Robotics MSc Project

Harry Assayag (1670940)

Using Deep Learning to Solve for the Parameter Values of a Subtractive Synthesiser Sound

Supervisor: Peter Tino

School of Computer Science, University of Birmingham

August 31, 2020

UNIVERSITY OF BIRMINGHAM

Abstract

Sound synthesis is a complex tool which requires prior knowledge and technique to utilise properly. This paper investigates the use of deep learning to calculate the corresponding parameter configuration for a subtractive synthesiser when given an input audio sample. The positive effects of L2 regularisation were demonstrated through the analysis of the cross-validation set. After systematic tuning of the hyperparameters using a grid-search algorithm, the frequency spectra for a test dataset was analysed, but unfortunately, the model failed to generalise to different audio samples. Referring to the literature in this field provided a basis for an explanation of the model's potential shortcomings.

Keywords: **synthesiser parameters deep neural inverse problem**

Acknowledgements

I would like to thank my supervisor Peter for his frequent batches of wisdom and advice. His area of interests have fortunately overlapped with mine, making this whole project that much more enjoyable. My good friend Daniel has been a great help for bouncing around ideas, but more importantly lending a sympathetic ear to my frustrations.

Contents

February 3, 2025

1 Introduction

Sound synthesis is a relatively recent technique of producing artificial, electronic sounds. Synthesisers are incredibly powerful tools for creating and shaping audio, and are a staple in almost all music. These instruments are comprised of many different parameters which the user tunes to create their desired output sound. Finding a particular timbre and tone of sound can prove to be challenging to those people who do not possess an adept command of the instrument, so this project aims to provide a solution for those people.

Creating this tool would require solving an 'inverse problem' [7], where we must analyse the effect in order to find the cause. In this case, an audio sample would be analysed to find the synthesiser parameter positions which replicate that sample as closely as possible. This relies on the fact that the audio would even be replicable using the synthesiser, so to ensure this, the synthesiser will only be tested on its own sounds. This method could be extrapolated to acoustic instrument samples, however for the scope of this project, the audio data was limited only to these electronic samples.

The powerful tool of neural network models have served many complex tasks in audio analysis. Most commonly, audio categorisation, such as identifying the instrument of an input audio sample using a CNN (convolutional neural network) [5]. This problem is solved using supervised learning, where each sample of audio is accompanied by a label (the type of instrument in this case) [20]. For this project, an audio sample will be labelled by a vector of the synthesiser's parameter values which characterise how the sound was created. For this reason, this project is comparable to a classification problem and can possibly use similar methodology.

This dataset was created using an existing software synthesiser (soft-synth) called 'Monark' [14]. This is a powerful subtractive synthesiser which bases its synthesiser engine on one of the earliest analog synthesisers - the Minimoog Model D, pictured in Figure 1. Although generating a large dataset can take some time, it is incredibly useful to be able to tailor the dataset to the needs of the experiment. A synthesiser built into the coding environment would have perhaps been a more fluid process, however creating a functional synthesiser in Python proved to be a poor use of time, so this industry-standard piece of software was instead used as the instrument for the duration of this project.

In this paper, I will discuss the signal processing used in subtractive synthesis and explain the method for creating a custom dataset. Following this is an explanation of the theory for each of the components which make up the neural network, along with the results of systematic hyperparameter tuning. With a functional neural network and dataset in place, the performance of the network will be evaluated and analysed using the training and cross-validation losses. Then for a full understanding of its performance, the model's output will be fed back into the synthesiser, allowing for an analysis of the audio spectra both quantitatively and qualitatively.

Figure 1: Image of the Minimoog Model D analog synthesiser - the first portable synthesiser to be created.

1.1 Related Work

The 'InverSynth' [16] project has created a very promising model for identifying the synthesiser parameters using a multitude of different neural architectures, including both CNNs (convolutional neural networks) and FCNNs (fully connected neural networks). The results of this work showed great functionality both when analysed numerically and auditorily. The InverSynth investigated either deep CNNs or deep FCNNs, however my project has utilised a combination of a few convolutional layers, followed by a deep FCNN.

Another paper investigated the use of FM (frequency modulation) synthesis [13], which is arguably a much more complicated form, utilising up to 132 parameters in this case. Many machine learning algorithms were implemented, including a hill climber algorithm, genetic algorithm, and various deep networks. My investigation is purely based on subtractive synthesis, but it is still significant due to the similar concepts explored in the field of audio analysis.

2 Background

2.1 Subtractive Synthesis

Subtractive synthesisers work fundamentally by taking a harmonically-rich function, such as a square wave, and altering its frequency spectrum using various operations. These operations include a low-pass filter (LPF) and an amplifier [12]. The parameters within the synthesiser contain both static and dynamic parameters. Static ones, such as the oscillator type, are defined once, whereas dynamic ones, such as the amplitude, are modulated over time by envelope functions [12]. The details of the envelope functions, along with all information related to the parameters are discussed in Section 2.1.1.

Figure 2 depicts the signal chain of the synthesiser with the corresponding parameters for each section. The signal chain begins with an input pitch from a keyboard which defines the original pitch of each oscillator, however following this, each range and tuning parameter for the oscillators ultimately define their pitches. Each oscillator is then combined into a single output via the mixer, which alters the volume of each oscillator. This is then fed into a LPF which filters out all the higher frequencies above a cutoff value [12]. Finally, an amplifier alters the volume of the synthesiser's output audio. Both the cutoff frequency and the amplifier volume are each modulated by an envelope function.

The soft-synth used for this project has many other features, such as a noise generator and frequency modulation tools [14], however these were switched off for this experiment in order to simplify the system.

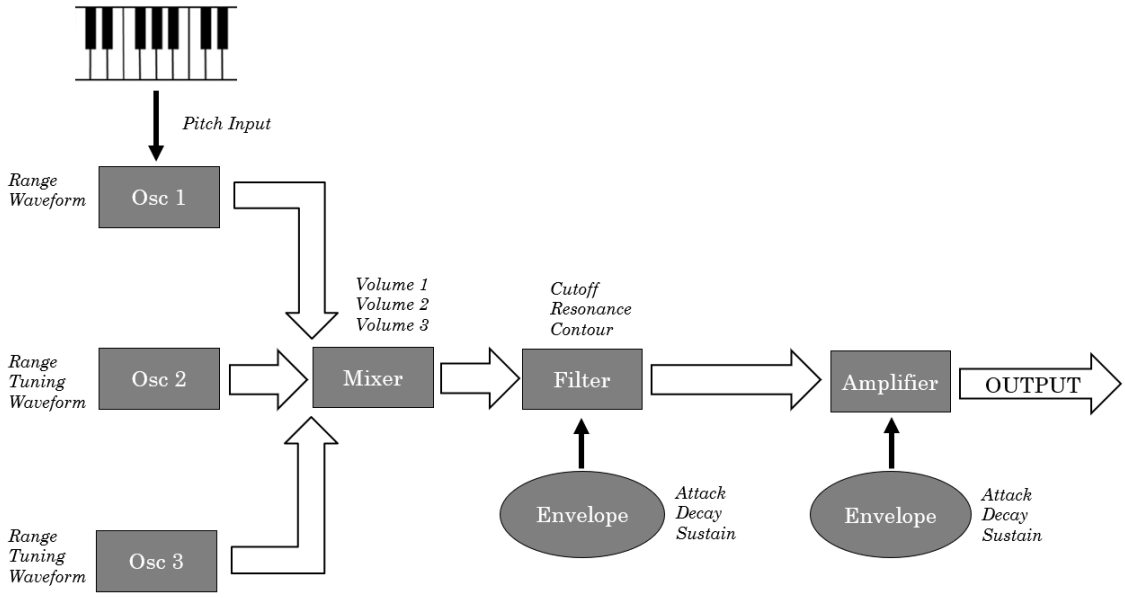


Figure 2: Signal chain of the subtractive synthesiser, depicting the parameters for each section of the signal chain.

