Reverse Engineering Audio Synthesiser Sounds

One Semester Individual Project
Final Report

Author: Harrison Taylor
Supervisor: Professor. A D Marshall

School of Computer Science & Informatics
Cardiff University
May 2018

Abstract

Audio synthesisers are capable of producing a wide variety of sounds, stemming from their numerous amount of parameters that are used to affect the ouput sound. An issue in the area of audio engineering is that significant time is needed with each model of synthesiser in order to gain a core understanding of how the underlying components affect the sound that is produced at the end of the chain.

This project presents a robust methodology for analysing a synthesiser's sonic capabilities, and a framework for using Deep Learning methods in order to 'reverse engineer' how a synthesiser can make a given sound.

Acknowledgements

My sincere gratitude to Prof. Dave Marshall for his fantastic teaching on the subject area, and unrelenting enthusiasm for inspiring everyone to build cool things.

Contents

Chapter 1

Introduction

Audio Synthesisers[1] can produce an incredible array of different sounds depending on the control parameters offered to the player. Some synthesisers such as the Roland TB303[2] are able to be easily programmed by ear as they have a relatively simple synthesis circuit and a small number of parameters. Conversely, synthesisers such as the Yamaha DX7[3] are incredibly challenging to program by amateur musicians due to the amount of knowledge and experience required to fully understand FM synthesis[4] and it's large amount of parameters.
For the most part musicians with some experience with audio synthesisers can intuitively program a synthesiser to approximate a given sound. Using this concept an assumption is made that it is possible to train a Neural Network model to recognise the effect that each parameter on a synthesiser has on a it's output sound. Provided that the model has a good level of accuracy, it can then be used to approximate parameter settings for a target sound, for example, the output from a different synthesiser.
This project consists of research into the feasibility of using Deep Learning[5] techniques to create a synthesiser programming toolkit that is able to reproduce a close approximation of a given sound almost instantaneously. A system that provides a quick result and doesn't interrupt "the flow" is crucial in order to be included in a musician's creative process. Using a pre-trained neural network model is perfectly suited for this application as predictions on new input data are computed in almost real time.

Aim

The aim of this project is to develop a deep neural network architecture that can predict synthesiser configurations for a given target sound, and to prove that the system works by performing tests on a real freely available audio synthesiser.

Project Outcomes

This project presents an efficient Convolutional Neural Network architecture for reliably learning a close approximation of synthesiser parameters, and a comparison of common data representions including a novel representation suited for this task.
Included is a library for performing common preprocessing tasks for this project, including setting neural network architecture parameters, generating sample data, and converting data such that it is suitable for the neural network to use.
There is a focus on proving the system's capability with a simple synthesiser, however, given enough resources and time the system could work effectively with a much more complex synthesiser such as the DX7.

Chapter 2

Background

2.1 Context

This chapter contains a basic overview on a number of topics that are required to be understood for the context of the rest of the report.

Audio Synthesisers

Audio Synthesisers can be thought of as a black-box function that takes a list of parameter settings as an input, and produces a sound based on these input parameters.

Audio Synthesisers come in a range of different forms, operating on different sound synthesis principles and different control types. Fortunately for the scope of this project, there is an abundance of both hardware and software synthesisers, with software synthesisers becoming increasingly more sophisticated in recent years. Software Synthesisers are now so capable that a large majority of musicians almost exclusively use Software equivalents in favour of expensive hardware synthesisers. This is good for the applications of this software as it results in more potential users of the system.
The design of this project can be implemented with either hardware or software synthesisers, however due to the timescale for development, software synthesisers have a significant advantage. This is due to a couple of reasons: in the best case scenario, a hardware synthesiser can be computer controlled via MIDI & SYSEX MIDI[7], but the audio used as sample data needs to be recorded in real time. While this can be automated, it would take a significant amount of time and space to collect a sufficient amount of training data to train a Neural Network on. As this project is focussing on building

a foundational framework, fast iteration times in development are essential. A key advantage that software synthesisers have over hardware synthesisers is their ability to render audio samples offline. As the audio synthesis engine is essentially a pipeline of mathematical functions, and the resulting audio is represented as an array, there is no need to perform online (real-time) synthesis.

This results in a significant speedup for generating sample data. For example, given a 3 second long snippet of sample data, online synthesis requires at least the 3 seconds plus the time to transmit program change via midi, which is roughly 200ms. This results in roughly 0.4 Samples generated per second. Using offline synthesis the same sample can be generated in 160ms, resulting in 6.2 samples generated per second.

Virtual Studio Technology

There are many different formats that a software synthesiser can be implemented in, but fundamentally they are all implemented in a low level or real time programming langauge such as C/C++ or MATLAB. Software Synthesisers however need to be distributed in a conventional manner that is trivial for non-programmers to use. There are a few platform specific formats such as Apple's Audio Units, however, Steinberg's Virtual Studio Technology (VST)[8] is by far the most prolific. This is probably due to VST being the first successful standard for audio plugins, which came as standard with Steinberg's Digital Audio Workstation (DAW), Cubase. VST plugins can come in two forms - audio processors, or instruments (VSTi). Audio processors take an audio stream as input, and produce an audio stream with some processing applied to it as output. VSTi's however take performance data (via MIDI) as input, and generate an audio stream as output.

In order for a VSTi to generate audio, it needs to be initialised in an environment known as a VST Host. Fortunately Steinberg's VST SDK[9] is freely available to any developer, and as such there exists a host of alternative VST hosts that are feasible for use in this project. A VST host does not provide any recording utility; it simply instantiates a VST plugin and provides an interface for setting parameters and rendering audio based off an audio input or performance data provided by MIDI.

Using a VSTi with a bare-essentials VST host in this project is ideal, as there is minimal overhead for generating sample data which allows the focus of the application to be on generating useful patch settings.

The Fourier Transform

The Fourier Transform[10] breaks a temporally varying signal into it's fundamental frequency components, which can then be manipulated or presented for spectral analysis. In the context of audio processing, the FT is analogous to a graphical equaliser, where the frequency bands can be changed to affect the amplitude of different frequency bands of the signal.

In the context of this project, the Fourier Transform is essential in order to efficiently and effectively process audio information. This is because it is possible to visualise the result of the Fourier Transform as a Spectrogram using the Short Term Fourier Transform, which is essentially a plot of the relationship between frequency power against time.

Short-Term Fourier Transform

The Short Term Fourier Transform (STFT) is a way of applying the Fourier Transform to an audio signal. A sliding window function is used in order to apply the 1D fourier transform over the frequency bins for each time section in an audio signal.
The STFT is used in this project as a baseline method to develop the neural network. It is a lossless function meaning that there have been no modifications made to the representation. This is useful for determining baseline performance of the neural network and for debugging any potential issues during development.

Constant-Q Transform

The Constant-Q Transform[11] is a form of the STFT with the unique property of having a filter bank with geometrically spaced center frequecies that ensure that a constant Q factor is retained over all frequencies. The Q factor is a measure used to decide the size of each filter in the FT's filterbank, where: $Q = \frac{\text{center frequency}}{\text{bandwidth}}$.
The motivation behind the Constant Q Transform is to have a spectral representation that is tuned for musical analysis. As the transform maintains a constant Q factor that follows the equal-tempered scale of western music, it is a much more useful representation to use when analysing the output from a musical instrument. Where the STFT would squash together the lower frequencies due to it's scaling factor, the Constant Q transform ensures that each lower frequency is well defined. This is useful for the analysis of synthesiser audio as synthesisers produce complex waveforms that range

from both ends of the scale. By using this representation the fine detail in extreme frequency bands is represented equally.

Cepstrum Analysis

The Cepstrum[12] of a signal is calculated by computing the Inverse Fourier Transform of the logarithm of the spectrum of a signal. A Cepstrum is information about the rate of change in different spectrum bands, and is commonly used in speech and musical analysis as features of an audio dataset.

The Discrete Cosine Transform

The Discrete Cosine Transform[13] is a lossy transform that transforms a 2D matrix into a set of basis functions that are used to represent the most frequent components in the matrix. By taking a subset (i.e. the first 20) bases of a DCT, a good idea of most important aspects of the original data can be collected, discarding information that may be less useful.

Mel-Frequency Cepstrum Coefficients

The Mel scale is a scale that is tuned to the perceptual hearing of humans, and consists of a bank of filters that accomplish this scaling. MFCCs are essentially calculated by computing the STFT and applying the Mel-Filterbank to it. The DCT is then applied to the Cepstrums that have been extracted from the Mel-scaled STFT. The first n DCT coefficients (typically the first 20) are kept and presented as the MFCCs[14] as audio features.

MFCCs are commonly used as features in speech recognition and musical analysis applications due to it's use of combining cepstral features with the DFT. The most important part of the signal is kept and used as a feature which can significantly reduce computation time as the amount of data processed is reduced.

The MFCC data representation is being used as an alternative benchmark as it is a lossy representation, and to judge the feasability of using Cepstrum Coefficients as features for the Neural Network Model.

Machine Learning

Machine Learning is a blanket term given to a plethora of algorithms that all function upon the same principle: using example data to uncover hid-

den relationships in a dataset, and then use this gained knowledge to make accurate predictions about the relationships in new data. Supervised Machine Learning relies on labels that match with training data in order for the algorithm to evaluate it's performance. There are two main forms of predictions a supervised machine learning algorithm can make: classification, and regression. The formulation of this project's problem is as a multivariate regression problem, which means that the ML algorithm predicts the numerical value of multiple variables.

Neural Networks

Neural Networks (NNs) are a class of ML algorithms that are inspired by how the human brain functions. They can be represented as a graph of nodes, and data is propogated across the graph as it is processed. Each node mimics the behaviour of a neuron: a value is passed through each node that is based off the value that is passed in, which is defined by an activation function.

