RoboMozart: Generating music using LSTM networks trained per-tick on a MIDI collection with short music segments as input.

Joseph Weel
10321624

Bachelor thesis
Credits: 18 EC

Bachelor Opleiding Kunstmatige Intelligentie
University of Amsterdam
Faculty of Science
Science Park 904
1098 XH Amsterdam

Supervisor
Dr. E. Gavves
QUVA Lab
Faculty of Science
University of Amsterdam
Science Park 904
1098 XH Amsterdam

June 26th, 2016
Abstract

Many studies have gone into developing methods for automated music composition, but few have used neural networks. This thesis used a Long Short-Term Memory (LSTM) network and, through prediction of what musical pitches are most probable to follow a segment of input music, generated new music. The network was trained on a dataset of 74 Led Zeppelin songs in MIDI format. All MIDI files were converted into 2-dimensional arrays which mapped to musical pitch and MIDI tick. The content of the arrays was sequentially selected in batches for training, and four different methods of selection were explored, including a method where periods of silence in songs were removed. Four input songs were used as input from which music was generated, and the musical structure in the generated music was analyzed. This, in combination with a survey where participants were asked to listen to some samples of the generated music and rate it by pleasantness, showed that the method where silence was removed from the dataset for training was the most successful in generating music. The network struggled to learn how to transition between musical structures, and some methods are proposed to improve the results in future research, including significantly increasing the size of the dataset.
## Contents

1 Introduction .................................................. 4  
1.1 Prior Research ........................................... 4  
1.2 Scope ...................................................... 5  
2 Method .......................................................... 6  
2.1 Dataset ...................................................... 6  
2.2 MIDI Tick Array ........................................... 6  
  2.2.1 MIDI Format ........................................... 6  
  2.2.2 Encoding ............................................... 7  
  2.2.3 Decoding ............................................... 8  
2.3 LSTM Network .............................................. 9  
  2.3.1 Background ............................................ 9  
  2.3.2 Implementation ...................................... 12  
2.4 Variants .................................................... 13  
2.5 Prediction .................................................. 13  
2.6 Post-processing .......................................... 14  
3 Results ......................................................... 15  
  3.1 Standard Batch Selection ............................... 15  
  3.2 Removing Zero Vectors ................................ 17  
  3.3 Larger Sequence Sizes .................................. 20  
  3.4 Random Batch Selection ............................... 22  
  3.5 Evaluation ................................................. 24  
  3.6 Survey ..................................................... 25  
4 Discussion ...................................................... 26  
5 Conclusion .................................................... 27
1 Introduction

Music composition has been a human hobby since ancient times, dating back to the Seikilos epitaph of Ancient Greece, 200 BC\(^1\). In more recent history, people have sought ways to automate the process of music composition, the earliest study on this having been published in 1960 [1]. The act of composing music may be justified by the notion that people listen to music for entertainment and to regulate their mood [2]. Composing music is a time-intensive task, making automation valuable.

There are many approaches to automated music composition, including using grammar models with language processing tools [3], or stochastic methods [4], with varying levels of success. With the recent success of deep learning in many fields of science [5], a deep neural network-based approach to automated music composition may be warranted. For this reason, this thesis describes an attempt to automatically compose (generate) music using deep learning.

1.1 Prior Research

There is little prior research into music generation using deep learning.

Chen and Mikkulainen (2001) sought to find a neural network that could be used to find structure in music [6]. To find it, an evolutionary algorithm was used, with the goal of maximizing the probability of good predictions. Tonality and rhythm shaped the evolutionary algorithm. The network that was found could generate melodies that adhered to a correct structure. However, the music was rather simple, and the system could not work with multiple instruments.

Eck and Schmidhuber (2002) explored the problems that most recurrent neural networks have had with music generation in the past, and justified using Long Short-Term (LSTM) Networks in music generation by finding that these networks have been successful in other fields [7]. They then used LSTM networks to generate blues music successfully, with input music represented as chords. Emphasis was placed on how the network generated music that correctly adhered to the correct structure and rhythm of the used blues music.

Franklin (2006) examined the importance of relying on past events for music generation [8]. This justified the use of recurrent neural networks. The goal of this study was not specifically music generation, but instead music reproduction and computation. By using LSTMs, music was reproduced, and with reharmonization, new music was successfully generated. This involved substituting learned chords with other learned chords (that fit the overall structure), which led to newly generated music.

Sak, Senior and Beaufays (2014) explored speech recognition using recurrent neural networks [9]. The implementation was very successful at speech recognition. This is relevant because this provides a method for adapting a network to work with raw audio, as opposed to converting text-based representations of audio in the other mentioned studies.

Johnston (2016) used LSTM recurrent networks to generate music by training on a collection of music (in the form of text in ABC notation), and taking all previous characters (in the music files) as input on which to base a prediction of the next character [10]. By doing this continuously, new songs were generated, with each new character being fed back into the recurrent network. Different types of architecture were tested, but music could be generated. However, the method was only successful with very simple songs, as more complicated, polyphonic songs can not be notated in the correct format that can be interpreted by the neural network. While some time was also spent looking at an implementation using raw audio instead of files with ABC notation, this was not successful.

By building on the successful parts of all the simple implementations for neural network-based music generation provided by these five studies, a solid foundation for the implementation of a proposed LSTM network for music generation may be created. However, this thesis also aims to have its music generation be based on segments of input music. It will also not be using a grammar-based model, as MIDI files will be used instead. The most similar study worked only with monophonic music, whereas the system used in this thesis should allow for polyphonic music. It may also serve as another example of what is possible in the quickly growing field of deep machine learning, and provide a foundation for future work in this area.

