

**The Edward S. Rogers Sr. Department of
Electrical and Computer Engineering
University of Toronto**

ECE496Y Design Project Course

Group Final Report

Title: Music Generation With ML

Team #:

135

Team members:

Name:

Email:

Romal Peccia

romal.peccia@mail.utoronto.ca

Abhishek Paul

abhishek.paul@mail.utoronto.ca

Donald Lleshi

donald.lleshi@mail.utoronto.ca

Supervisor:

Jason Anderson

Section #:

6

Administrator:

John Taglione

Submission Date:

March 31, 2020

Group Final Report Attribution Table

Section	Student Names		
	Romal Peccia	Donald Lleshi	Abhishek Paul
Executive Summary	RD, MR		
Group Highlights	MR	RD, MR	
1.1/1.2 Introduction, Background/Goal	RS, RD, MR	RS	RS, RD, MR
1.3 Project Requirements	RS, RD, MR		MR
2.1 System-level overview	RS, RD, MR		
2.2 Module-level description	RS, RD, MR		
2.3 Module-level design	RS, RD, MR	RS, RD, MR	
2.4 Assessment of design		RS, RD, MR	RS, RD, MR
3.1 Verification table		RS, RD, MR	RS, RD, MR
3.2 Final Test Results	RS	RS, RD, MR	RS, RD, MR
4.0 Summary and Conclusion	RD		RD,MR
Appendix A: Gantt Chart History			RD, MR
Appendix B: Financial Plan	RS, RD, MR	RS, RD, MR	
Appendix C: Original Verification Table		RS, RD, MR	RS, RD, MR
Appendix D: Validation Results		RS, RD, MR	RS, RD, MR
All	FP, CM, ET	FP, CM, ET	FP, CM, ET

Abbreviation Codes:

Fill in abbreviations for roles for each of the required content elements. You do not have to fill in every cell. The “All” row refers to the complete report and should indicate who was responsible for the final compilation and final read through of the completed document.

RS – responsible for research of information

RD – wrote the first draft

MR – responsible for major revision

ET – edited for grammar, spelling, and expression

OR – other

“All” row abbreviations:

FP – final read through of complete document for flow and consistency

CM – responsible for compiling the elements into the complete document

OR - other




If you put OR (other) in a cell please put it in as OR1, OR2, etc. Explain briefly below the role referred to:

OR1: enter brief description here

OR2: enter brief description here

Signatures

By signing below, you verify that you have read the attribution table and agree that it accurately reflects your contribution to this document.

Name	Romal Peccia	Signature		Date:	31/02/2020
<hr/>					
Name	Donald Lleshi	Signature		Date:	31/02/2020
<hr/>					
Name	Abhishek Paul	Signature		Date:	31/02/2020
<hr/>					

Voluntary Document Release Consent Form

To all ECE496 students:

To better help future students, we would like to provide examples that are drawn from excerpts of past student reports. The examples will be used to illustrate general communication principles as well as how the document guidelines can be applied to a variety of categories of design projects (e.g. electronics, computer, software, networking, research).

Any material chosen for the examples will be altered so that all names are removed. In addition, where possible, much of the technical details will also be removed so that the structure or presentation style are highlighted rather than the original technical content. These examples will be made available to students on the course website, and in general may be accessible by the public. The original reports will not be released but will be accessible only to the course instructors and administrative staff.

Participation is completely voluntary and students may refuse to participate or may withdraw their permission at any time. Reports will only be used with the signed consent of all team members. Participating will have no influence on the grading of your work and there is no penalty for not taking part.

If your group agrees to take part, please have all members sign the bottom of this form. The original completed and signed form should be included in the hardcopies of the final report.

Sincerely,

Khoman Phang

Phil Anderson

ECE496Y Course Coordinators

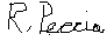


Consent Statement

We verify that we have read the above letter and are giving permission for the ECE496 course coordinator to use our reports as outlined above.

Team #: 135 Project Title: Music Generation With ML

Supervisor: Jason Anderson

Administrator: John Taglione

Name	Romal Peccia	Signature		Date:	31/02/2020
<hr/>		<hr/>			
Name	Donald Lleshi	Signature		Date:	31/02/2020
<hr/>		<hr/>			
Name	Abhishek Paul	Signature		Date:	31/02/2020
<hr/>		<hr/>			

EXECUTIVE SUMMARY (author: Romal Peccia)

One of the challenges faced by novice musicians is their inability to write complete songs. While many are capable of creating short melodies, they lack the necessary knowledge on musical structure to compose successive parts of their song. The motivation for this project lies in providing software that musicians could use to quickly prototype different songs, without needing any knowledge of music theory.

The goal of this project was to research the generation of music using neural networks and apply the results to create a music composition application. Using our application, a user is able to play a melody on a MIDI keyboard, and the application uses it to generate a continuation of that melody for the same length of time.

The first step in development was the curation of training data used to teach the neural network. A dataset of MIDI melodies was acquired, and a suite of transformation functions were made which lower the complexity of the data, making our task more viable. The transformed dataset is then transformed further into a format that can be passed through the neural network.

After data transformations, the neural network was developed. A Recurrent Neural Network (RNN) variant was used, which is commonly used for learning tasks involving sequential data. Variations of training code were developed, and the one that produced the best results was used to produce the final neural network model.

The final model is then integrated with a user interface and a physical MIDI keyboard. Users are prompted to play a melody, which is then transformed using the same transformations used on the dataset, and then fed through the final model to generate a continuation of the played notes. The user's recording and the model's generated continuation of their melody is then saved and displayed to them.

Overall, our team was successful in creating a music composition application using neural networks. We developed a useful framework for future development of neural network models for music generation. Much of our efforts was spent on variations of training code and data transformations which facilitate model learning, and the MIDI keyboard interface, however there is always more work that can be done on improving the models themselves. Our framework can be used in future development of models, with little code needed to be written, simply by using different datasets and alterations of model parameters. We think that the framework we developed is a powerful tool that can be used to improve our music composition application.

GROUP HIGHLIGHTS (author: Donald Lleshi, Romal Peccia)

Milestone 1: Data Collection/Representation¹

The team examined methods of music representation used to teach a neural network. Ultimately, Pianoroll (a representation of MIDI files as 2D arrays containing each note being played at each timestep) was selected to represent the data in a way that a model can learn from while meeting project requirements. A training dataset of MIDI files was acquired to train the model.

Milestone 2: Data Parsing/Transformation²

The team created parsing and transformation functions to simplify the learning task, reduce bias, and transform the MIDI dataset into encoded data that the Recurrent Neural Network could process. This was more challenging than expected, due to the list of transformations continually growing as we learned more about the project.

Milestone 3: Monophonic, 2-Octave Neural Network Models³

The team initially examined three types of neural networks (RNN, GAN, and CNN) to determine their viability. Ultimately, the RNN was determined to be the most feasible model that would facilitate model learning. An RNN model was created and trained continuously by varying the hyperparameters in order to obtain the best accuracy and minimize the training loss. Different iterations of training code and metrics were also developed and tested in order to verify and improve model performance.

Milestone 4: User Interactivity⁴

The framework and infrastructure necessary to interface a MIDI keyboard with our model was built in order to establish the overall system design. The MIDI events generated from the keyboard were interpreted to create an input MIDI file. This was processed using the transformation functions and passed into the model which generated an output. Finally, this output was converted back into a MIDI file and played back in the user interface.

¹ The acquired dataset: <https://drive.google.com/open?id=1w4qcJ0EeXg4r5Qt-LnYfWDJriNhs3x9W>

² Data transformation code: https://drive.google.com/open?id=1nN2G1TTjZhCrt3ZBgk6dwr8vjuJcajE_

³ Model Loading/Training code: https://drive.google.com/open?id=1Ko-CJPBI4RE9P3S8ExZz9R_9aoG2z2AM

⁴ User Interface code: https://drive.google.com/open?id=1y0T__Fb4Ih74NvfF75x_xbPY0F1yB-Qr

INDIVIDUAL CONTRIBUTIONS: ROMAL PECCIA

Milestone 1: Data Collection/Representation

I worked with the team to research the best way to represent music in a way that could be used to teach a neural network. Furthermore, I researched different datasets of MIDI music and chose one that best fit our needs, and wrote a script to parse and download such files.

Milestone 2: Data Parsing/Transformation

I was involved with writing or assisting in writing all of the data transformations. These functions required a careful understanding of how MIDI is represented as Pianoroll. This was where the bulk of our time was spent. Specifically, I wrote the functions to normalize tempo, split the dataset into smaller time intervals, and worked with the team to write the functions to convert the dataset from polyphonic to monophonic, remove data below the silence threshold and convert the Pianoroll inputs into one-hot encoding.

Milestone 3: Monophonic, 2-Octave Neural Network Models

I worked with Abhishek in researching and building the neural network, implementing different iterations of the training code, and testing functionality. I also wrote the functions to calculate overall accuracy, zero-accuracy, and non-zero accuracy, and used these to help diagnose issues in our training logic.

Milestone 4: User Interactivity

After Donald implemented code to interface and poll the MIDI keyboard, I worked with him on debugging as well as turning the Pygame MIDI events into an actual MIDI file. After this file was made, I adapted our transformation functions to transform the MIDI file, and pass it through the neural network. After the neural network's output was reconstructed back to MIDI, I wrote a function to improve the quality of the MIDI output.

INDIVIDUAL CONTRIBUTIONS: ABHISHEK PAUL

Milestone 1: Data Collection/Representation

Milestone 1 was a collaborative milestone where I helped research different music representations that we could feed to the neural network. I also researched different machine learning models that are used in music generation problems.

Milestone 2: Data Parsing/Transformation

Milestone 2 took the majority of our project as the entire data transformation infrastructure needed to be built from scratch. I was responsible for writing or assisting in most of the transformation functions. I created a function to look at the frequency of occurrence of the different notes in order to reduce the feature space. I worked with Romal in creating the function that converts the data from polyphonic to monophonic and changes the data to the 2-octave MIDI data. I also worked with the team to debug the problem of having mostly silent outputs by creating a silence threshold and doing further data manipulation on the dataset.

Milestone 3: Monophonic, 2-Octave Neural Network Models

I worked with Romal in researching and creating the Recurrent Neural Network (RNN) as we implemented different iterations of the training code. I was responsible for making the necessary transformation to the data in order to make it fit our training code. In the later stages of the project I was responsible for changing the parameters and training different iterations of the model in order to get the best music outputs. This involved setting up Google Cloud to run the training code on in order to speed up the training turn around.

Milestone 4: User Interactivity

Donald and Romal were mainly responsible for working on User Interactivity as I was responsible for training different model iterations. I however worked extensively in modifying the function which converted the model output back into MIDI.

INDIVIDUAL CONTRIBUTIONS: DONALD LLESHI

Milestone 1: Data Collection/Representation

My individual contributions to milestone 1 was to conduct background research on music generation models and to assist research on data representation and MIDI parsing libraries. This involved exploring alternatives and making design decisions on the data representation.

Milestone 2: Data Parsing/Transformation

My individual contributions to milestone 2 was to write parsing and transformation functions necessary to curate the data the neural network model will train and learn from. I worked on processing the MIDI songs split by intervals to generate pairs of the melody and the subsequent extension to the melody. Furthermore, I implemented the silence threshold functionality to exclude MIDI songs containing more than 50% silence in the training code.

Milestone 4: User Interactivity

In order to facilitate our overall system design, a user interface to connect the MIDI keyboard to our model was necessary. I researched different alternatives to interface the MIDI keyboard and ultimately decided that repurposing specific modules in the Python library Pygame mainly used for game development best satisfied our requirements. I implemented a framework that employs the Pygame library and built the infrastructure necessary to poll the keyboard and interpret the MIDI events. Similarly, I worked on building the infrastructure to transform these raw MIDI Note On and Note Off events polled from the keyboard to reconstruct a viable MIDI file.

Furthermore, I worked on building the functions on the output end of the system design. This involved interpreting the output the model generates and converting the generated Pianoroll output into a viable MIDI output file. Furthermore, in order to establish the pipeline from the model to the user interface, the Pygame library was utilized to create the functionality that automatically plays back the output MIDI file through the user interface.

