Developing A Machine Learning Model Able to Produce MIDI Files to Aid in Music Production

Cameron Herbert

MSc in Advanced Computer Science,
School of Computing Science, Newcastle University.
c.herbert2@ncl.ac.uk

Abstract: The evolution of technology continues to change the music making process. In recent years, machine learning has become a trending topic in the music world. Models that can generate seprate musical elements, or even complete songs have become increasingly popular. The majority of these models generate music solely for the purpose of enjoyment. Models that can be used for the music creation are mostly involved with the creation of samples or drum patterns. This project intends to develop a model that can create melodies to be used as the starting point in the process of creating an instrumental. These melodies will be created in the form of MIDI files, often used by producers looking to collaborate with others. These files contain just the melody, leaving the recipient to make their own choices regarding which sounds to use. This files also allow for the recipient to rearrange notes how they see fit. A model that can generate MIDI files suitable for the production of music is extremely useful for producers looking for inspiration.

1. Introduction

In the music production world, the process of beat-making plays a crucial role in the creation of catchy and rhythmically engaging tracks. As technology continues to evolve, the integration of Machine Learning technology has opened up new avenues for musicians and producers to explore. Musical Instrument Digital Interface (MIDI) files serve as the foundation for melodies, allowing producers to share and collaborate on music production. The advantage lies in the recipient's ability to customize the melody, rearrange notes, and experiment with different sounds. Models that generate music samples either for the purpose of enjoyment or music creation are quite common. Unlike samples, MIDI files allow recipients to modify the melody according to their preferences, making it an ideal format for collaborative music production.

A model that could generate MIDI files in a desired style is something that would have incredible use in the music production world. Producers often face writers when trying to create melodies for an instrumental. This is especially common for

producers who have to create a certain amount of melodies day. When facing writers block, the first thought is often to find a good sample or ask another producer for MIDI files. The model this project intends to create provides another option to solve this common issue. Newer styles of music often pull inspiration from multiple different genres. It is important that this model has the ability to generate melodies reflecting elements from different genres so it can keep up to date with the current trends in the music world.

1.1 Aims and Objectives

Aim: To develop a machine learning model that generates MIDI files, suitable for use as the starting point for the creation of instrumentals in a desired style drawing influence from multiple genres.

Objectives:

- Research types of machine learning models commonly used for music generation
- Gather a large amount of MIDI files
- Write code to parse MIDI files to store NOTE ON, NOTE OFF, and delta-time data as sequences in a data structure
- Develop a machine learning model that can learn to predict the next note in a given sequence and ultimately generate its own sequences that can be translated into MIDI files suitable for use in music production

2. Background

2.1. MIDI files

MIDI files can be thought of as a digital version of sheet music. The files contain information about the pitch of a note, when the note begins, and when it ends. The pitch is represented by an integer between 0 and 127 which is called a note number. There are 128 possible pitches for a note. The beginning of a note is represented by a NOTE ON message which contains the note number. This message also contains a number to represent velocity (how loud a note is played) as well as a number from 0 to 15 that corresponds to a MIDI channel. In the context of beat-making, the notes in MIDI are always sent with the default velocity of 64 since many of the digital instruments used by producers are not velocity sensitive. MIDI files used for beat-making also only contain one channel which allows the producer to control which notes get played by which instrument. A NOTE OFF message is used to represent the end of a note. It contains the same information as a NOTE ON message, but the velocity is set to 0. NOTE ON and NOTE OFF messages are referred to as events. Figure 1 shows how a MIDI file looks when opened by a Digital Audio Workstation (DAW) called Ableton.

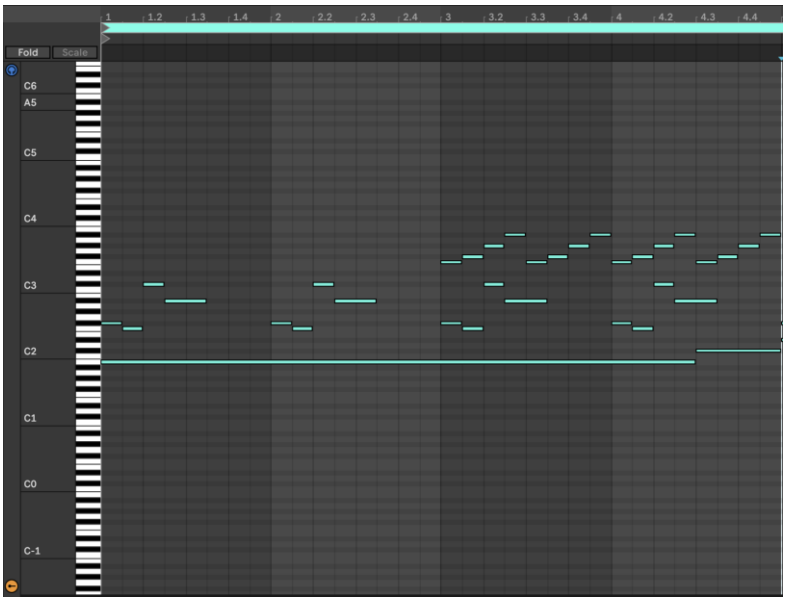


Figure 1. A MIDI file visualized as a digital piano roll in Ableton

A DAW is the software used by producers to make music. DAWs can be used to create and import midi files. Each row relates to one of the 128 note numbers used to identify pitch (not all 128 are shown). The columns, which are numbered at the top, each correspond to a beat which is a unit of time in music. The tempo of a track is represented by a unit of measurement called Beats Per Minute (BPM). In Figure 1, there are 4 beats between the numbers 1 and 2 at the top. MIDI files use ticks as a unit of time. There are 960 ticks per beat, allowing for more precision when creating notes. To handle timing, NOTE ON and NOTE OFF messages are sent with a delta-time value before them. This value represents the number of ticks that have passed since the previous event. Any notes that are played as soon as playback begins are given a delta-time value of 0. This means that 0 ticks should pass between the beginning of playback and the beginning of these notes. For example, the MIDI file shown in Figure 2 would begin with a NOTE ON message with a note number of 60 representing the pitch. This event would be preceded by a delta-time value of 0 since it should be played as soon as playback begins. The next event would be a NOTE OFF message with a note number of 60 representing the end of this first note. This NOTE OFF message would be preceded by a delta-time of 0 since the note ends one beat after the previous event which was the start of this note. The next event would be a NOTE ON message with a note number of 61. This event would be preceded by a delta-time of 0 since this note begins as soon as the previous note ends. The final event is a NOTE OFF message with a note number of 61, which is preceded with a delta-time value of 960 since the note ends 1 beat after the previous event.

