Spring 2018

VGM-RNN: Recurrent Neural Networks for Video Game Music Generation

Nicolas Mauthes
San Jose State University

SJSU SAN JOSÉ STATE UNIVERSITY

VGM-RNN: Recurrent Neural Networks for Video Game Music Generation

A project presented to the faculty of the

Department of Computer Science

in partial fulfillment of

the requirements for the degree of

Master of Science

by

Nicolas Mauthes

May 2018

Abstract

VGM-RNN: Recurrent Neural Networks for Video Game Music Generation

by Nicolas Mauthes

The recent explosion of interest in deep neural networks has affected and in some cases reinvigorated work in fields as diverse as natural language processing, image recognition, speech recognition and many more. For sequence learning tasks, recurrent neural networks and in particular LSTM-based networks have shown promising results. Recently there has been interest – for example in the research by Google's Magenta team – in applying so-called "language modeling" recurrent neural networks to musical tasks, including for the automatic generation of original music. In this work we demonstrate our own LSTM-based music language modeling recurrent network. We show that it is able to learn musical features from a MIDI dataset and generate output that is musically interesting while demonstrating features of melody, harmony and rhythm. We source our dataset from VGMusic.com, a collection of user-submitted MIDI transcriptions of video game songs, and attempt to generate output which emulates this kind of music.

Contents

Figures

1 Introduction

The recent success of machine learning in helping to solve difficult and previously intractable problems has led to a renewed interest in the field of artificial intelligence. After a period of stagnation during the 80s and 90s – known today as the “AI winter” – there has been a resurgence in the development of “learning” algorithms.

In particular, the improvement in computational power seen in the last decade along with the availability of vast amounts of data have made the implementation of previously neglected models such as deep neural networks feasible. Artificial intelligence research is no longer only the province of academic institutions and corporations. Today, there are many freely-available libraries and toolkits including TensorFlow, Keras, PyTorch and others that make machine learning accessible to the average computer scientist.

This has led to practical applications in natural language processing, bioinformatics, speech recognition, and still many more areas. However, it is one field in particular, which has only very recently come to prominence, that is most relevant to this work. That is the application of machine learning and in particular, deep neural networks, for creative purposes. The goal of this work is to demonstrate one such deep neural network for the modeling and automatic generation of music.

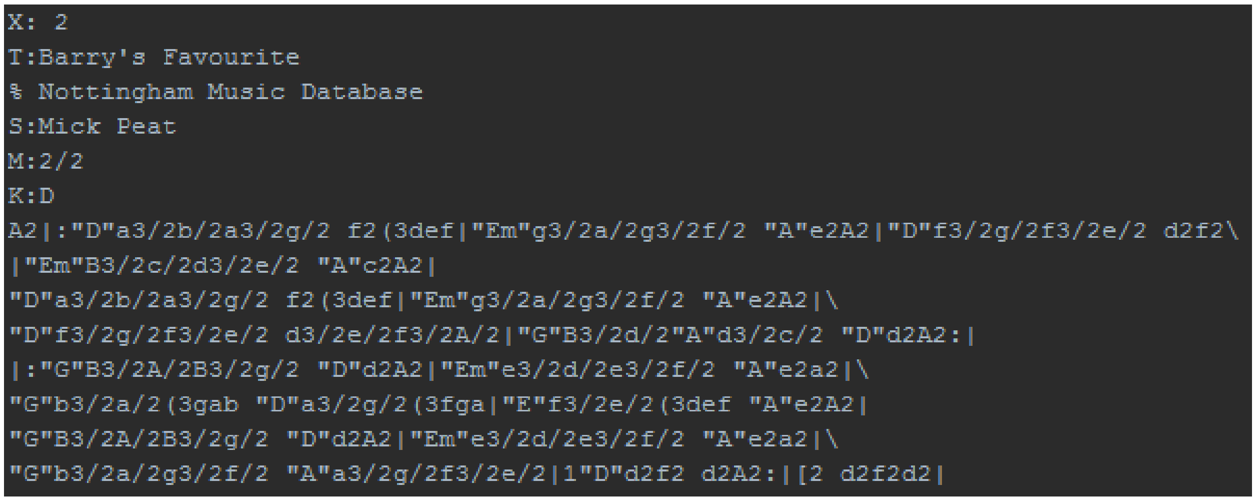
Attempts at modeling and creating music algorithmically have existed since at least the 18th century with the musical dice games of Mozart and Haydn, in which pre-composed sections of music were combined probabilistically. During the mid-20th century, composers like Cage and Xenakis experimented with chance procedures, albeit with very different approaches. The *stochastic music* of Xenakis applied ideas taken from group and set theory to the composition of musical scores. In 1957 Hiller and Isaacson used Markov chains and generative grammars to compose the *Illiac Suite*, one of the first pieces to be written using a computer.

More modern approaches to algorithmic composition include the use of rule-based expert systems, such as David Cope's *Experiments in Musical Intelligence* (EMI) from 1996, which was used to compose works in different historical styles. The use of Hidden Markov Models (HMMs) is today a common data-based approach to music modeling, used in e.g. Sony's *Flow Machines*. The advent of deep learning, however, has seen increased interest in the application of deep neural networks for modeling music and other creative endeavors.

Such efforts are the domain of the field known today as computational creativity. Though it only recently acquired its current moniker, the ideas at its core have been explored by computer scientists since the dawn of artificial intelligence. Computational creativity researchers attempt to answer questions such as: Can computers achieve human-level creativity? Can they create art that emulates that done by humans, or even something completely original? Can they create something that is indistinguishable from human-produced art? Note that the last question is a kind of artistic Turing test, and is a major concern of the field.

One group that is actively exploring these questions is Magenta, an offshoot of the Google Brain team. Magenta asks, "Can machines be creative?", and through their research in deep learning and language modeling attempt to answer that question. Magenta is perhaps one of the most well-known of the groups of researchers applying deep learning to music, with the backing of a major company.

Magenta's use of deep neural networks and in particular, LSTM-based recurrent neural networks, is in line with what has become a common approach to music language modeling. Interest in RNN models has largely replaced – but not completely eliminated – the interest in older modeling techniques, such as those based on Hidden Markov Models. They have shown promising results, especially for modeling polyphonic music (i.e. where more than one note can occur at the same time.)



```
X: 2
T:Barry's Favourite
% Nottingham Music Database
S:Mick Peat
M:2/2
K:D
A2|:"D"a3/2b/2a3/2g/2 f2(3def|"Em"g3/2a/2g3/2f/2 "A"e2A2|"D"f3/2g/2f3/2e/2 d2f2\
|"Em"B3/2c/2d3/2e/2 "A"c2A2|
"D"a3/2b/2a3/2g/2 f2(3def|"Em"g3/2a/2g3/2f/2 "A"e2A2|\
"D"f3/2g/2f3/2e/2 d3/2e/2f3/2A/2|"G"B3/2d/2"A"d3/2c/2 "D"d2A2:|
|:"G"B3/2A/2B3/2g/2 "D"d2A2|"Em"e3/2d/2e3/2f/2 "A"e2a2|\
"G"b3/2a/2(3gab "D"a3/2g/2(3fga|"E"f3/2e/2(3def "A"e2A2|
"G"B3/2A/2B3/2g/2 "D"d2A2|"Em"e3/2d/2e3/2f/2 "A"e2a2|\
"G"b3/2a/2g3/2f/2 "A"a3/2g/2f3/2e/2|1"D"d2f2 d2A2:|[2 d2f2d2|
```

Figure 1.1 – Example showing the ABC music format

Typically, generative language models attempt to model music by learning a probability distribution over music data, where the distribution determines how likely it is for a given note or chord to occur given the previous sequence of notes. Often the training data is in a symbolic music format like MIDI (Musical Instrument Digital Interface), or another representation such as ABC or MusicXML.

Perhaps the most widely used of these formats is MIDI. In spite of its age – it was first standardized in 1983 – MIDI is still in use today and continues to be the dominant format for representing music symbolically for digital applications. It is commonly used in synthesizers, digital audio workstations and other music software where accurate timing and synchronization is required. MIDI is the representation used in our work, and a more detailed overview will be given in Section 2.