2.1.1 Parameter Definitions

Each segment of the signal chain in Figure 2 is defined below, along with their respective parameters [14]:

- **Oscillators** - The sound sources which the synthesiser produces at the beginning of the signal chain. Each oscillator can be tuned to any pitch using the 'range' and 'tuning' parameters.

Osc 1 Parameters:

- **Range** - The octave (register in which a note is played) of the oscillator's pitch. Each octave is 12 st (semitones) apart, where 1 st is simply a measure of pitch (see Appendix A for a detailed description of an octave and semitone).
- **Waveform** - Type of waveform, selecting from a Triangle, Reverse Sawtooth, Sawtooth, Square, Narrow Pulse, and Very Narrow Pulse wave.
- **Tuning** - Fine tuning of the oscillator's pitch, where 0 would output the same pitch as the input. Ranges between -7 st and +7 st.

Osc 2/3 Parameters:

- **Range** - The octave of the oscillator's pitch.
- **Waveform** - Type of waveform.
- **Tuning** - Fine tuning of the oscillator's pitch, relative to the pitch of Osc 1, where 0 would correspond to the same pitch as Osc 1. Ranges between -7 st and 7 st.

- **Mixer** - Alters the maximum amplitude (volume) of each oscillator in the synthesiser.

  Parameters:

  - **Volume 1** - Maximum amplitude of Osc 1.
  - **Volume 2** - Maximum amplitude of Osc 2.
  - **Volume 3** - Maximum amplitude of Osc 3.

- **Filter** - This applies a LPF to the oscillator, whereby the frequency spectrum is modified according to its parameters. See Figure 3, which depicts the frequency response of a LPF.

  Parameters:

  - **Cutoff Frequency** - The boundary value at which all frequencies above it are removed.
  - **Resonance** - The amount of amplitude boost applied only to the cutoff frequency.
  - **Contour** - Determines how significantly the filter envelope will alter the cutoff frequency.

- **Envelope** - A function which modulates another part of the synthesiser when a note is played, which is defined by an Attack, Decay, and Sustain. Figure 4 shows a plot of the envelope function with relation to these three parameters.

  Parameters:

  - **Attack** - Time for the envelope to rise to its peak value.
  - **Decay** - Time for the envelope to lower to its sustain value.
  - **Sustain** - Value which the envelope decays to.

- **Amplitude Envelope** - This is an envelope which modulates the amplitude of the synthesiser's output.

- **Filter Envelope** - This is an envelope which modulates the cutoff frequency of the filter.

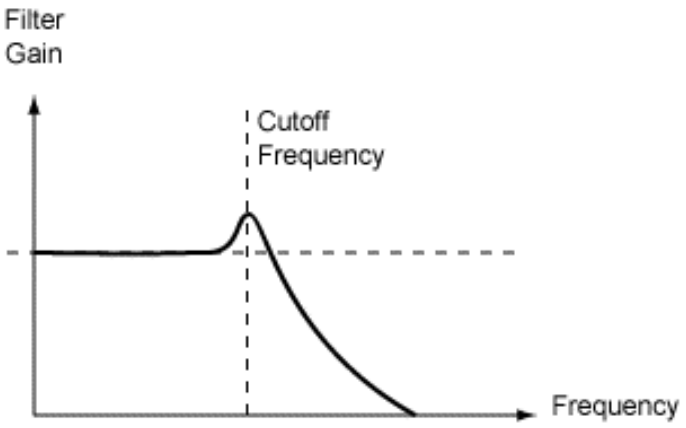


Figure 3: Plot of a low-pass filter frequency response. This modifies an input signal, whereby all frequencies above the cutoff frequency will be reduced by 24 dB in amplitude [14]. The resonance parameter boosts the amplitude of frequencies at the cutoff value.

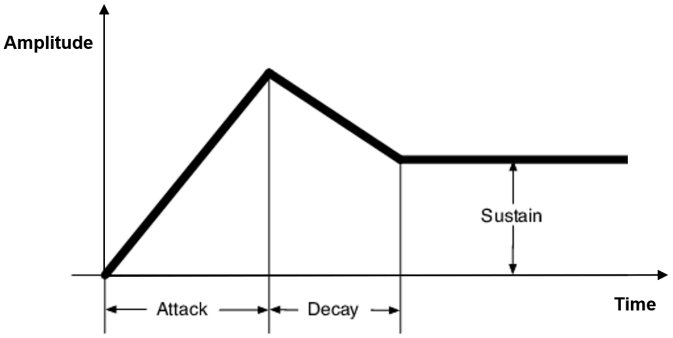


Figure 4: Envelope function plot which modulates the cutoff value and output amplitude of the signal. It is defined by the Attack, Decay and Sustain.

2.2 Deep Learning

Deep learning is a powerful tool for unravelling the incredibly complex patterns and features within a set of data. During this project, the deep learning method utilised was an Artificial Neural Network (ANN) made up of a combination of convolutional and fully connected layers. These layers compute various transformations on an input image to produce an output column vector containing the synthesiser parameters. By tuning these transformations suitably, the model can produce a desired output for the particular application. To achieve these correctly tuned transformations, supervised learning was used, whereby any output from the model could be compared with a known label. For each sample, the network's performance was evaluated by measuring the error between the output of the network and the label, which is known as the loss [6]. Deep learning fundamentally relies on attempting to minimise this loss by iteratively tuning the transformations made to the input data. These transformations are defined by their weight values, of which there can be thousands to tune throughout the whole network. With an appropriate architecture and dataset, the ANN should in theory be able to generalise to data outside of the training set. The applications of an ANN include object classification [23], image reconstruction [17], and time-series prediction [3].

2.2.1 Neuron

An ANN is comprised of a network of neurons, where each neuron receives an input either from a piece of data, or from another neuron [6]. A neuron takes a set of inputs and calculates a single output, as shown in Figure 5. Each input value a_i into a neuron is assigned a corresponding weight

value w_i. All the inputs and weights are compiled into column vectors, $\mathbf{a} = [a_1, a_2, a_3, ...]^T$ and $\mathbf{w} = [w_1, w_2, w_3, ...]^T$. Therefore, the biased weighted sum is written as

$$z = \mathbf{w}^T \cdot \mathbf{a} + b, \tag{1}$$

where b is a bias value [6].

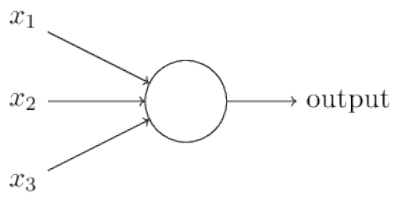


Figure 5: Image of a single neuron, where multiple input values are computed into a single output via an activation function.

The final output of a neuron is determined by an activation function $g(z)$, which is defined by the user. Two activation units were utilised during this project:

Rectified Linear Unit

The Rectified Linear Unit (ReLU), usually used on every layer, sets any negative value from the neuron to zero and maintains all positive values. This is depicted in Figure 6 and is defined as

$$g(z) = max(0, z) \quad [6] \tag{2}$$

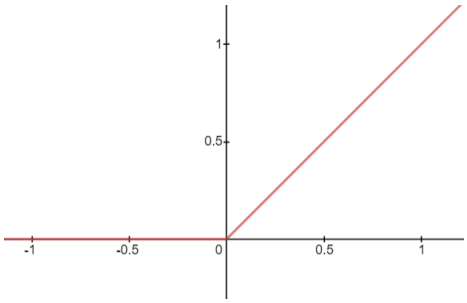


Figure 6: Plot of the ReLU activation function.

