

# **GENERATIVE ADVERSARIAL NETWORK ARCHITECTURE FOR MUSIC SYNTHESIS**

A project report submitted in partial fulfillment of the requirements for  
the award of the degree of

**B.Tech.**

**in**

**Electronics and Communication Engineering**

**by**

**SHRIKAR VENKATA TAMIRISA (620244)**

**ABBANABOINA MANJUNADH (620101)**

**GUJJARU MANOJ SATHWIK (620141)**



**Department of Electronics and Communication Engineering  
NATIONAL INSTITUTE OF TECHNOLOGY  
ANDHRA PRADESH-534101**

**APRIL 2024**

## **DECLARATION**

We declare that this written submission represents our ideas in our own words and where others' ideas or words have been included, we have adequately cited and referenced the original sources. We also declare that we have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in our submission. We understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Shrikar Venkata Tamirisa

Roll No.: 620244

Date: 7 May 2024

Abbanaboina Manjunadh

Roll No.: 620101

Date: 7 May 2024

Gujjaru Manoj Sathwik

Roll No.: 620141

Date: 7 May 2024

## **BONAFIDE CERTIFICATE**

This is to certify that the project titled **GENERATIVE ADVERSARIAL NETWORK ARCHITECTURE FOR MUSIC SYNTHESIS** is a bonafide record of the work done by

**SHRIKAR VENKATA TAMIRISA (620244)**

**ABBANABOINA MANJUNADH (620101)**

**GUJJARU MANOJ SATHWIK (620141)**

in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **ECE** of the **NATIONAL INSTITUTE OF TECHNOLOGY, ANDHRA PRADESH**, during the year 2023-2024.

**Dr. Thulasya Naik**

Project Guide

**Dr. S. Yuvaraj**

Head of the Department

## ABSTRACT

Generative machine learning models have been extensively applied to image and prompt response generation. The field of music generation though is a relatively uncharted territory. This can be attributed to factors such as inherent temporal dependencies of music, subjectivity, multi-modal nature, lack of explicit rules, high dimensionality, limited training data and requirement of high domain knowledge. This project aims to explore machine learning models to generate musical notes, in particular piano renditions. We start with LSTM networks which enable us to capture temporal dependencies, and later move on to Generative Adversarial Networks to combine definiteness of neural networks with capabilities of LSTM. A thorough analysis of the results obtained is done to compare generated music with the training dataset, including rendering of pitch class and note length transition matrices. Further, techniques such as Wasserstein loss function, random noise injection and L2 kernel regularization are adopted to deal with mode collapse of GAN and diversify the generated samples.

*Keywords:* **LSTM, GAN, WGAN, noise injection, regularization, mode collapse**

## ACKNOWLEDGEMENT

We would like to thank the following people for their support and guidance without whom the completion of this project in fruition would not be possible.

**Dr. Thulasya Naik**, our project guide, for helping and guiding us in the course of this project. His insightful guidance, constructive feedback, and unwavering support have been indispensable in navigating the complexities of the project and overcoming various challenges along the way.

**Dr. S. Yuvaraj**, the Head of the Department, Department of ECE. His visionary leadership and unwavering commitment to excellence have been instrumental in shaping the academic environment and fostering a culture of innovation within the department.

Our internal reviewers, **Dr. Puli Kishore Kumar, Dr. Kondalarao Jyothi, Dr. M. Ananda Reddy** for their insight and advice provided during the review sessions. Their collective efforts have played a significant role in shaping our academic journey and enriching our learning experience.

We would also like to thank our individual parents and friends for their constant support.

# TABLE OF CONTENTS

<b>Title</b>	<b>Page No.</b>
<b>ABSTRACT</b> .....	<b>iii</b>
<b>ACKNOWLEDGEMENT</b> .....	<b>iv</b>
<b>TABLE OF CONTENTS</b> .....	<b>v</b>
<b>LIST OF TABLES</b> .....	<b>vii</b>
<b>LIST OF FIGURES</b> .....	<b>viii</b>
<b>1 Introduction</b> .....	<b>1</b>
1.1 Motivation.....	2
1.2 Problem Statement.....	2
<b>2 Literature Survey</b> .....	<b>4</b>
<b>3 Proposed Methodology</b> .....	<b>11</b>
3.1 Long Short-Term Memory Networks.....	11
3.1.1 LSTM Internal Architecture .....	11
3.1.2 Architecture and training of LSTM for music generation .....	16
3.1.3 Temperature Sampling – hyperparameter in LSTM networks .....	17
3.1.4 Analysis and Limitations of LSTM .....	18
3.2 Generative Adversarial Network .....	18

3.2.1 Generic Architecture of GAN.....	19
3.2.2 Training Process of GAN .....	20
3.2.3 Core Principles of GAN.....	23
3.2.4 Architecture of GAN in our generative music synthesis model .....	23
3.3 Ways to combat mode collapse in GAN.....	<b>26</b>
3.3.1 Wasserstein GAN .....	27
3.3.2 Random Noise Injection .....	30
3.3.3 Regularization.....	31
3.4 Dataset and Preprocessing .....	31
<b>4 Simulation and Evaluation Parameters.....</b>	<b>32</b>
4.1 Classification Loss.....	32
4.2 Manhattan distance .....	33
4.3 Average pitch class transition matrix .....	33
4.4 Average note length transition matrix .....	33
4.5 Pitch .....	34
4.6 Kernel Density Estimate plots of intra-set distances .....	34
<b>5 Results.....</b>	<b>35</b>
<b>Conclusion and Future Scope .....</b>	<b>40</b>
<b>Bibliography .....</b>	<b>41</b>
<b>Appendices .....</b>	<b>44</b>
<b>A Code Attachments .....</b>	<b>45</b>

# List of Tables

Table 3.1: Proposed WGAN algorithm in literature.....	28
Table 5.1: Results.....	36



# List of Figures

Figure 3.1: RNN Repeating Module.....	11
Figure 3.2: LSTM repeating module .....	12
Figure 3.3: LSTM labelling .....	12
Figure 3.4: Cell State .....	12
Figure 3.5: Gates .....	13
Figure 3.6: Forget gate layer.....	14
Figure 3.7: Input gate layer.....	14
Figure 3.8: Output gate .....	15
Figure 3.9: LSTM Architecture .....	16
Figure 3.10: Generic GAN architecture .....	19
Figure 3.11: Training of discriminator .....	20
Figure 3.12: Training of Generator .....	21
Figure 3.13: Generator Architecture.....	23
Figure 3.14: Discriminator Architecture .....	24
Figure 3.15: Example of mode collapse for a GAN trained to produce handwritten digits .....	26
Figure 5.1: Noise Input.....	35
Figure 5.2: Piano Sample 1 .....	35
Figure 5.3: Piano Sample 2 .....	35
Figure 5.4: Loss vs Epoch for GAN .....	36
Figure 5.5: Loss vs Epoch for WGAN with noise and regularization.....	36
Figure 5.6: PCTM for original dataset .....	37
Figure 5.7: PCTM for generated dataset .....	37
Figure 5.8: NLTM for original dataset .....	38
Figure 5.9: NLTM for generated dataset .....	38
Figure 5.10: KDE plot for GAN .....	39
Figure 5.11: KDE plot for WGAN .....	39
Figure 5.12: KDE plot for WGAN with noise.....	39
Figure 5.13: KDE plot for WGAN with noise and regularization.....	39

# Chapter 1

## Introduction

Generative models stand as an indomitable force within the landscape of machine learning, representing a transformative approach to data representation and synthesis. At their core, these models seek to encapsulate the intricate distribution of a given dataset, thereby enabling the creation of novel samples that bear a striking resemblance to the original data. Unlike their discriminative counterparts, which are primarily concerned with delineating class boundaries and discerning patterns within the data, generative models operate on a grander scale, striving to capture the essence of the entire probability distribution underlying the dataset. This expansive scope endows generative models with a multifaceted utility that transcends conventional boundaries, finding applications across a diverse array of domains. From the synthesis of photorealistic images and the generation of coherent text passages to the composition of musical masterpieces and the discovery of novel drug compounds, generative models wield unparalleled versatility, serving as invaluable tools for a myriad of creative and analytical endeavors.

Generative machine learning, fueled by techniques like recurrent neural networks (RNNs), transformers, LSTMs and GANs can analyze vast datasets of existing music and learn the underlying patterns and structures. By understanding these patterns, they can generate new musical compositions that mimic the style and characteristics of the input data. From classical symphonies to modern electronic beats, generative models can produce diverse genres and styles, offering endless possibilities for musicians, composers, and enthusiasts alike. Moreover, these models allow for the exploration of entirely new sonic territories, pushing the boundaries of what we conceive as music.

In Chapter 1, the project is briefly introduced and the motivation behind working on the various techniques to generate music is discussed. In Chapter 2, literature survey of various papers which are helpful for this project are briefly discussed. In the Chapter 3, the detailed methodology is mentioned. In Chapter 4, important evaluation metrics that need to be analyzed to judge the performance of the models are described. In Chapter 5, all the obtained results are attached for the analysis purposes.

## 1.1 Motivation

In recent years, amidst the burgeoning advancements in generative modeling techniques, the domain of music generation has emerged as a captivating frontier ripe for exploration. Despite the remarkable strides witnessed in fields such as image synthesis and natural language processing, the realm of music composition remains relatively underexplored, presenting an untapped reservoir of creative potential and innovation. Recognizing the intrinsic allure and profound expressive power of music, our endeavor is dedicated to unlocking the latent capabilities of machine learning methodologies to revolutionize the landscape of musical composition. By delving deeply into the intricate interplay between data structures, temporal dependencies, and creative expression, we aspire to pave the way for a new era of musical exploration and innovation, where the boundaries of creativity are expanded and redefined through the lens of artificial intelligence.

## 1.2 Problem Statement

At the heart of our endeavor lies a profound appreciation for the temporal intricacies inherent in musical compositions, where the unfolding sequence of notes, rhythms, and harmonies imbues each piece with a unique identity and emotional resonance. Central to our approach is the endeavor to capture and emulate these temporal dependencies within the framework of generative models, thereby imbuing our musical compositions with a sense of coherence, continuity, and expressive depth. By meticulously analyzing the underlying structure of musical compositions and distilling the essence of temporal relationships, we seek to empower our generative models with the ability to compose music that transcends mere imitation, evoking genuine emotion and artistic inspiration.

Our exploration of generative modeling techniques is characterized by a comprehensive examination of deep learning architectures tailored specifically for music generation. In our quest for innovation, we delve deeply into the realm of Long Short-Term Memory (LSTM) networks and Generative Adversarial Networks (GANs), two prominent paradigms that hold immense promise in the domain of music composition. With LSTM networks, renowned for

their ability to capture long-range dependencies and sequential patterns, we seek to unravel the intricate tapestry of musical compositions, decoding the underlying structure and infusing our generative models with a nuanced understanding of musical expression. Concurrently, our exploration of GANs heralds a new frontier in music generation, where the adversarial interplay between generator and discriminator networks gives rise to a dynamic and iterative process of creative synthesis. By harnessing the latent potential of GANs to synthesize diverse and compelling musical compositions, we aim to transcend the limitations of traditional generative models, ushering in a new era of creative expression and artistic innovation.

Central to our exploration of GANs is the quest to overcome challenges such as mode collapse, a phenomenon where the generator network converges to a limited set of output samples, thereby diminishing the diversity and richness of the generated compositions. To address this challenge, we embark upon a multifaceted approach that encompasses the deployment of sophisticated techniques and methodologies aimed at enhancing the robustness and diversity of our generative models. One such technique involves the adoption of Wasserstein loss functions, which provide a more stable and informative training objective compared to traditional loss functions used in GANs. By leveraging the Wasserstein distance metric, we aim to imbue our generative models with a deeper understanding of the underlying data distribution, thereby fostering the creation of more diverse and authentic musical compositions.

Additionally, we explore the strategic integration of noise injection strategies, which serve to introduce stochasticity and variability into the generative process, thereby mitigating the risk of mode collapse and enhancing the richness and diversity of the generated compositions. Through judicious experimentation and meticulous fine-tuning, we seek to strike a delicate balance between coherence and creativity, allowing our generative models to synthesize musical compositions that evoke genuine emotion and artistic inspiration. Furthermore, we investigate the application of L2 regularization techniques, which serve to mitigate the risk of overfitting and promote generalization in our generative models. By imposing constraints on the complexity of the model parameters, we aim to foster a more robust and adaptable generative process, capable of synthesizing musical compositions that exhibit both fidelity to the original data distribution and creative innovation.

# Chapter 2

## Literature Survey

Traditional methods, such as Back-Propagation Through Time (BPTT, Williams, Zipser et al 1992 [2]) and Real-Time Recurrent Learning (RTRL, Robinson, Fallside et al 1987 [3]), often struggle with vanishing or exploding error signals over long sequences “flowing backwards in time”, impeding the network's ability to effectively learn temporal dependencies. Hochreiter et al (1991) [4] discuss about a method in which the temporal evolution of the backpropagated error exponentially depends on the size of the weights. Blowing up of error signal may lead to oscillating weights, while in case of vanishing gradients the learning to bridge long time lags takes a prohibitive amount of time, or does not work at all.

Hochreiter et al, 1997 [1] presents “Long Short-Term Memory” (LSTM), a novel recurrent network architecture in conjunction with an appropriate gradient-based learning algorithm. LSTM is designed to overcome these error back-flow problems. It can learn to bridge time intervals in excess of 1000 steps even in case of noisy, incompressible input sequences, without loss of short time lag capabilities. This is achieved by an efficient, gradient-based algorithm for an architecture enforcing constant (thus neither exploding nor vanishing) error flow through internal states of special units (provided the gradient computation is truncated at certain architecture-specific points – this does not affect long term error flow though).

However, LSTM networks are not without their limitations. They are known to be sensitive to initialization and hyperparameters, consume high amounts of memory, and have difficulty capturing extremely long-term dependencies. These limitations can pose challenges in training and may result in inconsistent output lengths, particularly in tasks like music generation. Despite these challenges, LSTM networks remain a powerful tool for sequence modeling and have found applications in various domains.

Previous generative models suffered with inability to capturing complex dependencies such as high-dimensional and multimodal data distributions, lack of adversarial training framework, less diversity in output distributions and struggled with generalization, especially when

trained on small or homogeneous datasets. Moreover, realistic image generation was challenging for traditional sequence-based models like LSTMs or RNNs.

A new framework was proposed by Goodfellow et al (2014) [5] for estimating generative models via an adversarial process, in which we simultaneously train two models: a generative model  $G$  that captures the data distribution, and a discriminative model  $D$  that estimates the probability that a sample came from the training data rather than  $G$ . The training procedure for  $G$  is to maximize the probability of  $D$  making a mistake. This framework corresponds to a minimax two-player game. In the space of arbitrary functions  $G$  and  $D$ , a unique solution exists, with  $G$  recovering the training data distribution and  $D$  equal to  $1/2$  everywhere. In the case where  $G$  and  $D$  are defined by multilayer perceptron, the entire system can be trained with backpropagation. There is no need for any Markov chains or unrolled approximate inference networks during either training or generation of samples. Experiments demonstrate the potential of the framework through qualitative and quantitative evaluation of the generated samples.

