Parameter automation for granular synthesis

author: Karol Bakunowski

supervisor: Michael Zbyszyński

BSc (Hons) Music Computing
Goldsmiths, University of London

2019

Abstract

My research is about stuff.

It begins with a study of some stuff, and then some other stuff and things.

Acknowledgements

Acknowledge all the things!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Aims and Objectives

The initial aim of the project was to implement a machine learning solution to the task of granular synthesizer programming based on sound matching. Consequently building a tool that would assist musicians in creating desired sounds more easily, provided an audio input to the system.

Throughout the entire development process, the focus remained mainly on creating a well functioning, usable program, despite some shortcomings in the areas of synthesis, audio analysis and machine learning respectively. Integrating these three modules well took priority over designing more sophisticated solutions to each individual problem. This allowed for creation of a tool functioning in near real-time, with the possibility of extending the modules that contribute to the whole.

The methods here differ from other approaches in literature (cite some papers that have done this, eg. Matthew, Leon) primarily in that here, one standalone tool has been created, usable on it's own. Users have direct ability to program the synthesizer using input from the microphone, and the entire process is handled by two buttons. No prior experience with programming is required, and the only assumption made about the user is knowledge of basic synthesizer programming.

The main objectives therefore are:

1. To build a tool that is helpful to artists in the process of creating sounds

2. To challenge the interaction between an artist and a preset as a starting point to synthesis

3. To achieve a response that is not only stimulating to the user but also differs from simply

randomizing the parameter values

In measuring the project's success, the subjective sonic coherence and similarity of predicted outputs may be considered the best indicator (the human discriminator?). Undoubtedly, if the predictions achieved are satisfying to the user, the main goal of this project was achieved. Additionally, users' opinions about whether the tool would be usefull in their work process will be a good indicator of the project's success.

Along with that, a quantitative analysis, such as comparing the audio descriptors on target and predicted sounds should prove itself useful. It will allow to do a quite generalisable, and objective comparison of target vs. predicted sounds. This process will enable the evaluation of neural networks used for predicting the parameter values, and help in assuming whether users will find the program useful.

EDIT!!!!!!:

In (future chapters) I provide some critique on established techniques of assessment of these types of problems and ultimately conclude that some stuff in the best way of doing it.

1.1.1 Deliverables

In the inerest of achieving the above decalred goals, I propose a standalone program, that allows users to set parameters of a granular synthesizer based on a one second buffer of audio, generated form the microphone input.

The program could be arbitrarily split into three separate modules, in order to clarify it's basic structure. Namely, synthesis, audio analysis and machine learning.

The synthesis is implemented in the "Juce" framework. Audio analysis tools from the "Essentia" library are used to help create training datasets, as well as help make predictions. And lastly, an implementation of a Multilayered Perceptron feedforward neural network with the "Keras" API is presented. The "Frugally Deep" library is responsible for deploying the "Keras" model into C++ code, allowing for near-real time performance, and removing the need for communication between C++ and Python.

Resulting program is a standalone "JUCE" application, capable of performing granluar synthesis, extracting audio features, and predicting parameter values of the synthesiser based on microphone input.

Chapter 2

Literature Review

2.1 Problem background

2.1.1 Synthesizer programming

Programming synthesizers can be enjoyable, fun and gratifying, yet very often presents a big challenge. The amount of adjustable parameters on such a device can be overwhelming, and consequently the range of sounds that can be achieved is usually very extensive.

Granulation, or any other corpus based synthesis technique for that matter, presents what could be called a special case in this domain. Not only do the parameters determine the final output, but also the input sample has a tremendous influence over the sound. In a granular synthesizer, users are presented with choices ranging from determining the audio to be sampled, to the amliptude envelope for each grain. A huge array of possibilities.

Therefore the ability to make decisions about programming a granulator to achieve desired results has to come from either a place of certainty about what each parameter is responsible for combined with intuition about the instrument, or a place of experimental though, and somewhat random parameter value assignments.

Users fairly new in the realm of synthesizer programming may encounter issues creating sounds they desire. This is to be expected, as like in any other area, novices have to get through a rather steep learning curve, to achieve certain intuition, and skill. This, however, often prevents people from creating what they want, or makes them settle for less, consequently limiting their creative output.

On the other hand, experienced musicians are constantly looking for inspiration and new ways of interaction with the process of creating exciting sounds[8]. Furthermore, programming synthesizers by following a not clearly defined intuition, combined with the use of presets and a mixture of experimantal methods is a popular approach[15], especially considering the amount of tools available today, and the ease of acquiring them. Having too many VST plugins in a DAW is a common problem today.

There are of course all the people in between the complete novice, and succesfull artist. They may have some experience, and can follow their own intuition, yet are trying to find their own sound, and define their style of production. Making that process easier, consequently making the music making process more accessible would be a great contribution to the community (EDIT, BUT IDEA IS GOOD).

2.1.2 Sound matching

The problem of sound matching seems to be almost unexistent for humans. We are able to hum along a song we have heard previously, or replicate sounds of different objects, like cars, or animals, like dogs. This activity does not present any real challenge for us. It could even be said that such behaviour is taken for granted.

It has been around since forever, and it is mostly how children learn language. The word "onomatopoeia" desribes a process of creating a word that phonetically resembles, or suggests the sound it describes (SOURCE) and has been around even before language came aobut (CHANGE) (ADD SOURCE)!

Conversly, creating a specific desired sound on an instrument of any kind takes serious practice, and sacrifices. Musicians who are able to translate what they hear in their head, directly to an instrument, be it a physical one like piano or guitar, or a software based instrument, are considered viruosic. Therefore, the ability to replicate any sound in a synthesizer, based solely on an audio input seems like a very intuitive way of interaction, and possibly an inspiring way of working for musicians. Additionally, it would blur the line between a begginer, and a virtuoso in that specific area, allowing people with less technical expertise to create music they want to hear. Primarily, however, such interaction would be an immensely helpful, and time-saving scenario for musicians, allowing them to spend more time on creative work, taking over, at least partially the task of programming a synthesiser. An attempt of solving this problem is undertaken in this papaer.

In the next three sections I go over each area of the project, and talk about previous work that has been done on each aspect needed for such an interaction to be possible.

2.2 Audio descriptors

Because of the way audio is represented digitally, mimicking sounds is not as intuitive for computers as it is for humans. Algorithms that are capable of generating meaningful data about given audio signal are needed in order to describe sounds in different ways, and based on different assumptions. (add more about why these are needed, and pure audio singal can't be used?)

A great amount of these analysis tools exist, many of which are appropriate for musical applications[16]. A lot of them are easily accessible either through libraries in different programming languages, such as 'libriosa'[11] for Python, or 'Essentia'[9] for C++, or directly through different audio software such as 'Audacity'[1] or most Digital Audio Workstations today.

The early work done with a focus on sound matching was using mainly spectral features[14], such as short time Fourier analysis[10]. More recently, however, the Mel-Frequency Cepstrum Coefficients seem to be the prefferable descriptor[5][21].

2.2.1 Mel-frequency Cepstrum Coefficients

A more detailed explenation of the algorithm is provided in the "Methods" section in chapter 3, and only an attempt to confirm their validity for musical applications is made below.

The majority of work wih MFCC have been directed at the problem of speech recognition and synthesis. However, an impressive amount of work has been done to prove the usefullness of the algorithm in istrument recognition and Music Information Retrieval (MIR) systems.

Logan provides a credible justification for the use of the MFCC in musical modelling, supporting the use of the warped frequency scale and the Discrete Cosine Transform[12]. Casey et al. present an overview of the state-of-the-art content based MIR; the use of MFCC is promoted for the purposes of timbral description[3]. Eronen shows that the MFCC is the best single feature in a comparison of features for monophonic instrument recognition tasks[6]. Brown et al. present a study where cepstral coefficient prove to be the most effective for woodwind instrument recognition tasks[2].

Additionally, MFCC vectors have been shown to be an effective audio feature for sound matching tasks[22][7]

What is more, Terasawa et al. determined that a timbre space based on MFCC is a good model for perceptual timbre space[18].(more !)