Sigmoid

The sigmoid function, usually used on the output layer, sets the output range of a neuron to between 0 and 1. These extrema are reached at an input of $-\infty$ and $+\infty$ respectively, however approximating to 4 decimal places, this range is only about $[-10, +10]$. This is depicted in Figure 7 and is defined as

$$g(z) = \frac{1}{1 + e^{-z}} \quad [10] \tag{3}$$

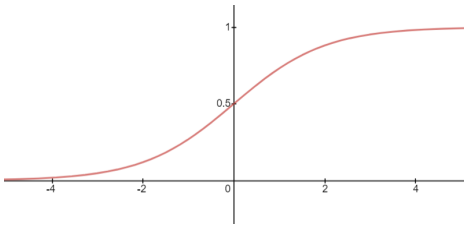


Figure 7: Plot of the sigmoid activation function.

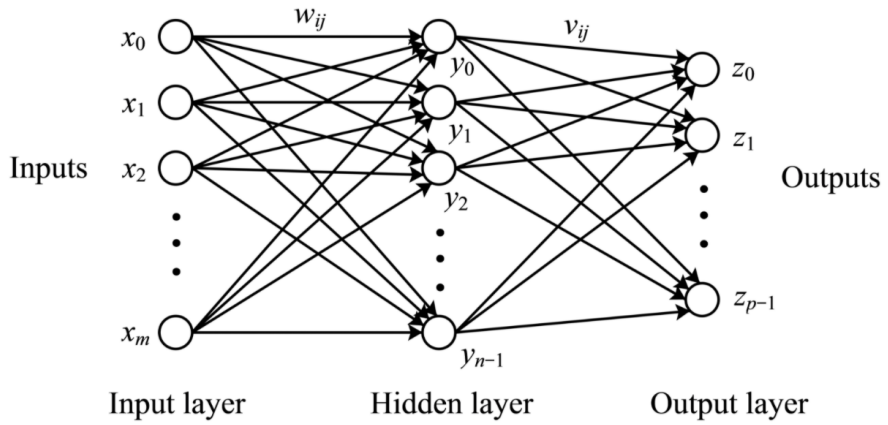


Figure 8: A generalised FCNN, containing an input, hidden, and output layer.

2.2.2 Fully Connected Neural Network

To form a FCNN, a set of neurons are arranged and connected in the form of layers [6]. The neurons in each layer only interact with the layer before and after them, creating a linear flow of information between the input and output. An ANN with this kind of data propagation is known as a Feedforward (FF) network [6]. See Figure 8 which depicts a generalised FCNN, comprising of an input layer, a 'hidden' layer, and an output layer. It is referred to as a hidden layer simply because it is between the input and output layers, and their neurons would not usually be observed. In this particular network, a column vector of size m is fed into the input layer, processed by each neuron based on their weights, and output as a column vector of size p.

2.2.3 Convolutional Neural Network

A CNN consists of convolutional layers which process data by convolving the input with any number of square kernels [6]. To compute a convolution, a kernel h is swept across an input image x, evaluating an output pixel at each point based on the cells currently overlapping with the kernel (see Figure 9). The convolution of a $k \times k$ kernel with an $N \times M$ image is computed for each pixel (i, j) as

$$y(i,j) = \sum_{i=0}^{N} \sum_{j=0}^{M} h(k,k) \cdot x(i-k, j-k). \quad [4] \tag{4}$$

Similar to the FCNN, a CNN uses weights which take the form of the values inside each kernel. It is possible that each different kernel in a single layer is detecting different features within the input image and combining their detected features in order to understand the complex patterns within the data.

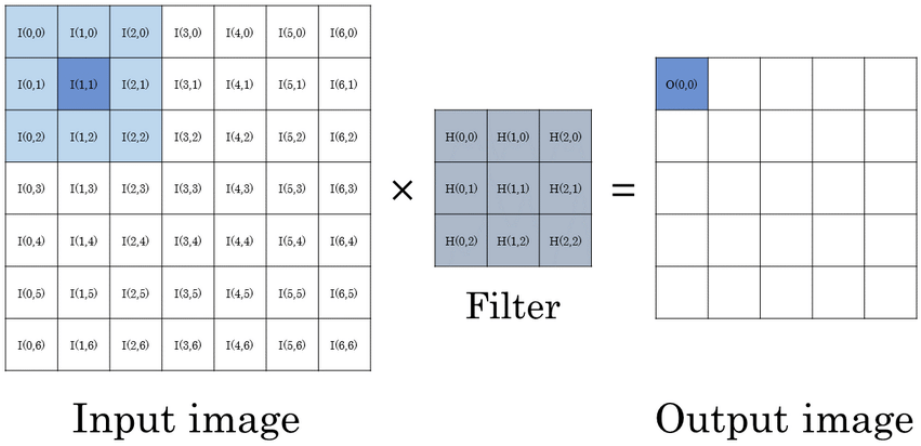


Figure 9: A visualisation of an input image being convolved with a square kernel of size 3. The dot product between the kernel and the image it is covering produces a single output pixel.

A convolutional layer also takes four parameters:

- Kernel Size - The size k of a square kernel.
- Depth - Total number of kernels applied to the input.
- Stride - The distance which the kernel moves after each output pixel calculation. This is usually set to 1, thus iterating over every pixel.
- Padding - The thickness of a border of zeroes applied to the input image before calculating the output. This is used to preserve the size of the image after the convolution. Figure 9 is an example of an input without padding, causing the output size to reduce by 2 in each dimension

Between convolutional layers in a network, max pooling layers were used to reduce the size of the image and thus the required computation. This layer applies a filter across the image, most commonly a 2×2 filter with a stride of 2. At each evaluated pixel, the maximum value inside the filter is taken as the output.

Convolutional layers are a much more effective method for image processing due to their efficient use of a single weight parameter. By sliding the same kernel over a whole image, there are fewer parameters to be adjusted by the network. If a FCNN were to be used on the same image, every pixel would require a weight parameter for each connection made to the next layer.

2.2.4 Network Training

ANNs such as these require that the weights are trained for their particular application. To train a network, a labelled dataset must be used. A dataset of size N is defined as

$$\mathbf{X} = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_N \end{bmatrix} = \begin{bmatrix} (x_1, y_1) \\ (x_2, y_2) \\ \vdots \\ (x_N, y_N) \end{bmatrix},$$

where a datapoint X_i is comprised of the data x_i and a label y_i. For example, this could be a dataset of dates and valuations for a stock market price, or a pixel array of a blurred image with the corresponding non-blurred image.

This comparison is defined by the loss function $J(y_i, z_i)$, which is a distance measure between z_i and y_i. Once this loss is evaluated, the weights and biases are updated in the direction which most efficiently reduces the loss. Using the SGD method (Stochastic Gradient Descent), the updated weight or bias is defined as

$$\theta_{t+1} = \theta_t - \eta \frac{\delta J}{\delta \theta_t}, \tag{5}$$

where the variable η is the learning rate which defines how significantly the weights adjust [6].

The purpose of this negative partial derivative is to move the loss towards a local minimum. See Figure 10, which depicts a loss function with a local and global minimum value. Take for example that the weight is currently to the right of the local minimum. The gradient is positive at this point, implying the weight must be moved in the opposite direction to the gradient in order to reduce the loss. The challenge is overcoming this local minimum in order to find a better solution.

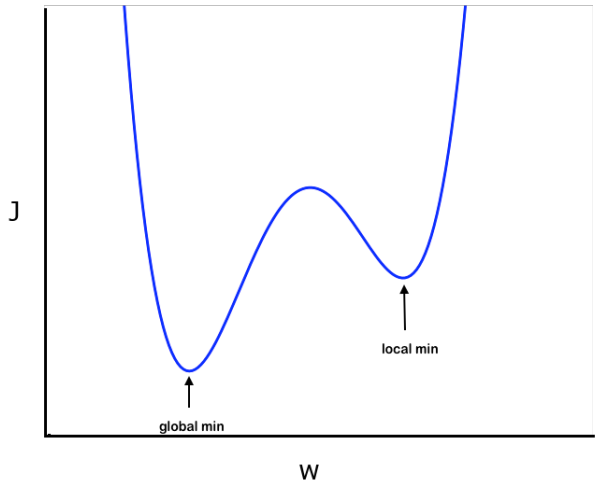


Figure 10: Example of local minimum for a single weight value. For an ANN, this would have more dimensions depending on the total number of weights in the network.

