Classifying Chaotic Synthesizers with a Multimodal Neural Network

Thesis Subtitle

By:
Landon Buell[1]

Advised By:
Dr. Kevin Short[2] **, Dr. Maurik Holtrop**[1]

A thesis presented for the degree of
Bachelor of Science in Physics

December 2020

[1]Department of Physics and Astronomy
[2]Department of Mathematics and Statistics

University of New Hampshire
Durham New Hampshire, USA

Contents

1 Introduction

2 Methodology

This multi-faceted project includes a great number of steps to ensure it's completion. Here we show a high-level outline of the pieces of this project.

2.1 Designing the Function

Consider the biological process of hearing a sound wave and matching it to a source. We can model this behavior by some unknown function F and approximate it with the function F^*. This can map the contents of a sound file to a potential source as to mimic F. For a set of inputs (called features or predictors) $\vec{x} = \left\{x_0, x_1, x_2, ..., x_{p-1}\right\}$ and a set of classes 0 through $k-1$, we denote this function as:

$$F^* : \vec{x} \rightarrow \left\{0, 1, 2, ..., k-1\right\} \tag{1}$$

In secs. (3.1) and (3.2), we introduce the nature and structure of a neural network and how it can use used in part to solve this problem.

2.2 Collecting and Pre-processing Raw Data

In order to train the Multimodal neural network to identify musical instrument sources, we need a suitable data set to present to the model as training data. University of Iowa Electronic Music Studio, and the London-Based Philharmonia Orchestra each have a large collection of publicly available audio files Citations!. These contain short segments of musical instruments performing a single note or a collection of notes in succession. Each sound file is labeled according to the instrument that is producing that sound.

To ensure that these data sets are formatted consistently, we read each sample from it's original format, *.AIF*, *.MP3*, or similar, and rewrite the data as a new *.WAV* files, sampled at 44.1 kHz with a 16 bit depth. This ensures that all data will have a consistent format when features are extracted. This stage is detailed in sec.(5.0.2).

2.3 Designing Classification Features

The performance of a neural network is largely dependent of designing an appropriate set of classification features. These are properties of wave forms that can be represented by a numerical value or several numerical values and are used as the primary tool in classification. We detail these features in sec. (5). These are used in place of a full waveform to represent a file's contents in a low-dimensional array. We use tools from physics, mathematics, and signal processing to define and explore a comprehensive set of features that enables a high performance of the classifier.

2.4 Designing A Complementary Network Architecture

With the appropriate set of features designed, and the number of output determined, we can organize the structure of the neural network function. We construct a multimodal neural network, explored in sec. (3.6), that processes two input arrays derived from the same audio sample. This network is designed to handle and process each respective input independently, and concatenate the results to produce a single output.

2.5 Testing and Evaluating Network Performance

We divide the raw data set up into a training and testing sets. We employ cross-validation and compute the results of performance metrics to ensure that out model is making reliable predictions, and generalize appropriately. In this stage, we also choose the value of hyper-parameters, activation functions and layer widths to best compliment the chosen features. This process is repeated and expanded upon until we have produced a model with a sufficient performance.

2.6 Running Predictions of Chaotic Synthesizer Files

Once we have established a suitable performance of the model, we allow the model to run predictions on the un-labeled chaotic synthesizer wave forms. We output the prediction results to a file, and compare the neural network predictions against human predictions. If further corrections are needed, we revert and re-deign the features, architecture, or hyper-parameters as needed.

3 The Neural Network

3.1 An Introduction to Neural Networks

The task of algorithmically matching sound files to musical instrument classes in exceedingly difficult through conventional computer programming techniques. Because of the complexity of sound recognition, the challenge arises as to how to build some sort of program that could function at a level above general procedural or explicit instructional rules. Rather than *hard-coding* a set of conditions or parameters for a classification program, we seek an solution that allows for a computer to *learn* from labeled training examples in order *teach* itself. One such an algorithm that has been used for this type of task is a *Neural Network* [2, 3, 7, 21].

As the name implies, Early neural networks were inspired from from the human brain. Former YouTube video classification team lead, and current machine learning consultant, Aurelien Geron writes about the relationship between biological brains and mathematical neural networks [1]:

> Birds inspired us to fly, burdock plants inspired velcro and nature has inspired many other inventions. It seems only logical, then, to look to the brain's architecture for inspiration on how to build an intelligent machine.

The result of such an analogy is a computer program that is reminiscent of the structure of the a brain. Where a biological brain uses these chemical and electrical exchanges to process input and make decisions, the computational counterpart uses arithmetic and numerical exchanges to similarly process input and make decisions [1, 7].

A neural network can then be considered as mathematical function that seeks to produce an approximation F^*, of some usually unknown function F . For audio recognition, F is some function or series of functions that occurs in the brain which allows for a listener to process the sensation of audio, and map it to a known source object. For a neural network, function F^* uses a set of parameters Θ, to map a series of inputs, x (called *features*) to a series of outputs, y (called *predictions*) [3, 5, 21].

Over the course of their lives, humans will learn to map sounds to sources almost effortlessly [21]. Given an example, and the appropriate label, humans can do this with very reasonable accuracy over a wide array of sounds [13, 22]. We can simplify the idea of humans recognizing sound as some function or operation that occurs within the brain, accepting an input sound, and produces an output label. Similarly, a neural network can be constructed, presented with *features* of multiple, labeled sound waves and learns a set of parameters Θ that allows for the mapping to a source.

For this particular project, a neural network has been constructed to perform a *classification* task. A classification task involves mapping the input data, x to one of k possible *classes*, each represented by an integer, 0 through $k-1$. Each of the k classes represents a particular musical instrument that could have produced the sound-wave in the audio file [3, 9, 21].

3.2 The Structure of a Neural Network

A Neural Network is simply a model of a mathematical function, composed of several smaller mathematical functions called *layers* [3, 9]. Each layer represents an operation that takes some real input, typically an array of real double-precision floating-point numbers, and returns a modified array of new double-precision floating-point numbers. The exact nature of this operation can be very different depending on the layer type or where it sits within the network. It is this process of transforming inputs successively in a particular order until an output is attained that makes up the core functionality of a neural network [1, 9]. This output encodes the models final "decision" given a unique input.

Other than inspiration from the brain, Neural Networks are name *networks* because of their nested composition nature. The model can be represented by a linked computational graph which successively maps exactly how the repeated composition is structured. Each node in the graph represents a *layer*, which executes a particular prescribed mathematical function [3]. In the case of a *linear, feed-forward* network, information passes in a single direction, to produce an output. We can represent this as a graph:

$$F(x) = x \rightarrow f^{(0)}(x) \rightarrow f^{(1)}(x) \rightarrow ..... \rightarrow f^{(L-2)}(x) \rightarrow f^{(L-1)}(x) \rightarrow y \tag{2}$$

or as a nested function:

$$F(x) = f^{(L-1)}\big[f^{(L-2)}\big[...f^{(1)}\big[f^{(0)}\big[x\big]\big]\big]...\big]\big] \tag{3}$$

Each function $f^{(l)}$ represents a layer of the network model. The number of layers in a neural network is referred to as the network *depth*. The dimensionality of each layer is referred to as the network *width* [1, 9].

A network model that contains L unique layers is said to be an L-Layer Neural Network, with each layer usually indexed by a superscipt, 0 through $L-1$. Layer 0 is said to be the *input layer* and layer $L-1$ is said to be the *output layer*. The function that represents a layer (l) is given by

$$f^{(l)} : x \in \mathbb{R} \rightarrow y \in \mathbb{R} \tag{4}$$

The value of x can also be index by layer, we call the array $x^{(l)}$ the *activations* of layer l [3, 9].

The model is recursive by nature, the activations from one layer, $l-1$, are used to directly produce the activations of the next successive layer l. Thus Eq. (4) can be alternatively written as:

$$f^{(l)} : x^{(l-1)} \in \mathbb{R} \rightarrow x^{(l)} \in \mathbb{R} \tag{5}$$

The array of activations, $x^{(0)}$, is the raw input given the neural network, called *input* activations. Conversely, $x^{(L-1)}$ is referred to as *output* activations [1, 5, 9].

Algorithm 1 Forward propagation system in a standard deep neural network. Each iteration in the main *for-loop* represents the execution of a layer, and passing the result to the "next" layer function. We assume each layer to follow a two-step structure, being (i) a linear transformation as in Eq. (6) and (ii) an element-wise non-linear activation function as in Eq. (8). A practical application of this algorithm should include batches of samples instead of a single sample. I Want to include this algorithm for context - especially on the programming end, but am not sure where to put it.

Require: Network with L layers - $0, 1, 2, ..., L-2, L-1$
Require: Set of layer functions $f^{(l)}, l \in 0, 1, 2, ..., L-2, L-1$ made up of a linear transformation and a non-linear transformation.
Require: Input sample - $x^{(0)}$
If training the network, the intermediate layer activations must be stored for back-propagation. Otherwise, they may be discarded.
$Z = \{\} \leftarrow$ array to hold pre-activations
$X = \{\} \leftarrow$ array to hold final activations
for $l = 1, 2, ..., L-2, L-1$ **do**
Evaluate Linear Transformation
$z^{(l)} \leftarrow W^{(l)} x^{(l-1)} + b^{(l)}$
Apply non-linear Activation Function
$x^{(l)} \leftarrow \sigma^{(l)}\left[z^{(l)}\right]$
Store values for later use in optimizer, otherwise can be discarded.
$Z \leftarrow Z.add(z^{(l)})$
$X \leftarrow X.add(x^{(l)})$
end for
Network final prediction is last object in X array.
$y^* \leftarrow x^{(L-1)} = X[-1]$
Return the pre-nonlinear activations and final-nonlinear activations.
return Z, X

3.3 Layers Used in Classification Neural Network

As stated previously, a neural network is composed of a series of functions that are called in succession to transform features (an input) into a prediction (an output). As shown in Eq. (5), each function feeds directly into the next as to form a sort of recursive computational graph [3].

Typically, a layer function can be divided into two portions: (i.) a Linear transformation, with a bias addition, and (ii.) an element-wise activation transformation. This activation function typically is a non-linear function which allows for the modeling of increasingly complex decision-boundaries. The Linear transformation can be given by Eq.(6) in the case of a 1D Dense layer (3.3.1) or Eq. (7) in the case of a 2D Convolution layer.

$$z^{(l)} = W^{(l)}x^{(l-1)} + b^{(l)} \tag{6}$$

$$z^{(l)} = \left(W^{(l)} * x^{(l-1)}\right) + b^{(l)} \tag{7}$$

Where $W^{(l)}x^{(l-1)}$ denotes a matrix-vector product, and $\left(W^{(l)} * x^{(l-1)}\right)$ denotes a convolution product. In both cases, $W^{(l)}$ is the *weighting-matrix* or *weighting-kernel* for layer l, $b^{(l)}$ is the *bias-vector* for layer l, $z^{(l)}$ are the *linear-activations* for layer l and $x^{(l-1)}$ is the final activations for layer $l-1$.