2.3 Synthesis

Early research done as an investigation into automation of parameters in synthesizers based on sound matching focused mainly on Frequency Modulation synthesis[10], although more recently experiments on different synthesis techniques have been done[4][21], including a wide variety of VST plug-ins[22].

Yee-King et al. have shown that the more advanced synthesizer engine, the more difficult it is to predict the parameters, despitie prediction algorithms used[19]. The different algorithms will be disscussed in the next section.

However, this applies purely to the amount of parameters, and not neccessairly the type of synthesis used. Using this approach on corpus-based synthesis remains an untapped area of research, worth investigating[13].

2.4 Predictions

One of the first approaches to consider automatic synthesizer programming was described by Horner et al.[10] and used a Genetic Algorithm (GA) aimed at predicting settings for an FM synthesizer. GA was also used in many systems after[4], and remains a relevant optimasation technique for this task.

Yee-King investigated different optimisation techniques, including Hill Climbing, Genetic Algorithms, Neural Networks and data driven approaches[20]. Showing that the Hill Climber algorithm, and data driven approaches offer strong performance in this task[21].

The search based algorithms, such as Hill Climber or a Genetic Algorithm offer slow performace, and require much computetional power to come up with a result. Therefore, they do not seem to offer the ability of prediction in real, or near real time, making them an unappealing choice for user facing systems.

On the other hand, a modeling approach such as a neural network, offer near-real time performance once trained. The training process can dependt on many factors, such as the size of a dataset, and number of parameter to predict[20], but the ability of real time prediction, makes them a much more valid approach for user facing systems.

A more in depth exploartion of possibilities in this domain was done by Yee-King et al. who compared and considered GA, HC, MLP, LSTM, and LSTM++ algorithms[19].

It seems that most of this research focuses on reproducing the original input[17]. Perhaps more interesting and novel sounds could arise as an effect of bad performance, but is does not seem to be the desired outcome in most of the literature.

2.5 Alternatives to sound matching

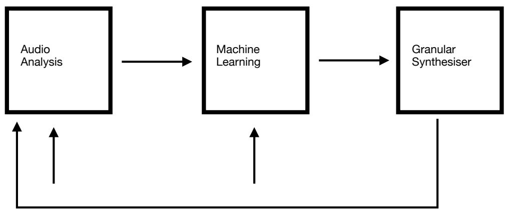
Another way of automating synthesizer parameters is a certain remapping to a 3D visual space...

Chapter 3

Methods

The general approach to solving the problem of automatic synthesizer programming is based on prior work done in this domain. The basic structure could be described like so:

Figure 3.1: Program flow



The box "Granular Synthesiser" in figure 3.1 represents the synthesis engine. It was implemented in C++, using the 'JUCE' framework (SOURCE).

"Machine Learning" stands for the predictions module, or more precisely the Multilayer Perceptron model built in "Keras", and later ported to C++ with the help of the "Frugally Deep" library.

The "Audio Analysis" square represents the aspect of the project responsible for extracting audio features from input sounds. This was done with the "Essentia" library, that allowed me to perform frame cutting, windowing, extracting the spectrum and finally MFCCs on any specified buffer.

Lastly, the "Audio Input" represents the microphone input, which is then analysed, and predictions are made on those values, in order to set parameters of the synthesizer.

3.1 Audio descriptors

The decision about which audio feature extraction algorithms were used was mainly based on previous research done in this domain. As mentioned in chapter 2 the MFCC algorithm is proven to be one of the most reliable desriptors available, and can not only describe the temporal qualities, but also be indicative of the changes that happen in the input signal[18].

Great results, and the easiness of quantative evaluation of the MFCC[19] were the main factors, outweighing different, potentially usefull feature extractors.

Based on that, it is the only algorithm used in the final version of the program, and further justifications of that decision are provided later in this chapter.

However, some amount of experimentation with different feature extractors was done, in order to see if certain assumptions made during early stages of the project were correct. Mainly, if certain descriptors could be used in direct correlation to different parameters of the synthesizer. Such as frequency content with "pitch" parameter, or rhythmical content with the the size of the grains.

3.1.1 Mel-frequency Cepstrum Coefficients

The validity of MFCC for musical applications was established in (CHAPTER 2). Below, I detail how the algorithm is computed generally, and provide my implementation in parallel to the explenation.

The spectrum, or a representaion of a signal in the frequency domain serves as a foundation for the MFCC algorithm. Conversion of a signal from time to frequency domain, can be done with the Fourier Transform algorithm. It can dissect a signal to it's most basic sinusoidal components, therefore determinig how much of which frequencies are present in the signal, allowing for the creation of a spectrogram.

The spectrum is calculated by performing the Fourier Transform on a windowed frame of a signal. (source, what if we took mfccs on the entire song lol?) Once that spectrum is computed, it's powers are mapped into the mel scale, using triangular overlapping windows. Next, logs of the powers at each of the mel frequencies is taken, creating a list of these values. Then, the discrete cosine transform of that list is computed. The MFCC are the ampliduted of the resulting spectrum.

To summarize, the Mel-frequency cepstrum is a representaion of the short-term power spectrum of sound, based on a linear cosine transform of a log power spectrum on a nonlinear mel scale of frequency (WTF, CHANGE AND ADD SOME REFERENCE TO THE DESCRIPTION). In turn, the MFCC, are coefficients that collectively make up the Mel Cepstrum.

There can be variations on this process, such as differences in the shape or spacing of the windows used to map the scale, or addition of dynamics features such as delta and deltadelta coefficients.(ADD SOURCE AND CHANGE - total ripoff)

The biggest advantage of this algorithm is that in the MFC the frequencies are spaced in a way which approximates the human auditory system more closely than the cepstrum used in a Fourier Transform.

Access to this algorithm is relatively easy, when using Essentia, because of it's high level API. It allows for arranging algorithms in a sequence, and connecting them with share inputs and outputs. Therefore, to compute the MFCC in Essentia, one has to first compute the spectrum, and it's output serves as an input to the MFCC algorithm, that computes the process described above.

The buffer on which MFCC are calculated is created in the following manner: sratatatatata

```
// computing the essentia algorithms
if (audioFeatureExtraction.playbackBuffer.index <
     audioFeatureExtraction.getLengthOfBuffer())
{
    const auto* channelData = bufferToFill.buffer->getWritePointer(0,
         bufferToFill.startSample);
    for (auto i = 0; i < bufferToFill.numSamples; ++i)
    {
        audioFeatureExtraction.pushNextSampleIntoEssentiaArray(channelData[i]);
    }
}
else
{
    //randomParameterWalkthrough();
}
```

The Fourier Transform is computed on frames of the signal, making the assumption that what is inside of that frame is a single period of repeating waveform. Most sounds are "locally stationary", meaning that over a short period of time they really do look like a regularily repeating function, which approximates enough for the Fourier Transform to work. It leads, however, to a situation where the endpoints of each frame are discontinuous, creating "spectral leakage". In an effort to reduce this effect, the technique known as overlapping is applied. The consequtive frames are overlapped by a certain amount of samples known as "hopsize". This allows for taking an average of multiple overlapping frames, and consequently a better representation of the time domain signal.

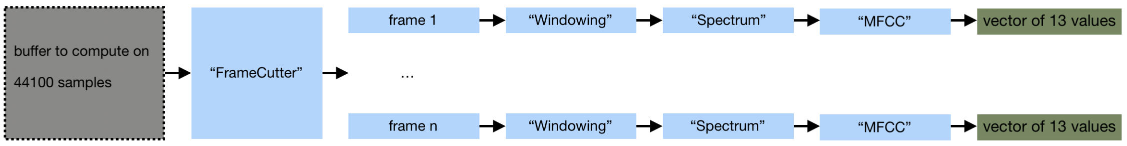
Lengths of both the frames, and the size of the hopsize vary, and depend on application. However the standard values are assumed to be 2048 samples and 1024 samples consequtively.

