University of Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme/specialisation: Computer Science	Spring semester, 2019 Open/~~Confidential~~
Author: Magnus Særsten Book	Magnus S. Book................... (signature of author)
Faculty supervisor: Vinay Jayarama Setty	
Title of Master's thesis: Generating Retro Video Game Music Using Deep Learning Techniques	
Credits: 30 ECTS	
Keywords: NES • Music Generation • Biaxial RNN • LSTM • Deep Learning • Tensorflow • Theano	Number of pages: 60 + supplemental material/other: - Zip file attached to PDF - Source code and generated music Stavanger, June 15 2019

University of Stavanger

Faculty of Science and Technology
Department of Electrical Engineering and Computer Science

Generating Retro Video Game Music Using Deep Learning Techniques

Master's Thesis in Computer Science
by

Magnus Særsten Book

Internal Supervisor

Vinay Jayarama Setty

June 15, 2019

“The time will come when diligent research over long periods will bring to light things which now lie hidden. A single lifetime, even though entirely devoted to the sky, would not be enough for the investigation of so vast a subject... And so this knowledge will be unfolded only through long successive ages. There will come a time when our descendants will be amazed that we did not know things that are so plain to them... Many discoveries are reserved for ages still to come, when memory of us will have been effaced.”

- Lucius Annaeus Seneca

Abstract

Music generation using deep learning is a widely studied field. This thesis focuses on music generation in a constrained and novel environment; retro video game music. The constraints imposed by the environment creates many unique challenges for the generation of musical compositions. In addition, the dataset consists of multi-instrument music, which is rarely studied due to its complexity. An extension to an existing architecture; the Biaxial RNN is presented in order to extend its capabilities to allow for generating multi-instrument arrangements. The resulting implementation is somewhat successful at fulfilling the proposed solution, although one component could not be implemented within the time limit. The result is not pleasant music, but it does give a view into the complex process of multi-instrumental music generation.

Acknowledgements

I would like to sincerely thank my supervisor Vinay Setty for his input and guidance on the project. In addition, I would also like to thank my family and friends for always supporting me and lending an ear through their kindness. Everyone mentioned have all been interested in the project, and have provided ideas and encouragement throughout the process. Among my friends and fellow students, I would especially acknowledge Tobias Helgeland for providing his expert guidance in the field of music, and the NES Synthesizer. His work was also the inspiration for the problem tackled in this thesis.

Contents

Abbreviations

GAN	**G**enerative **A**dverserial **N**etwork
LSTM	**L**ong **S**hort-**T**erm **M**emory
MIDI	**M**usical **I**nstrument **D**igital **I**nterface
NES	**N**intendo **E**ntertainment **S**ystem
NES-MDB	**N**intendo **E**ntertainment **S**ystem **D**ata**B**ase
RL	**R**einforcement **L**earning
RNN	**R**ecurrent **N**eural **N**etwork
TPU	**T**ensor **P**rocessing **U**nit

Symbols

symbol	name
σ	sigmoid function

Chapter 1

Introduction

1.1 Motivation

Composers are faced with creating complex music that often has many instruments harmonizing together. Developing tools to help the creative process in this regard could help inspire composers through the ability to generate artificial compositions. In the case of video games, there has been a recent trend for independent developers to develop games in a retro style. Because of this, the composers could benefit from artificially generated music that work within the retro limitations.

Another motivation is to further develop existing deep learning techniques. These developments could potentially be used for deep learning projects in fields outside music generation.

The biggest motivator is to see if this artificial intelligence is able to come close to mimicking music made by humans. This would push the boundary of artificial intelligence closer to that of human intelligence within the field of art.

1.2 Problem Definition

This thesis seeks to explore music generation for retro video game music using deep learning. In order to accomplish this, a corpus of existing music must be preprocessed to a format suitable for a neural network.

An architecture for the neural network must be decided on and developed. After the preprocessing step is finished, the network must be able to learn from the input data.

Finally, after the network has trained on the musical compositions long enough, the resulting network must have the ability to generate a new and distinct composition based on some seeded starting point.

1.3 Use Cases

If music generation of this type were to become sufficiently proficient, it could see use in numerous ways. The most obvious way is for people to generate their own music to use for their own purposes, whether for listening, sharing, or incorporating it into a soundtrack.

Another use case, as mentioned earlier, is for composers to be inspired by tracks they generate. For example, a composer could provide a section of a composition and let the network propose a continuation. Around this, tools could be created to this process easier for the musicians using it, by also allowing them to manually edit the generated parts. While editing, the network could present various alternatives for the next notes based on what it has learned.

1.4 Challenges

This project focuses on a music with four instruments. At this time, music with multiple instruments is the biggest challenge within the field of music generation. It is challenging since all instruments need to sound good while accompanying each other.

To overcome this challenge, a network architecture that is able to handle interaction between instruments must be developed. Here, each instrument needs to generate their own progression; deciding which note to play at what time. Finally, the possibilities all instruments deem the most likely must be combined to create good interplay between them.

Making generated notes more dynamic is another big challenge. This requires things like the volume of a note being altered over time to make it less bland.

All this also requires a lot of computing time due to the dimensionality of the data.

1.5 Contributions

This thesis contributes a novel architecture partially based on previous work within single-instrument music generation; adapted to generate music with multiple instruments. This is applied to a dataset consisting of music from the NES video game console, which has not been used for any major projects at the time of writing.

Alongside the proposed solution, other possible solutions for generating multi-instrumental music are discussed.

1.6 Outline

The following list gives an outline of the contents from all the following chapters:

Chapter 2 provides all the background information necessary to understand the architecture proposed in this thesis. It also serves to provide a short introduction to the necessary music theory, as well as the synthesizer, file formats utilized, and fundamental deep learning techniques. Lastly, a summary of related works is given; in order to give an idea of the current state of research around the topic.

Chapter 3 describes the approach behind the music generation. It lays a foundation by introducing a work that is heavily utilized within the proposed solution of this project. Afterwards, this approach is analyzed, and areas of potential improvement are explored. Finally, the proposed solution is described.

Chapter 4 presents the experimental setup, as well as the dataset used in the training of the proposed model. Finally, the results of generating music using the model is provided.

Chapter 5 serves as a discussion about the results, architecture, and, the implementation.

Chapter 6 concludes the thesis by summarizing the results and what has been learned from this project, as well as pointing to possible future directions and improvements.

Chapter 2

Background

This chapter aims to introduce all background topics relevant to the topic presented by the thesis. Included is some basic music theory, an introduction to the Nintendo Entertainment System (NES) synthesizer, as well as an explanation of the deep learning techniques used in the thesis.

2.1 Music Theory

For the purpose of understanding the method described in this thesis, some basic principles and terminology will be introduced in this section. This introduction will also serve as the foundation for understanding the NES Synthesizer as described in Section 2.2.

2.1.1 Timing

One of the most important topics in music theory is timing. There are rules that dictate when a note should be played. Table 2.1 shows the most commonly used note symbols. The name of the note indicates for how long it should be sustained. As an example, a half note is half as long as a whole note.

Note	Name
𝅝	Whole note
𝅗𝅥	Half Note
♩	Quarter note
♪	Eighth note
𝅘𝅥𝅯	Sixteenth note

Table 2.1: Relevant notes

Every arrangement is divided into measures, often called bars. Each bar should be filled with notes or rests so as to satisfy a time signature.

Time signatures allow for defining a rhythmic structure for an arrangement. The most common time signature is written like this: $\frac{4}{4}$. Here, the lower number represents which note is a beat, while the upper number gives how many such notes are in a bar. For example, the $\frac{4}{4}$ time signature indicates that there are four quarter notes in a bar.

2.1.2 Pitch

The pitch of a note is determined from the frequency of the sound wave. A higher frequency gives a ”higher” note. On the other side of the spectrum is a ”lower” note, which has more bass.

Certain pitches correspond to notes on a scale. It is traditionally represented by the first seven letters of the Latin alphabet; A-F. Going from one note to another is often called a whole-step. There are also half-steps between all notes except for between E and F. This is most often denoted by the ♯ symbol, for example with C♯. All these notes define pitch classes, this is because there are many pitches that produce the same note.

Octaves are defined as being the interval between a pitch and the doubled pitch. Notes that are exactly one or multiple octaves are denoted by the same letter, which means they are in the same pitch class.

2.1.3 Velocity

Another important term is velocity. Traditionally, this denotes how quickly a piano key is pressed down. Velocity determines how soft the resulting sound is. However, in digital music, velocity usually determines the volume of the sound.

2.1.4 Timbre

All musical instruments have a unique timbre. Timbre is an indication of the *color* or *quality* of the sound. The timbre of an instrument is determined by the physical mechanics that produce the sound. For example, a piano can play the same notes as an organ, but they still sound very distinct from each other.

