Spring 2016

Algorithmic Music Composition and Accompaniment Using Neural Networks

Daniel Wilton Risdon
Bard College

Algorithmic Music Composition and Accompaniment Using Neural Networks

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Daniel Risdon

Annandale-on-Hudson, New York
May, 2016

Abstract

The goal of this project was to use neural networks as a tool for live music performance. Specifically, the intention was to adapt a preexisting neural network code library to work in Max, a visual programming language commonly used to create instruments and effects for electronic music and audio processing. This was done using ConvNetJS, a JavaScript library created by Andrej Karpathy.

Several neural network models were trained using a range of different training data, including music from various genres. The resulting neural network-based instruments were used to play brief pieces of music, which they used as input to create unique musical output.

Max, while useful for live performance and audio processing, proved to be somewhat impractical for this project. Implementing too complex of a network caused performance issues and even crashing. Because of this, smaller networks, which are less robust in their prediction abilities had to be used, producing very simplistic musical patterns.

Contents

Dedication

For Kirby.

Acknowledgments

I wish to express my sincere thanks to Keith O'Hara, my advisor in computer science, and Bob Bielecki, my advisor in music, for all of their help on this project. Without their insight and guidance, none of this would have been possible.

I would also like to thank Sven Anderson for answering my many questions and for all of his assistance.

Thank you as well to all of the friends who helped to make the performance portion of this project a reality, particularly Marcel Rudin, John Carroll, and Pippa Kelmenson, my ensemble, as well as Eileen Goodrich and Andrew Feyer for their technical support.

Finally, I would like to thank my parents, Ron and Jane, my siblings, Matt and Caroline, and my dogs, Kirby and Casper, for all of their love and support.

1 Introduction

1.1 Background

Many varieties of music either include or even completely revolve around the concept of using a learned repertoire of skills and knowledge to create unique sounds right there on the spot. This has been an important part of my own music, whether that be jazz, experimental electronic music, or heavy metal. When I compose music, I leave space for improvisation. That improvisation may take the form of a guitar solo, a brand new electronic piece composed on the spot, or even a new riff simultaneously improvised by an entire band.

All of this comes back to a learned set of rhythms and melodies and sounds that a musician can call on to instantly create something new. This is, although a musical skill that takes time to learn, something that can be recreated with machine learning. Sequencers are not a new thing to the world of music. While in many cases, the parameters from which they create sequences of notes are already predefined, they still to some extent represent this idea of an easily called-upon catalogue of patterns that can be used to create music right as it is being performed. This is in some way where the idea for this project came from.

I have previously worked on other projects involving music software. One such project was a chord sequencer built in Max, a graphical programming language commonly used for audio and MIDI processing. It had the option to always start a measure with the first scale degree, allowed for the selection of the root note and quality of the key, and would always finish off a phrase with a cadence of some kind. This was not a project based in machine learning, but it still created new, albeit very simplistic, music on the spot from a set of known parameters. The only thing missing was the program actually learning these parameters itself instead of having them hard-coded in.

Machine learning is not completely new to electronic music. Neural networks have been used in the past to compose music, among many other things. Even on a more basic level, most digital audio workspaces (DAW) have some sort of sampling ability, which in a way learns a new sound, and is then able to apply it over an entire keyboard, allowing it to be played at a range of different pitches. What is less common is the use of machine learning for live performance, more specifically simultaneously with a human musician.

The goal of this project is to create a digital instrument capable of learning patterns of improvisation from a selected range of musical input, and then using that new knowledge to play itself live, based on the playing of a human musician. This is an important concept because improvisation requires listening just as much as it requires playing. This must be a program that not only learns patterns of playing, but also uses them to accompany another player without fighting for room.

There have been many different examples of algorithmic music, both learning-based and otherwise in the past. Music has been automated long before computers were conceived. Among the earliest examples is the player piano, which uses piano roll, which is in essence, a long sheet of paper with holes punched in it representing an ordering of notes to be played. This bears a very strong similarity to the modern day MIDI, which will be explored at length later on. This was essentially an early model of what is now called a sequencer. A sequencer, as its name suggests, primarily serves to repeat a sequence of notes. These notes are not typically learned, but given directly to the sequencer to repeat until told otherwise. While some electronic keyboards have built-in digital sequencers, the sequencing important to us is that done by software.

Modern music software has numerous examples of sequencing. It is a common feature in many DAW's and also exists in the form of many different plugins for those DAW's. Software sequencing commonly uses MIDI (musical instrument digital interface). MIDI messages carry with them specific pieces of data, ranging from pitch and velocity to more complicated parameters such as pitch bending. Being represented by simple integers makes processing MIDI considerably easier. For example, middle C has a MIDI pitch of 60. Velocity corresponds to the force with which a note is struck. A value of 0 signifies the note being turned off. Similarly to the way piano roll doesn't do anything without the player piano, MIDI needs a digital instrument to function. MIDI files and real time streams of MIDI messages are just a series of notes to then be played by whatever instrument is processing them.

This has its advantages and disadvantages. The numbers are easy to work with and the files don't take up much space, but the limits of having primarily a simple note on and note off system means the audio being produced lacks the natural articulation of a real musician's hand. This isn't as much of an issue when used for synthesizer music, but MIDI instruments meant to sound like string or horn instruments often sound particularly mechanical, due to the inability to imitate inflections such as vibrato and sliding.

Sequencing has its specific applications, but it should not be mistaken for real composition. Algorithmic composition is something more complicated. It requires algorithms capable of determining what notes come next depending one various parameters such as key, meter, position in the measure, musical style, and more. This is where machine learning comes into play. Neural networks have been used for analyzing and creating music as far back as the 1980s [1]. These earlier applications of neural networks to algorithmic composition consisted of, at their simplest forms, prediction of a note based on the note preceding it. Networks using this model, while producing strings of notes that made sense together, generally lacked an overall understanding of musical structure [7]. This is due to the fact that music consists of quite a bit more than just notes following other notes. A good piece of music has some kind of progression and structure over the course of its duration, made up of various elements such as key changes, melodic themes, and dynamics.

Despite the disadvantages of this note-by-note system, it is actually the system I choose to employ, due to the nature of this instrument as an accompaniment, not a solo composer. Those vital parameters will be determined during the performance by the musician in charge. Ultimately, the goal of this project is to create a machine learning-based software sequencer capable of being trained quickly with varying data based on desired use. This ideally means a compact, simple neural network model that can be trained with small datasets (even a single song) for specialized performance uses.

1.2 Music Software

There are two very different options for audio input and output that have been previously used for neural network-based composition. The first and more complex option is raw audio. Through some form of pitch detection, or analysis of the audio waveforms themselves, the training data for the network could be obtained. The simpler option is MIDI, due to the fact that the parameters MIDI signals are represented by integers.

In addition to the different input and output options, there were also many possible software environments in which this project could be implemented. The two primary options for implementation were either a VST plugin for Ableton Live, or a patch made in Max. VST is the standard format for audio plugins for most music software. Ableton Live is a DAW (digital audio workstation) commonly used for live performance due to its unique layout. A VST plugin could be anything from a delay effect to a completely new digital instrument. The idea with this project would be to make a plugin instrument that created its own MIDI signals in addition to those being sent by the musician.

The first advantage of using Ableton Live is that it is already my primary tool in composing and performing electronic music. Live is set up in such a way that makes composing and performing using the same software, and even the same file, rather easy. This is because of its arrangement and session interfaces. The arrangement view is more like your standard DAW. It has audio and MIDI tracks represented on the y-axis over time on the x-axis. These tracks play in the order they are placed on the arrangement.

The session view is what is unique about Live. It allows the user to represent tracks on a grid that does not notate time. Columns can represent a single MIDI instrument

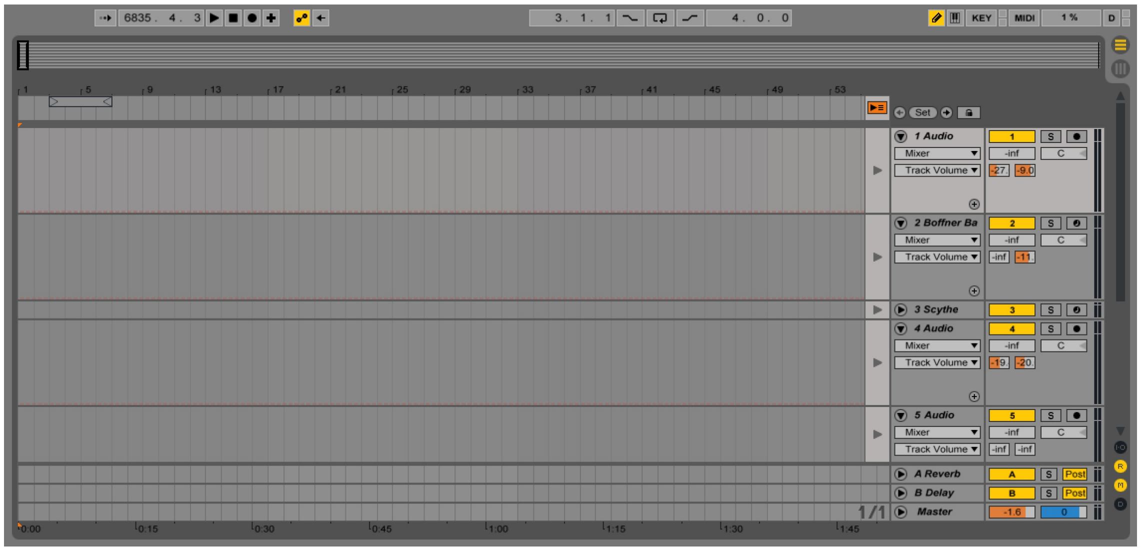

Figure 1.2.1. The Ableton Live Arrangement Window

or synth, with each item within it representing a clip, as well as a collection of various audio clips. These can be mapped to either the computer keyboard or whatever MIDI controller is currently connected. This allows for both the playing of a synth or MIDI instrument, or the triggering of MIDI and audio clips, which is very useful in a live setting, and makes it easily possible to control an electronic component of a piece of music while also playing a live instrument.

