Addressing the Gap Between Machine Learning and Casual Music Creation

CS39440 Major Project Report

Author: Daniel Swift (das82@aber.ac.uk)

Supervisor: Dr Chuan Lu (cul@aber.ac.uk)

05th May 2022

Version 1.9 (Release)

This report is submitted as partial fulfilment of a BSc degree in  
Computer Science (G401)

Department of Computer Science

Aberystwyth University

Aberystwyth

Ceredigion

SY23 3DB

Wales, UK

Declaration of originality

I confirm that:

* This submission is my own work, except where clearly indicated.
* I understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
* I have read the regulations on Unacceptable Academic Practice from the University’s Academic Registry (AR) and the relevant sections of the current Student Handbook of the Department of Computer Science.
* In submitting this work, I understand and agree to abide by the University’s regulations governing these issues.

Name Daniel Swift

Date 28/04/2022

Consent to share this work

By including my name below, I hereby agree to this project's report and technical work being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Name Daniel Swift

Date 28/04/2022

Acknowledgements

I am grateful to…

I’d like to thank…

Abstract

Music is one of the most expressive and emotive forms of human art. Deeply ingrained in the roots of different cultures and communities, music is widely created and enjoyed in various forms throughout the world. The rise of personal computers and innovations such as the Musical Instrument Digital Interface (MIDI) standard has born a new era of digital music creation, capitalised on from the biggest artists in the world perfecting their sound, to curious users experimenting with music scales in their bedroom.

Coinciding with this is the rise in machine learning, which is also becoming more accessible to a wider, non-specialised audience. However, as of this report, there are still many areas of machine learning that are too complicated or expensive for the more casual user. My project has tried to address this problem, allowing people who may have some background in digital music creation but none with machine learning or AI in general, and allow them to combine these relatively new fields as seamlessly as possible.

In this project I will compare different generative algorithms regarding their suitability for music generation. Furthermore, I will attempt to create software that is user-friendly and intuitive, whilst enabling users to generate music from various complex music generation algorithms.

Contents

1. Background, Analysis & Process 7

1.1. Background 7

1.1.1. Initial Motivations 7

1.1.2. Music and Sound Research 7

1.1.3. MIDI Standard and Useful Filetypes 8

1.1.4. Neural Network Models 8

1.1.5. Similar Existing Solutions 9

1.2. Relevant Tools and Libraries 10

1.2.1. Machine Learning Libraries 10

1.2.2. Development Environments and Package Managers 11

1.2.3. GUI Libraries 11

1.3. Analysis 12

1.3.1. Gather Data 12

1.3.2. Perform Tests with Provided LSTM Model Examples/Tutorials and Train with Custom Configurations and Architectures 12

1.3.3. Analyse the Magenta Pre-Trained RNNs and Attempt to Train a Custom Model 13

1.3.4. Develop a Program based on the Previous Experiments 13

1.3.5. Potential Security Issues 13

1.4. Process 13

2. Machine Learning for Music Generation 15

2.1. Introduction 15

2.2. Data 15

2.3. RNNs/LSTMs 15

2.4. Magenta pre-trained RNN (unfinished) 19

2.5. GANs 19

3. Interface Development 20

3.1. Sprint 1: 16th March – 19th March 20

3.1.1. Version Control 20

3.1.2. Software Setup and Package Management 20

3.1.3. Data Collection and Initial Colab Demos 20

3.2. Sprint 2: 4th April – 8th April 21

3.2.1. Plan and Objectives 21

3.2.2. Design 21

3.2.3. Implementation 23

3.2.4. Testing 24

3.2.5. Review 24

3.3. Sprint 3: 29th April – 4th May 24

3.3.1. Plan and Objectives 24

3.3.2. Design 25

3.3.3. Implementation 26

3.3.4. Testing 27

3.3.5. Review 27

4. Testing 28

4.1. Overall Approach to Testing 28

4.2. Automated Testing 28

4.3. Integration Testing 29

4.4. User Testing 29

5. Critical Evaluation 30

5.1. Objectives and Requirements 30

6. References 31

7. Appendices 34

A. Third-Party Code and Libraries 34

B. Code Samples 35

C. LSTM Experiment Tables 37

First LSTM Model Experiments Table 37

Second LSTM Model Experiments Table 40

Magenta Model Experiments Table 40

Screenshots 41

*D.* Test Tables 46

# Background, Analysis & Process

## Background

### Initial Motivations

My initial interest in pursuing this area for my project stemmed firstly from the fact that I love both listening to and playing guitar music. I started playing guitar around five years ago, and have accrued some knowledge in how to write my own music and some of the theory behind it. Furthermore, I believe that in those five years, the way I perceive and understand music that I listen to has changed drastically. Being able to play a musical instrument has made me appreciate skill and creativity a lot more, especially guitar music.

This is no coincidence, as musicians with different expertise selectively engage different sensorimotor regions in the brain when listening to the instruments they are able to play [1]. However, this is not to say that people with some kind of musical expertise are better at perceiving music than non-musicians, obviously non-musicians are able engage, understand, and build powerful connections with music they listen to. This is shaped by numerous factors, such as mood, setting, the piece itself[2], or the person’s familiarity with the piece and genre[3].

In terms of familiarity with machine learning, I spent my year in industry working in the marketing department for a company producing machine learning and AI related products, with emphasis on these products being relatively easy to program and deploy within businesses and schools. The focus on making AI and machine learning more user-accessible and widely accepted was another source of inspiration for me to partake in this specific project. I hoped that it might be an insightful way to get more people into both writing music and machine learning, as well as being a genuinely useful tool for basic music composition.

### Music and Sound Research

My initial research and reading predominantly regarded the different kind of neural network models that could be used for this project, the libraries that supported them, the languages or programming interfaces that I could use, and similar existing solutions that I could draw inspiration from.

Music itself is composed of many different elements, which combine to create the various melodies that we can perceive in a variety of ways. It can be conveniently described as a series of notes, which play out in a linear pattern from the beginning of a song until the end. These notes can be identified by their pitch (the frequency of the soundwave they produce), duration, velocity, volume, and many others. Different notes can be played at the same time to make up chords, and can be played in conjunction with other specific notes to make up scales.

It is important to understand what makes music sound “good”. Sure, pressing random keys on a piano or plucking at random strings on a guitar is probably not going to sound like a masterpiece, but in order to play something harmonious, notes need to stay in the same key, and should be played within defined scale intervals.

Following on, it could be easy enough to develop an algorithm where different scales are hardcoded, and it would select notes at random (or in a loosely determined pattern based on other inputs) in order to generate a piece that, while potentially sounding quite weird, would be in key. This would, in turn, reduce the sense of originality, as the piece would be the product of either over-determinism or pure randomness, where in reality we would require more of a balance.

This would justify a need for machine learning, as, for instance, a neural network model could be trained from music that is generally well regarded, and would likely follow different scale intervals and remain in key. The model would pick up on this so that when it generates its own music, the notes should all remain in key. However, remaining in key is not always the most important factor. It is worth acknowledging that notes can sound good sometimes when played out of key, and this should be reflected in the songs within the provided training datasets. Furthermore, obviously the concept of “good” music is subjective, and different individuals will form an impression based on the reasons aforementioned in 1.1.1.

### MIDI Standard and Useful Filetypes

MIDI is a technological standard that evolved to enable communication between digitally enabled instruments, such as 1980’s synthesisers. Later, it provided a platform for users with personal computers to record, store and edit music[9]. MIDI files themselves are normally relatively small, meaning that they do not require much disk space, due to the fact that they do not have any actual audio data. In place of this, they contain data regarding what notes are to be played, the time that they are played at, the duration of the note, and how loud it should be[10]. Due to this, many musical pieces that are converted to MIDI files will sound different to the original recording, and sometimes unpleasant.

### Neural Network Models

In terms of selecting a specific type of generation algorithm, there were a few options. Most early music generation techniques used Recurrent Neural Networks (RNN), such as Long Short-Term Memory (LSTM) networks, which would generate single-instrument music in the same way that natural language models work[4][5][6].

A big benefit of using RNNs is that they have the ability to apply contextual information when mapping between input and output sequences, however the drawback is that the range of this context is limited. To explain this further, within RNN architectures, the influence of a given input on the hidden layer, and then in consequence the output, either decays or blows up exponentially as it cycles through the network’s recurrent connections[7]. This is more commonly known as the vanishing gradient problem[8].

LSTMs have been developed to address these limitations, and have proven more successful at it[12]. Within the architecture of an LSTM is a constant error carousel (CEC), which basically affords protection to input gates by perturbing irrelevant inputs [13].

Another potential model could be a Generative Adversarial Network (GAN). GANs work using unsupervised learning, however it frames itself as a supervised learning algorithm, using two sub-models, a generator model, and a discriminator model. The generator model is trained to generate new data from an existing dataset, and the discriminator model attempts to classify data that is real (from the given dataset), or generated by the generator model. The two models work at the same time until the discriminator fails to correctly determine the difference between real and generated data at a pre-determined rate[19], the original proposed rate being 50%[20].

