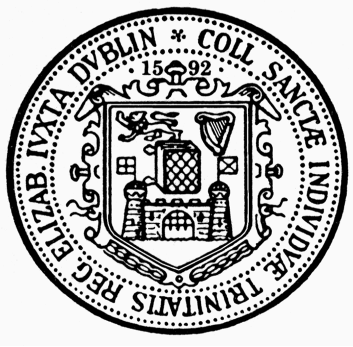
Black & White: Comparative Analysis of Virtual Analog Modeling Methods for Real-Time Audio

Thomas Garvey

Music & Media Technologies
School of Engineering
Trinity College Dublin



Submitted as part fulfilment for the degree of M.Phil.
Supervisor: Dr. Hugh O'Dwyer

2025

Declaration

I hereby declare that this thesis has not been submitted as an exercise for a degree at this or any other University and that it is entirely my own work.

I agree that the Library may lend or copy this thesis upon request.

Signed: _____ Date: August 12, 2025

Thomas Garvey

Acknowledgements

To Mom, Dad, Ciara, and Brian for your unwavering support. We made it! Also to my official and unofficial supervisors, Hugh, Jatin, and Eric for taking the time to answer my questions, no matter how ridiculous.

Table of Content

List of Figures

Abstract

Virtual analog modeling is the process of creating software versions of hardware. It is important in the audio industry as developers seek to preserve and digitize difficult to obtain audio effects. Among them, distortion effects offer unique challenges for emulation because of their nonlinear components. While more traditional white box circuit modeling works well, black box techniques, including Deep Learning, have grown in popularity in recent years for their fidelity and relative ease of use. Many plugin companies have adopted this practice and are very vocal about their use. However, does this new technique actually create more accurate emulations compared with component-based models? This thesis found that Deep Learning models were marginally more accurate in emulating a Tube Screamer TS9 pedal than the white box method using subjective listening and playing tests. However, comparison between methods proves difficult because of the inherent benefits and pitfalls of each.

1 Introduction

As a musician, what is the best way to steal someone's sound? For better or worse, one of the most common ways is to use the same gear as the original artist. However, this imitation is easier said than done. The Klon Centaur overdrive pedal, for example, has been made famous by artists such as Wilco, John Mayer, and Radiohead, but original models are now vintage and worth thousands of dollars. So, how can the average musician get their hands on the signature tone it creates? One solution is to create a digital version of it.

Virtual analog modeling is the practice of turning hardware into software. Creating these emulations is important for music and audio technology because many desirable pieces of hardware are increasingly difficult to obtain [Yeh, 2009]. Often, analog modeling offers a faithful emulation of this hardware in a less expensive digital format, which increases the access people have to these sounds – these people include musicians, producers, sound technicians, and most anyone involved in audio [Lambert, 2010]. However, while there are many methods that can be used to create these digital representations, each has unique benefits and drawbacks.

Recently new emulation techniques using Deep Learning have grown in popularity, following the rise in Artificial Intelligence development. While machine learning is not necessarily new to audio, Deep Learning has proven effective at emulating nonlinear components, which have previously been difficult to model with traditional techniques [Karjalainen & Pakarinen, 2006]. Many audio effects companies are being very vocal about its use, but are they actually creating a better emulation or is it just new and exciting?

1.1 Context

Distortion is an audio effect discovered by early rock-n-roll pioneers who played too loudly through broken guitar amplifiers [Poss, 1998]. Since then, the gritty, fuzzy sound has been explored by a wide variety of artists and countless different types of distortions have been developed. While each is slightly different, they nearly all work by clipping or limiting the audio signal and boosting harmonics, especially in the higher end of the frequency spectrum through

nonlinear components [Herbst, 2018]. One famous distortion is the Ibanez Tube Screamer. Originally developed in the 1970's, it has gone through many iterations and rereleases, though its core distortion component remains a diode. Its fuzziness has been praised by artists like Stevie Ray Vaughan, U2, and Keith Richards to name just a few. However, original Tube Screamers can go for thousands of dollars, making it ripe for emulation.

There are three main umbrellas under which emulation techniques for distortion typically fall: white, black, and gray box. First is white box modeling, wherein the entire system is known [Wright et. al, 2020]. These are the more traditional emulation techniques and typically include a direct mathematical representation of the hardware's circuits and components [Chowdhury, 2020]. These methods involve detailed analysis of a hardware's schematics to understand how a signal is processed as it passes through each component [Dempwolf, 2018]. They are often very high fidelity, especially for linear systems, but can become computationally expensive for highly non-linear and complex ones [Mačák, 2012]. Still, these methods are praised for how granular the developer can be with the specific circuit they are modeling – they are able to easily tweak individual parameters because the entire system is known and available to them [Ramírez & Weiss, 2019].

Black box techniques are the opposite – the system remains unknown [Wright et al, 2020]. Black box models have an input and output, but the developer cannot directly tweak the components that turn one into the other. These techniques include the Volterra Series, Weiner models [Südholt et al, 2022], and dynamic convolution, like Kemper guitars, which started as a simple snapshot of a system with one specific, unchangeable set of parameters [Düvel et al, 2020]. Recently, techniques involving Deep Learning through neural networks have grown in popularity [Ramírez et al, 2020]. Using these networks, it is theoretically possible to emulate entire signal chains with variable parameters, even for highly non-linear systems [Juvela et al, 2024]. However, pitfalls still exist. Neural Networks are more accurate when trained on larger amounts of data, but collecting the dataset can be a laborious experience [Damskägg et al, 2019a]. Furthermore, while advances in GPU technology and Deep Learning libraries like Keras and Pytorch have optimized network training, it can still be time consuming and computationally expensive [Wright et al, 2020].

Lastly, gray box modeling is often a combination of these two methods, or simply another emulation technique all together. These methods require the developer to work almost as an artist

in lieu of following a procedure like in white or black box modeling [Eichas et al, 2017]. While intriguing, gray box modeling will not be further discussed in this thesis. Instead, only pure white and black box models will be considered.

Figure 1.1: Neural DSP Quad Cortex, which is a hardware device that uses neural networks to capture and model guitar amp sounds [Neural DSP, 2024] .

1.2 Objectives

In recent years, Deep Learning seems to be everywhere in virtual analog modeling. So much so that companies like Neural DSP have developed hardware that uses inbuilt neural networks to train and replace entire signal chains (Fig 1.1). There is speculation that digital versions could eventually replace all hardware [Giampiccolo et al, 2025]. While perhaps the pipedream of a weary sound technician, this speculation opens up many questions, like which emulation technique offers higher fidelity to the original? And what are the real time implications for these methods? And how willing are artists to accept this digitization? Ultimately however, this thesis sought to answer one question: are white or black-box methods more accurate in emulating guitar distortion for real time?

1.3 Methodology

To answer this question, two VST plugins were made to model a Ibanez Tube Screamer TS9 pedal using both white and black box emulation. This pedal was chosen for both its desirable distortion sound and the high non-linearity of its circuit, which makes accurately emulating it difficult [Electrosmash, 2025]. Recordings were then made using both plugins and the pedal for listening and playing tests. In the latter, local guitar players compared the feel of playing on the three distortions to see which virtual tool has the higher fidelity. Answering if white or black box yielded more accurate results would help determine if neural-based music tech companies actually create a better product for their customers.

1.4 Thesis Overview

This thesis first discusses circuit fundamentals and various white box methods commonly used for virtual analog modeling of audio circuits. Then basic concepts related to neural networks, especially regarding Recurrent Neural Networks, are discussed before an in depth review of the current state of virtual analog modeling is presented. Next, the thesis' methodology is described before a detailed look at its implementation is given for both the white box, black box, and testing designs. Finally the testing results are evaluated and a conclusion is given.

2 Background

2.1 White Box modeling

2.1.1 Overview

White box modeling is a set of emulation techniques that often describe a 1:1 representation of the hardware schematics into code [Esqueda et al, 2021]. Often called component level modeling, they are praised for their accuracy and sonic fidelity [Li, 2022].

However, each of the methods in the white box umbrella require a software engineer with a specialty in audio digital signal processing (DSP) and electrical engineering. They also need access to accurate schematics or the ability to recreate the system from the hardware itself. Most of these white box techniques rely on developing a set of differential equations through circuit analysis before applying a Laplace transform along with a discretionary step [Tarr, 2022]. While hybrid models using multiple methods can be useful, each technique must be chosen depending on the desired end result [Vanhatalo et al, 2022].

2.1.2 Circuit Fundamentals

A circuit can be defined as a group of connected components that carry and pass electricity [Hayt, 2024]. There are many different types of components, all of which react to and shape electricity differently, but every circuit can be understood by its voltage, current, and resistance. Voltage is the difference in electrical charge between two points in the circuit; current is the rate at which the charge flows; and resistance is the impedance, or how a component resists the current [Mohan, 2011]. Many audio circuits take an input, like from an electric guitar, and shape it into an output depending on how the different components are laid out. There are, however, many different white box techniques for understanding this relationship between input and output.

Sticking with the basics, there are two key equations that represent the relationship between current, voltage, and impedance/resistance. First is Ohm's Law, which can be seen in Equation 1, where current (i) is equal to voltage (v) divided by impedance (Z). Note that across a component, the voltage would actually be the voltage drop (i.e, the change in voltage on either side of the component). Also, impedance is component-specific both in value and equation, but is typically written as just R for resistance.

$$i = \frac{v}{Z} \tag{1}$$

The second key equation is Kirchoff's Current Law (KCL). According to KCL, which can be seen in Equation 2a, the current going into a node must equal the current going out of that

node. Furthermore, the current across the entire system must remain equal, unless another current source is added. The overall current of any node can also be given as the sum of all currents going into it (Equation 2b). Likewise, the current leaving a node is equal to all the currents leaving it combined.

$$i_{in} = i_{out} \tag{2a.}$$

$$i_{in} = i_1 + i_2 \tag{2b.}$$

2.1.3 Continuous vs. Discrete Domains

One key problem with digital hardware emulation is the transform from a continuous signal to a discrete representation. Voltage in a circuit is continuous, meaning that an infinite amount of points can be taken to represent it [Smith, 2010]. More generally, a continuous system is one that changes continuously over time. Unfortunately, modern computers cannot store an infinite amount of data, so they instead take in just enough discrete points to represent the continuous signal.

There are many discretization methods that can be employed, but the most common one in audio hardware emulation is the Bilinear Transform, often called Trapezoidal Integration [Tarr, 2021]. Each method seeks to approximate the value of a function's integral using a finite number of evaluations, just in different ways [Germain & Werner, 2017]. Other methods include the Midpoint Method and Backwards Eulers, but the Bilinear Transform is often used for its frequency accuracy and system stability [Simper, 2013].

2.1.4 Method 1: Nodal Analysis

Nodal analysis is the process by which Ohm's Law and KCL are used to create a system of matrix equations that represents the relationship between the input and output of a circuit [Tarr, 2021]. These equations are created by analyzing intermediary nodes, which are defined as the space between components [Hayt, 2024]. Once KCL-based equations have been developed for each node, they are transformed into the s-domain via a Laplace Transform using the Laplace variable $s = j\omega$. The Laplace Transform can be thought of as a way to turn differential equations

into ones that can be solved algebraically [Pieter, 2024]. While a continuous transfer function can be useful for frequency analysis, it is really just a stepping stone here – a final transform into the z-domain is necessary for discrete digitization of the system [Chowdhury, 2020]. To read more on the Bilinear and Laplace Transforms, see [Pieter, 2024].

While the various transforms can be a bit tedious algebraically, nodal analysis is a relatively straightforward way to solve a circuit. However, there are a few drawbacks. First, nodal analysis cannot be used to solve for nonlinear systems. However, Modified Nodal Analysis (MNA) includes additional equations to solve for nonlinearities [Vanhatalo et al, 2022]. Still when actually implementing either in a digital framework, major parts of the system have to be recomputed if any of the components change values, which can prove disastrous for large, complex systems [Chowdhury, 2020]. Notably, MNA is used in part for Simulation Program with Integrated Circuit Emphasis (SPICE) simulations, which are incredibly accurate, but nowhere near real time safe.

2.1.5 Method 2: State Space

The state space method is a way to represent systems that extend well beyond just circuits and is commonly found in engineering of all types. However, for the purpose of virtual analog modeling, it is a representation of the current circuit parameters using the memory, or previous state, of the system [Tarr, 2021]. The general form for nonlinear state space equations can be seen in the Ordinary Differential Equations (ODE) of Equation 3.

$$\frac{dx}{dt} = Ax + Bu + Ci(v)$$
$$y = Dx + Eu + Fi(v)$$
$$v = Gx + Hu + Ki(v)$$

3.

Where x is the state variable vector, u is the input vector, y is the output, and i is the nonlinear function dependent on parameter v. A, B, C, D, E, F, G, H, and K all represent matrix coefficients that affect their corresponding variables [Dempwolf & Zolzer, 2011]. All the past states, which are from stateful components like capacitors, are summed with the input history

within the state variable, x. This method also creates a continuous representation of the system and requires a discretionary step to be computer-ready.

2.1.6 Method 3: Wave Digital Filters

Wave Digital Filters (WDF) are a bit of a black sheep within the white box family. Rather than using current and voltage, it uses an incident wave that passes through the circuit components [Werner et al, 2015]. Each component is considered a port containing a characteristic resistance [Chowdhury, 2020]. These ports exist within the wave domain and are modular in that they can be connected by series and parallel adaptors – their connection is often called a WDF tree.

This modularity is their key advantage. Each component can be treated completely differently and still remain in its position within the WDF tree. For example, each component can be discretized using different methods (i.e., Bilinear Transform, Euler's Method, etc.). Because the components are so independent, little recomputation is required if any value changes within a digital framework [Yeh & Smith, 2008]. However, this method struggles with highly complex and nonlinear systems [Chowdhury, 2020].

2.1.7 Method 4: DK-Method

Another more modern white box approach is the Discrete-Kirchoff method (DK-Method). It is an extension of MNA, but significantly reduces the complexity by substituting discrete time equivalent equations for each component directly into the KCL equations [Eichas et al, 2014]. Note that using both matrix notation along with ODE's or creating a system of equations via nodal substitution is valid [Tarr, 2021]. Basically, rather than go through various s and z-domain transforms, the schematic itself is rewritten in discrete time [Vanhatalo et al, 2022]. Furthermore, since the schematic is turned into its discrete time equivalent, it is already digitized and can easily be implemented as an audio plugin in a digital audio workspace (DAW), for example. To understand how the DK-Method is implemented, it is necessary to take a more detailed look at individual components and their discrete-time representations.

2.2 Components in the DK-Method

2.2.1 Linear Components

Linear components in the DK-Method are more straightforward to solve than nonlinear ones, but each comes with their own quirks that demand special attention. For example, currents across resistors require a direct application of Ohm's law, while currents across capacitors require both a discretionary integration and state variable update. It is necessary to understand how each component is implemented before completing nodal analysis via Kirchoff's Current Law (KCL).

Resistors

Figure 2.1: The American schematic symbol for a resistor [ElProcus, 2025].

The simplest component to model is a resistor. The current equation for an idealized resistor can be given by Ohm's Law (Equation 1) where i is the current across the resistor, v is the voltage drop, and R is the impedance value.

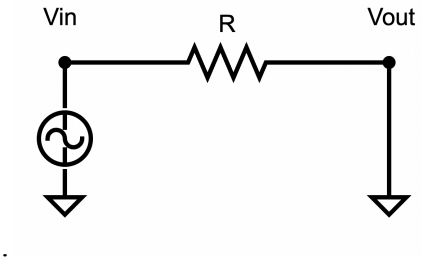


Figure 2.2: A simple circuit with one resistor.

In the case of a resistor, the voltage drop is the difference between voltages on either side of it and the impedance is just the resistance value. Substituting this into Ohm's law can be seen in Equation 5.

$$i_r = \frac{v_{in} - v_{out}}{R} \quad (5.)$$

Potentiometers

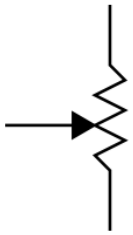


Figure 2.3: The schematic symbol for a potentiometer [Random Nerd Tutorials, 2025].

Potentiometers, or pots, can be thought of simply as resistors in series, where each sub-resistor has an inversely related scalar attached to it (Fig. 2.4). In hardware, they are often knobs that control the level of a given parameter. This scalar, alpha, is a number between 0.0 and 1.0, which can be thought of as the knob level itself [Tarr, 2021]. It also determines how much resistance is actually being applied to the voltage, meaning that the resistance is variable [Irwin & Elms, 2015]. Note that the total resistance of the system is not altered, but instead spread over the two resistors.

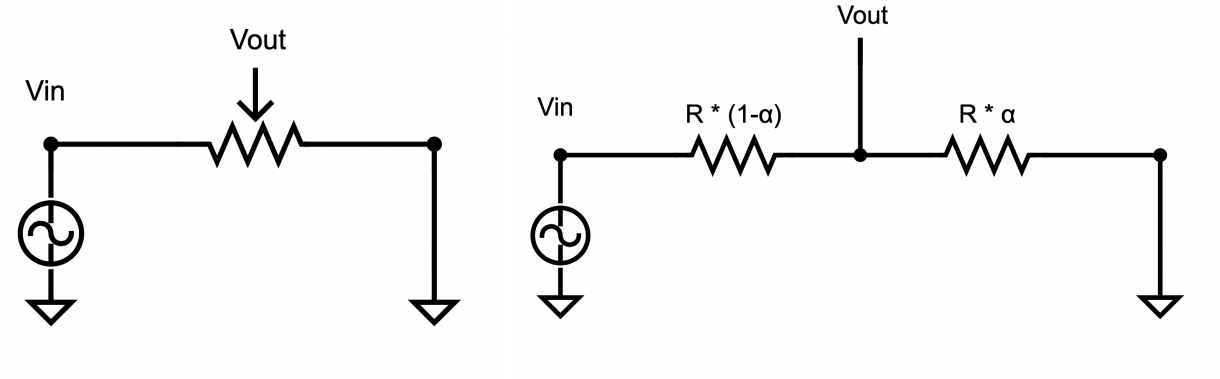


Figure 2.4: A simple schematic with a potentiometer written typically (left) and split up as variable, alpha-dependent resistors (right).