NNs 'learn' information about a dataset by adjusting the weighting of the effect of each node in the graph. These weights are adjusted at a rate of change known as the learning rate. The idea is that the weights are going to change dramatically at the start of a training session, but are going to be updated at very small increments towards the end of the training period. Training is completed over a series of epochs: where the same dataset is propogated over the network a set number of times, with the network updating the node weights with each iteration.

The NN uses a cost function as a way of getting feedback on the weight modifications that are made, with the goal of reducing the cost as much as possible. However loss is not an accurate metric on it's own, so it is often accompanied by an accuracy function which evaluates the rate at which the the NN is correctly classifying or processing the input data. A good accuracy metric reflects the problem domain by setting a threshold for what is an acceptable prediction in relation to the target data.

Deep Neural Networks

Deep Neural Networks (DNNs)have been broadly used with generally good performance since Hinton's classic ImageNet paper 2012, and have proven to be an efficient way of learning relationships in all forms of data. The main

concept is to stack layers of hidden layers in between an input and output layer. The neural network shifts it's weights of each node to compensate for activations in order to produce a reliable prediction. Typically the training phase takes a long time, however, once a model has been trained and has been proved to perform effectively, then the model can be used evaluate predictions in real time.

Convolutional Neural Networks

Convolutional Neural Networks are often seen as the de facto standard for Deep Neural Networks. They are made up of a combination of Convolutional layers, usually followed by Max Pooling Layers, in order to break down an input image into it's most important components. The final layer of the network then uses the strength of each of these components in order to output a prediction about the input data.

2.2 Previous Work and Alternative Approaches

CUROP Research

This work builds upon previously completed research for the same problem. The CUROP work approached the problem using MATLAB 2017a's audio toolbox, with a Command Line based VST host, and a 2D Convolutional Neural Network. MATLAB 2017a provides a comprehensive toolbox for audio analysis, which includes a suite of functions to analyse VST audio plugins. A few functions proved to be useful including the parameter response mapping *, and functionality polling function which is able to return the parameters a VST offers including the properties of such parameters - a function was implementated that is able to determine the cutoff points for parameters that aren't continuously variable, such as a waveform selector. In conjunction with MATLAB, which was used to generate a list of patch settings,a Command-Line based VST hostmrswatson was used to render audio samples offline and store them to disk. This method provides a great insight into how the VSTi functions, and laid the groundwork for this project, however, it ultimately proved to be too cumbersome and time/space inefficient for the task.

Genetic Algorithm Approach

An alternative approach to solving this project's problem is to use stochastic search algorithms in order to automatically program a synthesiser to sound like a target sound, as outlined in synthbot[15]. This method relies on using a genetic algorithm to perform gradient decent while trying to reduce the Standard Error between the target sound and generated sound. This method works with reasonable effectiveness, however, it takes roughly 3 minutes to compute a predicted patch, and there is no guarantee that the patch that is predicted will be 100% accurate. This is not essential as making music is a creative process, and as a creative tool SynthBot is a fantastic concept, achieving parameter settings that sometimes the user would not think to try.

However, the system does not learn a representation of the synthesiser's capabilities, and as such there is no way of understanding the system's 'reasoning' behind picking a certain setting. The potential with DNNs however is that by examining node activations it may be possible to gain an understanding of the features that a NN is learning to reverse engineer a synthesiser sound.

2.3 Tools & Software Used

IPython with Jupyter

Python was the programming language of choice due to it's flexibility when researching different aspects of the project. As there was a lot of experimenting and evaluating novel libraries to achieve the goals of the research question, the reduced development and iteration time that Python allowed was invaluable, especially with the short time scale of the project. Jupyter[16] was used to easily share results as well as it's seamless integration of visualising data.

RenderMan

RenderMan can be thought of as a VST host API for Python. It provides bindings for low level C++ interaction with a hosted VST synthesisers, and as a result it is very fast compared to other programmer-oriented VST hosts, and is very flexible due to the ability to interact directly with VST instruments using Python scripts.
While RenderMan does not provide as much in terms of audio analysis as the MATLAB Audio Toolbox, there are other libraries that can be used in conjunction to provide the same functionality. As RenderMan provides a 1:1 mapping to the C++ code running the VST host, it is possible to generate audio samples offline and store them directly to a list. This removes the bottleneck of having to write and load samples to disk, and ultimately making massive improvements to runtime while reducing space requirements.
RenderMan essentially provides a way to generate samples on-the-fly. The only downside to this method is that it is not as easy to analyse VST instruments during development, meaning that initial analysis and preprocessing took a little longer, however when it comes to runtime the benefits far outweigh this negative.

Librosa

LibRosa[18] is the most well featured audio analysis library currently available for Python, and it's tools have been invaluable. It is the main library used in this project for computing spectral representations of the audio sample data.

Keras

Keras[19] is a high level library for Deep Learning in Python. It is being used with the TensorFlow backend as the state of TensorFlow seems to be the most future proof.
Keras allows for quick learning and implementation of Deep Learning concepts with fast results. Using Keras greatly reduced development time for this project as the ideal Convolutional Neural Network was constructed using mostly predefined Keras layers.

2.4 Practical Technological Applications

The framework that has been developed is a platform for future work in the area, however an envisioned completed product would be invaluable for the following uses:

A Creative Tool for Musicians

There has been significantly more interest in synthesisers that make unconventional sounds, which has been expressed by the rising popularity of modular synthesisers and unique software synthesisers such as the Spectrasonics Omnisphere. This software could perhaps be used to generate patches that closely approximate a sample, which can be used in the creative process as an inspiration for new music. Musicians are constantly pushing the limits of their instruments, and this software would enable this even more.

Reverse Engineering VSTi's

This work could be used to benchmark VST performance or to help model the sonic capabilities of existing VST instruments, or to even be used as a way of unlocking the hidden potential of synthesisers that are incredibly complex. As the ideal system would operate in real time, an example could be to enable any synthesiser to perform a vocoder-esque sound by calculating a patch or a sequence of patches to be changed in quick succession that could mimic a recording of a person's voice.

2.5 Summary

Research Question

In order to demonstrate the achievement of the stated aim, this project evaluates real-world performance of a software system that is able to reliably generate accurate patches for a simple VSTi synthesiser. A sufficiently large dataset of 1000 unique patches is used to evaluate the performance, with experimental results outlined in the **Results** chapter.

Chapter 3

Method

3.1 Overview

This project is attempting to prove the feasibility of using Deep Learning to answer the proposed research question. As such, there is an initial focus on showing that the solution has potential, followed by research into how the solution may be improved to achieve higher performance. In order to achieve the aim of this project, a development framework has been designed as a test bed to compare and experiment with different approaches to solving the problem within the scope of Deep Learning. The framework utilises a library of functions that has been designed to abstract away all of the nonessential details from the development framework.

This project has a strict focus on outlining a solid foundation for automatically programming any synthesiser, and thus explores different combinations of tools in order to best achieve this task, and evaluates the cost / benefit of every approach. The project can be broken down into two distinct parts in order to solve the research question. Firstly, **Data Collection** outlines Data Discovery, Data Generation and Data Representation. **Network Architecture** includes Neural Network Design and Performance Metrics. A general development framework was designed and implemented to support quick development and iteration of both these parts, outlined in **Implementation Overview**.

Project Structure

The agile development workflow model was used during development to quickly answer questions about the solution and to guide the research focus. As such the components featured in this project have gone through many iterations, as the most suitable method was development. The main advantage of this meant that the simplest solvable version of the problem was answered relatively quickly, and lessons learned from that stage influenced the design and implementation for the rest of the project. The rough outline of development was as follows:

1. Problem Domain Research: Data Representations, Sample Data collection methods, Neural Network Architectures.

2. Exploratory Data Analysis: Manual analysis of sample data, tuning of sample data collection parameters, including calculation of train / test set parameters.

3. Framework Development: Build a platform for quick testing of synthesiser programming pipelines - Includes accounting for time and space constraints.

4. Experimentation: Evaluating performance of neural network architecures combined with different data representations.

5. Optimisation: Improving best performing architecture by experimenting with hyperparameter changes and network layers.

6. Evaluation: Analysis of overall results.

Steps 3 to 5 were iterated several times until the solution was optimised to a satisfactory level.

3.2 Data Collection

This problem has the unique chararacteristic of being able to generate new data for a Neural Network on demand, rather than having to base the solution on a given dataset. Therefore careful consideration has been made in regards to designing a dataset that will give a good tradeoff between model accuracy and compute time. Too small of a dataset and the network will not learn an accurate representation of the capabilities of the synthesiser, and too large a dataset is unfeasible by the amount of time taken to render

and process.
Synthesisers are programmed by setting each of the parameters available to a certain value in the range of 1 and 128. In order to recall a certain configuration of parameter settings, they are stored in a data structure known as a Patch. Patches are unique for each synthesiser model. Different patches yeild different sounds from a synthesiser, and can be thought of as the timbral encoding of a particular sound.
In order for the DNN to learn a useful representation of what sounds a synthesiser can make, a sufficiently large dataset of samples need to be rendered that accurately capture the sonic capability of the synthesiser. Usually random sampling would be sufficient to capture the essence of the data source, however, a lot of parameters in synthesisers have a non-linear response which results in a large change in the rendered audio during a small change, or can result in almost no discernable difference in audio at all over a large sweep. If this was constrained to one parameter or had predictable behaviour, then a sampling method could be devised that accounts for this. However, as parameters react to the change of other parameters it is not possible to predict a parameter's response. This effectively means that each parameter has a variable response in relation to every other parameter's setting in a patch. Therefore the most effective method of representing the sonic capabilities of a synthesiser is to create a map of every possible patch, and take equally spaced samples of patch settings to render. As an example, assume a synthesiser s has two parameters, x and y. It is a computationally trivial task to represent the sonic capability of s. Assuming that the synthesiser is adhering to the MIDI specification, there are 128 possible positions for x and y independently. The full capability of s is therefore represented by a bank of patches of size:

$$128^{(\text{number of parameters})} = 128^2 = 16,384$$

This method is acceptable for a synthesiser like s, as there are only two independent parameters. In reality it is not practical to use a two parameter synthesiser as the sonic capabilities will be severely limited.

