Computer Science Extended Essay

Topic:

Music Generation Using Neural Networks

Research Question:

To What Extent can LSTMs Replicate the Compositional Style of Specific Composers?

Word count:

3903 Words

IBDP Session:

May 2024

Contents

List of Figures

List of Tables

1 Introduction

In recent times, machine learning has taken over the world. Models such as large language models (LLMs) and generative imagery are able to achieve extraordinary results, all while being accessible to the public. Machine learning is a field of computer science, that addresses the question of how to build computers that improve automatically through experience (**jordan2015machine**). Recent development of advanced learning algorithms, availability of large volumes of training data from the internet and the exponential rise in computational power has facilitated this progress (**jordan2015machine**). Music is difficult to define, it is not merely the vibration of sound waves, rather it contains also emotional depth, storytelling, and expression of complex emotions, which might be difficult for models to replicate. It is ancient, pan cultural, and, given the spontaneous emersion of song in children, universal (**davies2012defining**). Exploring stylistic music generation can help gain insight into patterns and compositional style of composers.

This essay aims to explore replicating the compositional style of musicians, specifically that of JS Bach. Bach (1685–1750) was a German composer of the Baroque period, known for his distinctive musical style. His influence on western music means that his works are readily available online and in the public domain.

It is important to note that research like this also poses various legal and ethical question in regard to utilisation of other musician's work for training purposes. While being inspired from other's work is a part of the creative process, directly copying the work is a violation of copyright law and the question regarding AI in this matter is vague (**rickardgenerating**).

2 Background Information

2.1 Introduction to Neural Networks

‘ In its most basic form, the artificial neural network (ANN) is a computational and mathematical model that emulates the brain’s structure and functioning. The human brain consists of 86 billion interconnected nerve-cells, or *neurons* (**herculano2009humanbrain**). Each neuron receives input from thousands of other neurons in the form of electrical signals and outputs a signal if it passes a certain threshold.

Similarly, in the ANN the neuron/node is the basic unit of calculation, where each node is connected to other nodes with a weight value. A row of neurons forms a layer. A single neuron receives the weighted output of other interconnected neurons and adds a bias value and activation function to the value, which becomes its output. The bias value is another parameter which allows for linear transformation within the data (**BiasValuePico**), and the activation function is a function that introduces non-linearity within the data and bounds the data within a fixed range (**Kiraleos2023MLAstronomy**). Some activation functions are:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

$$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$$

A schematic of a network can be seen in Figure 1 and 2 shows the structure of the node. Mathematically, (**agarawalmath**)

$$z_{j+1} = W_j a_j + b$$

$$a_{j+1} = \varphi(z_{j+1})$$

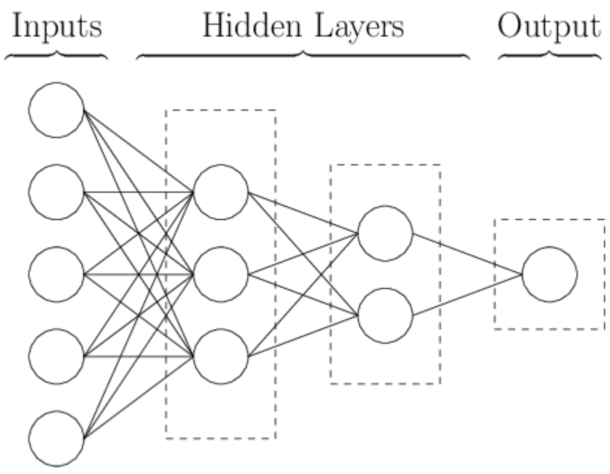

Figure 1: Schematic of a basic neural network (**cinelli2018vessel**)

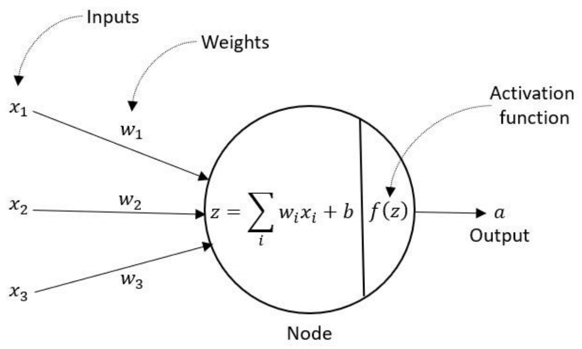

Figure 2: Schematic of a single neuron (**agarawalmath**)

Where,

- W_j = weight of position 'j'
- a_j = weight of position 'j'
- b = bias value
- φ = activation function

Initially, the output of the network will be erroneous, due to the weights and biases being randomised. To quantify the error, a cost function is used. One such cost function is the Sparse Categorical Cross entropy function, which is primarily used for classification related problems. (**rahmanmath**). Mathematically,

$$C = -\sum_{i=1}^{N} y_i \cdot \log(\hat{y}_i)$$

Where,

- y_i = Correct probability class for i th iteration
- $\hat{y}_i$ = Predicted distribution class of i th iteration
- N = Number of iterations

The loss function is used, in accordance with the softmax activation function, which normalises the outputs of the network into a probability distribution (**rahmanmath**). This loss function is useful in dataset with error in the labelling (**rahmanmath**) and is computationally efficient compared to other loss functions like the Mean-Squared.

Finally, the cost function is minimised using the gradient descent algorithm and the weights and biases parameters are adjusted accordingly. The gradient descent at a point takes interactive steps in the opposite direction of the gradient of the cost function (**amari1993backpropagation**). This is mathematically defined as

$$W_j^{k+1} = W_j^k - \eta \frac{\partial C}{\partial W}$$
$$b_j^{k+1} = b_j^k - \eta \frac{\partial C}{\partial b}$$

Where,

- j = position of a weight
- k = iteration
- W_{jk+1} = Weights in the next position
- W_{jk} = Weights in the current position
- b_{jk+1} = Bias in the next position
- b_{jk} = Bias in the current position

- $\frac{\partial C}{\partial W}$ = Partial derivative of the Cost function wrt. Weight
- $\frac{\partial C}{\partial b}$ = Partial derivative of the Cost function wrt. Bias
- η = Learning Rate

Here, the learning rate 'η' is a scaling factor that is manually set to speed up gradient descent. If it is too low, then the network will need more time and calculations to find a minimum and if it is too large, the network may skip past the minima (**smith2017cyclical**). While the optimum result would be finding the global minimum, often the network may be stuck at a local minimum, hence the learning rate parameter is useful to prevent that bottleneck.

Ultimately, this is a basic and an old model of a neural network and is unoptimised to predict music. Hence, a Recurrent Neural Network is used instead.