After settling on a musical representation with which to train a model, it is necessary to choose a dataset – that is, a collection of songs or musical sequences over which a probability distribution can be learned. For music language modeling, there are a few choices. MIDI is again a good selection because of its lightweight nature; thousands of MIDI files can be freely downloaded from the web. Notable datasets for music language modeling include piano-midi.de's collection of classical piano music MIDI files[1], and the scale-chords dataset[2], which consists of chords and scales in various musical keys.

Since our interest in this work is in modeling video game music, we use a custom dataset collected from VGMusic.com[3], a website which hosts a large number of MIDI songs from video game soundtracks. The files on VGMusic.com consist of user-submitted transcriptions and remixes of the original soundtracks. VGMusic's staff page lists four individuals in charge of quality control, but since the MIDI files are user-submitted there is no guarantee of complete accuracy or faithfulness to the original source. Nonetheless it is the author's experience that

[1] http://www.piano-midi.de/
[2] http://www.feelyoursound.com/scale-chords/
[3] https://www.vgmusic.com/

the MIDI transcriptions from VGMusic are generally of good quality. We discuss methods for filtering the collected MIDI files for quality and consistency in Section 3.

The reason for the selection of video game music is partly due to nostalgia on the part of the author, and partly due to the fact that pre-CD-audio video game music is generally simpler than most recorded music.

Due to the limitations of computing during the late 80s and early 90s, video game consoles of the time could only devote a limited amount of resources to the audio and music within games. Thus systems like the Nintendo Entertainment System (NES), Super Nintendo Entertainment System (SNES) and the Sega Genesis (known as the Mega Drive in some countries) relied on simple tone generators or synthesizers to create sound. For example, the NES had only three channels for generating tones as well as a noise channel and a DPCM channel for playing low-quality samples – for a total of no more than five simultaneous sounds at a time. Compare this with most recorded music where an essentially unlimited number of sounds can occur simultaneously.

For this reason we hypothesize that early video game music (sometimes called “chiptune”) can be more easily learned by an RNN model, since it is less vertically dense than other kinds of music.

In Section 2 of this work, we give further background on the subject of music language modeling and related concepts, including a survey of similar works. In Section 3 we describe in detail the implementation our own music language model, which we call *vgm-rnn* for “video game music RNN”. In Section 4 we show that our network is able to model aspects of music including melody, harmony and rhythm, and generates output that cannot be reliably differentiated from human-created music by participants in a specially-designed survey. We suggest applications and directions for future work in Section 5.

2 Background

2.1 Overview of early video game music

In order to better understand the assumptions on which our model is based and what we are trying to achieve with it, it is necessary to know a little bit about early video game music.[4]

During the late 1970s and early 1980s, when arcade and video games came to prominence as a popular form of entertainment, the limits on computer power of the time dictated not only the quality of the graphics and gameplay of a game, but also of its audio and music. Since a game's sound is arguably less important than other factors, especially the graphics and gameplay, it was typically given a lower priority during development. Thus early arcade and home console soundtracks consisted mostly of primitive beeps and other sound effects created using simple tone generators. Some early games did not include music at all, as developers preferred instead to use the limited audio resources for sound effects. Video game sound chips of the time were either monophonic (i.e. they could play only one note at a time) or otherwise had only limited support of polyphony.

Representative examples include the sounds of Pac-Man for the arcade, and various games for the Atari 2600 home console, which was released in 1977.

[4] For more information on video game music in general, see: https://en.wikipedia.org/wiki/Video_game_music

See also the Commodore 64 (1982), which was one of the first computers with a specially-designed sound chip.

Figure 2.1 – NES showing game cartridge and peripherals

In 1983, Nintendo released its breakthrough home console, known as the Famicom (“Family Computer”) in Japan, but released in the northern hemisphere and elsewhere as the Nintendo Entertainment System (often abbreviated as NES) in 1985.

The NES revitalized the stagnant video game industry of the mid-80s, and set the standard for home console quality at the time. It made impressive advances in the quality of gameplay, graphics and most importantly for our purposes, audio.

The distinctive sound of the Ricoh RP2A03, the NES's sound chip, is in part the reason that the music of NES games is still well-known and fondly remembered today[5], even by those (including the author) who have never owned an NES. It consists of five channels: two square wave generators, usually used for melody and chords, a triangle wave generator often reserved for bass, and a noise channel for percussion and sound effects; in addition there is a Delta Pulse-Code Modulation (DPCM) channel for playback of digital samples. In total this makes for up to five simultaneous sounds that can be played at a time, although in practice all five channels are generally not used at once.

The music of the NES is most interesting for our purposes since because of the restrictions on the system, it is in general relatively simple, but can still make for an enjoyable listening experience. As will be explained further in Section 3, we train our model primarily on a collection of MIDI transcriptions of NES songs sourced from VGMusic.com.

During the late 80s and throughout the first half of the 90s – the so-called "16-bit" era – two major consoles dominated the market: the Sega Genesis, released in North America in 1989, and the Super Nintendo Entertainment System (SNES), Nintendo's follow-up to the NES released in 1991.

Both consoles improved upon the capabilities of earlier 8-bit consoles such as the NES. In addition to improved graphics, the Super Nintendo featured an all-new sound chip designed by Sony. It improved on the NES's five voices with a DSP that was able to play up to eight sampled voices simultaneously. In contrast, the

[5] The canonical example of NES music is the Super Mario Bros. theme, which can be heard here: https://www.youtube.com/watch?v=NTa6Xbzfq1U

Genesis featured six channels of simultaneous audio. Rather than digital samples like the SNES, the Genesis used frequency-modulation (FM) synthesis, a technique developed by Yamaha, to generate sound. The video game music of this period is more complex than that of earlier consoles, with increased use of musical dynamics (i.e. the relative volume of each note) but is still rather primitive when compared to recorded music of the time.

With the advent of CD-ROM-based consoles like the Sony PlayStation, video game audio saw a vast improvement. Instead of playing each note directly using the sound chip as with earlier consoles, high-quality recorded audio could now be streamed from memory. This marked the end of the “chip” music of the 8- and 16-bit eras. From this point on the music of video games would be more or less on par with recorded music in terms of quality. Rather than having to be specifically written for the hardware of the console, music could now be composed and recorded in the usual manner, and then streamed from the CD during gameplay.

It should be noted that because of how audio data is stored on early consoles like the NES, SNES and Genesis, it is easy to extract, note-for-note, the music data contained within a game’s ROM. This is generally not the case with CD-ROM-based consoles like the PlayStation since it is impossible to extract a song’s individual note events from an audio file recording.

The ability to extract and examine music data at such a low level means that transcriptions of early video game music tend to be more accurate than for later consoles. Intuitively, this makes for higher quality transcriptions, and helps to justify our choice of dataset. There is a wealth of MIDI transcriptions and remixes of early video game music online, and it is from these that we draw our dataset. In order to familiarize the reader with the peculiarities of the MIDI format, we next give a short overview of the MIDI standard.

2.2 The MIDI format

The MIDI (Musical Instrument Digital Interface) standard was first specified in 1983 by a group of synthesizer manufacturers from the US and Japan. One of the goals of MIDI was to create a unified system to synchronize electronic musical instruments from different manufacturers, as well as to provide for computerized control of musical equipment. Prior to 1983, this had been achieved using a combination of analog control signals and gates, a system known as CV/gate.

MIDI improved upon CV/gate by allowing for universal compatibility between devices that supported the standard, as well as adding additional features including a finer resolution for rhythmic events and better support for polyphony.

The MIDI standard additionally specifies a file format for representing MIDI data, known as Standard MIDI File or SMF. MIDI files contain a sequence or sequences of note and other events (tracks), as well as metadata including song and track names, cues and lyrics.

Many computer sound cards from the late 80s and throughout the 90s supported MIDI, and nearly all major operating systems today are able to play MIDI files. This is thanks in part to another specification of the MIDI standard: General MIDI.