To carry on with the example, assume that the full sonic capability of a simple single osciallator synthesiser such as the Roland TB303 is to be captured. There are 5 parameters that effect the timbre of the output sound that are continuously variable, and one parameter that is a discrete 2 way

switch. Assuming the MIDI specification is used, the bank of patches that describes every possible sound that the synthesiser can make has a size of:

$$2 * 128^5 = 68,719,476,736$$

Assuming that the render of each patch is 3 seconds long, with offline processing it would take roughly 120 milliseconds to render each patch. To render the entire bank of patches it would take

$$120 * 68,719,476,736 = 8,246,337,208,320\text{ms}$$

This is equivalent to roughly 261 years. This obviously means that even for a very basic synthesiser it is not feasible to sample every possible sound.

The solution used for this project is to take the space of every possible patch that a synthesiser can make, and subsample patches at an evenly spaced rate. This results in a bank of patches that provides a reasonable representation of the sonic capabilities of the synthesiser, while being able to render in reasonable time. The parameter sample ratio is part of the experiment configuration, and can express a range of detail in the patch bank, which can be tweaked as required.
A proposal for a more sophisticated patch selection algorithm is outlined in 'Further Work'.

Exploratory Data Analysis

Before designing and constructing neural network architectures, and even data preprocessing methods, it is almost always essential for every machine learning problem to perform an initial analysis on the dataset that is going to be used to train the network. The dataset for this problem is generated by performing a form of sampling on the VST instrument. To help understand the data source better, the VST standard defines an interface for polling the VSTi for it's parameters, along with current patch information. This functionality is implemented in RenderMan and is used to help decide on render settings such as Note-On time, and Render time, and is also used to verify that a VST is working, and responds to paramater changes as expected which is useful for sanity checks.

Sample Data Representation

The audio that the VST synthesiser makes needs to be represented in some format that the Neural Network will be able to process. The simplest way

of doing this is to sample the amplitude of the waveform at regularly spaced intervals, at a rate that is at least twice the highest frequency of the waveform. The VST specification accounts for this in order to represent audio in a computer. A typical sample rate is 44100Hz, which is roughly CD-quality. Any higher results in a lot data being represented without any increase in perceptual quality. Audio data being represented in this way is referred to as an array of audio frames.

Audio frames are the purest way of representing the patch samples from the synthesiser, however it requires a lot of computing power to represent important underlying concepts using this. Fortunately there is a way of representing audio data in a way that can emphasise aspects of the signal, while drastically reducing required processing power. This aspect of the project offered an opportunity to experiment with different ways of representing the input data and to evaluate which representation offers the best performance. Experiments were designed to evaluate the performance of three distinct time-frequency representations:

- The Short Term Fourier Transform as a lossless baseline
- Constant Q Transform as an optimised representation
- Mel Frequency Cepstral Coefficients as a lossy, distilled representation

Due to good performance demonstrated in similar tasks, the MFCC is included as an alternative approach to representing the data. The computation time is less than both CQT and STFT, however it results in a lossy representation of the data. This is not a bad thing as it means that the network is only computing relevant information, however, as the Mel scale is tuned for perceptable human hearing representations, it means that some of the sonic character of more complex synthesisers will not be represented.

3.3 Network Architecture

The problem is classified as a Multivariate Regression problem, with an output space equal to the amount of learnable parameters on the synthesiser model. This problem style makes it difficult to accurately evaluate the network's confidence in each parameter, as the output space is simply the predictions for the whole patch. This can be solved by designing a network that provides classification confidence measures for each parameter in the

patch, however this is outside the scope of this project, and could be explored in future work.
The input shape for the network is the dimensions of the input data: the height being the amount of frequency bins in the spectral representation, and the width being the amount of time that the spectral representation is for. These parameters vary for the different spectral representations, so this input shape value needs to be variable.

Convolutional Neural Networks

Due to the success enjoyed by convolutional neural networks in image classification tasks and sound recognition[20], a well defined CNN is employed as a tradeoff between training time and parameter accuracy. The CNN used follows a relatively conventional architecture, with modifications made to suit the task. For example, there are no dropout layers used which introduce generality into the NN model. This is because each Neural Network model (i.e. the collection of weights combined with the architecture) is unique to each synthesiser model. There needs to be generality in the architecture, not in each individual model. Because of this, a desirable trait of the network is to overtrain the model on the data given more than usual networks. Typically a prediction is a close approximation of an input sound, meaning that the user simply needs to tweak one or two parameters in order to fully reach the desired sound.
Previous work in this problem domain used a 2D convolutional network in order to find spatial features - interesting components in the spectrogram's signal, with mixed results. Learning from this, the NN employed in this project uses 1D convolutions which are designed to detect features in the temporal range. The filter moves from start to finish with a window the same size as the frequency bin, similar to how a spectrogram is generated. This representation suits spectral analysis perfectly as the importance of each frequency bin per time step is effectively represented in the network.

Performance Metrics

In order for the Supervised Deep Neural Network to learn the correct parameters for a given input sound, it needs to have accurate feedback on it's performance. For this problem, Mean Squared Error is used as it is a well defined benchmark to rely on for the output data type.

In order to gauge how well the network performs while training, a cus-

tomised accuracy measure has been designed that takes into account the specific experiment's setup. This accuracy metric defines the threshold for success as a predicted parameter being within the resolution at which a dataset is generated at.

The general MIDI specification only allows parameters to take an integer value between 1 and 128, which significantly reduces the search space for a parameter. This has been taken into consideration and a soft quantisation function has been designed that squashes the NNs output to be closer to one of the 128 steps.

Chapter 4

Implementation

4.1 Overview

In order to satisfy the research question, a small library has been developed that includes commonly used methods when rendering a list of synthesiser patches as well as some data preprocessing and result analysis tools. This library was then used to implement code used in the neural network experiments. A general framework has been designed that streamlines the benchmarking process.
The target systems for this framework are Linux and MacOS. A Linux-based system with a CUDA-capable graphics card was used during development to quickly debug and test deep neural network models. The resulting code can be run on both Linux and MacOS systems, however, as MacOS systems typically do not utilise an Nvidia graphics card, training of the NN models will take more time. It is desirable to use a MacOS based system for this project, as the majority of popular VST instruments are only compatible with Mac. With this constraint in mind, an ideal feature of the implemented solution was to have an architecture that was able to be trained in reasonable time.
The Nekobi[1] VSTi was chosen as the synth to reverse engineer for this project. It is a monophonic single oscillator synthesiser based off the Roland TB303, with 6 parameters that affect the output sound's timbre: Waveform, Filter Cutoff, Filter Resonance, Envelope Modulation, Decay, and Accent.

[1] https://github.com/DISTRHO/Nekobi

Nekobi was chosen as it is a simple synthesiser, meaning that it was easy to debug any issues that arose during development, but still could produce a relatively wide range of sounds in order to present the capability of the automatic programming system.

4.2 Experimental Process

A broad overview of how the system functions is as follows:

- Generate Patch List based on sample resolution
- Shuffle the patch list
- separate patch list into batches of size 128, with a 0.8 split for test train
- Iterate through 5-7 for each batch of samples:
- Render the 128 patches of the batch
- Train network on 128 patches while CPU computes data representation of the audio
- Store the latest weights to disk

Notable implementation details are described in this chapter.

EDA and Parameter Configuration

RenderMan was used to host a VST instrument for the duration of the experiment. Exploratory Data Analysis consisted of polling the VST for it's parameters, and evaluating which parameters are within the scope of this project to be learning, and which parameters should remain fixed. For the Nekobi Synth, "Bypass", "Volume" and "Tuning" were fixed, as they are parameters which do not affect the timbral quality of the output sound. A few patch settings were auditioned to determine the render settings. This consisted of finding a suitable cutoff point for noteon and render length. 300ms was found to be a good length that meant that each patch was able

to express its own unique sound. 300ms also proved to be a good tradeoff for render time and prediction accuracy - tests where the note was allowed to fully decay (1500ms) took almost twice as long to train with notably poorer performance.

Patch List Generation

A function has been implemented that generates a list of patches to be used to generate sample data for the neural network to train on. Random generation of training samples has not been used for this project as it is unreliable to trust that a sufficiently large enough bank of random patches will cover all of the key sonic characteristics that a synthesiser has. A tradeoff for this project is to use a subsampling function which defines a "step" for each parameter based off a given resolution value:

$$step = \frac{1}{2^{(resolution)}}$$

The minimum resolution value is 1, which produces a stepsize of 0.5, resulting in the steps 0, 0.5, and 1 being used for that particular parameter. In order to achieve full sampling of a parameter, the resolution should be 7; ($\frac{1}{2^7} = \frac{1}{128}$). This enables a quick way of producing datasets that are small enough to test that a network is functioning, while allowing a facility to perform up to full sampling if required.

In conjunction to this, Python's Itertools package has been used to implement a function that returns a list of patches in suitable format for the VST host. It works by calculating the list of potential steps for an iterable parameter, and then computing the cartesean product of that list multiplied by itself for the number of iterable parameters. This is the basis for the algorithm, however some preprocessing and postprocessing steps are required:

1. Initialise empty patch, a list of tuples for each parameter, where t[0] = parameter_id, t[1] = parameter_value
2. Remove each index of excluded parameters via their index from the empty patch
3. Perform calculation of patch list settings, with a repeat value = number of iterable paramters
4. Reconstruct each patch by appending the fixed parameters to the template patch and sorting by parameter_id

5. Return patch list ready for generating sample data

Patch List Preprocessing

Two additional functions have been created that convert patch information into a label that is suitable for a network to use when trianing. This is essential as training predictions are output as an n-dimensional NumPy array, and the network expects a corresponding size array to be used as a label. For the structure of the network, only parameters that contribute to the timbre of the output signal are included - fixed paramters that are defined in the EDA stage are not to be predicted by the NN model. The functions label_to_patch and patch_to_label both take two arguments: the list to be converted, and the fixed parameters.