One of the most well-known limitations of GANs is mode collapse, where the generator learns to produce a limited variety of samples, ignoring the full diversity of the target distribution. This results in generated samples that lack diversity and do not cover the entire data space. Also, GAN training can be notoriously unstable, with the generator and discriminator networks oscillating between different equilibria during training. This instability can lead to difficulties in convergence, slow training, or even failure to converge altogether. Controlling the quality of generated samples in GANs can be difficult, especially when fine tuning the trade-off between sample quality and diversity. Balancing these factors often requires careful tuning of hyperparameters and training procedures.

WGANs, introduced by Arjovsky et al (2017) [6] cure the main training problems of GANs. In particular, training WGANs does not require maintaining a careful balance in training of the discriminator and the generator, and does not require a careful design of the network architecture either. The mode dropping phenomenon that is typical in GANs is also drastically reduced. One of the most compelling practical benefits of WGANs is the ability to continuously estimate the EM distance by training the discriminator to optimality. Plotting these learning curves is not only useful for debugging and hyperparameter searches, but also correlate remarkably well with the observed sample quality.

Wasserstein Generative Adversarial Network (WGAN) utilizes Wasserstein distance, or Earth Mover's distance, as a metric for dissimilarity between real and generated data distributions,

providing a more stable training objective compared to traditional GANs. By employing a critic network, trained to approximate Wasserstein distance by scoring real and generated samples, WGAN fosters informative gradients for both generator and critic. Through iterative updates, WGAN minimizes Wasserstein distance, offering stability and mitigating issues like mode collapse or training instability. WGAN introduces weight clipping or gradient penalty techniques to enforce Lipschitz continuity, stabilizing training and improving sample quality. By optimizing Wasserstein distance, WGAN encourages the generator to produce samples closer to the true data distribution, facilitating the generation of high-quality and diverse samples, particularly beneficial for tasks like image generation. Overall, WGANs offer a principled and effective approach to training generative models, advancing the field with stable training dynamics and improved sample generation capabilities.

The choice of hyperparameters, such as the clipping threshold or penalty coefficient, can significantly affect the performance of WGANs, requiring careful tuning. The gradient clipping or penalty techniques introduced in WGANs can lead to the vanishing or exploding gradient problem.

The technical report authored by Mitzenmacher, Owen et al (2000) [7] addresses the challenge of estimating the resemblance between MIDI (Musical Instrument Digital Interface) documents. MIDI files contain musical information such as notes, tempo, and instrument data, and comparing these files is essential for tasks like music recommendation systems, plagiarism detection, and music analysis. The authors propose a methodology for estimating the resemblance of MIDI documents by grouping adjacent pitches together and calculating the Manhattan distance between these pitch groups.

While the proposed methodology offers a straightforward approach to estimating resemblance, it may not capture more nuanced aspects of musical similarity, such as rhythm and dynamics. Additionally, the effectiveness of the approach may vary depending on factors such as the choice of distance metric and preprocessing techniques. Therefore, further research is needed to develop more comprehensive and accurate techniques for music comparison and analysis.

The paper by Feng, Zhao, Zha, et al (2020) [10] investigates the role of noise injection in improving the performance of generative adversarial networks (GANs). The problem addressed in the study revolves around enhancing the stability and diversity of GAN-generated samples, which often suffer from mode collapse and lack of variety. The methodology employed involves injecting Gaussian noise with carefully chosen parameters into the input or hidden layers of both the generator and discriminator networks. This noise

injection aims to encourage the models to explore a wider range of data distributions and produce more diverse and realistic outputs. However, while noise injection shows promise in mitigating mode collapse and improving sample quality, it also introduces additional complexity to the training process and may require fine-tuning of noise parameters to achieve optimal results. Moreover, excessive noise injection can lead to overfitting or destabilize the training process, highlighting the need for careful experimentation and validation of noise injection techniques in GANs.

The paper by Lee, Seok et al (2020) [11] provides an overview of recent studies on regularization methods for generative adversarial networks (GANs). The problem addressed in the study revolves around improving the stability, convergence, and generalization capabilities of GAN models, which often suffer from issues such as mode collapse, training instability, and overfitting. The methodology involves reviewing and summarizing various regularization techniques proposed in recent literature, including weight regularization, gradient penalties, spectral normalization, and consistency regularization. These regularization methods aim to impose constraints on the GAN's parameter space or loss function to encourage smoother training dynamics, reduce overfitting, and enhance the model's ability to generalize to unseen data. However, while regularization techniques show promise in addressing common challenges faced by GANs, they also come with their own limitations. For instance, some regularization methods may introduce additional hyperparameters that require tuning, leading to increased computational complexity and training time. Moreover, the effectiveness of regularization techniques may vary depending on the specific GAN architecture, dataset characteristics, and optimization settings, highlighting the need for careful evaluation and empirical validation in practical applications.

[3] Robinson and Fallside (1987) introduced a utility-driven dynamic error propagation network as a means to enhance the efficiency and accuracy of error propagation within neural networks. Their methodology focused on the dynamic adjustment of error signals during the training process to optimize learning outcomes. By incorporating utility-driven adjustments, the network aimed to prioritize and allocate resources effectively, thereby leading to more efficient error propagation and learning. This approach enabled the network to adaptively allocate resources based on the utility of error signals, allowing for more effective learning and convergence. However, limitations may arise when the network encounters complex or high-dimensional data, potentially resulting in difficulties in effectively propagating errors and optimizing learning outcomes. Addressing these limitations may require further research to explore adaptive techniques and mechanisms that can better handle diverse datasets and



learning scenarios. Additionally, investigating the scalability and generalization capabilities of the proposed approach across various tasks and domains could provide insights into its broader applicability in real-world settings.

[4] Sepp Hochreiter et al diploma thesis (1991) delved into the realm of dynamic neural networks, with a particular emphasis on addressing the challenge of training networks with long-term dependencies. Hochreiter proposed the Long Short-Term Memory (LSTM) architecture, which introduced memory cells and gating mechanisms to better capture long-range dependencies in sequential data. The LSTM architecture effectively tackled the vanishing gradient problem commonly encountered in traditional recurrent neural networks, enabling more effective training over extended sequences. By incorporating memory cells and gating mechanisms, LSTM networks could selectively retain and utilize information over time, facilitating the learning of long-term dependencies. While LSTM networks have gained widespread adoption and proven effective in various applications, they may still face challenges in capturing dependencies across exceptionally long sequences or dealing with extremely complex data. Further exploration is necessary to devise enhancements to LSTM architectures, such as improved gating mechanisms or attention mechanisms, to address these challenges and enhance their scalability and robustness for various applications.

[8] Yang et al (2020) undertook an evaluation of “generative models in the domain of music”, with a focus on assessing their ability to produce realistic and diverse musical compositions. Their methodology encompassed the assessment of various metrics, including pitch class transition matrices, to gauge the quality and diversity of the generated music. By scrutinizing the distribution of pitch transitions, they aimed to quantitatively measure the degree of musical creativity and diversity captured by the generative models. However, limitations may arise in accurately evaluating the subjective aspects of musical creativity and expression using quantitative metrics alone. Future research endeavors could delve into the exploration of novel evaluation metrics and methodologies, aiming to better encapsulate the nuanced qualities of musical compositions generated by these models and ensuring a more comprehensive assessment of their capabilities. Additionally, investigating the integration of qualitative evaluation methods, such as expert assessments or user studies, could provide complementary insights into the perceived quality and creativity of generated music.

[9] Li et al (2021) proposed a “Novel LSTM-GAN” architecture tailored for music generation, which seamlessly integrated LSTM networks with GANs to capture long-term

dependencies and generate realistic music samples. The methodology involved training the LSTM-GAN to produce music samples closely resembling the training data. By harnessing the discriminative power of the GAN discriminator and the sequential modeling capabilities of the LSTM generator, the proposed architecture aimed to yield high-quality and diverse musical compositions. Nevertheless, challenges may arise in effectively capturing the complexity and variability of musical compositions, particularly in encapsulating the subtle nuances of musical expression. Further investigations could explore advanced architectural designs and training methodologies to enhance the fidelity and diversity of the generated music samples, thereby advancing the capabilities of LSTM-GANs in music generation tasks. Exploring techniques such as hierarchical modeling or incorporating domain-specific constraints could offer avenues for improving the realism and diversity of the generated music samples, contributing to the advancement of music generation technologies.

[2] In their seminal work, Williams et al (1995) embarked on a meticulous exploration of gradient-based learning algorithms tailored explicitly for recurrent neural networks (RNNs), which posed unique computational challenges due to their inherent recurrent connections. Central to their endeavor was the development of efficient training methodologies, with a particular emphasis on the intricate process of backpropagation through time (BPTT) within recurrent architectures. By unraveling the complexities of gradient expressions specific to RNNs and probing various optimization strategies, they aimed to streamline the training process and alleviate computational bottlenecks. Their endeavor delved deep into the computational intricacies underlying the proposed algorithms, meticulously dissecting the computational complexity to unearth strategies that could render the training of RNNs more efficient and scalable. However, translating these theoretical advancements into practical applications may encounter hurdles, especially when confronted with larger datasets or more complex network architectures, necessitating further optimization efforts to mitigate computational overhead. Hence, future research endeavors could pivot towards fine-tuning and refining the proposed methodologies to enhance their scalability and applicability in real-world settings

In the paper "Improved training of Wasserstein GANs," Gulrajani et al [23] address the issue of training instability in Wasserstein GANs (WGANs), a powerful type of Generative Adversarial Network (GAN). While WGANs offered significant progress, they could sometimes generate low-quality samples or fail to converge. The authors identify the culprit as the weight clipping strategy used in WGANs to enforce a Lipschitz constraint on the critic,

which can lead to undesirable training behavior. Their proposed solution replaces weight clipping with a gradient penalty. This method penalizes the norm of the critic's gradient with respect to its input, encouraging smoother critic updates and providing the generator with more informative gradients. This methodology demonstrably improved WGAN training, enabling stable training across various GAN architectures with minimal hyperparameter tuning and achieving high-quality image generation on datasets like CIFAR-10 and LSUN bedrooms. However, the gradient penalty approach also has limitations. It requires careful selection of a hyperparameter that controls the strength of the penalty, and choosing an inappropriate value can lead to training instability.

Donahue et al. (2019) [12] address the challenge of unsupervised audio synthesis by proposing WaveGAN, a novel application of Generative Adversarial Networks (GANs) to generate raw audio waveforms directly. Prior to this work, GANs had primarily been utilized for image generation. The authors acknowledge the inherent difficulty in audio synthesis due to the high temporal resolution of audio signals and the need to capture structure across various timescales. WaveGAN tackles this problem by employing a GAN architecture specifically designed for raw waveforms. This methodology demonstrates the ability to synthesize one-second audio segments with global coherence, making it suitable for sound effect generation. While the experiments showcase promising results, including the generation of intelligible words from a small speech dataset, limitations exist. The current iteration focuses on short audio clips and may not be suitable for longer, more complex audio generation tasks.

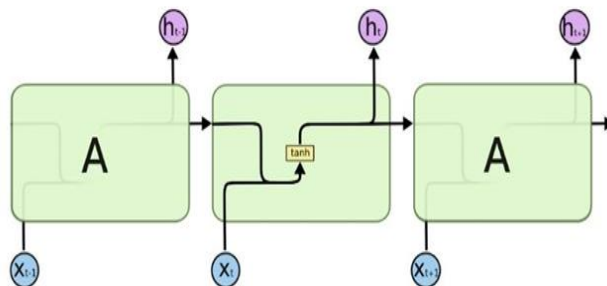
# Chapter 3

## Proposed Methodology

### 3.1 Long Short-Term Memory Networks

#### 3.1.1 LSTM Internal Architecture

Long Short-Term Memory networks – usually just called LSTMs – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter & Schmidhuber (1997) [1], and were refined and popularized by many people in following work. They work tremendously well on a large variety of problems, and are now widely used. LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn. All recurrent neural networks have the form of a chain of repeating modules of neural network, as shown in [Fig. 3.1](#). In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.



*Figure 3.1: RNN Repeating Module*

LSTMs also have this chain like structure, but the repeating module has a different structure, as shown in Fig 3.2. Instead of having a single neural network layer, there are four, interacting in a very special way.

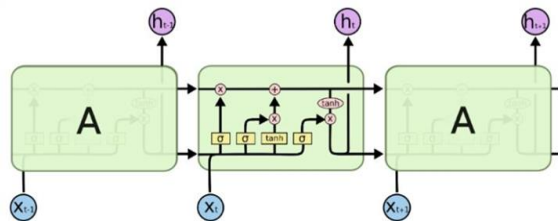


Figure 3.2: LSTM repeating module

In the below Fig 3.3, each line carries an entire vector, from the output of one node to the inputs of others. The pink circles represent point wise operations, like vector addition, while the yellow boxes are learned neural network layers. Lines merging denote concatenation, while a line forking denotes its content being copied and the copies going to different locations.

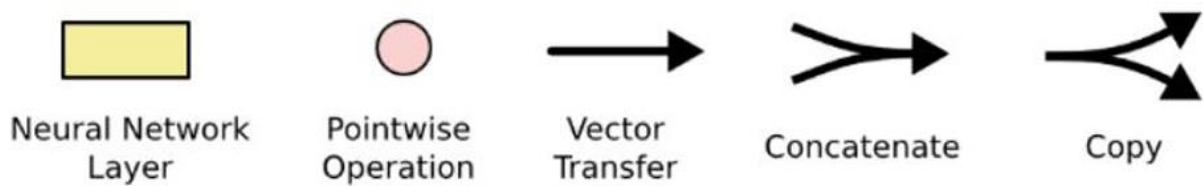


Figure 3.3: LSTM labelling

The key to LSTMs is the cell state, the horizontal line running through the top of Fig 3.4. The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It is very easy for information to just flow along it unchanged.

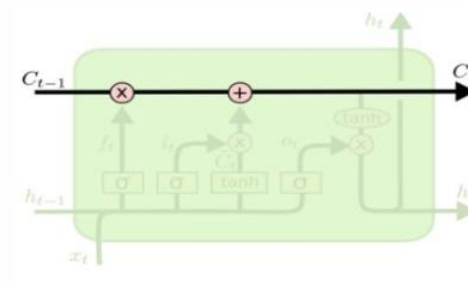
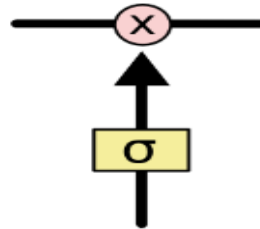


Figure 3.4: Cell State

The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates. Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a point wise multiplication operation, as shown in [Fig 3.5](#).



*Figure 3.5: Gates*

The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means “let nothing through,” while a value of one means “let everything through!” An LSTM has three of these gates, to protect and control the cell state.

The first step in our LSTM is to decide what information we are going to throw away from the cell state. This decision is made by a sigmoid layer called the “forget gate layer”, as shown in [Fig 3.6](#). It looks at  $h_{t-1}$  and  $x_t$ , and outputs a number between 0 and 1 for each number in the cell state  $C_{t-1}$ . A 1 represents “completely keep this” while a 0 represents “completely get rid of this.”

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

Let us go back to our example of a language model trying to predict the next word based on all the previous ones. In such a problem, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.