2.2 Recurrent Neural Networks (RNNs)

2.2.1 Introduction to Recurrent Neural Networks

Recurrent Neural Network (RNN) is a type of neural network that uses sequential or time series data. In other words, RNNs have the ability to utilise information from previous calculations to generate new information (**pascanu2014construct**). To better understand how RNNs work, we can take the following sentence

"Eww, This food is ____"

Here, we can easily fill in the blank. The word 'eww', implies a negative connotation and the 'food' suggests that the word should be related to food. In the same manner, any given note in a piece of music is dependent on previous notes. Random groupings of notes sound dissonant, and almost all the music contains within itself some mathematical pattern as defined in music theory. Hence, due to this relation, it can be hypothesised that RNNs excel in music generation. Figure 3 below, shows a schematic of an RNN.

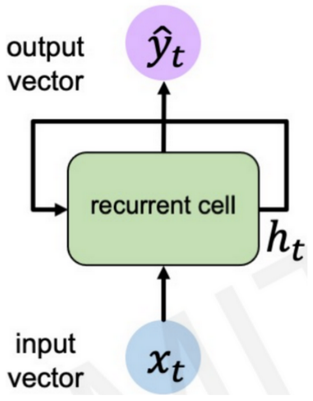

Figure 3: A schematic of a RNN (**mitamini**)

2.2.2 Structure of Recurrent Neural Networks

The RNN contains additional information called the hidden state or the memory, which feeds back into the network (**pascanu2014construct**). Mathematically, the hidden state is defined as (**mitamini**)

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

Where,

- h_t = current hidden state
- h_{t-1} = previous hidden state
- x_t = input value
- W_{hh} = weights at previous hidden state
- W_{xh} = weights at current input state

Finally, the output denoted by y is calculated by (**mitamini**)

$$y_t = W_{hy} \cdot h_t$$

Where,

- W_{hy} = Weights at the output state

2.2.3 Backpropagation in Recurrent Neural Networks

Backpropagation in RNNs is done using the backpropagation through time algorithm. At each timestep t_n , the loss value L_n is added up to create the overall loss L. The overall loss value is then backpropagated in reverse through time, until the initial timestep t_0. This can be visualised in the figure 4 (**mitamini**).

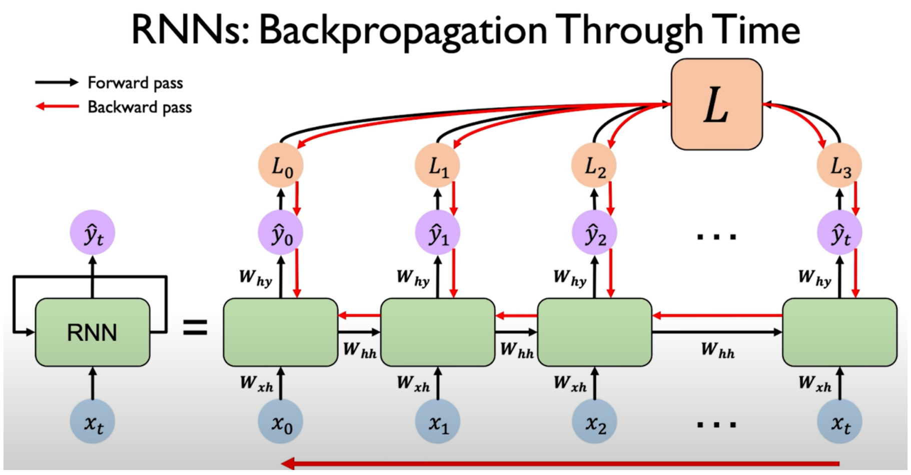

Figure 4: Backpropagation Through Time (**mitamini**)

Mathematically, (**mitamini**)

$$\frac{\partial L_n}{\partial W_x} = \sum_{i=1}^{n} \frac{\partial L_n}{\partial \hat{y_n}} \cdot \frac{\partial \hat{y_n}}{\partial \hat{h_i}} \cdot \frac{\partial \hat{h_i}}{\partial W_x}$$

2.2.4 Problems With Recurrent Neural Networks

Recurrent Neural Networks (RNNs) encounter challenges over long period of information. When performing backpropagation over a large amount of sequential data, the gradients diminish to zero, thus, the states that are far away from the current time step don't contribute to the parameters' gradient computing. This problem is known as the vanishing gradients problem. Likewise, the gradients may also grow exponentially large, which causes significant computational bottleneck and instability in the weight values. This problem is known as the exploding gradients problem(**salehinejad2018recent**).

2.3 Long Short-Term Memory (LSTMs)

2.3.1 Introduction to Long Short-Term Memory (LSTMs)

LSTM is an extended version of RNNs that can solve the vanishing and exploding gradients problem. In simple words, LSTM works by discarding unwanted information and preserving essential information in the hidden state. If we look at the example sentence above, *"Eww, This food is ____"*, we understand that the word 'Eww' holds more value than the word 'the' or 'is'. Hence, our brain automatically discards those words when processing information. With longer sequential information, huge amounts of unwanted information can be discarded by LSTMs. Structurally, LSTM contains gates that control the flow of information.

2.3.2 Forget Gate

As the name implies, the forget gate gets rid of unwanted information from the current state (**colah2015lstm**). Mathematically, it can be defined as

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

Where:

- f_t = Forget gate
- W_f = Weight matrix for the forget gate
- h_{t-1} = Output at the $t-1$ time step
- x_t = Input at the t time step
- b_f = Bias for the forget gate

The range of this function is [0,1] where 0 means no information will pass through and 1 means all the information will be passed through. Figure 5 shows the schematic of a forget gate.

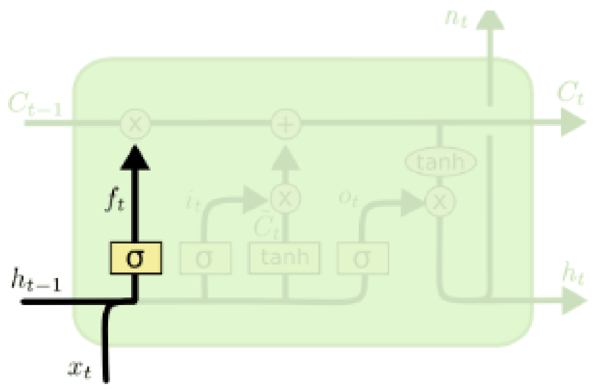

Figure 5: Forget Gate (**colah2015lstm**)

2.3.3 Input Gate

This gate is responsible for adding new information and updating the hidden state. It has two components: the input gate layer and the candidate layer (**colah2015lstm**). Mathematically, the input gate layer can be expressed as

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

Where:

- i_t = Input gate
- W_i = Weight matrix for the input gate
- h_{t-1} = Output at the $t-1$ time step
- x_t = Input at the t time step
- b_i = Bias for the input gate

The candidate layer generates a vector of new candidate values that could be added to the state (**colah2015lstm**).

$$\bar{C}_t = \tanh(W_C[h_{t-1}, x_t] + b_C)$$

Where:

- $\bar{C}_t$ = Candidate value for the cell state

- W_C = Weight matrix for the candidate value
- h_{t-1} = Output at the $t-1$ time step
- x_t = Input at the t time step
- b_C = Bias for the candidate layer

Figure 6 shows the schamtic for calculating the candidate values.

