck and Reconstruction Layers:

- **Encoding**: the output from the residual blocks passes through a bottleneck layer, which reduces its dimensionality and captures the most important features of the audio.
- **Decoding**: The encoded features are up sampled and passed through another set of residual blocks to reconstruct the audio waveform.

Output Layer: The final layer is a convolutional layer with a linear activation function that produces the predicted audio waveform.

3.4.2.2 Training procedure:

The training procedure for our WaveNet-based music generation model involved several key steps aimed at optimizing its performance and efficiency. We used Mean Absolute Error (MAE) as our loss function to measure how different the predicted audio waveforms were from the actual ones, helping the model to produce more accurate audio samples. We chose the Adam optimizer because it's known for its efficiency in training deep neural networks and set an initial learning rate of 0.001. To make the training process more robust, we added Early Stopping and ReduceLROnPlateau callbacks. Early Stopping kept an eye on the validation loss and stopped training if there was no improvement for a few epochs, which helped prevent overfitting and saved computational resources. ReduceLROnPlateau reduced the learning rate when the validation loss plateaued, allowing the model to fine-tune itself better.

We trained the model for 50 epochs with a batch size of 16, trying to balance enough training iterations with our computational limits. During testing, we made some improvements: we initially used dilation rates of [1, 2, 4, 8, 16] in the residual blocks, but this complexity led to the model not generating any music at all, indicating it wasn't learning properly within our time constraints. Since we couldn't afford training times longer than 24 hours, we reduced the dilation rates to [1, 4, 16], which helped the model learn better. Additionally, we initially trained the model on a diverse dataset with various instrument families, but this made it hard for the model to learn specific patterns. By focusing on one instrument family per training cycle, we achieved better learning outcomes and more coherent generated music. However, the generated music was still not up to the standards, and the model would require more complex architectures and more training on better computational resources to perform better.

3.4.3 Conditional Variational Autoencoder

Additionally, to the WaveNet model, we used a Conditional Variational Autoencoder model as well. A normal Variational Autoencoder model is a type of generative model that learns to encode input data into a latent space and then decode it back to reconstruct the original data. [8] It consists of two main parts: the encoder and the decoder. The encoder maps the input data to a probability distribution in the latent space, typically modeled as a Gaussian distribution. The decoder then samples from this distribution to generate new data points. VAE is trained to minimize the reconstruction error between the input and generated data and the KL divergence between the learned latent distribution and a prior distribution. This encourages the latent space to follow a known distribution, enabling efficient sampling and generation of new data points.

Building on the VAE, a Conditional Variational Autoencoder (CVAE) incorporates additional information (condition) into the model. This condition could be any attribute relevant to the data, such as labels, class information, or other additional information. By conditioning the model, i.e., encoder and decoder, on this additional information the CVAE can generate data that conforms to specific conditions.

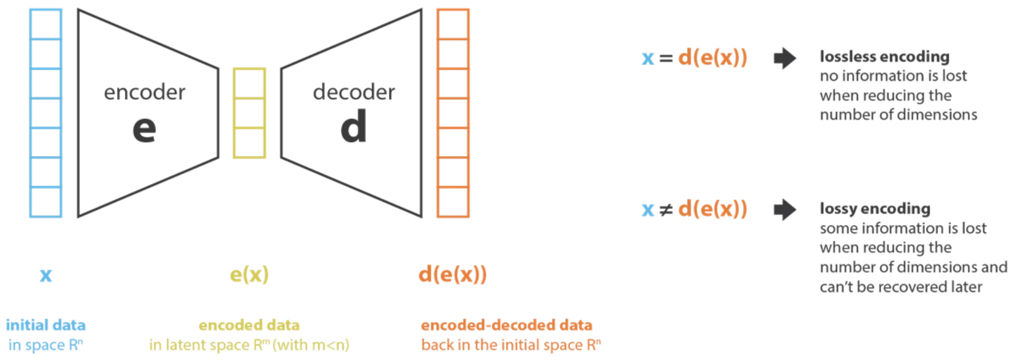

encoder
e
decoder
d
x
e(x)
d(e(x))
initial data
in space Rⁿ
encoded data
in latent space Rᵐ (with m<n)
encoded-decoded data
back in the initial space Rⁿ
x = d(e(x))
lossless encoding
no information is lost when reducing the number of dimensions
x ≠ d(e(x))
lossy encoding
some information is lost when reducing the number of dimensions and can't be recovered later

Fig-6: Conditional Variational Autoencoder Architecture

3.4.3.1 CVAE Architecture:

Architecture of the CVAE implementation for the project is as follows:

1. **Encoder**: The encoder network takes both the input audio data and the condition (instrument type) and maps them to a latent space. It outputs the mean and variance parameters of the latent distribution. The encoder architecture includes:
    1.1. Input layer for audio data.
    1.2. Multiple 1D convolutional layers with ReLU activation and padding.
    1.3. Flattening layer to prepare for dense layers.
    1.4. Two dense layers to output z_mean and z_log_var.
    1.5. Sampling layer to generate the latent vector z.
2. **Latent Space**: The latent space is sampled using the reparameterization trick, which allows the gradient to propagate through the sampling process. This trick involves sampling from a standard normal distribution and then scaling and shifting the samples using the mean and variance parameters from the encoder.
3. **Decoder**: The decoder network takes samples from the latent space along with the condition and generates the output audio data. The conditioning information is concatenated with the latent vectors to guide the generation process. The decoder architecture includes:
    3.1. The input layer for the latent vector ‘z’.
    3.2. Dense layers followed by reshaping to match the original audio dimensions
    3.3. Multiple transpose convolutional layers with ReLU activation to reconstruct the audio waveform.
    3.4. Output layer with sigmoid activation to produce the final audio output.
4. **Loss Function:** The training objective of the CVAE includes both the reconstruction loss, which measures how well the generated audio matches the input audio, and the KL divergence loss, which ensures that the latent space follows the prior distribution. The overall loss is a weighted sum of these two components.

