Simulation of non-linear analog effects through a WaveNet Neural Architecture

Thomas B. Slap

Department of Physics, Middlebury College, Middlebury, VT

Project report for PHYS 0704

May 19, 2021

Abstract

Music provides a wealth of opportunity for signal analysis since all aspects of music can be captured by the spectral content of a sound wave. Timbre, the quality of a sound, is an especially interesting topic of study. While this human perception is rooted in art and culture, it can be fully described through Fourier analysis, specifically by the amplitude ratios in the overtone series. Guitar effect pedals apply non-linear filters in order to distort a clean sound by changing the frequency content above the fundamental pitch. Rather than directly model each circuit component of a distortion pedal, this paper explores the use of a WaveNet to simulate the distortion without needing to consider the electronic components. When trained and tested on a single note, this model achieves an error to signal ratio of 0.14 ± 0.03 with respect to the target signals. This is a 26% reduction from the inputs to the model, which had an error to signal ratio of 0.19 ± 0.03.

Date Accepted: _____

Table of Contents

I Introduction

The problem of how to distort a guitar sound has many facets: there is the psychoacoustic question of what type of distortions are pleasing to the human ear, the artistic question of the emotions invoked by a given sound, the physics and engineering question of how the circuit components are able to distort a signal in the first place, and finally the question explored in this paper of how to simulate this distortion on a computer. Before diving into the specifics of the WaveNet simulation, it will be helpful to introduce guitar distortion in more depth.

The use of deliberate guitar distortion is often credited to the Johnny Burnette Trio's *Train Kept A Rollin'* in 1956 [1]. At the time, there were no distortion pedals so musicians resorted to intentionally damaging their equipment. They tried everything from punching holes in the sound cones to partially breaking off tube amplifiers. As rock and roll continued to gain popularity over the coming decades, purpose-built electronics were designed to achieve similar sonic effects without destroying their gear. No matter the method, distortion relies on the same basic principles rooted in Fourier analysis.

Timbre has an amorphous physical definition but fundamentally refers to the spectral content of sound. McAdams and Bregman describe timbre as "the psychoacoustician's multidimensional waste-basket category for everything that cannot be labeled pitch or sound" [2]. Despite this lack of formal definition, timbre is an essential feature of music and nearly everybody has an intuitive sense for it. A flute and a trumpet playing the same steady note at the same volume are identical except for their respective differences in the overtone series, i.e., a difference in timbre. The same is true of distortion pedals, which only change the overtone content of a signal while leaving fundamental pitch and phase untouched. There are effects, such as reverb and chorus, that also change the phase but these will not be considered in this paper. Mathematically, these phase shifts can be analyzed in the complex components of a Fourier transform, rather than only the magnitudes necessary to understand distortion pedals.

Distortion pedals use a series of electrical components in order to increase the high frequency content of a signal. This process results in an overall increase in amplitude, a longer sustain, and a distorted waveform. Figure 1 shows a simulated clean guitar sound, a simple sine wave, and the output of a popular distortion pedal, a Dunlop FuzzFace. While clean guitars have overtones of their own and therefore do not output a simple sine wave, Fig. 1 clearly demonstrates the key features of distortion.

Computer simulations of these complex circuits are possible but are computationally intensive. It is therefore difficult to make accurate, real-time simulations of distortion pedals. With the recent rise in machine learning, particularly in techniques relating to sound and time-series analysis, it is now possible to make accurate simulations using a black-box model rather than one based on physical first principles [4]. A black-box model is any model that matches inputs to outputs without requiring complete information about the inner workings of the model. In contrast, every step involved in a white-box model, such as a circuit simulation, could be thoroughly explained and justified.

This paper exclusively considers the WaveNet architecture first developed by van den Oord et al. in 2016 for the admittedly harder problem of text-to-speech [5]. Turning computer text into convincing human speech is a much more complicated problem due to the difference in domain between the input and output. Still, the basic architecture of the

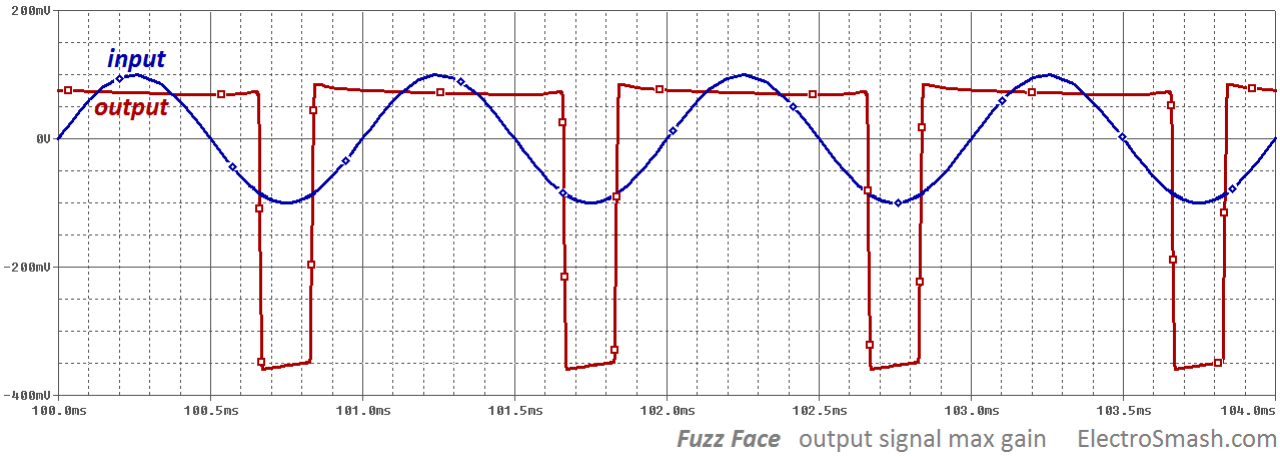

Fig. 1. The input and output of a Dunlop FuzzFace. [3]

WaveNet is incredibly useful for distortion as shown by Wright et al. in 2020. Their WaveNet produced samples indistinguishable from those created using actual distortion pedals [4]. The key operation in the WaveNet is to convolve a series of kernels with the input signal. This makes sense as a modeling strategy for simulating distortion pedals because of the similarity between clean and distorted signals. This high degree of structure between inputs and outputs allows a black-box model like a WaveNet to create a mapping between the two spaces without any first principles knowledge of the electronics that the network is simulating. This ideally results in a more efficient algorithm allowing for real-time implementation. This paper does not discuss a real-time implementation although the mathematics underpinning this network would remain largely unchanged.

