Learning Parameter Spaces in Neural Modeling of Audio Circuits

Otto Mikkonen

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 12.8.2022

Supervisor

Prof. Vesa Välimäki

Advisor

MSc Alec Wright

Aalto University
School of Electrical
Engineering

Aalto University
School of Electrical
Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Otto Mikkonen

Title Learning Parameter Spaces in Neural Modeling of Audio Circuits

Degree programme Computer, Communication and Information Sciences

Major Acoustics and Audio Technology **Code of major** ELEC3030

Supervisor Prof. Vesa Välimäki

Advisor MSc Alec Wright

Date 12.8.2022 **Number of pages** 60 **Language** English

Abstract

This thesis studies black-box virtual analog modeling formulated as a machine learning sequence modeling task within the category of supervised learning problems. The focus is on learning scenarios where the modeling targets have multiple user controls, and the aim of the thesis is to evaluate how the properties of the training datasets affect the generalization of the learning algorithm.

To study the problem, three nonlinear analogue sound processors were modeled using a recurrent neural network consisting of a Gated Recurrent Unit and a fully-connected output layer. For each target device, two groups of datasets, seven in total, were constructed, using SPICE simulations of the targets. The difference between the datasets is in the density of the sampling grid used for setting the user controls of the targets, as well as in the number of input/output pairs corresponding to each distinct value of each of the controls.

For the targets considered during the study, the sparsest sampling grid using only three possible values for each of the user controls was found inadequate for the models to generalize over the testsets used for evaluation. Increasing the sampling density was seen improving the model performance in most cases, with some targets also portraying clear advantages with increasing the number of input/output pairs corresponding to each distinct value of the user controls. According to the study, a sampling grid with five points would appear as a good baseline for training neural networks on targets with multiple user controls when no further investigations in the sampling density can be afforded.

For future work, the experiments could be extended to include global scaling of the dataset size while keeping the constraints for sampling the parameter spaces, as well as combining the data generation and training procedures to a single loop, allowing for potentially infinite variety within the datasets.

Keywords Virtual analog modeling, black-box modeling, machine learning, sequence modeling, recurrent neural networks

Preface

This master's thesis work was carried out at the Aalto Acoustics Lab in the Department of Signal Processing and Acoustics (SPA) during the first half of 2022, with funding provided by the Nordic Sound and Music Computing Network (NordicSMC) and computational resources by the Aalto Science-IT project.

I want to thank my supervisor Prof. Vesa Välimäki and my advisor MSc Alec Wright for the invaluable quidance and support along the way, including brainstorming for the thesis topic, the numerous indispensable instructions and discussions, as well as generally for being inspirational characters in the research landscape, just to name a few things. Without them, this thesis would have never reached the quality it did in the end.

I also want to thank Fabian and Julian from Native Instruments for encouraging me in taking my Master's Degree at the Aalto University, which proved to be an exceptionally good decision. I also want to express my gratitude to Christian and the KOMA Elektronik gang for providing me with an encouraging space to grow as a young audio engineer.

Finally, on a more personal level, I want to thank my mother for always being supportive with me finishing my studies, as well as my partner Emilia for being supportive, well, most of the time.

in Helsinki, Finland - August 12th, 2022

Otto Mikkonen

Contents

Symbols and abbreviations

Symbols

$[a, b]$	a closed interval from a to b, distinction to vector $[x_1, x_2]$ set by context
$\mathbb{B}$	a mini-batch
$\mathbb{D}$	a dataset
$\mathbb{N}$	the set of natural numbers
$\mathbb{R}$	the set of real numbers
$\mathbb{R}^n$	the n-dimensional space of real numbers
$\mathbb{R}^{n \times m}$	the $n \times m$-dimensional space of real numbers
$\mathcal{U}$	uniform distribution
x	a scalar
x_i	scalar-valued element of vector $\mathbf{x}$ at index i
x_{ij}	scalar-valued element of matrix $\mathbf{X}$ at row index i, column index j
$x[n]$	scalar-valued element of sequence $\mathbf{x}$ at time step n
$x^{(i)}$	i'th scalar-valued element of a set
$[x_1, x_2]$	a vector with elements x_1, x_2
$[x[1], x[2]]$	a scalar-valued sequence with elements $x[1], x[2]$
$\{x^{(1)}, x^{(2)}\}$	a set with elements $x^{(1)}, x^{(2)}$
(x_1, x_2)	a tuple with elements x_1, x_2
$\mathbf{x}$	a vector / scalar-valued sequence
$\mathbf{x}[n]$	vector-valued element of sequence $\mathbf{X}$ at time step n
$\mathbf{x}^{(i)}$	i'th vector-valued element of a set
$[\mathbf{x}_1, \mathbf{x}_2]$	a matrix with elements $\mathbf{x}_1, \mathbf{x}_2$
$[\mathbf{x}[1], \mathbf{x}[2]]$	a vector-valued sequence with elements $\mathbf{x}[1], \mathbf{x}[2]$
$\mathbf{X}$	a matrix / vector-valued sequence
$\mathbf{X}^{(i)}$	i'th matrix-valued element of a set
$\mathbb{X}$	a set
$\mathcal{X}$	a distribution
$\mathbb{Z}$	the set of integers
$\boldsymbol{\theta}$	neural network weights
λ	eigenvalue
$\boldsymbol{\lambda}$	eigenvector
$\boldsymbol{\Lambda}$	matrix of eigenvectors
ϕ	a single user control of a modeling target
$\boldsymbol{\phi}$	user controls of a modeling target

Operators

$\mathbf{x} \in \mathbb{R}^n$	vector $\mathbf{x}$ is a member of $\mathbb{R}^n$
$\mathbf{x} \sim \mathcal{D}$	vector $\mathbf{x}$ is drawn from distribution $\mathcal{D}$
$\mathbf{x}^T$	transpose of vector $\mathbf{x}$
$x \cdot y$	product of scalars x and y
$\mathbf{x}\mathbf{y}$	matrix product of vectors $\mathbf{x}$ and $\mathbf{y}$
$\mathbf{x} \odot \mathbf{y}$	Hadamard (element-wise) product of vectors $\mathbf{x}$ and $\mathbf{y}$
$\mathbf{x} * \mathbf{y}$	convolution of vectors $\mathbf{x}$ and $\mathbf{y}$
$\dot{\mathbf{x}}$	time derivative of $\mathbf{x}$
$\nabla_{\mathbf{x}}$	gradient with respect to $\mathbf{x}$
$f(\mathbf{x})$	a function of $\mathbf{x}$
$f_{\mathbf{x}}(\cdot)$	a function parametrized by $\mathbf{x}$
$f : \mathbb{A} \to \mathbb{B}$	a function with domain $\mathbb{A}$ and range $\mathbb{B}$
$\mathcal{L}(\cdot)$	loss function
$\mathcal{E}(\cdot)$	expected value
$\phi(\cdot)$	nonlinear transformation
$\sigma(\cdot)$	sigmoid function
$\lceil \cdot \rceil$	ceiling function
$\lvert x \rvert$	absolute value of x
$\lvert \mathbf{x} \rvert$	length of vector $\mathbf{x}$
$\lvert \mathbb{X} \rvert$	number of elements in $\mathbb{X}$
Δ	difference operator
$\mathbf{x} \approx \mathbf{y}$	elements of vectors $\mathbf{x}$ and $\mathbf{y}$ are approximately equal
$\hat{x}$	estimate of scalar x
$\hat{\mathbf{x}}$	estimate of a vector $\mathbf{x}$
$\hat{f}(\cdot)$	estimate of function f
$\sum_i$	sum over index i

Abbreviations

BJT	bipolar junction transistor
BPTT	backpropagation through time
CNN	convolutional neural network
DSP	digital signal processing
DDSP	differentiable digital signal processing
ESR	error-to-signal ratio
EQ	equalizer
FIR	finite impulse response
GRU	gated recurrent unit
IIR	infinite impulse response
JFET	junction field effect transistor
LSTM	long short-term memory
LTI	linear time-invariant
ML	machine learning
MLP	multilayer perceptron
NN	neural network
ODE	ordinary differential equation
op amp	operational amplifier
RNN	recurrent neural network
SGD	stochastic gradient descent
SSGRU	state-space gated recurrent unit
STD	standard deviation
STN	state trajectory network
SVF	state variable filter
TCN	temporal convolutional network
TBPTT	truncated backpropagation through time
VA	virtual analog

1 Introduction

Virtual Analog (VA) modeling is a branch of digital signal processing (DSP) that attempts to recreate analogue and electromechanical audio hardware devices in software form [1]. These attempts are motivated jointly by the ongoing digitization of the music production and composition environments; disadvantages associated with analogue equipment, such as higher costs and the need for maintenance; as well as the advantages brought by system virtualization [2]. Furthermore, as the sonic character and *feel* of many of these devices is found pleasing by both musicians and audiences alike, preserving these properties for future generations should be worth the effort.

A vast array of techniques have been developed within the last decades to construct digital systems that mimic the behavior of a great variety of analogue gear, including vintage synthesizer filters [3, 4, 5], synthesizer oscillators [6, 7, 8], studio equipment [9, 10], audio effects [11, 12, 13] and guitar amplifiers [14, 15, 16]. These techniques are typically categorized into either *white-box*, *black-box* or *gray-box* methods, depending on the quality of the system knowledge used as the basis for the modeling. White-box approaches use *full knowledge* of the target system, as represented by available circuit schematics and service manuals, in order to first identify and derive the equations governing the system, which are later discretized and simulated on a computer. In contrast, black-box approaches construct digital systems by measuring the *input-output relationships* collected from the target device and by adapting a general-purpose modeling framework to approximate the required behavior. Finally, gray-box methods use a hybrid solution that utilizes both partial system knowledge as well as input-output relationships as the basis for the modeling.

Both ends of these approaches have their respective advantages and disadvantages. Although white-box modeling methods have been able to produce digital systems that accurately and efficiently replicate the desired behavior [17, 18, 5], deriving the governing equations as well as constructing the digital models requires expert knowledge in electrical engineering as well as digital signal processing. Moreover, deriving the equations requires full knowledge of the target system to be available, and the work must be started from scratch when exchanging the target. On the other hand, although black-box methods use general-purpose frameworks to approximate the behavior of a range of devices, their accuracies have often been lower than their white-box counterparts [19, 20, 21]. Moreover, including the user controls of the target system into the digital models has been challenging when using black-box methods, and multiple digital systems are sometimes required for simulating different parametrizations of the target circuit.

Recent advances in machine learning (ML) research, especially within deep learning approaches, have provided VA modeling with a new set of tools to construct digital models. In recent years, state-of-the-art results in approximating the behavior of a variety of devices have been shown by utilizing common deep learning network structures, such as Convolutional Neural Networks (CNNs) [22, 23] and Recurrent Neural Networks (RNNs) [24, 25], that were trained with data collected from the targets. Motivated by these findings, further research has investigated various machine

learning based solutions to the problem, including ML-based state-space techniques [26, 27], embedding of traditional DSP components within deep learning networks [28, 29, 30], and using end-to-end optimization techniques to tune the parameters of explicitly derived white-box models [31].

Although these approaches have proven effective in the VA modeling context, they all require relatively large amounts of data to be available from the target device for the optimization stage in cases where the user controls of the target are to be included in the digital model. While these data can often be collected either manually using only a handful of common equipment or generated from a SPICE simulation assuming that a circuit schematic is available, the size of the dataset can grow considerably when the device has more than one user control. This stems from the need for exposing the networks to enough training data from all the circuit configurations in order to accurately learn the target behavior.

Since earlier research has also demonstrated the ability of the networks to mimic system behavior from outside the configurations that the networks have been exposed to [24], the question arises concerning which aspects of the dataset would be important for learning a desired behavior. Although it is common practice to report the used dataset sizes for learning the behavior of a specific device, no studies have explicitly investigated the effects of varying the dataset properties in learning tasks where the target systems have multiple user controls. Moreover, even though existing literature has provided methods for inclusion of the user controls into the neural models, the coverage of this aspect of the modeling challenge was found shallow in a recent review [32].

This thesis evaluates the effects of varying the dataset properties in learning scenarios where the target system has multiple user controls. To accomplish this aim, multiple target devices with varying complexities and uses are modeled using a proven recurrent neural network architecture consisting of a Gated Recurrent Unit (GRU) and a fully-connected output layer [24]. The neural networks are constructed using the *PyTorch* deep learning library. The target devices considered consist of a nonlinear analogue filter, a guitar distortion effect, as well as a saturating analogue equalizer (EQ) utilizing vacuum tubes within the circuit. For gathering the training data, SPICE simulations of the target devices are used, with the various circuit configurations and varying input data being handled programmatically through Python.

The rest of this thesis is organized as follows. Chapter 2 provides background knowledge in supervised machine learning problems, formulates the learning task, as well as introduces common building blocks used within larger network structures. Chapter 3 reviews existing methods used for neural modeling of audio circuits. Chapter 4 gives an overview of the modeling targets, as well as introduces the research methods used for answering the original research question. Chapter 5 presents the results from the conducted experiments. Finally, Chapter 6 concludes the work and proposes possible future research avenues.

2 Neural Modeling Tools

In order to understand virtual analog modeling, approached as a black-box machine learning problem, some central and cross-domain machine learning techniques are needed to be covered. This chapter attempts to cover enough ground necessary to understand the applied concepts further down the line, and sets up a common notation scheme for the chapters to follow. Chapter 2.1 describes the machine learning task known as sequence modeling, and defines its more specialized form encountered in black-box VA modeling. The same chapter also gives a brief introduction to neural network training using common techniques such as stochastic gradient descent. The remaining chapters review some common machine learning models that can be used as building blocks within bigger neural network architectures. This review covers the general-purpose Multilayer Perceptrons (Chapter 2.2), Convolutional Neural Networks (Chapter 2.3) and Recurrent Neural Networks (Chapter 2.4), as well as a more recent ML approach entitled Differentiable DSP (Chapter 2.5).

2.1 Machine Learning Primer

Virtual analog modeling, formulated as a black-box machine learning problem, can be understood as a *sequence modeling* task within the category of *supervised learning* problems [33]. In supervised learning problems, the learning algorithm, a.k.a the ML *model*, is given a dataset $\mathbb{D}$ of N input/outputs pairs $(\mathbf{X}^{(i)}, \mathbf{Y}^{(i)})$, $\mathbb{D} = \{(\mathbf{X}^{(i)}, \mathbf{Y}^{(i)})\}_{i=1}^{N}$, and the model *weights* $\boldsymbol{\theta}$ are tuned to associate these pairs together. In sequence modeling tasks, the inputs and the outputs $(\mathbf{X}, \mathbf{Y})$ are sequences of possibly differing dimensions, and the output of the system at index τ is a function of the current element of the input sequence, as well as the previous output element:

$$(\mathbf{X}, \mathbf{Y}) = ([\mathbf{x}[0], ..., \mathbf{x}[n]], [\mathbf{y}[0], ..., \mathbf{y}[m]]), \tag{1}$$

$$\mathbf{y}[\tau] = f(\mathbf{y}[\tau - 1], \mathbf{x}[\tau]). \tag{2}$$

In black-box virtual analog modeling, the inputs and the outputs $(\mathbf{X}, \mathbf{Y})$ consist of discrete-time observations of time-series audio data $(\mathbf{x}, \mathbf{y})$ going in to and coming out of some modeling target, which defines the mapping f between the quantities. In the machine learning solution to the problem, function f is approximated with a neural network $f_{\boldsymbol{\theta}}$, the weights $\boldsymbol{\theta}$ of which are trained on the dataset $\mathbb{D}$. These correspondences are visualized in Figure 1, as well as formulated in the equations below:

$$(\mathbf{X}, \mathbf{Y}) = (\mathbf{x}, \mathbf{y}) = ([x[0], ..., x[n]], [y[0], ..., y[n]]), \tag{3}$$

$$y[\tau] = f(y[\tau - 1], x[\tau]) \approx f_{\boldsymbol{\theta}}. \tag{4}$$

In supervised learning problems, the model weights $\boldsymbol{\theta}$ are tuned with respect to a training set $\mathbb{D}$ of input/output pairs $\{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^{N}$. In order to do this, the quality of the estimate resulting from Equation (4) is first evaluated according to some chosen criterion $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$, called a *loss* or *cost* function, such as the mean squared error (MSE) loss [33]. To tune $\boldsymbol{\theta}$, the gradient of the loss with respect to the model

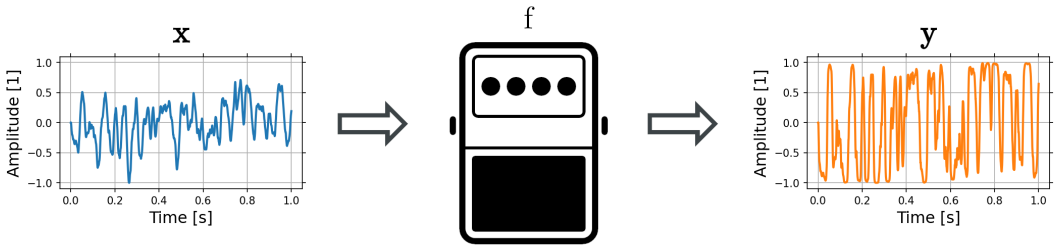


Figure 1: Data in Black-Box VA modeling.

weights $\nabla_{\boldsymbol{\theta}}\mathcal{L}$ is computed, and the weights are then stepped towards the negative gradient using some variant of *gradient descent*. The gradient of the loss can be computed efficiently via the *backpropagation* algorithm, exploiting the chain rule of partial differentiation and the partial derivatives evaluated at different points of the computational graph resulting from computing $f_{\boldsymbol{\theta}}$ [33].

To determine the real gradient, the learning algorithm would need to compute the loss for all training examples $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}) \in \mathbb{D}$ before each optimization step, which is computationally heavy for larger dataset sizes. In practice, *stochastic gradient descent (SGD)* is used, in which the full training set $\mathbb{D}$ of size $|\mathbb{D}| = N$ is divided into smaller portions $\mathbb{B} = \{(\mathbf{x}^{(i')}, \mathbf{y}^{(i')})\}_{i'=1}^{M}$, $M < N$ called *mini-batches*, and the model parameters are updated multiple times using estimates of the gradients computed for each of the mini-batches. One training loop over all of the mini-batches (the full training set) is called a training *epoch*. The number of elements in a mini-batch $|\mathbb{B}| = M$ is called the *batch-size*.

2.2 Multilayer Perceptron

Multilayer perceptrons (MLPs) are general-purpose feedforward models that can be used for approximating mappings $f : \mathbb{R}^n \to \mathbb{R}^m$ [33]. As the name feedforward implies, information is only propagated from the inputs of the network towards its output, without utilizing feedback connections. MLPs are comprised of multiple layers of neurons, each layer computing a nonlinear mapping from a set of inputs $\mathbf{x} \in \mathbb{R}^{n'}$ to a set of outputs $\mathbf{y} \in \mathbb{R}^{m'}$, visualized in Figure 2. The stacking of multiple nonlinear layers together gives multilayer perceptrons their high representational capabilities, up to the point that they are found to act as *universal approximators* for continuous functions under certain criteria [34].

