TEXAS A&M UNIVERSITY
Department of
Biomedical Engineering

Milestone 4: Final Report

Team: If Mozart Wasn't Dead (IMWD)

Pablo Loo

Ricardo Martinez

Machine Learning and its Role in Music Generation

Pablo Loo & Ricardo Martinez – April 26th , 2021
Texas A&M University- BMEN 489 Project 2

Abstract

The goal of this project is to create a model that is able to produce synthetic original music that mimics the stylings of historical composers such as Wolfgang Amadeus Mozart, Ludwig Van Beethoven, Johann Sebastian Bach, etc. Contrary to the common practice of using audio data for music applications of machine learning, it was decided to use a novel approach that consists of implementing text-generation models. This is achieved by converting the music tracks to text MIDI format, which allows the user to manipulate the data and feed it as input to the models. Two main machine learning models were taken into consideration in order to achieve this goal: a network of Gated Recurrent Units (GRU), and a Long-Short-Term-Memory (LSTM) network. Despite the similarity between GRUs and LSTMs, GRUs proved to be the most efficient and effective model with a 61% approval rate and an overwhelming 70.4% Mozart influence according to a survey conducted with a total of 28 participants and a minimum loss of 0.004.

1. Introduction

Music has long been a part of humanity. The audible sense of expression seems so natural and effortless that many have been led to believe that its creation is simple by nature. What many fail to realize, is the inner workings of harmony and mastery, the mathematical relationship between notes, timing, and frequencies that dictate the pleasantness and overall experience that is transcended into the ears of its listeners. Even if one simply understands this, experience and feeling are so closely intertwined in the production of music, that it takes years in order to build the relationship between oneself and music that is required to to produce an enjoyable piece of music, not to mention develop one's own musical styling.

Said stylings are determined by factors such as the frequency and duration of the notes being played, the patterns amongst these notes, the key, speed, tempo, signature, and many other elements that are intrinsic to the individual's experience. But what if there was a way to reproduce the experience of the great composers before us? What if that raw connection and ability could be harnessed and translated into a model and applied to the generation of new music? The need for a melodic production tool is crucial as the same tired melodies are being countlessly used, with no room for artistic variation. The "vi–IV–I–V" progression, for example, can be applied to any key [1]. The key then dictates the chords that the progression is related to [1]. A short list of songs that utilize this same progression can be seen below [1].

.

Song Title	Artist	Year
"21 Guns"	Green Day	2009
"A Letter to Elise"	The Cure	1992
"Any Way You Want It"	Journey	1980
"Bullet with Butterfly Wings"	The Smashing Pumpkins	1995
"Despacito"	Luis Fonsi	2017

The first step in pursuing this goal was to find a way to convert audio data to text data. This was achieved by converting sheet music into a MIDI (Musical Instrument Digital Interface) format. Many websites offer this functionality currently. This text-converted sheet music serves as the input for the models, and their compatibility and ease of manipulation with digital audio workspaces allows the user to listen to the produced tracks and analyze them both in text and audio formats. This model will prove the relationship between the input inspiration and the synthetic music that is produced, ultimately cracking the musical formula and putting it into the user's hands.

2. Related Work

Mastering the music creation process has been a long sought after goal. While many attempt to force themselves to practice music and involve themselves in perfect pitch training, each process is extremely time consuming. Despite this labor-intensive approach, there are also those who have attempted to automate the music production process. In an attempt to protect musicians against copyright violations of music that they have never heard of, (Damien Richl and Noah Rubin, 2020) developed a model to produce all 68 billion, 12 note melodies that can be played on an 88 note keyboard [2].

While the scope and application of this project is strikingly different from the model that is being presented in this article, there are some similarities in approach. Richl and Rubin utilized MIDI language as well as a combination of Python and Rust in order to produce 68 Billion, 12 note scales taken from the first 8 notes of the C scale [2]. Although there was no input data serving as reference for musical style, Richl and Rubin were able to verify the ability to artificially produce melodies from a synthetic pool of notes.

2.1 Standard AI Method

Given the scope of this project, there are no definitive AI methodologies currently in place to tackle this issue in the same way that is intended in this project, which is through text generators. As mentioned before, most of the AI projects that deal with music work with audible data. The following section, however, will discuss a similar approach undertaken by (Skúli et.al, 2017) to generate music artificially using a LSTM model and MIDI text data.

2.2 Other AI Methods

The LSTM model constructed by (Skúli et.al, 2017) is a very simplistic architecture consisting of a sequential model with three LSTM layers each with 512 neurons as well as a few batch normalization, dense and activation layers embedded in between.

```
""" create the structure of the neural network """
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')
```

Fig. 1: Baseline LSTM Model Code

There is no preprocessing taking place other than change in dimensionality of the data and parsing through several files. Aside from that, the LSTM is built as it would for any text generator model.

3. Data Sets, Features, and Preprocessing

In order to process the input track and analyze its intrinsic properties, the digital sheet music was converted from graphical representation--

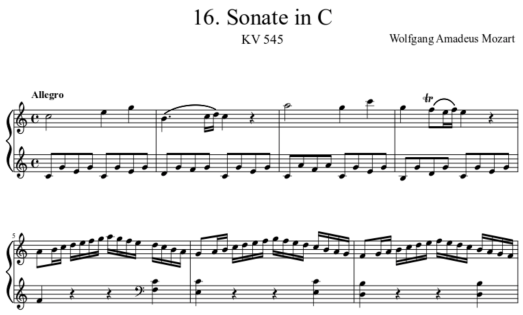

Fig.2: Sonata No. 16 in C major Sheet Music

--to a text-based representation.

```
note_on channel=0 note=71 velocity=49 time=0
control_change channel=0 control=2 value=49 time=0
note_on channel=0 note=71 velocity=0 time=119
note_on channel=0 note=69 velocity=49 time=1
note_on channel=0 note=69 velocity=0 time=119
note_on channel=0 note=68 velocity=49 time=1
note_on channel=0 note=68 velocity=0 time=119
note_on channel=0 note=69 velocity=49 time=1
note_on channel=0 note=69 velocity=0 time=119
```

Fig.3: Sonata No. 16 in C major, MIDI Format

This text-based formatting called MIDI, which was mentioned previously, allows for every single facet of the score, to be translated in a computer-friendly version. With a text version of the desired input track, the possibility to recognize patterns and distinguishable factors was now within reach.

This input track served as the input data set by providing a long list of translated information corresponding to key, number of notes, time played, speed, duration, and intensity, among other relevant pieces of information. This information would then be stored and recurrently used to draw inferences from in order to feed the musical predictions.