ACKNOWLEDGEMENTS (author: Donald Lleshi)

We would like to express our sincerest gratitude to our supervisor Professor Jason Anderson and our administrator John Taglione for helping us overcome the immense challenges with this project and offering creative solutions and motivation to overcome our obstacles. The weekly progress meetings with Professor Anderson kept us on track and ensured that we were never overwhelmed with the project workload. Professor Anderson guided us through each step of the design process. He was invested into our project and recognized the unique challenges and obstacles that required unique solutions. He encouraged us to think of creative solutions, to always explore alternatives and to maintain a laser focus on creating a viable product. Similarly, our administrator John Taglione was also invested into our project and helped us design a better product through the use of carefully orchestrated project requirements. He ensured that our project was a well defined engineering product that established clear requirements of success. He encouraged us to test thoroughly and clearly define the criteria for successful test cases. The feedback and direction we obtained from both our supervisor and our administrator was invaluable and ultimately helped us design and implement a better product.

We would also like to thank the owner of the “Free Piano Tutorials” Youtube channel for providing publically available MIDI files which were used to train our neural network models.

TABLE OF CONTENTS

	REPORT ATTRIBUTION TABLE	1
	RELEASE CONSENT FORM	3
	EXECUTIVE SUMMARY	5
	GROUP HIGHLIGHTS AND INDIVIDUAL CONTRIBUTIONS	6
	ACKNOWLEDGEMENTS	10
	TABLE OF CONTENTS	11
1.	INTRODUCTION	12
	1.1 BACKGROUND AND MOTIVATION	12
	1.2 PROJECT GOAL	13
	1.3 PROJECT REQUIREMENTS	14
2.	FINAL DESIGN	17
	2.1 SYSTEM LEVEL OVERVIEW/DIAGRAMS	17
	2.2 MODULE LEVEL DESCRIPTION	18
	2.3 MODULE LEVEL DESIGNS	21
	2.4 ASSESSMENT OF FINAL DESIGN	35
3.	TESTING AND VERIFICATION	
	3.1 VALIDATION AND ACCEPTANCE TESTS	37
	3.2 FINAL TEST RESULTS	41
4.	CONCLUSION	48
	 APPENDIX A: GANTT CHART HISTORY	 50
	APPENDIX B: FINANCIAL PLAN	52
	APPENDIX C: ORIGINAL VALIDATION AND ACCEPTANCE TESTS	54
	APPENDIX D: VALIDATION AND ACCEPTANCE TEST RESULTS	57
	REFERENCES	62

INTRODUCTION

This report summarizes the motivation, requirements, design and testing of our project “Generating Music With Machine Learning”, as part of our final year design project course ECE 496. This report walks through the initial research conducted in this area, the data manipulation and model training steps involved, analysis of the results of the model, and how the user interface works. This report concludes with potential applications of our project and future work.

1.1 BACKGROUND AND MOTIVATION

Recent advances in machine learning have seen it proliferate across multiple industries. One frontier that has still to be fully breached by AI is art. Music is an art form where an underlying musical structure is present [1]. This raises the question, is it possible for a machine learning model to learn to create such musical structure?

While work on using machine learning for music generation has been ongoing since the 1980s, it has traditionally focussed on randomly generating music from a trained set of musical compositions without any user input [2] . Although this method has its uses, it does not allow for collaboration between a musician and the program. Current state of the art programs for music generation like Google Magenta [3] have aimed at the more ambitious task of generating music based on user input. During the course of this project, Amazon released their own machine learning enabled music generator: Amazon DeepComposer [5]. Its functionality is similar to our project goal as it can generate accompaniments to music played on a two-octave keyboard.

The motivation for this project came from one team member, who is a casual musician. He is skilled enough to write simple melodies, but lacks the theoretical knowledge to write subsequent parts of a song that are based on the original melody. The goal of this project is to create a machine learning model that can take an input melody (the lead of the song, such as what would be played by the right hand on a piano), and generate a subsequent extension of the melody for the same length of time.

This aims to be beneficial to both aspiring and experienced composers. It will allow experienced musicians a chance to experiment with varieties of inputs during composing sessions, so they can quickly experiment with different melodies. It will also allow aspiring musicians who can play a simple melody to generate an extension of their melody without needing any understanding of music theory. While machine learning in music is primarily used for market research and recommendation customization, the prospect of music generation opens up many other possibilities [4]. Being able to quickly generate melody extensions has the potential to revolutionize the music industry, as it will allow a greater variety of melodies to be tested during the creative process.

Prior to starting this project one team member had already conducted research into this topic. In his project, he was able to generate qualitatively reasonable accompaniments (the supporting music of the song, such as what would be played by the left hand on a piano) based on input melodies. Inputs were in MIDI (Musical Instrument Digital Interface) format, which were then encoded into discrete sets of strings that represented relevant musical information. These inputs were then passed into a GRU (Gated Recurrent Unit) neural network. However there were some flaws, such as outputs being the wrong duration or occasionally getting stuck playing a single note, and a lack of proper metrics. How to represent the MIDI files, how to pass them into the network, which type of network and its parameters to use, and how to measure and compare the performance of models are all topics which will be explored in this research project.

1.2 PROJECT GOAL

The goal of this project was to research the generation of music using neural networks and apply the results to create a music composition application.

1.3 PROJECT REQUIREMENTS

ID	Project Requirement	Description
1.0	Input File Type: MIDI	Constraint: MIDI contains more easily interpretable data than other file formats, and can be separated into melody and accompaniment. It is also easily accessible for users via MIDI instruments or transcribing software.
2.0 2.1	MIDI Note Range: 2 Octaves (sets of 12 pitches) MIDI Note Range: 7.25 Octaves	Constraint: MIDI data contains a range of notes indexed from 0-128. The minimum range our inputs and outputs will use is 48-73, the smallest possible range for a MIDI piano. [6] Objective: Our inputs and outputs should ideally use the full range of MIDI piano notes, 21-108.
3.0 3.1	Input Type: Monophonic Input Type: Polyphonic	Constraint: Inputs and outputs of the model will be monophonic (at most one note playing at any given time step). Objective: Inputs and outputs of the model will be polyphonic (multiple notes played at any given time step). This increases the complexity of the model and its ability to learn.
4.0	Input Representation: Pianoroll	Functional Requirement: MIDI data must be converted into a form that a neural network can learn from. Pianoroll (a representation of MIDI files as 2D arrays containing each note being played at each timestep), will be used.

Table 1: Project Requirements

ID	Project Requirement	Description
5.0	Model Architectures <ul style="list-style-type: none"> - GANs (Generative Adversarial Network) - RNNs (Recurrent Neural Network) - CNN (Convolutional Neural Network) Autoencoder 	<p>Functional Requirement: We will investigate the use of generative RNNs for the purpose of music generation. We will experiment with different RNN configurations in order to get the best results.</p> <p>Objective: Two other neural network architectures will be investigated and compared, GANs and CNNs. GANs are commonly used for generative tasks, and CNNs with autoencoders are used for image generation tasks which can be applied to the Pianoroll representation</p>
6.0	Training Period: 24 hours	<p>Constraint: The training time of a model must be no more than 24 hours so that there is enough time to try different models and tune hyperparameters. Training time is directly correlated to model complexity, metric calculation and the size of the dataset used.</p>
7.0	Consistent duration of input and output: 5 ± 0.2 seconds	<p>Functional Requirement: The outputs generated by the model and the input melodies fed into the model should both be 5 seconds.</p>
7.1	Consistent duration of input and output: 10 ± 0.2 seconds	<p>Objective: The outputs generated by the model and the input melodies fed into the model should both be 10 seconds.</p>
8.0	Training Metrics (comparing generated music to target music) <ul style="list-style-type: none"> - Zero Accuracy - Non-Zero Accuracy - Overall Accuracy (Note: Zero and Non-Zero Accuracy are explained in Section 2.3.1: Metrics)	<p>Functional Requirement:</p> <ul style="list-style-type: none"> - Zero Accuracy will be at least 66% - Non-Zero Accuracy will be at least 66% - Overall accuracy will be at least 66% <p>Objective:</p> <ul style="list-style-type: none"> - Zero Accuracy should be at least 90% - Non-Zero accuracy should be at least 66% - Overall accuracy should be at least 70%

Table 1: Project Requirements (continued)

ID	Project Requirement	Description
9.0	Composition Metrics (comparing user input and output to statistics found in the dataset) <ul style="list-style-type: none"> - Number of leaps - Number of dissonances/consonances - Average rest time 	Objective: These metrics are used to help the user understand how far their input and output deviates from “real” music. They will improve the user experience, but will not be used to evaluate the overall performance of the model due to them greatly increasing the training period. Their implementations are not required for the overall model evaluation.
10.0	Output File Type: MIDI	Functional Requirement: Outputs of the model must be converted back into MIDI format.
11.0	User Interface	Sub Functional Requirement: Users should be able to connect a MIDI instrument to a computer using our software and immediately generate MIDI outputs after they record a melody.
11.1	Immediate Processing	
12.0	Standardized tempo (assigns a duration per timestep)	Sub-Functional Requirement: All input and output Pianorolls should have their tempo set to the same amount (the default tempo of 120 has been chosen).
13.0	Standardized velocity (volume)	Functional Requirement: All input and output MIDI files must have their velocity set to the same amount (the default velocity of 100 has been chosen).
14.0	Silence threshold	Sub-Functional Requirement: All input Pianorolls should contain no more than 50% timesteps that contain no notes.

Table 1: Project Requirements (continued)

2 FINAL DESIGN

2.1 SYSTEM LEVEL OVERVIEW AND DIAGRAM

In order to have a neural network generate music, it must be trained using music data.

System 1: Model Training

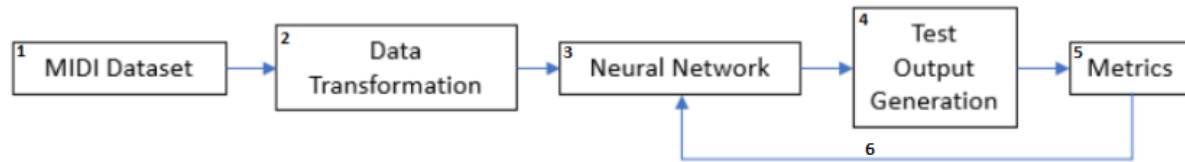


Figure 1: Model Training system flow

1. A training dataset of MIDI music was acquired.
2. This data is then transformed to meet project requirements, and then further transformed into a format that the network can learn from.
3. This transformed dataset is then passed through the network.
4. Outputs are then generated based on testing conditions. These outputs are used to adjust the weights of the network.
5. Metrics are calculated using the testing outputs to track performance of the network.
6. The model iterates through weight adjustments, output generation, and metric calculation until satisfactory metrics are achieved.

System 2: User-Facing Product

Once the model is trained sufficiently, it can be used to generate music for users.



Figure 2: User-Facing Product system flow

1. A user can connect their MIDI instrument to their computer.
2. The user interface will prompt the user to play a melody, and record what they played.
3. This melody is transformed using the same methods from step 2 of System 1.
4. The transformed data passes through the network and an output is generated.
5. The network output is converted back to MIDI and sent to the user interface for listening.

2.2 MODULE-LEVEL DESCRIPTIONS

2.2.1 SYSTEM 1: MODEL TRAINING

1. MIDI Dataset
Inputs: None
Outputs: MIDI dataset
Function: Training data is required for a neural network to learn to generate new outputs. MIDI data represented as Pianoroll was chosen because other formats break some of the project requirements, as outlined in the project proposal.

2. Data Transformation
Inputs: MIDI dataset
Outputs: - Transformed MIDI dataset - One-hot encoded pairs of melody and melody extension
Function: The acquired MIDI dataset is loaded into Pianoroll format, and transformations are applied to meet project requirements. Further transformations are made to convert the Pianorolls into a format that can pass through the neural network.

3. Neural Network
Inputs:- Neural network, with initial weights and parameters - One-hot encoded pairs of melody and melody extension - Loss
Outputs: - Generated one-hot encoded melodies under training conditions - Weight updates of the neural network module
Function: Encoded melodies and extensions are passed through the network, the outputs are compared to the extensions, and updates to the weights will propagate backwards through the network based to minimize the loss.