A single layer of an MLP, also known as a *fully-connected* or *dense* layer in ML-literature, is visualized in Figure 3. Each neuron in a fully-connected layer computes:

$$y = \phi(\mathbf{w}^T\mathbf{x} + b), \quad (\mathbf{x}, \mathbf{w}) \in \mathbb{R}^{n'}, (y, b) \in \mathbb{R}, \tag{5}$$

where $\mathbf{w}$ is a vector of weights, b a bias term and $\phi(\cdot)$ a nonlinear activation function, such as hyperbolic tangent. Stacking the weight vectors $\mathbf{w}$ of each neuron in the layer to a matrix $\mathbf{W}$ produces the full layer output:

$$\mathbf{y} = \phi(\mathbf{W}\mathbf{x} + \mathbf{b}), \quad \mathbf{W} \in \mathbb{R}^{m' \times n'}, (\mathbf{y}, \mathbf{b}) \in \mathbb{R}^{m'}. \tag{6}$$

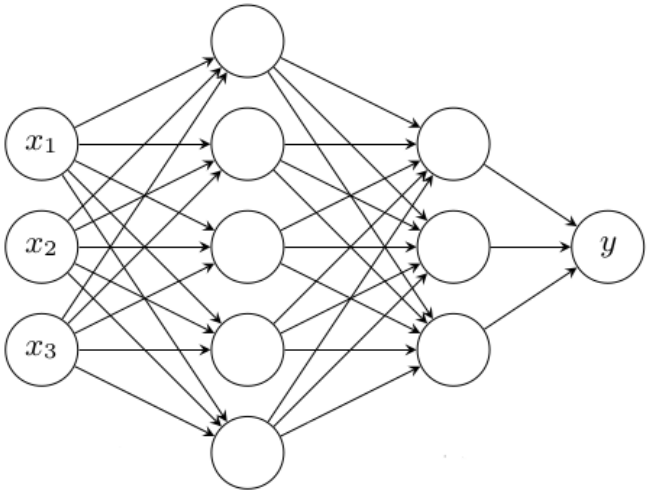


Figure 2: Multilayer Perceptron, adopted from [35].

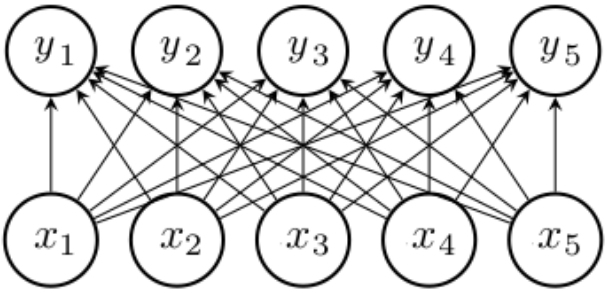


Figure 3: Fully-connected layer, adapted from [35].

Overall, the approximation of the mapping f is computed as:

$$\mathbf{y} = f(\mathbf{x}) \approx f_{\boldsymbol{\theta}}(\mathbf{x}), \quad \mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^m, \tag{7}$$

where $f_{\boldsymbol{\theta}}$ is the learned transformation applied by the network parametrized by $\boldsymbol{\theta}$: the set of all weights w_{ji} and biases b_j of all the neurons within the network.

Multilayer perceptrons do not take into account the ordering of the elements in $\mathbf{x}$, and each individual weight w_{ji} is learned independently. Thus, if a pattern appears at the start of an input $\mathbf{x}$, it is processed with a different set of weights than in the case when it appears at the end. This means, that when data with known relations between its elements, such as time-sequences $\mathbf{x} = [x[n], ..., x[n-N]]^T$ are fed into MLPs, these relations are not directly exploited in the learning task.

The layer that computes the output of an MLP is often called the *output layer*, while the remaining layers are called *hidden layers*. The name hidden stems from the notion that the output of each of these layers is not directly dictated by the learning data and that the model has to choose during training how to leverage these layers

appropriately. The number of stacked layers N, the hidden sizes m'_j, as well as the activation functions $\phi_j(\cdot)$ make up the *hyperparameters* of an MLP, and should be chosen according to the task.

2.3 Convolutional Neural Networks

Convolutional neural networks (CNNs) are specialized feedforward neural networks used for processing data with grid-like topology [33]. They are useful for processing sequential data, such as discrete-time audio, since they take into account the ordering of the elements in $\mathbf{x}$. The name of CNNs stem from the reminiscence of the applied processing to the *convolution* operation. An example of a CNN layer is visualized in Figure 4.

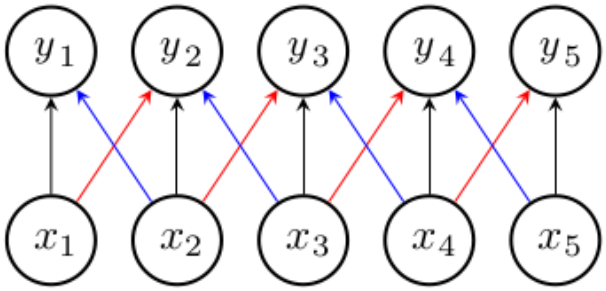


Figure 4: Convolutional neural network layer, adapted from [35]. The colors illustrate parameter sharing for equivalent sets of connections.

Assuming time-series data $\mathbf{x}$ and $\mathbf{y}$, similar to that shown in Equation (3), each neuron in a CNN layer computes [35]:

$$y[n] = \sum_{\Delta n \in \mathbb{Z}} w_{\Delta n} x[n + \Delta n] + b, \tag{8}$$

where $y[n]$ is an element of the output sequence $\mathbf{y}$, $x[n + \Delta n]$ are elements of the input sequence $\mathbf{x}$ and $\Delta n \in \mathbb{Z}$ are a set of time steps near n that are used as context for computing each element of the output sequence. Each neuron in a CNN layer is parametrized by the same set of weights $w_{\Delta n}$, called the (filter) *kernel*, and a bias term b. This is represented as using the same color for equivalent sets of connections in Figure 4, and is often called *parameter sharing* in the context of ML. Parameter sharing also makes is possible to process sequences of varying lengths, since only a portion of the input is used for the computations at any given time step.

Similarly as in the case of MLPs, the CNN layer output is often transformed with a nonlinear activation function $\phi(\cdot)$, which improves the representational capabilities of the network:

$$y'[n] = \phi(y[n]). \tag{9}$$

Overall, the approximation of the output element at discrete time n is given as:

$$y[n] \approx f_{\boldsymbol{\theta}}(\{x[n + \Delta n] \mid \Delta n \in \mathbb{Z}\}), \quad y \in \mathbb{R}, \tag{10}$$

where $\{x[n + \Delta n] \mid \Delta n \in \mathbb{Z})\}$ are elements of the input sequence considered within the layer computations and $f_{\boldsymbol{\theta}}$ is the learned transformation applied by the network parametrized by the set of all weights $\boldsymbol{\theta}$. Note, that due to parameter sharing, the set $\boldsymbol{\theta}$ is much smaller than would be for a similarly sized MLP.

The number of elements in $\{x[n + \Delta n] \mid \Delta n \in \mathbb{Z})\}$ considered by the network when computing each element of the output sequence is called its *receptive field*. In discrete-time regression tasks such as VA modeling, the future input sequence elements $\{x[n + \Delta n] \mid \Delta n > 0\}$ should not be considered by the model and Δn should be constrained to $\Delta n \leq 0$. This is known in ML literature as using *causal convolutions*.

The ordering of the elements in $\mathbf{x}$ is taken into account by a CNN by using *local connections*. Instead of considering every input position $n \in |\mathbf{x}|$ when computing the value of an element at a particular output position m, only a set of positions $n + \Delta n$ close to the output position are considered. Using shared kernel weights $w_{\Delta n}$ among the neurons allows the network to react similarly to stimuli appearing at different locations of the input sequence, known as *translation equivariance* in the context of ML.

CNNs can be though of as applying a sliding window of samples as context for calculating each element of the output sequence, similarly as how finite-impulse response (FIR) filters operate. Unlike most FIR filters, the usage of activation functions at the layer outputs make CNN layers behave nonlinearly.

2.4 Recurrent Neural Networks

Recurrent neural networks (RNNs) are specialized neural networks used for processing sequential data, such as time-series [33]. The name of RNNs stem from the utilization of feedback connections from layer outputs back to layer inputs, which separates them from the family of feedforward NNs. RNNs can be used to process input sequences of varying lengths, and the utilization of feedback makes the networks have potentially infinite memory without needing to use impractically long filter kernels, as would be the case with CNNs. A visualization of an RNN unfolded computational graph is shown in Figure 5.

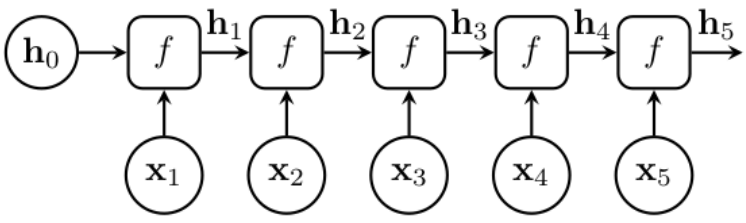


Figure 5: RNN unfolded computational graph, adopted from [35].

Assuming again time-series data, each element of the output sequence $\mathbf{h}[n]$ produced by an RNN layer is a function of the previous element $\mathbf{h}[n-1]$ and the

current input element $\mathbf{x}[n]$:

$$\mathbf{h}[n] = f_{\boldsymbol{\theta}}(\mathbf{h}[n-1], \mathbf{x}[n]), \quad \mathbf{x} \in \mathbb{R}^n, \mathbf{h} \in \mathbb{R}^m, \tag{11}$$

where $\mathbf{h}[n]$ and $\mathbf{h}[n-1]$ are the RNN *states* at current and previous time steps n and $n-1$; $\mathbf{x}[n]$ is an element of the input sequence $\mathbf{x}$ at current time step n and $f_{\boldsymbol{\theta}} : \mathbb{R}^n \to \mathbb{R}^m$ is the learned transformation applied by the RNN. The transformation $f_{\boldsymbol{\theta}}$ is applied at every time step using the same weights $\boldsymbol{\theta}$, which is similar to parameter sharing in the context of CNNs. The exact transformation applied is dependent on the *recurrent unit* used within the RNN, the most common types being *Long Short-Term Memory (LSTM)* unit [36] and *Gated Recurrent Unit (GRU)* [37], the latter of which will be explained below.

The governing equations for a GRU are listed in Equations (12)-(15), and its state update is visualized in Figure 6.

$$\mathbf{u}[n] = \sigma(\mathbf{U}_u\mathbf{x}[n] + \mathbf{b}_{xu} + \mathbf{W}_u\mathbf{h}[n-1] + \mathbf{b}_{hu}), \tag{12}$$

$$\mathbf{r}[n] = \sigma(\mathbf{U}_r\mathbf{x}[n] + \mathbf{b}_{xr} + \mathbf{W}_r\mathbf{h}[n-1] + \mathbf{b}_{hr}), \tag{13}$$

$$\mathbf{h}[n] = (1 - \mathbf{u}[n]) \odot \tilde{\mathbf{h}}[n] + \mathbf{u}[n] \odot \mathbf{h}[n-1], \tag{14}$$

$$\tilde{\mathbf{h}}[n] = \phi(\mathbf{U}_h\mathbf{x}[n] + \mathbf{b}_{xh} + \mathbf{r}[n] \odot (\mathbf{W}_h\mathbf{h}[n-1] + \mathbf{b}_{hh})). \tag{15}$$

Computing the GRU state update involves using the current input elements $\mathbf{x}[n]$ and previous state $\mathbf{h}[n-1]$ using a *new state candidate* $\tilde{\mathbf{h}}[n]$ as well as two gates: the *update gate* $\mathbf{u}[n]$ and the *reset gate* $\mathbf{r}[n]$:

- Both gates $\mathbf{u}[n]$ and $\mathbf{r}[n]$ are computed as combinations of the input elements $\mathbf{x}[n]$ and previous state $\mathbf{h}[n-1]$, using projection matrices $\{\mathbf{U}_u, \mathbf{W}_u\}$ or $\{\mathbf{U}_r, \mathbf{W}_r\}$ as well as bias terms $\{\mathbf{b}_{xu}, \mathbf{b}_{hu}\}$ or $\{\mathbf{b}_{xr}, \mathbf{b}_{hr}\}$. $\sigma(\cdot)$ denotes the sigmoid function.
- The new state $\mathbf{h}[n]$ is computed as a combination of the old-state $\mathbf{h}[n-1]$ and the new state candidate $\tilde{\mathbf{h}}[n]$, controlled by the update gate $\mathbf{u}[n]$. $\odot$ denotes the Hadamard (element-wise) product.
- Similarly as in the case of both gates, the new state candidate $\tilde{\mathbf{h}}[n]$ is computed as a combination of the input $\mathbf{x}[n]$ and previous state $\mathbf{h}[n-1]$ via matrices $\{\mathbf{U}_h, \mathbf{W}_h\}$, and biases $\{\mathbf{b}_{xh}, \mathbf{b}_{hh}\}$, on top of which the previous state is weighted by a reset gate $\mathbf{r}[n]$. $\phi(\cdot)$ is a nonlinear activation function, such as the hyperbolic tangent.

Similarly as in the case of CNNs, RNNs use shared parameters to allow for location-aware processing. The parameter sharing is a result of using the same update rule f to compute the output at every time step in a recursive fashion. Due to the nature of this processing, the input sequences can be of arbitrary length and there's no practical limit in the receptive field of the model, as with CNNs.

Noting the recursive state update performed by the recurrent unit, the RNN computations bear resemblance to the behavior of an infinite impulse response (IIR)

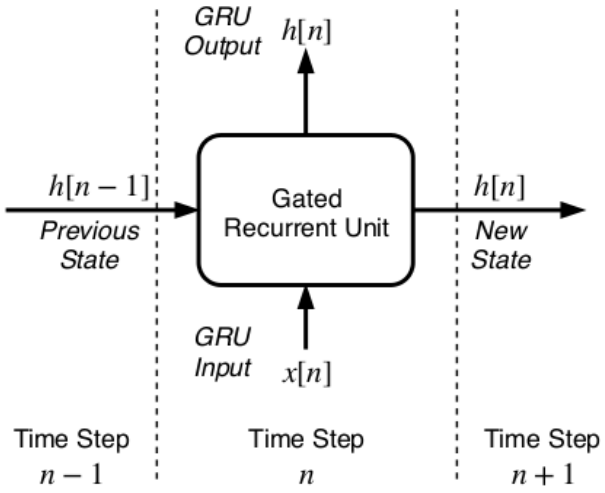


Figure 6: Gated Recurrent Unit (GRU), adopted from [24].

filter. The activations used withing the recurrent units state update makes the system behave nonlinearly, unlike most of its IIR counterparts.

Even though an RNN state contains information of the whole past sequence, learning long-term dependencies from the input data has proven challenging with RNNs [33]. The problem stems from the gradients having have to traverse through a very deep computational graph during backpropagation, making them either vanish or explode. The problem can be illustrated by considering a simplified recurrent update $\mathbf{h}[n] = \mathbf{W}\mathbf{h}[n-1]$ with a diagonalizable $\mathbf{W}$, which after τ steps takes the form:

$$\mathbf{h}[n] = \sum_{\tau=1}^{n} \mathbf{W}^{n-\tau}\mathbf{h}[0], \tag{16}$$

$$= \sum_{\tau=1}^{n} \mathbf{Q}\mathbf{\Lambda}^{n-\tau}\mathbf{Q}^{-1}\mathbf{h}[0]. \tag{17}$$

Depending on the eigenvalues λ_i in $\mathbf{\Lambda}$, some terms in $\mathbf{h}$ will either decay to zero or grow without bounds as $n \to \infty$ as a result of the summation.

2.5 Differentiable DSP

Differentiable digital signal processing (DDSP) is a method for embedding classical digital signal processing components into trainable deep learning models [28]. While general-purpose deep learning methods, such as RNN and CNNs, can be optimized to suit a wide variety of tasks, the generality comes with possibly higher computational costs and longer training times. The intuition in embedding traditional DSP components into the networks is to use structural knowledge of the type of processing happening in the modeled system, and choosing the DSP components accordingly. This way, the optimization routines used in machine learning can be combined with

the intuitiveness of transparent DSP blocks. The idea is visualized in Figure 7 for a neural network consisting of a filter cascade, the parameters of which are updated according to a loss computed during the forward pass.

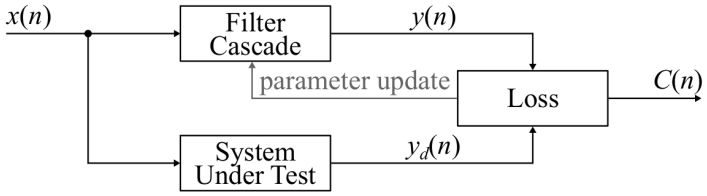


Figure 7: The intuition in Differentiable DSP, adopted from [38].

The computations happening within a DDSP-layer depend on the chosen processors and their interaction. The original DDSP library is comprised of various computational blocks, such as *FIR filters*, *oscillators* and *waveshapers*, the parameters $\boldsymbol{\theta}$ of which can be trained via backpropagation. Later work has followed the intuition in DDSP to allow for training the parameters of *IIR filters* as well [29], although these contributions were not made part of the DDSP-library.

As an example, assume the modeled system f is known to be in the family of first-order IIR filters. We can approximate its output at discrete time n recursively as:

$$y[n] = f(y[n-1], x[n], x[n-1]), \tag{18}$$

$$\approx f_{\boldsymbol{\theta}}(\hat{y}[n-1], x[n], x[n-1]) = \hat{b}_0 x[n] + \hat{b}_1 x[n-1] - \hat{a}_1 \hat{y}[n-1], \tag{19}$$

where $\hat{y}[n-1]$ is the approximation of the output at the previous time step and $\hat{b}_0, \hat{b}_1$ and $\hat{a}_1$ are the estimated (learned) values for the filter coefficients, making up the layer weights $\boldsymbol{\theta} = \{\hat{b}_0, \hat{b}_1, \hat{a}_1\}$.

3 Neural Modeling of Audio Circuits

In this chapter, past solutions to VA modeling using neural networks are covered. The focus is on methods that are both causal, i.e. only using current and past audio samples as context for the task, as well as capable of running in real-time. The solutions are divided into convolutional VA models (Chapter 3.1), recurrent VA models (Chapter 3.2), state-space VA models (Chapter 3.3) and Differentiable DSP for VA modeling (Chapter 3.4). Even though good results have also been achieved from autoencoder-based structures [39, 40, 41], these models are either not causal or currently incapable of running in real-time and thus left out of scope for this work. A very recent direction of research [42, 43], exploiting the fact that a lot of the studied circuits are governed by ordinary differential equations (ODEs) that can be learned directly from data, is also out of the scope.

3.1 Convolutional VA Models

Neural networks utilizing convolutional layers within their architectures as the main processing unit have been used successfully for the task of VA modeling. In [23], [44] and [16], a feedforward variant of the WaveNet-architecture [22] was used for modeling guitar distortion pedals and vacuum tube amplifiers. Later work has concentrated on extending the WaveNet receptive field to allow for modeling nonlinear circuits with longer-term memory, such as analogue dynamic range compressors [45, 46].