Step (ii.) is usually given by some *activation function*:

$$x^{(l)} = \sigma^{(l)}\left[z^{(l)}\right] \tag{8}$$

Where $x^{(l)}$ is the final activations for layer l and $z^{(l)}$ is given in equation (6). $\sigma^{(l)}$ is some activation function, which is often use to enable the modeling of more complex-decision boundaries.

The combination of Eq. (6) and Eq. (8), for a layer l , make up the function represented by that layer, as in Eq. (5). Below, we discuss and describe the types of layer functions that are used to produce the classification model in this project.

3.3.1 Dense Layer

The Linear Dense Layer, often just called a *Dense* Layer, or *Fully-Connected* layer, was one of the earliest function types used in artificial neural network models [2, 9, 10]. They are among the most commonly used layer types. A dense layer is composed of a layer of *artifical neurons*, each of which holds a numerical value within it, called the *activation* of that neuron. This idea was developed from McCulloch and Pitts' work [10] in attempting to model cognitive processes as mathematical functions, and was expanded upon by Frank Rosenblatt in 1957 [1].

We model a layer of neurons as a vector of floating-point numbers. Typically, it is required that the array be one-dimensional. Suppose a layer (l) contains n artificial neurons. We denote the array that hold those activations as $x^{(l)}$ and is given by:

$$\vec{x}^{(l)} = \Big[x_0, x_1, x_2, \ldots, x_{n-2}, x_{n-1}\Big]^T \tag{9}$$

The activation of each entry is given by a linear-combination of activations from the previous layer, Eq.(6) and the transformation in Eq.(8).

Suppose the layer $l-1$ contains m neurons. Then the weighting-matrix, $W^{(l)}$ has shape $m \times n$, the bias-vector $b^{(l)}$ has shape $m \times 1$. We can denote the function in a similar manner to Eq.(5) as:

$$f^{(l)} : x^{(l-1)} \in \mathbb{R}^{(m \times 1)} \rightarrow x^{(l)} \in \mathbb{R}^{(n \times 1)} \tag{10}$$

Thus for a dense layer l, the exact values of each activation is given by [1, 9]:

$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}^{(l)} = \sigma^{(l)} \left(\begin{bmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,m-1} \\ w_{1,0} & w_{1,1} & \dots & w_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n-1,0} & w_{n-1,1} & \dots & w_{n-1,m-1} \end{bmatrix}^{(l)} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{m-1} \end{bmatrix}^{(l-1)} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}^{(l)} \right) \tag{11}$$

or more compactly:

$$x^{(l)} = \sigma^{(l)} \left(W^{(l)} x^{(l-1)} + b^{(l)} \right) \tag{12}$$

Eq. (12) is generally referred to as the *dense layer feed-forward equation* [3]. How Do I finish this section? Uses of Dens layer, physical meaning?

3.3.2 2-Dimensional Convolution Layer

Convolution layers emerged from studying the brain's visual cortex, and have been used from image recognition related tasks for around 40 years. [1, 9]). This layer get's it's name from it's implementation of the mathematical *convolution* function. The 2D discrete-convolution of function or 2D array $A[i,j]$ and $B[i,j]$ is given by [3]:

$$C[i,j] = (A * B)[i,j] = \sum_u \sum_v A[i,j] B[i-u, j-v] \tag{13}$$

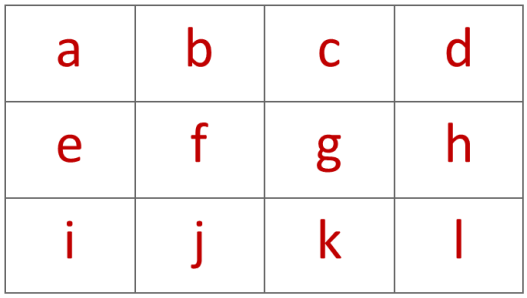
Note that convolution is commutative: $(A * B) = (B * A)$. We implement a convolutional layer $f^{(l)}$ by creating a series of K weighting matrices, called *filters*, each of size $m \times n$, where $m, n \in \mathbb{Z}$. Often we choose $m = n$ and call the shape the *convolution kernel size* [9, 3].

In the case of a 2D input array, $x^{(l-1)} \in \mathbb{R}^{(N \times M)}$, the convolutions filters "step" through the input data and repeatedly compute the element-wise product of each $m \times n$ weighting kernel, and the local activations of the $x^{(l-1)}$ array. The step size in each of the 2-dimension is known as the *stride*. Each of the K filters are used to generate a $m \times n$ feature map from the input activation. For an appropriately trained network, some filters may result in detecting vertical lines, horizontal lines, sharp edges, areas of mostly one color, etc. [1, 9].

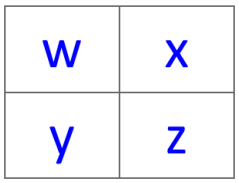
In general, for an $N \times M$ input, with kernel size $m \times n$, with stride size 1×1, and K feature maps.

$$f^{(l)} : x^{(l-1)} \in \mathbb{R}^{(N \times M)} \rightarrow x^{(l)} \in \mathbb{R}^{([N-n+1] \times [M-m+1] \times K)} \tag{14}$$

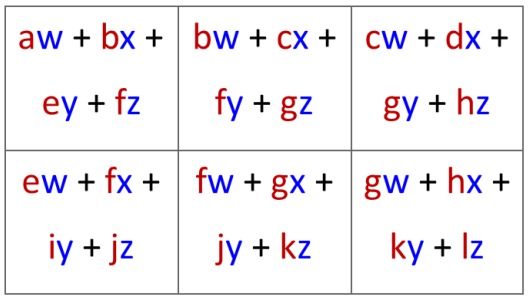
As an example, consider a single filter map over input $x \in \mathbb{R}^{(4 \times 3)}$ and filter map $W \in \mathbb{R}^{(2 \times 2)}$ as shown in Fig. (1):



(a) Input Activations



(b) Convolution Filter



(c) Convolution Result

Figure 1: The result of convolving an input (a) with an filter map (b) is a new set of activations (c). This Image was adapted from Goodfellow, pg. 325 [3]

Note that formal implementations include a bias vector and an activation function.

For 2D convolution over input $x^{(l-1)}$ with feature map $W_k^{(l)}$, producing activations $x^{(l)}$, we compute the activations $x_k^{(l)}$ as [3]

$$x_k^{(l)} = \sigma^{(l)}\left[\left(\sum_{u=0}^{m-1}\sum_{v=0}^{n-1} x^{(l-1)}(i,j)W_k^{(l)}(i-u,j-v)\right) + b_k^{(l)}\right] \tag{15}$$

Please check my math here! This operation repeats for each of the K feature maps. Each maps has it's own $n \times m$ weighting matrix and appropriately shaped bias .

The convolution layers allows for several advantages. Among these are (i) *sparse-connectivity*: not every single activation (pixel) is connect to every single output pixel, so it is more computationally efficient, (ii) *positional invariance*: key features can be identified regardless of where they are in the layer, and (iii.) *Automatic feature detection* as the training process will update the filters to identify dominant features in the data without human instruction [1, 3, 9].

3.3.3 2-Dimensional Maximum Pooling Layer

A Maximum Pooling layers simply returns the maximum neuron activation in a group of neurons. In the case of 2D Max Pooling, we choose a kernel size to be $m \times n$, just like in the 2D Convolution layer, and extract the maximum value in each window, and then step along accrdoing to the chosen stride size [9, 3].As an example, consider again figure (1a). Using the 2×2 kernel on the 3×4 input, each box would then contain the maximum value of the input activations. We detail this in Fig. (**??**):

Max(a,b,e,f)	Max(b,c,f,g)	Max(c,d,g,h)
Max(e,f,i,j)	Max(f,g,j,k)	Max(g,h,k,l)

Figure 2: The result of 2D maximum-pooling an input array. This image was adapted from Loy, pg. 126 [9]

Pooling layers, such as maximum pooling, average pooling, or similar are typically placed after a layer or a set of layers of convolution. The purpose of this arrangement is to reduce the number of weights, thereby dropping the complexity of the model and ensuring only features with large activation values are preserved [1, 9, 3].

3.3.4 1-Dimensional Flattening Layer

A flattening layer is used to compress an array with two or more dimensions down into a single dimension. For a flattening layer l, multidimensional activations in layer $l-1$ are rearranged down into a single dimensional array. Note that this is not like projecting the data into a lower dimension, but rather is simple the reorganization of values into an array of a single axis. We can use function notation to express this as:

$$f^{(l)} : x^{(l-1)} \in \mathbb{R}^{(M \times N \times ....)} \rightarrow x^{(l)} \in \mathbb{R}^{(MN... \times 1)} \tag{16}$$

The numerical value of each activation is left unchanged. For a layer with activation shape$N \times M$, the resulting activations are reshaped into $NM \times 1$ as shown in Eq.(16).

Flattening Layer are most commonly used to prepare activations for entry into dense layer or series of dense layers. For example, 2D or 3D images are typically processed with 2D convolution, which may output a 2D or 3D array of activations. Those values are then flattened to 1 dimensions, which can then be passed into dense layers for further processing.

3.3.5 1-Dimensional Concatenation Layer

The 1-Dimensional Concatenation Layer, also called 1D-Concat layer takes separate vectors of activations and combines them into a single 1D vector. Consider the layer activations $\vec{a}^{(l-1)}$ and $\vec{b}^{(l-1)}$ with shapes $1 \times \alpha$ and $1 \times \beta$ respectively. They are the outputs of two different layers:

$$\vec{a}^{(l-1)} = \big[a_0, a_1, ..., a_{\alpha-1}\big] \qquad \text{and} \qquad \vec{b}^{(l-1)} = \big[b_0, b_1, ..., b_{\beta-1}\big] \tag{17}$$

The result of the concatenation is a new 1D-array, $\vec{c}^{(l)}$ with size $1 \times \alpha + \beta$:

$$\vec{c}^{(l)} = \big[a_0, a_1, ..., a_{\alpha-1}, b_0, b_1, ..., b_{\beta-1}\big] \tag{18}$$

We can denote this for n arrays with function notation:

$$f^{(l)} : x_a^{(l-1)} \in \mathbb{R}^{(1\times\alpha)}, x_b^{(l-1)} \in \mathbb{R}^{(1\times\beta)}, ... \rightarrow x_z^{(l)} \in \mathbb{R}^{(1\times\alpha+\beta+...)} \tag{19}$$

The concatenation layer is used to combine activation non-adjacent, or non-related layers into a single new layer. In this case of the model used in the project, we use it to combine the activations that result from two distinct models into a single layer that is then transformed into a final output. This process is detailed further in sec. (3.6).

3.4 Activation Functions Used in Network Layers

Activation functions are a key parameter in the behavior of neural networks. As discussed in sec. (**??**) a layer function, $f^{(l)}$ is generally composed of a linear transformation, as in Eq. (6), and then an element-wise activation function as in Eq. (8). It is this second step that allows for neural networks to model the incredibly complex decision boundaries that are found in real-world problems [1, 9]. In this section, we detail the activation functions used in out classification neural network.

3.4.1 Rectified Linear Unit

The Rectified Linear Unit (ReLU) activation function acts element-wise on the activations in a given layer. If the activation of a neurons is non-zero or non-negative, the value is untouched, otherwise a 0 is returned. For an input activation x, ReLU is defined by:

$$\text{ReLU}\big[x\big] = \max\big(0, x\big) = \begin{cases} x & \text{if} \quad x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{20}$$

ReLU is only defined on a real domain. We provide a visualization of the function in Fig. (3).