### Similar Existing Solutions

#### SOUNDRAW

A screenshot of a computer

Description automatically generated with medium confidence“Soundraw” is a web-based music generator that uses pre-trained AI to generate songs at a selection of lengths, based on a user-given input of the song’s mood, genre (or style), tempo and instrument. The generated music can also be edited to cut its length, slow or quicken its beats per minute (bpm), change the key, and change the volume, amongst other advanced attributes at specific timings.

Figure 1: The GUI for soundraw.io[22], with its extensive plethora of features

The design of this app is very professional and fairly straightforward, and its use of well-developed pre-trained models is a good source of inspiration for how this project could work. However, this solution offers no way for users to train their own models, or select the different kind of generation algorithm they would like to use.

#### dopeloop.ai

The “dopeloop” melody generator is a web-based application that is able to generate melodies from three user-given inputs. The first is how many notes to generate (the length), the second the root note, and the third is the scale to generate the notes in. The notes are generated randomly within these given confinements. The user also has the option to change the instrument and the bpm. This application is also available as a mobile app.

Graphical user interface, application

Description automatically generated

Figure 2: The GUI for dopeloop.ai's[21] melody generator

Whilst the way that music is generated for this solution is not the same way that it will be generated in this project, the simplicity of the GUI design and the swiftness with which it operates provides a large source of inspiration for the practical outcome of this project, as well as advocating for a web-based solution.

## Relevant Tools and Libraries

### Machine Learning Libraries

#### Keras/TensorFlow

TensorFlow is an open-source library built specifically to allow developers to create and train machine learning models. Created by Google, TensorFlow is predominantly used with Python and JavaScript, and it is one of the most widely used libraries in relation to machine learning[11]. Keras is an extension of TensorFlow which tries to make it a little more user friendly.

#### PyTorch

Similarly to TensorFlow, PyTorch is an open-source library, but focused on accelerated deep neural network programming and developed by Facebook[14]. However, when compared to TensorFlow, it would probably fall a little short for this project. This is because TensorFlow has a superior and well-documented framework, more of an abundance of tutorials, and better visualisation of its processes, whereas PyTorch provides only limited visualisation. On the other hand, PyTorch gains optimal performance with native asynchronous execution within Python, whereas for TensorFlow this would have to be manually coded in[15].

#### Scikit-Learn

Scikit-Learn offers a wide range of machine learning algorithms, with alternatives to neural networks. Decision trees, Support Vector Machines (SVM), and linear regression, to name a few, are all available to use through this library.

#### Magenta

Magenta is a self-described open-source research project, created by Google using the TensorFlow library. It is specifically designed to explore the role of machine learning when undergoing the process of creating art and music, and is available for both Python and JavaScript[16]. Magenta can be used to train custom models, but also comes with a selection of pretrained models which can be changed certain parameters to affect the final output of the model[17].

### Development Environments and Package Managers

#### Anaconda

Anaconda is an open-source Python distribution platform, with a focus on making data science and machine learning easily accessible on one machine. It provides a user-friendly graphical user interface (GUI) which makes it easy to switch between different development environments, and features its own repository of thousands of open-source data science and machine learning related packages. This includes previously mentioned packages such as TensorFlow, PyTorch and Keras[18].

#### Google Colab

Google Colab is a Jupyter notebook environment that provides a free web-based python environment built specifically for research and data science. Whilst being able to run arbitrary Python code step-by-step in different coding chunks, the best feature of Colab is that it provides access to computing resources including RAM and GPUs for free. This allows users to perform large processes with ease that everyday PCs may struggle with, such as training a large complex model with a large dataset that requires strenuous pre-processing. Furthermore, Colab has plenty of built-in features that drastically help to improve the visualisation of data, such as being able to play midi files and plot graphs in-browser.

### GUI Libraries

#### PyQT5

PyQT5 is a GUI framework for Python written in C++, which is free and open source. Unlike some other GUI libraries for Python, it offers a visual framework that allows for developing GUIs via drag and drop of a wide selection of widgets to build these interfaces, obviously leaving the coding for the backend[23].

#### Tkinter

Tkinter is a built-in library for GUI development in Python. Alternatively to PyQT5, all of the GUI widgets and canvases are programmed manually, without the aid of a drag and drop visual framework. This may seem like it is trickier to learn, however I personally have had a lot of experience using it in the past. Furthermore, it is normally packaged in with the core Python libraries, so there is no need to externally install it[24].

## Analysis

Based on background research, a basic structure was devised for the preliminary approach for this project. The approach was decided as follows:

### Gather Data

Based on looking at numerous other models, it was clear that an abundance of data would need to be collected for training and validation purposes, and to increase the chance of the models producing music of a good standard. The data would need to be in the form of MIDI files, and sourced online. The option of creating my own MIDI files from my electric guitar was considered, however this would require specialist equipment, which was too expensive. Fortunately, there is a plethora of websites where midi files are freely available, so it would have been relatively easy to look through these websites and download as many as was needed.

Graphical user interface

Description automatically generated

Figure 3: A selection of thousands of available MIDI files from midiworld.com

### Perform Tests with Provided LSTM Model Examples/Tutorials and Train with Custom Configurations and Architectures

Initially, it was decided that the best libraries to work with for this project would be TensorFlow (and by extension Keras), and Magenta. This is because they are both very well-documented, have extensive, in-depth tutorials, and are both open source, meaning that it is easier to visualise how they are structured.

Consequently, this did mean that the majority of the development of this project would have to be conducted using Python, due to the fact that most of the provided tutorials and documentation use Python code (although Magenta is also compatible with JavaScript). On one hand, this removed the advantage of the highly accessible and functional user interface that a web-based project would have, although the GUI libraries provided in Python are also well-documented, and personally I have experience with GUI development so it should not be as steep a learning curve than learning JavaScript.

Furthermore, this opened up the strong advantage of being able to use Google Colab, as it was quickly realised that experimenting with machine learning would require some pretty powerful hardware, which is something that was not in my possession. It may have been possible to do something like log into a remote server and try to develop from that, but it was decided that this was unfeasible and unnecessary.

Using the example code provided by TensorFlow, a model would be trained, and the results analysed within Google Colab. The initial inputs and other configurations would then be altered and changed in order to get a better idea of what would cause the model to produce better outputs. Based on this, another model would be trained with the data gathered in the previous step, in order to evaluate the effectiveness of the model and the data that had been collected.

### Analyse the Magenta Pre-Trained RNNs and Attempt to Train a Custom Model

Similarly to the step before, the examples given by Magenta would be analysed, and the results of the given pre-trained models would be tested. Based on background research of the pretrained models, at a glance they are very effective at generating good music. Next, Magenta would be used to create and train a custom model, in order to see if it would be more or less effective than the previously experimented-with LSTM models.

### Develop a Program based on the Previous Experiments

Initially the focus would be on setting up the necessary software required, including some of the relevant tools and libraries mentioned in section 1.2. Since the models were created in Python, it made sense for compatibility reasons that the GUI should be written in Python too. It was decided that Tkinter would be used, for two reasons. The first was that as mentioned before in section 1.2.3, it comes packaged in with the Python’s standard built-in libraries.

Secondly, as I have some previous experience working with Tkinter, it would save time and resources to just use it again, as opposed to having to learn how to use a new library. PyQT5 did appear to be easy to use and learn, and if it were the case that I had to learn both libraries from scratch, then it most likely would have been selected instead. However, despite the moderately higher complexity of Tkinter, it is able to do pretty much everything that PyQT5 is able to provide.

The GUI should be able to load and play MIDI files, and have options to generate new music using a selected model, and furthermore have the ability to generate some kind of music tablature, such as a guitar tab, from the given MIDI file. There should be further options to alter the configuration of the models so that the sounds produced can be altered accordingly.

### Potential Security Issues

There did not appear to be any relevant potential security issues or threats regarding this project. There is no handling or storing of sensitive data, and it does not require the use of human or animal test subjects. A lot of the libraries used for this program are open source, which could leave them vulnerable to bugs and attacks, but most of them are now well-documented and continuously updated which reduces these risks to a minimum.

## Process

This project is a mixture of research and software engineering. Since a large part is taken up by research, the scale of the development side is reduced, and greatly affected by the results of the research. Therefore, the process of developing the GUI should reflect this. A methodology was needed that would allow for flexibility with the design and work through an iterative process. It was decided that an Agile approach would be used, with moderate elements of Scrum incorporated. This would work by having multiple sprints (as is common in Scrum[25]), and in each sprint plan, design, implement and test. Since the scope of this side of the project is relatively smaller, it was felt that it was achievable to develop it in this way.

