GENERATION OF MUSIC USING LONG SHORT TERM MEMORY NETWORKS

A MINI PROJECT REPORT

Submitted by

MUTHAMIZH P J (1920110030)

POOJHA M (1920110033)

SHRUSTIGA S R (1920110046)

in partial fulfilment for the award of the degree

of

BACHELOR OF TECHNOLOGY

IN

ARTIFICIAL INTELLIGENCE AND
DATA SCIENCE

SONA COLLEGE OF TECHNOLOGY, SALEM-5

(Autonomous)

ANNA UNIVERSITY: CHENNAI 600 025

NOVEMBER 2023

ANNA UNIVERSITY CHENNAI 600 025

BONAFIDE CERTIFICATE

Certified that this project report **"GENERATION OF MUSIC USING LSTM NETWORKS"** is the bona-fide work of "**MUTHAMIZH P J (1920110030), POOJHA M (1920110033), SHRUSTIGA S R (1920110046)"** who carried out the project work under my supervision.

SIGNATURE	**SIGNATURE**
Dr. J. Akilandeswari	Mr. J. L. Aldo Stalin
Professor	Assistant Professor
HEAD OF THE DEPARTMENT	**SUPERVISOR**
Department of Information Technology	Department of Information Technology,
Sona College of Technology,	Sona College of Technology,
Salem-636 005.	Salem-636 005.

Submitted for Mini Project viva voce examination held on 16.11.2023 to 18.11.2023

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ACKNOWLEDGEMENT

First and foremost, we thank to **Power of Almighty** for showing us inner peace and for all blessings. Special gratitude to our parents, for showing their support and love always.

We express our sincere thanks to Chairman **Sri.C.Valliappa** and Principal **Dr.S.R.R.Senthil Kumar** for providing adequate facilities to complete the project.

We are immensely grateful to Head of Information Technology, **Dr.J.Akilandeswari** for the continuous encouragement to complete the project.

We express our heartfelt thanks to our project supervisor **Mr.J.L.Aldo Stalin** for his valuable guidance and fruitful discussions throughout the course of the project work.

We express our heartfelt thanks to our project supervisor **Mr.C.Santhosh Kumar** for his valuable guidance and fruitful discussions throughout the course of the project work.

We feel proud in sharing this success with all our staff members and friends who helped directly or indirectly in completing this project successfully.

ABSTRACT

This project explores the fascinating world of generating music using machine learning and deep learning techniques. The project focuses on a special type of neural network called Long Short-Term Memory (LSTM), using the Keras library in Python, to compose music sequences. The project starts by introducing the idea of using AI to create music. It explains key terms like Recurrent Neural Networks (RNNs) and LSTM networks, which are excellent at capturing musical patterns due to their ability to remember information over long sequences. To make music generation possible, the Music21 Python toolkit is introduced. This toolkit helps work with musical notation from MIDI files and enables the manipulation of musical elements like notes and chords. In conclusion, this project showcases the power of AI and LSTM neural networks in creating music. The project contributes to the exploration of AI's creative potential and its role in reshaping the world of music composition.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

- **LSTM :** Long Short Term Memory
- **MIDI :** Musical Interface Digital Instrument

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION OF THE PROJECT

To develop a music generation system utilizing LSTM neural networks to analyze and generate music from MIDI files. We aim to capture the complex musical patterns and create AI generated compositions. With the help of Python inbuilt libraries, the MIDI files are processed and the music is generated. Our project focuses on generating music with the help of libraries. It focuses on generation of single instrumental music on the user preferences. Music generation using LSTM (Long Short-Term Memory) networks is a captivating field that combines the power of artificial intelligence with the artistry of music

This innovative project focuses on training deep learning models to autonomously create music compositions. By leveraging LSTM neural networks, it becomes possible to capture intricate musical patterns and temporal dependencies, enabling the generation of melodies, harmonies, and rhythms. This intersection of technology and creativity holds immense potential, from crafting background scores for multimedia to providing inspiration to musicians and composers. We've broken down our data into two main categories: 'Notes' and 'Chords'. Each of these categories has its own unique role in representing and interpreting musical information.

Note Objects

Note objects contain detailed information about particular musical notes in the dataset. This information includes:

- **Pitch**: The frequency of the note is represented by the letters (a, b, c, d, e, f, g), with A representing the highest note pitch and G representing the lowest note pitch.

- **Octave**: The octave parameter refers to the specific pitch range used in a musical composition.
- **Offset**: The offset parameter indicates the timing and sequence of the note in the musical composition.

Chord Objects

A chord is a container for a set of notes that play at the same time. This addscomplexity and richness to the musical arrangement.

1.1.1 LSTM (LONG SHORT TERM MEMORY)

LSTM stands for Long Short-Term Memory, and it is a type of recurrent neural network (RNN) that is specifically designed to learn long-term dependencies in sequential data. RNNs are a type of neural network that are well-suited for processing sequential data, such as text, speech, and time series data. However, traditional RNNs can suffer from the vanishing gradient problem, which makes it difficult to learn long-term dependencies.

LSTMs solve the vanishing gradient problem by using a special type of memory cell that can store information for long periods of time. The LSTM cell has three gates: the input gate, the forget gate, and the output gate. These gates control how information is added to, removed from, and read from the memory cell. The cell has two states **Cell state** and **Hidden State**. They are continuously updated and carry the information from the previous to the current time steps. The cell state is the “long-term” memory, while the hidden state is the “short-term” memory.

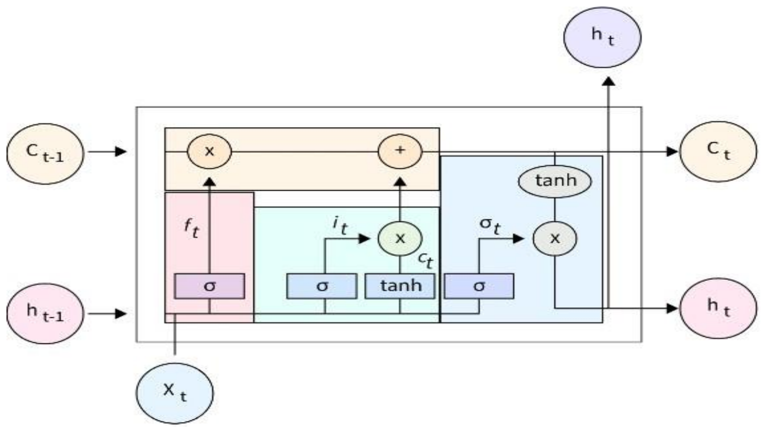


Fig.1.1 Architecture of LSTM

Forget Gate:

Forget gate is responsible for deciding what information should be removed from the cell state. It takes in the hidden state of the previous time-step and the current input and passes it to a Sigma Activation Function, which outputs a value between 0 and 1, where 0 means forget, and 1 means keep.

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

Input Gate:

The Input Gate considers the current input and the hidden state of the previous time step. The input gate is used to update the cell state value. It has two parts. The first part contains the Sigma activation function. Its purpose is to decide what percent of the information is required. The second part passes the two values to a Tanh activation function. It aims to map the data between -1 and 1. To obtain the relevant information required from the output of Tanh, we multiply it by the output of the Sigma function. This is the output of the Input gate, which updates the cell state.

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

$$C_t = \tanh\,(W_c \cdot [h_{t-1}, x_t] + b_f)$$

Output Gate:

The output gate returns the hidden state for the next time stamp. The output gate has two parts. The first part is a Sigma function, which serves the same purpose as the other two gates, to decide the percent of the relevant information required. Next, the newly updated cell state is passed through a Tanh function and multiplied by the output from the sigma function. This is now the new hidden state.