II Theory

II.1 Convolution

The essential structure in these networks are the convolutional kernels in each layer. A kernel is a small matrix which is convolved with a larger matrix. Originally proposed by in 1998 by Lecun et al. as the preeminent solution for computer vision, convolutional networks have also proved useful for 1-dimensional time series data [6]. Convolutional layers work by training a set of kernels that convolve with the inputted data to form the layer's output. Mathematically, the discrete, 1-dimensional convolution operation is defined by

$$f(t) * g(t) = \sum_{\tau=-\infty}^{\infty} f(t)g(t-\tau) \tag{1}$$

where $f(t)$ is the kernel and $g(t)$ the input. It is known that $f(t) * g(t) = g(t) * f(t)$, although in the discrete case with non-infinite domain for $f(t)$ and $g(t)$, it is often easier to only consider the case where the kernel moves around the input, not the other away around. The terms kernel and filter are often used interchangeably, although the filter nomenclature is more intuitive in image rather than sound processing. Figure 2 depicts a convolution in the context of image processing, although the mechanics remain almost identical for 1-dimensional convolutions.

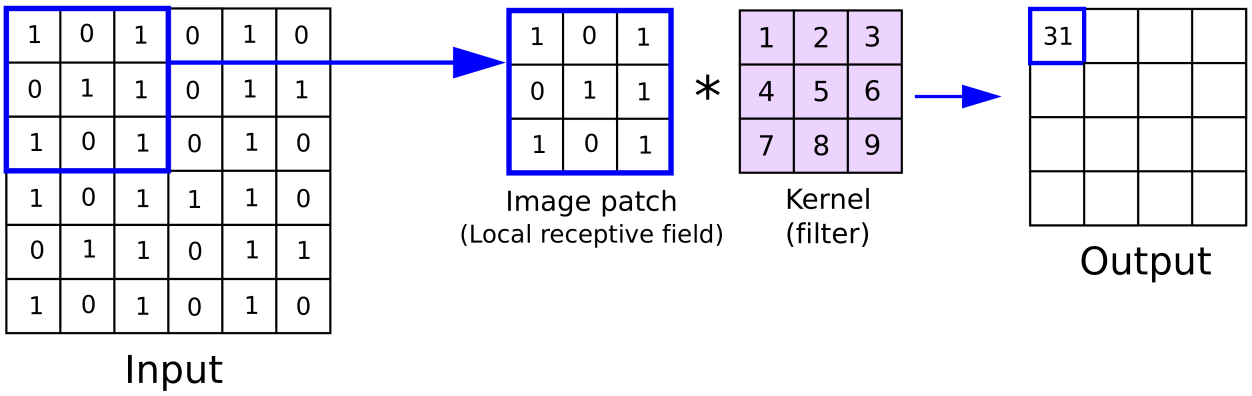

Fig. 2. Graphical depiction of how one output pixel is calculated in a convolutional neural net. [7]

Convolutional networks are ideal for distortion simulation because of the temporal relationship between input and output. Due to the high fundamental frequency shared between the input and output, only input values close in time to the predicted output need to be considered. When combined with the necessary conditions for a signal to still reasonably sound like music, this allows for the same kernel to be used for all time steps in the convolution. Therefore, only one set of parameters, the kernels, needs to be learned for each layer. This both reduces the computational complexity compared to a fully connected layer, and allows for easier simulation of features. The complexity is lower than in fully connected networks, because the same kernel can be used for each time step in the convolution. This allows for only one set of parameters to be tuned for each layer, rather than a weighted connection between each neuron in one layer to every neuron in the next layer. Kernels are ideal for feature simulation, since the kernels can be tuned in order to promote the desired features. For example, the kernels can be tuned to add the same high frequency content that results from physical distortion pedals.

II.2 Dilated Convolution

One unique and essential feature in DeepMind's WaveNet is the dilated convolution. Rather than have the kernel look at adjacent neurons in the input, in this approach, the kernel looks at neurons that are spread out in time, allowing for each neuron in a subsequent layer to be affected by a larger region of the input. Figure 3 depicts a standard, non-dilated convolution, while Fig. 4 depicts a dilated convolution where the dilation factor increases exponentially. This exponential increase in the dilation factor allows for the receptive field to grow exponentially rather than linearly with the number of layers. This is particularly useful in sound applications, because the sample rate of digital audio recordings, usually 44,100 Hz, requires a lot of information to be processed in order to analyze a very short clip in time.

These dilated kernels can be thought of as kernels of larger size but with holes between the values. This can be mathematically achieved by inserting ((dilation factor) - 1) zeros between the non-zero values and performing a traditional convolution. While this achieves the desired result, it brings none of the computational efficiency of a truly dilated convo-

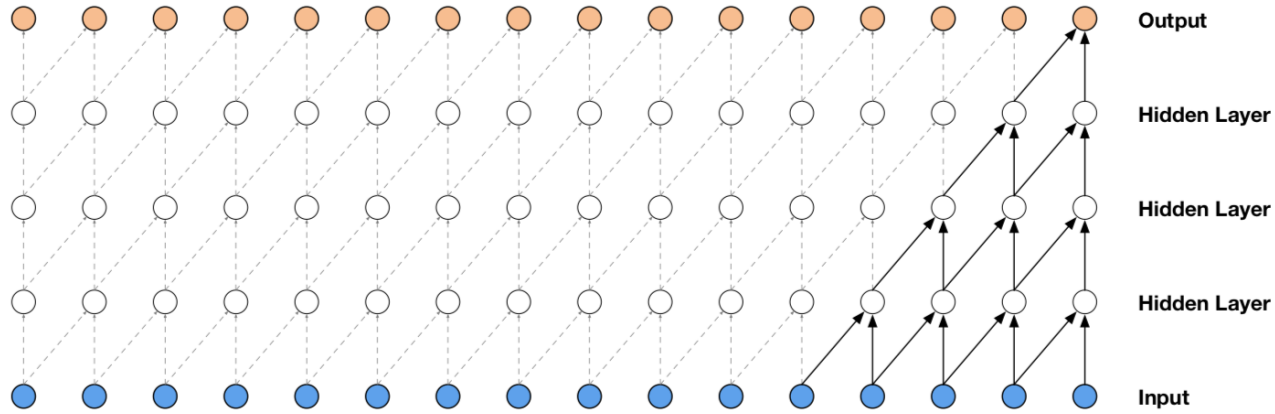

Fig. 3. Traditional convolution with a filter of length 2. Notice how the receptive field of the output neuron only increases by 1 with each additional layer. [5]