# Machine Learning for Music Generation

## Introduction

The concept of first using computers to generate music was proposed in writing by Ada Lovelace in 1843, where she wrote:

*“Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent….”*[26]

However, much of the progress in the field of machine learning, such as deep learning, has only really been made in the last 25 years, and only in the past decade has there been a wider availability for using it outside of research purposes, and there continues to be limitations. As *L Casini et.al* argue, while models are able to produce pieces that are good enough to be considered music to the casual listener, there still remains a vital human element, especially when generating longer pieces, which generally lack structure and meaning to more musically inclined listeners[27]. Furthermore, deep learning structures do not have direct ways to control how they generate results, and generate music without human interaction[28].

This chapter refers to my exploration of using different machine learning algorithms to generate music in order to see what settings and architectures are best suited for this task. The full list of my experiments and their accompanying screenshots can be found in Appendix C.

## Data

The rise in machine learning has undoubtedly coincided with the rise in data gathering. Algorithms used within the field of machine learning, specifically deep learning, continue to become larger and more complex, requiring more and more data to perfect their results[28]. Throughout the process of experimenting with the different models and training data that have been used for this project, a consistent issue was Google Colab timing out after or during training the models. Obviously, this is necessary in order for Google to continue to provide the service for free, by freeing up space and sharing resources across everyone who wants to use it, but, consequently, it was difficult to train with larger datasets and complex models since it could not be left for more than a few hours at most.

## RNNs/LSTMs

Using RNNs to generate music has become more popular in recent years, as their success with generating and predicting text sequences[31] can be used in the same way for music. Whereas text generation would use characters however, music generation would use notes, each with a pitch, duration step, velocity etc. RNNs on their own however, face challenges when working with longer sequences, and furthermore, music generation requires the ability of an algorithm to look back at its previous run-throughs, which is a limitation of the RNN. Furthermore, as mentioned in section 1.14, RNNs suffer from the vanishing gradient problem, which further affects the effectiveness of the model.

This draws the issue to the solution of LSTM models, which (as again mentioned in section 1.14), address the vanishing gradient problem. Previous research has found that even a single layer LSTM model can produce “stellar” results when composing new melodies[32]. This was what me convinced to start my experiments using a simple LSTM model, based on the tutorial provided by TensorFlow for generating music.

The initial dataset was the “Maestro 2.0” dataset provided by Google. This contained 1282 MIDI files that could be easily used to train[34]. The data was then processed and extracted into individual notes, which the model could use to train on. The training set as a whole was then created, and the sequence lengths were set, with the help of functions to convert notes to MIDI files and vice versa. The model with one LSTM layer was then created and trained. After the model is trained, it can be used to generate new sequences of notes, where the values for temperature and the input sequence can be played around with.

The results from these initial results were promising, with the music produced having some basic consistency and having a lively tempo. Increasing the temperature had the effect of adding some more variety to the different pitches produced, but in turn decreased the consistency. I finally tried changing the input sequence, to a jazz tune, which changed the mood of the result slightly but had no real change.

A picture containing chart

Description automatically generated

Figure 4: The loss per epoch for the first tests

I then decided to train the model again, but this time using the ADL Piano MIDI dataset[35][38]. This was much larger, and featured a greater variety of genres and songs. I hoped this would mean the model would train better and produce better results. However, this was not to be the case. Initially, I planned to train over 30 epochs, however it stopped the training after just 14. Figure 5 provides the first indication that something was not right:

Chart, line chart

Description automatically generated

Figure 5: The loss per epoch for the second tests

Rather than a consistent decreasing curve as initially expected, the loss had almost curved back to what it was at the start. It is possible that the model had just reached an early local minimum, but in any case the results were very poor. Firstly, all of the produced results lacked any variation in each note’s duration and pitch, resulting in a long, monotonous repetition of the same note over and over again.

A picture containing chart

Description automatically generated

Figure 6: The distribution of each note's pitch, step, and duration for test 1.0

These poor results continued despite altering the temperature and the input sequences, both of which had little to no effect.

In case the model had indeed just been stuck in a local minimum, I decided to increase the number of epochs to 60. I also reduced the dataset to just one genre, the classical genre with 1398 songs, in order to see if the large mix of genres was having an effect on the output. I also changed the initial input when generating music from the model to a classical song.

Chart, line chart

Description automatically generated

Figure 7: The loss per epoch for test 1.4

These changes however, seemed to again have very little effect, other than the frequency of the duration of each note was slightly more distributed. The frequency of the pitch stayed the same. I also tried increasing the temperature to 5.0, which is a lot higher, but this only had a limited effect on the variation of the pitches, with the produced piece being uninspired and directionless.

Chart, histogram

Description automatically generated

Figure 8: The distribution of each note's pitch, step, and duration for test 1.5

Due to the poor results from the simple models I was using, I instead tried using a different approach. The new model would have multiple LSTM layers, and would take advantage of using the ADL Piano MIDI dataset, specifically the Folk genre. My hope was that using data that was in the same genre would generate more cohesion and structural integrity within the produced music.

The new approach that was tried was based off a third-party module, “Classical Piano Composer” by Sigurour Sigurgeirsson[33], which I mainly used to help with the data pre-processing and the general structure of the model, although adjustments were made to both of these attributes. This code was also used to generate techno music, in testament to the wide-ranging possible implications of his module[30].

The setup for this on Google Colab was similar to the previous ones based on the TensorFlow tutorials, where the MIDI files are converted into individual notes. However, the architecture of the model was more complicated, now featuring 3 LSTM layers rather than one. This made training and fitting the model take a lot longer per epoch. So, to reduce the training time, I reduced the shape of each layer to 128, as opposed to original values of 512. I also limited the dataset to just use the Folk genre.

The model took nearly four hours to train, but had a nice, consistent loss curve:

A picture containing chart

Description automatically generated

Figure 9: The loss per epoch for test 2.0

The results that this model produced were by far better. The resulting music, named “output20”, featured a grounded, consistent alternating bass melody, that had streaks of creative notes flaring off it at varying times. There was a lot of repetition, especially at the beginning and at the end, but overall it actually sounded like a presentable piece.

Chart, scatter chart

Description automatically generated

Figure 10: The notes distribution over time for the resulting output of test 2.0

This would be a good model to use in the final application, as it was relatively easy to implement, and produced fairly good music.

## Magenta pre-trained RNN (unfinished)

The Magenta library also provides a framework for an RNN model, but this one was pre-trained. MelodyRNN

## GANs

As mentioned in section 1.1.4, GANs are also a viable way to generate music, specifically transformer GANs. GANs are widely used across many aspects of our lives, including in social media, for instance in face swapping[36]. In terms of audio generation, great strides have been taken in this area, for instance the release of WaveGAN, which can synthesize slices of audio waveforms that have global coherence[37]. Magenta also provides numerous resources for this.

# Interface Development

As per the chosen methodology for this project, the implementation of the GUI was done in three different sprints, with the second and third having their own planning, implementation, and testing stage. However, the first sprint was purely just to get things set up so did not need much discussion.

## Sprint 1: 16th March – 19th March

As previously stated, the first sprint was about setting up some basic tools and utilities in order to give myself a solid foundation to continue with the rest of the project. I also went about collecting data to use for training.

### Version Control

The project started with the setting up the chosen form of version control, Git, initially GitLab, of which a specialised account is provided by the university. This had numerous advantages, mainly that it provides a place to store backups for every part of the project, including the Google Colab notebooks, the datasets I collected, the Python files I would work with, and the reports necessary for this project. Another is that it provides a useful timeline of the project’s history when it comes to reviewing it. It is worth mentioning however, that I had to move to GitHub, as I could only access GitLab when I was connected to the university network, so when I moved home for Easter I was unable to gain access.

### Software Setup and Package Management

From the background research, it had been found that Anaconda was a useful tool for package management, and through Anaconda it would be easy to install packages such as TensorFlow easily, rather than having to do it manually or through pip, which could have been confusing or arduous. Anaconda comes with a Python IDE installed called Spyder, which was useful as it had syntax highlighting and debugging properties. I also installed IDLE, which would load quicker on my PC and was useful if there needed to be small quick changes to the code.

### Data Collection and Initial Colab Demos

I also looked for suitable datasets to use throughout this project to train the models I would create. As mentioned in section 1.3.1, there were plenty of available resources that could be used for this project. I initially collected around 100 songs that I could use to train the models, because the models were not planned to be too complex, and the way that the data pre-processing worked meant that the songs would be split into the notes that form them, which would expand the dataset widely.

The initial expectations that the simple models and relatively smaller datasets were ultimately proven false however, after the moderately poor results from the trials using this data (see section 2.3 regarding the performance of LSTMs). Therefore, I had to search for much larger quantities of datasets, such as the Maestro dataset provided by Google and the ADL Piano MIDI dataset. I decided to use the data I collected as data to use for testing purposes.