Here is how the frames, windowing, and overlapping are implemented using Essentia:

The process of computing that is as follows:

- Firstly, an audio buffer on which the computation will be performed has to be specified. Here, one second of sound is written into a buffer (C++ vector), that then gets passed to the first algorithm
- 'FrameCutter' is where the buffer first arrives. This function splits the buffer into multiple frames, as dictated by the parameters. Then, each frame, separately is sent further down the chain, with the help of a while loop
- Each consequtive frame of the buffer arrives at the 'Windowing' algorithm, which essentialy smoothes out the edges of each frame.
- Once the windowing is applied to the frame, the spectrum of the input signal is calculated.
- Finally, the Mel-Frequency Cepstrum Coefficients are calculated for the frame

Figure 3.2: logic of the MFCC computation



Let's assume we are trying to analyse audio input which is 10 seconds long. First, we have to split the signal into an appropriate amount of frames on which the Fourier Transform will be calculated. Each of these frames will be windowed, and overlapped.

With the values stated above our input would be split into frames of size 2048 (approximately x ms.), each frame 1024 samples further than the previous one.

For each frame, MFCC are calculated based on the extracted FFTs, as described above. The result for one frame is a vector of 13 floating point values, each corresponding to a different range of frequencies. (talk about why 13, because default essentia values?)

At a sampling rate of 44.1kHz, and a buffer size of 512 samples, this corresponds to exactly 45 vectors of 13 float values for one second of sound. Meaning that our hypothetical sound of length of 10 second would comporomise of 450 MFCCs.

Figure 3.2 in C++ code:

```cpp
void AudioFeatureExtraction::computeEssentia()
{
    while (true)
    {
      frameCutter->compute();

      if (!frame.size())
      {
          break;
      }
      if (isSilent(frame)) continue;

      windowing->compute();
      spectrum->compute();
      mfcc->compute();

      pool.add("lowlevel.mfcc", mfccCoeffs);
    }
    mergePool.merge(pool, "append");
}
```

This process is repeated for each frame, which at the sampling rate of 44.1kHz, and a buffer size of 512 samples, with frame size of 1024, and hopsize of 512 samples equals to exactly 45 frames per second.

As seen above, the results of the computation are written into "Essentia's" pool datastructure. Which in turn is written into a JSON file after a specified interval.

The process for audio feature exraction for input differs slightly, mainly in that the different algorithms with different inputs and outputs are defined and go through different functions.

3.1.2 Spectral Flux

Several different audio features have been experimented with, especially in the quest of finding universal ones, that could be applied to any source file in the granular synthesizer. One of such features is onset detection. It's implementation consists of the spectral flux algorithm, thresholding and peak finding, in order to estimate how many peaks, or onsets are in a specified buffer. (more description, or is this useless to talk about?)

However, during testing, which is described in chapter 4, such a connection was not found, therefore this project solely depends on MFCCs. MORE CONTENT, CITE PAPERS (not specifically used for sound matching tasks i think) In an effort to explore different algorithms, and the idea of assigning one feature to one parameter, exploration into the onset detection was done. Onset detection, which tries to estimate how many ‘peaks’ there are in a signal is a very useful algorithm to estimate how much rhythmical content there is present.

It can be achieved with an algorithm called “Spectral Flux”. It compares consequtive spectra, determining how much change has happend, producing a float value corresponding to it. By thresholding this change, based on the mean value of the flux, floats corresponding to the onsets in a signal can be derived (source, source).

3.2 Predictions

3.2.0.1 Definitions and Formulations

One of the main implicit requirements for this project is the ability to predict parameters in real, or near-real time, considering it is a user facing program. Literature shows that currently the best way of achieving that is through neural networks[19]. As mentioned in (CHAPTER 2), they take a considerable amount of time to be trained. Yet, once trained they can compute results in near real-time. Based on this assumption, neural networks were the only optimization algorithm used in this project.

Alternatively, in order to try and generalise the predictions for different input samples for the synthesizer, instead of just one the predictions could have been devided between different parameters. This approach should allow for creation of mappings between different parameters and audio descriptors. However, due to the fact, that no correlation was found between spectral flux and grian size parameter, possibly the most obvious bet at the beggining of the project, this concept was dropped. Additionally, I was not able to find any literature that would suggest such correlations. Therfore, and “easy route” was taken in order to achieve results quicker, in the spirit of achieving one fo the main goals for the project - to build a tool capable of parameter prediction in real time. Using an already established technique as suggested by Yee-King et al.

A neccessary step for training any neural network is the creation of a dataset, linking labels, in this case the MFCC vectors, with features to be predicted, in this case parameters of the granular synthesizer.

First, I detail how the dataset was created, and it’s implementation. Then I summarize the nature of neural networks created, as well as their implementation.

3.2.1 Dataset

Initially, I have set out to create a dataset of every possible combination of parameter values, and their corresponding MFCCs. The synthesizer would play for one second, the MFCC of that second would be taken, and parameters would be incremented. I decided that each parameter would be incremented by a tenth of it's value, for each iteration, based purely on intuition that this would provide enough data.

However, even though there are only 5 adjustable parameters, this process would have taken more than 1000 hours (ca. 41 days) to create the dataset. It is certainly possible to complete such a process, yet seemed like an unneccesary step, especially considering that the performance of neural networks was not known at this point.

Consequently, a different approach was taken. Instead of incrementing the parameters, a random value was assigned to each, after every iteration. The random number was always between the unique minimum and maximum of each parameter.

This was done inside of the C++ JUCE synthesis code. A function responsible for setting a random value for the parameters was implemented using the JUCE::Random class. Every second, after each iteration, the class instance is outputting a random value between specified ranges, unique for each parameter. This results in independent changes. It is a pseudorandom number generator, for which seed was not specified, in order to get different values, if multiple datasets were desired:

```
void MainComponent::randomParameterWalkthrough()
{

    int numOfSamples = 10000;
    Random rand = Random();

    audioFeatureExtraction.computeEssentia();

    audioFeatureExtraction.paramPool.add("parameters.position",
        audioFeatureExtraction.count);
    audioFeatureExtraction.paramPool.add("parameters.duration",
        audioFeatureExtraction.duration);
    audioFeatureExtraction.paramPool.add("parameters.spread",
        grainStream.filePositionOffset);
    audioFeatureExtraction.paramPool.add("parameters.numberOfGrains",
        audioFeatureExtraction.streamSize);
    audioFeatureExtraction.paramPool.add("parameters.pitch",
        grainStream.pitchOffsetForOneGrain);
    audioFeatureExtraction.mergePoolParams.merge(audioFeatureExtraction.paramPool,
        "append");

    audioFeatureExtraction.clearBufferAndPool();

    audioFeatureExtraction.count = rand.nextInt(Range<int>(1, grainStream.getFileSize()));
    grainStream.setFilePosition(audioFeatureExtraction.count);
    audioFeatureExtraction.duration = rand.nextInt(Range<int>(10, 1000));
    grainStream.setDuration(audioFeatureExtraction.duration);
    grainStream.filePositionOffset = rand.nextInt(Range<int>(0, 50000));
    audioFeatureExtraction.streamSize = rand.nextInt(Range<int>(1, 10));
    grainStream.setStreamSize(audioFeatureExtraction.streamSize);
    grainStream.pitchOffsetForOneGrain = rand.nextInt(Range<int>(-12, 12));

    audioFeatureExtraction.count2 += 1;
    cout << "round:" << "/t" << audioFeatureExtraction.count2 << endl;

    if (audioFeatureExtraction.count2 == numOfSamples)
    {
        audioFeatureExtraction.mergedMFCCs->compute();
        audioFeatureExtraction.mergedParameters->compute();
        changeState(TransportState::stopping);
    }
}
```

After each iteration, the MFCCs are computed, and added to the dataset. Then parameter values are added to the dataset. Next, the temporary vectors holding the values, as well as all of the Essentia algorithm's inputs and outputs, and the audio buffer used for computation are cleared.

