Music Generation Using Deep Learning: A Bidirectional LSTM-Layer Based Recurrent Neural Networks Experiment

Emrehan Gokcay
Computer Engineering
Istanbul Kultur University
Istanbul,Turkey
emrehan.gokcayyz@gmail.com
emrehan_gokcay@hotmail.com

***Abstract*—This paper presents a project focused on developing a sequential model to generate new and unique music using Long Short-Term Memory (LSTM) networks. The results may lead to the future of AI in music sectors.**

***Index Terms*—Music generation, LSTM, Recurrent Neural Networks, MIDI, deep learning, bidirectional LSTM**

I. Introduction

The project presented in this report focuses on developing a sequential model to generate new and unique music based on a diverse range of musical styles and classifications. This project explores the application of Long Short-Term Memory (LSTM) networks, a type of Recurrent Neural Network (RNN), in the field of music generation. The model leverages MIDI datasets and operates within a Python environment using Google Colab. The system addresses both functional and non-functional requirements, ensuring versatile and high-performance music generation. Key libraries such as Music21, TensorFlow, Keras, and NumPy are employed. Comparative analysis demonstrates the superior performance of bidirectional LSTM models over unidirectional ones. The main objective is to build a model capable of creating music, leveraging the capabilities of LSTM networks to capture temporal dependencies in music sequences.

A. Project Purpose, Scope, and Objectives

The purpose of the project is to explore music generation using a LSTM layer based model which is a type of Recurrent Neural Networks (RNNs) and build a model capable of creating music. It aims to study the application of RNNs in music generation and provide insights into the creative potential of AI in the field of music.

B. Technical Assumptions and Constraints

The project assumes access to comprehensive MIDI datasets containing diverse musical compositions. It also requires the availability of computational resources capable of training deep learning models, specifically within the context of Python programming and Google Colab. Key technical assumptions include the use of Python libraries such as Music21, TensorFlow, Keras, and NumPy. The constraints involve computational limitations, memory limitations, and the capabilities of the libraries and frameworks employed.

C. Naming Conventions

Standard practices in Python programming are followed for naming conventions, ensuring transparency and consistency in variable names, function names, and other identifiers throughout the codebase. This approach facilitates readability and maintainability of the code.

II. Requirements

A. Functional Requirements

1) Music Generation Requirements:

- Develop a sequential model capable of generating music sequences based on input data.
- The system must train a deep learning model, such as an LSTM neural network, on the preprocessed musical sequences.
- Evaluate the performance of the model in terms of its ability to produce stable levels of training and validation losses and produce coherent, aesthetically pleasing music.
- Train the model using different datasets folders to ensure versatility in music generation.

B. Non Functional Requirements

1) Performance Requirements:

- The model should quickly parse the midi files and respond to generation function.
- To avoid performance limitors such as getting stuck at parsing a single file for a very long estimated time it should skip the file in order to not to reduce performance.

2) Software Quality Attributes:

a) Maintainability:

- The codes should be well-organized and well-documented to establish maintenance and future improvements.
- Changes to the system should be easy to implement without affecting its overall functionality.

b) Reliability:

- The system should produce consistent and reliable results across with different set runs.
- It should handle and display informative error messages when issues arise other than compiler errors.

III. Other Requirements

1) Configuration Requirements:

- The system should allow for easy adjustment of parameters such as maximum processing time and maximum number of MIDI files to process.
- Parameters should be clearly defined and easily modifiable within the code for comparison purposes later.

2) Persistence Requirements:

- The system should support saving and loading of intermediate data and model checkpoints to facilitate experimentation and continuation of work.

3) Compatibility Requirements:

- The system should be compatible with standard Python environments commonly used for cross platform using purposes.
- It should run on commonly available operating systems without additional needs or setup requirements.