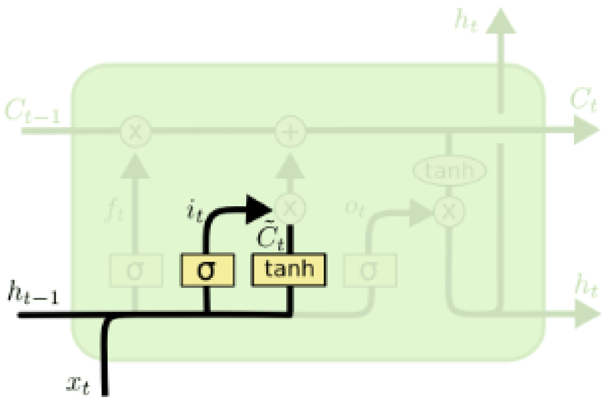

Figure 6: Calculate candidate values (**colah2015lstm**)

Finally, the cell state is updated by combining the old cell state and the new candidate value (**colah2015lstm**).

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \bar{C}_t$$

Where:

- C_{t-1} = Previous cell state
- f_t = Value from the forget gate
- i_t = Value from the input gate
- $\bar{C}_t$ = Candidate value

Figure 7 shows the schematic for updating the cell state by adding the candidate values.

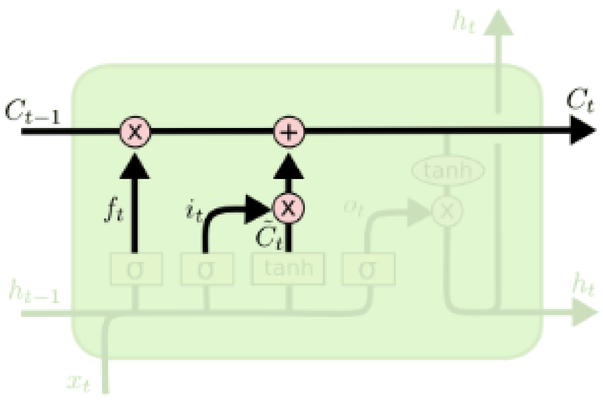

Figure 7: Update the cell state to add the candidate values (**colah2015lstm**)

2.3.4 Output Gate

The output gate determines which parts of the cell state to output at a given time step (**colah2015lstm**). Mathematically, the output gate is defined as

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

Where:

- o_t = Output gate value at the current time step
- W_o = Weight matrix for the output gate
- h_{t-1} = Output from the previous time step
- x_t = Input at the current time step
- b_o = Bias for the output gate

The final output of the LSTM unit is calculated by passing the cell state through a tanh function, which is then element-wise multiplied by the output gate value (**colah2015lstm**).

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

Where:

- h_t = Final output of the LSTM block at the current time step

- C_t = Cell state at the current time step

Figure 8 below shows the schematic for the output gate.

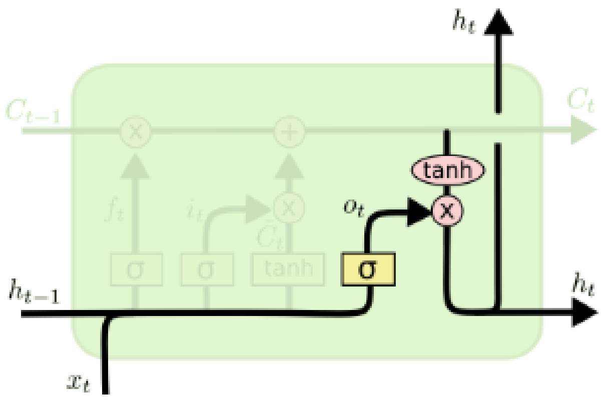

Figure 8: Output Gate (**colah2015lstm**)

2.4 MIDI Protocol

MIDI is a protocol that stores musical information in the form of multiple bytes, which creates small file sizes and the ability to analyse music numerically. It contains the following information.

Note message	Determines if a note is being played or not.
Note number	Numerical representation of a pitch ranging from 0 to 127
Key velocity	Intensity of the sound ranging of 0 to 127.
Channel	Determines the instrument being played
Timestamp	Determines when to start and stop playing a note

Table 1: Description of MIDI Protocol Parameters. (**c1s2midi**)

3 Experimentation

Experimentation is conducted on a code forked from the TensorFlow documentation (**tensorflow2021rnn**). The code for training and generation can be found in `EE.ipynb` and statistical analysis in `EE_Stats.ipynb` (**Koirala2024MusicLSTM**). Python3 was the programming language and the code was run on the Google Colab, which is a hosted Jupyter notebook. The experiment was conducted on a TPU, with 12.7 GB system RAM and 107.7 GB disk storage.

3.1 Dataset

The model was trained using the MAESTRO V3.0.0 dataset (**hawthorne2019maestro**), which contains about 200 hours of MIDI recordings, including those of JS Bach. The dataset also includes metadata in the form of a .csv file with the following information, shown in 2.

Field	**Description**
canonical_composer	Composer of the piece.
canonical_title	Title of the piece.
split	Suggested train/validation/test split.
year	Year of performance.
midi_filename	MIDI filename.
audio_filename	WAV filename.
duration	Duration in seconds, based on the MIDI file.

Table 2: Metadata Fields in the Dataset (**hawthorne2019maestro**)

3.2 Pre-processing

Before training the data, some pre-processing is required to decrease the complexity for training. Pre-processing involves transposing all the dataset into a single key. In music, there are 12 distinct notes, of which a key is a combination of these notes. Keys can be broadly categorised into Major and Minor, where the major key conveys a happy and joyful mood, and the minor key imparts a sad and dark mood. Additionally, each major key has a relative minor key composed of the same notes but in a different order. For example, the key of C Major consists of the notes C-D-E-F-G-A-B, while the key of G Major comprises G-A-B-C-D-E-F#. Similarly, the relative minor of the key of C Major is the key of A Minor, which contains the notes A-B-C-D-E-F-G. Transposing all the MIDI files to the key of C Major or A Minor reduces the number of unique notes that the network has to learn, and in theory should increase the performance of the network. The dataset was transposed using the 'music21' library in python. In conclusion, 145 files were transposed in the span of 4 hours.