General MIDI describes a collection of 128 instruments, from Piano to Bagpipes and beyond, as well as several technical requirements for devices that implement it. General MIDI ensures the consistent playback of MIDI sounds on different systems. For example, a MIDI track with a General MIDI program number of 0 (Piano) guarantees (or at least should guarantee) that some sort of piano-like sound will be heard when it is played back, regardless of the system it is played back on. It is for this reason that MIDI was often used on PC soundcards of the late 80s and 90s. General MIDI has a somewhat undeserved reputation for sounding "cheesy", but it is important to remember that it is only a specification, and the final sound that one will hear depends entirely on the system it is played back on.

This is perhaps the one aspect of MIDI that most confuses people – MIDI is not a sound, but rather a collection of specifications. A MIDI file is only a sequence of musical events in time, and how these events are interpreted and eventually played back is determined by the system it is played back on.

The most important events in the MIDI standard for our purposes are the Note events, consisting of Note On, which indicates the start of a note, and Note Off which indicates its end. Owing to its age, all MIDI data events are transmitted serially as 8-bit words. So, for example, if on our MIDI synthesizer we were to play the note C4 (known as "middle C") reasonably hard and then release it the following sequence of bytes would be transmitted (note that hexadecimal notation is used):

91 3C 6E

81 3C 00

The first byte of a message represents the message type ("Note On" for the first line) and the MIDI channel – in this case Channel 1. The second and third bytes represent the note's pitch (60 for middle C) and velocity (the force with which the note is pressed, in this case 110) respectively. A Note On event must always be terminated by a Note Off event (the second line above) or it will continue to sound indefinitely.

The way in which MIDI handles timing is somewhat tricky. MIDI time events are represented using "ticks" and a MIDI track's PPQ (Pulses Per Quarter) determines how many ticks occur in the space of a quarter note. Each MIDI event has an associated delta time, which is the time that it occurs relative to the event immediately preceding it. Thus in order to ascertain a note's absolute position within a MIDI track, it is necessary to first sum the delta times of all events in the track.

There are many other aspects of MIDI not described here, but hopefully it is clear to the reader that in spite of its widespread and continued use today MIDI retains many archaic features. In order to avoid some of the frustration of working with MIDI, we use the *pretty_midi* library for Python described in [9].

Despite its peculiarities, MIDI is a useful and lightweight way to represent musical data. As sequences of symbolic data, MIDI is ideal for use when training music language models. We describe the neural net structures most commonly used to build such models in Section 2.3.

2.3 RNNs, LSTMs and sequence learning

Recurrent neural networks, commonly known as RNNs, were first developed during the 1980s. In contrast to ordinary feedforward neural networks, the units of an RNN have recurrent connections – or loops – that allow previous outputs to be fed back into the network as input. In this way recurrent neural networks can be said to operate on sequences where each step of the sequence might represent a value in time or a position, for example a word or character in a sentence.

We can imagine a hidden layer in an RNN as a computational graph representing a chain of events unfolded across time:

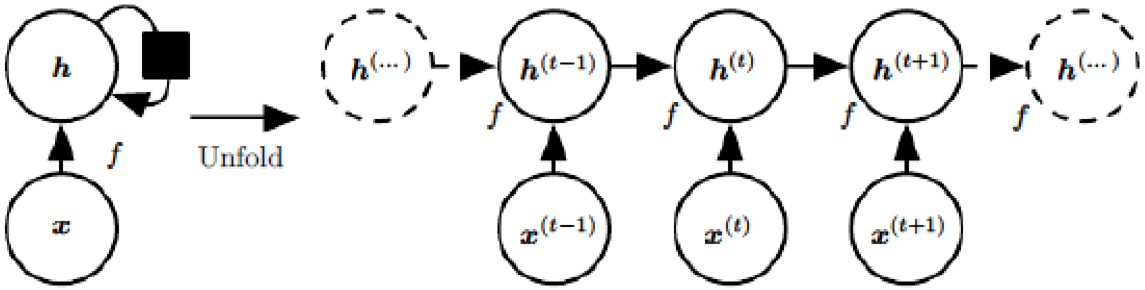


Figure 2.2 – RNN hidden layer shown as a computational graph unfolded across time

The output at step $\boldsymbol{t}$ (i.e. from $\boldsymbol{h}^{(t)}$) is fed as input to $\boldsymbol{h}^{(t+1)}$, and is dependent on the previous output from $\boldsymbol{h}^{(t-1)}$ as well as the current input $\boldsymbol{x}^{(t)}$. Thus, information about the output at previous steps is propagated forward in time. In this way recurrent networks can be said to have a "memory" of events occurring farther back in the past.

In order to learn the correct weights for a given task, RNNs typically use backpropagation through time, or BPTT, a variant of the ordinary backpropagation method used in feedforward networks. BPTT is performed on the unfolded graph of an RNN network in order to find the gradient of the cost, and thus use it to adjust the weights. In practical usage a more computationally efficient version called truncated BPTT is often used in place of the full method.

With ordinary RNNs, there is a limit to how far through time the gradient can be effectively propagated. It is difficult for a network to learn the importance of events occurring very far back in the sequence, and so the gradient tends to vanish (or less commonly explode), rendering the RNN unable to effectively learn long sequences. This is known as "the problem of long-term dependencies" and was identified as a drawback of the original RNN architecture as far back as the 1990s. Later innovations attempted to resolve this problem, one of the most successful being the long short-term memory, or LSTM, first proposed in 1997.

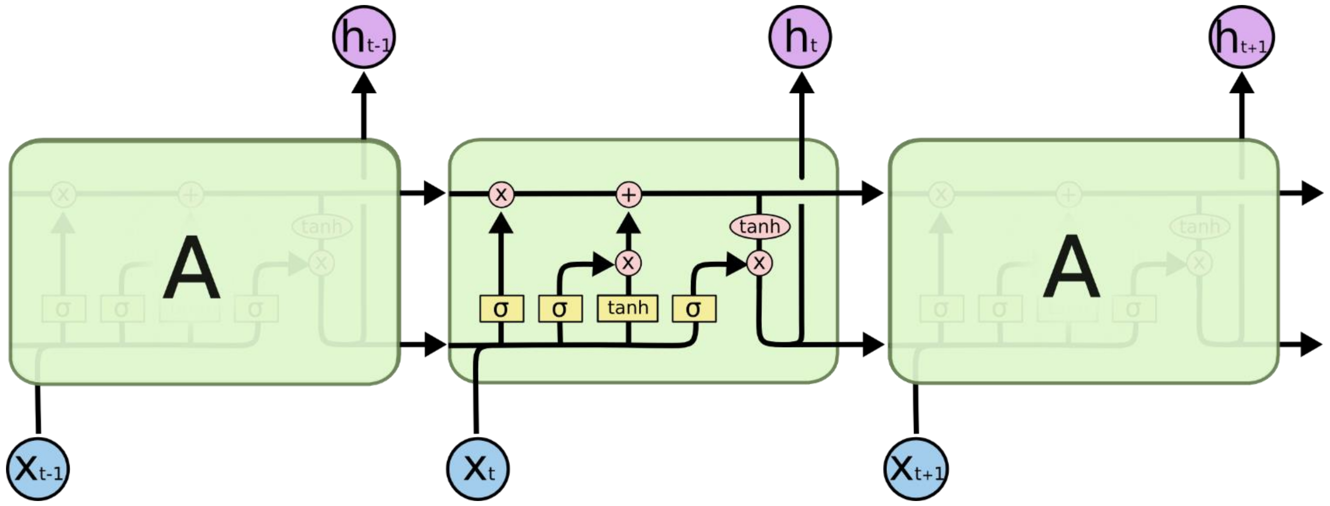


Figure 2.3 – Hidden layer showing LSTM units and gates

There are many articles and other resources[6] that describe LSTM networks in greater detail than will be done here. The main innovation of LSTM which allows it to alleviate the problem of long-term dependencies is the addition of a memory cell, consisting of several gates that allow an LSTM unit to more effectively remember or forget information about past events. For example, the "forget gate" decides whether previously learned information should be thrown

[6] See http://colah.github.io/posts/2015-08-Understanding-LSTMs/ for a particularly good explanation

away or not. In a language model – that is, one which attempts to learn to reproduce natural language – we might imagine the forget gate allowing the network to ignore the gender of a previous subject when conjugating verb forms after encountering a new subject with a different gender.

