Automatic Programming of VST Sound Synthesizers using Deep Networks and Other Techniques

Matthew John Yee-King, Leon Fedden and Mark d'Inverno

***Abstract*—Programming sound synthesizers is a complex and time-consuming task. Automatic synthesizer programming involves finding parameters for sound synthesizers using algorithmic methods. Sound matching is one application of automatic programming, where the aim is to find the parameters for a synthesizer that cause it to emit as close a sound as possible to a target sound. We describe and compare several sound matching techniques that can be used to automatically program the Dexed synthesizer, which is a virtual model of a Yamaha DX7. The techniques are a hill climber, a genetic algorithm and three deep neural networks that have not been applied to the problem before. We define a sound matching task based on six sets of sounds, which we derived from increasingly complex configurations of the Dexed synthesis algorithm. A bidirectional, long short-term memory network (LSTM) with highway layers performed better than any other technique and was able to match sounds closely in 25% of the test cases. This network was also able to match sounds in near real time, once trained, which provides a significant speed advantage over previously reported techniques that are based on search heuristics. We also describe our open source framework which makes it possible to repeat our study, and to adapt it to different synthesizers and algorithmic programming techniques.**

***Index Terms*—Computer generated music, Signal synthesis, Artificial neural networks, Genetic algorithms, Frequency modulation**

I. Introduction

SOUND synthesizer programming is an important and difficult activity which is carried out regularly by professional musicians, sound designers and composers. It first involves the selection of a synthesizer, and then the adjustment of its settings until the user is satisfied with the sound it produces.

Synthesizers can have hundreds of parameters, which means that the potential set of configurations is enormous. In this paper, for example, we study the Dexed synthesizer which has 155 real-valued parameters. Additionally, these parameters can be non-linear with respect to the timbre of the sound, and they can be interdependent such that one parameter's effect depends upon the value of one or more of the other parameters.

Human synthesizer programmers currently have two options: they can use libraries of preset sounds, possibly adjusting some parameters to fine tune the tone, or they can program the synthesizer from scratch. In this paper, we consider a third option: automatically deriving sound synthesis parameters using machine learning and optimisation techniques.

Specifically, we consider the problem of sound matching, where the aim is to program a synthesizer to generate sound as close as possible to a specific target. We address the sound matching problem at several levels of complexity, using several machine learning and optimisation techniques. We investigate the hypothesis that deep network architectures will outperform the previously reported sound matching techniques, in terms of speed and accuracy.

A. Previous work

There is a considerable body of work in automatic sound synthesizer programming. A key task in this work, as described above, is sound matching. Horner et al. described one of the first sound matching systems, which used a genetic algorithm (GA) to find settings for frequency modulation (FM) synthesis algorithms [1]. Several researchers have re-addressed the FM synthesizer parameter optimisation problem, for example, Mitchell et al. [2] and Roth [3].

Sound matching is one application of automatic sound synthesizer programming. Another application is to provide a more intuitive interface between the user and the parameters, where the user operates in an intuitive timbral or visual space, and the programming system generates the synthesizer parameters required. Arbib et al. used machine learning techniques to map from perceptual space to synthesizer parameter space [4]. Other researchers used 2D [5, p42] and 3D interfaces [6] to visualise the timbre space of synthesizers, enabling more efficient exploration. Timbre word to parameter mappings provide another intuitive means to find parameters [7], [8]. Perhaps the most intuitive way to program a synthesizer is by playing an instrument and deriving parameters from the live input [9] or even using the human voice [10], [11].

Some researchers have attempted to integrate automatic synthesizer programming technology with publicly and commercially available synthesizer systems. Dahlstedt used an interactive genetic algorithm (IGA) to help program the Nord Modular synthesizer, where the algorithm suggests variations on a synthesis patch to the user, who can then select their preferred variations for further 'breeding' [12]. More recently, Yee-King described a similar IGA driven synthesizer programmer implemented using web browser technology and released as open source software [13]. Both these systems followed earlier work with IGAs and sound synthesis by Johnson [14], and can be traced back further to Dawkins' Biomorphs concept [15].

IGA systems are not complete synthesizer programmers though; they aid in the process but do not carry it out. The SynthBot system was a complete synthesizer programmer, and it was able to program FM and analogue modelling synthesizers available in the industry standard, Virtual Studio Technology (VST) format [16]. Heise described a similar VST programmer, using a particle swarm optimisation technique instead of Synthbot's GA [17]. These complete programmer systems were able to match tones convincingly, but they also used simple synthesizers with less than 50 parameters. The work presented here uses a synthesizer in the VST format, but the synthesizer is considerably more complex than the synthesizers in the previous VST work.

We think that VSTs are an excellent and varied test bed for automatic synthesizer programmers - the KVR audio software database contains 3,765 VST plug-ins, with a variety of synthesis methods[1]. VSTs also offer the possibility of transferring the technology to a commercial product. This product would be able to interoperate with digital audio workstation (DAW) and plug-in software that musicians are already using. We note that VST is not the only option here - modular systems such as Native Instruments' Reaktor would be an interesting target, and researchers have already investigated automatic patch generation using the PureData system [18].

The last system we will review is Tatar et al.'s PresetGen, as it represents the current state of the art in sound matching with a commercially available synthesizer [19]. PresetGen used an advanced, multi-objective genetic algorithm to match real instrument and contrived sounds made using a software version of the OP-1 synthesizer made by Teenage Engineering. Contrived sounds are made with the synthesizer, so they can theoretically be matched perfectly if the settings can be found. The researchers also carried out a 'machine to human' trial, comparing human programmers to PresetGen, similarly to [16]. The system was found to have human competitive sound matching capabilities. We think that the PresetGen work could be extended by comparing the performance of the GA to other systems. Also, we think that the speed of the matching would need to be improved to make it viable for a commercial product, as it takes 5 hours on a 50 core supercomputer to match a 2 second sound. We address both of these challenges in our work - we compare several methods, and we investigate the possibility of near real-time sound matching.