4. Test Output Generation
Inputs: <ul style="list-style-type: none"> - Neural network - Melody Pianoroll dataset (without extension)
Outputs: Generated Pianoroll melodies under testing conditions Accuracy
Function: The outputs generated during training are influenced by the input melody and the target data. These outputs are used to help the model learn faster. To calculate metrics based on testing conditions (how a user would experience the model), new outputs must be generated which are only influenced by the melody.

5. Metrics
Inputs: Outputs from the test output generation module
Outputs: - Accuracy <ul style="list-style-type: none"> - Non-zero accuracy - Zero accuracy
Function: Metrics will be computed by comparing the test outputs to the targets. These are used to track how well the model is performing as it iterates through the training process.

2.2.2 SYSTEM 2: USER FACING PRODUCT

1. MIDI Instrument
Inputs: MIDI piano
Outputs: None
Function: Users can plug in a MIDI keyboard to a computer so they can record their melody.

2. User Interface
Inputs: <ul style="list-style-type: none"> - Computer - MIDI to USB interface cable - Key presses by the user
Outputs: <ul style="list-style-type: none"> - Signal for user to start recording - MIDI file and playback audio of recorded melody
Function: The user interface prompts the user to start recording their melody. The user presses keys for 5 seconds, and a MIDI file of the recording is saved.

3. Data Transformation
Inputs: Recorded melody MIDI file
Outputs: Transformed melody MIDI file to One-hot encoded melody
Function: Recorded MIDI data is transformed using the same methods used in the model training system. The transformed MIDI is saved for future use and played through the user interface so the user can hear their melody.

4. Trained Neural Network
Inputs: - Final trained model <ul style="list-style-type: none"> - Encoded melody
Outputs: Encoded output data
Function: The encoded melody passes through the trained model and an encoded output is generated. This output is further transformed to improve the quality of the output melody.

5. Conversion Back to MIDI
Inputs: Neural Network output
Outputs: MIDI audio file of generated melody.
Function: The output of the neural network is decoded back into MIDI format further post-processing is applied to improve music quality, and the MIDI is passed back to the user interface for the user to listen to and save.

2.3 MODULE LEVEL DESIGN

2.3.1 SYSTEM 1: MODEL TRAINING

1. MIDI Dataset (System 1)

The team examined methods of music representation including representing MIDI as a set of strings, MIDI represented as Pianoroll (a representation of MIDI files as 2D arrays containing each note being played at each timestep), or utilizing WAV data (data byte stream of the raw audio waveform data). Ultimately, Pianoroll was selected to represent the data, such that the model can learn from, while meeting project requirements as discussed in the project proposal.

386 MIDI (pop and classical music) songs ranging from 2-4 minutes were acquired. A web application⁵ was used to scrape video descriptions from the “Free Piano Tutorials” (FPT) Youtube channel⁶. This application outputs an excel spreadsheet. A trivial Python script was then written to parse and download the relevant MIDI files from the video descriptions in the spreadsheet.

One design decision was which MIDI dataset to use. The FPT data was considered over other typical MIDI datasets such as the Lahk MIDI Dataset (LMD) [7] due to its simple, two-track structure. FPT data contains 2 MIDI tracks, 1 for melody, and 1 for accompaniment. The LMD data contains 5 MIDI tracks, one for percussion, and 4 for other instruments. The melody could be contained in any of the 4 tracks, and is not distinguishable without human intervention for each MIDI file.

⁵ <http://www.williamsportwebdeveloper.com/FavBackUp.aspx>

⁶ <https://www.youtube.com/channel/UCBIE6pjsULp5gA7vl8-4F4A>

2. Data Transformation (System 1)

Many transformations were made to simplify the data to make it easier for the model to learn, as well as to transform it in a way that can pass through the neural network. The transformations are as follows:

Transformation 1: Normalizing Tempo

Each timestep in each Pianoroll represents time based on a tempo value. Since tempo varies from song to song (and sometimes within a single song), varying time steps can represent varying amounts of real time. All MIDI files had their tempos changed to 120 beats per minute (the default for Pianoroll) for their entire duration. This was done to ensure that real time was represented by the same number of timesteps for all songs. Although this means speeding up or slowing down songs, the melody is still preserved.

Transformation 2: Polyphonic to Monophonic Conversion

In order to simplify our model's learning, we wanted to use monophonic data (at most one note playing at any given time) rather than polyphonic data (zero or more notes played at any given time). By doing this, the amount of combinations of notes the model needs to learn is reduced to 88 from 2^{88} .

To teach our model to generate monophonic data, it needs to learn from examples of monophonic data. We could not find any monophonic MIDI songs, so we had to create our own from the polyphonic MIDI that we acquired. Our function reads a MIDI into Pianoroll and removes (set to zero) all but one random note from the set of notes played at each time step, and then saves the resulting Pianoroll as MIDI.

A problem with randomly sampling a note at each timestep occurred when a set of notes is held for multiple consecutive timesteps (which is usually the case). Each consecutive timestep had a different randomly chosen note from the set, resulting in choppy-sounding and unrealistic music. This was solved by tracking which notes are randomly chosen. When a note is randomly chosen, it gets saved, and if the next timestep contains the exact same set of notes as the previous timestep, the saved note gets chosen instead of randomly choosing it.

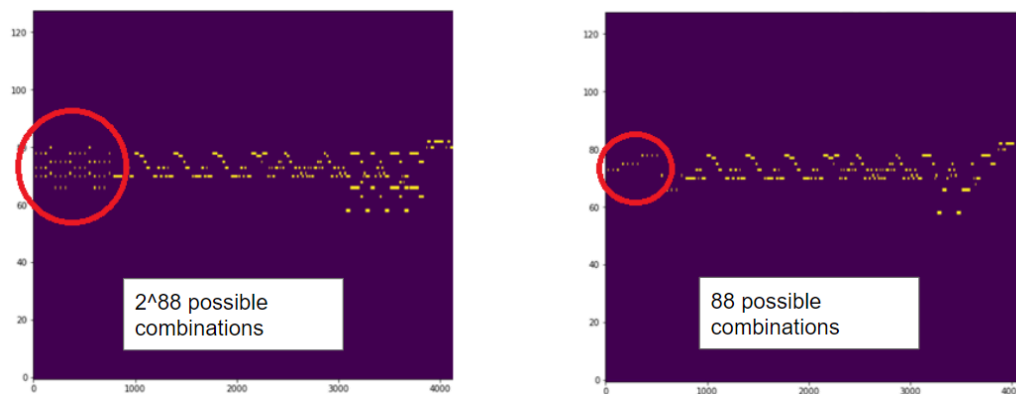


Figure 3: Examples of polyphonic data(left) and monophonic data(right) and their feature spaces

Transformation 3: Splitting Songs into 5 second intervals

The dataset containing 386 complete MIDI songs was converted into a new dataset containing 5 second MIDI samples. For each MIDI file, the following steps happen:

1. The Mido Python library is used to find the song length in real time.
2. The Pypianoroll library is used to load the MIDI as a Pianoroll object, containing a 2D array. One axis represents the number of timesteps. The other axis represents which notes are being played.
3. The number of timesteps is divided by the length of the song (in seconds) to determine the amount of timesteps per second. The timesteps/second are multiplied by 5 to determine how many timesteps represent 5 seconds.
4. The length of the song (in seconds) is divided by 5 to determine how many samples will be created.

5. Each segment is generated by sweeping through the original MIDI file and copying over a segment of it, with the length (in timesteps) of the segment being the timesteps per 5 seconds value calculated in (3). The copy is written to a new Pianoroll object and that is written to a MIDI file.

Using the process above, the original 386 MIDI files were segmented into 6708 files. The Mido library was used to verify the lengths of each output.

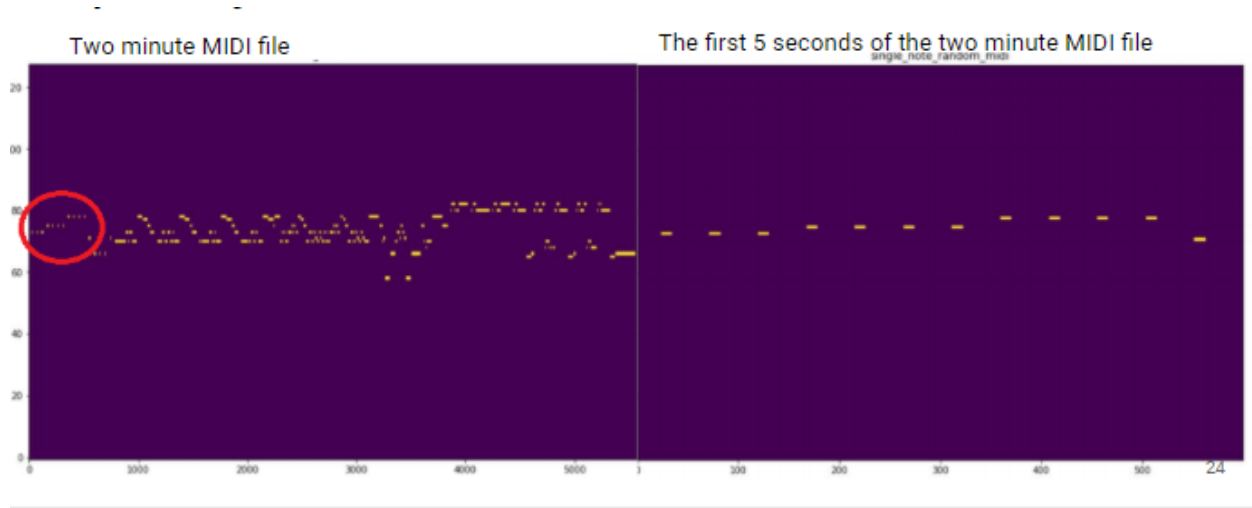


Figure 4: The first 5 seconds of a MIDI file split from an original 2 minute MIDI file

Transformation 4: Reducing the Range of Notes

A further simplification of the model learning was the reduction of range of possible notes in our data. 25 notes were chosen because it is the smallest size of commercially used MIDI keyboards. MIDI uses 128 notes, and we chose to use the range of notes 48 - 73 because it was the most commonly occurring range of notes in our dataset.

In order to reduce our note range, we created a function to inspect each timestep of each MIDI file and remove any notes that were outside of range 48-73. This ended up causing bias towards silence in our model's learning, as we introduced a lot of timesteps with no notes. We rectified

this by instead shifting each note that was outside the range up or down by 12 (representing an octave) until it was inside the range. Although the pitch of the notes are changed, they are still mostly functionally the same from a music theory perspective. There are slight exceptions to this, but it would require a more sophisticated algorithm using extensive music theory knowledge in order to solve them.

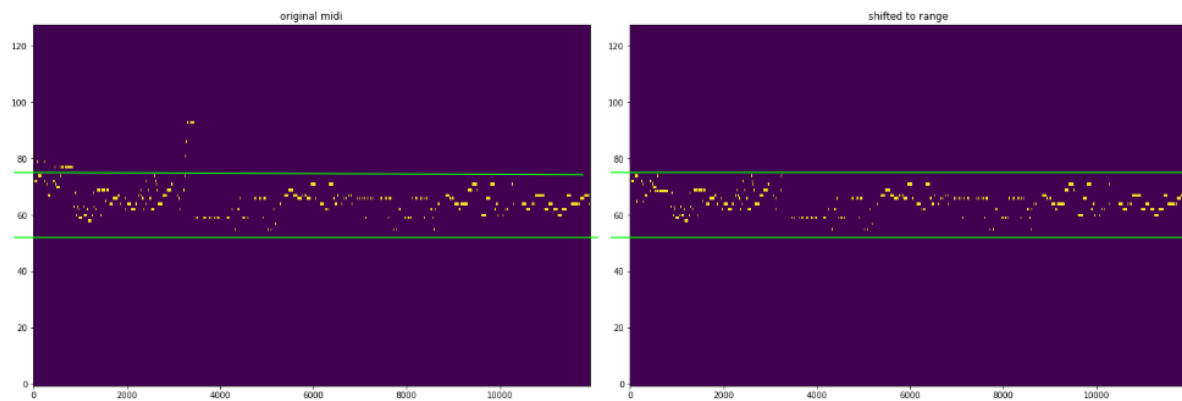


Figure 5: Example of a MIDI file before notes are shifted (left) and a MIDI file after its notes have been shifted (right)

Transformation 5: Pairing Melodies and Extensions

In order to teach the model to learn, it looks at initial melodies (5 seconds of music) and their extensions (the next 5 seconds of music). The model generates an output based on the initial melody, and compares what it generated to the real extension. It would therefore be helpful to have these pairs of 5 second MIDI files in a data structure that we could easily pass through.