LSTM networks have been shown to outperform ordinary "vanilla" RNNs and to more effectively learn long-term dependencies. This has led to their near-ubiquitous use in modern recurrent neural networks, to the point that the two terms have almost become synonymous.

The applications of LSTM-based recurrent networks are numerous, and there have been many successes in for example, fields like machine translation and speech recognition. Such networks are ideal for learning from sequential information such as text, video and audio. Recently there has been interest in applying techniques from language modeling to other areas besides text and the spoken word, including music. Music itself is a kind of language, or family of languages, albeit with greater dimensionality than text since more than one musical event can occur simultaneously. To this end there have been a few notable recent examples of such music language modeling networks.

2.4 Related work

The work described in [3] is one of the first attempts to solve the problem of long-term dependencies in music modeling using recurrent networks, and one of the earliest to use LSTM to do so. The authors show that their network is able to learn the structure of a 12-bar blues chord progression, and to generate idiomatic melodies that fit the chords.

The training dataset consists of a single set of blues chords, as well as melodies written by the first author to fit over the chords. To train the network, measure-long segments of these melodies and their accompanying chords are concatenated, and then randomly sampled to create the final dataset.

For their first experiment, the authors attempt to teach their network to reproduce the training chord progression, without a melody. This is done by first training the network using a cross entropy loss function, and then allowing it to predict the probability that a given note is on or off given some previous input. The predicted output is then used by the network to predict the sequence following that. Using this method, it was found that the LSTM network was easily able to learn the chord progression under a wide range of settings.

Following this, an attempt is made to teach the network the chords and the melody in parallel. The model is trained in a way similar to the first experiment, until it learns the chord progression and the objective loss reaches a plateau. It is then allowed to compose music by using a sequence of notes as input for the first step, predicting the next note in the sequence, and then using the predicted output as input for each subsequent step.

Although no objective statistical methods were used to assess the quality of the generated music, it was found that the network was able to successfully learn the chord progression and to improvise new melodies over it. Additionally it was shown that the LSTM model was able to learn the global structure of the music better than a traditional RNN. In spite of the relatively simple training dataset and the restriction of only being able to generate notes of a single fixed duration, this was an important early effort in the field of music language modeling with recurrent networks. It should be noted that one of the authors, Douglas Eck, now heads the Magenta team at Google.

Another important work in the field is [1]. It describes a system that attempts to model polyphonic piano music, by way of a representation known as piano-roll notation. Music data is modeled as a two-dimensional matrix where the x-axis represents time and the y-axis represents musical pitch (see Section 3 below.) In this way vertical structures of notes (i.e. chords) can be converted easily into a form suitable for input into a recurrent network.

The main contribution of [1] is to combine RNNs and restricted Boltzmann machines (also known as RBMs) to model a joint probability distribution of note probabilities at each timestep given the musical context, in a structure the authors call RNN-RBM. That is, the model uses not only the previous notes in the sequence to determine which notes will follow when predicting the next step, but also the notes above and below the current note occurring during the same timestep. This is motivated by the fact that while in principle any set of notes in music can be combined to form a chord, in practice only a few combinations are used. Thus the presence or absence of certain notes can be used to infer whether or not a certain different note or group of notes might be likely to occur at the same time. Combined with an RNN to model probabilities along the time axis, this means that RNN-RBM is able to more accurately model polyphonic music than simpler networks.

Inspired by this architecture, the author of [7] creates a new model called BALSTM (bi-axial LSTM) that like RNN-RBM uses a joint probability distribution to predict note states, but that is also translation invariant. The idea of translation invariance should be familiar to those with experience in image processing using convolutional neural networks. There it refers to the fact that an image that is moved – or "translated" – vertically or horizontally without any additional modifications is still essentially the same image. Ideally, this should be reflected in the design of the model. In image processing, this is done using kernels, which learn specific features of an image.

We can see how translation invariance applies to music if we consider that when a musical structure such as a chord is transposed (moved up or down) into another key, it retains its essential identity. For example, a C Major chord contains the three notes C, E and G. If we were to transpose this chord up two semitones (divisions of the octave) to D, we would get a D Major chord, which contains the notes D, F# and A. Though the individual notes are different, the essential character of the chord is the same since they are both major. That is, the "quality" of a chord – which includes labels like Major and Minor and roughly relates to how "happy" or "sad" it sounds – is more important than the actual notes it contains (its key.)

Translation invariance is achieved in [7] by adding a delta value to each entry of the input note vector, such that each note vector is shifted into the same space. Other innovations include a temporal position vector indicating the current position within the measure, which is fed as input to the network in parallel with the note vector. This helps the network to learn measure-level structure in the music that it generates.

The music generated by the BALSTM model shows that it has learned features of sophisticated counterpoint and local structure[7], though it will sometimes become stuck on a chord or sequence of notes, repeating the same sequence indefinitely. Like many other music language models, it also plagued by a lack of long term structure. This is a major hurdle of language modeling in general and has yet to be overcome even within the state of the art.

The work in [8] extends the BALSTM architecture by adding the capability to model different musical styles, as well as several other small improvements.

In addition to what was just described, there have been other music language models that, for example, encode music data in a text format and attempt to learn the resulting string at the character level as in [2] or model musical events as tokens using an embedding matrix as in [4]. There also the various models by the Magenta team, which are too numerous to describe here.[8]

3 Project

3.1 Goals

A major goal of this work is simply to generate interesting music. We take a computational creativity-based approach and attempt to answer questions including, "Can our recurrent network accurately model early video game music? How does it compare with music composed by humans? Can what it generates be considered in some way creative?" Since concepts like "interesting" and

[7] See: http://www.hexahedria.com/2015/08/03/composing-music-with-recurrent-neural-networks/
[8] More information about Magenta can be found at: https://magenta.tensorflow.org/blog/

"creative" are somewhat subjective, we conduct a survey using the music generated by our model, the results of which are detailed in Section 4.

In our experiments, we assess the effects of different network architectures and parameter settings on the output in an attempt to generate music that is as subjectively "good" as we are able to achieve using our model. We also try to apply knowledge from music theory, including concepts of rhythm, harmony and melody, in order to create a model that is designed from the ground up with these concepts in mind.

We now demonstrate our model, *vgm-rnn*, and attempt to explain the thought process that led to its construction.[9]

3.2 Acquiring the dataset

Our first step is to acquire the dataset, which consists of MIDI files sourced from VGMusic.com. As of this writing, there are slightly under 32,000 MIDI files available from the website; however, we are only interested in those that belong to soundtracks from the early game consoles such as the NES, SNES and Genesis. Luckily, VGMusic organizes files into separate directories for each console.

To get the files, we create a simple web scraper in Python (see *midi_scraper.py*). We use the *requests* library for HTTP requests and *BeautifulSoup* for HTML parsing. The scraper consists of only one method, *get_midi_files*, which takes as arguments the URL from which to scrape the files and the name of the folder where the files will be saved. The scraper is specifically

[9] The full code is available for reference at: https://github.com/nmauthes/vgm-rnn

designed for scraping files from VGMusic.com, and has not been tested using other websites.

We first begin by collecting all the links on the specified webpage that end with ".mid", indicating a downloadable MIDI file.

```python
source = requests.get(args.url).text
soup = BeautifulSoup(source, 'lxml')

links = soup.find_all('a', href=True)
links = [l for l in links if l['href'].endswith('.mid')]
```

Figure 3.1 – Showing method for collecting links to MIDI files from VGMusic.com

After collecting the links to each MIDI file, we iterate through the array of links and save each MIDI file to the folder specified in the arguments.

```python
errors = 0
for i, link in enumerate(links[:args.max_files]):
    print(f'Downloading file {i + 1} of {len(links) if args.max_files >= len(links)
else args.max_files}')

    try:
        resp = requests.get(urljoin(args.url, link['href']))
        open(os.path.join(args.data_folder, link['href']), 'wb').write(resp.content)
    except requests.exceptions.Timeout:
        errors += 1
```

Figure 3.2 – Showing loop in which MIDI links are downloaded from VGMusic.com

If the request to download a MIDI file times out, we simply log the error and try the next link.