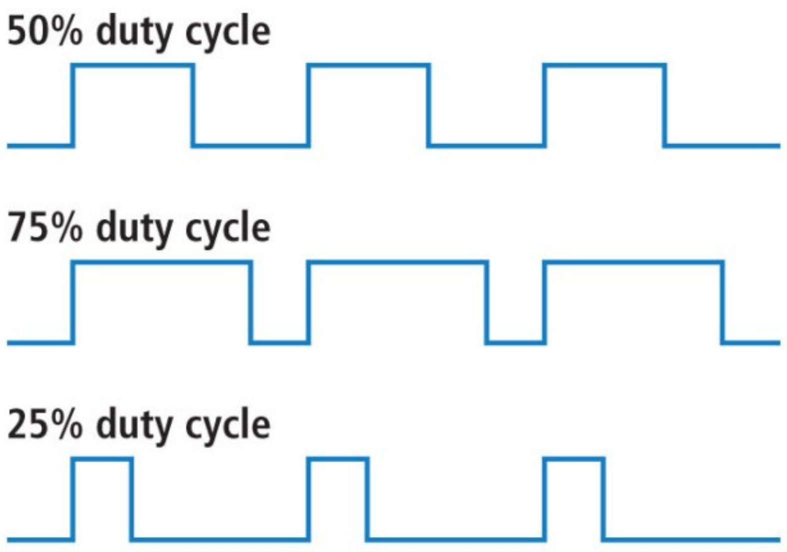

Figure 2.1: Pulse waves with varying duty cycles [1]

2.2 NES Synthesizer

A synthesizer is an electronic musical instrument. It is able to transform signals from some input and output sound. On a conceptual level, the NES synthesizer is very simple due to hardware constraints around the time of release. It contains five channels; two pulse wave channels, a triangle wave channel, noise channel, and a sample channel. The last channel will not be discussed, as it was rarely used by composers.

Later in the console's lifespan, composers were able to expand the range of possibilities by developing many techniques to compose more complex arrangements. Some of these methods will be described briefly to give an indication of their intricacy.

2.2.1 Pulse Waves

Pulse waves have a rectangular shape, which was previously illustrated in Figure 2.1. These waves can be modified by adjusting the duty cycle[1]; thus changing the timbre of the sound. In addition, velocity determines the volume of the resulting sound.

Composers usually used the pulse waves to create the melody of an arrangement. With two of these channels they could use both waves to create a deeper melody by playing the same notes in different octaves or forming chords. Another option was to use one of the channels for accompaniment by emulating another instrument.

[1]The NES has four available duty cycles: 12.5%, 25%, 50%, and 75%. It should also be noted that 25% and 75% give the same sound; because 75% is the negation of 25%

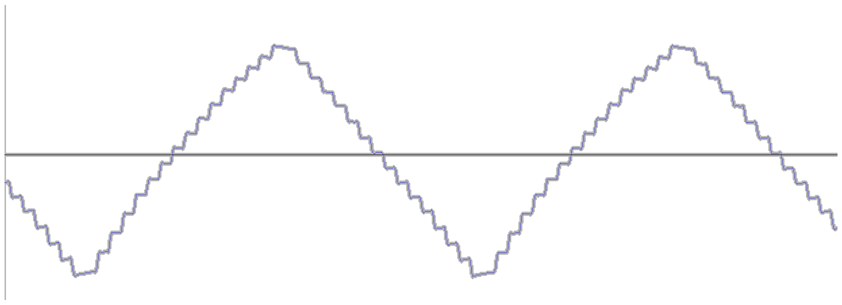

Figure 2.2: NES triangle wave [2]

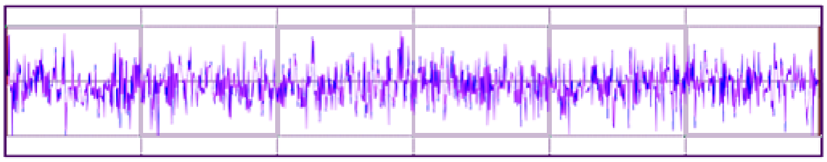

Figure 2.3: NES noise channel [4]

2.2.2 Triangle Wave

The triangle wave creates a deep and iconic sound. Because of this, it is most often used to emulate bass and bass drum. As seen in Figure 2.2, the wave is shaped like a triangle.

A side effect of the shape is the lack of volume control for this channel. This is because all amplitudes are used to create the shape.

2.2.3 Noise

There is a pseudorandom noise channel included in the synthesizer. The pattern of the wave is determined randomly based on an internal system timer. The pitch of the resulting tone can be changed the same way as the other channels. However, the noise channel has fewer available pitches to choose from compared to the other channels[2].

Figure 2.3 shows one of the possible waves generated by the noise channel. In most cases, the noise channel is used to emulate percussion by forming patterns of short activation with different pitches.

2.2.4 Advanced Compositional Techniques

As discussed earlier in this section, the synthesizer provides a very restricted environment for composers due to the limitations of the hardware. Nonetheless, over the life span

[2]The pulse and triangle channels can use 109 pitches, while the noise channel only has access to 17 [3]

of the NES, the composers were able to develop techniques to make more complicated music than what seemed possible.

These techniques, among others, served to better emulate real instruments and their capabilities. An echo effect could be achieved by having the two pulse channels play the same sequence with a temporal separation.

Changing the duty cycle while playing a note was also used to create the illusion of plucking the strings of a string instrument. When this was done rapidly, it also gives the effect of playing arpeggios[3] if done correctly [5].

By bending the pitch of a note while it is being played, the composers were able to replicate kick drums to create a thicker percussion compared to the noise channel. Using a similar technique, an echo could be created with a single pulse wave channel by making the pitch and volume lower over time.

These techniques usually make the arrangements much more complex in terms of composition and progression as the piece progresses.

2.3 Music Representation

There are many different ways to represent musical compositions; both in terms of file formats and visual representation. In this section, the two representations used for this thesis are introduced.

2.3.1 MIDI

A Musical Instrument Digital Interface (MIDI) file is used to store the notes to be played in a musical arrangement. In contrast to many other audio file formats, MIDI files do not contain any audio. Instead, they consist of instructions for which note to play at any given time, thereby making MIDI files much smaller than audio recording formats [6].

Each file can have a maximum of 16 instruments. An instrument track contains MIDI events that detail which note should in addition to other parameters such as the velocity, treble and timing of the note.

MIDI files can easily be both read and manipulated due to their structure, thus making it an ideal format to use for music generation.

[3]An arpeggio is a *chord* (multiple notes played at once) played individually in sequence

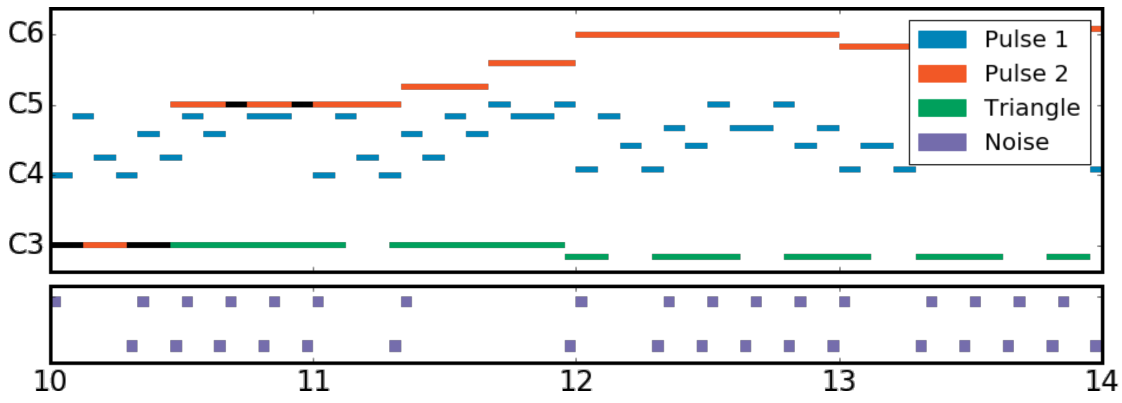

Figure 2.4: Example of a NES score represented with piano roll [3]

2.3.2 Piano Roll

A common way to represent music is through the use of musical notation. However, this notation may be confusing for people without a background in music. An alternate method to visualize an arrangement is to use piano roll.

Piano rolls are long rolls of paper with holes corresponding to the note to be played by a piano. These rolls were used with automated pianos, and would recreate the arrangement outlined by the roll. Although the mechanism for replaying these rolls are irrelevant to this thesis, the format can still be used to intuitively illustrate a musical arrangement.