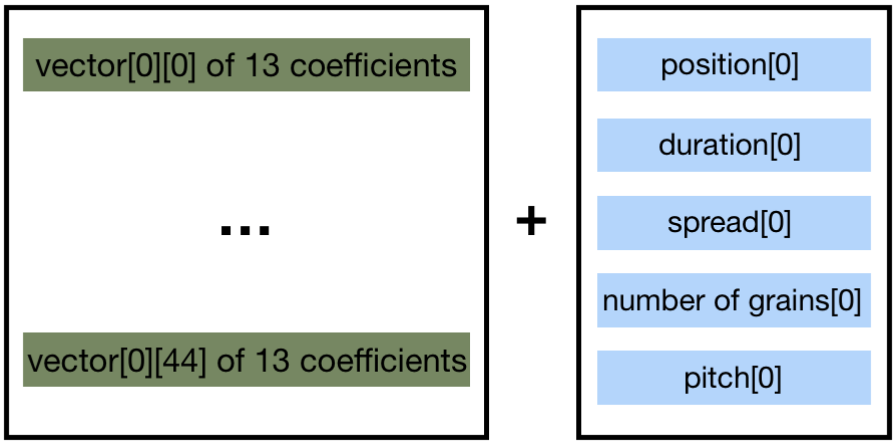
New parameter values are then set, and the count for number of iterations is incremented by one. When count reaches a specified number, the .JSON files are produced, and the output stopped.

Three datasets were created, with 1000, 10000, and 30000 instances. Each instance is made of a 2D MFCC vector, of size 45 by 13, and a vector of parameter values, as shown in figure 3.3.

Then, two arrays are created, which contain the audio analysis results, and parameter values. In order to conveniently export these results as .JSON files, the arrays are appended to a 'Pool' data structure from the 'Essentia' library. 'Pool' includes an elegant solution for saving it's contents into a .JSON file with the help of 'YamlOutput' function, like so:

```
mergedMFCCs = factory.create("YamlOutput",
```

Figure 3.3: Example of one instance of the dataset


vector[0][0] of 13 coefficients
...
vector[0][44] of 13 coefficients
+
position[0]
duration[0]
spread[0]
number of grains[0]
pitch[0]

```
                          "filename", "MFCC.json",
                          "format", "json",
                          "writeVersion", false);
mergedMFCCs->input("pool").set(mergePool);

mergePool.merge(pool, "append");
```

The resulting dataset is imported into a Python script, in order to perform the training of different predictive models.

3.2.2 Neural networks

There are many different possibilities when it comes to the architectures and types of networks. It was shown that a Multilayered Perceptron, as well as Long-Short Term Memory network are both viable choices, with Bidirectional LSTMs having the best performance[19].

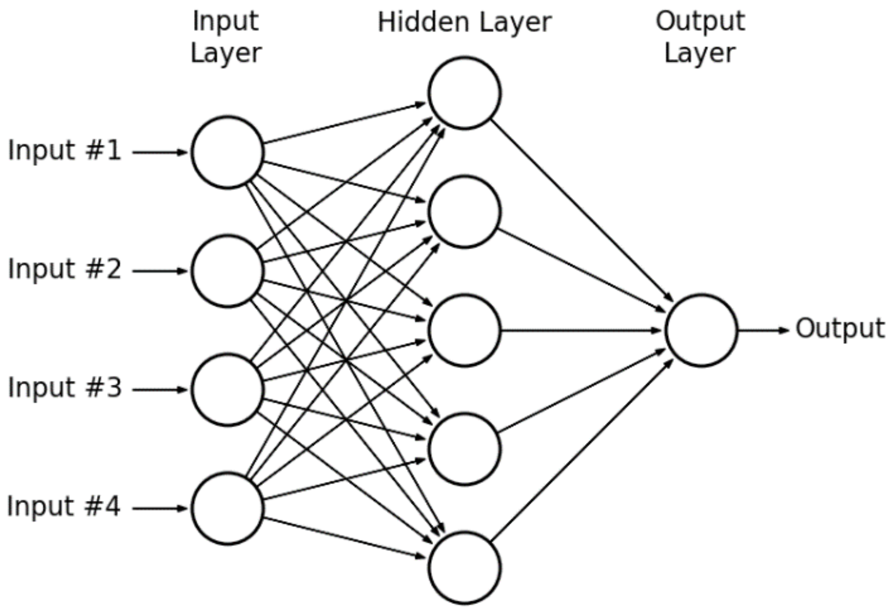
Due to time constraints, as well as the lack of expertiese in the field of deep learning, only two basic models were implemented. A feed forward Multilayered Perceptron, and a Long-Short term memory network. The popular "Keras" (source) library was used to create the architectures for both of these models in Python.

Next, I detail both implementations, as well as discuss the basic theory behind each network.

3.2.2.1 Multilayered Perceptron

A Multilayer Perceptron is considered to be a “vanilla” neural network. It consists of at least 3 layers of nodes - an input layer, a hidden layer, and an output layer. The data flows only in one direction, therefore it is often reffered to as a feedforward neural network. A minimum architecture for such a network can be expressed with a diagram, like so:

Figure 3.4: Minimal Multilayer Perceptron architecture



In a regression problem, such as one cosidered in this project, the nodes in the hidden layer(s) all use a nonlinear activation function. Possibly the most significant historically are the two “sigmoid” activation function, which essentialy maps any input between the values 0 and 1. (source?)

$y(v_i) = \tanh(v_i)$ and $y(v_i) = (1 + e^{-v_i})^{-1}$

Today, many different actiavtion functions can be used, with the most popular and universal being the “ReLu” rectifier:

$f(x) = \max(0, x)$

The training of an MLP utilizes a backpropagation (sorces from wiki) technique, which falls into the supervised learning category of machine learning. Backpropagation is commonly used by the gradient descent optimazation algorithm, to adjust weights of each node by calculating the gradient of a loss function.

There is a large number of loss functions used for this purpose They vary for different problems. For regression, the most commonly used are “mean squared error” and “mean absoulte error”.

All this is done automatically in “Keras”.

My implementation of the Multilayer Perceptron has the following architecture:

The data was split into training and test sets, where 80% served as the training, with the remaining 20% as test set.

The 45 by 13 MFCC vectors contained in the .JSON file, were imported into Python by using the standard json library. (source), and converted into numpy arrays, to allow further processing. Scaling of data between 0 and 1 is also done at this stage using the “MaxMinScalar” from the “sklearn” library.

```
import json
import numpy as np
from sklearn.preprocessing import MinMaxScaler

with open('MFCC.json') as f:
    d2 = json.load(f)

mfcc = np.array(d2["lowlevel"]["mfcc"])
scaler.fit(mfcc)
mfcc_norm = scaler.transform(mfcc)
mfcc_norm_reshaped = mfcc_norm.reshape(10000, 45, 13)
```

As seen in figure X.X, the input layer consited of 585 nodes. The second layer of 20 nodes, the third of 15, The amount of nodes in the output layer is equal to the number of parameters to be predicted, which in this case is 5. The 2D input vectors were flattened before feeding them to the hidden layers so that the entire second of MFCC values could be fed into the network at once. This model is defined as follows:

```
model = tf.keras.Sequential([
    layers.Flatten(input_shape=(45,13)),
    layers.Dense(585, activation='relu'),
    layers.Dense(128, activation='relu'),
    layers.Dense(64, activation='relu'),
    layers.Dense(5)])
```

Values of the hidden layers, are arbitrary, and were decided by experimentation with different hyperparameters. This is something that could be drastically improved, in order to get better results probably.

The activation function used for all hidden layers was the "ReLu" rectifier, and the "Adam" optimizer was used, with the mean absoulte error as the loss function. This translates to "Keras" code like so:

In the training of an early stopping function was used to prevent overfitting.

With this in place, the network was trained for 567 (change) epochs. The final loss function values were 0.9 for training set, and 0.12 for the test set.

The network was trained for X epochs, making use of an early stopping function, to prevent overfitting. The loss function values for training set is about 0.9 and test set 0.12.

3.2.2.2 LSTM

Leave for now, don't think I know enough to get into this business.

3.3 Synthesis

Many software based granular synthesizers exist, both in standalone (source, source) and VST form (sources). The implementations, and certain parameters etc. differ in each example. Nonetheless, this type of synthesis introduces an interesting problem in the contex of parameter prediction, which applies to any corpus based synthesis engine. The sounds created are heavily based on the sample fed into the synthesizer. Each parameter changes meaning significantly, once the input sample has changed.

Two possible solutions come to mind, when trying to overcome this problem.