Due to all of this, building the software as a plugin for Ableton Live was an ideal option. This VST plugin would be an instrument completely integrated into the software I already use for composition and performance. Unfortunately, a VST plugin would need to be built using a complicated and sparsely documented SDK, in order to be compatible with a DAW.

VSTs are written in C++, and while handling MIDI numbers through this library turned out to be fairly doable, the first major hurdle came in the form of the realization that Ableton Live would not allow MIDI signals to be created by the plugin

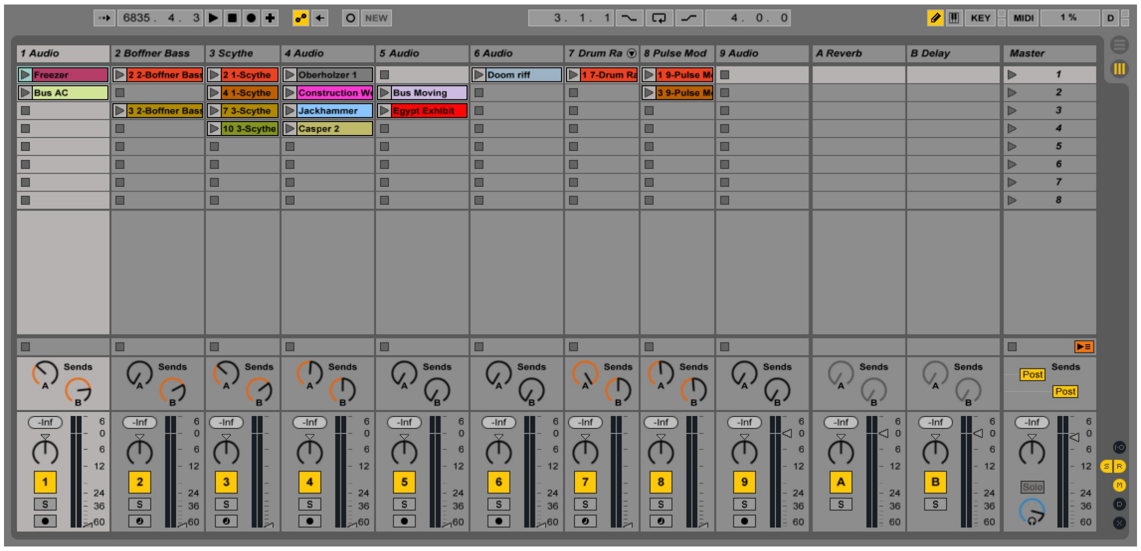

Figure 1.2.2. The Ableton Live Session Window

and then sent to another instrument. A synthesizer would have to be built into the plugin. This would have greatly increased the development time of this project.

1.3 Max

Max is a visual programming language, very commonly used for audio and MIDI processing. Max is created by a company called Cycling '74, and was originally written in the mid-1980's by Miller Puckette [10]. It is unique in its flexibility for audio and video processing. While dedicated synth programs might be more computationally efficient, they provide a more limited range of sounds and functions, specific to their chosen purpose. Max, however, allows a user (after somewhat of a learning curve) to create completely unique software instruments and other programs.

Programming in Max consists mostly of placing objects that perform particular functions, and sending data between them by connecting wires, similarly to other visual languages such as Simulink and Reaktor. The ports on the top of an object are the inputs, and those on the bottom are the outputs. A simple example would be two integer objects having their outputs fed into an object that performs some arithmetic on them and then prints the output to a message box (fig. 1.3.1). Max includes a massive library of objects, ranging from simple data types and operations to more complex functions such as splitting raw MIDI input into its separate components. A program made in Max is typically referred to as a “patch” or a “patcher”. I will be using the term “patch” in this paper.

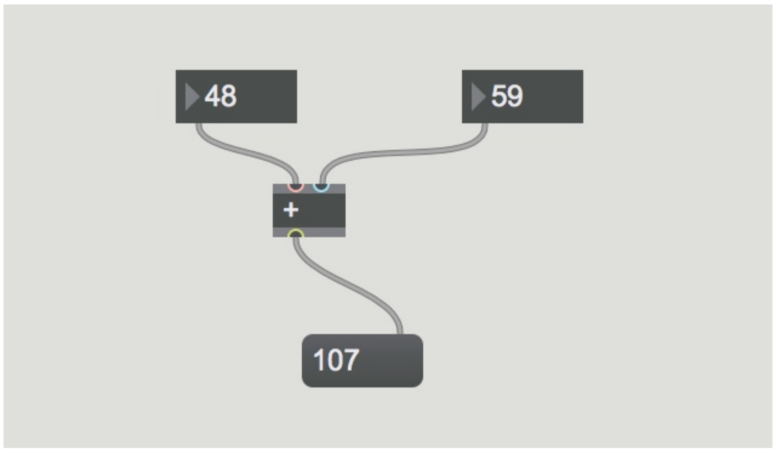

Figure 1.3.1. A Simple Example of Max Programming

Due to the unique nature of Max, it is possible and fairly simple to run multiple processes at the same time. For example, MIDI input from a controller or keyboard can be sent into two separate wires, one straight into a MIDI output object, and one into some sort of effect, before also being sent to an output. This is an important feature for this project, as it allows a musician to play notes in real time, and simultaneously send them to the neural network object to be processed there.

Max also allows for the creation of new objects. This can be done either by creating a sub-patch, which is essentially just another Max file contained within an object, or by using JavaScript. JavaScript files can be directly loaded into and compiled by Max’s `js` object. Once loaded and compiled, the files, or functions within, can be called using message objects. It is through this functionality that this project is made possible. While neural networks and other forms of machine learning have been implemented in Max before, the JavaScript functionality allows for more advance neural network libraries to be placed directly into a Max patch with some amount of modification. This implementation of neural networks into software more predisposed to live performance is at the center of this project.

Max patches can be run in two different modes. The first is patching mode, which displays all objects and wires in the patch, either for editing, or simply because user-friendly operation isn't the goal. The second mode, presentation mode, displays only the objects one wishes a user to see. This allows for fairly easy creation of a GUI. This project does not currently have a presentation mode, but as the patch is improved upon, one may be added.

1.4 Neural Networks

Neural networks are powerful machine learning tools which, once trained on large datasets, can predict patterns in subsequent data inputs. This can potentially have many different applications in solving many different problems, or even creating art. A neural network consists of an arrangement of nodes, or *neurons*, each of which performs a weighted sum of a set of inputs and then squashes the result between 0 and 1 using an activation function and adds a constant bias value. These nodes are arranged into layers, with nodes within a certain layer taking the output of the previous layer as their input. Each layer shares the same input among all of its nodes, but the nodes within will have different weights and biases. The layers performing theses weighted sums are called *hidden layers*, and are sandwiched between input and output layers.

The input layer determines the size of the input to the network, and simply takes it and sends it on to the hidden layers. The output layer can take different forms

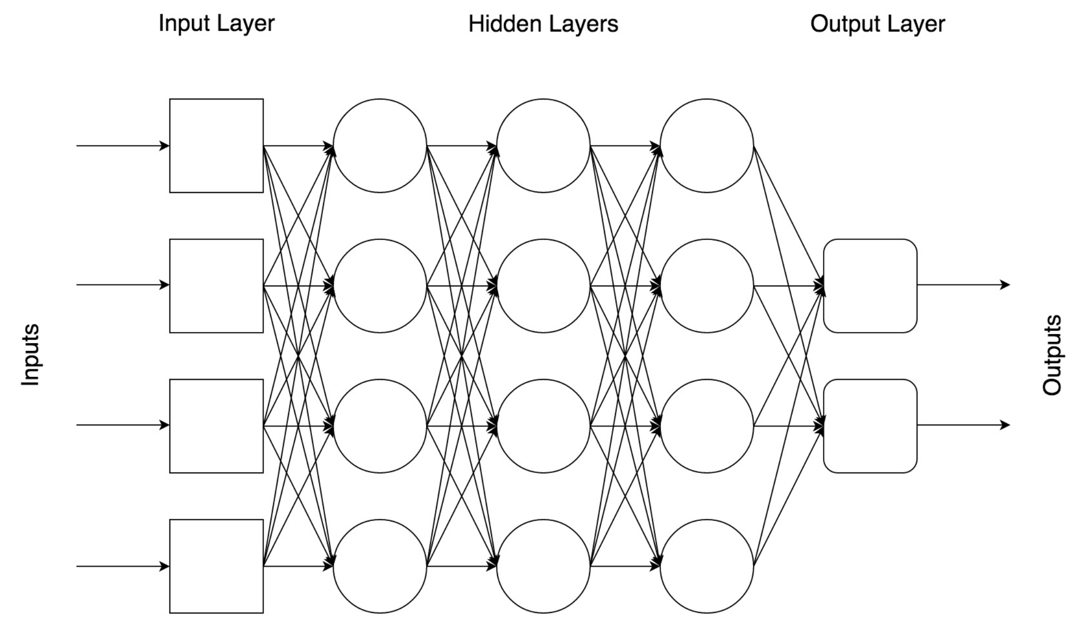

Figure 1.4.1. An Example of a Neural Network for Binary Classification

depending on the intended application of the network. Using regression, the network can predict real number output values. In this project however, a classification layer has been used. With a classification-based output layer, the network determines the probability of a predetermined number of classes. The classes in this case are the 128 possible MIDI notes. These are represented by 128 nodes in the output layer, each with their own biases and weights. On top of these differences, there is also the distinction between feedforward and recurrent neural networks.