1.2 Scope

The goal of this thesis is to create a program that can automatically generate music based on a few seconds of a melody. It should have trained a neural network on a collection of other music files. Based on the input melody, it should predict what musical pitches are most likely to continue the melody, using the weights the network learned from the music collection. Appending the prediction and the input melody then forms an input for the next prediction. By repeating these steps, music may be generated.

Of note is the way in which music is encoded. Most types of audio files have their content encoded in a system that describes periods of musical pitch triggered at specific timestamps. While with a large enough dataset a network may be able to learn how to work within this system, it is worthwhile to convert it into a system that is much easier to learn, as this may improve accuracy and greatly reduce time spent training. For example, as was mentioned earlier, many approaches to music generation define grammars in which to encode the music used in their studies. Most of the studies described in the literature review did this as well. The music used in this thesis is encoded in MIDI format, which because of its relative representation is difficult to learn. However, an algorithm may be written that should convert the content of the files into a system that the network can learn more easily and reproduce more accurately.

The MIDI file format will be properly described in Section 2.2, but what is most relevant is that it encodes music in such a way that an interpreter knows when to play which pitch based on a relative relation between encoded pitches, with time represented in ticks. The algorithm that transforms these files into a system (specifically, an array) that can be entered into a neural network must rewrite this relative representation of time into an absolute representation, with each time period being one tick.

The neural network that is used in this thesis is a Long Short-Term Memory (LSTM) network, a special form of a recurrent neural network. This will be thoroughly explained in Section 2.3. This type of neural network is used because it allows a sequential structure of input, output, and any computational nodes in-between. This is important because music is sequential: certain musical pitches follow other musical pitches, and this generally happens in sequential patterns (for example, hooks and choruses) as well.

The dataset of files used for this thesis is the topic of Section 2.1. After explaining the MIDI encoding system and the neural network implementation, some variations are outlined in Section 2.4, followed by the process of predicting new music in Section 2.5, and post-processing in Section 2.6. The results of the thesis are evaluated in Section 3, after which there will be a discussion of the thesis in Section 4 and a conclusion is given in Section 5.

With all this in mind, this research topic follows:

Generating music using LSTM networks trained per-tick on a MIDI collection with short music segments as input.
It is hypothesized that with a properly encoded system, it should be possible to generate polyphonic music from any MIDI file using LSTM-based machine learning, and that different ways of handling the converted dataset will affect the quality of the generated music.

2 Method

2.1 Dataset

For this thesis, a collection of 74 Led Zeppelin songs was used to create the dataset. Every song is encoded in a type 1 MIDI file (discussed in the next subsection). The files were obtained from zeppelinmidi.com, which provides instructions on how to download the files. Of these files, only the vocal tracks were processed, because vocal tracks generally encompass expressive parts of songs. This was chosen because this may lead to more expressive features in machine learning, which may make music easier to learn and reproduce. The machine learning may also be aided by having chosen to only use one band, as this may lead to less variance than when using different artists and bands.

After training is completed, the prediction is made using the beginning of 4 different files (see Section 3). The first of these was created by hand and only contains 4 notes. The second file is the trance song Let the Light Shine In (Arty Remix) by Darren Tate vs. Jono Grant, which was chosen because of its simple repetitive structure. Next was a more complicated song, the jazz track Stella by Starlight by Washington & Young. Finally, Stairway to Heaven, one of the songs in the dataset, was used. The content of files is shown in the appendix.

2.2 MIDI Tick Array

2.2.1 MIDI Format

The MIDI file format encodes songs by having a collection of events which describe what kind of audio is playing. These events are all relative to each other, separated by ticks, which are the measure of time for MIDI files (this is further influenced by the MIDI resolution, which signifies the number of pulses per quarter note, and the tempo, which is the beats per minute). This format cannot be entered into the deep learning framework that is used for this thesis, so it must be represented in a different way. For this reason, an algorithm was created that converts this format into a two-dimensional array, where one axis maps every tick of the song (absolutely instead of relatively), and the other maps the pitch that is being played. The points in this array correspond to how loud a pitch is being played (the velocity).

There are many different types of events in MIDI files, including meta events which contain information about the song (e.g. the name). The meta information is not relevant for this thesis, which is why only the events that handle actual audio are processed: NoteEvents, ControlChangeEvent, PitchWheelEvents, SysExEvents, ProgramChangeEvent and EndOfTrackEvents. The number of ticks asserted by every event of these types is added together in order to obtain the appropriate size for the array.

NoteEvents are events that send the command to play music for a specified pitch at a specified velocity (NoteOn) or send the command to stop playing on that pitch (NoteOff). The other events are used to manipulate the way the audio issued by events that follow it will sound (for example, to have events sound like a different instrument). The array is created for only one instrument, which means that changes made by these other events are ignored. Still, their number of ticks is stored, in order to maintain correct temporal structure.
Note that this also forces all audio to be played by the same instrument. The MIDI format knows three different types, which define how to format handles parallel tracks. A track is a sequence of events, and with type 1 MIDI files, every instrument used in a song will have its own track. Type 0 MIDI files have every instrument on one track, handled through extensive use of non-NoteEvents. Type 2 files are rarely used, and use a system of appended tracks. Since all files in the dataset are MIDI type 1, it does not matter that all audio is forced to one instrument, because only a single track (with one instrument) is processed, which is already one instrument. The encoding system can also be used on type 0 files (not in the dataset), which creates a somewhat more cluttered array than when used on type 1 files.