3.5 Post-Processing

Post-processing in this project mainly involves conversion of spectrograms back into audio waveforms to further evaluate them using different metrics. It involves creating a function which takes in a generated spectrogram from the model as an input and spits out an audio waveform version of this spectrogram. Initially, the function separates the magnitude and phase components from the spectrogram. The magnitude is then de-normalized to its original scale by reversing the normalization applied during the preprocessing stage. Next, the function reconstructs the complex-valued spectrogram by combining the de-normalized magnitude with the phase, which is scaled back to its original range. Lastly, we converted the complex spectrogram back into a time-domain audio signal by computing the inverse of Short-Time Fourier Transform (iSTFT), from librosa's '**.istft'** built-in method.

For VAE and WaveNet models, any audio samples generated using the Nsynth and Gtzan dataset contained a lot of noise over the main audio sample. Attempted to clean up the noise was made using the noisereduce python library. Noisereduce is a noise reduction algorithm in python that reduces noise in time-domain signals like speech, bioacoustics, and physiological signals. It relies on "spectral gating", a form of Noise Gate. It works by computing a spectrogram of a signal (and optionally a noise signal) and estimating a noise threshold (or gate) for each frequency band of that signal/noise. That threshold is used to compute a mask, which gates noise below the frequency-varying threshold. [9] However, it also removed important musical features from the clip along with the noise.

4. Experimental Setup

4.1. Dataset

NSynth

The NSynth dataset has a total of 305,979 different notes from a variety of instrument families, whose distribution is given below:

	SOURCE			
FAMILY	ACOUSTIC	ELECTRONIC	SYNTHETIC	TOTAL
Bass	200	8387	60368	68955
Brass	13760	70	0	13830
Flute	6572	35	2816	9423
Guitar	13343	16805	5275	35423
Keyboard	8508	42645	3838	54991
Mallet	27722	5581	1763	35066
Organ	176	36401	0	36577
Reed	14262	76	528	14866
String	20150	84	0	20594
Synth Lead	0	0	5501	5501
Vocal	3925	140	6688	10753
TOTAL	108978	110224	86777	305979

Table 1: Instrument-Wise Dataset Distribution

The samples are recorded at 16 KHz (frequency) and are 4 seconds long. They also contain information such as pitch, velocity and instrument type. The dataset has been used in multiple studies to generate high quality music and is a very realistic and large dataset. For this project, we chose audio clips whose pitch ranges between 24-84hz, as they are most likely to sound natural to an average listener.

GTZAN

The GTZAN dataset [6] is considered to be the MNIST of music generation. It has genre-based audio sets collected from CD's, radio and microphone recordings. There are 100, 30 second long, audio files along with corresponding Mel-Spectrograms for every genre. While being explored, this dataset was not used extensively as it had a lot of lyrical data and musical variation, which led to a lot of noise generated music.

4.2. Evaluation Metrics

We have proposed and used multiple metrics to compare the generative results of the different models. These are explored in more detail below.

We calculate the pitch-based metrics by using numpy arrays extracted using librosa. FAD and KL Divergence are calculated on Mel Frequency Cepstral Coefficients (MFCC's) which can be described as the spectrograms of spectrograms and is a feature extraction method for audio analysis. We use the first 13 coefficients, out of the total 40, which contain all the cepstral coefficients and the energy coefficient.

Number of Pitches

The number of pitches refers to the number of different notes present in the resultant output; as the ideal is producing one note with no noise, the lower this metric is, the better. The metric discriminates against models which produce many different pitches in the generated audio as that can relate to noisy and uncoherent audio.

Pitch Entropy

Pitch entropy refers to the entropy distribution of pitch classes in the resultant clip. Lower is better because that means that less classes are played overall. The formula is given below.

$$\boxed{pitch\_entropy = -\sum_{i=0}^{127} P(pitch = i) \log_2 P(pitch = i)}$$

Pitch Range

Pitch range refers to the difference between the highest pitch played and the lowest. This metric is used for inference for finding out whether there are wild, major swings in the pitches played, or if they are more coherent. More coherent audio has a lower pitch range.