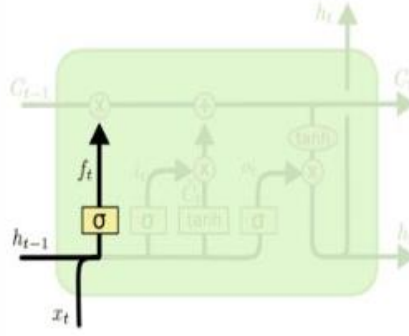


Figure 3.6: Forget gate layer

The next step is to decide what new information we’re going to store in the cell state. This has two parts. First, a sigmoid layer called the “input gate layer”, as shown in Fig. 3.7 decides which values we’ll update. Next, a tanh layer creates a vector of new candidate values,  $\tilde{C}_t$ , that could be added to the state. In the next step, we’ll combine these two to create an update to the state.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

In the example of our language model, we’d want to add the gender of the new subject to the cell state, to replace the old one we’re forgetting.

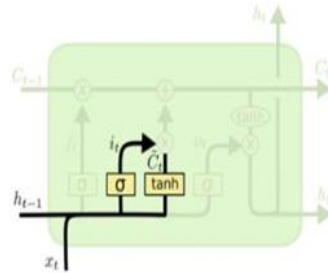


Figure 3.7: Input gate layer

It’s now time to update the old cell state,  $C_{t-1}$ , into the new cell state  $C_t$ . The previous steps already decided what to do, we just need to actually do it. We multiply the old state by  $f_t$ , forgetting the things we decided to forget earlier. Then we add  $i_t * \tilde{C}_t$ . This is the new candidate values, scaled by how much we decided to update each state value. In the case of the language model, this is where we’d actually drop the information about the old subject’s gender and add the new information, as we decided in the previous steps. Finally, we need to

decide what we're going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanh (to push the values to be between  $-1$  and  $1$ ) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to, as shown in Fig 3.8.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh(C_t)$$

For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next.

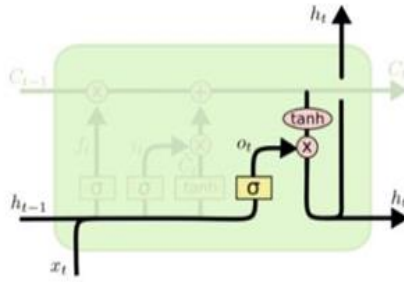


Figure 3.8: Output gate

In addition to the core functionalities of the LSTM described above, there are several additional aspects and variations that contribute to its versatility and effectiveness in various applications. One notable extension is the concept of peephole connections, where the gates in the LSTM can have direct access to the cell state, allowing them to better capture long-term dependencies. This modification enhances the model's ability to retain relevant information over longer sequences. Furthermore, techniques such as layer normalization and recurrent dropout can be applied to regularize and stabilize the training process, preventing overfitting and improving generalization performance.



### 3.1.2 Architecture and training of LSTM for music generation

In this configuration, a single LSTM layer with 256 units and equipped with a 0.2 dropout rate is employed, leveraging the power of recurrent neural networks (RNNs) to capture temporal dependencies within the input data. The choice of using LSTM (Long Short-Term Memory) layers is particularly beneficial for modeling sequential data, such as time series or text, due to their ability to retain information over extended time intervals. The incorporation of dropout helps prevent overfitting by randomly dropping a fraction of the LSTM units during training, encouraging the network to learn robust representations. The architecture is shown in Fig 3.9.

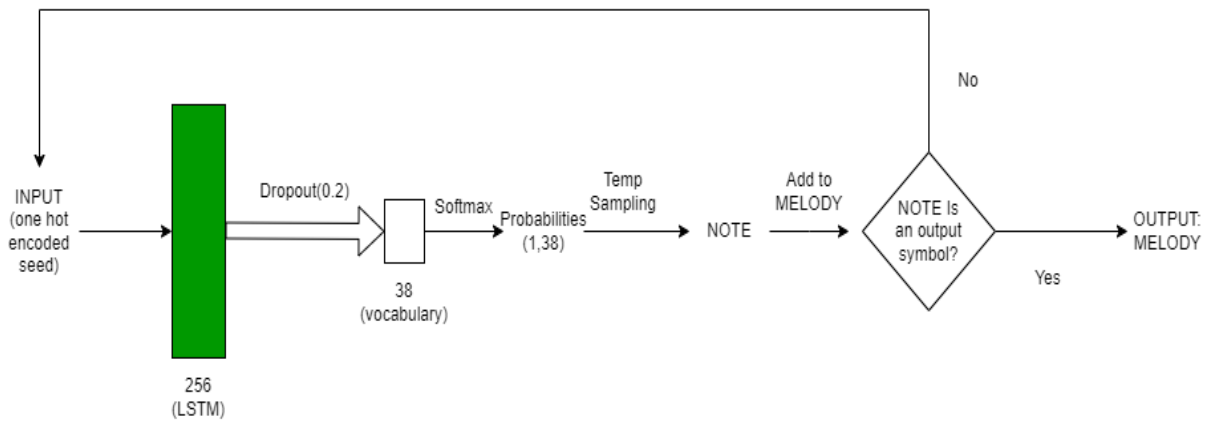


Figure 3.9: LSTM Architecture

For the output layer of 38 units, the softmax activation function is applied, enabling the model to produce probability distributions across multiple classes or categories. This is particularly useful in classification tasks, where the network needs to assign a probability to each class to make predictions. The softmax function ensures that the output values are normalized and interpretable as probabilities, facilitating decision-making based on the highest probability class.

The optimization process is driven by the Adam optimizer with a learning rate of 0.001, a popular choice known for its adaptive learning rate approach and efficient handling of sparse gradients. By adjusting the learning rate during training based on the past gradients' magnitude, Adam optimizes convergence speed and stability, leading to faster and more reliable model training. The training process spans 50 epochs, representing the number of times the entire dataset is passed forward and backward through the neural network.

Leveraging a T4 GPU for training accelerates the computational workload, allowing for faster

iterations and model experimentation.

Training involves giving a sequence of notes as input to the network and requiring it to predict the next note, using suitable back propagation to update the weights if there is a mismatch. Sparse Categorical Cross entropy is used as the loss function. Generation of melody involves a step-by-step process. It starts with the seed, which is processed by the LSTM and it predicts a note which can be appended to the seed, like NLP models. If the note is not an end character, it is appended to the seed, and is fed back to the network after one hot encoding.

Furthermore, the choice of utilizing a single LSTM layer with dropout and softmax activation reflects a balance between model complexity and computational efficiency. While deeper architectures might offer more capacity to capture intricate patterns in the data, a single LSTM layer suffices for many tasks, especially when dealing with moderate-sized datasets like the Erk subset of KernScores. Additionally, the incorporation of dropout helps mitigate the risk of overfitting, ensuring that the model generalizes well to unseen data.

### 3.1.3 Temperature Sampling – hyperparameter in LSTM networks

The output array comprises 38 values between 0 and 1, representing probabilities (softmax) for each note mapped to its corresponding index number. Typically, selecting the index with the highest probability yields the output note. However, introducing creativity into the melody involves considering notes with lower probabilities, adding an element of "unexpectedness" to the sequence. By occasionally choosing notes with lower probabilities, novel and more innovative melodies can be generated, enhancing the musical composition with unpredictability and uniqueness.

```
predictions = np.log(probabilites) / temperature
probabilites = np.exp(predictions) / np.sum(np.exp(predictions))

choices = range(len(probabilites))
index = np.random.choice(choices, p=probabilites)
```

### **3.1.4 Analysis and Limitations of LSTM**

The model exhibits erratic behavior, generating melodies with varying lengths for each output, which can result in unnatural compositions when forced to conform to a predetermined length. These melodies often display minimalist characteristics, featuring simplistic structures and low complexity. Despite efforts to enforce a specific length, the model's output may deviate from expected patterns, leading to compositions that lack coherence and fluidity. This erratic behavior poses challenges in achieving consistent and aesthetically pleasing musical results. Hence, we moved on to GAN architectures.

## **3.2 Generative Adversarial Network**

Generative Adversarial Networks (GANs) have emerged as a powerful paradigm in the field of artificial intelligence, revolutionizing the way machines generate new data samples. Introduced by Ian Goodfellow, et al (2014) [5], GANs are a class of machine learning algorithms that learn to generate data by pitting two neural networks against each other in a competitive setting.

At the heart of a GAN are two primary components: the generator and the discriminator. The generator network takes random noise as input and attempts to produce data samples that are indistinguishable from real ones. Conversely, the discriminator network learns to differentiate between real data samples and those generated by the generator. Through adversarial training, these two networks engage in a continuous game of one-upmanship, with the generator striving to produce increasingly realistic samples while the discriminator becomes more adept at distinguishing real from fake.

The GAN structure involves the generator learning to produce realistic data, which is then evaluated by the discriminator. The discriminator's role is to differentiate between real and fake data, providing feedback to the generator for improvement. This process continues iteratively until the generator can create data that fools the discriminator. GANs have since evolved with different architectures like Deep Convolutional GANs (DCGAN) and Cycle GAN, each tailored for specific tasks such as high-resolution image generation or style transformation between images.

### 3.2.1 Generic Architecture of GAN

The architecture of a Generative Adversarial Network (GAN) consists of two main components: the generator network and the discriminator network. These networks are trained simultaneously in a competitive manner, where the generator aims to produce realistic samples, and the discriminator aims to distinguish between real and fake samples.

#### Generator Network:

The generator network takes random noise as input, typically drawn from a simple distribution such as Gaussian or uniform distribution.

It consists of one or more layers of neural network units, usually implemented using convolutional or fully connected layers.

The purpose of the generator is to transform the input noise into meaningful samples that resemble the training data distribution.

The output of the generator is a generated sample, such as an image, audio clip, or text sequence.

#### Discriminator Network:

The discriminator network is tasked with distinguishing between real data samples from the training set and fake samples generated by the generator.

Like the generator, the discriminator also consists of one or more layers of neural network units. It takes as input either a real data sample or a fake sample generated by the generator, as shown in Fig 3.10. The discriminator outputs a probability score indicating the likelihood that the input sample is real (1) or fake (0).

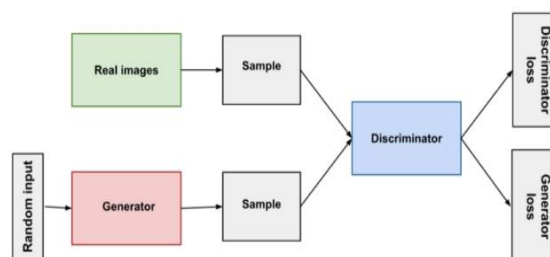


Figure 3.10: Generic GAN architecture

During training, the generator and discriminator networks are trained iteratively in a minimax game framework. The generator aims to maximize the probability that the discriminator incorrectly classifies its generated samples as real, while the discriminator aims to correctly classify real and fake samples. This adversarial training process leads to the generator learning to produce increasingly realistic samples, while the discriminator improves its ability to distinguish between real and fake data.

The architecture of GANs can vary depending on the specific application and domain. Researchers have proposed various modifications and enhancements to the basic GAN architecture, including techniques such as conditional GANs, Wasserstein GANs, and progressive GANs, to address challenges such as mode collapse, training instability, and mode dropping. Overall, the architecture of GANs represents a powerful framework for generating high-quality synthetic data and has led to significant advancements in the field of deep learning and artificial intelligence.

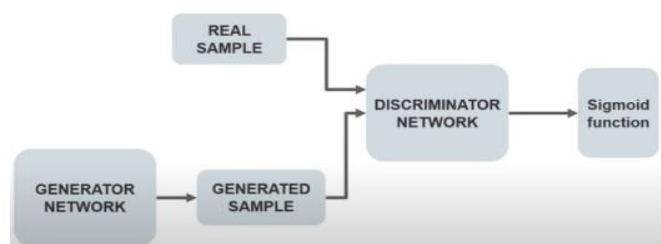
### 3.2.2 Training process of GAN

The training process of a Generative Adversarial Network (GAN) typically unfolds in two distinct phases, each focusing on training one of the two primary components: the discriminator and the generator.

#### Phase 1: Discriminator Training

During the initial phase, the discriminator network is trained while keeping the generator network frozen or fixed.

The training dataset, containing both real and generated samples, is fed into the discriminator, as shown in Fig 3.11. Real samples are labeled as true (1), while generated samples are labeled as false (0).



*Figure 3.11: Training of discriminator*

The discriminator learns to distinguish between real and fake samples by adjusting its parameters through backpropagation and gradient descent optimization to minimize the classification error.

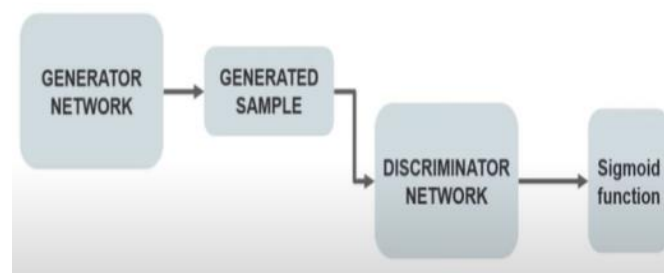
Meanwhile, the generator remains inactive during this phase, as its parameters are not updated. Instead, it serves the purpose of providing input for the discriminator to evaluate.

### Phase 2: Generator Training

Once the discriminator has been trained and achieved some level of discrimination accuracy, the focus shifts to training the generator.

In this phase, the generator network is unfrozen, while the discriminator is kept fixed. The generator now aims to improve its ability to generate realistic samples that can effectively fool the discriminator.

The generator takes random noise vectors as input and generates synthetic samples, such as images, music, or text sequences, as shown in [Fig 3.12](#).



*Figure 3.12: Training of Generator*

The generated samples are then fed into the fixed discriminator, and the resulting outputs (probabilities of authenticity) are used to compute the generator's loss.

Through backpropagation and optimization, the generator adjusts its parameters to minimize this loss, effectively learning to produce more realistic samples that are challenging for the discriminator to differentiate from real ones.

This adversarial training process continues iteratively, with alternating phases of discriminator and generator training, until both networks reach a state of equilibrium where the generator produces high-quality samples indistinguishable from real data.

By iteratively training and updating the discriminator and generator networks in alternating

phases, GANs achieve a dynamic equilibrium where the generator becomes increasingly adept at generating realistic samples, while the discriminator becomes more discerning in distinguishing between real and fake samples.

During the training process, the discriminator and generator engage in a continual game of one-upmanship. As the discriminator becomes more proficient at distinguishing between real and generated samples, it provides increasingly accurate feedback to the generator, pushing it to generate more realistic outputs. This dynamic feedback loop drives the generator to continually improve its ability to synthesize samples that resemble genuine data, while the discriminator adapts to the evolving nature of the generated samples, becoming more discerning in its evaluation.

Conversely, as the generator improves its ability to produce convincing samples, it challenges the discriminator to become more discerning and adapt to these evolving synthetic samples. This iterative adversarial interplay drives both networks to refine their respective strategies, with the generator striving to produce samples that are indistinguishable from real data and the discriminator enhancing its discriminatory capabilities to accurately classify between real and generated samples.

#### Adversarial Training:

- The training process involves a back-and-forth competition between the Generator and the Discriminator.
- The Generator improves its ability to generate realistic data by receiving feedback from the Discriminator.
- The Discriminator enhances its ability to differentiate between real and fake data through iterative training.