3.3 Processing

The MIDI files are separated into three components, `pitch`, `step` and `duration`. The `pitch` is the note number, `duration` is the time a note is sustained, and `step` represents the time difference between two consecutive notes. The `pitch` and `duration` are taken from the MIDI protocol, as seen in table 1. The training dataset is divided into groups of note sequences which will be the input to the network. The length of the note sequences is manually set according to `seq_length` variable. The label will the next note, which is to be predicted by the network (**tensorflow2021rnn**). All the data is converted into a TensorFlow dataset. Finally, to prevent overfitting, the dataset is divided into batches and shuffled randomly.

3.4 Model Architecture

Using the TensorFlow `model.summary()`, the model architecture can be seen below.
The model's input layer `input_2` is designed to accommodate sequences of notes, where each note is represented by a combination of three features: `pitch`, `step`, and `duration`. This is reflected in the input shape, which is set to (`seq_length`,3). This design choice allows the model to handle variable-length inputs.

There are two LSTM layers utilised where, the first LSTM layer has 256 units and returns sequences, which helps in the utilising the hidden information and the second LSTM has 128 units and does not return sequences. The use of double LSTMs helps in capturing

Table 3: Model Architecture Summary

Layer (type)	**Output Shape**	**Param #**	**Connected to**
input_2 (InputLayer)	$[(None, 10, 3)]$	0	[]
lstm_2 (LSTM)	$[(None, 10, 256)]$	266240	['input_2[0][0]']
dropout_1 (Dropout)	$[(None, 10, 256)]$	0	['lstm_2[0][0]']
lstm_3 (LSTM)	$[(None, 128)]$	197120	['dropout_1[0][0]']
duration (Dense)	$[(None, 1)]$	129	['lstm_3[0][0]']
pitch (Dense)	$[(None, 128)]$	16512	['lstm_3[0][0]']
step (Dense)	$[(None, 1)]$	129	['lstm_3[0][0]']
			Total params: 480130 (1.83 MB)
			Trainable params: 480130 (1.83 MB)
			Non-trainable params: 0 (0.00 Byte)

more complex data as each layer can be used to capture a different layer of abstraction and higher parameters means more data and hidden information can be captured and utilised (**LSTMMultipleLayer**). Likewise. a dropout layer is used to reduce overfitting by randomly dropping units during training to prevent their co-adaptation (**BALDI201478**).

The three components, all, utilise a dense layer. A dense layer is a type of layer in which each neuron is connected to all the neurons of the previous layer (**verma2021dense**). The `pitch` component has a dense layer with 128 units, corresponding to the MIDI note range, as seen in 1, whereas the `step` and `duration` only have a single value, corresponding to them either being on or off.

Here, two different loss functions are utilised. The `pitch` component utilises the sparse categorial crossentropy loss function, due to its suitability for classification related tasks (**rahmanmath**). While the loss function is used alongside the softmax activation function, which converts the output of a network into a probability distribution, the `pitch` value is treated as a logit, which is the unnormalised probability distribution of the network which decreases the computational complexity. For the `step` and `duration` outputs, a positive mean-squared-error is used, to prevent negative values.

Mathematically,

$$C = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + 10 \cdot \max(0, -y_i)$$

Where,

- y_i = Output Received
- $\hat{y}_i$ = Output Expected
- n = Number of training samples

Finally, the Adam optimiser is utilised with a learning rate schedule. The learning rate schedule dynamically sets the learning rate, which is more computationally efficient (**377972**). Adam is also a computationally efficient algorithm which also provides bias correction (**kingma2017adam**).

3.5 Training

Initially, the network was meant to be trained for 100 epochs, but auto-stopped after 50 epochs after no changes were made to the loss value after 5 consecutive epochs. For variation in generated files, the `temperature` is varied between the values ranging from 1.1 to 13.0.

Variable	Description	Value
`seed`	A random seed used for reproducibility. It ensures that random operations produce the same results each time the code is run.	42
`total_bach_files`	The total number of processed MIDI files.	145
`epochs`	The number of training epochs, which defines how many times the entire dataset will be used to train the neural network. The network has the ability to stop the training process early if there is no significant change.	100
`training_files`	The number of training files used for training the neural network.	72
`output_length`	The length of the generated output in terms of data points or time steps.	300
`input_length`	The length of the input data used for training.	2000
`temperature`	A parameter that controls the randomness of generated predictions. Higher values (e.g., 13) make the predictions more random, while lower values make them more deterministic.	1.1 - 13
`num_predictions`	The number of predictions or outputs to generate during the generation process.	300
`seq_length`	The length of sequences used for training or generating music.	10

Table 4: Dependent Variables

4 Analysis and Evaluation

For comparison, 20 generated, and training files will be chosen, out of which only the first 30 seconds from each will be considered. Due to music being a highly subjective art form, which is subject to culture, personal taste and beliefs, the comparison won't be done on the basis of "good" or "bad", rather it will be done on statistical factors. The comparison only aims to understand if the generated music is similar to the training music. The data is collected and visualised using a Jupyter notebook (**Koirala2024MusicLSTM**).

4.1 Distribution

For distribution, three components are analysed, which are the `step`, `pitch`, and `duration`. As mentioned above, `pitch` refers to the note being played, `step` to the time difference between two notes, and `duration` to how long a note is played. From these components, three distributions are found, which are found below.

4.1.1 Total Distribution

In regard to pitch, the generated and training files show strong similarities, which shows that the network was able to replicate most of the notes played by Bach. In figures 9 and 10, there is a strong similarity of notes being played in the range of 50-80, which corresponds to the common notes played by Bach. However, it is also important to note that there are also notes of range 0-20 and 100-120 in the generated files, not seen in the training files, which implies that the network played notes not present in Bach's music. Likewise, the count for the generated files is nearly half that of the training files at 200 count in contrast to 500 counts. This also suggests that fewer notes were being played. In conclusion, the network was able to perform pretty well in regard to playing the correct notes except for a few outliers.

On the other hand, the network performed poorly in `step` and `duration`. The training files distribution at figure 10 shows the `step` value ranging from 0.0 to 0.5, with the highest count centred at the range of 0.0 to 0.15. Likewise, the count value is 1600. The generated files had `step` value ranging from 0.00 to 1.75, which is quite high compared to that of the training files and the count value is 800, as seen in figure 10. This suggests that there is a lot of silence in the generated music, compared that to the training music, which is quite fast-paced. This can also be seen in the `duration` component, where the maximum of the time value ranges from 0.0-1.2 in case of the training files and 0-6 in case of the generated files. This also suggests that compared to the training files, a single note in the generated music is played for a very long time. Hence, both these graphs suggests that compared to Bach's music which is fast-paced with minimal silence between the notes, the generated files is quite slow, with long periods of silence and single notes being played for a long period of time, which suggests that generated music is monotonous and repetitive.