The type of network used in this project is a feedforward network. This takes the inputs and sends it through each layer linearly, not saving outputs between training steps. A recurrent network carries the output of each layer over to the next step as an additional input, allowing for a stronger understanding of patterns in time

[9]. Time is not as much of a factor in this application, so a feedforward network is enough to choose a note based on the recent notes played by the musician.

In order for a neural network to effectively make these predictions, it must first be trained. This involves giving it very large input sets and updating the weights of each neuron after a given number of steps. This process is called *backpropagation.* With each step, a cost is calculated, which in this case would be related to the probability of the correct note being predicted. This cost is used to calculate a gradient for each neuron. An optimization function is then used to adjust each neuron's weights in order to reduce this weight. The optimization function used for this project is *stochastic gradient descent* (SGD), which performs weight adjustment after a designated number of training steps, called the *batch size* (in this case a batch of 10). This is done over the course of thousands of potential inputs and expected outputs, resulting in more and more accurate data prediction as the training goes along.

Several other parameters in the network model also determine the effectiveness of training. *Learning rate*, which is chosen manually in an SGD model, determines how large the adjustments to the network's weights will be each training step. Several other optimization functions choose the learning rate dynamically during training. A dropout probability can also be applied, which gives each neuron's output a chance to be set to 0 at each training step, for the sake of regularizing the outputs of the network to avoid overfitting more common patterns in the training data.

1.5 ConvNetJS

Part of the goal of this project was to streamline the process of implementing neural networks into live performance by taking advantage of Max's compatibility with JavaScript, and implementing an already existing neural net library. This library is *ConvNetJS*, a JavaScript neural network library created by Andrej Karpathy, a PhD student at Stanford [11]. This library is intended for browser use or use with node.js, so implementing it into Max requires some slight modifications.

Within ConvNetJS, it is fairly simple to set up and train various neural network configurations (example in listing 1.1), and the documentation available provides information that makes this functionality accessible to those with little knowledge of neural networks or machine learning. For those with some knowledge of how neural networks function, it is fairly simple to configure a network. Individual layers are defined, starting with input, then hidden layers, then output, and then pushed into a list from which the network will be initialized. When defining a layer, depending on the type, one can set the parameters, such as input size, number of neurons in a hidden layer, and output size/type.

```

var layer_defs = []; // array of layers
layer_defs.push({type:'input', out_sx:1, out_sy:1, out_depth:1}); // input layer-
    1 input
layer_defs.push({type:'fc', num_neurons:10, activation:'sigmoid'}); // hidden
    layer - 10 neurons, sigmoid activation
layer_defs.push({type:'softmax', num_classes:2}); // output layer - 2 possible
    output classes
var net = new convnetjs.Net(); // empty network
net.makeLayers(layer_defs); // create layers within network
```

Listing 1.1. Initializing a Neural Network With ConvNetJS

Once a network has been initialized, a trainer can be created. The parameters of this trainer class include the optimization method, followed by things like the learning rate, batch size, and dropout (if any is desired). This trainer will then take an input, forward the network, and update its weights based on loss. Once trained, the network can be used for prediction. Saving the network is done using JSON objects. These objects can be turned into strings and then saved manually for later use.

2
Related Work

Neural networks have seen many previous applications in music composition. An early example is Bharucha and Todd (1989). Their research revolved around the idea of a separation of *schematic* and *veridical* expectancies. Expectancies refer to how a person expects a piece of music to play out based on their own knowledge of music and their own culture. Schematic expectancies have to do with cultural understandings of musical style. Veridical expectancies are more context-specific, and have to do with how a piece might progress through a certain sequence. The agreements and conflicts between these two expectations play a prominent part in the way any piece of music is listened to.

According to their research, it is fairly common for someone to have a strong understanding of common musical ideas in their own culture, and even in those lacking in formal music training, and understanding of the sensible progression of a piece based on current context. These understandings are very much learned, as people expect different things based on the tonal systems present in their cultures. Western music has a very different system of tonality from that of India, for example.

Bharucha and Todd modeled these expectancies using a neural network they called MUSACT. The resulting conclusion was that despite the environments used to train the neural networks being largely based in veridical expectancies, schematic concepts were still learned through passive exposure. This network, however, did not actually create new music, just predict it based on these expectancies.

Mozer (1994) used neural nets to incorporate not only the probability of the next note being played, but also an understanding of note duration and harmony. He did this using recurrent neural networks as well as in-depth representations of chords and key changes. The result was a network called CONCERT. This software had a training mode and a composition mode. In composition mode, it would be first seeded with a sequence of notes, from which it would be able to generate more without any further external input, due to its recurrent nature.

CONCERT used 49 possible notes (four octaves), uniquely represented by a helix (presented in the form of an activity vector). He calls this PHCCCF representation (pitch class, chroma circle, circle of fifths). This is due to the fact that notes an octave apart will be directly above or below one another in this helix, and notes near each other in the circle of fifths, or simply chromatically, will have fairly similar representations. This complex note representation is layered with representations for chords and time as well, resulting in a robust recurrent network. It was able to master a C major scale within about 30 training passes, as well as a training set of 28 different scales in about 55. It was even capable of recreating larger song forms such as AABA.

Feiten and Günzel (1994) used a neural network to address the difficulty of finding desired sounds in a database. Online databases of sounds are often unsorted, or sorted by types that can be highly subjective, making it difficult to find what one might be looking for. They did this using a fairly complex representation of sounds in a 3-dimensional space, that the user can apply their own definitions to, to allow for far greater subjectivity.

Their algorithm was thorough, but the computation time was very costly, and so the practicality of the solution is questionable. On synthetic sounds, however, the sound space is static, as there are a set number of parameters, which allowed for the computation time to be largely ignored.

Eck and Schmidhuber (2002) looked at how previous uses of recurrent neural networks had failed in learning larger musical forms, and built a long short-term memory (LSTM) model to try to succeed where others had failed. An LSTM model can remember certain values for longer periods of time than traditional recurrent networks. They successfully trained this model to produce interesting blues melodies.

Eldridge (2006) did not exactly use neural networks, but still developed a machine learning-based Max patch intended for algorithmic live performance in unison with a live musician. This "self-karaoke" machine takes samples of what an instrumentalist is playing and manipulates them in real time for interesting results, despite not actually learning formal patterns of music. The general idea behind this software is an important basis for my own: human performance accompanied in real time by the software's response.

The deeper details of Eldridge's project involve a physics simulation describing the motion of particles, and an implementation of what is called an Ashbian homeostat. She did this using a system somewhat similar to a neural network, in which a series of interconnected nodes have the weights connecting them adjusted whenever they exceed a particular threshold. This system is then used to control the parameters of a granular synthesizer.

Smith and Garnett (2012) used several variations on hierarchical neural networks to generate algorithmic music with a greater understanding of long-term musical

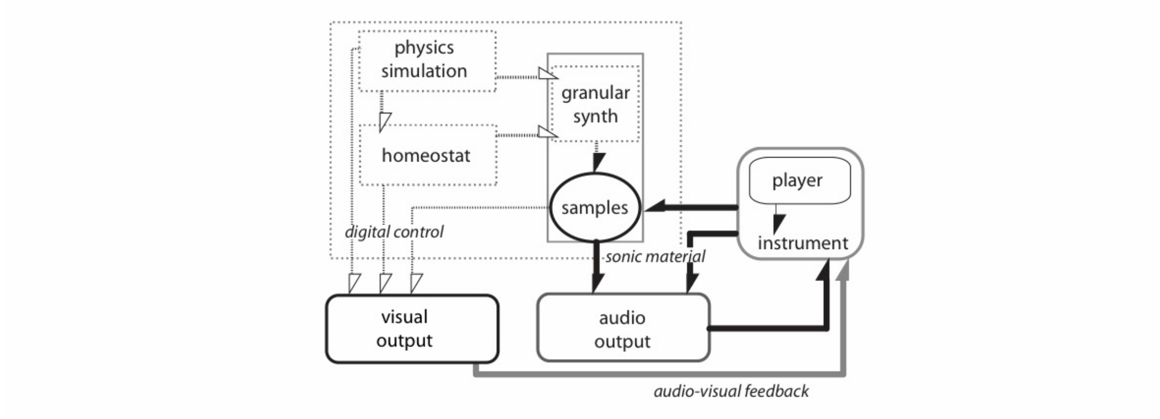

Figure 2.0.1. Eldrige's Self-Karaoke Machine Schematic

structure and key. Where simple neural networks, and to a lesser extent, recurrent networks tend to create music with a lack of overall structure, their model was able to compose pieces with some higher-level musical elements, including repeating patterns and recognizable sections.

Their model operates over a 4 octave range, using the western 12-note system. Instead of the note-by-note method used by many other systems, theirs looks at the piece more as a whole, and considers chunks of music at a time in order to determine what can be called a "novel" musical idea. This hierarchical model looked at these small novel events, and then further abstracted them into higher and higher concepts of musical structure. The main levels of structure displayed by their results are pattern repetition and manipulation.

Nayebi and Vitelli (2015) compared two different types of recurrent networks and their usefulness in generating algorithmic music: gated recurrent unit (GRU) and long short-term memory (LSTM). These models took mono WAV audio files as input and produced waveforms as output based on an initial seed. The results from the GRU model were largely white noise, and thus not musically viable by the standards they set, but the LSTM model was able to generate viable outputs. Due to time constraints and the immense computation time required for processing WAV files, they were unable to test on the wide range of genres they had initially planned.