Figure 2.4 gives an example of a score visualized using piano roll. Here, each instrument is given a unique color. The pitch of the notes being played are measured using the vertical axis. Time is depicted by the horizontal axis. This example illustrates how the piano roll representation is able to convey the pattern of the score in some time interval.

2.4 Deep Learning

Modern artificial intelligence is mostly based on the study of deep learning. Deep learning is fundamentally based on the inner workings of the human brain. There are many possible techniques for accomplishing a goal with deep learning. Each of them with their own advantages and disadvantages. This section aims to provide a basic understanding of the underlying concepts of the methods used in this thesis.

2.4.1 Feedforward Neural Networks

The most basic form of deep learning is an artificial neural network. As the name suggests, this concept is based on concepts observed in biological neural networks such as the human brain. Such a network is made up of neurons in multiple layers, where each neuron is connected to all neurons in the next layer.

In artificial neural networks, the connections between neurons are called the weights of the network. Whether or not a neuron fires depends both on the weight and value of the previous neurons connected to it. The weight of a connection is usually denoted as W_{ij}. Here, neuron i is the predecessor of j. In other words, the connection can be seen as going from i to j.

All outputs of the neurons can be calculated using matrix multiplication between the input and weight matrices. Equation 2.1 defines the previously discussed calculation of the output Y. Here, the input is denoted by X, while B is the matrix of biases of the network. This bias serves to change the difficulty of activating certain neurons.

$$Y = X \cdot W + B \tag{2.1}$$

2.4.1.1 Activation Functions

To decide if a neuron should be activated, an activation function is applied on the output. This function usually transforms the input in a non-linear manner. The simplest example of an activation function is a step function. Here, the neuron will be activated if $Y >$ threshold. Another activation function more commonly used is the sigmoid function displayed in Figure 2.5 and defined in Equation 2.2. This function is much more nuanced compared to the step function, since output can be anywhere between 0 and 1.

$$\sigma = \frac{e^x}{e^x + 1} \tag{2.2}$$

Activation functions are an important aspect within neural networks, and the choices made in this regard may heavily influence the performance of the network.

2.4.1.2 Architecture

Figure 2.6 provides an example of a feedforward neural network architecture. The reason it is called a feedforward neural network is due to the general direction of the connections between neurons. All connections point to the next layer of neurons, eventually leading to the output layer. This example is illustrated with only one hidden layer, but complex problems may call for additional hidden layers.

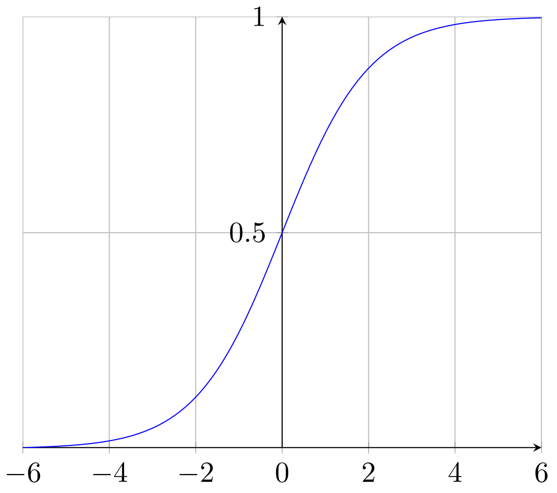

Figure 2.5: Sigmoid activation function

The input layer performs no computations. It is simply responsible for inserting the data into the network. For instance, if the input were to be defined by a binary number, each neuron in the input layer would correspond to one of the binary digits; their output being the value of their assigned digit.

Hidden layers are where where the network figures out which output to produce based on the input. The outcome is heavily dependent on the activation function, as well as the number of layers and the amount of neurons they contain. It can often be a challenge to tune these parameters to get a more efficient network with acceptable accuracy in its predictions.

Finally, the output layer represents the final prediction of the network based on the input data. Much like the input, it could for example represent a binary number. This number could for instance be a category, if the objective of the network is to classify input into various categories.

2.4.1.3 Learning

Initially, a neural network is most often given randomized weights. Most likely, these will not enable the network to make correct predictions. To improve the weights, the network needs to be trained using data that matches the usecase. The input data needs to have a correct solution associated with it, called the ground truth.

Training the network occurs in iterations until some condition is met, such as reaching a certain accuracy or iteration. All the steps for each training iteration are explained in the following list:

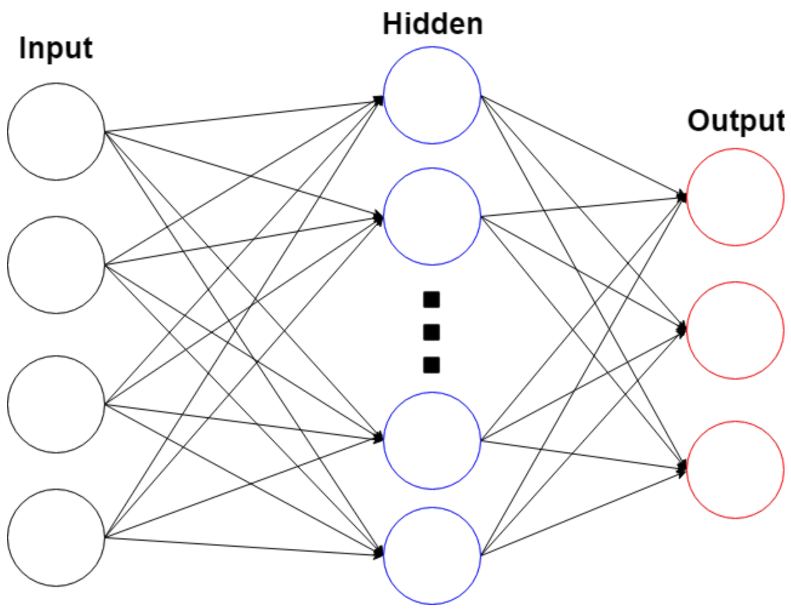

Figure 2.6: Example of a feedforward neural network architecture

1. **Feed-Forward:** To start off, input is inserted into the network, and is taken through the network, hence the name of this step. Once the network has produced an output, it moves onto the next step.

2. **Calculate loss:** Here, the objective is to determine how close the prediction from the previous step was compared to the ground truth associated with the input. Just like there are many activation functions, there are also many loss functions.

   A simple example of a loss function is to use $\text{loss} = |\text{truth} - \text{prediction}|$. The problem with this approach is that many small errors can end up amounting to a large sum across the entire dataset. A few large errors could also end up giving the same sum. One way to solve this issue is to square the absolute error, thus making big errors have a more substantial impact on the overall error of the network.

   The main goal of the neural network is to minimize the chosen loss function, which means it is a problem of optimization.

3. **Calculate the gradient:** By performing derivation, the gradient is produced. It can be used to optimize the loss function by approaching a local minima of the loss function.

   For simplicity, imagine a network with a single weight. In order to minimize this function, the derivative is calculated to determine if the weight should be increased or decreased. There are three possible scenarios here:

   - If the derivative is 0, the weight does not need to be changed, since a minima has been reached.

- If it is positive, that means the loss is increasing. Therefore, the weight should be decreased.
- If it is negative, the loss is decreasing, the weight should be increased.

The same applies applies for gradients, which are multi-dimensional derivatives that describe the impact on the loss for all weights averaged over the entire training set. Much like the simple case described earlier, the method consists of increasing or decreasing each of the weights according to the gradient. This concept is called gradient descent, and is a central concept within deep learning.

4. **Backpropagation:** Now that the gradient for the loss function has been calculated, the network must propagate the changes to the weights through the network. Starting at the output layer, each training example indicates which neuron should have the highest activation. In order to come closer to the desired outcome, the weights connected to this neuron need to be altered to change the output value.

   By increasing the weights from neurons with a positive activation, and decreasing it for negative activations, the activation of the output neuron will increase. The changes should be made in proportion to the activation of the neuron, in order to descend towards the minima quicker.

   This must also be done for all other neurons in the output layer to decrease their activation. The sum of all these changes to each weight are taken to determine its final change for this iteration.

   This same logic is then used to propagate further back in the network until the input layer is reached. When this process is performed across all training data, it approximates the gradient mentioned earlier.

There are many optimizations and changes that are made to the steps described here, but these steps serve as the basic intuition of how a neural network learns.

2.4.2 Recurrent Neural Networks

Feedforward neural networks are good at classifying many types of data. However, it is not optimal when used to predict the next value of a time series. In such a case, the data point to predict is most likely close to the previous value; following the trend of earlier data.