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

$$h_t = \mathrm{o_t} \cdot \tanh(C_t)$$

Cell State:

The forget gate and input gate update the cell state. The cell state of the previous state is multiplied by the output of the forget gate. The output of this state is then summed with the output of the input gate. This value is then used to calculate **hidden state** in the output gate.

$$c_t = f_t * c_{t-1} \text{ , } i_t * \overline{c_t}$$

1.1.2 BIDIRECTIONAL LSTM

A Bidirectional Long Short-Term Memory (BiLSTM) is a type of recurrent neural network (RNN) architecture that processes sequential data in both forward and backward directions simultaneously. It is an extension of the standard LSTM (Long Short-Term Memory) network, designed to capture dependencies in sequential data more effectively. BiLSTMs are commonly used in various natural language processing tasks, speech recognition, and other sequence modelling applications. A Bidirectional LSTM extends this by having two sets of hidden states: one moving forward through the sequence, and one moving backward. This bidirectional flow allows the network to capture context from both past and future time steps.

Forward layer:

The forward layer processes the input sequence from the beginning to the end. At each time step, the forward LSTM computes an output and updates its hidden state. The hidden state at time step t, denoted as h_t, is influenced by the previous hidden state h_(t-1) and the current input x_t. The forward LSTM generates a sequence of forward hidden states, {h_t_forward}.

Backward layer:

The backward layer processes the input sequence in reverse order, from the end to the beginning. At each time step, the backward LSTM computes an output and updates its hidden state. The hidden state at time step t in the backward pass is influenced by the previous hidden state h_(t+1) in the forward pass and the current input x_t. The backward LSTM generates a sequence of backward hidden states, {h_t_backward}.

Combining Forward and Backward layer:

Once both the forward and backward layers are complete, you have two sequences of hidden states, {h_t_forward} and {h_t_backward}. To obtain the final output at each time step, you can concatenate the corresponding forward and backward hidden states: h_t_combined = [h_t_forward, h_t_backward]. This combined hidden state captures information from both past and future context, allowing the model to make more informed predictions.

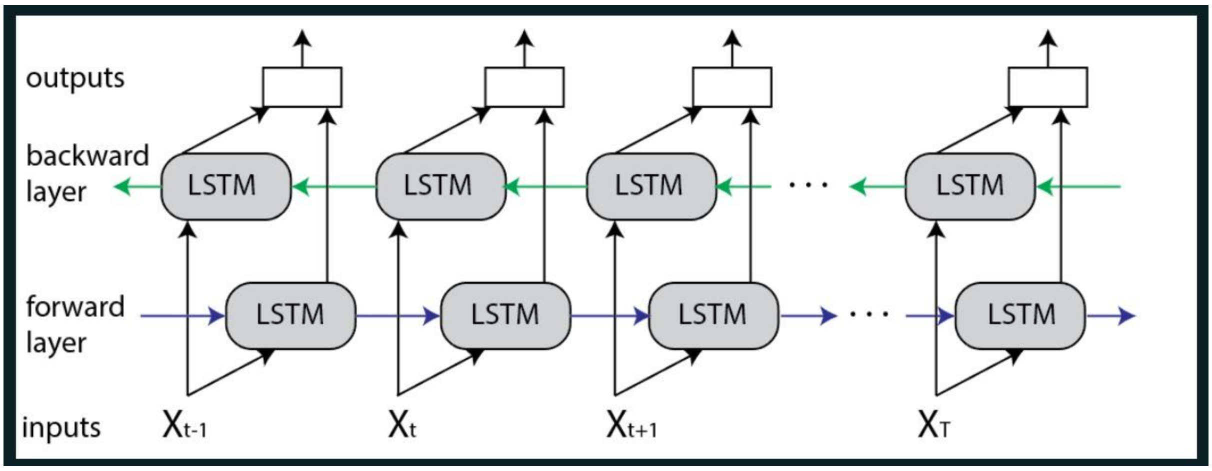


Fig.1.2 Architecture of Bidirectional LSTM

BiLSTMs are trained using backpropagation through time (BPTT), similar to regular LSTMs. Gradients are computed from both forward and backward passes, allowing the network to learn the bidirectional dependencies in the data. A Bidirectional LSTM has twice the number of parameters compared to a unidirectional LSTM, as it maintains separate parameters for the forward and backward directions.

1.2 SCOPE OF THE PROJECT

- Music composers and songwriters can use AI-generated music as inspiration. AI models can suggest chord progressions, melodies, and harmonies based on specific styles or artist preferences. This assists musicians in overcoming creative blocks and exploring new musical directions.
- AI-generated music can be used as a starting point for producers and arrangers. It can provide foundational elements like drum patterns, bass lines, or even entire instrumental sections, which can be further customized and enhanced to create full-fledged songs.
- Music streaming platforms can employ AI-generated music to create personalized playlists for users. By understanding individual preferences and listening habits, the platform can curate unique playlists, introducing users to new music that aligns with their tastes.
- Film directors, game developers, and multimedia creators can utilize AI-generated music to produce custom soundtracks. AI can adapt compositions based on the mood, tempo, and scene dynamics, ensuring a seamless integration of music with visual elements.
- AI-generated music can be used as educational material. Students studying music composition and theory can analyze AI-generated compositions, gaining insights into different styles, structures, and techniques, thereby enhancing their learning experience.

1.3 LITERATURE REVIEW

The application of artificial intelligence, specifically Long Short-Term Memory (LSTM) neural networks, in the generation of musical expression. Musical expression is defined as the performer's emotional interpretation of music, which encompasses aspects like dynamics, tempo, and articulation. The already proposed system aims to harness AI techniques to generate musical expression, with potential applications in various domains, including generative music, digital audio workstations, and score interpretation systems.

Generating musical expression is a complex challenge, as it involves capturing the nuanced, subjective, and often elusive qualities of human performances. Musical expression comprises the emotional depth and subtleties that musicians infuse into their renditions, making it a hallmark of authenticity in music. This text outlines the motivation, challenges, and potential solutions in applying AI to this intricate domain. The musical expression is associated with the performer's interpretation, involving changes in dynamics and tempo. Musicians, while following the structural elements of the composition, introduce their unique emotional nuances into their performances. These nuances can be subtle or pronounced and are often reflective of the performer's emotional state, experience, and personal style. This human touch is what gives a live performance its charm and uniqueness.

The system has the capacity to generate expressiveness by recognizing and applying these rules to a musical piece is the central theme of the paper. This capability has broad applications, spanning generative music, digital audio workstations (DAWs) for sound manipulation, score interpretation systems, and more. An underlying challenge in this pursuit is the fundamental properties of human creativity. Human performances are inherently variable and imperfect, and paradoxically, these imperfections are often celebrated as hallmarks of genuine perfection in the performing arts. Consequently, achieving a high level of expressiveness in AI-generated music requires specialized analytical tools to handle aspects like tempo changes in audio signals, as well as specialized generation tools to simulate human-like expressiveness.

One significant challenge in evaluating systems for generating musical expression is the subjective nature of music itself. Music is deeply connected to human emotions and imperfections, making it challenging to observe, analyze, and mathematically represent all its rules comprehensively. While some rule-based systems have demonstrated success, they necessitate the meticulous development of precise sets of musical rules, which can be demanding and inconvenient from a research perspective. Additionally, statistical models and regression systems often struggle due to the inherent complexity and diversity of musical expressions.

The existing models and approaches, such as the Canonical Correlation Analysis (CCA) technique, which correlates short pieces of the score with performance parameters to predict tempo and dynamics. Although CCA has shown promise, it doesn't always result in musically pleasing outcomes. Another example is a system developed by Roberto Bresin, which utilized artificial neural networks (ANNs) to learn piano music expression. However, this system had limitations due to the technology available at the time, and it required extensive manual intervention in the form of phrase grouping.