Training a model, for an instance of a granular synthesizer with a perticular sample uploaded as the source file, therefore teaching the "agent" how to predict parameters for that perticular sample is one solution. This carries some obvious disadvantages, such as the inability to change the input samples, limiting the synthesizer to one. To improve this approach, multiple models could be trained, on a library of samples to give the possibility of predictions on a larger scale.

Another, would be trying to fit some universal analysis algorithms, that could decribe rhythym, pitch, density of sound etc. independently of the original sample. This way any sample could be 'molded' into what would resemble the original prediction target. As described in (METHODS-PREDICTION) section, this approach was disregarded for this project, because of lack of evidence in literature, as well as one failed attempt at creating such a correlation.

Concatenative synthesis is an alternative approach to this problem. It could be beneficial, as it would try to find 'grains' as closely resembling parts of the target as possible, and recreate it using little parts, almost like puzzles, that the algorithm thinks fit. However, that approach also has limitations, as it could only recreate the target sound out of the samples stored in it's database, therefore making the output biased. (cataRT source)

The decision of using a granular synthesiser for this program was made based on my personal preference for this type of synthesis, and the general lack of research focusing on corpus-based synthesis in this domain, as mentioned in (CHAPTER 2). Even though efforts were made to direct this problem perticularily at an instance of a granular synthesizer, the approach used could be applied to just about any other synthesis engine out there.

In the next sections I first explain the basic concept of granular synthesis, and then go on to explain my own implementation, used in this project.

3.3.1 Granular Synthesis

Describing granular synthesis in one paragraph is impossible without cutting some edges, and leaving out many intricacies of the technique. However, a summary of sorts, describing the main concepts behind the algorithm will be attempted here.

Granular synthesis is a relatively recent synthesis technique. However, it has been theorised for decades before it could have been implemented as a computer program. It' seed lies within general ideas about the nature of sound. Quantum physics has shown that sound can be atomically reduced to physical particles (WIENER REF). Isaac Beeckman was one of the first people to envision this physical form of sound (Cohen 1984), arguing that sound travels through the air as globules of sonic data.

These particles of data are imitated and magnified in this synthesis method, and are commonly reffered to as grains, hence the name of the technique itself. Combining these grains leads to the creation of different sonic events.

The original research by Gabor (source), was not intended at musical applications, until it reached the hands of Xenakis, who saw the potential of this technique. His first trials at granular synthesis included physically cutting magnetic tape into tiny pieces, and taping them back together in different order.

This research extended into the realm of computers with the help of Roads, who, after attending a seminar by Xenakis, began experimenting with this idea at the University of XXX. First experiments of implementing this technique took days to render a second of monophonic sound.

First realised real-time implementation of the granular synthesis technique was realised in 1986, by Traux (source).

From this point on, granular synthesis has slowly become available to a growing number of musicians and sound artists.

In recent implementations of granular synthesis, an audio sample is used as the basis from which grains are extracted. Providing a waveform for each separate grain. The grains are essentially small snippets of the original sample, rearanged together in different ways.

The most basic granular synthesiser requires some sort of an envelope over the grain, grain waveform, whether generated by an oscillator or taken from a sampled audio, and some grain spatialization. Here is a graph representing a minimal implementation, taken from "Microsound" by Curtis Roads.

It is an extremely versatile method of synthesis, capable of generating anything between pitched sounds, continous sounds, to segmentation techniques, all dependent on 3 main parameters. Grain envelope, grain waveform, and grain length.

3.3.2 Implemnting Granular Synthesis in JUCE

In order to meet the goal of creating a standalone, user-facing program, with all the functionality mentioned in (CHAPTER 1) an original implementation of granular synthesis was made. It allowed for the other modules to be implemented inside of the synthesizer, cosequently making the program independent.

A possibility of creating only the audio feature extraction, and machine learning modules, and integrating them with an external, already existing synthesiser exists, such as done by Yee-King et al. However this approach would clearly not allow for a creation of a "all in one" program, as stated in (CHAPTER 1).

The "JUCE" library served as the foundation for the synthesizer. It is one of the most established libraries in the professional audio community, being used by companies such as 'Cycling 74', and 'Korg' amog others (cite website?). On top of all things audio, it provides a convenient way of creating a graphical user interface, which was an important part of this project. Thanks to 'JUCE', the implementation was kept minimal, and elegant.

Even though the concept of granular sythesis is very difficult, the implementation here is kept fairly minimal, to restiric the amount of dimensions, reducing learning time for the neural networks, and consequently minimizing the time needed to create an appropriately sized dataset.

In reponse to these requirements only 6 adjustable parameters were implemented.

Below i go through all the parameters, and basic concepts of the implementation of granular synthesis in code.

General architecture is based on these programs also implemented in JUCE: links to github.

All the functionality for grains is contained in the "GrainStream" class, defined in the "Grain.cpp" file (code included in the appendicies).

3.3.2.1 Loading audio files

Loading audio files is the first step to acheive any output from the synthesizer, as it relies on sourcing waveforms for each grain from preexisting audio. Handling of this behaviour is made easy thanks to the "JUCE" framework:

```
void GrainStream::setAudioSource(AudioFormatReader& newAudioFile)
{
  int length = static_cast<int>(newAudioFile.lengthInSamples);

  //update grain parameters
  this->fileSize = length;
  this->samplingRate = static_cast<int>(newAudioFile.sampleRate);

  //clear the audio source and read new WAV
  this->AudioSourceBuffer.reset(new
                              AudioSampleBuffer(newAudioFile.numChannels, length));
  newAudioFile.read(this->AudioSourceBuffer.get(), 0, length, 0,
                    true, true);
}
```

"JUCE::AudioFormatReader" and "JUCE::AudioSourceBuffer" classes are used to handle this behaviour. The length of the sample, as well as it's sampling rate are also read here, and used to update global values. Used for different purposes, which will all be detailed later in this chapter.

3.3.2.2 Grain creation

First, let's define what a grain is in the program. A single grain is defined as a "struct":

```
  // this contains data about a specific grain inside the grain stream
  struct oneGrain
  {
      double grainDataCurrentSample[2] = {0.0, 0.0};
```

```
    int grainDataStartPosition = 0;   // starting sample for a specific grain
    int grainDataEndPosition = 0;       // ending sample of a specific grain

    double grainDataPitchScalar = 1.0f; // scalar value for randomized pitch offset
    double grainDataGainScalar = 1.0f;  // scalar value for rand gain offset

    bool grainDataIsFinished = true;   // boolean whether grain needs to be replayed

    juce::ADSR adsr;
    juce::ADSR::Parameters adsrParams;

    bool inRelease = true;
};
```

We can see that each grain has two doubles for one sample on each channel. It has a start and end position, as well as two scalars - one for pitch and one for the grain of each grain. One grain also contains a bool, to determine is a specific grain is currently playing. As well as an "adsr" envelope, defined with the "JUCE::ADSR" class, and another bool to determine whether the envelop is in the state of release.

Grains are created by specifing samples from the audio input (.WAV file loaded before synthesis) to be loaded into the "oneGrain" struct.

```
void MainComponent::getNextAudioBlock (const AudioSourceChannelInfo& bufferToFill)
{
    if (grainStream.grainStreamIsActive && record == false)
    {
        for (auto channel = 0; channel < bufferToFill.buffer->getNumChannels(); ++channel)
        {
            // get a pointer to the start sample in the buffer for this audio output
                channel
            auto* buffer = bufferToFill.buffer->getWritePointer(channel,
                                                               bufferToFill.startSample);

            // fill each grain struct with samples
            for (auto sample = 0; sample < bufferToFill.numSamples; ++sample)
            {
                buffer[sample] = grainStream.createGrain(channel);
            }
        }
     }
}
```

The “createGrain” function above is determining from where exactly to take those samples, based on the parameters set by GUI dials. The parameters will be explained separately in the next sections.

3.3.2.3 Position

First parameter users have direct access to is the position in an audio file, from which grains will be generated. Value of this parameter is determined by the “FilePos” dial.

It’s value serves as a parameter to the ‘setFilePosition’ function, which in turn communicates the starting position for each grain with the use of “filePosition” integer.