In summary, automatic synthesizer programming is a fascinating and challenging area to investigate, with clear potential to impact on the methods used by industry professionals. In this paper, we build upon previous work but go further by making the three contributions enumerated below.

1) A description of three novel deep network approaches to automatic synthesizer programming.
2) A study comparing these new techniques to previously used techniques.
3) An open source, automatic synthesizer programming test bed, to support future work in this area.

II. Method

This section describes the sound matching problem in more detail and the methodology we have used to address it. It first describes the Dexed synthesizer and its parameters, then details the test sets used to evaluate our automatic programming techniques and finally describes the algorithmic techniques themselves.

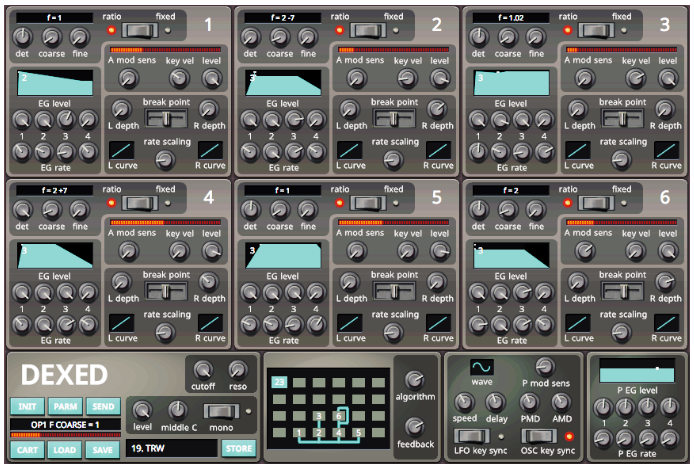


Fig. 1. The user interface for the Dexed synthesizer, showing repeated sets of controls for each of the six 'operators'.

A. The DX7 and Dexed synthesizers

The Dexed synthesizer is a virtual, software synthesizer which aims to model the synthesis algorithm found in the Yamaha DX7 synthesizer as closely as possible[2]. The DX7 implements frequency modulation (FM) synthesis, a technique which was first described by Chowning [20]. FM synthesis involves the generation of complex waveforms using a small number of oscillators (known as operators in DX7 terminology). These DX7 operators can modulate each other's frequency, where the modulation occurs at a rate higher than 20Hz, and which enables the synthesis of frequency sidebands. Dexed is available as a VST plug-in, which is a standard format developed by Steinberg, used to author effects processors and virtual synthesizers. VST plug-ins, such as Dexed, can be loaded into any VST compatible host program, meaning third parties can develop synthesizers, then musicians can use them inside a VST compatible, digital-audio workstation.

Yamaha released the DX7 in 1983, so it is not currently a state of the art synthesizer. However, several reasons make it an attractive subject for study. It is one of the best known and most commercially successful synthesizers of all time and is known for being difficult to program [21]. Also, Yamaha and other manufacturers have used variants of the underlying FM synthesis algorithm in newer synthesizers, making it a current technique. These synthesizers include the Native Instruments FM8 (2011), the Yamaha Montage (2016) and the Korg Volca FM (2016), the latter being a hardware emulation of the DX7. However, all these synthesizers rely heavily on preset sounds and do not offer innovative programming methods that give the user a greater deal of creative scope.

B. Details of the synthesis engine

The DX7 synthesis engine contains six digital oscillators, known as operators. Each operator has several available wave-

[1]http://www.kvraudio.com

[2]https://github.com/asb2m10/dexed

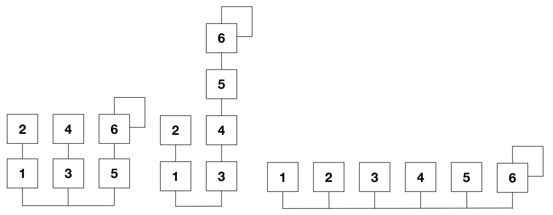


Fig. 2. Schematic of three of the available 32 Dexed/ DX7 algorithms. Numbered boxes are operators. Connections between operators imply the upper operator is modulating the frequency of the lower one. The synthesis engine connects operators on the bottom row to the audio output.

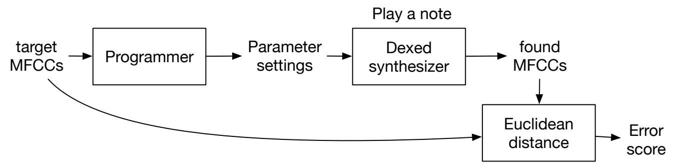


Fig. 3. The workflow we use to evaluate a programmer against a member of a test set. The programmer receives audio features as input. It generates settings for the synthesizer as output. We render the sound resulting from these settings and compared to the input sound to evaluate the programmer.

forms, two control envelopes, and various tuning and other controls. The Dexed DX7 emulator exposes the synthesis engine as 155, real-valued parameters in the range 0-1, which is the standard range for VST plug-ins. Figure 1 shows the Dexed graphical user interface (GUI), which reflects the structure of the underlying synthesizer. There are six operator control panels, labelled 1-6, showing the envelope, tuning, and other controls for each operator. There are global controls such as those for the overall amplitude envelope at the bottom right and there is a modulation routing display at bottom centre (the 'algorithm' in DX terminology). Figure 2 shows a clearer view of three out of the 32 available algorithms in the synthesizer. Each box represents an operator, and the lines represent modulation, where a modulator operator modulates the frequency of a carrier operator. Feedback is also present, where an operator modulates its own frequency.

C. The sound matching problem

The sound matching problem is solved by finding the parameters for a synthesizer that cause it to emit a sound that matches a target sound as closely as possible.

The experiment we have designed, and which is described below, aims to apply several automatic sound synthesizer programming techniques to the sound matching problem and to compare their performance.

We formalise the sound matching problem into a series of six test sets, each test more complex algorithmically than the last. A test set contains 30 members, where each member represents a single test for the synthesizer programmer. A test set member consists of a vector of parameters for the Dexed synthesizer and a vector of audio features. Therefore, a test set member represents a preset for the synthesizer and the sound it produces. The programmer must attempt to match the sound, without knowing the settings.