Patch_to_label works by first converting the list of tuples into a list of floats, which represent the parameter value, then removing the floats in the list that fixed_parameters defines as being values to be excluded from training. The returned array is then ready to be used as a training label for the network as a list.

When validating results after the network is trained, it is essential to sanity check sometimes and to also evaluate the performance on unseen data. For this reason, label_to_patch has been implemented which works by initialising a patch full of the fixed_parameters and iterating through it, inserting a parameter from the label when an index is missing. These simple conversion functions can be thought of as encoders and decoders for the network's patch predictions.

4.3 Data Representation

Using the patch list generated by the functions outlined in Patch Selection, a list of patches is used to generate sample audio as raw waveforms. In programming terms, the VST host which provides an API for the VST returns a 1D array containing a list of amplitudes as a floating point data representation. This array is the waveform for the output signal. As previously mentioned, the waveform needs to be transformed into a time-frequency representation before being given to the network as sample data. The Librosa is used to accomplish this using the STFT, CQT, and MFCC functions.

Short Term Fourier Transform

The Short Term Fourier Transform implementation that LibRosa uses has a good set of default settings, and is representative of a good benchmark. If the network can learn parameter settings using this standard representation, then a modified representation should yeild significant performance boosts.

Constant-Q Tranform

Librosa's implementation of the Constant-Q Transform allows a range of parameters to be manipulated, however the most relevant parameters that were modified were the number of frequency bins, and the bins per octave. The number of bins was increased to 176 from 88 in order to provide more resolution between the frequency bins. 88 frequency bins maps to each note on a standard grand piano, and as the synthesiser's output has a harmonically rich signal, the bin sizes have been doubled to 176 in order to capture as much sonic information as possible while still giving equal representation to all bins in the frequency scale.

Mel-Frequency Cepstral Coefficients

20 MFCCs are used per time step to represent the audio. This is a good amount in order to represent just the important information about the signal. Librosa's implementation of the MFCC calculation could be passed a melspectrogram as input, which is a mel-scaled STFT, however there was little benefit to tweaking these parameters as the MFCC representation is so different to the other two data representations in this test.

4.4 Deep Neural Network Architecture

The following table describes the architecture of the Convolutional Neural Network used for solving the research question:

Training Parameters

Mean Squared Error is used as initially the experiments are trying to reduce overall error when predicting the patch, and this is a good metric for this. Adam optimiser is used with a learning rate of 0.001 to ensure that the network reliably builds up knowledge slowly. Early experiments showed that 50 epochs per batch were enough to reach peak performance without stagnating, so this value was kept for the final experiments.

Table 4.1: SynthNet V1.0 CNN Architecture

Layer (Type)	Parameters
Input	Input Shape = Spectrogram Dimensions
1D Convolutional	Filters = 256, Kernel Size = 5, Activation = ReLU
Max Pooling 1D	Pool Size = 2
1D Convolutional	Filters = 256, Kernel Size = 5, Activation = ReLU, Padding = Causal
Max Pooling 1D	Pool Size = 2
1D Convolutional	Filters = 256, Kernel Size = 5, Activation = ReLU, Padding = Causal
Max Pooling 1D	Pool Size = 2
1D Convolutional	Filters = 84, Kernel Size = 5, Activation = ReLU, Padding = Causal
Max Pooling 1D	Pool Size = 2
1D Convolutional	Filters = 84, Kernel Size = 5, Activation = ReLU, Padding = Causal
Max Pooling 1D	Pool Size = 2
Flatten	
Dense	Units = 84
Dense	Units = (number of parameters), Activation = Soft Quantisation(Step Size)

Soft-Quantisation Layer

As the general MIDI implementation relys on steps of 1128, it makes sense to encode this limitation into the network. The parameter settings that the network are given are bound to this scale, so by encoding this fact into the network, it takes less work to reach a reliable prediction. While it is not possible to implement a hard quantisation layer that snaps predictions to steps of 1128, a soft-quantisation method has been used to transform tensor values closer to the 1/128 "snap" point.

Accuracy Metric

It is useful during training and model evaluation to see the real-life accuracy of the training predictions to see realistically how good the model is learning relationships in the data.

A custom accuracy function has been implemented with a modifiable threshold. For example, if the resolution of the generated steps is "2", then a step of 1/4 is produced - values in the range [0,14, 12, 34, 1] are used. As these are the only values possible for this resolution, the accuracy metric returns "true" if the predicted value is within 0.25 of the actual value. This gives a good way of objectively classifying the performance of different architectures and data representations.

4.5 Neural Network Training Framework

1. Parameter Configuration

EDA needs to be performed on the data source (the VSTi) in order to define experiment parameters. This is because every synthesiser has unique characteristics that cannot be easily discovered automatically. The following variables must be defined before NN training can commence:

- VST Path
- Parameter Subsampling Resolution
- Note Pitch
- Note Velocity
- Note-On Time
- Render Window Length
- Data Representation Type

Some of these values such as VST Path and Velocity do not need to be changed throughout the experiment, however, different results can be acheived by changing each parameter. Additionally, the fixed parameters for the synth must also be defined, along with their settings. This is represented as a list of tuples. The fixed parameters are typically those that do not change the timbre of the output sound.

2. Patch List Generation and Preparation

Sample Data is being rendered on the fly for each iteration of training the neural network, however, the instructions to generate the sounds need to be defined beforehand. The gen_patch_list function uses 2 parameters in addition to the VST path to generate a patch list that samples the sonic capabilities of the defined VST at the resolution defined.
When training, neural networks can tend to find patterns in consecutive data. This negatively affects the performance of the network's accuracy, so in order to offset this, the patch list is shuffled using numpy's random shuffle function.
Finally, the patch list is partitioned into individual patch banks of size 128. This is so that the sample data can be rendered and stored in memory. For example, the Nekobi synthesiser had a patch list of 4,096 when generated

with a resolution of 2. It is unfeasible to store all of this data in RAM to be used to train the network, therefore training the network on batches of data is the most efficient solution.

3. Neural Network Parameter Setting

The neural network architecture is dependent on two variables for each synthesiser and data representation combination. The input shape for the network is the same dimensions as the data representation. The output shape is the amount of parameters that the network needs to predict in a patch. This is equal to the amount of parameters on a synth minus the fixed parameters. Additionally the accuracy threshold metric may be set in accordance with the resolution of the generated samples and expected performance of the network when paired with a certain data representation.

4. Batch Training

For each batch of patches, the batch is split into test and train partitions, and the VST Host is used to render the raw audio according to the patch and configuration options. Once the raw audio has been rendered into the test and train arrays respectively, Keras is initialised to start training the model that was defined earlier. The NN model is treated as an object with Keras, so consecutive calls to fit the model on the data are appended, rather than reinitialising the model with the defined architecture.
As the experiment platform is Linux with a CUDA-capable GPU, the network is trained on the GPU. In order to efficiently use both CPU and GPU at the same time, Keras' fit_generator function is used. A Python generator has been implemented that converts raw audio samples into their corresponding spectrogram representations using LibRosa. This function is run on the CPU and shortens training time by reducing bottlenecks on the system.

5. Logging, Results, and Model Evaluation

During training, a few logging features were used to ensure that the system is functioning as expected. Firstly, TQDM* is used to monitor the progress of the overall batch training. The time between iterations is used to give an estimate of the amount of time it will take to fully train the network, and is also used to monitor the amount of time it takes to render the audio using the VST host. Keras' CSVLogger is used to log loss and accuracy over time

with each epoch. This is useful for identifying areas where the network's accuracy and loss reaches optimal levels, and to demonstrate the effectiveness of training, as some configurations reach peak performance much earlier in training than others.
After training, the model configuration is exported as a JSON file, and the weights of each layer are exported as a H5 archive. This allows the trained model to be reinitialised without having to wait for the network to train.
Each model was evaluated after training by predicting the patch settings for a fixed set of 1000 randomly generated patches for the Nekobi synth. The dataset for evaluation was kept constant in order to keep a fair comparison between the model's performance.

4.6 Unforeseen Development Issues

Unfortunately not every idea that was envisioned during the development process was implemented either due to time or knowledge constraints:

Encoder - Decoder Style Network

WaveNet* is a generative Autoencoder style network that is able to operate on pure waveforms of an input signal and reproduce a representation. WaveNet has impressive results, but also has significant hardware restrictions.

The core ideas of WaveNet inspired a network architecture design for this project. Utilising the AutoEncoder style design, a 'decoder' would be used to represent the network's patch predictions as time-frequency. The most optimal CNN architecture would be used to encode the values, and then in order to compute a more accurate loss function, the model would try and match the output to the input.

In theory this would have significant performance gains as it address the aspect of synthesisers that there is not always a direct 1:1 mapping of changing parameter settings to change in audio - there are sometimes black spots where the changing of a parameter has no effect. There is no way of encoding this aspect to the neural network using a simple CNN model. The encoder-decoder style would learn these 'blackspots' and therefore wouldnt apply penalisations to the encoder section for getting the wrong paramter setting.

This model would take significantly more processing power and time, but would ultimately produce a better model. Unfortunately, the author's experience with writing Keras / Tensorflow code was not enough to implement this during the project time period. The core idea is to embed the VST host code into a layer in the computational graph that takes predictions as input, and produces a spectrogram as an output. Attempted work involved the use of Keras Lambda functions and TensorFlow's py_func to embed the code, however an incomprehensible error was returned.