Despite the poor results produced by the initial models used in the Google Colab demos, the fact that the models were able to produce anything at all, and the data pre-processing worked with all the different datasets I used were a big bonus, and put me in good stead for future development.

## Sprint 2: 4th April – 8th April

### Plan and Objectives

Firstly, an initial design for how the GUI should look on paper was drawn up, then there was an attempt to replicate this through Python using the Tkinter library. The initial hand-drawn design was the first source of inspiration for the interface design for this project.

<Image>

As can be seen from the provided figure, the GUI is relatively simple, with a menu bar at the top, a display for visualisation purposes and some buttons for playing the MIDI files and to increase or decrease the volume. Furthermore, there are buttons that will allow the user to upload MIDI files to be played or experimented on, along with a button to generate sheet music for the uploaded MIDI file. The final major component of this design was a button that would use an uploaded MIDI file to generate music from a pretrained model, or allow the user to train their own model. This feature would be developed in a later feature to allow for the research to continue on Google Colab.

### Design

Since at this stage all that was being focused on was the GUI, it was decided against using an object-orientated approach (in hindsight this was a mistake, but not the worst-case scenario), and if it needed to be put into a class then it would not be too difficult to do so. The majority of the tasks performed via the GUI in this project would inherently be event-driven, such as the clicks of buttons. So, after the GUI was designed, I would attach each aspect to a different function. It was decided that the official name of the program would be “MidiGen”.

#### Window

The main window would be the main area for the application, as a Tk() object, called “window”. The window was given the title: “MidiGen by das82” and the default size when it was created was set to 700x500 pixels, as the application was only small so did not need to take up an entire screen.

#### Title, File, Volume and Button Labels

The labels for each aspect of the design were all Tkinter Label objects, and were named accordingly:

* title\_label: A big header displayed at the top of the window (underneath the menu-bar) . This label had a larger font as it was a header for the title.
* filename\_label: A label to display the name of an uploaded MIDI file, by default outputting the text “No song selected!” as when the application is opened there would not be any MIDI files loaded yet.
* volume\_label: A label used to display the current volume of the application, by default set to 0.5 which is half the maximum volume.

#### Buttons

The buttons were all Tkinter Button objects and, like the labels, are named according to their function:

* pause\_b: Button to pause the current song being played.
* play\_b: Button to play the current song, if pressed twice it should restart the song.
* volume\_up\_b: Button to increase the volume by 0.1.
* volume\_down\_b: Button to decrease the volume by 0.1.
* upload\_b: Button to upload a new MIDI file that can be played and used by the program.
* generate\_b: Button to open a new window enabling the user to interact with the generation part of the program.
* sheet\_b: Button to generate sheet music from the given MIDI file.

#### Other Widgets

The menu-bar at the top of the window was a Tkinter Menu object, called “menubar”. The menu-bar had different cascading submenus:

* file\_menu: This menu had options to upload new MIDI files, to save MIDI files, and exit the program.
* tools\_menu: This menu was implemented but not fleshed out at this stage.
* view\_menu: This menu was implemented but not fleshed out at this stage.
* help\_menu: This menu was implemented but not fleshed out at this stage.

There was also a separator widget, which was a Tkinter Separator object, to provide a split down the middle of the window just to make the appearance a little nicer for the user.

#### Placement

The widgets on the window were placed using relative scaling, so that their size and/or position would change in real time according to the size of the root window, which could be dragged and changed by the user.

#### Functions

There were numerous event-driven functions that needed to be implemented, which were usually called whenever a button was clicked:

* save\_midi(midi): Function to save a loaded or generated midi file to the local storage area.
* upload\_file(): Function to upload a midi file to the program (which ended up in turn calling the “load\_music” function.
* load\_music(): Function to load a song into a Pygame mixer (See section 4.2.3).
* generate\_music(midi): Function to generate music from a given MIDI file, at this stage does not do anything.
* get\_sheet\_music(midi\_file): Function to generate guitar tab from a given MIDI file, using the “tayuya” library (See section 4.2.3).
* pause\_music(): Function to pause a current song.
* play\_music(): Function to play a current song, and also to replay it if called while a song is already playing.
* volume\_down(): Function to reduce the volume by 0.1.
* volume\_up(): Function to increase the volume by 0.1.

Graphical user interface

Description automatically generated

Figure 11: The final look of the GUI after the second sprint

### Implementation

Firstly, the outline of the GUI was created, and the major widgets were implemented in their basic forms. Following this, the placement of the widgets became the priority, as my original method of using Tkinter methods such as “grid()” and “pack()” were simply not cut out for specific placements of multiple widgets and changing window sizes. I therefore had to switch to using the “place()” methods, which allowed for the use of relative scaling, meaning that the widgets would scale correctly with the size of the window. The downside was that if the window got too small that they overlapped, and if it got too big then some of them looked slightly off-centre, but other than those small problems it worked pretty well.

Focus then turned to the functions. I realised that I would probably need to use some third-party libraries to help with these, such as generating sheet music from MIDI files and getting them to play. I used two separate libraries for each:

#### Pygame

Pygame is a collection of modules for Python that were originally designed for programming video games, and can be installed via Anaconda or with pip[29]. The module within Pygame that was useful for this project is the “mixer” module, which is able to play MIDI sound files. This was therefore vital for the music-playing part of the project.

#### Tayuya

This Python library assists in the conversion of MIDI files to guitar tablature, which was useful for this project for generating the written music from generated MIDI files. When implementing this library into my program, I realised that it generates tabs per track on the MIDI file, so I had to develop a way to loop through each track on the MIDI and render each one each loop. There was scope for quite a few errors here, so I implemented around for try/catch boundaries for this process. This Is because one of the main functions used in the library, “render\_tabs()”, does not return the tabs as string, it just prints them, which was not ideal, but it was managed.

#### Problems

I encountered a few problems during the development of the GUI. Firstly, the Spyder IDE proved to be temperamental at best, however fortunately for what I was doing in this sprint I was able to switch to IDLE and use pip to install the necessary libraries. As mentioned, I had some general problems with the placement of widgets, but I managed to mostly fix these, although labels and buttons do overlap if the window is too small. I was unable to fully implement the “save\_midi” function, which allows the user to save MIDI files back to the local file area. This would be something to look at again in the third sprint. Furthermore, the rewind function does not work, but I believe this was a Pygame issue so was unlikely to be fixed.

#### Global Variables

I had to use some global variables throughout the application, which could have been eliminated had I used an OOP approach. These were the volume and the loaded midi file (and its name).

### Testing

Full test tables and screenshots for these tests are in Appendix D, and can be located as per the Test ID. For general interface testing, tests for this sprint start at Test\_1 up to and including Test\_14. For error testing, tests for this sprint start at Test\_12 up to and including Test\_14

### Review

In summary for this sprint, most of the objectives have been achieved to an acceptable standard. There were some problems as mentioned in section 4.2.3, but these were not major problems that affected the core functionality of the program, so this can be considered a successful sprint.

## Sprint 3: 29th April – 4th May

### Plan and Objectives

It was decided to use and modify part of a third-party library to help bridge the gap between my initial GUI program and the backend functionality of the generative model. This was the library regarding LSTM generation used in Section 2.3, however a different section, which allows for fitting models with pre-trained weights without the need to do the heavy processing of training a model from scratch. However, users should be able to fit their own pre-trained weights rather than the one that was generated by me if they want, it would just have to match the architecture of the model.

The plan, however, was to alter this module so that it could be turned into a class, with some extra methods and a helper class. There were a few reasons for this; firstly so that it was more structured than the original, which was a just a collection of functions. Secondly, it would allow for easier adjustments to the specific variables contained within the methods, as these could be assigned when the object is initialised, and then assigned accordingly. This removes the need for global variables, making the project cleaner as a whole. Thirdly, converting the function into class methods makes it easier for me or other people to make changes to the original code, model architecture, class attributes etc more easily, and it can be imported and used for other projects in the future.

In terms of the GUI, I needed to create a window for the music generation part of the program, and connect it to the aforementioned class so that the user could enter the values needed to generate each new song. There should be checks to make sure the values the user enters are reasonable, and that the generated midi is loaded back to the main player so that the user can listen to it back.

### Design

#### Generator Class

The Generator class is the class that combines all of the original functions into one callable object. It also has an extra method that was added to try and implement a custom input during the generation process. The Generator class therefore contains the following methods:

* generate(): A method initially from the library that uses the other methods in the class to generate the new MIDI, however it was altered so that it would return the file-path of the new generated midi that could be used by the GUI.
* prepare\_sequences(notes, pitchnames, n\_vocab): A method initially from the library that prepares the sequences to be run through my pre-trained model, which was mostly unaltered from the original.
* create\_network(network\_input, n\_vocab): A method initially from the library that builds the model architecture and loads the weights. This was modified so that the architecture would match the weights I produced and was slightly lighter so that it would not take so long to train.
* generate\_notes(model, network\_input, pitch\_names, n\_vocab): A method initially from the library that, as the name suggests, generates the notes for the new MIDI file. Part of this method was modified so that the starting sequence of notes could be user input.
* create\_midi(prediction\_output): A method originally from the library that converts the generated notes back to a MIDI file and saves it back to the local drive, although it was altered to also return the filename for use in the main GUI.
* midi\_to\_notes(midi): A method that I created and appended to the class based off another part of the library, which would convert a given MIDI file input by the user, and use it as the starting sequence of the generative model to try and vary the output.
* get\_midi\_info(): This is a small method I wrote that might help for testing purposes, which makes use of the MidiInfo class to generate information about each MIDI file provided.