### 2.2.2 Encoding

Since there are 128 different pitches that MIDI files can play, the columns of the array correspond to vectors, where each element contains the velocity of the corresponding pitch. Every tick in the MIDI file has its own vector, and these vectors create the 128 by (total number of ticks) matrix. The algorithm works as follows (in Python-esque pseudocode):

```python
total_ticks = 0
musical_events = []
track = MIDI_file.getBestTrack
for event in track:
    if isMusicalEvent(event):
        total_ticks += event.num_ticks
        musical_event.append(event)
grid = matrix(128, total_ticks)
current_vector = matrix(128, 1)
position_in_grid = 0
for event in musical_events:
    if isNonNoteEvent(event):
        position_in_grid += event.num_ticks
    else:
        if event.num_ticks != 0:
            for i in event.num_ticks:
                grid[:, position_in_grid] = current_vector
                position_in_grid ++
        if isNoteOffEvent(event):
            current_vector[event.pitch] = 0
        if isNoteOnEvent(event):
            current_vector[event.pitch] = event.velocity
```

The Python-Midi toolkit is used to easily access the content of the MIDI files. The track that is selected (pseudocode `getBestTrack`) is the first track that has NoteEvents. It would also be possible to take the track that has the most events (most activity), which often corresponds to a drum track. For this thesis, the vocal tracks were used, which in the dataset corresponded to the first tracks with NoteEvents.

In order to create the vector that contains the velocities for all 128 pitches, and because events contain information for only one pitch, and are ordered relatively to each other, the number of ticks contained in the event signifies whether to put consecutive per-tick vectors in the matrix.

---

Vectors are added when the number of ticks in an event does not equal zero, because this means that this event (relative to the previous event) is triggered later, so vectors are copied for this duration. Once vector placement is handled, the vectors can be changed for the next event. A velocity of 0 means that sound that was playing previously will no longer play. This is usually handled by a NoteOffEvent, but it is also possible to do this by having a NoteOnEvent with 0 velocity. The final line in the pseudo-code simply places the velocity of an event’s specified pitch in the to-be-placed vector, covering both cases. See Figures 1 and 2 for an example.

This algorithm encodes the MIDI file into a two dimensional array. This array can then be fed into the LSTM network described in Section 2.3.2.

2.2.3 Decoding

The array can also be decoded back into a MIDI file. The algorithm for this process is used after predictions are made, and converts a matrix of predicted ticks into a MIDI file, so that it can be played back. This prediction process is described in Section 2.5. The algorithm for decoding the arrays is (once again in Python-esque pseudocode) shown below:

```python
track = newMIDITrack()
previous_vector = grid[0] # first vector
for note_index in previous_vector:
    if note_velocity != 0:
        track.append(NoteOnEvent(0, previousvector[note_index], note_index))
tickoffset = 0
for vector in grid:
    if previousvector == vector:
        tickoffset ++
    else:
        for note_index in previous_vector:
            if previousvector[note_index] == vector[note_index]:
```
continue
if previousvector[note_index] != 0 and vector[note_index] == 0:
    track.append(NoteOffEvent(tickoffset, note_index))
else:
    track.append(NoteOnEvent(0, vector[note_index], note_index))
tickoffset = 0
tickoffset ++
previousvector = vector

The decoding algorithm starts by taking the content of the first vector in the array and triggering all corresponding events (for all non-zero elements). Afterwards, using a tickoffset, all vectors are iterated and compared. When consecutive vectors are identical, the tickoffset simply increases. Otherwise, the content is compared. Any elements that are not equal will trigger another event. This is either a NoteOffEvent if a pitch in the previous vector was non-zero and in the current vector is zero, or a NoteOnEvent otherwise. The tickoffset must be reset after any possible change within one vector, in order to maintain a relative relation between events.

2.3 LSTM Network

2.3.1 Background

A recurrent neural network (RNN) is a type of network that uses memory, as it learns to encode the transitions in temporal, sequential data. This is done by having nodes combine information from previous time steps with information in their current input. However, this type of network struggles with learning long-term dependency [11], meaning that relations learned relatively long ago tend to have low weights, as the weights decrease over time. This problem is solved in Long Short-Term Memory (LSTM) networks.

![Figure 3: A visualization of a module in an average LSTM. The cell state is the top horizontal line. It receives information from the second state, the line through the bottom. Both states received input from the previous module, but the second state receives additional input (X_t) and provides temporary output (h_t). The image is from a 2015 blog by C. Olah\(^3\).](http://colah.github.io/posts/2015-08-Understanding-LSTMs/)

Like regular recurrent neural networks, LSTM networks are built from neural network chains. The difference is that in regular RNNs, the networks are built from a simple structure (such as a module containing one layer that performs a computation between nodes), while in LSTMs the modules are built around a cell state, which is one of two managed states in the module (see Figure 3). The cell state contains the memory of the RNN. It is created by forwarding the output of the previous module, and receives information from the second state before forwarding itself to the cell state of the next module. The second state takes input (this is the input given to regular neural networks) and performs specific computations that determine what part of the new input is fed into the cell state (the memory), and to determine what part is then returned as momentary output. The exact computations may differ between LSTM implementations, but the standard framework is as follows:

- A calculation determines what information in the cell state should be forgotten, so that new information can take its place (this could for example be determined by calculating variance). This is the forget gate. In a standard LSTM network, this can be calculated with:

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

Where $\sigma$ refers to a sigmoid function. $f$ determines what is being forgotten. Since a sigmoid is used, this value is between 0 and 1, and this range corresponds to how much is forgotten: the smaller the value, the more is forgotten (removed from the cell state). $W$ is the corresponding weight of the neural network node. $x$ and $h$ are inputs, $x$ corresponding to new input and $h$ being forwarded from the previous module. $b$ is a constant.