4) Documentation Requirements:

- Clear and understandable documentation should be provided to aid understanding of the code and its functionality.
- Documentation should include explanations of key concepts, interpreting the results, performance graphs

IV. System Architecture

A. Architectural Design of the Model

1) Key libraries/structures explained:

- Music21 [11]: Music21 is a well known Python library known for its useful tools in generating, analyzing, and manipulating audio files such as songs and melodies. **In our project, we utilized this library to convert MIDI files into note sequences, classify these notes, and then convert the generated sequences back into MIDI format.**
- Tensorflow and Keras: TensorFlow and Keras are leading libraries in the field of deep learning. **In our project, we utilize these libraries to perform critical tasks such as creating, training, and evaluating deep neural networks (such as LSTM and RNN). TensorFlow provides a robust infrastructure for building and distributing computational graphs, while Keras is easy-to-use for rapidly developing deep learning models. This enables us to develop complex artificial intelligence projects more quickly and effectively.**
- Numpy: NumPy is an indispensable library in Python for scientific computing and data processing. This library enables efficient processing of multi-dimensional arrays and fast execution of mathematical operations. Additionally, it is widely used in various fields such as data analysis, artificial intelligence, and deep learning.**NumPy plays a key role in data processing and model training processes in our project. Especially, multi-dimensional arrays and mathematical operations are used to process datasets and prepare inputs and outputs for the model while parsing it. Additionally, NumPy functions are utilized for data manipulation and preprocessing steps during model training. Therefore, the NumPy library is one of the key components of the project, resulting efficient management of data processing flow.**
- Glob: Glob is a versatile library in Python that simplifies file path handling and enables efficient file search operations. **In our project, we use Glob to locate and parse MIDI files, lining the process of accessing and processing our dataset.**
- Pyplot and Matplotlib: Pyplot and Matplotlib are powerful libraries for creating visualizations and plots in Python. These libraries offer a wide range of functions and features for generating high-quality charts, graphs, and visual representations of data. **In our project, we utilize Pyplot and Matplotlib to visualize various aspects of our data, enabling us to gain insights and make informed decisions during the analysis and model development phases.**
- Time: Time is a fundamental library in Python that provides functions for measuring and managing time-related operations. **In our project, we use time library to track the execution time of different processes and operations. By monitoring the time taken for various tasks, we can optimize our workflows and improve the efficiency of our project.**
- LSTM [9](Long Short-Term Memory): LSTM (Long Short-Term Memory) networks are a type of recurrent neural network (RNN) architecture that excels in processing sequential data with dependencies. In this project, we take advantage of LSTM networks for their ability to effectively capture patterns and dependencies in music sequences. **We utilize LSTM networks to train our model on a set of MIDI files, enabling it to learn the intricate relationships between musical notes and structures. By employing LSTM networks, we can generate music that has patterns and structures. Figure 1 Shows an inside view of a LSTM layer cell.**

- Bidirectionality [7]: Bidirectionality in RNN, or Bidirectional Recurrent Neural Networks, is an important concept that makes these neural networks smarter. Instead of just looking at the input data from the past like traditional RNNs, bidirectional RNNs also consider future information when making predictions as visualized in **Figure 2**. This helps them understand context better and make more accurate predictions. **In our project, we use bidirectional RNNs to improve our model's performance and make better predictions on sequential data.**