KL Divergence

The Kullback-Liebler (KL) Divergence between two probability distributions is a measure of how much one distribution diverges from the first true distribution. In evaluating the generated music, KL divergence can provide insights into how well the model is capturing the underlying patterns and distributions present in real music data. The formula is given below.

$$\boxed{D_{KL}(P||Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}}$$

Fréchet Audio Distance

Fréchet Audio Distance (FAD) compares embedding statistics generated on the evaluation set with the generated music. It is a reference metric where ground truth audio is unavailable. A lower FAD signifies more similarity between generated music and the evaluation set. The metric was used to compare different generated audio files and the audio files within the evaluation set to evaluate the perceptual quality and diversity of synthetic distributions. The formula is given below.

$$\boxed{\mathbf{F}(\mathcal{N}_b, \mathcal{N}_e) = \|\mu_b - \mu_e\|^2 + tr(\Sigma_b + \Sigma_e - 2\sqrt{\Sigma_b \Sigma_e})}$$

where μ_b and μ_e are the mean vectors and Σ_b and Σ_e are the covariance matrices of the real and generated audio features, respectively. Tr represents the trace of the matrix.

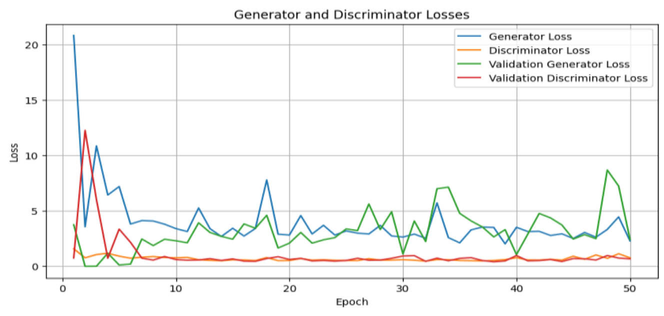
Human Evaluation

As music can be very subjective, we also use human evaluation to compare audio files generated by our models. This provides a holistic approach to evaluation rather than just numerical analysis. Friends and different users were asked to assess the quality and realism of the generated audio samples.

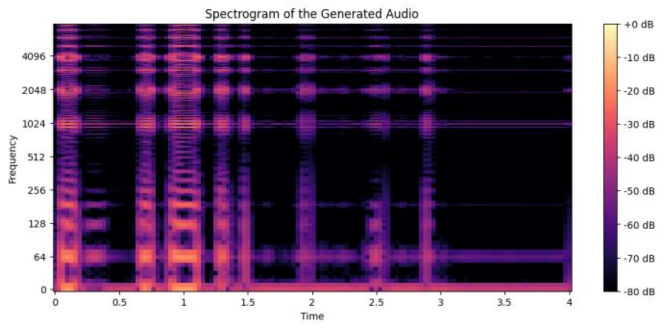
5. Model Results and Discussion

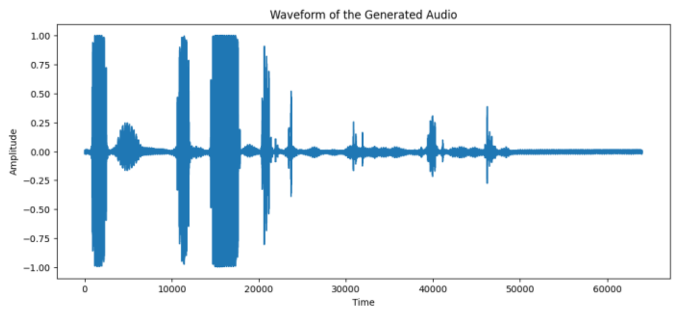
5.1 Model – GAN

We evaluated the performance of the Generative Adversarial Network (GAN) model using a variety of metrics, including pitch range, KL divergence, Fréchet audio distance (FAD), and human assessment. Despite having two options, we decided to go with the less complicated one since Wasserstein GAN with data augmentation was computationally very expensive and we couldn't train it on even the smallest dataset. Thus, let's start with a simple GAN technique. We Trained the model thrice, trained on subsets of 100, 1000, and 10000 guitar audio files to observe performance variations with different dataset sizes.

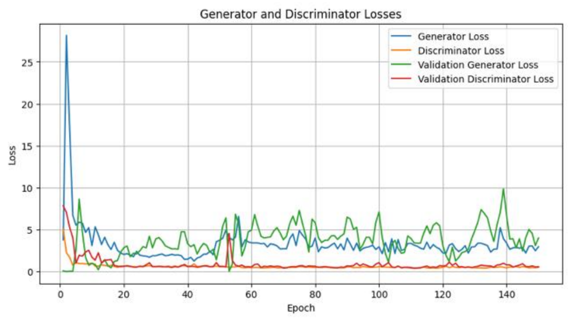


TRAIN 1 - EPOCHS 50, 1000 audio files The generator's performance improved significantly during the first training session. The generator had trouble at first, but as time went on, as seen by the declining loss, it started to function better. From the beginning, the discriminator was quite successful in differentiating between produced and actual data with minimal loss values. It's encouraging that the discriminator and generator both fared well on the validation set. The resulting audio featured a variety of characteristics; the spectrogram showed a range of frequency patterns, and the waveform clearly varied. This session showed a vast range of pitches, as seen by its very high Pitch Entropy of 4.38 and broad Pitch Range of 3809.27. The KL Divergence of 0.45 and Fréchet Distance of 30216.45 indicate that there are 1160 unique pitches.