Magenta, an open-source project from Google Brain, which explores machine learning's role in creating art, including music generation. Performance RNN, a part of Magenta, employs recurrent neural networks to generate music with expressive dynamics and timing. However, it's noted that Magenta doesn't provide a model for generating expressive interpretations for existing music files, focusing more on generating new compositions.

The technical background section of the text introduces key musical definitions, including notes, pitch, time signature, tempo, and musical phrases. It also delves into the fundamental concepts of artificial neural networks (ANNs), which mimic the working principle of the human brain. ANNs typically consist of multiple layers with many neurons, which perform computations involving weighted inputs and activation functions.

The text highlights the limitation of standard ANNs when it comes to handling sequential data. Standard ANNs lack the capacity to capture the temporal dependencies crucial in music. For example, the recurrence of themes at the beginning and end of a music piece necessitates a more sophisticated model.

To address this limitation, the text introduces Long Short-Term Memory (LSTM) networks, which have been designed to handle sequential data efficiently. LSTMs resolve the issue of short-term memory in standard RNNs by introducing a separate long-term memory state (cell state) and a gating mechanism that controls its modifications. LSTMs are well-suited for modeling music, which inherently involves the interplay of various temporal aspects.

The proposed solution in the paper centers around leveraging LSTM recurrent neural networks to model and generate musical expression.

This indicates that the results obtained from this proposed system will be evaluated by comparing the generated performances with human performances. Additionally, an analysis of the results of a survey conducted in this context will contribute to the evaluation process. This suggests that the effectiveness of the AI-generated musical expression will be assessed against the standards of human musical expression, which is inherently subjective and emotionally nuanced.

1.4 EXISTING SYSTEM DISADVANTAGES

- One notable disadvantage is the lack of information about where the music starts or ends. This lack of context can pose challenges when attempting to generate coherent musical expressions, as music often has distinct sections and structures. Without clear information on the boundaries of musical pieces, the generated content may lack the expected structure and coherence.

- Additionally, the introduction of sliding windows as part of the method results in a substantial amount of input data. The use of sliding windows is a technique to process sequential data, but it can lead to a large volume of input data to be handled by the model. This can increase the computational load and memory requirements.

- To mitigate the computational challenges and minimize the need for frequent weight updates in the neural network, they use of mini-batches. Mini-batches involve splitting the training data into smaller fragments or batches. Instead of updating the network's parameters at each time step, the average gradient from the entire batch is calculated and used for weight modifications. While this approach reduces computational demands and stabilizes convergence, it also introduces a challenge in determining the appropriate batch size.

- This specifies that the batch size for the tempo model was set to 32 windows, while the dynamics model used a batch size of 16 windows. Smaller batch sizes were found to provide more accurate results but significantly prolonged training times. This highlights a trade-off between computational efficiency and model accuracy.

- Despite these disadvantages and challenges, the text underscores the promising results achieved by the method. Even with a small training dataset, the model demonstrates an ability to recognize dependencies and generate performances that closely resemble human performance. This is supported by a comparison of the generated dynamics and tempo with real performances, as well as the findings of a survey that suggests listeners find it challenging to distinguish between human and computer-generated performances.

- However, this also acknowledges that the nature of musical expression is inherently subjective and "soft," making it challenging to treat as strictly as problems with well-defined correct answers. Furthermore, the method is limited in scope as it is currently applied only to a small selection of musical pieces, specifically Chopin's mazurkas. To expand the capabilities of the program and adapt it to different musical styles and time signatures, a larger and more diverse dataset would be required.
- In summary, the disadvantages and limitations of the proposed method for generating musical expression using LSTM neural networks include the lack of clear music boundaries, the introduction of a significant amount of input data with sliding windows, the challenge of determining an appropriate batch size, and the need to expand the system's capabilities beyond its current limited scope. Despite these challenges, the method shows promise in approximating human musical performance.

1.5 HARDWARE AND SOFTWARE REQUIREMENTS

1.5.1 SOFTWARE REQUIREMENT

1. **Python**

    - Python 3.x: The project code appears to be written in Python, hence Python 3.x should be installed. The specific version used in development is recommended for compatibility.
    - Required Libraries: Ensure the installation of the necessary Python libraries used in the project, including but not limited to:
        - pandas
        - numpy
        - tensorflow
        - seaborn
        - pretty_midi
        - fluidsynth
        - matplotlib
        - collections

2. **Jupyter Notebook:** Installation of Jupyter Notebook is required for interactive execution, documentation, and presentation of the code. Ensure the installation of Jupyter Notebook compatible with the Python version used.

    Jupyter Notebook is an open-source web application that allows the creation and sharing of documents that contain live code, equations, visualizations, and narrative text. It is widely used in data science, research, and education for interactive Features and Functionality

- **Code Execution:** Jupyter Notebook facilitates the execution of Python code cells, enabling the analysis of MIDI data using various libraries and tools.
- **Data Visualization:** The integration of visualization libraries within Jupyter Notebook allows the creation of interactive and informative visualizations for better data understanding.
- **Markdown Support:** The inclusion of markdown cells enables the documentation of the analysis process, providing explanations, insights, and interpretations alongside the code and visualizations.
- **Data Exploration:** Jupyter Notebook's interactive interface enables the exploration of MIDI data in real-time, allowing for dynamic adjustments and instant visual feedback.e computing and data analysis tasks.

3. **Google Colab:** As the code appears to be originally from a Google Colab environment, ensure compatibility with the Google Colab environment for seamless execution and integration with Google Drive.
4. **Data Analysis Tools:** Using a variety of software tools and packages is crucial to the project's data analysis process in order to analyse, manipulate, and explore data effectively. It has been determined that the following libraries –

    - **NumPy:** Due to its widespread use in managing a wide range of high-level mathematical functions as well as multi-dimensional arrays and matrices, this basic Python library will be utilised extensively. It will be essential to the numerical processing and modification of the musical data, allowing computing jobs to be completed quickly and efficiently.

- **Pandas:** We will be able to more efficiently organise, clean, and analyse the musical data by utilising the potent data manipulation capabilities of Pandas. We will be able to easily complete difficult data operations, data alignment, and data cleaning activities thanks to its user-friendly data structures, such DataFrame. The library will play a crucial role in offering incisive data summaries and enabling a thorough comprehension of the fundamental musical structures and patterns. and tools need to be installed and configured:

5. **Operating System Compatibility:** Ensure that all the software components are compatible with the operating system used (e.g., Windows, macOS, or Linux) for the development and execution of the project.

6. **Data Management and Visualization Tools:**

   - Installation of appropriate data management tools to handle and manipulate large datasets effectively.

   - Ensure the availability of visualization tools compatible with Jupyter Notebook for creating clear and insightful data visualizations.

   - To ensure the creation of comprehensive and insightful data visualizations in the Jupyter Notebook environment, we will focus on integrating visualization tools that seamlessly align with Jupyter Notebook's interface. The inclusion of the following tools is planned:

- **Seaborn**: This robust data visualization library, which is built on the foundations of Matplotlib, will play a key role in producing visually captivating statistical graphics. Its adeptness in handling intricate datasets and its user-friendly interface will be instrumental in crafting informative and visually engaging visualizations.

- **Matplotlib:** Serving as the backbone for a wide spectrum of data visualization tasks, Matplotlib will serve as the primary tool for generating an array of static, interactive, and dynamic visualizations. Its utilization will ensure a comprehensive portrayal of the inherent patterns within the musical data, contributing to a nuanced understanding of the dataset.

7. **Package Management:** Utilize a package management system such as pip or conda to ensure the installation, upgrading, and management of Python packages used in the project.