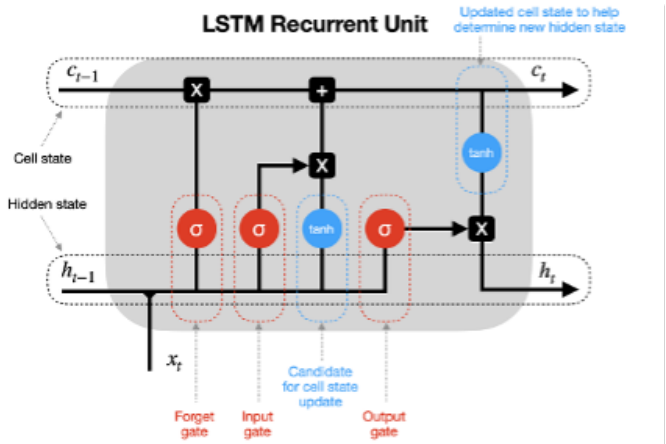


Fig. 1. Inside of a LSTM layer cell

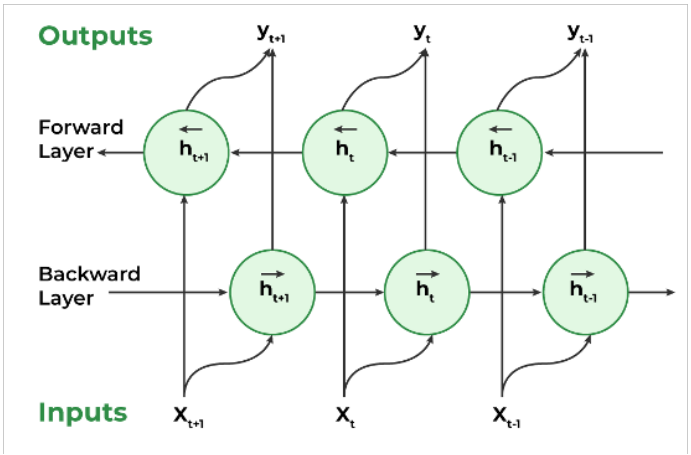


Fig. 2. Bidirectionality visualized.

B. Experiment Sets Explained

We used LAKH MIDI Dataset [1] . (which contains 340.680 midi music files, but in order to avoid overfitting we're taking samples of 100 while training the model) and MAESTRO v.2.0.0 Dataset [2](Which contains 1282 midi music files but in order to experiment evenly we're taking varying samples evenly in experiments while training this model also)

C. Explaining the model

The architecture of the model as shown in **Figure 3**, provides a LSTM(Bidirectional) based, powerful and flexible structure for music production [3]. It can be applied across different genres and styles of music for gaining different genres of music. Additionally, the modular nature of the model allows for easy customization and extension for specific musical types. Here is the explanation of each layer with descriptions of their working logic:

1) **Input Layer**:
   - Input Dimensions `network_input.shape[1] x network_input.shape[2]` (e.g., 100 time steps, 1 feature)
   - **Description**: This layer allows the model to accept time series data of notes or chords from a processed MIDI file.
2) **Bidirectional LSTM Layer (First Layer)**:
   - **Number of Neurons**: `64`
   - **return_sequences**: True
   - **Description**: The first bidirectional LSTM layer considers both past and future data points to provide a better understanding of the context. This layer helps the model to learn long-term dependencies in the time series data.
3) **Batch Normalization Layer (First Layer)**:
   - **Description**: This layer normalizes the activations for each mini-batch, which helps to speed up and stabilize the learning process.
4) **LeakyReLU Activation Layer (First Layer)**:
   - **Alpha**: `0.2`
   - **Description**: This layer addresses the drawbacks of ReLU by providing a small slope for negative values, helping the model to generalize better.
5) **Dropout Layer (First Layer)**:
   - **Dropout Rate**: `0.7`
   - **Description**: This layer prevents overfitting by randomly setting the activations of some neurons to zero during training.
6) **Bidirectional LSTM Layer (Second Layer)**:
   - **Number of Neurons**: `128`
   - **return_sequences**: True
   - **Description**: The second bidirectional LSTM layer allows the model to perceive more complex features and processes the information from the previous layer in more detail.
7) **Batch Normalization Layer (Second Layer)**:
   - **Description**: Normalizes the activations, which speeds up and stabilizes the training process.
8) **LeakyReLU Activation Layer (Second Layer)**:
   - **Alpha**: `0.2`
   - **Description**: Provides a small slope for negative values, addressing the drawbacks of ReLU.
9) **Dropout Layer (Second Layer)**:
   - **Dropout Rate**: `0.7`
   - **Description**: Enhances the model's generalization capability and reduces overfitting.
10) **Bidirectional LSTM Layer (Third Layer)**:
   - **Number of Neurons**: `64`
   - **return_sequences**: False
   - **Description**: The third bidirectional LSTM layer compresses the information learned to help the model interpret the results more effectively.
11) **Batch Normalization Layer (Third Layer)**:
   - **Description**: Normalizes the activations, which speeds up and stabilizes the training process.
12) **LeakyReLU Activation Layer (Third Layer)**:

- **Alpha**: `0.2`
- **Description**: Provides a small slope for negative values, addressing the drawbacks of ReLU.