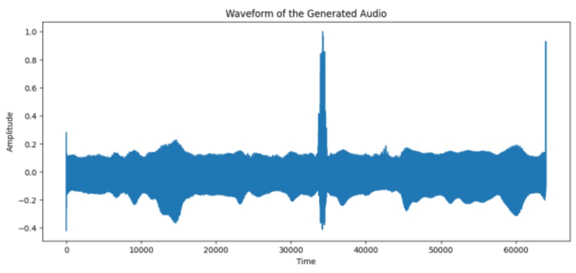


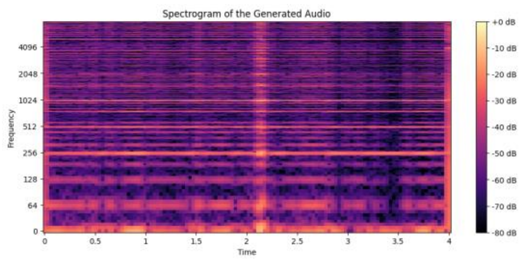


TRAIN 2 - 100 audio files, 150 Epochs, 64 Batch Size

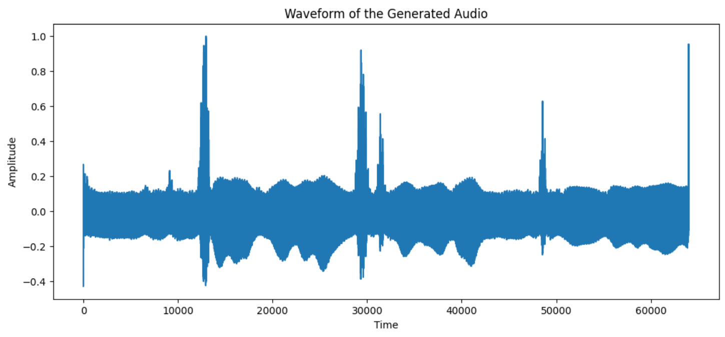


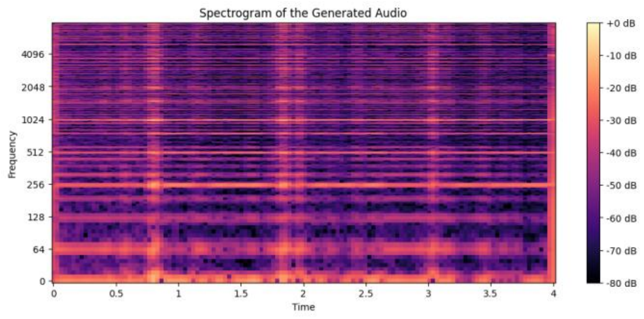
The generator's performance improved significantly during the first training session. The generator had trouble at first, but as time went on, as seen by the declining loss, it started to function better. From the beginning, the discriminator was quite successful in differentiating between produced and actual data with minimal loss values. It's encouraging that the discriminator and generator both fared well on the validation set. The resulting audio featured a variety of characteristics; the spectrogram showed a range of frequency patterns, and the waveform clearly varied. This session showed a vast range of pitches, as seen by its very high Pitch Entropy of 4.38 and broad Pitch Range of 3809.27. The KL Divergence of 0.45 and Fréchet Distance of 30216.45 indicate that there are 1160 unique pitches. These metrics suggest a decrease in pitch diversity and an increase in divergence compared to the first session, indicating that while the model remained stable, the variety and similarity to real audio decreased.





TRAIN 3 - 10000 audio files 10 Epochs and 32 Batch size

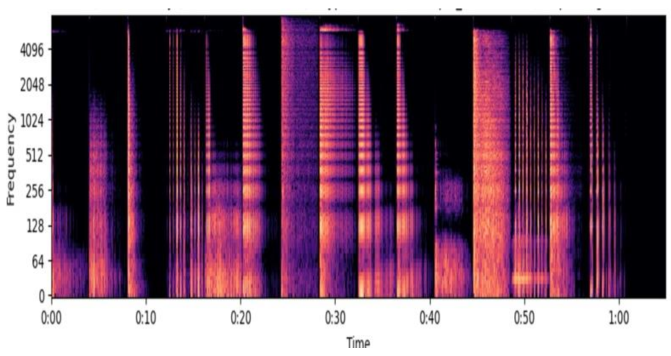




The third training on the trend of generating structured and coherent audio. The waveform showed consistent amplitude variations with regular peaks, indicating a steady pattern. The spectrogram continued to display structured frequency bands. Although we don't have the loss graphs for this session, the generated audio still exhibited high quality. Quantitative metrics included a Pitch Entropy of 3.25, a Pitch Range of 3746.90, and 540 pitches. The KL Divergence increased further to 4.87, and the Fréchet Distance reached 41400.22. These results show a continued trend of reduced pitch diversity and increased divergence from real audio files. Despite this, the generated audio remained complex and coherent, reflecting the model's ability to produce reliable outputs.