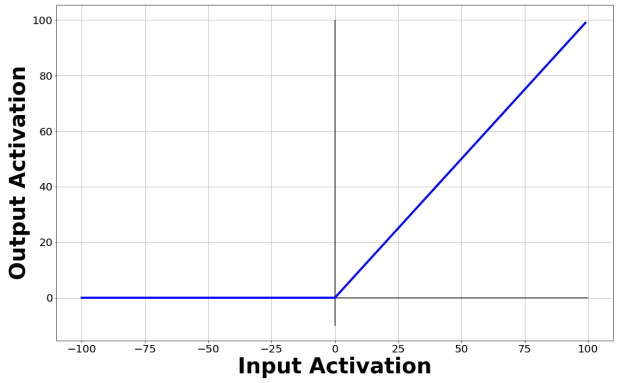


Figure 3: Rectifed Linear Unit (ReLU) activation function

For our chosen architecture, ReLU is applied to the activations in every single Convolution layer and Dense Layer, with the exception of the output dense layer.

3.4.2 Softmax

The softmax activation function is commonly used in the output layer of a multi-class classification network, such as the one we implement. When softmax acts on a vector or activation, the result a non-negative output vector. with an L_1 norm of 1 [1, 3, 21]. The i-th element in a softmax activation function is given by:

$$\text{Softmax}\big[x\big]_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{21}$$

This function is almost only used in the output layer of a neural network, and encodes a discrete categorical probability. For example, if Eq.(21) returns a vector of the form: $[0.75, 0.25]$, we interpret this as a sample having a 75% chance of beloinging to class 0 and a 25% change of belonging to class 1.

3.5 Training a Neural Network

A neural network's purpose is to produce a function F^* that approximates an unknown function F, using a set of parameters, Θ. The model must have a procedure for updating the parameters in Θ to allow for a reasonably close approximation of F. To better understand this, we turn to Tom Mitchell's explanation of a learning algorithm [3, 12]:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if it's performance at tasks in T, as measured by P, improves with experience E.

Without any direct human intervention, a model must update itself to improve it's performance at a give task a new information is presented to it. To do this, the model must be constructed with a training procedure in mind.

We consider the set of parameters Θ as a concatenation of each applicable layer's weighting matrix and bias vector such that:

$$\Theta = \left\{ W^{(0)}, b^{(0)}, W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, ..., W^{(L-2)}, b^{(L-2)}, W^{(L)}, b^{(L-1)}, \right\} \tag{22}$$

Each parameter is floating-point number, which is stored in a weight matrix $W^{(l)}$ or bias vector $b^{(l)}$. Each value for each parameter contributes to the output of the function, thus indirectly influences the final layer activations. Note that $W^{(l)}$ or $b^{(l)}$, themselves are not parameters, but arrays of parameters to be trained. Certain layers, such as pooling layers (**??**), concatenation layers (3.3.5), or flattening layers (3.3.4) do not contain any trainable parameters. Similarly, activation functions themselves are static and are not included in the list of parameters.

In increasingly complicated neural networks, there can be upwards of hundreds of thousands,or even millions of elements in Θ making a neural network a functions that exist is a very high dimensional parameter-space [1, 3, 7]. For the network that we have designed in this project, there are roughly $33,000$ parameters across 20 layers.

3.5.1 The Cost Function

Suppose we pass a training sample, given by the feature-vector $x^{(0)}$, with an expected outcome given by the one-hot-encoded vector y, into the neural network. After the network finishes processing, it's output activations are given by the vector $x^{(L-1)}$, also noted as y^*. For a reasonably trained model, we expect the vector y^* to be "similar" to y, indicating that with the provided data, the network has made a valid prediction. Conversely, for an untrained network, y^* is not likely to be "similar" to y at all, indicating that the network has made a poor prediction.

To quantify the difference between the model's prediction, y^* and the expected output, y, we introduce a *cost function*, $J(y^*, y)$, to the model [3, 5]. The cost function, also called a *loss* or *objective* function, compares the neural networks output, y^* and the expected output y. It returns a single floating-point number which measures the *poorness* of the prediction. A high cost value indicates a *poor* prediction, and a low cost indicates a reasonable prediction.

$J(y^*, y)$ can take many forms and is often dependent on particular task or data set provided [5]. For this k-classes classification task, we choose to use the *Categorical Crossentropy* (CXE) cost function. For a sample labeled by the one-hot-encoded vector y and corresponding prediction vector y^*, the CXE cost for a single sample is given by [3, 21]:

$$\mathrm{CXE}\big[y, y^*\big] = -\sum_{i=0}^{k-1} y_i \log_{10}(y_i^*) \tag{23}$$

Thus, the average loss over a mini-batch of b samples is given:

$$\big\langle \mathrm{CXE}\big[y, y^*\big] \big\rangle = -\frac{1}{b} \sum_{n=0}^{b-1} \sum_{i=0}^{k-1} y_i^{(n)} \log_{10}(y_i^{*(n)}) \tag{24}$$

Suppose that a given sample belongs to class j in a k-classes classifier. Since the label vector, y is one-hot-encoded, all entries are zero except $y_j = 1$.

$$y = \Big[y_0, y_1, ..., y_j, ..., y_{k-1}\Big]^T = \Big[0, 0, ..., 1, ..., 0\Big]^T \tag{25}$$

Thus the sum in Eq. (23) contains mostly zero terms. The only non-zero term is the produce of y_j and $\log_{10}(y_j^*)$. Since the vector y^* has been subject to the softmax activation function, we must have $y_j^* \in [0, 1]$. This means that taking the log of this value returns a negative number. Multiplying by negative number returns a high loss when the activation of y_j^* is low and a low cost when $y_j^* \approx 1$. We visualize this relationship in Fig. (4):

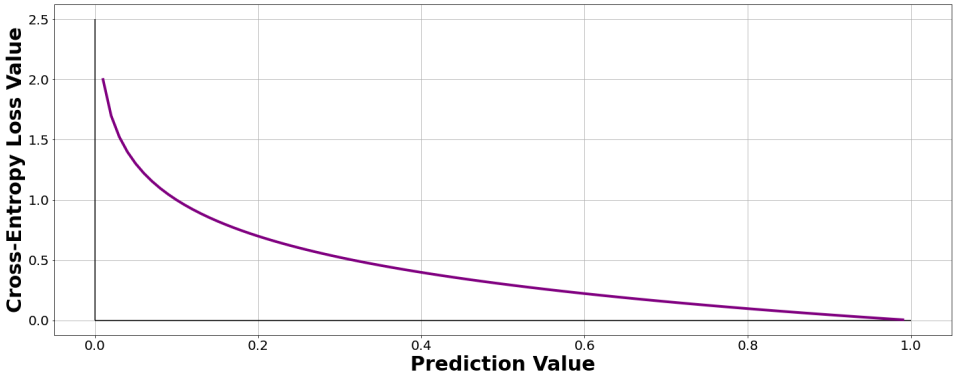


Figure 4: Plot of how y_j^* affects the output values of the CXE cost function

By optimizing the parameters Θ in the network model, we allow for the output of a consistently low cost function to be produced across all samples in the data set.

3.5.2 Gradient Based Learning

We have developed a method for quantifying the difference between a given output y^* and an expected output y with the inclusion of a cost function. For a trained neural network, we expect that samples consistently produce a low cost value, particularly on previous unseen inputs, which indicates that the model has appropriately *generalized* [5, 9]. The process of training the model is to choose the parameters in Θ that allows for this behavior of the cost function. We then treat the training process of a neural network as a higher dimensional *optimization* problem [3].

A neural network uses *indirect optimization*, which contrasts from pure optimization. Deep Learning expert Ian Goodfellow described this [3]:

> In most learning scenarios, we care about some performance measure P,... We reduce a different cost function, $J(\theta)$ in the hope that doing so will also improve P.

By choosing an appropriate cost function, and selecting the parameters Θ that minimize the average cost over a data set, we also assume that doing so minimizes classification error and maximizes the performance P.

The cost of a sample is dependent on training labels y and the network output y^*. The output is given by the final layer activation $x^{(L-1)}$, which in turn are produced by the previous layer and so forth, as demonstrated in Eq. (5), This recursive nature combined with the dimensionality of the parameter object Θ makes an analytical solution to the optimization impractical [1, 3, 5]. We instead optimize the cost function with a gradient-based method.

We can reduce the cost given a set of parameters, $J(\Theta)$, by moving each element in Θ with small steps in the direction of the negative partial derivative of each parameter. In the case of the high-dimensional Θ object, we compute the partial derivative with respect to each weight and bias in the form of the *gradient vector*. We denote the gradient of J with respect to the parameters in Θ as:

$$\nabla_{\Theta}\big[J\big] = \Big[\frac{\partial J}{\partial W^{(0)}}, \frac{\partial J}{\partial b^{(0)}}, \frac{\partial J}{\partial W^{(1)}}, \frac{\partial J}{\partial b^{(1)}}, \ldots, \frac{\partial J}{\partial W^{(L-1)}}, \frac{\partial J}{\partial b^{(L-1)}}\Big]^T \tag{26}$$

Where $\frac{\partial J}{\partial W^{(l)}}$ is taking the partial derivative of each element in the $W^{(l)}$ matrix, and preserves the shape.

Due to the nested composition of the network output in Eq. (3), and subsequently the cost itself, we must use the chain rule of calculus to work backwards through computational graph of the neural network to compute the partial derivative with respect to each parameter in Θ. The process of working backwards to compute each element in the gradient vector is called *back-propagation* [1]. Below we detail an algorithm in pseudo-code to back-propagate through a fully-connected neural network, such as the multilayer perceptron outlines in sec.(**??**)

Algorithm 2 Backwards propagation system, in a standard densely connected deep neural network. Each iteration in the *for-loop* computes the gradient of the cost function J with respect to the weight and bias arrays. Each element in those arrays is then the discrete gradient of that parameter. A practical application of this algorithm should include batches of samples instead of a single sample

Require: Cost/Objective function J and learning rate α
Require: Set of weighting parameters - $W^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of bias parameters - $b^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of layer activation function - $\sigma^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of layer activation function derivatives - $\partial\sigma^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of pre-nonlinear activation - $Z^{(i)}, i \in \{0, 1, ..., L-1\}$
Require: Set of post-nonlinear activation - $X^{(i)}, i \in \{0, 1, ..., L-1\}$
  Execute forward pass in algorithm (4) and compute the gradient of the cost with respect to the final layer activations
  $dx \leftarrow \nabla_{(y^*)} J(y, y^*)$
  Initialize ∇J as output object, should have same shape as Θ
  for $L-1, L-2, ..., 2, 1$ **do**
    Compute gradient w.r.t pre-nonlinear activation portion of layer function
    $dx^{(l)} \leftarrow \nabla_{Z^{(l)}} J = dx^{(l)} \odot \partial\sigma^{(l)}[Z^{(l)}]$
    Compute gradient w.r.t weighting and bias elements
    $db \leftarrow \nabla_{b^{(l)}} J = dx^{(l)}$
    $dW \leftarrow \nabla_{W^{(l)}} J = dx^{(l)} \cdot X^{(l-1)}$
    Add db and dW steps to ∇J object
    $\nabla J = \nabla J.Add(dW, db)$
  end for
  Return gradient w.r.t to each parameter in Θ
  return ∇J

After (i) computing the gradient, we can scale it by a desired learning rate, α and (ii) add the gradient vector element-wise to the existing elements in Θ object. By repeating steps (i) and (ii) in succession, we gradually drive the cost function to produces consistently lower and lower values across a data set [1]. This is called *gradient descent* and is the basis for many optimization algorithms. We show the general update rule on iteration (i) for gradient based learning over a batch of m samples in Eq. (27).

$$\Theta^{(i)} = \Theta^{(i-1)} + (-\alpha)\frac{1}{m}\nabla_{\Theta}\Big[\sum_{n=1}^{m} J(y^{*(n)}, y^{(n)})\Big] \tag{27}$$

3.5.3 The Optimizer

Standard gradient-based learning, as given by the parameter update rule in Eq. (27) is cumbersome and often prone to numerical error. This can include phenomena such as *vanishing gradients*, *exploding gradients*, *poor conditioning*, or unstable solution-spaces. [1,

3, 9]. To combat this, we require a more stable and robust optimizer than the bare gradient descent method. Many gradient-based optimizers have been developed over the years to mitigate some f these vulnerabilities.