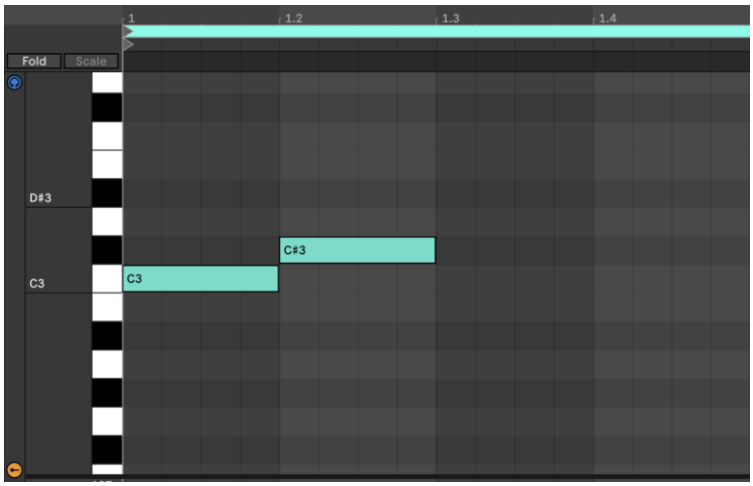


Figure 2. A simple MIDI file visualized as a digital piano roll in Ableton

2.2. Recurrent Neural Networks (RNNs)

Deep learning, a subset of machine learning, has been instrumental in various domains, including music generation and analysis. Neural networks, the backbone of deep learning, consist of interconnected processing units that can learn to represent complex patterns from data. Neural networks have been employed for various music related tasks such as chord classification, genre detection, sound synthesis, pitch perception modeling, and melody creation [1, 2, 3, 4]. Recurrent Neural Networks (RNNs) stand out due to their ability to recognize patterns in sequential data. Unlike traditional feedforward networks, RNNs possess connections that loop back, enabling them to retain information from previous inputs. This "memory" feature allows RNNs to consider historical context when producing an output, making them particularly suitable for tasks like music generation where past notes influence future ones.

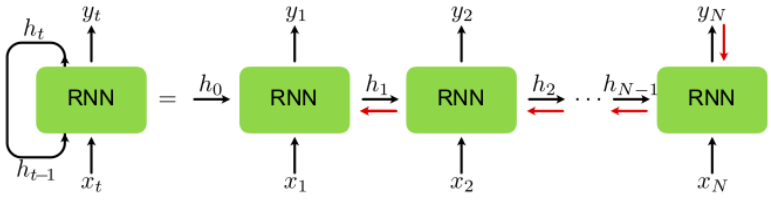


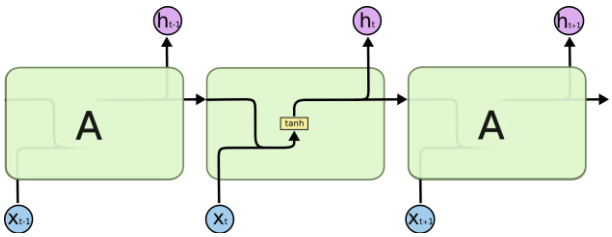
Figure 3. Diagram of structure of an RNN. Input x_t is fed into the RNN cell which produces output y_t where t represents a time step. h_{t-1} represents the hidden memory from the previous timestep which is used in the next time step to produce the next output. The red arrows show how gradients travel back through the network during a backwards pass [5].

In Figure 3, this hidden memory value is what makes RNNs suitable for sequential data. For a simple RNN, the hidden memory value is just the output from the previous time step. This is where RNNs get their pattern recognition abilities from.

The process depicted in Figure 3 by the black arrows is called a forward pass [5]. The next step is to compute the loss, which is a measure of how far off the model's predictions (outputs) were from the actual values. A smaller loss value means more accurate predictions. This is how an RNN evaluates its performance during training. The most common function used to calculate loss is called the cross-entropy function [6].
The loss is used to update weights throughout the network during a process called a backward pass, depicted in Figure 3 by the black arrows [5]. The inputs, outputs, and hidden memory values all have weights associated with them. The backwards pass's objective is to adjust these weights to minimize the loss. The most common way to do this is with an algorithm called Back Propagation Through Time (BPTT) [7]. This algorithm computes a gradient which tells it how much the loss will change when adjusting each weight. This process causes issues for RNNs when training with data that requires long term memory due to the vanishing gradient problem [8]. To solve this problem, Hochreiter and Schmidhuber [9] developed the Long Short Term Memory model.

2.3. Long Short Term Memory (LSTM)

LSTMs, a special kind of RNN, are designed to remember long-term dependencies in sequences much better than traditional RNNs. LSTMs, with their unique cell state and gating mechanisms, can remember valuable information over extended periods [8]. This ability makes them much better at learning and generating music, where motifs and themes might recur after a certain period. Figure 4 depicts the structure of both an RNN cell and an LSTM cell. It is easy to see how much more complex the structure of the



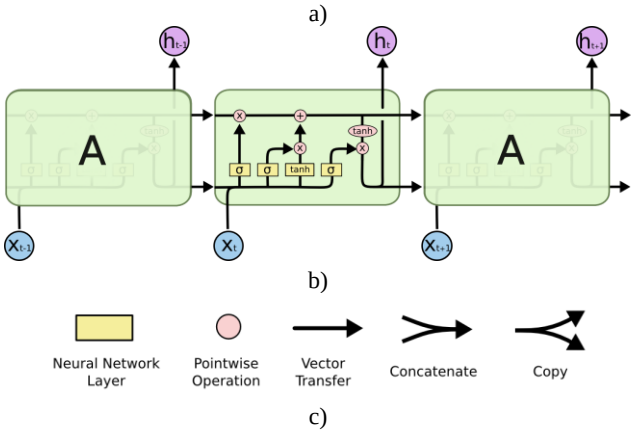


Figure 4. a) Diagram of the structure of an RNN cell A with a single tanh layer [10].
b) Diagram of the structure of an LSTM cell with 4 different layers [10].
c) Notation for symbols used in Figures 4a and 4b [10].

LSTM cell is. It contains 4 neural network layers as well as pointwise operations. To better understand how an LSTM cell works we will breakdown what these layers do step by step. Figure 5 shows the cell state of an LSTM cell as well as a gate. The cell state is very important to the way an LSTM RNN functions. It runs horiontally

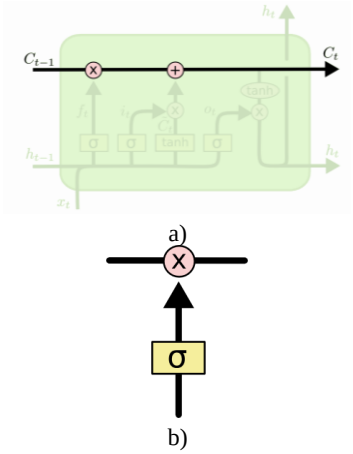


Figure 5. a) Diagram of the structure of an LSTM cell highlighting the cell state [10].
b) Diagram depicting a gate within an LSTM cell [10].

through all time steps [10]. This is where the LSTM gets its long-term memory from. The gate depicted in Figure 5a is comprised of a sigmoid neural network layer and a pointwise multiplication operation [10]. Each of the gates update the information in the cell state in different ways. Figure 6 depicts a forget gate inside an LSTM cell. This gate updates the cell state by deciding which information is forgotten [10]. This gate receives the hidden memory value (h_{t-1}) and the input (x_t) then produces a number

ranging from 0 to 1 for all the values in the cell state (c_{t-1}) [10]. A 1 means keep this value and a 0

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

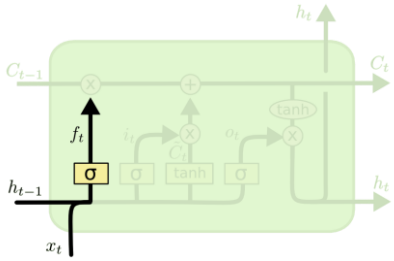


Figure 6. Diagram of the structure of a forget gate in an LSTM cell [10].

means get rid of this value [10]. The calculation of this value is shown in Formula 1, where W_f is the weight associated with the forget gate and b_f is the bias associated with the forget gate.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \quad (2)$$

$$\tilde{C_t} = \tanh\left(W_c \cdot \left[h_{t-1,} x_t\right] + b_c\right) \quad (3)$$

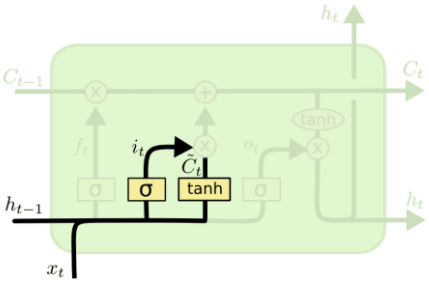


Figure 7. Diagram of the structure of an input gate in an LSTM cell [11].