Op Amps

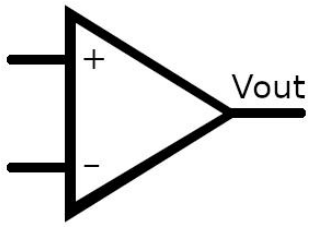


Figure 2.5: The schematic symbol for an operational amplifier [Stewart-Frazier Tools, 2020].

Operational amplifiers, or op amps, are crucial for many audio circuits, especially when amplification is involved. Unlike resistors, potentiometers, or other passive components, op amps are considered active, meaning that they require an additional voltage source [Irwin & Elms, 2015]. Op amps are made up of various subcircuits, but are typically thought of as one unique, idealized component. Just the conceptual behaviour of how it handles inputs and outputs are sufficient for this thesis – the additional voltage source is not considered and the op amp is ideal.

An ideal op amp will take the difference in voltage between its positive and negative terminals and amplify it [Yildiz & Kelebekler, 2013]. Open loop op amps are rare and instead, the output of the component is often fed back into one of the terminal inputs [Karki, 2021]. An ideal feedback loop will remove any voltage difference between the two terminals, meaning that their voltage would be identical [Pandey, 2024]. Pairing this feedback loop with the assumption that there exists an open loop between the terminals (i.e., the internal resistance between them is infinite), it can be stated that no current flows into an ideal op amp. So in summary, the two key assumptions are that,

1. The voltage is the same at each terminal.

2. No current flows into the component.

Applying these assumptions to the op amp schematic in Fig. 2.6, two statements can be made. First, since no current flows through the op amp, all of the current must flow through R2. Second, since the positive terminal goes to ground (where the voltage is zero), then the voltage at the negative terminal is also zero. For simplicity's sake, both terminals can be considered one node (Vx) in nodal analysis.

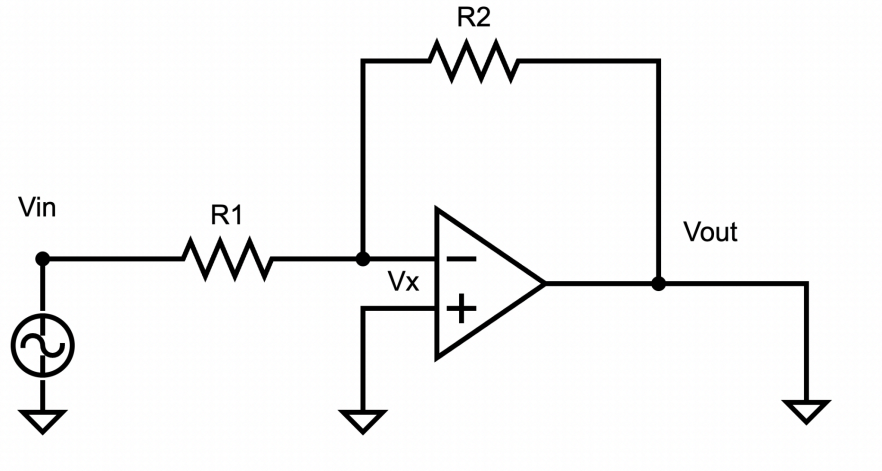


Figure 2.6: A simple schematic for an inverting op amp.

Using nodal analysis, the circuit in Fig. 2.6 can be solved in Equation 6.

$$i_{in} = \frac{v_{in} - 0}{R_1} = \frac{0 - v_{out}}{R_2} = i_{out}$$

6.

Capacitors

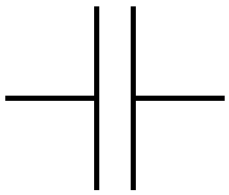


Figure 2.7: The schematic symbol for a capacitor [Ersa Electronics, 2024].

Perhaps the most important component for the DK-Method and much of circuit emulation as a whole is the capacitor. Capacitors, and inductors, which were not used in this thesis and will not be covered, are unlike other components because the current flowing through them is proportional to the rate of change of their voltage [Yildiz & Kelebekler, 2013]. Capacitors need to be 'filled' or 'warmed up' and when the energy flow across it ceases, they may release their charge slowly [Irwin & Elms, 2015]. This ability allows capacitors to retain a memory of the overall system – capacitors are stateful. Understanding how a capacitor's voltage changes over time allows the developer to understand how the system as a whole changes over time [Tarr, 2021]. In the DK-Method, capacitors can be implemented through two equations (Equation 15) that abstract the change in rate of a capacitor's voltage to include Bilinear Integration to discretize it and update its state [Yeh et al, 2010].

To understand the derivation of these DK-Method capacitor equations, it is important to know how to discretize regular nodal analysis equations without them. It starts with Ohm's law (Equation 1). For capacitors, impedance can be described as the components elastance, or the inverse of the capacitance (C) [Thompson, 2014]. Furthermore, remember that the current across a capacitor is dependent on the *rate of change of voltage* instead of the voltage itself (Equation 7) – it will be different if the capacitor is full or empty, for example.

$$Ohm's\ Law \rightarrow v = i \cdot Z$$
$$\frac{dv}{dt} = i \cdot \frac{1}{C}$$

7.

While a good start, this equation still needs to be transferred into the discrete domain. To better handle the included derivative, first it undergoes a Laplace transform, which can be thought of as the Fourier Transform continuous equivalent. For the purposes of this thesis, this updated continuous domain equation acts as a stepping stone into the Bilinear Transform, but can be used to study the frequency spectrum of capacitor filters. Moving on, Equation 7 can be written using integral notation to better visualize the transition into the s-domain form (Equation 8).

$$v(t) = \frac{1}{C} \cdot \int i(t) \cdot dt$$

8.

Where t equals time. Then, it undergoes a Laplace Transform (Equation 9):

$$v(s) = \frac{1}{sC} \cdot I(s)$$

9

Where s is a complex variable of the s-domain. This equation seems complex, but is really just Ohm's Law, yet again! Applying a z-transform from here is very simple. Specifically the Bilinear Transform, or Trapezoidal Integration, is used because of its stability, but it is certainly not the only transform available to do so [Simper, 2020]. Still, the process is as simple as swapping all instances of 's' in the s-domain with Equation 10, where T is the timestep and z is the complex variable of the discrete-time (z) domain. With this substitution, the KCL equations have been discretized and after some basic algebra, can be used computationally.

$$s \approx \frac{2}{T}\left(\frac{z-1}{z+1}\right)$$
$$z = e^{sT}$$

10.

This process of schematic to Laplace (s) Transform to Bilinear (z) Transform to usable differential equations is accurate, but also incredibly cumbersome and sometimes overwhelming. Instead, the DK-Method drastically simplifies it. Basically, the capacitor equation used in nodal analysis can be substituted immediately with its discretized form. First, returning to the change in voltage of a capacitor, remember that it can be represented three ways: derivative form, integral form, and as a change from one point in time to another (Equation 11).

$$\frac{dv}{dt} = i \cdot \frac{1}{C}$$

$$v(t) = \frac{1}{C} \cdot \int i(t)\, dt$$

$$\frac{dv(t)}{dt} = v[n] - v[n-1]$$

11.

Then, just apply the Bilinear Transform to the integral form! Note that the numerical integration of the time-domain application of the Bilinear Transform is just called the Trapezoidal Rule. In general, the Trapezoidal Rule approximates the area under a curve using a summed series of trapezoids – in other words, it approximates the integral of a given function…like the integral form above, for example. In Equation 12, the Trapezoidal Rule is implemented, where v[n] and i[n] are discrete samples of the voltage and current across the capacitor at time n.

$$v(t) = \frac{1}{C} \cdot \int i(t)\, dt$$

$$v[n] - v[n-1] = \frac{T}{C}\left(\frac{i[n]}{2} + \frac{i[n+1]}{2}\right)$$

12.

Equation 12 can be rewritten in a more familiar KCL format as seen in equation 13.

$$i[n] = \frac{2C}{T}(v[n]) - \frac{2C}{T}(v[n-1]) - i[n-1]$$

13.

Because the back half of that equation is concerned with information from past samples, it can be used to define the 'state' of the past. For clarity's sake, this equation can be broken up into current and past states as seen in Equation 14.

$$i[n] = \frac{v[n]}{Z} - x[n-1]$$
$$x[n-1] = \frac{v[n-1]}{Z} - i[n-1]$$
$$Z = \frac{T}{2C}$$

14.

Equation 14 can be taken one step further by first defining the current state, not the state of one sample in the past – x[n], instead of x[n-1]. Then, i[n] can be substituted and simplified, providing the final capacitor equations for the DK-Method in Equation 15.

$$i[n] = \frac{v[n]}{Z} - x[n-1]$$
$$x[n] = \frac{2}{Z}(v[n]) - x[n-1]$$

15.

Wonderful! These are the exact equations that can be used while performing KCL directly from the schematics without having to perform any extra transforms – they are currently derived from the Bilinear Transform and are already discrete, meaning that they are perfectly suited for standard DSP! Note that x[0] is initialized as 0.

Lastly, capacitors are uniquely suited for basic audio signal processing tasks for their inherent frequency filtering capabilities [Anderson, 2020]. They can be considered frequency dependent where low frequencies are met with an open circuit (infinite resistance) and high frequencies are met with a short circuit (no resistance) [Smith, 2010]. This ability is why a

simple RC circuit is a basic low pass filter – Fig 2.8 is an example of how to use the DK-Method to solve for such a filter.

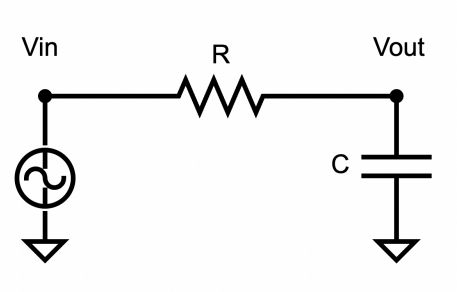


Figure 2.8: RC Circuit.

Using KCL, this circuit can be represented as:

$$i_{in} = i_{out}$$

Now applying the component equations that solve for the current of each component, the below can be substituted:

$$i_{in} = \frac{V_{in} - V_{out}}{R} = \frac{(V_{out} - 0)}{Z} - x[n-1] = i_{out}$$

Solve for Vout:

$$V_{out} = V_{in}\left(\frac{R}{R+Z}\right) + \left(\frac{RZ}{R+Z}\right) \cdot x[n-1]$$

The state variable gets updated as follows:

$$x[n] = \frac{2}{Z}(V_{out} - 0) - x[n-1]$$

$$Z = \frac{T}{2C}$$

Lastly, it may be helpful to consider the capacitor equation as composed of two separate parts. First, there is the present current across the component, which is represented by Ohm's Law (Equation 16). This section is the current at this given sample or moment as a function of voltage drop and impedance.

$$i[n] = \frac{v[n]}{Z} \quad 16.$$

However, this current also draws from the past states ($x[n-1]$), which can be treated like a current source going in the opposing direction, hence why it is subtracted in Equation 15 [Tarr, 2021]. So, the schematic for each capacitor can be rewritten as the combination of a resistor (or general impedance) and an opposing current source, to capture its relationship to both the present and the past states (Fig 2.9). This change is mostly stylistic, but can be helpful when analyzing larger circuits with many capacitors and state variables to keep track of.

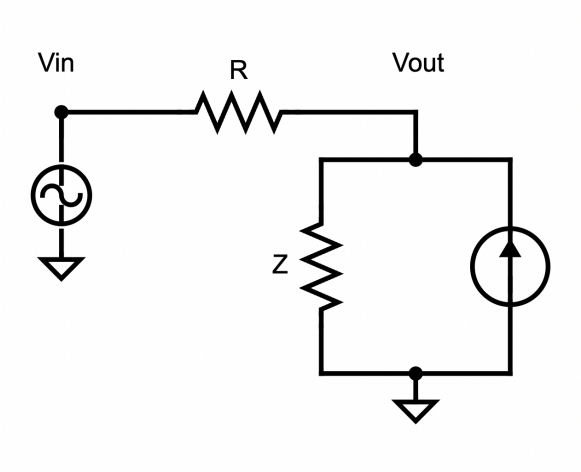


Figure 2.9: An RC circuit rewritten for the DK-Method.

2.2.2 Non-Linear Components

Non-linearity is crucial for distortion effects [Saber, 2020]. While clipping distortion can be made using just a diode, more complex distortions utilize vacuum tubes, triodes, bipolar junction transistors, and other non-linear components [Yeh, 2009]. However, only diodes are used in the Tube Screamer schematic, so only this component will be discussed.

Non-linear components also offer a unique problem with the DK-Method. Their non-linearity can make it incredibly difficult to solve the nodal equations to produce an output as a function of the input. Instead, a Newton-Raphson solver, which uses iterative guess-and-check through the equation's derivative, is typically used [Bernadini et al, 2021].

Diodes

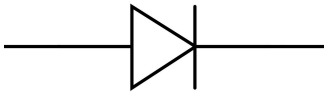


Figure 2.10: A typical diode schematic symbol [Next PCB, 2025].

At their most basic, diodes are non-linear components that allow the flow of current in one direction only [Thurston, 1960]. While they can be made of either Silicon or Germanium, the current across any diode can be expressed through the Shockley Diode Equation (Equation 17).

$$i_{diode} = Is(e^{\frac{V}{\eta V_t}} - 1) \quad (17)$$

Where *Is* equals the saturation current, *Vt* is the thermal voltage, and η is the emission coefficient. Note that the values for these variables depend on whether the diode is Silicon or Germanium-based [Smith, 2010]. This equation can be used within any KCL format for the current across a diode. For an example implementation, take the simple circuit in Fig 2.11:

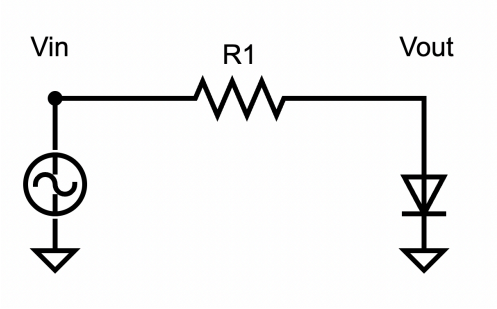


Figure 2.11: A simple diode circuit schematic.

Using KCL, this circuit can be represented as:

$$i_{in} = i_{out}$$

Where, i_{in} is the current going into the Vout node, which is the current across the resistor, and i_{out} is the current leaving the Vout node, which is the current across the diode. So, the KCL equation can be rewritten as follows:

$$i_{in} = \frac{V_{in} - V_{out}}{R1} = Is(e^{\frac{V_{out} - 0}{\eta V_t}} - 1) = i_{out}$$

This equation is 'implicit' – it is very difficult to isolate Vout and set it equal to the rest of the equation because it exists in the numerator of the exponent [Tarr, 2021]. Because it is implicit, it does not contain an analytical solution. Instead, a numerical solution, like the Newton-Raphson method, must be employed.

2.3 Black Box modeling

2.3.1 Overview

Black box modeling is an umbrella term used to describe emulation techniques where the system is unknown [Comunitá, 2025]. These techniques are agnostic to the circuit components and include the Volterra Series, Weiner models, and Deep Learning. The math and concepts behind Deep Learning have been around for decades, but increases in computational efficiency have only recently allowed for their practical use [Bognar, 2022]. Deep Learning is a type of machine learning algorithm that focuses on using neural networks to extract features and patterns from a dataset [Ramírez, 2020]. For the extent of this thesis, black box modeling will refer to only Deep Learning.

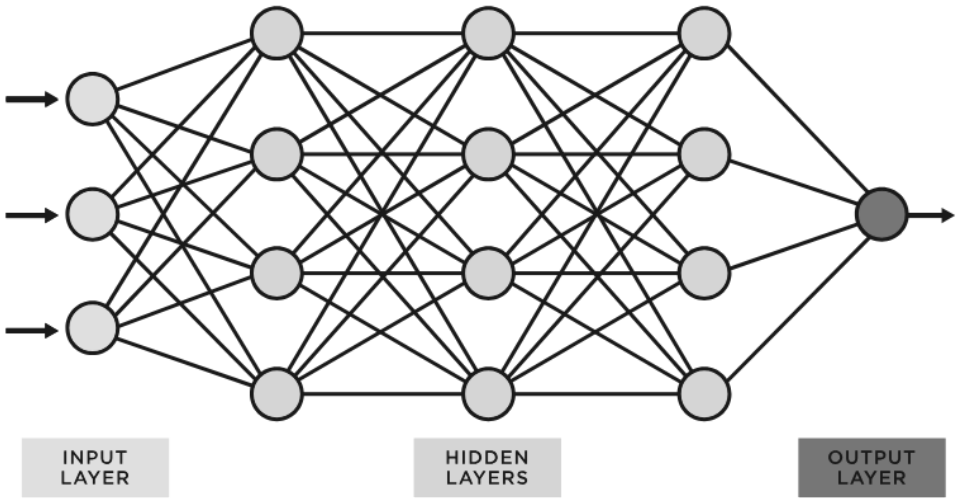


Figure 2.12: The general outline of a neural network. The input layer may include multiple neurons that are connected to each hidden layer through weights and biases. Each hidden layer also may have a unique activation function [Spotfire, 2025].

2.3.2 Neural Networks

A neural network is a machine learning algorithm made up of connected neurons and activation functions. These neurons and functions are organized into layers, called hidden layers.

As an input passes through them, it is multiplied by a vector of weights and bias. Then the activation function is applied and it is sent to the next layer until an output, or prediction, is made [Vanhatalo et al, 2022]. This process of passing an input through these layers to make a prediction is called forward propagation [Kelleher, 2019]. That prediction is then compared against the expected result and the system weights are updated according to that comparison, or loss, using an optimization task based on stochastic gradient descent. The weights are updated starting from the last layer in a process called back propagation [Bengio et al, 2015]. The system is run again and again, with each iteration called an epoch, until there are no more epochs or the loss has been minimized to the desired amount. In general, the smaller the loss value, the closer the prediction and the target are – the gradual reduction in loss over epochs is called convergence [Bengio et al, 2015].