To generate the test sets, we sampled the parameters from a uniform distribution in the range 0-1. We then obtained the audio features by setting the parameters on the synthesizer, playing a note for one second, then extracting the features from the sound signal generated at the output of the synthesizer. The experimental framework described later allows this process to be completed automatically.

1) Audio features and evaluation: Many different audio features are appropriate for musical applications [22]. The early sound matching work used simple spectral features, but more recent work uses Mel Frequency Cepstrum Coefficients (MFCCs) [23]. We also use MFCCs in this study as they have been an effective audio feature for sound matching tasks [16], [17], monophonic instrument recognition tasks [24], and listener studies have shown that a movement in MFCC space is associated with a similar 'sized' movement in human perceived timbre space [25].

We used pyAudioAnalysis, an audio feature extraction library for Python to extract the MFCCs [26]. It uses the MFCC implementation from scikits.talkbox[3], which in turn uses the FFT and discrete cosine transform implementations from scipy.fftpack[4]. We set the system sample rate to 44,100Hz and the FFT window size to 2048 frames, which is approximately 50ms. We extracted the features from overlapping windows with a hop size of 1024 frames. We extracted 13 MFCC coefficients per frame, with the Mel filter bank beginning at 133Hz.

We evaluate an automatic synthesizer programmer against a member of a test set using the process shown in Figure 3. The programmer receives the target feature vector as its input. It then has to find out what the settings are that will cause the synthesizer to generate these features. The programmer outputs its estimate of the parameter settings. These settings are used to set up the synthesizer, which then plays a single note (MIDI note 24). We extract audio features from the output of the synthesizer and compute the Euclidean distance between these features and the target features. A good programmer will be able to get close to the target sound in feature space.

We evaluate the synthesizer programmers against every member in every test set, and this provides an overall score for each programmer. The test set contains sounds that have been generated with the synthesizer, referred to as *contrived* sounds in [19]. A perfect synthesizer programmer should be able to resynthesize contrived target sounds exactly.

Another approach would be to create a test set containing real instrument sounds, which the synthesizer could only approximate. We consider this to be a valid approach, but we prefer the contrived sound approach here because it provides a simple baseline against which the programmers can be evaluated, i.e. the possibility of an error of zero. With a real instrument sound as a target, there is no way to know if the algorithm is performing well in an absolute sense - it cannot be known if it has, in fact, found the closest sound that synthesizer can make to that instrument sound. With

[3]https://pypi.python.org/pypi/scikits.talkbox
[4]https://docs.scipy.org/doc/scipy/reference/fftpack.html

TABLE I
PARAMETER COUNTS AND OPERATOR COUNTS IN THE TEST SETS. EACH SUCCESSIVE TEST SET UNLOCKS MORE PARAMETERS AND AN EXTRA SYNTHESIZER OPERATOR MODULE.

Test set	Parameter count	Operator count
T1	22	1
T2	44	2
T3	66	3
T4	88	4
T5	110	5
T6	132	6

the contrived sounds, a perfect programmer can find a zero error sound. This evaluation does not consider the actual parameters that were used to generate the target sound, as the timbral distance is more important than the parameter distance. Therefore, it is acceptable for a programmer to make a sound using different settings than those in the test set, so long as it achieves a timbral match.

2) Difficulty of the test sets: Parameter freezing is used to make the test sets increasingly difficult. Set 1 is probably the easiest set as 133 out of the 155 parameters are frozen. This means that only 22 parameters vary between the items in the set. The variable parameters include the global envelope parameters and operator specific parameters. In test set 1, we choose the variable parameters such that the synthesizer is effectively only using one operator. In test set 2, two operators are unlocked, through to the full six operators in set 6. This arrangement requires that we choose the modulation routing algorithm carefully, to ensure that the varying parameters affect operators that will change the sound. There are 155 parameters in total within the synthesizer, and 23 of them are global parameters which affect the overall sound.

D. Automatic synthesizer programming techniques

We implemented five different synthesizer programming algorithms for this study, and we will begin this section by explaining why we chose these particular algorithms. In later sections, we present more details about the algorithms themselves and the hyperparameters that we selected.

The algorithms were a hill climber algorithm, a genetic algorithm, and three different neural networks. The hill climber was the most straightforward algorithm, and we included it in the set to provide a benchmark that could be used for comparison with the more sophisticated algorithms. We included a genetic algorithm (GA) because many previous studies have used them successfully. The inclusion of a GA also provides the opportunity for comparison with previous work in this area.

The three neural networks we used were a multilayer perceptron (MLP), a long, short-term memory network (LSTM) and a bi-directional LSTM with highway layers which we refer to as LSTM++ [27], [28], [29].

The MLP was the most straightforward network algorithm in this set of three, and we included it as a benchmark for the other more complex neural algorithms. The first of these more complex algorithms was an LSTM, which is an example of a recurrent neural network. We added the LSTM as it is reported to perform better at sequential data modelling tasks than the MLP, especially at speech analysis tasks which have similarities to timbre analysis tasks. For example, Graves et al. showed that LSTMs could outperform deep feedforward architectures on the TIMIT phoneme recognition benchmark [30].

While there are simpler recurrent networks than an LSTM, we chose to use it because its architecture provides better memory persistence, reducing the vanishing gradient problem which can limit the memory to a few frames in simpler recurrent neural networks [31].

The final neural network algorithm was LSTM++. It begins with a bidirectional LSTM, which is an extended form of an LSTM. It allows the network access to the complete sequence of input frames in forward and reverse order, instead of just forward order as in the standard LSTM [28]. The bidirectional LSTM network, therefore, has access to the past and future context of an input frame, which has proven useful in other audio signal analysis tasks such as onset detection [32].

