ECM3401: Individual Literature Review and Project
Algorithmic Generation of Novel House Music

1st May, 2019

Abstract

A primarily research-based project investigating the algorithmic generation of novel house music through the application of character-level recurrent neural networks. A multi-instrument encoding scheme is created to accomplish this, capable of accurately representing the input MIDI dataset. The quality of the generated music is objectively evaluated in an audience scenario, while the novelty is assessed using an innovative approach involving an LCS-based string metric.

I certify that all material in this dissertation which is not my own work has been identified

Contents

1 Introduction

Music has long been the target of algorithmic generation, with genres such as blues, classical and rock receiving an abundance of attention [1, 2, 3, 4]. Per contra, the electronic dance music (EDM) world has not been delved into and house music, a sub-genre of EDM, remains untouched. This lacuna can likely be attributed to the perceived complexity of the music. The blend of multiple diverse electronic instruments contributes to this, leading many to believe that it is too difficult to generate such intricate arrangements. In fact, house music is an ideal candidate for artificial synthesis. The repetitive tempo, lack of thematic variation and omission of vocals all contribute towards practicable pattern detection, a crucial ingredient in algorithmic generation. Furthermore, due to their digital production, house tracks are noise-free which reduces result distortion [4].

There are plenty of techniques for generating music. For instance, evolutionary computation and Markov models [5] both have foundations in academia and have proven to be viable approaches. In spite of this, this project explores the application of neural networks for music composition. Specifically, recurrent neural networks (RNN) [6] are recruited for the task, possessing valuable characteristics which make it preferable for music generation.

A key facet of the final outcome is song cohesion. Notes of a track should not be disjoint, but rather have some relation to its neighbours and predecessors. This ensures that the end product retains musical robustness, allowing it to sound pleasant to the human ear. Variations of neural networks, such as the RNN, can achieve this through the learning of long-term dependencies. However, there exist a multitude of obstacles to overcome before this is possible. In particular, the network input and its representation play a significant role in determining what the output sounds like, so it is essential that this is handled in an appropriate manner. Both MIDI and raw audio waveform representations shall be examined in this project. Similarly, the network architecture also strongly affects the outcome. The type and quantity of layers in the model determine how well notes can be predicted. Moreover, this factor helps to prevent occurrences of overfitting, a side-effect of training neural networks where parts of the input compositions are replicated.

Generating quality music is onerous, but generating *novel* music is an entirely different type of challenge. In this project, and most others of a similar nature, novelty implies *style imitation* [5], where the domain is restricted to specific genre – in this case, house. The contrary to this is *genuine composition*, where entirely unique creations of musical art are produced. The novelty of generated music falls within the scope of the research questions. The exact hypotheses to be addressed are as follows:

"How effectively can a neural network generate harmonious and musically coherent house music?"
"To what extent is the generated music novel?"

These questions shall be thoroughly evaluated but in order to reach this stage, there exist several preceding sections. First of all, a summary of the subject area and project specification shall be presented, allowing the report to be read as a self-contained document. Next, the design of the overall system is comprehensively detailed, at an abstract level. The development section goes into greater depth about *how* this system was created, specifically what prototypes were constructed and which challenges were encountered along the way. The final product then undergoes experimental analysis before being evaluated through several different means, as mentioned. The report finishes with a critical assessment of the project, along with a conclusion in which the project is summarised and future changes are suggested.

2 Summary of Literature Review & Project Specification

A summary of the literature review and project specification shall be presented in this chapter, for completeness.

2.1 Literature Review

This section presents an overview of the literature review in order to provide the necessary topical knowledge to wholly understand this paper. The literature review investigated several computational techniques for the artificial generation of music, with the project specification proposing that neural networks [7] were the optimal method in this instance. These systems are loosely modelled on biological neurons, with the objective of independently learning to perform specific tasks, given a set of input examples. For example, as in this project, a network can be used to predict musical sequences after being fed existing tracks.

An extensive range of research has been undertaken using neural networks within the musical field. Most notably for the scope of this project, Goodman and Pai [8] used deep networks to compose rock music, exploiting an original encoding, and Nayebi and Vitelli [4] operated on the waveforms of the musical data using variable network architectures.

The structure of an artificial neural network is somewhat flexible. All networks have an *input* layer, where nodes are supplied with an identical set of inputs, which are then multiplied by some arbitrary weight before having a bias constant added. The aggregate sum is then clamped using a nonlinear activation function. Networks also have an *output* layer, where the output variables are discharged. Between this and the input layer, there exist zero-to-many transitional *hidden* layers; existence of such layers categorises the model into the deep learning domain.

Once a network has been given an input, it is weighted accordingly and a cost (or loss) function determines the quality of the produced output, compared to the expected output. Backpropagation is then utilised to calculate the cost gradient and distribute it backwards through the network layers, before an optimisation method is used to revise the weights with the aim of *minimising* cost. Once the network has finished its training (i.e., the specified number of epochs has elapsed), these weights can be loaded in and the output sequences predicted.

There exist multiple variations of artificial neural networks. The most affluent being *feed-forward* neural networks, where information only flows from input to output. This is as opposed to a *recurrent* neural network, which has no directional constraint on the movement of information. Output from hidden layers can be recycled as a further input, allowing the network to utilise its internal state without incurring substantial overhead. This provides a sense of *memory*.

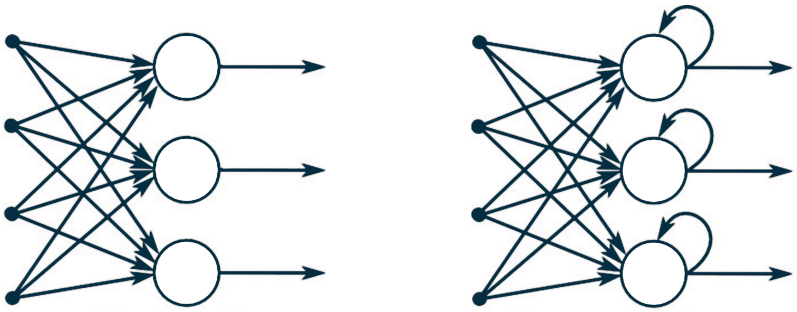

Figure 1: The structure of a feed-forward (left) and a recurrent (right) neural network

The memory capabilities can be further improved by employing Long Short-Term Memory (LSTM) units. These permit values to be propagated down more than just a singular time step, allowing long-term dependencies to be learnt. This is particular useful in generating music as it means global thematic structure can be transcended through the network. These units also help solve both the *exploding* and *vanishing* gradient problems [1, 9], improving network accuracy and avoiding the propagation of errors.

What makes neural networks so apt for novel generation is their ability to generalise [10], allowing the model to excel on unseen, unfamiliar data and not just on the training data. For the system to generalise well, the dataset needs to be divided into three constituent parts: a training set, to train the network; a validation set, for tuning hyperparameters; a test set, for evaluating

the full model to confirm its predictive power.

The input encoding and representation is a critical factor in determining the success of a model. The representation is "the act of presenting somebody/something in a particular way", while the encoding signifies how "to change information into a form that can be processed" by a network [11]. If either of these are handled incorrectly, the network will be unable to properly train and/or errors will be introduced. In this project, the two possible representations of the musical data include the raw audio waveform and the Musical Instrument Digital Interface (MIDI) format. Both have their advantages and disadvantages, discussed in more detail during the development stage, and have been used in previous studies [4, 8]. The encoding is dependent on which of these representations is chosen.

2.2 Project Specification

The promises of the project specification shall be listed below, with a brief insight into why not all of these turned out to be acceptable choices.

It was stated that recurrent neural networks would be used, with one or more hidden layers consisting of LSTM units. The input representation for this network was said to be the audio waveform of existing copyright-free house tracks. The dataset would consist of thirty of such tracks, assembled manually based on their popularity, and not contain vocal elements in order to reduce irregularities in results. This dataset would be split; 60% of which would be the training set and 20% for each of the validation and test sets. Training samples would be ten seconds in length, with training performed over the course of 2000 epochs. Python was declared to be the programming language of choice, coupled with the high-level neural network API, Keras [12], using a Theano backend. Amazon Web Services (AWS) instances were intended to be used for more efficient model training and for scalability reasons.

While the project focus remains consistent, with recurrent neural networks still being utilised, some other alterations have been made since supplementary practical knowledge has been gained in the subject area. These changes shall be outlined, with further precise justification supplied during the development section. First and foremost, the input representation used was in fact the MIDI format. The waveform representation was tested in the second prototype, being deemed inadequate for the end goal. The dataset changed as a result, to consist of 111 house MIDI files, and was segregated differently. In terms of training, the length of sequences was defined by the number of elements instead, at 100, and the number of epochs was reduced to 200. The Keras library was dropped in lieu of a lower-level approach, with the network being constructed using raw TensorFlow [13]. This was as opposed to Theano, because TensorFlow is more popular and has greater support within the AI community. The Google Cloud Platform was favoured ahead of AWS, due to its ease–of–use and ability to seamlessly adjust quotas.

The evaluation criteria, explained in the relevant section, largely has not changed, apart from the novelty evaluation technique which had to be altered due to the representational switch.

3 Design

As the development process advances and changes are made to each product iteration, the system will be further refined. In particular, factors such as the encoding and the recurrent neural network architecture cannot yet be determined. These will instead be assiduously depicted in later sections of the report.

A high-level overview of the proposed system environment is presented in Figure 2. In regard to it, the system components communicate via the labelled channels:

1. The input dataset is read into the Python implementation. Here, if necessary, it is converted into the appropriate representation and then encoded into a format suitable to be fed into the recurrent neural network. This model is constructed and setup through a neural network library (e.g. Keras or TensorFlow).

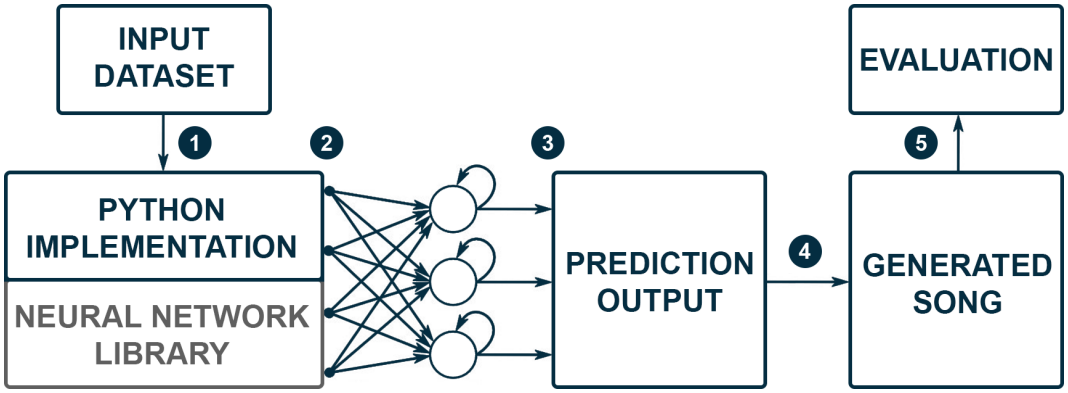

Figure 2: Abstract design of the system environment

2. The training data is delivered to the network, where it is then processed using the parameters (e.g., number of epochs, learning rate) already defined.
3. The network has finished training and is then sampled to obtain the prediction output.
4. This output is decoded in a pertinent manner, providing an algorithmically generated song in the correct format.
5. The music is then objectively evaluated, in a quantitative fashion. Its quality and novelty are assessed, in accordance to the aforementioned research hypotheses.