13) **Dropout Layer (Third Layer)**:
- **Dropout Rate**: `0.7`
- **Description**: Reduces the model's tendency to overfit.

14) **Dense Layer**:
- **Number of Neurons**: `128`
- **kernel_regularizer**: `l2(0.004)`
- **Description**: This dense layer performs a linear transformation of the features learned by previous layers. It retains the raw information from the preceding layers, allowing the model to capture complex relationships within the data.

15) **Batch Normalization Layer (Dense Layer)**:
- **Description**: Normalizes the activations, which speeds up and stabilizes the training process.

16) **LeakyReLU Activation Layer (Dense Layer)**:
- **Alpha**: `0.2`
- **Description**: Provides a small slope for negative values, addressing the drawbacks of ReLU.

17) **Dropout Layer (Dense Layer)**:
- **Dropout Rate**: `0.7`
- **Description**: Prevents overfitting by randomly setting the activations of some neurons to zero during training.

18) **Dense Layer**:
- **Number of Neurons**: `n_vocab` (represents the number of unique notes or chords the model can predict)
- **Description**: This layer transforms the outputs of the model to all possible notes or chords.

19) **Activation Layer (Softmax)**:
- **Activation Function**: 'softmax'
- **Description**: The softmax activation layer converts the outputs into a probability distribution, enabling the model to predict the next note or chord.

20) **Optimizer**:
- **Type**: Adam
- **Learning Rate**: `0.000001`
- **Beta_1**: `0.9`
- **Description**: Adam optimizer adapts the learning rate and momentum terms to ensure the model learns quickly and effectively.

V. Optimization, Results & Comparing With Other Models

A. Steps for Optimization

In order to achieve a model that produces the most similar output to human-grade natural music, we made significant differences to our model. **Figures 4.a, 4.b and Figure 5** are the variations of our model with the versions of significant changes added. (layers with numbers and underscores mean different version in a same colab run.)

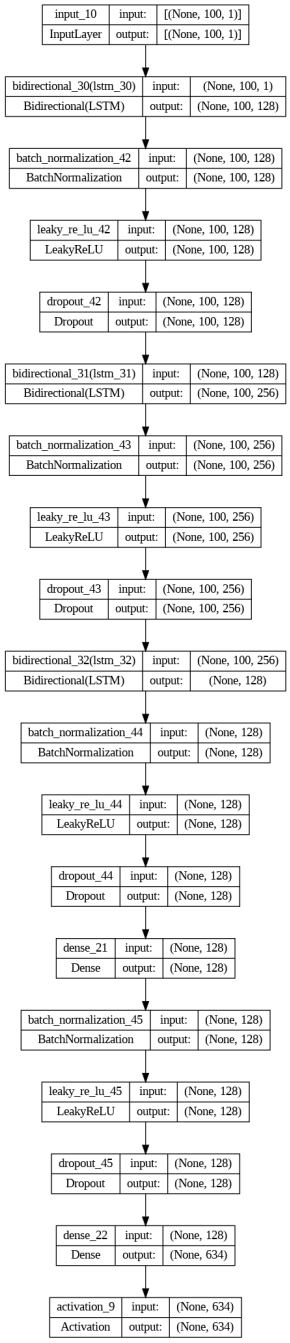
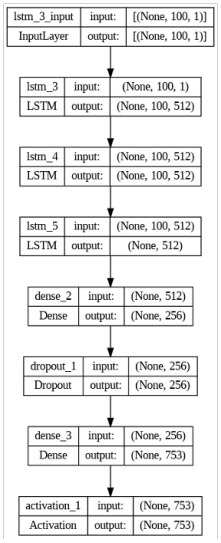


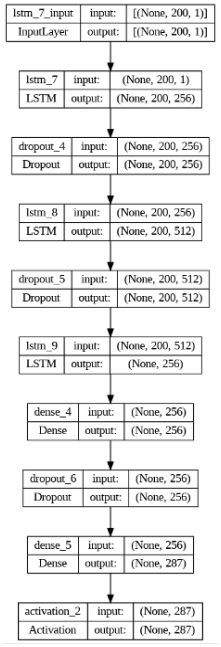
Fig. 3. The architecture of the model in its final state.