For this project, we employ an *Adaptive-Moments* optimization algorithm, also called *ADAM* for short. It is a powerful optimizer with uses adaptive learning rate and momentum parameter. ADAM tracks an exponentially decaying average of past gradients, as well as exponentially decaying average of past squared gradients [1]. This produces a far more aggressive optimizer at a higher computational cost.

This means that as the cost function iterates through each step, the effect will tend to "snowball". If the previous step was found to reduce the cost function by a large or small amount, the step size for the previous step will update accordingly. Note that the upper index (i) or $(i-1)$F gives an iteration index, while the upper iteration i means to raise a value to the power of the iteration. For a given step (i), The ADAM update is given:

$$\begin{aligned}
s^{(i)} &= \rho_1 s^{(i-1)} + (1-\rho_1)\frac{1}{m}\nabla_\Theta\Big[\sum_{n=1}^{m} J(y^{*(n)}, y^{(n)})\Big] \\
r^{(i)} &= \rho_2 r^{(i-1)} + (1-\rho_2)\frac{1}{m}\nabla_\Theta\Big[\sum_{n=1}^{m} J(y^{*(n)}, y^{(n)})\Big] \odot \frac{1}{m}\nabla_\Theta\Big[\sum_{n=1}^{m} J(y^{*(n)}, y^{(n)})\Big] \\
s'^{(i)} &= \frac{s^{(i)}}{1-\rho_1^i} \\
r'^{(i)} &= \frac{r^{(i)}}{1-\rho_2^i} \\
\Theta^{(i)} &= \Theta^{(i-1)} + (-\alpha)\frac{s'^{(i)}}{\sqrt{r'^{(i)}} + \delta}
\end{aligned} \tag{28}$$

ADAM has experimentally shown to be a very powerful and robust optimizer useful for a wide range of tasks. Because of the two decay constants ρ_1 and ρ_2, we an compound and accumulate the values of past gradients to continue to push the cost to lower and lower values, even if the magnitude of the gradient becomes very small [1].

Algorithm 3 Adaptive-Moments (ADAM) optimizer for a neural network

Require: Step size α
Require: Small constant δ for numerical stabilization, usually about 10^{-7}.
Require: constants ρ_1, ρ_2 used got exponential decay rates, usually 0.9 and 0.999 respectively.
Require: Subroutine/function to compute gradient of cost function.
Require: Mini-batch size, m
Require: Stopping criterion S
Initialize moment variables and iteration counter $s = 0$, $r = 0$, $i = 0$
while Stopping Criterion S is **false do**
Extract a mini-batch of m samples from larger data set X. $[x^{(0)}, x^{(1)}, ..., x^{(m-1)}]$ and corresponding target values $[y^{(0)}, y^{(1)}, ..., y^{(m-1)}]$.
Compute numerical gradient estimate of each sample in batch. This can be done with standard back-propagation in algorithm (2) and normalize by batch size m:
$\nabla J \leftarrow \frac{1}{m}\nabla_\Theta\Big[\sum_{n=1}^{m} J(y^{*(n)}, y^{(n)})\Big]$
Compute first bias moment: $s \leftarrow \rho_1 s + (1-\rho_1)\nabla J$
Compute second bias moment: $r \leftarrow \rho_2 s + (1-\rho_2)\nabla J \odot \nabla J$
First bias correction: $s' \leftarrow \frac{s}{1-\rho_1^i}$
Second bias correction: $r' \leftarrow \frac{r}{1-\rho_2^i}$
Compute And Apply update: $\Delta\Theta = (-\alpha)\frac{s'}{\sqrt{r'}+\delta}$
$\Theta = \Theta + \Delta\Theta$
Update Iteration number: $i \leftarrow i + 1$
end while

3.6 Chosen Model Architecture

The success of a neural network algorithm is enormously dependent of the strength of the chosen features, as well as a complementary architecture, rather than the quantity of available data [1, 6, 8]. We have derived an appropriate set of inputs arrays, explore in sec. (5) and constructed a neural network architecture to coincide with these features. From each sample, we produce two input arrays, one being a matrix and the other being a column-vector, and present those to a *multimodal neural network* [21]. This type of hybrid network accepts one array at each of two input layers, passes them through a series of non-overlapping layers, and combines separate neural networks into a single model to produce a one prediction resulting from both inputs citation for multimodal networks.

The full classification neural network used for this project consists of two distinct entry points. Rather than presenting the network with one set of input data, X, we present the network with two different arrays, X_1 and X_2. Both arrays are a product of the same audio file sound wave, and thus share a common training label, y. Each of the two entry layers leads to it's own branch and the given information is process roughly in parallel. Once a certain number of layer functions have been applied, each arm of the neural network is left

with a single dense layer of activations. These two activation vectors are concatenated into a single new dense layer, and then passed through one final layer to generate the model's final output. A visual representation of this architecture can be found in Fig. (**??**).

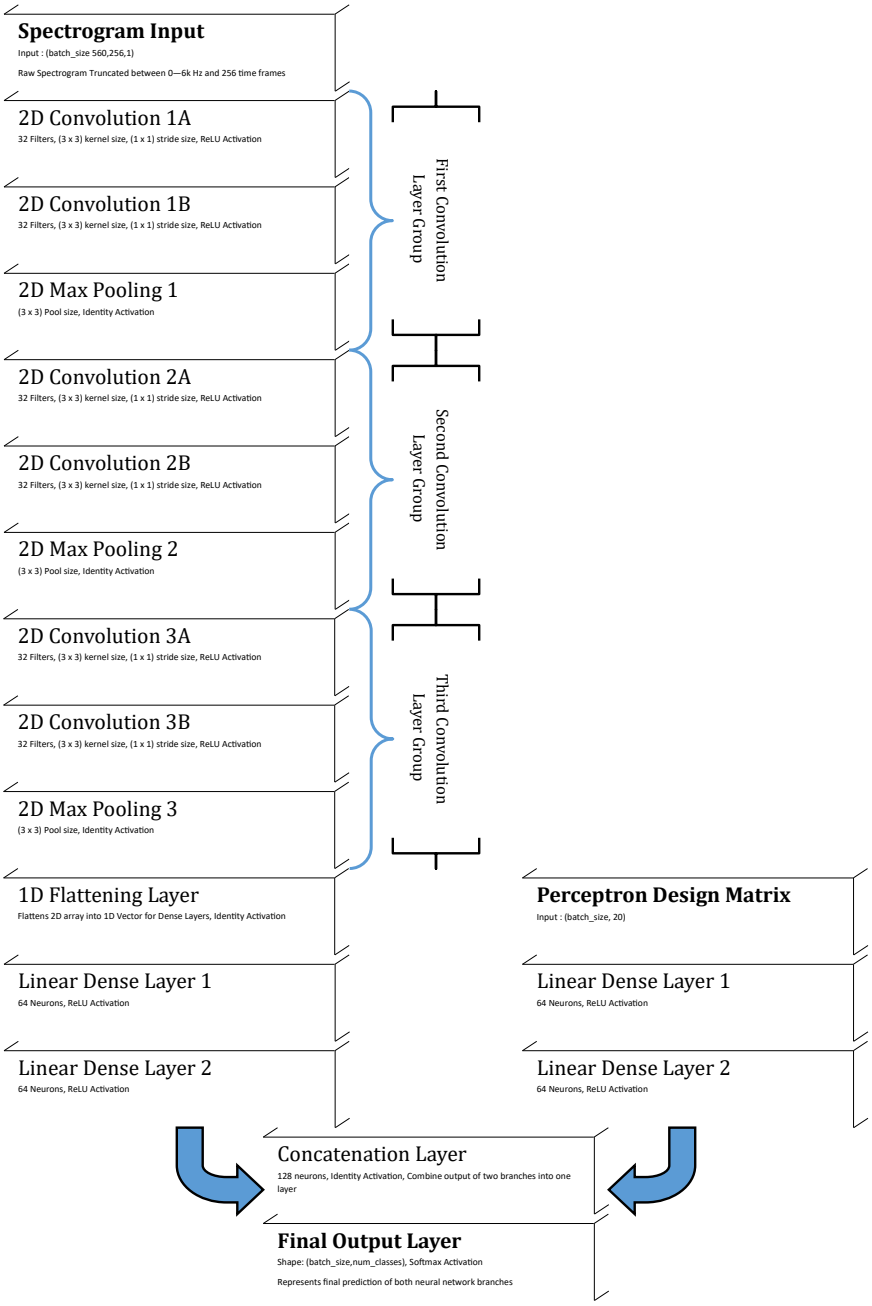


Figure 5: The developed architecture of the audio file classification neural network. The Left branch process an image-like input, the right branch processes a vector-like input. The activations are then merged, and then a single output is produced

3.6.1 The Spectrogram Branch

The spectrogram branch is pictured on the left side of Fig. (**??**). A spectrogram is a representation of a sound wave, or signal by representing energy distributions as a function of *time* and *frequency* [22, 13, 6]. Details on the creating of the spectrogram can be found in the *features* section of this document.

The input layer of the spectrogram accepts a 4-Dimensional array. The axes, in order of indexing, represents (i) the size of the *mini-batch* of samples, (ii) the pixel width of each sample, (iii) the pixel height of each sample, and (iv) the number of channels in each sample. As a model hyper-parameter, we have chosen each batch to contain 64 samples. We have also chosen to truncate time and frequency axes (ii & iii) to contain 560 frequency bins, and 256 time-frames. Each image also contains a single channel, which make it gray-scale when visualized. We can denote the 4D shape of the input object into this branch as:

$$X_1 \in \mathbb{R}^{(64\times560\times256\times1)} \tag{29}$$

Any other shape will be rejected by the model, and an error is raised.