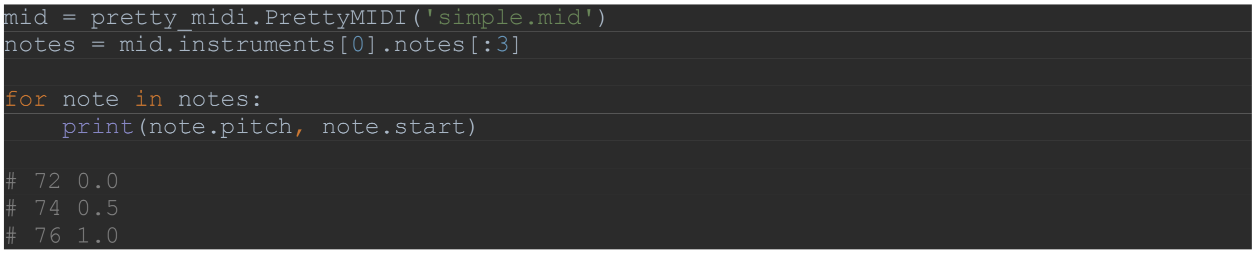
This is the core of the file scraper. We download all the MIDI files on the specified webpage, then filter them during the parsing process once they are on disk. Using this method we were able to scrape 4,194 MIDI files from VGMusic's NES page.

3.3 Parsing the dataset

In order to prepare the MIDI data for use by our model, it is first necessary to parse it and arrange it into a format suitable for input into our recurrent network. To this end we create several utility functions for parsing, filtering and modifying MIDI data; these are collected in *midi_parser.py*.

We mainly use the *numpy* and *pretty_midi* Python libraries for our parsing functions. The *numpy* library adds support for scientific computing and large multi-dimensional arrays and matrices, and should be familiar to anyone who has ever used Python for machine learning applications.

The *pretty_midi* library provides an intuitive and Pythonic interface for parsing, creating and manipulating MIDI data. Unlike other lower-level MIDI libraries, *pretty_midi* includes several conveniences that make working with the format easier. It arranges MIDI data into a list of instruments, each of which contains a separate list of notes and other events associated with that instrument. This is in contrast to the ordinary MIDI file format, where events are often spread chaotically across different MIDI tracks. For example, say we want to load a MIDI file from disk and then print the first three notes played by the first instrument in the file along with their start times in seconds. With *pretty_midi*, the code is straightforward:



```
mid = pretty_midi.PrettyMIDI('simple.mid')
notes = mid.instruments[0].notes[:3]

for note in notes:
    print(note.pitch, note.start)

# 72 0.0
# 74 0.5
# 76 1.0
```

Figure 3.3 – Example pretty_midi code

pretty_midi will automatically sum MIDI delta times so that the absolute position of note events can be easily determined. It also supports conversion from MIDI ticks to clock time and vice versa.

We use *pretty_midi* throughout the parsing routines contained within *midi_parser.py*. Of these functions, the three most important are *filter_midi_files*, *pretty_midi_to_piano_roll* and *piano_roll_to_pretty_midi*. Since the full code is too extensive to show in its entirety here, we instead give a brief overview of each.

As its name suggests, the *filter_midi_files* function is used to filter MIDI data – following conversion to *pretty_midi* object form – based on criteria including key and time signature. On completion, the function returns a list containing all the MIDI objects meeting the specified criteria. An option is provided to serialize the list and save it to disk, but this is discouraged as the serialized file itself can get quite large. We use this function in order to build our training set, by screening MIDI objects that may negatively affect training. This is important for assuring consistency of the dataset and can also be used as a form of quality control. The criteria for which to filter the MIDI objects is provided to the function as lists of strings. For example, if we want only the MIDI files from a certain folder that are in 4/4 time and in the key of C Major or D Minor, we might call the function like this:

```
midis = filter_midi_files(‘midi_folder’, [‘4/4’],[‘C major’, ‘D minor’])
```

Figure 3.4 – Example usage of the filter_midi_files function

By default, the list of filtered files is not saved to disk, so we must either use it immediately or, in this case, assign the list to a variable.

The two functions *pretty_midi_to_piano_roll* and *piano_roll_to_pretty_midi* are used to convert to and from piano-roll notation, respectively.

Using the *pretty_midi_to_piano_roll* function we can encode MIDI data into a format suitable for input to our RNN model. In this case we use a 2D piano-roll matrix such that the x-axis represents time (defined as some subdivision of the quarter note in MIDI ticks) and the y-axis represents MIDI pitch. To indicate that a note is played, a 1 is placed at the entry corresponding to the pitch and time where it occurs within the piano-roll; otherwise a 0 is used to indicate that no note is played at that position. In the example piano-roll chunk ***P*** below, a single note is played at the first timestep at $\boldsymbol{P_{1,1}}$ and a two-note chord is played three steps later at $\boldsymbol{P_{4,1}}$ and $\boldsymbol{P_{4,4}}$:

$$\boldsymbol{P} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The resolution at which note events in the MIDI object are sampled is determined by the *subdivision* parameter. In our case *subdivision* corresponds to the number of divisions per quarter note. For example, a subdivision of 4 would mean that a MIDI object is sampled at 16th note intervals. We also provide a parameter that we call *sensitivity* that specifies a tolerance as a percentage of the step length, such that notes that do not fall precisely on the musical grid can be included during sampling. This is useful when sampling, for example, MIDI files that were played by a human rather than manually inputted, in which case rhythms tend to be less exact.

By default, we ignore drum instruments (i.e. instruments on MIDI channel 10) and only sample notes from melodic instruments when converting from MIDI to piano roll. We hope to address drums in a later version of our model.

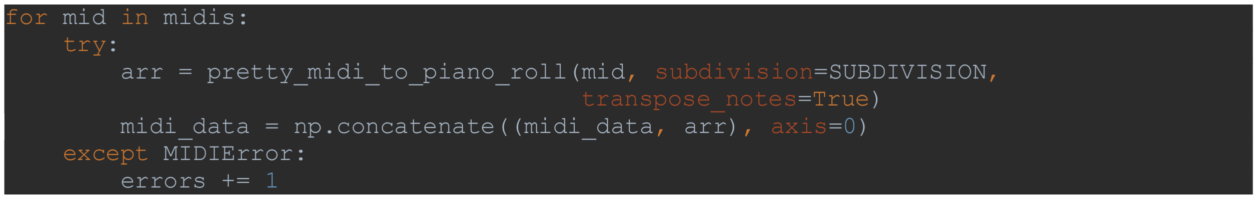
In order to reverse the process just described, we use the function *piano_roll_to_pretty_midi* to convert from a piano-roll matrix representation to a *pretty_midi* object, which can then be written to disk as a MIDI file. This is mostly useful during generation for converting the output of the recurrent network into a MIDI file format for listening. This function also has a *subdivision* parameter, which is used to specify the subdivision that was used when encoding the piano-roll matrix. It is important that the subdivision values during encoding and decoding match so that the generated MIDI file plays back at the right speed.

3.4 Training

All functions related to training of the model are contained in the file *rnn_train.py*. We use Keras – a high-level interface for deep-learning libraries – with a TensorFlow backend to implement our recurrent neural network.