5.2 Model - LSTM

As discussed under section 3.3, we mainly used two approaches to process the output of the LSTM. The key findings about approach 1 were that the generated spectrograms had discontinuities between consecutive frames causing unnatural transitions and dissonant sounds due to lack of harmonic coherence. Also, it is computationally intensive to process each frame individually. From approach 2, the key findings from these spectrograms where the boundaries between frames had more horizontal lines and introduced artifacts, which might indicate a more constant or repetitive sound which might also indicate less variation of sound in the audio. The fig below shows how the spectrograms would look using these two approaches:



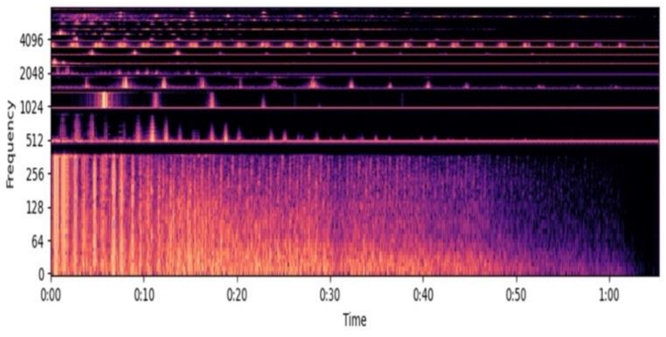


Fig 6: The left spectrogram from approach 1 has more vertical lines with unnatural transitions from one frame to another. The right spectrogram from approach 2 has more horizontal lines which introduced some artifacts.

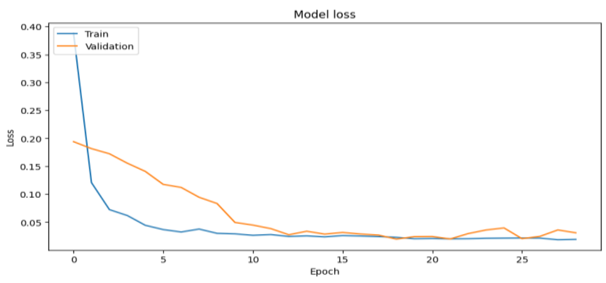
In the notebook, the user defined **'extra_frames'** variable refers to the number of frames to be predicted by the model after the initial **'seed_sequence'**, which serves as the starting point for generating extra frames, where 'seed_sequence' is any chosen sequence from validation set. The table below shows how well LSTM performs in generating spectrograms which are instrument-specific using the evaluation metrics discussed under section 4.2. Note that, the table depicts results when 'extra_frames' is set to 5, which means 5 frames will be predicted with each 4 sec long, so 4 x 5 = 20 sec audio clips. The value for the metrics will very much likely to change by tuning 'extra_frames' to different values.

EVALUATION METRICS						
APPROACH	INSTRUMENT	NO. OF PITCHES	PITCH RANGE	PITCH ENTROPY	FAD	KL DIVERGENCE
Approach 1	Guitar	19337	3833.8809	7.457	48236.392	0.066
	Mallet	33701	3843.244	7.914	12862.674	0.042
	Flute	82543	3843.695	8.154	106232.539	1.34
Approach 2	Guitar	1536	3327.1401	2.422	44329.987	0.395
	Mallet	1479	3445.3003	1.631	28081.952	0.429
	Flute	3863	3838.5535	5.565	94368.237	1.171

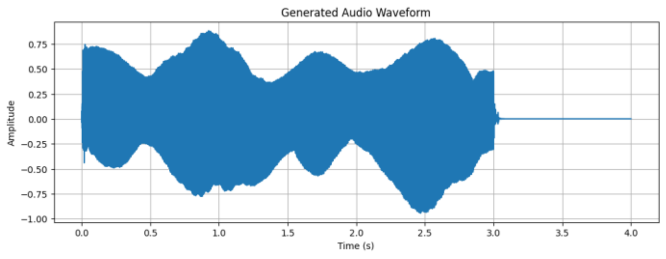
Table 2: Generated Audio Evaluation by LSTM

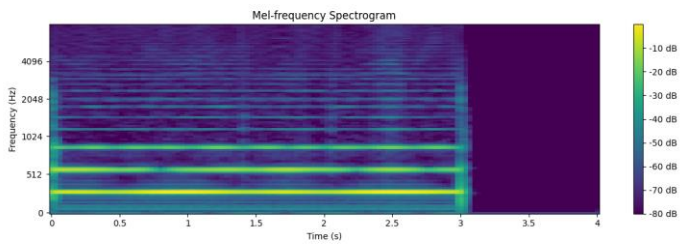
These are the metrics that indicate that Approach 1 has generated a higher number of pitches with greater pitch entropy over all the given instruments which indicates that generated audio has more diversity. Also, it has also resulted in higher FAD values for guitar and flute, which means it has a large deviation from real audio samples. Metrics from Approach 2 have generated fewer pitches, with low pitch entropy indicating less diversity in generated audio but resulting in lower FAD values for Mallet, which shows close similarity to real audio samples for the given instrument. The KL divergence varies across all the instruments and all the approaches, which highlights the differences in distribution of generated pitches compared to real audio samples. As part of human evaluation, friends and users were asked to evaluate by listening to the generated music, one common thing they suggested is to remove noise as it was difficult to hear instrument sounds clearly, removing noise will be part of our future work discussed under section 6.2.2.

5.3 Model – WaveNet Autoencoder



The graph depicts the training and validation losses of our WaveNet autoencoder model for 25 epochs. Initially, both losses fall substantially, suggesting the model's rapid understanding of the underlying acoustic patterns. As the epochs proceed, the validation loss gradually stabilizes, indicating that the model is honing its capacity to generalize to new data. A substantial difference between training and validation losses in the early stages shows some overfitting, but this gap narrows over time, showing increased generalization as the model adapts to the complexity of the audio input.