#### Optimization:

- Both networks are optimized simultaneously using different loss functions tailored to their specific roles.
- The loss functions guide the networks towards convergence, where the Generator produces high-quality data that can deceive the Discriminator effectively.

### 3.2.3 Core principles of GAN:

The three core principles of Generative Adversarial Networks (GANs) are:

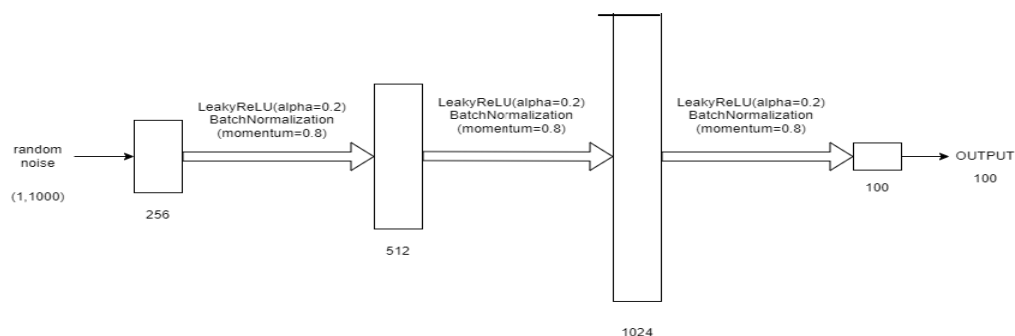
**Generative:** GANs aim to capture the underlying distribution of the data being modeled, allowing them to generate new data that matches the characteristics of the original data.

**Adversarial:** GANs operate in an adversarial environment, where the generator competes with the discriminator to produce more realistic data. The discriminator judges the validity of the generated data, guiding the generator to improve its output.

**Networks:** GANs consist of two neural networks: the generator and the discriminator. The generator creates new data, while the discriminator checks the validity of the newly generated data.

### 3.2.4 Architecture of GAN in our generative music synthesis model

The architecture of a Generative Adversarial Network (GAN) consists of two primary components: the generator and the discriminator. The generator generates synthetic data from random noise, while the discriminator evaluates whether the generated data looks real or fake. During training, the generator tries to fool the discriminator, and the discriminator tries to correctly classify the generated data. The generator updates its parameters based on the feedback provided by the discriminator, and vice versa. The goal is to reach a balance where the generator can produce realistic data that the discriminator cannot distinguish from real data. The architecture is shown in [Fig 3.13](#).



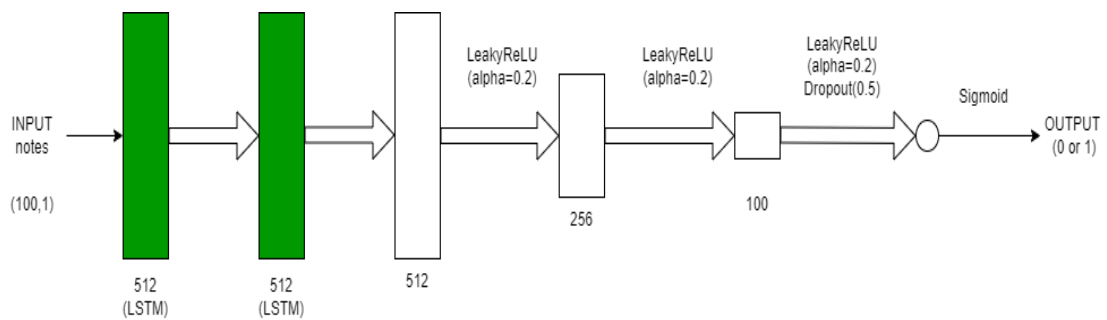
*Figure 3.13: Generator Architecture*



The generator network is a crucial element within the Generative Adversarial Network (GAN) framework, tasked with generating synthetic samples that closely mimic the distribution of real data. It operates as a feed forward neural network, comprising multiple layers arranged sequentially to process random noise inputs and produce meaningful outputs.

The architecture consists of four densely connected layers, each playing a distinct role in the transformation process. At the outset, the initial layer, composed of 256 units, acts as the point of entry for the input noise. Subsequent layers, featuring 512 and 1024 units, respectively, progressively extract and refine features from the input noise, allowing the network to capture intricate patterns and structures present in the data. Finally, the output layer, consisting of 100 units, is responsible for generating the output sequence representing the desired melody length.

To introduce non-linearity and facilitate information flow throughout the network, Leaky Rectified Linear Unit (ReLU) activation functions with a slope of 0.2 are applied after each hidden layer. These activation functions help overcome the vanishing gradient problem and enable the network to learn complex representations effectively. Additionally, batch normalization is integrated after every layer, incorporating a momentum parameter of 0.8. This technique aids in stabilizing and accelerating the training process by normalizing the activations within each layer and reducing the effects of internal covariate shift. By maintaining a consistent distribution of inputs to subsequent layers, batch normalization promotes smoother and more efficient training, ultimately enhancing the generator's ability to produce high-quality synthetic samples.



*Figure 3.14: Discriminator Architecture*

The discriminator in GAN architecture uses a bidirectional LSTM layer with 512 units to analyze temporal dependencies in melody sequences, as shown in Fig 3.14. Two densely connected layers (512 and 256 units) refine features for discrimination. This architecture enhances the discriminator's ability to differentiate between real and generated samples.

The discriminator incorporates dropout (rate of 0.5) to prevent overfitting and enhance generalization. It also employs mini batch discrimination to encourage diversity in generated samples by capturing global statistics across mini-batches. The output layer consists of a single unit with sigmoid activation for binary classification, indicating the authenticity of input melodies. Leaky ReLU activation functions (slope of 0.2) after each hidden layer enhances discriminative capabilities.

The architecture of a Generative Adversarial Network (GAN) plays a critical role in determining its performance. Key factors affecting GAN performance include the choice of the generator architecture, the discriminator architecture, the loss function, and the overall structure of the network. The generator architecture determines how the generator processes input noise to generate synthetic data. Common choices include convolutional layers for image data and recurrent layers for sequence data. The discriminator architecture defines how the discriminator distinguishes between real and fake data. The loss function is important because it dictates how the generator and discriminator compete during training. If the loss function is too easy, the generator may never achieve good performance; if it is too hard, the generator may never get close enough to fool the discriminator. The overall structure of the network, including the number of layers, neurons, and type of activation functions, must be balanced to ensure stable training and avoid mode collapse.

The output of the generator (and correspondingly the input of the discriminator) is normalized between -1 and +1. They are scaled back to integers using:

```
pred_notes = [x * (n_vocab / 2) + (n_vocab / 2)
               for x in predictions[0]]
```

These integers are converted back to notes using a mapping. Adam optimizer with learning rate = 0.0002, beta\_1 = 0.5 is used. Binary cross entropy is utilized as the loss function.

### 3.3 Ways to combat mode collapse in GAN

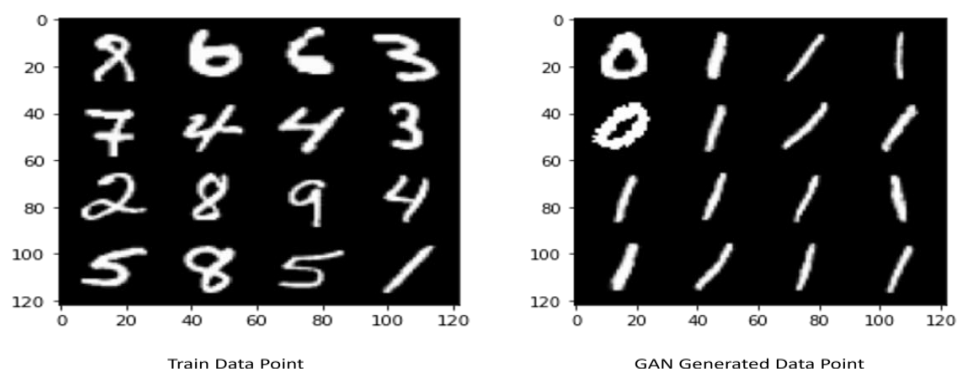
Mode collapse is a major obstacle in training Generative Adversarial Networks (GANs). It occurs when the generator, tasked with creating realistic data, gets stuck in a rut. Instead of exploring the full diversity of the real data distribution, it fixates on generating a limited set of outputs, often resembling only a few dominant modes. This leads to a lack of variety and faithfulness to the true data in the generated samples.

Here is a breakdown of why mode collapse happens:

**Unbalanced Training:** The training process can become lopsided. The discriminator, responsible for distinguishing real data from the generator's creations, might become too adept at identifying fakes for certain modes. This leaves the generator struggling to replicate those modes effectively, while neglecting other parts of the data distribution.

**Difficulty of Learning Complex Distributions:** Complex data often has multiple "modes" (areas of high probability). For GANs, especially with simpler architectures, learning these intricate distributions can be challenging. The generator might settle for easier-to-generate, simpler modes, leading to a lack of diversity.

Other reasons include insufficient model capacity and improper choice of loss function.



*Figure 3.15: An example of mode collapse for a GAN trained to produce handwritten digits*

### 3.3.1 Wasserstein GAN

It is an extension of the GAN that seeks an alternate way of training the generator model to better approximate the distribution of data observed in a given training dataset.

Instead of using a discriminator to classify or predict the probability of generated images as being real or fake, the WGAN changes or replaces the discriminator model with a critic that scores the realness or fakeness of a given image.

This change is motivated by a mathematical argument that training the generator should seek a minimization of the distance between the distribution of the data observed in the training dataset and the distribution observed in generated examples. The argument contrasts different distribution distance measures, such as Kullback-Leibler (KL) divergence, Jensen-Shannon (JS) divergence, and the Earth-Mover (EM) distance, referred to as Wasserstein distance.

This change is motivated by the mathematical concept of the Earth-Mover (EM) distance, also known as Wasserstein distance, which measures the distance between probability distributions. By approximating this distance, the WGAN facilitates stable training and generates higher quality images. Importantly, the Wasserstein distance is continuous and differentiable, providing reliable gradients even after the critic is well trained. Unlike traditional discriminators, which may fail to provide useful gradient information, the critic in WGANs converges to a linear function, offering clean gradients throughout training. This stability in training makes WGANs less sensitive to model architecture and hyperparameter configurations, reducing the phenomenon of mode dropping.

Furthermore, the loss of the critic directly correlates with the quality of generated images, allowing for convergence rather than equilibrium between the generator and discriminator. This property represents a significant advancement in GAN literature, providing insights into model performance without the need for manual inspection of generated samples.

The implementation details of the Wasserstein Generative Adversarial Network (WGAN) involve several key adjustments compared to standard deep convolutional GANs (DCGANs):

**Activation Function:** Use a linear activation function in the output layer of the critic model instead of the sigmoid activation function typically used in discriminator models.

**Loss Function:** Utilize Wasserstein loss to train both the critic and generator models. This loss function encourages a larger difference between scores for real and generated images, promoting better training dynamics.

**Weight Constraint:** Constrain the weights of the critic model to a limited range after each mini-batch update, for example, within the range  $[-0.01, 0.01]$ . This constraint helps ensure that the parameters of the critic lie in a compact space.

**Critic Model Updates:** Update the critic model more times than the generator each iteration. For instance, updating the critic five times for every update of the generator helps stabilize the training process.

**Optimizer:** Use the RMSProp version of gradient descent with a small learning rate and no momentum, such as 0.00005.

These adjustments are aimed at improving the stability and convergence properties of the WGAN during training, making it less sensitive to hyperparameters and architecture choices.

*Table 3.1: Proposed WGAN algorithm in literature*

<b>Algorithm:</b> WGAN, proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$ , $c = 0.01$ , $m = 64$ , $n_{\text{critic}} = 5$ .	
<b>Require:</b> $\alpha$ , the learning rate. $c$ , the clipping parameter. $M$ , the batch size. $n_{\text{critic}}$ , the number of iterations of the critic per generator iteration.	
<b>Require:</b> $w_0$ , initial critic parameters. $\theta_0$ , initial generator's parameters.	
1:	<b>while</b> $\theta$ has not converged <b>do</b>
2:	<b>for</b> $t = 0, \dots, n_{\text{critic}}$ <b>do</b>
3:	Sample $\{x^{(i)}\}_{i=1}^m \sim P_r$ a batch from the real data.
4:	Sample $\{z^{(i)}\}_{i=1}^m \sim p(z)$ a batch of prior samples.
5:	$g_w \leftarrow \nabla_w [\frac{1}{m} \sum_{i=1}^m f_w(x^{(i)}) - \frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)}))]$
6:	$w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
7:	$w \leftarrow \text{clip}(w, -c, c)$
8:	<b>end for</b>
9:	Sample $\{z^{(i)}\}_{i=1}^m \sim p(z)$ a batch of prior samples
10:	$g_\theta \leftarrow -\nabla_\theta [\frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)}))]$
11:	$\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
12:	<b>end while</b>

WGAN model requires a conceptual shift away from a discriminator that predicts the probability of a generated image being “real” and toward the idea of a critic model that scores the “realness” of a given image.

This conceptual shift is motivated mathematically using the earth mover distance, or Wasserstein distance, to train the GAN that measures the distance between the data distribution observed in the training dataset and the distribution observed in the generated examples.

According to Arjovsky et al[6],

$$\text{Critic Loss} = [\text{average critic score on real data}] - [\text{average critic score on fake data}]$$

$$\text{Generator Loss} = - [\text{average critic score on fake data}]$$

Where the average scores are calculated across a mini-batch of samples.

In the case of the generator, a larger score from the critic will result in a smaller loss for the generator, encouraging the critic to output larger scores for fake images. For example, an average score of 10 becomes -10, an average score of 50 becomes -50, which is smaller, and so on. In the case of the critic, a larger score for real images results in a larger resulting loss for the critic, penalizing the model. This encourages the critic to output smaller scores for real images. For example, an average score of 20 for real images and 50 for fake images results in a loss of -30; an average score of 10 for real images and 50 for fake images results in a loss of -40, which is better, and so on. The sign of the loss does not matter in this case, as long as loss for real images is a small number and the loss for fake images is a large number. The Wasserstein loss encourages the critic to separate these numbers.

In the Keras deep learning library (and some others), we cannot implement the Wasserstein loss function directly as described in the paper and as implemented in PyTorch and TensorFlow. Instead, we can achieve the same effect without having the calculation of the loss for the critic dependent upon the loss calculated for real and fake images.

A good way to think about this is a negative score for real images and a positive score for fake images, although this negative/positive split of scores learned during training is not required;

just larger and smaller is sufficient.

Small Critic Score (e.g. < 0): Real

Large Critic Score (e.g. >0): Fake

We can multiply the average predicted score by -1 in the case of fake images so that larger averages become smaller averages and the gradient is in the correct direction, i.e. minimizing loss. For example, average scores on fake images of [0.5, 0.8, and 1.0] across three batches of fake images would become [-0.5, -0.8, and -1.0] when calculating weight updates.

$$\text{Loss For Fake Images} = -1 * \text{Average Critic Score}$$

No change is needed for the case of real scores, as we want to encourage smaller average scores for real images.

$$\text{Loss For Real Images} = \text{Average Critic Score}$$

This can be implemented consistently by assigning an expected outcome target of -1 for fake images and 1 for real images and implementing the loss function as the expected label multiplied by the average score. The -1 label will be multiplied by the average score for fake images and encourage a larger predicted average, and the +1 label will be multiplied by the average score for real images and have no effect, encouraging a smaller predicted average.

$$\text{Wasserstein Loss} = \text{Label} * \text{Average Critic Score}$$

Or

$$\text{Wasserstein Loss(Real Images)} = 1 * \text{Average Predicted Score}$$

$$\text{Wasserstein Loss(Fake Images)} = -1 * \text{Average Predicted Score}$$

We can implement this in Keras by assigning the expected labels of -1 and 1 for fake and real images respectively.