After the input layer assert the appropriate shape, the array X_1, which is a collection of 64 spectrograms, is passed into the first of three *Convolution Layer Groups*. These layer groups are inspired from the *VGG-16 Neural Network* architecture [citation needed]. Each convolution layer group in composed of three individual layers:

1. A 2-Dimensional Convolution layer, using 32 filters, a 3×3 kernel, a 1×1 step size, and a ReLU activation function,
2. A 2-Dimensional Convolution layer, using 32 filters, a 3×3 kernel, a 1×1 step size, and a ReLU activation function,
3. A 2-Dimensional Maximum Pooling layer using a 3×3 pooling size, and an Identity activation function

The Convolution layers convoluted over the middle two axes of the data (over space and time in the spectrogram).

By grouping layers in this structure, we use the convolution layers to reduce the number of features, and then the pooling layer to extract only the largest activations of the remianing features. This ensures that in the full spectrogram image, only the dominant features and feature shapes are preserved and passed into the next layer set for processing.

3.6.2 The Perceptron Branch

The Perceptron branch is picture on the right side of Fig. (**??**), notice that it is considerably smaller in side and complexity than it's neighbor. Rather than accept an image-like input,

the perception simply takes a vector-like input of properties that are derived from the audio file. We call these properties *features* [1, 6, 17]. Details on the creating of these features can be found in the *features* section of this document.

The input layer of the perceptron accepts a 2-Dimensional array. The axes, in order of indexing, represent (i) the size of the *mini-batch* of samples, (ii) the number of features for each sample. We use the same model hyper-parameter of 64 samples per batch, and have developed 20 unique classification features that have been derived from time-space and frequency-space representations of the audio file data. We can denote the 2D shape if the input object into this branch as:

$$X_2 \in \mathbb{R}^{(64\times 20)} \tag{30}$$

This 2D is referred to as a *design matrix* [5, 9]. Any other shape will be rejected by the model, and an error is raised.

In perceptron models, scaling the design matrix appropriately drastically improves the classification performance. A design matrix is scaled by taking all samples in a column (a particular feature from each class), subtracting the average, and then scaling it such that it has unit variance [1, 5]. This ensures that no one feature dominates the performance of the model and that layer activations to not get saturated as data progresses through the network.

3.6.3 The Final Output Branch

The last layer in each of the two branches is a ReLU-activated Dense layer containing 64 neurons, represented by a 64×1 column vectors. We combine these model layers by using a *concatenation layer* to fuse the two column vectors together, "vertically", such that the result is a 128×1 column vector. This vector is then passed into the final dense layer, which used the softmax activation, and encodes the joint predictions based on both model branches.

4 Properties of Musical Instruments

As discussed in Sec. (3), performance quality of a neural network is greatly dependent on the properties of the chosen features [21, 8]. To ensure that a the classifier model performs adequately and consistently, we must choose these features to have high variance between classes, and low variance within each class. This enables a neural network to development clear decision boundaries between each unique class [5, 17]. To develop this set of features, we explore the physical and mechanical properties of musical instruments and the sounds that they create.

We have assembled training data samples than can be grouping in 38 classes of unique sources listed below:

1. Alto Flute
2. Alto Saxophone
3. Banjo
4. Bass
5. Bass Clairnet
6. Bass Flute
7. Bassoon
8. Bass Tromobe
9. B♭ Clarinet
10. Bells
11. Cello
12. Clarinet Check this!
13. Contrabassoon
14. Crotale
15. E♭ Clarinet
16. English Horn
17. Flute
18. Guitar
19. Hi-Hat
20. Horn
21. Mandolin
22. Marimba
23. Oboe
24. Sawtooth Wave
25. Saxophone (Tenor)
26. Sine Wave
27. Soprano Saxophone
28. Square Wave
29. Tenor Trombone
30. Triangle Wave
31. Trombone
32. Trumpet (B♭)
33. Tuba
34. Vibraphone
35. Violin
36. Viola
37. White Noise
38. Xylophone

39. 40.

Rather than explore each class individually, we can examine the properties of groupings of musical instrumentsby using the *Hornbostel-Sachs* system *red*Citation needed!. In this syetem, musical instruments or sound sources can be divided up into four broad categories based on the nature of the sound-producing material. These categories are (i) idiophones, (ii) membranophones, (iii) chordophones, and (iv) aerophones. In this section, we explore the mechanical properties of each group and show how the physics of the source instrument influences our feature choice.

4.1 Idiophones

An idiophone is an instrument that produces sound through the vibration of the full body of the object. This generally includes most percussive instruments excluding drums citation needed. From the set of classes that we use, bells, crotales, Hi-hats, vibraphones, and xylophones are all examples of idiophones.

4.2 Membranophones

4.3 Chordophones

A chordophone is a musical instrument that produces sound through the vibration of strings that are stretched between two fixed points citation needed. From our data set, banjos, basses, cellos, guitars, mandolins, violins, and violas are all examples of chordophones.

4.3.1 The 1-Dimension Vibrating String

We can model a chordophone as having a string of length L, with a constant linear mass density, μ, subject to a tension force F_T. The string is flexible, and free to vibrate in a single dimension. We describe the state of string over space and time with a function $U(x,t)$. The behavior of the string can be modeled by D'Alembert's 1-dimensional wave equation [4, 19]:

$$\frac{\partial^2 U}{\partial t^2} = v^2 \frac{\partial^2 U}{\partial x^2} \tag{31}$$

Where v is the wave propagation velocity, given by $\sqrt{\frac{F_T}{\mu}}$. Additionally, the string is fixed in place at both ends, which can be modeled by the boundary conditions:

$$U(0,t) = 0 \qquad \text{and} \qquad U(L,t) = 0 \tag{32}$$

In order to specify a solution, we must also specificy two intitial conditions. We described the initial state of the string at time $t = 0$ by the two conditions:

$$U(x,0) = f(x) \qquad \text{and} \qquad \frac{\partial U}{\partial t}(x,0) = g(x) \tag{33}$$

Depending on the exact nature of the source instrument, $f(x)$ and $g(x)$ can take on different forms. For example, for plucked stringed instruments, $f(x)$ may appear to be as triangle-like function, with a peak at some position $0 < x_0 < L$ where then string was struck. $g(x)$

We explore solutions that take the form [4]:

$$U(x,t) = \psi(x)h(t) \tag{34}$$

From this separation, we find that $\psi(x)$ and $h(t)$ each are composed of a linear combination of orthogonal trigonometric functions. Applying the boundary and initial conditions to the system allows for a linear combination of solutions [4]:

$$U(x,t) = \sum_{n=1}^{\infty} \left(A_n \sin\left(\frac{n\pi x}{L}\right) \cos\left(\frac{n\pi v t}{L}\right) + B_n \sin\left(\frac{n\pi x}{L}\right) \sin\left(\frac{n\pi v t}{L}\right) \right) \tag{35}$$

The initial conditions are satisfied if:

$$\begin{array}{l} f(x) = \sum_{n=1}^{\infty} A_n \sin\left(\frac{n\pi x}{L}\right) \\ g(x) = \sum_{n=1}^{\infty} B_n \left(\frac{n\pi v}{L}\right) \sin\left(\frac{n\pi x}{L}\right) \end{array} \tag{36}$$

Where A_n and B_n are given by:

$$\begin{array}{l} A_n = \frac{2}{L} \int_0^L f(x) \sin\left(\frac{n\pi x}{L}\right) dx \\ B_n = \frac{2}{n\pi v} \int_0^L g(x) \sin\left(\frac{n\pi x}{L}\right) dx \end{array} \tag{37}$$

4.3.2 Consequences of the Wave Equation

The solution that we have derived from the wave equation shows that the behavior of $U(x,t)$ is greatly dependent on the nature of the string's initial shape, $f(x)$, when struck. In the case of picked or plucked instruments, the initial shape of $f(x)$ would be something like a "ramp function":

$$f(x) = \tag{38}$$

4.4 Aerophones

4.5 Other Generated Sounds

We group sounds produced from non-physical musical instruments in this final category. Form our data set, this category includes the four simplest waveforms, (i) sine waves, (ii) sawtooth waves, (iii) square waves, and (iv) triangle waves, as well as any color noises [22]. Since these sounds are only produced synthetically through a MATLAB program, we choose not to include them in the Hornbostel-Sachs grouping above. We explore the properties of each of these generated sounds.

4.5.1 Sine Wave

A sine wave is the simplest of sound waves, and makes up the foundation for all periodic wave forms [22, 21]. It consists of a single oscillatory term with a fixed fundamental frequency f_0. A sine wave's amplitude as a function of time is given by

$$x(t) = A_0 \sin(2\pi f_0 t + \phi) \tag{39}$$

Where A_0 gives an amplitude, f_0 is the fundamental frequency, and ϕ is a phase shift. We show a simple sine wave in the time and frequency domain in Fig. (6).

[Image of Sine Wave Waveform] [Image of Sine Wave Freq. Spect]

Figure 6: A sine wave in the (a) time-domian and (b) frequency domain

A pure sine wave contains no overtones, only a time-independent fundamental frequency, which can take on any value due to synthetic generation.

4.5.2 Sawtooth Wave

4.5.3 Square Wave

A square wave is a piece-wise defined waveform whose value alternatively alternates between $+1$ and -1 [22, 13]. A square wave with period f_0 has a waveform defined over a single period as:

$$x(t) = \begin{cases} +1 & 0 < x \leq T_0/2 \\ -1 & T_0/2 < x < T_0 \end{cases} \tag{40}$$

where $T_0 \equiv 1/f_0$.

We can use Fourier decomposition to examine the constituent sinusoidal wave that make up a square wave. The frequency space representation of a square wave indicates that is can be approximated as a sum of higher odd-numbered harmonics, with the amplitude of each decreasing by the reciprocal of the harmonic number [22]. Thus a square wave in the time domain can be approximated as:

$$x(t) \approx \sum_{n=1}^{\infty} \frac{1}{2n-1} \sin\Big(2\pi(2n-1)f_0 t\Big) \tag{41}$$

4.5.4 Triangle Wave

4.5.5 White Noise

[Image of Square Wave Waveform] [Image of Sqaure Wave Freq. Spect]

Figure 7: A square wave in the (a) time-domain and (b) frequency domain

5 Feature Selections

Classification tasks require a set of inputs that are used to make the decision as to which class the sample belongs to. These inputs are called *predictors* or *features* [5, 9, 17]. Features are quantitative, low-dimensional representations of each sample that convey key characteristics of the sample. Audio machine learning researchers M. Kashif Saeed Khan and Wasfi G. Al-Khatib discusses the vitality of feature selection [**?**]:

> The data reduction stage which is also called feature extraction, consists of discovering a few important facts about each class. The choice of features is critical as it greatly affects the accuracy of audio classification.

In the case of the biological process, the features of the audio is simply the raw time series waveform as processed by the brain. This information is then used into categorize the source. In the case of the neural network model, the time series waveform alone has shown to be unreliable in classification tasks [17]. We must then develop a set of features or predictors that can describe important properties of the waveform in a far more efficient manner [3, 5].

It becomes quickly apparent that the choice of features is extraordinarily important to the process of training and evaluation of the classifier model. Consider the task of identifying cats and dogs from images, given only images consisting of the top-most row of pixels. The inappropriate choice features does not reflect the significant characteristics of the data.