The provided spectrogram and waveform visuals depict outputs from a WaveNet autoencoder trained on organ audio samples. The spectrogram reveals that most energy is concentrated in the lower frequencies, which is characteristic of organ music with its deep, resonant tones. Listening to the audio sample, the audio sample is very close to the original tone with minimal noise.

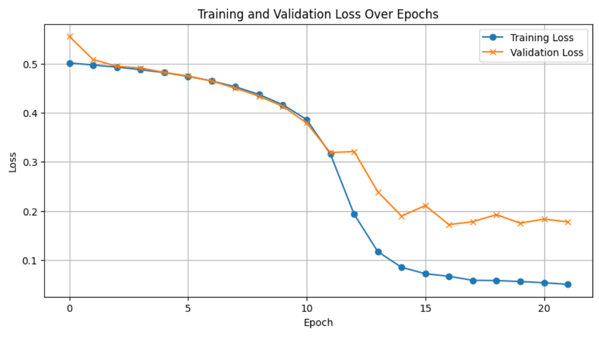
EVALUATION METRICS					
INSTRUMENT	NO. OF PITCHES	PITCH RANGE	PITCH ENTROPY	FAD	KL DIVERGENCE
Organ	290	954.2742	2.605	2531.7144	0.1544

5.4 Model – Conditional Variational Autoencoder

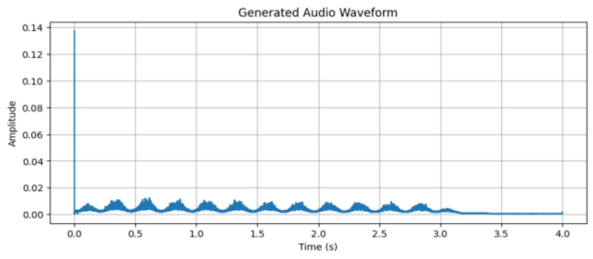
We have trained the Conditional Variational Autoencoder (CVAE) model three times for 50 Epochs using Early Stopping callback, focusing each session on a specific instrument—organ, guitar, and reed, in that order. As previously mentioned in se 3.4.3.1, the loss values are calculated using reconstruction loss and KL divergence loss. The Graphs below are the findings and visualizations of the results from these training sessions.

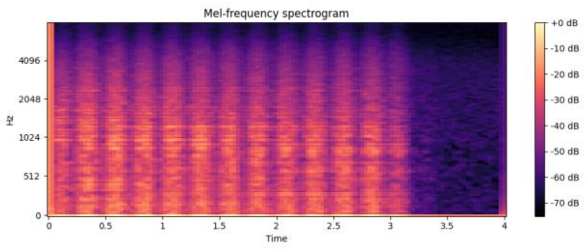
While the model was also run on the GTZAN dataset, it generated a lot of noise which completely made it hard to hear the faint musical features being generated, any attempt to clean up the noise also failed, leading us to make the decision to drop the gtzan dataset in favor of contining the testing on different instruments on the NSynth.

Organ Training and Audio Samples:

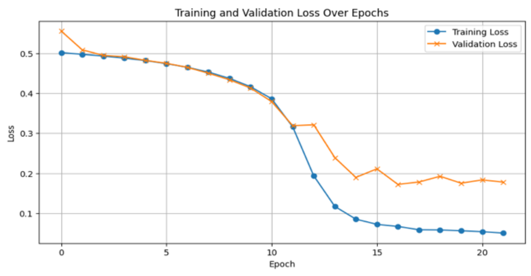


This graph, primarily focusing on organ instruments across 20 epochs, shows the training and validation loss values for the CVAE model. At first, both loss values drop quickly, showing that the model is successfully applying what it has learned from the training set to new data; however, as the training continues, the training loss continues to decrease, but the validation loss levels off and even shows a slight increase starting from the tenth epoch. This suggests that while the model is still improving, it may be starting to overfit.

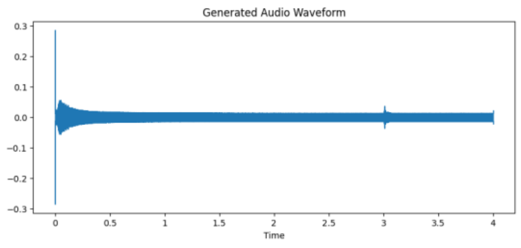


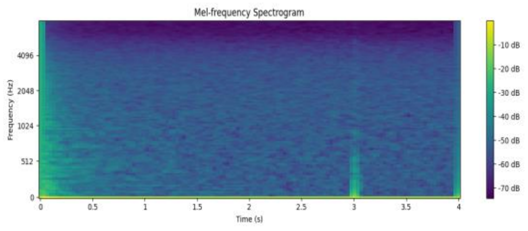


Guitar Training and Audio Samples:

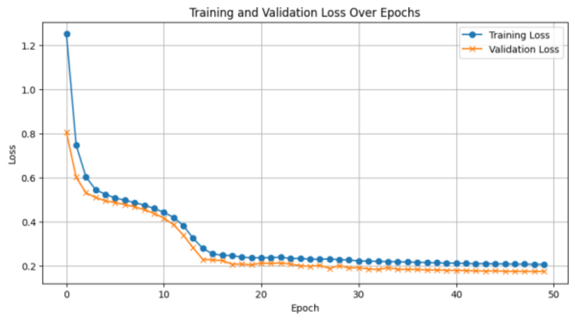


The graph displays the training and validation loss values over 20 epochs for the CVAE model being trained on the Guitar Instrument. The training proceeds the same as it did for the organ musical instrument.