The following code is responsible for this task:

```
void GrainStream::setFilePosition(int startingSample)
{
  this->filePosition = startingSample - 1;
}
```

This value is then sent to the SETUP GRAIN function, where the “grainDataStartPosition” integer for each grain is set, consequently making the grain spawn in that position.

```
grain.grainDataStartPosition = this->filePosition;
```

3.3.2.4 Grain Size

Grain size parameter is and can be defined as the difference between the start and end position of the grain, in other words delta.

$\Delta y = y_2 - y_1$

Where y_2 is the end position, and y_2 is the start position of the grain. This delta is calculated by deviding the grain size in milliseconds (it’s duration), by 1000 and multiplying it by the current sampling rate, in order to convert that value to samples. The end position is then calculated by simply adding the delta to the start position.

The size of each grain varies between the minimum of 10ms and maximum of 1 second. It can be easily set using the Grain Size GUI dial. Value of the dial serves as a parameter to the ‘setDuration’ function:

```
void GrainStream::setDuration(int duration)
{
  //update duration and compute sampleDelta
```

```
    this->durationOfStream = duration;
    this->sampleDelta = static_cast<int>(this->samplingRate *
    (duration/1000.0f));

    for (oneGrain grain : grains)
    {
      grain.grainDataEndPosition = grain.grainDataEndPosition +
      this->sampleDelta;

      if (grain.grainDataEndPosition >= this->fileSize)
          grain.grainDataEndPosition = (this->fileSize - 1);
    }
  }
```

Simple but effective error handling is then implemented, by ensuring that the end position of a grain will never be bigger than the length of the sample.

3.3.2.5 Spread

The spread parameter is determining the size of an area around the file position, from which grains will be spawned. In other words, it randomizes the starting position for each grain, within a certain range.

This range is determined by the spread parameter itself, which in code is represented as "filePositionOffset".

```
Random rand = Random();

// Randomize the Starting Sample
grain.grainDataStartPosition = rand.nextInt(Range<int>
                                            (this->filePosition - filePositionOffset,
                                             this->filePosition + filePositionOffset));

// Prevent samples from being outside of file range
if (grain.grainDataStartPosition < 0)
    grain.grainDataStartPosition = 0;
else if (grain.grainDataStartPosition >= this->fileSize)
    grain.grainDataStartPosition = (this->fileSize - 1);
```

Again, the JUCE:Random class is used as the pseudorandom number generator. For each grain, the code above generates a new start position, in the range of current position plus and minus the offset determined by the spread parameter.

To give an example, let's assume that the position is set to 10000 samples, which would correspond to approximately 4 seconds into the file, assuming a samapling rate of 44,1kH. With the spread at 0, each consequtive grain would start at exactly 10000 samples, play for it's duration, and die, creating space for another grain to be spawned, again at the starting position of 1000 samples. Once the dial has a value bigger than 0, a random number is generated in the appropriate range, consequently creating a new start position for each grain. With a spread value of 200, each grain's new start position would equal to some number in the range of 9800 and 10200 samples.

This allows for each grain to have independent starting positions, creating a more interesting clusters, that do not sound so repetative and static.

The case of the spread parameter being zero:

```
if (filePositionOffset == 0 || (this->filePosition - filePositionOffset) <=0)
grain.grainDataStartPosition = this->filePosition;
```

3.3.2.6 Number of active grains

```
// vector containing grains
vector<oneGrain> grains;
```

The amount of grains available is predetermined arbitrarily by the range of values available within the parameter dial. These values are set to a range between 1 and 10, meaning that the 2D vector "grains" can contain up to 10 "oneGrain" structures, as defined in section 3.3.2.2.

The GUI dial value is sent directly into the function shown below, as the "size" parameter, which then decides whether to add or remove grains from the vector.

```
void GrainStream::setStreamSize(int size)
{
    if (size > this->grainStreamSize)
        addGrainsToStream(size - this->grainStreamSize);
    else if (size < this->grainStreamSize)
        removeGrainsFromStream(this->grainStreamSize - size);
}
```

Adding grains to the stream is done by simply adding a grain data structure to the 2D vector containing all the grains; the stream of grains, defined above.

```
void GrainStream::addGrainsToStream(int count)
{
    for (int i = 0; i < count; i++)
    {
        // append a grain the stream
        grains.push_back(oneGrain());
    }

    // update stream size
    this->grainStreamSize += count;
}
```

Following the same principle, the removal of grains is done like so:

```
void GrainStream::removeGrainsFromStream(int count)
{
    for (int i = 0; i < count; i++)
        grains.pop_back();

    this->grainStreamSize -= count;
}
```

Because of this, the interaction is kept quite simple, and the size can be decided with the use of only one dial. If the value of the dial if bigger than current size of the stream, we remove grains to match the value. Conversly, if the dial value is smaller, we add grains to the 2D vector ‘grains’.

3.3.2.7 Pitch

Each grain, as defined in the ‘OneGrain’ struct defined above, has a variable called grainDataPitchScalar, responsible for changing the pitch of grains independently. In the spirit of simplicity, however, in this perticular implementation, the pitch scalar is kept global, with the possibility of extension.

It is binded to the Pitch dial visible in the GUI, which changes the ‘pitchOffsetForOneGrain’ variable, which is then sent to the :

The grain.grainDataPitchScalar can therefore only have values between THIS and THAT, consequently either speeding up or slowing down the playback of each grain, and affecting the pitch.

3.3.2.8 Global gain

The global gain of the signal is simply controlled by the globalGain variable, which is used for scaling each individual sample sent to the audio buffer like so:

```
  sample *= static_cast<float>(globalGain)
```

The globalGain variable is directly controlled with the global gain dial in the GUI.

3.3.2.9 Sound output

The sound output is mostly handled by the JUCE library, with the use of the provided 'getNextAudioBlock' function. The audio block, or buffer is being filled with samples determined in the Grain class, more specifically, in the 'createGrain' function contained within that class:

All the functionality needed to determine all the previously mentioned parameters is here. What is being returned is samples, one by one with which to fill both channels of the buffer.

3.4 Integration

Some form of communication between all 3 main modules had to be implemented, in order to achieve real-time, or near real-time results. Whatever sound chosen to serve as the target for prediction has to first be analysed, in order to extract relevant MFCC. Then, these values have to somehow be sent into the neural network model, to get a prediction of parameters, for these MFCC. Then, these results have to be assigned to the actual parameters in the synthesizer.

Different approaches were considered for handling this communication.

During the first few iterations of the project, starting with the first prototype, OSC messages were used to send audio analysis values to Python, predict parameters there, and send the predictions back to C++.

This process however, was not optimal. It was relatively slow, not efficient, and required two separate programs to be running in order to give basic functionality.

The most intuitive way of doing that in my eyes, was to somehow integrate this entire process into the synthesis program, so that these processes could be easily accessed, and done within one window, with as much simplicity to the process as possible.

During the later stages of the production, I have found a satisfying solution to this problem. The "Frugally Deep" library (source). It allows for converting a model saved in "Keras" as a .json file, into XXX, and allow for it's usage directly in the C++ code. Consequently, allowing for real time predictions.

Consequently, the model in Keras was saved after training with like so:

With the toolset supplied by the 'Frugally Deep' library, it was then converted into XXX like so:

The data had to be sent into the model, and for that i'm using this thing:

the model was trained on scaled data, therefore to get the right predictions we have to scale it down in c++ also

another thing is that we have to scale it back up to the right parameter ranges, this is done with the values from MinMaxScaler copied from python

As seen above, with the file converted, only a few lines of code were needed to make use of the model directly in the C++ synthesis program, creating a standalone application, silmultaniously allowing for much quicker and simpler performance, without the need of any additional software running.

Chapter 4

Results and Analysis

As stated in chapter 1, the success of this project is dependent on three objectives, Firstly, the tool built has to be somewhat helpful to artists in creating new sounds. It will ideally challenge the interaction between a user, and presets, as a starting point to synthesis. Lastly, the results of parameter predictions must in fact bear some resemblance to the target sound, and be more than a mere randomisation of parameters.