This was accomplished by creating a data structure that pairs each melody segment with its associated extension segment. The collection of 5 second segments was initially ordered and grouped together based on the song the collection of segments belonged to. Subsequently, for each song, the collection of segments was iterated through and dynamically paired the melody and its associated extension into the data structure.

Transformation 6: Silence Threshold

Our initial tests of the models ended up having a bias toward generating silence. This was because after segmenting our dataset, many of the 5-second MIDIIs contained mostly silence. We reduced the bias by doing the following:

1. For each melody/extension pair, check the number of timesteps that contain no notes (we refer to these timesteps as “zeros”) in both the melody and the extension
2. If either the melody or the extension contain more than 50% zeros, discard the pair.

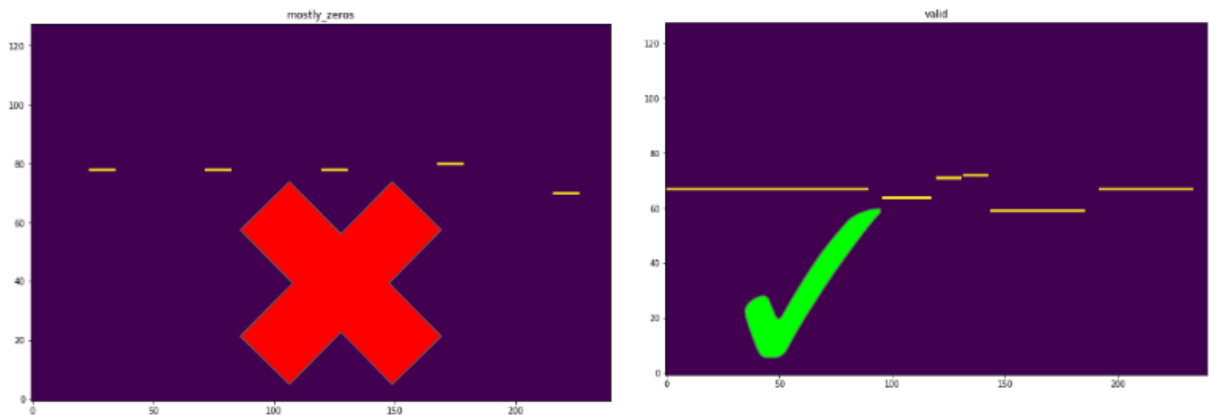


Figure 6: Example of MIDI files that break the silence threshold (left) and do meet it (right)

Transformation 7: One-Hot Encoding

The RNN model expects a one-hot encoding of the inputs to be passed in. This means that it expects an array of arrays, with one dimension representing time and the other representing which combination of notes is being played. The arrays at each timestep have one value being 1, and the rest 0. Given that we converted our data to be monophonic, the Pianoroll representations are already close to being in one-hot encoding. The next steps were:

1. Remove all indexes from the note dimension that were outside of the range 48-73.
2. Add an index to represent no notes being played
3. Set all velocities (volumes) of each note to equal 1.

Each Pianoroll was therefore transformed into a 26 by N array (with N being the number of timesteps), where index 0 on the note axis represents no notes being played, and index 1-25 represents notes 48-73.

3. Neural Network and 4. Test Output Generation (System 1)

Note: Part 3 and 4 were combined because they both influenced the iterations of our training code.

A GRU network (Gated Recurrent Unit, a sophisticated type of RNN) model was chosen to learn how to generate music. RNN variants are typically used for generation tasks involving sequential data. This network and the basic training code was built using the Pytorch library (one of Python's neural network frameworks). Our training code went through many iterations before one was settled on. In all iterations, training follows the same process:

1. Input data is forward passed through the neural network
2. Output data is generated.
3. The output data is compared to the extension corresponding to the input data
4. Loss and accuracy are calculated
5. Updates are backward passed to the weights of the neural network based on the loss.
6. Steps 1-5 are repeated for every melody-extension pair in the dataset. (This is known as an epoch).
7. Steps 1-6 are repeated until satisfactory metrics are achieved.

What changed between training code iterations is how we represent our inputs, and how we calculate the metrics based on outputs.

Training was particularly challenging for our generation task because we are generating based on an input melody, and attempting to recreate an extension. In contrast, simpler generation tasks the team has seen in the past only ever involved recreating the training data, with no influence from any inputs.

Training Code Iteration 1:

The input data was a concatenation of the initial melody and melody extension, representing 10 seconds of music. The output also has the same duration, but the first half is discarded, as the second half is what corresponds to the melody extension. This output is then used to calculate loss and accuracy. Loss is used to update the weights of the network, and accuracy is used for us to quantify how well our model is learning.

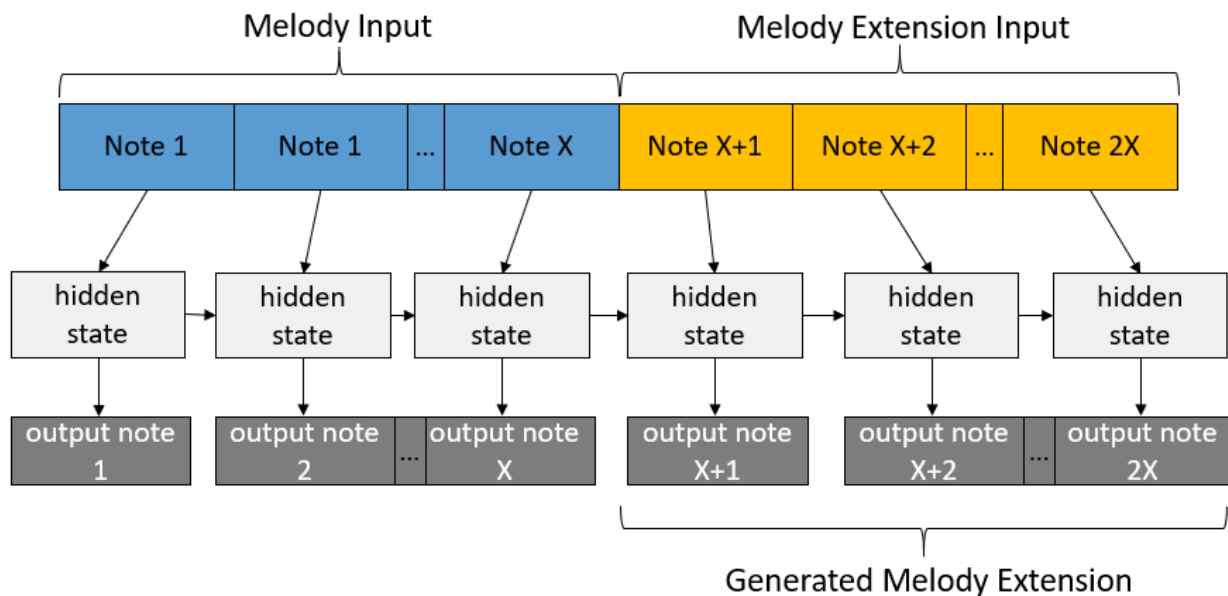
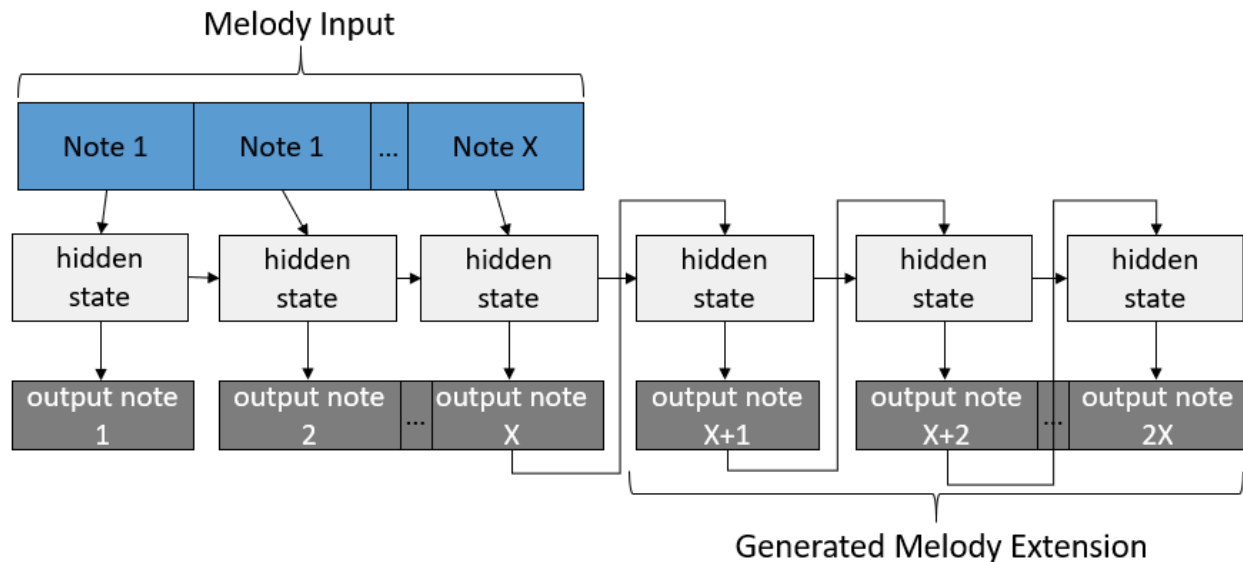


Figure 7: Generation of Melody Extensions using training conditions

When we trained our model, we saw that accuracy reached almost 60%. However, when melody extensions were generated during testing, the results were qualitatively poor. We realized that the accuracy during training was not indicative of what was actually happening during testing.

During testing, there is no melody extension for the model to learn from. The model only has the input melody, and therefore less information to work with. The input is passed through the model, returning a hidden state of the RNN representing that input. Notes are then individually generated and passed through the model and updating the hidden state, one at a time, until a sequence of notes corresponding to 5 seconds is generated. This variation in generation during testing versus training is what caused our accuracy to be inaccurate.



Training Code Iteration 2:

In this iteration we continued using the same inputs and outputs to calculate the loss. However, a second forward pass is done using only the melody as input. The output of this forward pass is then used to generate a test output as shown in Figure 8.

When the model was trained using the updated training code, accuracy became almost 10%. Although accuracy is significantly worse, it was an accurate reflection of how our model was actually performing. We realized that our model was performing poorly because the training code was trying to minimize the loss of one type of output, while calculating accuracy for another type of output.

Training Code Iteration 3:

In order to rectify the issues of iterations 1 and 2, outputs used to calculate loss and accuracy should be the same, and the outputs should be based on testing conditions. We generated these outputs as shown in Figure 8.

In this iteration, accuracy reached about 70% after about 800 epochs.

5. Metrics (System 1)

Accuracy is calculated by comparing the notes at each timestep in the generated outputs to the notes at each timestep in the real extensions. As the model gets better at generating real extensions, the accuracy improved. In early trials of model training, using a small dataset, the model was able to achieve 90% accuracy. However, when listening to the outputs, they were completely silent.

In order to diagnose this problem, we created two new accuracy metrics: zero accuracy, and non-zero accuracy. Zero accuracy measures the accuracy of timesteps that contain no notes (a zero), while non-zero accuracy measures the accuracy of timesteps that do not contain a zero. This initial model was reaching 100% zero accuracy, and 0% non-zero accuracy, meaning that it was predicting that every timestep in every output had no notes played, and that 90% of our dataset contained zeros.

We rectified this problem by adding the silence threshold (data transformation 6), and shifting notes out of range into the range instead of discarding them (data transformation 4). Our resulting dataset contained less than 10% zeros, and our final iteration of training code reached a non-zero accuracy of 68.74% and zero accuracy of 94.62%, with overall accuracy of 76.0272%.