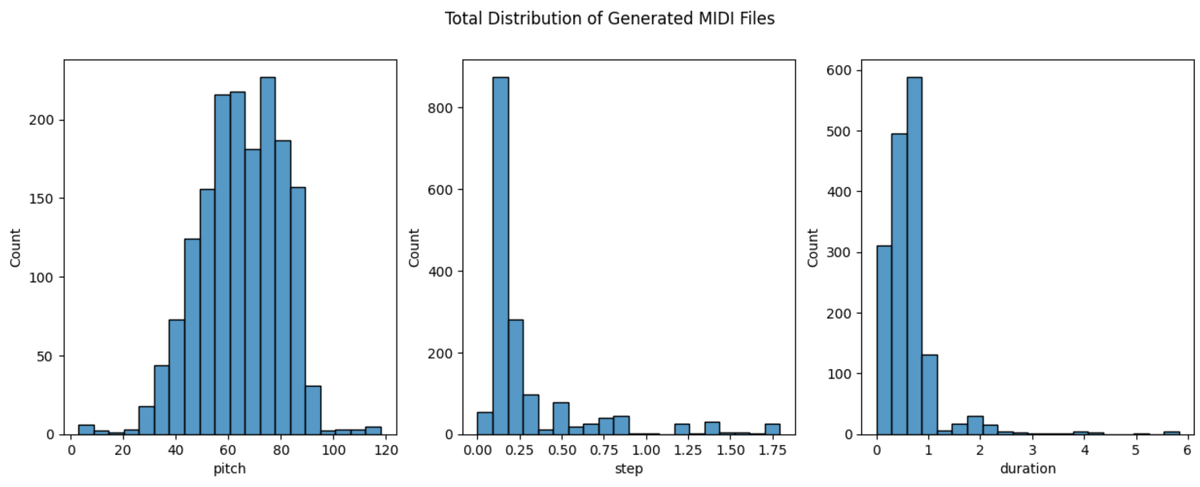

Figure 9: Generated Total Distribution

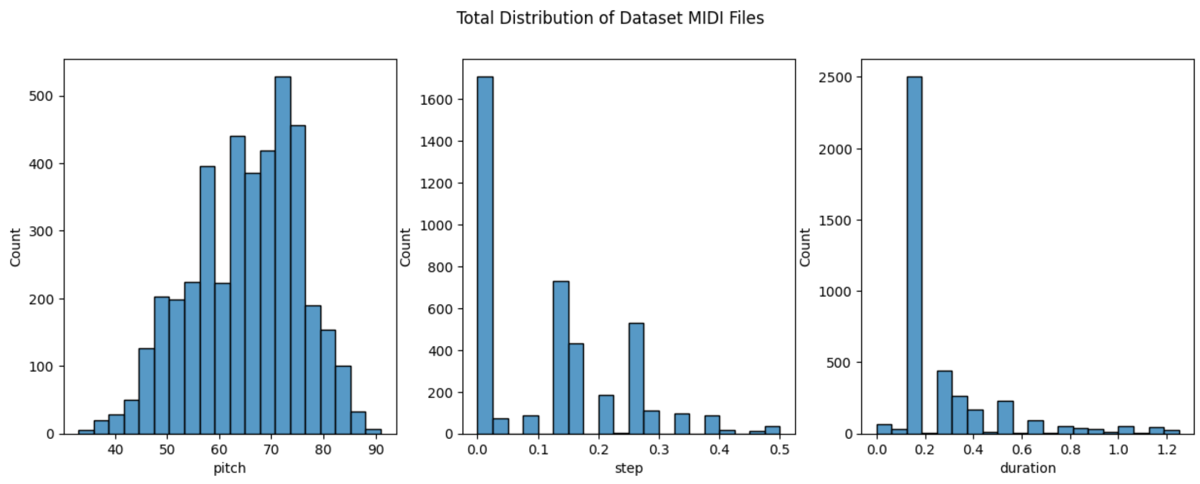

Figure 10: Training Total Distribution

4.1.2 Average Distribution

The data of the distribution of the components, in average, suggests similar conclusions to the section above. Figures 11 and 12 are quite similar to that of Figures 9 and 10.