The full code for training our network is contained in the main method of *rnn_train.py*. Before training begins, we first prepare the dataset. To save time, the code first checks for the presence of a previously serialized dataset, in the form of a *.npy* file; if one is not found it will build the dataset from scratch by filtering all the MIDI files in our data folder using the *filter_midi_files* function. The filtering process typically takes several minutes. We specify the criteria for filtering using two constants, *ALLOWED_KEYS* and *ALLOWED_TIME_SIGS*, which correspond to the allowed key signatures and time signatures respectively. Owing to a finding in [10], we allow all major and minor key signatures when filtering with the exception of C Major. C Major was shown in [10] to be a weak indicator of the actual key of a transcribed MIDI file, since many MIDI software packages insert it automatically as a default key signature when creating files. Additionally, we only include MIDI files that are in 4/4 time, the most common time signature in music. After running this procedure on our collection of scraped NES MIDI files, our dataset consists of 181 MIDI objects.

After filtering, we convert the collected MIDI objects into a piano-roll format using *pretty_midi_to_piano_roll*:

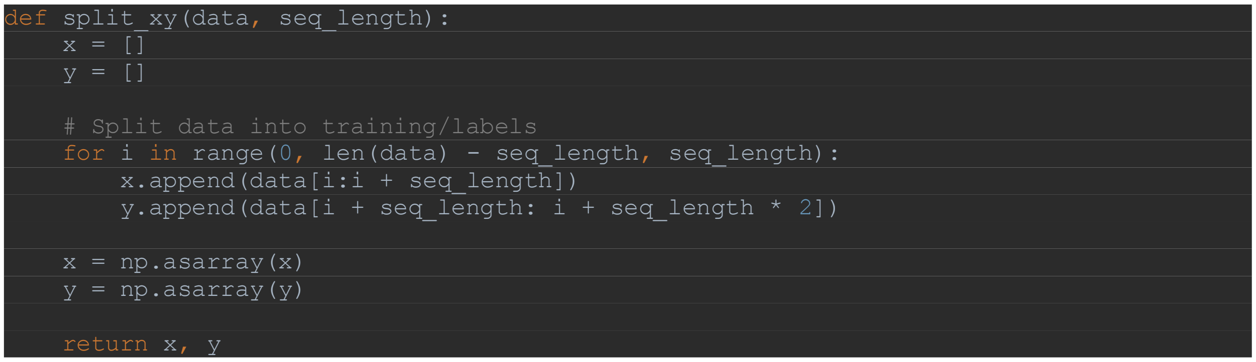


```
for mid in midis:
    try:
        arr = pretty_midi_to_piano_roll(mid, subdivision=SUBDIVISION,
                                        transpose_notes=True)
        midi_data = np.concatenate((midi_data, arr), axis=0)
    except MIDIError:
        errors += 1
```

Figure 3.5 – Showing conversion and concatenation of MIDI objects before training

We sample notes using a *subdivision* of 4 for a 16th note resolution and transpose all MIDI objects to a common key of C Major, due to a suggestion from [4]. If there is an error during conversion, we simply try the next MIDI object. We concatenate the converted piano-rolls into a single *numpy* array to get our final dataset. In total, this makes for 156,272 total timesteps worth of data.

We then divide our MIDI dataset into sequences, and split them into a training set and a set of labels using the function *split_xy*:



```
def split_xy(data, seq_length):
    x = []
    y = []

    # Split data into training/labels
    for i in range(0, len(data) - seq_length, seq_length):
        x.append(data[i:i + seq_length])
        y.append(data[i + seq_length: i + seq_length * 2])

    x = np.asarray(x)
    y = np.asarray(y)

    return x, y
```

Figure 3.6 – The split_xy function for dividing the dataset into training and label data

For each given sequence in the training set, its associated label consists of the sequence immediately following it; i.e. if a training sequence starts at a timestep ***i***, then its label will start at ***i + sequence_length***. Following splitting we clamp the pitch axis of each array (i.e. decrease the MIDI pitch range) in order to reduce the dimensionality of the data.

At this point the data is ready for input into our recurrent neural network. We then build and compile our model using Keras. For our loss function, we use the default categorical cross entropy in Keras, and use Adam as our optimizer.

We construct an RNN model consisting of a single LSTM layer with 256 hidden units. The number 256 was chosen based on a finding from [13], where it was determined that this was the optimal number of hidden units for a simple LSTM model, while avoiding being too computationally expensive. The final output layer in our recurrent network uses a softmax activation function to build a categorical distribution of note probabilities. A diagram of this network is shown below:

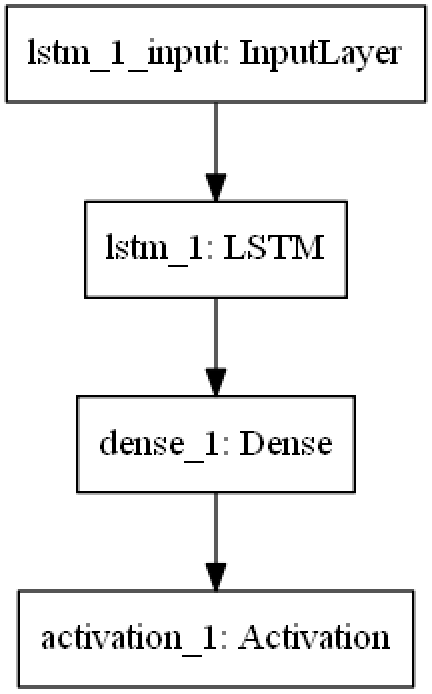


Figure 3.7 – Showing architecture of LSTM model

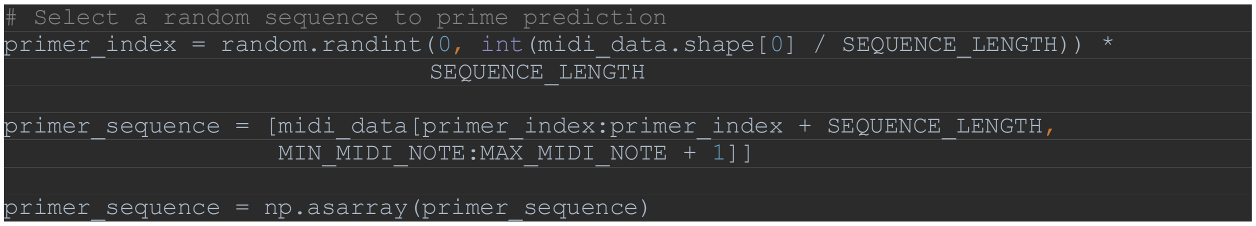
During training, we use a learning rate of 0.01, again based on a finding from [13]. We use a batch size of 50 and a sequence length during training of 64, which in our representation corresponds to 4 measures of music in 4/4 time. We clamp the MIDI pitch range to between the MIDI notes 36 and 84 – corresponding to a range of four octaves from C2 to C6.

We typically train for between 50-200 epochs. After training has completed, the final weight settings learned by the model are serialized and saved to disk; they can then be used during generation to predict new note sequences.

3.5 Generation

The generation process is more straightforward and requires less preparation than for training. All the code relating to generation is contained in the file *rnn_generate.py*, and the actual steps that make up this process are found in the main method therein.

In order to generate new MIDI data, we must first prime the network with a sequence of notes. During prediction, the network will use this primer sequence in order to predict a new sequence consisting of, for each note vector in the sequence, the probability with which each note in the vector is likely to occur, given the preceding sequence. We select the primer sequence at random from our dataset using the following code:

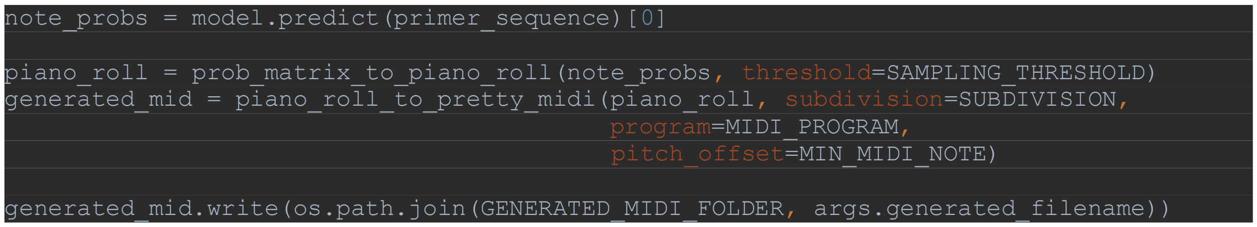


```
# Select a random sequence to prime prediction
primer_index = random.randint(0, int(midi_data.shape[0] / SEQUENCE_LENGTH)) *
                              SEQUENCE_LENGTH

primer_sequence = [midi_data[primer_index:primer_index + SEQUENCE_LENGTH,
                   MIN_MIDI_NOTE:MAX_MIDI_NOTE + 1]]

primer_sequence = np.asarray(primer_sequence)
```

Figure 3.8 – Code for selecting the primer sequence before generation

We arrange the primer data into a 3-dimensional array so that it is in the proper format for input to the RNN model.