Overfitting is a common occurence with ANNs, and occurs when the network is unable to perform well when using data besides the training dataset. This lack of generalisation must be remedied, and by using L2 regularisation, that was achieved during this project. L2 regularisation modifies the weight update, whereby the new loss function is modified as

$$J(y_i, z_i)' = J(y_i, z_i) + \lambda \sum_{i=1}^{N} \theta_i^2, \tag{6}$$

where N is the total number of weights and λ is a constant known as the weight decay [6].

This essentially penalises the model for having very large weights, which in turn can cause some of the weights to approach zero. By making certain neurons redundant, the complexity of the model is gradually reduced until an optimal network is found. Both too complex a model and too simple a model can overfit the data, so this middle ground can be quickly found using this method.

Evaluating these gradients for each weight and bias is accomplished using a built in 'backpropagation' algorithm in PyTorch [19]. This algorithm evaluates the gradient for each weight, beginning at the final layer and moving backwards to finish at the first layer.

The training algorithm used in this project utilised mini-batch learning, whereby the weights were only updated after processing a batch of b datapoints. During each batch, the losses were accumulated and averaged. The network then updated the weights and repeated for all batches,

accumulating the average loss for analysis purposes. Utilising mini-batch learning prevented any noise in the data from significantly affecting the weight updates, allowing for an increased learning potential. The pseudocode for this algorithm is written as

Data: $\mathbf{X}$, b
Result: Losses
initialise weights
Losses = [ ]
for *all epochs* **do**
  loss_total = 0
  for *all batches* **do**
    loss_batch = 0
    for X *in batch* **do**
      $\mathrm{x}_i, y_i = X_i$
      $\mathrm{z}_i = model(x_i)$
      loss_total += $\mathrm{J}(\mathrm{y}_i, z_i)/b$
      compute gradients
      update weights
    end
    loss_total += loss_batch
  end
  Losses.append(loss_total)
end

Algorithm 1: Algorithm for training an ANN using mini-batch leaning. The mean loss is accumulated over each batch before updating the weights, which reduces the noise of the data.

2.2.5 Optimiser

The optimiser of a neural network is the algorithm which determines how the weights update. During this project, three optimiser algorithms were compared: The SGD (stochastic gradient descent), AdaGrad (adaptive gradient), and AdaDelta. The SGD optimiser maintains a constant learning rate at each epoch, whereas the other methods modify the learning rate with each update.

SGD
This is the standard update algorithm and maintains a constant learning rate throughout the model's learning process.

For an initial weight value θ_t, the updated value is defined as

$$\theta_{t+1} = \theta_t - \eta g_t, \tag{7}$$

where η is a constant (learning rate) and g_t is the partial derivative of the loss, $\frac{\delta J}{\delta \theta_t}$ [6].

AdaGrad
This algorithm updates the learning rate with respect to the sum of the squares of all previous gradients, G_t. As this term increases with each update, the learning rate decreases, approaching a very low learning rate in order to facilitate a finer tuning of the weights.

For an initial weight value θ_t, the updated value is defined as

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} g_t, \tag{8}$$

where ϵ is a 'smoothing term' which prevents a division by zero in the case of $G_t = 0$ [9].

AdaDelta
This is an adjustment of AdaGrad which only accumulates all gradients within a window instead of over the whole training process. Storing all of the gradients over a time window would be inefficient,

so instead, all previous gradients are decayed with proportion to a momentum factor ρ. This is defined as an expectation for the previous gradients

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho)g_t^2. \tag{9}$$

Then, similar to Equation 8, the updated weight is defined as

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t. \quad [15] \tag{10}$$

3 Dataset

No dataset exists for this kind of application, so a custom dataset was created using the soft-synth described in Section 2.1. By using a digital instrument, it is possible to precisely interact with the parameters of the soft-synth using MIDI (Musical Instrument Digital Interface) data.

3.1 MIDI Communication

Software such as the soft-synth used during this project communicates with hardware using MIDI data. After creating a virtual MIDI port using loopMIDI [22], Python was used to communicate with the soft-synth. The main type of MIDI message used in this project is a Control Change (CC) message. This sends a value from 0-127 to any parameter index 0-127. By firstly assigning each parameter a unique parameter index, all of the parameters were easily tuned by iterating over each parameter index and sending a MIDI CC message. The other type of MIDI message switches a musical key either on or off, and takes a string input denoting the note to be played on the keyboard input.

3.2 Sample Recording

To create a datapoint, a set of parameters were selected and each of their values were randomly sampled and assigned using MIDI CC messages. Then a middle C (C4) is played by switching on the MIDI note. The audio output is then recorded as a WAV (Waveform Audio File) whilst the note is on. Although only one pitch is ever played on the keyboard input, the tuning parameters on the synthesiser allow for a full range of pitches to be played, making the choice of note irrelevant. The pseudocode algorithm is written as

Data: N, *params_list*
Result: Create Dataset.
param_values = [ ]
for *param in param_list* **do**
    value = randomSample()
    midiCC(param, value)
    param_values.append(value)
end
save param_values
midiNote(C4, 'on')
record audio
midiNote(C4, 'off')
save audio

Algorithm 2: Algorithm for creating a dataset of size N.

3.3 Parameter Randomisation

Each parameter takes a value between 0 and 127, but as described in Table 1, some parameters have fewer than 128 positions, where each position would correspond to a range of values. In order to minimise the total number of combinations of parameter values, the value choices were reduced down to the number of different positions for each parameter. For example, the 'Type' parameter for an oscillator has 6 positions and would have a value between 0 and 5. The neural network model aims to understand patterns within the data, so to facilitate efficient learning, only one value was assigned for these discrete parameters.

Section	**Oscillators**				**Filter**			**Envelopes**		
Parameter	*Range*	*Tuning*	*Waveform*	*Volume*	*Cutoff*	*Res*	*Contour*	*A*	*D*	*S*
Positions	0-5	0-127	0-5	0-127	0-127	0-127	0-127	0-127	0-127	0-127

Table 1: Table of all the possible parameter positions, sorted by their section on the synthesiser

Position	0	1	2	3	4	5
Values	0-25	26-50	51-76	77-101	102-126	127

Table 2: Table of the ranges of values which correspond to the discrete parameter positions

3.3.1 Value Sampling

When randomising the majority of the parameter values, a value r in the range [0,1] was uniformly sampled. Since the synth receives an input in the range [0,127], $127r$ was evaluated and rounded to give the final parameter value. For a discrete parameter with 6 unique positions, $6r$ was evaluated and rounded, and was then used as an index to select a single parameter value from an array of length 6. This array contains the lower bound value for each parameter range, which is shown in Table 2.

Sampling from a uniform distribution is useful for creating a variety of data samples, however certain parameters would benefit from being within a certain range, such as the oscillator tuning parameters. This is because the relative tuning between the three oscillators should be close enough to produce a single distinguishable pitch. Therefore, a Gaussian distribution was used to sample r instead, vastly reducing the number of datapoints with an undesirable audio output. This Gaussian has a mean of 0.5 and a standard deviation of 0.03. Similarly to the first sampling method, this value is multiplied by 127 and rounded to attain the correct MIDI value range. A value of 0.03 was chosen so that for two points sampled within one standard deviation from the mean, the maximum distance in pitch would be less than 1 st. This method is important when combining two or more oscillators together, as a slight detuning between oscillators creates a desirable 'phasing' effect [12], whereas a large detuning would effectively be two separate notes sounding simultaneously. Since one standard deviation about the mean, on average, consists of about 68% of the total distribution (known as the 'empirical rule' [1]), the majority of the dataset can be considered as single-pitched sounds.