To improve learning for applications concerned with temporal data, Recurrent Neural Networks (RNNs) were developed [7]. The main improvement to the feedforward neural network is the inclusion of a recurrent connection from the end of the hidden layer to

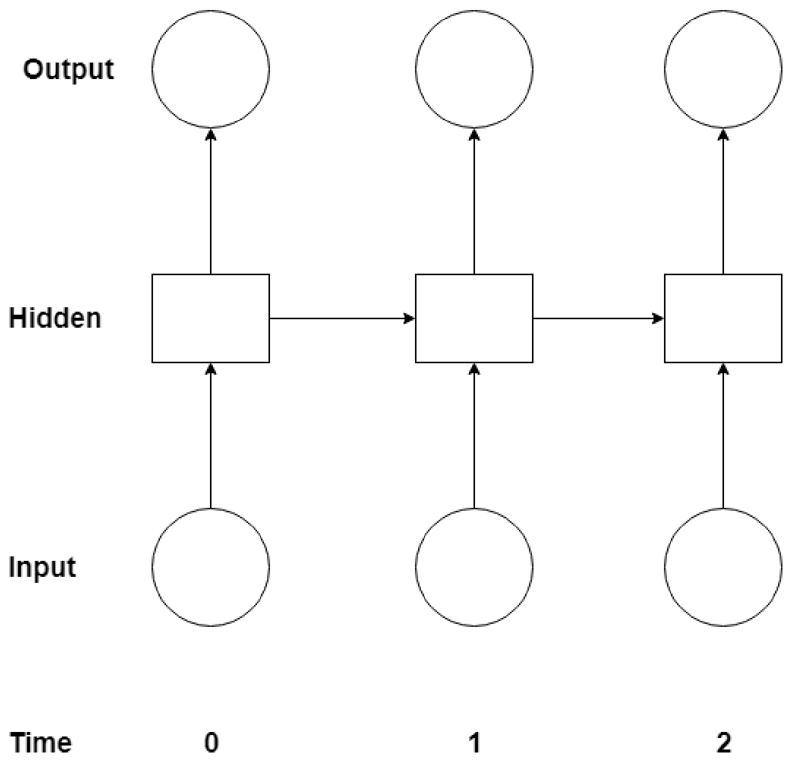

Figure 2.7: Architecture of a simple recurrent neural network

the beginning. This allows the network to retain information from previous data. Figure 2.7 visualizes how a RNN connects the hidden state from previous time steps with the hidden state in the current time step.

Although this technique manages to achieve relatively good accuracy on many problems involving temporal data, there are still some problems. These are the problems of vanishing and exploding gradients. Both these problems arise because of the recurrent connection, which makes it so the gradients are constantly being multiplied. Depending on the values of the gradients, the problem can evolve to vanishing gradients or exploding gradients.

Exploding gradients occur when weights have a high value. If this is the case, the value of the weight will explode; as it is continually multiplied. This problem could eventually lead to an overflow and terminate the model. However, there is a reasonably simple solution to this problem. By truncating the weight within a certain range, a complete explosion can be prevented.

Conversely, vanishing gradients occur when weights smaller than 1 are continually multiplied. Eventually this leads to many weights becoming inconsequential. Figure 2.8 shows how repeated applications of the sigmoid function are affected by the recurrent connection. There are a few ways to deal with this problem, but the most popular method is explained in Section 2.4.3.

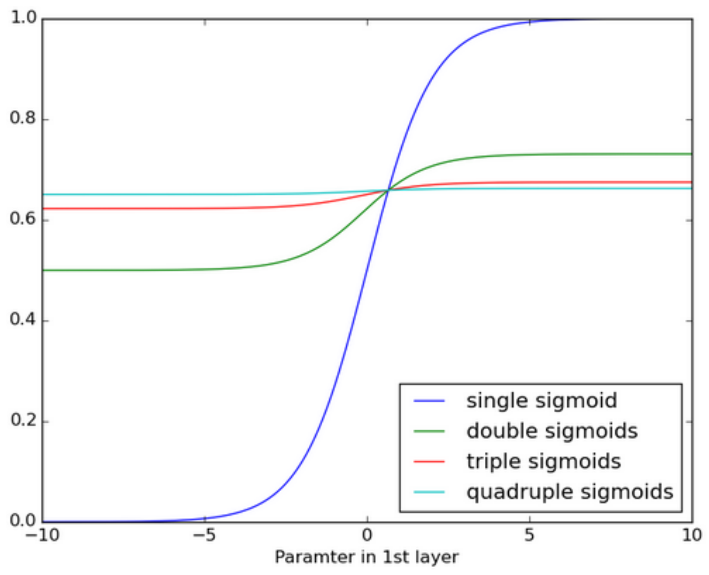

Figure 2.8: Vanishing gradients illustrated using the sigmoid function [8]

Another benefit from the RNN architecture is the ability to provide different forms of output. It is possible to output a prediction at each time step, generate multiple outputs from one input, or classifying multiple data points. This significantly increases the breadth of problems where the network may be used.

2.4.3 Long Short-Term Memory Neural Networks

As seen in Section 2.4.2, RNNs face the problem of vanishing gradients. The solution most often implemented is a Long Short-Term Memory (LSTM) neural network. It introduces a new mechanism to combat the vanishing gradient problem, namely memory cells.

A schematic for a memory cell is depicted in Figure 2.9. It shows the input node, as well as three different gates. Each gate can be in either be open or shut at each time step. In the figure, the blue arrows can be safely ignored, as they are seldom used, and only contribute to the performance of the architecture.

The input gate determines whether or not new input should be let into the block. If the forget gate is open, the cell will forget its state. When the output gate is open, it will output its state; allowing subsequent blocks to read the output.

Each gate has its own, separate set of weights that is adjusted through learning to learn when it is beneficial to open or close each gate.

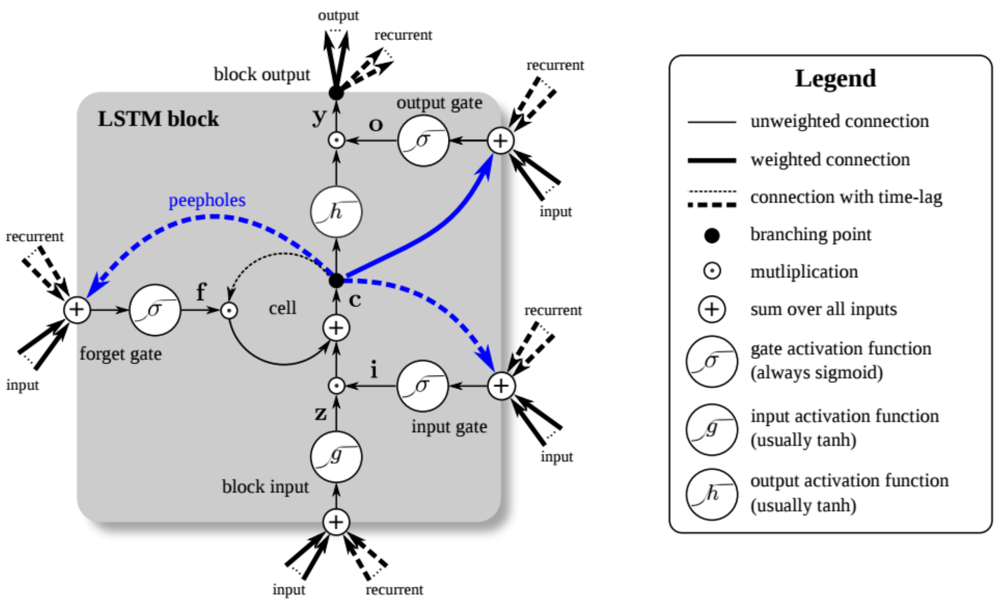

Figure 2.9: A schematic showing the structure of a single LSTM cell [9]

The reason gates that allow for ignoring input, not producing output, or forgetting the state is due to enabling learning on different time scales. For example, this could help in recognizing seasons, and their impact on a time series, while also predicting the next data point based on the previous.

One of the biggest reasons why this architecture is able to avoid the vanishing gradient problem is because of the central addition operation. In a regular RNN, the states are multiplied, while they are added in a LSTM block. In essence, the addition preserves a relatively constant error in the system, unlike multiplication.

2.5 Related Work

Music generation is a broad topic containing many different problems, each of which often has its own solution. Some examples are the differences between generating a melody, accompaniment, or synthesizing a sound. Each of these problems have had various deep learning architectures applied in order to produce a good result. Table 2.2 gives a brief summary of the related works that involve music generation. Most of the works mentioned here are also described in the comprehensive literature survey by Briot, Hadjeres and Pachet [10].

Few related works presented in this section are very closely related to the work presented in this paper. This is largely because there is a distinct lack of works focusing on music generation with multiple instruments. However, the related works presented here are works that stand out in their ability to generate single instrument music. Hence, the novel feature of the work presented in this thesis is the generation of multi-instrument music, as well generating music in the genre of retro video game music.

In addition to the works presented in this section, a deeper exploration of a work that serves as a building block is described in Section 3.1.