Figure 7 depicts an input gate in an LSTM cell. This gate decides what new information is stored in the cell state [10]. This is done using a sigmoid layer that decides which values in the cell state to update, denoted by i_t [10]. There is also a tanh layer that creates a vector of new values that may be added to the cell state denoted by $\tilde{C}_t$. [10]. Figure 8 shows what happens to the three values computed by the forget and

$$C_t = f_t * C_{t-1} + i_t * C_t' \quad (4)$$

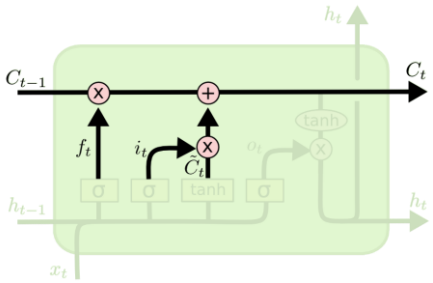


Figure 8. Diagram depicting how the values from forget and input gates layers are used to update the cell state [10].

input gates. In [10] Christopher Olah says "We multiply the old state by f_t, forgetting the things we decided to forget earlier. Then we add $i_t * \tilde{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value" [10] to describe this process. Figure 9 shows how an LSTM cell calculates its output.

$$o_t = \sigma(W_o \cdot [h_{t-1,} x_t] + b_o) \quad (5)$$

$$h_t = o_t \cdot \tanh(C_t) \quad (6)$$

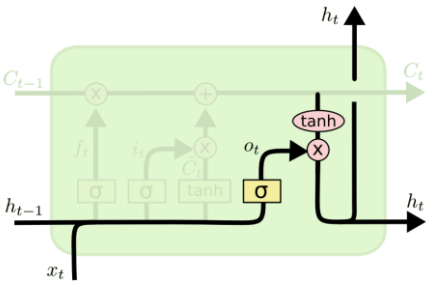


Figure 9. Diagram depicting the structure of an output gate in an LSTM cell [10].

Like the forget gate, a sigmoid layer takes the input and hidden memory and produces a number from 0 to 1 for each value in the cell state. This number determines how much each value in the cell state should influence the output. The Cell state goes through tanh layer then is multiplied by the output of the sigmoid layer [10].

Earlier on in Figure 3, h_t was used to represent the hidden state value that is output by an RNN cell to serve as the hidden memory (h_{t-1}) for the cell during the next time step. The output was represented by a separate value, y_t. The h_t and y_t values can be thought of as the same value. The hidden memory is just the output from the previous time step which is used to produce the output at the current time step. This process is how an RNN gets its short-term memory capabilities. RNNs possess the same short-term memory capabilities, but also have long-term memory capabilities due to the addition of the cell state. This long-term memory is very important for the music related tasks LSTMs are commonly used for. For the generation of music, the short-term memory capabilities, provided by the hidden memory, are used to continue

shorter trends based off recent notes. The long-term memory capabilities, provided by the cell state, are used for the continuation of longer trends based off notes much further back in the sequence. The cell state affects outputs with information that is too far back for the hidden memory to recall.

2.4. Related Work

The most helpful information came from a research paper written by Oore et al., outlining the process of training an LSTM to generate MIDI files that replicate a human classical piano performance [11]. To do this, a large database of MIDI files created from classical piano performances was used. These performances were done on a piano that could save each performance as MIDI file. When training the LSTM model, Oore et al. devised a very useful method for representing the MIDI data. A vocabulary of 413 events was used to represent the MIDI data as a sequence of events. Since there are 128 possible MIDI pitches, the vocabulary contains 128 NOTE ON and 128 NOTE OFF events. There are also 125 time-shift events and 32 velocity events. To explain the time-shift events, Oore et al writes "While the minimal time step is a fixed absolute size (8 ms), the model can skip forward in time to the next note event. Thus, any time steps that contain rests or simply hold existing notes can be skipped with a single event. The largest possible single time shift in our case is 1 second but time shifts can be applied consecutively to allow effectively longer shifts. The combination of fine quantization and time-shift events helps maintain expressiveness in note timings while greatly reducing sequence length compared to an uncompressed representation" [11]. The 32 velocity events are used to represent the 32 possible velocities a note can be played at.
The LSTM model operates on a one-hot encoding scheme using this event vocabulary. This means that at each step, the input to the model is a singular one-hot vector with 413 dimensions. Figure 5 created by Oore et al. shows how this vocabulary is used to represent a small excerpt from a MIDI file. Notice how each TIME-SHIFT is a multiple of 8.

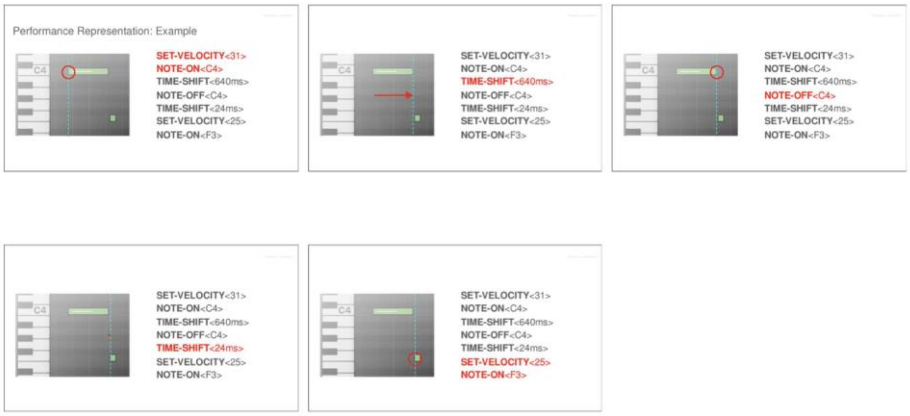


Figure 5. "Example of Representation used for PerformanceRNN. The progression illustrates how a MIDI sequence (e.g. shown as a MIDI roll consisting of a long note

followed by a shorter note) is converted into a sequence of commands (on the right hand side) in our event vocabulary. Note that an arbitrary number of events can in principle occur between two time shifts" [11].

There are certain elements of the work done by Oore et al. that do not pertain to the aim of this project, which is the generation of MIDI files that can be used for the creation of instrumentals. First was the addition of the velocity events which were left out for reasons explained earlier. This decreases the size of the event vocabulary from 413 to 381. The other element was the prediction of a sustain pedal. The sustain pedal increases the duration of notes played while the pedal is being held down. This element is unnecessary since the MIDI files used for the creation of instrumentals are not only meant for pianos. A producer may take one part of the MIDI file and have a guitar play the selected notes, then take another part of the file and have a synthesizer play the other notes. Oore et al. Also transposed the training data up and down, as well as stretching the timing of the training data to create multiple alterations of the same MIDI file. This could not be implemented due to memory constraints.