This overview is extremely high level and each neural network's architecture and design, or hyperparameters, are variable and should be determined by both the type of data that is being trained and the desired output [Kelleher, 2019]. In this thesis, the training data is audio, so many audio-specific considerations need to be made. First, audio is a progression of sound through time, so some sort of temporal or memory aspect is needed. Second, audio is sequential. The amount of data that represents an audio signal can be given by its sample rate, which is the amount of samples taken every second. That means that there are 44,100 data points per second of audio for a sample rate of equal length. If a neural network is training on even a minute of audio, that is an incredible amount of data to process! Because audio is sequential, the order of these samples matters. Third, what is the purpose of this neural network? If real time prediction is necessary, then the model architecture must be as lightweight as possible to allow for rapid inferencing [Juvela et al, 2024]. Fourth, the spectral content of the training data may need to be considered [Wright et al, 2020]. Distortion, for example, contains more high end spectral content than low end, so a bias for the high end might be needed. Lastly, what would be an appropriate loss function for audio to determine how well the prediction matches the expected output?

2.3.3 Recurrent Neural Networks (RNN)

RNN's are stateful neural networks that train on sequential data and contain memory [Yu et al, 2019]. One simple example of an RNN in practice is a network that predicts the next word

in a sentence – all of the previous words and their order matter for predicting what comes next. The same concept can be applied to a piece of audio – the previous samples can be used to predict the next one.

As the name would suggest, an RNN also relies on a feedback loop for recursion. Conceptually, RNN's can be likened to a regular forward propagation network that processes each time step sequentially using the same parameters and weights for each [Sherstinsky, 2020]. They are often unrolled to better visualize them at each step (Fig. 2.13).

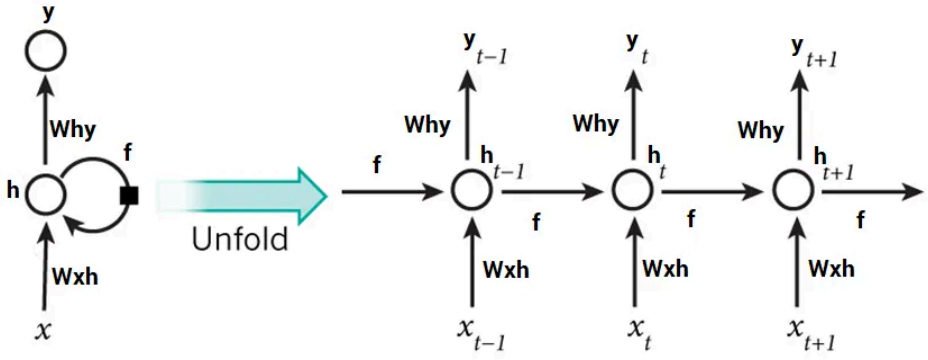


Figure 2.13: An unfolded recurrent neural network [T.J.J, 2020].

The most lightweight RNN model, the Gated Recurrent Unit (GRU), has proven to be successful at replicating distortion, but research suggests that a single Long, Short Term Memory (LSTM) layer with between 32 and 96 hidden units into a fully connected dense layer is sufficient [Wright et al., 2020; Chowdhury, 2020]. Further research also suggests that this architecture is suitable for end-to-end modeling, which is the modeling of discrete switches or continuous knobs, which are often found in audio hardware [Juvela et al, 2024].

A dense layer is a fully connected, non-recursive hidden layer much more common in neural networks [Bengio et al, 2015]. This type of hidden layer is foundational to these algorithms and consists of a set of weights & biases that go into neurons that have the same activation function. However, for the purposes of this thesis, the fully connected dense layer that follows the LSTM can be considered a reformatting of the LSTM output for inferencing use. It contains no activation function so effectively works as an affine transformation of the previous layer [Südholt et al, 2022].

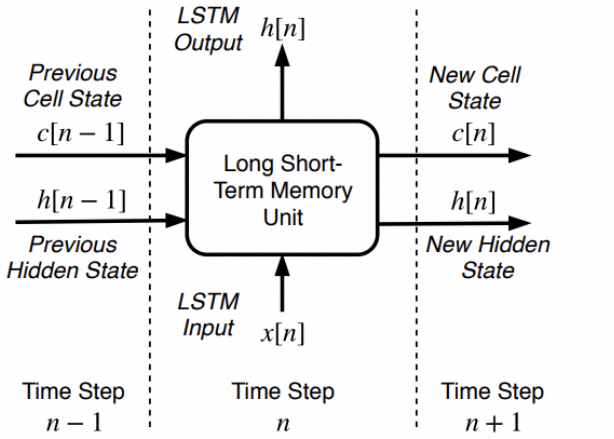


Figure 2.14: An LSTM unit overview [Wright et al, 2019].

2.3.4 LSTM

An LSTM is a stateful layer composed of three subcomponents, or gates, which together make a hidden unit [Sherstinsky, 2020]. An LSTM's state is used and updated at each individual timestep (i.e. sample), which makes it perfectly suited for audio and long data sequences. As the name suggests, an LSTM carries both long term memory (cell state) and short term memory (hidden state) [Staudemeyer & Morris, 2019]. The cell state is an aggregate of the entire sequence memory, while the hidden state is the processed results of the previous time step [Mienye et al, 2024]. Both the hidden and cell states are updated while the current timestep is processed within the LSTM cell. The vector shape of the hidden and cell states is dependent on the number of hidden units in the LSTM layer [Wright et al., 2019]. These hidden units are the amount of LSTM cells that run concurrently at each timestep.

An LSTM unit contains three subcomponents, the forget gate, input gate, and output gate (Fig. 2.15) [Staudemeyer & Morris, 2019]. Each of these gates contains either a logistic sigmoid and/or hyperbolic tangent (tanh) activation function. The tanh's encode normalized data as either memory or an output. The sigmoid functions determine what aspects of the two memories are amplified or diminished, essentially deciding which timesteps are more or less important to the overall sequence [Sherstinsky, 2020].

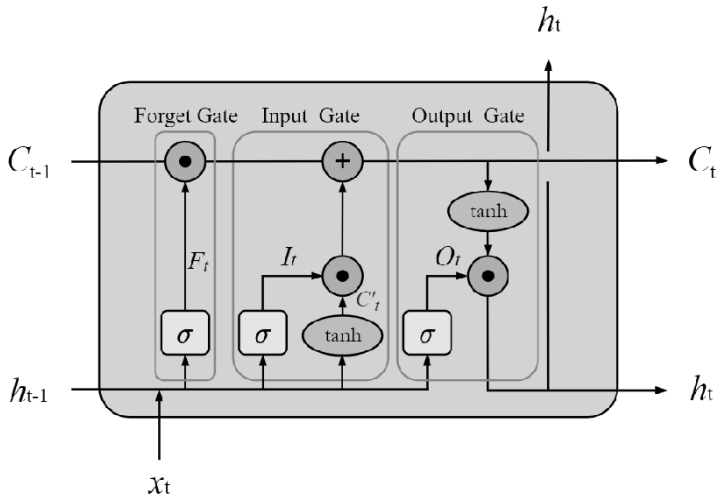


Figure 2.15: An LSTM cell or unit with its subcomponents shown [Chen, 2024].

The forget gate is responsible for determining what information should be forgotten from the long term memory based on both the current input and the previous time step (i.e. the hidden state). Next, the input gate processes the current input into the cell state. While the forget gate can be considered as the subtraction of non-useful information from the long term memory, the input gate is the addition of useful information into it [Dolphin, 2020]. First, feature extraction occurs to actually encode the input using a tanh function, then a sigmoid function determines how important the current hidden state and timestep actually are, before committing them into the cell state. Lastly, the output gate determines the next hidden state, which can be used as both the predicted output of the current input and the hidden state for the next time step. This distinction is important for audio modeling, because it allows the system to output entire data sequences. For more information breaking down an LSTM cell see [Sherstinsky, 2020].

The equations for each step in a LSTM cell can be seen in Equation 18, where *i[n]* is the input gate, *f[n]* is the forget gate, *c~[n]* is the cell state, *o[n]* is the output gate, *c[n]* is the updated cell state, *h[n]* is the updated hidden state, and all weights and biases are given as *W* and *b*'s. The weight matrices and bias vector sizes are determined by the input size and number of LSTM hidden units – the vector shape of the two states is also given by the number of LSTM hidden units [Wright et al., 2019].

$$
\begin{aligned}
i[n] &= \sigma(W_{ii}x[n] + b_{ii} + W_{hi}h[n-1] + b_{hi}) \\
f[n] &= \sigma(W_{if}x[n] + b_{if} + W_{hf}h[n-1] + b_{hf}) \\
c\sim[n] &= tanh(W_{ic}x[n] + b_{ic} + W_{hc}h[n-1] + b_{hc}) \\
o[n] &= \sigma(W_{io}x[n] + b_{io} + W_{ho}h[n-1] + b_{ho}) \\
c[n] &= f[n]c[n-1] + i[n]c\sim[n] \\
h[n] &= o[n]tanh(c[n])
\end{aligned}
$$

18.

Note that how much a timestep is remembered (i.e., how important it is) is given by the sigmoid activation function, a set of weights, and a set of biases. The weights are updated via back propagation of the entire system, but remain static within an epoch so the same parameters are applied to each time step. The biases are simple offsets to stabilize the equations and are in no way connected to the biases found in electronics [Wright et al., 2019].

Lastly, while Keras and Pytorch training functions are highly optimized and easy to implement, the temporal nature of the LSTM layer necessitates that it be warmed-up [Bourdin et al, 2025]. Training the LSTM while it has no memory (i.e. an empty state) will create skewed results – returning to the sentence analogy, this would be like asking the model to predict the next word after not having seen any of the other words that already make up the sentence! However, during this warm-up period, no loss calculation nor gradient descent actually occurs – the model does not learn, it just provides the LSTM state.

2.3.5 Data Collection for Guitar Distortion

The type of data used and how it is processed is just as important as the model architecture [Vanhatalo et al, 2022]. For some distortion models, only between 4-6 minutes of guitar playing per knob position should be enough for training, which is surprisingly low [Miklánek et al, 2023]. The data should include playing styles and guitar qualities of all types to limit overfitting – in layman's terms, overfitting refers to the model doing a fantastic job of

learning on limited data, but struggling with any variations from that specific dataset [Chowdhury, 2020].

While the breadth of the dataset should be wide, everything else should be exact. Specifically, it is necessary that the matching input (clean guitar) and target (distorted guitar) are aligned as well as they possibly can be. If they are out of step, then the loss functions will be effectively meaningless because it is comparing samples between the target and prediction that have little to no correlation.

2.3.6 Truncated Back Propagation Through Time

Regular back propagation across long data sequences in RNN's can often lead to exploding or vanishing gradients, which occurs when the gradients used to adjust the network weights either get exponentially bigger or smaller. LSTM's were actually developed to fix this problem, but can be taken a step further [Staudemeyer & Morris, 2019]. BPTT mitigates this error by unrolling the RNN and back propagating through it like a regular feedforward network, but with the same parameters that appear at each step [Werbos, 1990]. However, the data sequences going through this pseudo-feedward network can still be incredibly long, so instead loss is calculated and back propagation occurs on truncated sections of the data (i.e. every 2048 samples for audio) [Wright et al, 2019]. This process is commonly referred to as truncated back propagation through time (truncated BPTT).

2.3.7 Loss Functions for Audio

A loss function determines how different a model's prediction is from the target output, often called the 'ground-truth' [Kelleher, 2019]. It is crucial that an appropriate loss function is used depending on the training data. While more standard loss functions like Mean Squared Error are applicable to a simple dataset, modeling guitar distortion requires a more extensive approach [Damskägg et al, 2019a]. The loss function for audio also depends on if the audio exists in the time or frequency domain.

Guitar distortion models that input/output raw audio waveforms have really succeeded using an Error-to-Signal Ratio (ESR) loss function in combination with a DC error offset and a

premphasis filter [Wright et al, 2019]. ESR can be defined as the sum-of-squares error function that has been normalized by the target energy [Steinmetz & Reiss, 2020]. Without this normalization, the training data with the most energy would dominate and skew the loss [Damskägg et al, 2019a].

The ESR evaluates how well the model mimics the target waveform's shape, while the additional DC offset loss makes sure that the waveform remains centered around zero amplitude. Without this DC offset, the ESR loss alone may converge, but place the waveform symmetrical across a non-zero amplitude, which may lead to clipping and reduced headroom through a skewed dynamic range [Wright et al, 2019; Bradley, 2006]. Lastly, for distortion modeling, which mainly includes the addition of higher frequency harmonics, it is beneficial for the loss function to be focused on how the predicted matches the target in the higher end of the frequency spectrum. So, before the ESR is calculated, the predicted/target pair is placed through a preemphasis filter [Wright et al, 2019]. Conceptually, this filter acts as a high pass filter so that the ESR is more focused on the higher frequency data included in the distortion. It has been shown that the inclusion of an A-weighted preemphasis filter, which reduces frequencies below 500 Hz, can create a more accurate distortion model without adding additional computational costs [Wright & Välimäki, 2019].

2.3.8 Batch Processing

Many RNN architectures for distortion modeling suggest splitting up the training data into half second segments to allow for dataset shuffling, which helps reduce overfitting [Kelleher, 2019]. However, to warm-up and perform truncated BPTT on every half second clip with a sample rate of 44,100 from ~4 minutes of audio *sequentially*, while resetting the LSTM state between each half second clip would yield very accurate results, but it would take incredibly long. Instead, Keras and Pytorch batching can be used to significantly reduce training time. This batching means warming-up and training with truncated BPTT on multiple half second clips in parallel. This method significantly reduces training time if, for example, the entire dataset is run concurrently at each epoch, but may include a slight negative effect because the LSTM state is reset between each epoch and not technically between each half second sequence [Wright et al, 2019].

2.3.9 Real Time Implementation

After a model is trained, just the weights and model parameters can be exported and turned into an inferencing engine. An inferencing engine will take an input and create an output through forward propagation, without any additional training step [Chowdhury, 2021]. The engine must be lightweight enough to provide real time inferencing if used within a JUCE or another plugin framework for real time use. Real time audio can be considered real time if it is played back within around a 10 ms window – for modern computers, a buffer size of 1-5 ms should be achievable [Bencina, 2011]. Large machine learning libraries like Keras or Pytorch can create inferencing engines from a model export, but often do not consider real time applications. Instead, libraries like RTNeural were created with real time audio inferencing in mind, using explicit vectorization (SIMD instructions), WIP approximations, and memory optimizations [Chowdhury, 2021].

2.4 When Does Emulation Not Work?

2.4.1 White Box

White box emulation is the idealized version of a circuit. However, physical circuits have heat loss and component specific quirks that are smoothed out in their idealized version [Simper, 2020]. The individual characteristics and sonic quality of a given circuit are in part determined by their unideal conditions. The sound of a hardware device is composed by who made the components, how they were assembled, how old they are, how often they are used, and much more [Chowdhury, 2023]. A diode made in one factory may not sound the same as a diode made in another, for example. Taking an absurdly granular view, these conditions are influenced by everything from global trade to what kind of day the designer was having! An idealized circuit will not reflect these imperfections and inevitably sound different. Of course, a white box emulation is only idealized if the schematic is accurate.

To even begin with a white box emulation, the developer must either have the schematic or be able to take apart the hardware [Mačák, 2012]. Even if they found the schematics, how can the developer ensure that it is actually accurate? If even one resistor value is off, then the model cannot be considered an accurate emulation.

2.4.2 Black Box

While the schematics are of no concern for a black box emulation, there are still pitfalls regarding this technique. White box techniques can be considered the general idealized version of a circuit, but training a neural network on one specific piece of hardware will yield an emulation of that *specific* piece of hardware. If it has any malfunctions or inconsistencies in how it should sound, those will be reflected in the model. This hyperspecificity can be overcome by the inclusion of different instances of the same hardware, but this is not always possible given the developer's budget.

The other and possibly more glaring problem with black box modeling is data collection. For end-to-end modeling, 10 discrete positions per knob is suggested [Juvela et al, 2024]. While feasible for hardware with just one knob, the amount of data is scaled by a factor of ten for each additional knob. With a four minute audio training clip and a piece of hardware with five knobs, which is common for guitar amplifiers, it would take a minimum of 6,666.6 hours to record the proper dataset. Researchers at Neural DSP have scaled this number down by collecting data through random knob positions [Juvela et al, 2024]. However, it is still a huge undertaking to collect a vast amount of audio data by hand. So, these researchers have developed a knob turning data collection robot to handle this for them [Neural DSP, 2024]. Unfortunately, most developers do not have this same access.

3 Review of Literature

There is an abundance of comparative analysis between virtual analog modeling techniques specifically for guitar distortion. Unfortunately, it is difficult to find research that is all encompassing because there are so many methods. Furthermore, each piece of hardware typically requires a bespoke solution and emulation technique, which makes analytic comparison even more difficult. One reason neural networks are being lauded is that they offer a more

general approach that can be applied to various distortion circuits with only minor hyperparameter tweaks [Wright, 2023]. However, there is plenty of discourse on what network architecture is best.

Most white box techniques are founded in well established physics and have been used to model a litany of distortions. Within an academic setting, state space alone has been used to model a Fuzz Face pedal [Dempwolf & Zolzer, 2011], a Dunlop CryBaby Wah [Holters & Zölzer, 2011], and an MXR Phase 90 pedal [Eichas et al, 2014], among others. Countless more have been made for production and retail. Even newer techniques like WDF have proven effective at modeling pedals like the Klon Centaur for real time use [Chowdhury, 2020]. Just as well, the DK-Method is explored in great detail for real time application of guitar distortion in [Yeh, 2009]. It does seem that while there are some tradeoffs and benefits unique to each white box method, the technique the DSP engineer selects is ultimately up to their discretion and more importantly, preference.

For neural network based models, there are two broad architectures used successfully in literature. First are variations of convolutional neural networks (CNN). The WaveNet model is especially notable, using casual convolution layers to train directly on raw audio data [Oord et al, 2016]. A feedforward version of this architecture, called a Temporal Convolution Network (TCN), was developed in [Damskägg et al, 2019b] and included stacked dilation layers with various patterns. This work was followed up in [Damskägg et al, 2019a] to explore real time implementation. [Steinmetz & Reiss, 2021] showed later that these TCN networks could be scaled back significantly if the dilation was increased dramatically for each convolution layer.