8. **Code Execution Environment:** Ensure the availability of a suitable code execution environment that supports the required software components and libraries for efficient code execution and debugging.

1.5.2 HARDWARE REQUIREMENT

1. **Processor and Memory:**

   - A multi-core processor (preferably quad-core or higher) to enable parallel processing and efficient handling of data-intensive operations.

   - At least 8 GB of RAM (Random Access Memory) to ensure smooth execution and processing of a large number of MIDI files. For optimal performance, 16 GB or more is recommended, especially for handling extensive data sets.

2. **Storage:** Adequate storage space to accommodate the MIDI files, the processed data, and any additional files generated during the analysis. The size of the storage will depend on the size of the MIDI dataset, but several gigabytes of free space are typically required.

3. **Graphics Processing Unit (GPU):** While not mandatory for this specific project, the use of a dedicated GPU can significantly accelerate data processing, especially for complex visualizations and computationally intensive tasks. GPUs with good parallel processing capabilities can enhance the performance of certain data analysis operations.

4. **Operating System:** The project is compatible with various operating systems, including Windows, macOS, and Linux. Ensure that the chosen operating system is compatible with the required software components.

5. **Internet Connectivity:** Reliable internet connectivity may be required to access external data sources, libraries, or resources during the data analysis process. Additionally, it enables the seamless integration of online tools and resources into the project.

6. **Backup and Security:** Regular backup solutions should be in place to prevent data loss or corruption. Implement security measures to protect sensitive data and ensure the integrity of the analysis process.

7. **Additional Peripherals:**

   - A high-resolution monitor for effective visualization of data and analysis outputs.

CHAPTER 2

DESIGN AND IMPLEMENTATION

2.1 PROPOSED SYSTEM

The provided Python script utilizes LSTM (Long Short-Term Memory) neural networks to generate music from MIDI files. It comprises two primary functions, 'train_network' and 'generate,' along with supporting functions. The code's objective is to establish a music composition system encompassing neural network training and music generation.

In the training phase, musical notes and chords are gathered from MIDI files in a specified directory using the 'get_notes' function, with the 'music21' library aiding in parsing and data extraction. Data is pre-processed to create input and output sequences, facilitated by the 'prepare_sequences' function. This step involves mapping notes to integers and preparing the data for LSTM compatibility. A Sequential neural network model is constructed using Keras, featuring three LSTM layers, each accompanied by Batch Normalization and Dropout layers, and two Dense layers. The model is configured with categorical cross-entropy loss, the RMSprop optimizer, and loaded pre-trained weights. However, the 'train' function, crucial for completing the training, is missing from the provided code.

In the music generation phase, the 'generate' function employs the trained neural network model. It commences by loading training notes from a file and preparing input data to align with the model's format. The 'create_network' function constructs the neural network model, and pre-trained weights are loaded. Music generation starts with a random input sequence. The model iteratively generates a sequence of notes and chords, repeating this process 500 times to form a list of predicted notes. Predicted notes are transformed into note and chord objects, resulting in the creation of a MIDI file using the 'create_midi' function. The output is saved as 'test_output.mid,' containing the generated music. While the code serves as a strong foundation for music generation, several areas for improvement and enhancements should be considered. Comprehensive documentation and comments should be incorporated to enhance code readability and understanding. The missing 'train' function for model training should be added for a complete training process.

Hyperparameter tuning is essential to optimize model architecture and settings, including the number of LSTM layers, dropout rates, and sequence length. Experimentation and validation can fine-tune these aspects. Integrating music theory knowledge can enhance musical coherence, applying rules like chord progressions and key signatures. Expanding and diversifying the training dataset can improve the model's creative capacity. Evaluation mechanisms using metrics like pitch accuracy and harmony can assess model performance. Developing a user-friendly interface or integration into music software applications enhances accessibility. Real-time music generation features represent an exciting avenue for further development. Incorporating mechanisms for model persistence allows users to save and load trained models. Parallelization, leveraging GPU acceleration and parallel processing, enhances computational efficiency for training and music generation.

2.1.1 ADVANTAGES OF PROPOSED SYSTEM

Firstly, it provides a creative and automated approach to music composition, allowing musicians and composers to explore new musical ideas and generate compositions more efficiently. This can be particularly valuable for artists looking for inspiration or working on projects that require a large volume of music.

The system's use of deep learning techniques, specifically LSTM networks, enables it to capture complex patterns and structures in music, resulting in compositions that are not only coherent but also exhibit a level of creativity and diversity that can be challenging to achieve through traditional means. The expandable and diverse training dataset contributes to the system's adaptability. As the dataset grows, the model becomes capable of generating a wider range of musical styles and genres, enhancing its versatility. Finally, the system's ability to save and load trained models ensures that users can preserve their progress and continue working on their compositions at their convenience.