B. Results & Comparing Our Model

1) ***Comparing Performance Between Unidirectional LSTM and Bidirectional LSTM Layer Based Model Versions****:* For this part we're testing LAKH MIDI Dataset and test it through same number of epochs below for expect to show the difference of loss performance between unidirectional LSTM layer based model and bidirectional LSTM based model.



(a) Our model's first step,
repetitive music around same note.
Needs to be changed.



(b) We lowered input neurons
of this model in order to reduce complexity and added some
dropout layers in order to reduce overfitting.
Therefore planned to reducing loss and creating less repetitive music.

Fig. 4. First variations of our model.

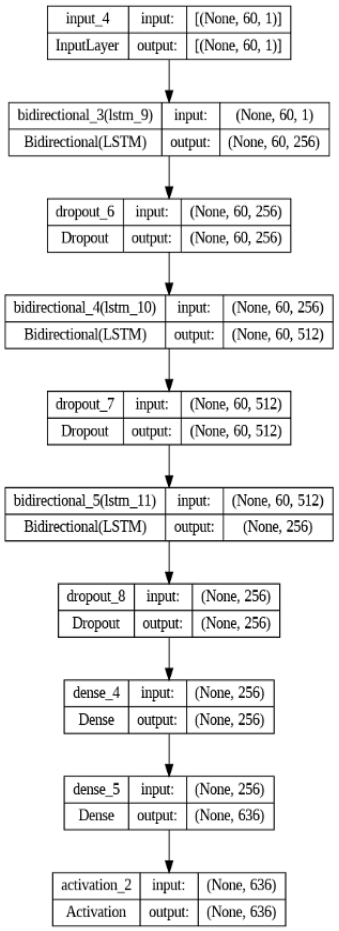


Fig. 5. This is the most similar structure that we have developed until now. Experiments [3], [4] have shown that using Bidirectional LSTM layers instead of regular LSTM layers(unidirectional) lets us achieve to create more human-like music.

Parameters, that are kept constant in this experiment are listed in **Table 1**:

TABLE I
TRAINING PARAMETERS

Parameter	Value
Epochs	200
Batch size	64
Sequence length	100
Optimizer Learning Rate	0.000001
Optimizer	Adam
Music Samples	100
Trained GPU	NVIDIA A100(40GB)

The following analysis compares the performance of two LSTM-based models: one utilizing unidirectional LSTM layers and the other utilizing bidirectional LSTM layers. These models are evaluated based on their training and validation loss over 200 epochs, as shown in Figures 6 and 7.

*a) **Unidirectional LSTM Model**:*

• **Initial Loss:** The model's training loss starts at 5.7806,

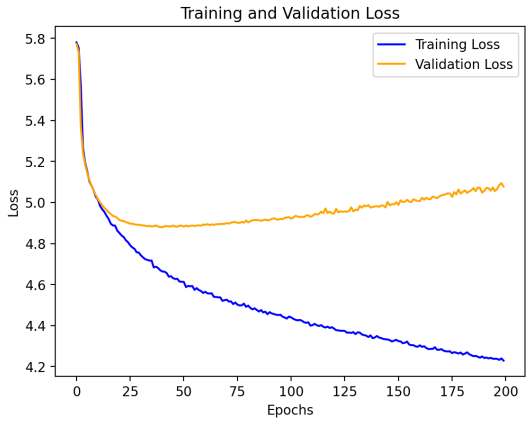


Fig. 6. Training and Validation Loss for Unidirectional LSTM Model

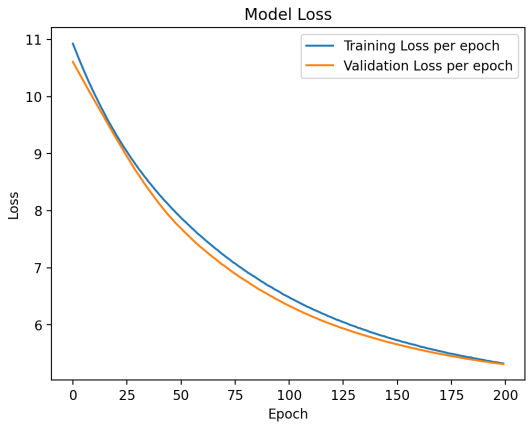


Fig. 7. Training and Validation Loss for Bidirectional LSTM Model

slightly near at `5.8` , as shown in Figure 6.