Johnson (2015) used a biaxial recurrent neural network model that used MIDI as input and output. His model used a fairly complex vector to represent the MIDI input which included notations of the integer MIDI value, its pitch class, vicinity to the previous note, context of the current note in relation to the frequency of recent notes, and position in a measure of 4/4 time. The output not only included the probability of a certain note being played next, but also a probability of it being articulated or held if it has already been played.

The model was trained on a large number of midi files obtained from an online database of classical piano, taken in batches of 10 random 8-measure snippets. The resulting compositions consist not only of single note melodies, but of chords and more sophisticated rhythmic ideas. The melodies can sometimes linger on certain notes for longer than desired, but otherwise sound much like real human-composed classical piano pieces.

3
Implementation

3.1 Development Process

This project went through several iterations before reaching a finalized form, each one incrementally adding the necessary functionality. The first of these was a simple proof-of-concept, taking MIDI input from a keyboard and saving the last 20 notes played to a list. It would then randomly select notes from this list to play each time the keyboard was played. This was a simple test leading up to the actual development of the complete Max patch.

The next step was to implement ConvNetJS into a `js` object. The primary modification needed was the removal of any code referring to `Node.js`. `Node.js` can be made compatible with Max, but is not by default, and is unnecessary for the purpose of this project. Due to JavaScript's nature as a non-modular language, the entire library of ConvNetJS must be copied into each `js` object in order for it to be used. The first incarnation of the patch had just one `js` object, but for the sake of modularity within the Max patch, this was increased to three.

PATCH VERSION	FEATURES
1	Basic MIDI keyboard support, sequencing via list of last 20 notes
2	ConvNetJS implementation, basic training using a single input set for testing
3	Training from MIDI keyboard, very inefficient due to constantly saving and reloading network
4	Global network and trainer objects, training from parsed array of notes, MIDI file support
Final	Separation of training and performance modes

Table 3.1.1. Versions of the Patch

Each iteration of the patch leading up to the final version added some important functionality. The first was MIDI input from a keyboard, followed by the porting of ConvNetJS, then later MIDI file inputs, as well as improvements to training efficiency and the ability to save and reload the parameters of the neural network. Table 3.1.1 lists each iteration of the patch and the features it added.

3.2 Final Max Patch

3.2.1 Overview

The finalized version of this Max patch is intended to run in two different modes: training and performance. Training mode takes MIDI files and parses them into a simple list of integers, before training the neural network using that list. Performance mode takes MIDI input from a controller of keyboard, and uses the most recently played notes to forward the neural network, and output its predicted note. This is done with Max's `gate` object, which takes an input and sends it out of a chosen output, essentially serving as a switch between the two modes so that the patch need only rely on a single input source. This overall structure essentially consists of a shared form of input, and a switch, deciding which part of the patch receives said input.

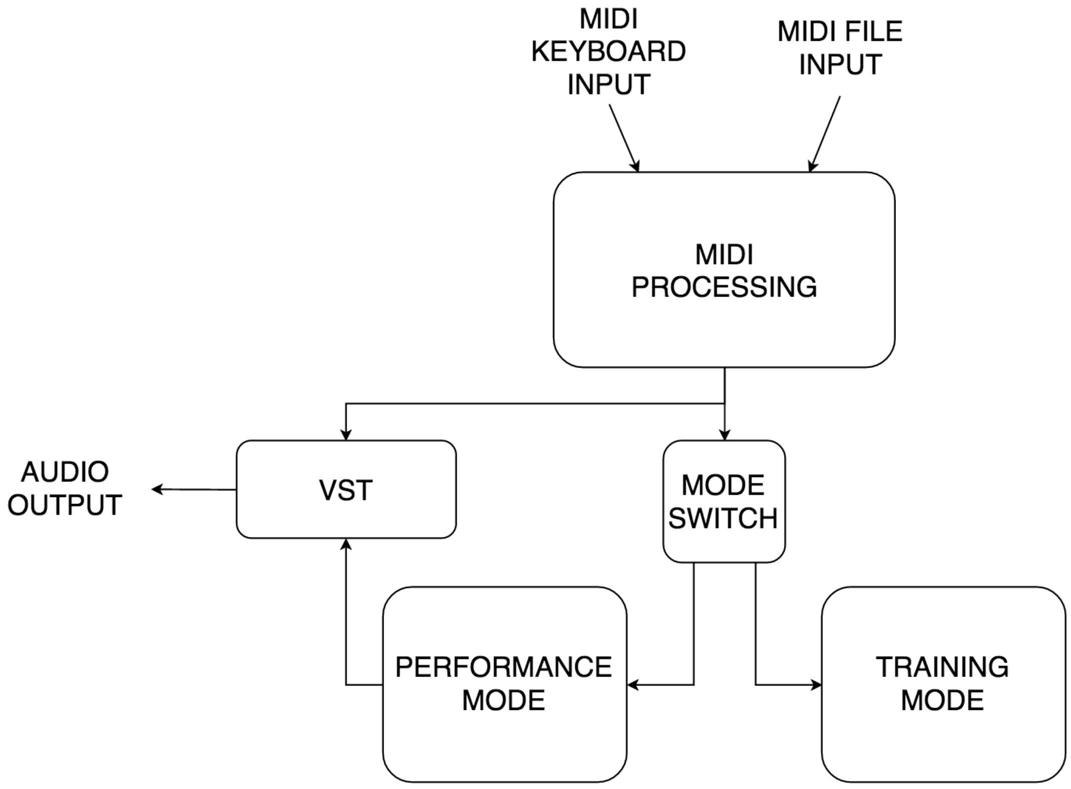

Figure 3.2.1. Overview of the Patch's Structure

3.2.2 MIDI Processing

The first important piece of this project is a basic framework for the input and output of MIDI signals. Max's `midiin` and `midiparse` objects are central to the input side of this purpose. `midiin` takes raw midi signals from a designated input and simply outputs them through a single port. `midiparse` then takes those raw signals and separates them into several important parameters, including pitch and velocity. This is necessary so that these signals can be formatted for output through a synth instrument, rather than the computer's default MIDI piano sound.

The audio output from this patch comes on the form of a VST plugin, as described earlier. VST's are a standard audio plugin format and are compatible with Max through the use of the `vst` object. This object allows a user to easily load up any VST they have installed. For this project, having the ability to change the voice of the instrument with ease is a useful feature. For testing I used a free synth plugin called Scythe for some fairly straightforward sounds.

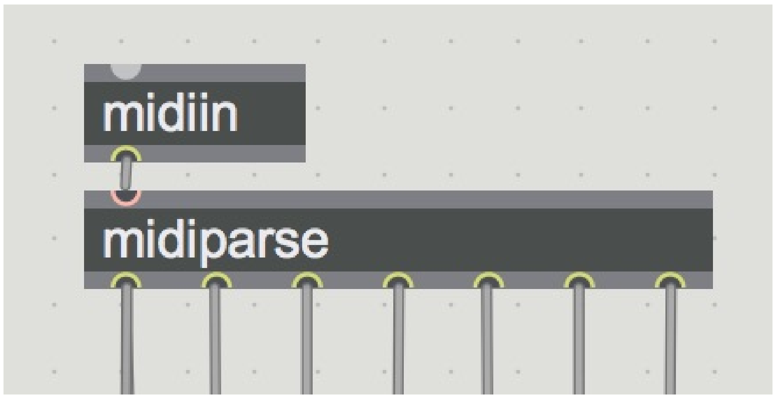

Figure 3.2.2. The `midiin` and `midiparse` objects

The other form of MIDI input (the training data) comes from MIDI song files that can be loaded into an object called `detonate`. MIDI files can be one of 2 types: 0 or 1. Type 0 files have all tracks combined into one, although the channel information is still present so it still has the effect of separate tracks. Type 1 files have each MIDI channel as a separate track. `detonate` takes each midi message and separates it into the important parameters such as pitch, velocity, and channel. This information can then be sent wherever it is needed.

Once the MIDI messages are "detonated," they are filtered by channel. Typically, drums are on channel 10, and drums are undesirable for this patch, since they use pitch numbers in a very different way, and could cause severe errors in training. Channel 10 and several other channels are thus blocked, so that the main melody of the piece can be used for training.

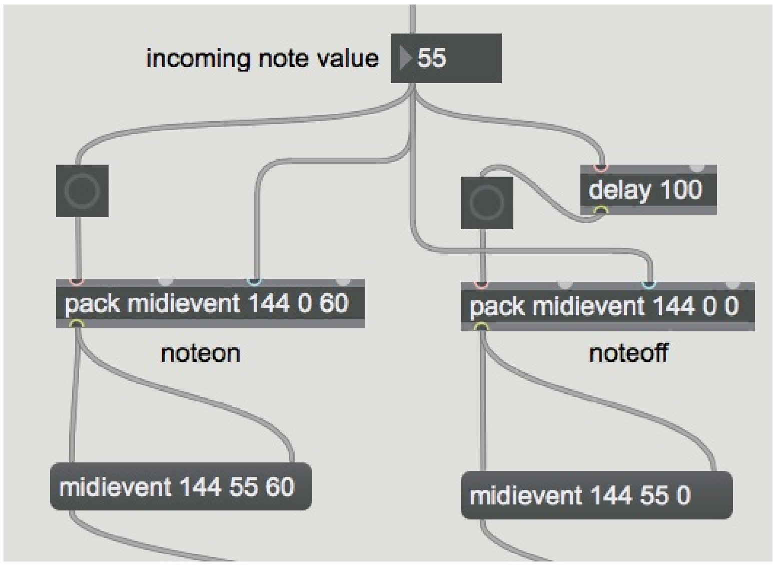

Figure 3.2.3. Generating `midievent` messages

3.2.3 Training Mode

The first step involved in training mode is to parse the MIDI song file into an array, note by note. This array, while potentially containing thousands of indexes, is still far more efficient for training than simply playing through the song repeatedly in real time. This array implementation is a part of one of the three `js` objects used in the patch. One of these `js` objects is for initializing the network, another is for training the network, and the last is for performing using the trained network.