A number of preprocessing steps needed to be implemented prior to being fully utilized. To start, a forward and reverse StringLookup layer needed to be instantiated in order to vectorize the dataset. That is, every character needed to be converted to a special character ID, based on tokens. These ID's would serve as unique identifiers that are more readily able to be processed by computers, in line with a unicode formatting. Next, the data was normalized to values between 0 and 1. With prediction in mind, the next step of preprocessing was to further separate the dataset into input sequences of *n* characters. For every input sequence, a matched target sequence of the same length was created, shifted to the right by one character [4].

Lastly, the final preprocessing technique involved track selection. More specifically, each of the two tracks correlate to the hand being played on a piano. Track [0] correlates to the right hand melody while track [1] correlates to the left hand melody being played. Since the sheet music denotes both hand melodies on the same score, the model needs to be told which hand to learn from, prior to training. In virtue of this, track [1] (left hand) was selected due to its more predictable nature and stability.

4. Preliminary Model Architectures

ORIGINAL LSTM ARCHITECTURE	
LAYER	**DESCRIPTION**
LSTM (3)	Multiple "hidden" LSTM layers. Increases levels of abstraction, relating each output to an individual input
Dropout (2)	Helps to reduce overfitting by removing units. Dropout = 0.3
Dense (2)	Dense layer of 256 neurons, embedded and output dense layer with n* number of neurons
Activation (2)	"relu", "softmax"
Optimizer	rmsprop

n is the vocabulary size*
Table 2.Original LSTM Architecture

GRU ARCHITECTURE	
LAYER	**DESCRIPTION**
Embedded	Serves as the input layer. Trainable lookup table that matches ID to a vector of 256 dimensions
GRU	Main recurrent layer. Contains 1024 RNN units
Dense	Output layers, whose size is determined by the length of unique characters in the dataset. Outputs the likelihood of each character's appearance using the logits function

Table 3. GRU Architecture

4.1 Baseline Model or Algorithm

In order to gauge initial success, a baseline model was initialized. This baseline model is a LSTM network based on the model fabricated by (Skúli et.al, 2017), which was also used for Music Generation through text data. This model was discovered, post project-initialization, which only confirms the demand for machine learning in music applications. Although a GRU approach was primarily taken initially, this LSTM approach was branded as a baseline, given the relative success with the other team. Initial results revealed that a lack of temperature, which is the ability to internally dictate the degree of randomness of predictions, caused the model to repeatedly produce the same note for the entire sequence duration. While the loss, which can be seen below, was relatively low, it was clear that the initial model needed extreme revamping.

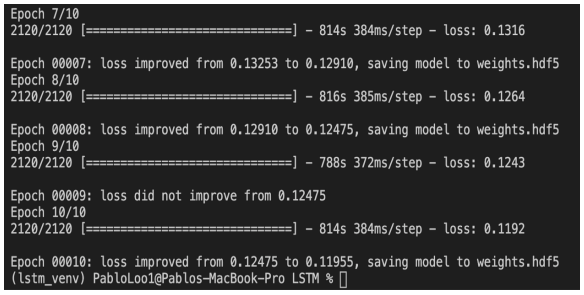

Fig.4: LSTM Training Loss

4.2 Comparative Models

For the sake of comparison, the initial approach was expanded to include two alternative models: GRU and a modified version of the LSTM. The potential for LSTM application has already been realized, to an extent, therefore expanding beyond this initial model is crucial in developing efficiency and improving results. The following two sections will delve further into both model's architectures.

4.2.1 GRU Model

The GRU presented in this paper is a simple model consisting of only a few layers. The first is an embedded layer, whose main task is to map each character ID to a vector with *N* embedding dimensions. Afterwards, the data goes through a GRU layer, which is a type of RNN layer similar to LSTM. The last layer consists of a dense layer with *n* **logit** outputs, where *n* is equal to the vocabulary size [5]. The architecture can be seen below in *Fig. 5*.

```
# Class for first model
class MyModel(tf.keras.Model):

    def __init__(self, vocab_size, embedding_dim, rnn_units):
        super().__init__(self)

        # Create input layer:
        # A trainable lookup table that will map each character-ID
        # to a vector with embedding_dim dimensions
        self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)

        # Create GRU layer:
        # A type of RNN with size units=rnn_units (You can
        # also use an LSTM layer here)
        self.gru = tf.keras.layers.GRU(rnn_units,
                                       return_sequences=True,
                                       return_state=True)

        # Create output layer:
        # Layer with vocab_size logit outputs.
        # logit is the logarithmic
        # of the odds for each character in the vocabulary
        self.dense = tf.keras.layers.Dense(vocab_size)
```

Fig. 5: GRU Architecture Code

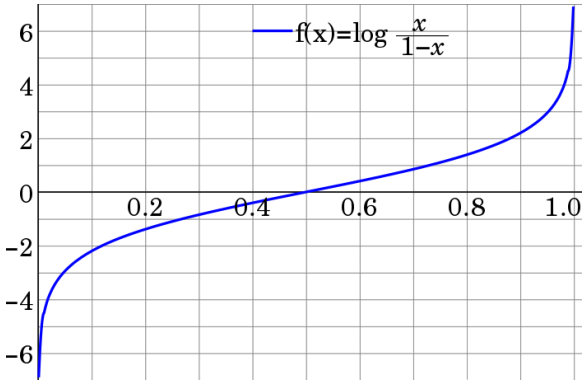

Fig. 6: Logarithmic of the Odds Function

The logit function, also known as the logarithmic of the odds is a "type of function that creates a map of probability values from (0,1) to (-∞, ∞)" [4].

$$logit\,(p) \;=\; log\,(\tfrac{p}{1-p});\; p \;=\; probability$$

The resulting probability is used in a recurrent process to determine what is the next most likely outcome (character). In order to perform this process, single step predictions are made with a controlled temperature. In order to prevent any errors that may arise due to incompatibility or failure to recognize a set of data, two filters were created, one during preprocessing and another after new data is created by the model.

The first filter is a mask, which prevents any unknown ([UNK]) or blank ([" "]) values from interfering with the process. In order to achieve this, these outliers are assigned a predictive value equal to -∞ so that they are placed in a very inconvenient spot in the logit function probabilistic map.

The second filter prevents any unrecognizable midi instruction created from interfering with the process. This time, instead of masking the outlier, it is simply skipped. Such a task is achieved by using the try-except method, which allows the model to create instructions based on whether they are valid or not. The last step is to feed the stepwise output of the model into a single string, which is then processed to resemble a set of midi instructions. Once the string has been formatted, it is then converted into real instruction using the eval() function, which evaluates a string and converts it into a usable form.