Chapter 5

Results

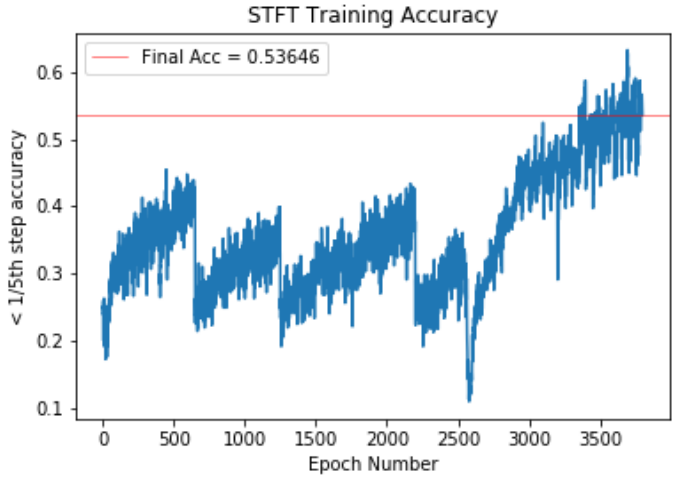
To reasonably compare the spectral representations, the custom accuracy metric that was implemented had a threshold that took into account the resolution at which the patch parameters were sampled at. Therefore for these experiments, at a patch sample rate of $\frac{1}{4}$, if the network's prediction is within $\frac{1}{4}$ of the target setting, then it is classified as being correct. While there could still be some variance in the actual output sound, there is often not much modifying to the target patch required in order to fully achieve the target sound.

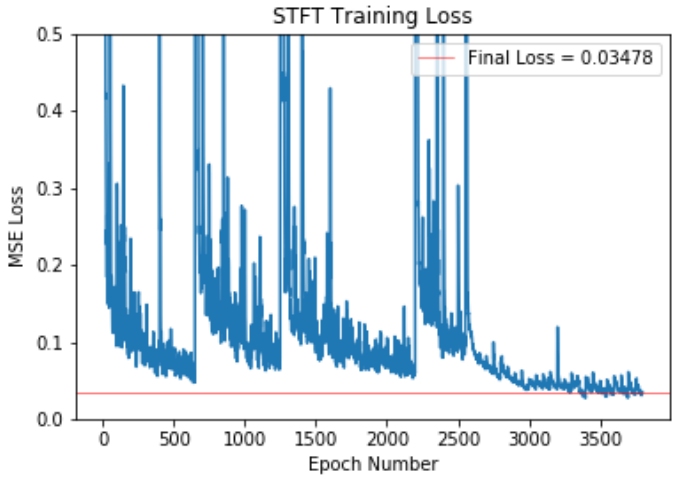
All experiments were conducted using the Convolutional Neural Network outlined in **Implementation**, with each batch of samples having 50 epochs to train the model on. Results were gathered using the Jupyter Notebook environment on a computer running Linux Mint 18 with a CUDA-capable NVidia GTX1050ti graphics card.

Performance is evaluated in a number of ways. Firstly the training history is taken into account, as some configurations reach peak performance long before other configurations. Next, real-world performance is evaluated. A dataset of 1000 randomly generated patches has been created to fairly compare each representation against each other. The restriction of a quarter turn has been lifted, and each parameter can take an integer value in the range 1-128. This is to make the test more realistic for the trained models, and to evaluate how well they perform.

5.1 Short Term Fourier Transform

Training The Network



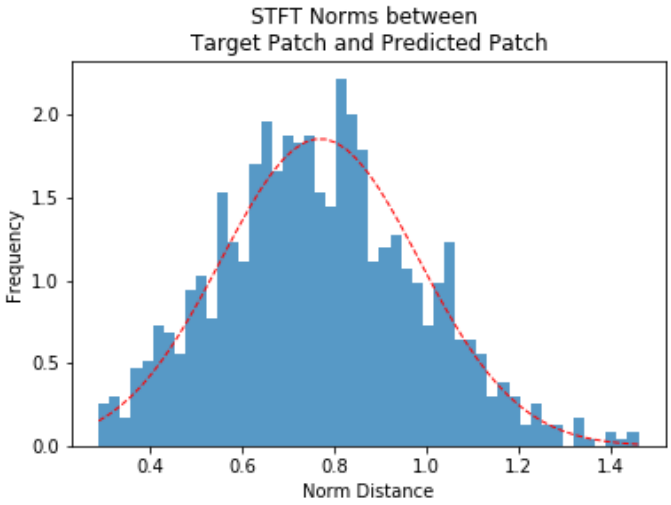


The STFT representation took 1hr32mins to fully train on the data that was provided. As shown in the training loss and accuracy graphs, the performance drops at the start of each batch, but quickly recovers and improves at the end of each stage in training. The final loss of 0.035 is the MSE between the predicted patch and the patch given for training, and the accuracy shown of 53.6% is the rate at which the predicted patch is within the defined threshold of each predicted parameter being no further than 0.25 away from the target parameter.

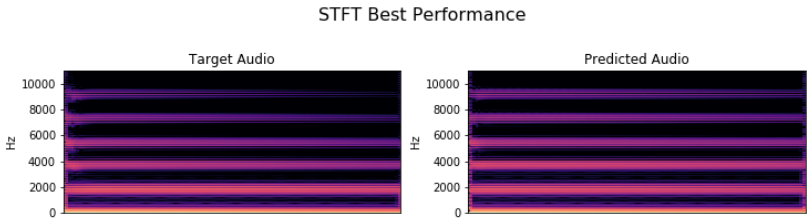
Performance

There are two ways to evaluate the real-world performance of the trained NN model, **Patch Settings** and **Audio Output**. The neural network used has been trained with the goal of trying to reduce the distance between a target patch setting and a predicted patch setting. By evaluating both of these factors, a clearer picture of practical application of the model is achieved.

The difference between each predicted patch and target patch has been calculated, and NumPy's linalg.norm* feature has been used to represent each predicted patch as a measure of total distance from the target patch. A distribution of these norms is shown in fig x.



The same idea has been applied to the target sound and predicted sound. Each predicted patch has been rendered, and the difference calculated for the corressponding target sound. For each difference, the norm is then calculated and used as a measure of the predictive performance in terms of real-world audio. The Worst, Best, and Median performance is presented below.

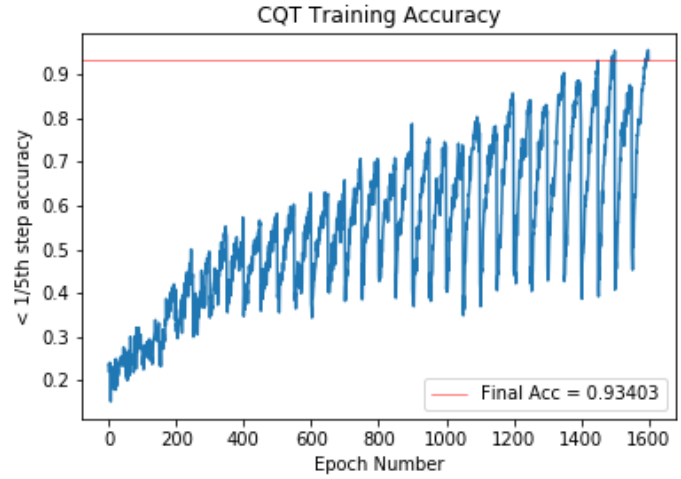


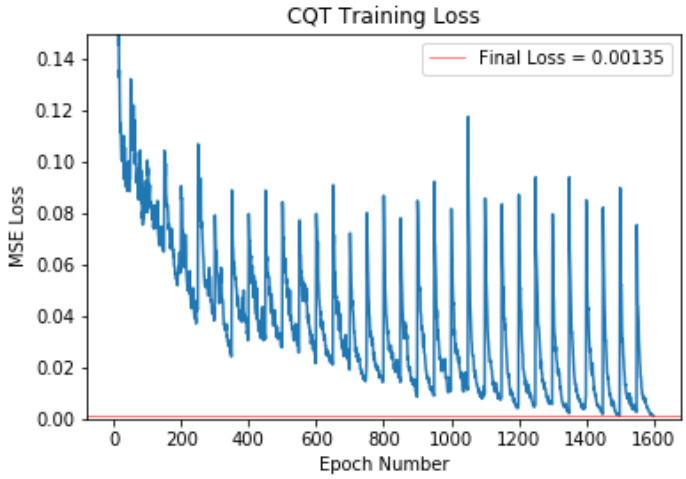
As shown, there is not much difference between the target and predicted sounds even for the worst case scenario. The model has a good representation of what sounds the synthesiser can make, and can predict fairly reliably the patches needed.

The Short Term Fourier Transform provides insight into the baseline performance of the system for this problem. Even though a relatively small subset of the sounds that the target synthesiser can make is used, it is shown that with an untuned data representation, the neural network architecture is able to predict target patch settings to a reasonable degree of accuracy. Given the worst case scenario shown here, it only requires a small modification of one parameter to achieve a sound that is almost identical to the target sound.
Given the random chance probability for this problem, the STFT gives reasonable performance and is enough to prove that the designed system is feasible to be used to reliably predict patch parameters.