### 3.3.2 Random noise injection

Injecting Gaussian noise (standard deviation = 0.1) at various points, like after each layer of the generator and discriminator, fosters exploration of diverse outputs, reducing memorization

of specific inputs. This enhances sample diversity and realism. It also boosts model capacity, capturing complex patterns effectively while preventing overfitting. Noise presence ensures robustness to input perturbations, stabilizing training and enhancing overall model performance and generalization.

### 3.3.3 Regularization

To prevent overfitting and encourage generalization, L2 kernel regularization with a lambda parameter, typically denoted as  $\lambda = 0.1$ , is applied. This penalizes large weights, making the networks less sensitive to noise and more robust. Smaller weights mitigate overfitting, enhancing the discriminator's performance and promoting the learning of discriminative features by the generator.

$$\text{Modified Loss Function} = \text{Loss Function} + \lambda \cdot \sum_{i=1}^n W_i^2$$

## 3.4 Dataset and Preprocessing

The Erk subset, a segment of the larger KernScores Essen Deutschl dataset, encompasses 1700 distinct German piano tunes, meticulously processed utilizing the music21 library, renowned for its prowess in .krn file manipulation. These compositions underwent a series of preprocessing steps to ensure their suitability for subsequent analysis and modeling. Initially, the melodies were subjected to filtering based on acceptable duration criteria, ensuring consistency and coherence in the resulting dataset. Additionally, to enhance the dataset's versatility and usability, the melodies were transposed as necessary to standardize their key signatures and facilitate comparison across compositions. Subsequently, the processed melodies were encoded into a standardized string format, employing appropriate delimiters to delineate individual musical elements and ensure data integrity. Furthermore, to facilitate computational analysis and model training, a comprehensive mapping of string characters to integers was established, enabling the generation of training sequences comprising input and target integers. This meticulous preprocessing not only ensures the quality and consistency of the dataset but also lays the foundation for a wide array of subsequent analyses, including but not limited to pattern recognition, music generation, and stylistic analysis.



# Chapter 4

## Simulation and Evaluation Parameters

### 4.1 Classification Loss

We used binary cross-entropy loss function to judge the performance of the generator and discriminator models in accordance with Goodfellow et al[5]. Binary cross-entropy loss, commonly used in binary classification tasks, measures the discrepancy between predicted probabilities and true binary labels. It calculates the loss by averaging the negative log likelihood of the true labels given the predicted probabilities. In essence, when the predicted probability aligns with the true label (i.e., high probability for positive samples and low probability for negative samples), the loss approaches zero. However, as the predicted probability deviates from the true label, the loss increases exponentially, penalizing incorrect predictions more severely. This loss function encourages the model to learn accurate probability estimates and effectively discriminates between the two classes. It is particularly well-suited for training neural networks in binary classification tasks, providing a clear and interpretable measure of model performance that guides the optimization process towards minimizing classification errors.

$$\text{Binary Cross Entropy Loss} = -(y * \log(p) + (1 - y) * \log(1 - p))$$

However, these losses do not directly assess the quality or diversity of the generated outputs. The objective function for the generator and the discriminator usually measures how well they are doing **relative** to the opponent. Later researchers have established that it is not a good

metric in measuring the output quality or its diversity. Additional parameters are required for a deeper analysis of the model performance, as described below.

## 4.2 Manhattan distance

The study by Mitzenmacher and Owen (2000) [7] introduces a method for comparing MIDI files based on the grouping of adjacent pitches and computing the Manhattan distance between these pitch groups.

$$d(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

## 4.3 Average pitch class transition matrix

The two-dimensional pitch class transition matrix is a histogram-like representation computed by counting the pitch transitions for each (ordered) pair of notes. The resulting feature dimensionality is  $12 \times 12$ . According to Yang et al [8] the Average Pitch Class Transition Matrix (APCTM) is a representation used in music analysis to capture the statistical relationships between different pitch classes in a piece of music. It quantifies how often each pitch class transitions to another pitch class within a musical composition, providing insights into the harmonic structure, tonality, and overall flow of the music. By averaging the transition matrices of multiple pieces of music, the APCTM provides a summary of typical pitch class transitions across a musical genre, composer, or style. This matrix is particularly useful in tasks such as music genre classification, style recognition, and composer attribution, where analyzing the distribution of pitch class transitions can reveal characteristic patterns and distinguish between different musical contexts. The APCTM serves as a valuable tool for musicologists, composers, and machine learning practitioners interested in understanding the underlying structure and organization of musical compositions.

## 4.4 Average note length transition matrix

According to Yang et al [8] the Average Note Length Transition Matrix (ANLTM) is a representation commonly utilized in music analysis to characterize the temporal relationships

between different note lengths within a musical composition. It quantifies the likelihood of transitioning from one note length to another, offering insights into the rhythmic structure, pacing, and phrasing of the music. By averaging the transition matrices of multiple pieces of music, the ANLTM provides a condensed summary of typical note length transitions across various musical contexts, genres, or composers. This matrix serves as a valuable tool for musicologists, composers, and machine learning practitioners interested in understanding the rhythmic patterns and temporal dynamics of musical compositions. Additionally, it finds applications in tasks such as music genre classification, rhythmic pattern recognition, and automated music composition, where analyzing note length transitions contributes to a deeper understanding of musical structure and expression. This is an 8x8 matrix.

## 4.5 Pitch

We can compare similarities in the generated and training dataset by comparing the mean and standard deviation of the pitch used in both datasets.

## 4.6 Kernel Density Estimate plots of intra-set distances

These are plots of Kernel Density of estimated Euclidean distances between melodies within the same dataset, calculated by leave-one-out methodology. A one-to-one comparison between plots of original dataset (used to train the model, `intra_set2`) with that of the generated dataset (`intra_set1`) could help in identifying potential mode collapse issues. The original dataset is a subset of the training dataset with 100 randomly chosen melodies, and the generated dataset is a collection of 100 melodies generated by the model. These plots have been drawn in accordance with Yang L.C. et al (2020) [8].

# Chapter 5

## Results

Below are two samples of generated piano melodies, generated by giving an input noise of an array of 1000 normally distributed random numbers between 0 and 1, as shown in [Fig 5.1](#):



*Figure 5.1: Noise Input*

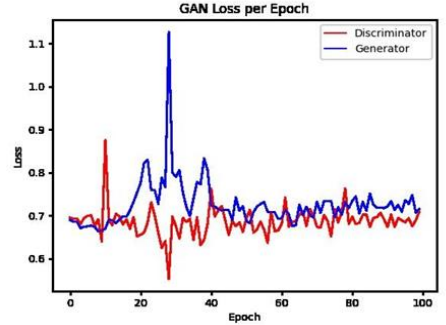
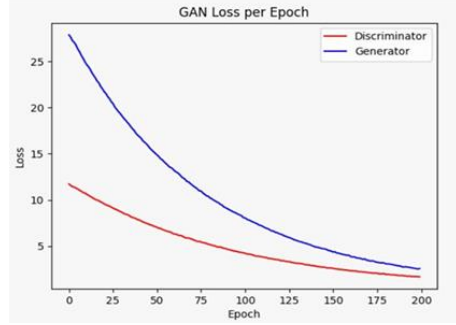


*Figure 5.2: Piano Sample 1*



*Figure 5.3: Piano Sample 2*

Table 5.1: Results

Evaluation Metric	Result	
Classification Loss of GANs	Normal GAN	<p>Generator Loss = 0.715</p> <p>Discriminator Loss = 0.709</p>  <p><i>Figure 5.4: Loss vs Epoch for GAN</i></p>
	WGAN with noise injection and regularization	<p>Generator Loss = 2.56</p> <p>Discriminator Loss = 1.66</p>  <p><i>Figure 5.5: Loss vs Epoch for WGAN with noise and regularization</i></p>

Manhattan Distance

Original Dataset : 141.35  
Generated Dataset : 234.88

The generated music is not too similar to the samples of the dataset, but it can be a desirable factor as it means the generated samples are diverse and creative.

Average pitch class transition matrix

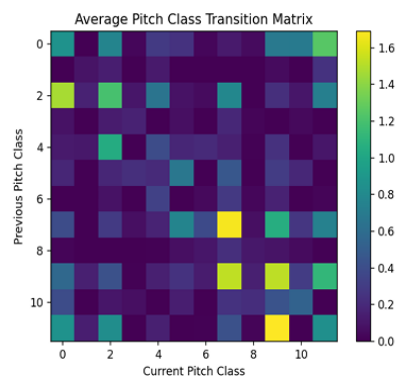


Figure 5.6: PCTM for original dataset

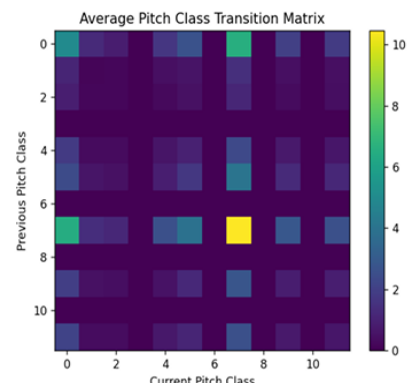
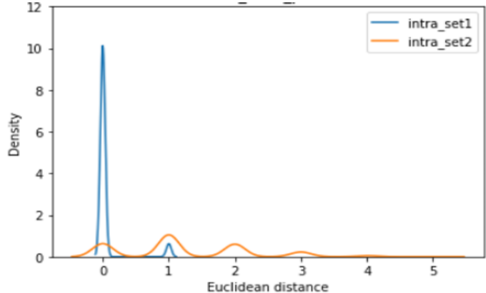
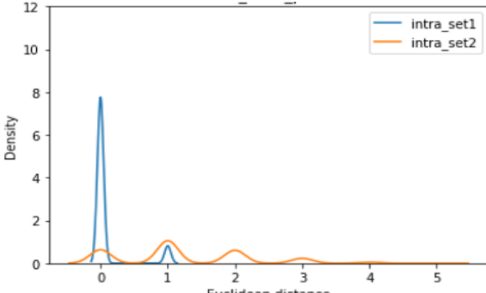
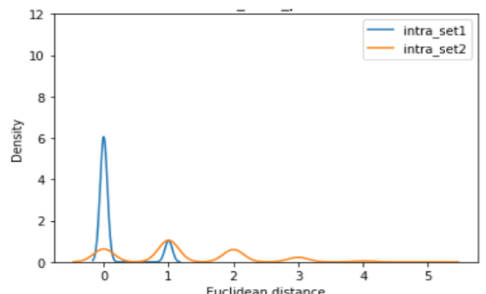
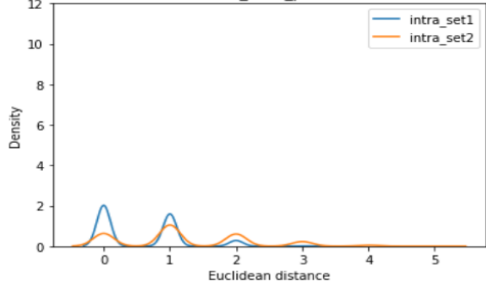


Figure 5.7: PCTM for generated dataset

	<p>While not exactly similar, essential qualities of the original dataset like high transition from pitch 7 to pitch 7, 7 to 9 and 9 to 7 have been captured.</p>
Average note length transition matrix	<div data-bbox="938 405 1347 757" data-label="Figure"> </div> <p><i>Figure 5.8: NLTM for original dataset</i></p> <div data-bbox="938 896 1347 1256" data-label="Figure"> </div> <p><i>Figure 5.9: NLTM for generated dataset</i></p> <p>While not exactly similar, essential qualities of the original dataset like high transition from note length 2 to 2, 2 to 4 and 4 to 2 and 4 to 4 have been captured.</p>
Pitch	<p>Original Dataset: mean: 7.61, standard deviation: 1.11260955</p> <p>Generated Dataset: mean: 7.97, standard deviation: 0.17058722</p>

	<p>KDE plot for normal GAN architecture</p>	 <p>Figure 5.10: KDE plot for GAN</p>
<p>Kernel Density Estimate plots of intra-set distances</p>	<p>KDE plot for Wasserstein GAN architecture</p>	 <p>Figure 5.11: KDE plot for WGAN</p>
	<p>KDE plot for WGAN with random noise injection</p>	 <p>Figure 5.12: KDE plot for WGAN with noise</p>
	<p>KDE plot for WGAN with random noise injection and L2 regularization</p>	 <p>Figure 5.13: KDE plot for WGAN with noise, regularization</p>



# Conclusion and Future Scope

We started out with using LSTMs as a generative model for music synthesis but quickly realized that this approach has the inherent demerit of not producing complex and constant length melodies. GANs can overcome these problems but are susceptible to mode collapse. We solved this issue by modifying the GAN architecture, by employing Wasserstein loss function, random noise injection and L2 regularization. By adopting these techniques, we essentially have reduced the effects of mode collapse by around 80% (drawing inference from the kernel density plots).

WGAN with random noise injection and L2 regularization has demonstrated the ability to produce results of definite length, allowing for the generation of data with specific characteristics or durations, along with enabling the creation of intricate and detailed outputs that closely resembles diverse real data. It has been successful in capturing abstract concepts like the "rise and fall of notes," showcasing its capacity to understand and replicate nuanced patterns and structures within musical compositions.

Thus, we have successfully developed a robust generative model architecture capable of learning the intricacies of music theory given a diverse enough dataset of melodies.

Generative performance of this architecture could be further enhanced by using self-attention mechanisms. By Dockerizing the model and deploying it on a commercial cloud platform like AWS or GCP, this project could be made production ready.

The model can be trained on vast datasets of long-gone legendary composers like Bach, Mozart, or Beethoven to bring their composition skills back to life. If deployed on user end devices the model can be used to spawn an infinite number of distinct and diverse ringtones, alarms, and notification sounds, adding an element of freshness to daily use.

# Bibliography

- [1] Hochreiter, Sepp, and Jürgen Schmidhuber. "Long Short-term Memory." *Neural Computation* 9 (1997): 1735-80. <https://doi.org/10.1162/neco.1997.9.8.1735>.
- [2] Williams, Ronald J., and David Zipser. "Gradient-based learning algorithms for recurrent networks and their computational complexity." In *Backpropagation: Theory, Architectures, and Applications*, 433–486. L. Erlbaum Associates Inc., USA, 1995.
- [3] Robinson, Tony, and F. Fallside. "The utility driven dynamic error propagation network." 1987.
- [4] Hochreiter, Sepp. *Untersuchungen zu dynamischen neuronalen Netzen*. Diploma thesis, TU Munich, 1991.
- [5] Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative adversarial nets." In *Advances in Neural Information Processing Systems*, pp. 2672–2680, 2014.
- [6] Arjovsky, Martin, Soumith Chintala, and Léon Bottou. "Wasserstein generative adversarial networks." In *International Conference on Machine Learning*, pp. 214-223, 2017.
- [7] Mitzenmacher, Michael, and Sean Owen. "Estimating Resemblance of MIDI Documents." *Harvard Computer Science Group Technical Report TR-06-00*, 2000.
- [8] Yang, LC., and A. Lerch. "On the evaluation of generative models in music." *Neural Computing & Applications* 32 (2020): 4773–4784.
- [9] Li, G., S. Ding, and Y. Li. "Novel LSTM-GAN Based Music Generation." In *2021 13th International Conference on Wireless Communications and Signal Processing (WCSP)*, pp. 1-6. Changsha, China, 2021. <https://doi.org/10.1109/WCSP52459.2021.9613311>.