- **Final Loss:** The training loss finishes at `4.2298`.
- **Validation Loss Dynamics:**
  - The validation loss starts at a similar value as the training loss, `5.7736`, slightly near `5.8` .
  - Initially, both training and validation losses decrease.
  - Around epoch 25, the validation loss starts to diverge from the training loss, increasing gradually after reaching a minimum.
  - By the end of training, the validation loss is at `5.0777`, which is above training loss, indicating overfitting.

*b) **Bidirectional LSTM Model**:*

- **Initial Loss:** The model's training loss starts at `10.9257`, around `11`, as visualized in Figure 7.
- **Final Loss:** The training loss finishes at `5.3216`.
- **Validation Loss Dynamics:**
  - The validation loss starts at `10.6082`, slightly lower than the training loss.
  - Both training and validation losses consistently decrease throughout the training process.
  - By the end of training, the validation loss comes close to the training loss, `5.3090` to `5.3216` respectively, indicating better generalization.

2) Comparative Analysis:

-Performance: As seen in Figure 7, the bidirectional LSTM model shows higher loss at both initial and final stages, but it produces a stabilized curve rather than bidirectional LSTM which validation loss curve goes off around 25'th epochs, indicating heavy overfitting.

-Adaptation Speed: The bidirectional model adapts faster just as in experiments conducted by T. Jiang et. al. [4], with both training and validation losses decreasing steadily without divergence.

-Overfitting: Figure 6 shows that the unidirectional model has signs of overfitting as indicated by the increasing validation loss after the initial epochs, whereas the bidirectional model maintains a consistent validation loss, suggesting better generalization.

3) Comparing Our Music Output Side-by-Side to Created Real Person Music: For this part we're using the help from a bunch of web-based AI application such as called Cyanite.ai [5], which analyzes your uploaded music in various ways and picks a bunch of music that has similarities to your music.

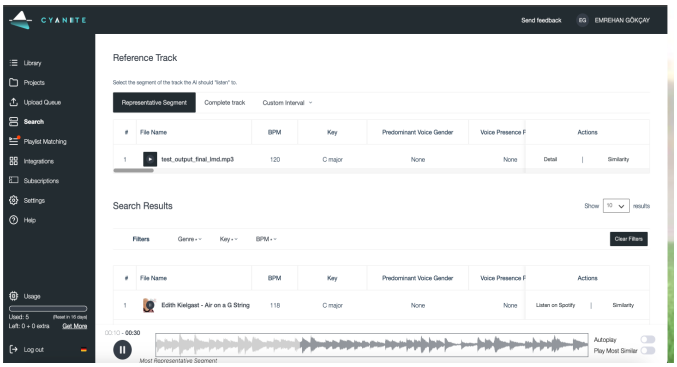


Fig. 8. Inspecting our final music trained with LMD dataset

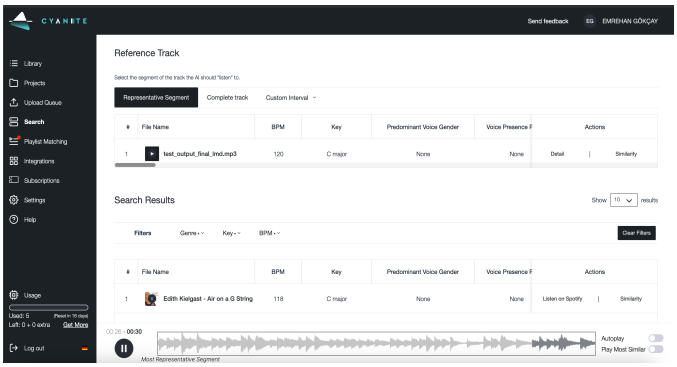


Fig. 9. Inspecting our final music trained with MAESTRO dataset

The AI analyzed our music that is created by both Lakh MIDI dataset at (**Figure 8**) and MAESTRO dataset (**Figure 9**),

we found out that in our music created by LMD's sequences between 2:45-3:15 is the most similar to, **Edith Kiegelgast — Air on a G String**'s sequences between 0:00-0:30 .
And the music that is created by MAESTRO's sequences between 3:15-3:45 is the most similar to, **Erik Satie & Marcel Worms — Sports et divertissements: No. 1, Choral inappétissant**'s sequences between 0:00-0:30.