This note-by-note parsing of the song is done without regard to actual rhythm. The song is played with each note being a set number of milliseconds apart (in this case, 10). This allows the song to be parsed much more quickly, without any detriment to the training data, since rhythm is not a factor to begin with. Note off signals are also ignored, so that training simply includes series of pitch values. MIDI files send a unique note message to indicate the end of the file, at which playback is stopped.

Once parsed, the array representation of the song is used as the training data. The `js` object used for training, `trainNet.js`, takes this input, and iterates through it, training on each note and the 7 preceding it, in order. The number of times the training set is iterated through is chosen by the user. Figure 3.2.4 depicts the overall structure of training mode.

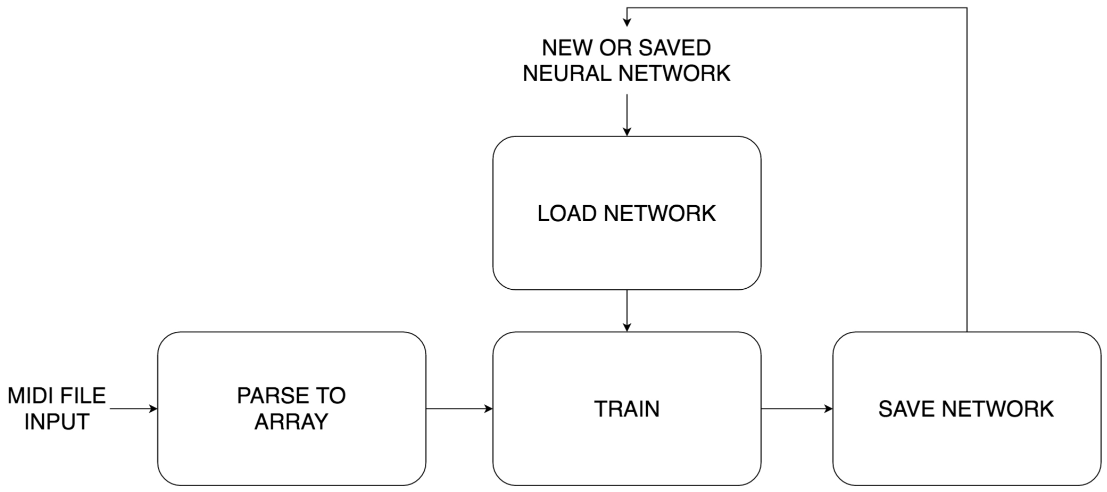

Figure 3.2.4. Training Mode

3.2.4 Performance Mode

In performance mode, MIDI input from a controller or keyboard is sent into a list-processing object called `zl`, which in this context, is used to save a stream of the last 7 notes played. This list is updated every time a new note is played. The `js` object involved in performance mode, `perform.js`, uses this list as its input in order to generate new notes. The main function of this object is called upon each click of a metronome, running at a tempo chosen by the user. This is the basis of playing music using this neural network-based instrument. The user choses a tempo before playing, and then plays a series of notes, while the neural network generates an accompanying melody in time with the metronome.

Because the patch ignores note off values in the input during performance, note off signals must be generated, or else notes will be sustained indefinitely. This is done by sending a bang through a ‘delay’ object, which 100 milliseconds later sends a bang signal to a note off message which is then used to turn off the note. While this is fairly simplistic, and does not allow for more interesting sustained notes of varying durations, it allows training to be simplified down to simple progressions of notes, which in turn allows the whole project to be distilled down to applying machine learning to a live note sequencing environment. Figure 3.2.5 shows the flow of data in performance mode, while 3.2.5 shows the Max implementation.

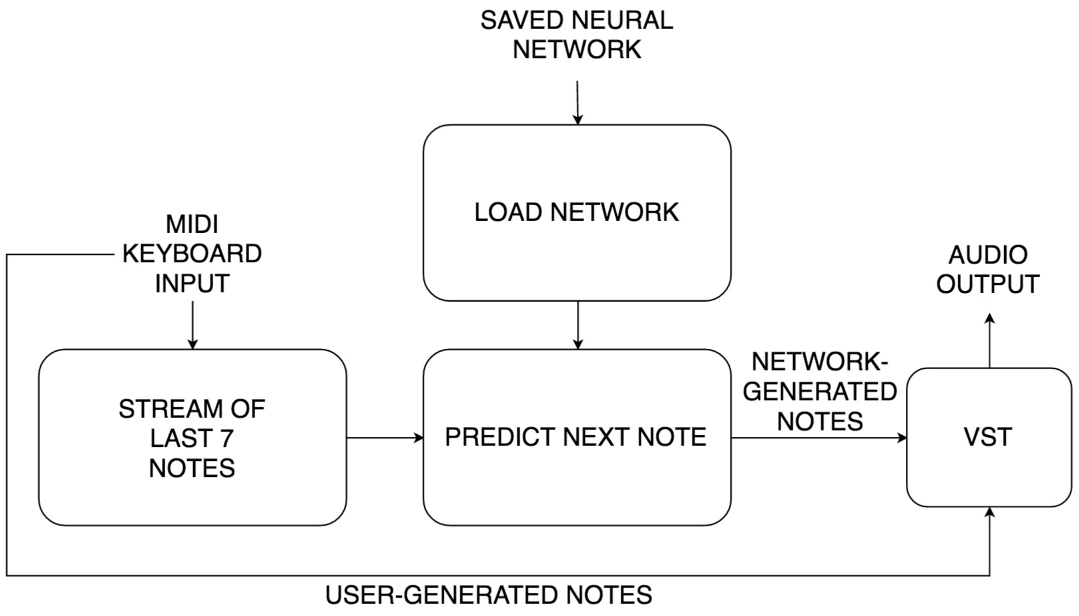

Figure 3.2.5. Performance Mode

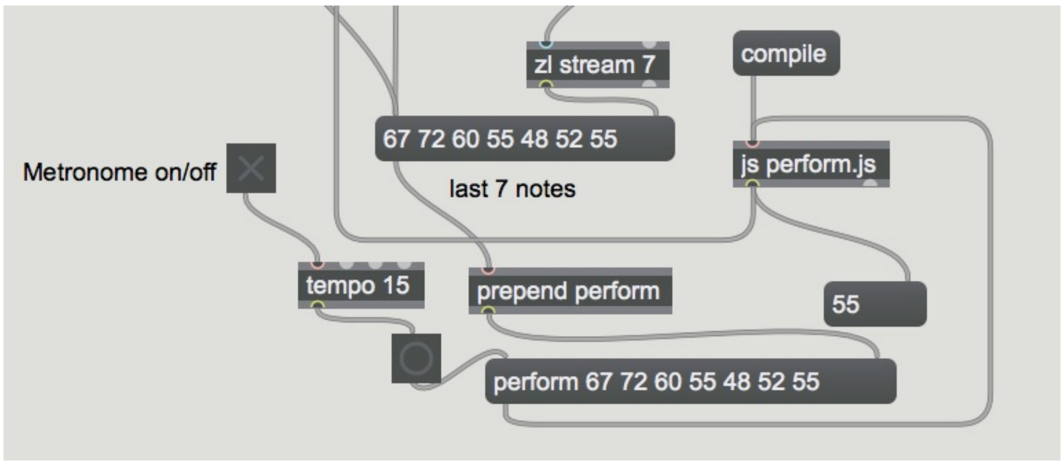

Figure 3.2.6. The Objects in Performance Mode

3.3 JavaScript Implementation

The JavaScript portion of the project has been described in general terms previously. There are three objects containing JavaScript code, intended for initialization, training, and performance. Each of these objects also contains ConvNetJS in its entirety. The functions within each file are called from message objects within the Max patch.

The first of these objects, for the purpose of initializing new networks and trainers, as well as loading and saving trained networks, is `defaultnet.js`. The first function within this file, `makeDefaultNet()` initializes the default network and a trainer. `saveNet()` converts an existing network to a JSON object and then to a string, which it outputs to a message object in the Max patch. `loadNet(jsonstring)` takes whatever string is present in the message box and converts it back to a JSON object to then be initialized into the global network object. This `js` object has a single outlet, for sending the JSON string to the message box.

The next object is `trainNet.js`, which is used for parsing the training data from a MIDI song file and then using it to train. The first function, `emptyArray()`, initializes the array to be filled with the training data. `addNote(x)` is then called every time the object receives a new note from a MIDI file or keyboard input, and appends that note to the array. `emptyArray()` can then be called again to clear everything from the training set. This functionality allows for training data to be easily interchangeable, enabling different network configurations to be created for different performance purposes.

The next important function is `trainLoop(times)`, which trains the network on the full array, step by step, a selected number of times. The notes in the training set are first standardized by being scaled between -0.5 and 0.5 to increase training efficiency and help prevent overfitting. This object also contains a few debugging functions and a `printSong()` function for making sure the notes have been successfully parsed into the array.