As it is a user centred program, the users must feel like it indeed generates sounds that they find similar to the supplied input, and not less importantly, that the program is interesting to them, that an interaction it commences has some value, and to put it simply, it is fun to use. Additionally, the software must be intuitive, and easy to use, and ideally the user interface should not bring any confusion.

4.1 Methodologies

Firstly, it has to be ensured that all parts of the system are working correctly and as expected. In order to do that, some quantitative evaluation has to be done on each module of the system.

Namely, the correctness of extracted MFCC from the signal has to be confirmed. The synthesizer has to work as expected. The neural networks implemented have to give predictions in expected ranges, and finally the sounds achieved through them have to be evaluated to determine if they are if fact similar to the targets.

Qualitative evaluation had to be done to determine if users confirm the results of quantitative evaluation at least to some extent, as well as to determine if it achieves objectives #1 and #2.

More specifically, how easy the user experience is, unrelated to the quality of predictions.

The quantitative evaluation was conducted by comparing spectra of target, and predicted sound. Both generated with the instrument itself, as well as with input form the microphone.to provide a more realistic scenario. Euclidean distance between MFCCs was measured, as a metric of sound similarity.

The qualitative user testing, was made up of interviews, and a questionnaire. The users were briefly introduced to the concept of the program, and given most basic usage instructions. During the testing, users were allowed to comment freely, and after the session they had to fill in a questionnaire. This has allowed for immediate feedback, as well as a possibility to reflect upon the experience, and the ability to share those reflections later. Interviews were recorded, which allowed for immediate feedback, and the opportunity to investigate them later.

4.2 Audio descriptors

To determine the correctness of MFCC implementation, a simple test of the sizes of MFCC vectors is suffice. If the number of coefficients for one frame is equal to the expected value, and if the number of MFCC vectors in a second of sound is of expected length, then the features obtained should be correct.

Looking at the implementation of the MFCC algorithm, within "Essentia's" documentation, it is clear that the length of each vector containing MFCCs for one frame should be 13. Meaning that exactly 13 coefficient should be extracted for each frame.

Below, we can see that, in fact the length of one MFCC vector is as expected.

In turn, in order to check the validity of the legth of the vector containing MFCCs for one second of sound, we have to dive deeper into parameters for each consecutive algorithm computed on the signal before MFCC.

Assuming that the length of one frame, onto which our one second of sound should be cut into is 2048 samples, with a hopsize of 1024 samples, we should end up with 43.066 frames at the sample rate of 44.1kHz. This is of course impossible, and the most common way of dealing with such a problem is zero-padding. "FrameCutter" does this automatically and is zero-padding incomplete frames, or frames that are going past the end of the buffer. Therefore we end up with exactly 45 frames, or MFCC vectors per second of sound. The 44th frame is zero-padded. Noise is added to the 45th frame for equal extraction, as in accordance to the "siletFrames" parameter of the "FrameCutter" algorithm.

Here we can see the output of the 2D vector containg all MFCCs for a one second buffer.

We can see, that the implementation of the MFCC algorithm is therefore sound, as we get exactly 45 vectors each containing 13 MFCCs, as expected.

4.3 Granular Synthesis

The evaluation of the synthesis method is perhaps the most straightforward. All that has to be made sure of, is that all functions are working correctly, and each parameter does what it is supposed to. This is ensured by a simple evaluation by myself.

4.4 Neural Networks

In turn, the implementation of neural networks is possibly the part of the project which will have most obvious influence over the final results, and consequently this will decide whether predicted sounds are in fact similar, and how similar to the targets.

Other aspects of course also have influence over this, however it seems like neural networks will definitely determine the biggest aspect of this problem. If they fail, the predictions will be completely unrelated.

The thing that has to be taken care of first, is the dataset used for training. In order to ensure, that the data is coming into the model as expected, the numpy arrays are simply printed to the console.

We can see that the loss function is lowering in value, until the 500th epoch here, and after that stays relatively stable, leading to overfitting.

4.5 Sound similarity

4.5.1 Quantitative evaluation

In order to determine, if predictions made by the program are viable, and to judge how well they work, quantitative testing was performed in form of measuring a Euclidean distance between target and predicted MFCCs.

First, the testing was done on target sounds created with the granulator itself. Meaning, that technically, the sounds should be reproducible exactly, because they were made with the same instrument we are trying to recreate it on. I now describe the process for one iteration of such testing, and put results of one 10 instances in a XXX table.

First, random parameter values were set, by using the function for random parameter walkthrough. Once the parameters were set, one second recording of the synthesizer's output was recorder into a buffer, on which MFCCs were extracted. These values were saved, for future access. Then, based on those MFCCs, a prediction was made using the "Keras" model, which automatically set values of all parameters based on it's results. Then that was recorded into a buffer, and MFCCs were extracted. With these two vectors of MFCCs, a Euclidean distance was calculated between them, using the "scikit learn" library.

This entire process was later repeated with input from the microphone, instead of output from the synthesiser, to determine how the model behaves on data it has never seen before.

The reason behind comparing the Euclidean distance rather than the parameters themeselves was the fact that as a matter of fact the parameter values predicted can differ quite largely from the target values, as certain sounds are likely possible to achieve using more than one parameter setting. (source)

plan for this testing:

set some parameter values with a random button record a second into a buffer save MFCCs predict parameter values record prediction into the buffer save MFCCs

calculate euclidean distance between them in python

do that same thing with microphone input

additionally try to save them so that mfcc's can be taken. either into ableton (easiest) or write to a file (maybe time consuming)

4.5.1.1 MFCCs distance

4.5.1.2 Spectra

4.5.1.3 Random button vs. predictions

4.5.2 Qualitative evaluation

With the the technical assessment established, the testing could proceed "into the real" world, where it's validity among users would be tested. I believe, that with this software the assessment of how similar the predictions are to the input sounds is equally important as the quantitative results.

Additionally, their enjoyment of the experience, as well as their overall impressions are an important factor in the assessment of success of this project. During the tests, I was also hoping to gain feedback about things that I have overlooked during the production, and suggested ways of improving the user experience.

Therefore, several testing sessions with potential users of this software were conducted. All participants had some background in programming synthesizers, as well as background in music production. In order to ensure that they would somewhat resemble the user group this software is aimed at.

The session consisted of a brief explanation of what the project is meant to do, and a basic user guide, about how to use it. More specifically an explanation of how the "record" and "predict" buttons work. However, the information given was fairly minimal, and is only what was needed to introduce participant to this new concept of interaction. This was done in hope to later access the easiness of use, once the basic idea of interaction was established.

After the session, participants were shown a form to fill in, summarising their experience, and looking to establish a translation of their experience into quantitative data.

4.5.2.1 Results

4.6 User experience

Chapter 5

Discussion

5.1 Summary of Findings

To summarise what I concluded (via either qualitative or quantitative analysis) in the last chapter:

- The implementation of the MFCCs, neural network and the synthesizer are sound
- The MFCCs in sounds predicted on output of the synthesizer itself are very close to the target MFCCs.
- The Euclidean distance of MFCCs predicted on from the microphone input is much bigger than in predictions made from synthesizer output
- In both cases however, the distance in much smaller than compared to MFCCs achieved through pure randomisation of parameters
- Most of the testers agree that this tool has potential to be useful in the process of creation of new sounds
- 100% of the testers think that this process would be a viable alternative to presets
- On a scale from 0 to 5, the mean answer for similarity of sounds predicted is 3.8
- Most tester agree that the program was intuitive to use

5.2 Evaluation

5.2.1 Project objective 1: a helpful tool

The overall opinion of all the people that have tested the program seem to be that it would indeed be helpful for them in the process of creating sound. There has been some suggestions about using it with in a live environment, with a vocalist. Which leads to an interesting extension of the project as well as a possibility for more research.

It would be possible to pre-train a model on certain samples, before a live show, and then improvise with the use of that array of sounds, live.

Most respondents agreed that they would be interested in implementing such a program into their music production workflow. It would certainly need more polishing, and work, yet already at it's current state it can serve as an extension to music production arsenal.