#### MidiInfo Class

This is a small class written by me to provide information about given MIDI files. This class is used by the Generator Class. The MidiInfo class contains the following methods:

* get\_file\_info(): This method returns the information about a given MIDI file, output to the Python shell.
* get\_track\_messages(track): This method returns information about a specific track on a MIDI file, specifically the messages it contains, indicated by an input track number. The output is to the Python shell.
* Get\_num\_messages(track): This method returns the number of messages within a given track on the MIDI.

#### GUI Generation Window

The generation window is a small popup window that allows the user to input specific options for the generator. The title of the window is “Generation Settings” and its default initial size is 300x120 pixels. As there are three settings that the user can change, the window consists of three entry boxes, all labelled, and a button to confirm the values. All of the widgets featured in this window, as with the root window, are positioned relative to the size of the window.

#### Entry Labels

* temp\_label: Label to indicate the entry box for where the temperature can be altered.
* gen\_length\_label: Label to indicate the entry box where the user can enter the number of notes they wish to generate.
* seq\_length\_label: Label to a indicate the entry box where the user can alter the length of the sequences fed into the model to try and alter the output.

#### Entry Boxes

* temp\_entry: The entry box for the temperature.
* gen\_length\_entry: The entry box for the number of notes the user wants to generate.
* seq\_length\_entry: The entry box for the length of each sequence the user would like to use

#### Confirmation Button

This is the button that confirms all the values and calls a nested function to initiate the Generator object with the values in the entry boxes as the given input.

Graphical user interface, application

Description automatically generated

Figure 12: The final Generation Settings window

### Implementation

The implementation for this stage began with building the Generator class. Since most of the code I was using was from the library, it did not prove to be much of an issue to convert the functions into methods and build the class around them. Problems did occur however surrounding the new method that I implemented regarding using a given midi file as an input for the generation. The issue was to do with process of converting the one MIDI file to an array of notes that could be used as the starting pattern, and I was unable to fix it before the end of the sprint, so as of this report it remains a bug.

Furthermore, there was not time in the end to implement a different model into the program for the user to use. For the superior GAN model (see section 2.4), there were major compatibility issues with the Magenta library, and I was unable to train my own model using Magenta. A lot of these issues came from the fact I was developing on a Windows machine, and while Magenta is compatible with windows, it is more suited for Linux development. Despite, going as far as downloading Linux for Windows, unfortunately it was to no avail.

The implementation for the GUI window and linking it to the generator class was managed again without too many difficulties. The model produces the results quickly, and the new MIDI is able to be played within a short time, as well as being saved to the file area. This does alleviate the need for the save button however, leaving it a redundant option, along with some of the other cascades on the menu bar, which as of this report mainly serve for aesthetic purposes only.

The MidiInfo class was relatively straightforward to implement as it is just a small, simple class that is not used massively in the main program other than for testing purposes.

### Testing

Full test tables and screenshots for these tests are in Appendix D, and can be located as per the Test ID. For general interface testing, tests for this sprint start at Test\_15. For error testing, tests for this sprint start at Test\_21.

### Review

In a practical sense, many of the objectives set out at the beginning of this sprint were indeed met. The program as a whole is able to fulfil the duty of using a model that I trained to generate music that does not sound terrible, and there is some (albeit limited) input that the user is able to provide to change the overall output of the model. The GUI is, while simple, smooth and performs all the core tasks well. On the other hand, there are numerous features that I would have liked to have implemented that I could not, such as the Magenta model, and there are a small number of remaining bugs, but these do not affect the overall running of the program too much.

After testing, in particular the stress testing, a large majority of the tests did not fully pass, but were not fatal to the program. If the user were not to look at the Python shell, they would never notice that they had caused an error. This is not a good thing overall, as a system for having warnings appear in the GUI should have been implemented to warn the user if they are doing something wrong.

# Testing

## Overall Approach to Testing

As part of my chosen process, I performed manual tests at the end of each sprint. The following table is an example of what my testing strategy looked like; the full test tables can be found in Appendix D:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test ID | Description | Steps to perform | Inputs | Expected Result | Actual Result | Pass/Fail |
| Test\_7 | Check that the volume-up button works | Load a song, play it, and increase the volume three times | A MIDI file to play | The volume of the audio will increase, as well as the number on the volume label | Screenshot for Test\_7  Volume did audibly increase | Pass |

The “Test ID” field is to identify the test and relate it to the screenshot table if needed. There is then a brief description regarding the test, and the steps the user or tester can take to replicate it. If there are specific inputs needed for the test, then this is clarified, and there are then fields for the expected and actual results, for the reader to compare. The “Actual Result” field may just reference a screenshot, or give a description of the test outcome, or both (as is the case with the above example). The final field indicates whether the test was a pass or a failure, although on some occasions tests have only partially failed.

Two types of manual testing were performed. The first was general interface testing. These tests covered the basic functionality of the application, for instance making sure all of the widgets load correctly, making sure that each button works properly, and generally helping to measure how well the application fit the objectives devised at the start of each sprint. The second kind of testing was erroneous testing, which covered things like bad inputs and how the application handled these errors.

## Automated Testing

Text

Description automatically generatedI performed some limited automated unit testing, mostly regarding the Generator class. Whilst Python does have libraries available specifically for automated testing, the tests I performed were relatively simple, and just to test the creation of the class and how it could handle bad inputs. The full code for these tests is in Appendix B.

Figure 13: A test written for the Generator class

## Integration Testing

Due to the process I decided upon, the integration testing for this project was merged into the general interface testing, as the full application was combined and integrated in Sprint 3. The full test tables for this can be found in Appendix D.

## User Testing

Unfortunately, all of the user testing was conducted by me. In hindsight, since the part of the nature of this application is to help people gain a better understanding of music generation and machine learning, performing even some minor user testing from an external user would have been advantageous. However, this is always something that could be worked upon in the future, as the insight gained would be valuable.

# Critical Evaluation

Examiners expect to find a section addressing questions such as:

* Were the requirements correctly identified?
* Were the design decisions correct?
* Could a more suitable set of tools have been chosen?
* How well did the software meet the needs of those who were expecting to use it?
* How well were any other project aims achieved?
* If you were starting again, what would you do differently?

Other questions can be addressed as appropriate for a project.

* Heavy leaning on third party libraries
* Achieved basic functionality
* Much more options needed
* Research on LSTMs was good but could have tried to build a GAN model
* Spyder and Colab useful but annoying
  + GUI simple and functional, but not great looking
  + Icon etc
* Should have made the whole project OOP(use of globall variables) etc, however this didn’t affect the final application too much
* MidiInfo could have been fleshed out more
* Manual testing was fine but more automated tests needed
* Starting again:
  + Start with an OOP approach
  + Implement the ability to use different models
  + Implement the ability to train own model(this comes with risks)
  + Implement some kind of display
* Ultimate goal would be to assist the musician, not to do it for them

## Objectives and Requirements

This project was a mixture between a research-based project and an engineering-based project. My initial goal was to produce software that based on my research that would allow people who did were into music production but perhaps did not have much experience with machine learning or AI to generate their own songs

# References

This final section should list all relevant resources that you have consulted in researching your project.