Figure 4: This illustration depicts the forget gate.

- A combination of determining what information in the cell state will be updated, and determining what information (from the input) then overwrites it in the cell state. This is the input gate. The result of these combinations are then added (mathematical addition) on to the new cell state. This addition solves the problem of vanishing gradients which occurs in different types of (recurrent) neural networks, as instead of possibly multiplying very small numbers (leading too smaller numbers, and thus the vanishing gradient), they are added. In a standard LSTM network, the calculations are:

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

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$$

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

$$h_t = O_t \cdot \tanh(C_t)$$

Figure 5: This illustration depicts the input gate.
This determines which part is updated. Here $i$ is the information that is being input to replace what was previously forgotten, $C$ is the new cell state. $\text{tanh}$ can be used to get a value between $-1$ and $1$, if the network is built for such a range. With these values, updating the cell state follows:

$$C_t = f_t \ast C_{t-1} + i_t \ast \tilde{C}_t$$

The $f$ calculated previously is multiplied by the previous cell state, and then added to a new cell state $i \ast \tilde{C}$.

- A combination of calculating what part from the second process is outputted and what part of the cell state. These calculations are multiplied, and the result is sent to the output, as well as to the second process of the next module. This is the output gate. In a standard LSTM network, this is calculated as:

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

$$h_t = o_t \ast \text{tanh}(C_t)$$

Here $o$ is the proposed output, which when multiplied by the cell state creates a state which can be sent to the output node of the network, and is also sent to the next module (where the entire process starts again).
learned. By having the output gate last, modules output what was learned during their own cycle. What has been described above is the standard LSTM network, and the one that is used in this thesis. It is implemented in Python using the Keras\(^4\) and Theano [13] frameworks.

### 2.3.2 Implementation

The network was made up of 3 layers, as shown in Figure 7. The input was entered into an LSTM layer with 512 nodes. Next, dropout regularization was used in order to reduce overfitting [14], after which there was another LSTM layer with 512 nodes. A Dense layer then lowered the number of nodes to 128 (corresponding to 128 possible pitches), which was sent to the output. This structure was determined through experimentation. Mean squared error was used as loss function for training, with linear activation. RMSProp optimization [12] was used to speed up the training of the network.

Figure 7: A visualization of the layers of the neural network. The first LSTM layer with 512 nodes is given 2-dimensional arrays created from encoded MIDI files. Dropout regularization is performed afterwards, followed by another LSTM layer with 512 nodes. A dense layer then connects these nodes into 128x1 vectors, for output.

The network trains on a collection of MIDI files. Each of these files is converted into a matrix as described in the previous section. The content of these matrices is then copied into sequences of input vectors each with a corresponding label vector. The length of these sequences should be sufficiently large so that the input vectors encompass different pitches (ideally, a change in melody), but not so large that it fails to find relation between input and label (leading to high loss in the loss function). The label vector is the vector that is one column further than the last of the sequence vectors in the matrix.

Figure 8: A visualization of selecting the input and label vectors from a converted MIDI matrix. In red, a sequence of vectors are selected for input (with sequence length 9). In blue, the label vector is selected.

Although specific sequence sizes were determined through experimentation, it is worthwhile to use multitudes of 12, because (as was mentioned earlier) the ticks in MIDI format correspond through MIDI resolution to number of pulses per quarter note, and 12 can be divided by both 4 and 6, which allow different musical tempos to be incorporated. However, the music in the dataset could be incorporated using multitudes of 4, which was why all sequences sizes used were multitudes of 4.

After selecting one sequence of input and label vectors, the selection moves a specified step size of columns to the right and then selects the next sequence of input vectors and the corresponding label vector. These sequences and vectors are added to two lists. Once all files in the dataset have been processed, batches of the sequences and label vectors are entered into the neural network. Doing this in batches is necessary because of hardware limitation, as processing too many sequences at once will quickly overflow CPU RAM. The network trained 50 epochs on each batch. Training and predicting was done on the Distributed ASCI Supercomputer 4 (DAS-4)\(^5\).

### 2.4 Variants

Four different variations for handling the previously described batches were implemented, in order to gauge how these would affect the quality of the generated music.

- **Regular/standard batch selection**: Batches were selected consecutively from the encoded music arrays. The size of each batch/sequence was 64 vectors. This means that all possible sequences of 64 vectors (plus one label vector for each sequence) in a song are learned by the neural network.

- **Removing zero vectors**: Music sometimes involves periods of silence, and this is particularly common in vocal tracks (including the vocal tracks from the dataset). In an encoded array, these periods are represented by consecutive zero vectors. While learning silent periods (and the transition to and from these) may be useful (especially with significantly larger datasets), it may also lead to the network favoring the predicting of long periods of silence because of how similar the sequence and corresponding label vector will be. For this reason, a variation on the handling of batches for training is to remove all zero vectors from the encoded array.