4 Experimentation

4.1 Pre-processing Data for the Network

4.1.1 Audio Processing

The characteristic information of any audio can be found by analysing the frequency spectrum. A common method of finding this spectrum is by calculating the fourier transform. A fourier transform converts any data from the time domain t to the frequency domain f. Mathematically, this is defined as

$$X(f) = \int_{-\infty}^{\infty} x(t)exp(-2\pi ift)dt. \tag{11}$$

In the realm of the discrete datapoints which define an audio sample, computing such an integral is impossible. This is where the Discrete Fourier Transform (DFT) is utilised. Instead of iterating over an infinite number of points, the DFT iterates over N datapoints. Similarly, instead of accessing any frequency f, the overall spectrum is divided into a number of frequency bins. For the kth bin, the DFT is defined as

$$X(k) = \sum_{n=0}^{N-1} x(n)exp(-\frac{2\pi ikn}{N}). \quad [11] \tag{12}$$

For this proejct, a time-dependent frequency plot was required to analyse certain features, such as filter sweeps and beating effects. This kind of plot is known as a spectrogram, and was calculated using the SciPy spectrogram function. This function divides the audio sample into many short overlapping time windows (the overlap maintains continuity between each window), calculates the DFT for each window, and then combines them all to form a complete spectrogram of the sample.

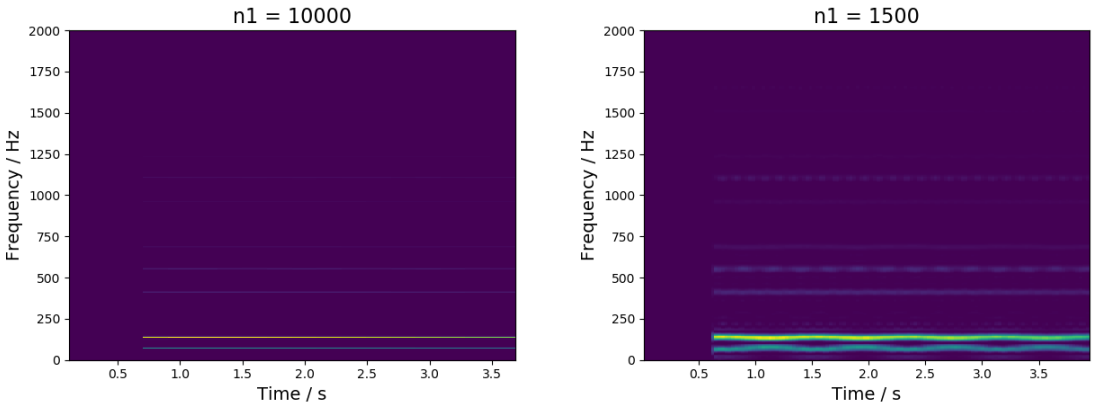
One problem encountered with this technique was choosing the length of each time window $n1$ and the number of frequency bands $n2$. After some experimentation, it appeared that there was a trade off between the time precision and accuracy of the image when selecting $n1$. See Figure 11 for a comparison of different values of $n1$ when creating a spectrogram of an audio sample with three oscillators. For $n1 = 500$ and 100, there appear to be discontinuities in the audio, when in reality the sound is a continuous note. These time windows are far too short to collect a sufficient number of datapoints. Where there should be distinct harmonics (horizontal lines), there are wide bands of frequencies which blend into eachother. The subtractive synthesiser used only produces clean tones with distinguishable harmonics, so this is simply not an accurate image.

Using a very long time window of 10000 datapoints (spanning roughly 0.2s at a sample rate of 44.1kHz), the precision in time, and thus any sublte changes in frequency during the sample are lost in this example. This is clear when looking at a value of $n1 = 1500$, where all the the harmonics are clearly defined, with the finer detail of the lower frequencies beating together and creating an oscillating amplitude. Therefore, a value of $n1 = 1500$ was settled on for the duration of the experiment, since it provided an accurate depiction of the wave without causing any discontinuities between windows. $n2 = 10000$ was chosen for the purpose of using a sufficiently high frequency precision. This corresponds to 5000 frequency bands, each of which are 4Hz in length, making it capable of distinguishing the vast majority of pitches.

The spectrogram pixel values were normalised so that only the relative distribution of the spectra would be measured. Differences in volume between two spectra should not be considered into calculating the error between them.

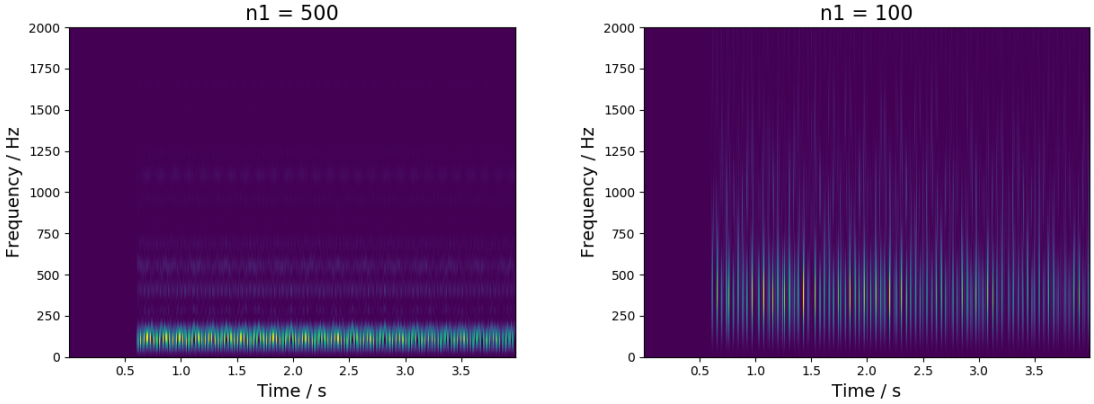
4.1.2 Parameter Processing

The initial parameter values are in the discrete range [0,127], containing only integer values. In order to utilise the sigmoid activation function, all parameter labels were divided by 127 in order to modify the range suitably to [0,1].



(a) Very long time windows

(b) Optimal time windows



(c) Short time windows

(d) Very short time windows

Figure 11: Spectrogram images of an audio sample, each using a different size for the time windows, $n1$. The images have been cropped as most of the spectrum is within the 0-2000Hz range for this sample.

4.2 Performance Measures

4.2.1 Training and Cross-Validation Loss

The loss function used during this project was the MSE (mean squared error), and is defined as

$$J(y_i, z_i) = \sum_{i=1}^{p} (y_i - z_i)^2, \tag{13}$$

where p is the length of the parameter vector [6].

In order to evaluate the quality of the fit of a model, a cross-validation resampling method was implemented. For each training regime, 20% of the training data was randomly selected and moved into a validation set. The validation set is essentially treated as 'unseen' data to the model, as the weights are not updated when evaluating the losses. The mean training and validation loss were recorded after each epoch. This portrays how the model's performance changes with each iteration, but more importantly, allows for analysis of how the model responds to the unseen data in the validation set. The goal for this measure is to see a constant decrease in both losses, with a validation loss which is as close as possible to the training loss.

4.2.2 Power Spectral Density

By feeding the model's output parameters back into the synthesiser, an audio sample can be compared to its resulting audio output. Comparing the two spectra was accomplished by calculating the PSD (power spectral density) of each. The PSD is calculated by firstly calculating the DFT, as described in Section 4.1. Then for each of the frequency bands ω, the PSD can be calculated as

$$PSD(\omega) = \lim_{T \to \infty} \mathbf{E}\left[|X(\omega)|^2\right], \tag{14}$$

where $X(\omega)$ is the DFT of the input audio signal $x(t)$, and T is the time window (number of points) which the PSD averaged over [18].