3. Development approach

1.2 Collecting Data

The development process began with gathering a large selection of MIDI files that fit the style of music the LSTM aims to generate. The MIDI files generated should be useful as the starting point for making melodies for the newer style of hip hop music. These melodies pull a lot of influence from pop, electronic dance music, rock, and an older style of hip hop music called trap. In order to create a balanced data set, MIDI files were collected from producers that are influenced by the four genres previously mentioned, as well as MIDI files that were created within the realms of these four genres. This ensured a blend of both authentic and influenced melodies, creating a training environment that allows the LSTM to learn how to combine certain features from each genre. The end of this process resulted in a diverse collection of 1261 MIDI files.

1.3 First Approach

The first approach began before the paper written by Oore et al. [11] was found. It was decided early on that Python would be the programming language used for this project. Python was especially useful for this project since it has libraries for pulling data from MIDI files and training machine learning models. Jupyter Notebook was the chosen environment for this project. The ability to execute small sections of code at a time and immediately view the results was greatly beneficial during the development stage. This notebook was hosted on Microsoft Visual Studio Code. The first step

was to devise a way to extract the necessary data from the MIDI files that were to be used for training. This was done using the pretty_midi library created by Colin Rafael and Daniel P. W. Ellis [12].

Utilizing the pretty_midi library, the script reads a specified MIDI file and iterates over the instruments and notes contained therein. For each note encountered the name of the midi file it is from, the pitch, the start time, and the duration of the note are extracted and aggregated into a pandas DataFrame. The data in the DataFrame is then grouped by the file name then added into a dictionary as separate DataFrames. The file name column is dropped during this process. Sequences containing the data from 20 notes are then extracted from the DataFrames using the sliding window approach. These notes are then put into two different lists. One list will be used as the inputs for training and the other list will be used as the targets. Each element of the input list will contain tdata representing 20 sequential notes, and each element in the target list will contain data representing 21st note. To prepare for training, the input and target lists are converted into NumPy arrays.

The LSTM model was set up using the Keras library which runs on top of the TensorFlow library. The Keras library was chosen for the first approach since it is made to be user friendly. However, the user-friendly nature of their APIs does not take away from the power they provide as they are used in some high-grade applications such as the recommendations on YouTube. The model was set up with five different layers. The first was an LSTM layer with 256 cells. The input shape for this layer is set to (20, 3), because this layer receives sequences of 20 notes and each note is represented by a 3-dimensional vector. These vectors contain the pitch, start time, and duration of each note. The LSTM will take these notes and process them sequentially attempting to identify patterns and dependecies. The output from the LSTM layer will have the shape (20, 256). Now each note is represented by a much more complex 256-dimensional vector. This vector primarily contains information from hidden state, generated during the processes that occur in each LSTM cell explained earlier. This output is then passed through a flattening layer. This layer takes the 2D array of shape (20, 256) and flattens it to a 1D array with 5120 elements.

This array is then passed through three dense layers with three units each. These layers take the complex representation in this array and perform transformations on it to capture patterns and relationships in this data, learning how to produce an accurate prediction. The output from the final dense layer is a 3-dimensional vector representing model's prediction of the next note in the sequence. The relationship between the LSTM layer and the dense layers is very important. To put it simply, the LSTM layer identifies the "building blocks" of the musical data. The dense layers learn how to assemble these blocks. The dense layers take this knowledge and use it to predict the next note.

A batch size of 64 was chosen for this approach, meaning 64 input sequences are fed into this model at once. The model then produces a prediction for the next note for each of these 64 sequences. This process is the forward pass which was explained earlier. For this approach, the Mean Squared Error (MSE) function was chosen to calculate the loss, and the Adam algorithm was chosen for optimization. The MSE function calculates the loss by comparing the model's prediction to the target. The

Adam optimizer takes the output from the MSE function and uses this to update the weights and biases throughout the model to minimize the loss value. This process is the backward pass explained earlier, which concludes a single training iteration which is called an epoch.

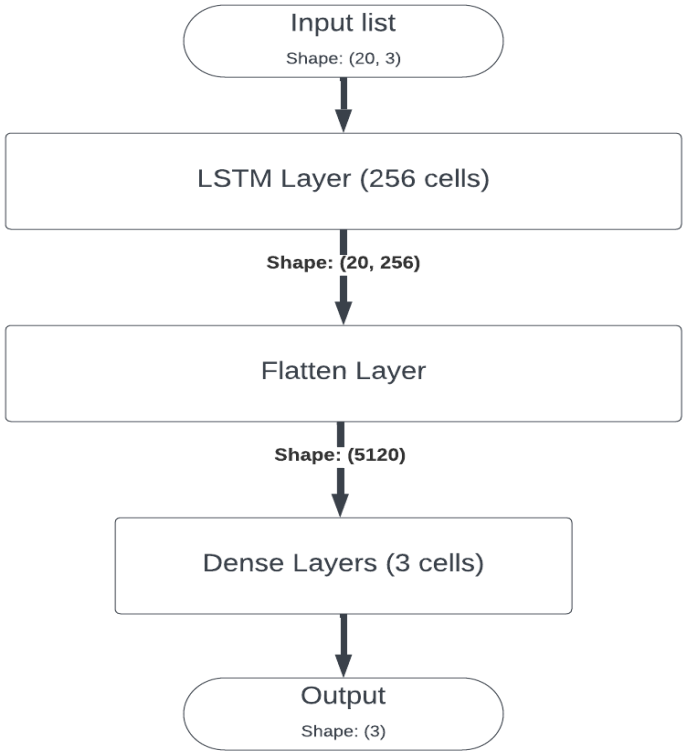


Figure 6. Diagram depicting the structure of the model created during the first approach

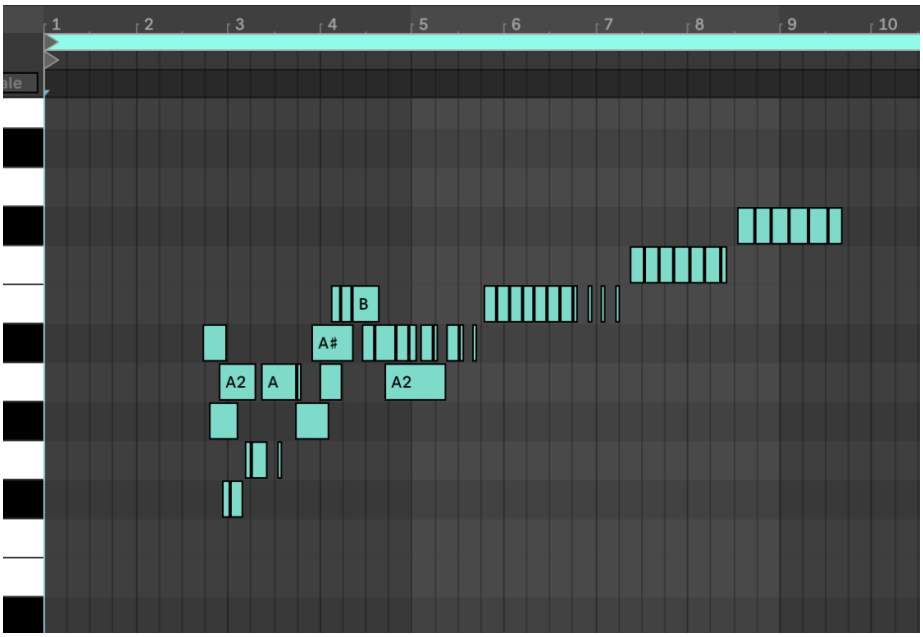
To create this model, a *Sequential* model object is instantiated from the Keras library. Then the *add* method is used to add the LSTM, flatten, and dense layers to this model. For the LSTM layer, the number of cells and input shape are specified as parameters. A parameter called *return_sequences* is also set to *True* which means this layer will produce an output for every time step. For the dense layers, the number of cells at each layer is also specified. Once the model structure is defined, it is then compiled using the *compile* method available in Keras. The compile method takes the chosen optimizer and loss functions as parameters. The training process is then initiated by calling the *fit* method which takes the input list, output list, batch size, and number of epochs as parameters.