As the design evolves throughout the course of development, each stage will be analysed in greater depth.

4 Development

This section describes the development lifecycle in its entirety. There have been three major iterations of the product and each one shall be explained in detail, in regard to its design, the hurdles that were encountered and how the revisions differ from each other. Specifically, each iteration shall be divided into the following parts: its input representation, dataset, encoding, model preparation and architecture, training and finally, its generation.

4.1 Prototype One: Beethoven Emulation

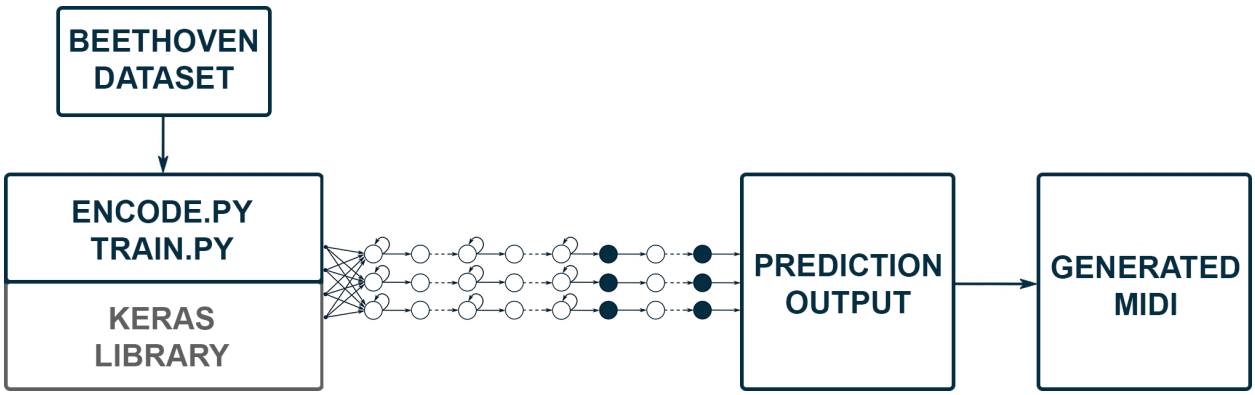

Figure 3: Refined design of the first prototype

The goal of this prototype was to implement the simplest system possible to generate some form of music, yet one which was exhaustive and covered all aspects of what the final product would entail. This first encompassed selecting an input representation, gathering a preliminary dataset and then encoding this input into the appropriate format.

4.1.1 Input Representation

For this prototype, the target was to generate classical music. This genre was chosen as it is less sophisticated than house music, with the complexity being further reduced by assuming just one instrument, the piano. For the representation of the classical music, the MIDI file format was selected.

The MIDI standard is prevalent within the music production industry, making it an obvious avenue to explore. A MIDI file consists of a collection of event messages, with each one specifying an action relating to the musical data. For example, a message may indicate turning a note "on", containing a variety of information about this note – for instance, it's notation, pitch and velocity. The messages are organised into parallel tracks, which may be synchronous or asynchronous, and may be executed in one of sixteen channels. This means that sections of music can be blended or kept separate. However, in this first prototype, a single track and channel are sufficient as no note overlapping occurs.

4.1.2 Dataset

The intention of this prototype was to compose pieces specifically in the style of Beethoven, a prolific figure in the transitional period between Classical and Romantic eras. To achieve this, solely Beethoven compositions were collated from an online repository [14], all in the required MIDI file format. Twenty-nine tracks were collected in total, varying from 2 to 13 minutes in length, which provided an ample amount of training data for this initial prototype. By using input data from a single artist, the generated output should be able to emulate their traits.

4.1.3 Encoding

An especially useful Python library was discovered, music21 [15], which is a toolkit for computer-aided musicology. It has a diverse range of capabilities but, most importantly, the toolkit offers an interface for obtaining musical notation from MIDI files. This avoids having to directly parse the MIDI bytes, as depicted in Figure 4, to determine which sequences correspond to which MIDI message. Instead, this is abstracted away and performed under the hood.

```
00000a00: 6230 8927 4000 8924 4030 9931 6200 9924  b0.'@..$@0.1b..$
00000a10: 6248 8924 4000 8931 4082 38b9 6500 00b9  bH.$@..1@.8.e...
00000a20: 6400 00b9 060c 00b9 0a40 00b9 0764 00e9  d........@...d..
00000a30: 0040 01b9 6500 00b9 6400 00b9 060c 00b9  .@..e...d.......
00000a40: 0a40 00b9 0764 00e9 0040 00b9 6500 00b9  .@...d...@..e...
00000a50: 6400 00b9 060c 00b9 0a40 00b9 0764 00e9  d........@...d..
00000a60: 0040 00ff 2f00 4d54 726b 0000 00b9 00ff  .@../.MTrk......
00000a70: 0314 5669 7461 6c69 6320 2d20 4c61 2052  ..Vitalic - La R
00000a80: 6f63 6b20 3031 00b0 0a40 00b0 0764 00e0  ock 01...@...d..
00000a90: 0040 00b0 6500 00b0 6400 00b0 060c 00b0  .@..e...d.......
00000aa0: 0a40 00b0 0764 00e0 0040 00b0 6500 00b0  .@...d...@..e...
00000ab0: 6400 00b0 060c 00b0 0a40 00b0 0764 00e0  d........@...d..
00000ac0: 0040 00b0 6500 00b0 6400 00b0 060c 00b0  .@..e...d.......
00000ad0: 0a40 00b0 0764 00e0 0040 b300 b065 0000  .@...d...@...e..
00000ae0: b064 0000 b006 0c00 b00a 4000 b007 6400  .d........@...d.
00000af0: e000 4001 b065 0000 b064 0000 b006 0c00  ..@..e...d......
00000b00: b00a 4000 b007 6400 e000 4000 b065 0000  ..@...d...@..e..
00000b10: b064 0000 b006 0c00 b00a 4000 b007 6400  .d........@...d.
```

Figure 4: Hex dump of an input MIDI file

The toolkit translates a file into a format which is easier to manipulate, through the use of Chord and Note objects. It also makes the process of constructing new MIDI files far smoother. A Chord object is purely a container for a set of notes which are played concurrently. Note objects store

information about a single note, such as its pitch, octave and offset. The pitch is represented by a letter X, where $X \in \{A, B, C, D, E, F, G\}$, and corresponds to a particular frequency. The octave refers to the set of pitches used, where a note one octave above another has double its frequency. The offset defines when a note is played, with the interval between notes being fixed at the model value, 0.5. This assumption leads to a less convoluted encoding.

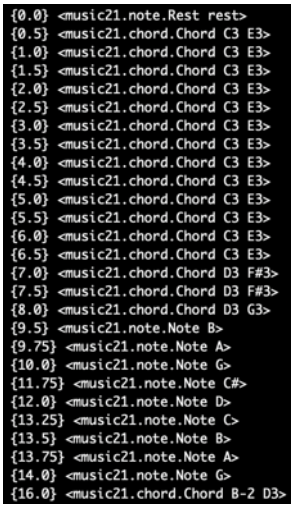

Figure 5: Excerpt from a MIDI file read in using music21

After parsing the MIDI file, the file is partitioned by instrument and only the first piano portion continued with. The elements in this part are then iterated through, with each note's pitch being appended to an overarching list of notes. If the element is a chord, the contained notes in normal form are joined and delimited with a full stop character, before also being added to the list. For example:

```
['A3', '7.11', '7.11', 'B3', '7.11', ..., 'G3', '7.11', '7.11', 'C#6', 'D6']
```

As a result of this encoding, the generated network output can be easily recompiled back into the respective Note and Chord objects. The sequential list of notes is not reset for each file, resulting in one long array of elements which can then be prepared into sequences, ready to be fed in as input to the recurrent neural network.

4.1.4 Model Preparation & Architecture

Before the sequences can be prepared, the categorical data must be transformed into numerical data for improved performance when training the neural network. This is accomplished by mapping unique strings to an incrementing index, as seen below, to provide a vocabulary consisting of 279 distinct values.

```
['A3', '7.11', '7.11', 'B3', '7.11'] => [0, 1, 1, 2, 1]
```

The sequence length is fixed at 100 elements, meaning that a single output note is predicted using this number of prior items. Sequences are created by simply splicing the notes array and then appending the integer-based representation to the network input. This input is then normalised, making it suitable to be loaded into the network. The output is one-hot encoded, which involves removing the existing value and introducing a new binary variable for each unique integer. This enhances performance and reduces unexpected results, such as predictions *between* categories.

Four different classes of layers were used in network architecture. Three LSTM layers were deployed, to take advantage of their memory capabilities which can assist in learning the long-term dependencies of the input compositions. Two Dropout layers were used to prevent overfitting; this

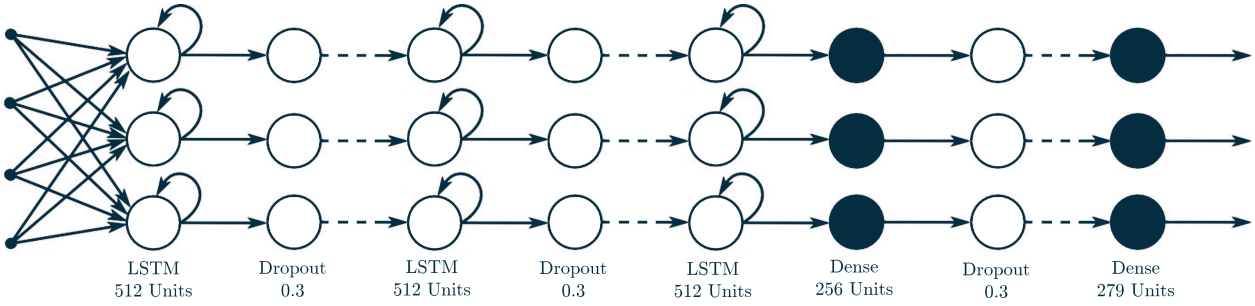

Figure 6: Architecture of the first prototype

is achieved by resetting a portion of the input nodes to zero. Two Dense layers were also inserted, where each input unit is connected to each output unit (i.e., fully connected). The final Dense layer has the same quantity of units as the number of different outputs the system has, to ensure a direct mapping. Finally, an Activation layer is added at the end, which states the activation function to be applied – the softmax function is adopted in this case. This network is called a language model; a similar approach is used to predict the next word in a sentence based on words that have preceded it. The creation of the network is effortless with Keras, with new layers added without difficulty:

```
model.add(Dropout(0.3))
model.add(LSTM(512))
model.add(Dense(256))
```

The model is then compiled using the categorical cross entropy cost function, as there are more than two classes, and the RMSprop optimisation function, which is the de facto choice for recurrent neural networks.

4.1.5 Training

As mentioned in section 2.2, the Google Cloud compute engine is used to train the models. This greatly saves personal resources and permits continuous training, without disturbances. Most importantly, the platform provides access to a powerful graphics processing unit (GPU), which can be taken advantage of to enormously accelerate training. The exact hardware specifications of the system are: 4x Intel Sandy Bridge processors, 15GB memory, 30GB SSD and crucially, an NVIDIA Tesla K80 GPU.