Mathematically, this is for a static signal, so the average is taken up to an infinite limit, however in this situation it is simply the length of the audio sample. The final metric will be the sum of the MSEs for all frequency bands, which after subbing in $\omega = \frac{2\pi k}{N}$, is defined as

$$\sum_{k=0}^{N} \text{MSE}(\text{PSD}_1(k_i), \text{PSD}_2(k_i)), \tag{15}$$

where there are N frequency bands.

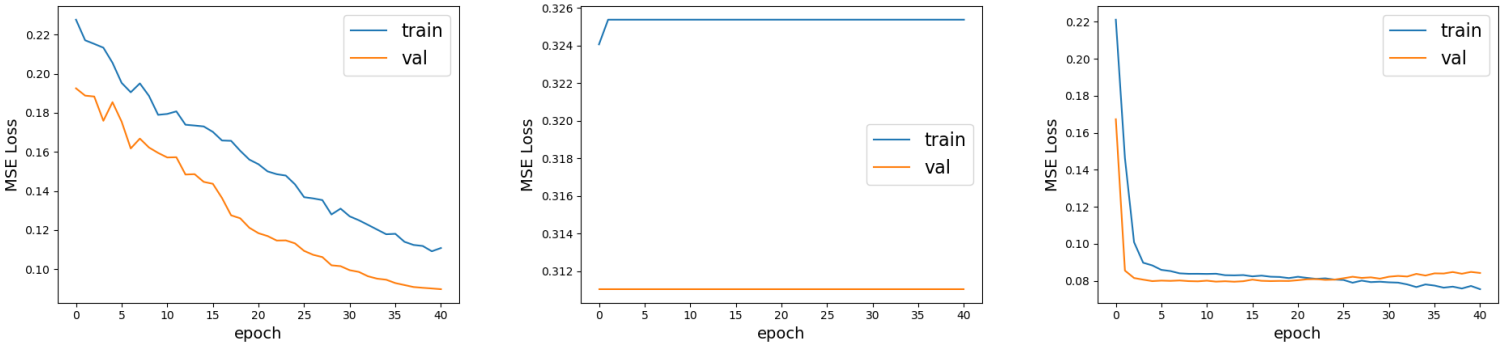
Since ultimately the model is aiming to imitate the harmonic content of the input audio, the performance of the final network will be assessed using this tool.

4.3 Neural Architecture

The image classification tasks processed by the AlexNet are quite successful [2], so this was used as a starting point for the architecture. This led to an architecture including three convolutional layers with max-pooling between each, and a FCNN leading up to the output. Only three convolutional layers were used, and although the InverSynth [16] successfully utilised a network with 6 convolutional layers, increasing the number for this network showed a reduction in performance. As to reduce the nosie from the data,a batch size of 5 was chosen for the duration of this project.

4.3.1 Optimiser Choice

The three optimisers discussed in Section 2.2.5 were tested using a sample size of 500. Figure 12 depicts the training and validation loss plots for each of the optimisers. The AdaDelta performed best in this case with the lowest final loss of 0.0842. Although the SGD optimiser had a similarly low loss value, the training and validation loss for the AdaDelta optimiser are much closer, indicating a better fit. Therefore, AdaDelta was selected as the algorithm for the weight updates during all further experimentation.



(a) SGD Optimiser
Final validation loss of 0.0897.

(b) AdaGrad Optimiser
Final validation loss of 0.3110.

(c) AdaDelta Optimiser
Final validation loss of 0.0842.

Figure 12: Training and validation loss plots to determine the best of three optimiser algorithms. The AdaDelta optimiser performed the best with the lowest final loss and best fit.

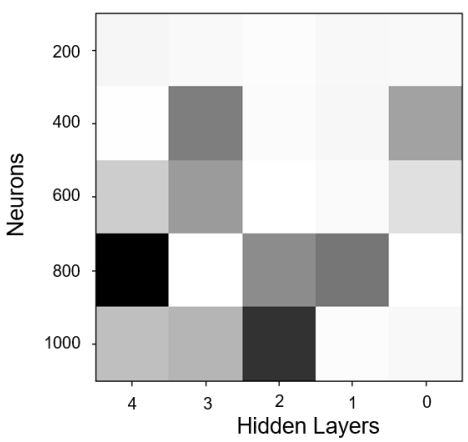
4.3.2 Hyperparameter Tuning

For simplicity, only one oscillator on the synthesiser was utilised, resulting in a total of 12 parameters. If successful, the model should be capable of expanding to utilise more of the synthesiser's harmonic possibilities.

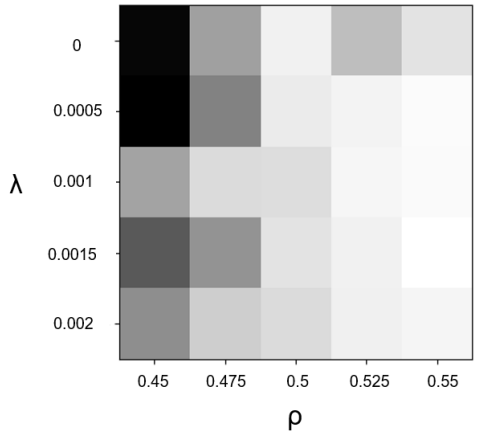
Tuning the initial parameters, such as the kernel sizes and weight initialisation, involved some trial and improvement on some very small datasets. The weight initialisation was done randomly at the beginning of each training regime, using ranges of [-0.5, 0.5] and [-0.1,0.1] for the convolutional and FC layers respectively. The default initialisation in PyTorch was simply too narrow for this application as it required a wider search space for the weights in order to reach a solution.

Once a suitable network was almost fully implemented, a more rigorous hyperparameter tuning method was used. A grid-search algorithm was implemented for two pairs of hyperparameters, whereby all possible combinations of a set of parameters was exhaustively tested until the best combination was found. Firstly, the FC layers were tuned, both in the number of hidden layers and the number of neurons in each. Once the optimal values for these two parameters were selected, another search was carried out for the optimiser algorithm, whereby the momentum factor ρ and the weight decay λ were tuned. For both grid-search tests, 5 values for each parameter were selected to be investigated, and the model was trained on 500 samples using all 25 parameter combinations. To compensate for the inherent randomness of the model, mainly due to random weight initialisation, this training process was carried out three times. To choose the best parameters, the minimum validation loss was evaluated and averaged over the three iterations.

The performance for the layer parameters can be visualised in an image plot, as shown in Figure 13a. The best parameters coincide with the pixel of value 0.1345 (darkest), which corresponds to 4 hidden FC layers, each with 800 neurons. Using these parameters for the optimiser test, the same procedure was repeated, producing the plot shown in Figure 13b. The best parameters here coincide with the pixel of value 0.1219, which corresponds to $\rho = 0.45$ and $\lambda = 0.0005$.



(a) Layer Parameters



(b) AdaDelta Optimiser Parameters

Figure 13: Grid-search algorithm results for choosing the layer and AdaDelta hyperparameters. The darker the cell, the lower the average validation loss.

5 Results

Setting the network's hyperparameters to those found from the grid-search experiments, the final architecture is described in Table 3.