The third `js` object contains `perform.js`, which is used in performance mode. It contains the prediction functions for the network. There is a test prediction function meant for simpler networks, but there is otherwise only one important function in this object, `perform(n1, n2, n3, n4, n5, n6, n7)`, which feeds the last 7 notes played (which have been stored in the `zl` stream object in the patch) into the network for prediction. It then will loop through the possible classes of the network (0 - 127) and select the one with the highest probability to be played. It has some extra variables in place to prevent overplaying of the same note, as well as a random element to allow for the occasional decision to not play a certain note.

```

function perform(n1, n2, n3, n4, n5, n6, n7) {
/* Take 7 notes and predict the next. stand(n) is the standardization function to
    scale inputs between -0.5 and 0.5. chooseornot is either 0 or 1 and decides if
    a note will be played, assuming its probability would permit it. lastnote is a
    global object that maintains the last note played to prevent notes from being
    repeated over and over again.
*/
  var x = new convnetjs.Vol([stand(n1),stand(n2),stand(n3),
                stand(n4),stand(n5),stand(n6),stand(n7)]); // input set
  var predicted_values = globalNet.net.forward(x); // forward the network
  var note = 0; // note starts at 0
  var note_prob = predicted_values.w[0]; // note probability
  var chooseornot = 0;
  for (var i = 0; i < 128; i++) { // loop through all output nodes, if prob is
      higher than current, update
    if (predicted_values.w[i] > note_prob && i != lastnote.note) {
      chooseornot = Math.floor(Math.random() * (1 - 0 + 1)) + 0;
      if (chooseornot == 1) { // play the note, otherwise continue
        note = i;
        note_prob = predicted_values.w[i];

      }
    }
  }
  lastnote.note = note; // maintain last note played
  post("note: " + note + ". prob: " + note_prob + ". " + "\n"); // print note and
      prob
  outlet(0, note); // outlet note to max patch
}
```

Listing 3.1. JavaScript implementation of the `peform` function

3.4 Network Configuration

Several parameters are kept standard for the networks tested in this project. For output, a *softmax* classification layer is used. This layer has 128 nodes representing all possible MIDI outputs. During performance, each node has a probability of being played, according to the input given. The softmax function squashes these probability scores to have a sum of 1. The note with the highest probability is played.

For hidden layers, sigmoid activation function is used. A fairly high learning rate is used in training due to the relatively small training sets and desire for fast and easy use, so *rectified linear units* (ReLU), which can be thrown off drastically by a large gradient, are not ideal.

The trainer also stays constant. The network is trained using momentum-based *stochastic gradient descent*, meaning weight updates after every batch of 10 training steps. A learning rate of 0.05 is used so that the relatively small dataset can be learned quickly. An L2 decay of 0.005 works to regularize training. Training is done by iterating through the song array 1 note at a time. This note is treated as the desired output, while the seven preceding it are the inputs. This entire loop is repeated however many times the user wishes.

Where the networks used in testing vary is in the number of neurons. Only a single hidden layer was ever used due to the somewhat sub-par performance during live playing. For early tests this single hidden layer contained 5 neurons. This was later increased to 10.

Because of certain issues with overtraining and small training sets, a few workarounds were added to performance mode. These included a flag for the last note played to prevent overplaying of the same note, as well as a small chance to completely ignore a note and play the next most probable pitch.

4
Results

4.1 Concert Performance

The first test of this project was a live performance, that performance being a part of my own senior project in music. This was done using an unfinished version of the patch, but was an effective way to hit the ground running. The Max framework was largely in place, but the configuration of the network had not been fully decided upon. The network used for this performance consisted of a single hidden layer of 5 neurons, used ReLU activation, and did not have standardized inputs.

This more simplistic and untested network was initially tested with several different small training sets. These included Jimi Hendrix songs, some video game themes, and Aha's "Take On Me," but ultimately, the song that produced the most interesting data was Scott Joplin's "The Entertainer." The network was trained on this song 2,000 times, which, with the additional limitations placed upon the playing of consecutive notes, produced reasonably interesting note patterns. Without those restrictions, however, the network exclusively produced middle C (60). The exact piece was not saved, but short examples of music played and graphs showing the frequency of certain notes during 2 minutes of playing with and without the repetition workarounds are shown in figures 4.1.1-4.

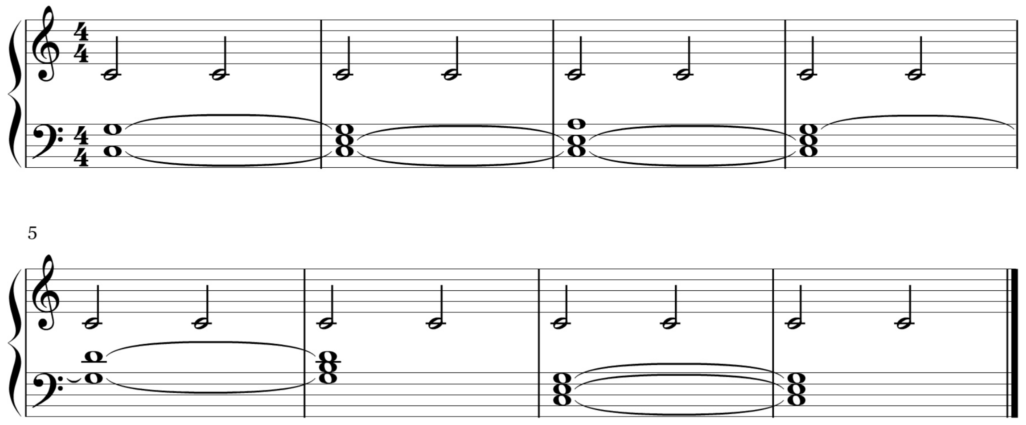

Figure 4.1.1. 8 Bars Played With The Network *Without* Workarounds

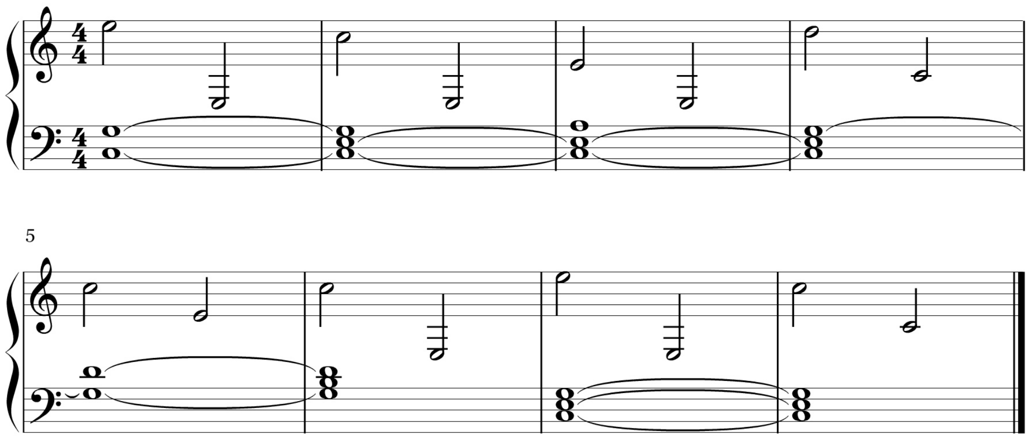

Figure 4.1.2. 8 Bars Played With The Network *With* Workarounds

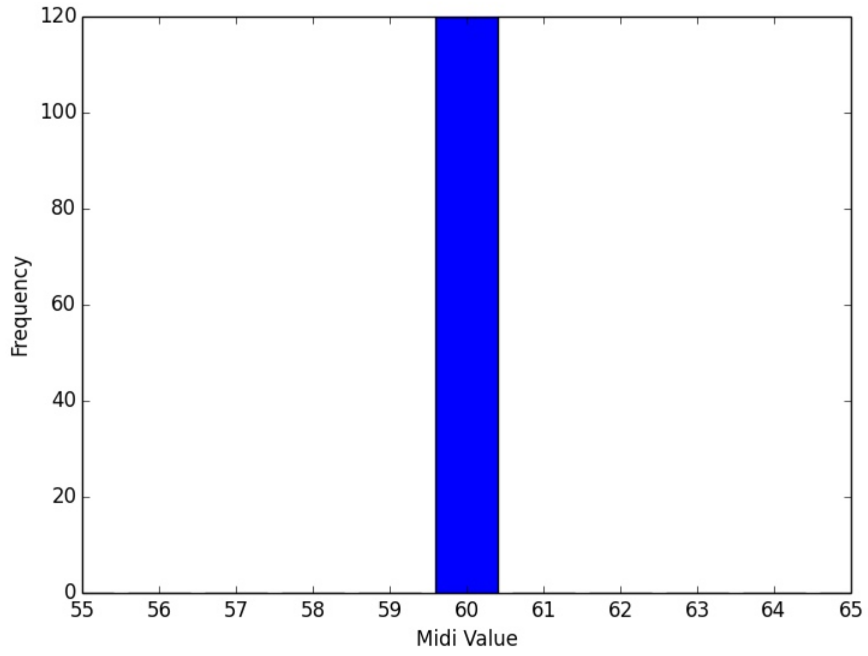

Figure 4.1.3. Notes Played By The Network *Without* Workarounds

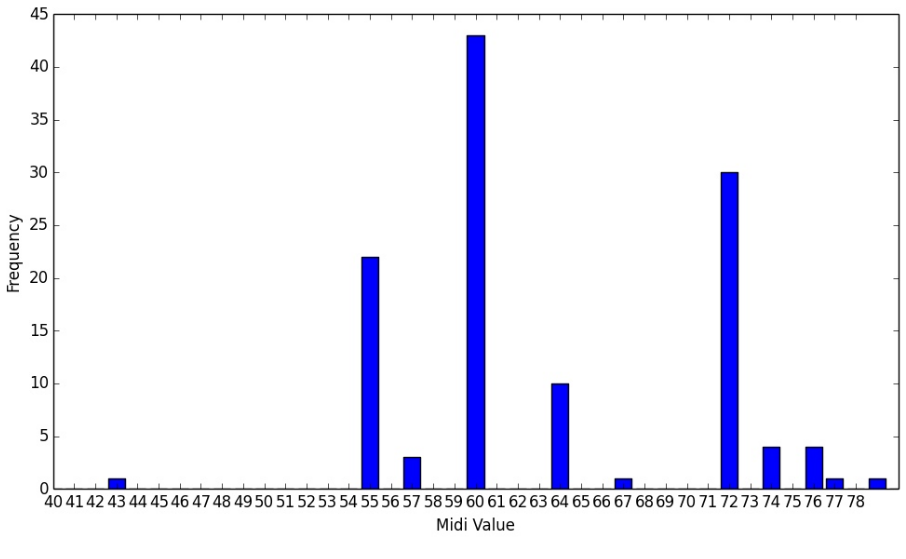

Figure 4.1.4. Notes Played By The Network *With* Workarounds

Figure 4.1.5. The Performance Using the Patch

4.2 Testing & Results

After updating the network implementation to have standardized inputs and 10 neurons instead of 5, a simpler test was run to more thoroughly make sure the network library was running properly. The input size was reduced to a single note, and the output was reduced to be a consistent perfect fifth above the input note (7 half steps). After some adjusting of the learning rate, as well as implementing a way to indicate training cost, the network was trained on a randomly sorted array of every possible note and its fifth. The network was trained on this list 5,000 times, and then tested over an octave of possible notes (C4-C5) to see how accurately it could find the fifth. The patch was not quite consistently able to play a fifth harmony, but the notes played each time were accurate within a few steps of the desired notes.