Highway layers provide a means for data to flow through several layers of the network uninhibited, as well as through the standard means of nodes with associated activation functions. Highway layers are reported in previous work to alleviate further the vanishing gradient problem such that the network can better model longer input sequences, and LSTMs with these layers have been shown to outperform standard LSTM networks in speech recognition tasks [29]. We selected this architecture as the sound synthesis algorithm and feature extractor both have strong forward and reverse relationships in their data. The synthesis algorithm also has interdependence between its components. This architecture seemed to provide the maximum potential for modelling these characteristics.

Now we shall describe our specific implementations of these algorithms, along with their hyperparameters.

1) Hill climber: The hill climber (HC) is the simplest method used in the evaluation. We based it on the continuous space HC described in [33]. The HC begins at a random point in parameter space by sampling the synthesizer parameters from a uniform distribution in the range 0-1. It then iterates through the parameters, creating five variations of each parameter in turn. The five variations consist of a large decrement, a small decrement, nothing, a small increment and a large increment. The synthesizer plays a note with each variation of the parameter settings, and we extract audio features. The algorithm compares each variant to the target sound in audio feature space. The error is the Euclidean distance between the two sounds. HC chooses the variant with the lowest error and moves to the next parameter on that variant. If the lowest error is the ‘nothing’ variant, the size of the variations is made smaller for the next round. The HC stops when it fails to decrease the error for several iterations.

2) Genetic algorithm: The genetic algorithm (GA) used in this study is a standard model which uses roulette wheel selection, crossover and elitism. The GA works by manipulating a population of solutions to the sound matching problem, where each solution consists of a set of parameters for the synthesizer. The GA initialises the solutions in the population to random vectors, sampled from a uniform distribution. It assigns each a fitness score by rendering a sound with those settings, extracting features and comparing them to the target features. The fitness is the reciprocal of the Euclidean distance between the candidate features and the target features. The closer the sounds are to the target sound, the higher their fitness.

The GA generates the next generation of solutions by selecting multiple pairs of 'parents', where the probability of selection is proportional to the fitness. The parameter vectors of the two parents are crossed over to produce a 'child' containing some parameters from each parent. The child undergoes mutation, where some of its parameter values are increased or decreased by a fixed amount, and the mutant is added to the next generation. The GA includes elitism, which results in 10 of the fittest individuals being added unchanged to the next generation. The optimisation process ends when the population has converged, which means it has not increased in mean fitness for some number of iterations.

The GA parameters were as follows: the population size was 200, the chance of mutation per parameter per child was 0.01, the mutation size was 0.1, and the number of elite individuals per generation was 10.

3) Multilayer perceptron network: The multilayer perceptron (MLP) network used in this study is a simple form of feed forward neural network, provided as a baseline for comparison with more sophisticated network architectures. We implemented the MLP and the other neural networks using the Tensorflow machine learning library [34].

The MLP had an input layer, an output layer and three hidden layers. The input layer had one unit per MFCC coefficient per frame, which allowed one complete feature vector to be passed to the input. As mentioned above, we rendered the sounds for 1 second, which resulted in 27 MFCC frames, with 13 coefficients each. Therefore, the input layer had 351 units. The output layer had the same number of units as the number of parameters for the test set, for example, test set one had 22 parameters. The hidden layers have 50, 40 and 30 units, respectively. We experimented with several topologies, including deeper and broader ones, but we chose this topology because it showed a smooth decrease in the training set error over time, compared to other topologies. The units use the standard rectified linear unit (ReLU) for their activation function, which is commonly used in audio analysis tasks [35].

A training set consisted of 60,000 audio feature - parameter pairs, in the same format as the test set members. There was a different training set for each test set, with the frozen parameters configured accordingly. The error function for training was the root mean squared error between the predicted settings and the actual settings used to make the target sound. This parameter based error is different from the error function in the HC and GA, which used the error between the actual sound the synthesizer generates and the target sound. It was necessary to use the parameter error in the neural networks because it is not possible to backpropagate the error via the synthesizer itself or to otherwise include sound rendering in the training process.

For each of the six test sets, we trained the network for 1000 epochs using the Adam optimiser. Adam is a stochastic, gradient-based optimisation technique which is appropriate for complex network topologies and large data sets [36]. It also has intuitive hyperparameters, making it more straightforward to tune than other optimisers. We used dropout in the training process for the MLP. Dropout is a regularisation technique that randomly blocks units from propagating the signal during training, and which has been shown to prevent overfitting caused by co-adaptation between units [37].

4) Long Short-Term Memory network: The next neural network architecture was a Long Short-Term Memory network (LSTM).

LSTMs use memory units to achieve the gain in memory persistence. A memory unit is a data storage unit in the network that uses gates to control read and write access. We used the tflearn LSTM implementation[5], which is based on the standard LSTM [27].

We constructed an LSTM network layer with 128 units. The input layer had 13 units so that we could feed in the features frame by frame. It had the same number of output units as the MLP, equal to the number of parameters in that particular test set. The LSTM layer connects to the outputs via an additional, fully connected layer with ReLU activation functions.

We used an Adam optimiser to train the network for 1000 epochs, with the same 60,000 item training sets used to train the MLP. We also used the same synthesizer parameter error metric. The network was trained in batch mode with batches of 32 and a learning rate of 0.001. To evaluate the network against the test set, we used the output values generated after the last input frame had been fed in since the output varied with each frame.

5) Bidirectional LSTM with highway layers: The final neural network architecture was a bidirectional LSTM with highway layers (LSTM++). Figure 4 illustrates the architecture of the LSTM++ network. We based this architecture on a standard model from the tflearn library, then found it worked well on our tests, so did not modify it further.

The LSTM++ begins with an input layer with 26 nodes, 13 for the forward and 13 for the reverse audio feature sequence. The input layer feeds into two separate instances of our previous 128 unit LSTM, one for each direction. The LSTM blocks are fully connected to a re-shaping layer, which scales down to 64 units for input to the highway layers and uses an exponential linear unit (ELU) activation function. The units in the highway layers also use the ELU activation function. They are stacked on top of each other, each with 64 units, each fully connected to the previous layer. The final highway layer is fully connected to the output. An Adam optimiser trained the network using the standard 60,000 item training set and parameter error metric, for 1000 epochs.