The feedforward WaveNet network architecture is shown in Figure 8. The network is comprised of a series of convolutional layers (*'CONV'*), each connected to a global post-processing layer generating the network output. The convolutional layers are conditioned with the device parameters, affecting their internal processing.

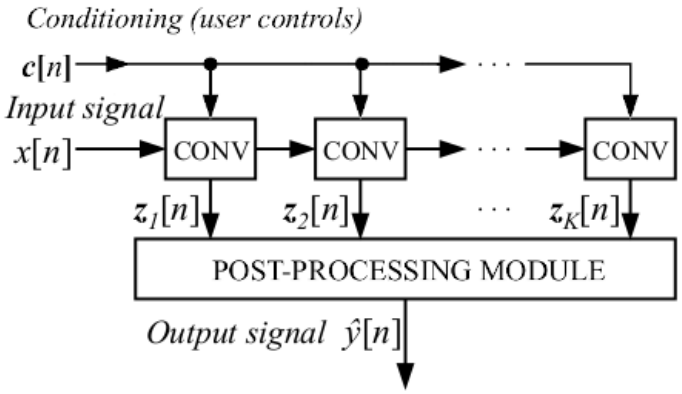


Figure 8: Feedforward WaveNet architecture, adapted from [44] & [23].

The k'th convolutional layer is shown in Figure 9. Its governing equations can be written out as:

$$\mathbf{z}_k[n] = \phi((\mathbf{H}_k * \mathbf{x}_k)[n] + \mathbf{b}_k), \tag{20}$$

$$\mathbf{x}_{k+1}[n] = \mathbf{W}_k \mathbf{z}_k[n] + \mathbf{x}_k[n], \tag{21}$$

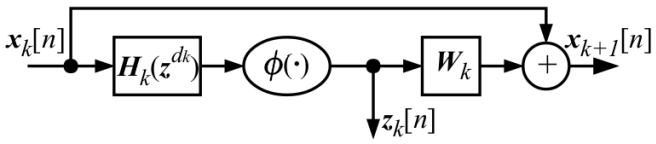


Figure 9: Convolutional layer, adapted from [44].

where $\mathbf{z}_k[n]$ is the layer output, $\mathbf{x}_k[n]$ the layer input, and $\mathbf{x}_{k+1}[n]$ the input to the next layer. $*$ denotes the convolution operation. The layer output is computed by filtering the input sequence $\mathbf{x}_k[n]$ with a convolutional kernel $\mathbf{H}_k$, adding a vector of biases $\mathbf{b}_k$ and applying a nonlinearity $\phi(\cdot)$ to the result. The input to the next layer is computed by taking a sum of the current layer input and a portion of the layer output as defined by a mixing matrix $\mathbf{W}_k$. In the context of ML, this type of direct connection from a layer input to its output is called a *skip connection*.

Two variants for the post-processing block have been given in [23] and [44, 16]. In the original work [23], the post-processing block comprised of a series of two 1×1 convolutional layers each driving a nonlinearity, as well as an MLP. The first convolutional layer used a gated activation function, and the second a hyperbolic tangent. In the following works [44, 16], the post-processing block was simplified to only include a single 1×1 convolutional layer without a nonlinearity. In the context of ML, a 1×1 convolution signifies linearly mixing the inputs to the outputs according to some learned mixing factors, which is equivalent to applying the same linear layer at every time step.

In the WaveNet architecture, the model receptive field is extended by using *stacked dilated convolutions*. Dilation is a technique used to extend the effective filter size of the convolutional kernels by skipping elements of the input at certain time steps. In the WaveNet architecture, multiple dilated convolutional layers are connected in series and the dilation factors of the layers are increased as a function of the layer index, shown in Figure 10. This allows the model to achieve large receptive fields while maintaining a fine small-scale resolution simultaneously in a computationally efficient manner. The WaveNet receptive field can be computed as:

$$N = (M-1)\sum_{k=1}^{K} d_k + 1, \tag{22}$$

where K is the number of convolutional layers, M is the filter size, and d_k is the dilation factor of the k'th layer.

Two types of conditioning methods for the network were introduced in the original research paper introducing the WaveNet-architecture [22]: local and global conditioning. While global conditioning can be used for adapting the model performance across all time steps (e.g. speaker identity), local conditioning allows for adjusting the model behavior in a time-varying manner (e.g. word-level features). Noting that the parameters of audio processing equipment are often adjusted online, the latter

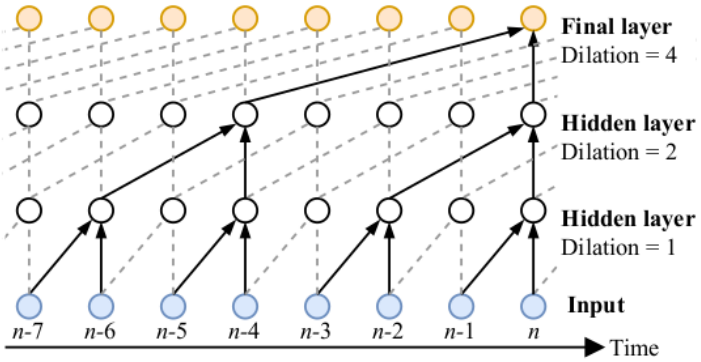


Figure 10: WaveNet receptive field / dilation, adopted from [44].

type of conditioning was used to control the predictions of the feedforward WaveNet [23].

The WaveNet local conditioning is applied as follows:

$$\mathbf{z} = \tanh(\mathbf{W}_f * \mathbf{x} + \mathbf{V}_f * \boldsymbol{\phi}) \odot \sigma(\mathbf{W}_g * \mathbf{x} + \mathbf{V}_g * \boldsymbol{\phi}). \tag{23}$$

When local conditioning is applied, the convolutional layer output computations (Equation (20)) are modified to accept a second sequence of inputs $\boldsymbol{\phi}$, comprising of the conditioning values. Both the sequence of inputs $\mathbf{x}$ and the conditionings $\boldsymbol{\phi}$ are filtered with two convolutional kernels $\{\mathbf{W}_f, \mathbf{W}_g\}$ or $\{\mathbf{V}_f, \mathbf{V}_g\}$. After the filtering, two sequences are formed by summing the filtered sequences together, which are then used to drive two nonlinear activation functions, a hyperbolic tangent $\tanh(\cdot)$ and a sigmoid function $\sigma(\cdot)$. The layer output $\mathbf{z}$ is formed by combining the transformations via the Hadamard-product $\odot$.

Overall, the model prediction at each time step $\hat{y}[n]$ is computed as:

$$\hat{y}[n] = f_{\boldsymbol{\theta}}([x[n], ..., x[n-N]]^T, \boldsymbol{\phi}[n]), \tag{24}$$

where $[x[n], ..., x[n-N]]^T$ are a series of $N+1$ current and past input samples, $\boldsymbol{\phi}[n]$ is a vector of conditioning values at the current time step and $f_{\boldsymbol{\theta}}$ is the learned transformation applied by the network parametrized by weights $\boldsymbol{\theta}$.

3.2 Recurrent VA Models

Recurrent neural networks have been used successfully for VA modeling, utilizing both LSTMs and GRUs as the recurrent unit within the architecture. The first reported experiments using RNN-based VA models utilize LSTMs to model the behavior of vacuum tube amplifiers, without incorporating the user controls of the targets into the digital systems [47, 48]. In [24], both LSTM and GRU-based networks for modeling a guitar distortion pedal as well as a vacuum tube amplifier were evaluated, this time with the inclusion of user controls. The work in [25] uses the GRU-based network from [24] for modeling a guitar distortion effect and focuses on tuning

the network hyperparameters to achieve real-time performance on an embedded system. In [49] and [13], an LSTM-based RNN is used to model the behavior of an analogue phaser and a flanger, requiring the network to pay attention to longer-term time-dependencies than in the case of saturation or distortion effects.

A typical RNN-based architecture for VA modeling is shown in Figure 11. The network consists of a recurrent unit as well as an output layer for computing the output prediction $\hat{y}[n]$ at each time step. The recurrent unit receives as an input the current sampled value of the input waveform $x[n]$, as well as its previous state(s) $\mathbf{s}[n-1]$, and updates its states(s) for the current time step $\mathbf{s}[n]$. In the case of an LSTM, the states consists of both a hidden state, as well as a cell state $\mathbf{s} = \{\mathbf{h}, \mathbf{c}\}$, while a GRU only uses one state, the hidden state $\mathbf{s} = \mathbf{h}$. In both cases, the current hidden state $\mathbf{h}[n]$ is passed on to the output layer. For an overview of the state update computations for a GRU, refer to Chapter 2.4. An overview of the state update for an LSTM can be found, for example, from [24].

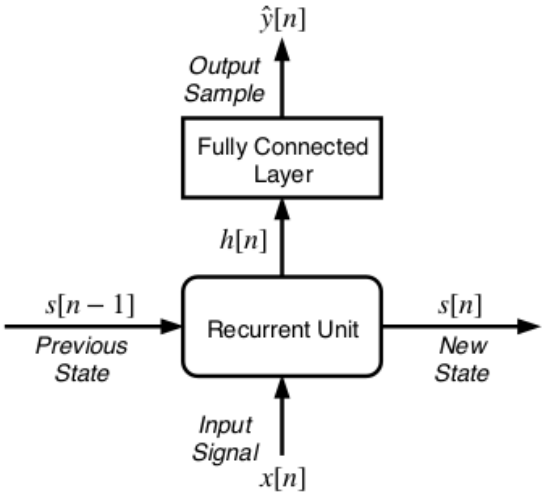


Figure 11: Typical RNN architecture, adopted from [24].

For computing the output prediction for the current time step $\hat{y}[n]$, the hidden state $\mathbf{h}[n]$ is passed to a single fully-connected layer, which maps it to the desired output dimensionality. The governing equations for a fully-connected layer can be found from Chapter 2.2.

To allow for the network to take into account the user controls of the target system, the input can be extended to a vector storing both the current sampled input $x[n]$, as well as the current user controls $\boldsymbol{\phi}[n]$, demonstrated in Equation (25). To allow for this change in input dimensionality, the projection matrices $\mathbf{U}_i$ in Equations (12)-(15) are required to be updated to shape $\mathbf{U}_i \in \mathbb{R}^{m \times n}$, for keeping the mappings $\mathbf{x} \in \mathbb{R}^n \rightarrow \mathbf{u}, \mathbf{r}, \hat{\mathbf{h}} \in \mathbb{R}^m$ valid.

$$\mathbf{x}[n] = [x[n], \boldsymbol{\phi}[n]]^T \tag{25}$$

Overall, the model predicts the current output sample based on the current vector

of inputs $\mathbf{x}[n]$ as well as the recurrent unit state(s) at the previous time step $\mathbf{s}[n-1]$:

$$\hat{y}[n] = f_{\boldsymbol{\theta}}(\mathbf{x}[n], \mathbf{s}[n-1]), \tag{26}$$

where $\mathbf{x}[n]$ stores both the current input sample and conditioning values as per Equation (25); $\mathbf{s}[n-1] = \mathbf{h}[n-1]$ for a GRU and $\mathbf{s}[n-1] = \{\mathbf{h}[n-1], \mathbf{c}[n-1]\}$ for an LSTM; and $f_{\boldsymbol{\theta}}$ is the learned transformation applied by the network parametrized by weights $\boldsymbol{\theta}$.

3.3 State-Space VA Models

All electronic devices with energy storing elements like capacitors and inductors can be seen as state-space systems whose behavior through time can be understood and predicted by observing their internal states, as well as the inputs at each instance of time [50]. Having access to the internal states of a given target system allows these quantities to be exploited in the modeling task, making these approaches fall into the category of gray-box modeling when used together with the corresponding input-output relationships.

In the context of neural VA modeling, two different approaches in exploiting the state-space of the target system have been demonstrated. In [26], the nonlinear mappings within the target system state-space is modeled with an embedded MLP, a method the authors call the *State Trajectory Network (STN)*. In [27], the modeling of a stateful target is done with a GRU, similar to Chapter 3.2, while conditioning the model to tie its hidden states with the physical states of the target. The latter approach was entitled by the authors as the *State-Space GRU (SSGRU)*. Both of these approaches were used for modeling nonlinear circuits with short-term memory, such as distortion effects and nonlinear filters, while the latter approach was later used to model an analogue phaser [51]. In the following subchapters, both of these methods are given an overview.

3.3.1 State-Trajectory Network

The formulation of the STN solution goes as according to [26]. The ordinary differential equations (ODEs) describing a discrete-time state-space system can be written as:

$$\mathbf{x}[n+1] = f(\mathbf{u}[n], \mathbf{x}[n]), \tag{27}$$

$$\mathbf{y}[n] = g(\mathbf{u}[n], \mathbf{x}[n]), \tag{28}$$

where $\mathbf{x}$, $\mathbf{u}$ and $\mathbf{y}$ are the system states, inputs and outputs, respectively. The system state update is comprised of two (nonlinear) functions f and g, of which the former is used to compute the system states for the next time step $\mathbf{x}[n+1]$ and the latter for computing the system outputs $\mathbf{y}[n]$ at the current time step.

Concatenating the system states with the system inputs allows for combining functions f and g to a single (nonlinear) mapping f_d:

$$\begin{bmatrix} \mathbf{x}[n+1] \\ \mathbf{y}[n] \end{bmatrix} = f_d \left(\begin{bmatrix} \mathbf{u}[n] \\ \mathbf{x}[n] \end{bmatrix} \right), \tag{29}$$

where f_d maps from the space of concatenated states and inputs to the space of the concatenated states and outputs. In the STN solution, this mapping was approximated with an MLP, motivated by their universal approximation capabilities. It should be noted, that since function f_d is a memoryless mapping, i.e. it only considers the current inputs and states for computing its output, there's no need to use networks with embedded memory, like RNNs.

A visualization of the STN model architecture is shown in Figure 12. The model comprises of an MLP with k layers denoted as l_i, where $i \in [1, k]$. The input to the MLP is the concatenation of the system inputs and states $[\mathbf{u}[n], \mathbf{x}[n]]^T$, producing as an output the current output prediction $\mathbf{y}[n]$, as well as the states for the next time step $\mathbf{x}[n+1]$. The network is trained to learn the mapping from the current states to the next ones using a skip connection, denoted with a summing node $\oplus$. The additional scaling term h_τ is used to alter the evolution of the states through time, effectively allowing the system sampling frequency to be modified during inference.

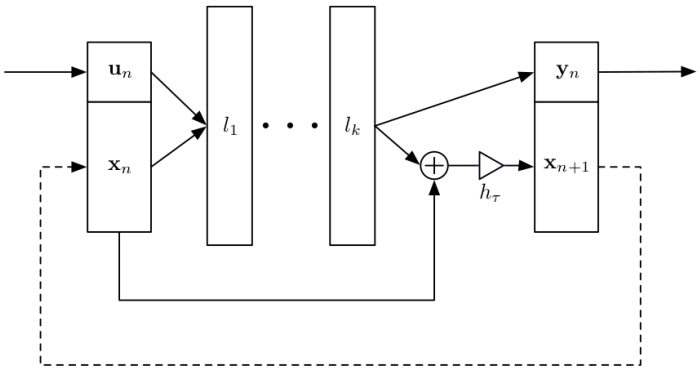


Figure 12: State Trajectory Network (STN), adopted from [26].

The inclusion of user controls into the STN model is not addressed in the original paper and is only mentioned as a possible future research contribution.

3.3.2 State-Space GRU

The SSGRU solution provides an alternative for utilizing the target system states in the modeling task. It can be seen as a hybrid between using an RNN in the conventional black-box manner as in Chapter 3.2, and the more explicit state-space formulation in the previous subchapter.

A GRU is a stateful system whose state update is explained in detail in Chapter 2.4. The state update is comprised of taking the previous hidden state $\mathbf{h}[n-1]$ as well as the current inputs $\mathbf{x}[n]$, to compute the hidden state for the current time step $\mathbf{h}[n]$. By making the inputs store both the current input audio $u[n]$, as well as the current states of the target device $\mathbf{x}_t[n]$, $\mathbf{x}[n] = [u[n], \mathbf{x}_t[n]]^T$, and training the network to match its hidden state with the system state, we can model the system state-space with a GRU. This idea is visualized in Figure 13, showing the model architecture of a SSGRU as seen during training.

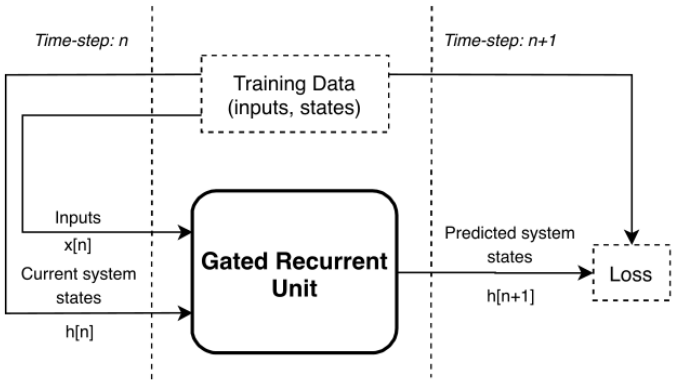


Figure 13: State-Space GRU, adopted from [27].

To produce the system output, the observed states were chosen in [27] such that one of the states corresponds to the system outputs:

$$\mathbf{h}[n] = [h_1[n], \dots, h_{N-1}[n], y[n]]^T, \quad \mathbf{h}[n] \in \mathbb{R}^N, \tag{30}$$

where $y[n]$ is the system output and N is the number of states observed. It should be noted, that while the system output might not correspond directly to an energy storing element, this selection is still valid since the choice of the states is not stiff, as noted in [26]. This is due to the fact, that while the real states of the system are the energy storing elements, all the other elements are linked to those states with a mapping defined by the circuit configuration, and thus carry related information.

Since the SSGRU solution utilizes a standard GRU and the only difference is in altering the inputs and the training procedure to include the system states, the SSGRU solution can be made to account for user controls similarly to as in the standard GRU solution, covered in Chapter 3.2.

3.4 Differentiable DSP for VA Modeling

The differentiable DSP paradigm has been applied successfully for VA modeling in [29, 30, 31]. In [29], a differentiable Wiener-Hammerstein model for nonlinear circuit modeling was constructed via implementing the filters and the static nonlinearity within the system with differentiable IIR filters and an MLP, respectively. A similar approach was taken in [30], where a modeling framework consisting of multiple cascaded blocks of differentiable IIR filters driving static nonlinearities was proposed. Both of these methods were utilized to model guitar distortion effects. In [31], the intuition of trainable DSP blocks was used to build fully-fledged white-box circuit models by hand, and optimizing the system parameters with end-to-end training. This latter approach was used to model a static nonlinear diode clipper, a linear parametric guitar tone-stack, as well as a guitar overdrive stage. The following two subchapters gives an overview of these approaches.

3.4.1 Differentiable Block-Oriented VA Models

Block-oriented VA models are black-box modeling methods using series connections of linear filters and static nonlinearities as general-purpose modeling topologies for a variety of nonlinear circuits with short-term memory, such as guitar distortion pedals. Different topologies have been proposed and analyzed in the past, the differences being in the ordering and the number of elements in the system. An example of such systems is the Wiener-Hammerstein model, consisting of two linear time-invariant (LTI) filters $\mathbf{H}_i$ and a static nonlinearity $g(\cdot)$, shown in Figure 14.