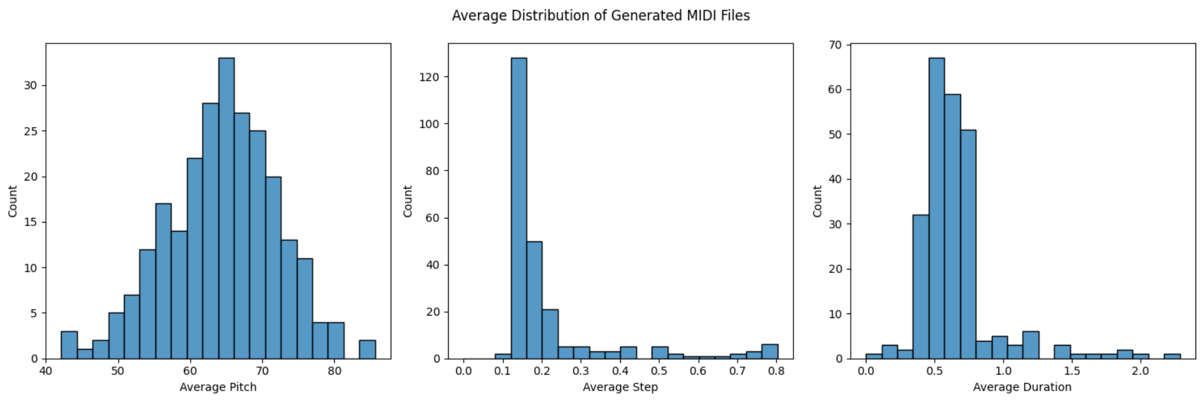

Figure 11: Generated Average Distribution

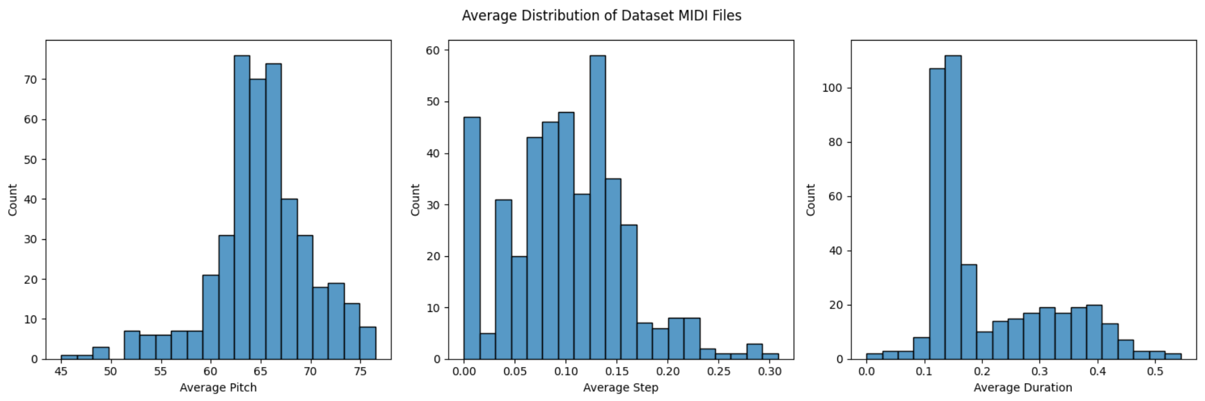

Figure 12: Training Average Distribution

4.1.3 Sample Distribution

Figures 13,14 and 15 are the distribution of sample generated files and Figures 16, 17 and 18 represent the training files. While the training files are consistent with each other, vast inconsistencies can be seen in the generated files. For example, Figure 13 shows that most of the notes were played for 1.75 seconds, with each note being played every 1.75 seconds. Likewise, the `pitch` distribution of figure 13, suggests only 7 notes were played. In contrast, the `duration` component of figure 15 is skewed rightwards, which suggests some similarity to Bach's music, but the `pitch` is still not similar to that of the training files.

While the inconsistencies can be attributed to the differing `temperature` values, it shows that the music generated is not consistent and quite random. The generated music has no thematic patterns or recognizable melodies, which is found in Bach's music.