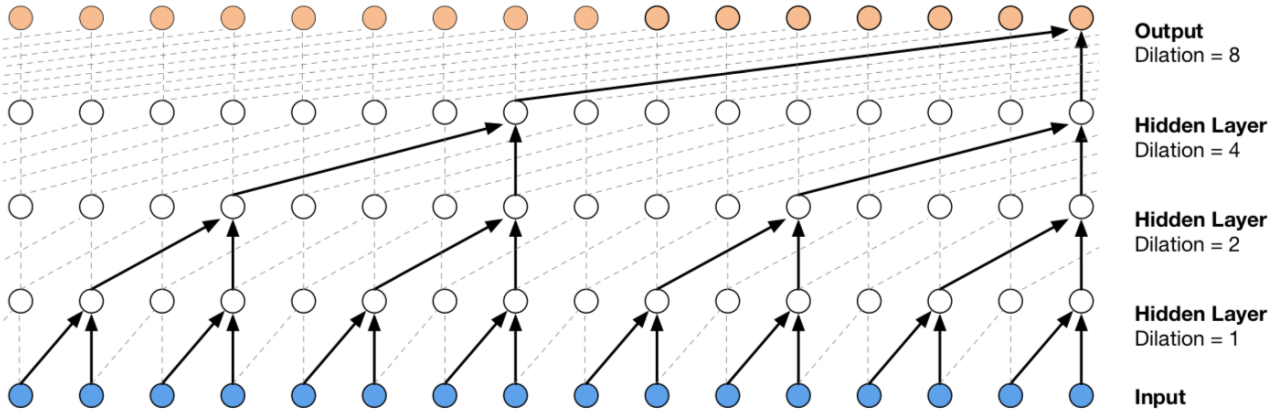

Fig. 4. Dilated convolution, still with a filter of length 2. The dilation factor goes as 2^l for the lth layer resulting in an exponentially increasing receptive field. [5]

lution, since the computer does not know that a product with 0 is trivial. Additionally, in machine learning, unless careful steps are taken to avoid this, the zero values will become non-zero after the first round of training. It is therefore preferable to keep the kernel size the same as in a traditional convolution, but redefine the convolution operation to hit only certain values of the input signal. The discrete, dilated convolution is defined as

$$f(t) * g(t) = \sum_{\tau=-\infty}^{\infty} f(t)g(t - l\tau) \tag{2}$$

where $l \in \mathbb{N}$ is the dilation factor of the convolution.

II.3 Forward Pass

Before addressing the specific mathematics implemented in this network, it is useful to establish a convention regarding the dimensionality of the various objects involved in the network. Lower case letters (e.g., $\hat{y}$) correspond to vectors, capital letters (e.g., Z_0) correspond to matrices, and bolded capital letters (e.g., $\mathbf{Z}$) correspond to rank-3 tensors. It is important to note that nearly all of these higher rank objects use 1-dimensional vectors as their building blocks and are merely convenient ways to store multiple pieces of information. For example, K_1, the kernel block for layer 1, will contain the individual kernels $k_{1,0}$ through $k_{1,i}$ for a network with i kernels per layer.

The network consists of a series of convolutional layers, each applying a set of kernels K_l to their respective input signals x_l. The outputs of each kernel's convolution are mixed together with a mixing kernel w_l before passing the output onto the next layer. The unmixed outputs are also passed to a final linear mixer. This final mixer, $W_{\mathbf{Z}}$, applies a 1×1 convolution, essentially a weighted sum, to a tensor $\mathbf{Z}$ made up of all the matrix outputs of the layers stacked together to form a rank-3 tensor. The 1×1 convolution brings the output dimension of the network back down to a vector, which is the appropriate dimension for a sound wave. The matrix outputs of each layer are constructed similarly by stacking the vector outputs of each convolution into a 2-dimensional matrix before mixing them back together through a separate 1×1 convolution. The interaction between each layer can be seen in Fig. 5.

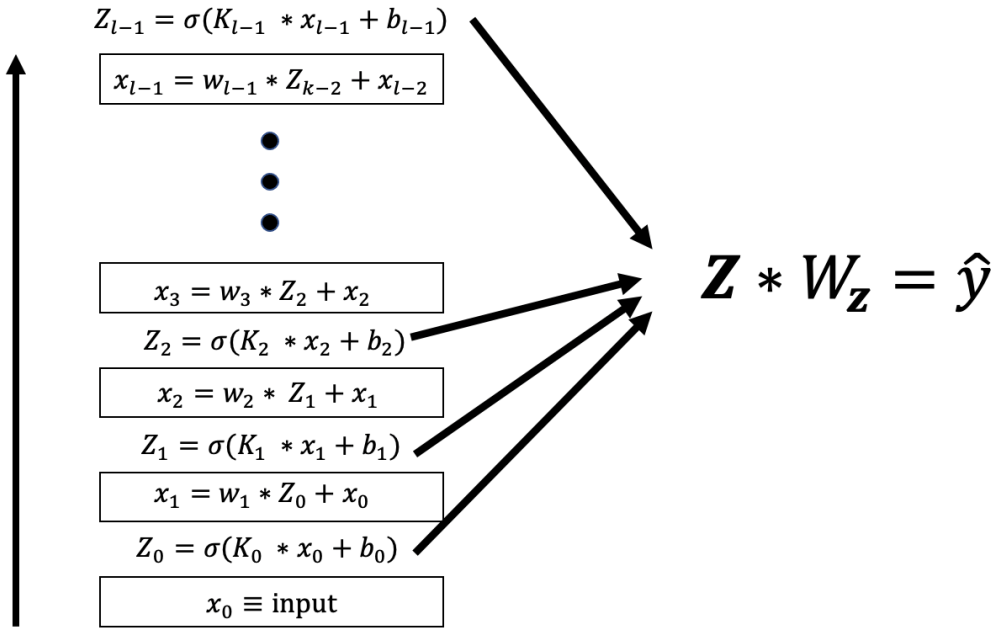

Fig. 5. A network consisting of l layers. Input x_0 is passed into the first layer, where kernels K_0 are applied, a bias is then added, and a sigmoid function is applied to form the output Z_0. This process is repeated for all l layers. Finally, all the outputs, Z_l, are stacked into the tensor $\mathbf{Z}$, which is convolved with $W_{\mathbf{Z}}$ to form the final output $\hat{y}$.