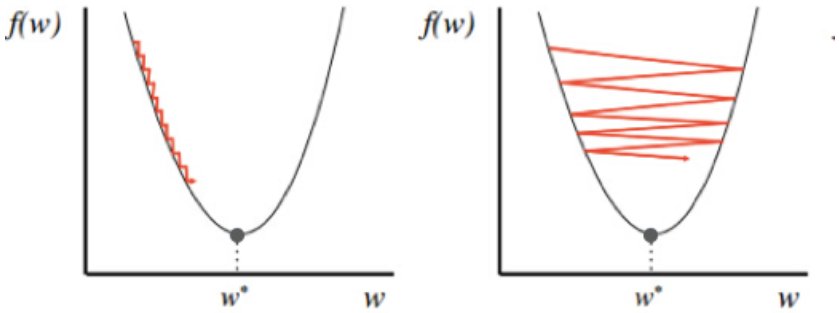

Figure 7: The effects of different learning rates. Converges slowly if too small (left); diverges if too big (right)

The entire dataset was exploited for training, as validation and test sets were not necessary this early in the development process. The learning rate was left untouched and remained at the default

value of 0.001. However, the training loss was not consistently decreasing. This divergent behaviour indicated that the model was failing to properly train, visualised in Figure 7. To solve this, the learning rate was decreased to 0.0001, allowing the model to converge.

An epoch is one forward and one backward pass of all training examples, with the batch size defining the number of examples propagated in each pass. In this prototype, a batch size of 64 was set and the network was trained for a total of 200 epochs. Model checkpoints were also used, saving the weights after every epoch. This is useful as it allows for progressive testing of the weights and provides a level of security, protecting the results in the event of failure (e.g., the Google Cloud instance randomly terminating). The network took approximately 10 hours to train.

4.1.6 Generation

In order to generate music samples, the model has to be fixed into the same state as before and the recently saved weights loaded into it. Five hundred notes are predicted, which corresponds to just over two minutes of generated music, providing the model with enough time to form a melody of some description. A random index is selected as the starting point for the network input, allowing different results to be obtained on each run. The model then invokes its predict method, returning a NumPy array of predictions, with the value having the maximum weight (i.e., most likely) being selected. This value is then transformed back into its categorical form, using the reverse of the mapping previously used, and appended to the prediction output. For the next note, the input pattern consists of the previous sequence without the first element, but with the value just outputted.

Now that the prediction output has been attained, it needs to be decoded in order to construct a new MIDI file; this is essentially the inverse of the encoding process. First, Note objects are recreated using its pitch string representation. If the element being parsed is a chord, denoted by the presence of the dot delimiter, it is split and its constituent notes iterated through, creating a Note object for each one and appending it to an encasing Chord object. As previously mentioned, the offset between each element is set to 0.5. Once the list of output elements has been collected, the music21 library is used to convert it into a MIDI stream and then writes it to a file.

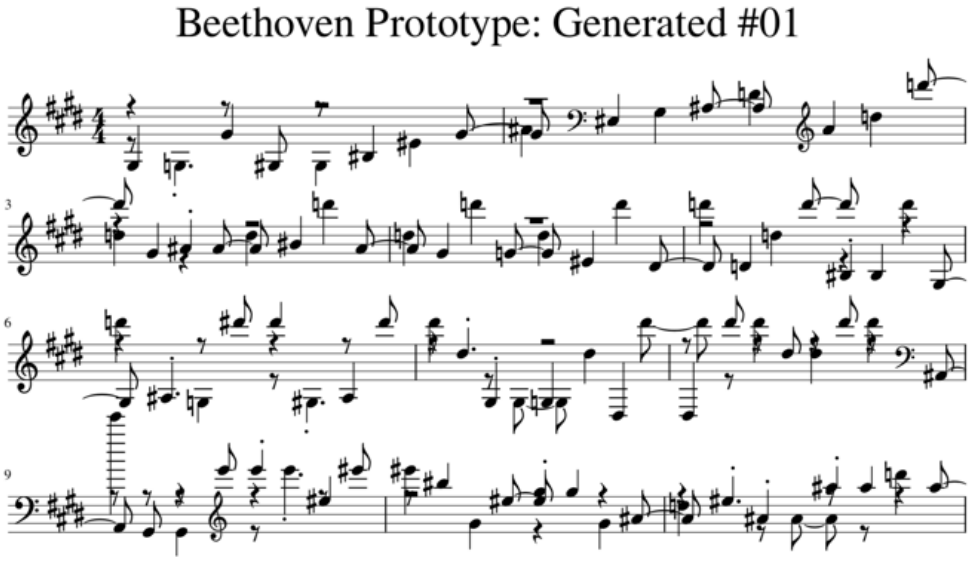

Figure 8: A snippet of the generated composition's sheet music

The results achieved are impressive considering this was only the first prototype, and a relatively basic one at that. The generated music has a sense of melody, as can be seen in Figure 8, with some parts' style being almost indistinguishable from the input sequences, to an untrained ear. There are some anomalous notes, but this is to be expected for such a primitive implementation

using a rather constrained dataset. There is also fairly prominent overfitting taking place; for example, Beethoven's 5^{th} symphony can quite clearly be recognised in one of the tracks. Again, this problem is unsurprising given no hyperparameter optimisation was undertaken – something which was deemed an unnecessary use of time and computational resources at this phase. To combat this overfitting, a larger dataset could have been used, to lower the weighting of each individual piece, and the dropout rates increased. For reference, some sample generated tracks for this prototype can be found online [16].

In summary, this initial prototype was deemed a success as it managed to achieve its goal of generating music. As this is a precursor to the final product, the next step is to escalate the complexity of the system in order to generate house music. However, before doing this, an alternative representation shall be explored.

4.2 Prototype Two: Waveform Operation

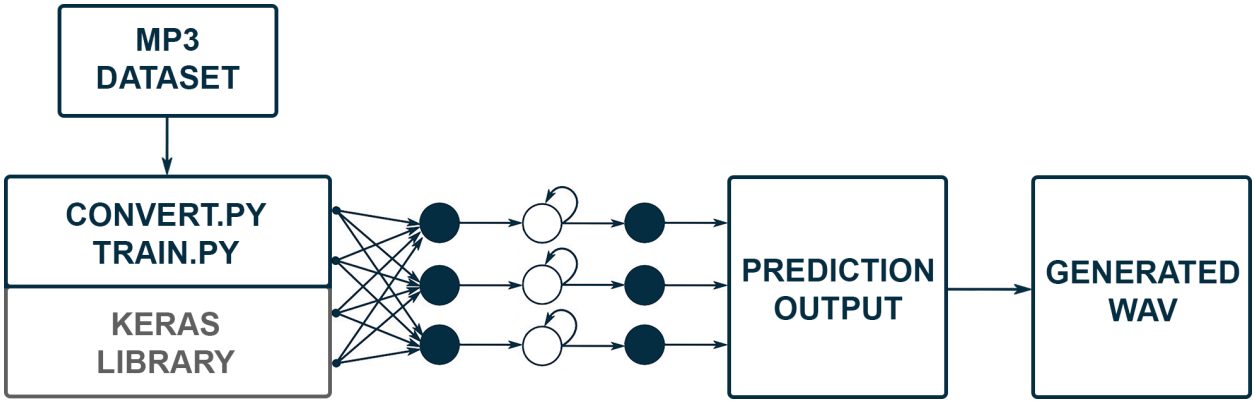

Figure 9: Design of the second prototype

Although the first prototype performed well, there does exist another methodology which has seen lucrative results in prior studies [4]. In order to consider different avenues and avoid becoming fixated on a singular approach, this method was investigated. The aim of this prototype was to surpass the results of the preceding one. If successful, it shall be adopted for the final product.

4.2.1 Input Representation

The main difference between the two techniques lies in the input representation and the subsequent encoding. As opposed to the MIDI file format, this prototype operates directly on the raw audio waveform representation of the musical data. The benefit of this is that any manner of audio data can be used as input to the neural network model, once converted into the appropriate form. As a result, existing personal music libraries can be used without any issues, so there is no struggle to collect a sufficient amount of suitable data.

4.2.2 Dataset

The dataset consists of just five copyright-free house songs, in mp3 format. This limited size was intentional in order to allow for rapid training of the model, to gauge whether the method was viable. Overfitting was not a concern at this stage and was considered inevitable to occur with such a small dataset. Nonetheless, the dataset was enlarged at a later stage.

4.2.3 Encoding

The selection of input mp3 files are initially converted into Waveform Audio File Format (WAV) using the Pydub library [17], specifically into *monaural* sound. Monaural, or mono-channel, sound

involves a singular channel which is then broadcasted to all speakers, meaning that the sound heard in both ears is identical. This reduces the dimensionality of the audio data, causing the ensuing NumPy arrays to be significantly smaller. This reduces memory requirements as well as the time taken to train the network, while having no effect on the generational capabilities.

The WAV files are next read into NumPy arrays, using the inbuilt *scipy.io.wavfile.read()* function. The 16-bit integers are then normalised to a range of $[-1, 1]$, to assist with the speed of gradient descent. The audio signal is now ready to be converted from a continuous state into a discrete one, so that constant sizes can be used for training and generation. To do this, the signal is sampled at regular intervals, at a frequency of 44,100 Hz. This sampling frequency is used because the human ear is only sensitive to values up to approximately 20,000 Hz, so anything above this will make no audible difference, and it must be at least double this value, as per Nyquist's Sampling Theorem [18].

After sampling, the NumPy arrays are split into smaller blocks of dimension N. If necessary, the last block is zero padded to reach this size limit, which has no effect the produced content. The audio data is currently within the time domain, but it is more useful in its frequency representation. To achieve this conversion, which is visualised in Figure 10, the discrete-Fourier algorithm (DFT) is applied to the blocks. Specifically, NumPy's Fast Fourier Transform implementation is used.

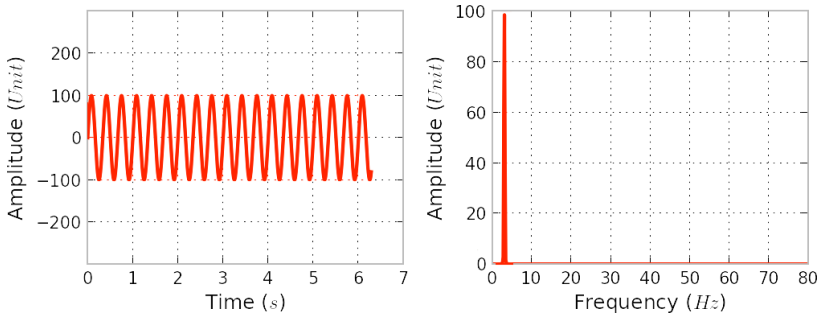

Figure 10: Time domain (left) to frequency domain (right) conversion using DFT

The block size, N, is ideally a power of two, for example 2048, to improve the efficiency of this algorithm. The transformation yields vectors of N real and imaginary numbers, which are concatenated to create a $2N$ vector, completing the input encoding.

4.2.4 Model Preparation & Architecture

The sequences are prepared by dividing up the frequency data into fixed sized chunks, with the length being determined by:

$$s * c/b$$

Where s is the sampling frequency, c the clip length and b the block size. These chunks are then centred, before their tensors are saved to file.

The language model previously used is not suitable for this model representation, but a similar architecture is adopted. A linear shallow network is used, which typically have just one hidden layer, with 1024 LSTM units in this scenario. The input layer takes in 44,100 sequences and the output layer produces an identical number. To convert from frequency space to hidden space and vice versa, Time Distributed Dense layers are used either side of the LSTM layer. These act like the Dense layers in the first prototype but apply the (fully-connected) operation to *every* timestep. They maintain the many-to-many architecture, while keeping one-to-one relations between input and output. The RMSprop optimiser is utilised again, with the default learning rate this time, and a mean squared error loss function is used instead of a categorical cross-entropy one.