2.3.2 SYSTEM 2: USER FACING PRODUCT

1. MIDI Instrument (System 2)

A MIDI keyboard connects to a computer using a USB A to USB B cable. This keyboard includes at least 2 octaves with note range 48-73 available to be played. The smallest MIDI keyboards can play a range of up to 25 notes, while larger keyboards will contain undesired notes out of our range. While the user can potentially play undesired notes, or play polyphonic music, our transformations of the user's inputs prevent them from passing in data that our model is not expecting.

2. User Interface (System 2)

The user interface is displayed through the command line using Python. This interface prompts the user to play their melody, and displays to them what they played as well as what the model generated for them. In order to construct the user interface, a framework was built to interface the MIDI keyboard with our Python code, interpret the MIDI events and construct a MIDI file representing the recorded melody.

2.1 Interfacing the MIDI Keyboard

In order to interface the MIDI keyboard with the user interface, a framework to read and interpret the MIDI events was created. Since there is no native library in Python to interface with a MIDI keyboard, specific modules in the Python Pygame library (typically used for game development) were repurposed to interface with the MIDI keyboard.

Each MIDI instrument connected to the computer is represented with a unique ID in Pygame. After initializing the Pygame MIDI modules using the keyboard's ID, the MIDI keyboard is polled for 5 seconds. During this 5 second duration, if the user presses a key on the MIDI keyboard, the keyboard will send MIDI events to the interface containing the MIDI status, note played, the volume of the note and the timestamp of the note. When a note on the keyboard is pressed, a MIDI Note On event (status 144) is sent to the interface. When the key pressed is released, a MIDI Note off event (status 128) is sent to the user interface. During the 5 second polling, the MIDI events are parsed to extract the necessary information.

2.2 Constructing the input MIDI file

The Mido Python library was utilized to construct a MIDI file representing the recorded melody. Initially, a MIDI track object was initialized by setting the track number, tempo and time signature. Subsequently, the data structure of saved pairs of MIDI Note On and Off events was utilized to extract the MIDI information. The extracted information was used to create MIDI message objects for each event with the specified status, note and velocity/volume. Furthermore, the timing in MIDI files is represented as ticks (smallest unit of time in MIDI), where each beat (quarter note) is divided into ticks. Each MIDI message object requires a delta time attribute (in ticks) that represents how many ticks have passed since the last MIDI message object.

In order to place each of these MIDI message objects(Note On and Off events) at the correct time slot on the MIDI track, the delta time between each MIDI message object represented as ticks had to be calculated. The Mido library default ticks per quarter note is set to 480 ticks/quarter note. The default microseconds per quarter note is set to 500 000 for the standard tempo of 120 beats per minute (bpm). Therefore, the microseconds per tick is calculated as $\text{microseconds/tick} = (\text{microseconds/quarter note}) / (\text{ticks/quarter note}) = 500\,000/480$. Finally, to calculate the time in ticks, the following formula was utilized: $\text{ticks} = (\text{time in microseconds} / \text{microseconds per tick})$. Furthermore, in the case that the user stopped playing notes on the keyboard before the end of the 5 second polling duration, the ticks for the last MIDI Note off message was extended to reach the 5 second duration. As a result of the above processing steps, a MIDI file representing the recorded melody was constructed and saved. This MIDI file was then used as the input file that is further processed by the data transformation steps before passing through the model.

3. Data Transformation (System 2)

Once the user's melody is saved as a MIDI file, it undergoes some of the same transformations as done during training. Specifically, the polyphonic timesteps are converted to monophonic (transformation 2) and the data is converted into one-hot encoded form (transformation 7). Furthermore, during recording, if a note played falls outside the note range 48-73, it is ignored (which implicitly applies transformation 4). The transformed MIDI is saved to disk and played back for the user, so they can see the discrepancies from what they played caused by the monophonic conversion and reduction in note range.

4. Trained Neural Network (System 2)

The user's encoded melody is passed through the model using the test generation approach as shown in Figure 8. An output array is returned that must be converted back to MIDI.

5. Conversion Back to MIDI (System 2)

Once the model generates an output, the data transformation steps are undone in order to generate a melody extension MIDI file. The 26-index model output array is extended to a 128-index array, and indexes 1-25 are shifted to indexes 48-73. Index 0 is set to 0 as it represents silence (no note being played). This array is turned into a Pianoroll track object with a standardized tempo of 120 bpm and velocity of 100, which is converted into a MIDI file and saved to the directory. The melody extension is then automatically played back to the user through the interface utilizing Pygame modules.

When we listened to some of the generated melodies, we noticed that many of them had a rapid changing of notes in consecutive timesteps. Each timestep represents approximately 0.02 seconds, so the rapid change of notes in such short periods lead to unrealistic and unappealing sounding music. In order to rectify this, post processing of the model outputs was implemented

as follows:

1. Iterate through the model output array until a non-zero is seen.
2. Continue iterating through the array for N more timesteps, replacing the values in the array with the non-zero value seen before.
3. Continue steps 1 and 2 until the entire array has been iterated through.

In short, every note played in the generated extension is held for N timesteps, where N can be chosen by the user depending on how granular they want their music to be.

2.4 ASSESSMENT OF FINAL DESIGN

Overall, our team was able to build an end-to-end system that meets our goal of researching the generation of music using neural networks and applying the results to create a music composition application.

2.4.1 Assessment of the Model

The final model was trained on a dataset of 2500 5-second melody/extension pairs for about 800 epochs. The accuracy of the model at generating melody extensions from the training set and from a validation set (data the model was not trained on) is shown in Table 2. The testing method is outlined in Appendix D.

Accuracy Type	Training	Validation
Zero	94.62 %	74.3572 %
Non-Zero	68.74294 %	45.6182 %
Overall	76.0272 %	48.6963 %

Table 2: Final Model Results

The model was able to learn from the data set as the overall training accuracy was 76.0272%. To show that the model could generalize to new music inputs, we presented it with validation data and the overall accuracy was 48.6963%. These values show that the model is able to effectively learn from its training data and to a certain extent generalize what it learned to new data.

2.4.2 Assessment of the User Interface

The user interface enables an end to end system that prompts users through the command line to play a music segment for a 5 second duration on the connected MIDI keyboard and establishes a pipeline to process the input through the model and play the results back to the user. This fulfills the requirements of connecting a MIDI instrument, feeding an input file type of MIDI into the model, generating a MIDI output extension and immediate processing. The user interface is displayed through the command line using Python. This interface counts down for a 3 second

interval before prompting the user to play their melody on the keyboard. Subsequently, the MIDI events are read from the keyboard, and displayed back on the terminal. MIDI input file and output file are at a consistent duration of 5 seconds \pm 0.2 seconds and played back to the user. An output is generated in approximately 0.39 seconds and played back to the user fulfilling the immediate processing requirement of less than 1 second.

```
3
2
1
0
Play
Reading input for 5 seconds
instruction note MIDI number velocity timestamp
144 b 59 100 548
128 b 59 64 655
144 f 65 100 910
128 f 65 64 1045
144 c 60 100 1287
128 c 60 64 1401
144 d 62 100 1662
128 d 62 64 1753
144 e 64 100 2118
128 e 64 64 2283
144 d# 63 100 2552
128 d# 63 64 2667
144 c# 61 100 2876
128 c# 61 64 2982
144 a# 58 100 3245
128 a# 58 64 3370
144 a 57 100 3627
128 a 57 64 3773
144 c 60 100 4001
128 c 60 64 4104
144 d 62 100 4398
144 e 64 100 4409
128 e 64 64 4463
128 d 62 64 4479
id in db
Processing time: 0.3875401020050049
initial: 4.986458333333333
test: 4.9818181818181815
converted: 5.002272727272727
final: 5.002272727272727
What you played
What was generated
```

Figure 9: Sample terminal output of end to end process

Refer to the below link for demos of the end to end system with different inputs played on the keyboard: https://drive.google.com/open?id=1QKmIJpIqUDwyGN9xReh0dCHvuJXI_VeA

3.0 TESTING AND VERIFICATION

3.1 TESTING AND VERIFICATION

Id	Project Requirement	Verification Method	Verification Result and Proof
1.0	Input File Type: MIDI	REVIEW OF DESIGN: The model must take as input a MIDI file with file extension “.midi”. Any other file type as input is considered not valid. MIDI reading Python libraries must be able to read this MIDI.	PASS: Refer to Section 3.2
2.0	MIDI Note Range: 48-73	<p>TEST 1: The MIDI input file will be examined and the note range measured must fall between range 48-73. Any MIDI with notes outside of this range is considered not valid.</p> <p>TEST 2: The MIDI output file generated by the model will be examined and the note range measured must fall between range 48-73. Any MIDI with notes outside of this range is considered not valid.</p>	<p>TEST 1: PASS. Refer to Section 3.2</p> <p>TEST 2: PASS. Refer to Section 3.2</p>
2.1	Objective: MIDI Note Range: 21-108	<p>TEST 1: The MIDI input files will be examined and the note range measured must fall between range 21-108 (piano full range). Any MIDI with notes outside of this range is considered not valid.</p> <p>TEST 2: The MIDI output files generated by the model will be examined and the note range measured must fall between range 21-108 (piano full range). Any MIDI with notes outside this range is considered not valid</p>	<p>TEST 1: Passed but not implemented in the final design</p> <p>TEST 2: Passed but not implemented in the final design.</p>

Table 3: Requirements, Testing, and Verification

Id	Project Requirement	Verification Method	Verification Result and Proof
3.0	Input Type: Monophonic	TEST: Monophonic: For each MIDI input and output file, iterate through each timestep. At each timestep, the number of notes being played must be one or zero.	Monophonic: PASS: Refer to Section 3.2
3.1	Objective: Input Type: Polyphonic	TEST: Polyphonic: For each MIDI input and output file, iterate through each timestep. Across all timesteps, there should be instances of more than one note being played at least once.	Polyphonic: NOT IMPLEMENTED
4.0	Input Representations: Pianoroll	REVIEW OF DESIGN: The MIDI input file will be parsed into Pianoroll representation. This representation must be a 2D array containing each note in MIDI note range 48-73 (minimum range) or 21-108 (full piano range) being played at each timestep.	PASS: Refer to Section 3.2
5.0	Model Architectures i) RNNs (Recurrent Neural Networks)	TEST: Each of the examined model architecture's validity will be measured using overfitting. A small dataset will be fed into each of the models to be overfitted and reproduce a known result. Each model must produce 100% accuracy with the overfitted data to be considered a valid model during this testing procedure.	RNN: PASS: Refer to Section 3.2
5.1	Objective: ii) CNNs(Convolutional Neural Networks) iii) GANs (Generative Adversarial iii) Networks)		CNN: Implemented (by other team member) but not tested GAN: NOT IMPLEMENTED

Table 3: Requirements, Testing, and Verification (continued)

Id	Project Requirement	Verification Method	Verification Result and Proof
6.0	Training Period: 24 hours	TEST: A model will stop its training period if it exceeds the maximum 24 hours constraint set in the requirements.	NOT MET. Refer to Section 3.2
7.0	Consistent duration of input/output: 5 ± 0.2 seconds	TEST: The duration of the input MIDI is measured and compared to the duration of the output MIDI. The durations of input and output must be equal to the specified amount (5 or 10 ± 0.2 seconds) .	Consistent Input: PASS: Refer to Section 3.2.
7.1	Objective: Consistent duration of input/output: 10 ± 0.2 seconds		Consistent Output: PASS. Refer to Section 3.2 Objective: NOT IMPLEMENTED
8.0	Training Metrics (comparing generated music to target music) <ul style="list-style-type: none"> - Zero Accuracy - Non-Zero Accuracy - Overall Accuracy 	TEST: Test accuracy of the model will be evaluated on these metrics: <ul style="list-style-type: none"> - Zero accuracy will be at least 66% - Non-Zero accuracy will be at least 66% - Overall accuracy will be at least 66% 	PASS: Refer to Section 3.2
9.0	Objective: Composition Metrics (comparing user input and output to statistics found in the dataset)	REVIEW OF DESIGN: The metrics as outlined in the project proposal should be employed to enhance the music composition experience of the user. If the composition metrics are not employed, this objective is not met.	NOT IMPLEMENTED: This objective was meant to enhance the user experience, but was not required for functionality. Due to the extensive research and development time needed to implement this objective, it was ignored in favor of improving other project features.