Each layer, identified by an iterator variable l, takes in an input x_l and applies a series of 1-dimensional kernels k_i stored in a matrix K_l which is the kernel block for the l^{th} layer. Both the number and length of the kernels is a

hyperparameter of the network, meaning that it is not a learned trait and is set during initialization. Each individual kernel k_i is convolved with the input. These convolutions are then stacked together to form the matrix Z_l, the output of the layer. Eq.(3) describes a preliminary output for the i^{th} vector in the matrix output Z_l.

$$z_{l,i} = k_i * x_l \tag{3}$$

This form of $z_{l,i}$ captures the essence of each convolutional layer but is not an optimal solution. For one, the magnitude of the convolutions can vary drastically. This inhibits the gradient descent algorithm, since the gradients are sensitive to the magnitude of Z_l, and will therefore also vary drastically, preventing the network from reaching an optimal solution. To keep the outputs of each layer within a known range, a sigmoid function, defined in Eq.(4), is used to compress all the values into the range (0,1).

$$\sigma(x) \equiv \frac{1}{1+e^{-x}} \tag{4}$$

Because it relies exclusively on convolutions, these layers would not be able to map inputs to orthogonal outputs. To allow the output to have components orthogonal to the input, a bias vector b_l must be added to the convolution. This allows for zero values in the input to be mapped to non-zero outputs. Otherwise, the network would not be able to perform even the simple transformation from $sin(x)$ to $cos(x)$. This bias term introduces high frequency noise into the signal which is not found in analog distortion. This noise can be removed in post processing, but for real-time simulation a different method would likely be needed. These amendments result in the $z_{l,i}$ definition shown in Fig. 5 and ultimately used in the network. This definition is

$$z_{l,i} = \sigma\left(k_i * x_l + b_l\right). \tag{5}$$

Before passing the output onto the next layer of the network, the input to the layer x_l is added to a 1×1 convolution of Z_l with a kernel w_{l+1}. A 1×1 convolution is essentially a weighted sum, since it applies the same kernel weights to each time step in order to turn the matrix Z_l back into a 1-dimensional vector. This vector can then be summed with the layer's input to form the layer's output. The 1×1 is nomenclature taken from computer vision and just means that the kernel size matches the smallest unit in time. This does not mean that the kernel is actually a 1×1 matrix as the dimensionality of the kernel depends on the shape of the matrix Z_l. Let w_{l+1} be the kernel for this 1×1 convolution such that the output passed to the next layer in the network is

$$x_{l+1} = (w_{l+1} * Z_l) + x_l. \tag{6}$$

Finally, all of the layer matrix outputs Z_l are stacked into a rank-3 tensor $\mathbf{Z}$. To get the final output $\hat{y}$, as defined in Eq. (7), $\mathbf{Z}$ goes through a 1×1 convolution, this time with matrix W_Z, in order to collapse the rank-3 tensor into a 1 dimensional output with the same shape as the input.

$$\hat{y} = \mathbf{Z} * W_Z \tag{7}$$

The forward pass for a simplified network is represented visually in Fig. 6. Grasping the dimensionality of the network at each stage of the forward pass is crucial for gaining intuition for how the network learns, as well as for implementing the mathematics into computer software.

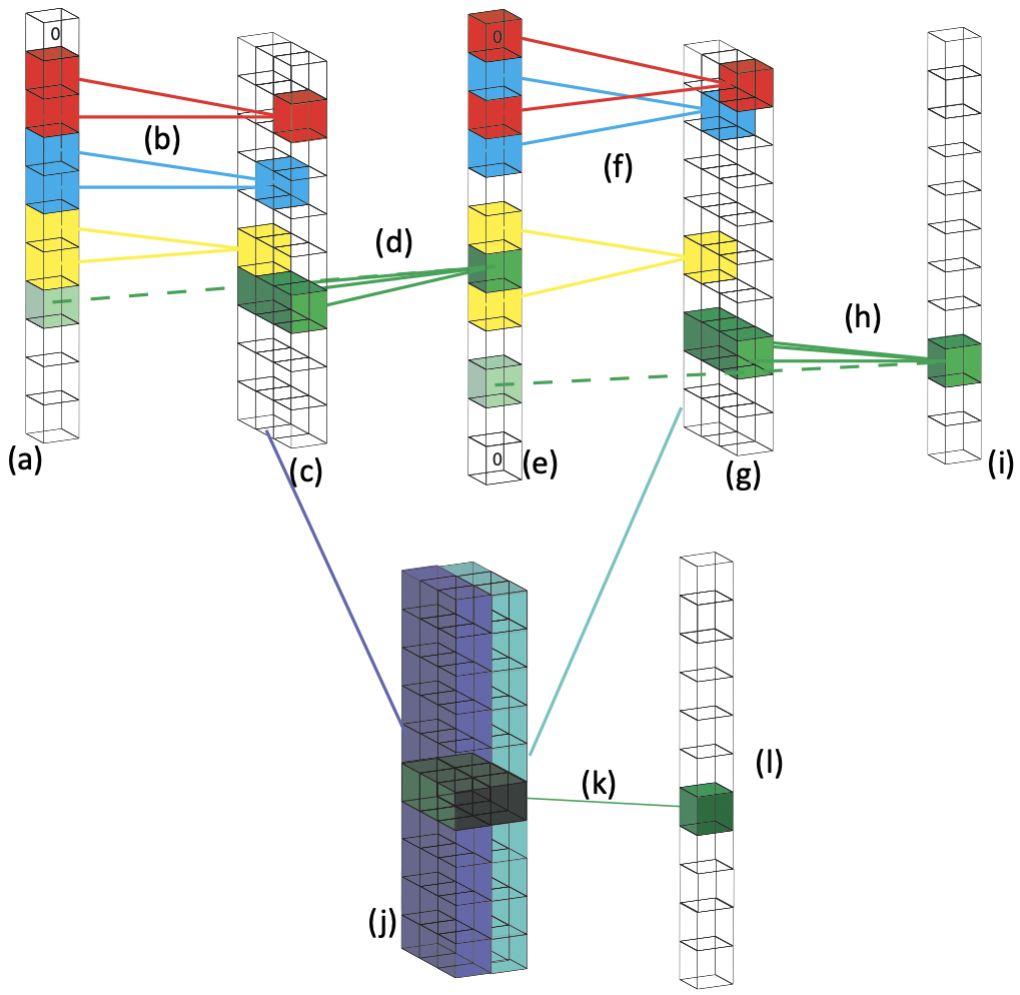

Fig. 6. A visual representation of a forward pass for a simplified network containing only two layers, 3 kernels per layer, and a 2^l dilation scheme. Each cube represents a single neuron. (a) the vector input x_0. (b) The kernel block K_0, consisting of three distinct kernels: red, blue, and yellow, each with dilation factor zero. Only one time step of the convolution is shown for each kernel. (c) The matrix, Z_0, consisting of vectors corresponding to each kernel in K_0. (d) The 1×1 convolution that uses the kernel w_1 to create a weighted average at each time-step of Z_0 and x_0 in order to produce x_1. (e) x_1. (f) The kernel block K_1 still with 3 kernels but now with dilation factor one. (g) Z_1. (h) w_2. (i) x_2. (j) The rank-3 tensor $\mathbf{Z}$ consisting of the stacked layer outputs Z_0 and Z_1. (k) The kernel $W_{\mathbf{Z}}$ for the final 1×1 convolution. (l) The output $\hat{y}$.