4.2.2 Modified LSTM Model

The modified version of the LSTM is simply the original baseline model with an increase of epoch count from 200 to 2000. A block of code was also added to prevent the same note from being generated over and over, as it did with the original baseline LSTM network. The strategy that was used was to generate a random number from 1 to *k*, and the value that *k* took on in a given iteration would be used to generate the k^{th} most likely predicted value from a sorted list of the predictions, from

lowest to highest, rendering the last element as the most likely to be next. This idea was tested and it successfully assisted in getting rid of nodal stacking.

```
                            prediction    =
model.predict(prediction_input,
verbose=0)        i = randrange(1, 5)
                                  index     =
sorted(list(prediction[0])).index(sorte
d(list(prediction[0]))[-i])
       result = int_to_note[index]
```

Fig. 7: Modified LSTM Prediction Code Block

5. Training

This model required a large dedication of time towards optimization in order to get a more harmonic output track. The most direct means by which to achieve a better trained model is to increase epoch count. With this in mind, initial variation took place between 5 and 200 epochs. As expected, the lower epoch counts produced tracks that were less than desirable. Given past experience with comparative models, it was initially believed that 200 epochs would be more than enough training, and would likely lead to an analogous version of overfitting for generative models. In a shocking turn of events, it actually failed to produce a harmonious track yet again. A positive observation that was taking place, however, was that after importing the list of produced tracks into GarageBand and analyzing their scales, it became clear that the output tracks were in fact maintaining the same note range as the input track, which meant that at least the model was not generating random notes out of the range of the input data, which was speculated, would be impossible. Thus, this event confirmed the hypothesis.

With this fact in mind, training became more intensive, with an epoch count of 2,000 epochs. At this level of training, there was a huge leap in quality. Since this project is no longer dealing with categorical models, loss is the only quantitative measure of this model's success. In general, there was no apparent overfitting that occurred at any of the epoch count trials. In terms of loss plateaus, each successive trial, even at 2,000 epochs continued to minimize loss as training proceeded. A sample loss vs epoch count tensorBoard plot for a 200 epoch trial can be seen below for visualization.

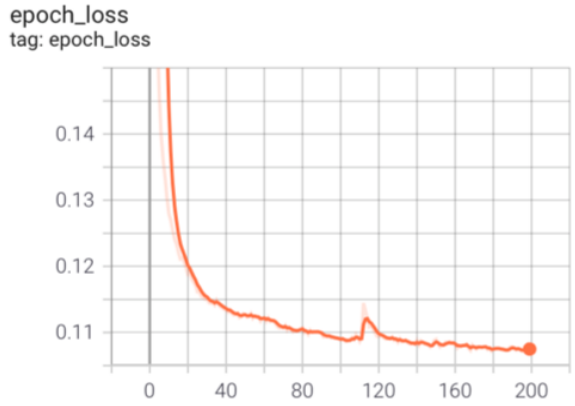

Fig. 8: GRU Tensorboard Loss at 200 Epochs

6.0 Experiments and Results

In order to gain a better understanding of which model would present the best potential for improved outputs, a set of optimization trials were run on both the GRU model and modified LSTM model. While there were a few key differences in testing, the generalized parameters as well as their baseline values can be seen in *Table 4.*

Parameter	Baseline Value
Loss vs Epoch Count Plot	N/A
Sequence Length	100
Output Range	10,000
Batch Size	64
No. of MIDI tracks	1
Range of Output Notes	C3-D5
Metadata	None
Temperature	1
Epoch Count	20
Prompt	[‘control_change:’]
Additional RNN Layers	Only default GRU Layer
Input Songs	Turkish March

Table 4. Baseline Parameter Values

As noted, a parameters study was conducted in order to distinguish the most beneficial variables related to the model’s success as well as pinpoint their optimal values. Of the 5 main parameters: sequence length, prompt, song choice, output range, and temperature, sequence length was the first to undergo experimentation. Sequence length serves as the length by which the input data is split for recurrent analysis. Values were varied from a length of 5 to 100 characters, allowing for a quick realization that a minimum length count of 80 characters was required in order to avoid misspelling in outputting track information, since words were being cut off prematurely. Ultimately, a sequence length of 100 was maintained for optimal performance.

Prompt was the next variable that underwent variation. Prompt serves as the phrase that initiates the recurrent generation. The baseline prompt was set as *control_change*, and after having run enough initial testing, it was quickly changed to *note_on*. Repeated testing across all epoch counts soon revealed a lack of variation between the two. After a few more variations with less intense testing, prompt proved to be indifferent to the harmonic change that was expected. With this said, *control_change* was maintained as the optimal prompt.

When it came to song choice, it was initially believed that the complexity of Turkish march would benefit the model, as to have more information to pull from, but the opposite proved to be true. When the song choice was switched to the more simple left hand melody of Sonata Number 16, the model drastically improved results. This improvement is likely due to the fact that the model was now able to confidently map the patterns and store the intrinsic style of the piece in its memory. With this storage, the output track proved to be cohesive in its sentences and arrangement of notes.

Output range reflects the duration of the desired output, which started off with 10,000 time units, but needed a longer track to give room for variation. With that, the value was optimized to an output range of 100,000 for about a minute and a half worth of audio.

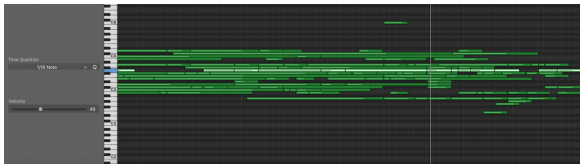

Figure 9. Output with: Epochs: 20. Prompt: Control change. Note Range: C3 - D5

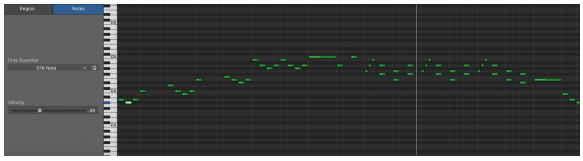

Figure 10. Turkish March Input. Note Range: C3 - D5

Finally, the last parameter to get optimized, was temperature. Temperature determines the randomness of the predictions, and when tinkering with values from 0.2 to 5, it was quickly discovered that by going too high or too low with randomness, the model would underperform. With that, the optimal value for temperature was set at 0.8. The final set of optimized values can be seen in *Table 4.*

Parameter	Optimal Value
Sequence Length	80 -100
Output Range	>10,000
Batch Size	64
No. of MIDI tracks	>1
Range of Output Notes	C3-D5
Metadata	Yes
Temperature	0.8 - 1.0
Epoch Count	>2000
Prompt	*NOE
Additional RNN Layers	*NOE
Input Songs	Sonata No.16 in C major

* NOE = No Observable Effect

Table 5. Optimized Parameter Values

6.1 Model Evaluation

As mentioned before, in order to evaluate the performance of both comparative models, which are the modified version of the baseline LSTM and the GRU network, more quantitative data was needed besides loss, so a survey was conducted to serve as a human expert validation method. The following are graphs that portray the results pertaining to the overall approvals of the respective melodies composed by each model. The details of each graph will be explained in a description below belonging to each graph.