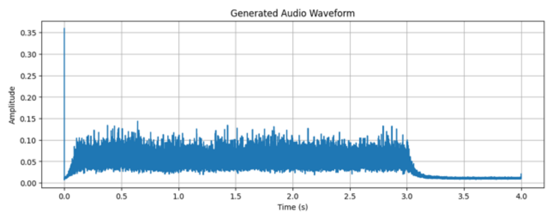


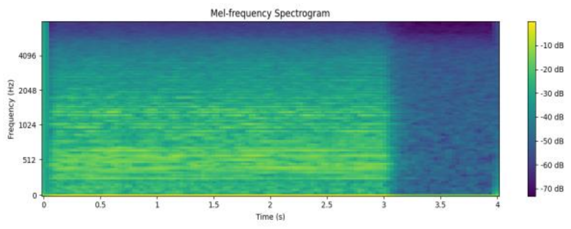


Reed Training and Audio Sample:



This graph displays the training and validation loss values over 50 epochs for the CVAE model being trained on the Reed Instrument, showing a significant improvement in convergence behavior compared to the earlier training samples. The validation loss closely tracks the training loss throughout the training process, showing that the model is not overfitting as it was in the previous training iteration.





Key Findings:

In all three models, we can observe the following key findings:

- At first, all models learn from the training data effectively, as seen by a large reduction in training and validation losses. The validation loss reaches a peak and starts to increase during the ninth epoch, while the training loss keeps going down.
- Overfitting can be seen.
- The model's capacity to generalize decreases as training progresses, as seen by the widening gap between training and validation losses. For some instruments, the model did not overfit as can be seen by the validation loss closely tracks the training loss throughout the training process for the reed instrument.
- The model found it harder to learn certain instruments, generating mildly accurate sounds for Guitar whereas, more accurate, as compared to guitar, results for reed or organ.

	EVALUATION METRICS	
INSTRUMENT	**FAD**	**KL DIVERGENCE**
Organ	22897.378	0.7201
Guitar	20942.8504	0.24104
Reed	15836.84	0.515

6 Conclusion

In conclusion, our multi-faceted approach to audio generation using GANs, LSTMs, and autoencoders yielded some progress but ultimately fell short of producing the high-quality, realistic audio we aimed for. The challenges we faced throughout the project have provided crucial learning experiences and highlighted key areas for future research and development. By addressing these challenges and building on our findings, we hope to make more substantial progress in the field of audio synthesis. Continued exploration and innovation in this area hold great potential for advancements and practical applications. Additionally, users can play the generated music through our Jupyter notebooks, offering an interactive way to engage with the results.

6.1 Summary of Work

Our exploration into audio generation has been a challenging yet educational journey. We employed a combination of Generative Adversarial Networks (GANs), Long Short-Term Memory networks (LSTMs), and autoencoders to tackle this complex problem. By focusing on a specific type of architecture tailored for this purpose, we were able to create realistic audio waveforms from random noise inputs. The intricate interplay between different approaches, each with their unique roles and architectures, forms the cornerstone of our model's success. While we made some progress, the results were not as realistic or high-quality as we had hoped. The generated audio exhibited noticeable artifacts and lacked the consistency of real samples, underscoring the hurdles we faced throughout the project.

6.1.1 Key Findings

- High Fidelity Audio Generation: Our model managed to produce audio waveforms from random noise inputs, but the output often lacked realism and contained artifacts. Despite extensive training, achieving the desired level of audio quality proved to be elusive.

- Efficient Use of Limited Data: Despite the resource constraints that limited our dataset to guitar samples, the model was able to generalize well, showcasing the robustness of our approach.
- Training Stability: By implementing techniques such as batch normalization, learning rate adjustments, and regularization, we managed to address common challenges in training, such as mode collapse and instability.

6.2 Limitations and Future Work

6.2.1 Limitations

Limitations	Proposed Solutions
The devices used for training were extremely limited and not originally meant for this purpose. • 3 M1 MacBook Pro's with 16 GB RAM • M2 MacBook Air with 8 GB RAM • Workaround we tried was Google Colab o Issues with loading dataset using G-Drive o Drive has a time out on free version o Limited GPU available i.e.,1 hour each day	Proper training would at least require a dedicated training setup with a dedicated Graphics card (GPU) which could train the model for days at a time. This is being proposed keeping in mind, loading in over 200k audio files, complex model architecture to capture all details of the audio, and training times over 1-2 days at minimum. For reference, Google used the following setups • 32 K40 GPU's for over 10 days (200k Epochs) for WaveNet • V100 GPU for over 5 days for GAN
Systems were completely unusable, while training models, for further research and coding	Having more allotted time or dedicated machines for training
Existing libraries for calculating metrics are mostly based around MIDI files with certain datasets in mind	We created custom solutions to calculate the metrics described earlier
The performance of the models heavily relies on the quality and diversity of the dataset. Limited quantity and variation in the dataset can severely hinder the model's ability to generalize across different styles of music and instrument families.	Current computational resources available for end-to-end users are not sufficient, when dealing with training deep learning models, even more so when dealing with audio data, as they require significant computational power.