The second type of model architecture is an RNN. A subtype of RNN's called a Nonlinear AutoRegressive eXogenous (NARX) was first used in 2013 to train on audio sequences – this network mimics neuron dropout through limited connectivity, which helps to reduce exploding/vanishing gradients [Covert & Livingston, 2013]. However, a more elegant solution was found in [Zhang et al, 2018] by using LSTM's, whose cell architecture already solves this issue. This architecture included multiple stacked layers with varying hidden unit sizes – subjective listening tests proved it to be unsatisfactory. Instead, [Wright et al, 2019] showed that an architecture of a single LSTM layer followed by a fully connected layer would be sufficient in modeling distortion given the proper number of hidden units. They also experimented on swapping in GRUs to help lower computational costs even further. Building on

this research, [Wright et al, 2020] showed that this LSTM network sometimes, but not always, beat the Wavenet Models in terms of accuracy.

Lastly, research into using hybrid and regular deep neural networks for distortions has also been performed. These such architectures include lesser used layers like bi-directional LSTMs, max pooling, and convolutional autoencoders – these models have been less widely adopted [Vanhatalo et al, 2022].

Comparing these architectures is a complicated endeavor given how different their hyperparameters can be, but possible because of their general similarity in dataset, training style, and loss functions. [Vanhatalo et al, 2022] found that an LSTM to dense layer architecture with 32 hidden units outperformed all other models in terms of accuracy, while there is slight discussion on whether it performed best in processing speed.

4 Methodology

4.1 Overview

The design of this thesis is two fold. First, two separate VST plugins were made to emulate the same guitar distortion pedal, the Tube Screamer TS9. One plugin is a white box emulation and the other is a black box. The focus of this thesis is to see how these different methods handle more challenging, non-linear circuits, so while the Tube Screamer contains a tone, volume, and drive knob, only the drive (made via diodes) was focused on for both emulations. In the second stage, listening and playing tests were performed to see which emulation has a higher fidelity to the original plugin. It is important to distinguish that the purpose of these tests is not to see which distortion the listeners and players prefer, but understand which one they thought sounded closer to the original pedal. However, a brief discussion of preference is included in the discussion section.

4.2 Chosen Techniques

While SPICE simulations and state space representations offer highly accurate emulations, the DK-Method was used to create the white box emulation for three reasons. First, it allows for real time audio [Tarr, 2021]. Second, it is a much more practical technique that is accessible for anyone with a fundamental understanding of electronics [Benois, 2013]. Lastly, the nodal equations are easily written in C++ and included in a simple audio plugin developed within the JUCE framework. This framework allows for the easy creation of plugin formats including VST, VST3, and AU, making it ideal for cross DAW development.

The black box emulation for this thesis followed the principles laid out by Alec Wright and corroborated by Jatin Chowdhury, which proved a single RNN layer, especially an LSTM with 32-96 hidden units, that fed into a dense layer would be sufficient for accurate modeling without sacrificing real time capabilities [Wright et al, 2020]. However, rather than building an inferencing engine or using one developed by Keras or Pytorch, the real time audio inferencing library RTNeural, also developed by Chowdhury, was included within another JUCE plugin [Chowdhury, 2021].

5 Implementation

5.1 White Box

A look behind the curtain is required to understand the implementation of the white box emulation in this thesis. Originally, a pure state space representation was attempted for the entirety of the preamp stage of a guitar amp. However, the equations derived never produced a workable model. Instead, a state space representation of the Tube Screamer pedal was attempted, instead. While the schematics of this pedal were much more simple, the matrices derived from the state space equations quickly grew out of hand. After discussion with industry professionals and working DSP engineers, who agreed that a DK-Method approach is more than sufficient for modeling this pedal, the DK-Method was used instead. Unfortunately, a significant amount of time had passed trying to sort the state space equations, so the math and code used is a modified

version of Eric Tarr's work. While Tarr's reputation in the audio field elicits confidence in the fidelity of his model and accuracy of his derivations, certain changes had to be made.

First was a matter of schematics. The pedal used in this thesis was a TS9, while Tarr's emulation focused on the original TS808. The TS808 could not be used for this thesis because of budget constraints. Fortunately for emulation purposes, only the subcircuits that actually shape the sound needed to be included, specifically the tone and clipping sections – there is no difference in component layout nor values between these two pedals for these subcircuits. However, it is worth noting that depending on time of production, the actual diodes and op amps used may differ between the two pedals [Chowdhury, 2019].

Second, two major changes were made to the JUCE implementation. First, all included UI was removed and swapped with generic knob settings. Rather than a ranged-knob, which is often included in audio plugins, this knob directly changes the values of the drive potentiometer from 0-1 to provide a baseline representation of the clipping stage. Within the code, these knobs were also attached to the process block through an audio processor value tree.

The second major change to the code base was to make this plugin multichannel safe. The original code was safe for a mono output signal, but would change the value of the state variables of the capacitors between each channel when more were included. So if it was in stereo, the left channel would process a sample and then update the state variables, which would then be sent to the right channel for processing and updates. This difference would cause significant artifacting in the sound. Instead, each channel was provided with its own set of state variables that are not accessible outside of that specific channel.

Actually creating a DK-Method emulation can be broken down into the following steps.

1. Schematic Analysis
2. Node-by-Node KCL Derivation
3. Formation of Transfer Equation
4. C++ Implementation.

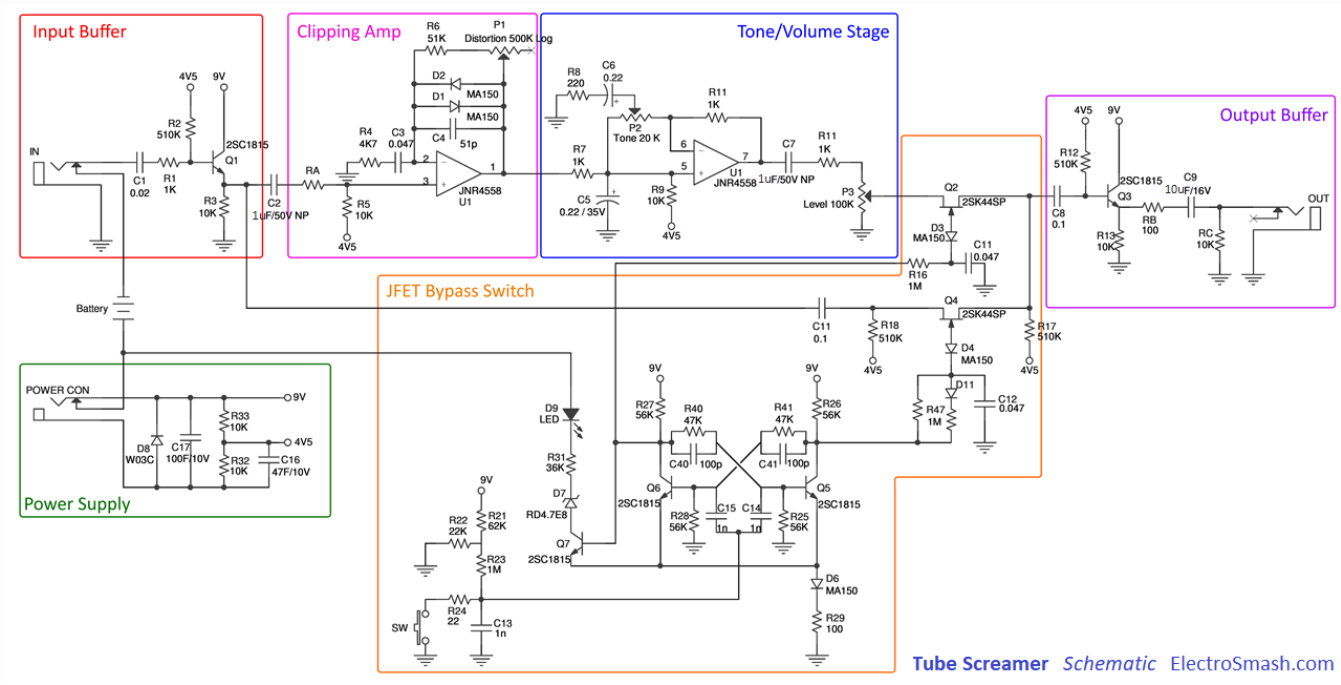


Figure 5.1: Overall schematic of a TS808 pedal [Electrosmash, 2025].

5.1.1 Tone Stage

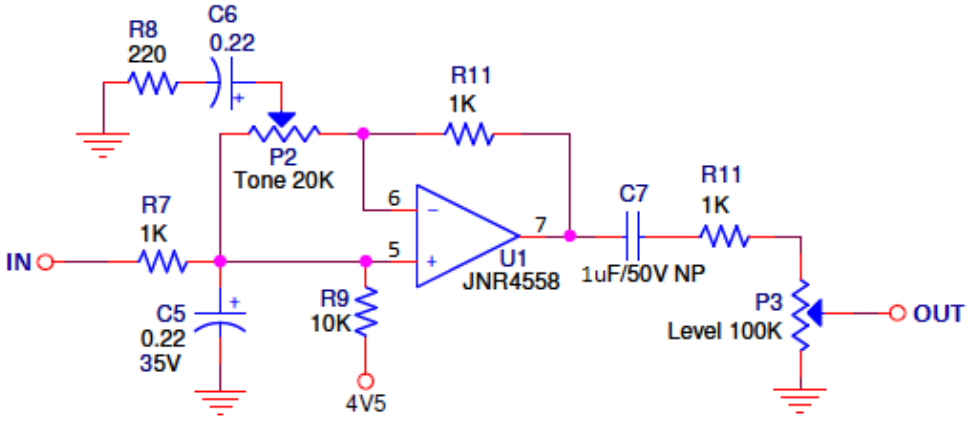


Figure 5.2: The tone stage schematic of both the TS808 and TS9 [Electrosmash, 2025].

The tone stage accepts an input directly from the previous clipping stage (Fig. 5.2). First the signal enters a variable low-pass filter through the initial passive components before entering the main active component, the op amp. This component boosts and flattens the treble in the signal to compensate for the previous low pass filtering [Electrosmash, 2025]. Overall, this stage consists of an op amp, three capacitors, and a handful of resistors/potentiometers. There is, however, one modification that can be made to the schematic for better clarity. When splitting the P2 potentiometer into the two alpha controlled variable resistors, it can instead be split between P2 and R9, where P2 is multiplied by (1 - alpha) and R9 is multiplied by alpha. In this case, a copy of C6 and R8 should also follow in series after R9 on the same branch. The updated schematic, with an expanded capacitor view for the DK-Method can be seen in Fig 5.3. Note that the resistor labels have changed to account for the additional components.

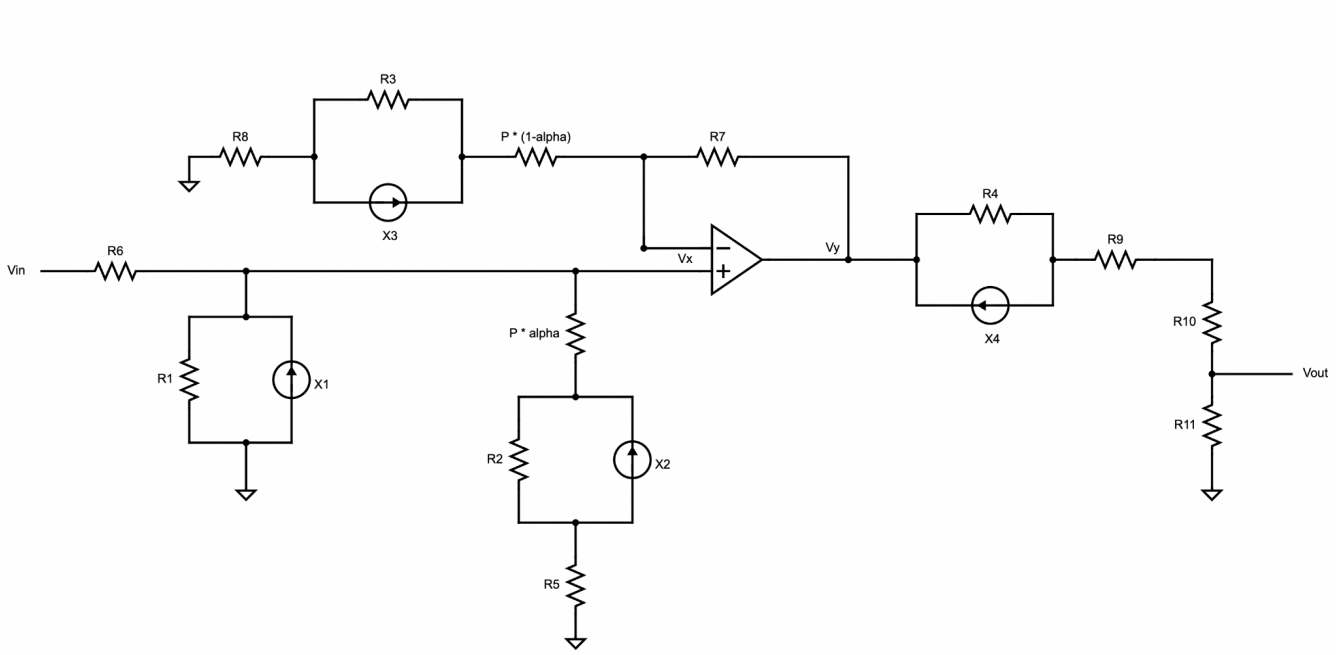


Figure 5.3: The tone stage schematic of both the TS808 and TS9 rewritten for simplification and better DK-Method visualization.

Next, nodes can be selected and the DK-method can be applied to create KCL-based equations for each. The nodes Vin, Vx, Vy, and Vout can be seen in Fig 5.3. Working backwards, Vy can be solved for as a function of Vout with the following:

$$V_y = V_{out} \cdot \left(1 + \frac{R_{10}}{R_{11}} + \frac{R_9}{R_{11}} + \frac{R_4}{R_{11}}\right) + R_4 \cdot X_4$$

Next, the two terminals of the op amp can be solved and set equal to each other. The negative terminal can be solved for by:

$$V_y = V_x + \frac{R_7}{G_3 \cdot P(1-\alpha)} \cdot V_x + \frac{R_7 \cdot R_3}{G_3 \cdot P(1-\alpha)} \cdot x_3$$

$$G_3 = 1 + \frac{R_3}{P(1-\alpha)} + \frac{R_8}{P(1-\alpha)}$$

Substituting Vy, the final negative terminal equation can be made in terms of Vx and Vout.

$$V_x = \frac{G_0}{G_x} \cdot V_{out} + \frac{R_7 \cdot R_3}{G_x \cdot G_3 \cdot P(1-\alpha)} + \frac{R_4}{G_x} \cdot x_4$$

$$G_0 = 1 + \frac{R_{10}}{R_{11}} + \frac{R_9}{R_{11}} + \frac{R_4}{R_{11}}$$

$$G_x = 1 + \frac{R_7}{G_3 \cdot P(1-\alpha)}$$

Now that Vx is a function of Vout, Vx must be substituted for Vin for a final transfer function. With op amps, this substitution can sometimes be performed by solving for the other terminal and setting both equal to each other. Remember that an ideal op amp is being considered here, so it is assumed that both terminals will be equal at all times. The positive terminal can be represented as follows.

$$V_x = \frac{V_{in}}{R_6 \cdot G_z} + \frac{X_1}{G_z} + \frac{R_2 \cdot X_2}{G_2 \cdot G_z \cdot (P \cdot \alpha)}$$

$$G_z = \frac{1}{R_1} + \frac{1}{G_2 \cdot (P \cdot \alpha)} + \frac{1}{R_6}$$

$$G_2 = 1 + \frac{R_2}{(P \cdot \alpha)} + \frac{R_5}{R_6}$$

After setting the two terminals equal and simplifying terms, the final transfer equation for the tone stage can be given as follows:

$$V_{out} = \frac{G_x}{R_6 \cdot G_z \cdot G_0} \cdot V_{in} + \frac{G_x}{G_z \cdot G_0} \cdot X_1 + \frac{G_x \cdot R_2}{G_2 \cdot G_0 \cdot G_z \cdot (P \cdot \alpha)} \cdot X_2 + \frac{R_3 \cdot R_7}{G_0 \cdot G_3 \cdot P(1-\alpha)} \cdot X_3 + \frac{R_{11}}{G_0} \cdot x_4$$

Lastly, each capacitor needs its own state variable update equation based on its capacitance and voltage drop according to the DK-Method. The boilerplate equation for these state variable updates using the Bilinear Transform is given in Equation 15, but KCL may be required to solve for the voltage across the component if it does not feed directly into ground. The four state equations for the tone stage can be given as follows.

$$x_1[n] = \frac{2}{R_1}(V_x) - x_1[n-1]$$

$$x_2[n] = \frac{2}{R_2}\left(\frac{V_x}{G_r} + \left(\frac{(P \cdot \alpha) + R_5}{G_r} \cdot x_2[n-1]\right)\right) - x_2[n-1]$$

$$x_3[n] = \frac{2}{R_3}\left(\frac{V_x}{G_s} + \left(\frac{P(1-\alpha) + R_8}{G_s} \cdot x_3[n-1]\right)\right) - x_3[n-1]$$

$$x_4[n] = \frac{2}{R_4}(V_{out}) - x_4[n-1]$$

$$G_r = 1 + \frac{(P \cdot \alpha)}{R_2} + \frac{R_5}{R_2}$$

$$G_s = 1 + \frac{P(1-\alpha)}{R_3} + \frac{R_8}{R_3}$$

Note that the R's that relate to the impedance of the capacitors are given as follows, where T is the timestep and C is the capacitance value:

$$Z = \frac{T}{2C}$$

5.1.2 Clipping Stage

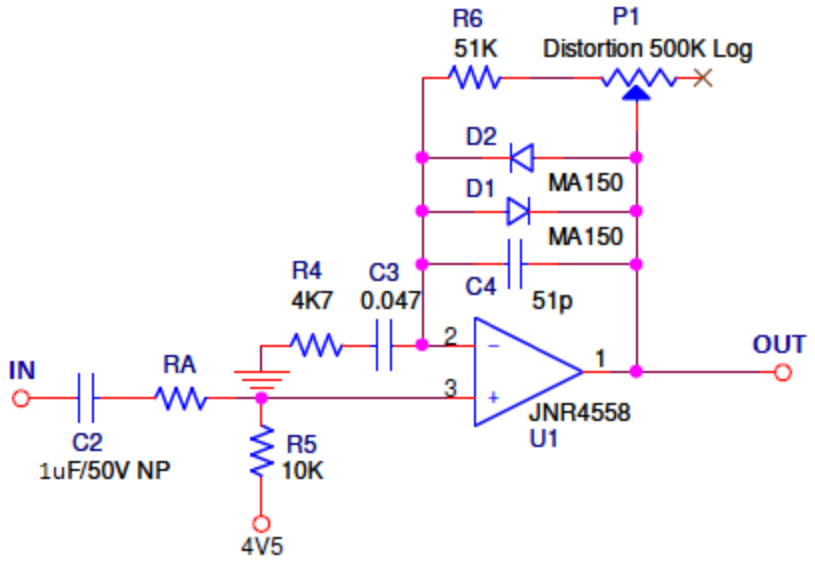


Figure 5.4: The clipping stage schematic of both the TS808 and TS9 [Electrosmash, 2025].