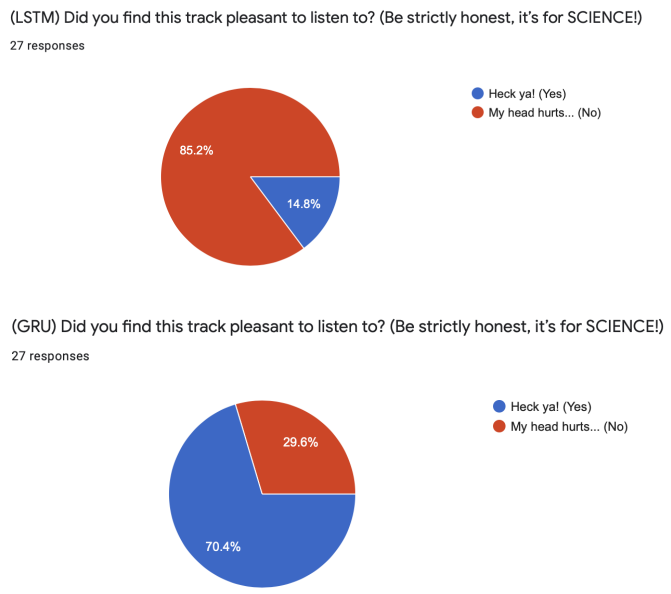

Fig.11: Poll Results

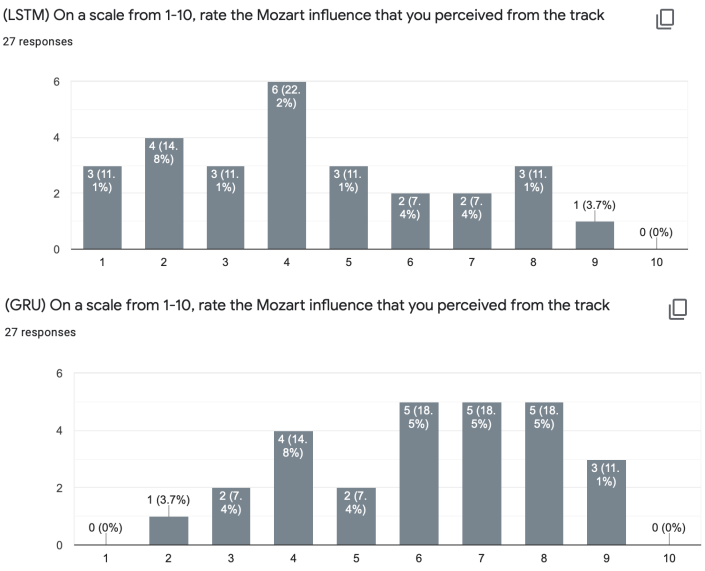

Fig. 12: LSTM/GRU Similarity Distributions

This LSTM model served as this project's baseline model, and it was soon realized that there are a few advantages that the GRU

approach, undertaken in this paper, has over the baseline LSTM, as well as a few details that will serve as future improvements. First, the current LSTM implementation does not support varying duration of notes or different offsets between them. The GRU model in this paper, however, does in fact support varying duration of notes as well as offsets between notes. This is achieved through the use of the "time" variable found in MIDI language, which denotes the duration and location of a note in reference to a previously played note. Another point worth noting is that the LSTM, due to a combination of the number of layers, parameters, and the amount of data needed to produce a satisfying result, among other things, cause the LSTM to require a much more involved training time. With this longer training period, a higher epoch count and computational power is then required. This project's GRU approach, which only contains three layers in its architecture as well as a one song requirement, makes it much more efficient when compared to the baseline LSTM model.

To further support this statement, (Skúli et.al, 2017) stated that "it took approximately 20 hours to train the network for 200 epochs". Comparing such observations with the fact that it took the GRU presented in this paper roughly 13 hours to complete a total 2000 epochs. Overall, this indicates that the GRU in this paper is 1,438% more efficient than the LSTM.

Another major advantage that the GRU presented in this paper has over the baseline LSTM, is that the LSTM model failed to include a method to handle unknown notes. As it is now, the network would enter a fail state if it encounters a note that it does not know. The GRU presented on this paper, on the other hand has a total of two filters that deal with unknown values. One filter prior to training, during the preprocessing step and the other one post training during the conversion from string to MIDI instructions. The first filter is a "mask" and the second is a "try-except" built-in python method. Both of these filters serve to prevent ["UNK"] or ValueErrors from hindering the processes of the model and will be further elaborated on in the following sections.

7. Final Model

A total of two models were evaluated in this study. The first model, which happens to be the baseline model, is a modified version of the LSTM network created by (Skúli et.al, 2017), with the only differences being that the epoch count was increased by a factor of 10 for a total of 2000 epochs and since the LSTM lacks a temperature parameter, a block of code was added that will take a random value among the best "k" number of predicted values. The last model that was tested was a GRU network. After experimenting with the networks and evaluating the results of each one and performing a survey for validation, a decision was made to select the GRU as the final model.
The following is the GRU model summary for visualization.

```
Model: "my_model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        multiple                  11264
_________________________________________________________________
gru (GRU)                    multiple                  3938304
_________________________________________________________________
dense (Dense)                multiple                  45100
=================================================================
Total params: 3,994,668
Trainable params: 3,994,668
Non-trainable params: 0
_________________________________________________________________
None
```

Fig. 13: GRU Model Summary

Because this is not a categorization problem, there was no need to use regularization, augmentation, object detection, landmarks, etc. Confusion matrices are specific to categorical models, so they are not applicable either. Because the model works with text data rather than image data, there was no need to use the Jetson Nano or a GPU. However, there

were some preprocessing methods applied which are discussed in Section 3.

8. Discussion

As has been mentioned before, this is not a classification problem, so there are no metrics inherent to a confusion matrix such as accuracy, prevalence, F score, Diagnostic odds ratio, true positive rate, False negative rates, etc. There are, however, other metrics that make up for this, such as loss and validation, which are referenced throughout the paper.

An important point to touch upon were the data challenges that were stumbled upon at the early and late stages of the project. Although underrated, it is widely known throughout the AI community that data can often be the most influential factor in a model and sometimes it can even become a problem that puts the whole endeavour in jeopardy. Fortunately, this was not the case, there was the advantage of having an abundant amount of data that not only is stored in a universal format (MIDI) but can also be extracted from many different websites located in the biggest library in the world which is the Internet.

Many websites offer MIDI versions of nearly any popular song or melody that has existed since the beginnings of the modern era, which provides the user with a surplus of data to choose from. For this application, however, it was decided to restrict the data to one song until a better understanding of the model was gained. To summarize the information above, in this case data did not represent the slightest burden to this project. The real problems arose when working with several models. It was soon learned that it was necessary to create a new environment for every model that was worked on for a total of two environments. It was soon discovered how specific Tensorboard can be in terms of its requirements. Tensorboard has a way of restricting the version of every package that one works with in a given environment for compatibility. This led to a myriad of problems and even the abandonment of using Policy Gradient Generative Adversarial Networks (SeqGAN) for this project.