Title	Description
Performance RNN (Simon & Oore, 2017)[11]	Generates music with expressive timing and dynamics by training on real piano performances. It uses a dataset consisting of performances by skilled human players to attempt to capture the characteristics of human musicians. A RNN architecture is used to generate the scores. This work relates to the architecture presented in this thesis because it uses RNNs to generate music.
Modelling High-Dimensional Sequences with LSTM-RTRBM: Application to Polyphonic Music Generation (Lyu et al., 2015) [12]	Combines a LSTM architecture with a Restricted Boltzmann Machine (RBM) to improve the performance on high-dimensional data such as music. The resulting network is capable of generating music in a variety of genres. Similarly to this work, this thesis also uses LSTM as a base technology, but uses a more advanced architecture.
C-RNN-GAN: Continuous recurrent neural networks with adversarial training (Mogren, 2016)[13]	Uses a Generative Adverserial Network (GAN) to generate classical piano music. This network uses RNNs for learning from the corpus. This work can be seen as a counterpoint to the architecture presented in this thesis, being another architecture that has potential to generate multi-instrument music if expanded to provide this feature. GAN networks have been shown to perform well in many projects where the goal is to generate novel content.
Tuning Recurrent Neural Networks With Reinforcement Learning (Jaques et al., 2017)[14]	This work proposes a method for tuning the result generated from a RNN based architecture using Reinforcement Learning (RL). Using this technique, the network will be incentivized to follow rules from music theory. This would for instance prevent large jumps in pitch. This work relates to the thesis through the use of a RNN based architecture. In addition to this, it could be an interesting future work to implement the RL tuning component to make generated arrangements more musically sound.
DeepBach: a Steerable Model for Bach Chorales Generation (Hadjeres et al., 2017)[15]	Generates convincing pieces based on the compositions of Johann Sebastian Bach. It is unique in allowing for a user to steer generation through providing parameters to the model. It uses two different RNNs; one for predicting backwards from the future, and one to predict forwards from the past. It is also able to generate harmonic accompaniment given a melody. It relates to this thesis by being able to successfully generate pleasant classical accompanies to a melody. Because of this ability, it could be possible to use this model for generating accompaniment to a generated melody, thus producing entirely novel music. Another aspect of this work is the ability to steer the compositions by providing parameters to the network. This could be a future feature of the work presented in this thesis.

Table 2.2: A summary of related works within the field of music generation

Chapter 3

Solution Approach

The aim of this chapter is to explain both the proposed solution for generating music from this thesis. First, the underlying neural network architecture that is utilized in the solution is explained and analyzed. Finally, the architecture of the proposed solution is described; along with how it is trained.

3.1 Existing Approach

One of the most successful architectures for generating music with a single instrument is called the Biaxial RNN, proposed by Daniel Johnson [16]. The name of the network comes from its unique structure of having two networks; one working in the time axis, the other in the note axis. Figure 3.1 illustrates the structure of a Biaxial RNN. Here, the note axis is represented in the vertical direction, while the loops represent the time axis.

As mentioned earlier, each axis is defined by a neural network, specifically, they are LSTM networks. In each time step, the time network produces an output first, which is then fed into the note network.

The initial input into the time axis LSTM consists of:

- **Position** gives the MIDI value of the current note.
- **Pitch class** is a vector indicating which pitch class the current note belongs to. It contains 12 values, starting with the A note, and subsequently increasing by a half-step.
- **Previous vicinity** provides information on surrounding notes in the previous time step, ranging from an octave up, to an octave down.

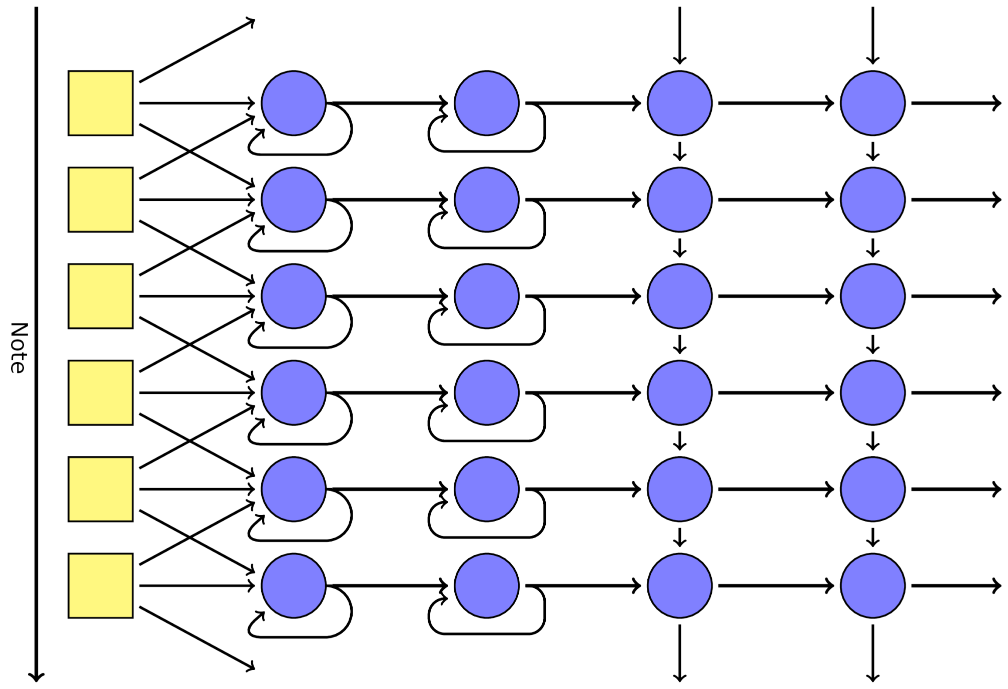

Figure 3.1: Biaxial RNN schematic [17]

- **Previous context** indicates how many times a note x was played in the last time step. It is represented by a vector of size 12 (one value per pitch class), where the value at index i is defined as $(x - i - \text{pitch class}) \mod 12$.
- **Beat** is a vector containing binary strings, and represents at what beat notes are played. If using a resolution of sixteenth notes, there will be sixteen digits in the binary string, each representing whether a note is played at this beat. The example describes a measure with four beat inputs, represented by the rows, and each time step equivalent to a column:

```
1001101000110100
1010101010101010
0001001100100101
0010100110011011
```

On the other hand, the note axis network iterates through note steps, which are similar to time steps, but ranging from the lowest note to the highest note. This network takes the following input:

- **Note state** is an input directly from the last layer of the time axis LSTM stack. It is a vector describing the time pattern of the notes.

- **Previous note state** is a value that describes if the note in the previous note step was played (it can be either 0 or 1).
- **Previous note attenuation** resembles whether the note in the previous note step was sustained.

After the LSTM networks, there is an output layer that outputs the play probability for the notes, as well as an articulation probability. In this case, the articulation probability refers to whether or not a note should be sustained if it is played.

The loss function is a modified version of cross-entropy, which involves calculating the likelihood of playing the correct note; according to the training sequence.

Training this network is different from many other neural networks. Often, a training epoch is defined to be a complete iteration on the training dataset. Since music is a creative process, this type of training could end up with an overfit network. Another problem is the large amount of data due to the amount of time steps in the data. The solution used in this network is to randomly sample a song from the training set. From the selected song, a small sequence is randomly extracted and used as the target for the training epoch.

When generating a new composition, an existing song not in the training set is used as a seed. A small section at the start is used as a starting point. The task of the network is to generate the rest of the song based on the start of the seed.

3.2 Analysis

Listening to some of the music generated by the network from David Johnson's original blog post [17], the results are quite convincing. All the generated music uses advanced timing, sometimes altering it dynamically through the song. In addition to this, it also generates chords and arpeggios that work together and sound good.

However, there are a few problems that stand out while listening to the samples. When a long song is generated, there are stretches where the network repeats the same note or chord for a long time, thereby making some sections monotonous. Another possible problem is that the network is discouraged from creating new sequences, since it has to learn from the songs in a dataset. This could possibly make the output somewhat formulaic; without much experimentation with regards to timing or note choices.

A nice aspect of the Biaxial RNN architecture is a training optimization, allowing for all the operations of the time axis LSTM stack to be batched together. This results in larger matrix operations, which is ideal for making use of a GPU. Despite this optimization being applicable during training, it is not possible to utilize during generation. This is because each note must be decided before determining anything about the next time step.

Even though this network is good at producing polyphonic[1] music with a single instrument, there is no support for multi-instrumental music. When it performs its preprocessing, it collapses all tracks into one, thus making it impossible to distinguish them later. This is the main problem for this architecture within the context of this thesis, where the goal is to generate multi-instrumental music. The proposed solution is described in Section 3.3, where Biaxial RNNs are used as a sub-component.