The clipping stage requires an overall less intense transfer function derivation, but includes a potentially tricky non-linear component, a diode (Fig. 54.). The remainder of the components are much more familiar, consisting of an op amp, capacitors, and resistors/potentiometers. However, one major simplification can be made based on the properties of an ideal op amp. As previously stated, an ideal op amp does not pass any current across it. Because of this property, there is no voltage drop across the components leading into the positive terminal, meaning that the voltage going into it is equal to Vin. Since the voltages at each terminal of an ideal op amp are equal, then the voltage at the negative terminal is also Vin. So, for all intents and purposes, the op amp can be ignored. While it has functionality regarding the entire pedal circuit, it is irrelevant for the clipping distortion.

Two more simplifications can be made in this schematic. First, the potentiometer lies in series with another resistor. So, when the potentiometer is rewritten as an alpha-dependent set of variable resistors, the first resistor can just be combined with them. Second, Shockley's Diode

equation will still be used, but for two parallel and opposing diodes, the combined equations can be simplified to Equation 19.

$$Is(e^{\frac{V}{\eta V_t}} - 1) - Is(e^{\frac{V}{\eta V_t}} - 1) = 2 \cdot Is \cdot sinh(e^{\frac{V}{\eta V_t}}) \quad 19.$$

Rewritten with these simplifications, the updated schematic can be viewed in Fig 5.16 – the combined voltage drop is labelled as Vd.

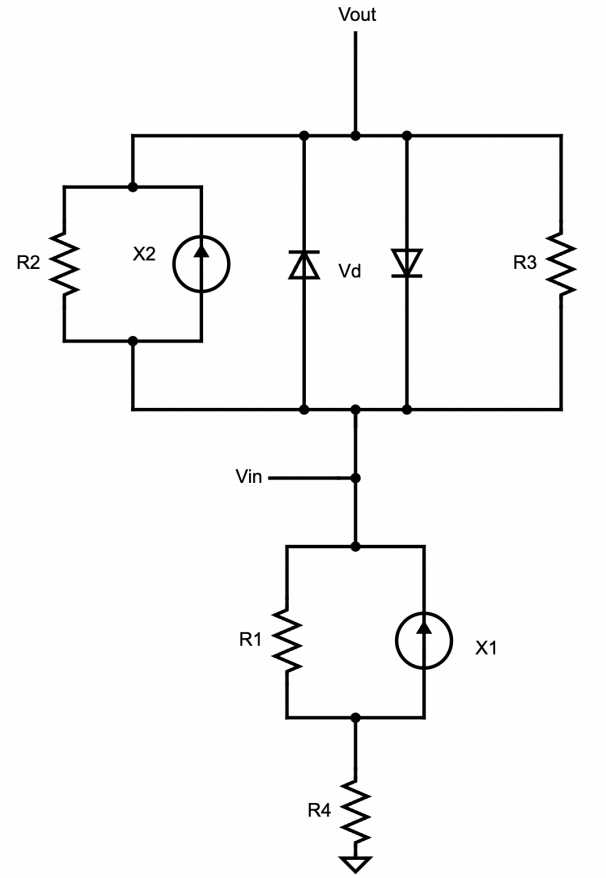


Figure 5.6: The clipping stage schematic of both the TS808 and TS9 rewritten for simplification and better DK-Method visualization.

Luckily, there are no extra nodes between Vin and Vout, so creating a transfer function should be relatively simple. Also, because the components between Vin and Vout are all in parallel, their voltage drop can be assumed as equal:

$$V_d = V_{out} - V_{in}$$
$$V_d = V_{R2} = V_{R3}$$

So, as current flows from Vout to ground, the KCL equation from Vout to Vin can be written as:

$$i_{Vout\ to\ Vin} = \frac{V_d}{R_2} - X_2 + 2 \cdot Is \cdot sinh(e^{\frac{V_d}{\eta V_t}}) + \frac{V_d}{R_3}$$

Moving on from Vout, the current of the system can also be expressed as,

$$i_{Vin\ to\ Ground} = \frac{V_{R4} - 0}{R_4}$$

Now it is a matter of creating a substitution for VR4 to put this current equation in terms of Vin, where Vin is the combination VR4 and VR1. This combination can actually occur because of another rule called Kirchoff's Voltage Law.

$$V_{in} = V_{R4} + V_{R1} \quad \Rightarrow \quad V_{R4} = V_{in} - V_{R1}$$

It is also known that the current through both the capacitor and the resistor is equal,

$$\frac{V_{R4}}{R_4} = \frac{V_{R1}}{1} - x_1$$

Which simplifies to,

$$V_{R1} = \frac{V_{in}}{G_4} - \frac{R_1}{G_4} \cdot x_1$$

$$G_4 = \frac{R_1}{R_4} + 1$$

So, substituting in VR1 into the current equation from Vin to ground,

$$i_{Vin\ to\ Ground} = \frac{V_{R4}}{R_4}$$

$$i_{Vin\ to\ Ground} = \frac{1}{R_4} \cdot \left(\frac{V_{in}}{G_4} - \frac{R_1}{G_4} \cdot x_1\right)$$

Then, using KCL, the overall current equations can be set equal. This equation is then solved for zero and the derivative is found for later output mapping using a Newton-Raphson solver. Remember, the R relating the capacitor is actually equal to $\frac{T}{2C}$.

$$\frac{V_d}{R_2} - X_2 + 2 \cdot Is \cdot sinh\left(e^{\frac{V_d}{\eta V_t}}\right) + \frac{V_d}{R_3} = \frac{1}{R_4} \cdot \frac{V_{in}}{G_4} - \frac{R_1}{G_4} \cdot x_1$$

$$0 = \frac{1}{R_4} \cdot \frac{V_{in}}{G_4} - \frac{R_1}{G_4} \cdot x_1 - \left(\frac{V_d}{R_2} - X_2 + 2 \cdot Is \cdot sinh\left(e^{\frac{V_d}{\eta V_t}}\right) + \frac{V_d}{R_3}\right)$$

$$f'(V_d) = \frac{1}{R_2} + \frac{1}{R_3} + \frac{2 \cdot Is}{\eta V_t}\left(sinh\left(\frac{V_d}{\eta V_t}\right)\right)$$

$$V_d = V_{out} - V_{in}$$

5.1.3 C++ Implementation

Actually implementing these equations in C++ within a JUCE framework is remarkably easy compared to their derivation. A class for each subcircuit is created with four methods and all of the components within it. The first method is a process function that takes in a sample (Vin) and outputs the Vout value. It also handles all state updates. The next method reads in the sample rate to determine the time step along with the number of channels used. Next, there is a method that checks whenever a knob value has changed. The last method, which updates all of the variable component values, is called when the sample rate or knob position has changed.

5.2 Black Box

5.2.1 Overview

The neural network architecture established in [Wright et al, 2019] was chosen for the black box implementation. Not only does it pose a very elegant solution, but it has been proven to be one of the most accurate and efficient modeling techniques for distortion effects [Vanhatalo et al, 2022]. Furthermore, one underlying motivation for this thesis is to determine if audio companies specializing in neural modeling are actually creating a better product. This architecture was researched and developed largely by researchers at Neural DSP, which is one of the leading neural plugin development companies at the moment, so using it is appropriate [Wright et al, 2019].

5.2.2 Preprocessing

Prior to building and training the model, data had to be collected, cleaned, and reformatted. First, ~5 minutes of clean guitar playing was passed through the pedal at 0.25, 0.75, and 1.0 distortion. The tone knob was set at 1.0 the entire time. This clean guitar recording was a collection of various playing styles, guitars, pickups, and even additional effects like reverb. Lastly, clean/distorted pairs were exported from the DAW, Reaper in this case, for processing in a Python script.

This script read in each input/target pair based on their names, checked alignment of the waveform, and reformatted them into LSTM ready tensors of 0.5 second sequences. If any sequence pair had a misalignment of greater than 50 samples, then it was discarded from the dataset. This misalignment occurs as latency is introduced in the recording process and can lead to significantly skewed results during training. Each 0.5 second sequence was added to a larger input or target list, respectively, which were both then saved to a .npz file to be read by the model. The key difference between the input and output was that the input had two features, amplitude data from the waveform and a knob setting, while the output was just a waveform. While this process could be included within just one large model script, separating this preprocessing out was beneficial for debugging, encapsulation, and ensuring that the inputting tensor shapes were correct for the RNN.

5.2.3 Architecture & Hyperparameters

The model used was a sequential recurrent neural network consisting of a stateful LSTM[64] layer into a dense[1] layer with no activation function, as suggested in [Wright et al, 2019]. While this paper shows an LSTM with 32 hidden units was most accurate, trial and error demonstrated a lowest validation loss with 64 units for this thesis' dataset. The model trained on a Macbook M1 microchip using Conda/Tensorflow-Metal environment for 500 epochs with a Adam optimizer and initial learning rate of 5e-4. The model also trained with a batch size of 500 – for more rapid training, the entire dataset could have been trained as one big batch rather than semi-micro batches, but the dataset was too large for GPU processing on the M1. Instead, a batch size of 500 was the compromise, through guess-and-checking.

During training, each 0.5 second sequence first warmed-up the model for 1,000 samples to allow for the development of a cell state before truncated BPTT occurred every 2,048 samples until the end of the sequence. Validation, which made up just 5% of the dataset, occurred every other epoch. The validation set was low because ~5 minutes of audio is not a lot – validation was included really just to ensure no overfitting occurred, while the majority of the dataset was reserved for training. Validation occurred through a custom built inference model that had a stateless, but otherwise identical architecture as the training model. The LSTM state of the training model was reset at the start of each epoch. The model also trained using a custom loss function combining a DC offset and ESR, with a premphasis filter with coefficient 0.85. Lastly, the gradients were clipped to limit randomly exploding losses, which occurred in the corresponding literature [Wright et al, 2019].

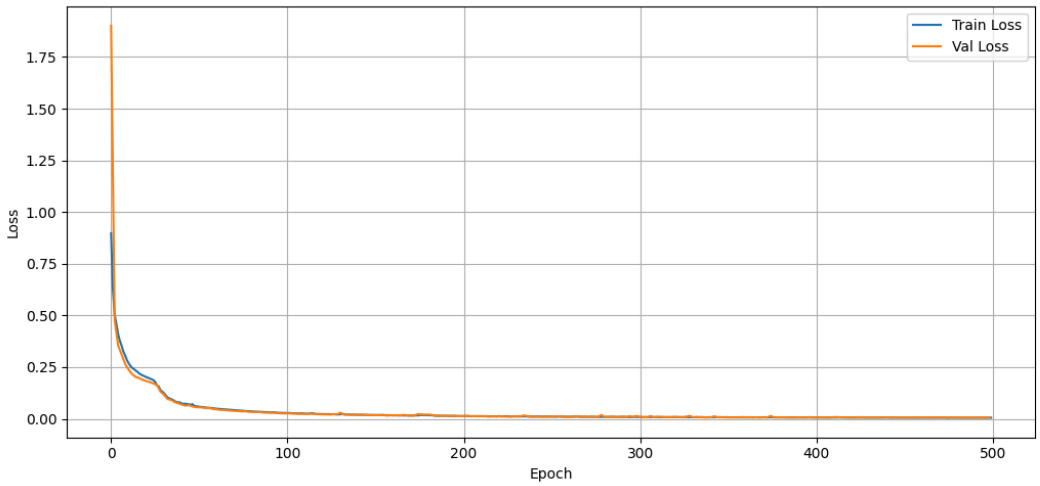


Figure 5.7: Validation and Training loss curve.

The model trained for ~6 hours and ended with a training loss of 0.005258 and a validation loss of 0.005946 along with a healthy loss curve, which suggests training occurred over 500 epochs (Fig. 5.7). There was slight, but consistent overfitting when this model trained on a smaller dataset (~4 minutes), but when it was expanded, the overfitting lessened dramatically. Fig 6.2 shows that the predicted waveforms do match up with the target distortion well, but are not exact. A larger dataset, more epochs, a heavier model, and a greater high frequency emphasis are among the many things that might allow for greater precision. Lastly, the weights were exported using an RTNeural specific export function to be used in JUCE.

5.2.4 C++ Implementation

Just like with the white box emulation, the JUCE implementation is significantly easier, especially with RTNeural. Essentially, the library acts as an inference engine – basic steps were followed in the API to create a dynamically compiled model that runs in real time in Reaper.

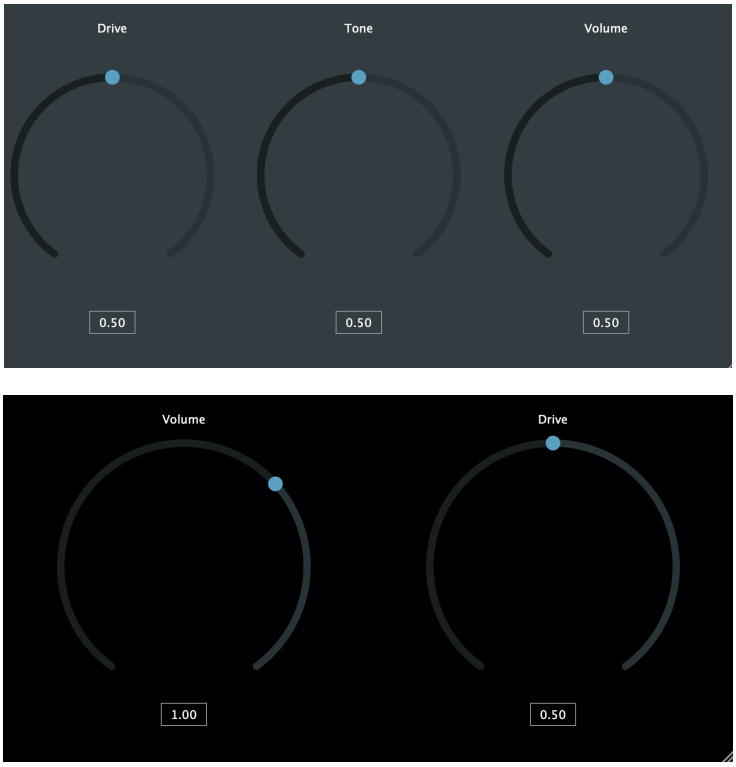


Figure 5.8: UI for the white box (top) and black box (bottom) plugins.

6 Evaluation

6.1 Testing Design

Two different tests were made to gauge the accuracy of each model. First, a Multiple Stimuli with Hidden Reference and Anchor (MUSHRA) using prerecorded audio was performed. This audio was guitar playing in three different styles with different pickups and tones being used. It was then run through the pedal and both plugins with a distortion level of 0.25, 0.75, and 1.0. Listeners were asked to gauge how similar each distortion sound was to the original pedal audio on a scale of 0-100. All tests were performed in lab conditions using Prenonus HD7 headphones. This listening test also included audio recorded through a TS Mini pedal, which is another Tube Screamer version and was included to further diversify the amount of selections in the MUSHRA test.

The second test was also a MUSHRA test of the distortion at 0.25, 0.75, and 1.0, but instead of listening, guitarists actually played through the pedal and plugins in real time. The players were asked to first play through the pedal for however long they like to get acquainted with the sound. Then, they were blindly presented with either the clean signal, the white box plugin, the black box plugin, or the original pedal again and asked to rate how similar each sounds to the original from 0-100. Like in the listening test, the players were able to listen to the reference or any previous track at any point to recontextualize their ears. All playing tests were performed on a Fender Telecaster guitar. All guitar settings were set by the participant at the start of the test and not changed for the duration of it. Lastly, participants listened to their real time play back on a single Genelec 1029A active monitor placed three feet before them at chest level.

6.2 Methodology of Previous Work

While many related works rely on listening tests like MUSHRA to discern the fidelity of one model vs another, they differ from this thesis in two major ways [Vanhatalo et al, 2022]. First, many publications in this field focus on just one emulation method, white or black [Wright & Välimäki, 2019; Juvela et al, 2024]. For white box, they typically compare the fidelity of different types of techniques, like MNA versus Wave Digital filters, for example [Dempwolf, 2018]. The efficacy of these techniques is often tied to how well they can model nonlinear effects, much like this thesis [Mačák, 2012]. For black box publications, these papers typically focus on comparing different hyperparameters or slight changes in model architecture [Wright et al, 2020]. This thesis seeks to compare not minute differences, but large scale emulation techniques.

If there is to be any push toward further real time digitization in the future, it is imperative to collect data on how it actually feels for guitarists to play through these emulations. Theoretically, guitarists are much more in tune with their own instruments and would be able to accurately discern differences in sonic quality that a non-player might miss.