Next, we rebuild the model by calling the *build_model* function from *rnn_train.py*. This will return a Keras model object identical to the one used during training. The trained weights are then loaded from disk into the model. The actual process of predicting the new sequence, converting it to *pretty_midi* format, and then writing the MIDI file to disk consists of only a few lines:



```
note_probs = model.predict(primer_sequence)[0]

piano_roll = prob_matrix_to_piano_roll(note_probs, threshold=SAMPLING_THRESHOLD)
generated_mid = piano_roll_to_pretty_midi(piano_roll, subdivision=SUBDIVISION,
                                          program=MIDI_PROGRAM,
                                          pitch_offset=MIN_MIDI_NOTE)

generated_mid.write(os.path.join(GENERATED_MIDI_FOLDER, args.generated_filename))
```

Figure 3.9 – Code for predicting, generating and writing a new MIDI sequence to disk

The option to save the primer sequence to disk for reference is also available. By repeating this process, we are able to generate as many new MIDI files as we want.

It is important to note that what is returned by Keras' *predict* method in this case is not a piano-roll, but rather a 3-dimensional matrix of note probabilities. Therefore, to convert it to a piano-roll format so that it can be used by our *piano_roll_to_pretty_midi* function requires a few additional steps. We achieve this using the *prob_matrix_to_piano_roll* function in *rnn_generate.py*, which converts from a probability matrix to a 2D piano-roll. To do this requires a threshold, that we specify as a constant called *SAMPLING_THRESHOLD*, which is used to determined which notes will make it into the final piano-roll. That is, for each entry in the note probability matrix that is returned by the model after prediction, only those notes that have been predicted to be played above a certain probability – the *SAMPLING_THRESHOLD* – will be included in the piano-roll. By decreasing this threshold, we allow notes with a low probability of being played to be included in the matrix, which can make for chaotic-sounding results. As we increase the threshold, note choices become more conservative, and the piano-roll tends to be somewhat sparse. Therefore, it is important to find a value for the *SAMPLING_THRESHOLD* that is somewhere in between these two extremes.

During generation, we use a *SAMPLING_THRESHOLD* value of 0.35. From our experimentation we found that this value results in the subjectively best sounding music, while also avoiding including notes with a low probability of being played.

We now describe the results of our project, including our experiments and evaluation.

4 Results

4.1 Experiments

We find that our network begins to generate coherent results after being trained for between 20-50 epochs. The network tends to converge quickly using the above settings, and the training time for a single epoch is typically one to several minutes. Training the network for the full number of epochs takes an hour or more using GPU-acceleration. The model was trained on a home laptop with an NVIDIA GeForce GT 740M graphics card using *tensorflow-gpu*.

The actual output of the network shows that it has successfully learned some features of the training data. It is clear, at least based on the author's musical training, that the generated examples[10] reflect the tonality (i.e. the overall key) of the primer sequences used to generate them. That is, the network will typically generate the correct notes for each key, and there is a minimum of dissonance (clashes between notes), at least to the extent that it was not already present in the primer sequence. This should hopefully be clear to the lay-listener

[10] Examples of the output from our model can be heard at: https://soundcloud.com/vgm-rnn

as a general sense that all the notes sound “nice” together and reflect a consistent mood. These results are especially interesting given the relative simplicity of this model.

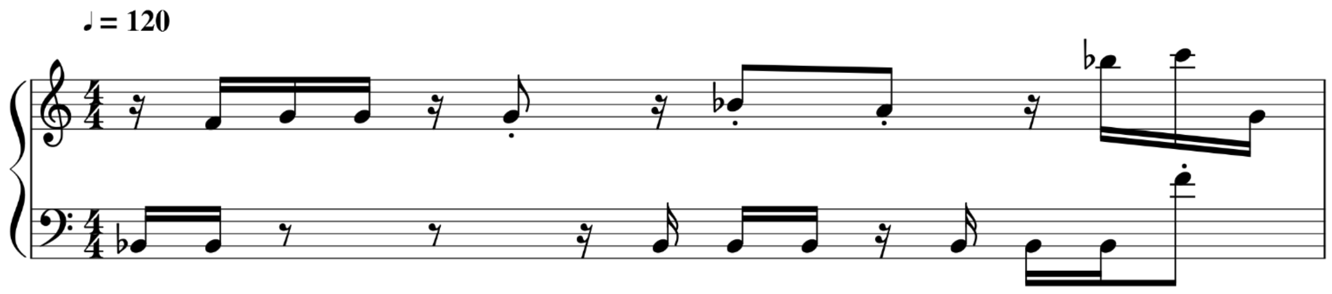


Figure 4.1 – Example output from single-layer LSTM model

The above snippet of example output shows that the network appears to have learned a repeating bassline and a melody, as well as simple chords.

Although the notes tend to be more or less correct, the rhythms generated by the network sometimes tend to wander. In particular, the resulting output often contains excessively repeated notes – that is, the same note is played repeatedly for several timesteps. Often this is not reflective of the rhythms present in the primer sequence. Because we are typically generating 4- to 8-measure long sequences, we cannot make any conclusions about the long-term structure of the music.

Nonetheless, there is the sense of a consistent rhythmic pulse underlying most of the examples, thanks in part to our musically-informed representation of MIDI data. The musical examples generated using this architecture are interesting (in the author's opinion at least) and show features that are consistent with those of the primer sequence that was used to generate them.

Interestingly, our attempts to make a more complex model by adding additional LSTM layers as well as dropout layers for regularization seemed to have a negative effect on the generated output. The generated MIDI files tended to be sparser and less musically interesting than those from the single-layer version with only early stopping for regularization. It has been shown in some cases that even simple RNN architectures can achieve impressive results, and are sometimes able to perform similarly to or better than more complex models.[11]

4.2 Survey

As a way to assess the subjective quality of the music produced by our model, we prepare a survey, which is administered online using Google Forms.[12] In it, participants are first asked about their musical experience, including whether they play an instrument, listen to music casually, or have studied music theory and compose music. They are then shown five ten-second clips of music. Each clip is either a piece of original music generated by our system, or a sequence taken from the training dataset.

The participant is informed that at least one of the clips they will be shown was composed by a computer and at least one is from an actual video game soundtrack composed by a human. After listening to each clip, they are asked whether they think it was composed by a human or a computer. They are also

[11] For example, see: http://karpathy.github.io/2015/05/21/rnn-effectiveness/
[12] The survey can be found here: https://goo.gl/forms/zuk02sCxuk9Tjckm1

asked to rate the quality of the clip on a scale of 1 to 10, where 1 indicates very poor quality and 10 indicates very good quality.

In this way, the survey acts as a kind of “musical Turing test”, where participants assess the quality of the music at the same time they try to determine whether it was composed by a human or computer.

In total, six respondents answered our survey. Because of time constraints, we were not able to conduct a more extensive study. Results indicate that those polled did not have much musical experience, with 50% of the respondents indicating “I enjoy listening to music in my free time, but I don't play an instrument” and the remaining 50% choosing “I have some experience playing an instrument, but I only play occasionally”. This is to be expected, since only members of the computer science community at SJSU were polled. Future surveys could include more diverse populations, including those with greater musical experience.

The average quality of each clip according to user ratings was 5.56, indicating middling quality according to our scale, though with a slight positive bias. There did not seem to be a significant difference between the average rating for human-created clips (5.42) and those generated by *vgm-rnn* (5.67). Interestingly, the highest rated clip, with a score of 6.33, was generated by our system and was identified as such by 100% of respondents.