Table 3: Requirements, Testing, and Verification (continued)

Id	Project Requirement	Verification Method	Verification Result and Proof
10.0	Output File Type: MIDI	REVIEW OF DESIGN: The model must output a MIDI file with file extension “.midi”. Any other file type generated as an output is considered not valid. MIDI reading Python libraries must be able to read this MIDI.	PASS: Refer to Section 3.2.
11.0	User Interface	REVIEW OF DESIGN: A MIDI instrument should connect and interface with the software.	PASS. Refer to Section 3.2
11.1	Immediate Processing	TEST: A MIDI output file should be generated less than 1 second after the time the input melody is finished playing.	PASS Refer to Section 3.2
12	Standardized tempo	TEST: For each MIDI input, the Mido Python library will be used to check if the tempo value matches the standard value (120).	PASS: Refer to Section 3.2.
13	Standardized velocity	TEST: For each MIDI input, the velocity of each note will be checked to see if it matches the standard velocity of 100.	PASS: Refer to Section 3.2.
14	Silence threshold	TEST: For each MIDI input, the number of timesteps with no notes being played will be compared to the total number of timesteps, and should not exceed 50%.	PASS Refer to Section 3.2

Table 3: Requirements, Testing, and Verification (continued)

3.2 FINAL TEST RESULTS

ID 1.0 : Input File Type: MIDI

Testing & Verification, Final Results:

The input files fed into the model during training are loaded using the Pypianoroll library. Pypianoroll would return an error if any of these MIDI files were incorrectly loaded.

The input file fed into the model in the user interface is a MIDI file created using a MIDI compatible keyboard and the Mido python library. The user interface would return an error if the MIDI file was incorrectly created. Refer to Appendix D for the recorded input MIDI file.

ID 2.0 : MIDI Note Range 48-73

Testing & Verification, Final Results:

Since the RNN expects 26 possible values (its vocabulary size), the model training would throw an error if a one-hot-encoding of more or less than 26 values was passed instead. A for loop then ran through the MIDI files to verify that the 26 values are within the note range.

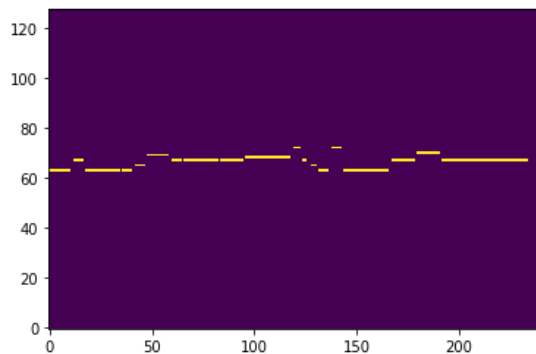


Figure 10 : In the Pianoroll above all notes are within the 43-78 range

ID 3.0: Input Type (Monophonic)

Testing & Verification, Final Results:

The monophonic nature was verified by iterating over the timesteps of each MIDI and checking whether there was more than one note that was non-zero.

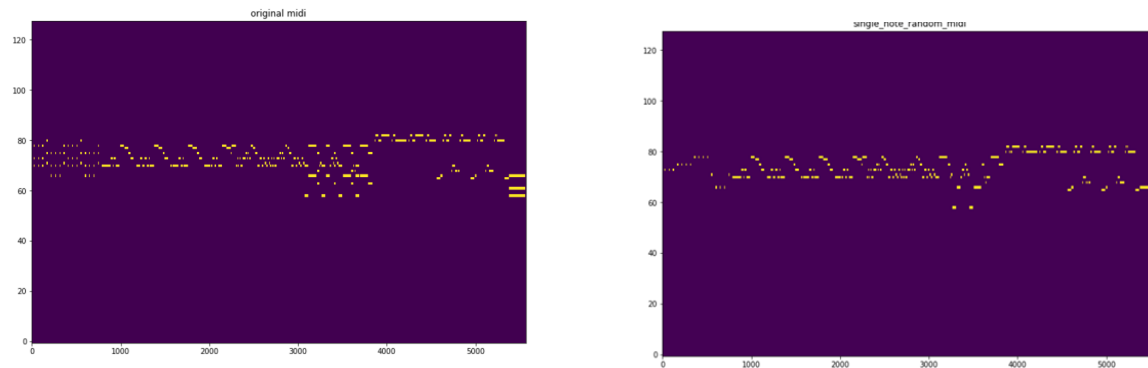


Figure 11 : In the figure above we have a polyphonic (left) converted to a monophonic (right)

ID 4.0: Input Representation (Pianoroll)

Testing & Verification, Final Results:

The dataset is loaded using Pypianoroll library functions. Our model will explicitly only take pianoroll inputs, since if the input representation is not pianoroll, Pypianoroll will flag an error.

ID 5.0 : Model Architectures: RNN

ID 5.1 (Objective): Model Architectures: CNN, GAN

Testing & Verification, Final Results:

The RNN model was trained with a small subset of data (100, 5 second samples), in order to test capability of overfitting. In the overfitted sample, the loss tended to 0 and the accuracy reached 100%. During regular training (4000, 5 second samples) accuracy generally improved and loss decreased each subsequent epoch, which showed that the model was learning.

The CNN model was split off as a separate subtask for Sadman Ahmed

The GAN was not implemented.

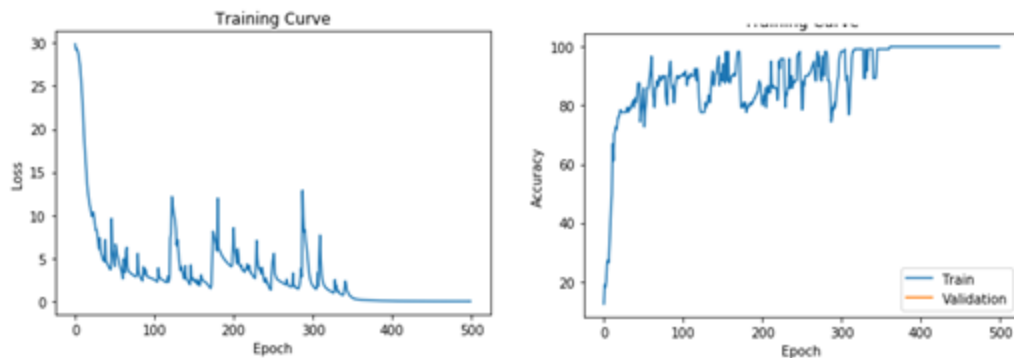


Figure 12: The figure above shows the loss (left) and accuracy (right) curves of the overfitted RNN

ID 6.0 : Training Period of 24 hours

Testing & Verification, Final Results:

Since we are no longer investigating 3 types of models, we decided to focus on training one model for a longer period of time. The 24 hour requirement no longer was necessary.

Rather than training many RNN models with different parameters for 24 hours, we chose to train one RNN model for a longer period. This way we could end up with one well-trained model, instead of many poorly-trained models.

The model was able to be trained for 110 epochs per 24 hours [Appendix D]. The final model was trained for 800 epochs, which took approximately 200 hours.

ID 7.0 : Consistent duration of input/output: 5 ± 0.2 seconds

Testing & Verification, Final Results:

After the MIDI events were interpreted from the keyboard and dynamically converted into a MIDI file, the converted duration of the input file was 5.002 seconds. After the model process the input file the output duration was 5.002 seconds. Refer to appendix D for the measurements of the input/output duration.

ID 8.0 : Training Metrics (comparing generated music to target music)

- Zero Accuracy
- Non-Zero Accuracy
- Overall Accuracy

Testing & Verification, Final Results:

The zero, non-zero and overall accuracy was verified by comparing the target music segment to the generated music segment and then calculating how accurately the model could predict the notes being played and when no notes were being played. In our training it surpasses the 66% set in the requirements

```
750 798.8420611132765 s loss: 2976.7159226208955 accuracy 74.22878315556335
zero accuracy 96.16960043983119 nonzero accuracy 66.17080079226982
751 810.8980757764774 s loss: 2949.4601503517356 accuracy 74.89380923742004
zero accuracy 96.05157954248901 nonzero accuracy 66.92960434438014
752 811.3463377987538 s loss: 2920.285188077907 accuracy 75.48058926861283
zero accuracy 95.76277097003138 nonzero accuracy 67.81085052515554
753 829.9664488080609 s loss: 2909.308751395322 accuracy 76.02720043880967
zero accuracy 96.15248024972121 nonzero accuracy 68.74294709001431
```

Figure 13: Zero, Non-Zero and Overall Accuracy as they reach the values set in the requirements

ID 10.0 : Output File Type: MIDI

Testing & Verification, Final Results:

The output from the model is converted into a MIDI file using Pypianoroll functions and then played back to the user. The Pypianoroll function should output a MIDI. If it is unsuccessful in converting from Pianoroll to MIDI it will return an error. Furthermore, the Pygame module that plays back the output MIDI will return an error if it is unable to read it.

ID 11.0 : User Interface

Testing & Verification, Final Results:

The framework and infrastructure built to poll and read MIDI events from the keyboard shown below in the terminal output reads and interprets MIDI note on and MIDI note off events. These events were then utilized to dynamically create an input MIDI file to be processed by the model.

```
Reading input for 5 seconds
instruction note MIDI number velocity timestamp
144 c 60 100 906
128 c 60 64 1014
144 d# 63 100 1205
128 d# 63 64 1308
144 b 71 100 1561
128 b 71 64 1698
144 f# 66 100 1914
128 f# 66 64 2034
144 c 60 100 2268
144 d 62 100 2280
128 d 62 64 2295
128 c 60 64 2350
144 b 59 100 2625
128 b 59 64 2720
144 c# 61 100 2945
128 c# 61 64 3073
144 d# 63 100 3281
128 d# 63 64 3377
144 f# 66 100 3611
128 f# 66 64 3704
144 e 64 100 3992
128 e 64 64 4047
```

Figure 14: Polling MIDI Note On (status 144) events and MIDI Note Off (status 128) events from the MIDI keyboard

ID 11.1 : Immediate Processing

Testing & Verification, Final Results:

A MIDI output file is generated 0.39 seconds (less than 1 second) after the time the input melody is finished playing to the time all the data transformation steps and model processing steps finish. Refer to Appendix D for the measurements.

ID 12.0 : Standardized tempo

Testing & Verification, Final Results:

The tempo of the input MIDI music file is set to 120 beats per minute (bpm). This is verified through a sampler function that parses the generated MIDI files and checks that the tempos of the different music segments were at the standard tempo of 120 bpm. Refer to appendix D.

ID 13.0 : Standardized velocity

Testing & Verification, Final Results:

A standardized velocity of 100 was used for the input MIDI file. This was verified through a sampler function that goes through the generated MIDI files and checked that the velocity values of the different music segments were the same. Refer to Appendix D.

ID 14.0 : Silence threshold

Testing & Verification, Final Results:

A function that iterates through the input data and discards all music segments that are silent for more than 50% of its timesteps was employed in order to satisfy the silence threshold requirement.

```
pair 2620 extension silence threshold 0.5083333333333333
pair 2620 input silence threshold 0.5083333333333333
pair 2632 dont match size 240 216
pair 2633 dont match size 216 240
pair 2649 extension silence threshold 0.55
pair 2649 input silence threshold 0.55
pair 2654 extension silence threshold 0.5458333333333333
pair 2654 input silence threshold 0.5458333333333333
pair 2655 dont match size 240 216
pair 2656 dont match size 216 240
pair 2671 extension silence threshold 0.5791666666666667
pair 2671 input silence threshold 0.5791666666666667
pair 2674 extension silence threshold 0.5166666666666667
pair 2674 input silence threshold 0.5166666666666667
pair 2698 extension silence threshold 0.6166666666666667
pair 2698 input silence threshold 0.6166666666666667
pair 2698 input silence threshold 0.6708333333333333
pair 2704 extension silence threshold 0.5333333333333333
pair 2704 input silence threshold 0.5333333333333333
pair 2704 input silence threshold 0.6125
pair 2712 extension silence threshold 0.5708333333333333
pair 2712 input silence threshold 0.5708333333333333
pair 2712 extension silence threshold 0.9083333333333333
```

Figure 15 : A sample of segments that were removed the percentage of the segment that is silent

4.0 CONCLUSION (author: Abhishek Paul, Romal Peccia)

Our team was successful in meeting our goal of creating a music composition application using neural networks. Based on our testing and verification results, it is clear that we have met the project requirements of being able to generate the extensions of monophonic, 2-octave, 5-second music segments. The target accuracy of 66% was exceeded for both the zero accuracy and non-zero accuracy which proved that the RNN model was able to learn from our dataset. However, the objectives of increasing the complexity of our input data (7-octave, polyphonic models), and trying different types of model architectures were abandoned. Model training time was a limiting factor, and many of the data transformations that reduced the complexity of our data were required in order to reduce training time. Furthermore, these data transformations ended up requiring an unexpected amount of development time, due to our inexperience in working with music data. Despite our models not using the complex data outlined in our objectives, we are content with our progress. The fact that Amazon DeepComposer[5] was only recently able to create a 2-octave, polyphonic model validates how difficult this task really is.