- [10] Feng, R., D. Zhao, and Z. Zha. "On noise injection in generative adversarial networks." arXiv preprint arXiv:2006.05891, 2020.
- [11] Lee, M., and J. Seok. "Regularization methods for generative adversarial networks: An overview of recent studies." arXiv preprint arXiv:2005.09165, 2020.
- [12] Donahue, Chris, Julian McAuley, and Miller Puckette. "Adversarial audio synthesis." In Advances in Neural Information Processing Systems, pp. 6945-6955. 2019.
- [13] Roberts, Adam, Jesse Engel, and Douglas Eck. "A hierarchical latent vector model for learning long-term structure in music." arXiv preprint arXiv:1803.05428 (2018).
- [14] Dhariwal, Prafulla, et al. "Jukebox: A generative model for music." arXiv preprint arXiv:2106.03776 (2021).
- [15] Oord, Aaron van den, et al. "WaveNet: A generative model for raw audio." arXiv preprint arXiv:1609.03499 (2016).
- [16] Engel, Jesse, et al. "GANSynth: Adversarial neural audio synthesis." In Advances in Neural Information Processing Systems, pp. 1910-1920. 2019.
- [17] Li, G., S. Ding, and Y. Li. "Novel LSTM-GAN Based Music Generation." In 2021 13th International Conference on Wireless Communications and Signal Processing (WCSP), pp. 1-6. Changsha, China, 2021.
- [18] Salimans, Tim, et al. "Improved techniques for training GANs." In Advances in Neural Information Processing Systems, pp. 2234-2242. 2016.
- [19] Berthelot, David, Tom Schumm, and Luke Metz. "BEGAN: Boundary equilibrium generative adversarial networks." arXiv preprint arXiv:1703.10717 (2017).
- [20] Mescheder, Lars, Andreas Geiger, and Sebastian Nowozin. "Which training methods for GANs do actually converge?" In International Conference on Machine Learning, pp. 3481-3490. 2018.

- [21] Arora, Sanjeev, et al. "A theoretical analysis of contrastive unsupervised representation learning." arXiv preprint arXiv:2007.02269 (2020).
- [22] Azadi, Samaneh, et al. "Discriminator rejection sampling." In Advances in Neural Information Processing Systems, pp. 3300-3309. 2020.
- [23] Gulrajani, Ishaan, et al. "Improved training of Wasserstein GANs." In Advances in Neural Information Processing Systems 30, pp. 5767-5777. 2017.
- [24] Yang, Li-Chia, Szu-Yu Chou, and Yi-Hsuan Yang. "MidiNet: A convolutional generative adversarial network for symbolic-domain music generation." arXiv preprint arXiv:1703.10847 (2017).
- [25] Zhang, Haohang, Letian Xie, and Kaiyi Qi. "Implement music generation with gan: A systematic review." In 2021 International Conference on Computer Engineering and Application (ICCEA), pp. 352-355. IEEE, 2021.
- [26] Zhang, Guanghao, Enmei Tu, and Dongshun Cui. "Stable and improved generative adversarial nets (GANS): A constructive survey." In 2017 IEEE International Conference on Image Processing (ICIP), pp. 1871-1875. IEEE, 2017.
- [27] Kurach, Karol, Mario Lučić, Xiaohua Zhai, Marcin Michalski, and Sylvain Gelly. "A large-scale study on regularization and normalization in GANs." In International conference on machine learning, pp. 3581-3590. PMLR, 2019.

# **Appendices**

# Appendix A

## Code Attachments

### A.1 Long Short-Term Memory Architecture

#### A.1.1 Preprocessing

```
import os
import json
import music21 as m21
import numpy as np
import tensorflow.keras as keras

KERN_DATASET_PATH = "D:/Music-LSTM/essen/europa/deutschl/erk"
SAVE_DIR = "dataset"
SINGLE_FILE_DATASET = "file_dataset"
MAPPING_PATH = "mapping.json"
SEQUENCE_LENGTH = 64

# durations are expressed in quarter length
ACCEPTABLE_DURATIONS = [
    0.25, # 16th note
    0.5, # 8th note
    0.75,
    1.0, # quarter note
    1.5,
    2, # half note
    3,
    4 # whole note
]

def load_songs_in_kern(dataset_path):

    songs = []

    for path, subdirs, files in os.walk(dataset_path):
        for file in files:

            if file[-3:] == ".kern":
                song = m21.converter.parse(os.path.join(path, file))
```

```

        songs.append(song)
    return songs

def has_acceptable_durations(song, acceptable_durations):
    for note in song.flat.notesAndRests:
        if note.duration.quarterLength not in acceptable_durations:
            return False
    return True

def transpose(song):
    # get key from the song
    parts = song.getElementsByClass(m21.stream.Part)
    measures_part0 = parts[0].getElementsByClass(m21.stream.Measure)
    key = measures_part0[0][4]

    if not isinstance(key, m21.key.Key):
        key = song.analyze("key")

    if key.mode == "major":
        interval = m21.interval.Interval(key.tonic, m21.pitch.Pitch("C"))
    elif key.mode == "minor":
        interval = m21.interval.Interval(key.tonic, m21.pitch.Pitch("A"))

    # transpose song by calculated interval
    transposed_song = song.transpose(interval)
    return transposed_song

def encode_song(song, time_step=0.25):
    encoded_song = []

    for event in song.flat.notesAndRests:
        # handle notes
        if isinstance(event, m21.note.Note):
            symbol = event.pitch.midi # 60
        # handle rests
        elif isinstance(event, m21.note.Rest):
            symbol = "r"

        # convert the note/rest into time series notation
        steps = int(event.duration.quarterLength / time_step)
        for step in range(steps):
            # if it's the first time we see a note/rest, let's encode it.
            Otherwise, it means we're carrying the same
            # symbol in a new time step
            if step == 0:
                encoded_song.append(symbol)
            else:
                encoded_song.append("_")

    # cast encoded song to str
    encoded_song = " ".join(map(str, encoded_song))

```

```

    return encoded_song

def preprocess(dataset_path):

    # Load folk songs
    print("Loading songs...")
    songs = load_songs_in_kern(dataset_path)
    print(f"Loaded {len(songs)} songs.")

    for i, song in enumerate(songs):

        # filter out songs that have non-acceptable durations
        if not has_acceptable_durations(song, ACCEPTABLE_DURATIONS):
            continue

        # transpose songs to Cmaj/Amin
        song = transpose(song)

        # encode songs with music time series representation
        encoded_song = encode_song(song)

        # save songs to text file
        save_path = os.path.join(SAVE_DIR, str(i))
        with open(save_path, "w") as fp:
            fp.write(encoded_song)

def load(file_path):
    with open(file_path, "r") as fp:
        song = fp.read()
    return song

def create_single_file_dataset(dataset_path, file_dataset_path,
sequence_length):

    new_song_delimiter = "/" * sequence_length
    songs = ""

    # Load encoded songs and add delimiters
    for path, _, files in os.walk(dataset_path):
        for file in files:
            file_path = os.path.join(path, file)
            song = load(file_path)
            songs = songs + song + " " + new_song_delimiter

    # remove empty space from last character of string
    songs = songs[:-1]

    # save string that contains all the dataset
    with open(file_dataset_path, "w") as fp:
        fp.write(songs)

    return songs

def create_mapping(songs, mapping_path):

```



```

mappings = {}

# identify the vocabulary
songs = songs.split()
vocabulary = list(set(songs))

# create mappings
for i, symbol in enumerate(vocabulary):
    mappings[symbol] = i

# save vocabulary to a json file
with open(mapping_path, "w") as fp:
    json.dump(mappings, fp, indent=4)

def convert_songs_to_int(songs):
    int_songs = []

    # load mappings
    with open(MAPPING_PATH, "r") as fp:
        mappings = json.load(fp)

    # transform songs string to list
    songs = songs.split()

    # map songs to int
    for symbol in songs:
        int_songs.append(mappings[symbol])

    return int_songs

def generate_training_sequences(sequence_length):

    # load songs and map them to int
    songs = load(SINGLE_FILE_DATASET)
    int_songs = convert_songs_to_int(songs)

    inputs = []
    targets = []

    # generate the training sequences
    num_sequences = len(int_songs) - sequence_length
    for i in range(num_sequences):
        inputs.append(int_songs[i:i+sequence_length])
        targets.append(int_songs[i+sequence_length])

    # one-hot encode the sequences
    vocabulary_size = len(set(int_songs))
    # inputs size: (# of sequences, sequence length, vocabulary size)
    inputs = keras.utils.to_categorical(inputs, num_classes=vocabulary_size)
    targets = np.array(targets)

    return inputs, targets

def main():
    preprocess(KERN_DATASET_PATH)
    songs = create_single_file_dataset(SAVE_DIR, SINGLE_FILE_DATASET,
SEQUENCE_LENGTH)

```

```

create_mapping(songs, MAPPING_PATH)
inputs, targets = generate_training_sequences(SEQUENCE_LENGTH)

if __name__ == "__main__":
    main()

```

## A.1.2 Model Architecture and Training

```

import tensorflow.keras as keras
from preprocessing import generate_training_sequences, SEQUENCE_LENGTH

OUTPUT_UNITS = 38
NUM_UNITS = [256]
LOSS = "sparse_categorical_crossentropy"
LEARNING_RATE = 0.001
EPOCHS = 50
BATCH_SIZE = 64
SAVE_MODEL_PATH = "model.h5"

def build_model(output_units, num_units, loss, learning_rate):

    # create the model architecture
    input = keras.layers.Input(shape=(None, output_units))
    x = keras.layers.LSTM(num_units[0])(input)
    x = keras.layers.Dropout(0.2)(x)

    output = keras.layers.Dense(output_units, activation="softmax")(x)

    model = keras.Model(input, output)

    # compile model
    model.compile(loss=loss,
optimizer=keras.optimizers.Adam(learning_rate=learning_rate),
metrics=["accuracy"])

    model.summary()

    return model

def train(output_units=OUTPUT_UNITS, num_units=NUM_UNITS, loss=LOSS,
learning_rate=LEARNING_RATE):

    # generate the training sequences
    inputs, targets = generate_training_sequences(SEQUENCE_LENGTH)

    # build the network
    model = build_model(output_units, num_units, loss, learning_rate)

```

```

# train the model
model.fit(inputs, targets, epochs=EPOCHS, batch_size=BATCH_SIZE)

# save the model
model.save(SAVE_MODEL_PATH)

if __name__ == "__main__":
    train()

```

### A.1.3 Generating melodies with LSTM

```

import json
import numpy as np
import tensorflow.keras as keras
import music21 as m21
from preprocessing import SEQUENCE_LENGTH, MAPPING_PATH

class MelodyGenerator:

    def __init__(self, model_path="model.h5"):

        self.model_path = model_path
        self.model = keras.models.load_model(model_path, compile=False)

        with open(MAPPING_PATH, "r") as fp:
            self._mappings = json.load(fp)

        self._start_symbols = ["/"] * SEQUENCE_LENGTH

    def generate_melody(self, seed, num_steps, max_sequence_length,
temperature):

        # create seed with start symbols
        seed = seed.split()
        melody = seed
        seed = self._start_symbols + seed

        # map seed to int
        seed = [self._mappings[symbol] for symbol in seed]

        for _ in range(num_steps):

            # limit the seed to max_sequence_length
            seed = seed[-max_sequence_length:]

            # one-hot encode the seed
            onehot_seed = keras.utils.to_categorical(seed,
num_classes=len(self._mappings))
            # (1, max_sequence_length, num of symbols in the vocabulary)
            onehot_seed = onehot_seed[np.newaxis, ...]

            # make a prediction
            probabilities = self.model.predict(onehot_seed)[0]

```

```

        # [0.1, 0.2, 0.1, 0.6] -> 1
        output_int = self._sample_with_temperature(probabilities,
temperature)

        # update seed
        seed.append(output_int)

        # map int to our encoding
        output_symbol = [k for k, v in self._mappings.items() if v ==
output_int][0]

        # check whether we're at the end of a melody
        if output_symbol == "/":
            break

        # update melody
        melody.append(output_symbol)

    return melody

def _sample_with_temperature(self, probabilities, temperature):

    predictions = np.log(probabilities) / temperature
    probabilities = np.exp(predictions) / np.sum(np.exp(predictions))

    choices = range(len(probabilities))
    index = np.random.choice(choices, p=probabilities)

    return index

def save_melody(self, melody, step_duration=0.25, format="midi",
file_name="mel.mid"):

    # create a music21 stream
    stream = m21.stream.Stream()

    start_symbol = None
    step_counter = 1

    # parse all the symbols in the melody and create note/rest objects
    for i, symbol in enumerate(melody):

        # handle case in which we have a note/rest
        if symbol != "_" or i + 1 == len(melody):

            # ensure we're dealing with note/rest beyond the first one
            if start_symbol is not None:

                quarter_length_duration = step_duration * step_counter #
0.25 * 4 = 1

                # handle rest
                if start_symbol == "r":
                    m21_event =
m21.note.Rest(quarterLength=quarter_length_duration)

                # handle note

```

```

        else:
            m21_event = m21.note.Note(int(start_symbol),
quarterLength=quarter_length_duration)

            stream.append(m21_event)

            # reset the step counter
            step_counter = 1

            start_symbol = symbol

            # handle case in which we have a prolongation sign "_"
        else:
            step_counter += 1

        # write the m21 stream to a midi file
        stream.write(format, file_name)

if __name__ == "__main__":
    mg = MelodyGenerator()
    seed = "67 _ 67 _ 67 _ _ 65 64 _ 64 _ 64 _ _"
    melody = mg.generate_melody(seed, 500, SEQUENCE_LENGTH, 0.9)
    print(melody)
    mg.save_melody(melody, file_name="mel.mid")

```

## A.2 Generative Adversarial Network Architecture

### A.2.1 Model Architecture and Training

```

import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense, Reshape, Dropout, LSTM,
Bidirectional
from tensorflow.keras.layers import BatchNormalization, LeakyReLU
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
from music21 import converter, instrument, note, chord, stream
from pathlib import Path
import matplotlib.pyplot as plt

SEQUENCE_LENGTH = 100
LATENT_DIMENSION = 1000
BATCH_SIZE = 16
EPOCHS = 100
SAMPLE_INTERVAL = 1

def get_notes():
    """ Get all the notes and chords from the midi files """
    notes = []

    for file in Path("archive").glob("*.mid"):
        midi = converter.parse(file)