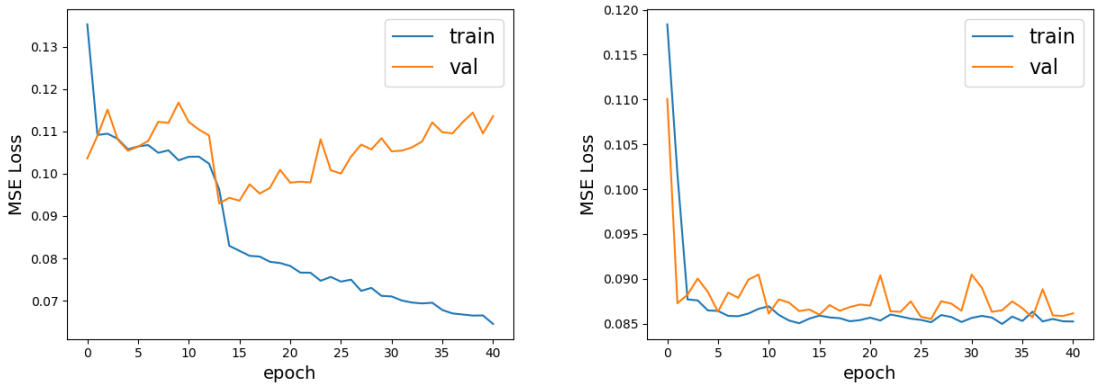
Layer	**Parameters**
Input	Size = (5001,30)
2D Conv	Kernels = 32, Activation = ReLU
MaxPool	Size = (2,2)
2D Conv	Kernels = 32, Activation = ReLU
MaxPool	Size = (2,2)
2D Conv	Kernels = 32, Activation = ReLU
MaxPool	Size = (2,2)
Flatten	Size = 1875
FC	Neurons = 800, Activation = ReLU
FC	Neurons = 800, Activation = ReLU
FC	Neurons = 800, Activation = ReLU
FC	Neurons = 800, Activation = ReLU
Output	Size = 12, Activation = Sigmoid

Table 3: Table depicting each layer of the neural architecture with their respective parameters. All convolutional layers have a kernel size of 5, a stride of 1, and a padding of 1.

This final model was trained using 10000 samples for a total of 40 epochs. See Figure 14, which depicts the training and validation losses for a model trained both with and without L2 regularisation. Without regularisation, the model appears to have overfit very strongly, as shown by the validation loss becoming larger than the training loss. This overfitting increases with each epoch from epoch 13, and is known as overtraining [6]. Applying a weight decay of 0.0005, the model no longer overfits the training data, as shown by a similar validation and training loss. A problem, however, is that this model's performance has saturated at only 1 epoch, implying that the loss is stuck within a local minimum.

Using a test dataset of size 100, their corresponding output parameters were computed using the regularised model. Then these parameters were fed back into the synthesiser and recorded similarly to the dataset. See Figure 15, which depicts the spectrogram and PSD for the original audio sample and the output audio sample. This example is the result which achieved the lowest total MSE for the PSD distributions - 0.0112. This very high accuracy, however, does not recur when observing

the other datapoints, as shown in Figure 16, which depicts a very wide distribution of the total MSE for each test point.



(a) No regularisation - $\lambda = 0$

(b) Regularisation - $\lambda = 0.0005$

Figure 14: A graph of the training and validation losses for each epoch after training the model with and without regularisation. The addition of regularisation improves the fit of the plot, thus improving the performance on new data.

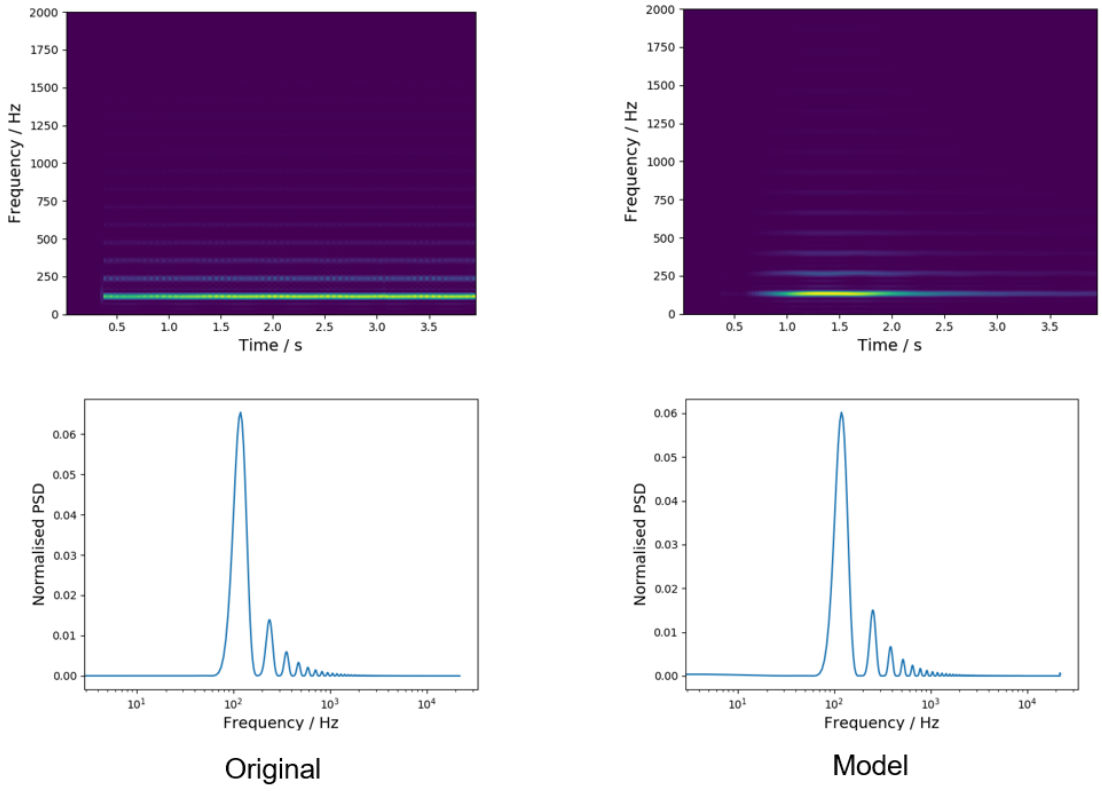


Figure 15: An example of one of the test datapoints, depicting the spectra and PSD for both the original sample and the model's output sample.

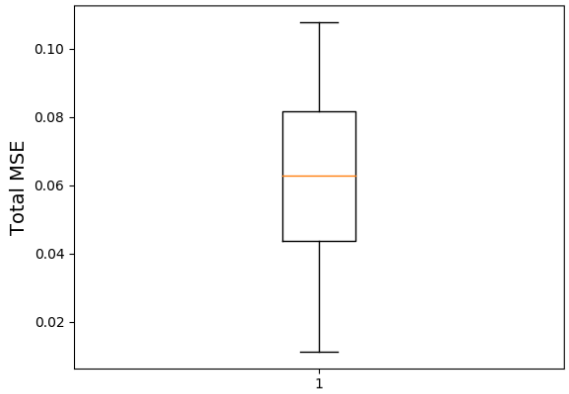
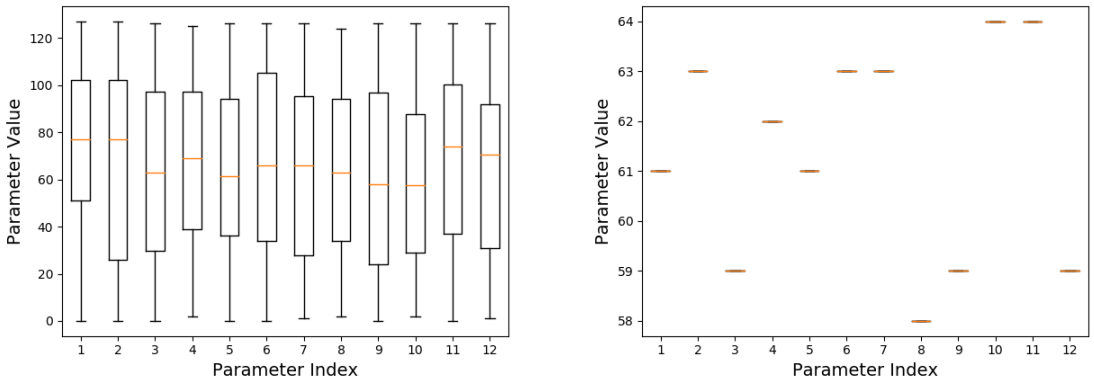


Figure 16: Distribution of the total MSE between the input and output PSD plots for each test datapoint.