5.2 Constant Q Transform

Training The Network

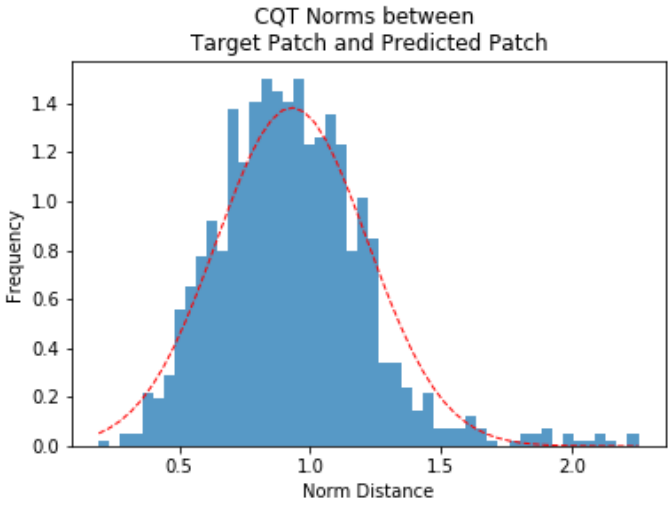




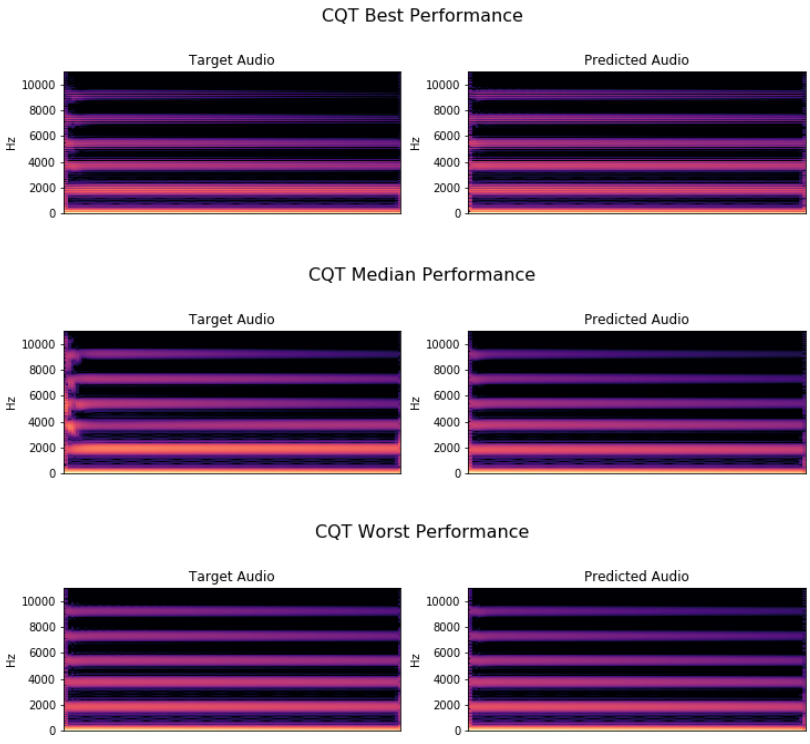
The CQT representation took 1hr12mins to fully train on the data that was provided. Again, the performance drops at the start of each batch which is to be expected, however high performance is reached much faster than the STFT representation. A final accuracy of 93% is very good, and probably can be attributed to the fact that the Nekobi synth has a lot of low frequency information that the CQT helps to represent.

Performance

Patch Settings



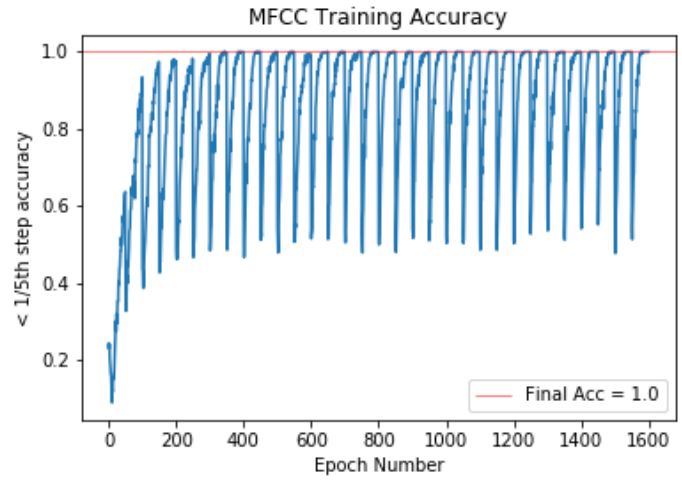
Audio Output

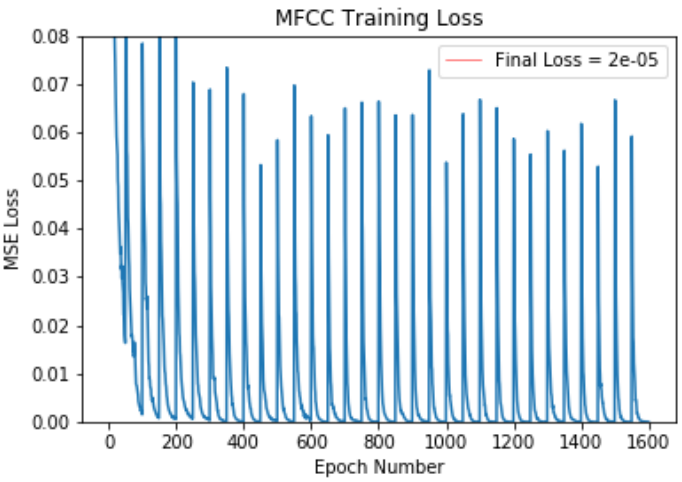


Again the worst performing patch still has reasonable accuracy which means a target sound can be reached with some small adjustments from the user. This is more practical when a more complex sound is trying to be achieved, but for the purposes of this project, it shows that the system works reliably.

5.3 Mel Frequency Cepstral Coefficients

Training The Network



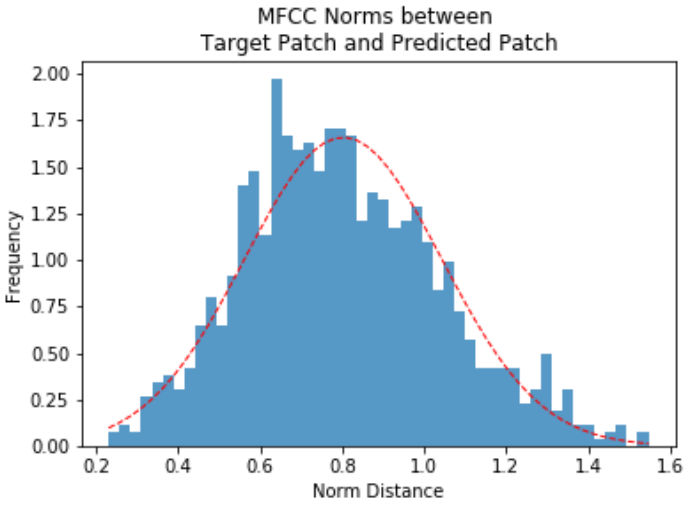


The MFCC representation took 53 mins to fully train on the data that

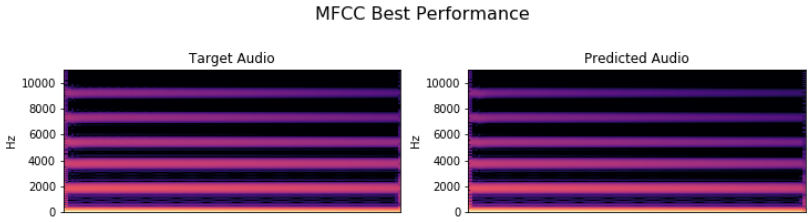
was provided, probably due to the lossy representation of the data that is used. The MFCC reaches 100% Accuracy within 400 epochs during training, which is much higher performance than both the STFT-based spectrograms. Additionally, a very low loss is achieved at around 400 epochs, ending on 0.00002 MSE loss. This is exceptional performance, and would have to have been reduced by including dropouts in the network architecture if we weren't trying to overfit the data.

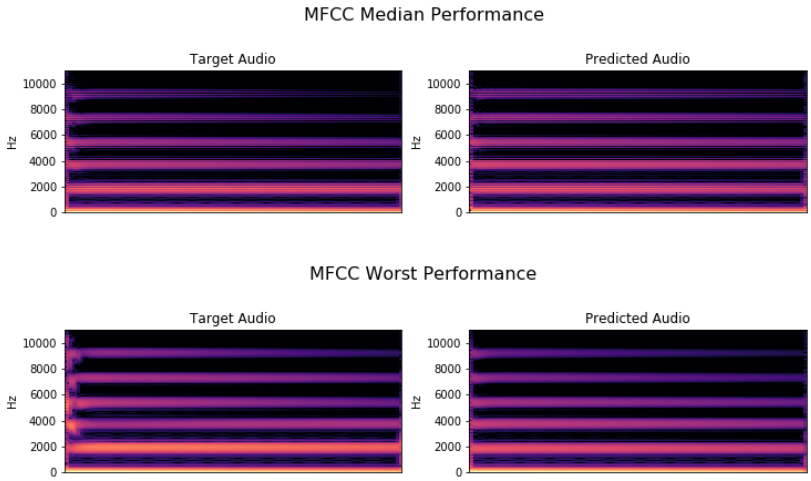
Performance

Patch Settings



Audio Output





Again, even the worst performing audio render is still very accurate.

5.4 Summary

As expected the Short Term Fourier Transform was the worst performing representaiton out of the test group, but it still was able to predict correct patches with a higher accuracy than random chance, which indicated that the solution was viable. The Constant Q performed better, with a higher accuracy rate which is to be expected as a lot of frequency information for this particular bass synthesiser is detailed in the lower bands which the STFT tended to bunch together.
An unexpected outcome of this experiment is that the Mel-Frequency Cepstral Coefficients representation of the data performed significantly better during training than both the Constant Q representation and the Short Term Fourier Transform representation, reaching an accuracy of 100% very early on in training. This was not anticipated, however it is not unusual due to the success enjoyed by using MFCCs as the data representation in other Deep Learning applications such a voice recognition, or other musical analysis.