C. Comparing the Performance of Our Model's Latest Version Across Different Datasets

For this part we're fixing the parameter values below while testing LAKH MIDI Dataset and MAESTRO v.2.0.0 dataset while testing it through same number of epochs below for expect to show the difference of loss performance between our model trained with LAKH MIDI dataset and MAESTRO dataset.

TABLE II
TRAINING PARAMETERS FOR DATASET COMPARISON

Parameter	Value
Epochs	200
Batch size	64
Sequence length	100
Optimizer Learning Rate	0.000001
Optimizer	Adam
Music Samples	100
Trained GPU	NVIDIA A100(40GB)

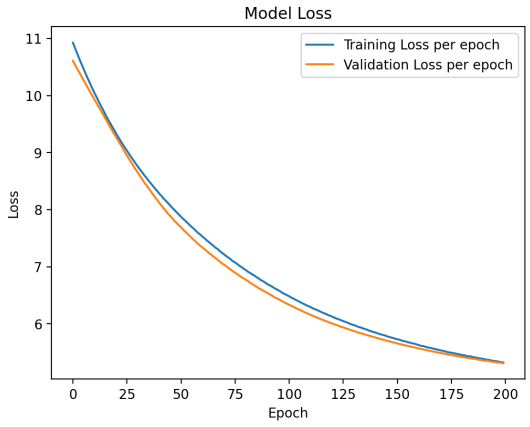


Fig. 10. This is the graph of network's losses after using to train **Lakh Midi Dataset**

As visualized in **Figure 10** and **Figure 11**, our model has a better performance in terms of training and validation losses while training with the **LAKH MIDI Dataset**. Due to MAESTRO dataset's music files are containing longer music compared to the LMD dataset, this resulted in higher performance losses in MAESTRO dataset compared to the LMD.

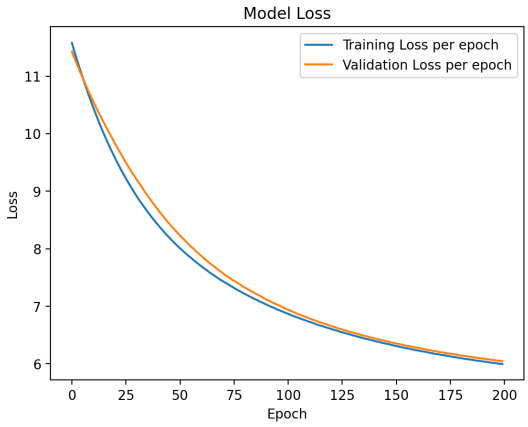


Fig. 11. This is the graph of network's losses after using to train **MAESTRO V2.0.0 Dataset**

D. What We've Done In Order to Eliminate Overfitting

In our efforts to eliminate overfitting in the model, we are implemented several strategies and techniques to ensure the model generalizes well to unseen data. These strategies include:

1) **Increasing Dropout Rates**:
   - **Description**: We increased the dropout rates to `0.7` in multiple layers of the model. Dropout is a regularization technique where randomly selected neurons are ignored during training. This prevents the network from becoming too reliant on specific neurons, thus reducing overfitting.
2) **L2 Regularization** [12]:
   - **Description**: L2 regularization (also known as weight decay) was applied to the LSTM and Dense layers with a regularization factor of `0.004`. This technique adds a penalty equal to the sum of the squared weights to the loss function, discouraging large weights and thus reducing overfitting.
3) **Batch Normalization** [12]:
   - **Description**: Batch normalization was added after each LSTM and Dense layer. This technique normalizes the activations of the previous layer for each mini-batch, which helps to stabilize and speed up the training process, and can also reduce overfitting.
4) **Leaky ReLU Activation** [8]:
   - **Description**: Instead of the standard ReLU activation, we used Leaky ReLU with an alpha value of `0.2`. Leaky ReLU allows for small non-zero outputs for negative input values, which helps retain important information that might otherwise be ignored. This can make it more effective than ReLU, especially when the data contains a lot of noise or unusual values.
5) **Bidirectional LSTM Layers**:
   - **Description**: We used bidirectional LSTM layers to capture dependencies from both past and future