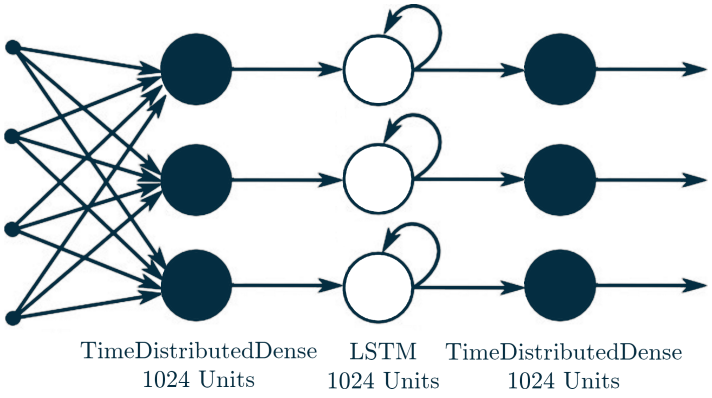

Figure 11: Architecture of the second prototype

4.2.5 Training

The model was trained in a similar fashion, for 200 epochs with batch sizes of 64. Again, no validation or test sets were used as these are reserved for the evaluation of the final product in order to save resources. Training each epoch took approximately 30 seconds. This was not a problem due to the small dataset, but it would have become one once the dataset was extended. However, GPU support was added, instigating a speed-up by a factor of 10, resulting in a single epoch taking just 3 seconds. Leveraging CUDA-based GPU acceleration proved rather challenging. Initially, the environment was setup manually, but this failed so a docker approach was then taken. The docker image approach was also unsuccessful, leading to the deployment of an Intel Deep Learning virtual machine image to rectify the situation.

4.2.6 Generation

Given a set of vectors, $X_0, X_1, X_2, \ldots, X_t$, representing the waveforms at successive time intervals, X_{t+1} is generated for the next interval. This is iteratively performed until the sequence is of sufficient length – equivalent to 10 seconds in this scenario.

Once a sequence has been generated, the data centring process is undone to obtain valid frequencies. The Fast Fourier Transform operation is inverted to convert the data back into the time domain, before it is then written to a WAV file.

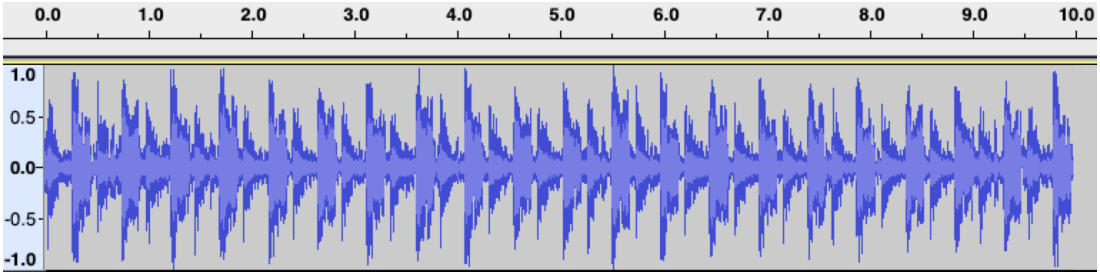

Figure 12: A generated song loaded into Audacity

The results were rather varied. After the first run, the results were incredibly noisy with only a faint hint of melody. It was clear that the model was insufficiently trained, so the number of epochs was increased from 200 to 1000. The results of this subsequent experiment were substantially better, with noise still present but the musical part being much more prominent. Total epochs were further increased to 2000 and to 5000 in order to try improving results again, to no avail; it seems there is a limit as to how much noise could be substituted. The learning rate was also played around with in an attempt to produce better results.

Experiment	Dataset Size	Epochs	Learning Rate	Training Loss
1	Small	1000	0.001	0.3655
2	Small	2000	0.001	0.3549
3	Small	2000	0.0001	0.2576
4	Large	1000	0.001	0.6491
5	Large	2000	0.001	0.6400
6	Large	2000	0.0001	0.5161
7	Large	5000	0.00001	0.5716

Table 1: Experiments undertaken for prototype two

Noise filtering techniques were employed, in particular Audacity's noise reduction utility, but no significant enhancement could be achieved using either high- or low-pass filters. The presence of this noise is a serious complication for the evaluation stage, as it would lead to the straightforward detection of generated pieces in a Turing-test scenario.

Another problem with the results was hefty overfitting. This was initially thought to be connected to the restricted dataset size. However, after the dataset was extended to include more songs, overfitting was still a prominent issue. Finally, any novel section of the track (i.e., which has not overfit) does not sound melodious at all, seemingly consisting of unrelated frequencies. This, of course, violates the research hypotheses. The cause of this may be due to the lack of patterns which the LSTM RNN can detect. Data in the time domain would exhibit more temporal patterns, however this representation cannot be used effectively. This is because there may well exist two similar signals, yet if they are out of phase at some instances then the loss function will assume there is no relation.

Perhaps through the tweaking of the representation and further optimisation, better results could be achieved. However, due to the unsuitability for the evaluation criteria and the time constraints of the project, it was deemed that shifting the input representational focus back to what it was in the original prototype had a lower opportunity cost.

4.3 Final Product: Multi-Instrument MIDI

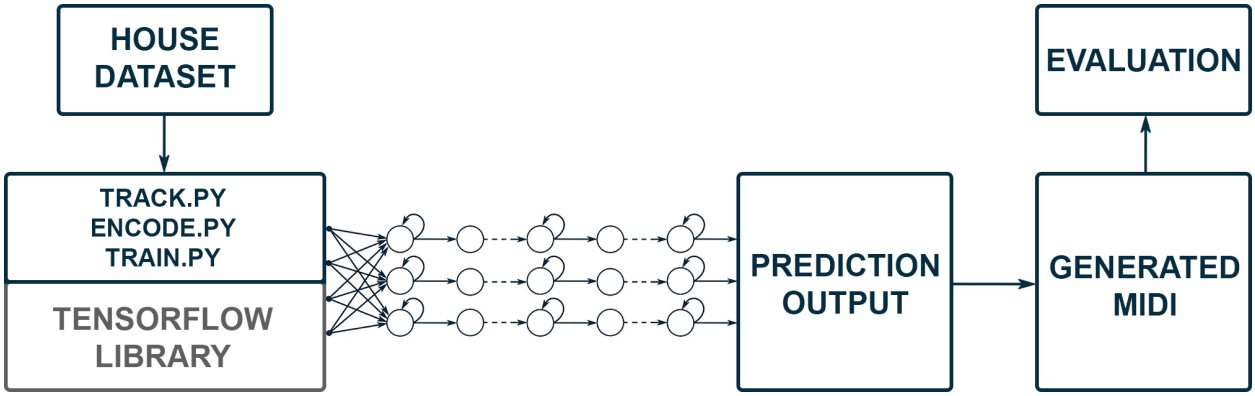

Figure 13: Design of the final product

The first prototype behaved well for composing music in the style of Beethoven, but unfortunately it lacks the representational capabilities for generating house music. In particular, there is no support for rests, varying note durations, fluctuating note offsets and, perhaps most importantly, multiple instruments. All of these factors are crucial in order for the end product to satisfy the aforementioned research hypotheses. Therefore, the objective of this final project iteration is to

implement support for each of these features, allowing house music to be fully generated once this has been accomplished.

4.3.1 Input Representation

The MIDI file format shall be enlisted again, as it was satisfactory for the Beethoven prototype. Refer back to section 4.1.1 for more information regarding this standard.

4.3.2 Dataset

Naturally, an identical dataset cannot be used given that it is of the improper classification. A new dataset was manually gathered, consisting of 111 reconstructed house tracks from two online repositories [19, 20]. House tracks are not natively in MIDI format, with the mp3 format being most universal. Unfortunately, mp3 files cannot be consistently converted into MIDI format without major degradation and loss of salient features. This meant that it was difficult to locate viable input tracks, and as a result the dataset is limited in size, consisting only of tracks which are human transcribed into the applicable format. The absence of a large corpus may prove to be an issue in the future, as copious amounts of data is required to effectively train deep neural networks. This problem is worsened by the splitting of the dataset into validation and test sets. However, this issue was minimised by seizing only 10% of the dataset for each additional set, instead of the proposed 20% as stated in the initial project specification.

4.3.3 Encoding

The previous encoding scheme is no longer suitable for this more elaborate system. The addition of rests, timing and multi-instrument support all require supplementary extensions to the vocabulary.

Initially, music21 was utilised again to parse each file. However, after encoding the files and then instantly decoding them, exact replicas could not be achieved. It seemed that the toolkit was too high-level, abstracting away a lot of necessary musical information which meant that the tracks could not be accurately reproduced. Specifically, the library lacked sufficient support for some of the instruments found in house tracks. In order to solve this issue, a new Python module was identified, Mido [21]. Similar to music21, this is a library for working with MIDI messages and ports, except it does not dissociate any of the musical information. This allows the MIDI tracks to be perfectly recreated.

A MIDI file can be imported using Mido, providing access to a list of its contained tracks. Up until now, a *track* was synonymous with *song* in this report. However, in this section, a track is a Mido object containing a list of messages and/or meta messages. Messages correspond directly to MIDI messages, such as *note on* and *control change* events, whereas meta messages cannot be sent on a port but otherwise behave like normal messages, detailing information pertinent to a piece.

There exist five common message types within each house track: *note on*, *note off*, *program change*, *control change* and *pitchwheel*. Control change and pitchwheel are responsible for a channels control value and pitch, respectively. These two types do not seem have a compelling effect on the final outcome, so they are omitted in the encoding scheme. This is important as it reduces the number of distinct strings, resulting in a leaner vocabulary which will perform better during training. The remaining three messages play significant roles, with note on and off being self-explanatory and program change being used to specify the instrument for each channel. This direct mapping from MIDI messages to Mido is incredibly helpful. The presence of note off is especially valuable as it means notes can be easily played for varying durations – one of the objectives for this product iteration. This is realised by distinguishing between note on and off events in the encoding, through the addition of an articulation suffix (an asterisk) to each note on event. A similar approach was taken for the program change message. To differentiate it from the other two supported messages, an octothorpe (or hash sign) suffix is appended.

There are also prefix symbols before each message value, which correlate to the track index. The maximum number of tracks in the dataset used is fifteen, with the first two being ignored

```
<meta message track_name name='Lead' time=0>
control_change channel=14 control=10 value=64 time=0
control_change channel=14 control=7 value=100 time=0
pitchwheel channel=14 pitch=0 time=0
control_change channel=14 control=101 value=0 time=0
control_change channel=14 control=100 value=0 time=0
control_change channel=14 control=6 value=12 time=0
control_change channel=14 control=10 value=64 time=0
control_change channel=14 control=7 value=100 time=0
pitchwheel channel=14 pitch=0 time=0
program_change channel=14 program=0 time=0
control_change channel=14 control=101 value=0 time=0
control_change channel=14 control=100 value=0 time=0
control_change channel=14 control=6 value=12 time=0
control_change channel=14 control=10 value=64 time=0
control_change channel=14 control=7 value=100 time=0
pitchwheel channel=14 pitch=0 time=0
program_change channel=14 program=0 time=0
control_change channel=14 control=101 value=0 time=0
control_change channel=14 control=100 value=0 time=0
control_change channel=14 control=6 value=12 time=0
control_change channel=14 control=10 value=64 time=0
control_change channel=14 control=7 value=100 time=0
pitchwheel channel=14 pitch=0 time=0
program_change channel=14 program=0 time=0
note_on channel=14 note=60 velocity=100 time=48
note_on channel=14 note=57 velocity=100 time=0
note_on channel=14 note=69 velocity=100 time=0
note_off channel=14 note=69 velocity=64 time=48
note_off channel=14 note=60 velocity=64 time=0
note_off channel=14 note=57 velocity=64 time=0
```

Figure 14: Example Mido messages

at this stage as they only contain meta-information. This leaves thirteen remaining tracks, with the upper eight being disregarded as they are infrequently used, as per Table 2. Without these exclusions, the ratio of unique values to the total number of values would be very low, resulting in the encoding scheme being too complicated. As found through testing, this causes excessive variety, making it difficult for patterns to be discovered and the network to generalise well. This is exceptionally essential to avoid because the total number of symbols cannot easily be increased, as it is a genuine challenge to locate more songs to extend the dataset with.