Tuomas Virtanen, machine learning and audio engineer writes in his book, "Computational Analysis of Sound Scene and Events" about specifically how features must be chosen [21]:

> For recognition algorithms, the necessary property of the acoustic features is low variability among features extracted from examples assigned to the same class, and at the same time high variability allowing distinction between features extracted from examples assigned to different classes.

To ensure the construction of a suitable model, we derive features based from three representations of the audio file: (i) a spectrogram matrix of the waveform, (ii) the time-space representation of the waveform, and (iii) the frequency-space representation of the waveform. It is important to note that although this algorithm will classify sound data to instruments, the model will never actually be presented with a waveform directly, instead it will rely solely on the features.

To ensure suitable performance of this sound wave classification neural network, a great deal of time has been devoted to the construction of the elements of the feature vector. These features are derived are principles of music, digital signal processing, previous work, and underlying physics that governs the behavior of instruments. In the following sections, we outline the set of 21 features used in the classification process. For each feature, we visualize the distribution in across each class in the form as box-plots. Each feature has been normalize to have a zero mean, and unit-variance acorss all classes.

5.0.1 The Design Matrix

The spectrogram and 20 features from each sample are organized into one of two array structures, each called a *design matrix*. A design matrix contains real floating-point numbers that stores the features in a convenient way for the neural network to digest. For this hybrid neural-network, we use two matrices, X_1 and X_2, each containing data from b audio file samples. each has shape given by:

$$X_1 \in \mathbb{R}^{(b \times N' \times k \times 1)} \tag{42}$$

$$X_2 \in \mathbb{R}^{(b \times p)} \tag{43}$$

Where $N' \times k \times 1$ gives the shape of a single spectrogram matrix S, described in sect. (5.1). Similarly, p is the number of features derived from the time and frequency-representation of features outlined in sec. (5.2) and sec.(5.3).The first axis, with size b indicates that there are b audio samples stored in that design matrix.

5.0.2 Audio Preprocessing

Preprocessing a data set is a necessary step to execute prior to feature extraction [2, 5]. In the case of audio files, preprocessing usually consists of ensuring that the data set contains the following:

1. A suitably sized number of files, of reasonable audio quality, with normalized amplitudes
2. Audio encoded in a standard, and consistent format
3. A consistent sample rate and bit depth between audio files
4. A consistent number of channels

Note that different projects may require a different set of requirement from preprocessing [21]. For this project, we have chosen to use the following parameters:

1. Roughly $16,000$ audio files Professionally or semi-professionally recorded in a studio. Citation needed!. All ampltiudes have been normalized to ± 1 unit.
2. All audio has be converted into *.WAV* files from other formats, such as *.AIF* or *.MP3* using a MATLAB program
3. All audio is sampled at $44,100$ Hz and Bit depth needed - 16? 24?
4. All audio has been down-mixed into mono-channel waveforms.

5.1 Spectrogram Features

The field of neural classification is well studied in the application of image-processing. Many large-scale, and even introductory projects find themselves under the umbrella of image classification [1, 3, 9, 11]. As a result, model architectures for image processing related tasks are well-explored and have shown experimentally successful behavior. Following this success, it make senses to provide an image-like representation of a sound wave as a feature. We do this in the form of a spectrogram matrix.

A spectrogram is a representation of the energy distribution of a sound wave as a function of both frequency and time. In a conventional spectrogram, the passing of time is shown along the x-axis, and the frequency spectrum is shown on the y-axis. Thus each point in the 2-Dimensional space is an energy at a given time and frequency. Examples spectrograms from the wave form data set are shown in Fig. (**??**).

Insert spectrograms here

Figure 8: Spectrogram representations of various waveforms

A spectrogram is produced by the method of *frame-blocking*, which is very prevalent in audio signal classification [8, 23]. Frame-blocking takes a raw waveform or signal, s and decomposes it into a set of analysis frames, a_i, with each being N samples in length, and has a fixed overlap with the next adjacent frame. Each of the k frames then allows for a section of the signal in quazi-stationary state [6, 17].

For this project, we have chosen to use frames of size $N = 4096$ with a 75% or 3072 sample overlap. Since each audio file contains a different number of samples, we choose the number of frames, k to be less than or equal to 256. If $k > 256$, the waveform in truncated, if $k \leq 256$ the frames left as is. The audio has been sampled at $f_s = 44100$ samples/second, so each frame represents a slice of time that is about 0.1 seconds long.

We concatenate each analysis frame, $a_i, i \in [0, k-1]$ into a single $k \times N$ matrix, called A. Each row is a frame, each column is an index in that frame

$$A = \{a_0, a_1, a_2, ..., a_{k-1}\} = \begin{bmatrix} a_0[0] & a_0[1] & a_0[2] & \dots & a_0[N-1] \\ a_1[0] & a_1[1] & a_1[2] & \dots & a_1[N-1] \\ a_2[0] & a_2[1] & a_2[2] & \dots & a_2[N-1] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{k-1}[0] & a_{k-1}[1] & a_{k-1}[2] & \dots & a_{k-1}[N-1] \end{bmatrix} \tag{44}$$

We use bracket notation, $a[j]$ to indicate that each analysis frame a_i is array-like. The following indexing conventions for matrix A all represent the same entry:

$$A_{i,j} = A_i[j] = A[i][j] = A[i,j] \tag{45}$$

After frame-blocking, we apply a *windowing function* to each frame. A standard *Hann Window* of N samples is generated as a $1 \times N$ row-array, H. The n-th index in a Hann window with N samples is defined:

$$H[n] = \frac{1}{2}\left[1 - \cos\left(\frac{2\pi n}{N-1}\right)\right] \tag{46}$$

This window function is applied to each analysis frame by computing the element-wise product of the Hann window array, H and each row of the analysis frames matrix, $A_{i,:}$. Result is an $k \times N$ array, $\widetilde{A}$:

$$\widetilde{A_i} = A_i \odot H \tag{47}$$

The window function allows for a cleaner transform into frequency space [21]. Expand on this. HOW? Why do we use a windowing function?

Finally, we perform a *Discrete Fourier Transform* (DFT) to bring each analysis frame from a time domain into a frequency domain [**?**, 14]. The Discrete Fourier Transform is applied by producing an $N \times N$ *transform matrix*, often noted at $\mathbb{W}$. Let $\omega^k = e^{\frac{-2\pi i}{N}k}$, then the DFT matrix for a time-space containing N samples

$$\mathbb{W} = \frac{1}{\sqrt{N}}\begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \dots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \dots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \dots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \dots & \omega^{(N-1)^2} \end{bmatrix} \tag{48}$$

Each column of the matrix is a complex sinusoidal oscillating with an integer number of periods within the N-sample length window [18, 14].The DFT is applied by a taking the matrix - product of $\mathbb{W}$ and $\widetilde{A}^T$. The transpose of $\widetilde{A}$ then makes each analysis frame into a column vector, which gives the appropriate dimension for multiplication.

$$\text{DFT}\left[\widetilde{A}\right] = \mathbb{W}\widetilde{A}^T \tag{49}$$

Most standard implementations of neural network models require all activations, weights, and biases to be real floating-point numbers. Since the the DFT matrix introduces complex values, we compute the square of the element-wise $L2$-norm of the resultant array.

$$S_{i,j} = \left\|(\mathbb{W}\widetilde{A}^T)_{i,j}\right\|_2^2 \tag{50}$$

Where $\mathbb{W}$ is the DFT matrix from Eq. (48) and $\widetilde{A}^T$ is the transpose of the analysis frames matrix from Eq. (47). The matrix S, is the spectrogram representation of the initial waveform, and has shape $N \times k$ of real floating-point numbers. We can index matrix S similarly to that of matrix A in Eq. (45):

$$S_{i,j} = S[i][j] = S[i,j] \tag{51}$$

Each column of the S matrix is now a single-frame that has been moved into a frequency-space representation, thus there are k columns, just as there were k time-series analysis frames. Given the discretized nature of digital audio, the frequency-space representation is not a continuous function, but rather a column vector, where the frequency has been assigned to one of N bins, ranging from $-f_s/2$ to $+f_s/2$. To ensure homogeneous input sizes between all samples, we zero-pad the matrix S with additional columns until $k = 256$. Recall that wave forms were truncated to ensure that $k \leq 256$ analysis frames.

Standard western musical instruments seldom extend above 6 kHz [**?**, 21, 22]. This means that when constructing the spectrogram, we will rarely ever see energy present above this frequency at any time, and the S matrix will contain mostly zero, or zero-like entries. To condense the size of the matrix, we select onlt the frequency bins that correspond to energies between 0 Hz and 6000 Hz. This makes the input array smaller, and eliminates redundant and non-useful information. The number rows in the S matrix is reduced from N down to N'.

Each spectrogram is now $N' \times k \times 1$ (The last axis represents a single pixel channel, as opposed to 3 channels for RGB inputs, etc.) and effectively encodes the energy distribution of the waveform as a function of both time and frequency. The spectrogram is the first feature used in this model. For this classifier, we have chosen $N' = 558$ and $k = 256$. For training, a batch of b samples are concatenated into a single array object. For a batch of b samples of $N' \times k \times 1$ spectrograms, we shape X_1 such that:

$$X_1 = \left\{ S^{(0)}, S^{(1)}, S^{(2)}, ..., S^{(b-1)} \right\} \in \mathbb{R}^{(b \times N' \times k \times 1)} \tag{52}$$

Which is consistent with the shape of the X_1 matrix outlined in eqn(42). This matrix is presented to the *Convolution* branch of the neural network for processing.

5.2 Time-Space Features

The features described in this section are derived from time-domain representations of each audio sample.For consistency between smaples, each waveform is padded or truncated to contain exactly M samples. Time space is indexed by $s[i]$ with $i \in [0, 1, 2, 3, ..., M-2, M-1]$

From time space, we use the following 7 features:

- Time Domain Envelope
- Zero Crossing Rate
- Temporal Center of Mass
- Auto Correlation Coefficients ($\times 4$)

5.2.1 Time Domain Envelope

The time domain envelope (TDE) is a method of determining which parts of the sound wave contains most of the energy of the signal Often, we compute the RMS-Energy of each frame - a larger energy of the frame, the larger the average amplitude of the waveform in that frame. For frames with smaller amplitude, it is likely that the signal has either not begun, or is decaying.

We adapt this feature to **compute the RMS-Energy of the full waveform**. The RMS-Energy of the full waveform, s is given by [13, 17]

$$\text{RMS}\big[s\big] = \sqrt{\frac{1}{M}\sum_{i=0}^{M-1} s[i]^2} \tag{53}$$

By using the full waveform, we acquire a rough estimate for the energy in the entirety of the audio file [8]. For instruments with long sustain or release times, such as strings or undampened mallet percussion, we expect a comparably large waveform RMS when compared to those instruments without sustain such as plucked stings or dampened percussion Citation?

PLOT OF TDE FEATURES HERE

Figure 9: Time domain envelope visualized in feature-space

5.2.2 Zero Crossing Rate

The zero crossing rate (ZXR) of a signal or frame is use to measure how many times that a signal crosses it's equilibrium point. This can be done per total sound wave, per unit time, or per analysis-frame. This feature is most commonly associated with differentiating speech from music, because speech presents a more jagged and often less periodic waveform. In some cases, the ZXR can be correlated to frequency as well [6, 23].