1. Krishnan, S., Lima, C.F., Evans, S., Chen, S., Guldner, S., Yeff, H., Manly, T. and Scott, S.K., 2018. Beatboxers and guitarists engage sensorimotor regions selectively when listening to the instruments they can play. Cerebral Cortex, 28(11), pp.4063-4079.
2. Blood, A.J. and Zatorre, R.J., 2001. Intensely pleasurable responses to music correlate with activity in brain regions implicated in reward and emotion. Proceedings of the national academy of sciences, 98(20), pp.11818-11823.
3. Leaver, A.M., Van Lare, J., Zielinski, B., Halpern, A.R. and Rauschecker, J.P., 2009. Brain activation during anticipation of sound sequences. Journal of neuroscience, 29(8), pp.2477-2485.
4. Tham, I., 2021. Generating Music using Deep Learning. [online] Medium. Available at: <https://towardsdatascience.com/generating-music-using-deep-learning-cb5843a9d55e#:~:text=Recently%2C%20Convolutional%20Neural%20Networks%20(CNNs,convolutions%20to%20generate%20raw%20audio.> [Accessed 13 March 2022].
5. Skuli, S., 2017. How I Built A Lo-fi Hip-Hop Music Generator. [online] Medium. Available at: <https://ai.plainenglish.io/building-a-lo-fi-hip-hop-generator-e24a005d0144> [Accessed 21 March 2022].
6. Medium. 2020. How I Built A Lo-fi Hip-Hop Music Generator. [online] Available at: <https://ai.plainenglish.io/building-a-lo-fi-hip-hop-generator-e24a005d0144> [Accessed 19 March 2022].
7. Graves, A., 2012. Long short-term memory. Supervised sequence labelling with recurrent neural networks, pp.37-45.Hochreiter, S., 1998. Recurrent neural net learning and vanishing gradient. International Journal Of Uncertainity, Fuzziness and Knowledge-Based Systems, 6(2), pp.107-116.
8. Hochreiter, S., 1998. Recurrent neural net learning and vanishing gradient. International Journal Of Uncertainity, Fuzziness and Knowledge-Based Systems, 6(2), pp.107-116.
9. University, I., 2022. The MIDI Standard: Introduction to MIDI and Computer Music: Center for Electronic and Computer Music: Jacobs School of Music. [online] Cecm.indiana.edu. Available at: <https://cecm.indiana.edu/361/midi.html> [Accessed 29 March 2022].
10. Fisher, T., 2022. What's a MIDI File and How Do You Open One?. [online] Lifewire. Available at: <https://www.lifewire.com/midi-file-2621979> [Accessed 13 April 2022].
11. Yegulalp, S., 2019. What is TensorFlow? The machine learning library explained. [online] InfoWorld. Available at: <https://www.infoworld.com/article/3278008/what-is-tensorflow-the-machine-learning-library-explained.html> [Accessed 17 April 2022].
12. Graves, A. and Schmidhuber, J., 2005, July. Framewise phoneme classification with bidirectional LSTM networks. In Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005. (Vol. 4, pp. 2047-2052). IEEE.
13. Hochreiter, S. and Schmidhuber, J., 1997. Long short-term memory. Neural computation, 9(8), pp.1735-1780.
14. Yegulalp, S., 2017. Facebook brings GPU-powered machine learning to Python. [online] InfoWorld. Available at: <https://www.infoworld.com/article/3159120/facebook-brings-gpu-powered-machine-learning-to-python.html> [Accessed 21 April 2022].
15. Terra, J., 2022. [online] Available at: <https://www.simplilearn.com/keras-vs-tensorflow-vs-pytorch-article#:~:text=TensorFlow%20is%20an%20open%2Dsourced,because%20it's%20built%2Din%20Python.> [Accessed 13 April 2022].
16. Magenta. 2022. Magenta. [online] Available at: <https://magenta.tensorflow.org/> [Accessed 10 April 2022].
17. Sidor, S., 2020. magenta/magenta/models at main · magenta/magenta. [online] GitHub. Available at: <https://github.com/magenta/magenta/tree/main/magenta/models> [Accessed 10 April 2022].
18. Dominodatalab.com. 2022. What is Anaconda? | Domino Data Science Dictionary. [online] Available at: <https://www.dominodatalab.com/data-science-dictionary/anaconda> [Accessed 25 April 2022].
19. Brownlee, J., 2019. A Gentle Introduction to Generative Adversarial Networks (GANs). [online] Machine Learning Mastery. Available at: <https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/> [Accessed 24 April 2022].
20. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. and Bengio, Y., 2014. Generative adversarial nets. Advances in neural information processing systems, 27.
21. McCormick, C., n.d. Melody Generator. [online] Melody Generator. Available at: <https://dopeloop.ai/melody-generator/?s=7019330677856559> [Accessed 1 May 2022].
22. Soundraw.io. n.d. Soundraw. [online] Available at: <https://soundraw.io/#about-soundraw> [Accessed 1 May 2022].
23. PyPI. n.d. PyQt5. [online] Available at: <https://pypi.org/project/PyQt5/> [Accessed 1 May 2022].
24. Amos, D., 2022. Python GUI Programming With Tkinter – Real Python. [online] Realpython.com. Available at: <https://realpython.com/python-gui-tkinter/> [Accessed 1 May 2022].
25. Sutherland, J., 1993. What is Scrum?. [online] Scrum.org. Available at: <https://www.scrum.org/resources/what-is-scrum> [Accessed 1 May 2022].
26. Lovelace, A., 1843. Analytical Engine :: Museum of Imaginary Musical Instruments. [online] Imaginaryinstruments.org. Available at: <http://imaginaryinstruments.org/lovelace-analytical-engine/> [Accessed 2 May 2022].
27. L. Casini, G. Marfia and M. Roccetti, "Some Reflections on the Potential and Limitations of Deep Learning for Automated Music Generation," 2018 IEEE 29th Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC), 2018, pp. 27-31, doi: 10.1109/PIMRC.2018.8581038.
28. Briot, JP., Pachet, F. Deep learning for music generation: challenges and directions. Neural Comput & Applic 32, 981–993 (2020). <https://doi.org/10.1007/s00521-018-3813-6>
29. Pygame.org. n.d. *About - pygame wiki*. [online] Available at: <https://www.pygame.org/wiki/about> [Accessed 3 May 2022].
30. Surkis, L., 2019. *How to Generate Techno Music using Deep Learning*. [online] Medium. Available at: <https://medium.com/@leesurkis/how-to-generate-techno-music-using-deep-learning-17c06910e1b3> [Accessed 2 May 2022].
31. Karpathy, A., 2015. *The Unreasonable Effectiveness of Recurrent Neural Networks*. [online] Karpathy.github.io. Available at: <http://karpathy.github.io/2015/05/21/rnn-effectiveness/> [Accessed 3 May 2022].
32. Mangal, S., Modak, R. and Joshi, P., 2019. Lstm based music generation system. *arXiv preprint arXiv:1908.01080*.
33. Sigurgeirsson, S., 2019. *Classical-Piano-Composer/predict.py at master · Skuldur/Classical-Piano-Composer*. [online] GitHub. Available at: <https://github.com/Skuldur/Classical-Piano-Composer/blob/master/predict.py#L2> [Accessed 2 May 2022].
34. Google Research. n.d. *MAESTRO – Google Research*. [online] Available at: <https://research.google/tools/datasets/maestro/> [Accessed 1 May 2022].
35. GitHub. 2020. *GitHub - lucasnfe/adl-piano-midi: A dataset of piano midi files organized by genres, sub-genres and artists.*. [online] Available at: <https://github.com/lucasnfe/adl-piano-midi> [Accessed 3 May 2022].
36. Ali, S., DiPaola, D. and Breazeal, C., 2021, May. What are GANs?: Introducing Generative Adversarial Networks to Middle School Students. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 35, No. 17, pp. 15472-15479).
37. Donahue, C., McAuley, J. and Puckette, M., 2018. Adversarial audio synthesis. *arXiv preprint arXiv:1802.04208*.
38. Ferreira, L., Lelis, L. and Whitehead, J., 2020, October. Computer-generated music for tabletop role-playing games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (Vol. 16, No. 1, pp. 59-65).
39. Sharma, V., 2019. *GitHub - vipul-sharma20/tayuya: Python library to generate guitar tabs from MIDI files*. [online] GitHub. Available at: <https://github.com/vipul-sharma20/tayuya> [Accessed 4 May 2022].
40. Web.mit.edu. n.d. *Authors, Acknowledgments, Contributing, and Licensing — music21 Documentation*. [online] Available at: <https://web.mit.edu/music21/doc/about/about.html> [Accessed 4 May 2022].
41. Bjørndalen, M., n.d. *License — Mido 1.2.10 documentation*. [online] Mido.readthedocs.io. Available at: <https://mido.readthedocs.io/en/latest/license.html> [Accessed 4 May 2022].
42. Colin Raffel and Daniel P. W. Ellis. Intuitive Analysis, Creation and Manipulation of MIDI Data with pretty\_midi. In 15th International Conference on Music Information Retrieval Late Breaking and Demo Papers, 2014.

# Appendices

* 1. Third-Party Code and Libraries

#### TensorFlow Library

TensorFlow has been used throughout this project, mainly in building and training models, and is vital in the process of generating music. This library is open source and is available at tensorflow.org[11], and is licensed using the Apache 2.0 License. This library was used without modification.

#### Classical-Piano-Composer by Skuldur

This module has helped in this project during the experimental stage in section 2.3, and final application. The code is available from his public GitHub[33], and is licensed using the MIT License. Credit has been given in the source code where it is used. I have modified some of the code provided by putting some of the functions in “lstm.py” into the Generator class and changing the architecture of the models used, as well as using my own generated weights.

#### Tayuya library by Vipul Sharma

This library was used in my application to generate guitar tab from a given MIDI file. The library is available from his public GitHub[39], and is licensed using the MIT License. Credit has been given in the source code where it is used. This library was used without modification.