Track	**Number of Messages**	**Number of Rests**	**Rest Proportion (%)**
2	31125	68164	68.65
3	24285	70231	74.31
4	23601	70852	75.01
5	16739	75194	81.79
6	13479	77952	85.28
7	3475	83488	96.00
8	1745	84428	97.98
9	2294	84043	97.34
10	1914	84314	97.78
11	662	85101	99.23
12	2260	84269	97.39
13	3238	84803	96.32
14	6	85420	99.99

Table 2: The rest proportions over the entire dataset; >90% and the corresponding track is discarded

As a result, five tracks are encoded, with each being prefixed by a character designated by its alphabet index; track one is preceded by 'a', track two by 'b', and so on up until 'e'. These prefixes allow the parallelism and separateness between tracks to be maintained throughout the entire process. They are also used when defining the channel during the decoding, ensuring it is always independent. This is an assumption, as the channels do occasionally overlap to integrate certain parts, but it would add to the overall intricacy of the encoding for minimal return.

Once a MIDI file has been imported, the file's tracks are iterated through. In each track, there

exists a list of messages of undefined length. If the message is one of the three desired types, its value, plus any suffix, is added to a separate list at the correct time slot. This list has a predefined size of 5000, which was calculated beforehand as the maximum size needed, and is then trimmed down if any elements are not required. The step between each element in the list represents a time gap of 12 ticks. Previously, each step represented 1 tick, but this extended the list sizes substantially inducing far more arduous data processing. A trend was since discovered, where each message's time attribute was a multiple of 12, permitting the reduction in size. To clarify, a message's time signifies the time elapsed since the last yielded message. This time is measured in ticks, the smallest unit of time in MIDI, with multiple ticks equalling a beat. The exact number is stored as the *ticks_per_beat* variable of the Midi file object, with it being 96 for every song in the dataset.

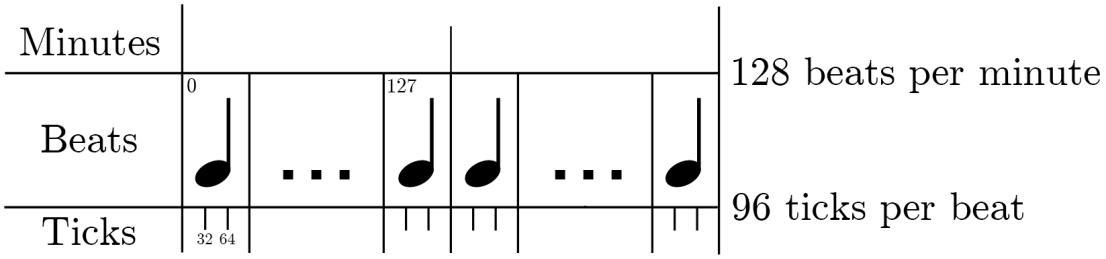

Figure 15: MIDI tempo and beat resolution

At this stage, time steps for a *single track* looks as follows:

```
['69* 59* 55*', '', '59', ..., '69', '55', '']
```

Once all messages and tracks have been processed, the encoding is then post-processed. This involves collating each track's list of elements into one encompassing string, to make it suitable for the RNN variant, along with several adjustments. First, an empty element represents no messages being played at that time step, so is replaced by an R character, indicating a rest. Secondly, messages at the same time step are collected together. A sequence is created, initiated with the track's prefix, with each event separated by a dot delimiter. This is undertaken to help the network learn dependencies between time steps at specific tracks. Finally, the events are also sorted ascendingly. This provokes enhanced prediction during generation because it reduces the number of possible outcomes. For instance, $a43.22$ will never be seen and will always be encoded as $a22.43$. After post-processing, the augmented encoding scheme at a *single time step* looks as follows:

```
a55*.59*.69* b62* c41*.43 dR eR
```

An alternative to this scheme was considered whereby each track is handled separately. In this approach, there would be an independent string for each track, for a total of five strings. Each string would then be individually fed into the neural network. This would make the encoding processing simpler, as no combining of terms or time steps would be necessary. It would also affect the power of the sequence length. With a sequence length of 50 and one encompassing string, only ten previous messages are considered within each track. With five isolated strings, all fifty messages are relevant. However, while this may help to improve the quality of music for each respective track, it would cause a divergence where the relationships between tracks are diminished. This would result in a mash-up of different sounds with no notion of unity and thus harmony. For this reason, the substitute avenue is rejected.

4.3.4 Model Preparation & Architecture

At first, the model architecture was consistent with the Beethoven prototype and remained unchanged, given that it previously performed so well. This time, there were 31997 symbols, with

1227 of them being unique. This is considerably more than the previous prototype, which had a distinct vocabulary size of 279. The sequences were prepared verbatim, with an initial sequence length of 100 as before. Unfortunately, the results of this model were poor. In the generated songs, there was almost no rhythm or melodic sequences, rarely any program changes and frequent completely blank sections. The abundance of rests indicates that the network became fixated on the *R* character, assigning it a high weighting. While this should be the most prevalent character in any track, the problem is that it is only ever intermittently interleaved with other elements. The encoding was modified several times in an attempt to reduce this dependence, to no success. Even after some model tuning was undertaken, through varying the learning rate and number of epochs and switching the optimisation function from RMSprop to Adam, there was no noticeable improvement in the generated music. In the end, it was deemed that there was an integral flaw in the network architecture. While this structure managed to cope with the relatively simple encoding scheme of the first prototype, it could not handle the enhanced version with the broader vocabulary to anywhere near of the same effect. Consequently, massive revisions to the model were undertaken in respect of this problem.

It was decided that the language model needed to be refined, to the character level. The benefit of this is greater flexibility and a substantially smaller vocabulary, consisting of just twenty characters (0-9, *a*-*e*, *R*, *, #, ., ' '), which will improve the ability of the network to learn dependencies. The downside of this structural shift is that different networks are now required to handle this change, which are slower to train, and invalid combinations can now be produced, as there is no longer a direct mapping to valid elements. For instance, out-of-range MIDI note values or duplicate suffixes may occur.

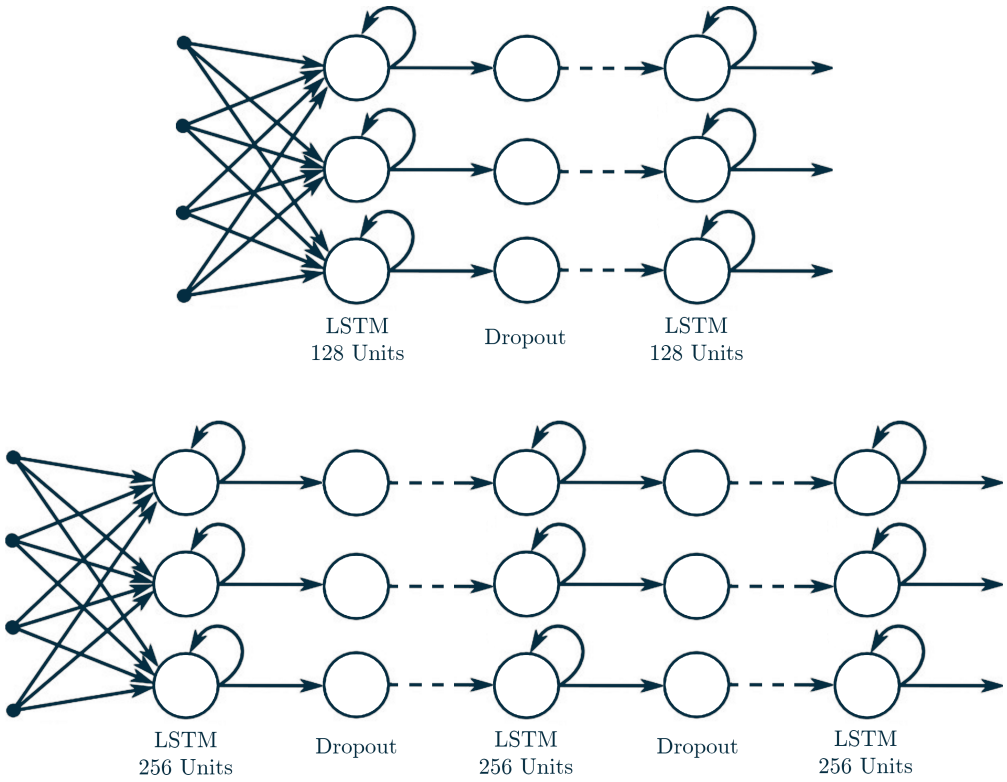

Figure 16: Shallow (top) and deep (bottom) architectures of the final product

To adopt this new model, the Keras implementation was abandoned in favour of an established raw TensorFlow implementation of character-RNNs [22]. By using a model which has successfully generated characters in studies before [8], any encountered issues can be associated with the input representation, encoding scheme and tunable model parameters. Therefore, the benefit of this constrained approach is that it allows full concentration on and responsibility attribution to these

factors.

Several variants of this model were designed and analysed. With the shallowest network being a two-layer LSTM RNN, having a hidden state size of 128. The deepest model deployed was a 256 hidden size three-layer network. As standard for character-level RNNs, the Adam optimiser is used, with a learning rate of 0.002 set to help accelerate training.

4.3.5 Training

Initial hyperparameters include a batch size of 64, sequence length of 50 and no dropout. The model was trained for 200 epochs, as before, and both training and validation loss were measured in this project iteration so that it could be properly assessed. This validation loss was measured by testing the validation set every 1000 iterations.

The training duration for each experiment was approximately 60-120 minutes. This is a reasonable length, with the relative shortness being attributed to the limited dataset size. To visualise the learning process, TensorBoard hooks are placed which help make it easier to understand, debug and optimise the model.

4.3.6 Generation

The model is sampled using the input vocabulary. The number of characters generated is exactly 13,178, as this was the calculated mean length of a single song. This number is substantially higher than in the Beethoven prototype, which produced 500 elements, due to differences in tempo and beat resolution. There also exists another parameter during sampling, *temperature*, which affects the prediction output. This variable takes a value in the range $[0, 1]$ and is used to divide the predicted log probabilities before the softmax activation function. Therefore, a lower temperature will result in more conservative predictions, while a higher temperature will take more risks, diversifying results at the cost of more mistakes. Different temperature values were tried, with lower values producing far too many rests (the R character is the safest option). In the end, a value of 0.9 was chosen. This promotes a healthy balance between producing novel output, satisfying the second research hypothesis, and ensuring invalid sequences do not occur too often.