5.5 Further Developments

More research and development has been completed in an attempt to increase network performance for more complex synthesisers. The MFCC

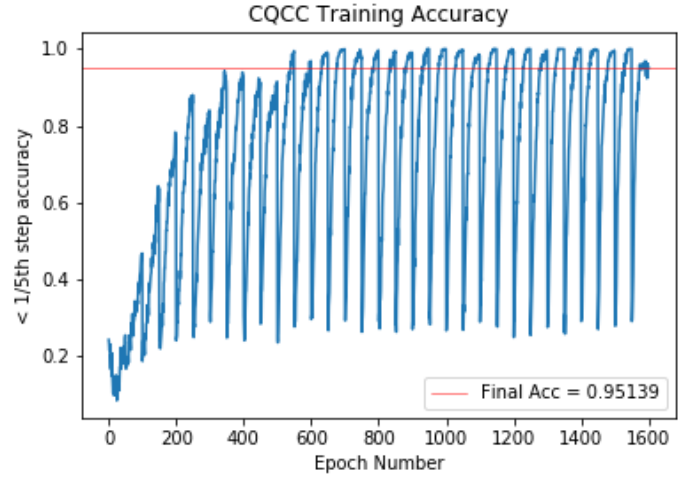
data representation has been used to develop further as it's performance in the previous experiment was overall better than both the STFT and CQT representations.
A shortcoming of the MFCC as a data representation in this task is that complex audio synthesisers produce frequencies that are equally important to the overall sound across the audio spectrum. The Mel-Frequency scale does not accommodate this as it is designed to represent perceptual pitch to the human ear. In typical musical or speech analysis tasks this is not an issue, however, as the system is trying to predict patches that are as accurate to the source sound as possible, using the Mel-Frequency scale is potentially an issue, especially when complex low end frequncies need to be considered.
In order to address this issue whilst maintaining performance of the neural network, a fairly novel data representation known as Constant-Q Cepstral Coefficients has been used to represent rendered synthesiser audio to the neural network model.

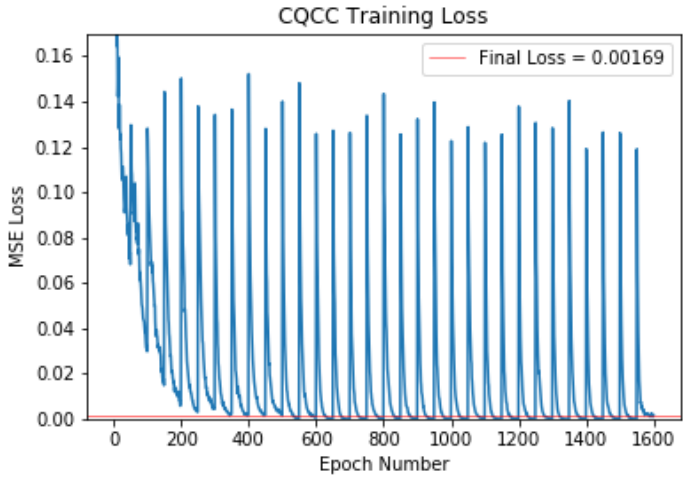
Constant-Q Cepstral Coefficients

As outlined in *Computer Identification of Musical Instruments using Pattern Recognition with Cepstral Coefficients as features*, Constant-Q Cepstral Coefficients are an effective way of representing musical data that addresses the shortcomings of the MFCC for this application.
They are produced in essentially the same way as Mel-Frequency Cepstral Coefficients, except that a constant Q power spectrum is used rather than a Mel-Frequency scaled power spectrum to calculate the DCT from. A Python adaptation of Delgado[22]'s MATLAB implentation for calculating CQCCs has been used to represent rendered sample data for the CNN.

5.6 Constant-Q Cepstral Coefficients Results

Training The Network



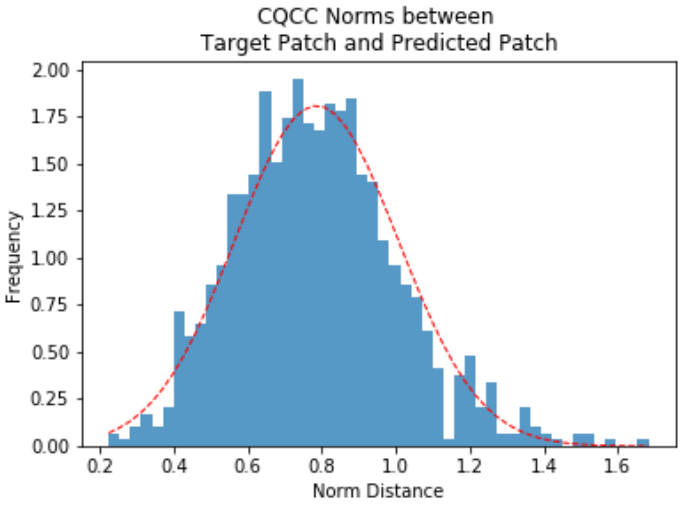


The CQCC representation took 55 mins to fully train. This extra time taken can possibly be attributed to the extra computation required to calculate the CQCC. The MFCC essentially takes the DCT of the Mel-Scaled Spectrogram, whereas the CQCC needs to be resampled before calculating the DCT. This is because the DCT requires an orthoganal frequency basis in order to work correctly, however, the Constant-Q Transform results in a geometric frequency basis due to it's nature. Peak accuracy is reached a little later on in training, at around epoch 600, with peak loss being reached at

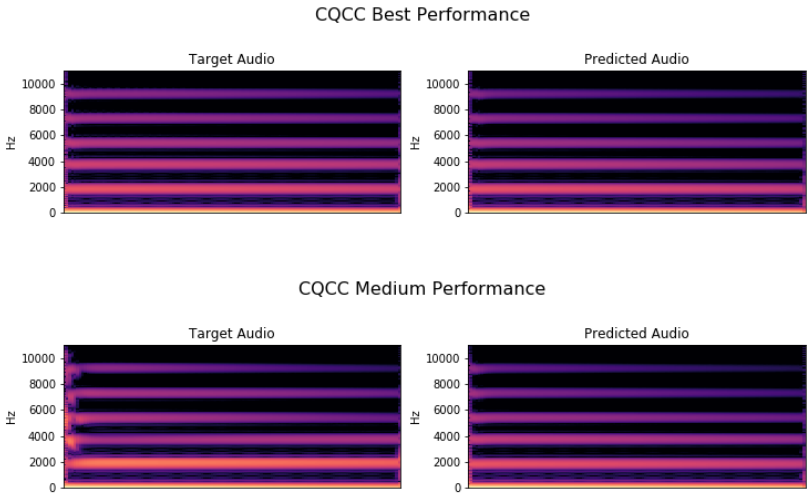
around epoch 400, the same time as the MFCC.

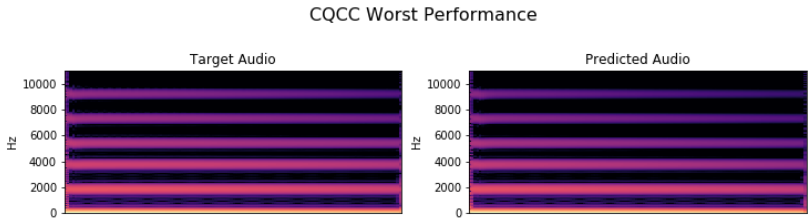
Performance

Patch Settings



Audio Output





5.7 Evaluation

The results have shown that all 4 data representations are viable for the network used, as even the worst case of the worst performing data representation still has reasonable real-world accuracy. All of the predicted patches are a very close approximation of the target patch. It was anticipated that the CQCC representation would provide a significant performance boost over the other data representations, however it appears that the neural network used had the largest effect on the accuracy. Early prototypes of networks were barely able to reach 0.5 MSE loss at full training, so the CNN architecture used for this task is appropriate.
Overall these results prove that the designed system solves the research question with reasonable accuracy, and in reasonable time.

5.8 Critical Appraisal

While this project has shown that it is possible to estimate a synthesisers parameters to a very close degree, it is clear that some modifications need to be made going forward in order to produce a more reliable system for more complicated synthesisers. The author's knowledge of DSP is not sufficient enough to reliably evaluate the accuracy of the Constant-Q Cepstral Coefficent algorithm that has been implemented, so it is possible that a more suited implementation of the algorithm would have more performance over the well-proven MFCC algorithm implemented by LibRosa.
It is also clear from the results presented that the data representation does not have as much of an effect on the accuracy of the predicted sound as other aspects of the system do, such as neural network layers and patch sampling resolution.
The current system gets most of the way there in terms of accuracy of predicting patches, however the fine tunable details are often missed out. Utilisation of a more low level framework such as using TensorFlow directly

would result in being able to implement more complex neural network architectures that serve the problem domain better in order to perfect the fine details of the patch.
These experiments have resulted in a reliable way of automatically programming the Nekobi synthesiser to reproduce a given sound with reasonable accuracy. However there are a couple of shortcomings that need to be addressed:

1. These experiments only considered timbre variance, not pitch variance.
2. The Nekobi synthesiser is not as harmonically rich as other synthesisers that are commonly available.

A critical aspect of this project was the choice of VSTi. All models and data representations have very good performance in predicting the sounds that the Nekobi synth can make, however, it is possible that the tools employed are overkill for such a simple synthesiser. A much more complex synthesiser should be tested with the network archtecture to truly find the strengths and weaknesses of each data representation. The most notable difference between the network models is the amount of time taken to reach a good level of accuracy. This should change if the parameter space were to be dramatically increased from 6 to 50 for example, and as should the spread of performance in patch prediction.

Chapter 6

Conclusion

The project's inital aims were to evaluate the feasiblity of using Deep Learning to reverse engineer an audio synthesiser's sounds. This project has shown that this is possible, and has explored some methods that improve the performance of a system that is developed to complete this.
The mean time for a model to train was roughly 1 hour, and every experiment ended up reaching peak accuracy early on. This suggests that with performance tweaks and more careful choice of sample patch data, the NN could potentially be trained in less time or on less powerful hardware in the same amout of time.
The rendertools library takes a few already existing concepts and combines them all to provide a good abstraction for preprocessing for this project. The general framework that has been designed is fairly flexible and was designed with modularity in mind for the sake of future work.
It was not expected that a relatively off-the-shelf Convolutional Neural Network would have such good performance in this task. It is promising that a simple architecture is able to predict patches in such a manner, as future work with more complex problems have a lot of scope to build off.