6.3 Results

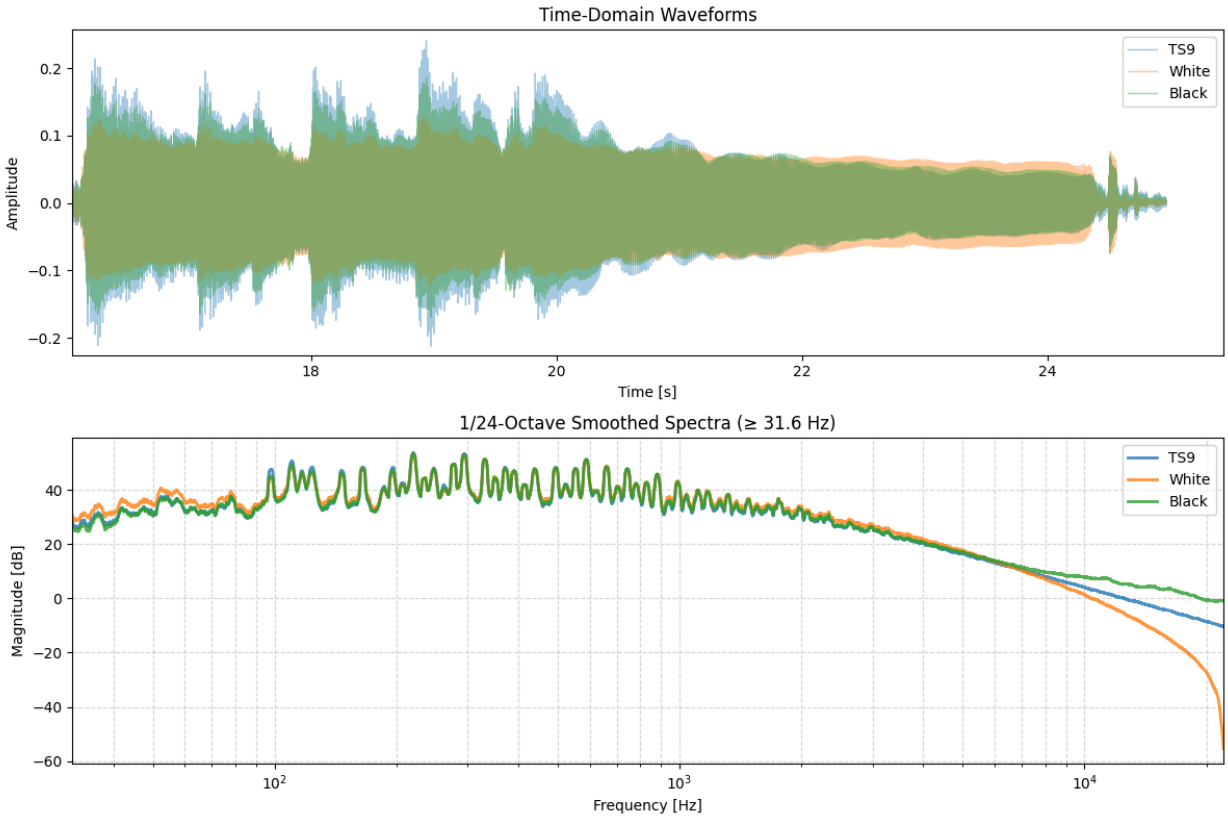


Figure 6.1: Recreation of waveform for each plugin (top). FFT transform of the same audio to show frequency response of each plugin compared to original, smoothed at 1/24-octave (bottom).

As seen in Fig 6.1, both the white and black box plugins were able to recreate general patterns of the predicted waveform. The resulting FFT spectral response shows high accuracy in the mid range frequencies, but divergence in the low and high end of the spectrum. Focusing on the high end, the white box plugin attenuates these frequencies in a sharp dropoff indicative of a low pass filter, while the black box actually boosts them, but maintains the same dropoff pattern as the original pedal. Zooming in on the black box predicted waveforms in Fig 6.2, the general pattern is matched, but the neural network struggles to predict the fine behaviour of the pedal, especially at sharp peaks and valleys in the amplitude.

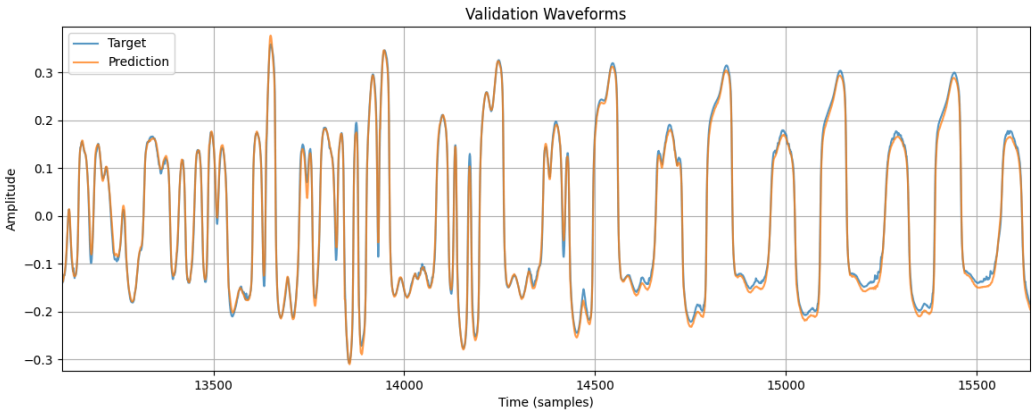


Figure 6.2: Zoomed in waveform of target and prediction from validation dataset for the neural network.

MUSHRA Listening Test Results

Distortion Level	White Box	Black Box
0.25	74.50	64.75
0.75	48.83	67.25
1.0	60.00	61.17

MUSHRA Playing Test Results

Distortion Level	White Box	Black Box
0.25	57.50	50.90
0.75	52.50	63.00
1.0	56.10	57.30

Figure 6.3: Result averages for both the listening (top) and playing (bottom) tests.

The results of both the listening and playing tests indicate a slight preference for the black box model, especially at 0.75 distortion (Fig 6.3). However, before the results were analyzed, participants who were unable to accurately label the hidden reference as the closest sounding option and the clean signal as the least faithful option were removed from the dataset. Starting with the lowest distortion level, the white box was deemed more accurate at recreating the pedal by ~10% and ~7% in the listening and playing tests respectively. However, 0.75 distortion shows a dramatic reversal, with the black box plugin deemed more accurate by about ~19% in the listening test and ~10% in the playing test. Lastly for the distortion all the way up, the blackbox plugin was seen as more accurate in both tests by ~1%. Lastly for the playing tests, both plugins were run in Reaper, where no latency was reported even with a buffer size of only 32 samples, showing that both plugins are real time safe.

MUSHRA Averaged Test Results

	White Box	Black Box
Listening Test	61.11	64.39
Playing Test	55.37	57.07

Figure 6.4: Averaged tests results for both listening and playing tests across all distortion levels.

The best performance for the listening test was the white box at 0.25 distortion with 74.50% accuracy; the best performance for the playing test was the black box at 0.75 distortion with 63.00% accuracy. These results demonstrate that neither model was a perfect emulation of the TS9 pedal. Averaging the results of all the distortion levels, the black box method was seen as slightly more accurate for both tests (Fig 6.4). Both plugins were also viewed as slightly more accurate in the listening tests compared to the playing tests, which may result from the guitarists having more familiarity, and so scrutiny, with distortion sounds.

6.4 Discussion

The neural network created an overall more accurate emulation, but neither plugin proved perfect. While the white box method showed initial promise with low distortion levels, its edge was swiftly lost as more distortion was introduced. However, there was the least amount of difference in the plugins' subjective accuracy with maximum distortion, which may be a factor of either the plugins sounding excellent or overall difficulty in distinguishing minute spectral differences amidst overwhelming distortion. The spectral responses (Fig. 6.1) show a stark divide in how each model affects the high end, with neither matching the pedal perfectly. It seems that from this spectral response, the testing results, and close inspection of the waveforms, both techniques were able to accurately emulate the general patterns and feel of a distortion, but were unable to perfectly mimic the TS9.

While incredibly subjective, conversation with the playing participants both during and after the test corroborated this claim. Almost every participant described both plugins as decent standalone distortion or overdrive effects, but neither held up well as emulations against direct testing with the actual plugin. Many participants actually likened their sounds to other well known overdrive pedals, especially the Boss Blues Driver. Participants whose playing style was more subdued, often involving light finger picking, expressed the need to play more intensely to elicit more accurate responses from the plugins. Still, the black box was viewed as slightly more accurate – some participants claimed that it had a more natural feel, alluding to the line noise and buzzing that was present both in it and the pedal. Others thought the white box plugin seemed gated and had an almost too perfect response to silence, which might be caused by its idealized nature.

6.5 Weaknesses

There are weaknesses and limitations in both the models and in the principal researcher that may affect this thesis' methodology and design. First, both plugins represent purely white and black box techniques as demonstrated by top researchers in the field and were not tailored exactly to the TS9 sound. The purpose of using these objectively white and black box methods hopes to show which pure technique, as described by professionals, creates a closer baseline

emulation of distortion. There is no doubt that tweaking each system in a variety of manners could create a more accurate representation of the pedal.

The white box technique may have been limited by the schematics. The schematics provided by Electrosmash are recognized as quite accurate throughout the industry, but that does not mean that the component values actually match those in the pedal used [Tarr, 2021]. This white box emulation creates an accurate *ideal* representation of the TS9 clipping and tone stage, but to create a more accurate representation of *the actual pedal used for testing*, each component in these subcircuits would have to be measured. Furthermore, the schematics used were from the original TS808 while the TS9 pedal was modeled. While the clipping and tone stages are theoretically identical, the final output signal may be influenced by other subcircuits that were not considered.

Originally, a TS mini was used for this thesis, not a TS9. However, this pedal proved faulty, which highlights a major pitfall of the neural network emulation. This technique provides a decently accurate emulation of the hardware it is trained on. However, if that hardware is faulty, then the emulation will also sound faulty. While this point may seem obvious, it shows that this method is nearly useless without proper access to the hardware, or at least a dataset.

Lastly, and most significantly, the results of this thesis were limited by the ability of the principal researcher. While both of the emulation techniques are very much in use within the virtual analog modeling industry, they were chosen for their learnability. The researcher had no previous virtual analog modeling experience. While the support of their supervisor was immensely helpful, they too had limited modeling experience, so no code reviews were conducted. While both plugins offer a decent distortion sound, the math and development behind them have not been reviewed properly.

6.6 Future Work

There is a stark difference in how easily both plugin methodologies can be updated for greater accuracy. For the white box method, trial and error of component value tweaks can offer slight improvements in emulation fidelity, but wholesale changes to the DK-Method would be needed to see major changes. For the black box model, however, improvements can be implemented with little effort. For example, end-to-end modeling suggests a benchmark of 10

discrete positions per knob. Because of both limited time and computational resources, only 5 knob positions were used for this thesis. By including a larger dataset this way, improvements can be expected. Furthermore, greater amounts of hidden units, training times, different preemphasis filter coefficients, and a more varied dataset can all be expected to better recreate the sound of the pedal.

Not only does using neural networks provide a more accurate model than the component based one, but it is a boon for developers for its efficiency and reusability. Rather than spend time solving complex systems of equations and then implementing them into C++, a developer need only create a two layer RNN and import its weights using RTNeural. While there are some quirks regarding truncated BPTT and complicated loss functions, there is not as much room for error compared to deriving a system of equations through various domain transforms. More importantly, once the developer has created a working neural network, it can be reused for similar sounding effects. Circuit modeling requires a boutique solution specific to each circuit that is often not modular. However, neural networks are agnostic to this specificity and can emulate similar *sounding* effects with little changes, even if the hardware design of those effects is wildly different. This reusability significantly cuts development time down, while reducing potential for mistakes and creating a more accurate emulation.

If neural networks do all of these things, why bother with circuit modeling anymore? To take a slightly philosophical approach, they may be easier for most development situations, but do not make a better developer. Their black box nature does not allow the developer to understand how effects are made, which would offer them a deeper understanding of DSP as a whole. This understanding may be irrelevant if pure emulation is desired, but is necessary if that developer wishes to make their own effects and to create something different, new, and exciting!

7 Conclusion

This thesis sought to answer if white or black box methods are more accurate in emulating guitar distortion, especially in real time. It did so by creating two VST plugins – one using the DK-Method based in Nodal Analysis and the other using RNN's and Deep Learning. Both techniques were chosen for their proven efficacy in virtual analog modeling. Subjective listening and playing tests were then used to see which plugin was a more faithful emulation of a

TS9 pedal. In nearly all cases, the black box plugin proved to be slightly more accurate. Throughout the development process it was also discovered how this black box methodology can be used to create emulations much more quickly and efficiently.

These results demonstrate that the emergence of audio effect companies promoting their neural products is not a simple marketing ploy. Ultimately, however, trying to compare the efficacy of white vs black box methods is incredibly difficult given the amount of variation within the techniques in each category, the amount of parameters involved, and the ability of the developer. Regardless, it is expected that neural networks will continue their rise in popularity within the virtual analog modeling world because of their accuracy, relative ease of use, and reusability.

Works Cited

[Anderson, 2020] Anderson, I.Z., 2020. *Evaluating Electrolytic Capacitors Specified for Audio Use: A Comparative Analysis of Electrical Measure-ments and Capacitor Distortion Products in Line Level Interstage Coupling Applications.* Journal of the Audio Engineering Society, 68(7/8), pp.559-567.

[Bencina, 2011] Bencina, R., 2013. *Real-time audio programming 101: time waits for nothing.* Aug 2011. URL http://www.rossbencina.com/. Last retrieved in the 28th of September.

[Bengio et al, 2015] Bengio, Y., Goodfellow, I. and Courville, A., 2017. Deep learning. (Vol. 1, pp. 23-24). Cambridge, MA, USA: MIT press.

[Benois, 2013] Benois, P.R., 2013. *Simulation framework for analog audio circuits based on nodal DK method*. Master's thesis, Helmut Schmidt University, Hamburg, Germany.

[Bernadini et al, 2021] Bernardini, A., Bozzo, E., Fontana, F. and Sarti, A., 2021. *A wave digital Newton-Raphson method for virtual analog modeling of audio circuits with multiple one-port nonlinearities.* IEEE/ACM Transactions on Audio, Speech, and Language Processing, 29, pp.2162-2173.

[Bognar, 2022] Bognár, P., 2022. *Audio effect modeling with deep learning methods*. Doctoral dissertation, Wien.

[Bourdin et al, 2025] Bourdin, Y., Montbonnot-Saint-Martin, F., Team, A.I., Bordeaux, I., Legrand, P., ENSC, B.I. and Roche, F., 2025. *EMPIRICAL RESULTS FOR ADJUSTING TRUNCATED BACKPROPAGATION THROUGH TIME WHILE TRAINING NEURAL AUDIO EFFECTS.*

[Chen, 2024] Chen, N. 2024. *Exploring the development and application of LSTM variants.* Applied and Computational Engineering. 53. 103-107. 10.54254/2755-2721/53/20241288.

[Chowdhury, 2019] Chowdhury, J. 2019. *Bad Circuit Modelling Episode 1: Component Tolerances.* Medium.

[Chowdhury, 2020] Chowdhury, J., 2020. *A comparison of virtual analog modelling techniques for desktop and embedded implementations.* arXiv preprint arXiv:2009.02833.

[Chowdhury, 2021] Chowdhury, J., 2021. *Rtneural: Fast neural inferencing for real-time systems*. arXiv preprint arXiv:2106.03037.

[Chowdhury, 2023] Chowdhury, J. 2023. *Some Thoughts on Virtual Analog Modelling.* Medium.

[Comunitá, 2025] Comunità, M., Steinmetz, C.J. and Reiss, J., 2025. *Differentiable black-box and gray-box modeling of nonlinear audio effects.* Frontiers in Signal Processing, 5, p.1580395.

[Covert & Livingston, 2013] Covert, J. and Livingston, D.L., 2013, April. *A vacuum-tube guitar amplifier model using a recurrent neural network.* In 2013 Proceedings of IEEE Southeastcon (pp. 1-5). IEEE.

[Damskägg et al, 2019a] Damskägg, E.P., Juvela, L. and Välimäki, V., 2019. *Real-time modeling of audio distortion circuits with deep learning.* In Sound and music computing conference (pp. 332-339). Sound and Music Computing Association.

[Damskägg et al, 2019b] Damskägg, E.P., Juvela, L., Thuillier, E. and Välimäki, V., 2019, May. *Deep learning for tube amplifier emulation.* In ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) (pp. 471-475). IEEE.

[Dempwolf & Zolzer, 2011] Dempwolf, K. & Zölzer, U. 2011, *Discrete State-Space Model of the Fuzz-Face.* Proceedings of Forum Acusticum, Aalborg, Denmark (European Acoustics Association).

[Dempwolf, 2018] Dempwolf, K. 2018. *Authentic Modeling of Guitar Amplifiers and Effects Boxes.* 44th Annual Convention on Acoustics (DAGA 2018), pp. 729 - 732.

[Dolphin, 2020] Dolphin, R. 2020. *LSTM Networks | A Detailed Explanation*. Toward Data Science.

[Eichas et al, 2014] Eichas, F., Fink, M., Holters, M. and Zölzer, U., 2014, September. *Physical Modeling of the MXR Phase 90 Guitar Effect Pedal.* In DAFx (pp. 153-158).

[Eichas et al, 2017] Eichas, F., Möller, S. and Zölzer, U., 2017, September. *Block-oriented gray box modeling of guitar amplifiers.* In Proceedings of the International Conference on Digital Audio Effects (DAFx), Edinburgh, UK (pp. 5-9).

[Electrosmash, 2025] Electrosmash, 2025. *Tube Screamer Analysis.* Electrosmash.

[ElProcus, 2025] ElProcus, 2025. *What is a Resistor? Construction, Circuit Diagram, and Applications.* Elprocus.

[Ersa Electronics, 2024] Ersa Electronics, 2024. *Basics of Capacitor: Capacitor Symbols*. Ersa Electronics.

[Esqueda et al, 2021] Esqueda, F., Kuznetsov, B. and Parker, J.D., 2021, September. *Differentiable white-box virtual analog modeling.* In 2021 24th International Conference on Digital Audio Effects (DAFx) (pp. 41-48). IEEE.

[Germain & Werner, 2017] F. G. Germain and K. J. Werner, 2017. *Optimizing differentiated discretization for audio circuits beyond driving point transfer functions*. IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA), New Paltz, NY, USA, 2017, pp. 384-388, doi: 10.1109/WASPAA.2017.8170060.

[Giampiccolo et al, 2025] Giampiccolo, R., Gafencu, S.C. and Bernardini, A., 2025. *Explicit Modeling of Audio Circuits with Multiple Nonlinearities for Virtual Analog Applications*. IEEE Open Journal of Signal Processing.

[Hayt, 2024] Hayt, W., et al, 2024. *Engineering Circuit Analysis*. McGraw-Hill.

[Herbst, 2018] Herbst, J.P., 2018. *Heaviness and the electric guitar: Considering the interaction between distortion and harmonic structures.* Metal Music Studies, 4(1), pp.95-113.