II.4 Backpropogation and Gradient Descent

The forward pass described above in section II.1, particularly the dilated convolutions, is crucial to the performance of the network. However, the network still needs a method to learn the optimal values of the weight matrices, kernels, and biases. This learning is done by calculating the gradient in the loss-landscape for each parameter and then stepping downhill in order to approach a more optimal solution. This basic concept is shared amongst all machine learning algorithms.

The loss, L, a scalar defined by the cost function in Eq. (8), is the error to signal ratio (ESR) of the two signals.L is also very similar to the mean squared error between $\vec{y}$ and $\hat{y}$ This similarity makes intuitive sense since the output $\hat{y}$ would have low bias, the offset between $\hat{y}$ and the target signal $\vec{y}$, and low variance since consistent performance is important for practical applications.

$$L = \frac{1}{|y|^2} \sum_{t=0}^{N} (y_t - \hat{y}_t)^2 \tag{8}$$

This is a useful loss function because its gradient is easy to compute and it is independent of the direction of the difference between $\hat{y}$ and $\vec{y}$ while still capturing this directional information in the derivative. This gradient, $\frac{\partial L}{\partial \hat{y}}$, is crucial since the updates to every parameter use the derivative of the ESR with respect to $\hat{y}$ to determine the magnitude and direction of each adjustment to the learned parameters,

$$\left(\frac{\partial L}{\partial \hat{y}}\right)_t = \frac{2(y_t - \hat{y}_t)}{|y|^2}. \tag{9}$$

With $\left(\frac{\partial L}{\partial \hat{y}}\right)_t$ in hand, we can now begin to see how each component of the network affects the loss by using the chain rule. While this process could be correctly identified as backpropogation, in this network very few gradients are actually passed backwards through the layers. This was a design decision because while it is true that the kernel in layer 1 will affect the output of layer 1 and therefore all subsequent layer outputs, rather than analytically calculating solutions for the perfect step for the kernels to take based on every subsequent layer's output, the subsequent layer's kernels can instead learn what types of values to expect. This results in a faster but noisier learning process.

II.4.1 Adjusting the Linear Mixer W_Z

The effect that the linear mixer W_Z has on the loss is determined by the effect that W_Z has on $\hat{y}$ and the effect $\hat{y}$ has on L,

$$\frac{\partial L}{\partial W_Z} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W_Z}. \tag{10}$$

Because $\frac{\partial L}{\partial \hat{y}}$ is already known, we just need to calculate $\frac{\partial \hat{y}}{\partial W_Z}$. Since $\hat{y} = W_Z * Z$ and the discrete convolution is defined such that $\hat{y}_t = W_Z \odot Z_t$, where $\odot$ is the Hadamard product operation, the total effect that W_Z has on the loss will be a sum of the product between the residual between $\vec{y}$ and $\hat{y}$ at each time step and the matrix Z_t, which contains all the outputs from each filter for time t. The Hadamard product is the elementwise multiplication of two matrices of the same dimension. The sum of a Hadamard product can be thought of as a dot product between two matrices. $\frac{\partial L}{\partial W_Z}$ is therefore given by

$$\frac{\partial L}{\partial W_Z} = \sum_{t=0}^{N} \frac{2(\vec{y}_t - \hat{y}_t)}{|y|^2} Z_t. \tag{11}$$

II.4.2 Adjusting the Kernels K_l and Bias b_l

The effect that a specific layer's kernel has on the loss is determined by how the kernel affects the output of the layer and how that layer subsequently affects the loss. Each individual filter within the kernel block must be treated separately. The gradient of the loss with respect to the i^{th} kernel of the l^{th} layer's kernel block is given by

$$\frac{\partial L}{\partial k_{l,i}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_{l,i}} \frac{\partial z_{l,i}}{\partial \sigma(...)} \frac{\partial \sigma(...)}{\partial k_{l,i}} \tag{12}$$

Here, $\sigma(...)$ represents the argument $\sigma(k_{l,i} * x_l + b_l)$ so that $\frac{\partial z_{l,i}}{\partial \sigma(...)}$ is how the output $z_{l,i}$ is dependent on the output of the sigmoid function and $\frac{\partial \sigma(...)}{\partial k_{l,i}}$ is how the output of the sigmoid function is dependent on the individual kernels $k_{l,i}$ contained in the layer's kernel block K_l.

We already know $\frac{\partial L}{\partial \hat{y}}$, leaving $\frac{\partial \hat{y}}{\partial z_{l,i}}$, $\frac{\partial z_{l,i}}{\partial \sigma(...)}$, and $\frac{\partial \sigma(...)}{\partial k_{l,i}}$ to be calculated. For each time step in the signal, the dependence of the output signal $\hat{y}$ on the stacked outputs of the layers $\mathbf{Z}$ is dependent on the mixing kernel W_Z, and is therefore given by

$$\left(\frac{\partial \hat{y}}{\partial z_{l,i}}\right)_t = W_{Z_{l,i}}. \tag{13}$$

$\frac{\partial z_{l,i}}{\partial k_{l,i}}$ is dependent on how the sigmoid function defined in Eq. (4) intermediates the effects of the filter and bias on the output, as well as on how the convolution $k_{l,i} * x_l$ is dependent on the filter $k_{l,i}$. The derivative of the sigmoid function can be shown to be $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. Therefore, the gradient for a specific kernel within a specific layer at a given time t is

$$(\frac{\partial z_{l,i}}{\partial k_{l,i}})_t = \frac{\partial z_{l,i}}{\partial \sigma(...)} \frac{\partial \sigma(...)}{\partial k_{l,i}} = \sigma(k_{l,i} * x_l + b_l)_t (1 - \sigma(k_{l,i} * x_l + b_l)_t) x_{l_t}. \tag{14}$$

It is important to note that each derivative involved in $\frac{\partial L}{\partial k_{l,i}}$ is only for a single time step of the signal. Because the same filter is applied to each time step, in order to learn the optimal kernels, the adjustment to the kernel must be an average of the gradient at each time step. The full adjustment to the kernel is therefore given by

$$\frac{\partial L}{\partial k_{l,i}} = \frac{1}{N} \sum_{t=0}^{N} \frac{2(y_t - \hat{y}_t)}{|y|^2} W_{Z_{l,i}} \sigma(k_{l,i} * x_l + b_l)_t (1 - \sigma(k_{l,i} * x_l + b_l)_t) x_{l_t} \tag{15}$$

The adjustments to each layer's bias b_l is very similar with the only difference coming from the replacement of $\frac{\partial z_{l,i}}{\partial k_{l,i}}$ with $\frac{\partial z_{l,i}}{\partial b_l}$. Because the derivative of the linear addition of a vector is 1, the adjustment to each layer's bias is given by

$$\frac{\partial L}{\partial b_l} = \frac{1}{N} \sum_{t=0}^{N} \frac{2(y_t - \hat{y}_t)}{|y|^2} W_{Z_{l,i}} \sigma(k_{l,i} * x_l + b_l)_t (1 - \sigma(k_{l,i} * x_l + b_l)_t). \tag{16}$$

II.4.3 Adjusting the Mixing Weights w_l

The mixing weights w_l control the strength of the residual connection between layers. This learned parameter allows the network to find the optimal weighted sum between the input and output of a given layer to pass on to the next layer. While it is technically true that the mixing weight w_1 affects every subsequent layer in the network, these effects quickly diminish and can be accounted for by the other learned parameters in the network. To reduce both the mathematical and computational complexity, only the effect on the immediately subsequent layer will be considered.