3.3 Proposed Solution

Generating a multi-instrumental composition has some challenges that must be overcome in order for the result to be listenable. Most important of all, is the interplay between all the instruments of the composition. Figure 3.2 illustrates the architecture of the proposed system.

The solution integrates multiple Biaxial RNNs; one for each MIDI channel. Each network is responsible for generating the probabilities for playing each note in the next time step for its assigned instrument. Theoretically, the networks should be able to adapt to the play style of their assigned MIDI track.

Although each generated track should be of good quality; as discussed in Section 3.2, they have no coordination. It may be postulated that there should be some method of instilling interplay between the generated music tracks. In the architecture, this is achieved through the use of a neural network; called the composer.

The composer component is a LSTM network responsible for selecting the note combinations that compliment each other. It takes the play probability and articulation probability outputs from all connected Biaxial RNNs. It is trained to recognize good combinations across the instruments based on what is present in the training set. A cutoff for play probabilities that are considered is used to promote the inclusion of notes that have been determined to work with the previous context by the Biaxial RNNs. The composer is trained together with the Biaxial RNNs by using their predictions as input, and attempting to predict the correct note combination at that time step.

[1]The term polyphonic indicates that multiple independent notes can be played simultaneously by an instrument

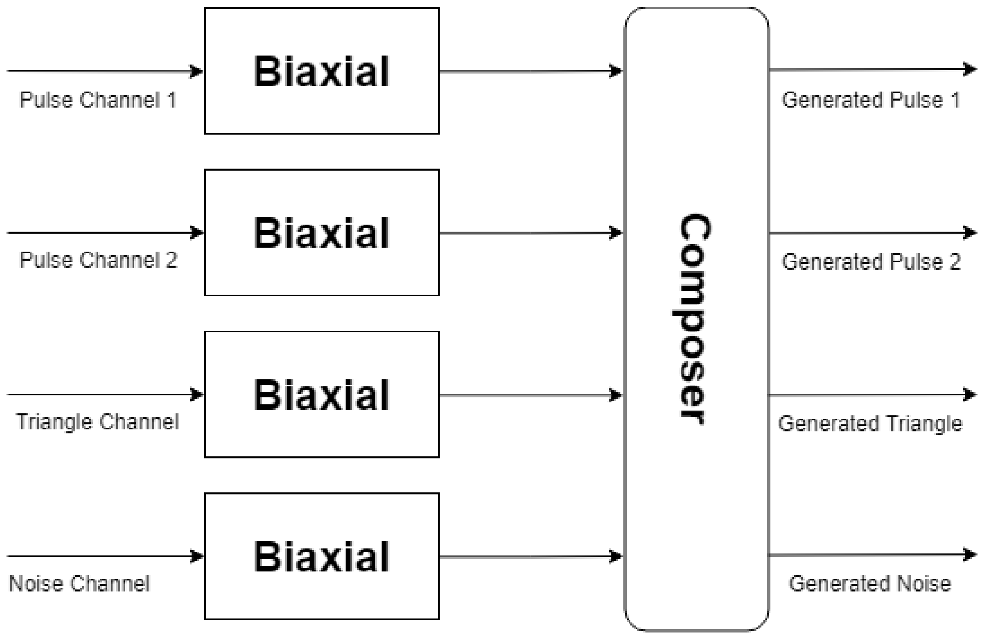

Figure 3.2: Proposed architecture for generating NES music using four Biaxial RNNs and a composer component to improve interplay between channels

After the composer makes a prediction, it outputs the chosen note at that time step for each of the channels. Therefore, during generation, the output can be appended to the generated tracks directly.

Based on the structure and components of the architecture, it should be able to generate some coherent compositions. The results are presented in Section 4.2, while some advantages and disadvantages with the network architecture are discussed in Section 5.2.

Chapter 4

Experimental Evaluation

This chapter presents the results gained from running the project and generating music. The experimental setup is described to give an indication of how the project was executed. Afterwards, the dataset is explored; along with preprocessing steps taken before feeding the data to the model. Finally, the results obtained from the experiment are presented.

4.1 Experimental Setup and Data Set

As seen in Section 3.3, the proposed solution requires a complex architecture. In order to reduce the time required to train the network and generate music, the network has to be run on a computer with a fast GPU. For this project, a virtual machine was created on the Google Cloud platform [18]. It uses a Tesla V100 GPU, which is optimized to perform common deep learning computations. Ideally, a Tensor Processing Unit (TPU) would be used, since it has better performance, but the most central operations in this project have yet to be given TPU support at the time of writing.

Each Biaxial RNN was configured with two layers for each axis. The amount of nodes per layer in the time axis was 300 for both, while the note layer used 100 in the first layer, and 50 in the other layer. However, an exception was made for the network responsible for the noise channel. Since there are a lot fewer possible note values in this channel, the note layer sizes were set to 50 and 25 respectively. The time axis was left the same as for the other channels because the note placement is just as complex.

Due to some last-minute difficulties with the implementation, as discussed in Section 5.3, there was not enough time to run it for as long as planned. The implementation was trained for approximately 30 hours, completing 3000 epochs. Ideally, it would have been training for 10000 epochs.

The dataset used for this project is the NES Music DataBase (NES-MDB) [3]. It consists of 5278 songs from the soundtracks of 397 games. All songs were scraped and converted to multiple formats that are easier to work with. They provide MIDI, piano roll, natural language, and a raw format. For this project, the MIDI format was chosen. This decision was made since it is the most commonly used format within the field of music generation. Since the format has been present for a long time, there are also many tools that aid in conversion and other facets.

Although the MIDI format is made to be played by sound cards, the format is not ideal for deep learning projects. To solve this, an array of state matrices is computed and stored on disk during preprocessing. Each track (i.e., instrument) in the MIDI file is represented by a state matrix. A state matrix is a two dimensional matrix that describes the state of the track at each time step. In the state dimension, an entry corresponds to each possible MIDI pitch. If the note is played, the value will be represented as 1; otherwise, it will be 0.

The state matrix format is also the output of the generation process. Hence, it needs to be converted back to a playable format. Converting to MIDI is trivial, since it is simply reversing the process for converting from MIDI to state matrices.

Before running the experiment, the dataset was filtered to remove outliers. An outlier in this case is a song that does not follow the musical structure of other songs. Most songs are made to be looped from the end to the beginning, but the dataset contains some songs that are only meant to play once. These pieces are more characteristic similar to jingles. An example most games have is a short song played when the player looses. What these songs have in common is their short length. Because of this, any songs of 10 seconds or less were filtered out of the dataset to make the input to the network more uniform.

After filtering the dataset, there were 3400 songs left. Out of these, 2000 songs were converted to the state matrix format through random sampling and used as training data for the network.

4.2 Experimental Results

As mentioned in Section 4.1, the training had to be cut short due to time constraints. Nonetheless, the results will be presented in this section, both in terms of the loss of the model, and some generated music.

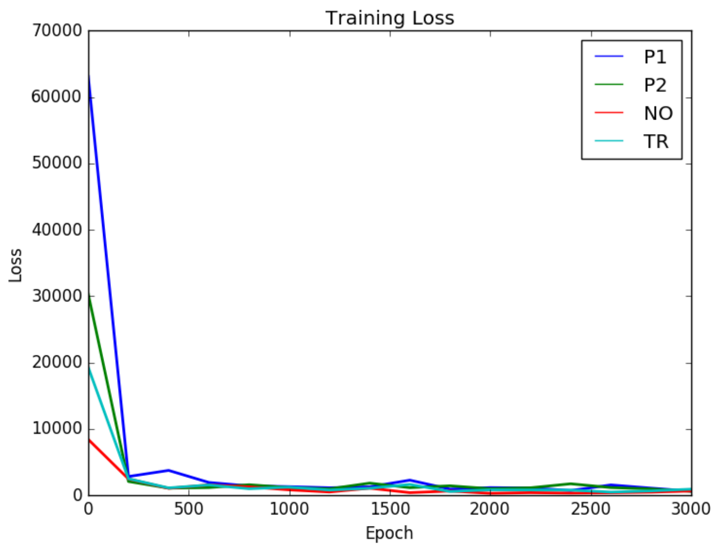

Figure 4.1: Training loss gathered from 3000 epochs, recorded every 200 epochs

Figure 4.1 shows the loss of the model over the 3000 epochs it was run. This plot may indicate a difference in complexity between the different sound channels. For example, the noise channel, which has many fewer possible notes starts with a lower loss compared to the others. Conversely, the primary pulse wave channel starts with a much higher loss. However, all Biaxial RNNs seem to converge rapidly, indicating it is training effectively.