[Holters & Zölzer, 2011] Holters, M. & Zölzer, U. 2011. *Physical modeling of a wahwah effect pedal as a case study for application of the nodal dk method to circuits with variable parts.* Proc. Digital Audio Effects (DAFx-11), Paris, France, Sept. 19–23, pp. 31–35.

[Irwin & Elms, 2015] Irwin, J.D. and Nelms, R.M., 2020. *Basic engineering circuit analysis*. John Wiley & Sons.

[Juvela et al, 2024] Juvela, L., Damskägg, E.P., Peussa, A., Mäkinen, J., Sherson, T., Mimilakis, S.I., Rauhanen, K. and Gotsopoulos, A., 2023, June. *End-to-end amp modeling: from data to controllable guitar amplifier models.* In ICASSP 2023-2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) (pp. 1-5). IEEE.

[Karjalainen & Pakarinen, 2006] Karjalainen, M. and Pakarinen, J., 2006, *Wave digital simulation of a vacuum-tube amplifier*. In 2006 IEEE International Conference on Acoustics Speech and Signal Processing Proceedings (Vol. 5, pp. V-V). IEEE.

[Karki, 2021] Karki, J. and Amplifiers, A.O., 1998. *Understanding operational amplifier specifications*. Texas Instruments White Paper SLOA011.

[Kelleher, 2019] Kelleher, J.D., 2019. *Deep learning*. MIT press.

[Lambert, 2010] Lambert, M., 2010. *Plug-in Modelling: How Industry Experts Do It.*

[Li, 2022] Li, X., 2022. *Differentiable White Box Modeling of Moog VC.* Thesis, MSc Acoustics and Music Technology, University of Edinburgh.

[Mačák, 2012] Mačák, J., 2012. *Real-time digital simulation of guitar amplifiers as audio effects.* Doctoral dissertation, Ph. D. thesis, Brno University of Technology, Brno.

[Mienye et al, 2024] Mienye, I.D., Swart, T.G. and Obaido, G., 2024. *Recurrent neural networks: A comprehensive review of architectures, variants, and applications.* Information, 15(9), p.517.

[Miklánek et al, 2023] Miklanek, S., Wright, A., Välimäki, V. and Schimmel, J., 2023, September. *Neural grey-box guitar amplifier modelling with limited data*. In International Conference on Digital Audio Effects (pp. 151-158). Aalborg University.

[Mohan, 2011] Mohan, N. 2011. *Electric Machines and Drives.* Wiley.

[Neural DSP, 2024] Neural DSP, 2024. *Neural DSP Amplifier Modeling Technology.* Neural DSP.

[Next PCB, 2025] Next PCB, 2025. *Electrical & Electronic Symbols: A Basic Introduction with Chart.* Next PCB.

[Düvel et al, 2020] Düvel, N., Kopiez, R., Wolf, A. and Weihe, P., 2020. *Confusingly similar: Discerning between hardware guitar amplifier sounds and simulations with the Kemper Profiling Amp.* Music & Science, 3, p.2059204320901952.

OpenAI ChatGPT (2025) ChatGPT response to Tom Garvey, June.

[Oord et al, 2016] Van Den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A. and Kavukcuoglu, K., 2016. *Wavenet: A generative model for raw audio.* arXiv preprint arXiv:1609.03499, 12, p.1.

[Pandey, 2024] Pandey, S. 2024. *Basics of Op-Amp (Operational Amplifier).* 10.13140/RG.2.2.20049.10082.

[Pieter, 2024] Pieter, P. 2024. *Bilinear Transform.*

[Poss, 1998] Poss, R.M., 1998. *Distortion is truth.* Leonardo Music Journal, pp.45-48.

[Ramírez & Weiss, 2019] Ramírez, M.A.M. and Reiss, J.D., 2019. *Modeling nonlinear audio effects with end-to-end deep neural networks.* In ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) (pp. 171-175). IEEE.

[Ramírez et al, 2020] Martínez Ramírez, M.A., Benetos, E. and Reiss, J.D., 2020. *Deep learning for black-box modeling of audio effects*. Applied Sciences, 10(2), p.638.

[Random Nerd Tutorials, 2025] Random Nerd Tutorials, 2025. *Electronics Basics – How a Potentiometer Works*. Random Nerd Tutorials.

[Saber, 2020] Saber, D., 2020. *Theoretical Analysis and Design of Analog Distortion Circuitry.*

[Sherstinsky, 2020] Sherstinsky, A, 2020. *Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network.* Physica D: Nonlinear Phenomena, Volume 404, 132306.

[Simper, 2013] Simper A. 2013. *Direct Numerical Integration of a One Pole Linear Low Pass Filter*. Cytomic.

[Simper, 2020] Simper, A. 2020. *From Circuit to Code: Under the Hood of Analog Modelling*. ADC20.

[Smith, 2010] Smith, J.O., 2010. *Physical Audio Signal Processing for Virtual Musical Instruments And Audio Effects*. CCRMA, Stanford.

[Spotfire, 2025] Spotfire, 2025. *What is a neural network?* Spotfire.

[Staudemeyer & Morris, 2019] Staudemeyer, R.C. and Morris, E.R., 2019. *Understanding LSTM--a tutorial into long short-term memory recurrent neural networks.* arXiv preprint arXiv:1909.09586.

[Steinmetz & Reiss, 2020] Steinmetz, C.J. and Reiss, J.D., 2020, December. *auraloss: Audio focused loss functions in PyTorch.* In Digital music research network one-day workshop (DMRN+ 15).

[Steinmetz & Reiss, 2021] Steinmetz, C.J. and Reiss, J.D., 2021. *Efficient neural networks for real-time analog audio effect modeling*. arXiv preprint arXiv:2102.06200.

[Stewart-Frazier Tools, 2020] Stewart-Frazier Tools, 2020. *Operational Amplifiers.* Stewart-Frazier Tools.

[Südholt et al, 2022] Südholt, D., Wright, A., Erkut, C. and Välimäki, V., 2022. *Pruning deep neural network models of guitar distortion effects*. IEEE/ACM Transactions on Audio, Speech, and Language Processing, 31, pp.256-264.

[T.J.J, 2020] T.J.J, 2020. *LSTMs Explained: A Complete, Technically Accurate, Conceptual Guide with Keras*. Medium.

[Tarr, 2021] Tarr, E. 2021. *Audio Circuit Modeling Tutorial.* Hack Audio.

[Tarr, 2022] Tarr, E., 2022. *Digital models of analog circuits for musical audio production: A review of techniques and library for automated circuit solving*. The Journal of the Acoustical Society of America, 152(4_Supplement), pp.A179-A179.

[Thompson, 2014] Thompson, M., 2014. *Intuitive analog circuit design*. Elsevier.

[Thurston, 1960] M. O. Thurston, 1960. *Diodes*. in IRE Transactions on Education, vol. 3, no. 4, pp. 128-133 doi: 10.1109/TE.1960.4322153.

[Vanhatalo et al, 2022] Vanhatalo, T., Legrand, P., Desainte-Catherine, M., Hanna, P., Brusco, A., Pille, G. and Bayle, Y., 2022. *A review of neural network-based emulation of guitar amplifiers.* Applied Sciences, 12(12), p.5894.

[Werbos, 1990] Werbos, P.J., 2002. *Backpropagation through time: what it does and how to do it.* Proceedings of the IEEE, 78(10), pp.1550-1560.

[Werner et al, 2015] Werner, K.J., Nangia, V., Bernardini, A., Smith III, J.O. and Sarti, A., 2015, October. *An improved and generalized diode clipper model for wave digital filters*. In Audio Engineering Society Convention (Vol. 139).

[Wright & Välimäki, 2019] Wright, A. and Välimäki, V., 2020, May. *Perceptual loss function for neural modeling of audio systems.* In ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) (pp. 251-255). IEEE.

[Wright et al, 2019] Wright, A., Damskägg, E.P. and Välimäki, V., 2019, September. *Real-time black-box modelling with recurrent neural networks*. In International Conference on Digital Audio Effects. University of Birmingham.

[Wright et al, 2020] Wright, A., Esqueda, F., Juvela, L. & Välimäki, V. 2020, *Real-Time Guitar Amplifier Emulation with Deep Learning*. Applied Sciences, vol. 10(3), pp. 766.

[Wright, 2023] Wright, A., 2023. *Neural modelling of audio effects.*

[Yeh & Smith, 2008] Yeh, D.T. and Smith, J.O., 2008. *Simulating guitar distortion circuits using wave digital and nonlinear state-space formulations.* Proc. Digital Audio Effects (DAFx-08), Espoo, Finland, pp.19-26.

[Yeh et al, 2010] Yeh, D.T., Abel, J.S. and Smith, J.O., 2009. *Automated physical modeling of nonlinear audio circuits for real-time audio effects—Part I: Theoretical development.* IEEE transactions on audio, speech, and language processing, 18(4), pp.728-737.

[Yeh, 2009] Yeh, D.T.M., 2009. *Digital implementation of musical distortion circuits by analysis and simulation*. Stanford University.

[Yildiz & Kelebekler, 2013] Yildiz, A.B. and Kelebekler, E., 2013. *A Simplified Approach to Analyze of Active Circuits Containing Operational Amplifiers.* Electronic Components and Materials, 43(1), pp.67-73.

[Yu et al, 2019] Yong Yu, Xiaosheng Si, Changhua Hu, Jianxun Zhang, 2019. *A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures.* Neural Comput 31 (7): 1235–1270.

[Zhang et al, 2018] Zhang, Z., Olbrych, E., Bruchalski, J., McCormick, T.J. and Livingston, D.L., 2018, April. *A vacuum-tube guitar amplifier model using long/short-term memory networks*. In SoutheastCon 2018 (pp. 1-5). IEEE.

Appendix A

All code can be found in the following two Github Repositories. The black box model code is included below, but it is suggested that the reader views it in Github for easier reading.

White Box: https://github.com/erictarrbelmont/TSPedal

Black Box: https://github.com/tgarvs/NeuralScreamer

A.1 Preprocessing.py

```
import os
import librosa
import numpy as np
import soundfile as sf
from sklearn.utils import shuffle
import matplotlib.pyplot as plt


PATH_TO_AUDIO = "./audio"
SAMPLE_RATE = 44100
CHUNK_SIZE = SAMPLE_RATE // 2  #0.5 seconds
HOP_SIZE = CHUNK_SIZE  # non-overlapping
SAVE_PATH = "./audio/postproc/test-dataset.npz"
LOAD_SUBDIR = "preproc"
EXPORT_DIR_INPUTS = "./audio/chunks/inputs"
EXPORT_DIR_TARGETS = "./audio/chunks/targets"
MAX_MISALIGNMENT_SAMPLES = 55  # Maximum allowed misalignment in samples


os.makedirs(EXPORT_DIR_INPUTS, exist_ok=True)
os.makedirs(EXPORT_DIR_TARGETS, exist_ok=True)
os.makedirs(os.path.dirname(SAVE_PATH), exist_ok=True)



# ── COLLECT FILES ──────────────────────────────────────────────────
input_files = {}
target_files = {}
```

```
for root, _, files in os.walk(PATH_TO_AUDIO):
   if os.path.basename(root) == LOAD_SUBDIR:
       for fname in files:
           if fname.endswith("-input.wav"):
               input_files[fname] = os.path.join(root, fname)
           elif fname.endswith("-target.wav"):
               target_files[fname] = os.path.join(root, fname)



# ── PROCESS ──────────────────────────────────────────────────────────
inputs = []
targets = []
chunk_idx = 0

#Cycle through all file names
for in_fname, in_path in sorted(input_files.items()):

   #Search for target that corresponds with each input
   tgt_fname = in_fname.replace("-input.wav", "-target.wav")
   print(f"Processing {in_fname} and {tgt_fname}")
   if tgt_fname not in target_files:
       print(f"Missing target for: {in_fname}")
       continue

   # Extract audio signal
   try:
       x, _ = librosa.load(in_path, sr=SAMPLE_RATE, mono=True)
       y, _ = librosa.load(target_files[tgt_fname], sr=SAMPLE_RATE, mono=True)
   except Exception as e:
       print(f"Error loading {in_fname}: {e}")
       continue


   # Extract knob position
   knob_position = in_fname.split("-")[1]

   # Trim signals so chunks are the perfect length
   L = min(len(x), len(y))
   if L < CHUNK_SIZE:
       continue
```

```
    x = x[:L]
    y = y[:L]

    # Normalize globally
    peak = max(np.max(np.abs(x)), np.max(np.abs(y)))
    x /= peak
    y /= peak


    #PLOT1: Checking the overall waveforms of both original big signals
    plt.figure(figsize=(14, 5))
    plt.plot(x, label='Input', alpha=0.50)
    plt.plot(y, label='Target', alpha=0.50)
    plt.title("Input vs Target Waveform")
    plt.xlabel("Time (samples)")
    plt.ylabel("Amplitude")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()


    # Chunk
    for start in range(0, L - CHUNK_SIZE + 1, HOP_SIZE):
        x_chunk = x[start:start + CHUNK_SIZE]
        y_chunk = y[start:start + CHUNK_SIZE]

        # ── MISALIGNMENT CHECK ──────────────────────────────────────
        corr = np.correlate(x_chunk, y_chunk, mode='full')
        lag = np.argmax(corr) - (len(x_chunk) - 1)
        if abs(lag) > MAX_MISALIGNMENT_SAMPLES:
            print(f"Skipped chunk at sample {start} due to misalignment (lag={lag}
samples)")
            continue  # Skip misaligned chunk

        #Probably don't need to export the 0.5 seconds chunks, but still kinda nice to
hear them
        chunk_idx += 1
        sf.write(f"{EXPORT_DIR_INPUTS}/input_{chunk_idx:05d}.wav", x_chunk,
SAMPLE_RATE)
        sf.write(f"{EXPORT_DIR_TARGETS}/target_{chunk_idx:05d}.wav", y_chunk,
SAMPLE_RATE)
```

```
        kn = np.full(CHUNK_SIZE, knob_position) #make another array the same size as
the chunk but filled with knob pos
        x_chunk = np.stack([x_chunk, kn], axis=-1) #stack knob array and x_chunk so
that each 0.5 second chunk has its own personal knob position


        #Append each 0.5 second chunk to a bigger list
        inputs.append(x_chunk)
        targets.append(y_chunk)



if not inputs:
    print("No valid chunks found.")
    exit(1)

# ── SAVE ──────────────────────────────────────────────────────────────
X = np.array(inputs, dtype=np.float32)#[..., np.newaxis]
Y = np.array(targets, dtype=np.float32)[..., np.newaxis]


print(f"Final dataset: X={X.shape}, Y={Y.shape}")
np.savez(SAVE_PATH, inputs=X, targets=Y)
print(f"Saved dataset to: {SAVE_PATH}")
```