E. Synthesizer programmer framework

We think that the lack of a set of repeatable benchmarks and the lack of a reusable software framework have hampered

[5]https://github.com/tflearn/tflearn

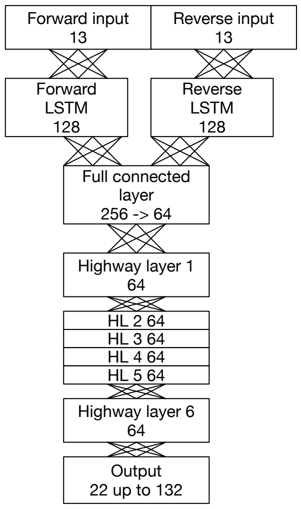


Fig. 4. The network topology for the LSTM++ network. There are two parallel LSTM blocks for forward and reverse input processing. These blocks feed into a reshaping layer, which prepares the signal for the highway layers. There are six highway layers, each one fully connected to the previous one. The output layer has the number of outputs dictated by the test set.

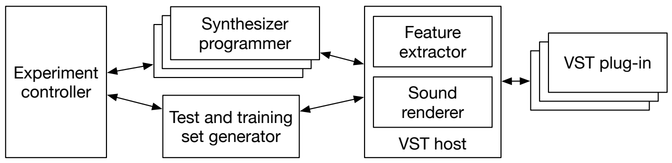


Fig. 5. The synthesizer programmer testing framework. The experiment controller carries out the synthesizer programming experiment by interacting with the programmers and the test and training data generator. The test and training set generator interacts with the VST host to produce sets of parameter settings and feature pairs. The VST host component loads VST plug-ins, renders their output and extracts features.

the development of research into automated synthesizer programming. Benchmarks and software frameworks are common in other areas of machine learning research. Therefore, we provide the software framework used for this study as an open source repository [38], [39]. The framework is accessible from Python, and it provides a VST host with offline sound rendering and feature extraction functions. It also contains our machine learning models with testing and training data. Figure 5 illustrates the main components of the framework. We hope that these tools make the investigation of machine learning and synthesizer programming more accessible for future researchers.

III. Results

This section presents the results of running the synthesizer programming techniques across the 180 items in the six test sets. Table II presents the mean results achieved in each test set by each technique, along with the mean result per test set across all techniques. The final column shows the total error over all items in all test sets per programming technique. Lower numbers indicate better performance.

Figure 6 visualises the performance of each technique against every item in the test sets. The darkness of a line indicates the distance in feature space between the target sound and the sound generated by the programmer. Dark lines represent poor performance, and lighter lines indicate better performance. This visualisation makes it possible to gain an insight into the consistency of the performance of a given technique. Consistent performance would manifest as a horizontal strip with many similarly shaded lines.

Figure 7 shows the distributions of MFCC errors over all 180 test items for all techniques. This visualisation provides another perspective on the consistency of performance, and it illustrates the range of errors achieved. Lower errors are desirable, so distributions weighted to the left indicate better performance. Informal listening tests with the resulting sounds indicated that very similar sounding sounds have MFCC distances of 10 to 15 or less. Sounds with an MFCC error in excess of 20 are unlikely to sound very similar.

TABLE II
Summary of results for all techniques in all test sets. The values shown are the mean values achieved across each test set, except for the final column which shows the total errors over all test tests per algorithm.

Method	T1	T2	T3	T4	T5	T6	Total
HC	**17.29**	**20.95**	23.18	**19.9**	22.35	**21.96**	3769
GA	23.2	37.59	32.03	27.71	28.12	31.32	5399
MLP	24.53	36.13	33.66	26.76	32.78	31.38	5557
LSTM	24.01	37.26	33.17	24.93	29.71	32.76	5455
LTSM++	19.47	23.15	**21.48**	20.66	**18.14**	22.59	**3765**
Mean	21.7	31.01	28.70	23.99	26.22	28.00	

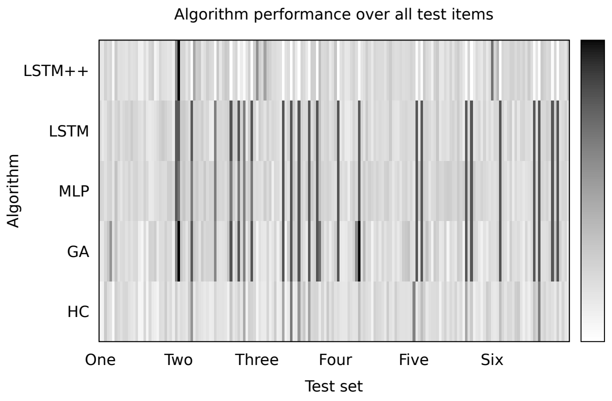


Fig. 6. Performance of the techniques against all test items. Each vertical strip represents a technique being applied to a single test item. A light colour indicates a good match, darker indicates a worse match.

IV. Analysis

This section discusses the performance of the different synthesizer programmers regarding accuracy and speed. We consider the difference between the search-oriented techniques (hill climber and genetic algorithm) and the modelling oriented techniques (neural networks) and the reasons for the variation in performance between them. We end the section by discussing the feasibility of making this technology available to end users.

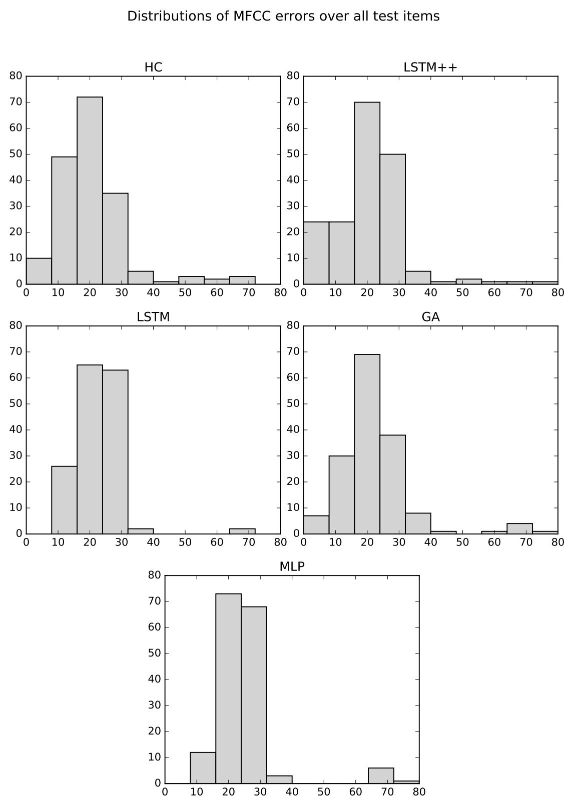


Fig. 7. A comparison of the distribution of MFCC errors for all programmers over the complete, 180 item test set. The LSTM++ has more very low errors. Errors of less than 10 indicate perceptually very similar sounds.