time steps. This provides a more comprehensive understanding of the sequence, which can improve the model's ability to generalize.

6) **Reduced Learning Rate**:
   - **Description**: The learning rate was set to a very very low value of `0.000001`. A lower learning rate helped our the model to converge more slowly but more reliably, leading to a better generalization.
7) **Modified Adam Optimizer**:
   - **Description**: We adjusted the `beta_1` parameter of the Adam optimizer to `0.9`. This modification helps to improved generalization.
8) **Validation Set**:
   - **Description**: The data was split into training and validation sets. The validation set is used to monitor the model's performance on unseen data during training, which helps in tuning the model and identifying overfitting.
9) **Learning Rate Reduction on Plateau** [12]:
   - **Description**: We used **ReduceLROnPlateau**, which reduces the learning rate by a factor of `0.2` if the validation loss does not improve for `2` back to back epochs. This allows the model adapt quickly to the overfitting situations by changing the learning rate dynamics.
10) **Early Stopping**:
   - **Description**: We used early stopping with a patience of `10` epochs. This technique monitors the validation loss, and if it does not improve for `10` back to back epochs, training is stopped. This helps to fight with overfitting by stopping the training process after model shows signs to overfit.

VI. CONCLUSION

In this study, we wanted to see how Long Short-Term Memory (LSTM) networks can create new music. We found that bidirectional LSTM models are very good at understanding the patterns and creating better music. They work better than the regular LSTM models. We used two datasets, the LAKH MIDI Dataset and the MAESTRO Dataset, to train our models. Our bidirectional LSTM model showed that it can learn and perform well with both datasets. It is a strong and flexible model for making music. We used popular tools like TensorFlow, Keras, and Music21. These tools helped us build a system that can make music that sounds harmonical to average human ears. This is an important step for using deep learning in creative tasks like music generation. Our work shows that neural networks can understand and can create sequences just like in real human music. In summary, this study helps us understand how to use deep learning for generating music. As we move forward, combining AI and human creativity will bring new possibilities. By learning from this project, we can develop even better models and applications and if we can push the boundaries of limitations, we can generate human-like music that is difficult to diversify by normal ears. This can be a leading research about its usage in later researches.

REFERENCES

[1] https://colinraffel.com/projects/lmd/
[2] https://www.magenta.tensorflow.org/datasets/groove
[3] A. Ranjan, V.N. J. Behera, M. Reza, "Using a Bi-directional LSTM Model with Attention Mechanism trained on MIDI Data for Generating Unique Music", Available on : https://arxiv.org/pdf/2011.00773
[4] T. Jiang, X. Yin, Q. Xiao, "Music Generation Using Bidirectional Recurrent Network", Available on: https://ieeexplore.ieee.org/document/8839399
[5] https://app.cyanite.ai
[6] https://cifkao.github.io/html-midi-player/
[7] https://www.geeksforgeeks.org/bidirectional-recurrent-neural-network/
[8] https://medium.com/@sreeku.ralla/activation-functions-relu-vs-leaky-relu-b8272dc0b1be
[9] https://www.analyticsvidhya.com/blog/2021/03/introduction-to-long-short-term-memory-lstm/
[10] https://medium.com/@souro400.nath/why-is-bilstm-better-than-lstm-a7eb0090c1e4
[11] https://web.mit.edu/music21/doc/index.html
[12] https://keras.io/api/
[13] https://towardsdatascience.com/how-to-generate-music-using-a-lstm-neural-network-in-keras-68786834d4c5