In order to find the effect that a given mixing weight w_l has on the loss, the effect of w_l on the subsequent layer and that layer's effect on the loss must be determined. This relationship is

$$\frac{\partial L}{\partial w_l} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial Z_l} \frac{\partial Z_l}{\partial \sigma(...)} \frac{\partial \sigma(...)}{\partial x_l} \frac{\partial x_l}{\partial w_l}. \tag{17}$$

Here, $\sigma(...)$ represents the argument $\sigma(K_l * x_l + b_l)$. Many of these derivatives have already been solved for above, leaving only $\frac{\partial \sigma(...)}{\partial x_k}$ and $\frac{\partial x_l}{\partial w_l}$ left to consider. The effect that x_k has on the output of the sigmoid function is only dependent on the kernel block K_l, since the convolution operator at any time step is just a dot product between the input x_k and an individual kernel $k_{l,i}$. Because the effect of the mixing weights on the loss varies for each time step, the adjustments to w_l must again be averaged over all time steps before updating the weight matrix. The gradient of the sigmoid term with respect to x_l at all times t is then

$$\left(\frac{\partial \sigma(...)}{\partial x_l} \right)_t = K_l. \tag{18}$$

The same time-averaging technique must also be used for $\frac{\partial x_l}{\partial w_l}$, since the weight matrix does not change in time while the gradient with respect to the loss does. Again, at any time step, the output $x_{l,t}$ is a dot product of the weights w_l and the previous output $z_{l-1,t}$ implying that $\frac{\partial x_l}{\partial w_{l,i}} = Z_{l-1,t}$ Putting all of these derivatives together results in a final gradient of the loss with respect to the i^{th} value in w_l,

$$\frac{\partial L}{\partial w_{l,i}} = \frac{1}{N} \sum_{t=0}^{N} \frac{2(y_t - \hat{y}_t)}{|y|^2} W_{Z_{l,i}} \sigma(k_{l,i} * x_l + b_l)_t (1 - \sigma(k_{l,i} * x_l + b_l)_t) k_{l,i} z_{k-1,t}. \tag{19}$$

However, despite confidence in the mathematics underpinning this gradient, this part of the learning process was not able to be implemented in the network tested below due to author's lack of computer science expertise and the added complexities of passing information between layers.

Now that the gradient with respect to the loss function for every learned parameter of the network is known, the training process can commence. Training happens by feeding the network an input/output pair from the data set. The network makes a forward pass with the input, calculates the loss of the network's output compared with the target, and calculates the gradient for each parameter. Then, each parameter is updated by subtracting the gradient multiplied by the learning rate. The learning rate is a hyperparameter, an unlearned parameter determined during initialization, which controls how large each adjustment is. If the rate is too high, the network often bounces around the loss landscape sporadically until an overflow error occurs. If it is too low, then it will be less efficient and require more data to reach the same level of optimization. This also leads to a higher probability of settling into a local minimum, since a small step size reduces the likelihood that the network will escape a given minimum.

III Methodology

III.1 Data Pre-Processing

The data set used to train this model was collected by the Fraunhofer Institute for Digital Media Technology [8]. It contains 625 two-second clips of a single note being played on a non-distorted guitar and a corresponding note that has been passed through an unidentified distortion pedal. Since the clips use a high sample rate of 44100 Hz, these signals were split into

sub-clips in order to reduce computation time for each pass through the network. There is also a large period of silence before each note starts that was removed from each clip in order to avoid passing the network a signal containing no useful information. After breaking the signal into clips with a length of 1000 samples, the amplitudes needed to be aligned in order to reduce unimportant differences between the input and output. While real distortion pedals do often increase the amplitude of a signal, this can easily be accomplished in post-processing, and is therefore ignored during the training stage. The amplitude was standardized for each pair of input and output signals by dividing the amplitude at each time step by the root mean square (RMS) of the entire clip. The RMS of a signal is defined in Eq. (20). After this RMS standardization, the amplitudes of the clips were still not aligned, so the inputs were multiplied by an arbitrarily chosen scalar to bring the amplitudes even closer together. Figure 7 shows a randomly chosen clip before and after this data cleaning process.

$$RMS = \sqrt{\frac{1}{n}\sum_{n=0}^{n-1} x_n^2} \tag{20}$$

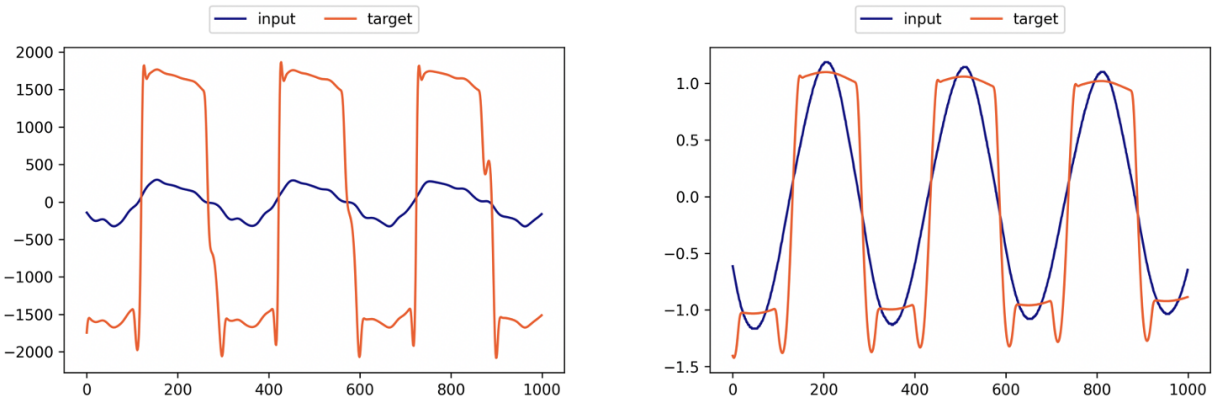

Fig. 7. Unprocessed input/output pair (left) and the same input/output pair after going through RMS standardization (right).