Our team also designed a user interface which prompts users to play a 5 second music segment on a keyboard and then automatically pipelines the entire process so it can generate an extension and play it back to the user, fulfilling our UI and immediate processing requirements. Our application is a useful tool for musicians who can play a simple melody, but need assistance in extending the melody into a full piece of music. It is designed so that users can easily try out different melodies without needing to know how the model works. Using machine learning generated extensions based on input music provides aspiring or experienced composers an innovative and quick way to experiment with a variety of different melodies during their creative process.

Furthermore, the data transformation and user interface framework we created can be a powerful tool for developers who wish to further research music generation using neural networks. These developers can swap in different datasets or models as they wish, without needing to develop, alter, or understand the framework that we built. Models could be specialized in certain styles

simply by changing the dataset they learn from. For example, a model could specialize in music of a certain key, genre, or artist. Furthermore, a team with much greater resources than ours could also use our framework to train different model architectures with more complex layers, as well as use larger datasets, and longer training periods. Many possibilities have been opened up by our framework, as long as there are resources to use the framework with.

The key conclusion to be drawn from our project is that it is indeed possible to create RNN models that are able to learn and generate realistic extensions as both validation and training accuracies generally improve each epoch. Although there is still room for significant improvement in the performance of the models, we hope that the framework we developed can help future developers be even more successful in generating music extensions with neural networks.

APPENDIX A : GANTT CHART HISTORY

Below is the Gantt Chart that the team designed for the Project Proposal.

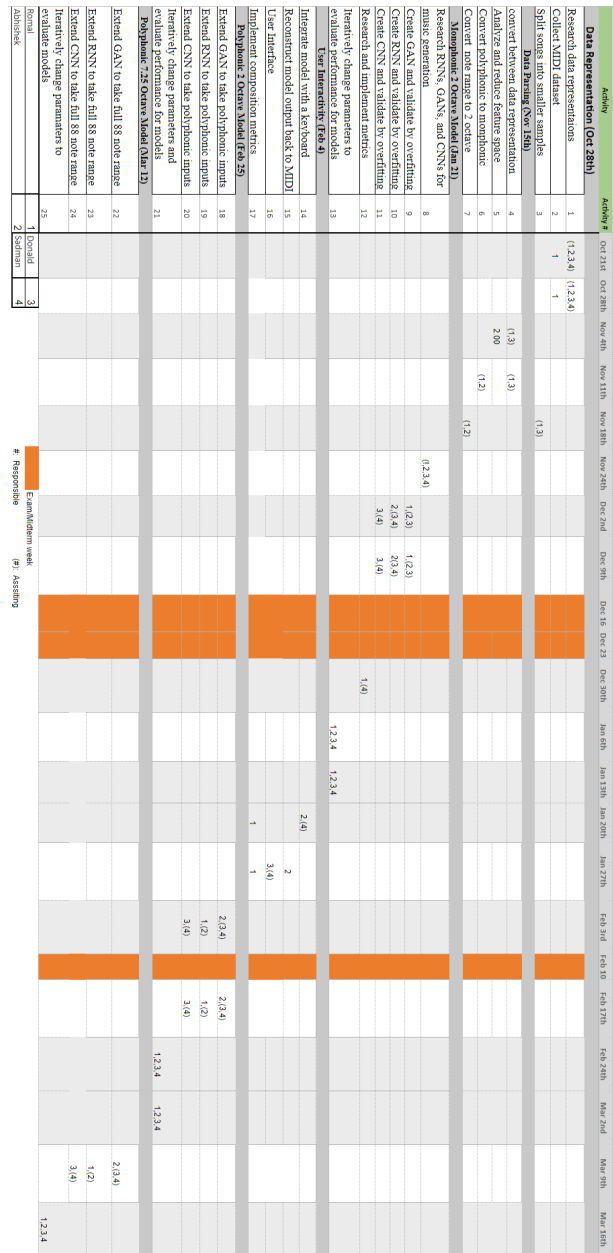


Figure 16: Gantt Chart from the project proposal

Below is the updated Gantt Chart based on our progress:

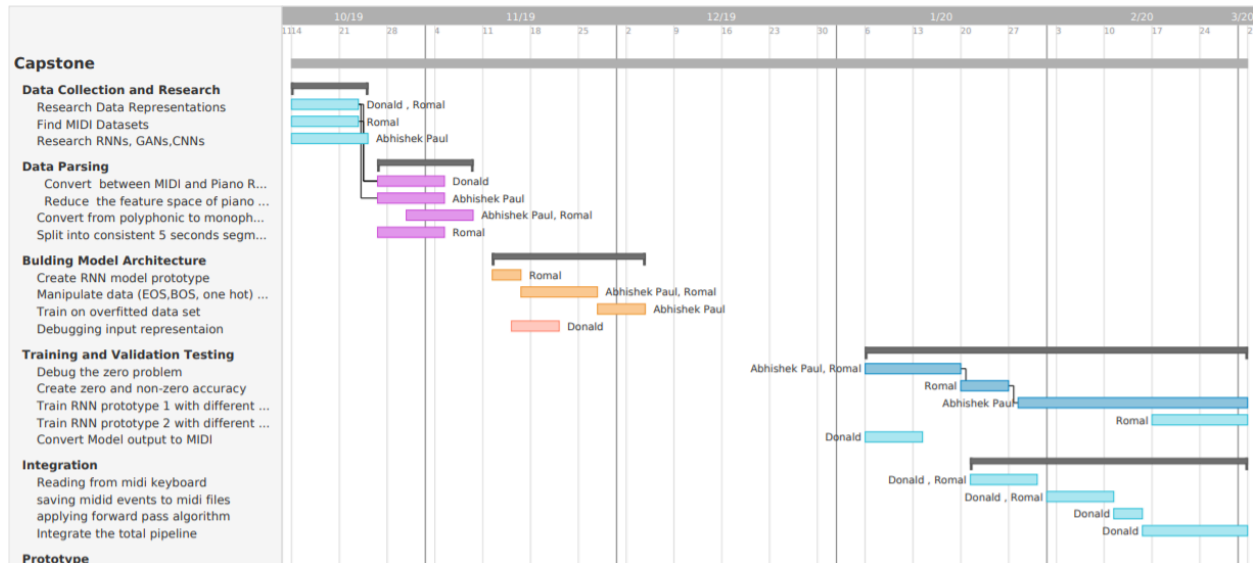


Figure 17: Updated Gantt Chart

One of the major changes to the updated Gantt Chart as can be seen above is that the scope of the project was reduced from creating and testing three models to just the RNN. This was partially caused by the fact that one of our teammates did not contribute to the project at all and we needed to distribute his parts among the other three. Furthermore, the data manipulation stage of the project was much more work than anticipated, so the training of the model was delayed until January. The updated Gantt Chart has clarified the manner in which the team worked on these tasks in a more accurate representation, where parts assigned to the non-contributing member had to be reassigned.

APPENDIX B: FINANCIAL PLAN

Item	Priority	Cost/Unit	Quantity(# or hours)	Total Cost	Requires Funding
Google Cloud NVIDIA Tesla K80 GPU [8]	1	Effective Hourly Rate (1 GPU) = \$0.609/hour	(12 hours/day) (30 days/month) (4 months) = 1,440 hours	\$876.96	Y
Internet Data Plan	1	Average monthly price Toronto ~= \$51/mo [12]	6 months	\$306	N
Labor of undergrad engineering researchers	1	\$14/hr [13]	1,360 (split across 4 engineers)	\$19,040	N
Total Consumables / Services				\$20,222.96	
Total Requiring Funding				\$876.96	

Table 4: Cost of Consumables / Services

Item	Priority	Cost/Unit	Quantity	Total Cost	Requires Funding	Paid for by Students
88 key MIDI keyboard	1	\$337.87 [9]	1	\$337.87	N	Y
MIDI cable	1	\$45.19 [10]	1	\$45.19	N	Y
Headphones for live demo	2	\$171.52 [14]	1	\$171.52	N	Y
NVIDIA TitanX GPU	2	\$3,794.42 [15]	1	\$3,794.42	N	N
Total Capital Equipment				\$4349		
Total Requiring Funding				\$0		

Table 5: Cost of Capital Equipment

Funding Source	Amount
Students	(\$100 contribution)*(4 members) = \$400
Supervisor	\$500
Free Google Cloud Credit (\$300 each)[8]	(\$300 each)*(4 members) = \$1,200 Free Credits
Google Research Credit	\$0 - \$5000 Free Credits (based on application process[11])
Total Funding	\$1,200 - 6,200 free credits + \$900 other funding = \$2,100 - 7,100 Total Funding

Table 6: Funding

APPENDIX C: ORIGINAL VALIDATION AND ACCEPTANCE TESTS

Id	Project Requirement	Verification Method
1.0	Input File Type: MIDI	REVIEW OF DESIGN: The model must take as input a MIDI file with file extension “.midi”. Any other file type as input is considered not valid. MIDI reading Python libraries must be able to read this MIDI.
2.0	MIDI Note Range: 48-73	TEST 1: The MIDI input file will be examined and the note range measured must fall between range 48-73. Any MIDI with notes outside of this range is considered not valid. TEST 2: The MIDI output file generated by the model will be examined and the note range measured must fall between range 48-73. Any MIDI with notes outside of this range is considered invalid.
2.1	MIDI Note Range: 21-108	TEST 1: The MIDI input files will be examined and the note range measured must fall between range 21-108 (piano full range). Any MIDI with notes outside of this range is considered not valid. TEST 2: The MIDI output files generated by the model will be examined and the note range measured must fall between range 21-108 (piano full range). Any MIDI with notes outside this range is considered not valid.
3.0 3.1	Input Type: Monophonic Input Type: Polyphonic	TEST: Monophonic: For each MIDI input and output file, iterate through each timestep. At each timestep, the number of notes being played must be one or zero. TEST: Polyphonic: For each MIDI input and output file, iterate through each timestep. Across all timesteps, there should be instances of more than one note being played at least once.
4.0	Input Representations: Pianoroll	REVIEW OF DESIGN: The MIDI input file will be parsed into Pianoroll representation. This representation must be a 2D array containing each note in note range 48-72 (minimum range) or 21-108 (full piano range) being played at each timestep.

Table 7: Original Validation and Acceptance Tests

Id	Project Requirement	Verification Method
5.0	Model Architectures <ul style="list-style-type: none"> - GANs (Generative Adversarial Networks) - RNNs (Recurrent Neural Networks) - CNNs (Convolutional Neural Networks) 	TEST: Each of the examined model architecture's validity will be measured using overfitting. A small data set (single batch) will be fed into each of the models to be overfitted and reproduce a known result. Each model must produce 100% accuracy with the overfitted data to be considered a valid model during this testing procedure.
6.0	Training Period: 24 hours	TEST: A model will stop its training period if it exceeds the maximum 24 hours constraint set in the requirements.
7.0 7.1	Consistent duration of input/output: 5 seconds Consistent duration of input/output: 10 seconds	TEST: The duration of the input MIDI is measured and compared to the duration of the output MIDI. The durations of input and output must be equal to the specified amount (5 or 10 seconds) .
8.0	Training Metrics (comparing generated music to target music) <ul style="list-style-type: none"> - Pitch Accuracy - Beat Accuracy - Overall Accuracy 	TEST: Test accuracy of the model will be evaluated on these metrics: <ul style="list-style-type: none"> - Pitch accuracy will be at least 66% - Beat accuracy will be at least 66% - Overall accuracy will be at least 33% Calculations of these metrics are outlined in [Appendix B]
9.0	Composition Metrics (comparing user input and output to statistics found in the dataset)[Appendix C] <ul style="list-style-type: none"> - Number of leaps - Number of dissonances/consonances - Average rest time 	REVIEW OF DESIGN: The metrics as outlined in [Appendix C] should be employed to enhance the music composition experience of the user. If the composition metrics are not employed, it is considered a failure of this objective.