6.2.2 Future Work

We had to severely limit our model’s architecture as per our computational resources available to us, severely limiting our progress and improvement in the model. Assuming we had the computational resources available, we would have proposed improving our models in the following ways:

- **Larger datasets and increased diversity**: As of now out of the 200k files in the training dataset, we could only manage to train the model on 500 to 1000 files at most which severely affecting its training and ability to learn to create innovative and coherent musical compositions. Expanding this dataset to include a larger variety of instrument families, genre and musical styles would provide the model with a more comprehensive understanding of music.
    - This will also prevent the current issue of model overfitting due to not having enough training samples.
- **Advanced Model Architectures:**
    - Experimenting with more complex architectures for our models such as the use of ResNet-style residual blocks, or Transformers within the encoder and decoder can improve the model’s ability to capture and generate intricate musical aspects and features.
    - Implementing more effective training techniques in all the models can prevent any further overfitting, stabilize and improve the training process and improve the generalization ability of the model. This will also allow us to utilize higher learning rates in the training process.

- **Longer training times**: Training models with more complex architectures, especially features which help reduce the noise would no doubt yield better results.
- **Interpolation:** Once we have a working model able to produce accurate and more coherent generated music, we plan to use interpolation techniques between these sounds to produce new and original musical compositions.

7. References

[1] Google LLC. (2021). "NSynth: A Large-Scale and High-Quality Dataset of Annotated Musical Notes". Available: https://www.tensorflow.org/datasets/catalog/nsynth

[2] J. Engel, Agrawal K. K., Chen, S., Gulrajani, I., Donahue, C., & Roberts, A, (2019). "GANSynth: Adversarial Neural Audio Synthesis". Available: https://arxiv.org/pdf/1902.08710v2

[3] J. Engel, C. Resnick, A. Roberts, S. Dieleman, D. Eck, K. Simonyan, and M. Norouzi, (2017). "Neural Audio Synthesis of Musical Notes with WaveNet Autoencoders". Available: https://arxiv.org/pdf/1704.01279.

[4] P. Dhariwal, H. Jun, C. Payne, J. W. Kim, A. Radford, and I. Sutskever, (2020) "Jukebox: A Generative Model for Music," 2020. Available: https://doi.org/10.48550/arXiv.2005.00341.

[5] Mangal, Sanidhya, Rahul Modak, and Poorva Joshi. "LSTM based music generation system." arXiv preprint arXiv:1908.01080 (2019). Available: https://arxiv.org/abs/1908.01080

[6] Andrada, (2020). "GTZAN Dataset - Music Genre Classification". Available: https://www.kaggle.com/datasets/andradaolteanu/gtzan-dataset-music-genre-classification

[7] Van den Oord, A., Dieleman, S., & Zen, H. (2016). "WaveNet: A Generative Model for Raw Audio". arXiv preprint arXiv:1609.03499. Available: https://arxiv.org/abs/1609.03499

[8] Alexander, E. (2024). "Variational Autoencoders (VAEs) in Generative Music: A New Era of Creativity - EmiTechLogic ". [online] EmiTechLogic - Bringing Technology to Life. Available: https://emitechlogic.com/variational-autoencoders-vaes-in-generative-music-a-new-era-of-creativity/

[9] PyPI. (n.d.). "Noisereduce: Noise reduction using Spectral Gating in python". Available: https://pypi.org/project/noisereduce/.

8. Appendix

8.1 Appendix A: Individual Contributions

8.1.1 Anand

Contributed to the research, development, testing and the reporting of the models: Generative Adversarial Network on nsynth datasets. Furthermore, in the report, I contributed to writing the sections: Model, Conclusion.

8.1.2 Nishant

Contributed to the research, development, testing and the reporting of the models: WaveNet and Conditional Variational Autoencoder on gtzan and nsynth datasets. Furthermore, in the report, contributed to writing the sections: Introduction, Limitations and Future Work.

8.1.3 Vishesh

Contributed to this project particularly in coding and researching about the music generation capability of LSTM. He also researched the properties of spectrograms and the methods that are involved in preprocessing and postprocessing them. At the start, he studied the sliding window approach to create sequences of audio files as LSTM's input and used two approaches to process the model's output. Other than that, he equally contributed with the team to find research papers that are linked to music generation domain.

In the report, Vishesh contributed to sections 3, 3.1, 3.3, 3.3.1, 3.3.2, 3.5, 5.1.2, 5.2, 7 and appendix.

8.1.4 Vishvesh

Contributed to the research and development of WaveNet AutoEncoders and the evaluation metrics for generated audio files. Was also involved in the research of feature extraction from audio files and encoding and decoding MIDI features. In the report, contributed to Related Work, Datasets, Evaluation Metrics and the Limitations sections.