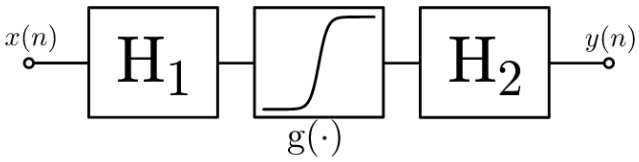


Figure 14: Wiener-Hammerstein model, adopted from [52].

In [29], the LTI filters in a Wiener-Hammerstein model were implemented with various differentiable second-order IIR structures, and the nonlinearity $g(\cdot)$ was approximated with an MLP. The paper evaluated several differentiable IIR implementations for the filtering task, namely a differentiable state-variable (SVF) filter, a differentiable linear state-space (LSS) filter as well as a differentiable transposed direct form-II (TDF-II) filter, with the authors reporting slight differences in their performance.

In [30], the differentiable block-oriented method was extended to include user controls, and multiple processing blocks each resembling a Wiener-model was used for the modeling. A Wiener-model, consisting of a single linear filter as well as a static nonlinearity, is shown in Figure 15, and the full model architecture from [30] is shown in Figure 16.

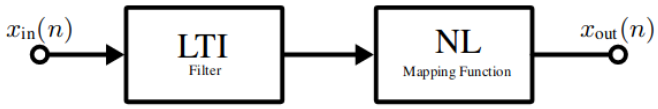


Figure 15: Wiener model, adopted from [21].

The model consists of multiple processing stages $s \in S$, each consisting of a cascade of biquads CB_s, a gain term α_s, and a static nonlinearity NL, except for the final stage s_{S-1}, which doesn't include the nonlinearity. Before the first stage, a delay stage implemented as a differentiable linear fractional delay filter [53] is used to align the output prediction with the ground-truth.

Each of the filters CB_s, as well as the gain terms α_s are controlled using a hyperconditioning block HC_s, that takes as an input a vector of user controls $\mathbf{c}_s$ and maps them to the desired internal parameters of the system, such as the filter

coefficients. The hyperconditioning blocks are implemented as convolutional layers each consisting of a single 1×1 convolution.

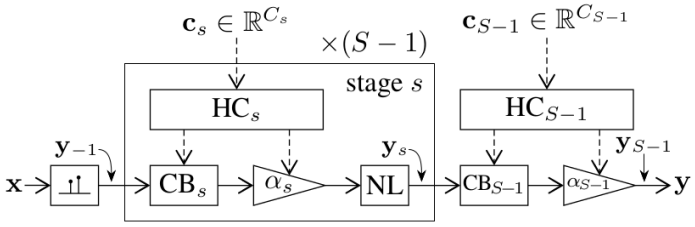


Figure 16: Differentiable extended Wiener-model, adopted from [30].

3.4.2 Differentiable White-Box Modeling

The formulation of the differentiable white-box modeling solution follows similar trajectories as the STN solution in Chapter 3.3.1. The ODEs describing a continuous-time state-space of an electronic circuit can be formulated with the physical system parameters written out as:

$$\dot{\mathbf{x}}(t) = f_{\boldsymbol{\theta}}(\mathbf{x}(t), \mathbf{u}(t)), \tag{31}$$
$$\mathbf{y}(t) = g_{\boldsymbol{\theta}}(\mathbf{x}(t), \mathbf{u}(t)), \tag{32}$$

where $\mathbf{x}$, $\mathbf{u}$ and $\mathbf{y}$ are the system states, inputs and outputs, respectively; $\dot{\mathbf{x}}$ denotes time-derivate of $\mathbf{x}$; and $\boldsymbol{\theta}$ are the internal system parameters such as component values. In white-box modeling, these ODEs are discovered via any standard circuit analysis method, such as nodal analysis [54, Chapter 4].

After discovering and writing out the ODEs, they can then be discretized using some numerical differentiation scheme as:

$$\dot{\mathbf{x}}[n+1] = f'_{\boldsymbol{\theta}}([\mathbf{x}[n+1], ..., \mathbf{x}[n-k]]^T, [\mathbf{u}[n], ..., \mathbf{u}[n-k]]^T), \tag{33}$$
$$\mathbf{y}[n] = g'_{\boldsymbol{\theta}}([\mathbf{x}[n], ..., \mathbf{x}[n-k]]^T, [\mathbf{u}[n], ..., \mathbf{u}[n-k]]^T), \tag{34}$$

where functions $f'_{\boldsymbol{\theta}}$ and $g'_{\boldsymbol{\theta}}$, as well as the the number of past samples k used, is determined by the used discretization scheme. Some common discretization methods used in VA modeling include the trapezoidal rule and forward/backward Euler [55].

In the differentiable white-box modeling solution, the difference equations (33) & (34) are implemented as the forward pass of a processing block using an auto-differentiation library like PyTorch [56], and the system parameters $\boldsymbol{\theta}$ are optimized during the backwards pass according to some loss criterion. The end result is an interpretable white-box model of the circuit, with the parameters of the circuit optimized with respect to the training data.

The user controls $\boldsymbol{\phi}$ of the target can be included into the differentiable white-box model by noting that the system parameters $\boldsymbol{\theta}$ are comprised of both static parameters

$\boldsymbol{\theta}_s$ as well as user controllable parameters $\boldsymbol{\theta}_v$, forming the set $\boldsymbol{\theta} = \{\boldsymbol{\theta}_s, \boldsymbol{\theta}_v\}$. The mapping from the user controls $\boldsymbol{\phi}$ into the variable parameters can be formulated as:

$$\boldsymbol{\theta}_v = f_v(\boldsymbol{\phi}), \tag{35}$$

where f_v defines the possibly nonlinear relationship between the quantities. When the system parameters are being optimized, the influence of the user controls can be taken into account by implementing f_v as a trainable function, such as an MLP.

4 Learning Parameter Spaces

The aim of this thesis is to evaluate how the properties of the training dataset affect generalization of the learning algorithm in learning scenarios where the modeling target has multiple user controls. Having covered the essentials in applied machine learning, as well as some proven techniques in neural VA modeling, this chapter focuses on the research methods applied to answer the research question. To cover this aim, three analogue sound processors with varying complexities are modeled using techniques introduced in the previous chapter, focusing on the role the training dataset has on the generalization of the networks.

The review of the research methods is divided into three chapters. In Chapter 4.1, the devices used as the modeling targets are given a brief technical overview. Chapter 4.2 reviews the collection of the training data, as well as proposes two approaches for sampling the parameter spaces of the modeling targets. Chapter 4.3 introduces the neural network architecture and the loss function used for the modeling, as well as reviews the training procedure.

4.1 Devices

The modeling targets include three nonlinear devices - the nonlinear filtering stage found in the Korg MS10 and early Korg MS20 analogue synthesizers, a guitar distortion pedal (ProCo RAT), as well as a nonlinear tube-based analogue equalizer (Pultec EQP-1). While all of these effects are mainly used for sculpting the timbral qualities of the input audio [57, Chapter 1], the choice of target devices was motivated by their varying number of user controls, which was thought to affect the modeling challenge in a meaningful way. The following subchapter covers each of these devices, in an order dictated by the number of user controls in the original device.

4.1.1 MS10 Filter

The Korg MS-series is a popular line of synthesizers with a rich heritage from the late 1970s, with some models being produced still at the time of writing this Thesis [58]. The nonlinear filter stage found in the Korg MS10 and early Korg MS20 analogue synthesizers, hereby referred to as the MS10 Filter, is fancied by musicians of many disciplines, presumably due to its characteristic nonlinear behavior. In this subchapter, the key characteristics of the original circuit, the circuit topology and schematic of which are shown in Figures 17 & 18, are reviewed, and an overview of the available user controls is given.

A detailed circuit analysis of the MS10 Filter, as well as its later variant, is presented in [59]. The circuit consists of an ingenious implementation of a second-order Sallen-Key low-pass filter, the topology of which is shown in Figure 17, consisting of two filter poles constructed with resistor-capacitor pairs, as well as two gain blocks denoted *k1* and *k2*. The correspondence of these elements to the circuit schematic is highlighted in Figure 18 with orange boxes encapsulating the relevant components. The filter poles of the real circuit are constructed using a pair of bipolar junction

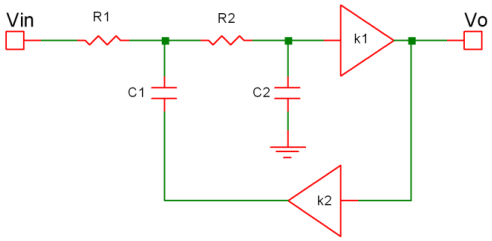


Figure 17: MS10 Filter Sallen-Key structure, adapted from [59].

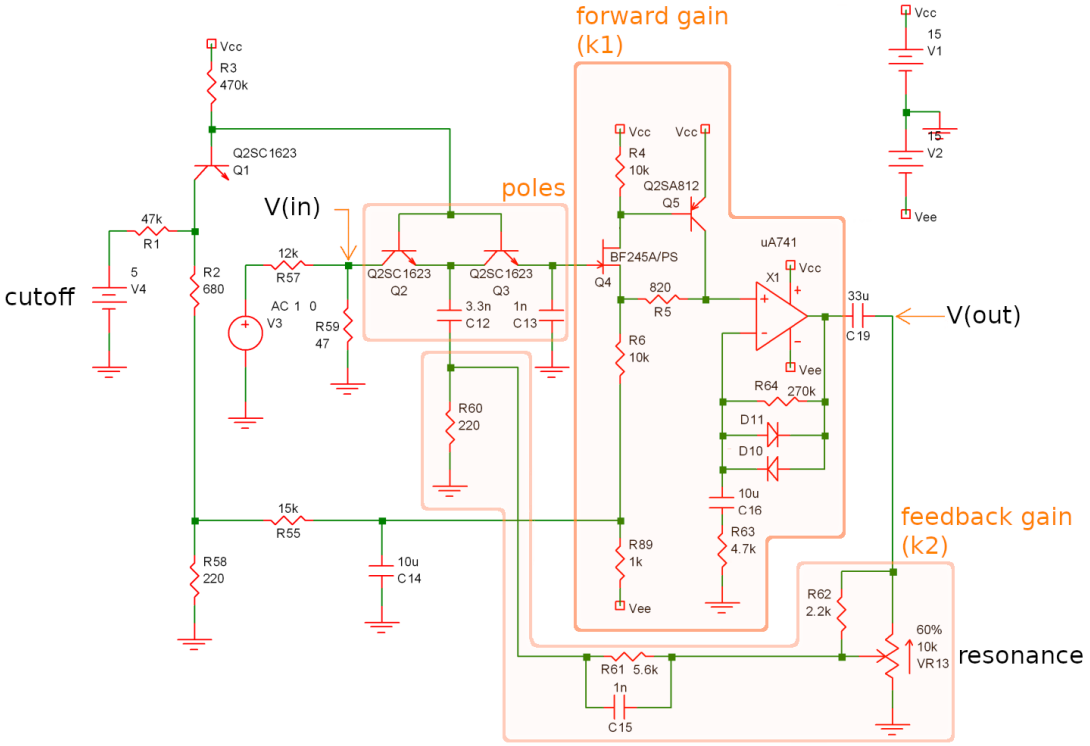


Figure 18: MS10 Filter, adapted from [59].

transistors (BJTs) working as current controlled resistors driving two individual filtering capacitors. The forward gain block *k1* is made up of both a hybrid junction field effect transistor (JFET)/BJT buffer and a nonlinear operational amplifier (op amp) amplification stage, while the feedback gain block *k2* is implemented with a series connection of two voltage dividers, the first of which is made variable with a potentiometer. The nonlinear behavior of the circuit is produced for the most part by the soft-clipping diodes in the feedback loop of the op amp, while the BJTs used to form the filter poles, as well as the hybrid buffering stage, produce lower order harmonic distortion.

The user controls $\boldsymbol{\phi}_{MS10}$ of the circuit include two continuously variable parameters; the cutoff frequency of the filter ϕ_{cutoff}, as well as the filter resonance ϕ_Q; making the vector of user controls $\boldsymbol{\phi}_{MS10} = [\phi_{cutoff}, \phi_Q]$ of size $N_\phi = |\boldsymbol{\phi}_{MS10}| = 2$.

While in the original circuit the filter resonance is controlled with a potentiometer (*VR13* in Fig. 18), the cutoff frequency is set via a control voltage (*V4*), allowing for its automated adjustment.

To get an idea of the nonlinear behavior of the MS10 Filter, Figure 19 shows the circuit response (orange line) to a logarithmic sine sweep (blue line) with the user controls set to $\boldsymbol{\phi}_{MS10} = [0.5, 0.8]$. In the visualization, the additional peaking near the cutoff frequency and the transition band are produced by the nonlinear break-up and saturation of the filter core.

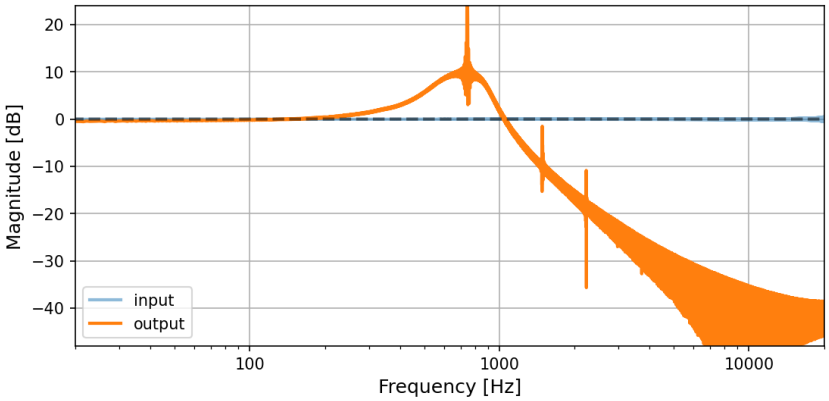


Figure 19: MS10 Filter example response, $\boldsymbol{\phi}_{MS10} = [0.5, 0.8]$.

4.1.2 ProCo RAT

The ProCo RAT is a popular distortion pedal introduced in the late 1970s, with a number of variants introduced throughout the years ever since [60]. While originally intended for electric guitar, it is also often used together with synthesizers to introduce harmonic distortion and complexity to the input audio. In this subchapter, the key characteristics of the circuit, shown in Figure 20, are reviewed and the range of user controls is covered.

A detailed circuit analysis of the ProCo RAT is presented in [60], and similar circuits are also analyzed in [61]. The circuit topology consists of a clipping stage, a tone control stage, as well as an output stage (highlighted with red, green and orange boxes in Fig. 20), not counting the power supply stage. The clipping stage serves both as an input buffer as well as an amplifier producing hard clipping with high enough amplitudes. The amplification is produced by an op amp and the surrounding passive components, while the hard clipping is produced by the anti-parallel connected pair of diodes. The tone stage is implemented as a passive one-pole low-pass filter, directly coupled to the clipping stage. The output stage serves as a buffer, isolating the clipping stage and the tone stage from the downstream circuitry, implemented as a simple JFET voltage follower. The nonlinear behavior of the circuit is produced for the most part by the clipping diodes as well as the JFET, while it is also possible to distort the op amp with high enough input amplitudes, although this is generally not desired.

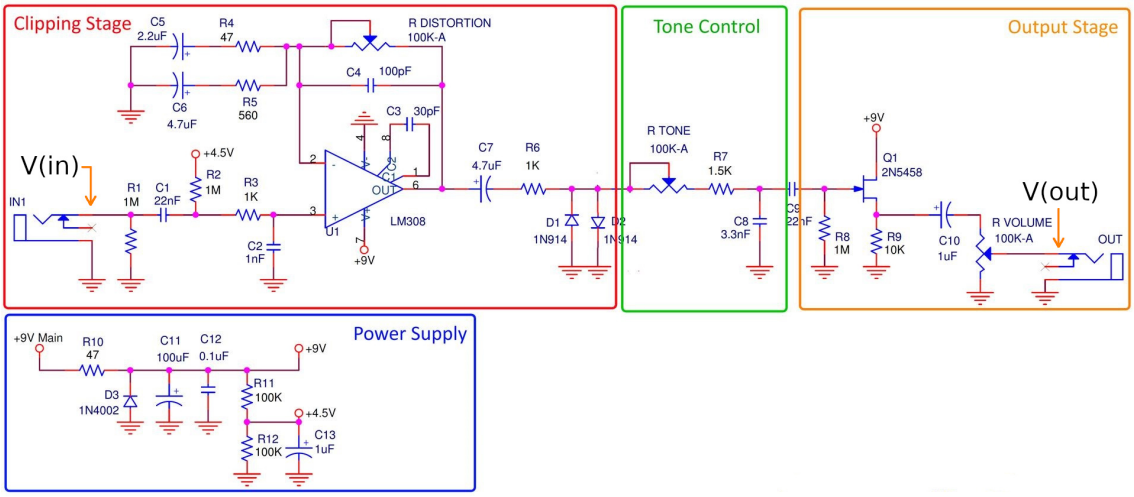


Figure 20: ProCo RAT, adapted from [60].

The user controls $\boldsymbol{\phi}_{RAT}$ of the circuit include three continuously variable parameters, one in each functional block. The controllable parameters are the distortion amount ϕ_{dist} (*R DISTORTION* in Fig. 20), the cutoff frequency of the tone stage ϕ_{tone} (*R TONE*), as well as the output volume ϕ_{vol} (*R VOLUME*), making the vector of user controls $\boldsymbol{\phi}_{RAT} = [\phi_{dist}, \phi_{tone}, \phi_{vol}]$ of size $N_\phi = |\boldsymbol{\phi}_{RAT}| = 3$.

Figure 21 shows the time-domain response of the circuit (orange line) with the user controls set to $\boldsymbol{\phi}_{RAT} = [1.0, 0.75, 0.75]$, in response to a passage of guitar playing (blue line). While the output follows the shape of the input audio, it is highly distorted, which can be seen as 'squaring up' of the waveform.

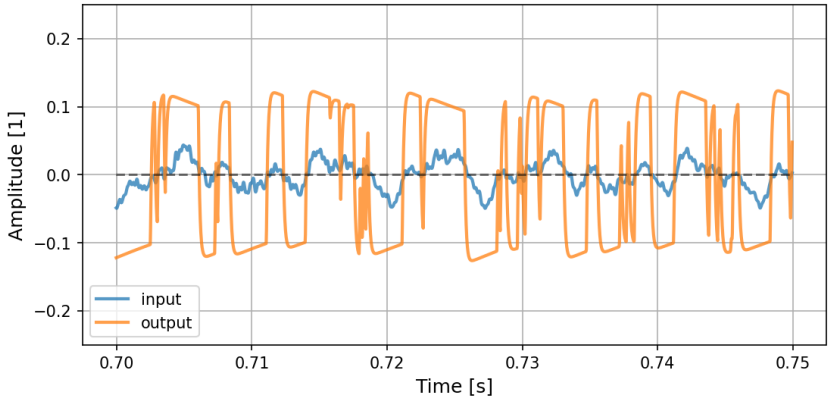


Figure 21: ProCo RAT example response, $\boldsymbol{\phi}_{RAT} = [1.0, 0.75, 0.75]$.