- **Larger sequence sizes**: Different sizes for the sizes of the selected batches/sequences were implemented (mainly because of an issue with looping predictions that will be described in Section 3). Sequence sizes of 96 (medium) and 160 (large) were used, in an attempt to lower overfitting and learn larger musical structures.

- **Random batch selection**: Instead of using a step size between consecutive batches of vectors, this variation on handling batch selection selected 1000 random batches from an encoded array, for every musical file in the dataset, in another attempt to reduce possible overfitting, and perhaps bolster the creativity of the network. With a sequence size of 64, most files in the dataset contained between 3 and 5 thousand possible batches, meaning that the network trained significantly less than when using regular batch selection. Thus, another variation took 3000 random batches instead of 1000.

### 2.5 Prediction

Once training is complete, the prediction process can begin. This is based on a small segment of an input song, in the same per-tick array format that was used for the songs during training.

---

The same sequence length used then is used now to determine the size of the segment. With this as a matrix and the learned weights, a new vector is predicted, corresponding to the next tick of music. The values of these vectors should be those that the network deemed most likely to follow the segment (determined by the loss function during training).

After a vector has been predicted, every vector in the array that corresponded to the input segment is pushed one position (column-wise) to the left, and its first vector is removed. The predicted vector then becomes the last vector (furthest to the right) in the array. With this array, another vector is then predicted, and the array is adjusted again. By repeating this process, sequential music may be generated.

Every predicted vector is stored in a separate list so that the entire prediction, as an array, can then be converted into a MIDI file. The values inside the vectors may need to be clipped, to not be lower than 0 or higher than 127, due to the nature of the linear-based neural network.

2.6 Post-processing

After predicting has finished, some post-processing may be necessary, in order to smooth predicted vectors so that minor differences in pitches are replaced by consecutive same pitches, which sounds better. This is done by looking at every velocity for every vector in the array of predictions. For every possible pitch, the highest occurring velocity over the entire array is found and stored.

Next, the velocity values of every pitch for every vector is compared to the highest occurring velocity on that pitch. If the difference between the two velocities is higher than 10% of the highest occurring velocity on that pitch, the velocity for that pitch in the corresponding vector is set to 0. Otherwise, the velocity is set to 100. This causes all velocity values in the eventual file to be 100, which is a velocity that is used commonly in MIDI files (most of the files in the dataset used in this thesis only contain velocity values of 100).

Using this percentage-based comparison is necessary to capture pitches that were deemed less likely during prediction but still somewhat important to the melody (e.g. a supporting melody
on the background). Using a constant comparison threshold would fail to capture these, or cause high velocity values to become single long pitches (due to small differences being seen as insignificant and thus being smoothed).

In addition to this method of smoothing, very low velocity values are removed altogether. These values correspond to the network determining that a pitch is extremely unlikely. If left in the array, it becomes soft background noise. By using the highest occurring velocity value for every pitch, it is easy to determine whether or not a pitch contains any relevant music. If the highest occurring velocity is lower than 5, all velocity values on that pitch for every vector in the array are set to 0.

![Figure 10: A visualization of a predicted song before and after post-processing.](image)

It is worth noting that with more training and especially with a larger dataset, the amount of post-processing required may be greatly reduced.

## 3 Results

Evaluation is difficult for this thesis, as music is subjective. Nevertheless, the predicted music may be analyzed and apparent structure through common occurrences may be inferred, which will be handled first: The results of four variants of batch selection are examined, and this is followed by an analysis.

Afterwards, an empiric study in the form of a survey is processed, with the goal to objectively analyze the subjective music.

### 3.1 Standard Batch Selection

With standard batch selection, structured music appeared to be generated with the example.mid file as input (see Figure 11). The generation starts out with a short period of disorganized noise, which also happens when using jono.mid and stella.mid as input (Figure 12). These two files however fail to generate anything other than silence with extremely low velocity values which are filtered out during post-processing. With stair.mid (Figure 13), the generation resembles the beginning of Stairway to Heaven for approximately 288 ticks, until it becomes disorganized noise. While it eventually breaks out of this and begins a structured melody, this does not resemble the original song.
Note that in the following figures, the input music (in the first 64 ticks) may not completely resemble the content of the input music as can be found in the Appendix. This is because during post-processing, some parts of the input may have been filtered out. This does not mean that it was not completely used during generation, however, as post-processing occurs after the prediction process has finished.

Figure 11: A visualization of the predicted music using regular batch selection with example.mid as input. The segments separated by grey vertical lines contain 32 ticks. The input music is shown in the first 64 ticks. The generated music (which begins on tick 64) starts out noisy and disorganized, before settling into a melody on tick 135.

Figure 12: Visualization using regular batch selection with jono.mid (left) and stella.mid (right) as input. Both contain only a short sequence of disorganized noise, and failed to generate further.
Figure 13: A visualization of the predicted music using regular batch selection with stair.mid as input. The image is zoomed out in order to show more of the song. After the 64 input ticks, the generated melody resembles Stairway to Heaven for approximately 288 ticks, after which there is a period of disorganized noise, followed by a repetitive melody that is not part of the original song.