Next, the same network was tested on a smaller training set. This set consisted of only half of the possible notes and their fifths. The network was once again trained 5,000 times and then tested over an octave. The network was able to learn this data much more quickly, and with less training error, but also became much more likely to predict the wrong note. With fewer notes in the training data, there was less to learn, but this left the network less capable of dealing with brand new inputs (the other half of the notes). Figures 4.2.1-2 show the training error over time for both tests, while tables 4.2.1-2 show the notes played, correct fifths, and notes predicted by each network.

After completing this test, the original problem was returned to. The learning rate was increased up to 0.05, and the new, larger number of neurons was maintained. While retrying training using "The Entertainer" produced slightly more varied inputs, the minimal size and variance of the training data still did not allow for particularly diverse music generation. Instead, the network was trained on a medley of several well-known themes from the *Star Wars* movies. The presence of several pieces, in shortened form, in this new training data allowed the network to be trained on much more varied input, even though the overall size of the dataset was similar. This, as a result, produced much more varied output. However, this new output was still not quite fully generalized, and was at times very dissonant.

Similarly to the network used in the live performance, this new network was tested first by playing eight-measure musical segments and recording each note produced, both with and without the workarounds used in the concert. It was then also tested by playing for two minutes and recording the frequency with which each possible note was played in that timespan. Figures 4.2.3-6 show the results of this testing.

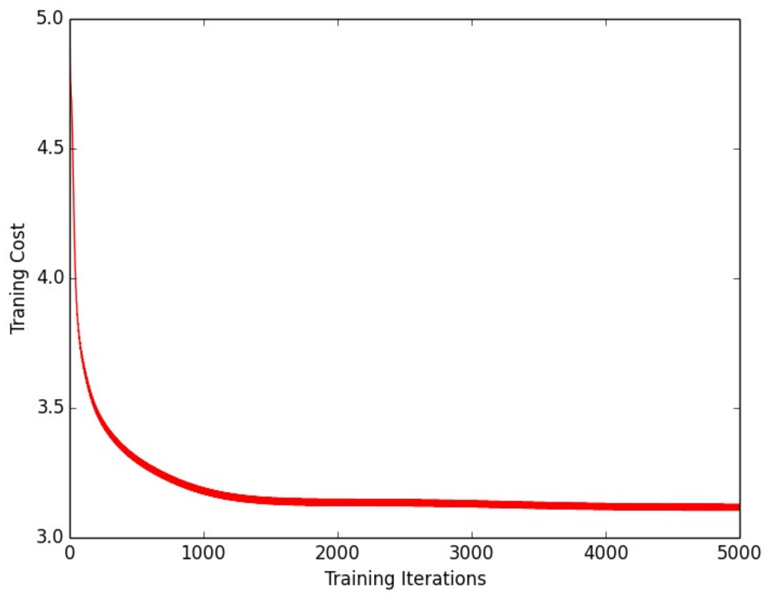

Figure 4.2.1. Training Cost of The Network Being Trained With All Possible Fifths

Input Note	Correct Note	Predicted Note
60 (C)	67 (G)	66 (F#)
61 (C#)	68 (G#)	69 (A)
62 (D)	69 (A)	69 (A)
63 (D#)	70 (A#)	69 (A)
64 (E)	71 (B)	71 (B)
65 (F)	72 (C)	71 (B)
66 (F#)	73 (C#)	71 (B)
67 (G)	74 (D)	71 (B)
68 (G#)	75 (D#)	77 (F)
69 (A)	76 (E)	77 (F)
70 (A#)	77 (F)	78 (F#)
71 (B)	78 (F#)	78 (F#)
72 (C)	79 (G)	78 (F#)

Table 4.2.1. Results of Training Network With All Possible Fifths

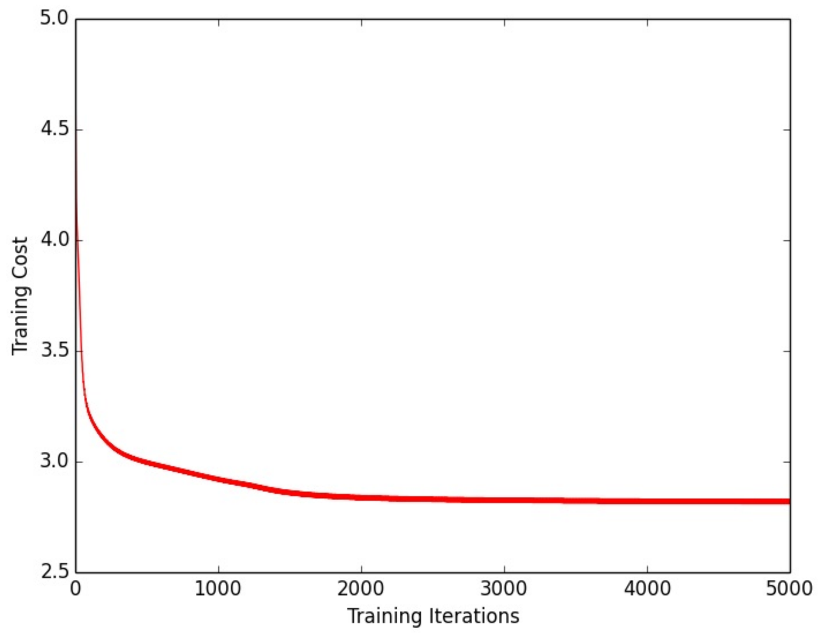

Figure 4.2.2. Training Cost of The Network Being Trained With Half of the Possible Fifths

Input Note	Correct Note	Predicted Note
60 (C)	67 (G)	67 (G)
61 (C#)	68 (G#)	67 (G)
62 (D)	69 (A)	67 (G)
63 (D#)	70 (A#)	67 (G)
64 (E)	71 (B)	71 (B)
65 (F)	72 (C)	73 (C#)
66 (F#)	73 (C#)	75 (D#)
67 (G)	74 (D)	75 (D#)
68 (G#)	75 (D#)	75 (D#)
69 (A)	76 (E)	75 (D#)
70 (A#)	77 (F)	75 (D#)
71 (B)	78 (F#)	75 (D#)
72 (C)	79 (G)	75 (D#)

Table 4.2.2. Results of Training Network With Half of the Possible Fifths

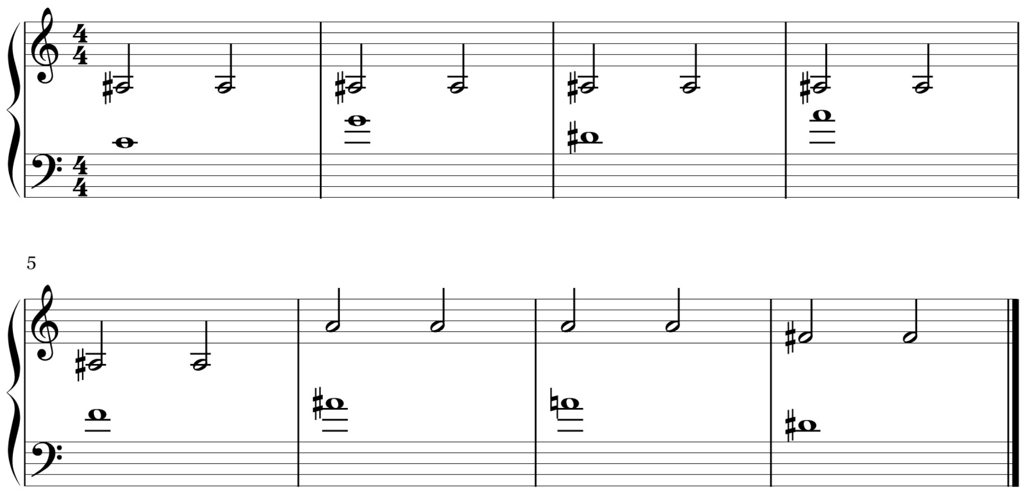
Figure 4.2.3. 8 Bars Played With The Finalized Network *Without* Workarounds