The sampled characters are saved to an output file, which is then read in and processed by the decoding algorithm. This algorithm is naturally different to the one in the first prototype, given that an altered encoding format has been used. A MidiFile object is instantiated, specifying that all tracks are synchronous (i.e., start at the same time) by passing in a type parameter. The two meta-tracks are then created and appended to the MidiFile object. These tracks are constant in every generated piece, representing the time signature and tempo, where the tempo is fixed at a value of 468,750 – the modal value seen across the input songs. Five other empty tracks are then added to the object, which will contain lists of messages once the prediction output has been completely decoded.

The prediction output is iterated through, checking for the presence and type of suffixes to determine which message each element conforms to. The appropriate message is then added to the track denoted by the character prefix. *Note on* messages have a fixed velocity of 100, while *note off* messages have a value of 64. While this is another assumption, it does not have any real audible bearing on the output. As touched upon earlier, invalid sequences are encountered from time to time. These are currently handled by simply ignoring them. In the future, this could be changed so that the most similar elements are found and then replace them.

After each track has been iterated through at a given time step, the ticks for each track, indicating when a message was last received, are incremented by twelve. The ticks for each track are correspondingly reset at each stage if the element is not a rest. Finally, after all time steps have been processed, the resulting MidiFile object is saved.

4.4 Testing

The test-driven development methodology was adhered to throughout the project's timeline. PyUnit was used for unit testing, with each relevant method being asserted that it functions as intended. In particular, it was checked that the output was decoded in the correct manner and that any invalid characters were handled properly.

Name	Stmts	Miss	Cover
encode.py	72	21	71%
test_encode.py	21	0	100%
track.py	15	0	100%
decode.py	51	15	71%
test_decode.py	61	0	100%
evaluation.py	35	18	49%
test_evaluation.py	18	0	100%

Figure 17: PyUnit code coverage

To assess how well the neural network predicted, a plethora of experiments were run. These shall be comprehensively detailed in the following section.

5 Experiments & Results

The algorithmic generation system has now been constructed. The next step is to carry out experiments and analyse the subsequent results in order to gauge to what extent the research hypotheses are met. The model's parameters need to be tuned so that its full effectiveness can be reached, and the limits of the model acknowledged.

Tuning a model involves a lot of trial and error, with every situation requiring a different approach. Several areas have been identified which require experimental testing. The architecture is of paramount importance, which the number of layers and size of the hidden state determine. Other points of interest include, from a training view, the learning rate and number of epochs, and from a generational view, the sequence length and dropout rate.

The model's ability to learn is quantitatively assessed through the training and validation loss, particularly effectively with the aid of the TensorBoard visualisations. However, the generational side will be judged in a qualitative manner. It should be noted that this will not be the case in the evaluation of the final product, where insight shall be as objective as possible coupled with extensive quantitative metrics.

Experiment	Hidden Size	Layers	Sequence Length	Dropout Keep Rate
1	128	2	50	1.0
2	**256**	2	50	1.0
3	256	**3**	50	1.0
4	256	3	**100**	1.0
5	256	3	100	**0.8**
6	256	3	100	**0.6**
7	256	3	100	**0.4**
8	256	3	100	**0.2**
9	256	3	**200**	0.6
10	256	3	**50**	0.6

Table 3: Final product experimentation; values are bolded if changed between runs

A plan was constructed to systematically explore all these variable options. This involved altering just a single parameter or element of the model between each experiment, so that any changes incurred can be instantly attributed to that specific factor. A total of ten experiments were undertaken, listed above.

A baseline is first established using the shallowest network, a 128 size two-layer LSTM RNN, with the initial hyperparameters stated in section 4.3.5. The first three experiments look at manipulating the architecture of the network, through amending the hidden state size and number of layers. These are the key parameters in controlling the model. The hidden state size is doubled to 256 in experiment two, before an additional layer is then added in experiment three. Deeper networks than this need not be considered due to the limited amount of input data available.

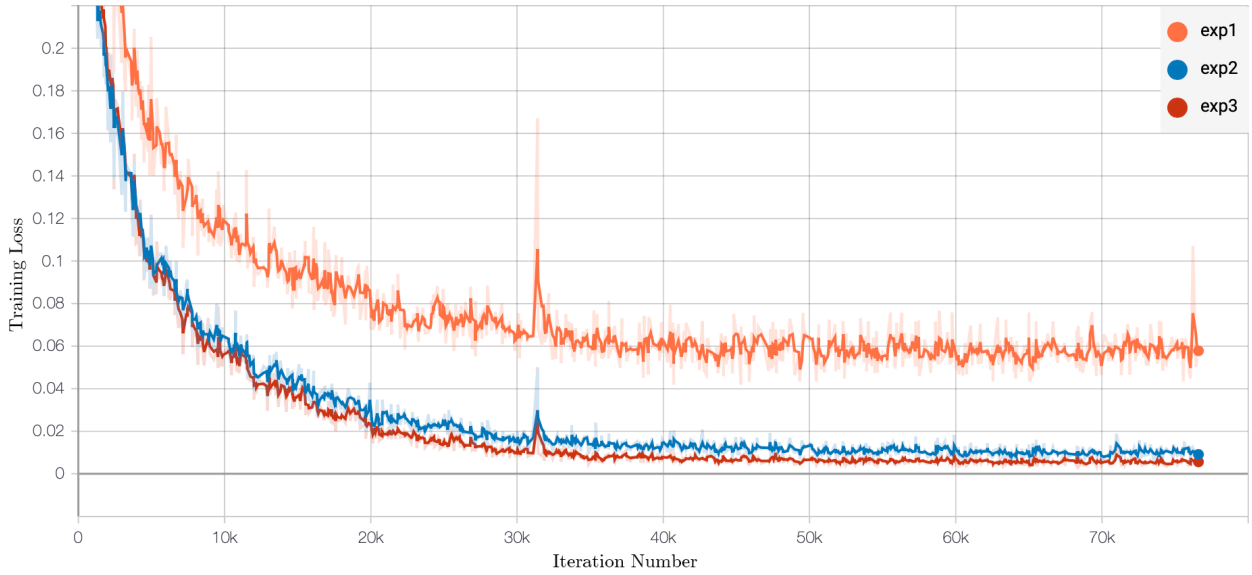

Figure 18: Training loss of experiments 1-3

As can be seen from the graph, the model is training well. The decreasing gradient signifies that the training loss is getting smaller, indicating the network is adequately learning. For this reason, the learning rate does not need to be adjusted. The number of epochs can also remain constant at 200, as the training loss rate of change starts to stagnate after approximately 40,000 iterations. This means that training for additional epochs would have a negligible impact on the training loss and therefore on the final outcome.

In terms of the individual experiments, it is clear that by extending the architecture and deepening the network, the model's performance improves. For instance, after training has complete, the shallower architecture from experiment one has a training loss over ten times greater than the deeper network used in experiment three. This performance increase is largely caused by the inflated hidden state, as opposed to the additional layer. This is clear from the disparity of gradients between experiments one and two and experiments two and three, with the former having a greater distance. The only downside of using this deeper configuration is the elongated training time, with it taking double the time for experiment three in relation to experiment one. However, this time was only 2 hours and 18 minutes – still relatively conservative and not an obstacle that needs to be addressed, even considering the time constraints of the project.

Now that the model architecture has been determined, the next hyperparameter to consider is the sequence length. Initially this was set at 50 and then boosted to 100 in experiment four. This should improve dependencies within sequences. However, the end training loss is practically equivalent, demonstrating that it does not seem to have a significant influence. Although, it could be the case that the quality of the music is highly dependent on the sequence length, something which cannot be gauged from the training loss alone. For this reason, the sequence length will be

revisited once the generated music is ready to be examined (i.e., after the other variables have been honed).

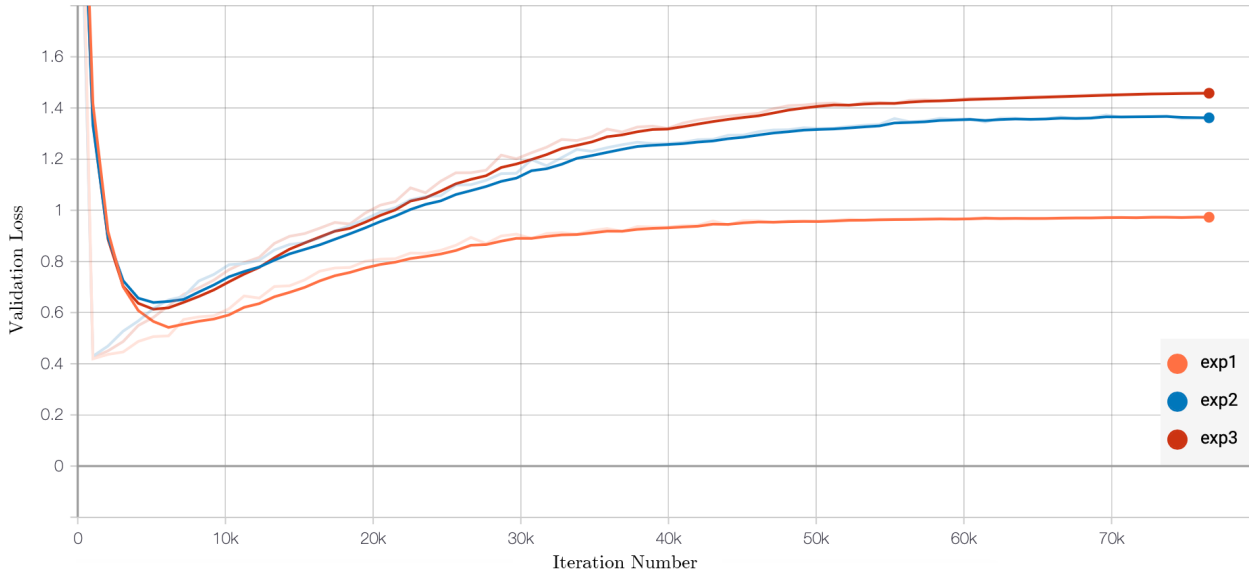

Figure 19: Validation loss of experiments 1-3

Another by-product of the deeper network, and the enhanced learning ability it induces, is the problem of overfitting. Without even listening to any music, it is apparent that this problem is worse with a deeper network, as expected given that the model is learning better. This is corroborated by the validation loss graph in Figure 19, which shows the third experiment having the highest validation loss. Not only is this value the highest, it is also considerably higher than the training loss, indicating definite overfitting. There are two ways to combat this problem. The first involves increasing the amount of training data, which will help the model to generalise better, avoiding overfitting. As mentioned, this is not possible due to the finite resources available. The second solution, and the one which shall be introduced, is the notion of dropout. This is a regularisation method which helps the network to not overfit. By reducing the proportion of nodes which are kept (i.e., not reset to zero), a different view is given to the subsequent layers which simulates a range of different inputs.