5.2.2 Project objective 2: alternative to presets

In the questionnaire, 100% of participants agreed that this workflow is a viable alternative to present. One participant suggested, that this interaction brings the user into the position of control, and creates space for more control over the sound, than mindless scrolling through presets. To quote:

“This project puts a little bit more responsibility with the user by making them take a more active part in the creation of sounds rather than scrolling through presets. Whilst presets can be a good way of producing sounds I think that this makes users think more critically about what sounds they are wanting to make and then use the tools provided to replicate them.“

5.2.3 Project objective 3: more than randomisation

By measuring the distance between the MFCC vectors, as well as repeated listening evaluation on my own, and with different users, it is clear that sounds predicted by this system carry clear coorelation with the target sounds. Less so with samples from the microphone input, than the sounds created by the granulator, however still satisfying to the end user. The result of this could be clearly improved by spending more time tweaking the neural networks used, as well as experimenting with separation of parameters and audio features.

To summarize, the quantitative evaluation as well as user testing shows that result obtained with this software are more than just a randomisation of parameters.

5.3 Future Work

5.3.1 Implementing more sophisticated neural networks

5.3.2 Creating a database of samples that can be used

Chapter 6

Conclusion

In this report I presented a system capable of near real-time parameter prediction for a granular synthesizer based on sound input, consisting of three main modules:

- Granular Synthesizer, implemented using “JUCE”
- MFCC extractor in “Essentia”
- Neural networks in “Keras”

These come together to create a standalone program, with focus on user experience, and ease of use. As well as the overall correctness of predicted sounds.

The overall performance of the program could be drastically improved, by further enhancing the capabilities of each of the above mentioned modules. Yet, I have managed to create a program helpful in creating sounds, that has the potential to challange the popular use of presets, and achieve predictions that differ from simply randomizing values of the synthesizer, meeting all three of my project’s objectives, as determined by both quantative and qualitative analysis.

The predictions obtained from a microphone input, even in noisy environments, were rated well by testers, and achieved a fairly good score on the similiarity scale. Additionally, questionnaire respondents deemed the software intuitive and easy to use. The qualitative feedback received from the respondents suggests clear way of improving the software, yet suggests that there is real interest in using such tool in a studio environment.

A standalone synthesizer, capable of near real-time predictions, that integrates machine learning solutions is sufficiently different from alternatives, despite using realtively naive methods, and I submit this as a justification for the work’s value in relation to state of the art literature.

.1 Title of Appendix A

Bibliography

[1] *Audacity ® \textbar Free, open source, cross-platform audio software for multi-track recording and editing.* en-US. URL: `https://www.audacityteam.org/` (visited on 05/03/2019).

[2] Judith C. Brown, Olivier Houix, and Stephen McAdams. "Feature dependence in the automatic identification of musical woodwind instruments". en. In: *The Journal of the Acoustical Society of America* 109.3 (Mar. 2001), pp. 1064–1072. ISSN: 0001-4966. DOI: `10.1121/1.1342075`. URL: `http://scitation.aip.org/content/asa/journal/jasa/109/3/10.1121/1.1342075` (visited on 05/11/2019).

[3] M.A. Casey et al. "Content-Based Music Information Retrieval: Current Directions and Future Challenges". en. In: *Proceedings of the IEEE* 96.4 (Apr. 2008), pp. 668–696. ISSN: 0018-9219, 1558-2256. DOI: `10.1109/JPROC.2008.916370`. URL: `http://ieeexplore.ieee.org/document/4472077/` (visited on 05/11/2019).

[4] Palle Dahlstedt. "Creating and Exploring Huge Parameter Spaces: Interactive Evolution as a Tool for Sound Generation". en. In: (), p. 8.

[5] S. Davis and P. Mermelstein. "Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 28.4 (Aug. 1980), pp. 357–366. ISSN: 0096-3518. DOI: `10.1109/TASSP.1980.1163420`.

[6] A. Eronen. "Comparison of features for musical instrument recognition". en. In: *Proceedings of the 2001 IEEE Workshop on the Applications of Signal Processing to Audio and Acoustics (Cat. No.01TH8575).* New Platz, NY, USA: IEEE, 2001, pp. 19–22. ISBN: 978-0-7803-7126-2. DOI: `10.1109/ASPAA.2001.969532`. URL: `http://ieeexplore.ieee.org/document/969532/` (visited on 05/03/2019).

[7] Sebastian Heise, Michael Hlatky, and Jörn Loviscach. "Automated Cloning of Recorded Sounds by Software Synthesizers". en. In: *New York* (2009), p. 8.

[8] Matthew Herbert. *manifesto \textbar Matthew Herbert.* en-US. 2011. URL: `https://matthewherbert.com/about-contact/manifesto/` (visited on 11/18/2018).

[9] *Homepage — Essentia 2.1-beta5-dev documentation.* URL: `https://essentia.upf.edu/documentation/` (visited on 05/03/2019).

[10] Andrew Horner, James Beauchamp, and Lippold Haken. "Machine Tongues XVI: Genetic Algorithms and Their Application to FM Matching Synthesis". en. In: *Computer Music Journal* 17.4 (1993), p. 17. ISSN: 01489267. DOI: 10.2307/3680541. URL: https://www.jstor.org/stable/3680541?origin=crossref (visited on 05/03/2019).

[11] *Librosa*. URL: https://librosa.github.io/ (visited on 05/03/2019).

[12] *logan00mel.pdf*. URL: http://musicweb.ucsd.edu/~sdubnov/CATbox/Reader/logan00mel.pdf (visited on 05/11/2019).

[13] Kyle McDonald. *Neural Nets for Generating Music*. Aug. 2017. URL: https://medium.com/artists-and-machine-intelligence/neural-nets-for-generating-music-f46dffac21c0 (visited on 11/18/2018).

[14] Thomas J Mitchell and J Charles W Sullivan. "Frequency Modulation Tone Matching Using a Fuzzy Clustering Evolution Strategy". en. In: (2005), p. 12.

[15] *Oneohtrix Point Never on the process behind his latest leap into the unknown, Garden of Delete*. EN_GB. Feb. 2016. URL: https://www.musicradar.com/news/tech/oneohtrix-point-never-on-the-process-behind-his-latest-leap-into-the-unknown-garden-of-delete-633280 (visited on 11/18/2018).

[16] Geoffroy Peeters et al. "The Timbre Toolbox: Extracting audio descriptors from musical signals". en. In: *The Journal of the Acoustical Society of America* 130.5 (Nov. 2011), pp. 2902–2916. ISSN: 0001-4966. DOI: 10.1121/1.3642604. URL: http://asa.scitation.org/doi/10.1121/1.3642604 (visited on 05/11/2019).

[17] Kıvanç Tatar, Matthieu Macret, and Philippe Pasquier. "Automatic Synthesizer Preset Generation with *PresetGen*". en. In: *Journal of New Music Research* 45.2 (Apr. 2016), pp. 124–144. ISSN: 0929-8215, 1744-5027. DOI: 10.1080/09298215.2016.1175481. URL: http://www.tandfonline.com/doi/full/10.1080/09298215.2016.1175481 (visited on 11/19/2018).

[18] Hiroko Terasawa, Malcolm Slaney, and Jonathan Berger. "Center for Computer Research in Music and Acoustics (CCRMA) Department of Music, Stanford University Stanford, California". en. In: (2005), p. 8.

[19] M. J. Yee-King, L. Fedden, and M. d'Inverno. "Automatic Programming of VST Sound Synthesizers Using Deep Networks and Other Techniques". In: *IEEE Transactions on Emerging Topics in Computational Intelligence* 2.2 (Apr. 2018), pp. 150–159. ISSN: 2471-285X. DOI: 10.1109/TETCI.2017.2783885.

[20] Matthew John Yee-King. "Automatic Sound Synthesizer Programming: Techniques and Applications". en. In: (), p. 180.

[21] Matthew Yee-King and Martin Roth. "A Comparison of Parametric Optimisation Techniques for Musical Instrument Tone Matching". en. In: (2011), p. 9.

[22] Matthew Yee-King and Martin Roth. "SYNTHBOT: AN UNSUPERVISED SOFTWARE SYNTHESIZER PROGRAMMER". en. In: (), p. 4.