4.1.3 Pultec EQ

The Pultec EQP-1, hereby referred to as the Pultec EQ, is a nonlinear analogue equalizer used for mixing and mastering purposes. While the original Pultec EQ was among the first equalizers with multiple continuously adjustable bands, the device remains popular to this day due to its musical equalizing curves, as well as the soft

harmonic distortion produced by the vacuum tubes [62] and the transformers [63] utilized in the circuit. In this subchapter, the key characteristics of the Pultec EQ, shown in Figure 22, are reviewed and an overview of the user controls is given.

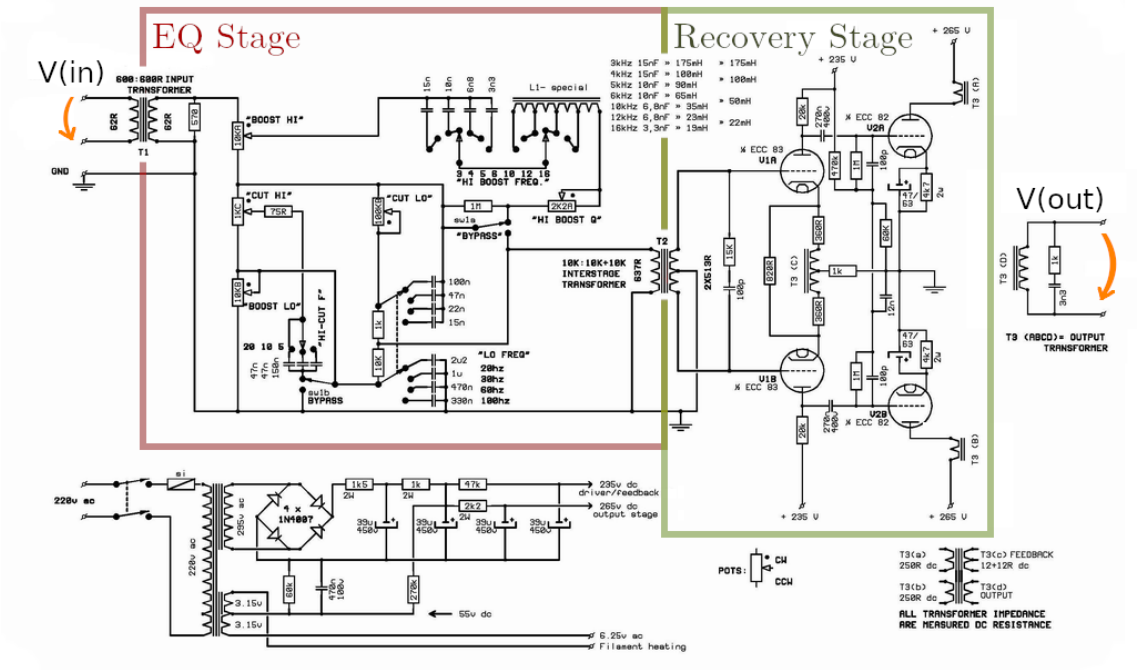


Figure 22: Pultec EQ, adapted from [64].

The circuit topology of the Pultec EQ consists of a passive (and linear) equalizer stage, a recovery stage, as well as a number of transformers connecting the blocks together, not counting the power supply. The equalizer stage is implemented entirely using capacitors and resistors, and includes all of the user controllable points of the circuit. The attenuated signal from the equalizer stage is brought up in level by the recovery stage, which is implemented as a vacuum tube amplifier in a push-pull configuration. The coupling of the global inputs and outputs of the circuit, as well as the interconnections between the functional blocks, is done utilizing various transformers. The nonlinear behavior of the circuit is produced by both the vacuum tubes in the recovery stage, as well as the various coupling transformers.

The user controls $\boldsymbol{\phi}_{PULTEC}$ of the circuit include five continuously variable parameters, as well as three switchable characteristics, not counting the bypass switch. The five continuously variable parameters are high frequency boost $\phi_{hiboost}$ (*BOOST HI* in Fig. 22), high frequency cut ϕ_{hicut} (*CUT HI*), low frequency boost $\phi_{loboost}$ (*BOOST LO*), low frequency cut ϕ_{locut} (*CUT LO*) as well as a resonance control for the high frequency boost ϕ_{hiQ} (*HI BOOST Q*). The switchable characteristics include individual controls for high frequency boost and high frequency cut cutoff frequencies (*HI BOOST FREQ.* and *HI-CUT F*), as well as a combined low frequency boost/cut cutoff frequency control (*LO FREQ*). For the scope of this work, the switchable characteristics were kept constant, making the vector of user controls $\boldsymbol{\phi}_{PULTEC} = [\phi_{hiboost}, \phi_{hicut}, \phi_{loboost}, \phi_{locut}, \phi_{hiQ}]$ of size $N_\phi = |\boldsymbol{\phi}_{PULTEC}| = 5$.

Figure 23 shows an example of a (linear) equalizing curve produced by the Pultec EQ (orange line) with the user controls set to $\phi_{PULTEC} = [0.75, 0.75, 1.0, 0.4, 0.0]$. In the example, the effects of the low frequency boost, low frequency cut, high frequency boost, and high frequency cut can be seen as various dips and notches in the magnitude response.

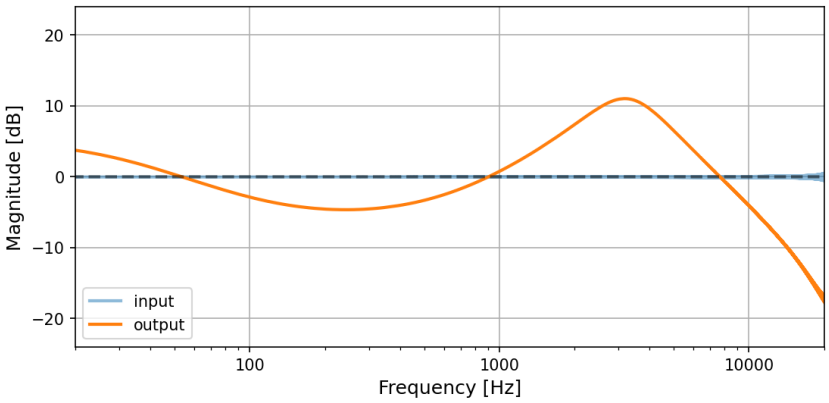


Figure 23: Pultec EQ example response, $\phi_{PULTEC} = [0.75, 0.75, 1.0, 0.4, 0.0]$.

4.2 Data

In this chapter, the procedure for generating the training data as well as the various datasets used in the experiments are given a detailed description. In order to gather training data for the various circuits presented in the previous chapter, SPICE simulations of each of the circuits are used, covered in Chapter 4.2.1. The method for sampling the space of parameters available in the original circuits is covered in Chapter 4.2.2, which also includes an overview of the various datasets used in the experiments.

4.2.1 Simulations

To allow for collecting training data from the circuits presented in the previous chapter, SPICE-simulations of each of the devices were used. LTSpice [65] was used for both creating the netlists, as well as running the simulations. In order to cover the circuit behavior in various operating conditions, the audio into the circuit, as well as the user controllable parameters were altered between each simulation round. The PyLTSpice [66] Python-wrapper for LTSpice was used to allow for adjusting these operating conditions programmatically. An example visualization of the ProCo RAT netlist, as rendered in LTSpice, is shown in Figure 24.

In all simulations, the various potentiometer laws other than the linear law encountered in the original circuits were modeled after the power approximation given in [67]. To allow for automated manipulation of the user controls, the potentiometer and voltage control points were parametrized using the SPICE parameter-directive and by letting PyLTSpice adjust these points with the provided methods. An example

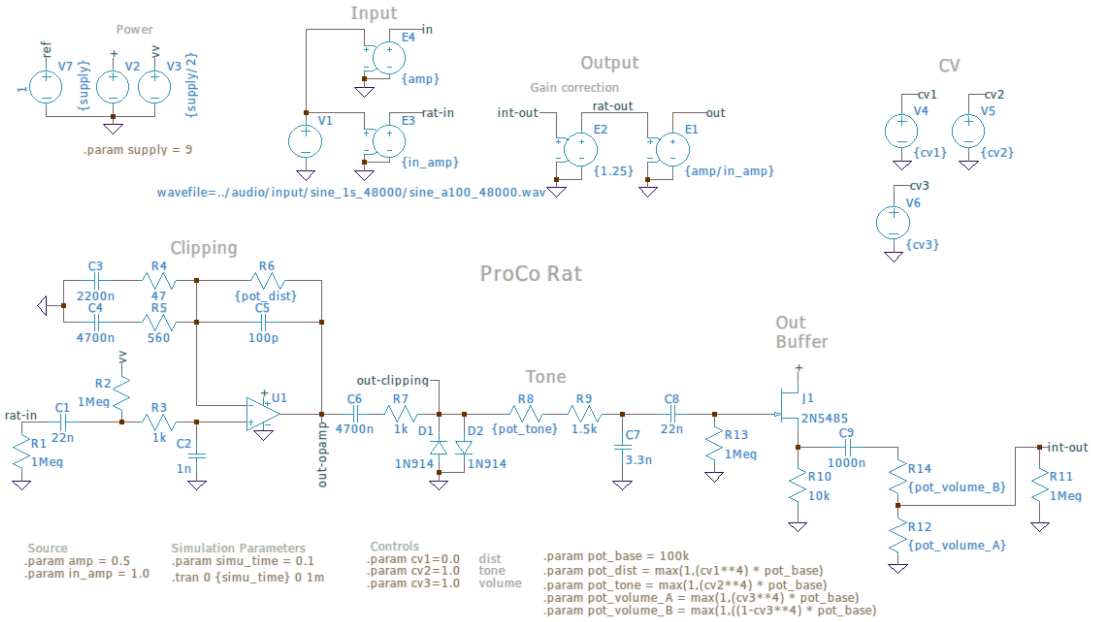


Figure 24: ProCo RAT Netlist.

parametrization of the potentiometers for the ProCo RAT can be seen in Figure 24 under the *Controls*-heading. All of the user controls $\phi_j \in \boldsymbol{\phi} = [\phi_1, ..., \phi_{N_\phi}]$ were parametrized so that their possible values are within a normalized interval $[0, 1]$.

An important detail in using the simulations was properly gain staging the audio into and out of the circuit, since the WAV-files used as the inputs and outputs only allow for amplitudes between $[-1, 1]$. Since the amplitude values of the WAV-files are directly converted into voltages when read by LTSpice, these amplitudes had to be scaled accordingly when entering and coming out of the circuit to make sure the nonlinear regions of the circuits were excited, as well as to prevent the output from clipping. In Figure 24, the gain staging is done with the ideal amplifier blocks under the *Input*- and *Output*-headings.

4.2.2 Parameter Sampling and Datasets

To tackle the original problem of studying the dataset properties affecting the modeling of the target circuits, a number of training sets for each of the targets were constructed. The study was divided into two experiments: one using constant-sized datasets and another using varying-sized datasets. The following paragraphs give a detailed description of the source audio used throughout the studies, the methods used to divide the space of parameters, as well as the resulting datasets.

The source audio for both of the experiments included passages of guitar and bass playing from [68] and [69], as well as synthesized white noise and logarithmic sine sweep segments at varying amplitudes. A sampling rate f_s of 44.1 kHz was used throughout the experiments. 60 seconds (1 minute) of audio from each of the categories was used, forming a 240 second (4 minute) collection of recordings, which was then divided into segments of length $N_{seq} = 1\,\text{s}$ to form a partial dataset

$\mathbb{D}' = \{\mathbf{x}^{(i)}\}_{i=1}^{240}$ of size $|\mathbb{D}'| = 240$ and length $|\mathbb{D}'| \cdot N_{seq} = 240\,\text{s}$. This collection of source audio is referred to as the *combined* (dataset) in the coming sections. The overall length of the dataset was chosen to be similar to the lengths of datasets used in related problems, for example [44, 16, 25].

In both of the experiments, each user control $\phi_j,\ j \in [1, N_\phi]$ of the target circuit was randomly sampled from a discrete uniform distribution $\mathcal{U}\{0, 1\}$ for each simulation round, and the number of allowed points k within the interval, hereby referred to as the *sampling density*, was made variable:

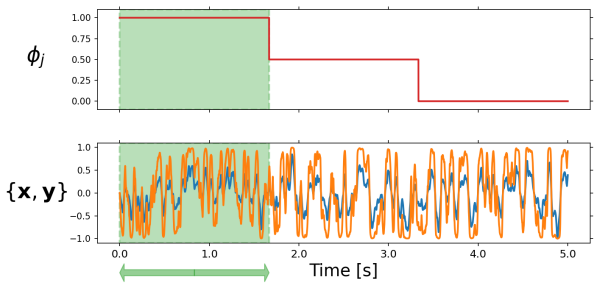
$$\phi_j \sim \mathcal{U}\{0, 1\}[k], \quad k \in \mathbb{N}^+. \tag{36}$$

For the datasets in the first experiment, the partial dataset $\mathbb{D}'$ was driven through each of the circuits, and each of the user controllable parameters ϕ_j of the circuit was sampled independently from $\mathcal{U}\{0, 1\}[k]$ for each simulation round, using four different sampling densities $k = [3, 5, 11, 101]$. The produced outputs $\mathbf{y}^{(i)}$, as well as the corresponding parameter values $\boldsymbol{\phi}^{(i)} = [\phi_1^{(i)}, ..., \phi_{N_\phi}^{(i)}]^T$, were collected to form four distinct datasets $\mathbb{D}_k = \{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}, \boldsymbol{\phi}^{(i)}\}_{i=1}^{240},\ k = [3, 5, 11, 101]$ of sizes $|\mathbb{D}_k| = 240$ and lengths $|\mathbb{D}_k| \cdot N_{seq} = 240\,\text{s}$ for each of the targets. These datasets are hereby referred to as *combined-3, combined-5, combined-11* and *combined-101*.

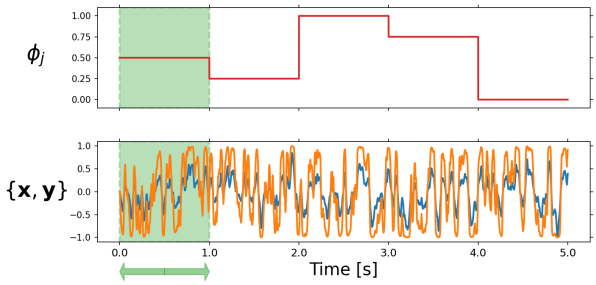
Constructing the datasets this way leads to four constantly sized datasets $|\mathbb{D}_3| = |\mathbb{D}_5| = |\mathbb{D}_{11}| = |\mathbb{D}_{101}| = 240$, with the expected value of the number of input/output-pairs N_{xy} produced with any of the user controls ϕ_j being set to any specific value $\phi_j = c$ being inversely proportional to the sampling density:

$$\mathcal{E}(N_{xy}) = \mathcal{E}(\,|\{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}, \boldsymbol{\phi}^{(i)} \mid \ \phi_j^{(i)} = c\}|\,) \propto \frac{1}{k}, \quad c \in \mathcal{U}\{0, 1\}[k]. \tag{37}$$

The intuition of the procedure is visualized in Figure 25 for a single parameter ϕ_j with two different sampling densities $k = [3, 5]$ and for an imaginary total dataset length of 5 seconds without further segmenting. The shaded green areas as well as the arrows in the figure visualize the length of audio corresponding to a specific parameter value, demonstrating the inverse proportionality. An overview of the properties of the Experiment 1 datasets are summarized in Table 1.



(a) Sampling density $k = 3$



(b) Sampling density $k = 5$

Figure 25: The inverse proportionality of the sampling density k and the length of audio produced with a specific parameter setting $\phi_j = c \in \mathcal{U}\{0, 1\}[k]$ in Exp. 1.

Table 1: Experiment 1 datasets

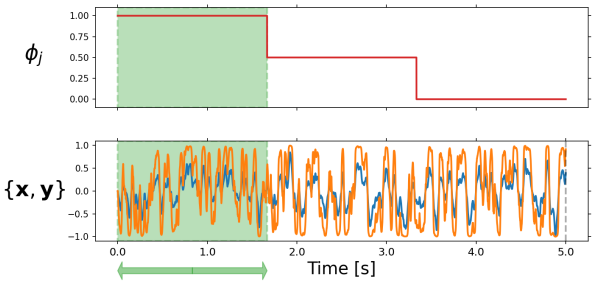
Name	Notation	$\lvert\mathbb{D}\rvert$	ϕ_j	$\mathcal{E}(N_{xy})$
combined-3	$\mathbb{D}_3$	240	$\sim \mathcal{U}\{0, 1\}[3]$	80
combined-5	$\mathbb{D}_5$	240	$\sim \mathcal{U}\{0, 1\}[5]$	48
combined-11	$\mathbb{D}_{11}$	240	$\sim \mathcal{U}\{0, 1\}[11]$	≈ 22
combined-101	$\mathbb{D}_{101}$	240	$\sim \mathcal{U}\{0, 1\}[101]$	≈ 2

For the datasets in the second experiment, the inverse proportionality introduced in Equation (37) was compensated by reusing a portion of the original source audio $\mathbb{D}'$ such that the expectation of the number of input/output pairs $\mathcal{E}(N_{xy})$ corresponding to any specific user control value $\phi_j = c$ was independent of the sampling density used:

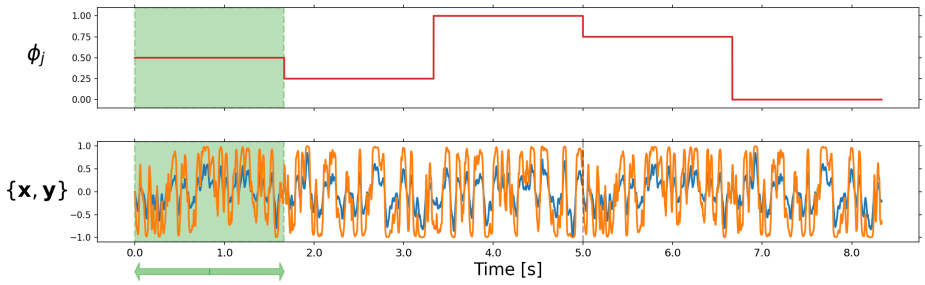
$$\mathcal{E}(N_{xy}) = \mathcal{E}(\,|\{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}, \boldsymbol{\phi}^{(i)} \mid \phi_j^{(i)} = c\}|\,) = d, \quad c \in \mathcal{U}\{0, 1\}[k],\ d \in \mathbb{R}^+. \tag{38}$$

In practice, this was achieved by taking the expectation of the number of pairs N_{xy} produced by the sparsest sampling $\mathcal{E}(N_{xy}[k = 3]) = 80$, and reusing $\mathbb{D}'$ such that all of the samplings produced the same expectation. The intuition is again visualized in Figure 26 for a single parameter ϕ_j, sampling densities $k = [3, 5]$ and total dataset length of 5 seconds. As can be seen from the figure, the inverse proportionality of

the sampling density and the length of audio content corresponding to a specific parameter value is compensated for.