2.2 ARCHITECTURE OF THE PROPOSED SYSTEM

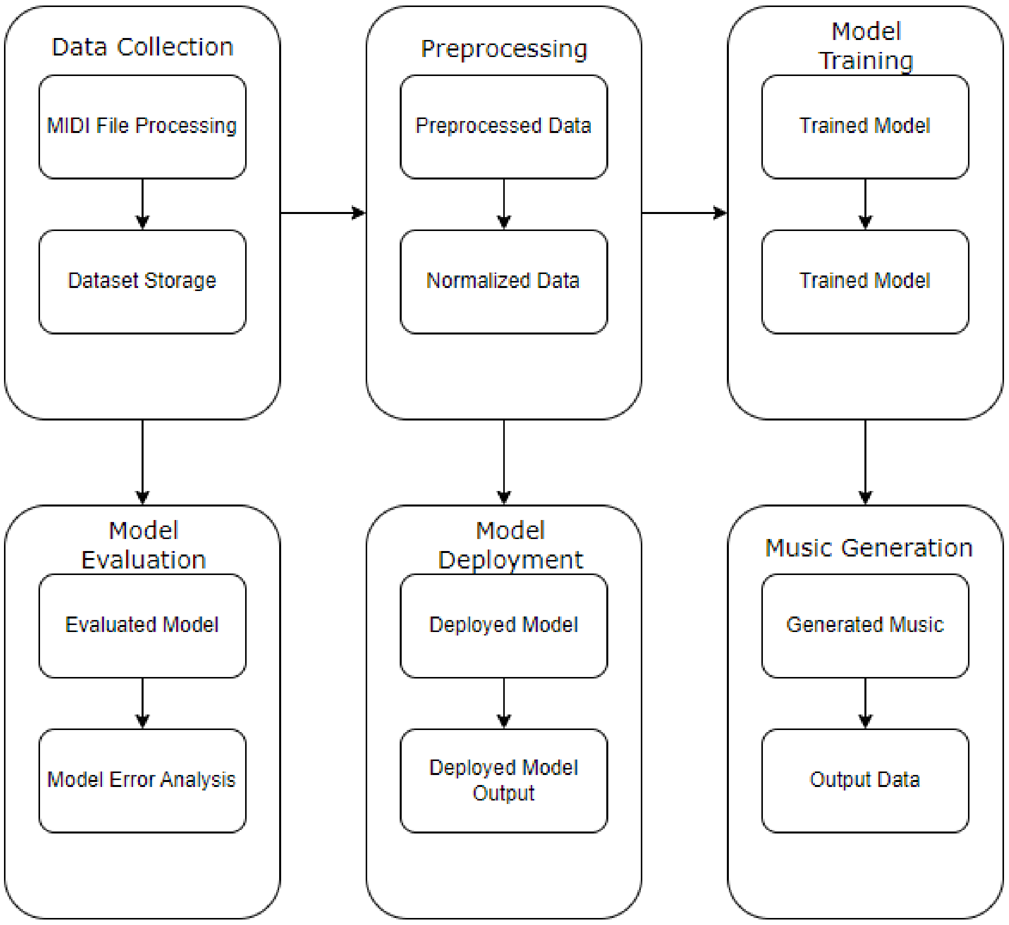


Fig.2.1 Project Architecture

2.3 MODULES OF THE PROJECT

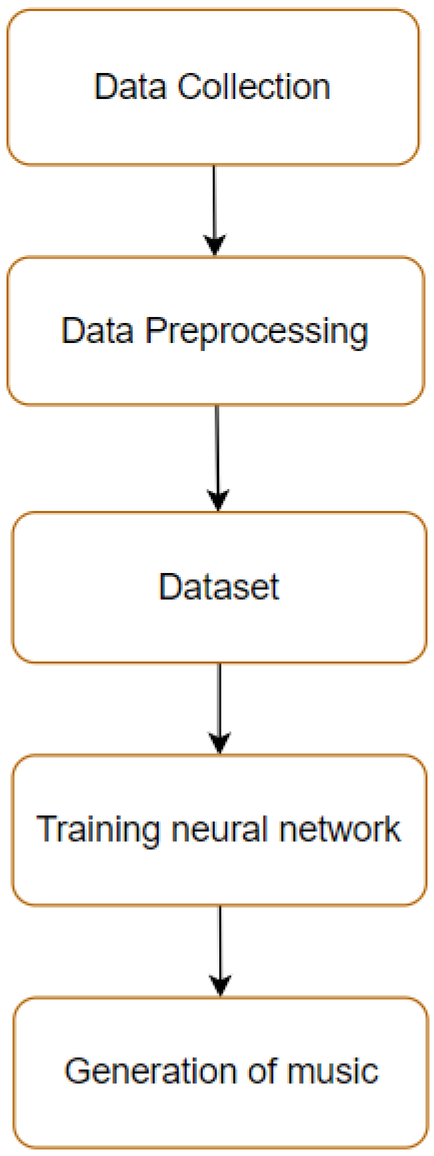


Fig.2.2 Modules

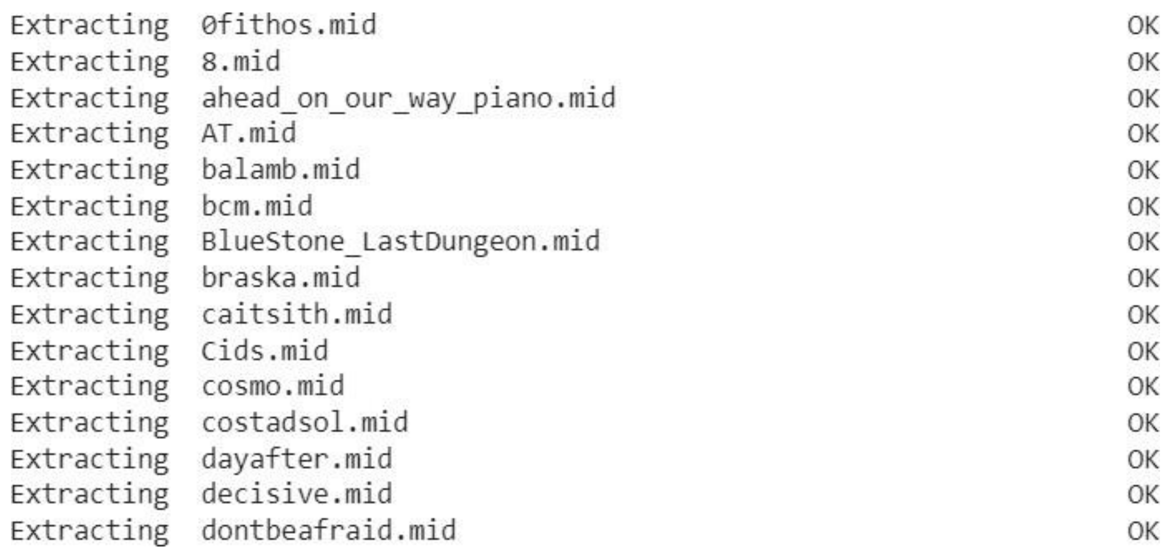
2.3.1 DATASET PREPROCESSING

In the process of generating music using Long Short-Term Memory (LSTM) networks, several pre-processing steps are essential to prepare the raw musical data for training the neural network effectively.

- Parsing MIDI Files
- Extracting Notes and Chords
- Mapping Notes to Integers
- Creating Sequences
- Reshaping and Normalizing Input Data
- One-Hot Encoding Output

Parsing MIDI Files

This reads MIDI files that contain musical information. MIDI files store data about notes, chords, and other musical elements in a digital format.

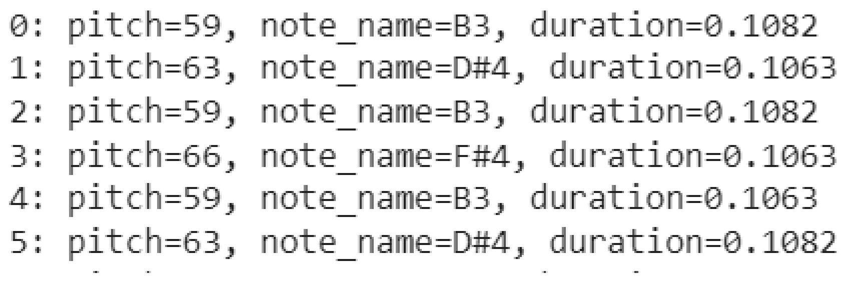


```
Extracting  0fithos.mid                                                 OK
Extracting  8.mid                                                       OK
Extracting  ahead_on_our_way_piano.mid                                  OK
Extracting  AT.mid                                                      OK
Extracting  balamb.mid                                                  OK
Extracting  bcm.mid                                                     OK
Extracting  BlueStone_LastDungeon.mid                                   OK
Extracting  braska.mid                                                  OK
Extracting  caitsith.mid                                                OK
Extracting  Cids.mid                                                    OK
Extracting  cosmo.mid                                                   OK
Extracting  costadsol.mid                                               OK
Extracting  dayafter.mid                                                OK
Extracting  decisive.mid                                                OK
Extracting  dontbeafraid.mid                                            OK
```

Fig.2.3 Extraction of MIDI files

Extracting Notes and Chords

The code distinguishes between individual notes and chords within the MIDI files. Notes and chords are fundamental musical elements that form the basis of the generated music.



```
0: pitch=59, note_name=B3, duration=0.1082
1: pitch=63, note_name=D#4, duration=0.1063
2: pitch=59, note_name=B3, duration=0.1082
3: pitch=66, note_name=F#4, duration=0.1063
4: pitch=59, note_name=B3, duration=0.1063
5: pitch=63, note_name=D#4, duration=0.1082
```

Fig.2.4 Notes and Chords

Mapping Notes to Integers

To process the textual representation of notes and chords, they are mapped to unique integers. This mapping is crucial as neural networks operate with numerical data.

One-Hot Encoding Output:

The output, representing the next note or chord in the sequence, is one-hot encoded. One-hot encoding converts categorical data into a binary matrix, making it compatible with the network's output layer.