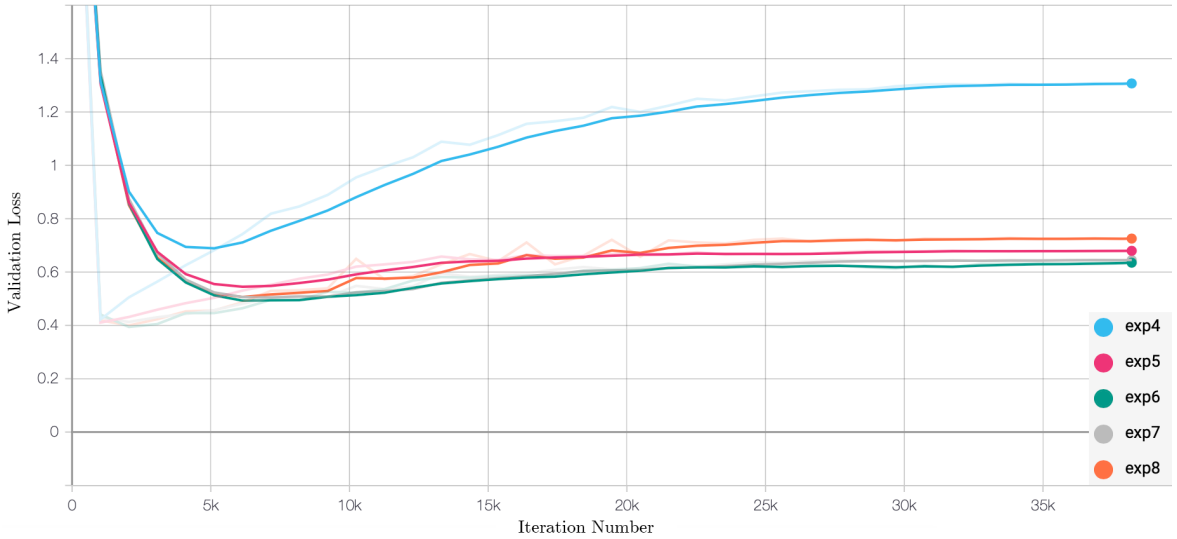

Figure 20: Validation loss of experiments 4-8

To investigate the effect of dropout, four experiments were carried out (experiments 5-8 in Table 3), each with a dropout keep rate of 0.2 less than its predecessor. These are then compared to each other, and to experiment four, which has an identical configuration but with no dropout active (keep probability of 1.0). As can be seen from the graph in Figure 20, experiment six has the lowest validation loss. Although the difference is relatively minimal, it will still have beneficial impact on the level of overfitting. For this reason, a dropout keep probability of *0.6* shall be set.

The model has been almost entirely tuned, so it is time to generate music to qualitatively decide on the remaining variable, the sequence length, and then whether it is suitable to subject it to further evaluation using the criteria stated in the project specification. In the final two experiments, the sequence length was both doubled to 200 (experiment nine) and halved to 50 (experiment ten), with all other parameters untouched. These were compared to experiment six, which has a sequence length of 100. No discernible difference was noticed between sequence lengths of 100 and 200. However, these both seemed to exhibit more musical tendencies than a sequence length of 50. In the end, it was decided that a length of 200 was to be used, as there are no drawbacks in comparison to a length of 100, resulting in experiment nine's configuration being selected for the final product.

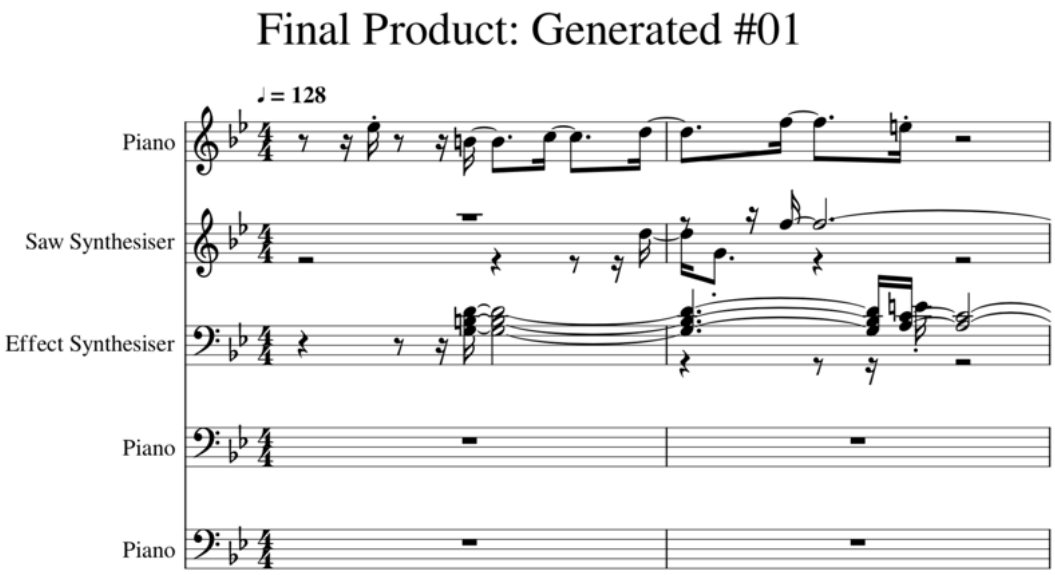

Figure 21: Sheet music fragment of a 128 beats per minute generated composition

The produced results are very promising but shall be fully evaluated, in an objective manner, in the following section.

6 Evaluation

In this section, the generated music shall be evaluated with respect to the two research hypotheses. The technical component requirements listed in the project specification shall also be appraised.

Music and other forms of art have always suffered during the evaluation stage [23]. The judging of music is subjective, resulting in a conducive and quantitative measure difficult to obtain. However, there are actions that can be taken to reduce this subjectivity and objectify the results, while also providing a numerical conclusion.

6.1 Quantity Evaluation

"How effectively can a neural network generate harmonious and musically coherent house music?"

To answer this research question, a concert-like scenario with a human audience is constructed. Having an audience, as opposed to an individual, balances the subjectivity of an individual's opinion, musical preferences and musical knowledge. These three factors all play a part in the evaluation of a musical piece; hence they are compensated for by averaging them over many people.

The concert-like scenario was simulated using an online survey, fully answered by 78 distinct people. This avoided the need for physical attendance, which allowed for a greater number of responses. The survey involved two aspects: the rating of music, between 1 (poor) and 10 (excellent), and the decision of whether it was naturally (human) or artificially (computer) composed. The latter criterion was inspired by the Turing test [24]. These two areas were combined into the same survey for ease, instead of being separate as stated in the project specification.

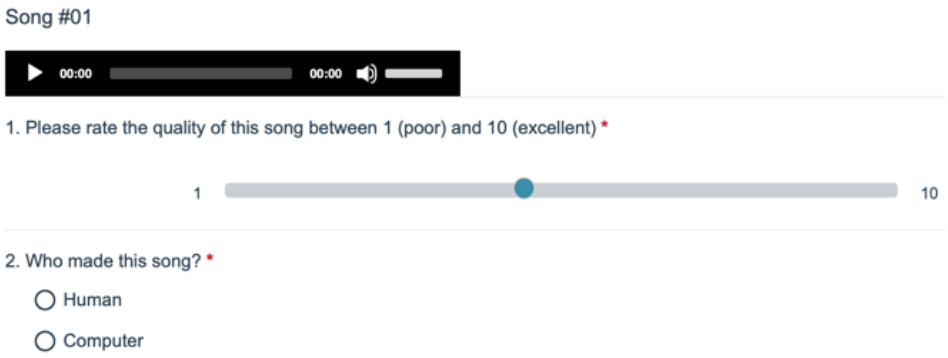

Figure 22: Sample survey question

The survey consisted of six unnamed songs. There was an equal split of those made by a human, taken from the input dataset, and those generated by a computer, presented in a stochastic order. Each song was truncated to ten seconds in duration, reducing the time taken to complete the survey. This helped to avoid subjects suffering from fatigue [25]. A minimum time limit of 75 seconds was set for the entire survey. This was an attempt to ensure people listened to each track in full before answering the corresponding questions.

To eliminate bias, the latest generated songs were selected (i.e., not cherry-picked) and randomly trimmed to the required length. Furthermore, the survey was pushed out to online forums and therefore anonymously completed. It was purposefully not given to acquaintances, as they may have been more inclined to respond in an impartial manner.

Song	**Rating (1-10)**		**Correct Identification (%)**
	Mean	**Standard Deviation**	
Human #01	3.8	2.3	71.8
Human #02	6.3	2.2	78.2
Human #03	6.8	2.0	87.2
Generated #01	4.3	2.4	65.4
Generated #02	4.1	2.3	79.5
Generated #03	3.7	2.1	91.0

Table 4: Online survey results

The survey results provided valuable insight into the performance of the model. Generated #01 was the best performing track algorithmically created by the system. This had a mean quality rating of 4.3, only 2.5 points shy of the best human track, and a mode of 6, as can be seen below in Figure 23. It also managed to fool 34.6% of participants into believing that a human had composed it. To cause more than one in three people to incorrectly identify an artificially generated track is an impressive result.

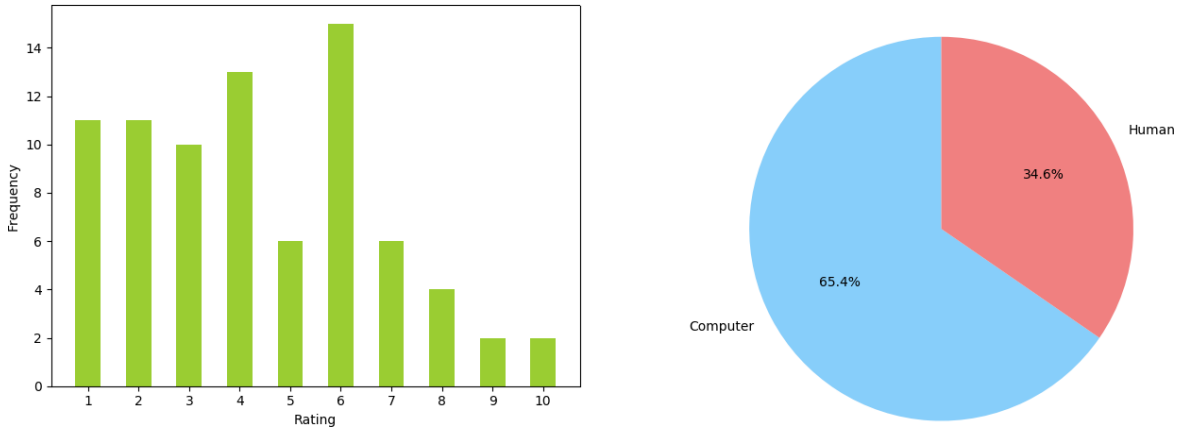

Figure 23: Matplotlib visualisations of the gathered data from Generated #01. Specifically, a bar chart (right) showing rating frequencies and a pie chart (left) depicting the identification proportions

The other two generated songs did not perform quite as well, with 79.5% and 91% of respondents managing to identify their origin, but still had respectable results. The disparity between Generated #01 and its siblings show that the prediction output can vary by a fair margin between generations. Interestingly, as shown in Figure 24, two of the three generated songs had an average enjoyment score surpassing one of the human produced songs, Human #01. Generated #03 is also very close to beating it too. This proves that the model is, to a certain extent, competitive.