Perhaps a more useful approach to this project should have been to focus more on developing a more purpose built network such as the Encoder Decoder Architecture outlined in Further Work, however, this required transferring all Keras code over to it's equivalent TensorFlow code and implementing a significant amount of custom functions. While this was possible within the scope of the project, TensorFlow has a significant learning curve from Keras, and there is little to no documentation on how to approach implementing the ideas outlined. Ultimately however it has proven useful

to show that there is negligable difference between the data representations used when the network model is trained on a relatively simple synthesiser.

Overall this project has been a good learning experience, with unexpected problem domains in a range of different areas, and has been completed with plenty of scope left for promising future work.

6.1 Further Work

Future work should first focus on testing the current neural network model on more complex VSTis to evaluate a more realistic performance level of the system. Some more suggestions are outlined below:

Optimize processing pipeline

There are currently a couple of bottlenecks in the system. As such it was only feasible to let the network train for a certain amount of time. This constraint could be lifted if the bottlenecks were reduced. A potential solution is to reduce the amount of time it takes to render the audio data used in training, as currently almost half the time that the network is training for, the GPU is idle and is not training the model. This could be solved in two ways:

- **Parallelize Patch Rendering** As the VST host and VSTi are both written in C++, it is feasible to use CUDA in order to speedup the patch rendering section of the training session.
- **Asynchronous Patch Rendering** When the CPU has finished processing the sample data into time-frequency representations of the data, it could be feasible to start rendering the next batch of audio sample data whilst the 50 epochs of training are still being processed. This asynchronous model should be trivial to implement, but may provide significant performance gains and lead to optimal hardware usage.

Encoder Decoder Network Architecture

Unfortunately it proved non-trivial to attempt to implement this architecture using pure Keras, however, using a more flexible deep learning library such as TensorFlow or PyTorch, it may be possible to implement a WaveNet[23]-inspired autoencoder style network. The core idea is to use the CNN architecture described in Implementation as the Encoder to produce

the patch settings, and then instead of having the network calculate loss based on Mean Squared Error of patch settings, have the netwrok calculate loss based on the rendered audio of the patch settings it prediceted in the encoder half of the model. This requires making a decoder layer that takes patch settings as input and outputs a spectrogram as an output, and then compiling the whole model as an autoencoder style network. A suitable loss function should be binary cross entropy as the network will be attempting to match the output with the input. In practice this network may take significantly more resources and time to run, as each individual prediction needs to be rendered within the network, however it may ultimately result in a more accurate model of a given synthesiser. The encoder part of the model can then be used as the neural network used in production to make very accurate predictions of patch settings based off input audio.

Parameter Resolution Feedback Loop

As outlined in chapter 3, a problem with this topic is the question of how often to sample each individual parameter as each parameter has a variable non-linear response. One way to resolve this is to employ an algorithm that explores the sound space that a synthesiser can make, and uses a feedback loop to note the spaces in a parameter that have the most effect on a given sound. More thought is required to elegantly solve this problem.
Another potential way to solve this is to let the neural network decide what needs to be focussed on: during training, keep a running total of the individual error / loss of each parameter in the patch. After a batch of training is completed, evaluate the loss of each parameter and assign a higher resolution to the parameters which have a mean loss that is larger than the mean overall loss. This way the network provides a sort of feedback that is able to essentially interactively explore the sound space of the synthesiser. A problem with this approach however is that some parameters on some synthesisers can be switched off by other parameters - this would need to be taken into account, as a parameter that is switched off would have much higher loss than the other parameters in the patch, and focussing on this parameter would lead to wasted resources and even lower accuracy than expected.

Parameter Confidence included in NN predictions

Currently the neural network model is only able to output 1 value for each parameter. This doesn't leave any room for error, and doesn't accommodate

for situations where the network is 'undecided' about a few settings. The idea for this is to extend the current output space such that each parameter has 128 possible options. A percentage distribution can then be used to accurately display a range of predicted values.

Reverse Engineering Polyphonic Patches

A limitation of the VST host that was used is that only one note can be rendered at a time - this is known as being monophonic. Polyphony is where more than one note is played at once. There are a lot of polyphonic synthesisers available, and they often produce very harmonically rich sounds. An extension of this work would be to develop a system that can produce a high quality dataset with polyphonic representations, and a network that would be able to discern the patch settings to reproduce the timbre that is created by using these notes. A potential way of separating timbral and pitch information is to use a metadata style network, where note information is given to the network as a separate vector and applied in a network merge. This would essentially ensure that the network can focus on reproducing appropriate timbre for both monophonic and polyphonic performance data.

Transfer Learning for Synthesiser Patch Extraction

It is very rare to find isolated samples of synthesiser sounds in order to be used as training data for this network. One direction that further work could take would be to apply the principles of this project to a network that is able to pick out an instrument in a scene and replicate the timbral features for a given synthesiser that matches the instrument in that scene. An ideal system would be able to accommodate for other instruments in the audio, and would be able to hone in on one instrument in the mix and provide patch settings to emulate the sound that the instrument makes, or potentially sounds for every instrument in the mix, provided the synthesiser is sufficiently capable.

Chapter 7

Reflection

This project has been a fantastic learning experience in all sorts of domains. A considerable amount of this project has been spent researching the project's problem domains including DSP, Deep Learning, HPC, and a lot of failed attempts at putting ideas together in order to solve the individual problems that were presented at each stage of development.
Upon completion of this project I feel I have gained a good amount of practical understanding and experience in the field of deep learning that can be taken forward and applied to different problems without much of a transition. I also contributed to the development of Fedden's RenderMan VST Host library during the completion of this project, and as a result I have a more in depth knowledge of python-c++ interoperability, as well as how Steinberg's VST SDK works.
As a result of the work done in this project, I can comfortably say that I have developed a good work ethic for researching and learning about topics that are out of my depth. A few fairly niche concepts were required to be understood in order to have a good understanding of how to solve a particular problem encountered during the project.

I have also grown to appreciate the complexities of audio synthesisers and synthesis methods, and have strived to use them more as a fine tuned instrument in my music rather than as a tool to fill in the background.

Ultimately I am glad to have pursued this research area, and I am excited to see where the results of future work will lead to.

Bibliography

[1] Synthesiser Definition
https://www.britannica.com/art/music-synthesizer

[2] Roland TB-303
https://en.wikipedia.org/wiki/Roland_TB-303

[3] Yamaha DX7
http://www.vintagesynth.com/yamaha/dx7.php

[4] John M. Chowning. (1973), *The Synthesis of Complex Audio Spectra by Means of Frequency Modulation*
https://web.eecs.umich.edu/ fessler/course/100/misc/chowning-73-tso.pdf

[5] Yann LeCun, Yoshua Bengio, Geoffrey Hinton. (2015) *Review: Deep Learning*
http://pages.cs.wisc.edu/ dyer/cs540/handouts/deep-learning-nature2015.pdf
doi:10.1038/nature14539

[6] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton. (2012) *ImageNet Classification with Deep Convolutional Neural Networks*
https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-

[7] Musical Instrument Digital Interface
https://en.wikipedia.org/wiki/MIDI

[8] Virtual Studio Technology
https://en.wikipedia.org/wiki/Virtual_Studio_Technology

[9] Steinberg VST SDK
https://www.steinberg.net/en/company/technologies/vst3.html

[10] Hagit Shatkay. (1995) *The Fourier Transform - A Primer*
https://pdfs.semanticscholar.org/fe79/085198a13f7bd7ee95393dcb82e715537add.pdf

[11] Judith C. Brown. (1991) *Calculation of a constant Q spectral transform*
https://www.ee.columbia.edu/ dpwe/papers/Brown91-cqt.pdf

[12] Sadaoki Furui. (1981) *Cepstral Analysis Technique for Automatic Speaker Verification*
https://www.researchgate.net/profile/Sadaoki_Furui/publication/3176892_Cepstral_ana

[13] N. Ahmed, T. Natarajan, K.R. Rao. (1974) *Discrete Cosine Transform*
https://doi.org/10.1109/T-C.1974.223784

[14] Shikha Gupta, Jafreezal Jaafar, Wan Fatimah wan Ahmad, Arpit Bansal (2013) *Feature Extraction using MFCC*
http://aircconline.com/sipij/V4N4/4413sipij08.pdf

[15] Yee-King, Matthew and Roth, Martin *SYNTHBOT: AN UNSUPERVISED SOFTWARE SYNTHESIZER PROGRAMMER*
http://yeeking.net/pdf/roth_yee-king_ICMC2008.pdf

[16] *Project Jupyter*
http://jupyter.org

[17] Leon Fedden, Matthew Yee-King *RenderMan VST Host*
https://github.com/fedden/RenderMan

[18] Brian McFee, Colin Raffel, Dawen Liang, Daniel P.W. Ellis, Matt McVicar, Eric BattenBerg, Oriol Nieto. (2015) *Librosa: Audio and Music Signal Analysis in Python*
http://conference.scipy.org/proceedings/scipy2015/pdfs/brian_mcfee.pdf

[19] Chollet Francois and others. (2015) *Keras*
https://keras.io

[20] Yusuf Aytar, Carl Vondrick, Antonio Torralba (2016) *SoundNet: Learning Sound Representations from Unlabeled Video*
https://arxiv.org/pdf/1610.09001.pdf

[21] Judith M. Brown (1999) *Computer Identification of Musical Instruments using Pattern Recognition with Cepstral Coefficients as features*
https://doi.org/10.1121/1.426728

[22] Massimiliano Todisco, Hector Delgado and Nicholas Evans (2016) *A New Feature for Automatic Speaker Verification Anti-Spoofing: Constant Q Cepstral Coefficients*
http://www.odyssey2016.org/papers/pdfs_stamped/59.pdf

[23] avdnoord, sedielem, heigazen, simonyan, vinyals, gravesa, nalk, andrewsenior, korayk, (2016) *WAVENET: A GENERATIVE MODEL FOR RAW AUDIO*
https://arxiv.org/pdf/1609.03499.pdf