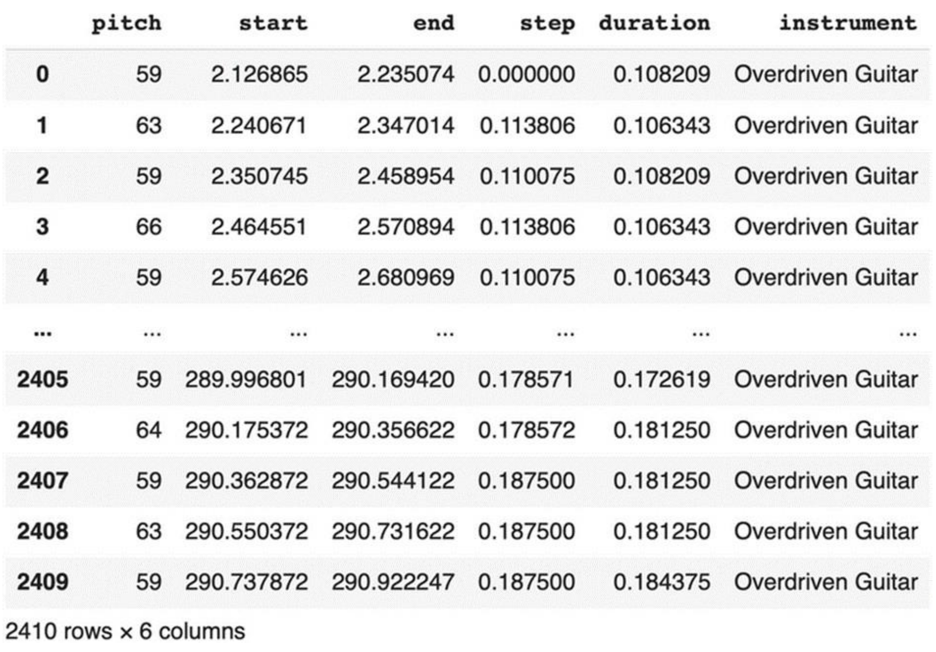
Reshaping and Normalizing Input Data

The input sequences are reshaped to fit the LSTM network's input layer requirements. Additionally, the input data is normalized to a range suitable for neural network training.

One-Hot Encoding Output

The output, representing the next note or chord in the sequence, is one-hot encoded. One-hot encoding converts categorical data into a binary matrix, making it compatible with the network's output layer.

2.3.2 DATASET



	pitch	start	end	step	duration	instrument
0	59	2.126865	2.235074	0.000000	0.108209	Overdriven Guitar
1	63	2.240671	2.347014	0.113806	0.106343	Overdriven Guitar
2	59	2.350745	2.458954	0.110075	0.108209	Overdriven Guitar
3	66	2.464551	2.570894	0.113806	0.106343	Overdriven Guitar
4	59	2.574626	2.680969	0.110075	0.106343	Overdriven Guitar
...	...	...	...	...	...	...
2405	59	289.996801	290.169420	0.178571	0.172619	Overdriven Guitar
2406	64	290.175372	290.356622	0.178572	0.181250	Overdriven Guitar
2407	59	290.362872	290.544122	0.187500	0.181250	Overdriven Guitar
2408	63	290.550372	290.731622	0.187500	0.181250	Overdriven Guitar
2409	59	290.737872	290.922247	0.187500	0.184375	Overdriven Guitar

2410 rows × 6 columns

Fig.2.5 Dataset

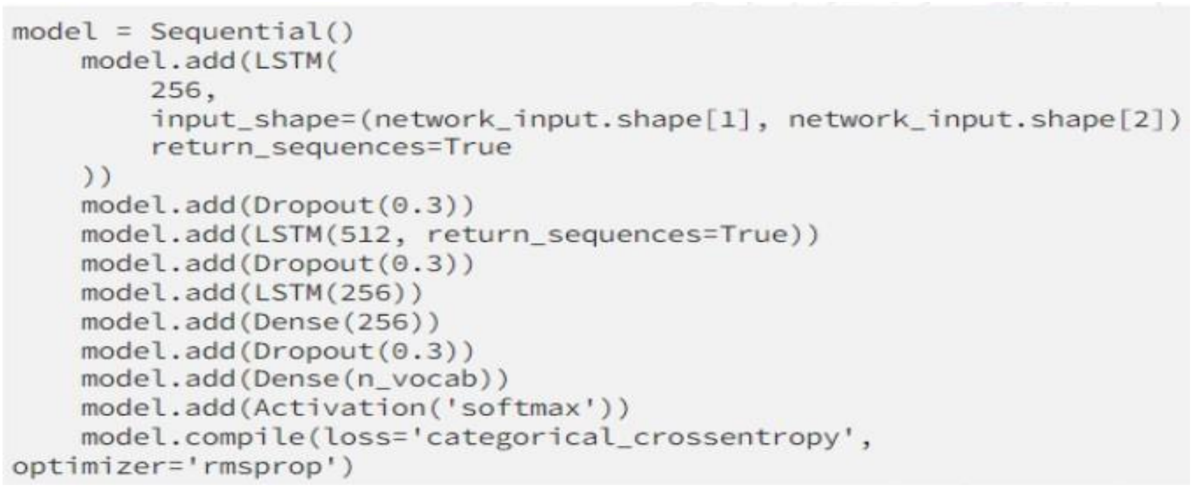
The above is the dataset that we got upon pre-processing the dataset with various techniques. Now this can we used further for getting more insightful information.

2.3.3 TRAINING OF NEURAL NETWORK

The purpose of the training process is to teach a Long Short-Term Memory (LSTM) neural network to generate music. This involves capturing patterns from a dataset of musical notes and chords and using these patterns to compose new, original music. The neural network is structured with three

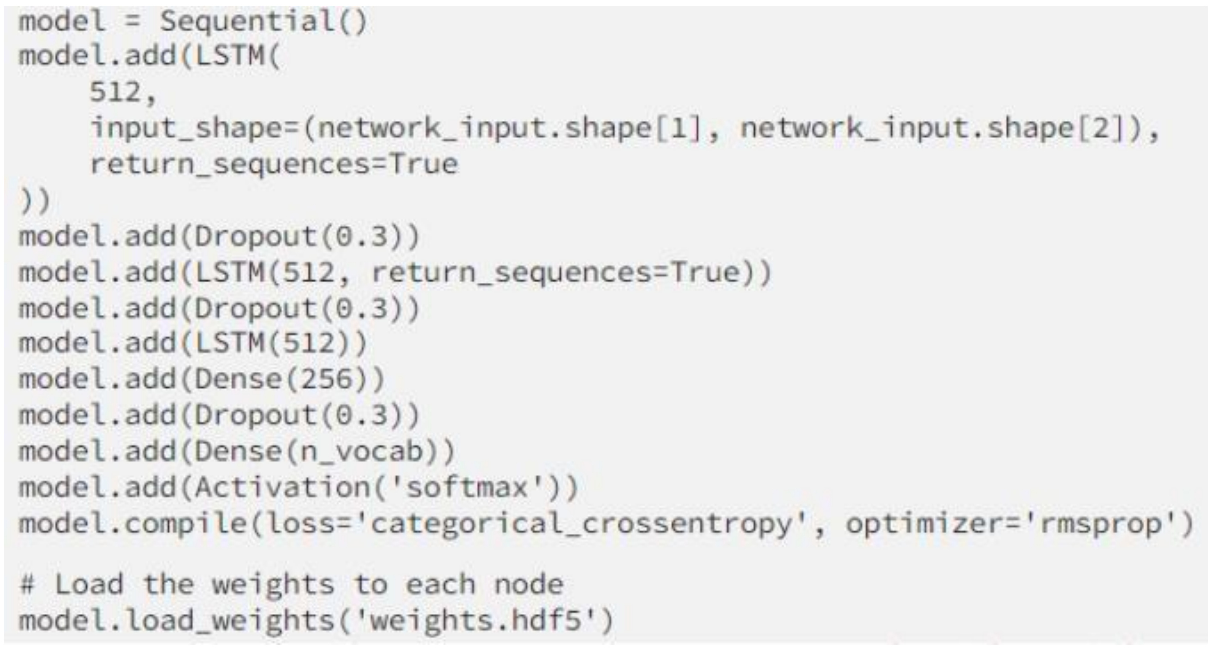
LSTM layers, each containing 512 units. Recurrent dropout with a rate of 0.3 is applied to introduce regularization and prevent over fitting. Batch normalization is used to stabilize and accelerate the training process. Two dense layers with 256 and n_vocab units (the number of unique pitch names) respectively, are employed, followed by ReLU activation functions for introducing non-linearity. Finally, a softmax activation function is applied to the output layer to generate probabilities for each pitch, determining the next musical element in the sequence. The following are the training process that are carries out.