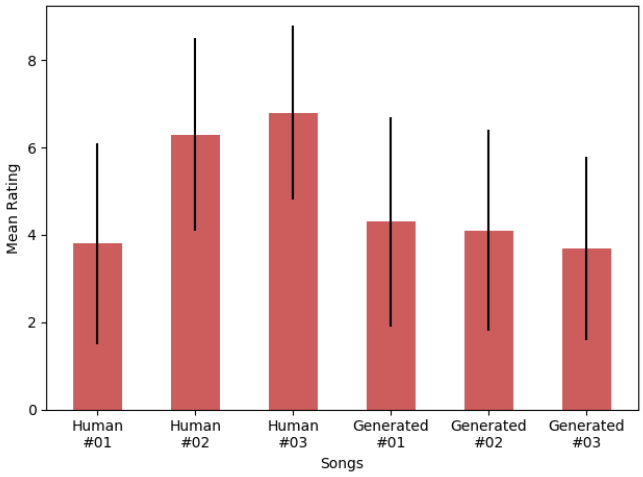

Figure 24: Bar chart showing the mean ratings of the evaluated songs, with standard deviation error bars

Although the results are not perfect, they definitely exceeded expectations and show that recurrent neural network generation has the ability to contend with human composition. As a result, the quality evaluation criteria of the generated songs can be said to have been met, due to the comparable ratings and the fact that some people believed they had been composed by a human.

6.2 Novelty Evaluation

"To what extent is the generated music novel?"

Previously, it was proposed that the novelty of the music was to be evaluated using frequency spectrograms. Given that the input representation has since shifted from the frequency domain,

this is no longer suitable. Instead, as the input encoding and prediction output are both strings, similarity metrics can be used; this is an innovative approach to novelty evaluation of music.

In particular, the *Longest Common Subsequence* (LCS) distance metric [26] is taken advantage of. This is suited to this application as if significant overfitting is present, the track will share parts of the encoding with an input song. A normalised response in the range $[0, 1]$ is presented; two identical strings are represented by 0, whereas two dissimilar strings have a value of 1. This result can be subtracted from one and multiplied by one hundred to provide a similarity percentage.

As the generated song is likely to overfit on just one song in the dataset, the entire dataset is iterated through and the minimum LCS distance taken. During comparison, the two strings are truncated to the same length to eliminate irregular results. Across five different generated outputs from the final product, a mean similarity percentage of 74.3% is achieved. Ignoring rests, this average similarity is reduced to 55.7%. This latter metric is likely a superior novelty indicator. This is because rests are frequent in all songs, leading to long sequences of them being common, hence skewing the novelty evaluation.

```
LCS distance -
                With rests:      0.257528055671283
                Without rests:   0.442809001333029
```

Figure 25: Output from evaluation.py

Nonetheless, this similarity score does seem rather high. However, after running input songs through the evaluation metric (and disregarding the inevitable perfect match to itself), some perspective is provided. Mean similarities of 83.0% and 56.7% were recorded, with and without rests respectively, signifying that the generated tracks out-performed the existing songs. This is a superb result showing that novel music has indeed been created, hence satisfying the second research hypothesis. In spite of this, it should be noted that the LCS metric may not in fact be a viable metric, whereby a low distance may not completely correlate to similar songs. Therefore, this result should be taken with some reservations until further analysis of this method, and its suitability for music evaluation, can be embarked upon.

6.3 Technical Component Evaluation

While this is predominantly a research project, there of course exists a technical core which was used to undertake the research analysis. This environment was satisfactory for the problem at hand and was capable of dealing with the project needs. The questions listed in the project specification shall now be acknowledged.

"Has the input been encoded in a fitting fashion?"

The encoding was suitable for the input representation. It expressed a sufficient level of complexity found within the musical data, allowing music of a similar complexity to be generated. There were problems with the conveying of this musical information, however these issues were successfully resolved during the project's iterations.

"Has the dataset been able to sufficiently train the neural network?"

Yes, to a degree. Although the network trained well and the results were acceptable, a larger dataset would likely have further augmented these results. This extended dataset would have been especially useful in reducing overfitting and helping the model to learn more temporal patterns, permitting novel music to be of a higher standard. Unfortunately, this dataset could not easily be enlarged due to the finite number of resources available online.

"Did the neural network learn patterns and thematic structures in the style of house music?"

The neural network learnt patterns and long-term dependencies which allowed it to produce music the same style as the input dataset. As the input dataset consisted of house tracks, the generated music possessed behaviour and characteristics of a similar nature. Such features included a fast, constant rhythm and persistent use of drums and synthesisers.

"Did the software used, e.g., TensorFlow, function as intended?"

The software environment was able to accommodate everything requested of it. The use of Python and its libraries functioned seamlessly. TensorFlow operated correctly, successfully training the recurrent neural network, and integrated well with the GPU to provide CUDA-based acceleration which greatly sped-up training.

In summary, the technical component was sufficient for the project's requirements. The only qualm was that of the restricted dataset size.

7 Critical Assessment

The project underwent several revisions, due to the incessant challenges encountered, but the final product performed relatively successfully. As discussed in the evaluation section, some survey respondents believed that the generated tracks were created by a human. The quality was also comparable to that of human-produced tracks. However, the novelty could perhaps be improved. This could be achieved through increasing the dropout rate and using a temperature of 1.0 during sampling, but this is would likely have a detrimental effect on the quality. The most significant bottleneck of the project was the dataset; although unavoidable, it was realistically too small for the deep neural network. A larger dataset would allow the model to generalise better, consequently improving the quality and novelty of generated music.

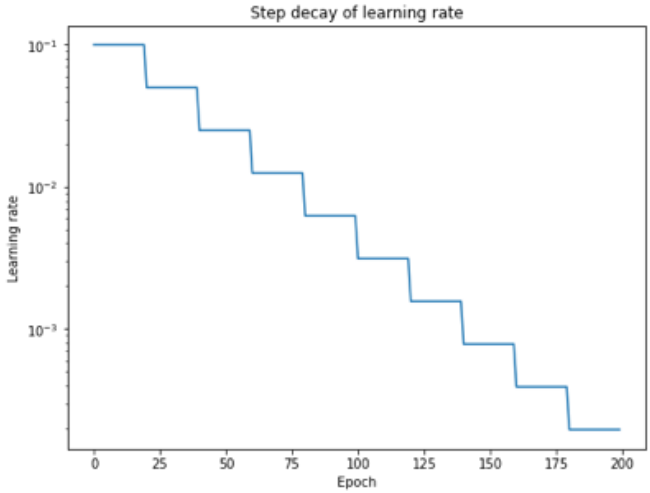

Figure 26: Learning rate annealing

Recurrent neural networks were an acceptable technique for the generation of music. The only trouble with them was the tuning of the model, which is essentially a *"dark art"* at this point. However, the visualisation tool, TensorBoard, did tremendously help with this. In hindsight, it may have been worth using step decaying annealing methods to help hone hyperparameters. As seen above, this is where variables are reduced by some percentage after a specified number of epochs has elapsed.

Despite prototype two not contributing a lot to the final product, yet costing time and effort, it was still a valuable endeavour. It exposed a range of different tools and the avenue could well have proved to be favourable. Most importantly, it showed that this project was dedicated to producing the best possible house music by not being afraid to change directions and push boundaries in order to achieve this.

The technical component was adequate. Additionally, the Google Cloud Platform had a major impact on the project's success, allowing the recurrent neural networks to be trained without interruption and at a much faster rate. Without the use of this technology, considerably less experimentation would have been possible.

8 Conclusion

In summary, the results were more than satisfactory, fulfilling the research questions, and will hopefully inspire others to launch research projects into this field. The unique multi-instrument encoding scheme provided sufficient representational capabilities to produce house music conveying a diverse range of traits. The audience evaluation, simulated through the online survey, allowed this music to then be objectively judged. The innovative novelty evaluation technique also worked well, providing a quantitative metric for something which is otherwise difficult to measure, but does necessitate some further research.

In the future, there are paths which could be examined to provide an even more superior final outcome. The scope of the research could be broadened, targeting the EDM genre as a whole instead of a specific sub-genre. This would help solve the dataset limitations. Furthermore, a biaxial multi-directional neural network [27] approach could be investigated. This is a distinct architecture which can be used to support multiple instruments. Unfortunately, due to time constraints, this method could not be explored. It would also be beneficial to evaluate music generated by RNNs with respect to other generational techniques, such as genetic programming. This could lead to a hybrid approach being developed, harnessing the power of both methods to potentially out-perform its constituent parts.

While algorithmic music generation will not replace human composition anytime soon, hopefully more unified environments will be seen whereby human artists create their songs with the assistance of artificial intelligence. For example, using a recurrent neural network to generate original drum loops.

References

[1] D. Eck and J. Schmidhuber, "A first look at music composition using lstm recurrent neural networks," tech. rep., 2002.

[2] H. Hild, J. Feulner, and W. Menzel, "Harmonet: A neural net for harmonizing chorales in the style of j.s.bach," in *Proceedings of the 4th International Conference on Neural Information Processing Systems*, NIPS'91, (San Francisco, CA, USA), pp. 267–274, Morgan Kaufmann Publishers Inc., 1991.

[3] A. Huang and R. Wu, "Deep learning for music," *CoRR*, vol. abs/1606.04930, 2016.

[4] A. Nayebi and M. Vitelli, "Gruv : Algorithmic music generation using recurrent neural networks," 2015.

[5] G. Nierhaus, *Algorithmic Composition: Paradigms of Automated Music Generation*. Springer, 2010.

[6] J. Schmidhuber, "Deep Learning in Neural Networks: An Overview," apr 2014.

[7] W. S. McCulloch and W. H. Pitts, "A Logical Calculus of the Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics*, vol. 7, pp. 115–133, 1943.

[8] I. Goodman and S. Pai, "CS 224D Final Project DeepRock," 2016.

[9] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber, "Gradient flow in recurrent nets: the difficulty of learning long-term dependencies," 2001.

[10] M. T. Hagan, H. B. Demuth, M. Beale, and O. De Jesus, *Neural Network Design*. 2 ed., 2014.

[11] "Oxford Learner's Dictionaries." https://www.oxfordlearnersdictionaries.com, accessed 2018-11-21.

[12] "Keras: The Python Deep Learning library." https://keras.io, accessed 2019-02-11.

[13] "TensorFlow: An open source machine learning library for research and production." https://www.tensorflow.org, accessed 2019-02-11.

[14] "Classical Piano Midi Page." http://www.piano-midi.de, accessed 2018-12-21.

[15] "music21: a Toolkit for Computer-Aided Musicology." https://web.mit.edu/music21, accessed 2018-12-22.

[16] "Sample Generated Tracks." https://fred.glass/housyn.html, accessed 2019-03-19.

[17] "Pydub by Jiaaro." http://pydub.com, accessed 2019-03-19.

[18] A. M. Noll, "Memories: A Personal History of Bell Telephone Laboratories," 1959.

[19] "Carlo's MIDI." https://www.cprato.com, accessed 2019-03-24.

[20] "nonstop2k: Electronic Dance Music MIDI Files." https://www.nonstop2k.com, accessed 2019-03-24.

[21] "Mido - MIDI Object for Python." https://mido.readthedocs.io/en/latest, accessed 2019-03-24.

[22] "Multi-layer Recurrent Neural Networks (LSTM, RNN) for character-level language models." https://github.com/sherjilozair/char-rnn-tensorflow, accessed 2019-03-27.

[23] M. Mozer, "Neural network music composition by prediction: Exploring the benefits of psychoacoustic constraints and multi-scale processing," *Connection Science - CONNECTION*, vol. 6, pp. 247–280, 01 1994.

[24] A. M. Turing, "Computing Machinery and Intelligence," *Mind*, vol. LIX, no. 236, pp. 433–460, 1950.

[25] J. Romero and P. Machado, *The Art of Artificial Evolution.* Springer, 2007.

[26] D. Bakkelund, "An lcs-based string metric," 2009.

[27] D. Johnson, "Generating polyphonic music using tied parallel networks," 2017.