A. Sound matching accuracy

Table II presents the performance of each programmer against each test set. The final column contains the summed errors over all test sets. It shows that the LSTM++ network achieved the lowest total error. The simplest programmer, HC, achieved the second lowest error. The visualisation in Figure 6 shows that HC was a more consistent performer than the LSTM++ network. The HC row in the figure contains mostly light grey lines, indicating consistently low errors. The LSTM++ row contains a mixture of grey, black and white lines, indicating low, high and medium errors.

Figure 7 shows the numerical range of MFCC errors achieved by the different programmers. We suggested above that errors greater than 20 indicate sounds that are not perceptually similar, based on informal listening tests. LSTM++ and HC both have approximately 25% of their matches in this 'close' category, but LSTM++ has twice as many as HC in the 'very close', $0 - 10$ error range.

Surprisingly, the GA, which is considered the standard technique in the literature, did not perform as well as the simpler HC programmer. It generated several large errors in the $60 - 80$ range. However, it was a very simplistic implementation, and other studies ran their GAs much longer, with larger populations. We shall discuss the time required to run the GA, and how this might impact its usefulness in a user-facing system, below.

B. Search vs. modelling

The techniques fall into two categories: search and modelling. The HC and the GA search through the space of possible sounds by iteratively varying the parameters of the synthesizer and examining the resulting audio features. The neural networks are modelling a mapping from feature space to parameter space.

Successful searching of the space of possible sounds depends on factors such as the smoothness of the error surface and the presence therein of local minima. A smooth error surface in the context of synthesizer programming is one where small changes in parameter space result in small changes in feature space. Roth and Yee-King considered the error surface of a simple, two parameter FM synthesizer, with parameters in the range 0-1 [3]. They noted that a change of 0.1 in one parameter could produce a large change in feature space, equivalent to the feature difference between two real instruments. We have not analysed the Dexed synthesizer in this way, but several of its parameters are comparable to those in that simple FM synthesizer, so it is likely to have a rough error surface.

Hill climbers have trouble with rough error surfaces as they become trapped behind 'error barriers'. The 'five variations' feature of our HC alleviates this somewhat, enabling different sized jumps through parameter space. HCs also have trouble with local minima. Premature convergence indicates that the search space contains local minima. The horizontal grey HC strip in Figure 6, and the central weighting of the HC histogram in Figure 7 show that the hill climber consistently converged on sub-optimal solutions, given the lowest possible error was zero. Therefore the search space does contain local minima.

The modelling based methods face a different set of problems. The training data for the models contains feature vectors as inputs and parameter settings as outputs. They must learn a mapping from inputs to outputs, which is effectively the inverse of the process of sound synthesis and feature extraction. We have not examined the trained network models to find out how they work, but certain characteristics of the problem might explain the performance of the different architectures - consistency through time and the structure of the synthesis algorithm.

The timbral output of the synthesizer is consistent through time as it is designed to model musical instruments, which also produce consistent timbres, once the attack portion of the sound is complete. Therefore, there is repetition in the feature vector, especially between frames that are close together in time. Architectures that are capable of modelling the consistent nature of the feature vector, perhaps through recurrency, are likely to perform better. The LSTM network did not perform much better than the MLP though. The LSTM++ added bidirectional and highway layer components to the LSTM, and this seemed to improve the performance significantly. The bidirectional component is intended to unlock more information about the signal, and the highway component is intended to alleviate the vanishing gradient problem further. It is not clear which of these provided the greatest impact upon performance, but the LSTM++ network showed strong potential, with several very low error scores.

The LSTM++ did not achieve the best performance across all the test sets. This observation is most likely a side effect of the limited size of the test sets, as opposed to being a characteristic of the synthesizer parameters available in each test set. Small test sets might contain ‘easy’ or ‘hard’ sounds by chance, whereas larger test sets would contain a fairer representation of the difficulty of a given set of un-frozen parameters. The requirement to test against all items in the test set with all techniques limited the practical size of the test sets, so it was not possible to go further in examining the relative difficulty of the different test set parameter options.

C. Sound matching speed

The algorithms have a training phase and a sound matching phase. The time taken to complete these stages varies significantly between the search and modelling approaches.

The search based approaches did not require training, but the modelling approaches did. We trained the MLP for 1000 epochs with a training set of 60,000 members, per test set, which took approximately a day on a Thinkpad workstation with an Intel i7 CPU and NVIDIA Quadro 2200 GPU. It took a similar length of time to train the other neural networks. However, once trained, the networks could generate parameter settings in near real time.

By contrast, the search based approaches needed to carry out a complete search for each sound match. The evaluation of a candidate in the HC and GA took around 40ms, which included rendering the sound, extracting features and comparing to the target sound. One epoch in the HC involved evaluating five variations of up to 132 parameters, which takes around 26 seconds, or seven hours for 1000 epochs of the most complex test set. The GA evaluates less per epoch and is not sensitive to the number of parameters, so it takes around two hours for 1000 epochs.

For comparison, the state of the art PresetGen system took five hours on a 50 core supercomputer to match a two second sound. Our simple search implementations would likely approach this speed if they were multi-objective, had a similar population size and if the synthesizer was as complex as the OP-1. We do not think it is feasible to implement a user-facing system if it requires that many CPU cycles per sound match, so there is strong motivation to pursue the network approach given our positive, preliminary results.