Once this model is trained, it is used to generate a new sequence of notes. First, an input sequence is selected to use as the starting point by the model during generation. The *predict* method takes the seed sequence as a parameter and outputs a

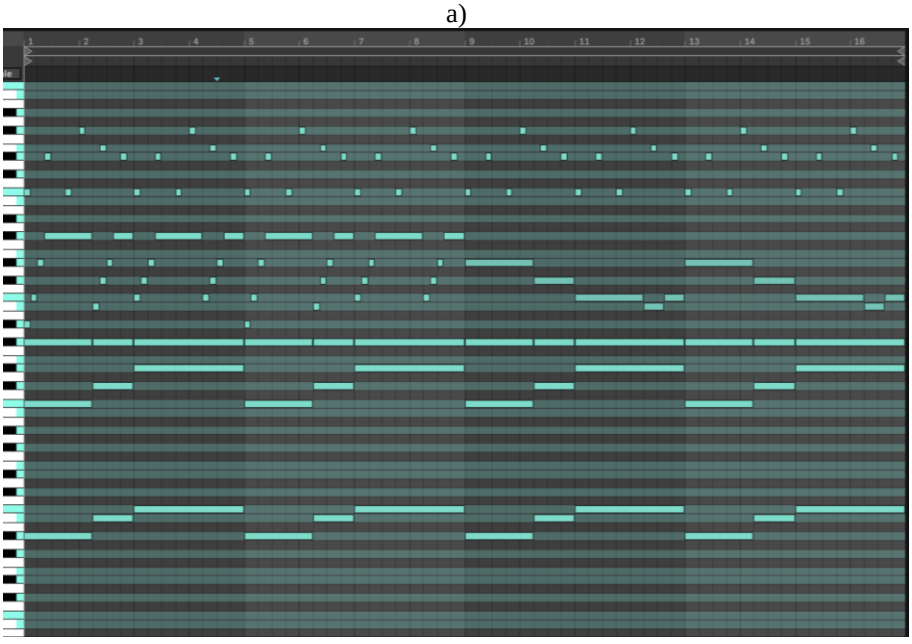
prediction for the next note. This predicted note is added to an output list. The *vstack* method from the NumPy library is then used to update the seed sequence. It takes the seed sequence with the first note removed, and the predicted note as parameters. It outputs the updated seed sequence to be used for the next prediction. To explain further, if the seed sequence has notes ABCD, note E is predicted then the seed sequence is updated to contain notes BCDE. For the first approach, the generation is done using a seed sequence with a length of 20. This process was done in a loop that iterates 50 times to generate a new sequence of 50 notes. The final step is to take this sequence and convert it into a MIDI file which is done using the pretty_midi library.

1.4 First Approach Results

The initial training of this model was done for 10 epochs and resulted in a loss of 43.4228. The MIDI file generated by this model was nowhere near suitable for use in music production. The timing of the notes was extremely random with no sense of rhythm or tempo being displayed. The notes were only generated at eight different pitches that were all consecutive (54-61) and didn't follow any kind of key signature. This generated MIDI file is shown in Figure 7a. To produce a more thoroughly trained model, the number of epochs was increased from 10 to 500. This model was able to achieve a loss of 0.1912, but the MIDI file it generated was not much better. The pitch of the notes in this new file ranged from 46 to 69 and were not all consecutive. However, these notes still did not follow any kind of key signature and the timing of the notes was again very random. Figure 7b depicts one of the MIDI files from the training data. The pitch of the notes follow a key signature (F# Major) which has been highlighted. There are never more than two notes that are right next to each other in a key signature. There is also a pattern in the timing of these notes. The starts and ends of these notes line up with the columns that represent a beat. The notes either start and end directly on these beats, or exactly halfway through these beats. There is also an overall structure that these notes follow which is important to the function of this specific type of MIDI file. Towards the bottom there are notes being played at the same time which are held for longer, these represent chords. Above these notes there are melodies being played to complement the chords. There is also a repetitive structure being displayed. The first half is almost identical to the second half, except the melody just above the chords is changed. MIDI files are often organized this way so the producer can maybe use one half for the chorus of the instrumental and the other half for the verse. It is also common for these MIDI files to be completely symmetrical, allowing the producer to make their own modifications to the second half. It is easy to see that the model trained during this first approach was having trouble recognizing the patterns necessary for it to create a suitable MIDI file.



a)



b)

Figure 7. a) MIDI file that was generated by the model trained during the first approach for 10 epochs visualized as a digital piano roll in Ableton.
b) MIDI file from the training data set visualized as a digital piano roll in Ableton. A Major scale is highlighted to show the notes follow a key signature

Changes were made to the sequence length, batch size, and the number of cells at the LSTM and dense layers, but the quality of generated MIDI files was still very poor. While searching for existing work to find a better method for training a model using MIDI data, a GitHub repository for training a transformer (a different kind of neural network) to generate MIDI files belonging to Aditya Gomatam was found [13]. In the "readme" description for this repository, Gomatam mentions the event vocabulary from [11] which was explained earlier in the related work section.

1.5 Second Approach

For the second approach, it was decided that the vocabulary outlined by Oore et al. should be used to represent the MIDI data. It was also decided that the Jupyter Notebook for this approach should be hosted on Google Colab, since it provides access to powerful GPUs and TPUs which significantly speed up training time (as seen in [13]). It also has certain Python libraries pre-installed. Using these libraries on Microsoft Visual Studio Code required setting up a Python virtual environment. The implementation for this approach was aslo done using PyTorch instead of Keras, after seeing that this is what Gomatam used in [13]. Using Keras would have prevented the need to learn a new library, but it was decided that having experience with both would be beneficial in the long run.