#### Music21 Library

Music21 is an open source library and was used when handling MIDI files and converting them into notes. Music21 is available at <https://web.mit.edu/music21/> and is licensed under the BSD License[40]. This library was used without modification.

#### Pygame library

Pygame is an open source library, and was used as part of my application to play the MIDI files, specifically the mixer module. Pygame is available at <https://www.pygame.org/> and is released under the LGPL license.

#### Mido library

The Mido library was used for the MidiInfo class to help get information about MIDI files. Mido is available at <https://mido.readthedocs.io/en/latest/>, and is licensed under the MIT license[41].

#### Magenta library

Magenta was used within my experiments in section 2 for its MelodyRNN, and I would have liked to use it in my application, however it did not make it. Magenta is open source, is licensed under the Apache 2.0 License, and the code is available at https://github.com/magenta/magenta/blob/main/ [16].

#### Pretty Midi library

Pretty Midi is a library that contains modules for handling MIDI data. It was used throughout my experimenting in section 2. Pretty Midi is licensed under the MIT License, and is available at <https://github.com/craffel/pretty-midi/blob/main/> [42].

* 1. Code Samples

#### Unit Tests for the Generator Class

def test\_generate():

file = "tremfals.mid"

try:

generator = Generator(file, 50, 50)

generator.generate()

print("test\_generate passed")

except Exception as e:

print(e)

print("test\_generate failed")

def test\_generate\_with\_wrong\_inputs():

file = "tremfals.mid"

print("Generating With Float Inputs...")

print("")

try:

generator = Generator(file, 10.2, 100)

generator.generate()

print("float inputs tests failed")

except Exception as e:

#print(e)

print("float inputs tests passed")

try:

generator = Generator(file, 25, 30.1)

generator.generate()

print("float inputs tests failed")

except Exception as e:

#print(e)

print("float inputs tests passed")

try:

generator = Generator(file, 25.2, 30.1)

generator.generate()

print("float inputs tests failed")

except Exception as e:

#print(e)

print("float inputs tests passed")

print("Generating With String Inputs...")

print("")

try:

generator = Generator(file, "HI", 100)

generator.generate()

print("string inputs tests failed")

except Exception as e:

#print(e)

print("string inputs tests passed")

try:

generator = Generator(file, 25, "HI")

generator.generate()

print("string inputs tests failed")

except Exception as e:

#print(e)

print("string inputs tests passed")

try:

generator = Generator(file, "TEST", "HI")

generator.generate()

print("string inputs tests failed")

except Exception as e:

#print(e)

print("string inputs tests passed")

def run\_tests():

print("Running Tests...")

print("")

test\_generate()

test\_generate\_with\_wrong\_inputs()

run\_tests()

* 1. LSTM Experiment Tables

### First LSTM Model Experiments Table

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Number | Description | Training Dataset | Architecture | Pre-generation Inputs | Training Time | Analysis |
| 0.0 | Using the tutorial provided by TensorFlow to provide a starting point for creating my own model. | The ‘maestro-v2.0.0’ dataset from Google, containing 1282 midi files and 7.13 million notes. | 1 Input Layer, 1 LSTM Layer connected to the Input Layer, and 3 Dense Layers connected to the LSTM Layer, for Duration, Pitch and Step.  There is an initial length of each sequence to be 50 notes, over 100 epochs | Temperature: 1.0  Input Notes:  From Dataset. | 1h 9min | The model generated a relatively fast-paced melody, with some repetition in the notes providing at least some basic consistency. |
| 0.1 | Changing the temperature setting to generate a more deviated output. | As above | As in 0.0 | Temperature: 2.0  Input Notes:  From Dataset. | As in 0.0 | The change in temperature had little effect on the overall steps and duration of each note, resulting in a similar lively tempo, but the consistency of the pitch is much more varied, but lacks the consistency and progressiveness of the piece produced in Test 0.0. |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0.2 | Changing the input notes to a song not from the training set, instead using a jazz tune (hymmj.mid) | As above | As in 0.0 | Temperature: 1.0  Input Notes:  “hymmj.mid”  Comment:  Changed the number of notes produced to double to 120 | As in 0.0 | The change in the initial input notes slightly affected the mood of the produced tune, with added higher notes and a consistency with two main lower notes. However, the duration and step values again seem to have no real change. |
| 1.0 | Attempting to train the model using a different, larger dataset | The “ADL Piano MIDI”, consisting of 11086 piano pieces from a large range of genres | As in 0.0 running over 30 epochs | Temperature: 1.0  Input Notes:  From Dataset | 27min | The resulting output was extremely poor, with almost no variation in each note’s pitch or duration, and was only marginally better regarding the steps. The loss over time also had very little degradation, and it seems the algorithm stopped at a local rather than global minimum. |
| 1.1 | Increasing the temperature to try and get a more varied result | As in 1.0 | As in 1.0 | Temperature: 3.0  Input Notes:  From Dataset | As in 1.0 | A little more variety in the pitch but not too much, very slow tempo with a creepy sound. |
| 1.2 | Further increase of temperature | As in 1.0 | As in 1.0 | Temperature: 5.0  Input Notes: From Dataset | As in 1.0 | Again, this sounded slightly better, but increasing the temperature would remove some of the cohesion between notes that the model tries to achieve. A common theme seems to be the massive lack of variety in duration, with it mostly being around 1.5 seconds for each note, which in turn increases the steps and decreases the tempo |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1.3 | Changing the initial input to a song not from the original dataset (‘tremfals.mid’) | As in 1.0 | As in 1.0 | Temperature: 5.0  Input Notes:  ‘tremfals.mid’ | As in 1.0 | The beginning at least sounded fairly cohesive, and would probably sound ok if the majority of the durations of each note was not so high. |
| 1.4 | Reducing the sequence length to try and improve the productivity of the LSTM, and Reducing the training dataset to just one music genre in order to try and generate a more refined piece, along with more epochs | From within the same dataset from 1.0, just the Classical genre will be used, reducing the dataset to 1398 songs | As in 1.0, over 60 epochs | Temperature: 1.0  Input Notes:  ‘tremfals.mid’ | 36min | The change in the dataset to be the same genre looked like it had pretty much no effect on the performance of the model, again generating a poor music sequence |
| 1.5 | Increasing the temperature to try and get a more varied result | As in 1.4 | As in 1.4 | Temperature: 5.0  Input Notes: ‘tremfals.mid’ | As in 1.4 | Again, a slightly more varied but uninspired and with no real direction. |

### Second LSTM Model Experiments Table

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2.0 | Changing the model architecture to achieve different results | From within the same dataset from 1.0, just the Folk genre will be used, reducing the dataset to 61679 notes | 3 LSTM layers, 3 dropout layers, 2 dense layers and 1 activation layer, trained over 50 epochs | A random starting sequence from the training data, all with a length of 50 | 3 hours 44 mins | The music produced is a marked improvement from any of the music generated previously as of this test. As can been seen from the distribution of notes, there is much greater variety, with notes being played at different durations, multiple steps, and sometimes at the same time, the latter something not seen before. There remains a large amount of repetition, however compared to the other tests this lies in clear contrast. |

### Magenta Model Experiments Table

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3.0 | Testing the pre-trained RNN model to see if it is any better than the model I trained in 2.0 | Pre-trained, so I just used the “tremfals.mid” file as the initial input | Pre-trained | Temperature: 1.0  200 Notes to be produced | Pre-trained | This model is supposed to generate music specifically based on the starting input. The original song has a very specific repetition of a certain note, but the produced sound does not replicate this. So, while the notes produced are coherent, they do not reflect the initial input that well at all. |

### Screenshots

|  |  |  |
| --- | --- | --- |
| Test Number | Loss over time | Pitch, Step and Duration Distributions |
| 0.0 | A picture containing chart  Description automatically generated | Histogram  Description automatically generated |
| 0.1 | N/A | Chart, histogram  Description automatically generated with medium confidence |
| 0.2 | N/A | A picture containing histogram  Description automatically generated |
| 1.0 | Chart, line chart  Description automatically generated | A picture containing chart  Description automatically generated |
| 1.1 | N/A | Chart, histogram  Description automatically generated |
| 1.2 | N/A | Chart  Description automatically generated with medium confidence |
| 1.3 | N/A |  |
| 1.4 | Chart, line chart  Description automatically generated | Histogram  Description automatically generated with low confidence |
| 1.5 | N/A | Chart, histogram  Description automatically generated |

|  |  |  |
| --- | --- | --- |
| Test Number | Loss Over Time(epochs) | Distribution of Notes Over Time |
| 2.0 | A picture containing chart  Description automatically generated | Chart, scatter chart  Description automatically generated |