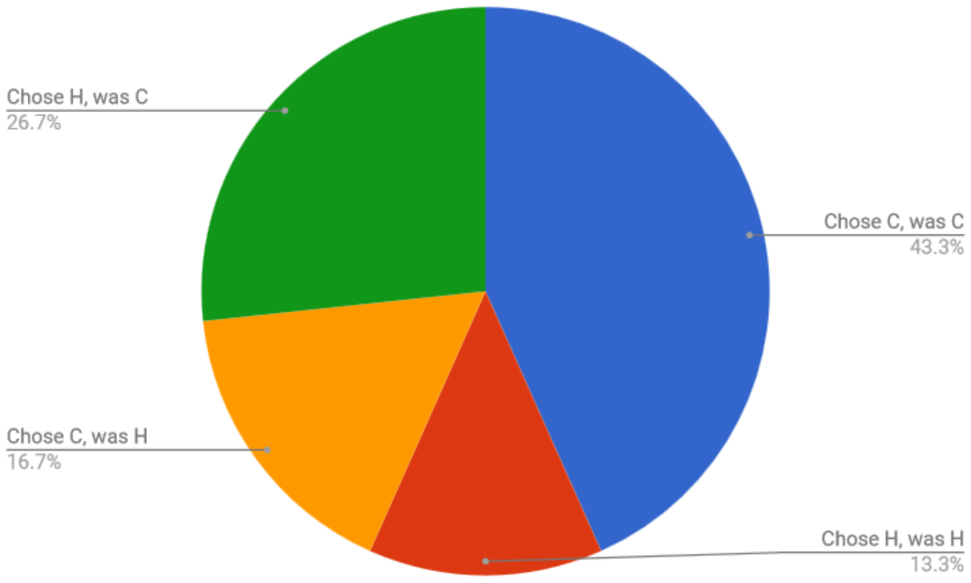


Figure 4.2 – Chart showing the results of our Turing-style test from the survey

Our results indicate that respondents had difficulty correctly determining whether a clip was composed by a human or computer. According to our survey, they were only able to do so 56.6% of the time, only slightly better than chance. These results are similar to those obtained by the BachBot project[13], whose respondents were only able to guess correctly 59% of the time. However, that survey was promoted over social media and thus conducted using a much larger pool of respondents, with more diverse backgrounds. Since the BachBot survey did not ask users about the quality of its musical examples, we are unable to compare our own results in this regard.

[13] See http://www.bachbot.com/

5 Conclusions

5.1 General conclusions

We found that even with a relatively simple RNN architecture, we were able to model musical features including melody, harmony and rhythm. Furthermore, we were able to show that the music generated by our system could not be reliably differentiated from human-produced songs by the respondents of our survey. While there are certainly areas where our model could be improved (see Section 5.3 for future work) we find these results promising and believe they are testament to the power of recurrent neural networks for sequence learning tasks, including for automatic music generation.

It is the author's belief that with additional improvements, a model such as the one we have shown here could be made to produce output that is very close – perhaps even indistinguishable – from video game music produced by humans.

Whether the music that our model generates can be considered "creative" is a philosophical question that we will not attempt to answer here, although given the recent advances in artificial intelligence, perhaps it will not be long before a music language modeling system similar to what has been shown here is created that is able to pass the musical Turing test and receive high marks in terms of quality – assuming it hasn't been created already.

5.2 Applications

There are several practical and theoretical applications of music language models like ours. Perhaps one of the most common, especially in the field of music information retrieval, is for use in automatic music transcription, or AMT. AMT uses machine learning techniques in order to automatically generate a symbolic representation from an audio recording of music. This is useful if, for example, one would like to study in detail the musical structure of a song or other piece of recorded music. In [13] it is shown that music language models using LSTM networks can be used for this purpose.

Our network is designed to model a very specific kind of music – early video game music – and although it could theoretically be used for automatic transcription of such music, the already wide availability of video game music transcriptions makes this somewhat unnecessary.

Perhaps a more interesting application for our model is to generate music for actual games. For independent game developers who lack the skills to compose their own music and are either unwilling or unable to pay someone else to write music for their game, using music that is automatically generated by a music language model such as *vgm-rnn* may be a viable option. Although our network is designed to model early video game music and not more contemporary styles, the recent popularity of "retro"-style games by independent developers suggests that there is indeed a demand for this kind of music.

5.3 Future work

There are several possible directions for future work. As of this writing, our model only considers melodic instruments when learning from MIDI training data. Since drums are also important to video game music, augmenting our recurrent network so that it can model percussion instruments in addition to melodic parts is of interest. There have been a few music language models devoted exclusively to modeling drums, such as in [2] and [5]. Ideally, our model would be able to generate drum parts in parallel with the melody, chords and bassline in a way that is idiomatic and musically pleasing.

In the future, we would like to try training the model on datasets other than the collection of NES songs which was used in this work – on soundtracks from the Super Nintendo and Sega Genesis consoles for example. It would be interesting to see how well our model is able to learn the more complex music of these systems. It may even be possible to implement a system similar to [8], in which we are able to learn and reproduce the different "styles" of music of each console.

Additionally, we believe that a more sophisticated architecture would further improve the generated output of our model. The biaxial LSTM network used in [7] and [8] has shown particularly impressive results in the author's opinion, and could be a possible model for future work. This and other structural improvements such as the ability to reproduce notes of different durations and to better model short-term rhythmic structure could help in generating results closer to human-created music. Finally, we would ideally like for our network to generate music with a well-defined long-term structure over time, although this has eluded even the most sophisticated models.

The author looks forward to implementing these and other improvements in the future.

6 References

1 Boulanger-Lewandowski, N., Bengio, Y., & Vincent, P. (2012). Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. In *In ICML 29*.

2 Choi, K., Fazekas, G., & Sandler, M. (2016). Text-based LSTM networks for automatic music composition. *arXiv preprint arXiv:1604.05358*.

3 Eck, D., & Schmidhuber, J. (2002). A first look at music composition using LSTM recurrent neural networks. *Istituto Dalle Molle Di Studi Sull Intelligenza Artificiale*, *103*.

4 Huang, A., & Wu, R. (2016). Deep learning for music. *arXiv preprint arXiv:1606.04930*.

5 Hutchings, P. (2017, May). Talking Drums: Generating drum grooves with neural networks. In *Proceedings of the First International Conference on Deep Learning and Music, Anchorage, US, May, 2017., pp. 43-47* (pp. 43-47).

6 Jaques, N., Gu, S., Turner, R. E., & Eck, D. (2016). Generating Music by Fine-Tuning Recurrent Neural Networks with Reinforcement Learning.

7 Johnson, D. D. (2017, April). Generating polyphonic music using tied parallel networks. In *International Conference on Evolutionary and Biologically Inspired Music and Art* (pp. 128-143). Springer, Cham.

8 Mao, H. H., Shin, T., & Cottrell, G. W. (2018). DeepJ: Style-Specific Music Generation. *arXiv preprint arXiv:1801.00887*.

9 Raffel, C. and Ellis, D. P. (2014) Intuitive Analysis, Creation and Manipulation of MIDI Data with pretty_midi. In *Proceedings of the 15th International Conference on Music Information Retrieval Late Breaking and Demo Papers*.

10 Raffel, C., & Ellis, D. P. (2016, August). Extracting Ground-Truth Information from MIDI Files: A MIDIfesto. In *ISMIR* (pp. 796-802).

11 Vogl, R., Dorfer, M., & Knees, P. (2017, March). Drum transcription from polyphonic music with recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on* (pp. 201-205). IEEE.

12 Weel, J., Intelligentie, B. O. K., & Gavves, E. (2016). RoboMozart: Generating music using LSTM networks trained per-tick on a MIDI collection with short music segments as input.

13 Ycart, A., & Benetos, E. (2017). A study on LSTM networks for polyphonic music sequence modelling. ISMIR.