- **Loss Function:** The categorical cross-entropy loss function is utilized, suitable for multi-class classification problems like this.
- **Optimizer:** RMSprop optimizer is used, which adapts the learning rates of each parameter individually, ensuring efficient and quicker convergence.
- **Metrics:** The accuracy metric is employed to measure the network's performance during training.
- **Epochs:** The network is trained over 20 epochs, indicating 20 complete passes through the entire training dataset.
- **Batch Size:** Training data is divided into batches of 128 sequences, enhancing computational efficiency



```
model = Sequential()
    model.add(LSTM(
        256,
        input_shape=(network_input.shape[1], network_input.shape[2])
        return_sequences=True
    ))
    model.add(Dropout(0.3))
    model.add(LSTM(512, return_sequences=True))
    model.add(Dropout(0.3))
    model.add(LSTM(256))
    model.add(Dense(256))
    model.add(Dropout(0.3))
    model.add(Dense(n_vocab))
    model.add(Activation('softmax'))
    model.compile(loss='categorical_crossentropy',
optimizer='rmsprop')
```

Fig.2.6 Architecture of neural network



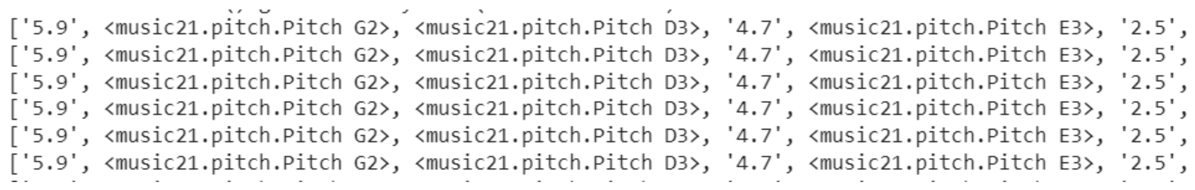
```
model = Sequential()
model.add(LSTM(
    512,
    input_shape=(network_input.shape[1], network_input.shape[2]),
    return_sequences=True
))
model.add(Dropout(0.3))
model.add(LSTM(512, return_sequences=True))
model.add(Dropout(0.3))
model.add(LSTM(512))
model.add(Dense(256))
model.add(Dropout(0.3))
model.add(Dense(n_vocab))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy', optimizer='rmsprop')

# Load the weights to each node
model.load_weights('weights.hdf5')
```

Fig.2.7 Architecture of neural network for music generation

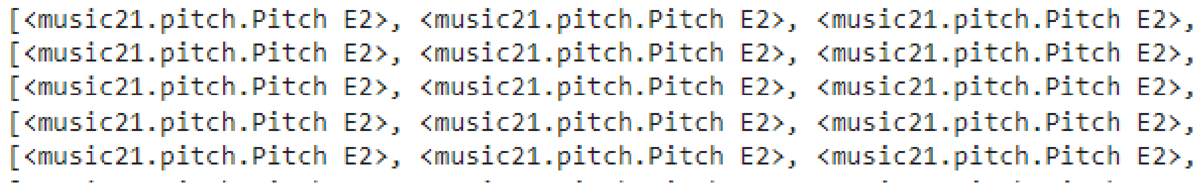
2.3.4 GENERATION OF MUSIC

We load the pre-existing musical notes used for training a neural network model. These notes are essential for the network to understand the patterns and structures of music. It then prepares sequences of musical notes, mapping them to integers for compatibility with the neural network. The network's architecture consists of Long Short-Term Memory (LSTM) layers designed to model musical sequences. After configuring the network, the script initiates the music generation process. It begins with a random seed sequence, and iteratively predicts and appends new musical notes based on the previously generated ones, ultimately forming a sequence of 500 notes. These generated notes are then transformed into a MIDI file, where chords and individual notes are represented. The offset between notes ensures they are not played simultaneously, creating a harmonious composition. The resulting MIDI file captures the neural network's creative musical output, stored as 'test_output3.mid,' ready for playback or further exploration.



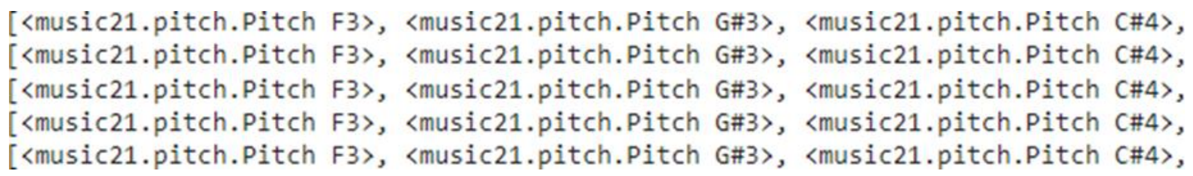
```
['5.9', <music21.pitch.Pitch G2>, <music21.pitch.Pitch D3>, '4.7', <music21.pitch.Pitch E3>, '2.5',
['5.9', <music21.pitch.Pitch G2>, <music21.pitch.Pitch D3>, '4.7', <music21.pitch.Pitch E3>, '2.5',
['5.9', <music21.pitch.Pitch G2>, <music21.pitch.Pitch D3>, '4.7', <music21.pitch.Pitch E3>, '2.5',
['5.9', <music21.pitch.Pitch G2>, <music21.pitch.Pitch D3>, '4.7', <music21.pitch.Pitch E3>, '2.5',
['5.9', <music21.pitch.Pitch G2>, <music21.pitch.Pitch D3>, '4.7', <music21.pitch.Pitch E3>, '2.5',
['5.9', <music21.pitch.Pitch G2>, <music21.pitch.Pitch D3>, '4.7', <music21.pitch.Pitch E3>, '2.5',
```

Fig.2.8 Test output 1 MIDI notes visualization



```
[<music21.pitch.Pitch E2>, <music21.pitch.Pitch E2>, <music21.pitch.Pitch E2>,
[<music21.pitch.Pitch E2>, <music21.pitch.Pitch E2>, <music21.pitch.Pitch E2>,
[<music21.pitch.Pitch E2>, <music21.pitch.Pitch E2>, <music21.pitch.Pitch E2>,
[<music21.pitch.Pitch E2>, <music21.pitch.Pitch E2>, <music21.pitch.Pitch E2>,
[<music21.pitch.Pitch E2>, <music21.pitch.Pitch E2>, <music21.pitch.Pitch E2>,
```

Fig.2.9 Test output 2 MIDI notes visualization



```
[<music21.pitch.Pitch F3>, <music21.pitch.Pitch G#3>, <music21.pitch.Pitch C#4>,
[<music21.pitch.Pitch F3>, <music21.pitch.Pitch G#3>, <music21.pitch.Pitch C#4>,
[<music21.pitch.Pitch F3>, <music21.pitch.Pitch G#3>, <music21.pitch.Pitch C#4>,
[<music21.pitch.Pitch F3>, <music21.pitch.Pitch G#3>, <music21.pitch.Pitch C#4>,
[<music21.pitch.Pitch F3>, <music21.pitch.Pitch G#3>, <music21.pitch.Pitch C#4>,
```

Fig.2.10 Test output 3 MIDI notes visualization

CHAPTER 3

CONCLUSION

3.1 CONCLUSION

In conclusion, this paper introduced the concept of using LSTM recurrent neural networks to generate musical expression, specifically focusing on dynamics and tempo in the context of Chopin's mazurkas. The study demonstrated that it is indeed possible to create computer-generated performances that closely resemble human interpretations. The generated dynamics and tempo values were compared to real performances, showing promising results in terms of accuracy and similarity.