The song represented in the piano roll format in Figure 4.2 shows a generated composition after training for 3000 epochs. One thing to notice is the amount of concurrent notes in the triangle channel. Since the implemented portion of the architecture only gets the output from the Biaxial RNNs, the notes that are above a threshold of play probability are all played. This would not be supported by the NES, but it does give some insight into which notes the network was likely to pick at any time. All of the channels except for the triangle channel display some competence in terms of note timing, while also producing some sequences that sound fine when played separately.

For the sake of comparison, the song visualized in Figure 4.3 was generated during training at 1000 epochs. Here, the second pulse channel is almost empty, perhaps indicating that the seed did not contain any notes in that channel. Another possible explanation is a lack of learned note connection for that Biaxial RNN's note axis. Another anomaly to notice is the large jumps between notes in the triangle channel, which is most often not a pleasant sound.

Finally, Figure 4.4 is another song generated after 3000 epochs, which is a lot sparser and more musically sound compared to the other generated songs shown. Here, the noise channel creates an interesting rhythm pattern with its percussion. It should also be

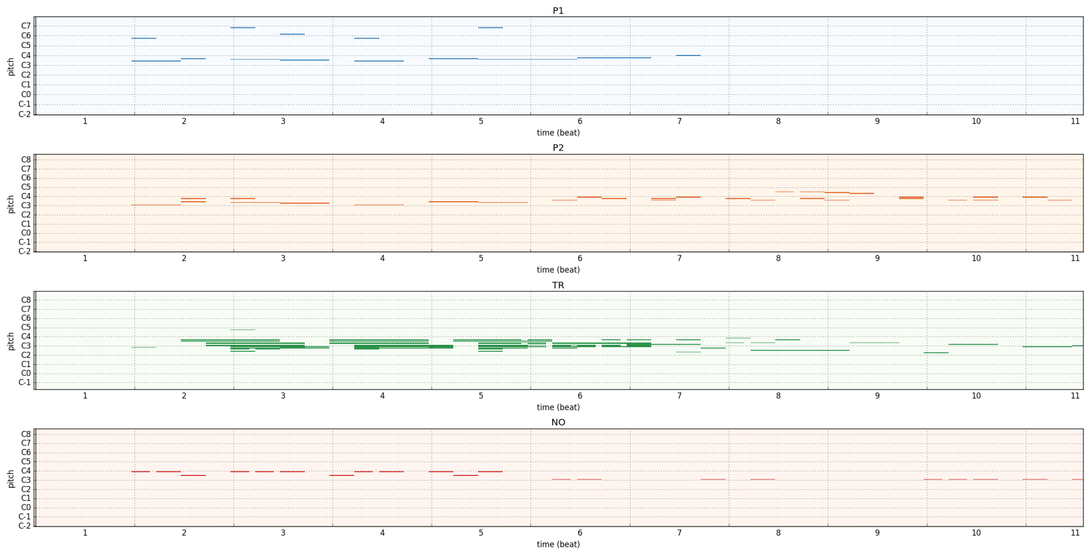

Figure 4.2: Generated composition with many simultaneous notes at 3000 epochs

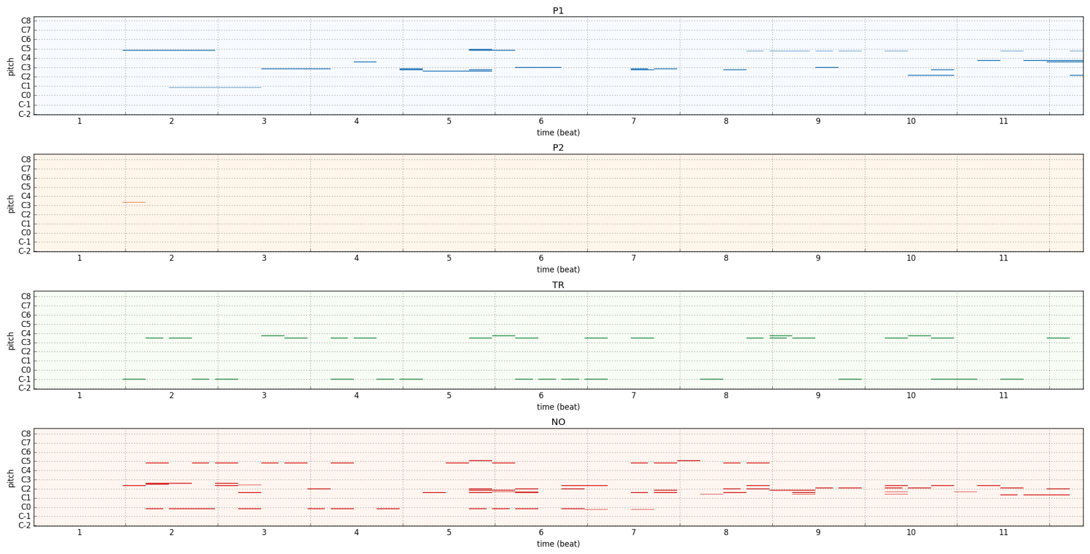

Figure 4.3: Generated sample after 1000 epochs

noted that this song contains very few simultaneous notes being played compared to the other samples.

After generating these songs, they can be converted to other formats more portable formats, such as WAW and MP3. One problem encountered with this conversion, however, was that the noise channel could not be assigned a percussion instrument. This resulted in lots of very low notes in the converted audio versions embedded with this thesis.

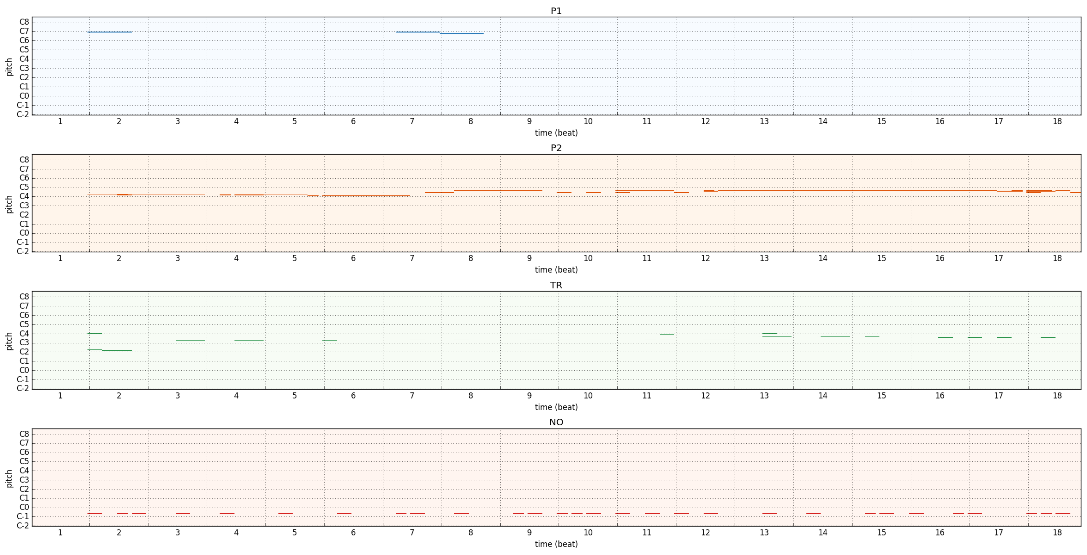

Figure 4.4: Sparse generated piece generated after 3000 epochs

Chapter 5

Discussion

This chapter aims to discuss multiple topics related to the solution and premise of the thesis. Results are discussed to provide some reflection and possible explanations for why the results from Section 4.2 were achieved. Subsequently, the architecture, as presented in Section 3.3 is discussed; discussing the advantages and disadvantages, as well as some alternative architectures to be considered. Finally, the implementation will be discussed, both the original implementation that was abandoned, in addition to the final implementation.

5.1 Results

From Figure 4.1, all the Biaxial RNN models appear to converge quickly, which is usually a sign that it may be time to stop training. However, within the field of music generation, this may not be the case. After 200 epochs, when the models appear to converge, they have only learned from 200 short samples. It will probably be useful to continue training, if only for the sake of expanding the repertoire of the models.

Due to the random sampling of the Biaxial RNNs, there is very little danger of overfitting any of the networks, since there the amount of valid training samples is vast. This is the reason why the model would ideally run for many more epochs and probably produce better results.

Since the composer component was never implemented, in addition to lacking a simple way to select the most probable, the result is not of the expected format. Even so, it can potentially provide some insight into how the networks learn over time, both in terms of placing notes, and which notes can be played in sequence for each individual track.

Other factors have probably affected the generation and learning processes. One of factor could be the advanced compositional techniques discussed in Section 2.2.4. Another factor could be that many songs in the dataset do not use all channels, which could be misleading; both to the time axis network and note axis network.