(a) Sampling density $k = 3$



(b) Sampling density $k = 5$

Figure 26: The independence of the sampling density k and the length of audio produced with a specific parameter setting $\phi_j = c \in \mathcal{U}\{0, 1\}[k]$ in Exp. 2.

Constructing the datasets this way leads to datasets $\mathbb{D}_{k,ext}$ of varying sizes $|\mathbb{D}_{k,ext}|$, but with the expected value of number of input/output pairs $\mathcal{E}(N_{xy})$ corresponding to each specific value c of any of the parameters ϕ_j being constant, in contrast to the first experiment, where the situation was (roughly) the opposite. Another difference in the second formulation is the higher computational costs for generating these possibly larger datasets, stemming from having to run a higher number of simulation rounds to generate the needed data. As such, only 3 out of the 4 sampling densities from the first experiment were considered, namely $k = [3, 5, 11]$, with the resulting datasets referred to as *combined-3-extended, combined-5-extended* and *combined-11-extended*:

Table 2: Experiment 2 datasets

Name	Notation	$\lvert\mathbb{D}\rvert$	ϕ_j	$\mathcal{E}(N_{xy})$
combined-3-extended	$\mathbb{D}_{3,ext}$	240	$\sim \mathcal{U}\{0, 1\}[3]$	80
combined-5-extended	$\mathbb{D}_{5,ext}$	400	$\sim \mathcal{U}\{0, 1\}[5]$	80
combined-11-extended	$\mathbb{D}_{11,ext}$	880	$\sim \mathcal{U}\{0, 1\}[11]$	80

Finally, to keep the number of optimizer steps constant for the models trained in the second experiment, the number of training epochs was scaled according to the

dataset length $|\mathbb{D}|$. This is explained at the end of the following chapter (Chapter 4.3.4).

4.3 Recurrent Modeling and Training

This chapter gives an overview of the neural aspects of the modeling task, including the used network architecture, the loss function, as well as the training scheme used for tuning the model parameters. Each of these aspects are covered in their respective subchapters. Finally, at the end of the chapter, the method for normalizing the compute across the experiments is reviewed.

4.3.1 Model

For learning the behavior of each of the modeling targets as represented by the collected datasets, a recurrent neural network consisting of a GRU and a fully-connected output layer was used, similar to what was described in Chapter 3.2. During the initial experiments a convolutional model, similar to what was described in Chapter 3.1, was also used, but the model was not chosen for further study as the preliminary results showed it performed worse than the recurrent model. To choose the RNN hyperparameters, namely, the hidden size of the GRU, a hyperparameter search was conducted. The input size of the MLP was set to match the hidden-size of the GRU. The results of the hyperparameter search are covered together with the other results in Chapter 5.

The model was implemented in Python, using the PyTorch [56] machine learning framework. In PyTorch the neural networks are defined as subclasses of the `torch.nn.Module`-class, which takes care of proper initializations and internal methods; the neural network architecture is defined in class member variables and the forward pass for computing the model output is defined as a `forward`-function. During training, PyTorch keeps track of the computational graph resulting from the forward pass, allowing for efficient computation of the gradient with backpropagation. Since the used model architecture was comprised of standard ML blocks, readily implemented (and optimized in C++) classes for the GRU and the MLP were used. A visualization of the model implementation source code is shown in Figure 27.

4.3.2 Loss

For evaluating the quality of the predictions produced by the model, the error-to-signal ratio (ESR) loss was used, which is a common criterion used in VA modeling tasks [16, 25, 51]. ESR loss is defined as:

$$\varepsilon = \frac{\sum_{n=0}^{N-1} |y[n] - \hat{y}[n]|^2}{\sum_{n=0}^{N-1} |y[n]|^2}, \tag{39}$$

where $y[n]$ is the target output, $\hat{y}[n]$ the model output and N is the length of the sequences. The term in the denominator normalizes the metric with respect to

```python
class NeuralNetwork(nn.Module):

    def __init__(self, hidden_size, input_size):

        super(NeuralNetwork, self).__init__() # Initialize base-class

        # Object parameters
        self.hidden_size = hidden_size # hidden size of GRU
        self.input_size = input_size   # input features at timestep n
        self.num_layers = 1            # number of GRUs
        self.output_size = 1           # output size at timestep n

        # Define Network
        self.gru = nn.GRU(input_size=self.input_size,
                          hidden_size=self.hidden_size,
                          num_layers=self.num_layers,
                          batch_first=True)
        self.linear = nn.Linear(self.hidden_size, self.output_size)

        # Initialize hidden state
        self.reset_hidden()

    def forward(self, input):

        hidden_out, self.hidden = self.gru(input, self.hidden) # GRU output
        y_hat = self.linear(hidden_out)                        # Linear layer output
        return y_hat

    def reset_hidden(self):

        self.hidden = None
```

Figure 27: Model implementation in Python/PyTorch.

the energy of the target signal, preventing high energy segments from biasing the computations.

Even though the findings in [70] would suggest using a perceptually motivated pre-emphasis filter before computing the loss to help the model focus on perceptually relevant audio content, this was not done for the scope of this work due to the findings during the early experimental phase. Similarly, a combined loss comprising of multiple objectives taking into account different aspects of the target output [16, 45] was not considered.

4.3.3 Training

For training the models, the different datasets $\mathbb{D}$ were randomly split into training and validation sets of proportions $0.85 : 0.15$, out of which the former was used for optimizing the model parameters, and the latter for validating the performance of the model during training. To compute the gradient of the loss with respect to the model parameters, a variant of regular backpropagation called backpropagation through time (TBPTT) was used. For updating the model parameters, a flavor of SGD implemented via the popular Adam optimizer [71] was utilized.

When applying backpropagation to recursive neural networks like the one used for this study, the computations within the recursive computational graph are

unfolded to a regular directional graph, allowing the gradient to be computed using standard backpropagation rules as long as the parameter sharing between the computational steps are taken into account. This time-unfolded variant of the backpropagation algorithm is called backpropagation through time (BPTT). In truncated backpropagation through time (TBPTT), instead of traversing the whole of the input sequence when computing the gradient, the input sequence is divided into smaller segments and the optimizer is called at the end of each segment. This truncated variant of the BPTT algorithm results in higher number of optimizer steps per epoch, as well as decreases the computational costs for a single backpropagation call.

To update the model parameters after the gradient computations, the Adam optimizer was used. The optimization algorithm implemented in Adam takes into account a number factors in each optimization step, such as the momentum of the gradient and the differing magnitudes of the different partial derivatives within it, and dynamically adjusts the learning rate according to the error surface to form a highly sophisticated way of updating the model parameters. As such, Adam is one of the most popular optimizer algorithms used for stochastic gradient descent, and often encountered in neural VA modeling tasks.

4.3.4 Epoch Scaling

To keep the number of optimizer steps constant for models trained with datasets of varying sizes (Chapter 4.2.2), the number of training epochs E was scaled according to the dataset size $|\mathbb{D}|$. The number of optimizer steps η per epoch can be computed, when applying TBPTT, as:

$$\eta = N_B \lceil \frac{N_{seq} - N_{init}}{N_{TBPTT}} \rceil, \tag{40}$$

where N_B is the number of mini-batches in an epoch, N_{seq} is the sequence length used within a mini-batch in samples, N_{init} is the number of time steps the model is allowed to initialize for before tracking the gradient and N_{TBPTT} is the number of time steps used for computing the gradient. $\lceil \cdot \rceil$ denotes the ceiling function.

The total number of optimizer steps is then:

$$\eta_{total} = E\eta, \tag{41}$$

where E is the number of epochs used for training. Since N_B depends on the size of the dataset $|\mathbb{D}|$ as well as the used batch-size $|\mathbb{B}|$, the total number of optimizer steps η_{total} can be kept constant by decreasing E for the larger datasets. As such, using a batch-size of $|\mathbb{B}| = 128$, $N_{seq} = 44100$ and $N_{init} = N_{TBPTT} = 2^{10} = 1024$ the following number of training epochs was used for the different datasets:

Table 3: Epoch scaling, GCF denotes the *greatest common factor*

Dataset	$\lvert\mathbb{D}\rvert$	η	E	η_{total}
$\mathbb{D}_3, \mathbb{D}_5, \mathbb{D}_{11}, \mathbb{D}_{101}$	240	86	$GCF(86, 129, 258) \cdot 3 = 43 \cdot 3 = \mathbf{129}$	11094
$\mathbb{D}_{3,ext}$	240	86	**129**	11094
$\mathbb{D}_{5,ext}$	400	129	$11094/129 = \mathbf{86}$	11094
$\mathbb{D}_{11,ext}$	880	258	$11094/258 = \mathbf{43}$	11094

5 Results

In this chapter, the results gathered from the experiments conducted throughout the study are presented. For the evaluation of the performance of the models in the various experiments, a number of separate testsets are constructed, presented in Chapter 5.1. The actual results include the outcomes from the conducted hyperparameter search for tuning the model parameters (Chapter 5.2), as well as the outcomes from using the different datasets for training the models (Chapter 5.3). Finally, to relate the achieved loss metrics to the quality of the trained models, Chapter 5.4 presents an additional listening test conducted to provide a subjective evaluation of the trained models.

5.1 Testsets

For evaluating the trained models, an additional partial dataset $\mathbb{D}'_{test}$ was constructed, comprising of unseen guitar and bass passages from the same sources that were used for the training [68, 69]. 60 seconds (1 minute) of audio from both of the categories was used, forming a 120 second (2 minute) collection of recordings, which was then divided into segments of length $N_{seq} = 1\,\mathrm{s}$ to make up the partial testset $\mathbb{D}'_{test} = \{\mathbf{x}^{(i)}\}_{i=1}^{120}$.

To construct the full testset, the partial dataset $\mathbb{D}'_{test}$ was driven through each of the target circuits, with each user control ϕ_j being sampled from a uniform distribution $\mathcal{U}\{0, 1\}$ with a dense sampling grid $k = 101$ for each simulation round. This dense sampling grid, together with the unseen audio passages, was meant to mimic the conditions during inference. This procedure was repeated five times to produce five testsets $\mathbb{D}_{test,l} = \{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}, \boldsymbol{\phi}^{(i)}\}_{i=1}^{120},\ l \in [1, 5]$ for each target. Note, that due to the random nature of the sampling procedure for setting ϕ_j, the contents of each of the datasets $\mathbb{D}_{test,l}$ differ from one another, even though the generation procedure is the same. To evaluate the different models in the coming chapters, the average of the minimum losses achieved over these testsets is used.

5.2 Hyperparameter Search

Table 4 lists the averages of the minimum losses achieved over the testsets for each modeling target using RNNs with varying hidden sizes for the GRU. The GRU hidden sizes $h_s = [2, 4, 8, 16, 32]$ were tested, with the maximum hidden size chosen as per the findings in [24]. In each row, the minimum achieved average loss is highlighted in bold.

Table 4: Hyperparameter search

	Hidden size h_s				
Target	2	4	8	16	32
MS10 Filter	0.671	0.646	0.639	0.656	**0.635**
ProCo RAT	0.591	0.400	0.525	0.349	**0.328**
Pultec EQ	0.740	0.196	0.128	0.091	**0.055**

Across the targets, increasing the hidden size improved the model performance when evaluated on the testsets, as expected. The most dramatic improvement can be seen for the Pultec EQ, for which the largest model achieved a loss over $\times 13$ smaller compared to the smallest model. On the other hand, increasing the model size for the MS10 Filter didn't bring much of an improvement for the larger models, with the loss metric being only $\approx \times 1.05$ better for the largest model in comparison to the smallest model. This finding would point to the model architecture not being adequate for modeling this specific target circuit. The results for the ProCo RAT were in between the extremes set by the other targets, with the largest model producing a $\times 2$ improvement for the loss in comparison to the smallest model. For an unknown reason, the ProCo RAT model with a hidden size $h_s = 8$ produced an average loss that was larger than the loss for the slightly smaller model with $h_s = 4$, standing out from the general pattern. Still, motivated by the overall trend, the largest models with hidden sizes $h_s = 32$ were used for modeling all of the targets, and for the rest of the experiments.

5.3 Learning Parameter Spaces

The following subchapters present the main research outcomes for the original task of generalizing over the parameter spaces of modeling targets with multiple user controls. The performance of the models trained with the differing datasets from Chapter 4.2.2 are evaluated in terms of the averages and standard deviations of the minimum losses achieved over the testsets from Chapter 5.1. The results are presented separately for the different modeling targets in an order dictated by the number of adjustable user controls.

5.3.1 MS10 Filter

The results for the MS10 Filter are shown in Figure 28, where the loss metrics are presented as a combined bar graph, as well as listed separately for the two groups of datasets in Tables 5 and 6. The minimum achieved average loss, as well as the minimum achieved standard deviation is highlighted in bold globally across the training sets.

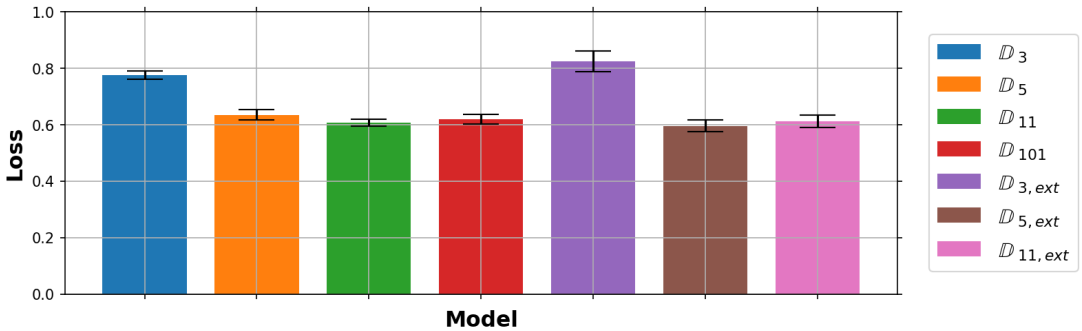


Figure 28: Results - Learning Parameter Spaces, MS10 Filter.

Table 5: MS10 Filter - Experiment 1 losses

	Training set			
Loss	$\mathbb{D}_3$	$\mathbb{D}_5$	$\mathbb{D}_{11}$	$\mathbb{D}_{101}$
Average	0.776	0.635	0.607	0.62
STD	0.014	0.019	**0.013**	0.017

Table 6: MS10 Filter - Experiment 2 losses

	Training set		
Loss	$\mathbb{D}_{3,ext}$	$\mathbb{D}_{5,ext}$	$\mathbb{D}_{11,ext}$
Average	0.824	**0.596**	0.612
STD	0.037	0.021	0.021

As can be seen from Figure 28 and Tables 5 and 6, the performance of the models trained with sampling densities $k \geq 5$ are relatively uniform, and the losses of the models trained using the sparsest sampling $k = 3$ are considerably larger in comparison. The average minimum achieved loss for the best performing model is approximately $\times 1.4$ better than for the worst performing model. Comparing the losses of the models with $k \geq 5$ trained with or without extending the datasets shows both slight decrease ($\mathbb{D}_5$ vs. $\mathbb{D}_{5,ext}$) as well as slight increase ($\mathbb{D}_{11}$ vs. $\mathbb{D}_{11,ext}$) of the average achieved loss. The minimum average loss was achieved with the model trained with the $\mathbb{D}_{5,ext}$ set, although taking into account the standard deviations this might hold in the general sense. Taking into account the computational costs of the dataset generation procedures, the results would indicate that the denser leaning datasets without extensions (e.g. $\mathbb{D}_{11}$) work best for this task, since the dataset extension doesn't bring much of an improvement. On the other hand, since the hyperparameter search for the modeling target showed incapability of the chosen model architecture to learn the target behavior better even with much improved theoretical generalization capabilities provided by the increasing hidden size, it might very well be that the chosen network architecture is inadequate for the task, invalidating partly these results.

5.3.2 ProCo RAT

The results for the ProCo RAT are shown in Figure 29, with the loss metrics shown again as a combined bar graph, as well as listed separately in Tables 7 and 8. The best metrics are highlighted in bold globally across the groups of training sets.

As can be seen from the results, the difference between the best performing and the worst performing model is much bigger than was the case with the MS10 Filter, approximately $\times 3.2$ for the average minimum loss achieved. Similarly as before, the models trained with the sparsest sampling density $k = 3$ are seen performing worst for both groups of experiments, indicating that the sparse sampling is not capturing the target behavior well. It is somewhat peculiar, that the "extended" sparsest dataset $\mathbb{D}_{3,ext}$ is seen giving much better performance metrics compared to the "unextended"

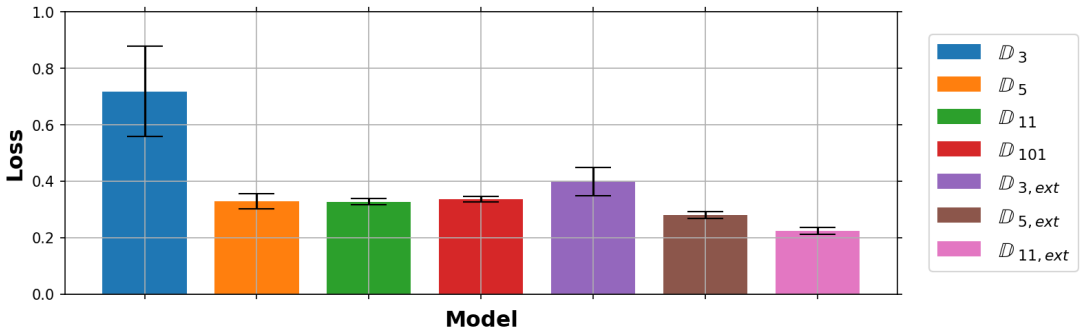


Figure 29: Results - Learning Parameter Spaces, ProCo RAT.

Table 7: ProCo RAT - Experiment 1 losses

	Training set			
Loss	$\mathbb{D}_3$	$\mathbb{D}_5$	$\mathbb{D}_{11}$	$\mathbb{D}_{101}$
Average	0.718	0.328	0.327	0.336
STD	0.159	0.027	0.011	**0.010**

Table 8: ProCo RAT - Experiment 2 losses

	Training set		
Loss	$\mathbb{D}_{3,ext}$	$\mathbb{D}_{5,ext}$	$\mathbb{D}_{11,ext}$
Average	0.399	0.280	**0.223**
STD	0.050	0.013	0.012

one $\mathbb{D}_3$ across the testsets, since the extension in this case doesn't actually produce a larger dataset, as was listed in Table 3. Having said this, extending the datasets improves the average minimum losses achieved over the testsets for all the sampling densities compared, to the point that the minimum average loss over the testsets was achieved with the densest and largest dataset $\mathbb{D}_{11,ext}$. Since the amount of compute is normalized across the different experiments as per Chapter 4.3.4, it should be safe to say that in this case, extending the datasets gives additional benefits in modeling the target behavior, in comparison to just using the available compute to optimize the parameters of the model with the already seen data for higher number of epochs.

5.3.3 Pultec EQ

The results for the Pultec EQ are shown in Figure 30, with the loss metrics shown once again as a combined bar graph and listed separately in Tables 9 and 10, with the best global metrics highlighted in bold. It should be noted, that the scale of the y-axis is magnified in comparison to earlier, due to the overall loss metrics being much lower than before. This does not directly indicate better performing models, since the effects produced by the modeling targets are both different in nature, and also require different gain-staging for the inputs and outputs, affecting the overall

scale of the metrics.