```

```

    print("Parsing %s" % file)

    notes_to_parse = midi.flat.notes

    for element in notes_to_parse:
        if isinstance(element, note.Note):
            notes.append(str(element.pitch))
        elif isinstance(element, chord.Chord):
            notes.append('.'.join(str(n) for n in element.normalOrder))

    return notes

def prepare_sequences(notes, n_vocab):
    """ Prepare the sequences used by the Neural Network """
    sequence_length = 100

    # Get all pitch names
    pitchnames = sorted(set(item for item in notes))

    # Create a dictionary to map pitches to integers
    note_to_int = dict((note, number) for number, note in
enumerate(pitchnames))

    network_input = []
    network_output = []

    # create input sequences and the corresponding outputs
    for i in range(0, len(notes) - sequence_length, 1):
        sequence_in = notes[i:i + sequence_length]
        sequence_out = notes[i + sequence_length]
        network_input.append([note_to_int[char] for char in sequence_in])
        network_output.append(note_to_int[sequence_out])

    n_patterns = len(network_input)

    # Reshape the input into a format compatible with LSTM layers
    network_input = np.reshape(network_input, (n_patterns, sequence_length,
1))

    # Normalize input between -1 and 1
    network_input = (network_input - float(n_vocab) / 2) / (float(n_vocab) /
2)

    network_output = to_categorical(network_output, num_classes=n_vocab) #
Use to_categorical from TensorFlow's Keras

    return network_input, network_output # Add this return statement

def create_midi(prediction_output, filename):
    """ convert the output from the prediction to notes and create a midi file
    from the notes """
    offset = 0
    output_notes = []

    # create note and chord objects based on the values generated by the model
    for item in prediction_output:
        pattern = item[0]
        # pattern is a chord

```

```

if ('.' in pattern) or pattern.isdigit():
    notes_in_chord = pattern.split('.')
    notes = []
    for current_note in notes_in_chord:
        new_note = note.Note(int(current_note))
        new_note.storedInstrument = instrument.Piano()
        notes.append(new_note)
    new_chord = chord.Chord(notes)
    new_chord.offset = offset
    output_notes.append(new_chord)
# pattern is a note
else:
    new_note = note.Note(pattern)
    new_note.offset = offset
    new_note.storedInstrument = instrument.Piano()
    output_notes.append(new_note)

# increase offset each iteration so that notes do not stack
offset += 0.5

midi_stream = stream.Stream(output_notes)
midi_stream.write('midi', fp='{}.mid'.format(filename))

class GAN():
    def __init__(self, rows):
        self.seq_length = rows
        self.seq_shape = (self.seq_length, 1)
        self.latent_dim = 1000
        self.disc_loss = []
        self.gen_loss = []

        optimizer = Adam(0.0002, 0.5)

        # Build and compile the discriminator
        self.discriminator = self.build_discriminator()
        self.discriminator.compile(loss='binary_crossentropy',
optimizer=optimizer, metrics=['accuracy'])

        # Build the generator
        self.generator = self.build_generator()

        # The generator takes noise as input and generates note sequences
        z = Input(shape=(self.latent_dim,))
        generated_seq = self.generator(z)

        # For the combined model we will only train the generator
        self.discriminator.trainable = False

        # The discriminator takes generated images as input and determines
validity
        validity = self.discriminator(generated_seq)

        # The combined model (stacked generator and discriminator)
        # Trains the generator to fool the discriminator
        self.combined = Model(z, validity)
        self.combined.compile(loss='binary_crossentropy', optimizer=optimizer)

    def build_discriminator(self):
        model = Sequential()

```

```

        model.add(LSTM(512, input_shape=self.seq_shape,
return_sequences=True))
        model.add(Bidirectional(LSTM(512)))
        model.add(Dense(512))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dense(256))
        model.add(LeakyReLU(alpha=0.2))

        # Adding Minibatch Discrimination
        model.add(Dense(100))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dropout(0.5))
        model.add(Dense(1, activation='sigmoid'))
        model.summary()

        seq = Input(shape=self.seq_shape)
        validity = model(seq)

        return Model(seq, validity)

def build_generator(self):

    model = Sequential()
    model.add(Dense(256, input_dim=self.latent_dim))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(512))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(1024))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(np.prod(self.seq_shape), activation='tanh'))
    model.add(Reshape(self.seq_shape))
    model.summary()

    noise = Input(shape=(self.latent_dim,))
    seq = model(noise)

    return Model(noise, seq)

def train(self, epochs, batch_size=128, sample_interval=50):

    # Load and convert the data
    notes = get_notes()
    n_vocab = len(set(notes))
    X_train, y_train = prepare_sequences(notes, n_vocab)

    # Adversarial ground truths
    real = np.ones((batch_size, 1))
    fake = np.zeros((batch_size, 1))

    # Training the model
    for epoch in range(epochs):

        # Training the discriminator
        # Select a random batch of note sequences
        idx = np.random.randint(0, X_train.shape[0], batch_size)
        real_seqs = X_train[idx]

```



```

        #noise = np.random.choice(range(484), (batch_size,
self.latent_dim))
        #noise = (noise-242)/242
        noise = np.random.normal(0, 1, (batch_size, self.latent_dim))

        # Generate a batch of new note sequences
        gen_seqs = self.generator.predict(noise)

        # Train the discriminator
        d_loss_real = self.discriminator.train_on_batch(real_seqs, real)
        d_loss_fake = self.discriminator.train_on_batch(gen_seqs, fake)
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

        # Training the Generator
        noise = np.random.normal(0, 1, (batch_size, self.latent_dim))

        # Train the generator (to have the discriminator label samples as
real)
        g_loss = self.combined.train_on_batch(noise, real)

        # Print the progress and save into loss lists
        if epoch % sample_interval == 0:
            print ("%d [D loss: %f, acc.: %.2f%%] [G loss: %f]" % (epoch,
d_loss[0], 100*d_loss[1], g_loss))
            self.disc_loss.append(d_loss[0])
            self.gen_loss.append(g_loss)

        self.generate(notes)
        self.plot_loss()

    def generate(self, input_notes):
        # Get pitch names and store in a dictionary
        notes = input_notes
        pitchnames = sorted(set(item for item in notes))
        int_to_note = dict((number, note) for number, note in
enumerate(pitchnames))

        # Use random noise to generate sequences
        noise = np.random.normal(0, 1, (1, self.latent_dim))
        predictions = self.generator.predict(noise)

        pred_notes = [x*242+242 for x in predictions[0]]

        # Map generated integer indices to note names, with error handling
        pred_notes_mapped = []
        for x in pred_notes:
            index = int(x)
            if index in int_to_note:
                pred_notes_mapped.append(int_to_note[index])
            else:
                # Fallback mechanism: Choose a default note when the index is
out of range
                pred_notes_mapped.append('C5') # You can choose any default
note here

        create_midi(pred_notes_mapped, 'gan_final')

```

```

def plot_loss(self):
    plt.plot(self.disc_loss, c='red')
    plt.plot(self.gen_loss, c='blue')
    plt.title("GAN Loss per Epoch")
    plt.legend(['Discriminator', 'Generator'])
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.savefig('GAN_Loss_per_Epoch_final.png', transparent=True)
    plt.close()

if __name__ == '__main__':
    gan = GAN(rows=SEQUENCE_LENGTH)
    gan.train(epochs=EPOCHS, batch_size=BATCH_SIZE,
sample_interval=SAMPLE_INTERVAL)

    # Save the generator and discriminator models
    gan.generator.save("generator_model.h5")
    gan.discriminator.save("discriminator_model.h5")

```

## A.2.2 Generating melodies with GAN

```

import numpy as np
import tensorflow as tf
from tensorflow.keras.models import load_model
from music21 import stream, note, chord, instrument
from create_generator_model import get_notes, LATENT_DIMENSION

instr = instrument.Violin()

def create_midi(prediction_output, filename):
    """ convert the output from the prediction to notes and create a midi file
    from the notes """
    offset = 0
    output_notes = []

    # create note and chord objects based on the values generated by the model
    for item in prediction_output:
        pattern = item[0]
        # pattern is a chord
        if ('.' in pattern) or pattern.isdigit():
            notes_in_chord = pattern.split('.')
            notes = []
            for current_note in notes_in_chord:
                new_note = note.Note(int(current_note))
                new_note.storedInstrument = instr
                notes.append(new_note)
            new_chord = chord.Chord(notes)
            new_chord.offset = offset
            output_notes.append(new_chord)
        # pattern is a note
        else:
            new_note = note.Note(pattern)
            new_note.offset = offset
            new_note.storedInstrument = instr
            output_notes.append(new_note)

```

```

        # increase offset each iteration so that notes do not stack
        offset += 0.5

    midi_stream = stream.Stream(output_notes)
    midi_stream.write('midi', fp='{}.mid'.format(filename))

def generate_music(generator_model, latent_dim, n_vocab, length=500):
    """ Generate new music using the trained generator model """
    # Create random noise as input to the generator
    noise = np.random.normal(0, 1, (1, latent_dim))
    predictions = generator_model.predict(noise)

    # Scale back the predictions to the original range
    pred_notes = [x * (n_vocab / 2) + (n_vocab / 2) for x in predictions[0]]

    # Map generated integer indices to note names
    pitchnames = sorted(set(item for item in notes))
    int_to_note = dict((number, note) for number, note in
enumerate(pitchnames))
    pred_notes_mapped = [int_to_note[int(x)] for x in pred_notes]

    return pred_notes_mapped[:length]

if __name__ == '__main__':
    # Load the trained generator model
    generator_model = load_model("generator_model.h5")

    # Load the processed notes and get the number of unique pitches
    notes = get_notes()
    n_vocab = len(set(notes))

    # Generate new music sequence
    generated_music = generate_music(generator_model, LATENT_DIMENSION,
n_vocab)

    # Create a MIDI file from the generated music
    create_midi(generated_music, 'generated_music')

```

## A.3 Evaluation of melodies

### A.3.1 Manhattan Distance

```

import mido
import editdistance
import os

def manhattan_distance(list1, list2):
    distance = 0
    for sub1, sub2 in zip(list1, list2):
        sub_distance = 0
        for x, y in zip(sub1, sub2):
            sub_distance += abs(x - y)
        distance += sub_distance

```

```

    return distance

def compare_midi_files2(file1, file2):
    # Load midi files
    mid1 = mido.MidiFile(file1)
    mid2 = mido.MidiFile(file2)

    # Extract notes from midi files and group adjacent pitches together
    notes1 = []
    for msg in mido.merge_tracks(mid1.tracks):
        if 'note_on' in msg.type:
            pitch = msg.note
            if notes1 and pitch == notes1[-1][-1] + 1:
                # Append pitch to last group
                notes1[-1].append(pitch)
            else:
                # Create new group for pitch
                notes1.append([pitch])

    notes2 = []
    for msg in mido.merge_tracks(mid2.tracks):
        if 'note_on' in msg.type:
            pitch = msg.note
            if notes2 and pitch == notes2[-1][-1] + 1:
                # Append pitch to last group
                notes2[-1].append(pitch)
            else:
                # Create new group for pitch
                notes2.append([pitch])

    # Calculate similarity for each group of pitches
    similarity_scores = []
    return manhattan_distance(notes1, notes2)

original_folder_path = r'archive'
generated_file_path = r'archive\deut0568.mid'
#print(compare_midi_files2(original_folder_path, generated_file_path))

sim = 0

for filename in os.listdir(original_folder_path):
    if filename.endswith('.mid'):
        file_path = os.path.join(original_folder_path, filename)
        sim += compare_midi_files2(file_path, generated_file_path)

print(sim/1700)

```

### A.3.2 Pitch Class Transition Matrix

```

import os
import pretty_midi
import numpy as np
import matplotlib.pyplot as plt

```

```

def calculate_pitch_class_transition_matrix(midi_file):
    midi_data = pretty_midi.PrettyMIDI(midi_file)

    pitches = []
    for instrument in midi_data.instruments:
        for note in instrument.notes:
            pitches.append(note.pitch)

    pitch_classes = [pitch % 12 for pitch in pitches]

    transition_matrix = np.zeros((12, 12))
    for i in range(len(pitch_classes) - 1):
        prev_class = pitch_classes[i]
        current_class = pitch_classes[i + 1]
        transition_matrix[prev_class, current_class] += 1

    return transition_matrix

def calculate_average_transition_matrix(folder_path):
    """
    Calculates the average pitch class transition matrix from all MIDI files
    in a folder.

    Args:
        folder_path (str): Path to the folder containing MIDI files.

    Returns:
        numpy.ndarray: Average pitch class transition matrix.
    """
    total_matrix = np.zeros((12, 12))
    num_files = 0

    for filename in os.listdir(folder_path):
        if filename.endswith('.mid'):
            file_path = os.path.join(folder_path, filename)
            transition_matrix =
calculate_pitch_class_transition_matrix(file_path)
            total_matrix += transition_matrix
            num_files += 1

    # Check if any files were found
    if num_files > 0:
        return total_matrix / num_files
    else:
        print("No MIDI files found in the specified folder.")
        return None

def display_transition_matrix(matrix):
    plt.imshow(matrix, cmap='viridis', interpolation='nearest')
    plt.colorbar()
    plt.xlabel('Current Pitch Class')
    plt.ylabel('Previous Pitch Class')
    plt.title('Average Pitch Class Transition Matrix')
    plt.show()

# --- Example Usage ---
folder_path = r"D:\Music-GAN\Generate_Your_Own_Music-main\rnwgan_gen_dataset"
# Replace with your folder path
average_matrix = calculate_average_transition_matrix(folder_path)

```

```

if average_matrix is not None:
    display_transition_matrix(average_matrix)

```

### A.3.3 Note Length Transition Matrix

```

import os
import pretty_midi
import numpy as np
import matplotlib.pyplot as plt

def quantize_note_length(note_length, quantization_level):
    """Quantizes note length to a discrete category based on quantization
    level.

    Args:
        note_length (float): Note length in quarter notes.
        quantization_level (int): Number of quantization levels.

    Returns:
        int: Quantized note length category (0 to quantization_level-1).
    """
    quantized_index = int(note_length * quantization_level)
    return min(quantized_index, quantization_level - 1)

def calculate_note_length_transition_matrix(midi_file, quantization_level=8):
    midi_data = pretty_midi.PrettyMIDI(midi_file)

    note_lengths = []
    for instrument in midi_data.instruments:
        for note in instrument.notes:
            note_lengths.append(note.end - note.start) # Length in seconds

    quantized_lengths = [quantize_note_length(length, quantization_level)
                        for length in note_lengths]

    transition_matrix = np.zeros((quantization_level, quantization_level))
    for i in range(len(quantized_lengths) - 1):
        prev_length = quantized_lengths[i]
        current_length = quantized_lengths[i + 1]
        transition_matrix[prev_length, current_length] += 1

    return transition_matrix

def calculate_average_transition_matrix(folder_path, quantization_level=8):
    total_matrix = np.zeros((quantization_level, quantization_level))
    num_files = 0

    for filename in os.listdir(folder_path):
        if filename.endswith('.mid'):
            file_path = os.path.join(folder_path, filename)
            transition_matrix =
calculate_note_length_transition_matrix(file_path, quantization_level)
            total_matrix += transition_matrix
            num_files += 1