Another challenge that arose in the early stages due to errors in compatibility between Tensorboard and other packages was that the software was unable to save the models in their full form. This led to the solution of saving only the weights in the form of checkpoints and breaking the main code into separate modules, each with their own function. Outliers are another common problem in Artificial Intelligence. The model did encounter outliers in the data even when they were trained for 200 epochs, however, it was a simple task to get rid of this problem by increasing the epoch count by a factor of ten. This would not be a good idea for categorical models since it is known that this would probably cause overfitting. The good thing about generative models is that they are not bound by the same training constraints as categorical models such as overfitting. Technically speaking, overfitting takes place when the accuracy of the model with respect to the training dataset is much larger than that of the validation and testing datasets. It has been mentioned before that these models do not have validation or testing datasets, and thus there is no such a thing as overfitting in a generative context.

Nevertheless, there are analogous consequences to training the model with a substantial amount of epochs, and they are the possibilities that the model will simply unintentionally end up replicating the original dataset, which is intuitively analogous to overfitting, given that the model has become too good at predicting/generating the training data, but it is incompetent when it comes to generating an original track.

9. Conclusions

Based on what was learnt in this project, it is evident that deciding on using GRUs for the intended application was a good decision. LSTM, on the other hand, leaves much to be desired. It is speculated that the main reason that LSTM did not work as expected was that when (Skúli et.al, 2017) trained their model, they fed 92 songs all belonging to the Final Fantasy saga. On this project, on the other hand, the training data was restricted to one song. This was a result of experimentation with several songs created by Mozart. The result was unsatisfactory, as it was clear that Mozart did not constrain his music to a single style. This can be observed when comparing only two of his most famous compositions: "The Requiem in D minor, K. 626" and "The Piano Sonata No. 11 in A major, K. 331 / 300i", also known as *Rondo Alla Turca*. Both pieces were composed by Mozart, and yet both show very different tones, tempos, innuendo and emotions. Intuitively, the next logical step would be to feed data that is mostly similar so that the output can stick to a single style.

10. Future Work

Despite the success that was garnered through optimization of the GRU approach, there is always room for improvement. There were a few things that, given more time and resources, would be able to be improved upon. A list of future improvements include: GAN model development, batch song choice, genre variation, the ability to incorporate multiple instruments simultaneously, dual track training, and input data restriction based on style. First, the GAN model will be touched on. Although the GRU approach was novel in this application, there is still unlocked potential within a GAN approach. The current reference models in place lack the proper documentation to be reproduced, on top of the fact that the majority of the preprocessing data is stored in an inaccessible pickle file. This information holds the formatting that is required to be processed with the GAN approach in question. With the ability to reformat the input data that was used in both the GRU and LSTM approaches, the GAN approach would be fully accessible and would thereby exhaust the potential solutions.

Speaking of exhaustive potential, the model has currently operated under unary conditions. The single genre input style has served the model well, however, the ability to expand to other genres and recognize/implement patterns in music that are more intertwined with other instruments, would push the design to its limits and thereby offer a more useful product in the long run. The applications would then be limitless.

Dual track training is the next area of improvement. The GRU preprocesses the input data by operating under specific track selection, where both track 0 and track 1 correspond to the right and left hand melodies, respectively. Being able to incorporate both tracks/hands at the same time would theoretically allow the model to better encompass the input tracks/data. This leads to the potential for a more responsive model. It would be interesting to see if it could be expanded to support a whole orchestra.

Similarly, the ability to incorporate multiple instruments into its learned capabilities would better trin the holistic capabilities of the model. Once again, expanding the product potential and thereby, the market size.

More remarks can be discovered when comparing the GRU model to the original LSTM model by (Skúli et.al, 2017) is that both models are unable to distinguish beginnings and endings to pieces. As the network is now there is no distinction between pieces, that is to say the network does not know where one piece ends

and another one begins. This would allow the network to generate a piece from start to finish instead of ending the generated piece abruptly as it does now. Compared to their LSTM network, the GRU model is only able to handle one song at a time and a single hand only, so it would be convenient to modify the code to be able to parse through a folder with several songs and learn from every single track.

Finally, adding more instruments to the dataset would be ideal. As it is now, the network only supports pieces that only have a single instrument, in this case, the right hand playing the piano. It would be interesting to see if it could be expanded to support a whole orchestra.