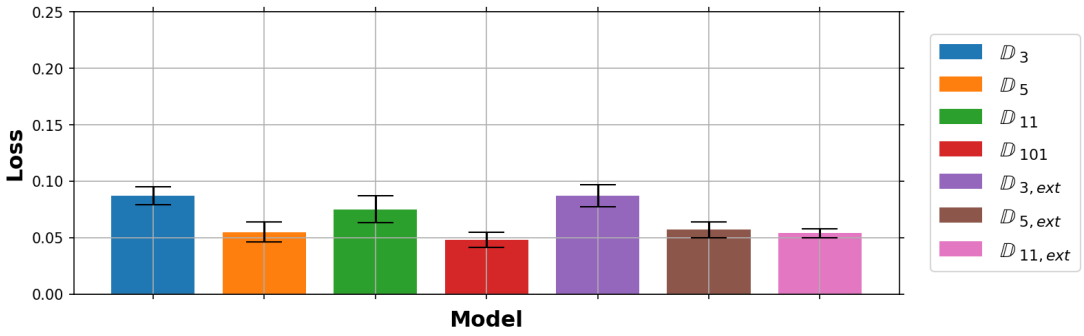


Figure 30: Results - Learning Parameter Spaces, Pultec EQ.

Table 9: Pultec EQ - Experiment 1 losses

	Training set			
Loss	$\mathbb{D}_3$	$\mathbb{D}_5$	$\mathbb{D}_{11}$	$\mathbb{D}_{101}$
Average	0.087	0.055	0.075	**0.048**
STD	0.008	0.009	0.012	0.007

Table 10: Pultec EQ - Experiment 2 losses

	Training set		
Loss	$\mathbb{D}_{3,ext}$	$\mathbb{D}_{5,ext}$	$\mathbb{D}_{11,ext}$
Average	0.087	0.057	0.054
STD	0.010	0.007	**0.004**

The results for the Pultec EQ show differences between the best performing and the worst performing models in between the extremes reported earlier, approximately $\times 1.8$ for the minimum achieved average loss. Similarly as before, the denser sampling densities are seen giving better overall performance in both of the experiments, with the only exception being the unexpectedly large average minimum loss achieved by the model trained on the $\mathbb{D}_{11}$ set. The sparsest sampling density $k = 3$, like before, seems to be inadequate for capturing the target behavior well enough, producing the worst models for both of the groups. While the lowest average minimum loss is achieved with the model trained with the densest sampling grid $k = 101$, together with the standard deviation this might not hold in the general sense. Extending the dataset sizes is seen to decrease the standard deviation of the error, resulting in a more consistent performance over the testsets. Still, taking into account the increased computational costs in creating the larger datasets, it is not necessarily clear whether the more involved generation procedure can be justified in this case.

5.4 Listening Test

To gain further insight into the loss metrics and their correspondence with the quality of the trained models, a listening test was conducted for further evaluation. The following paragraphs present the setup of the listening test, as well as the results.

The listening test was organized following the MUSHRA-standard [72] implemented using the webMUSHRA-framework [73], the interface of which is shown in Figure 31. In MUSHRA-style listening tests, the participants are asked to give an overall assessment of the sound quality of various test items, called *conditions*, in comparison to a *reference*, in multiple individual tests called *trials*. In each trial, the sound quality of the conditions is given as a single number between $[0, 100]$, with 0 corresponding to worst possible perceived sound quality, and 100 corresponding to no perceived difference. An additional perceptual scale ranging from *Bad* to *Excellent* is given on top of the numerical scale. Since these ratings are highly subjective in nature, MUSHRA-style tests use one or more low-quality *anchors* and a *hidden reference* within the conditions, in order to fix the low and high ends of the scale.

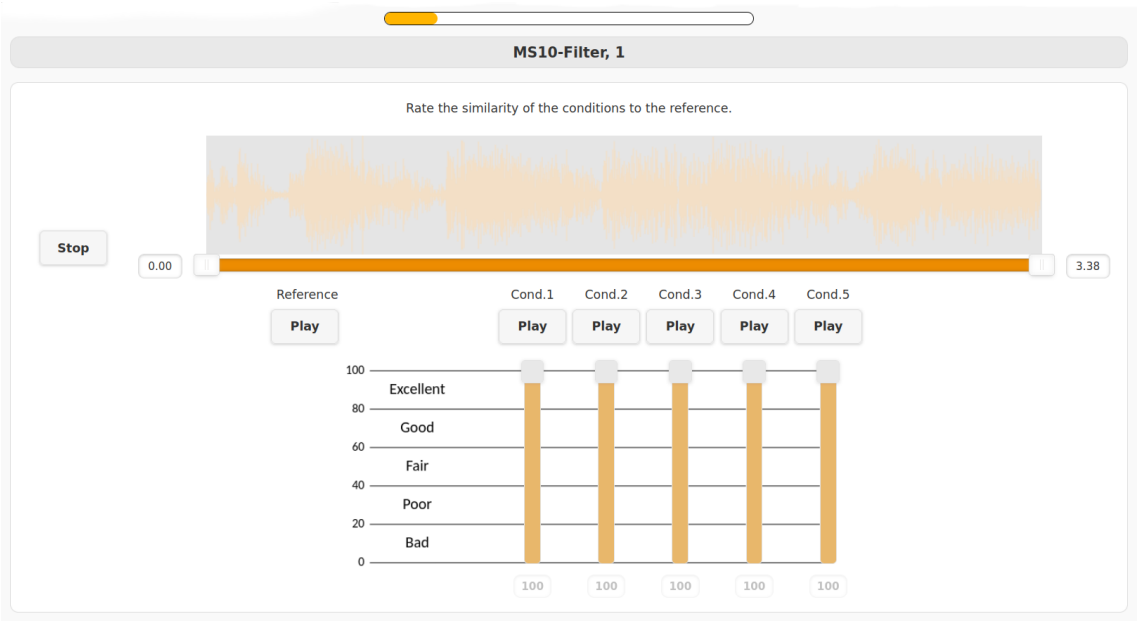


Figure 31: webMUSHRA interface.

In order to understand how the error metrics produced by the various trained models correspond to their perceived sound quality, the models trained with the most differing datasets were chosen for each of the targets. In practice, this meant taking the models trained with the $\mathbb{D}_3$ and $\mathbb{D}_{11,ext}$ datasets, which were also found producing contrasting error metrics in the previous chapter. In addition to this, two anchors were used in attempt to set the low end of the scale, namely, the smallest models with hidden size $h_s = 2$ trained during the hyperparameter search, as well as a static model consisting of a `tanh()`-function driven with a gain of 25 and filtered with a first-order high-shelving filter having the cutoff and gain at $\approx 5.5\,\text{kHz}$ and $-18\,\text{dB}$ respectively. With the hidden reference and the two anchors, the total number of

conditions for each trial was thus 5.

To assess the quality of the chosen models, yet another set of source audio consisting of short segments of various genres of music was collected, and each of these segments was driven with various randomly generated parameter settings through each of the models. To match the perceived loudness of the conditions, the *pyloudnorm*-library for Python [74] was used to normalize the perceived level to −23 dB LUFS according to broadcast audio loudness metering standards [75]. 10 trials were produced for each of the targets, making up a total of 30 trials for the full listening test. During this pre-listening and selection stage, it was found that none of the MS10 Filter models were able to replicate the self-oscillating character of the original device, illustrated in Figure 32, giving insight into the relatively high error metrics produced earlier. Trials involving matching this behavior were not chosen for the listening test.

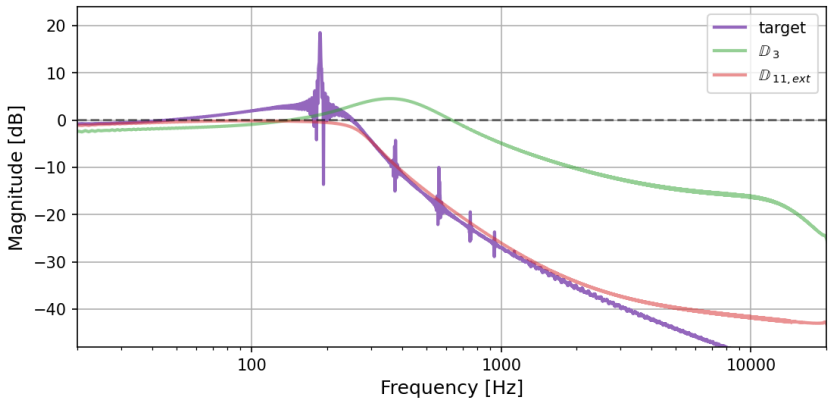


Figure 32: MS10 Filter model responses - self-oscillation problem.

The listening test was conducted in soundproofed listening booths equipped with sets of Sennheiser HD650 headphones and Objective2 ODAC heapdhone amplifiers. In total, 19 participants without reported hearing impairments took the listening test with each participant taking between 30 and 45 minutes to finish the full test. The results from the listening tests are shown in Figure 33, with each target shown in a separate subfigure. In each subfigure, the mean of the ratings given by the participants, as well as the corresponding 95% confidence interval, is shown, together with a scatter plot representing the underlying distribution. The 95% confidence intervals were computed as according to the student's t inverse cumulative distribution function [76].

As can be seen from Figure 33, the reference sound (illustrated in violet) and the `Tanh()`-anchor (illustrated in blue) were given the best and the worst overall ratings, verifying the collected results. In all cases, the models trained with the most diverse sets of data ($\mathbb{D}_{11,ext}$, illustrated in red), can be seen performing better than the models trained with the most limited sets of data ($\mathbb{D}_3$, illustrated in green) and both of the anchors. In all cases, this resulted in the models trained with the most diverse sets of data receiving average ratings that are at least one perceptual rating category higher than the rest.

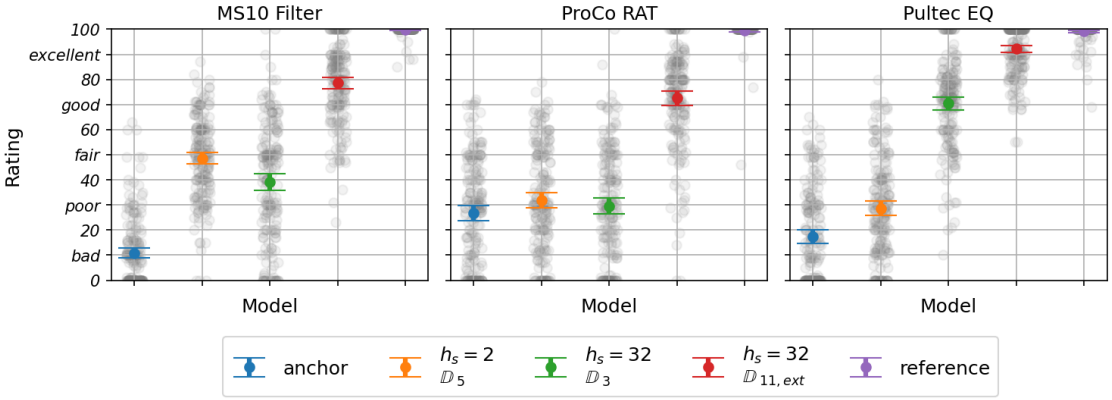


Figure 33: MUSHRA results.

In the case of the MS10 Filter, all of the models, including the best performing one, had difficulties in replicating the most extreme settings of the cutoff parameter. This behavior degraded the overall ratings of all of the models, and is illustrated in Figure 34. The model that was trained with the most diverse set of data received an average rating between *Good* and *Excellent*, leaving room for improvements to match the reference. The model that was trained with the most limited set of data can be seen performing worse than the hidden-size $h_s = 2$ anchor (illustrated in orange in Fig. 33), gaining no benefit from the much larger model size.

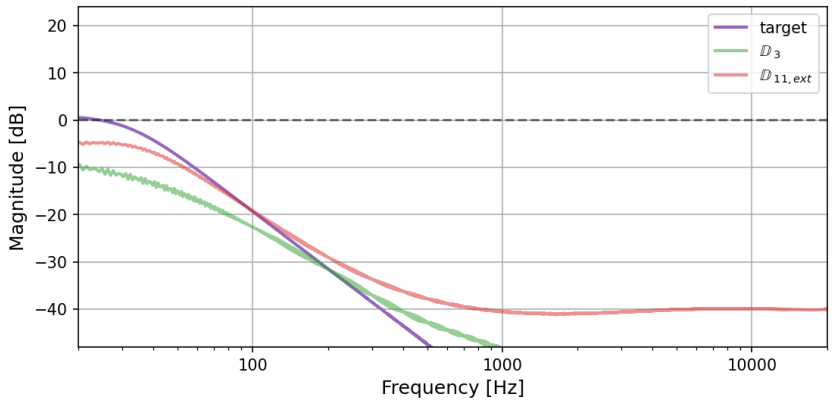


Figure 34: MS10 Filter model responses - extreme cutoff problem.

In the case of the ProCo RAT, all of the models, including the best performing one, had difficulties in replicating the highest gain settings well, reducing the overall ratings of all of the models. This behavior is illustrated in Figure 35. The model that was trained with the most diverse set of data received an average rating of *Good*, leaving room for further improvements. The model that was trained with the most limited set of data and the two anchors can be seen performing statistically the same, all receiving a *Poor* rating. It is interesting to note, how using a highly saturated and low-passed variant of the input as one of the anchors didn't produce a consistent *Bad* rating, which would have been desired.

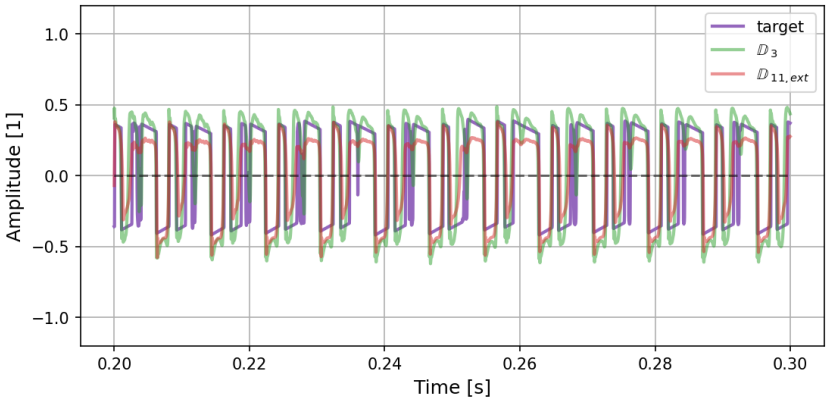


Figure 35: ProCo RAT model responses - high gain problem.

In the case of the Pultec EQ, the compared models (not including the two anchors) were seen performing the best across the different targets, *Excellent* for the model that had the most diverse training set, and *Good* for the model that had the most limited dataset. Even though both of the compared models were able to replicate the target behavior well, the match was better for the model that was trained with the most diverse training set, as illustrated in Figure 36. This was the only case when the model that was trained with the most limited dataset was seen performing better than both of the anchors. The design of the anchor models was also the most successful in this case, with both of them receiving a *Poor* rating or worse.

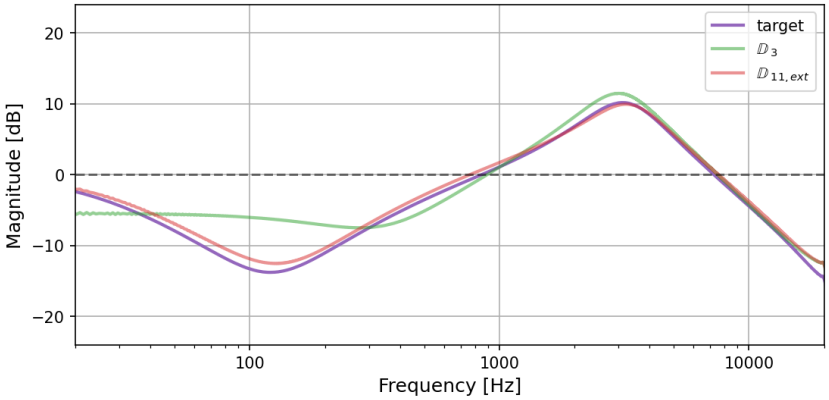


Figure 36: Pultec EQ model responses.

6 Conclusions

This thesis studied black-box VA modeling formulated as a machine learning sequence modeling problem. More specifically, the focus was on learning tasks where the target devices have multiple user controls, and the goal was to evaluate the effects the training dataset properties have on the generalization of the learning algorithm.

The problem was studied via taking multiple modeling targets, generating multiple differing training datasets for each of them, and evaluating and comparing the performance of neural networks trained on the datasets against one another. The target devices for the study included three nonlinear analogue sound processing devices with varying complexities, uses and numbers of user controls: a nonlinear filtering stage of an analogue synthesizer, a nonlinear guitar distortion effect, as well as a saturating analogue equalizer. These devices are referred to as the *(Korg) MS10 Filter*, *ProCo RAT* and *Pultec EQ*, respectively.

For collecting the training datasets for each of the targets, SPICE simulations were used. The datasets consisted of training examples of input/output pairs of discrete-time audio sequences, together with the target device configurations as represented by the user controls. For generating each of the training examples, the values of the user controls were sampled independently from a uniform distribution spanning the available range of the settings. For each of the targets, two groups of datasets were constructed, seven in total, with the first group consisting of constantly-sized datasets, and the second of varying-sized datasets. The difference between the datasets is in the density of the sampling grid k used for setting the user controls of the targets, as well as in the number of input/output pairs corresponding to any distinct value of any of the controls.

For modeling the targets, a recurrent neural network consisting of a Gated Recurrent Unit and a fully-connected layer was used, with the hidden size of the GRU set to 32 according to a conducted hyperparameter search. Seven neural networks were trained for comparison for each of the targets, one for each training set. To normalize the compute available during training for the models trained on the possibly differently sized datasets, the number of training epochs was scaled as a function of the dataset size. To evaluate the performance of the models, the average minimum loss achieved over additional five unseen testsets, well as the corresponding standard deviation, was used.

For the targets considered during the study, the sparsest evaluated sampling density $k = 3$ was found inadequate for the models to generalize over the testsets, which could be seen as the error metrics being the worst for the models trained on the corresponding datasets. For the models that were trained on datasets with higher sampling densities $k \geq 5$, the results were somewhat target dependent, while still providing some useful insights. For the ProCo RAT, extending the dataset size together with using higher sampling densities improved the model performance consistently, with the best performance achieved with the model trained on the largest and densest dataset. For the Pultec EQ, increasing the sampling density was seen improving the average of the minimum loss for both groups of datasets, with an exception being the model trained on the $k = 11$ dataset without extension, for

which the performance was worse than expected. Finally, for the MS10 Filter, the performance of the models trained on datasets with sampling densities $k \geq 5$ was relatively uniform without large differences in either the averages nor the standard deviations of the achieved minimum losses. Further inspection revealed that the chosen network architecture was inadequate for replicating the target behavior across all device configurations, possibly explaining why altering the datasets didn't produce drastic differences in the model performances.