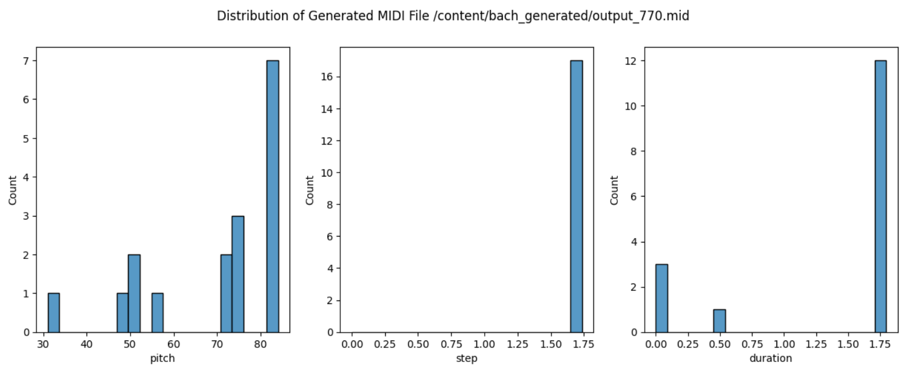

Figure 13: Generated Sample A

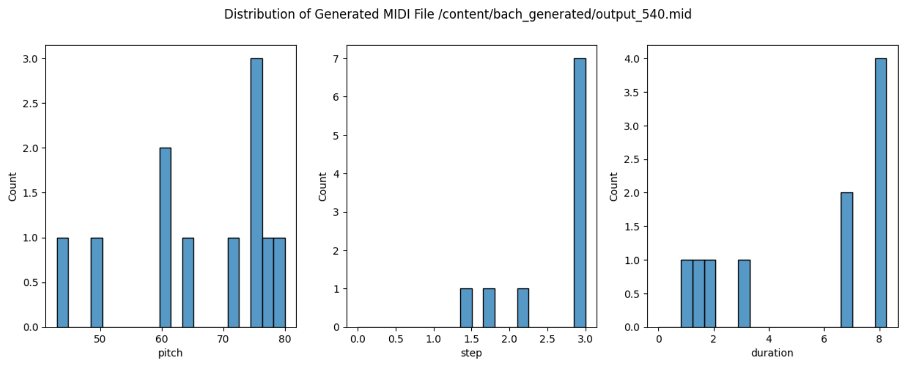

Figure 14: Generated Sample B

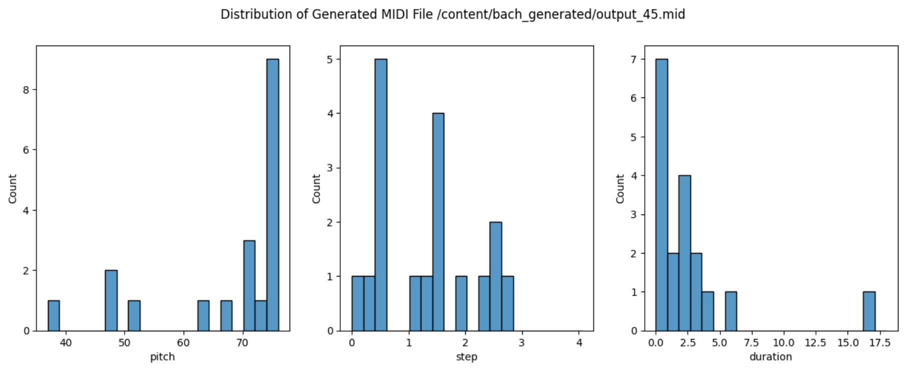

Figure 15: Generated Sample C

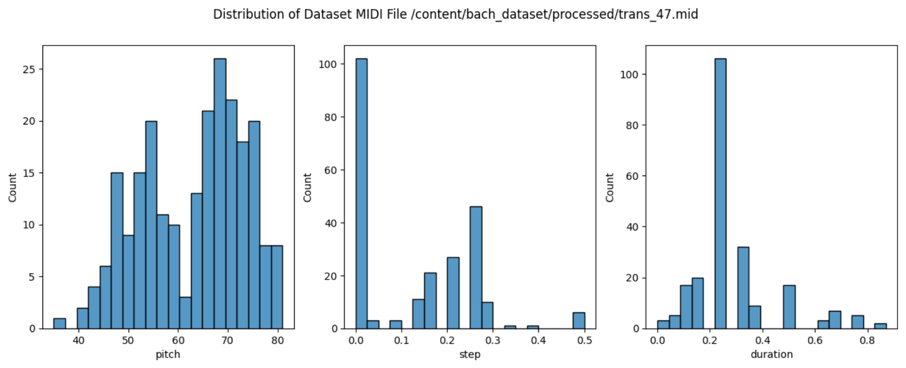

Figure 16: Dataset Sample A

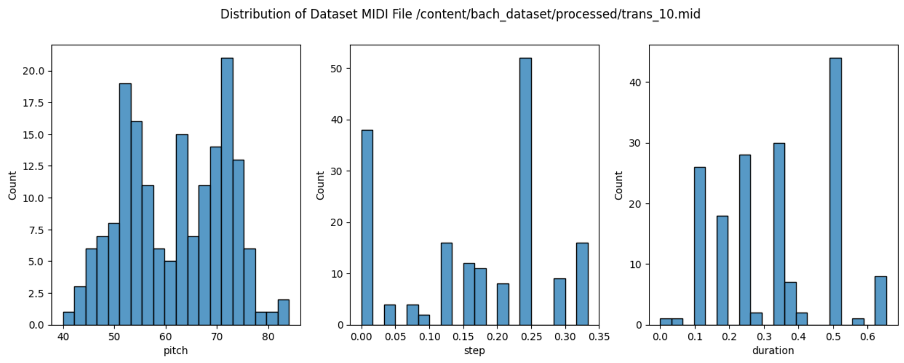

Figure 17: Dataset Sample B

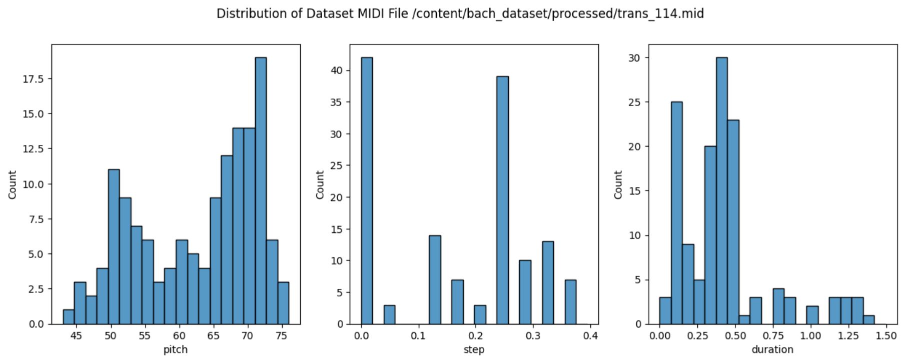

Figure 18: Dataset Sample C

4.2 Piano Roll Analysis

The piano roll compares time and `pitch`, to visualise the changes in notes being played over time and the duration of a note being played. Figure 19 shows the average generated piano roll and figure 20 shows the average training files piano roll. Figure 19 is much dense compared to that of the generated piano roll at Figure20. This again supports the conclusion that the network played a few notes for long periods of time, with longer silence in between notes. Moreover, the training piano roll at figure 20 is much more concentrated with in a span of a few notes, whereas the generated piano roll at figure 19 is spread all over the place with more varying `pitch` values. This suggests that the dataset piano roll is far more organised, having a fix structure, and the repetition of the pitches implies the existence of melodies within Bach's music. This again proves that the generated music is much more random in nature and has no structure.

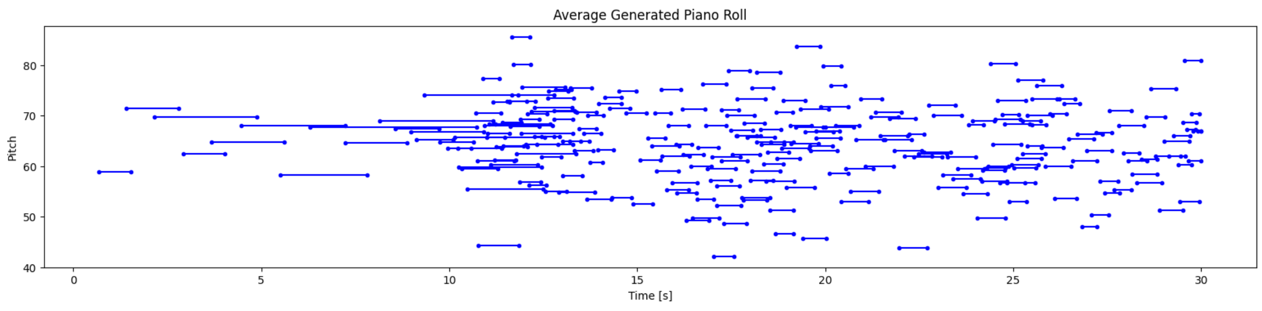

Figure 19: Generated Piano Roll

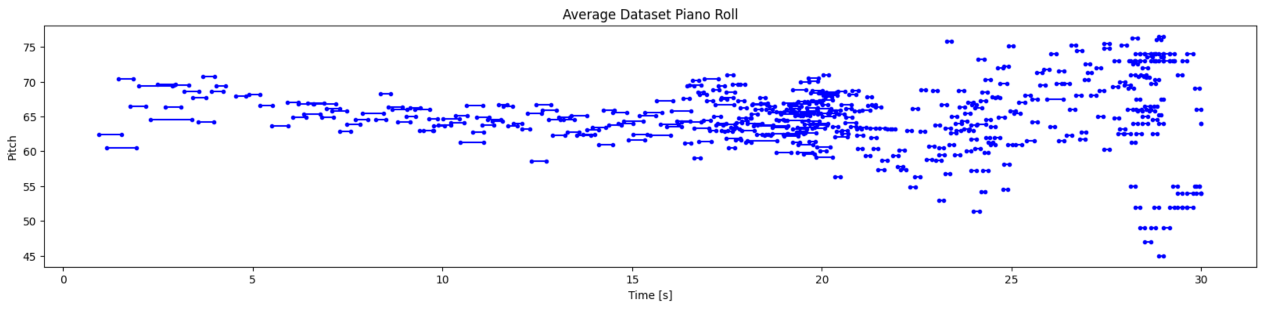

Figure 20: Training Piano Roll

4.3 Standard Deviation and Variance

Standard deviation is used to understand how close any given data is to its mean value (**BlandAltman1996**). The value of the standard deviation is directly proportional to how far the data is spread, which means the value of a higher standard deviation means that the data is highly inconsistent. Likewise, variance is the square of the standard deviation. While they are quite similar, variance helps us better understand the dispersion, by squaring the values.

Table 5 shows the Standard Deviation and Table 6 shows the variance.

MIDI type	**Pitch**	**Step**	**Duration**
Generated	13.65	0.2390	1.640
Training	9.036	0.2762	0.3899

Table 5: Standard Deviation

MIDI type	Pitch	Step	Duration
Generated	201.45	0.2171	7.270
Training	83.31	0.1086	0.2591

Table 6: Variance

The table presented above illustrates a significant deviation in the case of `pitch`. This may be attributed to the fundamental principle that each musical piece is distinct from another due to the different notes that are played. Songs that play the same notes are not considered distinct. Despite that, the variance for the generated files is quite high compared to that of the training files, which again suggests that the generated music has no structure or uniformity. Likewise, the variance for both the `step` and `duration`, is quite high, which suggests that some generated output could be very fast-paced while other outputs could be very slow-paced. This is also supported by the distribution in `step` and `duration` as seen in figures 13 and 14.