The repository from Gomatam provided a midi_parser function for extracting data from the MIDI files, then translating it into the Oore et al. vocabulary. This function uses the Mido library as opposed to the pretty_midi library used in the first approach. Some changes were made to this function to remove parts dealing with the velocity and sustain pedal. This function utilizes both a time_cutter and time_to_events function created by Gomatam to handle the time shift events explained earlier when parsing the MIDI files from the training data set. This midi_parser function returns both an index list and event list. The event list contains a representation of the MIDI file using the Oore et al. vocabulary in the form of strings. This would be difficult to use for training, so Gomatam provided a way to represent the MIDI data numerically while still utilizing the event vocabulary. The index list returned contains integers that each point to an index in the vocab list where each element is an event from the vocabulary as stored as a string [13]. The creation of this vocab list is shown in Figure 8. This implementation includes the addition of a pad token which is not a part of the vocabulary created by Oore et al.

```python
note_on_events = 128
note_off_events = note_on_events
note_events = note_on_events + note_off_events
time_shift = 8
max_time_shift = 1000
time_shift_events = max_time_shift // time_shift
total_midi_events = note_on_events + note_off_events + time_shift_events

# create vocabulary
note_on_vocab = [f"note_on_{i}" for i in range(note_on_events)]
note_off_vocab = [f"note_off_{i}" for i in range(note_off_events)]
time_shift_vocab = [f"time_shift_{i}" for i in range(time_shift_events)]
vocab = ['<pad>'] + note_on_vocab + note_off_vocab + time_shift_vocab
vocab_size = len(vocab)
pad_token = vocab.index("<pad>")
```

Figure 8. Python code from Gomatam's repository that creates the vocab list allowing MIDI files to be represented numerically [13]. Minor alterations were made removing the creation of velocity events, renaming variables for easier understanding, and removing start and end tokens.

Each MIDI file is parsed using Gomatam's midi_parser function. When the index list is returned, it is converted to LongTensor which is a type of PyTorch tensor that holds integer values. These tensors are then collected into a larger list and shortened to a length of 220. When converting back into MIDI files, it was observed that a length of 220 was long enough to capture the repetition of certain patterns in the training files. The average length of the training MIDI files was 220. Increasing this number would result in the majority of the sequences needing to be padded. Any tensor shorter than 220 is padded using the pad_sequence function from PyTorch. This process ensures a uniform length for each tensor, which is important since this is a requirement for batch processing. The result is a list of tensors, representing a sequence of events from each of the MIDI files from the training data set. Figure 9 displays both the index tensor and event list representation of a MIDI file after it has been parsed.

```
Index List:
tensor([ 87,  75,  70,  44,  60, 286, 215, 203, 286, 198, 172, 188,  70,  44,
         60,  77, 286, 205,  75, 286, 198, 172, 188, 203,  89, 286, 217,  75,
        286, 203,  77, 286, 205,  79, 286, 207,  91,  46,  62,  74, 286, 219,
        286, 174, 190, 202,  46,  62,  74,  77, 286, 205,  75, 286, 174, 190,
        202, 203,  89,  74,  62,  46, 286, 217,  75, 286, 202, 190, 174, 203,
         46,  62,  74,  82, 286, 210,  74, 286, 174, 190, 202, 202,  48,  63,
         75,  87, 286, 215, 286, 176, 191, 203,  77,  48,  63,  75, 286, 205,
         75, 286, 176, 191, 203, 203,  89, 286, 217,  75, 286, 203,  77, 286,
        205,  79, 286, 207,  55,  70,  62,  91, 286, 219, 286, 183, 198, 190,
         77,  62,  70,  55, 286, 205,  75, 286, 190, 198, 183, 203,  94,  62,
         70,  55, 286, 222,  75, 286, 190, 198, 183, 203,  82,  62,  70,  55,
        286, 210,  74, 286, 190, 198, 183, 202,  87,  75,  70,  44,  60, 286,
        215, 203, 286, 198, 172, 188,  70,  44,  60,  77, 286, 205,  75, 286,
        198, 172, 188, 203,  89, 286, 217,  75, 286, 203,  77, 286, 205,  79,
        286, 207,  91,  46,  58,  65,  74, 286, 219, 286, 174, 186, 193, 202,
         46,  58,  65,  74,  77, 286, 205,  75, 286, 174])
Event List:
['note_on_86', 'note_on_74', 'note_on_69', 'note_on_43', 'note_on_59', 'time_shift_29', 'note_off_86', 'note_off_74', 'time_shift_29', 'note_off_69', 'note_off_43',
'note_off_59', 'note_on_69', 'note_on_43', 'note_on_59', 'note_on_76', 'time_shift_29', 'note_off_76', 'note_on_74', 'time_shift_29', 'note_off_69', 'note_off_43',
'note_off_59', 'note_off_74', 'note_on_88', 'time_shift_29', 'note_off_88', 'note_on_74', 'time_shift_29', 'note_off_74', 'note_on_76', 'time_shift_29', 'note_off_76',
'note_on_78', 'time_shift_29', 'note_off_78', 'note_on_90', 'note_on_45', 'note_on_61', 'note_on_73', 'time_shift_29', 'note_off_90', 'time_shift_29', 'note_off_45',
'note_off_61', 'note_off_73', 'note_on_45', 'note_on_61', 'note_on_73', 'note_on_76', 'time_shift_29', 'note_off_76', 'note_on_74', 'time_shift_29', 'note_off_45',
'note_off_61', 'note_off_73', 'note_off_74', 'note_on_88', 'note_on_73', 'note_on_61', 'note_on_45', 'time_shift_29', 'note_off_88', 'note_on_74', 'time_shift_29',
'note_off_73', 'note_off_61', 'note_off_45', 'note_off_74', 'note_on_45', 'note_on_61', 'note_on_73', 'note_on_81', 'time_shift_29', 'note_off_81', 'note_on_73',
'time_shift_29', 'note_off_45', 'note_off_61', 'note_off_73', 'note_off_73', 'note_on_47', 'note_on_62', 'note_on_74', 'note_on_86', 'time_shift_29', 'note_off_86',
'time_shift_29', 'note_off_47', 'note_off_62', 'note_off_74', 'note_on_76', 'note_on_47', 'note_on_62', 'note_on_74', 'time_shift_29', 'note_off_76', 'note_on_74',
'time_shift_29', 'note_off_47', 'note_off_62', 'note_off_74', 'note_off_74', 'note_on_88', 'time_shift_29', 'note_off_88', 'note_on_74', 'time_shift_29', 'note_off_74',
'note_on_76', 'time_shift_29', 'note_off_76', 'note_on_78', 'time_shift_29', 'note_off_78', 'note_on_54', 'note_on_69', 'note_on_61', 'note_on_90', 'time_shift_29',
'note_off_90', 'time_shift_29', 'note_off_54', 'note_off_69', 'note_off_61', 'note_on_76', 'note_on_61', 'note_on_69', 'note_on_54', 'time_shift_29', 'note_off_76',
'note_on_74', 'time_shift_29', 'note_off_61', 'note_off_69', 'note_off_54', 'note_off_74', 'note_on_93', 'note_on_61', 'note_on_69', 'note_on_54', 'time_shift_29',
'note_off_93', 'note_on_74', 'time_shift_29', 'note_off_61', 'note_off_69', 'note_off_54', 'note_off_74', 'note_on_81', 'note_on_61', 'note_on_69', 'note_on_54',
'time_shift_29', 'note_off_81', 'note_on_73', 'time_shift_29', 'note_off_61', 'note_off_69', 'note_off_54', 'note_off_73', 'note_on_86', 'note_on_74', 'note_on_69',
'note_on_43', 'note_on_59', 'time_shift_29', 'note_off_86', 'note_off_74', 'time_shift_29', 'note_off_69', 'note_off_43', 'note_off_59', 'note_on_69', 'note_on_43',
'note_on_59', 'note_on_76', 'time_shift_29', 'note_off_76', 'note_on_74', 'time_shift_29', 'note_off_69', 'note_off_43', 'note_off_59', 'note_off_74', 'note_on_88',
'time_shift_29', 'note_off_88', 'note_on_74', 'time_shift_29', 'note_off_74', 'note_on_76', 'time_shift_29', 'note_off_76', 'note_on_78', 'time_shift_29', 'note_off_78',
'note_on_90', 'note_on_45', 'note_on_57', 'note_on_64', 'note_on_73', 'time_shift_29', 'note_off_90', 'time_shift_29', 'note_off_45', 'note_off_57', 'note_off_64',
'note_off_73', 'note_on_45', 'note_on_57', 'note_on_64', 'note_on_73', 'note_on_76', 'time_shift_29', 'note_off_76', 'note_on_74', 'time_shift_29', 'note_off_45']
```

Figure 9. Representation of a MIDI file as both an index tensor (top) and an event list (bottom) after being parsed using the midi_parser function from Gomatam [13].