III.2 Batched Gradient Descent

In order to avoid the network overfitting to the most recent training examples, mini-batch gradient descent was implemented. Mini-batch gradient decent passes a small subset of the training data through the network before updating the parameters with an averaged gradient. Compared with stochastic gradient descent (SGD), which updates the parameters after each training signal, mini-batch gradient descent more smoothly approaches a generally optimal solution [9]. While the randomness introduced by SGD can help prevent the network from getting stuck in a local minimum, mini-batch gradient descent is more likely to lower the loss more quickly. This is an especially important consideration due to the relatively small amount of data available for distorting a single note. Mini-batch gradient descent also lowers the probability that the last training sample causes the network's loss to increase dramatically. This is again especially important due to the increased variance in the network's loss due to the small data set.

III.3 Hyperparameter selection

A hyperparameter is any adjustable feature of the network that is fixed rather than learned. These are set when initiating the network and remain unchanged throughout the network's operation. The one exception is that changing the learning rate, which multiplies the adjustment of each learned feature, is often a component of more advanced learning strategies. These strategies are not considered here. Table I contains the hyperparameters chosen for the networks analyzed in section IV.

Table I. A list of the adjustable hyperparameters of the network and their respective chosen values.

Hyperparameter	Proof of Concept	Single Note
Number of Layers	9	8
Dilation Scheme	$1, 2, 4, 8, ..., 512$	$1, 2, 4, 8, ..., 256$
Number of kernels per layer	6	10
Kernel Length	3	3
Batch Size	1	5
Learning Rate	0.1	0.05

IV Results

IV.1 Proof of concept

In order to demonstrate that this network can learn to match a clean input to a distorted output, it was fed the same input/output pair repeatedly. This is an extremely contrived situation, since the whole purpose of machine learning is to find generalizable patterns in the data. For a single pair of signals, an analytic solution can be found, making machine learning a needlessly complicated strategy. Still, this is a good proof of concept since any egregious errors in the learning method would make an appearance in this test of the network. Figure 8 demonstrates that the learning algorithms described in section II can approach optimal solutions in this case.

IV.2 Single Note Distortion

For a more robust test, the network was trained to distort a single note. This still falls short of gauging how well the network would perform as a distortion pedal, since an effective network would need to generalize across the many frequencies a guitar is capable of producing. However, due to time constraints, it was infeasible to train the network on the entire data set.

For this test, a note was arbitrarily chosen, thereby narrowing the data set to one clean input sound and one distorted output sound. The data cleaning described in section III.1 was applied, resulting in 65 clips with 1000 samples in each clip. 50 of these clips were randomly chosen as training data, leaving the other 15 as testing data. The network then went

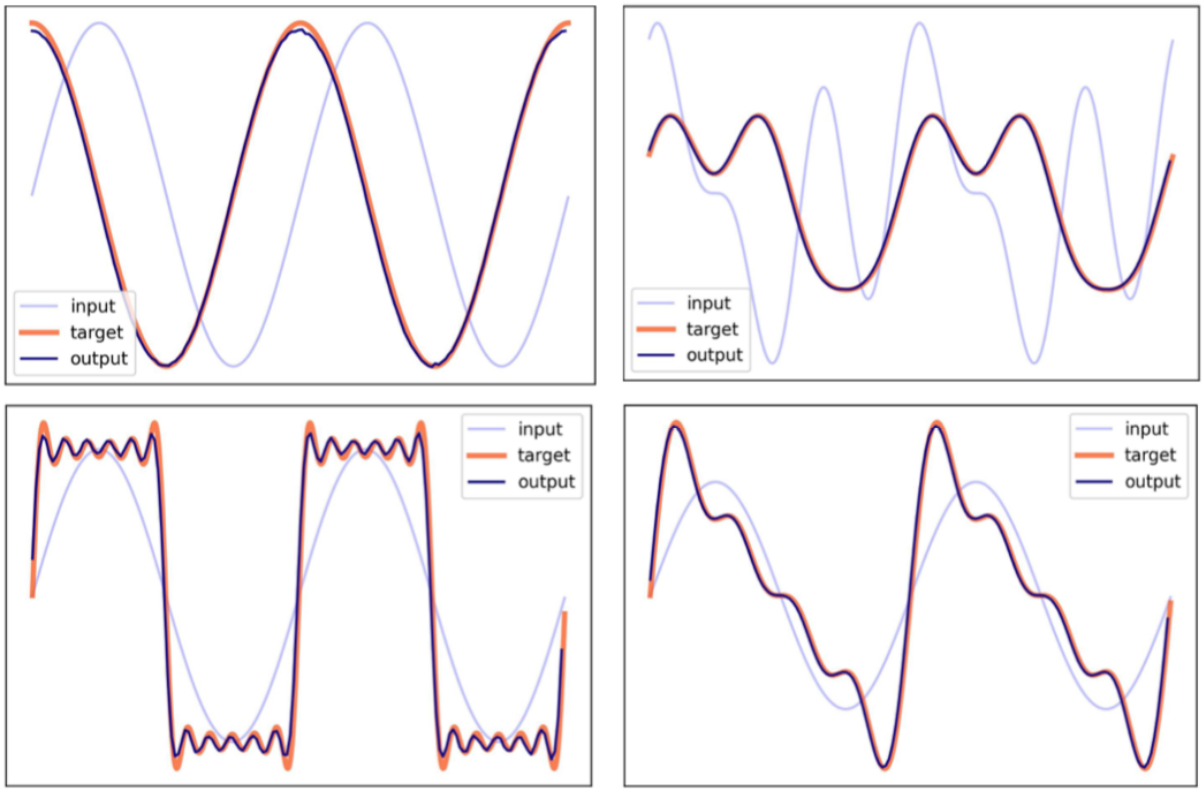

Fig. 8. The input, output, and target signal are shown for 4 different examples of input output pairs. Upper left: network's ability to transform an input to an orthogonal output. Upper right: input and output are both randomly chosen sums of sine and cosine waves. Bottom left: a realistic guitar distortion effect of turning a sine wave into a square wave. Bottom right: a similarly realistic situation of turning a sine wave into a sawtooth wave.

through a mini-batch gradient descent process on the training data 5 times before the network was tested. The test data was then passed through the network and the outputs were compared with the target signals.

The output had a consistent offset from both the input and target signals. To account for this, the mean of the output was brought in line with mean of the input. This constant offset does not affect the sound of the signal. As alluded to earlier in section II.3, the network outputs a very noisy signal due to the bias term in each layer. To eliminate this noise, high frequencies were filtered out by taking the Fourier transform of each output clip and zeroing all the frequencies except for the 100 frequencies closest to 0 Hz. While this transformation has a clear impact on the sound, because it does not use any information about the target signal it could be used in practical applications of the distortion pedal. The the only drawback is the added computation time to take both a Fourier transform and an inverse Fourier transform. A randomly chosen output from the test is shown in Fig. 9.