3.2 Removing Zero Vectors

This variant used the same sequence size of 64 ticks (for better comparison) as regular batch selection, but all zero vectors were removed during training. What was hypothesized was that the network would no longer end up in periods of silence where it failed to generate anything, and this appeared to correct. Because the predicted songs contained more variance than the ones shown previously, the following figures are all significantly zoomed out to show many ticks. Surprisingly, the network appeared to have more successfully learned how to transition from one pattern to another with this variant of batch selection than when it used regular batch selection. The same segments of disorganized noise found in regular batch selection are found here, however.
Figure 14: A visualization of the predicted music using zero vector removal with example.mid as input. After the input, there is a period of disorganized noise for approximately 32 ticks, followed by approximately 48 ticks of a melody, after which a different melody begins, which repeats continuously.

Figure 15: A visualization of the predicted music using zero vector removal with jono.mid as input. Following the disorganized noise, there is a long period of long tones with little variety. A short interval that appears afterwards is followed by a more melodious structure which repeats continuously.
Figure 16: A visualization of the predicted music using zero vector removal with stella.mid as input. While a lot is generated (once again after a very short period of disorganized noise), it is very repetitive and filled with outliers.

Figure 17: A visualization of the predicted music using zero vector removal with stair.mid as input. The prediction resembles Stairway to Heaven only in the very beginning for 96 ticks. A simple pattern follows, there is a very melodious segment for 160 ticks before transitioning back to the simple pattern. This melody is not part of the original song, however.
3.3 Larger Sequence Sizes

With larger sequence sizes, the network may be able to process longer musical structures and handle transitions better than with the regular 64 tick size. The input sequence in the following figures is larger as a result, with 3 segments separated by grey lines corresponding to 96 ticks for medium batch size, and 5 segments to 160 ticks for large batch size. The figures show the predictions did not improve, however. With medium batch size, the network could predict almost 100 ticks before getting stuck in silence or on one constant pitch. With large batch size, the network would either generate a very short segment of disorganized noise, or generate nothing at all.

![Figure 18: A visualization of using larger sequence sized batch selection with example.mid as input. On the left, sequence size 96 (medium). On the right, sequence size 160 (large). Some music was generated with medium sequence size, but nothing with large.](image-url)
Figure 19: A visualization of using larger sequence sized batch selection with jono.mid as input. On the left, sequence size 96 (medium). On the right, sequence size 160 (large). Medium sequence size generated some structure (with pitches being turned sporadically on and off). Large sequence size generated mostly noise, before stopping generation altogether.

Figure 20: A visualization of using larger sequence sized batch selection with stella.mid as input. On the left, sequence size 96 (medium). On the right, sequence size 160 (large). Again, little is generated, with medium sequence size getting stuck on one constant pitch, and large generating a few sporadic tones.
Figure 21: A visualization of using larger sequence sized batch selection with stair.mid as input. On the left, sequence size 96 (medium). On the right, sequence size 160 (large). With medium sequence size, the prediction slightly resembles the original song only in the very beginning. With large sequence size, only noise is generated.

3.4 Random Batch Selection

This variant of batch selection was attempted with 1000 randomly selected batches as well as 3000. Because the network failed to predict anything with many input songs for both of these attempts, only three of the results are shown in the following figures. Large batch size was used for all these, which means the input size is 160 ticks and corresponds to 5 segments separated by grey vertical lines.

Figure 22: A visualization of the predicted music using 1000 randomly selected batches with jono.mid as input. The prediction gets stuck on the same pitches and predicts these continuously.
Figure 23: A visualization of the predicted music using 3000 randomly selected batches with jono.mid as input. Interestingly, the network predicted a long period of silence but eventually transitioned back to non-silence.

Figure 24: A visualization of the predicted music using 1000 randomly selected batches with stella.mid as input. Towards the end of the long consecutive tone that starts on tick 172, the network predicted a melodious transition towards the next long consecutive tones.
3.5 Evaluation

With regular batch selection, both the generation that used jono.mid and the generation that used stella.mid were unable to generate much beyond the small segments of disorganized noise. This may have been because of the vocal tracks that were used for training. Vocal tracks contain long periods of silence, which may have led to the network ending up stuck in a sequence of continuing predicted silence, as its memory is not large enough to enclose transition into non-silence again. The batch selection variant where zero vectors are removed was created as a possible way to address this.

A possible explanation for the generation of disorganized noise (found in all of the variants of batch selection) is that the LSTM network failed to find a connection between its short memory of input ticks and what it had learned during training from the dataset, as the input music is either in a key that no music in the dataset is ever in, or is not in a key at all (in the case of example.mid, which is just a few random tones). Since the network does not recognize the key, it fails to find one likely pitch, and instead returns many unlikely ones. This is continued until a pattern emerges which the network can recognize, and a learned structure follows. This explanation is backed up by the fact that with regular batch selection, the network did not have trouble continuing Stairway to Heaven (until as the generation continued, it became harder and harder to further the original song), as it recognizes the key in the input music, which is in the dataset.

With regular batch selection, the generations eventually end up with a repeating structure after a period of disorganized noise. While this may imply that the network correctly learned musical structure, an infinite repetition of a very short melody is monotonous, and not something found in songs in the dataset. Ideally, the network should transition into different melodious structures. The reason this happens may be that certain structures that were learned during training fit exactly into the specified sequence size (batch): If a melodious structure occurs consecutively in a dataset, and one occurrence fits exactly within the sequence size, then at the end of having predicted the structure, the network may find it likely that this is the first occurrence, and that it should repeat it. Since the sequence size determines the memory of the network, it is incapable of remembering that it has already predicted the structure multiple times. Larger sequence sizes however failed to generate much music due to large segments of disorganized noise, so it is difficult to say with these results whether larger sequence sizes will help reduce repetition and inspire transition.