The structure and set up of the LSTM model was also changed in this second approach. This started with the way the input and target sequences are created. The training process done in the first approach was limited. The model only learned to predict one note as the target from a sequence of 20 notes. This provided the model with no context to any of the longer-term patterns. In this approach, the input and target sequences created both have a length of 219. The input sequence is composed of events 1-219, and the target sequence contains events 2-220. The model takes the first event from the input sequence and tries to predict the second event. The model then takes the second event from the input sequence and tries to predict the third event. This process continues until it has made predictions for events 2-220 which are then compared to the target sequence. After creating a list for the input and target sequences, they are each converted into a single tensor using the stack function from PyTorch. These tensors are then converted to a TensorDataset, where each element is a tuple consisting of the input sequence and corresponding training sequence. A dataloader is then created from this dataset, allowing for batch processing during training. This dataloader also provides the ability to shuffle the data after each epoch. This prevents the model from learning patterns based on the order of the sequences.

The model was set up with three LSTM layers, followed by a single linear layer which is very similar to a Keras dense layer. The choice to use three LSTM layers was made since this is what Oore et al. found most effective in [11]. The first step was to define the class for the model's structure which inherits from the PyTorch class nn.module. This class contains an init function where layers of the model are established. The class also contains a forward function that handles the initiation of a forward pass which returns the output from the fully connected layer. It was decided to have 430 cells at each layer. The model created by Oore et al. used 512 cells, but the event vocabulary was longer since it included velocity. It is also mentioned in [11] that the model was not sensitive to changes made to this number which was also true for this model. The function used to calculate the loss was also changed in this approach. It was decided to switch to using Cross entropy after reading a paper written by Sebastian Garcia-Valencia. In [14], Garcia-Valencia finds that the cross entropy function is an optimal choice for a music generation model, showing that the use of this function reduced the randomness of outputs when compared to the use of the mean squared error function [14].

The training is done using a for loop where each iteration of this loop represents in epoch. In this loop, a nested for loop iterates over each batch of inputs and targets in the data loader. The inputs are converted into one-hot vectors using the one_hot function from PyTorch. These vectors are then passed to the model which initiates a forward pass. The outputs from this forward pass are then passed to the cross entropy function, provided by PyTorch, along with the targets. The loss is calculated then a backward pass is initiated using the adam optimizer algorithm. This process is repeated for each batch until all of the sequences from the training data have been used,

which concludes an epoch. At the conclusion of the training process, the model is saved as a pt file.
To generate new MIDI files, a random integer from 37 to 100 is generated as a seed. This range of integers represents the indices possible note on events for the generated sequence to start with, leaving out the first 36 and last 28 possible note on events to avoid stariting with a note that is too high or too low. This generated seed is checked for uniqueness. Since multiple files are generated at a time, having each file begin with a different note on event ensures more variation. The generated seed is then added to a list, converted to a tensor, then one-hot encoded. This one-hot vector is fed to the model which takes it as an input then produces an output based on the patterns it learned during the training process. This output is then added to the seed list. This process gets repeated 220 times to produce a list of indices representing a MIDI file. These indices are then converted to a MIDI file and saved to Google Drive using the audiate and list_parser funcitons provided by Gomatam in [13]. This process can be repeated a specified number of times to produce any number of MIDI files.

1.6 Second Approach Results

This model was trained for 150 epochs, resulting in a loss of 0.0456. The MIDI files generated showed an extreme improvement from the first approach. The model was able to pick up on patterns from the training data allowing it to produce melodies that made sense. There was improvement in both the pitch of the notes as well as the timing. Figure 9 shows an example of a MIDI file generated using the second approach.

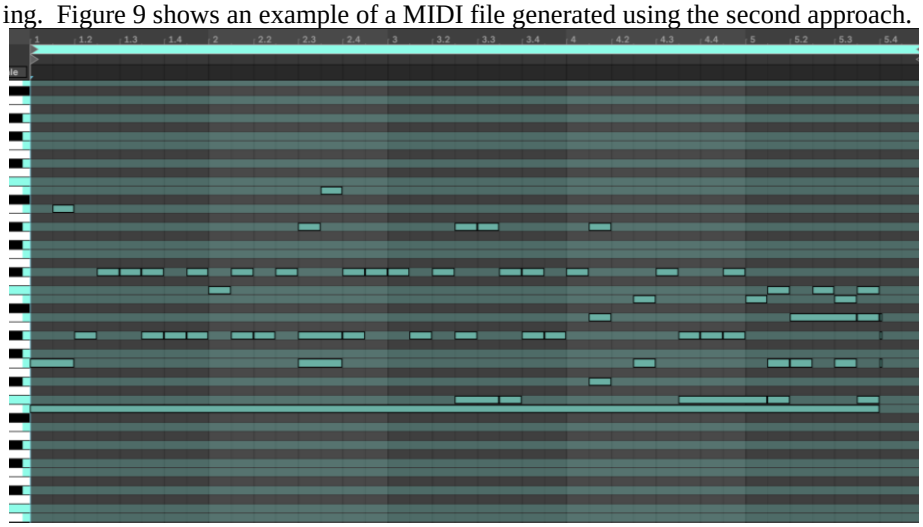


Figure 9. Midi file generated by LSTM model using the second approach. G# Major scale is highlighted to show the notes follow a key signature

The notes all fall within the G# Major scale, displaying how the LSTM model was able to pickup on the pattern of key signatures when predicting the pitch of notes. The notes allso all either start or end on a beat (with the exception of the three notes on the end), showing that the model was able to pickup on rhythmic patterns as well. This

output does have some noticeable flaws. The pattern of repetition, shown in Figure 7b, was not reflected by this MIDI file. The file also ends abruptly, with three very short notes at the end. Thie file still contained certain melodic elements that were useful for creating a hip hop instrumental drawing influence from rock, pop, and electronic dance music. This was one of a few MIDI files sent to a producer who goes by the name Atrxyu. He was able to create an instrumental using some of the notes from this MIDI file, which can be listened to on SoundCloud by following this link:

https://on.soundcloud.com/r3zRD

When asked for feedback on the generated MIDI files, Atrxyu said:

"So overall I think the midis are really cool and show a lot of promise. Some of the time it seems to get stuck on notes which needs to be ironed out...I think it'll be really useful for more skilled or talented musicians because they'll be able to take bits and pieces of the midis and build on them easily. I took one of them and just used it as a base and made an entirely new melody from it."