We adapt this feature to **compute the zero crossing rate for the full waveform**. The ZXR for the full waveform s is given by [17, 8]

$$\text{ZXR}\big[s\big] = \frac{1}{2}\sum_{i=1}^{M-1}\Big|\,\text{sign}(s[i]) - \text{sign}(s[i-1])\Big| \tag{54}$$

Where sign(x) returns $+1$ if $x > 0$, -1 if $x < 0$ and 0 if $x = 0$. This provides a rough estimate for the average frequency in the full waveform,n and can help discern clean periodic signals (low ZXR) from those that may have more noise or volatile behavior (high ZXR)[17].

PLOT OF ZXR FEATURES HERE

Figure 10: Zero crossing rate visualized in feature-space

5.2.3 Temporal Center of Mass

The temporal center of mass (TCM) of a signal is used to compute roughly where in time the amplitude of the waveform *bunches up*. We treat the full waveform array, s, with M samples as a discrete 1-Dimensional mass distribution. The TCM of that waveform is then given:

$$\text{TCM}\big[s\big] = \frac{\sum_{i=0}^{M-1} is[i]}{\sum_{i=0}^{M-1} s[i]} \tag{55}$$

This includes negative amplitude values acting as "negative masses". Should I use the absolute value of the signal instead?

The TCM turns out to be a very unique feature, and very powerful in classification due to it's variance textcolorredCitation needed. For instruments with heavier attacks, we expect the TCM to be a very near the start of the waveform. For instruments with long sustain times, we expect a very centrally located center of mass, and for instruments with long release times, we expect a later TCM value.

PLOT OF TCM FEATURES HERE

Figure 11: Temporal center-of-mass visualized in feature-space

5.2.4 Auto Correlation Coefficients

Auto correlation coefficients (ACC) are rough estimates of the signal spectral distribution . We can compute any number of ACC's and their value changed depending on the index chosen. It is common to only compute the first K ACC's [17]. For a full waveform signal s, with M samples, the k-th ACC (indexed from 1 to K) is given by:

$$\text{ACC}_k\big[s\big] = \frac{\sum_{i=0}^{M-k-1} s[i]s[i+k]}{\sqrt{\sum_{i=0}^{M-k-1} s^2[i]}\sqrt{\sum_{i=0}^{M-k-1} s^2[i+k]}} \tag{56}$$

An ACC can be thought of as convolving a signal with a time-delayed version of itself. This measures similarity between samples related to the time difference between them Citation needed. I may need to try a different set of ACC's for this

PLOT OF ACC FEATURES HERE

Figure 12: First four auto correlation coefficients visualized in feature-space

5.3 Frequency-Space Features

The features described in this section are derived from the frequency-domain representations of each audio sample. Frequency space is represented by the transposed spectrogram matrix, S^T, described in sec. (5.1). This is done to ensure that each row is now an analysis frame, each column is a frequency bin. This gives a similar structure to the time-space analysis frames matrix, A, in Eq. (44). For each feature, we detail the physical significance and provide a visualization in feature-space.

From frequency space, we use the following 13 features:

- Mel Frequency Cepstral Coefficients ($\times 12$)
- Frequency Center of Mass

5.3.1 Mel Filter Bank Energies

Mel filter banks are divisions of the frequency spectrum of a signal into R overlapping triangular bins [16]. These filter banks allows us to group sounds based on their energy distribution in frequency space. Each filter is triangularly shaped, over a certain band in frequency space, and zero elsewhere. This way, when computing the the dot product of any filter with frequency space, we get an approximation of energy in that filter bank [16, 17].

Rather than producing filter banks based on the linear Hertz scale, the frequency axis of the signal is transformed into units of *Mels*, which is used to account for the non-linearity in human pitch perception [17, 6]. Filter banks are produced to be evenly spaced on the Mel scale, and then transformed back into the Hertz scale. This has the effect of producing triangular filter banks with grow in width as the frequency increases. The Hertz to Mel and Mel to Hertz transforms are given [17, 6]:

$$M_f[h] = 2595 \log_{10}\left(1 + \frac{h}{700}\right) \tag{57}$$

$$H_f[m] = 700\left(10^{\left(\frac{m}{2595}\right)} - 1\right) \tag{58}$$

Where M_f is the frequency in units of Mels, given $[h]$, a frequency in Hertz, and H_f is the frequency in Hertz given $[m]$ a frequency in Mels.

Mel filters are produced by grouping frequency-space into R overlapping bins called *Mel Filter-banks*. In units of Mels, each filter bank, R_i remains a constant width, but when shown in Hertz, the bins increase logarithmically.

Figure of a few Mel Filter Banks goes here

Figure 13: Mel Filter Banks shown in frequency space with units of Hertz

Each of the R filters is created to be N' samples long, to match the width of the cropped frequency space in the spectrogram, and transformed back into units of Hertz. When applied to an analysis frame in the frequency spectrum, the dot-product between the filter and the spectrum gives an approximation of the energy in that filter bank. Each filter is concatenated into a matrix M of shape $R \times N'$, where each row is a filter. We apply the Mel Filter banks to the spectrogram to create matrix B:

$$B = S^T M^T \tag{59}$$

Matrix B has shape $k \times R$.

Using S^T ensures that each row of the matrix is an analysis frame in frequency space. Similarly, each column of the matrix M is a appropriately scaled Mel filter-bank. This way, the matrix product in Eq. (59) allows that $B_{i,j}$ is the dot product between the i-th analysis frame and the j-th filter-bank. Finally, we compute the average energy across all k frames, into array $\widetilde{B}$ with shape $1 \times R$. For this project, we have chosen to use $R = 12$ filter-banks, which are all appended to the feature vector.

PLOT OF MFCC FEATURES HERE

Figure 14: Mel Frequency Cepstral Coefficients in feature-space

5.3.2 Mel Frequency Cepstral Coeffecients

Ceptstral coefficients are the result of computing the inverse discrete Fourier transform (IDFT) of the logarithm of the frequency Spectrum. Mel Frequency Cepstral Coefficients (MFCC's) are the most commonly used in digital signal processing. They are computed throught he inverse discrete cosine transform of the log energy in in each Mel frequency bank [17, 16]. The c-th coefficient is given by:

$$\text{MFCC}[c] = \sqrt{\frac{2}{R}} \sum_{i=1}^{R} \log\left(\widetilde{B[i]}\right) \cos\left(\frac{c(i-\frac{1}{2})\pi}{R[i]}\right) \tag{60}$$

Where $\widetilde{B}$ are the column average of the Mel filter bank energies computed in Eq. (59), and R is the number of filter banks used.

Physically, MFCC's are a transform of a transform. This allows us to investigate the periodicity of the frequency spectrum, which highlights phenomena such as overtones or echoes [21]. Ceptrum coefficients are commonly used to speech identification and are very prolific in sound recognition tasks [17, 16, 8]. We provide a representation of MFCC properties below.

PLOT OF MFCC FEATURES HERE

Figure 15: Mel Frequency Cepstral Coefficients visualized in feature-space

5.3.3 Frequency Center of Mass

The frequency center-of-mass (FCM) for an audio file provides a representation of how overtones and energy is distributed in the signal's frequency domain. As with the temporal center-of-mass, we treat each row of the S^T matrix as it's own 1-Dimensional mass distribution, and compute the center of mass of that row. This encodes the FCM for a single analysis frame (in frequency-space) in the waveform. For an frequency-analysis frame, $s^{(i)}$, the FCM is given by:

$$\text{FCM}_i\left[s^{(i)}\right] = \frac{\sum_{j=0}^{N'-1} j s^{(i)}[j]}{\sum_{j=0}^{N'-1} s^{(i)}[j]} \tag{61}$$

We compute the FCM for each of the k' frames, and then average the results. We use the average FCM across k' frames to compute the FCM feature:

$$FCM = \sum_{i=0}^{k'} \frac{1}{k'} \text{FCM}_i\left[s^{(i)}\right] \tag{62}$$

The average FCM gives a strong approximation of the instrument or signal source's range. For example, a flute or violin will have a considerably high FCM value, even in their lower registers. Similarly, basses or tubas will have considerably low FCM values. Given that the standard frequency range of some musical instruments is fixed, it is guaranteed that for any particular instrument, the FCM will consistently remain within certain bounds [**?**, 22].

PLOT OF FCM FEATURES HERE

Figure 16: Frequency center-of-mass visualized in feature-space

5.4 Assembling the Design Matrices

I would like to use some figures here to graphically represent how the feature vectors and spectrograms are concatenated into the design matrices. I may help with the geometry of the situation

6 Evaluating Performance

Before making predictions on unlabeled data such as the Chaotic Synthesizers, we must confirm that our model performs reasonably well on data that it has never interacted with. The most common practice is to divide a full data set into a subset of *training* samples, and *testing* samples. As the names imply, the training subset is used to fit the model, and we use the labeled testing data set to evaluate. The exact ratio of sample volume between these subsets varies depending on the task [3, 2, 12]. Since the testing data is labeled, we can compare the classifiers predictions to the *ground truth* labels.

6.1 Cross Validation

Performance evaluations are commonly implemented in the form of *K-Fold Cross-Validation* (Also called X-val) [1, 3]. This process involves taking a data set containing N unique samples and dividing it into K non-overlapping subsets. 1 subset is reserved to evaluate the model, and $K-1$ remaining subsets are used to train the model. This belongs a larger family of statistical validations called *Resampling methods* [5]. Below, we detail pseudo-code for a K-Fold Cross Validation algorithm.

Algorithm 4 A K-Fold Cross Validation program.

Require: Untrained Network or related learning algorithm, F^*
Require: A full labeled data set of N samples. $X^{(i)}$, $i \in 0, 1, 2, ..., N-1$
Require: Number of splits in Cross validation, K
Require: Performance metric function(s), P
  Divide Data into K non-overlapping subsets x_i, each with roughly N/K samples
  $X \rightarrow \left\{x_0, x_1, x_2, ..., x_{K-2}, x_{K-1}\right\}$
  Performance History $\leftarrow \{\}$
  for $j = 0, 1, 2, 3, ..., K-2, K-1$ **do**
    Reset all parameters in F^* to a random "untrained" state
    Set aside testing data subset
    $X_{test} \leftarrow x_j$
    Concatenate the remaining subsets into training data set
    $X_{train} =\leftarrow x_{i \neq j}$
    Train the model, F^* with the X_{train} data set
    Evaluate the trained model with the X_{train} dataset, and compute value of metric function(s) P
    Store Performance P in Performance History array
  end for
  Compare performance results, and adjust model, parameters as needed, and repeat if desired.

The output of cross validation is K models that are all trained and evaluated on overlapping subsets of the full data. Each model F^*, shows one possible outcome of training the network.

It has a corresponding set of learned parameters Θ^*, as defined in Eq. (22) that it uses to make any predictions.