11.Flowchart

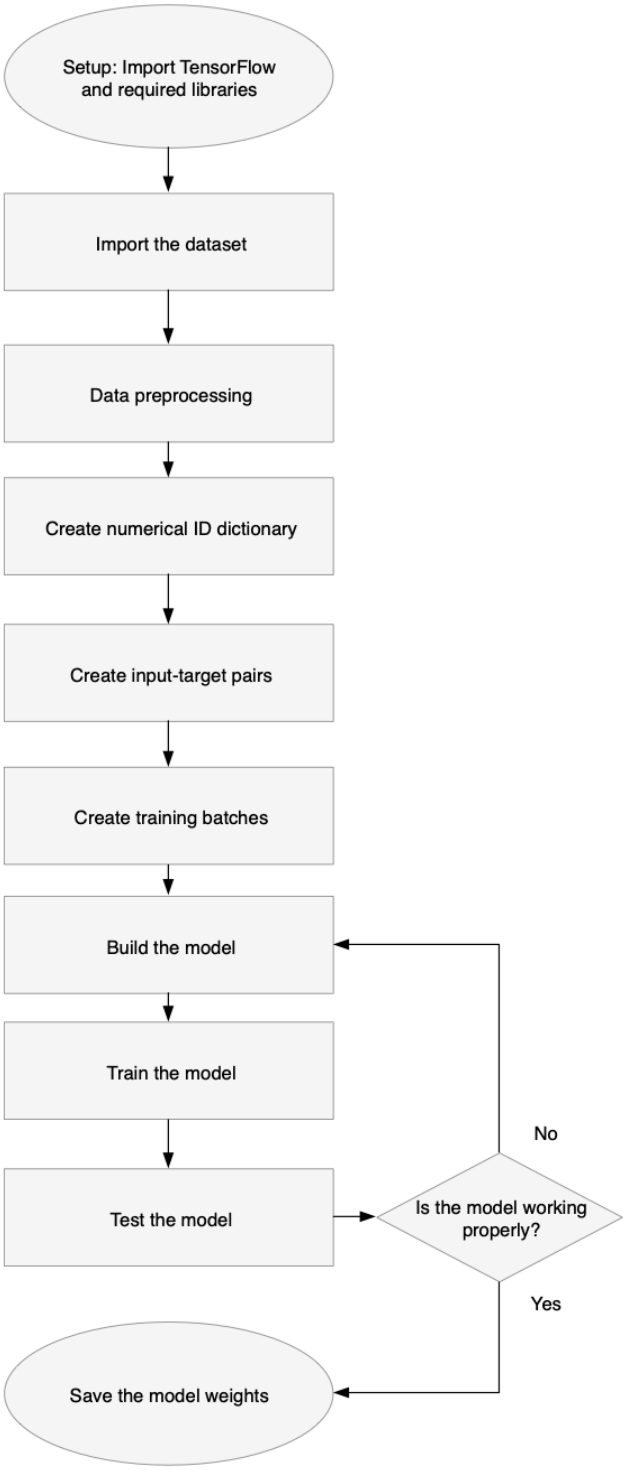

Fig. 14. Model Flowchart

12. References

[1]I–v–vi–iv progression. (2021, April 30). Retrieved May 03, 2021, from https://en.wikipedia.org/wiki/I%E2%80%93V%E2%80%93vi%E2%80%93IV_progression

[2]Madrigal, A. (2020, February 26). The hard drive with 68 Billion Melodies. Retrieved May 03, 2021, from https://www.theatlantic.com/technology/archive/2020/02/whats-the-point-of-writing-every-possible-melody/607120/

[3] Skúli, S. (2017, December 09). How to generate music using a LSTM neural network in Keras. Retrieved May 03, 2021, from https://towardsdatascience.com/how-to-generate-music-using-a-lstm-neural-network-in-keras-68786834d4c5

[4]Text generation with an rnn : Tensorflow core. (n.d.). Retrieved May 03, 2021, from https://www.tensorflow.org/tutorials/text/text_generation

[5]The origins and development of the logit model. (2003). *Logit Models from Economics and Other Fields,* 149-157. doi:10.1017/cbo9780511615412.010