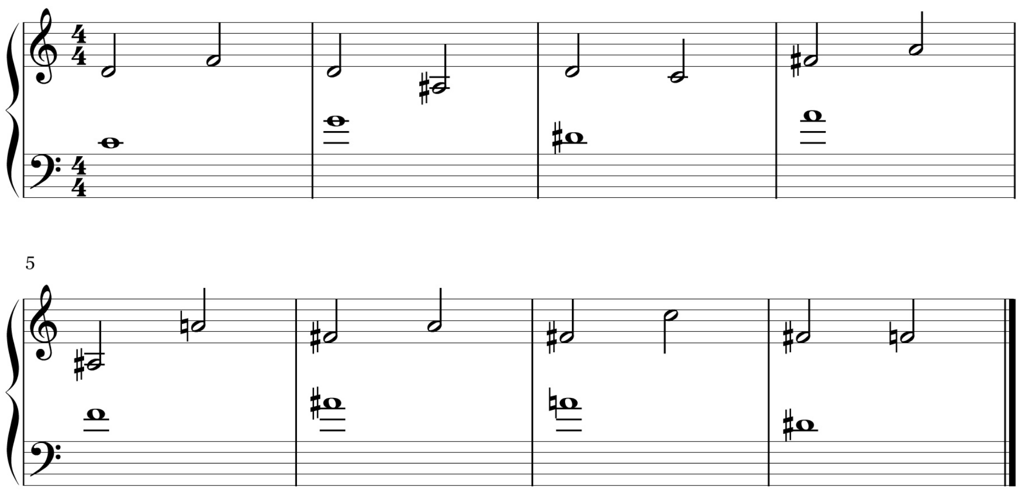
Figure 4.2.4. 8 Bars Played With The Finalized Network *With* Workarounds

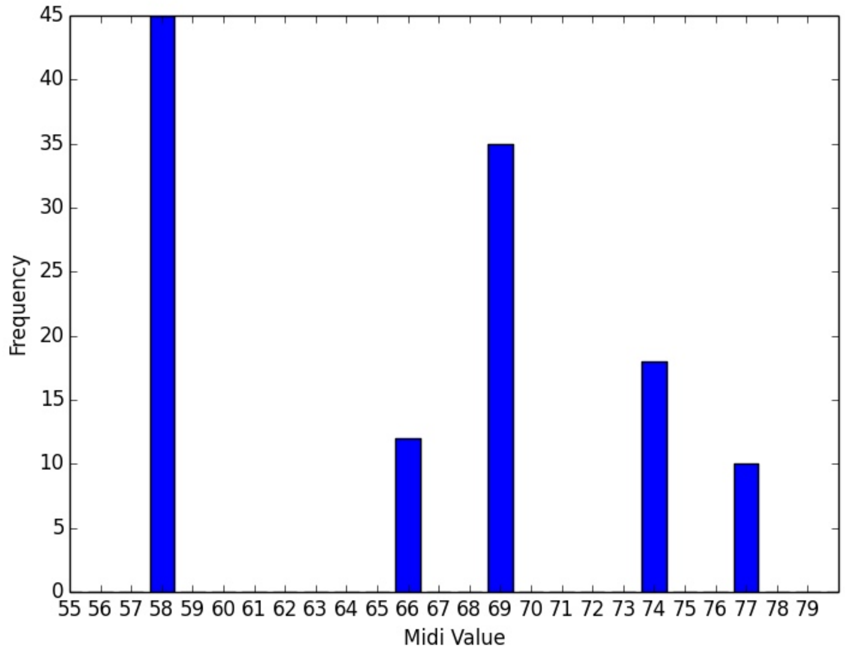

Figure 4.2.5. Notes Played By The Finalized Network *Without* Workarounds

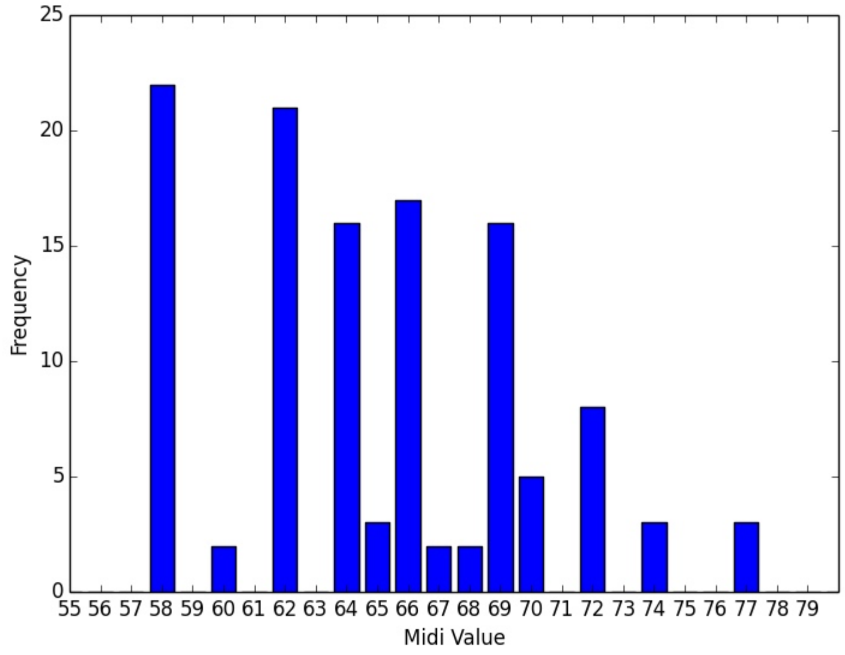

Figure 4.2.6. Notes Played By The Finalized Network *With* Workarounds

5
Conclusion

The tests run over the course of this project all point towards a similar conclusion. The networks simple enough to run smoothly within this implementation do not have the power to produce truly unique and varied musical output. Many other implementations of neural network-based music generation use recurrent models with hundreds of neurons, spread out over multiple layers. Attempting to implement this large of a network caused serious performance issues, as well as occasional crashing. Even with the noticeable improvements made to the network with the addition of standardized inputs and a greater number of neurons, the network still produced relatively simplistic, and sometimes dissonant music.

This was also partially due to the very small training sets used. Consisting of a single song, or even short samples of several songs, these datasets did not provide varied enough patterns to predict much beyond the most commonly appearing notes in the pieces. With some training sets, this mean only a single note was ever predicted. The use of a more varied medley of songs improved the variation in notes played, but at times created very dissonant music.

Perhaps the most successful and promising experiment was the harmonizer. While unable to predict fifths with much accuracy, the resulting network produced some interesting and unexpected harmonies in unison with the notes being played by the user. Training a neural network to do something that can be done with a simple equation, however, is a massive wast of processing power. Harmonizers very much already exist, and do not require machine learning. However, if the intention were to produce interesting harmonies instead of consistent ones, the use of a neural network would be far more justifiable.

There are still some possibilities for improving the prediction ability of these very simple neural networks. The first would be to abandon the idea of quick, small training sets, in favor of larger, more varied ones. These would take considerably longer to train with, but would produce far more variance in note prediction, resulting in more interesting musical output. Another improvement to the training process would be to rethink the linear way in which the training data is iterated through. Taking random segments of the training data each training step instead of iterating through the song one note at a time might prevent the network from simply learning which notes are most common.

Larger changes would have to be made for any more significant improvement. The implementation of a library written in one language into software written in another is not the most efficient way to do things. It is somewhat more simple and quick to implement, as well as being fairly modular, but unfortunately takes a lot of processing power to accomplish even the simple results produced here. A Max machine learning library called *ml-lib* was developed in 2014, and is available online, and while it would require modification for live musical performance, it might produce results far more favorable than those produced by this project.

The harmonizer network could be expanded upon quite extensively. Providing varied training including a range of harmonies that would sound consonant together could improve its functionality considerably. Both the harmonizer and the larger composition models could benefit greatly from a recurrent implementation. Taking previous predictions into account could improve the overall progression of a piece of music instead of the network predicting single notes, isolated from one another.

While this implementation of neural networks in a live music setting may not have been entirely successful, the framework it was built upon could be vastly improved. With such improvements as a recurrent model, larger training sets, and a full Max-integrated machine learning library, this idea of an instrument with a mind of its own could be much closer to reality. I am eager to continue this work and improve upon this concept.

Bibliography

[1] Jamshed J. Bharucha and Peter M. Todd, *Modeling the Perception of Tonal Structure with Neural Nets*, Computer Music Journal **Vol. 13, No. 4** (1989), 44-53.

[2] Douglas Eck and Jürgen Schmidhuber, *A First Look at Music Composition using LSTM Recurrent Neural Networks*, Technical report, IDSIA USI-SUPSI Instituto Dalle Molle (2002).

[3] Alice Eldridge, *Collaborating with the Behaving Machine: Simple Adaptive Dynamical Systems for Generative and Interactive Music*, 2007, `http://www.ecila.org/ecila_files/content/papers/thesis/behavingMachines_aliceEldridgeThesis_07.pdf`.

[4] Bernhard Feiten and Stefan Günzel, *Automatic Indexing of a Sound Database Using Self-Organizing Neural Nets*, Computer Music Journal **Vol. 18, No. 3** (1993), 53-65.

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

[6] Andrej Karpathy, *Hacker's Guide to Neural Networks*, `http://karpathy.github.io/neuralnets/`.

[7] Michael C. Mozer, *Neural network music composition by prediction: Exploring the benefits of psychoacoustic constraints and multiscale processing*, Connection Science **6** (1994), 447-462.

[8] Aran Nayebi and Matt Vitelli, *GRUV: Algorithmic Music Generation using Recurrent Neural Networks*, 2015, `http://cs224d.stanford.edu/reports/NayebiAran.pdf`.

[9] Benjamin D. Smith and Guy E. Garnett, *Improvising Musical Structure with Hierarchical Neural Nets*, Musical Metacreation: Papers from the 2012 AIIDE Workshop (2012), 63-67.

[10] *FAQ: Max 4*, `https://cycling74.com/support/faq_max4/#1`.

[11] *ConvNetJS*, `http://cs.stanford.edu/people/karpathy/convnetjs/index.html`.