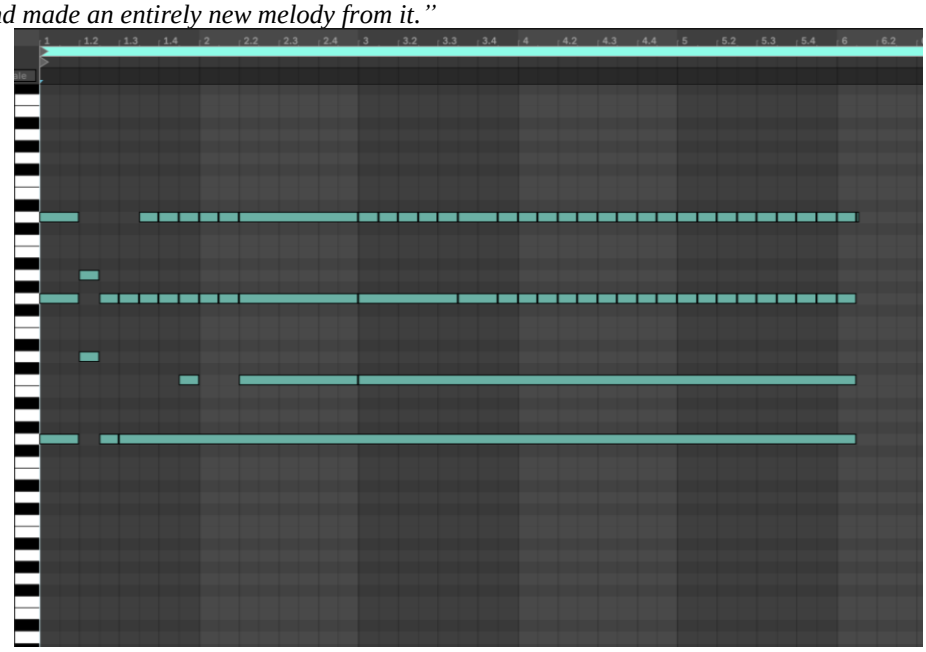


Figure 10. Midi file generated by LSTM model using the second approach.

Figure 10 shows another generated MIDI file that was sent to Atrxyu. It can clearly be seen that the model got stuck on generating the same three notes for the majority of the file. Even with the ability to rearrange notes, it would still be difficult for the producer to make anything from this MIDI file since it has almost no variation.

1.7 Difficulties Faced

During the first approach, there were a lot of issues using the pretty_midi library to extract data from the MIDI files. Some of the files were note compatible so they needed to be removed. It was also difficult figuring out a way to keep input sequences from containing notes from two different MIDI files. This was solved by adding a filename to the original DataFrame containg the data from all the MIDI files, then grouping by filename to created a dictionary of separate DataFrames for each MIDI file.

During the second approach, figuring out how to train the model using the Oore et al. vocabulary was very difficult. Originally, the data was not being one-hot encoded so figuring out the input shape and output shapes was tough. This lack of one-hot encoding also lead to extrememly high loss values. The process of implementing the code from Gomatam's repository [13] also came with some difficulties. Removing the parts not needed led to some errors which were difficult to troubleshoot. It was also discovered that the preprocessing stage was scaling down the MIDI files from the training data set by a factor of 2.5. Figuring out how to fix this was difficult. It was discovered that this could be fixed by scaling the time shift values up by a factor of 5. Scaling up by 2.5 was the first idea, but this lead to some of the values becoming floats, rendering them useless as indices. There was also a lot of trouble figuring out how to implement the model using PyTorch which is a bit more complicated then Keras.

4. Conclusion

The aim of this project was to develop a machine learning model that generates MIDI files, suitable for use as the starting point for the creation of instrumentals in a desired style drawing influence from multiple genres. The LSTM model developed was able to pick up on elements from the training data and produce MIDI files reflecting elements of the sepcified genres. The changes made in the second approach allowed the model to much better pick up on these elements resulting in an improvement in the files produced. There is still much to be done to imporve this model. The elements of repetition and overall structure were still not shown in the MIDI files generated by this LSTM. This model was also only evaluated on one combination of musical genres.

For future work, changes could be made to the structure of the model or the representation of the data to try and improve the model's ability to learn theses patterns. A larger data set might have also led to improvements. The model could also be evaluated on training data sets with MIDI files created for differnet genres than the ones used for this project. It would be interesting to see if the model performs better or worse for certain genre combinations. It would also be interesting to see how the model would perform when trained with MIDI files of only one genre.

Acknowledgments. I would like to thank my advisor, Dr. Colqohoun, for providing me with guidance throughout this process. I would also like to acknowledge Aditya Gomatam who's repository provided the foundation for my second approach.

References

[1] Laden, B., Keefe, D. H.: "The Representation of Pitch in a Neural Net Model of Chord Classification." In Music and Connectionism, edited by P. M. Todd and E. D. Loy. MIT Press, Cambridge, MA, 1991.

[2] Dannenberg, R., Thom, B., Watson, D.: "A Machine Learning Approach to Musical Style Recognition." Proceedings of the International Computer Music Conference. ICMA, San Francisco, CA, 1997, pp. 344–347.

[3] Todd, P. M., Loy, E. D.: "Music and Connectionism." MIT Press, Cambridge, MA, 1991.

[4] Griffith, N., Todd, P.: "Musical Networks: Parallel Distributed Perception and Performance." MIT Press, Cambridge, MA, 1999.

[5] Machine Learning for Science Team. "Recurrent Neural Networks and Long Short-Term Memory." Machine Learning for Scientists, 2020, https://ml-lectures.org/docs/supervised_learning_w_NNs/ml_rnn.html. *Accessed* [13/09/2023].

[6] Fei, R., Yao, Q., Zhu, Y., Xu, Q., Li, A., Wu, H., Hu, B.: "Deep Learning Structure for Cross-Domain Sentiment Classification Based on Improved Cross Entropy and Weight." Scientific Programming, vol. 2020, Article ID 3810261, 2020, https://www.hindawi.com/journals/sp/2020/3810261/. *Accessed* [13/09/2023].

[7] Werbos, P. J.: "Backpropagation through time: what it does and how to do it," in Proceedings of the IEEE, vol. 78, no. 10, pp. 1550-1560, Oct. 1990, doi: 10.1109/5.58337.

[8] Noh, S.-H.: Analysis of Gradient Vanishing of RNNs and Performance Comparison. Information 2021, 12, 442. https://doi.org/10.3390/info12110442

[9] Hochreiter, S., and Schmidhuber, J. "Long Short-Term Memory." Neural Computation, vol. 9, no. 8, 1997, pp. 1735-1780. https://doi.org/10.1162/neco.1997.9.8.1735.

[10] Olah, C. "Understanding LSTM Networks." Colah's Blog, 27 August 2015, https://colah.github.io/posts/2015-08-Understanding-LSTMs/. *Accessed* [13/09/2023].

[11] Oore, S., Simon, I., Dieleman, S., Eck, D., Simonyan, K.: "This Time with Feeling: Learning Expressive Musical Performance." arXiv:1808.03715v1, 10 August 2018, https://arxiv.org/abs/1808.03715v1. *Accessed* [13/09/2023].

[12] Raffel, C., Ellis, D.: "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.

[13] Gomatam, A.: "muisc-transformer" GitHub, 2023, https://github.com/spectraldoy/music-transformer. *Accessed* [13/09/2023].

[14] Garcia-Valencia, S.: "Cross entropy as objective function for music generative models." Computer Science Department, Universidad EAFIT, Medellin, Colombia; Research and Development Department, AIVA Technologies, Luxembourg City, Luxembourg, 4 June 2020, https://arxiv.org/pdf/2006.02217.pdf. *Accessed* [13/09/2023].