10. Appendix

GRU Code

Main.py

```
import os
import time
import numpy as np
import tensorflow as tf
from Data_Retriever import midiString
from tensorflow.keras.layers.experimental import preprocessing
from mido import Message, MidiFile, MidiTrack

# Import Data
text = midiString

# Number of characters in the imported data
# print('Length of text: {} characters'.format(len(text)))

# First 250 characters in text
# print("First 250 characters:")
```

```
# print(text[:250])

# Number of unique characters in the file
vocab = sorted(set(text))
# print('There are {} unique characters in the text data'.format(len(vocab)))

# Create StringLookup layer to convert character into a numeric ID:
ids_from_chars = preprocessing.StringLookup(
   vocabulary=list(vocab))

# And layer to invert conversion
chars_from_ids = tf.keras.layers.experimental.preprocessing.StringLookup(
   vocabulary=ids_from_chars.get_vocabulary(), invert=True)


# Create a function that inverts the conversion and joins the characters
def text_from_ids(ids):
   return tf.strings.reduce_join(chars_from_ids(ids), axis=-1)


# Convert the text into a stream of numeric IDs split into tokens.
all_ids = ids_from_chars(tf.strings.unicode_split(text, 'UTF-8'))
# print("Character IDs: ")
# print(all_ids)
# <tf.Tensor: shape=(1115394,), dtype=int64, numpy=array([20, 49,
#  58, ..., 47, 10,  2])>

# Convert to dataset form
ids_dataset = tf.data.Dataset.from_tensor_slices(all_ids)
# for ids in ids_dataset.take(10):
#     print(chars_from_ids(ids).numpy().decode('utf-8'))
# <
# m
# e
# t
# a

# m
# e
```

```
# s
# s

# Sequence length
seq_length = 100

# Examples per epoch
examples_per_epoch = len(text)//(seq_length+1)

# convert individual characters to sequences of desired size
sequences = ids_dataset.batch(seq_length+1, drop_remainder=True)
# for seq in sequences.take(1):
#   print(chars_from_ids(seq))
# tf.Tensor(
# [b'<' b'm' b'e' b't' b'a' b' ' b'm' b'e' b's' b's' b'a' b'g' b'e' b' '
#  b't' b'r' b'a' b'c' b'k' b'_' b'n' b'a' b'm' b'e' b' ' b'n' b'a' b'm'
#  b'e' b'=' b"'" b'P' b'i' b'a' b'n' b'o' b'\\' b'x' b'0' b'0' b"'" b' '
#  b't' b'i' b'm' b'e' b'=' b'0' b'>' b'\n' b'<' b'm' b'e' b't' b'a' b' '
#  b'm' b'e' b's' b's' b'a' b'g' b'e' b' ' b't' b'i' b'm' b'e' b'_' b's'
#  b'i' b'g' b'n' b'a' b't' b'u' b'r' b'e' b' ' b'n' b'u' b'm' b'e' b'r'
#  b'a' b't' b'o' b'r' b'=' b'2' b' ' b'd' b'e' b'n' b'o' b'm' b'i' b'n'
#  b'a' b't' b'o'], shape=(101,), dtype=string)

# Convert to text for visualization
# for seq in sequences.take(5):
#     print(text_from_ids(seq).numpy())
# b"<meta message track_name name='Piano\\x00' time=0>\n<meta message
# time_signature numerator=2 denominato"
# b"r=8 clocks_per_click=24 notated_32nd_notes_per_beat=8 time=0>\n<meta
# message key_signature key='C' tim"
# b'e=0>\n<meta message set_tempo tempo=500000 time=0>\ncontrol_change
# channel=0 control=121 value=0 time=0'
# b'\nprogram_change channel=0 program=0 time=0\ncontrol_change channel=0
# control=7 value=100 time=0\ncontro'
# b'l_change channel=0 control=10 value=64 time=0\ncontrol_change channel=0
# control=91 value=30 time=0\ncon'


# Create function to create input/target pairs
```

```
def split_input_target(sequence):
    input_text = sequence[:-1]
    target_text = sequence[1:]
    return input_text, target_text


# Map function
dataset = sequences.map(split_input_target)

# for input_example, target_example in dataset.take(1):
#     print("Input :", text_from_ids(input_example).numpy())
#     print("Target:", text_from_ids(target_example).numpy())
# Input : b"<meta message track_name name='Piano\\x00' time=0>\n<meta message
#  time_signature numerator=2 denominat"
# Target: b"meta message track_name name='Piano\\x00' time=0>\n<meta message
# time_signature numerator=2 denominato"

# Batch size
BATCH_SIZE = 64

# Buffer size to shuffle the dataset
# (TF data is designed to work with possibly infinite sequences,
# so it doesn't attempt to shuffle the entire sequence in memory. Instead,
# it maintains a buffer in which it shuffles elements).
BUFFER_SIZE = 10000

# Create dataset
dataset = (
    dataset
    .shuffle(BUFFER_SIZE)
    .batch(BATCH_SIZE, drop_remainder=True)
    .prefetch(tf.data.experimental.AUTOTUNE))

# Length of the vocabulary in chars
vocab_size = len(vocab)

# The embedding dimension
embedding_dim = 256
```

```
# Number of RNN units
rnn_units = 1024  # Can probably tinker with rnn units as well



# Class for first model
class MyModel(tf.keras.Model):

   def __init__(self, vocab_size, embedding_dim, rnn_units):
       super().__init__(self)

       # Create input layer:
       # A trainable lookup table that will map each character-ID
       # to a vector with embedding_dim dimensions
       self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)

       # Create GRU layer:
       # A type of RNN with size units=rnn_units (You can
       # also use an LSTM layer here)
       self.gru = tf.keras.layers.GRU(rnn_units,
                                      return_sequences=True,
                                      return_state=True)

       # Create output layer:
       # Layer with vocab_size logit outputs.
       # logit is the logarithmic
       # of the odds for each character in the vocabulary
       self.dense = tf.keras.layers.Dense(vocab_size)

   # For each character the model looks up the embedding, runs the GRU
   # one timestep with the embedding as input, and applies the dense layer
   # to generate logits predicting the log-likelihood of the next character:
   def call(self, inputs, states=None, return_state=False, training=False):
       x = inputs
       x = self.embedding(x, training=training)
       if states is None:
           states = self.gru.get_initial_state(x)
       x, states = self.gru(x, initial_state=states, training=training)
       x = self.dense(x, training=training)
```

```
        if return_state:
            return x, states
        else:
            return x


# Left Off
# Instantiate model
model = MyModel(
    # Be sure the vocabulary size matches the `StringLookup` layers.
    vocab_size=len(ids_from_chars.get_vocabulary()),
    embedding_dim=embedding_dim,
    rnn_units=rnn_units)

# Try the model:
for input_example_batch, target_example_batch in dataset.take(1):
    example_batch_predictions = model(input_example_batch)
    # print("(batch_size, sequence_length, vocab_size):")
    # print(example_batch_predictions.shape)
    # (64, 100, 67)  # (batch_size, sequence_length, vocab_size)

# Model Summary
# model.summary()

# sampled_indices = tf.random.categorical(example_batch_predictions[0],
#                                         num_samples=1)
# sampled_indices = tf.squeeze(sampled_indices, axis=-1).numpy()

# This gives us, at each timestep, a prediction of the next character index:
# array([ 7, 57, 47, 49,  4, 11, 33, 22, 23, 45, 26,  4, 16, 40, 53, 32, 27,
#         6, 42, 11, 43, 50, 25, 13, 52, 37,  5,  3, 35, 50, 21, 18, 26, 55,
#        23, 30,  6, 49, 25, 52, 11, 45, 61,  6, 52, 42, 15, 57, 40, 31, 61,
#        18, 52, 18, 57, 15,  8, 17, 24, 34, 58, 57, 34, 50, 64, 53, 23, 52,
#        56, 26,  1, 63, 35, 35, 46, 57, 24, 35, 20, 49, 31, 15, 11, 52, 41,
#        20, 45, 44, 50, 48, 59, 60, 46,  3,  5, 48, 28,  4, 64, 57])


# Decode these to see the text predicted by this untrained model:
# print("Input:\n", text_from_ids(input_example_batch[0]).numpy())
```

```
# print()
# print("Next Char Predictions:\n", text_from_ids(sampled_indices).numpy())

loss = tf.losses.SparseCategoricalCrossentropy(from_logits=True)

example_batch_loss = loss(target_example_batch, example_batch_predictions)
mean_loss = example_batch_loss.numpy().mean()
# print("Prediction shape: ", example_batch_predictions.shape,
#       " # (batch_size, sequence_length, vocab_size)")
# print("Mean loss:        ", mean_loss)


# A newly initialized model shouldn't be too sure of itself, the output
# logits should all have similar magnitudes. To confirm this you can check
# that the exponential of the mean loss is approximately equal to
# the vocabulary size. A much higher loss means the model is sure of its
# wrong answers, and is badly initialized:

# print("Mean loss:" + str(tf.exp(mean_loss).numpy()))
# print("Vocabulary Size:" + str(vocab_size))

model.compile(optimizer='adam', loss=loss)

# Configure checkpoints
# Use a tf.keras.callbacks.ModelCheckpoint to
# ensure that checkpoints are saved during training:

# Directory where the checkpoints will be saved
checkpoint_dir = './training_checkpoints'
# Name of the checkpoint files
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt_{epoch}")

checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_prefix,
    save_weights_only=True)

# # Execute the training
# EPOCHS = 1
# print("\n" + "Training method 1" + "\n")
```

```
# history = model.fit(dataset, epochs=EPOCHS, callbacks=[checkpoint_callback])


# Each time you call the model you pass in some text and an internal state.
# The model returns a prediction for the next character and its new state.
# Pass the prediction and state back in to continue generating text.


# The following makes a single step prediction:
class OneStep(tf.keras.Model):
   # temperature parameter to generate more or less random predictions.
   def __init__(self, model, chars_from_ids, ids_from_chars, temperature=0.8):
       super().__init__()
       self.temperature = temperature
       self.model = model
       self.chars_from_ids = chars_from_ids
       self.ids_from_chars = ids_from_chars

   # Create a mask to prevent "" or "[UNK]" from being generated.
       skip_ids = self.ids_from_chars(['', '[UNK]'])[:, None]
       sparse_mask = tf.SparseTensor(
           # Put a -inf at each bad index.
           values=[-float('inf')]*len(skip_ids),
           indices=skip_ids,
           # Match the shape to the vocabulary
           dense_shape=[len(ids_from_chars.get_vocabulary())])
       self.prediction_mask = tf.sparse.to_dense(sparse_mask)

   @tf.function
   def generate_one_step(self, inputs, states=None):
       # Convert strings to token IDs.
       input_chars = tf.strings.unicode_split(inputs, 'UTF-8')
       input_ids = self.ids_from_chars(input_chars).to_tensor()

       # Run the model.
       # predicted_logits.shape is [batch, char, next_char_logits]
       predicted_logits, states = self.model(inputs=input_ids, states=states,
                                             return_state=True)
       # Only use the last prediction.
```

```
        predicted_logits = predicted_logits[:, -1, :]
        predicted_logits = predicted_logits/self.temperature
        # Apply the prediction mask:
        # prevent "" or "[UNK]" from being generated.
        predicted_logits = predicted_logits + self.prediction_mask

        # Sample the output logits to generate token IDs.
        predicted_ids = tf.random.categorical(predicted_logits, num_samples=1)
        predicted_ids = tf.squeeze(predicted_ids, axis=-1)

        # Convert from token ids to characters
        predicted_chars = self.chars_from_ids(predicted_ids)

        # Return the characters and model state.
        return predicted_chars, states


one_step_model = OneStep(model, chars_from_ids, ids_from_chars)
```