6.2 Performance Metrics

In the case of the multi-category classifier, it is important that we choose the appropriate performance metrics to confirm that the network is completeing it's assigned as as expected [1]. While the neural network itself uses the cost function as it's sole objective to optimize, we also require a set of more human-readable functions. For example, a loss score of 1.075 over a given subset of previously unseen samples provides us with no real information as to how well the network is performing at it's designed task. In this section, we introduce a set of functions and metrics than enable a more tangible interpretation of the model's performance. To evaluate any performance metric, we require a set of samples with ground truth labels, y, and a model's prediction for those labels, y^* [3, 5].

6.2.1 Confusion Matrix

The confusion matrix (also called a confusion table) is a very quick, often graphical model that can be used to show how a classifier model performs over a subset of predictions. The general idea of this object is count the number of times class i is predicted to be in class j, and vice-versa [1]. If we see that these classes are being repeatedly *confused* in the model's prediction process, then we can modify the model or features to account for it.

For a k-classes classifier, a confusion matrix will have shape $k \times k$, and every element is a non-negative integer. Each row represents the "ground truth" or labeled class, and each column represents a predicted class. Thus for a confusion matrix, C, we can say that:

$$C_{i,j} = \text{Number of samples that}$$
$$\text{belong to class } i\text{, and were predicted to be in class } j$$

Thus, indexes where $i = j$ represents a correct prediction, and $i \neq k$ indicates an incorrect prediction. A confusion matrix with relatively large values in main diagonal indicates a model that consistently predicts correct labels [1].

Images of Confusion Matrices Go Here! Images of Confusion Matrices Go Here!
Images of Confusion Matrices Go Here! Images of Confusion Matrices Go Here!

Figure 17: Example Confusion Matrices for k-classes tasks

The confusion matrix can be a cumbersome, so often it is useful to express the performance in terms of more concise quantities. For this we use *accuracy score*, *precision score*, and *recall score*.

6.2.2 Precision Score

Precision score (also called *specificity* or *positive predictive value*) is the ratio of chosen elements to all relevant elements. This bounds precision to the range $(0, 1)$, with a higher value more desirable. For a classifier with $k = 2$ unique classes, we define the precision score of a model as:

$$\text{Prec} = \frac{TP}{TP + FP} \tag{63}$$

Where TP is the number of *true-positive*, and FP is the number of false-positive predictions. For a k-classes confusion matrix, C, the precision score of a class j, is given by the entry $C_{j,j}$ divided by the sum of row j:

$$\text{Prec}_j = \frac{C_{j,j}}{\sum_{i=0}^{k-1} C_{j,i}} \tag{64}$$

For multi-class problems, we report the precision score as averaged over all k classes.

The quantity $TP + FP$ is a measurement of the total number chosen items in that class. Thus, the quantity precision gives the ratio of correct selections to all selections that were made. More intuitively, we can describe precision as the fraction of selected items that are relevant to the problem at hand [1].

6.2.3 Recall Score

Recall score (also called *sensitivity* or *true positive rate*) also offers a more concise performance metric than a confusion metric. For a classifier with $k = 2$ unique classes, we define the recall score of a model as:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{65}$$

Where TP is the number of *true-positive*, and FN is the number of *false-negative* predictions. For a k-classes confusion matrix, C, the precision score of a class j is given by the entry $C_{j,j}$ divided by the sum of column j:

$$\text{Recall}_j = \frac{C_{j,j}}{\sum_{i=0}^{k-1} C_{i,j}} \tag{66}$$

For multi-class problems, we report the precision score as averaged over all k classes.

The quantity $TP + FN$ is a measurement of the total number items in that class, independent of selection. Thus, the quantity recall gives the ratio of correct selections to all selectes that were made. More intuitively, we can describe recall as the fraction of relevant items that were selected [1].

6.2.4 Accuracy Score

Accuracy score, while not commonly used is the most intuitive of all of the performance metrics. It is the ratio of

6.3 Tracking Metrics over a Period of Training

A period of training is characterized by fitting the parameters Θ of a model F^* to a set of data X and corresponding labels Y [3, 21]. This is done by passing subsets of data, called *mini-batches*, into the neural network for training. The average cost for the mini-batch is then used to compute the gradient vector $\nabla_\Theta J$, and subsequently update the model according to the optimizer chosen [1, 3].

We execute computation in batches to reduce the amount of memory required for the operation. Pushing a full data set through a model at one time would require more RAM than is usually available , thus repeated subset of data tend to avoid memory exhaustion errors. Additionally, every mini-batch used equates to one step in the optimizer update rule [3]. Thus, a single pass over a full data set allows for multiple iterations of the optimizer to reduce the value of the cost function. This can be combined with multiple passes over the full data set, called *epochs* [5, 9].

Large mini-batches require lots of RAM, but prevent the model from being over-fit to to any one sample, or class of samples. Small mini-batches require less RAM, but often may bias the optimizer to over-fit the given samples, and make optimization unstable [1, 5]. To ensure that the model is optimizing properly, we can track how the metrics behave over a training period. This allows us to monitor the *rate of convergence* of the model. If the cost function is dropping to quickly or slowly, this may indicate that the optimizer learning rate is too high or too low. This may lead to an over-fit or under-fit model, or indicate an inappropriate set of feature is being used [1, 3]. In Some cases, it can be used to initiate *early-stopping*, which halts training when a set of parameters are met.

Given the high dimensionality of this model's parameter space,the large volume of sound files, and the large amount of RAM required to store the spectrogram matrix and the feature vector for each sample, we do not directly load the full data set into memory at once. Instead, we produce a large subset of the full data set of 256 samples which we dub a *mega-batch*. From this mega-batch, we produce the two deign matrices required for input, and load the corresponding labels. We then use a subset of this group as the *mini-batches* for training. Each mini-batch contains 32 samples. Once all mini-batches are fit, we discard the design matrices in the mega batch ans repeat for the next 256 samples.

Along each training step, we record the precision, recall, and loss scores. Our program stores these values locally in a *training-history* file, which we can examine after the program completes. Below, we visualize the value of each metric score at each training epoch.

Loss Score Goes Here

Figure 18: The loss function score decreases with each training step, indicating that optimization is performing correctly

similarly, we can visualize the precision score and recall score at each training step.

Precision Score Goes Here Recall Score Goes Here

Figure 19: The precision score (a) and recall score (b) increases with each training step

6.4 Boosted Aggregation

Many learning models such as neural networks, or decision trees suffer from *high variance* [5]. This means that randomly splitting a full dataset into halves and fitting two different model to each half has the potential to produce wildly different fit parameters. In the case of neural networks, high variance can be amplified by the nature of the randomization in multidimensional optimizer [1]. To mitigate this effect, we can employ a variant of the *bagging* algorithm. Bagging uses an average, or weighted average, of parameters in multiple models that have been trained on overlapping data sets.

Consider if we have K models, all trained on overlapping training/testing data sets:

$$\text{Models} = \left\{ F_0^*, F_1^*, F_2^*, ..., F_{K-2}^*, F_{K-1}^* \right\} \tag{67}$$

Each model contains a corresponding set of parameters, Θ^i, $i \in [0, K-1]$. We could then aggregate the parameters according to a weighted average to create a new set of paramaters, Θ' which can be used to construct a new model, F'.

This algorithm is often used after K-Folds Cross validation, as outlined in in subsec. (6.1). This grants the K models and corresponding parameters that could be used in the bagging technique [5]. This is very not finished yet!

7 Experimental Results

8 Conclusion

9 Acknowledgments

The completion of this project would not have been possible without the various contributions from the following people:

- Dakota Buell, John Parker, Nathan Richard, Morgan Saidel for consultation on physics, mathematics, and programming topics.
- Madeline Edwards for differing me to this project, as well as constant support and consultation on mathematical topics.
- Dr. Kevin Short and Dr. Maurik Holtrop for various consultation and guidance throughout this project
- University of Iowa, Electronic Music Studio and Philharmonica Symphony Orchestra for the digital sound library used for training data samples.

References

[1] Geron, Aurelien. *Hands-on Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly, 2017.

[2] Geron, Aurelien. *Hands-on Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 2nd ed., O'Reilly, 2019.

[3] Goodfellow, Ian, et al.*Deep Learning*. MIT Press, 2017.

[4] "Wave Equation: Vibrating Strings and Membranes." *Applied Partial Differential Equations with Fourier Series and Boundary Value Problems*, by Richard Haberman, Pearson, 2019, pp. 130–150.

[5] James, Gareth, et al. *An Introduction to Statistical Learning with Applications in R*. Springer, 2017.

[6] Khan, M. Kashif Saeed, and Wasfi G. Al-Khatib. "Machine-Learning Based Classification of Speech and Music." Multimedia Systems, vol. 12, no. 1, 2006, pp. 55–67., doi:10.1007/s00530-006-0034-0.

[7] Levine, Daniel S. *Introduction to Neural and Cognitive Modeling*. 3rd ed., Routledge, 2019.

[8] Liu, Zhu, et al. "Audio Feature Extraction and Analysis for Scene Segmentation and Classification." Journal of VLSI Signal Processing, vol. 20, 1998, pp. 61–79.

[9] Loy, James , *Neural Network Projects with Python*. Packt Publishing, 2019

[10] McCulloch, Warren S., and Walter Pitts. "A Logical Calculus of the Ideas Immanent in Nervous Activity." *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, 1943, pp. 115–133.

[11] Mierswa, Ingo, and Katharina Morik. "Automatic Feature Extraction for Classifying Audio Data." *Machine Learning*, vol. 58, no. 2-3, 2005, pp. 127–149., doi:10.1007/s10994-005-5824-7.

[12] Mitchell, Tom Michael. Machine Learning. 1st ed., McGraw-Hill, 1997.

[13] Olson, Harry E. *Music, Physics and Engineering*. 2nd ed., Dover Publications, 1967.

[14] Peatross, Justin, and Michael Ware. *Physics of Light and Optics*. Brigham Young University, Department of Physics, 2015.

[15] Petrik, Marek. "Introduction to Deep Learning." Machine Learning. 20 April. 2020, Durham, New Hampshire.

[16] Sahidullah, Goutam S. "Design, Analysis and Experimental Evaluation of Block Based Transformation in MFCC Computation for Speaker Recognition." 18 Nov. 2011.

[17] Serizel, Roman, et al. "Acoustic Features for Environmental Sound Analysis." Computational Analysis of Sound Scenes and Events, by Tuomas Virtanen, Springer, 2018, pp. 71–101.

[18] Short, K. and Garcia R.A. 2006. "Signal Analysis Using the Complex Spectral Phase Evolution (CSPE) Method." AES: *Audio Engineering Society Convention Paper.*

[19] "Continuum Mechanics." *Classical Mechanics*, by John Robert Taylor, University Science Books, 2005, pp. 681–738.

[20] TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[21] Virtanen, Tuomas, et al. *Computational Analysis of Sound Scenes and Events.* Springer, 2018.

[22] White, Harvey Elliott, and Donald H. White. *Physics and Music: the Science of Musical Sound.* Dover Publications, Inc., 2019.

[23] Zhang, Tong, and C.-C. Jay Kuo. "Content-Based Classification and Retrieval of Audio." *Advanced Signal Processing Algorithms, Architectures, and Implementations VIII*, 2 Oct. 1998, pp. 432–443., doi:10.1117/12.325703.