A.2 model.py

```
"""
@author Thomas Garvey
@date August 11, 2025
```

```
@file model.py
@brief A RNN to model guitar distortion. The architecture for this model is given in
https://dafx.de/paper-archive/2019/DAFx2019_paper_43.pdf
"""



import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, optimizers
from tqdm.auto import tqdm
from RTNeural.python.model_utils import save_model
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle

DATA_PATH = './audio/postproc/test-dataset.npz'

# ------------------------------------------------
# Data Loading (ensure shape (N, T, 1))
# ------------------------------------------------
def load_dataset(path):
    data = np.load(path)
    X = data['inputs'].astype('float32')
    y = data['targets'].astype('float32')

    #Uncomment to test truncated dataset during debugging
    # X = X[:5]
    # y = y[:5]

    print(f"Loaded Dataset Shape: X={X.shape}, Y={y.shape}")
    return X, y


def prepare_dataset(val_percentage=0.05):
    X, Y = load_dataset(DATA_PATH)
    X, Y = shuffle(X, Y, random_state=42) #sklearn.utils shuffle function keeps the
pairs synced
    print(f"In prepare data, X shape : {X.shape}")

    #split into training/validation
```

```
    X_train, X_validation, Y_train, Y_validation = train_test_split(
        X, Y, test_size=val_percentage, shuffle=False)

    print(f"In prepare data, X_train shape AFTER traintestsplit : {X_train.shape}")
    print(f"In prepare data, X_val shape AFTER traintestsplit : {X_validation.shape}")

    return X_train, Y_train, X_validation, Y_validation



# ------------------------------------------------
# Loss Functions
# ------------------------------------------------

#High pass filter to boost high end --> good for high harmonic distortion
def pre_emphasis_filter(x, coeff=0.85):
 return tf.concat([x[:, 0:1, :], x[:, 1:, :] - coeff*x[:, :-1, :]], axis=1)

#DC offset
def dc_loss(target_y, predicted_y):
    return tf.reduce_mean(tf.square(tf.reduce_mean(target_y) -
tf.reduce_mean(predicted_y))) / tf.reduce_mean(tf.square(target_y))

# Error-to-signal loss
def esr_loss(target_y, predicted_y, emphasis_func=lambda x : x):
    target_yp = emphasis_func(target_y)
    pred_yp = emphasis_func(predicted_y)
    return tf.reduce_sum(tf.square(target_yp - pred_yp)) /
tf.reduce_sum(tf.square(target_yp))

#Final loss function put together
def combined_loss(y_true, y_pred):
    return esr_loss(y_true, y_pred) + dc_loss(y_true, y_pred)



# ------------------------------------------------
# Build Stateful Model
# ------------------------------------------------
def build_model(batch_size):
    tf.keras.backend.clear_session()
    m = tf.keras.Sequential([
        layers.Input(batch_shape=(batch_size, None, 2)), #two inputs used
```

```
        layers.LSTM(64, return_sequences=True, stateful=True, name='stateful_lstm'),
        layers.Dense(1, activation=None),
    ])
    return m

# ------------------------------------------------
# Build Stateless Inference Engine for Validation
# ------------------------------------------------
def build_inference_model():
    tf.keras.backend.clear_session()
    m = tf.keras.Sequential([
        layers.Input(shape=(None,2)), #two inputs used
        layers.LSTM(64, return_sequences=True, stateful=False, name='stateful_lstm'),
        layers.Dense(1, activation=None),
    ])
    return m

# ------------------------------------------------
# Training Loop Function
# ------------------------------------------------
def train_model(model, in_batches, out_batches, in_val, out_val, optimizer, epochs,
warmup_len, segment_len, inf):

    #Lists to hold loss results for later graphing
    train_history = []
    val_history = []

    for ep in range(epochs):
        print(f"\n MiniBatch Epoch {ep+1}/{epochs}")
        batch_losses = []

        #shuffle the training set at the start of each epoch
        in_batches, out_batches = shuffle(in_batches, out_batches)

        for xb, yb in zip(in_batches, out_batches): #cycles through the batches until
the entire dataset is processed
            # print(f"shape of xb in train : {xb.shape}")
            model.get_layer('stateful_lstm').reset_states() #reset between each batch
            model(xb[:, :warmup_len, :]) #warm up before each batch

            #calculate loss per sequence
            loss = step_learn(model, xb, yb, optimizer, warmup_len, segment_len)
```

```
            batch_losses.append(loss.numpy()) #append all the sequence losses in one
list

        #average loss across the epoch
        avg_train = float(np.mean(batch_losses))
        train_history.append(avg_train)
        print(f" Avg Train Loss: {avg_train:.6f}")


        if ep % 2 == 0:
            # print(f"Shape of validation dataset per epoch: {in_val.shape}")

            inf.get_layer('stateful_lstm').reset_states()
            inf.set_weights(model.get_weights())
            y_pred_val = inf.predict(in_val, batch_size=32)
            # print("VAL PREDICTED")
            val_loss = combined_loss(out_val, y_pred_val).numpy()
            val_history.append(val_loss)
            print(f"  Val Loss: {val_loss:.6f}")

    return train_history, val_history


# ------------------------------------------------
# Single-step train (stateful). Separated out into a tf.function to run faster...not
exactly sure how this works here...
# ------------------------------------------------
@tf.function
def step_learn(model, x_seq, y_seq, optimizer, warmup, seglen):
    total_loss = tf.constant(0.0)
    count = tf.constant(0)

    for n in tf.range(warmup, tf.shape(x_seq)[1] - seglen, seglen):
        target = y_seq[:, n:n+seglen, :]

        with tf.GradientTape() as tape:
            pred = model(x_seq[:, n:n+seglen, :])
            loss = combined_loss(target, pred)

        grads = tape.gradient(loss, model.trainable_variables)
        grads, _ = tf.clip_by_global_norm(grads, 1.0) #clip gradient
```

```
        optimizer.apply_gradients(zip(grads, model.trainable_variables))
        total_loss += loss
        count += 1

    #returns average loss across all the batch
    return total_loss / tf.cast(tf.maximum(count, 1), tf.float32)


# ------------------------------------------------
# Plot Waveforms
# ------------------------------------------------
def check_waveform(model, X, y, label):
    for i in range(3):
        x_vis = X[i:i+1]
        y_vis = y[i].squeeze()
        y_pred = model.predict(x_vis, verbose=0).squeeze()
        plt.figure(figsize=(14,5))
        plt.plot(y_vis, label='Target', alpha=0.75)
        plt.plot(y_pred, label='Prediction', alpha=0.75)
        plt.title(label)
        plt.xlabel("Time (samples)")
        plt.ylabel("Amplitude")
        plt.legend()
        plt.grid(True)
        plt.show()

# ------------------------------------------------
# Main
# ------------------------------------------------
if __name__ == '__main__':
    # Load & split
    X_train, y_train, X_val, y_val = prepare_dataset()

    # Parameters
    BATCH_SIZE = 500
    warmup_len = 1000
    segment_len = 2048
    epochs = 500

    # Prepare train batches so theyre the exact shape with no wrong sized batches left
over
    NUM_TRAIN, NUM_SAMPLES, _ = X_train.shape
```

```
    n_batches = NUM_TRAIN // BATCH_SIZE

    X_trunc = X_train[:n_batches * BATCH_SIZE, :, :]
    y_trunc = y_train[:n_batches * BATCH_SIZE, :, :]
    in_batches  = X_trunc.reshape((n_batches, BATCH_SIZE, NUM_SAMPLES, 2)) #i.e. 2
batches of 40, each with 22050 samples and 1 feature
    out_batches = y_trunc.reshape((n_batches, BATCH_SIZE, NUM_SAMPLES, 1))

    # Prepare validation data
    IN_val  = X_val # .reshape((X_val.shape[0], NUM_SAMPLES, 2))
    OUT_val = y_val # .reshape((y_val.shape[0], NUM_SAMPLES, 1))


    # Build model and optimizer
    model = build_model(batch_size=BATCH_SIZE)
    inf = build_inference_model()
    optimizer = optimizers.Adam(5e-4)

    # Train
    train_hist, val_hist = train_model(model, in_batches, out_batches, IN_val, OUT_val,
optimizer, epochs, warmup_len, segment_len, inf)

    # Save model
    save_model(model, './model_export/ts_nine.json')


    # Plot training history
    plt.plot(train_hist, label='Train Loss')
    plt.plot(list(range(0, epochs, 2)), val_hist, label='Val Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Inference waveform checks
    infer = build_inference_model()
    infer.set_weights(model.get_weights())
    check_waveform(infer, X_val,   y_val,   "Validation Waveforms")
```

A.3 PluginProcessor.h (JUCE)

```
/*

==============================================================================
    This file contains the basic framework code for a JUCE plugin processor.

==============================================================================
*/
#pragma once
#define RTNEURAL_USE_EIGEN 1
#define RTNEURAL_DEFAULT_ALIGNMENT 16
#include <JuceHeader.h>
#include "RTNeural.h"
//==============================================================================
/**
*/
class Two_inputAudioProcessor  : public juce::AudioProcessor
{
public:

//==============================================================================
    Two_inputAudioProcessor();
    ~Two_inputAudioProcessor() override;

//==============================================================================
    void prepareToPlay (double sampleRate, int samplesPerBlock) override;
    void releaseResources() override;
   #ifndef JucePlugin_PreferredChannelConfigurations
    bool isBusesLayoutSupported (const BusesLayout& layouts) const override;
   #endif
    void processBlock (juce::AudioBuffer<float>&, juce::MidiBuffer&) override;

//==============================================================================
    juce::AudioProcessorEditor* createEditor() override;
    bool hasEditor() const override;

//==============================================================================
    const juce::String getName() const override;
    bool acceptsMidi() const override;
    bool producesMidi() const override;
    bool isMidiEffect() const override;
```

```
    double getTailLengthSeconds() const override;

//==============================================================================
    int getNumPrograms() override;
    int getCurrentProgram() override;
    void setCurrentProgram (int index) override;
    const juce::String getProgramName (int index) override;
    void changeProgramName (int index, const juce::String& newName) override;

//==============================================================================
    void getStateInformation (juce::MemoryBlock& destData) override;
    void setStateInformation (const void* data, int sizeInBytes) override;

    juce::AudioProcessorValueTreeState apvts;
    juce::AudioProcessorValueTreeState::ParameterLayout createParams();
private:

//==============================================================================
//    RTNeural::ModelT<float, 2, 64,
//        RTNeural::Conv1DT<float, 2, 36, 12, 1>,
//        RTNeural::Conv1DT<float, 36, 36, 12, 1>,
//        RTNeural::LSTMLayerT<float, 36, 96>,
//        RTNeural::DenseT<float, 96, 64>
//    > neuralNet[2];

    RTNeural::ModelT<float, 2, 1,
    // LSTM<inSz,hiddenSz>
    RTNeural::LSTMLayerT<float, 2, 64>,
    // Dense<inSz,outSz>
    RTNeural::DenseT<float, 64, 1>
    > neuralNet[2];


    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (Two_inputAudioProcessor)
};
```

A.4 PluginProcessor.cpp (JUCE)

```
==============================================================================
*/
#include "PluginProcessor.h"
```

```
#include "PluginEditor.h"
//==============================================================================
Two_inputAudioProcessor::Two_inputAudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
    : AudioProcessor (BusesProperties()
                     #if ! JucePlugin_IsMidiEffect
                      #if ! JucePlugin_IsSynth
                       .withInput  ("Input",  juce::AudioChannelSet::stereo(), true)
                      #endif
                       .withOutput ("Output", juce::AudioChannelSet::stereo(), true)
                     #endif
                       ), apvts(*this, nullptr, "Parameters", createParams())
#endif
{

//    juce::MemoryInputStream jsonStream (BinaryData::ts_mini_json, BinaryData::ts_mini_jsonSize, false);
//    auto jsonInput = nlohmann::json::parse (jsonStream.readEntireStreamAsString().toStdString());
//    neuralNet[0].parseJson (jsonInput);
//    neuralNet[1].parseJson (jsonInput);

    juce::MemoryInputStream jsonStream (BinaryData::ts_nine_json, BinaryData::ts_nine_jsonSize, false);
    auto jsonInput = nlohmann::json::parse (jsonStream.readEntireStreamAsString().toStdString());
    neuralNet[0].parseJson (jsonInput);
    neuralNet[1].parseJson (jsonInput);

}
Two_inputAudioProcessor::~Two_inputAudioProcessor()
{
}
//==============================================================================
const juce::String Two_inputAudioProcessor::getName() const
{
    return JucePlugin_Name;
}
bool Two_inputAudioProcessor::acceptsMidi() const
{
   #if JucePlugin_WantsMidiInput
    return true;
   #else
    return false;
   #endif
}
bool Two_inputAudioProcessor::producesMidi() const
{
   #if JucePlugin_ProducesMidiOutput
    return true;
   #else
```

```
    return false;
   #endif
}
bool Two_inputAudioProcessor::isMidiEffect() const
{
   #if JucePlugin_IsMidiEffect
    return true;
   #else
    return false;
   #endif
}
double Two_inputAudioProcessor::getTailLengthSeconds() const
{
    return 0.0;
}
int Two_inputAudioProcessor::getNumPrograms()
{
    return 1;   // NB: some hosts don't cope very well if you tell them there are 0 programs,
                // so this should be at least 1, even if you're not really implementing programs.
}
int Two_inputAudioProcessor::getCurrentProgram()
{
    return 0;
}
void Two_inputAudioProcessor::setCurrentProgram (int index)
{
}
const juce::String Two_inputAudioProcessor::getProgramName (int index)
{
    return {};
}
void Two_inputAudioProcessor::changeProgramName (int index, const juce::String& newName)
{
}
//==============================================================================
void Two_inputAudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)
{
    // Use this method as the place to do any pre-playback
    // initialisation that you need..
    neuralNet[0].reset();
    neuralNet[1].reset();
}
void Two_inputAudioProcessor::releaseResources()
{
    // When playback stops, you can use this as an opportunity to free up any
    // spare memory, etc.
}
#ifndef JucePlugin_PreferredChannelConfigurations
```

```
bool Two_inputAudioProcessor::isBusesLayoutSupported (const BusesLayout& layouts) const
{
  #if JucePlugin_IsMidiEffect
    juce::ignoreUnused (layouts);
    return true;
  #else
    // This is the place where you check if the layout is supported.
    // In this template code we only support mono or stereo.
    // Some plugin hosts, such as certain GarageBand versions, will only
    // load plugins that support stereo bus layouts.
    if (layouts.getMainOutputChannelSet() != juce::AudioChannelSet::mono()
     && layouts.getMainOutputChannelSet() != juce::AudioChannelSet::stereo())
        return false;
    // This checks if the input layout matches the output layout
   #if ! JucePlugin_IsSynth
    if (layouts.getMainOutputChannelSet() != layouts.getMainInputChannelSet())
        return false;
   #endif
    return true;
  #endif
}
#endif
void Two_inputAudioProcessor::processBlock (juce::AudioBuffer<float>& buffer, juce::MidiBuffer&
midiMessages)
{
    juce::ScopedNoDenormals noDenormals;
    auto d {apvts.getRawParameterValue("DRIVE")};
    auto drive = d-> load();

    auto v {apvts.getRawParameterValue("VOLUME")};
    auto volume = v->load();

    for (int ch = 0; ch < buffer.getNumChannels(); ++ch)
    {
        auto* x = buffer.getWritePointer (ch);
        for (int n = 0; n < buffer.getNumSamples(); ++n)
        {
            float input[] = { x[n], drive }; //needs to be a pointer to a float (i.e. a single array?)

            x[n] = (neuralNet[ch].forward(input)) * volume;
        }
    }
}
//==============================================================================
bool Two_inputAudioProcessor::hasEditor() const
{
    return true; // (change this to false if you choose to not supply an editor)
}
```

```
juce::AudioProcessorEditor* Two_inputAudioProcessor::createEditor()
{
    return new Two_inputAudioProcessorEditor (*this);
}
//==============================================================================
void Two_inputAudioProcessor::getStateInformation (juce::MemoryBlock& destData)
{
    // You should use this method to store your parameters in the memory block.
    // You could do that either as raw data, or use the XML or ValueTree classes
    // as intermediaries to make it easy to save and load complex data.
}
void Two_inputAudioProcessor::setStateInformation (const void* data, int sizeInBytes)
{
    // You should use this method to restore your parameters from this memory block,
    // whose contents will have been created by the getStateInformation() call.
}
//==============================================================================
// This creates new instances of the plugin..
juce::AudioProcessor* JUCE_CALLTYPE createPluginFilter()
{
    return new Two_inputAudioProcessor();
}
juce::AudioProcessorValueTreeState::ParameterLayout Two_inputAudioProcessor::createParams(){
    std::vector<std::unique_ptr<juce::RangedAudioParameter>> params;
    params.push_back(std::make_unique<juce::AudioParameterFloat> (juce::ParameterID("DRIVE", 1), "drive",
0.0f, 1.0f, 0.5f));
    params.push_back(std::make_unique<juce::AudioParameterFloat> (juce::ParameterID("VOLUME", 2),
"volume", 0.0f, 1.5f, 1.0f));
    return {params.begin(), params.end()};
}
```

A.5 PluginEditor.h (JUCE)

```
/*

==============================================================================
==========
    This file contains the basic framework code for a JUCE plugin editor.

==============================================================================
==========
*/
#pragma once
#include <JuceHeader.h>
```

```
#include "PluginProcessor.h"
//==============================================================================
/**
*/
class Two_inputAudioProcessorEditor  : public juce::AudioProcessorEditor
{
public:
    Two_inputAudioProcessorEditor (Two_inputAudioProcessor&);
    ~Two_inputAudioProcessorEditor() override;

//==============================================================================
    void paint (juce::Graphics&) override;
    void resized() override;
private:
    // This reference is provided as a quick way for your editor to
    // access the processor object that created it.
    Two_inputAudioProcessor& audioProcessor;

    juce::Slider driveSlider;
    juce::Label driveLabel;
    std::unique_ptr<juce::AudioProcessorValueTreeState::SliderAttachment> driveSliderAttachment;

    juce::Slider volSlider;
    juce::Label volLabel;
    std::unique_ptr<juce::AudioProcessorValueTreeState::SliderAttachment> volSliderAttachment;

//    juce::DrawableRectangle outline_one;
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (Two_inputAudioProcessorEditor)
};
```

A.6 PluginEditor.cpp (JUCE)

```
/*

==============================================================================
    This file contains the basic framework code for a JUCE plugin editor.

==============================================================================
*/
#include "PluginProcessor.h"
#include "PluginEditor.h"
```

```
//==============================================================================
Two_inputAudioProcessorEditor::Two_inputAudioProcessorEditor (Two_inputAudioProcessor& p)
    : AudioProcessorEditor (&p), audioProcessor (p)
{
    // Make sure that before the constructor has finished, you've set the
    // editor's size to whatever you need it to be.
    setSize (800, 400);

    //Gain slider info
    driveSlider.setSliderStyle(juce::Slider::SliderStyle::RotaryVerticalDrag);
    driveSlider.setTextBoxStyle(juce::Slider::TextEntryBoxPosition::TextBoxBelow, true, 50, 25);
    addAndMakeVisible(driveSlider);
    driveSliderAttachment = std::make_unique<juce::AudioProcessorValueTreeState::SliderAttachment>
(audioProcessor.apvts, "DRIVE", driveSlider);

    driveLabel.setText("Drive", juce::dontSendNotification);
    driveLabel.attachToComponent(&driveSlider, false);
    driveLabel.setJustificationType(juce::Justification::centred);
    addAndMakeVisible(driveLabel);


    //Vol Slider info
    volSlider.setSliderStyle(juce::Slider::SliderStyle::RotaryVerticalDrag);
    volSlider.setTextBoxStyle(juce::Slider::TextEntryBoxPosition::TextBoxBelow, true, 50, 25);
    addAndMakeVisible(volSlider);
    volSliderAttachment = std::make_unique<juce::AudioProcessorValueTreeState::SliderAttachment>
(audioProcessor.apvts, "VOLUME", volSlider);

    volLabel.setText("Volume", juce::dontSendNotification);
    volLabel.attachToComponent(&volSlider, false);
    volLabel.setJustificationType(juce::Justification::centred);
    addAndMakeVisible(volLabel);
    //to make the plugin resizable --> kinda wonky on Reaper?
    setResizable(true, true);
    getConstrainer()->setFixedAspectRatio(getWidth()/getHeight());


    //Outlines
//    outline_one.setFill(juce::Colours::blue);
//    outline_one.setStrokeFill(juce::Colours::pink);
//    outline_one.setStrokeThickness(5.f);
//    addAndMakeVisible(outline_one);

}
Two_inputAudioProcessorEditor::~Two_inputAudioProcessorEditor()
{
}
```

```
//==============================================================================
==========
void Two_inputAudioProcessorEditor::paint (juce::Graphics& g)
{
    // (Our component is opaque, so we must completely fill the background with a solid colour)
    g.fillAll (juce::Colours::black);
}
void Two_inputAudioProcessorEditor::resized()
{
    // This is generally where you'll want to lay out the positions of any
    // subcomponents in your editor..
    auto sliderWidth = getWidth() * 0.5;
    auto sliderHeight = getHeight() * 0.8;
    auto heightOffset = getHeight()/2 * 0.2;


    driveSlider.setBounds(getWidth()/2, heightOffset, sliderWidth, sliderHeight);
    volSlider.setBounds(getWidth()/2 - sliderWidth, heightOffset, sliderWidth, sliderHeight);

//    outline_one.setBounds(getWidth()/2, getHeight()/2, 100, 200);
//    outline_one.setRectangle (outline_one.getLocalBounds().toFloat().reduced (10.5f));
}
```