```

```

if num_files > 0:
    return total_matrix / num_files
else:
    print("No MIDI files found in the specified folder.")
    return None

def display_transition_matrix(matrix):
    plt.imshow(matrix, cmap='viridis', interpolation='nearest')
    plt.colorbar()
    plt.xlabel('Current Quantized Note Length')
    plt.ylabel('Previous Quantized Note Length')
    plt.title('Note Length Transition Matrix')
    plt.show()

# --- Example Usage ---
folder_path = r"D:\Music-GAN\Generate_Your_Own_Music-main\rnwgan_gen_dataset"
# Replace with your folder path
quantization_level = 8 # Adjust for desired granularity

average_matrix = calculate_average_transition_matrix(folder_path,
quantization_level)

if average_matrix is not None:
    display_transition_matrix(average_matrix)

```

### A.3.4 Utils (for KDE plotting)

```

# coding:utf-8
"""utils.py
Include distance calculation for evaluation metrics
"""

import sys
import os
import glob
import math
import sklearn
import numpy as np
from scipy import stats, integrate

# Calculate overlap between the two PDF
def overlap_area(A, B):
    pdf_A = stats.gaussian_kde(A)
    pdf_B = stats.gaussian_kde(B)
    return integrate.quad(lambda x: min(pdf_A(x), pdf_B(x)),
np.min((np.min(A), np.min(B))), np.max((np.max(A), np.max(B))))[0]

# Calculate KL distance between the two PDF
def kl_dist(A, B, num_sample=1000):
    pdf_A = stats.gaussian_kde(A)
    pdf_B = stats.gaussian_kde(B)
    sample_A = np.linspace(np.min(A), np.max(A), num_sample)
    sample_B = np.linspace(np.min(B), np.max(B), num_sample)
    return stats.entropy(pdf_A(sample_A), pdf_B(sample_B))

```

```

def c_dist(A, B, mode='None', normalize=0):
    c_dist = np.zeros(len(B))
    for i in range(0, len(B)):
        if mode == 'None':
            c_dist[i] = np.linalg.norm(A - B[i])
        elif mode == 'EMD':
            if normalize == 1:
                A_ = sklearn.preprocessing.normalize(A.reshape(1, -1),
norm='l1')[0]
                B_ = sklearn.preprocessing.normalize(B[i].reshape(1, -1),
norm='l1')[0]
            else:
                A_ = A.reshape(1, -1)[0]
                B_ = B[i].reshape(1, -1)[0]

            c_dist[i] = stats.wasserstein_distance(A_, B_)

        elif mode == 'KL':
            if normalize == 1:
                A_ = sklearn.preprocessing.normalize(A.reshape(1, -1),
norm='l1')[0]
                B_ = sklearn.preprocessing.normalize(B[i].reshape(1, -1),
norm='l1')[0]
            else:
                A_ = A.reshape(1, -1)[0]
                B_ = B[i].reshape(1, -1)[0]

            B_[B_ == 0] = 0.00000001
            c_dist[i] = stats.entropy(A_, B_)
    return c_dist

```

### A.3.5 KDE Plot of Intra Set Euclidean Distances

```

import midi
import glob
import numpy as np
import pretty_midi
import seaborn as sns
import matplotlib.pyplot as plt
from mgeval import core, utils
from sklearn.model_selection import LeaveOneOut

set1 = glob.glob(r'D:\Music-GAN\Generate_Your_Own_Music-main\gen_dataset\*')

num_samples = 100

set1_eval = {'total_used_pitch':np.zeros((num_samples,1))}
metrics_list = list(set1_eval.keys())
for i in range(0, num_samples):
    feature = core.extract_feature(set1[i])
    set1_eval[metrics_list[0]][i] = getattr(core.metrics(),
metrics_list[0])(feature)
# set2 = glob.glob('../data/set2/*.mid')
set2 = glob.glob('D:\Music-GAN\Generate_Your_Own_Music-main\org_dataset\*')
set2_eval = {'total_used_pitch':np.zeros((num_samples,1))}
for i in range(0, num_samples):

```



```

        feature = core.extract_feature(set2[i])
        set2_eval[metrics_list[0]][i] = getattr(core.metrics(),
metrics_list[0])(feature)

loo = LeaveOneOut()
loo.get_n_splits(np.arange(num_samples))
set1_intra = np.zeros((num_samples, len(metrics_list), num_samples))
set2_intra = np.zeros((num_samples, len(metrics_list), num_samples))
for i in range(len(metrics_list)):
    for train_index, test_index in loo.split(np.arange(num_samples)):
        set1_intra[test_index[0]][i][: -1] =
utils.c_dist(set1_eval[metrics_list[i]][test_index],
set1_eval[metrics_list[i]][train_index])
        set2_intra[test_index[0]][i][: -1] =
utils.c_dist(set2_eval[metrics_list[i]][test_index],
set2_eval[metrics_list[i]][train_index])

loo = LeaveOneOut()
loo.get_n_splits(np.arange(num_samples))
sets_inter = np.zeros((num_samples, len(metrics_list), num_samples))

for i in range(len(metrics_list)):
    for train_index, test_index in loo.split(np.arange(num_samples)):
        sets_inter[test_index[0]][i] =
utils.c_dist(set1_eval[metrics_list[i]][test_index],
set2_eval[metrics_list[i]])

plot_set1_intra = np.transpose(set1_intra,(1, 0,
2)).reshape(len(metrics_list), -1)
plot_set2_intra = np.transpose(set2_intra,(1, 0,
2)).reshape(len(metrics_list), -1)
plot_sets_inter = np.transpose(sets_inter,(1, 0,
2)).reshape(len(metrics_list), -1)
for i in range(0,len(metrics_list)):
    sns.kdeplot(plot_set1_intra[i], label='intra_set1')
    #sns.kdeplot(plot_sets_inter[i], label='inter')
    sns.kdeplot(plot_set2_intra[i], label='intra_set2')

    plt.ylim(0,12)
    plt.title(metrics_list[i])
    plt.xlabel('Euclidean distance')
    plt.legend()
    plt.show()

```

## A.4 WGAN with random noise injection and L2 regularization

```

import numpy as np
from keras.layers import Input, Dense, Reshape, Dropout, LSTM, Bidirectional
from tensorflow.keras.layers import BatchNormalization, LeakyReLU
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.optimizers.legacy import Adam
from tensorflow.keras.utils import to_categorical
from music21 import converter, instrument, note, chord, stream
from pathlib import Path
import matplotlib.pyplot as plt

```

```

from keras.layers import GaussianNoise
import tensorflow as tf

print(tf.__version__)

SEQUENCE_LENGTH = 100
LATENT_DIMENSION = 1000
BATCH_SIZE = 16
EPOCHS = 200
SAMPLE_INTERVAL = 1

def get_notes():
    """ Get all the notes and chords from the midi files """
    notes = []

    for file in Path("archive").glob("*.mid"):
        midi = converter.parse(file)

        print("Parsing %s" % file)

        notes_to_parse = midi.flat.notes

        for element in notes_to_parse:
            if isinstance(element, note.Note):
                notes.append(str(element.pitch))
            elif isinstance(element, chord.Chord):
                notes.append('.'.join(str(n) for n in element.normalOrder))

    return notes

def prepare_sequences(notes, n_vocab):
    """ Prepare the sequences used by the Neural Network """
    sequence_length = 100

    # Get all pitch names
    pitchnames = sorted(set(item for item in notes))

    # Create a dictionary to map pitches to integers
    note_to_int = dict((note, number) for number, note in
                        enumerate(pitchnames))

    network_input = []
    network_output = []

    # create input sequences and the corresponding outputs
    for i in range(0, len(notes) - sequence_length, 1):
        sequence_in = notes[i:i + sequence_length]
        sequence_out = notes[i + sequence_length]
        network_input.append([note_to_int[char] for char in sequence_in])
        network_output.append(note_to_int[sequence_out])

    n_patterns = len(network_input)

    # Reshape the input into a format compatible with LSTM layers
    network_input = np.reshape(network_input, (n_patterns, sequence_length,
1))

    # Normalize input between -1 and 1

```

```

network_input = (network_input - float(n_vocab) / 2) / (float(n_vocab) /
2)
network_output = to_categorical(network_output, num_classes=n_vocab) #
Use to_categorical from TensorFlow's Keras

return network_input, network_output # Add this return statement

def create_midi(prediction_output, filename):
    """ convert the output from the prediction to notes and create a midi file
    from the notes """
    offset = 0
    output_notes = []

    # create note and chord objects based on the values generated by the model
    for item in prediction_output:
        pattern = item[0]
        # pattern is a chord
        if ('.' in pattern) or pattern.isdigit():
            notes_in_chord = pattern.split('.')
            notes = []
            for current_note in notes_in_chord:
                new_note = note.Note(int(current_note))
                new_note.storedInstrument = instrument.Piano()
                notes.append(new_note)
            new_chord = chord.Chord(notes)
            new_chord.offset = offset
            output_notes.append(new_chord)
        # pattern is a note
        else:
            new_note = note.Note(pattern)
            new_note.offset = offset
            new_note.storedInstrument = instrument.Piano()
            output_notes.append(new_note)

    # increase offset each iteration so that notes do not stack
    offset += 0.5

    midi_stream = stream.Stream(output_notes)
    midi_stream.write('midi', fp='{}.mid'.format(filename))

def wasserstein_loss(y_true, y_pred):
    return tf.keras.backend.mean(y_true * y_pred)

class GAN():
    def __init__(self, rows):
        self.seq_length = rows
        self.seq_shape = (self.seq_length, 1)
        self.latent_dim = 1000
        self.disc_loss = []
        self.gen_loss = []

        optimizer = Adam(0.0002, 0.5)

        # Build and compile the discriminator
        self.discriminator = self.build_discriminator()
        self.discriminator.compile(loss=wasserstein_loss, optimizer=optimizer,
metrics=['accuracy'])

```

```

# Build the generator
self.generator = self.build_generator()

# The generator takes noise as input and generates note sequences
z = Input(shape=(self.latent_dim,))
generated_seq = self.generator(z)

# For the combined model we will only train the generator
self.discriminator.trainable = False

# The discriminator takes generated images as input and determines
validity
validity = self.discriminator(generated_seq)

# The combined model (stacked generator and discriminator)
# Trains the generator to fool the discriminator
self.combined = Model(z, validity)
self.combined.compile(loss=wasserstein_loss, optimizer=optimizer)

def build_discriminator(self):
    model = Sequential()
    model.add(LSTM(512, input_shape=self.seq_shape,
return_sequences=True,))
    model.add(GaussianNoise(0.1))
    model.add(Bidirectional(LSTM(512)))
    model.add(GaussianNoise(0.1))
    model.add(Dense(512, kernel_regularizer='l2'))
    model.add(GaussianNoise(0.1))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dense(256, kernel_regularizer='l2'))
    model.add(GaussianNoise(0.1))
    model.add(LeakyReLU(alpha=0.2))

    # Adding Minibatch Discrimination
    model.add(Dense(100, kernel_regularizer='l2'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.5))
    model.add(Dense(1))
    model.summary()

    seq = Input(shape=self.seq_shape)
    validity = model(seq)

    return Model(seq, validity)

def build_generator(self):
    model = Sequential()
    model.add(Dense(256, input_dim=self.latent_dim,
kernel_regularizer='l2'))
    model.add(GaussianNoise(0.1))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(512, kernel_regularizer='l2'))
    model.add(GaussianNoise(0.1))
    model.add(LeakyReLU(alpha=0.2))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(1024, kernel_regularizer='l2'))
    model.add(GaussianNoise(0.1))

```

```

        model.add(LeakyReLU(alpha=0.2))
        model.add(BatchNormalization(momentum=0.8))
        model.add(Dense(np.prod(self.seq_shape), activation='tanh',
kernel_regularizer='l2'))
        model.add(GaussianNoise(0.1))
        model.add(Reshape(self.seq_shape))
        model.summary()

        noise = Input(shape=(self.latent_dim,))
        seq = model(noise)

        return Model(noise, seq)

    def train(self, epochs, batch_size=128, sample_interval=50):

        # Load and convert the data
        notes = get_notes()
        n_vocab = len(set(notes))
        X_train, y_train = prepare_sequences(notes, n_vocab)

        # Adversarial ground truths
        real = -1*np.ones((batch_size, 1))
        fake = np.ones((batch_size, 1))

        # Training the model
        for epoch in range(epochs):

            # Training the discriminator
            # Select a random batch of note sequences
            idx = np.random.randint(0, X_train.shape[0], batch_size)
            real_seqs = X_train[idx]

            #noise = np.random.choice(range(484), (batch_size,
self.latent_dim))
            #noise = (noise-242)/242
            noise = np.random.normal(0, 1, (batch_size, self.latent_dim))

            # Generate a batch of new note sequences
            gen_seqs = self.generator.predict(noise)

            # Train the discriminator
            d_loss_real = self.discriminator.train_on_batch(real_seqs, real)
            d_loss_fake = self.discriminator.train_on_batch(gen_seqs, fake)
            d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

            # clip_value = 0.01
            # for l in self.discriminator.layers:
            #     weights = l.get_weights()
            #     weights = [
            #         np.clip(w, -clip_value, clip_value) for w in weights
            #     ]
            #     l.set_weights(weights)

            # Training the Generator
            noise = np.random.normal(0, 1, (batch_size, self.latent_dim))

            # Train the generator (to have the discriminator label samples as
real)

```

```

        g_loss = self.combined.train_on_batch(noise, real)

        # Print the progress and save into loss lists
        if epoch % sample_interval == 0:
            print ("%d [D loss: %f, acc.: %.2f%%] [G loss: %f]" % (epoch,
d_loss[0], 100*d_loss[1], g_loss))
            self.disc_loss.append(d_loss[0])
            self.gen_loss.append(g_loss)

        self.generate(noises)
        self.plot_loss()

    def generate(self, input_notes):
        # Get pitch names and store in a dictionary
        notes = input_notes
        pitchnames = sorted(set(item for item in notes))
        int_to_note = dict((number, note) for number, note in
enumerate(pitchnames))

        # Use random noise to generate sequences
        noise = np.random.normal(0, 1, (1, self.latent_dim))
        predictions = self.generator.predict(noise)

        pred_notes = [x*242+242 for x in predictions[0]]

        # Map generated integer indices to note names, with error handling
        pred_notes_mapped = []
        for x in pred_notes:
            index = int(x)
            if index in int_to_note:
                pred_notes_mapped.append(int_to_note[index])
            else:
                # Fallback mechanism: Choose a default note when the index is
out of range
                pred_notes_mapped.append('C5') # You can choose any default
note here

        create_midi(pred_notes_mapped, 'gan_final')

    def plot_loss(self):
        plt.plot(self.disc_loss, c='red')
        plt.plot(self.gen_loss, c='blue')
        plt.title("GAN Loss per Epoch")
        plt.legend(['Discriminator', 'Generator'])
        plt.xlabel('Epoch')
        plt.ylabel('Loss')
        plt.savefig('GAN_Loss_per_Epoch_final.png', transparent=True)
        plt.close()

if __name__ == '__main__':
    gan = GAN(rows=SEQUENCE_LENGTH)
    gan.train(epochs=EPOCHS, batch_size=BATCH_SIZE,
sample_interval=SAMPLE_INTERVAL)

    # Save the generator and discriminator models
    gan.generator.save("generator_model.h5")
    gan.discriminator.save("discriminator_model.h5")

```