In general, with larger sequence sizes, specifically with a size of 160 ticks, the network had a much more difficult time generating music (it did not generate anything at all with the example.mid file). This may mean that finding the correct musical key gets more important the larger the sequence size is. It may also be possible that because the example file is not long enough to make up the entire 160 tick input segment and zero vectors had to be appended, that the network had too many zero vectors in its input, and generated only silence because of this. This does not explain the issues with the other files, however.

With random batch selection, the number of periods of disorganized noise was much smaller than with other variants of batch selection. However, in the few cases that the network did manage to predict music, the prediction would often get stuck on the same pitches. This may be because transitions are more difficult to learn when randomly selecting batches. While, one of the generated songs did contain a melodious transition of one structure to another, this may just be the result of having randomly found some structure with this transition that also occurred in other randomly selected batches during training. It is difficult to interpret the results of the random selection when it failed to predict music so often.

Many of the predicted songs (with all variants of batch selection) contained outliers. While
it may be possible to filter some of these out using different threshold values during post-processing, it should be noted that the pitches of these outliers were always still close to the pitches of whatever main pattern the predicted songs created. No pitch was ever further than 20 pitches away from the highest pitch found in its main pattern. This may not imply that the network had started learning how to work within musical key, but it did learn that extreme pitches are very unlikely in music, or at least in the type of music that fits the dataset.

From these results, removing zero vectors appears to have led to the best predictions. Musical structure appear melodious and there are transitions between patterns. However, music remains subjective, and while the generated music from zero vector removal may look structurally correct, it is still up to people to decide the subjective quality of the songs.

### 3.6 Survey

Ten participants were asked to listen to samples of 9 generated songs:

1. Regular batch selection with example.mid
2. Regular batch selection with stair.mid
3. Zero vector removal with example.mid
4. Zero vector removal with jono.mid
5. Zero vector removal with stella.mid
6. Zero vector removal with stair.mid
7. Medium sized batch selection with jono.mid
8. Medium sized batch selection with stair.mid
9. Random batch selection with stella.mid

For each sample, participants were asked to grade the pleasantness of the sample on a grade of 1 to 5, with 5 being most pleasant. The average grade of each sample was:

<table>
<thead>
<tr>
<th></th>
<th>example.mid</th>
<th>jono.mid</th>
<th>stella.mid</th>
<th>stair.mid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular</td>
<td>2.9</td>
<td></td>
<td></td>
<td>3.3</td>
</tr>
<tr>
<td>Zero Vector Removal</td>
<td>3.3</td>
<td>3.6</td>
<td>2.3</td>
<td>2.0</td>
</tr>
<tr>
<td>Medium Batch Size</td>
<td>2.3</td>
<td></td>
<td>2.4</td>
<td></td>
</tr>
<tr>
<td>Random Batch Selection</td>
<td></td>
<td></td>
<td></td>
<td>3.2</td>
</tr>
</tbody>
</table>

Zero vector removal received the highest individual average grade, while medium sized batch selection received the lowest grades averaged over its two samples. This is consistent with the examination of the structure of the predictions, where predictions using zero vector removal appeared the most melodious. The network failed to generate much music with larger batch size selection, and what it did generate was often very noisy, so this is also consistent.
4 Discussion

While the created format for representing MIDI files can be successfully implemented in the LSTM network, it can only handle one channel (MIDI channels usually represent different instruments). In order to handle multiple channels, the algorithm would either need to be rewritten to construct a higher dimensional grid (one dimension is added, with the different channels as its axis), or the values in the 2-dimensional grid that is used now (the velocity (loudness) of the notes) need to have a system where ranges of values map to different channels. A possible method for doing this would involve having each channel receive its 128 notes, where higher channels start with a value 128 higher than the previous channel. Then the channel that the value maps to can be found by dividing by 128, and the value by using the modulo operator with 128.

The MIDI encoding system cannot handle consecutive same-velocity and same-pitch tones. While such tones are rare (it is impossible for acoustic music, as no human can stop and start playing an instrument at the same time), it does occur in some electronic music. These tones, after being encoded, become one continuous tone. This is a limitation of the algorithm. A possible solution would involve pre-processing where MIDI files are checked for this behavior, and manually setting the sequence of velocity values one higher or lower than the previous sequence.

The Keras framework was used because it enables high level implementations of difficult concepts. This meant that more time could be spent on other parts of the thesis. However, the framework is stuck with a somewhat fixed system of relation between input and output (e.g. the sequence length must remain constant throughout). This could be circumvented by changing source code manually, which defeats the purpose of saving time because it would take a lot of time to understand. This fixed system meant that various experiments, such as experimenting with alternating sequence lengths, or having output that were longer than one vector (arrays as output), were not possible. Some RNN implementations rely on output sequences, which are not possible in Keras, but may be beneficial for the topic of this thesis and future work. Thus, future research should experiment with different frameworks that allow more sequential output.

Though different variants of batch selection were explored, they all ignored the use of a variable step size between batches. Random batch selection does not use a step size at all, and the other variants all used a step size of 1. Early on during experimentation a step size of 1 appeared to more successful than larger step sizes, which led to future experimenting with other step sizes becoming an afterthought. Nevertheless, it may be worthwhile to do more experimentation with larger step sizes on different datasets. In particular, this may be effective for songs with highly repetitive structure, as batches may skip over some of these parts. It may also help in lowering the likelihood of music generation getting stuck in loops, albeit at the risk of less learned structure. Also of note is that larger step sizes will reduce the total number of trained batches, which may greatly reduce the time the network has to spend training.