D. Generalisation across pitch ranges

The test sets contain a single sound for each synthesizer preset, which is generated by playing a fixed musical note (in this case MIDI note 24) on the synthesizer for a fixed length of time and with a fixed intensity. We did not evaluate the timbral variation when playing different notes from the chromatic scale with the same parameters, nor when varying the intensity of the chosen note. The timbre of real instruments varies significantly with intensity and fundamental frequency, as discussed by Loureiro et al. [40]. We would need to consider this factor when designing presets to emulate musical instruments across a range of pitches and intensities. For example, we could generate multiple presets for different pitches and intensities.

E. Real world application

We will now consider how the techniques discussed in the paper might be made available to end users. The technology has two characteristics which make it amenable to this: it uses an industry standard plug-in format, and it can match sounds in near real-time on consumer hardware. Using the VST plug-in format means the system can work with synthesizer software that professional musicians already own. Real-time sound matching means the user can work interactively with the programmer and more easily integrate it into their workflow.

The system would consist of a library of pre-trained network models for different VSTs and a user-facing program. The library of models would reside on a server and would be regularly updated with new models and plug-ins. The user-facing program would search for installed plug-ins on the user’s machine, then pull down the models for those plug-ins from the server. To match a sound, the user would provide the target sound; then the system would instantiate the trained models for their synthesizers and generate settings. It would play back the results using the plug-ins themselves, and the user could save their favourite sound match as a preset for the plug-in, accessible from other programs (such as DAWs).

However, we should note that the sound matching performance of our best performing architecture is not good enough for a user-facing application yet. At least, not for synthesizers as complex as the Dexed. Currently, less than 25% of the matches can be considered close matches, as indicated by our informal listening tests. A user-facing, sound matching system should present the user with sounds that they are likely to consider similar to the target, most of the time. A formal listening test would allow us to establish the error constraints for such a system more clearly.

V. Conclusion

In this paper, we have applied several machine learning and optimisation techniques to the problem of sound matching. We will now discuss the contributions claimed in the introduction, in light of the results and analysis we have presented.

The first contribution is the design and evaluation of three deep network architectures suitable for application to the sound matching problem. The key feature of these architectures, going beyond the previous work, is their ability to match sounds in near real-time. This feature offers the possibility that users can interact with more complicated synthesis architectures more efficiently and intuitively than they could previously, opening up new creative possibilities.

The second contribution is a comparative study of several synthesizer programming techniques. This study and its method are important because they allow us to benchmark new techniques against techniques reported in previous work. With these benchmarks, it is easier to demonstrate progress in the field. There are many potential extensions to our method, such as comparing different synthesizers, matching real instrument or environmental sounds, using different audio features, and including a formal listener study. We hope that this comparative study model will feature in future research in this area.

The final contribution is an open source, automatic synthesizer programmer test bed. This system provides a VST host designed for this task, with offline rendering and feature extracting capabilities, a set of example synthesizer programmers and example training and test sets. With these elements, future researchers can more easily repeat our study and develop the extensions mentioned above.

In future work, we aim to improve the sound matching performance of the deep network architecture and to evaluate its ability to program a range of different synthesizers. Part of this work will involve the investigation of more audio feature types, for example, the Wavenet autoencoder offers a novel means to model the features of audio signals using machine learning [41]. A listener study would allow us to understand the connection between measured and perceived feature distance better, and therefore to better evaluate system performance. If we can improve the performance sufficiently, we would like to make it available to professional creatives who work with synthesizers to see what the creative possibilities are for a real-time, automatic synthesizer programmer.

REFERENCES

[1] A. Horner, J. Beauchamp, and L. Haken, "Machine tongues XVI: Genetic algorithms and their application to FM matching synthesis," *Computer Music Journal*, vol. 17, no. 4, pp. 17–29, 1993.

[2] T. Mitchell, J. Charles, and W. Sullivan, "Frequency modulation tone matching using a fuzzy clustering evolution strategy," in *Audio Engineering Society 118th Convention, Barcelona, Spain,*, 2005.

[3] M. Roth and M. Yee-King, "A Comparison of Parametric Optimization Techniques for Musical Instrument Tone Matching," in *Audio Engineering Society Convention 130*, 2011.

[4] D. Arfib, J. M. Couturier, L. Kessous, and V. Verfaille, "Strategies of mapping between gesture data and synthesis model parameters using perceptual spaces," *Organised Sound*, vol. 7, no. 2, pp. 127–144, 2002.

[5] M. J. Yee-King, "Automatic sound synthesizer programming: techniques and applications," Ph.D. dissertation, University of Sussex, 2011.

[6] S. Fasciani, "TSAM : a tool for analyzing , modeling , and mapping the timbre of sound synthesizers," in *Proceedings of 13th Sound and Music Computing Conference (SMC 2016)*, 2016, pp. 129–136.

[7] C. G. Johnson and A. Gounaropoulos, "Timbre interfaces using adjectives and adverbs," in *Proceedings of the 2006 conference on New interfaces for musical expression*, 2006, pp. 101–102.

[8] G. Kreković, A. Pošćić, and D. Petrinović, "An algorithm for controlling arbitrary sound synthesizers using adjectives An algorithm for controlling arbitrary sound synthesizers using," *Journal of New Music Research*, vol. 45, no. 4, pp. 375–390, 2016.

[9] M. Yee-King, "An Automated Music Improviser Using a Genetic Algorithm Driven Synthesis Engine," in *Workshops on Applications of Evolutionary Computation,*, ser. Lecture Notes in Computer Science, vol. 4448. Springer, 2007, pp. 567–577.

[10] D. Stowell, "Making music through real-time voice timbre analysis: machine learning and timbral control," Thesis (PhD), Queen Mary University of London, 2010.

[11] M. Cartwright and B. Pardo, "SynthAssist: Querying an Audio Synthesizer by Vocal Imitation," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2014, pp. 363–366.