In order to explain this result, the output parameter values from the network were inspected more closely. The distribution of the input labels and the model's outputs have been plotted in Figure 17, and it reveals that the network does not seem to adjust for each datapoint. The distributions depict how despite the input parameters covering the full range of values, [0,127], the output maintains constant values for all parameters. The model has most likely found a local minimum, in which the parameter sits in the center of this range to minimise errors due to points at the extrema.

It is quite possible that the network is unable to detect the complex features within the dataset, especially one which contains images with 150,030 pixels. In order to investigate this possibility, the spectrogram was recalculated using only 500 frequency bands, reducing the number of pixels by 10 times. In addition to this reduction, the range parameter was removed from the variables used in the dataset. This is because the range parameter can cause drastic changes in the image from just one change in value. For example, if the parameter value is on the border between positions 2 and 3 (see Table 2), the slightest change in the value would cause the pitch of the oscillator to increase by an octave. Since an octave increase doubles the frequency of the note, this could in theory significantly shift the frequency bands across the image. This kind of movement is absolutely not conducive to the gradual learning process of an ANN. Unfortunately, after repeating the experimentation using this simpler approach, there was no improvement to the results. During training, the model's parameter output approached the center of the range for all parameters, leading to the exact same conclusion as discussed previously.



(a) Input parameter distribution

(b) Output parameter distribution

Figure 17: Boxplots displaying the distributions of the parameter labels and outputs for the test dataset.

Making comparison to the literature (most notably the InverSynth), it is likely that the training dataset is far too small for the complex features to be drawn out from the data. The InverSynth succesfully implemented this kind of model with 200 thousand datapoints [16], and is likely the missing element of this model's training regime. Such a large set would either require much more powerful computing power, or a reduced sample resolution. The InverSynth implemented principal component analysis (PCA) [8] as a method of reducing the dimensionality of the dataset, and thus the required computation. Although information was removed, about 99% of the variance [7] was maintained in the dataset, allowing for effective feature detection by the CNN.

In future development of this project, the previously-discussed local minima issue will be investigated much further. Firstly the model will be stripped back in order to identify at which point it breaks down. This could begin with a single oscillator and filter cutoff value, removing all temporal elements. Massively reducing the number of variables and features would allow for a significantly larger dataset, and could produce a more generalised model.

6 Conclusion

This project has successfully created a functional subtractive synthesiser dataset with full customisation of the parameters used. The final neural network model was also functional, showing a well-fitted loss plot when L2 regularisation was applied. This model, however, did not perform as well when the output audio samples were analysed. If the loss function was capable of escaping the recurring local minimum, then it would likely find a better solution. Unfortunately, escaping a local minimum is no simple task and may require a combination of various methods. These include the use of a different optimiser algorithm, more datapoints, or dimensionality reduction. Dimensionality reduction techniques, such as PCA [8], can help draw out important features within the data whilst maintaining the majority of the variance. With my limited computational power, PCA would allow for more datapoints being used for the same amount of computation time.

Despite the shortcomings of these results, there is a good foundation set in place for future testing. I have found that in neural computation, the bug fixing and optimisation can become tedious and complex. Referring back to a proven methodology would hopefully teach a more robust approach to solving the challenges faced in this investigation.

Using deep nets to find the synthesiser parameters has been found to work in the literature ([16], [13]). This kind of program is a fantastic step into the use of artificial intelligence in music, as it could open up a lot of possibilities when it comes to synthesis techniques too complex for a human to control (such as using physical modelling synthesis [21]). The hopes of this project was to achieve a proof-of-concept for such a tool using a less complex form of synthesis. However, despite the inadequate results, this project has succeeded in discussing the potential improvements which could be implemented to take one step closer to that goal.

References

[1] Hayes A. *Empirical Rule*. 2020.

[2] Krizhevsky A et al. *ImageNet Classification with Deep Convolutional Neural Networks*. 2012.

[3] Oancea B. *Time Series Forecasting Using Neural Networks*. 2014.

[4] Duudgeon D and Mersereau R. *Multidimensional Digital Signal Processing*. 1983.

[5] Seipel F. *Music Instrument Identification using Convolutional Neural Networks*. 2018.

[6] A. Goodfellow et al. *Deep Learning*. 2016.

[7] Li H et al. *Inverse Problems*. 2020.

[8] Jolliffe I. *Principal Component Analysis*. 2002.

[9] Duchi J et al. *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*. 2011.

[10] Han J. *From Natural to Artificial Neural Computation*. 1995.

[11] Frigo M et al. *A Modified Split-Radix FFT With Fewer Arithmetic Operations*. 2007.

[12] Russ M. *Sound Synthesis and Sampling*. 2009.

[13] Yee-King M et al. *Automatic Programming of VST Sound Synthesizers Using Deep Networks and Other Techniques*. 2018.

[14] Zagler M, Gover D, and Daliot M. "Native Instruments Monark Manual". In: *https://www.manualslib.com/manual/*
Instruments-Monark.html (2017).

[15] Zeiler M. *ADADELTA: An Adaptive Learning Rate Method*. 2012.

[16] Barkan O et al. *InverSynth: Deep Estimation of Synthesizer Parameter Configurations from Audio Signals*. 2018.

[17] Liu P. *Image Reconstruction Using Deep Learning*. 2018.

[18] Stoica P and Moses R. *Spectral Analysis of Signals*. 2005.

[19] PyTorch. "Automatic Differentiation Package - torch.autograd". In: *https://pytorch.org/docs/stable/autograd.html* (2016).

[20] Russel R and Norvig P. *Artificial Intelligence: A Modern Approach*. 2010.

[21] Bilbao S et al. *Large-Scale Real-Time Modular Physical Modelling Sound Synthesis*. 2019.

[22] Erichsen T. "loopMIDI". In: *https://www.tobias-erichsen.de/software/loopmidi.html* (2010).

[23] Zhao Z. *Object Detection with Deep Learning: A Review*. 2017.

A Musical Intervals

An octave is a measurement between two notes with the same pitch label, but different frequencies. See the example in Figure 18, where the note 'C' appears on a piano keyboard multiple times. The distance between each adjacent C would be called an octave. When moving up or down an octave, the frequency of the note will double or halve respectively.

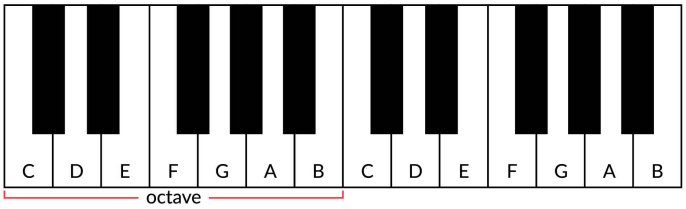


Figure 18: A piano depicting the length an octave, beginning and ending on the note 'C'.

A semitone is the result of the way in which western music has been notated, and is defined as $1/12$ th of an octave interval. This is the smallest possible interval seen in Figure 18. For a starting frequency a, the new frequency b is defined as

$$b = a \times 2^{\frac{n}{12}}, \tag{16}$$

where n is the number of semitones from the starting note.

B GitLab Repository

link - *https://git-teaching.cs.bham.ac.uk/mod-msc-proj-2019/haa640*

- **model.py** - the program used to define and train the neural network.
- **dataset.py** - generates the custom dataset by sending MIDI data to the synthesiser.
- **process.py** - converts audio data into frequency images and compiles each image and parameter vector into a hickle file.
- **loss.py** - loads the training and validation losses stored from **model.py** and plots them together.
- **test.py** - loads the entire model saved from **model.py** and passes the test dataset.
- **dataset/** - includes a sample of the data which would be produced by dataset.py

Each file is run separately depending on what is needed. The dataset creation will require access to some other software, so a sample set has been included. This is processed using process.py, and can then be analysed using either the loss or test programs.