* 1. Test Tables

#### General Interface Testing

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test ID | Description | Steps to perform | Inputs | Expected Result | Actual Result | Pass/Fail |
| Test\_1 | Check that on start-up, the application and widgets load correctly | Run the application | None | Application loads with all implemented widgets visible | Screenshot for Test\_1 | Pass |
| Test\_2 | Check that all the options under the “File” cascade menu are displayed | Run the application, select the “File” option in the menu bar | None | The “File” cascade will appear, revealing the four options within it | Screenshot for Test\_2 | Pass |
| Test\_3 | Check that the upload file button works | Select the “Upload File” button, select a MIDI file from the pop-up | A MIDI file from the pop-up menu, specifically “Beatles\_Blackb-ird.mid” | The selected file will be successfully loaded, and displayed by the label in the centre of the application window | Screenshot for Test\_3 | Pass |
| Test\_4 | Check that the play button works | With a song selected, press the play button and make sure the song plays | A MIDI file to play | The selected MIDI file will play | File played as expected | Pass |
| Test\_5 | Check that the pause button works | While a loaded song is playing, press the pause button and make sure the music stops playing | A MIDI file to play | The Selected MIDI file that is in the process of playing will pause | File paused as expected | Pass |
| Test\_6 | Check that the play button will restart a current song | While a loaded song is playing, press the play button again and make sure the music restarts | A MIDI file to play | The selected MIDI file in the process of playing will restart | File did not restart as expected, just kept playing even after multiple attempts | Fail |
| Test\_7 | Check that the volume-up button works | Load a song, play it, and increase the volume three times | A MIDI file to play | The volume of the audio will increase, as well as the number on the volume label | Screenshot for Test\_7  Volume did audibly increase | Pass |
| Test\_8 | Check that the volume-down button works | Load a song, play it, and decrease the volume two times | A MIDI file to play | The volume of the audio will decrease, as well as the number on the volume label | Screenshot for Test\_8  Volume did audibly decrease | Pass |
| Test\_9 | Check that the volume cannot exceed a value of 1.0 | Load a song, play it, and increase the volume to maximum, then continue pressing the volume-up button | A MIDI file to play | The volume of the audio will audibly stay the same after it reaches the max, and the value on the volume label will remain at 1.0 | Volume did stay the same and value on the label did remain at 1.0 | Pass |
| Test\_10 | Check that the volume cannot go below 0.0, and when it hits that mark the volume label should change to “Mute” | Load a song, play it, and decrease the volume to minimum, then continue pressing the volume-down button | A MIDI file to play | The volume of the audio will audibly vanish and remain that way, and the label should change to “Mute” | Volume did turn off, but the volume displayed 0.0 before displaying “Mute” | Partial Failure |
| Test\_11 | Check to see if the “Generate Sheet Music” button works | Load a MIDI file, then click on the “Generate Sheet Music” button | A MIDI file to load | The guitar tablature will be printed into the shell, for each available track on the MIDI | Tab was generated as expected | Pass |
| Test\_15 | When the “Generate Music” button is clicked without a MIDI file loaded, the settings window should not open | Run the application, click on the “Generate Music” button | None, a non-MIDI file | Nothing should happen in the GUI | Nothing happened as expected when nothing was loaded, but when a non-MIDI file was loaded it opened the window | Partial failure |
| Test\_16 | When a MIDI file is loaded and the “Generate Music” button is clicked, it should open a the settings window with all the necessary widgets | Upload a MIDI file and then click on “Generate Music” | A MIDI file | The window with the correct widgets should pop-up, with the entry widgets having pre-determined values in them | Screenshot for Test\_16 | Pass |
| Test\_17 | When “Confirm Values” is clicked, a new MIDI file named “midi-gen.mid” should be created and available to play | Upload a MIDI file, click on “Generate Music”, and then confirm the default values | A MIDI values, the default values in the Generation Settings window | A new MIDI is displayed and able to be played | Screenshot for Test\_17  New MIDI was generated quickly and effectively | Pass |
| Test\_18 | Check that the generation length can be changed | Upload a MIDI file, click on “Generate Music”, and then change the length of the generation to 200 instead of the default 100 | A MIDI file and the generation length set to 200 | A new MIDI is displayed, and its length is twice as long as the default one | Screenshot for Test\_18  New MIDI of the correct length was generated | Pass |
| Test\_19 | Check that the sequence length can be changed | Upload a MIDI file, click on “Generate Music”, and then change the sequence length to 25 | A MIDI file with the sequence length set to 25 | A new MIDI is successfully generated and displayed | Screenshot for Test\_19  New MIDI was successfully generated | Pass |
| Test\_20 | Check that the temperature can be changed | Upload a MIDI file, click on “Generate Music”, and then change the temperature to 3.0 | A MIDI file with the temperature set to 3.0 | A new MIDI is successfully generated and displayed, although the temperature value should not have any affect anyway | Screenshot for Test\_20 | Pass |

#### Error Testing

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test ID | Description | Steps to perform | Inputs | Expected Result | Actual Result | Pass/Fail |
| Test\_12 | Check to see what happens if a non-MIDI file is uploaded | Click the upload file button, and select a file type that is not MIDI | A non-MIDI file | The label should show the file name but state that it cannot be played | The label did as expected | Pass |
| Test\_13 | Check to see what happens when “Generate Sheet Music” is clicked when a MIDI file is not loaded | Upload a non-MIDI file, and then click “Generate Sheet Music” | A non-MIDI file | Nothing should happen, other a warning in the Python shell | Three warnings were displayed in the Python shell, but other than it was as expected | Pass |
| Test\_14 | Check to see that when the window is resized the widgets scale accordingly | Run the application, then resize the window to be as big as possible, then as small as possible | None | The widgets should scale according to the size of the window, but it is expected they will overlap when window becomes too small | Widgets scaled and overlapped as expected, but overlapping counts as a partial failure | Partial Failure |
| Test\_21 | Check to see if an integer value can be put in temperature box as input | Upload a MIDI, click on “Generate Music”, and then change the temperature to 3 | A MIDI file with the temperature set to 3 | The generation should proceed as normal, the temperature should be converted to float | Generation proceeded as normal | Pass |
| Test\_22 | Put a float in the generation length box | Upload a MIDI, click on “Generate Music”, and then change the generation length to 4.8 | A MIDI file with generation length set to 4.8 | The program should not allow the generation process to start | Screenshot for Test\_22  The program did not allow the generation to start, but an error was thrown in the Python shell | Partially Failed |
| Test\_23 | Put a float in the sequence length box | Upload a MIDI, click on “Generate Music”, and then change the sequence length to 27.9 | A MIDI file with sequence length set to 27.9 | The program should not allow the generation process to start | Screenshot for Test\_23  The program did not allow the generation to start, but an error was thrown in the Python shell | Partially Failed |
| Test\_24 | Put a string in the temperature box | Upload a MIDI, click on “Generate Music”, and then change the temperature to “t” | A MIDI file and a temperature set to “t” | The program should not allow the generation process to start | Screenshot for Test\_24  The program did not allow the generation to start, but an error was thrown in the Python shell | Partially Failed |
| Test\_25 | Put a string generation length box | Upload a MIDI, click on “Generate Music”, and then change the generation length to “g” | A MIDI file with the generation length set to “g” | The program should not allow the generation process to start | Screenshot for Test\_25  The program did not allow the generation to start, but an error was thrown in the Python shell | Partially Failed |
| Test\_26 | Put a string in the sequence length box | Upload a MIDI, click on “Generate Music”, and then change the sequence length to “s” | A MIDI file with sequence length set to “s” | The program should not allow the generation process to start | Screenshot for Test\_26  The program did not allow the generation to start, but an error was thrown in the Python shell | Partially Failed |
| Test\_27 | Leave boxes empty | Upload a MIDI, click on “Generate Music”, and remove all of the default values that appear in the entry boxes | A MIDI file, and the temperature, generation length, and the sequence length set to None | The program should not allow the generation process to start | Screenshot for Test\_27  The program did not allow the generation to start, but an error was thrown in the Python shell | Partially Failed |

#### Screenshots

|  |  |
| --- | --- |
| Test ID | Screenshot |
| Test\_1 | Graphical user interface  Description automatically generated |
| Test\_2 | Graphical user interface, text, application  Description automatically generated |
| Test\_3 | Graphical user interface  Description automatically generated |
| Test\_7 | Graphical user interface, application, Word  Description automatically generated |
| Test\_8 | Graphical user interface, application, Word  Description automatically generated |
| Test\_10 | Graphical user interface, text, application  Description automatically generated |
| Test\_11 | Text  Description automatically generated |
| Test\_12 | Graphical user interface, application  Description automatically generated |
| Test\_14 | Graphical user interface, application  Description automatically generated |
| Test\_16 |  |
| Test\_17 |  |
| Test\_18 |  |
| Test\_19 |  |
| Test\_20 |  |
| Test\_22 |  |
| Test\_23 |  |
| Test\_24 |  |
| Test\_25 |  |
| Test\_26 |  |
| Test\_27 |  |