Table 7: Original Validation and Acceptance Tests (continued)

Id	Project Requirement	Verification Method
10.0	Output File Type: MIDI	REVIEW OF DESIGN: The model must output a MIDI file with file extension “.midi”. Any other file type generated as an output is considered not valid. MIDI reading Python libraries must be able to read this MIDI.
11.0	User Interface	REVIEW OF DESIGN: A MIDI instrument should connect and interface with the software.
11.1	Immediate Processing	TEST: A MIDI output file should be generated less than 1 second after the time the input melody is finished playing.

Table 7: Original Validation and Acceptance Tests (continued)

Changes from the original requirements are as follows:

Requirement 7.0 Consistent duration: added a ± 0.2 margin in the duration due to granularity issues in MIDI processing libraries

Requirement 8.0: Training Metrics: Pitch Accuracy and Beat Accuracy were changed to Zero and Non-Zero accuracy. One of the original ideas was to have two models, one that focussed on timing, and one that focussed on pitch, but this was changed to the current model as development continued.

Requirements 12-14: Added standardized tempo, standardized velocity, and silence threshold requirements in order to reduce bias and improve consistency of the models.

APPENDIX D: VALIDATION AND ACCEPTANCE TEST RESULTS

Requirement 1.0: Input File Type: MIDI

The recorded melody from the MIDI keyboard is converted to MIDI format and saved into the directory. Then the saved recorded_melody.midi file is used as the input to the model.

```
note_range=end_note-begin_note  
midi_title = "./recorded_melody.mid"  
track_num = 0
```

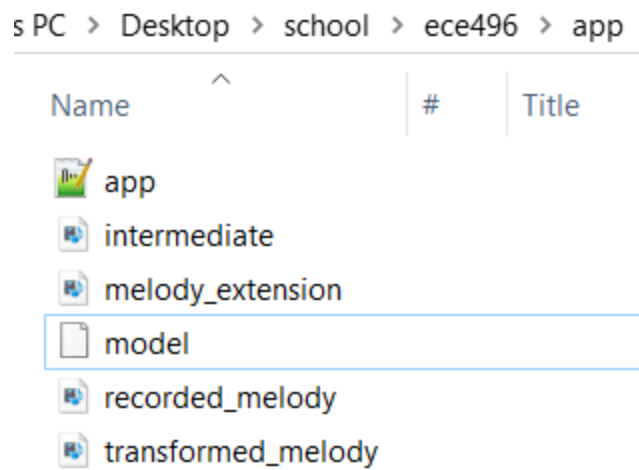
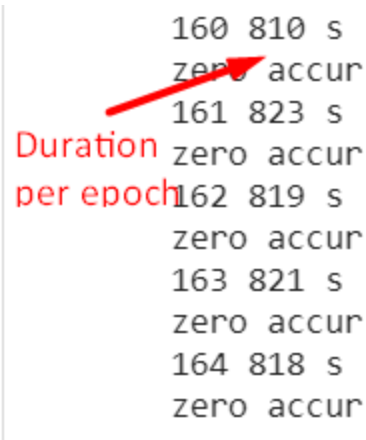


Figure 17: The generated recorded_melody.midi file is used as the input to the model

Requirement 6.0: Training Period 24 hours

Each epoch is about 13 minutes long which means that about 110 epochs can be run per 24 hours



```
160 810 s
zero accur
161 823 s
zero accur
162 819 s
zero accur
163 821 s
zero accur
164 818 s
zero accur
```

Duration
per epoch

Figure 18: Shows the duration of each epoch

Requirement 7.0: Consistent duration of input/output: 5 ± 0.2 seconds

The measurements for the converted duration of the input file as shown below (“converted”) was approximately 5.002 seconds. After the model processed the input file the output duration (“final”) was approximately 5.002 seconds.

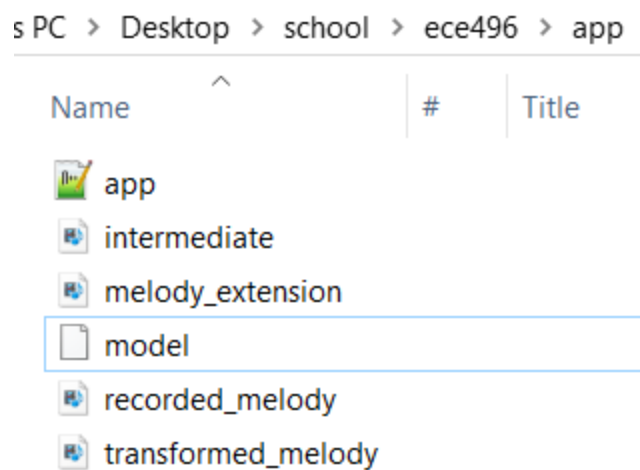
```
initial: 4.986458333333333
test: 4.9818181818181815
converted: 5.002272727272727
final: 5.002272727272727
```

Figure 19: Measurements of input and output MIDI file durations

Requirement 10.0: Output File Type: MIDI

The output from the model is converted into MIDI format and saved into the directory in order to be played back to the user.

```
music_file = "melody_extension.mid"  
play_music(music_file)
```



Figures 20 and 21: The melody_extension.midi file is the output MIDI file that is generated from the model

Requirement 11.1: Immediate Processing

Measurement of processing time after the input melody is finished playing to the time all the data transformation steps and model processing steps finish.

```
Processing time: 0.3875401020050049
```

Figure 22: Measurement of processing time

Requirement 12.0: Standardizing tempo

A sampler was run through the dataset and checks that the tempo is normalized to the chosen value (120). The sampler found that all the tempos were normalized.

```
tempo normalized
tempo normalized
tempo normalized
tempo normalized
tempo normalized
tempo normalized
tempo normalized
tempo normalized
tempo normalized
tempo normalized
```

Figure 23: Shows the output of the sampler function that goes through each file and checks if the tempo has been normalized

Requirement 13.0: Standardizing Velocity

A sampler was run through the dataset and checks that the velocity is normalized to the chosen value (100). The sampler found that all the velocities were normalized.

```
segment 57 velocity is 100
segment 58 velocity is 100
segment 59 velocity is 100
segment 60 velocity is 100
segment 61 velocity is 100
segment 62 velocity is 100
segment 63 velocity is 100
segment 64 velocity is 100
segment 65 velocity is 100
segment 66 velocity is 100
segment 67 velocity is 100
segment 68 velocity is 100
```

Figure 24: Shows the output of the sampler function that goes through each file and checks if the tempo has been normalized

Validation Test Accuracy (2.4.1 Assessment of Final Design)

The model was run on a dataset of 500 MIDI files that were not part of the training set in order to see how well the model can generalize to new music

[illegible]

Figure 25: Shows a screenshot of the validation accuracy results for the first 10 segments in our test set

REFERENCES

- [1] G. Wiggins,. and S. Alan “Musical Knowledge: What can Artificial Intelligence Bring to the Musician?” *Readings in Music and Artificial Intelligence*, 2000/ [Online]. Available: <https://pdfs.semanticscholar.org/aae5/4084c174e98619e88c5e5c7f641d09eb41bf.pdf> [Accessed: 18-Sep-2019]
- [2] J. P. Briot, F. Pachet and G. Hadjeres, “Deep Learning Techniques for Music Generation - A Survey” *arxiv*, 29-Aug-2017. [Online]. Available: <https://arxiv.org/pdf/1709.01620.pdf> [Accessed: 18-Sep-2019].
- [3] D. Eck et. al. , “Magenta,” [Online]. Available: <https://magenta.tensorflow.org/>. [Accessed: 26-Nov-2019].
- [4] K. Naveed, C. Watanabe, and P. Neittaanmäki, “Co-evolution between streaming and live music leads a way to the sustainable growth of music industry – Lessons from the US experiences,” *Technology in Society*, vol. 50, pp. 1–19, 2017.
- [5] Amazon Web Services, Inc. 2020. *AWS Deepcomposer*. [online] Available at: <<https://aws.amazon.com/deepcomposer/>> [Accessed 22 February 2020].
- [6] B. Wahler, “CHAPTER 3 – LIMITATIONS AND PITFALLS,” *MIDI Solutions Event Processor Guide Chapter 3*, 2019. [Online]. Available: <http://www.midisolutions.com/chapter3.htm>. [Accessed: 10-Oct-2019].
- [7] Colinraffel.com. 2020. *The Lakh MIDI Dataset V0.1*. [online] Available at: <<https://colinraffel.com/projects/lmd/>> [Accessed 16 November 2019].
- [8] *Google Cloud Platform Pricing Calculator | Google Cloud Platform | Google Cloud*. [Online]. Available: <https://cloud.google.com/products/calculator/#id=96ea1cc5-4103-4942-82e9-a6d6f1549f7e>. [Accessed: 10-Oct-2019].
- [9] “M-Audio Keystation 88 MkII 88-Key USB MIDI Controller,” Best Buy Canada. [Online]. Available: <https://www.bestbuy.ca/en-ca/product/m-audio-keystation-88-mkii-88-key-usb-midi-controller/10478421>. [Accessed: 10-Oct-2019].

- [10] “Roland MIDI Cable (UM-ONE-MK2): Amazon.ca: Musical Instruments, Stage & Studio,” [Online]. Available:
https://www.amazon.ca/ROLAND-UM-ONE-MK2-Roland-MIDI-Cable/dp/B00967UN50/ref=asc_df_B00967UN50/?tag=googleshopc0c-20&linkCode=df0&hvadid=292980589401&hvpos=1o3&hvnetw=g&hvrnd=15332265880211451964&hvpone=&hvptwo=&hvqmt=&hvdev=c&hvdvcmdl=&hvlocint=&hvlocphy=&hvtargid=pla-404766665199&psc=1. [Accessed: 10-Oct-2019].
- [11] “GCP Free Tier - Free Extended Trials and Always Free | Google Cloud,” Google. [Online]. Available: <https://cloud.google.com/free/>. [Accessed: 10-Oct-2019].
- [12] *Canada’s Internet Speed And Prices*. [Online]. Available:
<https://www.gonevoip.ca/canada-s-internet-speeds-and-prices/> [Accessed: 23-Oct-2019].
- [13] “University of Toronto Hourly Pay,” Glassdoor. [Online]. Available:
<https://www.glassdoor.ca/Hourly-Pay/University-of-Toronto-Hourly-Pay-E31043.htm>. [Accessed: 29-Nov-2019].
- [14] “ATH-M50x Professional Monitor Headphones,” Audio. [Online]. Available:
<https://www.audio-technica.com/cms/headphones/99aff89488ddd6b1/index.html>. [Accessed: 29-Nov-2019].
- [15] “NVIDIA Titan RTX Graphics Card: Amazon.ca: Computers & Tablets,” NVIDIA Titan RTX Graphics Card: Amazon.ca: Computers & Tablets. [Online]. Available:
https://www.amazon.ca/NVIDIA-Titan-RTX-Graphics-Card/dp/B07L8YGDL5/ref=pd_sbs_t_2/134-3130773-6312516?_encoding=UTF8&pd_rd_i=B07L8YGDL5&pd_rd_r=864eb91a-ca56-4aab-8d09-650d0f4a12ed&pd_rd_w=prvvW&pd_rd_wg=Jubv7&pf_rd_p=9926bb69-42b9-46e4-b788-f665992e326d&pf_rd_r=1K3QYZDYX9E4AKPT0FVZ&psc=1&refRID=1K3QYZDYX9E4AKPT0FVZ. [Accessed: 29-Nov-2019].