Post-processing proved useful in filtering out noise and in smoothing velocity values. However, in many predictions, some very short tones (usually of only one tick) still occurred sporadically throughout the entire predicted song. These tones were almost always off-key. While experimenting further with the current method of post-processing (increasing the minimum value threshold, and increasing the percentage of allowed difference between the highest and the processed velocity values) may result in better results, adding a new step to post-processing may also be beneficial. This new step would involve filtering out off-key velocity values. This would include determining the most recent key, and calculating whether predicted non-zero velocity values for pitches that had zero velocity values during the sequence are off-key compared to
the recent key. If they are, there can be another check to see if the vector could belong to a new sequence where the value would be in-key. If it fails both checks, the velocity can be set to 0. This will require a non-trivial function that determines whether a pitch is off-key or in-key given a sequence. However, there is some research into techniques for key detection [15]. For the second check, defining whether a vector belongs to a new sequence may simply involve checking further in the predicted array, as all this will be done in post-processing, which means that any vectors further into the array are already available.

Instead of doing the aforementioned during post-processing, it may also be interesting to make key detection a primary focus for a future automated music generation study. If a neural network can use the sequence of musical keys detected in a dataset of songs for training, it may be able to then randomly generate music in-key and then use the learned sequence transitions to transition various keys into melodies and songs. While this does make the research more similar to other studies that have worked with grammars that already encapsulate a sequence between which pitches must remain (essentially a key), doing this with MIDI files may still be innovative.

While the results of increasing sequence length and of using random batch selection to collect training sequences with corresponding label vectors were less than satisfactory, it is possible that increasing the size of the training set may significantly improve the results. With only 74 files in the dataset, it becomes more difficult to find structure in longer sequences. In fact, a larger dataset may improve the results of all methods of training. The dataset may simply be too small for the network to learn any generalizable structure. As explained in Section 2.1, the small dataset was chosen because of an expected similarity between songs when using only one band or artists. Nevertheless, experimenting with much larger datasets may be worthwhile.

Another variant for batch selection that was briefly tested during experimentation was the use of binary velocity values. All non-zero velocity values in an encoded array would be set to 1. The reasoning behind this at the time was that it may lower loss during training. This was scrapped when no immediate more successful predictions were made with this method than without it. However, it may still be interesting to look into this, as it may allow for a classification task instead of regression, where every pitch is a class. This would then allow more experimentation with different loss functions and activations in the neural network as well.

Also, the variations on batch selection may be combined. While combinations such as selecting random batches after removing zero vectors from an encoded array were not attempted for this thesis, they may provide interesting results. More experimentation with this may serve as a basis for future research.

For empirical evaluation with the survey, only a limited number of participants reviewed the selection of samples of predicted songs. Given the subjective nature of music and the problems with justifying generalizing results of small samples to larger populations, it may be worthwhile to repeat the survey or create new surveys with a larger number of participants.

5 Conclusion

It was hypothesized that with a dataset of songs encoded in MIDI format that are properly converted to a system that a recurrent neural network can use for training, it should be possible to generate polyphonic music from any MIDI file. Furthermore, the ways of handling the converted dataset may affect the quality of the generated music.

A dataset of 74 Led Zeppelin songs in MIDI format was encoded into arrays and used by an LSTM network with three layers for training. After training, the network could predict how
an input song should continue. There were four different methods of selecting data from the arrays, which affected how well the network could generate music. The most successful of these was where periods of silence in the songs in the dataset were removed. The structure of the generated songs with this method appeared the most melodious. This was consistent with a survey taken where the subjective quality of samples of generated music was better with this method than with the other methods. However, the generated music contains noise in the form of pitch outliers and the network struggles with repetitiveness, and this is reflected in the mediocre grades the generated music received in the survey. While it is definitely possible to generate music using LSTM networks trained per-tick on a MIDI collection with short music segments as input, as was the topic of this thesis, the algorithms should be improved upon before declaring that all musicians will soon be out of jobs.

References


Appendix

Input Files

Now follows a visualization for each of the four input files used for prediction.

Example File (example.mid)
The MIDI track in this file is only 90 ticks long, which meant that zero vectors had to be appended when using sequence sizes larger than 90 (for medium with sequence size 96 and large with sequence size 160).

Let The Light Shine In (jono.mid)
Since this is an actual song, the track goes on for quite a while and is much longer than shown in the image. However, it is essentially a repetition of what is shown here.
Stella by Starlight (stella.mid)
This is a non-repetitive jazz song with many different melodies. Only the first few ticks were used, however, as was described in this paper. Nevertheless, the first few ticks contain many different tones.

Stairway to Heaven (stair.mid)
This song was part of the dataset, and is another non-repetitive song. As was described, this leads to a messy continuation of the song.

Survey Song Samples
The following nine samples from generated songs were used in the survey.

Regular batch selection with example.mid (survey1.mid)
Zero vector removal with example.mid (survey2.mid)

Zero vector removal with jono.mid (survey3.mid)

Medium sequence size batch selection with jono.mid (survey4.mid)
Zero vector removal with stella.mid (survey5.mid)

Random batch selection with stella.mid (survey6.mid)

Regular batch selection with stair.mid (survey7.mid)
Zero vector removal with stair.mid (survey8.mid)

Medium sequence size batch selection with stair.mid (survey9.mid)