Due to the systematic shift in the unprocessed output, the ESR for the unadulterated output is significantly higher than that of the input. After the post-processing described above, the ESR fell to become statistically significantly better than that of the input. Table II shows the results of both the unprocessed and post-processed output as well as the input.

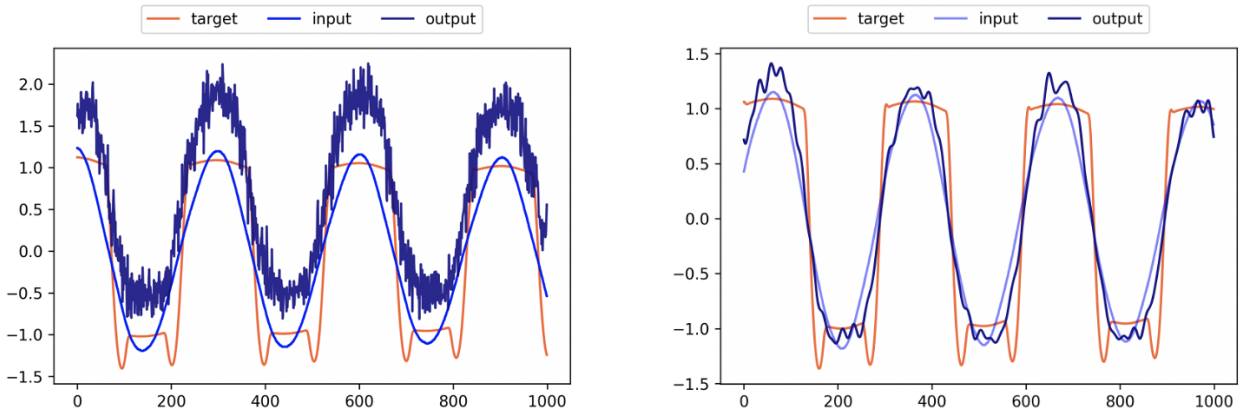

Fig. 9. The unprocessed output of the network (left) and the output after adjusting for the difference in means and high frequency noise (right).

Table II. Summary statistics comparing the input x_0, the output $\hat{y}$, and a processed $\hat{y}$ to the target output $\vec{y}$.

I/O	Average Loss	σ
Input	0.19	0.03
Output	0.51	0.03
Processed Output	0.14	0.03

IV.2.1 Statistical Analysis

Already these results are significant due to the larger than 1σ difference in performance between the network's output, and the case where the network does nothing, i.e the networks input. However, by rephrasing the question to ask how likely the network is to make the input signal closer to a target signal, we need slightly more sophisticated statistics. Taking a permutation test of the results yields a p-value of 8.38×10^{-16}. There therefore is a nearly infinitesimally small probability that these results arose by chance and it safe to conclude that the network does in fact output a signal closer to the target than the input. It makes sense to consider only whether or not the network improves a signal's ESR rather than the magnitude of that reduction because it is unlikely that the ESR of a signal accurately captures how the human brain processes sound, making the magnitude of the difference in MSE an arbitrary metric. Conducting a p-test is still arbitrary, but it asks a more accurate, even if less precise question about the relationship between the input and output.

This p-value was calculated by using the central limit theorem to assume that the differences in mean squared error between the input and output signals of the network would be normally distributed with $\mu = 0$ and $\sigma = \frac{\sigma_{in}}{\sqrt{15}}$. Then the probability that an observed difference is due to chance alone is given by the cumulative distribution function of this normal distribution. Here, the average difference is 0.06, which corresponds to a p-value of $8.38 * 10^{-16}$.

V Next Steps

While exciting progress was made towards a real-time distortion pedal simulator, there are many steps necessary to reach this ultimate goal. First, in order to use the full data set and possibly create larger networks, the implementation must be changed to utilize GPUs. This could be done by replacing the NumPy package with CuPy, a similar package with built-in GPU optimization, or by switching to a library containing pre-built machine learning features such as Google's TensorFlow. Taking advantage of the additional horsepower and corresponding drop in training time, a more rigorous analysis of hyperparameters would likely prove fruitful. The hyperparameters chosen for the two iterations of the model described above were arrived at by arbitrarily selecting values and qualitatively gauging how well the network performed. A more rigorous process would involve testing many different hyperparameter combinations, comparing the loss as a function of training samples and choosing an ideal configuration that results in $\hat{y}$ having the lowest ESR.

The learning regimen could also be improved as only a vanilla mini-batch gradient descent was implemented to train the network. More advanced techniques that improve efficiency and reduce the likelihood of being stuck in a local minimum could be implemented. Many of these techniques originate from physics concepts. One example is to give each parameter a momentum term, so that if training samples repeatedly push in one direction, the parameter is inclined to continue in that direction even if the gradient temporarily changes. This ideally would allow for parameters to crest over small hills in the landscape surrounding a local minima and allow the value to descend to a more optimal solution.

To allow for real-time simulation, extensive changes would need to be made. As mentioned above, the bias term in each layer creates noise in the signal that needs to be filtered out in post-processing. For real-time simulation, it would likely be better to change the bias term to be less random and more self-consistent. One idea would be to define an additional term in the cost function that measures how noisy the bias vector is. Additionally, the network would likely need to be written in a lower level language such as C++ in order to remove the speed limitations of Python. Finally, a streamlined pipeline of pre- and post-processing is also necessary for real-time simulation. The pre-processing would be less complex than that for the training data since there is no known output to align phase and amplitude with. Still, there would need to be an efficient process to record audio, window the audio into small clips, pass each clip through the network, and splice the clips back together into a coherent output signal. To avoid high frequency noise, a Hann window or similar windowing function could be used to improve the sound quality upon restitching.

References

[1] "Johnny burnette and the rock 'n roll trio," *Wikipedia*, Feb 2021.

[2] S. McAdams and A. Bregman, "Hearing musical streams," *Computer Music Journal*, vol. 3, no. 4, pp. 26–60, 1979.

[3] "Fuzz face analysis," *ElectroSmash*.

[4] A. Wright, E.-P. Damskägg, L. Juvela, and V. Välimäki, "Real-time guitar amplifier emulation with deep learning," *Applied Sciences*, vol. 10, no. 3, 2020.

[5] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. W. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *CoRR*, vol. abs/1609.03499, 2016.

[6] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, pp. 2278–2324, 1998.

[7] A. H. Reynolds, "Convolutional neural networks (cnns)," *Anh H. Reynolds*, Oct 2017.

[8] C. Christian Kehling, A. Männchen, and A. Eppler, *IDMT-SMT-GUITAR*. Fraunhofer Institute for Digital Music Technology.

[9] S. Ruder, "An overview of gradient descent optimization algorithms," *Sebastian Ruder*, Mar 2020.