Finally, the topic of evaluating results within the field of music generation should be discussed. There is no way to compute how good a piece of music is, since it is a subjective experience. Ideally, the best way to evaluate generated music is through the use of a Turing test [10], where a large quantity are surveyed. Each person could for example be given two songs; one generated, and one composed by a human. The goal would be to get the people to a success rate of identifying a generated song as close to 50% as possible. Such a result would indicate that participants in the survey are unable to tell the difference. The problem is, however, costly both in terms of time and resources. Because of this, most researchers opt for providing their own subjective opinion, as well as providing the generated music to the reader so they may form their own opinion.

5.2 Architecture

As mentioned in Section 3.3, the proposed architecture should produce listenable sequences given enough training time. Despite this, there are still some disadvantages to the proposed architecture.

The first problem is a reduction in output complexity that may be undesirable for many applications. This situation is caused by the composer component, which picks a single note for each instrument. For the dataset used in this project, this is no issue, since a channel may only play a single note at any given time. However, if generating music of another genre, like rock, which can have multiple instruments able to play more than one note at a time. The composer could possibly be rewritten to accommodate this difference, or a more flexible method could replace it.

In regards to alternate architectures, there are multiple possible solutions that may be better or worse than the solution proposed in this thesis. Among them are the possibility of extending the Biaxial RNN to support multi-instrument generation, and using one or more GANs in some configuration.

Currently, the main issue with using a single Biaxial RNN for multi-instrument music generation is the flattening of all instrument tracks into one. This process makes it impossible to determine which note is played by which instrument. It should be possible to maintain the mapping between instruments and their notes. However, this would likely require a fundamental shift in the implementation to also take notes played by other instruments into account, both while training and generating novel music.

Another architecture that may be able to generate a good score would be a solution involving one or more GANs. A GAN network is primarily focused on content generation; hence, the generation part of its name. Such a network consists of two networks; a generator, and a discriminator. Both networks are trained on the same dataset. The goal of the discriminator is to judge whether or not some input is similar to the dataset. On the other hand, the goal of the generator is to generate content that tricks the discriminator judges to be real. This process will eventually train the generator to create convincing content. By training such a network on music, it will therefore be able to generate realistic compositions; given time and data. As mentioned in Section 2.5, an implementation for generating music through a GAN was proposed by Olof Mogren [13]. However, at the time of writing, there does not seem to be any architectures for generating multi-instrumental music. Such an implementation could either take inspiration from the solution presented in this thesis, or it could involve creating a larger GAN designed for training on, and generating music for multiple instruments.

5.3 Implementation

As mentioned earlier, last-minute problems with the implementation caused a shift to extend an existing implementation for a Biaxial RNN, rather than porting it over. This was caused by a lack of experience with the machine learning library Theano [19], as well as with the low-level API for Tensorflow [20].

In order to achieve partial results, it was decided to drop the composer component from the implementation, and focus on implementing the collection of Biaxial RNN instances. The Biaxial RNN implementation chosen was written by Yoann Ponti[1], which is in turn based on the original implementation by Daniel Johnson[2]. The extended implementation chosen has a more robust structure, as well as easier configuration from the command line.

To make the project compatible with multi-instrument many aspects had to be changed, such as parsing and storing the state matrices for all channels in a song, creating multiple Biaxial RNN instances, training them, and generating songs with all instances at once.

[1]Improved Biaxial RNN implementation by Yoann Ponti: `https://github.com/onanypoint/epfl-semester-project-biaxialnn`

[2]Original Biaxial RNN implementation by Daniel Johnson: `https://github.com/hexahedria/biaxial-rnn-music-composition`

In hindsight, the best choice for the implementation would have been to learn Theano and extend the existing implementation. If this option was chosen, the composer component would probably also have been finished, and the entire solution could be tested.

Chapter 6

Conclusion and Future Directions

Multi-instrument music generation is an area in an early developmental stage. This thesis has attempted to explore this new field by further developing existing concepts, and connecting them in a novel structure. Although the implementation was not fully successful in reaching the proposed solution, and unforeseen problems left little time for training the resulting model, there were still interesting results.

The implementation was applied to a unique dataset that has not been explored to this extent before. Because of the constrained compositional environment of the NES, the dataset strikes a balance between complexity and simplicity; due to advanced compositional techniques, and each instrument only being able to play a single note at any point in time.

Although the results were less than satisfactory, there is a lot of potential in further developing both the solution, and especially the implementation. The highest priority for the future would be to implement the composer component from the proposed solution, thereby allowing the model to generate real music. It would also be interesting to compare the subjective quality of music generated by four independent Biaxial neural networks to a model with the composer component; to see if it makes a difference.

Another future development would be to create a working port to a more modern framework, since Theano is no longer under development. This would likely increase performance to make both training and music generation faster.

There is currently a lack of multi-instrumental music generation architectures, therefore, a possible future direction to take this topic would be to create another novel architecture, either one of the possibilities discussed in Section 5.3, or something entirely different.

There is also the possibility to use a different dataset, thereby possibly requiring a solution more suited to the chosen dataset.

Finally, there is the direction of making the work more consistent and configurable by implementing some of the features from other related works, like some of the features discussed in Section 2.2. An example would be to integrate the RL Tuner to produce a more musically strict result. There are also many other possible directions, like implementing the project using Magenta [21]. Magenta would enable interactive applications, where the user can create a starting point for a generated song, or edit the generated music.

This thesis has hopefully shed some light on an area within music generation that is rarely explored. The fundamental tools within deep learning should allow for some innovation within such an underdeveloped topic.

List of Figures

List of Tables

Bibliography

[1] Jordan Dee. Pulse Width Modulation. URL https://learn.sparkfun.com/tutorials/pulse-width-modulation/duty-cycle.

[2] Glenn Dubois. The More You Know: The FAMICOM/-NES/2A03 Triangle Channel. URL https://chiptuneswin.com/blog/the-more-you-know-the-famicomnes2a03-triangle-channel.

[3] Chris Donahue, Huanru Henry Mao, and Julian McAuley. The NES Music Database: A multi-instrumental dataset with expressive performance attributes. In *ISMIR*, 2018.

[4] Haroon Piracha. Inverse Phase's Music From Old Sound Chips. URL http://www.originalsoundversion.com/inverse-phases-music-from-old-sound-chips.

[5] Christopher J Hopkins. *Chiptune music: An exploration of compositional techniques as found in Sunsoft Games for the Nintendo Entertainment System and Famicom from 1988-1992*. PhD thesis, 2015.

[6] Walt Crawford. MIDI and Wave: Coping with the language. *Online*, 20(1):86–87, 1996.

[7] Zachary C. Lipton, John Berkowitz, and Charles Elkan. A critical review of recurrent neural networks for sequence learning, 2015.

[8] Skymind. A Beginner's Guide to LSTMs and Recurrent Neural Networks. URL https://skymind.ai/wiki/lstm.

[9] Nvidia. Long Short-Term Memory (LSTM). URL https://developer.nvidia.com/discover/lstm.

[10] Jean-Pierre Briot, Gaëtan Hadjeres, and François Pachet. Deep Learning Techniques for Music Generation - A Survey. *CoRR*, abs/1709.01620, 2017. URL http://arxiv.org/abs/1709.01620.

[11] Ian Simon and Sageev Oore. Performance RNN: Generating Music with Expressive Timing and Dynamics. https://magenta.tensorflow.org/performance-rnn, 2017.

[12] Qi Lyu, Zhiyong Wu, Jun Zhu, and Helen Meng. Modelling high-dimensional sequences with lstm-rtrbm: Application to polyphonic music generation. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[13] Olof Mogren. C-rnn-gan: Continuous recurrent neural networks with adversarial training. *arXiv preprint arXiv:1611.09904*, 2016.

[14] Natasha Jaques, Shixiang Gu, Richard E Turner, and Douglas Eck. Tuning recurrent neural networks with reinforcement learning. 2017.

[15] Gaëtan Hadjeres, François Pachet, and Frank Nielsen. Deepbach: a steerable model for bach chorales generation. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1362–1371. JMLR. org, 2017.

[16] Daniel D. Johnson. Generating polyphonic music using tied parallel networks. In João Correia, Vic Ciesielski, and Antonios Liapis, editors, *Computational Intelligence in Music, Sound, Art and Design*, pages 128–143, Cham, 2017. Springer International Publishing. ISBN 978-3-319-55750-2.

[17] Daniel Johnson. Composing Music With Recurrent Neural Networks. http://www.hexahedria.com/2015/08/03/composing-music-with-recurrent-neural-networks, 2015.

[18] Google. Cloud Computing Services | Google Cloud. https://cloud.google.com, 2019.

[19] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL http://arxiv.org/abs/1605.02688.

[20] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/. Software available from tensorflow.org.

[21] Google AI. Make Music and Art Using Machine Learning. https://magenta.tensorflow.org, 2019.