[12] P. Dahlstedt, "Creating and exploring huge parameter spaces: Interactive evolution as a tool for sound generation," in *Proceedings of the 2001 International Computer Music Conference*, 2001, pp. 235–242.

[13] M. J. Yee-King, "The Use of Interactive Genetic Algorithms in Sound Design : A Comparison Study," *ACM Computers in Entertainment*, vol. 14, no. 3, 2016.

[14] C. G. Johnson, "Exploring the sound-space of synthesis algorithms using interactive genetic algorithms ." in *Proceedings of the AISB'99 Symposium on Musical Creativity*, 1999.

[15] F. Morchen, A. Ultsch, M. Thies, and I. Lohken, "Modeling timbre distance with temporal statistics from polyphonic music," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 14, no. 1, pp. 81–90, 2006.

[16] M. Yee-King and M. Roth, "Synthbot: An unsupervised software synthesizer programmer," *Proc. International Computer Music Conference*, no. October, pp. 184–187, 2008.

[17] S. Heise, M. Hlatky, and Jörn Loviscach, "Automated Cloning of Recorded Sounds by Software Synthesizers," in *127th Audio Engineering Society Convention*, 2009.

[18] M. Macret and P. Pasquier, "Automatic Design of Sound Synthesizers as Pure Data Patches using Coevolutionary Mixed-typed Cartesian Genetic Programming," in *Proceedings of the 2014 Conference on Genetic and Evolutionary Computation*, 2014, pp. 309–316.

[19] K. Tatar, M. Macret, P. Pasquier, K. Tatar, M. Macret, and P. Pasquier, "Automatic Synthesizer Preset Generation with PresetGen Automatic Synthesizer Preset Generation with PresetGen," *Journal of New Music Research*, vol. 45, no. 2, pp. 124–144, 2016.

[20] J. Chowning and D. Bristow, *FM Theory and Applications: By Musicians for Musicians*. Hal Leonard Corp, 1987.

[21] B. K. Shepard, *Refining sound: A practical guide to synthesis and synthesizers*. Oxford University Press, 2013.

[22] G. Peeters, B. L. Giordano, P. Susini, N. Misdariis, and S. McAdams, "The timbre toolbox: Extracting audio descriptors from musical signals," *The Journal of the Acoustical Society of America*, vol. 130, no. 5, pp. 2902–2916, 2011.

[23] S. Davis and P. Mermelstein, "Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences," *IEEE transactions on acoustics, speech, and signal processing*, vol. 28, no. 4, pp. 357–366, 1980.

[24] A. Eronen, "Comparison of features for musical instrument recognition," in *IEEE Workshop on the Applications of Signal Processing to Audio and Acoustics*, 2001, pp. 19–22.

[25] H. Terasawa, M. Slaney, and J. Berger, "Perceptual distance in timbre space," in *Proc. Int. Conf. Auditory Display*, 2005, Conference proceedings (article).

[26] T. Giannakopoulos, "pyaudioanalysis: An open-source python library for audio signal analysis," *PloS one*, vol. 10, no. 12, 2015.

[27] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[28] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional LSTM and other neural network architectures," *Neural Networks*, vol. 18, no. 5, pp. 602–610, 2005.

[29] Y. Zhang, G. Chen, D. Yu, K. Yaco, S. Khudanpur, and J. Glass, "Highway long short-term memory rnns for distant speech recognition," in *IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2016, pp. 5755–5759.

[30] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *IEEE International Conference on Acoustics, Speech and Signal Processing*, no. 3, 2013, pp. 6645–6649.

[31] M. Sundermeyer, R. Schl, and H. Ney, "LSTM Neural Networks for Language Modeling," in *Thirteenth Annual Conference of the International Speech Communication Association*, 2012.

[32] F. Eyben, B. Sebastian, and A. Graves, "Universal Onset Detection with Bidirectional Long Short-Term Memory Neural Networks." in *11th International Society for Music Information Retrieval Conference*, 2010, pp. 589–594.

[33] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.

[34] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, and Others, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[35] M. D. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. V. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, and G. E. Hinton, "On rectified linear units for speech processing," in *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 3517–3521.

[36] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, pp. 1–15, 2015.

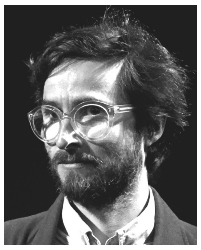
[37] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*, pp. 1–18, 2012.

[38] Leon Fedden, “Renderman: VST Host in Python,” 2017. [Online]. Available: https://doi.org/10.5281/zenodo.1079884

[39] L. Fedden and M. Yee-King, “Automatic VST Programmer Experimental Framework,” 2017. [Online]. Available: https://doi.org/10.5281/zenodo.1080859

[40] M. A. Loureiro, H. B. D. Paula, and H. C. Yehia, “Timbre Classification Of A Single Musical Instrument.” in *5th International Conference on Music Information Retrieval*, 2004.

[41] E. Kang, J. Min, and J. C. Ye, “A deep convolutional neural network using directional wavelets for low-dose X-ray CT reconstruction,” *arXiv preprint arXiv:1610.09736*, 2016.



Matthew John Yee-King is a Senior Lecturer in the Department of Computing at Goldsmiths College, University of London. In the past he has worked as a professional electronic music producer, releasing music on Warp and Rephlex records. His current research interests include the application of Artificial Intelligence techniques in the music composition and production process and the development of education technology for people learning how to write code.

Leon Fedden is a student at Goldsmiths, University of London. He recently was a member of the winning team in an international creative AI competition and is now completing his masters. His work focuses on creating, visualising and understanding computational learning systems that analyse and generate various types of media.

Mark dInverno is Professor of Computer Science at Goldsmiths, University of London. He has been a Principal or Co-Investigator on a wide range of EU and UK projects including sharing cultural experiences, AI systems to inspire human creativity and social media systems for online music learning. He has 200 peer-reviewed publications with recent focus on interdisciplinary research at the interface of computer science with the arts, humanities and social sciences.