Data_Retriever.py

```
from mido import MidiFile

midi = MidiFile('/Users/PabloLoo1/Documents/Texas A&M University/Spring 2021/'
                + 'BMEN 489/Project 2/'
                + 'score.mid',
                #+ 'WA_Mozart_Marche_Turque_Turkish_March_fingered.mid',
                clip=True)

midiString = ''

# There are two tracks, which correspont to each hand
# playing the piano

# 2 tracks, so midi.tracks[0] or midi.tracks[1]
```

```
for i in range(0, len(midi.tracks[1])):
   # print(midi.tracks[0][i])
   midiString += str(midi.tracks[1][i]) + '\n'

print('\n')
print('Input:')
print(midiString[:3000])
```

Train_Model.py

```
from Main import *
from tensorflow.keras.callbacks import TensorBoard

# Create TensorBoard callback
tensorboard = tf.keras.callbacks.TensorBoard(log_dir="tf/logs")

# Execute the training
EPOCHS = 2000
print("\n" + "Training method 1" + "\n")
history = model.fit(dataset, epochs=EPOCHS,
                    callbacks=[checkpoint_callback, tensorboard])

# Save the model
# tf.saved_model.save(one_step_model, 'one_step')
model.save_weights('./checkpoints/my_weights')

# Magic code to view tensorboard
```

New_Track_Creator.py

```
import Main_LSTM
import tensorflow as tf
import os
import numpy
from mido import Message, MidiFile, MidiTrack
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
```

```
from keras.layers import LSTM
from keras.callbacks import ModelCheckpoint
from keras.utils import np_utils



# one_step_reloaded = tf.saved_model.load('one_step')
model.load_weights('./checkpoints/my_weights')

# Run the imported model
states = None
print("\n")
next_char = tf.constant(["control_change"])
result = [next_char]

# one_step_model

for n in range(50000):
   next_char, states = one_step_model.generate_one_step(next_char,
                                                        states=states)
   result.append(next_char)

new_data = tf.strings.join(result)[0].numpy().decode("utf-8")
# print("\n" * 2)
# print(new_data.splitlines())

# Create some music baby!

print("\n")



def track_append(new_data):
   mid = MidiFile()
   track = MidiTrack()
   mid.tracks.append(track)

   for i in new_data.splitlines():
       if i == new_data.splitlines()[-1]:
           i = new_data.splitlines()[0]
       str = "track.append(Message("
```

```
        for j in i.split():
            if j == i.split()[0]:
                j = "'" + j + "'"
            str += j + ','
        str = str[:-1]
        str += "))"
        print(str)
        try:
            eval(str)
        except Exception:
            pass
    mid.save('score.mid')


track_append(new_data)
```

Modified LSTM Code

Lstm.py

```
""" This module prepares midi file data and feeds it to the neural
    network for training """
import glob
import pickle
import numpy
from music21 import converter, instrument, note, 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 Activation
from keras.layers import BatchNormalization as BatchNorm
from keras.utils import np_utils
from keras.callbacks import ModelCheckpoint

def train_network():
    """ Train a Neural Network to generate music """
    notes = get_notes()

    # get amount of pitch names
```

```
    n_vocab = len(set(notes))

    network_input, network_output = prepare_sequences(notes, n_vocab)

    model = create_network(network_input, n_vocab)

    train(model, network_input, network_output)

def get_notes():
    """ Get all the notes and chords from the midi files in the ./midi_songs directory
"""
    notes = []

    for file in glob.glob("midi_songs/*.mid"):
        midi = converter.parse(file)

        print("Parsing %s" % file)

        notes_to_parse = None

        try: # file has instrument parts
            s2 = instrument.partitionByInstrument(midi)
            notes_to_parse = s2.parts[0].recurse()
        except: # file has notes in a flat structure
            notes_to_parse = midi.flat.notes

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

    with open('data/notes', 'wb') as filepath:
        pickle.dump(notes, filepath)

    return notes

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

```
    sequence_length = 100

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

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

    network_input = []
    network_output = []

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

    n_patterns = len(network_input)

    # reshape the input into a format compatible with LSTM layers
    network_input = numpy.reshape(network_input, (n_patterns, sequence_length, 1))
    # normalize input
    network_input = network_input / float(n_vocab)

    network_output = np_utils.to_categorical(network_output)

    return (network_input, network_output)

def create_network(network_input, n_vocab):
    """ create the structure of the neural network """
    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')

    return model

def train(model, network_input, network_output):
    """ train the neural network """
    filepath = "weights/weights-improvement-{epoch:02d}-{loss:.4f}-bigger.hdf5"
    checkpoint = ModelCheckpoint(
        filepath,
        monitor='loss',
        verbose=0,
        save_best_only=True,
        mode='min'
    )
    callbacks_list = [checkpoint]

    model.fit(network_input, network_output, epochs=2000, batch_size=128,
callbacks=callbacks_list)

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

Predict.py

```
""" This module generates notes for a midi file using the
    trained neural network """
import pickle
import numpy
import numpy as np
from random import randrange
```

```
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():
    """ Generate a piano midi file """
    #load the notes used to train the model
    with open('data/notes', 'rb') as filepath:
        notes = pickle.load(filepath)

    # Get all pitch names
    pitchnames = sorted(set(item for item in notes))
    # Get all pitch names
    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):
    """ Prepare the sequences used by the Neural Network """
    # map between notes and integers and back
    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)
```

```
    # reshape the input into a format compatible with LSTM layers
    normalized_input = numpy.reshape(network_input, (n_patterns, sequence_length, 1))
    # normalize input
    normalized_input = normalized_input / float(n_vocab)

    return (network_input, normalized_input)

def create_network(network_input, n_vocab):
    """ create the structure of the neural network """
    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')

    # Load the weights to each node
    model.load_weights('weights/weights-improvement-481-0.0004-bigger.hdf5')

    return model

def generate_notes(model, network_input, pitchnames, n_vocab):
    """ Generate notes from the neural network based on a sequence of notes """
    # pick a random sequence from the input as a starting point for the prediction
    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 = []

    # generate 500 notes
    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)

        i = randrange(1, 10)
        index = sorted(list(prediction[0])).index(sorted(list(prediction[0]))[-i])
        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):
    """ convert the output from the prediction to notes and create a midi file
        from the notes """
    offset = 0.5
    output_notes = []

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

```
            new_chord.offset = offset
            output_notes.append(new_chord)
        # pattern is a note
        else:
            new_note = note.Note(pattern)
            new_note.offset = offset
            new_note.storedInstrument = instrument.Piano()
            output_notes.append(new_note)

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

    midi_stream = stream.Stream(output_notes)

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

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