To provide subjective evaluation to the trained models, an additional MUSHRA-style listening test was conducted. Two models for each of the targets were chosen for comparison and additional two low-quality anchors were used to set the low end of the perceptual evaluation scale. The models chosen for comparison for each of the targets were the model trained on the sparsest $k = 3$ sampling grid without dataset extension and the model trained on the extended dataset with the densest grid $k = 11$. Across the targets, the models trained on the dense and extended datasets were perceived as performing at least one perceptual rating category better than all the other models.

For future work, the study could be extended to include multiple neural network architectures for comparison, and the effects of scaling the overall dataset sizes across both groups of experiments could be evaluated. Moreover, since in its current form the chosen network architecture was found inadequate for replicating the target behavior well across all of the tested devices and configurations, future research should involve making informed adjustments to the model architecture in order to have the effects the training set properties have on the generalization stand out. Also, since in its current form the dataset generation and the network training are done separately, an interesting research direction could also be in combining the procedures to a single loop, allowing for the training sets to be practically infinite in variety, while still having control over the sampling densities used for the parameter generation.

References

[1] J. O. Smith, "Physical Modeling Synthesis Update," *Comput. Music J.*, vol. 20, no. 2, p. 44, 1996, ISSN: 01489267. DOI: 10.2307/3681331.

[2] V. Välimäki, F. Fontana, J. O. Smith, and U. Zolzer, "Introduction to the special issue on virtual analog audio effects and musical instruments," *IEEE Trans. Audio Speech Lang. Process.*, vol. 18, no. 4, pp. 713–714, May 2010, ISSN: 1558-7924. DOI: 10.1109/TASL.2010.2046449.

[3] A. Huovilainen, "Non-linear digital implementation of the Moog ladder filter," in *Proc. Int. Conf. Digital Audio Effects (DAFX)*, Naples, Italy, Oct. 2004, p. 4.

[4] F. Fontana and M. Civolani, "Modeling of the EMS VCS3 voltage-controlled filter as a nonlinear filter network," *IEEE Trans. Audio Speech Lang. Process.*, vol. 18, no. 4, pp. 760–772, May 2010, ISSN: 1558-7916, 1558-7924. DOI: 10.1109/TASL.2010.2046287.

[5] M. Rest, J. D. Parker, and K. J. Werner, "WDF modeling of a Korg MS-50 based non-linear diode bridge VCF," in *Proc. Int. Conf. Digital Audio Effects (DAFX)*, Edinburgh, UK, Sep. 2017, p. 7.

[6] T. Stilson and J. O. Smith, "Alias-free digital synthesis of classic analog waveforms," in *Proc. Int. Comput. Music Conf. (ICMC)*, Hong Kong, Aug. 1996, p. 12.

[7] J. Pekonen, V. Lazzarini, J. Timoney, J. Kleimola, and V. Välimäki, "Discrete-time modelling of the Moog sawtooth oscillator waveform," *EURASIP J. Adv. Signal Process.*, vol. 2011, no. 1, p. 15, Dec. 2011, ISSN: 1687-6180. DOI: 10.1155/2011/785103.

[8] L. Gabrielli, S. D'Angelo, and L. Turchet, "Analysis and emulation of early digitally-controlled oscillators based on the Walsh-Hadamard transform," in *Proc. Int. Conf. Digital Audio Effects (DAFX)*, Birmingham, UK, Sep. 2019, p. 7.

[9] J. Chowdhury, "Real-time physical modelling for analog tape machines," in *Proc. Int. Conf. Digital Audio Effects (DAFX)*, Birmingham, UK, Sep. 2019, p. 7.

[10] S. H. Hawley, B. Colburn, and S. I. Mimilakis, "SignalTrain: Profiling audio compressors with deep neural networks," May 2019. DOI: 10.48550/arXiv.1905.11928. arXiv: 1905.11928 [eess.AS].

[11] S. Bilbao and J. D. Parker, "A virtual model of spring reverberation," *IEEE Trans. Audio Speech Lang. Process.*, vol. 18, no. 4, pp. 799–808, May 2010, ISSN: 1558-7916, 1558-7924. DOI: 10.1109/TASL.2009.2031506.

[12] C. Darabundit, R. Wedelich, and P. Bischoff, "Digital Grey Box Model of the Uni-Vibe Effects Pedal," in *Proc. Int. Conf. Digital Audio Effects (DAFX)*, Birmingham, UK, Sep. 2019, p. 8.

[13] A. Wright and V. Välimäki, "Neural modeling of phaser and flanging effects," *J. Audio Eng. Soc.*, vol. 69, no. 7, p. 14, Jul. 2021.

[14] J. Pakarinen and M. Karjalainen, "Enhanced wave digital triode model for real-time tube amplifier emulation," *IEEE Trans. Audio Speech Lang. Process.*, vol. 18, no. 4, pp. 738–746, May 2010, ISSN: 1558-7916. DOI: 10.1109/TASL.2009.2033306.

[15] F. Eichas and U. Zölzer, "Gray-box modeling of guitar amplifiers," *J. Audio Eng. Soc.*, vol. 66, no. 12, pp. 1006–1015, Dec. 2018, ISSN: 15494950. DOI: 10.17743/jaes.2018.0052.

[16] A. Wright, E.-P. Damskägg, L. Juvela, and V. Välimäki, "Real-time guitar amplifier emulation with deep learning," *Appl. Sci.*, vol. 10, no. 3, p. 18, Jan. 2020.

[17] D. T. Yeh, J. S. Abel, and J. O. Smith, "Simplified physically informed models of distortion and overdrive guitar effects pedals," in *Proc. Int. Conf. Digital Audio Effects (DAFX)*, Bordeaux, France, Sep. 2007, p. 8.

[18] S. D'Angelo and V. Välimäki, "An improved virtual analog model of the Moog ladder filter," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, Vancouver, BC, Canada: IEEE, May 2013, pp. 729–733, ISBN: 978-1-4799-0356-6. DOI: 10.1109/ICASSP.2013.6637744.

[19] M. J. Kemp, "Analysis and simulation of non-linear audio processes using finite impulse responses derived at multiple impulse amplitudes," in *Proc. Audio Eng. Soc. 106th Conv.*, Munich, Germany: Audio Engineering Society, May 1999, p. 15.

[20] T. Helie, "Volterra series and state transformation for real-time simulations of audio circuits including saturations: Application to the Moog ladder filter," *IEEE Trans. Audio Speech Lang. Process.*, vol. 18, no. 4, pp. 747–759, May 2010, ISSN: 1558-7924. DOI: 10.1109/TASL.2009.2035211.

[21] F. Eichas and U. Zölzer, "Black-box modeling of distortion circuits with block-oriented models," in *Proc. Int. Conf. Digital Audio Effects (DAFX)*, Brno, Czech Republic, Sep. 2016, p. 8.

[22] A. van den Oord *et al.*, "WaveNet: A generative model for raw audio," Sep. 2016. DOI: 10.48550/arXiv.1609.03499. arXiv: 1609.03499 [cs.SD].

[23] E.-P. Damskägg, L. Juvela, E. Thuillier, and V. Välimäki, "Deep learning for tube amplifier emulation," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, Brighton, United Kingdom: IEEE, May 2019, pp. 471–475, ISBN: 978-1-4799-8131-1. DOI: 10.1109/ICASSP.2019.8682805.

[24] A. Wright, E.-P. Damskägg, and V. Välimäki, "Real-time black-box modelling with recurrent neural networks," in *Proc. Int. Conf. Digital Audio Effects (DAFX)*, Birmingham, UK, Sep. 2019, p. 8.

[25] J. Chowdhury, "A comparison of virtual analog modelling techniques for desktop and embedded implementations," p. 8, Sep. 2020. DOI: 10.48550/arXiv.2009.02833. arXiv: 2009.02833 [eess.AS].

[26] J. D. Parker, F. Esqueda, and A. Bergner, "Modelling of nonlinear state-space systems using a deep neural network," in *Proc. Int. Conf. Digital Audio Effects (DAFX)*, Birmingham, UK, Sep. 2019, p. 8.

[27] A. Peussa, "State-space virtual analogue modelling of audio circuits," Master's thesis, Aalto University, Espoo, Finland, Nov. 2020.

[28] J. Engel, L. Hantrakul, C. Gu, and A. Roberts, "DDSP: Differentiable digital signal processing," Jan. 2020. DOI: 10.48550/arXiv.2001.04643. arXiv: 2001.04643 [cs.LG].

[29] B. Kuznetsov, J. D. Parker, and F. Esqueda, "Differentiable IIR filters for machine learning applications," in *Proc. Int. Conf. Digital Audio Effects (DAFX)*, Vienna, Austria, Sep. 2020, p. 7.

[30] S. Nercessian, A. Sarroff, and K. J. Werner, "Lightweight and interpretable neural modeling of an audio distortion effect using hyperconditioned differentiable biquads," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, Toronto, ON, Canada: IEEE, Jun. 2021, pp. 890–894, ISBN: 978-1-72817-605-5. DOI: 10.1109/ICASSP39728.2021.9413996.

[31] F. Esqueda, B. Kuznetsov, and J. D. Parker, "Differentiable white-box virtual analog modeling," in *Proc. Int. Conf. Digital Audio Effects (DAFX)*, Vienna, Austria, Sep. 2021, p. 8.

[32] T. Vanhatalo *et al.*, "A review of neural network-based emulation of guitar amplifiers," *Appl. Sci.*, vol. 12, no. 12, p. 26, Jan. 2022, ISSN: 2076-3417. DOI: 10.3390/app12125894.

[33] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, ser. Adaptive Computation and Machine Learning. Cambridge, Massachusetts: The MIT Press, 2016, ISBN: 978-0-262-03561-3.

[34] P. Kidger and T. Lyons, "Universal approximation with deep narrow networks," Jun. 2020. DOI: 10.48550/arXiv.1905.08539. arXiv: 1905.08539 [cs.LG].

[35] A. Ilin, *Deep Learning - Lecture Notes of Course CS-E4890 at the Aalto University*. Espoo, Finland: Aalto University, 2022. [Online]. Available: https://mycourses.aalto.fi/course/view.php?id=32715&section=5 (visited on 06/17/2022).

[36] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997, ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735.

[37] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the Properties of Neural Machine Translation: Encoder-Decoder Approaches," Oct. 2014. DOI: 10.48550/arXiv.1409.1259. arXiv: 1409.1259 [cs.CL].

[38] P. Bhattacharya, P. Nowak, and U. Zölzer, "Optimization of cascaded parametric peak and shelving filters with backpropagation algorithm," in *Proc. Int. Conf. Digital Audio Effects (DAFX)*, Vienna, Austria, Sep. 2020, p. 8.

[39] M. A. Martínez Ramírez and J. D. Reiss, "Modeling nonlinear audio effects with end-to-end deep neural networks," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, Brighton, United Kingdom: IEEE, May 2019, pp. 171–175, ISBN: 978-1-4799-8131-1. DOI: 10.1109/ICASSP.2019.8683529.

[40] M. A. Martínez Ramírez, E. Benetos, and J. D. Reiss, "A general-purpose deep learning approach to model time varying audio effects," in *Proc. Int. Conf. Digital Audio Effects (DAFX)*, Birmingham, UK, Sep. 2019, p. 8.

[41] ——, "Deep learning for black-box modeling of audio effects," *Appl. Sci.*, vol. 10, no. 2, p. 25, Jan. 2020, ISSN: 2076-3417. DOI: 10.3390/app10020638.

[42] J. Wilczek, A. Wright, V. Välimäki, and E. Habets, "Virtual analog modeling of distortion circuits using neural ordinary differential equations," May 2022. DOI: 10.48550/arXiv.2205.01897. arXiv: 2205.01897 [eess.AS].

[43] J. D. Parker, S. J. Schlecht, R. Rabenstein, and M. Schäfer, "Physical Modeling using Recurrent Neural Networks with Fast Convolutional Layers," Apr. 2022. DOI: 10.48550/arXiv.2204.10125. arXiv: 2204.10125 [cs.SD].

[44] E.-P. Damskägg, L. Juvela, and V. Välimäki, "Real-time modeling of audio distortion circuits with deep learning," in *Proc. Sound and Music Comput. Conf. (SMC)*, Malaga, Spain, May 2019, p. 8.

[45] C. J. Steinmetz and J. D. Reiss, "Efficient neural networks for real-time analog audio effect modeling," Feb. 2021. DOI: 10.48550/arXiv.2102.06200. arXiv: 2102.06200 [eess.AS].

[46] ——, "Efficient neural networks for real-time modeling of analog dynamic range compression," Apr. 2022. DOI: 10.48550/arXiv.2102.06200. arXiv: 2102.06200 [eess.AS].

[47] T. Schmitz and J.-J. Embrechts, "Nonlinear real-time emulation of a tube amplifier with a long short term memory neural-network," in *Proc. Audio Eng. Soc. 144th Conv.*, Milan, Italy: Audio Engineering Society, May 2018, p. 6.

[48] Z. Zhang, E. Olbrych, J. Bruchalski, T. J. McCormick, and D. L. Livingston, "A vacuum-tube guitar amplifier model using long/short-term memory networks," in *Proc. SoutheastCon*, St. Petersburg, FL: IEEE, Apr. 2018, p. 5, ISBN: 978-1-5386-6133-8. DOI: 10.1109/SECON.2018.8479039.

[49] A. Wright and V. Välimäki, "Neural modelling of periodically modulated time-varying effects," in *Proc. Int. Conf. Digital Audio Effects (DAFX)*, Vienna, Austria, Sep. 2020, p. 8.

[50] E. R. Scheinerman, *Invitation to Dynamical Systems*. Upper Saddle River, N.J: Prentice Hall, 1996, ISBN: 978-0-13-185000-2.

[51] A. Peussa *et al.*, "Exposure bias and state matching in recurrent neural network virtual analog models," in *Proc. Int. Conf. Digital Audio Effects (DAFX)*, Vienna, Austria, Sep. 2021, p. 8.

[52] F. Eichas, S. Möller, and U. Zölzer, "Block-oriented modeling of distortion audio effects using iterative minimization," in *Proc. Int. Conf. Digital Audio Effects (DAFX)*, Trondheim, Norway, Nov. 2015, p. 6.

[53] T. I. Laakso, V. Välimäki, M. Karjalainen, and U. K. Laine, "Splitting the unit delay: Tools for fractional delay filter design," *IEEE Signal Process. Mag.*, vol. 13, no. 1, pp. 30–60, Jan. 1996, ISSN: 10535888. DOI: 10.1109/79.482137.

[54] J. W. Nilsson, *Electric Circuits*, 6th ed. Upper Saddle River (N.J.): Prentice-Hall, 2001, ISBN: 978-0-13-032120-6.

[55] J. O. Smith, *Physical Audio Signal Processing*. 2010. [Online]. Available: http://ccrma.stanford.edu/~jos/pasp/ (visited on 03/15/2021).

[56] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," Dec. 2019. DOI: 10.48550/arXiv.1912.01703. arXiv: 1912.01703 [cs.LG].

[57] U. Zölzer, *DAFX - Digital Audio Effects*, 2nd ed. Chichester, UK: John Wiley & Sons Ltd, 2011, ISBN: 978-0-470-66599-2.

[58] *History | MS-20 mini - MONOPHONIC SYNTHESIZER | KORG (USA)*. [Online]. Available: https://www.korg.com/us/products/synthesizers/ms_20mini/page_3.php (visited on 06/01/2022).

[59] T. E. Stinchcombe, "A study of the Korg MS10 & MS20 filters," Tech. Rep., Aug. 2006, p. 46.

[60] *ElectroSmash - ProCo Rat analysis*. [Online]. Available: https://www.electrosmash.com/proco-rat (visited on 06/17/2022).

[61] D. T. Yeh, "Digital implementation of musical distortion circuits by analysis and simulation," PhD thesis, Stanford University, Stanford, US, Jun. 2009.

[62] E. Barbour, "The cool sound of tubes [vacuum tube musical applications]," *IEEE Spectr.*, vol. 35, no. 8, pp. 24–35, Aug. 1998, ISSN: 1939-9340. DOI: 10.1109/6.708439.

[63] R. Cauduro Dias de Paiva, J. Pakarinen, V. Välimäki, and M. Tikander, "Real-time audio transformer emulation for virtual tube amplifiers," *EURASIP J. Adv. Signal Process.*, vol. 2011, no. 1, p. 15, Dec. 2011, ISSN: 1687-6180. DOI: 10.1155/2011/347645.

[64] *Do-A-Pultec page*. [Online]. Available: https://www.gyraf.dk/gy_pd/pultec/pultech.gif (visited on 01/07/2022).

[65] *LTspice simulator*, Analog Devices. [Online]. Available: https://www.analog.com/en/design-center/design-tools-and-calculators/ltspice-simulator.html (visited on 06/17/2022).

[66] N. Brum, *Nunobrum/PyLTSpice*, Jun. 2022. [Online]. Available: https://github.com/nunobrum/PyLTSpice (visited on 05/19/2022).

[67] B. Holmes and M. van Walstijn, "Potentiometer law modelling and identification for application in physics-based virtual analogue circuits," in *Proc. Int. Conf. Digital Audio Effects (DAFX)*, Birmingham, UK, Sep. 2019, p. 8.

[68] C. Kehling, J. Abeßer, C. Dittmar, and G. Schuller, "Automatic tablature transcription of electric guitar recordings by estimation of score- and instrument-related parameters," in *Proc. Int. Conf. Digital Audio Effects (DAFX)*, Erlangen, Germany, Sep. 2014, p. 8.

[69] J. Abeer, P. Kramer, C. Dittmar, and G. Schuller, "Parametric audio coding of bass guitar recordings using a tuned physical modeling algorithm," in *Proc. Int. Conf. Digital Audio Effects (DAFX)*, Maynooth, Ireland, Sep. 2013, p. 8.

[70] A. Wright and V. Välimäki, "Perceptual loss function for neural modeling of audio systems," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process. (ICASSP)*, Barcelona, Spain: IEEE, May 2020, pp. 251–255. DOI: 10.1109/ICASSP40776.2020.9052944.

[71] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," Jan. 2017. DOI: 10.48550/arXiv.1412.6980. arXiv: 1412.6980 [cs.LG].

[72] "BS.1534 : method for the subjective assessment of intermediate quality level of audio systems," International Telecommunication Union, Recommendation BS.1534, Oct. 2015. [Online]. Available: https://www.itu.int/rec/R-REC-BS.1534/en (visited on 06/08/2022).

[73] M. Schoeffler *et al.*, "webMUSHRA — a comprehensive framework for web-based listening tests," *J. Open Res. Softw.*, vol. 6, no. 1, p. 8, Feb. 2018, ISSN: 2049-9647. DOI: 10.5334/jors.187.

[74] C. J. Steinmetz, *Pyloudnorm*, May 2022. [Online]. Available: https://github.com/csteinmetz1/pyloudnorm (visited on 05/25/2022).

[75] "BS.1770 : algorithms to measure audio programme loudness and true-peak audio level," International Telecommunication Union, Recommendation BS.1770, Oct. 2015. [Online]. Available: https://www.itu.int/rec/R-REC-BS.1770 (visited on 06/08/2022).

[76] *MATLAB tinv - student's t inverse cumulative distribution function*. [Online]. Available: https://se.mathworks.com/help/stats/tinv.html (visited on 06/15/2022).