4.4 T-Test Hypothesis Testing

A T-test is a type of statistical test that is used to compare the means of two groups. (**kim2015t**). Conducting a T-test requires three fundamental data values, the difference between the mean values from each data set, the standard deviation of each group, and the number of data values. (**AdamHayesTtest**). Likewise, the p-value represents the probability of how likely it is that any observed difference between groups is due to chance (**dahiru2008p**). The result of the T-Test and p-value can be seen in table 7 below.

Category	T-Test	p-Value
Pitch	-0.1738	0.8621
Step	21.04	8.425×10^{-95}
Duration	17.38	5.548×10^{-66}

Table 7: T-Test for Training and Generated Datasets

Here, for the `pitch` component, the T-Test value is close to zero (0.8621), which suggests that there is no significant difference in notes being played between the generated and training files. This aligns with the distributions seen in figures 9 and 10. The negative sign of the T-test indicates that the mean of the generated files has slightly lower than the mean of the dataset notes, but this difference is not statistically significant. The high p-value implies that any observed difference in `pitch` is likely due to random chance.

The T-statistic for `step` and `duration` is significantly large and positive, indicating a substantial difference between the two outputs. Moreover, the p-value is minuscule, which indicates that the difference in `step duration` is highly statistically significant. Like the T-statistic for `pitch`, the data above, strongly correlates with the `duration` and `step` distribution.

In conclusion, the T-test results suggest similar findings to that of the distributions above where the `pitch` does not differ significantly between the two groups, but there are significant differences in `step` and `duration`. This means that the generated music tends to be more varied or erratic in terms of its rhythm and duration of notes being played. Likewise, the very low p-value fails to bypass the threshold of 0.05.

4.5 Time Signature

In music, time signature is a symbol used to denote the rhythm of a music. It is denoted as a fraction, where the denominator indicates the note value to be counted, and the numerator indicated how many such notes are there. A huge majority of western music uses the '4/4' time signature, which is considered as the standard rhythm. Figures 21 and 22 below shows the distribution of time signatures of both the dataset and generated files. Both datasets only contain the '4/4' time signature.

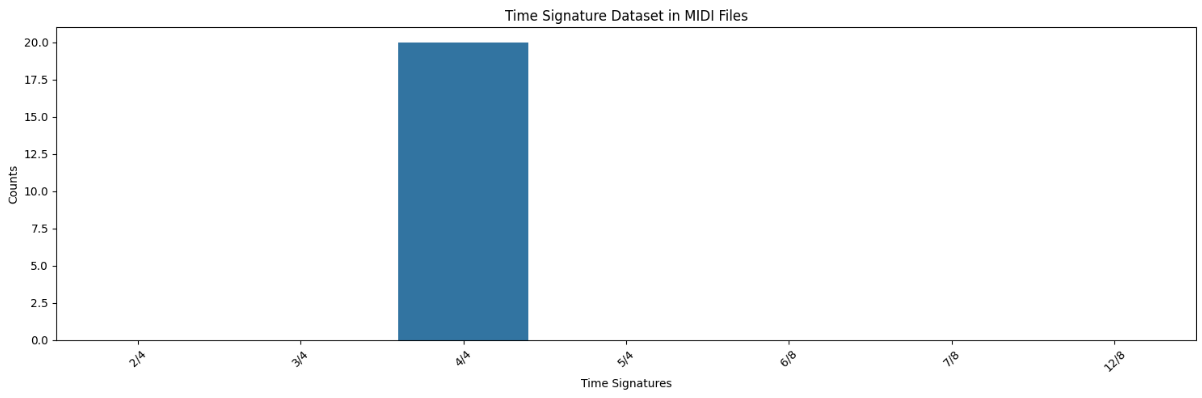

Figure 21: Time Signatures in Dataset

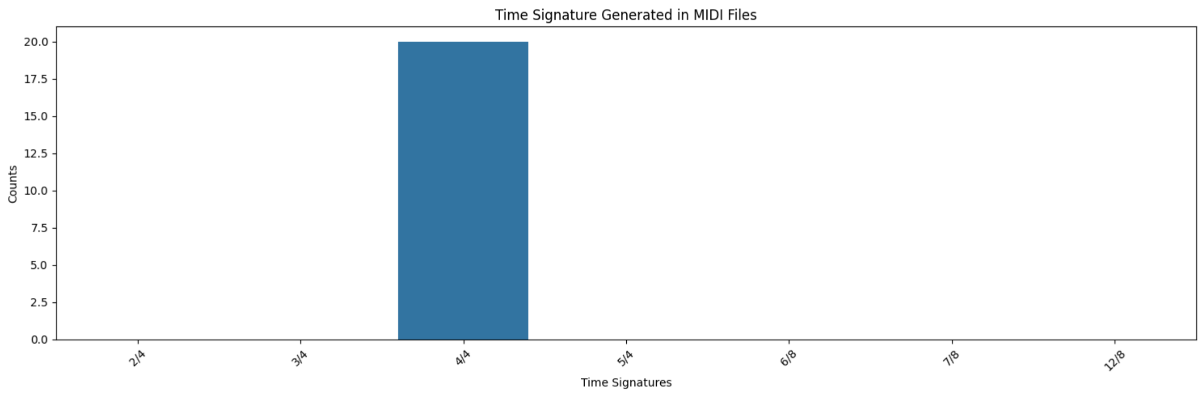

Figure 22: Time Signatures in Generated Output

4.6 Conclusion

In conclusion, all data suggests the same thing, that the network was not able to exactly replicate the music of JS Bach. While, it was able to replicate the correct notes (`pitch` component), the point of failure for the network was rhythm (both `step` and `duration`), as shown by the differing distributions, p-value and variances. While the network performed comparatively well on `pitch`, there will still some hallucinations, where the network played pitches of range 0-20 and 100-120. These pitches were not seen in the dataset at all, and the notes corresponding to the pitches could not be even played on instruments that Bach composed his music on.

4.7 Limitations and Further Research

There were a lot of problems and limitations faced in this experiment. The primary problem was overfitting, where only single notes were played for the entirely of the run time. One key factor that limited this research is also the use of sparse categorical cross entropy as the loss function, which is prone to outliers and overfitting (**rahmanmath**). Similarly, the use of a Generative Adversarial Network (GAN) alongside the LSTM could improve the quality of the music generation. Recent research has shown GAN's advantage in generating realistic data such as images and music (**goodfellow2020generative**).