The survey conducted to evaluate the computer-generated performances against human performances revealed that listeners found it challenging to distinguish between the two. This further supports the idea that the model is capable of capturing musical expression to a significant extent.

However, it's essential to acknowledge some limitations in this research. The training dataset was relatively small, and the model was primarily trained on Chopin's mazurkas. Expanding the model's capabilities to handle various musical styles and time signatures would require a more extensive and diverse dataset. Additionally, the use of score-based dynamics data in some cases may not accurately represent real performances, potentially affecting the model's performance.

In the future, further research could focus on improving the model's ability to handle a broader range of musical genres and incorporate more extensive and high-quality training datasets. This would enhance the model's capacity to capture the nuances of musical expression in different contexts.

3.2 FUTURE ENHANCEMENTS

- Using multiple models. Multiple models could be used to generate different parts of the music, such as the melody, harmony, and rhythm. This could lead to more complex and interesting music.

- Using feedback from humans. Feedback from humans could be used to train the model to generate music that is more pleasing to the ear.

- Use a dataset of music from different genres. This would allow the model to learn the different musical conventions of different genres. This would lead to the model being able to generate music in a wider variety of styles.

- Use a dataset of music that is more diverse in terms of instrumentation and tempo. This would allow the model to learn the different ways that notes and chords can be combined to create different musical effects. This would lead to the model being able to generate more interesting and varied music.

- Use a more sophisticated sampling technique. For example, the model could be trained to generate music that is more consistent with the style of the input sequence. This could be done by using a sampling technique that takes into account the probability of different notes and chords occurring in the input sequence.

- Use a model that can generate longer sequences of music. This would allow the model to generate more complex and interesting music.

REFERENCES

1. Morente-Molinera, J.A.; Kou, G.; Samuylov, K.; Ureña, R.; Herrera-Viedma, E. Carrying out consensual Group Decision Making processes under social networks using sentiment analysis over comparative expressions. Knowl. Based Syst. 2019, 165, 335–345. [Google Scholar] [CrossRef]
2. Abualigah, L.; Gandomi, A.H.; Elaziz, M.A.; Hussien, A.G.; Khasawneh, A.M.; Alshinwan, M.; Houssein, E.H. Nature-Inspired Optimization Algorithms for Text Document Clustering—A Comprehensive Analysis. Algorithms 2020, 13, 345.
3. Dennis Njagi, Z Zuping, Damien Hanyurwimfura, and Jun Long. 2015. A lexicon-based approach for hate speech detection. In International Journal of Multimedia and Ubiquitous Engineering.
4. Edel Greevy and Alan F. Smeaton. 2004. Classifying racist texts using a support vector machine. In SIGIR.
5. Spiros V. Georgakopoulos, Sotiris K. Tasoulis, Aristidis G. Vrahatis, and Vassilis P. Plagianakos. 2018. Convolutional neural networks for toxic comment classification. In SETN.
6. Michael Conner, Lucas Gral, Kevin Adams, David Hunger, Reagan Strelow and Alexander Neuwirth. 2022. Music Generation Using an LSTM.
7. Prashant Bhushan Shukla, Devesh Pratap Singh, Shaurya Deshwal, Kshitiz Tyagi and Chhaya Sharma. 2022. Implementation of Music Generation using LSTM Neural Network Training LSTM based RNN to generate automated music.
8. Shuqi Dai, Xichu Ma, Ye Wang and Roger B.Dannenberg. 2023. Personalised popular music generation using imitation and structure.

9. Jean-Pierre Briot, Gaëtan Hadjeres, Francois Pachet. 2019. Deep Learning Techniques for Music Generation.

10. Jian Pan,[1]Shaode Yu,[2]Zi Zhang,[1]Zhen Hu,[3]and Mingliang Wei. 2022.The Generation of Piano Music Using Deep Learning.

11. https://github.com/AI-Guru/music-generation-research

12. https://link.springer.com/article/10.1007/s00521-020-05399-0

13. https://arxiv.org/abs/1812.04186

14. https://ieeexplore.ieee.org/document/9960063

15. https://magenta.tensorflow.org/music-transformer

16. https://asmp-eurasipjournals.springeropen.com/

17. https://carlosholivan.github.io/DeepLearningMusicGeneration/

18. https://www.sciencedirect.com/science/article/pii/S095741742201353

19. https://www.researchgate.net/publication/365339518_Music_Generation_Using_LSTM_and_Its_Comparison_with_Traditional_Method

20. https://david-exiga.medium.com/music-generation

APPENDIX

PYTHON CODE:

```
import pickle
import numpy
from music21 import instrument, note, stream, chord
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.layers import BatchNormalization as BatchNorm
from keras.layers import Activation
def generate():
    with open('notes', 'rb') as filepath:
        notes = pickle.load(filepath)
    pitchnames = sorted(set(item for item in notes))
    n_vocab = len(set(notes))
    network_input, normalized_input = prepare_sequences(notes, pitchnames,
n_vocab)
    model = create_network(normalized_input, n_vocab)
    prediction_output = generate_notes(model, network_input, pitchnames, n_vocab)
    create_midi(prediction_output)
def prepare_sequences(notes, pitchnames, n_vocab)
    note_to_int = dict((note, number) for number, note in enumerate(pitchnames))
    sequence_length = 100
    network_input = []
    output = []
    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])
        output.append(note_to_int[sequence_out])
    n_patterns = len(network_input)
    normalized_input = numpy.reshape(network_input, (n_patterns,
sequence_length, 1))
    normalized_input = normalized_input / float(n_vocab)
    return (network_input, normalized_input
def create_network(network_input, n_vocab)
    model = Sequential()
    model.add(LSTM(
        512,
        input_shape=(network_input.shape[1], network_input.shape[2]),
        recurrent_dropout=0.3,
        return_sequences=True
    ))
    model.add(LSTM(512, return_sequences=True, recurrent_dropout=0.3,))
    model.add(LSTM(512))
    model.add(BatchNorm())
    model.add(Dropout(0.3))
    model.add(Dense(256))
    model.add(Activation('relu'))
    model.add(BatchNorm())
    model.add(Dropout(0.3))
    model.add(Dense(n_vocab))
    model.add(Activation('softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='rmsprop'
    model.load_weights('weights.hdf5')
    return model
def generate_notes(model, network_input, pitchnames, n_vocab)
```

```
    start = numpy.random.randint(0, len(network_input)-1)
    int_to_note = dict((number, note) for number, note in enumerate(pitchnames))
    pattern = network_input[start]
    prediction_output = []

    for note_index in range(500):
        prediction_input = numpy.reshape(pattern, (1, len(pattern), 1))
        prediction_input = prediction_input / float(n_vocab)
        prediction = model.predict(prediction_input, verbose=0)
        index = numpy.argmax(prediction)
        result = int_to_note[index]
        prediction_output.append(result)
        pattern.append(index)
        pattern = pattern[1:len(pattern)]
    return prediction_output
def create_midi(prediction_output):
    offset = 0
    output_notes = []

    for pattern in prediction_output:
        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)
        else:
            new_note = note.Note(pattern)
            new_note.offset = offset
            new_note.storedInstrument = instrument.Piano()
            output_notes.append(new_note)
        offset += 0.5
    midi_stream = stream.Stream(output_notes)

    midi_stream.write('midi', fp='test_output3.mid')

if __name__ == '__main__':
    generate()
```